Automatically Optimized FFT Codes for the BlueGene/L Supercomputer

Franz Franchetti[†], Stefan Kral[†], Juergen Lorenz[†], Markus Püschel[‡], Christoph W. Ueberhuber[†], and Peter Wurzinger[†]

[†] Institute for Analysis and Scientific Computing,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
franz.franchetti@tuwien.ac.at,
WWW home page: http://www.math.tuwien.ac.at/ascot

[‡] Dept. of Electrical and Computer Engineering,
Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213
pueschel@ece.cmu.edu,
WWW home page: http://www.ece.cmu.edu/~pueschel

Abstract. IBM's upcoming 360 Tflop/s supercomputer BlueGene/L featuring 65,536 processors is supposed to lead the Top 500 list when being installed in 2005. This paper presents one of the first numerical codes actually run on a small prototype of this machine.
Formal vectorization techniques, the Vienna MAP vectorizer (both developed for generic short vector SIMD extensions), and the automatic performance tuning approach provided by Spiral are combined to generate automatically optimized FFT codes for the BlueGene/L machine targeting its two-way short vector SIMD "double" floating-point unit.
The resulting FFT codes are 40 % faster than the best scalar Spiral generated code and 5 times faster than the mixed-radix FFT implementation provided by the Gnu scientific library GSL.

1 Introduction

IBM's BlueGene/L [3] planned to be in operation in 2005 will be an order of magnitude faster than the Earth Simulator, currently being the number one on the Top 500 list. It will feature eight times more processors than current massively parallel systems, providing 64k processors for solving new classes of problems. To tame this vast parallelism, new approaches and tools have to be developed. However, tuning software for this machine starts by optimizing computational kernels for its processors. And BlueGene/L comes with a twist on this level as well. It's processors will feature a custom floating-point unit—called "double" FPU—that provides support for complex arithmetic.

Efficient computation of fast Fourier transforms (FFTs) is required in many applications planned to be run on BlueGene/L. In most of these applications,

very fast one-dimensional FFT routines for small problem sizes (up to 2048 data points) running on a single processor are required as major building blocks for large scientific codes. In contrast to tedious hand-optimization, the library generator Spiral [39] as well as state-of-the-art FFT libraries like Fftw [19] and Uhfft [34] use an empirical approach to automatically optimize code for a given platform.

While floating-point support for complex arithmetic is a important tool to speed up large scientific codes, the utilization of non-standard FPUs in computational kernels like FFTs is not straight-forward. Optimization of these kernels leads to complicated data dependencies of real variables that cannot be mapped to BlueGene/L's complex FPU. This is especially true when applying automatic performance tuning techniques and needs to be addressed to obtain high-performance FFT implementations.

This paper introduces an FFT library which is the first numerical code not developed by IBM run on a BlueGene/L prototype. The FFT library takes full advantage of BlueGene/L's double FPU by means of shot vector SIMD vectorization. It was generated and tuned automatically by connecting Spiral to special purpose vectorization technology [27, 28, 29]. The resulting codes provide FFTs running five times as fast as industry-standard FFT non-adaptive libraries. 40 % speed-up over the best code not using the double FPU is achieved where IBM's vectorizing compiler produces 15 % slow-down.

2 The BlueGene/L Supercomputer

An initial small-scale prototype of IBM's future supercomputer BlueGene/L [3], equipped with just 1024 of the custom-made IBM PowerPC 440 FP2 processors (512 two-way SMP chips) achieved a Linpack performance of $R_{max} = 1.44$ Tflop/s, i. e., 70 % of its theoretical peak performance of 2.05 Tflop/s. This performance ranks the prototype machine already on position 73 of the Top 500 list (November 2003). The Blue Gene/L prototype machine is roughly 1/20th the physical size of machines of comparable compute power that exist today—such as Linux clusters.

The full BlueGene/L machine, which is being built for the Lawrence Livermore National Laboratory (LLNL) in California, will be 128 times larger, occupying 64 full racks. When completed in 2005, the Blue Gene/L supercomputer is expected to lead the Top 500 list. Compared with today's fastest supercomputers, it will be an order of magnitude faster, consume 1/15th of the power and be 10 times more compact than today's fastest supercomputers.

The BlueGene/L machine at LLNL will be built from 65,536 PowerPC 440 FP2 processors connected by a 3D torus network leading to 360 Tflop/s peak performance. The Earth Simulator, currently leading the Top 500 list, achieves 40 Tflop/s peak performance. BlueGene/L's processors will run at 700 MHz, whereas the current prototype runs at 500 MHz.

2.1 BlueGene/L's Floating-Point Unit

There are many areas of scientific computing like computational electronics where complex arithmetic plays an important role in various algorithms. Thus, it makes sense to integrate native complex arithmetic support into the FPU of computers that are mainly devoted such applications.

This new SIMOMD style (Single Instruction Multiple Operation Multiple Data) ISA extension can be regarded as either a complex FPU or a real 2-way vector FPU, depending on the techniques used for utilizing the relevant hardware features.

Programs using complex arithmetic can be mapped to BlueGene/L's custom FPU in a straight forward manner. Problems arise when the usage of *real* code is unavoidable. Even for purely complex code it may be necessary to express complex arithmetics in terms of real arithmetic. In particular switching to real code allows to apply common subexpression elimination, constant folding and copy propagation on the real and imaginary parts. For codes as DSP transforms this is required to obtain high performance.

As the extension includes all classical short vector SIMD style (inter-operand, parallel) instructions (i. e., as supported by Intel SSE2), it can also be used to accelerate real computations if the algorithm allows for enough parallelism to be extracted. As a consequence, real codes have to be vectorized as well.

BlueGene/L's floating-point "double" FPU was obtained by replicating the PowerPC 440's standard FPU and adding crossover data paths and sign change capabilities to support complex multiplication leading to the PowerPC 440 FP2. Up to four real floating-point operations (one two-way vector fused multiply-add operation) can be issued every cycle. This double FPU has many similarities to industry-standard two-way short vector SIMD extensions like AMD's 3DNow! or Intel's SSE 2. In particular, data to be processed by the double FPU has to be naturally alignend on 16-byte boundaries in memory.

However, the PowerPC 440 FP2 features some characteristics that are different from standard short vector SIMD implementations:

(*i*) Non-standard fused multiply-add (FMA) operations required for complex multiplications, (*ii*) computationally expensive data reorganization within two-way registers, and (*iii*) cheap intermix of scalar and vector operations.

The main problem in the context of presently available vectorization techniques is that a single data reorder operation within a short vector SIMD register is as expensive as an arithmetic 2-way FMA operation. In addition, every cycle either a floating-point operation *or* a data reorganization instruction can be issued. Thus, without tailor-made adaptation of established short vector SIMD vectorization techniques to the specific features of BlueGene/L's double FPU no high-performance short vector code can be obtained.

2.2 Utilizing the Double FPU in Programs

To utilize BlueGene/L's double FPU within a numerical library, three approaches can be pursued: (*i*) Implement the numerical kernels in C utilizing proprietary

directives such that IBM's VisualAge XL C compiler for BlueGene/L is able to vectorize these kernels, (*ii*) rewrite the numerical kernels in assembly language using the double FPU instructions, or (*iii*) rewrite the numerical kernels utilizing XL C's language extension to C99 that provides access to the double FPU on source level by means of data types and intrinsic functions.

The GNU C compiler port for BlueGene/L supports the utilization of not more than 32 temporary variables when accessing the double FPU. This constraint prevents automatic performance tuning on BlueGene/L using the GNU C compiler.

This paper describes how vector code can be generated following the third approach utilizing the XL C compiler's vector data types and intrinsic functions to access the double FPU. Thus, register allocation and instruction scheduling is left to the compiler while vectorization and instruction selection is done on source code level by the newly developed approach.

3 Automatic Tuning of DSP Software

In the field of scientific computing, digital signal processing (DSP) transforms, including the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), and the family of discrete sine and cosine transforms (DSTs, DCTs) are—despite numerical linear algebra algorithms—core algorithms of almost any computationally intensive software. Thus, the applications of DSP transforms range from small scale problems with stringent time constraints (for instance, in real time signal processing) up to large scale simulations and PDE programs running on the world's largest supercomputers. Therefore, high-performance software tailor-made for these applications is desperately needed.

All the transforms mentioned above are structurally complex, leading to complicated algorithms that are difficult to map onto standard hardware efficiently.

The traditional method for achieving highly optimized numerical code is hand coding in assembly language. Beside the fact that this approach requires a lot of expertise, its major drawback is that the resulting code is error prone and non portable. Thus, hand coding is an infeasible approach to performance portable software. Recently, a new software paradigm emerged in which optimized code for numerical computation is generated automatically. New standards were set by ATLAS in the field of numerical linear algebra and FFTW which introduced automatic performance tuning in FFT libraries. In the field of digital signal processing (DSP), SPIRAL is providing automatically tuned codes for large classes of DSP transforms by utilizing state-of-the-art coding and optimization techniques.

These software packages use code generators to produce code which cannot be structurally compared to hand written code. The code consists of up to thousands of lines of code in single static assignment style.

Nevertheless, the codes generated by ATLAS, SPIRAL and FFTW are translated using standard compilers enabling portability but achieving satisfactorily high performance in connection with this type of numerical code is impossible.

For top performance in connection with such codes, the exploitation of special processor features such as short vector SIMD or FMA instruction set architecture extensions is a must.

Unfortunately, approaches used by vectorizing compilers to vectorize loops or basic blocks lead to inefficient code when applied to automatically generated codes for DSP transforms. The vectorization techniques entail large overhead for data reordering operations on the resulting short vector code as they do not have domain specific knowledge about the codes' inherent parallelism.

3.1 SPIRAL

Spiral [39] is a generator for high performance code for discrete linear transforms like the discrete Fourier transform (DFT), the discrete cosine transforms (DCTs), and many others. Spiral uses a mathematical approach that commutes the implementation problem of discrete linear transforms to a search problem in the space of structurally different algorithms and their possible implementations to generate code that is adapted to a given computing platform.

Spiral's approach is to represent the multitude of different algorithms for a signal transform as formulas in a concise mathematical language based on the Kronecker product formalism. Spiral utilizes the signal processing language SPL to represent Kronecker product formulas in a high-level computer language. These formulas expressed in SPL are automatically generated by the formula generator and automatically translated into code by Spiral's special purpose SPL compiler, thus enabling automated search. See Fig. 1 for an overview.

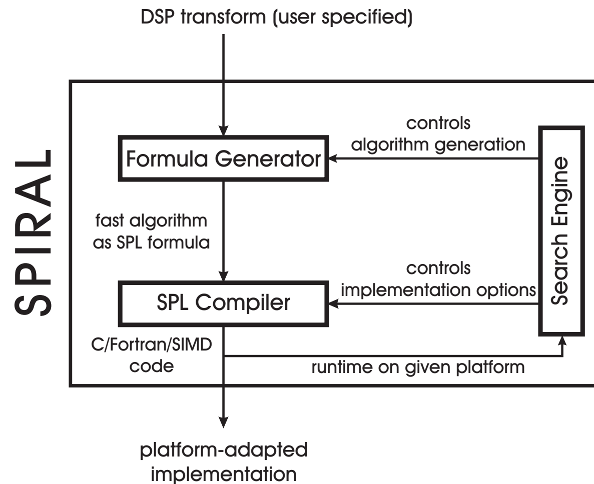

Fig. 1. Spiral's Architecture.

4 Generating Vector Code for BlueGene/L

In the following two approaches are presented, how to automatically vectorize FFT code generated by SPIRAL: (*i*) Formal vectorization of FFT algorithms, and (*ii*) vectorization of basic blocks. Both techniques are adapted to BlueGene/L's specifics and connected to SPIRAL's search capabilities leading to automatically tuned FFT implementations utilizing BlueGene/L's double FPU.

4.1 Formal Vectorization

The formal vectorization approach [12, 13, 14, 15, 16, 17] pursued to generate vector code for BlueGene/L is based on the following ideas:

- Target BlueGene/L's double FPU by porting generic formal vectorization techniques developed for classical short vector SIMD extensions like Intel's SSE family, AMD's 3DNow! family, and Motorola's AltiVec.
- Utilize an experimental short vector SIMD enabled SPIRAL version to automatically optimize code for BlueGene/L.

Formal vectorization is based on the SIMD vectorizing version of SPIRAL's SPL compiler [15]. This vectorizing DSP code generator was adapted to vectorize FFT code targeting BlueGene/L's double FPU.

Mapping DSP Code to Vector Code. Certain mathematical constructs used in SPIRAL's formula representation of a DSP algorithm can be mapped to vectorized code by SPIRAL's compiler. These vectorizable constructs occur in virtually every DSP algorithm. Furthermore, in several important cases, including the FFT, the formulas generated by SPIRAL are built exclusively from these constructs, and thus can be completely vectorized using the SPL compiler adapted to vectorizing SIMD.

In addition, a formally derived *short vector FFT variant* is utilized to generate vectorized FFT code [17]. This variant guarantees, that the automatically generated FFT code fits to the need of generic short vector SIMD extensions. The short vector FFT variant was adapted to support BlueGene/L's double FPU.

Utilizing both methods of formal vectorization, FFT formulas generated by SPIRAL's formula generator are translated into vector code utilizing BlueGene/L's double FPU efficiently.

Vectorization for Short Vector SIMD Extensions. Our method originates from the observation that neither original vector computer DSP transform algorithms [30] nor vectorizing compilers [23, 31] are capable of producing high-performance DSP transform implementations for short vector SIMD architectures, even in tandem with automatic performance tuning [17].

The intrinsic structure of DSP transforms prevents the successful application of these well-known methods. The main obstacle is that the structure of memory

access (given by permutations and loop carried array references) occurring in DSP transform algorithms is incompatible with the features offered by currently available short vector SIMD target architectures.

Our vectorization method is at the border of loop vectorization and the utilization of instruction-level parallelism. Although actually loops are vectorized, all vectorized loops feature the unusual small number of exactly ν iterations where ν is the machine's vector length ($\nu = 2$ for BlueGene/L as complex numbers are viewed as 2-vectors). This approach allows to support both the recursive nature of DSP algorithms as well as generic SIMD extensions (with arbitrary vector lengths ν) at the same time. Thus, it overcomes the limitations of classical loop vectorization and extraction of instruction-level parallelism in this particular field.

4.2 The Vienna MAP Vectorizer

This section introduces the Vienna MAP vectorizer [27, 28, 29] that automatically extracts 2-way SIMD parallelism out of given numerical straight-line code. A peephole optimizer directly following the MAP vectorizer additionally supports the extraction of SIMD fused multiply-add instructions.

Fundamentals of Vectorization. Existing approaches to vectorizing basic blocks [11, 31, 32] try to find an efficient mix of SIMD and scalar instructions to do the required computation, MAP's vectorization mandates that *all* computation is performed by SIMD instructions, while attempting to keep the SIMD reordering overhead reasonably small.

Some of this approaches have to introduce more SIMD data reordering instructions than necessary, as they cannot resort to a representation of the given numerical scalar DAG as vectorization input, where the inherent parallelism is obvious.

Such an approach fails, as already mentioned in Section 2.1, as SIMD data reordering operations are very expensive on IBM BlueGene/L's Double FPU.

MAP's vectorizer uses depth-first search with chronological backtracking to discover SIMD style parallelism in a scalar code block, aiming at a complete coverage of the scalar DAG by natively supported SIMD instructions. The goal is a satisfactory SIMD utilization which is tantamount to minimizing the amount of SIMD data reorganization and to maximizing the coverage.

The Vectorization Algorithm. The central goal of vectorization is to replace all scalar instructions by vector instructions. The source codes to be vectorized may contain array accesses, index computation, and arithmetic operations.

To automatically extract 2-way short vector SIMD parallelism out of scalar code blocks, pairs of scalar variables `s`, `t` are combined, i. e., *fused*, to form a SIMD variable `st = (s,t)` or `ts = (t,s)`. The arithmetic instructions operating on `s` and `t` are combined, i. e., *paired*, to a sequence of SIMD instructions, as depicted in Table 1.

This sequence must not include SIMD reorder instructions except for the unavoidable case when a fusion `(s,t)` is required, but its swapped counterpart `(t,s)`, i. e., a compatible fusion, is present. A swap operation is required to use `(t,s)` whenever `(s,t)` is used, in this case.

Swap operations are allowed, as they can be removed in a peephole optimization step directly following the vectorization process.

Before the actual vectorization process is carried out, the following preparatory steps are taken. First, dependencies of the scalar DAG are analyzed and instruction statistics are assembled. This data is used to speed up the vectorization process by avoiding futile vectorization. Then, store instructions are combined non-deterministically by fusing their respective source operands.

The actual vectorization algorithm consists of two steps.

(*i*) Pick `I1 = (op1,s1,t1,d1)` and `I2 = (op2,s2,t2,d2)`, two scalar instructions that have not been vectorized, with `(d1,d2)` or `(d2,d1)` being an existing fusion.

(*ii*) Non-deterministically pair the two scalar operations `op1` and `op2` into one SIMD operation. This step may produce new fusions or may required a compatible fusion for the respective source operands.

The vectorizer alternatingly applies these two steps 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 commits to the first solution of the search process.

Non-determinism in vectorization arises due to different vectorization choices. For a fusion `(d1,d2)` there may be several ways of fusing the source operands `s1,t1,s2,t2`, depending on the pairing `(op1,op2)`, as depicted in Fig. 1.

Peephole Based Optimization. After vectorization, a local rewriting system is used to implement peephole optimization on the vector DAG.

The first group of rewriting rules aims at (*i*) minimizing the number of instructions, (*ii*) eliminating redundancies and dead code, (*iii*) reducing the number of source operands, (*iv*) copy propagation, and (*v*) constant folding.

The second group of rules can be used to extract SIMD-FMA instructions.

The third group of rules rewrite unsupported SIMD instructions into sequences of SIMD instructions actually supported by the target architecture.

MAP was adapted to support BlueGene/L's double FPU and connected to the Spiral system to provide BlueGene/L specific vectorization of Spiral generated code. Spiral's SPL compiler was used to translate formulas generated by the formula generator into fully unrolled implementations leading to large straight-line codes. These codes were subsequently vectorized by MAP and thus transformed into a C program where all arithmetic operations and memory access operations are expressed using XL C's intrinsic functions. Finally the codes were compiled utilizing the XL C compiler with vectorization turned off, leaving register allocation, instruction scheduling and other low-level backend optimization to the XL C compiler.

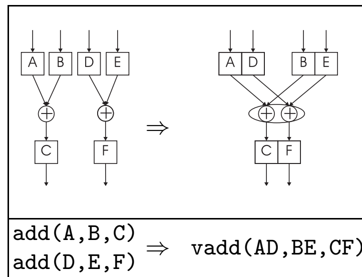

Table 1. Two-way Vectorization. Two scalar `add` instructions are transformed into one vector `vadd` instruction.

5 Experimental Results

The presented vectorization techniques were evaluated on an early BlueGene/L prototype. Performance data of 1D FFTs with vector lengths $N = 2^2, 2^3, \ldots, 2^{10}$ were obtained on a single PowerPC 440 FP2 running at 500 MHz.

In particular the following FFT implementations were tested: (*i*) The best vectorized code found by Spiral utilizing formal vectorization, (*ii*) the best vectorized code found by Spiral utilizing the Vienna MAP vectorizer, (*iii*) the best scalar FFT implementation found by Spiral (XL C's vectorizer and FMA extraction turned off), (*iv*) the best vectorized FFT implementation found by Spiral using the XL C compiler's vectorizer and FMA extraction turned on, and (*v*) the mixed-radix FFT implementation provided by the Gnu scientific library (GSL). Fig. 2 and 3 display the respective performance data.

The best scalar codes found by Spiral—referenced by (*iii*) in Fig. 2—serve as baseline for the assessment of the various vectorization techniques. These codes are very fast scalar implementations featuring no FMA instructions.

Formal vectorization—referenced by (*i*) in Fig. 2—provides up to 40 % speed-up w.r.t. the best scalar codes generated by Spiral for problem sizes $N \geq 64$. Thus formal vectorization provides significant speed-up for larger problem sizes.

The Vienna MAP vectorizer—referenced by (*ii*) in Fig. 2—is restricted to problem sizes that can fully be unrolled fitting into instruction cache and the resulting code can be handled well by the XL C compiler's register allocator. For problem sizes $N \leq 32$, the Vienna MAP vectorizer provides the same level of performance as formal vectorization for larger problem sizes.

The third-party Gnu GSL FFT library—referenced by (*v*) in Fig. 2—reaches about 30 % of the performance of the best scalar Spiral generated code thus performing badly.

XL C's vectorization and FMA extraction—referenced by (*iv*) in Fig. 2—produces code 15 % slower than scalar XL C without FMA extraction. Thus, the vectorization techniques to vectorize straight-line code currently used within the XL C compiler cannot handle Spiral generated FFT codes well.

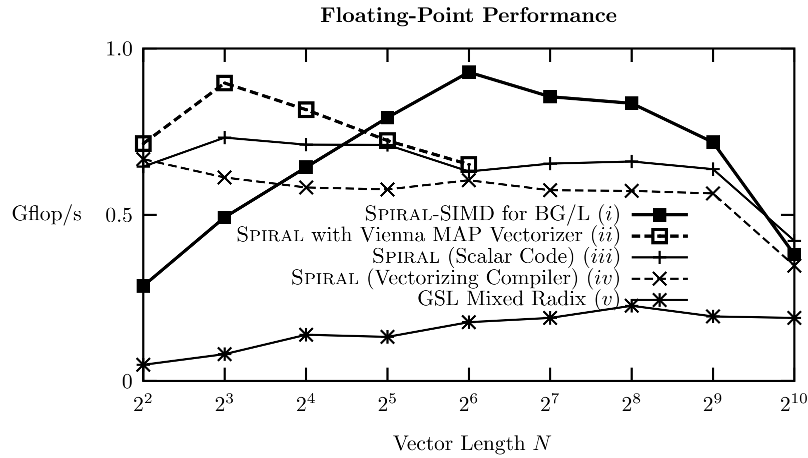

Fig. 2. Performance of the vectorization techniques applied by MAP vectorizer and formal vectorization (SPIRAL-SIMD for BG/L) compared to the best scalar code and the best vectorized code (utilizing the VisualAge XLC for BG/L vectorizing compiler) found by SPIRAL. Performance is displayed in *pseudo Gflop/s* ($5N \log N$/runtime with N being the vector length).

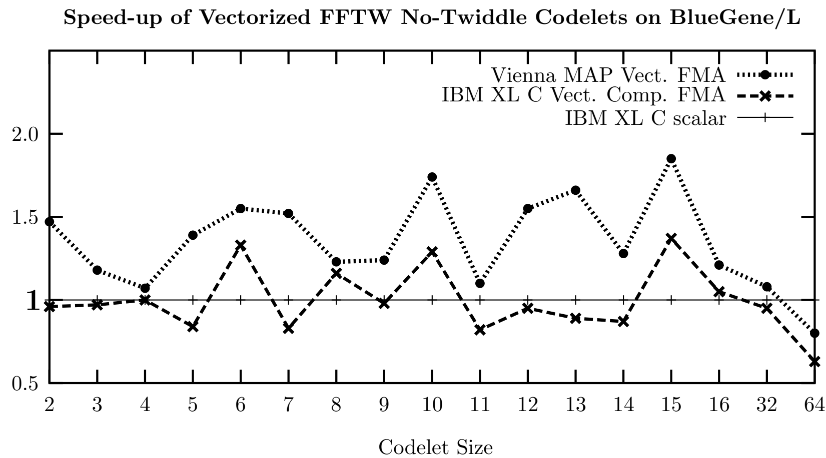

Fig. 3. Speed-up of the vectorization techniques applied by the MAP vectorizer compared to scalar code and code vectorized by IBM's VisualAge XL C compiler.

6 Conclusions and Outlook

As FFTs are indispensably required as integral parts of applications in practically all fields of scientific computing, they are urgently needed by BlueGene/L's scientific users. The performance portable vectorization techniques introduced in this paper allow timely software optimization concurrently done with IBM BlueGene/L's hardware development. Besides the formal vectorization techniques, the highly portable Vienna MAP vectorizer ca be used to automatically vectorize numerical straight line code generated with advanced automatic performance tuning software such as Spiral helping to develop high performance implementations of FFT kernels.

Performance experiments carried out on a BlueGene/L prototype show that automatic performance tuning in combination with the two newly developed vectorization approaches is able to speed up FFT code considerably, while vectorization by IBM's XL C compiler does not speed up the automatically generated scalar codes at all. The two vectorization approaches of this paper are able to provide high-performance FFT kernels for the BlueGene/L supercomputer by fully utilizing the new double FPU.

Nevertheless, even better performance results can be yielded by improving the current BlueGene/L version of the Vienna MAP vectorizer. An integral part of the future work will be to fully fold any SIMD data reorganization into SIMD fused multiply add instructions. Besides, a compiler backend is in development which uses a register allocation better suited for numerical straight-line code than that of IBM's XL C compiler.

Acknowledgements. 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 BlueGene/L prototype and for a very pleasant and fruitful cooperation.

The Center for Applied Scientific Computing at Lawrence Livermore National Laboratory (LLNL) in California deserves particular appreciation for ongoing support.

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

References

[1] D. Aberdeen and J. Baxter, "EMMERALD: a fast matrix-matrix multiply using Intel's SSE instructions," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 2, pp. 103–119, 2001.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] 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.

[4] *AMD Core Math Library (ACML) Manual*, Advanced Micro Devices Corporation, 2000.

[5] ANSI, "ISO/IEC 9899:1999(E), Programming Languages – C," American National Standard Institute (ANSI), New York, 1999.

[6] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, vol. 5, no. 2, 1966.

[7] J. Bilmes, K. Asanovic, C. W. Chin, J. Demmel, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Proceedings of the International Conference on Supercomputing, ACM, Vienna, Austria, pp. 340–347, 1997.

[8] R. Crandall and J. Klivington, "Supercomputer-style FFT library for the Apple G4," Advanced Computation Group, Apple Computer Inc., 2002.

[9] J. Demmel, J. Dongarra, V. Eijkhout, and K. Yelick, "Automatic performance tuning for large scale scientific applications." *IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2003.

[10] R. J. Fisher and H. G. Dietz, "The SCC Compiler: SWARing at MMX and 3DNow," in *12th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC99)*, 1999.

[11] ——, "Compiling for SIMD within a register," in *Languages and Compilers for Parallel Computing*, pp. 290–304, 1998. [Online]. Available: `citeseer.ist.psu.edu/fisher98compiling.html`

[12] F. Franchetti, "A portable short vector version of FFTW," in *Proc. Fourth IMACS Symposium on Mathematical Modelling (MATHMOD 2003)*, vol. 2, pp. 1539–1548, 2003.

[13] ——, "Performance portable short vector transforms," Ph. D. Thesis, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.

[14] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber, "Architecture independent short vector FFTs," in *Proc. ICASSP*, vol. 2, pp. 1109–1112, 2001.

[15] F. Franchetti and M. Püschel, "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," in *Proc. IPDPS*, pp. 20–26, 2002.

[16] ——, "Short vector code generation and adaptation for DSP algorithms." in *Proceedings of the International Conerence on Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP'03)*, vol. 2, pp. 537–540, 2003.

[17] ——, "Short vector code generation for the discrete Fourier transform." in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, pp. 58–67, 2003.

[18] M. Frigo, "A fast Fourier transform compiler," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation.* New York: ACM Press, pp. 169–180, 1999.

[19] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," in *ICASSP 98*, vol. 3, pp. 1381–1384, 1998, http://www.fftw.org

[20] ——, "The design and implementation of FFTW," *IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2003.

[21] J. Guo, M. Garzarán, and D. Padua, "The power of Belady's algorithm in register allocation for long basic blocks," *Proceedings of the LCPC*, 2003.

[22] Intel Corporation, "AP-808 split radix fast Fourier transform using streaming SIMD extensions," 1999.

[23] ——, "Intel C/C++ compiler user's guide," 2002.

[24] ——, "Math kernel library," 2002. [Online]. Available: http://www.intel.com/software/products/mkl

[25] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *IEEE Trans. on Circuits and Systems*, vol. 9, pp. 449–500, 1990.

[26] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," Digital Western Research Laboratory Palo Alto, California, WRL Research Report 7, 1989.

[27] S. Kral, F. Franchetti, J. Lorenz, and C. Ueberhuber, "SIMD vectorization of straight line FFT code," Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, pp. 251–260, 2003.

[28] ——, "FFT compiler techniques," Proceedings of the 13th International Conference on Compiler Construction LNCS 2790, pp. 217–231, 2004.

[29] ——, "Efficient Utilziation of SIMD Extensions," to appear in IEEE Proceedings Special Issue on Program Generation, Optimization, and Platform Adaption

[30] S. Lamson, "SCIPORT," 1995. [Online]. Available: http://www.netlib.org/scilib/

[31] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 145–156, 2000.

[32] R. Leupers and S. Bashford, "Graph-based code selection techniques for embedded processors," *ACM Transactions on Design Automation of Electronic Systems.*, vol. 5, no. 4, pp. 794–814, 2000. [Online]. http://citeseer.nj.nec.com/leupers00graph.html

[33] M. Lorenz, L. Wehmeyer, and T. Draeger, "Energy aware compilation for DSPs with SIMD instructions," *Proceedings of the 2002 Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software*

and Compilers for Embedded Systems (LCTES'02-SCOPES'02)., pp. 94–101, 2002. [Online]. http://citeseer.ist.psu.edu/lorenz02energy.html

[34] D. Mirkovic and S. L. Johnsson, "Automatic Performance Tuning in the UHFFT Library," in *Proc. ICCS 01*, pp. 71–80, 2001.

[35] J. M. F. Moura, J. Johnson, D. Padua, M. Püschel, and M. Veloso, "Spiral." *IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2003.

[36] S. S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, 1997.

[37] K. Nadehara, T. Miyazaki, and I. Kuroda, "Radix-4 FFT implementation using SIMD multi-media instructions," in *Proc. ICASSP 99*, pp. 2131–2135, 1999.

[38] I. Nicholson, "libSIMD," 2002. [Online]. Available: http://libsimd.sourceforge.net

[39] 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," *Journal on High Performance Computing and Applications*, special issue on Automatic Performance Tuning, Vol. 18, pp. 21–45, 2004, http://www.spiral.net

[40] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363–400, 2000.

[41] Y. Srikant and P. Shankar, *The Compiler Design Handbook.* Boca Raton London New York Washington D.C.: CRC Press LLC, 2003.

[42] P. N. Swarztrauber, "FFT algorithms for vector computers," *Parallel Comput.*, vol. 1, pp. 45–63, 1984.

[43] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, ser. Frontiers in Applied Mathematics. Philadelphia: Society for Industrial and Applied Mathematics, 1992, vol. 10.

[44] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Comput.*, vol. 27, pp. 3–35, 2001, http://math-atlas.sourceforge.net

[45] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A Language and Compiler for DSP Algorithms," in *Proceedings of the Conference on Programming Languages Design and Implementation* (PLDI), pp. 298–308, 2001.

[46] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York: ACM Press, 1991.