Preferred Strategies for Optimizing Convolution on VLIW DSP Architectures

Dr. Jagadeesh Sankaran, Applications Engineer, Texas Instruments, 8505 Forest Lane, Dallas, Texas, USA sankaran@ti.com
Dr. William Pervin, Dept of Electrical Engineering, University of Texas, Richardson Dallas, Texas, USA pervin@utdallas.edu
Dr. Cyrus D. Cantrell, Dept of Electrical Engineering, University of Texas, Richardson Dallas, Texas, USA cantrell@utdallas.edu

1. Abstract
Convolution is a central algorithm for implementing linear time invariant systems that constitute the heart of most digital signal processing algorithms. Performance on the linear convolution algorithm has been one of the primary benchmarks used to discern the performance of dedicated digital signal processing architectures (DSP). While DSP benchmarks are far more varied and complex than just convolution, this paper intends to focus on one of the most ancient benchmarks for which DSP architectures were initially architected to highlight how optimization strategies have changed on VLIW architectures even for simple algorithms.

This paper develops novel optimality metrics to provide a quantitative assessment on the efficiency of various implementations that are developed. While the convolution algorithm has been optimized before, in several settings for both conventional and VLIW DSP architectures, a systematic effort to study the benefit of various optimization strategies and the identification of a preferred strategy based on a specific metric across various strategies has not been undertaken in previous work. This paper studies specific implementations of convolution using finite impulse response and infinite impulse response (FIR and IIR) filter along with an associated study of the achieved efficiency to identify the preferred optimization strategies. Development of optimized implementations in several flavors starting with high level languages and culminating in native assembly allows effectiveness of optimizing compilers to be gauged. The systematic methodology followed in this paper clearly identifies loop coalescing along with unroll and jam as the preferred optimization strategy to be used for implementing linear convolution. This is achieved by using a quantitative manner to study the effectiveness of various mappings of the convolution algorithm to the underlying computer architectures. Various example mappings are presented in the paper use the TMS320C62x (VLIW) and the TMS320C64x (VLIW and SIMD) architectures to illustrate the achievable performance on today’s state of the art DSP architectures.

2. Introduction to FIR Filters
FIR filters implement the convolution as a weighted moving average of previous inputs and perform the evaluation of the output samples \( y(n) \) of an \( N \)-tap FIR filter and can be mathematically formulated as follows:

\[
y(i) = \sum_{n=0}^{N-1} x(n-i) h(n) \quad i = 0, 1, \ldots, M - 1
\]

It can then be seen that \( N \) multiplications and \( N - 1 \) additions are needed for each of the \( M \) output samples that need to be computed. The FIR filter allows for a parallel computation of multiple filter taps (inner loop unrolling) and multiple output samples (outer loop unrolling).

3. Introduction to TMS320C6x Architecture
Very Long Instruction Word (VLIW) architectures have proven to be an effective style for leveraging instruction (ILP) and data level parallelism (DLP) at compile time as opposed to having to determine parallelism at run time (superscalar architectures).

![Figure 1: TMS320C64x DSP Architecture.](image-url)
VLIW architectures thus have lower design complexity and achieve higher clock speeds. The C62x architecture is an 8-way VLIW device with two independent clusters (data paths) of computation and four ALU’s per data path. The four ALU’s include a logical unit (L), shifter unit (S), multiply unit (M) and a load-store (D) unit. The C62x architecture has sixteen registers per data path for a total of 32 registers on the device, with a 256-bit instruction fetch bandwidth with a 32-bit op-code and a 64-bit load-store bandwidth allowing for anywhere from one to eight instructions to be issued per cycle. The ability to perform multiple operations allows these devices to use ILP to achieve higher levels of performance. The TMS320C64x (C64x) architecture has 128 bits of load store bandwidth and adds packed data processing to allow these devices to take advantage of both instruction and data level parallelism. The C64x instruction set is a super set of the C62x with the same 8-way VLIW capability. However the ability to exploit these higher levels of performance is dependent on the optimizing power of compilers and the ability of users to expose the inherent parallelism.

4. Terminology for Various Optimized Flavors
The same optimization strategy is applied as a series of steps starting from implementations with high-level languages and culminating in hand-optimized assembly to allow the performance of automatic code generation tools to be evaluated. The natural C (CN) code, represents a “text book” implementation of the algorithm in C code, and is the simplest possible implementation of the algorithm. The optimized C (CO) code represents the first stage of optimization and makes use of compiler "pragma’s" and explicit loop transformations. The intrinsic C (C) code is the next stage of optimization and overlays instructions native to the C6x architecture that cannot be expressed in ANSI C, along with the algorithmic transformations and loads, at the maximum width of the architecture. The partitioned serial assembly (SA) is pipelined independent code written by the user to implement the algorithmic transformations by partitioning the instructions amongst the data units and the registers of the architecture, and makes use of the assembly optimizer to generate parallel optimized assembly code to take advantage of the multiple units on the C6x DSP. Finally hand assembly (ASM) represents the best human effort in writing parallel code for the architecture, taking into account instruction latencies and minimizing register usage.

5. Previous Work on Optimizing Convolution
The convolution algorithm being fairly generic has been implemented in many settings, as it constitutes the heart of linear filtering. Meyer and Schwarz [1] present an extensive review of several strategies that have been previously adopted for implementing convolution on conventional DSP architectures. VLIW based DSP architectures had not achieved widespread popularity at the time Meyer and Schwarz performed this study. Previous efforts have investigated the direct form filtering, fast convolution algorithms using FFT, and radix FIR filter structures. The radix FIR structures are particularly interesting as they strive to minimize the total number of multiplies needed by the algorithm to make it more efficient in terms of execution speed, memory and accuracy. The radix FIR structures use decimated sub-filters each implemented in direct form. In this approach for each output sample, 3N/4 multiplications are required which yields a 25% reduction as compared to the direct method. However this method requires one to pre-calculate the filter coefficients required for (h0 + h1) in advance. In addition, successive multiplies act on input data that is not contiguous, limiting the efficiency of this implementation on SIMD architectures. Hence this paper will not use this method as it is not amenable to VLIW and SIMD architectures.

6. Convolution Algorithm
The code shown in Figure 2 is referred to as the natural C code. This code has two loops, with the outer loop (k) iterating for as many output samples (M) that need to be computed and an inner loop (j) that iterates for as many filter taps (N) per output sample.

```c
for (k = 0; k < M; k++)
{
    xptr = x + k;        hptr = h + (N-1);
    sum = 0;
    for (j = 0; j < N; j++)
    {
        xdata = *xptr++;  hdata = *hptr--;
        sum += (xdata * hdata);
    }
    *yptr++ = ((sum + Qr) >> shift);
}
```

Figure 2: Simplest Implementation or Natural C Code for the FIR Algorithm.

VLIW architectures use a technique referred to as software pipelining which overlaps operations across several iterations to form a modified loop thereby improving the performance that can be achieved. The assembly code shown in Figure 3 constitutes the single cycle kernel (steady state code) of a software pipelined
loop (allowing the FIR computation to be performed in a single cycle in steady state) that is generated by the compiler from the natural C code. The resulting code shows that the assembly code issues only one multiply every cycle although the architecture can issue up to two sixteen bit multiplies per cycle and is clearly far from using the hardware in an optimal manner. The ability to predicate every instruction on both the C62x and the C64x architectures is a critical capability in generating compact code.

L5; PIPED LOOP KERNEL

Following the C64x architectures can perform six 32-bit additions per cycle and two 16-bit multiplies per cycle (C62x) and four 16-bit multiplies per cycle (C64x), the use of outer loop unrolling helps to minimize the load bandwidth and improve the multiplier utilization of the architecture. The convolution algorithm uses a set of nested loops wherein the outer loop iterates once for each output sample M and the inner loop iterates N times for computing the filter taps. Inner/outer loop unrolling does not result in optimized implementation when the outer loop iterates more times in comparison with the inner loop. This condition is referred to as loop unbalance.

The theoretical performance that can be expected from a given architecture can be estimated based on the number of multiplies required to implement the convolution algorithm (because multiplies have the highest request to architectural availability) and the rate at which they can be issued by an architecture (m multiples/cycle). This is true for all filtering algorithms which are computationally bound on the multiplier operation. In particular the C62x and the C64x architectures can perform six 32-bit additions per cycle and two 16-bit multiplies per cycle (C62x) and four 16-bit multiplies per cycle (C64x). Hence this paper defines a new and novel efficiency metric $\varepsilon$ that a given algorithmic transformation realizes as the ratio between the theoretical number of cycles that the algorithm should consume based on multiplies alone (numerator) to the achieved cycles based on a given software pipelined realization (denominator).

$$\varepsilon = \frac{M N}{m} \frac{\left( N l_i + l_o \right)}{\left( \frac{N l_i}{\alpha} + l_o \right)}$$

The ideal aim behind optimizing the code is to reach the point of optimality, which occurs for an implementation when the denominator is exactly equal to the numerator. At this point the architecture’s utilization is maxed out, as it is operating at a point where all the available hardware resources that are needed for computation are used in the fewest cycles. The more realistic aim behind optimization would be to make sure that as the number of output samples and filter taps vary, that one strives to keep the denominator and numerator as close to each other as possible to achieve efficiency metrics that are as close to unity as possible. While this metric will track the efficiency on the developed implementations relative to unity as possible. While this metric will track the efficiency on the developed implementations relative to unity as possible.
to each other (including certain memory effects such as bank conflicts in segmented memory), it does not comprehend higher order system effects such as additional cycles due to program and data cache misses. Thus this metric compares the efficiencies of various implementations assuming the code and data exist in the local memories being directly accessed by the CPU.

In equation (5) M stands for number of iterations of the outer loop (output samples produced), N stands for the number of filter taps to be computed, \( \alpha \) stands for the inner loop unroll factor, \( \underline{\alpha} \) stands for the outer loop unroll factor, \( l_i \) stands for the pipe up and pipe down overhead of the inner loop, \( l_o \) stands for the overhead of the outer loop, \( m \) stands for the number of multiplies that the architecture can issue every cycle and \( k_l \) represents the number of cycles in the software pipelined loop kernel. Optimal efficiency is obtained when the inner loop maximizes the multiplier utilization. Since the outer loop is unrolled by a factor \( \underline{\alpha} \), and the inner loop is unrolled by a factor \( \alpha \), \( \alpha \underline{\alpha} \) multiplies need to be performed by the inner loop, optimality requires that \( l_i = \alpha \underline{\alpha} / m \) in order to maximize the multiplier utilization of the architecture. Hence the formula for the optimum efficiency can now be stated as follows:

\[
\varepsilon_{opt} = 1/1 + (m_l_o / NM) + (m_l_i / N \underline{\alpha})
\]  
(6)

The formula for the optimum efficiency represents the maximum efficiency that can be achieved with inner and outer loop unrolling strategies. It can be seen from the above formula that the inner loop overhead \( l_i \) is scaled by merely the outer loop unroll factor \( \underline{\alpha} \), as opposed to the outer loop overhead \( l_o \), which is scaled by the outer loop iteration count M. Hence the inner loop overhead has a more significant impact on the achievable efficiency than the outer loop overhead. It can now be seen for a given outer loop unroll factor \( \underline{\alpha} \), that DSP architectures that have increased “m” (number of multiplies/cycle), will have a decrease in efficiency. It is precisely this problem that can be overcome by the use of the technique of unroll and jam (presented later).

8. Implementation of Inner/Outer Loop Unrolling

```c
for (k = 0; k < M; k += 2)
{
    xptr = x + k;  hptr = h + (N-2);
    hiptr = (int *) (hptr);
    sum_e = sum_o = 0;
    xdata_e = *xptr++;
    for (j = 0; j < N; j+=2)
    {
        hidata = *hiptr--;
        xdata_o = *xptr++;
        sum_e1 = _mpylh(xdata_e, hidata);
        sum_o1 =  _mpy(xdata_o,      hidata);
        xdata_e =   xdata_o;
        xdata_o =  *xptr++;
        sum_e2 = _mpylh(xdata_e,  hidata);
        sum_o2 = _mpy(xdata_o,      hidata);
        sum_e += sum_e1 + sum_o1;
        sum_o += sum_e2 + sum_o2;
        xdata_e = xdata_o;
    }
    *yptr++ = ((sum_e + Qr) >> shift);
    *yptr++ = ((sum_o + Qr) >> shift);
}
```

**Figure 4:** Intrinsic C code for Inner/Outer Loop Unrolling on C62x DSP.

```c
for (k = 0; k < M; k += 4)
{
    xptr = (double *) (x + k);
    hptr = (double *) (h + (N - 4))
    acc0 = acc1 = acc2 = acc3 = Qr;
    for (j = 0; j < N; j+=4)
    {
        hdword  = *hptr--;
        xword0 = xptr[0];    xword1 = xptr[1];
        xptr   +=       1;
        x3x2 = _hi(xword0);  x1x0 = _lo(xword0);
        x7x6 = _hi(xword1);  x5x4 = _lo(xword1);
        h3h2 = _hi(hdword);  h1h0 = _lo(hdword);
        h2h3_c = h2h3 = _packlh2(h3h2, h3h2);
        h0h1_c = h0h1 = _packlh2(h1h0, h1h0);
        x2x1 = _packhl2(x3x2, x1x0);
        x4x3 = _packhl2(x5x4, x3x2);
        x6x5 = _packhl2(x7x6, x5x4);
        p0 = _dotp2(x1x0, h2h3);
        p1 = _dotp2(x3x2, h0h1);
        p2 = _dotp2(x2x1, h2h3);
        p3 = _dotp2(x4x3, h0h1);
        p4 = _dotp2(x3x2, h2h3_c);
        p5 = _dotp2(x5x4, h0h1);
        p6 = _dotp2(x4x3, h2h3_c);
        p7 = _dotp2(x6x5, h0h1_c);
        tmp0  = p0 + p1;  tmp1  = p2 + p3;
        tmp2 = p4 + p5;  tmp3 = p6 + p7;
        acc0 += tmp0;      acc1 += tmp1;
        acc2 += tmp2;     acc3 += tmp3;
    }
    yptr[0] = (acc0 >> shift);
    yptr[1] = (acc1 >> shift);
    yptr[2] = (acc2 >> shift);
    yptr[3] = (acc3 >> shift);
    yptr += 4;
}
```

**Figure 5:** Intrinsic C code for Inner/Outer Loop Unrolling on C64x DSP.

The intrinsic C code shown in Figures 4 and 5 illustrates the application of inner and outer loop unrolling to C62x and C64x respectively. The use of
split multiplies provides the individual 16-bit inputs for the multiplier from the 32-bit register allows the inputs to be loaded as word-wide quantities in a seamless manner for the C62x DSP. The dot-product instruction allows for two multiplies and the intermediate addition to be performed with a single instruction on the packed sixteen bit inputs in a 32-bit register for the C64x DSP. It can be seen that the inner and outer loop are unrolled deeper to compensate for the longer latencies of the C64x multiply instruction (dotp2) as compared to the C62x multiply instruction (mpy). While inner/outer loop unrolling results in 100% multiplier utilization in the inner loop, the loop unbalance causes a significant loss in efficiency as will be seen from the results.

9. Efficiency Metric Results for Inner/Outer Loop Unrolling on C62x and C64x

Table 1: Inner/Outer Loop Unrolling on C62x DSP

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>CN</th>
<th>CO</th>
<th>C</th>
<th>SA</th>
<th>ASM</th>
<th>THR</th>
<th>EA</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>40</td>
<td>3695</td>
<td>392</td>
<td>1980</td>
<td>1858</td>
<td>1409</td>
<td>1040</td>
<td>0.73</td>
</tr>
<tr>
<td>42</td>
<td>40</td>
<td>2975</td>
<td>2812</td>
<td>1605</td>
<td>1508</td>
<td>1144</td>
<td>840</td>
<td>0.73</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>2255</td>
<td>2132</td>
<td>1230</td>
<td>1158</td>
<td>879</td>
<td>640</td>
<td>0.72</td>
</tr>
<tr>
<td>12</td>
<td>40</td>
<td>815</td>
<td>772</td>
<td>480</td>
<td>458</td>
<td>349</td>
<td>240</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Table 2: Inner/Outer Loop Unrolling on C64x DSP

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>CN</th>
<th>CO</th>
<th>C</th>
<th>SA</th>
<th>ASM</th>
<th>THR</th>
<th>EA</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>40</td>
<td>3695</td>
<td>3632</td>
<td>2025</td>
<td>1928</td>
<td>1564</td>
<td>1260</td>
<td>0.80</td>
</tr>
<tr>
<td>44</td>
<td>40</td>
<td>3223</td>
<td>2608</td>
<td>1335</td>
<td>949</td>
<td>652</td>
<td>440</td>
<td>0.67</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>2255</td>
<td>1978</td>
<td>978</td>
<td>697</td>
<td>478</td>
<td>320</td>
<td>0.66</td>
</tr>
<tr>
<td>20</td>
<td>40</td>
<td>1663</td>
<td>1348</td>
<td>740</td>
<td>529</td>
<td>362</td>
<td>200</td>
<td>0.55</td>
</tr>
</tbody>
</table>

Tables 1 and 2 shows the results of pure inner and outer loop unrolling as the number of output samples and filter taps are varied on the C62x and the C64x DSP. The variation of the efficiency metric for the hand optimized assembly code is computed (as an example) using Equation 5 (ratio of THR/ASM). Efficiencies for other versions of the code can be computed in a similar manner. The results for inner and outer loop unrolling shows that on the C62x, as the number of outer samples varies from [52, 12], a range of 4.33x, efficiency of the optimized code varies over [0.738, 0.687], a range of 1.07x. The efficiency does not change much with the variation of the outer samples as the outer loop overhead $l_o$ is amortized over MN iterations of computation. The variation of filter taps from [60-20] over a range of 3x results in a variation of the efficiency from [0.68-0.46] over a range of 1.48x. This variation of efficiency is seen to be more substantial for the C64x as compared to the C62x, when the number of output samples was varied as the inner loop overhead is amortized over $N^d$ as opposed to the outer loop overhead, which is amortized over NM. This shows that it will become tougher to achieve implementations of the same or higher efficiency on architectures that perform a higher number of multiplies per cycle (m), as outer loop overhead makes a more significant part of the runtime as compared to the core piped loop kernel.

10. Optimality Factor for Unroll and Jam

Loop coalescing (unroll and jam) has been defined as the operation of combining nested loops into a single loop with the original indices computed from the resulting single induction variable [2]. This approach has the benefit of removing the $(l_o/m/N^d)$ term in the efficiency expression. This optimization runs the operations corresponding to the original outer loop once for every complete evaluation of the original inner loop. This imposes additional restrictions on how the various operations can schedule amongst themselves making optimal code generation for this technique by the compiler a more difficult task. A specific output sample $y(n)$ can be computed as follows:

$$y(n) = \sum_{i=n}^{n+N-1} h(N-1-(i\%N))x(i), \ i = 0, 1, ..., M-1$$  \hspace{1cm} (7)$$

It can be seen that the stride index for the input data array, $x$ and the filter array $h$, is obtained from the index of the output sample, $n$. Loop coalescing results in the following efficiency expression:

$$\varepsilon_{opt} = 1/(1 + ml_o/NM)$$  \hspace{1cm} (8)$$

While this method removes the loss in performance
due to the pipe up and pipe down overhead of the inner loop, this optimization results in an increase in the register usage and conditional code to support the resetting of the input and filter pointers. In addition the conditional code creates dependencies between input and output pointers. This is avoided by keeping separate counters that reset input and output pointers to minimize dependencies.

11. Implementation of Loop Coalescing

```c
int itersj = N >> 3;
int itersk = (M - N + 2) >> 1;
k = j = 0;    jc = itersj;
sum_e = sum_o = 0;
for (kj = 0; kj < itersk * itersj; kj++)
{
    if (!j) {xiptr = (int *) (x + k); k += 2;
hptr = (int *) (hrst); j = itersj;}
    j--;
xword0  =  xiptr[0];   xword1  = xiptr[1];
xword2  =  xiptr[2];   xiptr  +=  4;
hidata0 = *hptr--; hidata1 = *hptr--;
sum_e10   = _mpylh(xword0,hidata0);
sum_o10   = _mpyhl(xword0,hidata0);
sum_e11   = _mpylh(xword1,hidata1);
sum_o11   = _mpyhl(xword1,hidata1);
sum_e12   = _mpylh(xword2,hidata2);
sum_o12   = _mpyhl(xword2,hidata2);
sum_e13   = _mpylh(xword3,hidata3);
sum_o13   = _mpyhl(xword3,hidata3);
sum_e20   = _mpyh(xword0, hidata0);
sum_o20   = _mpy(xword1, hidata0);
sum_e21   = _mpyh(xword1, hidata1);
sum_o21   = _mpy(xword2, hidata1);
sum_e22   = _mpyh(xword2, hidata2);
sum_o22   = _mpy(xword3, hidata2);
sum_e23   = _mpyh(xword3, hidata3);
sum_o23   = _mpy(xword4, hidata3);
sum_e += (sum_e10 + sum_o10 + sum_e11 +
        sum_o11 + sum_e12 + sum_o12 + sum_e13 +
        sum_o13);
sum_o += (sum_e20 + sum_o20 + sum_e21 +
        sum_o21 + sum_e22 + sum_o22 + sum_e23 +
        sum_o23);
sum_es = (sum_e >= shift);    sum_os = (sum_o >= shift);
jc--;
if (!jc) {ypt++ = sum_es;
if (!jc) yptr++ = sum_os;
if (!jc) sum_e = sum_o = 0;
if (!jc) jc = itersj;
}
}
```

Figure 6: Intrinsic C code for Loop Coalescing/Unrolling on C62x DSP.

Loop coalescing is also referred to as unroll/jam.

```c
int itersk = (M - N + 1) >> 2;
int itersj = (N >> 3);
k = j = 0;    int jc = itersj;
short nhrst = N >> 4;
if (!j) acc0 = acc1 = acc2 = acc3 = 0;
for (kj = 0; kj < (itersk + 1) * itersj; kj++)
{
    if (!j) {
        xiptr = (double *) (x + k);
hptr = (double *) (hrst);
        k += 2;
        j = itersj;
    }
    j --;
hword0 = *hptr--;    hword1 = *hptr--;    hword2 = *hptr--;    hword3 = *hptr--;
hword0 = *hptr--;    hword1 = *hptr--;    hword2 = *hptr--;    hword3 = *hptr--;
sum_e10   = _mpylh(xword0,hword0);
sum_o10   = _mpyhl(xword0,hword0);
sum_e11   = _mpylh(xword1,hword1);
sum_o11   = _mpyhl(xword1,hword1);
sum_e12   = _mpylh(xword2,hword2);
sum_o12   = _mpyhl(xword2,hword2);
sum_e13   = _mpylh(xword3,hword3);
sum_o13   = _mpyhl(xword3,hword3);
sum_e20   = _mpyh(xword0, hword0);
sum_o20   = _mpy(xword1, hword0);
sum_e21   = _mpyh(xword1, hword1);
sum_o21   = _mpy(xword2, hword1);
sum_e22   = _mpyh(xword2, hword2);
sum_o22   = _mpy(xword3, hword2);
sum_e23   = _mpyh(xword3, hword3);
sum_o23   = _mpy(xword4, hword3);
sum_e += (sum_e20 + sum_o20 + sum_e21 +
        sum_o21 + sum_e22 + sum_o22 + sum_e23 +
        sum_o23);
sum_o += (sum_e20 + sum_o20 + sum_e21 +
        sum_o21 + sum_e22 + sum_o22 + sum_e23 +
        sum_o23);
sum_es = (sum_e >= shift);
sum_os = (sum_o >= shift);
jc--;
if (!jc) {yptr[0] = acc0_s; yptr[1] = acc1_s;
yptr += 4;
acc0 = acc1 = acc2 = acc3 = 0;
    jc = itersj;
}
```

Figure 7: Intrinsic C code for Loop Coalescing/Unrolling on C64x DSP.
Figures 6 and 7 show the application of loop coalescing along with unroll and jam resulting in a combined loop nest where the operations of the original outer loop are run conditionally while those of the previous inner loop are run unconditionally. It can be seen that the code shown here is very similar to that of a pure inner/outer loop unrolling shown earlier except for the fused loop and the predication of several instructions. It can be seen that using separate variables \(j\) to predicate input loads and \(jc\) for output stores cuts down on the memory dependency. The resulting piped loop kernel is not shown here because of space constraints.

12. Efficiency Metric Results for Inner and Outer Loop Unrolling on C62x and C64x DSP

Table 3: Loop Coalescing on C62x DSP

<table>
<thead>
<tr>
<th>Optimization</th>
<th>C62x DSP</th>
<th>C64x DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Unrolling</td>
<td>512 bytes</td>
<td>346 bytes</td>
</tr>
<tr>
<td>Loop Coalescing</td>
<td>800 bytes</td>
<td>746 bytes</td>
</tr>
</tbody>
</table>

In addition it is able to increase instruction level parallelism beyond that allowed by pure inner and outer loop unrolling. However this technique has a drawback in that it increases predication of instructions, code-size and register pressure, number of filter taps and output samples as compared to a pure inner/outer loop unrolling approach. Loop coalescing combined with inner and outer loop unrolling produces a longer piped loop kernel, results in an increase in code size, (and increased register usage) as shown in Table 5. The code-size was measured on the hand-optimized implementations which yielded the smallest code-size. It can be seen that code-size increases by about 1.5-2 times in going to the loop coalescing with loop unrolling approach as compared to the inner and outer loop unrolling. This exact ratio is architecture dependent but this ratio is shown here to gain an understanding of the ratios that can be expected.

13. Direct Form II Factorization for IIR Filters

Infinite impulse response filters are commonly realized by recursively feeding back a weighted sum of past output values and adding this to a weighted sum of the previous and current input values [1]. The major advantage of IIR filters as compared to FIR filters is that shorter length IIR filters can realize a given frequency response as compared to FIR filters. However IIR filters suffer from stability, round-off noise, phase non-linearity issues in addition to the efficiency obtained from this optimization for the C64x is impressive, though not as high as that obtained for the C62x. Although the inner loop unroll amounts are the same, the C64x accomplishes the 16x16 multiplies in half the cycles as that of the C62x, hence the inner/outer loop overhead is amortized over fewer CPU cycles, decreasing efficiency. The fact that the C64x consumes the data faster implies that the outer loop needs to be unrolled further to keep the pipe primed longer. This technique of collapsing both the loops is also able to maintain efficiency better over a wider range of variation in output samples and filter taps as compared to pure inner and outer loop unrolling, thus solving the loop unbalance problem.

Table 5: Comparison of Various Loop Unrolling Strategies

The benefit of this optimization can be seen by the substantially higher efficiency numbers that can be obtained because of a combined loop that does not incur the pipe up or pipe down overhead of the inner loop in Table 3 and 4. A study of the results show that
sensitivity due to coefficient quantization due to fixed point effects. These reasons prompted the derivation of a direct form II realization of an IIR filter which has been shown in literature to have better stability to quantization effects due to fixed point effects. This realization is canonic as it has the fewest adds (4), multiplies (4) and delay elements (2) for each second order section. The individual second order sections in the direct form II realization are also referred to as biquads. The product of N biquad sections implements a transfer function of the following form in the z-domain.

\[ H(z) = K \prod_{i=1}^{N} \left( 1 + \frac{a_{1,n} z^{-1}}{1 + \frac{b_{1,n} z^{-1}}{b_{2,n} z^{-2}}} \right) \]

(9)

The natural C code (CN) for the IIR filter is shown in Figure 8.

```c
void iir_filter_cn(short *x, short *coef, short *history, short *y, int length, int nstages) {
    int i, j;
    short *hist1, *hist2;
    short yb0, ya0;
    short input;
    int output, new_hist;
    for (j = 0; j < length; j++) {
        input = x[j];           hist1  = history;
        hist2  = history + 1;
        output = input * coef[0];
        for (i = 0; i < nstages; i++) {
            yb0 = *hist1;       ya0 = *hist2;
            new_hist = output
                            - (yb0 * coef[4*i + 1])
                            - (ya0 * coef[4*i + 2]);
            output = new_hist
                    + (yb0 * coef[4*i + 3])
                    - (ya0 * coef[4*i + 4]);
            *hist2 = yb0;      *hist1 = new_hist;
            hist2 += 2;      hist1 += 2;
        }
        y[j] = (output + Qr) >> Qpt;
    }
}
```

Figure 8: Natural C Code (CN) for Direct Form II IIR Filter

The code shown above consists of two loops with the outer "j" loop iterating for as many output samples "length" as required and the inner "i" loop for iterating for "nstages" stages of biquads that need to be implemented. It can be seen that within each biquad, two history variables "yb0" and "ya0" are read from history addresses "hist1" and "hist2". It can be seen from Figure 8, that hist2 contains the oldest history and is updated by hist1, while hist1 is updated by the most recent history variable "new_hist". The contribution from the zeroes acts on the most recent history variable "new_hist" to produce the output of the current bi-quad which also forms the input for the next bi-quad.

\[ L5: \quad ; \text{PIPED LOOP KERNEL} \]

\[ \begin{align*}
C0: & \text{SUB} \quad .S1 \quad A8,A7,A4 \quad ; \quad \text{29} \\
    & \text{STH} \quad .D1T1 \quad A5,*A9++(4) \quad ; \quad \text{32} \\
    & \text{MPY} \quad .M2 \quad B7,B9,B8 \quad ; \quad \text{29} \\
    & \text{MPY} \quad .MIX \quad B6,A5,A7 \quad ; \quad \text{29} \\
    & \text{LDH} \quad .D2T2 \quad *-B5(4),B6 \quad ; \quad \text{30} \\
C1: & \text{SUB} \quad .L1X \quad A4,B8,A8 \quad ; \quad \text{29} \\
    & \text{[B0]SUB} \quad .S2 \quad B0,1,B0 \quad ; \quad \text{37} \\
    & \text{LDH} \quad .D1T1 \quad **A1(8),A4 \quad ; \quad \text{30} \\
    & \text{LDH} \quad .D2T2 \quad **B5(2),B7 \quad ; \quad \text{29} \\
C2: & \text{STH} \quad .D1T1 \quad A8,*A0++(4) \quad ; \quad \text{33} \\
    & \text{ADD} \quad .S1 \quad A8,A6,A4 \quad ; \quad \text{30} \\
    & \text{MPY} \quad .MIX \quad A4,B9,A3 \quad ; \quad \text{30} \\
    & \text{[B0]B} \quad .S2 \quad L5 \quad ; \quad \text{37} \\
    & \text{LDH} \quad .D2T2 \quad *B4++(4),B9 \quad ; \quad \text{29} \\
C3: & \text{SUB} \quad .S1 \quad A4,A3,A8 \quad ; \quad \text{30} \\
    & \text{MPY} \quad .MIX \quad B6,A5,A6 \quad ; \quad \text{30} \\
    & \text{LDH} \quad .D1T1 \quad **A0(8),A5 \quad ; \quad \text{29} \\
    & \text{LDH} \quad .D2T2 \quad **B5++(8),B6 \quad ; \quad \text{29} \\
\end{align*} \]

Figure 9: Piped Loop Kernel for Natural C Code for C62x DSP.

Analysis of the assembly code for the natural C code shows that four 16x16 multiplies in four cycles (C0-C3) achieving an average multiplier utilization of one 16x16 multiply/cycle when the multiplier bandwidth of the architecture is two (16x16) multiplies/cycle. Hence it should be possible to speed up the implementation of IIR filters by a factor of 2x in the absence of any other dependencies as C62x architecture can issue a maximum of 2 multiplies/cycle. The efficiency expression for IIR filters can be derived from the previous section noting now that N represents the number of second order sections, M (as before) represents the number of output samples and that four multiplications are required for each of the second order sections to yield Equation 10.

\[ e = \frac{4MN}{M \left( \frac{Nk}{\alpha} + l_i \right) + l_o} \]

(10)
Figure 10: Optimized C Code that Implements Loop Coalescing with Unroll/Jam

The compiler is not able to perform memory alias disambiguation between the history, input and output arrays and hence forces strict memory ordering between loads and stores resulting in a high loop carried dependency bound. This adversely affects the scheduling of the software pipelined loop kernel and results in a 17 cycle loop (not shown here) for computing one bi-quad for two output samples. Even with this, the performance obtained from optimized/intrinsic C code is better than that obtained from natural C code as it does not suffer from the pipe up and pipe down overhead of the nested loop structure used in the natural C code. Serial assembly is used to overcome some of these limitations and use other DSP centric features such as circular buffering for which there is no direct support in high level languages such as C on the TMS320C6x architecture.

15. Serial Assembly Code for Loop Coalescing
Serial assembly allows the users to specify a sequence of assembly instructions and control instruction set selection without having to worry about pipeline dependencies. Details of the instructions are not discussed here but are available in the TMS320C6x instruction set guide [3]. The compiler by way of the assembly optimizer software pipelines the sequence of assembly instructions. Figure 11 shows a serial assembly implementation of the C code shown in Figure 10 which implements loop coalescing with unroll and jam. Due to space limitations only the core inner loop has been shown and the initial setup code has been omitted. Additional modifications include separate read and write pointers to the history buffer as the values that have been just written into the history buffer are not read back again until the computation of the next two output samples. This fact can be used to allow the reads into the history buffer to advance ahead of the stores. The code shown for the serial assembly does not spend any additional cycles resetting the history and the coefficient buffers once for “nstages” iterations of the fused loop as use is made of circular buffering support found on most DSP architectures and on the C62x and C64x DSP. This decreases the number of instructions that need to be predicated and improves the scheduling of the resulting software pipelined loop.

```plaintext
LOOP:

[A_ik] SUB A_ik, 1, A_ik ;

[A_x10, A_k0, A_out1 ;

[X10, A_k0, A_out1 ;

[A_x10, A_out1 ;

[A_hist++] A_hist ++[2], A_yb0 ;

[B_hist++] B_hist ++[2], B_ya0 ;

LDW *A_coef++, A_alpha21 ;

LDW *B_coef++, A_alpha21 ;

MPY A_yb0, A_alpha21, A_out1 ;

MPYLH B_ya0, B_alpha21, A_out2 ;

MPY B_yb0, B_beta21, A_out3 ;

MPYLH B_ya0, B_beta21, A_out3 ;

ADD B_out2, A_out1,, B_out_p0 ;

ADD A_out3, B_out4, B_temp0 ;

SUB B_out0, B_out_p0, B_new_h0 ;

ADD B_new_h0, B_temp0, B_out0 ;

MPY B_new_h0, A_alpha21, B_out5 ;

MPYLH A_yb0, A_alpha21, A_out6 ;

MPY B_new_h0, B_beta21, B_out7 ;

MPYLH B_yb0, B_beta21, B_out7 ;

ADD A_out6, B_out5, A_out_p1 ;

SUB A_out1, A_out_p1, A_new_h1 ;

ADD B_out7, A_out8, A_temp1 ;

ADD A_new_h1, A_temp1, A_out1 ;

STH B_new_h0, *B_hist2_s++[2] ;

STH B_new_h1, *A_hist1_s++[2] ;

ADD B_out0, B_Qr, B_out_r ;

SHR B_out_r, 15, B_out_s ;

ADD A_out1, B_Qr, A_out_r ;

SHR A_out_r, 15, A_out_s ;

[B_ic] SUB B_ic, 1, B_ic ;


MV A_nstages, B_ic ;

A_it, 1, A_it ;

B LOOP ;

Figure 11: Serial Assembly For Loop Coalescing with Unroll/Jam for IIR Filters
```

The software pipelined loop that results from serial assembly of the above loop is shown in Figure 12. This code reveals that ten (16x16) multiplies are performed in 6 cycles numbered C0-C5 achieving an average multiplier utilization of 1.66 multiplies/cycle as compared to the peak multiplier utilization of 2 multiplies/cycle of the C62x architecture. It has been shown as part of the optimized DSP library suite (iir_cas4) from Texas Instruments Inc, that it is possible to obtain 100% multiplier utilization if only rounding operation is not performed on the accumulators. Combining the inner and outer loop unrolling optimization strategy with loop coalescing results in the number of multiplies issued/cycle being smaller than the peak throughput of the architecture but still results in an increase in performance as it prevents the loss due to pipe up and pipe down overhead of the nested loop structure. The individual results for running the IIR filter for different number of bi-quads and different number of output samples are not tabulated because of space limitations.

```plaintext
LOOP:

C0: [ A_it]B .S2 LOOP ;

|| SHR .S1 A_out_r, 15, A_out_s ;

|| MPY .M2 B_new_h0, B_beta21, B_out7 ;

|| ADD .L2X A_out3, B_out4, B_temp0 ;

|| A_ic]SUB .L1 A_ik, 1, A_ik ;

|| MPY .M1 A_yb0, A_alpha21, A_out1 ;

|| LDH .D1T1 *A_hist1++[2], A_yb0 ;

|| LDH .D2T2 *B_hist2++[2], B_ya0 ;

C1: ADD .S2 B_new_h0, B_temp0, B_out0 ;

|| ADD .L1X A_out6, B_out5, A_out_p1 ;

|| A_it]SUB .S1 A_it, 1, A_it ;

|| MPYLH .M2X B_ya0, A_alpha21, B_out2 ;

|| MPYLH .M1 A_yb0, A_alpha21, A_out6 ;
```
16. Conclusions
This paper discussed a systematic way of implementing loop optimization strategies for the convolution algorithm for both FIR and IIR filters and studied them in a quantitative manner. Loop coalescing along with loop unrolling has been demonstrated as a preferred alternative approach to use, instead of the exclusive use of pure inner/outer loop unrolling to enhance performance. This paper shows that this optimization technique will clearly be needed to achieve the architectural performance on future DSP devices that increase the multiplier capability of the architecture. In addition it is recognized that optimizing compilers in future, need to perform more advanced optimizations to achieve performance even on simple DSP algorithms. It can be seen that performance of VLIW architectures on DSP algorithms requires not only the use of the right optimization strategies but also the right combination of instructions to sustain the instruction parallelism amongst available units to achieve high performance implementations. VLIW devices are critically dependent on such high performance implementations as those presented in this paper, no matter what the clock speeds of these devices are. In addition compilers need to start employing these advanced strategies in an automatic fashion, with minimum user guidance as possible, to minimize the complexity in developing high performance software.

17. References
3. TMS320C6000 CPU and Instruction Set, 2000, Reference Guide SPRU189F.
Agenda. mAgic Instruction Level Parallelism DSP Optimized Library Main Optimization Techniques for...Â How to write optimized code for the mAgicV VLIW DSP Elena Pastorelli Atmel Roma elena.pastorelli@atmelroma.it CASTNESSâ€™07. Agenda. mAgic Instruction Level Parallelism DSP Optimized Library Main Optimization Techniques for mAgicV. Uploaded on Oct 06, 2014. Download Presentation. Generation of optimized DSP library for mAgicV VLIW DSP. An Image/Link below is provided (as is) to download presentation. Download Policy: Content on the Website is provided to you AS IS for your information and personal use and may not be sold / licensed / shared on other websites without getting consent from its auth