EXPLOITING MOTION ESTIMATION RESILIENCE TO APPROXIMATED METRICS ON SIMD-CAPABLE GENERAL PROCESSORS: FROM ATOM TO NEHALEM

Steven Pigeon and Stéphane Coulombe

Department of Software and IT Engineering
École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Qc, H3C 1K3

ABSTRACT

In the past, efforts to speed up motion estimation for video encoding were directed at finding better predictive search algorithms. Now, they are directed toward the shrewd exploitation of the machine's advanced architectural features such as multimedia extensions, especially for the computation of the error metric which is known to be expensive. In this paper, we extend previous work by further exploring efficient implementation of approximate fast metrics for motion estimation. We show that the proposed metrics can be implemented using SIMD instructions to yield impressive speed-ups, up to 12:1 relative to non-vectorized but otherwise optimized C code, while sacrificing less than 0.1 dB on image quality.

Index Terms— video coding, SIMD, motion estimation, motion compensation, error metric, approximate metric, compiler autovectorization, SSE, SSE2

1. INTRODUCTION

Modern video codecs, such as MPEG-4 and H.264, rely on motion-compensated predictive coding as their principal means of achieving high compression ratios through spatial and temporal redundancy reduction. To perform motion compensation, the motion within a scene must first be estimated using a motion estimation algorithm. Despite increasingly sophisticated algorithms, motion estimation remains quite computationally expensive, accounting for a large portion of the run-time in encoders—sometimes up to 60% [1, 2].

Motion estimation is an obvious target for speed optimization. Over the years, numerous methods have been devised to accelerate the process. The focus has shifted progressively from efficient search algorithms to predictive algorithms where a number of probable locations for the best matching block are generated. Predictive motion estimation combined with efficient search methods yields the best results, in both speed and quality. Alas, despite the algorithmic speed-ups offered by the most efficient algorithms, motion estimation remains a very expensive step in video coding.

This work was funded by Vantrix Corporation and by the Natural Sciences and Engineering Research Council of Canada under the Collaborative Research and Development Program (NSERC-CRD 326637-05). E-mails: {steven.pigeon,stephane.coulombe}@etsmtl.ca)

While considerable effort has been made to develop fast and accurate motion estimation algorithms, the metrics used to estimate the goodness of match in such algorithms have been studied comparatively little, even less so in combination with efficient motion estimation algorithms. The metrics available in most codecs are limited to the mean squared error (MSE) and the sum of absolute differences (SAD), with the latter usually favored, as it is deemed to be less expensive to compute than the MSE [3], a hypothesis verified only if the target computer sports efficient absolute value instructions. Both are usually computed with high precision—at greater cost—even though it is known that motion estimation algorithms are quite resilient to various types of errors in the estimation of the metric [4, 5].

Exploiting this resilience naturally leads to approximated, truncated, or even randomized algorithms for the evaluation of the metric, the goal being to balance the precision of the result with the cost of computation. Such algorithms have already been proposed but without any real regard to the implementation-specific impacts on the performance of the underlying machine [6–9]. The proposed solutions often result in highly branching code, or code that is difficult to parallelize, mitigates or even nullifies speed-ups, as they interfere with, amongst other things, the processor's branch prediction unit, thus yielding interesting but ultimately suboptimal results. Truly efficient implementations of approximated metrics must consider implementation-specific techniques, relying on the astute exploitation of the underlying machine's architecture and instruction set (known collectively as the ISA). In modern processors, this means that *full* advantage must be taken of any machine-specific ISA extension, and, in particular, multimedia and single instruction, multiple data (SIMD) extensions.

In this paper, we extend previously presented work [4] by exploring the pragmatics of the implementation of fast approximated metrics and present new results. We show that taking full advantage of the ISA leads to impressive performance gains on a variety of processors, while loss associated with the use of suitable approximated metrics remains negligible.

2. RESILIENCE AND APPROXIMATE METRICS

In [4], we proposed generalizing the SAD computed between two 16×16 image patches I and J as:

$$\mathrm{SAD}_M(I, J) = \sum_{x=1}^{16} \sum_{y=1}^{16} M_{x,y} \left| I_{x,y} - J_{x,y} \right| \qquad (1)$$

where M is a 16×16 binary matrix conditionally enabling comparison between pixels.

The matrix M allows us to define arbitrary masks for the SAD. Masks can be designed based on a number of criteria, maximizing sampling efficiency for a given number of pixels tested, for example. Another criterion of great interest is maximizing evaluation speed by exploiting the machine's ISA under the constraint of minimizing loss of quality. In Fig. 1 we show the masks considered in this paper, as well as in [4]. Of particular interest are the masks shown in Fig. 1 (c), (d), and (e). The most important feature of these masks is that they can be implemented using efficient machine instructions to compute the SAD between two rows of eight or sixteen pixels; instructions that are readily available on most SIMD-capable general processors.

In previous work, in order to demonstrate that motion estimation algorithms are resilient to approximated metrics, we performed motion compensation on a series of standard QCIF and CIF video sequences, such as Akyio, Bus, Foreman, etc., using motion estimation algorithms such as EPZS [3], PMVFAST [10], UMHexS [11], and Full Search, and measured the resulting compensated image quality in PSNR, without any further quantization. Figures 2 and 3 show the compensated image quality for the various approximate SAD algorithms for Full Search and UMHexS. Tables 1, 2 and 3 summarize the loss incurred by the approximate metrics on the standard CIF sequences using the Full Search, EPZS, and UMHexS motion estimation algorithms. More results can be found in [4].

What the results indicate is that the use of the proposed approximate metrics incur small losses, and that the resilience of motion estimation algorithms is confirmed. Even the most important losses associated with the interlaced and the sparse masks are shown to be less than 1dB before quantization, while the loss associated with Subsampled Deintinterlaced and Deinterlaced masks is at most 0.1dB. The use of the proposed approximated metrics is therefore shown to be a perfectly valid strategy for complexity reduction in motion estimation algorithms.

3. IMPLEMENTATION DETAILS

A machine-specific implementation must rely on the shrewd exploitation of the underlying machine's ISA. However, modern CPUs exhibit complex behavior in the interaction between the algorithms, the instruction set, and the underlying architecture implementation and, consequently, a naïve approach to implementation will often lead to unsatisfactory results. CPUs from a same family sporting compatible instruction sets may have very different hardware implementations and display very different performance characteristics. If the im-

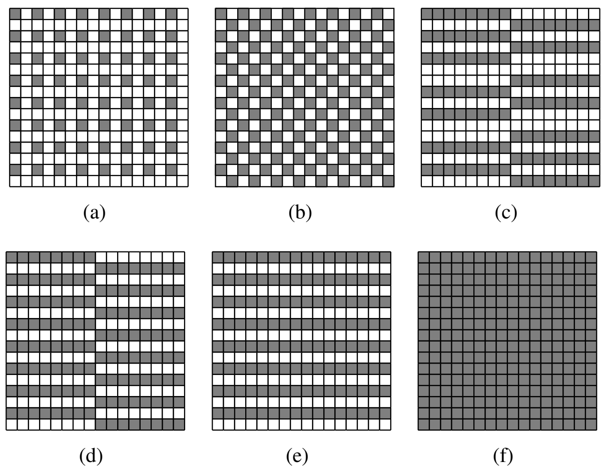

Fig. 1. Approximate metrics. (a) Sparse; (b) Quincunx; (c) Subsampled deinterlaced; (d) Deinterlaced; (e) Interlaced; (f) Full SAD. Shaded squares represent non zero elements in matrix M.

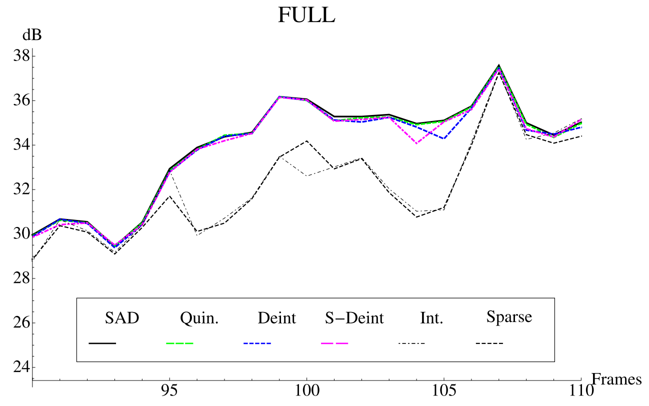

Fig. 2. PSNR resulting from the use of approximate metrics for the Foreman CIF sequence using Full Search. Frames 90–110 shown.

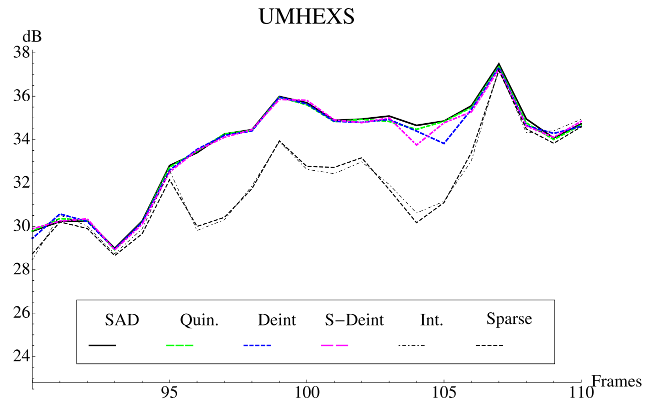

Fig. 3. PSNR resulting from the use of approximate metrics for the Foreman CIF sequence using UMHexS. Frames 90–110 shown.

CIF	SAD	Quin.	Deint	S-Deint	Int.	Sparse
Akiyo	42.8	-0.02	-0.05	-0.07	-0.05	-0.11
Bus	25.1	-0.01	-0.06	-0.10	-0.13	-0.34
Foreman	32.2	-0.03	-0.10	-0.11	-0.76	-0.86
News	36.5	-0.03	-0.06	-0.09	-0.10	-0.23
Mobile	25.2	-0.04	-0.07	-0.08	-0.09	-0.26
Stefan	26.0	-0.01	-0.08	-0.10	-0.12	-0.25
Tempete	27.0	-0.01	-0.04	-0.06	-0.05	-0.12

Table 1. PSNR (dB) for selected CIF sequences using Full Search. SAD, Quin., Deint, S-Deint, and Int. stand for the full SAD, the quincunx, the Deinterlaced, Sampled Deinterlaced and Interlaced patterns, respectively.

CIF	SAD	Quin.	Deint	S-Deint	Int.	Sparse
Akiyo	42.7	-0.02	-0.05	-0.07	-0.06	-0.11
Bus	24.3	-0.00	-0.05	-0.05	-0.17	-0.34
Foreman	31.9	-0.04	-0.11	-0.11	-0.73	-0.83
News	36.2	-0.03	-0.08	-0.12	-0.08	-0.24
Mobile	25.1	-0.03	-0.05	-0.06	-0.07	-0.23
Stefan	25.7	-0.01	-0.09	-0.09	-0.11	-0.22
Tempete	26.5	-0.02	-0.05	-0.07	-0.06	-0.13

Table 2. PSNR (dB) for selected CIF sequences using EPZS. See Table 1 for legend.

plementation of a given codec targets a particular CPU, the CPU's particular features can be exploited very effectively, but, in general, the implementation effort vs. resulting performance trade off will dictate that the implementation targets a CPU family, possibly of the same performance class, rather than a specific CPU. But, to optimize the implementation for a CPU family, one has several aspects to consider, which are discussed below.

Minimizing needed memory bandwidth and read latency is one such aspect. Bringing data into registers from memory, even when residing in cache, will incur a delay proportional to the width of the data read. To minimize read-related delays, the number of times the memory is accessed can be reduced by using alternative algorithms such as we propose here. The impact on the memory hierarchy can be further reduced by using non-temporal reads, hinting the memory controller to optimize cache usage. While we must ultimately give up on aligned-only memory access, alignment remains a concern. We can minimize alignment-related latency by carefully choosing the width of the data read from memory, given the target architecture. For example, on a system with a 32-bit memory bus, reading 64 bits will result in three reads $^3/_4$ of the time. On a system with a 128-bit memory bus, the same 64 bits reads result in two reads only $^7/_{16}$ of the time, yielding better performance, even at same system speed.

CIF	SAD	Quin.	Deint	S-Deint	Int.	Sparse
Akiyo	42.7	-0.02	-0.06	-0.07	-0.06	-0.12
Bus	24.6	-0.02	-0.06	-0.11	-0.17	-0.42
Foreman	31.9	-0.03	-0.11	-0.11	-0.74	-0.83
News	36.3	-0.04	-0.04	-0.07	-0.09	-0.24
Mobile	25.2	-0.04	-0.07	-0.09	-0.09	-0.33
Stefan	25.9	-0.01	-0.08	-0.10	-0.14	-0.24
Tempete	26.8	-0.02	-0.05	-0.07	-0.06	-0.15

Table 3. PSNR (dB) for selected CIF sequences using UMHexS. See Table 1 for legend.

CISC CPUs achieve high performance by breaking down complex instructions into micro-instructions, which are scheduled for super-scalar out-of-order execution [12]. To maximize throughput, instruction complexity and dependencies must be kept to a minimum. Addressing mode complexity, in particular, can be significantly reduced using compile time constant folding. While arbitrary image resolutions must be expected for still image processing, for video coding we can suppose that one of the few conventional formats—QCIF, CIF, VGA, SD, etc.—is used, so it is quite conceivable to specialize the most computationally intensive functions for these resolutions. In this case, specialization is achieved by precomputing relative displacements within the image buffer and propagating the constants as immediate values in the instructions themselves at compile time.

Data dependencies can also be reduced or eliminated. In the case of the computation of the metric, the dependencies are mostly read dependencies, as the pixels are read from memory without write-back, and the write dependencies are limited to the computation of the sum. Due to the additive nature of the metrics considered, the computation can be broken down into n independent sums (thereby reducing write dependencies by using separate registers), which can be later combined in at most $O(\lg n)$ to obtain the final value.

A modern processor minimizes the impact of conditional jumps by predicting whether or not jumps will be taken. The processor prepares for the most likely outcome—as determined by its branch prediction algorithm—by prefetching the instructions at the predicted jump location and starting to execute them. If the branch is taken as predicted, the execution flows into the prefetched (and possibly already executing) code. If the jump is mispredicted, however, severe performance penalty may be incurred. Not only does the processor need to fetch instructions at the alternate jump location, it may also have to discard code already executing with a penalty proportional to its pipelines' depths. Mispredicted jumps must be avoided and the surest way of achieving this is to unroll loops, which eliminates conditional jumps altogether. While we now face a code size vs. speed trade off, unrolling loops with a relatively small number of iterations is generally beneficial to execution speed.

High-level language features, like the calling convention, play a role in implementation-specific performance. The calling convention determines how arguments are passed and how functions are called. Favoring a faster calling convention whenever possible will lead to better performance. For example, the default calling convention on 32-bit mode x86 processors consists of pushing the function's arguments onto the stack before the call is performed. On 64-bit mode x86_64 processors, the calling convention uses the more efficient passing by register strategy. Skipping the construction of the local stack frame will also yield—although in this case modest—performance improvements. Call convention and stack frame building can be modified using compiler-specific language extensions.

Most, but not all, optimization aspects we discussed can be addressed through standard C (or C++) language programming and compiler extensions. A certain degree of control over parallelism, call conventions, and instruction generation is afforded through library and compiler-specific extensions such as "pragmas". While an optimizing compiler will use its knowledge about the target processor and sophisticated heuristics to detect vectorization and generate code using SIMD instructions, it will ultimately prove difficult to control for obtaining results that are always satisfactory, as we will show.

The solution is to write a small set of critical routines for the target processor(s), like the SAD, in assembly language directly. Because the programmer has total control over the instructions executed by the processor, it is possible to fine-tune the implementation until performance expectations are met, a result that would otherwise be impossible to attain using only high level C code and relying on the compiler and its various extensions for auto-vectorization and aggressive optimizations.

We implemented the proposed approximate metrics (the sparse, quincunx, subsampled deinterlaced, deinterlaced, and interlaced metrics), as well as the full SAD, as shown in Fig. 1, using three techniques. The first is an implementation in C++ language using non-vectorized but otherwise optimized compilation with ICC and G++. The second uses the same compilers but with vectorization and aggressive optimizations enabled. The third consists of an assembly language implementation using SSE2-level instructions and omitted stack frames. The assembly language implementations address the concerns discussed in this section for best performance. All implementations were ported to both x86 and x86_64 CPUs using the most adequate calling conventions.

The choice of a CPU to conduct a representative performance test is an arduous one. Rather than targeting a specific CPU or assuming a particular application (for example, high-end server for bulk coding or transcoding), we compared the speed-ups obtained on a number of readily available CPUs, covering the performance spectrum as widely as possible, ranging from the Intel Atom N270 to the Xeon E5530 (using the i7 architecture, codenamed *Nehalem*), including other CPUs such as the Intel Mobile Pentium 4 and the AMD 3500+. The full list is found in Table 4.

Tests were conducted on the GNU/Linux operating system, specifically the Ubuntu distribution, version 8.04 LTS, using the GNU C++ compiler (g++) v4.2 and the Intel C Compiler v11.0. We used the function computing the SAD (`ippiSAD16x16_8u32s` with option `IPPVC_MC_APX_FF`) from the Intel Performance Primitives Libraries v6.0 as a useful benchmark. Timings were obtained using an OS-specific timing function (`gettimeofday`) and are accurate to $\pm 2\%$. We present the performance results obtained in section 4 and discuss them in section 5.

Processor	Family	Clock (GHz)	Mode
Intel E5530	i7	2.40	x86_64
Intel N270	Atom	1.60	x86
Intel 6700	Core 2	2.66	x86
Intel 6400	Core 2	2.13	x86_64
Intel T2500	Core	2.00	x86
Intel 5130	Core 2	2.00	x86_64
Intel P8700	Core 2	2.53	x86_64
Intel Mobile P4	P4	3.06	x86
AMD 3500+	Athlon	2.21	x86_64

Table 4. Processors considered in our study.

4. RESULTS

Fig. 4 and Table 5 summarize the results for all the processors available to our study. The absolute performances of the implementation of the approximated metrics, given the processors, vary greatly, but a more careful examination of Fig. 4 reveals that the relative performance of the efficient implementation of approximate metrics remains about the same from one processor to another.

Figs. 5 and 6 present the relative performance of our proposed metrics implemented in SSE2-level assembly language versus what is gained through autovectorization and optimizing compilers. Again, the IPP implementation is used as the benchmark. Finally, Fig. 7 compares the Intel E5530 processor (i7 architecture) against the Intel N270 (Atom) on the same scale. The performance of the E5530 dwarfs that of the N270, but again we observe that the relative performances of methods (SAD vs. Sparse, for example) remains similar across processors.

5. DISCUSSION

Unless special quality requirements are set for coding or transcoding, the loss incurred by the use of the proposed metrics is quite acceptable, as we have shown in previous work [4]. While the approximate metrics using the interlaced and sparse masks are the fastest methods, their potentially significant losses, although always less than 1 dB, makes them comparatively uninteresting as the S-Deint method, which is only slightly slower, incurs losses of at most 0.1 dB.

From the various tables and figures, we can gather than the auto-vectorizing compiler does not always recognize the potential for vectorization from the C code, even though it was carefully crafted to help the compiler detect auto-vectorization. In Fig. 6, we see that both compilers fail to produce auto-vectorized code that comes even close to the proposed or IPP implementations. These results indicate that, in fact, the compilers either failed to recognize the potential for vectorization (for G++ 4.2) or failed to recognize the machine's full capabilities (for ICC 11.0) to generate efficient code. Examining Table 5, we see that on other machines, where vectorization succeeded the auto-vectorization

		Number of calls per µs								
Processors	GHz	C++	C++*	SAD	Sparse	Quin.	S-Deint	Deint	Int	IPP
Intel Xeon E5530	2.40	2.32	3.30	18.82	39.47	18.26	34.41	26.15	42.26	16.16
Intel Atom N270	1.60	0.75	0.75	1.82	3.12	1.71	2.94	2.59	3.07	*1.56*
Intel Core 2 6700	2.66	2.38	3.36	7.89	14.46	7.71	13.40	11.14	14.83	6.89
Intel Core 2 6400	2.13	1.95	2.73	6.15	10.97	6.05	10.38	8.60	11.00	5.83
Intel Core T2500	2.00	1.31	1.77	6.00	10.44	5.89	10.05	8.43	10.64	5.42
Intel Xeon 5130	2.00	2.55	2.55	5.79	10.70	5.77	10.90	8.82	10.90	*4.97*
Intel Centrino P8700	2.53	2.32	3.27	7.70	14.22	7.68	12.75	10.71	14.10	7.47
Mobile Pentium 4	3.06	1.58	1.58	8.21	12.70	7.28	13.95	12.09	14.07	*7.05*
AMD Athlon 3500+	2.21	1.32	1.32	3.99	7.35	3.95	6.09	5.55	7.59	*3.43*

Table 5. Performance results for all tested processors on CIF images, in metric computation per µs. C++ are results from the full SAD without vectorization, using G++ 4.2.x. C++* results are from the full SAD using G++ 4.2.x with vectorization. Other methods are SSE2 optimized. Results in italics because IPP was not available on the machine tested.

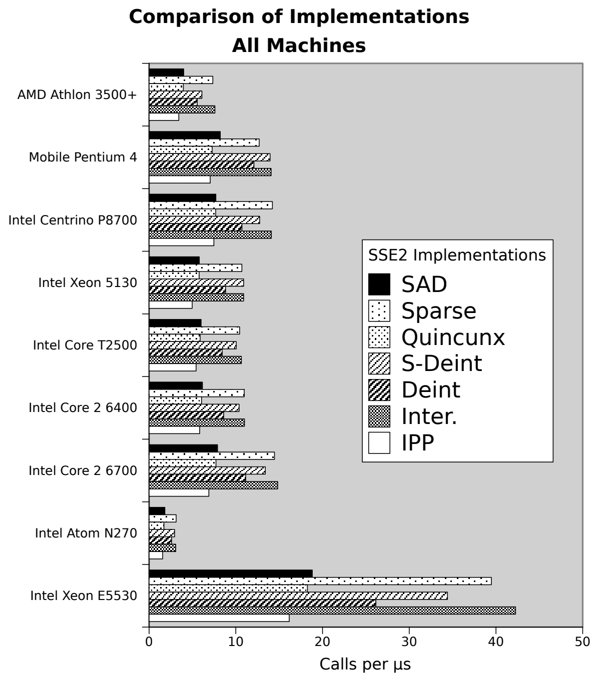

Fig. 4. Performance of SSE2-level implementations on all the machine considered, in calls by µs (more is better). IPP denotes Intel's Implementation of the full SAD.

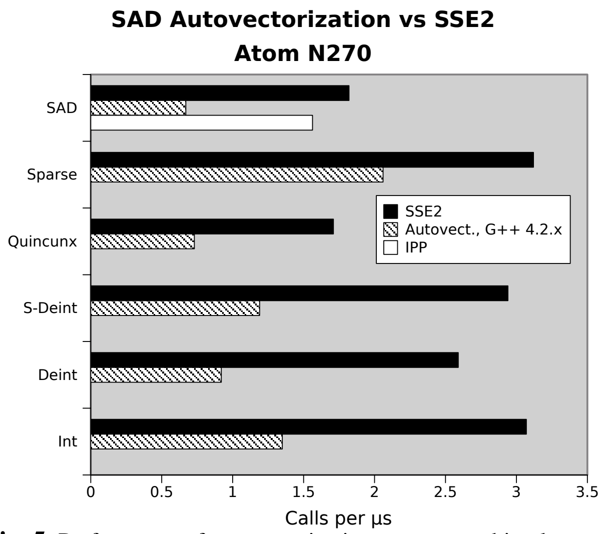

Fig. 5. Performance of autovectorization vs. proposed implementations for the Atom N270, in calls by µs (more is better).

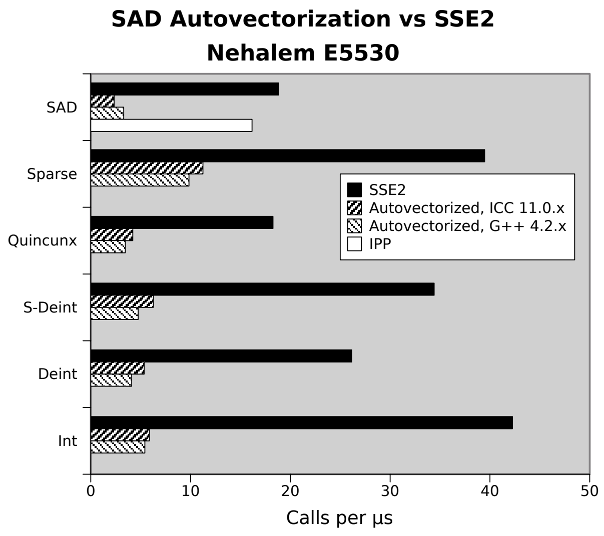

Fig. 6. Performance of autovectorization vs. proposed implementations for the Xeon E5530, in calls by µs (more is better).

yielded modest speed-ups, not even 2:1, while the carefully crafted SSE2-level assembly language implementations yielded speed-ups ranging from 6:1 to over 12:1 relative to auto-vectorized C code, depending on the sparseness of the approximated metric considered. Fig. 6 compares the auto-vectorization results from G++ with ICC. ICC bests G++ in most cases, but not by very much. The experiments therefore show that our remark that it will ultimately prove difficult to coax the compiler into generating efficient code is well founded.

The figures show that the constant-propagated full SAD implementation beats the IPP v6.0 implementation by $\approx 15\%$. The generic nature of IPP precludes the use of full constant propagation and instruction simplification, resulting in perfor-

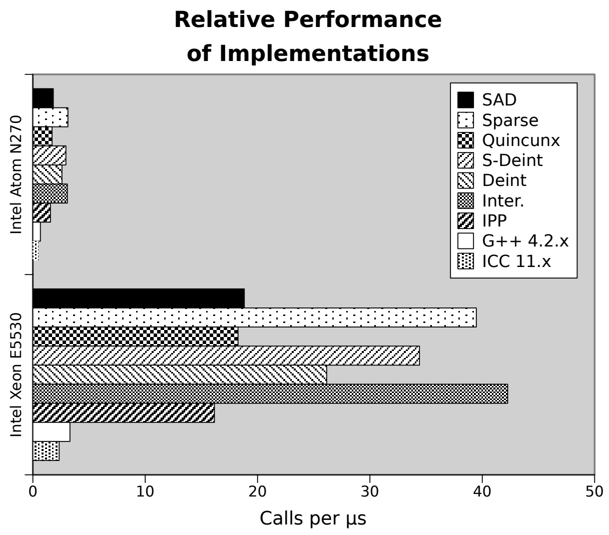

Fig. 7. Performance of implementations for the Xeon E5530 vs. the Atom N270, to scale, in calls by µs. Results from G++ 4.2 and ICC 11.0 correspond to non-vectorized optimized C++ code for the full SAD. Other methods are SSE2 optimized.

mance pessimization. While a gain of $\approx 15\%$ is interesting, the speed-ups obtained by using the proposed approximate metrics are even more so, as they reach as much as 2.6:1 relative to IPP's full SAD implementation, while leading to a negligible loss in quality, as shown in Tables 1–3.

While each of the CPU tested sports conspicuously different characteristics—architecture, power consumption, etc.—the results indicate that in all cases there is a performance gain to be had by using SSE2-level implementations of approximated metrics, and that the relative speed-ups are similar regardless of architecture. While exact magnitudes vary from processor to processor, the SSE2 implementation of the full SAD beats the IPP implementation by the same 10 to 15%, and the Interlaced, Sparse, and Subsampled Deinterlaced approximated metrics remain much faster than the other approximated metrics. These results indicate that we can afford considerable code specialization before seeing any machine-specific impact on performance, as the performance characteristics remain similar across a wide range of processors of different generations, families, and even makers.

We have shown that using the machine ISA to its full extent allows access to speed-ups that are impossible to obtain with C++ compiled with optimizations and auto-vectorization enabled. One reason for this is that the compilers are not always able to exploit the SIMD potential of C++ code. We also have shown that, if we are willing to sacrifice motion estimation precision by using approximate metrics, there are impressive speed-ups available. However, since we will also want to minimize the maximum average error, approximated metrics such as the Interlaced and Sparse approximated metrics are to be avoided. The Subsampled Deinterlaced and Deinterlaced metrics simultaneously afford large speed-ups and very small quality loss—less than about 0.1 dB.

6. CONCLUSIONS

In this paper, we have shown that the proposed implementations yield consistent speed-ups across many processors of different generations, families, and even makers. We have also shown that the use of approximate metrics and SSE2-level implementations addressing many architectural concerns yield speed-ups of as much as 12:1 relative to non-vectorized C code depending on the best approximated metrics considered. Future work will include characterization of speed-ups using approximated metrics in codecs such as MPEG-4 and MPEG-4 AVC/H.264.

7. REFERENCES

[1] P. M. Kuhn, G. Diebel, S. Hermann, A. Keil, H. Mooshofer, A. Kaup, R. Mayer, and W. Stechele, "Complexity and PSNR comparison of several motion estimation algorithms for MPEG-4," *Procs. SPIE*, pp. 486–489, 1998.

[2] Y.-L. Lai, Y.-Y. Tseng, C.-W. Lin, Z. Zhou, and M.-T. Sun, "H.264 encoder speed-up via joint algorithm/code-level optimization," *Procs. SPIE VCIP*, July 2005.

[3] A. M. Tourapis, "Enhanced predictive zonal search for single and multiple frame motion estimation," in *Visual Communications and Image Processing*, Jan. 2002, pp. 1069–1079.

[4] S. Pigeon and S. Coulombe, "Speeding up motion estimation in modern video encoders using approximate metrics and SIMD processors," *IEEE Symposium on Industrial Electronics and Applications (ISIEA)*, pp. 233–238, Oct. 2009.

[5] H.-Y. Cheong, I. S. Cheng, and A. Ortega, "Computation error tolerance in motion estimation algorithms," *Int. Conference on Image Processing (ICIP)*, pp. 3289–3292, Oct. 2006.

[6] F. Tombari and S. Mattoccia, "Template matching based on the l_p norm using sufficient conditions with incremental approximations," in *Procs. IEEE int. Conf. on Advanced Video and Signal-Based Surveillance*, Nov. 2006, pp. 20–26.

[7] B. Liu and A. Zaccarin, "New fast algorithms for the estimation of block motion vectors," *IEEE Trans. Circuits and Systems For Video Technology*, vol. 3, no. 2, pp. 148–157, Apr. 1993.

[8] C.-K. Cheung and L. m. Po, "A hierarchical block motion estimation algorithm using partial distortion measures," *Int. Conference on Image Processing (ICIP)*, vol. 3, pp. 606–609, 1997.

[9] Y.-L. Chan and W.-C. Siu, "New adaptive pixel decimation for block motion vector estimation," *IEEE Trans. Circuits and Systems For Video Technology*, vol. 6, no. 1, pp. 113–118, Jan. 1996.

[10] A. M. Tourapis, O. C. Au, and M. Liou, "Predictive motion vector field adaptive search technique (PMVFAST) - enhancing block based motion estimation," in *Int. Conference on Image Processing (ICIP)*, Jan. 2001.

[11] Z. Chen, P. Zhou, and Y. He, "Fast integer and fractional pel motion estimation for JVT," Tech. Rep. JVT-F017, Dec. 2002.

[12] J. Hennessy and D. Patterson, *Computer Architecture: A quantitative Approach*, Morgan Kauffman, 4th edition, 2006.