EUPILOT: towards an All-European RISC-V-based HPC demonstrator

Miquel Pericàs

Dept. Computer Science and Engineering

Chalmers University of Technology

9th EasyBuild User Meeting, 23-25 April 2024 @ Umeå, Sweden

Agenda

- RISC-V and the EU
- The EUPILOT project
- Some research around RISC-V
- Towards RISC-V based HPC

RISC-V in a nutshell

- RISC-V is an open instruction set architecture (ISA) gaining momentum in the tech industry.
- An ISA essentially acts as the language a processor understands
  - x86, ARM are other well-known ISAs
- Open in this context means the design is freely available for anyone to use and modify.
- Several advantages
  - no royalties
  - customization
  - shared SW ecosystem (if you adhere to the standards)
  - great community!

Why is the EU interested in RISC-V?

- EU has interest in:
  - development of an autochthonous HPC industry
  - need to increase HPC capacity + development of skills
  - open-source hardware, to complete the stack (below the kernel still closed)
- Why RISC-V?
  - Only option for Europe to try to regain position in computing
    - x86, ARM are under control of non-EU entities
  - RISC-V as "language" for academia to communicate ideas to industry
- Note that RISC-V is an open standard, not same as open source Hardware!

Panel on EU & RISC-V in RISC-V Summit Europe 2023
https://www.youtube.com/watch?app=desktop&v=7cLaO-MFY2s&ab_channel=RISC-VInternational

EuroHPC and RISC-V strategy

- EuroHPC Joint Undertaking https://eurohpc-ju.europa.eu/index_en
  - *"EuroHPC JU is a joint initiative between the EU, European countries and private partners to develop a World Class Supercomputing Ecosystem in Europe."*
- Funds big supercomputers, open for use by researchers
  - e.g. LUMI in Finland, Top #5 supercomputer 530 PetaFLOPS (peak)
- EuroHPC R&I projects in which Chalmers participates
  - **European Processor Initiative** (2019-2026)
    - Goal: building ARM host processor and RISC-V based accelerator (EPAC)
  - Development of pilots: **EUPILOT** and EUPEX (2021-)
    - EUPILOT focus on RISC-V accelerator for HPC (focus of this presentation)
  - **eProcessor** (2021-)
    - Developing RISC-V processor with extensions for HPC and Bioinformatics

European Processor Initiative

THE EUPILOT

eProcessor

EPI MAIN OBJECTIVE

To develop European microprocessor and accelerator technology

- Strengthen competitiveness of EU industry and science

SiPearl, Atos, CEA, UniBo, E4, UniPi, P&R, …

Rhea

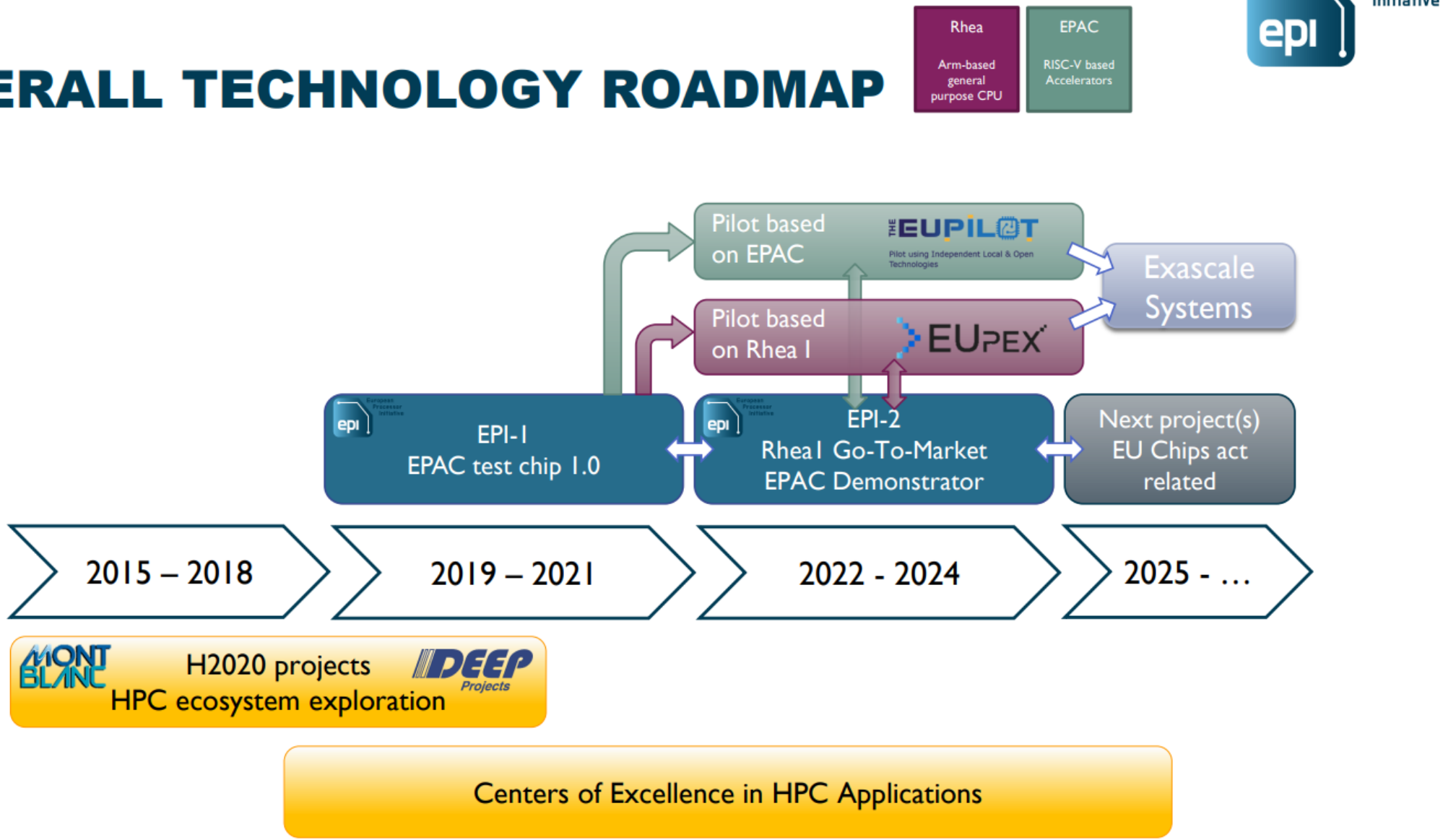
Arm-based general purpose CPU

EPAC

RISC-V based Accelerators

BSC, SemiDynamics, EXTOLL, FORTH, ETHZ, UniBo, UniZG, Chalmers, CEA, E4, Menta, ZPT, …

OVERALL TECHNOLOGY ROADMAP



Pilot using Independent Local & Open Technologies

The European PILOT

Project Overview

Based on slides by Carlos Puchol – Barcelona Supercomputing Center – January 18, 2024

The European PILOT project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No.101034126. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Italy, Switzerland, Germany, France, Greece, Sweden, Croatia and Turkey.

www.eupilot.eu

19 Partners

Academia + Research Institutions

Industrial Partners

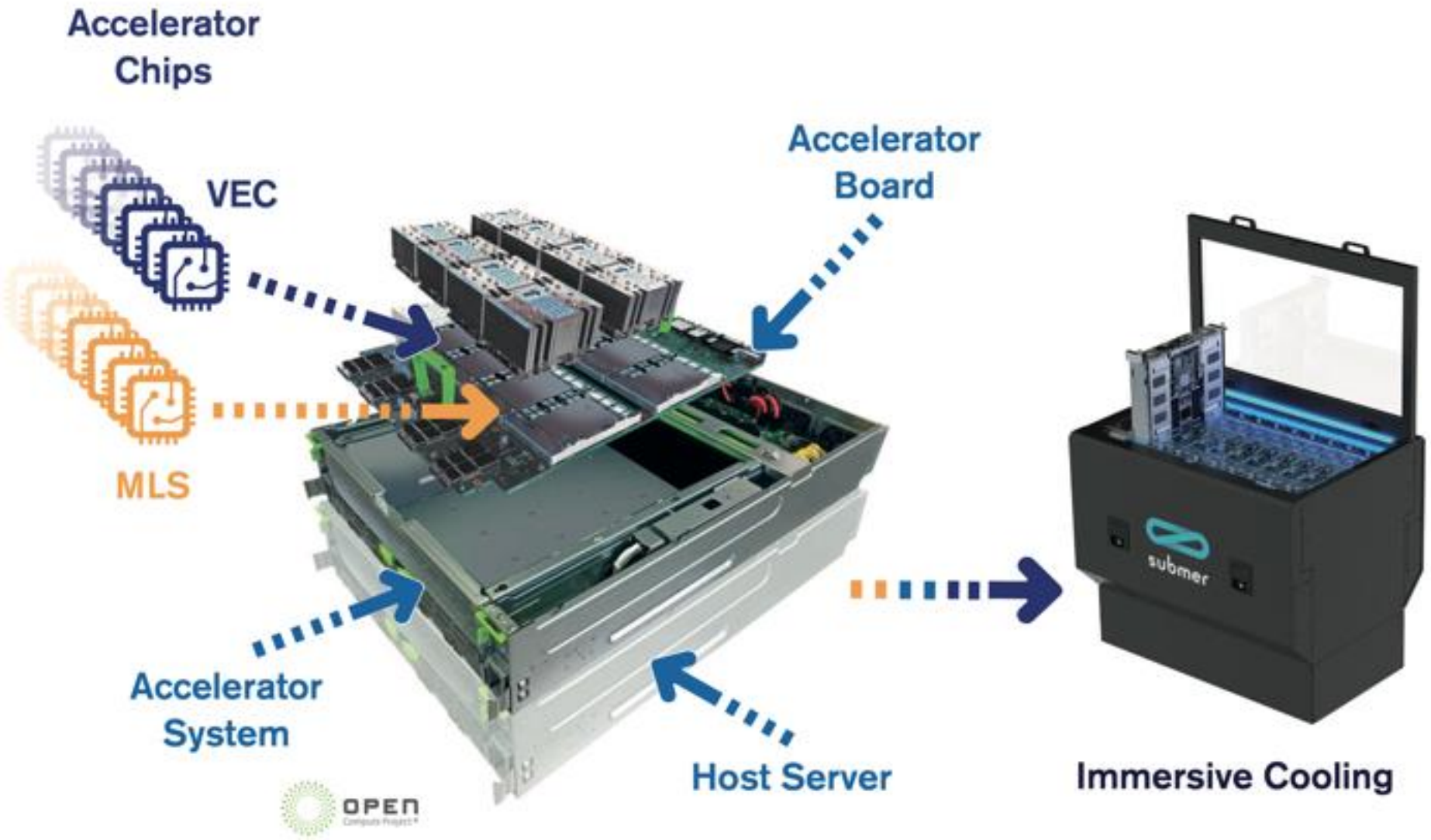
Objectives

- **Demonstrate a European pre-exascale accelerator platform**

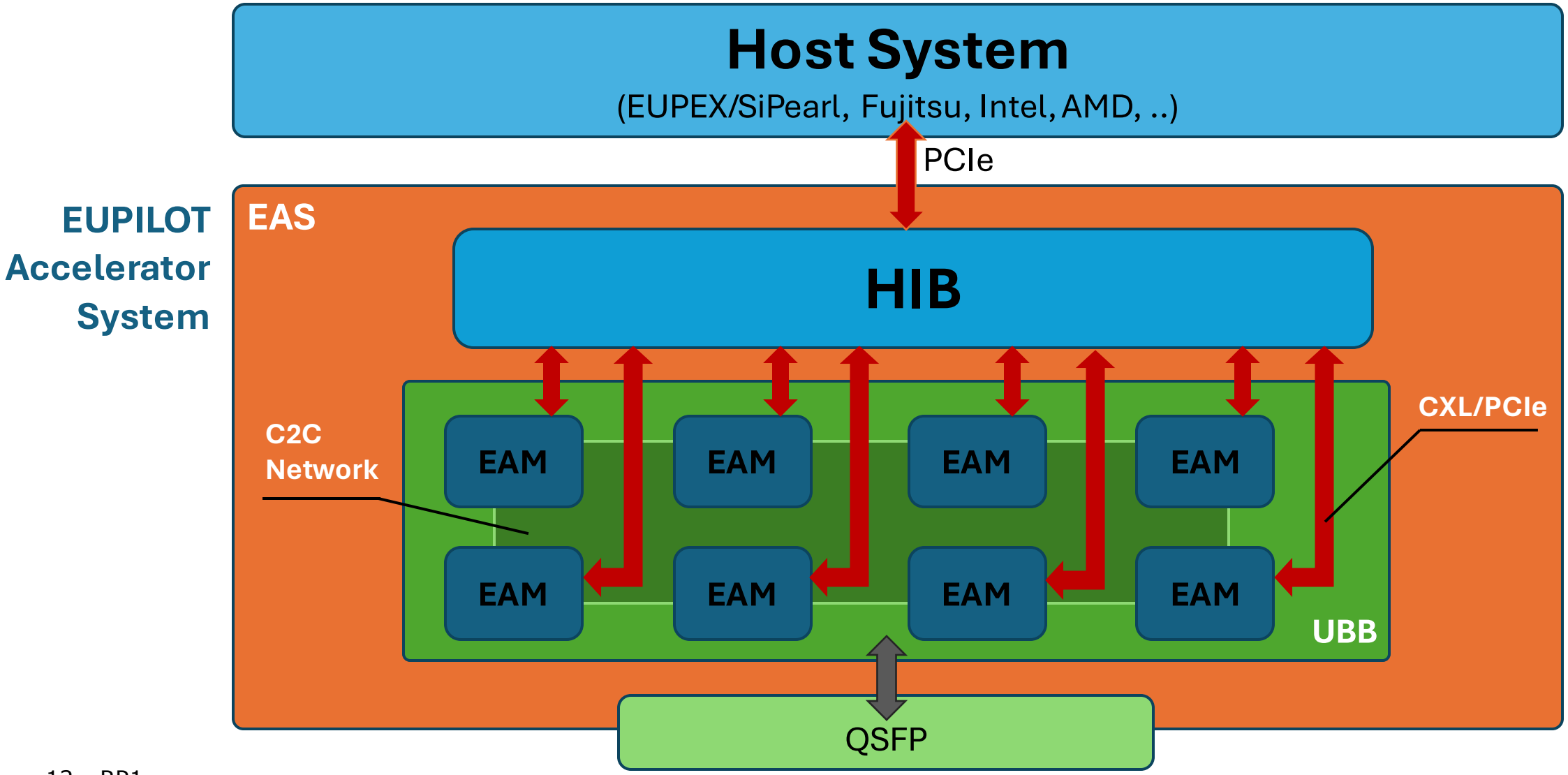
- Design, validate, build and deploy a fully-integrated accelerator platform
- Maximize use of European technology & assets
- Stimulate European collaboration, enable future exascale systems
- SW/HW co-design for improved performance and energy efficiency
- Further extend open source into hardware for HPC
- Leverage open source and the RISC-V ISA

- **Strengthen European digital autonomy and supply chain**

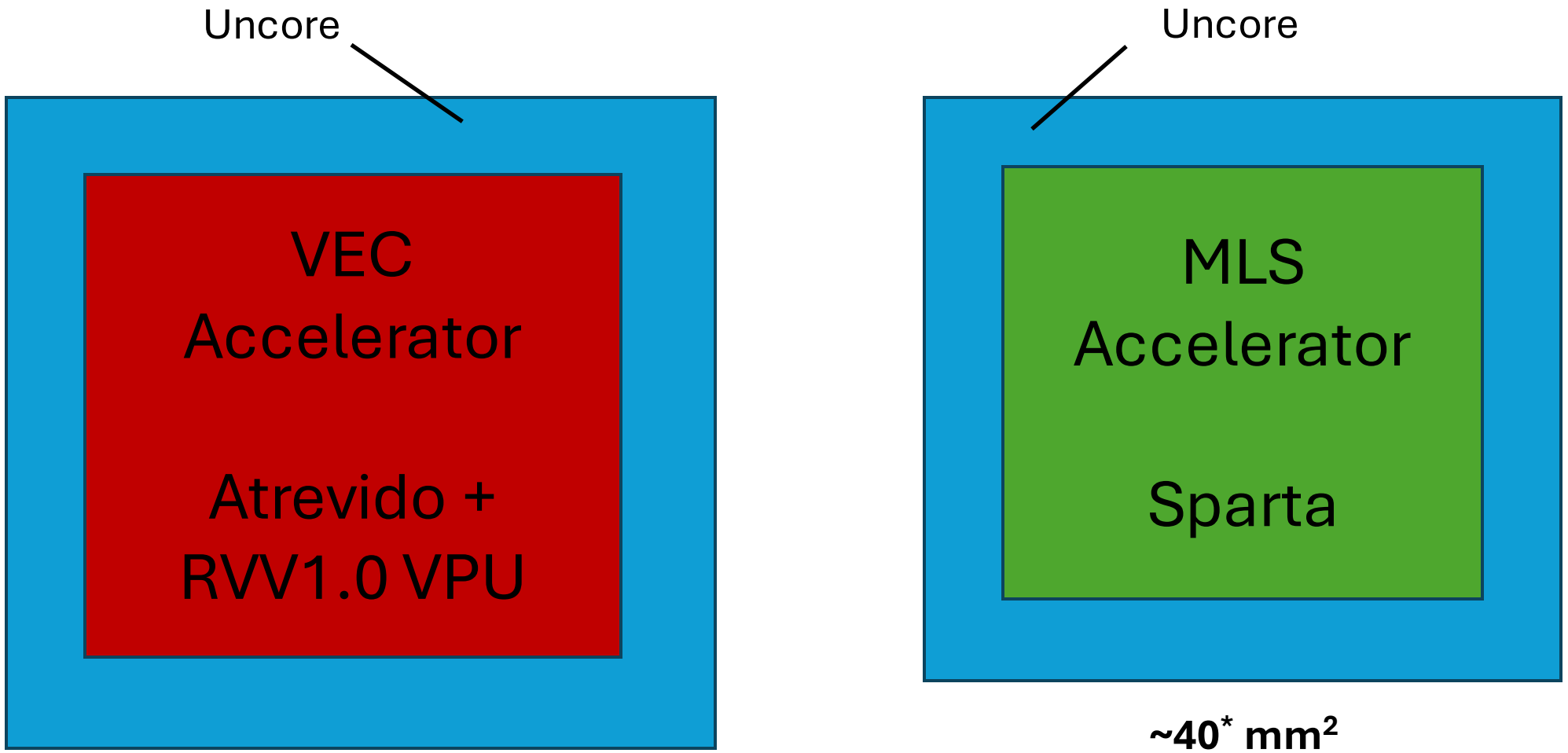
Top Level View



EUPILOT System Architecture



EUPILOT Accelerator Chips



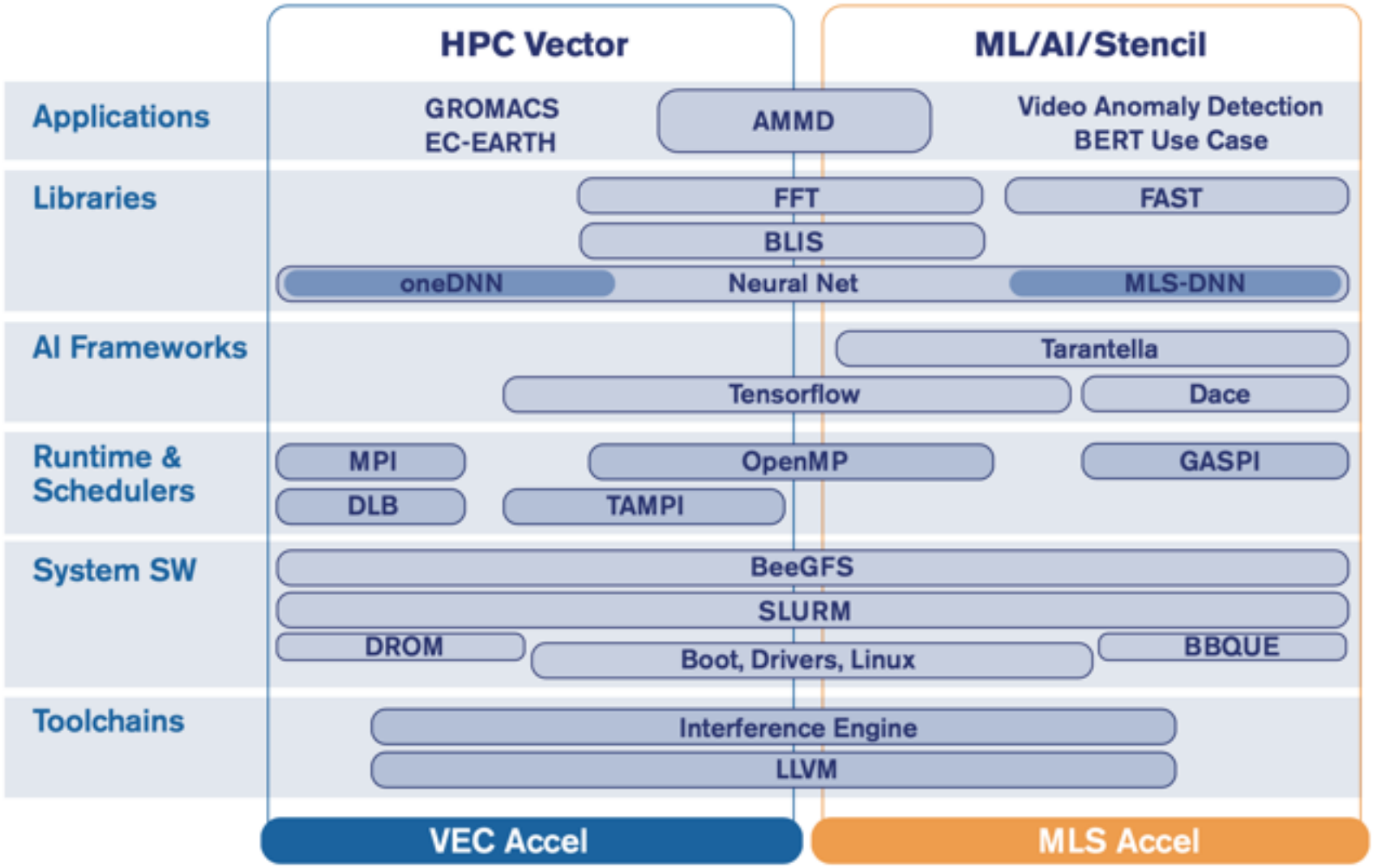
45~50* mm²

~40* mm²

12nm GF 12LPP process

* subject to budget constraints

Software Stack



EUPILOT Peak Performance Projections*

VEC

- 16-tile area budget of ~46mm^2
- Each VPU has 8x FPUs
- 2x FLOPs for FMA
- 64-bit ops
- Target at 1.5GHz
- 16x8x2x1.5 = 384 GFLOPs/chip
- w/RISC-V cores: 432 GFLOPs/chip

~3.5 TFLOPs/EAS (64-bit)

MLS

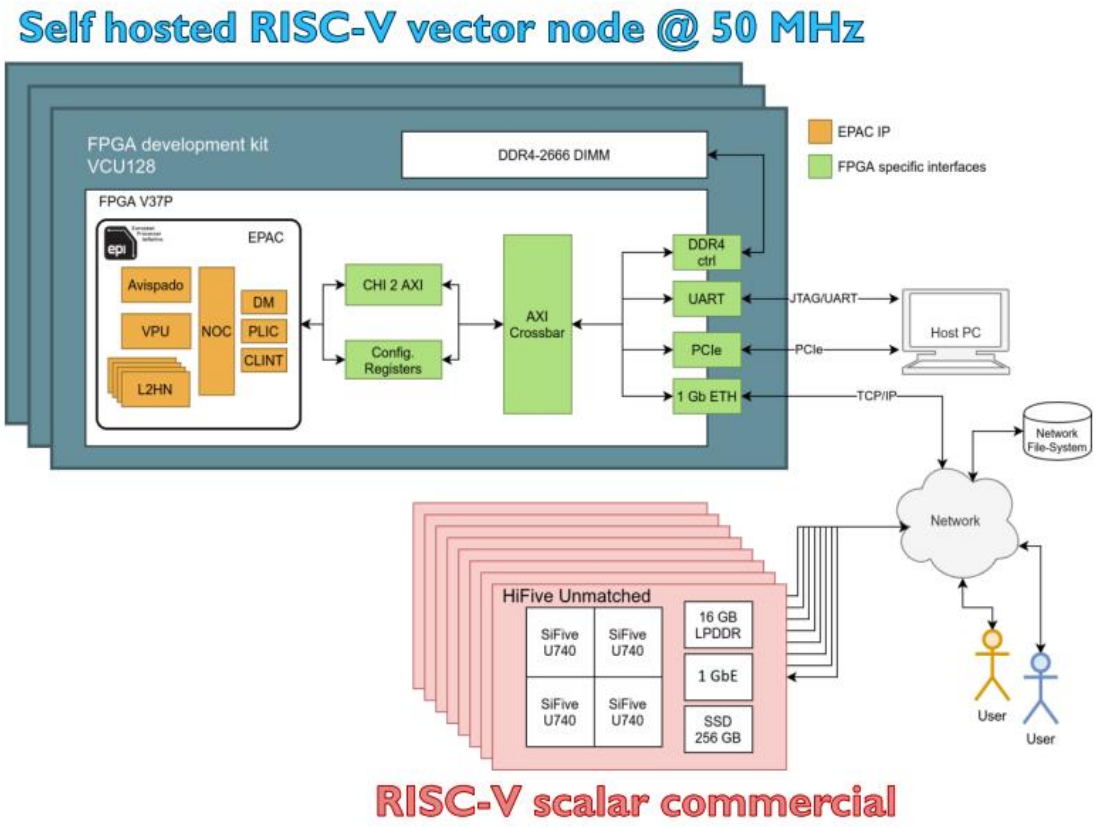
- Mid-area budget of ~35mm^2
- 2 Groups, 6 clusters
- Of (8+1) cores = total 108x FPUs
- 8-bit FP ops
- Target at 1GHz
- ~768 GFLOPs/chip

~6.1 TFLOPs/EAS (8-bit)

* subject to budget constraints

Software Development Vehicles in EPI/EUPILOT

RISC-V PLATFORMS: COMMERCIAL AND FPGA-BASED



Scalar RISC-V commercial platforms coupled with EPAC development platforms

- Hardware/Software infrastructure for Continuous Integration and RTL check
- Playground to demonstrate a full HPC software stack
  - Linux, compiler, libraries, job scheduler, MPI
- Platform to test latest RTL with complex codes
  - Advanced performance analysis tools
  - Accurate timing available

Chalmers' RISC-V tasks in EPI/EUPILOT

- **Software side**
  - Inference on VEC: vectorized convolutions
  - OpenMP runtime for VEC: vectorization of barriers and reductions
  - SYCL for RISC-V
- **Hardware side**
  - Cache coherence, both intra-chip, and inter-chip, based on AMBA CHI
  - Some Co-Design
- **Next Slides**
  - Overview of some of our R&D activities in EPI/EUPILOT

RISC-V Vector Extension

- Why vector?
  - Decode 1 instruction, execute N operations
  - Higher Performance and Energy Efficiency

- RISC-V Vector Extension (RVV) 1.0 ratified Nov 2021
  - EPAC/VEC RISC-VV implements very long vectors (16384b, := 256 DP)
    - In principle, longer vectors can lead to higher efficiency

- Supports Vector Length Agnostic (VLA) programming style
  - Vector ISA is decoupled from Vector Length, unlike e.g. x86 (AVX2, AVX512)
  - Single code supports different vector lengths but may be less performance than VLS (Vector Length Specific) code generation. Note: RVV supports both styles

- Can we use Long Vector RISC-VV accelerators as an alternative to GPGPUs?

RISC-V Vector Programming

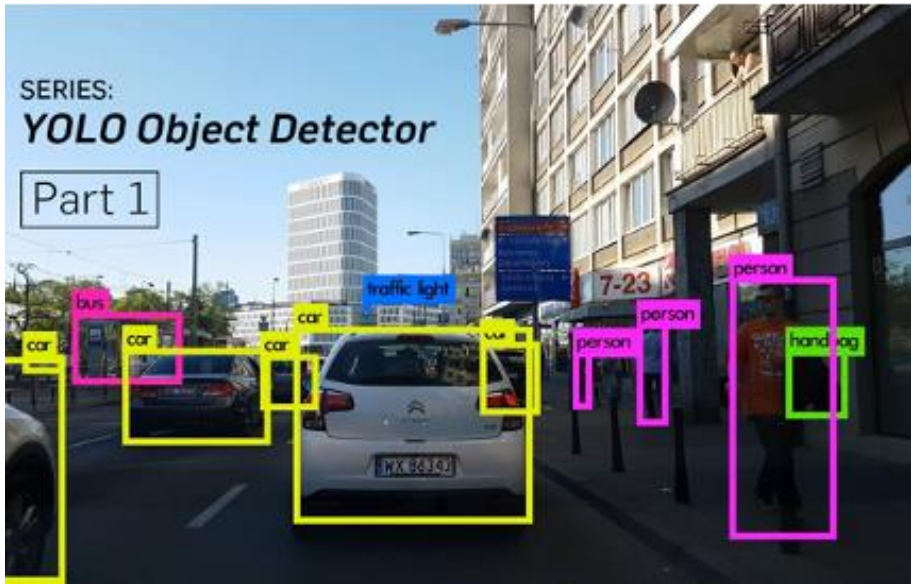
- **Assembly** (*most control*, *least productive*)
  - Very low level, no compiler support
  - Could be JIT (just in time): Similar to assembly, but can leverage runtime information

- **Intrinsics**
  - Compiler builtins that mirror ISA instructions
    - Can use variable names
    - Compiler handles register allocation
  - EPI developed an initial set of RVV builtins (e.g. `__builtin_epi_vsetvl(n - i, __epi_e32, __epi_m8)`)
  - RVV now has a set of standard intrinsics (e.g. `vsetvl_e32m8(n - i)`)
    - Frozen, pending ratification
    - Implementation in GCC, LLVM ongoing

- **"simd" clauses**
  - #pragma omp simd

- **Autovectorization** (*least control*, *most productive*)
  - Work in progress (EPI/EUPILOT focus is on LLVM)

Inference on VEC: CNNs



Inference via Convolutional Neural Network (CNNs) require high throughput and low latency

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

Vector processors can offer

- low latency
- high performance
- energy efficiency

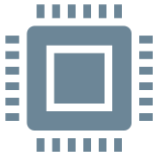
Can we use long vector architectures (eg RVV)?

Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericàs. 2023. *Challenges and Opportunities in the Co-design of Convolutions and RISC-V Vector Processors*. In Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23).

Objective

Algorithmic Optimizations

- Port a **Winograd** algorithm previously optimized for ARM-SVE to RVV by leveraging the available EPI RISC-V intrinsics
  - Identify challenges and propose potential solutions
- Goal to utilize the vector unit and vector registers effectively



Hardware Parameters Tuning

- Vector unit: how long should vector lengths be?
- Caches: how large should caches be for different vector lengths?

Tools

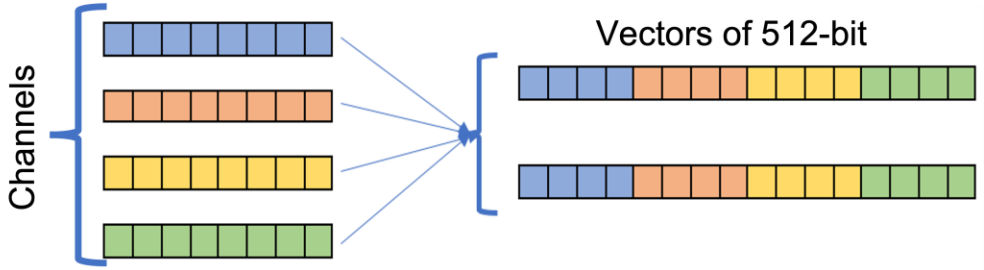
- Network models:
  - YOLOv3: 75 convolutional layers out of 107.
  - **VGG16: 13 convolutional layers out of 16**
  - Implemented in Darknet framework
- Algorithmic implementation
  - NNPACK library for Winograd implementation
- Hardware Exploration:
  - RISC-V Vector Extension: **Gem5 Simulator***
- Compiler
  - RISC-V LLVM/Clang toolchain from the European Processor Initiative (EPI)

*Gem5 Simulator – plctlab. 2022. plct-gem5 (https://github.com/plctlab/plct-gem5/), supports v1.0 “V” extension with max VL of 4096 bits

Winograd: initial ARM-SVE implementation

Transformations:

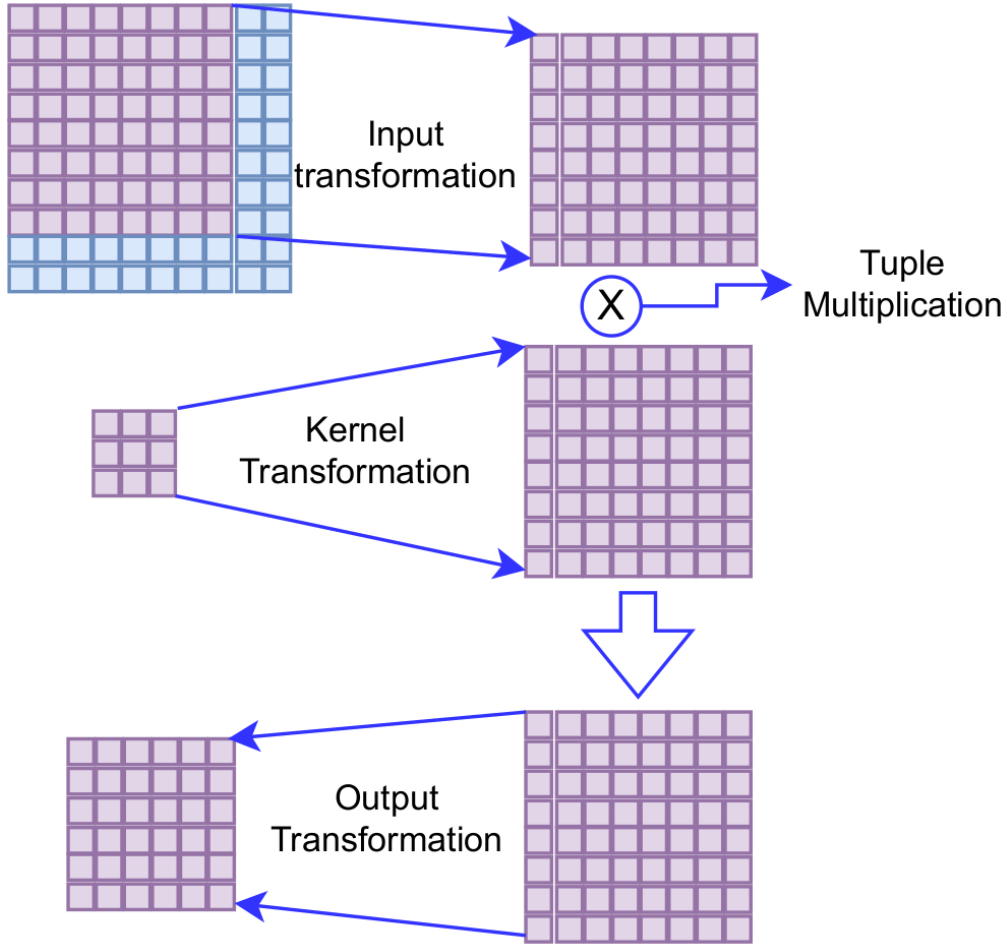
- 8x8 tile from one channel (NNPACK)
- Inter-tile Parallelism across the channels**
- Similarly, 32 channels to utilize 4096-bit VL



1 row of 8x8 tile from 4 channels

Tuple multiplication

- Increase tuple size to 32 with 4 elements in each block to utilize longer vector length.

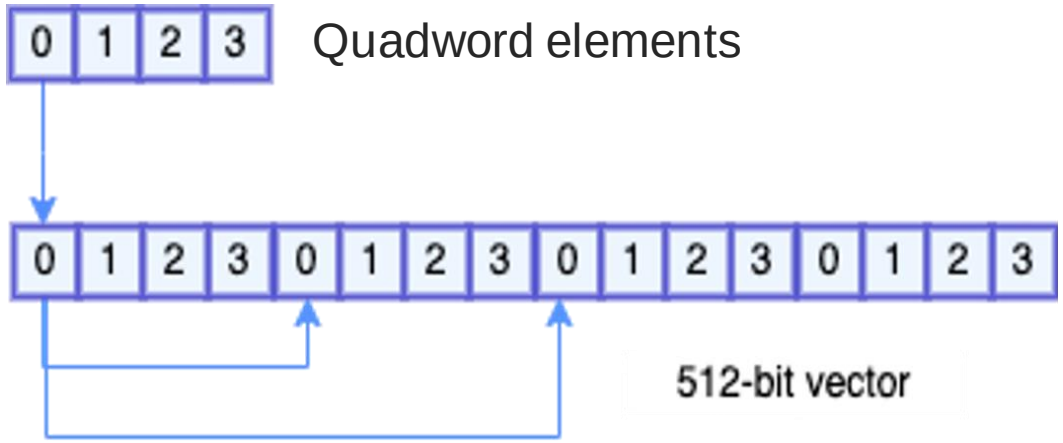


F(6x6,3x3) -> m+r-1 x m+r-1 tile
[m= output, r =kernel]
6x6 output and 3x3 kernel size = 8x8 Tile

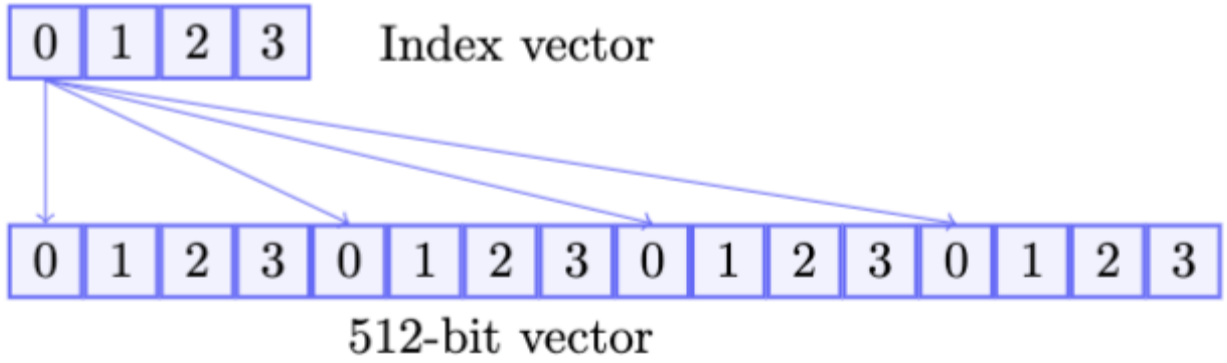
**Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericas. 2023. Accelerating CNN inference on long vector architectures via co-design. In 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 145–155.

Challenge #1: Tuple Multiplication

- Operation: Load Quadword elements in a vector and replicate:
  - No specialized RVV Instruction
- We test two alternatives
  - **Implementation 1: Indexed Load**
  - **Implementation 2: Slideup instructions**



Implementation 2



Implementation 1

Implementation 2 with slideup is ~2.3X faster than implementation 1 with indexed load.
Having specialized instruction likely to be faster, and reduce register pressure.

Challenge #2: Transformations – Transpose four vectors

- Operation: Transpose of 4 vectors in all transformations
  - Again, no RVV instruction is available.
  - **EPI custom extension provides transpose with 2 vectors.**
  - **We tested two alternatives:**
    - **Implementation 1**: unit-strided store followed by Indexed load
    - **Implementation 2:** Strided store followed by unit-strided load

Example for transposing 4 vector registers having elements from 1 channel

No significant difference in performance with both implementations

Potential RVV extension: vector transpose of 4 vectors, eliminates need for extra memory operations

Challenge #3: Transformations – Passing register references

Problem: Cannot pass references to vector registers as parameters to a procedure

- require intermediate vector registers to store the intermediate vector data.
- +require ~30 lines of code at 6 places in the input transformation kernel.
- Problems:
  - Register spilling
  - Less Programmability
- **Potential Workaround: Macros** can improve programmability, but it will still be required to have **intermediate registers. Hence the problem of register spilling will remain**

Being able to pass references to vector registers would improve programmability and reduce the chances of register spilling

VGG16: Analysis

VGG16:

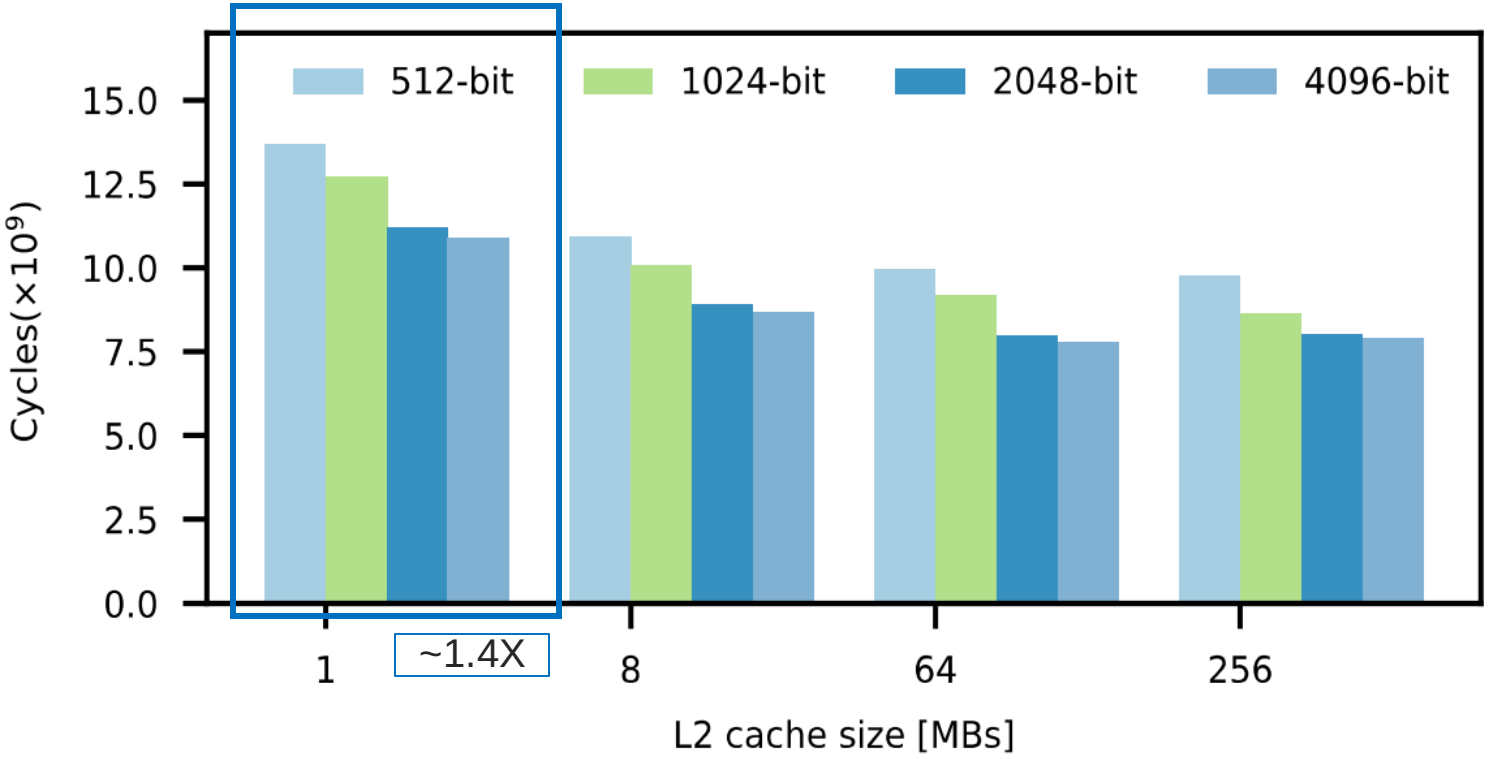
- 3x3 kernel size with stride 1: **Winograd**
- All the layers use Winograd algorithmic optimizations

Comparison with im2col+GEMM:

- 2048 bits VL and an L2 cache of 1MB modeled with gem5
- **1.2x performance improvement** compared to the **pure im2col+GEMM** approach.
- Similar performance compared to our optimized ARM-SVE implementation (on gem5)

HW Design Space: VGG16

Impact of vector lengths and L2 cache size with Winograd on RISC-VV@gem5 for VGG16.



Impact of Vector lengths:

- **No scalability beyond 2048-bit.**
- No significant difference in the number of instructions from 2048-bit to 4096-bit vector lengths

Impact of L2 caches from 1MB to 64MB:

- ~1.3X performance improvement
- **No performance improvement beyond 64MB L2 cache**

2K vector length with 64MB caches can provide up to ~1.8x speedup

Current work: Integration of RVV Winograd, im2col and Direct Conv into oneDNN. Potential contribution to EUPILOT Software Stack.

oneDNN

- Open-source performance library designed to accelerate deep learning applications
  - https://github.com/oneapi-src/oneDNN
- Optimized for Intel Architecture
- Available for Intel, AMD, ARM, IBM power, z/Architecture, RISC-V
- Provides highly optimized implementations for standard routines such as convolution, batch normalization, and pooling operations
- We are integrating optimized implementations of im2col/gemm and Winograd
  - Work-in-progress. No results yet

RVJIT

BSC developed a RISC-V JIT for oneDNN. Pushes code to be executed later

Example:

```
void vadd_vv(vr_t vd, vr_t vs1, vr_t vs2,
vmask_t vm = vmask::unmasked) {
      push(rvj_vadd_vv(vd, vs1, vs2, vm));}
```

Pushes `vadd.vv vd, vs2, vs1, vm # Vector-vector integer addition`

Following:

https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

https://github.com/riscv/riscv-opcodes/blob/master/rv_v

- Need to keep track of registers manually
- Creating assembly code versions of functions
+ Easy to extend and add instructions to JIT
+ Enables runtime optimizations

Adding instructions to the JIT

Find instructions in specifications

```
vmand.mm vd, vs2, vs1   # vd.mask[i] =   vs2.mask[i] &&  vs1.mask[i]
vmnand.mm vd, vs2, vs1  # vd.mask[i] = !(vs2.mask[i] &&  vs1.mask[i])
vmandn.mm vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] && !vs1.mask[i]
vmxor.mm  vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] ^^  vs1.mask[i]
vmor.mm   vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] ||  vs1.mask[i]
vmnor.mm  vd, vs2, vs1  # vd.mask[i] = !(vs2.mask[i] ||  vs1.mask[i])
vmorn.mm  vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] || !vs1.mask[i]
vmxnor.mm vd, vs2, vs1  # vd.mask[i] = !(vs2.mask[i] ^^  vs1.mask[i])
```

Look up opcodes

```
vmandn.mm     31..26=0x18 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmand.mm      31..26=0x19 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmor.mm       31..26=0x1a 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmxor.mm      31..26=0x1b 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmorn.mm      31..26=0x1c 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmnand.mm     31..26=0x1d 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmnor.mm      31..26=0x1e 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
vmxnor.mm     31..26=0x1f 25=1 vs2 vs1 14..12=0x2 vd 6..0=0x57
```

Add instruction to be used by program

```
void vmand_mm(vr_t vd, vr_t vs1, vr_t vs2) {
        push(rvj_vmand_mm(vd, vs1, vs2));
    }
```

Implement Instruction with opcode

```
rvj_instr rvj_vmand_mm(REGV vd, REGV vs1, REGV
vs2) {
    return opMVV(vd, vs1, vs2, 0x19,
rvj_unmasked);
}
```

RVJIT code example usage: for loop

```
const gpr_t channel = tmp.pick();
const gpr_t channel_end = tmp.pick();
load_constant(channel, 0);
load_constant(channel_end, channels_col);
L("channels");
//for (c = 0; c < channels_col; ++c)
 // Channel loop

addi(channel, channel, 1);
blt(channel, channel_end, "channels");
```

RVJIT code example usage: intrinsic conversion

```
 //Index calculation
vl(wcol, index, src_sew); // load
vmv_sx(OFFSET, w_offset); // broadcast
vmv_sx(PAD, pad_reg); //broadcast
vmv_sx(STRIDE, stride_reg); //broadcast
vmul_vv(intermediate1, STRIDE, wcol); // multiplication
vadd_vv(IMCOL, intermediate1, OFFSET); // addition

vmv_sx(WIDTHCOL, width_end); //broadcast
vmv_sx(INTER, intermediate); //broadcast
vmul_vv(intermediate2, INTER, WIDTHCOL); // multiplication
vadd_vv(colindex, intermediate2, wcol); // addition
vsub_vv(IMCOL, IMCOL, PAD); // subtract
```

```
//Index calculation
__epi_2xi32 wcol = __builtin_epi_vload_2xi32(&w_col[w+0], gvl); //load
__epi_2xi32 OFFSET = __builtin_epi_vmv_v_x_2xi32(w_offset, gvl); //broadcast
__epi_2xi32 PAD = __builtin_epi_vmv_v_x_2xi32(pad, gvl); //broadcast
__epi_2xi32 STRIDE = __builtin_epi_vmv_v_x_2xi32(stride, gvl); //broadcast
 __epi_2xi32 intermediate1 = __builtin_epi_vmul_2xi32(STRIDE, wcol, gvl);  //multiplication
 __epi_2xi32 imcol = __builtin_epi_vadd_2xi32(intermediate1, OFFSET, gvl);  //Addition

__epi_2xi32 WIDTHCOL = __builtin_epi_vmv_v_x_2xi32(width_col, gvl); //broadcast
__epi_2xi32 INTER = __builtin_epi_vmv_v_x_2xi32(intermediate, gvl); //broadcast
 __epi_2xi32 intermediate2 = __builtin_epi_vmul_2xi32(INTER, WIDTHCOL, gvl);  //multiplication
 __epi_2xi32 colindex = __builtin_epi_vadd_2xi32(intermediate2, wcol, gvl);  //addition
 imcol = __builtin_epi_vsub_2xi32(imcol, PAD, gvl);  //subtract
```

RVJIT code

EPI LLVM intrinsics

RVJIT code example usage: Pseudo assembly

```
if (i+1 < M)      {__builtin_epi_vstore_2xf32(&C[(i+1)*ldc+j], vc1, gvl);}
if (i+2 < M)      {__builtin_epi_vstore_2xf32(&C[(i+2)*ldc+j], vc2, gvl);}
if (i+3 < M)      {__builtin_epi_vstore_2xf32(&C[(i+3)*ldc+j], vc3, gvl);}
```

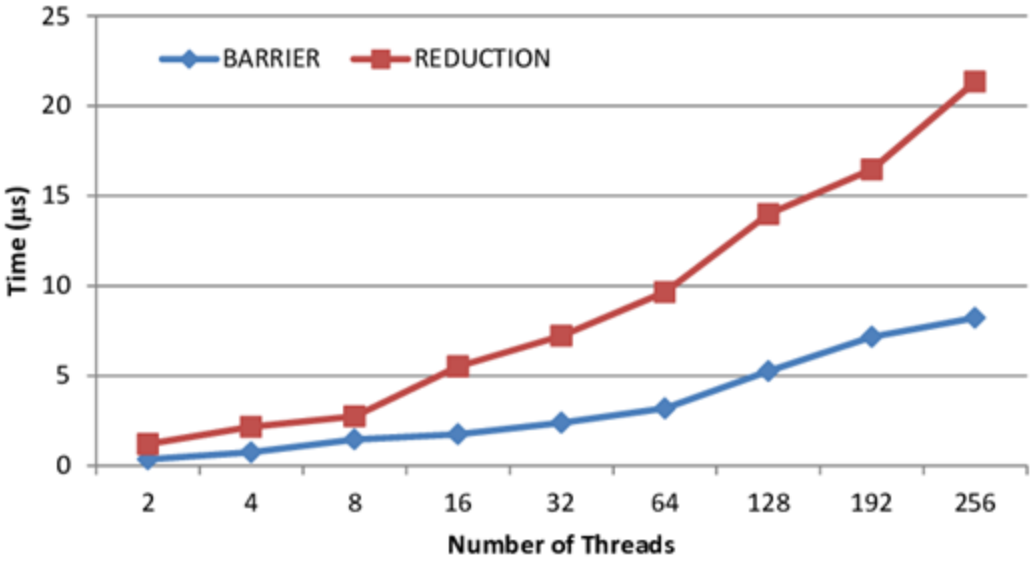
```
L("vc12");
addi(tmp1, i, 1);
bge(tmp1, M_reg, "vc22");
addi(c_index, i, 1);
mul(c_index, c_index, ld_reg);
add(c_index, c_index, j);
mul(c_index, c_index, sew);
add(c_index, c_index, vdst);
vs(vc1, c_index, src_sew);
```

```
L("vc22");
addi(tmp1, i, 2);
bge(tmp1, M_reg, "vc32");
addi(c_index, i, 2);
mul(c_index, c_index, ld_reg);
add(c_index, c_index, j);
mul(c_index, c_index, sew);
add(c_index, c_index, vdst);
vs(vc2, c_index, src_sew);
```

```
L("vc32");
addi(tmp1, i, 3);
bge(tmp1, M_reg, "skip3");
addi(c_index, i, 3);
mul(c_index, c_index, ld_reg);
add(c_index, c_index, j);
mul(c_index, c_index, sew);
add(c_index, c_index, vdst);
vs(vc3, c_index, src_sew);
L("skip3");
```

Vectorized Barrier and Reduction in LLVM OpenMP Runtime

- Barrier and Reduction commonly used operations in parallel programs
- Challenges
  - Resources wait idle
  - Overhead increases with increasing # of threads
  - # of cores/node are expected to increase at exascale
- Goals
  - Low overhead barrier and reduction in OpenMP
  - Utilize vector/simd unit to reduce overhead



LLVM OpenMP barrier and reduction overheads using the EPCC benchmark on Intel KNL

Muhammad Nufail Farooqi and Miquel Pericàs. 2021. Vectorized Barrier and Reduction in LLVM OpenMP Runtime. In OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021

Vectorized Barrier in LLVM

- **Data structure**
  - Shared array for each team of threads
  - Shared array is initialized to 1
  - Gather/Release -> see pseudocode
  - Tree pattern with configurable branching
    - shared array for each tree level
- **Implemented using Intrinsics**
  - RISC-V Vector Extension
  - Arm-SVE
  - Intel AVX
- Similar approach for Reductions, except that we also need to modify clang

Gather phase:

```
vec_array[thread_id]=0;
if(primary_thread){
  do {
    tmp_vec[1:VLEN]=0;
    for (i=0;i<nThreads;i+=VLEN) {
        tmp_vec |= vector_load(&vec_array[i],VLEN);}

  } while (tmp_vec);
```

Release phase:

```
if(primary_thread){
    vec_array[1:nThreads]=1;
} else {
     while(!vec_array[thread_id]);
  }
```

- VLEN is length of the vector unit
- nThreads are number of threads in a team

Evaluation

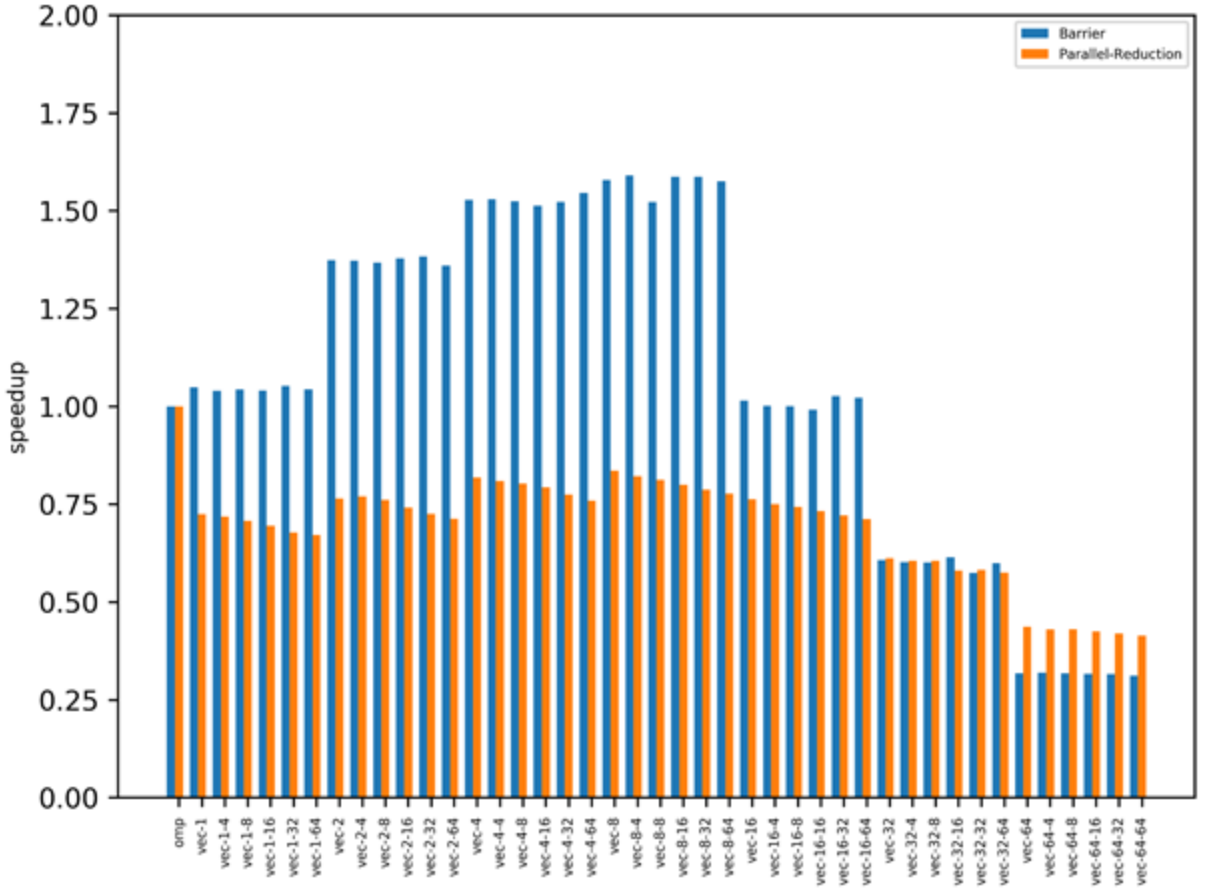
- OpenMP *parallel*, *barrier* and *reduction* microbenchmarks
- Baseline is LLVM OpenMP's default barrier and reduction
- Performance studies using Intel KNL and Fujitsu A64FX
- For RISC-V Vector Extensions, validated functional correctness using qemu+vehave (RISC-V vector emulator)

Table 1. Machine Specifications.

	Intel KNL	Fujitsu A64FX
Cores	68	48
L1	32 KB	64 KB
L2	34 MB (private)	32 MB (shared)
Memory	192 GB	32 GB
Bandwidth	90 GB/s	1 TB/s

Performance Results

- Use Intel KNL as proxy: supports 512-bit vectors, 64 cores w/ 4x HW multithreading
- Performance: trade-off between false sharing and on-chip memory traffic
- Best performance achieved when max branching factor is used (=linear pattern)
- Max 2.2x speedup on Intel KNL for 128 threads
- Next: evaluate barriers and reductions performance on EUPILOT prototype (FPGA with multiple cores and vector units)



Performance of vectorized barrier and reduction with varying padding for barrier flags and reduction array (256 Threads on Intel KNL with snc4 cluster mode)

Some thoughts on the road ahead for RISC-V based HPC

- Quite some progress has been made, but there is still more work
- Need progress at all levels of the stack
  - High Performance RISC-V based systems
  - Future RISC-V ISA extensions
  - Toolchain/Libraries readiness
  - What else?
- Getting started with RISC-V

RISC-V HPC Systems

- Today, few commercial RISC-V chips target HPC
- T-head C920, targeting v0.7.1 vector spec, only "independently evaluated" board out there (*)
  - Sophon SG2042 includes 64x C920 cores
  - Requires custom-made gcc compiler for vectorization
- But: there are many products on the horizon!
  - Tenstorrent (Tensix AI, Ascalon RISC-V Core)
  - Ventana Micro (Veyron V2)
  - Esperanto Technologies (ET-Minion, ET-Maxion)
  - Semidynamics (Atrevido, Custom Tensor Instructions)
  - InspireSemi (Thunderbird Compute Accelerator)
  - Probably many more
  - All of them support RVV-1.0 Spec, should offer good interoperability!
    - How about Performance portability?

(*) https://arxiv.org/pdf/2309.00381.pdf

RISC-V HPC specs

- RISC-V **Vector extension** continues to be developed
- Topics that are under discussion in the Vector SIG
  - ❑ Subsetting of the existing vector Spec (particularly for embedded devices)
  - ❑ Self-contained vector instructions (more arch. registers, multiple mask regs, direct encoding of properties)
  - ❑ Sparsity support
  - ❑ GPU-like capabilities (streaming, control flow divergence, large register files)

- Beyond vectors, **Matrix extensions** are critical for AI workloads, also HPC
  - ❑ Two specs are currently being developed
    - Integrated Matrix Extension
    - Attached Matrix Extension

- **Performance Events**:
  - ❑ Current Zihpm extension provides programmable performance events (But: implementation specific)
  - ❑ Performance events TG has been setup to standardize a set of performance events

Software Toolchains

- Rich SW ecosystem needs to be ported and optimized for RISC-V
- In EPI/EUPILOT we are working on several toolchains
  - ❑ LLVM (C/C++/Fortran), autovectorization, OpenMP, MPI, SLURM, DaCe, oneDNN, BLIS, FFT, SYCL, Linux, …
- RISE (RISC-V Software Ecosystem) Project (https://riseproject.dev/)
  - ❑ **What**: Collaborative effort led by industry to accelerate the development of open-source SW for the RISC-V architecture
  - ❑ Compilers: GCC, LLVM
  - ❑ System Libraries: OpenSSL, BoringSSL, glibc, bionic, zlib, OpenBLAS, SLEEF
  - ❑ Kernel and Virtualization: Linux, KVM
  - ❑ Language Runtimes: Java, Python, Go, .NET, Android RT, Javascript
  - ❑ Linux Distro Integration: Debian, Fedora, Ubuntu, GRUB
  - ❑ Debug and Profiling: GDB, Valgrind, AddressSanitizer
  - ❑ Simulators/Emulators: QEMU
  - ❑ System Firmware: TianoCore UEFI, U-Boot, Coreboot, TF-M

Software porting is not trivial

- GPU-like acceleration
  - ❑Applications written in CUDA will need to be rewritten to make use of RVV
  - ❑Alt: develop a path to target CUDA via other interfaces such as SYCL, hand tuning will still be required.
  - ❑RVV is not a GPU!
- Extracting long vectors is very algorithm-dependent
  - ❑EPI/EUPILOT developing long vector RVV implementation (256 DP words)!
  - ❑Most existing codes are not be able to make use very long vectors
  - ❑Need to research compiler support for long vectors
  - ❑+ research algorithmic transformations to expose the required parallelism

Getting started with RISC-V

- Boards: https://www.riscfive.com/risc-v-development-boards/
- EPI SDVs are also available to be used

In the absence of HW:

- **Spike**: Functional simulator considered as golden reference for RISC-V ISA

- Quick EMUlator (**QEMU**): dynamic binary translation (host ISA-> target ISA), execution on target hardware model
  - System mode: boot OS, run application as part of OS
  - User mode: no OS, run application directly

- If performance evaluation is required:
  - **Gem5** (state-of-the-art): modular system architecture research platform to model complete systems (hardware, OS, application software). Support RVV since Dec 23 (v23.1)

Conclusions

- RISC-V + HPC has seen a lot of work and progress on all fronts:
  - EPI/EUPILOT developing RVV multicore systems + SW toolchain
  - Community working on HW, SW toolchain and ecosystem, ISA extensions
- RVV is a big step ahead, but we need:
  - More "HPC-like" boards to play with, and better performance tools
  - RISC-V HPC hardware needs to focus on stable specs (RISC-V profiles)
  - Specific challenges
    - autovectorizers to extract long vectors
    - reworked algorithms to expose more vector parallelism
    - CUDA to RISC-V conversion strategies
    - A converged/frozen spec for matrix engines

Acknowledgements

- CHART: Chalmers Heterogeneous Architectures and Runtimes Team

ePProcessor THE EUPILOT European Processor Initiative epi

SSF Vetenskapsrådet

- Master Program in High Performance Computer Systems
  - https://www.chalmers.se/en/education/find-masters-programme/high-performance-computer-systems-msc/

High-performance computer systems, MSc
120 credits

Overview

Educational area
Computer engineering

Degree
Master of Science, MSc

Language
English

Place of study
Johanneberg

Duration:
2 Years

Rate of study:
Full-time, 100%

CHALMERS
UNIVERSITY OF TECHNOLOGY

THANK YOU

CHALMERS