Towards Automated Generation of Chiplet-Based Systems

Invited Paper

Ankur Limaye†, Claudio Barone†, Nicolas Bohm Agostini†, Marco Minutoli†,
Joseph Manzano†, Vito Giovanni Castellana†, Giovanni Gozzi*, Michele Fiorito*, Serena Curzel*,
Fabrizio Ferrandi*, Antonino Tumeo†
*Politecnico di Milano, †Pacific Northwest National Laboratory
†antonino.tumeo@pnnl.gov

2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) | 979-8-3503-9354-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASP-DAC58780.2024.10473980

Abstract— The Software Defined Architectures (SODA) Synthesizer is an open-source compiler-based tool able to automatically generate domain-specialized systems targeting Application-Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) starting from high-level programming. SODA is composed of a high-level frontend, SODA-OPT, which leverages the multilevel intermediate representation (MLIR) framework to interface with productive programming tools (e.g., machine learning frameworks), identify kernels suitable for acceleration, and perform high-level optimizations, and of a state-of-the-art high-level synthesis backend, Bambu from the PandA framework, to generate custom accelerators. One specific application of the SODA Synthesizer is the generation of accelerators to enable ultra-low latency inference and control on autonomous systems for scientific discovery (e.g., electron microscopes, sensors in particle accelerators, etc.). This talk will discuss ongoing work on the SODA synthesizer to enable no-human-in-the-loop generation and design space exploration of the chiplets for highly specialized artificial intelligence accelerators. Connecting these highly specialized chiplets to general-purpose cores or programmable accelerators will allow to quickly deploy autonomous systems for scientific discovery.

I. Introduction

An increasing number of domains require highly specialized systems to perform data analysis and decision-making at ultra-low latencies. Cyberphysical systems need to interact with the environment and, based on acquired sensor inputs, must quickly react to the events by establishing the next set of operations to perform. In autonomous science, for example, a system operating near or in an experimental instrument needs to quickly filter massive amounts of the acquired unstructured data and identify how to set the instrument parameters to observe a new physical phenomenon or set up the next set of experiments [1]. All these processes need to happen within the latency constraint of the physical system under experimentation. There has been an intense interest in leveraging artificial intelligence methods in areas like precision material synthesis and additive manufacturing to observe and control complex processes (e.g., layer deposition). Even the simplest models (typically shallow autoencoders based on convolutional neural networks) must process large amounts of multimodal data in a few nanoseconds, and the processing must happen on the edge because there is not enough time to transmit all the data to a central computing system and wait for an answer [2]. This obviously increases the need for power-efficient systems finely tuned for the specific application constraints (e.g., power availability, radiation hardening, security among components, etc.). While fitting within all these metrics can only be achieved by designing highly specialized accelerators, creating them and inserting them in a complex computational system typically requires deep hardware design expertise and long development times. Without the correct tools and approaches that could bridge this design gap, it would not be possible to meet the ever-growing requirements of these autonomous systems. We argue that tools that can translate artificial intelligence and data analysis methods developed in high-level programming frameworks in specialized hardware accelerators are critical to enable the realization of these domain-specific systems for autonomous systems. Even more, these tools need to consider the entire system-level aspects, from how accelerators are connected to the rest of the processing architecture, the memory, and their coordination. One clear opportunity is to leverage these types of tools within a chiplet ecosystem, interfacing the generated accelerators together and with other commodity components, further enabling to fine-tuning the system. This requires tools that are able to reason with interfaces, can generate a hardware representation of the accelerators along with considerations from the backend tools and have the knowledge of the entire application and system requirements. To address some of these challenges, we have introduced the SODA (SOftware-Defined Archictectures) synthesizer framework [3], a framework composed of open-source, interoperating hardware compilers. SODA is composed of a frontend, SODA-OPT [4][1], based on the MLIR framework, and a backend, based on the high-level synthesis (HLS) tool Bambu[2]. SODA-OPT interfaces with high-level programming tools (e.g., Python) and captures high-level application and system requirements simultaneously. It performs hardware/software partitioning of the initial specification and orchestrates accelerators' execution. The part of the specification deemed to execute in hardware is then pre-optimized and provided to Bambu [5] to generate the accelerators code in a hardware description language (e.g., Verilog). Accelerators are then combined together and in a system with more general processing elements. Bambu can interface with both Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs) logic synthesis tools, specializing the generated Verilog accordingly. In particular, it can interface with the OpenROAD flow, providing an end-to-end open-source solution. In this paper, we overview SODA and discuss

[1] SODA-OPT is available at: https://github.com/pnnl/soda-opt/

[2] Bambu is available at: https://github.com/ferrandi/PandA-bambu

the ongoing work to support system-level interfacing aspects. We focus, in particular, on how to support an ecosystem where accelerator chiplets might be combined with more general-purpose chiplets. We discuss approaches we are exploring to enable efficient memory access for the accelerators (which is critical for the performance, especially with larger and larger machine learning models) and system integration, providing an overview of opportunities and elements to further explore to enable a "push-button" generation of a specialized chiplet for a given model.

II. SODA Overview

The SODA Synthesizer (Figure 1a) is composed of SODA-OPT [4], the frontend compiler for system-level partitioning and high-level optimizations, and PandA-Bambu [5], the HLS backend. The toolchain takes algorithm descriptions written in high-level programming languages and data science frameworks as inputs. The outputs are the register-transfer level (RTL) descriptions of the accelerators, and the run-time calls for the general-purpose processor.

The high-level specification provided to the compilation pipeline is translated into a high-level intermediate representation (IR) in the early stages of the high-level optimizer. This IR actually consists of several *dialects* (i.e., specialized IRs derived from the same meta-IR) of the Multi-Level Intermediate Representation (MLIR) [6].

SODA-OPT performs hardware/software partitioning of the input program and architecture-independent optimizations by leveraging MLIR features. SODA-OPT generates two different types of LLVM IR outputs. The first one is an optimized LLVM IR file without any external dependencies representing the kernels identified for acceleration. This file is, in turn, is passed as an input to PandA-Bambu. The other output is an LLVM IR file representing the host program that orchestrates the calls to the accelerators. As previously highlighted, Bambu's outputs can be tuned for different synthesis targets. Furthermore, they can have different types of interfaces and be composed in different ways to achieve complex multi-accelerator systems. Both SODA-OPT and Bambu implement optimizations as different compiler transformations. As such, the design space exploration practically consists of exploring compiler optimizations passes and options.

The framework integrates a design space exploration engine that selects a suitable combination of compiler passes and parameters both at the frontend and at the backend level, optimizing the design for a chosen target metric (performance, area, power, etc.).

A. SODA Frontend

SODA-OPT [4] is the framework's compiler frontend (Figure 1b). SODA-OPT leverages LLVM's MLIR framework, a framework that allows for building modular and reusable compiler infrastructure by defining *dialects*, i.e., specialized and self-contained IRs compliant with MLIR's meta-IR syntax.

Conventional HLS tools, to provide high-quality results, typically require the user to rewrite the code of the algorithms (usually in an imperative language such as C/C++) and annotate it with pragma annotations that specify additional hardware information and trigger HLS-specific optimizations (e.g., loop unrolling, enabling loop pipelining, adding interfaces). The approach adopted in SODA-OPT, instead, leverages the semantic information carried by context-specific MLIR dialects to automatically apply high-level transformations while preparing the input program for hardware synthesis.

SODA-OPT provides compilation passes to *S*earch, *O*utline, *Optimize*, *D*ispatch, and *A*ccelerate parts of the initial specification coming from high-level frameworks. SODA-OPT defines the soda MLIR dialect, used for the automatic partitioning of the input application into a host program responsible for orchestrating the runtime execution, and the custom hardware accelerators [4].

In the search phase, SODA-OPT analyzes the MLIR representation of the specification to identify code regions suitable for acceleration. Such regions are then extracted into separate MLIR modules (outline), which undergo further optimization passes. SODA-OPT can use MLIR dialects and the associated optimizations directly included within the MLIR distribution in the LLVM compiler framework or provided externally. For example, SODA-OPT uses MLIR's linalg and affine dialects to identify operators and perform loop optimizations.

A growing number of machine learning and data science frameworks (e.g., TensorFlow, ONNX-MLIR, and TORCH-MLIR), scientific computing frameworks (e.g., NPCOMP), and even general-purpose programming languages (e.g., FLANG for Fortran) leverage MLIR and implement specific MLIR dialects as part of their compilation flow for program optimization. While the SODA-OPT flow is optimized and tested for TensorFlow, TF-Lite, and PyTorch inputs, it can potentially interface with all the frameworks that lower their language specific dialects to more general dialects provided in the MLIR distribution.

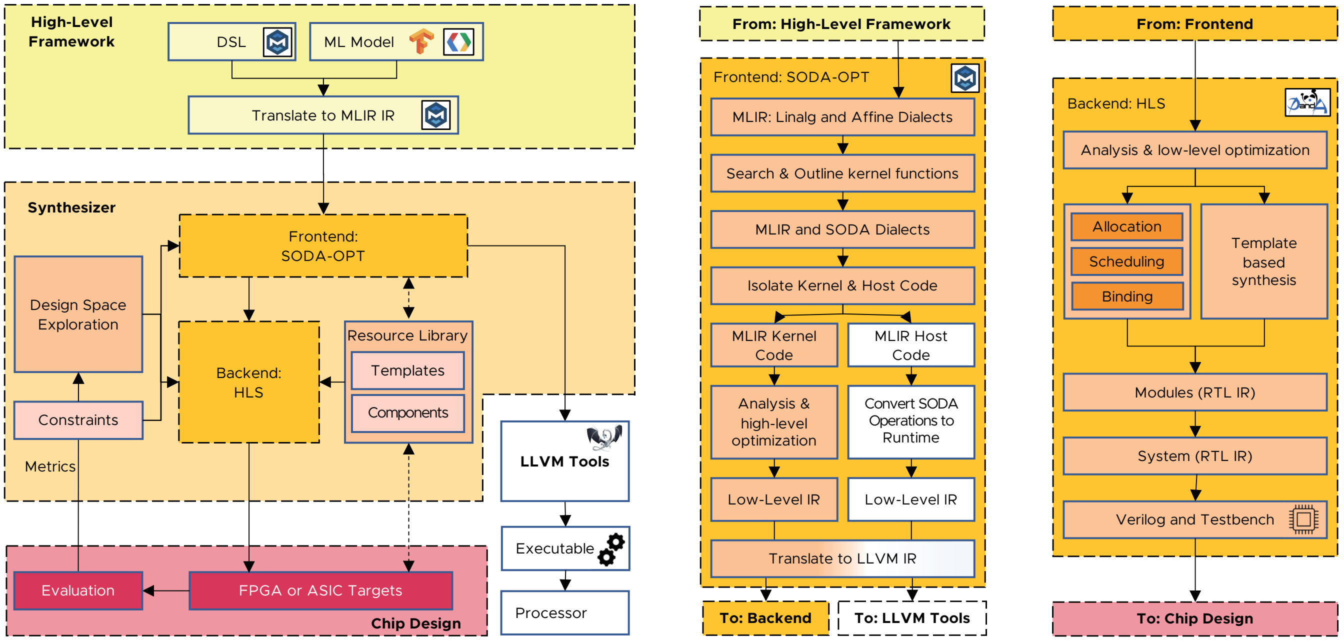
B. SODA Synthesizer Backend

Bambu (Figure 1c), the state-of-the-art HLS tool of the PandA framework, is the backend of the SODA framework.

Bambu, as a standalone tool, leverages conventional compiler frameworks (GCC and Clang) as frontends, supporting several languages with a specific focus on C and C++. It can also take LLVM IR as input through a dedicated Clang plugin. In the SODA framework, Bambu is fed with the optimized LLVM IR produced by SODA-OPT, providing higher quality results with respect to accelerators generated through specifications in C/C++ and processed with the other frontends.

Bambu performs the classical high-level synthesis steps (resource allocation, scheduling, and binding) by leveraging several specialized intermediate representations. It also performs several common (operation chaining, loop optimizations) and less common (range and bitwidth analysis) optimizations. Although we typically leverage the Verilog backend, Bambu can also generate RTL descriptions of the accelerators in VHDL. Together with the synthesizable RTL code, Bambu can also automatically generate testbenches and scripts for simulation tools (including Xilinx Isim, Mentor Modelsim, and verilator), enabling verification.

Through Bambu, the SODA framework can target FPGAs of a variety of vendors, including AMD/Xilinx, Intel/Altera, Lattice and Nanoexplore, and ASICs. For ASICs, SODA supports



(a) (b) (c)

Fig. 1: The SODA framework is an open-source, multi-level, modular, extensible, hardware generator composed of a high-level compiler and an HLS backend

Verilog-to-GDSII generation using both commercial (Synopsis Design Compiler) and open-source (OpenROAD [7]) logic synthesis tools. Bambu generates the scripts and invokes the logic synthesis tools with the right parameters.

To support the different target devices, Bambu integrates a resource characterization tool, aptly named *Eucalyptus*. Eucalyptus performs the logic synthesis for all components in Bambu's resource library, annotating relevant delay, area, and power information, and runs additional microbenchmarks to model the interconnect, allowing to drive the synthesis algorithms to a higher quality of results as the hardware designs as they get built. For example, the module binding and operations scheduling algorithms can use this information to optimize the design along various metrics (e.g., overall latency and area of the accelerator) while meeting synthesis constraints like a target frequency. For example, given a specific target technology and target frequency, two functional units may have a delay much lower than the clock period, so that they can be *chained* to execute in the same clock cycle (control step).

Bambu generates accelerators following the finite state machine with datapath model (FSMD). However, we have extended the tool to integrate methodologies for complex parallel accelerator designs. This includes composing the FSMD accelerators in coarse-grained dataflow designs [8] or dynamically scheduled arrays of multithreaded accelerators [9]. Bambu also integrates modular synthesis methodologies [10] to enable inter-procedural resource sharing across the design hierarchy.

Since MLIR descriptions intrinsically are parallel and hierarchical, controlling Bambu through the SODA-OPT frontend enables SODA to better deal with the design hierarchy and move some hardware-related optimizations at higher levels of abstraction, reducing the need for providing hardware-related information in the form of annotations.

The support of specialized interfaces to memory and to other accelerators is also of critical importance. Bambu enables connecting accelerators to a variety of memories (either on or off-chip) with the number of ports dependent either on parameters or inferred from the number of arguments of the synthesized functions. The ports might use either a simple interface protocol or the AMBA AXI standard [11]. The way accelerators can interface to the system, to other accelerators, to memory, and in general externally is critical when considering the generation of an accelerator chiplet.

III. Towards Chiplets Support in SODA

While there still are some gaps to bridge, all the necessary infrastructure to directly generate and enable chipset-based domain-specific systems is readily available in the SODA framework. Furthermore, this demonstrates the advantages of a modular and extensible compiler-based hardware design toolchain. The ongoing work on interfaces for other processing elements and accelerators and memory is specifically relevant, considering the flow capability to obtain the actual physical layouts of the accelerator through the logic synthesis tools.

In fact, leveraging the Bambu extendable resource library, it is possible to add components like Intel's Advanced Interface Bus (AIB) [12]. AIB is a specification at the physical layer of an interconnect for chiplets, which provides higher bandwidth than conventional serializer/deserializer (serdes) links . On top of it, it is possible to perform transactions following, for example, the AXI protocol. In the long term, it is possible to imagine the use of the Compute Express Link (CXL) [13] protocol layered on top of the Universal Chiplets Interconnect Express (UCIe) [14] physical communication layer. UCIe borrows from AIB to define standard physical layer specifications for chiplets interconnect. CXL intellectual property

(IP) components are already available to be instantiated on PCI Express accelerator boards with FPGA devices. This means that tools able to target the generation of accelerators that employ CXL as a communication protocol to connect with other accelerators and processing elements, and AXI to connect to memory controllers, and thus, could be easily translated from FPGA accelerator designs to chiplets.

In another instance, supporting interfaces towards integrating with other systems on chips enables generating a fully contained chiplet with related harness platforms. By supporting the AXI interfaces, for example, we are able to connect our accelerators to research platforms for heterogeneous systems-on-chips like the Embedded Scalable Platform (ESP) [15]. The ESP tiled architecture is tape-out proven, and can both provide a tiled chiplet architecture as well as act as a chiplet containing a specialized SoC (general-purpose processor with an accelerator).

As highlighted, AXI is also critical in supporting access to external memories (e.g., DRAM). Several memory controller IPs (e.g., Xilinx/AMD for FPGAs) are based on soft or hard hardware macros that expose AXI ports to the external processing elements. Hence, accessing the memory through the memory controller requires support for the AXI protocol, including some of its features like split transactions and bursting to keep the memory channels and the memory controller utilized. For a system-in-package (SIP) design, it is easy to imagine accelerator chiplets requesting data to memory controller and, in general I/O chiplets through AXI.

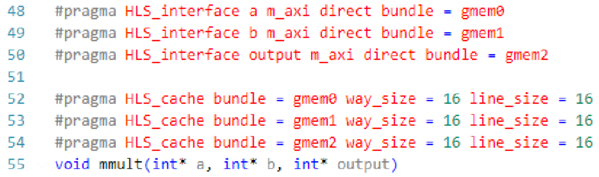
In the following section, we present some ongoing work to support the AXI protocol in our toolchain, including mechanisms to implement accelerator caches that also enable prefetchcing and cache-line size bursting.

IV. Designing AXI interfaces with caches

Conventional commercial HLS tools (e.g., Xilinx Vitis, Mentor Catapult C) only provide limited solutions to improve access patterns to DRAM memory. These typically require restructuring and annotating C code (specifically loops) to coalesce multiple memory accesses and trigger burst transfers. Additionally, it might not be effective if the accelerator is not directly interfacing with a memory controller. Instead, our approach is based on automatically generating AXI ports of variable size that can integrate caches. Similar to how caches are used in general-purpose processors, our approach allows for reducing the latency with which an HLS-generated accelerator accesses external memory through spatial and temporal locality. However, HLS-generated accelerators are highly specific to the original computational kernel, so it is necessary to provide the flexibility to customize caches as each accelerator is designed.

V. Caches Design

The architecture of our caches is inspired by the IObundle (IOb) caches [16]. However, we provide significant improvements and integration within the SODA flow. Our caches can be customized in size, ways, and behaviors (e.g., write-through/write-back) as needed by the application through specific parameters. The cache module presents an AXI4 master [11] interface towards the external memory that can be used to read or write data in bursts. Burst transactions optimize memory access by reducing traffic on the memory interface and access latency by transferring an entire cache line in a single transaction instead of an element at a time. Bursting also enables more efficient prefetching for applications having dense data structures and high locality, such as dense linear algebra operations. In HLS-generated accelerators, a function transformed in an accelerator can have a single channel to memory or multiple separate channels to different memories (for example, one per argument of the original function, if they are pointers that do not alias and access separate data structures in memory). Our approach can generate a separate cache for each of these channels. This allows to customize each cache for the access patterns of each channel, and removes cache line contention for different memory areas accessed in parallel.



```
#pragma HLS_interface a m_axi direct bundle = gmem0
#pragma HLS_interface b m_axi direct bundle = gmem1
#pragma HLS_interface output m_axi direct bundle = gmem2

#pragma HLS_cache bundle = gmem0 way_size = 16 line_size = 16
#pragma HLS_cache bundle = gmem1 way_size = 16 line_size = 16
#pragma HLS_cache bundle = gmem2 way_size = 16 line_size = 16
void mmult(int* a, int* b, int* output)
```

Fig. 2: Cache declarations for the C++ frontend

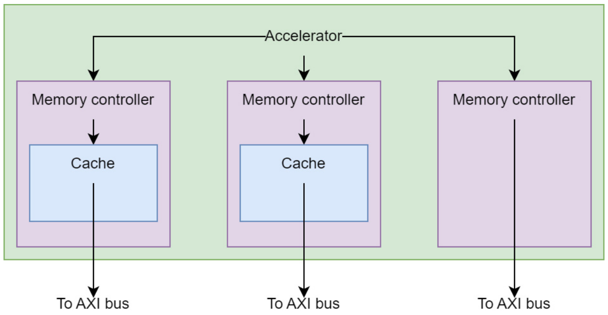


Fig. 3: Conceptual overview of the generated design interfaces

Adding caches to an accelerator with our framework can be achieved in two ways. If the source code is in C or C++, we can annotate the main kernel function with a limited set of directives that specifies instantiation and cache parameters for each of the desired AXI memory channels that are parsed through Clang with a specific plugin and saved in an XML that is then provided to Bambu. Figure 2 shows the set of pragmas used before the function (kernel) to transform in hardware. The code of the kernel does not require any modification. If, instead, the source specification comes from high-level programming frameworks, SODA-OPT takes charge of generating the XML file, extrapolating the required information through high-level analysis. The SODA design space exploration engine could also explore the set of parameters to target several quality of results metrics. Figure 3 shows a conceptual overview of the generated design interfaces, with or without the caches.

Our design provides several improvements with respect to the IOb caches. The most notable is a more efficient implementation of the write transactions. Our solution can support outstanding write requests, i.e., it can initiate a new transaction

TABLE I: Resource utilization overhead (for registers, LUTs - R,L) and speed up (as execution delay in clock cycles - C) with 50 clock cycles of memory latency - 2mm and doitgen

	No cache			16, 256			32, 256			64, 256			128, 256			256, 256			256, 16			256, 32			256, 64			256, 128		
	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C
2mm	6671	6629	136161	1.36	1.19	9.1	1.44	1.27	9.75	1.45	1.27	9.75	1.44	1.28	9.75	1.42	1.27	9.75	1.36	1.24	1.01	1.42	1.28	1.31	1.42	1.27	1.61	1.43	1.27	9.75
doitgen	4520	4901	739564	1.36	1.18	9.62	1.49	1.23	10.05	1.45	1.23	10.05	1.46	1.23	10.05	1.44	1.23	10.05	1.32	1.17	1.07	1.46	1.23	1.36	1.45	1.23	1.56	1.45	1.23	10.05

TABLE II: Resource utilization overhead (for registers, LUTs - R,L) and speed up (as execution delay in clock cycles - C) with 50 clock cycles of memory latency - atax, bicg, mvt

	No cache			16			32			64			128			256		
	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C
atax	7171	6393	7606	1.05	1.03	4.27	1.11	1.07	4.79	1.11	1.07	5.08	1.11	1.07	5.33	1.11	1.07	5.33
bicg	8094	6855	7332	1.06	1.04	2.97	1.11	1.07	3.32	1.11	1.07	3.94	1.24	1.07	4.09	1.11	1.07	6.37
mvt	4787	5513	14680	1.07	1.06	1.48	1.16	1.10	1.90	1.17	1.10	2.27	1.16	1.10	4.10	1.16	1.10	7.36

before receiving the response to the previous one. This further reduces channel latency, making our solution more efficient in case of frequent write transactions. Furthermore, our caches include a flushing mechanism, so that they can write back dirty cache lines to the external memory before the accelerator signals the completion of the execution, assuring that they moved all data to the external memory. Our caches do not require coherency mechanisms, as either the user or the high-level analysis assures that separate ports operate on data in different memory regions (i.e., there is no pointer aliasing and data sharing).

VI. Experimental results

To evaluate the impact of our caches for HLS-generated accelerators, we synthesized five kernels from the PolyBench [17] suite: *2mm*, *atax*, *bicg*, *doitgen*, *mvt*. We simulated the generated accelerators and compare their execution delay with and without caches, sweeping cache sizes from 16 to 256 words of 4 bytes each, and varying external memory latency from 5 to 50 clock cycles. We also compare the resource utilization by synthesizing the kernels for a Virtex7 FPGA device using Vivado 2020.2. We use an input of 10 elements for every vector and of 10 by 10 elements for every matrix. We write the HLS code to instantiate a different AXI memory port and cache for each input matrix, unless there is no cache contention due to how a kernel operates. For instance, in the *2mm* kernel, matrix C is only accessed after the computation on matrix B is over, and the two also have the same access pattern. Thus, the two inputs can share the same memory port and cache without impacting performance, reducing resource utilization. We do not generate separate channels and caches for inputs/outputs corresponding to vectors, reusing the ones for accessing data of matrices. Since we use vectors much smaller than matrices, we expect only limited cache interference. In Tables I and II, we show the area overhead (for look-up tables, L, and registers, R) and the execution delay in clock cycles (C) for each accelerator when accessing an external memory with a latency of 50 clock cycles. We express the cache sizes in the number of memory words. We only show absolute values for the baseline (no caches), showing overhead factors (L and R columns) and speedup (C column) for cache configurations instead. *2mm* and *doitgen* perform matrix multiplications, thus accessing first input matrix by row and second by column. For these kernels, we explore the size of the caches for the input matrices separately, reporting first the size of the cache pertaining to data accessed by row and then the size of the cache for data accessed by column. We show the analysis with the entire latency swipe only for *2mm* (Figure 4) and *atax* (Figure 5). The trends of the other benchmarks are similar. As expected, using caches provides higher speedup as the external memory latency increases. The best case configuration for *doitgen* reaches a speedup over 10× the baseline solution with no caches. With low external memory latency, caches become ineffective, to the point that handling misses and line replacements in some cases (e.g., small caches for matrices accessed by column in Figure 4) leads to a slowdown.

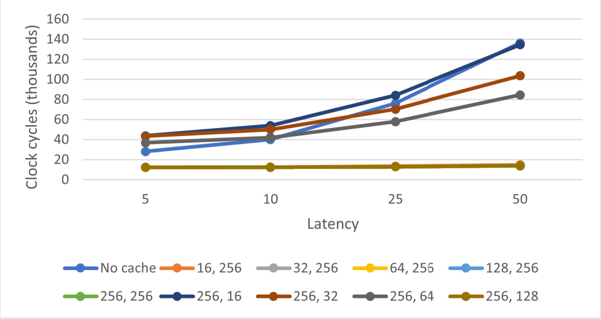


Fig. 4: Execution delay in clock cycles - 2mm

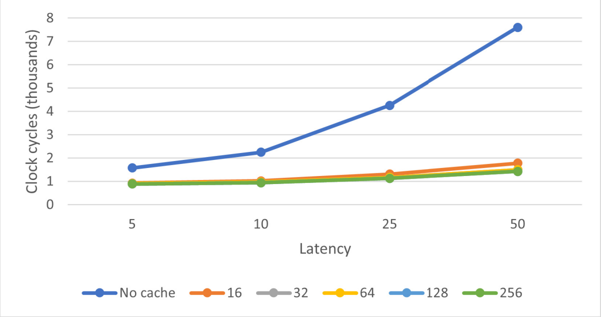


Fig. 5: Execution delay in clock cycles - Atax

VII. Conclusions

More and more application areas (from drones to scientific experimental systems) employ autonomous systems that must quickly make decisions depending on the acquired data. While the common aspect is applying data analytics and artificial intelligence methods, each and every domain requires a highly specialized system that is able to satisfy its specific constraints (performance, power, area, cooling, security, real-time awareness, and more). Chiplets provide an excellent opportunity to design such domain-specific systems by combining off-the-shelf components (such as general-purpose processing units and programmable accelerators) with highly specialized accelerators. This approach, however, requires adequate design automation tools that can enable the agile generation and design space exploration of the specialized accelerators. Such tools need to consider the generation of the accelerators in the whole chiplet-based system context, including interfaces between general-purpose processing elements, accelerators, and memories.

After providing a brief overview of the Software Defined Architectures (SODA) Synthesizer, our open-source end-to-end design automation toolchain from high-level programming frameworks to silicon, we discussed ongoing work to support the generation of chiplets, focusing in particular on interfacing. We then presented some ongoing work to enable efficient memory access through the automatic instantiation of caches for the AXI protocol.

We believe that our open-source tool could provide a solid basis to the community for the agile development of chiplet-based highly specialized systems.

Acknowledgments

This research was supported by the Compiler Frameworks and Hardware Generators to Support Innovative US Government (IUSG) Designs project at Pacific Northwest National Laboratory (PNNL), the Data Model Convergence (DMC) Laboratory Directed Research and Development (LDRD) Initiative, and the Adaptive Tunability for Synthesis and Control via Autonomous Learning on Edge (AT SCALE) LDRD Initiative.

References

[1] E. Bethel and eds. Report of the doe workshop on management, analysis, and visualization of experimental and observational data – the convergence of data and computing. Technical report, 2016.

[2] W. Snyder. Rapideels, 2022. Online, accessed 11-2022.

[3] Nicolas Bohm Agostini, Serena Curzel, Jeff Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. Bridging python to silicon: The soda toolchain. *IEEE Micro*, 2022.

[4] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. An mlir-based compiler flow for system-level design and hardware acceleration. In *41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD'22, page To appear, 2022.

[5] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *58th ACM/IEEE Design Automation Conference*, DAC'21, pages 1327–1330, 2021.

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'21, pages 2–14, 2021.

[7] Andrew B. Kahng and Tom Spyrou. The openroad project: Unleashing hardware innovation. In *Government Microcircuit Applications and Critical Technology Conference*, pages 1–6, 2021.

[8] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. High-level synthesis of parallel specifications coupling static and dynamic controllers. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS'21, pages 192–202, 2021.

[9] Marco Minutoli, Vito Giovanni Castellana, Nicola Saporetti, Stefano Devecchi, Marco Lattuada, Pietro Fezzardi, Antonino Tumeo, and Fabrizio Ferrandi. Svelto: High-level synthesis of multi-threaded accelerators for graph analytics. *IEEE Transactions on Computers*, 71(3):520–533, March 2022.

[10] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. Inter-procedural resource sharing in high level synthesis through function proxies. In *25th International Conference on Field Programmable Logic and Applications*, FPL'15, pages 1–8, 2015.

[11] AMBA AXI and ACE Protocol Specification. Technical report, ARM, 2021.

[12] David Kehlet et al. Accelerating innovation through a standard chiplet interface: The advanced interface bus (aib). *Intel White Paper*, 2017.

[13] Debendra Das Sharma. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *IEEE Micro*, 43(2):99–109, 2023.

[14] Debendra Das Sharma, Gerald Pasdast, Zhiguo Qian, and Kemal Aygun. Universal chiplet interconnect express (ucie): An open industry standard for innovations with chiplets at package level. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 12(9):1423–1431, 2022.

[15] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. Agile soc development with open esp. In *IEEE/ACM International Conference On Computer Aided Design*, ICCAD'20, pages 1–9, 2020.

[16] Mário P. Véstias João V. Roque, João D. Lopes and José T. de Sousa. Iob-cache: A high-performance configurable open-source cache. *Algorithms*, July 2021.

[17] Louis-Noël Pouchet and Tomofumi Yuki. Polybench/c 4.2.1, 2021.