Source-to-Source Transformations for Efficient SIMD Code Generation

Alejandro Berna¹, Marta Jiménez¹, Jose M. Llaberia¹

Abstract

In the last years, there has been much effort in commercial compilers to generate efficient SIMD instructions-based code sequences from conventional sequential programs. However, the small numbers of compilers that can automatically use these instructions achieve in most cases unsatisfactory results. Therefore, the code often has to be written manually in assembly language or using compiler built-in functions to achieve high performance. In this work, we present source-to-source transformations that help commercial vectorizing compilers to generate efficient SIMD code. Experimental results show that excellent performance can be achieved. In particular, for the problem of the problem of matrix product (SGEMM) we almost achieve as high performance as hand-optimized numerical libraries. Our source-to-source transformations are based on the scalar replacement and unroll and jam transformations presented by Callahan et all. In particular, we extend the use of scalar replacement to vectorial replacement and combine this transformation with unroll and jam and outer loop vectorization to fully exploit the vector register level and thus to help the compiler to generate efficient SIMD code. We will show experimentally the effectiveness of our proposal.

Categories and Subject Descriptors

D.3.4 [Processors]: compilers, optimization; C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]: Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms

Algorithms, Performance.

Keywords

SIMD; vectorization; source-to-source transformations; register tiling.

1. Introduction

The ISA of all today’s microprocessors has been extended with multimedia instructions [9]. Multimedia extensions follow the SIMD paradigm by exploiting wide data paths and functional units that simultaneously operate on narrow data paths of packed data elements (relatively short vectors that reside in memory or registers). The number of packed data elements (VL) supported by the SIMD instructions has been increased with each microprocessor generation, going from 64 bits data registers in the Pentium II with the MMX technology to the 256 bits data registers in Sandy Bridge with the AVX1 technology. Moreover, SIMD extensions have also evolved in number of instructions and data types. MMX technology has 57 SIMD instructions and handles only integer data types while AVX1 technology has hundreds of instructions and handles both integer and floating-point (single and double) data types[12][20].

SIMD instructions are useful in multimedia and signal processing applications [23][30], but also in scientific and numerical applications [1][8][11]. They offer deeper parallelism, good performance/power ratio, and better resource utilization. However, compilers still do not have good support for SIMD instructions due to the difficulty of automatically vectorizing conventional sequential programs. The few commercial compilers that can automatically use these instructions achieve in most cases unsatisfactory results.

To overcome the lack of adequate compiler support for SIMD extensions, often the code has to be written manually in assembly language or using compiler built-in functions [12]. However, these methods, although very effective, are tedious, error prone and result in highly machine-specific code, so that porting an application to a new target processor requires significant programming effort.

Manufacturers have tried to minimize the complexity of writing SIMD optimized codes by providing numerical libraries (such as MKL [11]) that attain high performance under their particular microprocessor. However, not all applications can take advantage of these libraries and there are many situations in which none of the routines provided can specifically solve the task at hand.

We believe that restructuring a code to better exploit SIMD capabilities should be the job of a compiler. Compilers, not programmers, should handle the machine-specific details required to obtain high performance on each particular architecture. Algorithm should be expressed in a natural machine-independent form and the compiler should apply the appropriate transformation to optimize the resulting code.

In this paper, we present high level (source-to-source) transformations that help actual commercial vectorizing compilers to generate efficient SIMD code on scientific numerical applications. The proposed transformations are simple enough to be suitable for automatic implementation by compilers.

Our proposal is based on an effective use of the vector registers. As already known, the existence of a gap between memory and CPU performance made effective use of the register file imperative for excellent performance. It is well-known that the allocation of array values that exhibit reuse to registers can significantly improve the memory performance of programs. However, in many production compilers array references are left as references to main memory rather than references to registers because the data flow analysis used by the compiler is not powerful enough to recognize most opportunities for reuse in subscripted variables. Callahan et all in [5] presented a source-to-source transformation, called scalar replacement, that exposed the reuse available in array references in an innermost loop. They also showed experimentally how another loop transformation called unroll and

¹Departament d’Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain, e-mail: {aberna, marta, llaberia}@ac.upc.edu
jam, could expose more opportunities for scalar replacement by moving reuse across an outer loop into the innermost loop. In this work, we will apply the idea of scalar replacement and unroll and jam to vectorized loop nests and show experimentally their effectiveness. We refer as vectorial replacement to the scalar replacement transformation applied to SIMD vectorized loop nests.

Summarizing, the contribution of this paper are the following:

- An approach that combines 3 source-to-source transformations (outer-loop vectorization, unroll and jam of vectorized loops and vectorial replacement) that help compilers to generate efficient SIMD code in scientific numerical applications.
- Experimental evaluation exhibiting the impact of these transformations using simple kernels of loop nests on a Nehalem platform.

The rest of this paper is organized as follows: Section 2 explains previous work related to source-to-source loop transformations. Section 3 describes our approach to help the compiler to vectorize outer loops and to apply unroll and jam and vectorial transformations. Section 4 gives an extended example using matrix product kernel. In Section 5 we show performance results of our approach compared to scalar version, inner-loop vectorized versions and vendor supplied numerical libraries. Finally, Section 6 concludes.

2. Related Work

Little published work exists which directly deals with high level code transformation techniques for processors with SIMD capabilities. Several researchers [3][7][15][19][21][24] have worked on vectorizing compilers, but not on high level (source-to-source) code transformations to help compilers to generate efficient SIMD codes. These researchers focus on automatically identifying vectorizable section of code and generate appropriate SIMD instructions. Their proposals are low level optimizations to be implemented inside compilers. Our work instead proposes high level transformations for generating efficient SIMD code while waiting for commercial compilers to implement novel approaches from previous researchers.

Moreover, most of these auto-vectorization approaches focus on innermost loops [7][15][24] or block vectorization [4]. Only Nuzman et al in [19] deals with outer loop vectorization and show its effectiveness. Their proposal consists on implementing in-place outer loop vectorization inside the GCC compiler. In contrast, we perform outer loop vectorization as a high level (source-to-source) transformation.

Additionally, Callahan et al in [5] presented a source-to-source transformation, called scalar replacement, that exposes the reuse available in array references in an innermost loop. They also showed experimentally how another loop transformation, called unroll and jam, could expose more opportunities for scalar replacement by moving reuse across an outer loop into the innermost loop. In our work, we extend the use of scalar replacement and unroll and jam to SIMD vectorized loop nests and show experimentally their effectiveness. We do not know any previous work that extends these techniques for SIMD codes.

Finally, there exist several hand-coded numerical libraries optimized for SIMD processors [11][25] that achieve very high performance for some particular class of microprocessors and for some particular functions. However, as already mentioned, not all applications can take advantage of these libraries and there are many situations in which none of the routines provided can specifically solve the task at hand. Our techniques, instead, can be applied to more general codes.

3. Source-to-Source Code Transformations

Our approach to combine source-to-source transformations proposed in this work are based on three observations. First, we observe that commercial compilers only perform inner loop vectorization. However, in most codes it is necessary to vectorize outer loops to achieve high performance.

Second, we observe that compilers are not able to unroll and jam loops with non unit stride. As we will see later, optimizing transformations like register tiling [6][13][14] requires inner loops to be fully unrolled. Therefore, when combining register tiling with vectorization it sometimes becomes necessary to fully unroll strip-mined (non-unit stride) loops and jam together the inner (vector) loops.

Third, we observe that compilers are not able to allocate adjacent array values to vector registers and exploit the reuse available in array references in an innermost loop. However, it is well-known that the allocation of array values that exhibit reuse to registers can significantly improve the memory performance of programs [6][13][14].

In the next subsections we show how we solve these three compilers limitations by applying source-to-source transformations. For the rest of this section and for simplicity, we assume that loop nests are fully permutable and perfectly nested, and loop bounds are constants. For handle more general loop bounds that are max or min functions of surrounding loop iteration variables, we would need to use the theory of unimodular transformations when performing loop permutation [16] and Index Set Splitting [29] for making sure that a particular loop perform a constant number of iterations.

We also assume that previous analysis to decide which loops should and could be vectorized has already been performed. This paper only focuses on the code generation phase of source-to-source transformations. Dependence and decision analysis to know if transformations are legal and to decide which loop is the best candidate to vectorize are out of the scope of this paper [2][17][22][23][26][27].

3.1 Outer Loop Vectorization

Let consider the following loop nest:

```
for (i=Li; i<Ui; i++)
  for (j=Lj; j<Uj; j++)
    for (k=Lk; k<Uk; k++)
      F(i,j,k)
```

and assume that loop \( i \) should be vectorized.

Outer-loop vectorization can be implemented by combining two well-known transformations: strip-mining and loop permutation. Strip-mining is used to partition one dimension of the iteration space into strips and loop permutation is a unimodular transformation [29] used to establish a new order of the loops in a nest.

Strip-mining decomposes a single loop into two nested loops; the outer loop steps between strips of consecutive iterations, and the inner loop (element loop) traverses the iterations within a strip. The loop bounds after strip-mining a loop are directly obtained by applying the following formula (assuming \( U \) is multiple of \( S \)):
for (i=L; i<U; i=i+S)
for (j=0; j<dimj; j++)
for (i=0; i<dimi; i++)
long int i, j;
int dimi, int dimj;

}  

*We use pragmas in the codes to force vectorization (see section 5).*
### 3.3 Vectorial Replacement

Vectorial replacement (VR) can be used to eliminate redundant vector loads and stores in the loop body. Most compilers fail to recognize even simplest opportunities for reuse of subscripted variables between iterations of the innermost loop. This happens in spite of the fact that standard optimization techniques are able to determine that the addresses of the subscripted variables are invariant in the inner loop. The principal reason for the problem is that the data-flow analysis used by standard compilers is not invariant in the inner loop. The principal reason for the problem is that the data-flow analysis used by standard compilers is not powerful enough to recognize most opportunities for reuse of array variables.

Scalar replacement, proposed by [5][6], is a source-to-source transformation that uses dependence information to find reuse of array values and expose it by replacing the references with scalar temporal variables.

We apply the idea of scalar replacement to vectors to help the compiler to eliminate redundant vector loads and stores in the innermost loop. For that, we identify individual array references with array variables and expose vector register reuse in the source code. In particular, for each invariant vectorized reference, we create a new temporary array variable of dimension VL. Then we replace each invariant vectorized reference by the new temporary array variable.

When using temporary pointers variables instead of arrays for the implementation of vectorial replacement.

### 4. Matrix product example

This section shows how efficient SIMD code can be obtained by applying all the transformations explained in section 3 to the register tiled matrix product (SGEMM).

First of all, we compiled the original matrix product, shown in Figure 6, using icc with all compiler optimizations (including vectorization) turned on. It can be seen that icc always permutes loads and stores from the loop body. In Figure 4, reference A[vi] and A[vi+VL+vi] are loaded and stored only once during the execution of the j-loop.

Finally, in Figure 5 we show the same example as Figure 4 but using temporary pointers variables instead of arrays for the implementation of vectorial replacement.

### Figures

Figure 3. Cross addition after performing outer-loop vectorization and unroll and jam to loop i. The left column shows the source code and the right the assembly code.

Figure 4. Cross addition after performing outer-loop vectorization, unroll and jam and vectorial replacement using temporary vectors variables. The left column shows the source code and the right the assembly code.

Figure 5. Cross addition after performing outer-loop vectorization, unroll&jam and VR using temporary pointers. The left column shows the source code and the right the assembly code.
The loop nest (no matters which is the original loop order) making loop j the innermost loop. Since icc only performs inner loop vectorization, this loop order allows icc to vectorize loop j. Moreover, loop j is unrolled by a factor of 8 (2 vectors). Finally, icc also exploits the reuse of the invariant reference of matrix A in the inner loop j by loading it only once in a vector register during the execution of loop j.

Our objective in this section is to generate an efficient code that fully exploits the register level of the memory hierarchy and the SIMD capabilities of the target machine. To this end, we first apply register tiling to the source code as shown in Figure 7a. BI and BJ are the tile sizes in dimension i and j, respectively, and their values depend on the available SIMD registers and their sizes on the target architecture. For simplicity and without loss of generalization, we assume dimi and dimj to be multiple of BI and BJ, respectively.

It is well-known that loop tiling is one of the most important loop transformations. A compiler can use this loop transformation to automatically create block algorithms. The advantage of block algorithms is that, while computing within a block, there is a high degree of data locality, allowing better register, cache or memory hierarchy performance. Loop tiling for the register level requires an extra phase not needed for other memory levels. Since registers are only addressable using the register number, it is necessary to fully unroll the loops that traverse the iterations inside the register tile (loop and j in Figure 7b). To fully unroll the strip-mined loop j we need to apply register tiling for the register level.

At this point icc vectorizes dimension j keeping loop k as innermost loop. However, icc does not remove redundant vector loads and stores from the new unrolled loop body. As we can see in Figure 8c, the elements of C are loaded and stored in each iteration of loop k unnecessarily. Therefore we need to apply vctorial replacement to reference C as explained in section 3.3.

Summarizing, by combining register tiling with the source-to-source transformations proposed in Section 3, we help icc compiler to generate efficient code that fully exploit the register level and the SIMD capabilities of the target machine.

5. Performance Results

First details of our evaluation environment are presented including a description of the architecture, compiler and kernels used. Then, kernel performance is described and analyzed.

5.1 Evaluation environment

All kernels in this study have been executed in the same machine and compiled by the same version of the icc with the same flags and options.

### Target architecture

The machine used for this work is the Intel Xeon E5520 which implements the Intel Nehalem architecture with 4 cores. Since we are evaluating single core executions, we only use one of the four available cores. The SIMD capabilities of these cores include from MMX and SSE to SSE4 instructions being SSE3 the most important for our purposes. The memory hierarchy characteristics offered by this machine are listed in Table 1.

This machine also provides CPU throttling and automatic prefetcher capabilities which have been disabled to prevent interactions with the performance measures. In the same way, we always execute an infinite loop on the 3 cores where our kernels are not running.
long int ii, jj, k, vj;
for (ii = 0; ii < dimi; ii++)
#pragma ivdep
for (vj = 0; vj < dimj; vj++)
C2[vj] += A2[k]*B1[vj];
C3[vj] += A1[k]*B2[vj];
for(vj = 0; vj<VL; vj++)
#pragma ivdep
B2 = &B[k*dimj+jj+VL];
B1 = &B[k*dimj+jj];
for (k = 0; k < dimk; k++) {
C4 = &C[(ii+1)*dimj+jj+4];
C3 = &C[ii*dimj+jj+4];
C2 = &C[(ii+1)*dimj+jj];
C1 = &C[ii*dimj+jj];
for (jj = 0; jj < dimj; jj+=2*VL) {
A2 = &A[(ii+1)*dimk];
A1 = &A[ii*dimk];
float *C1, *C2, *C3, *C4;
}
for(vj=jj; vj<jj+VL;vj++) {
#pragma ivdep
for (k = 0; k < dimk; k++)
for (jj = 0; jj < dimj; jj+=2*VL)
C[(ii+1)*dimj+vj+VL] += A[(ii+1)*dimk+k]*B[k*dimj+vj+VL];
}

long int ii, jj, k, vj;
float *C1, *C2, *C3, *C4;
for (ii = 0; ii < dimi; ii++)
A1 = &A[i*dimk];
A2 = &A[(ii+1)*dimk];
for (jj = 0; jj < dimj; jj++)
C1 = &C[i*dimj+jj];
C2 = &C[(ii+1)*dimj+jj];
C3 = &C[i*dimj+jj+VL];
C4 = &C[(ii+1)*dimj+jj+VL);
#pragma ivdep
for(vj=0; vj<VL; vj++)
C2[vj] += A2[k]*B1[vj];
C3[vj] += A1[k]*B2[vj];
for(vj = 0; vj<VL; vj++)
#pragma ivdep
for (vj = 0; vj < dimj; vj++)
C2[vj] += A2[k]*B1[vj];
C3[vj] += A1[k]*B2[vj];

5.2 Performance Results
In this section we will present the performance results obtained by our source-to-source transformations. To this end, we evaluate four different versions of each kernel: one is the original version (ORI) with no previously restructuring transformation, a second one generated after optimizing the ORI version for scalar execution (Scalar), a third one generated after applying outer-loop vectorization and unroll and jam to the original source code (SIMD), and the fourth one generated after applying all three transformations (outer loop vectorization, unroll and jam and vectorial replacement) to the original code (SIMD+VR). After generating the different versions for each program, we use the icc compiler as mentioned previously to generate the final executables.

Figure 10a shows the performance obtained on the Nehalem architecture for the cross addition kernel. In the ORI version, icc was able to perform inner loop vectorization of loop j and unroll it by a factor of 8 (2 vectors). Icc also performs scalar replacement on reference A. In the other three versions (Scalar, SIMD and SIMD+VR) loop j has been unrolled by a factor of 24 (6 vectors) and kept as the outermost loop. Moreover, in the Scalar and SIMD+VR version scalar and vectorial replacement has been respectively applied.

**Ice compiler**

Our kernels and code were compiled using Intel C compiler [10] version 11.1 for intel64 architectures. This version includes several vectorization capabilities as well as memory hierarchy optimizations. To enable them, all the kernels have been compiled with the flags –O3, -restrict, -fno-alias and –msse3. With the same objective the keyword “restrict” has been added in all the function’s headers. Finally, to help the compiler with the vectorization, the pragmas “ivdep” and “vector always” have been used.

**Kernels**

Three kernels have been evaluated to show the effectiveness of our transformations. Table 2 contains a short description and the characteristics of each of them. The third column indicates the iteration space (IS) shape of the loops being transformed. If the IS is not rectangular, then the loop nest contains bound components that are affine functions of the surrounding loops iteration variables. As pointed out in Section 3, for those kernels having non-rectangular iteration space, we use the theory of unimodular transformations to perform loop permutation [16] and Index Set Splitting [29] to make sure that a particular loop performs a constant number of iterations.

**Figure 8.** a) Source code of the register tiled matrix product after applying outer loop vectorization and unroll and jam to loop j. b) Source code of the register tiled matrix product after applying outer loop vectorization, unroll and jam and vectorial replacement. c) Assembly code of a. d) Assembly code of b.
In this case, ORI version again performs better than the Scalar version because SIMD does not apply vectorial replacement, but it applies scalar replacement to reference A in the innermost loop j. In the other three versions (Scalar, SIMD and SIMD+VR) we apply tiling at the register level with tile sizes 6 and 8 for dimensions i and j respectively and use Index Set Splitting [29] to distinguish loop nests that traverse (non-rectangular) boundary tiles from loop nests that traverse (rectangular) non-boundary tiles. These later loop nests can be vectorized and fully unrolled.

In this kernel, both ORI and Scalar versions are executed in scalar. The slight difference in performance between them is due to the loop order. The loop order in ORI version is ikj and therefore reference to A exhibit reuse between different iterations of the innermost loop. In the ORI version, the loop body contains three memory operations (1 load from B and C and 1 store from C). However, the loop order in Scalar version is jik and thus reference to C exhibit reuse between different iterations of the innermost loop. In this version, the loop body only contains two memory operations (1 load from A and B).

Again, we can also see in Figure 10c that SIMD version obtains better performance than ORI and Scalar versions thanks to the vector execution, but SIMD+VR outperforms them. In all three kernels, the SIMD version shows speedup of around 2x over the Scalar version and the SIMD+VR version obtains an additional 2x speedup over the SIMD version.

Finally, we want to point out the difference in performance for small problem sizes between the triangular and the rectangular matrix product kernels. We can observe that SIMD+VR obtains very high performance for small problem sizes (from 24 to 196) in the rectangular matrix product while the same version obtains very low performance in the triangular matrix product. The reason is that for very small problem sizes, the execution time wasted on boundary tiles in the triangular matrix product is significant and these tiles are not vectorized and unrolled.

Figure 9. Triangular matrix product.

We can observe that vector executions (ORI, SIMD and SIMD+VR) obtain always better performance than scalar executions (Scalar). On the other hand, SIMD version is still far away to the ORI version because SIMD does not apply vectorial replacement, performing therefore excessive redundant memory operations inside the innermost loop. Finally, it can be seen that SIMD+VR outperforms ORI version because better register reuse is done.

Figure 10b shows the performance obtained for the rectangular matrix product. In the ORI version of this kernel, icc was not able to vectorize because it does not support non-rectangular loop structure, but it applies scalar replacement to reference A in the innermost loop j. In the other three versions (Scalar, SIMD and SIMD+VR) we apply tiling at the register level with tile sizes 6 and 8 for dimensions i and j respectively and use Index Set Splitting [29] to distinguish loop nests that traverse (non-rectangular) boundary tiles from loop nests that traverse (rectangular) non-boundary tiles. These later loop nests can be vectorized and fully unrolled.

In this case, ORI version again performs better than the Scalar version since it is vectorized. However, the SIMD version performs slightly better than the ORI version because SIMD exploits better the register level due to the register tiling transformation. Although SIMD version does not perform vectorial replacement, it exploits reuses of accesses to A and B inside the register tile.

Finally version SIMD+VR again obtains highest performance since it highly reduces the memory operations (it avoids loads and stores of C in the innermost loop). Moreover, we can also see in Figure 10b that the performance of SIMD+VR starts to decrease at problem size of 216. For medium problem sizes, tiling only at the register level can substantially increase TLB misses and cache misses are not moderated. This problem can be solved by performing tiling also for higher levels of the memory hierarchy.

Figure 10c shows the performance obtained for the triangular matrix product. In the ORI version of this kernel, icc was not able to vectorize because it does not support non-rectangular loop structure, but it applies scalar replacement to reference A in the innermost loop j. In the other three versions (Scalar, SIMD and SIMD+VR) we apply tiling at the register level with tile sizes 6 and 8 for dimensions i and j respectively and use Index Set Splitting [29] to distinguish loop nests that traverse (non-rectangular) boundary tiles from loop nests that traverse (rectangular) non-boundary tiles. These later loop nests can be vectorized and fully unrolled.

In this kernel, both ORI and Scalar versions are executed in scalar. The slight difference in performance between them is due to the loop order. The loop order in ORI version is ikj and therefore reference to A exhibit reuse between different iterations of the innermost loop. In the ORI version, the loop body contains three memory operations (1 load from B and C and 1 store from C). However, the loop order in Scalar version is jik and thus reference to C exhibit reuse between different iterations of the innermost loop. In this version, the loop body only contains two memory operations (1 load from A and B).

Again, we can also see in Figure 10c that SIMD version obtains better performance than ORI and Scalar versions thanks to the vector execution, but SIMD+VR outperforms them. In all three kernels, the SIMD version shows speedup of around 2x over the Scalar version and the SIMD+VR version obtains an additional 2x speedup over the SIMD version.

Finally, we want to point out the difference in performance for small problem sizes between the triangular and the rectangular matrix product kernels. We can observe that SIMD+VR obtains very high performance for small problem sizes (from 24 to 196) in the rectangular matrix product while the same version obtains very low performance in the triangular matrix product. The reason is that for very small problem sizes, the execution time wasted on boundary tiles in the triangular matrix product is significant and these tiles are not vectorized and unrolled.

<table>
<thead>
<tr>
<th>Device</th>
<th>Size</th>
<th>Associativity/#</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 I-Cache</td>
<td>32 KB</td>
<td>4-way</td>
</tr>
<tr>
<td>L1 D-Cache</td>
<td>32 KB</td>
<td>8-way</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB</td>
<td>8-way</td>
</tr>
<tr>
<td>L3 shared Cache</td>
<td>8 MB</td>
<td>16-way</td>
</tr>
<tr>
<td>TLB1</td>
<td>32 entries</td>
<td>4-way</td>
</tr>
<tr>
<td>TLB2</td>
<td>512 entries</td>
<td>4-way</td>
</tr>
<tr>
<td>General Purpose Registers (GPRs)</td>
<td>64-bit-wide</td>
<td>16 registers</td>
</tr>
<tr>
<td>XMM registers</td>
<td>128-bit-wide</td>
<td>16 registers</td>
</tr>
</tbody>
</table>

Table 1. Memory hierarchy of the Intel Xeon E5520.

<table>
<thead>
<tr>
<th>Description</th>
<th>Loop depth</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross addition of 2 vectors</td>
<td>2</td>
<td>Rectangular</td>
</tr>
<tr>
<td>Rectangular matrix product</td>
<td>3</td>
<td>Rectangular</td>
</tr>
<tr>
<td>Triangular matrix product</td>
<td>3</td>
<td>Triangular</td>
</tr>
</tbody>
</table>

Table 2. Characteristics of the evaluated kernels.

void multiply(const float *restrict A,const float *restrict B,float *restrict C,int dimi,int dimj,int dimk){
    long int i,j,k;
    for(k = 0; k < dimk; k++)
        for(i = k; i < dimi; i++)
            for(j = k; j < dimj; j++)
                C[i*dimj+j] += A[i*dimk+k] * B[k*dimj+j];
}

Figure 10. a) Performance of cross addition of 2 vectors. b) Performance of rectangular matrix product. c) Performance of triangular matrix product. d) Performance of SGEMM for the ATLAS and MKL hand-optimized libraries and our best code (SIMD+VR + cache tiling).
At last, we compare our optimized codes against hand-optimized assembly-written numerical libraries. Figure 10d shows the SGEMM performance obtained by ATLAS [25] and MKL [11] and the performance obtained by our optimized rectangular matrix product. To do a fairly comparison, we add cache tiling to the SIMD+VR version of Figure 10d. Cache tiling is effective for reducing the capacity cache miss rate and moderating TLB misses. Thus, for medium matrix sizes that do not fit at the cache level it achieves the same performance level as for smaller sizes.

We can see that MKL achieves the peak performance of a core (2.26GHz * 4 Single Precision Floating Point elements per instruction * 2 instructions per cycle = 18,08 GFLOPS) On the other hand, ATLAS and SIMD+VR+Cache achieve a performance of 14 GFLOPS approximately (77% of the peak performance).

We can also observe in Figure 10d that for large matrix sizes ATLAS achieves slightly better performance than our optimized version. The reason is that ATLAS copies the matrices into small contiguous blocks in memory in order to minimize TLB misses and cache conflicts. In our optimized version we do not use data copying. However, for small problem sizes, our optimized code outperforms ATLAS.

Summarizing, results show that source-to-source optimized codes can almost achieve the same performance as hand-optimized assembly-written codes.

6. Conclusions

SIMD instructions are so far not really exploited by compilers for media processors. Taking advantage of such instructions is only possible if processor-specific assembly routines or compiler intrinsics are used, resulting in low portability of software.

The optimizations proposed in this paper are high-level (source-to-source) transformations that help compilers to generate efficient SIMD code. We have seen that the SIMD+VR version obtains speedups of around 4x over the Scalar version. Working at the source level prevent us from controlling many of the low level transformations typically performed by the compiler’s back-end (instruction scheduling, register allocation, etc.) making it difficult (if not impossible) to generate the optimal code. By integrating these transformations inside a production compiler, we could achieve even more better performance.

Acknowledgments

This research has been supported by an Intel-UPC Research Grant, the Spanish Ministry of Education (contract no. TIN2007-60625), and the European Union (under the HiPEAC-2 Network of Excellence, FP7/ICT 217068).

References