

SIMD Vectorization of Straight Line FFT Code

Stefan Kral, Franz Franchetti, Juergen Lorenz, and Christoph W. Ueberhuber

Institute for Applied Mathematics and Numerical Analysis,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
skral@complang.tuwien.ac.at,
WWW home page: <http://www.math.tuwien.ac.at/ascot>

Abstract. This paper presents compiler technology that targets general purpose microprocessors augmented with SIMD execution units for exploiting data level parallelism. FFT kernels are accelerated by automatically vectorizing blocks of straight line code for processors featuring two-way short vector SIMD extensions like AMD's 3DNow! and Intel's SSE 2. Additionally, a special compiler backend is introduced which is able to (i) utilize particular code properties, (ii) generate optimized address computation, and (iii) apply specialized register allocation and instruction scheduling.

Experiments show that automatic SIMD vectorization can achieve performance that is comparable to the optimal hand-generated code for FFT kernels. The newly developed methods have been integrated into the codelet generator of FFTW and successfully vectorized complicated code like real-to-halfcomplex non-power-of-two FFT kernels. The floating-point performance of FFTW's scalar version has been more than doubled, resulting in the fastest FFT implementation to date.

1 Introduction

Major vendors of general purpose microprocessors have included short vector single instruction multiple data (SIMD) extensions into their instruction set architecture (ISA) to improve the performance of multimedia applications by exploiting data level parallelism. The newly introduced instructions have the potential for outstanding speed-up, but they are difficult to utilize using standard algorithms and general purpose compilers.

Recently, a new software paradigm emerged in which optimized code for numerical computation is generated automatically [5, 6]. For example, FFTW has become the de facto standard for high performance FFT computation. However, the current version of FFTW does not utilize short vector SIMD extensions.

This paper presents compiler technology for automatically vectorizing numerical computation blocks as generated by automatic performance tuning systems like FFTW, SPIRAL, and ATLAS. The blocks to be vectorized may contain load and store operations, index computation, as well as arithmetic operations.

Moreover, the paper introduces a special compiler backend which generates assembly code optimized for short vector SIMD hardware. This backend is able

to utilize special features of straight line code, generates optimized address computation, and applies additional performance boosting optimization.

The newly developed methods have been integrated into FFTW's codelet generator yielding FFTW-GEL, a short vector SIMD version of FFTW featuring an outstanding floating-point performance (see Figure 1).

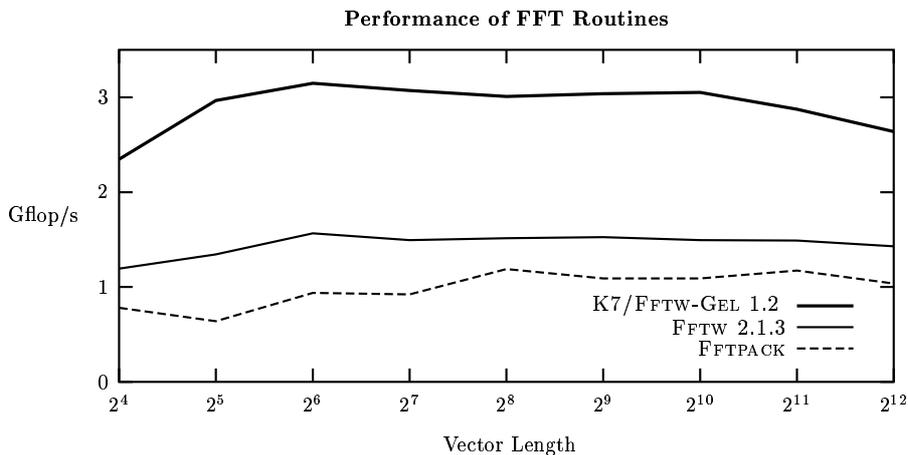


Fig. 1. Floating-point performance of the newly developed K7/FFTW-GEL (3DNow!) compared to FFTPACK and FFTW 2.1.3 on a 1.53 GHz AMD Athlon XP 1800+ carrying out complex-to-complex FFTs in single-precision.

Related Work. Established vectorization techniques mainly focus on loop-constructs satisfying certain constraints. For instance, Intel's C++ compiler and Codeplay's VECTOR C compiler are able to vectorize loop code for both integer and floating-point short vector extensions provided that arrays are accessed in a very specific way. The upgrade [8] to the SUIF compiler vectorizes loop code for the integer short vector extension MMX.

A SPIRAL based approach to portably vectorize discrete linear transforms utilizing structural knowledge is presented in [3].

Intel's math kernel library (MKL) and performance primitives (IPP) both support SSE and SSE 2 available on Pentium III and 4 processors and the Itanium processor family.

Synopsis. Section 2 describes FFTW-GEL, the new short vector SIMD version of FFTW. Section 3 describes the machine independent vectorization of basic blocks and related optimization techniques. Section 4 introduces the newly developed backend optimization techniques. The methods of Sections 3 and 4 are the core techniques used within the codelet generation of FFTW-GEL. Section 5 presents numerical results for the 3DNow! and the SSE 2 version of FFTW-GEL applied to complex-to-complex and real-to-halfcomplex FFTs.

2 FFTW for Short Vector Extensions: FFTW-GEL

FFTW-GEL¹ is an extended version of FFTW that supports two-way short vector SIMD extensions (see Figure 2). In FFTW-GEL the scalar computing cores of FFTW have been replaced by their short vector SIMD counterparts. Thus, the overall structure and all features of FFTW remain unchanged, only the computational speed is increased significantly. Details can be found in [2].

FFTW. The first effort to automatically generate and optimize FFT code using actual run times as an optimization criterion resulted in FFTW [4]. In many cases FFTW runs faster than other publicly available FFT codes like the industry standard FFTPACK².

FFTW is based on a recursive implementation of the Cooley-Tukey FFT algorithm and uses dynamic programming with measured run times as cost function to find a fast implementation for a given problem size on a given machine. FFTW consists of four fundamental parts: (i) the planner, (ii) the executor, (iii) the codelet generator `genfft`, and (iv) codelets.

Planner and executor provide for the adaptation of the FFT computation to the target machine at runtime while the actual computation is done within routines called codelets. Codelets are generated by the codelet generator `genfft`, a special purpose compiler.

Within the codelets a variety of FFT algorithms [9] is used, including the Cooley-Tukey algorithm, the split radix algorithm, the prime factor algorithm, and the Rader algorithm.

The methods presented in this paper were used to extend FFTW's codelet generator to produce vectorized codelets. Due to the structure of FFT algorithms, codelets feature a certain degree of parallelism allowing for the application of the new vectorization technique but preventing the application of standard methods.

These vectorized codelets are compatible with FFTW's framework and thus can be used instead of the original codelets on machines featuring two-way short vector SIMD extensions. That way FFTW's core routines are sped up while all features of standard FFTW remain supported.

Short Vector SIMD Extensions. Examples of two-way short vector SIMD extensions supporting both integer and floating-point operations include Intel's streaming SIMD extensions SSE 2 and AMD's 3DNow! family.

Double-precision short vector SIMD extensions paved the way to high performance scientific computing. However, special code is needed as conventional scalar code running on machines featuring these extensions utilizes only a small fraction of the potential performance.

Short vector SIMD extensions are advanced architectural features which are not easily utilized for producing high performance codes. Currently, two approaches are commonly used to utilize SIMD instructions.

¹ available from <http://www.fftw.org/~skral>

² available from <http://www.netlib.org/fftpack>

Vectorizing Compilers. The application of compiler vectorization is restricted to loop-level parallelism and requires special loop structures and data alignment. Whenever the compiler cannot prove that a loop can be vectorized optimally using short vector SIMD instructions, it has to resort to either emitting scalar code or to introducing additional code to handle special cases.

Hand-Coding. Compiler vendors provide extensions to the C language (data types and intrinsic or built-in interfaces) to enable the direct use of short vector SIMD extensions from within C programs. Of course, assembly level hand-coding is unavoidable if there is no compiler support.

FFTW-GEL. Both of the latter approaches are not directly utilizable to generate vectorized FFTW codelets that may act as a replacement for the respective standard codelets. Due to its internal structure, FFTW cannot be vectorized using compilers focusing on loop vectorization: (i) arrays are accessed at strides that are determined at runtime and may be not unit stride, preventing loop level vectorization, and (ii) a large class of codelets does not even contain loops. FFTW’s automatically generates code featuring large basic blocks (up to thousands of lines) of numerical straight line code. Accordingly, such basic blocks consisting of straight line code have to be vectorized rather than loops.

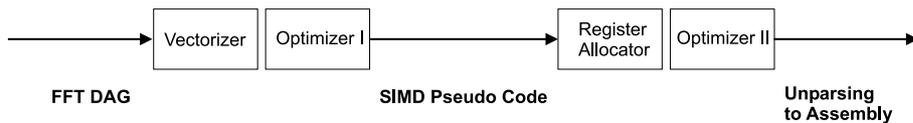


Fig. 2. The basic structure of FFTW-GEL. FFT DAGs generated by the codelet generator `genfft` are vectorized and subjected to advanced optimization.

Vectorization of Straight Line Code. Within the context of FFTW, codelets do the bulk of the computation. As codelets feature a function body consisting sometimes of thousands of lines of automatically generated straight line code without loops, the vectorization of straight line code is an indispensable requirement in this context. The challenge addressed by vectorizing straight line code is to extract parallelism out of this sequence of operations while maintaining data locality and utilizing special features of the target architecture’s short vector SIMD extensions.

Straight Line Code Assembly Backend. The straight line code of FFTW’s codelets features several characteristics that can be exploited by special backend optimization: (i) All input array elements of a codelet are loaded into a temporary variable exactly once, (ii) each output array element is stored exactly once, and (iii) all index computations are linear (base address + index × stride).

The main goal in optimized index computation is to avoid expensive integer multiplication by computing the required multiples of the stride by clever utilization of integer register contents and implicit operations supported by the

target instruction set (e. g., load/store instructions involving some effective address calculation or the “load effective address” instruction in the x86 ISA). The optimal cache utilization algorithm is used for register allocation. Furthermore, straight line code specific optimization techniques are applied.

3 Two-way SIMD Vectorization of Straight Line Code

An important core technology used in FFTW-GEL is the two-way SIMD vectorization of numerical straight line code. This process extracts vector instructions out of the stream of scalar floating-point operations by joining scalar operations together. These artificially assembled vector instructions allow to completely vectorize rather complex codes. However, some of these instructions may not be contained in the target architecture’s instruction set and thus have to be emulated by a sequence of available vector instructions.

The vectorization engine performs a non-deterministic matching against a directed acyclic graph (DAG) of scalar floating-point instructions to automatically extract two-way SIMD parallelism out of a basic block.

Additional optimization stages are required to translate SIMD pseudo-instructions into target instructions and to produce a topologically sorted sequence of SIMD instructions. This sequence is subsequently fed into the back-end described in the following section for further optimization and unparsing to assembly language.

3.1 The Vectorization Engine

The goal of the vectorization process is to replace all scalar floating-point instructions by SIMD instructions. To achieve this goal, pairings of scalar floating-point instructions (and their respective operands) are to be found yielding in each case one SIMD floating-point instruction. In some cases, additional SIMD instructions like swaps may be required. A given pair of scalar floating-point instructions can only be fused if it can be implemented by the available SIMD instructions without changing the semantics of the DAG.

The vectorization algorithm works as follows. (1) Pick two scalar instructions with minimal depth whose output variables are already paired. (2) Combine these and pair their input variables. A set of rules defines feasible pairs of scalar instructions and the resulting pairings of scalar variables for each alternative SIMD implementation. It has to be asserted globally that every scalar variable occurs in at most one pairing. Alternatives are saved on a stack.

Initially no scalar instructions are paired. Steps (1) and (2) are iterated until either all scalar instructions are vectorized, or the search process fails. Upon failure, the last alternative saved on the stack is chosen, resuming the search at that point. This implements a depth-first search with chronological backtracking.

The pairing ruleset is chosen to enable a large enough number of alternative SIMD implementations for pairs of scalar operations to enable vectorization while keeping the search space reasonably small.

The benefits of the vectorization process are the following: *(i)* Scalar floating-point instructions are replaced by SIMD instructions. Thus, the instruction count is nearly halved. *(ii)* The fusion of memory access operations potentially reduces the effort for calculating effective addresses. *(iii)* The wider SIMD register file allows for more efficient computation.

3.2 Vectorization Specific Optimization

After finishing the vectorization stage, further optimization is performed in a separate stage. This procedure keeps the implementation of the vectorizer simple and saves time needed for the vectorization process by postponing additional optimization until the backtracking search has succeeded.

The pseudo SIMD instructions generated by the vectorization engine are directly fed into the optimization stage. A rewriting system is used to simplify groups of target-architecture independent instructions as well as target-architecture specific combinations of instructions.

A first group of rewriting rules is applied to *(i)* minimize the number of instructions, *(ii)* eliminate redundancies, *(iii)* reduce the number of source operands, *(iv)* eliminate dead code, and *(v)* perform constant folding. Some rules shorten the critical path lengths of the data dependency graphs by exploiting specific properties of the target instruction set. Finally, some rules reduce the number of source operands necessary to perform an operation, thus effectively reducing register pressure.

A second group of rules is used to rewrite pseudo instructions into combinations of instructions actually supported by the target architecture.

In a third and last step the optimization process topologically sorts the DAG that will be translated subsequently. The respective scheduling algorithm was designed to minimize the lifetime of variables in the scheduled code by improving the locality of variable accesses. The scheduling process of FFTW-GEL extends the scheduler of FFTW 2.1.3 by additional postprocessing to further improve locality of variable accesses in some cases.

4 Backend Optimization for Straight Line Code

The backend translates a scheduled sequence of SIMD instructions into assembly language. It performs optimization in both an instruction set specific and a processor specific way. In particular, register allocation and the computation of effective addresses is optimized with respect to the target instruction set. Instruction scheduling and avoidance of address generation interlocks is done specifically for the target processor.

4.1 Instruction Set Specific Optimization

Instruction set specific optimization takes into account properties of the target microprocessor's instruction set.

Depending on whether a RISC or a CISC processor is used, additional constraints concerning source and target registers have to be satisfied. For instance, many CISC style instruction sets (like Intel's x86) require that one source register must be used as a destination register.

The utilization of memory operands, as provided by many CISC instruction sets, results in locally reduced register pressure. It decreases the number of instructions as two separate instructions (one load- and one use-instruction) can be merged into one (load-and-use). The code generator of FFTW-GEL uses this kind of instructions whenever some register content is used only once.

Many machines include complex instructions that are combinations of several simple instructions like a "shift by some constant and add" instruction. The quality of the code that calculates effective addresses can often be improved by utilizing this kind of instructions.

Register Allocation. FFTW codelets are (potentially very large) sequences of straight line C code. Knowing the complete code structure and data dependencies, it is possible to perform register allocation according to Belady's optimal caching algorithm [1].

Experiments have shown that this policy is superior to the ones used in mainstream C compilers (like the GNU C compiler, Compaq's C compiler for Alpha, and Intel's C compiler) for this particular type of code.

Efficient Calculation of Effective Addresses. FFTW codelets operate on arrays of input and output data. Both input and output arrays may not be stored with unit stride in memory. Thus access to element `in[i]` may result in a memory access at address `in + i*sizeof(in)*stride`. Both `in` and `stride` are parameters passed from the calling function. For a codelet of size N , all elements `in[i]` with $0 \leq i < N$ are accessed exactly once. As a consequence, the quality of code dealing with variable array strides is crucial for achieving high performance. In particular, all integer instructions apart from parameter passing occurring in FFTW codelets are used for calculating effective addresses.

To achieve an efficient computation of effective addresses, FFTW-GEL follows the following guidelines: (i) General multiplications are avoided. While it is tempting to use an integer multiplication instruction (`imul`) for general integer multiplication in effective address calculation, it is usually better to use equivalent sequences of simpler instructions (addition, subtraction or shift operations) instead. Typically, these instructions can be decoded faster, have shorter latencies, and are all fully pipelined. (ii) The load effective address instruction (`lea`) is a cheap alternative to integer multiplication on x86 compatible processors. It is used for combining up to two adds and one shift operation into a single instruction. (iii) Only integer multiplications are rewritten whose result can be obtained using less than n other operations, where n depends on the latency of the integer multiplication instruction on the target architecture.

In FFTW-GEL, the actual generation of code sequences for calculating effective addresses has been intertwined with the register allocation process. Whenever code for the calculation of an effective address is to be generated, the (locally) shortest possible sequence of instructions is chosen (determined by depth-

first iterative deepening), trying to reuse effective addresses that have already been calculated and that are still stored in some integer register.

4.2 Processor Specific Optimization

Typically, processor families supporting one and the same instruction set have different instruction latencies and throughput. Processor specific optimization has to take into account the execution properties of a *specific* processor. The code generator of FFTW-GEL uses two processor specific optimizations: (i) Instruction scheduling, and (ii) avoidance of address generation interlocks.

Instruction Scheduling. The instruction scheduler of FFTW-GEL tries to find an ordering of instructions optimized for the target processor. During the scheduling process it uses information about critical-path lengths of the underlying data dependency graph in a heuristic way. The basic principles of the instruction scheduler are illustrated in [7].

The instruction scheduler is extended by a relatively simple processor model that is used for estimating whether a sequence of instructions can be issued within the same cycle and executed in parallel by means of super-scalar execution. By using this prediction, the scheduler tries to resolve tie situations arising in the scheduling process.

In addition, the instruction scheduler serves as runtime estimator and is used for controlling optimizations that could either increase or decrease performance.

Avoiding Address Generation Interlocks. Although many modern super-scalar microprocessors allow an out-of-order execution, most architecture definitions require that memory accessing instructions must be executed in-order.

An address generation interlock (AGI) occurs whenever a memory operation that involves some expensive effective address calculation directly precedes a memory operation that involves inexpensive address calculation or no address calculation at all. Then the second access is stalled by the computation of the first access' effective address.

The backend tries to avoid AGIs by reordering instructions immediately *after* instruction scheduling. This is done in a separate stage to simplify the design of the instruction scheduler.

5 Experimental Results

Numerical experiments were carried out to demonstrate the applicability and the performance boosting effects of the newly developed techniques.

FFTW-GEL was investigated on two IA-32 compatible machines: (i) An Intel Pentium 4 running at 1.8 GHz and featuring two-way double-precision SIMD extensions (SSE 2), as well as (ii) an AMD Athlon XP 1800+ running at 1.53 GHz and featuring two-way single-precision SIMD extensions (3DNow! professional).

Performance data are displayed in pseudo-flop/s, i. e., $5N \log N/T$ for complex-to-complex and $2.5N \log N/T$ for real-to-halfcomplex FFTs. This unit of

measurement is a scaled inverse of the run time T and thus preserves run time relations and gives an indication of the absolute floating-point performance [5].

The speed-up achievable by vectorization (ignoring other effects like smaller code size, wider register files, etc.) is limited by a factor of two on both test machines. However, the additional backend optimization further improves the performance of the generated codes. All codes were generated using the GNU C compiler 2.95.2 and the GNU assembler 2.9.5. Both complex-to-complex FFTs for power-of-two problem sizes and real-to-halfcomplex FFTs for non-powers of two were evaluated. Experiments show that neither FFTW's executor nor the codelets are vectorized by Intel's C++ compiler. Thus, the newly developed vectorizer of FFTW-GEL is more advantageous than traditional compiler vectorization by a big margin.

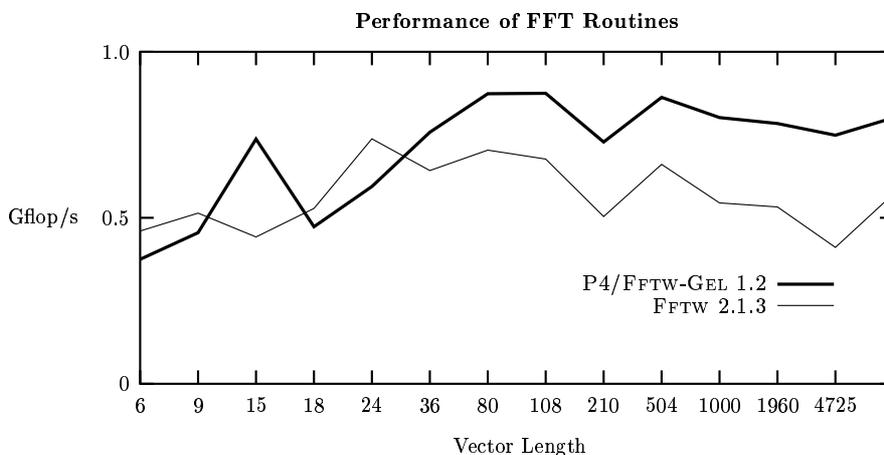


Fig. 3. Floating-point performance of the newly developed P4/FFTW-GEL (SSE 2) compared to FFTW (double-precision) on an Intel Pentium 4 running at 1.8 GHz for real-to-halfcomplex FFTs.

AMD 3DNow! Figure 1 (on page 2) shows the performance of FFTPACK, FFTW 2.1.3, and K7/FFTW-GEL on the Athlon XP in single-precision. The runtimes displayed refer to powers of two complex-to-complex FFTs whose data sets fit into L2 cache. K7/FFTW-GEL utilizes enhanced 3DNow! which provides two-way single-precision SIMD extensions. FFTW is up to two times faster than FFTPACK, the industry standard in non-hardware-adaptive FFT software.

FFTW-GEL is about twice as fast as FFTW which demonstrates that the performance boosting effect of vectorization and backend optimization is outstanding.

Intel SSE 2. Figure 3 shows the performance of FFTW 2.1.3 and P4/FFTW-GEL on the Pentium 4 in double-precision arithmetic. Runtimes were measured

for non-powers of two with real-to-halfcomplex FFTs whose data sets fit into L2 cache. P4/FFTW-GEL utilizes the two-way double-precision SIMD operations of SSE 2. For most problem sizes FFTW-GEL is faster than FFTW 2.1.3 (up to 60%). Note that real-to-halfcomplex FFTs are notoriously hard to vectorize (especially for non-powers of two) due to the much more complicated algorithmic structure compared to complex-to-complex FFT algorithms.

Conclusion

This paper presents a set of compilation techniques for automatically vectorizing numerical straight line code. As straight line code is in the center of all current numerical performance tuning software, the newly developed techniques are of particular importance in scientific computing.

Impressive performance results demonstrate the usefulness of the newly developed techniques which can even vectorize the complicated code of real-to-halfcomplex FFTs for non-powers of two.

Acknowledgement. We would like to thank Matteo Frigo and Steven Johnson for many years of fruitful cooperation and for making it possible for us to access non-public versions of FFTW.

Additionally, we would like to acknowledge the financial support of the Austrian science fund FWF.

References

1. Belady, L.A.: A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal* **5** (1966), 78–101
2. Franchetti F., Kral, S., Lorenz, J., Ueberhuber, C.: *AURORA Report Series on SIMD Vectorization Techniques for Straight Line Code*. Technical Report AURORA TR2003-01, TR2003-10, TR2003-11, TR2003-12, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology (2003)
3. Franchetti, F., Püschel, M.: A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms In: *Proc. International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Comp. Society Press, Los Alamitos (2002), 20–26
4. Frigo, M.: A Fast Fourier Transform Compiler. In: *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. ACM Press, New York (1999), 169–180
5. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: *ICASSP 98*. Volume 3. (1998) 1381–1384
6. Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V., Püschel, M., Veloso, M.M.: *SPIRAL: Portable Library of Optimized Signal Processing Algorithms* (1998)
7. Muchnick, S.S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco (1997)
8. Sreraman, N., Govindarajan, R.: A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming* **28** (2000) 363–400
9. Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia (1992)