Case Study on Combining Open-Source Tool Flows for Grids of Processing Cells

Lars Luchterhandt*, Vivek Govindasamy†, Yutong Wang†, Rainer Dömer†, Wolfgang Mueller*, Christoph Scheytt*
*Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany
†Center for Embedded and Cyber-physical Systems, University of California, Irvine, CA, USA

***Abstract*—Massively parallel computer architectures based on identical microprocessor tiles are well known for their high scalability and performance. In this work, we introduce an open-source tool flow for scalable on-chip grids of RISC-V processor cells that seamlessly combines high-level SystemC modeling with the generation and simulation of hardware models at RTL down to FPGA implementation featuring the Chipyard framework. Our experimental evaluation quantifies the speed-accuracy trade-offs at different abstraction levels and compares them with their physical implementation on an FPGA.**

I. Introduction

Complex applications demand higher performance and drive the design of truly scalable parallel architectures. Similar to the classical transputer architecture [1], a Grid of Processing Cells (GPC) [2] avoids the classic shared memory bottleneck by using separate local memories.

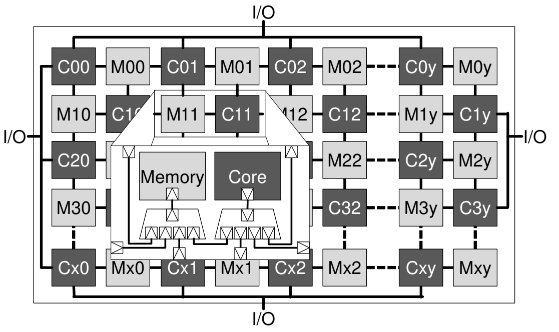

Fig. 1. Checkerboard Grid of Processing Cells (GPC) [2].

The GPC platform, as illustrated in Fig. 1, is a fully scalable grid where processor cores are paired with local on-chip memories. Each GPC cell consists of a processor Core and a local Memory, as well as local interconnects consisting of address decoders and memory arbiters, shown here with SystemC TLM-2.0 socket connectivity. Individual processor cores can only reach memory addresses in their cell and the immediate local neighborhood.

In this work, we use open-source software to generate and simulate GPC models with increasing levels of accuracy, synthesize them, and analyze their performance with clock-cycle accuracy. We generate 32-bit RISC-V Rocket processors [3] using custom Python scripts and the Chipyard framework [4] with the hardware construction language Chisel [5]. Our models are fully scalable in grid size, buffer depth, and memory size, and cell processors can be individually configured. As summarized in Table I, we cover 5 abstraction levels, namely (1) functional multi-threaded C++, (2) Transaction Level Modeling (TLM), (3) Instruction Set Simulation (ISS), (4) Register Transfer Level (RTL), and (5) Field Programmable Gate Array (FPGA). Our experimental evaluation quantifies the speed/accuracy trade-offs in build and runtime.

A. Related Work

In 1972, Flynn introduced a taxonomy for parallel computer architectures along different instruction and data streams: single/multiple instructions and single/multiple data [6]. In contrast to traditional single processor systems with single instruction single data (SISD) streams, modern architectures operate multiple instruction and/or multiple data streams in parallel, termed SIMD, MISD, or MIMD. The concepts of configurable processor grids for massively parallel processing stem from the principles of transputers, introduced in the 80s to overcome the Von-Neuman bottleneck [7]. A transputer is a single 32-bit RISC microprocessor with local memory and four serial message-passing IOs, establishing communication links to adjacent processors [8]. They were applied as building blocks of massively parallel supercomputers with up to several hundreds of nodes [1]. As such, transputers rely on the principles of a *nothing-shared* architecture with the advantage of high scalability and configurable identical cells that can be aligned to individual applications.

In prior studies, we have modeled GPCs as a nothing-shared architecture with SystemC TLM-2.0 [9], [10] and further synthesized a GPC with RISC-V processing cells down to FPGA and ASIC implementations using the Chipyard framework [3]. In this paper, we combine our previous approaches and present a seamless integration of the classical open-source SystemC design flow with RTL generation and FPGA synthesis from the open-source Chipyard framework. We present a case study demonstrating the flow from functional level C++ down to an FPGA implementation illustrated by mapping a proof-of-concept benchmark as software application onto a GPC.

II. Tool Flow

Our system design flow is a seamless integration of open-source EDA software with SystemC TLM-2.0 modeling, instruction-set simulation (ISS), and the Chipyard framework for RTL generation and FPGA synthesis. The tool flow over five abstraction levels is shown in Fig. 2.

A. Functional Specification and Modeling with SystemC

The tool flow starts with a pure functional model specified in multi-threaded C++ where tasks communicate via queues and perform the application, allowing the validation of functionality

TABLE I
TOP-DOWN DESIGN FLOW WITH GENERATED MODELS AT FIVE DIFFERENT ABSTRACTION LEVELS

Model	Abstraction Level	Accuracy	Software		Hardware	
			Language	Tool	Language	Tool
Functional	Untimed	Functions	Multi-threaded C++	g++	-	-
TLM-2.0	Loosely-timed	Transactions	C++	g++	SystemC	g++
ISS	Approximately-timed	Instructions	C++, Assembly	g++-rv32	SystemC	g++
RTL	Clock-cycle accurate	Clock cycles	C++, Assembly	g++-rv32	SystemVerilog	Verilator
FPGA	Real-time	Physical time	C++, Assembly	g++-rv32	SystemVerilog	Vivado

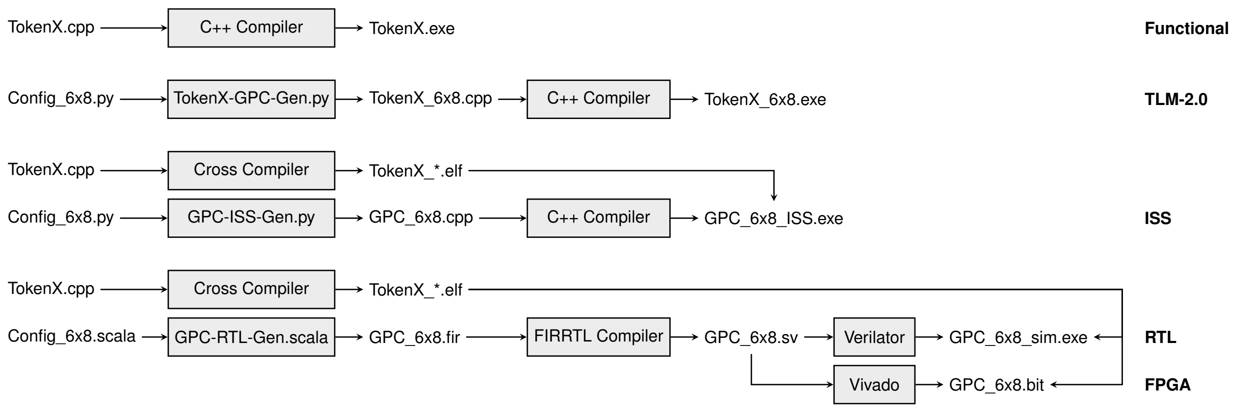

Fig. 2. Tool Flow with TokenX application on a 6×8 GPC

(Fig. 2 top). We map the tasks to a suitably sized GPC and generate a loosely timed model in SystemC TLM-2.0 with explicit communication via neighbor memories (Fig. 2 row 2).

For increased timing accuracy, we then generate an ISS (Instruction Set Simulation) model in SystemC where the application software is cross-compiled for a 32-bit RISC-V target. Using an interpreted open-source ISS [11], the RISC-V cores fetch instructions and data from their local memory and then execute the decoded instructions with the corresponding delay (Fig. 2 row 3).

B. Hardware Construction, Simulation, and Synthesis

To derive a cycle-accurate model with precise timing, we implemented a GPC RTL generator by extending the Rocket chip generator [12] of the open-source Chipyard framework [4]. At this step, we replaced the abstract instruction decoding and execution cycle with the constructed hardware of the individually configured processor and memory.

1) Hardware Construction: The GPC generator written in the hardware construction language Chisel [5] generates the desired grids and emits FIRRTL [13] based on the desired parameters for the individual GPC, e.g., grid size, memory size, or FPU support. The FIRRTL circuit compiler compiles the intermediate representation and generates a SystemVerilog model for one of the different Chipyard design flows, namely RTL simulation, accelerated FireSim simulation, FPGA prototyping, or the Hammer VLSI flow, where the remainder of this paper focuses on RTL simulation (Fig. 2 row 4) and FPGA prototyping (Fig. 2 bottom).

2) RTL Simulation: For RTL simulation of the generated GPC models, we use the open-source SystemVerilog simulator Verilator [14]. The Verilator compiles the SystemVerilog model into a multithreaded C++ model from which the simulator can be compiled. The application software for the GPC hardware model is cross-compiled for the RISC-V target and loaded into the scratchpad memories at the beginning of the simulation. RTL simulation time can be significantly decreased when we build and run many RTL simulators of different designs in parallel on a High-Performance Computing (HPC) cluster. We thus extended our flow to automate job submission for the open-source HPC workload manager Slurm [15].

3) FPGA Synthesis: To arrive at a hardware implementation with higher performance, we synthesize the RTL design for the AMD VCU108 FPGA using AMD Vivado. By extending the flow to support Slurm job submission, with our additional extensions for HPC workload management automation, many designs can be synthesized in parallel on an HPC cluster. After synthesis, the bitstream with the GPC hardware is loaded to the FPGA. Thereafter, the cross-compiled application software is loaded into the scratchpad memories via JTAG and executes on the FPGA at the given frequency. Hence, the FPGA prototype combines both, the performance from higher abstraction levels and the low-level accuracy at RTL.

III. EVALUATION

Our evaluations first introduce quantitative measures which are summarized in Table II before we outline our qualitative results on inter-cell communication.

A. Experimental Setup

For experiments on the functional, TLM-2.0, and ISS model, we use C++ and SystemC version 2.3.3 on a Linux workstation with an Intel Xeon E-2388G CPU (16 cores at 3.2 GHz). The

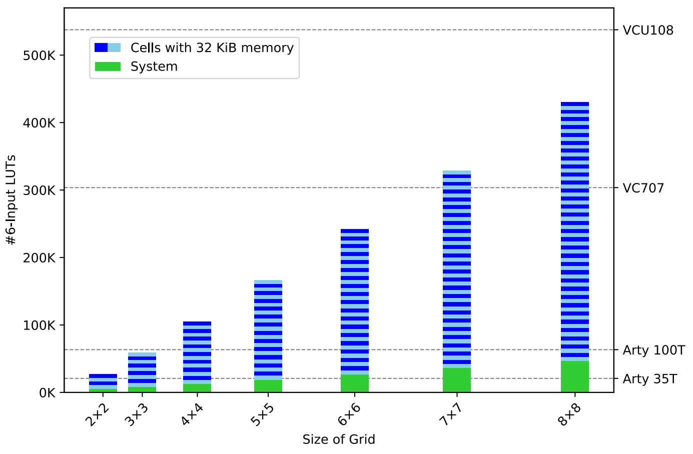

Fig. 3. #LUTs required for GPCs synthesized for VCU108 FPGA at 50 MHz

ISS model is based on the SystemC-based RISC-V VP from [11]. All experiments at RTL and on FPGA are fully automated with Chipyard 1.9.1 and our extensions in Python. We use Verilator for RTL simulation on an HPC cluster, where nodes are equipped with 256 GiB RAM and two AMD Milan 7763 CPUs (64 cores at 2.45 GHz). Larger designs require large nodes with 2 TiB RAM and AMD Milan 7713 CPUs. The FPGA experiments are conducted on an AMD VCU108 board featuring 537 K 6-input look-up tables (LUTs) with AMD Vivado for synthesis.

B. GPC Configurations

Our experiments cover GPC configurations from 2×2 to 16×16 cells with 32-bit RISC-V processors and 32 KiB of memory. Fig. 3 compares the utilization of different configurations and shows the limitation of popular FPGA boards (VCU108, VC707, Arty7 100T/35T) on the right-hand side.

C. Software Application Benchmark

For empirical evaluation of different GPC configurations, we implemented a scalable software application named TokenX. TokenX is a simple proof-of-concept benchmark for communication with low computational load. The software starts with an initial number of tokens in each cell and continuously exchanges tokens with neighbor cells in a pseudo-random fashion. It is organized in a number of iterations of computing and exchanging messages.

D. Experimental Results

To evaluate the speed-accuracy trade-off of different simulation platforms at different abstraction levels, we executed TokenX on grid sizes 6×6 and 8×8. Table II summarizes our results and quantifies the cost for conducting benchmarks on each abstraction level in build time and runtime. The simulation results include the executed number of transactions and instructions. At RTL and on FPGA, we also list the number of elapsed clock cycles based on values of the *mcycle* CSR before and after the execution of the software.

In general, the table shows an increase in build and runtime for each refinement down to RTL. Note here that the simulation results at RTL are linear projections based on 1 K iterations. It also shows that the FPGA model achieves a similar order of runtime as our initial functional implementation. Although TLM-2.0 has some overhead for the explicit modeling of memory transactions, the runtimes are within the same order of magnitude as the functional model. At ISS, we switched from host-compiled to RISC-V target-compiled software with instruction-accurate simulation. This causes an increase in runtime by two orders of magnitude.

As expected, cycle-accurate RTL simulation is the most expensive simulation in the entire design flow with respect to build time and runtime. Compared to ISS, we observe an increase in runtime by two to three orders of magnitude. This meets our expectations as we simulate fully constructed hardware models at this level.

We synthesized the RTL models for an FPGA with a frequency of 50 MHz and 100 MHz. Running at 100 MHz reduces the runtime by 50% at the cost of a doubled build time. The FPGA prototyping confirms the measured number of instructions on higher abstraction levels with less than 1% difference.

E. Grid vs. System Bus Communication

To highlight the nothing-shared advantage of the GPC, we compare its grid-based communication performance to a system-bus-based communication alternative. We generate and compare two different hardware configurations at RTL and on FPGA:
(1) instructions and data of a cell are fetched from the cell's local memory with local inter-cell access to the memories of its four surrounding neighbors (local memory) as introduced in Section I.
(2) instructions and data of a cell are fetched via a shared system bus (global memory).

For cycle-accurate performance evaluation of the two variants, we start with RTL simulation of TokenX for grid sizes from 2×2 to 16×16 as summarized in Fig. 4. The figure compares the two hardware configurations with local vs. global instruction and data access. While configuration 2 with full system bus access is clearly affected by contention, we can observe lower constant values for the GPC architecture (configuration 1). On the shared bus variant, the overhead in runtime caused by contention increases linearly with the number of cells in the design.

Fig. 5 shows corresponding results for TokenX on FPGA. The grid size is limited to 8×8 by the utilization of our FPGA. Compared with the RTL simulation, we see the results are accurate apart from small deviations caused by minor differences in the design (hardware UART) and ELF files (no syscalls on FPGA). Once there are cells with four neighbors in the grid starting from size 3×3, the GPC remains running with a constant number of cycles whereas the global memory variant runs into linear bus contention. With an increasing number of cells, more bus traffic is caused, which further increases contention. Across all grid sizes, the GPC achieves the lowest runtime compared to the global memory variant. This demonstrates the efficiency of the nothing-shared GPC architecture.

TABLE II
EXPERIMENTAL RESULTS ON 6×6 AND 8×8 GPC TARGETS RUNNING TOKENX WITH 1M ITERATIONS

Platform	Model	6×6 GPC			8×8 GPC		
		Build Time	Runtime	Simulation	Build Time	Runtime	Simulation
Workstation[a]	Functional	0.33s	12.41s	-	0.33s	21.71s	-
	TLM-2.0	3.43s	83.30s	240M transactions	4.19s	171.59s	448M transactions
	ISS	4.11s	1:31h	275M instructions	5.99s	3:09h	275M instructions
HPC[b]	RTL	15:00m	129 days*	272M instructions* 782M clock cycles*	39:21m	222 days*	274M instructions* 787M clock cycles*
HPC[b,1], FPGA[c]	FPGA	56:09m	15.45s	271.2M instructions 772.5M clock cycles	1:49h	15.46s	271.3M instructions 772.9M clock cycles

[a] Intel Xeon E-2388G, 16 cores, 3.2 GHz [b] 2x AMD Milan 7763, 64 cores, 2.45 GHz [c] AMD VCU108 FPGA, 50 MHz
[1] FPGA synthesis * Linear projection based on 1K iterations

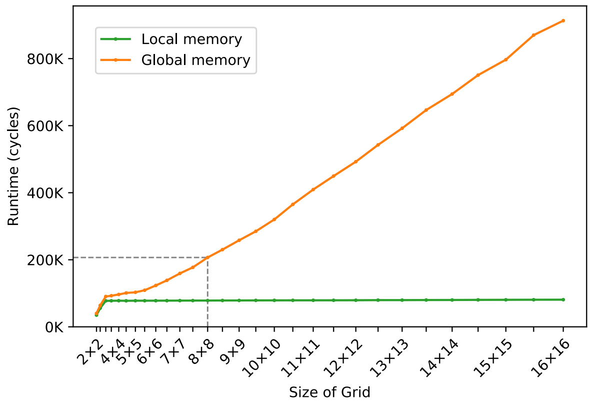

Fig. 4. TokenX runtime (100 iterations) of RTL simulation

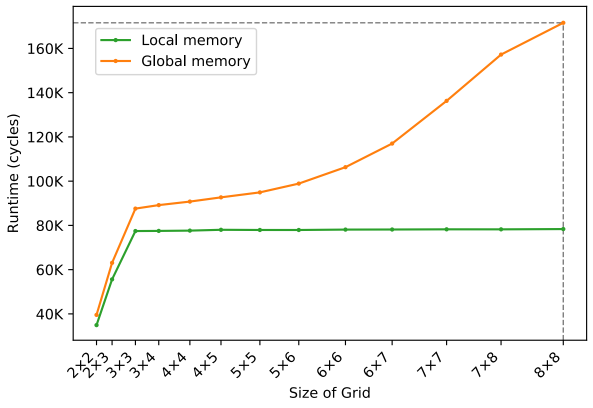

Fig. 5. TokenX runtime (100 iterations) on FPGA

IV. CONCLUSION

We introduced an open-source top-down tool flow for highly scalable Grids of Processing Cells (GPC) based on configurable 32-bit RISC-V cells with local memory. Our evaluation quantifies the accuracy and execution speeds at different abstraction levels over different simulation platforms and demonstrates the advantage of nothing-shared grid architectures. Our flow is highly automated and seamlessly combines SystemC with the open-source Chipyard framework [4]. We intend to make our models and scripts accessible by placing them in open repositories once we have completed our other experiments with FireSim and FireAxe. We only applied open-source tools with the exception of AMD Vivado which has been identified as the major obstacle for a fully fledged open-source tool flow.

Acknowledgements

The work described here is partially funded by the German Bundesministerium für Bildung und Forschung (BMBF) through the Scale4Edge project (16ME0133). We gratefully acknowledge the computing time and resources provided on the Noctua 2 HPC Cluster at the NHR Center PC2.

REFERENCES

[1] A. J. G. Hey, “Supercomputing with transputers—past, present and future,” in *Proceedings of the 4th International Conference on Supercomputing*, ser. ICS ’90. New York, NY, USA: Association for Computing Machinery, 1990, p. 479–489.
[2] R. Dömer, “A Grid of Processing Cells (GPC) with Local Memories,” Center for Embedded and Cyber-physical Systems, UCI, Tech. Rep. CECS-TR-22-01, Apr. 2022.
[3] L. Luchterhandt, T. Nellius, R. Beck, R. Dömer, P. Kneuper, W. Mueller, and B. Sadiye, “Implementation of different communication structures for a rocket chip based RISC-V grid of processing cells,” in *MBMV 2024; 27. Workshop*, 2024, pp. 79–89.
[4] A. Amid *et al.*, “Chipyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.
[5] J. Bachrach *et al.*, “Chisel: Constructing hardware in a scala embedded language,” in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.
[6] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.
[7] J. von Neumann, “First draft of a report on the EDVAC,” University of Pennsylvania, Tech. Rep., Jun. 1945.
[8] R. Dettmer, “Occam and the transputer,” *Electronics and Power*, vol. 31, no. 4, pp. 283–287, 1985.
[9] Y. Wang, A. Daroui, and R. Dömer, “Demonstrating Scalability of the Checkerboard GPC with SystemC TLM-2.0,” in *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.
[10] V. Govindasamy and R. Dömer, “Mixed-level modeling and evaluation of a cache-less grid of processing cells,” *ACM Transactions on Embedded Computing Systems*, Dec. 2024.
[11] V. Herdt, D. Große, and R. Drechsler, “Fast and accurate performance evaluation for risc-v using virtual prototypes,” in *2020 Design, Automation and Test in Europe Conference (DATE)*, 2020, pp. 618–621.
[12] K. Asanović *et al.*, “The rocket chip generator,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.
[13] A. Izraelevitz *et al.*, “Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 209–216.
[14] W. Snyder, P. Wasson, D. Galbi *et al.*, *Verilator*. [Online]. Available: https://verilator.org
[15] D. G. Feitelson, “Job scheduling strategies for parallel processing proceedings,” ser. Lecture notes in computer science 1459. Berlin: Springer, 1998.