FFT Compiler Techniques

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

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. Numerical applications are accelerated by automatically vectorizing blocks of straight line code for processors featuring two-way short vector SIMD extensions like Intel's SSE 2 on Pentium 4, SSE 3 on Pentium 5, 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 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 presented automatic SIMD vectorization can achieve performance that is comparable to the 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.

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 an 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.

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].

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 Pentium 5.

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.

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.

---

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

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.

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 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. FFTW 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. 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 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 Targets are summarized in the following items.

Minimizing Code Size. The greatest part of Fftw-Gel's optimizations is devoted to reducing a codelet's overall size. This kind of minimization is of the utmost importance because a processor's instruction cache is generally small. For instance, the Pentium 4's trace cache holds $12K$ of micro-operations. This is not overwhelmingly much, considering that each x86 instruction demands at least one micro-operation. Instructions using direct memory operands demand even more. Hence, a code size reduction helps to overcome the bottleneck of small instruction caches and improves the execution performance.
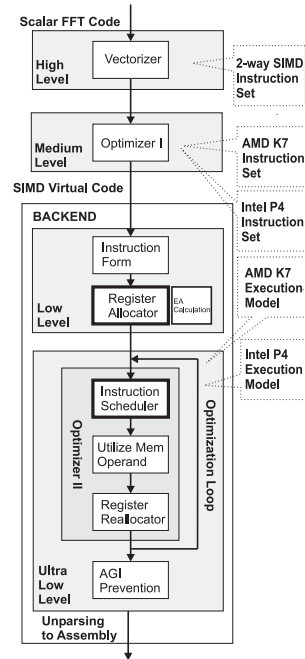
Speeding Up Integer Operations. Arithmetic integer operations in Fftw codelets are solely devoted to effective address calculations needed to access data arrays residing in memory. As general purpose compilers are normally not targeted at optimizing such integer calculations, Fftw-Gel introduces a special technique for automatically generating optimized code for effective address computations. The application of this optimization technique improves the performance of Fftw codelets significantly.

Fftw-Gel's vectorizer aims at the optimization of code size while the backend additionally aims at the optimization of integer arithmetics of scalar straight line Fftw codelets.

FFTW-GEL's Architecture. Fig. 1 schematically describes the various levels of optimization performed by Fftw-Gel.

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 efficient 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 a sequence of operations while maintaining data locality and utilizing special features of 2-way short vector SIMD extensions.

Fig. 1. Fftw-Gel's framework: Automatically generated FFT DAGs are vectorized and subjected to advanced optimization.

Optimization of Vectorized Code. Optimizer I comprises a local rewriting system using a set of rules to optimize the SIMD DAG obtained from the vectorizer. A first group of rules is used to reduce the number of instructions as far as possible, exploit redundancies, reduce the number of source operands, and finally eliminate dead code. A second group of rules is used to rewrite instructions

unsupported on a given target processor into supported ones. This process is hinted by processor specific instruction set models.

Straight Line Code Assembly Backend. 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 algorithm for the efficient calculation of effective addresses reuses already computed addresses as operands for other address computation code. This technique yields the shortest possible instruction sequences in address computation code.

The basic block register allocator that utilizes Belady's MIN algorithm as its spill heuristic, reduces the number of register spills and reloads. Hence, besides a performance improvement, the number of spill and reload instructions is considerably reduced in the final assembly code.

The instruction scheduler deals with the instruction latencies and throughputs of a specific target architecture by utilizing a model of the target processor's execution behavior.

Direct use of in-memory operands helps to discover weaknesses of the register allocator. Explicit Loads of some data from memory into a register are transformed into implicit loads if the data is used only once. Such register operands are discarded and substituted by equivalent memory operands. The register reallocator does not care about dropped out load instructions for discarded register operands in its subsequent allocation process.

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.

Finally, AGIs of load and store operations are detected and the conflicting instructions are reordered.

3 Vectorization of Scalar Straight Line Code

The goal of the vectorizer is to transform a scalar computation into short vector code while achieving the best possible utilization of SIMD resources. It automatically extracts *2-way SIMD parallelism* out of scalar FFTW code blocks (*i*) by fusing pairs of scalar temporary variables into SIMD cells, and (*ii*) by replacing the corresponding scalar instructions by vector instructions as illustrated by Tab. 1.

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


```
add(A,B,C)
            ⇒  vadd(AD,BE,CF)
add(D,E,F)
```
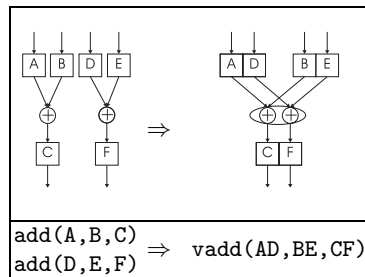
Table 1. 2-way Vectorization. Two scalar `add` instructions are transformed into a vector `vadd` instruction. The result of the vector addition of the fusions `AD` and `BE` is stored in `CF`.

On the set of vector variables, a sequence of SIMD instructions has to perform exactly the same computation as the given scalar code. This goal is achieved by pairing scalar instructions and replacing each pair by a sequence of semantically identical SIMD instructions operating on the corresponding SIMD cells. Thus, the vectorizer exhibits the following benefits:

Halving the Instruction Count. Optimally, every pair of scalar floating-point instructions is joined into one equivalent SIMD floating-point instruction, thus cutting the instruction count into half.

Diminishing Data Spills and Reloads. The wider SIMD register files allow for a more efficient computation. The register allocator indirectly benefits from the register file being twice as wide as in the scalar case.

Accelerating Effective Address Calculations. The pairing of memory access instructions potentially reduces the effort needed for calculating effective addresses by fifty percent.

To keep the amount of hardware specific details in the vectorization process as small as possible, *virtual machine models* are utilized. These models are sets of virtual instructions emulating the semantics of operations without incorporating any architecture specific syntactic idioms. The usage of such models enables portability and extensibility.

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. During the optimization process the resulting instructions are rewritten into instructions that are actually available in a specific architecture's instruction set.

3.1   The Vectorization Engine

The vectorization engine expects a scalar DAG represented by straight line code consisting of virtual scalar instructions in static single assignment (*SSA*) form as input. The goal of vectorization is to replace as many virtual scalar instructions as possible by virtual vector instructions. To achieve this goal, the vectorization engine has to find pairs of scalar floating-point instructions (and fusions of their respective operands), each yielding — in the optimal case — one SIMD floating-point instruction. In some cases, additional SIMD instructions may be required to obtain a SIMD construct that is semantically equivalent to the original pair of scalar instructions. The vectorization algorithm described hereby is the basis of Fftw-Gel's vectorization engine. First, some definitions:

Pairing Rules specify in detail ways in which pairs of scalar floating-point instructions can be transformed into a single or a sequence of semantically equivalent SIMD instructions. A pairing rule often provides several alternatives to do so. The rules used in Fftw-Gel's code generator can be classified according to the types of the two scalar instructions on which vectorization is to be performed, i. e., 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*) load/load, (*ii*) store/store, (*iii*) unary/unary, (*iv*) binary/binary, (*v*) unary/binary, (*vi*) unary/load, and (*vii*) load/binary.

Pairing Rule Sets comprise various pairing rules.

Instruction Pairing. Two scalar instructions can be vectorized, i.e., paired, if and only if neither of them is already involved in an existing pairing and the instruction types are matching a pairing rule from the utilized pairing rule set.

Operand Fusion. Two scalar operands `A` and `B` are fused, i.e., assigned together, to a SIMD cell `AB = (A,B)` or `BA = (B,A)` if and only if they are either source/source or destination/destination operands of instructions considered for pairing and neither of them is already involved in another fusion. The position of the scalar variables `A` and `B` inside a SIMD cell (either as its lower or its higher part) strictly defines the fusion, i.e., `AB ≠ BA`.

Compatible Fusion. A required fusion `X12 = (X1,X2)` is compatible to an already existing fusion `Y12 = (Y1,Y2)` if and only if `X12 = Y12` or `X1 = Y2` and `X2 = Y1`. In the second case, a special transformation is needed to allow the usage of fusion `Y12` whenever `X12` is needed.

Fftw-Gel's vectorization algorithm implements a depth first search with chronological backtracking. The search space is given by application of the rules in the currently utilized pairing rule set in arbitrary order. Depending on how restrictive the utilized rule set is in allowing instructions to be paired, there can be many, one or no possible solution at all.

3.2   The Vectorization Process

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

(2) Pick a fusion on the currently considered vectorization path, whose two writing instructions have not yet been paired. Because of the scalar code being in SSA form, there has to be exactly one instruction for each of the scalar variables present in the fusion that uses it as destination operand. According to 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 store 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.

(3) According to the chosen pairing rule, fuse the input variables 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.

(4) Pair the chosen scalar instructions, i.e., substitute them by one or more according SIMD instructions.

(5) If a fusion or pairing alternative does not lead to a valid vectorization, choose another one. 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 either all scalar instructions are vectorized, or the search process terminates without having found a result.

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 can be used to favor specific kinds of instruction sequences by ranking the corresponding rules before the others. For instance, the vectorization engine can be forced first to look for instructions directly supported by a given architecture, thus minimizing the number of extracted virtual instructions that have to be rewritten in the optimization step.

3.3 Vectorization Levels

Vectorization is a search process that requires to prune the search space as far as possible in order to minimize search time. On the other hand, the use of versatile rules, providing several vectorization alternatives, is indispensable to achieve a good vectorization result or to enable vectorization at all.

Taking both aspects into consideration, the concept of *vectorization levels* is introduced. Each level, embodied by an according rule set, allows a specific subset of the pairing rules to be applied.

Full Vectorization only provides rules for pairing instructions of the same type, because mixed pairings do not allow for a "clean vectorization". As it is the most restrictive vectorization level, it may happen that no full vectorization can be found at all.

Semi Vectorization allows all kinds of rules to be applied. Therefore, a valid vectorization is easily found. The drawback is that semi vectorization only results in "semi optimal" SIMD utilization.

Null Vectorization is used if a given scalar DAG cannot be vectorized at all. In this case, every scalar instruction is simply transformed into one corresponding SIMD instruction, leaving half its capacity unused.

The vectorization search space can be pruned by applying additional heuristic schemes that reduce the number of possible pairing partners in load/load and store/store vectorization. This pruning is done in a way to enforce the exploitation of obvious parallelism inherent in the code.

Full vectorization tries several heuristic schemes to vectorize memory access operations, whereas semi vectorization just relies on one of these schemes. Experiments have shown that the application of heuristic schemes significantly reduces the vectorization runtime.

3.4 Vectorization Specific Optimization

After the vectorizer terminates successfully by delivering a vectorization of the scalar DAG, Optimizer I performs several simple improvements to the resulting code (see Fig. 1). 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.

4 Backend Optimization for Straight Line Code

The backend translates a scheduled sequence of virtual 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 Low Level Optimization—Instruction Set Specific

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 accessed 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 for Straight Line Code

The codes to be dealt with in FFTW-GEL are potentially very large sequences of numerical straight line code produced by FFTW in SSA form. Thus, in such codes only one textual definition of a variable exists, there are no loops in the code, and every appearance of a temporary variable is known in advance. As a consequence of these properties, it is possible (i) to evaluate the effective live span of each temporary variable, and (ii) to perform register allocation according to Belady's MIN algorithm.

Belady's MIN Algorithm. Originally, *Belady's MIN algorithm* [1] was intended to provide a replacement strategy for paging in virtual memory: On replacement the page whose next utilization lies farthest in the future is chosen. However, in practice it is not always possible to predict the time when a page will be referenced again. The same difficulty, not to know when a register is to be used again, would exist if the MIN algorithm was used as a register replacement strategy for code containing loops. Consequently, general purpose compilers do not apply the MIN algorithm in their spilling schemes.

Numerical straight line code has more advantageous properties than general code. Therefore, it is possible in this case to rely on Belady's MIN algorithm. Whenever a register is to be spilled, the MIN algorithm chooses a register R that holds a value V such that the reference to V lies farther in the future than the references to the values held by all other registers.

Experiments carried out with numerical straight line code have shown that this simple spilling strategy is superior to the strategies utilized in general purpose C compilers [5].

Register Allocation in FFTW-GEL. The allocator's input is an instruction DAG in SSA form. This DAG is target processor specific, as it contains vector computation as well as integer address computation for the vector instructions' memory accesses to input and output arrays.

Taking these properties into consideration, registers have to be allocated for vector as well as integer registers in one overall allocation process. Two different target register files for vector and integer registers are assumed to exist.

The spilling scheme tries to find a *spill victim* in the following order: (i) take a fresh physical register if available, (ii) choose among the dead physical registers the one which has been dead for the longest time, and finally if no register of any other kind is available, (iii) choose a register from the spill candidates obtained by Belady's strategy.

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. The basic idea to achieve an efficient computation of effective array addresses is to use the powerful `lea` instruction of x86 compatible hardware architectures. The `lea` instruction 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. Thus, the `lea` instruction is a cheap alternative to general multiplication instructions.

The goal is to use the available integer register content as `base` or `index` operands in combination with one of the mandatory `scale` factors 1, 2, 4, or 8 to compute the desired effective address. On effective address generation, the shortest possible sequence of `lea` instructions is determined by depth-first iterative deepening. Therefore, effective addresses that have already been calculated and that are still stored in some integer register are reused as operands for other `lea` instructions whenever possible. New operands are put into vacant integer registers. Corresponding producer `lea` instructions are generated if and only if it is not possible to yield the effective address using available operands otherwise.

4.2   Ultra Low Level Optimization—Processor Specific

Typically, processor families supporting one and the same instruction set may still have different instruction latencies and throughput. Optimization therefore has to take into account the execution properties of a *specific* processor. These are provided by one execution model per processor.

The Optimizer II of FFTW-GEL uses two processor specific optimizations: (*i*) Instruction scheduling, and (*ii*) avoidance of address generation interlocks.

Target processor specific instruction scheduling is performed by taking into account the processor's execution behavior (latencies, etc.) and resources (issue

Integer Register File   *LEA Instructions*   *Result*

Reg	Entry
eax	stride*3
ebx	stride*5
edx	stride*1
esp	stride*4
esi	stride*9
edi	stride*-1

```
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
```

Table 2. Efficient Computation of ecx := 17*stride. Registers having an *Entry* in the *Integer Register File* are used as operands for the *LEA Instructions*. A combination of them with `lea`'s scale factor is to be chosen such that the factorization's result is equivalent to the direct computation of `17*stride`. This example illustrates an optimal case where all necessary operand factors to compute `ecx` already reside in some integer registers. The factors `1*stride` and `-1*stride` have an entry in the integer register file by default and are therefore assumed to be initially available.

slots, execution units, ports). These properties are provided as externally specified processor execution models, currently available for Intel's Pentium 4 and AMD's K7.

The Optimizer II can be regarded as one module executed on the vector instruction DAG in an optimization feedback loop. As long as the estimated execution time of the code can be improved, Optimizer II is applied. The runtime estimator models the target machine by simulating super-scalar execution of the code.

A final optimization technique applied to the output of Optimizer II is the reordering of memory access instructions to avoid address generation interlocks in assembly code.

Basic Block Instruction Scheduling

It is a highly demanding task to find a good execution order of instructions while (*i*) preserving data dependencies, (*ii*) efficiently utilizing execution units, and (*iii*) taking multiple instruction issue properties into account. This has to be considered whenever an optimal instruction schedule is to be created. The instruction scheduler of Fftw-Gel aims at single basic blocks.

The Processor Execution Model. As hardware properties not only differ from one vendor to another but also from one processor model to the next, it is laborious to implement a specific version of an optimizing backend for any single processor model. Fftw-Gel comes up with this issue by introducing a corresponding execution model for each target processor. These models specify (*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 available resources. For the sake of simplicity, the execution models do

not take address generation resources into account. In fact, the models assume that enough address generation units (AGUs) are available at any time.

List Instruction Scheduling. Fftw-Gel's instruction scheduler uses the information provided by the execution model to decide whether a given sequence of n instructions can be issued within the same cycle and executed in parallel by means of super-scalar execution while avoiding resource conflicts.

An instruction issue is possible if and only if (i) n is less or equal the number of processor slots per cycle, and (ii) enough distinct execution resources are available.

A local list scheduling algorithm is used as basically described in [10] and [12]. It utilizes information about critical path lengths of the underlying data dependency graph in a heuristic scheme to select instructions to be scheduled. The basic principles of this approach will be described in the following.

Instruction State. An instruction can be either in (i) ready state, (ii) preready state, or (iii) waiting state at a time. An instruction in *ready state* may be issued any time as its source data is already available. An instruction is in *preready state* if all instructions generating any of its source operands are already issued, but at least one of them is not yet finished. An instruction is in *waiting state* if at least one of its source producers is not yet issued.

Time Step. One time step corresponds to one clock cycle. At the outset, the time step counter is set to zero. As time steps advance, a state transition from waiting to preready and from preready to the ready state is imposed on every instruction. The time step is advanced, i.e., a new cycle is started, if and only if (i) there are no more ready instructions that can be issued without giving rise to an execution unit conflict, or (ii) when there are no more free instruction issue slots this cycle.

Issue Priority Heuristic. The issue priority heuristic selects among all ready instructions the instruction with (i) the longest critical path, and (ii) the suitability to be issued simultaneously with the instructions already selected earlier in the same time step according to their constraints.

One scheduling step of the list scheduling algorithm of Fftw-Gel's backend performs the following:

The process starts with time step 0.

(1) As long as there are ready instructions suitable to be issued and free issue slots are available, let the issue priority heuristic choose instructions to be issued in the current time step.

(2) Increase the time step and perform the instruction state transitions from waiting to preready and preready to ready state.

(3) If there are instructions left to be scheduled, commence with Step (1).

As the instruction scheduler is capable of guessing when every instruction will be executed with the help of the execution model, it serves as a basis for estimating

the runtime of the entire basic block. The estimated runtime, i. e., the time step value at the end of the scheduling process, can be used for controlling the optimization described in the following.

Direct Usage Of In-Memory Operands

After instruction scheduling and the succeeding low-level code motion, in-memory operands are directly applied. This results in an additional low level optimization.

As there is only a small number of physical registers available and many CISC instruction sets like those of Intel's Pentium 4 and Pentium 5, as well as AMD's K7 allow one source operand of an instruction to reside in memory, using in-memory operands can be advantageous. Usage of in-memory operands enables (i) to reduce register pressure, and (ii) to reduce the total number of instructions by merging one load and one use instruction into one load-and-use instruction.

Nevertheless, using in-memory operands can be harmful in some cases affecting both code size and performance. This may happen when (i) register files are not fully utilized, and (ii) superfluous loading of data occurs.

The goal is to use in-memory operands instead of registers for source operands if the data is accessed by only one instruction.

Register Reallocation

Register allocation is performed previous to instruction scheduling and the direct usage of in-memory operands in SIMD instructions. As instruction scheduling changes instruction order only locally, the spills and reloads of registers stay consistent and optimal. Due to the usage of in-memory operands, the utilization of the register file may not be optimal any more as entries in the register file are not referenced any longer, i. e., their usage as source operands is substituted by operands from memory.

Taking this observation into consideration, the register file assignments are reoptimized by performing register reallocation for SIMD registers. As the spills remain consistent and optimal, the task of allocation is simplified. The register reallocation does not have to take care about spills and reloads and reallocation can be performed using an LRU replacement strategy. Moreover, the reallocator has to perform a renaming of registers (i) to sustain the register's future usage dependencies, and (ii) to preserve the spills and reloads computed in the register allocator.

Integer instructions and the integer register file assignment remain unchanged, as direct use of in-memory operands has only been performed for SIMD instructions.

Avoiding Address Generation Interlocks

Current super-scalar microprocessors like Intel's Pentium 4 and Pentium 5, as well as AMD's K7 allow out-of-order execution. Nevertheless, instructions accessing memory, i.e., loads and stores, must still be executed in-order.

Problems may evolve whenever a memory operation involving expensive effective address calculation directly precedes a memory operation with cheap or no effective address calculation. In such cases, both memory operations are delayed for the duration of the first, expensive effective address calculation. An address generation interlock (AGI) occurs.

Example (Address Generation Interlock) Assume the following unoptimized x86 assembly code in AT&T-assembler syntax:

```
movq (%eax,%edx,8), %mm1 /* I1 */
movq (%eax), %mm2        /* I2: interlocked */
```

As instructions I1 and I2 are (read-after-write, write-after-read, and write-after-write) independent, one would expect them not to interfere with each other. However, as the Intel Pentium architecture definition requires load and store instructions to be executed in-order, the instruction I2 is forced to wait until instruction I1 has calculated its effective address and both dependent memory operations are stalled meanwhile.

Fftw-Gel tries to avoid address generation interlocks by reordering instructions immediately after instruction scheduling. The calculation of effective addresses does not involve constant costs on the Intel Pentium processors. Calculating effective addresses not involving shift operations by some constant factor are cheaper than those that do. Taking this knowledge into consideration, instructions are reordered.

**Example (Optimized Address Generation)** In the reordered code

```
movq (%eax), %mm2        /* I2: AGI avoided */
movq (%eax,%edx,8), %mm1 /* I1 */
```

from the above example, the order of the instructions has been adapted so that I2 occurs before I1 in the program text. Thus, the memory operation waiting for the result of I2 can almost immediately proceed without having to wait for the expensive calculation of I1 to finish.

5 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).

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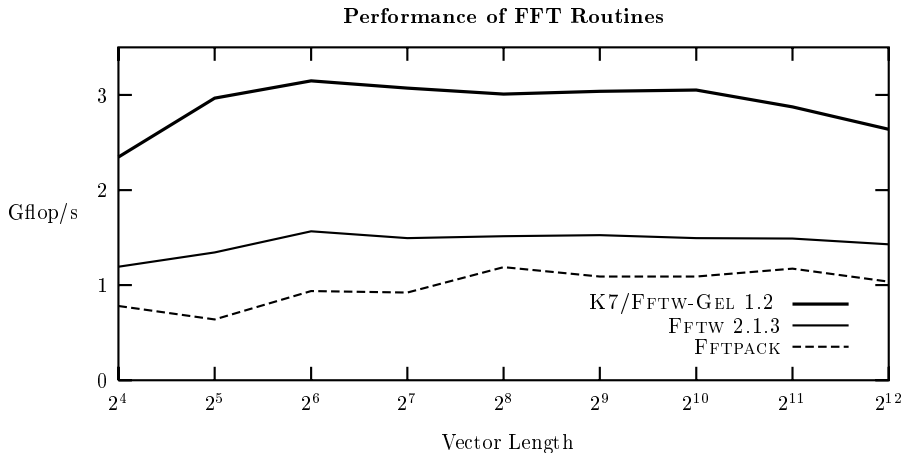
Fig. 2. Floating-point performance of the newly developed K7/Fftw-Gel (3DNow!) compared to Fftpack and Fftw 2.1.3 on a 1.53 GHz AMD Athlon XP 1800+ carrying out complex-to-complex FFTs in single-precision. Performance data are displayed in pseudo-Gflop/s, i. e., $5N \log N/T$.

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 Blue-Gene/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.

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.
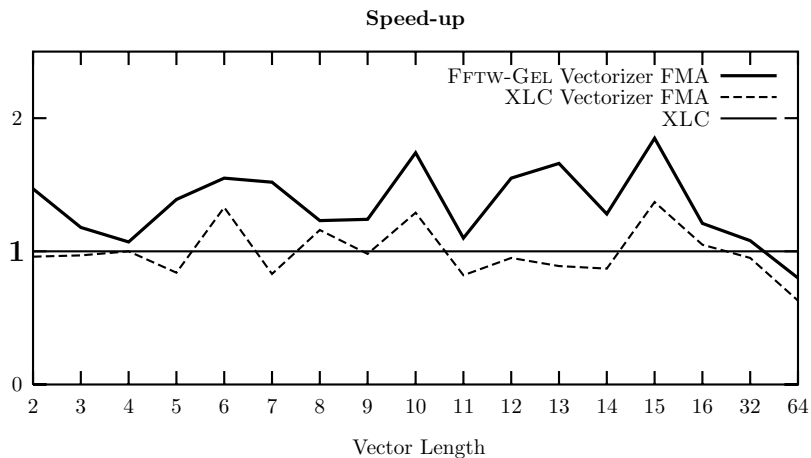
Speed-up

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.

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.

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 Beladys 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)