Stack-less SIMT reconvergence at low cost

Caroline Collange
ENS de Lyon, Université de Lyon, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL),
École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

September 12, 2011

Abstract

Parallel architectures following the SIMT model such as GPUs benefit from application regularity by issuing concurrent threads running in lockstep on SIMD units. As threads take different paths across the control-flow graph, lockstep execution is partially lost, and must be regained whenever possible in order to maximize the occupancy of SIMD units. In this paper, we propose a technique to handle SIMT control divergence that operates in constant space and handles indirect jumps and recursion. We describe a possible implementation which leverage the existing memory divergence management unit, ensuring a low hardware cost. In terms of performance, this solution is at least as efficient as existing techniques.

1 Introduction

Graphics processing units (GPUs) have gradually become credible alternatives to high-performance general-purpose processors for many applications that exhibit data parallelism. This applicative field exceeds by far the graphics rendering domain [10]. Indeed, the first place in the Top500 supercomputer ranking in October 2010 was belonging to a GPU-based computer.

All current-generation GPUs operate according to the SIMT (Single Instruction, Multiple Threads) execution model. From the programmer's and compiler's point of view, this model is similar to SPMD (Single Program, Multiple Data). The programmer writes a single program or *kernel*. A large number of instances of the kernel (or *threads*) are then run in parallel.

During execution on a GPU, transparent hardware mechanisms group threads in convoys named *warps* to execute their instructions on SIMD units. Unlike architectures having SIMD instruction sets dealing with explicit vectors, SIMT architectures apply vectorization at runtime rather than at compile-time [18].

Although this model offers a good balance of performance and programmability compared to classic SIMD instruction sets, handling thread divergence dynamically has a cost in area, power and hardware complexity.

In addition, all techniques currently in use are based on static scheduling decisions taken by the compiler. This initial choice may be sub-optimal during execution, and the micro-architecture is unable to affect it. Current SIMT mechanism also exclude the most common instruction sets, which do not provide re-convergence information. This hinders a broader adoption of the SIMT model, especially in general-purpose processors.

Finally, although a number of individual solutions have been proposed in the 1980s and 1990s, no systematic survey of divergence control seems to exist as of today.

The goal of this paper is to address each of these issues. First, we survey existing thread divergence control techniques in the context of SIMT, vector and SIMD architectures. Second, we propose a dynamic mechanism to allow thread resynchronization. It is implemented solely at the micro-architectural level, and can operate on conventional scalar instruction sets. Finally, we propose to implement this mechanism by reusing the memory access unit hardware present in SIMT processors, keeping the hardware overhead minimal.

We present the overview of existing work related to control divergence management in Section 2. We then describe the proposed technique in Section 3. The hardware implementation is described in Section 4. Finally, we discuss the correctness of the method and quantify its efficiency through simulation in Section 5, and offer some perspectives.

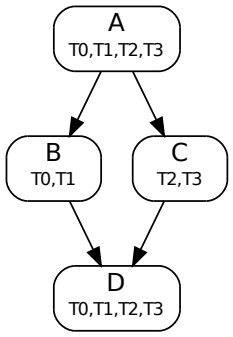
Originally published as *Étude comparée et simulation d'algorithmes de branchements pour le GPGPU* in proceedings of SympA'2011 [2].

2 Related work

In this section, we present various techniques that aim at maintaining and restoring synchronization between threads of a warp. The majority of these methods were designed in the context of SIMD and vector processors in the 1980s and 1990s. To better highlight similarities, we will describe them using contemporary GPU-related terminology.
We distinguish *explicit* methods, whose behavior is exposed at the architectural level and which require specific action from the compiler, from *implicit* methods, which are restricted to the micro-architectural level and can operate on standard scalar instruction sets.
Divergence control mechanisms address two distinct issues:

- In which order the control flow graph is traversed: which divergent branch will be executed first.
- How thread reconvergence is detected: when should divergent paths be re-united.

The traversal order has a significant influence on efficiency: an unoptimal order will lead to multiple execution of the same basic block.



(a) Control flow graph

BB	Active threads
A	0, 1, 2, 3
B	0, 1
C	2, 3
D	0, 1, 2, 3

(b) Optimal traversal order

BB	Active threads
A	0, 1, 2, 3
B	0, 1
D	0, 1
C	2, 3
D	2, 3

(c) Sub-optimal traversal order

Figure 1: Example of two possible SIMT traversals of the control flow graph of an `if-then-else` block run by 4 threads. Basic blocks are annotated with identifiers of threads that run them.

For instance, Figure 1 presents the example of a conditional block executed by 4 threads named T0 to T3. Threads T0 and T1 follow the branch containing block B, while threads T2 and T3 take the other branch. If block D is executed before block C while only threads T0 and T1 are active (Fig. 1c), it will have to be executed a second time on behalf of threads T2 and T3.
The control management unit also have to identify control flow reconvergence points to regroup threads. The mechanisms which will be presented in the following are summarized according the criteria of traversal order and reconvergence on Table 1.
For techniques which allow function calls, the call stack can be implemented in two ways. The difference between the solutions appear in cases of recursion.
Traditional parallel architectures share a single stack pointer per warp. This ensures that concurrent accesses to the call stack from different threads are always kept synchronized. On the other hand, this restricts SIMT execution to threads that share the same stack pointer in addition to sharing the same instruction pointer. Threads that are at different call nesting levels cannot run together.
The other solution consists in maintaining an independent stack pointer per thread. In case of recursion, control divergence is reduced. It is replaced by memory divergence: accesses to the stack become irregular.

2.1 Explicit, stack-based

Pixar Chap The Pixar Image Computer, a machine dedicated to image processing developed in the early 1980s constitutes a notable early work [14]. It is based on SIMD processors named Chap, whose instruction set includes control instructions that reflect usual control flow structures of imperative programming languages: `if-then`, `else`, `fi`, `while-do`, `done`.

Table 1: High-level comparison of divergence control schemes. We indicate for each technique whether it supports recursion and indirect branches. "ISA" refers to explicit annotations in the instruction set. "Sync" indicates that the stack pointers of threads are kept synchronized, as opposed to "Div".

Technique	Traversal order	Reconvergence	Recursion	Indirect
Lorie-Strong [16]	Priorities	ISA + priorities	No	No
Pixar Chap [14]	ISA	ISA + mask stack	No	No
AMD [1]	ISA	ISA + mask stack	Sync	No
POMP [13]	ISA	ISA + counters	Sync	No
Intel GMA [11]	ISA	ISA + counters	No	No
NVIDIA [7]	ISA + address stack	ISA + mask stack	Sync	Yes
Intel Sandy Bridge [12]	ISA	ISA + PC	No	No
Takahashi [21]	PC + activity bits	PC	No	No
SympA'13 [3]	PC + address stack	mask stack	No	No
This paper	PC	PC	Div	Yes

These instructions allows to describe independent control flow for each SIMD lane. Their semantics is described in terms of operations on the current predication mask and two mask stacks, respectively dedicated to conditional blocks (`if-then-else`) and to loops. Current GPUs from AMD follow a similar approach [1].

POMP An improvement proposed by Keryell and Paris for the POMP parallel computer project consists in noticing that masks stored in the stack always form an histogram shape, as a thread which is inactive at depth d will necessarily stay inactive at depth $d + 1$. Thus, the only information needed is the nesting level of the last activity of each thread. The amount of data to store becomes $O(log(n))$ versus $O(n)$ for a traditional stack, n being the maximal nesting depth [13]. Activity counters are used in integrated GPUs on Intel platforms prior to Sandy Bridge [11].

NVIDIA Tesla A design goal of the NVIDIA Tesla GPU architecture is the ability to run general-purpose application [15]. The Tesla instruction set includes conditional branch instructions resembling those of scalar processors, rather than structured control instructions as other GPUs. It is complemented by annotations pointing at divergence and reconvergence points.
Divergence is handled using a stack-based scheme, as in Chap and AMD GPUs. However, Tesla provides less information in program code than other instruction set. In particular, the matching between divergence points and reconvergence points is not explicit in the binary. Hence, the divergence stack needs to store the address of reconvergence points in addition to masks. This excludes an implementation based on activity counters.
It is possible to transform an arbitrary control flow graph into properly nested conditional blocks and loops at compile time, at the possible expanse of replicated code when the graph is not reducible [23]. It is thus not strictly necessary to rely on a scheme such as NVIDIA's to run arbitrary code. However, the mechanism used by Tesla can be extended to support indirect jumps, as offered by the Fermi architecture [18]. Additionally, this approach gives some flexibility in the traversal order that other techniques do not allow. It tends to move part of the scheduling decisions from the architecture to the micro-architecture.

2.2 Implicit, stack-based

We proposed a stack-based solution to manage divergence and reconvergence without compiler intervention in an earlier work [3]. It allows code written in standard scalar instruction sets to be run in SIMT mode without recompilation.
However, using a stack has several drawbacks. The first difficulty is designing an efficient hardware stack implementation. Current architectures such as Tesla use a dedicated cache which contain the few topmost entries of the stack for each warp. The other entries are spilled to the lower levels of the memory hierarchy. The cache costs area and power, and more importantly requires an extra read/write port to memory. For instance, Tesla employs a cache of 3 four-entry blocks per warp, each entry containing 64 bits, totalizing 3KB per warp processor[1].

[1] Pathscale *pscnv* driver documentation. `https://github.com/pathscale/pscnv/wiki/Nvidia_Compute`

Additionally, stack-based methods are weak against ill-behaved software. The stack can overflow, underflow or fall into an inconsistent state if divergence and reconvergence instructions are not properly nested. The hardware and the operating system or driver need to detect and handle these exceptional cases to abort the program and restore the stack in a clean state. A stack also adds a state which needs to be saved and restored during context switches.
Finally, stack-based methods are based on a complex state machine. It is challenging to validate to guarantee that the semantics of programs is observed and that the stack state stays consistent.
For all these reasons, we will now consider stack-less algorithms, whose associated state stays in constant-space.

2.3 Explicit, stack-less

Sandy Bridge The integrated GPU of Intel's Sandy Bridge processor uses a different technique than its predecessors. Rather than maintain stacks or counters, the GPU maintains one Program Counter (PC) per thread [12]. Thus, each thread has its own copy of all architectural registers and the PC, which is enough to fully characterize its state.
The instruction set is kept similar to previous generations, and contain explicit instructions to describe conditional blocks and loops. These instructions update the individual PC of each thread. The processor then just needs to compare the PC of each thread against the global PC to find active threads. Reconvergence happens when the PCs of different threads coincide.

Lorie-Strong Intel's multiple-PC solution is close to an older implementation described in an IBM patent by Lorie and Strong [16]. In this solution, the compiler performs a relaxed topological sort on the control-flow graph and labels all basic blocks following this order. This numbering controls both the order of traversal of the control flow graph and the detection of reconvergence.
The processor maintains both a PC and a block number per thread. In the case of divergence, the block with the smallest number will be run in priority. When a potential reconvergence point is reached, the block number of each thread is compared with the number of the next block the processor is about to execute. When a match is found, the associated thread is re-enabled.
In the two solutions just described, the divergence control mechanism is exposed at the architectural level. The order of traversal of the control flow graph is set statically during compile time.

2.4 Implicit with activity bit

A technique proposed by Takahashi in 1997 allows running code in SIMT mode without requiring annotations in the instruction set [21]. Each thread has its own PC. A control unit walks through the control flow graph. When a branch is encountered, priority is given to the branch whose entry point has the lowest address, as long as at least one thread is active. When no thread is active, the other branch is taken.
The assumption that appears to be made is that reconvergence points are found at the "lowest" point of the code they dominate, that is at the highest address. The strategy consists in attempting to always execute the instructions whose addresses are lowest when deciding which branch to execute, in order not to pass a potential reconvergence point.
The drawback of this method is that thread activity signals are delayed by one clock cycle as they are sent back to the control unit. At the time the control unit receives the value, the data associated with the last branch instruction are not available any more. The processor has to continue running the program while all threads are inactive. *Else* blocks are always run regardless of divergence, and loops always run one extra iteration.
Besides, the fact that the processor can speculatively follow branches which are not taken by any thread leaves the possibility that it encounters invalid instructions or runs past the code section. Unlike the case of branch predictors in superscalar processors, no provision is made to return to a non-speculative former state.
To circumvent those problems, the authors propose that branch instructions be duplicated by the compiler. However, this cancels the advantage of not depending on the compiler.

3 Implicit stack-less reconvergence

We now propose a technique which offers both the advantage of the multiple-PC solution (constant space state) and of our previous proposition (not exposed at the architectural level). Unlike the solution of Takahashi, it does

Listing 1: Example of a case when the min(PC) policy fails to reconverge threads.

```
void kernel() {
    ...
    if(c) {  // Divergent condition
        f();
    }
    ...
}

void f() {
    ...
}
```

not require to run through basic block with no active threads.
Despite its apparent simplicity, this method does not appear to have been studied specifically in the past. Fung considers several scheduling policies for Dynamic Warp Formation, some of which can be implemented in constant space, but does not consider their use in the context of "static" warp formation [8, 9].
Meng, Tarjan and Skadron compare opportunistic reconvergence when the PCs of several thread coincide with reconvergence at post-dominators in the context of Dynamic Warp Subdivision [17]. They still rely on a predicate stack as the primary means to achieve reconvergence.
DWF require a complete restructuring of the execution pipeline and is demanding in hardware resources. For instance, Fung estimates the area overhead of dynamic warp formation to be 1.6353mm^2 in 90nm technology. For reference, a complete SIMT processor of an NVIDIA G80 GPU in the same process node occupies 6.37mm^2 according to our measurements on a die photography.
In this paper, we consider a more modest architecture that aims at contained power consumption and area. We maintain a distinct PC per thread as in the Intel solution, and compute the common PC from the value of individual PCs.

Reconvergence We identify reconvergence situations by comparing the values of individual PCs with the selected common PC. Unlike the technique used in Intel GPUs, we operate the comparison continuously, on each cycle. It is thus unnecessary to signal explicitly potential reconvergence points in machine code.
Branch instructions are handled in a distributed way: each thread computes and update its own PC, as in a MIMD processor. Divergence happens when the local PC of a thread receives a different value than the PC of other threads.

Traversal order An arbiter determines the common PC, that is the address of the next instruction to fetch, from the values of individual PCs. As in the implicit solution we proposed at SympA'13 and to some extent like in the patent from Lorie and Strong, we choose the PC of minimal value. This choice corresponds to the DPC policy in Fung's Dynamic Warp Formation [8].
This amounts to assuming that reconvergence points lie at greater addresses than the code that precedes them. Collins, Tullsen and Wang have measured that this assumption was correct for 94 % of conditional branches in SPECint [6]. As current CUDA kernels offer a much more regular control flow, we did not encounter any counter-example in the parallel applications presented Section 5.2, with one major exception. An illustration of such problematic case is presented on Listing 1.
During a call to function `f` inside a conditional block, PCs of threads running the block point to `f`. When the code of this function resides at a higher address than its call site, the $\min(\mathrm{PC}_i)$ policy is inefficient, as it will execute first the code that resides after the end of the conditional block. Reconvergence with threads executing `f` will only be possible at the end of the program or the call to `f` by all other threads.
As early versions of the CUDA compiler would inline all functions calls, this issue has not arose previously.
A first solution consists in ensuring the compiler systematically places function definitions before all their call sites during link-time. Except in case of indirect recursion, this solution is always applicable. However, recompiling or relinking code is not always practical.
A more general solution that we adopt consists in considering the stack pointer (SP_i) of each thread i for the arbitration. To give priority to the deepest function call nesting level, we select the thread whose SP_I is minimal[2].

[2]We assume that the call stack is ordered by decreasing addresses.

In case of a tie, we select the minimal PC_i as before.
Table 2 presents the order of traversal of the code of figure 1, in which jump instructions have been inserted. The values of current individual PCs and Next Program Counters (NPCs) are indicated on each cycle. Active threads have their PC underlined in the table.

Table 2: Execution trace of an `if-then-else` structure

PC	Instruction	PC_i 0 1 2 3	NPC_i 0 1 2 3
1	A	<u>1</u> <u>1</u> <u>1</u> <u>1</u>	2 2 2 2
2	if(!c) br 5	<u>2</u> <u>2</u> <u>2</u> <u>2</u>	3 3 5 5
3	B	<u>3</u> <u>3</u> 5 5	4 4 5 5
4	br 6	<u>4</u> <u>4</u> 5 5	6 6 5 5
5	C	6 6 <u>5</u> <u>5</u>	6 6 6 6
6	D	<u>6</u> <u>6</u> <u>6</u> <u>6</u>	7 7 7 7

4 Hardware implementation

We consider an SIMT processor having a single instruction fetch, instruction decode and instruction scheduling datapath, and several arithmetic units, traditionally named *processing elements* (PE). The architecture examples presented here for illustrative purpose have 4 PEs, although the techniques we propose aim at architectures having 16 to 64 PEs like current GPUs as well.
In all existing SIMT architectures, pipeline latency is hidden using hardware multithreading[3]. Latency is not a critical factor in such throughput-oriented architectures.

4.1 Arbitration

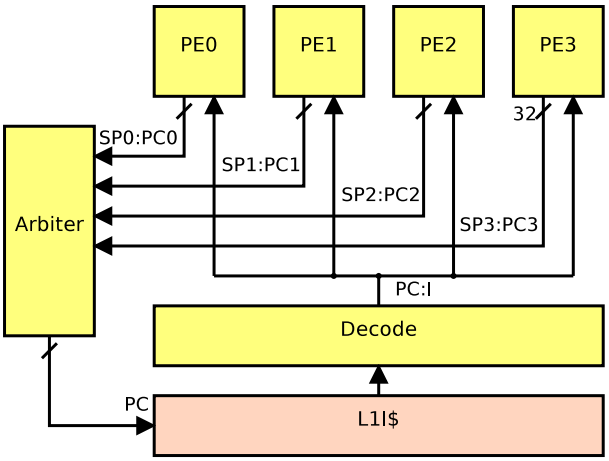


Figure 2: Paths of program counters and instructions in our first implementation. "I" represents the decoded instruction.

Figure 2 illustrates the proposed solution. In its simplest implementation, each PE_i communicates the value of its PC_i and SP_i to an arbiter. This arbiter consists in a reduction tree which computes the minimum $\mathrm{SP} : \mathrm{PC} = \min(\mathrm{SP}_i : \mathrm{PC}_i)$, where : stands for the concatenation operator. The value computed represents the common program counter PC. The instruction pointed by PC is fetched, decoded and broadcasted to all PE_i.

[3]For clarity, descriptions consider one single active thread per PE, but all mechanisms presented in this sections are amenable to generalization with multiple threads per PE, or equivalently multiple warps per processor.

A possible optimization consists in incrementing locally the common PC on each cycle. This way, arbitration is only necessary when an instruction that can affect the program flow (typically a jump instruction) is encountered. Otherwise, the common PC value follows the value of the minimal PC. Indeed, it can be shown that if the PC_i of a non-empty subset of active threads is incremented and the PC_j of all other threads is unaffected, then the new common PC is the incremented value of the former common PC.

4.2 Similarity with the memory access path

Let's consider the L1 data cache access unit in an SIMT architecture such as NVIDIA Fermi [18]. A simplified representation is presented on figure 3.
The L1 data cache offers a single port as wide as a cache line, which is 128 bits in our toy example. During a memory load instruction, each PE_i requests an address A_i. The high-order bits of this address that represent the cache line number are sent to an arbiter. The arbiter selects one of these addresses, using for instance a priority coder, then forwards it to the L1 data cache. On a cache hit, the matching cache line is sent back through a crossbar interconnect. The crossbar routes data to the PEs which requested it, according to the low-order bits of the address. The activity bits from the PEs to the arbiter and the acknowledge bits going back from the arbiter to the PEs are not represented on the figure.
Multiple addresses may point to the same cache line. In the ideal case, all PEs access the same cache line and can share the L1 cache port. In the opposite case, accesses to different cache lines are serialized [19].

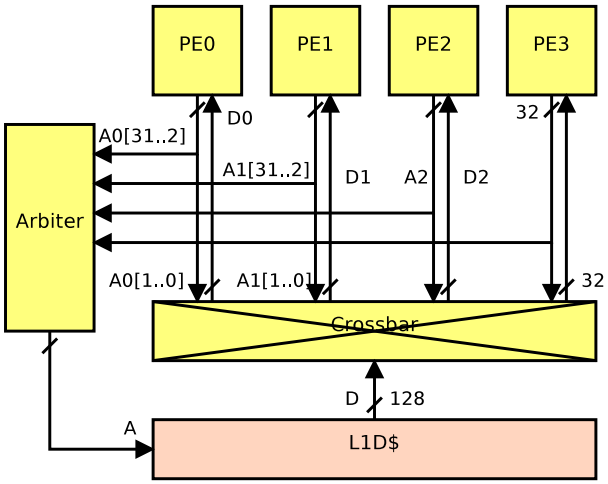
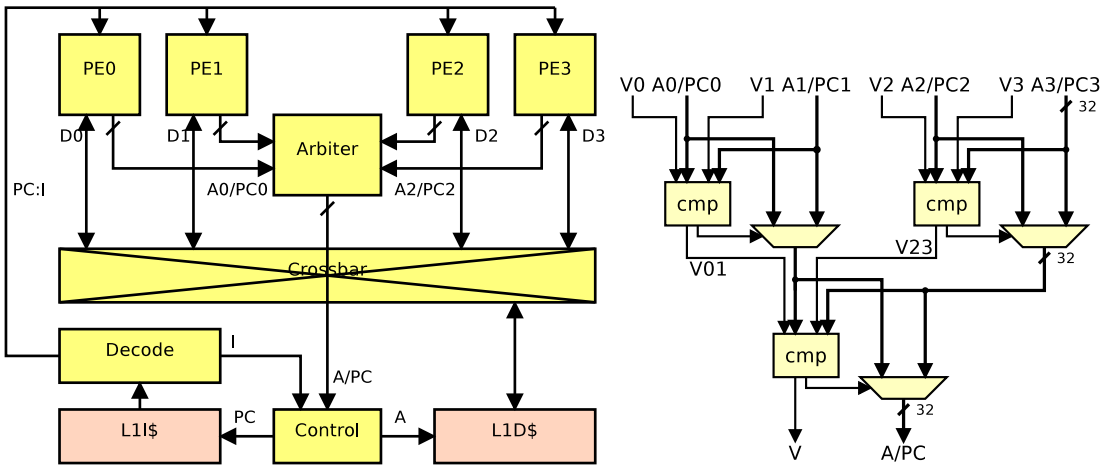


Figure 3: Diagram of the memory access unit of a SIMT architecture.

Bank conflicts and cache misses are dealt with in a straightforward way using this technique. When a thread i encounters a hit in the first cache level, its PC_i is incremented to signal it is ready to execute the next instruction. Threads which encounter a port conflict keep the same PC_i value, causing them to request a replay of the memory operation on the next scheduling phase. Threads which encounter a cache miss are flagged as inactive until data availability. Once data are available, a PC arbitration is performed to give the opportunity to threads that encountered a cache miss to "catch up". This mechanism allows threads that hit the cache to run ahead of threads that encountered a cache miss, providing a feature analogous to Dynamic Warp Subdivision [17] at no additional cost.

4.3 Unified architecture

Let's analyze similarities between the memory access unit just described and the PC arbitration unit we propose. In both units, an arbiter operates a reduction operation between data gathered from all PEs. In this section, we propose a unified architecture which allows to amortize the cost of PC arbitration by re-using and extending the existing memory access hardware.
The cost in terms of latency and power of a reduction/selection tree between PEs is dominated by wire delay and power, rather than the reduction computation itself. We can thus extend the memory access arbitration unit by allowing it to also compute the minimal PC for a marginal extra cost.



(a) Global architecture (b) Multi-function arbiter

Figure 4: Unified architecture handling both control divergence and memory divergence.

The proposed architecture is depicted on Figure 4a. For branch instructions, the arbiter receives the individual PC and SP of each PE and computes their minimum, then updates the common PC maintained by the control unit. For all other instructions, the control unit increments its private copy of the common PC. For memory load and store instructions, the arbiter is used to select the address to request in the data cache.
Figure 4b presents a diagram of the unified arbiter. For each PE, it takes as input a code or data address ($\mathrm{A}_i/\mathrm{PC}_i$), and a valid bit V_i. In PC arbitration mode, the valid bit V_i indicates that thread i is enabled. In memory arbitration mode, it indicates that thread i requests the memory address A_i. An OR reduction operation is performed on individual valid bits to form the valid bit V of the arbitration result. Thus, inactive threads are just ignored.

5 Validation

5.1 Correctness

The control flow graph traversal policy is solely an optimization heuristic. It has no incidence on the correctness of execution. In the worst case assuming no busy-waiting loops, instructions of each threads are executed sequentially on a single SIMD lane and performance is degraded to the level of scalar execution.
The policy that determines the traversal order of the control flow graph is the same as in the technique proposed at SympA'13 [3]. However, reconvergence can happen at additional locations than the reconvergence points selected by the stack-based solution.

5.2 Experimental validation

We modeled the proposed architecture in the Barra GPU simulator [4]. We consider warps of 32 threads as in current NVIDIA architectures. Benchmarks considered are applications from the CUDA SDK [20] as well as the FFT kernel of the UIUC Parboil benchmark suite [22]. Kernels of each application are listed on Table 3 along with their static and dynamic instruction counts.
Figure 5 compares the average number of active threads per warp when using the proposed technique, the baseline NVIDIA implementation and the implicit technique. Simulation of the FFT kernel fails with the implicit technique. This kernel contains function calls that are not supported by the implicit algorithm.
Throughput or latency improvements are not the primary goal of the method we propose. However, we notice an improvement in utilization rate of SIMD units of up to 7% in several applications compared to the baseline NVIDIA implementation, and up to 4% compared to the implicit stack-based solution.
These small differences are explained by the necessity to execute the same instruction several time during reconvergence in stack-based methods. The implicit stack-based technique reduces the number of re-executions; the one we propose eliminates this overhead completely.
In all cases, our method is always at least as efficient as the one employed in the Tesla and Fermi architecture and the solution proposed previously.

Table 3: Kernels of CUDA SDK and Parboil benchmarks used, and statistics on static and dynamic machine instructions.

Program	Kernel	Instructions	
		Static	Dynamic
FFT	Parboil_FFT	431	8771584
fastWalshTransform	fwtBatch1Kernel	108	57212928
	fwtBatch2Kernel	46	54263808
	modulateKernel	24	84246528
BlackScholes	BlackScholesGPU	99	5201694720
transpose	transpose_naive	29	1835008
	transpose	42	2752512
quasiRandomGenerator	inverseCNDKernel	147	3383048
sortingNetworks	bitonicSortShared	100	631696
	bitonicMergeGlobal	43	53760
	bitonicMergeShared	71	212736
	bitonicSortShared1	101	573456
binomialOptions	binomialOptions	109	277266944
convolutionSeparable	convolutionRows	214	39260160
	convolutionColumns	234	42946560
3dfd	stencil_3D_16x16_order8	162	3174912
MonteCarlo	MonteCarloOneBlock...	129	27394560
reduction	reduce5_sm10	40	3900
	reduce6_sm10	59	318513600
dwtHaar1D (DWT)	dwtHaar1D	87	9366
matrixMul (MM)	matrixMul	114	1785600

6 Conclusions and perspectives

We presented a method which handles thread divergence and reconvergence in the SIMT model. It does not require specific support from the compiler and does not affect the instruction set. This method can be implemented for a very low hardware cost, by leveraging the existing datapaths of the memory divergence handling unit.

The control flow graph traversal policy we consider in this paper consists in selecting the minimal $\mathrm{SP} : \mathrm{PC}$. It provides state-of-the-art performance for a low implementation cost. However, it may be advantageous to consider other scheduling policies. In particular, a speculative policy could hide the latency of PC arbitration. In this case, the control unit becomes a branch predictor. Insterestingly, in case of divergence in a conditional jump, the outcome of the branch predictor is always correct, and only affect the order in which paths are visited. Similarly, the PC comparison unit inside each PE becomes the commit unit, which checks that speculative decisions are correct before writing back the execution result to the register file.

Another analogy consists in noticing that the control unit plays the same role as the scalar unit in vector and SIMD computers. In a way similar to these architectures, we can consider including an ALU and a register file to factor out common computations run by several threads, which represent a significant portion of computations and registers [5].

Divergence and reconvergence management techniques also form the necessary foundation for higher-level thread scheduling policies such as Dynamic Warp Formation [9] and Dynamic Warp Subdivision [17]. Moving from explicit stack-based reconvergence to implicit PC-based reconvergence promises to offer more flexibility for such fine-grained thread management techniques.

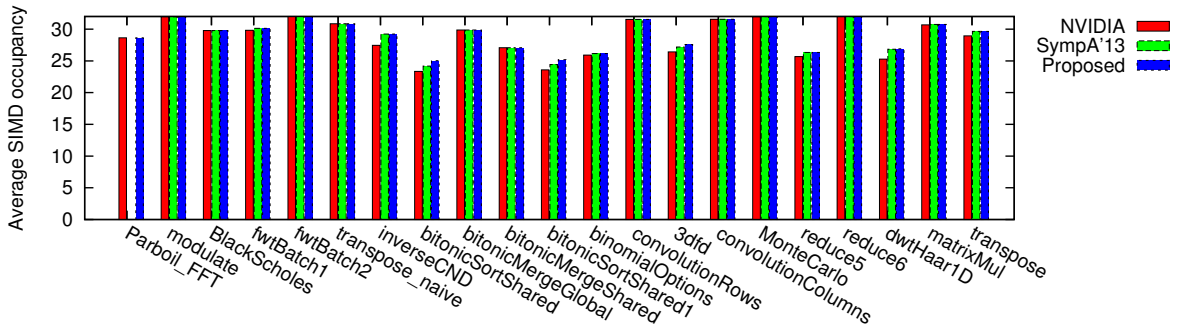


Figure 5: Average number of active thread per warp for the Tesla and Fermi reconvergence technique [7], the one proposed at Sympa'13 [3] and the one we proposed in this paper.

References

[1] AMD. *Evergreen Family Instruction Set Architecture: Instructions and Microcode*, December 2009.

[2] Caroline Collange. Une architecture unifiée pour traiter la divergence de contrôle et la divergence mémoire en SIMT. In *SYMPosium en Architectures*, Saint-Malo, France, May 2011.

[3] Caroline Collange, Marc Daumas, David Defour, and David Parello. Étude comparée et simulation d'algorithmes de branchements pour le GPGPU. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009.

[4] Caroline 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.

[5] Caroline Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, volume LNCS 6043, pages 46–55, 2009.

[6] Jamison D. Collins, Dean M. Tullsen, and Hong Wang. Control flow optimization via dynamic reconvergence prediction. In *IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE Computer Society, 2004.

[7] Brett W. Coon and John Erik Lindholm. System and method for managing divergent threads in a SIMD architecture. US Patent 7353369, April 2008.

[8] Wilson Wai Lun Fung. Dynamic warp formation: exploiting thread scheduling for efficient MIMD control flow on SIMD graphics hardware. Master's thesis, University of British Columbia, 2008.

[9] Wilson Wai Lun Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[11] Intel. *Intel G45 Express Chipset Graphics Controller PRM, Volume Four: Subsystem and Cores*, February 2009.

[12] Intel. *Intel HD Graphics OpenSource PRM Volume 4 Part 2: Subsystem and Cores – Message Gateway, URB, Video Motion, and ISA*, July 2010.

[13] Ronan Keryell and Nicolas Paris. Activity counter: New optimization for the dynamic scheduling of SIMD control flow. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 184–187, 1993.

[14] Adam Levinthal and Thomas Porter. Chap - a SIMD graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82, 1984.

[15] John Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[16] Raymond A. Lorie and Hovey R. Strong. Method for conditional branch execution in SIMD vector processors. US Patent 4435758, March 1984.

[17] 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.

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

[19] NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.2*, 2010.

[20] NVIDIA CUDA SDK, 2010.

[21] Yoshizo Takahashi. A mechanism for SIMD execution of SPMD programs. In *Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, HPC-ASIA '97, pages 529–534, 1997.

[22] UIUC Parboil Benchmarks, 2010. `http://impact.crhc.illinois.edu/parboil.php`.

[23] Fubo Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 30(4):231–245, 2004.