An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions *

Gang Ren
University of Illinois
at Urbana-Champaign
Urbana, IL 61801
gangren@cs.uiuc.edu

Peng Wu
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
pengwu@us.ibm.com

David Padua
University of Illinois
at Urbana-Champaign
Urbana, IL 61801
padua@cs.uiuc.edu

Abstract

Multimedia extensions (MME) are architectural extensions to general-purpose processors to boost the performance of multimedia workloads. Today, in-line assembly code, intrinsic functions and library routines are the most common means to program these extensions. A promising alternative is to exploit vectorization technology to automatically generate MME instructions from programs written in standard high-level languages. However, despite the early success of automatic vectorization for traditional vector supercomputers, state-of-the-art vectorizing compilers for multimedia extensions have yet to demonstrate their effectiveness, especially on multimedia workloads.

In this paper, we conducted an empirical study on the vectorization of media processing programs for multimedia extensions. Our study identified several new issues that are not handled by traditional vectorizers. These issues arise partly as the result of the unique features of MME architectures, partly due to the characteristics of media processing applications. We proposed several techniques to address some of these issues. We further assessed the effectiveness of our techniques by manually applying them to a set of multimedia programs. In addition, we found that further optimizations after vectorization are essential to benefit from multimedia extensions. In our experiments, 23 of 34 core procedures from the Berkeley Media Benchmark (BMW) were manually vectorized and 14 procedures achieved speedups of 1.10 to 3.39 on a Pentium 4 processor.

* This work was supported in part by the National Science Foundation under grant CCR 01-21401 ITR and by DARPA under contract NBCH30390004. This work is not necessarily representative of the positions or policies of the Army or Government.

1. Introduction

Multimedia processing has become one of the most important workloads on today's desktop systems. To accelerate these workloads, multimedia extensions (MME) were introduced on practically all general-purpose microprocessors including AltiVec for PowerPC [14] and SSE2 for Pentium [8]. Similar architectures can also be found in game consoles such as PS2 [27], graphics engines such as NVIDIA's GeForce [22], and some DSP(Digital Signal Processing) processors.

Most multimedia extensions can be characterized as SIMD units that support operations on packed, fixed-length vectors, usually no longer than 16 bytes. A typical SIMD instruction processes all elements of a vector simultaneously. Figure 1 illustrates a 128-bit SIMD unit performing a four-element vector add.

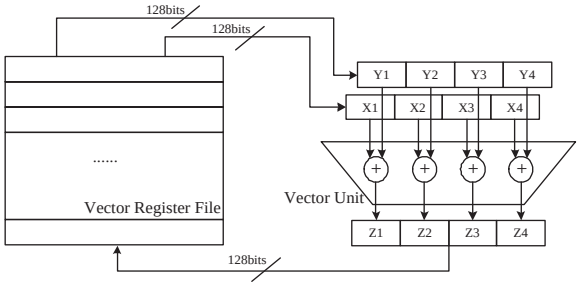

Figure 1. Multimedia extension architecture.

The common way to program multimedia extensions today is through in-line assembly code, intrinsic functions, and library routines. However, explicit low-level vector programming is both time-consuming and could lead to many errors. A promising alternative is to exploit vectorization technology to automatically generate MME instructions from programs written in conventional high-level languages.

There are several recent studies on vectorization for multimedia extensions [9, 15, 17, 28]. A few commercial MME vectorizers have also being developed [4, 6, 26, 29, 31]. Many of them, however, focused on the programs that are known to be vectorizable, such as dense linear algebra computations. However, practically no result has been reported on the vectorization of real multimedia programs.

Our experiments indicate that state-of-the-art vectorizing compilers for multimedia extensions still have a long way to go before becoming a viable alternative to manual vectorization, especially for multimedia applications. This is partly because multimedia applications exhibit distinct characteristics from traditionally vectorizable programs, partly because MME architectures impose more constraints to the compiler than traditional vector architectures do. In this paper, we focus on MME vectorization issues that arise because of application characteristics. Vectorization issues due to new architectural constraints have been addressed in several studies [4, 9, 17], and are not the focus of the paper.

To identify the issues faced by MME vectorization due to new application characteristics, we conducted a code study on a set of core procedures from the Berkeley Multimedia Workload (BMW) suite [25]. During the study, we identified several common code patterns that have significant impact on the effectiveness of MME vectorization. Furthermore, we proposed several techniques to address some of the issues identified, and assessed the effectiveness of these techniques by applying them to the core procedures by hand. Among the 34 procedures from BMW benchmark that we studied, we were able to manually vectorize 23 of them and 14 procedures achieved speedups of 1.10 to 3.39.

Another important conclusion of our study is that post-vectorization optimizations are essential to obtain performance gains from MME vectorization. Several such optimizations are proposed in the paper and their performance benefits are experimentally quantified.

The rest of the paper is organized as follows. Section 2 summarizes the applications used in our study, and presents an evaluation of the Intel vectorizer using those applications. Section 3 characterizes common code patterns in these applications and their impact on MME vectorization. Section 4 illustrates important techniques used in our manual vectorization study. Experimental results are presented in Section 5. Section 6 discusses related work and we conclude in Section 7.

2. Multimedia Applications

2.1. Berkeley Multimedia Workload

Our code study is based on the Berkeley Multimedia Workload (BMW) benchmark [25], which is an extension of MediaBench [19]. Table 1 lists the 12 applications from the BMW benchmark that considered in this paper. Some of the applications may consist of several standalone programs that share a common code base. For example, there are four programs in `Mesa` and the applications annotated with "(E/D)" include two programs, one for encoding and the other for decoding.

Name	Description	#Proc	%Ex.
ADPCM	Audio compression (E/D)	1/1	100/100
GSM	Speech compression (E/D)	2/1	73.8/74.0
LAME	MPEG audio encoder	2	30.7
mpg123	MPEG audio decoder	1	57.9
DVJU	Image compression (E/D)	2/1	80.9/84.0
JPEG	Image compression (E/D)	2/3	51.6/77.8
MPEG2	Video compression (E/D)	2/3	69.2/59.2
POVray	Ray tracer	1	15.5
Mesa	3D Graphics (Gear)	1	81.3
	3D Graphics (Morph3D)	2	23.8
	3D Graphics (Reflect)	1	48.7
	3D Graphics (Pointblast)	3	70.0
Doom	FPS video game	1	25.9
Rsynth	Speech synthesizer	1	70.5
Timidity	MIDI music rendering	3	52.2
Average		2	62.3

Table 1. BMW multimedia applications.

In our study, we focus on procedures that consume more than 10% of total execution time. We call them *core procedures*. As shown in Table 1, a few core procedures consume a large portion, 62% on average, of the total execution time of an application. Thus, speedups on these core procedures will lead to a significant whole-program performance improvement.

2.2. Evaluation of the Intel Compiler for SSE2

We first investigate the effectiveness of state-of-the-art MME vectorizing compilers. The experiment uses the Intel compiler v8.0 (ICC) and is conducted on a Pentium 4 machine with SSE2. Among all the commercial vectorizing compilers for SSE2, the Intel compiler is one of the most widely used and best documented (in both user manuals [2, 7] and research reports [3, 4]). In addition, the Intel compiler can successfully vectorize 73 out of 135 loops from the Callahan-Dongarra-Levine Fortran test suite [20]. This number is comparable with those of the vectorizers reported in [20]. It shows that the Intel compiler is a competent vectorizer according to traditional vectorization standards.

Table 2 summarizes the number of loops in the core procedures that are vectorized by the Intel compiler. Out of 160

loops in the core procedures, 114 are innermost loops, 14 of them are fully vectorized, 3 are reported by the Intel compiler as "vectorizable but seems inefficient". For loops that are not vectorized, Table 2 further classifies them according to the reasons why vectorization fails as reported by the Intel compiler.

Reasons Reported by ICC	# of Loops
Vectorized	14
Vectorizable but seems inefficient	3
Outer Loops	46
Irregular Loop Structure	30
Data Dependence	19
Unsupported Instructions	18
Unsupported Data Types	9
Conditions	18
Others	3
Total Number of Loops	160

Table 2. Loops vectorized by ICC v8.0.

3. Characteristics of Multimedia Programs

In this section, we summarize common code patterns we found in BMW benchmarks and their implications for vectorization.

3.1. Use of Pointers vs. Arrays

Traditional vectorization is very effective for programs that spend most of their time on tight loops involving explicit array accesses. Multimedia programs, however, often use pointers and pointer arithmetic to access data in computationally intensive loops. Of the twelve programs we studied, all of them use pointers in their core procedures, and six of them use pointer arithmetic in addition.

Pervasive use of pointers and pointer arithmetic has a great impact on both memory disambiguation and dependence testing for vectorization. One commonly used technique is to translates pointer accesses and pointer arithmetics onto array accesses. This technique is called array recovery [13]. Basically, it treats pointers as induction variables and expresses pointer accesses in terms of a base address and an closed form expression of the surrounding loop counters. We also observed loops which contain pointers that have no closed-forms. In this case, one can exploit the monotonicity of the pointers to estimate the access region as well as conducting dependence analysis [30].

3.2. Manually Optimized Codes

Many multimedia programs are hand optimized. Some of the manual optimizations can completely change the appearance of original algorithms and oftentimes make it more difficult for the compiler to vectorize. Furthermore, user optimizations make it more difficult to make vectorization profitable, especially when the optimization on scalar codes is not applicable to vectorized codes.

One of most common user optimizations is loop unrolling. One technique to vectorize unrolled loops is to first reroll the loop then apply loop-level vectorization [20]. Another technique is to pack isomorphic instructions within the unrolled loop into SIMD instructions [17].

Another common optimization is to use lookup tables and/or if-statements to shortcut expensive computation for a set of frequently encountered inputs. Figure 2 gives an example of using both lookup table and if-statement to shortcut the computation, `pow(x,0.75)-0.0946`. In this example, array `lutab` stores the precomputed value for `x` $<$ `1000` in the initialization (at statement 3). If an input value falls within the interval, the result is directly retrieved from `lutab` (statement 9) instead of being computed (statement 11). The code is further optimized by using if-statement shortcuts, where statements 6 and 7 return 0 or 1 as the result of `pow(x,0.75)-0.0946` when `x` is less than `1.862955`.

```
1. if (init==0)
2.   for (i=0;i<LUTABSIZE;i++)
3.     lutab[i]=nint(pow((float)i/10.0,
  0.75)-0.0946);
   ...
4. for (i=0;i<l_end;i++) {
5.   temp=istep*fabs(xr[i]);
6.   if (temp<0.499996) ix[i]=0;
7.   else if (temp<1.862955) ix[i]=1;
     ...
8.   else if (temp<1000.0)
9.     ix[i]=lutab[(INT32)(temp*10.0)];
10.  else
11.    ix[i]=(INT32)(sqrt(sqrt(temp)*
  temp)+0.4054);
12.   }
```

Figure 2. User-optimized code from LAME.

Since most multimedia extensions do not support indexed memory access, it is extremely difficult to vectorize loops with lookup table accesses. If we vectorize other computation in the loop and leave lookup tables in scalar forms, the overhead introduced by transferring data between scalar and vector registers could result in a significant slowdown (2-3 times in our experiments).

3.3. Subword Integer Operations

Multimedia programs often use 8-bit or 16-bit integers (referred to as *subword* integers) to represent media data,

such as colors or pixels. As shown in Table 3, 9 out of 12 applications we studied use subword integers as their primary data types. The rest use floating points.

Type	Applications
char	MPEG2, Doom, Mesa
short	ADPCM, GSM, DVJU, JPEG, Timidity, mpg123
single	Rsynth
double	LAME, POVray

Table 3. Major data types in BMW.

According to ANSI C semantics, all subword integers are automatically promoted to register-length integers before conducting any arithmetic operations. This is known as integral promotion [23] and is implemented in most commercial compilers. In terms of vectorization, integral promotion of subword types can waste more than half of the total computation bandwidth as well as incur the additional overhead of integer extension. Furthermore, multimedia extensions often provide better support for subword operations than for 32-bit integer operations. For example, SSE2 supports `max` and `min` for 8- and 16-bit integers, but not for 32-bit integers. Therefore, any efficient MME vectorizer needs to be able to eliminate redundant integral promotion without affecting the program semantics.

3.4. Saturated Operations

Saturated arithmetic is widely used in multimedia programs, especially in audio and image processing applications. Since C does not support saturated arithmetic as native operation, programmers must express saturated operations using other operations.

Figure 3 gives one implementation of saturated add in C. Using if-conversion, the code fragment in Figure 3 can be vectorized into a sequence of compare, mask, subtract and add. However, for multimedia extensions with native support for saturated arithmetic, the best performance can only be achieved by recognizing the sequence and transforming it into a saturated add instruction. Idiom recognition can be extended to identify these saturated operations [3].

```
/* MAX_WORD and MIN_WORD are constants */
/* short a, b; int ltmp; */
#define GSM_ADD(a, b) \
((unsigned)((ltmp=(int)(a)+(int)(b)) \
- MIN_WORD) > (MAX_WORD - MIN_WORD) ?  \
(ltmp > 0 ? MAX_WORD : MIN_WORD) : ltmp )
```

Figure 3. Saturated add in GSM.

Within the BMW benchmark, there are also other implementations of saturated operations. Figure 4 gives such an example. There array `Clip` is a constant array generated during initialization that maps a subscript to its corresponding saturated 8-bit value. It seems more difficult for a compiler to recognize this type of saturated operation.

```
/* short *bp; char *rfp; */
1.  for (i=0; i<8; i++)
2.    for (j=0; j<8; j++) {
3.      *rfp = Clip[*bp++ + *rfp];
4.      rfp++;
5.    }
```

Figure 4. Saturated Add in MPEG2

4. Vectorization Optimizations

In this section, we present several vectorization and post-vectorization optimizations that are essential to achieving performance benefits on multimedia extensions.

In the rest of this section, vector variable names have a prefix of "v", and, unless otherwise specified, variables used in the examples are either short, arrays of short, or vectors of short. Also, we introduce the following notation,

- `v(i)` represents the i-th element of a vector v.
- `VLoad(`$addr$`)` represents a vector load from $addr$.
- `VStore(`$v, addr$`)` stores a vector v to $addr$.
- `LeftShift(`v, i`)` shifts left vector v by i elements.
- `Recur(`s, v_{in}, op`)` computes a vector v_{out} from vector v_{in}, scalar s, and operation op, as following,
  $v_{out}(0) = v_{in}(0)\ op\ s$;
  $v_{out}(i) = v_{out}(i-1)\ op\ v_{in}(i), \forall i > 0$

4.1. Vectorization of Linear Recurrences

Consider the loop given in Figure 5. This loop is a simplified and scalar-expanded version of a core loop from `GSM` encoder. In order to vectorize the innermost i-loop, one must address the dependence cycle from `d[i+1]` to `d[i]` in statement 4. Fortunately, statement 4 can be recognized as a *linear recurrence* and there are known techniques to vectorize linear recurrences of this type, such as *parallel prefix* [16].

In Figure 6, we give the vectorized code, where the numbers in the comments to the right show the corresponding statements in the original loop. Note that the i-loop has a trip count of 8. Thus after the vectorization, the original inner loop is completely replaced by straightline vector statements. Figure 7 illustrates the vectorization of linear recurrence at statement 5 in Figure 6 using parallel prefix.

We would like to point out that, in `GSM`, saturated add instead of modulo add is used in the linear recurrence. Although saturated add on signed integers is not associative in

```
1.  for (; k_n--; s++) {
2.    d[0] = t[0] = *s;
3.    for (i = 0; i < 8; i++) {
4.       d[i+1] = d[i] + (rp[i] * u[i]);
5.       t[i+1] = u[i] + (rp[i] * d[i]);
6.       u[i]   = t[i];
7.    }
8.    *s = d[8];
9.  }
```

Figure 5. A simplified loop from GSM.

```
1. vrp = VLoad(&rp[0]);
2. for (; k_n--; s++) {
3.   d[0] = t[0] = *s;                /* 2 */
4.   vu = VLoad(&u[0]);               /* 4 */
5.   VStore(Recur(d[0], vrp*vu, +), &d[1]);
6.   vt = vu + (vrp * VLoad(&d[0])); /* 5 */
7.   VStore(vt, &t[1]);
8.   VStore(VLoad(&t[0]), &u[0]);    /* 6 */
9.   *s = d[8];                       /* 8 */
10.}
```

Figure 6. Vectorized loop from Figure 5.

```
1.  vd = vrp * vu;
2.  vd = LeftShift(vt, 1) + vd;
3.  vd = LeftShift(vt, 2) + vd;
4.  vd = LeftShift(vt, 4) + vd;
5.  vd = vd + d[0];
```

Figure 7. Vectorization of `Recur(d[0], vrp*vu,+)` using parallel prefix.

general, we can loosen the restriction based on programmer-provided directives and/or compiler-collected information. For two core procedures in `GSM`, parallel prefix adds 50% to the performance of the vectorized code and produces correct results for the given input data set.

4.2. Vector Copy Propagation

We can further optimize the vectorized loop shown in Figure 6 by vector copy propagation to eliminate redundant memory accesses. For example, there is a copy propagation opportunity between statements 5 and 6 as one stores to memory `d[1]` to `d[8]`, and the other loads from memory `d[0]` to `d[7]`. Similar pattern also occurs between statements 7 and 8, and between 8 and 4.

In the first two cases, memory accessed by the pair of store and load do not completely overlap, i.e., with one element difference. To propagate from stores to partially overlapped loads, additional data reorganization operations are needed. For example, one can construct the value of $\hat{d}$[0] to `d[7]` by combining scalar value `d[0]` with the vector that contains value of `d[1]` to `d[8]`. Figure 8 shows the vectorized loop after applying vector copy propagation. Since Pentium 4 is a "little endian" processor, `LeftShift` is used in statement 8 and 12.

```
1. vrp = VLoad(&rp[0]);
2. vu = VLoad(&u[0]);
3. for (; k_n--; s++) {
4.   d[0] = t[0] = *s;
5.   vu = vt2;
6.   vd = Recur(d[0], vrp*vu, +);
7.   VStore(vd, &d[1]);
8.   vd2 = LeftShift(vd, 1);
9.   vd2(0) = d[0];
10.  vt = vu + (vrp * vd2);
11.  VStore(vt, &t[1]);
12.  vt2 = LeftShift(vt, 1);
13.  vt2(0) = t[0];
14.  VStore(vt2, &u[0]);
15.  *s = d[8];
16.}
```

Figure 8. After vector copy propagation.

After all vector loads in Figure 6 are eliminated, the remaining vector stores may be moved out of the loop or eliminated as dead code, depending on whether the memory accessed by the store is used by later computation.

In `GSM`, vector copy propagation combined with dead store elimination improves the vectorization speedups from 0.88-1.27 to 2.08-2.97.

4.3. Vectorizing Outer Loops

Figure 9 gives another core loop extracted from GSM that implements a FIR(Finite Impulse Response) filter. In the original GSM code, the i-loop is completely unrolled by programmers. In this example, we rerolled the unrolled statements back to an inner loop. Both unrolled and rerolled implementations of FIR computation are very common in DSP and media processing domain.

```
1.  for(lambda=40; lambda<=120; lambda++)
2.  {
3.    sum = 0;
4.    for(int i=0; i<40; i++)
5.      sum += wt[i] * dp[i - lambda];
6.    L_result[lambda] = sum;
7.  }
```

Figure 9. Another Simplified Loop from GSM

Let us first consider the vectorization of the inner i-loop. The i-loop involves reduction and conversion from short to int (as sum is 32-bit, wt and dp are arrays of 16-bit integers) Vectorization of reduction is straightforward. Conversion can be vectorized into packing/unpacking operations.

After vectorizing the inner loop, we would still need to store *sum* into *L_result* in each iteration of the outer loop. It would be better to combine stores to contiguous locations in the outer loop into a vector store. This can be accomplished by vectorizing the outer loop. This transformation introduces data movement operations. For example, to store a 4-element vector to `L_result`, at least three data movement instructions are needed to pack the four (scalar) results. Therefore, the benefit of this optimization depends on the memory access latency, data movement overhead, and register pressure.

In our experiment, this optimization can up to achieve an additional 66% performance improvement on core procedures.

4.4. Subword Arithmetic Optimization

Consider the following operation from GSM,

$$vc = ((va * vb) + 16384) >> 15$$

where `va`, `vb` and `vc` are vectors of shorts.

In SSE2, a 16-bit multiply is implemented by two operations, `MultiplyLow` and `MultiplyHigh`, to produce the low and high halves of the 32-bit result as follows,

$$vx = MultiplyLow(va, vb)$$
$$vy = MultiplyHigh(va, vb)$$
$$va * vb = vy << 16 + vx$$

where vx and vy are both vectors of shorts [8].

By exploiting the arithmetic properties of ">>", "<<", and "+", we can simplify the original computation as follows,

$$\begin{aligned}&((va * vb) + 16384) >> 15\\ &\equiv ((vy << 16 + vx) + 16384) >> 15\\ &\equiv (vy << 2 + vx >> 14 + 16384 >> 14) >> 1\\ &\equiv (vy << 1) + (vx >> 14 + 1) >> 1\end{aligned}$$

The key to this simplification is the distribution of right shift over add. In general, a right shift can be distributed into an add only if at least one operand of the add has more trailing zeros than the number of bits to be shifted. Note that the constant 16384 has 14 trailing zeros. That is why we decompose the original 15-bit shift into a 14-bit shift and a 1-bit shift and distribute the 14-bit shift into the add.

In our experiment, subword arithmetic optimization can achieve additional 16-18% performance improvements on the core procedures.

5. Evaluation of Vectorization

5.1. Manual Transformation

To evaluate the effectiveness of our techniques, we selected 34 core procedures from the BMW benchmarks as candidates for applying manual vectorization. The characteristics of these core procedures are summarized in Tables 6 and 8. The vectorized codes invoke SSE2 operations via intrinsic functions and are subsequently compiled by the Intel compiler (v8.0).

During the manual transformation, we made the following assumptions. First, global pointer information is available for the procedures to be transformed. This information could be obtained either through compiler analysis or provided by programmers via pragmas. Second, a powerful idiom recognizer is employed to identify code patterns such as `min`, `max`, `average`, and saturated arithmetic. In essence, the manual transformation is designed to evaluate the effectiveness of vectorization transformations without being limited by the analysis-related issues discussed in Section 3.

5.2. Experiment Setup

The evaluation was conducted on a Linux-based Pentium 4 machine with SSE2 extension. Performance is measured using PAPI [21], a performance counter library for Pentium 4 processor. Table 4 summarizes the configuration of the machine and the tools used in the experiment.

Processor	Pentium 4 2.00GHz
Multimedia Extension	SSE2
Main Memory	1024 MB
Operating System	Linux v2.4
Compiler	icc v8.0
Baseline Optimization	-O3
Vectorization Option	-O3 -march=pentium4 -fno-alias
Time Measuring	PAPI library

Table 4. Experiment Platform

In Table 5, we summarize the latency and throughput of common P4 scalar instructions and SSE2 vector instructions. Note that the integer ALUs in P4 processor run at 2x clock rate while SSE2 unit can only process 64-bit packed integer operations (i.e., half of the vector length) per cycle. In other words, most integer SSE2 instructions have four times the latency and one-fourth of the throughput of the corresponding instructions of the P4 scalar unit [7].

5.3. Core Procedures Vectorized

Of the 34 core procedures, we are able to vectorize the 23 procedures listed in Table 6. Performance (measured in

Application	Core Procedures	%Ex	Loops	Ptr	SatOp	MT	Lookup	ManOpt
GSM.Encoder	`Calculation_of_the_LTP...`	36.51	1x2	P		*		LoopUnroll
GSM.Encoder	`Short_Term_Analysis_Filter`	37.30	1x2	P	*			
GSM.Decoder	`Short_Term_Synthesis_Filter`	74.00	1x2	P	*			
LAME	`quantize`	17.38	3x1	P	*	*	Math	Shortcuts
LAME	`calc_noise2`	13.30	2x2	P	*	*	Math	Shortcuts
mpg123	`synth_1to1`	57.92	2x1	P	*	*		LoopUnroll
MPEG2.Encoder	`dist1`	43.80	4x2	P		*		LoopUnroll
MPEG2.Encoder	`fdct`	25.38	2x3	P		*		
MPEG2.Decoder	`form_component_prediction`	12.14	4x2	P		*		
MPEG2.Decoder	`idct`	35.49	2x0	P		*	*	LoopUnroll
MPEG2.Decoder	`Dither_Frame`	11.57	4x2	PA		*	SatOp	
JPEG.Decoder	`ycc_rgb_convert`	16.67	1x2	PA	*	*		
JPEG.Decoder	`jpeg_idct_islow`	38.89	2x1	PA		*	SatOp	LoopUnroll
DJVU.Encoder	`IWPixmap::init (loop1)`	15.24	3x2	P	*	*		
DJVU.Encoder	`forward_filter`	65.65	3x1	P		*		
DJVU.Decoder	`backward_filter`	83.99	3x1	P		*		
Mesa.Morph3D	`gl_shade_rgba_fast`	13.77	1x2	P		*		
Mesa.Reflect	`persp_textured_triangle`	48.70	2x3	P		*		
Mesa.Pointblast	`gl_depth_test_span_generic`	27.22	12x1	PA				
POVray	`Dnoise`	15.49	1x0	PA		*	*	
Timidity	`rs_vib_loop`	29.31	1x2	PA		*		
Timidity	`mix_mystery_signal`	10.18	2x2	PA		*		
Timidity	`mix_single_signal`	12.75	2x2	PA	*	*		

(Abbr: %Ex-Percentage of Execution time; Loops-Important loops in the procedure, where *m*x*n* means there are *m n*-nest important loops in the procedure; Ptr-Pointer reference; P-Pointers as parameters; PA-Pointer Arithmetic; SatOp-Saturated Operations; MT-Mixed Types; Lookup-Table Lookups; ManOpt-Manual Optimizations)

Table 6. Characteristics of core procedures successfully vectorized.

Operations	Latency	Throughput
Scalar Integer Add/Sub.	0.5	0.5
Scalar Logical Ops	0.5	0.5
Scalar Integer Multiply	14-18	5
Scalar FP Add/Sub.	5	1
Scalar FP Multiply	7	2
Vector Integer Add/Sub.	2	2
Vector Logical Ops	2	2
Vector Integer Multiply	8	2
Vector FP Add/Sub.	4	2
Vector FP Multiply	6	2

Table 5. Latency and Throughput of P4 Inst.

speedups) of the vectorized procedures is given in Table 7. The table is divided into 3 sections. The first section contains procedures that can be vectorized without algorithm change and achieves 10% to 239% of improvement after vectorization. The middle section contains procedures that require algorithm changes to be vectorized. The last section contains procedures that show little performance speedups or even slowdowns as the result of vectorization.

In Table 7, column "Best" presents the best speedups we have achieved by applying all transformations to the procedure. The next columns, "-VCP", "-VOL", and "-SAC", show the speedups achieved if individual vectorization optimizations are excluded. For some procedures, we can achieve much better performance by replacing floating point algorithm with a fixed point algorithm that is more suitable for vectorization. Column "Alg." shows the speedups resulting from such algorithm substitution. For example, in `fdct`, we achieved an additional speedup of 3.23(=9.41/2.91). Column "ParP" shows the speedup resulting from vectorization of linear recurrences using parallel prefix. This transformation assumes associativity of saturated arithmetic.

We also measured the performance of the Intel compiler (v8.0) vectorization. The last column, "ICC", shows the performance achieved by vectorizing the procedure using the option listed in Table 4. This column only shows the speedups of those procedures that have at least one loop vectorized by the Intel compiler. The Intel compiler successfully vectorizes all innermost loops in `dist1` from

Application	Core Procedures	%Ex	Best	-VCP	-VOL	-SAO	Alg.	ParP	ICC
GSM.Encoder	`Calculation_of_the_LTP...`	36.51	2.46		1.85				0.69
GSM.Encoder	`Short_Term_Analysis_Filter`	37.30	2.08	0.88		1.79		2.59	
GSM.Decoder	`Short_Term_Synthesis_Filter`	74.00	2.97	1.27		2.51		3.46	
LAME	`calc_noise2`	13.30	1.52						0.95
mpg123	`synth_1to1`	57.92	1.75	1.28					
JPEG.Decoder	`jpeg_idct_islow`	38.89	1.10				1.44		
MPEG2.Encoder	`dist1`	43.83	3.39	3.37		3.20			3.05
MPEG2.Encoder	`fdct`	25.38	2.91				9.41		0.87
MPEG2.Decoder	`form_component_prediction`	12.14	2.12						0.90
MPEG2.Decoder	`idct`	35.49	1.18				3.68		
DJVU.Encoder	`IWPixmap::init (loop1)`	15.24	1.95		1.66				1.04
Mesa.Reflect	`persp_textured_triangle`	48.70	1.40						
Mesa.Pointblast	`gl_depth_test_span_generic`	27.22	1.45						0.99
Timidity	`mix_mystery_signal`	10.18	1.15						
LAME	`quantize`	17.38	2.00						1.17
JPEG.Decoder	`ycc_rgb_convert`	16.67	1.20		1.07				
MPEG2.Decoder	`Dither_Frame`	11.57	0.71	0.71					
DJVU.Encoder	`forward_filter`	65.65	1.00						
DJVU.Decoder	`backward_filter`	83.99	1.00						
Mesa.Morph3D	`gl_shade_rgba_fast`	13.77	0.89						
POVray	`Dnoise`	15.49	0.99						
Timidity	`rs_vib_loop`	29.31	1.01						
Timidity	`mix_single_signal`	12.75	0.45						

(Abbr: Best-The best speedup; -VCP-exclude Vector Copy Propagation; -VOL-exclude Vectorizing Outer Loops; -SAC-exclude Subword Arithmetic Optimization; Alg.-Algorithm Changes; ParP-Parallel Prefix (Assuming the associativity of saturated arithmetic);)

Table 7. Performance summary of manual vectorization on core procedures.

`MPEG.Encoder` and reaches near-optimal speedup on this procedure. However, it has little or negative impact on other core procedures. [1]

Some user optimizations, such as if-shortcut and lookup tables, preclude vectorization on otherwise vectorizable programs. In general, it is very difficult for the compiler to reverse engineer such optimizations. Therefore we segregated procedures that require manual "de-optimization" to be vectorized from the others to the middle section of Table 7. For example, `quantize` uses both lookup table and if-statement shortcuts, and `ycc_rgb_convert` uses a lookup table to implement saturated operations.

About one third of the vectorized procedures show little speedups over the original scalar codes as shown at the bottom section of Table 7. This is partly due to the inefficient implementation of SSE2. According to Table 5, the theoretical maximum speedups achievable on vectorizing int, short and char operations, except for multiplication, are 1, 2 and 4, respectively. Furthermore, many core procedures have been optimized by hand, such as using lookup tables, and need to be "de-optimized" to be vectorized. However, sometimes the performance gains of vectorizing unoptimized algorithm may not make up the performance loss of "de-optimization". Finally, the overhead of data reorganization, necessary to vectorize reduction and non unit-stride memory accesses (for example, `mix_single_signal`), can also offset the performance benefit of vectorization.

5.4. Non-Vectorizable Core Procedures

There are 11 core procedures (as listed in Table 8) that cannot be vectorized. Among them, 6 procedures have dependence cycles that cannot be expressed as either reduction or linear recurrences. For example, Figure 10 gives the simplified dependence graph of the ADPCM encoder. In the figure, the circles presents the different assignment to variables and the links shows the dependences between them. And the dotted lines represent complex paths involving operations such as table lookups and multiple if-statements.

In addition, three procedures from `Mesa` frequently in-

1 Some core procedures, such as `synth_1to1` from `mpg123`, benefit from SSE2 extension without vectorization.

Application	Core Procedures	%Ex	Loops	Ptr	SatOp	MT	Lookup	Description
ADPCM.Encoder	`adpcm_coder`	100.0	1x1	P	*	*	*	Dep. cycles
ADPCM.Decoder	`adpcm_decoder`	100.0	1x1	P	*	*	*	Dep. cycles
JPEG.Encoder	`encode_mcu_AC_refine`	37.38	1x2	P		*		Dep. cycles
JPEG.Encoder	`encode_mcu_AC_first`	14.20	1x2	P		*		Dep. cycles
JPEG.Decoder	`decode_mcu_AC_refine`	22.22	1x2	P		*	SatOp	Dep. cycles
Rsynth	`parwave`	70.49	1x2	P				Dep. cycles
Mesa.Gears	`flat_TRUECOLOR_...`	81.34	1x3	PA				I/O function
Mesa.Morph3D	`smooth_TRUECOLOR_...`	10.05	1x3	PA				I/O function
Mesa.Pointblast	`write_span_mono_ximage`	29.44	1x1	PA				I/O function
Mesa.Pointblast	`dist_atten_antialiased...`	18.33	2x3	PA		*		Non-close form
DOOM	`R_DrawColumn`	25.88	1x1	P			*	Lookup tables

Table 8. Characteristics of non-vectorizable core procedures.

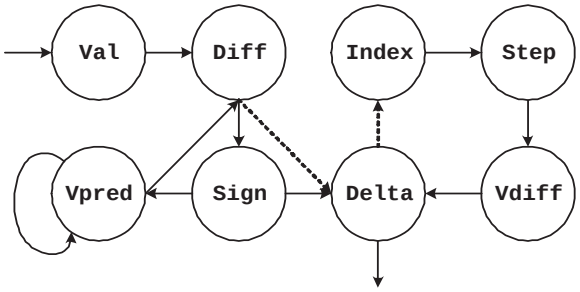

Figure 10. Dep. graph of ADPCM encoder.

voke the I/O function in their hot loops. Due to the overhead of transferring between scalar and vector registers, it is not beneficial to vectorize the other parts of the loop while leaving the function call in scalar form.

6. Related Work

Automatically compiling C programs to multimedia extension instructions has been tried out in both academia and industry. Most of them are based on traditional vectorization techniques, such as the Allen-Kennedy algorithm [1]. In [5], Cheong and Lam developed an optimizer for VIS, the SUN multimedia extension. The focus of this work was to address alignment issues during code generation. Krall and Lelait [15] also applied traditional vectorization to generate VIS code. Sreraman and Govindarajan [28] developed a vectorizer for the Intel MMX. However, only experiments with small kernels were reported in [5, 15, 28].

An alternative way of compiling for multimedia extensions is to pack isomorphic instructions from the same basic block to vector instructions [17, 24]. Speedups were reported on several kernels and few programs from SPECfp in [17]. In [28], a similar idea was introduced as "vectorization by loop unrolling".

To date, a few commercial compilers that support automatic vectorization for multimedia extensions are available. The Crescent Bay Software extends VAST to generate codes for AltiVec [26]. The Portland Group offers the PGI Workstation Fortran/C/C++ compilers that support automatic usage of SSE/SSE2 [29]. The Codeplay announces the VectorC compiler for all x86 extensions [6]. Also, Intel extended its own product compiler to vectorize for MMX/SSE/SSE2 [4] and IBM included automatic vectorization in its XL compilers [31].

In addition, several key issues in compilation for multimedia extensions have been addressed separately by researchers. Memory alignment analysis and enforcement are studied in [4, 9, 18]. Automatic recognition of saturated operations is addressed in [3]. In [10], a domain-specific C-like language, SWARC (SIMD-within-a-register), was developed to provide a portable way of programming for MMEs. Targeting the specific domain of signal processing, Franchetti et al described a vectorizing technique for multimedia extensions at the mathematical formula level in [11, 12].

7. Conclusion

Our study shows that despite the success of automatic vectorization for traditional vector processors, the vectorization for multimedia extensions still has a long way to go. The difficulties come from not only the architectural constraints of multimedia extensions but also the characteristics of multimedia applications.

By conducting an in-depth code study on multimedia applications from Berkeley Multimedia Workload, we identified the main obstacles to MME vectorization. We also proposed compiler techniques to address some of these difficulties and applied them by hand to real multimedia applications. The experimental results show promising performance improvements on the core procedures from the BMW benchmarks. In addition, we found that further opti-

mizations after vectorization are essential to obtain performance improvement from the MME vectorization.

As the first step toward unleashing the power of multimedia extensions through vectorization, our study focused on identifying the new requirements and challenges faced by the MME vectorization, proposing possible compiler solutions for them, and testing the solutions using manual transformations. The compiler techniques discussed in the paper are not general nor complete. Therefore, it necessary to generalize these techniques, make them complete and implement them on an existing compiler. It is also important to extend this study to other application domains, such as numerical applications. It would be interesting to see how effectively numerical programs can be vectorized and benefit from multimedia extensions.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[2] Aart J. C. Bik. *The Software Vectorization Handbook : Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[3] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic detection of saturation and clipping idioms. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computers*, 2002.

[4] 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, 2002.

[5] Gerald Cheong and Monica Lam. An optimizer for multimedia instruction sets. In *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[6] CodePlay. *VectorC PC Overview*. http://www.codeplay.com/vectorc/index_pc.html.

[7] Intel Corporation. *IA32 Intel Architecture Optimization*.

[8] Intel Corporation. *IA32 Intel Architecture Software Developer's Manual (Volume 1: Basic Architecture)*.

[9] Alexandre Eichenberger, Peng Wu, and Kevin O'Brien. Vectoization for short simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[10] Randall J. Fisher and Henry G. Dietz. Compiling for simd within a register. In *Processings of 11th International Workshop on Languages and Compilers for Parallel Processing*, pages 290–304, 1998.

[11] Franz Franchetti and Markus Puschel. A simd vectorizing compiler for digital signal processing algorithms. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS) 2002*, 2002.

[12] Franz Franchetti, Yevgen Voronenko, and Markus Puschel. Loop merging for signal transforms. In *To appear in Proc. PLDI 2005*, 2005.

[13] Bjorn Franke and Michael O'boyle. Array recovery and high-level transformations for dsp applications. *Trans. on Embedded Computing Sys.*, 2(2):132–162, 2003.

[14] Sam Fuller. Motorola's altivec technology. 1998.

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

[16] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[17] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pages 145–156. ACM Press, 2000.

[18] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29. IEEE Computer Society, 2002.

[19] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[20] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17(10-11):1223–1244, 1991.

[21] Philip J. Mucci, Kevin London, and Dan Terpstra. *PAPI Programmer's Reference v2.3*.

[22] nVIDIA. *nVIDIA GeForce FX*. http://www.nvidia.com/.

[23] International Standard Organization. *Programming Languages - C, ISO/IEC 9899*, 1999.

[24] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *CGO 05*, 2005.

[25] Nathan T. Slingerland and Alan Jay Smith. Design and characterization of the berkeley multimedia workload. *Multimedia Syst.*, 8(4):315–327, 2002.

[26] Crescent Bay Software. *VAST-C/AltiVec: Automatic C Vectorizer for Motorola AltiVec*.

[27] Manu Sporny, Gray Carper, and Jonathan Turner. *The Playstation 2 Linux Kit Handbook*, 2002.

[28] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–300, 2000.

[29] The Portland Group Compiler Technology. *PGI Users Guide : Parallel Fortran, C and C++ for Scientists and Engineers*, 2004.

[30] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *Proceedings of the 15th International Conference on Supercomputing*, pages 78–91. ACM Press, 2001.

[31] Peng Wu, Alexandre Eichenberger, and Amy Wang. Efficient simd code generation for runtime alignment and length conversion. In *CGO 05*, 2005.