A Chip MultiProcessor Accelerator for Video Decoding

Cor Meenderinck Ben Juurlink
Computer Engineering Lab
Faculty of Electrical Engineering Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
{Cor,Benj}@ce.et.tudelft.nl

***Abstract*—In this paper we propose architectural enhancements to specialize the Cell SPU for video decoding. Through thorough analysis of the H.264 video decoding kernels we identify the execution bottlenecks among which are matrix transposition, scalar operations, and lack of saturating arithmetic. Based on these bottlenecks we propose ISA extensions that speed up the execution. The speedup achieved on the IDCT8, IDCT4, and deblocking filter kernel are between 1.69 and 2.01.**

I. Introduction

We have entered the era of Chip MultiProcessors (CMPs) and at time of writing they are already being deployed in many market segments. It is generally expected that the number of cores will grow at a steady rate. Initially this growing TLP will deliver the increase in performance as predicted by Moore's Law.

However, the impact of the power wall is also increasing over time [1]. That means that the power budget will be the main limitation of performance and not technology or the transistor budget. Clock frequencies will have to be throttled down and eventually it will not be possible to use all available cores concurrently. In such a situation performance increase can only be obtained by improving the power efficiency of the system which will lead us to specialization of cores (often referred to as domain specific accelerators).

An important application domain of today as well as of the future is media. Therefore, in this paper we investigate the specialization of an existing core for media applications. Specifically we choose H.264 video decoding as the target application. H.264 is the best video coding standard in terms of compression ratio and picture quality currently available and is a good representative of future media workloads. As the baseline core for specialization we take the Cell SPE. The Cell broadband engine has eight SPEs that operate as multimedia accelerators for the PowerPC (PPE) core. Therefore it is an excellent starting point for specialization.

This paper is organized as follows. In Section II we describe the architecture of the SPE core. Section III describes the experimental setup used to obtain the results presented in this paper. We continue in Section IV by describing the proposed ISA extensions that specialize the SPE core for video decoding. The results of the performance evaluation of the proposed accelerator are presented in Section V. Finally, Section VI concludes the paper.

II. The SPE Architecture

The baseline architecture for this research is the Cell SPE [2]. The Cell processor is the most advanced chip multiprocessor available today. The SPE is a SIMD core targeted at multimedia, but not very specialized for the target domain. Thus it makes a perfect starting point for investigating specialization for video decoding.

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 core has direct access only to the LS, which contains both instructions and data. Accesses to global memory are done by sending DMA commands to the MFC. This way of memory access is based on the 'shopping list model'. Accessing memory takes a lot of cycles nowadays, just like going to a shopping mall takes a lot of time. In the shopping list model first a list of all the needed data is created, next the entire list of data is transferred from memory, and finally the computation is started. This is faster than the tradition way of starting computation directly and accessing memory (and thus stalling computation) every time some data is needed.

The SPU has 128 registers of 128-bit wide (called quadword). All data transfers between the SPU and the LS are 128-bit wide. Also the LS accesses are 128-bit aligned. The SPU is completely SIMD and the 128 bit vectors can be treated as one quadword, two double words, four words, eight halfwords, or 16 bytes. There is no scalar path in the SPU, but scalar operations are done 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.

The SPU has six functional units, each assigned to either the even or odd pipeline. The SPU can issue two instructions concurrently if they are located in the same halfword and if they execute in a different pipeline. Instructions within the same pipeline retire in order.

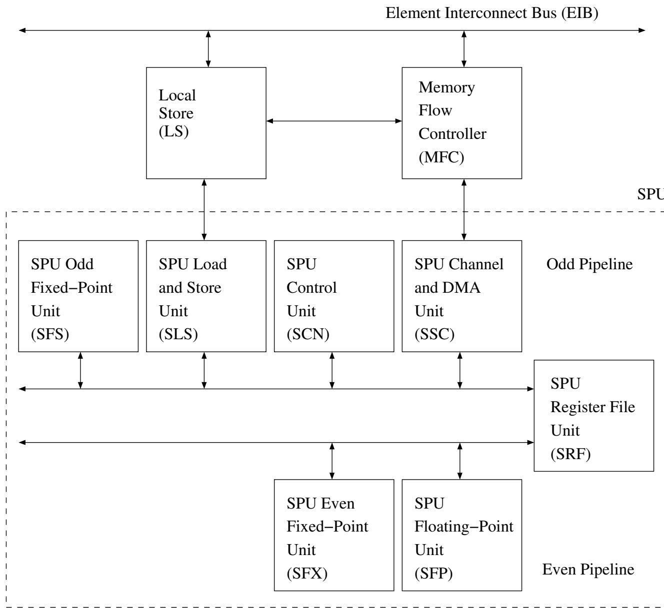

Fig. 1. Overview of the SPE architecture.

III. Experimental Setup

In this section we describe the experimental setup consisting of benchmark, compiler, and simulator.

A. Benchmark

As benchmark we used the kernels of H.264 video decoding. H.264 is currently the best video coding standard in terms of compression and quality [3], however it is also computationally very demanding. One of the best public available H.264 decoders is FFmpeg [4]. We used a version of the kernels of this application that were ported to the Cell SPE [5]. We used the following kernels of H.264 for the analysis in this work: IDCT8 (Inverse Discrete Cosine Transform), IDCT4, and the Deblocking Filter (DF).

B. Compiler

We used the spu-gcc compiler from toolchain 3.4 (that is gcc 4.1.1). Most of the architectural enhancements we implemented were made available to the programmer by intrinsics. Others were only available through using inline assembly. We modified the gcc toolchain to support the new intrinsics and instructions. Details on the procedure to modify the gcc compiler can be found in [6].

The spu-gcc compiler does not always produce the optimal code. Therefore we analyzed the assembly code produced by the compiler and the performance of the code by means of profiling and investigating trace outputs. Based on this analysis we optimized the code of the kernels in order to have a proper baseline to compare to.

C. Simulator

Analysis of the kernels on the Cell SPE could be done using the real processor. However, analyzing architectural enhancements requires a platform that can be modified. Therefore we used CellSim [7]; a Cell simulator build in the Unisim environment.

First, we implemented a profiling tool for CellSim as it did not have one yet. The profiling tool allows to gather many statistics (like cycles, instructions, IPC, etc.) either from the entire application or from a specific piece of selected code. The profiler is controllable from the native code (the code running on the simulated processor). Furthermore, the profiler allows to generate a trace output for detailed analysis of the execution of the kernels.

CellSim has many configuration parameters, of which most of them have a default value that match the real Cell processor. However, the latencies of the SPU pipelines were all set by default to 6 cycles. We adjusted those to match the ones reported in [8] as described in Table I. The latency of branch hints and resolutions and channel instructions are modelled in a different way in CellSim and were not adjusted. In the table we mention both the name of the instruction class as used in the IBM documentation as well as the pipeline name as used in CellSim and in spu-gcc. Each instruction class maps to one of the units depicted in Figure 1 (see [8] for more details).

TABLE I
SPU instruction latencies

Instruction Class	Latency (cycles)	Description	CellSim Pipe
SP	6	single-precision floating-point	FP6
FI	7	floating point integer	FP7
DP	13	double-precision floating point	FPD
FX	2	simple fixed point	FX2
WS	4	word rotate and shift	FX3
BO	4	byte operations	FXB
SH	4	shuffle	SHUF
LS	6	loads and stores	LSP

To improve simulation speed some adjustments were made to the PPE configuration parameters. The issue width was set to 1000 while the L1 cache latency was made zero cycles. These modifications have no effect on the simulation results as the kernels are run on the SPE. However, the simulation of the PPE, which is required to spawn the SPE thread, is improved.

To validate the configuration of CellSim we performed two tests. First, we checked the execution times measured in CellSim by comparing it to the IBM full system simulator SystemSim. The latter is more accurate then using a real processor. Execution times can be measured in the SPE using the system clock. However, this involves a system call which are very slow, compared to the length of the kernels, due to the way they are handled in the Cell processor.

For this analysis we used the DF kernel as it is the largest of all. Table II shows that the difference in execution time between the simulators is 2.1%.The number of executed instructions is slightly different. As the profiling was started and stopped at the exactly same point in the C code, we expect this difference in instruction count to be caused by the profiling

itself.

TABLE II
COMPARISON OF EXECUTION TIMES AND INSTRUCTION COUNT IN CELLSIM AND SYSTEMSIM

	CellSim	SystemSim	error
cycles	8225	8057	2.1 %
instructions	6689	6669	0.3%

Second, we compared trace files generated by CellSim with those generated by the IBM *spu_timing* tool. The latter is a static tool that takes as input an assembly file and generates a pipeline timing diagram. For this analysis we used the IDCT8 and IDCT4 kernels as their code is rather small and thus their trace files are comprehensible. Comparing the two trace files, we conclude that there are only minor differences. CellSim, as it is now, does not consider the odd and even pipeline and sometimes fetches two instructions that are in the same pipeline. In reality this is not possible and the two instructions will be fetched in consecutive cycles. However, we also saw from the trace files that such a faulty dual fetch is typically followed by an extra stall. Thus, in the end this flaw in CellSim does not really show in the results.

From the trace files we also compared the execution times of the inner loop of the IDCT8 and IDCT4 kernels. Table III presents the results and shows that the difference is approximately 3%.

TABLE III
COMPARISON OF EXECUTION TIMES IN CELLSIM AND SPU_TIMING

	CellSim	spu_timing	error
IDCT8	274	266	-3.0 %
IDCT4	117	121	3.3%

Altogether, from these results we conclude that CellSim is sufficiently accurate to evaluate the architectural enhancements described in this paper. Off course, when evaluating architectural enhancements we always compare between results that are all obtained with CellSim.

IV. ACCELERATOR ARCHITECTURE

To specialize the SPU core for video decoding we started by analyzing the bottlenecks in the execution of the kernels. Based on the identified bottlenecks in this section we propose a set of architectural enhancements. We will introduce them based on the observed bottleneck.

A. Matrix Transposition

The basic data element of video coding is the macroblock, which is a matrix of 16×16 pixels. Some operations (like the IDCT) are performed on sub-blocks of 8×8 or 4×4 pixels. Matrix based computation allows the utilization of SIMD instructions to exploit DLP. However, often computations have to be performed both row wise and column wise. Thus matrix transposes are required to rearrange the data for SIMD processing. Indeed, in all the kernels considered in this work, all the data processed is transposed twice.

The conventional way of doing MT with SIMD instructions is performing a series of permutations. For example, in Altivec the `vec_mergeh` and `vec_mergel` instructions can be used. In the SPU core the `shuffle` instruction can be used instead. An 8×8 matrix of halfwords (thus one row of the matrix fits in one SIMD register) can be transposed in three steps of eight instructions each. The result of each step is stored in a different set of registers than the input. Thus a double amount of registers is required. In general this approach requires $\log n$ steps of n instructions. Thus in total the latency is $n \log n$ instructions. For $n = 16$ 64 AltiVec instructions are required.

In this paper we propose the Butterfly instructions that accelerates MT by a factor of two. The new instructions are inspired by Eklundh's recursive matrix transposition algorithm [9], which is illustrated in Figure 2 for an 8×8 matrix. The new instructions have two source registers that are also the destination registers and swap the odd elements of register `ra` with the even elements of `rb`. Thus if `ra` contains the halfwords `a0 a1 a2 a3 a4 a5 a6 a7` and `rb` contains `b0 b1 b2 b3 b4 b5 b6 b7`, then after executing `bflyhw ra rb`, `ra` will contain `a0 b0 a2 b2 a4 b4 a6 b6` and `rb` will contain `a1 b1 a3 b3 a5 b5 a7 b7`. Besides `bflyhw` that swaps halfword elements there are also instructions for double words (`bflydw`), words (`bflyw`), and bytes (`bflyb`).

The Butterfly instructions directly implement the steps of Eklundh's matrix transpose algorithm. The 8×8 MT function using Butterfly instructions first performs four (`bflydw`) instructions, next four (`bflyw`) instructions, and finally four (`bflyhw`) instructions. This implementation requires 12 instructions. Note that this method performs the MT in-place, i.e., the destination registers are the source registers and no additional registers are required for temporal storage. Depending on register file size and the kernel the MT is used in, this might save a lot of loads and stores and further increase performance. For the DF kernel this was exactly the case.

In general for an $n \times n$ matrix each step can be accomplished using $n/2$ instructions. Since there are $\log n$ stages, the total number of instructions is $(n \log n)/2$, assuming the matrix is already loaded in the register file. Thus, when $n = 16$ 32 instructions are required. Indeed our approach also requires half the amount of instructions compared to the AltiVec implementation.

The Butterfly instructions have two register operands and thus fit the RR instruction format of the SPU [10], where the RB slot is not used. The RR instructions have an 11-bit opcode and there were enough opcodes available in the SPU ISA to fit the Butterfly instructions.

A point of interest is the latency of the newly created instructions. The SPU ISA contains a shuffle operation that can perform an arbitrary permutation in four clock cycles. Simple fixed point operations take two cycles. Therefore, it

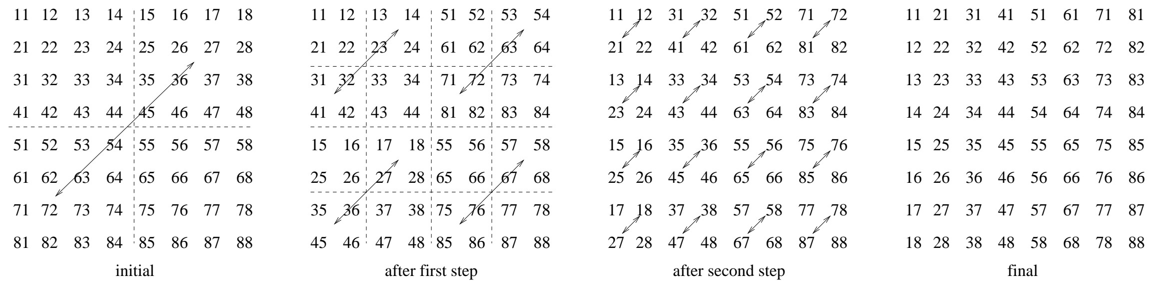

Fig. 2. Eklundh's matrix transpose algorithm

is reasonable to assume a latency of three clock cycles for these predefined permutations. However, the instruction writes to two registers instead of one, which we account for by adding one extra cycle. So in total the latency of the Butterfly instructions is four clock cycles.

B. Scalar Operations

In the SPU scalar operations are performed by placing the scalar in the preferred SIMD slot and using the SIMD FUs. For operations that involve only scalar variables this works fine but if one of the operands is an element of a vector or array variable a lot of overhead is required. In this section we propose two instruction classes that speed up such scalar computations.

1) Adds2v Instructions: Adds2v (ADD Scalar to Vector) instructions add a scalar to a user defined element of a SIMD vector. Figure 3 shows an example from the IDCT8 kernel where the value 32 has to be added to the first element of a vector of shorts, i.e., `block[0] += 32;`. The right side of the figure depicts how the normal SPU ISA handles this code. First, it is calculated (`ai`) by how much bytes the vector must be rotated to put the desired element in the preferred slot. Second, the vector is loaded (`lqd`) into the register file. Third, a mask is created (`chd`) needed by the shuffle. Fourth, the vector is rotated (`rotqby`). Fifth, the value 32 is added (`ahi`) to the desired vector element that now is in the preferred slot. Finally, the result of the addition is shuffled (`shufb`) back into its original position of the vector. Using the Adds2v instructions the same addition of the scalar to the vector can be done in one instruction (or two, including the load as in the example).

```
ai       $2,$4,14           lqd          $8,0($4)
lqd      $8,0($4)           add_shw2vi   $8,32,0
chd      $7,0($4)
rotqby   $2,$8,$2
ahi      $2,$2,32
shufb    $6,$2,$8,$7
```

Fig. 3. Example of how the Adds2v instructions speed up scalar operations, in this case: `block[0] += 32;`. On the left is the normal assembly code, while the right depicts the assembly code using Adds2v instructions.

In the example of Figure 3 we used the `add_shw2vi` instruction that adds a halfword immediate value to a vector of halfwords. The same instruction is available for non-immediate scalars (`add_shw2v`) as well as for the byte data type (`add_sb2vi` and `add_sb2v`). As media kernels usually operate on 8-bit and 16-bit operands these instructions suffice.

The Adds2v instructions fit the `RI7` instruction format except that `add_sb2vi` and `add_shw2vi` have two immediate values. The instructions were added to the simulator and the latency was set to three cycles. A possible implementation uses the FX2 (fixed point) pipeline and replicates the scalar to all SIMD slots of the first input of the ALU. The vector goes to the second input. Some additional control logic selects the proper SIMD slot to do the addition while the others transfer the value from the second input. The latency of the FX2 pipeline is two cycles. Adding another cycle for the increased complexity results in three cycles.

2) Ldsi Instruction: The Ldsi (LoaD Scalar Int) instruction loads a scalar integer from any address (also unaligned) inside the local store and stores it in the preferred slot of a register. The original SPU architecture allows only 16-byte aligned accesses with size of 128 bits. To load a scalar from an unaligned location a lot of additional instructions are required.

Figure 4 shows an example of an array access, i.e., `int a = array[i];` in normal C code. The left part of the figure shows the assembly code generated for the original SPU architecture. The first instruction (`shli`) performs a shift left on the index variable `i`, stored in `$5` by 2 bits. That is, the index is multiplied by 4 and the result is the distance of the desired element to the start of the array in bytes. The second instruction (`ila`) loads the start address of the array into a register. The third instruction (`lqx`) loads the quadword (128 bits) that contains the desired element. It adds the index in bytes to start address of the array, rounds it of downward to a multiple of 16, and accesses the quadword at that address. At this point the desired element of the array is in register `$4` but is not in the preferred slot. The fourth instruction (`a`) adds the index in bytes to the start address of the array. The four least significant bits of this result are used by the last instruction (`rotqby`) to rotate the loaded value such that the desired element is in the preferred slot.

Using the Ldsi instruction the same array access can be done in two instructions, as shown in the right side of Figure 4. The first instruction (`ila`) stores the start address of the array in a register. Next, the Ldsi instruction takes as input the index (`$5`), the start address of the array (`$6`), and the destination

```
shli    $2,$5,2     ila     $6,array
ila     $6,array    ldsi    $2,$6,$5
lqx     $4,$2,$6
a       $2,$2,$6
rotqby  $2,$4,$2
```

Fig. 4. Example of how the Ldsi instruction speeds up code. An array access `int a = array[i];` normally translates to the assembly code on the left, while the right depicts the assembly code using the Ldsi instruction.

register (`$2`). It loads the corresponding quadword from the local store, then picks the correct word and puts it in the preferred slot, and finally stores the whole in the destination register.

Loads and stores are handled by the SLS (SPU Load and Store) execution unit. The SLS has direct access to the local store and to the register file unit. Besides loads and stores, the SLS also handles DMA requests to the LS and load branch-target-buffer instructions. In order to implement the Ldsi instruction a few modifications to the SLS unit are required. First, computing the target address is slightly different then a normal load. Instead of adding the two operands, the index should be multiplied by four before adding it to the start address of the array. This multiplication can be done by a simple shift left. Second, for each outstanding load (probably maintained in a buffer) two more select bits are required to store the SIMD slot in which the element will be located. These two bits are bits 2 and 3 of the target address. For normal loads, these two bits should be set to zero. Finally, when the quadword is received from the local store, the correct word should be shifted to the preferred slot. This can be done by inserting a 4-to-1 mux in the data path which is controlled by the select bits.

The Ldsi instruction loads only integer scalars from the local store. Similar instructions could be implemented for other data types like shorts and chars, but we did not find that necessary for video decoding.

The Ldsi instruction fits the `RR` instruction format. The Ldsi instruction was added to the simulator and the compiler and the latency was set to seven cycles. A normal load or store operation takes six cycles, and we added one cycle to account for the extra mux in the datapath.

C. Lack of Saturating Arithmetic

All the kernels considered in this paper have as output a matrix of unsigned 8-bit values. However, in order not to lose precision in intermediate results, all computation is done using 16-bit values. This requires a saturation of all results between 0 and 255 before packing the 16-bit signed value to an 8-bit unsigned value.

The SPU architecture does not have any saturating instructions and therefore this operation is costly as it has to be performed using multiple compare and select instructions. Implementing a full set of saturating arithmetic instructions has limited value for video coding as the use of 16-bit values is necessary anyway. We propose two instruction classes that perform saturating operations and that suffices for video decoding. The first is very generic and useful for many application domains, while the second is more specific for the video decoding.

1) Clip Instructions: The Clip instructions saturate the elements of a SIMD vector between two values. A Clip instruction can be used to emulate normal saturating arithmetic but also to saturate a value between user defined boundaries. For example, in de deblocking filter kernel the index of a table lookup has to be clipped between 0 and 51. In the normal SPU architecture this operation is done using four instructions (see Figure 5). First, two masks are created by comparing the input vector with the lower and upper boundaries. Second, using the masks the result is selected from the input and the boundary vectors.

```
vsint16_t clip(vsint16_t a, vsint16_t min,
                                vsint16_t max) {
    vuint16_t min_mask, max_mask;
    min_mask = spu_cmpgt(min, a);
    max_mask = spu_cmpgt(a, max);
    vsint16_t temp = spu_sel(a, min, min_mask);
    return spu_sel(temp, max, max_mask);
}
```

Fig. 5. C code of a clip function using the normal SPU ISA. The Clip instructions perform this operation in one step.

The Clip operation is mainly used in signed integer and short computations. For other data types the function could be implemented as well, but we did not find that necessary.

The Clip instructions fit the `RR` instruction format if the source and destination register (of the vector to clip) are the same. The Clip instructions were added to the simulator and the latency was set to four cycles. A normal compare operation takes two cycles. The two compares can be done in parallel but that requires that three quadwords are read from the register file into the functional unit. Another approach would be to serialize the operation. First two operands are loaded, while these are compared and the result is selected the third operand is loaded, and finally the latter is compared to the result of the first compare. We added one cycle to account for loading three quadwords and also one cycle to account for the extra mux in the datapath that selects the correct value.

2) Asp Instructions: In video decoding, a saturation to 8-bit unsigned is generally followed by a pack operation. The Asp (Add, Saturate, Pack) instructions exploit this by combining an addition, a saturation, and a pack. The saturation is performed between the fixed points 0 and 255 and thus the 'min' and 'max' operands as in the Clip instructions are not necessary. This allows to have two operands that are added first, before saturating and packing. The code of Figure 6 is taken from the IDCT8 kernel and shows why this is useful.

The code adds the result of the IDCT to the predicted picture. First it adds two vectors of 16-bit values. Second, the elements of the vector are saturated between 0 and 255. Finally, the elements are packed to 8-bit and stored in the first half of the vector. Using the normal SPU ISA many instructions and auxiliary variables are needed. The Asp instructions

```
vsint16_t va, vb, vt, sat;
vuint8_t vt_u8;
const vuint8_t packu16 =
    {0x01,0x03,0x05,0x07,0x09,0x0B,0x0D,0x0F,\
     0x11,0x13,0x15,0x17,0x19,0x1B,0x1D,0x1F}
vsint16_t vzero = spu_splats(0);
vsint16_t vmax = spu_splats(255);
vt = spu_add(va,vb);
sat = spu_cmpgt(vt,vzero);
vt = spu_and(vt,sat);
sat = spu_cmpgt(vt,vmax);
vt = spu_sel(vt,vmax,sat);
vt_u8 = spu_shuffle(vt,vzero,packu16)
```

Fig. 6. Example code from the IDCT8 kernel that can be replaced by one Asp instruction. The code adds the vectors `va` and `vb`, saturates them between 0 and 255, and packs the elements to 8-bit values.

are tailored for this kind of operation and can perform the entire operation at once.

The Asp instructions fit the `RR` instruction format. For each type of operands an Asp instruction could be implemented, but we found it only necessary to implement one: `asp_u8`. It takes as input two vectors of 16-bit signed values (vsint16_t) and stores the output in a vector of 8-bit unsigned values (vuint8_t). The eight elements are stored in the first eight slots while the last eight slots are assigned value zero.

The instruction was added to the simulator and the latency was set to four cycles. A normal addition costs two cycles and we added two more cycles for the saturation and the data rearrangement.

D. Regular Simple Arithmetic Expressions

The SPU ISA contains all the basic arithmetic operations like add, subtract, multiply, and shift, and a few special (although standard in most ISAs) arithmetic operations as average and absolute difference. However, to increase efficiency and performance of the kernels certain regular expressions can be combined into one instruction. In this section we propose two such classes of instructions namely Sfxsh (Simple FiXed point and SHift) instructions and IDCT instructions.

1) Sfxsh Instructions: Sfxsh instructions combine simple fixed point operations with a shift operation. Figure 7 shows an example from the IDCT8 kernel where one operand is shifted and added to the second (`b1 = a7>>2 + a1;`). Using a normal ISA this one line of code would require two instructions.

```
rotmahi  $10,$9,2        shadd   $11,$8,$9,2
ah       $11,$10,$8
```

Fig. 7. Example of how the Sfxsh instructions speed up code, in this case `b1 = a7>>2 + a1;`. On the left is the normal assembly code, while the right depicts the assembly code using Sfxsh instructions.

This kind of code is found regular in the kernels we investigated. More general, they can be characterized as follows. There are two register operands, a simple arithmetic operation (like addition or subtraction), and one shift right by a small fixed amount of bits. Our analysis showed that for video decoding the instructions in Figure 8 are beneficial. These instructions could be implemented for each operand type, but we found it sufficient to implement them for a vector of 16-bit signed values.

```
shadd   rt,ra,rb,imm  \\rt = ra + rb>>imm
shsub   rt,ra,rb,imm  \\rt = ra - rb>>imm
addsh   rt,ra,rb,imm  \\rt = (ra + rb)>>imm
subsh   rt,ra,rb,imm  \\rt = (ra - rb)>>imm
```

Fig. 8. Overview of the Sfxsh instructions.

The Sfxsh instructions could fit the `RRR` instruction format of the SPU ISA if all operands would be stored in registers. The `RRR` format uses four bits for the opcode and 7 bits for each of the four register operands. However, this approach would cause an unnecessary load as the right operand of the shift can be an immediate. Furthermore, there are not enough opcodes of the `RRR` format available to implement all four instructions. It is the case, though, that the shift count is always between 0 and 7. Thus, if we use and immediate value for the shift count it can be represented by 3 bits only. Using this observation we defined a new instruction format RRI3 that uses 8 bits for the opcode, 3 bits for the immediate value, and 7 bits for each of the three register operands.

The simulator was modified to recognize the new `RRI3` instruction format and the new instructions. The latency of the Sfxsh instructions was set to five cycles. Normal shift and rotate instructions take four cycles. We added one cycle to this for the extra addition or subtraction.

2) IDCT Instructions: The IDCT kernel performs an inverse discrete cosine transform on either an 8×8 or 4×4 block of pixels. Performing the IDCT comprises of a one dimensional IDCT row wise followed by a one dimensional IDCT column wise. The computational structure of the one dimensional IDCT8 is depicted in Figure 9. If elements 0 through 7 are a row of the matrix, then this diagram represents a row wise one dimensional IDCT8. The diagram also shows that the IDCT consists of simple additions, subtractions, and some shifts.

The typical way to SIMDize this one dimensional IDCT is to perform the operations on vectors of elements. Thus for the row wise 1D IDCT a vector would be a column of the matrix, while for the column wise 1D IDCT a vector would be a row of the matrix. As a matrix is stored in memory with the elements of a row in consecutive addresses, the SIMDized 1D IDCT can only be performed column wise. To compute the row wise 1D IDCT the matrix is transposed. Thus, the whole IDCT is performed in four steps: matrix transpose, column wise 1D IDCT, matrix transpose, column wise 1D IDCT.

We propose the IDCT8 instruction that performs the row wise 1D IDCT8. This instruction takes one row of the matrix as input, performs the arithmetic operations as depicted in Figure 9, and stores the result in a register. Thus, for an 8×8 matrix the total row wise 1D IDCT8 takes eight instructions. Besides the reduction in the executed instructions, also the two matrix transposes are avoided because the entire IDCT

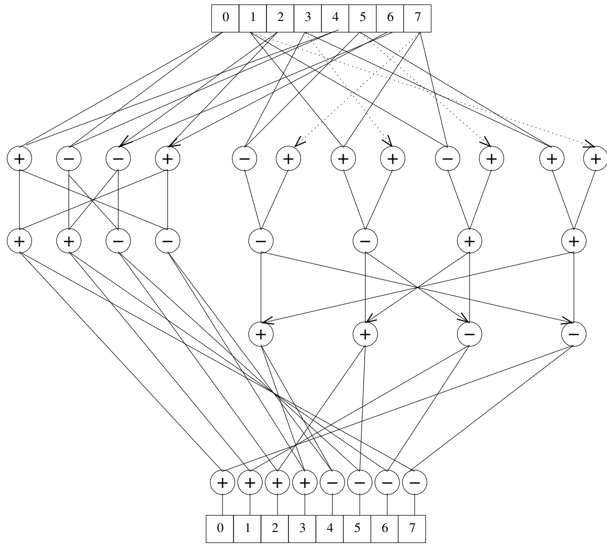

Fig. 9. Computational structure of the one dimensional IDCT8. Arrows indicate a shift right by one bit. A dotted line means the two operands are the same.

kernel can be performed as follows: row wise 1D IDCT (using ICDT8 instruction), column wise 1D IDCT (using traditional SIMDimization).

The IDCT8 instruction fits the RR format where the RB slot is not used. Register RA is the input register while RT is the output register. Usually these two would be the same but the instruction format allows them to be different. The IDCT8 instruction assumes its operand to be a vector of 16-bit signed values. The instruction was added to the simulator and the latency was set to eight cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The IDCT8 instruction performs four steps of simple fixed point operations and thus the latency of the new instruction is four times two making a total of eight cycles.

The IDCT4 kernel is similar to the IDCT8 kernel but operates on a 4×4 pixel block. The computational structure of the one dimensional IDCT4 is simpler than that of the IDCT8 as shown by Figure 10.

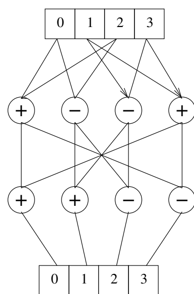

Fig. 10. Computational structure of the one dimensional IDCT4. An arrow indicates a shift right by one bit.

For the IDCT4 kernel a similar approach could be used as for the IDCT8. However, as a 4×4 matrix of 16-bit elements completely fits in two registers of 128-bits, the entire two dimensional IDCT4 can be performed using one instruction with two operands. This instruction reads the matrix from its two operand registers, performs a row wise and a column wise 1D IDCT4, and stores the result back in the two registers. Storing the result could be done in one cycle if the the register file has two write ports, or otherwise two cycles are required. The IDCT4 instruction also performs a shift right by 6 bits on the final result. This operation is part of the IDCT kernel.

The IDCT4 instruction also fits the RR format where the RB slot is not used. Register RA is the input and output register that contains the first two rows of the matrix while RT holds the last two rows. The IDCT4 instruction assumes its operands to be vectors of 16-bit signed values. The instruction was added to the simulator and the latency was set to ten cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The IDCT4 instruction performs four steps of simple fixed point operations making a total of eight. We also count one cycle for the final shift (although this could be hardwired) and one cycle for writing the second result to the register file.

E. Other Bottlenecks

Besides the bottlenecks presented above, there are two more worth mentioning. The solutions to these bottlenecks are well known, have low novelty, and thus are not investigated in this work.

First, there is a lot of overhead due to pack and unpack. These operations have to be performed explicitly using shuffle operations with the correct mask. The Asp instructions reduce the overhead of packing partially, by combining it with saturation and addition. The overhead could be further reduced by implementing load and store instructions with auto unpack and pack respectively.

Second, the alignment of the memory accesses introduces overhead as well. To access elements that are not 16 byte aligned, additional instruction are necessary like extra loads, computation of the shift count, performing the shift, etc. The solution is to allow accesses to unaligned memory addresses by modifying the local store.

V. Performance Evaluation

In the previous section we proposed architectural enhancements to specialize the Cell SPU core for video decoding. In this section we evaluate the impact of each of the architectural enhancements on the three kernels (IDCT8, IDCT4, and DF) as well as the aggregate effect of those.

A. IDCT8 Kernel

Table IV shows the enhancements that are beneficial to the IDCT8 kernel and state the effect on the execution time and the instruction count. First it shows what the effect is of each of the enhancements separately. Finally, also the combination of all enhancements was evaluated. The latter excludes the Butterfly instructions as they become obsolete when using the IDCT8 instruction, which eliminates matrix transposition.

TABLE IV

SPEEDUP AND REDUCTION IN INSTRUCTION COUNT OF THE IDCT8 KERNEL USING THE PROPOSED ARCHITECTURAL ENHANCEMENTS. THE EFFECT OF EACH SEPARATE ENHANCEMENT IS LISTED AS WELL AS THE AGGREGATE EFFECT.

Enhancement	Speedup	Reduction instructions
Butterfly	1.04	10.4%
Sfxsh	1.09	10.8%
Adds2v	1.00	2.8%
Asp	1.15	15.9%
IDCT8	1.37	31.5%
All (but Butterfly)	2.01	55.8%

The speedup achieved by the enhancements lies between 1.00x and 1.37x. The IDCT8 instruction achieves the largest speedup, mostly due to avoiding matrix transposition. The Adds2v instructions separately do not improve execution time. However, combining all enhancements it is responsible for 6% of the total speedup. A synergistic effect happens when combining all enhancements such that the total speedup is larger than the multiplication of the separate speedups. The source of this synergistic effect seems to be the compiler optimization process.

The speedup achieved when combining all architectural enhancements is 2.01x. Note that also a reduction in the number of instructions of 55.8% is achieved. The latter is important for power efficiency which has become a major design constraint in nowadays processor design.

B. IDCT4 Kernel

Although the IDCT4 kernel is very similar to the IDCT8, the Butterfly and Sfxsh instructions were of no value. Table V shows the remaining enhancements and their effect on the execution time and the instruction count. The speedup achieved by the enhancements lies between 1.07x and 1.24x. Again a synergistic effect happens when combining all enhancements such that the total speedup is larger than the multiplication of the separate speedups. The speedup achieved when combining all architectural enhancements is 1.69x and the number of instructions is reduced by 54.8%. The total speedup achieved is less than for the IDCT8 kernel. This is due to a relatively larger amount of kernel overhead (less computation, same overhead).

TABLE V

SPEEDUP AND REDUCTION IN INSTRUCTION COUNT OF THE IDCT4 KERNEL USING THE PROPOSED ARCHITECTURAL ENHANCEMENTS. THE EFFECT OF EACH SEPARATE ENHANCEMENT IS LISTED AS WELL AS THE AGGREGATE EFFECT.

Enhancement	Speedup	Reduction instructions
Adds2v	1.07	5.5%
Asp	1.14	18.6%
IDCT4	1.24	29.8%
All	1.69	54.8%

TABLE VI

SPEEDUP AND REDUCTION IN INSTRUCTION COUNT OF THE DF KERNEL USING THE PROPOSED ARCHITECTURAL ENHANCEMENTS. THE EFFECT OF EACH SEPARATE ENHANCEMENT IS LISTED AS WELL AS THE AGGREGATE EFFECT.

Enhancement	Speedup	Reduction instructions
Butterfly	1.34	26.6%
Ldsi	1.13	8.5%
Clip	1.06	5.1%
All	1.84	42.8%

C. Deblocking Filter Kernel

Table VI shows which enhancements were applied to the DF kernel and presents the speedup and reduction in instruction count. The Butterfly instructions have a large impact on the execution time, mainly because the matrix transposes have become in-place. This allows to keep the entire macroblock in the register file, and thus save a lot of loads and stores. To achieve this some modifications to the code were required such that the compiler could ascertain all data accesses at compiler time. These modifications included loop unrolling, substituting pointers for array indices, and inlining of functions.

Both the Ldsi and Clip instructions provide a moderate speedup but significant considering the total kernel. When combining all three enhancements again a synergistic effect happens as the total speedup is larger than the multiplication of the separate speedups. The reduction in the number of instructions is proportional to the speedups. Combining all enhancements the number of instructions is reduced with 42.8%.

VI. CONCLUSION

As a case study for future chip multiprocessors we have presented the specialization of the Cell SPE for video decoding. We analyzed the H.264 video decoding kernels and identified the execution bottlenecks. Based on these bottlenecks we proposed ISA extensions.

We proposed Butterfly instructions to speed up matrix transposition. Besides faster execution the matrix transposes are also performed in-place and thus uses half the amount of registers. In the deblocking filter kernel this saves many loads and stores. We enhanced scalar operation with two instruction classes. The Adds2v instructions add a scalar to a user defined element of a vector. The Ldsi instructions load an integer from an arbitrary memory address into the preferred slot of a register. Because of the lack of saturating arithmetic we proposed the Clip instructions, which saturate a value between user defined boundaries. The Asp instruction is tailored for the IDCT kernel and combines an addition, saturation, and a pack. We also proposed a few instruction classes that combine regular simple arithmetic expressions. The Sfxsh instructions combine an addition or subtraction with a shift right on one operand or the result. The IDCT8 instruction performs a one dimensional IDCT8 on a row of an 8×8 matrix. The IDCT4

instruction performs the entire IDCT4 computation on a 4×4 matrix, stored in two 128-bit registers.

The proposed architectural enhancements speed up the H.264 kernels up to a factor of 1.37 each. The aggregate speedup achieved by the enhancements is 2.01 for the IDCT8 kernel, 1.69 for the IDCT4 kernel, and 1.84 for the deblocking filter kernel. The number of executed instructions was reduced by 42.8% to 55.8%. This translates to a higher power efficiency, which is a major design constraint for CMPs.

REFERENCES

[1] C. Meenderinck and B. Juurlink, "(When) Will CMPs hit the Power Wall?" in *Proceedings of the Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2008.

[2] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty *et al.*, "The Microarchitecture of the Synergistic Processor for a CELL Processor," in *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[3] T. Oelbaum, V. Baroncini, T. Tan, and C. Fenimore, "Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard," in *Int. Broadcast Conference (IBC)*, 2004.

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

[5] A. Azevedo, C. Meenderinck, B. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of video filtering on the cell processor," in *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, May 2008.

[6] C. Meenderinck, "Implementing spu instructions in cellsim." [Online]. Available: http://pcsostres.ac.upc.edu/cellsim/doku.php/

[7] "Cellsim: Modular simulator for heterogeneous multiprocessor architectures." [Online]. Available: http://pcsostres.ac.upc.edu/cellsim/doku.php/

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

[9] J. Eklundh, "A fast computer method for matrix transposing," *IEEE Transactions on Computers*, vol. 100, no. 21, pp. 801–803, 1972.

[10] *Synergistic Processor Unit Instruction Set Architecture*, IBM, 2007. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326