Vectorization Techniques for BlueGene/L's Double FPU

Franz Franchetti, Stefan Kral,

Juergen Lorenz, Christoph W. Ueberhuber

Institute for Analysis and Scientific Computing
Vienna University of Technology (TU Wien)
Wiedner Hauptstrasse 8–10, A-1040 Wien, Austria

E-mail: **c.ueberhuber@tuwien.ac.at**

WWW home page: **http://www.math.tuwien.ac.at/ascot**

This work described in this paper was supported by the Special Research Program SFB F011 „AURORA" of the Austrian Science Fund FWF.

Abstract

This paper presents vectorization techniques tailored to meet the specifics of the two-way single-instruction multiple-data (SIMD) double-precision floating-point unit, which is a core element of the node ASICs of IBM's 360 Tflop/s supercomputer BlueGene/L.

The paper focuses on the general-purpose basic-block vectorization methods provided by the *Vienna MAP vectorizer*. In addition, the paper introduces vectorization techniques specific to discrete signal transforms. The presented vectorization methods are evaluated in connection with the state-of-the-art automatic performance tuning systems SPIRAL and FFTW.

The combination of automatic performance tuning and the presented vectorization techniques result in FFT codes tuned automatically to a single BlueGene/L processor which are up to 60% faster than the best scalar code generated by the respective systems and five times faster than the mixed-radix FFT implementation provided by the GNU scientific library GSL.

1 Introduction

IBM's supercomputer 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 Top500 list. BlueGene/L will feature eight times more processors than current massively parallel systems. To tame this vast parallelism, new approaches and tools have to be developed. However, developing highly efficient numerical 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 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 rather 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 optimization by hand, the library generator SPIRAL [40] as well as state-of-the-art FFT libraries like FFTW [19] and UHFFT [35] use an empirical approach to automatically optimize code for a given platform.

While floating-point support for complex arithmetic is an important prerequisite to speed up large scientific codes, the utilization of non-standard FPUs in computational kernels like FFTs is not straightforward. 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 short vector SIMD vectorization. It was generated and tuned automatically by connecting SPIRAL to special purpose vectorization technology [27, 28, 30]. The resulting codes provide FFTs running five times faster than industry standard non-adaptive FFT libraries. 40% speed-up over the best code not using the double FPU is achieved while applying IBM's vectorizing compiler leads to a 15% slow-down.

2 The BlueGene/L Supercomputer

An initial small-scale prototype of IBM's 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 Top500 list (November 2003).

The BlueGene/L prototype machine is roughly 1/20th the physical size of machines of comparable compute power— such as Linux clusters —that exist today.

The full-fledged 64k processor BlueGene/L machine that is currently being built for the Lawrence Livermore National Laboratory (LLNL) will be 128 times larger, occupying 64 full racks. When completed in 2005, the BlueGene/L supercomputer is expected to lead the Top500 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 Top500 list, achieves 40 Tflop/s peak performance. The final BlueGene/L processors will run at 700 MHz, whereas the current prototype runs at 500 MHz.

BlueGene/L's Floating-Point Unit. There are many areas of scientific computing like computational electronics where complex arithmetic plays an important role. Thus, it makes sense to integrate native complex arithmetic support into the FPU of computers that are mainly devoted to such applications.

BlueGene/L's new SIMOMD style (*Single Instruction, Multiple Operation, Multiple Data*) ISA extension provide the functionality of either a complex FPU or 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 straightforward manner. However, problems arise when the usage of real code is unavoidable. Even for purely complex code it may be necessary to express complex arithmetic in terms of real arithmetic. In particular, switching to real code allows to apply common sub-expression elimination, constant folding, and copy propagation on the real and imaginary parts. For code as occurring in DSP transforms this is required to obtain satisfactory performance.

As BlueGene/L's ISA extension includes all classical short vector SIMD style (inter-operand, parallel) instructions as supported by Intel's 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 SSE2. In particular, data to be processed by the double FPU has to be naturally aligned on 16 byte boundaries in memory.

However, the PowerPC 440 FP2 has some characteristics that are different from standard short vector SIMD implementations: (*i*) short vector fused multiply-add (SIMD FMA) operations with crossover dataflow (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 two-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.

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.

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 static single assignment (SSA) 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.

SPIRAL. SPIRAL [40] 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.

FFTW. The first effort to automatically generate high-performance FFT code was FFTW [19,20]. Typically, it 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 [44] provides for 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* produced by a program generator named GENFFT [18].

4 Generating Vector Code for BlueGene/L

This chapter describes two approaches to automatically vectorizing FFT code generated by SPIRAL [40]: (i) Formal vectorization of FFT algorithms, and (ii) vectorization of basic blocks of automatically generated code. Both techniques were adapted to BlueGene/L's specifics and both of them utilize SPIRAL's search capabilities leading to automatically tuned FFT implementations that run on BlueGene/L at very high efficiency.

Formal Vectorization. The formal vectorization approach [12, 13, 14, 15, 16, 17] developed for classical short vector SIMD extensions like Intel's SSE family, AMD's 3DNow! family, and Motorola's AltiVec has been ported successfully to BlueGene/L's FPUs [29]. This kind of vectorization is based on the SIMD vectorizing version of SPIRAL's SPL compiler [15] that enables the SPIRAL system to automatically optimize code targeted at one processor of the BlueGene/L machine.

Vectorizing FFTW 3.0. The new version FFTW 3.0.1 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 (RDFTs). An important part in porting FFTW 3.0 to BlueGene/L is the mapping of the SIMD instructions required by FFTW's implementation to actually existing instructions on BlueGene/L.

FFTW's SIMD implementation [20] is based on the linearity of DFTs,

$$DFT_N(\vec{a} + i\vec{b}) = RDFT_N(\vec{a}) + i \times RDFT_N(\vec{b}),$$

and requires the computation of $y = a + ib$, with complex a, b, and y, once for every output element per FFTW codelet. One key issue is to implement this operation efficiently utilizing BlueGene/L's special FMA instructions.

A straight-forward implementation of the multiplication by i that minimizes the number of arithmetic operations is given by

```
y = __fpadd(a,__fsneg(__fxmr(b)))
```

in XL C99 intrinsic syntax. This implementation requires three instructions for two flops and one sign change and thus wastes ten out of twelve flops (when a sign change is not considered as a flop).

A BlueGene/L specific implementation that utilizes the PowerPC 440 FP2 specific FMA instruction

$$\texttt{\_\_fxmadd}((a_p,a_s),(b_p,b_s),(c_p,c_s))=(a_p+b_s*c_p,\ a_s+b_p*c_s)$$

is given by

```
y = __fxmadd(c,b,__cmplx(-1,1))
```

leading to only one instruction. This implementation introduces artificial multiplications by one and minus one to allow the utilization of a cross FMA provided by the double FPU. In subsequent optimization, constant folding may even change the artificial multiplications into real ones.

The MAP Vectorizer. The Vienna MAP vectorizer [27, 28, 30] that extracts two-way SIMD parallelism out of automatically generated numerical straight-line code was adapted to support BlueGene/L's FPUs. As a supplement to the MAP vectorizer a peephole optimizer enables the extraction of fused multiply-add SIMD instructions. A thorough description of the Vienna MAP vectorizer can be found in Section 4.4.

Fig. 2 gives an example of short vector SIMD code obtained by vectorizing straight-line complex FFT code.

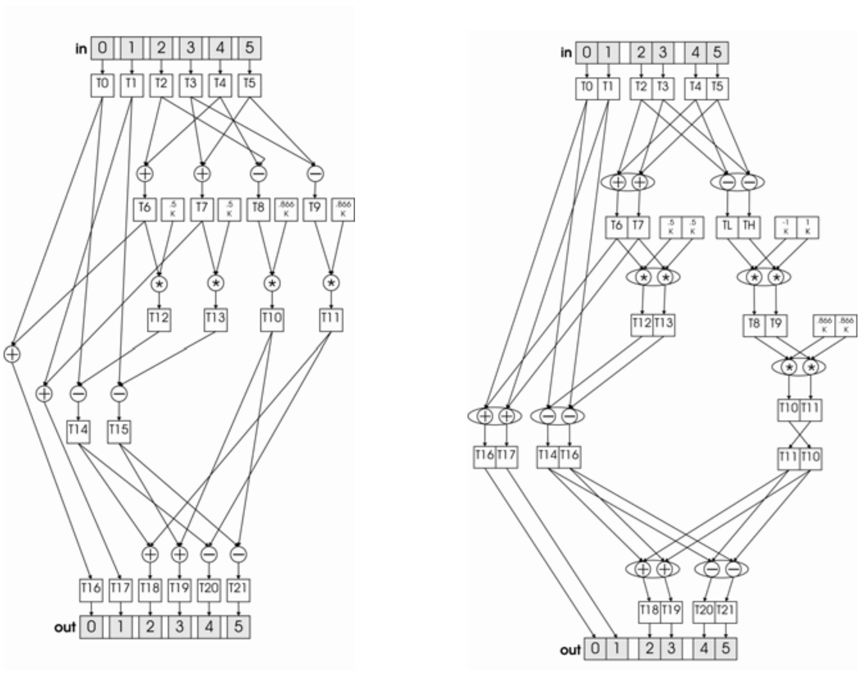


Figure 2: Vectorization of a Scalar FFT of Size 3. The scalar data flow in the left part of the illustration is computationally equivalent to the vectorized data flow depicted in the right part.

Related Vectorization Techniques. Some methods for vectorizing basic blocks [11, 32, 33] try to find an efficient mix of SIMD and scalar instructions to carry out the required computation whereas MAP's vectorization principle mandates that all computation is performed by

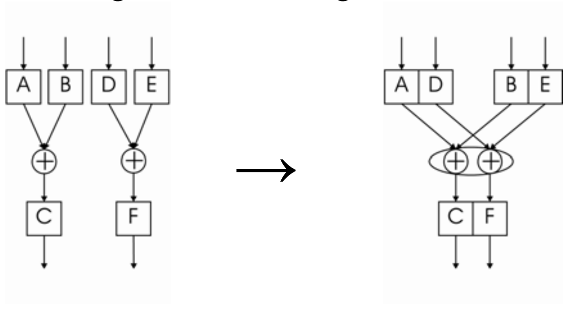
SIMD instructions. The MAP vectorizer makes an attempt to fully utilize the power of SIMD instructions while trying to keep the SIMD
reordering overhead reasonably small.

The vectorization method of [32] introduces more SIMD data reordering instructions than necessary, as it is unable to resort to a representation of the numerical scalar DAG as vectorization input, and is thus deprived of this parallelism revealing instrument. This approach is not a suitable choice for handling typical numerical codes, e. g., FFTs, efficiently, as explicit SIMD data reordering operations are very expensive on BlueGene/L's FPU.

5 The Vienna MAP Vectorizer

The Vienna MAP vectorizer automatically extracts two-way short vector SIMD parallelism out of a scalar code block by adequately combining scalar variables to SIMD variables and by joining the corresponding scalar instructions to one or more short vector SIMD instructions, as illustrated in Fig. 3. The MAP vectorizer targets automatically generated code that solely consists of arithmetic operations and read/write array access operations involving index computation.

Ideally, two-way vectorization transforms any pair of scalar instruction to one SIMD instruction yielding 100% SIMD utilization. This maximum is achievable only for completely parallel scalar DAGs. For DAGs with less parallelism, SIMD reordering instructions are required, at the costs of reduced SIMD utilization. As not all combinations of scalar operations may be joined into one SIMD instruction (as defined by the ISA extension of the target processor), the vectorizer's realistic goals are (i) to completely cover the given scalar DAG by natively supported SIMD instructions while achieving (ii) a satisfactory runtime performance, which is tantamount to minimizing SIMD data reorganization.



add(A,B,C), add(D,E,F) ⟶ v_add(AD,BE,CF)

Figure 3: Example of Two-way Vectorization. Ideally, two scalar instructions are transformed into one vector instruction to achieve optimal SIMD coverage.

5.1 Fundamentals of Vectorization

A thorough description of the two-way vectorization algorithm rests upon the following fundamentals.

Variable Fusion. Two scalar variables *A, B* can be fused either to a SIMD variable of the form *AB = (A,B)* or the other way round, *BA = (B,A)*, where *AB ≠ 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 to 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* requested compatible has not to be generated as AB can be used. In the second case, a SIMD swap operation is required to maintain data flow consistency when using fusion the AB instead of generating and using *CD* (see Fig. 4).

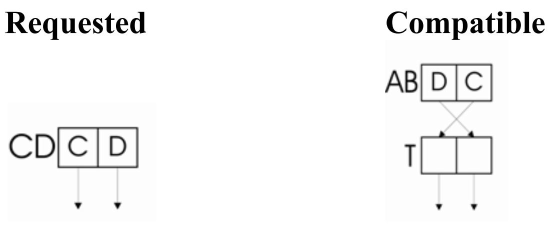


Figure 4: Compatible Fusion. The vectorization process requests a fusion *CD = (C,D)*. The existing fusion *AB = (D,C)* is used as input operand to a swap instruction whose output *T* can be used whenever *CD* is needed.

Operation Joining. Joining rules specify ways of pairwise transforming scalar instructions into one or more SIMD instructions. The MAP vectorizer supports transformations of the following instruction combinations: (i) load/load, (ii) store/store, (iii) unary/unary, (iv) binary/binary, (v) unary/binary, (vi) unary/load, and (vii) load/binary. Joining rules (i) and (ii) support the transformation of memory operations accessing consecutive and non-consecutive memory locations as illustrated in the example of Fig. 6. Rules (iii) and (iv) allow the pairing of instructions of the same type only, while (v) to (vii) also allow mixed type pairings (see Fig. 7).

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

Rules of type (iv) provide several alternatives (see Fig. 8). As they target two binary instructions *(bop1,S1,T1,D1)* and *(bop2,S2,T2,D2)*, different possibilities for choosing the fusion partners among the four source operands *S1, T1, S2* and *T2* arise. Thus, three layouts, ACC, PAR and CHI defining the possibilities for fusing the operand variables for binary instructions as shown in Fig. 5 are introduced. ACC is needed to fuse 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 Fig. 8.

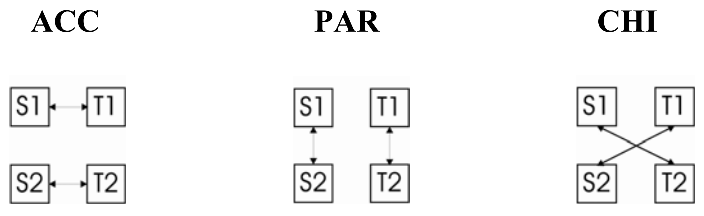


Figure 5: Fusion Layouts. Three layouts for fusing the source variables of the scalar instructions *(op1,S1,T1,D1)* and *(op2,S2,T2,D2)* are supported.

Vectorization Quality. To extract high performance short vector SIMD code distinguished by good SIMD utilization, the joining rules include 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 applying a peephole optimization directly after the vectorization process.

The vectorization engine starts out by constraining all SIMD memory operations to access consecutive locations and by disabling the sub-optimal pairing rules (v)-(vii). If these restrictions cause the vectorization process to fail, it is restarted after enabling operation pairing rules (v)-(vii) and support for less efficient, i.e., non-consecutive, memory access operations. This extension substantially augments the class of vectorizable codes by allowing the extraction of some less efficient instruction combinations.

5.2 The Vectorization Algorithm

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

First, a 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 heuristics to speed up the vectorization process by avoiding futile vectorization attempts.

Finally, store instructions are combined non-deterministically by fusing their respective source operands.

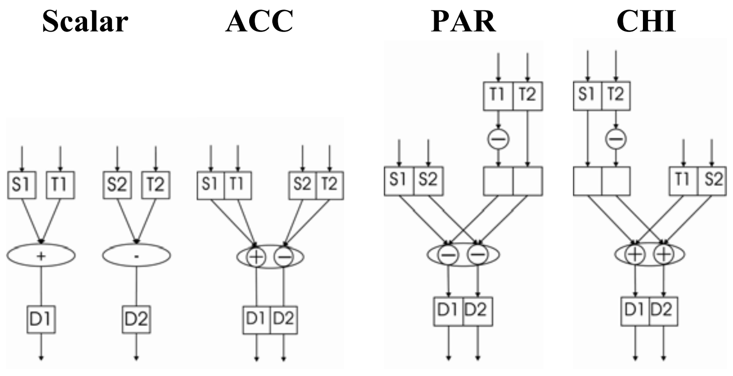


Figure 8: Vectorization Alternatives. Two scalar instructions, one addition and one subtraction, are transformed into a sequence of SIMD instructions in three different ways.

The Actual Vectorization. The vectorization algorithm consists of two steps.

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

(*ii*) The two scalar operations *op1* and *op2* are paired non-deterministically, 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 has to ensure that no scalar variable is part of two different fusions.

The vectorizer alternatingly applies steps (i) and (ii) 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.

Non-determinism in Vectorization. Non-determinism in vectorization arises due to vectorization alternatives like 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 Fig. 8. This kind of non-determinism widens the search space of the vectorizer's 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. As the vectorizer always commits to the first solution, the rule ranking is adapted such that the first solution favors instruction sequences which are particularly well-suited for the given target machine, taking the different costs of individual instructions (see Fig. 9) into account.

Nevertheless, the ranking must be seen as something like an extraction „hint". At every point of decision the search engine initially tries the rule that is ranked first. If this does not lead to a vectorization, later ranked rules are used as well, even if their application leads to 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.

ISA / SIMD Op	basic 3Dnow! (K6-II+)	ext. 3Dnow! (K7/K8)	SSE2 (P4/K8)	SSE3 (P4e)	IA64 (Itanium)	Double FPU (440FP2)
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

Figure 9: Relatives Costs of SIMD Operations. For a selection of ISA extensions, the number of actual SIMD instructions necessary to implement the respective SIMD operations, is given. This data directly influences the rule ranking used by the MAP vectorizer.

5.3 The Realization of the Vectorization Engine

MAP's vectorization algorithm is implemented using a depth-first search engine with chronological backtracking.

The vectorization engine's backtracking capability is indispensable when a fusion, requested by the current vectorization alternative, does not comply with the globally existing fusions, i.e., in cases when it is impossible to ensure that the scalar variables considered for the new fusion are not already participating in existing fusions.

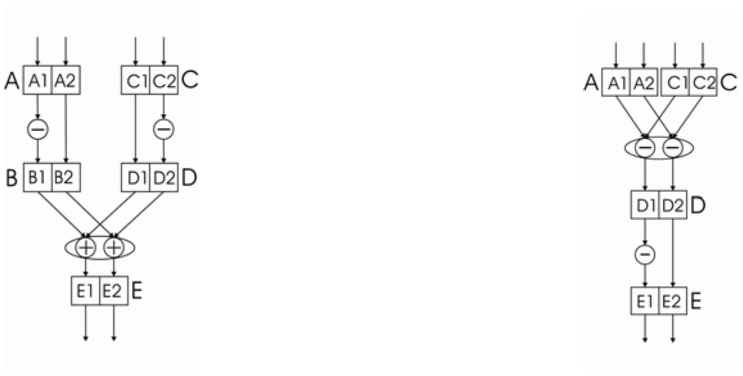
In such a case, the search engine backtracks to the last non-deterministic point of decision. There, another vectorization alternative is chosen and correspondingly fusions of different layout are requested and generated if necessary. If these fusions comply with the set of existing fusions the vectorization process commits to this rule. 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 was not able to find a valid fusion set for the given scalar DAG.

6 The Vienna MAP Optimizer

The Vienna MAP Optimizer is a rule-based local rewriting system that implements peephole optimization on vector DAGs. It post-processes 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 scheduler of GENFFT [18].

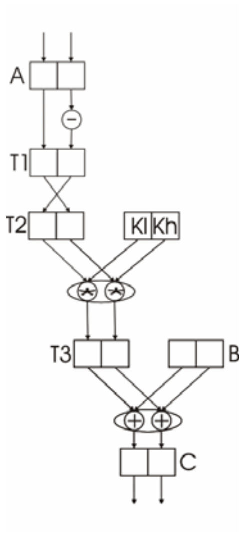
The first group of rewriting rules aims at general optimizations such as (*i*) the minimization of the instruction count, (*ii*) redundancy and dead code elimination, (*iii*) the reduction of the number of source operands (which reduces register pressure), (*iv*) the minimization of the critical path length of the vector DAG, (v) copy propagation, and (*vi*) constant folding. On target architectures supporting FMAs (Intel Itanium, IBM PowerPC 440FP2) FMAs are extracted by combining multiplications (or sign changes) with directly dependent additions (or subtractions 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.

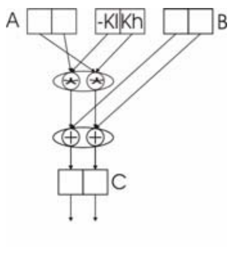
The second group of rewriting rules is target architecture specific. When optimizing for the IBM PowerPC 440 FP2 used in BlueGene/L, vector swap instructions are fold into FMAs, utilizing vector cross FMA instructions exclusively available on BlueGene/L, using a method similar to extracting FMAs.



v_chsLo(A,B) **v_chsHi(C,D)** **v_add(B,D,E)** → **v_sub(A,C,D)** **v_chsLo(D,E)**

Figure 9: Example of a 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.





v_chsHi(A,T1)
v_swap(T1,T2) → **v_cfmac(A,(-Kl,Kh),B,C)**
v_mulC(T2,(Kl,Kh),T3)
v_add(T3,B,C)

Figure 9: Example of a BlueGene/L Specific Optimization Rule. A vector mulconst instruction *v_mulC(T2,(Kl,Kh),T3)* taking the output of a vector swap instruction *v_swap(T1,T2)* 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.

6 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 addition, FFTW no-twiddle codelets for size 2, 3,…, 16, 32, and 64 were vectorized using the Vienna MAP vectorizer. For the same problem sizes, FFTW 3.0 SIMD codelets ported to BlueGene/L were assessed.

Fig. 12 compares different FFT implementations for vector lengths $N = 2^2, 2^3, \ldots, 2^{10}$. In particular the following FFT implementations were tested: (i) The best vectorized code obtained using all technologies presented in this paper (LIBDFT), (ii) the best scalar FFT implementation found by SPIRAL (XL C's vectorizer and FMA extraction turned off), (iii) the best vectorized FFT implementation found by SPIRAL using the XL C compiler's vectorizer and FMA extraction turned on, and (iv) the mixed-radix FFT implementation provided by the GNU scientific library (GSL).

```
static const _Complex double __align(16) VECT_CONST1 =
     __cmplx(-1.000000000000000, -1.000000000000000);
...
static const _Complex double __align(16) VECT_CONST21 =
     __cmplx(+0.634393284163645, +0.773010453362737);

void DFT_64(double *y, double *x)
{
   _Complex double f0;
   ...
   _Complex double f603;
   f0 = __lfpd((double *)(x+64));
```

```
    f1 = __lfpd((double *)(x+0));
    f2 = __fpadd(f0,f1);
    f3 = __fpmadd(f0,VECT_CONST1,f1);
    ...
    f417 = __cmplx(__creal(f415),__creal(f416));
    f418 = __cmplx(__cimag(f415),__cimag(f416));
    ...
    f602 = __fpmadd(f511,VECT_CONST2,f407);
    f603 = __fpmadd(f358,VECT_CONST3,f476);
    __stfpd((double *)(y+34), t602);
    __stfpd((double *)(y+98), t603);
}
```

Figure 11: Example output of BlueGene/L MAP vectorizer. Scalar code for a DFT_{64} generated by SPIRAL is vectorized using XL C99 intrinsics.

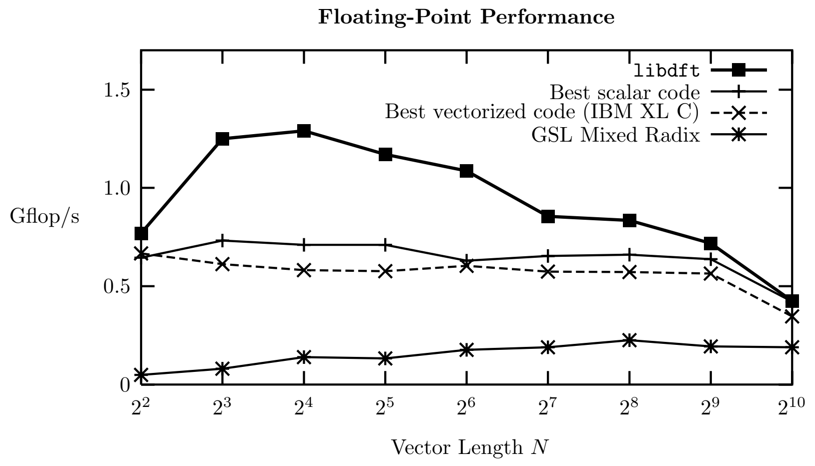


Figure 12: Performance of the vectorization techniques introduced by this paper (LIBDFT) compared to the best scalar code and the best vectorized code (utilizing the VisualAge XLC for BlueGene/L vectorizing compiler) found by SPIRAL. Performance is displayed in pseudo Gflop/s ($5N \log N$/runtime with N being the vector length).

The combination of all methods as implemented in LIBDFT leads to 60% speed-up w.r.t. the best scalar codes generated by SPIRAL for smaller problem sizes and 20% speed-up for larger problem sizes. Thus formal vectorization provides significant speed-up for larger problem sizes.

The third-party GNU GSL FFT library reaches about 30% of the performance of the best scalar SPIRAL generated code thus performing badly.

XL C's vectorization and FMA extraction 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. 13 compares the speed of scalar FFTW codelets to the speed of codelets vectorized by the XL C compiler and codelets vectorized by the Vienna MAP vectorizer. While the vectorization obtained by the Vienna MAP vectorizer speeds up the computation significantly, XL C compiler provides only small speed-ups and sometimes even slows down the computation.

Fig. 14 compares FFTW 3.0 SIMD codelets ported to BlueGene/L to the vectorization capabilities of the XL C compiler. The trend in Fig. 14 is similar to Fig. 13: The XL C compiler's vectorization provides small speed-ups and sometimes slows down the code while FFTW's SIMD codelets speed up the computation significantly.

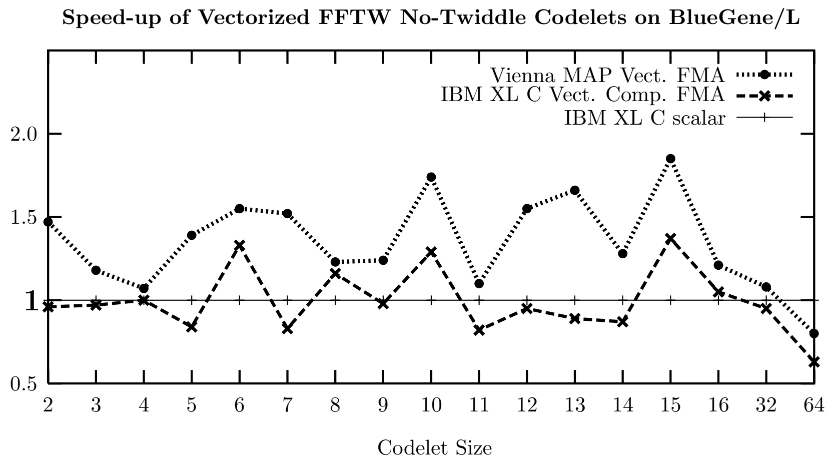


Figure 13: Speed-up of the vectorization techniques applied by the MAP vectorizer compared to scalar code and code vectorized by IBM's VisualAge XL C compiler.

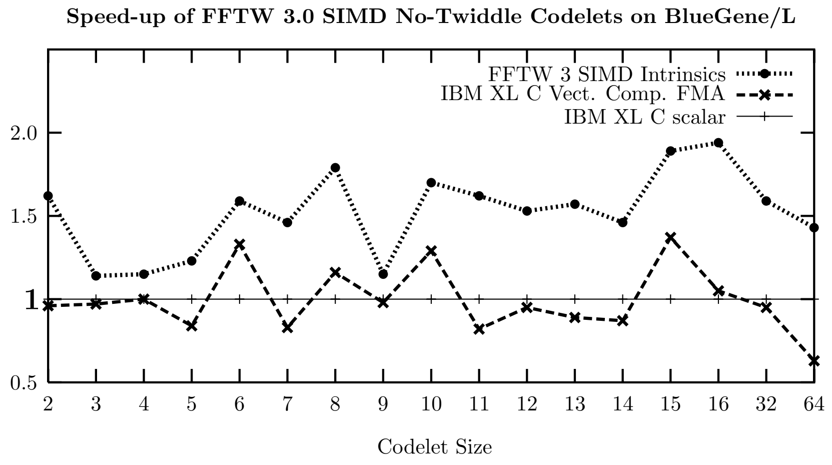


Figure 14: Speed-up of the FFTW 3.0 SIMD codelets compared to scalar code and code vectorized by IBM's VisualAge XL C compiler.

7 Conclusions and Outlook

As FFTs are indispensable parts of practically all kinds of applications in scientific computing, efficient FFT software is 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 can be used to automatically vectorize numerical straight line code generated by advanced automatic performance tuning software like FFFTW or SPIRAL helping to develop highly efficient 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 the current version of 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 will be obtained by further improving the current 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 that is better suited for numerical straight-line code than the backend of IBM's XL C compiler.

Acknowledgements. Special thanks go 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) 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," in 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." to appear in IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation.

[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 Proceedings 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 Proceedings ICASSP, vol. 2, pp. 1109–1112, 2001.

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

[16] —, „Short vector code generation and adaptation for DSP algorithms." in Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (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," to appear in IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation.

[21] J. Guo, M. Garzaran, 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 super-pipelined 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] S. Kral, F. Franchetti, J. Lorenz, M. Püschel, C. Ueberhuber, and P. Wurzinger, „Automatically Optimized FFT Codes for the BlueGene/L Supercomputer," in VecPar Proceedings on High Performance Computing for Computational Science, 2004.

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

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

[32] S. Larsen and S. Amarasinghe, „Exploiting super-word level parallelism with multimedia instruction sets," ACM SIGPLAN Notices, vol. 35, no. 5, pp. 145–156, 2000.

[33] 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

[34] 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

[35] D. Mirkovic and S. L. Johnsson, „Automatic Performance Tuning in the UHFFT Library," in Proceedings ICCS'01, pp. 71–80, 2001.

[36] J. M. F. Moura, J. Johnson, D. Padua, M. Püschel, and M. Veloso, „SPIRAL." to appear in IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation.

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

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

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

[40] 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 pro cessing algorithms," Journal on High Performance Computing and Applications, special issue on Automatic Performance Tuning, Vol. 18, pp. 21–45, 2004, http://www.SPIRAL.net

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

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

[43] P. N. Swarztrauber, „FFT algorithms for vector computers," Parallel Computing, vol. 1, pp. 45–63, 1984.

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

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

[46] 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.

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

[48] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.