Simty: generalized SIMT execution on RISC-V

Sylvain Collange

Simty: generalized SIMT execution on RISC-V

Sylvain Collange
Inria
sylvain.collange@inria.fr

ABSTRACT

We present Simty, a massively multi-threaded RISC-V processor core that acts as a proof of concept for dynamic inter-thread vectorization at the micro-architecture level. Simty runs groups of scalar threads executing SPMD code in lockstep, and assembles SIMD instructions dynamically across threads. Unlike existing SIMD or SIMT processors like GPUs or vector processors, Simty vectorizes scalar general-purpose binaries. It does not involve any instruction set extension or compiler change. Simty is described in synthesizable RTL. A FPGA prototype validates its scaling up to 2048 threads per core with 32-wide SIMD units. Simty provides an open platform for research on GPU micro-architecture, on hybrid CPU-GPU micro-architecture, or on heterogeneous platforms with throughput-optimized cores.

KEYWORDS

SIMT, SIMD, FPGA, RISC-V

ACM Reference Format:
Sylvain Collange. 2017. Simty: generalized SIMT execution on RISC-V. In *Proceedings of First Workshop on Computer Architecture Research with RISC-V, Boston, MA, USA, October 2017 (CARRV 2017),* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The *Single Instruction, Multiple Threads* (SIMT) execution model as implemented in NVIDIA Graphics Processing Units (GPUs) associates a multi-thread programming model with an SIMD execution model [24]. The flexibility obtained by running multi-thread programs on SIMD units makes SIMT attractive for heterogeneous many-core processors, where different cores may use different physical SIMD widths and multi-threading depths. However, current SIMT architectures demand specific instruction sets. In particular, they need specific branch instructions to manage thread divergence and convergence. SIMT GPUs have remained so far incompatible with traditional general-purpose CPU instruction sets.

We argue that the SIMT execution model can be generalized to a general-purpose instruction set at a very low hardware cost. As experimental evidence, we introduce Simty, a generalized-SIMT core that runs the RISC-V instruction set. Simty lifts the binary incompatibility between latency-oriented CPU cores and throughput-oriented GPU-like cores by letting them share a single, unified instruction set. It enables commodity compilers, operating systems and programming languages to target GPU-like SIMT cores. Beside simplifying software layers, the unification of CPU and GPU instruction sets eases prototyping and debugging of parallel programs. We have implemented Simty in synthesizable VHDL and synthesized it for an FPGA target.

We present our approach to generalizing the SIMT model in Section 2, then describe Simty's microarchitecture is Section 3, and study an FPGA synthesis case Section 4.

CARRV 2017, October 2017, Boston, MA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 CONTEXT

We introduce some background about the generalized SIMT model and clarify the terminology used in this paper.

2.1 General-purpose SIMT

The SIMT execution model consists in assembling vector instructions across different scalar threads of SPMD programs at the microarchitectural level. Current SIMT instruction sets are essentially scalar, with the exception of branch instructions that control thread divergence and convergence [8]. Equivalently, an SIMT instruction set can be considered as a pure SIMD or vector instruction set with fully masked instructions, including gather and scatter instructions, and providing hardware-assisted management of a per-lane activity mask [13]. The thread divergence and convergence hardware is exposed at the architecture level, so current SIMT instruction sets offer a variety of custom control-flow instruction instead of the usual conditional and indirect branches.

We have shown in prior work that the SIMT execution model could be generalized to general-purpose instruction sets, with no specific branch instructions [5, 8]. This technique is amenable to an efficient representation of program counters using a sorted list [5, 9]. Simty implements a sorted path list with one active path.

2.2 Terminology

A multi-thread or multi-core processor exposes a fixed number of hardware threads, upon which the operating system schedules software threads. Unless specified otherwise, a *thread* will designate in the rest of the paper a hardware thread, or hart in RISC-V terminology. In an SIMT architecture, threads are grouped in fixed-size *warps*. This partitionning is essentially invisible to software.

Register file and execution units are organized in an SIMD fashion. The SIMD width corresponds to the number of threads in a warp. Each thread of a warp is statically assigned a distinct SIMD lane. Its context resides fully in a private register file associated with its lane. By virtue of the multi-thread programming model, there is no need for direct communication between registers of different lanes.

As the programming model is multi-thread, each thread has its own architectural Program Counter (PC). However, the microarchitecture does not necessarily implement these architectural PCs

as separate physical registers. To account for thread-dependent control flow, we introduce an intermediate level between thread and warp that we call *Path* [9]. Each *path* tracks one of the different paths followed by threads in the program. The path concept is also found under the names *warp-split* [23], *context* [5], *fragment* [18], or *instruction stream* [14] in the literature. Threads of a path advance in lockstep and share a Common Program Counter (CPC) in the same address space. All threads in a given path belong to the same warp. Thus, each warp contains from 1 to m paths, where m is the number of threads per warp. The set of threads in a path is represented in hardware by an m-bit mask for each path. Mask bit i of path j is set when thread i of the warp belongs to path j. We can represent the state of a warp equivalently either as a vector of m PCs, or as a set of paths identified by (CPC, mask) pairs. In addition to the PC, the state of a path may contain information to help path scheduling, such as function call depth and privilege level.

Following branches in SIMT may be seen as traversing the control-flow graph. The processor follows one path of each warp, that we name the *active* path. The other paths are saved in a list. As threads of a warp take different directions through branches or get back to executing the same instructions, paths may *diverge* or *converge*.

- Control-flow divergence occurs when the active path encounters a branch, and its threads take different directions through the branch. It is implemented by splitting the active path into two new paths. One path is saved in the list, and execution continues on the other path.
- Control-flow convergence is detected dynamically. It consists in merging the active path with a path of the list, or two paths of the list, when they have the same PC.

3 SIMTY MICROARCHITECTURE

We present design principles, then an overview of the Simty pipeline, and detail path tracking mechanisms.

3.1 Design principles

As an SIMD-based architecture, the key idea of Simty is to factor out control logic (instruction fetch, decode and scheduling logic) while replicating datapaths (registers and execution units). The pipeline is thus build around a scalar multi-thread front-end and an SIMD backend.

General-purpose scalar ISA. We chose the RISC-V open instruction set, without any custom extension [26]. Indeed, RISC-V is well-suited to many-thread architectures thanks to its small per-thread state. Simty currently supports the RV32I subset (general-purpose user-level instructions and 32-bit integer arithmetic) and a subset of privileged instructions for hardware thread management.

Throughput-oriented architecture. Simty focuses on exploiting thread-level parallelism across warps to hide execution latency, as well as data-level parallelism inside paths to increase throughput. On the other hand, the initial implementation of Simty does not attempt to leverage instruction-level parallelism beyond simple pipelined execution to focus on SIMT-specific aspects. For instance, in order to simplify bypass logic, a given warp cannot have instructions in two successive pipeline stages. This restriction matches the limitations of industrial designs like Nvidia Fermi GPUs [24] or Intel Xeon Phi Knights Corner [7].

Configurable. Warp and thread counts are configurable at RTL synthesis time. The typical design space we consider ranges between 4 warps × 4 threads and 16 warps × 32 threads, to target design points comparable to a Xeon Phi core [7] or an Nvidia GPU SM [24]. The RTL code is written in synthesizable VHDL. Pipeline stages are split into distinct components whenever possible to enable easy pipeline re-balancing or deepening. All internal memory including the register file is made of single read port, single write port SRAM blocks. This enables easy synthesis using FPGA block RAMs or generic ASIC macros.

Leader-follower resource arbitration. Simty uses an non-blocking pipeline to maximize multi-thread throughput and simplify control logic. Instruction scheduling obeys data dependencies. Execution hazards like resource conflicts are handled by partially replaying the conflicting instruction as follows.

When a path accesses a shared resource (e.g. a data cache bank), the arbitration logic gives access to an arbitrary *leader* thread of the path. All other threads of the path then check whether they can share the resource access with the leader (e.g. same cache block). Threads that do are considered as *followers* and also participate in the access. When at least one thread of the path is neither leader nor follower, the path is split in two paths. A path containing the leader and followers advances to the next instruction. Another path containing the other threads keeps the same PC to have the instruction eventually replayed. This partial replay mechanism guarantees global forward progress, as each access serves at least the leader thread.

PC-based commit. Program counters are used to track the progress of threads in paths. By relying on per-thread architectural PCs, the partial instruction replay mechanism is interruptible and has no atomicity requirement. Indeed, the architectural state of each thread stays consistent at all times, as the programming model does not enforce any order between instructions of different threads. This avoids the challenges of implementing precise exceptions in traditional vector processing [17].

3.2 The Simty pipeline

Simty is build around a 10-stage, single-issue, in-order pipeline as shown on Figure 1. We briefly describe each stage.

Fetch steering selects one warp and looks up the speculative CPC of its active path in a table indexed by the warp identifier. Warp scheduling currently follows a round-robin order. The speculative CPC is then updated to the predicted address of the next instruction. Currently, the prediction logic merely increments the CPC, though any actual branch predictor may be used instead.

Instruction Fetch/Predecode fetches an instruction word from the instruction cache or dedicated memory. A first predecode stage checks which operands reference registers. The RISC-V encoding enables a straightforward predecode logic to collect all the data that scoreboarding logic needs to track register dependencies between instructions. The predecoded instruction is placed in an instruction buffer containing one entry per warp.

Scheduler issues instructions when their operands are ready. A qualification step selects warps whose next instruction is ready

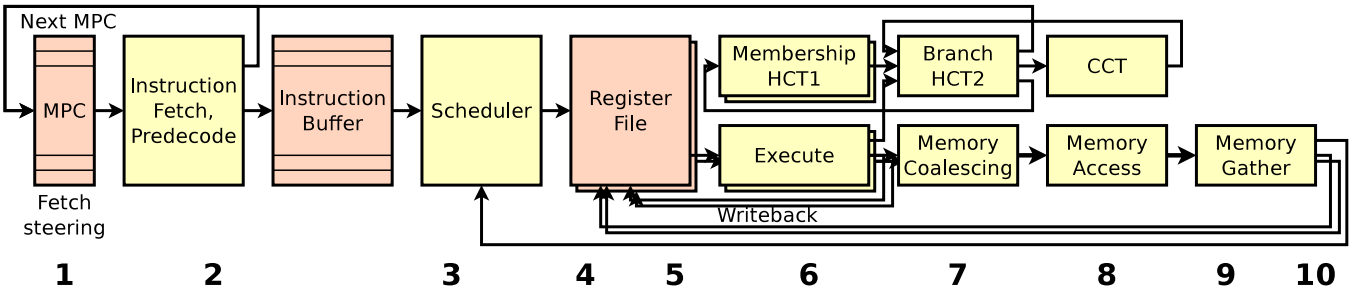


Figure 1: Simty pipeline.

to be executed. It considers both instruction dependencies and execution resource availability: bank conflicts in the register file are proactively avoided at this stage. A selection step then picks one of the ready warps marked by the qualification step. The current scheduling policy is loose round-robin: the first ready warp starting from the round-robin warp is selected. Other policies like pseudo-random scheduling are possible to avoid systematic conflicts in memory accesses.

Register File. Operand collection logic reads instruction operands from the register file (RF) during two pipeline stages. Each SIMD lane has its own RF. Each of these RFs is further partitioned into two banks based on the warp identifier. One bank handles warps of even ID, and the other bank handles warps of odd ID, an organization referred to as "thin" allocation [21]. Each bank has a single read port and a single write port.

- For instructions that take two source register operands, registers are read serially from the same bank over two cycles. To avoid conflicts, the next instruction has to belong to a warp whose registers are in the other RF bank. This constraint is enforced by the scheduler stage.
- For instructions that take a single source register operand, both banks are available to the next instruction.

The write port of each bank is available for writebacks from the ALUs. Output values from the memory access unit are opportunistically written back on free ports from a FIFO queue. Instructions are fully decoded at this stage.

Execute. Execution units consist of one 32-bit ALU supporting the RV32I instruction set in each lane. All arithmetic instructions are currently executed in a single cycle, though the design can accomodate pipelined floating point units and integer multipliers in the future.

Membership, HCT1 and HCT2 run concurrently with the execute stage. The membership stage recovers the path thread membership mask of the warp that identifies which threads are active. For arithmetic instructions, the mask is the eventual non-speculative commit mask. For memory access instructions, it is a speculative mask under the assumption no exception occurs. From the mask, a priority encoder computes the identifier of the leader thread by finding the first bit set in the membership mask. The membership unit also detects convergence between paths. It is based on three tables HCT1, HCT2 and CCT in three successive pipeline stages, and will be described in more details in Section 3.3.

Writeback conditionally writes results from the array of execution units back to the respective RFs. For each lane i, the output is written back when bit i of the membership mask is set. The mask can be all zeroes in case of branch misprediction. A bypass network also forwards the output to the input stage of execution units. The membership mask also controls the bypass network on a per-lane basis, in order to prevent an instruction that precedes a convergence point to forward stale data to instructions that follow the convergence point.

Branch. The branch unit splits paths when executing a divergent branch instruction. It takes as inputs a condition vector c from the ALUs and the membership mask m from the membership unit. Both conditional branches and indirect branches may diverge.

- For a conditional branch, the scalar destination address PC_d is computed from the PC and the instruction immediate. Two paths are created. One path tracks threads that follow the branch ($\mathrm{PC}_d, c \wedge m$), and a second path tracks threads that fall through the next instruction ($\mathrm{PC}+4, \bar{c} \wedge m$). Empty paths are subsequently discarded.
- An indirect branch which computed address vector contains multiple different targets may diverge into more than two paths. This case is handled as a resource conflict. Branches to each unique target are serialized in order to have at most two paths out of the branch unit at each time. Following the leader-follower strategy, all threads that have the same target PC_l as the leader take the branch, while the other threads keep the same PC and have the indirect branch instruction replayed.

Memory coalescing and Memory access. The memory arbiter coordinates the accesses that threads of a path perform concurrently to a shared memory or cache. It is optimized for two common cases: (1) when threads of a warp access consecutive words from the same cache block, and (2) when all threads access the same word. Both cases can be detected by comparisons between the address requested by each thread and the address requested by the leader thread. Threads that do not obey either pattern form a new path and their instruction is replayed.

Memory gather distributes data obtained from the cache block in memory back to the threads of the path. In particular, it can broadcast the word accessed by the leader thread to the other threads, as well as perform parallel access to aligned data in the cache line, akin to the coalescing rules of first-generation NVIDIA Tesla GPUs [20].

Future research will have to evaluate the design space and tradeoffs between such minimalist interconnect and a full crossbar.

3.3 Path tracking

The path tracking unit is responsible for three functions: (1) check thread membership within a path to compute the validity mask of an instruction, (2) merge the paths that have identical PC to let threads converge, and (3) select the active path that is followed by the front-end.

Simty leverages the path representation based on (CPC, mask) pairs presented in Section 2.2. Function 1 simply amounts to checking that the instruction address to commit is equal to the PC of the active path and reading the associated mask. Functions 2 and 3 are implemented based on a list of paths sorted by priority. The path with the highest priority is the active path. Priority is given to the path in the deepest nested function call level, and in case of a tie to the path with the minimal PC [8]. As priorities are based on CPC values and there is a singe active path, path convergence only needs to be detected between the active path and the second path by priority order.

Path entries are kept sorted by function call level and PC. To manage efficiently a large number of paths in hardware, we isolate the two head paths, sorted on-line, from all other inactive paths, sorted off-line by a state machine. The head paths are kept in the Hot Context Tables (HCTs) indexed by the warp identifier which are accessed every cycle. The Cold Context Table (CCT) maintains the other, infrequently-accessed paths (Figure 2) [5].

The sort/compact unit in Figure 2 gathers paths from the previous state of the warp, from the next PC and from the outputs of the branch unit and instruction replay logic. For any instruction, this represents at most 3 path. It then merges entries that have the same PC and same function call level, discards paths with empty masks and sorts the resulting paths by priority order. After compaction and sorting, the first two entries by priority order are stored in their respective HCT. When only a single valid path remains after compaction, it is stored in the HCT 1, while the new HCT 2 entry is popped from the head entry of the CCT. When three valid path remain, the third one is pushed to the head of the CCT.

The CCT size is adjusted for the worst case of one path per thread. It contains a stack of paths for each warp, which head is followed by a pointer. The offline sorting unit is a state machine with 3 states and one pointer per warp. It performs an iterative insertion sort opportunistically when the CCT ports are available. Each pointer walks through CCT entries of the warp, and the pointed entry is compared with the second HCT entry. When the order does not match the desired priority order, these entries are swapped atomically. Together with the online compaction-sorting network, the offline sorting state machine eventually converges toward a fully sorted path list in at most m^2 steps with m-thread warps. The performance of this low-overhead implementation is adequate for the purpose of sorting paths, as CCT sorting only occurs in the case of unstructured control flow, and the order of paths does not affect correctness.

4 FPGA IMPLEMENTATION

We illustrate and evaluate an implementation of Simty on an FPGA target. Circuit synthesis for FPGAs is a first prototyping step toward hardware synthesis. It also represents an application in its own right: Simty can serve as a parallel controller for a reconfigurable accelerator, as an alternative to vector soft-core processors [25]. We synthesized and tested Simty on an Altera Cyclone IV EP4CE115 FPGA of an Altera DE-2 115 development board, for a target frequency of 50 MHz.

The main microarchitectural parameters are the warp count n and the warp width m. Warp width determines the number of parallel execution units and thus the execution throughput of a core. A Simty architecture with few cores and wide warps will benefit applications which threads present an homogeneous behavior and that take advantage from dynamic vectorization. Conversely, more cores with narrower warps will offer a more stable performance on irregular parallel code. Warp count determines tolerance to execution latency, especially from external memory.

Figure 3 presents synthesis results after place-and-route as a function of thread and warp count. The architecture scales up to 64 warps × 32 threads and 8 warps × 64 threads. The cost of control logic is amortized over SIMD execution units. The sweet spot on this platform is obtained between 8 warps × 8 threads and 32 warps × 16 threads. Beyond this point, routing congestion causes a sensible cycle time increase and the extra area gains against multiple Simty cores are low.

These results show that the overhead of generalized SIMT can be easily amortized, even in the context of a microcontroller-class single-issue RISC pipeline. As we add execution resources such as integer multipliers and floating-point units, we expect the cost of the path tracking logic will eventually become negligible.

5 RELATED WORK

Simty complements the existing set of open-source tools of the parallel architecture and GPU research communities. Multiple open-source cycle-level simulators of GPU architectures are available, including GPGPU-Sim [3], MV5 [22], Barra [10], SST-MacSim [15] and Multi2Sim [12]. Software-based simulators can be used to evaluate timing and sometimes energy consumption, but give no direct indication of hardware implementation cost. Simulation also incurs a slowdown of multiple orders of magnitude compared to actual hardware.

To address these limitations of simulators, several hardware-synthesizable parallel processors have been recently released in the academic community. They include vector processors, SIMD cores, and GPU-based architectures.

Vector processors. HWACHA is a synthesizable vector processors running vector instruction set extensions to RISC-V [19]. Vector soft-cores such as VectorBlox MXP target FPGAs [25]. Vector processors traditionally rely on explicit vector instruction sets. For instance, HWACHA augments RISC-V with one set of vector instructions as well as two sets of scalar instructions.

SIMD cores. Guppy is a processor based on Leon that runs a SPARC-based SIMD instruction set with basic predication support [1]. Nyuzi is a multi-thread SIMD processor for graphics rendering [6].

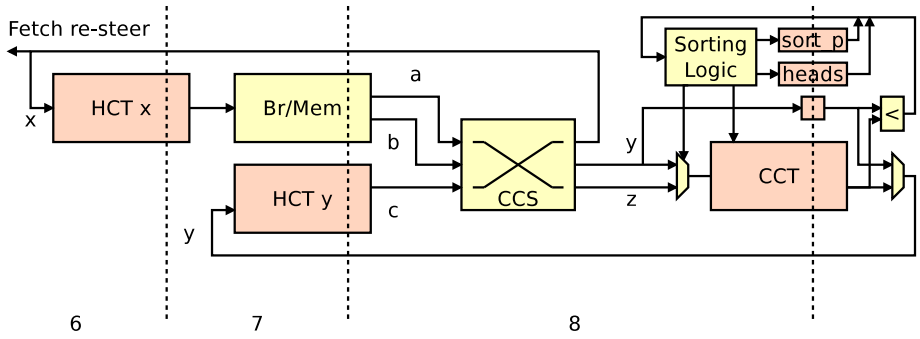
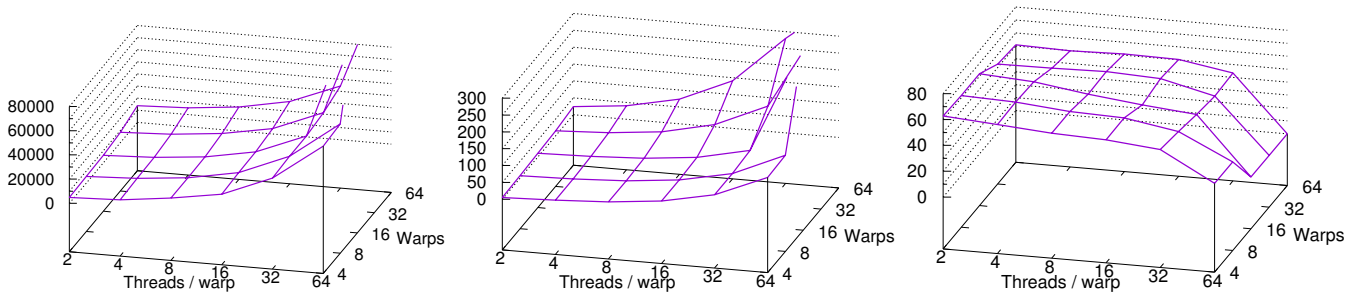


Figure 2: Detail of path tracking using path tables



(a) Area (LE) (b) Area (M9K) (c) Frequency (MHz)

Figure 3: Simty scaling on Altera Cyclone IV as a function of threads per warp m and warp count n. Frequency is the worst-case estimate at 85°C in MHz. Area is given in Logic Elements (LEs) and in 128 × 32-bit RAM block (M9K) count. Place-and-route fails on configurations with {16, 32, 64} warps × 64 threads as a result of routing congestion.

GPU-based architectures. Kingyens and Steffan propose a processor compatible with the fragment shader unit of the ATI R500 GPU architecture [16]. MIAOW is a synthesizable GPU based on the AMD GCN architecture [4]. Flexgrip [2] is a GPU based on the Nvidia Tesla architecture that follows the pipeline of the Barra simulator [10].

All the aforementioned processors are based either on explicit vector or SIMD instruction sets with per-lane predication, either on SIMT instruction sets with custom control flow instructions. Like Simty, the Maven architecture [18] uses a unified instruction set for both its scalar *control thread* and its vector *micro-threads*, but its RTL implementation is not publicly available. To our knowledge, Simty is the first open-source RTL-level SIMT processor that supports a scalar general-purpose instruction set.

6 FUTURE WORK

As a proof of concept, the initial Simty implementation focuses on clarity and simplicity. Following this proof of concept, we intend to incorporate floating-point, atomic and privileged instructions, and virtual memory support, as well as bringing up the software infrastructure, relying on RISC-V support in existing compiler toolchains and operating systems.

As in all conventional SIMT architectures prior to Nvidia Volta, thread synchronization and scheduling policies in Simty only guarantee global progress, ensuring at least one thread makes forward progress. However, individual threads have no guarantee of forward progress. Busy waiting loops may thus cause deadlocks [11]. Thus, the current implementation requires all inter-thread synchronization to use explicit instructions. In order to run more general SPMD code, path scheduling policies that balance fairness with convergence timeliness will be needed [14].

7 CONCLUSION

Simty demonstrates the hardware feasibility of microarchitecture-level SIMT. It implements the SIMT execution model purely at the microarchitecture level, preserving a scalar general-purpose instruction set at the architecture level. As such, it provides a building block for massively parallel many-core processors. Together with the rest of the RISC-V ecosystem, Simty provides an open platform for research on GPU micro-architecture, on hybrid CPU-GPU micro-architecture, or on heterogeneous platforms with throughput-optimized cores.

SOURCE CODE

Simty is distributed under a GPL-compatible CeCILL license at https://gforge.inria.fr/projects/simty.

REFERENCES

[1] Abdullah Al-Dujaili, Florian Deragisch, Andrei Hagiescu, and Weng-Fai Wong. 2012. Guppy: A GPU-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE, 57–60.

[2] Kevin Andryc, Murtaza Merchant, and Russell Tessier. 2013. FlexGrip: A soft GPGPU for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 230–237.

[3] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Boston, 163–174.

[4] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. 2015. Enabling GPGPU low-level hardware explorations with MIAOW: an open-source RTL implementation of a GPGPU. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 21.

[5] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. 2012. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *39th Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, United States, 49 – 60. https://doi.org/10.1109/ISCA.2012.6237005

[6] Jeff Bush, Philip Dexter, Timothy N Miller, and Aaron Carpenter. 2015. Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 173–182.

[7] George Chrysos. 2014. Intel® Xeon Phi™ Coprocessor-the Architecture. *Intel Whitepaper* (2014).

[8] Sylvain Collange. 2011. *Stack-less SIMT reconvergence at low cost.* Technical Report. HAL CCSD. https://hal.archives-ouvertes.fr/hal-00622654

[9] Sylvain Collange and Nicolas Brunie. 2017. *Path list traversal: a new class of SIMT flow tracking mechanisms.* Research Report RR-9073. Inria Rennes - Bretagne Atlantique. https://hal.inria.fr/hal-01533085

[10] Sylvain Collange, Marc Daumas, David Defour, and David Parello. 2010. Barra: a Parallel Functional Simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 351–360.

[11] Ahmed ElTantawy and Tor M Aamodt. 2016. MIMD Synchronization on SIMT Architectures. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[12] Xun Gong, Rafael Ubal, and David Kaeli. 2017. Multi2Sim Kepler: A detailed architectural GPU simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 269–278.

[13] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[14] Sajith Kalathingal, Sylvain Collange, Bharath Narasimha Swamy, and André Seznec. 2016. Dynamic Inter-Thread Vectorization Architecture: extracting DLP from TLP. In *IEEE International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*. https://hal.inria.fr/hal-01356202

[15] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. *Macsim: A CPU-GPU heterogeneous simulation framework user guide*. Georgia Institute of Technology.

[16] Jeffrey Kingyens and J Gregory Steffan. 2010. A GPU-inspired soft processor for high-throughput acceleration. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 1–8.

[17] Christos Kozyrakis and David Patterson. 2003. Overcoming the limitations of conventional vector processors. *ACM SIGARCH Computer Architecture News* 31, 2 (2003), 399–409.

[18] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2013. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 6.

[19] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanović. 2014. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*. IEEE, 199–202.

[20] John Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008), 39–55. https://doi.org/10.1109/MM.2008.31

[21] Samuel Liu, John Erik Lindholm, Ming Y Siu, Brett W Coon, and Stuart F Oberman. 2010. Operand collector architecture. US Patent 7,834,881. (nov 2010).

[22] Jiayuan Meng and Kevin Skadron. 2011. A reconfigurable simulator for large-scale heterogeneous multicore architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 119–120.

[23] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News* 38, 3 (2010), 235–246. https://doi.org/10.1145/1816038.1815992

[24] John Nickolls and William J. Dally. 2010. The GPU Computing Era. *IEEE Micro* 30 (March 2010), 56–69. Issue 2. http://dx.doi.org/10.1109/MM.2010.41

[25] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 117–126.

[26] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0.* Technical Report. DTIC Document.