LoopPara: An Architecture-Transparent Acceleration Framework for Loops by Exploiting Data-Level Parallelism

Mingtao Chen
School of Electronic and Computer Engineering
Peking University
Shenzhen, China
mingtaochen@pku.edu.cn

Xianfeng Li *
Macau University of Science and Technology
Macau, China
xifli@must.edu.mo

Weikang Zhou, Yang Li, Fan Deng
School of Electronic and Computer Engineering
Peking University
Shenzhen, China
{zwk1994, yl0806, dengfan}@pku.edu.cn

2021 IEEE 6th International Conference on Computer and Communication Systems (ICCCS) | 978-1-6654-1256-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCCS52626.2021.9449282

***Abstract*—Exploiting inter-iteration data-level parallelism (DLP) is the important focus of many loop accelerators. Target at this, we propose a loop parallelization acceleration framework LoopPara, which follows a hardware/software coordinated pipeline: 1) an offline parallelism analysis is used to extract and encode application behavioral semantic of loop instructions as configuration. 2) a specialized loop accelerator loads the configuration for hardware mapping and guide the loop parallelism execution. Unlike traditional loop accelerators, LoopPara does not need instruction set architecture (ISA) extension, compiler modification, and software kernel replacement, which has better platform portability and can be easily integrated with general- propose processors with any ISA. Experiments show that, when coupling to an in-order RISC-V Rocket core, the LoopPara achieves 2.31x-5.42x (4.17x on average) speedups, which out- performs other high-performance out-of-order BOOM cores.**

Keywords—loop, accelerator, data-level parallelism

I. INTRODUCTION

Recent years have witnessed an explosive growth of accelerators in computer systems to exploit the potential of Data-Level Parallelism (DLP) in the hot loops that dominate application performance and execution time. In essence, most loops with good DLP potential have small instruction foot-print, but execute a large number of iterations and cover most of the dynamic instructions of programs. Normally, hardware specialization and general-purpose processor (GPP) based methods are two solutions for exploring DLP in loops. DLP hardware specializations like TPU [1] achieves high performance and efficiency, but they only cover a specific algorithm and is limited in generality and adaptability. For the GPP-based method, mainstream architectures (X86, ARM) usually use SIMD extensions to accelerate vector operations. Moreover, thread-level parallelism [2] attempts to accelerate loops by exploiting multi-core scheduling. Above GPP-based solutions are more general and programmable but sacrifice too much efficiency compared with domain-specific hardware.

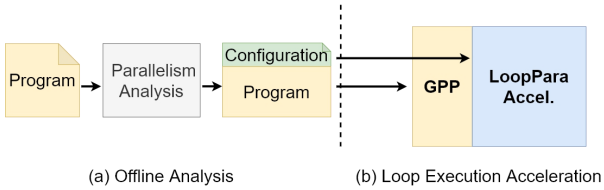

Fig. 1. LoopPara overview.

Recently, integrating specialized hardware into a GPP be- come a more popular way to maintain flexibility and relatively high performance ([3]–[6]). However, these methods usually need new ISA(Instruction Set Architecture) extensions and the corresponding compiler modifications to generate custom instructions. Programmers have to manually modify and replace the computing kernels of the source code with a set of well-designed software interfaces, which introduces too much manual effort and is not transparent enough. These heterogeneous solutions lead to binary incompatibility problems: an application compiled to utilize an accelerator cannot run on a processor without that accelerator. Moreover, mainstream platforms (X86,ARM) have integrated mature ISA extensions (AVX, SVE), and new ISA extensions by third parties are difficult for them to adopt.

In this paper, we propose LoopPara, an architecture-transparent loop acceleration framework that can overcome the application incompatibility discussed above (Figure 1). Unlike traditional loop accelerators, our LoopPara does not require ISA extension and follows a hardware/software coordinated pipeline:

* : Corresponding Author

Firstly, we introduce the offline program parallelism analysis based on our instruction operand source classification and dependency analysis. By using it, we can: 1) determine whether a hot loop can be accelerated in parallel. 2) get the configuration data. It is worth mentioning that the configuration is not custom instructions but a small amount of high-level application semantic information. Secondly, while loading the program, the accelerator will also load the configuration. When entering the loop region that can be accelerated, the GPP switches to the accelerator to execute those loop instructions. Moreover, the accelerator will utilize the configuration for hardware mapping.

The traditional vectorization technique of many compilers generates a binary with SIMD instructions embedded in it. That means the vectorized binary is tightly bound to a specific platform. In our framework, we decouple the program DLP semantics and the binary itself, the accelerator still executes the original loop instructions and only utilizes some software semantic information generated by the offline analysis. In this software and hardware collaborative design, ISA extensions, compiler modifications, and software kernel modifications are not required, which is simple, more transparent, and platform compatible. Our experiment results show that, when coupled to an in-order Rocket core [7], LoopPara is able to achieve 2.31x-5.42x (4.17x on average) speedups, which also outperforms other high performance multi-issue out-of-order BOOM processors [8].

The rest of this paper is organized as follows. In Section II we introduce the offline parallelism analysis. Our proposed loop accelerator is presented in Section III. Experimental results are provided in Section IV. Related work is presented in Section V. We conclude this paper in Section VI.

II. Offline Parallelism Analysis

The target of offline analysis is to identify loops that can be parallelized and extract the DLP semantics of loop instructions. Since the acceleration opportunity mainly comes from inter-iteration parallel computing, we focus on the arithmetic instructions (e.g., ADD/SUB/MUL) within a loop, which can be expressed as *Operand1 op Operand2* ← *Result*. The source of *Operand1* and *Operand2* determines the DLP pattern of instructions.

A. Classification of Instruction Operand Source

First of all, we introduce a classification of instruction operand source for convenience of explanation, the sources of *Operand1* and *Operand2* of instruction can be divided into five categories: A) Constant in a loop; B) Loop induction variable; C) The computing result of another instruction in the same iteration; D) The computing result of the instruction itself of the previous iteration; E) The computing result of another instruction of the previous iteration; We use a pair {*Category 1, Category 2*} to present a computing instruction whose two operand sources belong to Category 1 and Category 2 respectively.

B. Parallelism Analysis and Classification

For a computing instruction within a loop, the combination of operand source determines whether it can be parallel or not, as well as its DLP pattern. The overall analysis results are as follow:

Full Parallelism For a {*C,C*} instruction, if the two instructions which produce C can be executed in parallel, then {*C,C*} can also be accelerated in parallel. That means, conceptually, we can get the two operands' all values of different iterations and perform computing at the same time. We classify this situation as **full parallelism**, as shown in Figure 3(a). Similarly, if the instruction producing C can be parallel, The {*C,A*} can be parallel by broadcasting the constant A to compute with C of different iteration. And the {*C,B*} can also be executed in parallel by generating all the iteration versions of the loop induction variable B to perform computing simultaneously. Moreover, the {*B,B*}, {*B,A*} cases can also match the *full parallelism* pattern. Specially, a branch instruction also follows a *full parallelism* execution to generate an array of predicates (0/1 masks), and other instructions guarded by this branch only execute those iterations with predicate values of 1.

Tree Parallelism If all iteration versions of C can be provided simultaneously, the {*D,C*} instruction follows a loop-carry dependency pattern, which is usually seen in reduction operations(e.g., sum+=A[i]). This instruction itself is difficult to be parallel since the result of each iteration can only be produced sequentially. But the computing of its final result can be accelerated using a tree-like multi-stage pipeline. We classify this special form of DLP as **tree parallelism** (shown in Figure 3(b)). Similar cases includes {*D,A*}, {*D,B*} and {*D,E*}. In essence, the parallelism analysis is an instruction-level data dependency tracing. The original operand source of computing instruction usually comes from LOAD instruction, whose dependency pattern can affect the parallelism of arithmetic instructions (e.g., a[i] = a[i-3] + 2 has inter-iteration memory dependency between LOAD and STORE instruction and hence is difficult to provide operands for the computing instruction in parallel). Theoretically, when all arithmetic instructions in a loop can be parallel, this loop is suitable for hardware parallel acceleration.

C. DLP Pattern Classification and Hardware Mapping

Figure 2 shows the inner-loop instructions of *GEMM* example which performs matrix-multiplication using standard triple-nested loops. The instruction 3 (*mulw a4, a4, a6*) is {*C, C*} type and should be tagged with *FullParallel* pattern, since its two operands (*a4*, *a6*) of different iterations can be provided by the instruction 1 and 2 simultaneously. And it can be mapped to the PE array (Figure 3 (a)). The instruction 6 (*addw a1, a1, a4*) belongs to {*D, C*} type because the source of operand *a1* comes from the result of the instruction itself of the previous iteration and *a4* is from the instruction 3 of intra- iteration, which conforms the loop-carry pattern. So we mark the instruction 6 with *TreeParallel*. This *TreeParallel* instruc- tion can be mapped to the reduction tree hardware (Figure 3(b)) to perform multi-stage tree-like reduction operations. Specially, the {*B,A*} instruction, for instance, the instruction 4 in Figure 2, follows a calculation to increase *a4* value in each iteration, but all the *a4* versions can be generate in parallel by performing *base + iter idx * stride* calculation in PE array

(see Figure 3(c)), where the *a3* is the *base*, the constant 4 is the *stride* and *iter idx* is provided with each PEs' internal id. Although still in the full parallelism form, the parallel execution of this case has some differences in its hardware datapath, and we specifically categorized this case as **LIdx- Parallel**. Moreover, the {*B,A*} instruction is sometimes bound to the LOAD/STORE instruction for memory address computing(coincidently, like the instruction 4 and 5 in Figure 2), and we actually use it for memory unit configuration instead of mapping it to PE array (Figure 3(d)). For this reason, we specifically categorized this case as **MAddrParallel** pattern. That means *MAddrParallel* pattern takes priority when this instruction also matches *LIdxParallel* pattern.

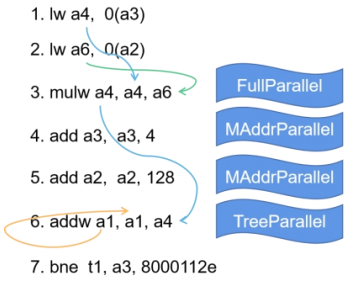

Fig. 2. Assembly Code and DLP Pattern Extraction.

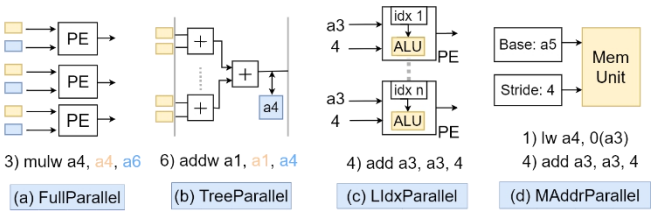

Fig. 3. DLP Pattern and Hardware Mapping.

D. Generating Configuration

With the offline analysis above, We can extract the DLP pattern information and output a configuration file (as shown in Figure 4). The header contains the list of the loops to be accelerated. *LAddr* and *#Ins* represent the start address of each loop and the number of instructions it has. The Body gives DLP patterns of instructions within each accelerated loop. This configuration file is the input of the loop accelerator, and the implementation details are in Section IV.

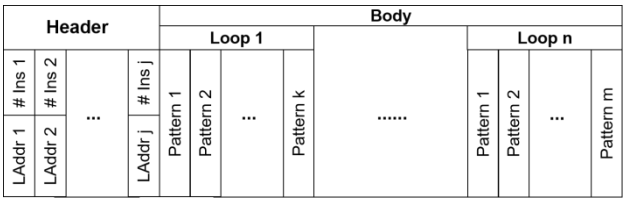

Fig. 4. Configuration format.

III. Microarchitecture Design

For accelerating the loops detected by our offline analysis, we design and implement a loop accelerator and integrate it into an open-source RISC-V in-order Rocket core [7] by extending the Rocket Custom Coprocessor Interface(RoCC) (as shown in Figure 5). Conceptually, every core in a system can be augmented with our accelerator without elaborate modifications. The difference between traditional accelerator architecture and our design is that our accelerator does not have ISA extension. In our framework, the accelerator still executes the original loop instructions and the configuration generated by the offline analysis is used as the input of our accelerator for DLP execution scheduling.

A. Specialized Execution Switching

Configuration Loading At first, the Rocket normally loads a program through the program loader. When the accelerator detects the configuration in the binary, it loads the configuration into the Loop Header Information Table (HIT) and the Loop Body Information Table (BIT) according to the configuration format parsing. HIT contains basic information of all the inner loop regions that can be accelerated, and BIT stores the DLP patterns of each instruction within these loops. **Execution Switching** By slightly modifying the decoder of the Rocket, we can realize the switching between traditional and specialized execution easily. Normally, the Rocket core forwards instructions through the pipeline and commits them at the write-back stage. And when entering a loop region in HIT, the decoder of Rocket generates special control signals to drive the Rocket directly sends those instructions of this loop through the pipeline without operations and forwards them to the Loop Instruction Buffer (LIB) in the accelerator side (along with the corresponding register values). After that, the Rocket core stalls the pipeline and switches the control to the accelerator side. Note that the accelerator needs to transmit the necessary context information back to the Rocket when the loop execution ends.

B. Controller

The main state machine of the controller is responsible for controlling the cooperation between the computing unit and the DMA engine.

For executing a loop, the decoder in the controller fetches the instructions from the Loop Instruction Buffer (LIB) and their corresponding DLP pattern from the BIT. The decoder generates control signals and triggers the configurator for task dispatch and hardware activation. In the decoding stage, our accelerator decodes the original instructions instead of custom instructions. With the help of DLP pattern in BIT, the configurator (low-level hardware) can understand the high-level behavioral semantic of instructions and send those instructions to different hardware unit for different function executions (as shown in Figure 3(a) (b) (c) (d)).

C. Memory Access Unit

In order to improve the ability of large-scale data access, we design and implement a DMA module instead of us- ing L1 cache memory access interface. The instruction with *MAddrParallel* pattern and its corresponding *LOAD/STORE* instruction are used for configuring DMA. The advantages of DMA engine come from two aspects: 1) Requesting data in a pipeline way. This means that one request, then a pile of data come in succession. 2) Execution/memory access decoupling. The memory access unit contains a buffer which enables the DMA engine perform data fetching while the computing unit is busy.

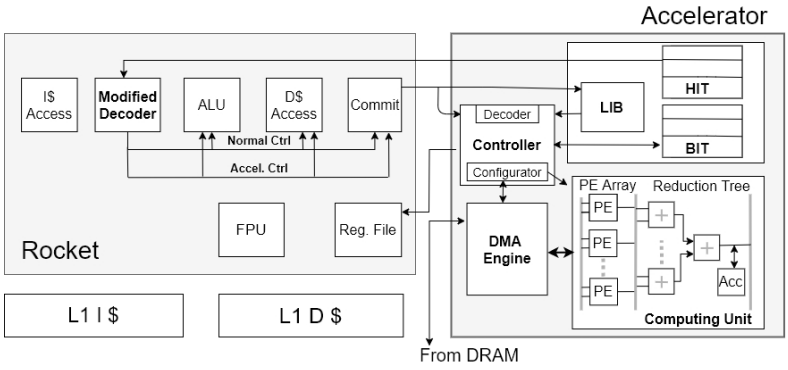

Fig. 5. The Microarchitecture of LoopPara.

D. Computing Unit

PE Array and Reduction Tree As discussed in Section II, computing instruction with *FullParallel* pattern is mapped to the PE array. It is noted that each PE in PE array has a local register file that has the same number of registers as the register file in Rocket. The *full parallelism* execution is as follows: 1) At first, each PE loads the corresponding elements of different iterations (which fed by the DMA engine) into their corresponding local register. 2) During computing, each PE fetches two operands from the local registers and perform computing at the same time. 3) Lastly, the result of each PE is written into its corresponding local destination register. The results of the previous instruction can be used as the operands of the next instruction by directly fetching data from the corresponding local registers of PEs, which reduces the data movement from memory.

Arithmetic instruction with *LIdxParallel* pattern is also executed in the PE array as discussed in Section II (C). Arithmetic instruction with *TreeParallel* pattern is mapped to tree-like hardware called reduction tree (see Figure 5), which performs multi-stage reduction acceleration. The initial input of the reduction tree comes from the PE array or DMA engine. The intermediate results can be stored in *Acc* register when meeting a very long vector.

IV. Implementation and Evaluation

A. Implementation and Framework Workflow

We use Chisel [9] for the RTL implementation of our accelerator. The PE arrry has 32 PEs, and the reduction tree consists of 31 reduction nodes. The parameter of Rocket core is shown in Table I. For parallelism analysis, we utilize LLVM [10] use-definition chain analysis to trace the source of instruction operands and memory dependency, then extract DLP patterns and generate the final configuration file.

The workflow of our proposed loop acceleration framework is shown in Figure 6. At the offline stage (step 2, 3, 4, 5), we perform parallelism analysis on programs and extract DLP patterns, our configuration is encoded in a *config.data* file, which is included into the main program file. Moreover, a Macro config macro, which contains a RoCC instruction (standard R-type RISC-V instruction, belongs to the Rocket's ISA), is used to tell the address of configuration to the accelerator at the earliest entry of the *main* function. In this way, we do not introduce any ISA extension, compiler and application kernel modifications. On the platform without the loop accelerator, we simply need to exclude the configuration file and re-compile it so that the program can execute on this platform.

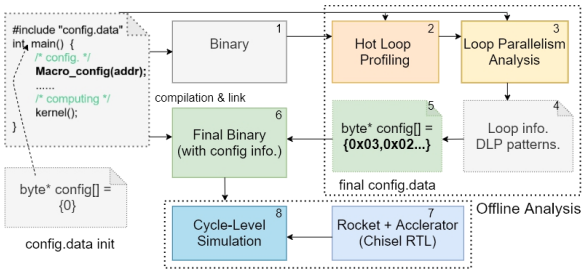

Fig. 6. LoopPara Framework Workflow.

B. Evaluation

Benchmark Analysis we select 7 benchmarks from Poly-Bench [11] and our own custom kernels. *gemm* implement ma- trix multiplication using standard triple-nested loops. *warshall* is the Floyd-Warshall algorithm. *axpy* implement simple vector operations. *mvt* performs matrix-vector product and transpose operations. *knn*, *svm* and *pca* are popular machine learning algorithms. We use our proposed offline parallelism analysis to detect the acceleratable inner loops in each benchmark and Table II reveals the characteristic of these hot inner loops: A small number of inner loop regions with a small number of static instructions can cover most of the execution cycles and dynamic instructions of the program. These loops can be mapped to our accelerator for parallel execution and get performance improvements.

Performance Comparison We integrate the proposed accelerator into the Rocket core and perform cycle-level performance comparison with other two different high-performance BOOM processors (see Table I). Experimental results are shown in Figure 7. The speedups are normalized to the Rocket core. The performance of the two BOOM cores is better than the in-order Rocket core, and the best one, the large BOOM gets 1.38x-2.63x speedups (2.12x on average). Augmented with the proposed accelerator, our framework LoopPara (Rocket+Accel.) is able to achieve 2.31x-5.42x (4.17x on average) speedups, which is significantly outperforms other high performance BOOM cores.

TABLE I: GPP BASELINE CHARACTERISTICS

	Rocket	Small BOOM	Large BOOM
Issue-width	1	1	3
Fetch-width	1	4	8
Issue slots	-	12	20
ROB entries	-	32	96
Ld/St entries	-	8/8	24/24
Registers(int/fp)	32/32	52/48	100/96
L1 I$ and D$ (KiB)	16/16	16/16	32/32
L2 D$ (KiB)	512	512	512

The Overhead of Configuration Loading When executing a program, our LoopPara needs to load the header of configuration into the HIT table at the beginning and load the DLP pattern information of the specific loop into the BIT table on demand when entering the corresponding loop on runtime. This results in some loading overheads for the acceleration execution and Figure 8 shows the proportion of execution cycles used for HIT and BIT loading. As the result presents, the configuration loading overheads of all benchmarks are less than 5%, which has negligible influence on the acceleration execution.

C. *Power and Area Estimation*

For the power and area estimation, we synthesize the Verilog code of the accelerator with Synopsis Design Compiler using TSMC 65nm technology library. As the result shown in Table III, Our LoopPara consumes less power than the large BOOM and has a smaller chip area while achieving higher performance than the large BOOM.

TABLE II: ACCELERA TABLE LOOP CHARACTERISTICS OF BENCHMARKS

Name	# Hot Loop	% Exec Cycles	Static Ins.	Iters	Dyn Ins.
gemm	1	99%	7	260K	1.8M
warshall	1	91%	9	1.9M	16M
mvt	2	87%	8	16K	131K
axpy	1	90%	9	12K	111K
knn	3	91%	8-12	260K-2M	2.6M-23M
svm	2	89%	7-11	4K-524K	28K-6M
pca	4	86%	5-14	160K-520K	810K-6M

* # Hot Loop = the number of inner loops that can be accelerated;
% Exec Cycles = % of all hot loop execution cycles; Iters = range for number of hot loop iterations; Static Ins = range for static instruction counts for each hot loop; Dyn Ins = range for dynamic instruction counts for each hot loop;

TABLE III: POWER/AREA ESTIMATION

	Area (mm^2)	Power (mW)	Power/Area (mW/mm^2)
Rocket	0.50	77.17	154.34
small BOOM	1.01	161.61	160.01
large BOOM	2.79	236.50	84.77
LoopPara	2.04	185.21	90.79

V. RELATED WORK

For loop acceleration, SIMD extensions and Advanced Vector Extensions (AVX) are commonly used in many mainstream general-purpose processors. GPUs have extremely strong data parallel processing capability compared with traditional vector processors but require a set of different ISA.

Loop accelerators integrated with GPPs are popular in recent years. The work in [12] has proposed a programmable loop accelerator and a compiler that is responsible for mapping loops onto this accelerator. DySER [5] enables the subgraph of a loop to execute on the accelerator, which is more likely to exploit intra-iteration instruction-level parallelism (ILP). Hwacha [6] is a vector accelerator based on Rocketchip and support multi-vector element operations for loop acceleration. XLOOP [4] not only explores inter-iteration DLP, but also supports other inter-iteration control dependency pattern. Therefore, its architecture is more complex and different from ours. Moreover, XLOOP encodes iteration pattern by introducing minor ISA modifications. Stream-Flow [3] provides an abstraction (requiring ISA extensions) that balance the tradeoffs of vector and spatial architectures, and obtains high efficiency on many data-processing workloads.

Overall, all these loop acceleration architectures above are not architectural transparent and software compatible because they all need ISA extensions and compiler modifications to generate specific instructions. Without needing ISA extensions, our framework is architectural transparent and has better platform compatibility.

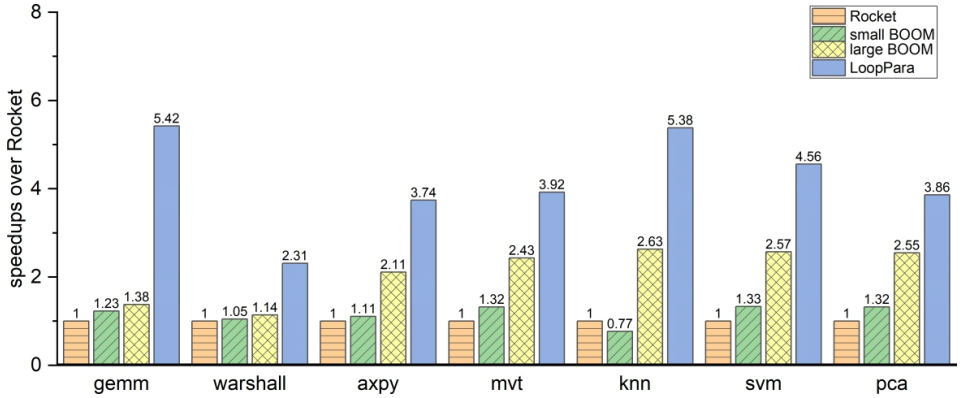

Fig. 7. Speedup of LoopPara and BOOM cores over Rocket.

Fig. 8. HIT&BIT Loading Overhead.

VI. CONCLUSIONS

In this paper, we introduce LoopPara, an architecture-transparent acceleration framework based on RISC-V for exploiting inter-iteration DLP. In our hardware/software pipeline, we only need to transmit a small amount of configuration information generated by an offline parallelism analysis to the accelerator without needing ISA extension, compiler modifications, and application kernel modifications. And the experimental results show significant performance improvement.

ACKNOWLEDGEMENTS

The work is sponsored by the Shenzhen Science and Technology Innovation Committee (SZSTI) under the grant for basic research (No. JCYJ20170818090006804).

REFERENCES

[1] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Boyle, R. (2017, June). In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture (pp. 1-12).

[2] Kim, J., Jiang, S., Torng, C., Wang, M., Srinath, S., Ilbeyi, B., ... & Batten, C. (2017, October). Using intra-core loop-task accelerators to improve the productivity and performance of task-based parallel programs. In 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 759-773). IEEE.

[3] Nowatzki, T., Gangadhar, V., Ardalani, N., Sankaralingam, K. (2017, June). Stream-dataflow acceleration. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA) (pp. 416-429). IEEE.

[4] Srinath, S., Ilbeyi, B., Tan, M., Liu, G., Zhang, Z., Batten, C. (2014, De- cember). Architectural specialization for inter-iteration loop dependence patterns. In 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 583-595). IEEE.

[5] Govindaraju, V., Ho, C. H., Sankaralingam, K. (2011, February). Dynamically specialized datapaths for energy efficient computing. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture (pp. 503-514). IEEE.

[6] Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanovic´, V., & Asanovic´, K. (2014, September). A 45nm 1.3 GHz 16.7 double- precision GFLOPS/W RISC-V processor with vector accelerators. In ESSCIRC 2014-40th European Solid State Circuits Conference (ESS- CIRC) (pp. 199-202). IEEE.

[7] Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., ... & Karandikar, S. (2016). The rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS- 2016-17.

[8] Asanovic, K., Patterson, D. A., & Celio, C. (2015). The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. University of California at Berkeley Berkeley United States.

[9] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Aviz˜ienis, R., ...& Asanovic´, K. (2012, June). Chisel: constructing hardware in a scala embedded language. In DAC Design Automation Conference 2012 (pp. 1212-1221). IEEE.

[10] Lattner, C. A. (2002). LLVM: An infrastructure for multi-stage optimiza- tion (Doctoral dissertation, University of Illinois at Urbana-Champaign).

[11] Pouchet, L. N. (2012). Polybench: The polyhedral benchmark suite. URL: http://www. cs. ucla. edu/pouchet/software/polybench, 437.

[12] Fan, K., Kudlur, M., Dasika, G., Mahlke, S. (2009, February). Bridging the computation gap between programmable processors and hardwired accelerators. In 2009 IEEE 15th International Symposium on High Performance Computer Architecture (pp. 313-322). IEEE.