SPADES: A Productive Design Flow for Versal Programmable Logic

Tan Nguyen
UC Berkeley
tan.nqd@berkeley.edu

Zachary Blair
AMD
zachary.blair@amd.com

Stephen Neuendorffer
AMD
stephen.neuendorffer@amd.com

John Wawrzynek
UC Berkeley
johnw@berkeley.edu

***Abstract*—With the increasing growth of complexity and heterogeneity of modern FPGA fabrics, the conventional "flat" design flow relying on the standard tools, from Synthesis, Implementation, to Bitstream Generation, has become more arduous than ever. This leads to an inordinate turn-around time which severely impacts the productivity of application developers in quest of design space exploration. We propose an open-source tool flow built around a customizable overlay of Spatially Distributed Socket Engines (SPADES) to address the FPGA productivity issue. SPADES organizes the computation and communication of an application in a parallel, distributed execution of socket engines. To tackle the compilation time issue, we exploit the hardened Network-on-Chip present in Versal, a novel commercial FPGA architecture from AMD, to alleviate the inter-socket routing task, as well as implement reusable socket netlist by utilizing regular programmable fabric regions. We use three data-parallel benchmarks to demonstrate that our tool flow achieves shorter compilation time than the standard, top-down AMD Vitis flow by 7x on average (from hours down to minutes) with comparable or better performance on the AMD Versal VCK5000 data center card.**

I. Introduction

FPGAs have increasingly gained their footings in many applications ranging from data centers in the clouds to embedded devices of the edges thanks to the inherent flexibility and spatial parallelism nature. To meet the growing computation and communication demands, FPGA vendors have been augmenting their devices with millions of logic elements and a hoard of heterogeneous hardened blocks. This further exacerbates the longstanding, pathological FPGA tooling issues which severely affects their usability and inhibits their accessibility and applicability outside the niche community of hardware design experts. In particulars, vendor tools may consume hours or days to generate a bitstream of a hardware implementation for some large and complex design. If the implementation were to fail to satisfy user-defined constraints, such as desirable operating frequency, area, or functionality, the whole development cycle would have to repeat, inevitably hurting the productivity. Therefore, we think that the FPGA compilation time is of the utmost important problem to be addressed as we observe the likely continual trend of future FPGA device and design scaling.

To mitigate the compile time issue, one could virtualize an FPGA with overlays. The overlay technique raises the abstraction level by allowing users to program FPGA using a higher-level programming language than HDLs in tandem with a software-like compilation experience, thereby improving the overall productivity. Yet, the key weakness of the Overlay approach is its suboptimal Quality-of-Result (QoR) in comparison to custom hardware design. Therefore, overlays are often domain-specific [1], [2], [3], and it is challenging to generalize an overlay design to a variety of applications.

The goal of this work is achieving a minute-scale compile time (i.e., from application to placed-and-routed netlist) with little or no impact to the Quality-of-Result. One common best practice of shortening the compile time is modularizing an application to smaller modules for separate compilation. This is even more advantageous in cases where some are reusable. An example is data-parallel kernels where one could launch multiple kernel invocations to operate on different chunks of data in parallel. Therefore, it is possible to reuse a kernel design to create an array of parallel processing elements. In this work, we focus on mapping applications that exhibit such data-parallel nature. We synthesize an application to a system of parallel socket overlay engines interconnected by a Network-on-Chip (NoC). Our socket overlay is *customizable*: there is a fixed, run-time programmable logic tightly coupled with a customizable partition. A prudent floorplanning process of a socket implementation is carried out to maximize its reusability across different fabric regions of the target FPGA architecture.

In summary, our contributions are as follows.

1. We build an open-source[1], fast, and agile FPGA design flow. Our toolchain enables fast and efficient mapping of data-parallel applications written in High-level Synthesis (HLS) C/C++ to FPGAs.

2. We demonstrate the effectiveness of our flow in the compilation time and performance over the standard AMD Vitis tool for several data-parallel benchmarks targeting the AMD Versal VCK5000 data center card.

II. Methodology

A. SPADES abstraction

SPADES presents a parallel system of distributed sockets. As illustrated in Figure 1, a socket is a processing engine that contains a Control Unit and a private Memory block in addition to a Compute kernel. The sockets operate independently, and communicate with one another and the

[1]Our work is available at: github.com/nqdtan/spades

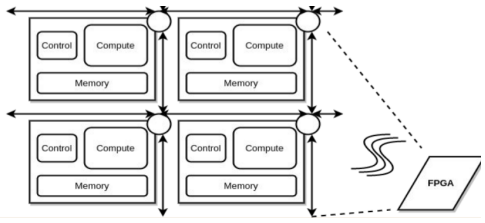

Fig. 1. SPADES featuring a parallel, distributed system of socket engines.

memory subsystems through an interconnect network. The model is sufficiently flexible to adapt to different parallelism models, such as Task-level parallelism (each socket executes different tasks), or Data-level parallelism (all sockets runs the same program, but operates on different workloads.). In this work, we target data-parallel benchmarks implemented in HLS C/C++. We apply loop tiling for data reuse and parallelize the outermost loop with multiple sockets by which each works on a tile of data in parallel. The sockets may need to synchronize their executions in case of data dependence.

B. Versal Network-on-Chip

Versal is the latest AMD FPGA architecture featuring a novel hardened NoC [4], [5]. The NoC presents a unified communication backbone with high-bandwidth data transfer between the Programmable Logic (PL), the Processing Systems, the DDR, and other platform subsystems. On the PL, the NoC endpoints, namely NoC Master Unit (NMU) and NoC Slave Unit (NSU) are laid out in columns. An NMU issues requests from the PL to the network, whereas an NSU services requests from the network to the PL. The NoC supports both AXI Memory-mapped and AXI Stream interface. Regarding our target FPGA device, the hard NoC runs at 1GHz with a 128-bit datapath. Hence, the maximum bandwidth of an NoC link is 16GB/s.

C. Socket Microarchitecture Design

Figure 2 shows the socket microarchitecture that consists of two components: Socket Control & Communication (CC), and Socket Custom Logic (CL). Socket CC is a fixed overlay logic, while Socket CL is customized to application-specific compute and memory demands.

1) Socket Control & Communication: A socket relies on the NoC endpoints to communicate with the NoC. As the NoC endpoints use AXI interface, we implement an AXI adapter, one for each endpoint present in a socket. In this work, we use AXI Memory-mapped interface, since our target applications require communication with the DDR. The sockets may possibly need to communicate with each other. As such, each socket is assigned a unique address in the system.

The AXI datawidth of an NMU and NSU on Versal PL is 512-bit. To reach the full NoC bandwidth of 16GB/s, the PL AXI-interface logic must either reach 500MHz at 256-bit, or 250MHz at 512-bit. We adopt the latter in our AXI adapter design since this timing target is more feasible; it would be challenging to meet 500MHz given the AXI logic needs to have adequate buffering for overlapping multiple in-flight requests to attain high throughput data transfer.

The DMA engine provides configurations to setup the M-AXI adapter for data exchange between external memory and on-chip memory blocks of the socket. The DMA is user-programmable supporting both block and cyclic access modes. For instance, it could fetch a rectangular tile from external memory to a group of on-chip RAMs whereby each RAM holds one row of the tile.

To program the DMA engines and Socket CL, we design a small soft core controller. Our soft core is a simple, 32-bit integer in-order RISC-V core (RV32I). The components of the socket are registered in a memory-mapped IO (MMIO) register bank. This allows us to write a single-threaded software code to control the operations of the socket at runtime through the MMIO logic. Moreover, besides setting up necessary runtime parameters, the softcore schedules the communication (DMA) operations and computation (Socket CL) executions of an application by a task queue (order of execution determined by user). The softcore could also provide a synchronizing mechanism with another socket if needed by exchanging messages via the NoC.

2) Socket Custom Logic: Socket CL implements the computation and memory requirement of an application. Instead of a monolithic custom logic module, we design the compute and memory logic separately, since the Socket CC's DMA engines also need access to the memory logic. The memory logic comprises of multiple memory blocks implemented by the on-chip RAM resources on the PL. The compute logic reads or writes data from the memory logic via RAM interfaces. In this work, we use HLS to design the compute logic. The memory logic is generated by a memory unit specification in which we configure the number of RAMs, their depths and types (BRAM/URAM/LUTRAM), and their connectivities with the compute logic's ports and DMA engines.

3) Flip-flop Bridge: We use the RAM interfaces for the communication between Socket CC and Socket CL, since the access latency between Socket CC and Socket CL are statically known by design. A RAM interface enables the DMA engine in Socket CC to access data of the memory logic from Socket CL. Likewise, the controller of Socket CC could write or read the MMIO registers of the compute logic of Socket CL through another RAM interface.

The RAM interfaces between Socket CC and CL are registered to improve timing by the *Flip-flop bridge*. The bridge, along with Socket CC, are pre-implemented and reused to reduce the compile time. The bridge anchors the interface of Socket CL to Socket CC's: we only need to route the interface of Socket CL to its side of the bridge.

Lastly, we leverage multi-clock domain design for our socket microarchitecture. In particular, the entire Socket CC region is clocked at half the speed of Socket CL (e.g., 250MHz vs. 500MHz). The rationale is that the AXI logic could achieve full bandwidth with 512-bit at 250MHz, and the wide-width AXI bus is capable of filling (or reading) multiple on-chip RAMs in parallel. In addition, for many data-parallel appli-

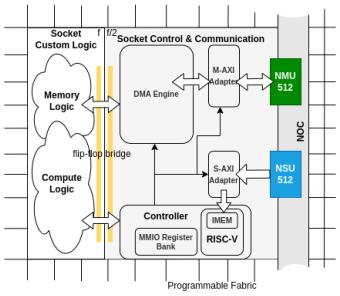

Fig. 2. Socket Microarchitecture. The thick arrows denote the direction of data flows (e.g., read/write, request/response). The thin black arrows denote control relationships.

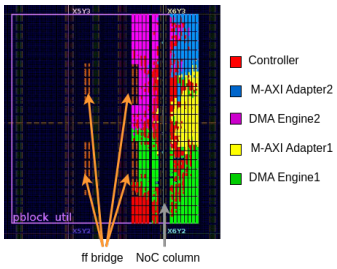

Fig. 3. Socket CC pre-implementation. There are two NMUs and one NSU in this implementation. The remaining logic of the PBlock is reserved for Socket CL netlist.

TABLE I
SOCKET AREA

Resource	socket_m	Socket CC util.
LUT	51,520	37.9%
FF	103,040	15.6%
BRAM+URAM	69+23	0
DSP	184	0
NMU	2	100%
NSU	2	50%

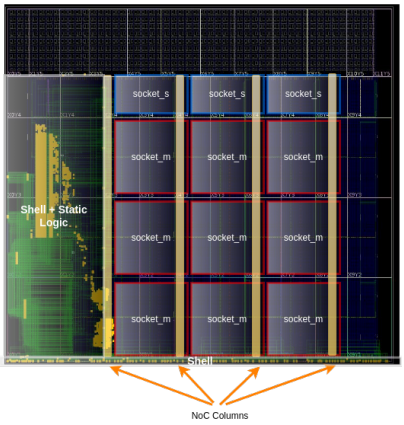

Fig. 4. SPADES floorplaning scheme. *socket_m* denotes medium-sized socket, whereas *socket_s* denotes small-sized socket. The leftmost NoC column is already used by the shell and static logic. *socket_s* is roughly half the size of *socket_m*.

cations, the control operations (e.g., task sequencing, general address/scalar parameter computations) are typically not on the critical path of the performance (as time spent mostly on data copy or computation). Therefore, it might be possible to slow down the non-critical portion of the design without severely affecting the overall performance. At the same time, the tool could expend more effort on optimizing the critical regions.

D. Socket Physical Implementation

Our target device is a single-SLR (Super Logic Region) architecture. We first identify fabric regions that have repeatable patterns to implement a socket. The guiding principle is relocatability: a socket that has already been implemented on a particular region should be relocatable to a different, compatible region without re-implementation.

One issue that may arise is route conflict between adjacent sockets. We found that, despite the CONTAIN_ROUTING constraint enabled for the socket floorplan (or PBlock), Vivado router still expands the routes to one INT tile column outside the Pblock. We work around this problem by leaving a gap of one CLB-tile column between the socket Pblocks.

We align a socket to one NoC column (NMU and NSU on the same column). This would enable us to implement Socket CC compactly to one side of the floorplan, and leave the other side to the custom logic as demonstrated in Figure 3.

The next problem is to determine the size of a socket, and the maximum number of sockets our flow can possibly accommodate. A tiny socket might not have enough resource to implement a custom logic optimally. On the other hand, a bulky socket likely entails significantly longer compile time. A socket requires at least one NMU for data and one NSU for control. Given that there are 28 NMUs and NSUs on the target device, our flow could theoretically support up to 28 sockets of size as small as a half-FSR (Fabric Sub Region) region. Nonetheless, to balance the compile time and the socket performance, a socket is sized such that the precompiled overlay logic could fit while leaving enough area for the custom logic (see Table I). Barring one NoC column utilized by the Shell, we could instantiate as many as 9 similar-sized sockets with this scheme, plus additional 3 smaller sockets shown in Figure 4. We could leverage these as 1x1 building blocks to construct a variety of different floorplan shapes and sizes as needed.

E. The Backend Compilation flow

Figure 5 shows our end-to-end SPADES compilation flow. Our backend flow, discussed hereafter, is fully automated starting from a Socket CL design input from the front-end stage. The final output is a bitstream of a complete SPADES implementation for configuring the dynamic region of the VCK5000 card.

1) Static logic Compilation: The static logic comprises of the Shell and the socket manager. The socket manager is in charge of dispatching the socket program binaries (generated by a host program) to all sockets present in the system at application startup over the NoC, as well as controlling and monitoring their status. The functionality of the static logic

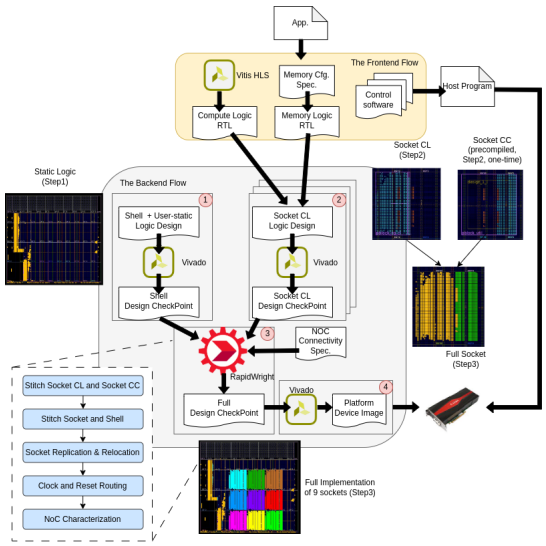

Fig. 5. The SPADES flow

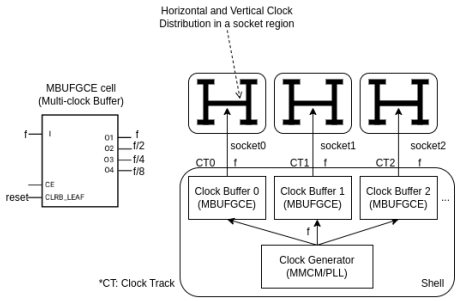

Fig. 6. Each socket owns a unique Clock track driven by a separate MBUFGCE clock buffer. Note that, although a socket uses two clock domains (f and $f/2$), there requires only a single physical clock net routed from a clock buffer. The frequency division is achieved by the buffer division cells (BUFDIV_LEAF) located in the horizontal clock distribution inside every socket region.

mainly concerns with system management. Therefore, it only requires a one-time compilation and is reusable across different applications. We use AMD Vivado to implement the static logic and generate a design checkpoint file (DCP) for the subsequent Design Assembly step.

2) Socket Logic Compilation: This step implements a socket design in an out-of-context flow by Vivado. No shell is pre-loaded ahead of the compilation to reduce IO overhead. As mentioned before, Socket CC is pre-compiled and invariant to target applications. Therefore, in this step, we only concern with generating the placed-and-routed netlist for Socket CL. For accurate timing analysis and optimization, we model the clock skew and latency constraints as if the socket were implemented in-context of a full system including the shell. The clock buffer used in this flow will later be detached and replaced with a clock buffer from the shell in the Design Assembly step. In addition, different socket designs could be implemented in parallel to reduce the compile time. This applies to sockets whose custom logic is unique, or identical socket designs with different floorplans. We generate a design checkpoint file for each socket implementation in this step.

3) Design Assembly: This step combines the design checkpoints of the static logic and the socket logic to produce a full system implementation. If a socket is reused multiple times (as in the case of data-parallel kernels), we stamp out as many copies of the socket implementations as needed and place them to the pre-determined socket floorplan slots. Our floorplan scheme, as stated above, guarantees conflict-free, hence we do not need to re-place or re-route any part of the implementation after assembling the sockets. We leverage the open-source netlist custom tool RapidWright [6] for design stitching, replication, and relocation in this step.

However, as a socket is generated from an out-of-context flow, it does not connect to a clock buffer in the shell. We eliminate the clock buffer used in the socket checkpoint, and connect the clock net from the shell's clock buffer to a clock entry node of the socket (the node that is immediately outside the socket floorplan). The internal clock routing of the socket remains intact to preserve the socket timing characteristics obtained from Step 2. But if a single clock buffer is routed to every socket, this will lead to an issue in Vivado *report_timing* as the tool complains that there are multiple clock roots, one per socket region, and the delay calculation might be inaccurate, which forces us to reroute the clock signal. To resolve this issue, we opt to employ more than one clock buffer, one for each socket. Each clock buffer drives a clock track as illustrated in Figure 6. There are 24 clock tracks available in the target Versal device. Excluding those used by the shell clocks, there remains 15 clock tracks which is sufficient to cater to the maximum number of sockets supported in our flow (9 to 12). Avoiding re-routing the clock improves the compile time. Additionally, this would help us preserve the clock tree structure of each socket as implemented in Step 2, which in turn reduce the impact on the maximum achievable frequency when adding more sockets to the system. Our tool performs track switching for each socket from the original track assigned in Step 2. In addition, we use the new multi-clock buffer cell (MBUFGCE) in Versal architecture to generate the fast and slow clock signals for our sockets since it only requires routing from a single clock buffer to a socket region, and the clock frequency is divided near socket logic cells by the primitive leaf buffer division cells, thus simplifying clock routing and reducing clock skew.

Note that there is no fabric routing at this point. The remaining step is characterizing the NoC connectivity between the sockets using RapidWright NoC API. The NoC characterization is application-specific. For instance, all sockets may need to communicate to the DDR. Socket-socket communication may be necessary if we need to synchronize their executions. Besides connectivity, we also set a bandwidth estimate for each connection. Since we are mapping data-parallel applications in this work, all paths to DDR likely require high bandwidth. Therefore, we share the DDR bandwidth equally for the participating sockets present in the system. The socket-socket

communication paths are mainly for control, thus we minimize the bandwidth requirements of these connections. The output of this step is a fully-routed design checkpoint.

4) Bitstream Generation: The full design checkpoint is imported to Vivado for NoC routing. The Vivado NoC compiler assesses the NoC constraints specified in the previous step to generate a NoC solution that meet the connectivity and bandwidth demands. We also generate the timing and area reports in this step to evaluate the QoR of the implementation. Finally, the tool emits a bitstream ready for programming the FPGA device.

F. The frontend flow

An application is manually re-structured in the frontend flow to ensure compatibility with our socket model. In particulars, we identify parts of an application for the socket custom logic implementation. This usually involves datapaths that perform computation over on-chip memory blocks. We also supply a memory unit configuration specifying the RAM blocks available to the compute unit and the DMA engines. The scheduling of custom logic execution, the data movement between off-chip and on-chip memory, and which work items to execute are written in a software C code running on each socket softcore.

III. Evaluation

A. Experimental Setup and Benchmarks

We use Vitis and Vivado 2022.2 for our experiments. The software CAD flows run on a single host machine (AMD Ryzen Threadripper 2990WX 32-Core @1.71GHz, 32GB RAM). We attach an AMD Versal VCK5000 card to one PCIe slot of the machine. Three benchmarks are selected from linear algebra (*matmul*, *cholesky*) and stencil computation domain (*jacobi-2d*). The benchmarks contain affine loop nests with 64-bit integer datatype. *cholesky* and *jacobi-2d* require synchronization between sockets to ensure correct behavior. We parallelize the outermost loop with 9 sockets (*socket_m*). Within a socket, we perform inner-loop (tiling, unrolling, pipelining, double buffering) optimizations in order to maximize the performance of a socket given the resource constraint of the floorplan. We use Vitis HLS to generate RTL for the custom logic, and port it to our backend compile flow. All sockets share the same implementation per benchmark. The baseline for comparison is the full top-down Vitis flow in which we also implement a 9-core design without any floorplaning. To ensure a fair assessment, each core has similar HLS optimizations to a socket, so their cycle counts are equivalent regarding the custom logic part. Note that the Vitis HLS-designed core does not employ our Socket CC; rather the compute, memory, scheduler, and AXI interface (256-bit) logic are generated altogether in the Vitis HLS-designed core. The target frequency for both SPADES and Vitis flow is 500MHz. We set Vivado *maxThreads* parameter to 32 in an attempt to utilize all CPU cores of our test machine per CAD run.

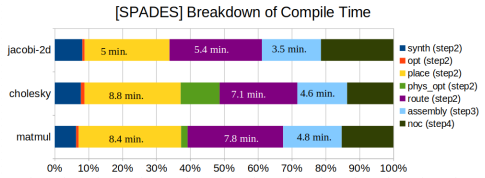

Fig. 7. A breakdown showing the task percentages of the overall SPADES compile time. Time-consuming tasks are annotated with real numbers.

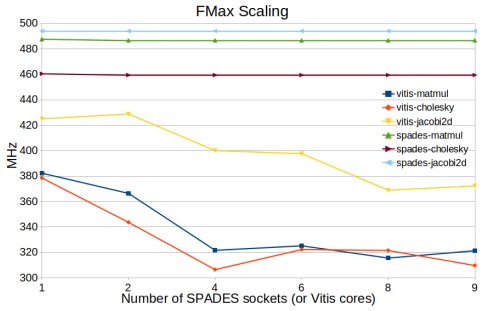

Fig. 8. Maximum Achievable Frequency scaling with respect to the number of sockets (cores) in the design

B. Compilation Time

Column 3 of Table II shows the comparison of SPADES and Vitis in compile time (from Synthesis to the final Placed-and-Routed netlist). Our flow is **7x** faster than Vitis on average and able to finish in minutes in contrasts to 2.5-3.5 hours required by Vitis. Figure 7 gives a breakdown of the time taken by each task in SPADES. The custom logic compilation (Step 2), especially placement and routing, dominates most of the time, whereas the Design Assembly step is relatively fast (3-5 min.) for our 9-socket design. We also found that precompiling Socket CC helps reduce the overall compile time by at most 1.9x (*jacobi-2d*), but might cause routing congestion issue due to limited routing resource for complex custom logic, hence worsening the route time (*matmul, cholesky*).

C. Quality-of-Result and On-board Performance

Column 4-7 of Table II compares the resource utilization of the full implementation. SPADES uses more LUTs and FFs due to the fixed Socket CC overlay logic (e.g., softcore, DMAs, etc.). Our on-chip RAM utilization is lower than Vitis because we use LUTRAMs for AXI request buffering instead of BRAMs as in Vitis HLS design. We also use less DSP resource since we circumvent the mutiplier usage when designing the address generation logic for the DMA engines. Thus, our BRAM/URAM and DSP utilization are fully dedicated to the custom logic.

The final column of Table II shows the maximum achievable frequency of both flows. Our flow is 42% better than Vitis on average in terms of FMax. Our exploit of multiple clock buffers allows us to add more sockets to a design with little or no frequency degradation, while Vitis struggles with keeping up the frequency when a design becomes increasingly complex with more cores as shown in Figure 8.

TABLE II
COMPILE TIME AND QUALITY-OF-RESULT COMPARISONS (SPADES / VITIS)

Benchmark	**Compile time (s)**	**LUT** (max. 899,840)	**FF** (max. 1,799,680)	**BRAM+URAM** (max. 967+463)	**DSP** (max. 1,968)	**Freq. (MHz)**
matmul	1659 / 12284	335,458 / 224,180	336,687 / 417,430	576+36 / 572+36	1,296 / 1,512	485 / 322
cholesky	1861 / 10512	294,805 / 205,270	339,504 / 314,948	576+72 / 716+45	918 / 1,206	458 / 310
jacobi-2d	1189 / 9465	275,221 / 223,375	295,215 / 254,919	432+0 / 608+0	648 / 756	495 / 374
Avg. improvement	7x	-38.8%	-14.2%	+14.4%	+17.5%	+42%

TABLE III
PERFORMANCE COMPARISON (SPADES / VITIS / SPADESTD12)

Benchmark	**Problem size**	**Exec. time (us)**
matmul	1024x1024	11319 / 17239 / 12092
cholesky	4096x4096	431935 / 426000 / 452278
jacobi-2d	8192x8192	117124 / 136188 / 119768

We then measure the actual benchmark performance on VCK5000. We program the card using the bitstreams generated from SPADES and Vitis flow. For each benchmark, we setup a host program to initialize and copy data from the x86 host to the FPGA DDR, invoke the PL execution, and copy the result back to the host for verification. We are only interested in comparing benchmark executions of SPADES versus Vitis implementations, not end-to-end system latency, therefore neither host-device memory transfer time nor configuration time is factored in our results. Since our approach could only scale up to 9 *socket_m*, we are also keen to study how it fares against full 12-socket implementations by the top-down SPADES flow without socket floorplaning (SPADEStd12). Table III shows that SPADES incurs virtually little to no performance loss to Vitis or SPADEStd12. We attribute these results to several factors as follows. (1) Although our cycle counts are worse than Vitis HLS cores' due to the control overhead of Socket CC, it is well-compensated by the improvement in clock frequency owing to our physical design choices and optimizations. (2) To mitigate the control overhead, we employ a task queue for asynchronous execution of the custom logic and the DMAs, thus effectively overlapping the work done by the controller (such as address generation, scalar computation, branches, loops) with the DMAs and the custom logic. (3) More sockets (or cores) do not necessarily lead to better performance, either due to lower achievable frequency or saturated memory bandwidth.

IV. RELATED WORK

On addressing the productivity issue, there has been a significant line of works on building some Overlay designs [7], [8], [9], [10], [11]. Some considers fast compilation or customization at the netlist level, as well as attends to specific architectural characteristics to achieve better frequency [12], [13]. On task partitioning for separate compilation, [14], [15], [16], [17] splits an HLS design to smaller modules for parallel compilation, and interconnect the modules by a soft NoC overlay. [18] implements a paging system for dynamic reconfiguration. Similarly, RapidStream [19], [20] compiles dataflow tasks in parallel and automatically inserts pipelined registers to improve timing on multi-SLR AMD Alveo (UltraScale+) FPGAs.

Aside from the one difference from the prior arts in the use of the novel hard NoC on Versal PL for the interconnect of the processing elements, our work primarily concentrates on building a parallel, distributed system of compute engines along with tools for speeding up the mapping of data-parallel applications to such system, thereby improving the design productivity. Nonetheless, while we only focus on mapping data-parallel benchmarks in this work, we believe that the ideas from previous works are applicable to our flow, and we could extend them to target streaming applications of heterogeneous tasks, or implement dynamic reconfiguration at the socket level in our flow.

V. CONCLUSION AND FUTURE WORK

We have shown a working end-to-end tool flow from application to placed-and-routed netlist for three data-parallel benchmarks targeting Versal VCK5000. Co-design with the target architecture, our flow exploits the application domain's modularity and reusability in tandem with hardware/software partitioning techniques to design and implement a system of parallel, distributed sockets for speeding up the compile time from hours (2.5-3.5 hrs.) to minutes (20-30 min.), all the while achieving comparable or better performance over the standard vendor flow.

The front-end application mapping and optimization is currently done manually. Hence, future work involves automation of this process to relieve the burden on users. This would also enable us to perform a design space exploration study for a more thorough comparison between the best SPADES designs and the best Vitis designs (e.g., not necessarily restricted to 9 cores or sockets of similar optimizations).

VI. ACKNOWLEDGMENTS

We would like to thank researchers from the UIUC-HACC seminar series for the constructive feedback. We are indebted to Dr. Chris Lavin (AMD) for the prompt technical assistance of the RapidWright software. We are grateful for the VCK5000 card donation from the AMD Xilinx University Program. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Program (SRC) sponsored by DARPA.

REFERENCES

[1] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[2] Xilinx DPU, https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Deep-Learning-Processor-Unit.

[3] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 65–77. [Online]. Available: https://doi.org/10.1145/3490422.3502357

[4] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 84–93. [Online]. Available: https://doi.org/10.1145/3289602.3293906

[5] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel, "Network-on-Chip Programmable Platform in VersalTM ACAP Architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 212–221. [Online]. Available: https://doi.org/10.1145/3289602.3293908

[6] C. Lavin and A. Kaviani, "RapidWright: Enabling Custom Crafted Implementations for FPGAs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 133–140.

[7] 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.

[8] F. Tombs, A. Mellat, and N. Kapre, "Mocarabe: High-Performance Time-Multiplexed Overlays for FPGAs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 115–123.

[9] C. Liu, H.-C. Ng, and H. K.-H. So, "QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay," in *2015 International Conference on Field Programmable Technology (FPT)*, 2015, pp. 56–63.

[10] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek, "Exploring Many-Core Design Templates for FPGAs and ASICs," *Int. J. Reconfig. Comput.*, vol. 2012, jan 2012. [Online]. Available: https://doi.org/10.1155/2012/439141

[11] S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Wang, L. Guo, J. Liu, M. Zhulin, R. Mani, L. Zhang, J. Cong, and T. Nowatzki, "OverGen: Improving FPGA Usability through Domain-specific Overlay Generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 35–56.

[12] 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," *SIGARCH Comput. Archit. News*, vol. 43, no. 4, p. 28–33, apr 2016. [Online]. Available: https://doi.org/10.1145/2927964.2927970

[13] L. Liu, J. Weng, and N. Kapre, "RapidRoute: Fast Assembly of Communication Structures for FPGA Overlays," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 61–64.

[14] Y. Xiao, A. Hota, D. Park, and A. DeHon, "HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 70–78.

[15] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon, "Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 153–161.

[16] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 933–945. [Online]. Available: https://doi.org/10.1145/3503222.3507740

[17] Y. Xiao, S. T. Ahmed, and A. DeHon, "Fast Linking of Separately-Compiled FPGA Blocks without a NoC," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 196–205.

[18] D. Park, Y. Xiao, and A. DeHon, "Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–10.

[19] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "RapidStream: Parallel Physical Implementation of FPGA HLS Designs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–12. [Online]. Available: https://doi.org/10.1145/3490422.3502361

[20] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 81–92. [Online]. Available: https://doi.org/10.1145/3431920.3439289