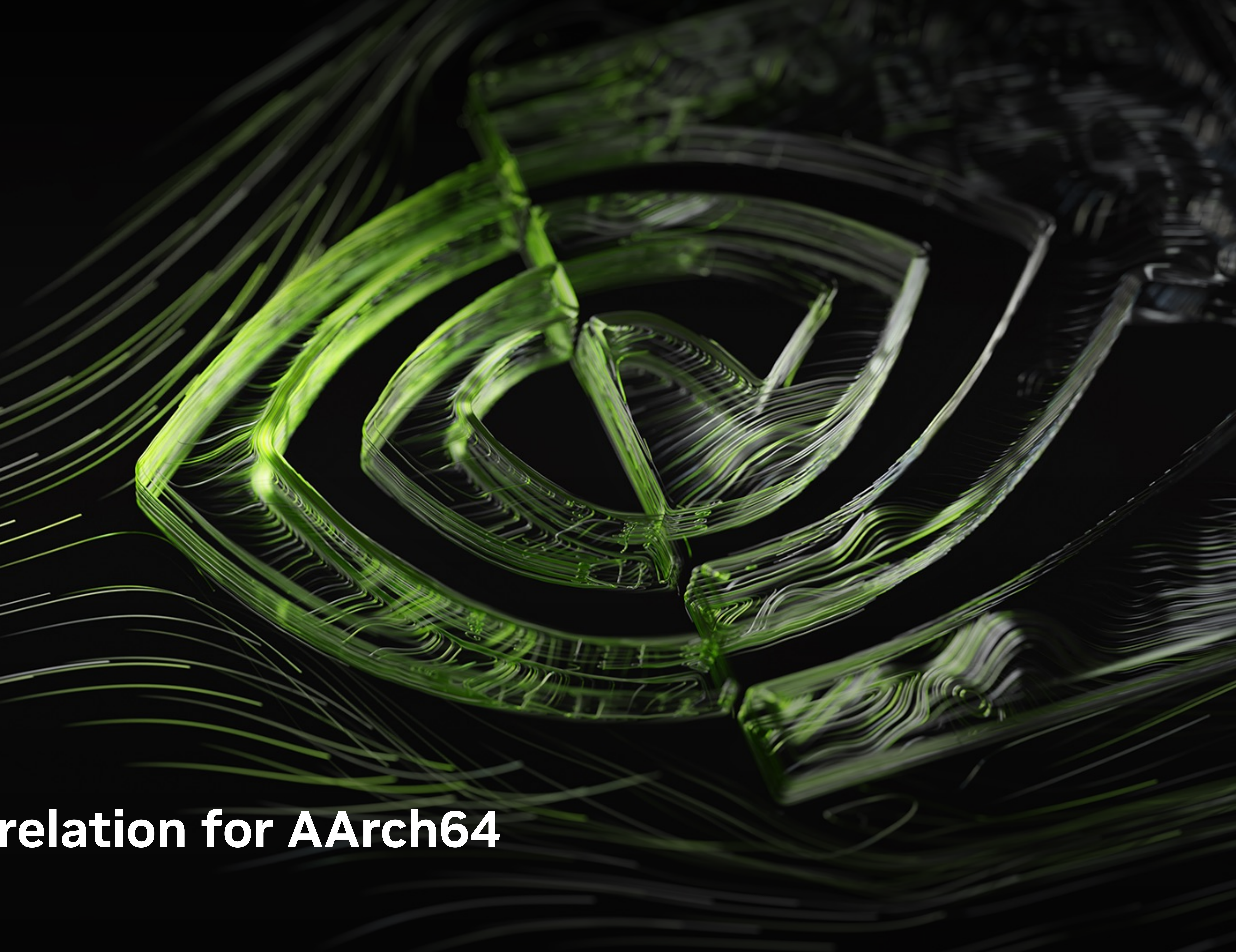


LLVM-MCA Correlation for AArch64

Ricardo Jesus & Sjoerd Meijer

A.k.a "Performance Analysis Journey for our new CPU"

Problem Statement

- Understanding this *anomaly*: small difference in straight-line asm code caused a 10% overall regression.

Slow Vector Code

```
ldr     h2, [sp, #166]
uzp1    v0.4s, v0.4s, v1.4s
ldur    q3, [sp, #200]
fadd    d11, d9, d11
add     x20, x20, #1
fadd    d12, d12, d8
mov     v0.s[1], v1.s[1]
ucvtf   s2, s2
mov     v0.s[3], v2.s[0]
fadd    v0.4s, v3.4s, v0.4s
stur    q0, [sp, #200]
```

Fast Scalar Code

```
ldp          s3, s4, [sp, #184]
fadd         d14, d9, d14
fadd         d10, d10, d8
add          x20, x20, #1
fadd         s2, s4, s2
fadd         s0, s3, s0
stp          s0, s2, [sp, #184]
ldp          s0, s2, [sp, #192]
fadd         s0, s0, s1
ldr          h1, [sp, #150]
ucvtf        s1, s1
fadd         s1, s2, s1
stp          s0, s1, [sp, #192]
```

- Observations/expectations:
  - Number of instructions is about the same,
  - Expect similar performance, maybe slightly worse, but not 10% worse
  - Can't tell anything more about this…
- Don't know why should we **not** vectorise this, or how to vectorise this differently.
- Missing a tool for compiler engineers and our new CPU to evaluate/implement different code-generation strategies

Outline

Performance analysis journey:

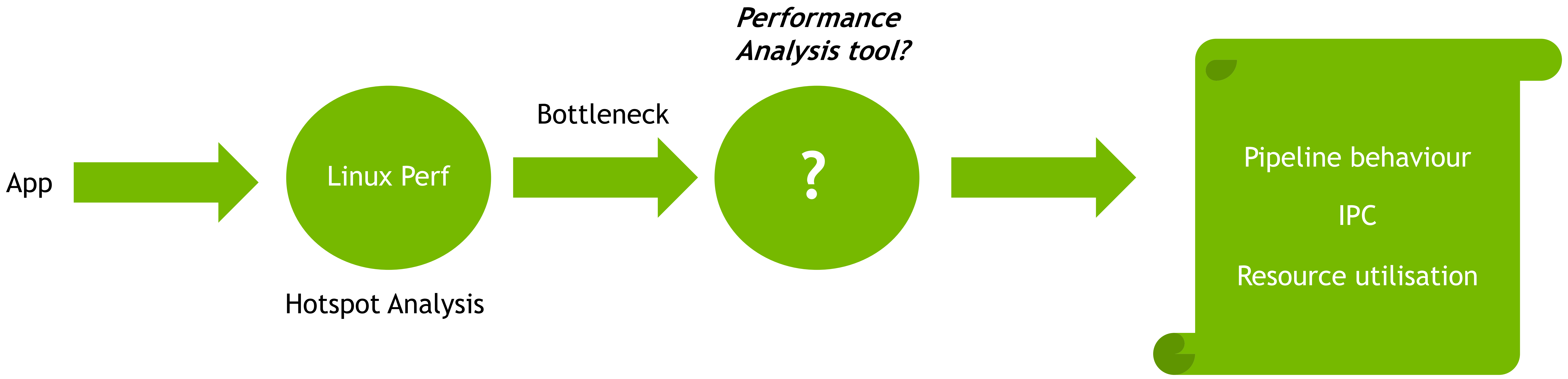
- **Part 1**: Looking for an open-source performance analysis tool for *compiler-engineers* to:
  - Understand, evaluate and choose different code-generation strategies.
- **Part 2**: Investigate the quality of this tool
  - Correlate predictions with results hardware results: how well do they match up?

- For the new NVIDIA Grace CPU Superchip:
  - High-performance CPU for HPC, data-centres and cloud computing
  - Up to 144 Arm Neoverse V2 CPU cores
  - Our findings are **not** specific to Grace: all (verified) generic AArch64 observations.

- Solution and contributions to enable this:
  - Step1: *Performance analysis tool*: enable LLVM-MCA for Grace
  - Step 2.1: *Correlation*: automatically extract hot code parts from workloads.
  - Step 2.2: Verify how good static performance predictions are with hardware results (correlation).
  - Step 2.3: If results don't match up, fix any issues, goto step 3.

Chapter 1: Performance Analysis Tools / Flow

- Create a micro-benchmark: time-consuming, error prone, may not give the insights we need!
- Another approach: cycle accurate simulation.
  - Most accurate and there's no substitute,
  - but slow, more difficult to use and not always available.
- We would like a performance analysis tool with different trade-offs:
  - Faster than cycle-accurate simulation and capturing the performance trend well.



Timeline view of instructions

Slow Vector Code

```
Timeline view:
                    0123456789
Index     0123456789

[0,0]     DeeeeeeER .    .   .   ldr    h2, [sp, #166]
[0,1]     DeeE----R .    .   .   uzp1   v0.4s, v0.4s, v1.4s
[0,2]     DeeeeeeER .    .   .   ldur   q3, [sp, #200]
[0,3]     DeeE----R .    .   .   fadd   d11, d9, d11
[0,4]     DeE-----R .    .   .   add    x20, x20, #1
[0,5]     D=eeE---R .    .   .   fadd   d12, d12, d8
[0,6]     D==eeeeeER.    .   .   mov    v0.s[1], v1.s[1]
[0,7]     D======eeER    .   .   ucvtf  s2, s2
[0,8]     .D=======eeeeeER   .   mov    v0.s[3], v2.s[0]
[0,9]     .D============eeER .   fadd   v0.4s, v3.4s, v0.4s
[0,10]    .D==============eeER   stur   q0, [sp, #200]
```

Fast Scalar Code

```
Timeline view:
                    012345
Index     0123456789

[0,0]     DeeeeeeER .    .   ldp        s3, s4, [sp, #184]
[0,1]     DeeE----R .    .   fadd       d14, d9, d14
[0,2]     DeeE----R .    .   fadd       d10, d10, d8
[0,3]     DeE-----R .    .   add        x20, x20, #1
[0,4]     D====eeER .    .   fadd       s2, s4, s2
[0,5]     D======eeER    .   fadd       s0, s3, s0
[0,6]     D========eeER  .   stp        s0, s2, [sp, #184]
[0,7]     .DeeeeeeE---R  .   ldp        s0, s2, [sp, #192]
[0,8]     .D======eeE-R  .   fadd       s0, s0, s1
[0,9]     .DeeeeeeE---R  .   ldr        h1, [sp, #150]
[0,10]    .D======eeE-R  .   ucvtf      s1, s1
[0,11]    .D========eeER .   fadd       s1, s2, s1
[0,12]    .D==========eeER   stp        s0, s1, [sp, #192]
```

- Now we can see that dependency-chains increase the critical path.
- Understanding the behaviour of instruction sequences is
  - fundamental to evaluate different code-generation strategies, and
  - ultimately select and implement the best one.

LLVM-MCA

Static low-level performance analysis tool

- llvm-mca is a static performance analysis tool ("***M***achine ***C***ode ***A***nalyser")
  - Part of the llvm-project and reuses different components,
  - Relies on Scheduling models, which are used for:
    1. instruction scheduling and code-generation (LLVM, compiler)
    2. performance analysis (LLVM-MCA)

- Given a sequence of assembly instructions:
  - Provides instruction information such as latency and reciprocal throughput
  - Estimates performance metrics such as IPC, µOps Per Cycle and Block RThroughput
  - Identifies hardware resources consumption and pressure
  - Trace execution reports with instructions' state transitions

- **Step 1**: Enable LLVM-MCA for the Grace CPU:
  1. Contributed a scheduling model for the Neoverse V2 core in D151894
  2. Checked that the model didn't lead to performance regressions
     - Neoverse V2 core used the Neoverse N2 model for instruction scheduling/analysis

- Result of playing and looking at LLVM-MCA reports and timelines:
  - LLVM instruction cost-model patches, e.g.: FADD/FSUB (D146033), LD1R (D141602), and MOV/INS (D144508)

Chapter 2: Correlation

- Are LLVM-MCA's estimates accurate? Can we trust the predictions?
  - Static performance predictions should show the same trends as hardware.

- LLVM-MCA has limitations, e.g.:
  - By design, it doesn’t predict the frontend throughput, and
  - Doesn't correctly model instructions that affect control flow.
  - Assumptions made by the processor model used by the tool.
  - Quality of the scheduling model affects the performance analysis.
  - Scheduling models do not describe all processors' details.

- Open questions:
  - Does this matter?
  - How do we define “accurate” for these performance predictions?

- We need to define some criteria and our correlation methodology.

Correlation Methodology

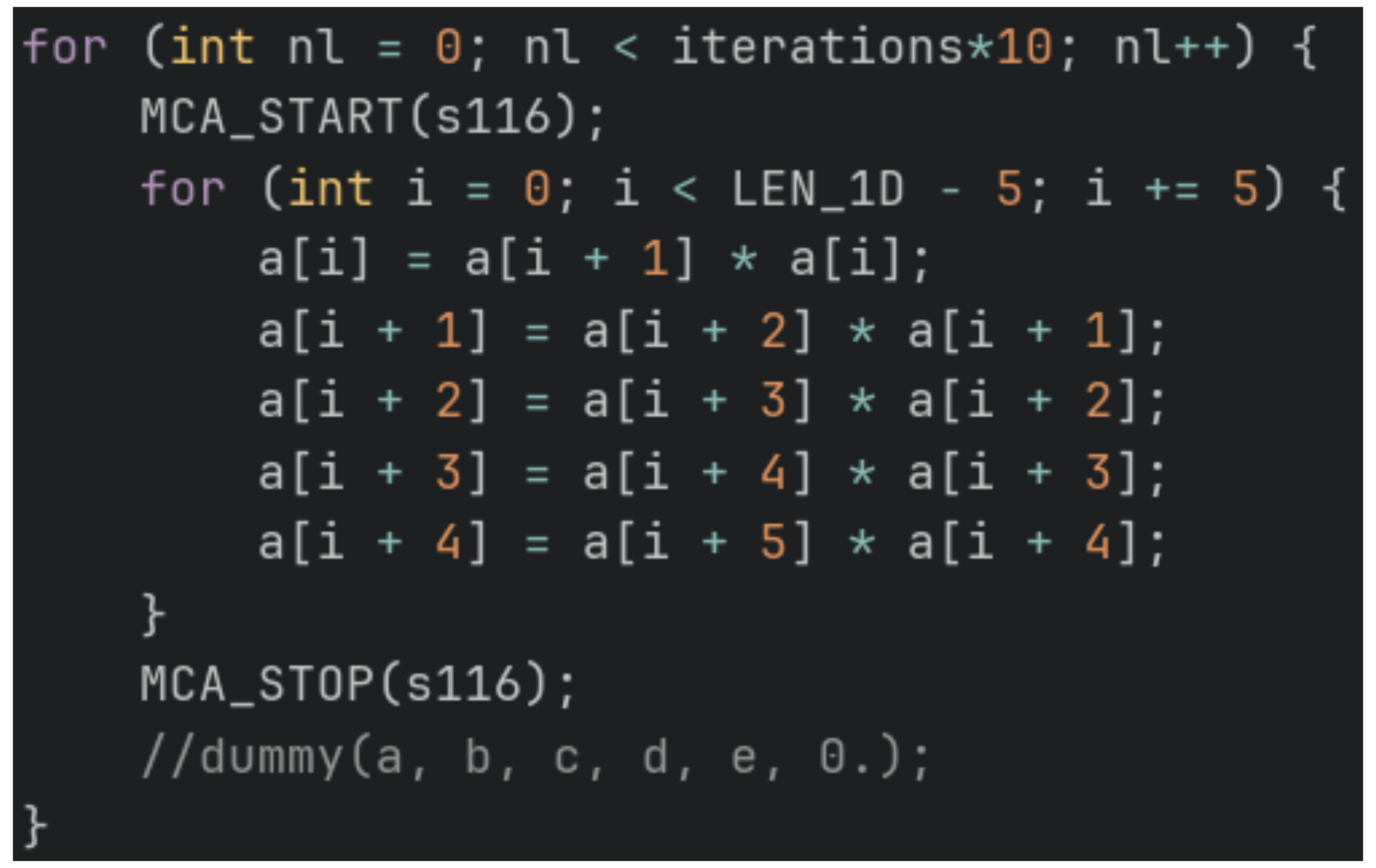
- For all apps in a set of interesting workloads, the predicted performance is e.g. within 5% of hardware:
  - $0.95 \leq \frac{predicted(app_i)}{runtime(app_i)} \leq 1.05$
  - Do this for a relatively large number of apps,
  - That should give confidence in the estimated performance numbers.
  - Is preferably automated.

- But LLVM_MCA is not the tool that calculates "$predicted(app_i)$"
  - It consumes assembly code and analyses straight line code.

- Our Approach: compare two equivalent assembly codes, generated from the same source-code

- Select ***one*** C/C++ kernel (inner-loop) from an app and generate:
  - ***Two*** assembly kernels ***A*** and ***B***, where $\boldsymbol{A} \approx \boldsymbol{B}$.
  - Equivalent = variants **A** and **B** process the same number of data elements.
  - Example when they are not equivalent:
    - If one variant has an unrolled loop.
    - If one variant has been loop-vectorized.
  - Generate these variants by compiling the source with different compiler options (allows automation).

Step 2.1: Extract Kernels for TSVC-2 (LLVM test-suite)

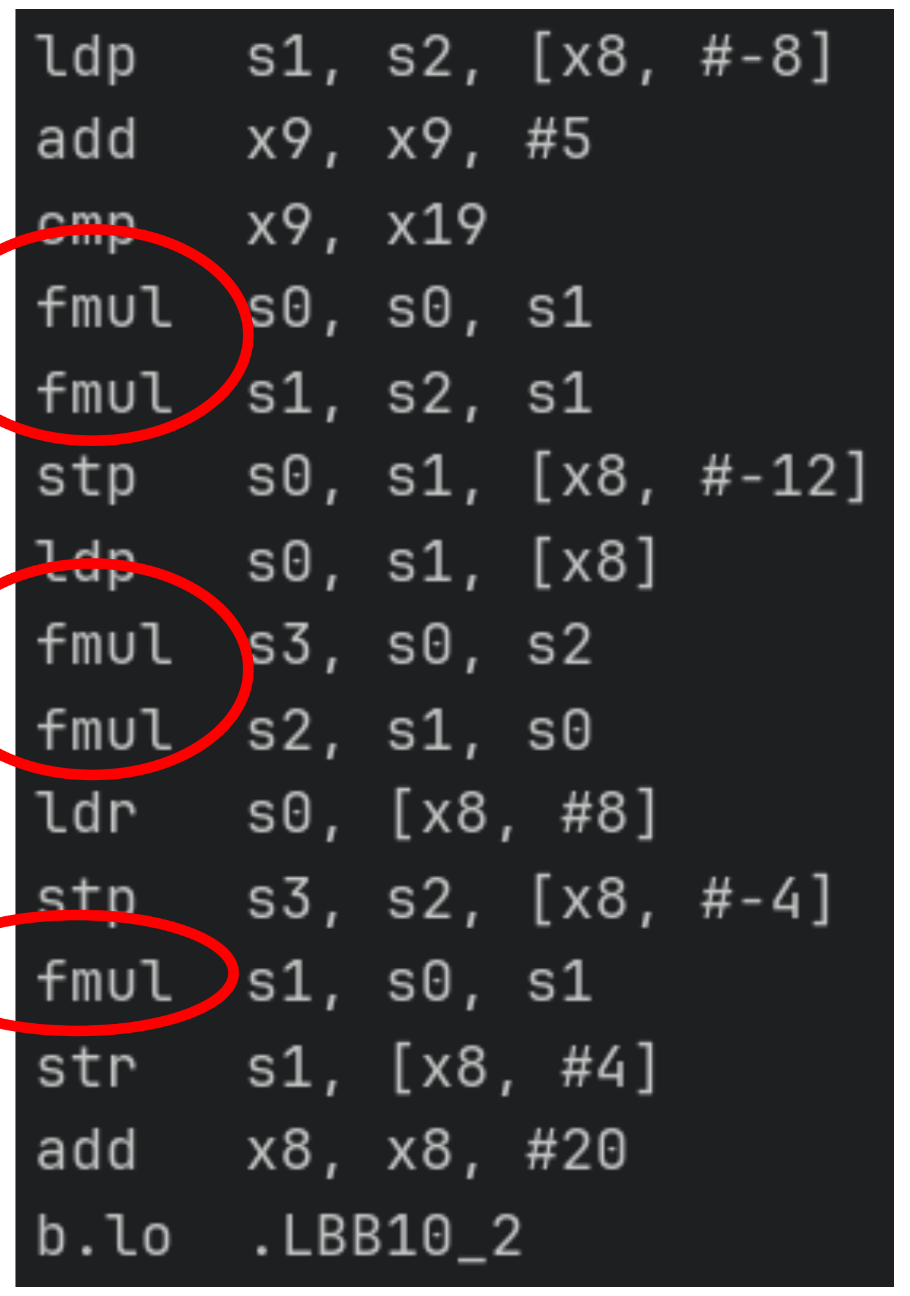


```c
for (int nl = 0; nl < iterations*10; nl++) {
    MCA_START(s116);
    for (int i = 0; i < LEN_1D - 5; i += 5) {
        a[i] = a[i + 1] * a[i];
        a[i + 1] = a[i + 2] * a[i + 1];
        a[i + 2] = a[i + 3] * a[i + 2];
        a[i + 3] = a[i + 4] * a[i + 3];
        a[i + 4] = a[i + 5] * a[i + 4];
    }
    MCA_STOP(s116);
    //dummy(a, b, c, d, e, 0.);
}
```

Disable Loop Vectoriser
Disable SLP Vectoriser

Disable Loop Vectoriser
Enable SLP Vectoriser

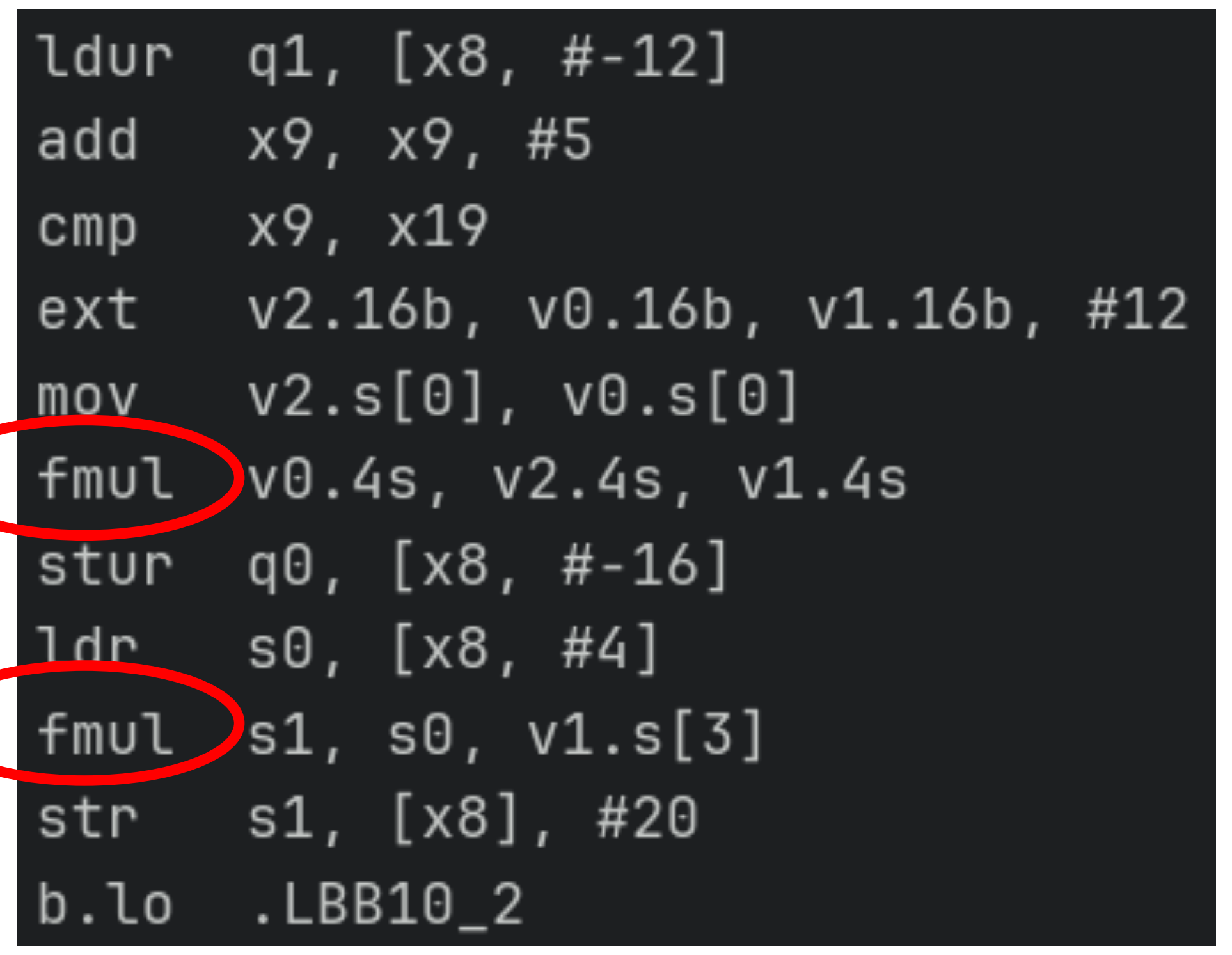
A



```
ldp   s1, s2, [x8, #-8]
add   x9, x9, #5
cmp   x9, x19
fmul  s0, s0, s1
fmul  s1, s2, s1
stp   s0, s1, [x8, #-12]
ldp   s0, s1, [x8]
fmul  s3, s0, s2
fmul  s2, s1, s0
ldr   s0, [x8, #8]
stp   s3, s2, [x8, #-4]
fmul  s1, s0, s1
str   s1, [x8, #4]
add   x8, x8, #20
b.lo  .LBB10_2
```

5 x 1 fmul

B



```
ldur  q1, [x8, #-12]
add   x9, x9, #5
cmp   x9, x19
ext   v2.16b, v0.16b, v1.16b, #12
mov   v2.s[0], v0.s[0]
fmul  v0.4s, v2.4s, v1.4s
stur  q0, [x8, #-16]
ldr   s0, [x8, #4]
fmul  s1, s0, v1.s[3]
str   s1, [x8], #20
b.lo  .LBB10_2
```

1 x 4 + 1 fmul

1. Annotate C kernels with MCA START / STOP markers
   - Markers influence codegen, so place them around inner-loops
2. Compile it to generate comparable ***A*** and ***B*** assembly version:
   - Disable Loop vectorizer: `-fno-vectorize`
   - Toggle SLP vectorizer on/off: `-fno-slp-vectorize`
3. Extract the kernels:
   - Find the MCA START/STOP markers in assembly,
   - Recognise the loop and extract the inner-loop body, and
   - Run LLVM-MCA on ***A*** and ***B***.
4. Correlation expectation is:

$$predicted(kernel\_A) < predicted(kernel\_B)$$

$$\Rightarrow$$

$$runtime(app\_A) < runtime(app\_B)$$

Step 2.2: Compare Predictions with Hardware Runs

Predicted(kernel_A) < Predicted(kernel_B)
=> runtime(app_A) < runtime(app_B)

speedup (%)
400
350
300
250
200
150
100
50
0
-50
-100
loop
Real (Grace) +
Estimated (MCA) ◻
(1)Correct predictions
(2) Incorrect predictions
(3)
(4)
(5)

Observations:

1. Correct predications: MCA correctly predicts speedups (within ~10% of measured values) for s351 & s352
2. Incorrect predictions: hardware shows ~10–20% speedups with SLP, MCA predicts ~75–85% slowdowns for s116 & s353
3. No MCA versions: limitations of scripts recognizing the markers and loops
4. Compiler flags didn't result in different codegen
5. Potentially interesting points

Step 2.3: Fix identified Issues

SchedModel:

1. Fixed the N2 model for some ALU instructions in D145370
   - Trivial fix, but high impact.
2. Fixed instruction forwarding descriptions (part of neoverse v2 schedmodel commit).
3. Fix Zero-latency moves that were not modelled correctly in D159433.
4. Fixed (partially) post/pre-index loads/stores in D159254

None had any (measured) impact on code-generation, so only beneficial for the LLVM-MCA perf analysis use-case.

LLVM-MCA:

1. Instruction fusion
   - Some instruction pairs can be accelerated when they are adjacent in program order (e.g. cmp + b.cond)
   - Important for SVE and other instructions (e.g. MVPRFX)
2. Accurate throughputs
   - Due to the way throughputs are computed in MCA it's not always possible to model exact values
   - Not crucial as it affects very few instructions, and difference between computed and real throughput is minimal

LLVM-MCA Findings, Cont’d

Post increment stores: two operations:

1. Store to the address,
2. Increment of the address

```
[0,0]    DeeeER    .    umulh  x2, x1, x1
[0,1]    D===eER   .    str    x2, [x1], #0
[1,0]    D====eeeER.    umulh  x2, x1, x1
[1,1]    D=======eER    str    x2, [x1], #0
```

```
[0,0]    DeeeER .    umulh  x2, x1, x1
[0,1]    D===eER.    str    x2, [x1]
[0,2]    DeE---R.    add    x1, x1, #0
[1,0]    D=eeeER.    umulh  x2, x1, x1
[1,1]    D====eER    str    x2, [x1]
[1,2]    D=eE---R    add    x1, x1, #0
```

- *STR* doesn’t start until **all** inputs are available
- Second *MUL* could start earlier, doesn’t need to wait for the *STR* to finish.
- Not accurate for OOO architectures with uOps.
- Not a tablegen fix, so is an open question, see our discourse discussion.

Observations, Lessons Learned & Recommendations

1. Found several issues in the cost-model and in the different scheduling models.
   - Recommendation**:** correlation seems to be a good exercise to find schedmodel problems.

2. Correlation is time-consuming and surprisingly (?) not very straightforward (lesson learned).
   - Practical issues that make doing a large sweep challenging.
   - Toggled the vectorizer on/off to generate different code, should try different ways, e.g. –O2 vs. –O3
   - Or a completely different way?

3. Have not collected enough data to conclude whether LLVM-MCA is well correlated,
   - But we do know now that predictions can be off for some cases,
   - And have found cases where it predicts the trend well.

4. LLVM-MCA is an excellent tool
   - Helped us in understanding and solving performance issues.
   - This correlation study and resulting experience that we got really helps reading/judging the predictions.