Evaluation of RVV-enabled COTS platforms with matrix multiplication and Exo

International workshop on RISC-V for HPC at ISC25

Francisco D. Igual
Universidad Complutense de Madrid (Spain)

Adrián Castelló
Enrique S. Quintana-Ortí
Universitat Politècnica de València (Spain)

Héctor Martínez
Universidad de Córdoba (Spain)

Sandra Catalán
Universitat Jaume I de Castelló (Spain)

Agenda

1. Introduction
2. GEMM basics
3. Automatic GEMM generation with EXO
4. Experimental evaluation
5. Conclusions

Introduction

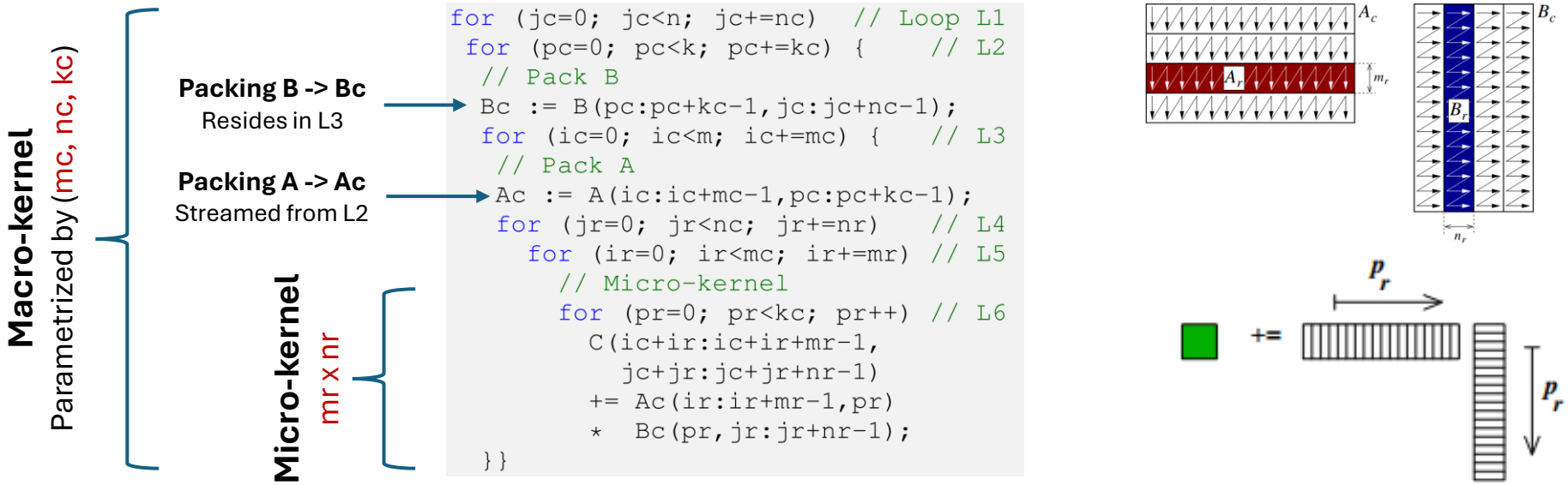
- RISC-V: open ISA, extension-based
- RVV (RISC-V “v” extension): vector extension for RISC-V
  - Standardized (v. 1.0) in september 2021
  - Vector-length agnostic
    - SIMD
    - Long-vector vector units
    - Matrix extensions (IME proposal, sharing RVV state/registers)
  - Toolchains, simulators and implementations already available
    - COTS (Commercial-off-the-shelf) platforms in the market

Introduction (II)

- RVV is vector-length agnostic by design
  - Portability across architectures, even with different VLEN
- Generating new (optimized) code for a new architecture is still a challenge
  - Micro-architectural details
  - Different cache characteristics
  - RVV version support (platforms only compliant with RVV 0.7.1)
- A prototypical example: General Matrix-Matrix Multiplication (GEMM)
- Automatic code generation to the rescue
  - Alternatives: TVM, MLIR, Halide, EXO

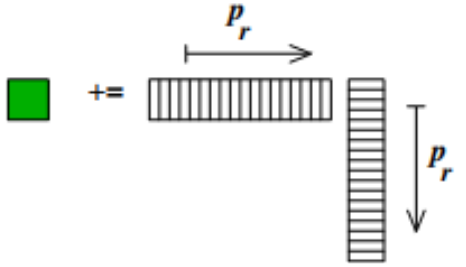
(High-performance) GEMM in a nutshell

- GEMM (General Matrix-Matrix Multiplication)
  - Typical benchmark to showcase the peak (artithmetic) performance of a new architecture
  - Base of many higher-level libraries and applications
- High-performance libraries (e.g. BLIS, OpenBLAS) follow GotoBLAS approach
  - Macro-kernel (five loops) + Packing routines + Micro-kernel
    - Loop strides (mc, nc, kc) + packing routines aim at exploiting cache hierarchy and allow unit-stride loads/stores
  - Micro-kernel:
    - Typically developed using intrinsics and/or assembly code
    - Tecnhiques to maximize register use and leverage arithmetic capabilities of the core. Parametrized by (mr, nr)



GEMM micro-kernel flavors

- Different strategies to implement the outer-product
  - Loop L6 (micro-kernel)




```
1  .L3:          # BCAST-AB
2    addi        t1,a5,4
3    addi        a7,a5,8
4    addi        a0,a5,12
5    addi        a2,a2,1
6    vle32.v     v24,0(a3)
7    vlse32.v    v8,0(a5),zero
8    vlse32.v    v31,0(t1),zero
9    vlse32.v    v30,0(a7),zero
10   vlse32.v    v29,0(a0),zero
11   add         a3,a3,a4
12   add         a5,a5,a6
13   vfmacc.vv   v25,v24,v8
14   vfmacc.vv   v28,v24,v31
15   vfmacc.vv   v27,v24,v30
16   vfmacc.vv   v26,v24,v29
17   bne         a1,a2,.L3
```

BCAST-AB (-BA)

```
1  .L3:          # GATHER-AB
2    addi        a2,a2,1
3    vle32.v     v24,0(a3)
4    vle32.v     v25,0(a5)
5    add         a3,a3,a4
6    vrgather.vi v9,v25,0
7    vrgather.vi v8,v25,1
8    vrgather.vi v31,v25,2
9    vrgather.vi v30,v25,3
10   add         a5,a5,a6
11   vfmacc.vv   v26,v24,v9
12   vfmacc.vv   v29,v24,v8
13   vfmacc.vv   v28,v24,v31
14   vfmacc.vv   v27,v24,v30
15   bne         a1,a2,.L3
```

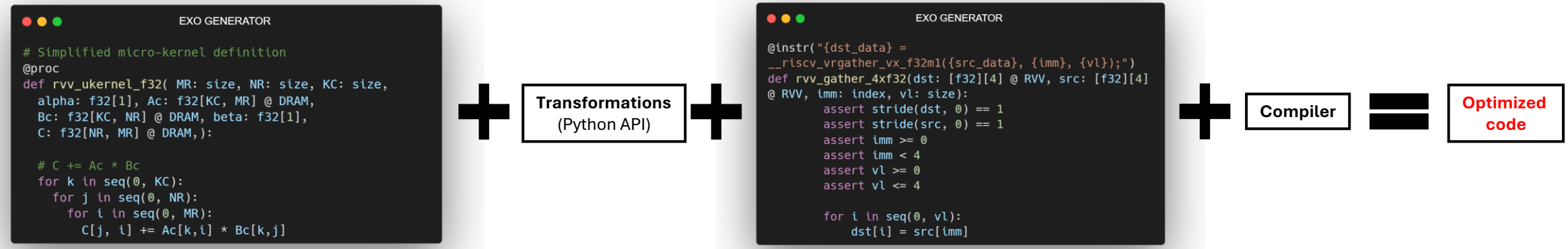
GATHER-AB (-BA)

```
1  .L3:          # DIRECT
2    flw         fa2,0(a5)
3    flw         fa3,4(a5)
4    flw         fa4,8(a5)
5    flw         fa5,12(a5)
6    addi        a2,a2,1
7    vle32.v     v24,0(a3)
8    add         a5,a5,a6
9    add         a3,a3,a4
10   vfmacc.vf   v25,fa2,v24
11   vfmacc.vf   v28,fa3,v24
12   vfmacc.vf   v27,fa4,v24
13   vfmacc.vf   v26,fa5,v24
14   bne         a1,a2,.L3
```

DIRECT

EXO

- Domain Specific Language for low-level Performance Engineers.
- Transforms simple Python code into high-performance C code.
- UC Berkeley (https://github.com/exo-lang/exo)
- Support for different vector ISAs (AVX, AVX512, Neon, SVE, ~~RVV~~)
- HPC experience required!!!



EXO generator for GEMM micro-kernel

EXO GENERATOR (I)

```
# Simplified micro-kernel definition
@proc
def rvv_ukernel_f32( MR: size, NR: size, KC: size,
  alpha: f32[1], Ac: f32[KC, MR] @ DRAM,
  Bc: f32[KC, NR] @ DRAM, beta: f32[1],
  C: f32[NR, MR] @ DRAM,):

  # C += Ac * Bc
  for k in seq(0, KC):
    for j in seq(0, NR):
      for i in seq(0, MR):
        C[j, i] += Ac[k,i] * Bc[k,j]

# BLOCK_1
p = rename(rvv_ukernel_f32, f"rvv_ukernel_8x6_f32")
p = p.partial_eval(MR=8,NR=6)

# BLOCK_2
p = divide_loop(p,'i', vl, ['it','itt'])
p = divide_loop(p,'j', vl, ['jt','jtt'])

# BLOCK_3
#  1) Map C buffer to vectorial register C_reg
Cp = 'C[j, 4 * it + itt]'
p = stage_mem(p, 'C[_] += _', Cp, 'C_reg')

#  2) Build a 3D structure of C_reg
p = expand_dim(p, 'C_reg', vl, 'itt', ...)
p = expand_dim(p, 'C_reg', MR//vl, 'it', ...)
p = expand_dim(p, 'C_reg', NR, 'j', ...)

#  3) Move the register declaration to the top
p = lift_alloc(p, 'C_reg', n_lifts=4)

#  4) Extract the C load and store from the k-loop
p = autofission(p, p.find('C_reg[_] = _').after(),
                n_lifts=4)
p = autofission(p, p.find('C[_] = _').before(),
                n_lifts=4)

#  5) Replace the indicated loops by RVV intrinsics
p = replace(p, 'for itt in _: _', rvv_vld_4xf32)
p = replace(p, 'for itt in _: _', rvv_vst_4xf32)

#  6) Set the C_reg memory to RVV
p = set_memory(p, 'C_reg', RVV)
```

Starting point: simplified EXO code for GEMM micro-kernel

Step1 . Specify kernel for mr, nr

Resulting EXO generated code:

EXO GENERATOR

```
# RESULTING EXO GENERATED CODE
def rvv_ukernel_8x6_f32( KC:size, alpha:f32[1] @DRAM,
  Ac: f32[KC, 8] @DRAM, Bc: f32[KC, 6] @DRAM,
  beta: f32[1] @DRAM, C: f32[6, 8] @DRAM):

  # C += Ac * Bc
  for k in seq(0, KC):
    for j in seq(0, 6):
      for i in seq(0, 8):
        C[j, i] += Ac[k, i] * Bc[k, j]
```

EXO generator for GEMM micro-kernel

EXO GENERATOR (I)

```python
# Simplified micro-kernel definition
@proc
def rvv_ukernel_f32( MR: size, NR: size, KC: size,
  alpha: f32[1], Ac: f32[KC, MR] @ DRAM,
  Bc: f32[KC, NR] @ DRAM, beta: f32[1],
  C: f32[NR, MR] @ DRAM,):

  # C += Ac * Bc
  for k in seq(0, KC):
    for j in seq(0, NR):
      for i in seq(0, MR):
        C[j, i] += Ac[k,i] * Bc[k,j]

# BLOCK_1
p = rename(rvv_ukernel_f32, f"rvv_ukernel_8x6_f32")
p = p.partial_eval(MR=8,NR=6)

# BLOCK_2
p = divide_loop(p,'i', vl, ['it','itt'])
p = divide_loop(p,'j', vl, ['jt','jtt'])

# BLOCK_3
#  1) Map C buffer to vectorial register C_reg
Cp = 'C[j, 4 * it + itt]'
p = stage_mem(p, 'C[_] += _', Cp, 'C_reg')

#  2) Build a 3D structure of C_reg
p = expand_dim(p, 'C_reg', vl, 'itt', ...)
p = expand_dim(p, 'C_reg', MR//vl, 'it', ...)
p = expand_dim(p, 'C_reg', NR, 'j', ...)

#  3) Move the register declaration to the top
p = lift_alloc(p, 'C_reg', n_lifts=4)

#  4) Extract the C load and store from the k-loop
p = autofission(p, p.find('C_reg[_] = _').after(),
                n_lifts=4)
p = autofission(p, p.find('C[_] = _').before(),
                n_lifts=4)

#  5) Replace the indicated loops by RVV intrinsics
p = replace(p, 'for itt in _: _', rvv_vld_4xf32)
p = replace(p, 'for itt in _: _', rvv_vst_4xf32)

#  6) Set the C_reg memory to RVV
p = set_memory(p, 'C_reg', RVV)
```

Step2 . Loop structure

- Adapt loop to VL of target architecture
- Loop i (traverses mr) Split to fit elements per vector
  - E.g. FP32 elements per vector register
- Loop i divided by vl (4)
  - Loops it and itt created

Resulting EXO generated code:

EXO GENERATOR

```python
# C += Ac * Bc
for k in seq(0, KC):
  for j in seq(0, 6):
    for it in seq(0, 2):
      for itt in seq(0, 4):
        C[j, itt + 4 * it] +=
          Ac[k, itt + 4 * it] * Bc[k, j]
```

EXO generator for GEMM micro-kernel

EXO GENERATOR (I)

```python
# Simplified micro-kernel definition
@proc
def rvv_ukernel_f32( MR: size, NR: size, KC: size,
  alpha: f32[1], Ac: f32[KC, MR] @ DRAM,
  Bc: f32[KC, NR] @ DRAM, beta: f32[1],
  C: f32[NR, MR] @ DRAM,):

  # C += Ac * Bc
  for k in seq(0, KC):
    for j in seq(0, NR):
      for i in seq(0, MR):
        C[j, i] += Ac[k,i] * Bc[k,j]

# BLOCK_1
p = rename(rvv_ukernel_f32, f"rvv_ukernel_8x6_f32")
p = p.partial_eval(MR=8,NR=6)

# BLOCK_2
p = divide_loop(p,'i', vl, ['it','itt'])
p = divide_loop(p,'j', vl, ['jt','jtt'])

# BLOCK_3
#  1) Map C buffer to vectorial register C_reg
Cp = 'C[j, 4 * it + itt]'
p = stage_mem(p, 'C[_] += _', Cp, 'C_reg')

#  2) Build a 3D structure of C_reg
p = expand_dim(p, 'C_reg', vl, 'itt', ...)
p = expand_dim(p, 'C_reg', MR//vl, 'it', ...)
p = expand_dim(p, 'C_reg', NR, 'j', ...)

#  3) Move the register declaration to the top
p = lift_alloc(p, 'C_reg', n_lifts=4)

#  4) Extract the C load and store from the k-loop
p = autofission(p, p.find('C_reg[_] = _').after(),
                n_lifts=4)
p = autofission(p, p.find('C[_] = _').before(),
                n_lifts=4)

#  5) Replace the indicated loops by RVV intrinsics
p = replace(p, 'for itt in _: _', rvv_vld_4xf32)
p = replace(p, 'for itt in _: _', rvv_vst_4xf32)

#  6) Set the C_reg memory to RVV
p = set_memory(p, 'C_reg', RVV)
```

Step3 . C matrix

- Binds C (mr x nr) to vector registers
- Sets register type to RVV registers
- Replaces vectorized loads/stores by vector intrinsics (RVV)

Resulting EXO generated code:

EXO GENERATOR

```python
def rvv_ukernel_8x6_f32( ... )

  # Registers for C
  C_reg: f32[6, 2, 4] @ RVV

  # Load C to registers
  for j in seq(0, 6):
    for it in seq(0, 2):
      rvv_vld_4xf32(
        C_reg[j, it, 0:4],
        C[j, 4 * it:4 * it + 4], 4 )

  # C += Ac * Bc omitted

  # Store C from registers
  for j in seq(0, 6):
    for it in seq(0, 2):
      rvv_vst_4xf32(
          C[j, 4 * it:4 * it + 4],
          C_reg[j, it, 0:4], 4 )
```

EXO generator for GEMM micro-kernel

EXO GENERATOR (II)

```
# BLOCK_4
#  1) Map Ac buffer to vectorial register A_reg
p = bind_expr(p, 'Ac[_]','A_reg')

#  2) Build a 2D structure of A_reg
p = expand_dim(p, 'A_reg' , vl, 'itt', ...)
p = expand_dim(p, 'A_reg', MR//vl, 'it', ...)

#  3) Move the register declaration to the top
p = lift_alloc(p, 'A_reg', n_lifts=4)

#  4) Move the Ac load to the k-loop
p = autofission(p, p.find('A_reg[_] = _').after(),
                n_lifts=3)

#  5) Replace the itt loop by RVV intrinsics
p = replace(p, 'for itt in _: _', rvv_vld_4xf32)

#  6) Set the A_reg memory to RVV
p = set_memory(p, 'A_reg', RVV)

# BLOCK_5
#  1) Divide the j loop by a factor vl
p = divide_loop(p,'j', vl, ['jt','jtt'])

#  2) Map Bc buffer to vectorial register Btmp
p = bind_expr(p, 'Bc[_]','Btmp')

#  3) Map Btmp to vectorial register B_reg
p = bind_expr(p, 'Btmp','B_reg')

#  4) Build a 2D structure of B_reg and Btmp
p = expand_dim(p, 'Btmp' , vl, 'jtt', ...)
p = expand_dim(p, 'B_reg', vl, 'itt', ...)
p = expand_dim(p, 'B_reg', NR, 'jt * 4 + jtt', ...)

#  5) Move the register declaration to the top
p = lift_alloc(p, 'B_reg', n_lifts=5)
p = lift_alloc(p, 'Btmp', n_lifts=5)

#  6) Move the Bc load to the k-loop
p = autofission(p, p.find('B_reg[_] = _').after(),
                n_lifts=4)
p = autofission(p, p.find('Btmp[_] = _').after(),
                n_lifts=2)

#  7) Replace the itt loop by RVV intrinsics
p = replace(p, 'for jtt in _: _', rvv_vld_4xf32)
p = replace(p, 'for itt in _: _', rvv_gather_4xf32)

#  8) Set the A_reg memory to RVV
p = set_memory(p, 'B_reg', RVV)
p = set_memory(p, 'Btmp', RVV)

# BLOCK_6
p = unroll_loop(p,'it')
p = unroll_loop(p,'jt')

# BLOCK_7
p = unroll_buffer(p, 'A_reg', 0)
```

Step4 . Operand Ac

- Schedules vectorized loads for Ac
- Maps submatrix Ac to vector (RVV) registers
- Replaces itt loop with RVV vector load

Resulting EXO generated code:

EXO GENERATOR

```
def rvv_ukernel_8x6_f32( KC:size, alpha:f32[1] @DRAM,
  Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @DRAM,
  beta: f32[1] @ DRAM, C: f32[12, 8] @DRAM
  ):

  # C load omitted for brevity

  # Registers for Ac
  A_reg: R[2, 4] @ RVV
  for k in seq(0, KC):
    # Load Ac to registers
    for it in seq(0, 2):
      rvv_vld_4xf32(
        A_reg[it, 0:4],
        Ac[k, 4 * it:4 + 4 * it], 4
      )
    # Computation omitted

  # C store omitted for brevity
```

EXO generator for GEMM micro-kernel

EXO GENERATOR (II)

```
# BLOCK_4
#  1) Map Ac buffer to vectorial register A_reg
p = bind_expr(p, 'Ac[_]','A_reg')

#  2) Build a 2D structure of A_reg
p = expand_dim(p, 'A_reg' , vl, 'itt', ...)
p = expand_dim(p, 'A_reg', MR//vl, 'it', ...)

#  3) Move the register declaration to the top
p = lift_alloc(p, 'A_reg', n_lifts=4)

#  4) Move the Ac load to the k-loop
p = autofission(p, p.find('A_reg[_] = _').after(),
                n_lifts=3)

#  5) Replace the itt loop by RVV intrinsics
p = replace(p, 'for itt in _: _', rvv_vld_4xf32)

#  6) Set the A_reg memory to RVV
p = set_memory(p, 'A_reg', RVV)

# BLOCK_5
#  1) Divide the j loop by a factor vl
p = divide_loop(p,'j', vl, ['jt','jtt'])

#  2) Map Bc buffer to vectorial register Btmp
p = bind_expr(p, 'Bc[_]','Btmp')

#  3) Map Btmp to vectorial register B_reg
p = bind_expr(p, 'Btmp','B_reg')

#  4) Build a 2D structure of B_reg and Btmp
p = expand_dim(p, 'Btmp' , vl, 'jtt', ...)
p = expand_dim(p, 'B_reg', vl, 'itt', ...)
p = expand_dim(p, 'B_reg', NR, 'jt * 4 + jtt', ...)

#  5) Move the register declaration to the top
p = lift_alloc(p, 'B_reg', n_lifts=5)
p = lift_alloc(p, 'Btmp', n_lifts=5)

#  6) Move the Bc load to the k-loop
p = autofission(p, p.find('B_reg[_] = _').after(),
                n_lifts=4)
p = autofission(p, p.find('Btmp[_] = _').after(),
                n_lifts=2)

#  7) Replace the itt loop by RVV intrinsics
p = replace(p, 'for jtt in _: _', rvv_vld_4xf32)
p = replace(p, 'for itt in _: _', rvv_gather_4xf32)

#  8) Set the A_reg memory to RVV
p = set_memory(p, 'B_reg', RVV)
p = set_memory(p, 'Btmp', RVV)

# BLOCK_6
p = unroll_loop(p,'it')
p = unroll_loop(p,'jt')

# BLOCK_7
p = unroll_buffer(p, 'A_reg', 0)
```

Step4 . Operand Bc

- Schedules vectorized loads for Bc
- For GATHER-AB variant:
  - Divide loop j by factor VL (create jt, jtt)
  - Maps vector registers for Bc
  - Replaces jtt by vector loads (RVV)
  - Replaces itt by gather intrinsic (RVV)

Resulting EXO generated code (BCAST-AB):

EXO GENERATOR

```
def rvv_ukernel_8x6_f32( ... )

  # C load omitted for brevity

  A_reg: R[2, 4] @ RVV
  Btmp: R[1, 4] @ RVV
  B_reg: R[6, 4] @ RVV
  for k in seq(0, KC):
   for it in seq(0, 2):
      rvv_vld_4xf32(
        A_reg[it, 0:4],
        Ac[k, 4 * it:4 + 4 * it], 4 )
    # Load Bc to registers
    for jt in seq(0, 1):
      rvv_vld_4xf32(
        B_tmp[0:4],
        Bc[k, 4 * jt:4 + 4 * jt], 4 )
      for jtt in seq(0, 4):
        rvv_gather_4xf32(
          B_reg[jtt + 4 * jt, 0:4],
          B_tmp[0:4], jtt, 4 )
    # Tail case omitted
    # Computation omitted
  # C store omitted for brevity
```

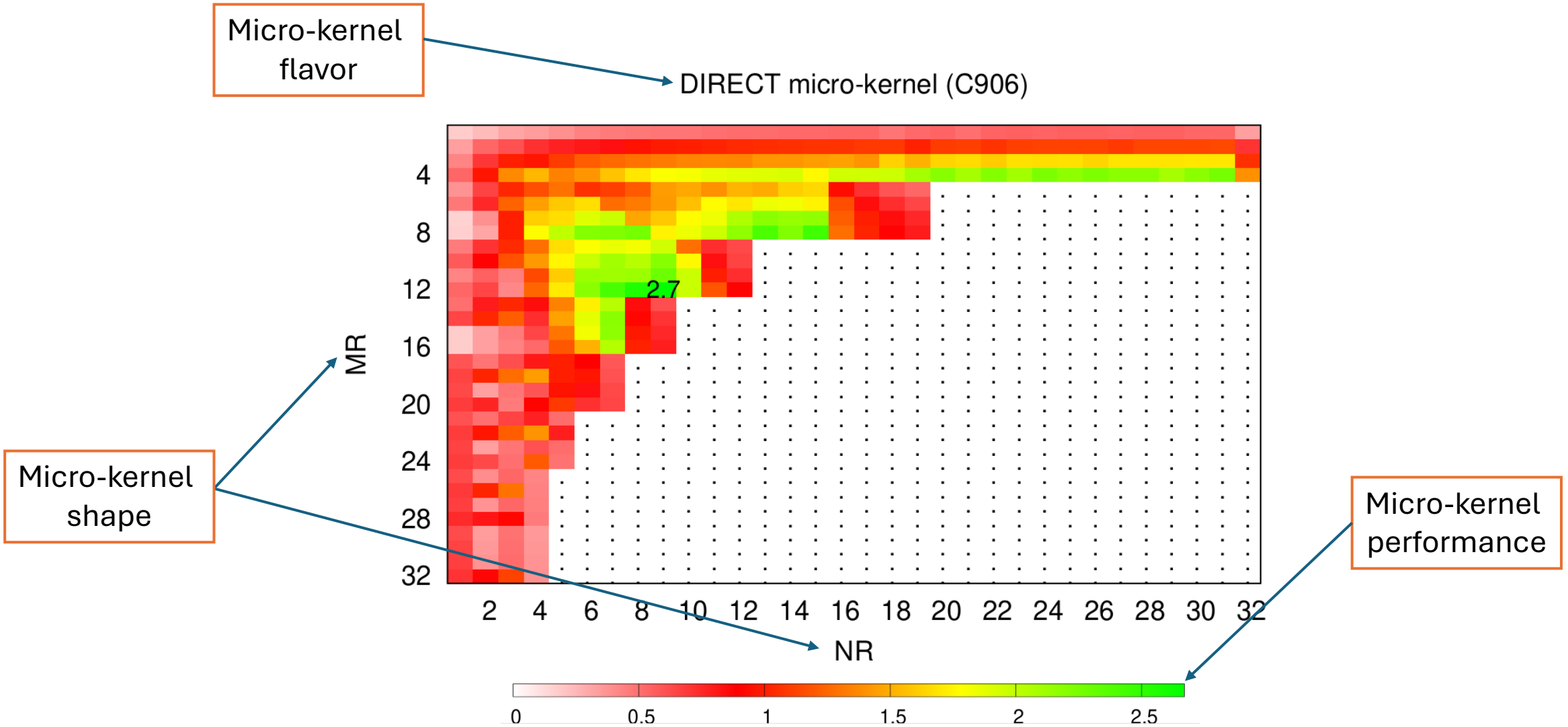
Experimental setup (I)

- Four state-of-the-art COTS platforms with RVV support

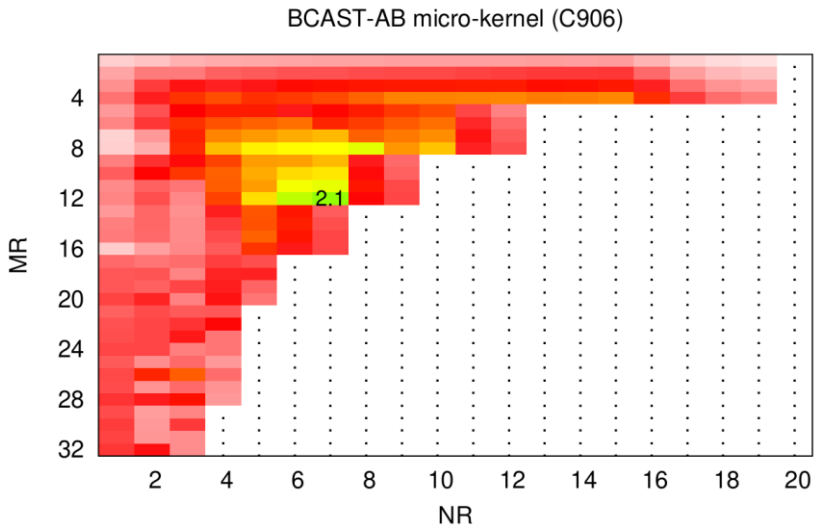
Name	SBC	Processor	Freq. (GHz)	#Cores	RVV version	Vector units	Vector length
C906	LicheeRV	XuanTie C906	1.00	1	0.7.1	1	128
C908	CanMV-K230	XuanTie C908	1.60	1	1.0	1	128
C910	LicheePi 4a	XuanTie C910	1.85	4	0.7.1	2	128
K1	BananaPi F3	SpaceMiT K1	1.60	8	1.0	1	256

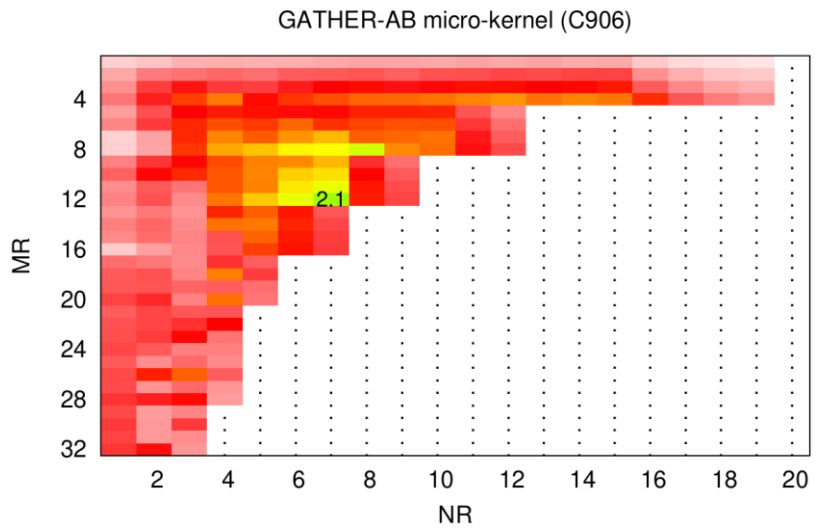
- EXO generated codes compiled via GCC
- Single-precision (FP32). Similar qualitative results for FP64
- Goals:
  1. Extract insights on optimal micro-kernel flavor and shape for each architecture
  2. Differences in micro-kernel flavor and shape across architectures
  3. Comparative performance analysis of GEMM macro-kernel with optimal micro-kernel and CCPs

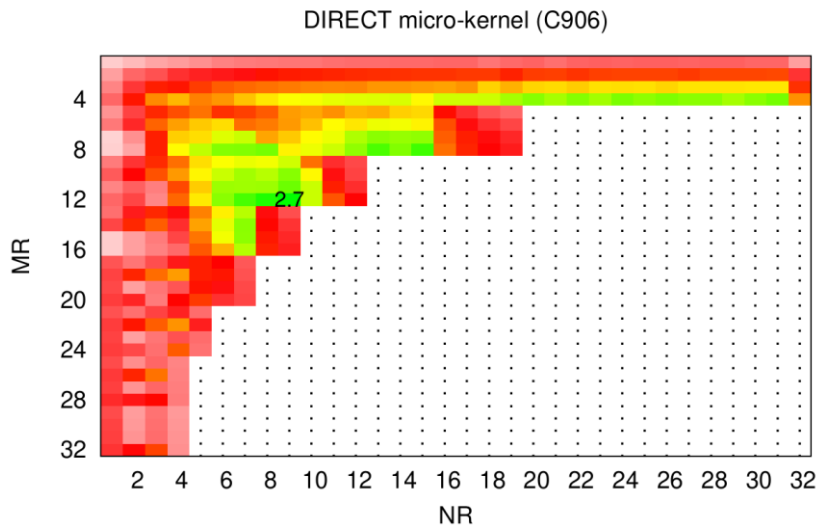
Performance heatmaps

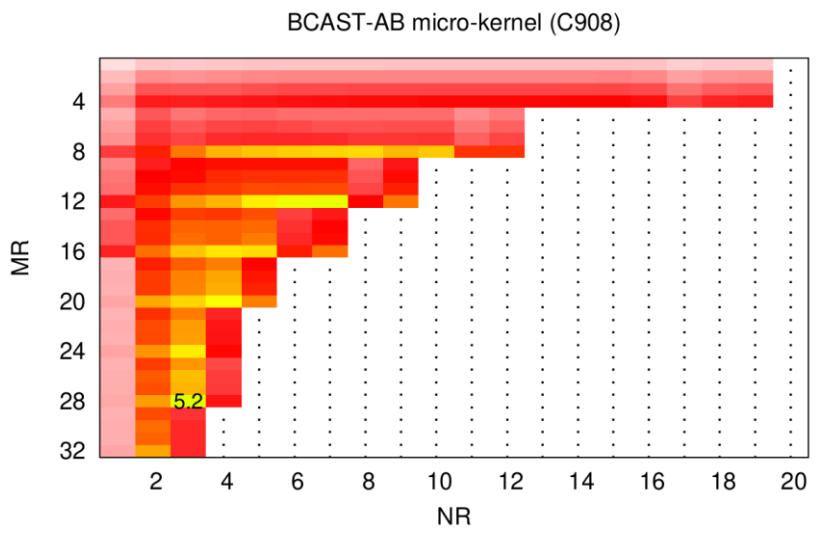


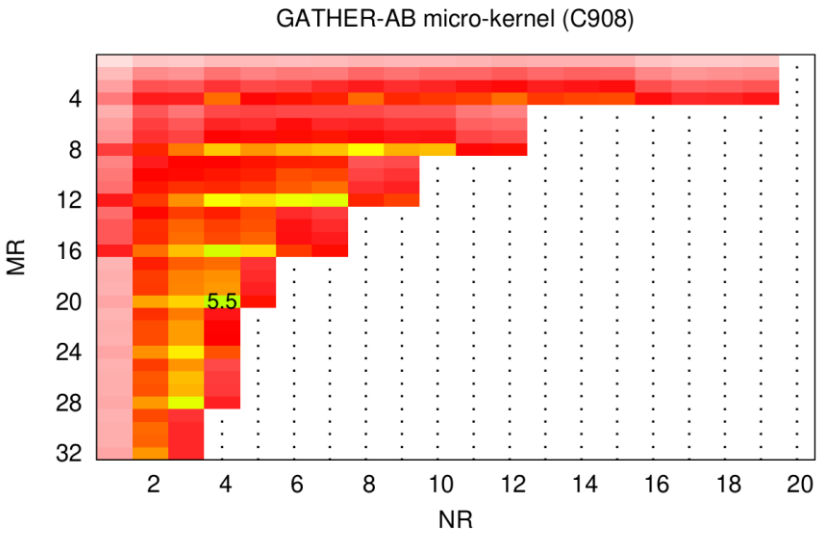
Best ukernel flavor for an architecture (C906 vs. C908)

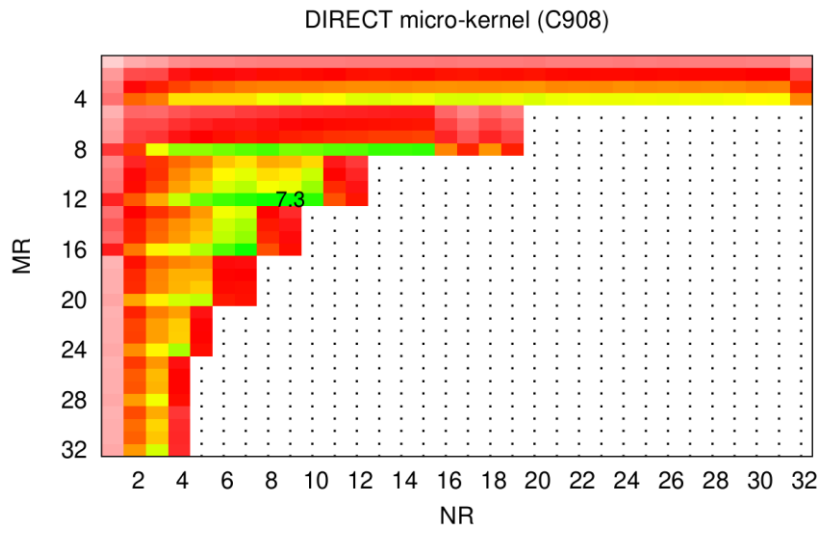




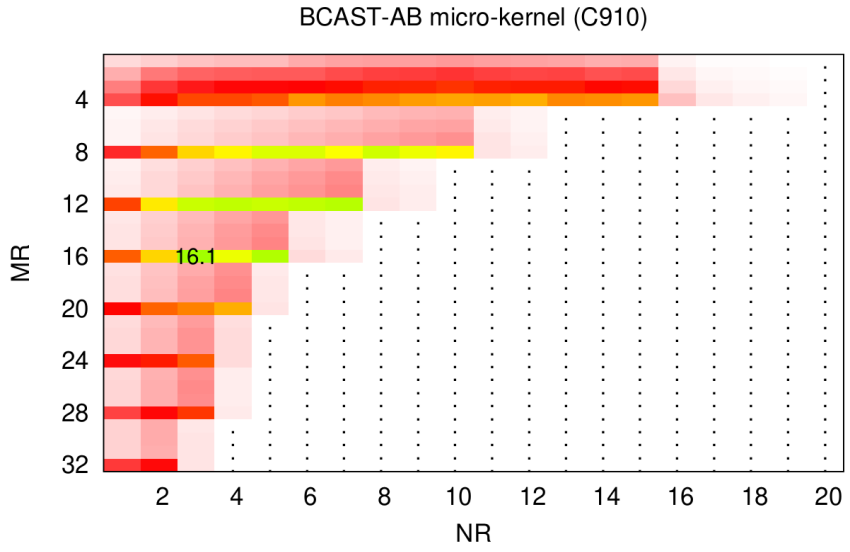


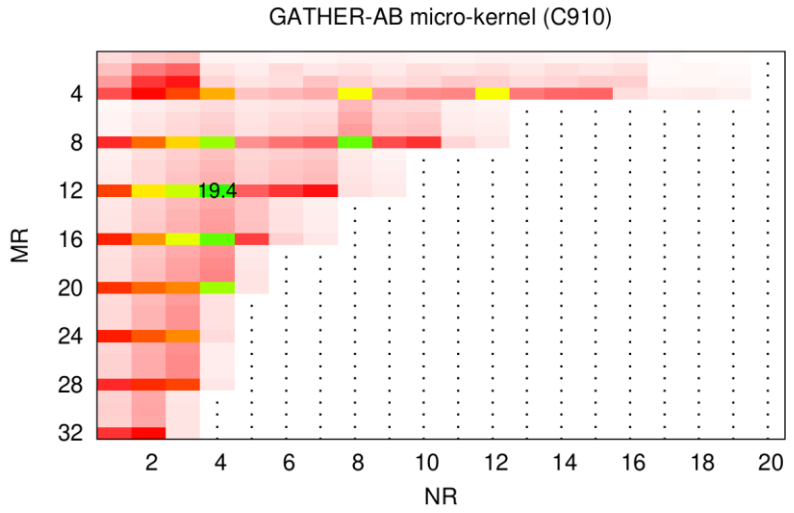


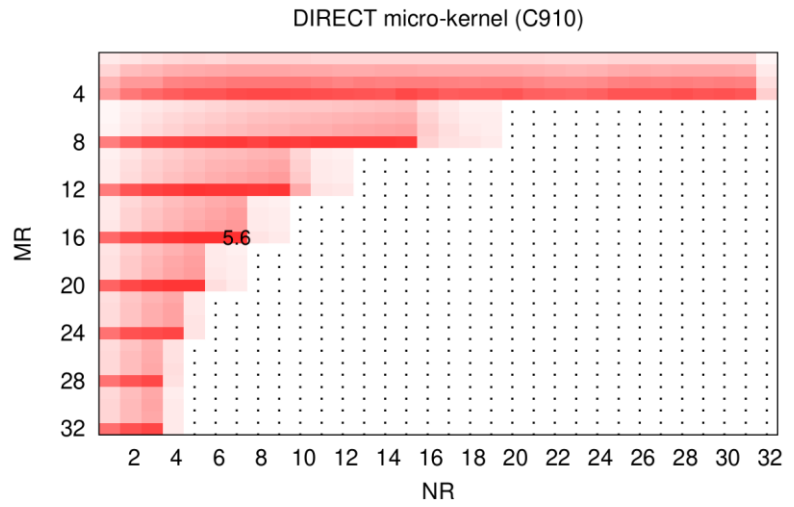


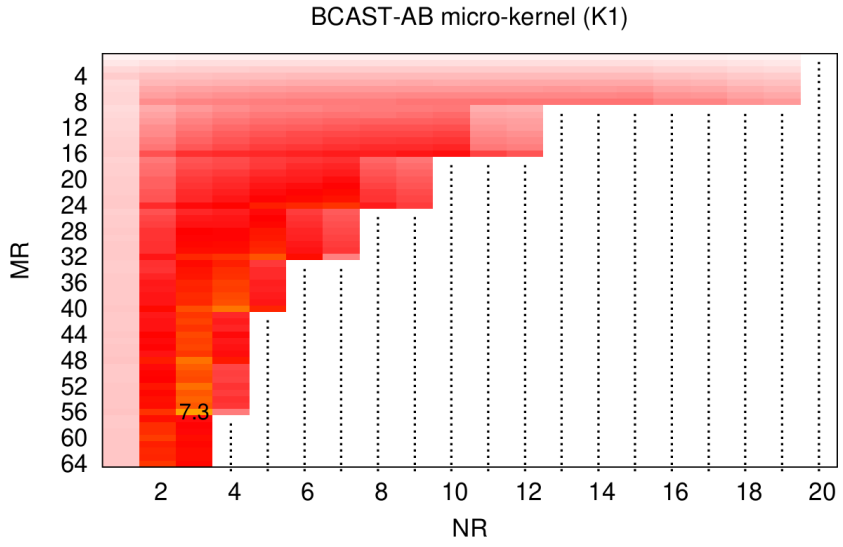


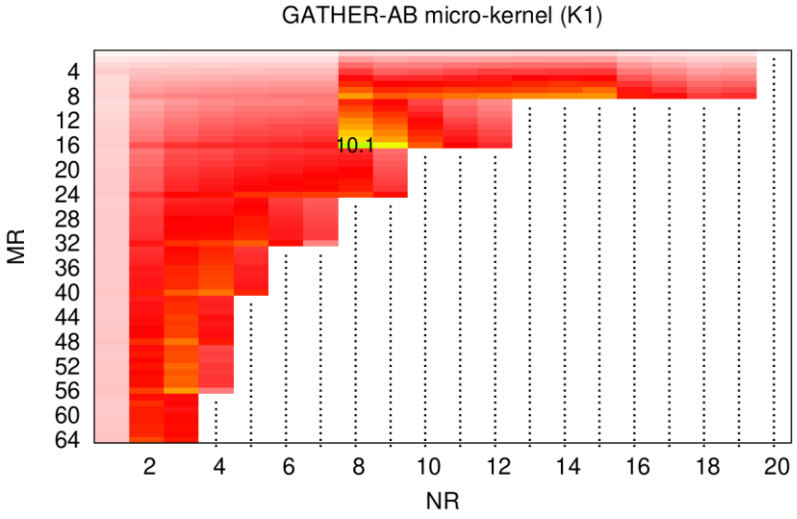
Best ukernel flavor for an architecture (C910 vs. K1)

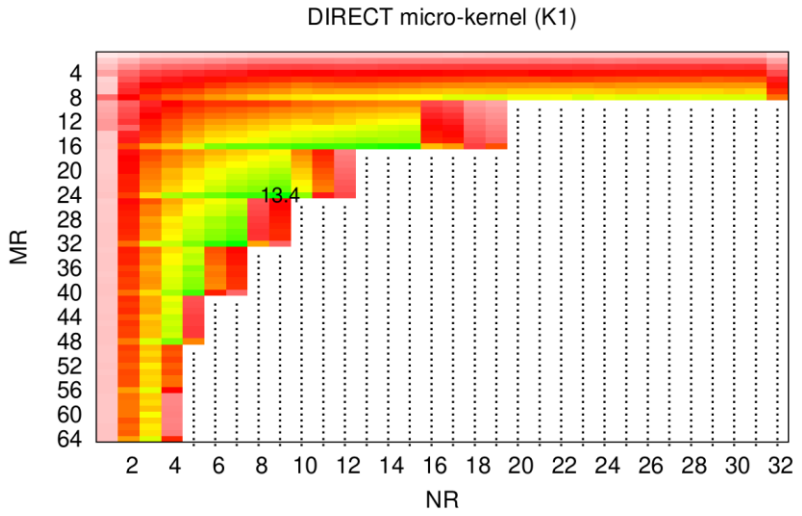




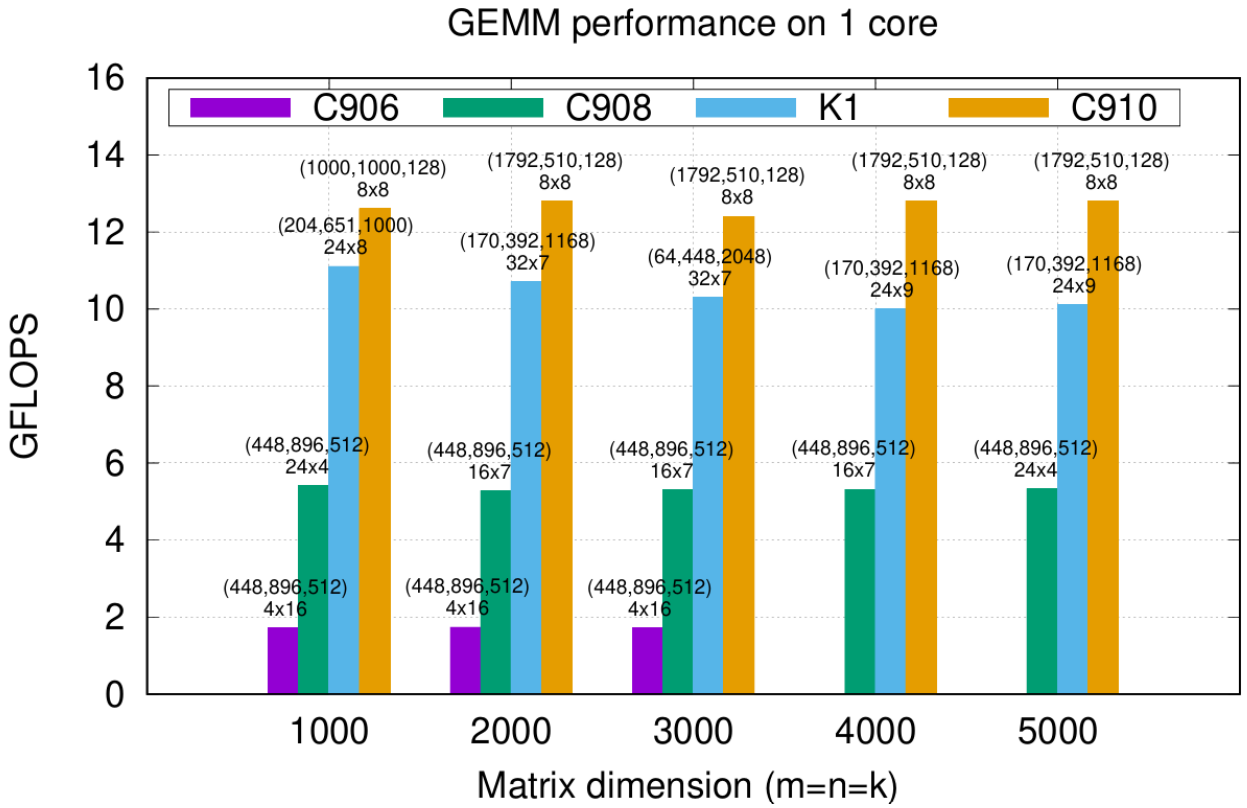


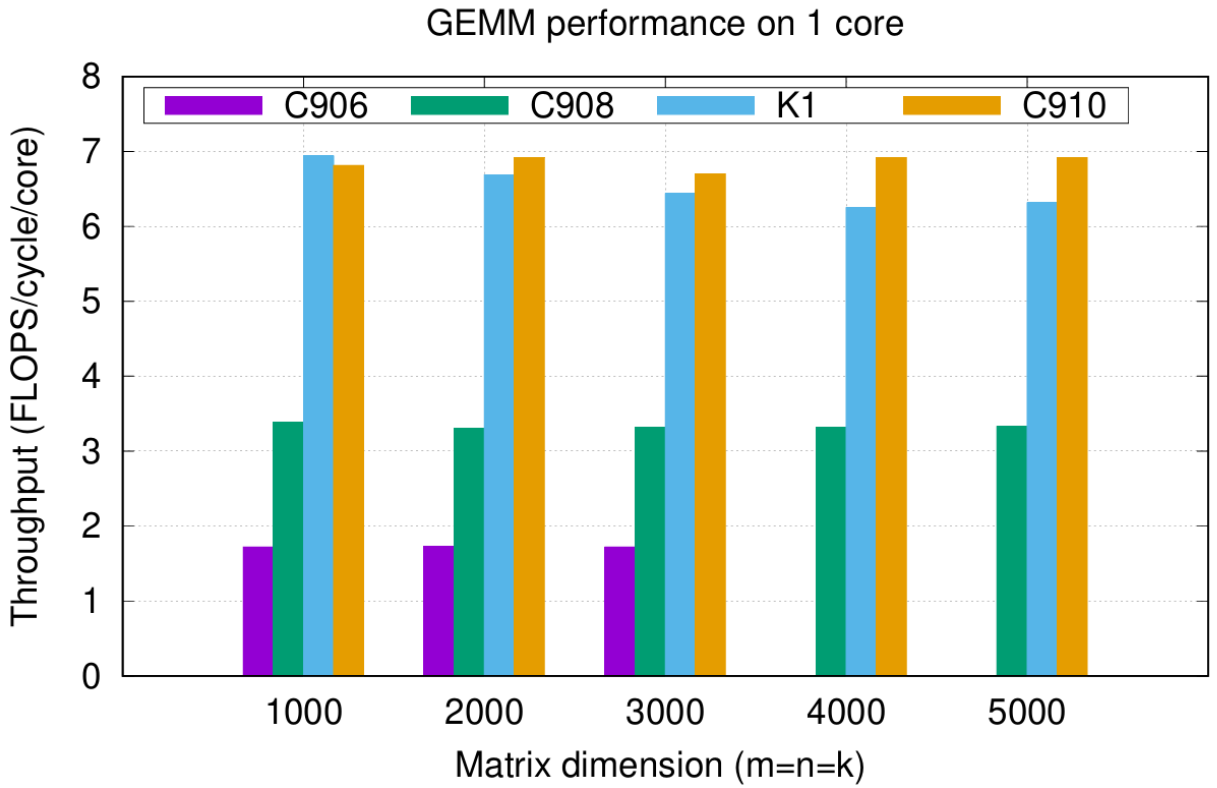




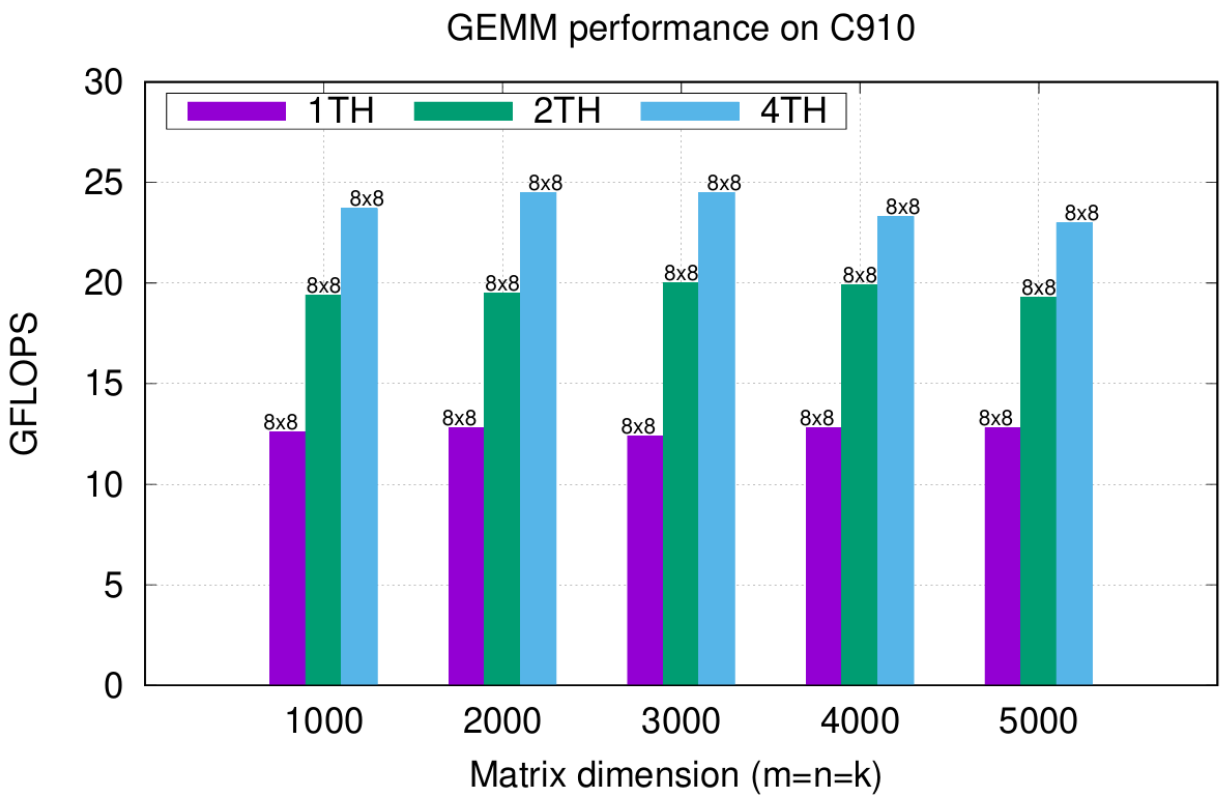


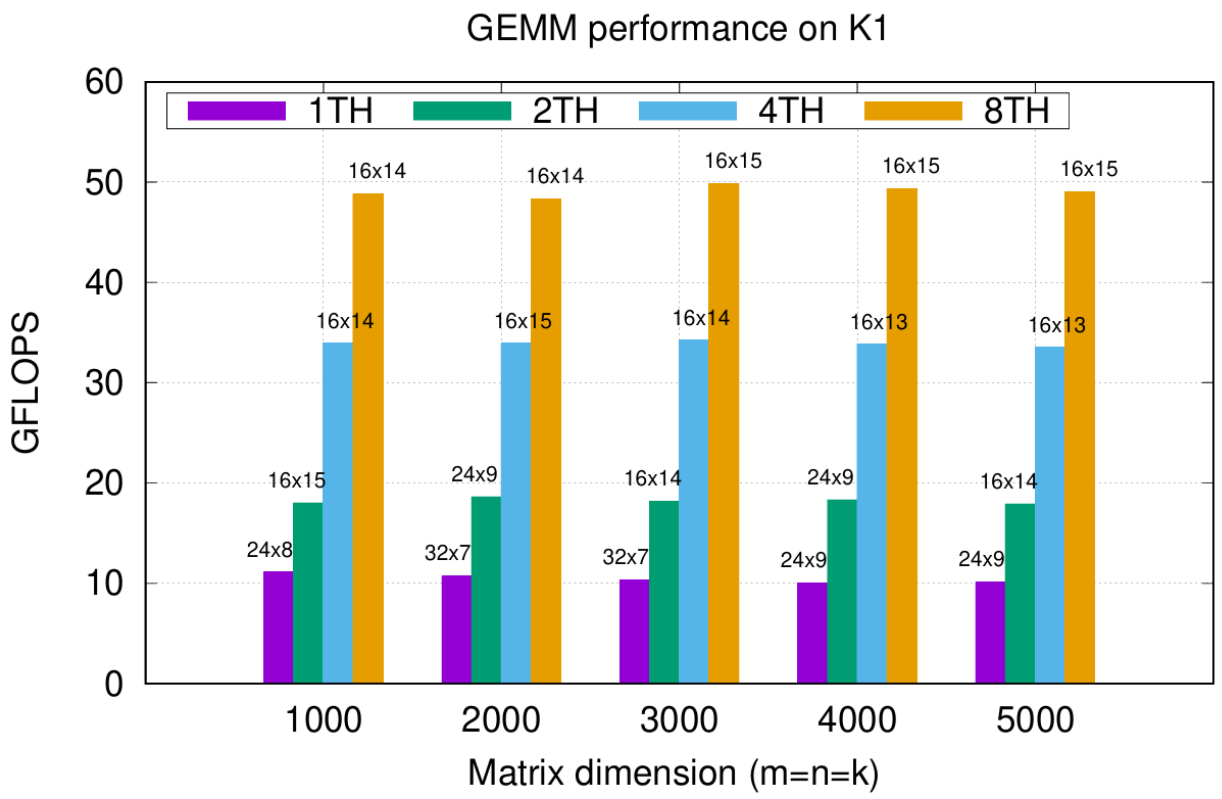
Comparative GEMM performance (1 core)



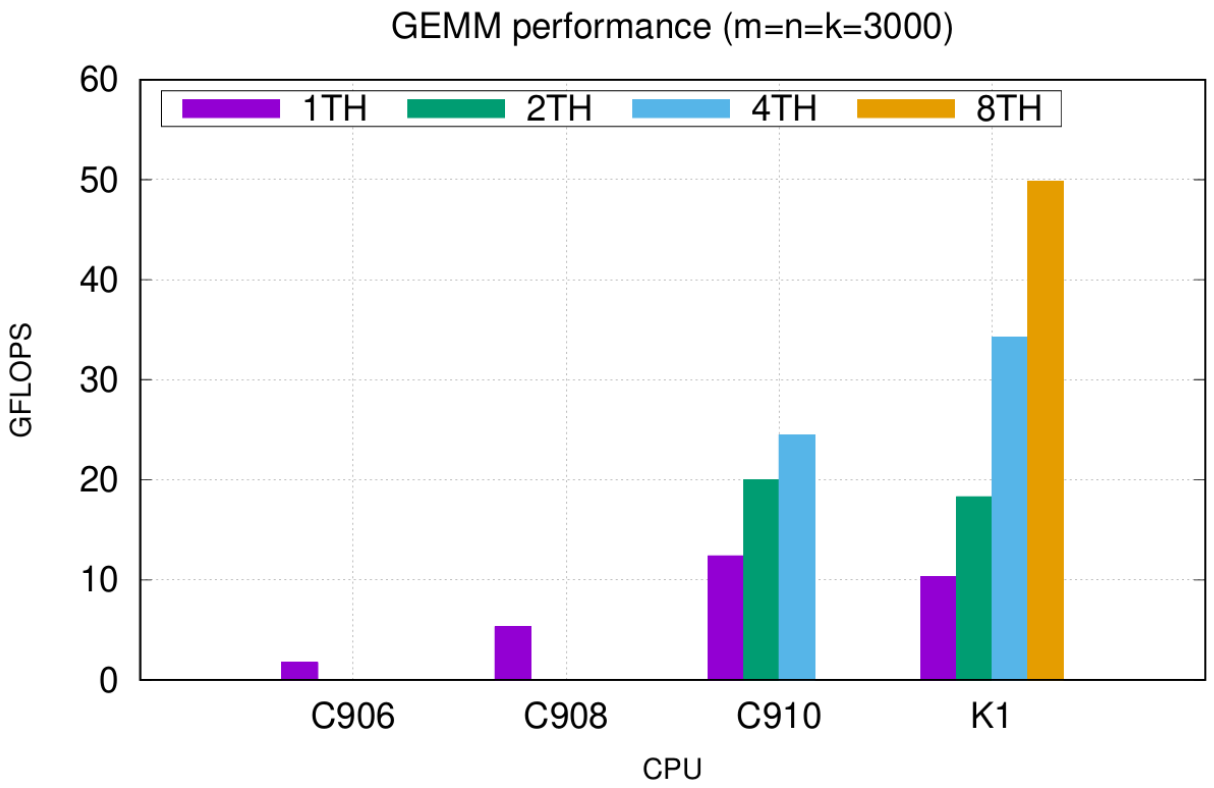


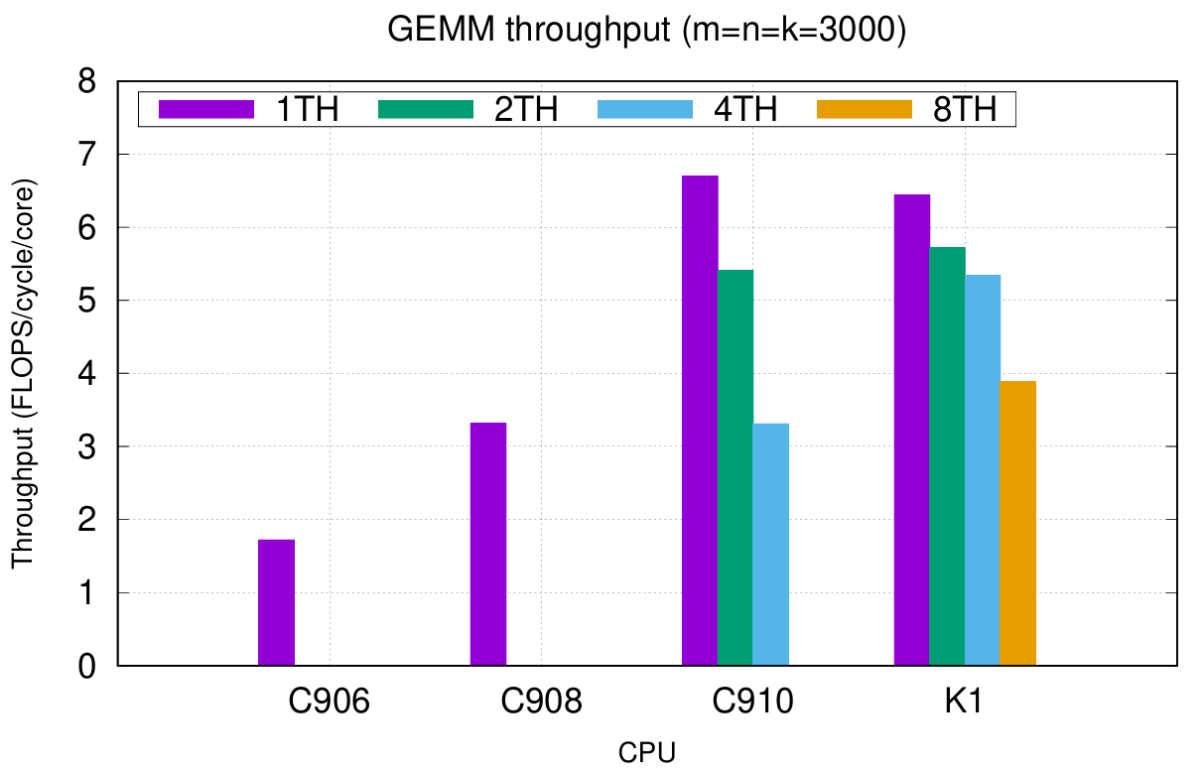
Comparative GEMM performance (multi-core)





Overall comparison (performance and throughput)





Conclusions

- Leveraged RVV backend and support for RVV
  - VL agnostic
  - Complete library of instructions-to-intrinsics mappings
- Evaluated on 4 different COTS platforms
  - Significant differences between flavors and micro-kernel shapes
  - Comparative performance and micro-kernel selection for GEMM macro-kernel
- Code availability and RVV backend in EXO

GENERATOR

GEMM BENCHMARKS

EXO

Evaluation of RVV-enabled COTS platforms with matrix multiplication and Exo

International workshop on RISC-V for HPC at ISC25

Francisco D. Igual
Universidad Complutense de Madrid (Spain)

Adrián Castelló
Enrique S. Quintana-Ortí
Universitat Politècnica de València (Spain)

Héctor Martínez
Universidad de Córdoba (Spain)

Sandra Catalán
Universitat Jaume I de Castelló (Spain)