Westfälische
Wilhelms-Universität
Münster

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!

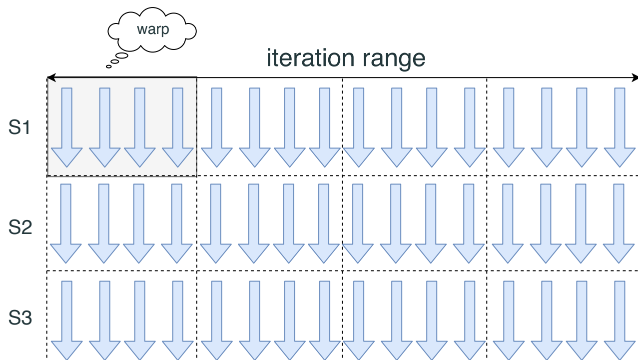
Current Vectorization Approaches for SIMD Extensions

- 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

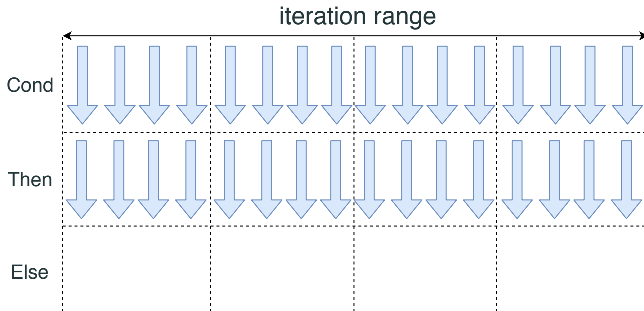
- Data-parallel applications are expressed as **compute kernels**.
- **Parallel execution** of different instances (**threads**) of a kernel.
- Groups of threads (**warps**) execute in **lock-step** mode (cf. GPUs).

Vectorized Execution in the SPMD-on-SIMD Paradigm



Vectorized Execution in the SPMD-on-SIMD Paradigm

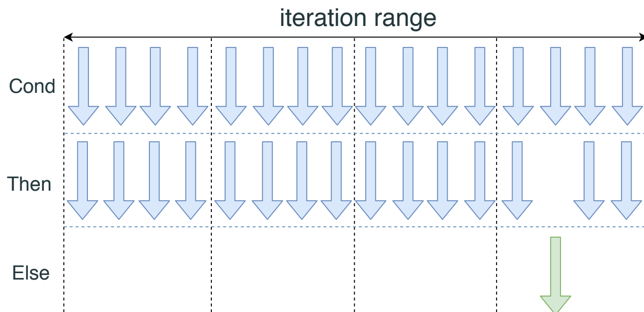
Conditional statements cause **control flow divergence**!



Ideal case: Conditions behave **uniform** across iteration range.

Vectorized Execution in the SPMD-on-SIMD Paradigm

Conditional statements cause **control flow divergence**!
Problem: **Diverging control flow between threads** of the same warp!

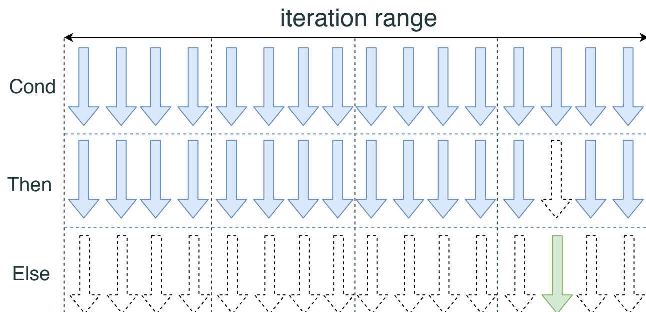


Real world: Control flow divergence occurs!

- 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)**.

A Toy Example for Control Flow Divergence

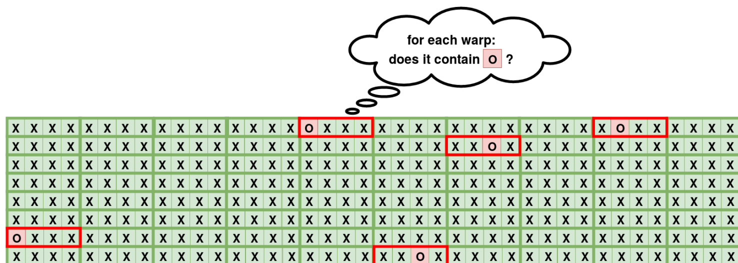
Consider a **map operation** on a large array of `elements` of type X or O:

```
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?



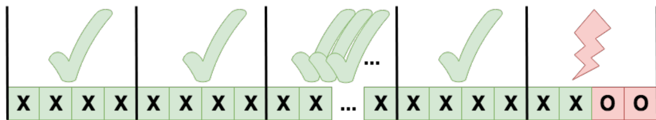
We find that for many applications, conditions

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

We call such conditions **warp-coherent conditions**.

Warp-Coherent Conditions in Real-World Applications

```
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];
  }
}
```

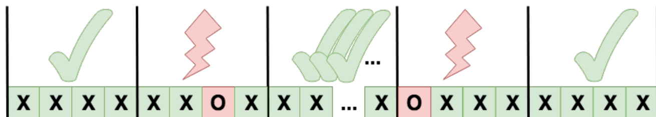


Boolean-step conditions:

- 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

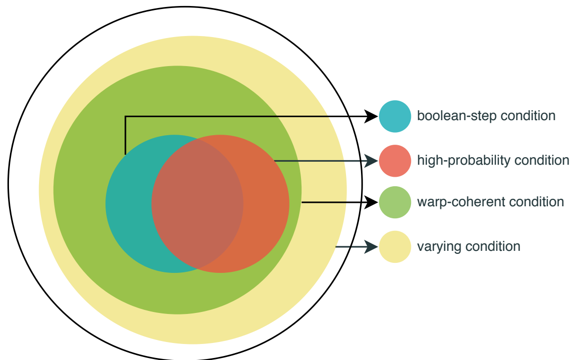
```
void compute_flux(int* elements) {
    int tid = get_global_id(0);
    nb = elements[tid];
    if (nb >= 0) { /* C1 */
        Branch 1(B1);
    } else if (nb == -1) { /* C2 */
        Branch 2 (B2);
    } else if (nb == -2) { /* C3 */
        Branch 3 (B3);
    }
}
```



High-probability conditions:

- Computational Fluid Dynamics processing aggregate-typed elements
- Raytracing with adjacent shading points and coherent ray directions

Warp-Coherent Conditions in Real-World Applications



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**

Boolean-Step Conditions: Affine Analysis

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 $tid < Size\text{-}1\text{-}t$ is a boolean-step condition.

High-Probability Conditions: Static Branch Estimation

```
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);
    }
}
```

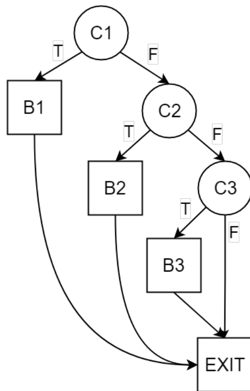
Figures gained by instrumented counters

- 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.

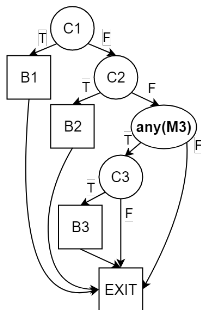
A Real-World Example: compute_flux

```
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);
    }
}
```

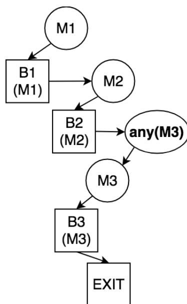


compute_flux: BOSCC - What previous approaches do

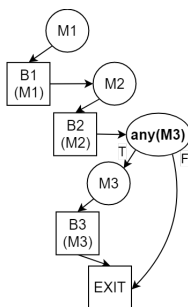
original:



linearization:



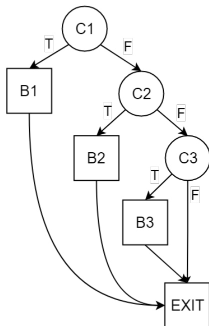
partial linearization:



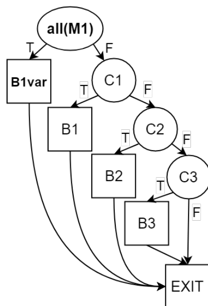
Partial linearization [Moll et al., 2018]

- ➔ **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!

Original CFG:



CFG after WCCV:



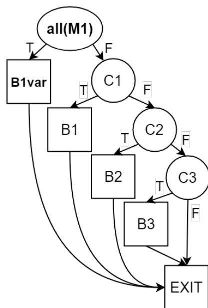
Our contribution: WCCV (Warp-Coherent Condition Vectorization)

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

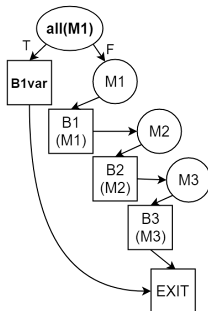
Advantage: **Skip more** and **eliminate masking**.

compute_flux: Partial Linearization after WCCV

original WCCV:



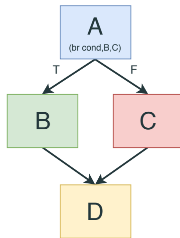
after partial linearization:



+ 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

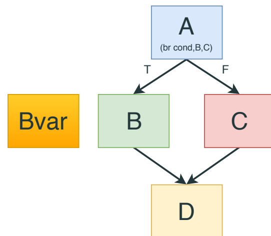
Implemented in the LLVM IR level.



Transformation - How WCCV is performed

Implemented in the LLVM IR level.

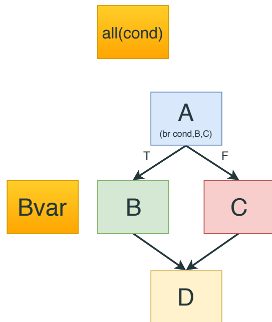
1. Clone the targeted branch



Transformation - How WCCV is performed

Implemented in the LLVM IR level.

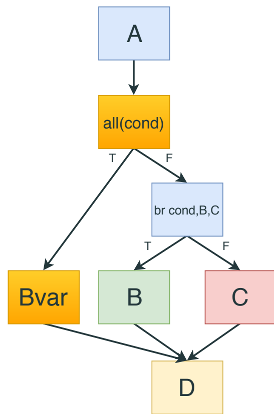
1. Clone the targeted branch
2. Generate the runtime **all** check



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



Operating Principle - How WCCV works during execution

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

Experimental Evaluation

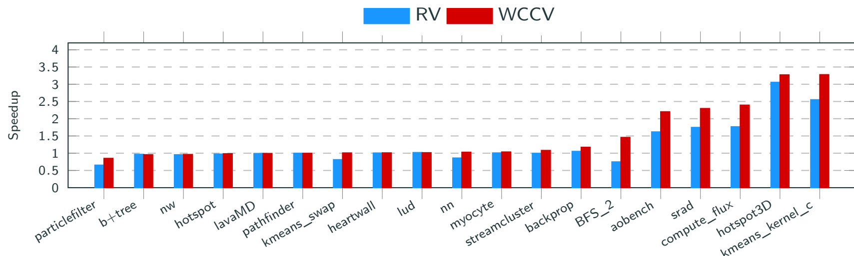
We implemented WCCV on top of Region Vectorizer (RV)

Our benchmarks:

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

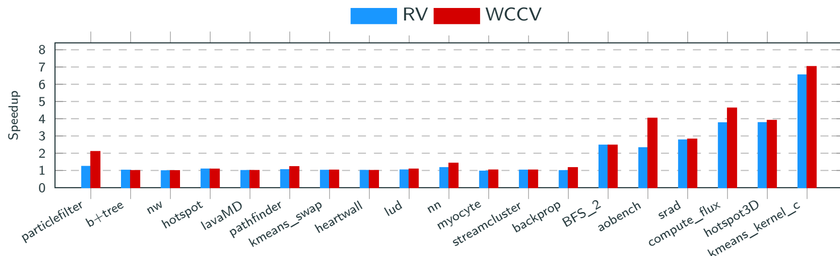
Source code available: https://github.com/HuihuiSun/WCCV

Runtime Speedups on Sandy Bridge



- ➔ 1.17× over RV
- ➔ 1.47× over scalar code

Runtime Speedups on Skylake



- ➔ 1.14× over RV
- ➔ 2.11× over scalar code

Impact of SIMD Vector Width on WCCV

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

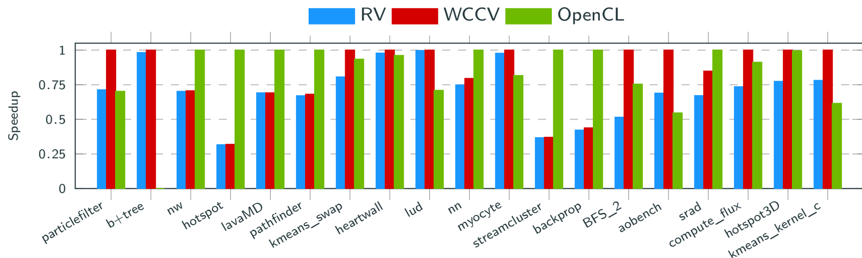
Kernel	vs. RV	vs. Scalar	masked RV	Masked WCCV
nn	1.19/1.22	1.03/1.42	42,764	4/12
kmeans_swap	1.22/1.08	1.00/1.11	494,020	4/4
particlefilter	1.29/1.70	0.85/2.09	40,000	0/0
srad	1.31/1.02	2.29/2.82	229,916	0/0
kmeans_kernel_c	1.28/1.07	3.28/7.02	494,020	4/4
aobench	1.36/1.74	2.20/4.03	4,179,522	10,342/23,762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

- 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!

Summary

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

Platform	SIMD extension	SIMD Width	Register number
Sandy Bridge	AVX	256-bit	16
Skylake	AVX512	512-bit	32

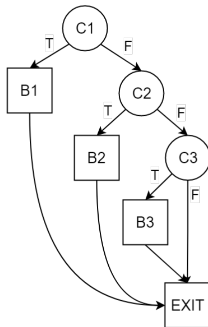
- Double the SIMD width and also double the register number

New in AVX512

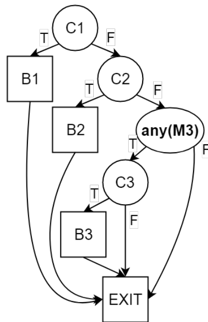
- 8 new opmask registers for masking most AVX-512 instructions used to control which values are written to the destinations
- New mini instruction extension operating on the opmask registers like KAND, which bitwise logical AND masks

compute_flux: BOSCC

Original CFG:



CFG after BOSCC:

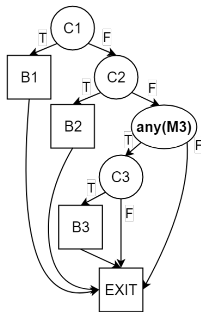


BOSCC(Bypass-On-Superword-Condition-Code) [Shin et al., 2005]

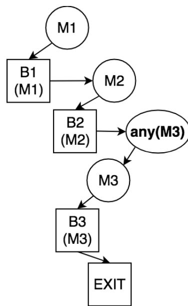
- ➔ inserts an **any** branch to skip the execution of a masked branch if its mask is all-false

compute_flux: Partial linearization after BOSCC

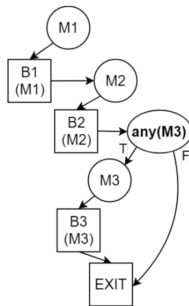
original:



linearization:



partial linearization:



Partial linearization[Moll et al., 2018]

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