Co-design exploration with vectorized CNNs

Sonia Rani Gupta, Miquel Pericàs

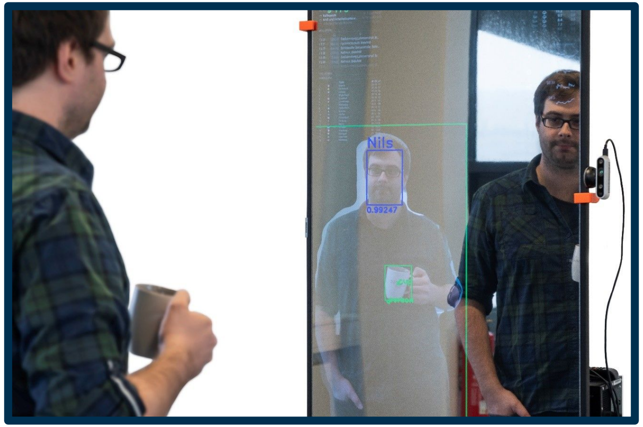
Chalmers University of Technology

AHUG ISC 22 Workshop

CNNs and vector processors

YOLOv3 - Real-time object detection

- several EU projects: LEGaTO, eProcessor, EUPiLOT



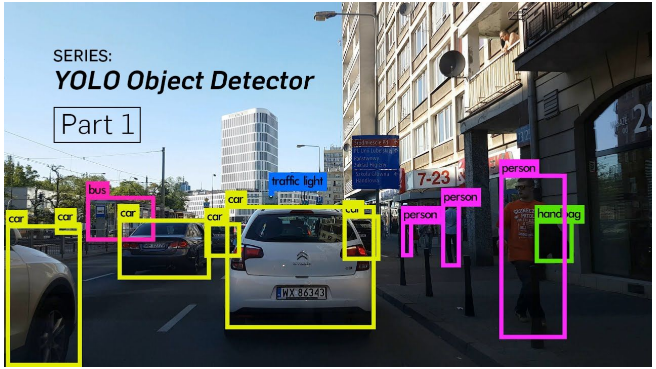
Smart mirror
UniBielefeld

Real-time and accurate CNNs:

➔ Compute intensive

➔ Require high optimizations

➔ Requires high performance computers with high energy efficiency



https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/

The resurgence of vector processors

Vector ISAs can provide high performance and energy efficiency

SIMD architectures:

- ARM-Neon, AVX-512
- No Vector Length Agnostic - ISA needs to change with VLs

Long Vector architectures

- ARM-SVE - up to 2048-bit
- RISC-V Vector - up to 16384-bit
- Very long vector length
- Vector Length Agnostic (VLA) programming: same code can exploit different vector lengths, VL becomes a HW implementation parameter

Can we achieve real-time object detection on vector processors based on ISAs like ARM-SVE or RISC-VV?

Outline

1. Introduction
2. CNNs, YOLOv3 and Darknet
3. Methodology
4. im2col + gemm
    a. Optimized kernels
    b. Hardware explorations: caches, vector lengths, vector lanes
    c. Parallelization
5. Winograd
    a. Optimized kernels
    b. Performance
6. Summary and Conclusions

Codesign of Vectors for CNN

Efficiency of implementation = HW design + Algorithmic Optimizations

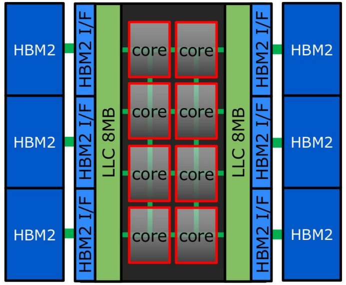
Algorithmic Optimizations:

- Highly optimized kernels on VLA vector architectures

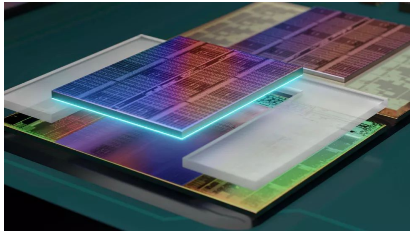
HW vector design:

- Impact of microarchitectural parameters on optimized kernels.
  - **Vector Length**, ie what is the impact of longer vector sizes?
  - **Cache size**, ie what is the impact of large stacked SRAM/DRAM?
  - **Vector lanes**, how much on-chip vector parallelism?

Goal: Design Space Exploration of vector architectures by studying combined implications of algorithmic optimizations and HW parameters tuning with CNN.



SX-Aurora



AMD 3D V-Cache (Image credit: AMD)

Structure of CNNs

Different types of layers repeating and interleaving in different ways.

Object detection YOLOv3 layers:

- **Convolutional layer** - Most time consuming. Applies filter on input tensor and create feature maps.
- **Shortcut(Res-unit)**: Add feature maps from layers.
- **Up-sample**: Scale feature maps by factor of stride
- **Concatenation**: Concatenates output of the layers
- **Yolo**: Detection Layer. There are 3 detection layers.

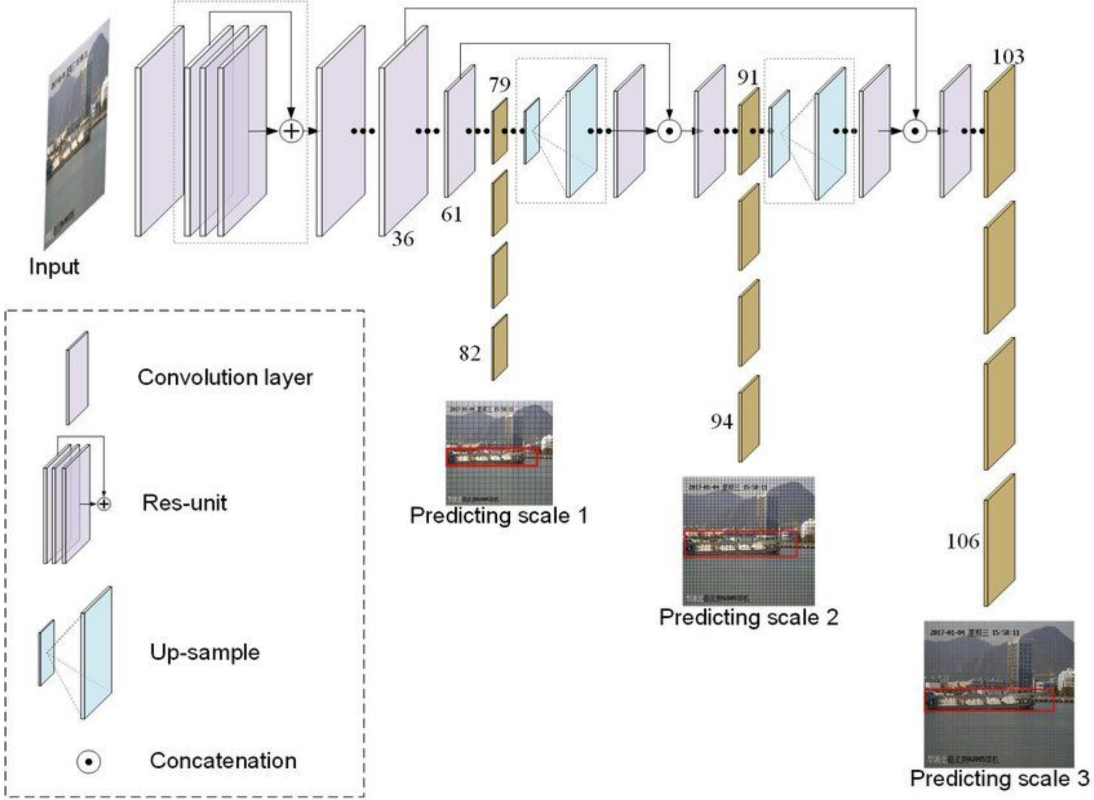


Image credit - Nie, Xin & Yang, Meifang & Liu, Wen. (2019). Deep Neural Network-Based Robust Ship Detection Under Different Weather Conditions. 10.1109/ITSC.2019.8917475.

Analysis of a CNN: YOLOv3 profile

Kernels	contributions
gemm_nn	93%
im2col_cpu	1.3%
Normalize_cpu	2.5%
fill_cpu	0.3%
copy_cpu	0.3%
add_bias	0.2%
scale_bias	0.2%
activate_array	2.0%

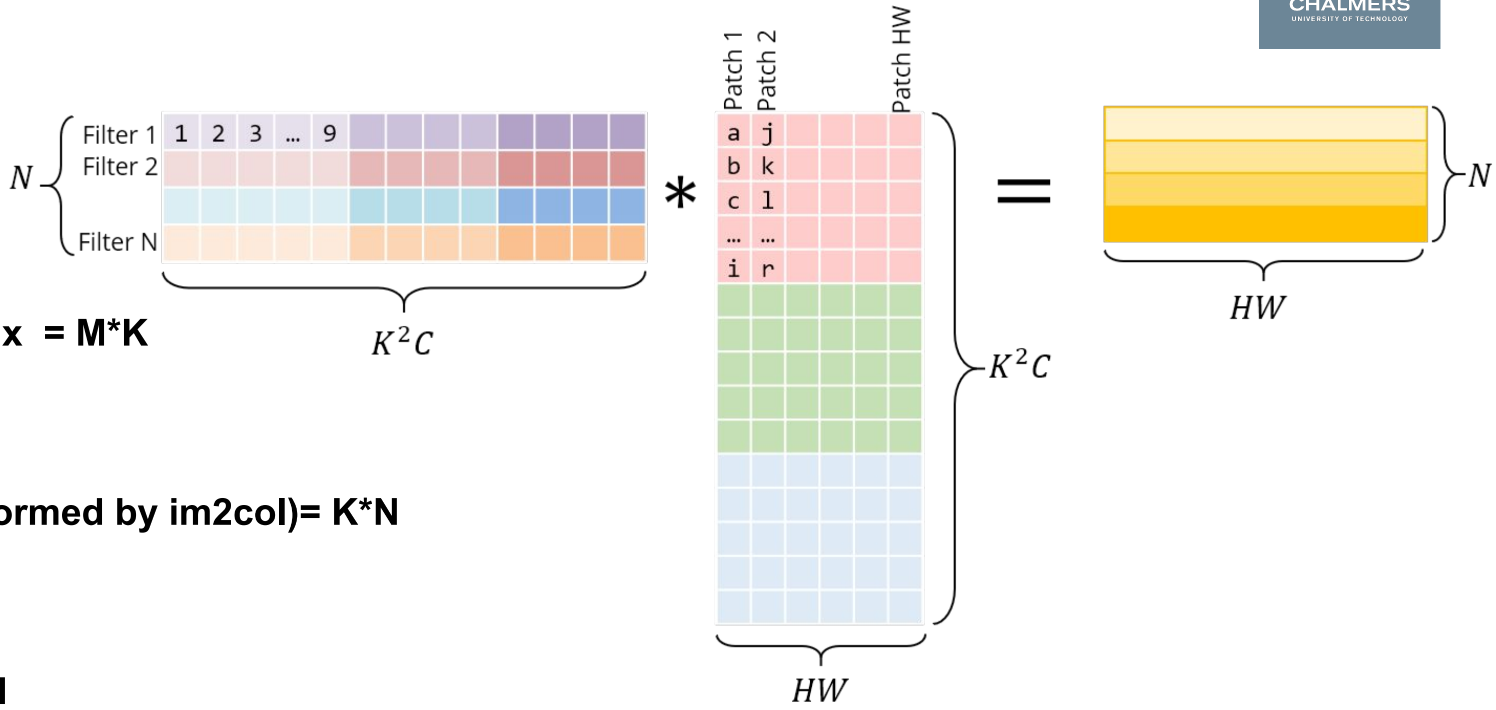
Darknet framework:

- No source modifications
- no vectorization

https://pjreddie.com/darknet/

Profiling of yolov3 network model is collected with "gcc / clang" compiler using "perf"

im2col + gemm for convolutional layer



Weight (Filter) matrix = M*K

M = Number of filters

K = k*k*c

Input matrix (transformed by im2col)= K*N

K = k*k*c

N = H*W

Output Matrix = M*N

M = Number of filters

N = H*W

Image credit: :Manas Sahni, Anatomy of a High Speed Convolution
Link-https://sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/

CNN models & codesign parameters

CNN models

- Darknet with YOLOv3. The dominant layer is the Convolutional layer.
  - YOLOv3: 237MB weights size, Total 107 layers out of which 75 are convolutional.
  - YOLOv3-tiny, ResNet50 models and VGG16 network models have also been explored, but will not be the focus of this presentation
- Input Image Dimension: 768×576

HW explorations (via gem5)

- Three different vector lengths for ARM-SVE : 512-bit , 1024-bit, 2048-bit
- Different vector length for RISC-V Vector: 512-bit, 1024-bit, 2048-bit, 4096-bit, 8192-bit and 16384-bit are considered
- Cache size varies from 256kB to 512MB
- Vector lanes varies from 2 to 8

Developing a fast CNN implementation

Kernels	contributions	ARM-SVE
`gemm_nn`	93%	**manually optimized**
`im2col_cpu`	1.3%	auto-vectorized (armclang)
`Normalize_cpu`	2.5%	manually vectorized
`fill_cpu`	0.3%	auto-vectorized (armclang)
`copy_cpu`	0.3%	auto-vectorized (armclang)
`add_bias`	0.2%	auto-vectorized (armclang)
`scale_bias`	0.2%	auto-vectorized (armclang)
`activate_array`	2.0%	manually vectorized

Algorithmic Optimizations - gemm

BLIS-like algorithmic optimizations in gemm:

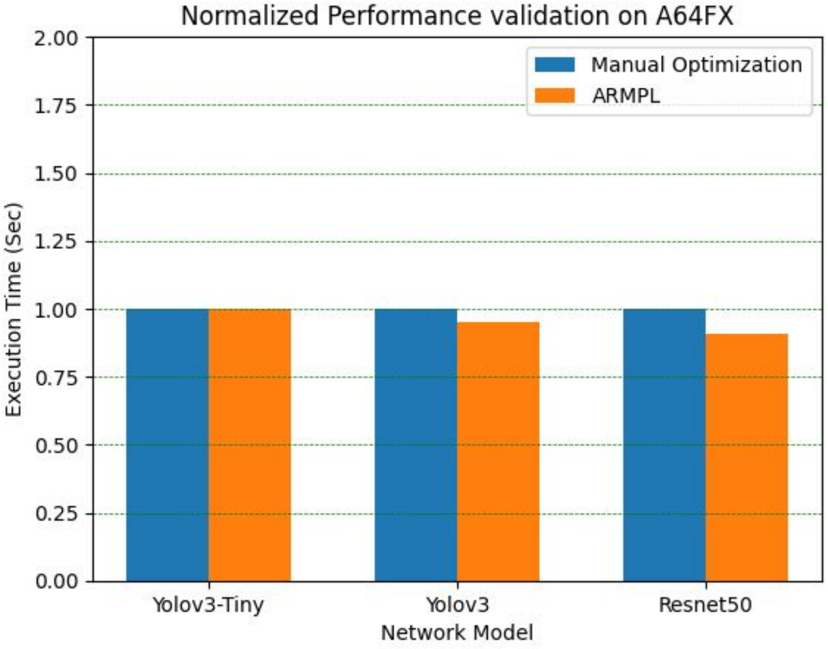
- Vectorization using intrinsic instructions
- 6-loops implementation
- Loop reorder
- Packing of matrices B and A
- Prefetching
- Tuning of block sizes
- Micro-block - VL x 16
- Contiguous memory access
- Maximize register utilization - Loop unrolling
- Fused Multiply-add instructions

Achieves 32x speed-up on A64FX (single core) compare to naive(no vectorization) for yolov3

One frame is processed in ~4 sec

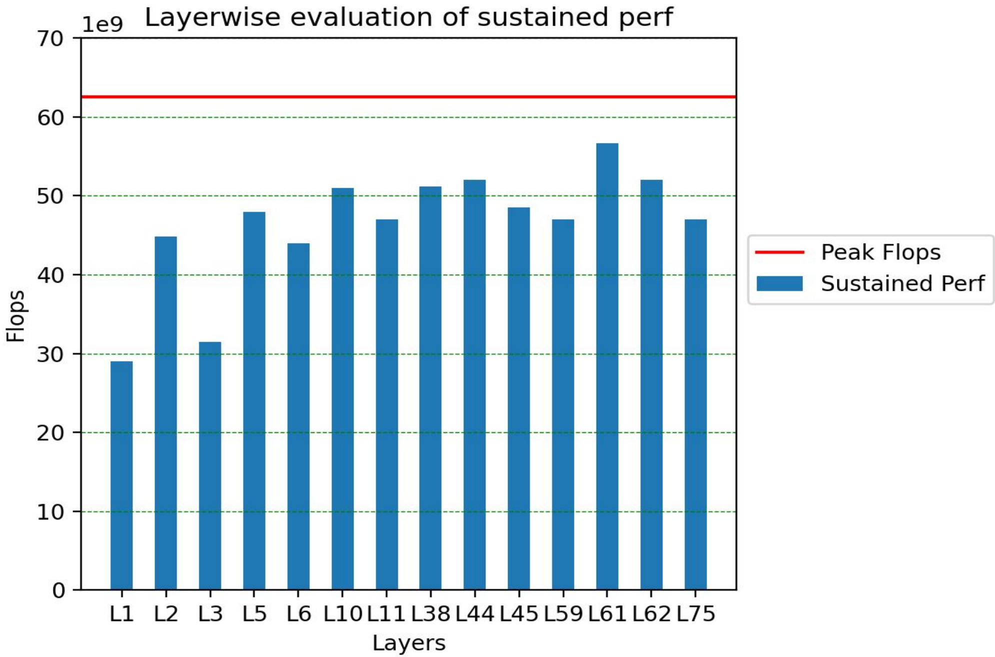
Compared to ArmPL:

- YOLOv3 : ~5-7% of performance diff



Normalized with manual optimization
Yolov3-tiny = 0.25sec, Yolov3 = 3.8 sec Resnet50 = 0.44 sec

A64FX: YOLOv3 - Peak vs achieved



Layers	M	N	K	AI	%
L1	32	369664	27	7.32	46
L2	64	92416	288	26	72
L3	32	92416	64	11	50
L5	128	23104	576	52	77
L6	64	23104	128	21	70
L10	256	5776	1152	101	81
L11	128	5776	256	42	75
L38	256	1444	512	76	82
L44	1024	361	4608	126	83
L45	512	361	1024	88	78
L59	255	361	1024	65	75
L61	256	1444	768	85	91
L62	512	1444	2304	162	83
L75	255	5776	256	63	75

% = Percentage of peak

Layers with small M and K (weight matrix very small)

- Low AI and low % of peak performance.
- L1,L3 achieve only 46% and 50% of peak

Overall some room for improvement, but not much

Gem5 setups for codesign study

➔ **Gem5 + ARM-SVE**

Public version of Gem5 is configured with MinorCPU (in-order processor model) with 2 level of caches.

64KB L1D latency 4 cycles – cache line 512-bit

1MB L2 cache size with latency 12 cycles – cache line 512-bit

Core frequency – 2GHz

➔ **Gem5 + RISC-V Vector (to analyze longer vector lengths and # of vector lanes)**

Gem5* is configured with with Minor (in-order processor model) with 2 level of caches.

64KB L1D latency 4 cycles - cache line 512-bit

1MB L2 cache size with latency 12 cycles – cache line 512-bit

Vector lanes - 8

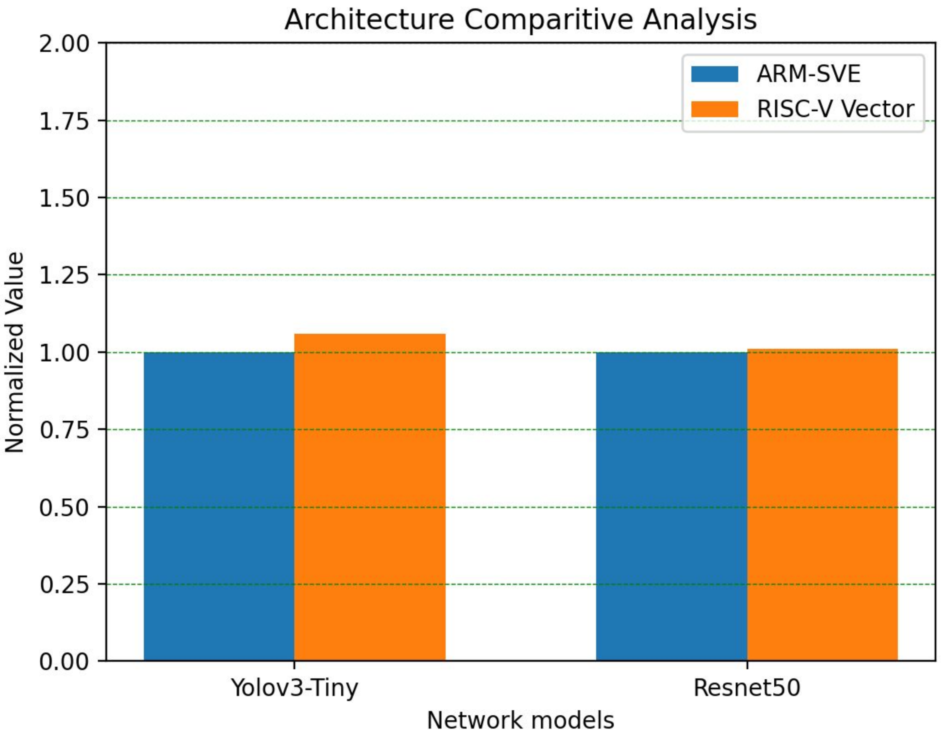
VPU connected with L2 cache.

2 KB VectorCache buffer- To provide data to Vector Engine and to maintain coherency.

➔ No performance difference was found when running the same network on both simulators with the same hardware parameters, **ie ISA difference do not play an important role in performance**

*For further information on Gem5+RISC-V Vector: "Ramírez et al. . 2020. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. ACM Trans. Archit. Code Optim. 17, 4, Article 38 (December 2020)". Github Link: https://github.com/RALC88/gem5/tree/develop

Impact of ISA selection: ARM-SVE vs RISC-V Vector



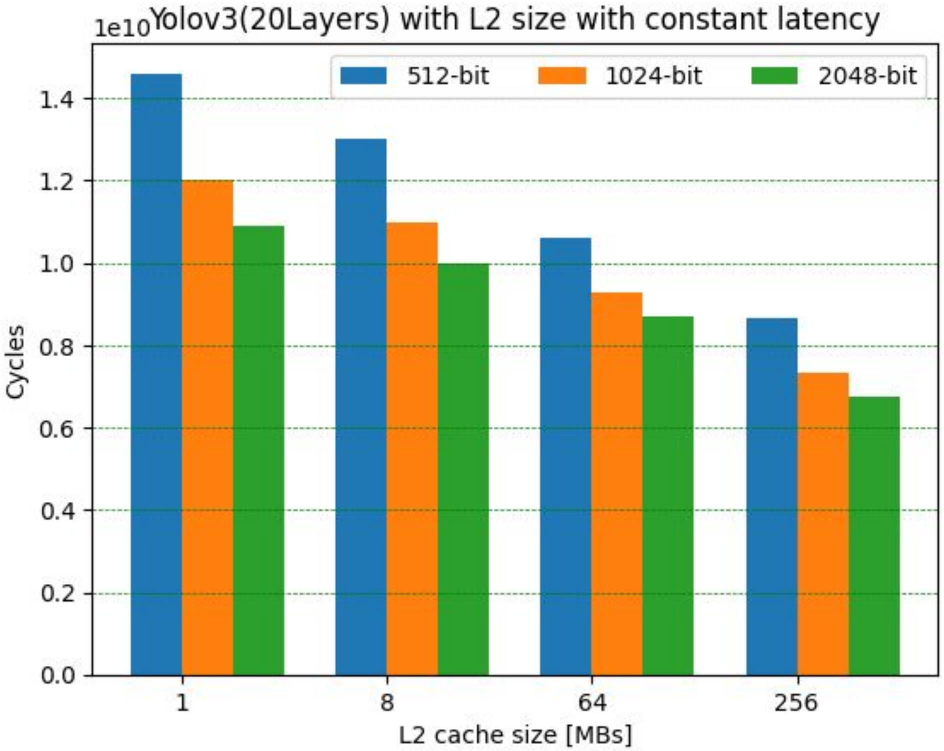
For the comparison, kernels are vectorized:

- intrinsic instructions
- loop unrolling
- vector utilizations
- (No Blis-like optimizations)
- 512-bit VL

Observations

- With same hardware parameters, performance for both RISC-V Vector and ARM-SVE are comparable.
- **ISA has no major impact**

Impact of large cache sizes



YOLOv3

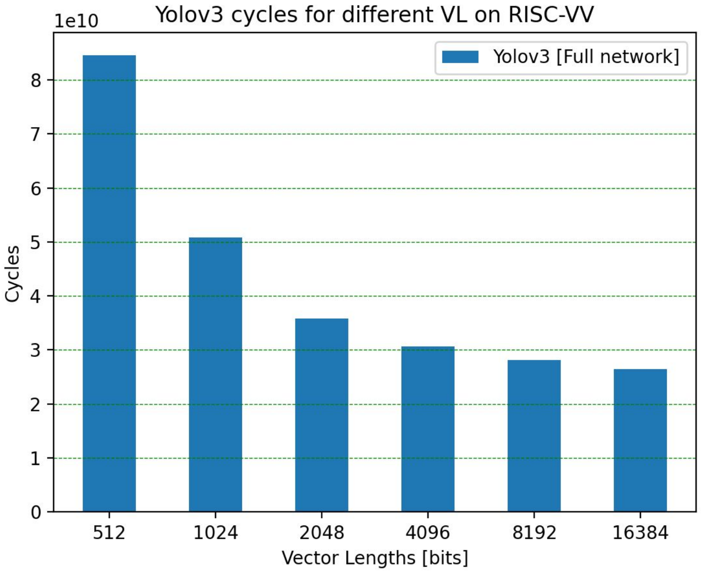
- First 20 layers
- Incl 15 convolutional layers

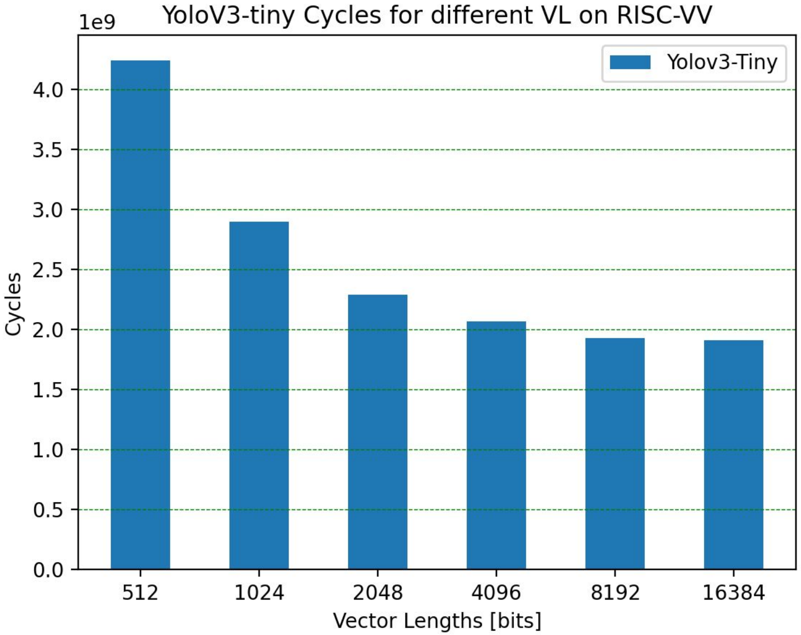
gem5 - arm-sve simulator

From 1MB to 256MB:
2048-bit VL: **1.6x** speed-up
1024-bit VL: **1.6x** speed-up
512-bit VL: **1.7x** speed-up

Fairly uniform performance improvement (1.6-1.7x) across vector lengths

Impact of very Long Vector Lengths

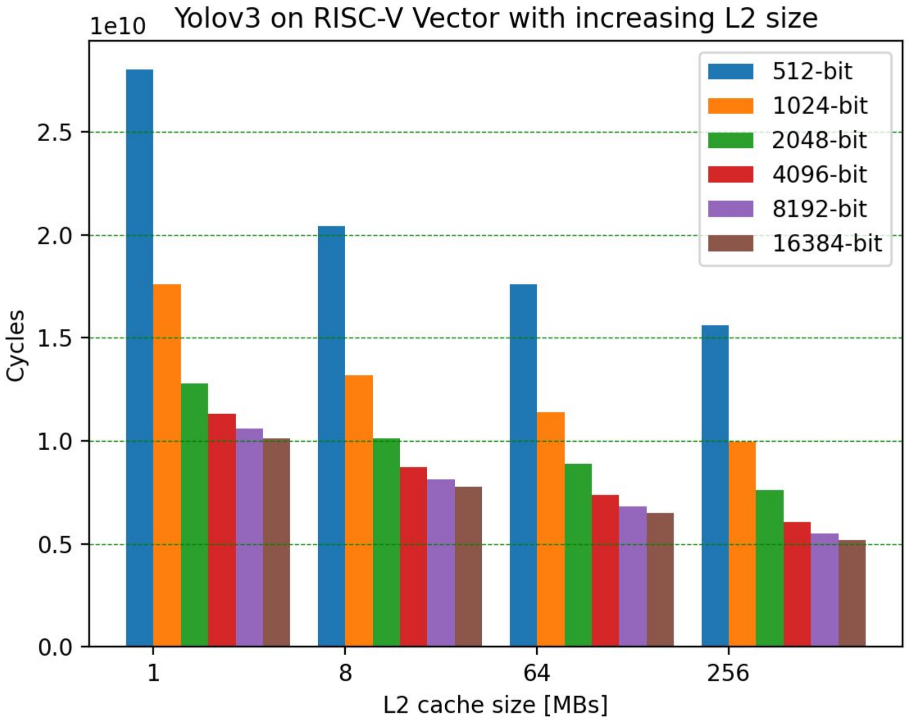




For **YOLOv3** scales upto 16384-bit VL.
512-bit to 16384-bit **Speedup** = **3.2x**
1024-bit to 16384-bit. **Speedup = 1.9x**

For **YOLOv3-tiny,** scales upto 8192-bit VL.
512-bit to 16384-bit **Speedup = 2.21x**
1024-bit to 16384-bit **Speedup = 1.5x**

Impact of larger caches and very long vectors



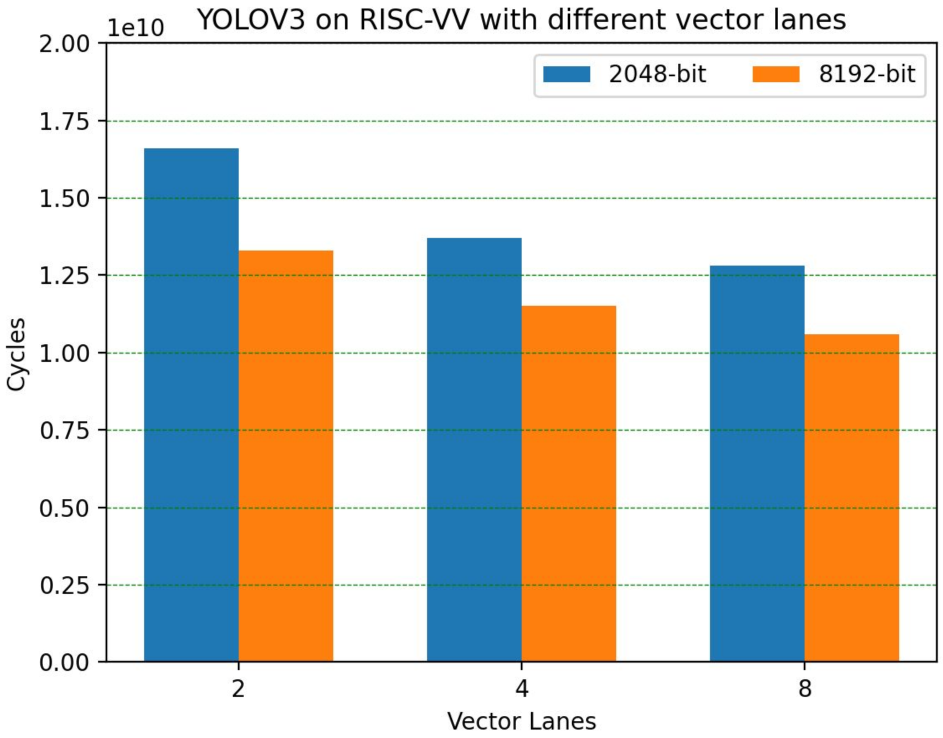
YOLOv3
Total Layers = 20
Convolutional layers = 15

Performance scalability from 1MB to 256MB:

512-bit: 1.8x speedup
1024-bit: 1.77x speedup
2048-bit: 1.72x speedup
4096-bit: 1.86x speedup
8192-bit: 1.93x speedup
16384-bit: 1.95x speedup

Larger caches' benefit improves slightly with vector length

Impact of Vector Lanes



Performance scalability with vector lanes:
Layers used = 20 out of which 15 are convolutional

2048-bit VL:

- 2-4 lanes = 12% of improvement
- 4-8 lanes = 7% of improvement

8192-bit VL:

- 2-4 lanes : 14% of improvement
- 4-8 lanes : 8% of improvement

Observations:

- **Slightly larger impact on longer vectors**
- Increases the performance by 1.25X from 2 lanes - 8 lanes with 8192-bit VL

Parallelization with OpenMP

OpenMP pragmas has been used for multi-core

On gemm kernel:

- OpenMP pragma “parallel for” has been used while doing the packing of matrices A and B.
- OpenMP pragma “parallel for” has been used on top of loop 4 which include loop 4, 5, 6.

On normalize_cpu:

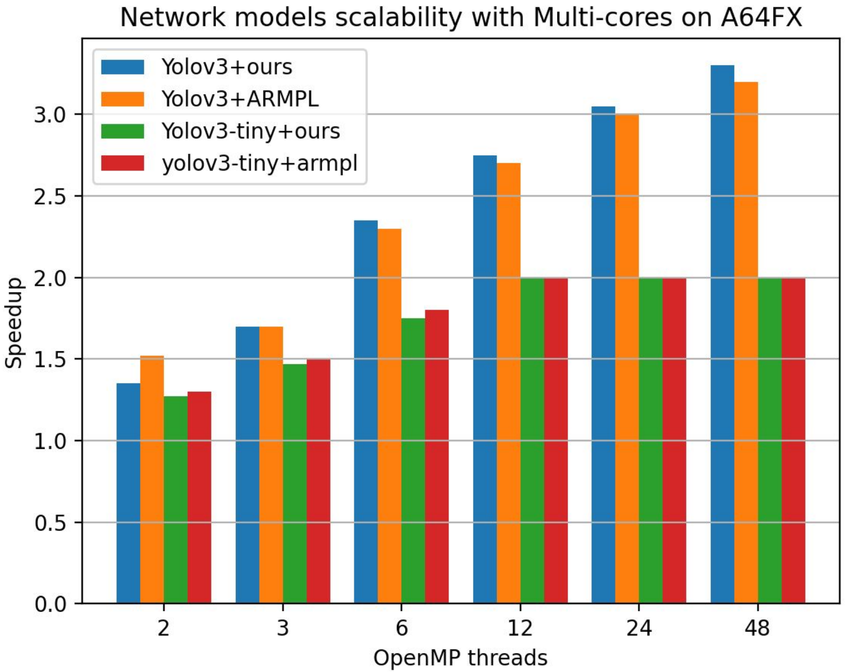
- OpenMP pragma with collapse(3) on outermost loop.

Activations(Leaky and Linear):

- OpenMP pragmas on top-most loop.

```
gemm_PseudoCode()
{
Loop1 (iterate through N increment by blockN)
{
  Loop 2 (iterate through K increment by blockK)
  {
   #pragma omp parallel for
   {
    //logic for pack MatrixB
   }
   Loop 3 (iterate through M increment by blockM)
   {
     #pragma omp parallel for
     {
     //logic for pack MatrixA
     }
       #pragma omp parallel
       {
       #pragma omp for
       Loop 4 (iterate through blockN increment by VL
       {
        Loop 5 (iterate through blockM increment by 16)
        {
         Loop6 (iterate through blockK increment by 1)
         {
           // main logic
         }
 }
```

Multicore scalability on A64FX



Different block sizes have been used for different number of threads to provide more work to each thread.

Observations:

- YOLOv3-tiny improves only up to 12 cores.
- YOLOv3 improves up to 48 cores

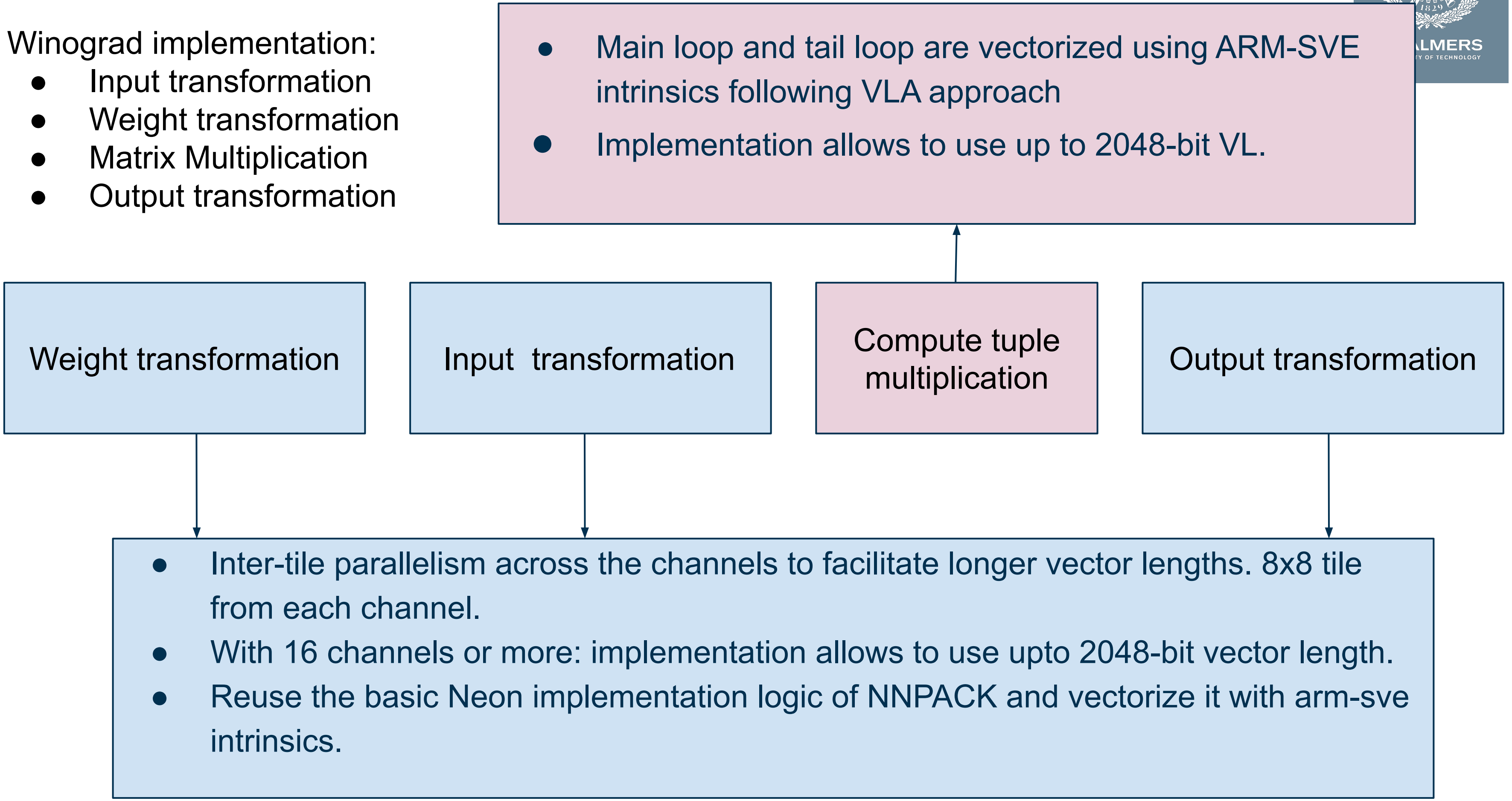
- Workload for each thread decreases but Thread creation and destruction overhead increases.
- By increasing OpenMP threads: most time spent waiting (“kmp_flag_64::wait”)
- **Overall very low parallel efficiency!**

Speedup with multi-core is normalized to single core performance.
For ArmPL = ArmPL with single core.
For ours = Our implementation with single core

Winograd: an alternative approach

Winograd implementation:

- Input transformation
- Weight transformation
- Matrix Multiplication
- Output transformation



- Main loop and tail loop are vectorized using ARM-SVE intrinsics following VLA approach
- Implementation allows to use up to 2048-bit VL.

Weight transformation

Input transformation

Compute tuple multiplication

Output transformation

- Inter-tile parallelism across the channels to facilitate longer vector lengths. 8x8 tile from each channel.
- With 16 channels or more: implementation allows to use upto 2048-bit vector length.
- Reuse the basic Neon implementation logic of NNPACK and vectorize it with arm-sve intrinsics.

Winograd implementation and early results

Transformations on arm-sve

- Used for convolutional layers with unit-stride or stride-2 when kernel size is 3x3.
- Weight transformation is one of the major bottlenecks (but can be done offline for inference – not implemented).

Matrix multiplication

- Main bottleneck - More optimizations are required (future work).

Current version of implementation tested on A64FX:

- 12% of performance* improvement compared to im2col+gemm with VGG16 - since all layers are with 3x3 kernel
- Almost same performance* as YOLOv3 - Mix layers of 3x3, 1x1 kernels with stride 1 or 2.

Tested with ArmIE emulator with different vector lengths (512-bit, 1024-bit, 2048-bit). Co-design study is left as future work

Main observations (I)

Our study describes how performance from vector architecture depends to tuning both HW and SW for CNN.

The following SW optimizations are useful for CNN on vector architectures

- Utilize the vector registers efficiently (to avoid register spilling) and reuse them as much as possible.
- Contiguous load and store to and from vector registers
- Loop unrolling to use multiple multiply-add operations to hide pipeline latency.
- Loop re-order to reuse the vector registers and reduce the pressure on memory
- Packing of matrices to facilitate contiguous cache accesses.
- Tuning of block sizes in such as a way that blockM < blockK< blockN

Main observations (II)

HW codesign:

- Bigger L2 Caches: L2 cache up-to 256MB helped in increasing the performance (as long as cache latency is not severely impacted).
- Longer vector lengths: For YOLOv3 performance scales to 16384-bit VL. For YOLOv3-tiny it saturates at 8192 bits.
- More vector lanes: # vector lanes / simd-units of 8 are beneficial to get high performance with longer vector lengths.
- Overall expected speed-up of 3.8x compared to a configuration similar to A64FX (for YOLOv3) on a single core (~1 FPS)
  - But multicore scalability needs to be improved. Only 3x with 48 cores for YOLOv3.
  - With 48 cores, 256MB L2, 16384-bit VL and 8 vector lanes, performance would only be around ~3.5 FPS

Winograd convolutional layer implementation:

- Improve performance for network models with kernel size of 3x3, up to 12% for VGG16

Thank you!

Backup

Tools and Platforms

Simulators / Emulators for ARM-SVE and RISC-V Vector

- Vanilla gem5 for ARM-SVE with MinorCPU configured with two level of caches
- gem5 for RISC-V Vector* with MinorCPU configured with two level of caches
- ArmIE emulator for validations on ARM-SVE. SPIKE emulator for validations on RISC-V Vector.

Hardware platforms

- Fujitsu (FX1000) A64FX (at BSC) - hardware for ARM-SVE native runs
- JUAWEI - ARM system at Jülich Supercomputing center, consists of Huawei Taishan 2280 servers with HiSilicon 1616 processors - used for compilation and validation with ArmIE
- Tetralith - cluster at Swedish National Supercomputing Center - used for gem5 simulations

Compilers and Libraries

- armclang (native compiler) on JUAWEI node - Arm C/C++/Fortran Compiler version 20.3: for A64FX
- Arm Performance Library - version 20.3 (same as compiler): for A64FX
- Gcc Cross compiler for ARM-SVE - version 10.2.1 20201103: for gem5 results
- Clang cross-compiler for RISC-V Vector - version 12.0: for gem5 results

*For further information on Gem5+RISC-V Vector: "Ramírez et al. . 2020. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. ACM Trans. Archit. Code Optim. 17, 4, Article 38 (December 2020)”. Github Link: https://github.com/RALC88/gem5/tree/develop