Electronic Research Archive

https://www.aimspress.com/journal/era

ERA, 33(5): 2698–2718.
DOI: 10.3934/era.2025119
Received: 16 January 2025
Revised: 14 April 2025
Accepted: 22 April 2025
Published: 30 April 2025

Research article

DSP-OPU: An FPGA-based overlay processor for digital signal processing

Yueyin Bai*, Song Zhang, Zhiyuan Ma, Enhao Tang and Jun Yu

School of Microelectronics, Fudan University, Shanghai, China

* **Correspondence:** Email: 20112020048@fudan.edu.cn.

Abstract: Digital signal processing (DSP) is an important technology in various research fields. However, mainstream implementations of DSP algorithms either lack flexibility and scalability or have performance saturation and bottleneck, which still have the potential to be optimized. In this work, we proposed DSP-OPU, an FPGA-based overlay processor for digital signal processing. Specifically, we designed an overlay architecture suitable for various DSP algorithms. A unique data path was proposed with multiple computation engines and a reconfigurable pipeline. Additionally, we realized software-hardware co-design for DSP-OPU. On the software side, we achieved excellent scalability with a customized instruction set and a user-friendly compiler. On the hardware side, the reconfigurable data path breaks the dependency between instructions and data, enhancing scheduling ability and transmission efficiency. In addition, we also decreased the performance saturation with increasing engine numbers. Compared to C6678 SoC, our DSP-OPU achieves up to 29 × speedup and 70 × higher energy efficiency for different DSP algorithms. Compared to FPGA-based implementations for a single DSP algorithm, we achieved up to 4.5 × speedup and 4.4 × better energy efficiency.

Keywords: digital signal processing (DSP); software-hardware co-design; reconfigurable architecture; overlay processor; field-programmable gate array (FPGA); system architecture; wireless communication

1. Introduction

With the rapid development of information science and wireless communication, there has been an explosive growth in digital signal processing (DSP) [1]. The DSP has become an indispensable technology in various fields, e.g., control systems [2, 3], communication systems [4, 5], and image processing [6]. Moreover, to meet the demands for low latency and high throughput in fields, e.g., the Internet of Things (IoT) [7], artificial intelligence [8], and 6G [9], the demands for performance enhancement in digital signal processors are becoming increasingly important.

Existing hardware implementations of digital signal processing designs are mainly divided into three types, i.e., the application-specific integrated circuit (ASIC), system on chip (SoC), and field-

programmable gate array (FPGA) [10]. Specifically, ASIC-based designs minimize the area and power consumption with high performance. However, their fixed hardware circuits are not programmable with high development costs [11, 12]. Moreover, SoC-based designs such as TMS320C6678 (C6678) can be programmable and customized to a certain extent, but their performance is likely to be saturated while leveraging multiple cores [13]. They also suffer from limited performance and massive power consumption due to low data transmitting speed and insufficient hardware optimization. In contrast, FPGA has become increasingly popular in DSP design due to its programmability, parallel computing units, and ability to significantly reduce development costs while maintaining high performance and flexibility [14].

There are already several works on implementing DSP algorithms with FPGA. For instance, Saeed et al. [15] introduced an FPGA-based compatible FFT/IFFT processor with lower area and latency. Paul et al. [16] migrated different forms of FIR and IIR filters to an FPGA. Vijay et al. [17] proposed a parallel pipeline based on FIR filters to efficiently implement IIR filters. Kavitha et al. [18] designed an LMS-based adaptive filter algorithm that reduces area and power consumption by eliminating multipliers.

Despite the good performances, the above FPGA-based works are specifically designed for accelerating one or two DSP algorithms. Constrained by the limited hardware resources, such designs cannot deploy all the required algorithms on target FPGA [19]. Even if all DSP algorithms are deployed, the execution efficiency of each algorithm is low due to the small amount of allocated resources. As a result, such works cannot support complex scenarios. For example, in multi-functional scenarios, where different DSP algorithms are required and updated, such designs need to re-design and re-generate related RTL circuits for diverse DSP algorithms, which is difficult to adapt to various applications.

It is a reliable solution to design an FPGA-based processor compatible with multiple DSP algorithms. This processor has sufficient parallel computing resources and is reconfigured to support new algorithms by modifying a small portion of the design. However, some challenges are still remaining to implement a general processor, such as data and control scheduling while users' requirements are updated, as well as reusable computation units compatible with different DSP algorithms.

In this paper, we propose an FPGA-based overlay processor for DSP algorithms, named DSP-OPU. Specifically, we present a customized instruction set and an overlay hardware architecture for the DSP-OPU. We further incorporate multiple reconfigurable computation engines (RCE) in DSP-OPU to mitigate performance saturation. These RCEs can be reused for different DSP algorithms, thereby reducing resource utilization and enhancing system integration. Additionally, we design an efficient compiler to schedule and convert algorithms into instructions that can be run directly on the FPGA.

In summary, the contributions of our work are as follows:

- **An overlay architecture for DSP.** We design a reconfigurable and domain-specific overlay processor for DSP algorithms. Specifically, DSP-OPU is capable of accelerating various DSP algorithms with a flexible and scalable hardware architecture. It is also optimized to be highly parallel, ensuring high computation efficiency.

- **Multiple reconfigurable computation engines.** In DSP-OPU, we propose a series of novel computation engines. The engines are highly flexible, enabling the reconfigurable interconnections to accelerate the required algorithm and thereby significantly reducing power consumption. Different from SoC-based C6678 chip, our DSP-OPU alleviates the performance saturation with the increasing of engine numbers due to efficient bandwidth management.

- **User-friendly SW-HW co-design.** On the software side, we customize an instruction set architecture (ISA) for DSP-OPU, which can be easily enhanced to support new emerging algorithms with strong scalability. A simple and efficient compiler is also proposed to generate executable instruction streams from users' detailed requirements. The compiler provides hardware-friendly optimizations such as model parsing, instruction generation, along with instruction and data optimization. On the hardware side, we leverage a reconfigurable data path related to our customized instructions. This data path breaks the dependency between instructions and data, therefore significantly improving the ability to schedule on-chip data and enhancing transmission efficiency.

- **High performance.** We implement our DSP-OPU on Kintex-7 XC7K325T (325T) FPGA and Xilinx Alveo U200 (U200) FPGA. With comprehensive evaluations, DSP-OPU demonstrates competitive performance, achieving up to 29 × lower latency and 70 × better energy efficiency compared to C6678 SoC. In addition, compared to other DSP implementations for a single algorithm on FPGA, DSP-OPU achieves up to 4.5 × speedup and 4.4 × better energy efficiency.

2. Background and motivation

DSP is an essential technology in multiple fields like wireless communication systems and IoT. There are various commonly used and efficient DSP algorithms for different functions, e.g., FFT, FIR, IIR, and LMS. Specifically, FFT facilitates time-frequency transformations by decomposing the signal spectrum into distinct frequency components. FIR and IIR are responsible for signal filtering and frequency response control. FIR processes the signal through convolution with a set of coefficients, and IIR introduces feedback into the signal output. LMS is an adaptive filtering algorithm that adjusts filter coefficients based on the input signal. The computational details of different DSP algorithms are outlined in Table 1.

Table 1. Descriptions of common DSP algorithms.

Algorithm	Expression of the algorithm
FFT	$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}$
FIR	$y(n) = \sum_{k=0}^{N-1} h(k)\, x(n-k)$
IIR	$y(n) = \sum_{k=0}^{N} b_k x(n-k) - \sum_{k=1}^{N} a_k y(n-k)$
	$y(n) = w^T(n)\, x(n)$
LMS	$e(n) = d(n) - y(n)$
	$w(n+1) = w(n) + 2\mu e(n)\, x(n)$

In recent years, there has been a continuous rise in performance demands for DSP technology, especially in terms of latency requirements for a single algorithm [20, 21]. However, existing DSP chips have not updated their production lines for several years with limited performance. In contrast, there is a growing number of high-performance DSP algorithm accelerators proposed on FPGA, which can accelerate DSP algorithms 2-300 × faster than existing DSP chips, as indicated in Table 2. Therefore, considering FPGA for implementing DSP algorithms is a worthy option [22]. The throughput in Table 2 represents the millions of samples (MS/s) that can be processed per second. The specific throughput is obtained by analyzing the processing time of samples.

Table 2. Practical throughput comparison on different platforms while implementing a single DSP algorithm.

Designs	Platform	Algorithm	Point	Throughput (MS/s)
Garrido et al. [23]	Virtex-7 XC7VS332T	FFT	1024	2720.00
Potsangbam et al. [24]	Virtex-7 XC7VS335T	FIR	\	195.00
		IIR	\	212.00
Ezilarasan et al. [25]	Virtex-5 XC5VLX30	LMS	\	112.00
DSP SoC	TMS320C6678	FFT	1024	97.00
		FIR	\	57.00
		IIR	\	116.00
		LMS	\	0.35

However, the mainstream FPGA-based DSP implementations are primarily tailored to support a single DSP algorithm and cannot simultaneously handle diverse algorithmic workflows. There is a feasible solution to accelerate different algorithms by deploying various DSP IPs on the FPGA. Nevertheless, such designs often require a trade-off between the number of supported algorithms, FPGA hardware resources, and the throughput of IPs. Moreover, deploying multiple IPs on an FPGA allocates limited resources for each IP. However, for most algorithms, fewer allocated resources always lead to lower throughput, which restricts the performance of each IP.

There are several widely used SoC designs in the industry such as C6678, which support the implementations of multiple DSP algorithms. The C6678 is based on SIMD instruction architecture and generates corresponding instructions for different algorithms through a specialized compiler. However, the C6678 still suffers from resource waste, resulting from the inability to leverage multiple computing cores simultaneously for single-task acceleration. Moreover, the practical performances in multi-task workloads are severely limited by transmission speed. Constrained by the available transmission bandwidth, the data transferring time of C6678 often surpasses the time of the computation part. This limitation becomes even greater with an increased core number and leads to performance saturation as shown in Figure 1.

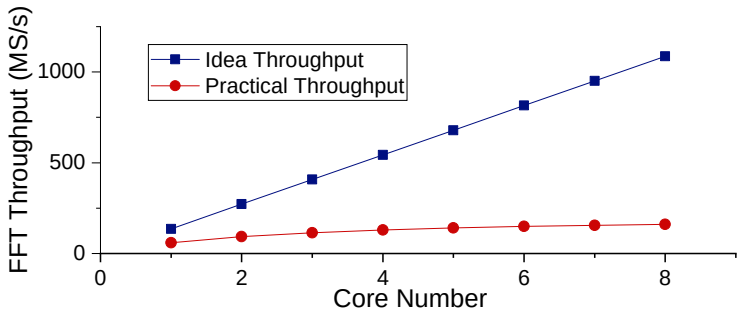

Figure 1. Performance saturation caused by transmission bandwidth limitation.

In order to address the above challenges, we devise an FPGA-based overlay processor called DSP-OPU to support the acceleration of various DSP algorithms with multiple RCEs and less performance

saturation. We introduce a multi-level datapath and integrate it with RCEs efficiently, achieving complete separation of instructions and data, which enhances overall throughput and computation efficiency. A flexible ISA is also proposed in our work with a user-friendly compiler. The compiler can transform the requirements of users into detailed instruction streams, which can be decoded and executed on the underlying hardware architecture.

3. Hardware architecture

3.1. DSP-OPU architecture overview

A substantial challenge in the design of the OPU lies in microarchitecture design. The microarchitecture must minimize control overhead while preserving the convenience of runtime adjustment and functionality. To address this challenge, we have developed a unique data path module that facilitates parameterized register-based customization and mode switching for the data path. These parameter registers directly receive parameters provided by instructions, enabling seamless transitions between different data paths. Additionally, this data path module can establish connections between different computation engines, forming a reconfigurable pipeline structure for high-performance DSP algorithm computations. Furthermore, we propose a multi-engine system with shared memory to meet the high-bandwidth demands when accessing data.

We deploy DSP-OPU on FPGA and design a dedicated instruction set on the software side. The microarchitecture is depicted in Figure 2. Our DSP-OPU primarily consists of the following components: data pre-processing, instruction subsystem, L3DP, multiple engines shared memory (MESM), multiple engines shared memory controller (MESMC), processing element of reconfigurable pipeline (PERP), and reconfigurable computation engines (RCE). Upon receiving user-supplied parameters and the data to be processed, the compiler initially converts them into corresponding instructions while simultaneously handling the data. Subsequently, the instructions and data are transmitted for execution on the hardware processor. A detailed description of the instruction set and compiler will be provided in the following chapter.

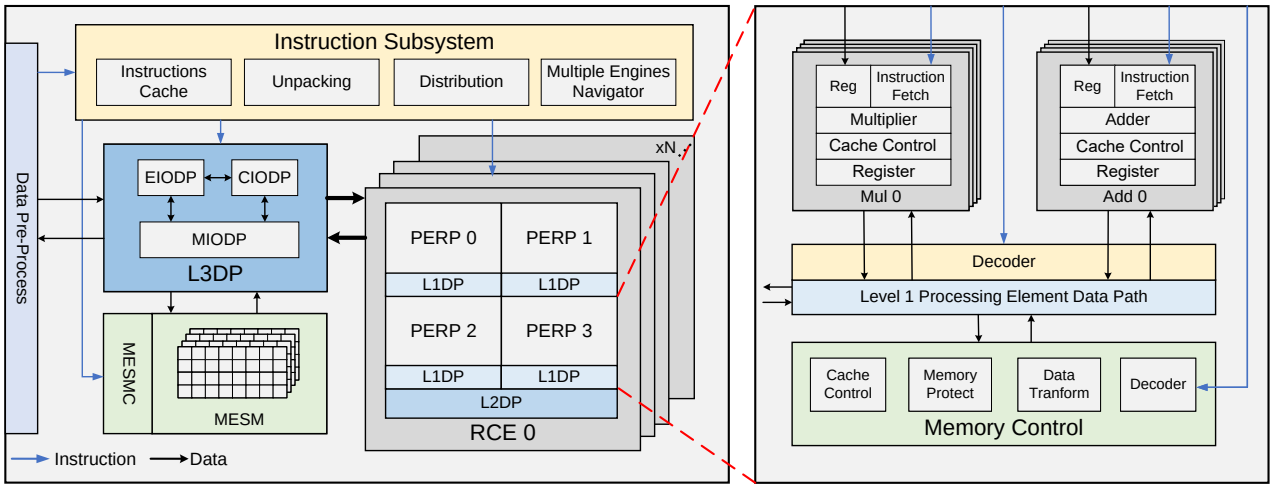

Figure 2. The hardware microarchitecture of DSP-OPU.

The data path module serves as the core of DSP-OPU, and the proposed microarchitecture, instruction set, and compiler are all designed with the data path as the central focus. The data path realizes the majority of data scheduling in the architecture. Once the data path is constructed, input data will be continuously transmitted according to the data path. Therefore, this architecture possesses innate hardware advantages for accelerating algorithms with fixed computation flows. To reduce logical complexity and resource consumption, the data path of our DSP-OPU can be divided into three levels, i.e., L1DP, L2DP, and L3DP. In L1DP, the data paths mainly exist between the adder, multiplier, register, and transformation unit, and these units form a computing module PERP. In L2DP, the data paths mainly exist between different PERPs, and four PERPs form an RCE along with the RCE input and output modules. In L3DP, the data paths mainly exist between architecture input and output, multiple RCEs, and MESM.

3.2. Reconfigurable computation engines (RCEs)

To meet the high parallel computing demands of DSP algorithms, a distinctive computing module has been devised, which houses multiple RCEs. Balancing control complexity and parallelism, four PERPs are employed instead of directly interconnecting all computing modules within the RCE. Within a single RCE, while maintaining a constant number of internal computing modules, an increase in the number of PERPs significantly reduces the number of data paths, concurrently lowering pipeline complexity. To strike a balance between control overhead and reconfigurable pipeline complexity, evaluations of various metrics on the RCE are performed, as shown in Figure 3. Pipeline complexity indicates the number of different pipeline types to which a single computational PERP can be reconfigured, while pipeline length denotes the number of PERPs in a single RCE. Each pipeline type corresponds to a computational process that a PERP can support. By considering control overhead, reconfigurable pipeline complexity, and deployable PERP count, and based on the curve intersections and instructions design in Figure 3, we conclude the utilization of four PERPs per RCE.

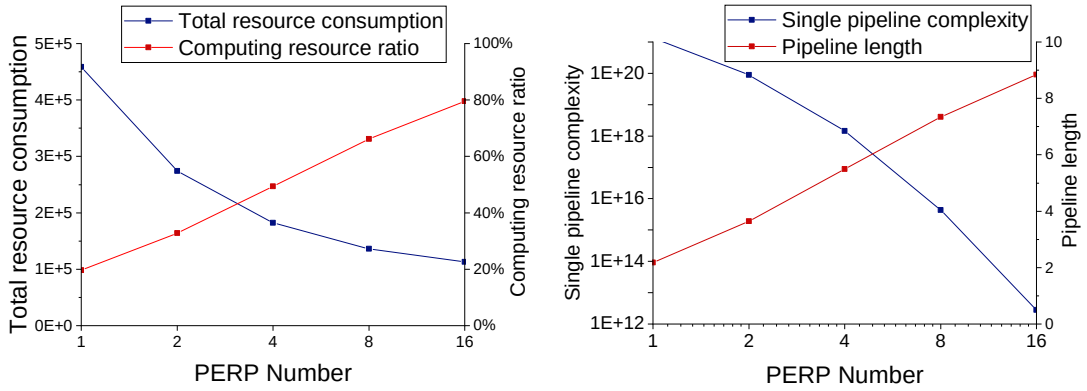

Figure 3. PERP number determination based on four aspects of total resource consumption, computing resource ratio, single pipeline complexity, and pipeline length.

Within the RCE, PERPs are connected via the Level 2 RCE data path (L2DP), as illustrated in Figure 4. All PERPs are connected through the data path. According to the selection of different data paths, PERPs can transmit data to the next PERP, obviating the need for data scheduling using caches. It is

worth noting that the simplified representation of data paths in the figure understates the actual count, given that each PERP encompasses multiple computing modules. The selection of paths is specified by instructions, empowering the RCE for reconfiguration to different pipelines through data paths, thereby substantially elevating hardware optimization for algorithms.

It is noteworthy that DSP-OPU achieves robust scalability and compatibility with new digital signal processing algorithms by increasing the number of RCEs based on FPGAs with more resources. This scalability can be achieved by simply concatenating additional computation modules, enhancing parallelism and pipeline length without significantly increasing control complexity. We retain instructions in the instruction set for switching RCEs, requiring no modification to the instruction set.

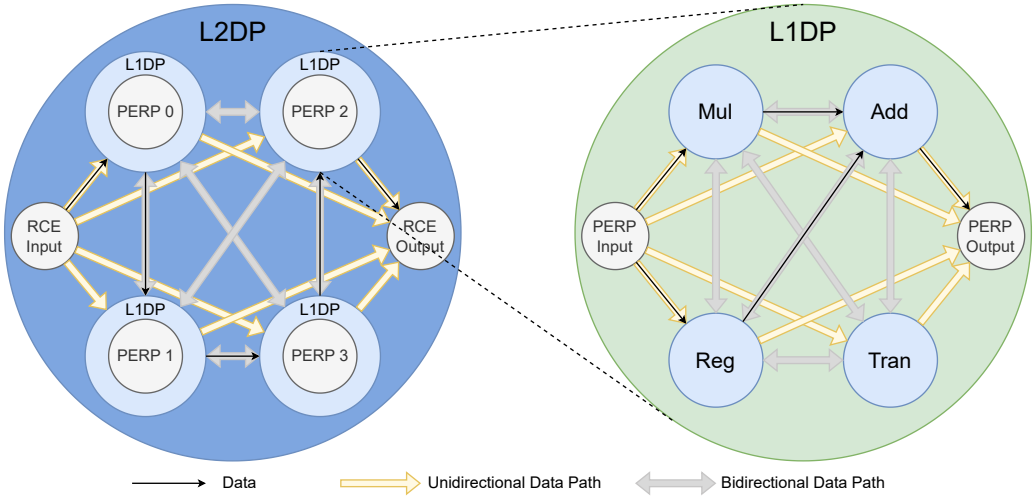

Figure 4. The structure overview of L2DP and L1DP.

3.3. Processing element of reconfigurable pipeline (PERP)

Each PERP comprises three functional modules: an addition module, a multiplication module, and a transformation module, as shown in Figure 3. PERPs are connected through a first-level processing element data path (L1DP), using the same connectivity method as L2DP.

The addition module can simultaneously perform four 32-bit floating-point additions or one 64-bit floating-point addition. The multiplication module can simultaneously execute four instances of 32-bit floating-point multiplication or one instance of 64-bit floating-point multiplication. The memory control module primarily performs logical operations and cache control on data, such as memory protection, data shifting, data rearrangement, and similar operations. Due to the different clock cycles required for various operations and to ensure that each piece of data enters the module at the correct time, each module can implement delayed or continuous data output. The required delay time is specified by the input together with instructions specifying the module's state.

3.4. Instruction subsystem

Given the DSP-OPU structure, there is no need for instructions and data to be input simultaneously. Additionally, there are no extra requirements for the order of instruction input. Therefore, the instructions generated by the compiler will be stored in the instruction cache module. After retrieving these instructions from the instruction cache, the unpacking module first segments them into control

instructions for different modules and transmits them to the decoders of each module. Instructions destined for multiple RCEs also undergo further processing through a multi-engine navigator to ensure rapid and precise instruction input.

3.5. Level 3 data path (L3DP)

The architecture we propose ensures versatility by configuring different data paths, achieving high parallelism for various functionalities. The L3DP, located external to the RCE, is a critical component ensuring structural generality and is primarily divided into three parts: the external input-output data path (EIODP), memory input-output data path (MIODP), and computation module input-output data path (CIODP). The data path connections among these three parts are similar to L2DP.

The EIODP in our DSP-OPU supports up to eight 32-bit data paths or four 64-bit data paths for input or output, with data paths directed by instructions to either the RCE or the BRAM components in the architecture. Without BRAM caching, sending data directly through the CIODP to the RCE can reduce instruction complexity and computation delay, and enhance the versatility of the structure in our design. When BRAM caching is needed, the design meets the data retrieval rules, enabling the characteristic of high parallelism of our architecture and supporting specified algorithms like FFT that meet the requirement of repeated data access.

The MIODP consists of 32 channel groups with 128 channels in total. Each channel is constructed to include two 32-bit data paths, facilitating efficient data transmission and processing. What sets the MIODP apart is its remarkable flexibility in data handling. The MIODP can support up to 256 32-bit data paths or 128 64-bit data paths for input or output according to related instructions. In order to ensure the flexibility of data access, each channel group can be connected to all other channel groups within the CIODP or EIODP.

The CIODP is equipped with 16 input channel groups and 16 output channel groups with 128 channels in total, and the channel includes two 32-bit data paths. Unlike the MIODP, the functions of the channel groups in the CIODP are fixed and do not support the conversion of input and output. To minimize control complexity and ensure the versatility of the structure, the CIODP configuration is based on channel groups, similar to that of the MIODP.

3.6. Multiple engines shared memory (MESM) and controller (MESMC)

According to the computation flows of various DSP algorithms, some algorithms require frequent access to previously used data. To ensure efficient parallel data retrieval, data caching is necessary. The caching capacity can be specified by allocating different block RAM (BRAM) according to the amount of resources in the target FPGA. The BRAM module in our DSP-OPU includes 32 BRAM groups, each with 4 BRAM instances and eight 32-bit data paths. This configuration supports up to 256 32-bit data paths or 128 64-bit data paths simultaneously. For simplified control and consistency, the channels in each BRAM group are mapped one-to-one with the channels in the MIODP.

3.7. Data pre-processing

Leveraging the distinctive structure of DSP-OPU, there is no simultaneous entry requirement for instructions and data into the FPGA. Consequently, the compiler transfers data to the FPGA in the form of instruction blocks and data blocks. Preliminary processing of the data received from the compiler is

essential to identify instruction and data blocks. Subsequently, these blocks are transformed into the necessary data structures for transmission to the instruction subsystem or L3DP.

4. Instruction set architecture

4.1. Instruction set overview

A concise and efficient instruction set is of paramount importance for a reconfigurable processor [26, 27]. In our DSP-OPU, we classify instructions into two types: computational instructions (C-type) and memory instructions (M-type). The C-type instructions are under the length of 32 bits, responsible for the state specifications of computing modules and configurations for various data paths. The M-type instructions are also 32-bit long and are responsible for configuring memory address, memory state, and initiating the commencement of data transferring.

The instruction set design enables our DSP-OPU to dynamically generate instruction streams that can be directly executed on hardware circuits. The instruction streams can control the operation mode of underlying hardware modules after loaded into the FPGA board. On the software side, the compiler analyzes user requirements, i.e., algorithm type, data type, data length, number of computation points and filter order, and transforms them into bitstreams with specific configurations. The DSP-OPU ISA includes instructions that control data path connections, computation module state, and data access. By calling and combining these instructions, we can describe the computation process and support the execution of arbitrary DSP algorithms. Note that our DSP-OPU can support multiple DSP algorithms simultaneously by specifying the number of algorithms with required types and configurations.

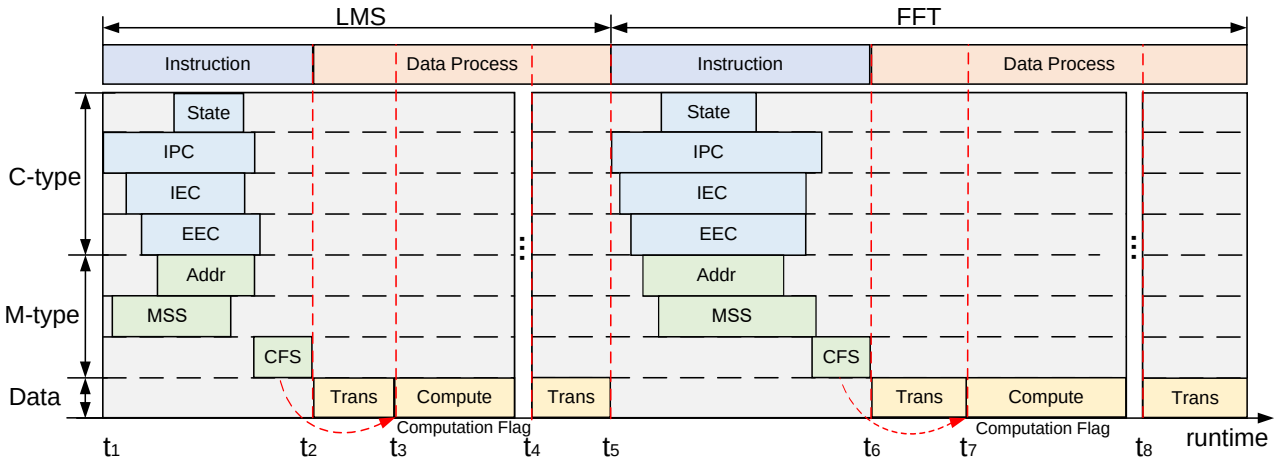

Figure 5. Instruction scheduling of a single proposed RCE.

The instruction operational process of the proposed RCE is elucidated in Figure 5. The beginning of the computation is predetermined by the Compute Flag Signal (CFS) instruction. At t_1, the compiler initiates the transmission of instructions to the hardware in the form of instruction blocks. The order of instruction block transmission is determined by the number of instructions of the current functional type rather than the instruction functionality. To be specific, we prefer to transmit blocks with a higher instruction number. This is because the construction of data paths does not need a specific order, but

the execution speed of instructions of the same functionality is limited. The more instructions of the same functional type, the longer the execution time required. Specifically, the underlying hardware executes instructions from t_1 to t_2. The execution time depends on the total number of instructions of the current instruction type. At t_2, algorithmic data transferring begins, triggering post-processing at t_3. Data processing and output end at t_4, followed by the configuration of the next algorithm. The configuration time is variable according to the complexity of different algorithms.

Based on the above execution process, we realize a complete separation of instructions and data, where they are not input at the same time. The advantage of this approach is that the execution of instructions can be simplified, and there is no need to adjust the instruction input order. Moreover, in the instruction optimization process, after the instructions and data are transferred into a certain core, other operations are not needed for this core. As a result, when a certain core is performing data processing, other cores can be configured in parallel, which makes full use of hardware caching and significantly improves the throughput of multi-core tasks.

4.2. Computation instruction

The C-type instructions are determined by the opcode and can be divided into four types.

Module state specification. The module status specification (state) instruction defines the operation type and synchronization operation of the computation module within the PE. It controls the operation type within a single PE and determines whether a delayed operation is necessary, along with specifying the delay duration. For example, an adder can be adjusted for operations such as 32-bit floating-point addition, 32-bit floating-point subtraction, or 64-bit floating-point addition. Considering the similarity among various operations, e.g., addition and subtraction, configuring the operation type through instructions facilitates minimal control overhead, while ensuring synchronized data output of distinct operation types with different computation cycles.

Inter-PE connection. Inter-PE connection (IPC) instruction is responsible for the connectivity of the L1DP within the RCE. The existence of L1DP serves as a partial substitute for the data scheduling operations instruction found in the general instruction set for the engines. Moreover, it significantly enhances the data scheduling capability of the engines, particularly in the context of DSP algorithms.

Inter-engine connection. Inter-engine connection (IEC) instruction specifies the connectivity of the L2DP between engines. Serving as a bridge between L1DP and L3DP, L2DP significantly reduces the resource overhead of data path control. Concurrently, the existence of L2DP simplifies multi-engine optimization. The collaborative acceleration of a single algorithm by multiple engines can be achieved by serially connecting them to the RCE through L2DP.

External engine connection. External engine connection (EEC) instruction is utilized to specify the connectivity of the L3DP for engines. L3DP functions as a bridge connecting external data, MESM, and RCE. The existence of L3DP broadens the algorithm support range for the entire processor. Complex algorithms can be supported by employing operations such as data caching and rescheduling through L3DP.

As mentioned above, C-type instructions are primarily employed for reconfiguring computation modules and various levels of data paths. The construction of diverse pipelines using different data paths enables the realization of a wide range of algorithmic operations. In multiple RCE architectures, the number of RCEs directly influences the complexity and total length of the pipelines. Thus, the number of RCEs directly affects the variety and types of algorithms that can be supported. Simultaneously, this

method efficiently utilizes hardware computation resources through a pipelined format, avoiding the need for complex data scheduling operations and effectively harnessing the abundant computational resources of FPGA.

4.3. Memory instruction

The M-type instruction is responsible for performing configuration operations related to memory functionality with a length of 32 bits. The definitions for three distinct types of M-type instructions are as follows:

BRAM group address specification. BRAM group address specification (Addr) instruction is designed to identify the BRAM group responsible for storing data and the initial address within that BRAM group. Due to various data requirements across multiple RCEs, this instruction is essential for precise data allocation, specifying exact data storage addresses, and providing offset addresses to ensure the accurate reading and storing of different data.

Memory state specification. Memory state specification (MSS) instruction is designed to specify the status of individual channels in the memory. It encompasses details such as whether the channels are enabled, operating in data input or data output mode, whether the output is looped, the total number of outputs, and other relevant parameters.

Computation flag signal. Compute flag signal (CFS) instruction is designed to control the output of data from the memory. The instruction indicates the computation commencement, where data starts to flow out from the BRAM. A single CFS instruction is adequate to initiate the execution of each computation stage. For algorithms requiring data buffering through memory, it is necessary to wait for data buffering to complete before the instruction input. Consequently, this instruction can specify the number of cycles to delay execution. Algorithms not requiring data buffering through memory do not need to issue this instruction.

5. Compiler

Compilers serve as vital tools to bridge the gap between user requirements and low-level hardware computation. Specifically tailored for our DSP-OPU, we carefully design a dedicated compiler for our end-to-end execution. This compiler parses models and optimizes data structures to reduce the overhead of algorithm execution. It enables the transformation of DSP algorithm models into a series of instructions executable by DSP-OPU through compilation. These instructions are then transmitted via PCIe to the cache of our DSP-OPU and subsequently executed by the decoding module. As shown in Figure 6(a), the end-to-end compiler consists of three different stages: model parsing, instruction generation, and data optimization. During the model parsing and instruction generation stages, the compiler's primary function is to extract essential information from the model file and generate the necessary instruction sequences. In the data optimization stage, the compiler must structurally optimize the model file and instruction sequences based on hardware characteristics.

5.1. Model parsing

We abandon the conventional general-purpose instruction architecture, trading off some universality to achieve enhanced processor performance comparable to custom accelerators. The instructions rely heavily on the interconnection between various levels of data paths. Consequently, for different

algorithms, even if the differences between them are minor, the types and quantities of corresponding instructions can be significantly diverse. It poses a significant challenge in designing a compiler that supports a series of algorithms.

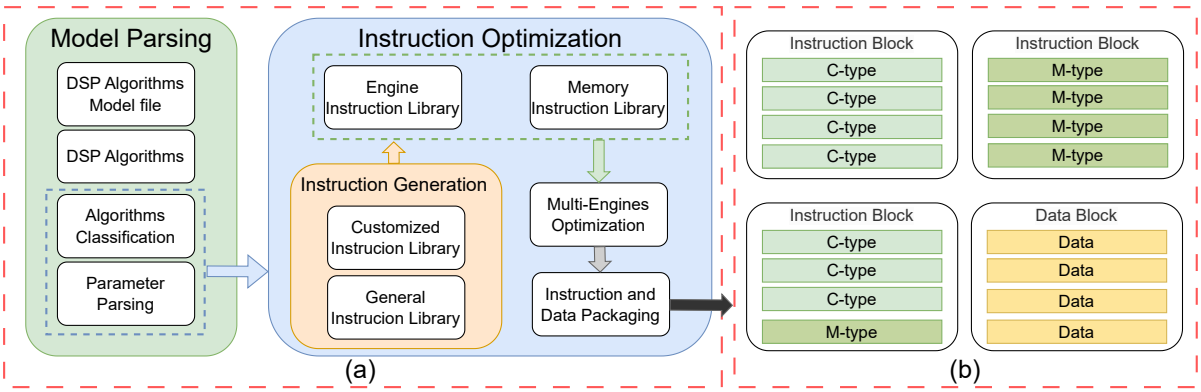

Figure 6. Overview of our flexible compiler. (a) The three stages of our compiler, i.e., model parsing, data and instruction optimization, and instruction generation. (b) Data block structure consisting of detailed data and related instructions.

Within the compiler, two instruction libraries are introduced, one contains pre-set algorithmic instruction sequences, and another comprises general computational flow instruction sequences. The compiler can adeptly assemble these instruction fragments, generating corresponding instruction sequences for various algorithms or models of different sizes, thus minimizing situations where certain algorithms are left entirely unsupported.

In order to fully harness these two instruction libraries, the compiler is initialized by determining the current algorithm type. The codes supporting the algorithms are stored within the compiler. In the first phase of the model parsing stage, the compiler can deduce the current algorithm type for digital signal processing based on user invocations of pre-stored algorithmic functions and extract the required parameters. Additionally, the compiler can analyze the needed algorithm type or instruction fragments based on the description provided by the user.

5.2. Instruction generation

Following the details of the algorithm type and parameters, the compiler determines whether the current algorithm is part of the existing algorithms. If so, the compiler retrieves the instruction fragments for that algorithm directly from the existing algorithmic instruction library and assembles the required instruction sequence. If not, the compiler chooses appropriate instruction segments from the computational flow instruction library based on the computational process of the current algorithm and generates the instruction sequence.

5.3. Data and instruction optimization

To fully exploit the hardware capability, it is crucial to further optimize the instructions and data based on the hardware architecture. DSP-OPU boasts multiple computation engines, and the key challenge in instruction optimization lies in achieving the multi-engine acceleration for algorithms. To tackle this challenge, we have integrated two additional instruction libraries into the compiler: the engine

instruction library and the memory instruction library. Similar to the existing algorithmic instruction library and computational flow instruction library, these two new libraries mainly store instruction fragments related to the linkage methods of engines and memory.

At the outset of the optimization phase, the compiler evaluates the scale of the current task based on the model parameters obtained in the previous stage. Subsequently, it selects the necessary instruction fragments from the two instruction libraries and integrates them into the existing instruction sequence to generate an optimized instruction sequence.

After optimizing the instruction sequence, we ensure the full utilization of all engines within the hardware. However, achieving accurate computations requires us to confirm the correct allocation of model data to their respective computation engines. According to the instruction design in our DSP-OPU, our instruction sequence imposes no requirements on the input order. Therefore, we evaluate the current method of instruction partitioning and data storage based on the data path configuration information and memory address details within the instructions. Using this information as a foundation, we appropriately partition the data and instructions.

We package the optimized instructions and data into data blocks, as shown in Figure 6(b), and transmit them to the hardware. Considering the notable differences in the quantity of different instructions, we prioritize the transmission of the larger quantity of instructions to maximize the transfer rate and hardware decoding efficiency. In addition, we employ distinct identification markers for partitioning, ensuring that the hardware can correctly distinguish between different types of instructions or data.

6. Experiment

6.1. Experimental setup

For the implementation of DSP-OPU, we deploy our work on the 325T FPGA and the U200 FPGA. Two RCE units are deployed on the 325T, and eight RCE units are deployed on the U200. We implement DSP-OPU by Verilog HDL and synthesis it by Vivado 2020.1. For the compiler, we design and implement it in C++.

6.2. Theoretical and practical speedup

In theoretical evaluations, we compared the theoretical and practical accelerations of DSP-OPU with the C6678 SoC, a widely used and high-performance digital signal processor, as depicted in Table 3. The theoretical values for C6678 are derived from its official documentation applicable to 32-bit floating-point data. In practice, we utilize the throughput of LMS as a benchmark for comparison. DSP-OPU exhibits actual acceleration far surpassing the theoretical ratio when compared to C6678, owing to the efficient data path mechanism that achieves acceleration effects comparable to custom accelerators.

Table 3. Theoretical and practical speedup of DSP-OPU compared with the C6678 SoC chip.

Designs	Theoretical RCE performance (GMAC/s)	Practical throughput (MS/s)
C6678	44.89 (1×)	0.952 (1×)
Ours-325T	38.4 (0.86×)	**10 (10.50×)**
Ours-U200	71.6 (1.60×)	**18.67 (19.61×)**

We also evaluated the proportion of different data in transmission, as depicted in Figure 7. Specif-

ically, we propose a statistical analysis of the proportion of logic and instruction data for various algorithms under different sizes. It is noticeable that the instruction data is of a smaller magnitude. This enables our architecture to be more effective, as the data transmitted from the host are mainly used for logic computation, alleviating the performance saturation issue resulting from transmission bandwidth limitations in a multi-task scenario.

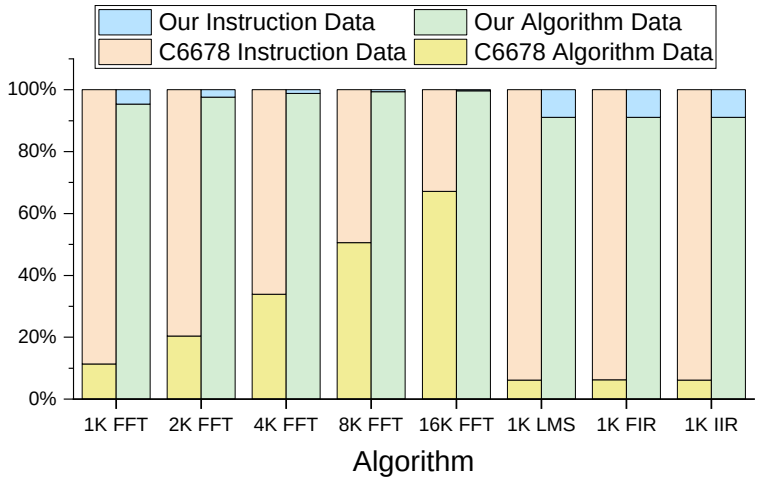

Figure 7. The proportion of instruction and logic data in DSP-OPU compared with C6678. The larger logic data scale indicates better transmission efficiency.

6.3. Resource consumption and overlay latency performance

In the context of the FFT algorithm, we establish a 1024-point radix-2 FFT as a reference benchmark. For filtering algorithms, our attention is focused on 32-tap LMS, 5th-order IIR, and 32-tap FIR filters, and the performance on the C6678 SoC is also included. Specifically, the results of both the C6678 SoC and DSP-OPU are based on 32-bit floating-point data.

Due to the absence of support for LMS and IIR algorithms in Xilinx IP, and the suboptimal throughput of the 1024-point FFT IP at 64 MS/s, we implement customized IPs based on Xilinx IP to address these limitations. The customized IPs employ 32-bit fixed-point data, providing certain advantages in terms of resource utilization and throughput. The customized IP leverages a similar implementation approach to DSP-OPU, except that it encapsulates the implementation of each algorithm as a separate IP, in order to evaluate the resource consumption when deploying multiple IPs simultaneously.

Table 4. The resource utilization of DSP-OPU. Results are based on two optional FPGA platforms under different resource sizes.

FPGA	Num of	LUT		FF		DSP		BRAM	
	RCE	Usage	Utilization	Usage	Utilization	Usage	Utilization	Usage	Utilization
XC7K325T	2	197,624	96.97%	118,736	29.13%	512	60.95%	256	57.53%
Alveo U200	8	770,496	65.17%	474,944	20.09%	2048	29.94%	256	11.85%

Table 4 details the hardware resource utilization. In this context, 325T employs 2 RCEs, and U200 utilizes 8 RCEs. The U200 retains unused resources suitable for additional RCE deployment, in order to

refrain from deploying more RCEs to avoid potential frequency impacts associated with excessive data paths. We further summarize the overlay performances of DSP-OPU compared with other works in Table 5. The frequency (MHz), latency (us), throughput (MS/s), power (W), LUT, FF, DSP, and BRAM utilization are reported. The power consumption in Table 5 is obtained by the analysis report of the Vivado Power Estimator, which reflects the design characteristics of the FPGA chip.

Table 5. The overlay performance of DSP-OPU compared with other implementations.

Designs	Plantform	WL	Alg	Order	Point	Freq (MHz)	Latency (ns)	Throughput (MS/s)	Power (W)	LUT	FF	DSP	BRAM
Pakize et al. [28]	Virtex-7 XC7VS330T	16	FFT	\	1024	339	\	1261	\	12.5k	22.5k	150	6
Garrido et al. [23]	Virtex-6 XC6VSX475T	16	FFT	\	1024	475	\	1900	\	10.3k	10.3k	12	0
	Virtex-7 XC7VS330T	16	FFT	\	1024	680	\	2720	1.6	10.5k	10.5k	12	0
Ezilarasan et al. [25]	Virtex-4 XC4VFX12	\	LMS	32	\	109	9.152	109	\	\	\	\	\
	Virtex-5 XC5VLX30	\	LMS	\	\	111	8.95	112	\	\	\	\	\
Potsangbam et al. [24]	Virtex-7 XC7AS200T	\	FIR	3	\	120	5.12	195	8.053	\	\	\	\
		\	IIR	1	\		4.72	212	5.293	\	\	\	\
DSP SoC	TMS320C6678	**32**	FFT	\	1024	1400	\	135.8	15.00	\	\	\	\
		32	LMS	16	\		1044	0.952					
		32	IIR	7	\		105	9.52					
		32	FIR	32	\		12.56	79.8					
Customized IP	Xilinx Alveo U200	32	FFT	\	1024	200	\	200	6.32	59k	116k	153	58
		32	LMS	16	\		20.08	50					
		32	IIR	7	\		5.09	196					
		32	FIR	32	\		5.09	197					
DSP-OPU	**Kintex-7 XC7K325T**	**32**	**FFT**	****	**1024**	**150**	****	**480**	**3.4**	**197k**	**118k**	**512**	**256**
		32	**LMS**	**16**	****		**100**	**10**					
		32	**IIR**	**7**	****		**6.67**	**150**					
		32	**FIR**	**32**	****		**6.67**	**150**					
DSP-OPU	**Xilinx Alveo U200**	**32**	**FFT**	****	**1024**	**280**	****	**896**	**12.3**	**770k**	**475k**	**2048**	**256**
		32	**LMS**	**16**	****		**53.57**	**18.67**					
		32	**IIR**	**7**	****		**3.57**	**280**					
		32	**FIR**	**32**	****		**3.57**	**280**					

Our implementation is superior to C6678 SoC in throughput and processing speed, which is mainly due to the high parallelism of the DSP-OPU engine. In terms of FFT performance, our architecture deployed on the U200 FPGA achieves 6.6 × higher throughput compared to C6678. For the LMS algorithm, the throughput is improved by 19 ×. As for the filter algorithm, we are 3.5–29 × faster and 1.2 × lower in power consumption with the same filter order. Compared with customized IP, the latency of DSP-OPU is higher in LMS due to the longer cycle of the floating-point data type. In other algorithms, the latency is basically the same in different data types.

Note that the four customized IPs can achieve execution throughput similar to DSP-OPU. However, they can only support the four algorithms in Table 5 with great limitations. Specifically, for the FFT algorithm, the customized IPs cannot support more than 1024 points. For the LMS algorithm, the customized IPs cannot support the case that the filter order is more than 16. At the same time, the customized IPs cannot support IIR and FIR algorithms with larger orders. In contrast, our DSP-OPU is compatible with complex algorithm configurations, and can support up to 100,000 points for the FFT algorithm and up to 128 filter orders for the FIR algorithm.

Compared with algorithm accelerators deployed on FPGA, our architecture demonstrates superior performance in all areas except FFT. In contrast to the accelerators designed by Pakize et al. [28] and Garrido et al. [23], our architecture uses 32-bit floating-point data, while their accelerators utilize 16-bit

fixed-point data. Doubling the word length within the same architecture results in a fourfold increase in throughput. Additionally, the use of floating-point computation generally offers better precision and higher latency compared to fixed-point computation. If the same word length is used, our architecture can achieve equal or even better throughput than other architectures.

6.4. Multi-task comparison

In Figure 8, a comparison is made between the DSP-OPU architecture and the C6678 chip in terms of time for different multi-task scenarios. We utilize a workload from the radar domain [29], where 16 segments of data are applied to the FFT, FIR, and IFFT operations. Each segment consists of 1024 32-bit floating-point data, and both the ideal and practical time are compared. The ideal time is the sum of the computation time for the algorithms on the cores, while the practical extra time includes the transmission time for instructions and algorithm data. The transmission rate is determined by the PCIE rate supported by C6678.

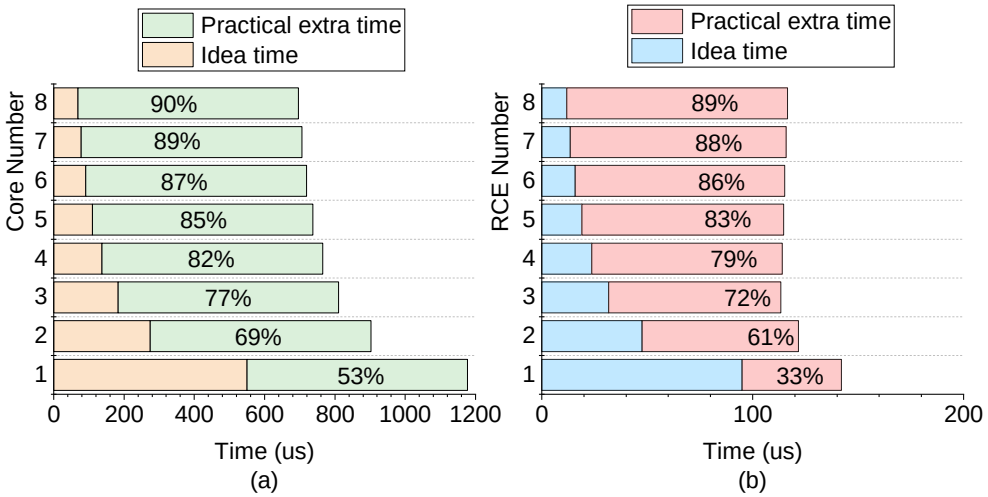

Figure 8. Comparisons of ideas and practical extra latency in multi-task workloads. (a) TMS320C6678 SoC chip, (b) DSP-OPU on Alveo U200.

It is evident that DSP-OPU outperforms C6678 by a significant margin in both ideal time and practical extra time. Moreover, the proportion of the practical extra time is also lower than that of C6678. If the instruction data in DSP-OPU have the same size as C6678, the proportion of practical extra time should be increased. Furthermore, our proportion of practical extra time is lower than C6678 and offers higher transmission efficiency.

6.5. Influence of different RCE numbers

In Table 6, we compare the influence of different numbers of computing engines. With an increased number of computing engines, longer reconfigurable pipeline lengths are available, making it more suitable for accelerating complex algorithms. In order to facilitate the performance analysis of multi-core design, we set the operating frequency of both 325T and U200 FPGAs to 150MHz. Our architecture has two computing engines when deployed on 325T, and there is a significant decrease in throughput

when the order of the LMS algorithm exceeds 32. On the other hand, the architecture is deployed on U200 with 8 computing engines. It can support higher-order LMS algorithms without any change in throughput. For the FFT algorithm, the throughput corresponds to detailed points, and has little relationship with the increase of the core number. The implementation of the FIR algorithm is relatively simple, and two computation engines can already support the filter order of 64, which indicates that the number of engines does not affect FIR throughput.

Table 6. The throughput of DSP-OPU under different RCE numbers.

Platform	Num of RCE	Algorithm	Order	Point	Frequency	Throughput (MS/s)
XC7K325T	2	FFT	\	1024	150	480
			\	2048		436
			\	4096		400
		FIR	16	\		150
			32	\		150
			64	\		150
		IIR	16	\		150
			32	\		150
			64	\		35
		LMS	16	\		10
			32	\		2
			64	\		0.8
Alveo U200	8	FFT	\	1024	150	480
			\	2048		436
			\	4096		400
		FIR	16	\		150
			32	\		150
			64	\		150
		IIR	16	\		150
			32	\		150
			64	\		150
		LMS	16	\		10
			32	\		10
			64	\		10

6.6. Energy efficiency comparison

In Figure 9, we compare the energy efficiency of the DSP-OPU architecture with the C6678 chip and customized IP. The efficiency is calculated by dividing throughput by power. It is worth noting that we use a logarithmic scale in the graph. Therefore, the actual differences are larger than the results in Figure 9. The efficiency of our architecture deployed on 325T is 8.3–70 × better than that of C6678.

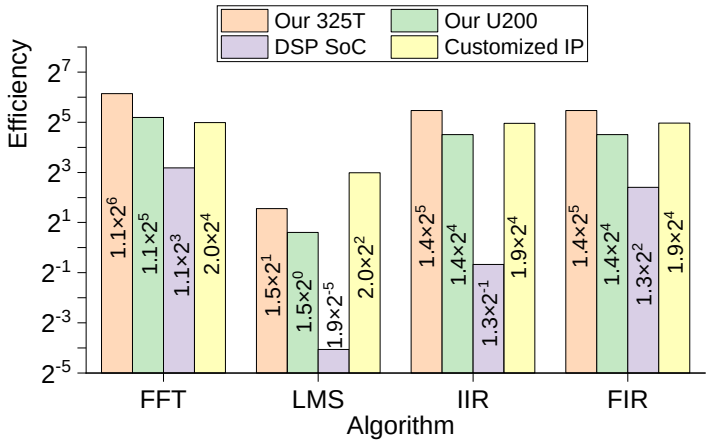

Figure 9. Energy efficiency of the DSP-OPU architecture compared with the C6678 chip and customized IP.

7. Conclusions

In this paper, we propose DSP-OPU, a reconfigurable and overlay processor based on FPGA to implement various DSP algorithms. First, we introduce a reconfigurable data path that allows for the reconfigurable pipeline of specific algorithms by connecting multiple computing cores through instructions. Second, we design an instruction set for DSP-OPU, ensuring the flexibility of our design. A simple and efficient compiler is also introduced to schedule and optimize the computation flow. Compared to C6678, our DSP-OPU achieves up to 29 × speedup and 70 × higher energy efficiency for different algorithms. Compared to FPGA-based implementations for a single algorithm, we achieve up to 4.5 × speedup and 4.4 × better energy efficiency.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

This work was financially supported in part by the National Key Research and Development Program of China under Grant 2021YFA1003602, in part by the Shanghai Pujiang Program under Grant 22PJD003.

Conflict of interest

The authors declare there are no conflicts of interest.

References

1. J. G. Proakis, *Digital Signal Processing: Principles, Algorithms and Applications*, Pearson Education India, 1996.

2. M. K. Masten, I. Panahi, Digital signal processors for modern control systems, *Control Eng. Pract.*, **5** (1997), 449–458. https://doi.org/10.1016/S0967-0661(97)00024-5

3. A. N. Sokolov, A. N. Ragozin, I. A. Pyatnitsky, S. K. Alabugin, Applying of digital signal processing techniques to improve the performance of machine learning-based cyber attack detection in industrial control system, in *Proceedings of the 12th International Conference on Security of Information and Networks*, (2019), 1–4. https://doi.org/10.1145/3357613.3357637

4. M. Frerking, *Digital Signal Processing in Communications Systems*, Springer Science & Business Media, 2013. https://doi.org/10.1007/978-1-4757-4990-8

5. Z. Q. Luo, Applications of convex optimization in signal processing and digital communication, *Math. Program.*, **97** (2003), 177–207. https://doi.org/10.1007/s10107-003-0442-2

6. W. K. Pratt, *Digital Image Processing: PIKS Scientific Inside*, 4^{th} edition, Wiley Online Library, 2007. https://doi.org/10.1002/0470097434

7. E. Sisinni, A. Saifullah, S. Han, U. Jennehag, M. Gidlund, Industrial internet of things: Challenges, opportunities, and directions, *IEEE Trans. Ind. Inf.*, **14** (2018), 4724–4734. https://doi.org/10.1109/TII.2018.2852491

8. D. M. D'Addona, S. Conte, W. N. Lopes, P. R. de Aguiar, E. C. Bianchi, R. Teti, Tool condition monitoring of single-point dressing operation by digital signal processing of AE and AI, *Procedia CIRP*, **67** (2018), 307–312. https://doi.org/10.1016/j.procir.2017.12.218

9. I. Tomkos, D. Klonidis, E. Pikasis, S. Theodoridis, Toward the 6G network era: Opportunities and challenges, *IT Prof.*, **22** (2020), 34–38. https://doi.org/10.1109/MITP.2019.2963491

10. Y. Liu, R. Chen, S. Li, J. Yang, S. Li, S. Bruno, FPGA-based sparse matrix multiplication accelerators: From state-of-the-art to future opportunities, *ACM Trans. Reconfigurable Technol. Syst.*, **17** (2024), 1–37. https://doi.org/10.1145/3687480

11. S. M. Noor, E. John, M. Panday, Design and implementation of an ultralow-energy FFT ASIC for processing ECG in cardiac pacemakers, *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, **27** (2019), 983–987. https://doi.org/10.1109/TVLSI.2018.2883642

12. G. N. Jyothi, S. Sriadibhatla, ASIC Implementation of Low Power, Area Efficient Adaptive FIR Filter Using Pipelined DA, in *Microelectronics, Electromagnetics and Telecommunications*, Springer, Singapore, **521** (2018), 385–394. https://doi.org/10.1007/978-981-13-1906-8_40

13. X. Li, Design of array signal processing system based on TMS320C6678, in *2013 5th International Conference on Intelligent Networking and Collaborative Systems*, IEEE, (2013), 611–616. https://doi.org/10.1109/INCoS.2013.114

14. L. Babitha, U. Somanaidu, C. H. Poojitha, K. Niharika, V. Mahesh, V. Vijay, An efficient implementation of programmable IIR filter for FPGA, in *Innovations in Signal Processing and Embedded Systems: Proceedings of ICISPES 2021*, Springer, (2022), 109–117. https://doi.org/10.1007/978-981-19-1669-4_10

15. A. Saeed, M. Elbably, G. Abdelfadeel, M. I. Eladawy, Efficient FPGA implementation of FFT/IFFT processor, *Int. J. Circuits, Syst. Signal Process.*, **3** (2009), 103–110.

16. A. Paul, T. Z. Khan, P. Podder, M. M. Hasan, T. Ahmed, Reconfigurable architecture design of FIR and IIR in FPGA, in *2015 2nd International Conference on Signal Processing and Integrated Networks (SPIN)*, (2015), 958–963. https://doi.org/10.1109/SPIN.2015.7095408

17. V. Vallabhuni, V. R. S. Rao, K. Chaitanya, S. C. Venkateshwarlu, C. S. Pittala, R. R. Vallabhuni, High-Performance IIR filter implementation using FPGA, in *2021 4th International Conference on Recent Trends in Computer Science and Technology (ICRTCST)*, (2022), 354–358. https://doi.org/10.1109/ICRTCST54752.2022.9781944

18. S. Kavitha, G. Sinduja, M. Srimathi, K. Yogalakshmi, An efficient FPGA implementation of the multiplier-less LMS adaptive filter, in *2023 7th International Conference on Computing Methodologies and Communication (ICCMC)*, (2023), 441–445. https://doi.org/10.1109/ICCMC56507.2023.10084108

19. R. Chen, H. Zhang, S. Li, E. Tang, J. Yu, K. Wang, Graph-OPU: A highly integrated FPGA-based overlay processor for graph neural networks, in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, (2023), 228–234. https://doi.org/10.1109/FPL60245.2023.00039

20. M. T. Khan, M. A. Alhartomi, S. Alzahrani, R. A. Shaik, R. Alsulami, Two distributed arithmetic based high throughput architectures of non-Pipelined LMS adaptive filters, *IEEE Access*, **10** (2022), 76693–76706. https://doi.org/10.1109/ACCESS.2022.3192619

21. M. Kowalczyk, T. Kryjak, Hardware architecture for high throughput event visual data filtering with matrix of IIR filters algorithm, in *2022 25th Euromicro Conference on Digital System Design (DSD)*, (2022), 284–291. https://doi.org/10.1109/DSD57027.2022.00046

22. V. Pathak, S. J. Nanda, A. M. Joshi, S. S. Sahu, High speed implementation of Notch/Anti-notch IIR filter on FPGA, in *2018 15th IEEE India Council International Conference (INDICON)*, (2018), 1–6. https://doi.org/10.1109/INDICON45594.2018.8986985

23. M. P. Garrido, The constant multiplier FFT, *IEEE Trans. Circuits Syst. I Regul. Pap.*, **68** (2020), 322–335. https://doi.org/10.1109/tcsi.2020.3031688

24. J. Potsangbam, M. Kumar, Design and implementation of combined pipelining and parallel processing architecture for FIR and IIR filters using VHDL, *Int. J. VLSI Des. Commun. Syst.*, **10** (2019), 1–16. https://doi.org/10.5121/vlsic.2019.10401

25. M. R. Ezilarasan, J. Britto Pari, M. F. Leung, Reconfigurable architecture for noise cancellation in acoustic environment using single multiply accumulate adaline filter, *Electronics*, **12** (2023), 810. https://doi.org/10.3390/electronics12040810

26. Y. Bai, H. Zhou, K. Zhao, H. Wang, J. Chen, J. Yu, et al., Fet-opu: A flexible and efficient fpga-based overlay processor for transformer networks, in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, (2023), 1–9. https://doi.org/10.1109/iccad57390.2023.10323752

27. Y. Bai, K. Zhao, Y. Liu, H. Wang, H. Zhou, X. Wu, et al., CSTrans-OPU: An FPGA-based overlay processor with full compilation for transformer networks via sparsity exploration, in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, (2024), 1–6. https://doi.org/10.1145/3649329.3657325

28. P. Ergül, H. F. Ugurdag, D. Davutoglu, HC-FFT: Highly configurable and efficient FFT implementation on FPGA, *Turk. J. Electr. Eng. Comput. Sci.*, **29** (2021), 3150–3164. https://doi.org/10.3906/elk-2101-56

29. S. S. Rajput, D. S. Bhadauria, Implementation of fir filter using efficient window function and its application in filtering a speech signal, *Int. J. Electr. Electron. Mech. Controls*, **1** (2012).

AIMS Press