Jan Lucas, Michael Andersch, Maurisio Álvarez-Mesa,
Ben Juurlink

Spatio-temporal SIMT and scalarization for improving GPU efficiency

Article, Postprint version
This version is available at http://dx.doi.org/10.14279/depositonce-6262

Suggested Citation
Lucas, J.; Andersch, M.; Álvarez-Mesa, M.; Juurlink, B.: Spatio-temporal SIMT and scalarization for improving GPU efficiency. - In: ACM Transactions on Architecture and Code Optimization:ACM TACO. - ISSN: 1544-3973 (online), 1544-3566 (print). - 12 (2015), 3. - Art.Nr. 32. - DOI: 10.1145/2811402. *(Postprint version is cited.)*

Terms of Use
© ACM, 2015. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Transactions on Architecture and Code Optimization (TACO), {VOL 12, ISS 3, (2015)} https://dl.acm.org/citation.cfm?doid=2818748.2811402.

WISSEN IM ZENTRUM UNIVERSITÄTSBIBLIOTHEK
Technische Universität Berlin

Spatio-Temporal SIMT and Scalarization for Improving GPU Efficiency

Jan Lucas, Technische Universität Berlin
Michael Andersch, Technische Universität Berlin
Mauricio Alvarez-Mesa, Technische Universität Berlin
Ben Juurlink, Technische Universität Berlin

Temporal SIMT (TSIMT) has been suggested as an alternative to conventional (spatial) SIMT for improving GPU performance on branch intensive code. Although TSIMT has been briefly mentioned before, it was not evaluated. Therefor we present a complete design and evaluation of TSIMT GPUs, along with the inclusion of scalarization and a combination of temporal and spatial SIMT, named Spatio-Temporal SIMT (STSIMT). Simulations show that TSIMT alone results in a performance reduction but a combination of Scalarization and STSIMT yields a mean performance enhancement of 19.6% and improve the energy-delay-product by 26.2% compared to SIMT.

CCS Concepts: •**Computer systems organization** → **Parallel architectures; Single instruction, multiple data;**

Additional Key Words and Phrases: GPUs, temporal SIMT, branch divergence, scalarization

1. INTRODUCTION

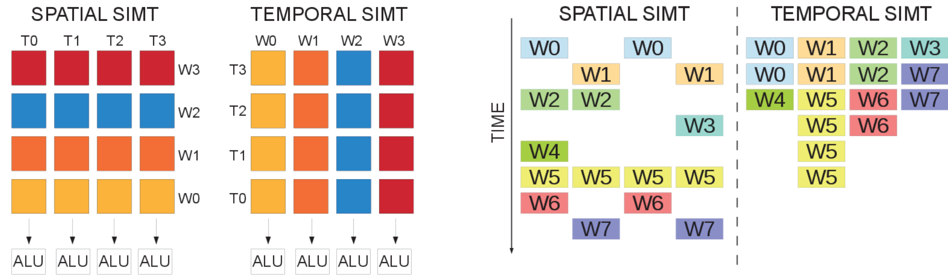
GPUs have pervaded computing systems as massively parallel accelerators, and integrated CPU-GPU systems along with the heterogeneous codes written for them have become relevant for both industry and academia. Programming interfaces such as CUDA and OpenCL have become commonplace, and GPU architectures have evolved specifically for general-purpose computing on GPUs (GPGPU).

One of the main design decisions in GPUs has been the ganging of execution threads into batches named *warps*, and in particular the sequencing of these warps onto an array of execution units in a single-instruction multiple-data (SIMD) fashion. This way of performing SIMD execution is supported by a hardware stack to manage divergent thread control flow, and the resulting execution paradigm has become widely known as single-instruction multiple-thread (SIMT) execution. While SIMT amortizes control hardware over many execution units, research over the past years has shown that this approach yields poor SIMD utilization under control divergence, i.e. when some or most of the threads in a warp are inactive, thus not performing any useful work [Fung et al. 2007].

In this paper, we evaluate an alternative approach to the classical *spatial* SIMT. It has been proposed to base GPU cores on *temporal* SIMT (TSIMT), which is a different way of mapping individual threads to execution units [Keckler et al. 2011]. Figure 1a compares the spatial and temporal approaches intuitively. In the figure it is shown how four warps (W0 - W3) consisting of 4 threads each are executed, once for spatial and once for temporal SIMT. On the left-hand side, spatial SIMT operates in a classical SIMD fashion: All threads in a warp execute the same instruction and thus the instruction word is broadcast to all execution units every clock cycle. On the right-hand side, temporal SIMT is shown as a *transpose* of the spatial SIMT mapping of warps and threads to datapath units. Here, each warp is executed on only one specific lane, and the threads corresponding to the warp are sequenced onto the scalar execution

---

This project received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the LPGPU Project (www.lpgpu.org), grant agreement n° 288653.
Authors's address: J. Lucas and M. Andersch and M. Alvarez-Mesa and B. Juurlink, Technische Universität Berlin, Sekretariat EN 12, Einsteinufer 17, 10587 Berlin,Germany



(a) Mapping of warps and threads to execution units in both spatial and temporal SIMT.

(b) Comparison of conventional and TSIMT execution for control-divergent code.

Fig. 1: Conventional SIMT vs. TSIMT. Different colors are used to symbolize different warps on the core. In this comparison, warps are assumed to be 4-wide and cores are assumed to have 4 execution units / 4 TSIMT lanes.

unit in that lane cycle by cycle. As such, TSIMT is reminiscent of a single lane vector processor [Lee et al. 2013].

Beside divergent branches another situation where SIMT architectures waste execution cycles is when all threads in a warp not only execute the same instruction, but do so on identical data as well. In this case, it would be more efficient to execute the instruction only once instead of once per thread, thereby freeing execution resources. By doing so the SIMD instruction is turned into a scalar instruction, and therefore this technique is known as *scalarization*. Besides releasing execution resources, scalarization also reduces the pressure on the register file. This requires, however, that scalar values are stored in such a way that all threads of a warp can access the value. In a conventional GPU this requires additional broadcast networks and often specialized dedicated scalar register files as well. In TSIMT GPUs, on the other hand, tiny modifications to the existing register file suffice and the register file space can be used for both scalar and vector values.

While the basic idea of TSIMT has been sketched in a patent [Krashinsky 2011] and has been briefly described as an additional idea in two papers [Lee et al. 2013; Keckler et al. 2011], a microarchitectural implementation and detailed performance analysis have not been presented before. To fill this gap, we introduce a GPU design based on TSIMT and perform a detailed analysis of its advantages and shortcomings.

Although TSIMT can improve the performance of control divergent workloads, it can, as we will show in the analysis of simulation results, suffer from load balancing issues. As a way to have both the control divergence mitigation of TSIMT and improve the load balancing, we also propose and evaluate an architecture that combines the traditional spatial SIMT with TSIMT, and we refer to it as spatio-temporal SIMT.

More concretely, the contributions of our work can be summarized as follows:

— For the first time, we present a detailed microarchitecture design and implementation of temporal SIMT for GPGPUs.
— We evaluate the TSIMT approach in detail and show that it is able to provide large performance benefits for control divergent GPU codes in µ-benchmarks, but suffers from significant *load balancing* problems in real applications.
— We propose two optimizations to the basic TSIMT microarchitecture that reduce the load balancing problem.

— We introduce and evaluate an architecture that combines spatial and temporal SIMT and demonstrate that it exhibits performance improvements compared to both spatial and temporal SIMT (STSIMT).
— We show how scalarization can be integrated in the proposed TSIMT architecture in a way that requires less hardware than its integration in conventional SIMT GPUs.
— Finally, we evaluate power, energy consumption and energy efficiency of the architectures presented and show that STSIMT with scalarization improves the energy-delay product (EDP) by more than 25% compared to the SIMT baseline.

This paper is organized as follows. Section 2 describes related work, both on temporal SIMT and on handling control divergence on GPUs. After that, Section 3.8 introduces and describes our TSIMT-based GPU design. Then, Section 4 describes Scalarization and its integeration in TSIMT architectures. The performance evaluation results are discussed in Section 5. Finally, Section 6 concludes this paper.

2. RELATED WORK

Related work can be grouped into three categories: First, work describing temporal SIMT, second, studies focusing on efficient execution of branch divergent codes, and last, work concerned with the integration of instruction scalarization.

Temporal SIMT. The general idea of temporal SIMT execution is described in an NVIDIA patent by [Krashinsky 2011], but the details provided are insufficient to derive an implementation and furthermore no performance evaluation is included. In the academic world, TSIMT has also been mentioned by [Keckler et al. 2011] in a paper that describes that moving data across the chip is more energy-consuming than actual computation. This paper introduces the *Echelon* GPU architecture that offers TSIMT execution as well as many other features. The authors mention the potential benefits of TSIMT for branch divergent applications, but neither, presents an implementation nor an evaluation of Echelon.

Branch Divergence. A large body of work has been performed on how to improve GPU performance when there is control divergence. Many techniques such as *Dynamic Warp Formation* [Fung et al. 2007], *Thread Block Compaction* [Fung and Aamodt 2011] and *Large Warp Microarchitecture* [Narasiman et al. 2011] reorder threads from multiple warps into fewer warps with more active threads per warp. All these techniques keep the spatial SIMD property: All lanes execute the same instruction at the same time, but differ in when and how threads are reordered. Furthermore, because these techniques reorder threads between warps, memory divergence can increase and correctness problems for applications that rely on warp-level synchronization can arise.

Another, related technique called *Simultaneous Branch and Warp Interweaving* was introduced by [Brunie et al. 2012]. They introduced enhancements to the GPU's microarchitecture to *a)* co-issue instructions from two different branch paths and *b)* co-issue instructions from different warps to the same SIMD unit. Interestingly, this architecture shares a property with TSIMT, namely that the different lanes do not need to share the same instruction. In SBWI, however, only two different instructions can be executed at the same time, while in TSIMT each lane can execute a different instruction.

A common disadvantage of the techniques described above is that they can only improve performance when the active mask meets certain conditions. One of the reasons for these conditions is that the individual contexts of the threads that run on the GPU are stored in a specific part of the GPU's register file [Jayasena et al. 2004]. As a result, it is generally not possible to freely swizzle and re-group threads for execution. TSIMT

takes a different approach that avoids this problem almost entirely at the expense of higher issue throughput requirements.

[Vaidya et al. 2013] proposed an architecture in which 16-wide SIMD instructions are executed over multiple cycles on 4-wide SIMD units. Two techniques are proposed to accelerate execution when only a subset of threads is active: Basic Cycle Compression (BCC), where SIMD subwords are skipped if no thread is active, and a more costly but also more powerful technique called Swizzled Cycle Compression (SCC) that employs crossbars to permute the operands prior to compaction to enable a more efficient compaction.

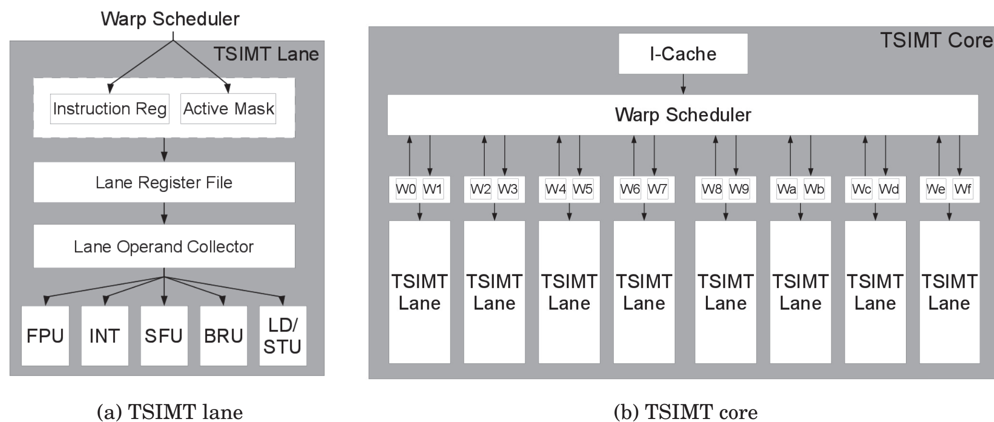
[Lee et al. 2011] group different possibilities for data parallel accelerators into five different groups: MIMD, Vector-SIMD, Subword-SIMD, SIMT and Vector-Thread (VT). TSIMT can be seen as another variant. The programming model is MIMD, but the execution units are similar to density-time vector lanes [Smith et al. 2000]. The lanes share the same instruction fetch and decode frontend but are not bundled in groups that execute the same instructions at the same time as in the architectures classified as Vector-SIMD and SIMT. On the other hand, the TSIMT-lanes are also not as independent as the lanes in the VT architecture. The control logic in each lane is limited to a register storing a single instruction, control logic for the sequential register fetch and the density-time execution of the stored instruction, while in VT each lane is able to fetch its own instructions and can use a shared control processor.

Scalarization. [Lee et al. 2013] discussed scalarization as well mentioned TSIMT. They developed a scalarizing compiler for SIMT architectures and evaluated architecture independent metrics such as percentage of scalar instructions but did not evaluate performance. They also described potential GPU architectures exploiting scalarized code, such as SIMT datapaths with an additional scalar unit, SIMT datapaths with scalars in a single SIMD lane, and TSIMT datapaths. The authors recognized that scalarization and TSIMT match each other well, but, as mentioned before, no actual implementation or evaluation is provided.

A similar analysis has been performed by [Collange 2011]. Like the previous study, Collange implemented compiler support for scalarization, but in a just-in-time form using GPUOcelot [Diamos et al. 2010]. The author presented the scalarization metrics of the resulting code, i.e. dynamic instruction counts of scalar and vector instructions, but no performance analysis is presented.

[Coutinho et al. 2011] transform GPU kernels to a SSA-based intermediate representation called µ-SIMD, afterward they analyze the divergence of the program in its µ-SIMD representation. They recognized that instructions can sometimes be scalarized, even during divergent control flow, if their output registers are not alive at the immediate post-dominator of the potentially divergence branch. However, they do not perform register allocation and thus only report how many of the register in SSA form could be scalarized, but do not report how many registers of which type are actually needed after register allocation. The scalarization algorithm presented in this paper directly works on the representation of GPU kernels used by NVIDIA and does not require a transformation to a SSA-based representation and back. In this paper register allocation is performed on the scalarized code and we discuss the modifications to register allocation that are required for scalarized code.

[Xiang et al. 2013] studied the problem of scalarization as well, but consider uniform values across warps as well. They introduce a hardware scalarization mechanism for intra-warp uniform instructions. This mechanism, however, does not enable higher occupancy. A scalar register file based architecture is also presented, but also does not enable higher occupancy but only reduces energy consumption. Scalars are processed using the same 8 element wide SIMD execution units as vector instructions, but scalar



(a) TSIMT lane

(b) TSIMT core

Fig. 2: Block diagrams visualizing the organization of a TSIMT lane and the construction of TSIMT SIMT cores from TSIMT lanes. In the figure, lanes are one-wide, and cores are constructed from 8 lanes and have an overall occupancy of 16 warps at maximum.

operations are finished in 1 cycle instead of 4. While this improves performance, it leaves 7 out of 8 ALUs unused during the execution of scalar instructions.

Finally, [Kim et al. 2013] studied the relationship of the different values processed by a warp. They named this value structure and identified several important classes such as uniform vector where all elements contain the same value and affine vectors where all elements share a simple affine relationship to *block-* and *threadids*. The authors focus on an architecture named fine-grained SIMT (FG-SIMT) that is closer to purely compute-focused SIMT accelerators than to GPUs. They propose microarchitectural mechanisms to exploit uniform and affine values, including an affine register file as well as dedicated affine execution units. The proposed mechanisms were evaluated on a conventional NVIDIA-like GPU architecture, but the authors state explicitly that the lack of public knowledge about GPGPU architectures prevents them from performing more than a preliminary design space exploration. Instead, they focused on an evaluation using a VLSI implementation of an FG-SIMT design.

3. A TEMPORAL SIMT GPU ARCHITECTURE

These sections describes the proposed TSIMT µ-architectures. As TSIMT-based GPUs are still GPUs, most parts of the µ-architecture are identical to conventional GPUs. A good overview over conventional GPU micro-architecture is provided by [Bakhoda et al. 2009]. Everything outside the GPU cores is unchanged: interconnect, memory controllers, PCIe interfaces and CTA scheduler. Many structures inside the GPU core are also unchanged: instruction fetch and decode, caches, scoreboards and the reconvergence stack. In the following sections we will thus concentrate on the element that are modified in TSIMT GPUs compared to conventional GPUs.

3.1. TSIMT Cores and Lanes

The basic building block of TSIMT GPU cores is the *TSIMT lane*. A block diagram is depicted in Figure 2a. Such a lane consists of four key components: An instruction register able to store a single warp instruction, a slice of the core's register file, an operand collector, private to the lane, and one-thread-wide execution resources for in-

teger, floating point, and memory instructions. Operationally, a TSIMT lane receives a warp instruction along with the corresponding thread active mask from the core's warp scheduler and stores them into dedicated instruction and mask registers. The instruction register is used to hold the instruction in place while the lane back-end sequences through the warp's threads, thereby decoupling the lane from the scheduler while the lane is executing.

Figure 2b indicates how TSIMT lanes are used to construct cores with arbitrary throughput. In a TSIMT core, multiple lanes operate independently and in parallel processing the instruction words stored in their instruction registers. The overall number of threads and warps held in the core are evenly divided over all TSIMT lanes, e.g. in a core with 8 lanes holding 64 warps at most, every lane will statically hold 8 of the warps. Warps cannot switch between lanes as the thread context associated with a warp is stored in the register file within the warp's lane. This subdivides the core's warp pool into separate pools for every lane. In the core's front-end, an instruction fetch unit accesses the instruction cache and fetches as well as decodes instructions into an instruction buffer (IB). The instructions buffer uses dedicated slots for each warp. A single warp scheduler (WS) for the whole core utilizes a scoreboard to monitor which instructions in the IBs have their dependencies fulfilled and are ready to issue. The WS also monitors when TSIMT lanes complete the execution of an instruction and, therefore, require a new instruction word to be sent to the lane's instruction register.

The execution resources are warp-wide (i.e. 32-thread-wide) in conventional spatial SIMT. In a TSIMT execution architecture, on the other hand, each TSIMT lane is one-wide, meaning that one thread instruction can be executed every cycle. There is, however, an entire spectrum of GPU architectures possible in between spatial and temporal SIMT, that combine both execution paradigms. Such *spatio-temporal SIMT* (STSIMT) architectures operate like TSIMT architectures, but each lane contains sufficient execution resources to execute instructions of multiple threads in a single clock cycle. For example, with a warp size of 32, one can construct a 4-way-spatial 8-way-temporal SIMT architecture where each TSIMT lane contains 4-wide execution resources. In this microarchitecture, a TSIMT lane sequences the 32-wide warp onto its 4-wide execution resources in 8 consecutive clock cycles. The performance trade-offs of STSIMT as an evolution of basic TSIMT are discussed in Section 5.7.

STSIMT is not completely new, single lane implementations without compaction have been used in existing GPU architectures such as NVIDIA Tesla [Lindholm et al. 2008a]. Tesla uses STSIMT to construct SIMT cores with lower throughput while maintaining a large warp size. For example, in Tesla, 32-wide warps are sequenced onto an 8-wide SIMD datapath over 4 clock cycles. One key difference to STSIMT as proposed here is that existing approaches do not implement compaction and are therefore unable to provide any benefit for the execution of divergent applications. As in both TSIMT and STSIMT lanes are usually busy for more than a single cycle, multiple lanes can share a single frontend for instruction issue and decode. Tesla, however, does not exploit this property of STSIMT. We use a baseline architecture similar to Tesla for the experimental evaluation in Section 5.

3.2. Control Divergence

The TSIMT concept can efficiently provide large performance benefits when executing control-divergent codes. Consider Figure 1b for example. It shows how a conventional GPU core with 4 execution units and a TSIMT-based core with 4 lanes execute instructions from 8 different warps, where each warp is coded with a different color. For the sake of conciseness, warps are assumed to consist of 4 threads each. The warp instructions executed are control divergent, i.e., some threads do not participate in the

execution and are inactive. For explanatory purposes, each warp instruction is shown with a different active mask, although such situations are rare.

The figure shows that the conventional GPU architecture always requires one clock cycle on all 4 ALUs to execute a warp instruction, regardless of the instruction's active mask. Threads that are switched off and the ALUs are left unused. In this example the conventional GPU completes 15 thread instructions in 8 clock cycles for an overall IPC of $16/8 = 2.0$. On the TSIMT-based core, on the other hand, *compaction* is performed: Warp instructions with some inactive threads are executed in fewer cycles than the warp width. In fact they are executed in the minimum possible number of clock cycles, e.g. a warp with 2 active threads utilizes one ALU for exactly 2 clock cycles. As such, no execution resources are wasted. Overall, in this hypothetical example, the TSIMT core completes the 15 thread instructions in 6 clock cycles, corresponding an overall IPC of $16/6 = 2.67$. We remark that the IPC in the TSIMT case would approach the ideal IPC of 4 if more work was available, as TSIMT lanes 0 (on the left) and 3 (on the right) are ready to receive new instructions after 3 clock cycles.

In spatio-temporal SIMT architectures, some compaction ability of TSIMT is lost. As an example, consider an STSIMT architecture where each TSIMT lane contains 4-wide execution resources. In this case, compaction only works for warp active masks with aligned bundles of 4 consecutive inactive threads (e.g. 111100001111...). Warp instructions with active masks that switch more often between active and inactive threads (e.g. 101010...), irregular (e.g. 100111010...) or unaligned (e.g. 1000011110000...) cannot be compacted. Therefore, STSIMT architectures lose compaction ability compared to TSIMT architectures, but exhibit larger latency hiding ability within each lane as the core's warp pool is partitioned over fewer lanes if the overall execution width of the core remains constant.

3.3. Instruction Issue

In essence, TSIMT effectively *trades instruction issue bandwidth for instruction execution bandwidth* when executing branch divergent code. In conventional GPUs, the instruction issue bandwidth is coupled with the execution bandwidth. If a SIMT GPU needs four cycles to execute a warp, it will only need to supply one new instruction every four cycles. If some improvement made it possible to execute instructions with a smaller number of active threads faster in the execution units, speed would not improve, because instructions could not be issued faster. In this work we assume a front end that is able to issue one instruction per clock cycle. This provides a 4 times higher instruction issue bandwidth than needed for perfectly convergent code. The SIMT GPU can use this additional issue bandwidth to execute instructions on SPs, SFUs and LDST units concurrently to exploit ILP. The total number of issued instructions is always the same in SIMT, TSIMT and STSIMT, only the peak issue rate increases in TSIMT. Each warp instruction needs to be issued only once, no matter how many threads are active. Executing the operation over multiple cycles on all active threads of the warp does not require additional issue cycles but is performed locally in the TSIMT lane. In conventional SIMT threads from each warp are executed in lock-step and explicit synchronization can be omitted when data is exchanged between threads of the same warp. Contrary to some other techniques for improving the performance of divergent applications such as thread block compaction [Fung and Aamodt 2011], where threads can get reassigned to a different hardware warp and programmers cannot expect that threads from same initial warp keep executing in lockstep, in TSIMT lockstep execution of threads of the same warp is preserved and no modifications are necessary to kernels.

3.4. Memory Access Coalescing

With respect to *memory coalescing*, a TSIMT architecture is largely unchanged from a conventional GPU architecture. When a lane executes a warp-wide memory instruction, it generates one thread memory address per clock until all memory addresses requested by the warp instruction are known. Then, the entire bundle is passed onto the coalescing hardware in the load-store unit that reduces the requests to the minimum number of memory transactions. Finally, the transactions access the L1 cache and, potentially, the lower levels of the memory hierarchy. If the L1 cache or the load-store unit are stalled, then stall signals are propagated back to the warp scheduler which will therefore be unable to issue memory instructions until the stall is resolved. Using instruction replay the coalescing hardware can be simplified by reusing major parts of the core [Diamond et al. 2014], which is also currently used in NVIDIA GPUs [Ziegler 2011]. Our simulator is based on GPGPU-Sim which, however, currently does not model instruction replay. For this reason we decided to model all architectures without instruction replay.

3.5. Shared Memory

While global memory request coalescing in TSIMT and SIMT is similar, shared memory instructions are handled differently by TSIMT GPUs. In a conventional GPU, threads within a warp must access different memory banks to prevent serialization due to bank conflicts. In TSIMT, on the other hand, threads within the same warp never produce bank conflicts as they are executed in consecutive cycles. Instead, warps on different lanes that try to access shared memory simultaneously may produce inter-warp bank-conflicts. Despite these differences the hardware needed for the shared memory is almost identical for TSIMT and SIMT. Each lane sends up to one address to the shared memory, the addresses are checked for conflicts and a crossbar connects several SRAM banks to the input and output ports of the lanes. In TSIMT some ports of the shared memory are often not used because the lane connected to that port is currently executing an arithmetic instruction, this can reduce the number of shared memory bank conflicts that occur. The lockstep execution of the warps in SIMT, on other hand, often causes all lanes to execute a shared memory instruction at the same time, which makes conflicting accesses more likely.

3.6. Latency Hiding

In conventional GPU architectures, the multitude of active warps residing on a core or warp scheduler is used to hide the latency of currently executing instructions. A rather large number of warps is required to hide the latency of deep pipelines or long-latency memory operations [Wong et al. 2010]. In the TSIMT architecture described above, the warps as well as the execution back-end of the core are partitioned into a number of TSIMT lanes. This means that while the pipeline depth and memory latency remain unchanged, the number of warps available for latency hiding within each lane decreases considerably (i.e. by a factor equal to the number of lanes per core). This does not necessarily impact performance negatively though, as a single instruction contributes significantly more latency hiding ability in a TSIMT GPU core than to a conventional one: In a conventional GPU, having one independent warp instruction available for execution corresponds to one clock cycle of latency hiding. In the TSIMT core, on the other hand, a single independent warp instruction keeps a TSIMT lane busy for up to 32 clock cycles, depending on its active mask.

3.7. Register File

TSIMT register files use the same basic design idea as the register files of conventional GPUs: Instead of using costly multiported memories, multiple single ported SRAM banks are used [Lindholm et al. 2008b; Gebhart et al. 2012]. These register banks are connected using a crossbar to a operand collector. The operand collector fetches the operants over multiple cycles. In TSIMT instead of using a single very wide register file with one 32-bit entry for each thread of a warp, each lane implements one small 32-bit wide register file. In a conventional SIMT register file only a whole warp wide register entry can be addressed. Even if just a single thread is active and we are only interested in the operand for that thread, a whole 1024-bit wide entry (32-bits for each of the 32 threads of warp) would be fetched. In TSIMT only the operands of active threads are fetched. Using individual register file lanes rather than a single monolithic register file gives more flexibility with the placement of the components and helps to keep distances between register file and execution units small. However, it also divides the register file into multiple parts. The register file in each lane stores the registers for warps assigned to that lane. Other lanes cannot execute instructions from these warps as there is no connection between the lanes that would allow operands from register file of lane to be passed to an execution unit located in a different lane. The register file of each lane provides only enough bandwidth for executing instructions at full speed in one lane. For this reason adding additional connections to allow the execution of warps on other lanes would not increase performance, as the execution units would stall because one register file lane cannot supply operands fast enough to keep multiple lanes running at full speed. All registers required by one warp must fit into a single lane it is not possible to store some registers of the warp in one lane and some registers of the warp in the register file of a different lane. No additional warp can be allocated, if all lanes together have sufficient free register resources for one or multiple additional warps, but no lane alone can provide enough space for an additional warp.

The operand collector reads and writes the operands in multiple cycles from multiple banks. Reads and writes of multiple instructions are overlapped. In TSIMT we can use this structure to fetch the operands for the different active threads. This multi-cycle operand fetch avoids the need for an area and power-hungry multiported register file. However, depending on the register allocation and divergence pattern, load balancing problems between the register banks can appear and cause stalls.

Two optimizations related to the register file allocation are explored in this paper: First, register resources are freed on warp exit instead of block exit and second, partially filled warps only allocate registers for each active thread instead of the entire warp. We use TSIMT+ to refer to an optimized version of TSIMT that implements these two optimizations.

The first optimization makes it possible to launch new thread blocks sooner: In the conventional GPU, as modeled by GPGPU-Sim, register resources are managed at the thread block level. Registers allocated to a warp can only be reused after the entire thread block has finished executing. This potentially leaves many register resources unused for extended periods of time. With the optimization, warp resources are freed as soon as a warp finishes execution. Consequently, new blocks are launched as soon as sufficient resources are available. A similar approach has been described for conventional SIMT GPUs by [Xiang et al. 2014]. Other than their solution, however, our solution only permits the launch of a new block if sufficient resources are available to launch a full thread block.

The second optimization can increase occupancy if thread block sizes are not divisible by the warp size. For example, if a thread block size of 112 is requested in a regular

GPU Register File using a single 256-bit wide lane

Component	Width	Size/Ports	Area (mm^2)	Number	Total Area (mm^2)
SRAM	256-bit	8192 Byte	0.0296	8	0.2365
Crossbar	256-bit	8x8	2.0302	1	2.0302
Total					2.2667

GPU Register File using 8 independent 32-bit wide lanes

Component	Width	Size/Ports	Area (mm^2)	Number	Total Area (mm^2)
SRAM	32-Bit	1024 Byte	0.0036	64	0.2279
Crossbar	32-bit	8x8	0.0333	8	0.2664
Total					0.4943

Table I: Area estimates for different possible register file implementations at 40nm

GPU, registers for 128 threads are allocated. Our optimization allocates only the registers for 112 threads, i.e. the restriction to allocate registers with warp size granularity is removed. In the case of 112 threads, three full warps of 32 threads and one half filled warp would be allocated. When the next block is allocated, another half filled warp is allocated to the lane where the half filled warp from the first block resides. This optimization is not possible in the register files of conventional GPUs as it is enabled by the ability of TSIMT register files to address registers with a per-thread granularity. The register files of conventional GPUs can only be addressed with a per-warp granularity, because of this limitation partially filled warps leave some register file space unusable in conventional GPUs.

3.8. Area

We expect that the die area required by a TSIMT GPU should be close to the area required by a SIMT GPU with an otherwise identical configuration. As already explained in Section , most structures are unchanged from a SIMT GPU. On die storage is almost unchanged: Only a few additional bits for the instruction register and the storage of the active mask are required per TSIMT lane.

The number of bits in the register file stays the same, however, they are distributed over a higher number of narrower banks. [Fung et al. 2009] estimated an 18.7% increase in register file area, but we think this is an overestimate. We used CACTI 6.5 to estimate the area of the SRAM banks and crossbars used in the GPU register file. We tested two potential designs: First, a monolithic 256-bit wide register file and second, a register file with 8 narrow 32-bit wide lanes with independent decoders and crossbars. We show the results of this estimation in Table I. The second option is more flexible and small at the same time. Even the SRAM banks are slightly smaller, but especially the crossbar is reduced in area: the narrow input and output ports greatly reduce the distances between the different ports. Splitting the register file into lanes, results in a interleaved implementation of the register file that reduces the length of required wiring. The additional flexibility offered by the second design is required for TSIMT, but the second design can also be used to implement the register file of a conventional GPU. As a conventional GPU does not require all the flexibility offered by this design, slightly less area is likely needed as some parts of the address decoders could be shared by multiple lanes.

4. SCALARIZATION

Previous work [Xiang et al. 2013; Collange 2011; Lee et al. 2013] has shown that in SIMT architectures several threads often redundantly perform the same calculation on the same vector operands. Such situations are common because in many cases it is easier and faster to recalculate results in different threads than to calculate the results only once and to broadcast them to all threads. Redundant calculation not only

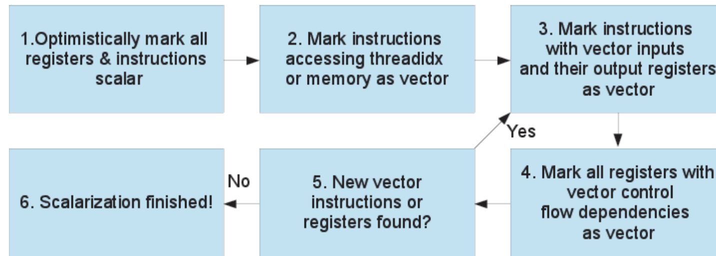


Fig. 3: Scalarization Algorithm

wastes execution throughput, it also wastes storage as well as energy since copies of the calculated values have to be stored for every thread.

Redundant calculations can be removed by applying a technique called scalarization [Xiang et al. 2013; Lee et al. 2013]. In this technique a static compiler algorithm is used to identify instructions that always use the same operands in all active threads of a warp. Likewise, it also identifies registers that always store the same value in all threads of a warp. These instructions are then only executed once per warp instead of once per thread, also the identified registers are stored once per warp instead of once per thread. The hardware cost of integrating scalarization in the proposed TSIMT architectures are much lower compared to regular GPUs, because since most of the execution resources can be reused for the scalar and vector datapaths. Furthermore, we improve upon the scalarization algorithm proposed in [Lee et al. 2013] by allowing scalarization, even with divergent control flow.

4.1. Hardware Support for Scalarization

In conventional SIMT GPU architectures, implementing scalarization requires separate execution units and register files for scalar values and a broadcast network to transport values from the scalar register file to the vector execution units. AMD's GCN architecture is an example of such an architecture [AMD 2012]. It combines scalarization and a spatial SIMT GPU. In a TSIMT based GPU, however, we may use the same ALU and register file for both scalar and vector operations. The additional logic required for scalarization is limited to small changes: An additional addressing mode in the register file is needed for scalar registers. Scalar registers can be packed more densely as we only need to store one value per warp instead of one value per thread. Beside this difference in register file addressing, execution of scalar instructions is handled just like execution of regular vector instructions with a single active thread. This reduces not only the additional hardware required for scalarization, but also enables more flexibility: As opposed to conventional GPU architectures, where the separate scalar ALUs stay idle when no scalar instructions are available from the currently active warps, in TSIMT GPUs with scalarization one type of ALU is used for both scalar and vector instructions. This enables flexible adjustment to any ratio of vector and scalar instructions.

4.2. Compiler Scalarization Algorithm

We present a new scalarization algorithm for code analysis, that is able to identify instructions that are guaranteed to use the same inputs in all active threads of a warp. It also identifies which registers always store scalar values. The algorithm is shown in Figure 3. The algorithm starts by optimistically marking all registers and all instructions as scalar (Step 1). Then, instructions reading the thread local memory or the *threadid* register are marked as non-scalar (Step 2). Instructions reading non-scalar

registers are also marked as non-scalar (Step 3). Registers written by non-scalar instructions are also marked non-scalar (also Step 3). In Step 4 registers with control flow dependencies on vector values are marked as vector. The main difference of our algorithm compared to [Lee et al. 2013] lies in this stage: In the algorithm from Lee et al. all registers and instructions are marked as non-scalar where convergent control flow cannot be guaranteed. We found that their criterion is safe but too strict. If convergent control flow cannot be guaranteed, we can still scalarize as long as the register goes dead before the reconvergence point. This way only a single version of the register per warp can exist at the same time and a scalar register can be used. After Step 4 we check if any new vector registers or instructions have been found, if yes we repeat steps 3 to 5, if nothing changes we have found all vector registers and instructions. All instructions and registers that are still marked scalar are guaranteed to be uniform for the whole warp and can benefit from the scalarization capabilities of the hardware.

4.3. Implementation of the Scalarization Algorithm

The experimental evaluation presented in Section 5 uses GPGPU-Sim 3.2.1 [Bakhoda et al. 2009] extended with our enhancements. By default this GPU simulator does not simulate a real instruction set of any GPU but simulates NVIDIA's PTX intermediate code instead. PTX uses a generic ISA with an unlimited number of virtual registers. In real systems the PTX code is mapped by the driver to the actual ISA of the employed GPU. We implemented the presented Scalarization algorithm in the PTX loader of the simulator. After parsing the PTX code and identifying the basic blocks and recovergence points, the algorithm explained in Section 4.2 identifies scalar instructions and registers as well as vectors instructions and registers. In a real system the driver would subsequently map the PTX code to the actual ISA of the GPU. This mapping includes register allocation. Unfortunately GPGPU-Sim is not able to simulate this part of the mapping process but instead simulates an unlimited register file and queries NVIDIA's *ptxas* tool to inquire the number of registers required per thread after register allocation. This partial information about the results of register allocation is used by the simulator to restrict the maximum occupancy. This simulation shortcut of GPGPU-Sim is problematic as not only the number of registers needed per thread influences the performance but the register mapping also. Register allocation changes the timing of the register fetch and additional pipeline stalls may happen due to write-after-read hazards, that were not present in the PTX code prior to register allocation. GPGPU-Sim is also able to simulate PTXPlus code, that closely resembles the ISA of NVIDIA's Tesla architecture, however, as NVIDIA's Tesla architecture does not support scalarization and scalarizing works with PTX code, all simulations in this paper use PTX instead of PTXPlus.

To solve the issues with PTX we added a register allocator to GPGPU-Sim. A standard register allocator based on graph-coloring [Muchnick 1997] was implemented. It first determines which virtual registers are alive at each instruction. Then an interference graph is constructed, in which every vertex represents a virtual register. The edges connect all virtual registers that are alive at the same time. Then all vertices are colored, so that no vertex is connected to another vertex of the same color. Each color represents a physical register. As GPUs support an adjustable number of registers per thread, we try to color using the smallest number of colors possible. As this is an NP-hard problem, we employ a heuristic [Lumsdaine and Gregor 2004]. To support Scalarization this algorithm is executed twice: Once for allocating vector registers and once for scalar registers.

An important change from the standard register allocation algorithm described above is required while constructing the interference graph: Additional edges need to be added to the graph to account for interferences between different threads from

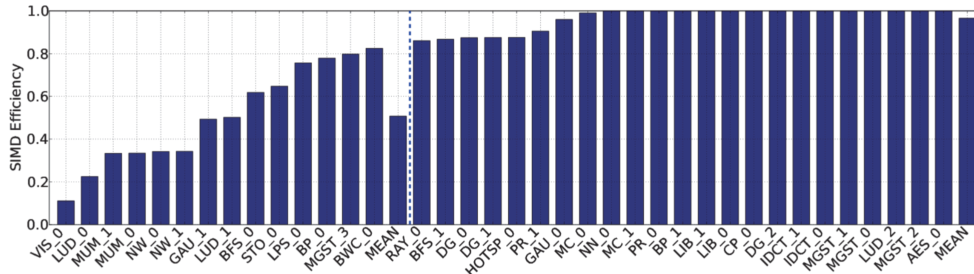


Fig. 4: Effective SIMD efficiency for conventional SIMT.

the same warp. Scalar registers are shared by all threads of a warp. For this reason the control flow of warp and the effect of the reconvergence stack need to be considered. When threads execute a divergent branch a scalar register can be alive on one branch direction but dead on the other branch direction. When such a branch is executed, scalar registers that were considered dead from the perspective of a single thread can be “resurrected” when the control flow reaches the reconvergence point. This would, however, fail if the space occupied by the scalar register had been reused in the branch path, where it is dead. For this reason all scalar registers that are alive at the first instruction after a potentially divergent branch must also be considered alive at all instructions of the other side of the branch.

5. EXPERIMENTAL EVALUATION

In this section the proposed TSIMT GPUs are experimentally evaluated using a GPU simulator. It is organized as follows: Section 5.1 describes the experimental platform as well as the benchmarks employed. Section 5.2 evaluates the properties of the TSIMT core using a synthetic microbenchmark. In Section 5.3 TSIMT is evaluated using real benchmarks, afterwards Section 5.4 explores load balacing issues we discovered in TSIMT. Section 5.5 describes how optimization to the resource allocation can reduce these issues and in Section 5.6 the performance effects of different design tradeoffs are examined. Section 5.7 evaluates STSIMT, and Sections 5.8 and 5.9 analyze the effects of Scalarization. Last, but not least Section 5.10 looks at the power and energy efficiency of TSIMT and STSIMT.

5.1. Experimental Platform and Benchmarks

For microarchitecture simulations, we utilized the cycle-level GPU simulator GPGPU-Sim 3.2.1 [Bakhoda et al. 2009] and extended it to support TSIMT. For the evaluation, we selected a large set of benchmarks listed in Table II from multiple widely-known sources such as the popular Rodinia benchmark suite [Che et al. 2009] and the GPGPU-Sim repository [Bakhoda et al. 2009]. We also included a version of breadth-first search using the virtual warp-centric programming model [Hong et al. 2011]. Table III shows our used GPU configuration. We selected a similar configuration as the configuration used in [Fung and Aamodt 2011], however, these results are not still not directly comparable, because Fung et al. used a much older version of GPGPU-Sim.

The benchmarks are selected to contain both very control-divergent kernels as well as almost and fully coherent kernels to be able to see the performance impact of TSIMT on both types of applications. The SIMT bars in Figure 4 quantify the degree of divergence by showing the average SIMD efficiency for each benchmark without com-

Abbr.	Description	Kernels	Domain	Blocks per Grid	Threads per Block	Source
AES	AES Encryption	1	Cryptography	257	256	[Bakhoda et al. 2009]
BFS	Breadth-first search	2	Graph Algorithms	1954	512	[Che et al. 2009]
BWC	BFS warp centric	1	Graph Algorithms	977	128	[Hong et al. 2011]
BP	Back Propagation	2	Pattern Recognition	4096	256	[Che et al. 2009]
CP	Coulombic potential calculation	1	Physics Simulation	256	128	[Bakhoda et al. 2009]
DG	Discontinuous Galerkin solver	3	Physics Simulation	268, 268, 603	84, 112, 256	[Bakhoda et al. 2009]
GAU	Gaussian Elimination	2	Linear Algebra	1,2704	512,16	[Che et al. 2009]
HOTSP	HotSpot	1	Physics Simulation	1849	256	[Che et al. 2009]
IDCT	H.264 IDCT	2	Video Compression	252,231, 100, 130	192	[Wang et al. 2013]
LIB	Libor stock option calculation	2	Computational Finance	64	64	[Bakhoda et al. 2009]
LPS	3D Laplace Solver	1	Physics Simulation	100	128	[Bakhoda et al. 2009]
LUD	LU decomposition	3	Linear Algebra	1-155	16,32,256	[Che et al. 2009]
MC	H.264 Motion Compensation	2	Video Compression	8160	64	[Wang et al. 2015]
MGST	Merge sort	4	Sorting	256,4, 4,2048	512,256, 256,128	[NVIDIA 2011]
MUM	DNA sequence alignment	2	Bioinformatics	196, 316	256,256	[Che et al. 2009]
NN	Nearest Neighbors	1	Data Mining	168	256	[Che et al. 2009]
NW	Needleman wunsch	2	Bioinformatics	1-128	16	[Che et al. 2009]
PR	Parallel reduction	2	Basic Parallel Algorithm	64,1	256,32	[NVIDIA 2011]
RAY	Raytracing	1	Computer Graphics	512	128	[Bakhoda et al. 2009]
STO	StoreGPU	1	Database	1-260	2-64	[Al-Kiswany et al. 2008]
VIS	Visibility Calculation	1	Game AI	24	256	AiGameDev

Table II: GPGPU benchmarks used for experimental evaluation.

Parameter	Value	Parameter	Value
GPU cores	30	SP units / core	8
Memory channels	8x 64-bit	SFU units / core	2
Core clock	1300 Mhz	L1 D-cache / core	32 KB
Interconnect clock	650 Mhz	L1 I-cache / core	4 KB
Memory clock	800 Mhz	L2 cache	1 MB
Shared mem. / core	64 KB	Max. warps / core	32
Max. blocks / core	16	Process Node	40nm

Table III: GPU configuration used for experimental evaluation.

paction. For each warp instruction, SIMD efficiency is defined as the ratio of active threads to the maximum number of threads per warp. The maximum SIMD efficiency that can be achieved is therefore 1.0. To arrive at the average SIMD efficiency for the entire kernel, the per-instruction SIMD efficiency is averaged over all executed instructions. Figure 4 groups the benchmark kernels into two categories separated by the blue dashed line. *Divergent benchmarks* are shown left of the line, these kernels have an average SIMD efficiency of less than 85%. The remaining kernels to the right of the line are called *coherent* benchmark kernels.

5.2. Synthetic Benchmark Analysis

To demonstrate the performance of a TSIMT-based GPU architecture in an isolated fashion, we developed a microbenchmark that enables us to precisely control both the *warp active masks* and the *overall number of active warps* (i.e. occupancy). We exe-

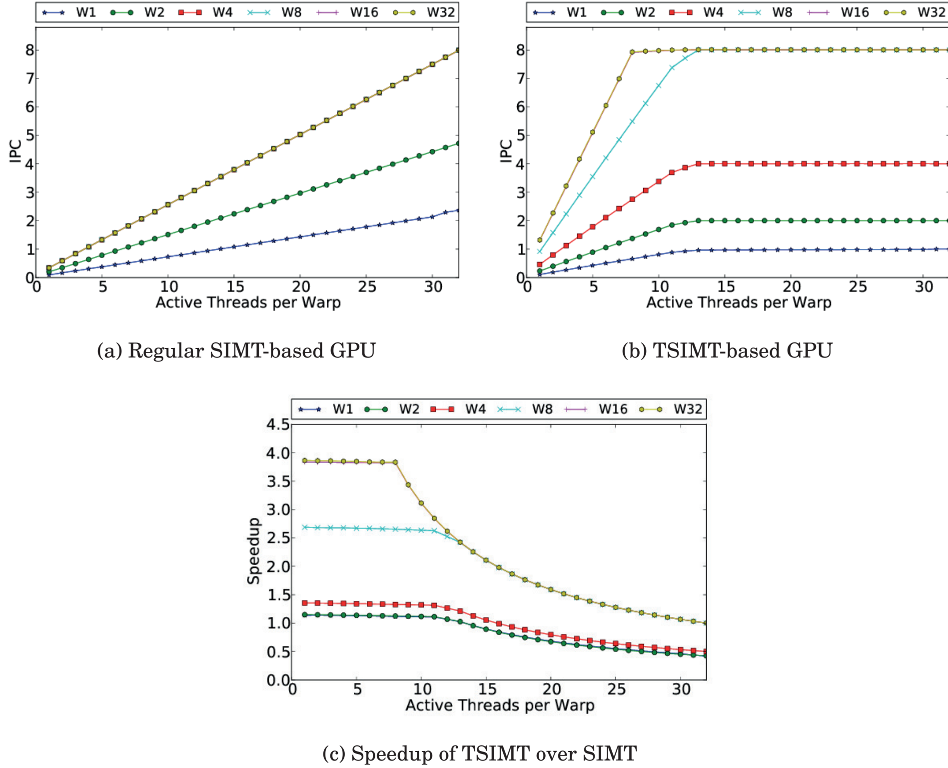
cuted this microbenchmark both on a conventional GPU core and on a TSIMT GPU core, while varying the number of active threads per warp and occupancy, and measured core IPC. For this experiment, the core's configuration is as described in Table III (execution throughput of 8 thread instructions, warp size of 32 threads). The results of these experiments are shown in Figures 5a and 5b. Both figures show the IPC achieved as a function of the number of active threads per warp for different numbers of active warps. (1 (W1) up to 32 (W32)).

Starting with the conventional GPU (Figure 5a), the effect of reducing branch divergence (i.e. the horizontal axis) is a corresponding linear increase in IPC (with an increase in active threads per warp). The maximum per-core IPC of 8 is only achieved when the execution is *coherent*, i.e. there is no control divergence. The number of active warps, on the other hand, has no effect on performance, provided it exceeds 4. The measurements for 4, 8 and 16 active warps are hidden behind the results of 32 active warps, as 4 warps are sufficient to provide full performance. Additional warps are not needed to tolerate the latency of the arithmetic pipeline but tolerate long latency memory accesses. As having many warps is important for hiding instruction latency on GPUs, the effect of having a small number of available warps is directly linked to the type of instructions executed. As our microbenchmark utilizes math instructions with relatively short latency, only the pipeline latency must be hidden. This effect is observed in the figure, where only the configurations with 1 and 2 active warps are insufficient to fully hide the latency. Therefore, these configurations cannot reach full performance.

The microbenchmarking results on the TSIMT architecture (Figure 5b) are vastly different. We begin by looking at the effect of the number of active threads per warp in the full-occupancy configuration with 32 active warps. The figure shows that the maximum IPC of 8 is reached much sooner than on the conventional GPU at only 8 active threads per warp. Below this number performance increases linearly, with the number of threads. This behavior is caused by insufficient instruction issue bandwidth: On a GPU configuration with 8 TSIMT lanes and an instruction issue bandwidth equal to that of the baseline GPU (i.e. one warp instruction per clock), the scheduler will be busy for exactly 8 clock cycles before it can re-issue to the same lane again. Therefore, each lane must be able to *at least* hide a number of clock cycles equal to the number of lanes per core with execution. To hide 8 clock cycles, a lane requires a warp instruction with at least 8 active threads. If there are fewer than 8 threads active in an instruction, the lane completes the instruction “too soon” and then remains idle until the warp scheduler issues a new warp instruction to it.

Next, we consider the effect of the number of active warps on TSIMT GPU performance. Figure 5b shows that a small number of active warps has a stronger impact on the performance of the TSIMT-based GPU than on the conventional GPU. In fact, having only a few warps available on the core enforces an *upper bound* on the achievable performance within that core. The figure demonstrates that this upper bound is equal to the number of available warps, e.g. with 4 active warps, the maximum achievable core IPC is 4. This can be explained by the warp allocation scheme in the TSIMT architecture: As the warp set is statically partitioned across all TSIMT lanes, having fewer than 8 active warps on the core means that some TSIMT lanes will not have any warps allocated to them. As a result, TSIMT cores can never reach full performance if the number of warps is so small that some lanes remain empty.

Comparing the results for SIMT and TSIMT reveals that TSIMT provides only relatively small speedups, when the average number of active threads per warp is high. Even severe slowdowns by 50% are possible in case less than 8 warps are available. On the other hand, speedups between 2.5x and 4x are possible if 8 or more warps are available and 12 or less threads per warp are enabled.



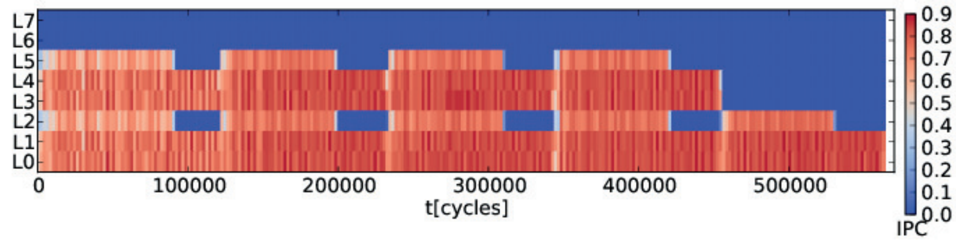
(a) Regular SIMT-based GPU

(b) TSIMT-based GPU

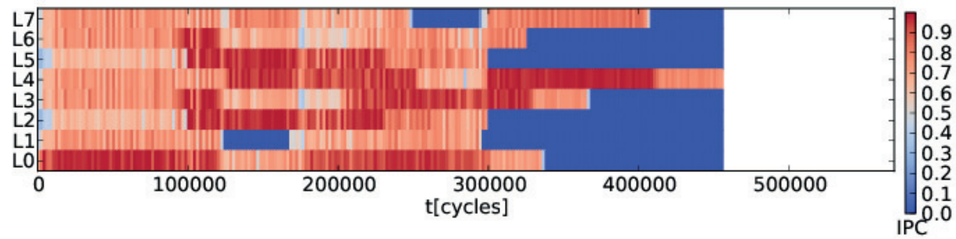
(c) Speedup of TSIMT over SIMT

Fig. 5: Microbenchmarking results showing the speedup of TSIMT over regular SIMT for different combinations of active warps and threads per warp.

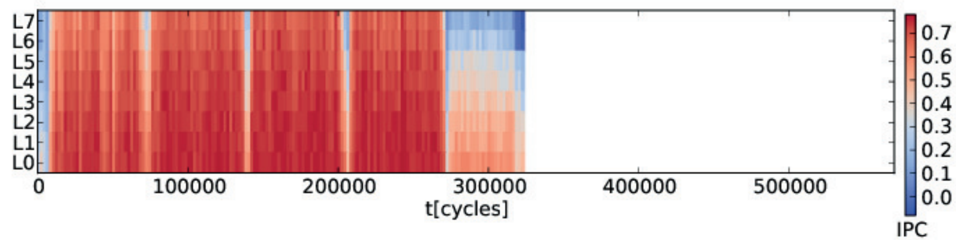
Additional insights can be gained by considering the speedup over regular SIMT. The speedup is shown in Figure 5. The speedup peaks close to four with 16 and 32 warps and 8 active threads per warp. For smaller numbers of active threads, TSIMT is unable to provide additional speedup as the warp scheduler cannot issue new instructions to the lanes any faster. In configurations with large numbers of active threads but small numbers of active warps, TSIMT exhibits slowdowns over regular SIMT. In the microbenchmark, the largest possible slowdown is equal to 0.5, as regular SIMT performance also decreases with only one or two active warps due to the inability to hide the pipeline latency. The slowdown of TSIMT over regular SIMT can be larger for real-world kernels, however, as such kernels normally contain some amount of ILP, which increases the ability of SIMT GPUs to hide the pipeline latency, even if only 1 or 2 warps are active. For TSIMT, on the other hand, ILP does not increase the performance when only a single warp is active. While ILP provides more independent instructions to the warp scheduler, these instructions can only be executed on the lane that is already busy. Lanes without any active warps remain idle. For this reason, benchmarks with only a few active warps per SM can experience drastic slowdowns with TSIMT. In the worst case (1 active warp, no control divergence, large amounts of ILP), TSIMT can never achieve more than one eight of SIMT's performance.



(a) DG_0 benchmark on unoptimized TSIMT



(b) DG_0 benchmark on optimized TSIMT



(c) DG_0 benchmark on TSIMT without lane lock

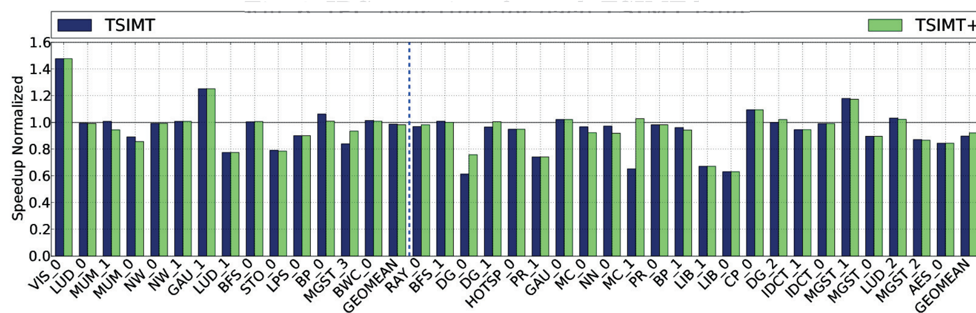


Fig. 7: TSIMT GPU Speedup compared to the baseline

5.3. Full Benchmark Analysis

Figure 7 depicts the speedup of a TSIMT-based GPU over the conventional GPU. The benchmarks on the horizontal axis are sorted by increasing average SIMD efficiency. The dotted line separates the divergent (left-hand side) from the coherent (right-hand side) benchmarks. For both types of benchmarks, the geometric mean is shown as well.

As the figure reveals, a straight implementation of TSIMT does not perform as well as one might expect. There are cases where TSIMT provides substantial performance benefits, but the overall effect from TSIMT is an average performance loss of 7.3%,

both for the divergent as well as for the coherent benchmarks. A significant performance improvement is obtained, e.g., for the GAU_1 kernel, but a severe slowdown is incurred in other kernels such as LUD_1 or the (coherent) DG_1 kernel. Some coherent benchmarks show increased performance due to the changed shared memory handling. As explained in Section 3.5 TSIMT can reduce the number of shared memory bank conflicts. Interestingly, the five most divergent kernels (LUD_0, MUM_0, MUM_1, NW_0, NW_1) experience either no change or a performance loss on TSIMT. Looking at the SIMD efficiency provides a first hint of the possible performance improvements. As Figure 4 reveals, even most divergent benchmarks have SIMD efficiencies of more than 50%, and only LUD_0, MUM_1, MUM_0, NW_0 and NW_1 have SIMD efficiencies below this level. As discussed already in the last section, kernels with high SIMD efficiency usually cannot benefit from TSIMT.

5.4. Load Balancing Issues

While investigating the matter we discovered that the limited performance improvements of TSIMT were due to load balancing issues. To illustrate these issues, we developed a tool that generates graphs that show IPC over time for each lane of one core. One these graphs is shown in Figure 6a for the DG_0 kernel. It can be seen that lanes 6 and 7 are completely empty, while lanes 2 and 5 frequently run out of work. DG_0's block size of 84 threads largely explains this behavior: A block of 84 threads is mapped to 2 full warps and 1 partially filled warp with 20 active threads. Furthermore, because of the high register requirements of this kernel, each core only holds at most 2 blocks simultaneously. The full warps are mapped to lanes 0, 1, 3, and 4 while lanes 2 and 5 execute the partially filled warps. Because the warps are only partially filled, they execute and finish much faster than the full warps. But this does not result in any performance advantage: The lanes must stay idle until the complete block is finished.

To gather event more insight into the effects of load balancing we recorded how many lanes on average had warps allocated to them and how many of these lanes had usable warps. Lanes can have warps allocated to them, but can still stall because all its warps are currently waiting for long latency memory operations. We recorded this information in the two lanes active columns of Table IV. Static means that at least one warp is allocated to the lane. However, some of these lanes are still stalled, because all warps allocated to them are waiting for long latency memory operations. The dynamic column shows how many lanes on average have at least one warp available, that is not stalled by a long latency operation. This can also be considered be the average effective width of the TSIMT core. In 19 out of 37 benchmarks more than 7 lanes on average have warps allocated to them, however, only 2 kernels have more than 7 lanes with usable warps. Some kernels such as BFS_0 or DG_2 have warps allocated to almost all lanes, but only a small number of lanes can be active because almost all warps are not available for scheduling since they are waiting for long latency memory operations.

5.5. Register Allocation Optimizations

As explained in Section 3.7 two optimizations of TSIMT register resource allocation can potentially improve the performance: First, resource deallocation on warp instead of block exit and second, allocating registers only for active threads instead of allocating register for the entire warp.

Figure 6b again shows IPC over time per TSIMT lane for an execution of DG_0 with these optimizations. Most lanes are now busy at the start of the kernel. But at the end of the kernel a strong tail effect is visible: only 1 of the 8 lanes are still busy.

The performance of these optimizations are shown in Figure 7. The overall effect of these optimizations is a slightly better performing version of TSIMT, called TSIMT+, with 6.0% performance loss compared to the SIMT baseline and a 1.4-% improvement

Kernel	Issue Conflicts			Lanes active		Warps
	1	2	≥ 3	Dynamic	Static	
AES_0	9.34%	1.45%	1.23%	4.70	7.66	7.66
BFS_0	8.92%	1.79%	0.75%	1.14	7.69	26.68
BFS_1	15.44%	4.45%	3.12%	3.06	6.84	15.25
BP_0	7.59%	2.05%	3.06%	6.72	7.97	23.14
BP_1	7.62%	0.81%	0.40%	7.02	7.97	15.16
BWC_0	5.11%	0.29%	0.03%	2.50	7.81	25.14
CP_0	4.95%	0.07%	0.01%	6.55	6.71	17.68
DG_0	7.18%	0.33%	0.01%	3.89	4.95	4.95
DG_1	10.17%	0.73%	0.04%	4.76	5.93	10.15
DG_2	10.45%	1.61%	0.27%	1.85	7.82	27.81
GAU_0	12.46%	1.78%	0.37%	1.11	1.61	2.41
GAU_1	17.87%	3.58%	0.80%	3.80	7.55	14.29
HOTSP_0	10.24%	2.12%	1.34%	5.34	7.72	7.72
IDCT_0	5.14%	1.14%	1.07%	4.65	7.47	19.89
IDCT_1	3.14%	0.44%	0.32%	4.86	7.67	20.47
LIB_0	1.44%	0.01%	0.00%	3.24	4.08	4.08
LIB_1	1.86%	0.02%	0.00%	2.87	4.11	4.11
LPS_0	16.61%	3.92%	1.42%	4.47	6.43	10.09
LUD_0	0.00%	0.00%	0.00%	0.13	0.33	0.33
LUD_1	0.00%	0.00%	0.00%	0.23	0.38	0.38
LUD_2	13.58%	1.78%	0.31%	4.92	7.05	20.95
MC_0	11.47%	1.17%	0.13%	7.47	7.86	19.31
MC_1	6.94%	0.45%	0.04%	5.14	7.85	20.73
MGST_0	7.04%	0.69%	0.28%	6.62	7.60	15.18
MGST_1	5.42%	0.73%	0.57%	0.87	2.57	2.57
MGST_2	10.13%	1.93%	1.09%	1.95	2.59	2.59
MGST_3	13.59%	2.03%	0.46%	5.22	7.77	28.56
MUM_0	7.26%	0.97%	0.31%	2.37	7.36	20.80
MUM_1	9.01%	1.05%	0.47%	1.08	5.98	8.91
NN_0	9.51%	1.31%	0.67%	6.53	7.06	21.11
NW_0	2.70%	0.05%	0.00%	0.23	2.54	2.54
NW_1	2.91%	0.06%	0.00%	0.20	2.55	2.55
PR_0	3.87%	0.08%	0.02%	1.95	7.92	16.77
PR_1	0.00%	0.00%	0.00%	0.13	0.33	0.33
RAY_0	8.91%	1.24%	0.31%	6.14	7.50	7.50
STO_0	2.43%	0.99%	1.05%	2.24	4.89	4.89
VIS_0	23.60%	6.71%	1.86%	0.66	3.03	3.03

Table IV: Benchmark Scheduling Statistics

over unoptimized TSIMT. For most benchmarks, the optimizations have no effect, but some particularly problematic cases (MGST_3, DG_0 and DG_1) show speedups between 5% and 10%. Unfortunately, in some benchmarks, the optimizations lead to extended tail effects, thereby causing slight slowdowns compared to TSIMT.

5.6. TSIMT Design Tradeoffs

Another potential bottleneck in TSIMT can be the instruction issue bandwidth. Multiple lanes can finish execution of their current warp instruction in the same cycle, but the frontend can only supply a new instruction to a single lane each cycle. If two or more lanes request new instructions at the same time, all but one lane will stall because the frontend cannot supply a new instruction fast enough. To discover how common these stall cycles are, we recorded how often they happen on average and show this information in the "Issue Conflicts" columns of Table IV. The table shows how often 1, 2 or 3 or more instructions could not be issued as soon as they were ready for issue and their lane was ready to accept them, but the instruction frontend was busy with issuing an instruction to a different lane. On average 7.9% of the cycles two lanes are awaiting for instructions. This can also be seen as a temporary reduction of average effective number of lanes. In some benchmarks such as VIS_0, GAU_1, BFS_1

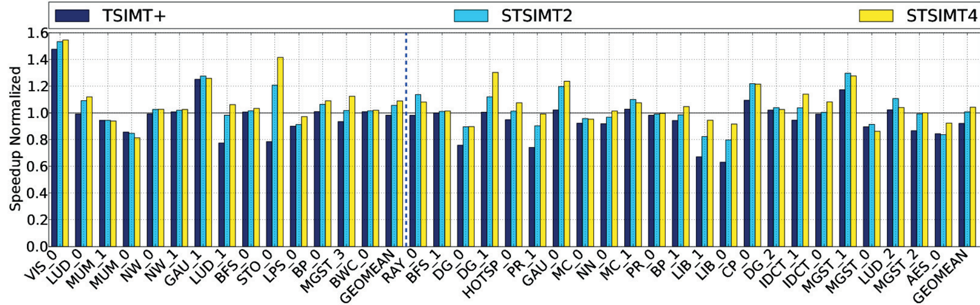


Fig. 8: Speedup for STSIMT with different lane width

and LPS_1 in more than 20% of the cycles one or more lanes are stalled because the frontend cannot supply instructions fast enough.

In the previous section we already noticed that load balancing issues between different lanes hurt the performance of TSIMT. Additionally, we cannot exploit ILP in TSIMT as all instructions, even if independent, need to be executed on the same lane. To determine how much these issues reduce the performance of TSIMT, we also simulated an unrealistic configuration of TSIMT, where all warps can issue instructions to all lanes. The overall effect of removing the locking of warps to a specific lane is a performance improvement of 7.6% compared to SIMT and of 16.4% over TSIMT.

A more realistic approach than removing the lane locking for improving the TSIMT load balancing is to reduce the warp size. We simulated a configuration with warp size of 16 and found that overall the performance improves by 1.4% over SIMT. Reducing the warp size improves the load balancing as more warps are available and the GPU can exploit ILP within a warp for additional performance. However, reducing the warp size increase the load of the frontend. With a warp size of 16, instruction fetches need to be amortized over a smaller number of threads. But the reduction of warp size can also have positive effects on divergent memory accesses. The results show that on average smaller warps are better than TSIMT with 32-wide warps.

5.7. Spatio-Temporal SIMT

While the optimizations presented Section 5.5 help some applications, they do not resolve the main problem of TSIMT: Severe performance reductions when only few warps are available and/or load balancing issues between the lanes. As described in Section 3.8, spatio-temporal SIMT reduces the impact of these problems, as the warp pool is partitioned across a smaller number of lanes: With only four or two lanes, it becomes much more likely that each lane receives at least one active warp and that the work is distributed equally over all lanes. At the same time it also reduces the latency

Figure 8 shows the performance of regular SIMT, TSIMT and STSIMT with two and four ALUs per lane. The number of lanes was adjusted to keep the number of functional units (FUs) identical in all configurations, i.e. TSIMT has 8 lanes with one FU each, STSIMT2 has 4 lanes with 2 FUs each, and STSIMT4 has 2 lanes with 4 FUs each. In the results, we observe improvements over the conventional SIMT architecture for both two and four wide STSIMT. The maximum speedup is observed in the STO benchmark where regular TSIMT experiences a 15% slowdown compared to the baseline while STSIMT4 shows a 51% speedup. Due to the low active warp count, the STO benchmark was not performing well on TSIMT. On average, STSIMT4 performs about 6% faster than the baseline on the divergent benchmark set and 10% faster than

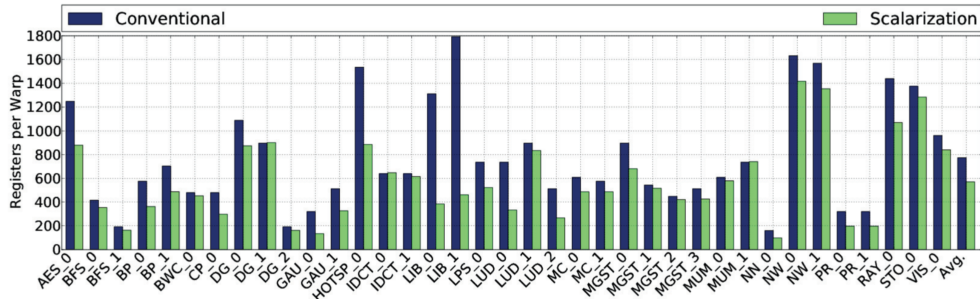


Fig. 9: Registers per Warp

TSIMT. For the coherent benchmarks, STSIMT4 is 12.6% faster than TSIMT and 5.9% faster than the baseline. The very short GAU_0 benchmark exhibits an unusual behavior: It runs much faster despite having almost no divergence. This happens because a large part of the divergent instructions in this benchmark are very slow division instructions that are responsible for tail effects. The convergent instructions are mostly simple and fast instructions, with little influence on the total runtime of the kernel.

5.8. Scalarization Results

We executed our new scalarization algorithm on the kernels described in Section 5.1. We verified that our algorithm works correctly and does not scalarize non-scalar registers or instructions by adding checks to the simulator. Figure 9 shows the number of registers required per warp, before and after scalarization. Figure 10 shows how many executed instructions were scalarized by our algorithm. Without scalarization the kernels required an average of 24.2 vector registers per thread. After scalarization 17.6 vector registers and 9.0 scalar registers are required on average per thread. While the sum of scalar+vector registers is slightly higher than the number of registers without scalarization, each scalar register is allocated only once per warp instead of once per thread. This reduces the register file space needed per warp significantly: With 32 thread wide warps, the kernels require an average of 773.2 registers per warp without scalarization, but with scalarization only 571.1 registers are needed per warp. With our new algorithm scalarization reduces register file space required per warp by 26.1%. If we restrict the algorithm to scalarize only when convergent control flow can be guaranteed, as was proposed in [Lee et al. 2013], then less scalarization is possible and 695.0 registers are required per warp. Likewise the number of instructions that could be scalarized is also lower: With our algorithm 31.1% of the static instructions are classified as scalar and 30.3% of the executed instructions are scalar instructions, but with the restriction to convergent control flow only 12.9% of the instructions could be scalarized and 13.5% of executed instructions could be scalarized.

All benchmarks but one execute at least 6% scalar instructions. Only the STO benchmark executes almost no scalar instructions (0.4%). The GAU_0 kernel has the highest percentage of scalar instructions executed with 64.2%, the fraction of scalar instruction identified is slightly lower at 57.1%. LIB_1 has the highest number of static scalar instructions at 66.1%, but the fraction of executed scalar instructions is only 48.8%. Compared to the previous restricted scalarization algorithm our algorithm scalarizes more than double the number of instructions and the number of registers is reduced by 17.8%. This scalarization algorithm is well suited for code with complex control flow, as it makes scalarization possible where convergent control flow cannot be guaranteed.

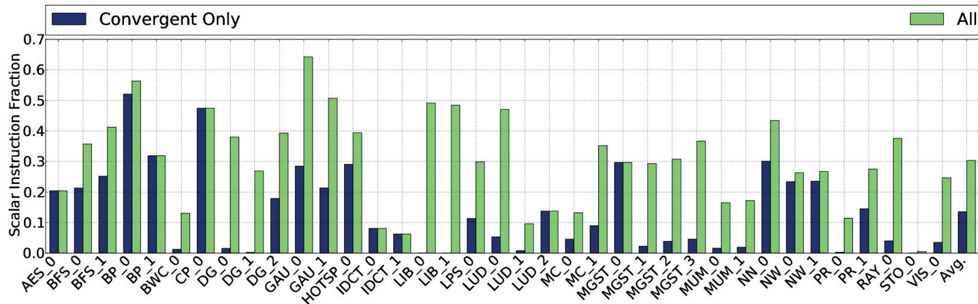


Fig. 10: Scalar fraction of executed instruction for regular algorithm (all) and algorithm restricted to convergent control flow

5.9. Putting it all together: TSIMT+Scalarization

We combined Scalarization with TSIMT, and performed the experiments again. Figure 11 compares the performance achieved with SIMT with a GCN-style scalarization (SIMT_SCALAR), TSIMT+, TSIMT+ with Scalarization and the best spatio-temporal SIMT configuration (STSIMT4) with and without scalarization (STSIMT4 and STSIMT4_SCALAR). SIMT without scalarization is employed as a baseline.

The geometric average of the speedup achieve by TSIMT+ due to scalarization to the optimized TSIMT configuration is 16.1%. Gains from applying Scalarization to STSIMT4 are slightly lower with a 13.0% higher performance than in STSIMT4 without scalarization. The highest scalarization speedup of 4.2x over TSIMT+ can be seen in the BP_0 kernel. Three reasons explain the very high speedup of this kernel: First, a high ratio of 56% scalar instructions. Second, many scalar instructions are low throughput SFU instructions, while the vector instructions are mostly high throughput instructions. Third, the performance differences reorder the memory accesses and this improves the DRAM efficiency significantly from 12% to 42%. In MC_1, on other hand, Scalarization causes a slowdown of almost 40%. In this case scalarization allows placing 4 instead of 3 warps in a lane. This increased occupancy is normally beneficial, but in some rare cases such as in this kernel, the higher occupancy causes the number of shared memory bank conflicts to increase by more than 10x and also decreases locality, which results in 65% more read misses in the L1 data cache. Some kernels show almost no change in performance. In many cases this is connected to a low fraction of scalar instructions such as in STO_0, PR_0, IDCT_0 and IDCT_1. Some kernels such as BFS_1 use many scalar instructions but still do not profit from Scalarization. This happens if the performance of the GPU kernel is not limited by compute throughput but by another bottleneck. The BFS benchmark, for example, is a graph benchmark and is limited mostly by memory bandwidth and latency rather than compute throughput.

We also evaluated Scalarization on a conventional GPU. Similar to AMD's GCN architecture, an entire scalar datapath including an additional scalar register file and scalar execution units as well as a broadcast network to transmit scalar values back to the vector datapath was added to the simulated architecture. This additional hardware overhead is significant, which needs to be considered when comparing it to TSIMT+Scalarization. On kernels with little divergence conventional SIMT+Scalarization performs slightly better (+1.3%) than STSIMT4+Scalarization. On benchmarks with higher divergence, however, STSIMT4 with Scalarization performs better than SIMT+Scalarization (+4.2%). Benchmarks that show high performance gains from Scalarization on one architecture such as BP_0, MGST_3, HOTSP_0 and AES_0 show high performance gains from Scalarization across all architectures.

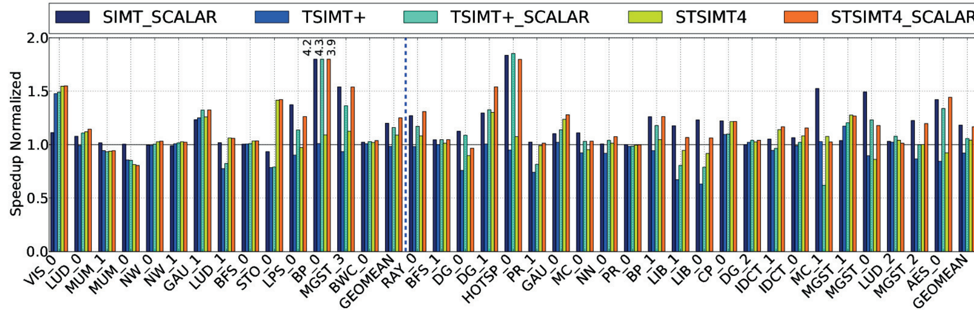


Fig. 11: Performance with and without Scalarization for TSIMT and STSIMT4 Configurations, normalized to SIMT

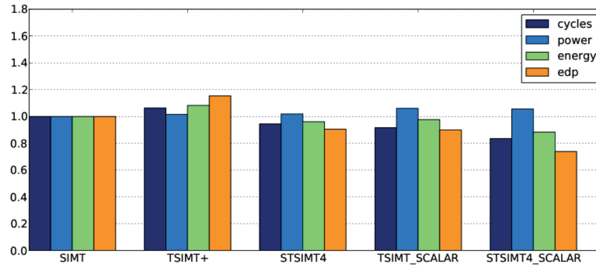


Fig. 12: Geometric Mean of Runtime, Power, Energy and EDP

Benchmarks such as PR_0 or NW_1 that do not benefit from Scalarization on a conventional GPU do not profit from adding Scalarization to TSIMT either.

5.10. Power and Energy

We extended GPGPU-Sim to allow power modeling of TSIMT as well as SIMT GPUs. Figure 12 shows the power estimates of this power model for different variants of TSIMT. Despite lower performance on average the power consumption of TSIMT+ is approximately the same as that of regular SIMT (101% of SIMT power). These configurations are thus less energy efficient than regular SIMT. The architecture with STSIMT4 provides higher performance (5.6% shorter runtime) dissipating only slightly higher power (1.8% higher power), but lower energy consumption (-4.1%) and thus improve energy efficiency (EDP reduced by 9.5%). Scalarization results in significant performance gains (+9.3% over baseline TSIMT, +20% with STSIMT4). Because Scalarization improves the utilization of resources it increases the power consumption (5.9% for TSIMT, 5.6% for STSIMT4), but overall energy consumption is decreased because of the shorter execution time (-8.3% for TSIMT, -16.4% for STSIMT4). EDP is improved by 10.1% for regular TSIMT with Scalarization and by 26.2% for STSIMT4 with Scalarization. These results show that by combining STSIMT and Scalarization the energy efficiency can be improved significantly.

6. CONCLUSIONS

This paper has presented a microarchitecture implementation and optimizations, and a rigorous performance evaluation of temporal SIMT GPUs. TSIMT aims at improving the performance of control divergent GPGPU workloads by executing warps over

time instead of over space as regular spatial SIMT GPUs do. A microbenchmark analysis has shown that TSIMT offers significant performance benefits compared to spatial SIMT, provided there are sufficient warps are available. When evaluated with complete benchmarks, however, the basic TSIMT approach generally achieves lower performance compared to spatial SIMT. A detailed performance analysis has revealed that TSIMT suffers from lane load balancing and occupancy issues and microarchitecture optimizations have been presented to improve this for some benchmarks.

In addition we have proposed and evaluated a more general solution, called spatio-temporal SIMT (STSIMT) that offers the control divergence mitigation of TSIMT while significantly reducing the high occupancy and load-balancing requirements of TSIMT. Using a particular configuration of STSIMT, an average speedup of 8% was achieved for control divergent benchmarks and 6% on average for all benchmarks.

Scalarization has been combined with TSIMT with a hardware cost that is much lower than a SIMT GPU with Scalarization. It improves performance by 16% over regular SIMT. We also showed that a previously published scalarization algorithm employs overly restrictive rules, and presented a scalarization and register allocation algorithm, that is well suited for extracting scalar instructions from kernels with divergent control flow. By applying this algorithm double the number of instructions could be scalarized and 26.1% fewer registers were required per warp.

It has also been shown that several of the proposed designs provide significant power and energy advantages. The most energy-efficient design (ST-SIMT4 with Scalarization) improves the energy delay product by 26.4% on average.

Future work includes new methods for reducing lane load imbalance such as flexible warp sizes and GPUs that can dynamically switch between SIMT and TSIMT operation modes. Further code and microarchitecture optimizations are possible to increase the performance of TSIMT architectures. Moreover, current benchmarks are not targeted at and optimized for TSIMT and therefore often incur a performance reduction. If the programmer is targeting a TSIMT-based execution architecture, divergent code can be implemented in a more straightforward way and still be executed with high performance by the GPU. We plan to demonstrate this in future work.

REFERENCES

S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. 2008. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proc. 17th Int. Symp. on High Performance Distributed Computing*.

AMD. 2012. AMD Graphics Core Next GCN Architecture White Paper. (2012).

A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software, ISPASS*.

N. Brunie, S. Collange, and G. Diamos. 2012. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proc. 39th Int. Symp. on Computer Architecture, ISCA*.

S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. IEEE Int. Symp. on Workload Characterization, IISWC*.

S. Collange. 2011. Identifying Scalar Behavior in CUDA Kernels. (2011).

B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira. 2011. Divergence Analysis and Optimizations. In *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT' 11)*. IEEE, 320–329.

J. R. Diamond, D. S. Fussell, and S. W. Keckler. 2014. Arbitrary Modulus Indexing. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM Int. Symp. on*. IEEE, 140–152.

G. F. Diamos, A. Robert Kerr, S. Yalamanchili, and N. Clark. 2010. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proc. 19th Int. Conference on Parallel Architectures and Compilation Techniques, PACT*.

W. W. L. Fung and T.M. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. In *Proc. 17th Int. Symp. on High Performance Computer Architecture, HPCA*.

W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. 40th Annual IEEE/ACM Int. Symp. on Microarchitecture, MICRO*.

W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. 2009. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Archit. Code Optim.* 6, 2, Article 7 (July 2009).

M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. 2012. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *Proc. 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE Computer Society, 96–106.

S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 267–276.

N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally. 2004. Stream Register Files with Indexed Access. In *Proc. 10th Int. Symp. on High Performance Computer Architecture, HPCA*.

S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31 (2011).

J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten. 2013. Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures. In *Proc. 40th Annual Int. Symp. on Computer Architecture*.

R. M. Krashinsky. 2011. Temporal SIMT Execution Optimization. (Aug. 2011). Patent No. US 2013/0042090 A1, Filed August 12th, 2011, Issued Februar 14th., 2013.

Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. 2011. Exploring the Trade-offs Between Programmability and Efficiency in Data-parallel Accelerators. In *Proc. 38th Annual Int. Symp. on Computer Architecture, ISCA*.

Y. Lee, R. Krashinsky, V. Grover, S.W. Keckler, and K. Asanovic. 2013. Convergence and Scalarization for Data-Parallel Architectures. In *Proc. Int. Symp. on Code Generation and Optimization (CGO)*.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008a. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* (March 2008).

J.E. Lindholm, M.Y. Siu, S.S. Moy, S. Liu, and J.R. Nickolls. 2008b. Simulating Multiported Memories using Lower Port Count Memories. (2008). Patent No. US 7339592 B2, Filed July 2004, Issued March 2008.

A. Lumsdaine and D. Gregor. 2004. Boost Graph Library: Sequential Vertex Coloring. http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/sequential_vertex_coloring.html. (2004).

S. S. Muchnick. 1997. *Advanced Compiler Design and Implementation*.

V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. 2011. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *Proc. 44th Annual IEEE/ACM Int. Symp. on Microarchitecture, MICRO*.

NVIDIA. 2011. NVidia GPU Computing SDK 3.1. (2011).

J. E. Smith, G. Faanes, and R. Sugumar. 2000. Vector Instruction Set Support for Conditional Operations. In *Proc. 27th Annual Int. Symp. on Computer Architecture, ISCA*.

A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi. 2013. SIMD Divergence Optimization Through Intra-warp Compaction. In *Proc. 40th Annual Int. Symp. on Computer Architecture, ISCA*.

B. Wang, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink. 2013. An Optimized Parallel IDCT on Graphics Processing Units. In *Euro-Par 2012: Parallel Processing Workshops*. Leture Notes in Computer Science, Vol. 7640. Springer Berlin Heidelberg, 155–164.

B. Wang, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink. 2015. Parallel H.264/AVC Motion Compensation for GPUs using OpenCL. *Cir. and Sys. for Video Technology, IEEE Trans. on* 25, 3 (March 2015), 525–531.

H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU Microarchitecture Through Microbenchmarking. In *Proc. IEEE Int. Symp. on Performance Analysis of Systems Software, ISPASS*.

P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou. 2013. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *Proc. 27th Int. ACM Conference on Int. Conference on Supercomputing*.

P. Xiang, Y. Yang, and H. Zhou. 2014. Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation. In *Proc. 20th Int. Symp. on High Performance Computer Architecture, HPCA*.

G. Ziegler. 2011. Analysis-Driven Optimization. (2011). http://www.nvidia.de/content/PDF/isc-2011/Ziegler.pdf.