DARM: Control-Flow Melding for SIMT Thread Divergence Reduction

Charitha Saumya, Kirshanthan Sundararajah, Milind Kulkarni

CGO ‘22 April 2nd – 6th 2022
General Purpose Computing on Graphics Processing Units (GPGPU)

General Purpose Computing on Graphics Processing Units (GPGPU)

Tricky to get performance on *Irregular Applications*

Irregularity in GPU Applications

Memory Divergence

Control-Flow Divergence
Irregularity in GPU Applications

Memory Divergence

Control-Flow Divergence
Irregularity in GPU Applications

Memory Divergence

Control-Flow Divergence

Why irregular control-flow is bad for performance?
Single-Instruction-Multiple-Threads (SIMT) Execution Model

- Threads are arranged in groups (warp/wavefront)
- Lockstep execution among threads in a group

Massive Data Parallelism

Relatively Energy Efficient

SPMD-style Programming
Single-Instruction-Multiple-Threads (SIMT) Execution Model

- Threads are arranged in groups (warp/wavefront)
- Lockstep execution among threads in a group

Massive Data Parallelism
- Relatively Energy Efficient
- SPMD-style Programming

\[
\]

```
LOAD T1[0:3], A[0:3]
LOAD T2[0:3], B[0:3]
MUL T3[0:3], T1[0:3], T2[0:3]
STORE P[0:3], T3[0:3]
```
Control-Flow Divergence

- Threads are arranged in groups (warp/wavefront)
- Lockstep execution among threads in a group


\[ \text{if } (P[tid] > 256) \]

\[ \ldots \]

\[ \text{else} \]

\[ \ldots \]

Threads can diverge at branches

\begin{align*}
\text{T0} & \quad \text{T1} & \quad \text{T2} & \quad \text{T3} \\
\text{LOAD T1[0:3], A[0:3]} & \quad & \quad & \\
\text{LOAD T2[0:3], B[0:3]} & \quad & \quad & \\
\text{MUL T3[0:3], T1[0:3], T2[0:3]} & \quad & \quad & \\
\text{STORE P[0:3], T3[0:3]} & \quad & \quad & \\
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
\text{BRANCH C[0:3], IF, ELSE} & \quad & \quad & \\
\text{??} & \quad \text{??} & \quad \text{??} & \quad \text{??} \\
\end{align*}

\[ \text{???} \]
Control-Flow Divergence

Contains a Divergent Branch
Control-Flow Divergence

'B:
...
%a = load ..
%a = mul ..
%b = div ..
....

'C:
...
%p = load ..
%q = mul ..
%r = div ..
....

D

B

A

C

D

A

B

C

D

time
Control-Flow Divergence

%B:
...  
%a = load .. 
%a = mul .... 
%b = div .... 
....

%C:
...  
%p = load .. 
%q = mul .... 
%r = div .... 
....

Convergent execution of common instructions
Existing Compiler-based Solutions

Move common instructions to convergent regions

Tail Merging

Identical instruction sequences are moved to common successor
Existing Compiler based Solutions

Branch Fusion using Instruction alignment

Existing Compiler based Solutions

Branch Fusion using Instruction alignment

Existing Compiler based Solutions

Branch Fusion using Instruction alignment

Limited to diamond-shaped control-flow

Convergent execution of common instructions

__global__ static void bitonicSort(int *values) {
    // copy data from global memory to shared memory
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if (((tid & k) == 0) {
                    if (shared[ixj] < shared[tid])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[ixj] > shared[tid])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
    } // write data back to global memory
```c
__global__ static void bitonicSort(int *values) {
    // copy data from global memory to shared memory
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[ixj] < shared[tid])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[ixj] > shared[tid])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
    }
    // write data back to global memory
}
```

Divergent branch
Bitonic Sort

```c
__global__ static void bitonicSort(int *values) {
    // copy data from global memory to shared memory
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[ixj] < shared[tid])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[ixj] > shared[tid])
                        swap(shared[tid], shared[ixj]);
                }
            }
        }
        __syncthreads();
    }
    // write data back to global memory
}
```

Divergent paths contain Control-flow Regions with Similar computations
Bitonic Sort

```c
__global__ static void bitonicSort(int *values) {
    // copy data from global memory to shared memory
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[ixj] < shared[tid])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[ixj] > shared[tid])
                        swap(shared[tid], shared[ixj]);
                }
            }
        }
    }
    __syncthreads();
}
```

Tail Merging or Branch Fusion cannot merge control-flow regions

Divergent paths contain Control-flow Regions with Similar computations
Control-Flow Melding

```c
__global__ static void bitonicSort(int *values) {
    // copy data from global memory to shared memory
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[ixj] < shared[tid]) {
                        swap(shared[tid], shared[ixj]);
                    }
                } else {
                    if (shared[ixj] > shared[tid]) {
                        swap(shared[tid], shared[ixj]);
                    }
                }
            __syncthreads();
        }
    }
    // write data back to global memory
}
```
Contributions

- **Divergence-Aware-Region-Melder (DARM)**, a realization of Control-Flow-Melding that can find and meld similar control-flow regions to reduce divergence

- Implementation of DARM in LLVM

- Evaluation of DARM on synthetic and real-world benchmarks showing its effectiveness
Detecting Divergent Regions

- **Divergent Region**
  - $E$ has a divergent branch
  - $E$’s two successors do not post-dominate each other
Single-Entry Single-Exit (SESE) Subgraphs

- Split the region into SESE subgraphs
- SESE subgraph
  - SESE region (e.g. $S_B$, $S_C$)
  - Single basic block with single successor and single predecessor (e.g. $S_A$, $S_D$)
Two subgraphs are **meldable** if they fall into one of following 3 categories

1. Region - Region
   - Contains more than one basic block
   - Isomorphic
Meldable Subgraphs

Two subgraphs are *meldable* if they fall into one of following 3 categories

1. Region - Region
   - Contains more than one basic block
   - Isomorphic

2. Basic block - Region
   - Single basic block expanded to match a region (aka *Region Replication*)
Meldable Subgraphs

Two subgraphs are *meldable* if they fall into one of following 3 categories

1. Region - Region
   - Contains more than one basic block
   - Isomorphic

2. Basic block - Region
   - Single basic block expanded to match a region (aka *Region Replication*)
Meldable Subgraphs

Two subgraphs are **meldable** if they fall into one of following 3 categories

1. Region - Region
   - Contains more than one basic block
   - Isomorphic

2. Basic block - Region
   - Single basic block expanded to match a region (aka Region Replication)
Meldable Subgraphs

Two subgraphs are **meldable** if they fall into one of following 3 categories

1. **Region - Region**
   - Contains more than one basic block
   - Isomorphic

2. **Basic block - Region**
   - Single basic block expanded to match a region (aka **Region Replication**)

**Always taken branch**

**PHI nodes added to flow values produced at %A**
Meldable Subgraphs

Two subgraphs are **meldable** if they fall into one of following 3 categories

1. Region - Region
   - Contains more than one basic block
   - Isomorphic

2. Basic block - Region
   - Single basic block expanded to match a region (aka Region Replication)

---

**Region - Region**

Always taken branch

**Basic block - Region**

PHI nodes added to flow values produced at %A

---

30
Meldable Subgraphs

Two subgraphs are **meldable** if they fall into one of following 3 categories

1. Region - Region
   - Contains more than one basic block
   - Isomorphic

2. Basic block - Region
   - Single basic block expanded to match a region (aka *Region Replication*)

3. **Basic block – Basic block**
Profitable Subgraph Alignment

- A divergent region $\rightarrow$ many SESE subgraphs

How to decide which subgraphs to meld?
Profitable Subgraph Alignment

- A divergent region $\rightarrow$ many SESE subgraphs

**How to decide which subgraphs to meld?**

- Subgraphs are aligned based on their *Melding Profitability*

- **Melding Profitability**: computed based on *instruction frequencies*
Subgraph Instruction Alignment

Pre-order Linearization
Subgraph Instruction Alignment

Pre-order Linearization

Instruction alignment for A and B

%add = add nsw i32 %0, %1
%sub = sub nsw i32 %2, %3
%or = or i32 %sub2, %4
%div = sdiv i32 %2, 5
%shl = shl i32 %add, 2
%mul = mul nsw i32 %4, 5
%and = and i32 %shl, %sub
br label %R

%add1 = add nsw i32 %4, %5
%sub2 = sub nsw i32 %2, %5
%xor = xor i32 %4, 5
%div3 = sdiv i32 %or, 4
%mul4 = mul nsw i32 %xor, %sub2
%add5 = and i32 %xor, %sub2
br label %S
DARM Code Generation
DARM Code Generation

Generated melded control-flow
Ensuring Correctness

- Melding can break the use-def chains that go through the melded regions
Ensuring Correctness

- Melding can break the def-use chains outside the melded regions

\[ %A: \]
\[ S_T \rightarrow %B: \]
\[ \ldots %a = \ldots \]
\[ \ldots \]
\[ \rightarrow %E: \]
\[ \ldots %x = \text{mul} %a \ldots \]
\[ \ldots \]

\[ %C: \]

\[ S_F \rightarrow %A: \]
\[ \ldots %a = \ldots \]
\[ \ldots \]
\[ \rightarrow %E: \]
\[ \ldots %x = \text{mul} %a \ldots \]
\[ \ldots \]

\[ %C: \]

\[ S_{T,F} \rightarrow %B: \]
\[ \ldots %a = \ldots \]
\[ \ldots \]
\[ \rightarrow %E: \]
\[ \ldots %x = \text{mul} %a \ldots \]
\[ \ldots \]

\[ %C: \]

\[ %a \text{ no longer dominates } %x ! ! \]
Ensuring Correctness

- Melding can break the def-use chains outside the melded regions
Ensuring Correctness

- Melding PHI nodes

%A:
... 
%a = .... 
...

%B:
... 
%b = .... 
...

%C:
%p1 = phi [a, %A], [%b, %B] 
....

%P:
... 
%p = .... 
...

%Q:
... 
%q = .... 
...

%R:
%p2 = phi [p, %P], [%q, %Q] 
....
Ensuring Correctness

- Melding PHI instructions

%A:
...  
%a = ....  
...

%B:
...  
%b = ....  
...

%C:
%p1 = phi [a, %A], [%b, %B]  
....

%P:
...  
%p = ....  
...

%Q:
...  
%q = ....  
...

%A_P:
%a_p = .....  
...

%B_Q:
...  
%b = ...  
%q = ....  
...

%C_R:
%p1 = phi [a_p, %A_P], [%b, %B_Q]  
%p2 = phi [a_p, %A_P], [%q, %B_Q]  
....
Ensuring Correctness

• Melding unaligned instructions

Full-predication of unaligned instructions can be problematic

• False positives in divergence analysis
• Instructions with side effects
Ensuring Correctness

- Unpredication

execute unaligned instructions conditionally

%add = add nsw i32 %0, %1
%sub = sub nsw i32 %2, %3
%div = sdiv i32 %2, 5

%add1 = add nsw i32 %4, %5
%sub2 = sub nsw i32 %2, %5
%a = or i32 %sub2, %4
%div3 = sdiv i32 %a, 4

%M:
  // instructions
  br i1 %cmp, label %M.tail, label %M.split

%M.split:
  %8 = or i32 %7, %4
  %9 = xor i32 %4, %5
  br label %M.tail

%M.tail:
  %10 = phi i32 [ %8, %M.split ], [ undef, %M]
  %11 = phi i32 [ %9, %M.split ], [ undef, %M]
  // instructions
Ensuring Correctness

- Melding unaligned instructions

More Details in the paper!
Implementation

- LLVM pass integrated with LLVM *opt*
- Uses LLVM’s built-in Divergence Analysis
- Currently Targeting AMD GPUs with *ROCm HIPCC* compiler
Implementation

- LLVM pass integrated with LLVM *opt*
- Uses LLVM’s built-in Divergence Analysis
- Currently Targeting AMD GPUs with *ROCm HIPCC* compiler

**Evaluation Setup**

*Hardware*: AMD Radeon Pro Vega 20 GPU + AMD Ryzen CPU

*Baseline*: Full-optimizations (-O3)

*Branch Fusion*: Implemented on top of DARM
Benchmarks

Synthetic Benchmarks

Access DARM’s generality using a variety of divergent control-flow patterns
Benchmarks

Synthetic Benchmarks

1. Bitonic Sort (BIT)
2. Parallel and Concurrent Merge (PCM)
3. Mergesort (MS)
4. LU-decomposition (LUD)
5. N-Queens (NQU)
6. Speckle Reducing Anisotropic Diffusion (SRAD)
7. DCT Quantization (DCT)

Real-World Benchmarks

Access DARM’s generality using a variety of divergent control-flow patterns

Already heavily optimized
DARM is more general than *Branch Fusion* and can reduce control-flow divergence in a variety of cases.
Real-World Benchmarks Performance

![Graph showing speedup for various benchmarks with DARM and BF categories.](image-url)
Real-World Benchmarks Performance

Only DARM can fully meld the control-flow structures present in BIT, PCM and NQU
Melding shared memory instructions is important for achieving good performance.
DARM either matches or improves the performance of Branch Fusion or Baseline (in most cases)
Compile-time Overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>O3 (seconds)</th>
<th>DARM (seconds)</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>0.4804</td>
<td>0.5018</td>
<td>1.0444</td>
</tr>
<tr>
<td>PCM</td>
<td>0.5690</td>
<td>0.5942</td>
<td>1.0443</td>
</tr>
<tr>
<td>MS</td>
<td>0.8037</td>
<td>0.8064</td>
<td>1.0035</td>
</tr>
<tr>
<td>LUD</td>
<td>0.5993</td>
<td>0.6294</td>
<td>1.0502</td>
</tr>
<tr>
<td>NQU</td>
<td>0.4687</td>
<td>0.4738</td>
<td>1.0109</td>
</tr>
<tr>
<td>SRAD</td>
<td>0.4999</td>
<td>0.5121</td>
<td>1.0244</td>
</tr>
<tr>
<td>DCT</td>
<td>0.4398</td>
<td>0.4439</td>
<td>1.0093</td>
</tr>
</tbody>
</table>

Compile-time overhead of DARM is not significant
# Compile-time Overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>O3 (seconds)</th>
<th>DARM (seconds)</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>0.4804</td>
<td>0.5018</td>
<td>1.0444</td>
</tr>
<tr>
<td>PCM</td>
<td>0.5690</td>
<td>0.5942</td>
<td>1.0443</td>
</tr>
<tr>
<td>MS</td>
<td>0.8037</td>
<td>0.8064</td>
<td>1.0035</td>
</tr>
<tr>
<td>LUD</td>
<td>0.5993</td>
<td>0.6294</td>
<td>1.0502</td>
</tr>
<tr>
<td>NQU</td>
<td>0.4687</td>
<td>0.4738</td>
<td>1.0109</td>
</tr>
<tr>
<td>SRAD</td>
<td>0.4999</td>
<td>0.5121</td>
<td>1.0244</td>
</tr>
<tr>
<td>DCT</td>
<td>0.4398</td>
<td>0.4439</td>
<td>1.0093</td>
</tr>
</tbody>
</table>

Checkout our paper for more detailed evaluation of DARM including ALU utilization, memory instruction counters.
Summary

- Control-flow divergence can be a significant performance bottleneck in GPU programs
- Existing compiler optimizations are not general enough
- We propose DARM, a general framework for reducing control-flow divergence in the presence of arbitrary control-flow
- DARM shows good performance on real-world benchmarks
- Future work: Applicability of DARM for code size reduction and accelerating program testing

Contact:
cgusthin@purdue.edu

Code:
github.com/charitha22/cgo22ae-darm-code

Benchmarks:
github.com/charitha22/cgo22ae-darm-benchmarks

This presentation and recording belong to the authors. No distribution is allowed without the authors' permission
DARM: Control-Flow Melding for SIMT Thread Divergence Reduction

Charitha Saumya, Kirshanthan Sundararajah, Milind Kulkarni

CGO ‘22  April 2nd – 6th  2022