

Multimedia Macros for Portable Optimized Programs

Juan Carlos Rojas and Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University, Boston

Abstract

Multimedia architectures can speed-up applications significantly when programmed manually. Optimized programs have been non-portable up to now, because of differences in instruction sets, register lengths, alignment requirements and programming styles. We solve all these problems by using a library of C pre-processor macros called MMM. We implemented three examples from video compression in MMM, and automatically translated them into optimized code for four distinct multimedia processors. Their performance is comparable, and in several cases better, than equivalent examples optimized by the processor vendors.

Problem

Multimedia computing has been one of the greatest challenges in computer engineering for the last decade. Computer designers have been challenged to come up with solutions capable of processing the enormous amounts of data required by multimedia applications. The solutions came in the form of multimedia processors, and multimedia extensions to general-purpose processors. Some well-known examples are AltiVec, MMX and its successors – SSE and SSE2, and TriMedia processors.

All multimedia architectures follow the same basic approach: they partition the registers into sections that represent multiple data elements, and operate on all the sections in parallel. In addition, complex instructions have been added to speed-up specific tasks found in multimedia applications. For example, some architectures include a Sum of Absolute Differences instruction, or a Multiply-High instruction (multiply and pack the most-significant part of the product).

Our experiments and other published results show that multimedia architectures can speed-up applications by factors of up to 15, but manual optimization is required in order to take full advantage of the complex instructions available. Manual optimization is very time consuming, and up to now has resulted in non-portable programs. This is in part because different multimedia architectures have different register lengths, different programming styles, different alignment requirements, and they support different partitioned instructions.

Solution: MMM

We solved the problem by creating MMM: a library of target-independent C pre-processor macros that implements a common set of parallel operations available or efficiently emulated on a given set of target architectures. MMM provides a unique interface to architectures with different register lengths and instruction sets. Long data vectors are simulated by using several small vectors, and operations of long vectors are emulated as a sequence of operations on short vectors. Similarly, vector operations that are missing on a given target are emulated using a sequence of simple vector operations, when it is efficient to do so. The same concept can be used to resolve different alignment requirements. Some architectures require that vector loads and stores are done at aligned addresses. If an unaligned load is required, one must load two aligned vectors and compose the desired result from them. All this can be encapsulated inside MMM load macros, and thus provide the programmer with a general unaligned load virtual instruction.

The table below shows simplified MMM macro definitions in different targets of the Sum of Absolute Differences of two 128-bit vectors partitioned into 8-bit sections. Two partial sums are returned in a vector.

SSE2 (128-bit registers)	AltiVec (128-bit registers)
<pre>#define SAD_U8x16(a,b,c) \ a = _mm_sad_epu8(b,c);</pre>	<pre>#define SAD_U8x16(a,b,c) \ a = vec_sum2s(vec_sum4s(vec_sub(vec_max(b,c), vec_min(b,c))));</pre>
MMX+SSE (64-bit registers)	TriMedia (32-bit registers)
<pre>#define SAD_U8x16(a,b,c) \ a##_0 = _m_psadbw(b##_0, c##_0); \ a##_1 = _m_psadbw(b##_1, c##_1);</pre>	<pre>#define SAD_U8x16(a,b,c) \ a##_0 = UME8UU(b##_0, c##_0); \ a##_0 += UME8UU(b##_1, c##_1); \ a##_1 = UME8UU(b##_2, c##_2); \ a##_1 += UME8UU(b##_3, c##_3);</pre>

Through emulation, MMM implements a large common virtual instruction set for several target architectures. By using MMM, it is possible to write multimedia applications that are portable among different multimedia processors, and take advantage of the complex partitioned operations available on them.

Other Approaches

Parallelizing compilers can generate some multimedia instructions from scalar code, but not the most complex ones. The problem is that it is not easy to express these complex parallel instructions in C. One can also write parallel programs explicitly using a data-parallel language, but this still does not solve the problem of expressing complex parallel instructions. Not even languages specifically designed for multimedia [1, 2] can express complex operations like Sum of Absolute Differences, or Multiply-High.

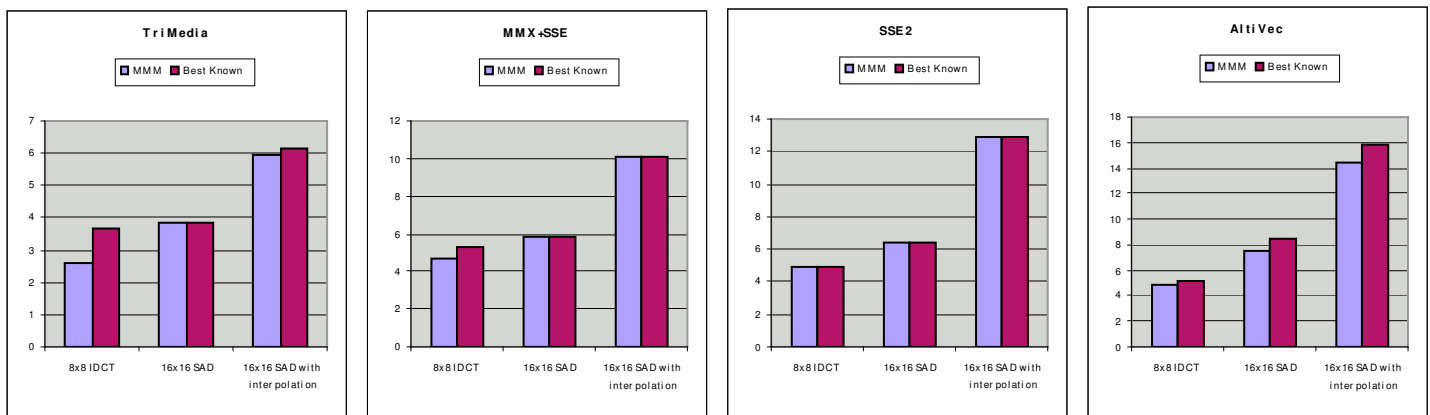
Multimedia code can also be generated from abstract descriptions, like in SPIRAL [4]. This approach is complementary to MMM: the code generator can experiment with different algorithm designs, and prototype them using MMM. Another possibility is to write multimedia applications based on optimized libraries that conform to a standardized API, like VSIPL [3]. This is a good approach for certain classes of applications, but not as flexible as MMM.

Experiments

We implemented an MMM library for four distinct multimedia architectures: AltiVec, MMX+SSE, SSE2, and TriMedia TM1300. These architectures are very diverse. Their register lengths vary from 32 to 128 bits, they have distinct instruction sets, alignment requirements and programming styles.

Then we implemented three example programs on MMM used in video compression: 8x8 Inverse Discrete Cosine Transform (IDCT), 16x16 Sum of Absolute Differences (SAD), and 16x16 SAD with horizontal and vertical interpolation. Through MMM libraries, the same programs were automatically converted into optimized code for all four target platforms.

We measured the execution time and instruction count of the MMM programs on all four targets, and compared them with equivalent programs hand-optimized by each processor vendor. In some cases our programs out-performed the vendor examples, so we attempted to further optimize our programs for each target using non-portable instructions, to serve as references. We compared all the optimized programs with reference scalar implementations, and computed the speedup and the reduction in instruction counts. The following charts show the speedup of the portable MMM versions compared to the best known optimized versions for each target:



The portable MMM programs obtained speedups that are comparable to the best known hand-optimized versions for each target. In some cases, they provide the best known performance. The portable programs written in MMM are indeed optimized for several architectures at the same time.

Conclusions

It is possible to write portable-optimized multimedia programs using MMM. These programs are portable among a diverse group of architectures that have different register lengths, instruction sets, alignment requirement and programming styles. The performance of portable MMM programs is comparable to the best known implementations for each target.

References

- [1] Paul Cockshott. *Vector Pascal, an Array Language*. Jan. 2002, <http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/vp-ver2.pdf> (current May 2003).
- [2] Randall J. Fisher, and Henry G. Dietz. "Compiling for SIMD Within a Register," *Lecture Notes in Computer Science*, vol. 1656, Springer, Berlin, 1998, pp. 292-304.
- [3] David Schwartz, et al. *VSIPL 1.01 API*, http://www.vsipl.org/CD/vsiplv1p01_final1.pdf (current May 2003).
- [4] Franz Franchetti, and Markus Püschel. "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.