IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 7, 2022, accepted February 17, 2022, date of publication February 21, 2022, date of current version March 3, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3153119

Using HW/SW Codesign for Deep Neural Network Hardware Accelerator Targeting Low-Resources Embedded Processors

EREZ MANOR[1] AND SHLOMO GREENBERG[1,2], (Member, IEEE)
[1]Department of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Be'er Sheva 84105, Israel
[2]Department of Computer Science, Sami Shamoon College of Engineering, Be'er Sheva 84100, Israel

Corresponding author: Shlomo Greenberg (shlomo.greenberg@gmail.com)

This work was supported in part by the High-Tech Scholarship Award, and in part by the GenPro Consortium within the framework of the Israel Innovation Authority's Magnet Program.

ABSTRACT The usage of RISC-based embedded processors, aimed at low cost and low power, is becoming an increasingly popular ecosystem for both hardware and software development. High performance yet low power embedded processors may be attained via the use of hardware acceleration and Instruction Set Architecture (ISA) extension. Efficient mapping of the computational load onto hardware and software resources is a key challenge for performance improvement while still keeping low power and area. Furthermore, exploring performance at an early stage of the design makes this challenge more difficult. Potential hardware accelerators can be identified and extracted from the high-level source code by graph analysis to enumerate common patterns. A scheduling algorithm is used to select an optimized sub-set of accelerators to meet real-time constraints. This paper proposes an efficient hardware/software codesign partitioning methodology applied to high-level programming language at an early stage of the design. The proposed methodology is based on graph analysis. The applied algorithms are presented by a synchronous directed acyclic graph. A constraint-driven method and unique scheduling algorithm are used for graph partitioning to obtain overall speedup and area requirements. The proposed hardware/software partitioning methodology has been evaluated for MLPerf Tiny benchmark. Experimental results demonstrate a speedup of up to 3 orders of magnitude compared to software-only implementation. For example, the resulting runtime for the KWS (Keyword Spotting) software implementation is reduced from 206 sec to only 181ms using the proposed hardware-acceleration approach.

INDEX TERMS HW/SW codesign, SDF Graph, extensible processors, MLPerf tiny.

I. INTRODUCTION

In the last years, the complexity of the embedded platform, such as Internet-of-Things (IoT) devices, has been increasing steadily with the conflicting requirements for high performance and real-time capabilities versus the minimal amount of power and size. Using a general-purpose RISC processor for such systems typically results in a design that fails to meet the application-specific requirement [1].

To achieve the application's desired requirements, one may use extensible RISC-based embedded processors, which offer the flexibility of integrating hardware accelerator and Instruction Set Architecture (ISA) extension. Examples for such processors can be found in the products of various ASIC providers such as Tensilica Xtensa from Cadence [2] and the ARC processor [3] from Synopsys or by FPGA-based hardware providers such as the Intel Nios [4] and Xilinx Microblaze [5].

Traditionally, hardware/software partitioning was carried out manually. However, this approach has become infeasible for complex designs and research efforts have been taken to find alternative approaches. A common approach for the acceleration of an application using an extensible processor usually follows the following stages [6]: (1) develop the algorithm in a high-level programming language (e.g., Matlab, Python); (2) translate the source code application into lower-level programming language (e.g., C), (3) compile the code to the appropriate target hardware machine, and evaluate performance and energy efficiency. Normally, at this stage, the profiling tools are used for further code optimization before applying new custom instructions and using hardware accelerator [7]. Porting an application written in high-level

The associate editor coordinating the review of this manuscript and approving it for publication was Arianna Dulizia.

language onto hardware consumes design time and engineering resources.

An alternative approach for hardware acceleration is based on graph analysis by conversion of an application source code into data flow graphs [8]. The data flow graph approach suggests a sub-graph enumeration and selection phases to extract the appropriate hardware acceleration units. However, the process of selection and enumeration for hardware/software partitioning is NP-hard [9], and therefore it is difficult to implement in large designs.

In this paper, we propose an efficient methodology for hardware/software partitioning applied to high-level programming language at an early stage of the design. The proposed methodology is based on a description of the applied algorithms as a synchronous directed acyclic graph. A constraint-driven method is used for graph partitioning and task scheduling to obtain overall speedup and area requirements. We suggest a unique framework that is based on the proposed methodology to analyze a given source code (in high-level), extract set of hardware accelerators, and implement them into a custom micro-architecture model. The proposed implementation model embodies a hybrid hardware accelerator and scheduler algorithm targeting microcontrollers with limited memory resources.

The key contributions of this paper are as follows:

- Developing a new methodology for hardware/software application partitioning applied to high-level programming language at an early stage of the design.
- Suggesting a constraint-driven method and a unique graph clustering and scheduling algorithm for SDF graph partitioning.
- An efficient framework for automatic custom instruction definition, sub-graph enumeration and selection, and hardware implementation.
- Evaluation of the proposed method using the common MLPerf Tiny benchmark and demonstrating a significant speedup compared to pure software implementation.

The rest of this paper is organized as follows: Section II presents general background related to this paper, while Section III presents an overview of related work. Section IV provides a thorough description of the proposed methodology. Finally, experimental results are presented in Section V, while conclusions are given in Section VI.

II. BACKGROUND

In this section, we provide background for two areas referred to in this work: (1) A particular class of data flow models called Synchronous Data Flow (SDF) graph, and (2) an ISA extension feature and the usage of hardware custom instruction.

A. SYNCHRONOUS DATA FLOW

Synchronous Data Flow (SDF) is a special case of the data flow model of computation [10]. Under the SDF paradigm, algorithms may be described as directed graphs where the nodes represent computations (or functions/tasks), and the arcs represent data paths. The data flow principle is that any node (i.e., task) can fire (perform its computation) whenever input data are available on its incoming arcs. A node with no input arcs may fire at any time. This implies that many nodes may fire simultaneously, and hence tasks are considered to be concurrently performed. Because the program execution is controlled by the availability of data, data flow programs are said to be data-driven [10].

SDF graphs may be mapped into various hardware/software partitioning to guarantee required performance while facing hardware resources constraints. Fig. 1 shows an example of implementing a given software application with four dedicated hardware accelerators. Any software/hardware partitioning should comply with a timing constraint, T_{max} which defines the total allowed application runtime.

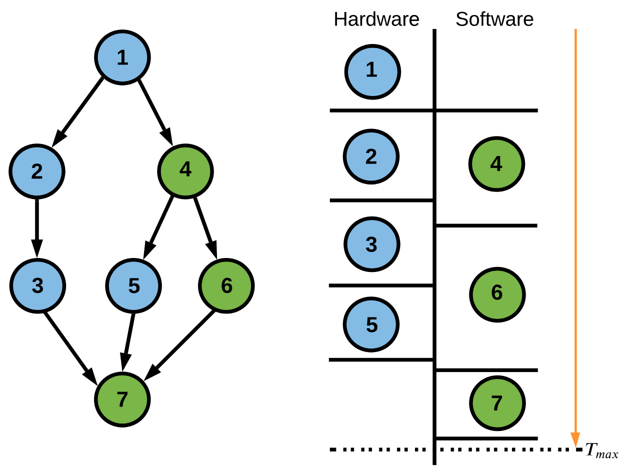

FIGURE 1. Hardware/software partitioning example using 4 dedicated hardware accelerators.

The structure of an SDF graph can be represented in a compact form using the topology matrix, Γ. Each row of Γ corresponds to an arc, and each column corresponds to a node in the SDF graph. An element $\Gamma(a, n)$ in the topology matrix specifies the rate of the data that flows from node n along arc a. SDF is a common approach, and its effectiveness for hardware and software partitioning has been approved [11], [12].

B. CUSTOM ISA EXTENSION

ISA extension is a common approach used in various microcontrollers and DSP, such as Cadence Tensilica Xtensa [2], Synopsys ARC processor [3], CEVA-Xtend [13], Intel Nios [4], and Xilinx Microblaze [5].

In this work, we adopt the Intel paradigm for custom instruction [14]. Fig. 2 shows the custom instruction architecture for the Nios-II embedded processor. As typically used in ALUs, the custom logic has two operand inputs (*dataa* and *datab*) and one output (*result*).

As shown in the figure (right side), there are three modes of custom instruction: (a) combinational, (b) multi-cycle, and

(c) extended mode. In a combinational custom instruction mode, an instruction is completed in a single clock cycle. This mode requires a *result* output port and may have two optional input ports (*dataa* and *datab*). The custom instruction receives values from two source registers and writes the result to a destination register. Multi-Cycle (sequential) custom instructions consist of a logic block that requires two or more clock cycles to complete an operation. The *start* and *done* ports participate in a handshaking scheme to determine when the custom instruction execution begins and is complete. This mode allows adding an interface to communicate with a logic outside the processor. An extended custom instruction allows a single custom logic block to implement several different operations. An extension index *n* is used to specify the logic operation. Ports *a*, *b*, and *c* specify the internal registers from which data is accessed.

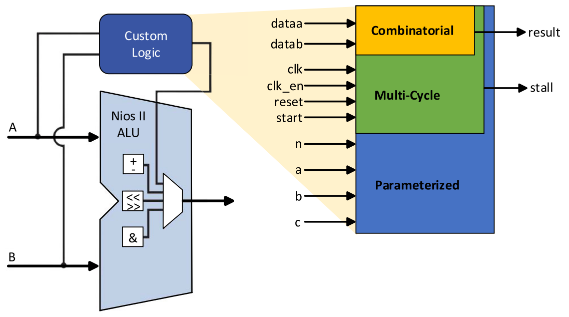

FIGURE 2. **A custom instruction architecture [14].**

III. RELATED WORK

In the following section, we review related work concerning the following specific fields: Graph-based methods for extracting hardware accelerators and accelerator coupling approaches.

A. GRAPH-BASED ACCELERATOR EXTRACTION METHODS

The common flow for extracting a custom instruction is as follows: First, the application code is transformed to an appropriate data-flow graph. Then, an enumeration phase based on the flow graph is carried out for enumerating all the sub-graphs satisfying a given set of micro-architectural constraints. Each sub-graph is considered as a possible candidate for custom instruction. Finally, a sub-graph selection phase is performed to select the most profitable sub-graph subsets as hardware accelerators.

Xiao *et al.* [8] present a design flow that accepts a C/C++ application program and translates it to a data-flow graph using the GeCoS compiler [15]. Then, a search for all the sub-graphs that satisfy a given set of constraints is performed. The algorithm enumerates all the convex connected sub-graphs in a top-down manner. Wang *et al.* [16] propose a more efficient algorithm to solve the time-consuming process of enumeration, adapting a parallel approach to enumerate all connected convex sub-graphs using the MapReduce tool [17]. Xiao *et al.* [18] propose an optimal enumeration algorithm that depends solely on the number of vertex and edges within the connected convex sub-graphs instead of the number of graph nodes. Xiao *et al.* [8] also suggest a subgraph selection technique for extracting custom instructions using genetic and heuristic algorithms.

Another approach for sub-graph enumeration and pattern selection is presented by Zacharopoulos *et al.* [19], [20]. They present a new approach that is based on intermediate representation analysis. The analysis is performed using the LLVM compiler toolchain [21]. The LLVM is a C compiler that defines a common, low-level code representation in Static Single Assignment (SSA) form. The analysis investigates a function-call graph to determine which parts of the application should be implemented in hardware. They provide a cost measure (required resources) and merit (potential speedup) for all candidate accelerators for both hardware and software implementation. The set of accelerators that results in a maximum speedup while still meeting the hardware resource constraints is selected as the optimized hardware/software partitioning choice.

Wijesundera *et al.* [22] suggest an approach for rapid hardware/software partitioning at a fine-grained (basic block) level that can be applied to resource constraint IoT applications. They present a methodology for analysis of data communication cost between basic blocks and memory components and a heuristic formulation to select the most profitable hardware/software partitioning. Zuo *et al.* [23] present a coarse-grained C code partitioning using a dedicated profiling tool to suggest some possible hardware and software implementations in terms of latency, power, and area.

We propose a different approach for sub-graph enumeration and pattern selection in this work. Our partitioning approach is at the basic block level and is intended for resource-constrained devices [22]. We support a higher level of abstraction compared to the C/C++ mentioned in previous works [8], [19], [22]. Using high-level language allows performing a codesign hardware/software analysis in an early stage of the design. We developed a unique graph conversion tool, specially designed for high-level code, to convert the application into SSA intermediate representation [19]. The SSA form is converted into an appropriate SDF Directed Acyclic Graph (DAG). Then, a sub-graph clustering is performed using a top-down manner approach presented in [8]. However, we suggest including disjoint convex sub-graphs, which share common inputs, as well. We adapt the heuristic approach for sub-graph selection presented in [8] and a similar cost function. We extend the selection methodology to support both clustering and scheduling for the optimal solution within design specifications.

B. ACCELERATOR COUPLING APPROACHES

There are two classic implementation models for coupling the accelerator to a given processor, as presented by Cota *et al.* [24]: Tightly-coupled and Loosely-coupled.

Tightly-coupled accelerators consist of one or more hardware functional units which can accelerate critical portions

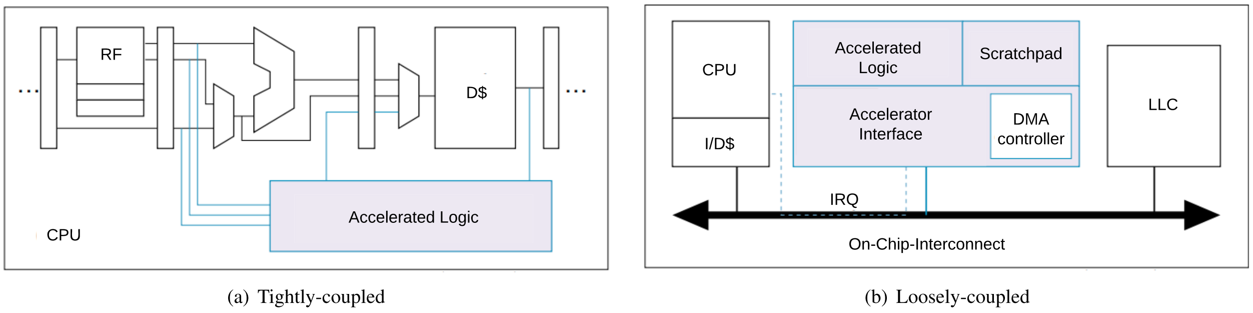

FIGURE 3. Host/accelerator interfacing: (a) Tightly coupled, (b) Loosely coupled [24].

of the application kernel. In a tightly-coupled approach, the accelerator is considered as an additional functional unit that is directly connected to the CPU data-path [8], [25], [26] as shown in Fig. 3.a. In this scenario, the accelerators are integrated into the conventional software flow and therefore are performed sequentially. A single custom instruction can control accelerators that are tightly coupled to the host pipeline through an instruction set extension.

Loosely-coupled accelerators are located outside the CPU core and interact with the CPU through an on-chip interconnect, as shown in Fig. 3.b. A loosely-coupled co-processor computation model, which executes computational-intensive parts that loosely interact with the application, is presented in [19], [27]. A Loosely-Coupled approach enables the CPU and the hardware accelerator to run concurrently. Data exchange with the processor is performed via shared memory. Therefore, the performance gain can be affected by the communication and control overhead.

Vassiliadis *et al.* [28] and Sun *et al.* [29] proposed a hybrid approach to integrate a tightly-coupled and loosely-coupled accelerator. An approach for a tightly-coupled accelerator with direct memory access has been proposed in [30]–[32]. This approach suggests hardware architecture in which the accelerator can directly access a section of the CPU memory-mapped address space for reading or writing operations. The memory-mapped access is performed without the intervention of the CPU. This approach extends the tightly-coupled accelerator approach by allowing a higher granularity of sub-graphs operations to be mapped onto it.

In this work, we propose a tightly-coupled accelerator using direct memory access and supporting multiple sub-graphs calls that are controlled by a unique internal scheduler. This approach enables the implementation of variable operations, represented by different sub-graphs, using a single custom instruction, reducing the number of processor interactions. This solution enables higher granularity of implementing sub-graphs operations compared to previous works, reducing the cost of area and operational latency.

IV. THE PROPOSED APPROACH

This section describes in detail the proposed hardware/software partitioning methodology to analyze algorithms written in high-level language at an early stage of the design.

We provide efficient hardware/software partitioning in terms of resource utilization and speedup merit, using graph analysis for extracting hardware accelerator.

First, we define the problem formulation based on our previous work [33]. Subsection IV-A describes the high-level modeling of the applications, the internal representation of the hardware, and the design space exploration approach.

The proposed methodology includes three design flow phases: 1) Graph Conversion (IV-B): which transforms the high-level algorithm into an analyzable SDF graph representation; 2) Sub-Graph Clustering (IV-C): which detects and groups repeated identical patterns; 3) Graph Scheduling (IV-D), which assigns the computational load to specific hardware and software resources. Fig. 4 depicts the design flow and the analysis tools that are used for the proposed flow.

The graph conversion tool (code-to-SDF) receives the application source code and generates represented SDF graph (including the cost and latency for each node/operation) according to the computational model, which defines the unique operation and their cost. Then a sub-graph clustering is carried out considering the specific system constraints, such as input/output constrain and graph makespan. Finally, a graph scheduling algorithm allocates both software and hardware tasks to satisfy the performance requirements.

Subsection IV-E describes the hardware acceleration implementation model and the system scheduler code.

A. PROBLEM FORMULATION

The space of all possible hardware/software implementations is defined by $S_{N\times M\times K}$ [33]. Where N is the number of nodes in the SDF graph Γ, M is the number of target platforms, and K is the number of possible partitioning. In our case, M is equal to two, representing one software platform and one hardware platform.

Each single partitioning ($k = 1 \ldots K$) is represented by the 2d-matrix ${S^k}_{N\times M}$. The column vector ($i = 1 \ldots N$) represent a specific node (n_i) and the row vector ($j = 1 \ldots M$) presents the target platform. Since a single target platform is allocated to a specific node, the row vector includes only one element with a value '1' while all others are '0'.

The cost for specific criteria (such as area and latency) is presented by the matrix $C_{N\times M\times L}$, where N and M represent the number of nodes and target platform, respectively, while L represents the cost criteria for evaluation (area and latency).

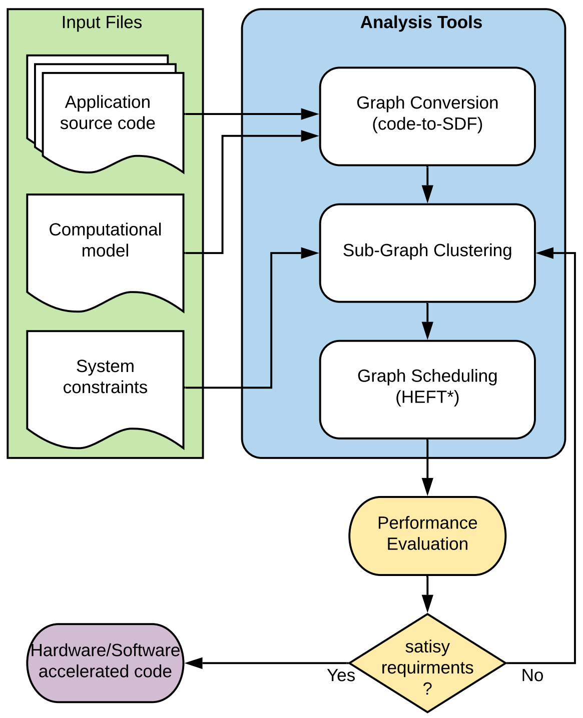

FIGURE 4. Hardware/software partitioning design flow.

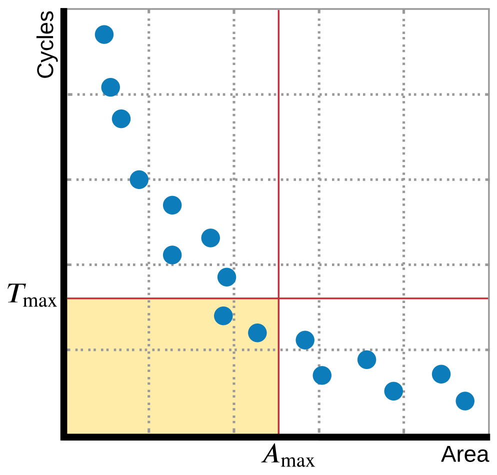

FIGURE 5. Hardware/software partitioning solutions that meet time and area constraints.

Eq. 1 describes the cost for each sub-graph partitioning P^k of the SDF graph in terms of area and latency [33]. The cost is given by the dot-product of the S^k matrix, representing the kth partition, and the C^l matrix, representing the cost of each node n in the SDF graph.

$$P^k{}_L = \sum_{n=1}^{N}\sum_{m=1}^{M}\left(S^k .* C^l\right)(n,m) \quad ,\forall l \in L \tag{1}$$

The proposed partitioning algorithm should comply with the total execution time and area constraint (i.e., total execution time, including communication overhead). Fig. 5 describes a set of possible hardware/software partitioning configurations. The highlighted square region represents a set of possible solutions bounded by the runtime constraint T_{max} and the available hardware resources represented by A_{max}.

B. GRAPH CONVERSION

To translate an application program into an SDF graph, we propose a unique Static Single Assignment (SSA) as an intermediate representation. The proposed SSA-SDF graph represents the data flow of the given algorithm. The SSA-SDF representation is required to identify code loops that can be recursively unrolled to single-line operations.

We developed a unique 'code-to-SDF' tool to perform the SSA conversion. Alg. 1 depicts the pseudo-code of the conversion algorithm. The computational model (defined by the matrix $C_{N\times M\times L}$) includes the cost and latency for each operation and is given by a specific XML file. The SSA-SDF provides a clear view of the application flow for further analysis which is presented in the following methodology phases.

The proposed SSA-SDF analysis assumes that the input size is fixed and, therefore, for each input size, a dedicated SSA-SDF graph should be rebuilt.

Algorithm 1 Code-to-SDF

```
 1: Input ← main function (Matlab source code)
 2: Flatten all sub-function calls into a single call function
    code
 3: Perform loop unrolling in the flattened code
 4: Split mathematical expressions with multiple operands
    into atomic operations
 5: Assign each atomic expression a unique SSA-variable ID
 6: for each variable in the SSA code, do do
 7:     Extract computational cost
 8:     Assign predecessors nodes
 9: end for
10: Build SSA-SDF adjacency matrix
11: Output ← SSA-SDF Graph
```

The 'code-to-SDF' tool provides a preliminary estimation for SSA-SDF cycles count since each node includes the computation cost for a software implementation. The tool also allows converting the SSA Matlab source code into a C code for performance evaluation using a specific processor. In this work, we choose the Nios-II as the target processor.

The Nios-II processor has two configurations: (1) The Nios-II/f, which includes specific hardware multiply and divide units. (2) The Nios-II/e is a more economical version of the Nios, characterized by low power and area but with limited computational power. The Nios-II/e has 1287 logic cells versus 2618 for the Nios-II/f and therefore has a lower power consumption, 5.91mW versus 15.92mW, respectively. Since the Nios-II/e emulates the multiply and divide operations, it is difficult to estimate the cycle count accurately [34].

For each processor configuration, we evaluate two memory architecture: (a) MEM-I: where the data and program memory reside in external SDRAM, and (b) MEM-II using Tightly Coupled Memory (TCM) for instruction and external SDRAM for data. Each processor configuration and memory architecture has a different computational model, which affects the latency. For example, the Nios-II/f with MEM-I requires ten cycles for an RD operation, three cycles for a WR operation, one cycle for MUL and AD, and 35 cycles for DIV.

The proposed methodology is demonstrated using a blur algorithm for a separable 3×3 box filter with $\mathcal{O}(Nr)$ complexity. Table 1 shows the evaluated runtimes of the original separable filter as described in Alg. 2 for both Nios-II/e and Nios-II/f. The best result is achieved with the Nios-II/f using TCM. A first-order software optimization has been applied using software GCC compiler with -Os flag for code size optimization for both Nios processors. This software implementation is used as the baseline reference for further speedup evaluations in all the following stages.

Algorithm 2 Separable Two 1D Pass Box Blur

```
1:  $I \leftarrow Input\ Image$
2:  $r \leftarrow Radius\ of\ the\ filter$
3:  $result \leftarrow zeros(N_{height} - r, N_{width} - r)$
4:  $ver \leftarrow zeros(N_{height} - r - 1, N_{width})$
5:  $hor \leftarrow zeros(N_{height} - r, N_{width} - r)$
6:  for $i = 1 : N_{height}$ do
7:      for $j = r + 1 : N_{width} - r$ do
8:          $sum_{ver} \leftarrow 0$
9:          for $k = -r : r$ do
10:             $sum_{ver} \leftarrow I[i, j + k]$
11:         end for
12:         $ver[i, j - r] \leftarrow sum_{ver}/(2r + 1)$
13:     end for
14: end for
15: for $i = r + 1 : N_{height} - r$ do
16:     for $j = 1 : N_{width} - 2r$ do
17:         $sum_{hor} \leftarrow 0$
18:         for $k = -r : r$ do
19:             $sum_{hor} \leftarrow ver[i + k, j]$
20:         end for
21:         $hor[i - r, j] \leftarrow sum_{hor}/(2r + 1)$
22:     end for
23: end for
```

TABLE 1. Nios-II f/e run cycles for a 3×3 box blur algorithm apply on different image sizes.

	Nios-II/f		Nios-II/e	
	MEM-I	MEM-II	MEM-I	MEM-II
5x5	11554	3273	38194	7339
16x16	166368	53513	789290	137567
32x32	726123	235777	3588347	634421
64x64	3019955	988410	15152061	2663528

The 'code-to-SDF' is used to convert a Matlab source code to SSA-SDF representation. Listing 1-3 show an example for the blur algorithm (Alg. 2). Listing 1 describes the Matlab source, while Listing 2 depicts an intermediate naïve representation code (basic convolution operations), and Listing 3 shows the final SSA conversion for each iteration. Finally, the SSA-SDF graph is converted into a compatible common C code.

Fig. 6 depicts SSA-SDF representation for a 3×3 blur algorithm applied to a 5×5 image. The input image is reduced to 5×5 pixels for visibility purposes. The arithmetic operations are represented by the blue circles, while red triangles are input/output matrices.

Table 2 shows a comparison of the estimated cycles count, derived from the SSA-SDF graph using the converted C code running on a Nios-II/f, for various image pixels size. A fixed speedup factor of about three is achieved by the proposed SSA-SDF algorithm, which is actually realized by loop unrolling techniques. For image size of 16×16 pixels and above, a fairly accurate estimation error of about 3% is achieved. The high error rate of a 5×5 pixels image size can be explained by the CPU initialization overhead.

Table 3 shows the code-to-SDF conversion runtime and the resulted code size. The runtime shows that the SSA-SDF conversion is performed in a reasonable time. The code size is larger the 32KB for images of 32×32 pixels and above and therefore can not be implemented using TCM (MEM-II architecture).

Comparison of the cycle count between the original C code (Table 1) and SSA C code (Table 2) show the SSA-SDF provide a speedup (up to $\times 3$) in terms of cycle counts for the Nios-II/f. This can be explained by the 'for' loop (Alg. 2), which involves a branch operation in each iteration.

Since the MEM-II architecture (using TCM) is the preferred choice for IoT devices (in terms of execution speed and cost) the code size limitation should be addressed. Therefore, we suggest using a clustering approach and unique implementation model to face code size limitations (as described in the following subsections).

TABLE 2. SSA-SDF cycles count for a 3×3 box blur algorithm applies to different image sizes.

	Nios-II/f	Tool est.	Est. Error	Speedup
MEM-I				
5x5	3053	3624	15.76%	x3.78
16x16	63749	65940	3.32%	x2.60
32x32	286905	280860	1.75%	x2.53
64x64	1201075	1181257	1.65%	x2.51
MEM-II				
5x5	1527	1754	14.87%	x2.14
16x16	35023	35989	2.76%	x1.52

C. SUB-GRAPH CLUSTERING

To find the optimal sub-graph clusters for partitioning the SSA-SDF graph into appropriate hardware and software resources, we propose a unique enumeration algorithm.

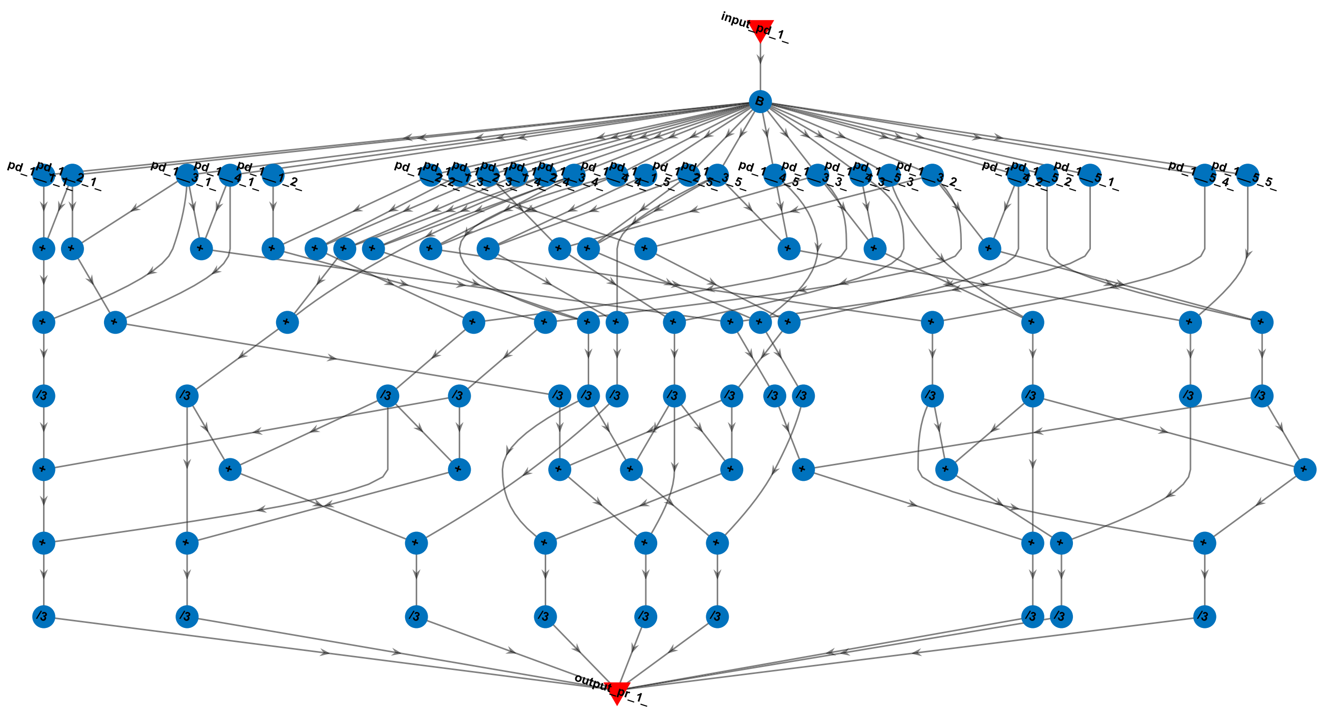

FIGURE 6. SSA-SDF representation for the blur algorithm (blue circles are arithmetic operations while red triangles are input/output).

```
input_pd_1_=rand(5);
f=[1/3;1/3;1/3];
output_pr_0_=conv2(input_pd_1_,f,'valid');
output_pr_1_=conv2(output_pr_0_,f','valid');
```

Listing 1. Matlab source code for a 3 × 3 box blur algorithm.

```
for i=1:size(input_pd_1_,1)-2
 for j=1:size(input_pd_1_,2)
  for k=1:size(f,1)
   if (k==1) add1=input_pd_1_(i,j);
   else add1=add1+input_pd_1_(i+k-1,j);end;end
  div1(i,j)=add1/3;end;end
output_pr_0_ = div1;
```

Listing 2. Intermediate naive representation code for convolution.

```
add1_1_1_1_ = input_pd_1__1_1_;
add1_1_1_2_ = add1_1_1_1_+input_pd_1__1_2_;
add1_1_1_3_ = add1_1_1_2_+input_pd_1__1_3_;
div1_1_1_   = add1_1_1_3_/3;
add1_1_2_1_ = input_pd_1__1_2_;
add1_1_2_2_ = add1_1_2_1_+input_pd_1__1_3_;
add1_1_2_3_ = add1_1_2_2_+input_pd_1__1_4_;
div1_1_2_   = add1_1_2_3_/3;
```

Listing 3. SSA conversion for each loop iteration.

The proposed algorithm identifies common convex sub-graphs in the SSA-SDF graph Γ, also known as induced sub-graphs $\Gamma[S]$. Then each sub-graph is replaced with a single "super-node", integrating several arithmetic operations into a single atomic operation. This process generates a higher granularity clustered graph Γ'. Each "super-node" should satisfy a predefined set of constraints $\vec{u}$, in order to be chosen as a suitable candidate for hardware acceleration.

TABLE 3. Code-to-sdf parameters for a 3 × 3 box blur algorithm apply to different image sizes.

	Runtime(s)	Code-Size(B)
5x5	2.82	8500
16x16	7.47	17928
32x32	15.22	52532
64x64	55.02	195396

The constraint set $\vec{u}$ for each sub-graph $\Gamma[s_n]$ include: (a) the maximum number of input operands, (b) the maximum number of output operands, (c) maximal length of the critical path, (d) the maximum number of nodes (e) the minimum number of the sub-graph occurrences.

We propose a unique graph search with two separated sequential passes on the graph: (a) clustering pass: the first pass searches only for parent-child pairs in purpose to extract sub-graph with a single output and (b) grouping pass: the second pass search for all node pairs (including parent-child) which have common inputs. This enables further acceleration by sharing common input operations.

The proposed clustering algorithm is based on the hill-climbing heuristic search [35]. Hill-climbing is a mathematical optimization technique that belongs to the local search family. It is an iterative algorithm that starts with an arbitrary solution and represents a target function $f(x)$, where x is a vector of continuous discrete values. At each iteration, hill climbing attempts to make an incremental change to the solution and determine whether the change improves the value of $f(x)$. This process continues until no more improvement is achieved in the target function.

The algorithm flowchart is depicted in Fig. 7. The following steps are performed on each stage of the clustering algorithm: (1) Extracting a representing list of all parent-child nodes in the SSA-SDF graph Γ. (2) The parent-child list is divided into several sets according to their specific mathematical operation. For example, all pairs where parent nodes represent 'multiplication' and the child nodes represent 'summation' are grouped into the same set (as shown in Fig. 8.a). All previous sets are excluded. (3) Then, the most popular arithmetic operation, identically the set which includes the maximum number of parent-child pairs, is selected as a sub-graph candidate. (4) The elements in the selected set are arranged by order according to their minimal distance from either a selected input or from the nearest input using breadth-first search [36]. (5) Then, the first element in the set is replaced with a single "super-node" creating graph Γ' and is discarded from the set. (6) The examined "super-node" should comply with the constraint set $\vec{u}$ in case the constraints are satisfied, the "super-node" is compared against all previously induced sub-graphs $\Gamma[s_n]$. Then it is defined as a new sub-graph $\Gamma[s_{n+1}]$ in case there is no identical already created "super-node" representing the same atomic operation. The comparison involves graph isomorphism search by means of commutative, associative, and distributive properties of the parent-child nodes. (7) Then, any repetitive elements caused by removing the last parent-child pair are discarded from the set. The left pair operation (multiplication and add) of Fig. 8.a is replaced with a single "super-node" operation (AB+), as shown in Fig. 8.b, therefore discarding the left pair operation in Fig. 8.a. (8). This process is iteratively repeated for all the remaining elements in the set. (9) Finally, the Γ graph is replaced with a temporary Γ' graph and $\Gamma[s_n]$ are updated accordingly. (10) This flow is repeated until all pairs in the graph are marked as excluded.

A similar flow is applied to the resulted SDF graph generated by the clustering pass. The two orange eclipses in Fig. 7 describe the two different phases between both passes. The grouping clustering is applied to all node pairs, while the clustering pass considers only parent-child pairs. In addition, the set in the grouping algorithm is organized by order using the number of common inputs as an order criterion.

The purpose of the clustering pass is to search for all node pairs (including parent-child) that have common inputs to share joint input operations. Fig. 9 demonstrates the grouping algorithm applied to the sub-graph shown in Fig. 8.c. Fig. 9.a shows two "super-nodes" where each one represents: four inputs, one output, and three atomic operations. We can notice that the inputs A/B/C are common to the two "super-nodes". The grouping algorithm results in a new "super-node", which includes five inputs, two outputs, and five atomic operations.

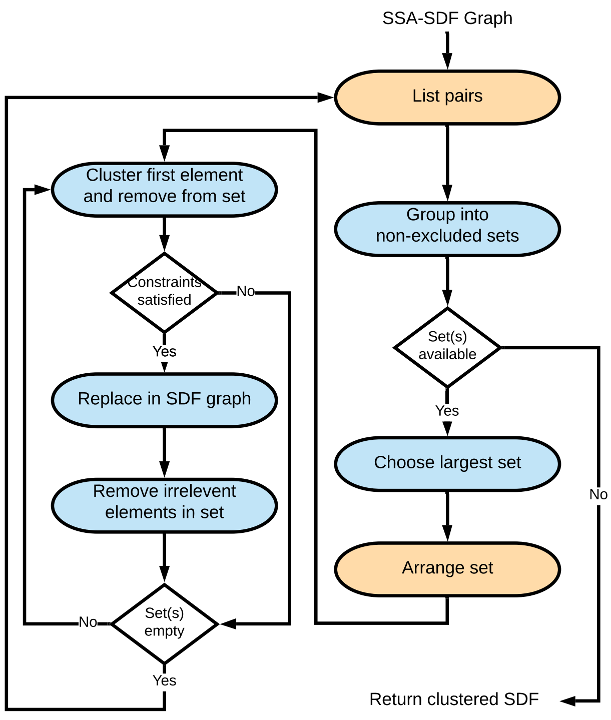

FIGURE 7. Clustering algorithm flow.

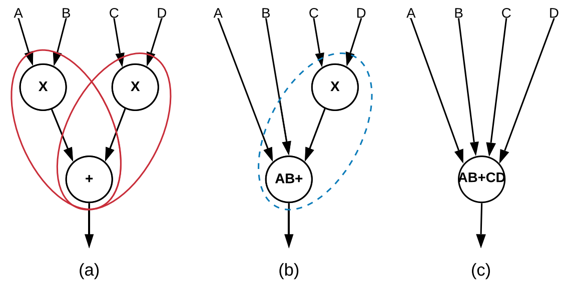

FIGURE 8. A clustering algorithm example: (a) MUL-ADD pair operation. (b) A single "super-node" operation (AB+). (c) Final "super-node" operation (AB+CB).

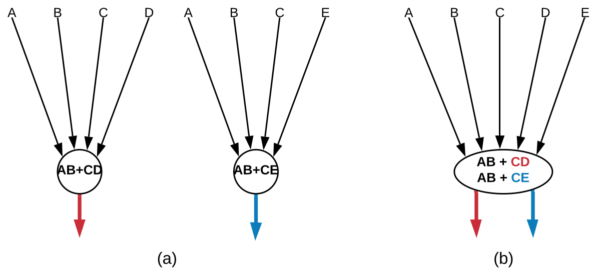

FIGURE 9. A cluster grouping example: (a) Two "super-nodes" each represents 3 atomic operations. (b) Final "super-node" represents 5 atomic operations.

Since $A \times B$ is common to both "super-nodes" the new integrated "super-node" required only five operations (instead of the original six operations). Moreover, since A/B/C are common inputs, less memory access is required, and simple fused custom instruction can be implemented.

Fig. 10 depicts the combined clustering-grouping SDF for the blur filter SSA-SDF shown in Fig. 6. This example supports up to eight inputs and four outputs defined by the $\vec{u}$ constraint vector and complies with 32-bit ISA. The clustering algorithm extracts nine different types of induced sub-graphs that satisfy the constraint. A possible solution using four induced sub-graphs is presented in Fig. 10. The statistic of the extracted sub-graphs is presented in Fig. 11, demonstrating the sub-graphs occurrences in the final SDF graph. Fig. 12 depicts the four sub-graphs, representing the six "super-nodes" (atomic operations) shown in Fig. 10 (yellow triangles). It can be seen from Fig. 11 that the sub-graph "SG Module #8" is used three times. Fig. 13 depicts three atomic sub-graphs which are used as building blocks in the sub-graphs shown in Fig. 12.

Table 4 shows the clustering results for the 3×3 blur filter for different image sizes. This table demonstrates the extracted nodes in each phase of the proposed clustering and grouping process. For example, for the 5×5 image, the original SSA-SDF graph includes 72 nodes (In - the first column). The clustering algorithm (pass one) extracts 18 nodes (Out - second column). Two possible sub-graphs selection based on statistical analysis are described in columns three and four. The top-1 case represents a choice of only one sub-graph with 12 occurrences for the 5×5 image, while the top-3 case represents a choice of three sub-graphs containing all 18 nodes. A similar grouping process is carried out in pass two, starting with the 18 nodes extracted in pass one. In some cases, more sub-graphs should be required in order to cover all the nodes in the graph. Fig. 14 depicts the clustering runtime as a function of the numbers of the nodes in the SDF graph. As expected, the results demonstrate a linear time complexity that matches the time complexity of previous graph-based acceleration methods [8], [16], [18].

TABLE 4. Clustering results for the blur algorithm.

	Pass	In	Out	Top-1	Top-3	Time(s)
5x5	(1)	72	18	12	18	
	(2)	18	6	3	5	1.65
16x16	(1)	1260	392	364	392	
	(2)	392	106	42	98	6.14
32x32	(1)	5580	1800	1740	1800	
	(2)	1800	466	210	450	26.52
64x64	(1)	23436	7688	7564	7688	
	(2)	7688	1954	930	1922	114.81

D. GRAPH SCHEDULING

Following the clustering algorithm, we propose to apply the graph scheduling algorithm to the resulting clustering SDF graph. The scheduling algorithm is aimed to find the optimal sub-graph (which was extracted in the previous stage) combinations Γ' which lead to maximum performance or minimum area while facing the given constraints in terms of PPA. While choosing more complicated sub-graphs, which include more atomic operations, might improve the performance, selecting several atomic sub-graphs may result in better area utilization.

In this work, Heterogeneous Earliest Finish Time (HEFT) algorithm [37] has been chosen as the basis scheduling process. The HEFT algorithm is a heuristic list-based scheduling algorithm that provides a simple schedule plan with a low computational complexity of $O(n^2 \cdot p)$, where n and p are the number of tasks and resources, respectively. The HEFT algorithm consists of two phases. In the first phase the tasks are ranked according to their execution priority. The task priority is determined as a function of the task weight and the inter task communication cost, and the search path in the SDF. The second phase is responsible to assign the processor's available resources to each task.

The original HEFT is applied to task scheduling in a multiprocessor environment. We suggest modifying the HEFT algorithm to support the optimal allocation of the extracted multiple sub-graphs to several co-processors. The proposed SSA-HEFT algorithm provides scheduling support for implementing multiple custom accelerators in a single-core environment sharing the same memory. Each induced sub-graphs $\Gamma[s_n]$ is considered a custom instruction that is accelerated using a unique co-processor. The scheduling algorithm support both blocking and non-blocking multi-cycle custom instruction. The number of required cycles is determined by the maximum length of the critical sub-graph path and the type of the atomic operations.

The basic configuration of the SSA-HEFT scheduler is defined using a single processor and several co-processors (a single instance for each sub-graph). The proposed scheduler should consider the following architectural information: the processor instructions execution time (in clock cycles), memory latency, and register file size. The size of the register file is used as a parameter while implementing the scheduler to adjust the execution time to a real-time software implementation. Fig. 15 depicts the clustered blur filter SSA-SDF scheduler results shown in Fig. 10.

Table 5 and Table 6 show the speedup contribution due to the clustering algorithm. Table 5 demonstrates the proposed hardware/software partitioning implementation's achieved speedup compared to the reference software implementation using Nios-II/f on the FPGA board. A speedup factor of about six is demonstrated for all the image sizes. This table also shows the estimation of the execution time as been extracted by our SSA-HEFT algorithm. It can be seen that the estimation error is lightly accurate and reasonable.

For example, a software implementation with no acceleration requires 3273 cycles (for the 5×5 images), while only 542 cycles are needed for implementing the custom instruction accelerated core. Table 6 shows similar results for the Nios-II/e running on the same board. In this case, the proposed approach demonstrates a speedup factor of about 15. As expected, the code size required for the software implementation is significantly reduced compared to Table 3.

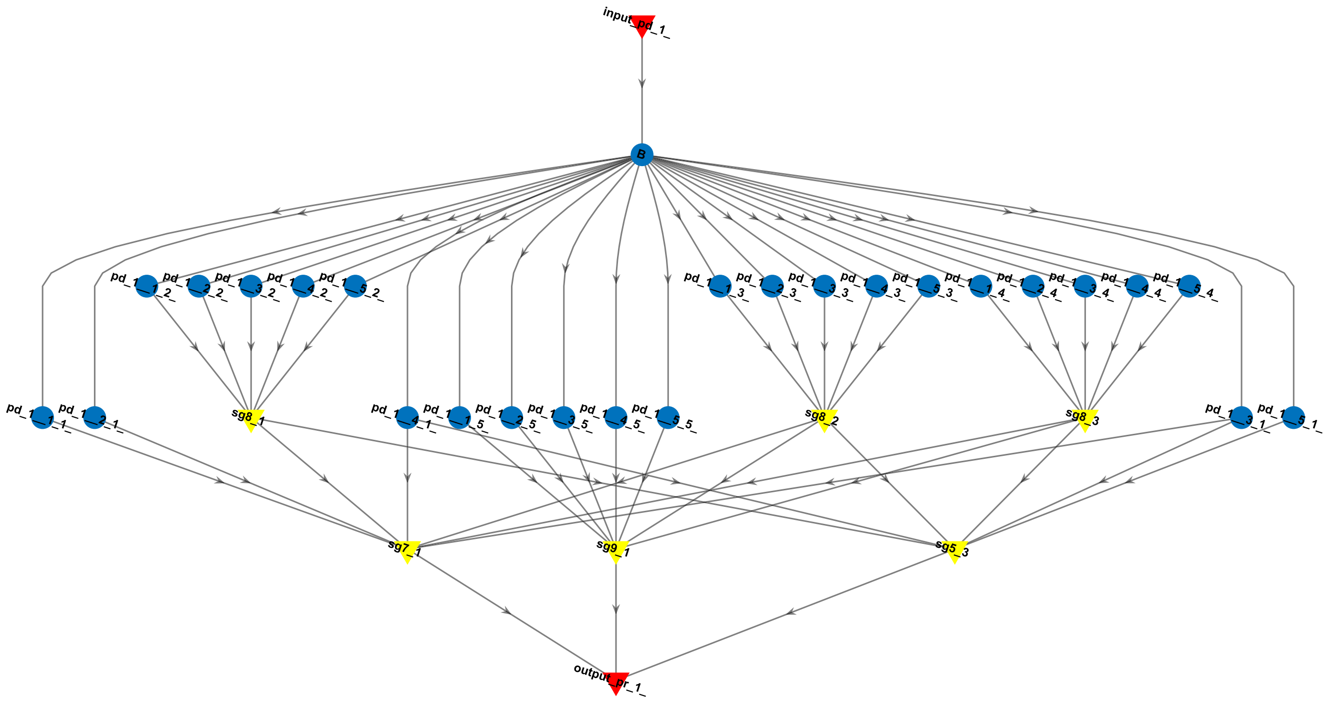

FIGURE 10. Combined clustering and grouping for the blur filter (5 × 5 image).

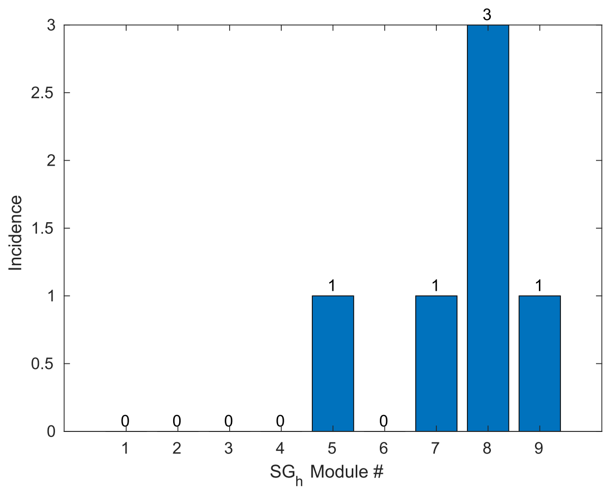

FIGURE 11. Induced sub-graphs Γ[*S*] statistics for a separable filter box blur filter on 5 × 5 image. [squeeze height].

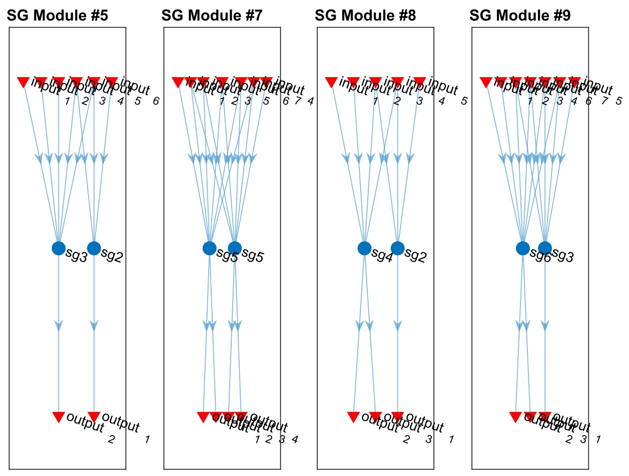

FIGURE 12. Clustering-Grouping pass: Induced sub-graphs Γ[*S*] of a separable filter box blur filter on 5 × 5 image. [squeeze height].

TABLE 5. Nios-II/f MEM-II scheduling for a 3 × 3 box blur algorithm applies to different image sizes.

	SW	SW /HW	Speedup	Est.	Est. Error
5x5	3273	542	x6.03	599	10.52%
16x16	53513	9022	x5.93	8582	4.88%
32x32	235777	41038	x5.74	38510	6.16%

TABLE 6. Nios-II/e MEM-II Scheduling for a 3 × 3 box blur algorithm applies to different image sizes.

	SW	SW/HW	Speedup	Code-Size(B)
5x5	7339	550	x13.34	9024
16x16	137567	9018	x15.25	14300
32x32	634421	41023	x15.46	33096

E. IMPLEMENTATION MODEL

The proposed hardware/software implementation model is composed of the following two main components: (1) a software package (written in C) that implements the SSA-HEFT scheduler, (2) a Custom Instruction Wrapper (CIW), which includes the hardware accelerators.

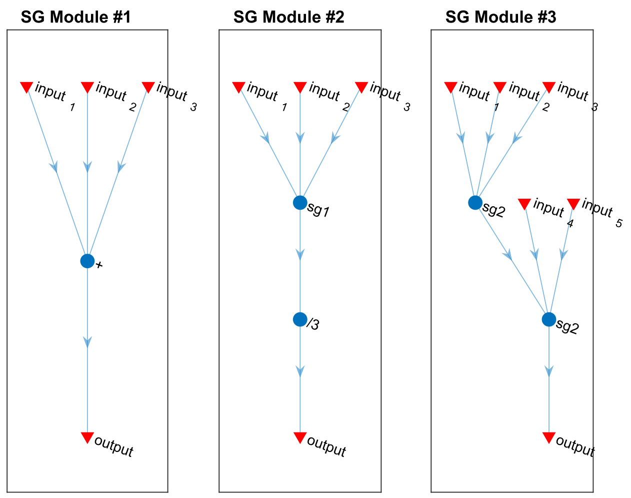

FIGURE 13. Singleton sub-graphs Γ[*S*] for a separable filter box blur filter on a 5 × 5 image.

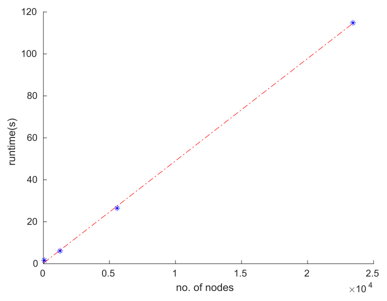

FIGURE 14. Clustering runtime complexity.

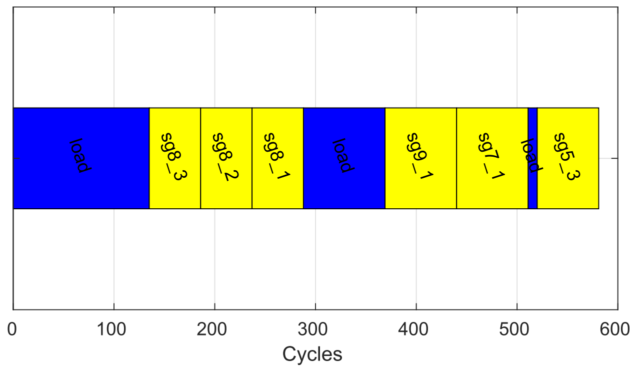

FIGURE 15. SSA-HEFT scheduler results for a separable filter box blur filter on 5 × 5 image.

The software scheduler has a hardware interface to communicate with the custom instruction module. The scheduler is responsible for both software and hardware task scheduling. While a task is a candidate for acceleration using hardware custom instruction, the following parameters should be transferred to the CIW: the type of the required operation (i.e., an index for a specific sub-graph), and pointers to the operands for this operations.

We propose using a single extended multi-cycle instruction to utilize the scenarios of repeated sub-graphs within the SSA-HEFT. A single custom instruction replaces consecutive repeated calling to the same sub-graph. For example, sub-graph SG8 has three instances (SG8_1, SG8_2, SG8_3) as depicted in Fig. 15. Therefore, it requires three different access to the hardware accelerator. Alternatively, we suggest using only a single call to the custom instruction wrapper. This implementation model significantly improves latency and code size performance, reducing the number of required interactions with the hardware accelerator. The proposed scheduler implementation has been further enhanced by adding the real-time hardware decompression technique [38], [39].

Fig. 16 depicts the Custom Instruction Wrapper (CIW). The CIW comprises of four main components: a Custom Instruction Controller (CIC), hardware accelerator unit, hardware scheduler control module, and data-path control module.

The CIC is responsible for the communication with the processor and the configurations of other CIW modules. The hardware accelerator unit implements the sub-graphs (a unique custom instruction per sub-graph). The hardware scheduler stores the operands required to complete the operation indexes. The data-path control module uses both software pointer and indexes to fetch the appropriate operands and store the results (using DMA controller). The module includes an internal memory to enable reuse of already fetched data (such as DNN weights).

The HW-Scheduler control is responsible for fetching an encoded data file for each sub-graph. The data file includes the specific sub-graphs IDs and indexes for various input operands arrays. The HW-Scheduler can manage several input operand arrays, each representing different data types, such as weights and data for DNN applications.

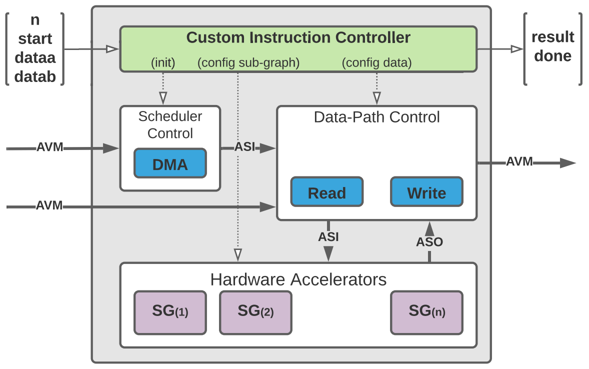

FIGURE 16. Custom Instruction Wrapper (CIW).

Table 7 shows the speedup results for the Nios-II/e using TCM for a 3 × 3 box blur algorithm. The results are compared to only software implementation for the blur algorithm. A significant speedup factor of up to 122 is demonstrated for the 64 × 64 images. The speedup increases as the image size increases. Although the C code for the software scheduler remains about the same for all images, the size of the data file required by the HW-Scheduler increases as the image size increases.

TABLE 7. Speedup results for Nios-II/e using TCM for a 3 × 3 box blur algorithm.

	SW	SW/ HW	SpeedUp	Code Size	Scheduler Size
5x5	7339	268	x27.38	8740	14
16x16	137567	3260	x42.19	8784	494
32x32	634421	7466	x84.97	8852	2254
64x64	2663528	21727	x122.59	8980	9567

V. EXPERIMENTAL AND RESULTS

The proposed methodology has been evaluated employing hardware accelerators for various Neural Network (NN) architectures. The proposed accelerator is compared to a software-only implementation using the open-source TensorFlow Lite for Micro-controllers (TFLM) [40]. The MLPerf Tiny benchmark [41], [42] is used for runtime and code-size comparison.

TABLE 8. Anomaly detection clustering results: Vertical (V) and Horizontal (H).

Pass		No. of Nodes	No. of SG Types	Sum of SGs Multipliers	Estimated Cycles
V	H				
-	-	798880	0	-	
1	-	262520	1	1	5007015
2	-	132096	2	3	4089423
3	-	67720	3	7	3374136
4	-	36368	4	15	2124189
5	-	34696	3	15	2040680
6	-	33024	2	15	1889251
6	1	33024	2	15	1889251

This benchmark consists of three sequential models for machine learning tasks: (a) Keyword Spotting (KWS), which uses a neural network that detects keywords from an audio spectrogram, (b) Visual Wake Words (VWW), a binary image classification task for determining the presence of a person in an image, and (c) Anomaly Detection (AD), which uses a neural network to identify abnormalities in machine operating sounds. To further evaluate the proposed methodology, we examined the common TFLM model for (a) Google network for 'Gesture Recognition Magic Wand' (GRMW) that was trained to detect wand gestures [43], and (b) an MNIST network used for Handwritten Digit Recognition (HDR) [44]. The speedup reported in this section is always with respect to the baseline software implementation as defined in Sec. IV.

The Nios-II embedded processor has been implemented on the Intel Cyclone-10 LP FPGA with the following configuration: 120MHz reference clock, 32 KBit on-chip RAM, and an external 8 MByte SDRAM. The RTL implementation of the Custom Instruction Wrapper (CIW) results in 627 logic elements. Each sub-graph includes additional dedicated logic according to the implemented function. The TFLM model is converted to an equivalent Matlab code, and the network's input data files are represented in HDF5 or JSON format.

The implementation of the sub-graph clustering (which has been performed on the SSA-SDF graph) is limited by the Cyclone-10 FPGA resources since the available number of the multiplier elements is limited to 132. The clustering phase results in various sub-graph configurations from which we choose two hardware implementations applied to different scenarios: (a) Low-Resource implementation and (b) High-Resource implementation. For the low-resource implementation, we search for sub-graph configuration that consists of no more than 15% of the total device multiplier elements, while for the high-resource implementation, the sub-graph configurations consist in the range of 15-85% of the device multiplier elements. Running the MLPerf Tiny benchmark on the proposed Cyclone10 LP-based implementation results in extra power consumption of up to 80mw.

Table 8 shows the clustering results for the low-resource implementation (with less than 20 multipliers) applied to the anomaly detection model [42]. The clustering process converges to two SG types, and therefore only two hardware accelerators are required for efficient implementation. Moreover, the number of nodes is reduced from about 800k elements in the original graph to only 30K elements, resulting in fewer cycles.

Table 9 demonstrates the overall hardware accelerator speedup compared to the software model for the two different hardware implementations of each benchmark model. For example, the resulting runtime for the anomaly detection model, running on the Nios-II/f, is 108ms, 17ms, and 8ms for the TFLM software implementation, the low-resource, and the high-resource hardware-accelerated implementation, respectively. The Nios-II/e results with 5.67s, 32ms, and 11ms for the software implementation, the low-resource, and the high-resource hardware-accelerated implementation, respectively. A speedup factor of 515 and 177 compared to software implementation is achieved for the high and low resource implementations, respectively. As expected, the high-resource implementation outperforms the low-resource implementation. The Nios-II/e demonstrates a speedup factor from 515 (AD) up to 3936 (GRMW) for the high-resource implementation, and a speedup factor from 177 (AD) up to 696 (HDR) for the low-resource implementation.

The table shows that while the Arm Cortex-M4 is characterized with 1.95 DMIPS/MHz, both Nios-II/f and Nios-II/e can perform only 0.75 and 0.1 DMIPS/MHz, respectively [45], [46]. However, we demonstrate that adding the proposed hardware accelerator to the low-performance Nios-II makes it a competitive choice for the Arm processor.

TABLE 9. Hardware accelerator speedup for the MLPerf tiny benchmark.

Model	**FLOPS**	**ARM**	**Nios**	**TFLM**	**Low-Resource**		**High-Resource**	
				latency	latency	speedup	latency	speedup
KWS	5393608	181ms [1]	(f)	12.92s	369ms	x35	161ms	x80
			(e)	206.23s	458ms	x450	181ms	x1139
VWW	14318102	603ms [1]	(f)	36.06s	1087ms	x33	479ms	x75
			(e)	546.94s	1347ms	x408	583ms	x943
AD	530056	10ms [1]	(f)	108.06ms	17ms	x6.5	8ms	x14
			(e)	5.67s	32ms	x177	11ms	x515
GRMW	131512	1ms [2]	(f)	333ms	3.3ms	x101	1.7ms	x195
			(e)	4.92s	9ms	x546	1.25ms	x3936
HDR	5646588	46ms [2]	(f)	13.92s	140ms	x100	91ms	x151
			(e)	227.56s	326ms	x696	130ms	x1750

[1] Arm-M4. [2] Arm-A53.

VI. CONCLUSION

This work suggests an efficient and automated codesign partitioning methodology to improve performance while keeping low power and area. Identifying the potential for hardware acceleration and resource utilization at an early development stage is essential information for the designer and may significantly affect design decisions. The proposed methodology is based on synchronous data graph analysis to detect common patterns as candidates for hardware acceleration.

We suggest a unique framework to analyze a given high-level source code extracting a set of hardware accelerators. The hardware accelerators are implemented using a custom micro-architecture targeting limited-resources processors.

A unique clustering and scheduling algorithm has been developed to obtain real-time constraints by selecting an optimized sub-set of accelerators. The proposed codesign methodology has been evaluated for some neural networks (NNs) architectures using the common MLPerf Tiny benchmark. The proposed accelerators are compared to a software-only implementation using the open-source TensorFlow Lite for Micro-controllers (TFLM). Experimental results demonstrate a significant speedup of up to 3 orders of magnitude compared to software-only implementation.

REFERENCES

[1] T. Adegbija, A. Rogacs, C. Patel, and A. Gordon-Ross, "Microprocessor optimizations for the Internet of Things: A survey," *IEEE Trans. CAD Integr. Circuits Syst.*, vol. 37, no. 1, pp. 7–20, May 2018.

[2] (Mar. 2020). Cadence. *Tensilica Customizable Processors*. [Online]. Available: https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable

[3] (Jun. 2020). Synopsys. *Arc EM Processor Family*. [Online]. Available: https://www.synopsys.com/designware-ip/processor-solutions/arc-em-family.htmll

[4] (Mar. 2020). Intel. *Nios II Processors*. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/processor% /nios-ii.html

[5] (Mar. 2020). Xilinx. *Microblaze Soft Processor Core*. [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html

[6] W.-S. Gan and S. M. Kuo, "Teaching DSP software development: From design to fixed-point implementations," *IEEE Trans. Educ.*, vol. 49, no. 1, pp. 122–131, Feb. 2006.

[7] B. A. Syrowik, B. Fort, and S. D. Brown, "Use of CPU performance counters for accelerator selection in HLS-generated CPU-accelerator systems," in *Proc. 9th Int. Symp. Highly-Efficient Accel. Reconfigurable Technol.*, New York, NY, USA, Jun. 2018, pp. 1–6.

[8] C. Xiao, E. Casseau, S. Wang, and W. Liu, "Automatic custom instruction identification for application-specific instruction set processors," *Microprocessors Microsyst.*, vol. 38, no. 8, pp. 1012–1024, Nov. 2014.

[9] P. Arató, S. Jáhasz, Z. Á. Mann, A. Orban, and D. Papp, "Hardware-software partitioning in embedded system design," in *Proc. IEEE Int. Symp. Intell. Signal Process.*, Sep. 2003, pp. 197–202.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[11] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis From Dataflow Graphs*. Norwell, MA, USA: Kluwer, 1996.

[12] M. Edwards and P. Green, "The implementation of synchronous dataflow graphs using reconfigurable hardware," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, R. W. Hartenstein and H. Grünbacher, Eds. Berlin, Germany: Springer, 2000, pp. 739–748.

[13] (Jan. 2019). CEVA. *CEVA BX Product Note*. [Online]. Available: https://www.ceva-dsp.com/wp-content/uploads/2019/01/CEVA_BX_Brochure_EN%_final_secure.pdf

[14] (Apr. 2020). Intel. *Nios II Custom Instruction User Guide*. [Online]. Available: https://www.intel.cn/content/dam/altera-www/global/en_US/pdfs/literatur% e/ug/ug_nios2_custom_instruction.pdf

[15] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. J. M. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys, "Gecos: A framework for prototyping custom hardware design flows," in *Proc. Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2013, pp. 100–105.

[16] S. Wang, C. Xiao, and W. Liu, "A faster algorithm for enumerating connected convex subgraphs in acyclic digraphs," *IEEE Embedded Syst. Lett.*, vol. 9, no. 1, pp. 9–12, Mar. 2017.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[18] C. Xiao, S. Wang, W. Liu, X. Wang, and E. Casseau, "An optimal algorithm for enumerating connected convex subgraphs in acyclic digraphs," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 1, pp. 261–265, Jan. 2021.

[19] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, and L. Pozzi, "RegionSeeker: Automatically identifying and selecting accelerators from application source code," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 741–754, Apr. 2019.

[20] G. Zacharopoulos, L. Ferretti, G. Ansaloni, G. Di Guglielmo, L. Carloni, and L. Pozzi, "Compiler-assisted selection of hardware acceleration candidates from application source code," in *Proc. IEEE 37th Int. Conf. Comput. Design (ICCD)*, Nov. 2019, pp. 129–137.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," Computer Science Dept., Univ. Illinois Urbana-Champaign, Champaign, IL, USA, Tech. Report UIUCDCS-R-2003-2380, Sep. 2003.

[22] D. Wijesundera, A. Prakash, T. Perera, K. Herath, and T. Srikanthan, "Wibheda+: Framework for data dependency-aware multi-constrained hardware-software partitioning in FPGA-based SoCs for IoT applications," in *Proc. 9th Int. Symp. Highly-Efficient Accel. Reconfigurable Technol.*, New York, NY, USA, Jun. 2018, pp. 1–6.

[23] W. Zuo, L.-N. Pouchet, A. Ayupov, T. Kim, C.-W. Lin, S. Shiraishi, and D. Chen, ''Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration,'' in *Proc. 54th Annu. Design Automat. Conf.*, Jun. 2017, pp. 1–6.

[24] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, ''An analysis of accelerator coupling in heterogeneous architectures,'' in *Proc. 52nd Annu. Design Automat. Conf.*, Jun. 2015, pp. 1–6.

[25] K. Atasu, L. Pozzi, and P. Ienne, ''Automatic application-specific instruction-set extensions under microarchitectural constraints,'' in *Proc. 40th Conf. Design Automat. (DAC)*, 2003, pp. 256–261.

[26] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, ''Constraint-driven instructions selection and application scheduling in the DURASE system,'' in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst., Architectures Processors*, Jul. 2009, pp. 145–152.

[27] J. R. Hauser and J. Wawrzynek, ''Garp: A MIPS processor with a reconfigurable coprocessor,'' in *Proc. 5th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 1997, pp. 12–21.

[28] N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, ''The arise approach for extending embedded processors with arbitrary hardware accelerators,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 2, pp. 221–233, Feb. 2009.

[29] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, ''A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors,'' *IEEE Trans. Comput.-Aided Design Integr.*, vol. 26, no. 11, pp. 2035–2045, Nov. 2007.

[30] J. Cong and K. Gururaj, ''Architecture support for custom instructions with memory operations,'' in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, New York, NY, USA, 2013, pp. 231–234.

[31] A. Prakash, C. T. Clarke, S.-K. Lam, and T. Srikanthan, ''Rapid memory-aware selection of hardware accelerators in programmable SoC design,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 445–456, Mar. 2018.

[32] E. Manor, A. Ben-David, and S. Greenberg, ''CORDIC hardware acceleration using DMA-based ISA extension,'' *J. Low Power Electron. Appl.*, vol. 12, no. 1, p. 4, Jan. 2022.

[33] E. Manor and S. Greenberg, ''Efficient Hardware/Software partitioning for heterogeneous embedded systems,'' in *Proc. IEEE Int. Conf. Sci. Electr. Eng. Isr. (ICSEE)*, Dec. 2018, pp. 1–4.

[34] A. Renbi, ''Data-stream-driven computers are power and energy efficient,'' in *Sustainable Practices*. Hershey, PA, USA: IGI Global, 2013, pp. 447–462.

[35] P. Stanicek and R. Farana, ''Chosen optimization methods for search data,'' in *Proc. 12th Int. Carpathian Control Conf. (ICCC)*, May 2011, pp. 370–373.

[36] S. Beamer, K. Asanovic, and D. Patterson, ''Direction-optimizing breadth-first search,'' in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–10.

[37] H. Topcuoglu, S. Hariri, and M.-Y. Wu, ''Task scheduling algorithms for heterogeneous processors,'' in *Proc. Heterogeneous Comput. Workshop*, Apr. 1999, pp. 3–14.

[38] T. Malach, S. Greenberg, and M. Haiut, ''Hardware-based real-time deep neural network lossless weights compression,'' *IEEE Access*, vol. 8, pp. 205051–205060, 2020.

[39] M. Ledwon, B. F. Cockburn, and J. Han, ''High-throughput FPGA-based hardware accelerators for deflate compression and decompression using high-level synthesis,'' *IEEE Access*, vol. 8, pp. 62207–62217, 2020.

[40] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, ''Tensorflow lite micro: Embedded machine learning on tinyml systems,'' 2020, *arXiv:2010.08678*.

[41] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. A. Patterson, D. Pau, J. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, ''Benchmarking tinyml systems: Challenges and direction,'' 2020, *arXiv:2003.04821*.

[42] C. Banbury, V. J. Reddi, P. Torelli, N. Jeffries, C. Kiraly, J. Holleman, P. Montino, D. Kanter, and P. Warden, ''MLPerf tiny benchmark,'' in *Proc. NIPS*, 2021, pp. 1–15.

[43] A. Williams. (Dec. 2018). *Magic Wand Learns Spells Through Machine Learning and an IMU*. [Online]. Available: https://hackaday.com/2018/12/07/magic-wand-learns-spells-through-machine-learning-and-an-imu/

[44] D. Tassopoulos. (Jul. 2019). *Machine Learning on Embedded*. [Online]. Available: https://www.stupid-projects.com/machine-learning-on-embedded-part-3/

[45] (Sep. 2020). Arm. *Arm Cortex-M4 Datasheet*. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4

[46] (Sep. 2020). Intel. *Nios II Performance Benchmarks*[Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literatur% e/ds/ds_nios2_perf.pdf

EREZ MANOR received the B.Sc. and M.Sc. degrees in electrical and computer engineering from the Ben-Gurion University of the Negev, Be'er Sheva, Israel, in 2008 and 2014, respectively. He is currently pursuing the Ph.D. degree. His primary research interests include AI, FPGA, and edge devices.

SHLOMO GREENBERG (Member, IEEE) received the B.Sc., M.Sc. (Hons.), and Ph.D. degrees in electrical and computer engineering from the Ben-Gurion University of the Negev, Be'er Sheva, Israel, in 1976, 1984, and 1997, respectively. He is currently an Associate Professor and the Head of the Computer Science Department, Sami Shamoon College of Engineering, and a Staff Member with the School of Electrical and Computer Engineering, Ben-Gurion University of the Negev. His main research interests include computer architecture, machine learning, image and digital signal processing, computer vision, and VLSI low-power design.