MISSIONE 4
ISTRUZIONE
RICERCA

# SPOKE 1

# FUTURE HPC & BIG DATA

# FLAGSHIP 2:
# Selection of candidate prototypes for software framework for acceleration Accel-SW-spec

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
ICSC Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

# EXECUTIVE SUMMARY

This document overviews and selects frameworks and tools for the acceleration of Deep Learning models for HPC systems, according to the main objectives described in the milestone #5, Spoke 1 - Flagship 2 WP2: Flagship on heterogeneous acceleration, architecture, tools, and software (Leader: POLIMI).

The field of deep learning has made significant progress in recent years, with many breakthroughs and advancements. As neural networks become more complex, there is an urgent need for efficient hardware accelerators. Creating these accelerators requires expertise from various fields, such as computer architecture, approximate computing, computational models, and deep learning algorithms. Moreover, existing architectures for accelerators in deep learning applications are highly heterogeneous. So, to achieve high performance and energy efficiency while minimizing power consumption and area metrics, designers have adopted various methodologies, including high-level synthesis methodologies, specific customized compilers, tools for design space exploration, modeling, profiling, partitioning, and mapping. These methods focus on maximizing parallelism and minimizing data movement to optimize accelerator design for deep learning applications. Understanding existing frameworks and tools is crucial for fostering innovation in this domain.

The document is structured according to different topics and sub-topics. Section 2 presents the design methodologies for the high-level synthesis of accelerators, while Section 3 discusses automated compilation and deployment technologies for deep learning-based applications. Section 4 introduces approaches and tools that have been proposed to distribute, partition and map deep-learning models on heterogeneous processing systems. Section 5 describes the modeling, simulation, profiling, and design exploration frameworks currently adopted for deep-learning-based applications. The last Section presents models of computations for HPC Deep Learning application workloads. Each of the sections in which this document has been organized covers existing notable and influential contributions and offers a perspective on each of the existing approaches with respect to the work that the research group will do in Flagship 2 Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing.

# Perspectives on Design Methodologies for Heterogeneous HPC Platforms for Deep Learning

FABRIZIO FERRANDI, SERENA CURZEL, LEANDRO FIORIN, DANIELE IELMINI, and CRISTINA SILVANO, Politecnico di Milano, Italy
FRANCESCO CONTI, ALESSIO BURRELLO, FRANCESCO BARCHI, and LUCA BENINI, Università di Bologna, Italy
LUCIANO LAVAGNO and TEODORO URSO, Politecnico di Torino, Italy
ENRICO CALORE, SEBASTIANO FABIO SCHIFANO, and CRISTIAN ZAMBELLI, Università degli Studi di Ferrara, Italy
MAURIZIO PALESI, GIUSEPPE ASCIA, and DAVIDE PATTI, Università degli Studi di Catania, Italy
STEFANIA PERRI, Università degli Studi della Calabria, Italy
NICOLA PETRA, DAVIDE DE CARO, and GENNARO DI MEO, Università degli Studi di Napoli Federico II, Italy
VALERIA CARDELLINI, SALVATORE FILIPPONE, and FRANCESCO LO PRESTI, Università degli Studi di Roma "Tor Vergata", Italy
FRANCESCO SILVESTRI, Università degli Studi di Padova, Italy
PAOLO PALAZZARI, ENEA, Italy

In recent years, the field of deep learning has seen significant advancements and breakthroughs. With the increasing complexity of deep neural networks, the need of efficient hardware accelerators has become more and more pressing. The design of such accelerators requires a multidisciplinary approach, combining expertise from computer architecture, approximate computing, computational models, and deep learning algorithms. Various methodologies have been adopted to design accelerators for deep learning, including high-level synthesis methodologies, specific customized compilers, tools for design space exploration, modeling, profiling, partitioning, and mapping. These methodologies aim to maximize parallelism and minimize data movement to achieve high performance and energy efficiency, while also reducing power consumption and area. This document represents a comprehensive survey that explores and evaluates the most notable approaches in the field and offers a perspective on each of the existing approaches with respect to the work done in Flagship 2 Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing.

## 1 INTRODUCTION

The field of deep learning has made significant progresses in recent years, with many breakthroughs and advancements. As artificial neural networks have become more complex, there is an urgent need of efficient hardware accelerators. Creating these accelerators requires expertise from various fields,

Authors' addresses: Fabrizio Ferrandi, fabrizio.ferrandi@polimi.it; Serena Curzel, serena.curzel@polimi.it; Leandro Fiorin, leandro.fiorin@polimi.it; Daniele Ielmini, daniele.ielmini@polimi.it; Cristina Silvano, cristina.silvano@polimi.it, Politecnico di Milano, Italy; Francesco Conti, f.conti@unibo.it; Alessio Burrello, alessio.burrello@unibo.it; Francesco Barchi, francesco.barchi@unibo.it; Luca Benini, luca.benini@unibo.it, Università di Bologna, Viale Carlo Pepoli, 3/2, 40123, Bologna, Italy; Luciano Lavagno, luciano.lavagno@polito.it; Teodoro Urso, teodoro.urso@polito.it, Politecnico di Torino, Italy; Enrico Calore, enrico.calore@infn.fe.it; Sebastiano Fabio Schifano, schsst@unife.it; Cristian Zambelli, cristian.zambelli@unife.it, Università degli Studi di Ferrara, Via Giuseppe Saragat, 1, 44122, Ferrara, Italy; Maurizio Palesi, maurizio.palesi@unict.it; Giuseppe Ascia, giuseppe.ascia@unict.it; Davide Patti, davide.patti@unict.it, Università degli Studi di Catania, Italy; Stefania Perri, s.perri@unical.it, Università degli Studi della Calabria, Italy; Nicola Petra, nicola.petra@unina.it; Davide De Caro, dadecaro@unina.it; Gennaro Di Meo, gennaro.dimeo@unina.it, Università degli Studi di Napoli Federico II, Italy; Valeria Cardellini, cardellini@ing.uniroma2.it; Salvatore Filippone, salvatore.filippone@uniroma2.it; Francesco Lo Presti, lopresti@info.uniroma2.it, Università degli Studi di Roma "Tor Vergata", Italy; Francesco Silvestri, francesco.silvestri@unipd.it, Università degli Studi di Padova, Italy; Paolo Palazzari, paolo.palazzari@enea.it, ENEA, Italy.

such as computer architecture, approximate computing, computational models, and deep learning algorithms. Moreover, existing architectures for accelerators in deep learning applications are highly heterogeneous. So, to achieve high performance and energy efficiency, while minimizing power consumption and area metrics, designers have adopted various methodologies, including high-level synthesis methodologies, specific customized compilers, tools for design space exploration, modeling, profiling, partitioning, and mapping. These methods focus on maximizing parallelism and minimizing data movement to optimize accelerator design for deep learning applications. Understanding existing frameworks and tools is crucial for fostering innovation in this domain.

**Aim and organization of the document.** The document is structured according to different topics and sub-topics. As shown in Figure 1, the document deals with the design methodologies for heterogeneous HPC platforms for Deep Learning. Each section covers notable and influential contributions and tries to put in perspective the work that the research group will do with respect to the Flagship 2 Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing.

The document is structured as follows: Section 2 presents the design methodologies for the high-level synthesis of accelerators, while Section 3 discusses automated compilation and deployment technologies for deep learning-based applications. Section 4 introduces approaches and tools that have been proposed to distribute, partition and map deep-learning models on heterogeneous processing systems. Section 5 describes the modeling, simulation, profiling, and design exploration frameworks currently adopted for DL-based applications. The last Section presents models of computations for HPC Deep Learning application workloads.

To conclude, we hope this survey could be useful for a wide range of readers, including computer architects, hardware & software developers, tool developers, HPC engineers, researchers, and technical professionals. A major effort was spent to use a clear and concise technical writing style: we hope this effort could be useful in particular to the young generations of master and Ph.D. students. To facilitate the reading, a list of acronyms is reported in Table 1.

Table 1. List of acronyms

| Acronym | Acronym | Acronym |
|---|---|---|
| **AI:** Artificial Intelligence | **ASIC:** Application Specific Integrated Circuit | **AXI:** Advanced eXtensible Interface |
| **BLAS:** Basic Linear Algebra Subprogram | **BRAM:** Block Random Access Memory | **CIM:** compute-in-memory |
| **CLA:** Carry-Look-Ahead | **CNN:** Convolutional Neural Network | **CPU:** Central Processing Unit |
| **DDDG:** Dynamic Data Dependence Graph | **DL:** Deep Learning | **DMA:** Direct Memory Access |
| **DNN:** Deep Neural Network | **DP:** Double Precision | **DRAM:** Dynamic Random Access Memory |
| **DSP:** Digital Signal Processing | **DSE:** Design Space Exploration | **FFT:** Fast Fourier Transform |
| **FIFO:** First In, First Out memory | **FP:** Floating Point | **FPGA:** Field-Programmable Gate Array |
| **GCC:** GNU Compiler Collection | **GEMM:** General Matrix Multiply | **GPU:** Graphics Processing Unit |
| **GNN:** Graph Neural Network | **HBM:** High Bandwidth Memory | **HDL:** Hardware Description Language |
| **HLS:** High Level Synthesis | **HPC:** High-Performance Computing | **HPZMO:** High Performance Zero-Memory Overhead |
| **IEEE:** Institute of Electrical and Electronics Engineers | **II:** Initiation Interval | **IP:** Intellectual property |
| **IR:** Intermediate Representation | **ISA:** Instruction Set Architecture | **LNS:** Logarithmic Number System |
| **LUT:** Lookup Table | **MCU:** Microcontroller Unit | **MEC:** Memory-efficient Convolution |
| **ML:** Machine Learning | **MLIR:** Multi-Level Intermediate Representation | **NN:** Neural Network |
| **NoC:** Network on Chip | **OpenMP:** Open Multi-Processing | **PE:** Processing Element |
| **PIM:** Processing In-Memory | **PPA:** Power, Performance, Area | **PPA:** Parallel-Prefix Architectures |
| **PRAM:** Parallel Random Access Machine | **QoR:** Quality of Results | **RAM:** Random Access Memory |
| **RAM:** Random Access Machine | **RISC:** Reduced Instruction Set Computer | **RTL:** Register transfer level |
| **RU:** Resource Units | **SIMD:** Single Instruction Multiple Data | **SoC:** System on Chip |
| **SP:** Single Precision | **SPP:** Structured Parallel Programming | **SRAM:** Static Random Access Memory |
| **VHDL:** Very High Speed Integrated Circuit (VHSIC) Hardware Description Language | **VLDA:** NVIDIA Deep Learning Accelerator | **XML:** Extensible Markup Language |
| **IMC:** In-Memory Computing | | |

## 2 HLS DESIGN-BASED METHODOLOGIES

In essence, high-level synthesis (HLS) serves as a link between software and hardware modeling, offering a variety of advantages. A first advantage is the ability to work at a higher level of abstraction when developing high-performance hardware, which boosts the productivity of hardware designers due to faster design changes and much faster functional verification. Moreover, HLS offers the ability

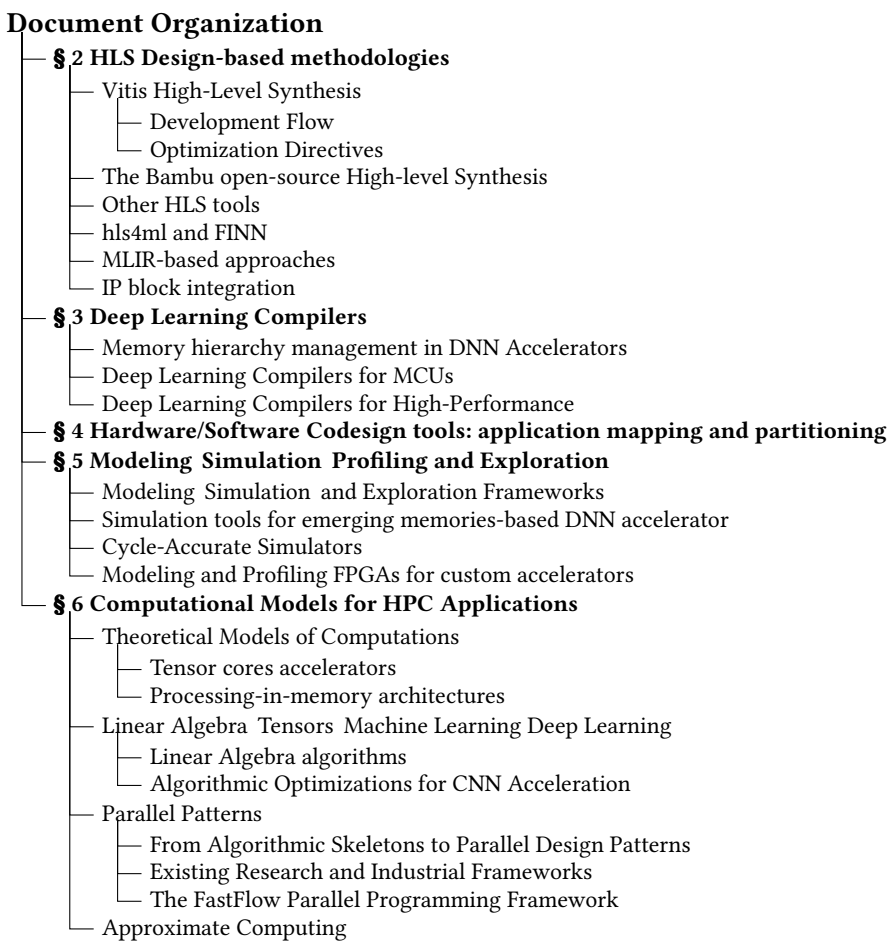


Fig. 1. Organization of the document

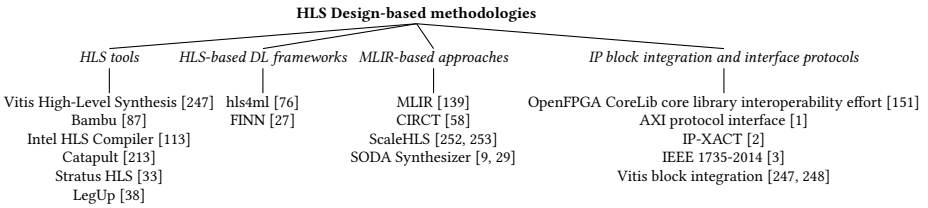


Fig. 2. HLS Design-based methodologies discussed in Section 2

to create various solutions on several platforms (e.g., larger or smaller FPGAs) without altering the C/C++ source code, by just changing design directives. This makes it possible to explore the design space and find the best implementation much faster than with low-level hardware design. Note that code must be written with a hardware implementation in mind in order to meet given performance and resource usage requirements. Arbitrary software code, written for a CPU target, can achieve very low performance, since it typically does not expose enough parallelism to exploit the spatial

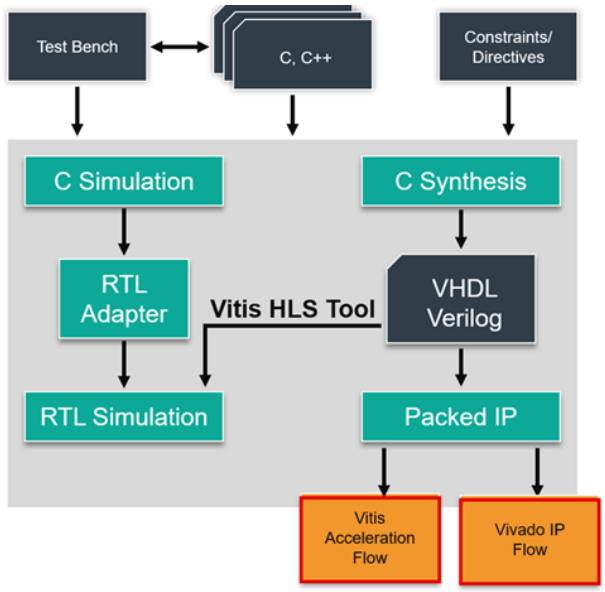


Fig. 3. Vitis HLS workflow (from [247])

concurrency available on an FPGA or an ASIC. A survey about HLS-based design methodologies, with a focus on the acceleration of deep learning (DL) models, is summarized in Figure 2 and detailed in the following sub-sections. Flagship2 selects as candidate software frameworks for the HLS of complex DL applications the commercial tool Vitis HLS and the open-source tool PandA-Bambu. This will foster innovation by allowing the implementation of new methodologies not yet available in closed-source tools and guarantee state-of-the-art quality of results provided by established commercial tools.

### 2.1 Vitis High-Level Synthesis

The Xilinx High-Level Synthesis tool, called Vitis HLS, allows designers to write a design to be implemented on an FPGA using high-level languages such as C and C++, rather than using RTL languages such as Verilog and VHDL. This design is then translated into RTL automatically, which in turn can be implemented on an FPGA. Vitis HLS significantly simplifies the tedious, time-consuming, and error-prone process of creating RTL code, which formerly required designers to grapple with low-level hardware implementation.

*2.1.1 Development Flow.* Figure 3 illustrates the Vitis HLS Development Flow:

- **Architect** the algorithm using C/C++, keeping in mind the need to expose parallelism when implemented, via pipelining and dataflow (see below).
- **C-Simulation**: Compile and execute the C/C++ code to simulate its behavior and ensure that it works as expected, by checking its functionality with a C/C++ testbench.
- **C-Synthesis**: Generate the RTL, by synthesizing the C/C++ top function. To instruct the synthesis process to carry out a certain optimization, HLS synthesis directives and constraints can be imposed directly, as discussed below. When C synthesis is finished, a comprehensive report with time and hardware resource usage estimation is produced, offering the designer crucial references for subsequent refinement and optimization.

- **Co-Simulation**: C/RTL Co-Simulation in Vitis HLS refers to the process of verifying and validating a hardware design written in RTL (register-transfer level) using the same C/C++ simulation testbench that was used for C functional simulation before. It thus provides the designer with cycle-accurate performance information (and it can also spot synthesis tool bugs).
- **QOR analysis**: Review and investigate the HLS synthesis reports and co-simulation reports.
- Repeat the prior steps until the desired Quality of Results has been achieved.

*2.1.2 Optimization Directives.* HLS synthesis directives, also known as *HLS pragmas*, are essential for optimizing the HLS process in Vitis HLS. Pragmas provide directives to the compiler, guiding the generation of hardware implementations from high-level C/C++ code. They enable users to control the micro-architectural aspects of the design, while the macro-architecture is defined by the C/C++ code, allowing for the optimization of specific objectives such as resource usage, performance (both number of clock cycles and clock period), or power consumption.

In Vitis HLS there are several types of synthesis directives, namely loop-level, variable-level, dataflow-level and operation-level. These optimizations can significantly improve the performance of the synthesized hardware, albeit often at the expense of increased resource utilization.

- **Loop-level pragmas** focus on optimizing the execution of loops in the design, supporting loop pipelining, loop unrolling, loop flattening, etc.
  Loop pipelining reduces the initiation interval (II), i.e. the number of cycles between successive executions of the loop body, by allowing the concurrent execution of operations. Figure 4 shows an example of the same function with and without loop pipelining (pipelining can also be applied to a function body, with the same effect).

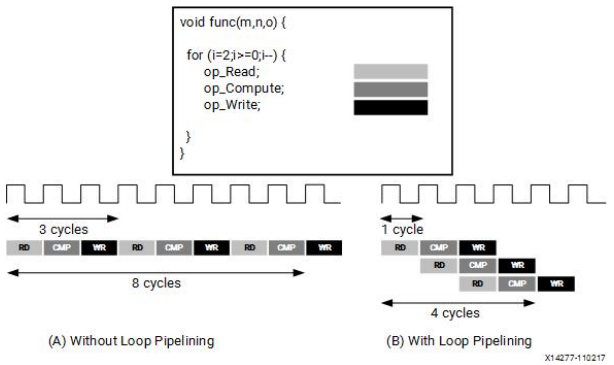


Fig. 4. Comparison of a function execution with and without loop pipelining in Vitis HLS (from [247])

Loop unrolling creates multiple independent copies of the loop, which enables some or all loop iterations to occur in parallel, with a significant resource cost.
Loop flattening allows nested loops to be flattened into a single loop, so that the body of the innermost loop is always restarted with its own II, without the cycle time overhead of restarting upper level loops.

- **Variable-level pragmas** are applied to specific variables in the design and can be used to control their storage or interface access characteristics. Examples include array partitioning, which can break an array into smaller sub-arrays, improving parallel access, and array reshaping, which changes the array's storage organization to optimize specific access patterns. They result in RTL with several small memories or multiple registers instead of

one large memory. They effectively increases the amount of read and write ports for the storage and hence potentially improves the throughput of the design. Of course, they also require more memory instances or registers, as is usual when improving performance. As is well known from HW design theory (consider e.g. the roofline diagram model), computation parallelism must be matched to memory access parallelism in order to create an optimal implementation.
These pragmas can also control the implementation of the underlying RAM, e.g. by defining the number of read and write ports, which in turn determines the RAM resource usage.
Other pragmas of this kind define the access protocol for top-level function argument, which become interface ports of the RTL. For example a scalar top argument can be read or written with an AXI stream protocol, using ready and valid signals for handshaking. Or it can be read from or written to a register which is also read or written by the code running on the processor that manages the FPGA logic (also called "host code"), for HW/SW interfacing.
A pointer top argument can be mapped to an on-chip RAM (also known as BRAM) or to the off-chip DRAM on the board, via an AXI-4 master port connected to the DRAM controller.

- **Dataflow pragmas** enable task-level, i.e. coarse-grained pipelining, in which functions calls are executed in a pipelined fashion, computing the same result as if they were executed one after the other like the original C/C++ code. This essentially enables hierarchical pipelining and can dramatically improve the performance of an application. Scalar or array variables written and read by functions that are in a "dataflow region" are converted into FIFO channels and Ping-Pong buffers in order to ensure functional correctness with respect to the C/C++ code.
- **Operation-level pragmas** determine the kind of resources to be used for specific operations. For example, a multiplier with small operands can be best implemented using Look-Up Tables (LUTs), while a wider one can exploit hardwired multiply and add resources, known as DSP units. Moreover, the designer can decide how many resources to allocate for a given kind of operation (e.g. a multiplication) in a given scope (e.g. a function or loop body). While Vitis HLS uses its own heuristics to make these resource kind and sharing selections, the designer can control them directly, to achieve a specific performance and resource QOR result.

### 2.2 The Bambu open-source High-level Synthesis

Bambu is a command-line tool developed by Politecnico di Milano aimed at assisting the designer during the HLS of complex applications. It supports most of the C/C++ constructs, including function calls and sharing of the modules, pointer arithmetic and dynamic resolution of memory accesses, accesses to arrays and structs, parameters passed by reference or copy, and many more. The whole flow is quite similar to a software compilation flow: it starts from a high-level specification, and it produces low-level code after a sequence of analysis and optimization steps. Like in a standard software compilation flow, Bambu has three phases (see Figure 5): front-end, middle-end, and back-end. In the front-end, the input code is parsed and translated in an intermediate representation used in the following parts of the flow. In the middle-end target-independent analyses and optimizations are performed. The back-end performs the actual synthesis of Verilog/VHDL code ready for simulation, logic synthesis, and implementation through external tools.

**Bambu front-end**. Bambu interfaces with existing compilers, such as GCC and Clang. With GCC, a plugin extracts the call graph and the control data flow graph of the functions under analysis from GCC's internal IR. Similarly, a Clang plugin extracts the same information and serializes them into a textual format easy to parse. Bambu then parses back all the compiler serialized information

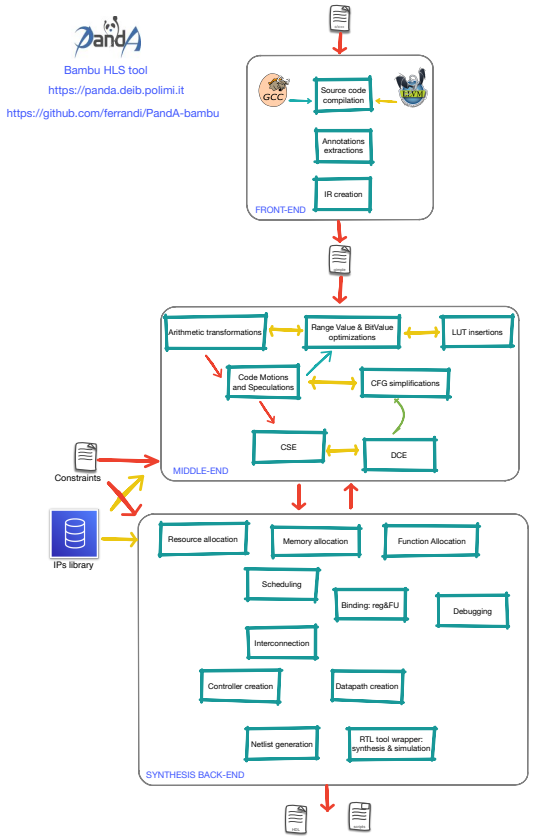


Fig. 5. Bambu Compilation flow.

plus all the annotations to build a Static Single Assignment in-memory IR. This approach decouples the compiler front-end code from the rest of the HLS process. Localizing all the changes in a GCC or LLVM/Clang plugin allows rapid and easy integration of many different versions of the compilers. Bambu supports GCC versions ranging from 4.5 to 8, and LLVM/CLANG versions ranging from 4.0 to 16. Moreover, the Vivado HLS front-end [20], based on a customized version of LLVM/CLANG and recently released in open-source, was effortlessly integrated into the Bambu framework.

**Bambu middle-end**. Starting from the intermediate representation extracted from GCC/Clang, Bambu rebuilds data structures, such as the Call Graph and the Control Data Flow Graphs, and builds additional data structures such as the Program Dependence Graphs. Next, it applies a set of device-independent analyses and transformations. Some of these steps are commonly used in a software compilation flow (e.g., data flow analysis, loop recognition, dead code elimination, constant propagation, LUT expression insertion, etc.). Multiplications and divisions by constant values are transformed into expressions that use only shifts and adders to reduce area utilization and improve timing. The resulting expression structure depends on the target device and technology, since adders and multipliers may have different performances on different devices. Differently from general-purpose software compilers, designed to target a processor with a fixed-sized data-path (usually 32 or 64 bits), a HLS compiler can exploit custom-size operators (e.g., a multiplier with the minimum number of I/O bits) and registers. Consequently, we can select the minimal number of bits required for the specific algorithm's operations and value storage, which leads to less area,

less power, and shorter critical paths. At this stage, Bambu also performs Bitwidth and Range Analysis, aiming at reducing the number of bits required by data-path operators. This analysis is crucial during the optimization process because it impacts all non-functional requirements (e.g., performance, area, power) of a design without affecting its behavior.

**Bambu synthesis back-end**. In this phase, Bambu performs the actual architectural synthesis of the specification. The synthesis process acts on each function separately. The resulting architecture reflects the structure of the call graph. A single function includes at least two sub-modules: the control logic and the data-path. Control logic modeled as a Finite State Machine handles the routing of the data values and the temporal execution of the operations. The data-path is a custom mux-based architecture with optimized data types to reduce the number of flip-flops and bit-level multiplexers. It implements all the operations and memories required during the function execution. The following paragraphs describe the sequence of steps that Bambu implements to generate control and data-path modules.

*Function Allocation.* Functions Allocation associates the high-level functions with specific resources available in the technology library associated with the target device. The technology library coming with Bambu integrates standard functions described in Verilog or VHDL, standard system libraries such as `libc` and `libm`, and designer-defined components written in Verilog or VHDL. Bambu supports function pointers and sharing of (sub)modules across module boundaries [158]. Sharing is obtained through function proxies, which act as forwarders of function calls in the original specification to shared modules. Sharing through function proxies provides valuable area savings when complex call graphs are considered, with no significant impact on the execution delays.

*Memory Allocation.* Memories Allocation defines the memories used to store aggregate variables (arrays and structures), global variables, and how the dynamic memory allocation is implemented. Bambu adopts an architecture for memory accesses that support a wide range of cases. Statically analyzing the memory accesses, Bambu builds a hierarchical data-path where memories can be classified as read-only, local, with aligned or unaligned memory accesses, or which require dynamic resolutions. The memory interconnection accordingly defines multiple busses connecting the load/store components to their respective memories. Dual-port BRAMs or memory controllers with complex parallel channels are supported by replicating such memory interconnections as needed. The same memory infrastructure can also connect to external components (e.g., scratchpads, caches, and DRAMs) or directly to the bus to access off-chip memory. Supporting protocol-based accesses (e.g., FIFO or stream-based access) is obtained by generating specific components that replace the load/store instructions.

*Resource Allocation.* Resource allocation associates operations not mapped on a function to resource units (RU) available in the resource library. During the middle-end phase, the specification is inspected to identify the characteristics of the operations: these include the type of the operation (e.g., addition, multiplication, etc.) and the types of the operands (e.g., integer, float, etc.). Floating-point operations are supported through the HLS of a soft-float library containing basic soft-float operators, or alternatively by exploiting the FloPoCo software [69], a generator of arithmetic Floating-Point Cores. The allocation step maps operations on the set of available RUs; their characterization includes information such as latency, area, and the number of pipeline stages. Usually, more operation/RU matchings are feasible: in this case, selecting a proper RU is driven by design constraints. The library of RUs used by Bambu is quite rich, and may include several implementations for the same operation. Moreover, the library contains RUs described as templates in a standard hardware description language (i.e., Verilog or VHDL). These templates can be retargeted and customized according to the characteristics of the target technology. In this case, it will be the underlying logic synthesis tool that determines the best architecture to implement

each operation (for example, multipliers can be mapped either on dedicated DSP blocks or implemented with LUTs). To perform aggressive optimizations, each library component is annotated with information useful during the entire HLS process, such as resource occupation and latency. Bambu adopts a pre-characterization approach: the performance estimation considers a generic template of the RU, which can be parametric with respect to the bit widths and pipeline stages; latency and resource occupation are then obtained by synthesizing each configuration and storing the resulting metrics in the library as an XML file.

*Scheduling*. By default, Bambu employs a List scheduling algorithm. In its basic formulation, List scheduling associates each operation with a priority according to particular metrics. The List scheduling proceeds iteratively, associating a set of operations to be executed with each control step. Ready operations (i.e., whose dependencies have been satisfied in previous iterations of the algorithm) can be scheduled in the current control step considering the availability of the resources. If multiple ready operations compete for a resource, then the one having a higher priority is scheduled. In addition to this old but efficient algorithm, Bambu also features a more aggressive scheduling algorithm, the Speculative scheduling algorithm based on System of Difference Constraints [141]. This algorithm builds an integer linear programming formulation of the scheduling problem, allowing code motion and speculation of operations that belong to different basic blocks.

*Module Binding*. Within the computed schedule, operations that execute concurrently are not allowed to share the same resource instance. In Bambu, binding is performed through a clique covering algorithm on a weighted compatibility graph [222]. The compatibility graph is built by analyzing the schedule: operations scheduled on different control steps are compatible. Weights express how much it is profitable for two operations to share the same hardware resource. They are computed considering area/delay trade-offs caused by sharing; for example, RUs that occupy a large area will be more likely shared. Weights computation also considers the cost of interconnections required by the steering logic. Bambu also offers several other algorithms for solving the covering problem on compatibility/conflict graphs.

*Register Binding*. Register binding associates storage values to registers and requires a preliminary analysis step, the liveness analysis [222]. Liveness analysis starts from the schedule to identify each variable's life intervals, i.e., the sequence of control steps in which a temporary value needs to be stored. Variables with non-overlapping life intervals may share the same register.

*Interconnection Binding*. Interconnections are bound according to the outcome of the previous steps: if a functional or memory resource is shared, then the algorithm introduces steering logic on its inputs. It also identifies the set of control signals that will be driven by the controller.

*Netlist Generation*. The final architecture is then generated and represented through a hypergraph, highlighting the interconnection between modules. The netlist generation step translates such representation in a register transfer-level (RTL) description in Verilog or VHDL. The process accesses the resource library, which embeds the RTL implementation of each resource. This process is target-dependent, and the hardware descriptions may differ for different technologies (e.g., ASIC or FPGA) or target devices.

*Generation of Synthesis and Simulation Scripts*. Bambu automatically generates synthesis and simulation scripts that can be customized via XML configuration files. The RTL-synthesis tools currently supported are AMD/Xilinx ISE, AMD/Xilinx Vivado, Yosis-Vivado, Intel/Altera Quartus, Lattice Diamond, NanoXplore, and OpenRoad. Supported simulators are Mentor Modelsim, Xilinx ISIM, Xilinx XSIM, Verilator, and Verilog Icarus.

### 2.3 Other HLS tools

For the most part, HLS tools are provided by FPGA vendors together with a full design suite that only supports the development of accelerators on FPGAs from the same company. The aforementioned

Vitis HLS, for example, is part of the AMD/Xilinx toolsuite and only supports Xilinx FPGAs. The Intel HLS Compiler [113] is part of the Quartus design suite, it compiles C++ functions into an RTL implementation for Intel FPGAs and optimizes them through a simple command-line interface. Intel recently announced that the HLS compiler will be deprecated in favor of the oneAPI toolkit [112], which could allow developers to seamlessly port OpenCL code across CPUs, GPUs, and FPGAs. Catapult [213] is a multi-target HLS and verification tool provided by Siemens, synthesizing C++ and SystemC code for FPGA and ASIC. Stratus HLS [33] from Cadence synthesizes SystemC code written with a lower-level perspective, i.e., requiring users to explicitly describe interface protocols between components. LegUp [38] is an open-source, LLVM-based HLS tool developed in academia, like Bambu, later acquired by Microchip and rebranded as SmartHLS [157].

### 2.4 hls4ml and FINN

HLS plays a crucial role in bridging the productivity gap between the design of a new deep learning model and its implementation on FPGA/ASIC. Several previous works proposed to exploit HLS by using C/C++ as an intermediate representation of the input model, augmenting it with tool-specific directives that drive the synthesis to obtain an efficient design. Two popular frameworks that help automate the design of ML accelerators are hls4ml [76] and FINN [27]. Both use commercial HLS tools as backend (mainly Vivado or Vitis HLS); they parse a model exported from popular ML frameworks and replace operators with C/C++ functions taken from a library of templates that already contains pragma directives. The HLS tool processes this intermediate C/C++ representation and produces a corresponding accelerator design without further manual intervention.

The library of templates in hls4ml and FINN is necessarily tied to a specific HLS tool and a narrow set of supported models, as it requires expert HLS developers to implement in advance the best version of all necessary operators for a pre-determined backend tool. Portability is a problem for HLS in general, as typically there is one commercial tool for each hardware vendor and each tool expects coding patterns, annotations, and configuration directives that are not recognized by other tools. In a framework that heavily relies on a library of templates, switching to a new hardware target thus requires a new version of the library, as incompatible coding patterns and directives would at best be ignored by the new HLS backend, resulting in inefficient designs.

Library-based frameworks usually focus on a narrow set of models, specifically deep and convolutional neural networks (DNNs/CNNs). Machine learning is an umbrella term that covers a broad spectrum of algorithms, while research works about hardware acceleration and HLS-based design flows have mostly been focused on the subset of ML models based on dense convolutions and matrix multiplications. Sometimes their scope is further limited by application requirements: for example, the original implementation of hls4ml was optimized for small, fully-connected models under tight latency constraints, reflecting the needs of a high-energy physics experiment at CERN. For this reason, hls4ml proposed to store network weights inside on-chip logic and unroll all loops to increase parallelism, which quickly depletes FPGA resources when considering a neural network with more layers and weights.

While it is true that DNNs and CNNs cover a significant part of ML applications (especially in the computer vision field), there is ample room for exploring other classes of models, for example to accelerate scientific applications that work on sparse data structures or graphs. Large models are often compressed to reduce their computation and memory requirements, either by employing low-precision data types (quantization) or by removing operations with zero values (pruning). Quantization is well suited to hardware acceleration since custom precision operators can be implemented quickly and efficiently (also through dedicated HLS libraries). Sparsity, on the other hand, implies irregular computation, communication, and memory access patterns, which result in poor efficiency when mapped on accelerators or templates designed for dense models. Graph

structures provide great expressive power to represent and analyze data in a variety of applications, from chemistry to language, social networks, recommendation systems, etc. Graph neural networks (GNNs) could benefit from hardware acceleration and require unique design choices: models that work on graphs include both sparse (aggregation) and dense (feature extraction) computation patterns, which are also affected by the input graph size; such characteristics could benefit from a task-based parallelism paradigm. Existing HLS-based design flows are good at extracting data- and instruction-level parallelism (e.g. by unrolling loops), but they are not equipped to deal with the irregular task-based patterns required by graph processing. Finally, a narrow focus limits the possibility of quickly adapting to new algorithmic approaches, which would instead be desirable in a rapidly evolving field such as ML (and data science in general).

In general, there are multiple research efforts that only use existing HLS tools as "black boxes", exploiting as much as possible the optimization opportunities they expose. In fact, research in this field is sometimes hindered by the proprietary nature of established HLS tools [176] (Bambu [87] is a notable exception). However, there is a trend toward the democratization of hardware design, as attested for example by the open-source release of the Xilinx Vitis HLS frontend [20] or by the OpenROAD project for ASIC synthesis [14].

### 2.5 MLIR-based approaches

The Multi-Level Intermediate Representation (MLIR) [139] is a reusable and extensible infrastructure in the LLVM project for the development of domain-specific compilers. MLIR allows defining specialized intermediate representations (IRs) called *dialects* to implement analysis and transformation passes at different levels of abstraction, and it can interface with multiple software programming frameworks, including the ones used to implement deep learning algorithms. MLIR has been used to build new design flows for the generation of hardware accelerators based on HLS.

The CIRCT project [58] intends to use MLIR to build a new generation of interoperable tools and compilers for hardware design, starting from the definition of circuit-level IRs and working upwards to higher levels of abstraction (e.g., dataflow models or finite state machines). Part of the project is dedicated to HLS [231], particularly to the implementation of static and dynamic scheduling through MLIR and CIRCT dialects. CIRCT could be an essential building block for future industrial and academic design flows; however, its degree of maturity is lower compared to HLS tools with optimized synthesis algorithms and resource libraries supported by decades of research.

ScaleHLS [252, 253] exploits MLIR to analyze and transform input code from C or PyTorch, generating annotated code for Vivado HLS (a slightly old version of the Xilinx HLS tool which does not apply any automated optimization). The multiple levels of abstraction provided by existing MLIR dialects allow ScaleHLS to reason about graph-level, loop-level, and directive-level optimizations; a custom dialect helps the translation into C++ with pragmas. A quality of results (QoR) estimator and a DSE engine automatically identify the best combination of optimizations following user-defined constraints, without requiring long simulation or synthesis runs to evaluate the effect of changes in the optimization directives.

The SOftware Defined Architectures (SODA) Synthesizer [9, 29] is an open-source, multi-level, modular, extensible, no-human-in-the-loop hardware compiler that translates high-level ML models into domain-specific accelerators. The SODA Synthesizer comprises a compiler-based frontend that leverages MLIR (SODA-OPT [28]) and a compiler-based backend that integrates state-of-the-art HLS methodologies (Bambu); it generates highly specialized designs that can be synthesized with both commercial and open-source tools on FPGAs or ASICs, and it allows the exploration of design metrics through compilation passes and parameters, enabling the identification of architectural trade-offs depending on the target application requirements.

### 2.6 IP block integration

When developing an HLS flow, the possibility to import designs produced by third parties (Intellectual Properties, IPs) as well as to export functionalities developed through the HLS flow as building blocks to be used in other designs is fundamental.

In the electronic world, it is a common practice to use IPs to add the desired functionalities to the system being developed. The use of IPs not only saves the development time that would be necessary if the functionality should be developed from scratch, but also frees the developers from the burden to qualify the behavior of the functionality: when acquiring an IP from an IP provider, we are buying not only the development time used by the developer but also the huge testing time that has been spent to qualify the IP.

On the other side, the high level of abstraction given by HLS flows and their maturity makes very interesting the possibility to export functionalities developed by the HLS as IPs. A first attempt to address this topic in a systematic way dates to 2008 [151], where the various strategies used by the HLS tools to integrate IPs were analyzed.

To allow IP generation/reuse, an interfacing standard is mandatory to allow interoperability among IPs. Currently, as a standard de facto, AXI4 [1] is used, among others, by companies like Synopsis and Xilinx. AXI4 standard includes AXI4, AXI4-stream, and AXI4-Lite protocols used to access memory banks, streaming channels, and memory-mapped registers.

Other than a common interface, a common language to describe the IP interfaces and the IP organization on the filesystem is needed. IP-XACT [2] is an XML format describing meta-data and interfaces of IPs and is widely adopted by IP providers to describe their IPs (file system organization, interfaces, source files, constraint files, ...).

The standardization of IPs structure for their distribution is ruled by the IEEE 1735-2014 standard [3]. As the IEEE 1735-2014 standard does not cover IP produced by HLS, let's see how Xilinx is managing the import/export of IPs in its Vitis flow.

Referring to the possibility of exporting a design produced by Vitis HLS as an IP to be used in other designs, as shown in Figure 3, Vitis flow allows exporting a kernel, compiled through Vitis HLS to generate an HDL IP, to be later incorporated in a design through the Vivado IP integrator flow [248]; so, in the export direction, through the Vivado IP flow there is complete support for using IPs generated by Vitis HLS in designs containing other IPs.

Looking at the opposite direction, Vitis flow partially opens to the import of external HDL IPs, giving the opportunity to add HDL blackbox functions [247]. In this case, HDL IPs are limited, being constrained to the adoption of AIX4 interface. As described in [79], when importing an HDL IP (that can be plain HDL, the synthesized netlist, or its encrypted version), also the C model of the IP can be provided, to allow the SW emulation of the design; this functionality is very useful to check the functional correctness of the design. Vitis provides a wrapper around the IP to make it compliant with the Vitis flow. It's worth to be underlined that the Xilinx IP flow manages the flow of IP between Vivado IP integrator and Vitis, being still an open issue the standardization of HLS IPs and their import/export among tools from different vendors.

## 3 DEEP LEARNING COMPILERS

Development of innovative hardware architecture, particularly for highly parallelizable applications such as Deep Learning, is only half of the picture. The other half is that of effective automated deployment technologies that enable to use novel architecture to run complex real-world applications. Managing the memory hierarchy and compile high-level signal processing and machine learning graphs into a representation is a complex research problem, which has been extensively studied in the pas few years (Fig. 6). In Flagship 2, the development of novel high-performance acceleration

techniques will be coupled with that of automated compilation and deployment technologies to boost real-world applicability of the developed hardware.

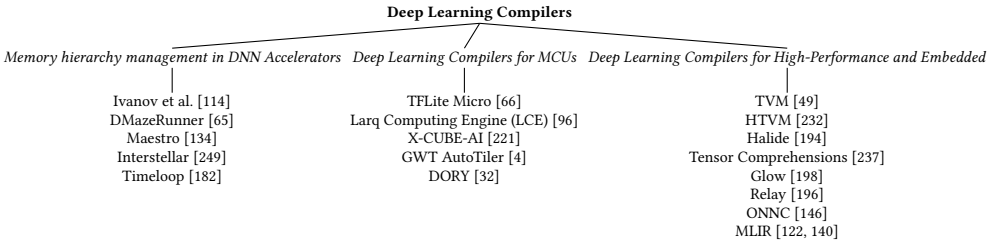


Fig. 6. Taxonomy of Deep Learning Compilers discussed in Section 3

### 3.1 Memory hierarchy management in DNN Accelerators

Effective management of memory hierarchy is a critical challenge in deploying deep neural networks (DNNs), which generate high amounts of weight and activation traffic between different levels of memory hierarchy. To tackle this problem, various methods have been proposed for data flow scheduling and generation across three broad classes of devices: high-performance computing systems, DNN accelerators, and embedded systems. For high-performance computing systems, Ivanov et al. [114] have proposed new transformer primitives to exploit data reuse and limit data movement. Meanwhile, DMazeRunner [65], Maestro [134], Interstellar [249], Timeloop [182] discuss DNN optimization on AI-specialized accelerators based on systolic arrays of processing elements (PEs), with a focus on loop tiling and/or reordering to optimize PE utilization. These tools can output an accelerator model to run a given DNN or spatial scheduling to maximize PE array utilization. MCU data flow scheduling tools are similar to frameworks like DMazeRunner as both optimize dataflow schedules given an externally known architecture. However, DNN execution on MCUs presents unique challenges such as adapting to a general-purpose architecture and limited memory. Additionally, kernel instructions are heavily influenced by the register file's limited size, resulting in increased load-store operations and a demand for optimal loop sizing to avoid register spilling overhead. Academic researchers and industries have investigated this aspect by incorporating specialized caches or explicitly managed scratchpad memories into their edge-node solutions. For example, NXP offers specialized caches in their Cortex M4/M0 MCU, as does STMicroelectronics with its STM32 Cube-AI toolflow; on the other hand, GreenWaves Technologies provides explicitly managed scratchpad memories [89], with a GAPFlow tool dedicated to managing them appropriately.

### 3.2 Deep Learning Compilers for MCUs

The introduction of the first generation of low-power neural-network oriented MCUs has increased this need, as these platforms need to utilize optimized software and ISA extensions for DNN computing alongside traditional control and I/O-bound activities. To allow for optimal execution of both types of tasks, these MCUs employ parallel and heterogeneous processing. ST Microelectronics[1] and NXP have recently introduced new-generation dual-core microcontrollers with an ARM M0 processor dedicated to I/O and an ARM M4 processor with single-cycle multiply-and-accumulate and SIMD capabilities. These platforms show an increased complexity in terms of memory hierarchy compared to conventional flat-memory MCUs, with an L1 memory optimized for speed and an L2

[1]https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html

optimized for capacity. At the same time, there is a trend towards explicit management of memory hierarchy, with hand-tunable data caches featuring locking for hand-crafted data management. For instance, the Kendrite K210 [2] is a RISC-V dual-core 64 bits system-on-chip with a neural network processor (KPU) on which the cores can offload the computation. It also includes dedicated memory banks for the NN accelerator and a DMA unit to explicitly manage the transfers. The SONY Spresense board [3] features a 6-cores M4 accelerator with a maximum clock speed of 156 MHz, 1.5 MB of SRAM and 8 MB of Flash. The GreenWaves Technologies GAP-8 [89] system-on-chip was introduced in 2018 as a commercial embodiment of the *Parallel Ultra-Low-Power* paradigm [60]: it features one I/O core and an 8-core SIMD-optimized DSP cluster accelerator using an extension of the RISC-V ISA. To manage this complexity, these MCUs include dedicated infrastructure for data marshaling, such as general-purpose DMA controllers to speed-up memory transfers and reduce the memory access bottleneck.

New tools such as TFLite Micro [66] and the Larq Computing Engine (LCE) [96] offer a model-agnostic deployment framework and overcome these problems. Both are non-vendor-locked tools supporting ARM Cortex-M and RISC-V cores. Their library memory footprints require only 16 kB on a Cortex-M3; however, by default they rely on graph interpretation at runtime, limiting achievable performance. To offset this limitation, TFLite Micro allows plugging in optimized kernels and declaring vectors in different memory regions. However, it does not include any tiling mechanism to execute layers that do not fit on-chip memory.

The two most powerful DNN deployment tools for microcontrollers available in the state-of-the-art have been proposed by the industry as proprietary, vendor-locked solutions for their own MCUs. X-CUBE-AI [221] from STMicroelectronics is an automatic NN library generator optimized on computation and memory. It converts a pre-trained DNN model from DNN tools such as Tensorflow into a precompiled library for the ARM Cortex-M cores embedded in STM32 series MCUs. X-CUBE-AI relies on relatively large on-chip L1 caches (up to 16 kB) to deliver performance on STM32 MCUs, and it does not tackle software-based memory management. On the other hand, GWT designed a tool called AutoTiler, to target the GAP-8 RISC-V based multi-core ultra-low-power microcontroller. One of its primary functions is to take a pre-trained DNN and generate code for memory tiling and efficient transfers of weight and activation data between all memory levels (on- and off-chip). The GWT AutoTiler directly tackles the data-movement and tile sizing challenge to optimize memory access, reaching state-of-the-art performance on the execution of many networks. The tool is proprietary, but its backend basic kernels are available as open-source as part of the GAP-8 SDK[4]. DORY [32] targets the same platform with an open-source tool. It optimizes the memory traffic for DNN deployment on specialized edge devices. By generating C code that tiles the execution of a dedicated kernel library, DORY reduces the size of intermediate buffers. This is crucial since microcontrollers often have limited level-1 (L1) memory. To achieve this, DORY formalizes tiling as an optimized constraint programming problem with kernel-specific heuristics. The produced code is more optimized but less general than previous solutions. Using DORY on a new architecture requires creating a new dedicated kernel library, new templates, and reprogramming the tiler to tailor it to specific hardware.

### 3.3 Deep Learning Compilers for High-Performance

A popular DNN deployment framework that targets both high-performance embedded and edge devices is TVM [49]. TVM's primary optimization mechanism is autotuning: it quickly compiles

[2] `https://canaan.io/product/kendryteai`

[3] `https://developer.sony.com/develop/spresense/`

[4] https://github.com/GreenWaves-Technologies/gap_sdk

differently-scheduled yet equivalent kernel implementations, and after running those on hardware, the most optimal kernel is selected. As such, TVM can implicitly improve the execution time on CPUs and GPUs and fine-grained general matrix multiply (GEMM) accelerators like VTA [164]. Moreover, TVM's runtime can link in (vendor-provided) optimized kernels in LLVM IR, CUDA C, C/C++ into a standalone artifact with the bring your own codegen (BYOC) [51] infrastructure. However, using TVM's autotuning pipeline is impractical for specialized coarse-grained accelerators since proving coarse-grained kernel equivalence requires complex loop nest analysis. This can be bypassed by using BYOC, but in this way, many of the automatic optimization opportunities presented by the framework are lost. HTVM [232] uses DORY as a backend of TVM employing this technique.

A popular research avenue has been to increase the level of abstraction to compile Deep Learning based applications, using Domain Specific Language that mainly address tensor-level representations, such as the early examples of Halide [194] and Tensor Comprehensions [237]. Dedicated Deep Learning compilers such as Glow [198] have been focused on graph lowering techniques, using these earlier developments and ideas to build up systems that take a high-level description of a Deep Learning program, typically in the form a data-flow graph of operators, lower it into a set of Intermediate Representations (IRs) still centered on tensor-aware operations, and then deploy on target machine-specific code. A common graphical format for the input of such lowering passes is ONNX[5], whereas intermediate representations can be custom and dedicated to one particular framework (e.g., Relay [196] for Amazon's open source NNVM compiler) or deployed as a specialization of a more general IR [146]. In this regard, the most relevant example is MLIR [122, 140], a framework proposed in the context of the LLVM project that enables building custom intermediate representations for domain-specific computing. While this tool is not exclusive to Deep Learning, it has been proposed in response to the needs of the Deep Learning community and quickly risen to prominence.

## 4 HARDWARE/SOFTWARE CODESIGN TOOLS: APPLICATION PARTITIONING AND MAPPING

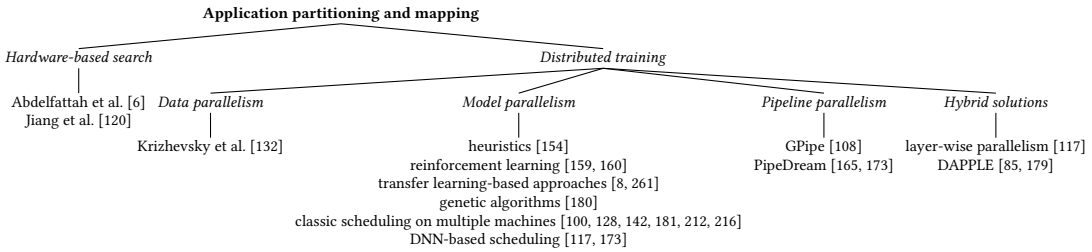


Fig. 7. Application partitioning and mapping discussed in Section 4

The ever increasing rate of data production in the era of Big Data, Internet of Things, and smart cyber physical systems pose incessantly escalating demands for massive data processing, storage and transmission as required by DL models training and inference[6].

[5]https://onnx.ai/

[6]Briefly speaking, the training process consists in adjusting the model parameters according to the results by backpropagation, while the inference process is enacted - without changing the parameters - when the network is then used to classify observed data.

Training large DL models with vast amounts of data and serving them (i.e., using trained DL models for inference) is a non-trivial task. Today, it is often performed in a distributed infrastructure composed of multiple, possibly heterogeneous compute nodes. The complexity is further exacerbated by the recent trend to integrate the high-performing computing and storage equipment in the cloud and HPC data centers to that provided at the edges of the network, where computing and memory resources are however constrained. The goals of this compute continuum trend are to achieve better privacy, higher autonomy and energy efficiency as well as to reduce response latency, cost, and bandwidth demand to the cloud [45]. In this complex and heterogeneous setting, designers need to optimize the complete system stack: from ML/DNN algorithms, to model optimization and compression, implementation of algorithms onto the hardware platforms enriched with DL accelerators as well as the underlying hardware architecture design [102, 115, 121, 228].

In this section, we review some approaches and tools that have been proposed in literature to distribute, partition and map DL training and inference applications on the processing nodes in the underlying computing infrastructure. First, we briefly analyze how DNN models can be optimized for execution on a plethora of hardware devices. We then focus on the approaches for training DL models in the context of distributed computing infrastructures. Finally, we conclude the section by identifying related open issues that can be addressed in the context of Flagship 2. They mainly stem from the need to reduce the energy footprint of DNN applications while keeping satisfactory levels of performance and accuracy. Figure 7 shows the main references to the methodologies, frameworks and tools that we discuss in the following of the this section.

The hardware-aware design of DNNs has recently received increasing attention to tackle hardware devices heterogeneity, especially to perform DNN inference. Indeed, to deploy computationally demanding DNNs for model inference in resource-constrained edge systems while maintaining acceptable performance, system designers have to trade off model accuracy against implementation efficiency. However, the plethora of available hardware devices available makes it very difficult to choose one solution for all cases. Therefore, in addition to techniques for model compression, such as quantization-aware training and pruning (e.g., [115, 228]), hardware-aware neural architecture search [52], that takes hardware characteristics like latency, power, or area into account, has become a central aspect in automating the process of designing efficient and accurate architectures for DNN applications executed at the network edges. Different methodologies have been exploited to search for the optimal performing model architecture, ranging from reinforcement learning to evolutionary algorithms. For example, in [6, 120] reinforcement learning-based neural architecture search is extended to include search for an accelerator configuration on FPGAs and optimize it for latency and area.

Nevertheless, in the context of a distributed infrastructure with an ever increasing number of available nodes and resources, it is parallelization which appears to offer the solution for the ever growing need of accelerating the training of DNN applications. DNN models lead themselves with many possibilities for parallelization, namely data, model, pipeline and hybrid parallelism.

In *data parallelism*, a number of workers (machines or devices, e.g., GPUs) load an identical copy of the DL model. The training data is split into non overlapping portions and fed into the model replicas of the workers for training [132]. Each worker performs the training on its portion of training data, which leads to updates of the model parameters. Hence, the parameters of the model among the workers need to be synchronized. The main advantage of data parallelism is that it is applicable to any DL model architecture without further domain knowledge of the model. It scales well for operations that are computationally intensive, but have only few parameters, such as CNNs. However, data parallelism is limited for operations that have many parameters, as the parameter synchronization leads to a significant communication overhead and may become the bottleneck [117]. To address such scalability and single point of failure bottleneck, the parameters

synchronization can occur in a decentralized manner [153], with the main disadvantage of increasing the communication cost among workers.

In contrast, in *model parallelism*, the DL model is partitioned into multiple parts and each worker loads a different part of the ML/DNN model for training. A major challenge of model parallelism is how to split the model into partitions that are assigned to the parallel workers [154]. In the context of ML/DNN workloads, model partitioning across different devices has initially mostly been a manual process driven by human experts. A common approach to find a good model splitting is to use reinforcement learning [159, 160]. Starting from some initial partitioning, permutations on that partitioning are performed, and performance is measured (e.g., for one training iteration) or learn a placement policy that can then be adjusted for new workloads via transfer learning, see e.g., [8, 261] or used to bootstrap a genetic algorithm [180]. Unfortunately, these methods are computationally expensive, as they need to evaluate large numbers of placements and measure the runtime of several inference/training steps. Alternatively, the problem is casted into an offline optimization problem of finding good partitions and schedules. This includes classic results in scheduling on multiple machines and/or devices [100, 128, 142, 181, 212, 216], as well as modern DNN scheduling works [117, 173]. Such algorithms use profiled compute time of each node (layer or operator) and data-transfer requirements between nodes in a graph, and the target deployment system infrastructure such as machine and network properties (e.g., measured bandwidths). However, such techniques do not evaluate the performance of splits in an online fashion. Nevertheless, it has been demonstrated that for well-defined cost models the objective function closely matches real performance, see, e.g., [118, 173].

*Pipeline parallelism* combines model parallelism with data parallelism. In pipeline parallelism, the model is split and each worker loads a different part of the DL model for training. Recent approaches that support pipeline parallelism include GPipe [108] and PipeDream [165, 173]. Specifically, in pipeline parallelism the model is divided among available workers, assigning a group of consecutive operators (called layers in DNN terminology) in the operator graph to each of them, and then overlapping the computation and communication of different inputs in a pipelined fashion. This process can greatly reduce inter-worker communication. While pipelining is a simple and widely adopted idea, DNN training poses an important challenge not present in traditional pipelining: DNN training is bi-directional, being the forward pass followed by a backward pass through the same layers in reverse order, using state and intermediate results from the forward pass. This results into low hardware efficiency or low statistical efficiency unless resorting to parallelization optimization [173].

Proposals related to pipeline training can be classified according to the temporal aspect, that is *synchronous* vs. *asynchronous* training. The first requires to execute gradient synchronizations between adjacent training iterations to ensure convergence [108]. However, it suffers from a significant memory consumption, that can be partially mitigated by re-computation. Asynchronous training inserts micro-batches into the pipeline concurrently to achieve maximum throughput, e.g., [173]. However, it is not a common practice due to convergence concerns and increased memory demand to store multiple versions of model parameters.

A few frameworks attempt to find a *hybrid* solution instead that combines some of the best properties of each model of parallelism and diminishes some of the drawbacks. For example, layer-wise parallelism [117] proposes to apply different parallelization strategies to each individual layer of the neural network rather than the same parallelization strategy (i.e., data or model parallelism) to all layers. The solution to find the optimal parallelization strategy for each layer is based on a dynamic programming based graph search algorithm. DAPPLE [85, 179] is a synchronous training framework which combines data parallelism and pipeline parallelism for large DNN models to ensure training convergence and reduce memory consumption. To this end, it exploits early

backward scheduling by scheduling backward tasks as early as possible to release the memory occupied by activations produced by corresponding forward tasks.

The approaches to distribute DL training and inference we have reviewed above aim typically to speed-up performance, for example by achieving better throughput and scalability, reducing communication costs, while improving (or at least without deteriorating) model accuracy. In the recent years, following a general trend within the industry at large, the reduction of carbon emission, the so called *green carbon footprint*, has started to receive increasing attention also within the HPC and ML/DNN communities in order to realize environmentally-responsible solutions, e.g., [243]. Given the high computational demand of DL training and inference jobs, there is a large opportunity for energy saving. For instance, it is possible to save energy while maintaining adequate level of accuracy at the software level by trading off model variants, i.e., low and high quality models. At the hardware level, multiple solutions can be exploited, ranging from the adoption of energy-efficient FPGAs to novel GPU partitioning schemes, that can reduce energy consumption by allowing GPU sharing [143]. Coupling with proper distributed resources scheduling, there is therefore a large opportunity for improving performance while reducing cost and carbon emission.

Within this general context, our work within the framework of Flagship 2 will address the need to develop DL application partitioning and mapping strategies for the edge-cloud continuum. Our goal is to design autonomic strategies optimized for both the training and inference phase which account for different non-functional requirements such as performance (e.g., training time), energy consumption as well as results accuracy in an ever growing, highly heterogeneous edge-cloud landscape in which ML/DNN workloads are executed. In this context, heterogeneity stems from the many different hardware/software platforms which comprise today's edge-cloud computing infrastructures. To this end, we plan to adopt reinforcement learning techniques, which has been widely used in the literature, see, e.g. [120, 160, 200], and DL in particular, to account for the large state space which characterizes the edge scenarios, whereby multiple nodes, possibly characterized by their own processing, memory, networking capabilities, and energy footprint are pooled to train and serve ML/DNN models.

## 5 MODELING, SIMULATION, PROFILING AND EXPLORATION

Hardware accelerators are becoming increasingly important in the field of deep learning, as they can significantly improve the speed and efficiency of deep learning computations. To effectively design a hardware accelerator for deep learning, it is essential to have access to powerful modeling tools that can provide detailed insights into the power consumption, performance, and area requirements of the accelerator. In this section, we will explore some of the most popular and effective tools available for modeling hardware accelerators for deep learning, and discuss their key features and capabilities. These tools enable designers to experiment with various design choices and configurations, and to optimize their designs for specific PPA metrics, such as power efficiency, throughput, or chip area. By leveraging these tools, designers can create hardware accelerators that meet the demanding performance and energy efficiency requirements of modern deep learning applications.

The simulation and exploration techniques, along with the profiling techniques discussed in this section, will serve as a foundation for selecting the most suitable approaches to address the design and optimization challenges of Flagship 2. Specifically, we are referring to challenges such as: a) Simulating complex heterogeneous accelerator architectures at a high level of abstraction, while being able to assess various figures of merit typically obtained at a low level of abstraction; b) Exploring the extensive design space encompassing architectural parameters and mapping possibilities to determine the optimal accelerator architecture for specific workloads; c) Utilizing appropriate techniques for modeling and profiling FPGAs specifically for custom accelerators.

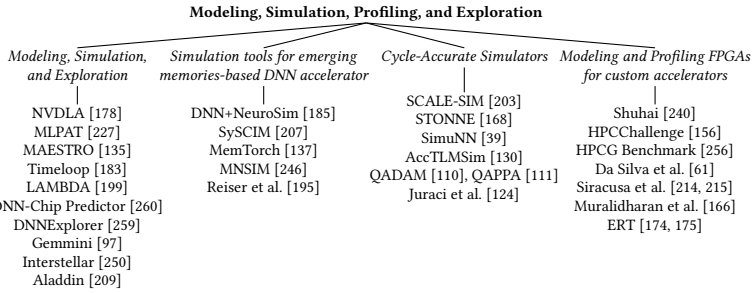


Fig. 8. Modeling, Simulation, Profiling, and Exploration tools and methodologies discussed in Section 5

This section is organized into four subsections as follows. Section 5.1 provides a review of representative simulation and exploration platforms that operate at a high level of abstraction. These platforms allow for the simulation and evaluation of domain-specific hardware accelerators, as well as the optimization of system architecture to achieve specific objectives such as delay and energy efficiency. Section 5.2 presents a collection of simulation tools and frameworks for assessing DNN accelerators that utilize emerging memory technologies. Section 5.3 focuses on simulation frameworks that ensure cycle accuracy. Lastly, Section 5.4 addresses the challenge of modeling and profiling FPGAs for custom accelerators. Figure 8 shows the references to the various tools, frameworks, and methodologies discussed in the aforementioned subsections.

### 5.1 Modeling, Simulation, and Exploration Frameworks

In this section, we aim to provide a comprehensive overview of the most influential frameworks utilized for modeling, simulating, and exploring the design space of hardware accelerators for deep learning. These frameworks can aid researchers in identifying the most effective hardware designs for specific deep learning tasks and can help accelerate the overall design process.

NVIDIA Deep Learning Accelerator (NVDLA) [178] is an open-source framework designed to facilitate the implementation of machine learning applications. It includes a complete training infrastructure and a compiler to convert existing models for use by NVDLA software. NVDLA can read a neural network from a front-end environment like Caffe and map it to the NVIDIA accelerator.

The MLPAT framework [227] enables modeling of power, area, and timing for machine learning accelerators, supporting components like systolic arrays, memory, and activation pipeline, as well as different precision types and dataflows. Input parameters include the accelerator architecture, circuit, and technology, and MLPAT generates an optimized chip representation to report results such as area, power, and performance.

MAESTRO [135] is a framework designed to analyze and describe neural network processing engines, providing information on the hardware cost required to implement a target architecture. It features a domain-specific language for dataflow description, which enables the specification of parameters such as the number of processing elements, memory size, and NoC bandwidth. The framework generates performance analysis results.

Timeloop [183] is an infrastructure that helps explore and evaluate the architecture design space of deep neural network (DNN) accelerators. It consists of two main components: a model that provides projections for performance, area, and energy, and a mapper that constructs and searches through the mapspace of a given workload on a targeted architecture. To use Timeloop, the user describes the architecture's organization using a configurable template that includes

Table 2. Modeling, simulation, and exploration tools.

| | Integration with NN frameworks | Model type | Full SoC | Evaluation metrics | Target | Estimation error |
|---|---|---|---|---|---|---|
| MLPAT [227] | No | Analytical | No | PPA | ASIC | <5% area<br><10% power |
| MAESTRO [135] | No | Empirical | No | Performance | ASIC | 5% |
| Timeloop [183] | No | Analytical/ Empirical | No | PPA | ASIC | 5% |
| LAMBDA [199] | No | Analytical/ Empirical | No | PPA | ASIC | 5% |
| DNN-Chip Predictor [260] | No | Analytical | No | Performance Energy | FPGA/ASIC | <18% |
| DNNExplorer [259] | Caffe, PyTorch | – | No | Performance | FPGA | – |
| Gemmini [97] | No | Simulation | Yes + OS support | Performance | FPGA/ASIC | – |
| Interstellar [250] | No | Analytical | No | PPA | ASIC | 2% |
| Aladdin [209] | No | Simulation Analytical | No | PPA | ASIC | 1% performance<br>5% power<br>7% area |
| SCALE-SIM [202, 203] | No | Empirical | Yes | Performance, Area | ASIC | – |
| STONNE [168] | Caffe | Cycle level simulation | Yes | Performance | ASIC | <3% |
| SimuNN [39] | TensorFlow | Cycle level simulation | Yes | PPA | FPGA/ASIC | – |
| AccTLMSim [130] | No | Cycle level simulation | Yes | Performance | ASIC | 3% |
| Juracy *et al.* [124] | TensorFlow | Cycle level simulation | No | PPA | ASIC | <7% |
| DNN-NeuroSim [185] | Tensorflow, PyTorch | Instruction accurate simulation | Yes | PPA | ASIC | - |
| SySCIM [207] | No | Circuit level simulation | No | Accuracy | ASIC | <4% accuracy |
| Memtorch [137] | PyTorch | Analytical/ Empirical | Yes | PPA | ASIC | - |
| MNSIM [246] | No | Cycle level simulation | Yes | PPA | ASIC | - |

abstractions for compute units, memories, and communication links. The mapper then constructs the mapspace and searches for an optimal mapping using the model's speed and accuracy. Timeloop's effectiveness has been validated against existing designs. PPA figures can be obtained by integrating it with Accelergy [245]. Accelergy is a versatile energy estimation technique that can be used for accelerators. It enables designers to create specifications using custom high-level compound components and low-level primitive components, which can be evaluated using third-party energy estimation plug-ins. LAMBDA [199] is a framework based on Timeloop/Accelergy infrastructure that allows exploring the design space of configurable DNN accelerators taking into account a variety of architectural and microarchitectural parameters.

The DNN-Chip Predictor [260] is a tool that can predict the energy consumption, throughput, and latency of DNN accelerators before they are implemented. It offers two advantages: (1) it uses an analytical performance formulation to enable rapid exploration and optimization of DNN ASIC/FPGA accelerator designs; and (2) it supports different algorithm-to-hardware mappings and hardware architectures. Experiments involving two DNN models and three ASIC/FPGA implementations demonstrated that the predicted performance of DNN-Chip Predictor differed from the chip measurements of FPGA/ASIC implementation by no more than 17%, even when using different DNN models, hardware architectures, and dataflows.

DNNExplorer [259] is a tool that helps to test customized hardware accelerators for DNNs and explore new accelerator designs with better performance and efficiency. It supports popular machine learning frameworks (Caffe and PyTorch) for analyzing DNN workloads and provides analytical

models for accelerator benchmarking. It has a high-dimensional design space and fine-grained adjustability to overcome design limitations, and a design space exploration engine to generate optimized accelerators based on targeted AI workloads and available hardware resources.

Gemmini [97] provides an effort to assess DNN accelerators, taking into account cross-stack and system-level effects in real-world scenarios. This enables a better understanding of the impact of SoC resource contention, OS overheads, and programming stack inefficiencies on overall performance and energy efficiency. Gemmini is an open-source DNN accelerator generator that enables users to design custom hardware accelerator systems for rapidly evolving DNN workloads. It provides a complete solution that spans both hardware and software stack, and is compatible with the RISC-V ecosystem. Gemmini's hardware design options can be tuned for performance, efficiency, and extensibility. It implements a multi-level software stack with an easy-to-use programming interface and tight integration with Linux-capable SoCs. Gemmini-generated accelerators have been successfully fabricated in TSMC 16 nm FinFET and Intel 22 nm FinFET Low Power process technologies, and deliver comparable performance to state-of-the-art commercial DNN accelerators.

DNN accelerator micro-architectures and their program mappings are specific choices of loop order and hardware parallelism for computing the seven nested loops of DNNs. It has been observed that these hardware variants can be precisely and concisely represented by Halide's scheduling language. In Interstellar [250], modifications were made to the Halide compiler to generate hardware that allows for fair comparisons with prior accelerators. Interstellar highlights the significance of optimizing the memory hierarchy since it is noted to have a greater impact on energy metrics than the selection of dataflow.

Aladdin [209] is a simulation tool that allows for quick exploration of design options for systems that are focused on accelerators. It is a pre-RTL and power-performance simulator that takes in algorithm descriptions in high-level languages and uses dynamic data dependence graphs (DDDG) to represent an accelerator without the need to generate RTL. It applies optimizations and constraints to an unconstrained program DDDG to create an accurate model of accelerator behavior. Its effectiveness has been confirmed through comparison with RTL implementations of accelerators created with both handwritten Verilog and commercial HLS tools, demonstrating that it can model performance, power, and area with high accuracy. Furthermore, Aladdin provides these estimates much more rapidly than traditional RTL flows, at over 100 times faster.

### 5.2 Simulation tools for emerging memories-based DNN accelerator

Another set of tools for DNN modeling, simulation and profiling is that related to emerging memories-based accelerators.

DNN+NeuroSim [185] is an integrated framework to benchmark compute-in-memory (CIM) accelerators for deep neural networks, with hierarchical design options from device level, to circuit-level and up to algorithm-level. A python wrapper is developed to interface NeuroSim with popular machine learning platforms such as Pytorch and Tensorflow. The framework supports automatic algorithm to hardware mapping, and evaluates both chip-level performance and inference accuracy with hardware constraints.

SySCIM [207] considers the impact of the non-idealities of the CIM components, including memristor device, memristor crossbar (interconnects), analog-to-digital converter, and transimpedance amplifier, on the vector-matrix multiplication performed by the CIM unit. The CIM modules are described in SystemC and SystemC-AMS to reach a higher simulation speed while maintaining high simulation accuracy. Experiments under different crossbar sizes show SySCIM performs simulations up to 117× faster than HSPICE with less than 4% accuracy loss.

MemTorch [137], is an open-source framework for customized large-scale memristive Deep Learning (DL) simulations, with a refined focus on the co-simulation of device non-idealities.

MemTorch also facilitates co-modeling of key crossbar peripheral circuitry. MemTorch adopts a modernized software engineering methodology and integrates directly with the well-known PyTorch Machine Learning (ML) library.

MNSIM [246] proposes a simulation platform for the memristor-based neuromorphic system with a hierarchical structure and flexible interfaces for customization. A detailed reference design is provided for large-scale applications like ISAAC or PRIME accelerators demonstrated in the previous deliverable. A behavior-level computing accuracy model is incorporated to evaluate the computing error rate affected by interconnect lines and nonideal device factors. Experimental results show that MNSIM achieves over 7000 times speed-up than SPICE simulation. MNSIM can optimize the design and estimate the tradeoff relationships among different performance metrics for users.

In [195], we wanted to propose a simulation framework which comes with the suitable abstractions to propagate the effects of those RRAM crossbar configuration parameters to their ultimate implications over inference performance stability. RRAM devices non-idealities result in significant inference accuracy drops compared with software baseline accuracy. A critical one is related to the drift of the conductance states appearing immediately at the end of program and verify algorithms that are mandatory for accurate multi-level conductance operation. The support of drift models in state-of-the-art simulation tools of memristive CIM is currently only in the early stage, since they overlook key device- and array-level parameters affecting drift resilience such as the programming algorithm of RRAM cells, the choice of target conductance states and the weight-to-conductance mapping scheme. In this work we fully exposed these parameters to RRAM crossbar designers as a multi-dimensional optimization space of drift resilience.

### 5.3 Cycle-Accurate Simulators

For accurate simulations, it is crucial that the simulation tools provide cycle accuracy, which means that they must model the behavior of the hardware accelerator at a cycle-by-cycle level, accounting for all the interactions between the different hardware components. This section will focus on simulation tools for hardware accelerators that offer cycle accuracy.

SCALE-SIM (SystoliC AcceLErator SIMulator) [202, 203] is a simulator that provides cycle-accurate modeling for DNN accelerators. It takes into account various factors such as on-chip and off-chip memory accesses, and interface bandwidth information for a given neural network. It has two primary components: (i) a compute unit that utilizes a systolic array that can be customized according to size and aspect ratio, and (ii) an accelerator memory system that features three double-buffered SRAM memories with user-specified sizes. These buffers store the matrices for two operands and one result. SCALE-SIM gets in input the layer dimensions of a specific neural network workload and the hardware architecture parameters and provides in output performance and energy figures.

STONNE (Simulation Tool for Neural Network Engines) [168] is a highly modular and extensible simulation framework that enables the end-to-end evaluation of flexible accelerator architectures running complete contemporary DNN models with cycle accuracy. STONNE has been validated by simulating the MAERI architecture and comparing the total executed cycles with that of a BSV-coded MAERI implementation. The results showed an average deviation of 15%. Like in Timeloop, STONNE uses the Accelergy energy estimation tool to estimate energy and area.

SimuNN [39] is a pre-RTL neural network simulator that allows for early phase verification and fast prototyping before the design is converted into hardware. It supports different data precision and is compatible with TensorFlow. SimuNN provides multi-level trace results that can be used as a reference for the final hardware design. Additionally, it can evaluate the hardware performance under various quantizations, dataflow, and configurations based on a generalized hardware model.

AccTLMSim [130] is a pre-RTL simulation tool which utilizes SystemC transaction-level modeling (TLM) to simulate convolutional neural network (CNN) accelerators with cycle accuracy. The tool includes a detailed model of the interface with the DRAM, allowing for precise tracking of each bus transaction between the accelerator and DRAM while considering the communication bandwidth. The validity of the simulation results is confirmed by comparing them to the implementation results on the Xilinx Zynq, resulting in an average estimation error of less than 10%.

QADAM [110] and its evolution QAPPA [111] are parameterized RTL frameworks that have been designed to model the power, performance, and area of quantization-aware deep neural network (DNN) accelerators. The frameworks allow for design space exploration and Pareto-efficiency analysis for a range of design choices, including bit precision, processing element (PE) type, scratchpad sizes of PEs, global buffer size, total number of PEs, and DNN configurations. By using QADAM/QAPPA, researchers can examine the impact that different bit precisions and PE types have on performance, area, and energy consumption.

In [124], a DSE approach for CNNs that is both fast and accurate is introduced. The approach employs an analytical model which is derived from the physical synthesis of hardware accelerators. This model is integrated into CNN frameworks such as TensorFlow, enabling it to produce precise outcomes. The analytical model provides estimations for various factors, including area, performance, power, energy, and memory accesses. The accuracy of the model was tested by comparing it to data obtained from physical synthesis, and it was observed that the average error was less than 7%.

### 5.4 Modeling and Profiling FPGAs for custom accelerators

The use of off-the-shelf highly parallel hardware accelerators to boost the performance of software applications, and deep learning algorithms in particular, is nowadays a very common option, adopted by a large and increasing share of HPC systems. In this sector, GPUs are definitively the most common accelerators, while FPGAs are hardly, or even not, used at all. Despite this, some data centers have recently started to adopt FPGAs to speed-up network interconnects [192], and specific workloads [19] such as Machine Learning (ML) inference algorithms [91, 210]. In fact, FPGAs could represent an interesting trade-off, allowing user customizations, as well as the use of off-the-shelf hardware, to implement custom deep-learning accelerators.

Given the rapidly increasing use of ML methods in several application fields, and the interest in reconfigurable architectures, which is rising in the HPC community since several years [80, 234, 238], we may expect FPGAs to become a more common option, as accelerators, for next generations of HPC systems. In the past, several reasons have prevented this. First, FPGAs were not designed to provide high floating-point (FP) computing performance [238], while typical HPC workloads usually require double-precision (DP) and single-precision (SP) FP computations. Secondly, FPGA programming could be a very time consuming process, requiring the use of specific hardware programming skills and the use of programming languages not common among HPC developers communities [23]. Thirdly, the code written for one FPGA could hardly run across different devices without a complete re-design, causing serious portability issues not acceptable for a wide set of HPC applications, for which even the porting to GPUs had been a long and suffered process [233].

However, more recently, these barriers started to fade thanks to improvements in hardware architectures and programming frameworks. In fact, latest generations of FPGAs integrate thousands of programmable DSPs (Digital Signal Processors) able to implement SP- and DP-FP operations [31, 197, 235], and may also embed custom FP DSP blocks. This is leading to devices able to reach a performance in the same order of magnitude as commodity HPC processors (i.e. TFLOP/s), and in some cases able to deliver a better energy-efficiency [25, 263]. At the same time, the recent improvements of synthesis tools, and the development of new programming approaches such as

HLS (High Level Synthesis) [171], allow programmers to develop codes using high level languages. As an example, OpenCL [263] could be used, as well as plain C/C++ annotated with *pragma* directives to guide the compiler to automatically map the code onto FPGA hardware resources [70]. These approaches are very similar to those (e.g. OpenMP and OpenACC) commonly used by HPC developers to target multi-core CPUs and other accelerators, which are also able to guarantee a fair level of code portability [30].

All the above improvements combined with the urging quest for higher energy-efficiency and lower latency interconnects in exascale HPC systems, are leading to a significant increase in the interest towards heterogeneity and specialized computing in the form of reconfigurable accelerators [251]. This makes the use of FPGAs very attractive as they allow to scale-out resources by enabling distributed computing, and can be programmed to be network-capable processors implementing custom interconnects featuring low-latency communications without involving the CPU control [138].

First prototypes of FPGA accelerated HPC systems are already being designed and deployed. One example is the Alveo FPGA Cluster installed at ETH Zurich in the context of the Xilinx Adaptive Compute Clusters (XACC) initiative, using commodity hardware to support novel research in adaptive compute acceleration for HPC. Another example is the EU-H2020 EuroEXA Project, which has developed a HPC system prototype with custom hardware, adopting FPGA based accelerators for both computing and networking [138].

Consequently, as a future scenario we may expect next generations of HPC systems to be equipped with FPGA-based accelerators, probably alongside other accelerators, such as GPUs, being programmed with high level languages, possibly based on pragma directives, allowing to address several kind of different accelerators in a uniformed way [30].

In this context, application developers need to estimate the performance achievable on target FPGAs, to decide whether an application kernel is worth to be ported, or which FPGA better fits its computing requirements. At the same time, system architects and engineers need to estimate the performance of a single FPGA, to feed performance models to tune, balance and optimize the performance at system level [251].

Several research works have investigated FPGAs performance when used as hardware accelerators, mostly using synthetic benchmarks to estimate the bandwidth of off-chip memories [169, 240, 264], and OpenCL kernels to measure the FPGA computing performance [123, 156, 256].

In [240] is presented the *Shuhai* Verilog benchmark, used to characterize the performance of HBM and DDR off-chip memories embedded in the Xilinx Alveo U280. In [156] is presented an OpenCL implementation of the HPCChallenge Benchmark Suite, reporting the results for different FPGAs. In [256] is reported a C/HLS implementation of the HPCG Benchmark targeting FPGAs. Interestingly, in this case the Roofline Model has been used, but only to assess the optimization level of the specific application, with respect to theoretical estimations.

In fact the Roofline Model has already been used in the past to evaluate the performance of specific applications [170], being ported to FPGAs. But few works provide a generic application-independent extension of this model for these architectures, mainly due to the difficulty in defining the maximum compute performance for a reconfigurable device. A first comprehensive work extending the Roofline Model to FPGAs has been presented in [61], here authors focus mainly on aiding developers to explore the design space options. Building on the same principle, more recently, in [215] and in its extended version [214], a semi-automated performance optimization methodology based on the Roofline model for FPGAs has been proposed. In this case the authors, aim for a tool to explore the design space, while in our case we aim to provide a benchmarking tool.

The first work proposing a methodology for the performance analysis of FPGAs allowing to make Roofline plots and cross-architectural comparisons, has been reported in [166]. In this case, the

authors use OpenCL as programming language to provide mini-apps, such as SHOCL0, LINPACK and STREAM, to measure the computing performance and the memory bandwidth of the off-chip memory. Using OpenCL also the ERT benchmark has been reported to run on FPGAs in [175] and in its extension [174].

In [35] have been reported the first C/HLS benchmark tool able to provide empirical Roofline plots for FPGAs. Later extended to support the Xilinx Vitis workflow to allow for a wider adoption [36]. This tool, named FER (FPGA Empirical Roofline) [37], and available as Free Software [34], has been developed by INFN and the University of Ferrara, and it allows for application-agnostic performance assessment of FPGA based accelerators, aiming to a comprehensive machine characterization, allowing for cross-architectural comparisons and for performance estimations of generic HPC kernels on a given device. To this aim, FER is able to measure both the computing peak performance of FPGAs, and the bandwidths of on-chip and off-chip memories. It is based on the Roofline Model and it is implemented having at its core a directives annotated C/HLS kernel, with tunable operational intensity and hardware resources usage. Moreover, it relies on a theoretical model aiming to strictly link the performance results to the available hardware resources. The choice of C/HLS allows at the same time to expose to the users low level fine tuning knobs, as well as to use a high-level programming paradigm that can easily be used by the HPC user community for development and porting.

FER has been used to measure the double-, single- and half-precison floating-point performance, as well as fixed-point precision performance, of several FPGAs [37], but could also be easily adapted to measure the performance of deep learning specific operations using custom precision, in the context of the Roofline theoretical model, easily highlighting performance limits of different hardware devices.

## 6 COMPUTATIONAL MODELS FOR HPC APPLICATIONS

For the design of next generation HPC systems, in addition to tools and platforms suitable for the realization of hardware accelerators, appropriate models of computations are highly desirable. In fact, a computational model provides the designer with a proper level of abstraction from the low-level details, thus making possible to capture the main features mostly affecting speed performances and power consumption achievable on the target hardware platform. For this reason, several tools and computational models have been developed in the past with the objective of supporting efficiently the design of hardware accelerators for DL by two ways: estimating detailed insights of features and capabilities of the accelerators under design; and improving their behavior in terms of power consumption, performance, and resources requirements. In this section, we will explore some of the most popular computational models suitable, on the one hand, to investigate strengths, weaknesses, limits and bottlenecks of hardware accelerators, and, on the other hand, to fit the computational requirements by choosing the proper parallelism level, by minimizing the number of operations and by reducing the power consumption.

As discussed in the following, some of the explored computational models allow performing a theoretical analysis of new hardware architectures (e.g. tensor cores and PIM), whereas others are useful to approach the use of linear algebra inside HPC applications. Finally, approximate computing is analyzed as an emerging computational model to reduce computational delay and energy consumption of typical HPC workloads.

This Section furnish to application developers a thorough description of computational models exploitable to estimate the performance achievable on different implementation platforms, to decide whether a given hardware architecture is worth to be adopted, or which approximate computing approach better fits the computing requirements.

### 6.1 Theoretical Models of Computations

A *model of computation* is a theoretical framework for the design and analysis of efficient algorithms for a given computational architecture: the goal of a computational model is to abstract from the low-level details and to capture the main features that affect the most the performance on the target architecture. The Random Access Machine (RAM) model is the most common computational model and it has been a cornerstone of the history of computing: its main goal is to design efficient algorithms that minimize the number of CPU operations and to investigate the limits of computing. However, the RAM model does not efficiently capture the main features and bottlenecks of modern hardware. A well-known example is provided by memory hierarchy: the RAM model does not capture the memory bottleneck and the different times to access data in different positions of the memory. Therefore several works have addressed how to extend the RAM model to include such characteristics: the most notable is the *External Memory model* that has been widely used for designing and analyzing efficient algorithms and data structures that fully exploit the memory hierarchy (see e.g. [239]). Other examples have been provided by the several computational models that have been developed for parallel architectures, such as BSP, PRAM, LogP, and MapReduce. We refer to [26] for a general exposition of theoretical computational models for parallel and hierarchical architectures.

As already presented in this survey, there are several emerging technologies, like tensor core accelerators and processing-in-memory (PIM) architectures, that can potentially speed up computations. However, the process of algorithm design must take into account these technologies. From a theoretical perspective, suitable models of computations should be introduced for these new architectures. In this section, we briefly review the recent results on computational models for tensor cores accelerators and PIM architectures.

*6.1.1 Tensor cores accelerators.* The most important feature of tensor cores is the ability to efficiently perform matrix multiplications thanks to suitable hardware. While tensor cores have been widely used for the training of neural networks, a few works have applied these architectures in other computational domains. Works like [62, 150, 218] have studied how to expand the application domain of tensor core accelerators by targeting linear algebra and graph analytics. These works provide an experimental analysis of the proposed algorithms, but do not provide a theoretical analysis of their performance, nor introduce a computational model for tensor cores. The first work in this direction appeared in [56]. This work introduces a computational model capturing the features of tensor cores: the model captures the ability of the tensor core to efficiently perform dense matrix multiplication of fixed size and it is characterized by two parameters $m$ and $\ell$. Specifically, the model enriches a traditional RAM model with a hardware unit to perform a matrix multiplication of given size $\sqrt{m} \times \sqrt{m}$ in time $O(m + \ell)$ where $\ell$ is a latency cost. This model has been then used for analyzing the performance of algorithms for linear algebra, graph, and stencil algorithms, sparse matrix multiplication [136], and similarity search [12].

*6.1.2 Processing-in-memory architectures.* With the advent of commercial accelerators such as UPMEM, several results have been published on algorithms optimized for this class of accelerators, although they do not provide theoretical guarantees. For instance, [265] proposed algorithms for skyline computations, while [55] described data structures like linked lists, FIFO queues, and skip lists. These works provided an empirical evaluation of algorithms for PIMs and did not define a model of computation. The paper [148] presented a performance model for PIM with parameters for the latency of memory access by a CPU core, the latency of local memory access by a PIM core, and the latency of last-level cache access by a CPU core. To the best of our knowledge, the only result with a model of computation for PIM and with a theoretical analysis is provided in [126]. The

paper proposed a computational model for PIM inspired by the UPMEM architecture: the model combines a CPU side consisting of parallel cores with fast access to a small shared memory of size $M$ words, and a PIM side consisting of $P$ PIM modules, each with a core and a local memory of size $\Theta(n/P)$ words, $n$ denotes the input size of the problem. The model has then been used for designing a skip-list [126] and an index for skewed data [127] that exploit PIM systems.

*6.1.3 Future work.* During the project, we will investigate how to further extend the aforementioned computational models for emerging technologies. We will then use these models to design and theoretically analyze efficient algorithms and data structures that fully exploit such technologies. We will in particular address important primitives for machine learning and data analysis.

### 6.2 Linear Algebra, Tensors, Machine Learning/Deep Learning

*6.2.1 Linear Algebra algorithms.* Linear Algebra is and will continue to be at the heart of HPC applications for the foreseeable future. An immense amount of research has been devoted to the efficient implementation of linear algebra; among these efforts we can identify some that are more concerned with the computational models needed in approaching the use of linear algebra inside applications.

Among these trends we find the emergence of the so-called task-based runtime environment [10, 11, 42, 105, 184]; these systems provide a novel way to encode complex algorithms by specifying a set of dependencies among various building blocks. The programmer builds a DAG (directed acyclic graph) specifying for each node a kernel to be executed on a certain portion of the data, and connecting the nodes with directed arcs to specify an input-output relationship among the various computations. The end result is the increase in programmability of various kinds of complex linear algebra algorithms [95, 193, 205].

As we mentioned previously, linear algebra has elicited an immense amount of research work, due to its importance in providing application building blocks; and yet, there is a certain amount of disconnection between the users and programmers of applications, and the developers of libraries. The libraries encompass a body of knowledge on what constitute efficient implementations, but the users tend to rely ever more on environments that may or may not provide an optimal mapping from problem to function calls. As noted in the survey [191], the mapping problem itself is NP-complete, hence there is a need for further activity in this field to help users identify the best possible ways to frame the applications in ways that are conducive to exploitation of exascale resources.

One of the essential ingredients of modern HPC architectures is their heterogeneity; handling heterogeneity in the applications has been addressed e.g. by using the techniques in [41, 44, 88]; more work is definitely needed in enabling end-users to handle heterogeneity in a convenient and transparent way.

Two major trends in recent years have been quite visibile and relevant. The first trend is the emergence of variants of existing Krylov methods designed to reduce communications; these topics have been the subject of very intense research, since the reduction of communication is a necessary step in the full exploitation of exascale architectures. The communication avoiding developments have been spearheaded by the group of J. Demmel‘[43, 71, 155] on both conventional and accelerator architectures.

Another major trend which is connected to the emergence of accelerators is the usage of mixed-precision algorithms. This trend has achieved prominence because on many accelerator plaftorms, the speed of single and reduced precision arithmetic is significantly faster than that of the usual double precision computations; thus, the study of algorithms that can be formulated using mixed precision modes is a very attractive feature. To get an overview, see the papers [5, 90, 107]. In the same vein, we can look at the use of randomization in algorithms [167].

Many important software techniques have been implemented in the form of high performance libraries. The most well known ones include LAPACK and ScaLAPACK [74], Trilinos [106] and PETSc [24]. Sparse linear algebra libraries have long been developed by our group, and we have recently introduced new versions [63, 77], where we implement some among the most effective solver techniques available, i.e. algebraic multigrid preconditioners coupled with Krylov subspace solvers. Our recent research has been focused on the implementation of more effective strategies for building the multigrid hierarchy; our research program has been granted early access to the new Leonardo computational facility at CINECA, where we are actively exploring extreme scalability of multigrid construction based on graph matching.

Sparse linear algebra is central in many applications, including machine learning; for a general discussion of the use of sparse linear algebra in machine learning, see [78, 94, 177]; a discussion of the GPU applications can be found in [94]. An interesting facet is that the relationship is also active in the other direction, i.e. use of machine learning tecchniques in sparse linear algebra; see e.g. [225].

#### *6.2.2 Algorithmic Optimizations for CNN Acceleration.*

Deep Convolutional Neural Networks (CNNs) have been extensively used for processing images, sounds, and more generic sensor data, for detecting objects, patterns, and events [133, 226, 254]. Convolutional layers in CNNs are expensive in terms of compute and memory resources. There are many ways in which a convolution layer can be implemented. Without loss of generality, the methods discussed in this subsection refer to the case of single-image inference. Batched inferences can be considered an extension of the single-image case.

Typically, the convolution is implemented using a traditional sliding window approach across the activation data matrix, together with the application of a kernel function [206]. However, this type of computation in HPC systems is not efficient due to the irregularity of the data access pattern.

In order to reduce the number of floating point operations needed for computing the convolution, Fast Fourier Transform (FFT)-based implementations were proposed [152, 236]. The convolution is computed in the frequency domain as a Hadamard product (element-wise matrix multiplication), after Fourier transforming the activation data. After the product, results are transformed back in the frequency domain applying an inverse FFT. Even though FFT provides an asymptotically superior approach, the gap between the input feature map size and kernel size makes it often very inefficient as, for the computation to be performed, the kernel weights have to be padded to the size of the input image, incurring in a significant memory overhead, in particular when the kernels themselves are small [258].

Another type of computational transformation, particularly efficient when processing small kernels (size $\leq 3$), and that can be applied to convolutions in which the stride is 1, is the *Winograd* minimal filter algorithm [7, 242]. The Winograd convolution algorithm first divides the output activation matrix into tiles and computes each tile as $A^T[(Gg) \odot (B^T d)]$, where $g$ is the convolution filters and $d$ is the input activation matrix. $A$, $B$, and $G$ are transformation matrices, which are constants for a given value of tile size and convolution filter size. $\odot$ denotes the Hadamard product. The Winograd convolution reduces the number of multiplications and, as the matrix multiplication of smaller transformed matrices has more independent workloads, increases the thread-level parallelism. However, this comes at the cost of extra floating-point additions and the extra global memory accesses that are needed to implement the matrices' transformations. This process, for large convolution filters, may overwhelm the benefits of multiplication reduction [116].

Another common approach is to reshape and selectively duplicate parts of the original input activation data to create a lowered matrix [21, 46, 54]. This allows to implement the convolution as a multiplication between the newly generated matrix of input data and a properly arranged

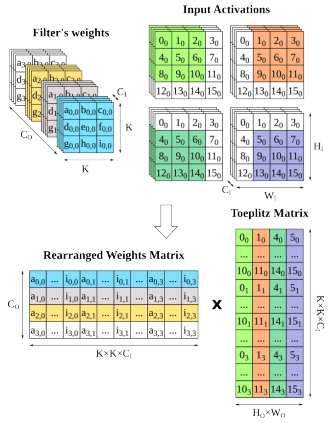


Fig. 9. *im2col* activation data packing.

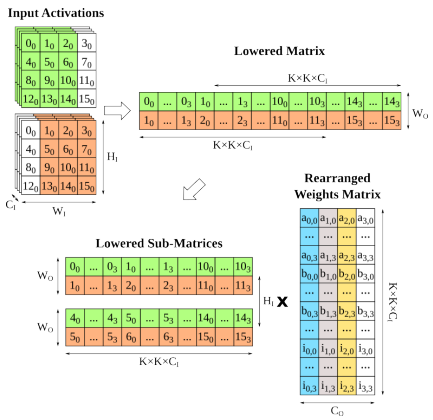


Fig. 10. *MEC* activation data packing.

matrix of kernel weights, and to leverage the highly optimized high performance matrix-matrix multiplication routines that can be found in Level 3 Basic Linear Algebra Subprogram (BLAS) libraries [73].

The first of these methods, the *image-to-column* (*im2col*) algorithm [46], transforms the input activation matrix into a Toeplitz matrix by unrolling overlapping patches of the input matrix into columns (Figure 9). In the dual approach, image-to-row (*im2row*), the lowered matrix is created by unrolling the patches into rows [21]. Both methods require an additional memory space of size $(K \times K \times C_I) \times (H_O \times W_O)$ for storing the lowered input matrix.

In the Memory-efficient Convolution (*MEC*) algorithm [54], differently then with the *im2row* algorithm, multiple rows of the activation matrix are lowered at once, by transforming $[H_I] \times [K] \times [C_I]$ submatrices into rows. As shown in Figure 10, the resulting lowered matrix is of size $W_O \times H_I \times K \times C_I$, i.e., $K$ smaller than the lowered matrix generated in the case of *im2row*. The convolution is computed by multiplying the weights matrix with $H_O$ submatrices of size $[W_O] \times [K \times K \times C_I]$, obtained by shifting, over the lowered matrix, the submatrix to the right by $s \times K$. *MEC* intuitively eliminates the vertical redundancy of the *im2row* approach, while recovering the information by shifting the submatrix by a constant interval [54].

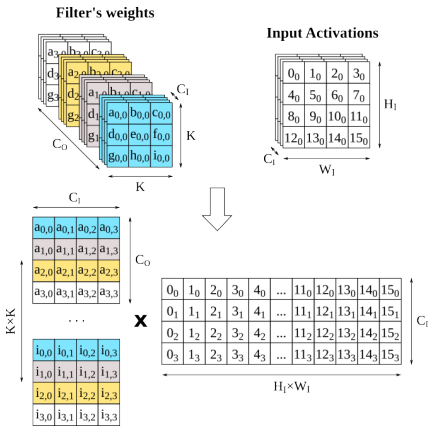


Fig. 11. *kn2row* activation data packing.

While enjoying the speed-up given in the execution by the use of architecture-optimized routines, these approaches suffer from the time penalty of implementing the bandwidth-bounded packing of the input matrix, as well as from the mismatch between the sizes of the matrices used in the calculation of the convolution and those for which traditional high-performance systems are optimized. Moreover, additional memory space is needed for storing the lowered matrices.

*Direct methods*, which do not involve the packing of matrices before the computation, have been also proposed. Vasudevan et al. [21] introduces the *kernel-to-row* (*kn2row*) algorithm to avoid data replication in the input, at the cost of increasing the size of the output. As shown in Figure 11, the convolution is computed as the sum of $K \times K$ separate $1 \times 1$ convolutions. Each $1 \times 1$ convolution is calculated by considering only one of the $K \times K$ kernel components at the time, and multiplying it with the input activation matrix, therefore by performing a matrix-matrix multiplication between the corresponding $[C_O] \times [C_I]$ weight matrix and the $[C_I] \times [H_I \times W_I]$ activation matrix. All the $K \times K$ separate $1 \times 1$ convolutions can be computed using a single matrix multiplication by reordering the filter matrix by laying out contiguously the $C_O$ channel data. This results in multiplying a $[K \times K \times C_O] \times [C_I]$ weight matrix to a $[C_I] \times [H_I \times W_I]$ activation matrix. The result matrix, of size $[K \times K \times C_O] \times [H_I \times W_I]$, is stored in memory at the end of the multiplication.

In order to obtain the desired $[C_O] \times [H_O \times W_O]$ output activation matrix, the results of the $K \times K$ separate $1 \times 1$ convolutions are added together by appropriately shifting the data vertically and/or horizontally, depending on the position of the relative weight with respect to the central kernel weight (i.e., top, bottom, left, right, and diagonal positions). Some of the results of the intermediate $1 \times 1$ convolutions are outside the boundaries of the final result matrix and they are discarded during the final sum. Filter weights are arranged in the desired position ahead of time.

In [21], the *kn2row* approach is modified by performing the shift-add operation at the end of the calculation of each separate $1 \times 1$ convolution. The needed temporary storage is reduced to $[2\delta + C_O] \times [H_I \times W_I]$, where $\delta$ is the number of extra rows in the result matrix needed to support the shifting of the result data.

By swapping the dimensions of the filter and of the input activation matrices to make $C_I$ the innermost dimension, it is possible to obtain the dual methods (*kernel-to-col*).

The High Performance Zero-Memory Overhead (*HPZMO*) Direct Convolutions approach [258] implements the convolution by reading the activation and filter weights' data directly from memory, without the need of additional memory space at the input or at the output. The *HPZMO* algorithm

is show in Algorithm 1, which is obtained by rearranging and optimizing the naive convolutional algorithm for taking into account its execution over multi-threaded Single-Instruction Multiple-Data (SIMD) architectures and an output-tiled approach, in which the partial results of the convolution of $W_{o,b}$ elements are accumulated into the register file. The HPZMO approach relies on the vector units of the computing architecture. The algorithm extracts parallelism in the output channel ($C_O$) dimension, which allows the sharing of the input data among threads/PEs for calculating different sets of output channels. Both input and output activation data, as well as the filter weights, are organized in the *channel-last* structure, and into blocks of $H \times W \times C_b$, where $C_b$ is a multiple of the SIMD vector length.

**Algorithm 1** Parallelized Direct Convolution Algorithm - HPZMO

**Input:** Activation I, Filter weights, Stride s = 1;
**Output:** Activation O;

```
for i ← 1 to C_O/C_{O,b} in Parallel do
  for k ← 1 to H_O do
    for l ← 1 to W_O/W_{O,b} do
      for m ← 1 to H_K do
        for n ← 1 to W_K do
          for j ← 1 to C_I do
            for ll ← 1 to W_{O,b} do
              for ii ← 1 to C_{O,b} do
                O_{i·C_{O,b}+ii,l·W_{O,b}+ll,k} += I_{j,l·W_{O,b}+ll+n,k+m} ×
                                                   F_{j,i·C_{O,b}+ii,n,m}
```

### 6.3 Parallel Patterns

This section will provide an overview of the methodology of *Structured Parallel Programming* (SPP), where complex parallel applications are provided as composition of few components properly combined and nested with each other. The general objective of this approach is to improve the productivity of parallel software, which should be implemented in an efficient manner and quite easily by a broad spectrum of programmers also including experts of application domains (which are not necessarily experts of concurrent/parallel programming). In the next part we review the idea of SPP, then we will describe existing research and industrial frameworks adopting this design methodology. Finally, we carefully consider `FastFlow` as a prototype framework for bringing the SPP approach in emerging application domains such as Data Stream Processing, Machine Learning and others.

*6.3.1 From Algorithmic Skeletons to Parallel Design Patterns.* SPP was originally made available through the *Algorithmic Skeletons* programming model [59], where parallelism can be expressed and orchestrated using structured compositions of predefined parallel components representing recurrent schemas of parallelism exploitation. Although predefined, such components are often parametrizable, e.g., in their parallelism degree (number of concurrent/parallel entities composing the component), and their regular structure in terms of interaction and synchronization allows performance metrics such as throughput and latency to be predicted using analytical cost models. Furthermore, the exposition of the complete structure of the parallel application allows the use of different heuristics when porting an application from one target architecture to a different one, so helping the so-called *performance portability* of parallel software. Example of such skeletons are

map, reduce, parallel-for, stencils (often applied in linear algebra problems), farm, pipeline, divide-and-conquer and iterative computations expressed, for example, by a macro dataflow engine [67]. Such skeletons are often provided as higher-order functions in functional programming languages, or as classes in object-oriented imperative programming languages that can be instantiated with the business logic code by hiding to the user all the low-level details of the underlying parallel implementation.

There is a certain difference between the original view of Algorithmic Skeletons and the more recent vision of *Parallel Design Patterns* [64]. The latter represents a renewed instantiation of the SPP programming approach, where instead of providing a predefined and somehow rigid set of skeletons, the programming framework provides some mechanisms and construct (a sort of intermediate abstractions) used by the application programmer to instantiate a parallelism pattern having specific properties. In this case, the programming burden is often higher than using pure skeletons, and the research is quite vivid in understanding which intermediate abstractions can be provided to application developer the help in building their own patterns easily and efficiently.

*6.3.2 Existing Research and Industrial Frameworks.* Over the years, a broad space of programming tools and frameworks adopting the SPP methodology have been provided both as research prototypes and as industrial tools. Historical tools fostering the Algorithmic Skeletons approach are Muesli, SkeTo, SkePU (which poses special emphasis on GPU accelerations), Lithium, Muskel, GrPPi, FastFlow, and many other as described in a survey on this topic [99]. More recently, with the renewed interest in SPP through the new design patterns perspective, several tools are now compliant with the pattern-based approach to parallel programming, such as Microsoft PPL, Intel TBB now subsumed by the OneAPI framework), OpenMP. Furthermore, in the field of Big Data processing tools, some open-source frameworks for data-intensive computing such as Apache Spark, Flink and Storm provide some dataflow abstractions and operators (e.g., working on streams in a structured manner) that might be considered as special instantiation of the parallel pattern idea to a specific application domain.

*6.3.3 The FastFlow Parallel Programming Framework.* `FastFlow` [16] is a C++ parallel programming environment based on the SPP methodology. It incorporates both Algorithmic Skeleton concepts as well as intermediate-level mechanisms and concepts useful to ease a quick prototyping of parallel patterns of different kinds. The layered structure of the framework is sketched in Fig. 12. The higher layer is represented by a set of high-level skeletons modeling built-in parallel patterns such as map, reduce, parallel-for and others. Each skeleton is implemented as a C++ class that can be created by providing the business logic code through functions or lambda objects.

The intermediate layer consists of a set of building blocks, which represent the available structures that can be used by programmers to instantiate their own parallel patterns that are not directly modeled by the set of built-in skeletons. Such building blocks allows the ease development of data-flow graphs of parallel activities, where nodes are implemented by dedicated threads, and communications are performed via shared memory by exchanging memory pointers. Building blocks can be classified in sequential blocks such as seq and combiners. The first wraps a sequential function applied in a streamed fashion on each input item in a single data-flow node. The second allows more seqs or other combiners to be incorporated into a data-flow node, so implementing the sequential composition or more functions applied on streams. Parallel building blocks allow sequential blocks, and recursively other parallel blocks, to be connected in complex but regular structures. The pipeline blocks model temporal parallelism, with each node working in parallel on different inputs. The farm block implements spatial parallelism, with the same function replicated in more nodes fed by an emitter node with scheduling purposes. Results are collected by a collector node multiplexing outputs into a single outcoming stream. Finally, the all-to-all block allows the

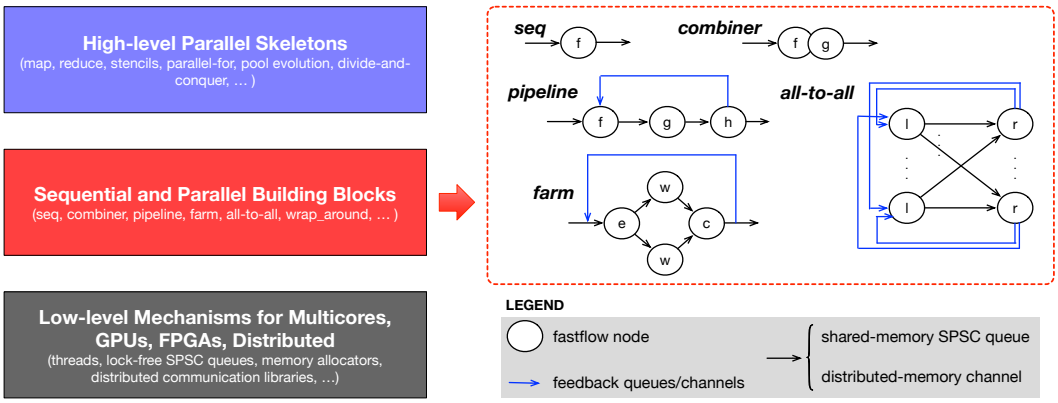


Fig. 12. Layered architecture of FastFlow.

complete connection between nodes (or blocks) in the left-hand set to the nodes (or blocks) of the right-and set. All the parallel blocks can be configured with the wrap-around modifier, which adds feedback connections creating controlled cycled in the data-flow graph. Such kind of feedbacks are of special importance to model parallel patterns incorporating iterative computations such as parallel graph and machine learning algorithms. In addition to the implementation of specific patterns, building blocks can be used by system programmers to develop new runtime systems for specific application domains. This idea has been recently applied to the WindFlow library for data stream processing [R], whose multi-core implementation is essentially based on formal compositions of the FastFlow's building blocks.

The last layer of the hierarchy is composed by a set of low-level mechanisms invisible to both domain experts using high-level parallel skeletons as well as by programmers using the building block abstractions. Examples of such mechanisms are lock-free queues used for pointer-passing between threads, which are available with different concurrency control approaches (e.g., based on busy-waiting synchronization or using exponential backoff and thread suspension).

### 6.4 Approximate Computing

In the last years, approximate computing has emerged as a powerful technique to reduce energy consumption and computational delay in error-resilient applications such as multimedia processing, deep learning (DL), digital signal processing, and wireless communications [17, 18, 101, 211]. Even though the basic principle is very simple, i.e. by relaxing the requirement of an exact computation, it is possible trading off the quality of the computation result for speed performances and energy dissipation, achieving the expected benefits is not trivial.

Nevertheless, approximate computing offers several opportunities to design efficient hardware accelerators for DL. For this reason, the contribution that the National Research Center can provide in this context is crucial. In fact, as summarized in Fig. 13, we are able to exploit approximate computing at different design levels: starting from the algorithm, passing through the architecture, up to the gate- and the transistor-level circuit topologies [119]. As an example, the approximation strategies presented in [40, 98, 204, 219, 220, 262] can be adopted at the algorithmic level to significantly reduce the complexity and the energy consumption of critical layers typically employed in DL models, such as the SoftMax and the convolutional layers interleaved by non-linear activations and down-sampling, at the expense of a reasonable accuracy loss.

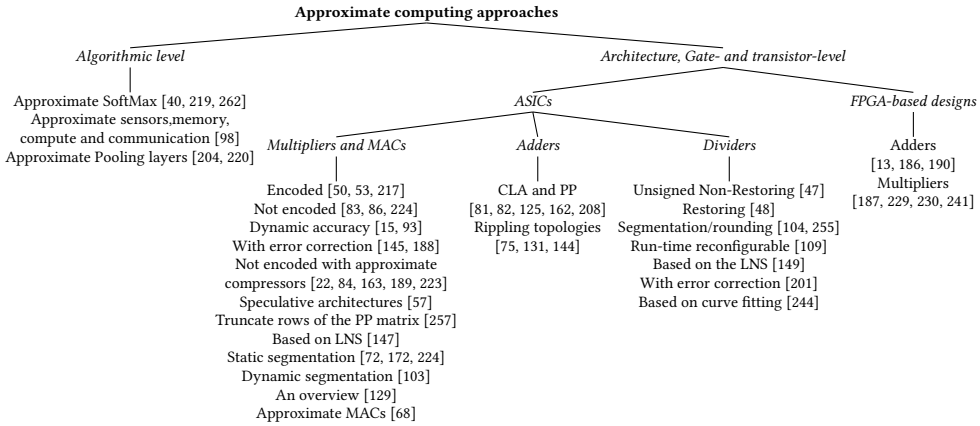


Fig. 13. Taxonomy of the approximate computing approaches discussed in Section 6.4

On the other hand, efficient solutions to exploit approximate computing at gate- and transistor-level have been recently demonstrated. To operate at a lower level of abstraction, we will focus our efforts on approximation strategies suitable to design approximate adders, multipliers and multiply-accumulate units for both Application Specific Integrated Circuits (ASICs) [50, 53, 83, 84, 86, 92, 93, 129, 217, 223, 224] and FPGA devices [186, 187, 229, 230, 241]. Such arithmetic operators receive a great deal of attention since they are the basic computational elements extensively used in DL models.

Typically, approximate computing is employed in arithmetic circuits at the architecture level by splitting the operands to be processed into sub-words. Then, some of the least significant bits are processed through an inaccurate circuit, whereas the remaining most significant bits are inputted to a precise circuit. Some strategies exploit static approximation that inflexibly sets the achieved accuracy at design time, while other solutions adopt dynamic approximation that allows tuning the quality target at run-time, thus leveraging the specificity of the data being processed and achieving graceful quality degradation.

Representative ASIC designs of approximate adders based on Carry-Look-Ahead (CLA) and Parallel-Prefix Architectures (PPA) are proposed in [81, 82, 125, 162, 208]. On the contrary, the adder topologies presented in [75, 131, 144] show how approximate logic can be exploited within carry-skip adders.

Despite the promising results achieved with ASIC approximate adders, their low-level optimizations cannot be applied directly on FPGAs. For this reason, efficient design approaches have been recently proposed to design approximate adders also within FPGA devices [13, 186, 190].

It is worth noting that, regardless of the adopted approach, both ASIC and FPGA-based approximate adders are designed taking into account that the propagation of the carry through a very long chain, over many bit positions, is an event with very low probability. Consequently, the carry propagation path can be broken in certain bit positions, thus reducing the computational delay, the power consumption and the amount of utilized hardware resources, but maintaining the quality of the final result still acceptable. On the basis of the target hardware implementation platform, we can contribute by identifying the proper architecture and approximation logic to comply with the desired speed performances and the available power budget.

In a similar way, several approximation techniques can be adopted in the design of multipliers. Some of the most efficient approaches approximate the partial product matrix compression step by

truncating some rows [257] or by involving inexact compressors [15, 22, 57, 68, 84, 92, 93, 145, 163, 188, 189, 223]. Others techniques use dynamic and static segmentation methods [72, 103, 172, 224]: the former downsizes the multiplier by selecting only a segment of the inputs starting from the leading one bit, whereas the latter processes only predefined portions of the multiplicands. Further examples of efficient approximate multipliers are those presented in [147] that take advantage of the logarithmic number system (LNS) [161].

Besides the above described approximation strategies oriented to ASIC designs, we want to exploit appropriate approaches to achieve high-performance designs of approximate multipliers also on FPGAs [187, 229, 230, 241]. Among the various solutions already known in literature, we will focus our attention on modular architectures that allow a n×n multiplier to be implemented utilizing four n/2×n/2 smaller sub-multipliers and approximate adders either for the partial product matrix compression step of each sub-multiplier or to sum the four computed sub-products, or both. As demonstrated in [187] such a technique can provide benefits also in ASIC designs.

As it is well known, HPC workloads often also require division operations that are even more complex than additions and multiplications. For this reason, efficient approaches suitable to approximate divisions are particularly desirable [47, 48, 104, 109, 149, 201, 244, 255]. Among such approximation strategies, two of the most representative are those based on approximate subtractors [47, 48] and on the signal segmentation [104].

Taking into account that several combinations of approximate adders, multipliers and dividers can be exploited for achieving the desired speed performances, power consumption, hardware resources requirements and overall accuracy, our contribution will be mainly concentrated towards the design of approximate computational modules for both ASIC and FPGA-based arithmetic circuits, and consequently on making available fast and energy efficient architectures that can serve as the basis for supporting the typical HPC workloads.

Definitely, we will exploit approximate computing at both algorithm and architecture levels, with the objectives of reducing on the one hand the computational complexity of layers typically employed within DL models, and, on the other hand, to optimize speed and power consumption introducing a reasonable accuracy loss.

## ACKNOWLEDGMENTS

This work has been (partially) supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - Next Generation EU (NGEU).

## REFERENCES

[1] 2013. AMBA® AXI™ and ACE™ Protocol Specification.
[2] 2018. Accellera IP-XACT working group: IP-XACT User Guide.
[3] 2018. IEEE Draft Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) . (February 2018), 1–100.
[4] 2020. GAP8 Auto-tiler Manual. https://greenwaves-technologies.com/manuals/BUILD/AUTOTILER/html/index.html.
[5] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, Jennifer Loe, Piotr Luszczek, Srikara Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry F Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M Tsai, and Ulrike Meier Yang. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 344–369. https://doi.org/10.1177/10943420211003313 arXiv:https://doi.org/10.1177/10943420211003313
[6] Mohamed S Abdelfattah, Łukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D Lane. 2020. Best of Both Worlds: AutoML Codesign of a CNN and Its Hardware Accelerator. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC '20)*. IEEE, 6 pages.

[7] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry. 2018. Accelerating CNN inference on FPGAs: A Survey. *CoRR* abs/1806.01683 (2018). arXiv:1806.01683 http://arxiv.org/abs/1806.01683

[8] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 358, 11 pages.

[9] Nicolas Bohm Agostini, Serena Curzel, Ankur Limaye, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, Antonino Tumeo, and Fabrizio Ferrandi. 2022. The SODA Approach: Leveraging High-Level Synthesis for Hardware/Software Co-Design and Hardware Specialization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*. 1359–1362.

[10] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. 2017. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems* (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2766064

[11] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. 2023. Task-Based Parallel Programming for Scalable Matrix Product Algorithms. *ACM Trans. Math. Softw.* (feb 2023). https://doi.org/10.1145/3583560 Just Accepted.

[12] Thomas D. Ahle and Francesco Silvestri. 2020. Similarity Search with Tensor Core Units. In *Proc. 13th Int. Conf. Similarity Search and Application (SISAP)*, Vol. 12440. 76–84.

[13] Waqar Ahmad, Berke Ayrancioglu, and Ilker Hamzaoglu. 2021. Low Error Efficient Approximate Adders for FPGAs. *IEEE Access* 9 (2021), 117232–117243. https://doi.org/10.1109/ACCESS.2021.3107370

[14] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. 2019. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*. 1–4.

[15] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2017. Dual-Quality 4:2 Compressors for Utilizing in Dynamic Accuracy Configurable Multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 4 (2017), 1352–1361. https://doi.org/10.1109/TVLSI.2016.2643003

[16] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *Fastflow: High-Level and Efficient Streaming on Multicore*. John Wiley & Sons, Ltd, Chapter 13, 261–280. https://doi.org/10.1002/9781119332015.ch13 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13

[17] Massimo Alioto. 2017. Energy-quality scalable adaptive VLSI circuits and systems beyond approximate computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 127–132. https://doi.org/10.23919/DATE.2017.7926970

[18] Massimo Alioto, Vivek De, and Andrea Marongiu. 2018. Guest Editorial for the Special Issue on Energy-Quality Scalable Circuits and Systems for Sensing and Computing: from Approximate, to Communication-Inspired and Learning-Based. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8 (08 2018), 1–1. https://doi.org/10.1109/JETCAS.2018.2865783

[19] G. Alonso. 2018. Research for practice: FPGAs in datacenters. *Commun. ACM* 61, 9 (2018), 48–49. https://doi.org/10.1145/3209275

[20] AMD-Xilinx. 2021. Vitis HLS LLVM 2021.2. https://github.com/Xilinx/HLS

[21] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2020. High-Performance Low-Memory Lowering: GEMM-based Algorithms for DNN Convolution. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 99–106. https://doi.org/10.1109/SBAC-PAD49847.2020.00024

[22] Mohammad Saeed Ansari, Honglan Jiang, Bruce F. Cockburn, and Jie Han. 2018. Low-Power Approximate Multipliers Using Encoded Partial Products and Approximate Compressors. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8, 3 (2018), 404–416. https://doi.org/10.1109/JETCAS.2018.2832204

[23] David F. Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Commun. ACM* 56, 4 (April 2013), 56–63. https://doi.org/10.1145/2436256.2436271

[24] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2023. PETSc Web page. https://petsc.org/. https://petsc.org/

[25] BDT. 2013. *Floating-point DSP Energy Efficiency on Altera 28 nm FPGAs.* Technical Report. Berkeley Design Technology Inc. http://www.altera.com/literature/wp/wp-01192-bdti-altera-fp-dsp-energy-efficiency.pdf An Independent Evaluation.

[26] Gianfranco Bilardi and Andrea Pietracaprina. 2011. *Models of Computation, Theoretical.* Springer US, Boston, MA, 1150–1158. https://doi.org/10.1007/978-0-387-09766-4_218

[27] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, et al. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (2018), 1–23.

[28] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[29] Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. 2022. Bridging Python to Silicon: The SODA Toolchain. *IEEE Micro* 42, 5 (2022), 78–88.

[30] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguade, and J. Labarta. 2018. Application Acceleration on FPGAs with OmpSs@FPGA. In *2018 International Conference on Field-Programmable Technology (FPT)*. 70–77. https://doi.org/10.1109/FPT.2018.00021

[31] F. Brosser, H. Y. Cheah, and S. A. Fahmy. 2013. Iterative floating point computation using FPGA DSP blocks. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–6. https://doi.org/10.1109/FPL.2013.6645531

[32] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. 2021. DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs. *IEEE Trans Comput.* (2021).

[33] Cadence. 2022. *Stratus High-Level Synthesis.* https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[34] E. Calore. 2020. https://baltig.infn.it/EuroEXA/FER

[35] E. Calore and S.F. Schifano. 2020. Energy-efficiency evaluation of FPGAs for floating-point intensive workloads. In *Parallel Computing is Everywhere (Advances in Parallel Computing, Vol. 36)*. 555–564. https://doi.org/10.3233/APC200085

[36] Enrico Calore and Sebastiano Fabio Schifano. 2021. Performance assessment of FPGAs as HPC accelerators using the FPGA Empirical Roofline. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 83–90. https://doi.org/10.1109/FPL53798.2021.00022

[37] Enrico Calore and Sebastiano Fabio Schifano. 2022. FER: A Benchmark for the Roofline Analysis of FPGA Based HPC Accelerators. *IEEE Access* 10 (2022), 94220–94234. https://doi.org/10.1109/ACCESS.2022.3203566

[38] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz S. Czajkowski, Stephen Dean Brown, and Jason Helge Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.* 13, 2 (2013), 24:1–24:27. https://doi.org/10.1145/2514740

[39] Shan Cao, Wei Deng, Zhenyi Bao, Chengbo Xue, Shugong Xu, and Shunqing Zhang. 2020. SimuNN: A Pre-RTL Inference, Simulation and Evaluation Framework for Neural Networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 10, 2 (2020), 217–230. https://doi.org/10.1109/JETCAS.2020.2993854

[40] Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Alberto Nannarelli, Marco Re, and Sergio Spanò. 2021. A pseudo-softmax function for hardware-based high speed image classification. *Scientific Reports* 11, 1 (28 Jul 2021), 15307. https://doi.org/10.1038/s41598-021-94691-7

[41] Valeria Cardellini, Salvatore Filippone, and Damian W. I. Rouson. 2014. Design Patterns for Sparse-Matrix Computations on Hybrid CPU/GPU Platforms. *Sci. Program.* 22, 1 (jan 2014), 1–19. https://doi.org/10.1155/2014/469753

[42] Rocío Carratalá-Sáez, Mathieu Faverge, Grégoire Pichon, Guillaume Sylvand, and Enrique S. Quintana-Ortí. 2020. Tiled Algorithms for Efficient Task-Parallel H-Matrix Solvers. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 757–766. https://doi.org/10.1109/IPDPSW50202.2020.00131

[43] Erin Carson, Nicholas Knight, and James Demmel. 2013. Avoiding Communication in Nonsymmetric Lanczos-Based Krylov Subspace Methods. *SIAM Journal on Scientific Computing* 35, 5 (2013), S42–S61. https://doi.org/10.1137/120881191 arXiv:https://doi.org/10.1137/120881191

[44] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[45] Victor Casamayor Pujol, Andrea Morichetta, Ilir Murturi, Praveen Kumar Donta, and Schahram Dustdar. 2023. Fundamental Research Challenges for Distributed Computing Continuum Systems. *Information* 14, 3 (2023).

[46] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High-Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition, Suvisoft*. 99–106.

https://doi.org/10.1109/SBAC-PAD49847.2020.00024

[47] Linbin Chen, Jie Han, Weiqiang Liu, and Fabrizio Lombardi. 2015. Design of Approximate Unsigned Integer Non-Restoring Divider for Inexact Computing. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI* (Pittsburgh, Pennsylvania, USA) *(GLSVLSI '15)*. Association for Computing Machinery, New York, NY, USA, 51–56. https://doi.org/10.1145/2742060.2742063

[48] Linbin Chen, Jie Han, Weiqiang Liu, and Fabrizio Lombardi. 2016. On the Design of Approximate Restoring Dividers for Error-Tolerant Applications. *IEEE Trans. Comput.* 65, 8 (2016), 2522–2533. https://doi.org/10.1109/TC.2015.2494005

[49] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *OSDI*.

[50] Yuan-Ho Chen and Tsin-Yuan Chang. 2012. A High-Accuracy Adaptive Conditional-Probability Estimator for Fixed-Width Booth Multipliers. *IEEE Transactions on Circuits and Systems I: Regular Papers* 59, 3 (2012), 594–603. https://doi.org/10.1109/TCSI.2011.2167275

[51] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring Your Own Codegen to Deep Learning Compiler. *arXiv preprint arXiv:2105.03215* (2021).

[52] Krishna Teja Chitty-Venkata and Arun K. Somani. 2023. Neural Architecture Search Survey: A Hardware Perspective. *Comput. Surveys* 55, 4, Article 78 (nov 2023), 36 pages.

[53] Kyung-Ju Cho, Kwang-Chul Lee, Jin-Gyun Chung, and K.K. Parhi. 2004. Design of low-error fixed-width modified booth multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 5 (2004), 522–531. https://doi.org/10.1109/TVLSI.2004.825853

[54] Minsik Cho and Daniel Brand. 2017. MEC: Memory-Efficient Convolution for Deep Neural Network. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 815–824.

[55] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: An Architecture-Aware Implementation. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (Phoenix, AZ, USA) *(SPAA '19)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/3323165.3323191

[56] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. 2021. Algorithm Design for Tensor Units. In *Euro-Par 2021: Parallel Processing*, Leonel Sousa, Nuno Roma, and Pedro Tomás (Eds.). Springer International Publishing, 353–367.

[57] Alessandro Cilardo, Davide De Caro, Nicola Petra, Francesco Caserta, Nicola Mazzocca, Ettore Napoli, and Antonio Giuseppe Maria Strollo. 2014. High Speed Speculative Multipliers Based on Speculative Carry-Save Tree. *IEEE Transactions on Circuits and Systems I: Regular Papers* 61, 12 (2014), 3426–3435. https://doi.org/10.1109/TCSI.2014.2337231

[58] CIRCT Developers. 2020. CIRCT / Circuit IR Compilers and Tools. https://circt.llvm.org/

[59] Murray Cole. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA.

[60] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. 2016. PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision. *Journal of Signal Processing Systems* 84, 3 (2016), 339–354.

[61] Bruno Da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. 2013. Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing* 2013 (2013). https://doi.org/10.1155/2013/428078

[62] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-Mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *Proc. Int. Conf. Supercomputing (ICS)*. 46–57.

[63] Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. 2021. AMG Preconditioners for Linear Solvers towards Extreme Scale. *SIAM Journal on Scientific Computing* 43, 5 (2021), S679–S703. https://doi.org/10.1137/20M134914X arXiv:https://doi.org/10.1137/20M134914X

[64] Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio González-Vélez, and Peter Kilpatrick. 2021. Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming. *Int. J. Parallel Program.* 49, 2 (2021), 177–198. https://doi.org/10.1007/s10766-020-00684-w

[65] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–27.

[66] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, et al. 2020. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *arXiv preprint arXiv:2010.08678* (2020).

[67] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. 2018. Transforming Loop Chains via Macro Dataflow Graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 265–277. https://doi.org/10.1145/3168832

[68] Davide De Caro, Nicola Petra, Antonio Giuseppe Maria Strollo, Fabio Tessitore, and Ettore Napoli. 2013. Fixed-Width Multipliers and Multipliers-Accumulators With Min-Max Approximation Error. *IEEE Transactions on Circuits and Systems I: Regular Papers* 60, 9 (2013), 2375–2388. https://doi.org/10.1109/TCSI.2013.2245252

[69] Florent de Dinechin et al. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (July 2011), 18–27.

[70] J. De Fine Licht, M. Besta, S. Meierhans, and T. Hoefler. 2021. Transformations of High-Level Synthesis Codes for High-Performance Computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2021), 1014–1029. https://doi.org/10.1109/TPDS.2020.3039409

[71] Jim Demmel. 2012. Communication avoiding algorithms. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 1942–2000. https://doi.org/10.1109/SC.Companion.2012.351

[72] Gennaro Di Meo, Gerardo Saggese, Antonio G. M. Strollo, and Davide De Caro. 2023. Design of Generalized Enhanced Static Segment Multiplier with Minimum Mean Square Error for Uniform and Nonuniform Input Distributions. *Electronics* 12, 2 (2023). https://doi.org/10.3390/electronics12020446

[73] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (mar 1990), 1–17. https://doi.org/10.1145/77626.79170

[74] Jack J. Dongarra and David W. Walker. 1995. Software Libraries for Linear Algebra Computations on High Performance Computers. *SIAM Rev.* 37, 2 (1995), 151–180. https://doi.org/10.1137/1037042 arXiv:https://doi.org/10.1137/1037042

[75] Kai Du, Peter Varman, and Kartik Mohanram. 2012. High performance reliable variable latency carry select addition. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1257–1262. https://doi.org/10.1109/DATE.2012.6176685

[76] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, J Ngadiuba, M Pierini, N Tran, and Z Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (2018), P07027.

[77] Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. 2023. Parallel Sparse Computation Toolkit. *Software Impacts* 15 (2023), 100463. https://doi.org/10.1016/j.simpa.2022.100463

[78] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *Proc. VLDB Endow.* 9, 12 (aug 2016), 960–971. https://doi.org/10.14778/2994509.2994515

[79] Ted Ennis. 2019. *Integrating optimized RTL Kernels into Accelerated Applications using Vitis Technology.* https://www.xilinx.com/developer/articles/Integrating-optimized-RTL-Kernels-into-Accelerated-Applications-using-Vitis.html

[80] F. A. Escobar, X. Chang, and C. Valderrama. 2016. Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2016), 600–612. https://doi.org/10.1109/TPDS.2015.2407896

[81] Darjn Esposito, Davide De Caro, Ettore Napoli, Nicola Petra, and Antonio Giuseppe Maria Strollo. 2015. Variable Latency Speculative Han-Carlson Adder. *IEEE Transactions on Circuits and Systems I: Regular Papers* 62, 5 (2015), 1353–1361. https://doi.org/10.1109/TCSI.2015.2403036

[82] Darjn Esposito, Davide De Caro, and Antonio Giuseppe Maria Strollo. 2016. Variable Latency Speculative Parallel Prefix Adders for Unsigned and Signed Operands. *IEEE Transactions on Circuits and Systems I: Regular Papers* 63, 8 (2016), 1200–1209. https://doi.org/10.1109/TCSI.2016.2564699

[83] Darjn Esposito, Antonio G. M. Strollo, and Massimo Alioto. 2017. Low-power approximate MAC unit. In *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*. 81–84. https://doi.org/10.1109/PRIME.2017.7974112

[84] Darjn Esposito, Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, and Nicola Petra. 2018. Approximate Multipliers Based on New Approximate Compressors. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4169–4182. https://doi.org/10.1109/TCSI.2018.2839266

[85] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. ACM, New York, NY, USA, 431–445. https://doi.org/10.1145/3437801.3441593

[86] Farzad Farshchi, Muhammad Saeed Abrishami, and Sied Mehdi Fakhraie. 2013. New approximate multiplier for low power digital signal processing. In *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013)*. 25–30. https://doi.org/10.1109/CADS.2013.6714233

[87] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *Proceedings*

*of the 58th ACM/IEEE Design Automation Conference (DAC)*. 1327–1330.

[88] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw.* 43, 4, Article 30 (jan 2017), 49 pages. https://doi.org/10.1145/3017994

[89] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. 2018. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–4.

[90] Goran Flegar, Hartwig Anzt, Terry Cojean, and Enrique S. Quintana-Ortí. 2021. Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software. *ACM Trans. Math. Softw.* 47, 2, Article 14 (apr 2021), 28 pages. https://doi.org/10.1145/3441850

[91] J. Fowers, K. Ovtcharov, M. K. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2019. Inside Project Brainwave's Cloud-Scale, Real-Time AI Processor. *IEEE Micro* 39, 3 (2019), 20–28. https://doi.org/10.1109/MM.2019.2910506

[92] Fabio Frustaci, Stefania Perri, Pasquale Corsonello, and Massimo Alioto. 2018. Energy-Quality Scalable Adders Based on Nonzeroing Bit Truncation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* PP (12 2018), 1–5. https://doi.org/10.1109/TVLSI.2018.2881326

[93] Fabio Frustaci, Stefania Perri, Pasquale Corsonello, and Massimo Alioto. 2020. Approximate Multipliers With Dynamic Truncation for Energy Reduction via Graceful Quality Degradation. *IEEE Transactions on Circuits and Systems II: Express Briefs* PP (06 2020), 1–1. https://doi.org/10.1109/TCSII.2020.2999131

[94] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00021

[95] Mark Gates, Asim YarKhan, Dalal Sukkari, Kadir Akbudak, Sebastien Cayrols, Daniel Bielich, Ahmad Abdelfattah, Mohammed Al Farhan, and Jack Dongarra. 2022. Portable and Efficient Dense Linear Algebra in the Beginning of the Exascale Era. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 36–46. https://doi.org/10.1109/P3HPC56579.2022.00009

[96] Lukas Geiger and Plumerai Team. 2020. Larq: An Open-Source Library for Training Binarized Neural Networks. *Journal of Open Source Software* 5, 45 (Jan. 2020), 1746.

[97] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[98] Soumendu Ghosh, Arnab Raha, and Vijay Raghunathan. 2020. Approximate inference systems (AxIS): end-to-end approximations for energy-efficient inference at the edge. 7–12. https://doi.org/10.1145/3370748.3406575

[99] Horacio González-Vélez and Mario Leyton. 2010. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Softw. Pract. Exper.* 40, 12 (nov 2010), 1135–1160.

[100] R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal* 45, 9 (1966), 1563–1581. https://doi.org/10.1002/j.1538-7305.1966.tb01709.x

[101] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. *Proceedings - 2013 18th IEEE European Test Symposium, ETS 2013*, 1–6. https://doi.org/10.1109/ETS.2013.6569370

[102] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:1510.00149 [cs.CV]

[103] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2015. DRUM: A Dynamic Range Unbiased Multiplier for approximate applications. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 418–425. https://doi.org/10.1109/ICCAD.2015.7372600

[104] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2016. A low-power dynamic divider for approximate applications. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2897965

[105] Thomas Herault, Joseph Schuchart, Edward F. Valeev, and George Bosilca. 2022. Composition of Algorithmic Building Blocks in Template Task Graphs. In *2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*. 26–38. https://doi.org/10.1109/PAW-ATM56565.2022.00008

[106] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.* 31, 3 (sep 2005), 397–423. https://doi.org/10.1145/1089014.1089021

[107] Nicholas J. Higham and Theo Mary. 2022. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31 (2022), 347–414. https://doi.org/10.1017/S0962492922000022

[108] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS '19)*. Curran Associates Inc., Red Hook, NY, USA, 10 pages.

[109] Mohsen Imani, Ricardo Garcia, Andrew Huang, and Tajana Rosing. 2019. CADE: Configurable Approximate Divider for Energy Efficiency. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 586–589. https://doi.org/10.23919/DATE.2019.8715112

[110] Ahmet Inci, Siri Garudanagiri Virupaksha, Aman Jain, Venkata Vivek Thallam, Ruizhou Ding, and Diana Marculescu. 2022. QADAM: Quantization-Aware DNN Accelerator Modeling for Pareto-Optimality. arXiv:2205.13045 [cs.AR]

[111] Ahmet Inci, Siri Garudanagiri Virupaksha, Aman Jain, Venkata Vivek Thallam, Ruizhou Ding, and Diana Marculescu. 2022. QAPPA: Quantization-Aware Power, Performance, and Area Modeling of DNN Accelerators. arXiv:2205.08648 [cs.AR]

[112] Intel. 2020. oneAPI Programming Model. https://www.oneapi.io.

[113] Intel. 2022. *Intel® High Level Synthesis Compiler Reference Manual*. https://www.intel.com/content/www/us/en/docs/programmable/683349/21-4/pro-edition-reference-manual.html

[114] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2020. Data Movement Is All You Need: A Case Study of Transformer Networks. *arXiv preprint arXiv:2007.00072* (2020).

[115] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-arithmetic-only Inference. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[116] Zhuoran Ji. 2019. HNMTP Conv: Optimize Convolution Algorithm for Single-Image Convolution Neural Network Inference on Mobile GPUs. *CoRR* abs/1909.02765 (2019). arXiv:1909.02765 http://arxiv.org/abs/1909.02765

[117] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML '18, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 2279–2288.

[118] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 1 (MLSys '19, Vol. 1)*. 1–13. https://proceedings.mlsys.org/paper_files/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

[119] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. 2020. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proc. IEEE* 108, 12 (2020), 2108–2135. https://doi.org/10.1109/JPROC.2020.3006451

[120] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. 2020. Hardware/software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4805–4815.

[121] Qing Jin, Linjie Yang, and Zhenyu Liao. 2019. Towards Efficient Training for Neural Network Quantization. *arXiv preprint arXiv:1912.10207* abs/1912.10207 (2019), 17 pages.

[122] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. 2020. Compiling ONNX Neural Network Models Using MLIR. https://doi.org/10.48550/arXiv.2008.08272 arXiv:2008.08272 [cs]

[123] Zheming Jin, Hal Finkel, Kazutomo Yoshii, and Franck Cappello. 2018. Evaluation of a Floating-Point Intensive Kernel on FPGA. In *Euro-Par 2017: Parallel Processing Workshops*. 664–675.

[124] Leonardo Rezende Juracy, Alexandre de Morais Amory, and Fernando Gehm Moraes. 2022. A Fast, Accurate, and Comprehensive PPA Estimation of Convolutional Hardware Accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 12 (2022), 5171–5184. https://doi.org/10.1109/TCSI.2022.3204932

[125] Andrew B. Kahng and Seokhyeong Kang. 2012. Accuracy-configurable adder for approximate arithmetic designs. In *DAC Design Automation Conference 2012*. 820–825. https://doi.org/10.1145/2228360.2228509

[126] Hongbo Kang, Phillip B. Gibbons, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/3409964.3461816

[127] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2022. PIM-tree: A Skew-resistant Index for Processing-in-Memory. *arXiv preprint arXiv:2211.10516* (2022).

[128] B. W. Kernighan and S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal* 49, 2 (1970), 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x

[129] Min Soo Kim, Alberto A. Del Barrio, HyunJin Kim, and Nader Bagherzadeh. 2022. The Effects of Approximate Multiplication on Convolutional Neural Networks. *IEEE Transactions on Emerging Topics in Computing* 10, 2 (2022), 904–916. https://doi.org/10.1109/TETC.2021.3050989

[130] Sunwoo Kim, Jooho Wang, Youngho Seo, Sanghun Lee, Yeji Park, Sungkyung Park, and Chester Sungchung Park. 2020. Transaction-level Model Simulator for Communication-Limited Accelerators. arXiv:2007.14897 [cs.AR]

[131] Yongtae Kim, Yong Zhang, and Peng Li. 2013. An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 130–137. https://doi.org/10.1109/ICCAD.2013.6691108

[132] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., 1097–1105.

[133] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (may 2017), 84–90. https://doi.org/10.1145/3065386

[134] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29.

[135] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963

[136] Paolo Sylos Labini, Massimo Bernaschi, Werner Nutt, Francesco Silvestri, and Flavio Vella. 2022. Blocking Sparse Matrices to Leverage Dense-Specific Multiplication. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 19–24. https://doi.org/10.1109/IA356718.2022.00009

[137] Corey Lammie, Wei Xiang, Bernabé Linares-Barranco, and Mostafa Rahimi Azghadi. 2022. MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems. *Neurocomputing* (2022). https://doi.org/10.1016/j.neucom.2022.02.043

[138] J. Lant, J. Navaridas, M. Luján, and J. Goodacre. 2020. Toward FPGA-Based HPC: Advancing Interconnect Technologies. *IEEE Micro* 40, 1 (2020), 25–34. https://doi.org/10.1109/MM.2019.2950655

[139] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.

[140] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[141] Marco Lattuada and Fabrizio Ferrandi. 2015. Code Transformations Based on Speculative SDC Scheduling. In *IEEE/ACM International Conference on Computer-Aided Design* (Austin, TX, USA) *(ICCAD '15)*. 71–77.

[142] Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. 1993. Chapter 9 Sequencing and scheduling: Algorithms and complexity. In *Logistics of Production and Inventory*. Handbooks in Operations Research and Management Science, Vol. 4. Elsevier, 445–522. https://doi.org/10.1016/S0927-0507(05)80189-6

[143] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2023. Green Carbon Footprint for Model Inference Serving via Exploiting Mixed-Quality Models and GPU Partitioning. *CoRR* abs/2304.09781 (2023). https://doi.org/10.48550/arXiv.2304.09781

[144] Li Li and Hai Zhou. 2014. On error modeling and analysis of approximate adders. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 511–518. https://doi.org/10.1109/ICCAD.2014.7001399

[145] Chia-Hao Lin and Ing-Chao Lin. 2013. High accuracy approximate multiplier with error correction. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 33–38. https://doi.org/10.1109/ICCD.2013.6657022

[146] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. 2019. ONNC: A Compilation Framework Connecting ONNX to Proprietary Deep Learning Accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 214–218. https://doi.org/10.1109/AICAS.2019.8771510

[147] Weiqiang Liu, Jiahua Xu, Danye Wang, Chenghua Wang, Paolo Montuschi, and Fabrizio Lombardi. 2018. Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 9 (2018), 2856–2868. https://doi.org/10.1109/TCSI.2018.2792902

[148] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/3087556.3087582

[149] Joshua Yung Lih Low and Ching Chuen Jong. 2013. Non-iterative high speed division computation based on Mitchell logarithmic method. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2219–2222. https://doi.org/10.1109/ISCAS.2013.6572317

[150] Tianjian Lu, Yi-Fan Chen, Blake Hechtman, Tao Wang, and John Anderson. 2021. Large-Scale Discrete Fourier Transform on TPUs. *IEEE Access* 9 (2021), 93422–93432. https://doi.org/10.1109/ACCESS.2021.3092312

[151] P. Sundararajan A. Coppola D. Pellerin W. Najjar R. Bruce M. Babst O. Pritchard P. Palazzari G. Kuzmanov M. Wirthlin, D. Poznanovic. 2008. OpenFPGA CoreLib core library interoperability effort. *Parallel Comput.* 34 (2008), 231–244.

[152] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast Training of Convolutional Networks through FFTs. arXiv:1312.5851 [cs.CV]

[153] Ruben Mayer and Hans-Arno Jacobsen. 2021. Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. *ACM Comput. Surv.* 53, 1 (2021), 37 pages. https://doi.org/10.1145/3363554

[154] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The Tensorflow Partitioning and Scheduling Problem: It's the Critical Path!. In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning (DIDL '17)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3154842.3154843

[155] Maryam MehriDehnavi, Yousef El-Kurdi, James Demmel, and Dennis Giannacopoulos. 2013. Communication-Avoiding Krylov Techniques on Graphic Processing Units. *IEEE Transactions on Magnetics* 49, 5 (2013), 1749–1752. https://doi.org/10.1109/TMAG.2013.2244861

[156] M. Meyer, T. Kenter, and C. Plessl. 2020. Evaluating FPGA Accelerator Performance with a Parameterized OpenCL Adaptation of Selected Benchmarks of the HPCChallenge Benchmark Suite. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 10–18. https://doi.org/10.1109/H2RC51942.2020.00007

[157] Microchip. 2020. Smart High-Level Synthesis Tool Suite. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler.

[158] M. Minutoli et al. 2015. Inter-procedural resource sharing in High Level Synthesis through function proxies. In *International Conference on Field Programmable Logic and Applications, FPL* (London, United Kingdom). 1–8.

[159] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A Hierarchical Model for Device Placement. In *Proceedings of Machine Learning and Systems (MLSys '18)*. 3 pages. https://mlsys.org/Conferences/doc/2018/150.pdf

[160] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML '17)*. JMLR.org, 2430–2439.

[161] John N. Mitchell. 1962. Computer Multiplication and Division Using Binary Logarithms. *IRE Transactions on Electronic Computers* EC-11, 4 (1962), 512–517. https://doi.org/10.1109/TEC.1962.5219391

[162] Debabrata Mohapatra, Vinay K. Chippa, Anand Raghunathan, and Kaushik Roy. 2011. Design of voltage-scalable meta-functions for approximate computing. In *2011 Design, Automation & Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763154

[163] Amir Momeni, Jie Han, Paolo Montuschi, and Fabrizio Lombardi. 2015. Design and Analysis of Approximate Compressors for Multiplication. *IEEE Trans. Comput.* 64, 4 (2015), 984–994. https://doi.org/10.1109/TC.2014.2308214

[164] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. 2019. A hardware–software blueprint for flexible deep learning specialization. *Micro* (2019).

[165] "msr fiddle". 2020. PipeDream: Pipeline Parallelism for DNN Training. https://github.com/msr-fiddle/pipedream

[166] Servesh Muralidharan, Kenneth O'Brien, and Christian Lalanne. 2015. A Semi-Automated Tool Flow for Roofline Anaylsis of OpenCL Kernels on Accelerators. In *First International Workshop on Heterogeneous High-performance Reconfigurable Computing*.

[167] Riley Murray, James Demmel, Michael W. Mahoney, N. Benjamin Erichson, Maksim Melnichenko, Osman Asif Malik, Laura Grigori, Piotr Luszczek, Michał Dereziński, Miles E. Lopes, Tianyu Liang, Hengrui Luo, and Jack Dongarra. 2023. Randomized Numerical Linear Algebra : A Perspective on the Field With an Eye to Software. arXiv:2302.11474 [math.NA]

[168] Francisco Muñoz-Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2020. STONNE: A Detailed Architectural Simulator for Flexible Neural Network Accelerators. arXiv:2006.07137 [eess.SP]

[169] S. W. Nabi and W. Vanderbauwhede. 2018. MP-STREAM: A Memory Performance Benchmark for Design Space Exploration on Heterogeneous HPC Devices. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 194–197. https://doi.org/10.1109/IPDPSW.2018.00036

[170] Kohei Nagasu, Kentaro Sano, Fumiya Kono, and Naohito Nakasato. 2017. FPGA-based tsunami simulation: Performance comparison with GPUs, and roofline model for scalability analysis. *J. Parallel and Distrib. Comput.* 106 (2017), 153–169. https://doi.org/10.1016/j.jpdc.2016.12.015

[171] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (Oct 2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673

[172] Srinivasan Narayanamoorthy, Hadi Asghari Moghaddam, Zhenhong Liu, Taejoon Park, and Nam Sung Kim. 2015. Energy-Efficient Approximate Multiplication for Digital Signal Processing and Classification Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 6 (2015), 1180–1184. https://doi.org/10.1109/TVLSI.2014.2333366

[173] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359646

[174] Tan Nguyen, Colin MacLean, Marco Siracusa, Douglas Doerfler, Nicholas J. Wright, and Samuel Williams. 2021. FPGA-based HPC accelerators: An evaluation on performance and energy efficiency. *Concurrency and Computation: Practice and Experience* n/a, n/a (2021), e6570. https://doi.org/10.1002/cpe.6570

[175] T. Nguyen, S. Williams, M. Siracusa, C. MacLean, D. Doerfler, and N. J. Wright. 2020. The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 8–19. https://doi.org/10.1109/PMBS51919.2020.00007

[176] Mostafa W. Numan, Braden J. Phillips, Gavin S. Puddy, and Katrina Falkner. 2020. Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains. *IEEE Access* 8 (2020), 174692–174722.

[177] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. 2015. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 109–116. https://doi.org/10.1109/CASES.2015.7324551

[178] NVIDIA. 2022. NVIDIA Deep Learning Accelerator. http://nvdla.org/.

[179] "Alibaba Group PAI". 2020. DAPPLE. https://github.com/AlibabaPAI/DAPPLE

[180] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2020. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. In *Proceedings of the 8th International Conference on Learning Representations (ICLR '20)*. 24 pages.

[181] Christos H. Papadimitriou and Mihalis Yannakakis. 1990. Towards an Architecture-Independent Analysis of Parallel Algorithms. *SIAM J. Comput.* 19, 2 (1990), 322–328. https://doi.org/10.1137/0219021

[182] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[183] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[184] Yu Pei, George Bosilca, and Jack Dongarra. 2022. Sequential Task Flow Runtime Model Improvements and Limitations. In *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. 1–8. https://doi.org/10.1109/ROSS56639.2022.00009

[185] Xiaochen Peng, Shanshi Huang, Yandong Luo, Xiaoyu Sun, and Shimeng Yu. 2019. DNN+NeuroSim: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators with Versatile Device Technologies. In *2019 IEEE International Electron Devices Meeting (IEDM)*. 32.5.1–32.5.4. https://doi.org/10.1109/IEDM19573.2019.8993491

[186] Stefania Perri, Fanny Spagnolo, Fabio Frustaci, and Pasquale Corsonello. 2020. Efficient Approximate Adders for FPGA-Based Data-Paths. *Electronics* 9, 9 (2020). https://doi.org/10.3390/electronics9091529

[187] Stefania Perri, Fanny Spagnolo, Fabio Frustaci, and Pasquale Corsonello. 2022. Designing Energy-Efficient Approximate Multipliers. *Journal of Low Power Electronics and Applications* 12, 4 (2022). https://doi.org/10.3390/jlpea12040049

[188] Nicola Petra, Davide De Caro, Valeria Garofalo, Ettore Napoli, and Antonio G. M. Strollo. 2010. Truncated Binary Multipliers With Variable Correction and Minimum Mean Square Error. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 6 (2010), 1312–1325. https://doi.org/10.1109/TCSI.2009.2033536

[189] Nicola Petra, Davide De Caro, Valeria Garofalo, Ettore Napoli, and Antonio Giuseppe Maria Strollo. 2011. Design of Fixed-Width Multipliers With Linear Compensation Function. *IEEE Transactions on Circuits and Systems I: Regular Papers* 58, 5 (2011), 947–960. https://doi.org/10.1109/TCSI.2010.2090572

[190] Bharath Srinivas Prabakaran, Semeen Rehman, Muhammad Abdullah Hanif, Salim Ullah, Ghazal Mazaheri, Akash Kumar, and Muhammad Shafique. 2018. DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 917–920.

https://doi.org/10.23919/DATE.2018.8342140

[191] Christos Psarras, Henrik Barthels, and Paolo Bientinesi. 2022. The Linear Algebra Mapping Problem. Current State of Linear Algebra Languages and Libraries. *ACM Trans. Math. Softw.* 48, 3, Article 26 (sep 2022), 30 pages. https://doi.org/10.1145/3549935

[192] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22. https://doi.org/10.1109/MM.2015.42

[193] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. 2009. Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism. *ACM Trans. Math. Softw.* 36, 3, Article 14 (jul 2009), 26 pages. https://doi.org/10.1145/1527286.1527288

[194] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[195] Daniel Reiser, Marc Reichenbach, Tommaso Rizzi, Andrea Baroni, Markus Fritscher, Christian Wenger, Cristian Zambelli, and Davide Bertozzi. 2023. Technology-Aware Drift Resilience Analysis of RRAM Crossbar Array Configurations. In *2023 IEEE NEWCAS*. in press.

[196] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 58–68. https://doi.org/10.1145/3211346.3211348

[197] B. Ronak and S. A. Fahmy. 2016. Mapping for Maximum Performance on FPGA DSP Blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 4 (April 2016), 573–585. https://doi.org/10.1109/TCAD.2015.2474363

[198] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2019. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv:1805.00907 [cs]* (April 2019). arXiv:1805.00907 [cs]

[199] Enrico Russo, Maurizio Palesi, Salvatore Monteleone, Davide Patti, Giuseppe Ascia, and Vincenzo Catania. 2021. LAMBDA: An Open Framework for Deep Neural Network Accelerators Simulation. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 161–166. https://doi.org/10.1109/PerComWorkshops51409.2021.9431078

[200] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2023. Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources. *ACM Trans. Auton. Adapt. Syst.* (2023). https://doi.org/10.1145/3597435

[201] Hassaan Saadat, Haris Javaid, and Sri Parameswaran. 2019. Approximate Integer and Floating-Point Dividers with Near-Zero Error Bias. 1–6. https://doi.org/10.1145/3316781.3317773

[202] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A systematic methodology for characterizing scalability of DNN accelerators using SCALE-sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 58–68.

[203] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator Simulator. *arXiv preprint arXiv:1811.02883* (2018).

[204] Aseem Sayal, Shirin Fathima, SS Nibhanupudi, and Jaydeep Kulkarni. 2020. COMPAC: Compressed Time-Domain, Pooling-Aware Convolution CNN Engine With Reduced Data Movement for Energy-Efficient AI Computing. *IEEE Journal of Solid-State Circuits* PP (12 2020), 1–1. https://doi.org/10.1109/JSSC.2020.3041502

[205] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. 2016. Parallel Matrix Multiplication: A Systematic Journey. *SIAM J. Sci. Comput.* 38, 6 (jan 2016), C748–C781. https://doi.org/10.1137/140993478

[206] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. 2014. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. arXiv:1312.6229 [cs.CV]

[207] Seyed Hossein Hashemi Shadmehri, Ali BanaGozar, Mehdi Kamal, Sander Stuijk, Ali Afzali-Kusha, Massoud Pedram, and Henk Corporaal. 2022. SySCIM: SystemC-AMS Simulation of Memristive Computation In-Memory. In *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1467–1472. https://doi.org/10.23919/DATE54114.2022.9774749

[208] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. 2015. A low latency generic accuracy configurable adder. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744778

[209] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 97–108. https://doi.org/10.1109/ISCA.2014.6853196

[210] A. Shawahna, S. M. Sait, and A. El-Maleh. 2019. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access* 7 (2019), 7823–7859. https://doi.org/10.1109/ACCESS.2018.2890150

[211] Doochul Shin and Sandeep K. Gupta. 2010. Approximate Logic Synthesis for Error Tolerant Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Dresden, Germany) *(DATE '10)*. European Design and Automation Association, Leuven, BEL, 957–960.

[212] David B. Shmoys and Éva Tardos. 1993. An Approximation Algorithm for the Generalized Assignment Problem. *Mathematical Programming* 62, 1 (1993), 461–474. https://doi.org/10.1007/BF01585178

[213] Siemens. 2022. *Catapult C++/Systemc Synthesis*. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/

[214] Marco Siracusa, Emanuele Delsozzo, Marco Rabozzi, Lorenzo Di Tucci, Samuel Williams, Donatella Sciuto, and Marco Domenico Santambrogio. 2021. A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model. *IEEE Trans. Comput.* (2021), 1–1. https://doi.org/10.1109/TC.2021.3111761

[215] M. Siracusa, M. Rabozzi, E. Del Sozzo, L. Di Tucci, S. Williams, and M. D. Santambrogio. 2020. A CAD-based methodology to optimize HLS code via the Roofline model. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1145/3400302.3415730

[216] Martin Skutella and Gerhard J. Woeginger. 1999. A PTAS for Minimizing the Weighted Sum of Job Completion Times on Parallel Machines. In *Proceedings of the 31 Annual ACM Symposium on Theory of Computing (STOC '99)*. ACM, New York, NY, USA, 400–407. https://doi.org/10.1145/301250.301356

[217] Min-An Song, Lan-Da Van, and Sy-Yen Kuo. 2007. Adaptive Low-Error Fixed-Width Booth Multipliers. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E90-A (06 2007). https://doi.org/10.1093/ietfec/e90-a.6.1180

[218] A. Sorna, X. Cheng, E. D'Azevedo, K. Won, and S. Tomov. 2018. Optimizing the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware. In *Proc. 25th Int. Conf. on High Performance Computing Workshops (HiPCW)*. 3–7.

[219] Fanny Spagnolo, Stefania Perri, and Pasquale Corsonello. 2022. Aggressive Approximation of the SoftMax Function for Power-Efficient Hardware Implementations. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 3 (2022), 1652–1656. https://doi.org/10.1109/TCSII.2021.3120495

[220] Fanny Spagnolo, Stefania Perri, and Pasquale Corsonello. 2022. Approximate Down-Sampling Strategy for Power-Constrained Intelligent Systems. *IEEE Access* 10 (2022), 7073–7081. https://doi.org/10.1109/ACCESS.2022.3142292

[221] ST Microelectronics. 2017. *X-CUBE-AI*. https://www.st.com/en/embedded-software/x-cube-ai.html

[222] Leon Stok. 1994. Data path synthesis. *Integration* 18, 1 (1994), 1–71.

[223] Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, Nicola Petra, and Gennaro Di Meo. 2020. Comparison and Extension of Approximate 4-2 Compressors for Low-Power Approximate Multipliers. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 9 (2020), 3021–3034. https://doi.org/10.1109/TCSI.2020.2988353

[224] Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, Nicola Petra, Gerardo Saggese, and Gennaro Di Meo. 2022. Approximate Multipliers Using Static Segmentation: Error Analysis and Improvements. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 6 (2022), 2449–2462. https://doi.org/10.1109/TCSI.2022.3152921

[225] Christodoulos Stylianou and Michele Weiland. 2023. Optimizing Sparse Linear Algebra Through Automatic Format Selection and Machine Learning. arXiv:2303.05098 [cs.LG]

[226] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014). arXiv:1409.4842 http://arxiv.org/abs/1409.4842

[227] T. Tang and Y. Xie. 2018. MLPAT: A power area timing modeling framework for machine learning accelerators. In *EEE International Workshop on Domain Specific System Architecture (DOSSA)*.

[228] Frederick Tung and Greg Mori. 2018. CLIP-Q: Deep Network Compression Learning by In-parallel Pruning-Quantization. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 7873–7882.

[229] Salim Ullah, Semeen Rehman, Muhammad Shafique, and Akash Kumar. 2022. High-Performance Accurate and Approximate Multipliers for FPGA-Based Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 2 (2022), 211–224. https://doi.org/10.1109/TCAD.2021.3056337

[230] Salim Ullah, Hendrik Schmidl, Siva Satyendra Sahoo, Semeen Rehman, and Akash Kumar. 2021. Area-Optimized Accurate and Approximate Softcore Signed Multiplier Architectures. *IEEE Trans. Comput.* 70, 3 (2021), 384–392. https://doi.org/10.1109/TC.2020.2988404

[231] Mike Urbach and Morten B Petersen. 2022. HLS from PyTorch to System Verilog with MLIR and CIRCT. 2nd Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE).

[232] Josse Van Delm, Maarten Vandersteegen, Alessio Burrello, Giuseppe Maria Sarda, Francesco Conti, Daniele Jahier Pagliari, Luca Benini, and Marian Verhelst. 2023. HTVM: Efficient Neural Network Deployment On Heterogeneous TinyML Platforms. In *Proceedings of the 2023 Conference & Exhibition on Design, Automation & Test in Europe*. Antwerp.

[233] Ben van Werkhoven, Willem Jan Palenstijn, and Alessio Sclocco. 2020. Lessons Learned in a Decade of Research Software Engineering GPU Applications. In *Computational Science – ICCS 2020*. Springer, Cham, 399–412.

[234] Wim Vanderbauwhede and Khaled Benkrid. 2013. *High-performance computing using FPGAs*. Vol. 3. Springer. https://doi.org/10.1007/978-1-4614-1791-0

[235] Tim Vanevenhoven. 2011. *High-Level Implementation of Bit- and Cycle-Accurate Floating-Point DSP Algorithms with Xilinx FPGAs*. Technical Report. Xilinx. White Paper: 7 Series FPGAs.

[236] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. arXiv:1412.7580 [cs.LG]

[237] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv:1802.04730 [cs]* (June 2018). arXiv:1802.04730 [cs]

[238] M. Véstias and H. Neto. 2014. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–6. https://doi.org/10.1109/FPL.2014.6927483

[239] Jeffrey Scott Vitter. 2001. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.* 33, 2 (jun 2001), 209–271.

[240] Z. Wang, H. Huang, J. Zhang, and G. Alonso. 2020. Shuhai: Benchmarking High Bandwidth Memory On FPGAs. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 111–119. https://doi.org/10.1109/FCCM48280.2020.00024

[241] Haroon Waris, Chenghua Wang, Weiqiang Liu, and Fabrizio Lombardi. 2021. AxBMs: Approximate Radix-8 Booth Multipliers for High-Performance FPGA-Based Accelerators. *IEEE Transactions on Circuits and Systems II: Express Briefs* 68, 5 (2021), 1566–1570. https://doi.org/10.1109/TCSII.2021.3065333

[242] Shmuel Winograd. 1980. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611970364 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611970364

[243] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga Behram, Jinshi Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin S. Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim M. Hazelwood. 2022. Sustainable AI: Environmental Implications, Challenges and Opportunities. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022*.

[244] Lei Wu and Ching Chuen Jong. 2015. A curve fitting approach for non-iterative divider design with accuracy and performance trade-off. In *2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS)*. 1–4. https://doi.org/10.1109/NEWCAS.2015.7182097

[245] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942149

[246] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Xiling Yin, Wenqin Huangfu, Pai-Yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, and Huazhong Yang. 2016. MNSIM: Simulation platform for memristor-based neuromorphic computing system. In *2016 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 469–474.

[247] Xilinx Inc. 2022. *Vitis High-Level Synthesis User Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf

[248] Xilinx Inc. 2022. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator. UG994 (v2022.2)*. https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug994-vivado-ip-subsystems.pdf

[249] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–383.

[250] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 369–383. https://doi.org/10.1145/3373376.3378514

[251] R. Yasudo, J. Coutinho, A. Varbanescu, W. Luk, H. Amano, and T. Becker. 2018. Performance Estimation for Exascale Reconfigurable Dataflow Platforms. In *2018 International Conference on Field-Programmable Technology (FPT)*. 314–317.

https://doi.org/10.1109/FPT.2018.00062

[252] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.

[253] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*. 1355–1358.

[254] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *Computer Vision – ECCV 2014*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer International Publishing, Cham, 818–833.

[255] Reza Zendegani, Mehdi Kamal, Arash Fayyazi, Ali Afzali-Kusha, Saeed Safari, and Massoud Pedram. 2016. SEERAD: A high speed yet energy-efficient rounding-based approximate divider. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1481–1484.

[256] Alberto Zeni, Kenneth O'Brien, Michaela Blott, and Marco D. Santambrogio. 2021. Optimized Implementation of the HPCG Benchmark on Reconfigurable Hardware. In *Euro-Par 2021: Parallel Processing*, Leonel Sousa, Nuno Roma, and Pedro Tomás (Eds.). 616–630. https://doi.org/10.1007/978-3-030-85665-6_38

[257] Georgios Zervakis, Kostas Tsoumanis, Sotirios Xydis, Dimitrios Soudris, and Kiamal Pekmestzi. 2016. Design-Efficient Approximate Multiplication Circuits Through Partial Product Perforation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 10 (2016), 3105–3117. https://doi.org/10.1109/TVLSI.2016.2535398

[258] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High Performance Zero-Memory Overhead Direct Convolutions. *CoRR* abs/1809.10170 (2018). arXiv:1809.10170 http://arxiv.org/abs/1809.10170

[259] Xiaofan Zhang, Hanchen Ye, and Deming Chen. 2021. Being-ahead: Benchmarking and Exploring Accelerators for Hardware-Efficient AI Deployment. arXiv:2104.02251 [cs.AR]

[260] Yang Zhao, Chaojian Li, Yue Wang, Pengfei Xu, Yongan Zhang, and Yingyan Lin. 2021. DNN-Chip Predictor: An Analytical Performance Predictor for DNN Accelerators with Various Dataflows and Hardware Architectures. arXiv:2002.11270 [cs.LG]

[261] Yanqi Zhou, Sudip Roy, AmirAli Abdolrashidi, Daniel Lin-Kit Wong, Peter C. Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, and James Laudon. 2019. GDP: Generalized Device Placement for Dataflow Graphs. *CoRR* abs/1910.01578 (2019), 11 pages. http://arxiv.org/abs/1910.01578

[262] Danyang Zhu, Siyuan Lu, Meiqi Wang, Jun Lin, and Zhongfeng Wang. 2020. Efficient Precision-Adjustable Architecture for Softmax Function in Deep Learning. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67 (2020), 3382–3386.

[263] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 409–420. https://doi.org/10.1109/SC.2016.34

[264] Hamid Reza Zohouri and Satoshi Matsuoka. 2019. The memory controller wall: Benchmarking the intel FPGA SDK for OpenCL memory interface. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 11–18. https://doi.org/10.1109/H2RC49586.2019.00007

[265] Vasileios Zois, Divya Gupta, Vassilis J. Tsotras, Walid A. Najjar, and Jean-Francois Roy. 2018. Massively Parallel Skyline Computation for Processing-in-Memory Architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3243176.3243187