A Survey on Coarse-Grained Reconfigurable Architectures from a Performance Perspective

1st Artur Podobas
Center for Computational Science
RIKEN
Kobe, Japan
artur@podobas.net

2nd Kentaro Sano
Center for Computational Science
RIKEN
Kobe, Japan
kentaro.sano@riken.jp

3rd Satoshi Matsuoka
Center for Computational Science
RIKEN
Kobe, Japan
matsu@acm.org

Abstract—With the end of both Dennard’s scaling and Moore’s law, computer users and researchers are aggressively exploring alternative forms of compute in order to continue the performance scaling that we have come to enjoy. Among the more salient and practical of the post-Moore alternatives are reconfigurable systems, with Coarse-Grained Reconfigurable Architectures (CGRAs) seemingly capable of striking a balance between performance and programmability.

In this paper, we survey the landscape of CGRAs. We summarize nearly three decades of literature on the subject, with particular focus on premises behind the different CGRA architectures and how they have evolved. Next, we compile metrics of available CGRAs and analyze their performance properties in order to understand and discover existing knowledge gaps and opportunities for future CGRA research specialized towards High-Performance Computing (HPC). We find that there are ample opportunities for future research on CGRAs, in particular with respect to size, functionality, support for parallel programming models, and to evaluate more complex applications.

Index Terms—Coarse-Grained Reconfigurable Architectures, CGRA, FPGA, Computing Trends, Reconfigurable systems

I. INTRODUCTION

With the end of Dennard’s scaling [1] and the looming threat that even Moore’s law [2] is about to end [3], computing is perhaps facing its most challenging moments. Today, computer researchers and practitioners are aggressively pursuing and exploring alternative forms of computing in order to try fill the void that an end of Moore’s law would leave behind. Today there are a plethora of emerging technologies with the promise of overcoming the limits of technology scaling, such as quantum- or neuromorphic-computing [4], [5]. However, not all Post-Moore architectures are intrusive and some merely require us to step away from the comforts that von-Neumann architecture offer. Among the more salient of these technologies are reconfigurable architectures [6].

Reconfigurable architectures are systems that attempts to retain some of the silicon plasticity that an ASIC solution usually throws away. These systems – at least conceptually – allow the silicon to be malleable and its functionality dynamically configurable. A reconfigurable system can for example mimic a processor architecture for some time (e.g. a RISC-V core [7]), and then be changed to mimic a LTE baseband station [8]. This property of reconfigurability is highly sought after, since it can mitigate the end of Moore’s law to some extent– we do not need more transistors, we just need to spatially configure the silicon to match the computation in time.

Recently, a particular branch of reconfigurable architecture – the Field-Programmable Gate Arrays (FPGAs) [9] – has experienced a surge of renewed interest for use in High-Performance computing (HPC), and recent research has shown performance- or power-benefits for multiple applications [10]–[14]. At the same time, many of the limitations that FPGAs have, such as slow configuration times, long compilations times, and (comparably) low clock frequencies, remain unsolved. These limitations have been recognized for decades (e.g. [15]–[17]), and have been used to drive forth a different branch of reconfigurable architecture: the Coarse-Grained Reconfigurable Architecture (CGRAs).

CGRAs trade some of the flexibility that FPGAs have to solve their limitations. A CGRA can operate at higher frequencies, can provide higher theoretical compute performance, and can drastically reduce compilation times. While CGRAs have traditionally been used in embedded systems (particularly for media-processing), lately, they too are considered for HPC. Even traditional FPGA vendors such as Xilinx [18] and Intel [19] are creating and/or investigating to coarsen their existing reconfigurable architecture to complement other forms of compute.

In this paper, we survey the literature of CGRAs, summarizing the different architectures and systems that have been introduced over time. We complement surveys written by our peers by focusing on understanding the trends in performance that CGRAs have been experiencing, providing insights into where the community is moving and any eventual gaps in knowledge that can/should be filled.

The contributions of our work are as follows:

- A survey over three decades of Coarse-Grained Reconfigurable Architectures, summarizing existing architecture types and properties,
- A quantitatively analysis over performance metrics of CGRA architecture as reported in their respective seminal papers, and
- An analysis on trends and observations regarding CGRAs with discussion
The remaining paper is organized in the following way. Section II introduces the motivation behind CGRAs, as well as their generic design for the unfamiliar reader. Section III positions this survey against existing surveys on the topic. Section IV quantitatively summarizes each architecture that we reviewed, describing key characteristics and the premise behind respective architecture. Section V analyzes the reviewed architecture from different perspectives (Sections VII, VIII, and VI), which we finally discuss at the end of the paper in section IX.

II. INTRODUCTION TO CGRAS

Before summarizing the now three-decades of Coarse-Grained Reconfigurable Architecture (CGRA) research, we start by describing the main aspirations and motivations behind them. To do so, we need to look at the CGRAs predecessor: The Field-Programmable Gate Array (FPGA).

FPGAs are devices that were developed to reduce the cost of simulation and developing Application-Specific Integrated Circuits (ASICs). Because any bug/fault that were left undiscovered post ASIC tape-out would incur a (potentially) great economical loss, FPGAs were (and still are) crucial to digital design. In order for FPGAs to mimic any digital design, they are made to have a large degree of fine-grained reconfigurability. This fine-grained reconfigurability was achieved by building FPGAs to contain a large amount of on-chip SRAM cells called Look-Up Tables (LUTs) [20]. Each LUT was interfaced by few input wires (usually 4-6) and produced an output (and its complement) as a function of the SRAM content and their inputs. Hence, depending on the sought-after functionality to be simulated, LUTs could be configured and – through a highly reconfigurable interconnect – could be connected to each other finally yield the expected designs. The design would naturally run one to three order of magnitude lower than the final standard-cell ASIC, but would nevertheless be an invaluable prototyping tool.

By the early 1990s, FPGAs had already found other uses (aside from digital development) within telecommunication, military, and automobile industries—the FPGA was seen as a compute device in owns right and there was some aspiration of use it for general-purpose computing, and not only something used in the niche market of prototyping digital designs. Despite this, several limitations of FPGAs were quickly identified that prohibited coverage of a wide range of applications. For example, unlike software compilation tools that take minutes to compile applications, the FPGA Electronic Design Automation (EDA) flow took significantly longer, often requiring hours or even days of compilation time. Similarly, if the expected application could not fit one a single device, the long reconfiguration overhead (the time it takes to program the FPGA) demotivated time-sharing or context-switching of its resources. Another limitation was that some important arithmetic operators did not map well to the FPGA; for example, a single integer multiplication could often consume larger fraction of the FPGA resources. Finally, FPGAs was relatively slow, running at a low clock frequency. Many of these challenges and limitations of applying FPGAs for general-purpose computing holds to this day.

Many early reconfigurable computing pioneers looked at the limitations of FPGAs and considered what would happen if one would increase the granularity at which it was programmed? By increasing the granularity, larger and more specialized units could be built, which would increase the performance (clock frequency) of the device. Also, since the larger units require less configuration state, reconfiguring the device would be significantly faster, allowing fine-grained time-sharing (multiple contexts) of the device. Finally, by coarsening the units of reconfiguration, one would include those units that maps poorly on FPGAs into the fabric (e.g. multiplications), making better use of the silicon and increasing generality of the device. These new devices would later be called: Coarse-Grained Reconfigurable Architecture (CGRAs).

An example of what a CGRA looks like from the architecture perspective is shown in Figure 1. In Figure 1:a we see a mesh of reconfigurable cells (RCs) or processing element (PEs), which is the smallest unit of reconfiguration that perform work, and it is through this mesh that a user (or compiler) decides how data flows through the system. There are multiple ways of bringing data in/out to/from the fabric. One common way is to map the device in the memory of a host processor (memory-mapped) and have the host processor orchestrate the execution. A different way is to include (generic-) address generations (AGs) that can be configured to access external memory using some pattern (often corresponding to the nested loops of the application) and push it through the array. A third option is to have the re-configurable cells do both the computation and address generation. Figure 1:b illustrates the internal of a RC element, which includes an ALU (integer and/or floating-point capable), two multiplexers (MUXs), and a local static RAM (SRAM) used for storage. The two multiplexers decide which of the external inputs to operate on. The inputs are usually the output of adjacent RCs, the local SRAM scratchpad, a constant, or a previous output (e.g. for accumulations). The output of the ALU is similarly connected to adjacent RCs, local SRAM, or back to one of the MUXes. Operation of the RC is governed by a configuration register, here briefly shown in Figure 1:c. For simplicity, we show a single register that hold the state—however, in many architectures, each RC can hold multiple configurations that are cycled through over the application lifetime. Each of the configuration can for example hold the computation for a particular basic block (where live-out variables are stored in SRAM) or discrete kernels.

Figure 1 illustrates how a majority of today’s CGRAs look like, but at the same time there are multiple variations. For example, early CGRAs often included fine-grained reconfigurable elements (Look-Up Tables, LUTs) inside the fabric. While the mesh topology is by far the most common, some work chose a ring or linear-array topology. Finally, the flow-control of data in the network can be of varying complexity (e.g. token or tagged-token). We describe many of these in our summary in the sections that follows.
A Generic CGRA design

![A Generic CGRA design](image)

- incoming data-paths
- outgoing data-paths

a) Mesh of RCs

b) Reconfigurable Cell

III. MOTIVATION BEHIND PRESENT SURVEY

Since their inception in early 1990s, CGRAs have been the subject of a plethora of research on their architecture, compilation strategies, mapping, and so on and forth. At the same time, surveys has closely monitored how the CGRA technology have evolved through time, and we can today enjoy solid and condensed material on the subject. Survey have covered most aspects of CGRA computing, including commercial CGRA adaptation [21], architectures [22], [23], tools and frameworks [24], and taxonomy/classification [25], [26].

The work in the present paper assumes a different position to surveying the field of CGRAs. Our paper complements the existing literature by attempting to summarize and condense the performance trends of CGRA architectures, and position these against architectures such as GPUs (which is what most systems use as accelerators) in order to understand what gaps future high-performance CGRA should strive to fill. To the best of our knowledge, this is the first survey providing such a comprehensive perspective within the field of CGRAs.

IV. SUMMARY OVER CGRA ARCHITECTURE RESEARCH

A. Early Pioneering CGRAs

Some early CGRAs were not much coarser than their respective FPGAs. For example, the GARP [17] infrastructure was reconfigurable at a 2-bit (rather than FPGA 1-bit) granularity. Here, each reconfigurable unit could connect to neighbors in both the horizontal (used for carry-outs) and vertical direction, as well as to dedicated bus lines for interfacing memory. By using several reconfigurable units along the horizontal axis, users could implement arithmetic operations of varying size (e.g. 18-bit additions). The arithmetic units created along the horizontal directions could have their output connected along the vertical direction, creating a computational data-path. An external processor (the MIPS-based [27] TinyRISC in GARP case) could then orchestrate the execution of this data-path. Using the CGRA as a co-accelerator was (and still is) a pronounced way of using them. The GARP project spanned several years, and included the development of a C compiler [28].

CHESS [29] – unlike GARP – operated on a reconfigurability width of 4-bits. CHESS, as the name implies, layout the individual reconfigurable elements in a fairly uniform mesh, where elements of routing and elements of compute is altered across the mesh. Here, each reconfigurable compute elements had access to all eight of its neighbors. Unlike GARP, whose reconfigurable elements were built around lookup tables (as FPGAs), CHESS used ALU-like structures with fixed functionalities that a user could chose (or configure) to use. Another interesting feature was that the compute elements could be reconfigured to act as (limited) on-chip scratchpads.
D-FABRIX [30] was based on CHESS and was taped-out as a commercial product.

RAW [16], [31], [32] takes a different approach to CGRAs. Rather than keeping the reconfigurable tiles minimalistic, it instead chose to make them software programmable. Where-as GARP was based on LUTs, and CHESS was based on a single 64-bit configuration register per RC, RAW RCs have a fully dedicated instruction memory and highly dynamic network-on-chip, along with necessary hardware to support it. In fact, the RAW architecture is very similar to the modern many-core architecture, albeit lacking shared-memory support such as cache coherency. RAW spanned several years, and had a mature software infrastructure and prototype chips was taped out in 2004 [33]. It was also the precursor to the modern many-core architecture TILERA [34], which was partially built on the outcome of RAW.

The REMARC [15], [35] architecture is an early – at the time quite coarse – architecture that operates on a 16-bit data-path. It is quite similar to modern CGRAs, since the reprogrammable elements all include an ALU, a small register-file, and are directly connected to their nearest neighbors in a mesh-like topology. Configuring the CGRA was done by programming the instruction RAM that was local to each tile with some particular functionality, where a global program counter (called nano-PC) synchronously orchestrated (or sequenced) the execution. Global communication wires run across the horizontal and vertical axis, allowing elements to communicate with external resource. As with GARP – but unlike the RAW – the REMARC architecture was designed to work as a co-processor.

Another early but influential CGRA was the MATRIX [36] architecture, which (similar to REMARC) revolved around ALUs as the main reconfigurable compute resource, but was slightly more fine-grained than REMARC due to choosing an 8-bit (contra REMARCs’ 16-bit) data-path. Despite their name, the ALUs functionality was actually more similar to an FPGAs, where a NOR-plane could be programmed to desired functionality (similar to a Programmable Logic Array, PLAs), but did also include native support for pattern matching. The MATRIX, for its time, had a remarkably advanced network topology, where compute elements could directly communicate with neighbors on a two-square Manhattan distance. Additionally, the network included by-pass layers for remote compute elements to communicate. The network also supports computing on the data that was routed, including both shift- and reduction operations.

The MorphoSys [37], [38] was similar to the REMARC architecture, both in structure, granularity (16-bit), and also in the application domains they target (media applications), designed to act as a co-processor. Another early CGRA that used the (today) well-known structure of CGRAs, MorphoSys had an ALU, a small register file, an output shifter (to assist fixed-point arithmetic) and two larger multiplexers driven by the outputs of neighbors. The compute elements are arranged hierarchically in two-layers: the first is a local quadrant where elements have access to all other compute elements along the

---

**Fig. 2:** Three well-known early CGRA architecture that represents different approaches to the concept, where (a) GARP represents RCs with fine granularity (1-2bits), (b) MorphoSys use the structure commonly found in modern CGRAs, and (c) RaPiD adopted a linear-array of heterogeneous units that are connected through a shared segmented bus.
vertical and horizontal axis, and the second layer are four quadrants composed into a mesh. Unlike previous CGRAs, MorphoSys had a dedicated multiplier inside the ALUs. A CGRA based on MorphoSys was also realized in silicon nearly seven years from its inception [39].

While most of the CGRA described so-far used a mesh topology of interconnection (with some connectivity), other topologies have been considered. RaPiD [40], [41] is an CGRA that arranged its reconfigurable processing elements in a single dimension. Here, each processing element is composed of a number of primitive blocks, such as ALUs, Multipliers, scratchpads, or registers. These primitive blocks are connected to each other through a number of local, segmented, tri-stated bus lines that can be configured to form a data-path—a so-called linear array. These processing elements can themselves be chained together to form the final CGRA. Interestingly, RaPiD can be partial reconfigured during execution in what the authors called “virtual execution”. RaPiD itself does not access data; instead, a number of generic address-pattern generator interface external memory and stream the data through the compute fabric.

The KressArray [42]–[44] was one of the earliest CGRA design to be created, and the project spanned nearly a decade with multiple versions and variants of the architecture. It features a hierarchical topology, where the lowest tier was composed of a mesh of processing elements. The processing elements interfaced neighbours, and also included predication signals (to map if-then-else primitives). Generic address generators supported the CGRA fabric by continuously streaming data to the architecture.

Chimaera [45] is a co-processor conceptually similar to GARP, with an array of reconfigurable processing elements operating at a quite fine granularity (similar to modern FPGAs) can be reconfigured to perform a particular operation. It is closely coupled to the host processor to the point where the register file is (in part) shadowed and shared. Mapping application to the architecture was assisted by a “simple” C compiler, and they demonstrated performance on Mediabench [46] and Honeywell [47].

PipeRench [48] applied a novel network topology that was a hybrid between that of a mesh and a linear array. Here, a large number of linear arrays were layered, where each layer sent data uni-directionally to the next layer. Several future CGRA would adopt this kind of structure, including data-flow machines (e.g. TARTAR) and loop-accelerators (e.g. FPCA). The layers themselves in PipeRench were fairly fine-grained and comparable to GARP as they had reconfigurable Look-Up Tables rather than fixed-function ALUs within. PipeRench introduced a virtualization technique that treated each separate layer as a discrete accelerator, where a partial reconfiguration traveled alongside with its associated data, reconfiguring the next layer according to its functionality in a pipeline fashion, which was new at the time. PipeRench was also later implemented in silicon [49].

The DREAM [50] architecture was explicitly designed to target (then) next-generation 3G networks, and argues that CGRAs are well suited for the upcoming standard with respect to software-defined radio and the flexibility to hot-fix bugs (through patches) and firmware. The system has a hierarchy of configuration managers and a mesh of simple, ALU-based, RPEs operating on 16-bit operands and with limited support for complex operations such as multiplications (since operations were realized through Look-Up Tables).

So-far, all architecture has been computing using integer arithmetic’s. Imagine [51] is among the early architectures that included hardware floating-point arithmetic units. The architecture itself is similar to RaPiD—it is a linear array, where each processing elements has number of resource (scratchpads, ALUs, etc.) all connecting using a global bus. Similar to RaPiD, the processing elements are passive, and external drivers are responsible for streaming data along the connect processing elements. The Imagine architecture had a prototype realized six years after its seminal paper [52].

1) Modern Coarse-Grained Reconfigurable Architectures:

Most modern CGRA architecture’s lineage can be linked back to those described in the previous section, and a majority of these architecture follows the generic template that was described in the previous section. However, while the overall template remains similar, many recent architectures specialize themselves towards a certain niche use (low-power, Deep-Learning, GPU-like programmable, etc.)

The ADRES CGRA systems [53], [54] has been a remarkably successful architecture template for embedded architectures, and is still widely used. ADRES – like many previous and future CGRAs – consists of a mesh of processing elements where each element has neighbor (or Manhattan distance-2) connectivity. Inside each element we find an ALU of varying capability and a register file, alongside the multiplexers configured to bring in data from neighbours. The first row in the mesh, however, is unique, as it only contains the ALU (and no scratchpad/RF to store state). Instead, an optional processor can extend its pipeline to support interfacing that very first row in a Very Long Instruction Word [55] (VLIW) fashion. ADRES, by design, is thus heterogeneous. ADRES comes with a compiler called DRESC [56]. ADRES as an architecture has (and still is) a popular platform for CGRA research, such as when exploring multi-threaded CGRA support [57], topologies [58], asynchronous further-than-neighbor communication(e.g. HyCube [59]), or CGRA designs frameworks/generators (e.g. CGRA-ME [60], [61]). Furthermore, ADRES has been taped out in silicon, for example in the Samsung Reconfigurable Processor (SRP) and the follow-up UL-SRP [62] architecture.

The Dynamically Reconfigurable ALU Array (DRAA) [63] is a generic CGRA template proposed in 2003 to encourage compilation research on CGRA architecture. Architecture-wise, DRAA allows changing many of the parameters that define an CGRA, such as the data-path width, the interconnections, size of register file, etc. Preceding both DySER and ADRES, DRAA as a template has been used to e.g. study the memory hierarchy of CGRAs [64].

The TRIPS/EDGE [65], [66] microarchitecture was a long-
running influential project that attempted to move away from the traditional approach of exploiting instruction-level parallelism in modern processors. The premise behind TRIPS was that as technology reduced the sizes of transistors, wire-delays and path would dominate latency, and that it would hard to scale the (often) communication wires traditional super-scalar processors [67]. Instead, by tightly coupling functional units in (for example) a mesh, direct neighbor communication could easily be scaled. In effect, TRIPS/EDGE replaced the traditional super-scalar Out-of-Order pipeline with a large CGRA array: single instructions were no longer scheduled, but instead a new compiler [68], [69] was developed that scheduled entire blocks (“CGRA configurations”) temporally on the processor, allowing up-to 16 instructions to be executed at a single time (and many more in-flight). The TRIPS architecture was taped out in silicon [70], [71] and – despite being discontinued – represented a milestone of true high-performance computing with CGRAs. An interesting observation, albeit not necessarily related to CGRAs, was that the Edge ISA has received renewed interest as an alternative to express large amounts of ILP in FPGA soft processors [72].

The DySER [73] architecture integrates a CGRA into the backend of a processor’s pipeline to complement (unlike e.g. TRIPS that replace) the functionality of the traditional (super-)scalar pipeline, and has been integrated in the OpenSPARC [74] platform [75]. The key premise behind DySer is that there are many local hot regions in program code and higher performance can be obtained by specializing on accelerating these inside the CPU. DySER was evaluated using both simulator-based (m5 [76]) and an FPGA implementation on well-known benchmarks (PARSEC [77] and SPECint) and compared with both CPU and GPU approaches, showing between 1.5x-15x improvements over SSE and comparable flexibility and performance to GPUs. Recently (2016 onwards), DySER has been the focus of much of the FPGA-overlay scene (see Section IV-F). Other, similar work to DySer that integrates CGRA-like structures into processing cores with various goals, includes CRReAMs/HARTMP [78], [79] (applies dynamic binary translation) or CGRA-sharing [80] (conceptually similar to what AMD Bulldozer architecture [81] and Ultrasparc T1/T2 did with their floating-point units).

The AMIDAR [82] is another long running exciting project that (amongst others) uses CGRA to accelerate performance critical sections. The AMIDAR CGRA extends the traditional CGRA PE architecture with direct interface to memory (through DMA). There is support for multiple contexts and hardware support for branching (through dedicated condition-boxes operating on predication signals), which also allow speculation. The AMIDA CGRA has been implemented and verified on a FPGA platform, and early results show that it can reach over 1 GHz of clock frequency when mapped to a 45 nm technology.

The MORA [83] architecture is a platform for CGRA-related research. MORA targets media-processing, and hence provide an 8-bit architecture with processing elements covering the most commonly used operations. MORA itself is similar to the previous MATRIX, with a simple 2D mesh structure with neighbour communication. Each processing element has a scratchpad (256 bytes large). MORA is programmable using a domain-specific language developed over C++ [84].

The CGRA Express [85] is yet another architecture that follows the concept of being a mesh with simple, ALU-like structures. The premise and motivation for their work is that most existing CGRA application are optimized for maximal graph coverage rather than sequential speed. The hypothesis is that – depending on the operators each PEs is configured to use – they can exploit the resulting positive clock slack of the operators and cascade (fuse) more operations per clock cycle than blindly registering the intermediate output. This, in turns, allow them to execute more instructions per cycles (or reduces the frequency) with little performance losses. In their architecture, they add an extra bypass network that can be configured to not be pipelined. They show both power and performance benefits on multimedia benchmarks with and without their approach. The work could be conceptually seen as the opposite to what modern FPGAs (e.g. Stratix 10) does with HyperFlex [86], but for CGRAs.

The Polymorphic Pipeline Array (PPA) [87] performed an interesting pilot study that drove the parameters of their CGRA: they simulated a large number of benchmarks scheduled on a hypothetical (infinite) CGRAs, with focus on modulo-scheduling and loop unrolling. They revealed that even with infinite larger CGRAs, the performance levels will be bound as a function of the instruction-level parallelism in the loops and the limitation of modulo-scheduling, and they argue that there is a definitive need to include other forms of parallelism to scale on CGRAs. While the PEs themselves follows a standard layout, they propose an interesting technique that allows multiple (unique) kernels to be executed concurrently on the CGRA, where each kernel communicate with each other either through DMA or shared memory. Kernels can also be resized to fully exploit the CGRA array.

The premise behind SIMD-RA [88] is similar to that of PPA: CGRAs relies too much on instructional-level parallelism, and opportunities from other forms of parallelism are lost. SIMD-RA focuses on embedding support to modularize the CGRA-array to supporting multiple discrete controllable regions that (may) operate in SIMD fashion. They found that using SIMD not only yielded better performance, but were also more area efficient compared to only using software-pipelining.

SmartCell [89] is a CGRA that aspires to be low-power with high-performance, supporting both SIMD- and MIMD-type parallelism. The architecture is effectively a 2D mesh, but with the mesh divided into 2x2 quadrants of processing elements. These 2x2 islands share a reconfigurable router and inter-quadrant communication is limited to the connectivity of these routers. The processing elements themselves are fairly standard and contain instruction memory whose instruction (configuration) is set either per processing element (MIMD) or sequenced globally (SIMD).
BilRC [90] is a heterogeneous mesh composed of three different blocks: generic ALU blocks, Multiplication/Shifter nodes, and memory blocks, following the (by now) traditional recipe of a CGRA. Unique to BilRC is that the architecture explicitly exposes the triggering of instruction, allowing the programmer and/or application fine-grained control over the amount of parallelism or when instructions are triggered.

The lack of floating-pointer support in CGRAs has also been a research driving force. FloRA [91] is 16-bit IEEE-754 floating-point capable CGRA. The architecture itself is composed of 64 RCs, and each RC is fairly standard and do not include dedicated floating-point cores themselves; instead, multiple (2) RCs can be combined to enable single-precision (32-bit) floating-point support, where mantissa- and exponent-computation is distributed among the pair.

Feng et al. [92] introduce a floating-point capable architecture specifically designed specifically radar signal processing. Despite the familiar mesh-based interconnection, the design deviates from the traditional approach since their processing elements are fairly diverse and heterogeneous. The CGRA itself was taped out in silicon and could reach up to 70 GFLOP/s floating-point performance.

PRET-driven (Precision Timed) CGRA aimed towards predictable real-time processing was developed by Siqueira et al. [93]. Interestingly, the CGRA has support for threads, which is a concept used more in High-Performance rather than Real-Time computing. The architecture is similar to what a SIMT (Simultaneous Multi-Threading) architecture looks like, where each processing element has a duplicate number of resource (primarily the register files) that are unique to each thread. Aside from having deterministic timing inside the CGRA, the authors also implement a predictable external memory access timing, required for real-time systems.

The recent SPU [94] architecture aspires to provide a CGRA for general-purpose computing. The main novelty is that SPU extends existing CGRA designs with support for two types of computational patterns: what they call "stream-joins" (e.g. sparse vector multiplication inner-product) and alias-free scatter/gather (regular loops with indirection). This is achieved by extending the typical CGRA with options to conditionally consume input tokens (re-use values), reset accumulators, or conditionally discard output tokens. Address-Generation units (linear and non-linear) reside inside on-chip SRAM controllers. The SPU targets general-purpose workloads with some favor towards deep-learning applications.

The eXtreme Processing Platform [103] (XPP) was a CGRA that focused on multiple levels of parallelism, including that of pipeline processing, data-flow computing, and task-level execution. XPP's interconnection was deep and complex, consisting of multiple levels of various functionality. At the lowest tier, small processing elements containing scratchpad, an ALUs, and an associated configuration manager reside in mesh-like connectivity called a cluster. These clusters themselves are connected through switch-boxes running along their vertical and horizontal axes. Each tier has a configuration manager that is responsible for all functionality of that layer (and below), allowing fine-grained control and partitioning of the functionality of the system. XPP was token-driven, and execution of operation occurs only when data is present at inputs.

The High-Performance Reconfigurable Processor [104] (HiPreP) is an ongoing CGRA research platform capable of floating-pointer computations. HiPreP has dedicated floating-pointer circuitry (unlike e.g. FloRA). Processing elements are organized in a mesh with a heterogeneous (in terms of
bandwidth) interconnect, and include address-generations units for driving data through the device. The HiPreP explicitly targets high-performance computing.

WaveScalar [105]–[107] was an exciting architecture that focused on whole-application mapping onto token-driven, data-flow CGRA-like architecture. Most CGRA systems limit the size of the CGRA fabric to the point where scheduling basic-blocks (instructions without branches) have to be split in temporally across the fabric. WaveScalar takes a different approach: do not limit the CGRA fabric; instead, create a large fabric and remove all materialization of control-flow in the application by converting them to data-dependencies, enabling them to map to the architecture. The WaveScalar architecture is larger, with over 2000 RCs, and – as with its rival TRIPS – was thoroughly evaluated on a healthy number of benchmarks.

Another hierarchical architecture is Tartan [108]. Tartan is a mix between a traditional mesh-based topology (e.g. ADRES) and a layered linear array (e.g. PipeRench) and focus on larger applications of SpecINT [109]. The system is fine-grained and operates on 8-bit data-path with primarily addition and logical operations (no native multiplication). However, the architecture is not easily reconfigured, and expects the full application (or the majority of it) to be fully placed onto the architecture to prevent any form of context-switching. Tartan itself represents the extreme case where the majority of silicon is spent on simple compute elements. The EGRA [110] architecture is similar in topology to Tartan, and is a mesh where each node consists of layered programmable ALUs. Both Tartan and EGRA were evaluated primarily using software simulators (SimpleScalar [111] and in-house simulator for Tartan).

While most CGRAs target a classic SISD architectural model – with a few exceptions for SIMD – the SGFM [112], [113] architecture instead focuses on the SIMT [114] (Single-Instruction Multiple-Thread) model that is commonly found in Graphics Processor Units (GPUs) and programmed using languages such as CUDA [115] or OpenCL [116]. The SGFM is claimed to be similar in area to the NVIDIA Fermi [117] or Kepler architecture. The architecture itself is a mesh-like CGRA with thread-tagged token flow-control and with support for synchronization. Although there are limits on what CUDA constructs can be mapped (e.g. atomic operations is not supported), the architecture itself is shown through simulation to be a viable and competitive alternative to existing GPUs.

Remus [118] is a relatively large – for embedded SoC standards – CGRA with 512 processing elements that are driven by two ARM processors. Remus is fairly standard in terms of layout and use a layered mesh, although with extra temporary registers capable of holding state along the horizontal lines of the mesh, increasing opportunities for routing and more aggressive pipelining (to increase operating frequency).

The Dynamically Reconfigurable Processor [119] (DRP) is a CGRA that target stream-based computation. The processing elements are divided into 8 tiles each containing 8x8 processing elements, where processing element has an ALU, input selectors (multiplexers) and an instruction memory (for
multiple contexts). A number of scratchpad memories sits at the fringes of each tile and are used to store streamed data. Controlling the operation of the processing elements is done through an instruction pointer that is governed by hierarchical sequencers (one per tile and one global). The sequencer – effectively a programmable FSM – dictates which context is being executed, and can (re)act on signals from the tiles themselves. Similarly, inside the memory units, we find a small set of ALUs coupled with programmable logic to interface these units. Similarly, inside the compute units we find the raw functional units (the ALUs) as well as programmable state for controlling them.

Inside the compute units we find the raw functional units (the ALUs) as well as programmable state for controlling them. The tiles are interconnected using horizontal and vertical busses that run in-between and through the mesh, and crossing the quadrants can only be done at border tile locations. Performance of selected applications (FIR, FFTs, Image processing) show two orders of magnitude better performance over the then state-of-the-art Pentium 4 processor.

The 167-processor architecture [121] borders C GRAs and conventional multi-core processors, but we include it here since the processing elements are simple and communication between them is only performed using direct (yet dynamically configured) connectivity (and not through shared-memory or cache coherence as done in multi-core). The main focus behind this work is low-power, and through a series of advanced low-level optimizations (DVFS, clock generation and distribution, GALS [122] etc.) They show performance of up to 196.7 GMAs/Watt when fabricated in 65 nm technology. Other similar architecture, based on programmable cores with limited connectivity, are the IMAPCAR [123]/Imap-CE CGRA [124] from NEC aimed towards image recognition in automobiles.

The Rhyme-Redefine [125], [126] architecture is a CGRA targeting High-Performance Computing (HPC) kernels. It follows a fairly typical CGRA design, where processing elements are connected through a torus network. The premise of their work is that there is a need to exploit multiple levels of parallelism (instruction-, loop- and task-level parallelism), albeit the current implementation focuses primarily on instruction-level parallelism through modulo-scheduling. The Rhyme-Redefine supports floating-point computations.

Plasticine [127] is a recent, large CGRA that focus on parallel patterns. At the highest abstraction layer, it is built of a mesh of units. There are two types of units: compute and memory units, both of which are programmable with patterns. Inside the compute units we find the raw functional units (the ALUs) as well as programmable state for controlling them. The compute units are built with both SISD- and SIMD-type parallelism in mind, and vector operations map natively to these units. Similarly, inside the memory units, we find a small set of ALUs coupled with programmable logic to interface the SRAM local to the units. The mesh itself interfaces external memory through a set of address generators and coalescing units. More importantly, Plasticine targets floating-point intensive applications, which is also shown in their evaluation (only three out of 13 applications are integer-only). Plasticine is programmable using Spatial [128] – a custom language based on patterns for data-flow computing.

Recently, the Cerebras Wafer Scale Engine [129] has been created explicitly for high-end deep-learning training. Little information is publicly available, but the architecture seems to be a hybrid solution between general-purpose processing code and specialized tiles for tensor computations, and could make it the single largest CGRA architecture to date with a size of over 46,225 mm².

C. Linear-Arrays and Loop-Accelerators

VEAL [130] is a linear array that explicitly targets accelerating small, compute-intensive loop-bodies. Similar to a before-mentioned PPA (Polymorphic Pipeline Array), the authors behind VEAL performed a rigid empirically evaluation of the benchmarks they target, and demonstrate that one of the main limitation to the performance of mapping said benchmarks to CGRA fabrics is not the number of resources, but actually the amount of instruction-level parallelism extracted by modulo-scheduling. The VEAL is linear array fed by a number of custom address-generators, which broadly corresponds to the induction variables of the loops that are executed. An interesting observation is that VEAL is among few CGRA work that use double-precision arithmetic’s. Another loop-accelerator similar to VEAL is FPCA [131].

The BERET [132] architecture is yet another linear array that is designed to accelerate hot traces found by the hosting general-purpose processing. One of main BERET’s main contribution was that they identified a small set of graphs that the processing elements should cover (called SEBs); the set was empirically extracted from the benchmark and has since then been used in other studies (e.g. SEED [133], which is similar but improved in concept).

D. Deep-Learning CGRAs

Deep-Learning [134], in particular the computationally regular Convolutional Neural Networks (CNNs), have lately become target for specialized CGRAs. Here the focus is to limit the generality and reconfigurability of traditional CGRA to fit the computational patterns of CNNs, and instead spend the gained logic on supporting specialized operations for the intended deep-learning workloads (such as compression, multicasting, etc.). Furthermore, these architectures often honor smaller (or mixed) number representations, since deep-learning often is amendable to lower-precision calculations [135].

The DT-CGRA [136], [137] architecture follows a CGRA design with fairly coarse processing elements that include up-to three multiple-accumulate instructions. The processing elements also include programmable delay lines to easier map temporally close data. Data inside the PEs is synchronized...
through tokens using FIFO empty/full signals as proxy. To support the different access patterns that modern deep-learning layers have (stride, type, etc.), the CGRA mesh is driven by a number of stream-buffer units that are programmable using in a VLIW-fashion and that control the address-generations to external memory to stream data.

The Sparse CNN (SCNN) [138] is a deep-learning architecture that targets (primarily) CNNs and can exploit sparseness in both activations and kernel weights. The architecture itself is composed of a mesh of RCs, where each element also includes a 4x4 multiplier array and a bank for accumulation registers. These RCs are driven- and orchestrated-by a layer sequencer, which fetches and broadcasts (compressed) weights and activations. SCNN supports inter-PE parallelism in the form of spatial blocking/tiling, where each block is artificially enlarged with a halo region, which is exchanged between adjacent tiles at the end of the computation. They also implement a dense version (DCNN) of the architecture in order to measure the area overhead and power- and performance-gains of including sparsity in the accelerator.

Liang et al. [139] introduce a CGRA accelerator that targets reinforced learning. The processing elements themselves are fairly static, with support for addition, multiplication, or a fuse of both. Additionally, a number of different activation functions (ReLU, sigmoid, and tanh) can be selected using the configuration register, and data can be temporally stored in a local scratchpad. Unlike most existing CGRAs that place address-generators in discrete units outside the RCs, Lian et al.’s RC include them inside. Global communication lines allow the user to control the reinforced training experience of the system.

The Eyeriss deep-learning inference engines [140], [141] follows a CGRA design methodology as well, albeit the specialize on re-configuring the network access-patterns rather than the compute (which mostly is based on multiply-accumulate operations). The CGRA itself is a mesh with a variety of options of point-to-point and broadcast operations, highly suitable for deep-learning convolution patterns. Additionally, the platform supports compression of data and exploits sparseness of intermediate activations to increase observed bandwidth. The Eyeriss architecture – depending on the type of neural network used – can utilize nearly 100% of the CGRA resources when inferring AlexNET.

One of the most recent (and perhaps radical) changes to the FPGAs is coming in the form of support for Deep-Learning CGRAs. The Xilinx Versal [142], [143] series occupies a large part of the silicon to a mesh-like CGRA structure of programmable, neighbor-communicating, processing elements. The elements themselves are fairly general-purpose, but are marketed as targeting deep-learning and telecommunication application. To remedy the eventuality of the AI engine missing crucial parts of deep-learning functionality that has yet to come, the AI engine can directly interface remaining parts of the reconfigurable (FPGA) silicon, which is in the form of the fine-grained reconfigurable cells that Xilinx are known for. The system itself is an attempt to combine the best of both the fine-grained and coarse-grained reconfigurable worlds.

E. Low-Power CGRAs

CGRAs has also been shown to be competitive in terms of power-consumption, particularly when compared to existing (low-power) processors and DSP -engines. The CGRAs in this domain follow the same concept as earlier CGRA design, but focus on both technology and architecture improvements to reduce the static and/or dynamic power of the fabric.

These CGRAs tend to focus on reduction the frequency and voltage as much as possible. Since the dynamic power-consumption of a system is a function of both frequency and voltage ($P_{\text{dynamic}} = C * V^2 * f_{\text{clk}}$), reducing frequency can have a dramatic effect on power-consumption. Several CGRAs in this area operate on near-MHz level, and some even remove the clock altogether.

The Cool Mega Array [144], [145] (CMA-1 and CMA-2) architecture builds on the following two premises: (i) the clock (clock-tree, flip-flops, state, etc.) is a culprit behind much of the consumed power on modern chip, and (ii) applications have adequate parallelism to freely trade silicon for performance where needed. The CMA-1 was a typical CGRA mesh architecture, but created without a single clock. The architecture focuses on stream-computing, where a processor presents inputs to the CGRA that – in due to time – are computed using the clock-less fabric. The architecture (and its follow up, CMA-2) is power-efficient, and experiments on taped-out versions showed that the leakage-current of the chip could be as low as 1 mW. The CMA architecture managers to reach up-to 89.28 GOPS/Watt using a 24-bit data-path. The CMA architecture is continued to be researched, and recent work has focused on improving performance (through variable-latency pipelines in VPCMA [146]) or further reduce power-consumption through body-biasing.

The SYSCORE [147] architecture is another similar architecture that focuses on low-power consumption, but leverage dynamically scaling both voltage and frequency (DVFS) for power-benefits, and using a fixed-point (and not floating-point) number representation. As with CMA-1/2, it is a 24-bit datapath with a standard mesh-like arrangements of CGRA-tiles.

Lopes et al. [148] evaluated a standard mesh-like CGRA for use in real-time bio-signal processing. The CGRA they constructed had the additional benefits of being able to power-down sections of the CGRA when unused to further extend battery-life. Another bio-medical CGRA was introduced by Duch et al. [149] uses a mesh-like composition and a 1 MHz clock-frequency to accelerate electrocardiogram (ECG) analysis kernels.

The Samsung UL-SRP was designed for bio-medical application. The UL-SRP [62] is based on the ADRES, and featured a hybrid high-power/high-performance mode and a low-power/low-performance mode covered the different needs and use-cases.

The PULP [150] cluster system features a 16 RC mesh to improve performance and energy-consumption for near-sensor
data analytics. The CGRA (called IPA [151]) is very standard in the design and adopt most concepts that we have described so far, with the RCs connected in a torus fashion. RCs are capable of 32-bit operations and feature a discrete power-controller (implementing clock-gating) for reducing energy usage when idle, and the array is capable at running at 100 MHz targeting various image processing kernels.

A different – yet equally interesting – form of power-reduction is to extend the CGRA RCs to support approximate computing. X-CGRA [152] is one such example that showed that adapting approximate computing in kernels that are resilience to error (e.g. image manipulation) can drastically reduce power-consumption by up-to 3.21x compared to exact-methods, with as little as 4% loss in quality.

F. Overlays: CGRAs on-top of FPGAs

Some of the original premises for incepting CGRAs was to build faster – still reconfigurable – hardware accelerators, citing FPGAs as inadequate with respect to programmability, performance, and reconfiguration overhead. While FPGA vendors did indeed remedy parts of these problems (even including floating-point DSPs [153]) into the fabric), problems with compilation times and reprogramming overhead still persist to this day. By the early 2010s, several research groups had started experiment on encapsulating the typical fine-grained resources on FPGAs with CGRA-like structures in hope to include some of their benefits; these architectures came to known as FPGA overlays.

Overlays remedies two large performance and usability problems with FPGAs: compilation times and reconfiguration overheads. It is well-known that compiling a design towards FPGAs is a time-consuming task. Modern FPGAs are larger with many unique characteristics making it non-trivial to place and route on them. Furthermore, as the FPGAs grows, so does also the memory footprint of the synthesis tools, and it not uncommon for tools compiling against a new devices (e.g. Intel Stratix 10) to consume a large portion of system memory, effectively restricting the number of parallel compilation possible at a machine. By coarsening and reducing the number of reconfigurable units, compilation times can be significantly reduced (e.g. nearly three orders of magnitude [154]). Furthermore, a FPGA – once programmed – are often expected to run for a long time, since context-switching incurs a large relatively large (seconds long) overhead. Coarsening the units decrease the size of the configurable state, and multiple contexts can (and are) stored within the same fabric.

QuKu [155]–[157] is one of the very earliest CGRA-like overlays on FPGAs, dating back to 2006, with active research all the way into 2012. It features a mesh of processing elements, each capable of addition or multiplication, with nearest neighbor communication and a token-like network. QuKu – as a prototype – was demonstrated on FIR-filters, Sobel and Laplacian operations, showing improvements over a similar software implementation in the soft-core Microblaze [158] processors or custom circuitry.

Fig. 4: (a) The QuKu overlay was among the earliest CGRAs running on an FPGA, and adopted a simple scheme where RCs connect to adjacent tiles and (an optional) address-generator for reading/writing external memory data. (b) The ZUMA architecture – while not technically a CGRA – was an FPGA-on-FPGA overlay whose RCs mimicked those of the actual FPGA. (c) The Xilinx DSP block is versatile and controllable without reconfiguration, and has been a popular choice to base many CGRA-overlays on.
Not strictly a CGRA, ZUMA [159] is an early effort to virtualize the fine-grained resources of an FPGA using a “virtual FPGA”, for reason of portability, compatibility, and FPGA-like reconfigurability inside of FPGA designs. Similar to a real FPGA, ZUMA discretized the FPGA into logic clusters that contains a crossbar and K-input Look-Up Table with an optional flip-flop capturing the output. Each cluster is connected to a switch-box that can be programmed to route the data around. The area cost of using virtual-FPGA can be as low as 40% more than the barebone FPGA, demonstrating its benefits. Other (even earlier) work was FIRM-core [160], as well some more recent efforts include the vFPGA [161].

Intermediate Fabrics (IF) [154] coarsen the FPGA logic by creating a mesh of computational elements of varying sizes, such as for example multipliers and square root functions; small connectivity boxes (routers) govern the traffic throughout the data-path. IF was evaluated on image processing (stencil) kernels, and overall showed an on average 17% drop in clock frequency against a gain of 700x in compilation time over using the FPGA alone.

The MIN Overlay architecture [162] approach the CGRA design differently: it uses a one-dimensional strip of processing elements whose output is connect to each other through an all-to-all interconnect. Hence, data-flow graphs are spliced and fit onto the linear array, and different parts of the graph are scheduled in time on the array and the interconnect. Different combinations and compositions of the processing elements were evaluated and the clock frequency, for the most part, ran at 100 MHz, competitive to soft processor cores at the time. Other, arguable less configurable, overlays follow a similar one-dimension strip design, such as the VectorBlox MXP Matrix Processor [163]. The FPCA loop-accelerator described earlier was also prototyped on FPGAs. The READY [164] architecture extend the linear array concept further by also having multiple threads running on the overlay.

An example of a layered CGRA overlay for FPGAs is the VDR architecture [165]. Here, computational resources are laid out in one-dimensional strip where each strip is fully-connected to downstream units. Links are unidirectional, and synchronization protocol guides data throughout the data-path. The VDR architecture runs at a clock frequency of 172 MHz, and was shown to be between 3 and 8 times faster compared to the NiosII processor [166] (a well-known soft-core used in FPGA design). Another architecture similar to VDR is the RALU [167].

A flurry of innovative overlays was introduced in the 2012 onwards, all centered around the modern FPGAs Digital Signal Block (DSP). The DSP blocks were originally included to allow the use of expensive operations that do not necessarily map well to FPGAs (e.g. multipliers). DSP blocks have since then continuously evolved to include more diverge (various-size multiplication, accumulation, etc.) or more complex (e.g. single-precision arithmetic [153]) functionality. Some of the vendors (Xilinx) directly exposed the interface to control the different functionality of the DSP blocks to the FPGA fabric, and it was not long before the idea to base CGRAs architecture around said DSP blocks. ReMorph [168] was one of the early architectures to adopt this style of reasoning. Several different architectures have been explored around the concept of DSPs, including various topologies (e.g. trees [169]) or adaptation of existing architectures (e.g. DySer using DSP blocks [170]). The strengths of these architectures lie in their near-native performance, where small overlays built around DSPs can run at 390 MHz (or higher).

Quickdough [171], [172] is a design framework for using CGRA overlays on FPGAs, specifically targeting them to assist CPU in accelerating compute-intensive program code. The overlay itself follows the standard layout with a mesh of processing elements, each containing a small instruction memory that sequences the ALU within the processing element. The mesh can interface external memory by enqueuing requests to an address unit. Unique for the architecture is that the two parts (the address unit and the PE mesh) runs at two distinctly different frequencies.

Most FPGA overlays presented so far focus unique on integer computation, likely because most FPGA overlay work target Xilinx devices, whose DSPs units do not contain hardened floating-point operations. The Mesh-of-ALU [173] is an exception that targets both integer and floating-point computation. The architecture is similar to other mesh-based approaches, but the work demonstrates high (at the time) performance capabilities of FPGAs also in floating-point operations, reaching nearly 20 GFLOPs on a Stratix IV [174] device. Using floating-point processing elements seem to incur a 33% area overhead, yielding a smaller CGRA mesh, and also a (arguably negligible) 13% reduction in clock frequency.

A different overlay architecture that target floating-point operations is the TILT array [175], [176]. The TILT array architecture is very similar conceptually to the MIN overlay. A linear array of processing elements is arranged to communicate with an all-to-all crossbar, which saves state into an on-chip RAM and relaying information to the computation in the next cycle. The authors illustrated the benefits of TILT over High-Level Synthesis (OpenCL) with both comparable performance and improved productivity, reaching operating frequency of up-to 387 MHz on a Stratix V [177].

The URUK [178] architecture takes a different approach on how the ALUs inside overlay should be implemented. Rather than having a fixed-function, URUK leverage partial reconfiguration [179], changing the RCs functionality throughout time.

Finally, tools for automatically creating CGRA overlays for FPGAs are emerging, such as the Rapid Overlay Builder [180] and CGRA-ME [60] that simplify generation and (in the case of CGRA-ME also) compilation of applications and overlays.

An interesting observation is that out of the 14 unique FPGA architecture that we surveyed here, 9 chose Xilinx as the target platform while 5 focused on Intel (then Altera) FPGAs. There seems to be a favoring of Xilinx architectures, which we believe is due to the more dynamic control that Xilinx offers in their DSP blocks compared to Intel. On the other side, Intel DSPs have (starting from Arria 10 onwards)
hardened support for IEEE-754 single-precision floating-point operations, encouraging floating-point heavy architecture to use those systems.

V. CGRA TRENDS AND CHARACTERISTICS

A. Method and Materials

For all previous surveyed and summarized work, we collected several metrics associated with each study. These were:

1) **Year** of publication,
2) **Size of the CGRA** array in terms of unique RCs,
3) **Data-path width** of the CGRA (e.g. MATRIX operates on 4-bit while RaPiD operates on 16-bit),
4) **Clock frequency** of operations ($f_{\text{max}}$) in MHz as reported in the study,
5) **Power consumption** in Watt. For studies that empirically measured this metric, we collected the (benchmark, power) tuple. Otherwise we used what is reported in the study (often the post place-and-route power estimation),
6) **Technology** (in nm) of architecture when either taped out in silicon, or the standard cell library used with the EDA tools,
7) **Area ($\text{mm}^2$)** of the fully synthesized chip as reported in the study. In some cases we had to manually calculate it based on the individual RC size or based on the gates used (after verification with authors). For FPGAs we used the chip (BGA) package size and assumed a chip-to-die ratio of 7:1, as has been reported in [181].
8) **Peak performance**, including peak operations-per-second (OP/s), peak multiple-accumulates/second (MAC/s), Peak Floating-Point Operations/second (FLOPS) as reported in the paper. We differentiate between integer MAC/s and OP/s because some architectures (e.g. EGRA) do not balance them, leading to a large theoretical OP/s but not a proportionally large MAC/s.
9) **Obtained Performance** out of the theoretical peak (%). We used what the authors reported. For those cases where authors did not report obtained performance (e.g. only reported absolute time), we calculated this metric manually where applicable, such as for example when the authors report both the input dimension and the execution time (in seconds or cycles) of known applications such as (non-Strassen) matrix-multiplication, FIR-filters, matrix-vector multiplication, etc.

For item 8-9 we ignored studies that showed relative performance improvements, as it is hard to reason around the performance of a baseline unless explicitly stated. All metrics included have either been directly reported in the seminal publication, have been verified by the authors, or we were confident in our understanding of the architecture to derive them ourselves. We position and related our obtained CGRA characteristics against those of modern GPUs. We used NVIDIA GPUs as references with data collected from [182] and integer performance calculated using methods described in [183].

VI. OVERALL ARCHITECTURAL TRENDS

We start by analyzing data that is associated with time, and how the CGRAs have grown as a function of time.

Figure 5 overviews how CGRAs have changed through time with respect to various metrics. The total number of RCs, as a function of the respective publication year, is shown in Figure 5:a. We see that a majority of CGRAs are quite small (median: 64 RCs) and even smaller for FPGA-based CGRAs (median: 25 RCs). This is in-line with the reasoning that most CGRAs focus on small kernels in the embedded application domain, honoring ILP rather than other forms of parallelism (e.g. thread- or task-level). There are several exceptions to this, such as GARP, which was an early CGRA that used 1/2-bit reconfigurable data-paths and thus needed a large number of RCs to implement various functionality. The other exception is TARTAN, where the author’s largest evaluated version is up to 25,600 RCs, making it likely the largest CGRA ever simulated; this awe-inspiring size was reached by severely restricting the functionality of RCs (e.g. there is no multiplication support). Thirdly, the Plasticine architecture can have up to 6208 RCs of varying sorts.

Figure 5:b shows transistor scaling of CGRAs and NVIDIA GPUs. As expected, the transistor dimensions have continuously grown smaller and smaller, as predicted by Moore. Note however that both FPGAs and GPUs is (on average) one transistor generation ahead of CGRAs, likely due to most CGRAs being developed by academia and thus restricted to those standard cell libraries available at that time (which usually is not the most recent).

Figure 5:c shows the area of the CGRAs as reported either by the ASIC synthesis tools, estimation by authors, or by the final taped-out chip. We also include the full-size of the FPGA die sizes (that FPGA-based CGRAs have access to), and we position these against the die-size of modern NVIDIA GPUs. We can see that the trend of CGRA research is – as with the size of CRGs – to favor smaller CRGs, and the average size of the CRGs are $13.117 \text{mm}^2$ Compared to GPUs, which has monotonically increased their size through time, CRGs have almost done the inverse, and decrease in size. There are two major exceptions: the first is the Imagine architecture, which reported an amazing size of 1000 $\text{mm}^2$ (later 144 $\text{mm}^2$ in the follow-up paper 6 years later)– larger than any CGRA or GPU reported to this day. The other larger architecture is CUDA-programmable the SGMF at 800 $\text{mm}^2$.

Figure 5:d shows how the reported power-consumption of ASIC-based CRGs has grown over time, and is compared to the Thermal-Design Package (TDP) reported for NVIDIA GPUs. The CRGs are experiencing on-average an exponential decrease in power-consumption, which is likely due to smaller standard cell libraries coupled with small CRG size (Figure 5:a,c,d). On the other hand, NVIDIA GPUs continuously consumes more and more power as time goes on (albeit even that is drawing to a halt due to Moore’s law). The highest and most power-consumption CGRA, out of those reporting, is the Plasticine architecture consuming a
maximum of 49 Watt, followed by RAW at 18.2 Watt and IPA at 11.26 Watt. On the opposite side, we find architectures that target bio-signal processing, where power-efficiency is critical. Examples include the 167-core processor (60 MHz version) at 99 mW, SYSCORE at 66.3 mW, CMA-1 at 11.2 mW, and CCSOTB2 at 3.45 mW.

Figure 5e shows how the clock-frequency has changed throughout time for both ASIC and FPGA-based CGRAs, as well as NVidia GPUs. The GPUs have continuously had their clock frequency increased, starting off well-around where CGRAs operated in the early 2000s to the high-frequency devices we see today. The CGRAs, as one would expect (following the trend of previous graphs), took a different path and instead focused on low-power (and thus low-frequency). The average CGRA will run around 200 MHz, with the frequency slowly increasing as a function of better and smaller standard cell libraries. There are but few CGRAs that operate at high-frequency, and these include: the loop-accelerator SEED at 2GHz, the GPU-like SGMF at 1.4 GHz, the 167-core processor (high-frequency version) at 1.17 GHz, and both Plasticity and Wavescalar at 1 GHz. At the opposite edge, we find extremely low frequency such as ULP-SRP at (as low as) 7 MHz and the Bio-CGRA [149] at 1 MHz. The FPGA-based CGRA have less of an opportunity to tune for frequency, as they are often bound
by limitation in the fabric itself; however, it is interesting to see that the operation frequency of FPGA-based CGRAs are rivaling most of the ASIC CGRAs.

Figure 5:f shows the chosen data-path width that CGRAs research tend to adopt. Most architecture adopt either a 16-bit (28%) or 32-bit (56%) data-path width; those targeting 16-bit data-path are usually more tailored towards a specific application, such as telecommunication or deep-learning, while those that target 32-bit (or beyond) is more general-purpose. A few (13.3%) target 8-bit architecture, but often have support for chaining 8-bit operations for 16-bit use. Matrix and GARP target very fine-grained reconfigurability, with 4-bit and 1/-2-bit respectively. Despite this, we can expect future architecture to include more support for low- or hybrid-precision, since it is a reliable way of obtaining more performance while mitigating memory-boundness for applications that permits it.

Figure 5:g shows the power-consumption of CGRAs and GPUs as a function of their respective die sizes. This graph complements the graph in Figure 5:d to show that the low power-consumption of CGRAs is mainly because they are small, with (out of those CGRAs that report both power- and area) only Plasticine coming closer to the trend of GPUs.

Finally, Figure 5:h shows how the individual RC area has grown throughout time, and we see that the size of RCs has been following the technology scaling, and continuously decreased in size. However, when normalizing the CGRAs manufacturing technology to that of 16 nm, we actually noticed a different trend, where the area of individual RCs is increasing, due to incorporating more complex elements (such as wider data-paths, more complex arithmetic units, etc.).

VII. INTEGER AND FLOATING-POINT PERFORMANCE ANALYSIS

Figure 6 overviews data associated with the pure performance of the CGRAs, often when positioned against that of NVidia GPUs.

Figure 6a-f shows the obtained integer performance. Here we distinguish between operations and MAC-based operations in order to reveal architecture which are starved of multipliers. For example, the Tartan CGRA can execute a large amount of operations per unit time, but has no support for multiplications, leading to a very low comparable multiple-add performance. Figure 6a-b show the GARP and Matrix architecture as the sole candidates for low-precision arithmetic, and that while both of these have comparable high performance (for their time), their multiplication (MAC) performance is lacking (in GARP, the overhead was 32x compared to an addition). Figure 7:c shows 8-bit integer performance, which has recently been of interested to the deep-learning inference community, and where next-generation Xilinx Versile architecture will be capable of reaching thousands of GOP/s of 8-bit integer performance. Figure 6d shows 16-bit integer performance, showing a continuous growth over the years. Note how the Tartan architecture claims to reach similar performance levels of the upcoming Xilinx Versile CGRA, despite being more than a decade old. Figure 6:e is a special case, and only a few CGRAs (e.g. Cool-Mega Array-1 and SYSCORE); despite their low visible performance, these devices are actually very power-efficient (see next section for discussion). Finally, 32-bit integer performance is shown in Figure 6:f, where we also included NVidia GPU integer performance for comparison. We see that CGRA has historically been comparable to that of NVidia GPUs, and even FPGAs are becoming a valid way of obtaining integer performance through CGRAs.

Figure 6:g shows the peak floating-point performance that CGRAs reported over the years. The number of floating-point capable CGRAs prohibits us from drawing any reasonable trend-line, unlike the one for GPUs that exponentially grows with years (together with the die-area, see Figure 5:c). However, those CGRAs that do include floating-point units can compete with the performance of modern GPUs—sometimes even outperform them. For example, the Plasticine architecture is capable of delivering 24.6 TFLOP/s of performance, rivaling GPUs from that generation, and the earlier Redefine and SGMF architecture could deliver 500 and 840 GFLOP/s respectively. Even earlier, the Wavescalar architecture was capable of 500 GFLOP/s, which was well ahead of GPUs at that time. At a lower performance, we find architecture such as FloRa (600 MFLOP/s) and the loop-accelerator VEAL (5.4 GFLOP/s).

Figure 6:h shows the distribution over the number of CGRAs that support floating-point versus those that support integer computations. Floating-point support is clearly under-represented, with more than a factor four more CGRAs that support integer computations.

VIII. PERFORMANCE USAGE ANALYSIS

Figure 7a shows the number of instructions-per-cycle (IPC) that applications experienced when executing on different CGRAs. We see that a majority of CGRAs operate in a fairly low performance domain, primarily due to their size, and most execute around 12 IPC (median). There are corner cases, such as the Rhyme-Redefine architecture which aims to explore CGRA in High-Performance Computing reaching 300+ IPC on selected workloads, or the Deep-Learning SDT-CGRA architecture on inference, reaching 172 IPC. Similarly, Eyeriss is capable to occupy 100% of its resources when inferring AlexNET, yielding astounding 700+ IPC. Most FPGA-based CGRA also execute less than 100 IPC; this is primarily since the size of most FPGA CGRAs are rather small (see Figure 5a).

Figure 7:b shows the performance applications experience when running on different CGRA architectures as a function of topology size, where we group CGRA architectures into three groups: small (≤16), medium (16-64) and large (≥64). As is expected, we see that the performance and obtained IPC grows as the architectures become larger, where applications running on small-sized CGRA experience on average 12.60 IPC, 27.69 IPC on medium-sized CGRAs, and large 79 IPC on larger, with outliers being capable of reaching much more. A complementary graph is seen in Figure 7:c, where we see the obtained performance as fraction of the raw peak performance.
The graph reveals that reaching peak-performance becomes harder as the CGRA architecture grows in size, where small architecture commonly reaching 25% and more of peak while large architecture reaching on average 17%, and medium-sized architecture somewhere in-between. Most of these architectures, as we will discuss later, rely primarily on using software-pipelining as their prime source of parallelism, which might not necessarily be able to fill architectures to their max.

Figure 7:d shows compute densities and how they have evolved throughout CGRA research history. For comparison, we use NVIDIA GPU performance for 32-bit integer and floating-point is included. Overall, the compute densities of CGRAs are in-fact comparable to GPUs, and often even surpasses. For example, most CGRAs that support 32-bit integer operations consistently packs more compute per \( mm^2 \) than GPUs. This also holds true for some CGRAs with floating-point support, such as VEAL and Plasticine. Overall it make sense that CGRAs pack more compute per unit area, since from an architectural perspective a CGRA trades much of the silicon used for orchestration (instruction-memory, caches, control-planes,...) for more compute. The compute densities correlate with the transistor size, and the more recent architecture have a more clear advantage. Note that CGRAs are likely to have an even higher benefit than what the graph shows since the transistor generation usually lags behind one generation compared to commercial GPUs, likely yielding an even higher benefit.

Figure 7:e shows how many operations that can be performed per unit power (OPS/Watt) given the architecture. This could arguably be the most important metric, as it serves in part to remedy dark silicon and is also a well-known metric for how power-efficient an architecture/system is (e.g. Green500 [184]). Compared to GPUs, a majority of CGRAs can execute more operations – integer and floating-point alike – per unit power. Some architectures, such as the Plasticine can be up-to two orders of magnitude more power-efficient than GPUs, which offsets any pessimism that using TDP for GPUs may incur. Similar to the reasoning on compute densities, technology scaling impacts power-consumption as well, and the reality is likely that the power-efficiency is even better than what the graph illustrate.

Figure 7:f-g shows how the area- and power-densities of compute change with respect to frequency. As expected we see that the CGRAs operate largely in a different region than what GPUs– they have high power-efficiency while operating at a lower frequency.

Figure 7:h shows the performance of historical CGRAs scaled to that of a NVIDIA Volta- 100. The NVIDIA Volta-100 is one of the largest commercial accelerator readily available today, featuring a 815mm\(^2\) die built on 12 nm technology.
The peak performance of the V100 reaches 15.67 TFLOP/s and 13.43 TOP/s (32-bit). We scaled historical CGRAs that reported both area and performance to that of a NVIDIA V100 by scaling up the respective chip areas up to the V100’s 814 mm² and also increase the performance of the chip as a function of the increase transistor densities (obtained by moving to 12 nm). The transistor densities we used are the same as the one that NVIDIA GPUs experienced, and can be seen in Figure 8. Albeit this analysis do have drawbacks, such as power- and thermal-effects (subject to dark silicon [185]) being unaccounted for and frequency un-scaled (we assumed original CGRA frequencies), it does provide perspective over a what-if CGRAs were on the same playfield as GPUs. The results are seen in Figure 7h. We see that while most CGRA architecture would still be below the performance of a V100 (primarily due to the lower clock frequency). However, those architectures whose target indeed is to provide high-performance, such as Wavescalar, EGRA, SGMF, and SEED show several times higher performance than the V100 in either 32-bit integer, floating-point or both. The highest obtained level is demonstrated by Plasticine, which – if given an 815 mm² design built on 12 nm technology – could
reach 0.374 PFLOP/s of performance. From a 32-bit integer performance perspective, most architecture would outperform the NVIDIA Volta-100. The highest performance on the graph is actually a 4x4 CGRA provided as an example by CGRA-ME [60] that, albeit unusable by itself (only compute and no orchestration is available), would reach a compute capacity of nearly 1 Peta-OP/s. While this extrapolation is indeed limited, it aims to show that CGRAs have the architectural capability of competing with modern GPUs designs, assuming we can fully utilize these (potentially over-provisioned) computing resources.

![NVIDIA GPU Transistor Density Scaling](image)

Fig. 8: Transistor-density scaling as experienced by NVIDIA GPU chips, which we used to scale existing CGRAs from various times in history.

IX. DISCUSSION AND CONCLUSION

As we saw in the previous section, a vast majority of CGRAs are fairly small and run at a (comparably) low frequency. This, in turn, lead to very power-efficient designs, allowing CGRAs to be placed into embedded devices such as mobile phones or wearables and operate for hours. This power-efficiency, with respect to the performance they provide, allow CGRAs to compete (and possible perform better) than GPUs, which in turn could lead to a partial remedy for dark silicon. From the analysis that we did in this study, CGRAs should be considered a serious competitor to GPUs, particularly in a future post-Moore era when power-efficiency becomes more important.

However, in order to reap the better compute-densities and better power-efficiency that CGRA offers, larger architectures must be more thoroughly researched. Larger CGRA architectures, in particularly those aimed towards aiding or accelerating general-purpose computing, will be challenging to keep occupied. As we saw in the final graph, CGRA scaled to the level of a V100 will potentially have a peak performance consisting of hundreds of teraflops, but the main question is: will we be able to map and fully utilize all those computing resources for anything but the most trivial kernels?

Several authors have already pointed out that in order to harvest larger CGRAs, we need to complement current ways of extracting instruction parallelism (primary modulo scheduling) with other forms of concurrency. While modern CGRAs (e.g. Plasticine, SIMD-RA) do exploit SIMD-level parallelism, there will without doubt be needed to research even further, and support programming models such as CUDA (SGMF move towards this direction), multi-threading or even multi-tasking (e.g. OpenMP [186]) should be more aggressively pursued both from an architectural and programmability viewpoint in order to leverage future, large-sized, CGRAs. For example, the recently added dependencies features in frameworks such as OpenMP and OmpSs [187] matches very well to clustered CGRAs that have islands of both compute and scratchpad, where the dependencies would dictate how data would flow on these CGRAs (exploiting both inter- and intra-task level parallelism and data locality).

Another limitation of existing architectures is the application domain on which they accelerate. A large majority of CGRAs target embedded applications such as filters, stencils, decoders, etc. Studies that integrate the CGRA into the backend of a processor (e.g. TRIPS, DySER) tend to have a more diverse set of benchmarks available, and those studies (e.g. TARTAN, SEED, SGMF) that rely only on simulation (without hardware being developed) have the richest set of application support. Despite this, CGRAs suffer from a similar problem that current FPGAs struggle with: we limit ourselves to small, simple kernels, rather than studying the impact of these architecture on more complex applications. To give a concrete example, there is to this day no reconfigurable architecture that has seriously considered any of the proxy applications that drive HPC system procurement, such as for example the RIKEN Fiber [188] or ECP benchmark suites [189]. For FPGAs and High-Level Synthesis, this might make sense, since there is always the danger that these large kernels might not fit onto a single FPGA; CGRAs, however, can store multiple contexts and kernels with little overhead in switching between them, opening up possibilities for whole-application executions as well as opportunities to exploit inter-kernel temporal- and spatial data locality.

A different challenge with the present (and similar future) surveys is in the amount of reporting that the different studies do. For example, studies that applies a simulation methodology often have a wider benchmark coverage, but fails to report hardware details (e.g. area or RTL-information). At the same time, many CGRAs that were actually implemented in hardware (or RTL) do report area and power-consumption, but limits the benchmark selection and information. This, in turn, leads to gaps in several graphs were a clear high-performance candidate is represented in one graph, but due to
limited information, is absent from the other. This could more clearly be seen in this graphs that a derive metric, such as performance per power (OPs/W). Similarly, many papers often prefer to report relative performance improvement, rather than absolute numbers, leading to difficulty in reasoning around performance across a wide range of CGRAs. We include this in the discussion section more as a observation for future studies that may attempt to perform a similar survey.

Overall, this survey has shown that there is plenty of room for CGRA research to grow and to continue to be an active research subjects for use in future architecture, particularly striving to design high-performance CGRAs that aim at niche or general-purpose computation at scale. As transistor dimensions stop shrinking and Moore’s law no longer allows us the architectural freedom of carelessly spending silicon, it is here that reconfigurable architectures such as CGRAs might excel at providing performance in a post-Moore era.

ACKNOWLEDGEMENTS

This article is based on results obtained from a project commissioned by the New energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfets with very small physical dimensions,” IEEE Journal of Solid-State Circuits, vol. 9, no. 5, pp. 256–268, 1974.

[2] G. E. Moore et al., “Cramming more components onto integrated circuits,” 1965.

[3] T. N. Theis and H.-S. P. Wong, “The end of moore’s law: A new beginning for information technology,” Computing in Science & Engineering, vol. 19, no. 2, p. 41, 2017.

[4] J. S. Vetter, E. P. DeBenedictis, and T. M. Conte, “Architectures for the post-moore era,” IEEE Micro, vol. 37, no. 4, pp. 6–8, 2017.

[5] C. D. Schuman, J. D. Birdwell, M. Dean, J. Plank, and G. Rose, “Neuromorphic computing: A post-moore’s law complementary architecture,” Oak Ridge National Lab. (ORNL), Oak Ridge, TN (United States), Tech. Rep., 2016.

[6] R. Tessier and W. Burleson, “Reconfigurable computing for digital signal processing: A survey,” Journal of VLSI signal processing systems for signal, image and video technology, vol. 28, no. 1-2, pp. 7–27, 2001.

[7] J. Gray, “Grvi phalanx: A massively parallel risc-v fpga accelerator architecture,” in 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2016, pp. 17–20.

[8] G. Wang, B. Yin, K. Amiri, Y. Sun, M. Wu, and J. R. Cavalar, “Fpga prototyping of a high data rate lite uplink baseband receiver,” in 2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers. IEEE, 2009, pp. 248–252.

[9] I. Kuon, R. Tessier, J. Rose et al., “Fpga architecture: Survey and challenges,” Foundations and Trends® in Electronic Design Automation, vol. 2, no. 2, pp. 135–253, 2008.

[10] C. Yang, T. Geng, T. Wang, C. Lin, J. Sheng, V. Sachdeva, W. Sherman, and M. Herbordt, “Molecular dynamics range-limited force evaluation optimized for fpgas,” in 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), vol. 2160. IEEE, 2019, pp. 263–271.

[11] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka, “Evaluating high-level design strategies on fpgas for high-performance computing,” in 2017 27th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2017, pp. 1–4.

[12] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuoka, and S. Matsuoka, “Evaluating and optimizing openc mkl kernels for high performance computing with fpgas,” in SC 16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2016, pp. 409–420.

[13] H. R. Zohouri, A. Podobas, and S. Matsuoka, “Combined spatial and temporal blocking for high-performance stencil computation on fpgas using openc," in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018, pp. 153–162.

[14] A. Podobas and S. Matsuoka, “Designing and accelerating spiking neural networks using openc for fpgas,” in 2017 International Conference on Field Programmable Technology (ICFPT). IEEE, 2017, pp. 255–258.

[15] M. Miyanori and K. Oltukun, “Remarc: Reconfigurable multimedia array coprocessor,” IEICE Transactions on information and systems, vol. 82, no. 2, pp. 389–397, 1999.

[16] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor, “The raw compiler project,” in Proceedings of the Second SUIF Compiler Workshop, Stanford, CA, 1997.

[17] J. R. Hauser and J. Wawrzynek, “Gar: A mips processor with a reconfigurable coprocessor,” in Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186). IEEE, 1997, pp. 12–21.

[18] S. Ahmad, S. Subramaniam, V. Boppana, S. Lakka, F.-H. Ho, T. Knopp, J. Noguera, G. Singh, and R. Wittig, “Tilxilinx first 7nm device: Versal ai core (vc1902),” in 2019 IEEE Hot Chips 31 Symposium (HCS). IEEE, 2019, pp. 1–28.

[19] K. E. Fleming, K. D. Glossop, and S. C. Steely, “Apparatus, methods, and systems with a configurable spatial accelerator,” Oct. 15 2019, uS Patent 10,445,250.

[20] S. Brown, “Fpga architectural research: a survey,” IEEE Design & Test of Computers, vol. 13, no. 4, pp. 9–15, 1996.

[21] H. Amano, “A survey on dynamically reconfigurable processors,” IEICE transactions on Communications, vol. 89, no. 12, pp. 3179–3187, 2006.

[22] V. Teher and R. Krishirsag, “Survey on coarse grained reconfigurable architectures,” International Journal of Computer Applications, vol. 48, no. 16, pp. 1–7, 2012.

[23] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in Proceedings of the conference on Design, automation and test in Europe. IEEE Press, 2001, pp. 642–649.

[24] G. Thedoridis, S. Dousrids, and S. Vassiliadis, “A survey of coarse-grain reconfigurable architectures and cad tools,” in Fine-and Coarse-Grain Reconfigurable Computing. Springer, 2007, pp. 89–149.

[25] M. Wijtvliet, L. Waejjen, and H. Corporaal, “Coarse grained reconfigurable architectures in the past 25 years: Overview and classification,” in 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS). IEEE, 2016, pp. 235–244.

[26] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, “A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications,” ACM Computing Surveys (CSUR), vol. 52, no. 6, p. 118, 2019.

[27] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, “Mips: A microprocessor architecture,” ACM SIGMICRO Newsletter, vol. 13, no. 4, pp. 17–22, 1982.

[28] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, “The garp architecture and c compiler,” Computer, vol. 33, no. 4, pp. 62–69, 2000.

[29] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, “A reconfigurable arithmetic array for multimedia applications,” in FPGA, vol. 99. Citeseer, 1999, pp. 135–143.

[30] T. Stansfield, “Using multiplexers for control and data in d-fabrix,” in International Conference on Field Programmable Logic and Applications. Springer, 2003, pp. 416–425.

[31] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua et al., “Baring it all to software: The raw machine,” 1997.

[32] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee et al., “The raw microprocessor: A computational fabric for software circuits and general-purpose programs,” IEEE micro, vol. 22, no. 2, pp. 25–35, 2002.

[33] M. B. Taylor, N. Noguera, G. Singh, and R. Wittig, “Tile64-processor: A 64-core soc with mesh interconnect,” in 2008 IEEE International Solid-
N. D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for gpgpus,” in ACM SIGARCH computer architecture news, vol. 42, no. 3. IEEE Press, 2014, pp. 205–216.

D. Voitsechov, O. Port, and Y. Etsion, “Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays,” in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018, pp. 42–54.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montery, “Nvidia tesla: A unified graphics and computing architecture,” IEEE micro, vol. 28, no. 2, pp. 39–55, 2008.

D. Kirk et al., “Nvidia cuda software and gpu parallel computing architecture,” in ISMM, vol. 7, 2007, pp. 103–104.

A. Munshi, “The opencl specification,” in 2009 IEEE Hot Chips 21 Symposium (HCS). IEEE, 2009, pp. 1–314.

C. M. Wittenbrink, E. Kilgariff, and A. Prabhah, “Fermi g100 gpu architecture,” IEEE Micro, vol. 31, no. 2, pp. 50–59, 2011.

M. Zhu, L. Liu, S. Yin, Y. Wang, W. Wang, and S. Wei, “A reconfigurable multi-processor soc for media applications,” in Proceedings of 2010 IEEE International Symposium on Circuits and Systems. IEEE, 2010, pp. 2011–2014.

M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi et al., “Stream applications on the dynamically reconfigurable processor,” in Proceedings of 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No. 04EX921). IEEE, 2004, pp. 137–144.

T. Sato, “Dapdna-a a dynamically reconfigurable processor with 376 32-bit processing elements,” in 2005 IEEE Hot Chips XVII Symposium (HCS). IEEE, 2005, pp. 1–24.

D. N. Truong, W. H. Cheng, T. Moshen, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeusen, C. Watanik, A. T. Tran, Z. Xiao et al., “A 167-processor computational platform in 65 nm cmos,” IEEE Journal of Solid-State Circuits, vol. 44, no. 4, pp. 1130–1144, 2009.

D. M. Shapiro, “Globally-asynchronous locally-synchronous systems.” Stanford Univ CA Dept of Computer Science, Tech. Rep., 1984.

S. Kyo and S. Okazaki, “Imapcar: A 100 gops in-vehicle vision processor based on 128 ring connected four-way vliw processing elements,” Journal of Signal Processing Systems, vol. 62, no. 1, pp. 5–16, 2011.

S. Kyo, T. Koga, and S. Okazaki, “Imap-ce: A 51.2 gops video rate image processor with 128 vliw processing elements,” in Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205), vol. 3. IEEE, 2001, pp. 294–297.

S. Das, N. Sivanandan, K. T. Madhu, S. K. Nandy, and R. Narayan, “Rhyme: Redefining hyper cell multicore for accelerating hpc kernels,” in 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID). IEEE, 2016, pp. 601–602.

K. T. Madhu, S. Das, S. Nalesh, S. Nandy, and R. Narayan, “Compiling hpc kernels for the redefine cgra,” in 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems. IEEE, 2015, pp. 405–410.

R. Prabhakar, Y. Zhang, D. Koepflinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Okhoton, “Plasticine: A reconfigurable architecture for parallel patterns,” in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017, pp. 389–402.

D. Koepflinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis et al., “Spatial: A language and compiler for application accelerators,” in ACM Sigplan Notices, vol. 53, no. 4. ACM, 2018, pp. 296–311.

C. systems, “Wafer-scale deep learning,” Hot-Chips 31, vol. https://www.hotchips.org/hc31/HC31_11_Cerbrus.SeanLe.v02.pdf, 2019.

N. Clark, A. Hormati, and S. Mahlke, “Veal: Virtualized execution accelerator for loops,” in ACM SIGARCH Computer Architecture News, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 389–400.

J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, “A fully pipelined and dynamically composible architecture of cgra,” in 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2014, pp. 9–16.
analytics,” in 2018 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2018, pp. 1–5.

[151] S. Das, D. Rosset, K. J. Martin, P. Coussy, and L. Benini, “A 142mops/mw integrated programmable array accelerator for smart visual processing,” in 2017 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2017, pp. 1–4.

[152] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique, “X-cgra: An energy-efficient approximate coarse-grained reconfigurable architecture,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019.

[153] M. Langhammer and B. Pasca, “Floating-point dsp block architecture for fpgas,” in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2015, pp. 117–125.

[154] G. Stitt and J. Coole, “Intermediate fabrics: Virtual architectures for near-instant fpga compilation,” IEEE Embedded Systems Letters, vol. 3, no. 3, pp. 81–84, 2011.

[155] S. Shukla, N. W. Bergmann, and J. Becker, “Quku: a two-level reconfigurable architecture,” in IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI’06). IEEE, 2006, pp. 6–pp.

[156] N. W. Bergmann, S. K. Shukla, and J. Becker, “Quku: a dual-layer reconfigurable architecture,” ACM Transactions on Embedded Computing Systems (TECS), vol. 12, no. 1s, p. 63, 2013.

[157] S. Shukla, N. W. Bergmann, and J. Becker, “Quku: A fpga based flexible coarse grain architecture design paradigm using process networks,” in 2007 IEEE International Parallel and Distributed Processing Symposium. IEEE, 2007, pp. 1–7.

[158] Xilinx. (2016) Microblaze processor reference guide (ug984). [Online]. Available: "https://www.xilinx.com/.../ug984-vivado-microblaze-pdf.pdf"

[159] A. Brant and G. G. Lemieux, “Zuma: An open fpga overlay architecture,” in 2012 IEEE 20th international symposium on field-programmable custom computing machines. IEEE, 2012, pp. 93–96.

[160] R. L. Lysecky, K. Miller, F. Vahid, and K. A. Vissers, “Firm-core virtual fpga for just-in-time fpga compilation,” in FPGA, 2005, p. 271.

[161] T. Myint, M. Amagasaki, Q. Zhao, and M. lida., “A slim-based overlay architecture for fine-grained virtual fpga,” IEEE Electronics Express, pp. 16–20 190610, 2019.

[162] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, “An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture,” in Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems. ACM, 2011, pp. 195–204.

[163] A. Severance and G. G. Lemieux, “Embedded supercomputing in fpgas with the vectorblox mxp matrix processor,” in Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. IEEE Press, 2013, p. 6.

[164] L. B. D. Silva, R. Ferreira, M. Canesche, M. M. Menezes, D. M. Vieira, J. Penha, P. Jamieson, and J. A. M. Nacif, “Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications,” ACM Transactions on Embedded Computing Systems (TECS), vol. 18, no. 5s, p. 56, 2019.

[165] D. Capalija and T. S. Abdelrahman, “Towards synthesis-free jit compilation to commodity fpgas,” in 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2011, pp. 202–205.

[166] J. Ball, “The nios ii family of configurable soft-core processors,” in 2005 IEEE Hot Chips XVII Symposium (HCS). IEEE, 2005, pp. 1–40.

[167] C. Feng and L. Yang, “Design and evaluation of a novel reconfigurable alu based on fpga,” in Proceedings 2013 International Conference on Mechatronics Sciences, Electric Engineering and Computer (MEC). IEEE, 2013, pp. 2286–2290.

[168] K. Paul, C. Dash, and M. S. Moghaddam, “remorph: a runtime reconfigurable architecture,” in 2012 15th Euromicro Conference on Digital System Design. IEEE, 2012, pp. 26–33.

[169] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy, “Deco: A dsp block based fpga accelerator overlay with low overhead interconnect,” in 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2016, pp. 1–8.

[170] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, “Adapting the dyser architecture with dsp blocks as an overlay for the xilinx zynq,” ACM SIGARCH Computer Architecture News, vol. 43, no. 4, pp. 28–33, 2016.

[171] C. Liu, H.-C. Ng, and H. K.-H. So, “Automatic nested loop acceleration on fpgas using soft cgra overlay,” arXiv preprint arXiv:1509.00042, 2015.

[172] —, “Quickdough: a rapid fpga loop accelerator design framework using soft cgra overlay,” in 2015 International Conference on Field Programmable Technology (FPT). IEEE, 2015, pp. 56–63.

[173] D. Capalija and T. S. Abdelrahman, “A high-performance overlay architecture for pipelined execution of data flow graphs,” in 2013 23rd International Conference on Field programmable Logic and Applications. IEEE, 2013, pp. 1–8.

[174] D. Mansur, “Stratis iv fpga and hardcopy iv ascii@ 40 nm,” in 2008 IEEE Hot Chips 20 Symposium (HCS). IEEE, 2008, pp. 1–22.

[175] K. Ovtcharov, I. Tili, and J. G. Steffan, “Tilt: a multithreaded vlv soft processor family,” in 2013 23rd International Conference on Field programmable Logic and Applications. IEEE, 2013, pp. 1–4.

[176] R. Rashid, J. G. Steffan, and V. Betz, “Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls,” in 2014 International Conference on Field-Programmable Technology (FPT). IEEE, 2014, pp. 20–27.

[177] D. Lewis, D. Cashman, M. Chan, J. Chromczak, G. Lai, A. Lee, T. Vanderhoek, and H. Yu, “Architectural enhancements in stratis iv™,” in Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2013, pp. 147–156.

[178] Z. T. Akhla, “A hybrid partially reconfigurable overlay supporting just-in-time assembly of custom accelerators on fpgas,” 2017.

[179] D. Koch, Partial Reconfiguration on FPGAs: Architectures, Tools and Applications. Springer Science & Business Media, 2012, vol. 153.

[180] M. X. Yue, D. Koch, and G. G. Lemieux, “Rapid overlay builder for xilinx fpgas,” in 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2015, pp. 17–20.

[181] R. Kisiel and Z. Szczepański, “Trends in assembling of advanced ic packages,” Journal of Telecommunications and information Technology, pp. 63–69, 2005.

[182] Techpowerup. (2020 (accessed: January)) Techpowerup. [Online]. Available: "https://www.techpowerup.com"

[183] NVIDIA. Nvidia cuda programming guides(s). [Online]. Available: developer.download.nvidia.com/.../NVIDIA_CUDA_ProgrammingGuide.pdf

[184] W.-c. Feng and K. Cameron, “The green500 list: Encouraging sustainable supercomputing.” Computer, vol. 40, no. 12, pp. 50–55, 2007.

[185] D. Burger, “Dark silicon and the end of multicore scaling,” in 2013 38th Annual international symposium on computer architecture (ISCA). IEEE, 2011, pp. 365–376.

[186] R. Chandra, D. Dagum, D. Kehr, R. Menon, D. Maydan, and J. McDonald, Parallel programming in OpenMP. Morgan kauffmann, 2001.

[187] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Mártil, and J. Planas, “Ompss: a proposal for programming heterogeneous multi-core architectures,” Parallel processing letters, vol. 21, no. 02, pp. 173–193, 2011.

[188] AICS. (2015) Fiber minapp suite. [Online]. Available: https://fiber-minapp.github.io/

[189] ExaScale-Project. (2018) Ecp proxy apps suite. [Online]. Available: https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/