Background Data Format Optimization for Efficient Sub-Word Parallel Program Generation

P. Op de Beeck†‡, M. Miranda†, F. Catthoor†‡and G. Deconinck‡
†IMEC, Leuven, Belgium
‡ESAT Lab. Katholieke Universiteit Leuven, Leuven, Belgium

Abstract

This paper illustrates the strong interaction between background data format organization and foreground data in the context of speed and power efficient Sub-Word level Parallel (SWP) program generation. Such interaction, if not considered well, results in an (un)packing and reordering overhead that is typically required to match the format of data stored in memory and the one required by the sub-word parallel processing units. We propose a conceptual methodology to minimize this overhead. The approach is demonstrated on two real-life application kernels. A significant reduction in packing instruction overhead, data- (up to a factor 12) and instruction memory accesses (up to a factor 6) is obtained.

1 Introduction

Besides deciding which operations are selected for parallel execution on instruction set processors with support for Sub-Word level Parallel (SWP) processing ([2, 1]), a compiler needs to minimize the (un)packing and reordering operation overhead. These operations are required because firstly the in- and output data to the sub-word operations have to be consumed and produced as full-length words and secondly because the data sub-words in these full-words might need to be reordered.

It is clear that these (un)packing operations strengthen the interaction between the data path and the data organization in background memory and to the best of our knowledge no research has been focused on systematically matching such coupling.

In this paper we show how to systematically analyze this interaction so we can (largely) eliminate the overhead in (un)packing and reordering operations by adapting the data layout in background memory to the data format required by the sub-word level units and stored in the register file. As a result, both packing instruction overhead and data and instruction memory accesses are significantly reduced.

[0]partially supported by FWO-project G.0160.02 ACTMA

```
for(i = 0; i < N; i++) {
    A[2*i] = f(B[i],B[i+32]);
    A[2*i+1] = g(B[i],B[i+32]);

    A[2*i+32] = f(B[i+16],B[i+48]);
    A[2*i+33] = g(B[i+16],B[i+48]);
}
```

Figure 1. A parallelizable loop

Results for the execution time and data memory energy for an optimized application running on a TriMedia TM1300 evaluation board shows at least a factor two gain in execution time and a factor 4 in data memory accesses.

2 Existing compiler techniques for SWP

A closer look at existing compiler-level techniques for SWP processing reveals that on their own they are important steps, but they fail to efficiently map certain real-life computation kernels when background memory issues are incorporated.

Figure 1 shows a piece of code extracted from a Viterbi butterfly (a processing kernel commonly used in wireless applications) to illustrate this. The technique described in [3] would require the i-loop in Figure 1 to be unrolled (twice in the case of a superword size of 4). The pairs *(A[2i],A[2i+1])* and *(A[2i+2],A[2i+3])* would then seed the initial pack list and the packing on *B* would follow through the wish list (see [3] for terminology). However, in some cases, after looking at a more global picture, it is beneficial to pack the subwords *A[2i],A[2i+1],A[2i+32],A[2i+33]* into one superword and this is not considered.

The SIMD code selection of [4] will require even more unrolling in Figure 1 because the adjacency constraint has to be fulfilled for all arrays participating in potential SIMD operations. Hence, also in this technique non-adjacent options are not considered and globally optimal solutions may be overlooked.

The approach in [6] is based on a space-time mapping of uniform recurrence equations. Equation 1 is an example of a non-uniform equation because a reduction operator (i.e. max) is present and thus it needs to be localized.

$$\delta_{bit}(j) = \max_{1 \le i \le N} \delta_{bit-1}(i) \quad (1)$$

However, further analysis reveals that the non-uniformity is only present going from one bit-plane to the next. Inside one bit-plane everything is nicely uniform and enough subword parallelism can be exploited (if $N \geq$ *#subwords*). Therefor an early decision is taken to partition the bit-dimension globally sequential. However, we believe that in the optimal case the space-time mapping and partitioning decisions may be different over different bit-planes. Again the local scope in which the analysis is taking place leads to suboptimal solutions.

In contrast, the goal of this paper is to show the presence of this coupling effect between background memory data layout and SIMD issues, and to propose a conceptual methodology that considers these issues and solves them using program transformations before the detailed compilation steps take place. It is our intention to re-use the existing parallelization techniques (e.g. the space-time mapping parallelization [6] and code selection [4]) as back-end compilation steps. We focus mainly on studying the data format compatibility issues along the global dependency chains of the data involved in SIMD operations.

3 Conceptual method

We use an updating algorithm (Figure 2) to illustrate the interaction between the SIMD compilation steps and the background data format Organisation and describe a conceptual methodology to minimize the overhead in (un)packing and reordering operations.

The update in this case translates into an explicit copy (Figure 2(a), line 12) which results in a feedback loop from array *Fout* to *Fin*. We will show that this feedback loop causes a data format mismatch between *Fout* in the current and *Fin* in the next iteration.

To analyze this mismatch we follow a number a steps:

Collecting data formats For each expression we can, in general, derive different parallel versions resulting in a number of different data formats for each array involved in the expression.

Finding data parallelism in the first loop nest for the expression on line 3 is straight forward. Due to the loop-carried dependency along the i-loop, we choose to parallelize across the j-loop. This results in Figure 2(b) line 20 . In the next loop kernel (line 5) 2 expressions are present. The first expression (line 8) can be parallelized along the i- or the j-loop. For instance for *A* this results in 2 different data formats, *A[j*

```
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        A[i][j] = f(A[i-1][j]);
{...}
while(th > TH) {
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++) {
            Fout[i][j] = h(A[j][i]);
            B[i][j] = g(Fin[i-1][j], Fin[i+1][j]);
        }
    {...}
    memcpy(Fout, Fin, sizeof(Fout));
}
```

(a) Original code

```
for(i = 0; i < N; i++)
    for(j = 0; j < N; j += 4)
        A[i][j : j+3] = f(A[i-1][j : j+3]);
{...}
while(th > TH) {
    for(i = 0; i < N; i += 4)
        for(j = 0; j < N; j++) {
            Fout[i : i+3][j] = h(A[j][i : i+3]);
        }
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j += 4) {
            B[i][j : j+3] = g(Fin[i-1][j : j+3],
                              Fin[i+1][j : j+3]);
        }
    {...}

    /* reorder Fout and copy to Fin */
    for(i = 0; i < N; i += 4)
        for(j = 0; j < N; j += 4) {
            t0 = MLB(Fout[i : i+3][j], Fout[i : i+3][j+1]);
            t1 = MLB(Fout[i : i+3][j+2], Fout[i : i+3][j+3]);
            t2 = MUB(Fout[i : i+3][j], Fout[i : i+3][j+1]);
            t3 = MUB(Fout[i : i+3][j+2], Fout[i : i+3][j+3]);

            Fin[i][j : j+3] = MLW(t0, t1);
            Fin[i+1][j+4 : j+7] = MUW(t0, t1);
            Fin[i+2][j+8 : j+11] = MLW(t2, t3);
            Fin[i+3][j+12 : j+15] = MUW(t2, t3);
}
```

(b) After SIMD parallelism on (a)

```
for(i = 0; i < N; i += 4)i
    for(j = 0; j < N; j += 4) {
        A0 = f(A[i-1][j : j+3]);
        A1 = f(A0);
        A2 = f(A1);
        A3 = f(A2);

        /* reorder A and store in AA*/
        ...
    }
{...}
while(th > TH) {
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j += 4) {
            Fout[i][j : j+3] = h(AA[j : j+3][i]);
            Fin[i][j : j+3] = g(Fin[i-1][j : j+3],
                               Fin[i+1][j : j+3]);
        }
    {...}
    memcpy(Fout, Fin, sizeof(Fout));
}
```

(c) After SIMD parallelism on (a) with a reordering of A

Figure 2. Example of an updating algorithm

: j+3][i] and *A[j][i : i+3]*. For the moment we keep both options.

For the second statement (line 9) we store the data format which results from parallelizing the j-loop.

Group related expressions The next step we collect all expressions which have to take data format constraints from each other, but we must keep control dependence information. This is important because different control paths can benefit from different data formats.

In our example all expressions are kept together in one control dependence graph. Expression 1 and 2 (Figure 2(a), line 3 and 8 respectively) are linked because they both use the *A* array. Expression 2 and 3 (Figure 2(a), line 9) are linked together as well due to the loop carried dependency between *Fout* and *Fin*.

Search for a valid data format strategy Starting from the root, for each expression in the control dependence subgraph we select a data format for each array involved and propagate these decisions to the directly reachable expressions. We continue to select compatible data formats. If at some point this fails we backtrack our previous decisions. In fact we try to reorder the computation such that we find a compatible data format. If this fails we insert data reordering instructions.

If we carry this out in our example we select the only data format for expression 1, namely *A[i][j : j+3]*. Expression two is the next reachable and it is annotated with two possible data formats. We select *A[j][i : i+3]* because it is compatible with our previous decision. This choice also fixes the data format for *Fout*. The third expression has a data format mismatch between *Fout* and *Fin* which cannot be remedied by backtracking previous decisions without introducing data format reordering expressions.

The compiler still has the freedom to choose where to apply the reformatting. Since there is superword level data reuse opportunity [5] which can be exploited in the first loop kernel this is also a good place to reorder the *A* array because the required data for reordering is already present in the register file (Figure 2(c)).

Apply code transformations Finally we perform the loop transformations (i.e. loop unrolling, unroll-and-jam and loop-splitting) resulting from the analysis in the previous step. Also emit the necessary data reordering operations.

4 Results

In this section we provide results before and after (hand) applying our conceptual methodology to two real life kernels within the DAB, namely the OFDM block and the Viterbi convolutional decoder. All versions have been executed on the TM1300@166MHz using the native TriMedia compiler.

Implementation	Exec. time(sec)	# D accesses (10^6)	# I accesses (10^6)
Ref-OFDM	0.94	36.4	32.4
+SIMD	0.11	2.9	7.5
Ref-Viterbi	1.74	121.5	154.4
+SIMD	0.40	16.8	23.7
Ref-DAB	2.98	176.9	230.0
+SIMD	0.77	31.9	71.4

Table 1. Impact of optimized SWP OFDM, Viterbi and DAB

In Table 1 we show the results for the OFDM, Viterbi and full DAB respectively. They show a decrease up to a factor 8 in system cycles and a factor 12 in data memory accesses. Furthermore there is up to a factor 6 reduction in instruction fetches.Finally, due to SWP the full DAB is running in real-time on the Trimedia board.

5 Conclusion

In this paper we show how to explore the organization of the data in background memory to decrease the (un)packing and reordering overhead when generating SWP code for SIMD enhanced instruction set architectures. If this source code transformation is omitted the sub-word level acceleration capabilities provided by the architecture will be sub-optimally exploited and the compiler will produce code with a considerable overhead in (un)packing and reordering instructions. This interaction is illustrated using several real-life demonstrators with substantial savings in both energy and execution time.

References

[1] http://focus.ti.com/docs/prod/folders/print/tms320c64x.html

[2] http://www.trimedia.com/products/briefs/cores.html

[3] S.Larsen, S.Amarasinghe, “Exploiting superword level parallelism with multimedia instruction sets“, *In Conf. Programming Language Design and Implementation*, Vancouver, BC Canada, pp.145-156, June 2000.

[4] R.Leupers and S.Bashford, “Graph based Code Selection Techniques for Embedded Processors”, in ACM Design Automation of Electronic Systems, vol. 5, no. 4, pp. 794-814, October 2000.

[5] J.Shin, J.Chame, M.W.Hall, “Compiler-controlled caching in superword register fi les for multimedia extension architectures“, *In proc. of PACT* , Charlottesville, USA, pp.45-53, September 2002.

[6] R.Schaffer , R. Merker, F.Catthoor, ”Systematic Design of Programs with Sub-Word Level Parallelism”, *In Proc. PARELEC*, Warsaw, September 2002.