Convergence and Scalarization for Data-Parallel Architectures

Yunsup Lee¹, Ronny Krashinsky², Vinod Grover², Stephen W. Keckler²,³, Krste Asanovic¹

¹University of California at Berkeley
²NVIDIA
³The University of Texas at Austin
Motivation: Overheads of SPMD

- SPMD languages (e.g., CUDA, OpenCL) are accessible and productive for programming data-parallel architectures
- SIMT architectures reduce some overheads
  - Program counter, instruction fetch
- But, redundancy across threads remains
  - Redundant loop bookkeeping, address calculations
  - Shared data
- **Scalarization** factors out redundancy in SPMD program
Outline

• Motivation and Paper Overview
• Scalarization Example
• Compiler Analyses
• Implementation and Evaluation
• Conclusion
Example: SGEMM, \( C = A \times B \)

```c
float C[32][32], A[32][32], B[32][32]; // assume C is zeroed

for (int i = 0; i < 32; i++)
    for (int j = 0; j < 32; j++)
        for (int k = 0; k < 32; k++)
            C[i][j] += A[i][k] * B[k][j];

float C[32][32], A[32][32], B[32][32]; // assume C is zeroed

sgemm<<<32,32>>>(C, A, B);

__global__ void sgemm(float** C, float** A, float** B)
{
    for (int k = 0; k < 32; k++)
        C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
}
```

Written in loops

Written in SPMD (CUDA)
float C[32][32], A[32][32], B[32][32]; // assume C is zeroed

for (int i = 0; i < 32; i++)
    for (int j = 0; j < 32; j++)
        for (int k = 0; k < 32; k++)
            C[i][j] += A[i][k] * B[k][j];
__global__ void sgemm(float** C, float** A, float** B)
{
    for (int k = 0; k < 32; k++)
        C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
}

Inefficiencies:
1) Redundant loop bookkeeping
2) Redundant address calculation
3) Shared data
4) Affine addresses

ld.f32 r6, [r1];
ld.f32 r0, [r2];
fma.f32 r7, r6, r0, r3;
iadd r0, r1, 4;
iadd r0, r2, 64;
__global__ void sgemm(float** C, float** A, float** B) {
    for (int k = 0; k < 32; k++)
        C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
}

Scalarizing SPMD Compiler
__global__ void sgemm(float** C, float** A, float** B) {
    for (int k = 0; k < 32; k++)
        C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
}

- **Scalarizing compiler** factors out redundant instructions and operands
  - One per warp instead of one per thread
This Paper: Scalarization

```c
__global__ void sgemm(float** C, float** A, float** B)
{
    for (int k = 0; k < 32; k++)
        C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
}
```

Scalarizing SPMD Compiler

- Variance across threads originates with use of
  - thread index (tid.x)
  - volatile and atomic memory accesses
This Paper: Scalarization

Scalarizing SPMD Compiler

• Compiler analyzes SPMD code with
  – Variance analysis: is value thread-invariant?
  – Convergence analysis: is BB convergent (all threads present)?
This Paper: Scalarization

Scalarizing SPMD Compiler

- Transform load/stores, which have unit-strided addresses into warp-sequential load/stores (traditionally vector loads/stores)
  - Lower address bandwidth to the memory system
  - Less reliance on dynamic memory coalescing
This Paper: Scalarization

__global__ void sgemm(float** C, float** A, float** B)
{
    for (int k = 0; k < 32; k++)
        C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
}

Thread-variant

Thread-invariant

Affine

Scalarizing SPMD Compiler

... @s ld.f32 s17, [s2];
    ldwseq.f32 r13, [s14];
    fma.f32 r18, s17, r13, r16;
@s iadd s2, s2, 4;
@s iadd s14, s14, 64;
...
Outline

• Motivation
• Scalarization Example
  • [Our Contribution] Compiler Analyses
    – Combined Convergence and Variance Analysis
    – Affine Analysis
• Implementation and Evaluation
• Conclusion
Variance Analysis

```c
kernel(a, b, c) {
    y = f(tid.x)
    w = f(a)
    u = f(b)
    z = f(y)
    ...
}
```

1. Assume all variables are invariant
Variance Analysis

```
kernel(a, b, c) {
  y = f(tid.x)
  w = f(a)
  u = f(b)
  z = f(y)
  ...
}
```

1. Assume all variables are invariant
2. Mark source of variance (tid.x) as variant
Variance Analysis

```c
kernel(a, b, c) {
    y = f(tid.x) ← Thread-variant
    w = f(a) ← Thread-invariant
    u = f(b)
    z = f(y) ← Thread-variant
    ...
}
```

1. Assume all variables are invariant
2. Mark source of variance (tid.x) as variant
3. Propagate variance forward
kernel(a, b, c) {
    ...
    if (a) {
        if (tid.x) {
            ...
        }
        ...
        ...
        ...
    } else {
        ...
        ...
    }
    ...
    ...
    ...
    if (tid.x) {
        ...
        ...
    }
}

kernel(a, b, c) {
    ...
    if (a) {
        if (tid.x) {
            ...
        }
        ...
    } else {
        ...
    }
    ...
    ...
    ...
    if (tid.x) {
        ...
    }
}
Convergence Analysis

```
kernel(a, b, c) {
    ...
    if (a) {
        if (tid.x) {
            ...
        }
        ...
    } else {
        ...
    }
    ...
}
```

Simple structural convergence analysis
Convergence Analysis

```c
kernel(a, b, c) {
    ...
    if (a) {
        if (tid.x) {
            ...
        }
    } else {
        ...
    }
    ...
    ...
    if (tid.x) {
        ...
    }
}
```

Simple structural convergence analysis

Exit optimization
Convergence Analysis

```
kernel(a, b, c) {
    ...
    if (a) { // scalar branch
        if (tid.x) {
            ...
        }
    }
    ...
    ...
} else {
    ...
}
    ...
    ...
    if (tid.x) {
        ...
    }
}
```

- Combined convergence and variance analysis
- Simple structural convergence analysis
- Exit optimization
Convergence Analysis

```c
kernel(a, b, c) {
  ...
  if (a) { // scalar branch
    if (tid.x) { // HW checks
      ...
    }
  }
  ...
  ...
  } else {
    ...
  }
  ...
  ...
  if (tid.x) {
    ...
  }
}
```

- Dynamic hardware convergence preservation
- Combined convergence and variance analysis
- Simple structural convergence analysis
- Exit optimization
Combined Convergence and Variance Analysis

- Optimistically mark every BB as convergent, every instruction as thread-invariant.
- Initialize a worklist of instructions that
  - Reads tid.x
  - Accesses memory atomically
- Iterate until worklist is empty:
  - Pop instruction from worklist, and mark as thread-variant
  - Add every thread-invariant data-flow successor to worklist
  - If conditional branch, propagate divergence to all convergent blocks that are iteratively control dependent. Add every instruction in BB that are newly marked as divergent to worklist.
Affine Analysis

• First, run the scalar evolution analysis and variance analysis

• Look at the SCEV expression of all load/store addresses, and see if it is affine (unit-strided)
  – ld.f32 rd, [base+idx]
  – ldwseq.f32 rd, [base]
Implementation and Evaluation

• Compiler
  – Modified production CUDA compiler, based on the LLVM infrastructure to support scalarization
  – NVIDIA in-house backend compiler
    • targets RISC-like machine ISA with scalar registers
    • register allocation and instruction scheduling

• Simulator
  – NVIDIA in-house simulator developed
  – Used Ocelot to obtain reference memory dumps before and after kernel launches

• Workloads
  – 23 Rodinia and Parboil Benchmarks
## Convergence Analysis Evaluation

<table>
<thead>
<tr>
<th>Analysis Level</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple structural convergence</td>
<td>32%</td>
</tr>
<tr>
<td>+ Combined convergence and variance</td>
<td>57%</td>
</tr>
<tr>
<td>+ Exit optimization</td>
<td>66%</td>
</tr>
<tr>
<td>+ Dynamic hardware convergence preservation</td>
<td>97%</td>
</tr>
<tr>
<td>Oracle</td>
<td>97%</td>
</tr>
</tbody>
</table>
Savings with Scalarization

Each group shows results for warp sizes of 4, 8, 16, 32.

<table>
<thead>
<tr>
<th>Microarchitectural Action</th>
<th>Savings (WS=32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions issued (in paper)</td>
<td>41%</td>
</tr>
<tr>
<td>Operations executed (graph above)</td>
<td>29%</td>
</tr>
<tr>
<td>Registers read (in paper)</td>
<td>31%</td>
</tr>
<tr>
<td>Registers written (in paper)</td>
<td>30%</td>
</tr>
<tr>
<td>Memory addresses and cache tag checks (in paper)</td>
<td>47%</td>
</tr>
<tr>
<td>Memory data elements accessed (in paper)</td>
<td>38%</td>
</tr>
</tbody>
</table>

*Baseline without scalarization (warpsize of 4)  Scalar  Converged warp (static)  Converged warp (dynamic)  Diverged thread
More Details are in the paper!

- Detailed description and proof of the combined convergence and variance analysis
- More compiler analysis examples
- More evaluation results
- Architecture implications of scalarization
  - Survey of potential SIMT microarchitectures
  - Stackless microarchitectures
- Related work
  - Stratton et al., Coutinho et al., Collange, Karrenberg and Hack, ISPC, and Kerr et al.
Conclusion

• Our combined convergence and variance analysis (static scalarization)
  – Identifies 66% of oracle convergence (100% with dynamic hardware convergence preservation)
  – Leads to 23—31% fewer operations executed and register accesses

• With scalar execution resources added to SIMT architecture, static scalarization can help performance, energy efficiency, and area efficiency

This research was funded in part by DARPA contracts HR0011-10-9-0008 and HR0011-11-C-0100, and an NVIDIA graduate fellowship.