Improving Superword Level Parallelism Support in Modern Compilers

Christian Tenllado† tenllado@dacya.ucm.es
Luis Piñuel† lpinuel@dacya.ucm.es
Manuel Prieto† mpmatias@dacya.ucm.es
Francisco Tirado† ptirado@dacya.ucm.es
F. Catthoor‡ catthoor@imec.be

†Dto. Arquitectura de Computadores y Automática
Universidad Complutense
Avd. Complutense s/n, 28040
Madrid, Spain

‡Interuniversity
MicroElectronic Center (IMEC)
Kapeldreef 75, B-3001
Leuven, Belgium

ABSTRACT

Multimedia vector instruction sets are becoming ubiquitous in most of the embedded systems used for multimedia, networking and communications. However, current compiler technology do not allow for an efficient exploitation of the inherent data parallelism available in many signal processing and multimedia applications. In this paper, we have explored the automatic vectorization of embedded applications. In particular, we have focused on algorithms in which the same computations are applied over a set of signals that are being processed simultaneously. Usually this set of signals is represented as a 2D array in which each row is an input signal that has to be filtered in some way. A motivating example, inspired by VoIP processing, illustrates that state-of-the-art vectorizing compilers inefficiently exploit the data parallelism inherent to this kind of applications. One of the main reasons behind this, is that they present inner loops that carry all the dependencies and external loops with strided memory accesses.

We propose a modification of the Superword Level Parallelism (SLP) compiler, proposed in [9], that tries to overcome these problems. Experimental results show that our approach clearly outperforms commercial compilers.

Categories and Subject Descriptors: D.3.4 Compilers: Optimization

General Terms: Algorithms, Performance, Languages.

Keywords: Superword Level Parallelism, FIR, Automatic Vectorization.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

1. INTRODUCTION

Nowadays there is a clear trend towards extending the use of short vector processing in modern architectures. Most of the advanced embedded architectures used for multimedia, networking or signal processing have already added SIMD extensions to their instruction set architectures (ISA). For instance, many Motorola (now Freescale) embedded systems provide AltiVec [4], the ARM11 incorporates its own SIMD ISA [1] and the proposed IBM's Cell [8] architecture includes eight SIMD processing elements connected to a PowerPC core. Despite this architectural trend, and even though some current compilers can generate SIMD instructions, SIMD units are still most effectively exploited by using SIMD data-types explicitly in the source code and restructure algorithms accordingly [5]. Therefore, compilers have to evolve in order to efficiently exploit this parallelism.

The general idea to bridge this gap is to reuse the work done during the 70's and 80's in the context of parallel compilers for shared memory and vector supercomputers. However, those methodologies need to evolve to match the the capabilities of the new technology. Larsen et al. proposed in [9] a novel approach to handle the short vector parallelism inherent to multimedia and scientific applications, which can be considered a first step towards this goal. They showed that the traditional loop parallelism, used in vector machines, can be considered a subset of their Superword Level Parallelism (SLP) approach, which seems to be more appropriate for short vector multimedia extensions.

These new SIMD capabilities help to face the increasing demand on both the features and the performance of new embedded systems for signal processing and multimedia.

We have focused on applications such as VoIP where usually, a single system has to give service to different communication channels, processing several signals simultaneously. This often requires an efficient exploitation of their SIMD capabilities in order to satisfy real time constraints. However, in application like this, inner loops carry all the data dependencies and modern compilers inefficiently exploit SIMD.

In this paper we introduce a methodology that tries to overcome some of these problems. We extend the SLP com-

piler [9], providing it with the capability to efficiently extract the data parallelism available from a external loop, free of dependencies. This technique permits, for some kind of signal and image processing algorithms, an efficient superword construction from data stored in non-adjacent memory addresses.

For evaluation purposes we have used a a set of FIR filters of different lengths. From a compiler perspective, this is not one of the worst scenarios given that our target application only exhibits input dependencies. However, even for this relatively simple case, state-of-the-art vectorizing compilers only achieve moderate speedups. Our methodology clearly outperforms commercial compilers providing an additional 75% speedup on average.

The rest of the paper is organized as follows. Section 2 describes the algorithms that can potentially benefit form the methodology introduced in this paper and illustrates the problems faced by current compilers. In Section 3 we give a brief overview of the original SLP compiler which will be modified in Section 4. The performance of the resultant methodology is evaluated in Section 5 and finally Section 6 summarizes the main conclusions of this work.

2. TARGET ALGORITHMS

In many communication, multimedia or signal processing systems we find algorithms constituted by loop nests that process a set of arrays. Often, inner loops iterate over adjacent memory locations in these arrays, and they are potential candidates for SIMD exploitation using multimedia extensions. However, when these inner loops carry dependencies, current compiling methodologies become inefficient, and only achieve marginal performance benefits.

In this paper, we focus on algorithms where the main loop nest can be split into two nest, denoted as inner nest (IN) and outer nest (ON) respectively, which satisfy the following conditions:

- All the dependencies are carried by the loops in IN. As dependencies we consider flow, anti, output and input dependencies.
- Loops in the IN iterate over the lowest dimension of the arrays, while the innermost loop within ON scans the highest dimension.

2.1 Formal Description

This kind of algorithms can formally be described by extending the definition of spatial locality given by Kandemir [6].

Definition 2.1. A given a loop nest of depth n exhibits spatial locality in the k innermost loops, with respect to a reference R (denoted by an $n \times m$ access matrix A_R) to a m-dimensional array if, for each vector $\bar{g}$ defining the memory layout,

$$\bar{g} \in Ker\{a_i\}, i = n - k + 1, \ldots, n$$

where a_i is the row vector form of the i-th column of A_R.

Based on this definition, our methodology targets algorithms that present spatial locality in all the loops within the IN.

2.2 Evaluation Kernel

For evaluation purposes we have chosen a system that has to process several signals simultaneously. As part of this processing each signal has to be filtered by a linear FIR. For instance, each signal could have been sampled from a different channel, similar to a VoIP algorithm. Often, locality constraints imposed by the following stages in the system impose the samples of each signal to be stored in adjacent memory locations. This is usually achieved by representing the set of signals as a 2D array in which each row represents a signal from a different channel (C/C++ arrays are stored in row-major order).

The algorithm used to filter those signals is described in Figure 1. This is a clear candidate for our methodology:

- Inner loops (j and k loops) scan different columns of the same array row, whereas outer loops (just the i loop in this example) scan different rows
- Data dependencies are only carried by the j an k loops

```
for(i=0;i<CHANNELS;i++)
{
   for (j=0; j<FILTER_LENGTH; j++)
   {
     aux = 0.0;

     for (k=FILTER_LENGTH-j-1; k<=FILTER_LENGTH-1; k++)
      aux = aux + input[i][1+j-FILTER_LENGTH+k]*filter[k];

      output[i][j] = aux;
   }

   for (j=FILTER_LENGTH; j<SAMPLES; j++)
   {
     aux = 0.0;

     for (k=0; k<FILTER_LENGTH; k++)
      aux = aux + input[i][1+j-FILTER_LENGTH+k]*filter[k];

     output[i][j] = aux;
   }
}
```

Figure 1: FIR filter bank applied on several signals from different channels.

The inner loop in the example we have just described is a reduction. This kind of algorithms can be vectorized by modern compilers, as we will show bellow. However, extracting SIMD parallelism from this loop implies a significant overhead due to alignment checking and loop peelings. As an alternative, a conventional vector compiler would suggest loop interchanging [13] to uncover the vector parallelism among rows. However, this is not appropriate given that interchanging causes memory access patterns with poor spatial locality.

More recent techniques, as the SLP approach proposed by Larsen et. al. [9], would unroll the inner loop and try to extract SLP from the basic block generated. The algorithm is then vectorizable at the expense of including several packing instructions that cannot be avoided due to the input dependencies. Moreover, Larsen's compiler cannot extract SLP from the external loop since vectorization is only enabled if packing candidates involve adjacent memory accesses.

Our methodology extends Larsen's algorithm and manages to process several signals in parallel by constructing superwords that combine samples from the different channels.

3. SLP COMPILER OVERVIEW

The methodology proposed in this paper uses some ideas extracted form the original SLP compiler proposed by Larsen et.al. [9]. However, the basic stages in its *core* have been modified in order to enhance the SLP extraction for the algorithms under scope. In this Section we briefly describe the original SLP compiler. It is convenient to remind some definitions introduced in [9]:

Definition 3.2. A Pack is a n-tuple, $\langle s_1, s_2, s_3, \cdots, s_n \rangle$, where $s_1, s_2, s_3, \cdots, s_n$ are independent isomorphic statements in a basic block.

Definition 3.3. A PackSet is a set of Packs.

Definition 3.4. A Pair is a Pack of size two, where the first statement is considered the left statement, and the second statement is considered the right element.

Definition 3.5. The SuperWord Size (sws), is the maximum number of data elements that can be packed in a short vector register on the target platform.

The SLP compiler extracts parallelism from the innermost basic block in a loop nest. To get a vectorizing compiler from it, i.e. to transform the loop parallelism into SLP, it does a pre-processing of each loop nest unrolling the innermost loop by a factor equal to the sws. This unrolling constructs a basic block with several consecutive instances of the same statement. If vector parallelism could be extracted from the original loop, it can now be extracted from the basic block.

The *core* of the SLP compiler is applied later on a three-address representation of the code. It is subdivided into four phases: *Adjacent Memory Identication*, *PackSet Extension*, *Combination* and *Scheduling*, which are described bellow.

In the *Adjacent Memory Identication* stage, the basic block is scanned searching for *Pairs* with adjacent memory references, which are grouped together forming the initial *PackSet*. The *Pairs* in it constitute a seed for the SIMD instructions obtained at the end of the process. Adjacency is determined using both alignment information and array analysis. No *Pairs* are formed that cross alignment boundaries.

As intermediate step, statements can belong simultaneously to two *Pairs* as long as they occupy the left and right positions in the two *Pairs* respectively. This allows the *Combination* stage to easily merge groups into larger clusters. A simple example of the process is described in Figure 2.

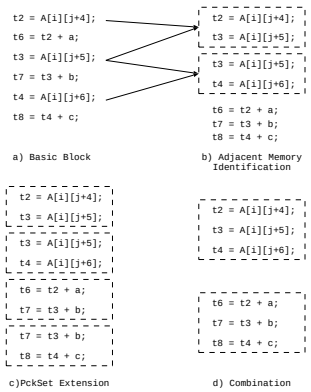

Figure 2: Simple example to illustrate the original SLP methodology.

More *Pairs* are added to the *PackSet* in the next stage. The compiler does it following the *use-def* and *def-use* chains of the *Pairs* that are currently in the *PackSet*. In this way the new members will consume superwords already formed or will provide the ones needed for an existing *Pair*. In all cases, alignment consistency is checked.

Once all the possible candidates have been discovered, the *combination* stage is started. Here two *Pairs* are combined if the right statement of the first *Pair* is the same than the left statement of the other (Figure 2). The combined *Pairs* form a *Pack*. This process continues till no further combination is possible. The alignment consistency guaranties that the *Packs* formed will never cross alignment boundaries and that its size wont be larger than the sws.

Finally the *PackSet* contains *Packs* of statements that have to be executed in parallel, using SIMD computations. It could happen that executing two groups of statements in parallel produces a dependency violation. A dependency cycle among *Packs* indicates that the set of chosen groups is invalid and at least one of the *Packs* has to be eliminated.

After *Scheduling*, every *Pack* in the *PackSet* corresponds to a SIMD instruction plus some possible pack/unpack instructions that could be necessary. The interested reader can refer to [9] for more details.

In our context, the weakest points of the SLP compiler are the following:

- It cannot efficiently extract SLP when dependencies are carried by the inner loop.
- The statement *Packing* is not steered. It could potentially be enhanced if the *Packing* process is performed according to some information extracted from data dependence analysis.
- It does not consider the chance of combining superwords to obtain new seeds for the *Packing* process. The number of *Packs* finally created could be larger if we take this into account.

Some of these points are covered in our methodology for the kind of algorithms under scope.

4. METHODOLOGY DESCRIPTION

We propose some modifications to the SLP compiler aimed to extract the vector parallelism from the external loop. The methodology starts with some loop transformations which expose the vector parallelism from the external loop as SLP in the innermost basic block. Additionally we incorporate some modifications into the Larsen's compiler core that allow us for an efficient exploitation of this SLP. We show that this strategy achieves dramatic speedups for these kind of algorithms.

4.1 Initial Loop Transformations

The loop transformations here presented try to turn the vector parallelism available in the external loop into potential SLP, which can be exploited by a compiler *core* focused on basic blocks. In the following we use loop unrolling factors equal to the sws.

First we perform an *unroll-and-jam* on the deepest loop in ON (Section 2). Our aim in applying this transformation is to uncover the available vector parallelism that can be exploited when different rows are processed concurrently. This transformation is always possible as the loop does not carry dependencies.

When possible, we perform an additional *unroll-and-jam* transformation for every loop in IN but the innermost one. It aims at increasing both the temporal locality and the number of adjacent memory accesses in the innermost basic block. Dependencies has to be checked on each loop to determine if it is available for this transformation.

Finally we unroll the innermost loop. Our aim in applying this unrolling is to create adjacent memory accesses in the innermost basic block.

Let us illustrate by way of example how this process works, using as a case study the FIR algorithm introduced in Figure 1. For the sake of simplicity we suppose a *sws* of 2.

The first *unroll-and-jam* produces a basic block in the innermost loop with accesses to elements from different rows in the input array (see Figure 3). This transformation has been added to extract the available SIMD parallelism using our modified version of the SLP *core*. In our case example, this parallelism comes from the processing of different signals (rows) simultaneously. In the next subsection we will understand how this is possible.

The subsequent step is again an *unroll-and-jam* transformation but on the j-loop (see Figure 4), which augments the basic block of the innermost loop with some statements that perform adjacent memory accesses.

```
for(i=0;i<CHANNELS;i+=2)
{

   ...//First loop not shown

   for (j=FILTER_LENGTH; j<SAMPLES; j++)
   {
       aux_i0 = 0.0;
       aux_i1 = 0.0;

       for (k=0; k<FILTER_LENGTH; k++)
       {
          aux_i0 = aux0 + input[i][1+j-FILTER_LENGTH+k]*filter[k];
          aux_i1 = aux1 + input[i+1][1+j-FILTER_LENGTH+k]*filter[k];
       }

       output[i][j]   = aux0_i0;
       output[i+1][j] = aux1_i1;

   }
}
```

Figure 3: *Unroll-and-jam* on the i-loop in Figure 1 ($sws = 2$).

```
for(i=0;i<CHANNELS;i+=2)
{

   ...//First loop not shown

   for (j=FILTER_LENGTH; j<SAMPLES; j+=2)
   {
       aux_i0_j0 = 0.0;
       aux_i1_j0 = 0.0;

       aux_i0_j1 = 0.0;
       aux_i1_j1 = 0.0;

       for (k=0; k<FILTER_LENGTH; k++)
       {
          aux_i0_j0 = aux0 + input[i][1+j-FILTER_LENGTH+k]*filter[k];
          aux_i1_j0 = aux1 + input[i+1][1+j-FILTER_LENGTH+k]*filter[k];

          aux_i0_j1 = aux0 + input[i][1+j+1-FILTER_LENGTH+k]*filter[k];
          aux_i1_j1 = aux1 + input[i+1][1+j+1-FILTER_LENGTH+k]*filter[k];
       }

       output[i][j]   = aux0_i0_j0;
       output[i+1][j] = aux1_i1_j0;

       output[i][j+1]   = aux0_i0_j1;
       output[i+1][j+1] = aux1_i1_j1;
   }
}
```

Figure 4: *Unroll-and-jam* on the j-loop in Figure 1 ($sws = 2$).

After applying these *unroll-and-jams*, the inner loop is unrolled producing the code shown in Figure 5. This final transformation adds only one new adjacent memory access. The reason is that some of the elements of the input matrix accessed by the new statements, are also accessed by the statement instances obtained by the previous j-loop unrolling (marked as bold in Figure 5). This translates to an improvement in temporal locality, which has been the main motivation behind the traditional unroll-and-jam transformation. This enhancement has shown to be also very effective in a SLP context [11]. We should highlight that the *unroll-and-jam* of the i-loop does not affect the temporal locality as it does not carry dependencies. It generates extra isomorphic statements that perform memory accesses on different rows of the arrays.

```
for(i=0;i<CHANNELS;i+=2)
{

   ...//First loop not shown

   for (j=FILTER_LENGTH; j<SAMPLES; j+=2)
   {
       aux_i0_j0 = 0.0;
       aux_i1_j0 = 0.0;

       aux_i0_j1 = 0.0;
       aux_i1_j1 = 0.0;

       for (k=0; k<FILTER_LENGTH; k++)
       {
          aux_i0_j0 = aux0 + input[i][1+j-FILTER_LENGTH+k]*filter[k];
          aux_i1_j0 = aux1 + input[i+1][1+j-FILTER_LENGTH+k]*filter[k];

          aux_i0_j1 = aux0 + input[i][2+j-FILTER_LENGTH+k]*filter[k];
          aux_i1_j1 = aux1 + input[i+1][2+j-FILTER_LENGTH+k]*filter[k];

          aux_i0_j0 = aux0 + input[i][2+j-FILTER_LENGTH+k]*filter[k+1];
          aux_i1_j0 = aux1 + input[i+1][2+j-FILTER_LENGTH+k]*filter[k+1];

          aux_i0_j1 = aux0 + input[i][3+j-FILTER_LENGTH+k]*filter[k+1];
          aux_i1_j1 = aux1 + input[i+1][3+j-FILTER_LENGTH+k]*filter[k+1];
       }

       output[i][j]   = aux0_i0_j0;
       output[i+1][j] = aux1_i1_j0;

       output[i][j+1]   = aux0_i0_j1;
       output[i+1][j+1] = aux1_i1_j1;
   }
}
```

Figure 5: *Unrolling* on the k-loop in Figure 1 ($sws = 2$).

After these additional loop transformations and some intermediate stages described in [9], the modified core of the SLP compiler can be applied. To improve the SLP exploitation, we should add to these intermediate steps the loop peeling and dynamic alignment detection techniques described in [10, 2, 7].

4.2 Modifications to the SLP compiler core

As we have explained in the previous Section, one of the keys to success of the SLP algorithm is its ability to seed an initial *PackSet* with pairs of statements that imply accesses to adjacent memory locations. This translates to a reduction in the number of load instructions and enables the compiler to find vector candidates that are already packed in memory. This adjacent memory accesses can also be found in the basic block generated by the process described in Figures 3, 4 and 5. Thus we do not modify the first phase.

However, we modify the original ordering performing the *combination* stage just after the *adjacent memory identification*. In this way, at the end of the *combination* phase the compiler has a *PackSet* ($\mathcal{P}_0$) that only contains *Packs* of statements that access adjacent memory locations. In the code generation phase, each of these *Packs* can be translated to a vector load. For the same reasons as in the original SLP compiler algorithm, these *Packs* are guaranteed to be less than or equal than the *sws* and will not cross alignment boundaries.

As a result of the loop transformations performed, we have in the basic block sets with *sws* equivalent *Packs*. These sets contain statements that perform the same computations on different rows of the arrays (Figure 6b). We will refer to a set of these equivalent *Packs* as a *Group*.

Each *Group* can be seen as a $sws \times sws$ matrix. We introduce a new *Pack Transposition* phase, in order to pack together statements that operate on data elements from different rows. It consist in transposing each of the *Groups* in the *PackSet*. The process is described in Figure 6 for the final basic block in Figure 5. A new *PackSet* ($\mathcal{P}_1$) is constructed from the transposition result (Figure 6c).

In the code generation phase, this *Pack Transposition* translates to a set of shuffling operations that transform each

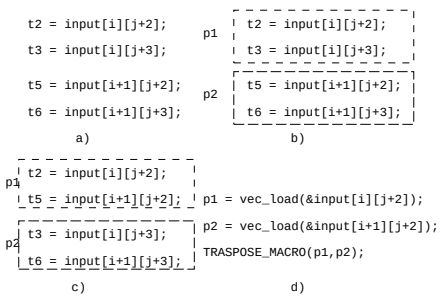

Figure 6: An example of the *Pack Transposition* phase and the respective vector code generation for the example in Figure 5 ($sws = 2$).

of the superwords processed by a *Group* in $\mathcal{P}_0$ to the corresponding superwords consumed by the new *Packs* in $\mathcal{P}_1$ (Figure 6d). As we will show, the shuffling overhead is by far compensated by an increase in the number of short vector instructions finally generated.

This process can be done in the context of the SLP compiler algorithm as multidimensional arrays are padded in the lowest dimension, which guaranties constant alignment among rows [9]. This new stage is an example of how the packing of elements can be steered according to a dependence analysis. Notice that this stage combines superwods that were found in memory making new superwords, that will be better consumed in the final SIMD algorithm. Dependencies gave us the information for this transformation.

Finally, the new *PackSet* ($\mathcal{P}_1$) is considered as seed for the *PackSet Extension* phase of the original SLP compiler algorithm, with one slight difference, the *PackSet Extension* phase has to work on *Packs* not on *Pairs* of statements. No modifications are done on the *scheduling* stage.

5. EXPERIMENTAL RESULTS

For evaluation purposes, our Pack Transposition SLP (PT-SLP) methodology has been manually applied on the algorithm shown in Figure 1. We have performed all the experiments on a Intel Pentium processor (512KB L2 cache, 8K L1 data cache) running Linux. We have chosen this platform given that it provides both a state-of-the-art short vector instruction set (Intel's SSE/SSE2 [12]) and a state-of-the-art commercial vector compiler (Intel C/C++ compiler version 8.1 [3]), which can also be used used as back-end for our methodology.

In all cases, we have done the same experiments modifying both the filter length and the array sizes. For the sake of simplicity we have used square arrays.

We have used as a reference the speedups achieved by a commercial compiler such as the Intel C/C++ compiler. Figure 7 shows the speedups achieved over the original scalar code introduced in Figure 1. As the Larsen's SLP compiler, it only extracts vector parallelism from the inner loop.

The theoretical maximum speedup is the sws, i.e. 4 in our case. In absence of real data dependencies carried by the innermost loop, the vector extraction from this loop becomes effective for large filters (large number of iterations in the loop).

Let us see which is the effect of applying instead our methodology on the original scalar code. Figure 8 shows impressive speedups. However, we are performing several optimizations in addition to a better SLP exploitation as

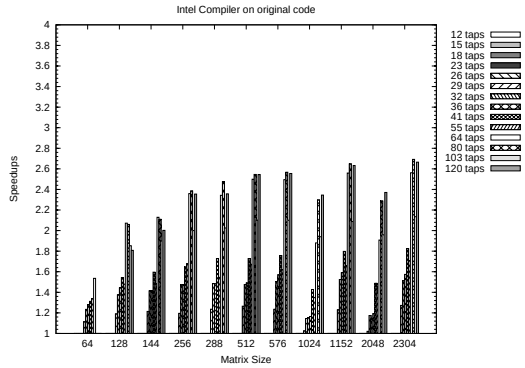

Figure 7: Speedups achieved by the automatic vectorization of the Intel compiler over the original scalar code.

lateral effect of applying our methodology. As in the reference case, performance is better for large filters.

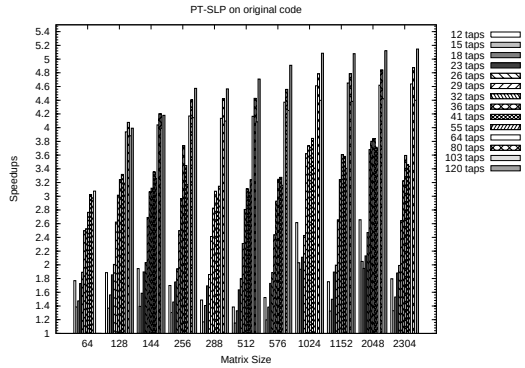

Figure 8: Speedups achieved by the Pack-Transposition SLP (PT-SLP) over the original scalar code.

We have analyzed step by step the methodology in order to isolate the SLP performance improvements from the rest of the optimizations. From this analysis we have concluded that the *unroll-and-jam* transformation made on the external loop provides important improvements on the execution time of the program, which are summarized in Figure 9. As can be noticed, the speedups are significant and for small sizes they are even larger than those obtained by Intel's automatic vectorization. Furthermore, we should highlight that these benefits are independent of the array size.

The effect of the additional SLP extraction enabled by our methodology can be isolated if we take as baseline scalar code a tuned version in which this *unroll-and-jam* transformation on the i-loop has already been performed. We repeat the experiments using this new baseline code to analyze only the benefits of the additional SLP extraction.

Results are shown in Figures 10 and 11 respectively. Whereas the Intel compiler hardly achieves any improvements, *Pack Transposition* is able to efficiently extract parallelism from the external loop. The speedups are around 2 for a large range of arrays and filter lengths, outperforming the Intel compiler by 70% (on average).

One of the keys to success is that the technique loads superwords already packed in memory and systematically reorder them in the register file. The overhead introduced

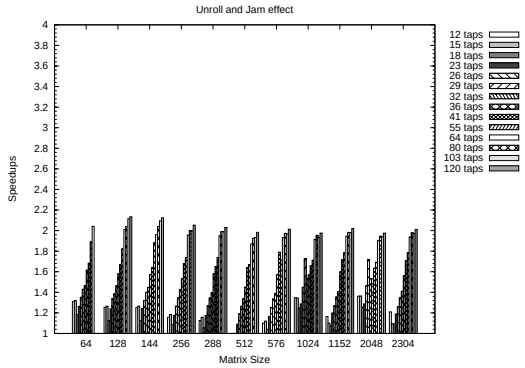

Figure 9: Speedup achieved by applying the *unroll-and-jam* on the external loop.

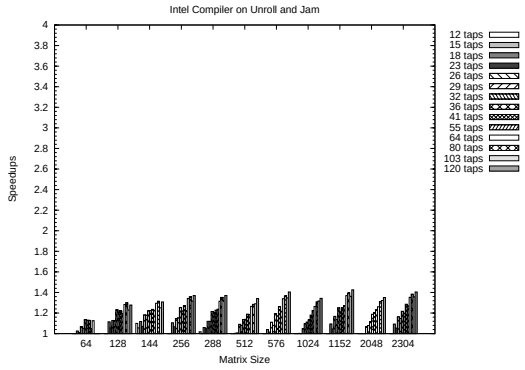

Figure 10: Speedup achieved the automatic vectorization of the Intel compiler over a tuned scalar code where an *unroll-and-jam* transformation applied on the external loop.

by the extra shuffling operations is by far compensated by the additional vector computations that can be generated from the new superwords. Finally, we should remark that, far from adversely affecting the performance for the scalar versions, the loop transformations needed to convert vector parallelism in the outer loop into potential SLP entail extra benefits.

6. CONCLUSIONS

In this paper we have presented a novel methodology to efficiently exploit short vector parallelism from the external loop in loop nests that process 2D arrays. These algorithms are extensively used in modern multimedia, networking or signal processing embedded systems.

The methodology consists in a modification of the SLP compiler presented in [9]. Some loop transformations are used to construct a basic block adequate to the modified SLP *core*. These transformations turn the vector parallelism of the external loop into potential SLP. The core is instructed to efficiently recombine groups of superwords into new superwords that contain elements from different rows of the array. Results show that the overhead introduced by this systematic recombination is by far compensated by the extra SLP exploited.

The methodology has been evaluated on a set of FIR filters independently applied to several signals. In a future work we plan to extend the evaluation to a larger set of applications. Given that our long term goal is to develop

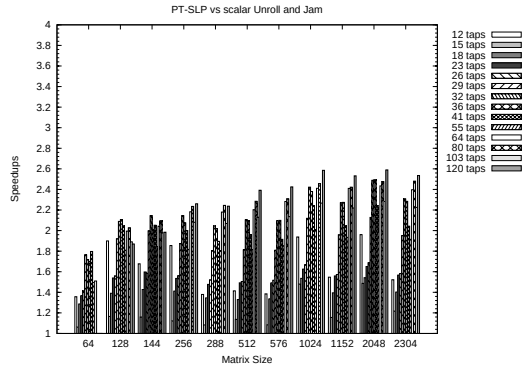

Figure 11: Speedups achieved by PT-SLP over a tuned scalar code where an *unroll-and-jam* transformation has been applied to the external loop

platform independent tools, we also plan to evaluate our methodology on different computing platforms in order to analyze the impact of some architectural parameters and model them accordingly. This study can also be extended by the analysis of different kinds of algorithms that would require other superword combinations.

7. ACKNOWLEDGMENTS

This work is supported by the Spanish Government Research Contract TIC2002-750 and the HiPEAC European Network of Excellence. Christian Tenllando was also supported by the Marie Curie Fellowship of the European Community. We also thank the anonymous reviewers of ISSS-CODES'05 for their helpful comments.

8. REFERENCES

[1] Arm11 family. http://www.arm.com/products/CPUs/families/ARM11Family.html.

[2] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems. *Intel Technology Journal*, 2001.

[3] I. Corpation. Intel c/c++ and intel fortran compilers for linux. Available at http://www.intel.com/software/products/compilers.

[4] S. Fuller. Motorola's AltiVec technology. Technical Report ALTIVECWP/D, MOTOROLA, 1998.

[5] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA*, pages 258–262, 2005.

[6] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, February 1999.

[7] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. Journal on Parallel Programing*, 28(4), 2000.

[8] K. Krewell. Cell moves into the limelight. *Microprocessor Report*, (2/14/05-01), February 2005.

[9] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):145–156, 2000.

[10] S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, MIT, USA, 2001.

[11] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Int. Conf. on Parallel Architectures and Compiler Techniques*, pages 45–55, 2002.

[12] S. T. Thakkar and T. Huff. Internet streaming simd extensions. *Computer*, 32(12):26–34, 1999.

[13] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Massachusetts, USA, 1991.