2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00042

An Architectural Template for FPGA Overlays Targeting Data Flow Applications

Anna Drewes*, Vitalii Burtsev*, Bala Gurumurthy*, Martin Wilhelm*, David Broneske*,
Gunter Saake† and Thilo Pionteck*
Otto-von-Guericke-Universität Magdeburg, Germany
*firstname.lastname@ovgu.de, †saake@iti.cs.uni-magdeburg.de

***Abstract*—In this paper, we present a flexible and scalable architectural template for designing domain-specific FPGA overlays. The template can be parametrized in terms of number and size of tiles, communication topology and connectivity to external RAM. In addition to direct streaming connections between adjacent tiles for bulk data, it uses a novel lightweight packet-based on-chip network for small data transfers. The tiles consist of configurable routing resources and the actual compute units, which are exchanged at runtime by means of dynamic partial reconfiguration. This allows the assembly of custom, varying data paths for processing data flow graphs in hardware. Since every connection within and at the edge of the template is AXI-Stream-compliant, it is highly extensible using standard IP cores and streaming-based High-Level Synthesis code, enabling plug-and-play deployment.**

The feasibility of our approach is demonstrated by means of an domain-specific overlay for analytical database query processing on a Xilinx Alveo U280 card. TPC-H queries and synthetic workloads are used as test cases. Insights from this demonstration system show that the resulting overlay does not affect the performance of the compute units and show the effectiveness of the lightweight packet-based on-chip network in saving routing resources.

***Index Terms*—FPGA Overlay, Architectural Template, Dynamic Partial Reconfiguration, Data Flow Graphs, Analytical Query Processing**

I. Introduction

With the advent of larger and more complex FPGAs, the traditional design flow for FPGAs is increasingly becoming a barrier to their mainstream adoption. There have been many approaches to closing this gap, of which High-Level Synthesis (HLS) is one of the most promising. HLS allows FPGA accelerators to be designed at a high level of abstraction using software languages such as C, C++ or OpenCL. This greatly simplifies the design process and reduces development time. However, HLS-based accelerators often use large kernels [1], [2], which are designed for accelerating only a handful or just a single computationally intensive software function. For application domains where processing requirements change at runtime, this is not feasible.

Another approach to solving the usability problem of FPGAs, which has been gaining popularity [3], are overlay architectures. Here, the raw FPGA resources are abstracted by implementing a design that structures them. By partitioning an application into small reusable kernels that are mapped to an overlay architecture, the disadvantages of large, monolithic HLS or HDL kernels are avoided. Yet, the implementation of an overlay is a huge undertaking in terms of developer resources. Generic overlays can also be inefficient for some application areas.

We address both of these issues with our architectural template for deploying domain-specific overlays. Our approach targets streaming applications, where FPGAs offer unique advantages compared to other hardware accelerators. The template consists of a scalable parametrized 2D-tiled overlay with local streaming connections and provides a standardized interface for user kernels (compute units, CU). At runtime, CUs can be loaded into any tile via dynamic partial reconfiguration (DPR). The local data streaming links are supported by a novel lightweight on-chip network, which handles small data transfers, reducing the number of data flow paths that need to be reserved for each computation. To simplify kernel design, all interfaces are AXI-Stream-compliant. This enables scenarios where a fixed domain-specific overlay is quickly deployed on an FPGA and at runtime varying data flow graphs are executed by loading pre-synthesized kernels into neighboring tiles. Using this template, the task of implementing a data flow processing overlay for any application domain is transformed from extensive development work to straightforward parameterization and deployment. We demonstrate this by constructing an overlay for analytical database query processing based on the proposed template.

Thus, this paper proposes a generalized template, usable with different FPGAs, for instantiating custom overlays from a single code base, such that users only need to add application-specific kernels. The template combines local streaming connections with a lightweight network-on-chip (NoC) for transferring scalar values and configuration parameters, reducing the amount of data flow routes required for each task. It also provides a floor planning tool to aid design space exploration. A proof-of-concept overlay is instantiated and performance issues independent of the template are uncovered and analysed.

The rest of this paper is organized as follows: After an overview of related work in Section II, we present the architectural overlay template and its features in Section III. We then cover the deployment of a parameterized version of the overlay for the application domain of analytical query processing in Section IV. In Section V, we evaluate the overlay template

This work was partially funded by the DFG (Grant No. PI 447/9, SA 465/51).

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00042

based on the results from the demonstration system.

II. Related Work

Overlay architectures for FPGAs fall into three categories: First, static designs, which provide a set of programmable compute units. Second, overlay architectures, which do support dynamic partial reconfiguration of compute units, but feature highly application-specific design choices. Third, generic reconfigurable overlay architectures that are suitable for a wider range of applications.

Many works in the first category target a specific task, such as [4]–[6] for neural networks. They trade the generality of FPGAs for tighter domain integration. Other work focuses on extracting kernels from software to automatically assemble FPGA overlays [7]. This is mainly useful for accelerating legacy code bases.

Coarse-grained reconfigurable architectures (CGRAs) such as [8], [9] also fall into the first category. Typically, they cannot change the supported operations at runtime. Just-in-Time (JIT) compilation of prepared compute units, as proposed in [9] can help to adapt a CGRA to preexisting software, but without DPR this is only possible at design time. Capalija and Abdelrahman propose a 2D grid-shaped overlay for data flow graph computations targeting Intel Stratix IV FPGAs [10]. Connectivity between cells is provided in a 4-neighborhood. Dedicated integer or floating point single-function units are placed into the overlay at design time according to a static pattern. FLexiTASK [11] is an FPGA overlay that demonstrates the interconnection of compute units through a Network on Chip (NoC). This increases flexibility and communication utilization efficiency because data flow routes do not require exclusive paths. However, the need to generate packets for each chunk of data to be sent creates huge runtime overhead and complicates application design. As a demonstration, a 5×5 network was implemented on an Altera Cyclone V FPGA, with four CUs per router, outperforming an embedded ARM core.

Examples of the second category are the works of Ziener et al. [12] and Schmid et al. [13]. Ziener et al. propose an FPGA accelerator for analytic query processing that targets row-oriented storage. Their system consists of 4 areas with reconfigurable partitions and is implemented on Xilinx Virtex-6 and Zynq-7000 FPGAs. Each partition has only a single streaming input and output. Thus, data must always move in one direction through all 16 reconfigurable partitions of an area. With row-oriented storage, reorder units are required for fitting data to CUs.

Schmid et al. present an automated FPGA accelerator design process for in-memory database query processing. HLS-generated kernels implementing a fixed interface with AXI-Stream data ports and an AXI-Lite configuration interface are combined into a complete FPGA design at the HDL level. The kernels can process data from memory or from other processing units via a global AXI-Stream crossbar. While the use of standard interfaces facilitates plug-and-play deployment, the static design and centralized style of interconnection limit usability and scalability.

The work of Backasch et al. [14] belongs to the third category. They propose a regular structure of processing modules interconnected by a NoC. The processing elements contain run-time reconfigurable elements. Thus, application-specific data paths can be generated at runtime. While this approach is similar to our work, Backasch et al. are limited in terms of flexibility by a fixed number of inputs and outputs. The design of the reconfigurable elements is also hand-crafted and does not rely on standard design flows.

III. Architectural Overlay Template

The proposed architectural overlay template is designed to support a wide range of application domains where the algorithms can be represented by data flow graphs. It is completely implemented in HDL, highly configurable and device-independent. To support HDL, IP and HLS kernels, AXI-Stream is used as communication interface throughout the overlay architecture. At deployment time, the number and size of tiles, internal communication resources, external interfaces and static hardware accelerators can be parameterized. At runtime, kernels are exchanged and communication paths configured, thus enabling the execution of data flow graphs. This allows the overlay to be extensively tailored to the requirements of a particular application domain, while keeping the hardware overhead low. If requirements change, or new tasks are devised, further compute units can be implemented at runtime.

An overview of the overlay template, embedded into a typical FPGA accelerator environment, is given in Figure 1. It consists of an array of tiles arranged in a 2D grid (dark gray), each of which contains a reconfigurable partition for a streaming-based compute unit (CU). The template also provides parametrized support infrastructure (light gray) for attaching DMA IPs to the edges of the overlay. The template allows processing multiple streams in parallel. The memory subsystem and other IP, such as a PCIe core for connection to the host, are not part of the proposed device-independent overlay template. We use vendor IP for these tasks (orange).

A. Communication Infrastructure

The communication topology must be defined at instantiation time. Several topologies are supported by our template, ranging from 2D feed-forward-only designs to full 8-neighborhood interconnection of tiles, as shown in Figure 3. The number of input and output streams per tile is configurable, as is the bit width.

A special feature of our overlay template is the lightweight, packet-based on-chip network. It is designed to carry all small data transfers in the system with minimal area overhead. In Figure 1 it is shown with dashed lines, starting at the host connection. Through this network, user software can communicate with all DMAs, tiles and CUs to issue commands, set parameters and independently receive small results or error codes. “Streams” travelling through the network do not require data flow routes, thereby reducing area overhead. The physical layer is based on AXI-Stream and each packet is preceded by

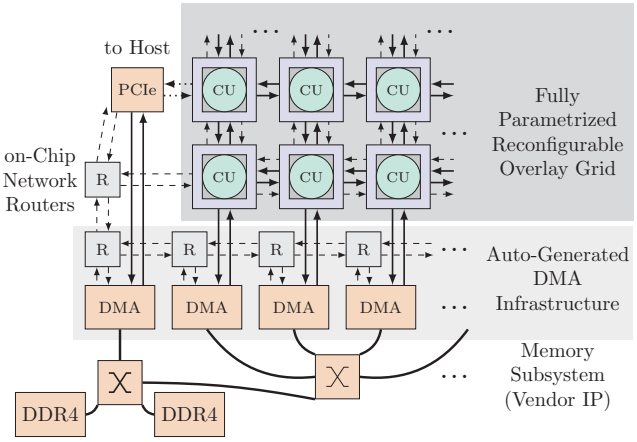

Fig. 1. Block diagram of the overlay architecture template (shaded) with supporting IP (orange).

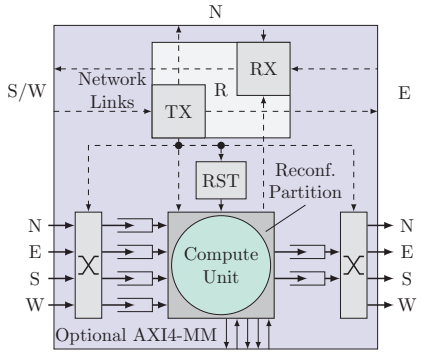

Fig. 2. Overlay tile architecture with buffered local data streaming connections, packet-based on-chip network and optional AXI4-MM port. Configuration as deployed in Section IV.

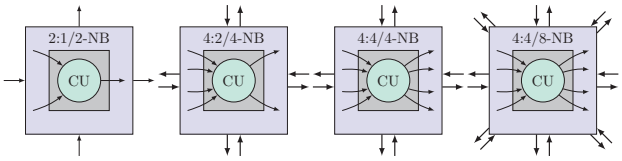

Fig. 3. Four exemplary streaming data grid topologies supported by the overlay template, described by $a{:}b/x$ configuration tuples representing CU Inputs, Outputs, and Tile Neighborhood.

a head flit containing routing information. To achieve the low resource footprint, the network is split into two separate paths, host-to-overlay and overlay-to-host. Compared to a bus with memory-mapped register access, the network is much more scalable in terms of resource usage, timing and throughput.

B. Tile Architecture

The internal structure of a tile is shown in Figure 2. Each tile contains data flow routing resources and a reconfigurable partition (RP) for a streaming-based compute unit (CU). A standardized interface at the RP boundary allows arbitrary CUs to be loaded into the RP at runtime. This interface mainly consists of high-bandwidth streaming ports, shown to the left and right of the partition. They are buffered for timing isolation and to deal with pipeline effects. For support of kernels requiring random memory access, each RP can be individually equipped with an additional full AXI4 Memory Mapped (AXI4-MM) interface at design time. Due to the increased cost and limited scalability, this is a secondary feature. The number and width of all ports is configurable at deployment time. The size of the reconfigurable partition within a tile can also be set according to the requirements of the application domain. Tiles with and without support for AXI4-MM-enabled CUs can have different RP sizes.

Finally, each tile contains a router pair for the on-chip network mentioned above. These are shown at the top of Figure 2. The network is used for runtime setup of the data flow routing and low-bandwidth data transfers to and from the CU. As the network interface is standardized, AXI-Stream-compliant and quickly implemented in HDL and HLS, CUs can easily be equipped with this additional communication option.

Within a tile, data flow can be routed freely via small custom AXI-Stream crossbars. This allows significant simplification of the CUs. The template provides predefined configurations as shown in Figure 3, ranging from only two CU inputs and one CU output (2:1) with tiles connected to only two immediate neighbors (2-NB), to 4 inputs/outputs per CU (4:4) and connections to all immediate neighbors (8-NB).

The buffers for the streaming ports can be full-throughput pipeline registers or FIFOs. In general, registers are sufficient to break all timing paths. However, having only a double buffer stage can easily propagate stalls, especially for unbalanced data flow graphs. It is up to the user to select one of the provided options, trading off resource impact versus capacity.

C. Floor Planning

Many aspects of the overlay template are meant to be tailored to the requirements of an application domain. One critical parameter that usually requires extensive design space exploration when developing a reconfigurable overlay is the size of the reconfigurable partitions to host the CUs. Large partitions simplify kernel design, but result in a high resource overhead. Small partitions improve scalability, but can limit future CUs if too few extra resources were allocated.

Finding the number of tiles for a certain RP size that are supported by the target FPGA and deriving a suitable layout usually requires in-depth knowledge. The overlay architecture template addresses this problem with a custom floor planning tool build on top of Xilinx RapidWright [15]. An optimistic (not greedy) approach combined with tight integration into the overlay parametrization work flow allows even non-expert users semiautomatic iterative floor planning.

IV. Domain-Specific Demonstration System

In this section we describe the deployment of a domain-specific overlay targeting analytical database query processing based on the proposed architectural template. The domain-specific overlay instance is created by parametrizing our overlay template and implementing the required kernels in C++ using Vivado HLS. The target platform is the Xilinx Alveo U280 data center accelerator card.

A. Analytical Database Query Processing

Database queries are typically written in specialized languages such as SQL. The statements are parsed and graphs of operations are generated with data dependencies as edges. These graphs correspond to the data flow graphs that serve as input to our system. Since new queries are created at runtime, using a static overlay for hardware acceleration is not feasible. In a design based on our overlay template, a host analyzes the data flow graph and maps it onto the overlay by loading kernels/CUs into the RPs and sending commands through the on-chip network to set up the data flow paths using the routing resources of the tiles. Therefore, instead of only supporting a fixed set of queries, a library of pre-synthesized kernels implementing all common database operators can be used to execute new queries arriving at runtime.

For the implementation of the database operators we follow the concept of database primitives as proposed in [16]–[18]. Complex computations are broken down into small, reusable, data-parallel function blocks. To store the data in memory, we chose a column-oriented format [19]. Here, data is organized in arrays of fields. Compared to row-oriented storage, this is more flexible and requires processing only the fields required by the current query. This approach also avoids the need for parsing or shuffling hardware that would be required for row-oriented storage.

The evaluation of query processing systems is commonly done using the TPC-H benchmark [20]. We have identified and implemented 31 primitives needed to realize the queries of this benchmark. Whenever possible, we use 4-way SIMD data parallelism for the primitives. The kernels have been compiled into a library.

B. Parametrization

To find a suitable communications topology, we analyzed typical TPC-H query data flow graph and primitive characteristics. We chose the 4:2/4-NB configuration for the demonstration system with each link 128 bits wide, allowing for four 32-bit words to be streamed in parallel. Four stream inputs is a sufficient number to allow for some degree of code fusion, i.e., merging commonly used combinations of primitives into larger operators [21]. Two stream outputs per CU allow data streams to be forked and crossed, reducing the number of tiles required for each mapping.

To determine the RP size, we synthesized all the primitives for the target device. The results are given in Table I. The largest streaming CU is the data-parallel ordered aggregation primitive, while the smallest CU is a dual FIFO, which is used when a mapping requires a tile just for data forwarding. The largest CU with the additional AXI4-MM interface is a AXI DataMover IP. Loading DMAs as a CUs into RPs provides extra channels for streaming data between memory and the overlay. Based on these maximum CU sizes, we set the Reconfigurable Partition (RP) size to 1440 LUTs for pure AXI stream tiles and to 2880 LUTs for RPs supporting CUs that include an AXI4-MM interface. The RP size is also constrained by the characteristics of the target FPGA.

TABLE I
RECONFIGURABLE PARTITION (RP) SIZE IN RELATION TO CU RESOURCE USE

	LUT total	*as Memory*	*FF*	*DSP*	*Throughput (GB/s)*
Streaming RP	**1440**	**<960**	**2880**	**24**	
Max. Used	662	160	1689	12	12.0
Min. Used	175	0	342	0	1.0
Str. + MM RP	**2880**	**<1440**	**5760**	**24**	
Max. Used	2736	487	3095	12	12.0

Off-the-shelf plug-and-play IPs were used for the memory subsystem and PCIe. PCIe connectivity and kernel drivers are provided by the XILLYBUS Rev. B IP [22], which provides a throughput of $1.6\,\mathrm{GB/s}$. The XILLYBUS IP also transports packets of the on-chip network between the user application and the root network node shown on the left in Figure 1. Faster PCIe cores are available and can be combined with the overlay template. Dynamic partial reconfiguration is handled over PCIe according to Xilinx guidelines [23]. The ICAP is used to upload partial bit streams at full throughput.

The demonstration system is implemented with a clock speed of 250MHz. To improve timing, the AXI Interconnect between the DMA engines and the DDR4 memory controllers is implemented in two layers, as shown in Figure 1. Eleven AXI DataMover IPs are connected to two edges of the overlay instance, streaming 11×128-bit data words into and out of the overlay. On the other side, these DMA engines are connected to a 512-bit AXI interconnect IP. With each of the two DDR4 controllers providing a $512\,\mathrm{bit}$ bus, this results in a theoretical maximum memory bandwidth of 8 parallel data streams into or out of the overlay before bus contention should start to limit performance.

C. Synthesis Results

For the demonstration system we chose an 11×4 tile grid, mainly to keep initial place and route time to a reasonable level. The floor planning tool was used to create a suitable layout of tiles/partitions.

The complete FPGA design occupies 259524 LUTs, 307221 FFs, 1065 DSPs and 201 BRAMs. Approximately half of this is covered by vendor-based infrastructure IPs (DDR4, interconnect, DMAs, PCIe, XILLYBUS). The actual overlay instance uses 134011 LUTs, 132703 FFs, 1056 DSPs and 20 BRAMs. These numbers include dummy CUs ($4\times32\,\mathrm{bit}$ multiply with reduce-add) loaded into every reconfigurable partition for the initial implementation step, which are used to force the place and route algorithm to efficiently place all partition boundary signals. Each tile, excluding the compute unit in its reconfigurable partition, requires 2287 LUTs (of which 480 are used as memory) and 1162 FFs. For the lightweight on-chip network, a pair of routers (upstream and downstream) requires only 65 LUTs and 77 FFs.

V. EVALUATION

For evaluation, we demonstrate the usage of our template based on two challenging queries from the TPC-H benchmark. Query 1 contains many different grouped aggregation outputs and its data flow graph is larger than the 4×11 demonstration

system. Query 6 is computationally intensive and hard to accelerate [24]. The placement of Query 6 is shown in Figure 4.

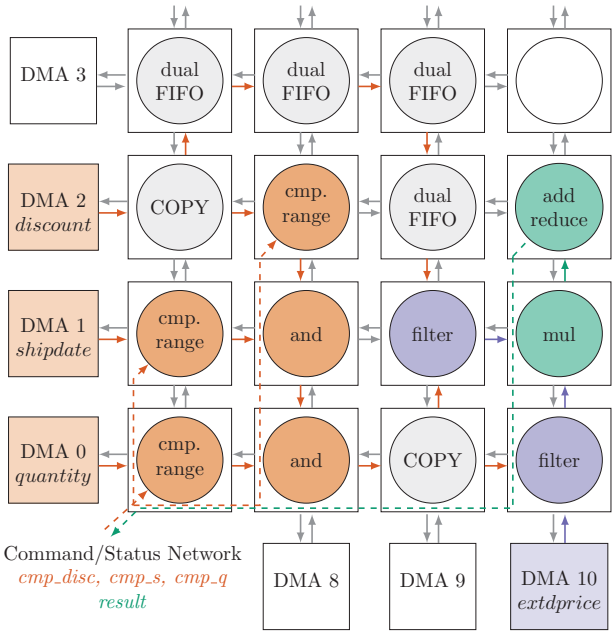

Fig. 4. Mapping of TPC-H Query 6 to the 11×4 demonstration system. Shaded tiles/DMAs are occupied. Data columns are written in cursive.

Four DMAs stream data through 15 CUs. The data flow routes and CUs are colour-coded. Orange represents the processing of the selection criteria, blue stands for materialization and green for the calculation of the query answer. Support CUs are highlighted in gray. The dashed arrows show data transferred via the on-chip network instead of conventional tile-to-tile connections. The shown mapping was derived manually to provide a clear layout. Capalija and Abdelrahman provide a place-and-route (PAR) algorithm based on standard simulated annealing tools for CGRA-like architectures that offers sufficient performance [10]. Due to the utilization of deep FIFO buffers for the streaming connections, our work requires even less effort towards a balanced DFG mapping.

TABLE II
OVERLAY RESOURCES OCCUPIED BY TPC-H QUERIES

	Tiles Occupied	*DMA Data Flow Streams*	*Network "Streams"*	*DPR (ms)*
Query 1 a	31	11	17	20.2
Query 1 b	32	4	28	12.1
Query 6	15	4	4	25.7

Table II shows numbers for the placement of Query 1 and Query 6. Due to its size, the data flow graph of Query 1 is split into two subgraphs (a, b), which run sequentially. In contrast, Query 6 uses less than half of the available tiles and could in principle be executed twice in parallel.

Table II also highlights the advantage of the on-chip network: The third column gives the number of I/O data streams implemented by DMAs, while the fourth column shows the number of data streams carried by the on-chip network. Without the on-chip network, the sum of both columns would have to be mapped to data flow routes and DMA channels. The on-chip network significantly reduces this amount. This simplifies graph mapping and reduces the need for support tiles and DMA channels.

In the last column of Table II, the partial reconfiguration times for the queries are shown. Time is measured end-to-end by the supervising user software. The differences between queries are due to different average bitstream sizes, as the queries do not occupy exactly the same set of tiles/CUs.

A. Performance Anomaly

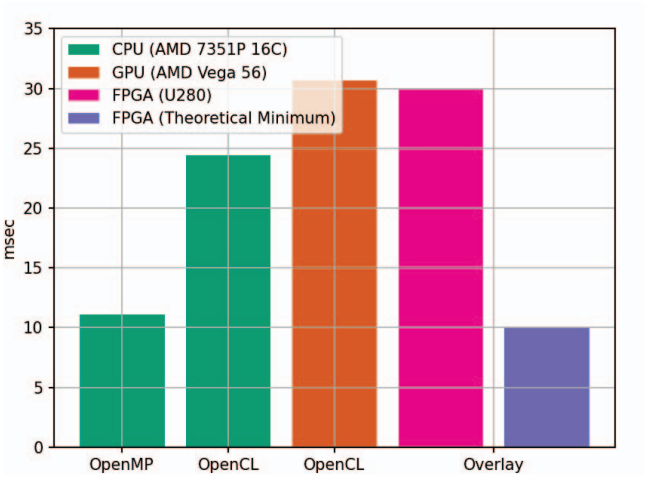

Fig. 5. Runtime of TPC-H Query 6 compared to other primitive-based implementations, at 10 million rows of input data.

Execution times were measured end-to-end from the issuing of commands by the user software to their completion. For CPU/GPU this includes thread dispatch and join. For the GPU and FPGA, the times include the latency of the kernel drivers and PCIe transfers of commands. Since all devices contain sufficient memory for the entire data set, only execution time is considered, not the initial data load. Figure 5 shows the measured runtime of Query 6 for 10 million rows on a 16-core AMD 7315 CPU, on an AMD Vega 56 GPU, and on the demonstration system implemented on the Xilinx Alveo U280 card. For the CPU, the queries are implemented using OpenCL and OpenMP. All implementations are based on the same processing principle using database primitives, so that the results are comparable. In addition to the measured values, Figure 5 shows the theoretical minimum runtime of the FPGA implementation. Due to the difficulty of component-level power measurements in off-the-shelf compute nodes [25]–[27], we can only report rough estimates. CPU and GPU data is provided by the vendor, while data for the FPGA card is extrapolated from the tool report. Power consumption for the demonstration overlay instance is 15 W to 25 W for the U280 PCIe card, including memory, compared to 150 W for the CPU (under load) and 250 W for the GPU.

It can be seen that while we didn't achieve acceleration for Query 6 compared to an optimized OpenMP version on the PC, energy consumption does favor the FPGA. Also, by design, the

demonstration system uses only 1/6 of the available FPGA resources for the overlay, and Query 6 can be instantiated twice even with the 11×4 grid size. Thus, there options for further optimization on the FPGA. Since the goal of the demonstration system was to illustrate the application of the proposed overlay template, we present an investigation into the performance anomaly encountered here. This will also provide insights that are applicable outside the scope of this work.

B. Memory Subsystem

The theoretical performance values are based on CU simulation data, which predicted one data beat per clock cycle and no stalls in the overlay. To eliminate potential pipeline stall effects, i.e., data dependencies between CUs, we created a synthetic benchmark with multiple independent CUs implementing reduce operations. All kernels have been verified in simulation to achieve full throughput. They are mapped to tiles adjacent to the DMAs, ensuring that there is no interaction between DMA channels and also no paths through the overlay. The synthetic benchmark is only $0.0001\,\%$ faster than Query 6 and $3.8\times$ slower than the ideal value, leading to the conclusion that the performance bottleneck is not caused by data dependencies within the overlay. Rather, it appears that the CUs are stalled because data was not delivered in time by the DMA engines.

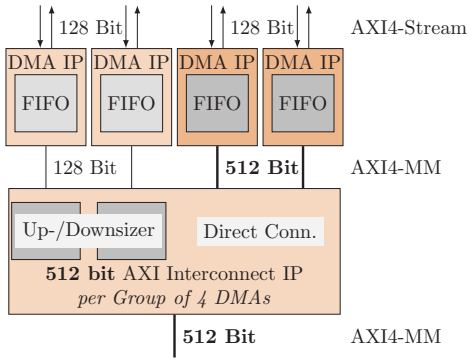

Fig. 6. Connections between DMAs and system interconnect.

As a possible source, we identified the bit width conversion between the 128-bit links and the 512-bit AXI interconnect IP, as shown in Figure 6. On the left, data width converters within the local AXI Interconnect IP are used to step up the 128-bit DMA interface to the 512-bit bus crossbar. This version is implemented in our demonstration system and results in the performance anomaly shown in Figure 5. As an alternative, we also evaluated the configuration on the right side of Figure 6. Here, the native memory-side DMA interface is set to 512 bits and the data width converter is removed from the AXI Interconnect IP. Although the performance of both variants should be identical, in real-world tests, we only achieved a streaming channel utilization close to $100\,\%$ with the latter variant. However, this performance improvement comes at the expense of resource utilization. While the first variant is resource efficient, in the second variant the BRAM-based FIFO buffer within the DMA has to be expanded, which increases that resource utilization by a factor of 2.4. It depends on the application domain whether this cost is appropriate.

VI. Conclusion

In this work, we present an overlay template that allows for a fast and user-friendly creation of overlay architectures for the realization of arbitrary data flow graphs at runtime. Through a demonstration system for analytical query processing, we show the feasibility of our approach. Performance analysis of TPC-H Query 6 and a synthetic benchmark shows that the overlay template does not affect the throughput of the kernels. Furthermore, the placement results show that the proposed lightweight on-chip network allows transporting small data apart from the local interconnections with minimal resource use, freeing up tiles for computation. Further research is required into the impact of the memory system on overall performance. However, this problem is about IP, and hence, orthogonal to the template. The consistency of the template's kernel interface with all current design techniques makes it easy to deploy. Together with the high flexibility, this makes it a good solution in application domains where user requirements may change on the fly.

References

[1] "Intel FPGA SDK for OpenCL," https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, accessed: 2023-01-10.

[2] "Xilinx: Vitis software development workflow," https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Vitis-Software-Development-Workflow, accessed: 2023-01-10.

[3] X. Li, C. F. Phung, and D. L. Maskell, "FPGA overlays: Hardware–based computing for the masses," in *Eighth International Conference On Advances in Computing, Electronics and Electrical Technology (CEET)*, 2018, pp. 25–31.

[4] R. Shi, Y. Ding, X. Wei, H. Li, H. Liu, H. K.-H. So, and C. Ding, "FTDL: A tailored FPGA-overlay for deep learning with high scalability," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[5] Z. Choudhury, S. Shrivastava, L. Ramapantulu, and S. Purini, "An FPGA overlay for CNN inference with fine-grained flexible parallelism," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 3, may 2022. [Online]. Available: https://doi.org/10.1145/3519598

[6] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2020.

[7] J. Mandebi Mbongue, D. Tchuinkou Kwadjo, and C. Bobda, "Automatic generation of application-specific FPGA overlays with RapidWright," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 303–306.

[8] S. A. Chin, K. P. Niu, M. Walker, S. Yin, A. Mertens, J. Lee, and J. H. Anderson, "Architecture exploration of standard-cell and FPGA-overlay CGRAs using the open-source CGRA-ME framework," in *Proceedings of the 2018 International Symposium on Physical Design*, ser. ISPD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 48–55. [Online]. Available: https://doi.org/10.1145/3177540.3177553

[9] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Coarse grained FPGA overlay for rapid just-in-time accelerator compilation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1478–1490, 2022.

[10] 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*, 2013, pp. 1–8.

[11] J. Mandebi Mbongue, D. Tchuinkou Kwadjo, and C. Bobda, "FLexiTASK: A flexible FPGA overlay for efficient multitasking," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 483–486. [Online]. Available: https://doi.org/10.1145/3194554.3194644

[12] D. Ziener, F. Bauer, A. Becher, C. Dennl, K. Meyer-Wegener, U. Schürfeld, J. Teich, J.-S. Vogt, and H. Weber, "FPGA-based dynamically reconfigurable SQL query processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 4, aug 2016. [Online]. Available: https://doi.org/10.1145/2845087

[13] R. Schmid, M. Plauth, L. Wenzel, F. Eberhardt, and A. Polze, "Accessible near-storage computing with FPGAs," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387557

[14] R. Backasch, G. Hempel, C. Blochwitz, S. Werner, S. Groppe, and T. Pionteck, "An architectural template for composing application specific datapaths at runtime," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015, pp. 1–6.

[15] "Xilinx: RapidWright," https://www.rapidwright.io, accessed: 2023-01-10.

[16] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner, "An application-specific instruction set for accelerating set-oriented database primitives," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 767–778. [Online]. Available: https://doi.org/10.1145/2588555.2593677

[17] B. Gurumurthy, D. Broneske, T. Drewes, T. Pionteck, and G. Saake, "Cooking DBMS operations using granular primitives - an overview on a primitive-based RDBMS query evaluation." *Datenbank-Spektrum*, vol. 18, no. 3, pp. 183–193, 2018. [Online]. Available: http://dblp.uni-trier.de/db/journals/dbsk/dbsk18.html#GurumurthyBDPS18

[18] M. Gowanlock, B. Karsin, Z. Fink, and J. Wright, "Accelerating the unacceleratable: Hybrid CPU/GPU algorithms for memory-bound database primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19. New York, NY, USA: Association for Computing Machinery, 2019.

[19] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*, 7th ed. New York: McGraw-Hill, 2019. [Online]. Available: http://www.db-book.com/

[20] "TPC-H decision support benchmark," https://www.tpc.org/tpch, accessed: 2023-01-10.

[21] A. Drewes, J. M. Joseph, B. Gurumurthy, D. Broneske, G. Saake, and T. Pionteck, "Optimising operator sets for analytical database processing on FPGAs," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 16th International Symposium, ARC 2020, Toledo, Spain, April 1-3, 2020, Proceedings [postponed]*, ser. Lecture Notes in Computer Science, F. Rincón, J. Barba, H. K. So, P. C. Diniz, and J. Caba, Eds., vol. 12083. Springer, 2020, pp. 30–44. [Online]. Available: https://doi.org/10.1007/978-3-030-44534-8_3

[22] "XILLYBUS: An FPGA IP core for easy DMA over PCIe with Windows and Linux," http://xillybus.com/, accessed: 2023-01-10.

[23] "Xilinx: Fast partial reconfiguration over PCI Express (XAPP1338), introduction," https://docs.xilinx.com/r/en-US/xapp1338-fast-partial-reconfiguration-pci-express/Introduction, accessed: 2023-01-10.

[24] A. Lu and Z. Fang, "SQL2FPGA: Automatic Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 184–194.

[25] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 194–204.

[26] R. A. Bridges, N. Imam, and T. M. Mintz, "understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods," *ACM Comput. Surv.*, vol. 49, no. 3, September 2016.

[27] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU Power with the K20 Built-in Sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: Association for Computing Machinery, 2014, p. 28–36.