VECTOR PROCESSING IN SCALAR PROCESSORS FOR SIGNAL PROCESSING ALGORITHMS

Michael T. Brady, J. Q. Trelewicz

IBM Printing Systems Division
6300 Diagonal Highway, Boulder, CO 80301
{mtbrady,trelewic}@us.ibm.com

Joan L. Mitchell

IBM T. J. Watson Research Center
Route 134, Yorktown Heights, NY 10598
joanm@us.ibm.com

ABSTRACT

Product requirements often dictate the use of off-the-shelf processors for very fast signal processing applications. Additionally, restrictions on cost, power, or size/weight may preclude the use of specialized vector processors for implementation of the algorithms. We discuss a new method for performing signed parallel processing in scalar, off-the-shelf processors for integerized signal processing algorithms. Uniform data precision may be used, but is not required for the method. It is shown that the reduction in execution cycles resulting from this implementation is approximately linear in the size of the registers, divided by the precision required.

1. INTRODUCTION

Fast product schedules often require the use of standard components, where available, for product development. In the case of signal processing products, this can mean the use of off-the-shelf processors, which may be organized into multiple-processor systems with parallel computing capability. However, many popular signal processing algorithms can be calculated with limited precision without affecting the acceptability of the result. For example, the Integer Cosine Transform (ICT) [1] can be used to approximate the DCT in integer processors with configurable precision. In these cases where limited precision may be used, the additional bits provided by 32-bit, 64-bit and larger processors are "wasted" during the calculation. Exploiting this redundant precision capability can afford the developer even greater parallel processing capability in conventional processor configurations.

The methods described in this paper[1] were developed for addressing this need with two's-complement arithmetic. Data which can be processed in parallel, because of the algorithm structure, are packed in a "vector" format (described below) into registers. Many signed arithmetic operations can be performed on these vectors, including addition, subtraction, multiplication by scalars, shifting, and others. When the parallel processing is completed, the vectors can be unpacked into scalar values for storage or subsequent processing. The importance of these methods lies in their handling of carries and borrows in the packed vectors format. The vector method can be especially advantageous in speeding up operations such as the direct 2-D DCT [2].

Methods for performing parallel processing of data streams in conventional registers have been used for applications such as the parallel parsing of data communication frames [3]. However, such bit parsing methods are not appropriate for signal processing applications, where a range of arithmetic operations must be applied to the data. A flexible arithmetic logic unit (ALU) with carry blocks between each bit has been designed [4]. This ALU allows registers to be partitioned into multiple data for performing parallel processing. In contrast, the method described in this paper allows multiple data to be processed as vectors in registers of conventional processors and ALUs.

2. VECTOR REPRESENTATION

Because carries and borrows in any one element, or member of the vector, will propagate left through the register, multiple signed data packed into a single register may be affected by operations performed on the other elements in the register. For this reason, the elements are stored in the register in a "vector" representation, which allows the signed elements to be broken back into scalar values after parallel processing is complete.

A vector comprising k packed elements α_j numbered from 1 at the right of the register through k at the left of the register will be denoted $(\alpha_k, \cdots, \alpha_2, \alpha_1)$. Denote the corresponding unpacked data as $\langle \alpha_k, \cdots, \alpha_2, \alpha_1 \rangle$.

2.1. Packing the vector

Before the vector can be packed, the number of bits required for the worst-case final value (after processing) must

[1] Patents have been applied for. Joan L. Mitchell is currently on temporary assignment at IBM PSD.

be known. For example, addition and subtraction operations each require no more than one bit of additional precision to accommodate their worst-case results, while multiplication by an m-bit number requires no more than m additional bits to hold the result. The worst-case additional precision required can be obtained by calculating the required precision for each operation. On the other hand, the programmer may also have specific knowledge of the data and the operations, which may allow the final precision to be allocated more tightly; e.g., the programmer may know that a specific algorithm taking m additions requires only $k < m$ bits for the result because of the structure of the data, allowing tighter packing of the vector.

Call the number of bits allocated for the operations n_{o}. Note that, because of the parallel processing, n_{o} must be the maximum needed for any element in the vector. An additional bit may need to be allocated if the one-cycle unpack described in Section 2.2 is used. Call the required number of these additional bits n_{up}, which may be larger than one if unpacking is performed several times during vector processing. (This additional bit is only required for elements which propagate borrows to the left in the register, so the additional bit is never required for the leftmost element.) Furthermore, as described in Section 2.3, the maximum-magnitude negative number cannot be permitted to occur in any element. This may require the allocation of an additional bit to prevent this condition, denoted $n_{\mathrm{n},j} \in \{0, 1\}$ for the jth element.

Call the input precision of the jth element $n_{\mathrm{i},j}$. Then, for an N-bit register, the packed vector can contain k elements only if

$$N \quad \geq \quad kn_{\mathrm{o}} + (k-1)n_{\mathrm{up}} + \sum_{j=1}^{k} (n_{\mathrm{n},j} + n_{\mathrm{i},j}) \, .$$

Packing of the input values is then performed as follows. The ℓth element to be packed is shifted left by $(\ell - 1)(n_{\mathrm{o}} + n_{\mathrm{up}}) + \sum_{j=1}^{\ell-1} n_{\mathrm{n},j} + n_{\mathrm{i},j}$ bits and added to the register contents. (Cycles may be saved by noticing that non-negative elements may all be shifted into position and added into the register at once, since these elements have no sign bits to propagate through the vector.) Packing is done in this manner to ensure that the sign bits of negative elements are propagated left through the vector, which is necessary to accommodate signed operations.

2.2. Unpacking the vector

When the vector operations have been completed, the elements may be unpacked for storage or subsequent processing of the elements. Unpacking may be done using one of two methods. The first method, which is used in the example of Section 4, requires an additional sign bit, so that $n_{\mathrm{up}} = 1$, but unpacking may then be performed in fewer steps. A mask is generated with a one bit at every element's leftmost bit (except for the leftmost element), and zeros elsewhere. The mask is "and"ed with the register contents and added back to the register (some processors allow a mask-and-copy operation to be performed in a single cycle.) This causes the sign bits, which reflect the previous propagation of borrows, to be propagated through the register, removing the effect of previous borrows. The leftmost bit of each element (except the leftmost element) is no longer a sign bit and should be discarded. Now, the jth element is stored in two's-complement notation in its own $n_{\mathrm{o}} + n_{\mathrm{n},j} + n_{\mathrm{i},j}$ bits of the register, and can be masked off and stored elsewhere.

Another method of unpacking allows $n_{\mathrm{up}} = 0$ but uses extra cycles of execution. Starting with the rightmost element in the register and moving left one element at a time, the leftmost (sign) bit of the element is masked, shifted left by one bit, and added to the next element. This removes the effects of propagated borrows in the register, one element at a time. It is simple to verify that this procedure cannot be performed in the single step of the single-step sign mask and add procedure described above: consider a vector comprising $(-1, 0, -1)$. With two total bits per element, the vector notation is 10 11 11, which, if added with the single-step shifted sign bit mask of 01 01 00 yields 00 00 11, the incorrect unpacked result.

2.3. Carries and borrows

An important feature of this vector method is its handling of carries and borrows between elements: a borrow or carry incurred on one element in the vector propagates through the register to the left, potentially affecting all bits (and thus all elements) to the left.

Firstly, a borrow can never follow a borrow in an element: borrows can only occur through sign reversal from non-negative to negative, and from underflow. The way in which precision is allocated in the register, including the use of the $n_{\mathrm{n},j}$ parameter, ensures that underflow can never occur. (If the maximum-magnitude negative number rule were not used, a maximum-magnitude negative number with a borrow propagating left from another element could underflow.)

Furthermore, the restriction on the allocated precision for each element ensures that an overflow cannot occur. Thus, the only way in which a carry can occur in an element is through a sign reversal from negative to non-negative. Note that the packing of the elements ensures that a negative element will always have propagated a borrow left as part of the packing. Furthermore, an element changing sign from non-negative to negative will always propagate a borrow left in the register as part of its sign reversal. Thus, a carry can only follow a borrow. The result is that the single borrow

in the element under consideration resembles the addition of -1 to the element immediately to the left, while the +1 of the carry cancels this -1.

The vector data may thus be related to the two's complement notation of unpacked elements by considering each element initiating a propagating borrow (i.e., changing sign from non-negative to negative) to add -1 to the element immediately to the left. Thus, the vector data can be mapped uniquely to the unpacked signed elements by the following:

$$\begin{aligned}&(\alpha_k, \cdots, \alpha_2, \alpha_1)\\ &\quad \equiv \langle \alpha_k[-b_{k-1}], \cdots, \alpha_2[-b_1], \alpha_1 \rangle\,, \end{aligned} \tag{1}$$

where $b_j \in \{0, 1\}$ signifies the occurrence or nonoccurrence of a propagating borrow from element j. Since a borrow only propagates left from element j if element j has changed sign (or been originally packed as negative), at most one borrow can occur on any one element, including the possible sign-change effect from propagating borrows. This is the justification for using the $n_{\mathrm{n},j}$ parameter.

3. VECTOR ARITHMETIC

This vector architecture supports a number of arithmetic operations, which are described in detail below.

3.1. Negation

As is well known, two's complement negation essentially inverts each bit in the register, then adding one to the rightmost bit. That register negation results in vector negation can be seen by considering (1):

$$\begin{aligned}&-\langle \alpha_k[-b_{k-1}], \cdots, \alpha_2[-b_1], \alpha_1 \rangle\\ &\quad = \langle -\alpha_k - 1[+b_{k-1}], \cdots, -\alpha_2 - 1[+b_1], -\alpha_1 \rangle\\ &\qquad \text{since the +1 is only added to the rightmost element}\\ &\quad = \left\langle -\alpha_k[-b'_{k-1}], \cdots, -\alpha_2[-b'_1], -\alpha_1 \right\rangle,\end{aligned}$$

where b'_j is one iff element j is negative (i.e., was positive before the negation), and zero else. That this works when $\alpha_j = 0$ results from the left-propagating carry resulting from the negation of zero: in this case, $b_j = 0$, so that $-1 + b_j + \text{carry} = 0 = b'_j$.

3.2. Addition and subtraction

Addition and subtraction between vectors require that the two vectors under consideration use the same element alignment. Addition which does not result in a carry or borrow in any one element is straightforward, since no carries or borrows will be propagated left through the register. The generation of a carry or borrow in any element will cause the propagation of such carry or borrow left through the register. However, since the ALU performs addition from right to left in the register, the carries and borrows from all elements are propagated essentially one at a time, following the same form as the packing of the register.

Subtraction works by a similar principle, since it is equivalent to negating the one vector and adding it to the other vector.

3.3. Scalar multiplication

Scalar multiplication by a positive number follows from addition (the effective adding of the vector to itself multiple times), while scalar multiplication by a negative number also follows from negation.

3.4. Shifts

The operation of a left shift is straightforward, since each single-bit shift left is essentially equivalent to multiplication by two, which was discussed above. Arithmetic shift right requires unpacking the elements, performing shifts and sign extensions, and then repacking the data.

3.5. Compares

Determining whether an element is in a range around zero (for example, as is done in quantization to zero), requires having the lower boundary of the range for comparison packed into a vector. The vector of boundaries is subtracted from the vector under test. The vector is then unpacked, and each element is compared to its range individually by shifting the unpacked vector left until the element under test is at the left of the register. Since the test is whether an element is less than the range in question, the remaining unpacked elements to the right of the element under test in the register do not affect the result of the comparison.

Arithmetic non-equality compares exploit the additional bit of precision that may be allocated to ensure that the maximum-magnitude negative number cannot occur in any element in the vector. The vector need not be unpacked prior to comparison. Consider the jth element under test, α_j. The leftmost bit of the $j-1$th element is one if a borrow has been propagated into the jth element, and zero otherwise. (The rightmost element can be considered to always have this adjacent bit be zero.) The comparison can then be done directly on the bits corresponding to the (packed) jth element concatenated with this extra bit of precision (call this the jth pattern.) If C is the number against which to compare, the comparison pattern will be $2C - 1$ to accommodate the possibility of the borrow. If no borrow occurred on the element under test, the test will compare $2C - 1$ to $2\alpha_j$. If a borrow occurred, the test compares to $2\alpha_j - 1$. This structure allows for testing $\geq, >, <, \leq$.

4. EXAMPLE

This example is intended to show the operation of this vector computing architecture. A small number of bits are used to illustrate the principles without introducing unneeded complexity.

Consider $k = 3$, $n_{\mathrm{o}} = 4$, $n_{\mathrm{u}} = 1$, and $n_{\mathrm{i},j} = 3$ for $j \in \{2, 3\}$ and 4 for $j = 1$. All steps are shown in Table 1. Then $v_1 = (2, 0, -7)$ is represented in the register as shown in the table to $\langle 2[-1], 0[-1], -7\rangle$. Similarly, $v_2 = (-1, 1, 5)$ is represented as $\langle -1, 1, 5\rangle$. The intermediate result $v_1 + v_2$ is expected to be $(1, 1, -2) = \langle 1, 1[-1], -2\rangle$, which is verified by unpacking. Multiplying by -2 is shown first as multiplying by 2 and then by negation (Only two bits were reserved in n_{o} for the multiplication, since the magnitude of the scalar multiplier takes only two bits.) The intermediate result is $(-2, -2, 4) = \langle -2[-1], -2, 4\rangle$, which is verified by unpacking. The final operation shown subtracts from $-2(v_1+v_2)$ the vector $v_3 = (1, -2, 6) = \langle 1[-1], -2, 6\rangle$. The unpacked final result is verified to be the expected $(-3, 0, -2) = \langle -3[-1], 0[-1], -2\rangle$.

This vector-arithmetic method can be applied to the DCT to increase the throughput of a JPEG encoder or decoder. With 8-bit input data, $n_{\mathrm{i}} = 8$ for all input data in the block. Consider the 1-D DCT, described by the matrix equation $DF = \hat{F}$. For an 8×8 block of data, the 8 columns of F may be packed into one or more registers in parallel, resulting in a computational speedup on the order of the number of elements in the vector (minus packing and unpacking overhead) by performing the transform on columns packed as vectors. The same method may be used for the transform of the rows with the transpose of D.

5. CONCLUSIONS

This vector method for parallel data processing has been shown to allow signed arithmetic operations on the vectors, with capability for unpacking to signed scalar data after parallel processing is complete. When the register accommodates n signed elements in a vector, a speedup on the order of n is shown, because of the low overhead associated with packing and unpacking. It is anticipated that this method can be extended beyond the DCT to a number of other signal processing algorithms that lend themselves to parallel execution of operations.

6. REFERENCES

[1] T.-C. J. Pang, C.-S. O. Choy, C.-F. Chan, and W.-K. Cham, "A self-timed ICT chip for image coding," *IEEE trans. circuits and systems for video technology*, vol. 9, no. 6, pp. 856–860, 1999.

[2] I. Kuroda, "Processor architecture driven algorithm optimization for fast 2D DCT," in *VLSI Signal Processing, VIII*, (Sakai, Japan), pp. 481–490, 1995.

[3] J. Calvignac, J. Feraud, B. Naudin, C. Pin, and E. Saint-Georges, *Parallel processing method and device for receiving and transmitting HDLC SDLC bit streams*. US Patent 5119478, 1992.

[4] M. D. Bates, N. D. Butler, A. C. Gay, J. H. Kim, and R. M. West, *Arithmetic logic unit*. US Patent 5081607, 1992.

	00000000	00000000	000000000
	00000010	00000000	000000000
+	11111111	11111111	111111001
	00000001	11111111	111111001

Packing v_1

	00000001	11111111	111111001
+	11111111	00000001	000000101
	00000001	00000000	111111110

$v_1 + v_2$

	00000001	00000000	111111110
+	00000000	00000000	100000000
	00000001	x0000001	x11111110

$v_1 + v_2$ unpacked

	00000001	00000000	111111110
×	00000000	00000000	000000010
	00000010	00000001	111111100
×	11111111	11111111	111111111
	11111101	11111110	000000100

$-2(v_1 + v_2)$

	11111101	11111110	000000100
+	00000000	10000000	000000000
	11111110	x1111110	x00000100

$-2(v_1 + v_2)$ unpacked

	11111101	11111110	000000100
-	00000000	11111110	000000110
	11111100	11111111	111111110

$-2(v_1 + v_2) - v_3$

	11111100	11111111	111111110
+	00000000	10000000	100000000
	11111101	x0000000	x11111110

$-2(v_1 + v_2) - v_3$ unpacked

Table 1: Vector arithmetic example