SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms

Steffen Frey, Guido Reina and Thomas Ertl

Visualization Research Center, University of Stuttgart
Quick Overview - The Point of Departure

• GPUs are SIMT architectures consisting of a set of multiprocessors
• Each multiprocessor creates, manages, schedules and executes threads in groups called warps (e.g. 32 threads)
• A warp computes one common instruction at a time (lock step)
• Threads not on the current branch path are disabled

“For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge.” (NVIDIA CUDA/OpenCL Programming Guide)
Quick Overview - Problem Classification

- Thread divergence within warp

Termination Divergence

Branch Divergence

<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Thread 1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Thread 2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Thread 3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp</td>
<td>Σ Iterations: 64  # Iterations Executed: 36  # Iterations Wasted: 28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Branch Condition | 0 | T | F | 1 | T | F | 2 | T | F | 3 | T | F | 4 | T | F | 5 | T | F | 6 | T | F | 7 | T | F |
| Thread 0         | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Thread 1         | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Thread 2         | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Thread 3         | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ...              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Warp             | Σ Code Blocks: 64  # Blocks Executed: 52  # Blocks Masked: 20 |
Quick Overview - The Approach

• We look at GPU kernels with iterative structure
  • Modify process to switch and assign tasks
Quick Overview - Approach Outline

- Termination Divergence (Loops)
  - Fetch new task and assign it when thread finishes
  - Start as many threads as are required to fully occupy the hardware -> problem independent

- Branch Divergence: Task Context Switching (if-else)
  - Attach/detach tasks such that optimally all threads need to execute a branch
Basic Terminology

• Task: Unique identification of a computation job
  • Simply an identification number
  • Necessary information to initiate a computation
• Task Context: State of in-process task computation
  • Program position
  • Per-thread variables
  • Other information necessary to store and resume jobs
• NVIDIA CUDA terminology in general
Branch Divergence

- Execute only one branch of an if-else construct
  - The branch which allows higher occupancy
  - Switch task contexts accordingly
- Remaining this section:
  - How to handle task contexts
  - How to determine which branch.
  - When to switch task contexts.
Branch Divergence - Handling of Task Contexts

- Attached Task context are in register space of thread
- Detached Task contexts are stored in an array in shared memory (so-called task context pool)
- When detaching task context, values are copied from registers to the task pool (vice versa for attaching)
- Branch map references task context pool
  - Stores upcoming branch path task context
  - Gives counts how many are stored for branch

<table>
<thead>
<tr>
<th>Condition 0, true</th>
<th>Task Context Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition 0, false</td>
<td></td>
</tr>
<tr>
<td>Condition 1, true</td>
<td></td>
</tr>
<tr>
<td>Condition 1, false</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Branch Divergence - Determine which branch

- All task contexts active at current condition vote on the branch they need
- Voting between threads for attached task contexts
- Add information from branch map for detached task contexts
Branch Divergence - Switch Task Contexts

- Threads “losing” the vote get a new task context if available
  - Consult branch map for a list of respective task contexts
  - Consider how many other losing threads there are and what IDs they have to avoid attaching the same context twice
- Depending on the cost of iteration step, it can make sense to only allow task switches every n-th iteration
Termination Divergence

• Fetch new tasks for finished threads
  1. Task-independent initialization (only done once)
  2. Fetch a task and execute task-dependent setup
  3. Run iteration steps until task is finished
  4. Then write the results and goto 2.
  5. Exit when there are no more tasks to fetch and all running tasks are complete
Task Fetching From Task Pools

- Tasks are fetched from so-called task pool
  - Contain a range of tasks
  - Represented by a counter and max capacity value
  - Threads fetch work by atomicAdd
    - Returns number that was incremented
    - Use this number as task id
- Implemented and evaluated three different task fetching schemes
  - Differ in hierarchy of task pools
Task Fetch Strategies - Local

- Threads of a block share the local task pool
- Fetch tasks directly from them
- Local task pool in shared memory
  - Task pool initialized at beginning of application
- Many local task pools (one per block), no exchange between them
Task Fetch Strategies - Global

• One Global task pool for all threads
  • resides in global memory
• Private task pool for each thread
  • Resides in register space
• Before fetching from global check whether a task in local pool is available
• Involves frequent fetches from global memory
Task Fetch Strategies - Hybrid

- Cache global task pool accesses in local task pool
  - When local task pool is empty, refill from the global task pool
  - One thread of a warp determined to do that
Task Fetch Strategy Impact

- Color shows task distribution on 2D array for different strategies
- Hybrid and Local structured, Global scattered
- Can have impact for data sharing, memory accesses etc.

Local, Colored Blocks

Global, Colored Warps

Hybrid, Colored Warps
Task Fetch Control

- When to exit for writing results and task fetch
- Options
  - As soon as one thread is out of work
  - When all threads are out of work
  - Anything in between
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 pre-defined outcome are temporarily invalidated. The default vote and the $n$ need to be adapted by the programmer to the problem at hand.

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 access.
Results

• Three Example Applications
  • Monte Carlo
  • Fractals
  • Isosurface Raycasting
• GTX 580, CUDA 4.0

divergent warps:
43752/131072
(33%)
Results - Monte Carlo - Branch Divergence

- Non-accelerated constant for high probability of thread entering the branch
- Accelerated grows linearly with actual workload
- Overhead of max 20%

```java
for(int i=0; i<1000; i++)
    if(norm_rand() < Branch Probability)
        DO SOMETHING EXPENSIVE
```
Results - Monte Carlo - Termination Divergence

- Speedups of 4 for 1% kill probability already
- Hybrid already achieves speedup for 10% kill ratio
- Fetch when all are done is approximately the vanilla time

while(norm_rand() < kill probability)
DO SOMETHING EXPENSIVE

non-accelerated := 100
Results - Fractals - Termination Divergence

- Offset parameter steers divergence

```
cnt=0
while (cnt < crunch_factor && not convergence)
    convergence = EVAL_FORMULA(offset)
    cnt++
```
Results - Fractals - Termination Divergence

• The higher the divergence, the higher the speedup (up to factor 4)
  • No speedup when there is no divergence
  • Speedup also increases with crunch factor
• Send ray from eye through volume

\[
\text{Ray} \quad \text{tmin} \quad \text{tmax} \\
\text{Camera} \quad \text{Image} \quad \text{Volume}
\]

\[y = f(x,y,z)\]

\[\text{isosurfaces } y = v_0, v_1\]

\[
\text{for}(t = t_{\text{min}}; \ t < t_{\text{max}}; \ t += \text{stepSize}) \\
\text{if}(\text{surpassed an isosurface}) \quad \text{localize intersection & lighting}
\]
Results - Isosurface Raycasting

- Up to 7 times faster for our testing parameter range
  - Depends on step size -> amount of isosurface hits in different step counts

![Graph showing Frames Per Second vs Step Size with different acceleration methods: Vanilla, Accelerated, Accelerated w/ Deferred.](image-url)
Conclusion and Future Work

- SIMT architectures potentially waste many clock cycles
- Proposed techniques to specifically overcome termination and branch divergence

Future Work
- Task generation in kernel
- Assign priorities to tasks
- More detailed study of memory access impact
Thank You!