# Loop Vectorization: how good is it?

Sjoerd Meijer

Madhur Amilkwanthar

Sebastian Pop

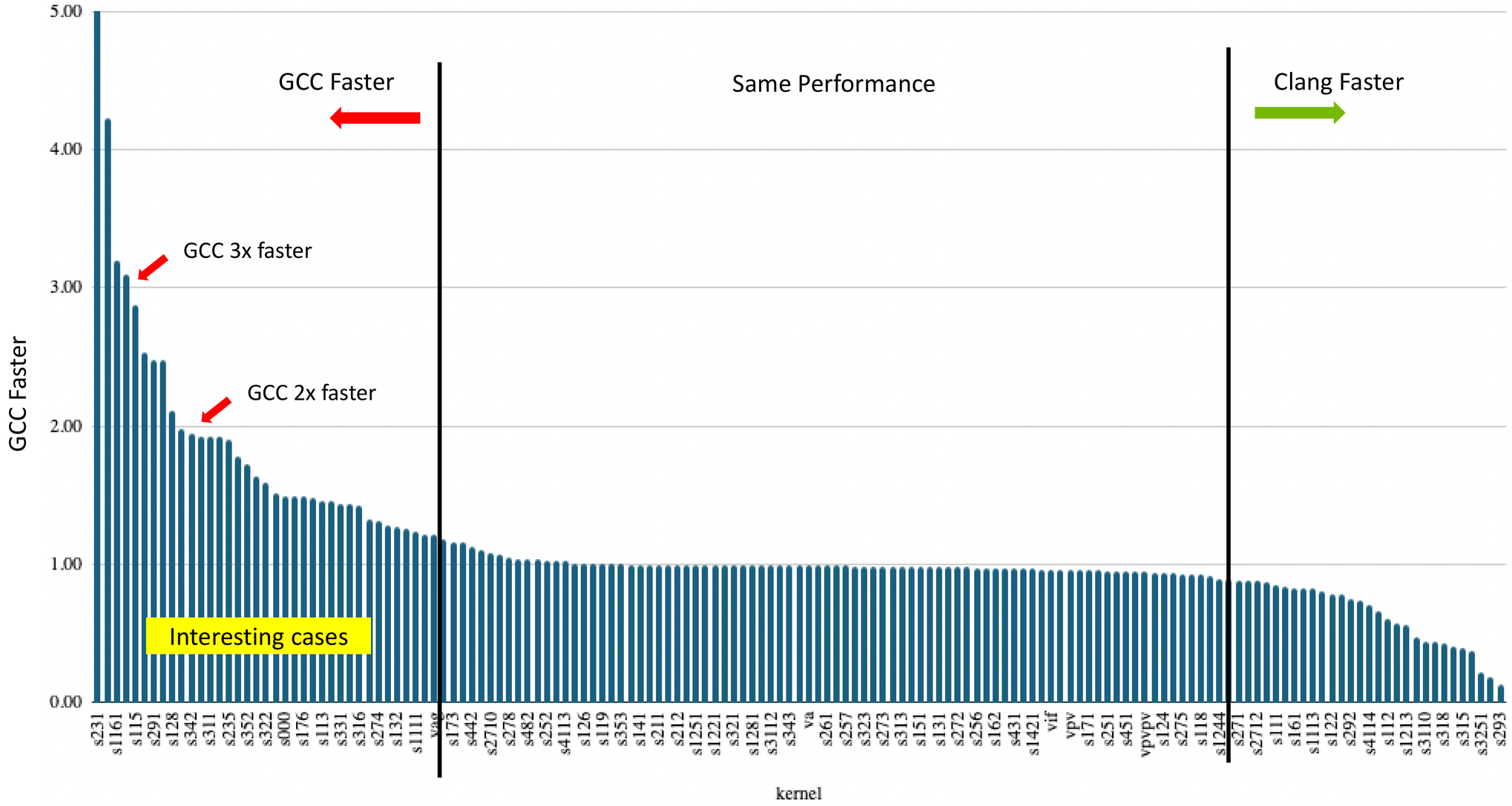
# Motivation

- Our target is an AArch64 CPU with a new SIMD extension: SVE2
  - Most findings will be target agnostic and generic

- **Q**: How well is the loop vectorizer performing?
  - Selected 2 benchmarks to quantify this: TSVC and RAJAPerf,
  - Benchmarked different compilers/options to find opportunities and regressions,
  - Identify root causes,
  - Create a plan for addressing these issues.

- In this presentation we will show results for TSVC-2:
  - **T**est **S**uite for **V**ectorizing **C**ompilers: part of the LLVM test-suite,
  - Consists of 152 small kernels that test different patterns that an auto-vectorizer could handle.

# Problem Statement: GCC13 is faster in a lot of cases



# Top Outliers TSVC-2: Clang vs. GCC13

| Kernel | Clang x-times worse | Reason |
|---|---|---|
| s231 | 16.5 | Loop interchange – issue #71521 |
| s2275 | 4.2 | Outerloop vectorization – issue #71520 |
| s1161 | 3.2 | If-convert / control-flow |
| s3111 | 3.1 | Interleaving |
| s31111 | 2.5 | SLP vectorization: FP reduction cost |
| s291 | 2.5 | Loop-peeling |
| s242 | 2.5 | Predictive commoning |
| s128 | 2.1 | Cost-model: scatter stores |
| s341 | 2.0 | Unnecessary SExt |
| s342 | 1.9 | Unnecessary SExt |
| s311 | 1.9 | Unrolling reductions cause dep chain |
| s235 | 1.9 | Outerloop vectorization |
| s352 | 1.7 | SLP more profitable |
| s322 | 1.6 | Different scalar codegen |
| s116 | 1.5 | Cost-model: vec slower scalar |
| s176 | 1.5 | Cost-model: VLA vs. VLS |
| s276 | 1.5 | If –convert / control-flow |

- Loop optimizations:
  - Interchange
  - Unroll-and-jam (outer-loop vec)
  - Loop peeling
- Loop vectorizer:
  - Interleaving
  - Epilogue vectorization
- Cost-model:
  - Scatter stores
  - Scalar vs. vector
  - Fixed vs. scalable
- Scalar optimization:
  - Predictive commoning
  - Sign extends

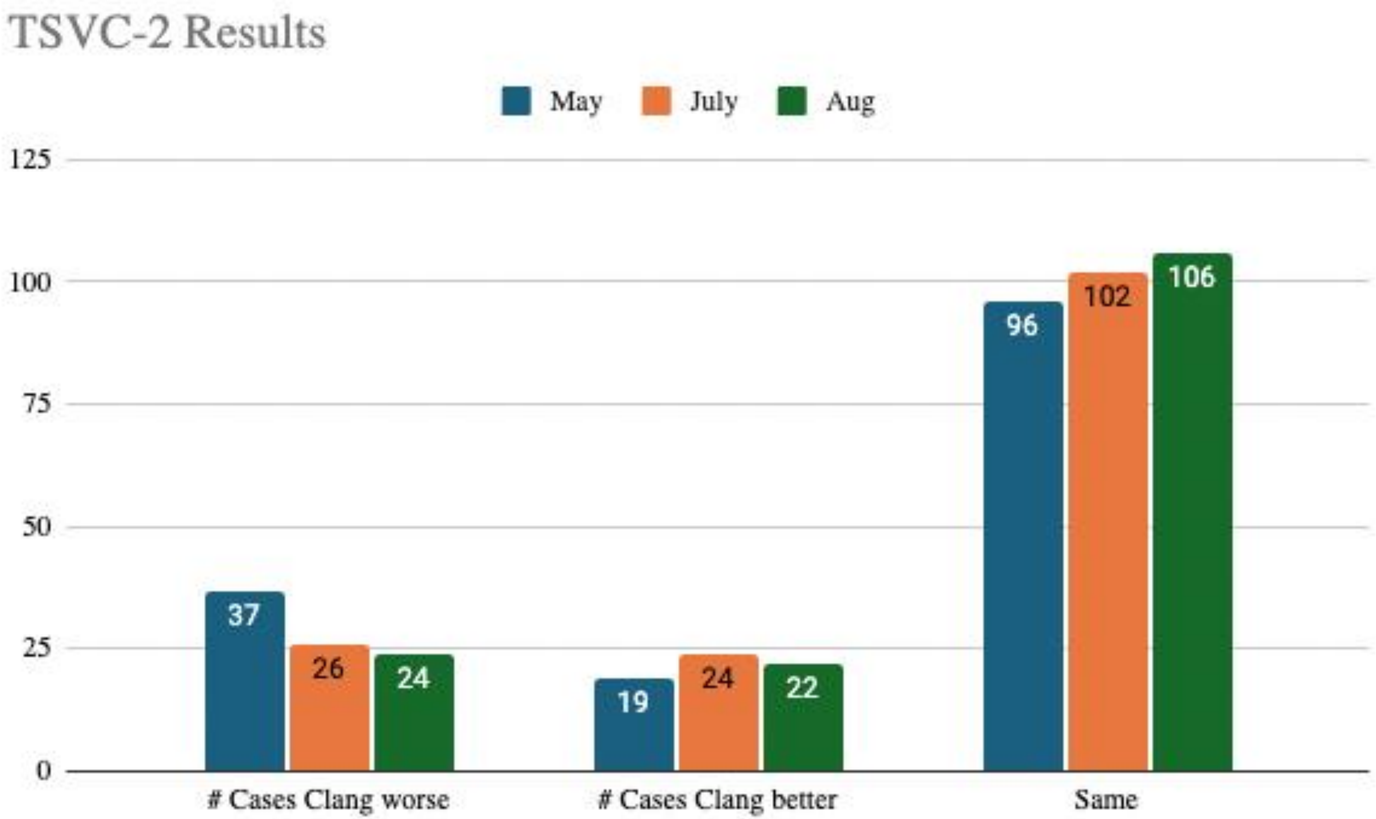
# What's the best we can do?

- Performed more manual experiments with some loop optimizations
- Loop distribute and interchange can make a big difference!
- We can vectorize more cases (in bold), and get better improvements:

| Kernel | Clang x-times worse | Reason |
|---|---|---|
| ***s2102*** | ***35.7*** | ***Loop distribute*** |
| s2275 | 22.3 | Loop distribute + interchange |
| s231 | 14.9 | Loop interchange |
| s235 | 13.8 | Outerloop vectorization |
| ***s2233*** | ***8.5*** | ***Loop distribute + interchange*** |
| ***s233*** | ***4.1*** | ***Loop distribute + interchange*** |

- No (fundamental) reason why the compiler wouldn't be able to do this.
- We leave a lot of performance on the table!

# Improvements Achieved So Far




Performance deficiencies fall in roughly two buckets:

1. Up to 2x improvements: codegen, heuristics, cost-model, etc.
   - First focused on this first, made 37 - 24 = 11 cases 2x faster.
2. Beyond 2x improvements: loop optimisations

# Up to 2x Improvements

- {Target}Subtarget.cpp parameter tuning:
  - **MaxInterleaveFactor**
    - Unrolling of vector body, default value is low (2),
    - Good default probably matches the number of SIMD pipes.
  - **ScatterOverhead**
    - There are default values for Gather/Scatter costs
    - Increased the value, for now.

- Loop-Vectorizer cost-model:
  - Introduced ***preferFixedOverScalableIfEqualCost()***
    - Cost is calculated for different VFs, but also scalable and fixed width
    - What if the cost model is a tie?
    - Adopted and implemented by RISC-V.
  - **Epilogue vectorization thresholds**
    - Loop body processes more elements,
    - If loop count is small, all time might be spent in the epilogue loop.
  - Loop-peeling:
    - Peel off the first iteration so that the loop becomes vectorizable

- And other cost-model changes to vectorize or scalarize cases.

# 1. The Cost-model Problem

- Works by recognizing and pattern matching IR snippets:
  - Mostly hand-written patterns,
  - It’s not expanded to a low-level IR,
  - It’s not using latency, throughput, resource descriptions in the back-end.

- Issues with this approach for new CPUs:
  - There’s always a missing case somewhere and is not scalable: endless tuning,
  - It is lacking precision because it works on IR.

- In an ideal world, we have a test-suite for the cost-model:
  - Set of test cases for which performance can be automatically evaluated,
    - Mapping of IR to current codegen
    - Extract codegen metrics with LLVM-MCA
    - Cost-tables are automatically updated.
  - We have RFCs for the other identified problems, unfortunately not for this one yet.

# Beyond 2x Improvements

- LLVM loop optimizations:
  - interchange,
  - unroll and jam,
  - distribute,
  - Fusion.

- Focus on:
  - Loop interchange:
    - Improves data access patterns,
    - Enables vectorization if dependences are in the way.
  - Unroll and jam:
    - Outer-loop vectorization: can be achieved by unroll-and-Jam followed by SLP vectorization.

# 2. The LLVM Loop Optimization Problem

All loop optimization passes are off by default in LLVM!

(unroll-and-jam, interchange, fusion, distribute)

Questions:

- **Q1**: Why are they all off? Can't we do loop optimizations in LLVM?
  - Type information about multi-dimensional arrays is missing in LLVM IR:
    - Delinearization is recovering this.
    - Getelementptr base + offset simplifications is making this more difficult,
    - However, it's robust enough to recognize a lot of cases.

- **Q2**: Is there no benefit in real world workloads?
  - That's something we can measure!

# Enable Loop-Interchange First

- Triggers a lot in the test-suite and in other benchmarks:

MultiSource/Applications/obsequi/tables.c:224
MultiSource/Benchmarks/mediabench/mpeg2/mpeg2dec/spatscal.c:298
MultiSource/Benchmarks/nbench/nbench1.c:283
MultiSource/Benchmarks/nbench/nbench1.c:1906
MultiSource/Benchmarks/nbench/nbench1.c:1907
SingleSource/Benchmarks/Misc/dt.c:15
SingleSource/Benchmarks/Polybench/datamining/covariance/covariance.c:81
SingleSource/Benchmarks/Polybench/datamining/correlation/correlation.c:103
SingleSource/Benchmarks/Polybench/linear-algebra/solvers/cholesky/cholesky.c:63
SingleSource/Benchmarks/Polybench/linear-algebra/solvers/cholesky/cholesky.c:64
SingleSource/Benchmarks/Polybench/linear-algebra/solvers/lu/lu.c
SingleSource/Benchmarks/Polybench/linear-algebra/solvers/ludcmp/ludcmp.c:62
SingleSource/Benchmarks/Linpack/linpack-pc.c:595
SingleSource/Benchmarks/Linpack/linpack-pc.c:1204

- Relatively simple and generic transformation,
- We would like to use loop-interchange to get a foot in the door
  - It's a user of DependenceAnalysis, so this is also about enabling DA.
  - Robust in testing but:
    - Fix the issues raised against interchange and DependenceAnalysis,
    - Collect compile-time numbers,
    - See our RFC here: https://discourse.llvm.org/t/enabling-loop-interchange/82589/

# 3. Evaluating Performance: The LLVM test-suite Problem

- “Why Most Published Research Findings are False”
  - The same is probably true for the LLVM test-suite and TSVC results

- TSVC problems, all kernels are placed in one executable:
  - Code layout issues: codegen changes in one function could greatly influence other functions.
  - Creates a lot of noise: e.g. +/-20% perf difference is probably not real.

- Other LLVM test-suite problems:
  - Most apps are very old,
  - A lot of apps have a very low runtime:
    - Can be very noisy,
    - And might not be compute bound

- Problematic for evaluating codegen changes and we need something better:
  - Update TSVC sources, then modify the sources and put each kernel into its own executable.
  - Add new benchmarks: e.g. RAJAPerf
  - Sanitize the current set: remove old ones, and increase the runtime of some apps.
  - See our RFC here: https://discourse.llvm.org/t/llvm-test-suite-improvements/82570

# Conclusions

- TSVC is an artificial benchmark, but has been very useful to identify a lot of different and generic opportunities,
  - Easy to compare different flags, optimization levels and compilers
  - Identified deficiencies in scalar optimizations, loop optimizations, heuristics, etc.

- The three problems:
  1. Disabled loop optimizations
     - Work on enabling loop interchange
  2. LLVM test-suite is unreliable for measuring vectorization changes
     - Would like to start working on improving TSVC, and investigate refreshing it.
  3. Cost-model tuning
     - A big task for which we don’t have any concrete plans yet.