FFT Compiler Techniques

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

Institute for Analysis and Scientific Computing,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
skral@complang.tuwien.ac.at,
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. Numerical applications are accelerated by automatically vectorizing blocks of straight line code to be run on processors featuring two-way short vector SIMD extensions like Intel's SSE 2 on Pentium 4, SSE 3 on Intel Prescott, AMD's 3DNow! , and IBM's SIMD operations implemented on the new processors of the BlueGene/L supercomputer.
The paper introduces a special compiler backend for Intel P4's SSE 2 and AMD's 3DNow! which is able (*i*) to exploit particular properties of FFT code, (*ii*) to generate optimized address computation, and (*iii*) to perform specialized register allocation and instruction scheduling.
Experiments show that the automatic SIMD vectorization techniques of this paper enable performance of hand optimized code for key benchmarks. 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 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 [4,9]. For example, FFTW has become the de facto standard for high performance FFT computation. The current version of FFTW includes the techniques presented in this paper to utilize SIMD extensions.

E. Duesterwald (Ed.): CC 2004, LNCS 2985, pp. 217–231, 2004.

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.

In particular, the paper introduces a special compiler backend which generates assembly code optimized for short vector SIMD hardware. This backend is able to exploit special features of straight-line FFT 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 [6], a short vector SIMD version of FFTW featuring outstanding floating-point performance (see Section 5).

Related Work. Established vectorization techniques mainly focus on loop-constructs. 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.

An upgrading to the SUIF compiler that vectorizes loop code for MMX is described in [11].

A SPIRAL based approach to portably vectorize discrete linear transforms utilizing structural knowledge is presented in [2].

A vectorizing compiler exploiting *superword level parallelism* (i. e., SIMD style parallelism) has been introduced in [8]. IBM's XL C compiler for BlueGene/L [13] utilizes this vectorization technique.

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, as well as the new SSE 3 on Intel Prescott.

A backend for straight-line code targeting MIPS processors is presented in [5].

2 FFTW for Short Vector Extensions: FFTW-GEL

FFTW-GEL [1] is an extended version of FFTW that supports two-way short vector SIMD extensions (see Fig. 1).

FFTW is an automatic FFT code generator based on a recursive implementation of the Cooley-Tukey FFT algorithm. FFTW 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 the following parts: *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 is used, including the Cooley-Tukey algorithm, the split radix algorithm, the prime factor algorithm, and the Rader algorithm.

[1] available from `http://www.fftw.org/~skral`

The compiler techniques presented in this paper were used to extend FFTW's codelet generator to produce vectorized codelets. 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 SSE 3, AMD's 3DNow! family, as well as IBM's PowerPC 440d processors in BlueGene/L supercomputers.

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.

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.

2.1 FFTW-GEL

Both vectorizing compilers and hand-coding are not directly applicable to generate vectorized FFTW codelets to replace the standard codelets. Due to its internal structure, FFTW cannot be vectorized using compilers focusing on loop vectorization. FFTW automatically generates code consisting of up to thousands of lines of numerical straight-line code. Accordingly, such basic blocks have to be vectorized rather than loops.

Besides, large codelets are favored over smaller codelets used in loop bodies in the creation of FFTW plans as they feature less reorder operations. Therefore, the automatic vectorization of large basic blocks is an important task. Vectorizing them by hand is generally unfeasible as the instruction counts in Fig. 2 illustrate.

FFTW-GEL comprises an extended, architecture-specific version of FFTW's `genfft`, supporting two-way short vector SIMD extensions. Moreover, it includes a special compiler backend for x86 architectures which generates assembly code optimized for short vector SIMD hardware. This backend is able to exploit special features of straight-line FFT code, generates optimized address computation, and applies additional performance boosting optimization.

FFTW-GEL's Architecture. Fig. 1 illustrates the various levels of optimization performed by FFTW-GEL.

Vectorization of Scalar Straight-Line Code. Within the context of FFTW, codelets do the bulk of the computation. The challenge addressed by the vectorizer is to extract parallelism out of of this sequences of scalar instructions while maintaining data locality and utilizing special features of 2-way short vector SIMD extensions.

Optimization of Vectorized Code. Optimizer I comprises a local rewriting system using a set of rules to optimize the DAG of SIMD instructions obtained from the vectorizer.

Assembly Backend for Straight-Line Code. The backend uses (*i*) efficient methods to compute effective addresses, (*ii*) a register allocator, (*iii*) an instruction scheduler, (*iv*) an optimizer for in-memory operands, (*v*) a register reallocator, and (*vi*) address generation interlock (AGI) prevention for code size reduction.

The register reallocator, the direct use of in-memory operands, and other optimization techniques and methods, are executed in a feedback driven optimization loop to further improve the optimization effect.

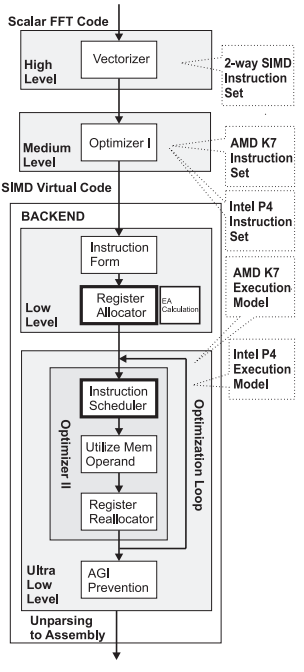


Fig. 1. FFTW-GEL's Architecture. Automatically generated FFT DAGs are vectorized and optimized.

3 Vectorization of Scalar Straight-Line Code

This section introduces FFTW-GEL [7], a vectorizer that automatically extracts *2-way SIMD parallelism* out of a sequence of operations from static single assignment (SSA) straight-line code while maintaining data locality and utilizing special features of short vector SIMD extensions.

FFTW-GEL produces vectorized code either (*i*) via a source-to-source transformation delivering macros compliant with the portable SIMD API [14] and additionally providing support for FMA instructions, or (*ii*) via a source-to-assembly transformation utilizing the FFTW-GEL backend of Section 5.

3.1 The Vectorization Approach

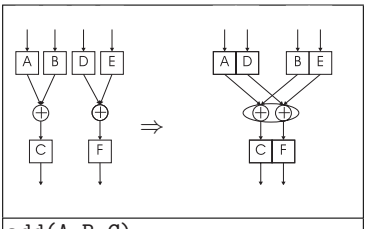
The vectorizer aims at vectorizing straight-line code containing arithmetic operations, array access operations and index computations. Such codes cannot be handled by established vectorizing compilers mainly focussing on loop vectorization [15]. Even standard methods for vectorizing straight-line code [8] fail in

producing high performance vector code as well. FFTW-GEL's vectorizer uses a backtracking search engine to automatically extract *2-way SIMD parallelism* out of scalar code blocks by fusing pairs of scalar temporary variables into SIMD variables, and by replacing the corresponding scalar instructions by vector instructions as illustrated by Table 1.

Tuples of scalar variables, i. e., *fusions*, are declared such that every scalar variable appears in exactly one of them. Each fusion is assigned one SIMD variable.

On the set of SIMD variables, SIMD instructions have to perform exactly the same computation as the originally given scalar instructions do on the scalar variables. This is achieved by rules specifying how each pair of scalar instructions is replaced by a sequence of semantically identical SIMD instructions operating on the corresponding SIMD variables.

Table 1. 2-way Vectorization. Two scalar `add` instructions are transformed into a vector `vadd` instruction. The result of the vector addition of the fusions `AD` and `BE` is stored in `CF`.




```
add(A,B,C)
add(D,E,F)  ⇒  vadd(AD,BE,CF)
```

Rules for pairs of scalar binary instructions allow up to three different, semantically equivalent ways to do so. Locally, this allows vectorizations of different efficiency. Globally, it widens the search space of the vectorizer's backtracking search engine.

The vectorizer's goal is to yield an optimal utilization of SIMD resources. This target and the fact that the backtracking engine uses a *pick first result* strategy to minimize vectorization runtime does not necessarily lead to the best possible vectorization in terms of an overall minimum of SIMD instructions but allows the vectorization of large FFTW codelets (see Table 2).
FFTW-GEL's vectorizer utilizes an automatic fallback to different vectorization levels in its vectorization process. Each level provides a set of pairing rules with different pairing restrictions. Thus, more restrictive rules are just applied to highly parallel codes resulting either in efficient vectorization or no vectorization at all. Less restrictive rules address a broader range of codes but result in less efficient vectorization.

This concept assists the searching for good vectorizations needing a minimum of data reorganization, provided such vectorizations exist. The fallback to a less restrictive vectorization level allows to vectorize even codes featuring less parallel parts to some extent.

Virtual Machine Models. To keep the amount of hardware specific details needed in the vectorization as small as possible, *virtual machine models* are introduced.

The virtual machine models used in the vectorizer are abstractions of scalar as well as 2-way SIMD architectures, i. e., the vectorizer transforms virtual scalar to virtual 2-way SIMD instructions. These are rewritten into 2-way SIMD in-

structions actually available in a specific architecture's instruction set in an optimization process, directly following the vectorization.

3.2 The Vectorization Engine

The input of the vectorization engine is a scalar DAG representing straight-line code of virtual scalar instructions in SSA form. In the vectorization engine all virtual scalar instructions are replaced by virtual SIMD instructions.

The vectorization algorithm described in the next section utilizes the infrastructure provided by FFTW-GEL's vectorization engine. To describe the concepts of the vectorization engine, some definitions and explanations are provided.

Pairing. *Pairing rules* specify ways of transforming pairs of scalar instructions into a single SIMD instruction or a sequence of semantically equivalent SIMD instructions. A pairing rule often provides several alternatives to do so. The rules used in the vectorizer are classified according to the type of the scalar instruction pair: unary (i. e., multiplication by a constant), binary (i. e., addition and subtraction), and memory access type (i. e., load and store instructions). Pairings of the following instruction combinations are supported: (*i*) consecutive load/load, (*ii*) arbitrary load/load, (*iii*) consecutive store/store, (*iv*) arbitrary store/store, (*v*) unary/unary, (*vi*) binary/binary, (*vii*) unary/binary, (*viii*) unary/load, and (*ix*) load/binary. Not all of the above pairing combination allow for an optimal utilization of SIMD resources.

A *pairing ruleset* comprises various pairing rules. At the moment, the pairing ruleset does not comprise a rule for vectorizing an odd number of scalar store instructions.

Two scalar instructions are vectorized, i. e., *paired*, if and only if neither of them is already paired and the instruction types are matching a pairing rule from the utilized pairing rule set.

Fusion. Two scalar operands `S` and `T` are assigned together, i. e., *fused*, to form a SIMD variable of layout `ST = (S,T)` or `TS = (T,S)` if and only if they are the corresponding operands of instructions considered for pairing and neither of them is already involved in another fusion. The position of the scalar variables `S` and `T` inside a SIMD variable (either as its lower or its higher part) strictly defines the fusion, i. e., `ST` $\neq$ `TS`. A special fusion, i. e., `TT = (T,T)`, is needed for some types of code partly containing non-parallel program flow.

An instruction pairing rule forces the fusion layout for the corresponding scalar operands. For two scalar binary instructions, three substantially different layouts for assigning the four operands to two SIMD variables exist, namely (*i*) accumulate (*ii*) parallel 1, and (*iii*) parallel 2.

A fusion `X12 = (X1,X2)` is *compatible* to another fusion `Y12 = (Y1,Y2)` if and only if `X12 = Y12` or `X1 = Y2` and `X2 = Y1`. In the second case, a swap operation is required as and additional "special transformation" to use `Y12` whenever `X12` is needed. The number of special transformations is minimized in a separate optimization step to minimize the runtime of the vectorization process. Section 4 describes this along other optimization techniques in more detail.

Vectorization Levels. A vectorization level embodies a subset of rules from the overall set of pairing rules. Different subsets belonging to different levels comprise pairing rules of different versatility. Generally, more versatile rule sets lead to less efficient vectorization but allow to operate on codes featuring less parallelism. In contrast, less versatile rule sets are more restrictive in their application and are thus only usable for highly parallel codes. They assure that if a good solution exists at all, it is found resulting in highly performant code.

The vectorization engine utilizes three levels of vectorization in its search process as follows: First, a vectorization is sought that utilizes the most restrictive level, i. e., *full vectorization*. This vectorization level only provides pairing rules for instructions of the same type and allows quadword memory access operations exclusively. If full vectorization is not obtainable, a fallback to a less restrictive vectorization level, i. e., *semi vectorization*, is made. This level provides versatile pairing rules for instructions of mixed type and allows doubleword memory access operations. In the worst case, if neither semi vectorization nor full vectorization is feasible, a fallback is made to a vector implementation of scalar code, i. e., *null vectorization* is applied. Null vectorization rules allow to vectorize any code by leaving half of each SIMD instruction's capacity unused. Even null vectorization results in better performance than using legacy x87 code. Besides, null vectorization is used as default, without trying full and semi vectorization, when it is known in advance that vectorization is impossible because, e. g., of an odd number of scalar store instructions, etc.

3.3 The Vectorization Algorithm

FFTW-GEL's vectorization algorithm implements a depth first search with chronological backtracking. The search space is given by appying rules given by the current vectorization level in an arbitrary order. Depending on how versatile/restrictive the utilized pairing rule set is, there can be many, one or no possible solution at all.

The vectorization algorithm performs the following steps:

Step 1: Initially, no scalar variables are fused and no scalar instructions are paired. The process starts by pairing two arbitrary store instructions and fusing the corresponding source operands. Should the algorithm backtrack without success, it tries possible pairings of store instructions, one after the other.

Step 2: Pick an existing fusion on the vectorization path currently being processed, whose two writing instructions have not yet been paired. As the scalar code is assumed to be in SSA form, there is exactly one instruction that uses each of the fusion's scalar variables as its destination operand. According to the vectorization level and the type of these instructions, an applicable pairing rule is chosen. If all existing fusions have already been processed, i. e., the dependency path has been successfully vectorized from the stores to all affected loads, start the vectorization of another dependency path by choosing two remaining stores. If all stores have been paired and no fusions are left to be processed, a solution has been found and the algorithm terminates.

Step 3: According to the chosen pairing rule, fuse the source operands of the scalar instructions if possible (i. e., none of them is already part of another fusion) or, if a compatible fusion exists use it instead.

Step 4: Pair the chosen instructions, i. e., substitute them by one or more according SIMD instructions.

Step 5: If a fusion or pairing alternative does not lead to a valid vectorization, choose another pairing rule. If none of the applicable rules leads to a solution, fail and backtrack to the most recent vectorization step.

Steps 2 to 5 are iterated until all scalar instructions are vectorized, possibly requiring new initialization carried out by Step 1 during the search process. If the search process terminates without having found a result, a fallback to the next more general vectorization level is tried, leading to null vectorization in the worst case.

If a given rule set is capable of delivering more than one valid solution, the order in which the pairing rules are tested is relevant for the result. This is used to favor specific kinds of instruction sequences by ranking the corresponding rules before the others. For instance, the vectorization engine is forced first to look for instructions implementing operation semantics directly supported by a given architecture, thus minimizing the number of extracted virtual instructions that have to be rewritten in the optimization step.

Table 2 shows runtimes of the vectorization algorithm for some representative FFTW codelets. Even large codelet, e. g., $n = 256$, can be vectorized in a few seconds.

Table 2. Vectorization Runtimes. The table shows instruction counts as well as vectorization runtimes in seconds for various twiddle complex-to-complex, no-twiddle complex-to-complex, and real-to-halfcomplex FFTW codelets. The vectorization algorithm has been implemented in Objective Caml. The runtimes have been determined using Objective Caml v3.06 with native-code compilation on an 800 MHz AMD K7 processor.

Code Name	*Loads/Stores*	*Adds+Subs*	*Muls*	*Vectorization Runtime*
`frc_32`	32/32	156	42	<.1s
`frc_30`	30/30	158	56	5.6s
`ftw_17`	66/34	328	180	<.1s
`fn_128`	256/256	2164	660	0.7s
`fn_256`	512/512	5008	1656	2.2s

4 Optimization of Vectorized Code

After the vectorizer terminates successfully by delivering a vectorization of the scalar DAG, Optimizer I performs several simple improvements to the resulting

code. For that purpose, a rewriting system is used to simplify combinations of virtual SIMD instructions, also with regard to target architecture specifications.

The first group of rewriting rules that is applied aims at (*i*) minimizing the number of instructions, (*ii*) eliminating redundancies and dead code, (*iii*) reducing the number of source operands (copy propagation), and (*iv*) performing constant folding. The critical path length of the DAG is shortened by exploiting specific properties of the target instruction set. Finally, the number of source operands necessary to perform an operation is reduced, thus minimizing register pressure.

The second group of rules is used to rewrite virtual instructions into combinations of (still virtual) instructions actually supported by the target architecture.

In a third and last step the Optimizer I schedules and topologically sorts the instructions of the vectorized DAG. The scheduling algorithm minimizes the lifespan of variables by improving the locality of variable accesses. It is an extension of FFTW 2.1.3's scheduler.

The code output by Optimizer I consists of instructions out of a subset of the virtual SIMD instruction set that corresponds to the target architecture.

5 Assembly Backend for Straight-Line Code

The FFTW-GEL backend [7] introduced in this section generates assembly code optimized for short vector SIMD hardware. It is able to exploit special features of automatically generated straight-line codes. The backend currently supports assembly code for x86 with 3DNow! or SSE2.

The automatically generated codes to be translated are large blocks of SSA code with the following properties: (*i*) there is no program control flow except the basic block's single point of entry and exit, (*ii*) there is full knowledge of future temporary variable usage, (*iii*) there are indexed memory accesses possibly with a stride as runtime parameter, and (*iv*) loads from and stores to data vector elements are performed only once.

Standard backends fail to compile such blocks of SSA code to performant machine code as they are targeted at a broad range of structurally different hand-written codes lacking the necessary domain specific meta information mentioned above. Thus, standard backends fail in register allocation when too many temporary variables are to be assigned to a small number of registers and they also have trouble with efficient calculation of effective addresses.

5.1 Optimization Techniques Used In FFTW-GEL

The FFTW-GEL backend performs optimization in both an *architecture specific* and a *processor specific* way.

In particular, (*i*) register allocation, (*ii*) computation of effective addresses, (*iii*) usage of in-memory operands, and (*iv*) the register reallocator are optimizations with respect to an architecture's instruction set. (*v*) Instruction scheduling

and (*vi*) AGI prevention is done specifically for the target processor in the last optimization step.

Architecture specific optimization takes into account very general properties of the target processor family. These are (*i*) the number of available physical registers, and (*ii*) instruction set specific properties, e. g., x86 instruction forms, and special x86 instructions, e. g., `lea`.

Processor families, all of whose members support one and the same instruction set, may still have different instruction latencies and throughput. Therefore, processor specific optimization has to take the execution properties of any specific processor into account. These properties are provided by processor-specific execution models specifying (*i*) how many instructions can be issued per clock cycle, (*ii*) the latency of each instruction, (*iii*) the execution resources required by each instruction, and (*iv*) the overall available resources.

5.2 One-Time Optimization

Register allocation, computation of effective addresses, and avoidance of address generation interlocks are optimization techniques performed only once.

Register Allocation for Straight-Line Code. The register allocator's input code contains vector computation as well as its integer address computation accessing input and output arrays in memory. Thus, registers have to be allocated for both register types assuming two different target register files.

The codes are in SSA form and thus only one textual definition of a variable exists. There is no control flow either. As a consequence of these properties, it is possible to evaluate the effective live span of each temporary variable, and thus, a farthest first algorithm [12] can be used as a spilling scheme that tries to find a *spill victim* in the following order: (*i*) take a fresh physical register if available, (*ii*) choose among the dead registers the one which has been dead for the longest time, and finally if no register of any other kind is available, (*iii*) choose the register whose usage is ahead furthermost.

General purpose compilers cannot apply the farthest first algorithm in their spilling schemes as they also address codes possibly containing complex control flow structures not allowing to precisely determine the life span of each temporary variable.

Optimized Index Computation. FFTW codelets operate on arrays of input and output data not necessarily stored contiguously 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.

All integer instructions, except the few ones used for parameter passing, are used for calculating effective addresses. For a codelet of size N, the elements `in[i]` with $i = 0, 1, \ldots, N-1$ are accessed exactly once. Still, the quality of code dealing with address computation is crucial for achieving a satisfactory performance.

Instead of using general integer multiplication instructions (`imull`, etc.), it has been demonstrated to be advantageous to utilize (*i*) equivalent sequences of simpler instructions (`add`, `sub`, `shift`) which can be decoded faster, have shorter latencies, and are all fully pipelined, and (*ii*) instructions with implicit integer computation, like the `lea` instruction. Moreover, (*iii*) reusing the content of integer registers using the integer register file is beneficial to avoid costly integer multiplications as far as possible.

The Load Effective Address Instruction. The generation of code sequences for calculating effective addresses is intertwined with the register allocation process. On x86 compatible hardware architectures, the basic idea to achieve an efficient computation of effective array addresses is to use the powerful `lea` instruction which combines up to two adds and one shift operation in one instruction: `base + index*scale + displacement`. This operation can be used to quickly calculate addresses of array elements and therefore is a cheap alternative to general multiplication instructions.

Whenever the register allocator needs to emit code for the calculation of an effective address, the (locally) shortest possible sequence of `lea` instructions is determined by depth-first iterative deepening (DFID) with a target architecture specific depth limit (e. g., 3 for the AMD K7). As the shortest sequences tend to eagerly reuse already calculated contents of the integer register file, a replacement policy based on the least recently used (LRU) heuristic is employed for integer registers.

The generation of code sequences for calculating effective addresses is intertwined with the register allocation process.

*Example 1 (Efficient Computation of 17*stride).* The `lea` instruction is used to compute `17*stride` from the current entries of the integer register file. In this example all required operands already reside in the register file providing three different ways of computing the result. The entries in the register file and the respective `lea` instructions are displayed below.

Integer Register File			
`eax`	`stride*3`	`ebx`	`stride*5`
`edx`	`stride*1`	`esp`	`stride*4`
`esi`	`stride*9`	`edi`	`stride*-1`

LEA Instructions	Result
`lea ecx, [ebx + 4*eax]`	$(5 + 4 * 3) * stride$
`lea ecx, [edi + 2*esi]`	$(-1 + 2 * 9) * stride$
`lea ecx, [edx + 4*esp]`	$(1 + 4 * 4) * stride$

Avoiding Address Generation Interlocks. Memory access operations do not have WAW dependencies in the straight-line FFT codes of this paper as writing to a specific memory location happens only once. Thus, any order of memory instructions obeys the given program semantics. This enables the FFTW-GEL backend to reorder store instructions to resolve address generation interlocks (AGIs). AGIs occur whenever a memory operation involving some expensive effective address calculation directly precedes a memory operation involving inexpensive address calculation or no address calculation at all. Generally, as there is no knowledge whether there are WAW dependencies between memory accessing

instructions or not in the codes, the second access is stalled by the computation of the first access' effective address. Example 2 illustrates the prevention of an AGI.

Example 2 (Address Generation Interlock). A write operation to `out[17]` is directly preceding a write operation to `out[0]` in the assumed code. As hardware requires an in-order memory access, writing to `out[0]` requires a completed writing to `out[17]` and thus the computation of the effective address of `out[17]`. Both would stall write to `out[0]`. By knowing that there are no WAW dependencies between `out[0]` and `out[17]`, these operations are swapped. The write to `out[0]` is done *concurrently* to computing the effective address of `out[17]` and thus its writing is not stalled.

```
addr = 17 * stride;      addr = 17 * stride;
out[addr] = temp0;   ⟶   out[0] = temp1;
out[0] = temp1;          out[addr] = temp1;
```

AGI prevention is performed as FFTW-GEL's last optimization after the feedback driven optimization loop.

5.3 Feedback Driven Optimization

Instruction scheduling, direct use of in-memory operands, and register reallocation are executed in a feedback driven optimization loop to further improve the optimization effect. The instruction scheduler serves as a basis for estimating the runtime of the entire basic block. As long as the code's estimated execution time can be improved, the feedback driven optimizations are executed.

Basic Block Instruction Scheduling. The instruction scheduler of the FFTW-GEL backend aims at single basic blocks. It uses a standard list scheduling algorithm [10,12]. The processor's execution behavior (latencies, etc.) and resources (issue slots, execution units, ports) are taken into account. These properties are provided by externally specified processor execution models, currently available for Intel's Pentium 4 and AMD's K6 and K7.

Direct Usage Of In-Memory Operands. The FFTW-GEL backend enables the usage of an in-memory operand instead of registers for an instruction's source operand. This is possible as many CISC instruction sets, e. g., x86, allow one source operand of an instruction to reside in memory. It enables a reduction of register pressure and a reduction of the total number of instructions by merging one load and one use instruction into one load-and-use instruction.

FFTW-GEL directly applies in-memory operands if and only if the data is accessed by only one instruction. This prevents register files from not being fully utilized and the superfluous loading of data.

This optimization is performed intertwined with the instruction scheduler and a postprocessing register reallocator.

Register Reallocation. Register allocation and instruction scheduling have conflicting goals. As the register allocator tries to minimize the number of register spills, it prefers introducing a false dependency on a dead logical register

over spilling a live logical register. The instruction scheduler, however, aims at the maximization of the pipeline usage of the processor, by spreading out the dependent parts of code sequences according to the latencies of the respective instructions.

As FFTW-GEL performs register allocation before doing instruction scheduling, it is clear that the false dependencies introduced by the register allocator severely reduce the number of possible choices of the instruction scheduler.

To address this problem, the register reallocator tries to lift some (too restrictive) data dependencies introduced by the register allocator (or by a previous pass of register reallocation), enabling a following pass of the instruction scheduler to do a better job. The register reallocator renames all logical SIMD registers using a least-recently-used (LRU) heuristic.

6 Experimental Results

Numerical experiments were carried out to demonstrate the applicability and the performance boosting effects of the newly developed compiler techniques [7].

FFTW-GEL was investigated on two IA-32 compatible machines: (*i*) An Intel Pentium 4 featuring SSE 2 two-way double-precision SIMD extensions [7], and (*ii*) an AMD Athlon XP featuring 3DNow! professional two-way single-precision SIMD extensions (see Fig. 2).

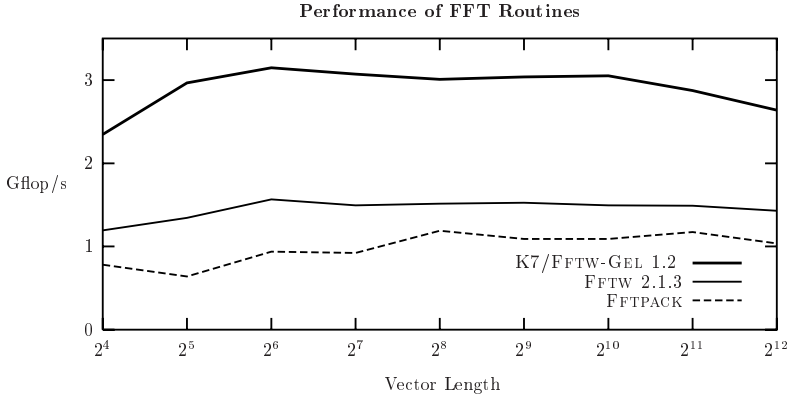


Fig. 2. Floating-point performance of the newly developed K7/FFTW-GEL (3DNow!) compared to FFTPACK and FFTW 2.1.3 on a 1.53 GHz AMD Athlon XP 1800+ carrying out complex-to-complex FFTs in single-precision. Performance data are displayed in pseudo-Gflop/s, i. e., $5N \log N/T$.

Fig. 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. The newly developed K7/FFTW-GEL utilizes the enhanced 3DNow! extensions which provide two-way single-precision SIMD operations. FFTW-GEL is about twice as

fast as FFTW 2.1.3, which demonstrates that the performance boosting effect of vectorization and backend optimization is outstanding.

Very recently, experiments were carried out on a prototype of IBM's BlueGene/L (BG/L) top performance supercomputer. Fig. 3 shows the relative performance of FFTW 2.5.1 no-twiddle codelets.

IBM's XLC compiler for BlueGene/L using code generation *with* SIMD vectorization and *with* FMA extraction (using the compiler techniques [8]) sometimes accelerates the code slightly but also slows down the code in some cases. FFTW-GEL's vectorization yields speed-up values up to 1.8 for sizes where the XLC compiler's register allocator generates reasonable code. For codes with more than 1000 lines (size 16, 32, 64) the performance degrades because of the lack of a good register allocation.

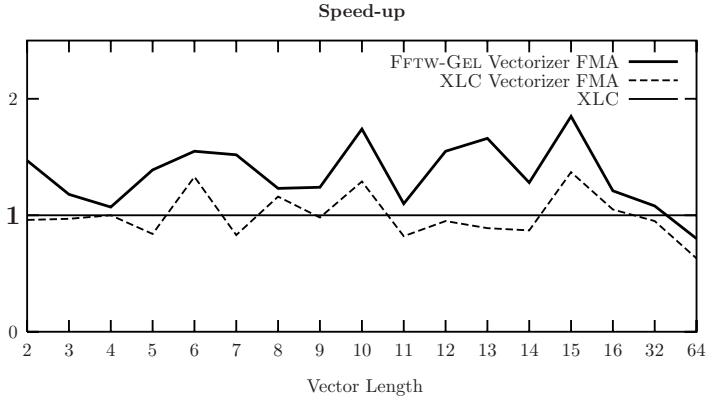


Fig. 3. Speed-up of (*i*) the newly developed BlueGene/L FFTW-GEL vectorizer *with* FMA extraction (but without backend optimizations) and (*ii*) FFTW codelets vectorized by XLC *with* FMA extraction compared to (*iii*) scalar FFTW codelets using XLC *without* FMAs, and *without* vectorization. The experiment has been carried out running no-twiddle codelets.

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 prospering cooperation and for making it possible for us to access non-public versions of FFTW.

Special thanks to Manish Gupta, José Moreira, and their group at IBM T. J. Watson Research Center (Yorktown Heights, N.Y.) for making it possible to work on the BG/L prototype and for a very pleasant and fruitful cooperation.

The Center for Applied Scientific Computing at LLNL deserves particular appreciation for ongoing support.

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., Püschel, M.: A SIMD vectorizing compiler for digital signal processing algorithms. In: Proc. International Parallel and Distributed Processing Symposium (IPDPS'02). (2002)
3. 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
4. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: ICASSP 98. Volume 3. (1998) 1381–1384
5. Guo, J., Garzarán, M.J., Padua, D.: The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks. In: Proceedings of the LCPC 2003
6. Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C.W.: SIMD vectorization techniques for straight line code. Technical Report AURORA TR2003-02, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology (2003)
7. Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C.W.: SIMD Vectorization of Straight Line FFT Code. In: Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, Springer-Verlag, Berlin (2003) 251–260
8. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. ACM SIGPLAN Notices Vol. **35** 5 (2000) 145–156
9. 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). `http://www.ece.cmu.edu/ spiral`
10. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufman Publishers, San Francisco (1997)
11. Sreraman, N., Govindarajan, R.: A vectorizing compiler for multimedia extensions. International Journal of Parallel Programming **28** (2000) 363–400
12. Srikant, Y.N., and Shankar, P.: The Compiler Design Handbook. CRC Press LLC, Boca Raton London New York Washington D.C. (2003)
13. G. Almasi et al., "An overview of the BlueGene/L system software organization," Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, 2003.
14. F. Franchetti and C. Ueberhuber, "An abstraction layer for SIMD extensions," Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, Technical Report AURORA TR2003-06, 2003.
15. Intel Corporation, "Intel C/C++ compiler user's guide," 2002.