

# Source code transformations for efficient SIMD code generation

A. BERNA, M. JIMENEZ, J. M. LLABERIA

Despite the effort inverted the last years in commercial compilers to generate efficient SIMD instructions-based code sequences from conventional sequential programs, the small numbers of compilers that can automatically use these instructions achieve in most cases unsatisfactory results. This work shows how exposing register level reuse in source codes helps vectorizing compilers as ICC to generate efficient SIMD code that exploits vectorial register reuse. To this end we use reverse-engineering to know the limitations of ICC compiler. We compare performance of compiler generated code to performance of hand-optimized assembly-written numerical libraries achieving around the 72% performance of MKL for example.

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

Additional Key Words and Phrases: SIMD, vectorization, source-to-source transformations, register tiling

## 1. INTRODUCTION

The ISA of all today's microprocessors has been extended with multimedia instructions [12]. 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[15][25].

SIMD instructions are useful in multimedia and signal processing applications [30][40], but also in scientific and numerical applications [1][10][22]. 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 [15]. 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 [14]) 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.

In this paper, we show how combining well-known high level (source-to-source) transformations we can help current commercial vectorizing compilers to generate efficient SIMD code on scientific numerical applications without the need of writing in assembly language.

Our proposal is based on an effective use of the vector registers. In particular, we show how to expose vector reuse in the source code as a step for helping ICC compiler [13] to apply vectorial replacement in order to reduce the gap between compiled code and hand-optimized libraries as ATLAS [35].

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 al in [6] 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 jam, could expose more opportunities for scalar replacement by moving reuse across an outer loop into the innermost loop.

In this work, we use the idea of scalar replacement and unroll and jam to improve vector reuse and show experimentally their effectiveness. We refer as vectorial replacement to the scalar replacement transformation applied to SIMD vectorized loop nests.

Since we do not have access to the ICC source code, we use reverse-engineering to discover ICC limitations to apply vectorial replacement. To this end, we perform simple modifications to source code examples until ICC performs vectorial replacement. At this point, we want to remark that, after applying well-known high level transformations to expose reuse, the keys for ICC to perform vectorial replacement are: a) identifying vectors locally in the source code by defining local arrays or pointers variables and b) linearization of subscript variables.

The modifications done in the source code are explained as a sequence of syntactic transformations. However these transformations can be implemented using polyhedral models and tools that transform annotated source code [5][11][26].

Summarizing, the contributions of this paper are the following:

- An approach that combines source-to-source transformations (outer-loop vectorization, unroll and jam of vectorized loops and vectorial replacement) and clever programming tricks to help ICC compiler to generate efficient SIMD codes that exploit register level reuse in scientific numerical applications.
- Experimental evaluation exhibiting the impact of these transformations using simple kernels of loop nests and some BLAS routines [20] 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 and outer loop vectorization. Section 3 describes our reverse-engineering work to discover ICC limitations and shows that source-to-source transformations and clever programming tricks help ICC compiler to exploit register level reuse. 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

Several researchers [3][9][18][24][28][32] have worked in the context of straight-line code vectorization. These researchers focus on automatically identify vectorizable sections of code and generate appropriate SIMD instructions. Moreover, most of these auto-vectorization approaches focus on innermost loops [9][18][33] or block vectorization [4]. Only Nuzman et al in [24] deals with outer loop vectorization and show its effectiveness. Their proposal use in-place outer loop vectorization and it is implemented in the GCC compiler.

Additionally, Callahan et al in [6] presented a source-to-source transformation, called scalar replacement, that exposes the reuse available in array references in an innermost loop. They also show experimentally how another loop transformation, called unroll and jam, could expose more opportunities for scalar replacement. In our work, we extend the use of scalar replacement and unroll and jam to SIMD vectorized loop nests and show experimentally their effectiveness. Shin et al in [32] also use the idea of scalar replacement and unroll and jam on SIMD codes to develop an algorithm and an implementation to exploit reuse of data in vector registers, their algorithm is a set of low level optimizations to be implemented in the context of straight-line code vectorization.

Other works address vectorization using polyhedral model representation [27][34]. In our work we use a sequence of high level (source-to-source) transformations in order to perform reverse-engineering on the ICC compiler to know how register level reuse has to be exposed in the source code. Thus compiler exploits vector register level when generating SIMD code. This sequence of transformations can be implemented using the polyhedral models and tools that transform annotated source code [5][26] [31].

Finally, there exist several hand-coded numerical libraries optimized for SIMD processors [14][35] 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. REVERSE ENGINEERING FOR COMPILER LIMITATIONS DISCOVERING

In this section we use the cross addition of two vectors as a code example for discovering ICC limitations by reverse-engineering. Fig. 1 shows the original code 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*.

By analyzing several codes generated by ICC (including our cross addition example) we did three observations. First, we observe that the compiler only perform inner loop vectorization. However, in most codes it is necessary to vectorize outer loops to achieve high performance.

Second, we observe that the compiler is not able to unroll and jam loops with non unit stride. As we will see later, optimizing transformations like register tiling [7][16][17] 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 the compiler is 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 [7][16][17].

Original Source code	ASM	
<pre> void cross_add(float *A, float *B, int dim1, int dimj){   long int i, j;   for (i=0; i&lt;dim1; i++)     for (j=0; j&lt;dimj; j++)       A[i]=A[i]+B[j]; } </pre>	<pre> .LOOP_I: movss (%rdi,%r8,4),%xmm0 xorps %xmm1,%xmm1 .LOOP_J: addps(%rsi,%rax,4),%xmm0 addps16(%rsi,%rax,4),%xmm1 addq \$8,%rax cmpq %r11,%rax jb ..LOOP_J addps %xmm1,%xmm0 haddps %xmm0,%xmm0 haddps %xmm0,%xmm0 movss %xmm0, (%rdi,%r8,4) incq %r8 cmpq %rcx,%r8 jb ..LOOP_I </pre>	<pre> #ld A[i] #A[i]+B[j:j+3] #A[i]+B[j+4:j+7] #st A[i] </pre>

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

In the next subsections we show how we can help the compiler to generate efficient SIMD code that exploits register level reuse by applying well-known 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 [19] and Index Set Splitting [39] for making sure that a particular loop perform a constant number of iterations. We describe the transformations in a syntactic form.

We also assume that previous analysis to decide which loops could be vectorized has already been performed. This paper only focuses on the code generation phase of source-to-source transformations. Dependence analysis to know if transformations are legal are out of the scope of this paper [2][21][29].

The transformations that we apply to help the compiler to generate efficient SIMD code that exploits register level reuse are: outer loop vectorization, unroll and jam and vectorial replacement. Moreover, we use clever programming tricks to help compiler to identify vectorial register reuse. In particular, we use local temporal pointer or array variables of vector register size and subscript variable linearization.

### 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 [39] 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 ( i_j=L_j; i_j<U_j; i_j=i_j+VL)
for ( vi_j=i_j; vi_j<i_j+VL; vi_j++)
....
for ( in=Ln; in<Un; in++)
    F(i1,...,vi_j,...,in)

```

**Step 2: Loop permutation to make  $vi_j$  innermost**

```

for ( i1=L1; i1<U1; i1++)
for ( i2=L2; i2<U2; i2++)
....
for ( i_j=L_j; i_j<U_j; i_j=i_j+VL)
....
for ( in=Ln; in<Un; in++)
    for ( vi_j=i_j; vi_j<i_j+VL; vi_j++)
        F(i1,...,vi_j,...,in)

```

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

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

Fig. 2 shows the SIMD optimized code (after applying outer-loop vectorization to the source code of Fig. 1) 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.

We also observed in Fig. 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 o

\*From now on, we use pragmas in the codes to force vectorization (see section 5).

we apply unroll and jam to loop  $i$  by a factor of  $U$ , we can eliminate 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  $dim_i/(UF*VL)$  times during the execution of the program

### 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).

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 ( i_j=L_j; i_j<U_j; i_j=i_j+VL)
    ...
    for ( v_i_j=i_j; v_i_j<i_j+VL; v_i_j++)
        F(v_i_j)      /* vector statement */

```

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

```

...
for ( i_j=L_j; i_j<=U_j; i_j=i_j+VL*UF)
{
    ...
    for ( v_i_j=i_j; v_i_j<i_j+VL; v_i_j++)
        F(v_i_j)
    for ( v_i_j=i_j+VL; v_i_j<i_j+2*VL; v_i_j++)
        F(v_i_j)
    ...
    for ( v_i_j=i_j+(UF-1)*VL; v_i_j<i_j+UF*VL; v_i_j++)
        F(v_i_j)      /* vector statements */
}

```

and after fusion becomes:

```

...
for ( i_j=L_j; i_j<U_j; i_j=i_j+VL*UF)
    ...
    for ( v_i_j=i_j; v_i_j<i_j+VL; v_i_j++){
        F(v_i_j)
        F(v_i_j+VL)
        F(v_i_j+2*VL)
        ...
        F(v_i_j+(UF-1)*VL)
    }
    /* vector statements */

```

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

Fig. 3 show the code example after applying unroll and jam to loop  $i$  in the source code. Although data reuse has been exposed, the ICC compiler is not able to eliminate redundant loads and stores in the new unrolled loop body. In Fig. 3, we can see that reference  $A[v_i]$  and  $A[v_i+VL]$  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 Fig. 2, reference  $A[v_i]$  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.

Source code	ASM	
<pre>void cross_add(float *A, float *B,               int dimi, int dimj){ long int i, j, vi; for (i=0; i&lt;dimi; i+=2*VL) for (j=0; j&lt;dimj; j++) #pragma vector always for(vi = i; vi &lt;i+VL; vi++){     A[vi]=A[vi]+B[j];     A[vi+VL]=A[vi+VL]+B[j]; } }</pre>	<pre>..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</pre>	<pre>#d A[i+3] #d A[i+4:i+7] #d B[j]  #A[i+3]+B[j] #st A[i+3] #A[i+4:i+7]+B[j] #st A[i+4:i+7]</pre>

Fig. 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.

### 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 [6][7], 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 Fig. 3 we obtain the code of Fig. 4. Notice that after applying vectorial replacement to the source code, the ICC compiler is able to remove redundant loads and stores from the loop body. In Fig. 4, reference  $A[i+vi]$  and  $A[i+VL+vi]$  are loaded and stored only once during the execution of the loop j.

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

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

Fig. 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.

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

Fig. 5. Cross addition after performing outer-loop vectorization, unroll and jam and vectorial replacement using temporary pointers. The left column shows the source code and the right the assembly code.

#### 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).

Source code	ASM
<pre>void multiply(float* A, float* B,              float* C, int dimi,              int dimk, int dimj){     long int i, j, k;     for (i = 0; i &lt; dimi; i++)         for (j = 0; j &lt; dimj; j++)             for (k = 0; k &lt; dimk; k++)                 C[*dimj+j]=A[*dimk+k]*B[k*dimj+j]; }</pre>	<pre>..LOOP_I: ... ..LOOP_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</pre>

Fig. 6. Matrix product. The left column shows the source code and the right the assembly code.

First of all, we compiled the original matrix product, shown in Fig. 6, using ICC with all compiler optimizations (including vectorization) turned on. It can be seen that ICC always permutes 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 [7][17][36] to the source code as shown in Fig. 7a.  $B_I$  and  $B_J$  are the tile sizes in dimension  $i$  and  $j$ , respectively, and

<pre>long int ii, jj, i, j, k; for (ii = 0; ii &lt; dimi; ii+=BI)     for (jj = 0; jj &lt; dimj; jj+=BJ)         for (k = 0; k &lt; dimk; k++)             for (j = jj; j &lt; jj+BJ; j++)                 for (i = ii; i &lt; ii+BI; i++)                     C[*dimj+j]=A[*dimk+k]*B[k*dimj+j];</pre>	a)	<pre>long int ii, jj, i, j, k, vj; for (ii = 0; ii &lt; dimi; ii+=BI)     for (jj = 0; jj &lt; dimj; jj+=BJ)         for (k = 0; k &lt; dimk; k++)             for (j = jj; j &lt; jj+BJ; j+=VL)                 for (i = ii; i &lt; ii+BI; i++)                     for (vj=j; vj&lt;j+VL; vj++)                         C[*dimj+vj]=A[*dimk+k]*B[k*dimj+vj];</pre>	b)
---	----	--	----

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

Source code	ASM
<pre> long int ii, jj, k, vj; for (ii = 0; ii &lt; dimi; ii+=2)   for (jj = 0; jj &lt; dimj; jj+=2*VL)     for (k = 0; k &lt; dimk; k++)       #pragma ivdep       for (vj=jj; vj&lt;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];       } </pre>	<pre> ..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 #ld C[ii*dimj+vj:ii*dimj+vj+3] movups16(%rax,rbp,4),%xmm6 #ld C[ii*dimj+vj+4:ii*dimj+vj+7] movups16(%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) #st C[ii*dimj+vj: ii*dimj+vj+3] addps %xmm2, %xmm3 #st C[ii*dimj+vj+4:ii*dimj+vj+7] movups%xmm3,16(%r12,%rcx) #ld C[(ii+1)*dimj+vj:(ii+1)*dimj+vj+3] movups(%rcx,%rsi,%xmm5) #ld C[(ii+1)*dimj+vj+4:(ii+1)*dimj+vj+7] movups16(%rcx,%rsi),%xmm8 addq %r10, %rbp addps %xmm4, %xmm5 #st C[(ii+1)*dimj+vj:(ii+1)*dimj+vj+3] movups %xmm5,(%rcx,%rsi) #st C[(ii+1)*dimj+vj+4:(ii+1)*dimj+vj+3] addps %xmm7, %xmm8 #st C[(ii+1)*dimj+vj+4:(ii+1)*dimj+vj+3] movups%xmm8,16(%rcx,%rsi) incq %r8 cmpq %r14, %r8 jb ..LOOP_K incq %rbx cmpq %r9, %rbx jb ..LOOP_J ... jb ..LOOP_I </pre>

Fig. 8. Register tiled matrix product after applying outer loop vectorization and unroll and jam to loop j. The left column shows the source code, the right the assembly.

their values depend on the available SIMD registers and their sizes on the target architecture. For simplicity and without loss of generalization, we assume  $\text{dim}_i$  and  $\text{dim}_j$  to be multiple of  $\text{BI}$  and  $\text{BJ}$ , respectively.

It is well-known that loop tiling [19] is a 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 Fig. 7a, it is necessary to fully unroll loops  $i$  and  $j$  to exploit the register level. At last, scalar replacement [6][7] 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  $\text{VL}$  and then permute the resulting element loop of  $\text{VL}$  iterations to become the innermost (the vector statement). The resulting code is shown in Fig. 7b assuming  $\text{BJ}$  is multiple of  $\text{VL}$  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 Fig. 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 Fig. 8, 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 Fig. 8, 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. Fig. 9 shows the resulting source code using pointers as temporary variables to identify the adjacent array references. We can see in Fig. 9 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.

Source code	ASM
<pre> long int ii, jj, k, vj; float *C1, *C2, *C3, *C4; const float *B1, *B2, *A1, *A2; for (ii = 0; ii &lt; dimi; ii+=2){   A1 = &amp;A[ii*dimk];   A2 = &amp;A[(ii+1)*dimk];   for (jj = 0; jj &lt; dimj; jj+=2*VL) {     C1 = &amp;C[ii*dimj+jj];     C2 = &amp;C[(ii+1)*dimj+jj];     C3 = &amp;C[ii*dimj+jj+4];     C4 = &amp;C[(ii+1)*dimj+jj+4];     for (k = 0; k &lt; dimk; k++) {       B1 = &amp;B[k*dimj+jj];       B2 = &amp;B[k*dimj+jj+VL];       #pragma ivdep       for(vj = 0; vj&lt;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];       }       /* vector statement */     }   } } </pre>	<pre> ..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,%xmm6 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 </pre>

Fig. 9. Register tiled matrix product after applying outer loop vectorization, unroll and jam and vectorial replacement. The left column shows the source code, the right the assembly.

## 5. PERFORMANCE RESULTS/EVALUATION

First, details of our evaluation environment are presented including a description of the architecture, compiler, execution decisions 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.

#### 5.1.1 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 I.

This machine also provides CPU throttling and hardware 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.

Table I. Memory hierarchy of the Intel Xeon E5520.

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

#### 5.1.2 Icc compiler

Our kernels and code were compiled using Intel C compiler [13] version 11.1 for intel64 architectures. This version includes several vectorization capabilities as well as memory hierarchy optimizations.

The ICC also provides some pragmas, flags and keywords to help the compiler to create more optimal codes. In our work we have compiled all kernels with the following flags:

- O3: enable the loop-intensive optimizations and vectorization.
- restrict: enable the "restrict" keyword for disambiguating pointers.
- fno-alias: assume no aliasing in the program.
- msse3: enable vectorization with SSE3 instruction set.

We have also used keyword "restrict" for all our function's headers. This keyword is used for pointer parameters or declarations and it indicate that for the lifetime of the "restricted" pointer, only it or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points. This limits the effect of the pointer aliasing.

- Moreover we have used two different pragmas in order to force ICC to vectorize:
- #pragma ivdep: instructs the compiler to ignore assumed vector dependencies.

#pragma vector always: instructs the compiler to override any efficiency heuristic during the decision to vectorize or not, and forces vectorization.

### 5.1.3 Execution decisions

Other decisions made for the performance measurements of the different kernels are listed below:

- 1.- Repeat several times the execution for a fixed problem size (at least 5 times, depending on the problem size).
- 2.- Flush all cache levels between executions.
- 3.- Use struct timeval, included in the sys/time.h, to estimate execution time with microsecond precision.

### 5.1.4 Kernels

Two groups of kernels have been evaluated to show the effectiveness of our proposal. The first group is composed by the kernels used in section 3 and 5 (cross addition of two vectors and matrix product) and the triangular matrix product. Table 2 contains a short description and the characteristics of each of them. The last 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 [19] and Index Set Splitting [39] to make sure that a particular loop performs a constant number of iterations. Table II also shows the vectorized dimension and the register tiling parameters when register tiling has been previously applied.

Table II. Characteristics of the first group of the evaluated kernels.

Description	Vectorized dimension	Reg. tiled dimensions	Reg. tile sizes	Loop depth	IS
Cross addition of 2 vectors (Fig. 1)	I	-	-	2	Rectangular
Rectangular matrix product (Fig. 6)	J	IR JR	IR = 6; JR = 8	3	Rectangular
Triangular matrix product (Fig. 10)	J	IR JR	IR = 6; JR = 8	3	Non-rectangular

```
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];
}
```

Fig. 10. Triangular matrix product.

The second group is composed by a subset of the Level 3 BLAS [20]. We apply vectorization and loop tiling at cache and register levels to these kernels in order to compare our proposal to hand-optimized assembly-written numerical libraries (the characteristics of these kernels are shown in Table III).

For the selection of the optimization parameters (vectorized dimension, tile dimensions and tile sizes at cache and register levels), we have develop a very simple heuristic that works very well for typical linear algebra problems (BLAS3 [20]). Remember that we use a combination of well-known high-level (source-to-source)

transformations. Of course, working at the source level prevents us from controlling many of the low level transformations typically performed by the compiler's back-end (instruction scheduling, register allocation, etc.). Therefore, our heuristic has been geared towards simplicity rather than trying to find optimal parameters, since there are too many aspects of the code generation process that escape from our control.

For selecting the vectorized dimension our heuristic use weighted spatial reuse vectors [36][38]. This weight is the number of references in the original loop body that have generated this reuse vector, considering reads and writes. We select the vectorized dimension that exposes more spatial locality (that is, dimensions whose corresponding reuse vector has the highest weight), avoiding dimensions whose corresponding loop traverses non-consecutive elements and minimizing reductions and replications of vector elements in the resulting vectorized codes.

For determining the tiling parameters for the register level, we consider that not tiling one loop that carries reuse reduces register pressure while maintaining data locality [23]. Moreover, by tiling more than one dimension of the iteration space, a reasonable amount of ILP is achieved. Thus, in our heuristics, we always tile two dimensions of the three-dimensional iteration space and focus on selecting the non-tiled loop that provides more temporal data locality, so that register reuse is improved. Then the tile sizes are computed in proportion to the quantity of reuse carried by each tiled direction, taking into account the number of available machine registers. If the tiled direction is also vectorized, the tile size must be multiple of the vector length in number of elements.

Finally, when tiling at multiple memory levels (cache and registers), the interaction between different levels must be considered [8]. We use MOB forms [23] when tiling at several levels. The basic rule in the construction of a MOB form is that the direction of blocks in adjacent levels should be different. The direction of a block is determined by the loop that is not tiled for this level. The orthogonality property of the MOB forms allows a sequential optimization to determine the order in which tiles are traversed and the size of the tiles level by level, beginning with the lowest level.

In our heuristic, we select as non-tiled loop at the cache level the one that provide more spatial reuse in order to minimize TLB misses. Then the tile sizes for the cache level are computed considering the available number of TLB entries and cache sizes. We select sizes that use less than 60% of the cache and do not exceed TLB entries. Moreover, tile sizes at the cache level should be multiple of tile sizes at the register level in the corresponding dimension.

Table III. Characteristics of the second group of the evaluated kernels.

Kernel	Vectorized dimension	Tiled dimensions (cache / register)	Cache tile size (square)	Register tile size	IS
SGEMM	J	IC KC / IR JR	120	IR = 6; JR = 8	Rectangular
STRMM	J	IC KC / IR JR	120	IR = 6; JR = 8	Non-rectangular
STRSM	J	IC KC / IR JR	120	IR = 6; JR = 8	Non-rectangular
SSYRK	K	IC KC / IR JR	192	IR = 3; JR = 4	Non-rectangular
SSYR2K	K	IC KC / IR JR	120	IR = 4; JR = 2	Non-rectangular

## 5.2 Performance Results

This subsection presents the performance results obtained by the two different groups of kernels.

For the first group of kernels (Table II), 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.

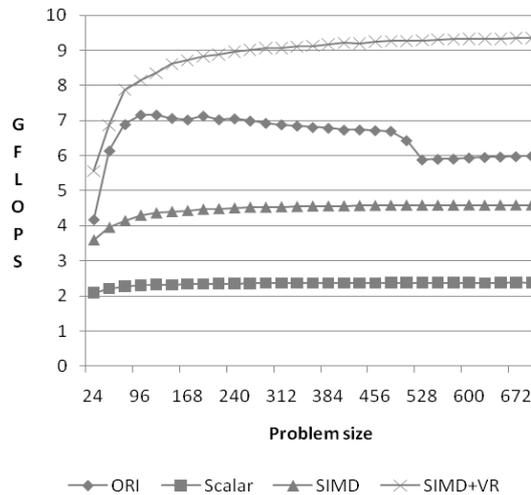


Fig. 11. Performance of cross addition of 2 vectors.

Fig. 11 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.

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 SIMD register reuse is done.

Fig. 12 shows the performance obtained for the rectangular matrix product. In the ORI version (code of Fig. 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 Fig. 12 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.

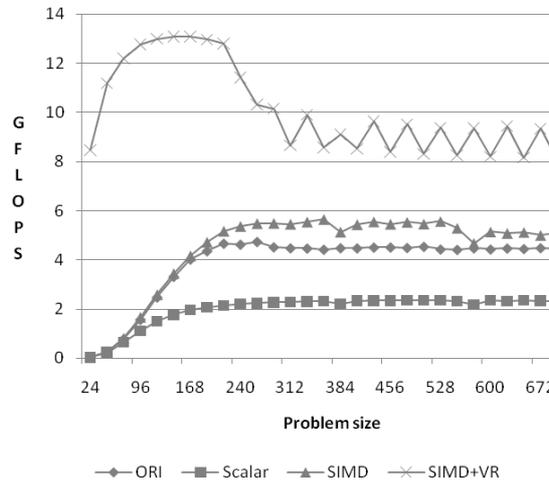


Fig. 12. Performance of rectangular matrix product.

Fig. 13 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 handle 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 [39] 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.

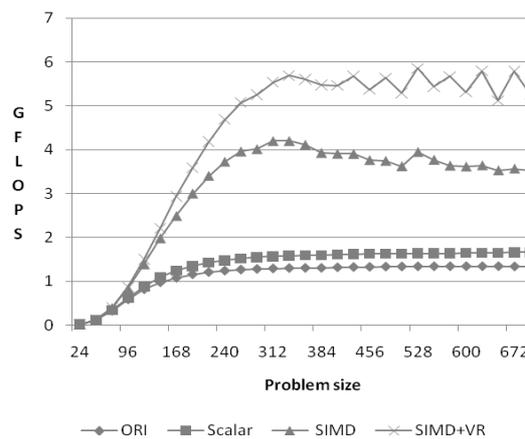
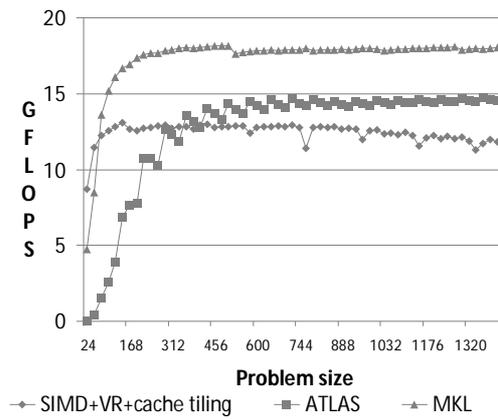
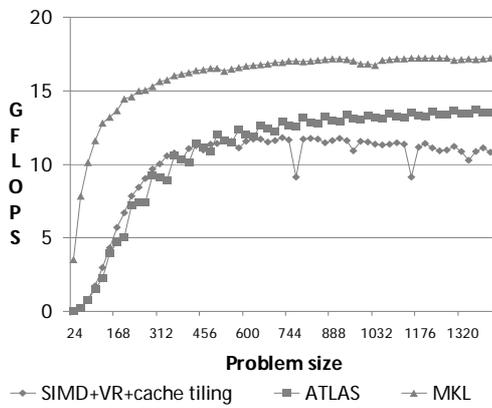


Fig. 13. Performance of triangular matrix product.

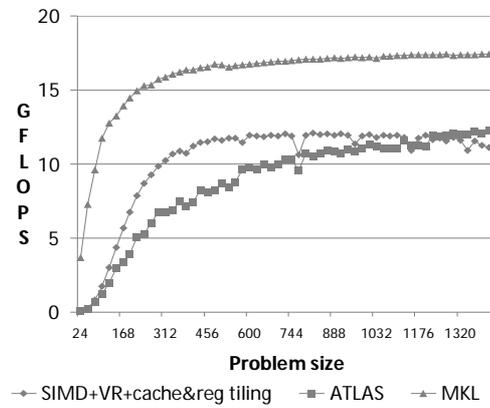
### SGEMM



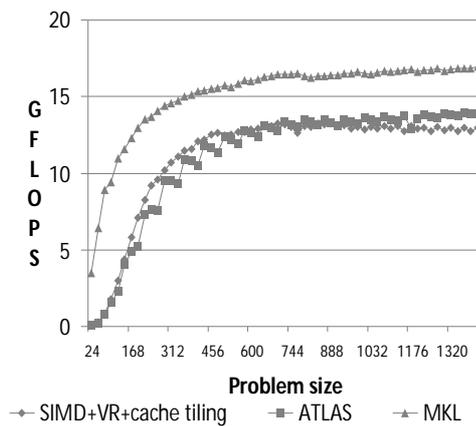
### STRMM



### STRSM



### SSYRK



### SSYR2K

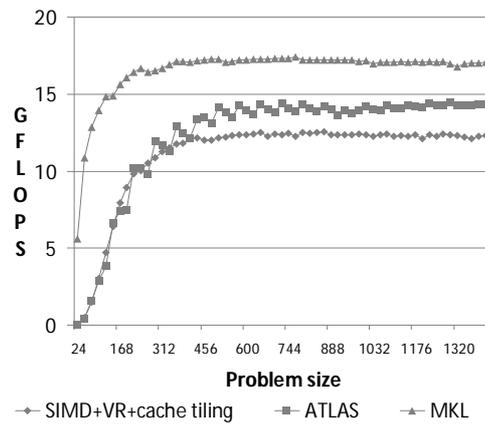


Fig. 14. Performance of STRSM, STRMM, SGEMM, SSYR2K and SSYRK for the ATLAS and MKL hand-optimized libraries and our best code (SIMD+VR + cache tiling).

In triangular matrix product 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 Fig. 13 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.

For the second group of kernels (Table III), we present the performance results obtained by our optimized codes and compare them to two hand-optimized assembly-written numerical libraries, ATLAS [35] and MKL [14], which include the Level 3 BLAS. As already mentioned and to do a fairly comparison with the numerical libraries, we perform cache tiling to this second group of kernels. Cache tiling is effective for reducing the capacity cache miss rate and moderating TLB misses [38]. Thus, for medium matrix sizes that do not fit at the cache level it achieves the same performance level as for smaller sizes.

In Fig. 14, we can see that for all benchmarks MKL almost achieves the peak performance of a core ( $2.26\text{GHz} * 4$  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 around 13 GFLOPS approximately (72% of the peak performance).

We can also observe in Fig. 14 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 performs better than ATLAS.

Summarizing, results show that we can help the compiler to generate efficient SIMD code by applying source-to-source transformations and 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.

Using reverse-engineering, we have shown in this paper that we can combine well-known high-level (source-to-source) transformation to help compilers to generate efficient SIMD code that exploits vector register reuse. 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. However, we have shown in Section 5 that we are able to achieve around 72% of MKL performance without writing in assembly language.

## 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, pp. 65-98, April 2002.
- [5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, A practical and fully automatic polyhedral program optimization system. *ACM SIGPLAN PLDI*, June 2008
- [6] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *PLDI '90*. pp. 53-65, June 1990.
- [7] S. Carr. *Memory-hierarchy management*. Ph.D. Thesis, Rice University, February 1993.
- [8] C. Chen, J. Shin, S. Kintali et al.. Model-Guided Empirical Optimization for Multimedia Extension Architectures: A Case Study. *IPDPS'07*, pp. 1-8, March 2007.
- [9] G. Cheong, M.S. Lam, An optimizer for multimedia instruction sets, in: *The Second SUIF Compiler Workshop*. Stanford University, USA, 1997.
- [10] Y.F. Fung, M.F. Ercan, T.K. Ho, and W.L. Cheung. A parallel solution to linear systems. *Microprocess. Microsyst.*, 26, pp. 39-44, 2002.
- [11] A. Hartono, B. Norris, P. Sadayappan.: Annotation-based empirical performance tuning using Orio. *IPDPS'09*, May 2009
- [12] 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): pp. 630-649, January 2008.
- [13] Intel Corporation. *Intel C/C++ Compiler User and Reference Guide*. Order Number 304968-023US.
- [14] Intel Corporation. *Intel Math Kernel Library Reference Manual*. Order Number 630813-038US.
- [15] Intel Corporation (2010). *Intel Advanced Vector Extensions Programming Reference*. Order Number 319433-009, December 2010.
- [16] M. Jimenez, J. Llberia, A. Fernandez. On the Performance of Hands vs. Automatically Optimized Numerical Codes. *HPCA-6 IEEE Computer Society 2000*, pp. 183-194, January 2000.
- [17] M. Jimenez, J. Llberia, A. Fernandez. Register Tiling in Nonrectangular Iteration Spaces. "ACM transactions on programming languages and systems" 2002, vol. 24, núm. 4, pp. 409-453, July 2002.
- [18] A. Krall and S. Lelait. *Compilation Techniques for Multimedia Processors*. *Int. J. of Parallel Programming*. 28, 4, pp. 347-361, August 2000.
- [19] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimization of Blocked Algorithms. *ASPLOS'91*, pp. 63-74, 1991.
- [20] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5 1979, pp. 308--323.
- [21] D. Maydan, J. Hennessy and M. Lam. Efficient and exact data dependence analysis. *PLDI'91*, pp. 1-14, June 1991.
- [22] 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.
- [23] J.J Navarro, T. Juan, T. Lang. MOB forms: a class of multilevel block algorithms for dense linear algebra operations. *ICS'94*, pp. 354-363, July 1994.

- [24]D. Nuzman, A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. PACT '08, pp.2-11, 2008.
- [25]A. Peleg, U. Weiser. MMX Technology Extension to the Intel Architecture, IEEE Micro, Vol. 16, No. 4, pp. 42-50, August 1996.
- [26]S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. GRAPHITE: Polyhedral Analyses and Optimizations for GCC. In Proceedings of the 2006 GCC Developers Summit, 2006.
- [27]L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. Iterative optimization in the polyhedral model: Part II, multidimensional time. PLDI'08, Tucson, Arizona, June 2008.
- [28]I. Pryanishnikov , A. Krall , N. Horspool. Compiler optimizations for processors with SIMD instructions. Software—Practice & Experience, v.37 n.1, pp.93-113, January 2007
- [29]W. Pugh. A practical algorithm for exact array dependence analysis. Communications of the ACM, Vol. 35, No. 8, pp. 102-114, 1992.
- [30]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.
- [31]L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. SC '09, pp. 1-12, 2009.
- [32]J. Shin, J. Chame and M. W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. PACT '02, pp. 45-55, September 2002.
- [33]N. Sreeraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. J. of Parallel Programming, 28, pp. 363-400. 2000.
- [34]K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. PACT'09, September 2009.
- [35]R. C. Whaley, A. Petitet, J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project, Parallel Computing, 27(1-2):3-35, 2001.
- [36]M. Wolf and M. Lam. A data locality optimizing algorithm. PLDI'91, pp. 30-44, June 1991.
- [37]M. Wolf, D. Maydan and D.K. Chen. Combining loop transformations considering caches and scheduling. MICRO-29, pp. 274-286, December 1996.
- [38]M. Wolf. Improving locality and parallelism in nested loops. Ph.D. Thesis, Stanford University, 1992.
- [39]M. Wolfe. High performance compilers for parallel computing. Addison-Wesley Publishing Company, 1996.
- [40]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), pp. 934-946, September 2000.
- [41]Y. Zhao and K. Kennedy. Scalarization on short vector machines. ISPASS'05, March 2005.