Divergence Analysis and Optimizations


yours truly
Our objective is to help compilers (and also programmers) to produce better code for Graphics Processing Units – also called GPUs.

In this work we do it by identifying divergent variables in CUDA programs.

- This information enables automatic code optimizations.
  - We demonstrate it designing branch fusion.
Our Contributions

• **Divergence Analysis**: this is an static analysis that points out uniform variables, i.e., variables that have always the same value for every processing element in a SIMD machine.

• **μ-SIMD**: the semantics of a SIMD machine, at the assembly level, which we have implemented in Prolog.

• **Branch Fusion**: an automatic code optimization that consists in interweaving different paths that originate at a divergent branch.
Single Instruction, Multiple Data

• SIMD: the firing squad analogy.
  – One captain: the instruction fetcher.
  – 8 (or so) soldiers: the arithmetic units.
    • And fake bullets for those who do not do “useful work”.

![Cartoon of soldiers with one giving a thumbs up and another saying, "Cigarette?" The one saying, "Thanks, my life is risky enough as it is." ]
Example of Divergent Program

• Below we have a simple kernel, and its Control Flow Graph (CFG):

```c
__global__ void ex (float* v) {
    if (v[tid] < 0.0) {
        v[tid] /= 2;
    } else {
        v[tid] = 0.0;
    }
}
```

• Why do we have divergences in this kernel?
## Divergences and Performance

<table>
<thead>
<tr>
<th>program counter</th>
<th>label</th>
<th>op</th>
<th>def</th>
<th>use&lt;sub&gt;1&lt;/sub&gt;</th>
<th>use&lt;sub&gt;2&lt;/sub&gt;</th>
<th>ALU&lt;sub&gt;1&lt;/sub&gt;</th>
<th>ALU&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B&lt;sub&gt;0&lt;/sub&gt;</td>
<td>addr</td>
<td>%r1</td>
<td>v</td>
<td>[%tid]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ld</td>
<td>%f1</td>
<td>%r1</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>set.lt</td>
<td>%p1</td>
<td>%f1</td>
<td>0.0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>bra</td>
<td>%p1</td>
<td>$ZR</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>B&lt;sub&gt;ZR&lt;/sub&gt;</td>
<td>div</td>
<td>%f2</td>
<td>%f1</td>
<td>2</td>
<td>●</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>jump</td>
<td></td>
<td>$ST</td>
<td></td>
<td>●</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>B&lt;sub&gt;1&lt;/sub&gt;</td>
<td>mov</td>
<td>%f2</td>
<td>0.0</td>
<td></td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>8</td>
<td>B&lt;sub&gt;ST&lt;/sub&gt;</td>
<td>st</td>
<td>%r1</td>
<td>%f2</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Given a SIMD program, which variables have always the same value for every processing element?

```
__global__ void dec2zero(int* v, int N) {
    int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
    if (xIndex < N) {
        while (v[xIndex] > 0) {
            v[xIndex]--;
        }
    }
}
```
Why not to use profiling?

• Input dependent.
• Relatively hard to use.
• Huge slow-down: 140 times!
• The analysis can be easily embedded into the compiler.
• The analysis is definitive: if it says that a branch is not divergent, then this branch will never diverge.
Divergence Analysis

A local variable is divergent if different threads see it with different values.

• Which variables are divergent?
  – \( v = \text{tid} \)
  – \text{atomic} \{ v = f(\ldots) \}
  – \( v \) is \textbf{data dependent} on a divergent variable \( u \).
  – \( v \) is \textbf{control dependent} on a divergent variable \( u \).
The thread id is always divergent

```c
__global__ void dec2zero(int* v, int N) {
    int xIndex = blockIdx.x*blockDim.x+threadIdx.x;  
    if (xIndex < N) {
        while (v[xIndex] > 0) {  
            v[xIndex]--;  
        }
    }
}
```

Each thread sees a different thread id, so...
Variables defined by atomic ops

• The macro ATOMINC increments a global memory position, and returns the value of the result.

```c
__global__
void ex_atomic (int index, float* v) {
    float f = 0.0;
    f = ATOMINC( v[index] );
}
```
Static Single Assignment Form

- An intermediate representation in which each variable is defined only once.
  - We need to understand phi-functions.

```
%a = mov %ixj
%b = mov %tid
%a1 = mov %ixj
%b1 = mov %tid
%a2 = mov %tid
%t2 = ld %shared[%b]
%t3 = ld %shared[%a]
%p3 = gt %t2 %t3
bra %p3 L7
```

(a) %b = mov %tid
%a = mov %ixj
%t2 = ld %shared[%b]
%t3 = ld %shared[%a]
%p3 = gt %t2 %t3
bra %p3 L7

(b) %b2 = mov %ixj
%a2 = mov %tid
%b = mov %tid
%a = mov %ixj
%t2 = ld %shared[%b]
%t3 = ld %shared[%a]
%p3 = gt %t2 %t3
bra %p3 L7

%a =ϕ (%a1, %a2)
%b =ϕ (%b1, %b2)
What is Data Dependence?

• If the program contains an assignment such as \( v = f(v_1, v_2, \ldots, v_n) \), then \( v \) is data dependent on every variable in the right side, i.e, \( v_1, v_2, \ldots, \) and \( v_n \).
Which data dependences do we have here?
Divergences due to data dependences

Why is SSA so good here?
Are we missing anything?
%i0 = ld v[%tid]  
%j0 = 0

%i = !(%i0, %i1)  
%j = !(%j0, %j3)

%p0 = %i < 100  
branch %p0 B2

%i1 = %i + 1  
%j1 = %j + 1  
%t0 = %j1 mod 2  
%p1 = %t0 = 0  
branch %p1 B4

%j2 = %j1 - 3

%p2 = %j > 100  
branch %p2 B7

%x0 = 1  
jump B8

%x1 = 2

%x = !(%x0, %x1)  
sync  
st v[%tid] %x0  
stop

%j3 = !(%j2, %j1)  
sync  
jump B1

Can variable j diverge?
%i0 = ld v[%tid]
%j0 = 0

%i = \phi(%i0, %i1)
%j = \phi(%j0, %j3)
%p0 = %i < 100
branch %p0 B2

%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B4

%sync %p2 = %j > 100
branch %p2 B7

%j2 = %j1 - 3
%j3 = \phi(%j2, %j1)
%sync
%st v[%tid] %x0
%stop

And variable x? Can it diverge?
Hard question: Can variable p1 diverge?
Sync Dependences

- A variable $v$ is sync dependent on a predicate variable $p$ if $v$ may reach a synchronization point with a different value for different threads, depending on how threads branch on $p$.

- How to find the sync dependences in a program?

```plaintext
B5
sync
%p2 = %j > 100
branch %p2 B7

B6
%x0 = 1
jump B8

B7
%x1 = 2

B8
%x = φ(%x0, %x1)
sync
st %v[%tid] %x0
stop
```
The influence region

- The influence region of a predicate is the set of basic blocks that may (or may not) be reached depending on the value of the predicate, up to the synchronization barrier.
  - The Synchronization barrier is placed at the immediate post-dominator of the branch.
  - Alert! Head is spinning: what is a post-dominator?
What is the influence region of p2?
The influence region of p2:

B₂
- \%i₁ = %i + 1
- \%j₁ = %j + 1
- \%t₀ = %j₁ mod 2
- \%p₁ = %t₀ = 0
- branch %p₁ B₄

B₃
- \%j₂ = %j₁ - 3

B₄
- \%j₃ = %j₂, %j₁
- sync
- st %v[%tid] %x₁
- stop
- jump B₁

B₅
- sync
- %p₂ = %j > 100
- branch %p₂ B₇

B₆
- \%x₀ = 1
- jump B₈

B₇
- \%x₁ = 2

B₈
- sync
- \%x₁ = \%x₀ XOR %x₁
- sync
- st %v[%tid] %x₀
- stop

B₀
- \%i₀ = ld v[%tid]
- \%j₀ = 0

B₁
- \%i = \%i₀, \%i₁
- \%j = \%j₀, \%j₃
- \%p₀ = %i < 100
- branch %p₀ B₂
What is the influence region of p1?

\[ %i0 = 1d v[\%tid] \]
\[ %j0 = 0 \]

\[ %i = \phi(%i0, %i1) \]
\[ %j = \phi(%j0, %j3) \]
\[ %p0 = %i < 100 \]
branch %p0 B₂

\[ %i1 = %i + 1 \]
\[ %j1 = %j + 1 \]
\[ %t0 = %j1 \mod 2 \]
\[ %p1 = %t0 = 0 \]
branch %p1 B₄

\[ %\text{sync} \]
\[ %p2 = %j > 100 \]
branch %p2 B₇

\[ %x0 = 1 \]
jump B₈

\[ %x = \phi(%x0, %x1) \]
sync
st \%v[\%tid] %x0
stop

\[ %j2 = %j1 - 3 \]

\[ %\text{sync} \]
\[ %x1 = 2 \]

\[ %\text{sync} \]
\[ %\text{jump} B₁ \]
%i0 = ld v[%tid]
%j0 = 0

%i = \phi(%i0, %i1)
%j = \phi(%j0, %j3)
%p0 = %i < 100
branch %p0 B2

%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B4

%j2 = %j1 - 3

sync
jump B8

%x0 = 1
jump B8

%x = \phi(%x0, %x1)
sync
st %v[%tid] %x0
stop

sync
jump B1

The influence region of p1:
What is the influence region of p0?
The influence region of p0:

\[
\begin{align*}
%i_0 &= \text{ld v[}%\text{tid}\text{]} \\
%j_0 &= 0 \\
%i_1 &= %i + 1 \\
%j_1 &= %j + 1 \\
\text{sync} \\
\text{jump} \ B_8 \\
\%x &= \phi(\%x_0, \%x_1) \\
\text{sync} \\
\text{st \ v[}%\text{tid}\text{]} \ %x_0 \\
\text{stop}
\end{align*}
\]
The law of sync dependences

- **Theorem:** Let \texttt{branch \%p B} be a branch instruction, and let \texttt{l/p} be its synchronization point. A variable \( v \) is sync dependent on \%p if, and only if, \( v \) is defined inside the influence region of the branch and \( v \) \textit{reaches} \texttt{l/p}.

  - Again: \texttt{l/p} is the post-dominator of the place where the branch is defined.

  - \textbf{And alert again}: what does it mean for a variable to \textsc{reach} a certain program point?
Which variables defined inside IR(%p0) reach its sync point?
How to transform sync dependences into data dependences?

- We augment phi-functions with predicates. For instance, to create a data dependence between %x and %p2:
How to transform sync dependences into data dependences?

- If a variable is alive outside the IR(%p), then we split its live range, with a unary phi-function.
%i0 = ld v[%tid]
%j0 = 0

%i = !(%i0, %i1)
%j = !(%j0, %j3)
%p0 = %i < 100
branch %p0 B2

%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B4

%j2 = %j1 - 3

%j4 = !(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B7

%x0 = 1
jump B8

%x = !(%x0, %x1), %p2
sync
st v[%tid] %x0
stop

%j3 = !(%j2, %j1), %p1
sync
jump B1
%i0 = ld v[%tid]
%j0 = 0

%i = !(%i0, %i1)
%j = !(%j0, %j3)
%p0 = %i < 100
branch %p0 B2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B4
%j2 = %j1 - 3
%j3 = !(%j2, %j1), %p1
sync
jump B1

%j4 = !(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B7
%x0 = 1
jump B8
%x1 = 2
%x = !(%x0, %x1), %p2
sync
st %v[%tid] %x0
stop

B0
B1
B2
B3
B4
B5
B6
B7
B8

GATED STATIC SINGLE ASSIGNMENT FORM
Sync dependence = data dependence
Great! We know which branches are divergent. **And so what?**

- Thread re-location [Zhang’11].
- Variable sharing.
- Barrier elimination.
- Branch fusion.

Let developers worry about their *algorithms*, while the *compiler* takes care of *divergences*. 
Branch Fusion

Find the longest sequences of common instructions in the two paths of a branch, and merge them.
Let’s consider an example:

```c
__global__ void exampleKernel
(float* u, float* v, float c1, float c2) {
    float y = u[tid];
    if (y != 0.0) {
        float x = v[tid];
        v[tid] = c1*x*x*x/y*y + c2*x;
    } else {
        float x = v[tid];
        v[tid] = c1*x*x/2.0 + c2*x*x;
    }
}
```

What is the longest chain of common instructions?
Let’s try to align the instructions:

\[
T = \begin{array}{ccccccccccc}
\checkmark & \checkmark & \checkmark & \times & \times & \checkmark & \times & \checkmark & \checkmark & \checkmark & \checkmark \\
\downarrow & \ast & \ast & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
\end{array}
\]

\[
F = \begin{array}{ccccccccccc}
\downarrow & \ast & \ast & \ast & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
\downarrow & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\
\end{array}
\]

How to find these chains automatically?
How do we do branch fusion?

• The answer comes from computational biology:
  – **Smith-Waterman** sequence alignment.
  – But, instead of *genes*, we match *instructions*. 
# Smith-Waterman

![Smith-Waterman Table](image)
The Profitability Matrix

- Each cell is the profit of merging two instructions.
- What represents the most profitable path inside the matrix?
- What are the diagonals in this path?
- What are the vertical and horizontal paths?
  - Is there a cost in leaving a diagonal?
  - Is there a cost in entering a diagonal?
Touché!

1. \( t_0 = u[t_{idx}] \)
2. \( p_2 = \text{ne } t_1 \ 0.0 \)
3. bra \( p_2 \) (12)

12. \( t_1 = v[t_{idx}] \)
13. \( t_2 = t_1 \ast t_1 \)
14. \( t_3 = t_2 \ast t_1 \)
15. \( t_4 = t_3 \ast c_1 \)
16. \( t_5 = t_4 / t_0 \)
17. \( t_6 = t_5 / t_0 \)
18. \( t_7 = t_1 \ast c_2 \)
19. \( t_8 = t_6 + t_7 \)
20. \( v[t_{idx}] = t_8 \)

4. \( t_9 = v[t_{idx}] \)
5. \( t_{10} = t_9 \ast t_9 \)
6. \( t_{11} = t_{10} \ast c_1 \)
7. \( t_{12} = t_{11} / 2.0 \)
8. \( t_{13} = t_9 \ast c_2 \)
9. \( t_{14} = t_{13} \ast t_9 \)
10. \( t_{15} = t_{12} + t_{14} \)
11. \( v[t_{idx}] = t_{15} \)

1. \( t_0 = u[t_{idx}] \)
2. \( p_2 = \text{ne } t_1 \ 0.0 \)
3. \( t_{1.9} = v[t_{idx}] \)
4. \( t_{2.10} = t_{1.9} \ast t_{1.9} \)
5. \( s_1 = p_2 \ ? t_{2.10} \ : c_1 \)
6. \( t_{3.11} = t_{2.10} \ast s_1 \)
7. bra \( p_2 \) (13)

13. \( t_4 = t_3 \ast c_1 \)
14. \( t_5 = t_4 / t_0 \)

8. \( s_2 = p_2 \ ? t_5 \ : t_3.11 \)
9. \( s_3 = p_2 \ ? t_0 \ : 2.0 \)
10. \( t_{6.12} = s_2 \ / s_3 \)
11. \( t_{7.13} = t_{1.9} \ast c_2 \)
12. bra \( p_2 \) (16)

15. \( t_{14} = t_{7.13} \ast t_{1.9} \)

16. \( t_{8.15} = t_{6.12} + t_{7.13} \)
17. \( v[t_{idx}] = t_{8.15} \)
The matrix and the program

1 \( \texttt{t0} = \texttt{u[\texttt{tidx}]} \)
2 \( \texttt{p2} = \texttt{ne \ t1 0.0} \)
3 \( \texttt{t1_9} = \texttt{v[\texttt{tidx}]} \)
4 \( \texttt{t2_10} = \texttt{t1_9 * t1_9} \)
5 \( \texttt{s1} = \texttt{p2 ? t2_10 : c1} \)
6 \( \texttt{t3_11} = \texttt{t2_10 * s1} \)
7 \texttt{bra \ p2 \ (13)}

13 \( \texttt{t4} = \texttt{t3 * c1} \)
14 \( \texttt{t5} = \texttt{t4 / t0} \)

8 \( \texttt{s2} = \texttt{p2 ? t5 : t3_11} \)
9 \( \texttt{s3} = \texttt{p2 ? t0 : 2.0} \)
10 \( \texttt{t6_12} = \texttt{s2 / s3} \)
11 \( \texttt{t7_13} = \texttt{t1_9 * c2} \)
12 \texttt{bra \ p2 \ (16)}

15 \( \texttt{t14} = \texttt{t7_13 * t1_9} \)
16 \( \texttt{t8_15} = \texttt{t6_12 + t7_13} \)
17 \( \texttt{v[\texttt{tidx}]} = \texttt{t8_15} \)
Experiments

• Analysis and optimization implemented on top of the Ocelot PTX compiler.
  – There is a tutorial in this PACT about Ocelot: **TM2 (Monday Afternoon)**

• We have tested our algorithms on CUDA, running on a GPU Nvidia GeForce GTX 260.
  – We compare results with data from a profiler.

• We have compiled 30 applications publicly available (80 kernels).
• 14,861 out of 38,150 variables are non-divergent (39%).
• 26% of all the branches are non-divergent.

Number of variables: 38,150
Number of divergent variables: 23,289
Percentage of non-divergent variables: 39%
Precision of the Divergence Analysis

Number of branches: 742

Number of correct predictions: 490

Hit rate: 66%
False positives: 34%
False negatives: 0
- The Divergence analysis is linear on the size of the dependence graph.
  - Might be quadratic on the number of variables, if this graph is dense.
  - In practice, the analysis runs in $O(V)$, where $V$ is the number of variables in the source program.
Opportunities for Branch Fusion

Number of branches: 742
Number of unifiable branches: 196
Number of unifiable divergent branches: 137
Number of profitable unifications: 30
Performance Gains of Branch Fusion

• Gains in the publicly available benchmarks were modest:
  – Rodinia LDU: 0.37% and 1.35% (2 branches)
  – Rodinia Heartwall: 0.62% (1 branch)
  – Rodinia SRAD: 0.48% (1 branch)
  – SDK Mergesort: 3.59% (1 branch)
  – Cederman’s Parallel Quicksort: 3.09% (1 branch)
• For our testing benchmarks, gains of up to 26.78%.
• Manual branch fusion gave us 9.2% in Cederman’s Quicksort, and 11.5% in SRAD.
  – The automatic optimization stops at the boundaries of basic blocks.
Conclusion

• We have presented a static analysis that finds non-divergent variables in SIMD programs.
• We have also presented branch fusion: an optimization that mitigates the impact of divergences.
• Code is publicly available in the Ocelot repository, and in our webpage (http://divmap.wordpress.com).