Automatically Tuned FFTs for BlueGene/L's Double FPU

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

[†] 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 is currently developing the new line of BlueGene/L supercomputers. The top-of-the-line installation is planned to be a 65,536 processors system featuring a peak performance of 360 Tflop/s. This system is supposed to lead the Top 500 list when being installed in 2005 at the Lawrence Livermore National Laboratory. This paper presents one of the first numerical kernels run on a prototype BlueGene/L machine. We tuned our formal vectorization approach as well as the Vienna MAP vectorizer to support BlueGene/L's custom two-way short vector SIMD "double" floating-point unit and connected the resulting methods to the automatic performance tuning systems Spiral and Fftw. Our approach produces automatically tuned high-performance FFT kernels for BlueGene/L that are up to 45 % faster than the best scalar Spiral generated code and up to 75 % faster than Fftw when run on a single BlueGene/L processor.

1 Introduction

IBM's BlueGene/L supercomputers [1] are a new class of massively parallel systems that focus not only on performance but also on lower power consumption, smaller footprint, and lower cost compared to current supercomputer systems. Although several computing centers plan to install smaller versions of BlueGene/L, the most impressive system will be the originally proposed system at Lawrence Livermore National Laboratory (LLNL), planned to be in operation in 2005. It will be an order of magnitude faster than the Earth Simulator, which currently is the number one on the Top 500 list. This system 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 systems come with a twist on this level as well. Their processors 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 systems. In most of these applications, very fast one-dimensional FFT routines for relatively small problem sizes (in the region of 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 [23], as well as the state-of-the-art FFT libraries Fftw [12] and Uhfft [22], use empirical approaches to automatically optimize code for a given platform.

Floating-point support for complex arithmetic can speed up large scientific codes significantly, but the utilization of non-standard FPUs in computational kernels like FFTs is not straightforward. Optimization of these kernels leads to complicated data dependencies that cannot be mapped directly to BlueGene/L's custom FPU. This complicacy becomes a governing factor when applying automatic performance tuning techniques and needs to be addressed to obtain high-performance FFT implementations.

Contributions of this Paper. In this paper we introduce and describe an FFT library and experimental FFT kernels (Fftw no-twiddle codelets) that take full advantage of BlueGene/L's double FPU by means of short vector SIMD vectorization. The library was automatically generated and tuned using Spiral in combination with formal vectorization [8] as well as by connecting Spiral to the Vienna MAP vectorizer [15, 16]. The resulting codes provide up to 45 % speed-up over the best Spiral generated code not utilizing the double FPU, and they are up to 75 % faster than off-the-shelf Fftw 2.1.5 ported to BlueGene/L without specific optimization for the double FPU.

We obtained Fftw no-twiddle codelets for BlueGene/L by connecting Fftw to the Vienna MAP vectorizer. The resulting *double FPU no-twiddle codelets* are considerably faster than their scalar counterparts and strongly encourage the adaptation of Fftw 2.1.5 to BlueGene/L [20]. Experiments show that our approach provides satisfactory speed-up while the vectorization facility of IBM's XL C compiler does not speed up FFT computations. Our FFT codes were the first numerical codes developed outside IBM that were run on the first BlueGene/L prototype systems DD1 and DD2.

2 The BlueGene/L Supercomputer

The currently largest prototype of IBM's supercomputer line BlueGene/L [1] is DD1, a machine equipped with 8192 custom-made IBM PowerPC 440 FP2 processors (4096 two-way SMP chips), which achieves a Linpack performance of $R_{max} = 11.68$ Tflop/s, i. e., 71 % of its theoretical peak performance of $R_{peak} =$

16.38 Tflop/s. This performance ranks the prototype machine on position 4 of the TOP 500 list (in June 2004). The BlueGene/L prototype machine is roughly 1/20th the physical size of machines of comparable compute power that exist today.

The largest BlueGene/L machine, which is being built for Lawrence Livermore National Laboratory (LLNL) in California, will be 8 times larger and will occupy 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. BlueGene/L's processors will run at 700 MHz, whereas the current prototype BlueGene/L DD1 runs at 500 MHz. As a comparison, the Earth Simulator which is currently leading the TOP 500 list achieves a LINPACK performance of $R_{max} = 36$ Tflop/s.

2.1 BlueGene/L's Floating-Point Unit

Dedicated hardware support for complex arithmetic has the potential to accelerate many applications in scientific computing.

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. The resulting PowerPC 440 CPU with its new custom FPU is called 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 3. In particular, data to be processed by the double FPU has to be alignend on 16-byte boundaries in memory.

However, the PowerPC 440 FP2 features some characteristics that are different from standard short vector SIMD architectures; namely (*i*) non-standard fused multiply-add (FMA) operations required for complex multiplications; (*ii*) computationally expensive data reorganization within two-way registers; and (*iii*) efficient support for mixing of scalar and vector operations.

Mapping Code to BlueGene/L's Double FPU. BlueGene/L's double FPU can be regarded either as a complex FPU or as a real two-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. However, even for purely complex code it may be beneficial to express complex arithmetics in terms of real arithmetic. In particular, switching to real arithmetic allows to apply common subexpression elimination, constant folding, and copy propagation on the real and imaginary parts of complex numbers separately. For FFT implementations this saves a significant number of arithmetic operations, leading to improved performance.

Since the double FPU supports all classical short vector SIMD style (inter-operand, parallel) instructions (e. g., as supported by Intel SSE 2), it can be used to accelerate real computations if the targeted algorithm exhibits fine-grain parallelism. Short vector SIMD vectorization techniques can be applied to speed up real codes.

The main challenge when vectorizing for the double FPU is that one data reorder operation within the two-way register file is as expensive as one arithmetic two-way FMA operation (i. e., *four* floating-point operations). In addition, every cycle either one floating-point two-way FMA operation *or* one data reorganization instruction can be issued. On other two-way short vector SIMD architectures like SSE 3 and 3DNow! data shuffle operations are much more efficient. Thus, the double FPU requires tailoring of vectorization techniques to these features in order to produce high-performance code.

Tools for Programming BlueGene/L's Double FPU. To utilize BlueGene/L's double FPU within a numerical library, three approaches can be pursued: (*i*) The implementation of the numerical kernels in C such that IBM's VisualAge XL C compiler for BlueGene/L is able to vectorize these kernels. (*ii*) Directly implement the numerical kernels in assembly language using double FPU instructions. (*iii*) Explicitly vectorize the numerical kernels utilizing XL C's proprietary 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 no more than 32 temporary variables when accessing the double FPU. This constraint prevents automatic performance tuning on BlueGene/L using the GNU C compiler.

In this paper, we provide vectorization techniques for FFT kernels tailored to the needs of BlueGene/L. All codes are generated automatically and utilize the XL C compiler's vector data types and intrinsic functions to access the double FPU to avoid utilizing assembly language. Thus, register allocation and instruction scheduling is left to the compiler, while vectorization (i. e., floating point SIMD instruction selection) is done at the source code level by our approach.

3 Automatic Performance Tuning of Numerical Software

Discrete Fourier transforms (DFTs) are, together with linear algebra algorithms, the most ubiquitously used kernels in scientific computing.

The applications of DFTs range from small scale problems with stringent time constraints (for instance, in real time signal processing) to large scale simulations and PDE programs running on the world's largest supercomputers. Therefore, the best possible performance of DFT software is of crucial importance. However, algorithms for DFTs have complicated structure, which makes their efficient implementation a difficult problem, even on standard platforms. It is an even harder problem to efficiently map these algorithms to special hardware like BlueGene/L's double FPU.

The traditional method for achieving highly optimized numerical code is hand coding, often in assembly language. However, this approach requires a lot of expertise and the resulting code is error-prone and non-portable.

Automatic Performance Tuning. Recently, a new paradigm for software creation has emerged: the automatic generation and optimization of numerical libraries. Examples include ATLAS [27] in the field of numerical linear algebra, and FFTW [12] which introduced automatic performance tuning in FFT libraries. In the field of digital signal processing (DSP), SPIRAL [23] automatically generates tuned codes for large classes of DSP transforms by utilizing state-of-the-art coding and optimization techniques. All these systems feature code generators that generate ANSI C code to maintain portability. The generated kernels consist of up to thousands of lines of code.

To achieve top performance in connection with such codes, the exploitation of special processor features such as short vector SIMD or FMA instruction set architecture extensions (like the double FPU) is a must. Unfortunately, approaches used by vectorizing compilers to vectorize loops [2, 3, 24, 29] or basic blocks [18, 19, 21] lead to inefficient code when applied to automatically generated codes for DSP transforms. In case of these algorithms, 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. The cost of this overhead becomes prohibitive on an architecture like BlueGene/L's double FPU where data reorganization is very expensive compared to floating-point operations. This implies that vectorization techniques have to be adapted to BlueGene/L's FPU architecture.

FFTW. FFTW implements the Cooley-Tukey FFT algorithm [26] recursively. This allows for flexible problem decomposition that is the key for FFTW's adaptability to a wide range of different computer systems.

In an initialization step FFTW's *planner* applies dynamic programming to find a problem decomposition that leads to fast FFT computation for a given size on the target machine. Whenever FFTs of planned sizes are to be computed, the *executor* applies these problem decompositions to perform the FFT computation. At the leafs of the recursion the actual work is done in code blocks called *codelets*. These codelets come in two flavors (twiddle codelets and no-twiddle codelets) and are automatically generated by a program generator named `genfft` [10, 11].

SPIRAL. SPIRAL [23] is a generator for high performance code for discrete signal transforms including the discrete Fourier transform (DFT), the discrete cosine transforms (DCTs), and many others. SPIRAL uses a mathematical approach that translates the implementation and optimization problem into a search problem in the space of structurally different algorithms and their possible implementations for the best match to a given computing platform.

SPIRAL represents the different algorithms for a signal transform as formulas in a concise mathematical language, called SPL, that is an extension of the Kronecker product formalism [14, 28]. The SPL formulas expressed in SPL are automatically generated by SPIRAL's *formula generator* and automatically

translated into code by SPIRAL's *special purpose SPL compiler*. Performance evaluation feeds back runtime information into SPIRAL's *search engine* which closes the feedback loop enabling automated search for good implementations.

4 Generating Vector Code for BlueGene/L

In this section we present two approaches to vectorizing FFT kernels generated by SPIRAL and FFTW targeted for BlueGene/L's double FPU.

- *Formal vectorization* exploits structural information about a given FFT algorithm and extends SPIRAL's formula generator and special-purpose SPL compiler.
- The *Vienna MAP vectorizer* extracts two-way parallelism out of straight-line code generated by SPIRAL or FFTW's codelet generator `genfft`.

Both methods are tailored to the specific requirements of BlueGene/L's double FPU. We explain the methods in the following.

4.1 Formal Vectorization

In previous work, we developed a formal vectorization approach [6] and applied it successfully across a wide range of short vector SIMD platforms for vector lengths of two and four both to FFTW [4, 5] and SPIRAL [7, 8, 9]. We showed that neither original vector computer FFT algorithms [17, 25] nor vectorizing compilers [13, 18] are capable of producing high-performance FFT implementations for short vector SIMD architectures, even in tandem with automatic performance tuning [9].

In this work we adapt the formal vectorization approach to the specific features of BlueGene/L's double FPU and the BlueGene/L system environment, and implement it within the SPIRAL system.

Short Vector Cooley Tukey FFT. Our DFT specific formal vectorization approach is the short vector Cooley-Tukey recursion [9]. For m and n being multiples of the machine's vector length ν, the short vector Cooley-Tukey FFT recursion translates a DFT_{mn} recursively into DFT_m and DFT_n such that *any* further decomposition of the smaller DFTs yields vector code for vector lengths ν. It adapts the Cooley-Tukey FFT recursion [26] to short vector SIMD architectures with memory hierarchies while resembling the Cooley-Tukey FFT recursion's data flow as close as possible to keep its favorable memory access structure.

In contrast to the standard Cooley-Tukey FFT recursion, all permutations in the short vector Cooley-Tukey FFT algorithm are either block permutations shuffling vectors of length ν or operate on $k\nu$ consecutive data items with k being small. For all current short vector SIMD architectures, the short vector Cooley-Tukey FFT algorithm can be implemented using *solely* vector memory access operations, vector arithmetic operations and data shuffling within vector registers.

Algorithm 1 (Short Vector Cooley-Tukey FFT)

```
ShortVectorFFT (mn, ν, y ← x)
    BlockStridePermutation(t0 ← y, mn/ν, m/ν, ν)
    for i = 0 ... m/ν − 1 do
      VectorFFT(ν × DFTn, t1 ← t0[iνn ... (i + 1)νn − 1])
      StridePermutationWithinBlocks(t2 ← t1, n/ν, ν^2, ν)
      BlockStridePermutation(t3[iνn ... (i + 1)νn − 1] ← t2, n, ν, ν)
    endfor
    ApplyTwiddleFactors(t4 ← t3)
    for i = 0 ... n/ν − 1 do
      VectorFFT(ν × DFTm, y[iνm ... (i + 1)νm − 1] ← t4[iνm ... (i + 1)νm − 1])
    endfor
```

Algorithm 1 describes the short vector Cooley-Tukey FFT algorithm in pseudo code. It computes a DFT_{mn} by recursively calling VectorFFT() to compute vectors of $\nu \times \mathrm{DFT}_m$ and $\nu \times \mathrm{DFT}_n$, respectively. BlockStridePermutation() applies a coarse-grain block stride permutation [1] to reorder blocks *of* ν elements. StridePermutationWithinBlocks() applies n/ν fine-grain stride permutations to shuffle elements *within* blocks of ν^2 elements. ApplyTwiddleFactors() multiplies a data vector by complex roots of unity called twiddle factors. In our implementation the functions BlockStridePermutation(), StridePermutationWithinBlocks(), and ApplyTwiddleFactors() are combined with the functions VectorFFT() to avoid multiple passes through the data.

We extended Spiral's formula generator and SPL compiler to implement the short vector Cooley-Tukey recursion. For a given DFT_{mn}, Spiral's search engine searches for the best choice of m and n as well as for the best decomposition of DFT_m and DFT_n.

Adaptation to BlueGene/L. The short vector Cooley-Tukey FFT algorithm was designed to support all short vector SIMD architectures currently available. It is built on top of a set of C macros called the portable SIMD API that abstracts the details of a given short vector SIMD architecture. To adapt Spiral's implementation of the short vector Cooley-Tukey FFT algorithm to BlueGene/L we had to (*i*) implement the portable SIMD API for BlueGene/L's double FPU (with $\nu = 2$), and (*ii*) utilize the double FPU's fused multiply-add instructions whenever possible.

Spiral's implementation of the portable SIMD API mandates the definition of vector memory operations and vector arithmetic operations. These operations were implemented using the intrinsic and C99 interface provided by IBM's XL C compiler for BlueGene/L. Fig. 1 shows a part of the portable SIMD API for BlueGene/L.

A detailed description of our formal vectorization method and its application to a wide range of short vector SIMD architectures can be found in [6, 7, 9].

[1] A stride permutations can be seen as a transposition of a rectangular two-dimensional array [14].

```
/* Vector arithmetic operations and declarations*/
#define VECT_ADD(c,a,b) (c) = __fpadd(a,b)
#define VECT_MSUB(d,a,b,c) (d) = __fpmsub(c,a,b)
#define DECLARE_VECT(vec) _Complex double vec

/* Data shuffling */
#define C99_TRANSPOSE(d1, d2, s1, s2) {\
   d1 = __cmplx (__creal(s1), __creal(s2));\
   d2 = __cmplx (__cimag(s1), __cimag(s2)); }

/* Memory access operations */
#define LOAD_VECT(trg, src) (trg) = __lfpd((double *)(src))

#define LOAD_INTERL_USTRIDE(re, im, in) {\
    DECLARE_VECT(in1); DECLARE_VECT(in2);\
    LOAD_VECT(in1, in); LOAD_VECT(in2, (in) + 2);\
    C99_TRANSPOSE(re, im, in1, in2); }
```

Fig. 1. Macros included in SPIRAL's implementation of the portable SIMD API for IBM's XL C compiler for BlueGene/L.

4.2 The Vienna MAP Vectorizer

This section introduces the Vienna MAP vectorizer [6, 15, 16] that automatically extracts two-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.

Vectorization of Straight-Line Code. Existing approaches to vectorizing basic blocks originate from very long instruction word (VLIW) or from SIMD digital signal processors (DSP) compiler research [3, 18, 19], and try to find either an efficient mix of SIMD and scalar instructions to do the required computation or insert data shuffling operations to allow for parallel computation. Due to the fact that SIMD data reordering operations are very expensive on IBM BlueGene/L's double FPU and FFT kernels have complicated data flow, these approaches are rendered suboptimal for the vectorization of FFT kernels for the double FPU.

MAP's vectorization requires that *all* computation is performed by SIMD instructions, while attempting to keep the SIMD reordering overhead reasonably small. The only explicit data shuffling operations allowed are swapping the two entries of a two-way vector register. In addition, MAP requires inter- and intra-operand arithmetic operations. On short vector extensions not supporting intra-operand arithmetic operations, additional shuffle operations may be required. When vectorizing complex FFT kernels for the double FPU, no data shuffle operations are required at all.

The Vectorization Algorithm. The MAP vectorizer uses depth-first search with chronological backtracking to discover SIMD style parallelism in a scalar code block. MAP's input is scalar straight-line codes that may contain array accesses, index computation, and arithmetic operations. Its output is vector straight-line code containing vector arithmetic and vector data shuffling operations, vector array accesses, and index computation. MAP describes its input as scalar directed acyclic graph (DAG) and its output as vector DAG.

In the vectorization process, pairs of scalar variables `s`, `t` are combined, i. e., *fused*, to form SIMD variables `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. This vectorization process translates nodes of the scalar DAG into nodes of the vector DAG. Table 1 shows an example of fusing two additions into a vector addition.

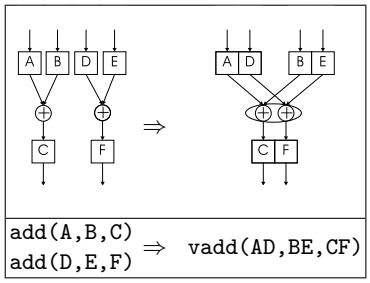

Table 1. Two-way Vectorization. Two scalar `add` instructions are transformed into one vector `vadd` instruction.

As initialization of the vectorization algorithm, 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 reuse a existing fusion, possibly requiring a data shuffle operation.

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, MAP 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)`.

Peephole Based Optimization. After the vectorization, a local rewriting system is used to implement peephole optimization on the obtained vector DAG. The rewriting rules are divided into three groups. 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 is used to extract SIMD-FMA instructions. The third group of rules rewrites unsupported SIMD instructions into sequences of SIMD instructions that are available on the target architecture.

Connecting MAP to SPIRAL and FFTW. MAP was adapted to support BlueGene/L's double FPU and connected to the SPIRAL system as backend 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. In addition, MAP was connected to FFTW's code generator `genfft` to vectorize no-twiddle codelets. MAP's output uses the intrinsic interface provided by IBM's XL C compiler for BlueGene/L. This allows MAP to vectorize computational kernels but leaves register allocation and instruction scheduling to the compiler.

5 Experimental Results

We evaluated both the formal vectorization approach and the Vienna MAP vectorizer in combination with SPIRAL, as well as the Vienna MAP vectorizer as backend to FFTW's `genfft`. We evaluated 1D FFTs with problem sizes $N = 2^2$, $2^3, \ldots, 2^{10}$ using SPIRAL, and FFT kernels (FFTW's no-twiddle codelets) of sizes $N = 2, 3, 4, \ldots, 16, 32, 64$. All experiments were performed on one PowerPC 440 FP2 processor of the BlueGene/L DD1 prototype running at 500 MHz. Performance data is given in *pseudo Gflop/s* ($5N \log_2(N)$/runtime) or as speed-up with respect to the best code without double FPU support (scalar code).

For vector lengths $N = 2^2, \ldots, 2^{10}$, we compared the following FFT implementations: (*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 vectorizer FFT implementation found by SPIRAL using the XL C compiler's vectorizer and FMA extraction; and (*v*) FFTW 2.1.5 ported to BlueGene/L (without double FPU support).

In the following we discuss the results summarized in Fig. 2. The best scalar code generated by SPIRAL without compiler vectorization (thin line, crosses) serves as baseline, as it is the fastest *scalar* code we obtained. Turning on the vectorization and FMA extraction provided by IBM's XL C compiler for BlueGene/L (dashed line, x) actually slows down SPIRAL generated FFT code by up to 15 %. FFTW 2.1.5 (thin line, asterisk) runs at approximately 80 % of the performance provided by the best scalar SPIRAL generated code. Utilizing the

Vienna MAP vectorizer as backend for SPIRAL (bold dashed line, hollow squares) produces the fastest code for problem sizes $N = 2^2$, 2^3, 2^4, for which it yields speed-ups of up to 55 %. For larger sizes MAP suffers from problems with the XL C compiler's register allocation; for problem sizes larger than 2^7, straight-line code becomes unfeasible. Formal vectorization (bold line, full squares) produces the fastest code for $N = 2^5, \ldots, 2^9$, speeding up the computation by up to up to 45 %. Smaller problem sizes are slower due to the overhead produced by formal vectorization; for $N = 2^{10}$ we again experienced problems with the XL C compiler. In addition, all codes slow down at $N = 2^{10}$ considerably due to limited data-cache capacity.

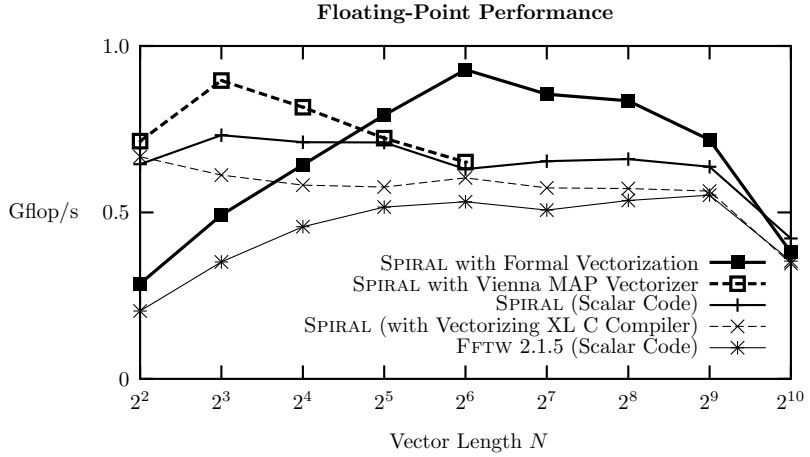

Fig. 2. Performance comparison of generated FFT scalar code and generated FFT vector code obtained with the MAP vectorizer and formal vectorization compared to the best scalar code and the best vectorized code (utilizing the VisualAge XLC for BG/L vectorizing compiler) found by SPIRAL. Performance is displayed in *pseudo Gflop/s* ($5N \log_2(N)$/runtime with N being the problem size).

In a second group of experiments summarized in Fig. 3 we applied the Vienna MAP vectorizer to FFTW's no-twiddle codelets. We again compared the performance of scalar code, code vectorized by IBM's XL C compiler for BlueGene/L, and code vectorized by the Vienna MAP vectorizer. The vectorization provided by IBM's XL C compiler speeds up FFTW no-twiddle codelets for certain sizes but yields slowdowns for others. Codelets vectorized by the Vienna MAP vectorizer are faster than both the scalar codelets and the XL C vectorized. For $N = 64$ we again experienced problems with IBM's XL C compiler.

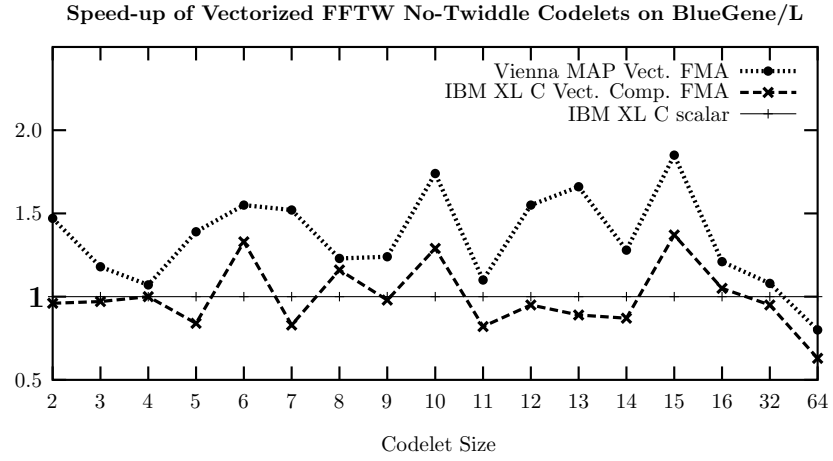

Fig. 3. Speed-up of the vectorization techniques applied by the MAP vectorizer compared to scalar code and code vectorized by IBM's VisualAge XL C compiler.

6 Conclusions and Outlook

FFTs are important tools in practically all fields of scientific computing. Both methods presented in this paper—formal vectorization techniques and the Vienna MAP vectorizer—can be used in conjunction with advanced automatic performance tuning software such as SPIRAL and FFTW helping to develop high performance implementations of FFT kernels on the most advanced computer systems.

Performance experiments carried out on a BlueGene/L prototype show that our two vectorization techniques are able to speed up FFT code considerably.

Nevertheless, even better performance results will be yielded by improving the current BlueGene/L version of the Vienna MAP vectorizer. An integral part of our future work will be to fully fold any SIMD data reorganization into special fused multiply add instructions (FMA) provided by BlueGene/L's double FPU. In addition, we are developing a compiler backend particularly suited for numerical straight-line code as output by /fftw and /spiral.

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.

Finally, we would like to acknowledge the financial support of the Austrian science fund FWF and the National Science Foundation (NSF awards 0234293 and 0325687).

References

[1] 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, pp. 543–555, 2003.

[2] R. J. Fisher and H. G. Dietz, "The Scc Compiler: SWARing at MMX and 3DNow," in *Languages and Compilers for Parallel Computing (LCPC99)*, LNCS 1863, pp.399–414, 2000.

[3] ——, "Compiling for SIMD within a register," in *Languages and Compilers for Parallel Computing*, LNCS 1656, pp. 290–304, 1998.

[4] 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.

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

[6] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient Utilization of SIMD Extensions," *IEEE Proceedings Special Issue on Program Generation, Optimization, and Platform Adaption, to appear.*

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

[8] ——, "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.

[9] ——, "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.

[10] 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.

[11] M. Frigo and S. Kral, "The Advanced FFT Program Generator Genfft," in Aurora *Technical Report TR2001-03*, vol. 3, 2001.

[12] M. Frigo and S. G. Johnson, "Fftw: An Adaptive Software Architecture for the FFT," in *ICASSP 98*, vol. 3, pp. 1381–1384, 1998.

[13] Intel Corporation, "Intel C/C++ compiler user's guide," 2002.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber, "Vectorization Techniques for BlueGene/L's Double FPU," *IBM Journal of Research and Development, to appear.*

[21] 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.

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

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