Simultaneous Branch and Warp Interweaving for Sustained GPU Performance

Nicolas Brunie  
Kalray and ENS Lyon  
nicolas.brunie@kalray.eu

Sylvain Collange  
Univ. Federal de Minas Gerais  
sylvain.collange@dcc.ufmg.br

Gregory Diamos  
NVIDIA Research  
gdiamos@nvidia.com

ISCA'39  
Portland, OR  
June 11, 2012
Mitigating GPU branch divergence cost

GPUs rely on SIMD execution

- Serialization of divergent branches $\rightarrow$ resource underutilization

Contribution: a second instruction scheduler to improve utilization
Outline

- GPU microarchitecture
  - SIMT model
  - The divergence problem
- Simultaneous branch interweaving
- Simultaneous warp interweaving
Context: GPU microarchitecture

Software: graphics shaders, OpenCL, CUDA...

1 program

```
kernnel void scale(float a, float * X) {
    X[tid] = a * X[tid];
}
```

Many threads

Architecture: multi-thread SPMD programming model

GPU microarchitecture

Hardware: GPU

Hardware datapaths: SIMD execution units
Single-Instruction Multi-Threaded (SIMT)

- Implicit SIMD execution model
  - Fetch 1 instruction for a warp of lockstepping threads
  - Execute on SIMD units

- Optimized for regular workloads
The control divergence problem

- Control divergence: conflict for shared fetch unit
- Serialize execution paths

1: if(!tid%2) {
  2:     a+b;
  3:   else {
  4:     a*b;
  5:   }

Efficiency loss
The control divergence problem

- Control divergence: conflict for shared fetch unit
- Serialize execution paths

```c
1: if(!tid%2) {
2:     a+b;
3: } else {
4:     a*b;
5: }
```

- Efficiency loss
Outline

- The GPU divergence problem
- Simultaneous branch interweaving
  - Double instruction fetch
  - Finding branch-level parallelism
  - Restoring lockstep execution
  - Implementation
- Simultaneous warp interweaving
Simultaneous Branch Interweaving

- Add a second fetch unit
  - Simultaneous execution of divergent branches

```plaintext
1: if(!tid%2) {
2:     a+b;
3: else {
4:         a*b;
5: }
```
Standard divergence control: mask stack

Code

```c
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
} pop

else {
    push
    x = 3;
    push
} pop
```

Mask Stack
1 activity bit / thread

tid=0
tid=1
tid=2
tid=3

- Problem: does not expose branch-level parallelism
Alternative to stack: 1 PC / thread

Code

```cpp
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
} else {
}
```

Program Counters (PCs)

<table>
<thead>
<tr>
<th>tid</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>→ active</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No match</td>
<td>→ inactive</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Policy: MPC = min(PC_i)

- Earliest reconvergence with code laid out in *Thread Frontiers* order
Run two branches simultaneously

**Code**

```java
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    }
    else {
        x = 3;
    }
}
```

**Program Counters (PCs)**

<table>
<thead>
<tr>
<th>tid</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \(\text{MPC}_1 = \min(\text{PC}_i)\)
- \(\text{MPC}_2 = \min(\text{PC}_i, \text{PC}_i \neq \text{MPC}_1)\)
Restoring lockstep execution

- Issue: unbalanced paths break lockstep execution
  - Power consumption, loss of memory locality
- Solution: implicit partial synchronization barrier

Control-flow graph

Greedy scheduling

Instruction 6, 7 broken down, issued twice

Earliest reconvergence

Synchronize before instruction 6
Enforcing control-flow reconvergence

- Wait for any thread of the warp between $PC_{\text{div}}$ and $PC_{\text{rec}}$
- Annotate reconvergence points with pointer to immediate dominator
Enforcing control-flow reconvergence

- Wait for any thread of the warp between $PC_{\text{div}}$ and $PC_{\text{rec}}$
- Annotate reconvergence points with pointer to immediate dominator

$T_0$ and $T_2$ (at $F$) wait for $T_1$ (in $D$).

$T_3$ (in $B$) can proceed in parallel.
Implementation: context table

- Common case: few different PCs
- Order stable in time
- Keep Common PCs+activity masks in sorted heap
Two-level context table

- Cache top 2 entries in the *Hot Context Table* register
  - Constant-time access to $\text{MPC}_i = \text{CPC}_i$, activity masks
- Other entries in the *Cold Context Table* linked list
  - Branch → incremental insertion in CCT

![Diagram of Two-level Context Table]

- **HCT**
  - $\text{CPC}_1$, $m_1$, $v_1$, $\text{CPC}_2$, $m_2$, $v_2$, $p$
  - HCT sorter

- **CCT**
  - $\text{CPC}_i$, $m_i$, next
  - Sideband
  - CCT sorter

Insert, Pop
Outline

- The GPU divergence problem
- Simultaneous branch interweaving
- Simultaneous warp interweaving
  - Idea
  - Dealing with lane conflicts
  - Implementation
  - Results
Simultaneous Warp Interweaving

- SBI limitation: often no secondary path
  - Single-sided if statements, loops...
- SWI: opportunistically schedule instructions from other warps in divergence gaps
  - “SMT for SIMD”

![Diagram showing simultaneous warp interweaving with instructions and warps labeled T0 to T7, PC=17 and PC=42, fetch and execute stages.]
Using divergence correlations

- **Issue:** unbalanced divergence introduces conflicts
  - e.g. Parallel reduction

- **Solution:** static lane shuffling
  - Apply different lane permutation for each warp
  - Preserves inter-thread memory locality
Detecting a compatible secondary warp

- **Bitset inclusion test:**
  - Content-Associative Memory
  - Treat zeros as don't care bits
  - Power-hungry!

- **Set-associative lookup**
  - Split warps in sets
  - Restrict lookup to 1 set
  - More power-efficient
Set-associative lookup is good enough

- **3-way**: captures 66% of performance potential
- **Direct-mapped**: 48%
Experimental configuration

Baseline: clustered SIMT architecture (Fermi, Kepler)
- Tie both clusters together to form twice bigger warps
- Direct both instructions to the same execution units

Baseline: warp size 32
2 warps / clock, 1 instruction / warp

SBI/SWI: warp size 64
1 warp / clock, 2 instructions / warp

Fetch-unit / execute-unit ratio maintained
Performance results

<table>
<thead>
<tr>
<th></th>
<th>Regular applications</th>
<th>Irregular applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBI</td>
<td>+15%</td>
<td>+41%</td>
</tr>
<tr>
<td>SWI</td>
<td>+25%</td>
<td>+33%</td>
</tr>
<tr>
<td>SBI+SWI</td>
<td>+23%</td>
<td>+40%</td>
</tr>
</tbody>
</table>
Perspective: SMT-GPU µarch convergence

- Converging trends in **SMT** and **GPU** architecture
  - Closing micro-architectural space between Clustered Multi-Threading and SIMD
  - Explore new tradeoffs between power efficiency and flexibility?

**Efficiency on regular MT apps**
Merge instructions from concurrent threads

**Flexibility**
Loosen constraints of SIMD execution
Simultaneous Branch and Warp Interweaving for Sustained GPU Performance

Nicolas Brunie  
Kalray and ENS Lyon  
nicolas.brunie@kalray.eu

Sylvain Collange  
Univ. Federal de Minas Gerais  
sylvain.collange@dcc.ufmg.br

Gregory Diamos  
NVIDIA Research  
gdiamos@nvidia.com

ISCA'39  
Portland, OR  
June 11, 2012
Backups
References

SWI implies SMT

- Heterogeneous execution units
- SWI improves utilization with superscalar execution
SBI vs. DWF

- **Dual fetch**
- Uses branch-level parallelism
  - Sensitive to branch unbalance
- Preserves in-warp locality

- **Single fetch**
- Uses warp-level parallelism
  - Sensitive to lane activity unbalance
- Dual fetch
- Uses warp-level parallelism
  - Sensitive to lane conflicts
- Preserves in-warp locality

- Single fetch
- Uses warp-level parallelism
  - Sensitive to lane activity
  - Unbalance, low occupancy
Simulation platform

- **Barra**: functional GPU simulator modeled after NVIDIA Tesla GPUs
  - Runs native Tesla SASS binaries
  - Reproduces SIMT execution
- **Timing-power model**
  - Cycle-accurate execution pipeline
  - Constant-latency, bandwidth-bound memory
  - Calibration from GPU microbenchmarks

SBI scoreboard logic

- Keep track of dependencies induced by thread divergence-reconvergence
- Transitive closure of dependency graph

```
\begin{align*}
D(t - 3, t - 2) &= \begin{bmatrix} 100 \\ 100 \\ 011 \end{bmatrix} \\
D(t - 2, t - 1) &= \begin{bmatrix} 100 \\ 010 \\ 001 \end{bmatrix} \\
D(t - 1, t) &= \begin{bmatrix} 110 \\ 001 \\ 001 \end{bmatrix}
\end{align*}
```
Goto considered harmful?

Control instructions in some CPU and GPU instruction sets

- **Control flow structure** is explicit
  - GPU-specific instruction sets
  - No support for arbitrary control flow
## Flynn's taxonomy revisited

<table>
<thead>
<tr>
<th>Resource type:</th>
<th>Instruction</th>
<th>Computation / registers</th>
<th>Memory port (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource count</td>
<td>Fetch</td>
<td>(Data)</td>
<td></td>
</tr>
</tbody>
</table>

### 1

- **SIMT**
- **SDMT**
- **SAMT**

### 2

- **DIMT**
- **DDMT**
- **DAMT**

### M

- **MIMT**
- **MDMT**
- **MAMT**

---

Examples: conventional design points

- Multi-core
  - MIMD(MAMT)

- Short-vector SIMD
  - SIMD(SAST)

- GPU
  - SI(MDSA)MT