SCRATCH: An End-to-End Application-Aware Soft-GPGPU Architecture and Trimming Tool

Pedro Duarte
Instituto de Telecomunicações
Dept. of Electrical and Computer Eng.
University of Coimbra, Portugal
pedro.duarte@student.uc.pt

Pedro Tomas
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
pedro.tomas@inesc-id.pt

Gabriel Falcao
Instituto de Telecomunicações
Dept. of Electrical and Computer Eng.
University of Coimbra, Portugal
gff@co.it.pt

ABSTRACT

Applying advanced signal processing and artificial intelligence algorithms is often constrained by power and energy consumption limitations, in high performance and embedded, cyber-physical and super-computing devices and systems. Although Graphics Processing Units (GPUs) helped to mitigate the throughput-per-Watt performance problem in many compute-intensive applications, dealing more efficiently with the autonomy requirements of intelligent systems demands power-oriented customized architectures that are specially tuned for each application, preferably without manual redesign of the entire hardware and capable of supporting legacy code. Hence, this work proposes a new SCRATCH framework that aims at automatically identifying the specific requirements of each application kernel, regarding instruction set and computing unit demands, allowing for the generation of application-specific and FPGA-implementable trimmed-down GPU-inspired architectures. The work is based on an improved version of the original MIAOW system (here named MIAOW2.0), which is herein extended to support a set of 156 instructions and enhanced to provide a fast prefetch memory system and a dual-clock domain. Experimental results with 17 highly relevant benchmarks, using integer and floating-point arithmetic, demonstrate that we have been able to achieve an average of 140× speedup and 115× higher energy-efficiency levels (instructions-per-Joule) when compared to the original MIAOW system, and a 2.4× speedup and 2.1× energy-efficiency gains compared against our optimized version without pruning.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; • **Hardware** → *Hardware-software codesign*; • **Computing methodologies** → *Machine learning*;

KEYWORDS

Soft-GPGPU, Architecture trimming, Low-power, FPGA customization

MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123953

ACM Reference format:
Pedro Duarte, Pedro Tomas, and Gabriel Falcao. 2017. SCRATCH: An End-to-End Application-Aware Soft-GPGPU Architecture and Trimming Tool. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123953

1 INTRODUCTION

The evolution of General-Purpose Processing on GPU (GPGPU) has been aided by the emergence of parallel programming frameworks, especially the Compute Unified Device Architecture (CUDA) [8, 27] and Open Computing Language (OpenCL) [6, 13]. These tools allow programmers to easily use the hardware resources of massively parallel processors for their applications, processing large amounts of data in relatively shorter periods of time when compared to previous Central Processing Unit (CPU)-based architectures. However, although GPUs provide high throughput performance, off-the-shelf devices demand high power levels to operate (~200W to ~300W per device) and have fixed designs that cannot be adapted towards the specific needs of the target applications.

Application-specific architectures can be designed and implemented in reconfigurable fabric and can be tailored to maximize the throughput-per-Watt performance [9, 25]. However, such fully customizable approaches often require a profound manual architectural redesign, even at the presence of minimal algorithmic changes. Moreover, such designs do not often deal well with legacy code, which can only be solved by constraining the efficiency of the design. Also, they involve a substantial design effort–either through manual register-transfer level (RTL), or using specialized OpenCL-to-hardware translation [3, 42] and bitstream completion [10, 12, 20, 24] tools–that many application developers do not have or are unwilling to take.

Recently, research efforts have been made in the development of general-purpose parallel and programmable architectures on reconfigurable fabric, often referred to as soft-general-purpose-GPU (soft-GPGPU), either by relying on existing commercial Instruction-Set Architectures (ISAs), or by developing special-purpose ones. In particular, Flexgrip [4], based on NVIDIA's G80 architecture, supports the Single-Instruction Multiple-Thread (SIMT) model and the utilization of code binaries generated directly from unmodified NVIDIA compilation tools. It implements 27 instructions and runs 5 benchmarks. Another distinct approach, also limited in terms of functionality, consists of FGPU [2], which is based on a custom ISA that takes into account the OpenCL specification. The implemented ISA mixes instructions from both MIPS and other, OpenCL-inspired,

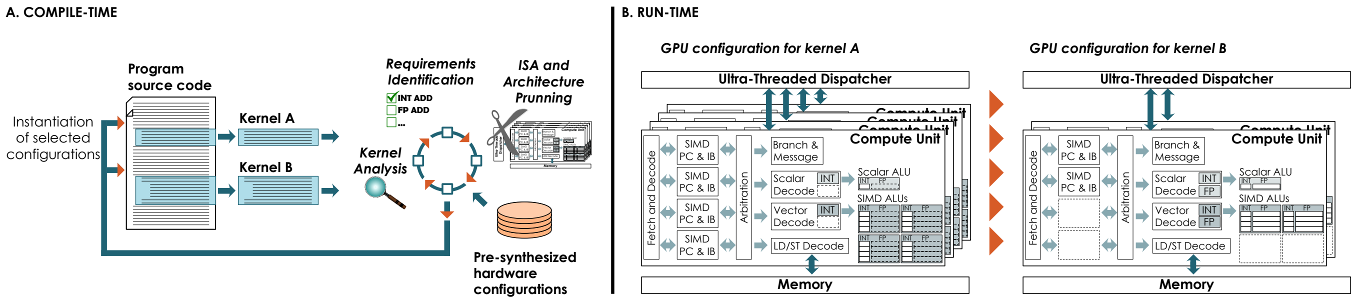


Figure 1: Two different trimmed architectures generated for two distinct soft-kernels. During compile time, the instructions present in kernel A indicate that only scalar and vectorized integer FUs shall be instantiated on the reconfigurable fabric. In kernel B both integer and floating-point instructions are detected and thus the architecture is trimmed (e.g., by considering available resources and/or power consumption limitations) to support both FU types.

architectures to create a soft SIMT processor. It supports 18 assembly instructions and runs 4 benchmarks. Although valid, these approaches present severe limitations at the functional level, which restrain the adoption by a vast set of real-world applications.

A new SCRATCH framework is herein proposed that builds on top of the recently developed MIAOW implementation [5] of the AMD Southern Islands architecture [17], which is extended in this work to support a wide set of 156 fully usable instructions (up from 42 in the original synthesizable design [5]), henceforth designated as MIAOW2.0 FPGA implementation. This comprehensive ISA extension allows testing a variety of well-known benchmarks (taken from the Rodinia [7] and the AMD SDK [18] suits) that were not possible in the original design. Furthermore, to tackle important performance bottlenecks associated with both the critical path of the original MIAOW architecture and its memory access latency, a set of architectural improvements were introduced, namely: a separate clock domain to decouple the critical paths of the compute unit and the ultra-threaded dispatcher, and a prefetch memory system to mitigate the penalizing data access latency of the original design, which was strictly based on accesses to slow global memory. The SCRATCH framework also includes a new compile-time tool that, by trimming down the implemented resources and instruction set (see Figure 1), allows tailoring the soft-GPGPU architecture towards the application-specific characteristics, therefore minimizing the used resources without compromising the correct program execution.

The proposed SCRATCH framework is therefore a full end-to-end solution, providing users a way to compile an OpenCL program (through AMD CodeXL [1]), trim the design to satisfy the application-specific requirements, synthesize, download it and run the application on Xilinx FPGAs.

The framework was validated on Xilinx Virtex 7 FPGAs, by relying on a set of both integer and single-precision floating-point benchmarks that have a direct application in the processing pipelines of many signal processing and Artificial Intelligence (AI) systems. In particular, it fully supports state-of-the-art image classification algorithms relying on Convolutional Neural Networks (CNNs) [11, 15, 33] that achieve the best accuracy results known so far [16, 19, 37].

The contributions of this work can be summarized as follows:

- MIAOW2.0, an extension to the original MIAOW architecture, has been fully validated on FPGA and bug-fixed to support a set of 156 instructions (including new floating-point ones); and enhanced with two clock domains, to isolate the critical paths of the compute unit and the remaining components, and a prefetch buffer within the compute unit to reduce the occurrence of slow global memory transactions;
- A compile-time trimming tool that is able to identify the requirements of each application and prune the MIAOW2.0 Compute Unit architecture to the application specific needs, releasing chip area for adding more compute units, cores and thus computational power to the parallel reconfigurable architecture;
- Demonstration of the performance and energy benefits of trimming down the architecture to the requirements of each specific application.

Such advantages are not only important for power and/or energy-constrained systems, including in exascale computing, where processing power is believed to be on the same order of magnitude as the human brain [29].

Hence, in this work the reader is confronted with a customized and flexible soft-GPGPU parallel architecture supporting native OpenCL code that is fully implementable on FPGA, as well as a trimming tool that helps adapting the architecture. Also, as Field-Programmable Gate Array (FPGA) technology progresses, it turns more amenable for system developers. As long as the desired performance metrics are met, the proposed soft-GPGPU tool is more friendly and attractive for application developers, which usually don't have the skills for programming the FPGA using Hardware Description Languages (HDLs). Furthermore, by releasing an open source soft-GPGPU that is implementable on FPGAs, it is possible to develop and test the impact of additional application-specific architectural optimizations that can provide additional performance gains, while validating its impact on area, frequency and power. For example, one can adjust the bitwidth of the datapath to provide additional gains in terms of area and power, since in many AI applications (including CNNs) it is perfectly acceptable to use less than 32 bit precision [22, 39].

The MIAOW2.0 architecture and SCRATCH tool are made publicly available online at `https://github.com/scratch-gpu`, under repositories `MIAOW2` and `Trimming-Tool`, respectively, for the community to enroll. They represent ongoing work and are therefore subject to continual updates with new releases.

2 THE MIAOW2.0 DESIGN

This work relies on the Many-Core Integrated Accelerator of Wisconsin (MIAOW), originally developed based on AMD's notion of an OpenCL compute unit [5] and on AMD's Southern Islands ISA.

2.1 MIAOW2.0 Core Architecture

The MIAOW Compute Unit (CU) has a single scalar ALU (SALU) and multiple–up to four–vector ALUs (VALUs), which can operate in, up to, 64 scalar words at once. MIAOW was mostly developed in Verilog, using a few C/C++ modules, through Programming Language Interface (PLI), to model the memories, memory controllers, and on-chip network (OCN). By using a synthesizable architecture mapped on an FPGA, instead of a simple simulator, conclusions can be drawn that are applicable to real hardware systems, which can then be constructed using the provided hardware description files.

2.1.1 The Pipeline: As shown in Figure 2, MIAOW's pipeline is composed of seven stages (Fetch, Decode, Issue, Schedule, Execute/Memory Access, and Write Back), with some being stages decomposed into multiple sub-stages, depending on instruction type. Hence, application kernels are executed as follows. The CU receives a program in wavefronts, i.e., a collection of 64 work-items, which share the same program counter. Each wavefront has a set of associated data, such as the program counter, the wavefronts' identifier, and the base address for both scalar and vector registers, and local memory. Since the Fetch unit is the input port for new wavefronts, it has to receive the supra-mentioned data and place the wavefront instructions on a queue (Wavepool), where they wait until being selected for decoding. The instruction fetch controller works in round-robin mode and supports up to 40 wavefronts to concurrently execute in the same CU.

Upon selection, a wavefront is passed to the Decode stage, which is responsible for extracting the operation to be executed; the source operands, which range from one to three; the destination; and multiple flags, depending on the type of instruction. Since some instructions use a 64-bit word length (namely, any VOP3 instruction and any other instructions using 32-bit literals), it requires two fetches and the joining of the two halves, before the decoding process can begin. Based on the extracted values, the Decode unit will automatically select the type of execution unit to be used, either VALU, SALU, or load-store unit (LSU), and translate logical register addresses into physical addresses. The decoded instruction then reaches the Issue stage where it waits until all dependencies have been resolved, only starting execution when all operands are ready to be accessed. If the instruction happens to be a barrier or a halt, the Issue unit will handle it immediately, not requiring any intervention from the remaining stages. For all other instructions, as soon as the operands are ready, it is scheduled for execution, causing a read from the register files. According to the unit selected in the Decode stage, one of three possible types of operation will be executed. If the instruction operates only on scalar operands, then SALU will be selected and an arithmetical or logical operation will be performed on the operands. A different scenario occurs for vector instructions, as VALU will be selected and multiple values will be operated at once, each value corresponding to a different work-item. To determine which work-items within a single wavefront are executed, an execute mask is used, which can be read and modified using normal scalar operations. Finally, if a memory instruction is casted, the LSU will be activated and a memory access will be performed. Before issuing the memory access request, however, the LSU performs an address calculation. Once execution finishes, a Write-Back will occur to either a result register, the execution mask, or to the conditional control flags, which, among other things, serve as a primary output for compare instructions.

2.1.2 Vector Register Direct Access Interface: To guarantee execution correctness across a wide set of applications, a set of registers must be initialized when a new workgroup starts executing. This is performed by the MicroBlaze, which acts as the Ultra-threaded Dispatcher. However, and although the original MIAOW system featured an interface for the scalar general-purpose registers (SGPRs), this was not implemented for vector operations. Adding an interface to provide concurrent access to vector general-purpose registers (VGPRs) required changing two major blocks: the advanced extensible interface (AXI) [43] interconnect peripheral, and the CU top level module, which is responsible for multiplexing input signals. The AXI interconnect peripheral has a set of memory mapped registers, through which the MicroBlaze processor [44] can communicate with the CU. The new interface requires expanding the existing register set to support it. Since the vector length is defined as 2048 bits[17], and a MicroBlaze processor only outputs 32 bit words, a set of 64 data registers is added to the peripheral. An address register is also set–controlling which VGPR is written. Additionally, to control which words of a vector are written, two 32 bit registers contain the write mask. Finally, a special address is defined to signal a write command, which causes the values in the data registers to be propagated to the VGPR.

2.1.3 ISA Extensions: The currently proposed CU architecture implementation supports 156 instructions from the AMD's Southern Islands ISA, being able to run unmodified OpenCL applications. This has been accomplished through exhaustive testing of the complete set of supported instructions, in order to validate and correct the CU behavior when dealing with all the supported scalar, vector, or memory instructions. For this, specific microbenchmarks (written in AMD's Southern Islands [17] machine code) and validation scripts were developed, each testing a different instruction domain. As a result of this validation procedure, the developed MIAOW 2.0 CU can run kernels generated by standard AMD OpenCL compilers (e.g. codeXL [1]) without hand tuning.

On the other hand, given that a large set of applications requires single-precision floating-point arithmetic, both the CU decode and execution stage were properly modified to incorporate new floating-point operations, thus considerably extending the range of possible applications, regarding other solutions [2, 4]. As in the previous case, the execution of the newly added instructions were also extensively validated to guarantee correctness.

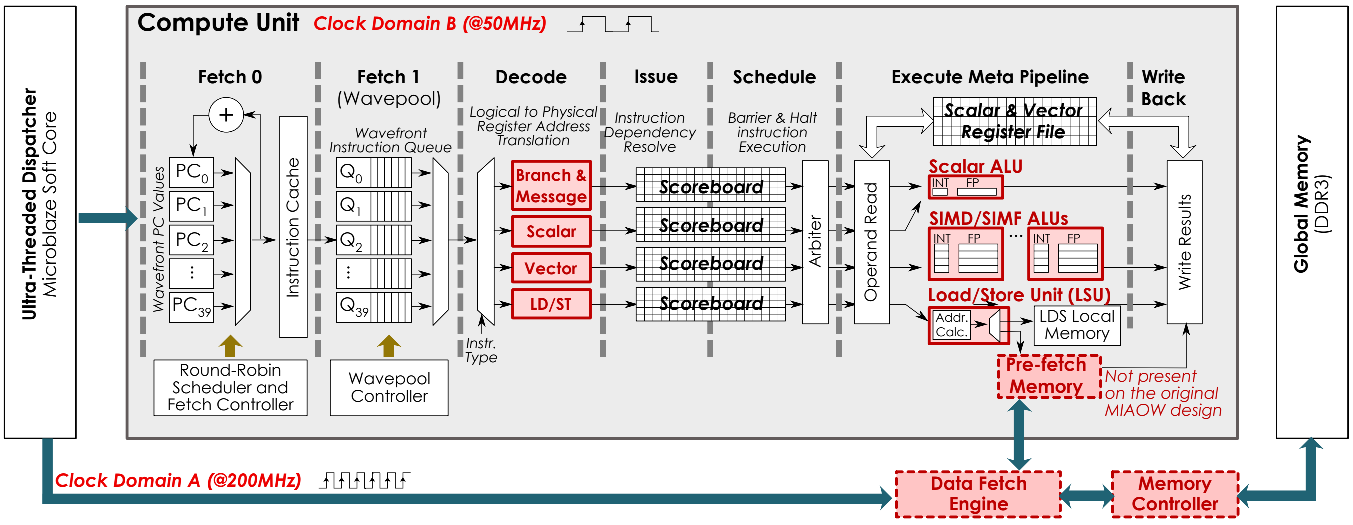


Figure 2: MIAOW2.0 GPGPU architecture and Compute Unit pipeline with the introduced optimizations. The blocks highlighted in red indicate changes to the original design, including architectural improvements to support additional instructions (solid red line) and new architectural elements to improve application performance (dashed red line).

2.1.4 Fast in-FPGA Prefetch Memory Buffer: To further decrease memory access latency, a set of FPGA block RAMs (BRAM) blocks are instantiated near the CU and used as a dedicated fast prefetch memory system. To maximize performance, at the beginning of execution, specific MicroBlaze commands pre-load the fast memory with application specific data. As a result, the average access latency becomes considerably reduced, not only because the data is closer to the execution units, but also because the access to such memory blocks does not require direct communication with a programmable processor/controller (a MicroBlaze in the current design, as described in the next subsection).

2.2 The FPGA Base System

2.2.1 Base Blocks and Debugging: The base system has four major components, which consist of a soft microprocessor from Xilinx–a MicroBlaze–, a DDR3 RAM memory controller, a timer and a debug module for the MicroBlaze. The timer module is used mainly to monitor MicroBlaze's execution times. MicroBlaze's debug module allows the debugging of a MicroBlaze processor through a joint test action group (JTAG) interface. It also emulates a Universal asynchronous receiver/transmitter (UART) module, not originally featured in the considered FPGA board (an AlphaData ADM-PCIE-7V3), enabling printing to the console. All major modules are connected to the MicroBlaze by an advanced extensible interface (AXI) bus [43], where the processor acts as a master, and all other peripherals are considered slaves. For debug purposes the general-purpose input/output (GPIO) pins were also instantiated as part of an AXI slave peripheral. These are responsible for on board light emitting diode (LED) lights.

The FPGA version of MIAOW (called NEKO in the original design [5] and MIAOW2.0 henceforth), like the other major components, needs to be connected to the MicroBlaze through an AXI interface. To guarantee that MIAOW2.0, or MIAOW for that matter, remains compatible with all FPGA boards, instead of adding AXI capability directly to the CU, this feature should be added to an intermediary interconnect peripheral. This peripheral acts as a bridge between the processor and the CU, by having a set of memory mapped registers that allows the compute unit to communicate with the processor, and vice versa.

After adding the interconnect peripheral as an AXI slave, it is necessary to guarantee that all timing constraints are satisfied. In order to do so, the clock frequency should be set to 50 MHz.

Once all modules are placed in the design, a top level entity (known as top level wrapper) is added to the project. This wrapper instantiates and connects the design and the CU and sets the system's inputs and outputs. Finally, the synthesis, implementation, and bitstream generation tools are executed. This allows the board to be programmed with the designed system. After downloading the system's bitstream to the FPGA, it can be directly programmed through Xilinx software development kit (SDK), in C. This programmability concedes an evaluation of the running state of the MIAOW2.0 CU.

2.2.2 Microblaze: The MicroBlaze soft processor is responsible for controlling the execution of OpenCL applications, working both as host processor and also acting as the Ultra-Threaded Dispatcher for the instantiated CUs. Hence, much like the host processor, it is responsible for implementing all non-accelerated application code, including data initializations, controlling the execution of the OpenCL kernels (by instructing the CU to execute new kernels), and retrieving the results.

On the other hand, the MicroBlaze also acts as Ultra-Threaded Dispatcher. Hence, it is responsible for managing and distributing the work across the CUs, tacking track of their execution state and launching new workgroups whenever the CU resources become ready. However, prior to execution, the CU needs to be initialized with state data registers, which is performed by the MicroBlaze, acting as an Ultra-Threaded Dispatcher. Naturally, since each application may require different register sets and state, such information must be retrieved at compile time. This is achieved through CodeXL (currently used for compilation purposes), which provides the detailed information about the initial register state.

Usually, the first three sets of scalar registers are used to contain memory descriptors. These are defined in AMD's Southern Islands ISA as a combination between a 48 bit address and state data for the memory access. The first set (`IMM_UAV`) is present in scalar registers four to seven and contains an offset for data gathering accesses. The second set of registers (`IMM_CONST_BUFFER0`), contains the base address of OpenCL call values. For instance, if a thread inquires its global ID this memory area is accessed, using a specific offset, to retrieve the required value. The third set of registers (`IMM_CONST_BUFFER1`), holds a pointer to the space in memory where the kernel arguments are kept. The fourth set of scalar registers contains the thread group ID across the three possible dimensions (X, Y, and Z). Naturally, if dimensions Y and Z are not set, only the first register of this set is initialized.

Finally, the vector registers (v0, v1 and v2) are pre-initialized to contain the thread IDs on the different dimensions (X, Y, or Z), where the dimensions depend on the type of application. A program whose data consists of an one-dimensional array only operates on the X dimension. If working on a two-, or three-dimensional matrix then the second, or third, dimension–Y and Z, respectively–, are also operated upon.

Although the MicroBlaze performs a wide set of operations, its execution code, highly resembles that of the OpenCL application host code, which must, however, be carefully modified in order to include the code responsible for the management of the workgroups and of the initialization of the prefetch memory. However, this can be easily achieved through a set of provided templates, which are constructed to implement basic algebraic functions.

2.2.3 *Memory interface controller:* The memory controller, also known as memory interface generator (MIG) [41], is responsible for intermediating memory accesses, as would be expected, but also for clocking and resetting the system. The latter is due to the constraint, set by Vivado, that MIG has to be connected to the boards clock and reset ports. In the used FPGA (AlphaData ADM-PCIE-7V3), the memory controller demands slight modifications in order to function properly. For instance, due to Vivado restrictions (version 2015.1 in the current work), the input clock period needs to be set to a value higher than 1500ps, while the original value is 1250ps. The value 2500ps (or 400MHz) was used, since this is the highest clock frequency directly available on the board. Moreover, it is desirable to have the system clock set with the highest possible frequency. MIG controls the system's clock by applying a ratio between the input and output clocks. This ratio is set to its lowest value (2:1), guaranteeing the maximum possible frequency for the system's clock. With this setting, for an input clock of 400MHz, the system's clock is 200MHz.

2.2.4 *Dual-clock Domain:* To run a program, a system including a MicroBlaze soft-processor and DDR3 memory is required in order to supply the instructions and data to the compute unit (see Figure 2). Also, the MIAOW's original system uses the MicroBlaze to communicate with the Memory Interface Generator (MIG). However, this significantly increases the latency for memory accesses and reduces the overall performance. Nonetheless, this delay can be mitigated by accelerating MicroBlaze's response time via an operating frequency increase (the architecture critical path resides within the CU pipeline), or by providing an infrastructure with lower memory access latency [23, 26, 35].

Hence, to directly tackle the first optimization direction, the original clock network was divided into two clock domains: the CUs domain with a clock frequency of 50 MHz, limited by the critical path of the Issue stage [14]; and the MicroBlaze and the memory access controllers domain, which operate at a substantially higher frequency (200 MHz).

2.3 MIAOW2.0 FPGA Validation

The complete design was carefully validated not only by ensuring a correct interaction between the system elements, but also by guaranteeing the proper execution of a wide set of applications and, in particular, their result. The connection between the modules was carefully assessed by writing values to the corresponding memory mapped registers and, afterward, by reading the values back after an operation by the corresponding component.

Following this step, a set of simple programs, specifically developed in AMD's Southern Islands machine code, was initially run on the CU, through the MicroBlaze processor. These programs consist of a few instructions that operate on the data registers, created by consulting AMD's ISA [17]. The generated binaries were hardcoded to the C program in hexadecimal form, through an `unsigned int` table. Accordingly, MicroBlaze was instructed to start by writing a few values to the CU's registers and populating the CU's instruction buffer with the given machine code. After initializing MIAOW2.0, the processor sends the start execution command and waits until the CU finishes. Finally, it recovers the resulting values present in the registers, and prints them on the screen.

The program flow allowed testing each type of instruction supported by the current implementation. Thus, a test script was developed with the goal of identifying the correctly implemented instructions from the complete listing present in the ISA [17]. The script was separated into three different programs, each working with either scalar, vector, or memory instructions. The main flow, for all three programs, was the same. For each type of instruction, one opcode (specific operation) is selected, following a sequential approach. The instruction binary was then generated in a set of functions which receive all the operands and output the corresponding machine code. Thereupon, the CU was initialized with both the instructions and data, and execution started. Once MIAOW2.0 finished executing, the results were recovered and compared with the expected output from a reference implementation.

3 SCRATCH SOFTWARE TOOLCHAIN

The SCRATCH framework was originally developed for the purpose of supporting a wide range of embedded applications, particularly related to signal processing and artificial intelligence. Hence, it provides the means for end-to-end OpenCL to FPGA implementation, by relying on a set of tools (as depicted in Figure 3), namely:

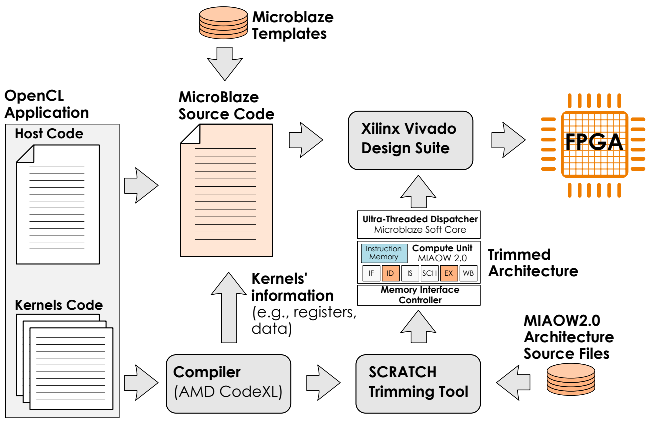


Figure 3: The SCRATCH end-to-end software toolchain, available online at `https://github.com/scratch-gpu`, under repositories `MIAOW2` and `Trimming-Tool`, respectively.

(i) an OpenCL compilation tool, currently implemented by AMD CodeXL; (ii) an architecture specialization tool, responsible for trimming down the architecture and allowing for the development of application-specific soft-GPGPU; and (iii) the Vivado Design suite (in this work relying on version 2015.1), to validate, map and test the design on Xilinx FPGAs.

Hence, this section describes the procedure used to generate the binary code, trim the architecture and run on the FPGA. It starts by motivating the need and design space for architecture trimming (subsection 3.1). It then describes the overall functionality of the SCRATCH trimming tool (subsection 3.2). Finally, it details the main steps in order to compile and execute the code on the FPGA (subsection 3.3).

3.1 The Need for Architecture Trimming

Even though a wide set of operations is required in a static design to support general-purpose applications, at the level of a single application, a significantly reduced number of operations is often required. Nonetheless, the hardware resources needed for supporting such a broad range of useless instructions still impact both area and power consumption. For example, consider a set of 25 benchmarks, retrieved from the AMD Application Parallel Programming (APP) software development kit (SDK) version 2.5 [18] presented in Figure 4. The different computational requirements are naturally translated into a different number and type of instructions. Hence, to evaluate such requirements, the applications kernels execution is analyzed using the Multi2Sim Simulator [38] (used to guarantee full support to all instructions including double-precision floating-point) and the instructions are classified into scalar or vector, and integer or floating-point operations of one of the following computational types: *mov* (register-to-register move operations); *logic* (including bit mask and bit compare); *shift* (including rotates); *bitwise* (bit search and bit count); *convert* (numeric format conversion); *control* (any control or communication operation, excluding logic and arithmetic compares); and arithmetic *add* (addition, subtraction and compare), *mul* (multiply with and without subsequent add), *div* (any divide or reciprocal operation), and *trans* (transcendental operations such as sine, cosine, exponential, and also square root and logarithmic operations).

Figure 4 presents the results of such an analysis at the level of the whole set of application kernels. By inspecting the figure, it can be concluded that although there is a small subset of instructions that is typically used by all benchmarks (namely those classified into *mov*, *logic* and *shift*), in most cases there is a substantial amount of instructions that is never used. For example, when considering arithmetic operations, 12 out of the 25 considered benchmarks require only add and multiply instructions, with half of them requiring no floating-point support. On the other hand, although *Black Scholes*, *DWT Haar 1D*, *Mersenne Twister* and *Monte Carlo Asian* require a large range of arithmetic operations, neither of those use double-precision floating-point arithmetic. On the same note, although several bit count and bit search operations are fundamental for bioinformatics applications (among others), neither of the considered benchmarks made use of these instructions.

Another interesting observation regards the use of the transcendental and other complex operations, whose implementation usually requires a high number of hardware resources. As a consequence, trimming unnecessary logic and functional units can provide for significant area and power savings, and can even lead to a processing throughput increase by using the saved area to instantiate additional compute units. The efficiency gains of the trimming methodology are presented in section 4.

3.2 Tool Overview

To deal with current FPGA limits and allow for an efficient exploitation of the reconfigurable logic, a Python-based trimming tool was developed to restrict the underlying logic to the specific application requirements. Hence, the SCRATCH trimming tool attempts to discard all functionality that is not required by each specific application. By disabling non-necessary functionalities, a simpler core is obtained, with a reduced size, saving area (resources) on the board. This core requires less power since there are fewer hardware components to feed, which translates into lower energy-consumption, given that the removal of unused resources does not affect execution time or throughput performance. The obtained core is, therefore, optimized regarding area, power, and energy for each application.

As shown in Figure 2, the CU is composed of eight major components, namely, fetching, decoding and scheduling units, scalar and vector register files, and execution units (LSU, SALU, and VALU). The units responsible for fetching and scheduling instructions–Fetch and Issue, respectively–have a relatively generic functionality and do not alter their behavior for a specific instruction. Furthermore, as previous studies also show [5], such components have limited impact on power ($< 11\%$) and area ($< 6\%$) requirements. On the other hand, the register file size can be significantly trimmed for particular applications. However, given the large bandwidth requirements of GPGPU applications and the specific arrangement of FPGA RAM blocks, it becomes difficult to exploit such design parameters. Nonetheless, these parameters remain as interesting optimization points in future architecture or technological revision, e.g., if the target implementation technology allows for fine-grained register file optimizations. The remainder units, highlighted with red color and with a solid line in Figure 2, perform specific tasks

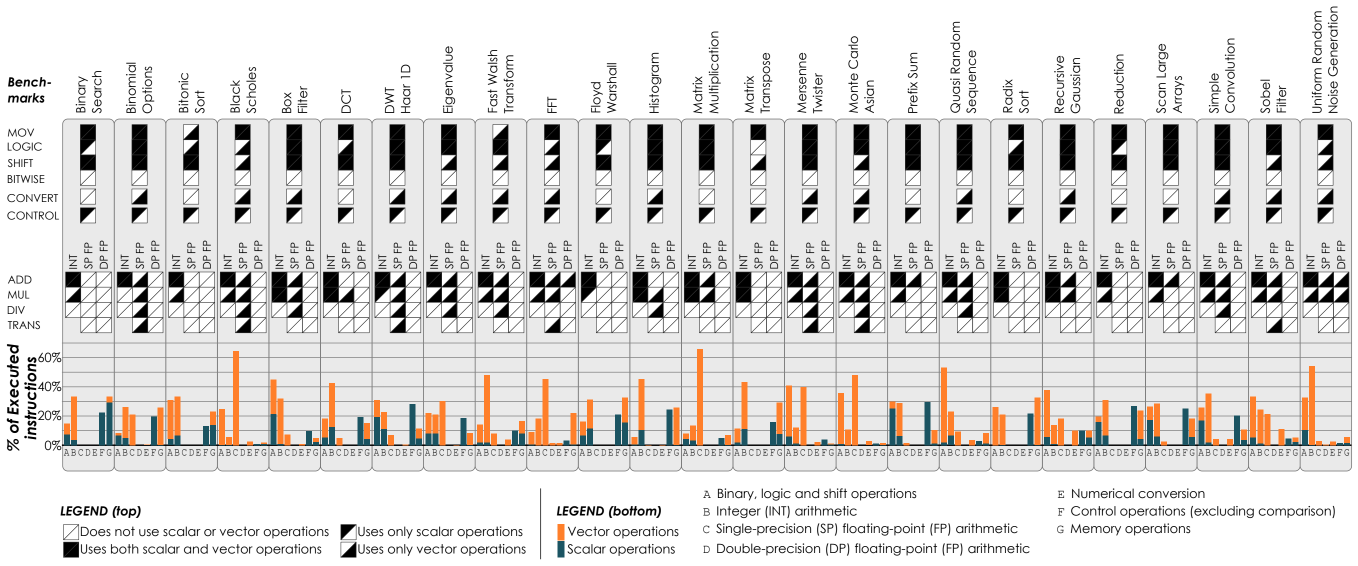


Figure 4: Characterization of the instructions executed on a AMD Southern Islands GPU for a set of 25 benchmark applications (retrieved from the AMD APP SDK version 2.5 [18]).

according to the instruction they are operating upon and were all specifically optimized while taking into consideration the application requirements.

The Decode unit receives the fetched instruction and produces a number of control signals, including the execution unit selector and register addresses. For each instruction, the control output changes. Hence, reducing the number of supported instructions simplifies the decode unit since part of the control circuit is eliminated.

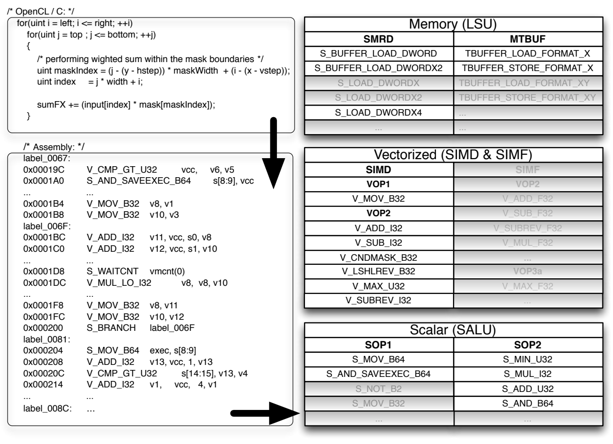
Each execution unit, or functional unit, performs a second instruction decode, which selects the correct operation to be executed. Moreover, they perform complex operations–from memory accesses (LSU) to floating-point scalar (SALU) or vector (VALU) arithmetic–becoming responsible for a large number of hardware resources (> 30%) and power consumption (> 50%) [5]. As a consequence, simplifying these units significantly helps to reduce the occupied area, which can then be used to instantiate additional (and application useful) computing resources (i.e., introducing parallelism).

With such architectural knowledge, the application-specific cores can be developed. The proposed tool constructs, at compile-time, a histogram of the number of instructions that use each decoding block and execution unit. Then, by taking the MIAOW's hardware description files as input, the tool, developed as a Python script, attempts to remove any unnecessary decoding or execution units, with the whole SIMD/SIMF units being removed if they remain unused by any instruction.

As shown in Algorithm 1, the proposed tool can be viewed as the sequence of two high-level steps. Initially, as shown in lines 2 to 11, the binary file of the application is processed and a list of required instructions per functional unit is returned. Upon obtaining this information from the binary, the framework is able to trim the core, in a second step, as illustrated in lines 13 to 28.

Algorithm 1 Core-trimming framework's algorithm

```
 1: import miaow
Ensure: Open the application's binary file
Ensure: required_instructions = empty_dictionary()
 2: First step:
 3: for line in app_binary_file: do
 4:     [opcode, operands, type, FU] = miaow.decode(line)
 5:     if FU not in required_instructions.keys(): then
 6:         required_instructions[FU] = empty_list()
 7:     end if
 8:     if opcode not in required_instructions[FU]: then
 9:         required_instructions[FU].append((opcode,type))
10:     end if
11: end for
Ensure: Step one is finished
Ensure: Get required_instructions dictionary
12: Second step:
Ensure: req_func_units = required_instructions.keys()
13: for FU in miaow.fu_list(): do
14:     if FU not in req_func_units: then
15:         out_signals = get_output_signals(FU)
16:         miaow.remove_instantiation(FU)
17:         miaow.ground_signals(out_signals)
18:     end if
19: end for
20: for FU in req_func_units: do
21:     FU_instructions = miaow[FU].instructions()
22:     for instruction in FU_instructions: do
23:         if instruction not in required_instructions[FU]: then
24:             miaow["decode"].remove(instruction)
25:             miaow[FU].remove(instruction)
26:         end if
27:     end for
28: end for
```



```
/* OpenCL / C: */
for(uint i = left; i <= right; ++i)
  for(uint j = top ; j <= bottom; ++j)
  {
    /* performing wighted sum within the mask boundaries */
    uint maskIndex = (j - (y - hstep)) * maskWidth  + (i - (x - vstep));
    uint index     = j * width + i;

    sumFX += (input[index] * mask[maskIndex]);
  }
```

```
/* Assembly: */
label_0067:
0x00019C     V_CMP_GT_U32          vcc,    v6, v5
0x0001A0     S_AND_SAVEEXEC_B64      s[8:9], vcc
...          ...
0x0001B4     V_MOV_B32    v8, v1
0x0001B8     V_MOV_B32    v10, v3
label_006F:
0x0001BC     V_ADD_I32    v11, vcc, s0, v8
0x0001C0     V_ADD_I32    v12, vcc, s1, v10
...          ...
0x0001D8     S_WAITCNT    vmcnt(0)
0x0001DC     V_MUL_LO_I32          v8,  v8, v10
...          ...
0x0001F8     V_MOV_B32    v8, v11
0x0001FC     V_MOV_B32    v10, v12
0x000200     S_BRANCH     label_006F
label_0081:
0x000204     S_MOV_B64    exec, s[8:9]
0x000208     V_ADD_I32    v13, vcc, 1, v13
0x00020C     V_CMP_GT_U32          s[14:15], v13, v4
0x000214     V_ADD_I32    v1,    vcc,   4, v1
...          ...
label_008C:  ...
```

Memory (LSU)

SMRD	MTBUF
S_BUFFER_LOAD_DWORD	TBUFFER_LOAD_FORMAT_X
S_BUFFER_LOAD_DWORDX2	TBUFFER_STORE_FORMAT_X
S_LOAD_DWORDX	TBUFFER_LOAD_FORMAT_XY
S_LOAD_DWORDX2	TBUFFER_STORE_FORMAT_XY
S_LOAD_DWORDX4	...
...	...

Vectorized (SIMD & SIMF)

SIMD	SIMF
VOP1	**VOP2**
V_MOV_B32	V_ADD_F32
VOP2	V_SUB_F32
V_ADD_I32	V_SUBREV_F32
V_SUB_I32	V_MUL_F32
V_CNDMASK_B32	...
V_LSHLREV_B32	**VOP3a**
V_MAX_U32	V_MAX_F32
V_SUBREV_I32	...

Scalar (SALU)

SOP1	SOP2
S_MOV_B64	S_MIN_U32
S_AND_SAVEEXEC_B64	S_MUL_I32
S_NOT_B2	S_ADD_U32
S_MOV_B32	S_AND_B64
...	...

Figure 5: High-level illustration of the SCRATCH core-trimming framework for an integer conv2D benchmark scenario. After receiving as input the hardware description files and the binary code for a certain application, the SCRATCH framework outputs the application-specific hardware description files. The figure illustrates a part of the Assembly code generated and the corresponding instructions synthesized for the particular case of the conv2D, where the shadowed instructions represent the units removed. For the particular case of the SIMF units, the complete FU was trimmed off from the design.

It is worth noticing that there are applications which use only some of the available execution units. This situation occurs for instance if a given application only performs integer operations, in which case the whole floating-point VALU (SIMF) can be removed. Such applications immediately achieve a significant decrease in resource usage as the floating-point VALU uses almost twice the resources of an integer VALU, becoming the single largest unit in the design [5].

Figure 5 illustrates the case of the conv2D benchmark. A part of the instruction set synthesized, resulting from the binary code produced, is depicted for the three types of SALU, VALU and LSU resources used. Also, it is shown in shadow a subgroup of unused instructions that result in area and power savings. For this particular case they are quite significant since the kernel strictly performs integer operations and thus the vectorized SIMF FU can be completely scratched from the design.

The released hardware resources can then reallocated for replicating dedicated compute or functional units, fostering the throughput acceleration of that particular benchmark.

3.3 From OpenCL to FPGA Implementation

The proposed SCRATCH framework also relies on a set of tools such as to provide end-to-end OpenCL to FPGA implementation. The full mechanism works as follows. The framework currently relies on the AMD CodeXL [1][1] to compile the application as well as to provide the necessary information for both the trimming tool and FPGA kernel loading. Hence, it is not only used to provide the Assembly code (which is currently used as the input to the Trimming Tool), but also identifies the set of registers that must be initialized when the Ultra-threaded Dispatcher (implemented by the MicroBlaze) is loading a new workgroup. Moreover, it is also used to generate the actual binary code, which must also be loaded by the MicroBlaze at the beginning of execution.

On the other hand, the MicroBlaze is currently used to perform a wide set of operations (see also subsection 2.2.2), such as host processor and Ultra-threaded Dispatcher. Hence, and although in practice the MicroBlaze code highly resembles that of the OpenCL host, a set of templates are provided to help the user develop the MicroBlaze program. Such templates, which are based on basic algebraic functions, extend beyond the traditional OpenCL host code by including the initialization of registers, the handling of data to/from the global main memory (i.e., which falls outside of the prefetch buffer) and the management of OpenCL kernels and workgroups.

The compilation of the MicroBlaze code is handled by the Xilinx Vivado Design Suite (that in the current work uses version 2015.1). This application is also used for logical validation, synthesis and FPGA implementation.

4 EXPERIMENTAL RESULTS

The developed work targeted an AlphaData ADM-PCIE-7V3 board, containing a Xilinx Virtex 7 XC7VX690T FPGA. The board was used in a CentOS 6.7 host system equipped with an Intel Core i7-4770K CPU operating at 3.50GHz. The hardware designs were synthesized and implemented using Vivado 2015.1 and the kernels were developed and compiled under OpenCL v1.2 using AMD's CodeXL version.

To validate the proposed work, benchmarks from both Rodinia [7] and the AMD OpenCL SDK 2.5 [18] were selected. Additional AI-specific benchmarks were also developed to evaluate CNN-based applications [19] in the context of automatic image classification challenges, namely: a CNN using a fully connected 3-layer topology, with 16 feature maps per layer, and a 2×2 max pooling function at the end of each layer; and a multi-layer Network-in-Network (NIN) adapted from [21, 28] with each convolutional Multi-Layer Perceptron (MLP) layer featuring 16 feature-maps, a partially sparse MLP-010 neural network [28], and an average pooling function at the output. The networks, implemented in both floating-point and fixed-point numerical precision, were used to classify RGB images of different input sizes (namely 32×32, which correspond to the CIFAR-10 challenge, as well as images of size up to 512×512).

A total of 17 fixed- and floating-point applications were benchmarked, namely: K-Means (K-M) and Gaussian elimination from the Rodinia benchmark suite; and both integer and floating-point matrix addition, multiplication and 2D convolution, as well as bitonic sorting, and matrix transpose from the AMD OpenCL SDK 2.5. Additionally, three pooling algorithms were implemented–namely max, median and average pooling in a 2×2 matrix vector–as well as the referred floating-point and integer CNNs and NINs.

From the selected applications a small subset requires host processing operations (implemented in the MicroBlaze), namely K-means clustering and Gaussian elimination. K-means is an iterative

[1]Requires the existence of an AMD GPU for correct operation.

algorithm which partitions N observations into K clusters. Between iterations, MicroBlaze has to recompute the center of mass for each cluster. Gaussian elimination, on the other hand, only requires MicroBlaze to act after MIAOW2.0 finishes. Initially, the CU puts the matrix in triangular form. Then, the MicroBlaze performs the back-substitution to obtain the final result.

The execution time, for all applications, was measured using a cycle counter internal to the CU, and the MicroBlaze processing time was quantified using the timer module presented in the design. To guarantee correct behavior, the output of all applications were compared and validated with the corresponding standard implementations.

4.1 Efficiency of the Trimmed Architecture

To evaluate the impact of architecture trimming, two distinct stages must be analyzed. First, the resource savings that are attained by trimming off the architecture are analyzed. At a second stage, the freed resources are directly employed to increase the architecture parallel processing power and improve performance and energy-efficiency.

4.1.1 Pruning and Resource Utilization. The results for the FPGA resource utilization for both the original MIAOW architecture, and also for the introduced dual-clock domain (DCD) and prefetch memory (PM) are presented in Figure 6. From the analysis of this figure, it is clear that the addition of a second clock domain (DCD) does not cause any increase in the reported resource utilization. The addition of the prefetch memory, however, significantly raises the utilization of BRAM units. However, this is a result of the used design methodology, which distributes most of the (otherwise unused) BRAM units to generate the CUs prefetch memory (since this generally leads to superior performance). Hence, while in this design most BRAM units are used by the prefetch memory of a single CU, for the multi-core parallel configurations (see below), these blocks are distributed across the instantiated CUs.

To analyze the architecture trimming potential for the considered benchmarks, Figure 6 (first column) presents the percentage of instructions used, when comparing with the original number of supported instructions (grouped by target execution unit). As it can be observed from the figure, many of the benchmarks use only a rather reduced number of instructions, especially when considering scalar and vector, logic and arithmetic operations. In particular, although floating-point arithmetic is fundamental in certain applications, the actual number of used floating-point instructions is rather low in most cases, with the 2D convolution benchmark (SP FP) using the highest number of such instructions (15%).

By relying on such instruction usage analysis, we then apply the proposed architecture adaptations and trim unnecessary resources, namely related to instruction decode and execution. Figure 6 (second column) presents the resource savings regarding the Baseline architecture (i.e., considering both DCD+PM). Such results show a significant decrease in the usage of both slice flip-flops and LUTs (average of 41% and 36%, respectively), mostly arising from the elimination of unused vector execution units. The most striking examples are related to matrix transpose and pooling benchmarks that, due to their intrinsic nature, require a very limited number of execution units. In such cases, the resource savings reach 72%

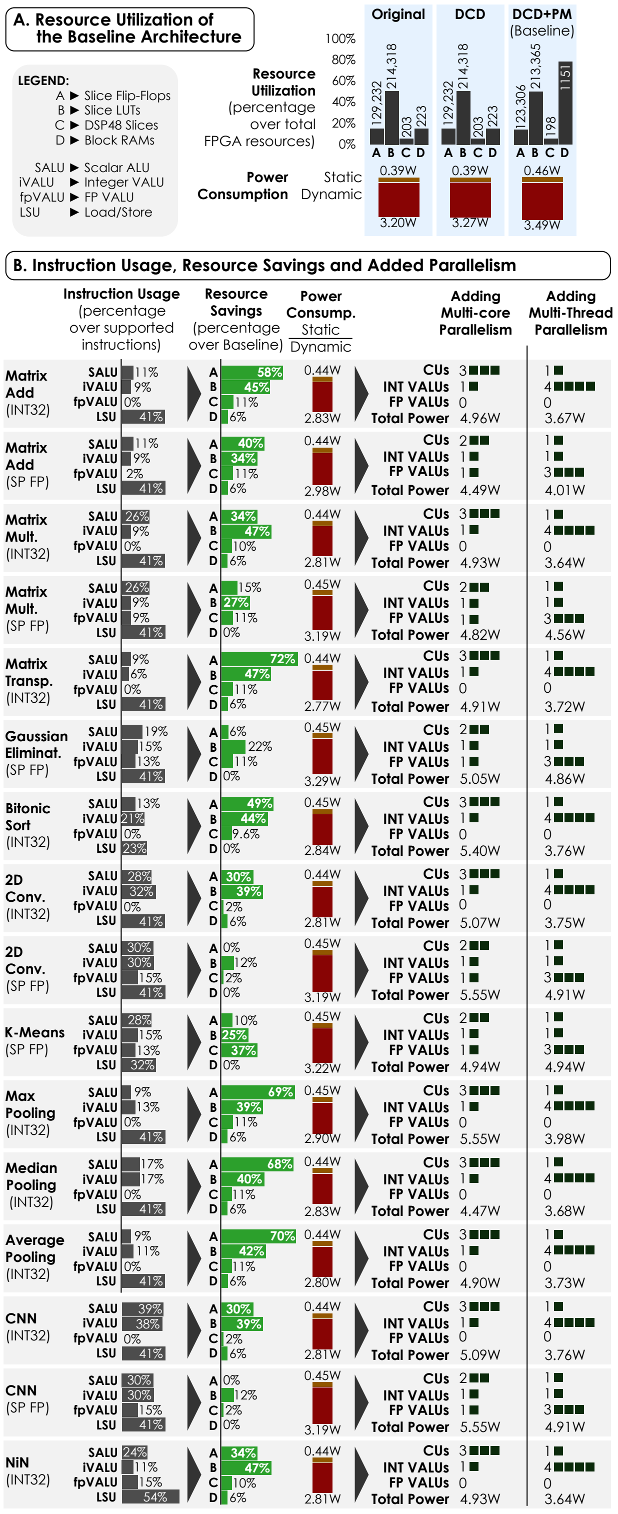


Figure 6: Resource and instruction utilization, power consumption, and exploited parallelism for both the original and application-aware trimmed architectures.

and 47% in Slice flip-flops and slice LUTs, respectively. On the other hand, the floating-point 2D convolution benchmark attains the lowest reduction in resource utilization. In particular, although the benchmark uses a small number of instructions, those instructions scatter along a large number of functional units, which reduces the potential for architecture trimming (although additional savings could be attained by specifically constraining the compiler to use certain operations). Nonetheless, when considering the total number of logic cells available in the considered FPGA, the attained area savings already provide the means to increase the number of instantiated computational elements, thus allowing for a throughput performance increase (see subsection 4.2).

When analyzing the area savings related with DSP48 slices, it can be perceived that only a limited reduction in resource utilization is attained. This is concerned with the fact that such resources are used in the context of general scalar and vector operations, typically required in the control flow of all benchmarks. Similar results are also obtained when considering BRAM units, where limited savings are achieved, since FPGA fixed size units do not allow for a more efficient adjustment of memory sizes.

4.1.2 Throughput and Energy-efficiency Gains. The results exposed in Figure 6 show that with each change proposed in Section 2 the system's power requirement increases, which could be seen as an undesirable result. The introduction of a second clock domain (DCD system), with a higher frequency, increases the power requirement by 1.02× while adding a prefetch memory block to the system increases the same unit by 1.10× (see Figure 6). In contrast with this relatively small increase in power requirement, there is a substantial decrease in computation time–and thus an increase in throughput performance, or speedup–, which offsets the corresponding energy-efficiency values. When comparing the DCD system with the original one, the minimum registered speedup for the integer 2D convolution of a 256×256 matrix with a 15×15 kernel, is 1.17×. Therefore, in the worst case scenario observed for the DCD system, the energy efficiency is increased by a factor of 1.147×–as a reminder $E = P \times t$. For the DCD+PM system, the minimum speedup achieved is 4.27×, which results in an energy efficiency increase of 3.88×, while the maximum speedup of 95.79× results in an 87× improvement on energy consumption. Attending to the data, the conclusion that can be drawn is that both improvements increase energy efficiency and throughput performance. On average, the energy-efficiency is increased by 1.17× for the DCD system and 55.87× for the DCD+PM case.

The customized cores optimize the DCD+PM system even further by removing non-necessary functionalities. When compared to the DCD+PM system, the matrix transpose achieved a further rise in energy efficiency of 1.23×, representing the system with the lowest power requirement. The floating-point matrix convolution system, on the other hand, achieved the lowest increase in energy efficiency, when compared to the DCD+PM, with an improvement of 1.02×. In general, all the non-floating-point systems achieved an improvement of at least 1.15×, compared to the DCD+PM, while the floating-point ones usually fared between 1.02× and 1.10×. The exception to the previous case is the floating-point matrix addition which observed an improvement of 1.15×, similar to the integer kernels. The aforementioned kernel does not contain floating-point multiplications or divisions which are considerably more complex than floating-point additions, therefore explaining the improvement seen in this kernel. Overall, when comparing to the original system, the floating-point matrix addition obtains the best energy efficiency improvement of more than 100×.

4.2 Multi-thread and Multi-core Parallelism

The last improvement consists in the exploitation of the attained resource savings to increase the systems' parallel processing power. Clearly, two parallelization approaches are possible: increasing the number of FUs per CU to introduce additional multi-threaded parallelism; or increasing the number of CUs in general to augment the multi-core parallelism. By taking into account the maximum number of units that can still fit in the considered FPGA, new application-aware architecture designs were constructed, as illustrated in the last two columns of Figure 6. The achieved throughput performance and energy-efficiency gains when increasing the systems' parallel processing power are presented in Figure 7.

For the multi-core parallel processing approach (illustrated in Figure 7A), throughput speedups of up to 240× and 3.0× regarding the original and the baseline (DCD+PM) designs, respectively, are attained. When considering the corresponding power increase, such performance improvements result in maximum energy-efficiency (instructions-per-Joule) gains of 220× and 3.3×, respectively, for the CNN benchmark, with the minimum values being observed for the Gaussian Elimination benchmark (20× and 1.5×, respectively). For the NIN benchmark, and following recent trends in Deep Neural Networks (DNNs) [31, 39], we also vary the numerical precision from a 32-bit format to shortened 8-bit format. As could be expected, this further reduces per-CU resource usage, and allows to instantiate up to 4 CUs. As a consequence, there is an additional performance and energy-efficiency gain, regarding the 32-bit solution.

For the multi-thread parallel processing approach (see Figure 7B), higher throughput speedups are observed. In particular, a maximum speedup of 3.5× regarding the baseline architecture (or 260× regarding the original solution) is observed for the CNN benchmarks. Such performance improvements are translated into a 3.7× (or 252×) energy-efficiency increase.

4.3 Discussion

Although constrained by a frequency of operation 20× inferior to those of conventional multi-core processors and by a limited amount of resources, which impose a maximum number of 3 CUs, the proposed trimmed architectures are more efficient in area and have lower power requirements. Furthermore, they are fully customizable, capable of accommodating additional computing elements (given larger FPGAs), and have applicability in many areas that do not require the throughput and compute-power of high-end thousand-core GPUs, which can be traded for power and energy-efficiency. Modern examples that impose such constraints are the use of robots, drones or other mobile equipment that typically have to capture and process image and other forms of data from sensors, that more often operate far away from power supplies. Additionally, even data centers can benefit from such approaches given that energy consumption has a large impact on their annual running costs. Hence, compute power can be more efficiently targeted

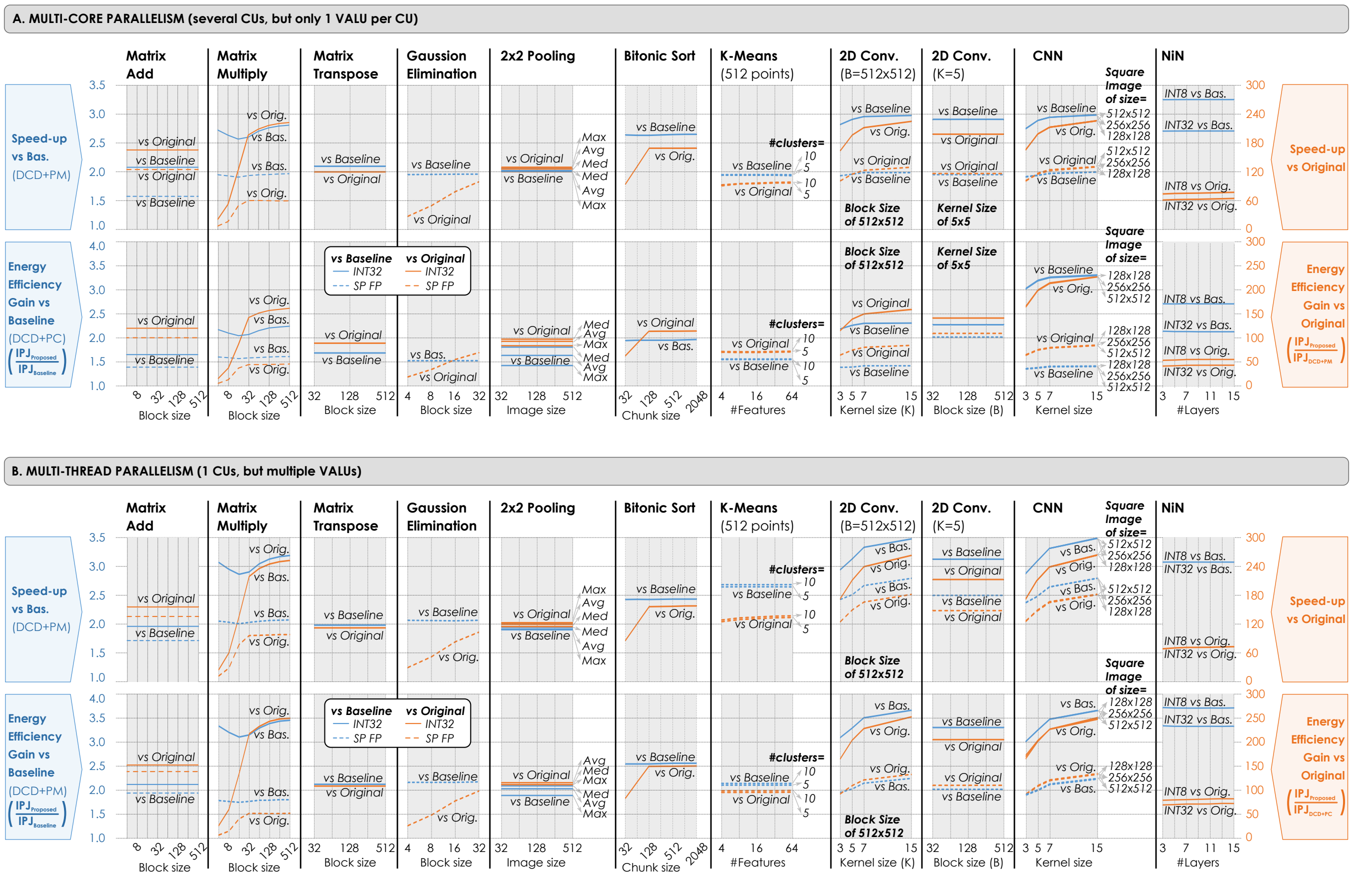


Figure 7: Throughput performance and energy-efficiency gains attained by first applying architecture trimming and then using the freed resources to increase multi-core/multi-thread parallelism.

for application-specific contexts, as it has been proposed recently for search engines [30], where considerable energy savings are obtained.

At this respect, and even though the reconfiguration of the FPGA represents an obvious advantage, one can go a step further of what is already performed in this work, where the soft-GPGPU is trimmed at a per-application level. Hence, trimming could be applied at a per-kernel level, with reconfiguration occurring between kernel calls. Naturally, this will impose additional overheads related with FPGA reconfiguration, whose actual impact on throughput performance depends on the ratio between kernel execution time and architecture reconfiguration time. To mitigate the latter, the best strategy would be to fix the number of CUs and, by applying careful floor-planning, to allow for partial reconfiguration [40] of the area allocated to vector execution units (SIMD and SIMF blocks in Figure 2). On the other hand, many applications are characterized by a reduced number of kernels with minimal differences being observed in the per-kernel requirements. For example, in many applications there is no specific requirements for floating-point arithmetic, even across different kernels (e.g., typical DNN inference applications where fixed-point arithmetic with reduced numerical precision can be adopted). Moreover, in many cases, a single functional unit can be used to implement multiple instructions. Under such conditions, it may be preferable to apply trimming at an application level, rather than at kernel level.

5 RELATED WORK

In recent years, significant research efforts have been made in the development of soft-GPGPUs, either by relying on existing commercial ISAs, or by developing special-purpose ones.

Soft-GPGPUs: Andryc et al. proposed FlexGrip [4] based on NVIDIA's G80 architecture. The reported implementation is based on a Single Instruction Multiple Thread model in which the Streaming Multiprocessor (SM) is composed of multiple Scalar Processors (SPs) all running the same instruction on different threads. The core has a five stage pipeline consisting of Fetch, Decode, Read, Execute and Write-Back, where the SPs correspond to the Execute stage. Each scalar processor operates on a single 32 bit word, with an SM being composed of 8 SPs in the FPGA implementation. The main novelty in FlexGrip is the support for direct GPU compilation, i.e., the binaries to program the system are generated by unmodified standard NVIDIA tools. The architecture has 27 CUDA 1.0 integer instructions implemented, being able to run five CUDA benchmarks. The design was successfully implemented on an ML605 Virtex-6, using a MicroBlaze microprocessor as a host processor that supplies the SM with both the program memory and the application data.

A different approach was followed with FGPU [2], which proposes a custom ISA taking into account the OpenCL specification. The implemented ISA mixes instructions from both MIPS and other, OpenCL-inspired, architectures in order to create a soft SIMT processor. The architecture's RTL design was performed using VHDL and optimized for FPGA implementation. Contrary to FlexGrip's approach, which presented a single SM with multiple SPs, FGPU was successfully implemented in a ZC706 Zynq board with up to 8 cores, each one containing 8 processing elements (PEs), with the on-board ARM processor acting as the host processor. Apart from the original MIPS instructions, designers implemented 18 assembly instructions, which, although limited, were sufficient to run four benchmarks, namely memcopy, vecmul and vecadd, FIR (5 taps), and cross correlation.

Application-specific Architecture Adaptation: the idea of GPU architecture adaptation has been previously exploited with different goals. In particular, [36] identifies that the usage of architecture components is highly dependent on the application kernel characteristics, and may even cause contention on the over-utilized components. The authors propose to dynamically monitor resource usage and adjust the concurrency level (number of threads), core frequency and memory frequency to improve performance or save energy. Similarly, but with the specific goal of minimizing energy consumption while fulfilling user-specified performance goals, [34] proposes a control scheme to optimize the number of compute units, the number of warps/wavefronts, as well as the core and memory operating frequencies. Although such approaches focus mostly on the memory/compute-bound duality, they represent complementary approaches to our work.

Another interesting methodology focused kernel-specific adaptations to mitigate flow divergence, memory divergence, and to improve locality [32]. To accomplish this, the authors rely on a hierarchical warp scheduler to force convergent threads to operate in lockstep, while divergent threads are allowed to split and execute different instruction flows. As in the previous case, this is an alternative strategy, which can complement the proposed approach to further improve the GPU energy-efficiency.

6 CONCLUSIONS

GPUs have emerged as an alternative approach to tackle the throughput performance-per-Watt problem, particularly in data-centers and embedded processing systems. However, off-the-shelf GPU designs are tailored for a wide range of applications, therefore supporting a large number of operations that most of time are not required by the target application. To overcome such an issue, an application-aware soft GPGPU architecture is proposed, which is based on the MIAOW's implementation of the AMD Southern Island architecture, although augmented to support a wider range of instructions (with 156 correctly implemented instructions) and to improve overall throughput performance. Additionally, an architecture trimming methodology is proposed that, by discarding unnecessary logic elements, is able to adapt the GPU architecture towards the characteristics of the target application. By implementing the proposed architecture on a Xilinx Virtex 7 FPGA (AlphaData ADM-PCIE-7V3 board), we show that the proposed methodology is able to significantly reduce the hardware resources, with an average 41% and 36% savings in slice flip-flops and LUTs, respectively. By employing such area savings to increase the GPU parallel processing power, these resource savings allow for a peak throughput performance increase in over 3× regarding the baseline architecture (260× regarding the original MIAOW design), and 3.5× (250×) energy-efficiency peak improvement.

The complete framework, including the MIAOW2.0 design and the SCRATCH trimming tool are made publicly available online at https://github.com/scratch-gpu/MIAOW2 and https://github.com/scratch-gpu/Trimming-Tool, respectively, for the community to enroll. The framework corresponds to ongoing work, being subject to continual updates and new releases.

7 ACKNOWLEDGEMENTS

The authors would like to acknowledge and thank financial support from Fundação para a Ciência e a Tecnologia (FCT), Instituto de Telecomunicações and INESC-ID under grants UID/EEA/50008/2013 and UID/CEC/50021/2013.

Also, the authors would like to thank João Vieira for the exhaustive tests and validation performed to the SCRATCH framework, trimming tool and MIAOW2.0 architecture.

REFERENCES

[1] Advanced Micro Devices Inc. 2017. CodeXL. (2017). http://gpuopen.com/compute-product/codexl/

[2] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. 2016. FGPU: An SIMT-Architecture for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, New York, NY, USA, 254–263. https://doi.org/10.1145/2847263.2847273

[3] Altera Corp. 2014. Altera SDK for OpenCL. Programming Guide. (2014).

[4] K. Andryc, M. Merchant, and R. Tessier. 2013. FlexGrip: A soft GPGPU for FPGAs. In *International Conference on Field-Programmable Technology (FPT)*. 230–237. https://doi.org/10.1109/FPT.2013.6718358

[5] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. 2015. Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2, Article 21 (June 2015), 21:1–21:25 pages. https://doi.org/10.1145/2764908

[6] A. Bourd. 2016. The OpenCL Specification: Version 2.2. (March 2016). khronos.org/registry/cl/specs/opencl-2.2.pdf

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. 2008. A Performance Study of General-purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing (JPDC)* 68, 10 (Oct. 2008), 1370–1380. https://doi.org/10.1016/j.jpdc.2008.05.014

[9] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Bespoke Processors for Applications with Ultra-low Area and Power Constraints. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, USA, 41–54. https://doi.org/10.1145/3079856.3080247

[10] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. 2012. From OpenCL to high-performance hardware on FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 531–534. https://doi.org/10.1109/FPL.2012.6339272

[11] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[12] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. D. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne. 2012. Shortening Design Time through Multiplatform Simulations with a Portable OpenCL Golden-model: The LDPC Decoder Case. In *IEEE 20th International Symposium on Field-Programmable*

Custom Computing Machines (FCCM). 224–231. https://doi.org/10.1109/FCCM.2012.46

[13] G. Falcao, V. Silva, L. Sousa, and J. Andrade. 2012. Portable LDPC Decoding on Multicores Using OpenCL [Applications Corner]. *IEEE Signal Processing Magazine* 29, 4 (July 2012), 81–109. https://doi.org/10.1109/MSP.2012.2192212

[14] V. Gangadhar, R. Balasubramanian, M. Drumond, Z. Guo, J. Menon, C. Joseph, R. Prakash, S. Prasad, P. Vallathol, and K. Sankaralingam. 2015. MIAOW: An open source GPGPU. In *IEEE Hot Chips 27 Symposium (HCS)*. 1–43. https://doi.org/10.1109/HOTCHIPS.2015.7477460

[15] K. He, X. Zhang, S. Ren, and J. Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *IEEE/CVF International Conference on Computer Vision (ICCV)*. 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[17] Advanced Micro Devices Inc. 2012. Southern Islands Series Instruction Set Architecture Reference Guide. (2012).

[18] Advanced Micro Devices Inc. 2016. Accelerated Parallel Processing (APP) Software Development Kit (SDK). (2016). http://developer.amd.com/sdks/amdappsdk/

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[20] K. Krommydas, A. E. Helal, A. Verma, and W. C. Feng. 2016. Bridging the Performance-Programmability Gap for FPGAs via OpenCL: A Case Study with OpenDwarfs. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 198–198. https://doi.org/10.1109/FCCM.2016.56

[21] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[22] Jose Marques, Jose Andrade, and Gabriel Falcao. 2017. Unreliable Memory Operation on a Convolutional Neural Network Processor. In *Proceedings of the 2017 IEEE International Workshop on Signal Processing Systems (SiPS) (SiPS'17)*. IEEE, New York, NY, USA, 1–6.

[23] Sparsh Mittal and Jeffrey S Vetter. 2015. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys (CSUR)* 47, 2 (2015), 19.

[24] A. Momeni, H. Tabkhi, G. Schirner, and D. Kaeli. 2016. Hardware thread reordering to boost OpenCL throughput on FPGAs. In *IEEE 34th International Conference on Computer Design (ICCD)*. 257–264. https://doi.org/10.1109/ICCD.2016.7753288

[25] N. Neves, N. Sebasti ao, D. Matos, P. Tomás, P. Flores, and N. Roma. 2015. Multicore SIMD ASIP for Next-Generation Sequencing and Alignment Biochip Platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 7 (July 2015), 1287–1300. https://doi.org/10.1109/TVLSI.2014.2333757

[26] N. Neves, P. Tomás, and N. Roma. 2017. Adaptive In-Cache Streaming for Efficient Data Management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 7 (July 2017), 2130–2143. https://doi.org/10.1109/TVLSI.2017.2671405

[27] NVIDIA. 2017. *CUDA C Programming Guide 8.0*. NVIDIA.

[28] Y. Pang, M. Sun, X. Jiang, and X. Li. 2017. Convolution in Convolution for Network in Network. *IEEE Transactions on Neural Networks and Learning Systems* PP, 99 (2017), 1–11. https://doi.org/10.1109/TNNLS.2017.2676130

[29] Human Brain Project. 2017. Human Brain Project. (June 2017). https://www.humanbrainproject.eu/en/

[30] Andrew Putnman, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 13–24.

[31] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 267–278.

[32] Timothy G Rogers, Daniel R Johnson, Mike O'Connor, and Stephen W Keckler. 2015. A variable warp size architecture. In *ACM/IEEE 42nd International Symposium on Computer Architecture (ISCA)*. IEEE, 489–501.

[33] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[34] Muhammad Husni Santriaji and Henry Hoffmann. 2016. GRAPE: Minimizing energy for GPU applications with performance requirements. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[35] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press, 73–82.

[36] Ankit Sethia and Scott Mahlke. 2014. Equalizer: Dynamic tuning of GPU resources for efficient execution. In *IEEE/ACM 47th Annual International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 647–658.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.

[38] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the ACM/IEEE 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, New York, NY, USA, 335–344. https://doi.org/10.1145/2370816.2370865

[39] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/3020078.3021744

[40] Kizheppatt Vipin and Suhaib A Fahmy. 2014. ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. *IEEE Embedded Systems Letters* 6, 3 (2014), 41–44.

[41] Xilinx. 2015. Zynq-7000 All Programmable SoC and 7 Series Devices Memory Interface Solutions v2.3 User Guide. (June 2015). http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v2_3/ug586_7Series_MIS.pdf

[42] Xilinx Inc. 2014. The Xilinx SDAccel Development Environment. (2014). https://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-backgrounder.pdf

[43] Xilinx Inc. 2015. Vivado Design Suite: AXI Reference Guide. (2015). http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[44] Xilinx Inc. 2017. MicroBlaze Soft Processor Core. (2017). https://www.xilinx.com/products/design-tools/microblaze.html