Simty: a Synthesizable General-Purpose SIMT Processor

Caroline Collange

Simty: a Synthesizable General-Purpose SIMT Processor

Caroline Collange

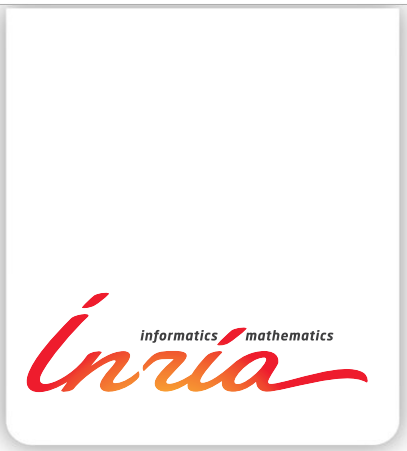
RESEARCH REPORT

N° 8944

August 2016

Project-Team PACAP

ISSN 0249-6399 ISRN INRIA/RR--8944--FR+ENG



Simty: a Synthesizable General-Purpose SIMT Processor

Caroline Collange

Project-Team PACAP

Research Report n° 8944 — August 2016 — 10 pages

Abstract: Simty is a massively multi-threaded processor core that dynamically assembles SIMD instructions from scalar multi-thread code. It runs the RISC-V (RV32-I) instruction set. Unlike existing SIMD or SIMT processors like GPUs, Simty takes binaries compiled for general-purpose processors without any instruction set extension or compiler changes. Simty is described in synthesizable RTL. A FPGA prototype validates its scaling up to 2048 threads per core with 32-wide SIMD units.

Key-words: SIMT, SIMD, FPGA, RISC-V

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Simty: un processeur SIMT généraliste synthétisable

Résumé : Nous présentons Simty, un processeur massivement multi-threadé qui assemble dynamiquement des instructions SIMD à partir de code scalaire multi-thread. Il exécute le jeu d'instructions RISC-V (RV32-I). Contrairement aux processeurs SIMD ou SIMT existants tels que les GPU, Simty accepte du code binaire compilé pour des processeurs généralistes sans nécessiter la moindre extension du jeu d'instructions ni modification du compilateur. Le processeur est décrit en RTL synthétisable. Un prototype sur FPGA valide le passage à l'échelle jusqu'à 64 warps ou 64 threads par warp.

Mots-clés : SIMT, SIMD, FPGA, RISC-V

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 [15]. It combines the simplicity of scalar code from the programmer's and compiler's perspective with the efficiency of SIMD execution units at the hardware level. However, current SIMT architectures demand specific instruction sets. In particular, they need specific branch instructions to manage thread divergence and convergence. Thus, SIMT GPUs have remained incompatible with traditional general-purpose CPU instruction sets.

We present Simty, an SIMT processor that lifts the instruction set incompatibility between CPUs and GPUs. By making CPUs and GPUs binary compatible, we enable commodity compilers, operating systems and programming languages to target GPU-like SIMT cores. Beside this simplification of the software layers, instruction set unification eases parallel program prototyping and debugging.

Inexpensive hardware prototyping solutions like FPGAs enable the academic community to develop hardware prototypes [12, 3, 5]. We have implemented Simty in synthesizable VHDL and synthesized it for FPGAs.

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.

2 Context

We introduce some background about the 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 [7]. 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 [10].

We have shown in prior work that the SIMT execution model could be generalized to general-purpose instruction sets, with no specific branch instructions, with two possible implementations [7, 4]. The simplest scheme requires no shared state between threads, relying only on the individual PC of each thread [7]. We then generalized it to support parallel execution of multiple paths, and proposed an efficient alternative representation of program counters using a sorted list [4]. Simty relies on the latter solution, applied to a single execution path.

As in conventional SIMT architectures, thread synchronization and scheduling policies 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. All inter-thread synchronization needs to use explicit instructions.

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 desig-

nate a hardware thread in the rest of the paper. 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 assigned statically a distinct SIMD lane. Its context resides fully in a private register file associated with its lane. No direct communication is possible between registers of different lanes.

As the programming model is multi-thread, each thread has its own architectural Program Counter (PC) from the software perspective, although this architectural PC may not be implemented as a separate physical register. We introduce an intermediate level between thread and warp that we call Instruction Stream (IS). The IS concept is also found under the terms warp-split [14], sub-warp, context [4] or fragment in the literature. Threads of an IS advance in lockstep and share a common PC in the same address space, referred to as Common Program Counter (CPC). In other words, each IS tracks one of the different paths followed by threads in the program. All threads in a given IS belong to the same warp. Thus, each warp contains from 1 to m ISs, with m being the thread count per warp. Thread in IS membership is represented in hardware by an m-bit mask for each IS. Mask bit i of IS j is set when thread i of the warp belongs to IS j. We can represent the state of a warp equivalently either as a vector of m PCs, or as a set of ISs identified by (CPC, mask) pairs.

3 Simty Microarchitecture

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

3.1 Design principles

As with standard SIMT architectures, 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 [17]. Simty currently supports the RV32I subset (general-purpose instructions and 32-bit integer arithmetic) and some privileged instructions for hardware thread management pending standardization.

Throughput-oriented architecture. Simty exploits thread-level parallelism across warps to hide execution latency, as well as data-level parallelism inside ISs to increase throughput. By contrast, we do 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. A similar limitation exists in industrial designs like Nvidia Fermi GPUs [15] or Intel Xeon Phi Knights Corner [6].

Genericity. 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, targeting design points comparable to a Xeon Phi core [6] or an Nvidia GPU SM [15]. 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 dependen-

cies. Execution hazards like resource conflicts are handled by partially replaying the conflicting instruction.

When an IS accesses a shared resource (e.g. a data cache bank), the arbitration logic gives access to an arbitrary *leader* thread of the IS. All other threads of the IS 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 one or more threads of the IS are neither leader nor followers, the IS is split in two: the leader and followers advance to the next instruction, while the other threads keep the same PC to have the instruction eventually replayed. This partial replay mechanism guarantees global forward progress, as each access satisfies at least the leader thread.

PC-based commit. Program counters are used to track the progress of threads in IS. 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.

Min(SP:PC) scheduling. In order to favor thread convergence, fetch priority is given to the IS in the deepest nested function call level, and in case of tie to the IS with the minimal PC [7].

3.2 Pipeline

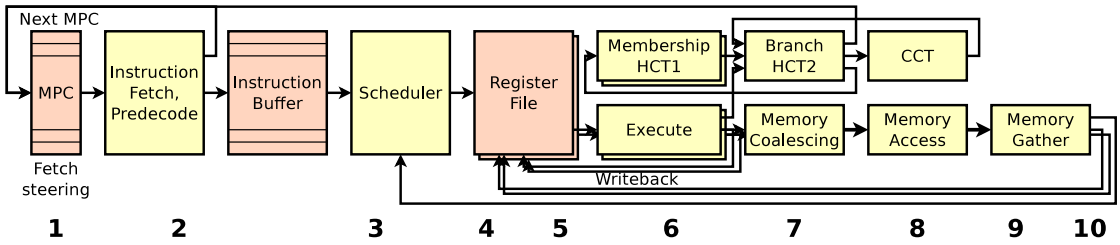


Figure 1: Simty pipeline.

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 current IS 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 an actual branch predictor may be used instead.

Fetch/predecode fetches an instruction word from the instruction cache or memory. A first predecode stage checks which operand come from registers. The RISC-V encoding enables straightforward predecode logic to collect all data needed for scoreboarding logic to track register dependencies between instructions. The predecoded instruction is placed in an instruction buffer containing one entry per warp.

Schedule issues instructions which operands are ready. A qualification step selects warps which next instruction can be executed. It considers both instruction dependencies and execution resource availability: bank conflicts in the register file are avoided at this stage. A selection step chooses one of the executable 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.

Operand collection consists in two stages. Each SIMD lane has its own register file (RF). Each RF is 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. Each bank has a single read port and a single write port. For instructions that take two source register operands, two read ports are emulated by alternate accesses over two cycles. The next instruction then needs to belong to a warp whose registers are in the other RF bank. This constraint is enforced by the scheduler stage. For instructions taking 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.

Execution. Execution units consist in one 32-bit ALU supporting the RV32I instruction set in each lane. All arithmetic instructions are currently executed in a single cycle, though we plan to integrate pipelined floating point units.

Membership. Concurrently to the execute stage, the IS thread membership mask is computed from the PC. For arithmetic instructions, the mask is the eventual non-speculative commit mask. For memory access instructions, it is a speculative mask that assumes no exception occurs. The leader thread is selected by a priority encoder from the first bit set in the membership mask. The membership unit also detects convergence between ISs. It is based on three tables HCT1, HCT2 and CCT in three successive pipeline staged, and will be described in more details in Section 3.3.

Writeback. The output of the execution stage is conditionally written back to each register file and forwarded through a bypass network. For lane i, writeback is performed when bit i of the membership mask is set. The mask can be all zeroes in the case of a branch misprediction. The bypass network is also controlled on a per-lane basis by the membership mask, 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 ISs upon a divergent branch instruction. It takes as inputs a condition vector c from the ALUs and the membership mask m from the membership unit. For a relative conditional branch, the scalar destination address PC_d is computed from the PC and the instruction immediate. Two ISs are created. One IS tracks threads that follow the branch $(\mathrm{PC}_d, c \wedge m)$, and a second IS tracks threads that fall through the next instruction $(\mathrm{PC} + 4, \bar{c} \wedge m)$. Empty ISs are subsequently discarded. An indirect branch which computed address vector contains multiple different targets may require as many ISs as threads in a warp to be created. In this case, branches to each unique target are serialized in order to have at most two ISs out of the branch unit. Following the leader-follower strategy, threads that have the same target PC_l as the leader thread have their state updated, while the other threads keep the same PC and have the indirect branch instruction replayed.

Memory arbitration and coalescing The memory arbiter coordinates concurrent accesses to a common memory from the threads of an IS. It is optimized for two common cases: when threads of a warp access consecutive words from the same cache block, and 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 IS and have their instruction replayed.

Memory gather distributes data from the cache block read back from memory to the threads of the IS. In particular, it can broadcast the word accessed by the leader thread to the other threads.

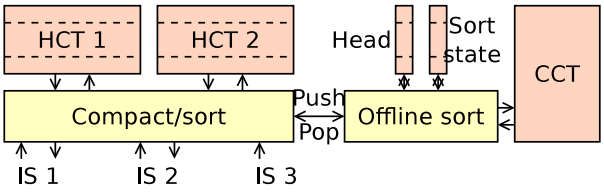


Figure 2: IS tracking unit

3.3 Instruction stream tracking

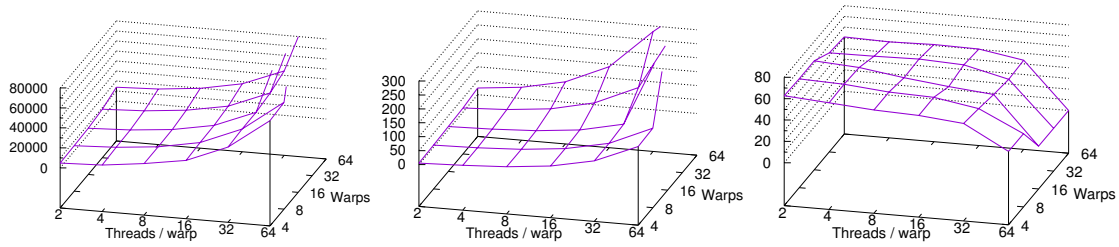
The IS tracking unit is responsible for three functions: 1. check thread membership within an IS to compute the validity mask of an instruction, 2. merge the ISs that have identical PC to let threads converge, and 3. select the active IS that is followed by the front-end.

A conceptually simple but relatively hardware-costly option to manage ISs is to assign one PC register per thread as in a conventional multi-thread processor. Functions 1 and 2 of assembling the IS within each warp are done by comparing thread PCs with each other each cycle. Function 3 of selecting the active IS is performed through an arbitration policy between all threads of a warp [7]. Simty follows a more efficient but functionally equivalent approach that leverages the alternative 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 IS and reading the associated mask. Functions 2 and 3 are implemented based on a list of ISs sorted by priority. Priorities being based on CPC values, IS convergence may only happen between the active IS and the second IS by priority order.

IS entries are kept sorted by function call level and PC. To manage efficiently large numbers of ISs in hardware, we isolate the two head ISs, sorted on-line, from all other inactive ISs, sorted off-line by a state machine. The head ISs 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 ISs (Figure 2) [4].

The sort/compact unit in Figure 2 gathers ISs 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 IS. It then merges entries that have the same PC and same function call level, discards ISs with empty masks and sorts the resulting ISs 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 IS 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 IS remain, the third one is pushed to the head of the CCT.

The CCT size is adjusted for the worst case of one IS per thread. It contains a stack of ISs for each warp, which head is followed by a pointer. The offline sorting unit is a 3-state state machine with one pointer per warp. It carries out an iterative insertion sort opportunistically when the CCT ports are available. Each pointer walk 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 converge toward a fully sorted IS list in m^2 steps at most with m-thread warps. The performance of this low-overhead implementation is adequate for the purpose of sorting ISs, as CCT sorting only occurs in the case of unstructured control flow.



(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.

4 FPGA implementation

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 [16]. 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. The total number of threads per core needed to hide latency is proportional to the latency × throughput product following Little's law [13].

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 that, routing congestion causes a notable cycle time increase and the extra area gains against multiple Simty cores are low.

5 Related work

Several synthesizable parallel processors have been recently released in the academic community.
Vector processors HWACHA is a synthesizable vector processors running vector instruction set extensions to RISC-V [12], and VectorBlox MXP is a vector soft-core for FPGA [16].
GPU-based architectures Kingyens and Steffan propose a processor compatible with the fragment shader unit of the ATI R500 GPU architecture [11]. MIAOW is a synthesizable GPU based on the AMD GCN architecture [3]. Flexgrip [2] is a GPU based on the Nvidia Tesla architecture that follows the pipeline of the Barra simulator [9].
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 [5]. All these processors are based either on explicit vector or SIMD instruction sets

with per-lane predication, either on SIMT instruction sets with specific branch instructions. Simty is the first SIMT processor supporting a scalar general-purpose instruction set.

6 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. 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. The choice of the RISC-V instruction set greatly eases the latter step through reliance on existing toolchains like gcc and LLVM. 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. Guppy: A GPU-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 57–60. IEEE, 2012.

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

[3] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. 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):21, 2015.

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

[5] Jeff Bush, Philip Dexter, Timothy N Miller, and Aaron Carpenter. 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*, pages 173–182. IEEE, 2015.

[6] George Chrysos. Intel® Xeon Phi™ coprocessor-the architecture. *Intel Whitepaper*, 2014.

[7] Caroline Collange. Stack-less SIMT reconvergence at low cost. Technical report, HAL CCSD, September 2011.

[8] Caroline Collange. Un processeur SIMT généraliste synthétisable. In *Conférence d'informatique en Parallélisme, Architecture et Système (Compas)*, Lorient, France, July 2016.

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

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

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

[12] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanović. 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*, pages 199–202. IEEE, 2014.

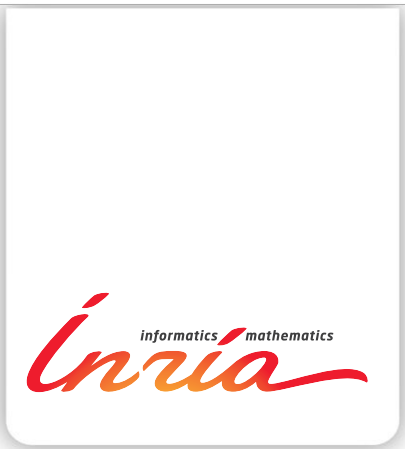
[13] John D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

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

[15] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

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

[17] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0. Technical report, DTIC Document, 2014.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399