WCCV: Improving the Vectorization of IF-statements with Warp-Coherent Conditions

Huihui Sun, Florian Fey, Jie Zhao, and Sergei Gorlatch

University of Münster, Germany
State key Lab of Mathematical Engineering & Advanced Computing, China
Vectorization with SIMD Extensions

Parallelize operations using **Single-Instruction-Multiple-Data (SIMD)**

Modern architectures provide extensive **hardware support for SIMD**. Less transistors for more flops (energy efficient).

→ **Great future impact** on next generation supercomputers expected!
Current Vectorization Approaches for SIMD Extensions

- Intrinsics: target-specific interface
- SLP or loop vectorization: conservative analysis often fails
- **SPMD-on-SIMD**: explicit data parallelism via data-parallel language

**Idea:** Single-Program-Multiple-Data parallelized with SIMD

- Data-parallel applications are expressed as **compute kernels**.
- **Parallel execution** of different instances (**threads**) of a kernel.
- Groups of threads (**warps**) execute in **lock-step** mode (cf. GPUs).
Vectorized Execution in the SPMD-on-SIMD Paradigm

iteration range

S1

S2

S3

warp
Conditional statements cause control flow divergence!

Ideal case: Conditions behave uniform across iteration range.
Conditional statements cause control flow divergence!

Problem: Diverging control flow between threads of the same warp!

Real world: Control flow divergence occurs!
- Only very few threads take an alternative branch.
Control flow divergence occurs

**IF Conversion (Linearization)** [Allen et al., 1983]

All threads have to execute all branches
→ inactive threads are disabled by **masking instructions**.

- **Widely used** in modern compilers like LLVM and GCC.
- Introduces **redundant computations**.
- Masking instructions and registers are **inefficient** (4x slower).
Consider a **map operation** on a large array of elements of type X or O:

```c
void compute_element()
{
    int tid = get_global_id();
    if (elements[tid] == 'X') // conditional statement
    {
        /* common case */
    }
    else
    {
        /* rare special case */
    }
}
```

**Different types of elements** require different processing.  
**Challenge:** **Branching** introduced by **conditional statements**!
Our Observation: How do conditions behave in practice?

We find that for many applications, conditions

- are divergent across the iteration range;
- but still **uniform across almost all warps**.

We call such conditions **warp-coherent conditions**.
void Gauss(float *m, float *a, int Size, int t) {
    int tid = cfg.get_global(0);
    if (tid < Size-1-t) {
        m[tid + t + 1] = a[Size * (tid + t + 1) + t] / a[Size * t + t];
    }
}

Boolean-step conditions:

- Partial Differential Equations solvers with boundary check conditions
- Search algorithms that evaluate a condition until the first match
Warp-Coherent Conditions in Real-World Applications

```c
void compute_flux(int* elements) {
    int tid = get_global_id(0);
    nb = elements[tid];
    if (nb >= 0) { /* C1 */
        Branch 1 (B1);
    } else if (nb == -1) { /* C2 */
        Branch 2 (B2);
    } else if (nb == -2) { /* C3 */
        Branch 3 (B3);
    }
}
```

High-probability conditions:

- Computational Fluid Dynamics processing aggregate-typed elements
- Raytracing with adjacent shading points and coherent ray directions
Many real-world applications comprise warp-coherent conditions. Many warp-coherent conditions are easy to detect.

Our focus: Boolean-step conditions and high-probability conditions
We detect boolean-step conditions based on static affine analysis. An expression $E(i)$ of the variable $i \in \mathbb{Z}$ is affine iff it can be expressed in the form $E(i) = ai + b$ with coefficients $a, b \in \mathbb{R}$.

We extend the traditional affine analysis for memory access patterns: coefficients and offsets are allowed to be real numbers.

In the previous example: $tid$ and $Size-1-t$ are affine, thus $tid < Size-1-t$ is a boolean-step condition.
void compute_flux(int* elements) {
    int tid = get_global_id(0);
    nb = elements[tid];
    if (nb >= 0) { /* C1 */
        Branch 1 (B1);
    } else if (nb == -1) { /* C2 */
        Branch 2 (B2);
    } else if (nb == -2) { /* C3 */
        Branch 3 (B3);
    }
}

Figures gained by instrumented counters

- Branch B1 was triggered with the highest probability of 97.9%
- Branch B2 with 2% probability
- Branch B3 with the probability of only 0.1%

Branch probability leverages the existing LLVM branch probability pass. Branch cost accumulates weighted cost of instructions.
void compute_flux(int* elements)
{
    int tid = get_global_id(0);
    nb = elements[tid];
    if (nb >= 0) /* C1 */
    {
        Branch 1 (B1);
    } else if (nb == -1) /* C2 */
    {
        Branch 2 (B2);
    } else if (nb == -2) /* C3 */
    {
        Branch 3 (B3);
    }
}
Partial linearization [Moll et al., 2018]

- **Assumption**: Linearization often not required.
- Only linearizes when control flow actually diverges.
- Dynamically checks (any) if branches can be skipped.
- Skips the least likely branch not executed by any thread!
Our contribution: WCCV (Warp-Coherent Condition Vectorization)

- Inserts an all branch and a code variant without masking.
- Skips linearization entirely for coherent warps.
- Works complementary to previous approaches.

Advantage: Skip more and eliminate masking.
In the average case, linearization is entirely skipped.
+ Most warps take the optimized non-masked branch.
+ We observe a speedup of **4.6X** over the scalar version!
Transformation - How WCCV is performed

Implemented in the LLVM IR level.
Transformation - How WCCV is performed

Implemented in the LLVM IR level.

1. Clone the targeted branch
Transformation - How WCCV is performed

Implemented in the LLVM IR level.

1. Clone the targeted branch
2. Generate the runtime **all** check
Transformation - How WCCV is performed

Implemented in the LLVM IR level.

1. Clone the targeted branch
2. Generate the runtime all check
3. Insert the new blocks
We perform a dynamic check once per warp:

- Is the condition’s corresponding mask all-true for all threads?
  - Yes: Call the optimized code variant without masking.
  - No: Call the existing linearized code variant with masking.

+ Avoids redundant computations
+ Reduces the amount of masking
- Only suited for structured control flow without goto labels
We implemented WCCV on top of Region Vectorizer (RV)

Our benchmarks:

- Sandy Bridge (AVX 256-bits) and Skylake (AVX512 512-bits)
- Rodinia Benchmark suite and AOBench (Raytracing)

Source code available: https://github.com/HuihuiSun/WCCV
→ 1.17× over RV

→ 1.47× over scalar code
Runtime Speedups on Skylake

- $\rightarrow 1.14 \times$ over RV
- $\rightarrow 2.11 \times$ over scalar code
Impact of SIMD Vector Width on WCCV

- Sandy Bridge AVX 256-bits
- Skylake AVX512 512-bits

<table>
<thead>
<tr>
<th>Kernel</th>
<th>vs. RV</th>
<th>vs. Scalar</th>
<th>masked RV</th>
<th>Masked WCCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>nn</td>
<td>1.19/1.22</td>
<td>1.03/1.42</td>
<td>42,764</td>
<td>4/12</td>
</tr>
<tr>
<td>kmeans_swap</td>
<td>1.22/1.08</td>
<td>1.00/1.11</td>
<td>494,020</td>
<td>4/4</td>
</tr>
<tr>
<td>particlefilter</td>
<td>1.29/1.70</td>
<td>0.85/2.09</td>
<td>40,000</td>
<td>0/0</td>
</tr>
<tr>
<td>srad</td>
<td>1.31/1.02</td>
<td>2.29/2.82</td>
<td>229,916</td>
<td>0/0</td>
</tr>
<tr>
<td>kmeans_kernel_c</td>
<td>1.28/1.07</td>
<td>3.28/7.02</td>
<td>494,020</td>
<td>4/4</td>
</tr>
<tr>
<td>aobench</td>
<td>1.36/1.74</td>
<td>2.20/4.03</td>
<td>4,179,522</td>
<td>10,342/23,762</td>
</tr>
<tr>
<td>compute_flux</td>
<td>1.36/1.23</td>
<td>2.40/4.62</td>
<td>2,147,483,600</td>
<td>117,767,998/176,711,998</td>
</tr>
</tbody>
</table>

- Skylake has longer vector registers than Sandy Bridge:
  - The runtime checks for WCCV passing less frequently.
  - Less potential for improvement relative to RV.
Comparison with OpenCL on Sandy Bridge

We outperform Intel OpenCL on half of the benchmarks!
WCCV (Warp-Coherent Condition Vectorization)

- detects and exploits WCC to improve vectorization;
- avoids redundant computations;
- reduces execution rate of masked instructions;
- is implemented entirely in LLVM-IR level;
- significantly improves performance for many applications.
Backup Slides
A comparison between Sandy Bridge and Skylake

<table>
<thead>
<tr>
<th>Platform</th>
<th>SIMD extension</th>
<th>SIMD Width</th>
<th>Register number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy Bridge</td>
<td>AVX</td>
<td>256-bit</td>
<td>16</td>
</tr>
<tr>
<td>Skylake</td>
<td>AVX512</td>
<td>512-bit</td>
<td>32</td>
</tr>
</tbody>
</table>

- Double the SIMD width and also double the register number

New in AVX512

- 8 new opmask registers for masking most AVX-512 instructions used to control which values are written to the destinations
- New mini instruction extension operating on the opmask registers like KAND, which bitwise logical AND masks
**compute_flux: BOSCC**

Original CFG:

```
   C1
      \--- B1 \--- C2
          |   \--- B2 \--- C3
              |         \--- B3
```

CFG after BOSCC:

```
   C1
      \--- B1 \--- C2
          |   \--- B2 \--- any(M3)
              |         \--- C3
```

**BOSCC (Bypass-On-Superword-Condition-Code)** [Shin et al., 2005]

- inserts an **any** branch to skip the execution of a masked branch if its mask is all-false
Partial linearization [Moll et al., 2018]

→ only linearizes varying branches (different values for different threads) while preserving uniform branches (same value for different threads)