Designing Application-Specific Heterogeneous Architectures from Performance Models

Thanh Cong, François Charot

Designing Application-Specific Heterogeneous Architectures from Performance Models

Thanh Cong, François Charot
Univ Rennes, INRIA, CNRS, IRISA
Email: minh-thanh.cong@irisa.fr, francois.charot@inria.fr

***Abstract*—In this paper, we propose an approach for designing application-specific heterogeneous systems based on performance models through combining accelerator and processor core models. An application-specific program is profiled by the dynamic execution trace and is used to construct a data flow model of the accelerator. Modeling of the processor is partitioned into an instruction set architecture (ISA) execution and a micro-architecture specific timing model. These models are implemented on FPGAs to take advantage of their parallelism and speed up the simulation when architecture complexity increases. This approach aims to ease the design of multi-core multi-accelerator architecture, consequently contributes to explore the design space by automating the design steps. A case study is conducted to confirm that presented design flow can model the accelerator starting from an algorithm, validate its integration in a simulation framework, allowing precise performance to be estimated. We also assess the performance of our RISC-V single-core and RISC-V-based heterogeneous architecture models.**

***Index Terms*—Heterogeneous Architecture Design, Performance Modeling, Accelerator, Simulation, FPGA, RISC-V**

I. Introduction

Nowadays, Artificial Intelligence is among the most emerging technologies which demand more computation power from hardware. Several highly demanding applications currently include these algorithms, such as self-driving vehicles, 5G communication, video monitoring and analytics. In an effort to support these applications on vehicles and other mobile devices, scientists and researchers focus on embedded high-performance computing architectures to solve these problems effectively and quickly.

To address these significant computational demands, some companies have already proposed solutions based on processor core duplication and pushed up the number of cores on a chip as high as a thousand [1]. However, the number of cores of these multi/many-core chips, which can actively switch at full speed within the chips power budget will decrease (utilization wall). The remaining silicon which is left unpowered (referred to as dark silicon) will grow exponentially with each new technology generation [2]. To address this, heterogeneous architectures combining processor cores and hardware accelerators [3], [4] have been proposed by designing heterogeneous systems that trade dark general purpose cores for a collection of specialized but transiently powered accelerators. Architectures with customized accelerators give orders-of-magnitude increased performance and energy efficiency if compared to performing the same task on a general-purpose CPU.

When considering designing multi-core or many-core architectures, the number of possible design combinations leads to a huge design space, with subtle trade-offs and design interactions. To reason about what design is best for a given target application requires detailed simulation of many different possible solutions. Many of the simulators used commercially and academically for cycle accurate or approximate simulation are designed around single threaded discrete event simulation, with SystemC, Simics and Gem5 based simulation [5], Gem5 [6] being one of the most popular tools. Although software-based simulation has gone a long way in validating computer architecture research, its ability to quickly evaluate points in design space may be deteriorating. Unfortunately, software-based simulators are notoriously challenging to parallelize, that is to say, to take advantage of multi-core host machines, and as a result, software-based simulator performance continues to decline.

To tackle this simulation gap and to perform cycle accurate simulation fast, perhaps the only way is to use dedicated hardware to accelerate the most complicated parts of the target architecture model. Several research groups have been exploring the use of hardware to build various forms of hardware-accelerated architecture simulators. There are successful projects which use Field Programmable Gate Arrays (FPGAs) to implement all or part of the simulation (usually micro-architecture and interconnect models at least), and industry makes extensive use of large FPGA-based modeling and prototyping systems [7]. This research has led to FPGA Accelerated Model Execution (FAME) techniques, where the desired target architecture is mapped to an FPGA for evaluation [8]. Another example of specialized hardware is the use of Graphics Processing Units [9]. Although FPGA can help to improve the performance of simulators, the process of designing a performance model targeting an FPGA is more complicated than designing a simulator in software. Moreover, once the FPGA has been designed, it also needs to be debugged. The risk is therefore that the idea of deployment of performance models on FPGA will fail not because of lack of performance but because of increased model development time.

In this paper, we present a design approach for application-specific heterogeneous architectures based on the usage of performance models of hardware components (processor cores, domain-specific accelerators, memories, interconnects). The simulator of a particular heterogeneous architecture can be

built from these models and deployed on an FPGA-based system thus helping to speed up the simulation. It is a question of taking into account data movement between hardware components and coherency management for accelerators during the simulation. This aspect is often neglected and its impact has been well shown by Shao et al. in [10].

We focus in particular on the way the accelerator models are designed starting from the C code of the algorithm and show how there are integrated with the other components of the heterogeneous architecture. The data flow graph of the algorithm is used as a representation of the accelerator without having to generate HDL (Hardware Description Language) code. From this representation, the scheduling of the graph is realized, and the control for a generic template of an accelerator model is produced.

The proposed simulation infrastructure is based on the use of two simulation tools: Aladdin [11] and HAsim [12]. The Aladdin accelerator simulator provides a framework modeling power, performance, and cycle-level activity of standalone, fixed-function accelerators without needing to generate RTL. The HAsim FPGA-based simulator builds processor timing models and memory system, including support for cache coherence protocols and interconnects models. The contributions presented in this work are as follows.

- A methodology for generating performance models of accelerators targeting FPGA by exploiting a part of Aladdin flow is proposed. The application trace is generated in a compact graph scheduled trace (SGT) format. It can be directly interpreted by the timing accelerator model on the FPGA to re-create the original computation and memory behaviors of the application.
- A cycle accurate and cost-effective simulation framework is presented. It simulates the whole system of the accelerator-processor architecture, including RISC-V core, dedicated accelerators, coherent cache/scratchpad with shared memory, and routers. The framework is targeting current CPU-FPGA platforms with the goal of speeding up the simulation.

The remainder of the paper is organized as follows. Section II describes the design flow and the different steps of the proposed design approach. Section III explains the building of processor models within the HAsim framework. A performance model of a RISC-V-based processor is proposed. Section IV addresses a case study. Experiments for a subset of the MachSuite benchmark suite are presented and their results are discussed. Finally, related works are discussed in Section V.

II. Design Approach

Our approach for designing application-specific heterogeneous architectures is based on a trace-based accelerator simulator that profiles the dynamic execution of a program through the integration of hardware accelerators coupled to a single core processor. We construct a dynamic data dependence graph (DDDG) as a data flow representation of an accelerator. Fig. 1

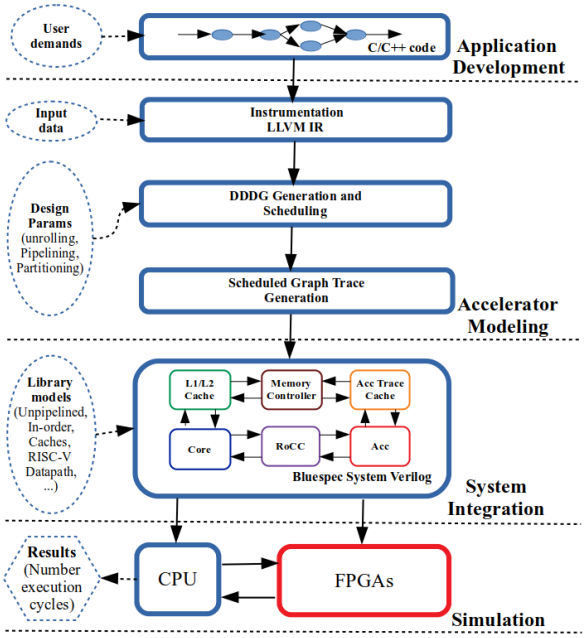


Fig. 1. Illustration of the generic design flow.

shows the generic flow of the proposed design method. It consists of the following steps: application development in C/C++ programming language (II-A); modeling of the accelerator to generate performance models of dedicated hardware blocks (II-B); system integration where the hardware accelerator is integrated with processor models (II-C); and simulation where all models are simulated and evaluated on the FPGA-based simulator (II-D).

A. Application Development

The design method aims at building an efficient architecture for a domain-specific application by integrating task-specific custom hardware to a single core. It takes as input a high-level specification (C/C++) of an algorithm without any modification and generates its execution trace. According to optimization pragmas provided by the designer (loop unrolling, loop pipelining and array partitioning), a sub-trace is extracted and the corresponding DDDG is generated. After applying a number of optimizations on the DDDG, the tool schedules the nodes of the graph with resource constraints to obtain an early performance estimate of the accelerator.

Hardware architects can exploit the obtained results to improve the application. Additionally, results can be used to adjust the application to make it more suitable for use in architecture development, especially in cases where the acceleration of the application is desired, but the implementation takes too much time.

B. Accelerator Modeling

a) Instrumentation: Our accelerator modeling methodology is based on the Aladdin simulator to which modifications have been brought [11]. To define the phases of the accelerator modeling, we start from a C description of an

algorithm before passing through an instrumentation phase where an execution trace of the application is generated. In this process, the Low-Level Virtual Machine (LLVM) [13] is leveraged for instrumentation and trace collection. The core of LLVM is a Static Single Assignment (SSA) based Intermediate Representation (IR). The IR is machine-independent and uses unlimited virtual registers. The Execution Engine, an LLVM Just-in-Time (JIT) compiler integrated into the simulator, is invoked to perform execution of the instrumented IR and generates the run-time trace. The generated trace contains runtime instances of static instructions, including instruction IDs, opcodes, operands, virtual register IDs, memory addresses (for load/store instructions) and basic block IDs.

b) DDDG generation and scheduling: After the instrumentation process, a DDDG is initiated as a directed and acyclic graph, where nodes represent dynamic instances of LLVM IR instructions, while edges denote dependencies between nodes including register/memory-dependence. Edges only consider true dependencies and do not include output dependencies. These are dynamic traces, so control dependencies are not concerned.

Before scheduling, the DDDG is optimized by performing tree-height reduction to decrease the height of long expression chains and exposing potential parallelism similar to Shao's work [14]. Focusing only on actual computations, we remove supporting instructions and dependencies between loop index variables that are not relevant to accelerators. They are assigned zero latency to the associated nodes. Removing redundant load/store operations are also considered as an efficient way to save memory bandwidth. For instance, two load operations can be reduced to one load node if they have the same memory address, so there is no store operation in between with the same address. The load can be eliminated by adding edges between the store and successors of the load if a store is a direct predecessor of a load with the same memory address. After removing redundancies, we map the memory address of load/store operations to a memory bank according to array partitioning factor.

The graph is then scheduled for execution through a breadth-first traversal, while considering for user-defined hardware parameters: loop unrolling, loop pipelining, and memory ports.

c) Scheduled graph trace (SGT) generation: The scheduled graph is then translated into a compact representation format suited to its interpretation by the hardware module corresponding to the accelerator datapath model and implemented in the FPGA. The basic strategy is to reuse scheduling of accelerator simulator and simulated communication strategies between CPU and accelerator.

The graph is translated into three instruction categories: ALU instructions (i.e., register to register operations), load/store instructions with their data addresses, other instructions that do not constitute loops. The format of SGT is illustrated in Fig. 3e. Fig. 2 illustrates how the verilog template executes the SGT format.

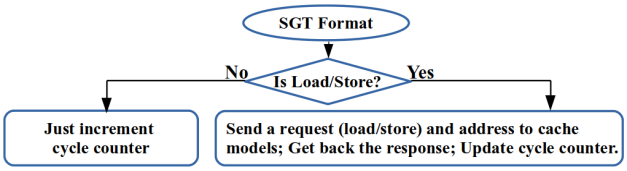


Fig. 2. Actions taken by the accelerator datapath for SGT format.

d) Flow explanation by an example: Fig. 3 illustrates by the way of an example the different phase of the flow. The chosen algorithm performs the addition of two vectors a and b. Let us assume that the add unit, memory load, and store operations have all 1-cycle latency. We suppose the designer chose to apply a loop unrolling with a factor of 2, without loop pipelining and a cyclic partitioning of array c with a factor of 2. Starting from the C program, IR trace is extracted (Fig. 3b) and the DDDG is generated (Fig. 3c).

After optimizations applied on the DDDG, the optimized DDDG, shown in Fig. 3d, reflects the dataflow nature of the accelerator. Edges in the DDDG represent flow dependencies, and DDDG node latencies are added to edges as edge weights. Fig. 3d shows the schedule generated by the breadth-first traversal algorithm. As each memory partition has two read ports, memory load operations for vector a and b can be executed in the same cycle. As vector c has a cyclic partitioning with a factor of 2, two memory stores accessing different memory banks can be executed in the same cycle. Fig. 3d represents a feasible schedule in which the loop iteration takes eight cycles.

The scheduled graph trace is then generated, as shown in Fig. 3e. The basic strategy is to remove all possible redundancies from the program execution trace while preserving fidelity. The format of this trace consists of two parts: *Opcode* and *isNewCycle* bit. The opcode comes from the LLVM individual instructions. The instructions are classified into three categories: ALU instructions (i.e., register to register operations); Load/Store instructions; control flow instructions (i.e., branch, jump, and loops). A 32-bit memory address is added to a Load/Store instruction. The SGT format preserves the execution order (scheduling) of the original program without keeping any instruction addresses. When the *isNewCycle* bit is set, the interpreter increases the execution cycle of the accelerator. The interface between accelerator and processor relies on the Rocket Custom Coprocessor Interface, as defined in [15].

C. System Integration

The representation trace of the accelerator generated in the previous step should to be interpreted on the FPGA and integrated to the processor core. Hardware modules especially designed in Bluespec System Verilog (BSV) [16], a high-level functional hardware description programming language, are the building-blocks of the accelerators performance model. As illustrated in Fig. 4 each accelerator model is composed of the

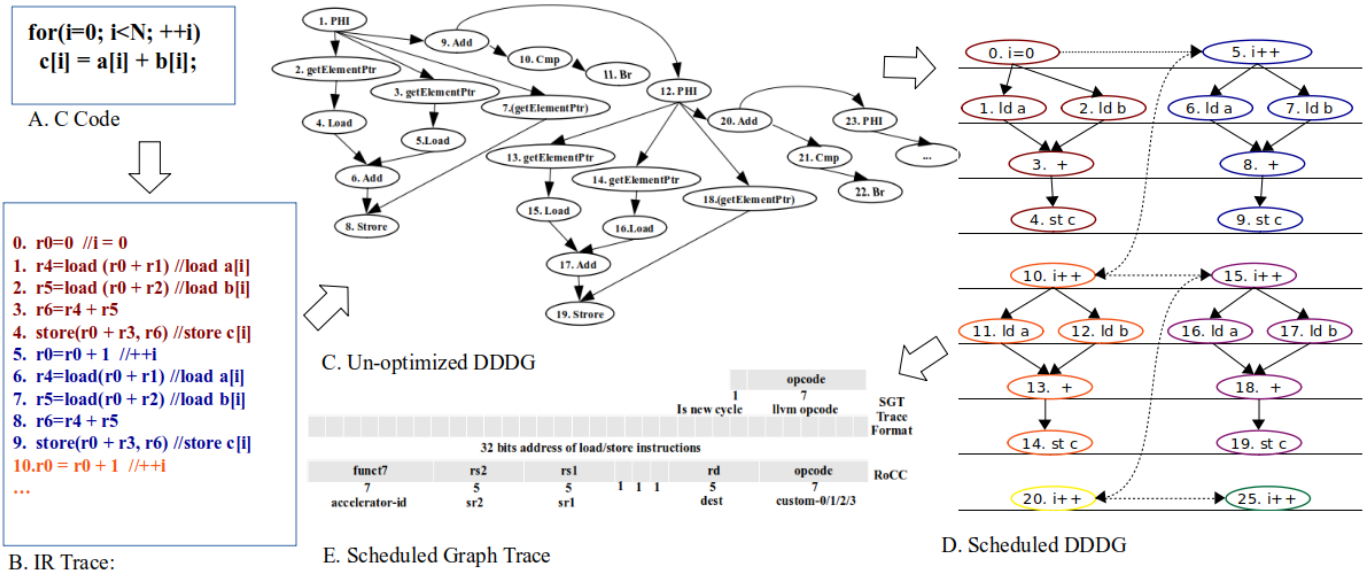


Fig. 3. An example of an accelerator model with a factor of 2 loop iteration parallelism, partitioning factor 2, and without loop pipelining.

accelerator datapath with its trace cache and a router. The trace is loaded into cache and is interpreted by the datapath.

To be able to invoke the accelerator from the processor, we added custom instructions to the RISC-V instruction set. The accelerator interfacing is then done by the inlining of an assembly function in the C code of the program running on the core processor. This function is a mix of C code and RISC-V Assembly macros that allows the generation of custom RISC-V instructions for communicating with the accelerator. The instruction is considered as Rocket Custom Coprocessor (RoCC) instruction, as shown in Fig. 3E.

D. Simulation

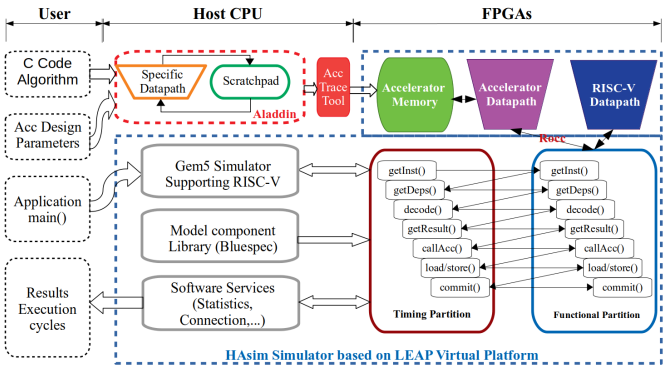


Fig. 4. Structure of the proposed simulation platform.

Fig. 4 presents the structure of the proposed HAsim-based simulation platform. The application executables, algorithms, and structural design parameters are entered by users. The Host CPU includes a software layer between the user and the FPGA-based simulator. It comprises a tool suite that includes Aladdin, gem5, the trace generation tool, and other software services.

The simulator uses the Aladdin flow to generate and schedule a DDDG graph. It then creates a compact STG format using the accelerator trace tool. The predefined accelerator datapath module (written in BSV), interfaces with processor simulation modules (functional partition, timing partition, memory system, and router). The Bluespec System Compiler is used to synthesize the simulation models and to generate the FPGA bitstream. The simulator finally configures the FPGA and loads it with the STG trace and RISC-V binary.

Then, the simulator returns the results back to the host CPU and presents it to the user. Additionally, to achieve the flexibility and to reduce the development effort when designers require to change the simulation architecture, a library of pre-defined modeling components allows architects to adapt pre-existing modules to their experiments such as branch predictors, caches, processor models, network models, etc. More details about the simulation framework is given in III-A.

III. Building RISC-V Models within the HAsim Framework

We chose to build processor models based on the RISC-V instruction set [17], originally developed in the Computer Science Division department at the University of California, Berkeley. Two processor models have been built and validated: an unpipelined and an in-order pipelined processor model. In this section, we first describe the foundations of the HAsim [12] framework, then explain the base of the functional and timing partitions of a HAsim model which is implemented in Bluespec SystemVerilog.

A. HAsim Framework Overview

HAsim (*Hardware-based Architecture Simulator*) is designed as a framework for constructing efficient simulators out of a library of reusable components, rather than emphasizing any particular target processor. The key idea is to reduce

development effort by reducing the amount of code that the architects must change in order to construct their design space exploration.

As shown in Fig. 4, HAsim is a hybrid model consisting of code running on a general purpose host CPU as well as on an FPGA. This scheme leverages the strength of each physical platform: the FPGA for fine-grained parallelism, and the CPU for rare-but difficult to implement events, such as system calls. The framework is divided into four major components: the functional partition is responsible for correct ISA level execution of the instruction stream; the timing partition (or timing model) is responsible for tracking micro-architectural specific timings (such as branch predictors and cache misses); the library of predefined modeling components (such as branch predictors and caches); and the unmodel component refers to all functionality not directly related to simulation (including the ability to track statistics and parameters, as well as the virtual platform necessary to interact with the host CPU).

In order to make the programming of HAsim models easier, the LEAP platform was developed [18]. LEAP (Latency-insensitive Environment for Application Programming) offers the support for the development of FPGA-based applications. It is similar to an operating system, but with stronger compilation support. LEAP has a standardized set of interfaces for the FPGA to talk to the outside world that helps reducing development efforts. FPGA developers can focus on implementing core functionality without spending time and energy debugging low-level device drivers because of a set of virtualized device abstractions of LEAP virtual platform is provided.

B. The Functional Partition

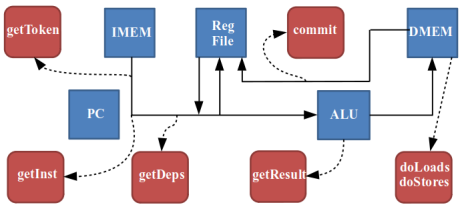
The role of the functional partition is to calculate the new state of the processor after executing an instruction. It is managed with eight operations, as shown in Table I, corresponding to traditional microprocessor pipeline stages. The timing partition invokes operations and determines the state of the machine in the order. The functional partition provides several operations which are used by the timing model to produce a cycle accurate simulation. The same functional partition can be reused across different timing models to simulate different micro-architectures. For a single instruction, the operations are typically invoked in the order. This corresponds to instructions flowing through pipeline stages in a real computer: the instruction is fetched *(getInstruction)* before it is decoded *(getDependencies)*, which takes place before register read *(getOperands)* and so on. The order in which the timing model invokes these operations on separate instructions determines the state of the machine.

Most of the components needed for the specification of the functional partition come from existing HAsim modeling projects from MIT; they can be easily instantiated by way of the AWB tool [19]. In order to design RISC-V models, only information related to instruction decoding (number of sources, destinations, and barrier information), and to the hardware datapath for executing common instructions have to be specified. This is the advantage of the ISA-independent datapath feature of HAsim.

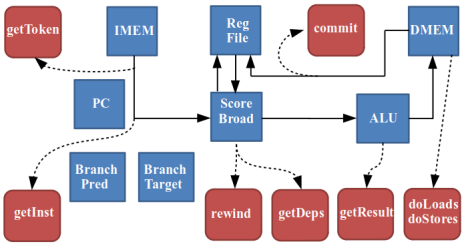
TABLE I
FUNCTIONAL PARTITION OPERATIONS

Operation	Behavior
getInstruction	Get memory address, return corresponding instruction.
getDependencies	Allocate the destination physical register, look up physical registers containing the operands.
getOperands	Read the physical register file and the instruction, return opcodes and immediate operands.
getResult	Execute the instruction, return the result including branch information, effective address.
doLoads	Read the value from memory, write to register.
doStores	Read the register file, write the value to memory.
commit	Commit the instruction's changes, remove instruction.

C. The Timing Model Specification



A. The unpipelined processor model.



B. The in-order processor model.

Fig. 5. Target processor and their modeling.

In this paper, two target architectures are considered and are defined to be used with this functional partition: an unpipelined, and an in-order pipeline processor. Each model does the same fundamental amount of work; the only change is related to time modeling. The goal is here to demonstrate that the same functional partition can be reused across different timing models to simulate different micro-architectures. The timing model does not need to implement all structures, as the functional partition handles some of their functionality.

Fig. 5A, shows a basic processor timing model that is to say an unpipelined processor that executes every instruction in a single clock cycle. The implementation executes each functional partition operation before incrementing model time. Operations of the timing model are getToken(), getInst(), getDeps(), getResult(), doLoads()/doStores() and commit(). The ALU, IMEM, and DMEM are simulated entirely via functional partition operations.

Fig. 5B shows an in-order processor target. The branch predictor structure is implemented entirely in the timing model, as it controls which address the timing model will pass to getInstruction(). On mispredicts, a rewind() is issued to represent a pipeline flush and simulated no operations are passed through the pipeline.

IV. Evaluation Results

A. Experiment Setup

In this section, we present a case study of the system level design and evaluation for heterogeneous SoC architectures using the FPGA-based simulator to demonstrate the utility and power of our design method. First, we illustrate how to design a hardware specific accelerator for the Blocked General Matrix Multiply algorithm [20]. Our result is compared to the one obtained with the Vivado High-Level Synthesis (HLS) tool. Xilinx Vivado HLS version 2018.3 is used and the accelerator clock frequency is set to 100MHz. Furthermore, we expand this case study (in simulation only) to seven benchmarks of the MachSuite benchmarks suite [21] and compare the performance of the stand-alone processor to different heterogeneous architectures.

The different architectural models developed with HAsim have been synthesized targeting the integrated BDW+FPGA platform from Intel (Broadwell Xeon CPUs (E5-2600v4) with an integrated in-package Arria 10GX1150 FPGA (10AX115U3F45E2SGE3).

B. Case study: Blocked Matrix Multiply Accelerator

General Matrix Multiply (GEMM) is a common algorithm in linear algebra, machine learning, statistics, and many other domains. It provides a trade-off between space and time, as there are many ways to partition the computation. Matrix multiply is more commonly computed using a blocked loop structure. Commuting the arithmetic to reuse all of the elements in one block before moving onto the next dramatically improves memory locality. Our application design uses a fixed blocking factor of 4 and is based on the algorithm proposed in [20]. Fig. 6 illustrates the core computation of blocked GEMM.

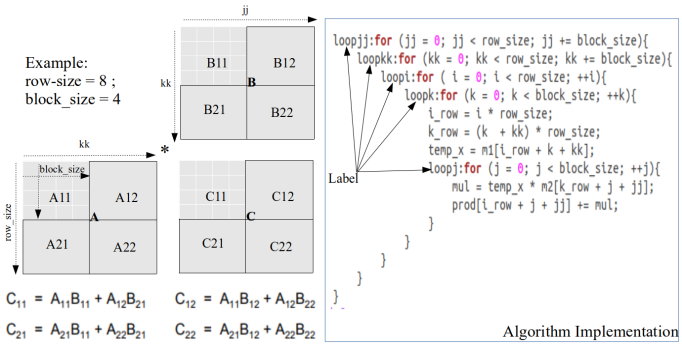


Fig. 6. Designing accelerator for blocked GEMM.

Starting from the algorithm, the accelerator model is automatically generated, its simulation is then performed with HAsim. Vivado HLS is used as a reference tool for evaluation. A C code application program is also developed which includes the following steps: loading input data, invoking accelerator, waiting for accelerator response, and storing results. The main program is compiled to RISC-V binary and loaded into the processor model.

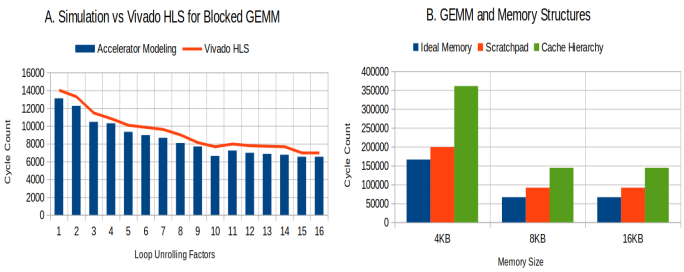


Fig. 7. Blocked GEMM evaluation.

As shown in Fig. 7A, the cycle count for blocked GEMM application while considering loop unrolling, pipelining and array partitioning pragmas is predicted. The x-axis denotes the loop unrolling factor ranging from 1 to 16. For each configuration, we set the loop unrolling factor, pipelining all loops in the algorithm implementation (the array partition is not considered in this experiment). According to this evaluation, we can see the number of cycles decrease according to the evolution of the loop unrolling factor. In addition, the number of cycles predicted by the accelerator simulator corresponds very closely to that provided by Vivado HLS.

Fig. 7B compares the performance of an accelerator for the blocked GEMM benchmark according to the usage of different memory structures: one cycle memory latency (ideal memory), scratchpad memory having a fixed latency for each request, and cache model. In order to decompose the execution time, we performed simulations with the heterogeneous simulation model (processor, accelerator, cache, scratchpad, and router). This highlights that memory access time takes up a significant portion of the execution time as the complexity of memory increases. Moreover, the number of cycles decreases as the memory size grows from 4 KB to 8 KB. This reduction is explained by the fact that a 4 KB memory size is too small to store the blocked data size (a 16*16 matrix). This evaluation shows that it is essential to study in detail the effect of memory behavior on the performance of the accelerator.

C. Machsuite Benchmarks

We considered seven benchmarks from different domains and studied their implementation with our FPGA-based simulator. Benchmarks are chosen from Machsuite [21], with some modifications to fit the gcc RISC-V compiler. Fig. 8 shows the estimated performance of these algorithms for different instantiations of the processor-accelerator pair. For all the applications, configurations for loop unrolling factors, pipelining options, and array partitioning factors/types are identical.

To assess the performance of a system combining a processor core and an accelerator models, we considered that the core executes a program that make a call to the accelerator that implements each of these benchmarks.

It is not surprising to see that for every application the in-order pipelined with cache is the slowest architecture because it simulates the complex behavior of the architecture. We also see improved performance of heterogeneous models over processor-only models. The key motivation behind the proposed design method is to rapidly evaluate the processor/accelerator performance in an architecture exploration context.

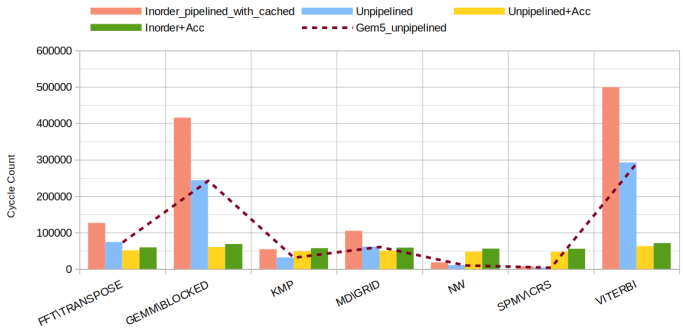


Fig. 8. Performance validation under different architectures.

Finally, we evaluated the simulator accuracy of our framework. As shown in Fig. 8, the line shows the simulation results for the baseline gem5 simulator (unpipelined model), which is measured in number of simulated execution cycles. Experiments show that our simulator has the same level of precision as the gem5 simulator.

V. Related Work

The work presented in this paper lies at the interface of several areas: performance models, FPGA-based simulators, accelerator-rich architectures, design of domain-specific heterogeneous architectures. We first present several performance modeling approaches of multi-core processors on FPGAs. We then discuss on accelerator-rich architectural simulation. The last part of the section focuses on processor design and custom hardware generation from high-level languages which share various similarities with our approach.

A. Performance Models on FPGAs

Most of the different approaches that use FPGA as an execution platform to accelerate high detail performance models have emerged in the context of the RAMP Project [22], [23]. Indeed, the FPGA's highly parallel, programmable execution substrate is suitable for the requirements of multi-core simulation. It can provide multiple orders of magnitude speedup over pure software simulators for detailed models, as shown in different papers. Efforts to explore the FPGA-accelerated simulation gave rise to several simulators: FAST, ProtoFlex, RAMP Gold and HAsim. These simulators all employ the functional and timing models partitioning, but they differ in the way these two partitions are mapped to the execution platform (composed of the FPGA board and the host computer). This leads to different architectures of simulators as classified in [24] : time-directed (RAMP-Gold, HAsim), functional-only (ProtoFlex), and speculative functional-first (FAST). It is important to note that modeling techniques do not necessarily run the target hardware RTL description on the FPGA directly; instead, the FPGA executes a model of the target system. Simulation may take multiple FPGA clock cycles to execute a single target clock cycle. Separating the FPGA cycle from the model cycle allows much higher scaling of features that can fit within a single FPGA. This feature is exploited in our simulator which can be seen as an extension of HAsim to future heterogeneous multi-core architectures.

A trace-driven simulation framework for multi-processors using FPGA is presented in [25]. The simulator's input is a specially developed compact execution trace (CET) of the application. The application trace is generated in an architecture-agnostic executable format that can be directly interpreted by the timing model on the FPGA to re-create the original application's execution events. Our modeling of accelerators exploits a comparable technique.

B. Accelerator-centric architectures

Simulation platforms for accelerator-centric architectures are studied in two recent projects gem5-Aladdin [10] and PARADE [26]. They provide software accelerator simulators enabling the exploration of many accelerator designs, but there is a trade-off between accuracy and speed. With gem5-Aladdin, users can study the complex behaviors and interactions between general-purpose CPU and hardware accelerator. PARADE is similar to gem5-Aladdin in that it enables simulation of accelerator workloads within a SoC framework. However, it only models the traditional DMA-based accelerators where all data must be copied to local scratchpads before starting the computation. These two projects share the same limitation inherent to software simulators when complexity of the simulation model increases.

C. Design and generation of custom hardware

Suleyman et al. [27] present a generic methodology for designing domain-specific heterogeneous many-core architectures based on accelerators integrated into simple cores, which is comparable to ours. They reveal the steps undertaken to integrate the accelerators into an open source core and use them via custom instructions. They automate custom hardware generation to facilitate design space exploration of heterogeneous architectures. Our method is faster, simpler and requires fewer development efforts thanks to the architecture models which avoid prototyping on the FPGA.

VI. Conclusion and Future Work

To overcome the power and utilization wall in today's architectures, computer architects replace some processors by accelerators that can achieve orders-of-magnitude performance and energy gains. In this paper, we introduced an approach

for designing application-specific heterogeneous architectures based on performance models through integrating an accelerator to a RISC-V core model. Based on the Aladdin simulator, we developed a tool to automatically generate the dataflow graph and the scheduled trace of an accelerator from C code applications. We integrated accelerator models into a core that executes the custom RISC-V instruction set. By using the HAsim framework, we modeled unpipelined and in-order-pipelined RISC-V processor interfacing accelerator and memory system. To evaluate our approach, we implemented a Blocked GEMM case study. It is shown that accelerator can be produced from the application program without investing any extra effort into developing the hardware. Additionally, several architectures were explored by running MachSuite benchmarks with different processor and accelerator models.

In the future, we plan to define a heterogeneous multi-core architecture by integrating multi-processor, multi-accelerator, shared memory systems with a network-on-chip. Moreover, we will also focus on the memory hierarchy and model the memory access patterns for accelerators.

ACKNOWLEDGMENTS

The authors would like to thank Bluespec for providing us the Bluespec tools and also Intel Labs for giving us access to a cluster of the integrated BDW/FPGAs, within IL's vLab academic environment.

REFERENCES

[1] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas, "Kilocore: A 32-nm 1000-processor computational array," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 891–902, April 2017.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, June 2011.

[3] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich cmps," in *DAC Design Automation Conference 2012*, pp. 843–849, June 2012.

[4] M. Lyons, M. Hempstead, G. Wei, and D. Brooks, "The accelerator store framework for high-performance, low-power accelerator-based systems," *IEEE Computer Architecture Letters*, vol. 9, pp. 53–56, Feb 2010.

[5] R. Leupers and O. Temam, eds., *Processor and System-on-Chip Simulation*. 2010.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[7] Synopsys, "HAPS ® Family of FPGA-Based Prototyping Solutions." http://www.synopsys.com/Systems/FPGABasedPrototyping/Pages/HAPS.aspx, 2014.

[8] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A Case for FAME: FPGA Architecture Model Execution," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 290–301, 2010.

[9] C. Pinto, S. Raghav, A. Marongiu, M. Ruggiero, D. Atienza, and L. Benini, "GPGPU-Accelerated Parallel and Fast Simulation of Thousand-Core Platforms," in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 53–62, 2011.

[10] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[11] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks, "The aladdin approach to accelerator design and modeling," *IEEE Micro*, vol. 35, pp. 58–70, May 2015.

[12] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 16:1–16:26, Sept. 2009.

[13] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, March 2004.

[14] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 97–108, June 2014.

[15] J. Martin, "Risc-v, rocket, and rocc." https://inst.eecs.berkeley.edu/~cs250/sp17/disc/lab2-disc.pdf, 2017.

[16] R. Nikhil and K. Czeck, *BSV by Example: The Next-generation Language for Electronic System Design*. Bluespec, 2010.

[17] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[18] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The LEAP FPGA Operating System," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept. 2014.

[19] J. Emer, C. Beckmann, and M. Pellauer, "Awb: The asim architect's workbench," in *3rd Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS 2007)*, 2007.

[20] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, (New York, NY, USA), pp. 63–74, ACM, 1991.

[21] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 110–119, Oct 2014.

[22] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," in *Proceedings of the 47th Design Automation Conference*, pp. 463–468, 2010.

[23] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 406–417, Feb. 2011.

[24] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 249–261, 2007.

[25] M. E. S. Elrabaa, A. Hroub, M. F. Mudawar, A. Al-Aghbari, M. Al-Asli, and A. Khayyat, "A very fast trace-driven simulation platform for chip-multiprocessors architectural explorations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 3033–3045, Nov 2017.

[26] J. Cong, Z. Fang, M. Gill, and G. Reinman, "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 380–387, Nov 2015.

[27] S. Savas, Z. Ul-Abdin, and T. Nordström, "Designing domain-specific heterogeneous architectures from dataflow programs," *Computers*, vol. 7, p. 27, 04 2018.