SIMD Programming by Expansion

by

Jaewook Shin

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA

This work was supported by the Mathematical Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357

SIMD Programming by Expansion

Abstract

Since its advent 30 years ago, single-instruction multiple-data (SIMD) functional units continue to provide an opportunity for high performance at a low hardware cost. However, a general consensus is that only a class of well-formed computations is suitable for SIMD execution. We believe that the boundary of the class should be pushed so that more applications can get the benefit of SIMD parallelism. Our goal is to provide programmers tools that will allow easier access to SIMD functional units. In this paper, we describe a new method to generate SIMD instructions automatically. Unlike the current approaches that target either loops or basic blocks, our approach targets a whole function. Instead of trying to keep the sequential execution semantics, we semantically transform the given input function by replacing the operators and operands with their SIMD counterparts. The output functions generated this way take vector arguments and return a vector value. We have implemented the new method in a compiler, called EXPAND, and show how to use it for user applications. To demonstrate the effectiveness of the new method, we apply the EXPAND compiler to 12 GNU math library intrinsic functions. When measured on a PowerPC G5, the transformed output codes achieve speedups ranging from 2.05 to 11.37 over the scalar baseline.

1 Introduction

Single-instruction multiple-data (SIMD) functional units are employed in most modern microprocessors because of the potential for high performance and relatively low hardware cost [8, 10, 15, 20]. However, automatic vectorization by compilers remains far from satisfactory. Therefore, programmers commonly write SIMD instructions manually, either at assembly level or at a higher level using SIMD extensions to programming languages. However, writing parallel programs manually is not easy, is error prone, and undermines the portability of the code. Our goal is to help programmers generate portable parallel programs by providing software tools such as compilers.

Two approaches are commonly used to generate SIMD instructions automatically. The first is to adapt the vectorization technique developed for the conventional vector machines. In this approach, each loop is examined for the possibility of vectorization. If vectorizing the loop is both possible and profitable, vectorization is performed on the loop. More recently, a new technique has been developed to generate SIMD instructions from basic blocks [7, 6, 9]. In this approach, loops are unrolled to increase the amount of SIMD parallelism. Currently, several compilers can generate SIMD instructions automatically [7, 2, 13, 5]. For such compilers to generate efficient SIMD code, however, the input program must not have certain factors that can limit vectorization. These factors include alignment, irregular memory accesses, unknown loop bounds during run time, function calls, true data dependency, and not enough SIMD parallelism. To address these problems, we need to develop new techniques, or even a completely different approach.

In this paper, we describe a new approach to program SIMD functional units. Unlike the current techniques that target loops or basic blocks, our approach targets whole functions. Furthermore, our approach is a semantic transformation by which the sequential input function is transformed to a parallel version. We use a simple example below to illustrate how this transformation works. For a sequential function in (a) given as an input, the output of this transformation is the parallel version shown in (b).

```
int scalar_add(int a, int b)
{
    return a + b;
}
```

(a) scalar input

```
vector int vector_add(vector int va, vector int vb)
{
    return vec_add(va, vb);
}
```

(b) vector output

In this transformation, scalar operators and operands in the input function are replaced with their vector counterparts. Consequently, the original scalar semantics remain as the operations applied to an element of the vectors in the output code, and the output code as a whole performs a sequence of vector operations, each of which is the vector counterpart of each operation in the scalar input code.

We call this transformation *expansion*, and we say that the given input function is *expanded* when it is transformed this way because this transform resembles *scalar expansion* in conventional vectorization.

Unlike scalar expansion, however, independent scalar operands can be processed in different vector lanes when expanded. Although we have presented a simple example, our compiler can expand functions that have arbitrarily complex control flow with multiple return-statements and function calls. The applicability of this transformation is significantly expanded thanks to a new technique that can generate efficient SIMD code in the presence of control flow [16].

We suggest this approach as a general programming paradigm for SIMD functional units. Programmers can *outline*[1] the loopbody of a loop as a separate function that does not have any side effect and can use our compiler to generate the corresponding vector function. This vector function can be called from the original loop using vector arguments or inlined back to the loop. We implemented the proposed approach in a compiler named *EXPAND*. We demonstrate the effectiveness of the suggested approach by applying the compiler to 12 GNU math library intrinsic functions.

The contributions of this paper are summarized below.

- Presentation of a new approach to program SIMD functional units
- Development and implementation of the algorithm for the proposed approach
- Experimental evaluation of the implementation on 12 math intrinsic functions

In the next section, we describe the terms and concepts necessary to understand the later sections. We introduce our approach in Section 3 and describe the algorithm in Section 4. In Section 5, we present our implementation and the experimental results on 12 math intrinsic functions. In Section 6, we discuss related work, and in Section 7, we summarize this paper and discuss the future work.

2 Background

We begin by describing background information about vectorization in the presence of control flow.

[1]Outlining is the reverse transformation of function inlining.

2.1 Vectorization

A vector register has multiple scalar datapaths side by side. The maximum number of scalar values that can be held in a vector register is called *vector length*. In modern microprocessors, the vector register width is fixed, but the vector length varies depending on the type size of the scalar value. For example, PowerPC G5 has 32 128-bit vector registers. Each vector register can hold 16 8-bit operands, 8 16-bit operands, or 4 32-bit operands. We say a vector memory access is *aligned* if the issued address is always a multiple of the vector register width. Many modern SIMD architectures do not support strided memory accesses where the data elements are not contiguous in memory. To load such noncontiguous data elements into a vector register, they have to be copied into contiguous memory addresses before being loaded by using a vector load. This process of gathering data elements into a contiguous, aligned memory is called *packing*. *Unpacking* moves data elements from the packed locations back to their original locations. Figure 1 illustrates these two operations. For 32-bit data elements and 128-bit vector registers, a vector register can hold four data elements. Since moving one data element takes one scalar load and one scalar store (one arrow in Figure 1(a)), moving all four elements takes eight scalar memory accesses. One more vector load at the end completes the packing operation. Likewise, the unpacking operation takes the same number of scalar memory accesses but one vector store before the scalar memory accesses for the same conditions.

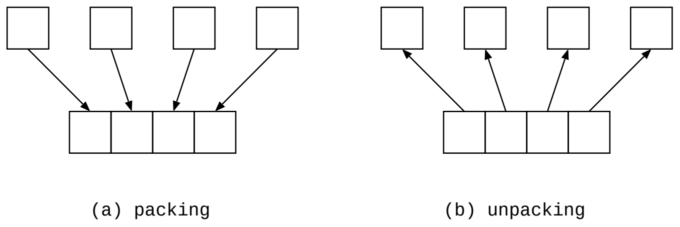


Figure 1: Packing and unpacking.

Vectorization is a process of transforming the given scalar code into a vector code to use vector registers and vector functional units, while keeping the original semantics of the scalar code. If this transformation is performed automatically by the compiler, it is called automatic vectorization.

2.2 If-conversion

If-conversion removes control flow by introducing predicates. For vectorization, if-conversion plays a crucial role when there is control flow. Since the scalar code might take different control paths for the vector length consecutive executions, for vectorization the statements in all control paths are executed in vector mode and the values are merged using the vector predicates representing the taken control path information. Given a predicated instruction, the destination operand is updated with the result of the instruction if the predicate is true. Otherwise, the result of the instruction is ignored, and the destination operand remains unchained. We use Park and Schlansker's RK-algorithm, which is optimal in number predicates and predicate defining instructions [14].

2.3 `Select` Instruction

After if-conversion is applied to remove control flow, the scalar code forms a large basic block of predicated instructions. Then, vectorization is performed by replacing each scalar operator and operand with the vector counterparts. At this stage, the basic block contains vector instructions guarded by vector predicates and can be executed if the machine supports predicated execution. For modern vector architectures that do not support predicated execution, however, the vector predicates must be removed. To remove vector predicates, we use `select` instructions that are common in modern vector ISAs [18].

2.4 Branch-on-superword-condition-code

Branch-on-superword-condition-code (BOSCC) is a branch instruction that can be conditionally taken based on the comparison result of vector variables. AltiVec supports BOSCC instructions with *AltiVec predicates* [11]. For example, the vector predicated instruction

```
Vdst = vec_operation; <Vpred>
```

can be bypassed by introducing a BOSCC instruction as follows.

```
NotTaken = vec_any_ne(Vpred, ZeroVector)
if (NotTaken) { Vdst = vec_operation; <Vpred> }
```

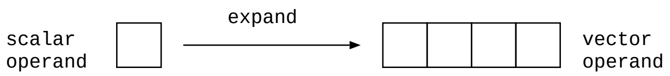
The `vec_any_ne` instruction returns *true* if *any* field of `Vpred` does not match the corresponding field of `ZeroVector`. Assuming `ZeroVector` contains *false* values in all fields, `NotTaken` will be set to *false* only when all fields of `Vpred` are false. We use *BOSCC region* to refer to the sequence of instructions enclosed by a BOSCC. A BOSCC region may contain a large number of instructions including even other BOSCCs, enabling the vector code to bapass unnecessary instructions much like the scalar branches in the scalar baseline [17, 16].

3 EXPAND

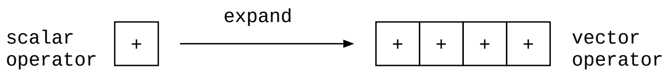
In Section 1, we briefly introduced the concept of a new transformation called *expansion*. In this section, we formally define *expand* and describe its features in relation with other existing approaches.

Definition 1 *A scalar operand, operator, statement, and function are said to be* expanded *when they are replaced by the vector counterpart.*

When a scalar operand is expanded, it is replaced by a vector operand whose elements have the type of the original scalar operand. When a scalar operator that takes n scalar source operands is expanded, it is



replaced by the corresponding vector operator that takes n vector operands. The effect of the vector operator is to apply the original scalar operator to each element of the vector operands. When a scalar statement of

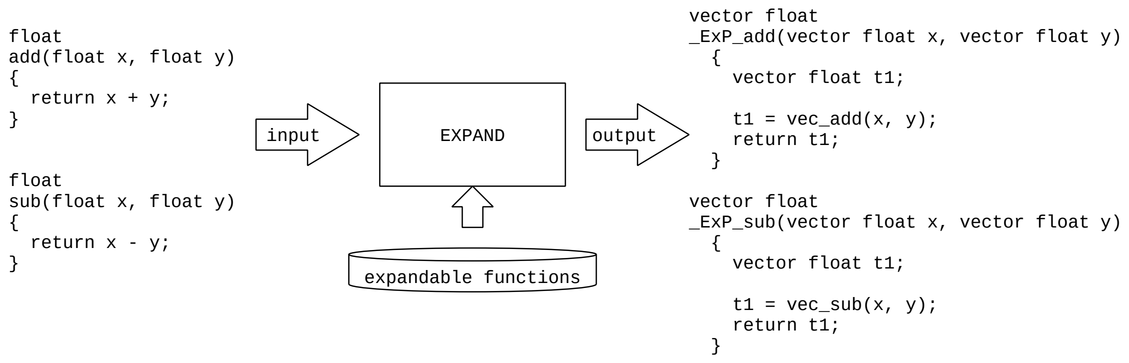


a sequential program is expanded, the consisting operands and operators are expanded to form an expanded vector statement. Likewise, when a function is expanded, all statements in the function are expanded to make a vector function that takes vector arguments and returns a vector value. For pointer type operands, we expand the data objects being pointed to; that is, their reference types are expanded.

Intuitively, given a single scalar data path, the expansion transformation replicates the scalar data path as many as the vector length times. The replicated data paths are executed at the same time, but they never interfere with each other.

3.1 EXPAND: A Compiler That Expands Functions

EXPAND is the name of a compiler that we have developed, and performs the *expansion* transformation of the given functions. Figure 2 shows the input and output files together with the compiler. The input file may contain multiple functions, and the functions in the input file are expanded as shown in the figure. Since each function in the same input file is processed independently as if it is a unique function in the input, in the remainder of this paper we focus on expanding a single function. In addition to the input and



```
float
add(float x, float y)
{
  return x + y;
}

float
sub(float x, float y)
{
  return x - y;
}
```

```
vector float
_ExP_add(vector float x, vector float y)
  {
    vector float t1;

    t1 = vec_add(x, y);
    return t1;
  }

vector float
_ExP_sub(vector float x, vector float y)
  {
    vector float t1;

    t1 = vec_sub(x, y);
    return t1;
  }
```

Figure 2: The EXPAND compiler at work.

output files, EXPAND takes a list of expandable functions. This list provides information as to whether the functions called from inside the input function should be redirected to their expanded versions. If a called function is expandable, we pass expanded function arguments and expect an expanded return value as well. Otherwise, we perform unpack and pack operations to use in the call to the scalar function. This point is illustrated in the next subsection by an example.

Currently, only those input functions that satisfy the following three conditions can be expanded by our compiler. First, loops are not allowed in the input function. Second, nonconstant array subscript expression values are not supported. Third, only scalar type function arguments are allowed. These conditions are for the convenience of implementation, however, and we expect that they will be lifted as we improve our implementation. For example, nonconstant array subscript expressions can be supported by accessing them in scalar mode as we do for nonexpandable function calls, as illustrated in the next subsection.

```
#define GET_FLOAT_WORD(i,d)\
do { union { \
        float value; \
        int word; \
    } gf_u; \
    gf_u.value = (d);\
    (i) = gf_u.word; \
} while (0);
static const float two = 2.0;
static const float bp[] = {1.0, 1.5};

void mot(float x, float *y) {
    int hx, ix, k;

    GET_FLOAT_WORD(hx,x);
    ix = hx&3;
    if(ix¡=7) k=0;
    else k=1;

    if(ix==0) {
        *y = powf(bp[k], two);
        return; }
    *y = sqrtf(ix); }
```

(a) Scalar original

```
void mot(float x, float *y) {
    do {
        union __tmp_union1 { float value;
            int word; };
        union __tmp_union1 gf_u;
        gf_u.value = x;
        hx = gf_u.word;
    } while (0);
    ix = (unsigned int)hx & 3u;
    if (ix ¡= 7) {
        k = 0;
        bp_k = *bp;
    } else {
        k = 1;
        bp_k = bp[1];
    }
    if (ix == 0) {
        *y = powf(bp_k, two);
        return;
    }
    *y = sqrtf(ix);
    return;}
```

(b) After making array subscripts constants

```
void mot(float x, float *y) {
    union __tmp_union1 { float value;
        int word; };
    union __tmp_union1 gf_u;

    gf_u.value = x;
    hx = gf_u.word;
    ix = hx & 3;
    t2 = ix ¡= 7;
    if (!t2) goto L4;
    bp_k = *bp;
    goto __done6;
    L4:
    bp_k = bp[1];
    __done6:
    t4 = ix == 0;
    if (!t4) goto L5;
    *y = powf(bp_k, two);
    goto EXIT;
    L5:
    *y = sqrtf(ix);
    EXIT: return; }
```

(c) After making it single-entry, single-exit

```
void _ExP_mot(vector float x, vector float *y){
    union __tmp_union1 { vector float value;
    vector signed int word; };
    union __tmp_union1 gf_u;

    vec_st(x, 0, (float *)&gf_u.value);
    hx = vec_ld(0, (int *)&gf_u.word);
    t = (vector signed int)(3, 3, 3, 3);
    ix = vec_and(hx, t);
    t1 = (vector signed int)(7, 7, 7, 7);
    p8 = vec_cmplt(t1, ix);
    t9 = (vector bool int)(0, 0, 0, 0);
    p2 = vec_nor(p8, t9);
    bp_k = vec_ld(0, (const float *)bp);
    t11 = vec_ld(16, (const float *)bp);
    bp_k = vec_sel(bp_k, t11, p8);
    t3 = (vector signed int)(0, 0, 0, 0);
    p4 = vec_cmpeq(ix, t3);
    t8 = (vector bool int)(0, 0, 0, 0);
    p7 = vec_nor(p4, t8);
    t6 = _ExP_powf(bp_k, two);
    t12 = vec_ld(0, y);
    t12 = vec_sel(t12, t6, p4);
    vec_st(t12, 0, y);
    t5 = vec_ctf(ix, 0);
    *(float *)&t7 = sqrtf(*(float *)&t5);
    ((float *)&t7)[1] = sqrtf(((float *)&t5)[1]);
    ((float *)&t7)[2] = sqrtf(((float *)&t5)[2]);
    ((float *)&t7)[3] = sqrtf(((float *)&t5)[3]);
    t13 = vec_ld(0, y);
    t13 = vec_sel(t13, t7, p7);
    vec_st(t13, 0, y);
    return; }
```

(d) After predication and *expansion*

```
static vector float two = {2.0, 2.0, 2.0, 2.0};
static vector float bp[2] =
{(vector float){1.0,1.0,1.0,1.0},(vector float){1.5,1.5,1.5,1.5}};

void _ExP_mot(vector float x, vector float *y) {
    hx = (vector signed int)x;
    t9 = (vector bool int)(0, 0, 0, 0);
    ix = vec_and(hx, (vector signed int)(3, 3, 3, 3));
    p8 = vec_cmplt((vector signed int)(7, 7, 7, 7), ix);
    p4 = vec_cmpeq(ix, (vector signed int)(0, 0, 0, 0));
    p2 = vec_nor(p8, t9);
    if (vec_any_ge((vector signed int)(7, 7, 7, 7), ix)) {
        bp_k = vec_ld(0, (const float *)bp); }
    if (vec_any_ne(p8, t9)) {
        bp_k = vec_sel(bp_k, vec_ld(16, (const float *)bp), p8); }
    t14 = (vector bool int)vec_splat_u32(0);
    if (vec_any_ne(p4, t14)) {
        t12 = vec_ld(0, y);
        t6 = _ExP_powf(bp_k, two);
        t12 = vec_sel(t12, t6, p4);
        vec_st(t12, 0, y); }
    if (vec_any_ne(ix, (vector signed int)(0, 0, 0, 0))) {
        p7 = vec_nor(p4, t9);
        t5 = vec_ctf(ix, 0);
        *(float *)&t7 = sqrtf(*(float *)&t5);
        ((float *)&t7)[1] = sqrtf(((float *)&t5)[1]);
        ((float *)&t7)[2] = sqrtf(((float *)&t5)[2]);
        ((float *)&t7)[3] = sqrtf(((float *)&t5)[3]);
        t10 = vec_ld(0, y);
        t13 = vec_sel(t10, t7, p7);
        vec_st(t13, 0, y);
    }
    return; }
```

(e) After inserting BOSCCs

Figure 3: A motivating example.

3.2 EXPAND by an Example

To describe how EXPAND works, we use an example, shown in Figure 3(a). The example is designed to have the features frequently found in GNU math library implementations.

First, we transform all array accesses to have constants in their subscript expressions. Although array accesses with nonconstant subscript expressions can be accessed in scalar mode, it is still faster to access them in vector mode whenever possible. For this transformation, all array subscript expressions should have constant reaching definitions. If so, we copy the array access next to the definition of each subscript expression. The code generated by this transformation is shown in Figure 3(b). An array access $bp[k]$ is replicated in each definition of the subscript expression, k. Then the subscript expression is replaced with the corresponding definition, which is a constant. A new variable bp_k is assigned at each replication point and used in the original use of the array access. At the final code shown in Figure 3(e), these two array accesses with constant subscript expressions are converted to vector loads from the expanded global variable bp.

Next, we replace `return` statements in the middle of the function with `goto` statements jumping to the label at the end of the function. This transformation results in a control flow graph with single entry and single exit. The use of the RK-algorithm [14] for if-conversion mandates the control flow graph of the input function to have single entry and single exit. In the transformed code in Figure 3(c), the return statement in the middle is replaced with a `goto` statement branching to the label `EXIT`. If the function returns a value, an assignment statement is inserted before the `goto` statement. In the assignment, the value returned at that point is assigned to the variable returned at the end.

Figure 3(d) shows the code after it is predicated and expanded. Types of all variables are changed to the corresponding vector types, and all operators are replaced with the vector counterparts, except for the function call to `sqrtf`. In this example, we assume that `sqrtf` is not in the list of expandable functions. All such nonexpandable functions have to be accessed in scalar mode. Thus, `sqrtf` is called as many as the vector length times, 4 in this example, with scalar arguments and scalar destination operands. For the scalar function arguments, an expanded variable `t5` is unpacked, and the scalar return values are packed back into another expanded variable `t7`. Note that we did not rename `sqrtf`, whereas others such as `powf` and the input function itself are renamed by prepending `_ExP_`. This function renaming is necessary to distinguish the expanded function from the original scalar function.

Finally, we remove redundant computations and insert BOSCCs to bypass the statements that are unnecessary during run time. The generated code is shown in Figure 3(e). Note that a pair of vector store and load to the address of `gf_u` is replaced with a type conversion statement. A load preceded by a store statement to the same address can be replaced by the operand being stored as long as no other memory accesses to the same address intervene between them. The remaining stores into the address of local variables can be deleted if there is no use of the variables or loads from the same address. `GET_FLOAT_WORD` in (a) is a macro used to move values from floating point registers to integer registers. For scalar registers, this data movement goes through memory because there is no direct datapath between different register files. However, changing interpretation[2] of bits in vector registers does not require any additional operation. The two global variables in Figure 3(a) have definitions. These global variables are read only, and their definitions are expanded as shown in Figure 3(e).

3.3 Programming by EXPAND

In this subsection, we introduce SIMD programming using the EXPAND compiler. Although we like to leave the future possibilities of EXPAND open, the current implementation has certain limitations. For example, loops cannot be expanded. Thus, one model we present in this subsection is to *outline* the loopbody of the compute intensive loop into a function. The outlined function is subsequently expanded by EXPAND. Now, the loop is modified to call the expanded function with expanded arguments. We use an example in Figure 4(a) to illustrate this procedure.

There are several benefits in the p

The loop in Figure 4(a) implements a simple chroma keying. The function in Figure 4(b) shows the outlined function, which is in turn expanded by EXPAND to produce `_ExP_chroma` similar to the one in Figure 3(e). Figure 4(c) shows the loop modified to use the expanded function. In this programming model, arrays are accessed outside the function and passed as function arguments. Compared with the scalar

[2]whether the bits should be interpreted as an integer or a floating-point value

```
for(i=0; i< npix; i++){
    fred = f_red[i];
    fgreen = f_green[i];
    fblue = f_blue[i];

    if(fred != 0 || fgreen != 0 \
        || fblue != 255){
        b_red[i] = fred;
        b_green[i] = fgreen;
        b_blue[i] = fblue;
    }
}
```

(a) Scalar original

```
void chroma(unsigned char fred,
    unsigned char fgreen, unsigned char fblue,
    unsigned char* bred, unsigned char* bgreen,
    unsigned char* bblue){
    if(fred != 0 || fgreen != 0 || fblue != 255){
        *bred = fred;
        *bgreen = fgreen;
        *bblue = fblue;
    }
}
```

(b) Loopbody outlined into a function

```
for(i=0; i< npix; i+=16){
    vfred = vec_ld(0, &f_red[i]);
    vfgreen = vec_ld(0, &f_green[i]);
    vfblue = vec_ld(0, &f_blue[i]);
    vbred = vec_ld(0, &b_red[i]);
    vbgreen = vec_ld(0, &b_green[i]);
    vbblue = vec_ld(0, &b_blue[i]);

    _ExP_chroma(vfred, vfgreen, vfblue,
        &vbred, &vbgreen, &vbblue);

    vec_st(vbred , 0, &b_red[i] );
    vec_st(vbgreen, 0, &b_green[i]);
    vec_st(vbblue , 0, &b_blue[i] );
}
```

(c) Loop modified to call the expanded function

Figure 4: An example to show how to program by EXPAND.

baseline, this implementation has the disadvantage of always accessing the three arrays of the background image: `b_red`, `b_green`, and `b_blue`. While multiple output operands can be passed as pointer type arguments as in this example, if there is a single output, the return value can be used to get the output instead of using the pointer of the output variable in the function argument list. Since the output of EXPAND is also in C, programmers can use the expanded code for further optimization. For example, the expanded output function can be inlined back into the original loop to eliminate the function call overhead.

Programming by EXPAND provides several benefits over the current approaches. First, function calls are no longer a barrier. As described in the previous subsection, the compiler can expand the whole input function, and the function calls within it can also be expanded to call the expanded functions if they are specified in the expandable function list. Moreover, this approach avoids dealing with the problem of alignment. In the example code shown in Figure 4(c), the aligned memory accesses are used, assuming that the array objects start at the aligned addresses. If the array objects cannot be guaranteed to have aligned addresses, the explicit alignment operations must be inserted manually. If the data elements are scattered in memory, they must be packed into an aligned contiguous memory location before being passed to the expanded function. While this avoidance of the alignment problem simplifies the compiler, at the same time it places more burden on the users. Since the alignment problem is one of the major limiting factors in contemporary vectorizing compilers, however, requiring a small effort for alignment by users can be paid off with a larger class of SIMD applications by completely relieving the compiler of the alignment problem. In addition, the EXPAND compiler does not need to check for data dependence between the vector elements. The independence among the data elements of the expanded code is guaranteed by the definition of *expansion* and the way the compiler is implemented.

3.4 Discussion

The current practice of generating SIMD instructions is either a fully manual SIMD programming or a fully automatic generation of SIMD instructions by compilers. Both of these extreme approaches have some benefits and some drawbacks. For example, fine control over the generated SIMD code is a benefit of fully manual programming but at the same time a drawback of automatic vectorization. Our approach positions itself between the two extremes. We assume that users play a role in providing aligned memory addresses and in modifying loops. This role relieves the compiler of major hurdles that have been critical for automatic vectorization. Yet, most of the time-consuming and error-prone SIMD translation is performed automatically by the EXPAND compiler. For the simple example shown in Figure 4, users may not save much time by using the EXPAND compiler. The more complicated the input function, however, the more time and effort saved by users, as with any other automated approach.

Our approach is applicable to a wide range of applications thanks to Shin et al.'s technique for generating efficient code in the presence of complex control flow [16]. Without this technique, the expanded code must execute all control flow paths, possibly leading to slowdowns compared with the scalar baseline that might bypass a large portion of the code.

The best feature of our approach is simplicity. The concept of expansion is simple and clear, and the expand transformation is nothing more than replacing each operator and operand with a vector counterpart. Because of this simplicity and the high-level language output, users can easily understand what is happening in the transformation process and hence can further optimize the expanded code.

4 Algorithm

In this section, we present the algorithm for our approach.

Figure 5 shows the component passes in the EXPAND compiler. Initially, we examine each array reference

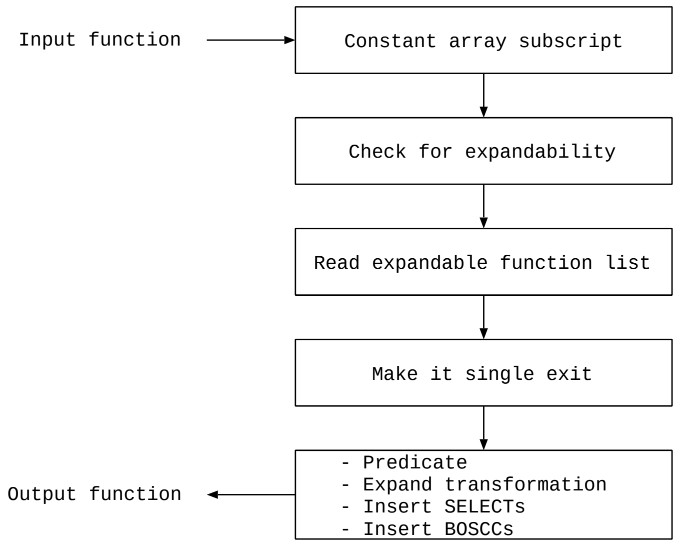


Figure 5: Flow of the algorithm.

to see whether its subscript expression can be transformed to a constant. For each nonconstant array subscript expression, we identify its reaching definitions. If all reaching definitions are constants, we hoist the array reference to where the subscript expression is defined and replace the subscript expression with the corresponding constant. Also, a new variable is created and used at the use of the array reference to convey the value of the array reference.

Next, we check whether the given input function can be expanded. Currently, this pass checks for three conditions. First, loops are not allowed in the input function. Second, nonconstant array subscript expressions are not allowed; however, this restriction can be removed by accessing the arrays in scalar mode and packing them into a vector operand. Third, only scalar arguments are allowed in the function argument list; that is, only integer, float, enumumeration, and pointer types are allowed for the arguments. We expect that many of these restrictions will be eliminated as we improve the implementation. If an input function is expandable, this pass adds an annotation to the function. All later passes examine the annotation to see whether the function should be processed further for expansion.

A list of expandable functions is provided by users. Although the expandability of the input function itself is determined by the compiler, the expandability of those functions called from inside the input function should be given.

Functions might have multiple return-statements resulting in a control flow graph with multiple exit nodes. Since we use the RK-algorithm for predication and it requires the control flow graph to have a single entry and single exit, we replace all return-statements in the middle of the input function with goto-statements branching to the label at the end of the function. When a return-statement is replaced, the return value is assigned to the variable returned at the end. After this transformation, the control flow graph will be still acyclic but may not be structured anymore.

Finally, the expand transformation is performed to expand scalar statements to vector statements. A major work in this transformation is to change the scalar types of operands to the corresponding vector types, and scalar operators with the vector counterparts. If the input function contains control flow, it is predicated to remove the control flow before being expanded, and `select` instructions are inserted to remove predicates so that the output code can be executed on the architectures that do not support predicated execution. However, predicate information is preserved as annotations for the BOSCC insertion passes. A BOSCC is generated whenever the BOSCC region contains instructions with nonzero costs [17, 16]. For the calls to the expandable functions, the function names are modified to use the expanded function, and function arguments are also expanded. For the calls to nonexpandable functions, we unpack the function arguments, and the function call is replicated as many as vector length times using the unpacked scalar operands. In this case, the output operands and the return value are also scalar and are packed back into vector operands for use in subsequent vector operations. The name and type of the input function are modified to reflect the transformation so that it takes vector arguments and returns a vector value if there is any.

5 Experiments

To evaluate the effectiveness of the suggested approach, we implemented the algorithm described in the previous section and applied it to 12 GNU math library intrinsic functions. In this section, we describe our implementation, the experimental environment, the intrinsic functions used in the experiments, and the experimental results.

5.1 Implementation

The algorithm described in Section 4 is implemented by using the SUIF compiler infrastructure [4]. The boxes representing the component algorithms in Figure 5 also closely match the individual passes in our implementation. Input to the compiler is a sequential code written in C, and the output is also a C code augmented with vector intrinsics [11].

Table 1: Benchmark programs.

Name	Description	source file	# lines
isinff	nonzero if argument is infinite	s_isinff.c	29
finitef	nonzero if argument is finite	s_finitef.c	35
fabsf	absolute value of floating-point number	s_fabsf.c	39
isnanf	is not-a-number	s_isnanf.c	42
copysignf	copy sign of a number	s_copysignf.c	42
truncf	round to integer, towards zero	s_truncf.c	52
ceilf	ceiling function	s_ceilf.c	62
scalbnf	multiply floating-point number by integral power of radix	s_scalbnf.c	64
floorf	largest integral value not greater than argument	s_floorf.c	71
roundf	round to nearest integer, away from zero	s_roundf.c	73
logf	natural logarithmic function	e_logf.c	99
powf	power function	e_powf.c	258

5.2 Methodology

In order to evaluate our implementation, we used 12 intrinsic functions taken from GNU math library glibc-2.4. Using our EXPAND compiler implementing the proposed approach, we automatically expanded the intrinsic functions listed in Table 1. For convenience, we did not use the profitability model that requires profile information [17]. Instead, BOSCCs are generated for all BOSCC regions that have at least one statement. For all intrinsic functions, the element data size is 32-bits. Thus, four scalar elements are processed in one vector operation.

For each intrinsic function, we manually created two driver programs, one for the scalar version and the other for the expanded version. In the driver for the vector version, we used aligned vector memory accesses by aligning the array objects, as the users of this approach would do whenever possible. In the same sense, the return value is stored back, also using the aligned vector stores. For both driver programs, we used the same sequence of uniform random numbers for the function input arguments. To compile the expanded

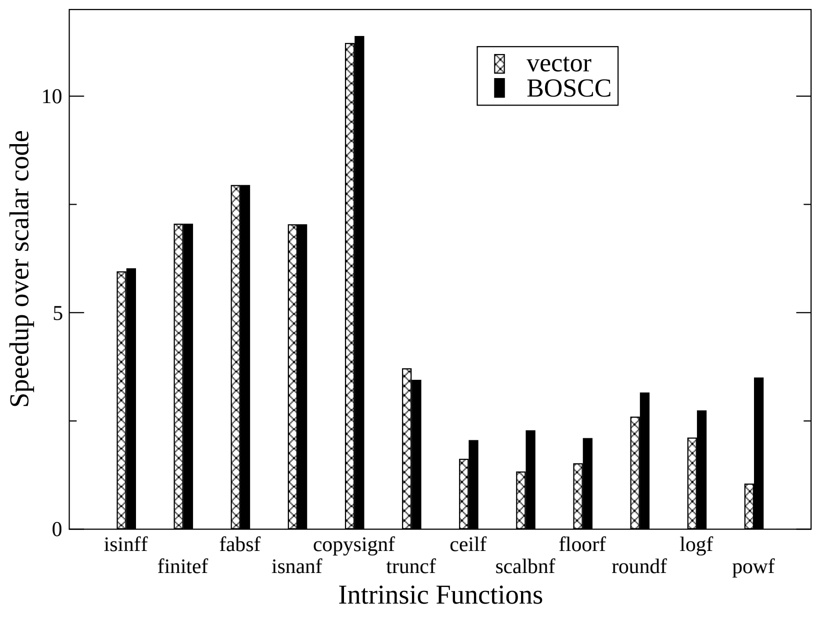


Figure 6: Speedup over scalar baseline with the data size that fits in the L1 cache.

output codes, we used gcc 4.0.1 with -O2 option on Mac OS X 10.4.7. The generated executables were run on a Power Mac G5, with 32 KB L1 data cache, 512 KB L2 integrated cache, and 8 GB of memory.

5.3 Results

Figure 6 shows the speedups of the expanded versions over the scalar baseline for 12 intrinsic functions. Two expanded versions are measured. The first versions, labeled "vector," do not use BOSCCs, while the other versions, labeled "BOSCC," use them to bypass vector instructions. Whenever the expandable intrinsic functions are called from inside the given input function, the same version is used. For example, `powf` calls `scalbnf`. In order to measure the run time of the "vector" version of `powf`, the "vector" version of `scalbnf` is used. For the "BOSCC" version of `powf`, the "BOSCC" version of `scalbnf` is used.

The speedups for the "vector" versions range from 1.04 to 11.21, and for the "BOSCC" versions the speedups range from 2.05 to 11.37. For the first five intrinsic functions, the speedups are identical for both versions because they don't have control flow and no BOSCCs are generated even for the "BOSCC" versions. BOSCCs are generated for the rightmost seven intrinsic functions because the scalar baseline has control flow. When BOSCCs are generated, the "BOSCC" versions are faster except for `truncf`. For `truncf`, the "BOSCC" version is slower than the "vector" version for several reasons. First, two out of the five generated BOSCCs are only overheads because they are never taken. Second, the "vector" version had fewer statements because the redundancy elimination pass could identify more redundancy when the statements are not guarded by predicates. When the unprofitable BOSCCs are not generated, the "BOSCC" version was faster by 6 % than the "vector" version. The speedups of the first five functions in the left are distinctively higher than the other ones. For the five functions, the scalar baselines have data movements through memory between the floating-point register file and the integer register file. For SIMD processing, no data movements are necessary because the same vector register can be used for both integer type and floating-point type, as illustrated in Section 3.2.

Several factors have affected the speedups of this experiments. First, the vector versions exploit SIMD parallelism not only at the vector functions but also at the driver programs by using aligned vector loads and stores. If the data elements are not contiguous in memory, pack and unpack operations should be performed at the vector driver program, thereby reducing the speedups for the vector versions. Second, we have used small data sizes that fits in the L1 data cache to filter out the factors coming from memory latencies. If memory access time dominates the run time, the speedups will not be as significant as in this experiment. Third, if we inlined the generated code, higher speedups would have been obtained because there would be no function call overheads. Finally, in addition to the regular function call overhead, an additional overhead has been observed for vector functions. PowerPC G5 has a `VRSAVE` register that keeps information on which vector registers need to be saved when contexts have to be switched. When the expanded functions are compiled, the GCC generated instructions to manage the VRSAVE register.

6 Related Work

The SIMD programming technique proposed in this paper is most relevant to automatic vectorization techniques. Two different approaches are used for automatic vectorization. The most popular approach is to adapt the vectorization developed for the conventional pipelined vector machines [2, 19, 3]. If a loop can be vectorized, the scalar operands and operators are replaced with vector operands and operators, similar to expansion in our approach. More recently, a new approach has been developed that exploits SIMD parallelism from basic blocks rather than loops [7, 6, 9, 5]; In order to increase the amount of SIMD parallelism, the loops are unrolled. Unlike these two approaches, our approach targets whole functions. Because of this fundamental difference, function calls can also be expanded by our approach. Another difference is that the *expand* transformation in our approach is a semantic transformation, whereas the other techniques generate semantically identical vector code.

Using libraries for vector math intrinsic functions is similar to our approach except that such libraries are written manually [1, 12], and hence, will probably outperform the vector functions generated by our compiler. However, library implementations have the same drawbacks as does manual SIMD programming: they are not portable, there are error prone, and they take a huge amount of time and effort. On the contrary, our approach is also intended to be used by programmers in general for their application programs.

7 Conclusion

Although SIMD functional units are ubiquitous in contemporary microprocessors, compared with the scalar functional units, they are significantly underutilized. SIMD parallelism is orthogonal to the parallelism in other levels and cannot be left out for the highest possible performance. To make it easy to program the SIMD functional units, we propose a new approach, where the given scalar functions are transformed into the corresponding vector functions. Our approach positions itself between the fully automated vectorization by compilers and manual SIMD programming. In experiments on 12 math intrinsic functions, the vector functions transformed by our compiler achieve speedups from 2.05 to 11.37. Extending this approach to applications that are far beyond the current state-of-the-art vectorization techniques is left as a future work.

References

[1] AMD. *AMD Core Math Library (ACML)*, 2006. Version 3.6.0, http://developer.amd.com/assets/acml_userguide.pdf.

[2] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.

[3] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.

[4] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.

[5] IBM. *Exploiting the Dual Floating Point Units in Blue Gene/L*, March 2006. http://www-1.ibm.com/support/ docview.wss?uid=swg27007511.

[6] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[7] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC, Canada, June 2000.

[8] Ruby Lee. Subword parallelism with MAX-2. *ACM/IEEE International Symposium on Microarchitecture*, 16(4):51–59, August 1996.

[9] Rainer Leupers. Code selection for media processors with SIMD instructions. In *ACM/IEEE Conference on Design Automation and Test in Europe*, pages 4–8, 2000.

[10] Motorola. *AltiVec Technology Programming Environments Manual, Rev. 0.1*, November 1998. ftp://www.motorola.com/ SPS/PowerPC/teksupport/teklibrary/manuals/altivec_pem.pdf.

[11] Motorola. *AltiVec Technology Programming Interface Manual*, June 1999. http://e-www.motorola.com/brdata/PDFDB/ docs/ALTIVECPIM.pdf.

[12] Gary L. Mullen-Schultz. *Blue Gene/L: Application Development.* IBM, December 2005. http://www.redbooks.ibm.com/redbooks/pdfs/sg247179.pdf.

[13] Dorit Naishlos. Autovectorization in GCC. In *The 2004 GCC Developers' Summit*, pages 105–118, 2004.

[14] Joseph C. H. Park and Mike Schlansker. On predicated execution, May 1991. Software and Systems Laboratory, HPL-91-58.

[15] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[16] Jaewook Shin. Introducing control flow into vectorized code. Preprint ANL/MCS-P1411-0407, Argonne National Laboratory, April 2007.

[17] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. In *6th Workshop on Media and Streaming Processors*, December 2004.

[18] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *International Symposium on Code Generation and Optimization*, March 2005.

[19] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–400, 2000.

[20] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, August 1996.