The Dual-Path Execution Model for Efficient GPU Control Flow

Minsoo Rhu  Mattan Erez

HPCA 2013
Outline

• Background
  – Stack-based reconvergence
  – Dynamic warp subdivision
• Dual-path execution model
• Evaluation
• Conclusion
Stack-Based Reconvergence

• When the control flow of different threads within a single warp diverges, execution of concurrent control paths is serialized with every divergence.

• Threads reconverge at the immediate post-dominator instruction of that branch
Stack-Based Reconvergence

• The way to implement reconvergence: treat control flow execution as a serial stack.
  Each time control diverges, both the taken and not taken paths are pushed onto a stack (in arbitrary order) and the path at the new top of stack is executed.
• When the control path reaches its reconvergence point, the entry is popped off of the stack and execution now follows the alternate direction of the diverging branch.

<table>
<thead>
<tr>
<th>PC</th>
<th>Active mask</th>
<th>Reconvergence PC(RPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reconvergence stack and its operation

When a warp first starts executing, the stack is initialized with a single entry.
Reconvergence stack and its operation

Conditional branch: PC-> PC value of the reconvergence point
PC of the right path, its corresponding active mask and RPC
PC of the left path, its corresponding active mask and RPC

Single-path execution (SPE)
Reconvergence stack and its operation

When the current PC of a warp matches the RPC field value at that warp’s TOS, the entry at the TOS is popped off.
Reconvergence stack and its operation

Divergent branch at the end of block C
Reconvergence stack and its operation

control flow is reconverged back to the path that started at block C
Reconvergence stack and its operation

execution continues along a single path with a full active mask
Reconvergence stack and its operation

Deficiencies:
SIMD utilization decreases every time control flow diverges;
Execution is serialized.

Bubbles: idle execution resources (masked lanes or zero ready warps available for scheduling in the SM)

(g) Execution flow using baseline stack architecture.
Dynamic Warp Subdivision

- allow warps to interleave the scheduling of instructions from concurrently executable paths (left and right paths)
- a divergent branch may either utilize the baseline single-path stack, or instead, ignore the stack and utilize an additional hardware structure, the warp-split table (WST), which is used to track the independently-schedulable warp-splits

<table>
<thead>
<tr>
<th>PC</th>
<th>Active mask</th>
<th>Reconvergence PC(RPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Active mask</td>
<td>Reconvergence PC(RPC)</td>
</tr>
</tbody>
</table>

- warp-split: independent scheduling entities and are treated equally as warps by the scheduler (the left and right paths of a divergent)
When BR(B-C) is executed, the number of instructions in block G (which is 3) is larger than the subdivision threshold determined by heuristics, the warp is not subdivided.
DWS operation

BR(D-E) has a post-dominator (block F) smaller than the threshold -> subdivide

WST's RPC field is updated to path G which equals the RPC field value at the TOS

---

**Single-path stack**

<table>
<thead>
<tr>
<th>PC</th>
<th>Active Mask</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>1111</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>0111</td>
<td>G</td>
</tr>
</tbody>
</table>

**Warp-split table**

<table>
<thead>
<tr>
<th>PC</th>
<th>Active Mask</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0100</td>
<td>G</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>G</td>
</tr>
</tbody>
</table>
DWS operation

Warp-splits registered in WST continue execution until their PC matches its RPC field.

Once warp-splits arrive path G, the two entries in WST are invalidated, and the reconvergence stack is used to execute path G.
DWS operation

Compared with baseline architecture: increases parallelism and potential latency hiding

Deficiency: reduced SIMD utilization (the stack could have reconverged nested branches whereas the WST cannot)
Motivation

• Single Path Execution maximize SIMD utilization with structured control flow, but always serializes execution with only a single path schedulable at any given time.

• Dynamic Warp Subdivision can interleave the scheduling of multiple paths and increase TLP, but this sacrifices SIMD lane utilization.

• Goal: matches the utilization and SIMD efficiency of the baseline SPE while still enhancing TLP in some cases
Dual-Path execution model

• Dual-Path stack structure

  – Idea: instead of pushing the taken and fall-through paths onto the stack one after the other, in effect serializing their execution, the two paths are maintained in parallel.

  – Stack entry:
  – a) PC and active mask value of the left path (Path L)
  – b) PC and active mask value of the right path (Path R)
  – c) the RPC (reconvergence PC) of the two paths.
Dual-Path execution

Initial status
Dual-Path execution

When BR(B-C) is executed, a single entry is pushed onto the stack. Because it contains the information for both paths, the single TOS entry enables the warp scheduler to interleave the scheduling of active threads at both paths.
Dual-Path execution

If both paths are active at the time of divergence, the one to diverge (block C) first pushes an entry onto the stack, and in effect, suspends the other path (block B) until control returns to this stack Entry.
Dual-Path execution

When either one of the basic blocks at the TOS arrives at the reconvergence point and its PC matches the RPC, the block is invalidated (block D)
Dual-Path execution

Once both paths arrive at the RPC, the stack is popped and control is returned to the next stack entry.
Dual-Path execution
Dual-Path execution

Compared with baseline architecture

(g) Execution flow using the dual-path stack model.
Data dependency I

- Unresolved pending writes before divergence (e.g., r0 on path A) should be visible to the other path (e.g., r0 on path C) after divergence, and further, both paths need to know when r0 is written back.

```
// Path A
load r0, MEM[~];

if()
{
    // Path B
    load r1, MEM[~];

    if()
    {
        // Path D
        add r4, r1, r3;
    }

    else
    {
        // Path E
        sub r4, r1, r3;
    }

    else
    {
        // Path C
        add r5, r0, r2;
        ...
    }

    Divergence

    Reconvergence

    // Path F
    ...

    load r7, MEM[~];
}

Reconvergence

// Path G
add r8, r1, r7;
```
Data dependency II

- Unresolved pending writes before reconvergence (e.g., r7 on path F) should be visible to the other path (r7 at path G) after reconvergence.

```c
// Path A
load r0, MEM[~];

if () { // Path B
    load r1, MEM[~];
} else { // Path C
    add r5, r0, r2;
    ...
}

if () { // Path D
    add r4, r1, r3;
} else { // Path E
    sub r4, r1, r3;
}

// Path F
...
load r7, MEM[~];

// Path G
add r8, r1, r7;
```
Data dependency III

- If a register number is the destination register of an instruction past a divergence point, then this register should not be confused with the same register number on the other path. (e.g., r1 on path B is a different register than r1 on paths D/E)

<table>
<thead>
<tr>
<th>Left Path</th>
<th>Right Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Path A</td>
<td></td>
</tr>
<tr>
<td>load r0, MEM[~];</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>if() { // Path B</td>
<td>else { // Path C</td>
</tr>
<tr>
<td>load r1, MEM[~];</td>
<td>add r5, r0, r2;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>if() { // Path D</td>
<td>else { // Path E</td>
</tr>
<tr>
<td>add r4, r1, r3;</td>
<td>sub r4, r1, r3;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>// Path G</td>
<td></td>
</tr>
<tr>
<td>add r8, r1, r7;</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing control flow and register usage](diagram.png)
Data dependency IV

- if the register number on two different paths is a destination in both paths concurrently, then writes to this register number from the two paths should not be confused with one another. (e.g., r4 on path D is a different register than r4 on paths E)

```c
// Path A
load r0, MEM[~];

// Path B
if ( ) { // Path C
    load r1, MEM[~];
    add r5, r0, r2;
    ...
}
else { // Path D
    add r4, r1, r3;
}

// Path E
else { // Path F
    sub r4, r1, r3;
    ...
} // Path G
load r7, MEM[~];

// Path G
add r8, r1, r7;
```
Scoreboard

• Per-warp scoreboard to track data dependencies.
• Content-addressable-memory (CAM) structure: indexed with a register number and a warp ID which returns whether that register is pending write-back for that warp
• Once an instruction is scheduled for execution, the scoreboard is updated to show the instruction’s destination register as pending.

(Warp ID, Reg)
(W₀, 0)

Scoreboard

- P: Pending writes
- S: Shadow bit
Modified Scoreboard

• Modification: extend the scoreboard to track the left and right path separately; add a shadow bit to each scoreboard entry.

when a path diverges, the pending bits of its scoreboard are copied to its shadow bits.

When querying the scoreboard for a register in one path, the shadow bits in the other path are also examined. If either the path’s scoreboard or the shadow in the other scoreboard indicate a pending write, the path stalls.
Scoreboard example

Path A's load to r0 results in allocating a pending entry at Scoreboard-L. After BR(B-C) is executed, P-bits are all copied into the S-bits.
Scoreboard example

// Path A
load r0, MEM[~];

--- Divergence ---

if ( ){ // Path B
    load r1, MEM[~];
}
else{ // Path C
    add r5, r0, r2;
    ...

--- Divergence ---

if( ){ // Path D
    add r4, r1, r3;
} else{ // Path E
    sub r4, r1, r3;
}

--- Reconvergence ---

// Path F
...
load r7, MEM[~];

--- Reconvergence ---

// Path G
add r8, r1, r7;

(Warp ID, Path, Reg)
(W_0, R, 0)

Scoreboard_L
Reg | P | S
--- | --- | ---
0   | 1  | 1
-   | 0  | 0
-   | 0  | 0
-   | 0  | 0

Scoreboard_R
Reg | P | S
--- | --- | ---
-   | 0  | 0
-   | 0  | 0
-   | 0  | 0
-   | 0  | 0

Path C, which is executed in the right path, detects pending write from predivergence
Scoreboard example

Path A’s load to r0 is resolved and clears its entry. Path B’s load to r1 has its S-bit set when BR(D-E) is executed, and path D sees a RAW hazard when executing add.
Scoreboard example

Path F's load to r7 has its S-bit set when path B and F have both reconverged. Path G, which will execute add in the left path, can therefore see the RAW hazard for r7.
Evaluation

• Benchmark

<table>
<thead>
<tr>
<th>Interleavable</th>
<th>#Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>LUD</td>
<td>LU Decomposition</td>
</tr>
<tr>
<td>QSort</td>
<td>Quick Sort</td>
</tr>
<tr>
<td>Stencil</td>
<td>3D Stencil Operation</td>
</tr>
<tr>
<td>RAY</td>
<td>Ray Tracing</td>
</tr>
<tr>
<td>LPS</td>
<td>Laplace Solver</td>
</tr>
<tr>
<td>MUMpp</td>
<td>MUMmerGPU++</td>
</tr>
<tr>
<td>MCML</td>
<td>Monte Carlo for ML Media</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-interleavable</th>
<th>#Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>DXTC</td>
<td>DXT Compression</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td>PathFind</td>
<td>Path Finder</td>
</tr>
<tr>
<td>NW</td>
<td>Needleman-Wunsch</td>
</tr>
<tr>
<td>HOTSPOT</td>
<td>Hot-Spot</td>
</tr>
<tr>
<td>BFS2</td>
<td>Breadth-First Search 2</td>
</tr>
<tr>
<td>BACKP</td>
<td>Back Propagation</td>
</tr>
</tbody>
</table>

\[
\text{Avg_{Path}} = \frac{\sum_{i=1}^{N} \text{NumPath}_i}{N}
\]

N: the total number of warp instructions issued throughout the execution of the kernel

\text{NumPath}_i: the total number of concurrently schedulable paths available within the issued warp when the \(i\)-th warp instruction is issued.

• Not all divergent branches are interleavable because many branches have only an if clause with no else.

• Benefit from DPE: \(\text{Avg}_{\text{Path}} > 1\)
Evaluation

Except for MUMpp, whose IPC is degraded by 1:1%, due to its increased L1 miss rate, DPE provides an improvement in performance across all the interleavable workloads (14:9% on average) while never degrading the performance of non-interleavable ones.
Conclusion

• Dual-path execution either matches the performance of the baseline single-path stack architecture or outperforms single-path execution by 14.9% on average and by over 30% in some cases.