SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms

Steffen Frey, Guido Reina and Thomas Ertl
Visualization Research Center, University of Stuttgart (VISUS)
Allmandring 19, 70197 Stuttgart, Germany
{steffen.frey|guido.reina|thomas.ertl}@visus.uni-stuttgart.de

Abstract

The global scheduler of a current GPU distributes thread blocks to symmetric multiprocessors (SM), which schedule threads for execution with the granularity of a warp. Threads in a warp execute the same code path in lockstep, which potentially leads to a large amount of wasted cycles for divergent control flow. In order to overcome this general issue of SIMT architectures, we propose techniques to relax divergence on the fly within a computation kernel in order to achieve a much higher total utilization of processing cores. We propose techniques for branch and loop divergence (which may also be combined) switching to suitable tasks during a GPU kernel run every time divergence occurs. Our newly introduced techniques can easily be applied to arbitrary iterative algorithms and we evaluate the performance and effectiveness of our approach exemplarily via synthetic and real world applications.

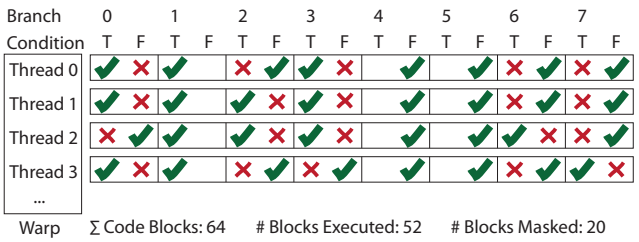
1 Introduction

Current GPUs feature multiprocessors with SIMT architecture which create, schedule, and execute threads in groups called warps. Threads of a warp execute the same instructions (lockstep), but they have their own register state and masking is employed in order to allow divergent control flow. Thus, full efficiency is only realized when all threads of a warp take the same execution path. Three types of issues can be distinguished that lead to wasted computation cycles on GPUs:

Memory Divergence A subset of threads of a warp perform costly memory accesses and stall, forcing the other threads to idle.

Termination Divergence Terminated threads waste compute cycles until all threads of their warp are finished with their computation steps (Fig. 1(a)).

(a) *Termination Divergence:* All threads run until every thread of the warp is finished with all its computation steps.



(b) *Branch Divergence:* If at least one threads needs to execute a branch, all threads need to step through it.

Figure 1. *Threads of a warp are executed in lockstep, which results in termination divergence 1(a) and branch divergence 1(b).*

Branch Divergence A warp serially executes each branch path taken, masking threads that are not on that path (Fig. 1(b)).

In this paper, we aim to eliminate the wasted cycles caused by branch and termination divergence. We distinguish between the two because different approaches need to be taken to account for the occurring issues.

The terms *task* and *task context* are fundamental in the discussion of our approaches in the following. A task uniquely identifies a job, thus denoting the information necessary for initialization. This is typically determined from the thread id in parallel environments. Our approaches for termination divergence are based on threads fetching new tasks upon finishing their old task. A task context contains the complete description of the state of a computation,

which is necessary for storing and resuming jobs at certain positions in the code. Branch Divergence is tackled by utilizing the same condition for all threads (i.e. all threads take the same path) and by allowing threads to switch to a fitting task context on the fly to minimize wasted clock cycles.

In this work, we assume that tasks are independent. Considering task dependencies would basically be possible with our technique with a few extensions, but remains for future work. We will use the CUDA terminology (NVIDIA 2011) in the following: *global memory* denotes graphics memory, *shared memory* is on-chip memory and the term *thread block* specifies groups of warps which can exchange data using shared memory. Note that the warp size can vary with the employed hardware architecture and without loss of generality we assume it to be 32 in the following as it is the case for NVIDIA GPUs.

2 Related Work

A technique called *Persistent Threads* dealing with strongly varying iteration counts has been introduced by (Aila and Laine 2009) in the context of ray tracing. They launch just enough threads to occupy the hardware and allow thread blocks to fetch new tasks from a task pool in global memory. A modification of Persistent Threads to fetch tasks with thread granularity is also suggested briefly to improve the handling of termination divergence, but the authors note that it did not prove to be beneficial in their scenario. (Tzeng et al. 2010) also employ persistent threads to address irregular parallel work in their GPU task management system. In contrast to persistent threads that only balance workload between warps, we additionally balance workload between threads of a warp. (Novák et al. 2010) propose an application-specific strategy that tackles the problem of terminated rays leaving threads in an idle state in a GPU-based path tracer by dynamically generating new rays that result in improved image quality and higher ray throughput.

Moreover, several generic software approaches targeting branch divergence have been proposed. (Han and Abdelrahman 2011) target the problem of divergent branches within loops and propose a software solution that only executes one branch per loop iteration, proposing different strategies for choosing and switching the active branch. (Zhang et al. 2010) handle conditional branches by runtime data remapping between threads using a CPU-GPU pipelining scheme. Besides branches within loops, their approach can also handle differing loop iterations counts. However, in contrast to our purely GPU-based technique, the iteration counts need to be known beforehand and several kernel calls are required.

Another line of research concerns itself with finding hardware solutions for the divergence problem. (Aila and Karras 2010) focus on handling incoherent rays efficiently. (Meng et al. 2010) tackle cache-hit-induced divergence in memory access latency by utilizing several independent scheduling entities that allow divergent branch paths to interleave their execution. (Fung et al. 2007) dynamically regroup threads into new warps on the fly following the occurrence of a divergent branch, which is in principle similar to our branch divergence resolving technique. Other approaches for branch divergence include compiler-generated priority (Lorie and Strong 1984), hardware stacks (Woop et al. 2005), task serialization (Moy and Lindholm 2005), regrouping (Cervini 2005) and micro-kernels (Steffen and Zambreno 2010).

3 Tackling Branch Divergence: Task Context Switching

The basic idea to avoid branch divergence and the resulting serialization of execution paths is to choose only one (active) branch for execution and subsequently switching task contexts such that preferably all threads are active (i.e. no clock cycles are wasted) during the execution of that branch (Fig. 2). The active branch is determinined by the largest amount of task contexts that need to execute either of the branches (similar to (Han and Abdelrahman 2011)). Subsequently, we increase this amount by swapping task contexts. Branches which we apply our technique to are denoted as *managed branches* in the following. We limit ourselves to *if-then-else* statements in the following discussion without loss of generality.

3.1 Task Contexts

Task contexts can be attached to and detached from threads whereas exactly one task context is active for a thread at any time. Task contexts that are currently not attached are shared amongst threads of a warp through the *task context pool* residing in shared memory. The *branch map* contains a list of references to the task context pool for each managed branch and each possible condition leading to a different branch there (typically *true* and *false* for an *if*-statement). The branch map is used when switching task contexts to provide the information which contexts can be loaded that fit the upcoming branch path.

Besides the task-specific information a task context contains, it also features an integer giving information about the current state of the context (0: Active; 1-254: Temporarily invalidated before a managed branch with the respective number; 255: Permanently invalidated). Task contexts are invalidated temporarily if the branch path it actually needs to execute does not match with the executed branch path. Temporarily invalidated task contexts are not

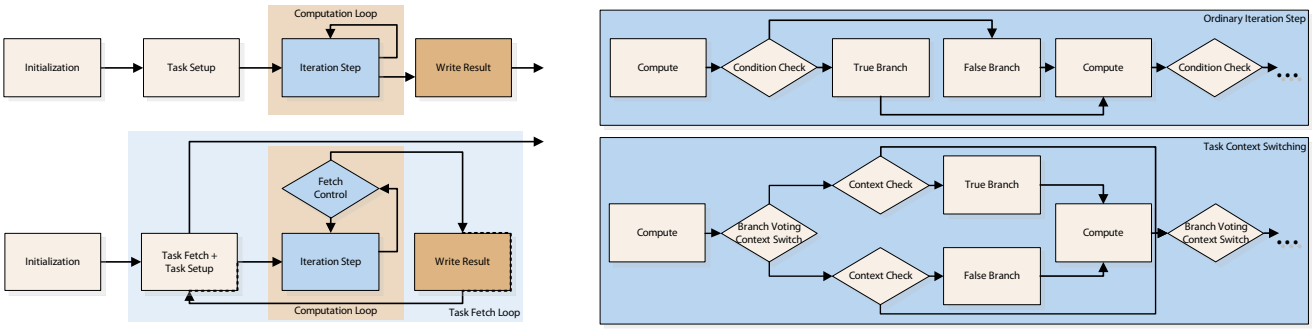


Figure 2. *The basic kernel structure of an ordinary iterative application (top left) and a generic iteration step (top right).(Bottom left) and (bottom right) show our (simplified) modified versions tackling termination divergence and branch divergence respectively.*

allowed to vote on any upcoming managed branches until the initially diverging branch is reached again in order not to corrupt code semantics. Task contexts are invalidated permanently when the respective task is completed. A certain number of task contexts (typically two to four times the warp size) is initialized at the very start of a kernel with distinct tasks (similar to the local task fetching strategy discussed int Sec. 4.2).

3.2 Task Context Switching

In order to determine the branch path with the highest saturation, votes on the upcoming branch consider both currently attached and detached task contexts. Threads not agreeing with the upcoming branch path attempt to switch their current task context with one that fits the upcoming path. The references to these detached task contexts are looked up from the branch map. Whether a task context for a certain thread is available (and which one) is determined by a continuous load offset that is unique to any loading thread and starts from zero. The load offset is determined efficiently from bit masks involved in the voting process (refer to Listing 1 for details). If the load offset exceeds the amount of detached task contexts available for the upcoming branch, the thread cannot switch its context and the current context is temporarily invalidated until this managed branch is reached the next time. A distinct store offset needs to be determined for storing a task context as threads with permanently invalidated task contexts only load but do not store contexts (i.e., permanently invalidated task contexts are discarded). The kernel is exited when there are no more contexts referenced in the branch map and all attached contexts are permanently invalid.

3.3 Deferred Context Switching

In cases in which a single iteration step is cheap, the task context switching procedure might introduce significant overhead. *Deferred context switching* can circumvent this issue by carrying out the task switching procedure every *n*th iteration only and using a pre-defined voting outcome otherwise. Task contexts not complying with the predefined outcome are temporarily invalidated. The default vote and the *n* need to be adapted by the programmer to the problem at hand.

	none	local	global	hybrid	persistent
collaboration	thread	block	thread	warp	block
atomics	none	shared	global	both	global
coherency	high	high	low	medium	high
grid size	task	task	gpu	gpu	gpu

Table 1. *Feature overview of our task fetching strategies and others.*

4 Tackling Termination Divergence: Work Distribution Approaches

Our approach to alleviate termination divergence is to fetch new tasks for threads finished with their old ones.

4.1 Task Pools

Tasks are fetched from *task pools* which are organized hierarchically and can be distinguished in terms of the group of threads which have collaborative access to it. All threads have access to the *Global Task Pool* (in global memory), while only threads of the same thread block or warp (depending on the technique) have access to the same *Local Task Pool* (in shared memory). The *Private Task Pool* (in register space) may be used only by one thread. Since tasks are embodied by monotonically increasing, continuous integers, all task pools are represented by two counters, one for the current task and one for the last available task. Tasks are cached from the global task pool to the local or the private task pool in *chunks* to reduce costly global memory

accesses. The amount of tasks fetched from a higher level task pool at a time is controlled by the user-defined *chunk size* parameter.

While tasks from global task pools are fetched using atomic additions in global memory, local task pools can either be accessed using atomic additions in shared memory or a combination of ballot and popc ((NVIDIA 2011) for details on these functions, we used the latter implementation in the results).

4.2 Task Fetching

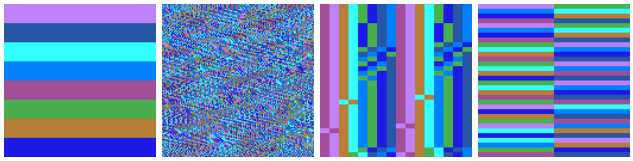
We introduce three basic work distribution techniques (Local, Global and Hybrid), which can be distinguished by the employed task pool hierarchy (Fig. 4). The distribution of tasks logged from an actual run can be seen in Fig. 3 and characteristics of different work distribution techniques are summarized in Table 1.

Local

The local work acquisition strategy only uses local task pools (Fig. 4 *left*) with one being assigned to each thread block. Task pools are statically initialized before device kernel invocation with *#chunk size* tasks and no transfer of tasks from or to other task pools is executed. As the local task pool only contains a small subset of all tasks, high divergence cannot be cushioned as smoothly as with a global pool. However, task fetching is cheap as no operations on global memory are required. The amount of started thread blocks is determined by the total amount of tasks divided by the chunk size.

Global

The global task fetching strategy uses one global task pool (containing all tasks) and one private task pool per thread (Fig. 4 *middle*). When its private task pool is empty (Locality Check), a thread attempts to refill it with tasks from the global task pool. The number of transferred tasks depends on the user-defined chunk size. If the private task pool is still empty after the global task fetch (because of a lack of tasks in the global task pool) the thread exits the task fetch loop. Especially for small chunk sizes, this strategy allows a very fine-granular distribution of tasks leading to very good iteration length divergence compensation. However, this also introduces high costs due the large amount of global memory accesses required to refill the private task pools. Additionally, this leads to a strong scattering of tasks that are processed by threads of the same warp. This might have a negative impact if this results in scattered memory accesses (Fig. 3(b)). Unlike the local strategy, the number of thread blocks which are created at the beginning of the kernel invocation is independent of the actual problem size but is chosen such that the device can be fully occupied.



(a) Local, Colored Blocks (b) Global, Colored Warps (c) Hybrid, Colored Warps (d) Persistent, Colored Blocks

Figure 3. *Color depicts warp or block membership of a thread per pixel of a 128x128 image with different work distribution techniques (computed from Fractals example from Fig. 6, block consists of eight warps). Note that only the persistent technique starts the processing of tasks simultaneously for all threads of a block.*

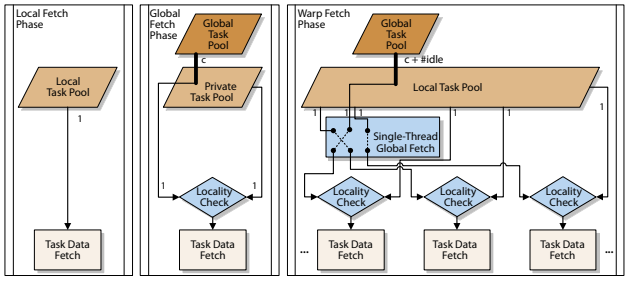


Figure 4. *Work acquisition for the local, global and hybrid strategies (from left to right). c denotes the chunk size.*

Hybrid

The hybrid strategy uses both local and global task pools (Fig. 4 *right*). Threads which are out of work attempt to fetch new tasks from the local task pool. If threads cannot get a new task due to the local task pool being empty (Locality Check), the thread that gets the smallest task id beyond the valid range transfers tasks from the global to the local task pool (Single-Thread Global Fetch). The amount of fetched tasks is determined by the number of threads which are currently out of work plus the chunk size. As the local task pool in the hybrid technique is shared by a warp, all involved threads run in lockstep which avoids expensive thread block level synchronization calls after updating the local task pool. This technique uses significantly fewer global memory fetches and task locality within a warp can be preserved to a large extent (Fig. 3(c)). Similar to the global method, the amount of thread blocks generated only depends on the targeted hardware. Initially, the local task pool is empty.

4.3 Task Fetch Control

The *task fetch control* determines when threads exit the computation loop to fetch new tasks (Fig. 2). We experimented with the following variants:

Any Leave the loop as soon as any thread is out of work

All Only leave the computation loop when all threads are finished with their tasks (similar to the persistent threads technique).

Ballot/Atomic Exit computation loop when a user-defined number of threads is finished. (Ballot uses the warp vote function, while atomic uses atomicAdd to determine the amount of idle threads).

Leaving the computation loop early (*Any*) leads to the masking of result writing and task fetching. Exiting the computation loop late (*All*), however, induces the masking of iterations of the computation loop, which means that it does not alleviate iteration divergence.

5 Results

Our techniques are evaluated by means of a synthetic Monte Carlo program (Sec. 5.1, termination and branch divergence) and two real world applications: Fractals (Sec. 5.2, termination divergence) and isosurface ray casting (Sec. 5.3, branch divergence). We concentrate on removing computational divergence only and try to avoid other effects influencing the timing results like caching in memory accesses as much as possible (this remains for future work).

All tests were run on an NVIDIA GeForce GTX580 using CUDA 4.0. Reasonable chunk sizes for tackling termination divergence were determined empirically, ranging between 256 and 12000 for *Local*, 1 and 8 for *Global* and 0 to 128 for *Hybrid*. The number of valid idle threads for *Ballot* and *Atomic* techniques was varied between 1 and 32. Variants of techniques resolving iteration divergence are denoted in the form [Task Fetching Technique] [Task Fetch Control Technique]. For resolving branch divergence, the most beneficial amount of task contexts per warp was empirically determined to be 160 for Monte Carlo and 64 for ray casting.

5.1 Monte Carlo

In each iteration of this generic, synthetic test case, a random number is generated using the Sobol32 generator of NVIDIA's CURAND Library. Depending on the testing scenario, this number either decides when to exit the computation loop (for termination divergence) or when a branch is entered that sums up several random numbers within an inner loop (for branch divergence). In both cases, a single iteration is cheap, but there can be significant overhead for stepping through a masked branch path (branch divergence) or having a large iteration count difference between threads of a warp respectively (termination divergence).

Fig. 5 *(left)* shows the results for different scenarios with increasing termination divergence through a decreasing kill probability. Techniques that do not waste computation iterations by leaving the computation loop early (*Any* and *Ballot*/*Atomic* with a low amount of idle threads) perform best with speedups up to factors of 4 to 5. In general, speedups increase with divergence. As new tasks are assigned immediately to idling threads they most effectively cope with significant termination divergence. With a kill ratio of 0.1 (meaning low divergence and few iterations) it shows that task fetch overhead can also be an important factor, especially when the computation steps are cheap like in this scenario. In particular *Global Any*—which for higher load divergences is very close to our best result—suffers from the overhead of frequently accessing the global task pool, especially because many accesses to the pool happen simultaneously. Conversely, *Hybrid Any* and *Hybrid Ballot* are even able to achieve a notable speedup of almost 1.2 in this scenario due to their small overhead but nonetheless good scheduling quality. Techniques that do not address thread divergence (*All* and *Persistent*) do not achieve any speedup.

Fig. 5 *(right)* shows the performance of the branch divergence test case with increasing probability to enter the computation branch—both the loop surrounding and the loop inside of the branch run 1000 iterations. The effect of branch serialization and the resulting negative performance impact show clearly in the vanilla case: there is no performance difference between one thread entering the branch or all threads entering the branch. At least one thread enters the branch reliably in each iteration from a branch probability of ≈ 0.1. In contrast, the runtime with our technique almost linearly scales with the actual work that needs to be done. Speedups of up to 15 were achieved with our technique for low branch probabilities. If the the branch probability is higher than 84% however (i.e. the efficiency loss due to divergence is low), the usage of our technique is not beneficial and even results in slower runtimes due to the introduced overhead.

5.2 Fractals

This test case evaluates our techniques for resolving termination divergence by computing Julia set fractals with a resolution of 2048×2048 using the implementation from the CUDA SDK. The Julia set is computed iteratively by evaluating a formula for every pixel until it either converges or the maximum amount of iterations (specified by

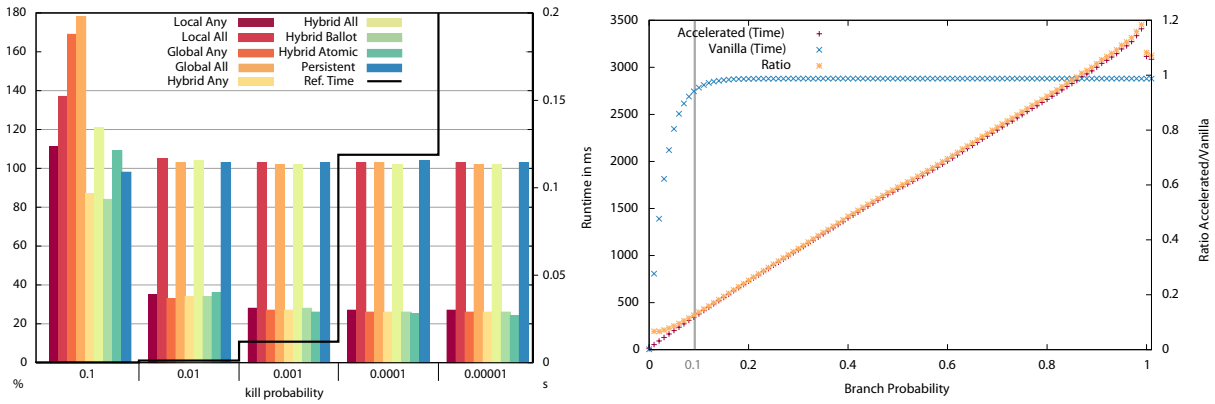


Figure 5. *The Monte Carlo testing scenario for termination (left) and branch divergence (right). Termination digergence results are given relative to the vanilla variant (=100) whose absolute values are depicted in seconds by the black line (right y-axis). Measurements were performed over a variety of chunk sizes, only the results for the best test case are depicted. Right: Branch divergence was measured with varying probabilities for entering the branch.*

the *crunch factor*) is exceeded. Besides the crunch factor, we also modify the precision (single or double) and offset parameters that influence the distribution of necessary iteration counts from strongly divergent (offset 0) to constant (offset 2) (Fig. 6). This allows us to cover the most important execution characteristics of parallel programs: iteration cost is steered by the precision (computations with double precision take significantly longer), iteration count depends on the crunch factor and iteration count distribution is a function of the offset parameter.

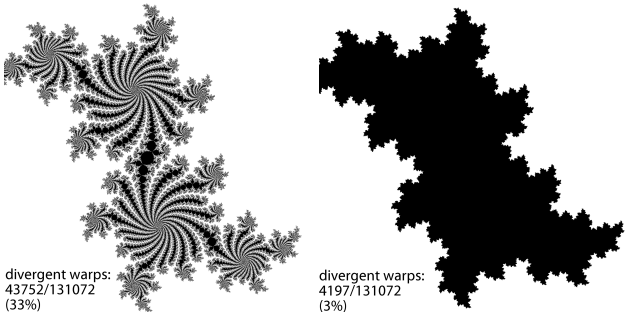


Figure 6. *The amount of iterations (black = high) necessary for computing a pixel in the Julia set with different offsets (left: offset 0, right: offset 1). The divergent warp counts are based on a vanilla kernel with* 8×4 *warp tiles. Note that offset 2 would result in a black box.*

Our measurements show that fetching new tasks immediately when a task is finished (Any and Ballot/Atomic with minimal idle thread threshold) is most beneficial in almost all scenarios (Fig. 7, *top*). When divergence is low, thread level work distribution is largely reduced to warp level work distribution (like *Persistent* and *All*) leading to similar speedups across all techniques doing task fetching. Our *Persistent Threads* implementation in the framework and the original implementation actually perform equally well when starting them with the same amount of blocks. The difference in the diagram results from the fact that we spawn three blocks per SM, while we left the original implementation completely untouched (one block per SM). We believe that this effect is due to a better occupancy with more active warps per SM that can be used for latency hiding.

Naturally, the speedup potential is higher for more divergent cases (julia offset 0 allows for higher speedups than julia offset 1). In the constant case (julia offset 2), there are no speedups since no cycles are wasted by the vanilla variant. The difference bars (Fig. 7, *bottom*) show that once determined good settings for chunk size and idle threads remain close-to-optimal across a wide range of scenarios. Across all our tests (including Monte Carlo), chunk sizes favoring the most flexible task distribution (at the expense of higher task fetching costs) proved to be most beneficial (small chunk sizes for *Global* and *Hybrid* as well as large chunk sizes for the static task pool of *Local*). Similarly, the best idle thread granularity setting is 1 across all scenarios for *Atomic* and *Ballot*, basically reducing both to *Any*.

5.3 Raycasting Isosurfaces

Significant branch divergence occurs in our last test case: multi-layer isosurface raycasting of scalar 3D functions. Rays are cast from the view point into the function domain. When a ray intersects one of eight isosurfaces (a surface representing all points of a constant value), it determines an accurate intersection position using bisection. For rendering the isosurface (which has a distinct color), the first and second order derivative are evaluated numerically at

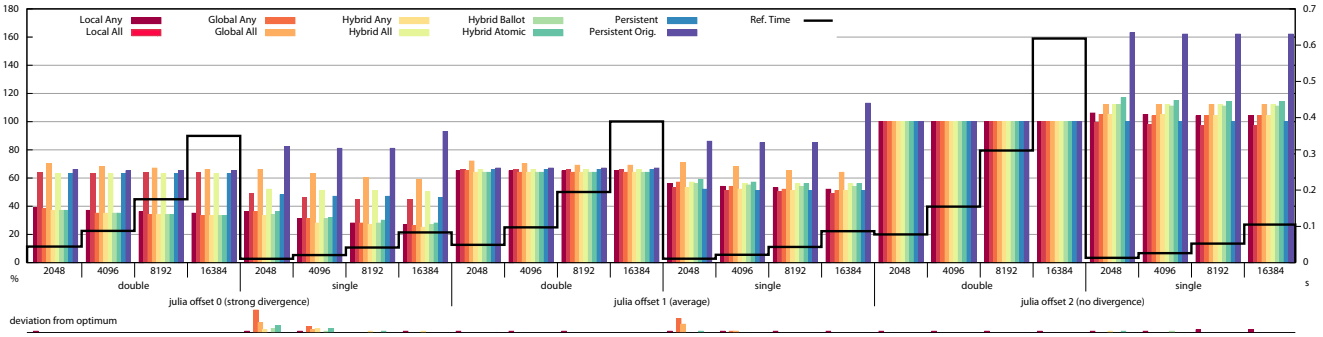


Figure 7. *Timing results in relation to vanilla technique (=100, absolute values depicted in seconds by the black line (right y-axis)) the Julia set test scenarios. Persistent Orig. refers to the unmodified implementation of the CUDA SDK, while Persistent refers to our implementation using more thread blocks. The top bars depict the best results for a given scenario across a wide range of chunk size and idle thread settings while the bottom bars show the difference between these and the best-performing constant parameter settings across all test scenarios.*

this position using the SOBEL operator. Blinn-Phong shading is employed for lighting using the first order derivative (i.e. the gradient), while the opacity of the isosurface is calculated using the magnitude of the second order derivative (Fig. 8 *(left)*, similar to (Parker et al. 1998) and (Hadwiger et al. 2005) among others). Branch divergence incurs from rays hitting isosurfaces after a different amount of volume sampling iterations, which forces threads to step through the isosurface rendering procedure significantly more often than actually needed.

We measured speedups of up to 7 using our proposed technique (Fig. 8 *(right)*). By increasing the sampling distance (the step size), higher performance is achieved but at the cost of lower precision. More divergence and thus higher speedups were measured with smaller step sizes because fewer rays of a warp hit an isosurface at the exact same step count. In contrast to the Monte Carlo test case, using deferred context switching in this test case (with $n = 32$) achieved speedups of up to 30% towards the standard branch divergence alleviation technique. As expected, it is more beneficial for small step sizes as there is a large amount of iterations in which no isosurface is hit, resulting in a relatively large computational overhead of our framework.

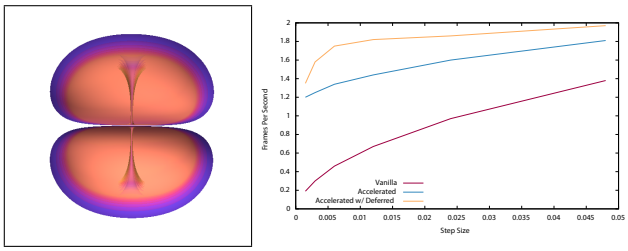


Figure 8. *Left: Raycasting the hydrogen atomic wave function $\psi_{3,2,1}$ with a resolution of 512×512 and on-the-fly evaluation. Right: Timing results with varying step sizes.*

6 Conclusion

Current SIMT architectures potentially waste a large amount of clock cycles due to diverging execution paths of threads within a warp. We introduced techniques to resolve both branch and termination divergence. Previously presented generic software techniques targeting this issue have significant shortcomings, either they are not fine-granular enough to really resolve the issue (e.g. persistent threads only fetch additional tasks at the warp level (Aila and Karras 2010)) or induce significant overhead, several kernel calls and the offloading of scheduling to the CPU (e.g. (Zhang et al. 2010)). We achieve sizeable speedups both in synthetic test cases and real world scenarios, even for code that has been highly optimized already (between $1.5\times$ and $3\times$ for fractals compared to tweaked persistent threads, up to $7\times$ in isosurface raycasting). We implemented all of our techniques in a framework that is minimally invasive and thus easy to apply to existing algorithms. It further directly supports empirically evaluating the best acceleration parameters for a specific problem by automatically generating performance test cases for any parameter combination in a certain range.

For future work, we plan to conduct a more detailed analysis of memory coherency effects. Furthermore, we also intend to introduce additional features that can easily implemented on top of task fetching. These include the generation of tasks inside a kernel—which are then processed in the very same kernel call—and the assignment of priorities to tasks. Additionally, we plan to make the framework publicly available.

Appendix

```
voteMask v = ballot([Enter if branch])
invalidMask i = ballot(context.invalid > 0)
consensus c = popc(v & ∼i) + |pool[voteTrue]| >
              popc(∼v & ∼i) + |pool[!voteTrue]|

if [Enter if branch] != c or context.invalid > 0)
 v = adjust(v, c)
 loadOffset = popc((∼v | i) << threadId)
 // if other context available switch, else invalidate
 if loadOffset < |pool[c]|)
  storeIndex = |pool[!c]|+threadId
  pool[!c][storeIndex] = context
  loadIndex = |pool[c]|−loadOffset
  context = pool[c][load]
 else if context.invalid == 0
  context.invalid = branchNumber
if popc(i) == warpSize and |pool| == 0
 exit_computation()
```

Listing 1. *Simplified implementation of vote and switch (not considering permanently invalidated contexts). Shared memory is used for task context pool/branch map.*

References

AILA, T., AND KARRAS, T. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, 113–122.

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*, 145–149.

CERVINI, S., 2005. European Patent EP 1531391 A2: System and Method for Efficiently Executing Single Program Multiple Data (SPMD) Programs.

FUNG, W. W. L., SHAM, I., YUAN, G., AND AAMODT, T. M. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 407–420.

HADWIGER, M., SIGG, C., SCHARSACH, H., BHLER, K., AND GROSS, M. 2005. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3, 303–312.

HAN, T., AND ABDELRAHMAN, T. S. 2011. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ACM, New York, NY, USA, GPGPU-4, 3:1–3:8.

LORIE, R. A., AND STRONG, H. R., 1984. US Patent 4,435,758: Method for conditional branch execution in SIMD vector processors.

MENG, J., TARJAN, D., AND SKADRON, K. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, 235–246.

MOY, S., AND LINDHOLM, E., 2005. US Patent 6,947,047: Method and System for Programmable Pipelined Graphics Processing with Branching Instructions.

NOVÁK, J., HAVRAN, V., AND DACHSBACHER, C. 2010. Path Regeneration for Interactive Path Tracing. In *Proceedings of EUROGRAPHICS 2010, short papers*, 61–64.

NVIDIA, 2011. NVIDIA CUDA Programming Guide 4.0.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. 1998. Interactive ray tracing for isosurface rendering. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, VIS '98, 233–238.

STEFFEN, M., AND ZAMBRENO, J. 2010. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 237–248.

TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, M. Doggett, S. Laine, and W. Hunt, Eds., 29–37.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. Graph. 24* (July), 434–444.

ZHANG, E. Z., JIANG, Y., GUO, Z., AND SHEN, X. 2010. Streamlining GPU applications on the fly: Thread Divergence Elimination through Runtime Thread-data Remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ACM, New York, NY, USA, ICS '10, 115–126.