Operator Language: A Program Generation Framework for Fast Kernels *

Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213, USA
{franzf, fdemesma, dmcfarli, pueschel}@ece.cmu.edu

Abstract. We present the Operator Language (OL), a framework to automatically generate fast numerical kernels. OL provides the structure to extend the program generation system Spiral beyond the transform domain. Using OL, we show how to automatically generate library functionality for the fast Fourier transform and multiple non-transform kernels, including matrix-matrix multiplication, synthetic aperture radar (SAR), circular convolution, sorting networks, and Viterbi decoding. The control flow of the kernels is data-independent, which allows us to cast their algorithms as operator expressions. Using rewriting systems, a structural architecture model and empirical search, we automatically generate very fast C implementations for state-of-the-art multicore CPUs that rival hand-tuned implementations.

Key words: Library generation, program generation, automatic performance tuning, high performance software, multicore CPU

1 Introduction

In many software applications, runtime performance is crucial. This is particularly true for compute-intensive software that is needed in a wide range of application domains including scientific computing, image/video processing, communication and control. In many of these applications, the bulk of the work is performed by well-defined mathematical functions or *kernels* such as matrix-matrix multiplication (MMM), the discrete Fourier transform (DFT), convolution, or others. These functions are typically provided by high performance libraries developed by expert programmers. A good example is Intel's Integrated Performance Primitives (IPP) [1], which provides around 10,000 kernels that are used by commercial developers worldwide.

Unfortunately, the optimization of kernels has become extraordinarily difficult due to the complexity of current computing platforms. Specifically, to run fast on, say, an off-the-shelf Core 2 Duo, a kernel routine has to be optimized for the memory hierarchy, use

* This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH1050009), the ARO grant W911NF0710416, and by Intel Corp. and Mercury Computer Systems, Inc.

SSE vector instructions, and has to be multithreaded. Without these optimizations, the performance loss can be significant. To illustrate this, we show in Fig. 1 the performance of four implementations of MMM (all compiled with the latest Intel compiler) measured in giga-floating point operations per second (Gflop/s). At the bottom is a naive triple loop. Optimizing for the memory hierarchy yields about 20x improvement, explicit use of SSE instructions another 2x, and threading for 4 cores another 4x for a total of 160x.

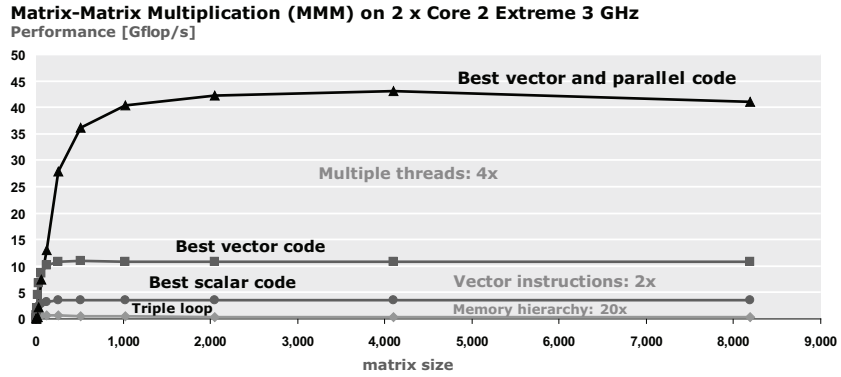


Fig. 1. Performance of four double precision implementations of matrix-matrix multiplication. The operations count is exactly the same. The plot is taken from [2].

In other words, the compiler cannot perform the necessary optimizations for two main reasons. First, many optimizations require high level algorithm or domain knowledge; second, there are simply too many optimization choices that a (deterministic) compiler cannot explore. So the optimization is left with the programmer and is often platform specific, which means it has to be repeated whenever a new platform is released.

We believe that the above poses an ideal scenario for the application of domain-specific languages (DSLs) and program generation techniques to automatically generate fast code for kernel functions directly from a specification. First, many kernel functions are fixed which helps with the design of a language describing their algorithms, and available algorithms have to be formalized only once. Second, since many optimizations require domain knowledge, there is arguably no way around a DSL if automation is desired. Third, the problem has real-world relevance and also has to address the very timely issue of parallelism.

Contribution of this paper. In this paper we present a program generation framework for kernel functions such as MMM, linear transforms (e.g. DFT), Viterbi decoding, and others. The generator produces code directly from a kernel specification and the performance of the code is, for the kernels considered, often competitive with the best available hand-written implementations. We briefly discuss the main challenges and how they are addressed by our approach:

– *Algorithm generation.* We express algorithms at a high abstraction level using a DSL called operator language (OL). Since our domain is mathematical, OL is de-

rived from mathematics and it is declarative in that it describes the structure of a computation in an implementation-independent form. Divide-and-conquer algorithms are described as OL breakdown rules. By recursively applying these rules a space of algorithms for a desired kernel can be generated. The OL compiler translates OL into actual C code.

– *Algorithm optimization* We describe platforms using a very small set of parameters such as the vector datatype length or the number of processors. Then we identify OL optimization rules that restructure algorithms. Using these rules together with the breakdown rules yields optimized algorithms. It is an example of rule-based programming and effectively solves the problem of domain specific optimizations. Beyond that, we explore and time choices for further optimization.

Our approach has to date a number of limitations. First, we require that the kernels are for fixed input size. Second, the kernels have to be input data independent. Third, the algorithms for a kernel have to possess a suitable structure to be handled efficiently.

This paper builds on our prior work on Spiral, which targets program generation for linear transforms based on the language SPL [3–6]. Here we show for the first time how to go beyond this domain by extending SPL to OL using a few kernels as examples.

Related work. The area of program generation (also called generative programming) has gained considerable interest in recent years [7–11]. The basic goal is to reduce the development, maintenance, and analysis of software. Among the key tools for achieving these goals, domain-specific languages provide a compact representation that raises the level of abstraction for specific problems and hence enables the manipulation of programs [12–16]. However, this work has to date rarely considered numerical problems and focused on correctness rather than performance.

ATLAS [17] is an automatic performance tuning system that optimizes basic linear algebra functionality (BLAS) using a code generator to generate kernels that are used in a blocked parameterized library. OSKI (and its predecessor Sparsity) [18] is an automatic performance tuning system for matrix-vector multiplication for sparse matrices. The adaptive FFT library FFTW [19] implements an adaptive version of the Cooley-Tukey FFT algorithm. It contains automatically generated code blocks (codelets) [20]. The TCE [21] automates implementing tensor contraction equations used in quantum chemistry. FLAME [22] automates the derivation and implementation of dense linear algebra algorithms.

Our approach draws inspiration and concepts from symbolic computation and rule-based programming. Rewriting systems are reviewed in [23]. Logic programming is discussed in [24]. An overview of functional programming can be found in [25].

2 Program Generation Framework for Fast Kernels

The goal of this paper is to develop a program generator for high performance kernels. A kernel is a fixed function that is used as library routine in different important applica-

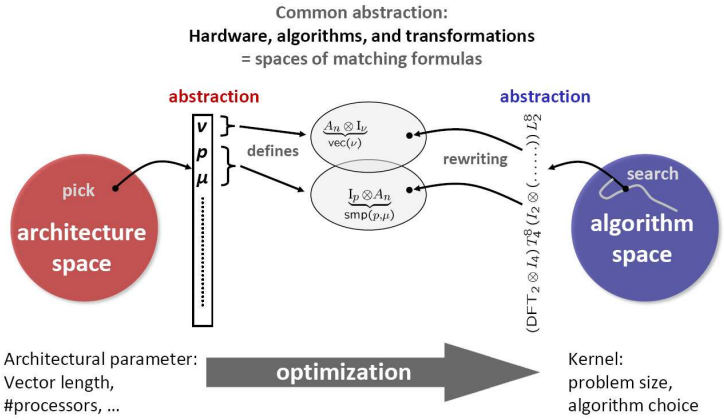


Fig. 2. Our approach to program generation: The spaces of architectures (left) and algorithms (right) are abstracted and joined using the operator language framework.

tions. We require that the input size for the kernel is fixed. Examples of kernels include a discrete Fourier transform for input size 64 or the multiplication of two 8×8 matrices.

To achieve high performance, our generator produces programs that are optimized to the specifics of the targeted computing platform. This includes tuning to the memory hierarchy and the efficient use of vector instructions and threading (if available). To do this efficiently, a few platform parameters are provided to the generator and the platform is assumed to be available for benchmarking of the generated code, so an exploration of alternatives is possible.

In summary, the input to our generator is a kernel specification (e.g., DFT_{64}) and platform parameters (e.g., 2 cores); the output is a fast C function for the kernel. Performance is achieved through both structural optimization of available algorithms based on the platform parameters and search over alternatives for the fastest. Intuitively, the search optimizes for the memory hierarchy by considering different divide-and-conquer strategies.

Fig. 2 shows a very high level description of our approach, which consists of three key components: 1) A DSL called operator language (OL) to describe algorithms for kernels (right bubble); 2) the use of tags to describe architecture features (left bubble); 3) a common abstraction of architecture and algorithms using tagged OL. We briefly describe the three components and then give detailed explanations in separate subsections.

Operator language (OL): Kernels and algorithms. OL is a mathematical DSL to describe structured divide-and-conquer algorithms for data-independent kernels. These algorithms are encoded as breakdown rules and included into our generator. By recursively applying these rules, the generator can produce a large space of alternative algorithms for the desired kernel. OL is platform-independent and declarative in nature; it is an extension of SPL [3, 4] that underlies Spiral.

Hardware abstraction: Tags. We divide hardware features into parameterized paradigms. In this paper we focus on two examples: vector processing abilities (e.g., SSE on Intel P4 and later) with vector length ν, and shared memory parallelism with p cores.

Common abstraction: Tagged operator language. The key challenge in achieving performance is to optimize algorithms for a given paradigm. We do this in steps. First, we introduce the hardware paradigms as tags in OL. Second, we identify basic OL expressions that can be efficiently implemented on that paradigm. These will span a sublanguage of OL that can be implemented efficiently. Finally, we add generic OL rewriting rules in addition to the breakdown rules describing algorithms. The goal is that the joint set of rules now produces structurally optimized algorithms for the desired kernel. If there are still choices, feedback driven search is used to find the fastest solution.

Besides the above, our generator requires a compiler that translates OL into actual C, performing additional low level optimizations (Section 3). However, the focus of this paper is the OL framework described next in detail.

2.1 Operator Language: Kernels and Algorithms

In this section we introduce the Operator Language (OL), a domain-specific language designed to describe structured algorithms for data-independent kernel functions. The language is declarative in that it describes the structure of the dataflow and the data layout of a computation, thus enabling algorithm manipulation and structural optimization at a high level of abstraction. OL is a generalization of SPL [3, 4], which is designed for linear transforms.

The main building blocks of OL are *operators*, combined into *operator formulas* by *higher-order operators*. We use OL to describe recursive (divide-and-conquer) algorithms for important kernels as *breakdown rules*. The combination of these rules then produces a space of alternative algorithms for this kernel.

Operators. Operators are n-ary functions on vectors: an operator of arity (r, s) consumes r vectors and produces s vectors. An operator can be (multi)linear or not. Linear operators of arity $(1, 1)$ are precisely linear transforms, i.e., mappings $x \mapsto Mx$, where M is a fixed matrix. We often refer to linear transforms as matrices. When necessary, we will denote $A_{m\times n\to p}$ an operator A going from $\mathbb{C}^m \times \mathbb{C}^n$ into $\mathbb{C}^p$.

Matrices are viewed as vectors stored linearized in memory in row major order. For example, the operator that transposes an $m \times n$ matrix[1], denoted by L^{mn}_n, is of arity $(1, 1)$. Table 1 defines a set of basic operators that we use.

Kernels. In this paper, a computational kernel is an operator for which we want to generate fast code. We will use matrix-matrix multiplication and the discrete Fourier transform as running examples to describe OL concepts. However, we also used OL to capture other kernels briefly introduced later, namely: circular convolution, sorting networks, Viterbi decoding, and synthetic aperture radar (SAR) image formation.

[1] The transposition operator is often referred to as the *stride* permutation or *corner turn*.

We define the *matrix-multiplication* $\mathrm{MMM}_{m,k,n}$ as an operator that consumes two matrices and produces one[2]:

$$\mathrm{MMM}_{m,k,n} : \mathbb{R}^{mk} \times \mathbb{R}^{kn} \to \mathbb{R}^{mn};\ (\mathbf{A}, \mathbf{B}) \mapsto \mathbf{AB}$$

The *discrete Fourier transform* DFT_n is a linear operator of arity $(1, 1)$ that performs the following matrix-vector product:

$$\mathrm{DFT}_n : \mathbb{C}^n \to \mathbb{C}^n;\ \mathbf{x} \mapsto [e^{-2\pi ikl/n}]_{0 \le k,l < n}\mathbf{x}$$

Higher-order operators. Higher-order operators are functions on operators. A simple example is the *composition*, denoted in standard infix notation by $\circ$. For instance,

$$L_n^{mn} \circ \mathrm{P}_{mn}$$

is the arity $(2, 1)$ operator that first multiplies point-wise two matrices of size $m \times n$, and then transposes the result.

The *cross product* of two operators applies the first operator to the first input set and the second operator to the second input set, and then combines the outputs. For example,

$$L_n^{mn} \times \mathrm{P}_{mn}$$

is the arity $(3, 2)$ operator that transposes its first argument and multiplies the second and third argument pointwise, producing two output vectors.

The most important higher order operator in this paper is the *tensor product*. For linear operators A, B of arity (1,1) (i.e., matrices), the tensor product corresponds to the tensor or Kronecker product of matrices:

$$A \otimes B = [a_{k,l}B], \quad A = [a_{k,l}].$$

An important special case is the tensor product of an identity matrix and an arbitrary matrix,

$$I_n \otimes A = \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix}.$$

This can be interpreted as applying A to a list of n contiguous subvectors of the input vector. Conversely, $A \otimes I_n$ applies A multiple times to subvectors extracted at stride n.

The Kronecker product is known to be useful for concisely describing DFT algorithms as fully developed by Van Loan [26] and is the key construct in the program generator Spiral for linear transforms [4]. Its usefulness is in the concise way that it captures loops, data independence, and parallelism.

We now formally extend the tensor product definition to more general operators, focusing on the case of two operators with arity (2,1); generalization is straightforward.

[2] We use this definition for explanation purposes; the MMM required by BLAS [17] has a more general interface.

name	definition
Linear, arity (1,1)	
identity	$I_n : \mathbb{C}^n \to \mathbb{C}^n;\ \mathbf{x} \mapsto \mathbf{x}$
vector flip	$J_n : \mathbb{C}^n \to \mathbb{C}^n;\ (x_i) \mapsto (x_{n-i})$
transposition of an $m \times n$ matrix	$L^{mn}_m : \mathbb{C}^{mn} \to \mathbb{C}^{mn};\ \mathbf{A} \mapsto \mathbf{A}^T$
matrix $M \in \mathbb{C}^{m\times n}$	$\mathrm{M} : \mathbb{C}^n \to \mathbb{C}^m; \mathbf{x} \mapsto M\mathbf{x}$
Bilinear, arity (2,1)	
Point-wise product	$\mathrm{P}_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n;\ ((x_i), (y_i)) \mapsto (x_i y_i)$
Scalar product	$\mathrm{S}_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C};\ ((x_i), (y_i)) \mapsto \Sigma(x_i y_i)$
Kronecker product	$\mathrm{K}_{m\times n} : \mathbb{C}^m \times \mathbb{C}^n \to \mathbb{C}^{mn};\ ((x_i), \mathbf{y})) \mapsto (x_i \mathbf{y})$
Others	
Fork	$\mathrm{Fork}_n : \mathbb{C}^n \to \mathbb{C}^n \times \mathbb{C}^n;\ \mathbf{x} \mapsto (\mathbf{x}, \mathbf{x})$
Split	$\mathrm{Split}_n : \mathbb{C}^n \to \mathbb{C}^{n/2} \times \mathbb{C}^{n/2};\ \mathbf{x} \mapsto (\mathbf{x}^U, \mathbf{x}^L)$
Concatenate	$\oplus_n : \mathbb{C}^{n/2} \times \mathbb{C}^{n/2} \to \mathbb{C}^n;\ (\mathbf{x}^U, \mathbf{x}^L) \mapsto \mathbf{x}$
Duplication	$\mathrm{dup}^m_n : \mathbb{C}^n \to \mathbb{C}^{nm};\ \mathbf{x} \mapsto \mathbf{x} \otimes I_m$
Min	$\min_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n;\ (\mathbf{x}, \mathbf{y}) \mapsto (\min(x_i, y_i))$
Max	$\max_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n;\ (\mathbf{x}, \mathbf{y}) \mapsto (\max(x_i, y_i))$

Table 1. Definition of basic operators. The operators are assumed to operate on complex numbers but other base sets are possible. Boldface fonts represent vectors or matrices linearized in memory. Superscripts U and L represent the upper and lower half of a vector. A vector is sometimes written as $\mathbf{x} = (x_i)$ to identify the components.

Let $\mathcal{A} :\ \mathbb{C}^p \times \mathbb{C}^q \to \mathbb{C}^r$ be a *multi-linear* operator and let $\mathrm{B} :\ \mathbb{C}^m \times \mathbb{C}^n \to \mathbb{C}^k$ be *any* operator. We denote the ith canonical basis vector of $\mathbb{C}^n$ with $\mathbf{e}^n_i$. Then

$$(\mathcal{A} \otimes \mathrm{B})(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} \mathcal{A}(\mathbf{e}^p_i, \mathbf{e}^q_j) \otimes \mathrm{B}((\mathbf{e}^{p^T}_i \otimes I_m)\boldsymbol{x}, (\mathbf{e}^{q^T}_j \otimes I_n)\boldsymbol{y})$$

$$(\mathrm{B} \otimes \mathcal{A})(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} \mathrm{B}((I_m \otimes \mathbf{e}^{p^T}_i)\boldsymbol{x}, (I_n \otimes \mathbf{e}^{q^T}_j)\boldsymbol{y}) \otimes \mathcal{A}(\mathbf{e}^p_i, \mathbf{e}^q_j)$$

Intuitively, $\mathcal{A}$ describes the coarse structure of the algorithm and captures how to operate on the chunks of data produced by B. Therefore, the structure of the operations $\mathcal{A}$ and $\mathcal{A} \otimes \mathrm{B}$ is similar. For instance, consider the point-wise product P_2 and the tensor product $\mathrm{P}_2 \otimes \mathrm{B}$ (we denote with the superscripts U and L the upper and lower halves of a vector):

$$\begin{pmatrix} x_0 \\ \hline x_1 \end{pmatrix} \times \begin{pmatrix} y_0 \\ \hline y_1 \end{pmatrix} \xrightarrow{\mathrm{P}_2} \begin{pmatrix} x_0 \cdot y_0 \\ \hline x_1 \cdot y_1 \end{pmatrix} \qquad \begin{pmatrix} \mathbf{x}^U \\ \hline \mathbf{x}^L \end{pmatrix} \times \begin{pmatrix} \mathbf{y}^U \\ \hline \mathbf{y}^L \end{pmatrix} \xrightarrow{\mathrm{P}_2 \otimes \mathrm{B}} \begin{pmatrix} \mathrm{B}(\mathbf{x}^U, \mathbf{y}^U) \\ \hline \mathrm{B}(\mathbf{x}^L, \mathbf{y}^L) \end{pmatrix}.$$

We show another example by selecting the Kronecker product $\mathrm{K}_{2\times 2}$ (now viewed as operator of arity $(2, 1)$ on vectors, see Table 1, not viewed as higher order operator). Again, the multilinear part of the tensor product describes how blocks are arranged and

(a) Horizontal blocking (b) Vertical blocking (c) Depth blocking

Fig. 3. Blocking matrix multiplication along each one of the three dimensions. For the horizontal and vertical blocking, the white (black) part of the result is computed by multiplying the white (black) part of the blocked input with the other, gray, input. For the depth blocking, the result is computed by multiplying both white parts and both black parts and adding the results.

the non-linear part B prescribes what operations to perform on the blocks:

$$\begin{pmatrix} x_0 \\ \hline x_1 \end{pmatrix} \times \begin{pmatrix} y_0 \\ \hline y_1 \end{pmatrix} \xrightarrow{\mathrm{K}_{2\times 2}} \begin{pmatrix} x_0 \cdot y_0 \\ \hline x_0 \cdot y_1 \\ \hline x_1 \cdot y_0 \\ \hline x_1 \cdot y_1 \end{pmatrix} \qquad \begin{pmatrix} \mathbf{x}^U \\ \hline \mathbf{x}^L \end{pmatrix} \times \begin{pmatrix} \mathbf{y}^U \\ \hline \mathbf{y}^L \end{pmatrix} \xrightarrow{\mathrm{K}_{2\times 2} \otimes \mathrm{B}} \begin{pmatrix} \mathrm{B}(\mathbf{x}^U, \mathbf{y}^U) \\ \hline \mathrm{B}(\mathbf{x}^U, \mathbf{y}^L) \\ \hline \mathrm{B}(\mathbf{x}^L, \mathbf{y}^U) \\ \hline \mathrm{B}(\mathbf{x}^L, \mathbf{y}^L) \end{pmatrix}.$$

Comparing these two examples, $\mathcal{A} = \mathrm{P}$ yields a tensor product in which only corresponding parts of the input vectors are computed on, whereas $\mathcal{A} = \mathrm{K}$ yields a tensor product in which *all* combinations are computed on.

Recursive algorithms as OL breakdown rules. We express recursive algorithms for kernels as OL equations written as *breakdown rules*.

The first example we consider is a blocked matrix multiplication. While it does not improve the arithmetic cost over a naive implementation, blocking increases reuse and therefore can improve performance [27, 28]. We start with blocking along one dimension.

Fig. 3a shows a picture of a horizontally blocked matrix. Each part of the result $\boldsymbol{C}$ is produced by multiplying the corresponding part of $\boldsymbol{A}$ by the whole matrix $\boldsymbol{B}$. In OL, this is expressed by a tensor product with a Kronecker product:

$$\mathrm{MMM}_{m,k,n} \rightarrow \mathrm{K}_{m/m_b \times 1} \otimes \mathrm{MMM}_{m_b,k,n}\,. \tag{1}$$

Note that the number of blocks m/m_b is a degree of freedom under the constraint that m is divisible by m_b[3]; in the picture, m/m_b is equal to 2 (white block and black block).

Fig. 3b shows a picture of a vertically tiled matrix. The result is computed by multiplying parts of the matrix $\boldsymbol{B}$ with $\boldsymbol{A}$ so the underlying tensor product again uses a Kronecker product. However, since matrices are linearized in row-major order, we now need two additional stages: a pre-processing stage where the parts of $\boldsymbol{B}$ are de-interleaved and a post-processing stage where the parts of $\boldsymbol{C}$ are re-interleaved[4]:

$$\mathrm{MMM}_{m,k,b} \rightarrow \big(I_m \otimes L^n_{n/n_b}\big) \circ \big(\,\mathrm{MMM}_{m,k,n_b} \otimes \mathrm{K}_{1\times n/n_b}\,\big) \circ \big(I_{km} \times (I_k \otimes L^n_{n_b})\big). \tag{2}$$

[3] In general, blocks may be of different sizes and thus a more general blocking rule can be formulated.

[4] As we will explain later, stages may be fused during the loop merging optimization, so three stages do not necessary imply three different passes through the data. In this case, all stages would merge.

Finally, Fig. 3c shows a picture of a matrix tiled in the "depth". This time, parts of one input corresponds to parts of the other input but all results are added together. Therefore, the corresponding tensor product is not done with a Kronecker product but with a scalar product:

$$\mathrm{MMM}_{m,k,n} \rightarrow \left(\mathrm{S}_{k/k_b} \otimes \mathrm{MMM}_{m,k_b,n}\right) \circ \left((L^{mk/k_b}_{k/k_b} \otimes I_{k_b}) \times I_{kn}\right). \tag{3}$$

The three blocking rules we just described can actually be combined into a single rule with three degrees of freedom:

$$\begin{aligned}\mathrm{MMM}_{m,k,n} \rightarrow & (I_{m/m_b} \otimes L^{m_b n/n_b}_{m_b} \otimes I_{n_b}) \circ (\mathrm{MMM}_{m/m_b,k/k_b,n/n_b} \otimes \mathrm{MMM}_{m_b,k_b,n_b}) \\ & \circ ((I_{m/m_b} \otimes L^{m_b k/k_b}_{k/k_b} \otimes I_{k_b}) \times (I_{k/k_b} \otimes L^{k_b n/n_b}_{n/n_b} \otimes I_{n_b})).\end{aligned} \tag{4}$$

The above rule captures the well-known mathematical fact that a multiplication of size (m, k, n) can be done by repeatedly using block multiplications of size (m_b, k_b, n_b). Note that the coarse structure of a blocked matrix multiplication is itself a matrix multiplication. The fact that blocking can be captured as a tensor product was already observed by [29].

The second example we consider is the famous Cooley-Tukey fast Fourier transform (FFT) algorithm. It reduces the asymptotic cost of two-power sizes DFT_n from $O(n^2)$ to $O(n \log n)$.

In this case the OL rule is equivalent to a matrix factorization and takes the same form as in [26] with the only difference that the matrix product is written as composition (of linear operators):

$$\mathrm{DFT}_n \rightarrow (\mathrm{DFT}_k \otimes I_m) \circ \mathrm{diag}\,(c_i) \circ (I_k \otimes \mathrm{DFT}_m) \circ L^n_k, \qquad n = km. \tag{5}$$

Here, $\mathrm{diag}(c_i)$ is a diagonal matrix whose exact form is not of importance here [26].

As depicted in Fig. 4, this algorithm consists of 4 stages: The input vector is first permuted by L^n_m, then multiple DFT_m are applied to subvectors of the result. The result is scaled by $\mathrm{diag}(c_i)$ and finally again multiple DFT_k are computed, this time on strided subvectors.

Note that this algorithm requires the size n to be composite and leaves a degree of freedom in the integer factors[5]. Prime sizes require a different algorithm called Rader FFT [26].

Other examples of algorithms, written as OL rules, are presented in the end of the section.

Base cases. All recursive algorithms need to be terminated by base cases. In our case, these correspond to kernel sizes for which the computation is straightforward.

In the blocked multiplication case, the three dimensions can be reduced independently. Therefore, it is sufficient to know how to handle each one to be able to tackle any size.

[5] When the algorithm is applied recursively, this degree of freedom is often called the *radix*.

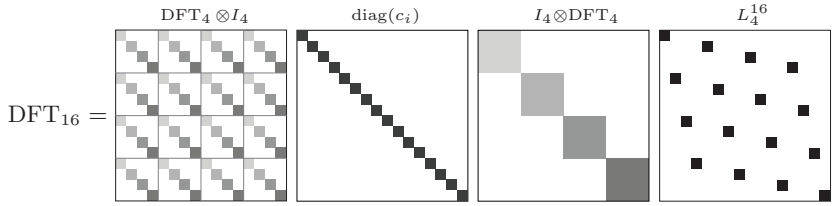


Fig. 4. Representation of the 4 stages of the Cooley-Tukey algorithm. Each frame corresponds to the matrix associated with one of the 4 operators from the equation (5), specialized with $n = 16$ and $m = 4$. Only non-zero values are plotted. Shades of gray represent values that belong to the same tensor substructure.

In the first two cases, the matrix multiplication degenerates into Kronecker products; in the last case, it simplifies into a scalar product:

$$\mathrm{MMM}_{m,1,1} \to K_{m \times 1}, \tag{6}$$

$$\mathrm{MMM}_{1,1,n} \to K_{1 \times n}, \tag{7}$$

$$\mathrm{MMM}_{1,k,1} \to S_k. \tag{8}$$

Note that these three rules are degenerate special cases of the blocking rules (1)–(3).

Other bases cases could be used. For instance, Strassen's method to multiply 2×2 matrices uses only 7 multiplications instead of 8 but requires more additions [29]:

$$\mathrm{MMM}_{2,2,2} \to \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \circ \mathrm{P}_7 \circ \left(\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \right) \tag{9}$$

Note that, due to the fact that blocking is a tensor product of two MMMs (4), the above base case can also be used in the structural part of the tensor, yielding a block Strassen algorithm of general sizes.

For the DFT of two-power sizes, the Cooley-Tukey FFT is sufficient together with a single base case, the DFT_2 nicknamed *butterfly*:

$$\mathrm{DFT}_2 \to \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{10}$$

Algorithm space. In this paper, we focus on implementing kernels of *fixed size*. In most applications that need to be fast, sizes are known at compilation time and therefore this generative approach is optimal because it removes all overhead. *General-size* code can also be generated from our domain specific language but it is mostly a different problem [30, 31].

We say that a formula is *terminated* or *maximally expanded* if it does not contain any kernel symbols. Using different breakdown rules or different degrees of freedom, the

same kernel can be expanded in different formulas. Each one of them represent a different algorithm to compute the kernel. The algorithm space can be explored using empiric search, strategies such as dynamic programming or machine learning algorithms [4].

For instance, we show here two different expansions of the same kernel, $\mathrm{MMM}_{2,2,2}$. They are generated by applying rules (1)-(3) and base cases (6)-(8) in two different orders and simplifying:

$$\mathrm{MMM}_{2,2,2} \to \big(\mathrm{S}_2 \otimes \mathrm{K}_{2\times 2}\big) \circ \big(L^4_2 \times I_4\big)$$
$$\mathrm{MMM}_{2,2,2} \to \circ\big((\mathrm{K}_{2\times 1} \otimes \mathrm{S}_2) \otimes \mathrm{K}_{1\times 2}\big).$$

We now present additional kernels and algorithms:

Circular convolution. The circular convolution [26] Conv_n is an arity $(2,1)$ operator defined by

$$\mathrm{Conv}_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n;\ (\mathbf{x},\mathbf{y}) \mapsto \big(\Sigma_{j=0}^{n-1} x_i y_{i-j \bmod n}\big)_{0\le i<n}.$$

Circular convolution can be computed by going to the spectral domain using DFTs and inverse DFTs:

$$\mathrm{Conv}_n \to \mathrm{iDFT}_n \circ \mathrm{P}_n \circ (\mathrm{DFT}_n \times \mathrm{DFT}_n). \tag{11}$$

Sorting network. A sorting network [32] sorts a vector of length n using a fixed (data-independent) algorithm. We define the ascending sort kernel χ_n and the descending sort kernel Θ_n:

$$\chi_n : \mathbb{N}^n \to \mathbb{N}^n;\ (a_i)_{0\le i<n} \mapsto (a_{\sigma(i)})_{0\le i<n},\ a_{\sigma(j)} \le a_{\sigma(k)} \text{ for } j \le k,$$
$$\Theta_n : \mathbb{N}^n \to \mathbb{N}^n;\ (a_i)_{0\le i<n} \mapsto (a_{\sigma(i)})_{0\le i<n},\ a_{\sigma(j)} \ge a_{\sigma(k)} \text{ for } j \le k.$$

The bitonic merge operator M_n merges an ascending sorted sequence $(a_i)_{0\le i<n/2}$ and a descending sorted sequence $(b_i)_{0\le i<n/2}$ into an ascending sorted sequence $(c_i)_{0\le i<n}$:

$$M_n : \mathbb{R}^n \to \mathbb{R}^n;\ (a_i)_{0\le i<n/2} \oplus_n (b_i)_{0\le i<n/2} \mapsto (c_i)_{0\le i<n} \text{ with } c_{i-1} \le c_i.$$

The bitonic sorting network is described by mutually recursive breakdown rules:

$$\chi_n \to M_n \circ \oplus_n \circ \big(\chi_{n/2} \times \Theta_{n/2}\big) \circ \mathrm{Split}_n \tag{12}$$
$$M_n \to \big(I_2 \otimes M_{n/2}\big) \circ \big(\Theta_2 \otimes I_{n/2}\big) \tag{13}$$
$$\Theta_n \to J_n \circ \chi_n \tag{14}$$

Finally, sorting the base case is given by

$$\chi_2 \to \oplus_2 \circ (\min_1 \times \max_1) \circ \mathrm{Fork}_2 \tag{15}$$

Viterbi decoding. A Viterbi decoder computes the most likely convolutionally encoded message that was received over a noisy channel [33]. The computational bottleneck of

the decoding is the *forward pass* $\mathrm{Vit}_{K,F}$ where F and K are the frame and constraint length of the message.

The algorithm structure in OL is:

$$\mathrm{Vit}_{K,F} \to \prod_{i=1}^{F} \left((I_{2^{K-2}} \otimes_j B_{F-i,j}) L^{2^{K-1}}_{2^{K-2}} \right) . \tag{16}$$

$B_{F-i,j}$ is called the *viterbi butterfly* and is a base case. This formula features the indexed composition and the indexed tensor whose subscripts describe the number of the current iteration. More details are available in [34].

Synthetic Aperture Radar (SAR). The Polar formatting SAR operator $\mathrm{SAR}_{s,k}$ computes an image from radar pulses sent by a moving sensor [35]. Multiple measurement are synthesized into one aperture. There are many algorithms and methods to reconstruct the image, and we focus on an interpolation- and FFT-based variant, called polar formatting. In this paper we only consider two high-level parameters: a scenario (geometry) $s = (m_1, n_1, m_2, n_2)$, and an interpolator $k = (k_r, k_a)$, which parameterize the SAR operator,

$$\mathrm{SAR}_{s,k} : \mathbb{C}^{m_1 \times n_1} \to \mathbb{C}^{m_2 \times n_2}.$$

The polar format SAR algorithm is defined by the following breakdown rules:

$$\mathrm{SAR}_{s,k} \to \mathrm{DFT}_{m_2 \times n_2} \circ \text{2D-Intp}_k \tag{17}$$

$$\text{2D-Intp}_k \to (\mathrm{Intp}^{n_1 \to n_2}_{k_a(i)} \otimes_i I_{m_2}) \circ (I_{n_1} \otimes_i \mathrm{Intp}^{m_1 \to m_2}_{k_r(i)}) \tag{18}$$

$$\mathrm{Intp}^{m \to n}_{(w,k)} \to G^{km \to n}_{w} \circ \mathrm{iDFT}_{km} \circ Z^{m \to km} \circ \mathrm{DFT}_m \tag{19}$$

Above, $Z^{m \to km}$ describes zero-padding, $G^{km \to n}_{w}$ non-uniform data gathering. Details are not essential for this paper and can be found in [36].

2.2 Abstracting Hardware Into Tags

Our goal is to automatically optimize algorithms by matching them to the target hardware. For portability, the actual computing *platforms* are abstracted behind simpler descriptions, the *hardware paradigms*. Paradigms capture essential properties of families of hardware. When more detailed properties are needed, one can always refer to the actual platform.

Paradigms. Paradigms are composable coarse structural descriptions of machines. They establish a set of properties that are common to certain classes of hardware. Actual platforms and instructions are abstracted behind this common layer. In Fig. 5 we show two examples: the single instruction multiple data (SIMD) vector instruction paradigm and the shared memory paradigm.

The *SIMD vector paradigm* models a class of processors with the following characteristics:

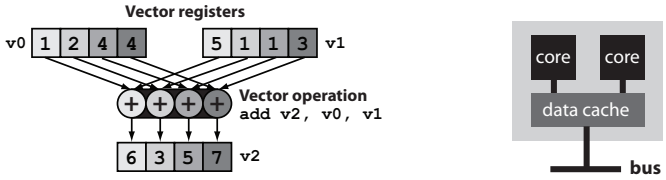


Fig. 5. The SIMD vector paradigm and the shared memory paradigm.

- The processor implements a vector register file and standard vector operations that operate pointwise: addition, multiplication and others. Using vector operations provide a significant speed-up over scalar operations.
- The most efficient data movement between memory and the vector register file is through aligned vector loads and stores. Unaligned memory accesses or subvector memory accesses are more expensive.
- The processor implements shuffle instructions that rearrange data inside a vector register (intra-register moves).

Most important examples of vector instruction sets are Intel's SSE family, the newly announced Intel extensions AVX and the Larrabee GPU native instructions, AMD's 3DNow! family, Motorola's AltiVec family including the IBM-developed variants for the Cell and Power processors.

The *shared memory paradigm* models a class of multiprocessor systems with the following characteristics:

- The system has multiple processors (or cores) that are all of the same type.
- The processors share a main, directly addressable memory.
- The system has a memory hierarchy and one cache line size is dominant.

Important processors modeled by the shared memory paradigm include Intel's and AMD's multicores and systems built with multiple of them. Non-uniform cache topologies are also supported as long as there is a shared memory abstraction.

Paradigms can be hierarchically composed. For instance, in single precision floating-point mode, an Intel Core2 Duo processor is characterized as two-processor shared memory system (cache line is 64 bytes). Both CPUs are 4-way SIMD vector units.

Besides these two paradigms, our framework experimentally[6] supports the following other paradigms: 1) distributed memory through message passing, 2) hardware streaming for FPGA design, 3) software streaming through multibuffering, 4) hardware-software partitioning, 5) general purpose programming on graphics card, and 6) adaptive dynamic voltage and frequency scaling.

Platforms. To ensure best cross-platform portability we capture most of the performance-critical structural information at the paradigm level, and avoid utilizing the ac-

[6] For the experimental paradigms, we can only generate a limited subset of kernels.

tual platform information within the algorithm manipulation rules. Each generic operation required by the paradigms has to be implemented as efficiently as possible on the given platforms.

For instance, we required any hardware covered by the vector paradigm to provide some instructions for data reorganizations within a vector (called vector *shuffles*). However, the actual capabilities of the vector units depend vastly on the vector instruction family and worse, on the version of the family. Therefore, it is not portable enough to describe algorithms using directly these instructions and we choose to abstract shuffles at the paradigm level. By doing that, each platform disposes of its own custom implementation of the same algorithm. Note that the task of deriving efficient platform-specific implementations of the shuffles required by the vector paradigm can be also fully automated, straight from the instructions specification [37].

The actual platform information is also used further down in our program generation tool chain. In particular, our compiler translating the domain-specific language into C code relies on it.

Paradigm tags. We denote paradigms using *tags* which are a name plus some parameters. For instance, we describe a shared memory 2-core system with cache line length of 16 `float` (64 bytes) by "$\mathrm{smp}(2, 16)$" and a 4-way `float` vector unit by "$\mathrm{vec}(4)$". Tags can be parameterized by symbolic constants: we will use $\mathrm{smp}(p, \mu)$, and $\mathrm{vec}(\nu)$ throughout the paper.

Tags can be concatenated to describe multiple aspects of a target hardware. For instance, the 2-core shared memory machine above where each core is a 4-way vector unit will be described by "$\mathrm{smp}(2, 16), \mathrm{vec}(4)$".

When compiling the OL formulas, the paradigms tags are replaced by the actual platform. For instance, the shared memory tag $\mathrm{smp}(2, 16)$ leads to multi-threaded code using OpenMP or pthreads, the vector tag $\mathrm{vec}(4)$ creates either Intel's SSE or Cell SPU code.

2.3 Common Abstraction: Tagged Operator Language

In this section we show how to build a space of algorithms optimized for a target platform by introducing the paradigm tags into the operator language. The main idea is that the tags contain meta-information that enables the system to transform algorithms in an architecture-conscious way. These transformations are performed by adding paradigm-specific rewriting rules on top of the algorithm breakdown rules. The joint rule set spans a space of different formulas for a given tagged kernel where all formulas are proven to have good implementations on the target paradigm and thus platform.

We first discuss the components of the system and then show their interaction with a few illustrative examples.

Tagged OL formula. A formula A tagged by a tag t, denoted

$$\underbrace{A}_{\text{t}}$$

expresses the fact that A will be structurally optimized for t which can either be a paradigm or an architecture. We have multiple types of tagged formulas: 1) problem specifications (tagged kernels), 2) fully expanded expressions (terminated formulas), 3) partially expanded expressions, and 4) base cases. These tagged formulas serve various purposes during the algorithm construction process and are crucial to the underlying rewriting process.

The input to our algorithm construction is a *problem specification* given by a tagged kernel. For instance,

$$\underbrace{\text{MMM}_{m,k,n}}_{\text{smp}(p,\mu)}$$

asks the system to generate an OL formula for a MMM that is structurally optimized for a shared memory machine with p processors and cache line length μ. Note that, while in the paper we always use variables, the actual input is a fixed-size kernel, and thus all variables have known values.

Any OL expression is a formula. The rewriting process continually changes formulas. A *terminated formula* does not have any kernel left, and no further rule is applicable. Any terminated formula that was derived from a tagged kernel is structurally optimized for the tag and is guaranteed to map well to the target hardware. Any OL formula that still contains kernels is only *partially expanded* and needs further rewriting to obtain an optimized OL formula.

A *base case* is a tagged formula that cannot be further broken down by any breakdown rule and the system has a paradigm-specific (or platform-specific) implementation template for the formula. The goal is to have as few base cases per paradigm as possible, but to support enough cases to be able to implement kernels based on them. Any terminated formula is built from base cases and OL operations that are compatible with the target paradigm.

Rewriting system. At the heart of the algorithm construction process is a rewriting system that starts with a tagged kernel and attempts to rewrite it into a fully expanded (terminated) tagged OL formula. The system uses pattern matching against the left-hand side of rewrite rules like (1) to find subexpressions within OL formulas and replaces them with equivalent new expressions derived from the right-hand side of the rule. It selects one of the applicable rules and chooses a degree of freedom if the rule has one. The system keeps track of the choices to be able to backtrack and pick different rules or parameters in case the current choice is not leading to a terminated formula. The process is very similar to Prolog's computation of proofs.

The system uses a combined rule set that contains algorithm *breakdown rules and base cases* (1)–(19), *paradigm base cases* (20)–(24), and *paradigm-specific manipulation*

rules (25)–(33). The breakdown rules are described in Section 2.1. In the remainder of the section we will describe the remaining two rule classes in more detail.

Base cases. Tagged base cases are OL formulas that have a known good implementation on every platforms covered by the paradigm. Every formula built only from these base cases is guaranteed to perform well.

We now discuss the base cases for the *shared memory* paradigm, expressed by the tag $\text{smp}(p, \mu)$. The tag states that our target system has p processors and cache length of μ. This information is used to obtain load balanced, false-sharing free base cases [5]. OL operations in base cases are also tagged to mark the base cases as fully expanded. In the shared memory case we introduce three new tagged operators, $\otimes_{\parallel}$, $\bar{\otimes}$, $\bigoplus_{||}$, which have the same mathematical meaning as their un-tagged counterparts. The following linear operators are shared memory base cases:

$$I_p \otimes_{\parallel} A_{m\mu\to n\mu}, \quad \bigoplus_{i=0}^{p-1}{}_{\parallel} A^i_{m\mu\to n\mu}, \quad M\bar{\otimes}I_\mu \text{ with } M \text{ a permutation matrix} \tag{20}$$

The first two expression encodes embarrassingly parallel, load-balanced computations that distribute the data so that no cache line is shared by multiple processors. The third expression encodes data transfer that occurs on a cache line granularity, also avoiding false sharing. The following non-linear arity (2,1) operators are shared memory base cases. They generalize the idea of $I_p \otimes_{\parallel} A_{m\mu\to n\mu}$ in (20),

$$\mathrm{P}_p \otimes_{\parallel} \mathrm{A}_{k\mu\times m\mu\to n\mu}, \quad \mathrm{K}_{q\times r} \otimes_{\parallel} \mathrm{A}_{k\mu\times m\mu\to n\mu} \text{ where } qr = p. \tag{21}$$

Building on these base cases we can build fully optimized (i.e., terminated) tagged OL formulas using OL operations. For instance, any formula $A \circ B$ where A and B are fully optimized is also fully optimized. Similarly, $\times$ allows to construct higher-arity operators that are still terminated. For instance,

$$(M\bar{\otimes}I_\mu) \times (N\bar{\otimes}I_\mu),\ M,\ N \text{ are permutation matrices} \tag{22}$$

produces terminated formulas. Not all OL operations can be used to build larger terminated OL formulas from smaller ones. For instance, if A is an arity (1,1) shared memory base case, then in general $A \otimes I_k$ is no shared memory base case.

Next we discuss the base cases for the *SIMD vector* paradigm. The tag for the SIMD vectorization paradigm is $\text{vec}(\nu)$, and implies ν-way vector units which require all memory access operations to be naturally aligned vector loads or stores. Similar to the shared memory base cases, we introduce two special markers to denote base cases. The operator $\hat{\otimes}$ denotes a basic block that can be implemented using solely vector arithmetic and vector memory access. Furthermore, the vector paradigm requires a set of base cases that have architecture-dependent implementations but can be implemented well on all architectures described by the SIMD vector paradigm. The exact implementation of these base cases is part of the architecture description. We limit our discussion to the Intel SSE instruction set, and mark such base with the following symbol,

$$\underbrace{A}_{\text{base(sse)}},$$

and imply the vector length $\nu = 4$. Other architectures or vector lengths would require similarly marked base cases with implementations stored in the architecture definition data base.

The base case library contains the implementation descriptions for the following linear arity (1,1) operators:

$$A_{m\to n}\hat{\otimes}I_\nu, \qquad \underbrace{L^{\nu^2}_{\nu}}_{\text{base(sse)}}, \quad \underbrace{\mathrm{diag}(c_i)}_{\text{base(sse)}}. \tag{23}$$

The formula $A_{m\to n}\hat{\otimes}I_\nu$ is of special importance, as it can be implemented solely using vector instructions, independently of $A_{m\to n}$ [38]. The generalization of the $A_{m\to n}\hat{\otimes}I_\nu$ to arity (2,1) is given by

$$A_{k\times m\to n}\hat{\otimes}\,\mathrm{P}_\nu\,. \tag{24}$$

Similar to the shared memory base cases, some OL operations allow to construct larger fully optimized (terminated) OL formulas from smaller ones. For instance, if A and B are terminated, then $A \circ B$ is terminated. In addition, if A is terminated, then $I_n \otimes A$ is terminated as well.

Paradigm-specific rewrite rules. Our rewriting system employs paradigm-specific rewriting rules to extract paradigm-specific base cases and ultimately obtain a fully optimized (terminated) OL formula. These rules are annotated mathematical identities, which allow for proving correctness of formula transformations. The linear arity (1,1) rules are derived from matrix identities [26], and non-linear and higher arity identities are based on generalizations of these identities.

Table 2 summarizes our shared memory rewrite rules. Using (25)–(29), the system transforms formulas into OL expressions that are built from the base cases defined in (20)–(21). Some of the arity (1,1) identities are taken from [5].

$$\underbrace{\big(I_k \otimes L^{mn}_n\big)}_{\mathrm{smp}(p,\mu)} \circ \underbrace{L^{kmn}_{km}}_{\mathrm{smp}(p,\mu)} \to \big(L^{kn}_k \otimes I_{m/\mu}\big)\bar{\otimes}I_\mu \tag{25}$$

$$\underbrace{L^{kmn}_{n}}_{\mathrm{smp}(p,\mu)} \circ \underbrace{\big(I_k \otimes L^{mn}_m\big)}_{\mathrm{smp}(p,\mu)} \to \big(L^{kn}_n \otimes I_{m/\mu}\big)\bar{\otimes}I_\mu \tag{26}$$

$$\underbrace{\mathrm{A}_{k\times m\to n} \otimes \mathrm{K}_{1\times p}}_{\mathrm{smp}(p,\mu)} \to \underbrace{L^{pn}_n}_{\mathrm{smp}(p,\mu)} \circ \big(\,\mathrm{K}_{1\times p} \otimes_\parallel \mathrm{A}_{k\times m\to n}\,\big) \circ \underbrace{\big(I_k \times L^{pm}_p\big)}_{\mathrm{smp}(p,\mu)} \tag{27}$$

$$\underbrace{\big(\,\mathrm{A}\times\mathrm{B}\,\big)}_{\mathrm{smp}(p,\mu)} \circ \underbrace{\big(\,\mathrm{C}\times\mathrm{D}\,\big)}_{\mathrm{smp}(p,\mu)} \to \underbrace{\big(\,\mathrm{A}\circ\mathrm{C}\,\big)}_{\mathrm{smp}(p,\mu)} \times \underbrace{\big(\,\mathrm{B}\circ\mathrm{D}\,\big)}_{\mathrm{smp}(p,\mu)} \quad \text{(if arities are compatible)} \tag{28}$$

$$\underbrace{\mathrm{A}\circ\mathrm{B}}_{\mathrm{smp}(p,\mu)} \to \underbrace{\mathrm{A}}_{\mathrm{smp}(p,\mu)} \circ \underbrace{\mathrm{B}}_{\mathrm{smp}(p,\mu)} \tag{29}$$

Table 2. OL rewriting rules for SMP parallelization.

Table 3 summarizes SIMD vectorization-specific rewriting rules. Some of the arity (1,1) identities can be found in [6]. These rules translate unterminated OL formulas into formulas built from SIMD base cases.

$$\big(\underbrace{A_{n\to n} \otimes I_m}_{\mathrm{vec}(\nu)}\big) \to \big(A_{n\to n} \otimes I_{m/\nu}\big)\hat{\otimes} I_\nu \tag{30}$$

$$\underbrace{\big(I_m \otimes A_{n\to n}\big) \circ L^{mn}_m}_{\mathrm{vec}(\nu)} \to \Big(I_{m/\nu} \otimes \underbrace{L^{n\nu}_\nu}_{\mathrm{vec}(\nu)} \circ \big(A_{n\to n}\hat{\otimes} I_\nu\big)\Big) \circ \big(L^{mn/\nu}_{m/\nu}\hat{\otimes} I_\nu\big) \tag{31}$$

$$\underbrace{L^{n\nu}_n}_{\mathrm{vec}(\nu)} \to \big(I_{n/\nu} \otimes \underbrace{L^{\nu^2}_\nu}_{\mathrm{base(sse)}}\big) \circ \big(L^n_{n/\nu}\hat{\otimes} I_\nu\big) \tag{32}$$

$$\underbrace{A_{k\times m\to n} \otimes \mathrm{K}_{1\times\nu}}_{\mathrm{vec}(\nu)} \to (A_{k\times m\to n} \otimes \mathrm{P}_\nu) \circ (\mathrm{dup}^\nu_k \times I_{m\nu}) \tag{33}$$

Table 3. OL vectorization rules.

Examples. We now show the result of the rewriting process for shared memory and SIMD vectorization, for DFT and MMM. We specify a DFT_{mn} kernel tagged with the SIMD vector tag $\mathrm{vec}(\nu)$ to instruct the system to produce a SIMD vectorized fully expanded OL formula; m and n are fixed numbers. The system applies the breakdown rules (5) and together with the SIMD vector-specific rewriting rules (30)–(33). The rewriting process yields

$$\begin{aligned}\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{vec}(\nu)} \to \big((\mathrm{DFT}_m \otimes I_{n/\nu})\hat{\otimes} I_\nu\big) \circ \underbrace{\mathrm{diag}(c_i)}_{\mathrm{base(sse)}} \\ \circ \big(I_{m/\nu} \otimes (I_{n/\nu} \otimes \underbrace{L^{\nu^2}_\nu}_{\mathrm{base(sse)}}) \circ (L^n_{n/\nu}\hat{\otimes} I_\nu) \circ (\mathrm{DFT}_n \hat{\otimes} I_\nu)\big) \circ \big(L^{mn/\nu}_{m/\nu}\hat{\otimes} I_\nu\big),\end{aligned}$$

which will be terminated independently of how DFT_m and DFT_n are further expanded by the rewriting system. A detailed earlier (SPL-based) version of the rewriting process can be found in [6].

Similarly, we tag DFT_{mn} with $\mathrm{smp}(p,\mu)$ to instruct the rewriting system to produce a DFT OL formula that is fully optimized for the shared memory paradigm. The algorithm breakdown rules (5) are applied together with the paradigm-specific rewriting rules, (25)–(29). The rewriting process yields

$$\begin{aligned}&\underbrace{\mathrm{DFT}_{mn}}_{\mathrm{smp}(p,\mu)} \to \big((L^{mp}_m \otimes I_{n/p\mu})\bar{\otimes} I_\mu\big) \circ \big(I_p \otimes_\parallel (\mathrm{DFT}_m \otimes I_{n/p})\big) \circ \big((L^{mp}_p \otimes I_{n/p\mu})\bar{\otimes} I_\mu\big) \\ &\circ \left(\bigoplus_{i=0}^{p-1}{}_\parallel D^i_{m,n}\right) \circ \Big(I_p \otimes_\parallel (I_{m/p} \otimes \mathrm{DFT}_n)\Big) \circ \big(I_p \otimes_\parallel L^{mn/p}_{m/p}\big) \circ \big((L^{pn}_p \otimes I_{m/p\mu})\bar{\otimes} I_\mu\big).\end{aligned}$$

Again, the above formula is terminated independently of how DFT_m and DFT_n are further expanded by the rewriting system. A detailed earlier (SPL-based) version of the rewriting process can be found in [5].

As next example we show the shared memory optimization of MMM. The kernel specification is given by $\mathrm{MMM}_{m,k,n}$ tagged by $\mathrm{smp}(p,\mu)$. The algorithm breakdown rules is (4) and the shared memory-specific rewriting rules are again (25)–(29). The rewriting process finds the following result

$$\underbrace{\mathrm{MMM}_{m,k,n}}_{\mathrm{smp}(p,\mu)} \rightarrow \mathrm{K}_{p\times 1} \otimes_{\|} \mathrm{MMM}_{m/p,k,n},$$

which cuts the first matrix horizontally and distributes slices among different cores. The rewriting process also discovers automatically that it could distribute jobs by splitting the second matrix vertically:

$$\underbrace{\mathrm{MMM}_{m,k,n}}_{\mathrm{smp}(p,\mu)} \rightarrow \big((L^{mp}_m \otimes I_{n/(p\mu)})\bar{\otimes} I_\mu\big) \\ \circ \big(\mathrm{K}_{1\times p} \otimes_{\|} \mathrm{MMM}_{m,k,n/p}\big) \circ \big((I_{km/\mu}\bar{\otimes} I_\mu) \times ((L^{kp}_p \otimes I_{n/(p\mu)})\bar{\otimes} I_\mu)\big).$$

Our final example is the ν-way vectorization of a $\mathrm{MMM}_{m,k,n}$. Using the matrix blocking rule (4) and the vector rules (30)–(33), the rewriting system yields

$$\underbrace{\mathrm{MMM}_{m,k,n}}_{\mathrm{vec}(\nu)} \rightarrow \big(\mathrm{MMM}_{m,k,n/\nu} \hat{\otimes} \mathrm{P}_\nu\big) \circ (\mathrm{dup}^\nu_{mk} \times I_{kn}).$$

3 Generating Programs for OL Formulas

In the last section we explained how Spiral constructs an algorithm that solves the specified problem on the specified hardware. The output is an OL formula that encodes the data flow, data layout, and implementation choices. In this section we describe how Spiral compiles this OL formula to a target program, usually C with library calls or pragmas.

Formula rewriting. An OL formula encodes the data flow of an algorithm and the data layout. It does not make explicit any control flow and loops. The intermediate representation Σ-OL (which extends Σ-SPL [39] to OL and is beyond the scope of this paper) makes loops explicit while still being a declarative mathematical domain-specific language. Rewriting of Σ-OL formulas allows in particular to fuse data permutations inserted by paradigm-specific rewriting with neighboring looped computational kernels. The result is domain-specific loop merging, that is beyond the capabilities of traditional compilers but essential for achieving high performance.

Σ-OL formulas are still only parameterized by paradigms, not actual architectures. This provides portability of domain-specific loop optimizations within a paradigm, as rewriting rules are at least reused for all architectures of a paradigm.

Σ-OL compiler. The final Σ-OL expression is a declarative representation of a loop-based program that implements the algorithm on the chosen hardware. The Σ-OL compiler (an extension of the SPL compiler [3] and the Σ-SPL compiler [39]) translates this expression into an internal code representation (resembling a C program), using rules such as the ones in Table 4. The resulting code is further optimized using standard compiler optimizations like unrolling, common subexpression elimination, strength reduction, and constant propagation.

Base case library. During compilation, paradigm-specific constructs are fetched from the platform-specific base case library. After inclusion in the algorithm, a platform-specific strength reduction pass is performed. Details on the base case library can be found in [38, 5, 37].

Unparser. In a final step, the internal code representation is outputted as a C program with library calls and macros. The platform description carries the information of how to unparse special instructions and how to invoke the requisite libraries (for instance pthreads). Code can be easily retargeted to different libraries by simply using a different unparser. For instance, an OpenMP parallel program and a pthreads parallel program have the same internal representation and the target threading facility is only committed to when unparsing the program.

4 Experimental Results

We implement the language and rewriting system inside GAP [40] and evaluate the code generator on the MMM, DFT, SAR imaging and Viterbi decoding problems. In each case, we generate optimized code and compare it to the best available implementations, which are often optimized in assembly. As we detail below, we achieve comparable performance on all scenarios. In some cases, human developers cannot spend the time to support all functionalities on all platforms. In these cases, having a code generator based approach is a huge win.

MMM. We evaluate our MMM code on two platforms: an Intel Xeon (Fig. 6a) and an IBM Cell BE (Fig. 6b). On the Intel platform, we compare the generated vectorized single threaded code to the Intel Math Kernel Library (MKL) 10.0 and the Goto BLAS 1.26 which are both hand-optimized in assembly [41, 42]. On the Cell, we compare the generated code that runs on a single SPU to the assembly code provided by D. Hackenberg [43]. Due to restrictions on the local store, [43] only provides code for sizes that are multiple of 64.

On both platforms, the generated vectorized code achieves 70% of the theoretical peak performance and is comparable to or slightly slower than hand-optimized code. The Cell platform offers a demonstration of the versatility of a code generator: due to the particularity of this architecture, matrix multiplications of sizes not multiple of 64 are simply not provided by high-performance experts; a code generator allows us to support any sizes that the user may be interested in.

operator formula	code
operators	
$\mathbf{r} = \mathrm{P}_n(\mathbf{x}, \mathbf{y})$	`for (i=0;i<n;i++)` `  r[i] = x[i]*y[i];`

$r = \mathrm{S}_n(\mathbf{x}, \mathbf{y})$	`r=0;`
	`for (i=0;i<n;i++)`
	`  r += x[i]*y[i];`

$\mathbf{r} = \mathrm{K}_{m\times n}(\mathbf{x}, \mathbf{y})$	`for (i=0;i<m;i++)`
	`  for (j=0;j<n;j++)`
	`  r[i*m+j] = x[i]*y[j];`

$\mathbf{r} = L^{mn}_m(\mathbf{x})$	`for (i=0;i<m;i++)`
	`  for (j=0;j<n;j++)`
	`  r[i+m*j] = x[n*i+j];`

higher-order operators	
$\mathbf{r} = (A \circ B)(\mathbf{x})$	`t = B(x);` `r = A(t);`
$(\mathbf{r}, \mathbf{s}) = (\mathrm{A} \times \mathrm{B})(\mathbf{x}, \mathbf{y})$	`r = A(x);` `s = B(y);`
$\mathbf{r} = (I_m \otimes A_n)(\mathbf{x})$	`for (i=0;i<m;i++)` `  r[in:1:(i+1)n-1]= A(x[in:1:(i+1)n-1]);`
$\mathbf{r} = (A_m \otimes I_n)(\mathbf{x})$	`for (i=0;i<m;i++)` `  r[i:n:i+n(m-1)]= A(x[i:n:i+n(m-1)]);`

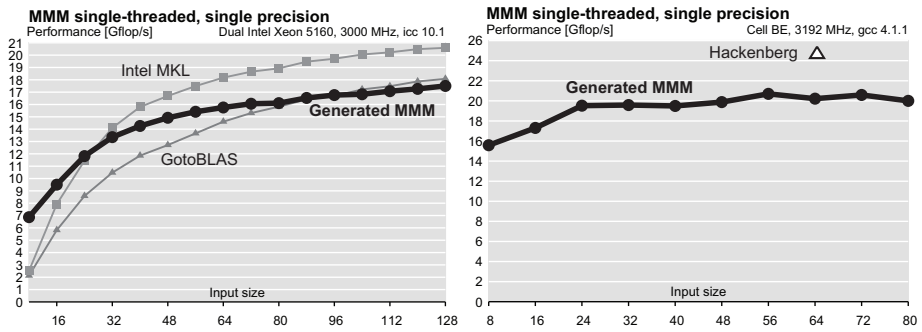
$\mathbf{r} = (\mathrm{P}_p \otimes A_{m\times n\to k})(\mathbf{x}, \mathbf{y})$	`for (i=0;i<p;i++)`
	`  r[ik:1:(i+1)k-1] =`
	`  A(x[i*m:1:(i+1)m-1],`
	`  r[i*n:1:(i+1)n-1]);`

$\mathbf{r} = (A_{m\times n\to k} \otimes \mathrm{P}_p)(\mathbf{x}, \mathbf{y})$	`for (i=0;i<p;i++)`
	`  r[i:k:i+k(p-1))] =`
	`  A(x[i:m:i+m(p-1))]`
	`  r[i:n:i+n(p-1)]);`

$\mathbf{r} = (\mathrm{K}_{p\times q} \otimes A_{m\times n\to k})(\mathbf{x}, \mathbf{y})$	`for (i=0;i<p;i++)`
	`  for (j=0;j<q;j++)`
	`  r[(i*p+j)k:1:(i*p+j+1)k-1] =`
	`  A(x[i*m:1:(i+1)m-1],`
	`  r[j*n:1:(j+1)n-1]);`

$\mathbf{r} = (A_{m\times n\to k} \otimes \mathrm{K}_{p\times q})(\mathbf{x}, \mathbf{y})$	`for (i=0;i<p;i++)`
	`  for (j=0;j<q;j++)`
	`  r[i*p+j:p*q:i*p+j+p*q*(k-1))] =`
	`  A(x[i:p:i+p(m-1))],`
	`  r[j:q:j+q(n-1)]);`

Table 4. Translating operator formulas to code. $\mathbf{x}$ and $\mathbf{y}$ denote the input and $\mathbf{r}$ and $\mathbf{s}$ the output vectors. The subscripts of A and B specify the signatures of the operators when they are relevant. We use Matlab-like notation: `x[b:s:e]` denotes the subvector of `x` starting at `b`, ending at `e` and extracted at stride `s`.



(a) on an Intel Xeon (b) on a Cell SPU

Fig. 6. Matrix Multiplication on two different platforms. All implementations are vectorized and single-threaded.

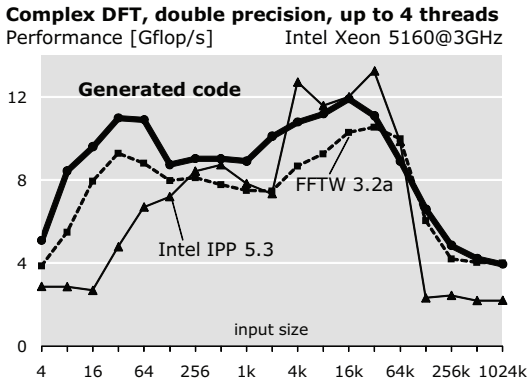


Fig. 7. Discrete Fourier transform (DFT) performance comparison between our generated code, Intel Performance Primitives (IPP) and FFTW. All implementations are vectorized and multi-threaded.

DFT. We evaluate our generated kernels for the DFT against the Intel Performance Primitives (IPP) and the FFTW library [1, 19]. While IPP is optimized in assembly, FFTW's approach shows some similarities with ours since small sizes are automatically generated.

Fig. 7 shows that our performance is comparable or better than both libraries for a standard Intel platform. All libraries are vectorized and multi-threaded.

SAR. We evaluate the generated SAR code on an Intel Core2 Quad server (QX9650). We observe a steady performance of 34 Gigaflops/sec for 16 and 100 Megapixel SAR images with runtimes of 0.6 and 4.45 seconds respectively. This performance numbers are comparable with [44] who developed hand-optimized assembly code for a Cell Blade.

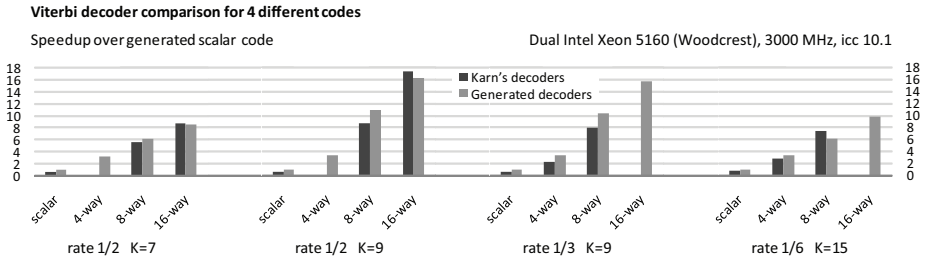


Fig. 8. Performance comparison between generated Viterbi decoders and hand-optimized decoders from Karn's Forward Error Correction (FEC) library. For each of the 4 different convolutional codes supported by the FEC library, Karn provides up to 4 implementations that vary in vector length. All performances are normalized to the scalar performance of the generated decoder.

Viterbi decoding. We evaluate our generated decoders against Karn's hand-written decoders [45]. Karn's forward error correction software supports four different common convolutional codes and four different vector lengths. For each pair of code and vector length, the forward pass is written and optimized in assembly. Due to the amount of work required, some combinations are not provided by the library.

In Fig. 8, we generated an optimized decoder for each possible combination and compared their performances to Karn's hand-optimized implementations. Analysis of the plot shows that our generated decoders have roughly the same performance than the hand-optimized assembly code from [45]. However, due to the code generator approach, we cover the full cross-product of codes and architectures. In particular, note that we are not limited to these four codes. An online interface is provided to generate decoders on demand [34, 46].

5 Conclusion

In this paper we presented OL, a framework to automatically generate high-performance implementations for a set of important kernels. Our approach aims at fully automating the implementation process, from algorithm selection and manipulation down to compiler-domain optimizations. It builds on and extends the library generation system for linear transforms, Spiral, to support multi-input/output and nonlinear kernels. The performance of the implementations our system produces rivals expertly hand-tuned implementations for the considered kernels. The approach is currently restricted to kernels with data-independent control flow, and we currently support limited functionality from a broad selection of domains. We plan to extend the approach to more kernels and domains in future work.

References

1. Intel: Integrated Performance Primitives 5.3, User Guide

2. Chellappa, S., Franchetti, F., Püschel, M.: How to write fast numerical code: A small introduction. In: Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE). Volume 5235 of Lecture Notes in Computer Science., Springer (2008) 196–259
3. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A language and compiler for DSP algorithms. In: Proc. Programming Language Design and Implementation (PLDI). (2001) 298–308
4. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232– 275
5. Franchetti, F., Voronenko, Y., Püschel, M.: FFT program generation for shared memory: SMP and multicore. In: Proc. Supercomputing. (2006)
6. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: High Performance Computing for Computational Science (VECPAR). Volume 4395 of Lecture Notes in Computer Science., Springer (2006) 363–377
7. GPCE: ACM conference on generative programming and component engineering
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
9. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: A case study. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(2) (2002) 191–214
10. Batory, D., Lopez-Herrejon, R., Martin, J.P.: Generating product-lines of product-families. In: Proc. Automated Software Engineering Conference (ASE). (2002)
11. Smith, D.R.: Mechanizing the development of software. In Broy, M., ed.: Calculational System Design, Proc. of the International Summer School Marktoberdorf, NATO ASI Series, IOS Press (1999) Kestrel Institute Technical Report KES.U.99.1.
12. Gough, K.J.: Little language processing, an alternative to courses on compiler construction. SIGCSE Bulletin **13**(3) (1981) 31–34
13. Bentley, J.: Programming pearls: little languages. Communications of the ACM **29**(8) (1986) 711–721
14. Hudak, P.: Domain specific languages. Available from author on request (1997)
15. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Dagstuhl Workshop on Domain-specific Program Generation. LNCS. LNCS (2004)
16. Taha, W.: Domain-specific languages. In: Proceedings of International Conference on Computer Engineering and Systems (ICCES) 2008. (2008)
17. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software (ATLAS). In: Proc. Supercomputing. (1998) `math-atlas.sourceforge.net`.
18. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. Int'l J. High Performance Computing Applications **18**(1) (2004)
19. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 216–231
20. Frigo, M.: A fast Fourier transform compiler. In: Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI). (1999) 169–180
21. Baumgartner, G., Auer, A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishanmoorthy, S., Krishnan, S., Lam, C.C., Lu, Q., Nooijen, M., Pitzer, R.M., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. of the IEEE **93**(2) (2005) special issue on "Program Generation, Optimization, and Adaptation".

22. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Orti, E., van de Geijn, R.: The science of deriving dense linear algebra algorithms. TOMS **31**(1) (March 2005) 1–26
23. Dershowitz, N., Plaisted, D.A.: Rewriting. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume 1. Elsevier (2001) 535–610
24. Nilsson, U., Maluszynski, J.: Logic, Programming and Prolog. 2nd edn. John Wiley & Sons Inc (1995)
25. Field, A.J., Harrison, P.G.: Functional Programming. Addison-Wesley (1988)
26. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)
27. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2) (2001) 3–35
28. Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P.: A comparison of empirical and model-driven optimization. Proc. of the IEEE **93**(2) (2005) special issue on "Program Generation, Optimization, and Adaptation".
29. Johnson, R.W., Huang, C.H., Johnson, J.R.: Multilinear algebra and parallel programming. In: Supercomputing '90: Proceedings of the 1990 conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (1990) 20–31
30. Voronenko, Y.: Library Generation for Linear Transforms. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2008)
31. Voronenko, Y., de Mesmay, F., Püschel, M.: Computer generation of general size linear transform libraries. In: Intl. Symposium on Code Generation and Optimization (CGO). (2009)
32. Batcher, K.: Sorting networks and their applications. In: Proc. AFIPS Spring Joint Comput. Conf. Volume 32. (1968) 307–314
33. Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. Information Theory, IEEE Transactions on **13**(2) (Apr 1967) 260–269
34. de Mesmay, F., Chellappa, S., Franchetti, F., Püschel, M.: Computer generation of efficient software Viterbi decoders : submitted for publication.
35. Carrara, W.G., Goodman, R.S., Majewski, R.M.: Spotlight Synthetic Aperture Radar: Signal Processing Algorithms. Artech House (1995)
36. McFarlin, D., Franchetti, F., Moura, J.M.F., Püschel, M.: High performance synthetic aperture radar image formation on commodity architectures. In: SPIE Conference on Defense, Security, and Sensing. (2009)
37. Franchetti, F., Voronenko, Y., Milder, P.A., Chellappa, S., Telgarsky, M., Shen, H., D'Alberto, P., de Mesmay, F., Hoe, J.C., Moura, J.M.F., Püschel, M.: Domain-specific library generation for parallel software and hardware platforms. In: NSF Next Generation Software Program workshop (NSFNGS). (2008)
38. Franchetti, F., Püschel, M.: Short vector code generation for the discrete Fourier transform. In: Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS). (2003) 58–67
39. Franchetti, F., Voronenko, Y., Püschel, M.: Loop merging for signal transforms. In: Proc. Programming Language Design and Implementation (PLDI). (2005) 315–326
40. The GAP Team University of St. Andrews, Scotland: GAP—Groups, Algorithms, and Programming. (1997) `www-gap.dcs.st-and.ac.uk/~gap/`.
41. Intel: Math Kernel Library 10.0, Reference Manual
42. Goto, K.: GotoBLAS 1.26, `http://www.tacc.utexas.edu/resources/software/#blas` (2008)
43. Hackenberg, D.: Fast matrix multiplication on Cell (SMP) systems, `http://www.tu-dresden.de/zih/cell/matmul`
44. Rudin, J.A.: Implementation of polar format SAR image formation on the IBM Cell Broadband Engine. In: Proc. High Performance Embedded Computing (HPEC). (2007)
45. Karn, P.: FEC library version 3.0.1, `http://www.ka9q.net/code/fec/` (Aug 2007)
46. de Mesmay, F.: Online generator for Viterbi decoders, `http://www.spiral.net/software/viterbi.html` (2008)