Reducing Branch Divergence in GPU Programs

Tianyi David Han
Tarek S. Abdelrahman

March 5, 2011
Agenda

• Background — branch divergence

• Two optimizations
  – Iteration delaying
  – Branch distribution

• Experimental results
  – Two synthetic benchmarks
  – A real-world medical application: MCML

• Conclusions & Future work
Introduction

• Most GPU optimizations target its complex and explicit memory hierarchy
  – Utilizing on-chip cache
  – Memory coalescing, avoiding bank conflicts
  – ...

• Less attention paid to GPU’s SIMD execution model and the impact of divergent execution
  – Real applications deal with corner cases
Branch Divergence

- A GPU has tens of SIMD execution engines, or stream multi-processors

```plaintext
if ((x = A[tid]) > 0.0F)
    B[tid] = logf(x);
else
    C[tid] = x/2.0F;
```

Only ~50% ALU utilization!
Iteration Delaying

• Targets a branch within a kernel loop

```c
for (i = 0; i < N; ++i)
{
    if ((x = A[tid]) > 0.0F)
        B[tid] = logf(x);
    else
        C[tid] = x/2.0F;
}
```
Iteration Delaying

- In each iteration, we only execute threads that take one direction.
- Those that should take the other direction are delayed.

~33% reduction in dynamic instruction count!
Iteration Delaying

- Which (convergent) branch direction should the threads take in each loop iteration?

<table>
<thead>
<tr>
<th></th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_1</td>
<td>[Diagram]</td>
<td>[Diagram]</td>
<td>[Diagram]</td>
</tr>
<tr>
<td>I_2</td>
<td>[Diagram]</td>
<td>[Diagram]</td>
<td>[Diagram]</td>
</tr>
<tr>
<td>I_3</td>
<td>[Diagram]</td>
<td>[Diagram]</td>
<td>[Diagram]</td>
</tr>
</tbody>
</table>

- ~17% reduction in dynamic instruction count!
Iteration Delaying

• We investigate two strategies:
  – Majority-vote
  – Round-robin

• Majority-vote
  – Threads communicate (in each iteration) and select the direction most of them take
  – Parameter: \textit{threshold}

• Round-robin
  – Static scheduling e.g., NNTNNTNNT...
  – Parameter: \textit{cycle}
for (i = 0; i < N; ++i) {
    if ((x = A[tid]) > 0.0F)
        B[tid] = logf(x);
    else
        C[tid] = x/2.0F;
}

int delayed = 0;
int cond;
for (i = 0; i < N; )
{
    if (!delayed)
        cond = ((x = A[tid]) > 0.0F);
    else
        cond = 1 - cond;
    int cond_for_all = __popc(__ballot(cond)) >= threshold;
    delayed = (cond_for_all != cond);
    if (!delayed) {
        if (cond_for_all)
            B[tid] = logf(x);
        else
            C[tid] = x/2.0F;
        ++i;
    }
}
Discussion

• Pros
  – Reduces the number of dynamic instructions for the branch bodies

• Cons
  – Introduces instruction overhead
  – Forgoes non-branch code in delayed threads

• Relevant application characteristics
  – Size of the branch code
  – Size of the non-branch code
  – Branching pattern
Branch Distribution

• Targets an arbitrary multi-path branch
• Factors out common code from the branch paths

```c
if (c > 0) {
    x1 = x2 * y1;
    a = a1; b = b1;
    x = x1; y = y1;
} else {
    x2 = x2 * a2 + b2;
    y2 = y2 * a2 + b2;
}
```

### Prologue
- `x1 = x2 * y1;
- `a = a1; b = b1;
- `x = x1; y = y1;`

### Common Body
- `x2 = x2 * a2 + b2;
- `y2 = y2 * a2 + b2;

### Epilogue
- `if (c > 0) {
    x1 = x; y1 = y;
} else {
    x2 = x; y2 = y;
}`
Discussion

• Pros
  – Potentially reduces the length of divergent code

• Cons
  – Introduces instruction overhead (prologue, epilogue)
  – May increase register usage
  – May reduce the level of Instruction-Level Parallelism
Experimental Evaluation

• Three benchmarks
  – Two synthetic ones with configurable parameters: SYN-ITDELAY, SYN-BRD\text{DIS}
  – One real-world medical application: MCML

• Platform
  – Intel Core 2 Quad 9440 with 4GB main memory
  – Geforce GTX 480
  – CUDA 3.0 on Ubuntu 8.04
Benchmark - SYN-ITDELAY

• Synthetic benchmark for iteration delaying

```
for (i = 0; i < N_ITERATIONS; ++i)
{
    non-branch code
    if (cond)
        then-path code
    else
        else-path code
}
```

• Random branch direction
• Configurable branch freq.
• Configurable size
Benchmark - SYN-BRDIS

- Synthetic benchmark for branch distribution

```c
if (cond) {
    divergent_prologue_1
    code_factored_out
    divergent_epilogue_1
} else {
    divergent_prologue_2
    code_factored_out
    divergent_epilogue_2
}
```

- Fixed, guaranteed to diverge
- Configurable size
MCML

- Models the scattering and absorption of photons in tissues
- Highly optimized for the memory hierarchy

```
Initialize the photon
while (true)
{
    Move the photon in its current direction
    if (it hits a tissue layer boundary) {
        Does the photon transmit or reflect?
        Update the photon direction
    } else {
        Drop a portion of the photon weight to the current tissue location
    }
    ...
}

iteration delaying

Branch distribution

~200 PTX instrs.
~170 PTX instrs.
```
Experiments on SYN-ITDELAY

- Performance benefit as a function of \( r \) (branch ratio)
  
  \[ r = \frac{\text{branch code size}}{\text{non-branch code size}} \]
Experiments on SYN-ITDELAY

- Performance benefit as a function of *branch frequency* $f = \text{the fraction of time the branch is taken}$
- When branch ratio $r = 8$
Experiments on SYN-BRDIS

• Performance benefit as a function of common code ratio 
  \( R = \text{size of code factored out} / \text{size of prologue & epilogue} \)
MCML Performance

• Iteration delaying improves its performance by up to **12%**
  – Achieved on a 7-layer skin model that results in branch ratio $r = 8$
  – Using a round-robin strategy with $cycle = 75$

• Branch distribution improves its performance by up to **16%**
  – Common code ratio $\approx 2$

• Using branch distribution on top of iteration delaying gives **20%** performance improvement
Related Work

• Loop collapsing (S. Lee et al.)
  – Reduces divergence due to varying loop trip-count
  – Targets a very specific pattern of loop nest

• Dynamic warp formation (W. Fung et al.)
  – Reform warps by grouping threads that take the same path
  – Requires hardware support

• Thread-data remapping (Zhang et al.)
  – Requires host-GPU communication

• Dynamic warp splitting (Meng et al.) & Branch splitting (Carrillo et al.)
  – Improve the level of parallelism during divergent execution

• Procedural abstraction (Debray et al.)
  – Used to compress code for embedded system, as opposed to improving performance
Conclusions & Future work

• Two GPU optimizations for reducing branch divergence
  – Iteration delaying & branch distribution

• Preliminary performance evaluation looks promising
  – 1.12X and 1.16X speedup on MCML

• We are encouraged to pursue automation of these optimizations in a compiler
  – Parameter tuning
  – Evaluation on a larger set of benchmarks
Thank you!
Experiments on SYN-ITDELAY

- Performance impact of branch direction scheduling
- When $r = 8$, $f = 10\%$

<table>
<thead>
<tr>
<th>Majority-vote</th>
<th>Speedup</th>
<th>Round-robabn</th>
<th>cycle*</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>threshold</td>
<td>Speedup</td>
<td>cycle*</td>
<td>Speedup</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.89</td>
<td>90%</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.15</td>
<td>70%</td>
<td>1.04</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.11</td>
<td>50%</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0.95</td>
<td>30%</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.73</td>
<td>10%</td>
<td>0.55</td>
<td></td>
</tr>
</tbody>
</table>

* cycle = the fraction of loop iterations the branch is NOT taken
Experiments on SYN-BRDIS

- Register usage as a function of
  
  # of inputs/outputs of the common code

- When $R = 6$

![Graph showing register usage as a function of number of inputs/outputs of the common code before and after optimization.]