Practice on Optimizing SPEC CPU 2017 for Sunway Architecture

Compiler Optimizations and Performance Analysis

January 31, 2026

Authors

Yingchi Long, Jun Jiang, Yanhe Zhai, Yaohui Han
Ying Liu, Zheng Lin, Yuyang Zhang, Zhongcheng Zhang
Jiahao Shan, Zhenchuan Chen, Xiaobing Feng, Huimin Cui

Affiliations

SKLP, ICT, UCAS
Wuxi Institute of Advanced Technology
Beijing & Wuxi, China

Agenda

Overview of today's presentation on SPEC CPU 2017 optimization for Sunway

Background: SPEC CPU & Sunway Architecture

Four Key Optimization Techniques (2 vector + 2 scalar)

Interleaved Case Studies

Results Summary

Lessons Learned & Q&A

Background: Sunway Architecture

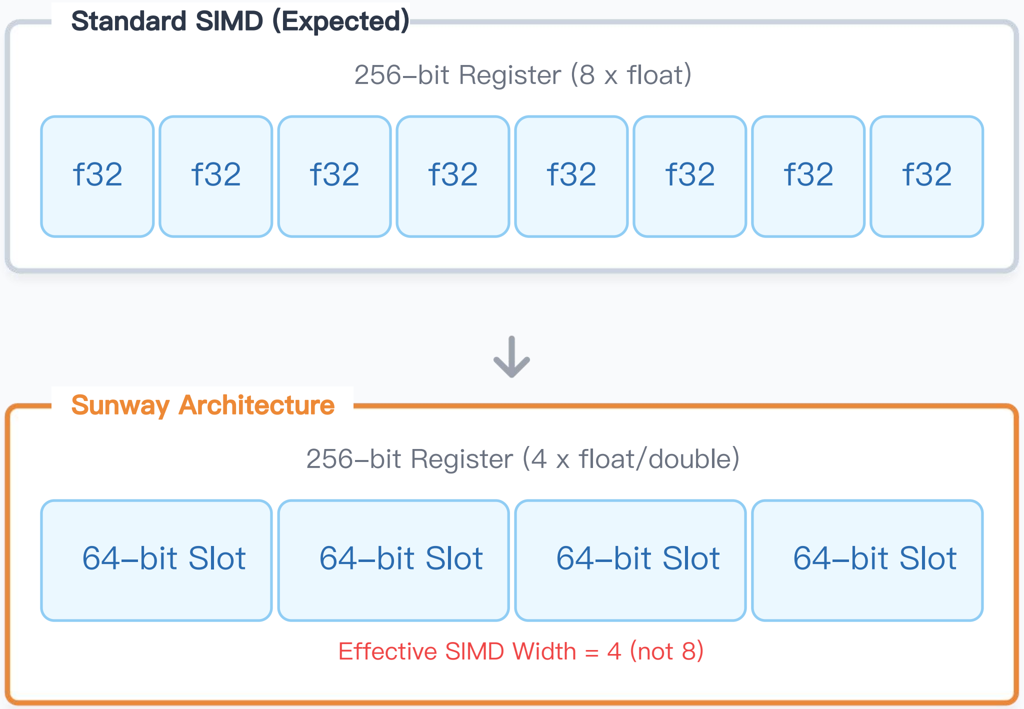
Processor Family

- Sunway SW3231
- WX-H8000 processors

Vector Instruction Set Status

- SIMD Limitation: ISD::{ADD, SUB} are only legal for v8i32. Sub-word Handling: i8/i16 require manual extload + truncstore.
- Memory vs. Register: FP32 occupies 64-bit in registers but 32-bit in memory.

Auto-vectorization on this arch basically does not exist before our work.



Implication

Requires explicit "Vector Factor" correction in LLVM to prevent over–vectorization errors.

Background: SPEC CPU 2017

Industry Standard: The definitive benchmark suite for measuring computation–intensive integer and floating–point performance.

Evaluation Mode: "Speed" mode to measure execution time, utilizing parallel processing capabilities.

Experimental Configurations

Base Config
–O2 Optimization Level

Peak Config
–O3 + LTO (Link Time Optimization)

Parallel Config
–O3 + LTO + OpenMP (64 Threads)

Technique 1: Vectorized ExtLoad & TruncStore

Solution to SIMD Type Conversion

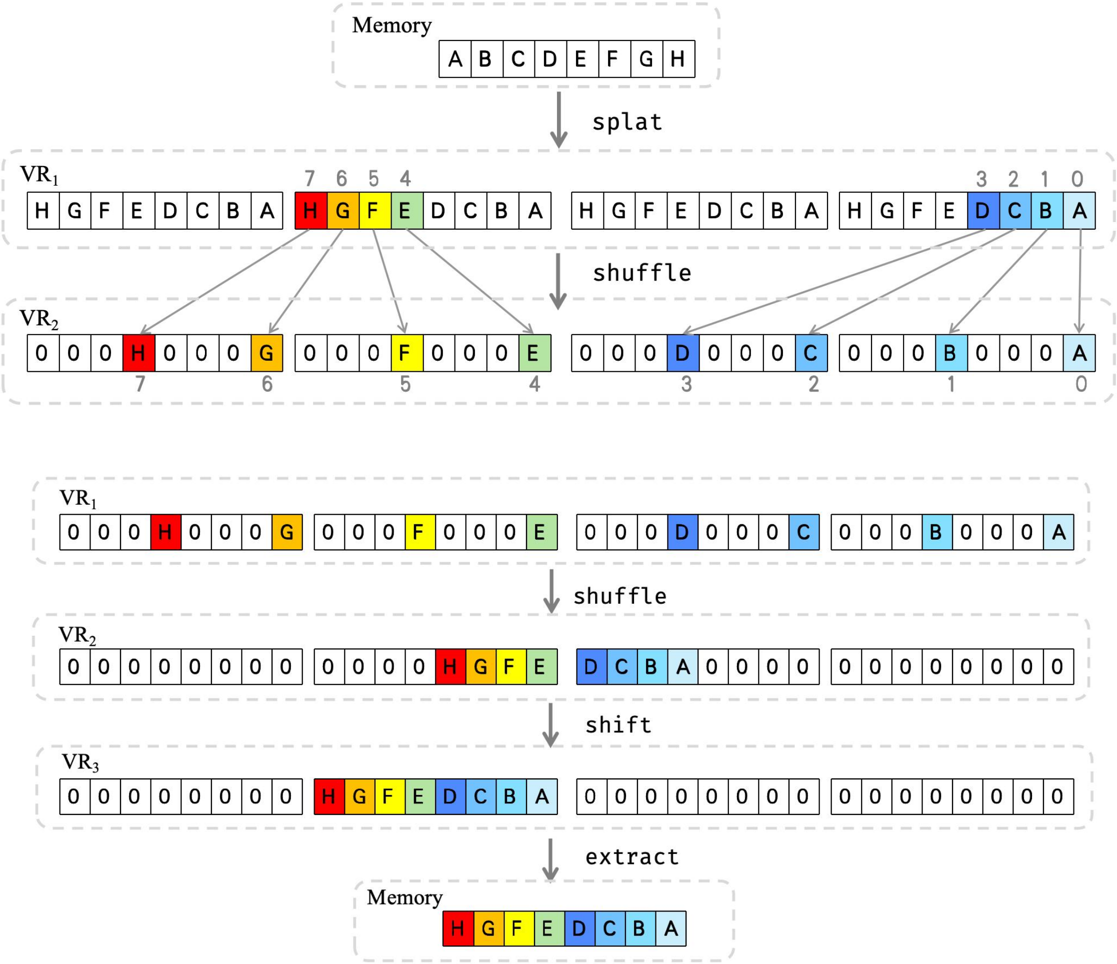
- **Goal: To avoid scalarization**
  - **which means:**
    load scalar + insertelement
- CUSTOMized Lowering for LOAD & STORE

For ISD::LOAD

- splat load ISD::LOAD + ISD::SPLAT_VECTOR
- shuffle (special) ISD::VECTOR_SHUFFLE

For ISD::STORE

- shuffle (special) ISD::VECTOR_SHUFFLE
- shift ISD::SRL
- extract ISD::EXTRACT_VECTOR_ELT



Case Study 1: 625.x264 & 638.imagick

The Bottleneck

Mismatch: Kernels operate on byte-oriented memory (i8) but compute with wider types (i32).

Consequence: LLVM default behavior scalarizes these loops, preventing SIMD vectorization entirely.

Applied Technique

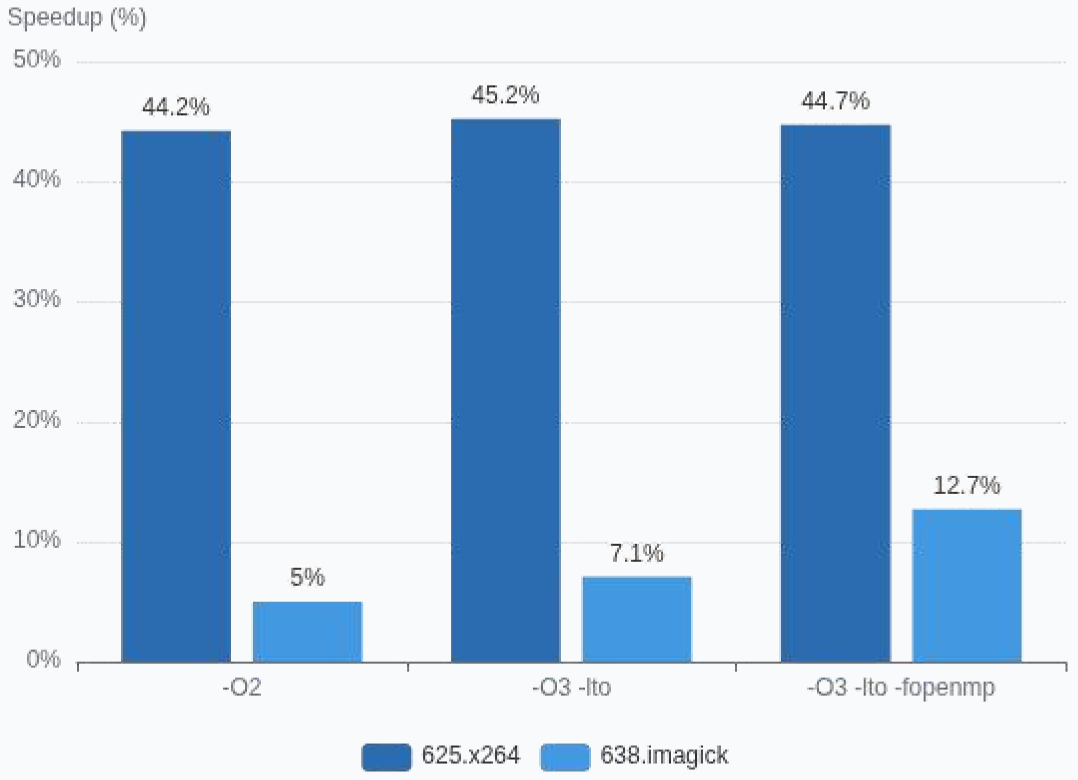
Technique 1: Customized Vectorized Load/Store

- Load: Splat + Extract (i8 → i32)
- Store: Shuffle + Shift + Extract (i32 → i8)

- Still profitable w/ 1/4 register width utilized!
- Must NOT use scalar load & insert/extract

Performance Results (WX-H8000)

Improvement over baseline



Technique 2: VF Calculation fixup

Observation

In Sunway architecture, vector registers allocate a full 64-bit slot for every floating-point element, whether it is single-precision (float) or double-precision.

The Issue

SLV infers Vectorization Factor (VF) from memory bit-width (DataLayout). For 32-bit floats, it assumes VF=8 (256/32), leading to "over-vectorization" incompatible with hardware.

The Fix

Extended TargetTransformInfo (TTI) with a new method getTypeWidthInReg. For Sunway, this returns 64 bits for all FP types.

VF Calculation Logic Comparison

Default LLVM Logic — Incorrect

- Check DataLayout (Memory Width)
- Float size = `32 bits`

VF = 256 / 32 = 8 (Too Wide)

↓

Sunway-Optimized Logic — Correct

- Call **getTypeWidthInReg()**
- Register Slot = `64 bits`

VF = 256 / 64 = 4 (Hardware Native)

This prevents generation of invalid vector code and fallback to scalar execution.

Technique 3: Loop-Carried Partial Redundancy Elimination

Problem: Invariant Pointers

In benchmarks like 602.gcc, pointer dereferences often remain invariant along specific execution paths across loop iterations. Standard optimization passes fail to catch these partial redundancies.

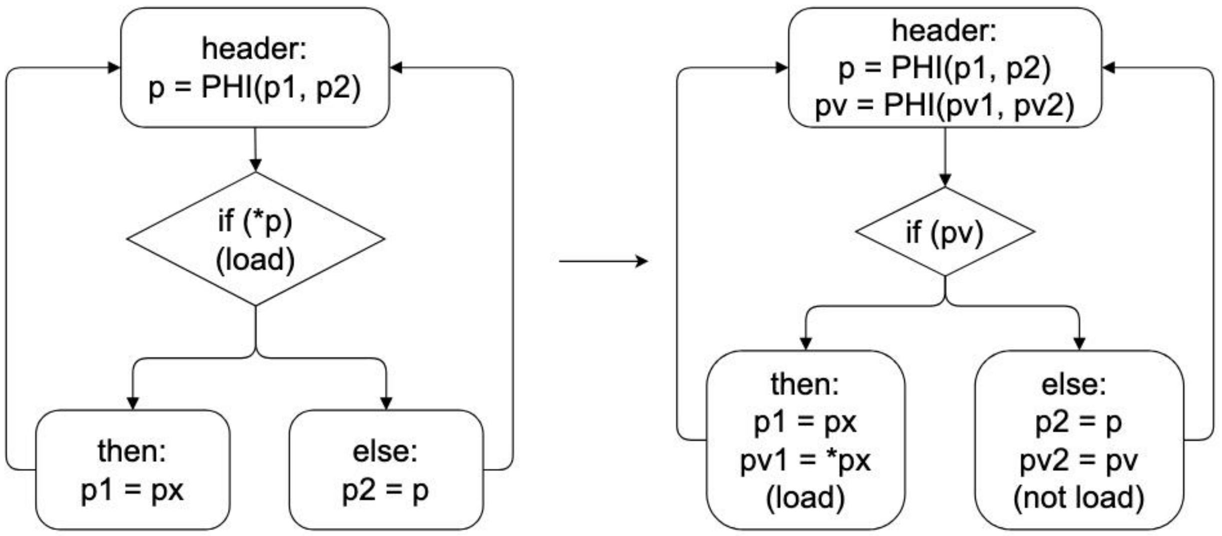
Transformation Strategy

We hoist the load instruction to the loop header using a PHI node. This effectively caches the value, removing redundant memory accesses on the "else" paths of subsequent iterations.

Analysis Methodology

- Canonical loop detection
- Dominance & Post-Dominance checks
- PHI node reconstruction

Optimization Logic



Technique 4: Fortran Argument Constant Propagation

The Challenge

Fortran passes arguments by reference. Standard LLVM SCCP is intraprocedural and cannot track constants across function boundaries, blocking optimization.

Our Solution

A custom Interprocedural Pass (inspire from GCC): 1. Identify constant arguments at call sites. 2. Clone the callee function. 3. Replace memory loads with constant values in the clone. 4. Iterate deeply along nested call chains.

Optimization Logic

STEP 1: DETECTION

Call Site main.f

```
call solve(N=50)
```

Passed by Ref (Ptr)

↓

New Pass

STEP 2: CLONING & PROPAGATING

Cloned Function solve_const_50

```
// Inside Clone
// load r1, *N -> Replaced
val = 50 (Constant)
```

↓

Synergy Effect

Combines with Inlining

Result: Loop bounds become constants

Case Study 2: 621.wrf & 603.bwaves

621.wrf Technique 2: VF Correction

Problem: LLVM incorrectly calculated Vectorization Factor (VF=8 for float) due to memory width.
Solution: Corrected VF to 4 based on register width (64-bit lanes).

603.bwaves Technique 4: Constant Propagation

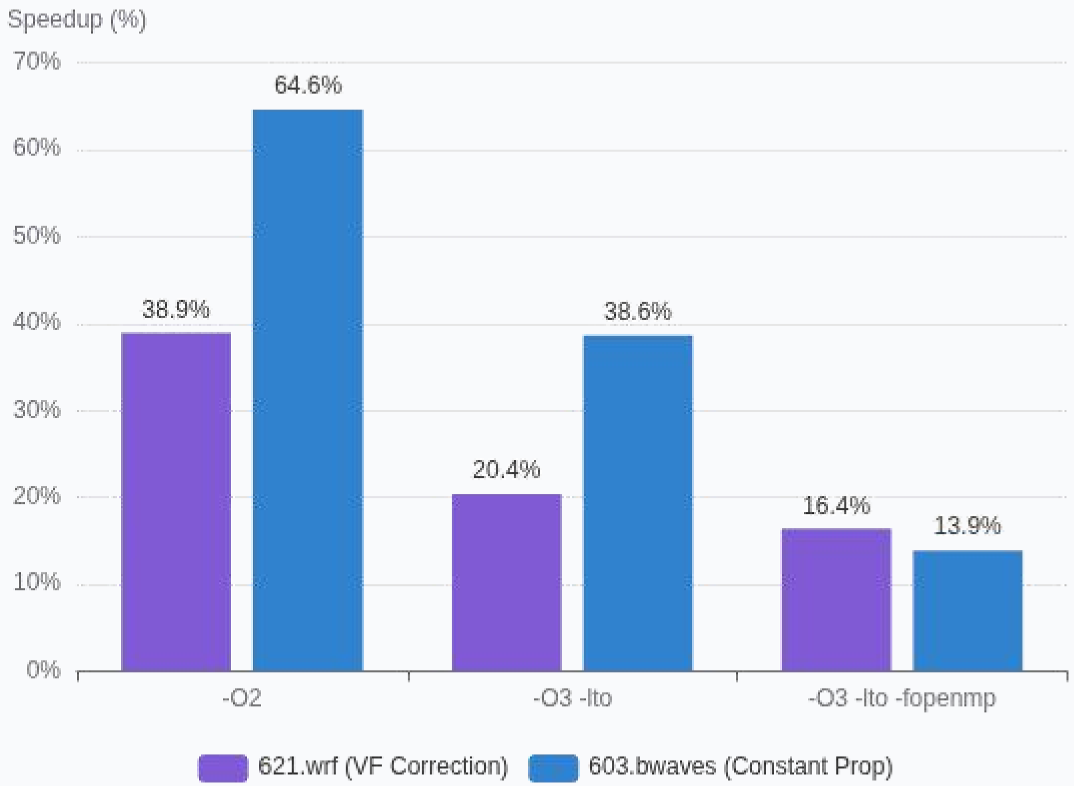
Problem: Fortran pass-by-reference blocked constant propagation across calls.

Solution: Interprocedural constant propagation via cloning.

Impact: Enabled deeper loop unrolling and better vectorization opportunities.

Performance Gains (WX-H8000)

Improvement over baseline across configurations



Lessons Learned

Architecture-Aware Vectorization is Essential

Standard LLVM logic often fails on specialized architectures. Custom instruction patterns (like our customized load/store) are critical for unlocking SIMD potential.

Accurate TTI Prevents Performance Regression

Providing the correct register width via TargetTransformInfo (TTI) is vital. It prevents the vectorizer from generating code that is "mathematically correct" but hardware-inefficient.

Redundancy Elimination Saves Memory Bandwidth

Loop-carried Partial Redundancy Elimination (LCPRE) effectively reduces memory traffic.

Interprocedural Analysis for Fortran

Since Fortran passes by reference, standard intra-procedural constant propagation is insufficient.

Questions?

Thank you for your attention

longyingchi24s@ict.ac.cn

SKLP, ICT, UCAS