Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection

Farzad Khorasani
farkhor@gatech.edu

The paper presented at
The 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015

Thread Divergence: Problem Overview

• One PC for the SIMD group (warp):
  • One instruction fetch and decode for the whole warp.
  • Reduces the die size and the power consumption.
  • Warp lanes must run in lockstep.
• When facing intra-warp divergence:
  • Mask-off inactive threads.
  • Hold the re-convergence PC in a stack.
  • Some execution units are reserved but not utilized until re-convergence.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BFS</th>
<th>DQG</th>
<th>EMIES</th>
<th>FF</th>
<th>HASH</th>
<th>IEFA</th>
<th>RT</th>
<th>SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp Exec. Eff. (%)</td>
<td>58</td>
<td>37</td>
<td>44</td>
<td>13</td>
<td>25</td>
<td>41</td>
<td>67</td>
<td>64</td>
</tr>
</tbody>
</table>
Divergence Handling

![Control flow graph](image)

Figure courtesy of Narasiman et al., MICRO 2011.

Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection | Farzad Khorasani

### Thread Divergence in Repetitive Tasks: Example

```c
__global__ void CUDA_kernel_BFS(
    const int numV, const int curr, int* levels,
    const int* v, const int* e, bool* done ) {
    for(
        int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
        vIdx < numV;
        vIdx += blockDim.x * gridDim.x ) {
        bool p = levels[ vIdx ] == curr; // Block A.
        if( p )
            process_nbrs( vIdx,
                          curr, levels, v, e, done ); // Block B.
    }
}
```

Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection | Farzad Khorasani
Thread Divergence: Visualization

Collaborative Context Collection:
Main Idea

- Keep collecting divergent tasks until there are enough tasks to keep all warp lanes busy.
- If the aggregation of collected divergent tasks and new divergent tasks equals to or exceeds the warp size, execute.
Collaborative Context Collection: Visualization

Collaborative Context Collection: Principles

- **Execution discipline:** *all-or-none.*
- **Context:** minimum set of variables (thread’s registers) describing the divergent task.
  - Registers defined prior to divergent task path as a function of thread-specific special registers.
  - Used inside the divergent task path.
- **Context stack:** a warp specific shared memory region to collect insufficient divergent task contexts.
- **Required assumption:** repetitive divergent tasks with independent iterations.
Collaborative Context Collection: Applying to CUDA Kernels (1/2)

```c
__global__ void CUDA_kernel_BFS_CCC(
    const int numV, const int curr, int* levels,
    const int* v, const int* e, bool* done ) {
    volatile __shared__ int cxtStack[ CTA_WIDTH ];
    int stackTop = 0;
    int wOffset = threadIdx.x & ( ~31 );
    int lanemask_le = getLaneMaskLE_PTXWrapper();
    for( int vIdx = threadIdx.x + blockIdx.x * blockDim.x; vIdx < numV;
        vIdx += blockDim.x * gridDim.x ) {
        bool p = levels[ vIdx ] == curr; // Block A.
        int jIdx = vIdx;
        int pthBlt = __ballot( !p );
        reducedNTaken = __popc( pthBlt );
        if( stackTop >= reducedNTaken ) { // All take path.
            int wScan = __popc( pthBlt & lanemask_le );
            int pos = wOffset + stackTop - wScan;
            if( !p ) jIdx = cxtStack[ pos ]; // Pop.
            stackTop -= reducedNTaken;
            process_nbrs( jIdx, curr, levels, v, e, done ); // Block B.
        } else { // None take path.
            int wScan = __popc( ~pthBlt & lanemask_le );
            int pos = wOffset + stackTop + wScan - 1;
            if( p ) cxtStack[ pos ] = jIdx; // Push.
            stackTop += warpSize - reducedNTaken;
        }
    }
}
```

Collaborative Context Collection: Applying to CUDA Kernels (2/2)

```c
... if( stackTop >= redNTaken ) { // All take path.
    int wScan = __popc( pthBlt & lanemask_le );
    int pos = wOffset + stackTop - wScan;
    if( !p ) jIdx = cxtStack[ pos ]; // Pop.
    stackTop -= reducedNTaken;
    process_nbrs( jIdx, curr, levels, v, e, done ); // Block B.
} else { // None take path.
    int wScan = __popc( ~pthBlt & lanemask_le );
    int pos = wOffset + stackTop + wScan - 1;
    if( p ) cxtStack[ pos ] = jIdx; // Push.
    stackTop += warpSize - reducedNTaken;
} }
```
Collaborative Context Collection: Transformations

- **Grid-Stride Loops**
  - Enable task repetition over a divergent GPU kernel.

- **Loops with variable Trip-Count**
  - Reduce the largest trip-count with an intra-warp butterfly shuffle reduction.
  - Select the resulting value as the uniform trip-count.
  - Wrap the code inside the loop by a condition check.

- **Recursive Device Functions & Loops with Unknown Trip-Count**
- **Nested and Multi-path Context Collection**
  - A separate and independent context stack for each divergent path.

---

Grid-Stride Loops

**Before**

```c
__global__ void CUDA_kernel_BFS(int numV, int* curr, int* levels, int* v, int* e, bool* done) {
    int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
    if(vIdx < numV) {
        bool p = levels[vIdx] == curr;
        if(p) { // process_nbrs(vIdx, curr, levels, v, e, done); }
    }
}
```

**After**

```c
__global__ void CUDA_kernel_BFS_with_gridstride_loop(int numV, int* curr, int* levels, int* v, int* e, bool* done) {
    int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
    if(vIdx < numV) {
        bool p = levels[vIdx] == curr;
        if(p) { // process_nbrs(vIdx, curr, levels, v, e, done); }
    }
}
```

- Launch enough thread-blocks to keep SMs busy.
- Let threads iterate over the tasks.
- Allowing persistency of threads (Aila et al., HPG, 2009).
Collaborative Context Collection: Optimizations

- **Context Compression**
  - If a context’s register can be computed from another context register simply, stack only one.
  - Calculate (rematerialize) the other one during the retrieval.

- **Memory Divergence Avoidance**
  - Take the coalesced memory access out of the divergent path to keep it aligned.

- **Prioritizing the Costliest Branches**
  - To avoid restricting occupancy, apply CCC only to the most expensive branches: longest branches with the least probability of traversal.

**Collaborative Context Collection: Automation**
Collaborative Context Collection: Performance Improvement

Collaborative Context Collection: Sensitivity on # of Diverging Warp Lanes
Collaborative Context Collection: Sensitivity on Divergent Path Length

Summary

- Collaborative Context Collection [and Consumption] (CCC) as a software/compiler technique to overcome thread divergence.
- CCC collects the context of divergent threads at a stack inside the shared memory in order to implement all-or-none principle.
- Transformations enable applying CCC to wider class of applications.
- Optimizations improve the performance in certain situations.
- CCC can be automated as a compiler technique.
- CCC results in warp execution efficiency increase and the speedup in applications with certain repetitive patterns especially for compute-intensive ones.
The Case for **Volta**: *Independent Thread Scheduling*

Pre-Volta
Program Counter (PC) and Stack (S)

32 thread warp

Volta
Convergence Optimizer

32 thread warp with independent scheduling


Warp Execution Model: Pascal vs Volta

Pascal Warp Execution Model

If (threadIdx.x < 4) {
    A;
    __syncewarp();
    B;
} else {
    X;
    __syncewarp();
    Y;
}

No Synchronization Permitted

Volta Warp Execution Model

Synchronization may lead to interleaved scheduling!

If (threadIdx.x < 4) {
    A;
    __syncewarp();
    B;
} else {
    X;
    __syncewarp();
    Y;
    __syncewarp();
}

Individualized PC Tracking

<table>
<thead>
<tr>
<th>GPU</th>
<th>Kepler GK100</th>
<th>Maxwell GM200</th>
<th>Pascal GP100</th>
<th>Volta GV100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / SM</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / SM</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / SM</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max 32-bit Registers / SM</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Block</td>
<td>65536</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255^1</td>
</tr>
<tr>
<td>Max Thread Block Size</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>FP32 Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Ratio of SM Registers to FP32 Cores</td>
<td>341</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16 KB/32 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>Configurable up to 96 KB</td>
</tr>
</tbody>
</table>

^1 The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.


A Simple Example: SAXPY

```c
__global__ void saxpy(float* out, float a, float* x, float* y) {
  out[threadIdx.x] += a * x[threadIdx.x] + y[threadIdx.x];
}
```

Register usage reported: 8

```c
__global__ void saxpy2(float* out, float a, float* x, float* y) {
  out[threadIdx.x] += a * x[threadIdx.x] + y[threadIdx.x];
}
```

Register usage reported: 10
The Case for *Volta*: *Independent Thread Scheduling*

[Diagram showing pre-Volta and Volta thread scheduling]


---

An Instance of Indirect Communication?

[Diagram showing hardware managed memory update]

Volta SM (left) vs Pascal SM (right)

Maybe Dynamic Warp Formation? (W. Fung et al., MICRO 2007)

Picture from Fung's slides.