Scheduling paths leveraging dynamic information in SIMT architectures

Lily Blanleuil, Caroline Collange

Compas'2021 : Parallélisme/ Architecture / Système/ Temps Réel
MILC - Lyon, France, 5-9 juillet 2021

Scheduling paths leveraging dynamic information in SIMT architectures

Arthur Blanleuil, Caroline Collange

Univ Rennes, Inria, CNRS, IRISA

Résumé
Thread divergence optimization in GPU architectures have long been hindered by restrictive control-flow mechanisms based on stacks of execution masks. However, GPU architectures recently began implementing more flexible hardware mechanisms, presumably based on path tables. We leverage this opportunity by proposing a hardware implementation of iteration shifting, a divergence optimization that enables lockstep execution across arbitrary iterations of a loop. Although software implementations of iteration shifting have been previously proposed, implementing this scheduling technique in hardware lets us leverage dynamic information such as divergence patterns and memory stalls. Evaluation using simulation suggest that the expected performance improvements will remain modest or even nonexistent unless the organization of the memory access path is also revisited.

Mots-clés : GPU, SIMT, divergence, microarchitecture

1. Introduction

Graphics Processing Units (GPUs) execute multi-thread programs (kernels) on SIMD units by grouping threads running in lockstep into so-called warps. This model is called SIMT (Single Instruction Multiple Threads) [7]. As the multi-thread programming model allows branching, independent threads withing a warp can diverge. To maintain SIMD execution, SIMT processors keep track of divergent paths inside each warp. The execution pipeline selectively enables which threads execute the currently fetched instruction using an execution mask. Therefore divergence hurts performance as it lowers the warp occupancy.
Both software and hardware techniques have been proposed to reduce the divergence or its cost. Software approaches mainly focus on compiler optimization changing control-flow to remove divergence. One of them is Novak's loop scheduling [8] which consists in allowing threads to reconverge inside loops at different iterations to increase warp occupancy.
The model most commonly used in the GPU literature for handling divergence is the SIMT stack [1]. Each time a group of threads diverge, execution masks that correspond to each path are pushed onto the stack, and the top most mask is used untill it reaches a reconvergence point. At a reconvergence point, the current mask is removed from the stack and the second path is executed. This approach constrains path scheduling and reconvergence.
Since the Volta architecture from NVIDIA, the SIMT stack model appears obsolete as the architecture supports interleaved execution of divergent threads on a warp, a technology referred to as Independent Thread Scheduling [9]. To account for these new features, we then use a path table [2], which we presume resembles Volta architecture. Each warp keeps track of every path

taken by its threads, represented by their current PC and execution mask. This allows to select which path to run at any moment.
Using this model, we propose to explore more complex path scheduling policies. In particular, we revisit the loop scheduling technique, so far only applied in software [8] and generalize it to a purely-hardware implementation. We implement it by allowing threads to reconverge inside loops at different iterations to increase warp occupancy.

2. The potential for control-flow divergence optimizations

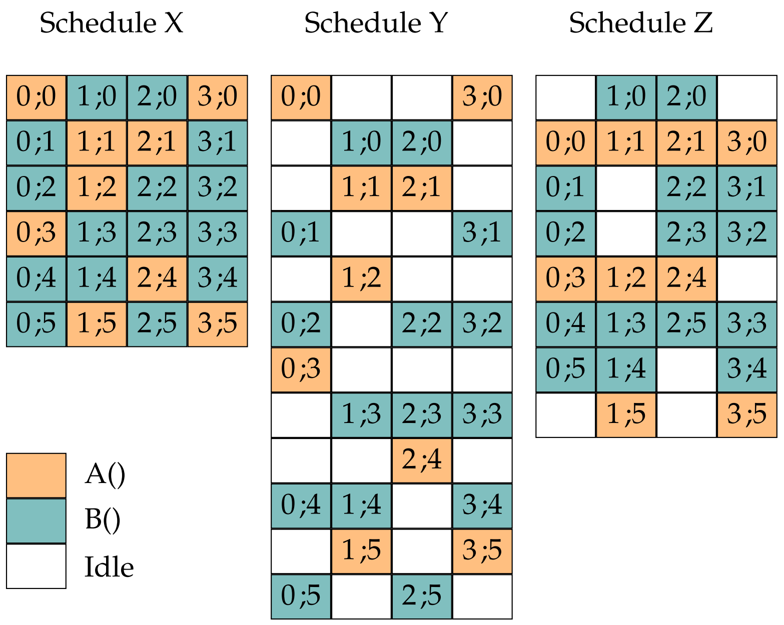
When a group of threads running in lockstep executes a conditional branch instruction, all threads won't necessarily take the same path. They may diverge. As SIMD cannot execute different instructions at once, we carry a mask telling us which thread needs to execute a given instruction.
Traditionally, this mechanism is implemented using an SIMT stack [1]. Whenever divergence occurs, the execution masks of threads taking the branch are pushed onto the stack along with their PC. The warp then executes the topmost path until its reconvergence point, then pops the current path off the stack, continuing with a new path. This simplifies hardware and allows for more parallel hardware threads. Reconvergence points are typically marked statically by the compiler, at the immediate post-dominator (PDOM) of the branch. The immediate post-dominator is the first node in the control-flow graph such that all execution paths from the branch to the exit node go through the node.
The way we schedule paths execution will affect warp occupancy as paths will reconvergence differently, especially with loops. In this case, threads may re-execute the same static instructions (same binary addresses) multiple times. This provides an opportunity to assemble SIMD instructions from threads executing the same static instruction from different iterations. This can be the case when a group of threads advance to the iteration $n + 1$ while the others are still waiting in iteration n. For instance, the algorithm of Figure 1a illustrates a two-sided conditional statement within a loop.
Prior work using software modification to allow mimic Volta's independent thread scheduling has shown that delaying some iterations to enable converged execution across different iterations can lead to better performance [8]. In Figure 1b, we show for instance, the difference between PDOM reconvergence and a schedule which allows iteration shift (Z) on the example of Algorithm 1a.
The PDOM schedule (Y) forces all paths to execute, even though some paths are not populated a lot. This is why each iteration always has the full length of the `then` and `else` part. Novak's idea is to use a majority policy (The Z schedule). This policy tries to maximize the population of an executed path, and allows iteration shifting. As it allows iteration shift, threads which took the `then` path can converge with threads which took the `else` path, increasing its population. The policy also tries to keep a low iteration difference between threads to tackle issues with paths not taken frequently which can hit performances.

3. Loop Aware Scheduler

In this section, we describe our implementation of the Loop Aware Scheduler (LAS), and how we used it to implement Novak's loop optimization scheduling policy in hardware.

```
for i = 0 to 5 do
    if Cond(tid, i) then
        A()
    else
        B()
    end
end
```

(a) Example of a simple loop kernel. When executed by a GPU, it is executed by many threads which have their own ID (`tid`).



Schedule X

🟧 0;0	🟦 1;0	🟦 2;0	🟧 3;0
🟦 0;1	🟧 1;1	🟧 2;1	🟦 3;1
🟦 0;2	🟧 1;2	🟦 2;2	🟦 3;2
🟧 0;3	🟦 1;3	🟦 2;3	🟦 3;3
🟦 0;4	🟦 1;4	🟧 2;4	🟦 3;4
🟦 0;5	🟧 1;5	🟦 2;5	🟧 3;5

Schedule Y

🟧 0;0	⬜	⬜	🟧 3;0
⬜	🟦 1;0	🟦 2;0	⬜
⬜	🟧 1;1	🟧 2;1	⬜
🟦 0;1	⬜	⬜	🟦 3;1
⬜	🟧 1;2	⬜	⬜
🟦 0;2	⬜	🟦 2;2	🟦 3;2
🟧 0;3	⬜	⬜	⬜
⬜	🟦 1;3	🟦 2;3	🟦 3;3
⬜	⬜	🟧 2;4	⬜
🟦 0;4	🟦 1;4	⬜	🟦 3;4
⬜	🟧 1;5	⬜	🟧 3;5
🟦 0;5	⬜	🟦 2;5	⬜

Schedule Z

⬜	🟦 1;0	🟦 2;0	⬜
🟧 0;0	🟧 1;1	🟧 2;1	🟧 3;0
🟦 0;1	⬜	🟦 2;2	🟦 3;1
🟦 0;2	⬜	🟦 2;3	🟦 3;2
🟧 0;3	🟧 1;2	🟧 2;4	⬜
🟦 0;4	🟦 1;3	🟦 2;5	🟦 3;3
🟦 0;5	🟦 1;4	⬜	🟦 3;4
⬜	🟧 1;5	⬜	🟧 3;5

🟧	A()
🟦	B()
⬜	Idle

(b) Different thread schedulings of algorithm 1a. Schedule X, Y, Z represent respectively an MIMD execution, an SIMT execution with PDOM reconvergence, and an SIMT execution with majority policy. Coordinates in each block represent the thread ID followed by the iteration number.

FIGURE 1 – Example of a GPU kernel with a loop, and different execution traces obtained with different scheduling policies on 4 threads. In figure 1b, we can see that schedule Z allows threads to execute different iterations at the same time which makes the schedule more compact.

3.1. Path table

Because SIMT stacks can only be affected by instructions, the only way we can schedule paths is by using the software to manage the stack. This method has a runtime cost if we want to implement complex scheduling policies. This is why we propose to implement it in hardware. In order to do it, we need to relax the traditional SIMT stack model and use a paths table [2].

Each time a divergence occurs, a path is split in 2, and a new path is added to the list. The scheduler then choses a path from this table given a priority function. This priority function can switch between two policies at runtime given dynamic information about the process as the loop table (figure 2).

3.2. Loop detection

As the scheduling policy focuses on loops with conditional branches, we need to detect them in hardware.

We assume loops are entirely defined by an address range between a backward branch and its target [4]. This approximation can lead to non-loop basic blocks inside detected ranges, but this scenario is rare, especially for optimized code. When a backward-jump instruction is encountered, LAS adds the detected loop's entry in the loop table. In the case the loop is already detected we update its information. The loop table is a directly-mapped cache.

Its entries are indexed by a hash of the PC of the loop's backward branch as two loops can not

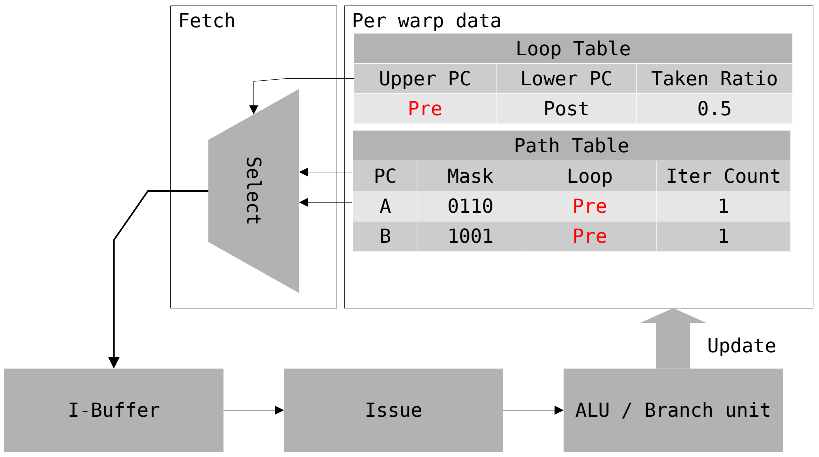


FIGURE 2 – Schematic view of LAS. The state of Per warp data is a snapshot of the example in Figure 1b right after the first diverging branch.

have the same, even in nested loops cases. Whenever a new loop is overlapping a previously detected one, there are two cases. If one loop include the other, we have nested loops so we keep both. Either way, we merge them by removing the backmost entry from the cache. Each entry has a valid bit telling if the data represent a loop or nothing.

3.3. Scheduling policies

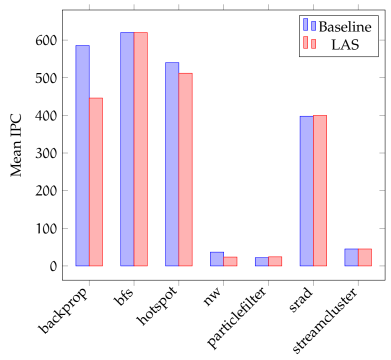
When a loop is detected, information such as trip count and divergence stats is gathered in a table for this particular loop. Loops are indexed by their backward-jump instruction address. Trip count is increased as each individual thread take the backward-jump of the loop. Divergence stats include if there is some divergence inside the loop, and the ratio of taken vs not taken outcomes for each thread. Because we don't use statically defined reconvergence points, we use a Min-PC policy by default (execute the path with the minimal PC), which is the closest from the PDOM. Whenever the path which should be executed is inside a loop, another check is performed based on paths in the same loop and the loop's statistics.
For the second phase, we use the majority policy, biased with threads maximum iteration count. More information is kept inside the loop table for future policies. Taken ratio is the ratio of taken/not taken for the first branch.

4. Experiments and results

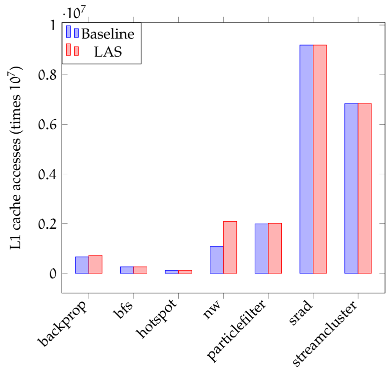
4.1. Experimental setup

We tested LAS on the Rodinia CUDA benchmark suite. We implemented LAS on the GPGPU-Sim [5] simulator with a GTX480 configuration. For comparison, the unmodified simulator was used as the baseline. It features the same architecture but with a classic SIMT stack instead of a path list.
We measured the mean number of instructions executed per cycle (IPC) to evaluate performance and plot it on Figure 3a. We also evaluate the number of accesses to the L1 data cache in Figure 3b and the L1 data cache miss rate in Figure 3c.
We measured the mean number of instructions executed per cycle (IPC) because it allows us to compare performances between architectures. We also show memory accesses and L1 cache miss rate because it is the main factor of performance difference.

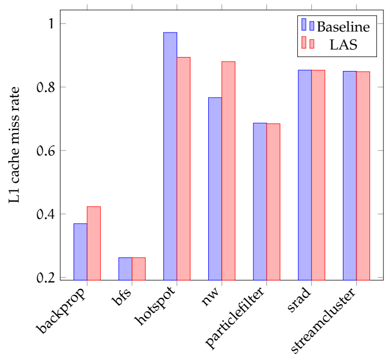
We could not compare LAS with an oracle which would give us a perfect (or near perfect) scheduling because this scheduling problem is too computationally-intensive [6]. We found that the oracle fails to compute a schedule withing a week even with a short trace of 20 iterations on 32 thread-wide warps.



(a) Mean IPC per benchmark



(b) Number of L1 cache access per benchmark



(c) L1 cache miss rate per benchmark

FIGURE 3 – Results of benchmarks executed with the baseline architecture and LAS

4.2. Discussion

Figure 3a shows an equal or lower performance for LAS, due to equal or lower warp occupancy. Kernels with no difference are kernels with no loops, or loops with no divergence. Because LAS does not trigger iteration shift in the latter case, it keeps the Min-PC scheduling policy and is equivalent to the static scheduling policy of the baseline.

We observe a significant slowdown in the *backprop*, *hotspot*, and *nw* benchmarks (Figure 3a). All these benchmarks effectively trigger iteration shifting.

The *backprop* and *nw* benchmarks show the problem of iteration shifting on cache miss rate. Shifting iterations makes kernels access memory in a non-contiguous manner, resulting in more cache misses (Figure 3c). As the scheduling algorithm implemented in GPGPU-Sim does not take into account stalled warps because of cache misses, extra pipeline stalls due to memory lower the IPC. Secondly, conditional paths in these kernels are very unbalanced in term of execution time, because they are of the *if-then* form. The iteration overhead accumulated by shifting has more impact than with balanced branches.

hotspot has fewer cache misses overall with sensibly the same amount of cache accesses (Figures 3b, 3c). It is due to a lower warp occupancy with LAS. Some instructions are executed by divergent thread groups, so early threads trigger a cache miss but not later threads which arrive after the cache is populated.

4.3. Future work

For now, iteration shifting does not show performance improvements at least in the Rodinia benchmark suite. We highlighted metrics which can badly impact performance and explain why the LAS technique did not do well. In order to fix these issues, we will need to modify the inclusion criteria for loops using dynamic information such as cache statistics.

For now the whole warp is waiting when a memory access occurs. Having a path table should

also allow the architecture to advance paths in a warp which are not waiting for a memory instruction. Implementing this behavior should reduce impact of memory on performances.
We also aim at testing LAS with more divergence-heavy CUDA kernels, because the kernels we considered were mostly not using iteration shifting policy.
One kind of divergence we did not cover is the termination divergence [3]. Kernels with very regular loops often have conditional branches which selects which thread needs to do heavy computation or not. Threads with no work could be recycled by starting new blocks early, thus increasing warp occupancy.

5. Conclusion

SIMT architectures based on path tables open new opportunities to implement path scheduling policies at the microarchitecture level. Implementing them in hardware lets us leverage dynamic information such as divergence patterns and memory stalls.
In this paper we leverage this opportunity for a hardware implementation of the iteration shifting algorithm, which was reported to be efficient in software [8]. We implement the algorithm by modifying the classical Post Dominator path scheduling policy for loops that contain divergence. This scheduler was tested against a Fermi architecture GPU (GTX480). The simulations highlighted several technical issues preventing performance gains, both due to actual limitations of the architecture and to artifacts of the simulation platform and benchmarks. The insights obtained suggest that special care must be taken to memory access patterns when implementing iteration shifting optimizations.

Bibliographie

1. Brunie (N.) et Collange (C.). – Reconvergence de contrôle implicite pour les architectures SIMT. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, vol. 32, n2, février 2013, pp. 153–178.
2. Collange (C.) et Brunie (N.). – Parcours par liste de chemins : une nouvelle classe de mécanismes de suivi de flot SIMT. – In *ComPAS 2017 - Conférence d'informatique en Parallélisme, Architecture et Système*, Sophia Antipolis, France, juin 2017.
3. Frey (S.), Reina (G.) et Ertl (T.). – SIMT Microscheduling : Reducing Thread Stalling in Divergent Iterative Algorithms. – In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 399–406, février 2012.
4. Gordon-Ross (A.) et Vahid (F.). – Frequent loop detection using efficient nonintrusive on-chip hardware. *IEEE Transactions on Computers*, vol. 54, n10, octobre 2005, pp. 1203–1215.
5. Khairy (M.), Shen (Z.), Aamodt (T. M.) et Rogers (T. G.). – Accel-sim : An extensible simulation framework for validated gpu modeling. – In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 473–486, 2020.
6. Milanez (T.), Collange (S.), Quintão Pereira (F. M.), Meira (W.) et Ferreira (R. A.). – Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads. *Parallel Computing*, vol. 40, n9, octobre 2014, pp. 548–558.
7. Nickolls (J.) et Dally (W. J.). – The GPU Computing Era. *IEEE Micro*, vol. 30, n2, mars 2010, pp. 56–69.
8. Novak (R.). – Loop Optimization for Divergence Reduction on GPUs with SIMT Architecture. *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, n6, juin 2015, pp. 1633–1642.
9. NVIDIA. – *NVIDIA Tesla V100 GPU Architecture*, 2017.