International Journal of Parallel Programming (2025) 53:8
https://doi.org/10.1007/s10766-025-00790-7



Automatic Heterogeneous Runtime Using Signal Processing Domain-Specific and Parallel Patterns

Yaseen Zaidi[1] · **Simon Winberg**[2]

Received: 5 July 2024 / Accepted: 13 February 2025
© The Author(s) 2025

Abstract
Parallel and signal processing patterns for large-scale radio data applications have been captured with a new domain-specific language (DSL), OptiSDR. The intermediate representations (IR) of the code are optimized at the frontend with the Delite compiler targeted for heterogeneous computing architecture (HCA). The design flow begins at the abstract programming model and, at a later stage, addresses tuning of the domain applications to the target HCA comprising GPUs and MCPU, as well as scheduling and optimal memory access. For a variety of signal processing use cases, the framework has shown up to two orders of magnitude higher computational speed than MPI and C++ and comparable performance to architecture-specific CUDA. Additional gains are ease of use, scalable computing, and efficient runtime.

Keywords Compiler · Domain-specific language (DSL) · Heterogeneous computing architecture (HCA)

1 Introduction

A new class of domain-specific languages (DSLs) aimed to solve problems at high abstraction using limited constructs and patterns natural to the particular domain is on the rise [1–4]. These languages are generally derivative of languages with more comprehensive constructs, and therefore high reusability in common tasks is instinctive. Developing a new DSL offers the challenges of complexity and knowledge of the new domain [5] to develop relevant semantics for the problem domain. These challenges are further aggravating since DSLs aim to achieve high productivity and high performance, and are more restrictive than general-purpose languages (GPLs)

✉ Yaseen Zaidi
yaseen.zaidi@cranfield.ac.uk

[1] Centre for Systems, Simulation and Analytics, Cranfield Defence and Security, Cranfield University, Defence Academy of the United Kingdom, Shrivenham SN6 8LA, UK

[2] Radar Remote Sensing Group, University of Cape Town, Rondebosch, Cape Town 7700, South Africa

Published online: 25 February 2025

but expressive in the particular domain, so a great deal of optimization, machine translation, and parallelism should be supported for the tenable goals that inspired the decision for the new DSL. The expressive power of a DSL enables abstractions that describe the aim rather than the computation itself, which results in concise and maintainable code. The code is usually intelligible enough to expose the underlying meaning of the application and the domain-specific tasks being tackled. On the other hand, the GPLs lack semantical information to map highly abstract program blocks to the hardware where the fine-grain code executes. This problem is usually handled through language extensions such as message passing interface (MPI), open multi-processing (OpenMP) that parallelize the program, and pthreads for multithreading the application. The downside of using such programming models is that the code is not easy to maintain, since it is tightly mapped to the machine resources.

This work was inspired by the several signal processing problems that arose in the radar remote sensing and radio astronomy signal processing problems in the radar and remote sensing group at the University of Cape Town. In these applications, the data characteristics, e.g., size and homogeneity, lead to selecting a handful of common signal processing algorithms. However, the computational architecture and compiler configuration lead to significantly different processing times. The engineering effort to understand the problem, the data, and mapping the compiler and hardware resources to each problem is an additional workload. Earlier we had shown the beginning of a new DSL called OptiSDR (optimal software-defined radio) and used it in distributing software-defined radio (SDR) computations to the available computing elements (CE) [6]. A major task in radar remote sensing applications is data compression, which was demonstrated in the NetRAD project [7]. We used additional parallel patterns through new data structures and class methods for windowing and filters [8]. The earlier work did not relate to compiler construction and optimization, a major step in developing high-speed applications. Therefore, the focus of the present work is on the gains in usability and productivity through the proper use of language semantics and compiler optimization to match the higher-order program to the underlying machine architecture. We generalize that in embedded DSL-based development (DSL internal to a hosting language), compiler frontend and intermediate optimizations outweigh the exhausting conventional backend optimizations comprising iterative space searching, scheduling, and machine-specific code generation. We tailor the open-source Delite parallel heterogeneous framework [9]. The parallel patterns and linear algebra-based digital signal processing (DSP) operations are expressed in a single dataflow type pipeline. In our approach, the developers are the domain experts, and so the usual knowledge and communication gap between otherwise disparate parties is inhibited. We show that if the domain is scoped out correctly, little effort can result in high productivity gain through a few new and reusable constructs.

2 The Need for a DSP DSL

A large-scale, DSP problem involves problem analysis and code development that suits the machine architecture and alleviate the compiler passes. The areas of top-down analysis are breaking the computation into chunks, understanding whether the computation should be executed in sequential or parallel, mapping code to hardware architecture and its resources, minimizing the matrix transformation, the iterative compiler passes for substituting the deferred types, and minimizing cache hits, to mention a few. Several aspects of these problems have been researched and solutions presented, but usually, literature addresses one or two aspects. We, however, address the problem in one go—from language design to machine resources. We regularly touch on the rich opportunities the DSP domain offers to develop semantics of a new DSL particularly adapted for signal processing and that exploits speed through parallelism and efficient use of hardware resources. Moreover, a compiler bridge between the program constructs and the hardware is needed that can tap the language semantics and machine architecture while translating the program. We show a compiler framework and its optimization.

2.1 Related Work

Feldspar [10] is a DSP domain DSL supporting vector indices data types. It is largely a derivative of Haskell functional language. The backend generator, Glasgow Haskell Compiler (GHC) emits C99 C applying the standard optimizations for dataflow-dominated software [11] [12]. Feldspar has been extended by a DSP library [13] to apply mathematical operations on data streams abstracted as a signal data type. Obsidian [14] is an embedded DSL suited for generating compute unified device architecture (CUDA) kernels for GPU targets. Outside the DSL domain, other works exist such as SIGNAL [15] and Lustre which are synchronous programming languages that are favorite for research and academia. The programming language transform description language (TDL) [16] has been used for waveform decomposition and polyphase filtering. SPIRAL [17] a functional language has been used for computing linear transforms. A unique feature of SPIRAL is automatic algorithm selection, i.e., machine learning. In the code compilation area, the work of Li [18] employs LLVM C/C++/Objective-C native compiler and libraries for developing a 5 G signal processing application. The target chip architecture is a mix of signal instruction multiple data (SIMD) for functional list processing and very long instruction word (VLIW) for parallelism. The LLVM compiler infrastructure can call OpenCL and OpenMP libraries. Khan et al. [19] have attempted to emit CUDA kernels by tuning the compiler. The OptiSDR language and framework draw from several of the concepts presented above in a single development environment.

Other efforts have given rise to Intel ArBB, Chapel, and the Ziria DSL for wireless systems programming. Accelerate is another domain-specific language for GPU targets. The Accelerate language uses Haskell syntax but compilation differs from Haskell and a CUDA package generates data structures for GPU. Similarly, Haskell embedded DSL implementations have been reported.

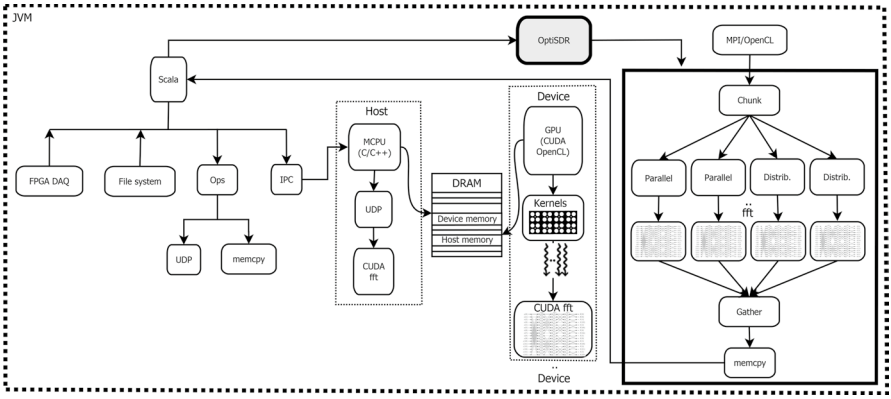


Fig. 1 The OptiSDR platform elements and interfaces. Besides GPUs and MCPU, the development methodology is agile to generate code from multiple GPLs for nodes of computer clusters due to compatibility with CUDA and MPI

2.2 OptiSDR

Illustrated in Fig. 1, the OptiSDR accepts either offline signal processing dataset or realtime stream through FPGA-based data acquisition (DAQ). The realtime need arises in NeXtRAD radar-based remote sensing [20]. A high-level, single source, specification code may be compiled by the OptiSDR embedded compiler and subsequently automatically deployed onto the target heterogeneous computing architecture (HCA) comprising GPUs and multicore central processing unit (MCPU). The compiler transforms the high-level SDR specification into the heterogeneous parallel binaries targeted for a particular configuration of HCA. The DSP description is entered in Scala under the java virtual machine (JVM) environment. Optionally, using the operating system services, the input realtime data from an FPGA-based acquisition system or a file of recorded samples may be read in. The description includes DSP constructs and parallel patterns. The compiler-passes optimize the intermediate representation (IR) of the program and tune the memory access patterns. The compiler queries the machine resources at walktime and schedules the program on the favorable machine while admitting any deferred dynamic optimizations at runtime.

In addition to the work noted in Sect. 2.1, there are other programming models and supported languages that too have achieved qualified success in parallel computing. These are, however, general-purpose, homogeneous, handcrafted, tend to be architecture-specific, and lack affinity to the particular domain. Examples are CUDA, Intel Cilk Plus, Intel MKL, and MapReduce. On the contrary, our approach is heterogeneous and exploits high abstraction and inheritance from sister DSLs. The programming relief obtained with the usability of OptiSDR outmatches the laborious CUDA. In Sect. 3 we will explain how domain knowledge is mapped to the parallel patterns. The framework and the compiler mechanics toward optimization are considered in Sect. 4.

Late dynamic tuning for best performance is presented in Sect. 5. The performance is assessed in Sect. 6 followed by the concluding remarks.

3 Domain Knowledge and Mapping Patterns

In large-scale signal processing problems, the complication lies in organizing the data or transforming the matrices in a fashion to reduce the computation overhead. In scientific numerical codes, the bulk of the computation time is spent in the linear solution stage. Whether the target architecture is parallel, distributed, or multicore, the optimization is chiefly concerned with loop (matrix) transformations techniques [21, 22] such as tiling, fusion, unrolling, partitioning, interchanging as well as with the reordering of the compiler directives by detecting dependencies [23, 24]. Both problems require developing a dependency graph constructed from memory references. The graph is represented in linear diophantine equations and inequalities whose variables are integers, i.e., of nested array subscripts or indices. The linearity of the program is necessary for the compiler to render optimized code.

In the DSP domain, the standard design and analysis methodology is decomposing a signal into harmonics to obtain finer spectral characteristics and reconstruction by obtaining the inverse transform. The DSP system is described in a recursive discrete structure comprising arithmetic operations. Depending on the data rate and in turn the number of inputs and outputs, the number of computations could be very high, and the computation speed is sought after for performance. Occasionally, the underlying linear system involves matrix entries with magnitudes less than the precision of the machine (sparsity). The data space is usually symmetric and correlated. The typical matrix algebra-based operations in signal processing codes e.g., Fast Fourier Transform (FFT), matrix transformation, and decomposition are repetitive and therefore computationally intensive. These features of the signal processing domain lend themselves to mapping to parallel computing.

The literature frequently treats one or two aspects of the aforementioned DSP problem domain. In the present work, we address the problem in unison with the OptiSDR language and compiler optimization.

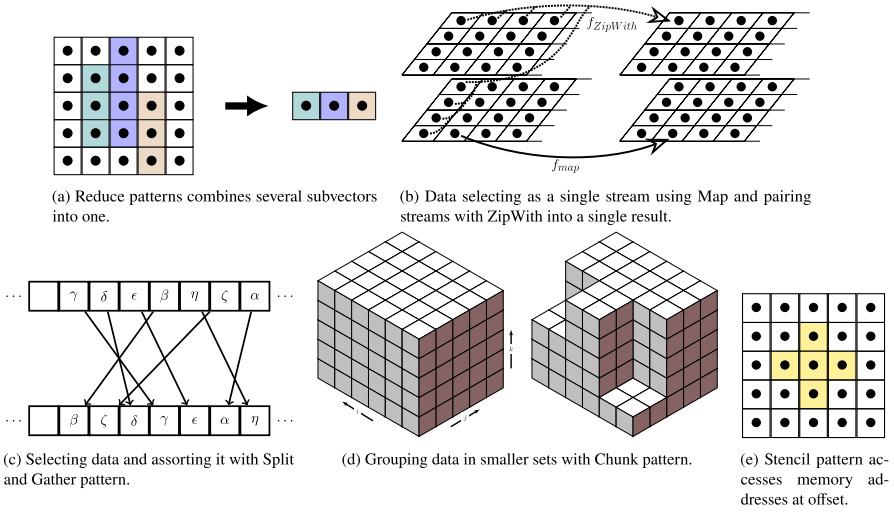
3.1 DSP Operations in Large-scale Aero/Astro Applications

The use cases in the target domain of OptiSDR are data acquisitions in radar remote sensing, radio astronomy, and interference identification. The observation collecting instruments in these applications measure signals at proximity with slight delays, and therefore correlation and alignment of the acquisitions is a primary concern. In radar applications, the useful information about the target is buried in the autocorrelation of the signal that carries an echo or reflected components off the clutter. The echo is either an interferer needing cancellation or identifies a sub-target. In the radio frequency interference (RFI) dataset, the dynamic range of interfering frequencies could be quite wide. For example, the RFI analysis in the acquisitions made by

South African Radio Astronomy Observatory (SARAO) telescopes takes into consideration the signals from low earth orbiting and geostationary satellites, aircraft transponders as well as cellular signals, and electromechanical equipment operating in the nearby farms in the Karoo region. To isolate spurs, frequency sampling finite impulse response (FIR) filters are generally used to isolate the resonant frequency content in narrowband and wideband ranges. These are parallel filterbank structures that can split data according to frequencies and then apply filtering in parallel to the whole dataset. Similarly, the computational noise in Discrete Fourier Transform (DFT) results in spectral leakage which might as well contain useful radio astronomical content. Such leakage can be reduced by applying polyphase filtering, i.e., windowing data then chunking, adding chunks element-wise, and then applying DFT. Although this implementation has a higher computational cost than FFT (1.5×) the frequency response tends to be flatter, and the spectrum is preserved. Many other broadband spectral applications in which multi-rates are inherent such as spectrometry, radar, and radio astronomy employ parallelization in a channelizer.

3.2 Scoping the DSP Domain

Many DSP structures exhibit regularity to exploit recursion and the algorithms are limited to convolution, FFT, windowing, and filtering. Computationally, windowing is similar to filtering, and convolution and correlation are similar to FFT. These matrix multiplication problems suit several parallel computation concepts [25–27] of mapping, gathering, reducing, chunking, factoring, zipping, and stencil.



(a) Reduce patterns combines several subvectors into one.

(b) Data selecting as a single stream using Map and pairing streams with ZipWith into a single result.

(c) Selecting data and assorting it with Split and Gather pattern.

(d) Grouping data in smaller sets with Chunk pattern.

(e) Stencil pattern accesses memory addresses at offset.

Fig. 2 Parallel patterns adopted in OptiSDR from the functional programming model. The patterns can be combined for complex signal processing tasks

3.3 Parallel Patterns

We reuse Map, Reduce, and ZipWith patterns [9] which are most relevant for DSP operations on large data. Depicted in Fig. 2 a set of intermediate representation (IR) nodes are defined to transform the high-level DSL code into parallel executable patterns.

These patterns are combined with different DSP operations and executed in the synchronous dataflow (SDF) model of computation.

- Map- The Map pattern having roots in functional languages is an "apply-all" higher-order operation. All data in a list or input sampled signal $\overrightarrow{x_i}$ as a discrete event $e_i(t, v)$ of time t and amplitude v is applied with a singular mapping function f_{map} for an output $\overrightarrow{y_o}$, preserving their order: $f_{map}(x_0, x_1, \ldots x_n) \implies \overrightarrow{x_i} \mapsto \overrightarrow{y_o} = (y_0, y_1, \ldots y_n)$. The mapping function is any arithmetic, boolean, test condition expression, a simple mathematical assignment, shifting of bits, slicing, returning whether data is even, odd, or prime, and returning size or precision of data, or a complex DSP operation.

- Reduce- The Reduce pattern is a binary tree reduction operation that is independent of order (associative) or grouping (commutative). The pattern produces a single result. Reduce may take the form of an arithmetic function or a DSP operation. Many DSP operations (filtering and channelization, convolution, windowing, decimation, interpolation, mixing, and heterodyning) require two inputs. For instance, the Hilbert transform $\mathcal{H}(\omega) = -sgn(\omega)\mathcal{F}(\omega)$ where $sgn(\omega) = -1, 0, 1$ at $\omega < 0, = 0 > 1$)) and cross-correlation compute FFT on data from two sources. In such computations, A and B are multiplied through Reduce. Note that the Hilbert transform is a multiplication of the Fourier transform with signed frequencies. Moreover, Reduce pattern may perform any boolean, statistical, or arithmetic operation.

- MapReduce - A paired stream is mapped and then reduced: $f_{MapRreduce}(x_0, x_1, \ldots x_n,\ u_0, u_1, \ldots u_n) \implies (\overrightarrow{x_i}, \overrightarrow{u_i}) \mapsto \overrightarrow{w_i} \ni \overrightarrow{y_j} = (y_0, y_1, \ldots y_m)$ where $j < i$ and $m < n$. For instance, in a prefiltering followed by decimation operation, reduction may be accomplished to downsample a signal. Similarly, reduce operation may be applied for tree reduction in FFT butterflies.

- ZipWith- Pairing of two data streams is done by ZipWith for operations that take in two signals e.g., correlation, convolution, windowing, moving averages and filtering: $f_{ZipWith}(x_0, x_1, \ldots x_n,\ u_0, u_1, \ldots u_n) \implies (\overrightarrow{x_i}, \overrightarrow{u_i})$. Similarly, real and imaginary parts of complex numbers can be sequenced into pairs. A channelizer or a digital downconverter converts a radio frequency (RF) to intermediate frequency (IF). The operation may be broken down into splitting a broadband signal into narrowband spectra, filtering the data, and aggregating the frequency responses. This chain may be implemented in ZipWith and MapReduce.

- Split and Gather - Several DSP algorithms transpose matrices and spice data into different, smaller matrices or vectors. For example, during FFT index shuffling, the receiving indices or addresses are tracked (split) and at the gathering write addresses are carefully mapped to avoid contention: $f_{Split}(x_0, x_1, \ldots x_n) \mapsto (x_{n-1}, x_0, \ldots x_{n-m}) \implies f_{Gather} \mapsto (x_1, x_{n-m}, \ldots x_{n-1})$. Similar needs arise in image processing that requires transposing of pixel matrices.

- Chunk - The chunk pattern carves up the dataset into smaller sizes to be allocated for processing to different computing units, e.g., a multiprocessor within a GPU, hardware threads or CUDA streams: $f_{Chunk}(x_0, x_1, \ldots x_n, u_0, u_1, \ldots u_m) \mapsto (x_0, x_1, \ldots x_p, u_0, u_1, \ldots u_q)$ where $p << n$ and $q << m$. Since the cache of a multiprocessor is very small, to maintain speed it is sensible to exploit data locality by splitting the chunks into data that fit the cache. Parallelism is achieved since each multiprocessor executes blocks of threads independently of the other.

- Stencil - The stencil pattern acts on the neighboring data samples as in overlapping operation during convolution or selecting neighboring pixels in pattern recognition: $f_{Stencil}(x_0, x_1, \ldots x_m, u_0, u_1, \ldots u_n, w_0, w_1, \ldots w_p) \mapsto (x_0, x_1, \ldots x_i, u_0, u_1, \ldots u_j, w_0, w_1, \ldots w_k)$ where $i < m, j < n$ and $k < p$. Stencil is a repetitive pattern, generally useful on input data geometries with either spatial or temporal overlapping. For instance, Stencil may be iterated in folding, displacing, multiplication, and integration operations, i.e., the convolution algorithm. Similarly, in image processing, convolution requires neighborhood pixels of an input pixel. Time-domain filtering in OptiSDR employs a convolution filter implemented as an FIR filter with data access through the Stencil.

The OptiSDR developer can easily develop new patterns through the combination of existing patterns. For example, in multiplications or transformations, ZipWith may be used to establish a correspondence between the elements of the rows and columns of the matrices, Reduce to produce the dot product result, and Reduce to sum the dot products of a particular row and a column. Lastly, Map will repeat the operations on all pairs of rows and columns and store the overall result of the row and column mapping in a new matrix.

3.4 Iterative Patterns

The primary construct for iteration in OptiSDR for a parallelized loop is OptiSDROpParWhile whereas for parallel stream processing the pattern is OptiSDROpPSPP. These parallel indexed loops are suitable for SDR operations exploiting parallel loops in which the elements impose no dependencies on each other, for instance, the in-phase and quadrature (IQ) streams in quadrature phase-shift keying (QPSK) modulated transmissions or orthogonal frequency-division multiplexing (OFDM)

transmission. Many algorithms incur data dependencies and pipeline stalls. For example, the least square error filters, adaptive filters, echo cancelers, systolic arrays, channel equalization, estimators and predictors, and system identification. In these cases, the OptiSDR simply transforms high-level constructs for embarrassingly parallel and multi-threaded asynchronous execution. This is achieved with Map and asynchronous parallel loops.

3.4.1 Optimizing FFT

The ecumenical FFT is one of the main algorithms in radar [28] and radio astronomy [29] signal processing [30]. Generally, in efficient FFT algorithms, 1-D input and output vectors are mapped to 2-D with their addresses shuffled. The address shuffling problem for reorganizing the data is a Map operation that is combined with any suitable pattern for DFT on column entries, element-wise complex multiplication, and DFT of row entries.

The CUDA C parallel FFT (cuFFT) implementation, being domain agnostic, requires a comprehensive plan specific to the architecture for computing the FFT. The plan includes information about multiple kernel launches, the dimension of FFT, the number of parallel transforms, stridden data layouts, implicit memory allocation, memory transfer to CUDA device, FFT execution, and memory transfer back to the host MCPU system, and releasing of computing resources. Typically, to keep memory latency small, minimal stride distance is preferred in input and output data [31]. While the CUDA planning style is granular and more suitable for massively parallel hardware, it is overexertion for OptiSDR target domains and increases effort. We use the following improvements in OptiSDR. The plan simply passes the FFT length and chunk size. We split the computation into two chunks for two GPUs. Using more GPUs does not necessarily speed up computation because the frequent data transfers between the GPU device and the MCPU host (D2H/H2D) impose an overhead on the PCIe bus communication as well as the allocation and deallocation of memory and execution of the overall computation plan. Unit strides are used when successive samples are expected to contain unique information in the spatial domain. In a 1-D transform, every element is used. The ith input signal has the memory allocation as $input[i * idist + x * istride]$ so $istride$ is set to 1 and $idist$ denotes distance between the input chunks. If the samples are very closely correlated, the stride can be increased to skip samples. This abstraction reproduces downsampling in a digital down conversion at the input or the output or RF to IF conversion. Another advantage of using a large stride is the efficiency in the locality of spatial data, which is necessary for the management of small memory in GPUs.

To speed up, standard FFT numerical codes (Cooley-Tukey, Good-Thomas, and Winograd) for DFT and convolution implementations employ linear transformation of index [32] to map the original large 1-D computation to a multidimensional one [33] with smaller indices (N_1, N_2..). That is, the FFT is computed as decimation in time. The summation is mapped to shorter even $\sum_{n=0}^{N/2} / - 1 \quad x(2n)e^{-2j\pi/N}$ and odd $\sum_{n=0}^{N/2} / - 1 \quad x(2n+1) \quad e^{-2j\pi/N}$ sequences. The efficiency is $\mathcal{O}(n\log n)$. The implementations with input size given as smaller prime are known to be faster. OptiSDR

uses prime 2 which is the fastest. When the index factors are not relatively prime [34], i.e., multiples of powers of a prime between 2 and 127, the Bluestein algorithm [35] is used at the cost of more arithmetic operations. Real to complex forward FFT and complex to real inverse FFT are supported on double precision data.

3.4.2 Hilbert Transform

To make the mathematics simpler, in many DSP applications it is often convenient to re-cast a real signal such as obtained from a radar antenna to an equivalent complex helical with phase and radius equal to amplitude using Hilbert transform [36, 37]. The time variation of phase allows the determination of the instantaneous frequency of a monocomponent or chirp. The instantaneous frequency is a figure of interest in Doppler radar applications to determine the trajectory of a moving target through the history of range, velocity, and acceleration.

To sum up, Table 1 lists important DSP operations and corresponding parallel patterns developed in OptiSDR.

4 OptiSDR DSL Integration Framework

The semantical characteristics of OptiSDR discussed in Sect. 3 relate to the program design. The development framework applies optimization at different passes to speed up execution. Delite is a lightweight compiler framework that hosts several DSLs built upon shared principles and constructs. The penalty is paid early in compilation, but the runtime is fast. The elements of the compiler are developed to complement the signal processing semantics in the OptiSDR language. We now explain the mechanics of the framework and compiler optimization to

Table 1 DSP ops and corresponding parallel patterns

DSP Op	Description	Parallel Pattern
fft	Fast fourier transform	MapReduce
ifft	Inverse fast fourier transform	MapReduce
hilbert	Hilbert transform	FFT and ZipWith
xcorr	Cross-correlation	FFT and ZipWith
conv	Convolution	FFT and Stencil
fir	Finite impulse response filter	Convolution-Remez and Stencil
mathematical	Absolute, conjugate, rounding, power, trigonometric, statistical	Map
fftshift	Shift FFT output	Map
dft	Discrete fourier transform	Nested loops
Lnspace	Linear spaced elements	Map

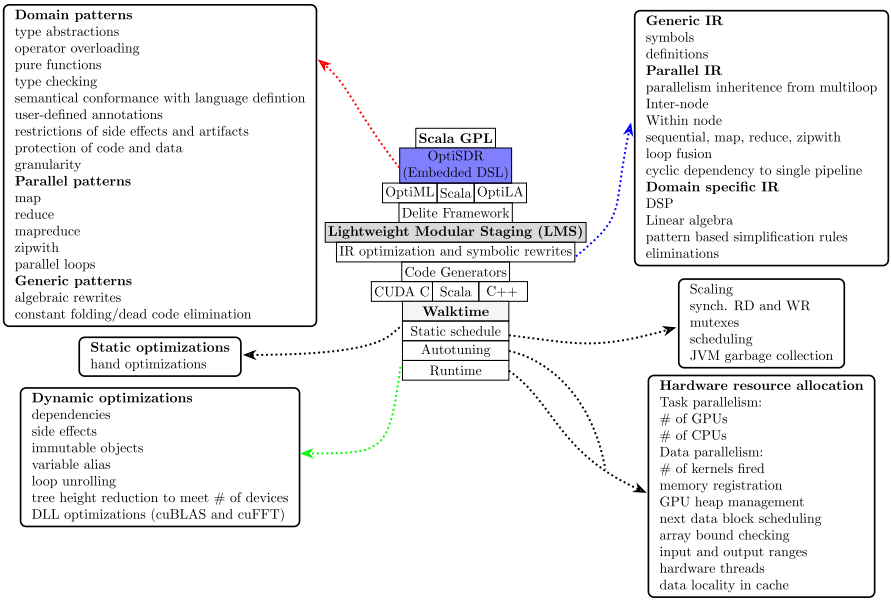


Fig. 3 OptiSDR framework, backend, and frontend layers and static/dynamic optimization steps using generic and domain-specific intermediate representations and tuning of the runtime. The framework achieves agility through the use of scaffold of libraries and staging by refining constructs late in the compilation

obtain efficiency. This computing domain has been a key learning outcome for us in the radar and remote sensing research group to leverage the speed and obtain the advantage of the DSL. This paper is the result of our experience on the learning ladder. The framework architecture is illustrated in Fig. 3.

4.1 Scala and Domain-Specific Data Types

The application developer of the embedded OptiSDR DSL captures the description at the highest level of abstraction, i.e., in the Scala language that allows exploiting abstract constructs and data types, indicated by the red arrow in Fig. 3. The main advantages of using Scala as the entry point (Fig. 1) are productivity gain to developers in accessing a multitude of libraries, tools, and services (interprocess communication, file system, user datagram protocol) without having to write wrappers. Since Scala is a declarative functional language, it is natively suitable for parallelizing computation on immutable large data. Scala's trait concept allows abstract member types. Moreover, user member types can be easily extended, which is a necessity for embedding a DSL. Sister DSLs, OptiML (machine learning) and OptiLA (linear algebra) [38], offer compatibility and additional expressive power to OptiSDR. Besides the linear algebra-related parameterized data types of OptiLA, the new data types include complex types ComplexDenseVector vectors and ComplexDenseMatrix matrices which are used to represent the baseband signals, often in the form of

I/Q samples. The default numerical data type for each element of a complex vector and matrix container is the float type. The return of real, imaginary, conjugate, absolute, slicing, and element-wise addition and multiplication operations are possible.

Alongside the domain-specific data structures, the OptiDSR developer may write appropriate parallel ops on the data samples, e.g., DeliteOpMap, and DeliteOpZipWith. At this phase, any possible optimizations will be exploited to concise the code, such as constant folding and algebraic rewrites. The code thus created when executes generates IR nodes. The IR nodes are scheduled and presented to the code generators for the compiler backend processing.

4.2 Consideration for the Side Effects

The patterns Map and ZipWith are order independent of their elements and highly concurrent, do not incur synchronization problems, and therefore side effects free. Reduce is both associative and commutative; if the order of input is not handled correctly, the result might be inaccurate. This is most certainly true in matrix multiplications involving floating point additions and dot products that are not purely associative.

4.3 Staging

Central to compiler construction is the late integration of the libraries. The frontend of the compiler is based on lightweight modular staging (LMS) shown as a gray block in Fig. 3. The LMS [39] is Scala's refinement methodology to integrate libraries to design code generators. The LMS refines abstractions and uses types within a single semantics of code to discern between the generated code and the logic that generated it. The generator treats the meta code as data, i.e., uses annotations to convert program constructs to a syntax tree of symbol data structures. While doing so, certain restrictions are applied to ensure minimal side effects, most notably the safety of types at runtime. The LMS approach is aimed at deferral execution [40] of selected parts of the code in a *staging* manner. This is a notion highly desirable since the code is oblivious to the size of the task and the data (loop bounds and array indices). Moreover, since types are abstract, they will be dynamically optimized and cast concrete later than more information becomes available during staging toward the runtime. For example, to evaluate the condition $5^3 < x^7 < 4^4$, the generated code is partially optimal since 5^3 and 4^4 are treated as static constants and 7 is a compile-time deferred constant consumed at runtime by the power function. These types can be overloaded for vector computations.

The recursive DSP computations, e.g., digital infinite impulse response (IIR) filters enormously benefit from staging. In such cases, it is sometimes possible to separate future and intermediate computations through different annotations on uninstantiated objects. Similarly, to avoid the code size getting bigger, recursion may be exited through a certain annotation, e.g., when the impulse response has sufficiently subdued.

4.4 Symbolic Rewrites

Once LMS staging has been applied, the domain-specific and generic optimizations are taken up. The aspects of IR optimization are illustrated with the blue arrow in Fig. 3. Significant savings can be achieved in generic IRs. The data structures are written to a table of symbols that are optimized through pattern matching during compiler passes at the frontend (lexical, syntactic, and semantic) and the backend (intermediate code, symbolic optimizations, intermediate representation). The optimization passes save up on the expensive computations that would decelerate the runtime. This is accomplished by algebraic rewrites, e.g., the operations on a variable x such as $3 \times x \rightarrow x + x + x$, $x/4 \rightarrow x \times 0.25$, $x^2 \rightarrow x \times x$.

We can further improve the rules for the generator or even optimize by hand; examples are additions with the identity matrices and multiplications with the null matrices, redundant subexpressions, and extraneous floating point arithmetic, i.e., $a + 0.0 \rightarrow a$, $a \times 0.0 \rightarrow 0.0$, $a_{i,j} < \{machine\ precision\} \rightarrow a_{i,j} = 0.0$ and $a_{i,j} \approx 1.0 \rightarrow a_{i,j} = 1.0$.

Therefore, leveraging highly precise and numerically accurate data to sparse entries. Other standard optimizations are also performed: constant folding $C_1 + C_2 \rightarrow C$. Besides the canonical transformations, more aggressive rules may be specified, depending on where the domain-specific numerical code spends time during execution.

4.5 Parallel IR Optimization and Code Generation

Once local optimizations are applied, the compiler generates an IR of the dataflow nodes. The domain-specific, parallel and generic patterns are applied to exploit Delite op. The parallel IRs exhibit internal or external parallelism. E.g., DeliteOpReduce will be a nice pipeline IR while DeliteOpZipwith can parallelize matrix multiplications and DeliteOpMap can parallelize both inner and outer loops. Generic IR will produce a DeliteOpSequential IR and would break down the three-address code. Recall that each IR node is a symbol. If the code has been constructed free of side effects, the IR will be free of it also. If symbol definitions exist (pattern matched), they are reused for elimination.

A finished IR is traversed to check what global optimization is possible. At this stage, the abstraction view is closer to implementation. The loops and arrays are further optimized for memory patterns, data locality, and fetch times. The loops are fused to eliminate temporaries and dependencies, and data is chunked to fit in the cache registers (loop tiling). Once canonical, the entire IR is constructed into a dataflow graph that is presented to the code generator, which retains Scala or generates CUDA or C++ (Fig. 3).

5 Walktime and Runtime

At walktime [41] when the statically optimized code is loaded, the IR dataflow graph contains superimposed dependency information. Also contained in the IR are the kernel opcode for the runtime, parallel opcodes which may require refinement at runtime, and the data structures with associated function definition for copying data between the JVM environment and the CEs (MCPU and GPU). At this stage, all the encoded information in the dataflow graph is either handcrafted or compiler-generated. A device query service detects the hardware limitations, e.g., the number of GPUs, each GPU's global memory, each GPU's compute capability, MCPU system memory, and the number of cores. However, the user may overwrite the result of the query, for example, the user may force the application to run on a single GPU. The walktime then combines the static graph with machine resource information and schedules the graph. All schedules are statically determined as branches in the graph are resolved, and the partial schedules are incrementally applied. The easiest way to resolve dependencies is to schedule those ops on the next available resource.

5.1 Considerations for Domain-Tuned Execution

In the backend, our focus is on tuning the memory access, i.e., the optimization goal is data locality. A GPU target suits low latency applications that achieve the intensity of data-parallelism and outdo the control and communication overheads. If the application is regular, it is likely to be local as the data addresses will be well-defined on a geometric affine set or will be contiguous as well the data structures will be static. The data could easily fit in the cache line when it is pulled in. The application could work at the L1 cache level. If the application exhibits irregularity, i.e., it is time asynchronous and data spatial, the data structures are dynamic and computation is temporally spread or the time to solution point is indeterminable. Consequently, there will be hits to the shared memory and fully coherent L2, which would increase inter-thread communications overhead. In irregular algorithms [42], the data addresses may not be known at walktime and the compiler must aggressively work at runtime to determine them. In large data signal processing, we expect the runtime to mostly scheduling the GPUs among the available CEs. In the domains targeted for OptiSDR, a majority of the algorithms are discrete structures in which the mathematical intensity is high and samples express high regularity in time and storage (computational fluid dynamics applications are also largely regular with a few exceptions e.g., Kelvin-Helmholtz instability arising due to temporal and spatial granularity) and are independent to be chunked and distributed to different cores within a streaming multiprocessor of the GPU.

5.2 Space Splitting and Access Tuning

Each higher level of the cache hierarchy is much larger than the immediately lower level. The major bottleneck is the L1 efficiency in terms of occupancy, hits, and misses, and this has been a captivating research topic on computer architectures and compilers. The L1 optimization further boils down to determining the optimum tile

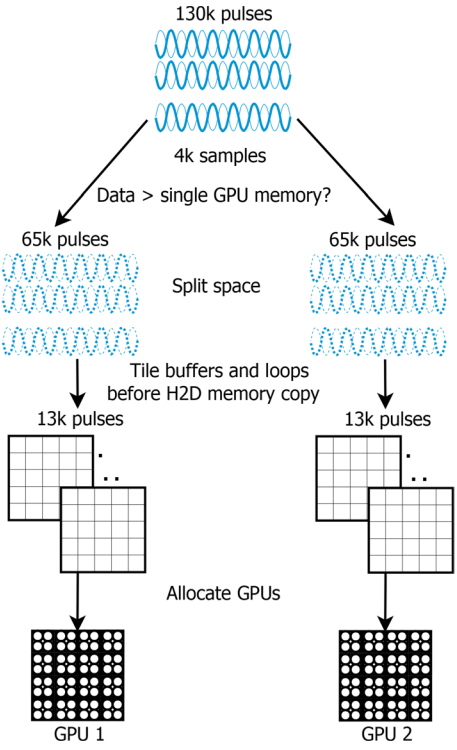


Fig. 4 Space split computation for compressing a radar pulse

size by trading off multiple objectives such as cache utilization, self and cross-conflict, data replacement strategies, and cost. However, the excessive complexity of tile size computation and accuracy are not justified and are usually expensive.

Figure 4 shows a typical problem size exceeding (10×) standard memory configuration on the Fermi GTX 480 device (1536 MB). The foreknown size is used for tuning applications at compile-time. Most empirical tuning techniques rely on knowledge about the cache hierarchy and the size of the L1. The problem is tackled by multiplying, employing the iterative space splitting method. We equally divide the problem into two GPUs and tile the data for each. Each data tile is processed by a CUDA stream.

Let a tile shape be a Euclidean space given in height and width as $h \times w$ and cache capacity C, where $1 \leq h$ and $w \leq n$, for the number of rows n. Dense matrix kernels favor large blocks. The best tile size depends on the kind of program. LU decomposition favors $h/w = b$ where b is the size of the cache line, whereas matrix multiplication favors squares ($h/w = 1$). Hsu [43] has computed spatial costs for several famous tiling algorithms dispatched to matrix multiplication kernels. In a majority of these algorithms, the cost is not a function of the cache size. When the cache size is a strong factor usually the cost is $C/(h * w)$ and when it is a weak factor the cost is dominated by a $1/h + 1/w$ term. In any shaping algorithm and depending on the kernel itself, a battle exists between cache efficiency $(h * w)/C$ and tile shape (h/w) and is related to the lopsidedness of the shape. Due to the reuse of matrix values,

cross- and self-conflict arise in large tiles. For these reasons, we prefer a fixed, contiguous, rectangular shape tile at a cost $(h + w)/hw$.

The shared memory and L1 partition on the Fermi architecture is configurable as 48 kB/16 kB or 16 kB/48 kB. Large L1 partition is suitable for irregular scientific codes requiring non-linear or random access to memory (sparse kernels in finite element based computations, ray tracing, and rasterization). In such applications, high bandwidth to framebuffer is serviced through L2, and data locality need is generally infrequent. Smaller L1 partition is feasible for regular loop structures. Our tile size strictly depends on capacity, line size, and data size, i.e., $\mathcal{T}_{i,j} = \{w \times h \ldots\} \subseteq (C, b, n)$. Following the strategy proposed by Panda [44], for a single array (no cross-reference), the tile occupies all the cache ($T = C = h * w$) assuming no residual occupancy in kernel tasks. For double precision general matrix multiplication (DGEMM) kernels where cross conflict is ineluctable, we factor associativity (a) to yield a tile size equal to the effective cache size ($T = C_{eff} = (\frac{a-1}{a})/C$) [45]. In compressed sensing applications of radar [46] and radio astronomy [47], it is often that operations of a sparse matrix $\mathbf{Y}$ simultaneously occur to several dense vectors x, y. In the kernel $y = y + \mathbf{Y}\mathbf{Y}^{\mathbf{T}} \cdot x$, a full load of elements of $\mathbf{A}$ may require several reads. The computations $\mathbf{Y}^{\mathbf{T}} \cdot x$ and $y + \mathbf{Y}(\mathbf{Y}^{\mathbf{T}} \cdot x)$ could reuse the tile for $\mathbf{Y}$.

Our technique is more profitable when the dataset is large and the copying cost is low which is likely to accelerate the DSP kernel, whereas for the small dataset the copying is likely to be costlier than algorithmic latency [48]. The best performance in terms of hitting the least number of cache lines is achieved when the cache access coalesces. This happens when 1) a warp access address does not span over multiple cache lines and 2) the threads are organized in a warp such that they execute similar tasks that do not branch out. A warp is 32 threads that begin at the same program address in single instruction multiple thread organization, but may relinquish to independent branches. If a conditional branch is encountered then the branch paths will be executed serially and upon completion of all branch traversals, the execution resumes to the main path. In other words, address jumps compromise cache hits, instruction as well as thread-level parallelisms.

5.3 Computation Scheduling

Exploiting the regularity property of data, we tune the access patterns (Fig. 3). This process is embedded in the computation and illustrated in Algorithm 1.

Algorithm 1 Iterative space splitting and loop tiling

```
Require: buf, matrix size, number of rows = bsize
Require: CEs = number of CUDA capable GPUs
Require: output submatrix
 1: function SPLIT_AND_ACCESS(buf, bsize, CEs)
 2:     NumDevices ← CEs                                  ▷ NumDevices: schedulable devices
 3:     y ← bsize/NumDevices                              ▷ y: size of data chunks
 4:     s ← calcMaxStreams()                              ▷ s: number of streams
 5:     δ ← y/s                                           ▷ δ: size of stream chunks
 6:     NumThreads ← maxGPUThreads                        ▷ NumThreads: CUDA block dim.
 7:     for DevID ← 1 to NumDevices do                    ▷ workload distribution
 8:         setCudaDevice(DevID)
 9:         from ← DevID ∗ y
10:         to ← DevID ∗ y + y
11:         sbuf ← getChunk(sbuf, from, to)               ▷ data tile
12:         for m ← start to end do
13:             f ← m ∗ δ
14:             t ← m ∗ δ + δ
15:             hkin ← getChunk(sbuf, h, w)               ▷ loop tile (h × w)
16:             hkout ← malloc(...)                       ▷ stream output matrix
17:             S ← cudaStream[s]                         ▷ S: array of streams
18:             NumBlocks ← ceil(y/NumThreads)            ▷ G: blocks of threads
19:             for i ← 1 to s do
20:                 cudaMallocHost(...)
21:                 cudaMemcpy(kin_i, hkin_i, ...)        ▷ Host2Device
22:                 k_i <<< NumBlocks, NumThreads, 0, S_i >>> (kin_i, kout_i, ...)   ▷ kernel launches
23:                 cudaMemcpy(kout_i, hkout_i...)        ▷ Device2Host
24:             submatrix[f :: t] ← hkout                 ▷ get output chunk
25:     return submatrix
```

The device IDs as seen from the host are set. The workload is assigned equally to the number of available devices. The data is chunked equally for each device and the number of streams is determined. A stream is a batch of commands that are issued by the host MCPU threads to the GPU devices. The operations are stuffed in different streams. The commands within a single stream execute in the order of the queue, but the streams are asynchronous to each other or may execute concurrently. Each stream is associated with a particular device. The commands are sandwiched in the queue bounded by the event variables `start` and `stop`. The chunked data is equally distributed to each stream of a device. Each stream copies its allocation of the input array of an array of pointers in the device memory. In DSP and linear algebra kernels, usually, there are several references within a tile during access [43]. The stream then processes arrays by calling kernels. The kernels are based on the highly optimized and reusable basic linear algebra subprograms (BLAS) library routines such as triangular, rectangular, symmetric, Hermitian matrix-matrix multiply, triangular solve or rank updates—all of which are cases of GEMM routine and use floating pointing division [45]. Once the execution is complete, the result is copied back to the original allocation (lines 22-23). To save memory, the default resolution is single precision.

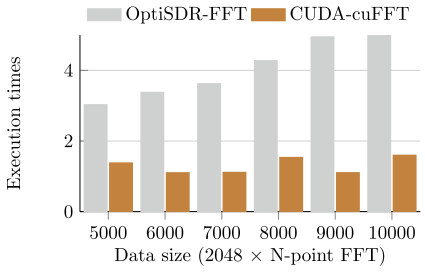
6 Experiment Based Evaluation

A set of representative experiments were carried out to evaluate the effects of frontend and backend compilation. The experiments were performed on Intel i7 3.2 GHz eight-core MCPU with 24GB DRAM and two GeForce GTX 480 GPUs (compute capability 2.0).

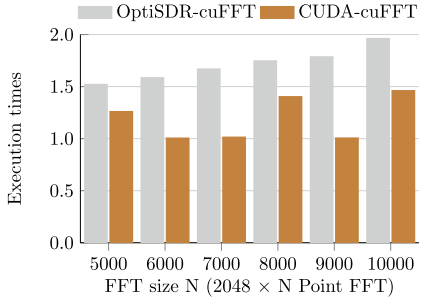
6.1 FFT Centered Algorithms

First, the OptiSDR was compared with handcrafted CUDA. In Fig. 5a, the unoptimized OptiSDR FFT is examined for single precision floating point operations per second (FLOPS) for different lengths. CUDA turns out 3-4× better. Figure 5b shows that the OptiSDR performance is improved when the FFT IR node loads the cuFFT (based on FFTW), although handcrafted CUDA is still better. However, when FFT is implemented as part of higher algorithms (Fig. 5c), OptiSDR outperforms on larger data sizes and its performance is more evident in Fig. 5d which is of time-domain convolution parallelized as `MapReduce`. Notice also that within OptiSDR the advantage of combining patterns, i.e., parallel stream processing achieved better performance with stencil access over the while loop. The floating point operations are given by $2TN$, where N is the number of data samples, T is the filter tap order. T corresponds to the number of reduction parts of `MapReduce` performed.

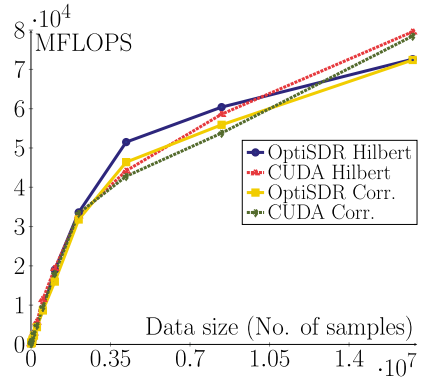
When the number of available threads is smaller than the sample sizes, the performance is degraded significantly, as can be seen for the OpenMP implementation in Fig. 6a and Fig. 6b. In these OptiSDR and CUDA executions, when mathematical



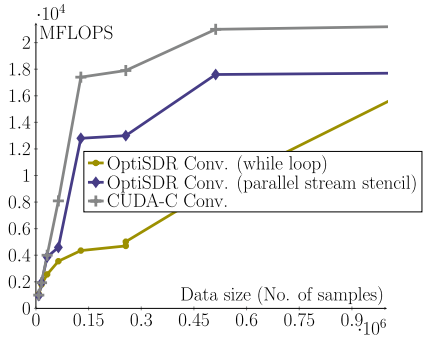
(a) Standard FFT performance in OptiSDR versus the handcrafted cuFFT.



(b) Performance using the external cuFFT library in OptiSDR versus the handcrafted cuFFT.



(c) FLOP count against CUDA in Hilbert transform and correlation computations.



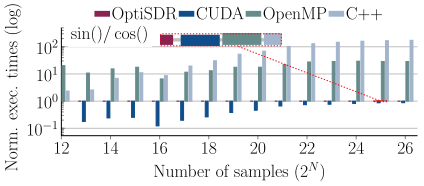
(d) FLOP count with parallelization applied through patterns and loops.

Fig. 5 Fourier and Hilbert transforms performance using OptiSDR and handcrafted CUDA

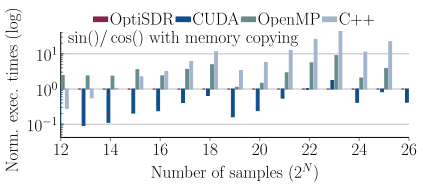
operations are instantiated with other memory and compute-intensive FFT-based applications, the maximum number of samples which can be processed in parallel was found to be approximately 2^{26} double floating point complex values. This maximum number of the sample may change to larger or smaller data sizes depending on the GPU used.

In Fig. 6a and Fig. 6c, the serial C++ implementation gets worse as the number of samples increase. Further, the OptiSDR achieves a maximum speedup of 180× and 87× for the parallel sinusoidal generator and power spectral density (PSD) computations, respectively. As compared to the OpenMP implementation, the OptiSDR speeds up to 20× for the parallel sinusoidal generator, and up to 85× for the PSD computations.

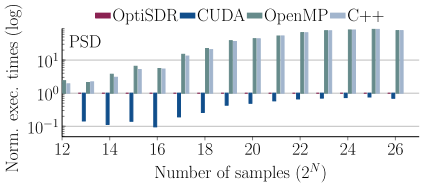
The executions time and memory copying latencies introduced in GPU based implementations are presented in Fig. 6b and d. Significant degradation is taken on when memory copying is included in both CUDA and OptiSDR implementations. Around data size of 2^{24}, the speedup is reduced significantly, this is because the memory copying latencies for data sizes greater than that are irregular. In comparison with OpenMP library and serial C++ implementations, speedup of 12× and 49× are achieved respectively for sinusoidal data generators. Speedup of 39× for the PSD computation is achieved when compared to OpenMP and C++ implementations. Both OptiSDR and CUDA perform significantly better over C++ and OpenMP when memory copying is considered. Furthermore, the OptiSDR turns out close to handcrafted CUDA performance.



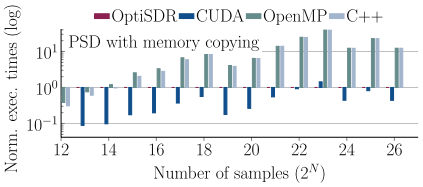
(a) Serial (C++/OpenMP) vs. parallel (CUDA/OptiSDR).



(b) D2H and H2D transfers.



(c) Serial (C++/OpenMP) vs. parallel (CUDA/OptiSDR).



(d) D2H and H2D transfers.

Fig. 6 Runtime performance for the same dataset against different serial and parallel programming models

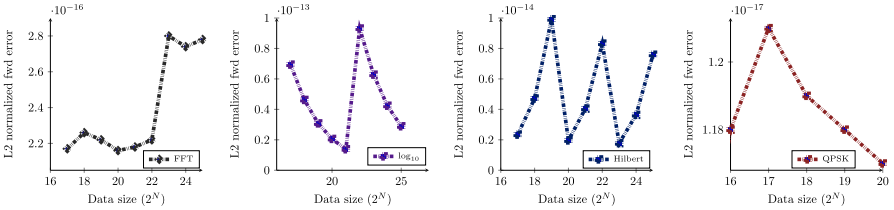


Fig. 7 Accuracy benchmarking of OptiSDR using different algorithms

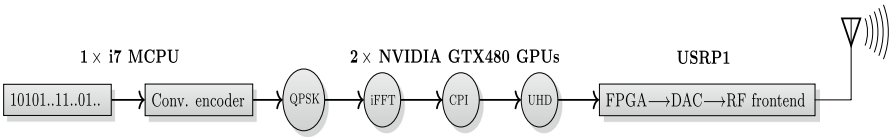


Fig. 8 OptiSDR platform may be used for realtime processing of radio data on a heterogeneous MCPU-GPU machine

6.2 L2 Norm

For typical computations ($\mathbf{AX} = \mathbf{B}, \mathbf{A^T}$), the forward error is the computation noise given as the least square error in component vectors (L^2 norm) $||x|| = \sqrt{\sum_{n=1}^{N} |x_k|^2}$ and the overall error between the true and the approximate matrix is $E = \mathbf{A_{approx}} - \mathbf{A}$. We compute these errors in various algorithms as shown in Fig. 7. An average of ten samples is taken for each computation. The resolution was restricted to a double precision fixed point Q52.12 format.

6.3 Realtime OFDM

Realtime cognitive spectrum sensing, radar doppler filter, constant false alarm rate, and coherent dedispersion in radio astronomy are applications that suit the parallel programming model. Consider Fig. 8 of an FPGA based OFDM transmitter processing chain using UHD USB 2.0 I/O modules and frontend [49]. The system has been set up for processing a batch of pulses at a coherent processing interval (CPI).

The sequential implementation of encoding and modulation blocks in PSPP streamprocess mimic the realtime processing. The OptiSDR compiler extracts parallelism in each DSP algorithm queued in the PSPP task graph and schedules them accordingly for execution in the CUDA device while removing memory copying back and forth from the CUDA device to the high-level environment. An alternate implementation used while(true) which mimics realtime processing. Within the loop, the DSP functions were computed sequentially, but each parallelized in the GPU. Each DSP operation was mapped to its equivalent structured parallel pattern, thus embracing some specific internal parallelism. The PSPP improved the performance over sequential processing in a loop. Also, the entire implementation was within the CUDA C environment. The results are demonstrated in Table 2. The

Table 2 Realtime OFDM execution times

Size	Parallel stream processing pattern (ms)				While loop (ms)			
(k)	Encoder	QPSK	iFFT	CPI	Encoder	QPSK	iFFT	CPI
200	0.745	0.760	2.886	0.970	1.398	1.405	5.106	2.035
500	1.094	1.020	3.013	1.540	2.141	2.200	7.792	2.080

Table 3 Latency (ms) in realtime PSD computations

Size (M)	UHD and H2D Mem. copy	OptiSDR	CUDA-Qt	Octave
1	132.745	31.637	20.870	122.958
2	259.940	32.004	21.913	203.958
4	542.876	33.307	23.105	390.358
6	769.469	35.366	26.836	428.593
8	1027.310	37.585	28.778	560.149

PSPP model outperforms the while-loop version by the speedup of 2×. As presented in [50], the FFT, QPSK/QAM, and CPI execution times in GPU were all under 1 ms for a sample size of 1024, where the OptiSDR PSPP versions were just under 2 ms for a sample size of 200 ksamples.

6.4 Spectrum Sweep

An FPGA-based tunable realtime scanning receiver (similar to the signal path in Fig. 8) was set up using the universal software radio peripheral (USRP) and RFX2400 frontend. To set the USRP streaming rate, the interconnection between the USRP platform and the MCPU-GPU system limits the maximum sampling frequency. Thus, the maximum RF sweep step is 8MHz in 2.5 GHz to 2.532 GHz band.

Table 3 lists the recorded latencies resulting from the benchmark of OptiSDR against CUDA-Qt and Octave PSD applications while varying the number of samples. This result is linear and exposes no difficulty in parallelization due to the lack of data dependencies in the samples.

6.5 Number of Operations and Wattage Profile

Within the while loop, the OFDM waveform generation required memory copying back and forth between the JVM environment and the CUDA device. To limit the overhead, we lowered the abstraction of all modules to CUDA-C level to ensure that there is no device to host (D2H) memory copying until the kernel executions of the DSP tasks queued in the PSPP task graph are completed. This resulted in more computations at less power, as illustrated in Fig. 9.

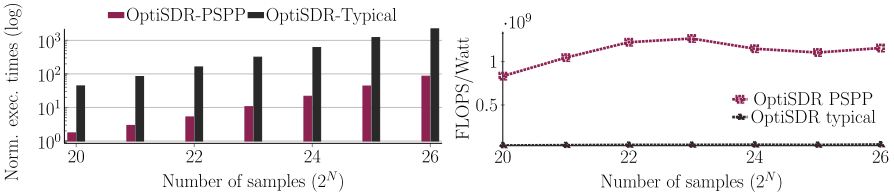


Fig. 9 Speed and wattage comparison with two OptiSDR implementations: PSPP versus Scala's top-down approach involving memory copying within JVM

7 Discussion

Further speedup is possible by dynamically determining the tile size for each application. Due to limited hardware resources, many implementations favor storing the twiddle factors $\mathbf{W_n}^k = e^{-2j\pi/N}$ in lookup tables and calling the library functions over and over for trigonometric computations or computing $\mathbf{W_n}^{\mathbf{k}} = \mathbf{W_n^{k-1}}\,\mathbf{W_n}$ and storing $\mathbf{W_n}$. This technique saves up on $\mathbf{W_n X_n}$ computation, and it is feasible for applications in which the size of the FFT (number of samples) is known beforehand and allocated for in as many memory locations. All these avenues should be taken up for a particular class of problems in which speed is a significant factor. The programmer should watch out for non-linear accesses through excessive branching, dereferencing, and function calls that result in a violation. For contiguous traversals, matrices should be stored in a row-major array layout. However, there will be some latency if the code uses cuBLAS which uses a column-major layout. For backend optimization, it is worth remarking that heroic tuning of the compiler leads to undecidable and unreachable problems. Suffice it to mention that most of the reported efforts are non-trivial, as it is generally required to have a significant insight into the compiler algorithms to reason out where the compiler spends most of the time in which fragment of the program. Most solutions are multi-objective, and therefore the parametric tradespace is typically huge. Some techniques favor hacks, heuristics, and combinations of thereof with conventional optimization. All in all, none of these techniques is conclusive by any measure in the theory of optimization. Similarly, the best effort of the handcrafted code for efficient utilization of the hardware resources results in a modest gain. Moreover, according to Amdahl's law, adding more CEs does not necessarily help.

The standard recursive FFTW code [51], when unfolded for concrete code generator through pure functional staged calls (e.g., staging the size of the input [52] or the real and imaginary components), has shown comparable performance. Lengauer et al. [53] have staged FFT. The operations carried out were shuffle, parallel butterflies of addition and subtraction, element-wise multiplication with twiddle constants, and bit reversal permutations. The inspiration of [52] and implementations of hardware generator by Lengauer through Scala software-based staging have paved the way to target FPGA hardware as well as using the signal type extensions for realtime streaming acquisition and processing in the HCA platform. This development is a good future path for extending OptiSDR for the FPGA target, since similar semantics and staging are already in place.

8 Conclusion

Efficient design for signal processing applications has been presented with a DSL at the frontend and compiler optimization at the backend. Several potential opportunities for boosting the application runtime at various stages of development have been regularly indicated. DSLs are lightweight and are implemented as a library. We outlined essential features in the radio signal processing domain and defined guidelines for domain-specific optimizations and constructs. For speed gain and usability, we have favored top-level DSL constructs corresponding to DSP and parallel patterns in the OptiSDR DSL. We have applied automatic compilation. While trading usability, we have enforced optimizations through the intermediate restriction rules and symbolic-assisted computations. This approach bridges the gap between the high-level abstraction and concrete implementation at runtime. On the whole, the OptiSDR performance is better than standard languages when the domain-specific DSP and parallel patterns are correctly applied, memory access tuned, and the compiler is set up to match the optimization to the patterns. We do not consider it a significant concern that handcrafted CUDA has slightly better performance than OptiSDR; the important aspect is OptiSDR ease of use and embedding domain knowledge, parallel patterns, and optimization into the application. The reuse of DSP and parallel patterns for several signal processing problems have shown promising performance and strengthened the notion that in resource unaware and generative programming, the serviceability to the user is imperative. Our objective was not to be exhaustive, but we presented general principles

Author contributions YZ 50%, SW 50%.

Funding This document is the result of the research funded by the South African Radio Astronomy Observatory (SARAO), which is a facility of the National Research Foundation, an agency of the Department of Science and Technology.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Chugh, N., Vasista, V., Purini, S., and Bondhugula, U.: A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs, *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, Association for Computing Machinery, New York, NY, USA, (2016) p. 327-338
2. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages **37**, 4 (2005)
3. Rompf, T., Sujeeth, A.K., Lee, H., Brown, K.J., Chafi, H., Odersky, M., Olukotun, K.: Building-Blocks for Performance Oriented DSLs, Electronic Proceedings in Theoretical Computer. Science **66**, 93–117 (2011)
4. Serre, F., Püschel, M.: DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks. ACM Trans. Reconfigurable Technol. Syst. **13**, 1 (2019)
5. Cong, J., Sarkar, V., Reinman, G., Bui, A.: Customizable Domain-Specific Computing. IEEE Design Test of Computers **28**(2), 6–15 (2011)
6. Mohapi, L.J., Winberg, S., and Inggs, M.: A domain-specific language to facilitate software defined radio parallel executable patterns deployment on heterogeneous architectures, *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8 (2014)
7. Mohapi, L.J., Winberg, S., Inggs, M.: Using a domain specific language for SDR to facilitate radar signal processing in heterogeneous computing architectures. IEEE Radar Conference **2015**, 306–311 (2015)
8. Winberg, S., Mohapi, L., Inggs, M.: OptiSDR - A domain specific language to improve developer productivity for software defined radio. IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech) **2016**, 20–26 (2016)
9. Sujeeth A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages, Vol. 13, No. 4s, (2014)
10. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: A domain specific language for digital signal processing algorithms, *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pp. 169–178 (2010)
11. Khedker, U., Sanyal, A., Karkare, B.: *Data Flow Analysis: Theory and Practice*, 1st ed., CRC Press, Inc., USA, (2009)
12. Falk, H., Marwedel, P.: Source Code Optimization Techniques for Data Flow Dominated Embedded Software. Springer, Boston (2004)
13. Aronsson, M., Axelsson, E., Sheeran, M.: Stream Processing for Embedded Domain Specific Languages. Association for Computing Machinery, New York, NY, USA (2014)
14. Claessen, K., Sheeran, M., Svensson, B.J.: Expressive Array Constructs in an Embedded GPU Kernel Programming Language. Association for Computing Machinery, New York, NY, USA (2012)
15. Gamatié, A.: Designing Embedded Systems with the SIGNAL Programming Language. Springer-Verlag, New York (2010)
16. Parfieniuk, K.M., Marek, Kowalewski, K.: *Towards Modern Collaborative Knowledge Sharing Systems*, Springer, Berlin, Chaps. A Compiler for a Domain-Specific Language for Rapid Implementation of DSP Transforms and Filter Banks (2012)
17. Puschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proc. IEEE **93**(2), 232–275 (2005)
18. Li, H., Ren, H., Wu, J.: Domain Specific Compiler for Coordinated Signal Processing in 5G Testbed. Association for Computing Machinery, New York, NY, USA (2017)
19. Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., Chame, J.: A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code, Vol. 9, No. 4, (2013)
20. Alhuwaimel, S., Coetzee, S., Cheng, P., Du Plessis, D., Fioranelli, F., Griffiths, H., Inggs, M.R., Jordan, D., Miceli, W., O'Hagan, D., Palama, R., Paine, S., Ritchie, M., Sandenbergh, J.S., Stevens, A.: First measurements with NeXtRAD, a polarimetric X/L Band radar network. IEEE Radar Conference (RadarConf) **2017**, 1663–1668 (2017)
21. Banerjee, U.: Loop Transformations for Restructuring Compilers. Springer, Boston (1993)
22. Xue, J.: Loop Tiling for Parallelism. Springer, Boston (2000)
23. Kennedy, K., Allen, J.R.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

24. Morgan, R.: Building an Optimizing Compiler. Digital Press, USA (1998)
25. McCool, M., Reinders, J., Robison, A.: *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, (2012)
26. Hendrickson, B., Leland, R.: An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. SIAM J. Sci. Comput. **16**(2), 452–469 (1995)
27. Gebali, F.: *Algorithms and Parallel Computing*, 1st ed., Wiley Publishing, (2011)
28. Brandwood, D.: Fourier Transforms in Radar and Signal Processing. Artech House, Boston, MA, USA (2003)
29. Fiorin, L., Jongerius, R., Vermij, E., van Lunteren, J., Hagleitner, C.: Near-Memory Acceleration for Radio Astronomy. IEEE Trans. Parallel Distrib. Syst. **29**(1), 115–128 (2018)
30. Tegmark, M., Zaldarriaga, M.: Fast Fourier transform telescope. Phys. Rev. D **79**, 083530 (2009)
31. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. Society for Industrial and Applied Mathematics, USA (1992)
32. Burrus, C.: Index mappings for multidimensional formulation of the DFT and convolution. IEEE Trans. Acoust. Speech Signal Process. **25**(3), 239–242 (1977)
33. Sorensen, H., Jones, D., Heideman, M., Burrus, C.: Real-valued fast Fourier transform algorithms. IEEE Trans. Acoust. Speech Signal Process. **35**(6), 849–863 (1987)
34. Kolba, D., Parks, T.: A prime factor FFT algorithm using high-speed convolution. IEEE Trans. Acoust. Speech Signal Process. **25**(4), 281–294 (1977)
35. Bluestein, L.: A linear filtering approach to the computation of discrete Fourier transform. IEEE Trans. Audio Electroacoust. **18**(4), 451–455 (1970)
36. Galati, G.: *Advanced Radar Techniques and Systems*, Institution of Engineering and Technology, (1993)
37. Levanon, N., Eli Mozeson, E.: *Radar Signals*, Wiley-IEEE Press, USA, (2004)
38. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and Reuse with Compiled Domain-Specific Languages, ECOOP,: Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1–5, 2013. Proceedings, Lecture Notes in Computer Science, Vol. 7920, edited by G. Castagna, Springer **2013**, 52–78 (2013)
39. Rompf, T., Odersky, M.: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, Vol. 46, No. 2, (2010)
40. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Institute of Physics Publishing (2005)
41. Chen, C.-M., King, C.-T.: Walk-time address adjustment for improving the accuracy of dynamic branch prediction. IEEE Trans. Comput. **48**(5), 457–469 (1999)
42. Ferreira, R.J.E.: Alfonso. Lecture Notes in Computer Science, Springer, Boston (1995)
43. Hsu, C.-h., Kremer, U.: A Quantitative Analysis of Tile Size Selection Algorithms, Vol. 27, No. 3, (2004)
44. Panda, P.R., Nakamura, H., Dutt, N.D., Nicolau, A.: Augmenting loop tiling with data alignment for improved cache performance. IEEE Trans. Comput. **48**(2), 142–149 (1999)
45. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R.C., Yelick, K.: Self-Adapting Linear Algebra Algorithms and Software. Proc. IEEE **93**(2), 293–312 (2005)
46. Potter, L.C., Ertin, E., Parker, J.T., Cetin, M.: Sparsity and Compressed Sensing in Radar Imaging. Proc. IEEE **98**(6), 1006–1020 (2010)
47. Starck, J., Bobin, J.: Astronomical Data Analysis and Sparsity: From Wavelets to Compressed Sensing. Proc. IEEE **98**(6), 1021–1030 (2010)
48. Whaley, R., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project, *Parallel Computing*, Vol. 27, No. 1, pp. 3–35. (2001) New Trends in High Performance Computing
49. Wamicha, J.N.: Software defined radio ofdm implementation, mixed signal circuit design and ebg antenna design, Master's thesis, Dept. of Electrical Engineering, University of Cape Town, Rondebosch, (2011). http://www.radarmasters.uct.ac.za/sites/default/files/image_tool/images/39/publication_PDF/MSc/2/jwamicha_thesis.pdf
50. Xiao Ma, Zhao, H., Geng Li, Yuping Zhao: Implementation of a high-throughput OFDM system using Graphics Processing Units, *2013 15th IEEE International Conference on Communication Technology*, pp. 639–644 (2013)
51. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)
52. Kiselyov, O., Swadi, K.N., Taha, W.: A Methodology for Generating Verified Combinatorial Circuits. Association for Computing Machinery, New York, NY, USA (2004)

53. Lengauer, C., Batory, D.S., Consel, C., Odersky, M.: (eds.) *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, Lecture Notes in Computer Science, Vol. 3016, Springer, (2004)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



https://dspace.org/

Automatic heterogeneous runtime using signal processing domain-specific and parallel patterns

Zaidi, Yaseen; Winberg, Simon

2025-04-01
Springer
https://doi.org/10.1007/s10766-025-00790-7
Zaidi Y, Winberg S. (2025) Automatic heterogeneous runtime using signal processing domain-specific and parallel patterns. International Journal of Parallel Programming, Volume 53, Issue 2, April 2025, Article number 8

Downloaded from a DSpace repository. https://dspace.org/