Source-to-Source Transformations for Efficient SIMD Code Generation

Alejandro Berna[1], Marta Jiménez[1], Jose M. Llabería[1]

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 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][18]. They offer higher performance, a 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

[1]Departament 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 replacement. 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 identify 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 all 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.

Aditionally, Callahan et all 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 microporcessors 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 ( i1=L1; i1<U1; i1++)
  for ( i2=L2; i2<U2; i2++)
    ....
      for ( in=Ln; in<Un; in++){
        F(i1,…,in)
      }
```

and assume that loop i_j 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++)
                ↓ Strip-mining loop i
for ( i=L; i<U; i=i+S)
  for (vi=i; vi<i+S; i++)
```

where `i` is the outer loop, `vi` is the element loop and `S` is the strip size.

To perform outer-loop vectorization, we apply strip-mining to the desired vector loop i_j with step size equal to the vector length (VL) and then permute the resulting element loop of VL iterations to become innermost. Thus, we expose the vector statement as an inner loop and commercial compilers are able to vectorize it. After vectorizing loop i_j, we obtain the following code:

Step 1: Strip-mining i_j

```
for ( i1=L1; i1<U1; i1++)
 for ( i2=L2; i2<U2; i2++)
  ....
   for ( ij=Lj; ij<Uj; ij=ij+VL)
    for (vij=ij; vij<ij+VL;vij++)
     ....
      for ( in=Ln; in<Un; in++)
           F(i1,…,vij,…,in)
```

Step 2: Loop permutation to make vi_j innermost

```
for ( i1=L1; i1<U1; i1++)
 for ( i2=L2; i2<U2; i2++)
  ....
   for ( ij=Lj; ij<Uj; ij=ij+VL)
   ....
     for ( in=Ln; in<Un; in++)
      for (vij=ij; vij<ij+VL;vij++)
           F(i1,…,vij,…,in)
```

As an example, Figure 1 shows the original code of a cross addition of two vectors and how this code is vectorized by the icc compiler. We can see that icc performs inner loop vectorization (vectorize loop `j`) and also unrolls loop `j` by a factor of 8 (two vectors). Finally, icc performs a reduction to store the result in vector A. Figure 2 shows the SIMD optimized code (after applying outer-loop vectorization) and how this code is vectorized by the compiler. We can observe that the compiler remove loop `vi` and convert it to a set of vector instructions and thus outer loop `i` has been vectorized. Later in Section 5, we will see the difference in performance between these two codes.

3.2 Unroll and Jam

Unroll and Jam is a transformation that can be used to shorten the distances between references to the same array location and therefore it enhances register reuse. It consists in unrolling an outer loop and then fusing the inner loops back together. As we will see later in Section 4 unroll and jam is a necessary transformation when combining register tiling with vectorization. As already mentioned, we observed that commercial compilers are not able to unroll strip-mined loops (loops with non-unit stride).

Original Source code	ASM	
void cross_add(float *A, float *B, int dimi, int dimj){	.LOOP_I:	
long int i, j;	movss (%rdi,%r8,4), %xmm0	#ld A[i]
for (i=0; i<dimi; i++)	xorps %xmm1, %xmm1	
for (j=0; j<dimj; j++)	.LOOP_J:	
A[i]=A[i]+B[j];	addps (%rsi,%rax,4), %xmm0	#A[i]+B[j:j+3]
}	addps 16(%rsi,%rax,4),%xmm1	#A[i]+B[j+4:j+7]
	addq $8, %rax	
	cmpq %r11, %rax	
	jb ..LOOP_J	
	addps %xmm1, %xmm0	
	haddps %xmm0, %xmm0	
	haddps %xmm0, %xmm0	
	movss %xmm0, (%rdi,%r8,4)	#st A[i]
	incq %r8	
	cmpq %rcx, %r8	
	jb ..LOOP_I	

Figure 1. Cross addition of two vectors. The left column shows the source code and the right the assembly code.

Optimized code	ASM	
void cross_add(float *A, float *B, int dimi, int dimj){	..LOOP_I:	
long int i, j, vi;	xorl %r9d, %r9d	
for (i=0; i<dimi; i+=VL)	movq %rcx, %r10	
for (j=0; j<dimj; j++)	shlq $4, %r10	
#pragma vector always	movups (%r10,%rdi), %xmm0	#ld A[i:i+3]
for(vi=i;vi<i+VL;vi++)	..LOOP_J:	
A[vi]=A[vi]+B[j];	movss (%rsi,%r9,4), %xmm1	#ld B[j]
}	shufps $0, %xmm1, %xmm1	
	addps %xmm1, %xmm0	#A[i:i+3]+B[j]
	incq %r9	
	cmpq %rax, %r9	
	jb..LOOP_J	
	movups %xmm0, (%r10,%rdi)	#st A[i]
	incq %rcx	
	cmpq %rdx, %rcx	
	jb ..LOOP_I	

Figure 2. Cross addition after applying outer-loop vectorization. The left column shows the source code*, the right the assembly.

However, to generate efficient SIMD code we need the compiler to perform this transformation. To this end, we help the compiler by directly unrolling the strip-mined loop in the source code and jaming together the inner loops as follows:
Consider the following loop nest where outer loop vectorization has been applied to loop i_j:

```
....
for ( ij=Lj; ij<Uj; ij=ij+VL)
 ....
  for (vij=ij; vij<ij+VL;vij++)
       F(vij)        /* vector statement */
```

After unrolling loop i_j with an unroll factor of UF, we obtain the following code:

```
....
for ( ij=Lj; ij<=Uj; ij=ij+VL*UF)
{  ...
   for (vij=ij; vij<ij+VL;vij++)
         F(vij)
   for (vij=ij+VL;vij<ij+2*VL;vij++)
         F(vij)
    ....
   for(vij=ij+(UF-1)*VL; vij<ij+UF*VL;vij++)
         F(vij)                  /* vector statements */
}
```

and after fusion becomes:

```
....
for ( ij=Lj; ij<Uj; ij=ij+VL*UF)
 ...
   for (vij=ij; vij<ij+VL;vij++){
         F(vij)
         F(vij+VL)
         F(vij+2*VL)
         ....
         F(vij+(UF-1)*VL)
   }              /* vector statements */
```

Now, reuse between several vector statements are exposed in the loop body.

Using again the example of cross addition from Section 3.1, we observed in Figure 2 that icc does not unroll loop i after applying outer loop vectorization. Vector B is loaded `dimi/VL` times during the execution of the program. In each iteration of loop j, B[j] is loaded on register %xmm1.

However, if we apply unroll and jam to loop i by a factor UF, we can enhance data reuse of reference B[j] by keeping this value in a register during the execution of the unrolled loop body. Thus, vector B will only be loaded `dimi/(UF*VL)` times during the execution of the program. In Figure 3 we show the code of Figure 2 after applying unroll and jam by a factor of 2 to loop i. Element B[j] is loaded on register %xmm1 only once during the execution of the unrolled loop body, thus enhancing reuse of B by a factor of 2.

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

Source code	ASM	

void cross_add(float *A, float *B,
int dimi, int dimj){
long int i, j, vi;
for (i=0; i<dimi; i+=2*VL)
for (j=0; j<dimj; j++)
#pragma vector always
for(vi = i; vi <i+VL; vi++){
A[vi]=A[vi]+B[j];
A[vi+VL]=A[vi+VL]+B[j];
}
}
	..LOOP_I:
	xorl %r10d, %r10d
	..B7.4:
	movq %rax, %r9
	shlq $5, %r9
	..LOOP_J:
	movups (%r9,%rdi), %xmm0
	movups16(%r9,%rdi),%xmm2
	movss(%rsi,%r10,4), %xmm1
	shufps $0, %xmm1, %xmm1
	addps %xmm1, %xmm0
	movups %xmm0, (%r9,%rdi)
	addps %xmm1, %xmm2
	movups%xmm2,16(%r9,%rdi)
	incq %r10
	cmpq %rcx, %r10
	jb..LOOP_J
	incq %rax
	cmpq %rdx, %rax
	jb..LOOP_I
		#ld A[i:i+3]
		#ld A[i+4:i+7]
		#ld B[j]
		#A[i:i+3]+B[j]
		#st A[i:i+3]
		#A[i+4:i+7]+B[j]
		#st A[i+4:i+7]

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.

Although data reuse has been exposed by applying unroll and jam to loop i in the source code, the icc compiler is not able to eliminate redundant loads and stores in the new unrolled loop body. In Figure 3, we can see that reference A[vi] and A[vi+VL] are loaded on/stored from registers %xmm0 and %xmm2, respectively, in each iteration of loop j. However these two references are invariant with respect to loop j. Note that this problem does not happen if we do not perform unroll and jam to loop `i`. In Figure 2, reference A[vi] is loaded on/stored from register %xmm0 only once during the execution of loop j. To overcome this problem, we also need to perform vectorial replacement to the source code.

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 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 and expose data reuse in the source code by initializing and storing the temporary arrays out of the innermost loop. Vectorial replacement can be implemented using both temporary arrays variables or pointer variables.

Continuing with the cross addition example, after applying vectorial replacement to the code of Figure 3 we obtain the code of Figure 4. Notice that after applying vectorial replacement to the source code, the icc compiler is able to remove redundant

Source code	ASM	

void cross_add(float *A, float *B,
int dimi, int dimj){
long int i, j, vi;
float A1[VL], A2[VL];
for (i=0; i<dimi; i+=2*VL){
for(vi = 0; vi < VL; vi++){
A1[vi]=A[i+vi];
A2[vi]=A[i+VL+vi];
}
for (j=0; j<dimj; j++)
#pragma vector always
#pragma ivdep
for(vi = 0; vi < VL; vi++){
A1[vi]=A1[vi]+B[j];
A2[vi]=A2[vi]+B[j];
}
#pragma vector always
for(vi = 0; vi < VL; vi++){
A[i+vi]=A1[vi];
A[i+VL+vi]=A2[vi];
}}}
	..LOOP_I:
	movups (%rax,%rdi),%xmm1
	movups 16(%rax,%rdi),%xmm0
	xorl %r9d, %r9d
	testq %r8, %r8
	jle ..B7.7
	..LOOP_J:
	movss (%rsi,%r9,4), %xmm2
	shufps $0, %xmm2, %xmm2
	addps %xmm2, %xmm1
	addps %xmm2, %xmm0
	incq %r9
	cmpq %r8, %r9
	jb ..LOOP_J
	..B7.7:
	movups %xmm1, (%rax,%rdi)
	movups %xmm0,16(%rax,%rdi)
	addq $32, %rax
	incq %rcx
	cmpq %rdx, %rcx
	jb ..LOOP_I
		#ld A[i:i+3]
		#ld A[i+4:i+7]
		#ld B[j]
		#A[i:i+3]+B[j]
		#A[i+4:i+7]+B[j]
		#st A[i:i+3]
		#st A[i+4:i+7]

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.

loads and stores from the loop body. In Figure 4, reference A[i+vi] and A[i+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.

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

Source code	ASM	

void cross_add(float *A, float *B,
int dimi, int dimj){
long int i, j, vi;
float *A1, *A2;
A1 = A;
A2 = A1+VL;
for (i=0; i<dimi; i+=2*VL){
for (j=0; j<dimi; j++)
#pragma vector always
#pragma ivdep
for(vi = 0; vi < VL; vi++){
A1[vi]=A1[vi]+B[j];
A2[vi]=A2[vi]+B[j];
}
A1+=2*VL;
A2+=2*VL;
}
}
	..LOOP_I:
	xorl %r8d, %r8d
	movq %rcx, %r9
	shlq $5, %r9
	movups16(%r9,%rdi),%xmm1
	movups (%r9,%rdi), %xmm0
	..LOOP_J:
	movss (%rsi,%r8,4), %xmm2
	shufps $0, %xmm2, %xmm2
	addps %xmm2, %xmm0
	addps %xmm2, %xmm1
	incq %r8
	cmpq %rax, %r8
	jb..LOOP_J
	movups%xmm1,16(%r9,%rdi)
	movups %xmm0, (%r9,%rdi)
	incq %rcx
	cmpq %rdx, %rcx
	jb..LOOP_I
		#ldA[i+4:i+7]
		#ldA[i:i+3]
		#ld B[j]
		#A[i:i+3]+B[j]
		#A[i+4:i+7]+B[j]
		#st A[i+4:i+7]
		#st A[i:i+3]

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.

Source code	ASM

void multiply(float* A, float* B,
float* C, int dimi,
int dimk, int dimj){
long int i, j, k;
for (i = 0; i < dimi; i++)
for (j = 0; j < dimj; j++)
for (k = 0; k < dimk; k++)
C[i*dimj+j]+=A[i*dimk+k]*
B[k*dimj+j];
}
	..LOOP_I:
	...
	..LOOOP_K:
	movq %rbx, %rcx
	movq %r14, %rcx
	movss (%r12,%rbp,4), %xmm0
	shufps $0, %xmm0, %xmm0
	..LOOP_J:
	movups (%rdx,%rcx,4), %xmm1
	movups 16(%rdx,%rcx,4),%xmm2
	mulps %xmm0, %xmm1
	mulps %xmm0, %xmm2
	addps (%rsi,%rcx,4), %xmm1
	addps 16(%rsi,%rcx,4),%xmm2
	movaps %xmm1, (%rsi,%rcx,4)
	movaps %xmm2, 16(%rsi,%rcx,4)
	addq $8, %rcx
	cmpq %r10, %rcx
	jb ..LOOP_J
	addq %r9, %r15
	incq %rbp
	cmpq %r8, %rbp
	jb ..LOOP_K
	...
	jb ..LOOP_I

Figure 6. Matrix product. 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 [6][14][26] 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 [16] is loop transformation that a compiler can use 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 any memory level can be implemented by combining two well-known transformations: strip-mining and loop interchange. However, the implementation of 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 tiles. Therefore, in our example of Figure 7a, it is necessary to fully unroll loops i and j to exploit the register level. At last, scalar replacement [5][6] can be used to eliminate redundant loads and stores in the new unrolled loop body.

When combining register tiling with vectorization we need first vectorize the desired loop (loop j, in our example) before fully unroll the register tile. Thus, the outer loop j is vectorized as explained in subsection 3.1. We apply strip-mining to loop j with a step size of VL and then permute the resulting element

a)

```
long int ii, jj, i, j, k;
for (ii = 0; ii < dimi; ii+=BI)
 for (jj = 0; jj < dimj; jj+=BJ)
  for (k = 0; k < dimk; k++)
   for(j = jj; j < jj+BJ; j++)
    for(i = ii; i < ii+BI; i++)
     C[i*dimj+j]+=A[i*dimk+k]*B[k*dimj+j];
```

b)

```
long int ii, jj, i, j, k, vj;
for (ii = 0; ii < dimi; ii+=BI)
 for (jj = 0; jj < dimj; jj+=BJ)
  for (k = 0; k < dimk; k++)
   for(j = jj; j < jj+BJ; j+=VL)
    for(i = ii; i < ii+BI; i++)
     for(vj=j; vj<j+VL;vj++)
      C[i*dimj+vj]+=A[i*dimk+k]*B[k*dimj+vj];
```

Figure 7. a) Register tiled matrix product. b) Register tiled matrix product after applying outer loop vectorization to loop j.

loop of VL iterations to become the innermost (the vector statement). The resulting code is shown in Figure 7b assuming BJ is multiple of VL for simplicity.

As already mentioned, now it is necessary to fully unroll the loops that traverse the iterations inside the register tile (loop i and j in Figure 7b). To fully unroll the strip-mined loop j we perform unroll and jam as explained in Section 3.2. The resulting code is shown in Figure 8a, assuming BI = 2 and BJ=2*VL.

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 vectorial replacement to reference C as explained in section 3.3. Figure 8b shows the resulting source code using pointers as temporary variables to identify the adjacent array references. We can see in Figure 8d how icc is now able to remove redundant memory instructions.

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.

Source code

a)

```
long int ii, jj, k, vj;
for (ii = 0; ii < dimi; ii+=2)
  for (jj = 0; jj < dimj; jj+=2*VL)
    for (k = 0; k < dimk; k++)
      #pragma ivdep
      for(vj=jj; vj<jj+VL;vj++) {
       C[ii*dimj+vj]+=A[ii*dimk+k]*B[k*dimj+vj];
       C[ii*dimj+vj+VL]+=A[ii*dimk+k]*B[k*dimj+vj+VL];
       C[(ii+1)*dimj+vj]+=A[(ii+1)*dimk+k]*B[k*dimj+vj];
       C[(ii+1)*dimj+vj+VL]+=A[(ii+1)*dimk+k]*B[k*dimj+vj+VL];
      }
```

b)

```
long int ii, jj, k, vj;
float *C1, *C2, *C3, *C4;
const float *B1, *B2, *A1, *A2;
for (ii = 0; ii < dimi; ii+=2){
  A1 = &A[ii*dimk];
  A2 = &A[(ii+1)*dimk];
  for (jj = 0; jj < dimj; jj+=2*VL) {
    C1 = &C[ii*dimj+jj];
    C2 = &C[(ii+1)*dimj+jj];
    C3 = &C[ii*dimj+jj+4];
    C4 = &C[(ii+1)*dimj+jj+4];
    for (k = 0; k < dimk; k++) {
      B1 = &B[k*dimj+jj];
      B2 = &B[k*dimj+jj+VL];
      #pragma ivdep
      for(vj = 0; vj<VL; vj++){
        C1[vj] += A1[k]*B1[vj];
        C3[vj] += A1[k]*B2[vj];
        C2[vj] += A2[k]*B1[vj];
        C4[vj] += A2[k]*B2[vj];
      }
    }
  }
```

ASM

c)

```
..LOOP_I:
...
..LOOP_J
xorl    %r8d, %r8d
xorl    %ebp, %ebp
movq    %rbx, %rcx
shlq    $5, %rcx
lea     (%r11,%rcx), %rax
..LOOP_K:
movups  (%rax,%rbp,4), %xmm4
movups  (%r12,%rcx), %xmm1
movups  16(%rax,%rbp,4), %xmm6
movups  16(%r12,%rcx), %xmm3
movss   (%rdx,%r8,4), %xmm2
movss   (%r13,%r8,4), %xmm7
shufps  $0, %xmm2, %xmm2
movaps  %xmm2, %xmm0
mulps   %xmm6, %xmm2
mulps   %xmm4, %xmm0
shufps  $0, %xmm7, %xmm7
mulps   %xmm7, %xmm4
mulps   %xmm6, %xmm7
addps   %xmm0, %xmm1
movups  %xmm1, (%r12,%rcx)
addps   %xmm2, %xmm3
movups  %xmm3, 16(%r12,%rcx)
movups  (%rcx,%rsi), %xmm5
movups  16(%rcx,%rsi), %xmm8
addq    %r10, %rbp
addps   %xmm4, %xmm5
movups  %xmm5, (%rcx,%rsi)
addps   %xmm7, %xmm8
movups  %xmm8, 16(%rcx,%rsi)
incq    %r8
cmpq    %r14, %r8
jb      ..LOOP_K
incq    %rbx
cmpq    %r9, %rbx
jb      ..LOOP_J
...
jb      ..LOOP_I
```

d)

```
..LOOP_I:
...
..LOOP_J:
xorl    %ebp, %ebp
xorl    %ecx, %ecx
movq    %r12, %rsi
shlq    $5, %rsi
movups  (%r8,%rsi), %xmm3
movups  16(%r8,%rsi), %xmm2
movups  (%rdx,%rsi), %xmm1
movups  16(%rsi,%rdx), %xmm0
lea     (%r11,%rsi), %rax
..LOOP_K:
movups  (%rcx,%rax), %xmm6
movups  16(%rcx,%rax), %xmm7
movss   (%r13,%rbp,4), %xmm5
movss   (%rbx,%rbp,4), %xmm8
shufps  $0, %xmm5, %xmm5
movaps  %xmm5, %xmm4
mulps   %xmm7, %xmm5
mulps   %xmm6, %xmm4
shufps  $0, %xmm8, %xmm8
mulps   %xmm8, %xmm6
mulps   %xmm7, %
addps   %xmm4, %xmm3
addps   %xmm5, %xmm2
addps   %xmm6, %xmm1
addps   %xmm8, %xmm0
lea     (%rcx,%r10,4), %rcx
incq    %rbp
cmpq    %r14, %rbp
jb      ..LOOP_K
movups  %xmm0, 16(%rsi,%rdx)
movups  %xmm1, (%rdx,%rsi)
movups  %xmm2, 16(%r8,%rsi)
movups  %xmm3, (%r8,%rsi)
incq    %r12
cmpq    %r9, %r12
jb      ..LOOP_J
...
jb      ..LOOP_I
```

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.

Icc 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.

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 i 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.

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

Table 1. Memory hierarchy of the Intel Xeon E5520.

Description	Loop depth	IS
Cross addition of 2 vectors (Figure 1)	2	Rectangular
Rectangular matrix product (Figure 6)	3	Rectangular
Triangular matrix product (Figure 9)	3	Triangular

Table 2. Characteristics of the evaluated kernels.

```
void multiply(const float *restrict A,const float *restrict B,float *restrict C,int dimi,
              int dimk,int dimj){
  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 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 (code of Figure 6) of this kernel, icc was able to vectorize loop j (inner loop vectorization) and unroll it by a factor of 8 (2 vectors). Again, icc was also able to perform scalar replacement to reference A of the loop body. In the other three versions (Scalar, SIMD and SIMD+VR) register tiling has been applied with tile sizes 6 and 8 for dimension i and j, respectively. Moreover, in the Scalar and SIMD+VR version scalar and vectorial replacement has been respectively applied.

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 ijk 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 obtain 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.

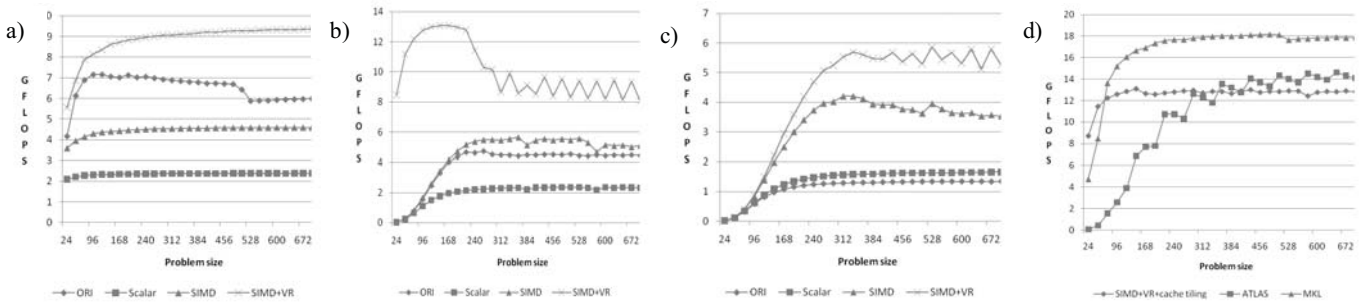

Figure 10. a) Performance of cross addition of 2 vectors. b) Performance of rectangular matrix product. c) Performance of triangular matrix product. d) Performance of SGEMM for the ATLAS and MKL hand-optimized libraries and our best code (SIMD+VR + cache tiling).

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 Excelence, FP7/ICT 217068).

References

[1] D. Aberdeen, and J. Baxter. EMMERALD: a fast matrix-matrix multiply using Intel's SSE instructions, J. Concurrency Comput.: Pract. Exp., 13, 103-119, 2001.

[2] U. Banerjee. Dependence analysis for supercomputing. Norwell, Mass. Kluwer Academic Publishers, 1988.

[3] A. Bik. The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance". Intel Press. 2004.

[4] A. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic Intra-Register Vectorization for the Intel Architecture. Int. J. Parallel Program. 30, 2, 65-98. April 2002.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. PLDI '90. pp. 53-65. June 1990.

[6] S. Carr. Memory-hierarchy management. Ph.D. Thesis, Rice University, February 1993.

[7] G. Cheong, M.S. Lam, An optimizer for multimedia instruction sets, in: The Second SUIF Compiler Workshop, Stanford University, USA, 1997.

[8] Y.F. Fung, M.F. Ercan, T.K. Ho, and W.L. Cheung. A parallel solution to linear systems. Microprocess. Microsyst., 26, 39-44, 2002.

[9] M. Hassaballah, S. Omran, and Y. B. Mahdy. A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications. The Computer Journal, Vol. 51 (6): 630-649. January 2008.

[10] Intel Corporation. Intel C/C++ Compiler User and Reference Guide. Order Number 304968-023US.

[11] Intel Corporation. Intel Math Kernel Library Reference Manual. Order Number 630813-038US.

[12] Intel Corporation (2010). Intel Advanced Vector Extensions Programming Reference. Order Number 319433-009, December 2010.

[13] M. Jimenez, J. Llaberia, A. Fernandez. On the Performance of Hands vs. Automatically Optimized Numerical Codes. HPCA-6 IEEE Computer Society, January 2000, p. 183-194

[14] M. Jimenez, J. Llaberia, A. Fernandez. Register Tiling in Nonrectangular Iteration Spaces. "ACM transactions on programming languages and systems", Juliol 2002, vol. 24, núm. 4, p. 409-453.

[15] A. Krall and S. Lelait. Compilation Techniques for Multimedia Processors. Int. J. of Parallel Programming. 28, 4, 347-361. August 2000.

[16] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimization of Blocked Algorithms. ASPLOS'91, pp. 63-74, 1991.

[17] D. Maydan, J. Hennessy and M. Lam. Efficient and exact data dependence analysis. PLDI'91, pp. 1-14, June 1991.

[18] A. Muezerie, R.J. Nakashima, G. Travieso, and J. Slaets. Matrix calculations with SIMD floating point instructions on x86 processors. HPCA'01, pp. 50-55, September 2001.

[19] D. Nuzman, A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. PACT '08, pp.2-11, 2008.

[20] A. Peleg, U. Weiser. MMX Technology Extension to the Intel Architecture, IEEE Micro, Vol. 16, No. 4, pp. 42-50, August 1996.

[21] I. Pryanishnikov , A. Krall , N. Horspool. Compiler optimizations for processors with SIMD instructions. Software—Practice & Experience, v.37 n.1, p.93-113, January 2007

[22] W. Pugh. "A practical algorithm for exact array dependence analysis". Communications of the ACM, Vol. 35, No. 8, pp. 102-114, 1992.

[23] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. ISCA '99, 124-135, 1999.

[24] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. J. of Parallel Programming, 28, 363-400. 2000.

[25] R. C. Whaley, A. Petitet, J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project, Parallel Computing, 27(1-2):3-35, 2001.

[26] M. Wolf and M. Lam. A data locality optimizing algorithm. PLDI'91, pp. 30-44, June 1991.

[27] M. Wolf, D. Maydan and D.K. Chen. "Combining loop transformations considering caches and scheduling". MICRO-29, pp. 274-286, December 1996.

[28] M. Wolf. Improving locality and parallelism in nested loops. Ph.D. Thesis, Stanford University, 1992.

[29] M. Wolfe. High performance compilers for parallel computing. Addison-Wesley Publishing Company, 1996.

[30] C.-L. Yang, B. Sano, and A. R. Lebeck. Exploiting Parallelism in Geometry Processing with General Purpose Processors and Floating-Point SIMD Instructions, IEEE Transactions on Computers, 49(9), 934-946, September 2000.