Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures

Yunsup Lee, UC Berkeley
Vinod Grover, NVIDIA
Ronny Krashinsky, NVIDIA
Mark Stephenson, NVIDIA
Stephen W. Keckler, NVIDIA
Krste Asanovic, UC Berkeley
Why is Supporting Control Flow Challenging in Data-Parallel Architectures?

```c
for (i=0; i<16; i++) {
    a[i] = op0;
    b[i] = op1;
    if (a[i] < b[i]) {
        c[i] = op2;
    } else {
        c[i] = op3;
    }
    d[i] = op4;
}
```

- The divergence management architecture must not only partially sequence all execution paths for correctness,
- but also reconverge threads from different execution paths for efficiency.
For GPUs, Supporting Complex Control Flow in a SPMD Program is not Optional!

- Traditional vector compilers can always give up and run complex control flow on the control processor.
- However, for the SPMD compiler, supporting complex control flow is a functional requirement rather than an optional performance optimization.
Design Space of Divergence Management

Software
Explicitly Scheduled by Compiler

Hardware
Implicitly Managed by Microarchitecture

Predication + Fork/Join

Vector Predication
(Used in Limited Cases)

We can perform a design space exploration of divergence management on NVIDIA GPU silicon

Divergence Stack
(Compiler Figures Out Reconvergence Points)
Executive Summary, Contributions of Paper

- 28 Benchmarks written in CUDA (Parboil, Rodinia, FFT, nqueens)
- CUDA 6.5 Production Compiler
  - Divergence Stack
  - Limited Predication
- Modified CUDA 6.5 Production Compiler
  - Full Predication with New Compiler Algorithms
- CUDA Binary
- NVIDIA Tesla K20c (Kepler, GK110)
- Performance, Statistics

(1) Detailed Explanation and Categorization of Hardware and Software Divergence Management Schemes
(2) SPMD Predication Compiler Algorithms
(3) Apples-to-Apples Comparison using Production Silicon and Compiler

Performance with predication is on par compared to performance with divergence stack
How does the hardware divergence stack and software predication handle control flow?
If-Then-Else Example: Divergence Stack

- LLVM compiler takes a CUDA program and generates PTX, which encodes all data/control dependences
- Note, all instructions on the right (despite being a scalar instruction) are executed in a SIMD fashion
If-Then-Else Example: Divergence Stack

- “ptxas” backend compiler takes PTX and generates SASS instructions, which executes natively on GPU
- Reconvergence points are analyzed and inserted by the backend compiler

CUDA Program ➔ LLVM Compiler ➔ PTX ➔ “ptxas” Backend Compiler ➔ SASS

Kernel() {
    a = op0;
    b = op1;
    if (a < b) {
        c = op2;
    } else {
        c = op3;
    }
    d = op4;
}

Kernel<<<n>>>();

a = op0
b = op1
p = slt a, b
push ipdom
branch.eqz p, else
c = op2.pop
else:
c = op3.pop
ipdom:
d = op4
### If-Then-Else Example: Divergence Stack

- **Push:** pushes `<reconverge pc, current mask>` to stack
- **Pop:** disregards current `<pc, mask>`, pops top of stack, and executes deferred `<pc, mask>`

#### Mask | OP | Divergence Stack
| 1111 | op0 |  
| 1111 | op1 |  
| 1111 | slt |  
|  | push |  
| 1111 | branch |  
| 1100 | op2 |  
|  | pop |  
| 0011 | op3 |  
|  | pop |  
| 1111 | op4 |  

Assume 4 threads are executing.
Assume thread 0 and 1 took the branch.

```plaintext
a = op0
b = op1
p = slt a,b
push ipdom
branch.eqz p, else
c = op2.pop

e else:
c = op3.pop
ipdom:
d = op4
```
If-Then-Else Example: Predication

The compiler can schedule instructions
Predicates also encode reconvergence information
Uniform Branch Conditions Across All Threads

- What if the branch condition is uniform across all threads?
  - Execute instructions with a null predicate
  - Branches don’t push a token to the divergence stack when branch condition is uniform.

**Divergence Stack**

\[
\begin{align*}
  a &= \text{op0} \\
  b &= \text{op1} \\
  p &= \text{slt } a, b \\
  \text{push } \text{ipdom} \\
  \text{branch.eqz } p, \text{ else} \\
  c &= \text{op2.pop} \\
\end{align*}
\]

**Predication**

\[
\begin{align*}
  a &= \text{op0} \\
  b &= \text{op1} \\
  f0 &= \text{slt } a, b \\
  \text{@f0 } c &= \text{op2} \\
  \text{@!f0 } c &= \text{op3} \\
  d &= \text{op4} \\
\end{align*}
\]
Runtime Branch-Uniformity Optimization with Consensual Branches

Kernel() {
    a = op0;
    b = op1;
    if (a < b) {
        c = op2;
    } else {
        c = op3;
    }
    d = op4;
}
Kernel<<<n>>>()

a = op0
b = op1
f0 = slt a,b
cbranch.ifnull f0, else
@f0 c = op2
else:
cbranch.ifnull !f0, ipdom
@!f0 c = op3
ipdom:
d = op4

• We can optimize the predicated code with a *consensual branch* (*cbranch*), which is taken only when all threads *consensually* agree on the branch condition (*ifnull*).
• The code may jump around unnecessary work.
Static Branch-Uniformity Optimization with Consensual Branches

Kernel() {
    a = op0;
    b = op1;
    if (a < b) {
        c = op2;
    } else {
        c = op3;
    }
    d = op4;
}
Kernel<<<n>>>();

• If the compiler can prove that the branch condition is uniform across all threads, the compiler can omit the guard predicates.
Loop Example: Consensual Branches are Key to Compile Loops with Predication

```c
Kernel() {
    done = false;
    while (!done) {
        a = op0;
        b = op1;
        done = a < b;
    }
    c = op2;
}
Kernel<<<n>>>();
```

- f0 = true
- loop:
  - `cbranch.ifnull f0, exit`
  - @f0 a = op0
  - @f0 b = op1
  - @f0 f1 = slt a, b
  - f0 = and f0, !f1
  - j loop
- exit:
  - c = op2

Thread-Aware Predication

- Intuitively, the compiler needs to sequence loop until all threads are done executing the loop.
- A consensual branch (`cbranch.ifnull`) is used to check whether loop mask (f0) is null.
Thread-Aware (TA) Predication Compiler Algorithms
Kernel() {
    N1; N2;
    if (!P1) {
        N3;
    } else {
        N4;
        if (!P2) {
            N5;
        } else {
            N6;
        }
    }
    N7;
}  
N8;
Kernel<<<n>>>(());

Control Flow Graph (CFG)

Guard Predicate = P1 && !P2

Generate CFG, CDG

Walk CDG to Get Guard Predicates for all BBs

Linearize Control Flow

Predicate all instructions with guard predicate and rewire all BBs
Runtime Branch-Uniformity Optimization

Generate CFG, CDG
Walk CDG to Get Guard Predicates for all BBs
Linearize Control Flow
Predicate all instructions with guard predicate and rewire all BBs

Kernel() {
  N1; N2;
  if (!P1) {
    N3;
  } else {
    N4;
    if (!P2) {
      N5;
    } else {
      N6;
    }
  }
  N7;
  N8;
}
Kernel<<<n>>>();

Assume compiler cannot prove that P1 is uniform across all threads

Add consensual branch if P1 is null
Add consensual branch if !P1 is null
**Static Branch-Uniformity Optimization**

1. **Generate CFG, CDG**
2. **Walk CDG to Get Guard Predicates for all BBs**
3. **Linearize Control Flow**
4. **Predicate all instructions with guard predicate and rewire all BBs**

**Kernel()**

```c
Kernel() {
    N1; N2;
    if (!P1) {
        N3;
    } else {
        N4;
        if (!P2) {
            N5;
        } else {
            N6;
        }
        N7;
    }
    N8;
}
```

Assume compiler can prove that P1 is uniform across all threads.

Guard Predicate = P2
Predication Compiler Algorithms: Loops

Kernel() {
    if (!P1) {
        while (!P2) {
            if (!P3) {
                break;
            }
        }
    }
}
Kernel<<<n>>>();
Supporting Complex Control Flow

- **Function Calls**
  - Support by a straightforward calling convention

- **Virtual Function Calls**
  ```
  @p2 jalr r3
  
  With Divergence Stack
  
  With Predication
  ```
  ```
  @p0 jalr r4
  p2 = p2 and !p0
  cbranch.ifany p2, loop
  ```

- **Irreducible Control Flow**
  - Find smallest region containing irreducible control flow, and insert sequencing code at the entry block and exit block to sequence active threads through the region one-by-one.
Evaluation
Theoretically we could have implemented our thread-aware predication pass in ptxas.

- Implemented bulk of the predication pass in LLVM for fast prototyping.
Evaluation: Quick Recap

28 Benchmarks written in CUDA (Parboil, Rodinia, FFT, nqueens)

CUDA 6.5 Production Compiler
  Divergence Stack
  Limited Predication

Modified CUDA 6.5 Production Compiler
  Full Predication with TA Compiler Algorithms

CUDA Binary

NVIDIA Tesla K20c (Kepler, GK110)

Performance, Statistics

Five Bars
1) Baseline (Divergence Stack)
2) Limited Predication
3) TA Predication
4) TA+SBU (static branch-uniformity optimization)
5) TA+SBU+RBU (runtime branch-uniformity optimization)

Compare performance and statistics
Performance Results: Geomean

- Thread-aware predication compiler is competitive with the baseline compiler (divergence stack)
- Both static and runtime branch-uniformity optimizations play an important role
- Performance doesn’t change with limited if-conversion heuristic implemented in production compiler
Performance Results: Speedups

- Predication can expose better scheduling opportunities
- Extra consensual branches added for +RBU may act as scheduling barriers

Yunsup Lee – UC Berkeley
Performance Results: Slowdowns

- Ten benchmarks are inconclusive (>90%, <100%)
Performance Results: Slowdowns

- Ten benchmarks are inconclusive (>90%, <100%)
- Five benchmarks in <90% range
  - Increase in register pressure can reduce occupancy, which sometimes reduce performance
  - Compiler is not able to optimize for all branch uniformity exhibited during runtime
Discussion on Area, Power, Energy

• Hard to quantify impact of predication on area, power, energy
  • Experiment done on GPU silicon
• Primary motivation of software divergence management is to reduce \textit{hardware design complexity} and \textit{associated verification costs}
• Area and power overhead of divergence stack are not significant
  • Performance $\approx$ power/energy consumption
  • Power/Energy consumption of software divergence mgmt. $\approx$ Power/Energy of hardware divergence mgmt.
Fundamental Advantages of Software Divergence Management

- Short-circuit example
- Divergence stack cannot reconverge threads at N4
- Predication can reconverge threads at N4

Control Flow Graph
Conclusions

• Advantages of Divergence Stack
  • Enables a fairly conventional thread compilation model
  • Register allocation easier
  • Simplifies the task of supporting irreducible control flow

• Advantages of Predication
  • Simplifies the hardware without sacrificing programmability
  • Actual cases where predication can outperform divergence stack
    • Better scheduling opportunities
    • Better reconvergence of threads

• For divergence management, pushing complexity to the compiler is a better choice

This work was funded by DARPA award HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, an NVIDIA graduate fellowship, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, Oracle, and Samsung. It was also funded by DOE contract B599861. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.