Vectorization techniques for the Blue Gene/L double FPU

J. Lorenz
S. Kral
F. Franchetti
C. W. Ueberhuber

This paper presents vectorization techniques tailored to meet the specifics of the two-way single-instruction multiple-data (SIMD) double-precision floating-point unit (FPU), which is a core element of the node application-specific integrated circuit (ASIC) chips of the IBM 360-teraflops Blue Gene®/L supercomputer. This paper focuses on the general-purpose basic-block vectorization and optimization methods as they are incorporated in the Vienna MAP vectorizer and optimizer. The innovative technologies presented here, which have consistently delivered superior performance and portability across a wide range of platforms, were carried over to prototypes of Blue Gene/L and joined with the automatic performance-tuning system known as Fastest Fourier Transform in the West (FFTW). FFTW performance-optimization facilities working with the compiler technologies presented in this paper are able to produce vectorized fast Fourier transform (FFT) codes that are tuned automatically to single Blue Gene/L processors and are up to 80% faster than the best-performing scalar FFT codes generated by FFTW.

Introduction

The IBM Blue Gene*/L (BG/L) supercomputer [1], planned to be in operation in 2005, will be an order of magnitude faster than the Earth Simulator. BG/L will feature eight times more processors than current massively parallel systems. To tame this vast parallelism, new approaches and tools have had to be developed. However, developing highly efficient numerical software has to start with optimizing the computational kernels for the nonstandard floating-point unit (FPU) of the BG/L processors. This so-called *double FPU* provides support for complex arithmetic as an important prerequisite to speed up large scientific codes.

The utilization of nonstandard FPUs in computational kernels, like fast Fourier transforms (FFTs), is by no means straightforward. Optimization of FFT kernels leads to complicated data dependencies of real variables that cannot easily be mapped to the elaborate BG/L FPU. This problem is particularly demanding in the context of automatic performance tuning, but it must be addressed in order to obtain high-performance FFT implementations, which are required as major building blocks for the large scientific codes planned to be run on BG/L. Most of these applications require very fast one-dimensional FFT routines to be run on a single processor for computing relatively small transforms.

This paper introduces a new FFT library, BGL/FFTW–GEL, that runs efficiently on the BG/L prototypes. This library is the first numerical library for BG/L not developed by IBM. It takes full advantage of the double FPU by means of short-vector single-instruction multiple-data (SIMD) vectorization.

BGL/FFTW–GEL is the result of a combination of FFTW with special-purpose vectorization technology in the Vienna MAP vectorizer [2–4]. FFT codes produced by BGL/FFTW–GEL are running five times faster than standard nonadaptive FFT libraries [2]. On the DD2 prototype, speeds up to 1.8 times greater than the best FFT code not utilizing the special features of the BG/L double FPU were achieved.

The Blue Gene/L supercomputer

The initial DD1 prototype of the IBM Blue Gene/L supercomputer [1], equipped with 8,192 custom-made

0018-8646/05/$5.00 © 2005 IBM

IBM PowerPC* 440 FP2 processors running at 500 MHz, achieved a Linpack performance of R_{max} = 11.7 teraflops, i.e., 71% of its theoretical peak performance of R_{peak} = 16.4 teraflops. This performance ranks the BG/L prototype machine at position four of the Top500 list (June 2004) [5]. The prototype machine is roughly 1/20th the physical size of machines of comparable compute power—such as Linux** clusters—that exist today.

The 64K-processor BG/L machine currently being built for the Lawrence Livermore National Laboratory (LLNL) will be eight times larger, occupying 64 full racks. When completed in 2005, the LLNL supercomputer—featuring 360 teraflops peak performance—is expected to lead the Top500 list. Compared with the fastest supercomputers of today, it will be an order of magnitude faster, consume 1/15th of the power, and be ten times more compact.

Complex and real arithmetic

Since there are many areas of scientific computing, such as computational electronics, in which complex arithmetic plays an important role, its native support has been integrated into the FPUs of computers devoted to such applications. Nevertheless, even algorithms using complex arithmetic may have to be reformulated in terms of real arithmetic to allow for the application of the inevitable optimization techniques to achieve satisfactory performance of scientific codes: common subexpression elimination, constant folding, and copy propagation on the real and imaginary parts.

BG/L double FPU

The BG/L PowerPC 440 (PPC440) double floating-point FPU (FP2) was obtained by replicating the standard PPC440 FPU and adding crossover datapaths and sign-change capabilities to allow the short-vector SIMD fused multiply–add (FMA) operations to support complex multiplication. Up to four real floating-point operations (one SIMD FMA) can be issued every cycle, and efficient intermixing of scalar and vector operations is possible. The data to be processed has to be naturally aligned on 16-byte boundaries in memory.

The PPC440 FP2 exhibits some problematic characteristics: a single data reorder operation within a short-vector SIMD register is as expensive as one arithmetic two-way FMA operation, and alternatively, either a floating-point operation or a data reorganization instruction can be issued every cycle. Conventional vectorization techniques are not able to deal efficiently with these architectural shortcomings. Thus, without a tailor-made adaptation of established short-vector SIMD vectorization techniques to the specific features of the BG/L double FPU, no high-performance short-vector code can be obtained.

Blue Gene/L ISA extension

The Blue Gene/L new single-instruction, multiple-operation, multiple-data (SIMOMD) instruction set architecture (ISA) extension includes all well-known short-vector SIMD-style (interoperand, parallel) instructions, such as the ones supported by the Intel** Streaming SIMD Extensions 2 (SSE2) or AMD 3DNow!**. This ISA extension allows the use of the double FPU either as a complex FPU or as a real two-way vector FPU.

When it is used as a complex FPU, programs using complex arithmetic can be mapped to the Blue Gene/L double FPU in a straightforward manner. The alternative, using it as a real two-way vector FPU where real code is presupposed, is applicable only if the underlying algorithm allows for enough parallelism to be extracted. Moreover, the mapping is significantly more complicated in this case.

In the context of automatic performance-tuning software, the GNU C compiler port for Blue Gene/L is not suitable because it supports only 32 temporary variables when accessing the double FPU. Thus, only the IBM XL C compiler is a reasonable choice.

To employ the Blue Gene/L double FPU in automatic performance-tuning software, three approaches are possible:

1. Implement the numerical kernels in C using proprietary directives such that the XL C compiler vectorization possibilities prove successful.
2. Rewrite the numerical kernels in assembly language using specific double FPU instructions.
3. Rewrite the numerical kernels utilizing the XL C language extension to C99 that provides access to the double FPU on the source level by means of special data types and intrinsic functions.

This paper describes how vector code can be generated automatically by following the third approach. Thus, register allocation and instruction scheduling is left to the compiler while vectorization and instruction selection is done at the source-code level by the newly developed approach presented in the following sections.

Self-adapting FFT software

The FFT algorithm is among the most important computational innovations of the twentieth century and continues to be a focus of current research. In scientific computing, FFT algorithms are—in addition to linear algebra algorithms—core algorithms of almost any computationally intensive numerical software. Accordingly, the application of FFT transforms ranges from small-scale problems with stringent time constraints (for instance, in real-time signal processing) up to large-scale simulations and partial differential equation (PDE)

solvers running on the largest supercomputers in the world. Thus, high-performance software tailor-made for such applications is desperately needed.

FFT algorithms are structurally complex and difficult to efficiently map onto standard hardware. Until recently, FFT packages required large, finely tuned, machine-specific libraries produced by highly skilled software developers. Therefore, these packages failed to perform well for a variety of architectures.

In 1997, the state of the art in scientific software production changed dramatically when automatic software generators producing highly optimized code entered the field. New standards were set by FFTW, a free collection of fast C routines for computing the discrete Fourier transform (DFT) in one or more dimensions [6, 7]. FFTW was designed for producing automatically tuned FFT libraries and automatically tuned linear algebra software (ATLAS) [8] that generates highly efficient basic linear algebra subroutines (BLAS).

Typically, FFTW produces code that runs faster than publicly available FFT codes and compares well to vendor libraries. A dynamic programming approach relying on a recursive implementation of the Cooley/Tukey FFT algorithm [9] enables the adaptation of the FFT computation of a given size to a given target machine at runtime. The actual computation is done within routines called *twiddle* and *no-twiddle codelets*, which are produced by the code generator GENFFT [10], whose output consists of basic blocks of thousands of lines of code that can be transformed into static single assignment (SSA) form.

Code generated by automatic performance-tuning software such as FFTW and ATLAS is supposed to be translated by standard compilers to enable portability. However, automatically generated numerical code translated by standard compilers is often not able to achieve satisfactory performance. To accomplish top performance in such cases, the exploitation of special processor features, such as short-vector SIMD or FMA ISA extensions, is imperative.

Unfortunately, the methods used by conventional vectorizing compilers to deal with loops or basic blocks lead to inefficient results when applied to automatically generated numerical codes. These inefficiencies are due, among other things, to the inability of such compilers to utilize domain-specific information revealing the parallelism inherent in the codes. For instance, conventional vectorization techniques entail unacceptably large overhead by applying data-reordering operations that are, in principle, nonessential.

Related work

The main topic of this paper is the automatic vectorization of basic blocks of automatically generated code. Besides this kind of vectorization, there are also other ways to automatically vectorize FFT code.

Formal FFT vectorization

The formal vectorization approach [11–16] developed for classical short-vector SIMD extensions, such as the Intel SSE family, the AMD 3DNow! family, and the Motorola AltiVec**, has been ported successfully to Blue Gene/L FPUs [17]. This type of vectorization is based on the SIMD vectorizing version of the synchronous programming language (SPL) compiler [14] that enables the SPIRAL system [18, 19] to automatically optimize code targeted at the double FPUs of Blue Gene/L.

Vectorization in FFTW 3

Version 3.0.1 of FFTW supports the SIMD extensions SSE, SSE2, 3DNow!, and AltiVec. A new algorithm is used to compute complex DFTs by means of two-way parallel computation of real DFTs. Porting FFTW 3.0.1 to Blue Gene/L requires the mapping of the SIMD instructions required by FFTW to instructions that exist on Blue Gene/L. Preliminary experiments carried out with no-twiddle codelets show promising results.

Other vectorization techniques

Some methods for vectorizing basic blocks [20–22] attempt to find an efficient mix of SIMD and scalar instructions to carry out the required computation, whereas the vectorization techniques introduced in the next section aim at a full utilization of the power of SIMD instructions while trying to keep the SIMD reordering overhead reasonably small.

The vectorization method of [21] introduces more SIMD data-reordering instructions than necessary, because it is unable to use a representation of the numerical scalar directed acyclic graph (DAG) as vectorization input, and is thus deprived of this parallelism-revealing instrument. This approach is not a suitable choice for the efficient handling of typical numerical codes, such as FFTs, since explicit SIMD data-reordering operations are very expensive on the Blue Gene/L FPU.

Vienna MAP vectorizer

The Vienna MAP vectorizer [2–4, 23] automatically extracts two-way short-vector SIMD parallelism from a scalar code block by adequately combining scalar variables into SIMD variables and by joining the corresponding scalar instructions to one or more short-vector SIMD instructions, as illustrated in **Figure 1**. The MAP vectorizer targets automatically generated code that consists solely of arithmetic operations and read/write array access operations involving index computation.

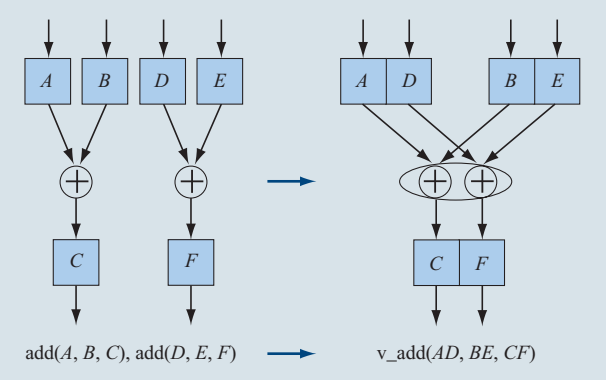

Figure 1

Example of two-way vectorization. Ideally, two scalar instructions are transformed into one vector instruction to achieve optimal SIMD coverage.

Ideally, two-way vectorization transforms any pair of scalar instructions to one SIMD instruction yielding 100% SIMD utilization (Figure 1). This maximum is achievable only for completely parallel scalar DAGs. For DAGs with less parallelism, SIMD reordering instructions are required, at the cost of reduced SIMD utilization.

Since not all combinations of scalar operations may be joined into one SIMD instruction (as defined by the ISA extension of the target processor), a realistic goal for the vectorizer is to completely cover the given scalar DAG by natively supported SIMD instructions while achieving a satisfactory runtime performance, which is tantamount to minimizing SIMD data reorganization. **Figure 2** gives an example of short-vector SIMD code obtained by vectorizing straight-line complex FFT code.

The Vienna MAP vectorizer was adapted to support Blue Gene/L FPUs. As a supplement to the MAP vectorizer, a peephole optimizer enables the extraction of fused multiply–add SIMD instructions.

Fundamentals of vectorization

Fusion of variables

Two scalar variables A and B can be fused either into a SIMD variable of the form $AB = (A, B)$ or vice versa, $BA = (B, A)$, where $AB \neq BA$. Moreover, no scalar variable can be part of two different SIMD variables.

An already existing fusion $AB = (A, B)$ is said to be compatible with another fusion $CD = (C, D)$ requested in the vectorization process if and only if $AB = CD$ or $A = D$ and $B = C$. In the first case, fusion CD does not have to be generated, since AB can be used. In the second case, a SIMD swap operation is required to maintain

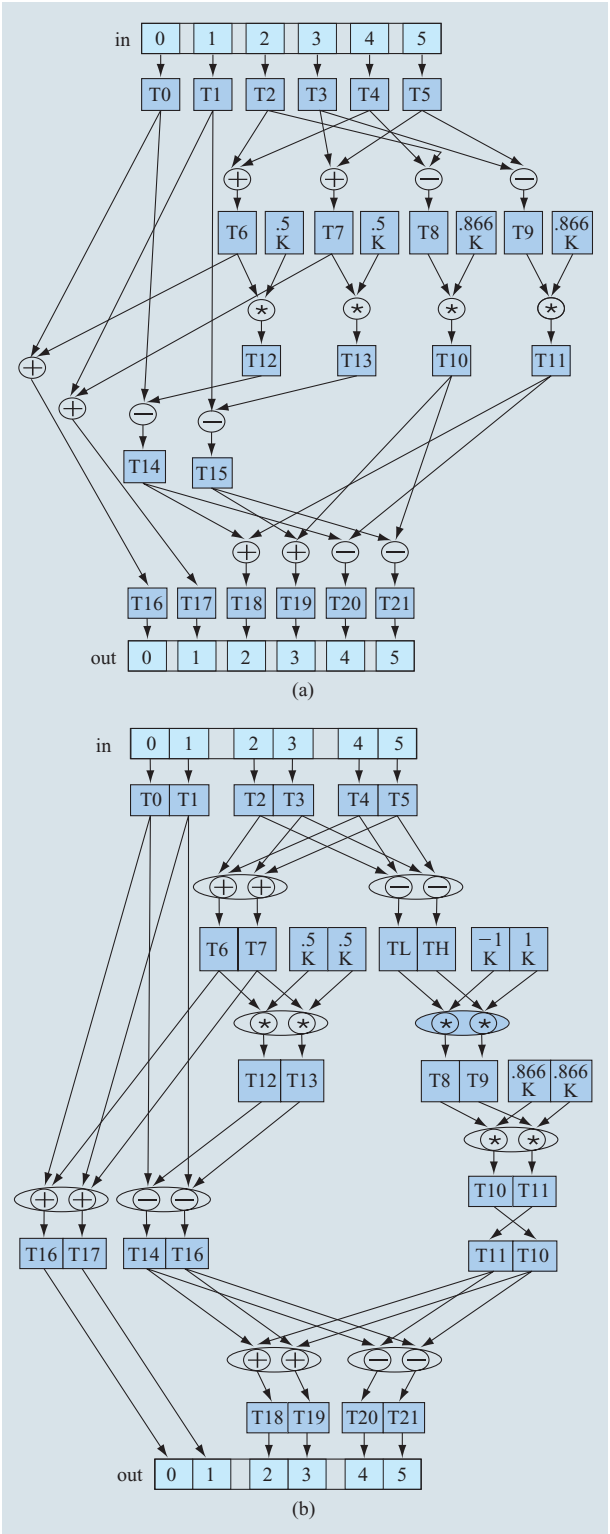

Figure 2

Vectorization of a scalar FFT of size 3. The scalar data flow in part (a) is computationally equivalent to the vectorized data flow of part (b).

data-flow consistency when using fusion *AB* instead of generating and using *CD* (**Figure 3**).

Combination of operations

Joining rules specify the allowed ways of pairwise transforming scalar instructions into one or more SIMD instructions. The MAP vectorizer supports transformations of the following instruction combinations: 1) load/load, 2) store/store, 3) unary/unary, 4) binary/binary, 5) unary/binary, 6) unary/load, and 7) load/binary.

Joining rules 1 and 2 supports the transformation of memory operations accessing consecutive and nonconsecutive memory locations. Rules 3 and 4 allow the pairing of instructions of only the same type, while rules 5 to 7 also allow mixed-type pairings.

Rules of type 3 fuse the two source operands *S1* and *S2* for transforming two unary instructions (*uop1*, *S1*, *D1*) and (*uop2*, *S2*, *D2*).

Rules of type 4 provide several alternatives (**Figure 4**). Because they target two binary instructions (*bop1*, *S1*, *T1*, *D1*) and (*bop2*, *S2*, *T2*, *D2*), different possibilities arise for choosing the fusion partners among the four source operands *S1*, *T1*, *S2*, and *T2*. Thus, three layouts—ACC, PAR, and CHI, which define the possibilities for fusing the operand variables for binary instructions as shown in **Figure 5**—are introduced. ACC is needed for fusing variables used as operands for SIMD instructions of intra-operand style, whereas PAR and CHI are meant for those of parallel style, as illustrated in Figure 4.

Vectorization quality

To extract high-performance short-vector SIMD code distinguished by good SIMD utilization, the joining rules issue SIMD reorder instructions only in the unavoidable case of a compatible fusion demanding a swap instruction. The majority of the extracted swaps can be removed by next applying a peephole optimization after the vectorization process.

The vectorization engine begins by constraining all SIMD memory operations to access consecutive locations and by disabling the suboptimal pairing rules 5–7. If these restrictions cause the vectorization process to fail, it is restarted after enabling operation pairing rules 5–7 and support for less efficient (that is, nonconsecutive) memory access operations. Abandonment of restrictions substantially augments the class of vectorizable codes by allowing the extraction of some less efficient instruction combinations.

MAP vectorization algorithm

Before the actual vectorization process is started, the following preparatory steps are taken. First, a

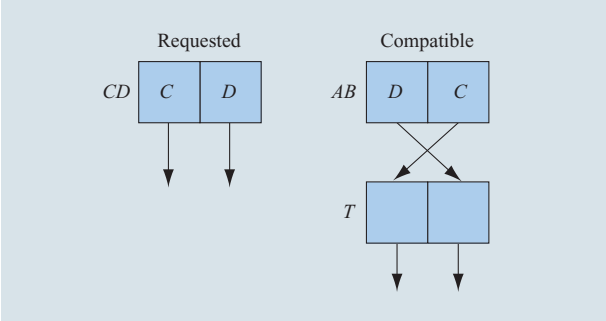

Figure 3

Compatible fusion. The vectorization process requests the fusion $CD = (C, D)$. The existing fusion $AB = (D, C)$ is used as the input operand of a swap instruction whose output T can be used whenever CD is needed.

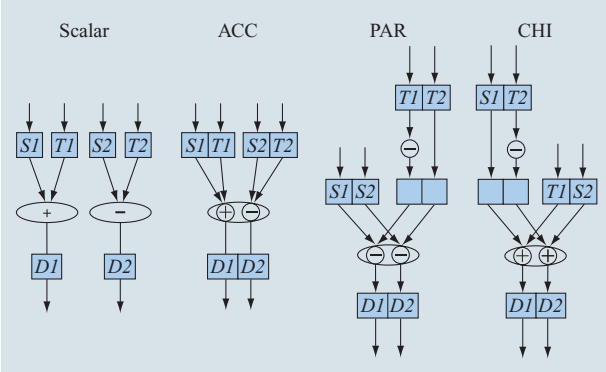

Figure 4

Vectorization alternatives. Two scalar instructions, one addition and one subtraction operation, are transformed into SIMD instructions in three different ways.

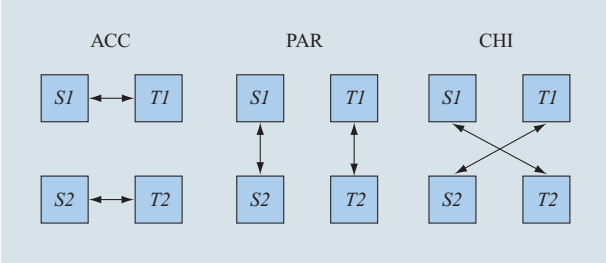

Figure 5

Fusion layouts. Three layouts for fusing the source variables of the scalar instructions (*op1*, *S1*, *T1*, *D1*) and (*op2*, *S2*, *T2*, *D2*) are supported.

Table 1 Relative costs of SIMD operations. For a selection of ISA extensions, the actual number of SIMD instructions required to implement the respective SIMD operations is given. This data directly influences the rule ranking underlying the MAP vectorizer.

ISA / *SIMD op*	*Basic 3DNow! (K6-II+)*	*Extended 3DNow! (K7/K8)*	*SSE2 (P4/K8)*	*SSE3 (P4e)*	*IA64 (Intel Itanium**)*	*Double FPU (440 FP2)*
Load/store	1	1	1	1	1	1
Uniform unpack	1	1	1	1	1	2
Mixed unpack	2	2	2	2	1	2
Uniform ACC	1	1	3	1	3	5
Mixed ACC	2	1	4	2	3	5
Uniform PAR	1	1	1	1	1	1
Mixed PAR	2	2	2	1	1	1
PAR FMA	2	2	2	2	1	1

dependency analysis is performed on the scalar DAG. Then, instruction statistics are assembled which provide instruction counts for each instruction type and operation. Data gathered in these first two steps is used as a heuristic to speed up the vectorization process by avoiding futile vectorization attempts. Finally, store instructions are combined nondeterministically by fusing their respective source operands.

Vectorization algorithm

The MAP vectorization algorithm consists of two steps:

1. Pick $I1 = (op1, s1, t1, d1)$ and $I2 = (op2, s2, t2, d2)$, i.e., two scalar instructions that have not yet been vectorized, with $(d1, d2)$ or $(d2, d1)$ being an existing fusion.
2. The two scalar operations *op1* and *op2* are paired nondeterministically, yielding an equivalent sequence of SIMD operations. This step may impose the need for new fusions if no compatible fusions are available. In this case, the layout for the fusion of the respective source operands *s1*, *t1*, *s2*, and *t2* is mandated by the pairing rule. The vectorization process must ensure that no scalar variable is part of two different fusions.

The vectorizer alternately applies step 1 and step 2 until either the vectorization succeeds (i.e., thereafter all scalar variables are part of at most one fusion, and all scalar operations have been paired) or the vectorization fails. If the vectorizer succeeds, it immediately commits to the first solution of the search process, which keeps the vectorization runtime reasonably small. Although a search for the solution that achieves the shortest runtime would be desirable, it is not feasible using the current version of the vectorizer, even for relatively small straight-line codes.

Nondeterminism in vectorization

Nondeterminism in vectorization arises due to vectorization alternatives such as ACC, PAR, and CHI, for binary/binary pairings. For a fusion $(d1, d2)$, there may be several layouts for fusing the source operands *s1*, *t1*, *s2*, and *t2*, depending on the pairing (*op1*, *op2*), as illustrated in Figure 4. This kind of nondeterminism widens the search space of the vectorizer backtracking search engine.

The rule ranking, i.e., the order in which vectorization alternatives are tried, may influence the order of the solutions of the vectorization process. Because the vectorizer always commits to the first solution, the rule ranking is adapted such that the first solution favors instruction sequences that are particularly well-suited for the given target machine, taking into account the different costs of individual instructions (**Table 1**).

The rule ranking has to be considered as a kind of extraction "hint." At every point of decision, the search engine initially tries the rule that is ranked first. If this does not succeed, i.e., does not lead to a vectorization, later-ranked rules are also used, even if their application effectuates the extraction of pseudo instructions that are not supported on the target ISA. This kind of retreat is unavoidable, as a complete vectorization is the central goal.

Realization of the vectorization engine

The MAP vectorization algorithm is implemented using a depth-first search engine with chronological backtracking. This backtracking capability is indispensable when a fusion that is requested by the

current vectorization alternative does not comply with the globally existing fusions. In these cases, the search engine backtracks to the last nondeterministic point of decision. There, another vectorization alternative is chosen, and corresponding fusions of different layouts are requested and generated if necessary. If these fusions comply with the set of existing fusions, the rule fires and the vectorization process continues. Otherwise, backtracking is chronologically applied repeatedly until either a vectorization is obtained or the search space is exhausted. In the latter case, the vectorization engine is unable to find a valid fusion set for the given scalar DAG.

Vienna MAP optimizer

The Vienna MAP optimizer is a rule-based local rewriting system that implements peephole optimization on vector DAGs. It postprocesses the output of the MAP vectorizer and comprises two groups of rewriting rules. Finally, the optimized output is sorted topologically in an attempt to minimize the lifespan of variables by improving the locality of variable accesses, using a scheduling algorithm based on the FFTW scheduler GENFFT [10].

General set of rules

The first group of rewriting rules consists of general optimization rules (**Figure 6**) aiming at minimization of the instruction count, elimination of redundancy and dead code, reduction of the number of source operands (which reduces register pressure), minimization of the critical path length of the vector DAG, copy propagation, and constant folding.

With target architectures that support FMAs, such as the Blue Gene/L double FPU, the FMAs are extracted by combining multiplications (or sign changes) with directly dependent addition operations (or subtraction operations or already existing FMAs) into FMAs. If this direct combination is not possible at first, the respective instructions are moved down in the DAG in an attempt to fold them into other instructions.

Specific set of rules

The second group of rewriting rules is specific to target architecture. When optimizing for the Blue Gene/L PPC440 FP2, vector swap instructions are folded into FMAs utilizing vector FMA instructions with crossed datapaths, exclusively available on Blue Gene/L, using a method similar to FMA extraction (**Figure 7**).

Experimental results

Numerical experiments based on one-dimensional FFTs applied to vectors with power-of-2 lengths $N = 2^3, 2^4, \cdots, 2^{18} = 262{,}144$ and non-power-of-2

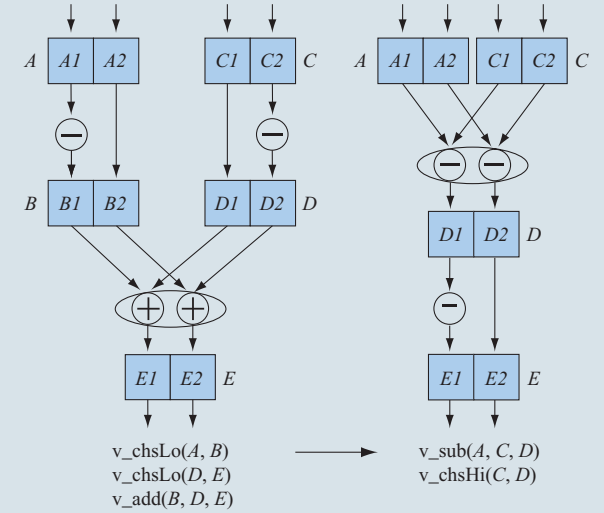

Figure 6

General optimization rule. A vector add instruction v_add(B, D, E)—taking the output of two sign change instructions, one on the lower part v_chsLo(A, B) and another on the higher part v_chsHi(C, D) of two different registers as its inputs—is transformed into a vector subtraction v_sub(A, C, D) and a subsequent vector sign change v_chsLo(D, E) instruction.

lengths $N = 9, 12, \cdots, 27{,}000$ were carried out using FFTW combined with the new vectorization and optimization techniques as they are implemented in the Vienna MAP vectorizer and optimizer.

In these experiments the new vectorization and optimization techniques were built into BGL/FFTW–GEL and evaluated on the Blue Gene/L DD2 prototype using one single PowerPC 440 FP2 processor running at 700 MHz. The best-performing scalar code generated by FFTW as well as FFTW code vectorized by the XL C compiler is used as a performance benchmark.

The floating-point performance displayed in **Figures 8(a)** and **8(b)** is given in pseudo Gflops, i.e., $5N \log N/T$, with N being the vector length and T the measured runtime in nanoseconds.

Figures 8(a) and 8(b) compare different FFTW library implementations:

- The best vectorized code obtained using all technologies presented in this paper (BGL/FFTW–GEL).
- The best scalar FFTW implementation (with the XL C vectorizer and FMA extraction facility turned off).
- The best vectorized FFTW implementation obtained with the activated vectorizer and FMA extraction facility of the XL C compiler.

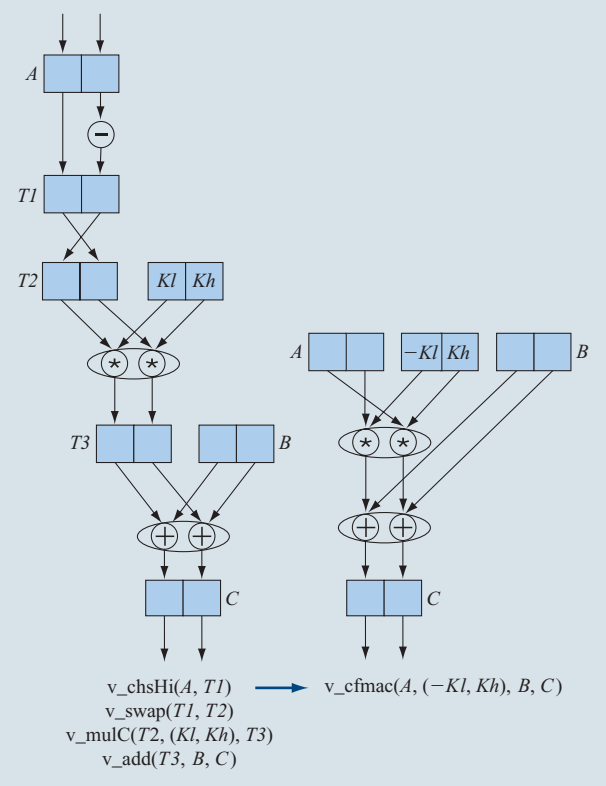

Figure 7

Blue Gene/L specific optimization rule. A vector multiplication with a constant v_mulC(*T2*, (*Kl*, *Kh*), *T3*) taking the output of v_swap(*T1*, *T2*), i.e., a vector swap instruction, preceded by a vector sign change v_chsHi(*A*, *T1*) is transformed into a vector cross FMA instruction v_cfmac(*A*, (−*Kl*, *Kh*), *B*, *C*) if the contents of the temporary variables *T1*, *T2*, and *T3* are not referenced anywhere else in the vector DAG.

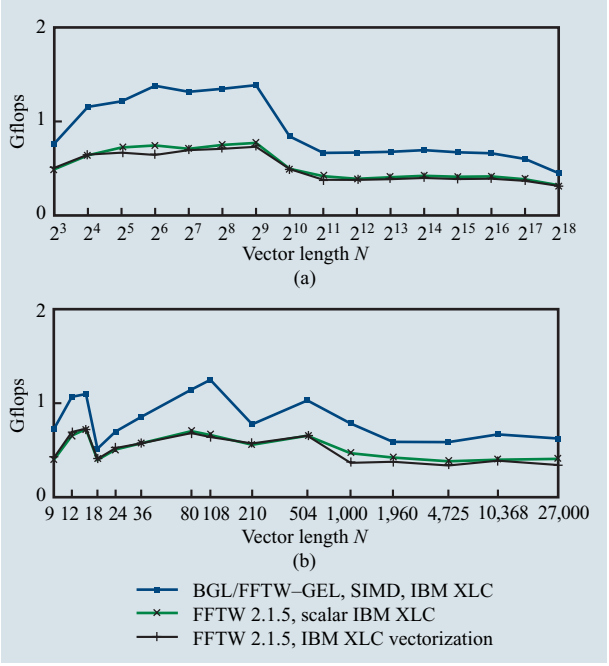

Figure 8

Performance of one-dimensional complex to complex (a) power-of-two and (b) non-power-of-two FFTs, each running on a single PowerPC 440 FP2 node at 700 MHz.

The impressive performance gain attainable with BGL/FFTW–GEL over scalar code generated by FFTW is shown in **Figure 9**. For power-of-2 FFTs, BGL/FFTW–GEL yields speedup figures up to 80% with respect to the best-performing scalar code, i.e., the scalar FFTW library (which serves as a baseline in the diagram). For large problem sizes, BGL/FFTW–GEL still yields a speedup of 60%. Thus, the vectorization and optimization methods of this paper have been demonstrated to effectuate significant performance improvements. XL C, with its vectorization and FMA extraction facility turned on, when applied to FFTW-generated code without taking advantage of the techniques presented in this paper, produces vectorized code that runs at the same speed or slightly slower than scalar XL C code.

Conclusions and outlook

FFTs are indispensable parts of nearly every kind of scientific computing application. Thus, efficient FFT software is needed by Blue Gene/L scientific users. The performance portable vectorization techniques introduced in this paper allow timely software optimization to be done concurrently with IBM hardware development on Blue Gene/L.

The highly portable Vienna MAP vectorizer can be used to automatically vectorize numerical straight-line code generated by state-of-the-art automatic performance-tuning software, such as FFTW, thereby helping to develop highly efficient implementations of FFT kernels.

Performance experiments carried out on Blue Gene/L prototypes show that the newly developed vectorization approach in combination with state-of-the-art performance-tuning software is able to speed up numerical codes considerably. The vectorization approach of this paper has been demonstrated to produce high-performance FFT kernels for the Blue Gene/L supercomputers that fully utilize the new double FPU.

An integral part of our current work is a compiler back end that will be particularly well-suited for the compilation of numerical straight-line code.

Acknowledgments

We thank Matteo Frigo and Steven G. Johnson for years of pleasant and productive cooperation. Special thanks go to Manish Gupta, José Moreira, and their group at the

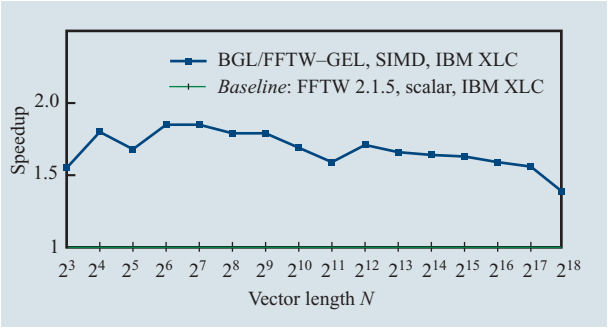

Figure 9

Speedup achieved by the vectorization and optimization techniques presented in this paper, compared with scalar code. In both cases the XL C compiler back end was used.

IBM Thomas J. Watson Research Center for making it possible to work on the Blue Gene/L prototype and for a very pleasant and fruitful cooperation. The Center for Applied Scientific Computing at Lawrence Livermore National Laboratory deserves particular appreciation for ongoing support. Additionally, we would like to acknowledge the financial support of the Austrian Science Fund FWF. The work described in this paper was supported by the Special Research Program SFB F011 "Aurora" of the Austrian Science Fund FWF. Franz Franchetti was supported by the Austrian Science Fund FWF's Erwin Schroedinger Fellowship J2322.

References

1. J. E. Moreira, G. Almási, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castaños, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith, and G. G. Stewart, "Blue Gene/L Programming and Operating Environment," *IBM J. Res. & Dev.* **49**, No. 2/3, 367–376 (2005, this issue).
2. F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient Utilization of SIMD Extensions," *Proc. IEEE* **93**, No. 2, special issue on "Program Generation, Optimization, and Platform Adaptation" (2005).
3. S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber, "SIMD Vectorization of Straight Line FFT Code," *Proceedings of the EuroPar Conference on Parallel and Distributed Computing,* 2003, pp. 251–260.
4. S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber, "FFT Compiler Techniques," *Proceedings of the 13th International Conference on Compiler Construction*, 2004, pp. 217–231.
5. "IBM Surges Past HP To Lead in Global Supercomputing," Armonk, NY, June 20, 2004. See *http://domino.research.ibm.com/comm/pr.nsf/pages/news.20040620_bluegene.html*.
6. M. Frigo and S. G. Johnson, "Fftw: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 1998, pp. 1381–1384.
7. M. Frigo and S. G. Johnson, "The Design and Implementation of Fftw 3," *Proc. IEEE* **93**, No. 2, special issue on "Program Generation, Optimization, and Platform Adaptation" (2005).
8. R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimizations of Software and the Atlas Project," *Parallel Computing* **27**, No. 1/2, 3–35 (2001).
9. C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Frontiers in Applied Mathematics Series, Vol. 10, Society for Industrial and Applied Mathematics Press, Philadelphia, 1992; ISBN: 0–89871-285–8.
10. M. Frigo, "A Fast Fourier Transform Compiler," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999, pp. 169–180.
11. F. Franchetti, "A Portable Short Vector Version of Fftw," *Proceedings of the 4th IMACS Symposium on Mathematical Modeling (MATHMOD)*, 2003, pp. 1539–1548.
12. F. Franchetti, "Performance Portable Short Vector Transforms," Ph.D. thesis, Institute for Analysis and Scientific Computing, Vienna University of Technology, Austria, 2003.
13. F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber, "Architecture Independent Short Vector FFTs," *Proceedings of the International Conference an Acoustics, Speech, and Signal Processing (ICASSP)*, 2001, pp. 1109–1112.
14. F. Franchetti and M. Püschel, "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002, pp. 20–26.
15. F. Franchetti and M. Püschel, "Short Vector Code Generation and Adaptation for DSP Algorithms," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2003, 537–540.
16. F. Franchetti and M. Püschel, "Short Vector Code Generation for the Discrete Fourier Transform," *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003, pp. 58–67.
17. S. Kral, F. Franchetti, J. Lorenz, M. Püschel, C. W. Ueberhuber, and P. Wurzinger, "Automatically Optimized FFT Codes for the BlueGene/L Supercomputer," *Proceedings of VecPar'04, 6th International Conference on High Performance Computing for Computational Sciences*, 2004, pp. 233–246.
18. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proc. IEEE* **93**, No. 2, special issue on "Program Generation, Optimization, and Platform Adaptation" (2005).
19. M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *Int. J. High Performance Computing Appl.* **18**, No. 1, 21–45 (2004).
20. R. J. Fisher and H. G. Dietz, "Compiling for SIMD Within a Register," *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1998, pp. 290–304.
21. S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 145–156.
22. R. Leupers and S. Bashford, "Graph-Based Code Selection Techniques for Embedded Processors," *ACM Trans. Design Automation Electron. Syst. (TODAES)* **5**, No. 4, 794–814 (2000).
23. J. Lorenz, "Automatic SIMD Vectorization," Ph.D. thesis, Institute for Analysis and Scientific Computing, Vienna University of Technology, Austria, 2004.

Received June 17, 2004; accepted for publication August 23, 2004; Internet publication March 31, 2005

Juergen Lorenz *Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10/101, A-1040 Vienna, Austria (juergen.lorenz@aurora.anum.tuwien.ac.at)*. Dr. Lorenz received Dipl.-Ing. and Ph.D. degrees in computer science from the Vienna University of Technology in 2002 and 2004, respectively. His research interests include parallel programming and special-purpose compilers. Dr. Lorenz is currently a Research Associate at the Vienna University of Technology.

Stefan Kral *Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10/101, A-1040 Vienna, Austria (skral@mips.complang.tuwien.ac.at)*. Mr. Kral received a Dipl.-Ing. degree in computer science from the Vienna University of Technology in 2004. His research interests include logic programming and compiler back ends. Mr. Kral is currently a Research Associate at the Vienna University of Technology.

Franz Franchetti *Electrical and Computer Engineering Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213 (franzf@ece.cmu.edu)*. Dr. Franchetti received Dipl.-Ing. and Ph.D. degrees in technical mathematics from the Vienna University of Technology in 2000 and 2003, respectively. He has been with the Vienna University of Technology since 1997. His research interests are concentrated on the development of high-performance digital signal processing (DSP) algorithms. Dr. Franchetti is currently a Research Associate at Carnegie Mellon University.

Christoph W. Ueberhuber *Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10/101, A-1040 Vienna, Austria (c.ueberhuber@tuwien.ac.at)*. Dr. Ueberhuber received Dipl.-Ing. and Ph.D. degrees in technical mathematics from the Vienna University of Technology in 1973 and 1976, respectively, and the *venia docendi habilitation* for numerical mathematics in 1979. He has been with the Vienna University of Technology since 1973 and is currently a professor of numerical mathematics. His research interests include numerical analysis, high-performance numerical computing, and advanced scientific computing. Dr. Ueberhuber has published 16 books and more than 100 publications in journals, books, and conference proceedings.