Smaller and faster Intel SSE Code

Stefan Kral

skral@complang.tuwien.ac.at,
WWW home page: http://www.complang.tuwien.ac.at/skral

Abstract. Codes exhibiting suitable data-flow parallelism can often profit from using Intel SSE, a SIMD extension to the CISC style Intel 64 instruction set architecture. As SSE instructions are, on average, larger than scalar instructions, they exhibit a heavier load on instruction pre-decoding, decoding, and caching hardware. For long straight-line SSE codes, instruction lengths become an obstacle to high performance that is not adequately handled by available optimizing compilers. This paper presents three orthogonal code optimization techniques that minimize SSE code size and runtime: (*i*) address code generation specifically aimed at signal transform access patterns, (*ii*) offset assignment for values residing on the stack, (*iii*) SSE register reassignment using genetic optimization. For compute kernels of the renowned signal transform library FFTW, the techniques in this paper, when combined, yield up to 30% improvement in code size and performance, compared to the best codes produced by the optimizing Intel C compiler.[1]

Keywords: Special purpose compilation, code optimization, signal transforms, Intel 64, AMD64, SSE

1 Introduction

Over the course of its more than 30 year history, the Intel x86 [13] instruction set architecture (ISA) has evolved substantially while retaining binary compatibility: Starting out as a 16-bit extension of Intel's earlier 8-bit instruction set architecture in 1978, it became a 32-bit architecture with virtual memory protection and multitasking capabilities in 1985. In 2003, it gained 64-bit integer and 64-bit virtual address space support and was named "Intel 64".

Today, the vast majority of the more than one billion PCs [7] in worldwide use and almost 90% of the world's most powerful TOP500 supercomputers [17] are powered by Intel 64 compatible processors.

SSE1–4. Between 1999 and 2008, Intel introduced several instruction set extensions to the Intel x86 and Intel 64 instruction set architectures named Streaming SIMD Extensions [13]. SSE aims at accelerating multimedia and scientific applications by speeding up compute-intensive codes exhibiting Single Instruction Multiple Data (SIMD) style parallelism. SSE is successfully used in fields like

[1] Supported by the Austrian Science Fund (FWF), Contract No. P19939-N13

linear algebra, digital signal processing, video encoding, image processing, and 3D computer graphics.

SSE instructions operate on fixed-size 128-bit vectors, consisting of multiple scalar values packed together: sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, two 64-bit integers, four 32-bit floating-point numbers, or two 64-bit floating-point numbers.

SSE defines a dedicated vector register file that is distinct from the general-purpose scalar integer register file. In total, SSE1–4 comprises 306 instructions, used for integer and floating-point arithmetics, data-transfer to/from memory, data permutation within vectors, and cache/memory management.

Intel Core Microarchitecture. Originally introduced in 2006, the Intel Core microarchitecture and its derivatives form the base of most Intel 64 mobile, desktop, and server processors. Intel Core based processors have a market share of approximately 80% both among PCs and Top500 supercomputers.

The roots of the Intel Core microarchitecture reach back to the Intel Pentium Pro processor introduced in 1995. Intel Core is a 4-way superscalar [16], 14-stage super-pipelined design [2] with in-order Intel 64 instruction fetch, in-order Intel 64 instruction predecoding, in-order dynamic translation of Intel 64 instructions to fixed-width, RISC-style micro-operations (instruction decoding), aggressive out-of-order execution of micro-operations, and in-order Intel 64 instruction completion.

Instruction Predecoding. Intel 64 instructions are variable in length. To enable superscalar execution, the processor needs to determine multiple instruction lengths in parallel during instruction predecoding [12].

The Intel Core microarchitecture does not cache the results of instruction predecoding—a design choice which helps save chip space, but may necessitate repeated predecoding [2]. As Intel Core predecoders process at most 16 bytes per cycle, predecoding becomes a bottleneck whenever average instruction lengths exceed 4 bytes, which is common in large numerical SSE kernel routines used in scientific computing.

2 Intel 64 Instruction Lengths

Intel 64 is a complex instruction set computing (CISC) architecture: instructions may have different lengths to allow the encoding of complex addressing modes, immediate integer constants, and the direct usage of in-memory operands.

These features can help reduce the instruction count but contribute to bigger instruction lengths: instruction encoding is most compact with simple addressing modes, immediate constants no larger than 8-bit, and in-register operands.

Intel 64's New Registers. Compared to Intel x86, Intel 64 doubles the number of scalar general-purpose integer registers and SSE vector registers available to 64-bit code on the assembly level, which can reduce register pressure significantly.

Because of Intel 64 opcode space expansion details, many instructions get encoded differently when using any of the new registers, making their instruction lengths bigger by one byte.

SSE Instruction Lengths. Table 1 compares the instruction lengths of a selection of packed double-precision floating-point 128-bit SSE vector instructions: For different operations and their corresponding assembly-level instruction format, instruction lengths in bytes are presented. Instructions using one or more SSE registers and instructions with in-memory operands vary considerably in length, depending on the addressing modes and SSE vector registers used.

Table 1. Comparison of 128-bit SSE vector instructions

Operation	Instruction Format	Length
Copy	movapd reg,reg	4–5
Load	movapd mem,reg	4–10
Store	movapd reg,mem	4–10
Add	addpd reg,reg	4–5
Add	addpd mem,reg	4–10
Multiply	mulpd reg,reg	4–5
Multiply	mulpd mem,reg	4–10

Scalar Integer Instruction Lengths. Table 2 compares the instruction lengths of a selection of scalar integer instructions commonly used for address calculation: For different operations and their corresponding assembly-level instruction format, instruction lengths in bytes are presented. Like in table 1, instructions vary in length, depending on the registers, adressing modes, and size of the immediate constant `K` used.

Table 2. Comparison of 64-bit scalar integer instructions

Operation	Instruction Format	Length
Copy	movq reg,reg	3
Load	movq mem,reg	3–8
Store	movq reg,mem	3–8
Shift left and add	leaq mem,reg	3–8
Multiply by constant `K`	imulq `K`,reg,reg	4–7
Multiply by constant `K`	imulq `K`,mem,reg	4–12
Add	addq reg,reg	3
Subtract	subq reg,reg	3
Shift left by constant `K`	shlq `K`,reg	3–4

Instruction Mix. Compute-intensive numerical SSE kernel codes often consist of a large number of SSE instructions, which do the bulk of the computation, and a relatively small portion of scalar integer code used for address generation.

3 SSE Code Optimization

All three optimization techniques presented in this paper target large straight-line Intel 64 SSE codes: Subsection 3.1 introduces a technique generating address code for accessing variably strided 1D arrays, which is of particular interest for numerical codes in the signal transform domain. Subsection 3.2 presents a heuristic for rearranging the layout of the thread-local procedure stack frame to replace as many 32-bit integer constants by 8-bit constants as possible. Subsection 3.3 shows how SSE registers can be reassigned to further reduce overall codesize.

Related Work. [18] aims at constructing FPGA/ASIC multiplier blocks for multiple constant multiplications with the least number of additions and subtractions.

Previous work [10] targeted address code generation for a different hardware architecture, IBM PowerPC, a reduced instruction set computing (RISC) architecture that offers specific auto-modify addressing modes typical of DSP's.

An alternative solution to accessing variably strided arrays is implemented in [5], precomputation of constant integer multiplications.

Offset assignment [11] is a prominent research issue on digital signal processors (DSP), which often feature addressing modes like auto-modify with a constant range. Intel 64 processors lack these addressing modes.

3.1 M1: Address Code for accessing variably strided 1D Arrays

Addressing modes supported by most processors, including Intel 64, are targeted at consecutive memory accesses. Variably-strided 1D accesses—and accesses to multi-dimensional arrays stored as single contiguous blocks [8]—incur, by comparison, relatively high costs for address generation and decrease performance.

The technique presented here heavily relies on Intel 64 addressing modes, which generally have the form *reg + reg * {1,2,4,8} + offset*: At most, this allows to combine two additions and one (limited) shift left with a instruction using an in-memory operand.

Example. A 6-point signal transform kernel code operates on an input array X with variable stride S and an output array Y with variable stride T. It may first read the input array elements $X[4*S], X[0], X[3*S], X[5*S], X[2*S], X[1]$, then process all the data, and may finally write the transformed data to the output array elements $Y[0], Y[3*T], Y[4*T], Y[2*T], Y[T], Y[5*T]$.

No assumptions may be taken about the particular order in which the input elements are read or the output elements are written, primarily as the particular order may be the result of domain-specific high-level code scheduling [4].

Optimization Preparation. As we operate on the assembly-level, we first perform abstract interpretation of the code to reconstruct the expression trees used for address generation.

Then, we determine which combinations of arrays, strides, and integer constants are used.

Upon encountering access patterns similar to the one in the example, we eliminate the code corresponding to the expression trees if it is solely used for address generation, which frees integer registers that then become available for new, optimized auxiliary calculations.

Address Code Generation. For every array-stride combination we consider splitting the input array into 2, 3, 4, or 8 equally-sized parts. For each part, a new base register is allocated and calculated.

When generating code for a single array access, the nearest possible base register is used, which considerably reduces the number of auxiliary integer instructions required. Whenever possible, constant multiplications by integers are replaced by cheaper operations, like the Intel 64 specific limited-shift-and-add integer instruction `lea`.

Whenever register pressure gets to high, index and base registers are reused using a least-recently-used heuristics.

Analysis shows that the splitting techniques reduces well in practice, especially for very large codes.

3.2 M2: Offset Assignment

When the amount of live variables in a block exceeds the number of available registers, auxiliary space needs to be allocated on the stack to hold excess values.

The Application Binary Interface (ABI) [15] for Intel 64 processors defines one particular general-purpose integer register (`%rsp`) as the stack pointer that must always point to the end of the latest allocated procedure stack frame.

Table 3. Comparison of SSE vector loads with different offsets and base registers

Assembly instruction	Offset encoding	Length
`movapd 0(%rsp),%xmm0`	special encoding for zero	5
`movapd 32(%rsp),%xmm0`	8-bit signed integer	6
`movapd 1024(%rsp),%xmm0`	32-bit integer	9
`movapd 0(%rax),%xmm0`	special encoding for zero	4
`movapd 32(%rax),%xmm0`	8-bit signed integer	5
`movapd 1024(%rax),%xmm0`	32-bit integer	8

Compact Address Encoding. Table 3 shows SSE load instructions performing a register-indirect memory access with three different constant offsets and two different base registers, `%rsp` and `%rax`.

The constant offset zero is encoded most compactly, followed by 8-bit signed integers, trailed by 32-bit integers.

Accesses using register `%rsp` as base are encoded less compactly than accesses using integer registers `%rax` (shown in table 3), `%rbx`, `%rcx`, `%rdx`, `%rdi`, and `%rsi`.

Stack Pointer Aliasing. N of the above integer registers may be aliased with `%rsp` to serve as auxiliary stack pointers. With proper offsetting, they cover disjoint 256-byte stack regions accessible using small offsets.

Integer registers are scarse on Intel 64. As they are also used for passing parameters and calculating effective addresses, we found $N = 1$ to be the best choice for the signal transform kernel codes considered in this paper.

Stack Reassignment. To minimize the number of stack accesses having offsets not fitting 8-bit integers, stack variables are reassigned.

First, the stack is grown by $256(N + 1)$ bytes. All auxiliary stack pointers are properly offsetted from `%rsp`.

Then, the largest possible number of stack variables is assigned to the newly allocated stack area by using a greedy approximation algorithm: For every location L in the newly allocated stack area, select an unassigned stack-variable V, whose lifespan does not overlap with any variable already assigned to L, and assign V to L. Repeat until all locations are full or all variables are assigned.

When selecting stack-variables in the above algorithm, we heuristically prefer the ones with shorter lifespans.

3.3 M3: SSE Register Reassignment

While SSE registers `%xmm0` through `%xmm7` (xmm0-7) are available with Intel x86, new registers `%xmm8` through `%xmm15` (xmm8-15) are only available with Intel 64. Instructions get bigger when using xmm8-15, as shown in table 4.

Table 4. Comparison of variants of SSE instruction “addpd”

Assembly instruction	Length
`addpd %xmm0,%xmm1`	4
`addpd %xmm0,%xmm8`	5
`addpd %xmm8,%xmm0`	5
`addpd %xmm8,%xmm9`	5
`addpd (%rax),%xmm0`	4
`addpd (%rax),%xmm8`	5

Few Constraints. Unlike the scalar part of the Intel x86 instruction set, only very few SSE instructions use fixed or implicit registers.

The Intel 64 ABI [15] declares all SSE registers as caller-saved. When not holding procedure arguments, all registers are freely available as scratch registers.

Preparation. Intel 64 instructions use at least zero and at most three SSE registers as operands. For every instruction, we consider the number of SSE registers, r, and build 2^r distinct variants by mapping each of the r SSE operand registers alternately to `%xmm0` and `%xmm8`. All variants are collected and passed on to the assembler to precisely determine instruction lengths.

Procedure-level data-flow analysis [3] is performed to obtained liveness information, determining which SSE registers are fully local to each basic block.

SSE Register Reassignment. We operate on single basic blocks: Initially, all block-local SSE registers are marked free. After the last use of block-local SSE register, the register is marked as free. Whenever a block-local SSE register is fully overwritten defined, that register may be replaced by any free register from either xmm0-7 or xmm8-15, provided that all subsequent uses are replaced consistently—to preserve program semantics.

Replacement choices can easily be steered by a bit-string: "Prefer xmm0-7 if bit is zero, xmm8-15 otherwise." This eases testing different settings (constant-0, constant-1, random), local and glocal optimization methods.

We evaluated a number of combinations and found the following two particularly noteworthy: (1) "Start out preferring xmm0-7 whenever possible, and then perform local hill-climbing search" produces good solutions in less than a second. (2) "Randomize initial population of size 1000, run generational genetic algorithm, and then perform local hill-climbing search" takes substantially more time to complete, but consistently produces the best results.

4 Experimental Setup and Results

All techniques presented in this paper were implemented in a Intel 64 assembly level source-to-source code optimizer named NXyn [9].

To assess the impact on scientific SSE codes, kernel routines of the renowned discrete Fourier transform library FFTW [6] were compiled with the latest version 11.1 of the optimizing Intel C compiler [14] using maximum optimization flags. The assembly codes produced by the Intel C compiler served as baseline and were post-processed by NXyn.

All performance tests were conducted on a Intel Core i3-350M processor, as detailed in table 5. Only a single processor core was used in the experiments.

4.1 Code Size Improvements

Table 6 shows the impact of optimization on a selection of FFTW double-precision DFT kernel codes: The column "icc" refers to the code that was produced by the Intel C compiler with optimization flags "-O3 -inline-forceinline".

Table 5. Hardware setup

Processor name	Intel Core i3-350M (“Arrandale”)
Microarchitecture	Westmere
Process technology	32nm
Number of cores	2
Number of threads	4
Clock speed	2.26 GHz
L1 instruction cache size	32 kB
L1 data cache size	32 kB
L2 cache size	2x 256 kB
L3 cache size	3 MB
Memory interface	Dual-channel DDR3

It serves as a baseline for comparison. The column “icc precalc” refers to code that uses effective address precomputation. The column “M0” refers to NXyn without any of the optimizations shown in this paper. The column “M1” shows the effect of optimized address generation, as presented in subsection 3.1. The column “M1–2” shows the addition of offset assignment, as detailed in subsection 3.2. The column “M1–3” shows the addition of SSE register reassignment, as shown in subsection 3.3.

Discussion. Every optimization (M1,M2, and M3) reduced the codesize of some kernel code by at least 5%. Typically, M1 yielded the largest, M2 the second largest, and M3 the smallest reductions. Only in very few, typically small codes, a combination which meant to improve codesize actually increases it (emphasized in the table data.)

While effective address precomputation “icc precalc” gave noticable codesize reductions, its impact was less than M1 in almost all cases (including all shown in above table).

Manual analysis of the produced assembly codes reveals that, for large kernel codes, all optimizations were maxed out.

4.2 Runtime Improvements

Figures 1 and 2 show the performance of the generated codes, for power-of-two and non-power-of-two transform sizes respectively.

Measurements show that, in the power-of-two case (figure 1), the performance of already highly-optimized code is typically increased 10% or more by applying methods M1–3.

Performance increases for non-power-of-two cases (figure 2) are smaller than in the power-of-two cases, mainly due to the smaller codesizes of the kernels employed.

Table 6. Comparison of FFTW double-precision DFT kernel code sizes

Kernel name	icc (100%)	icc precalc	M0	M1	M1–2	M1–3
`n1_15`	2976	91.3%	103.2%	85.0%	75.8%	70.5%
`n1_16`	2576	84.4%	98.8%	77.7%	72.1%	67.1%
`n1_20`	3872	88.4%	101.2%	81.4%	72.3%	67.8%
`n1_32`	6384	87.9%	102.0%	83.4%	74.2%	69.2%
`n1_64`	16000	90.4%	101.4%	86.2%	76.7%	71.4%
`n1fv_12`	1040	73.8%	100.1%	69.4%	*70.9%*	66.3%
`n1fv_13`	1744	83.4%	100.1%	79.0%	73.5%	68.9%
`n1fv_16`	1536	70.8%	100.1%	64.7%	62.6%	60.5%
`n1fv_20`	2176	77.9%	103.0%	70.6%	64.8%	61.8%
`n1fv_25`	5040	86.9%	101.6%	84.4%	73.0%	68.6%
`n1fv_32`	3776	77.9%	100.9%	67.4%	61.4%	58.1%
`n1fv_64`	8816	79.6%	103.6%	71.5%	66.4%	61.5%
`n1fv_128`	20560	83.8%	103.2%	75.3%	68.1%	65.5%
`n1fv_256`	46592	85.7%	103.1%	77.9%	70.4%	66.9%
`t1fv_15`	2096	89.3%	98.5%	98.5%	*99.3%*	91.7%
`t1fv_16`	2080	85.3%	97.0%	97.0%	96.2%	90.1%
`t1fv_32`	4688	93.8%	97.6%	80.9%	75.8%	69.6%
`t1fv_64`	10912	92.6%	99.8%	81.9%	77.2%	70.6%
`t1fv_128`	24256	93.5%	100.7%	84.3%	79.7%	78.1%

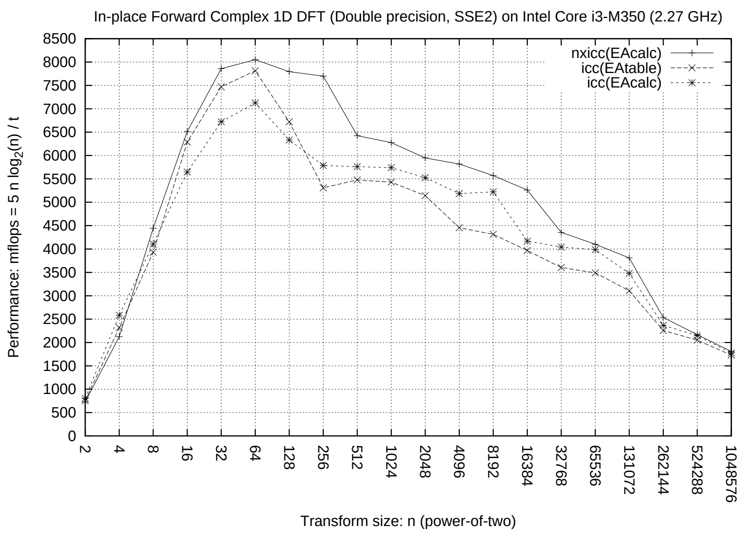

Fig. 1. Comparison of In-place Complex DFT Performance (powers-of-two)

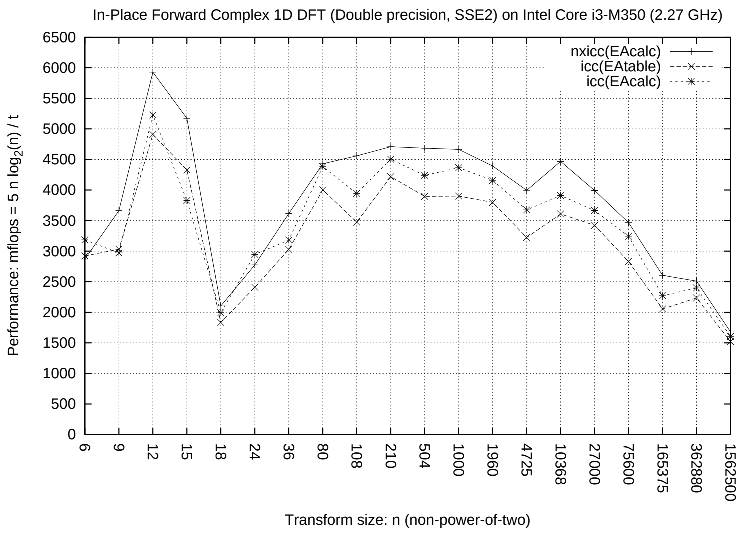

Fig. 2. Comparison of Complex DFT Performance (non-powers-of-two)

5 Conclusion

This paper presented three code optimization methods for large SSE codes, which are mostly orthogonal to each other and to existing optimization techniques in the state-of-the-art Intel C compiler. The methods can significantly optimize already highly optimized code, both with regard to codesize and performance, as shown by measurements on current Intel Core i3 processor.

References

1. Advanced Micro Devices, Inc.: Software Optimization Guide for AMD Family 10h and 12h Processors. (2011) `http://support.amd.com/us/Processor_TechDocs/40546.pdf`
2. Agner Fog: The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. (2011) `http://agner.org/optimize/microarchitecture.pdf`
3. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge, United Kingdom. (2004)
4. Frigo, M.: A Fast Fourier Transform Compiler, Proceedings of PLDI, 169–180 (1999)
5. Frigo, M., Johnson, S.G.: FFTW 3.2.2 Library Source-Code. `http://fftw.org/fftw-3.2.2.tar.gz`
6. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3, Proceedings of the IEEE 93 (2), 216–231 (2005)

7. Gartner, Inc.: Gartner Says More than 1 Billion PCs In Use Worldwide and Headed to 2 Billion Units by 2014. Press Release. (2008)
`http://www.gartner.com/it/page.jsp?id=703807`
8. Knuth, D.E.: The Art of Computer Programming. Volume 1: Fundamental Algorithms. Third edition. Addison-Wesley, New York (1997)
9. Kral, S.: NXyn Home Page. `http://www.complang.tuwien.ac.at/skral/NXyn/`
10. Kral, S., Triska, M., Ueberhuber, C.W.: Compiler Technology for Blue Gene Systems, In Proceedings of Euro-Par 2006 – International Conference on Parallel and Distributed Computing, 279–288. Springer Lecture Notes in Computer Science (LNCS), vol. 4128.
11. Leupers, R.: Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In 12th International Conference on Compiler Construction, 290–302, Warsaw, April 2003.
12. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual. (2011)
`http://www.intel.com/Assets/PDF/manual/248966.pdf`
13. Intel Corporation: Intel 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture. (2011)
`http://www.intel.com/Assets/PDF/manual/253665.pdf`
14. Intel Corporation: Intel Software Network: Compilers and Libraries.
`http://software.intel.com/en-us/articles/intel-compilers/`
15. Matz, M., Hubička, J., Jaeger, A., Mitchell, M.: System V Application Binary Interface—AMD64 Architecture Processor Supplement. (2010)
`http://www.x86-64.org/documentation/abi.pdf`
16. Patterson, D.A., Hennessy, J.L.: Computer Organization & Design: The Hardware/Software Interface. Second edition. Morgan Kaufmann, San Francisco (1998)
17. TOP500.org: TOP500 Supercomputing Sites: Processor Family Share Over Time.
`http://www.top500.org/overtime/list/36/procfam`
18. Voronenko, Y., Püschel, M.: Multiplierless Multiple Constant Multiplication, ACM Transactions on Algorithms, Vol. 3, No. 2, 2007