Automatic Generation of Micro-kernels for Performance Portability of Matrix Multiplication on RISC-V Vector Processors

Second International workshop on RISC-V for HPC

Francisco D. Igual

Luis Piñuel

Universidad Complutense de Madrid

Héctor Martínez

Universidad de Córdoba

Sandra Catalán

Universitat Jaume I de Castelló

Adrián Castelló

Enrique S. Quintana-Ortí

Universitat Politècnica de València

Motivation

- High-performance BLAS implementations rely on simple micro-kernels
  - Adapted to the underlying architecture
  - Hand-written in assembly/intrinsics
  - Typically well-structured, semi-automatic development
- Automatic generation of GEMM micro-kernels for RVV
  - Basic building block for a complete Level-3 BLAS
- First experiences with **C910**/C906
  - Both supporting RVV 0.7.1
  - Necessary optimizations to improve performance vs. existing libraries (e.g. OpenBLAS)

Outline

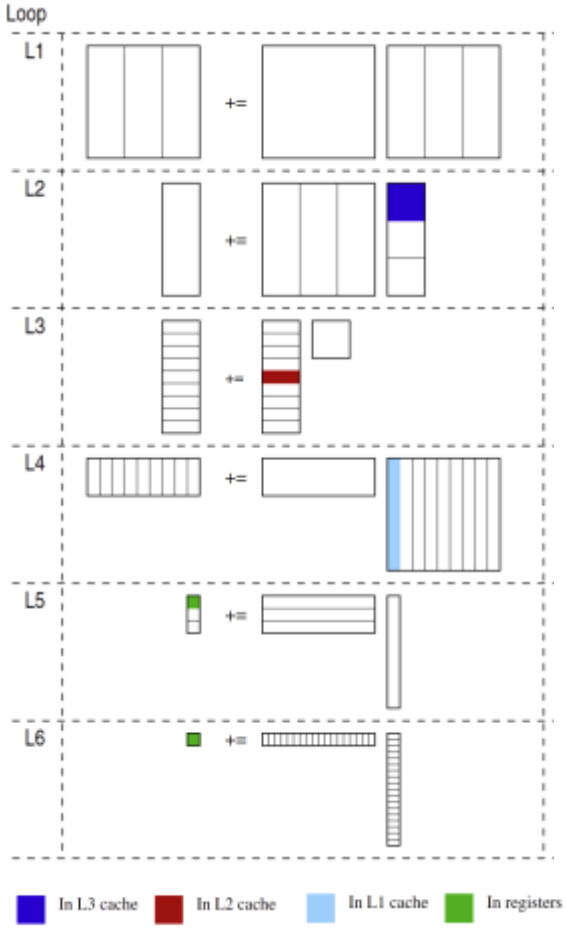
1. **Background on High-Performance GEMM**
2. **GEMM optimizations for RVV**
    1. Hand-tuned
    2. Automatic generation
3. **Experimental results**
4. **Conclusions**

Background

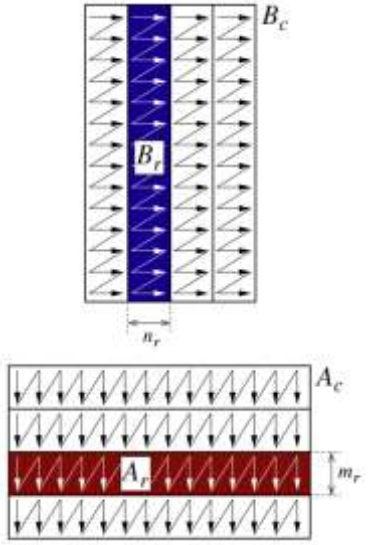
Anatomy of a high-performance GEMM

C = C + AB

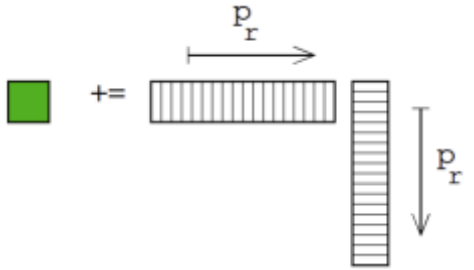
C: *mxn*; A: *mxk*; B: *kxn*



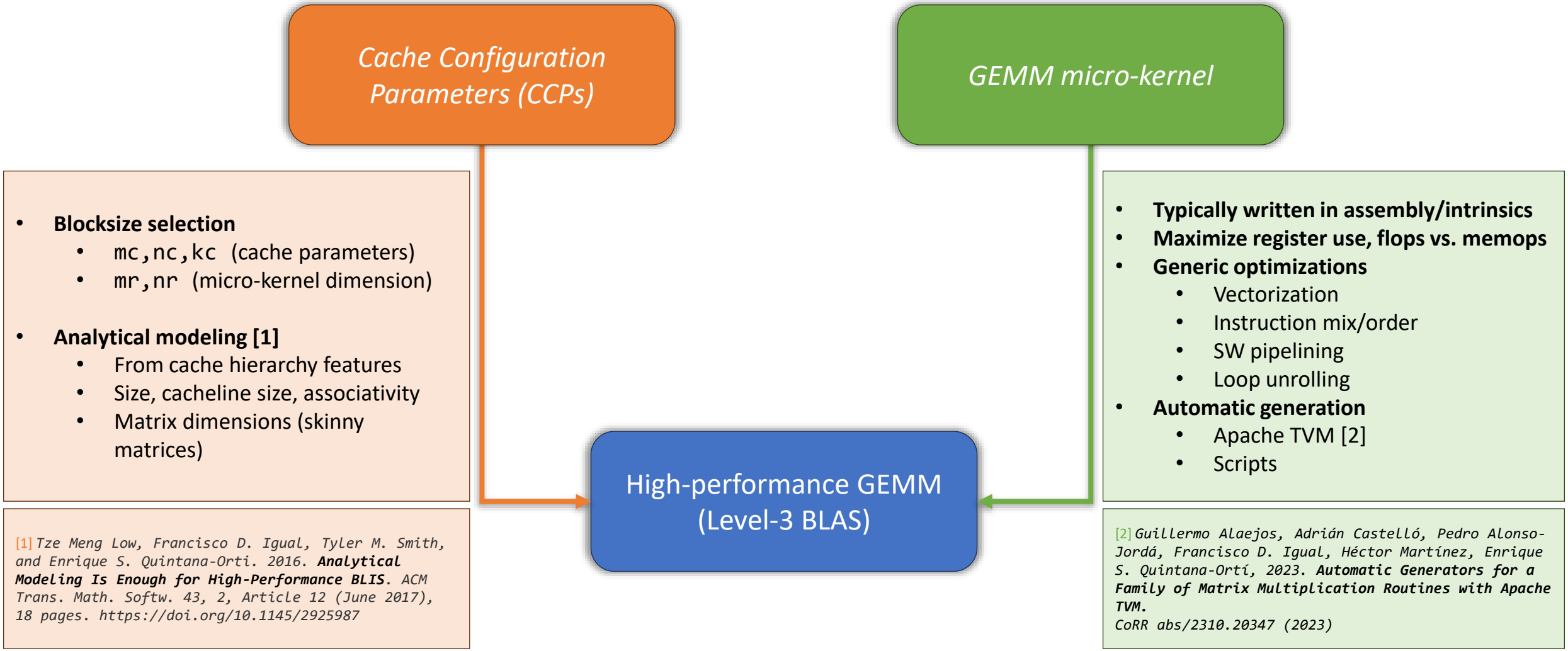

```
1  for (jc=0; jc<n; jc+=nc) // Loop L1
2   for (pc=0; pc<k; pc+=kc) {   // L2
3    // Pack B
4    Bc := B(pc:pc+kc-1,jc:jc+nc-1);
5    for (ic=0; ic<m; ic+=mc) {  // L3
6     // Pack A
7     Ac := A(ic:ic+mc-1,pc:pc+kc-1);
8     for (jr=0; jr<nc; jr+=nr)  // L4
9      for (ir=0; ir<mc; ir+=mr) // L5
10      // Micro-kernel
11      C(ic+ir:ic+ir+mr-1,
12        jc+jr:jc+jr+nr-1)
13        += Ac(ir:ir+mr-1,0:kc-1)
14        *  Bc(0:kc-1,jr:jr+nr-1);
15  }}
```




```
1  for (pr=0; pr<kc; pr++) // Loop L6
2    C(ic+ir:ic+ir+mr-1,
3      jc+jr:jc+jr+nr-1)
4      += Ac(ir:ir+mr-1,pr)
5      *  Bc(pr,jr:jr+nr-1);
```



Automation of a high-performance GEMM



- **Blocksize selection**
  - `mc,nc,kc` (cache parameters)
  - `mr,nr` (micro-kernel dimension)
- **Analytical modeling [1]**
  - From cache hierarchy features
  - Size, cacheline size, associativity
  - Matrix dimensions (skinny matrices)

- **Typically written in assembly/intrinsics**
- **Maximize register use, flops vs. memops**
- **Generic optimizations**
  - Vectorization
  - Instruction mix/order
  - SW pipelining
  - Loop unrolling
- **Automatic generation**
  - Apache TVM [2]
  - Scripts

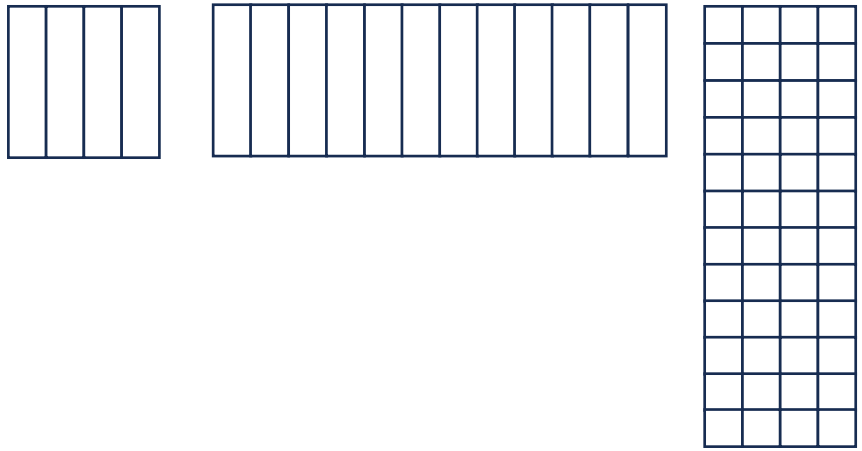
[1] *Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016.* ***Analytical Modeling Is Enough for High-Performance BLIS****. ACM Trans. Math. Softw. 43, 2, Article 12 (June 2017), 18 pages. https://doi.org/10.1145/2925987*

[2] *Guillermo Alaejos, Adrián Castelló, Pedro Alonso-Jordá, Francisco D. Igual, Héctor Martínez, Enrique S. Quintana-Ortí, 2023.* ***Automatic Generators for a Family of Matrix Multiplication Routines with Apache TVM.*** *CoRR abs/2310.20347 (2023)*

GEMM optimizations for RVV

Baseline ASM micro-kernel (4x4)

1. Vector load (`vle`) of column of Ar
2. Scalar load (`flw`) of elements of row of Br
3. Accumulation using vector-scalar (`vfmacc.vf`)



```
.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)          # Load the pr-th column of
  addi Ar, Ar, 16            # Ar into vector registers

  flw ft0, 0(Br)             # Scalar load the pr-th row of
  flw ft1, 4(Br)             # Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)
  addi Br, Br, 16

  vfmacc.vf C00, ft0, A0     # Scalar-vector accum. (Col. 0)
  vfmacc.vf C01, ft1, A0     # Scalar-vector accum. (Col. 1)
  vfmacc.vf C02, ft2, A0     # Scalar-vector accum. (Col. 2)
  vfmacc.vf C03, ft3, A0     # Scalar-vector accum. (Col. 3)
.endm
```

Stage 1. Load micro-tile Cr to V.Regs.

Stage 2. Updates Cr at each iteration

Stage 3. Writes back Cr to main memory

```
// gemm_ukernel_4x4(int kc, float *Ar, float *Br,
//                  float *C, int ldC)
// mr x nr = 4 x 4 micro-kernel
// Inputs:
//   - kc:  k-dimension of micro-kernel
//   - Ar:  packed micro-panel of Ac, with leading dimension mr
//   - Br:  packed micro-panel of Bc, with leading dimension nr
//   - C:   micro-tile of C stored in column-major order
//   - ldC: leading dimension of C
//
.text
.align  2
.global gemm_ukernel_asm_4x4
#define kc          a0
#define Ar          a1
#define Br          a2
#define C           a3
#define ldC         a4
#define C01_ptr     t0
#define C02_ptr     t1
#define C03_ptr     t2
                            # Vector registers for...
#define C00         v0      # 4x4 micro-tile of C
#define C01         v1
#define C02         v2
#define C03         v3
#define A0          v4      # Single column of Ar

.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)         # Load the pr-th column of
  addi Ar, Ar, 16           # Ar into vector registers

  flw ft0, 0(Br)            # Scalar load the pr-th row of
  flw ft1, 4(Br)            # Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)
  addi Br, Br, 16

  vfmacc.vf C00, ft0, A0    # Scalar-vector accum. (Col. 0)
  vfmacc.vf C01, ft1, A0    # Scalar-vector accum. (Col. 1)
  vfmacc.vf C02, ft2, A0    # Scalar-vector accum. (Col. 2)
  vfmacc.vf C03, ft3, A0    # Scalar-vector accum. (Col. 3)
.endm

gemm_ukernel_asm_4x4:
  li t1, 4
  vsetvli t1, t1, e32, m1

  add C01_ptr, C, ldC
  add C02_ptr, C01_ptr, ldC
  add C03_ptr, C02_ptr, ldC

  vle32.v C00, (C)          # Load the micro-tile
  vle32.v C01, (C01_ptr)    # of C into vector
  vle32.v C02, (C02_ptr)    # registers
  vle32.v C03, (C03_ptr)

LOOP:                       # Main loop
  LOOP_BODY_4x4             # for (pr=0; pr<kc; pr++)
  addi kc, kc, -1
  bne kc, zero, LOOP_4x4

  vse32.v C00, (C)          # Store the micro-tile
  vse32.v C01, (C01_ptr)    # of C back into memory
  vse32.v C02, (C02_ptr)
  vse32.v C03, (C03_ptr)

  ret
```

Optimization 1: broacasting (vfmv)

1. Vector load (`vle`) of a column of Ar
2. Scalar load (`flw`) of elements of row of Br
3. **Broadcast (`vfmv.v.f`) to vector registers**
4. **Accumulation using vector-vector (`vfmacc.vv`)**

```
.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)       # Load the pr-th column of
  addi Ar, Ar, 16         # Ar into vector registers

  flw ft0, 0(Br)          # Scalar load of the pr-th row
  flw ft1, 4(Br)          # of Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)

  vfmv.v.f B0, ft0        # Broadcast of scalar elements of
  vfmv.v.f B1, ft1        # Br into vector registers
  vfmv.v.f B2, ft2
  vfmv.v.f B3, ft3
  addi Br, Br, 16

  vfmacc.vv C00, A0, B0  # Vector-vector accumulation (Col. 0)
  vfmacc.vv C01, A0, B1  # Vector-vector accumulation (Col. 1)
  vfmacc.vv C02, A0, B2  # Vector-vector accumulation (Col. 2)
  vfmacc.vv C03, A0, B3  # Vector-vector accumulation (Col. 3)
.endm
```

```
// gemm_ukernel_4x4(int kc, float *Ar, float *Br,
//                  float *C, int ldC)
// mr x nr = 4 x 4 micro-kernel
// Inputs:
//   - kc:  k-dimension of micro-kernel
//   - Ar:  packed micro-panel of Ac, with leading dimension mr
//   - Br:  packed micro-panel of Bc, with leading dimension nr
//   - C:   micro-tile of C stored in column-major order
//   - ldC: leading dimension of C
//
.text
.align  2
.global gemm_ukernel_asm_4x4
#define kc          a0
#define Ar          a1
#define Br          a2
#define C           a3
#define ldC         a4
#define C01_ptr     t0
#define C02_ptr     t1
#define C03_ptr     t2
                            # Vector registers for...
#define C00         v0      # 4x4 micro-tile of C
#define C01         v1
#define C02         v2
#define C03         v3
#define A0          v4      # Single column of Ar

.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)         # Load the pr-th column of
  addi Ar, Ar, 16           # Ar into vector registers

  flw ft0, 0(Br)            # Scalar load the pr-th row of
  flw ft1, 4(Br)            # Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)
  addi Br, Br, 16

  vfmacc.vf C00, ft0, A0    # Scalar-vector accum. (Col. 0)
  vfmacc.vf C01, ft1, A0    # Scalar-vector accum. (Col. 1)
  vfmacc.vf C02, ft2, A0    # Scalar-vector accum. (Col. 2)
  vfmacc.vf C03, ft3, A0    # Scalar-vector accum. (Col. 3)
.endm

gemm_ukernel_asm_4x4:
  li t1, 4
  vsetvli t1, t1, e32, m1

  add C01_ptr, C, ldC
  add C02_ptr, C01_ptr, ldC
  add C03_ptr, C02_ptr, ldC

  vle32.v C00, (C)          # Load the micro-tile
  vle32.v C01, (C01_ptr)    # of C into vector
  vle32.v C02, (C02_ptr)    # registers
  vle32.v C03, (C03_ptr)

LOOP:                       # Main loop
  LOOP_BODY_4x4             # for (pr=0; pr<kc; pr++)
  addi kc, kc, -1
  bne kc, zero, LOOP_4x4

  vse32.v C00, (C)          # Store the micro-tile
  vse32.v C01, (C01_ptr)    # of C back into memory
  vse32.v C02, (C02_ptr)
  vse32.v C03, (C03_ptr)

  ret
```

Optimization 2: broacasting (vrgather)

1. Vector load (`vle`) of a column of Ar
2. **Vector load (`vle`) of a row of Br**
3. **Use `vrgather.vi` to splat individual elements of Br**
4. Accumulation using vector-vector (`vfmacc.vv`)

```
.macro LOOP_BODY_4x4
  vle32.v A0, (Ar)         # Load the pr-th column of
  addi Ar, Ar, 16          # Ar into vector registers

  vle32.v Btmp, (Br)       # Load the pr-th row of
  addi Br, Br, 16          # Br into vector registers

  vrgather.vi B0, Btmp, 0 # Splat individual elements (lanes)
  vrgather.vi B1, Btmp, 1 # of Br into vector registers.
  vrgather.vi B2, Btmp, 2
  vrgather.vi B3, Btmp, 3

  vfmacc.vv C00, A0, B0    # Vector-vector accumulation (Col. 0)
  vfmacc.vv C01, A0, B1    # Vector-vector accumulation (Col. 1)
  vfmacc.vv C02, A0, B2    # Vector-vector accumulation (Col. 2)
  vfmacc.vv C03, A0, B3    # Vector-vector accumulation (Col. 3)
.endm
```

```
// gemm_ukernel_4x4(int kc, float *Ar, float *Br,
//                  float *C, int ldC)
// mr x nr = 4 x 4 micro-kernel
// Inputs:
//   - kc:  k-dimension of micro-kernel
//   - Ar:  packed micro-panel of Ac, with leading dimension mr
//   - Br:  packed micro-panel of Bc, with leading dimension nr
//   - C:   micro-tile of C stored in column-major order
//   - ldC: leading dimension of C
//
.text
.align  2
.global gemm_ukernel_asm_4x4
#define kc          a0
#define Ar          a1
#define Br          a2
#define C           a3
#define ldC         a4
#define C01_ptr     t0
#define C02_ptr     t1
#define C03_ptr     t2
                            # Vector registers for...
#define C00         v0      # 4x4 micro-tile of C
#define C01         v1
#define C02         v2
#define C03         v3
#define A0          v4      # Single column of Ar

.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)         # Load the pr-th column of
  addi Ar, Ar, 16           # Ar into vector registers

  flw ft0, 0(Br)            # Scalar load the pr-th row of
  flw ft1, 4(Br)            # Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)
  addi Br, Br, 16

  vfmacc.vf C00, ft0, A0    # Scalar-vector accum. (Col. 0)
  vfmacc.vf C01, ft1, A0    # Scalar-vector accum. (Col. 1)
  vfmacc.vf C02, ft2, A0    # Scalar-vector accum. (Col. 2)
  vfmacc.vf C03, ft3, A0    # Scalar-vector accum. (Col. 3)
.endm

gemm_ukernel_asm_4x4:
  li t1, 4
  vsetvli t1, t1, e32, m1

  add C01_ptr, C, ldC
  add C02_ptr, C01_ptr, ldC
  add C03_ptr, C02_ptr, ldC

  vle32.v C00, (C)          # Load the micro-tile
  vle32.v C01, (C01_ptr)    # of C into vector
  vle32.v C02, (C02_ptr)    # registers
  vle32.v C03, (C03_ptr)

LOOP:                       # Main loop
  LOOP_BODY_4x4             # for (pr=0; pr<kc; pr++)
  addi kc, kc, -1
  bne kc, zero, LOOP_4x4

  vse32.v C00, (C)          # Store the micro-tile
  vse32.v C01, (C01_ptr)    # of C back into memory
  vse32.v C02, (C02_ptr)
  vse32.v C03, (C03_ptr)

  ret
```

Optimization 3: load order (B->A)

- Rearrange load order:
  - Elements of Br loaded **before** Ar

```
.macro LOOP_BODY_4x4
 vle32.v Btmp, (Br)        # Load the pr-th row of
 addi Br, Br, 16           # Br into vector registers

 vle32.v A0, (Ar)          # Load the pr-th column of
 addi Ar, Ar, 16           # Ar into vector registers

 # (Rest of the micro-kernel ommited for brevity)
```

Stage 1. Load micro-tile Cr to V.Regs.

Stage 2. Updates Cr at each iteration

Stage 3. Writes back Cr to main memory

```
// gemm_ukernel_4x4(int kc, float *Ar, float *Br,
//                  float *C, int ldC)
// mr x nr = 4 x 4 micro-kernel
// Inputs:
//   - kc:  k-dimension of micro-kernel
//   - Ar:  packed micro-panel of Ac, with leading dimension mr
//   - Br:  packed micro-panel of Bc, with leading dimension nr
//   - C:   micro-tile of C stored in column-major order
//   - ldC: leading dimension of C
//
.text
.align  2
.global gemm_ukernel_asm_4x4
#define kc          a0
#define Ar          a1
#define Br          a2
#define C           a3
#define ldC         a4
#define C01_ptr     t0
#define C02_ptr     t1
#define C03_ptr     t2
                            # Vector registers for...
#define C00         v0      # 4x4 micro-tile of C
#define C01         v1
#define C02         v2
#define C03         v3
#define A0          v4      # Single column of Ar

.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)         # Load the pr-th column of
  addi Ar, Ar, 16           # Ar into vector registers

  flw ft0, 0(Br)            # Scalar load the pr-th row of
  flw ft1, 4(Br)            # Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)
  addi Br, Br, 16

  vfmacc.vf C00, ft0, A0    # Scalar-vector accum. (Col. 0)
  vfmacc.vf C01, ft1, A0    # Scalar-vector accum. (Col. 1)
  vfmacc.vf C02, ft2, A0    # Scalar-vector accum. (Col. 2)
  vfmacc.vf C03, ft3, A0    # Scalar-vector accum. (Col. 3)
.endm

gemm_ukernel_asm_4x4:
  li t1, 4
  vsetvli t1, t1, e32, m1

  add C01_ptr, C, ldC
  add C02_ptr, C01_ptr, ldC
  add C03_ptr, C02_ptr, ldC

  vle32.v C00, (C)          # Load the micro-tile
  vle32.v C01, (C01_ptr)    # of C into vector
  vle32.v C02, (C02_ptr)    # registers
  vle32.v C03, (C03_ptr)

LOOP:                       # Main loop
  LOOP_BODY_4x4             # for (pr=0; pr<kc; pr++)
  addi kc, kc, -1
  bne kc, zero, LOOP_4x4

  vse32.v C00, (C)          # Store the micro-tile
  vse32.v C01, (C01_ptr)    # of C back into memory
  vse32.v C02, (C02_ptr)
  vse32.v C03, (C03_ptr)

  ret
```

Optimization 4: general techniques

- Combined with previous optimizations:
  1. Loop unrolling
  2. Software pipelining

```
// gemm_ukernel_4x4(int kc, float *Ar, float *Br,
//                  float *C, int ldC)
// mr x nr = 4 x 4 micro-kernel
// Inputs:
//   - kc:  k-dimension of micro-kernel
//   - Ar:  packed micro-panel of Ac, with leading dimension mr
//   - Br:  packed micro-panel of Bc, with leading dimension nr
//   - C:   micro-tile of C stored in column-major order
//   - ldC: leading dimension of C
//
.text
.align  2
.global gemm_ukernel_asm_4x4
#define kc          a0
#define Ar          a1
#define Br          a2
#define C           a3
#define ldC         a4
#define C01_ptr     t0
#define C02_ptr     t1
#define C03_ptr     t2
                            # Vector registers for...
#define C00         v0      # 4x4 micro-tile of C
#define C01         v1
#define C02         v2
#define C03         v3
#define A0          v4      # Single column of Ar

.macro LOOP_BODY_4x4
  vle32.v A0,  (Ar)         # Load the pr-th column of
  addi Ar, Ar, 16           # Ar into vector registers

  flw ft0, 0(Br)            # Scalar load the pr-th row of
  flw ft1, 4(Br)            # Br into scalar registers
  flw ft2, 8(Br)
  flw ft3, 12(Br)
  addi Br, Br, 16

  vfmacc.vf C00, ft0, A0    # Scalar-vector accum. (Col. 0)
  vfmacc.vf C01, ft1, A0    # Scalar-vector accum. (Col. 1)
  vfmacc.vf C02, ft2, A0    # Scalar-vector accum. (Col. 2)
  vfmacc.vf C03, ft3, A0    # Scalar-vector accum. (Col. 3)
.endm

gemm_ukernel_asm_4x4:
  li t1, 4
  vsetvli t1, t1, e32, m1

  add C01_ptr, C, ldC
  add C02_ptr, C01_ptr, ldC
  add C03_ptr, C02_ptr, ldC

  vle32.v C00, (C)          # Load the micro-tile
  vle32.v C01, (C01_ptr)    # of C into vector
  vle32.v C02, (C02_ptr)    # registers
  vle32.v C03, (C03_ptr)

LOOP:                       # Main loop
  LOOP_BODY_4x4             # for (pr=0; pr<kc; pr++)
  addi kc, kc, -1
  bne kc, zero, LOOP_4x4

  vse32.v C00, (C)          # Store the micro-tile
  vse32.v C01, (C01_ptr)    # of C back into memory
  vse32.v C02, (C02_ptr)
  vse32.v C03, (C03_ptr)

  ret
```

Stage 1. Load micro-tile Cr to V.Regs.

Stage 2. Updates Cr at each iteration

Stage 3. Writes back Cr to main memory

Optimization summary

Name	Load A	Load B	Load Order	Broadcast of B	Accumulation	Pipelining	Unroll
AUTO-BASELINE	Vector	Scalar	AB	–	VF	–	–
AUTO-OP1	Vector	Scalar	AB	VFMV	VV	–	–
AUTO-OP2	Vector	Vector	AB	Gather	VV	–	–
AUTO-OP3	Vector	Vector	BA	Gather	VV	–	–
AUTO-OP4	Vector	Vector	BA	Gather	VV	Yes	Yes (2)

Automatic micro-kernel generation

- Previous optimizations exhibit a very regular structure
- Python generator routines parametrized by:
  - Micro-kernel dimensions (mr, nr)
  - Vector length (vl)
  - Datatype
- Generator driver:
  1. Parametrized by (mr, nr)
  2. Receives desired optimizations
  3. Applies analytical modeling for CCPs (mc, nc, kc)
  4. Generates GEMM codes that apply partitioning + packing + **optimized micro-kernel**

```
#define LOOP_BODY_COMMON_4x4(Bmacro, Amacro) \
    GATHER B0, Bmacro, 0                     \
    GATHER B1, Bmacro, 1                     \
    GATHER B2, Bmacro, 2                     \
    GATHER B3, Bmacro, 3                     \
                                             \
    VFMACC C00, B0, Amacro                   \
    VFMACC C01, B1, Amacro                   \
    VFMACC C02, B2, Amacro                   \
    VFMACC C03, B3, Amacro

#define LOOP_BODY_4x4(Bmacro, Amacro)        \
    LOAD B0, (B0_ptr)                        \
    LOAD Amacro, (A0_ptr)                    \
                                             \
    LOOP_BODY_COMMON_4x4(Bmacro, Amacro)     \
                                             \
    ADDI A0_ptr, A0_ptr, 16                  \
    ADDI B0_ptr, B0_ptr, 16

// gemm_ukernel_auto_op4_4x4(int kc, float *Ar, float *Br,
//                           float *C, int ldC )
// mr x nr = 4 x 4 micro - kernel for auto-Op4 (see Table 1)
// Inputs as in the baseline implementation in Figure 2
.text
.align      2
.global     gemm_ukernel_asm_auto_op4_4x4
// Macros for routine parameters, defines and header function
// Omitted for brevity

gemm_ukernel_asm_auto_op4_4x4:
  LI t1, 4
  VSETVLI t1, t1, e32, m1

  ADD C01_ptr, C00_ptr, ldC
  ADD C02_ptr, C01_ptr, ldC
  ADD C03_ptr, C02_ptr, ldC

  LOAD BTMP0_prfm, (B0_ptr)
  LOAD A0_prfm, (A0_ptr)

  LI unroll, 2
  DIV kc_iter, kc, unroll
  MUL kc_left, kc_iter, unroll
  SUB kc_left, kc, kc_left

  ADDI A0_ptr, A0_ptr, 16
  ADDI B0_ptr, B0_ptr, 16

  FCVT  ft8, zero
  FLW   BETA, (beta_ptr)
  FEQ   TMP, BETA, ft8
  BEQ   TMP, zero, LOAD_4x4

LOAD_4x4:
  LOAD C00, (C00_ptr)
  LOAD C01, (C01_ptr)
  LOAD C02, (C02_ptr)
  LOAD C03, (C03_ptr)

PREV_LOOP_4x4:
  BEQ kc_iter, zero, LOOP_LEFT_4x4
LOOP_4x4:
  LOOP_BODY_4x4(BTMP0_prfm, A0)
  LOOP_BODY_4x4(BTMP0, A0_prfm)

  ADDI kc_iter, kc_iter, -1
  BNE kc_iter, zero, LOOP_4x4

  BEQ kc_left, zero, STORE_4x4
LOOP_LEFT_4x4:
  LOOP_BODY_COMMON_4x4(BTMP0_prfm, A0_prfm)
STORE_4x4:
  STORE C00, (C00_ptr)
  STORE C01, (C01_ptr)
  STORE C02, (C02_ptr)
  STORE C03, (C03_ptr)
END:

ret
```

Experimental results

Platforms and experimental setup

XuanTie C910

- **T-HEAD 1520 SoC**
  - 4 x C910@1.85GHz
  - 12-stage, out-of-order superscalar
  - 2 vector slices (pipelines), 128-bit (VLEN)
  - RVV 0.7.1
- **L1**: 64 KiB, 2-way. **L2**: 1 MiB, 16-way

XuanTie C906

- **Allwinner D1 SoC**
  - 1 x C906@1GHz
  - 5-stage, in-order
  - 1 vector slice, 128-bit (VLEN)
  - RVV 0.7.1
- **L1**: 32 KiB, 4-way

Compiler:

- GCC toolchain 10.2 (port by T-HEAD, versión 2.6.1)
- Flags: `-march=rv64imafdcv0p7_zfh_xtheadc -mabi=lp64d, and -mtune=c910|c906`

OpenBLAS:

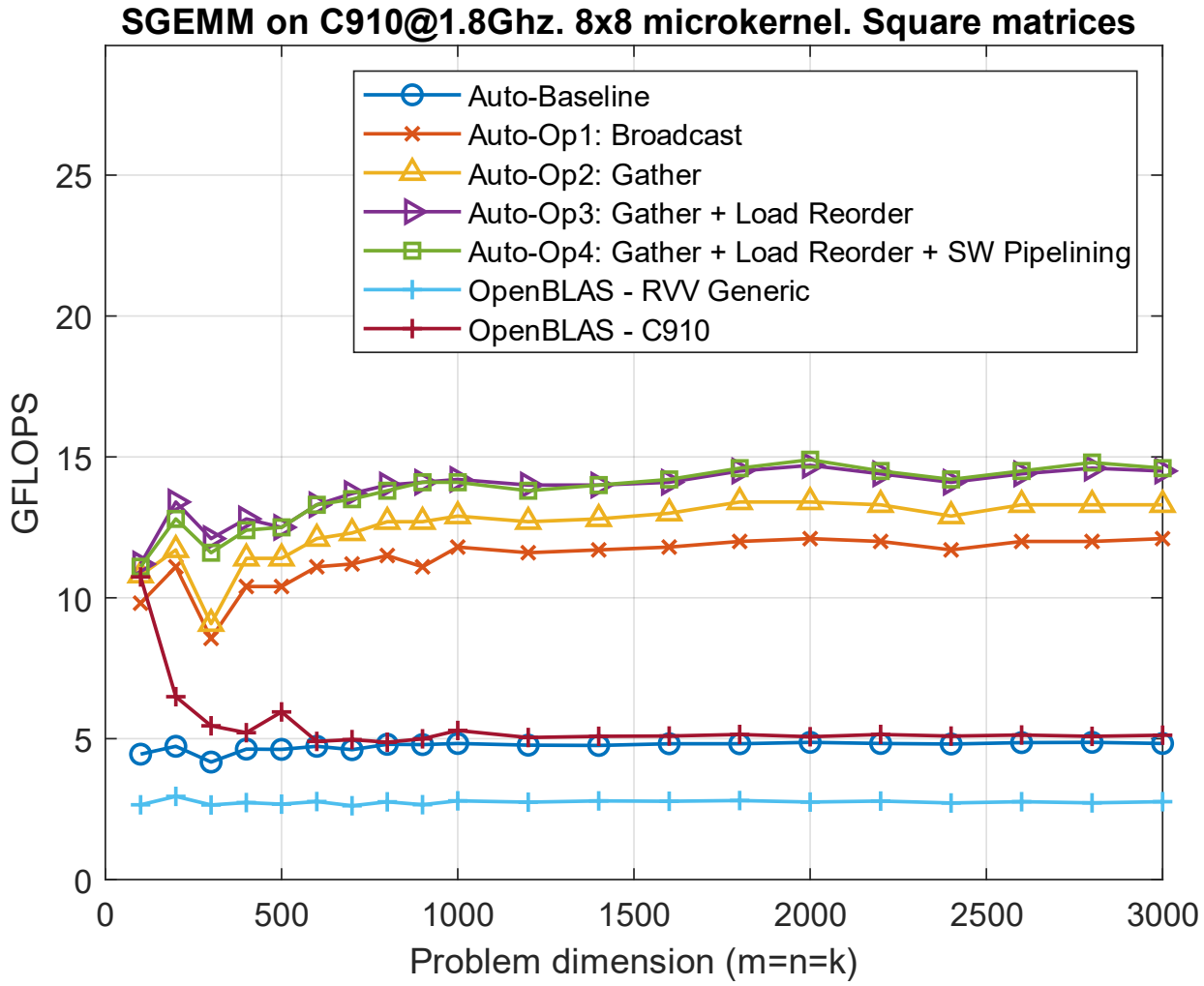
- Commit 23693f0
- Two configurations: RVV generic, C910

Experimental conditions

1. FP32, single core
2. 8x8 and 16x4 micro-kernels (examples)
3. Square matrices (*m=n=k*)
4. Resnet-50 (*rectangular matrices*)

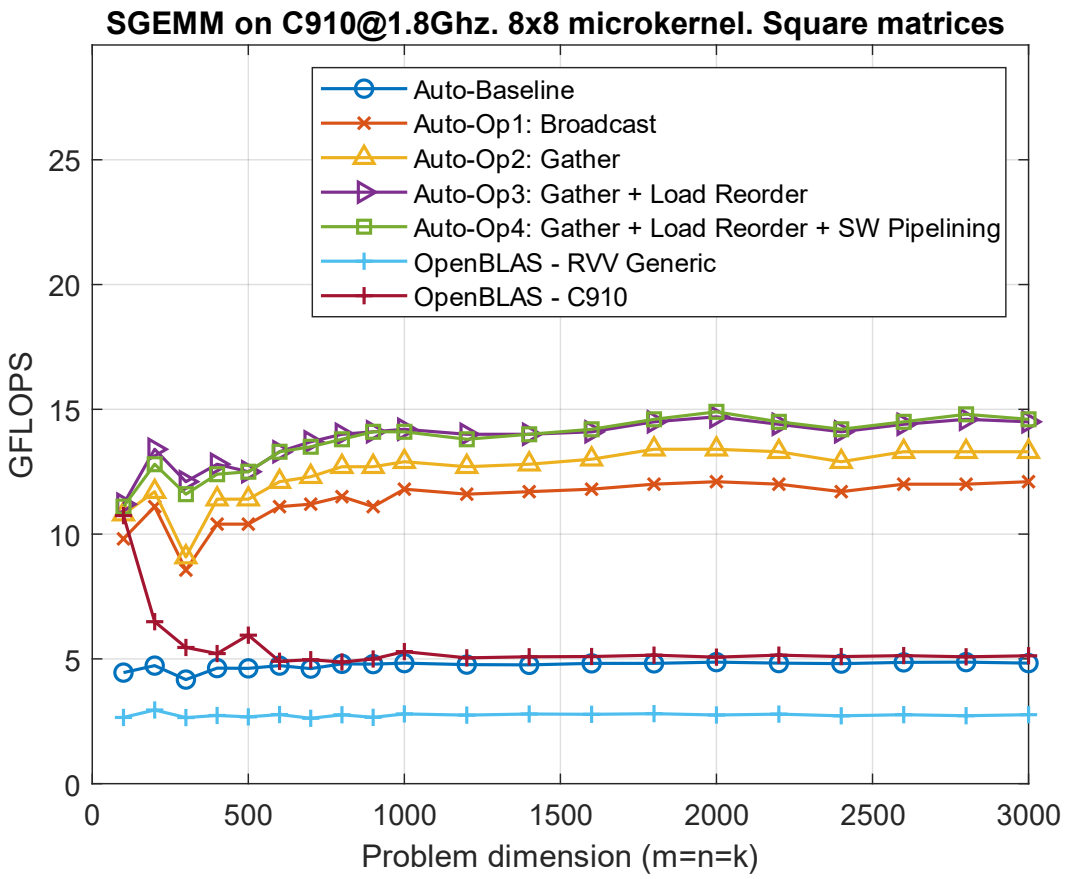
Layer type id.	Layer numbers in ResNet50 v1.5	*m*	*n*	*k*	Layer type id.	Layer numbers in ResNet50 v1.5	*m*	*n*	*k*
1	001	1,605,632	64	147	11	080	100,352	256	512
2	006	401,408	64	64	12	083/095/105/115/125/135	25,088	256	2,304
3	009/021/031	401,408	64	576	13	086/098/108/118/128/138	25,088	1,024	256
4	012/014/024/034	401,408	256	64	14	088	25,088	1,024	512
5	018/028	401,408	64	256	15	092/102/112/122/132	25,088	256	1,024
6	038	401,408	128	256	16	142	25,088	512	1,024
7	041/053/063/073	100,352	128	1,152	17	145/157/167	6,272	512	4,608
8	044/056/066/076	100,352	512	128	18	148/160/170	6,272	2,048	512
9	046	100,352	512	256	19	150	6,272	2,048	1,024
10	050/060/070	100,352	128	512	20	154/164	6,272	512	2,048

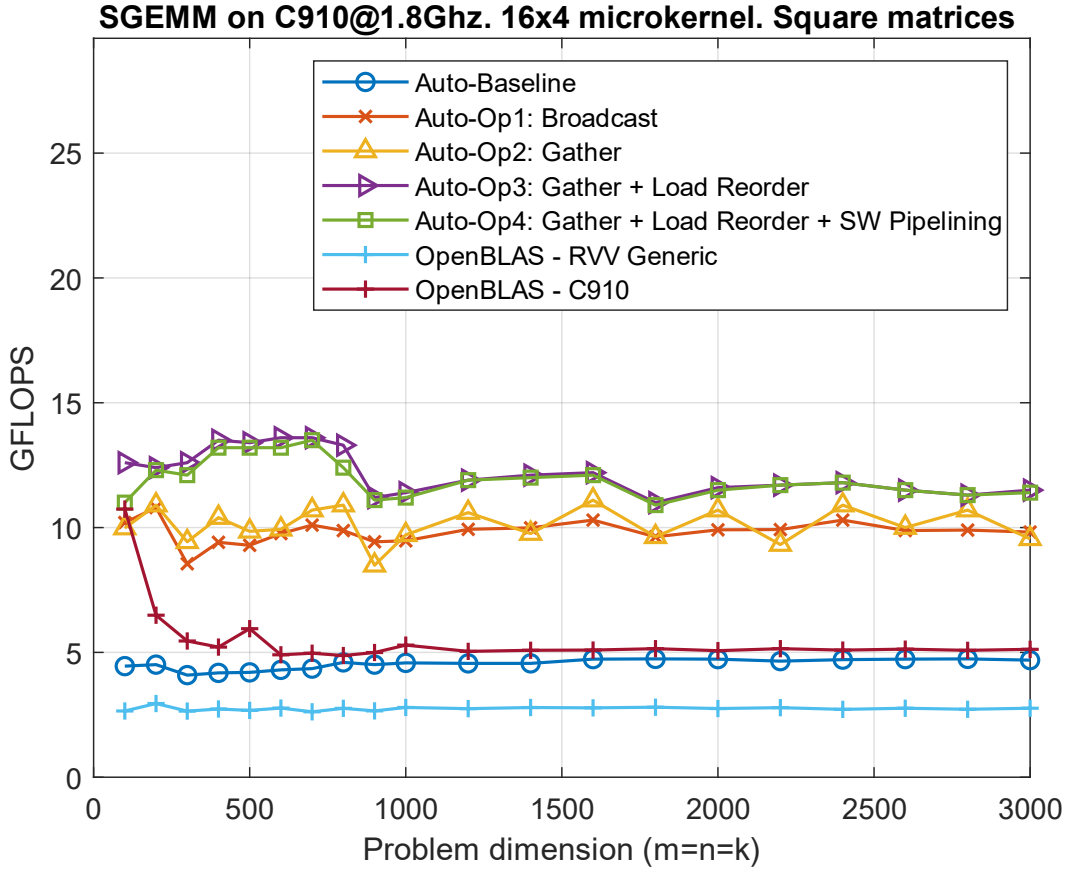
Results - C910, 8x8 microkernel, square matrices



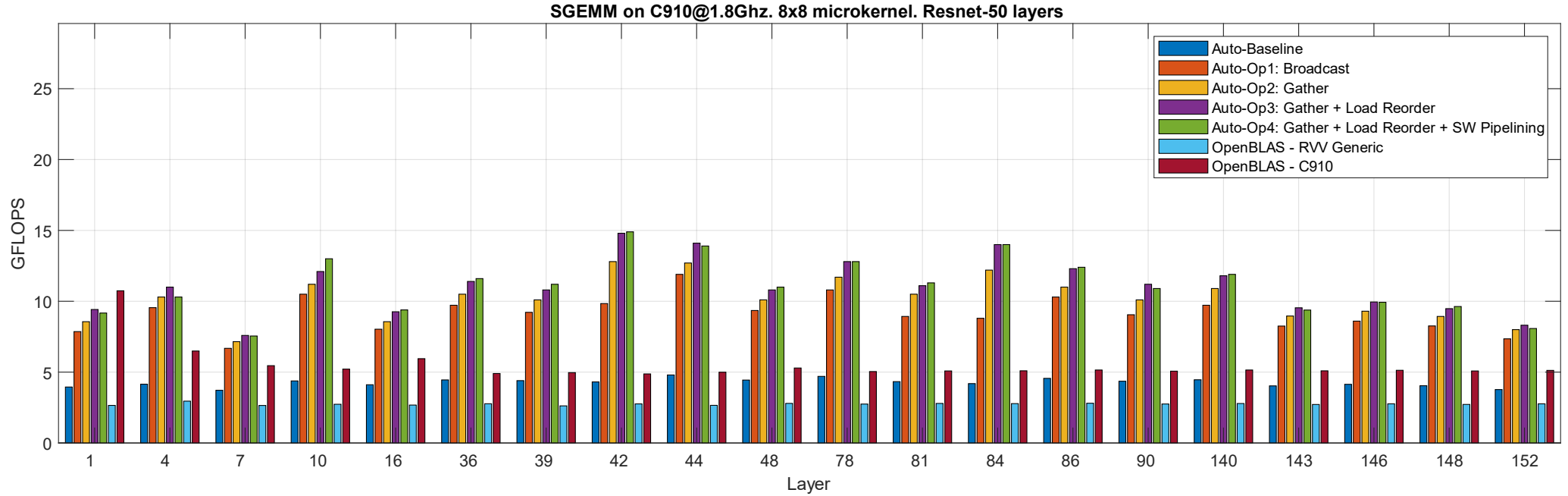
- **Auto-Baseline vs. OpenBLAS**
  - 1.72x improvement vs. OpenBLAS RVV Generic
  - Similar performance than OpenBLAS C910
- **Auto-Op1 (*bcast*)**
  - 2.38x improvement vs. Auto-Baseline
- **Auto-Op2 (*gather*)**
  - 2.62x improvement vs. Auto-Baseline
- **Auto-Op3 (*load reorder*)**
  - **2.90x improvement vs. Auto-Baseline**
  - **2.59x improvement vs. C910 OpenBLAS**
- **Auto-Op4 (*SW pipelining*)**
  - 2.88x improvement vs. Auto-Baseline

Results - C910, microkernel comparison, square matrices

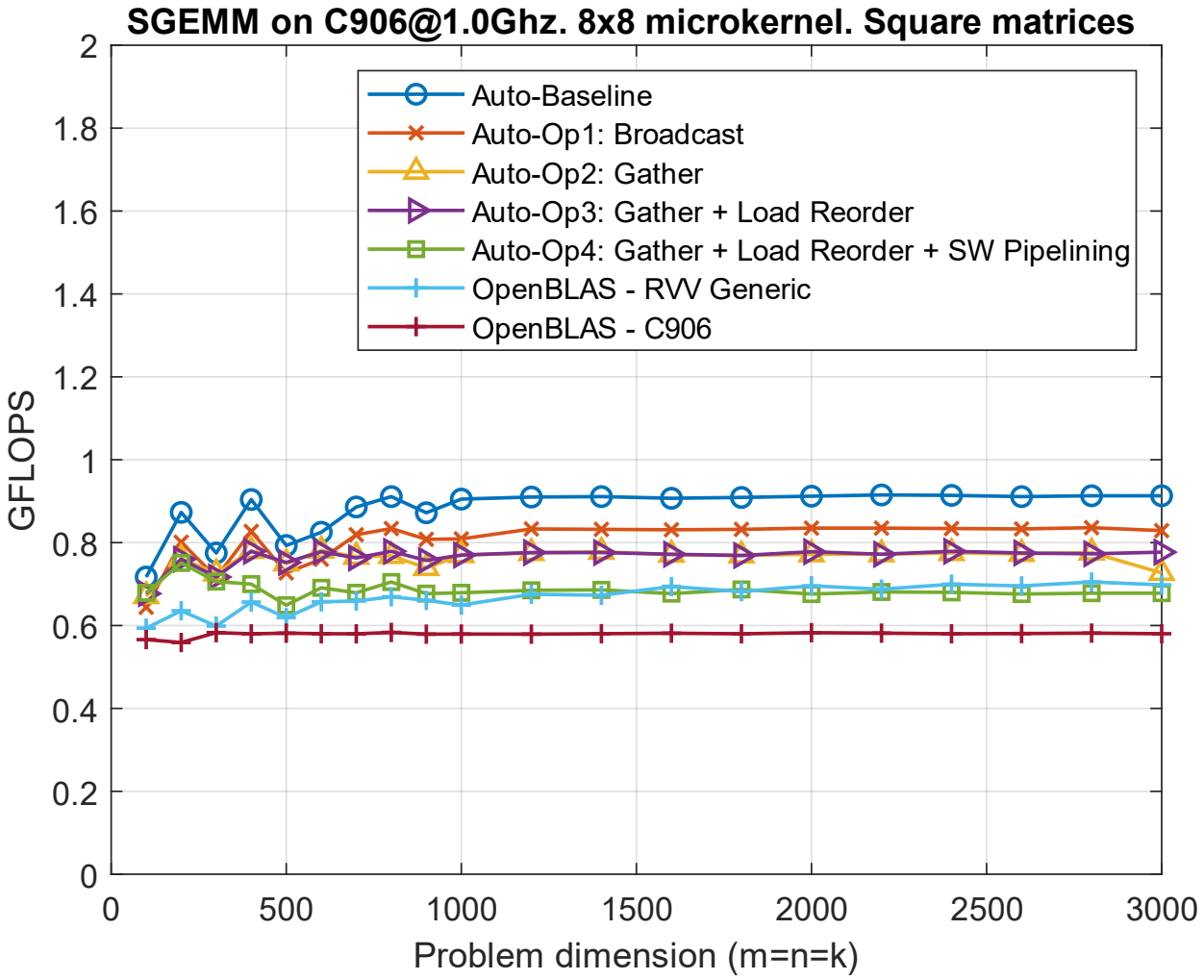




Results - C910, microkernel comparison, Resnet-50

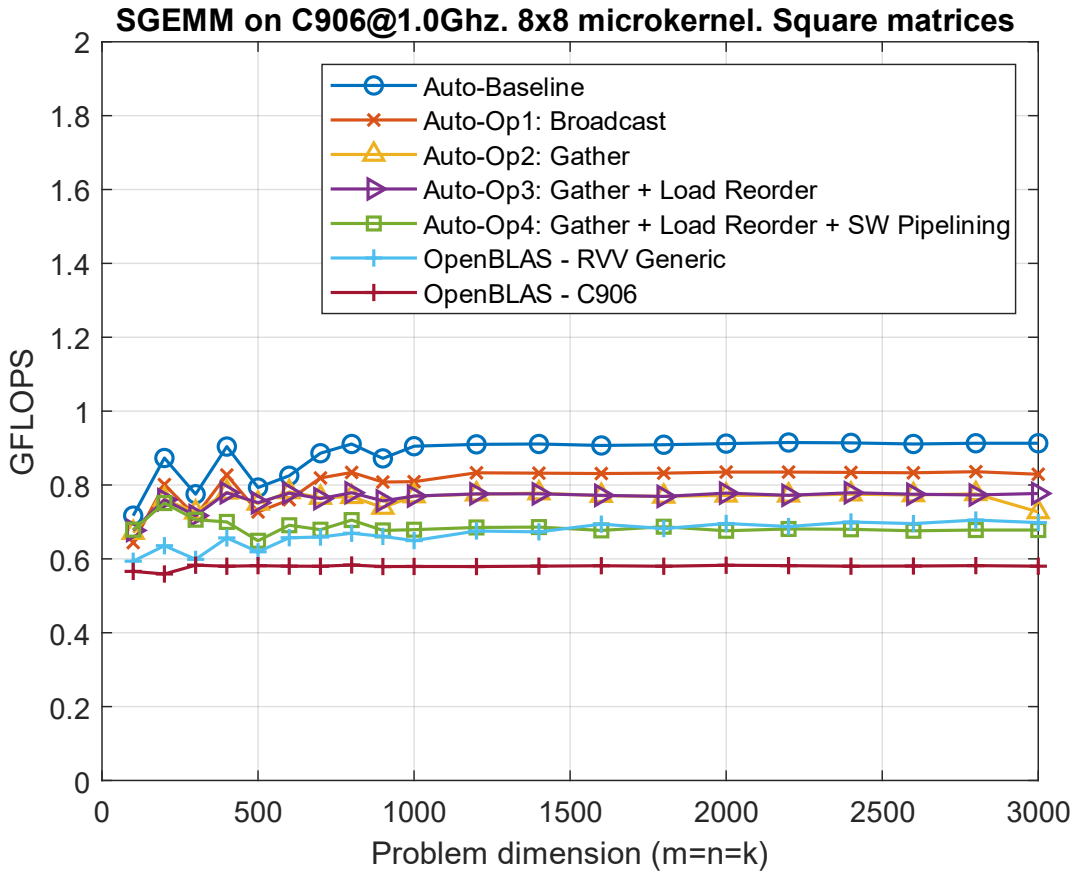


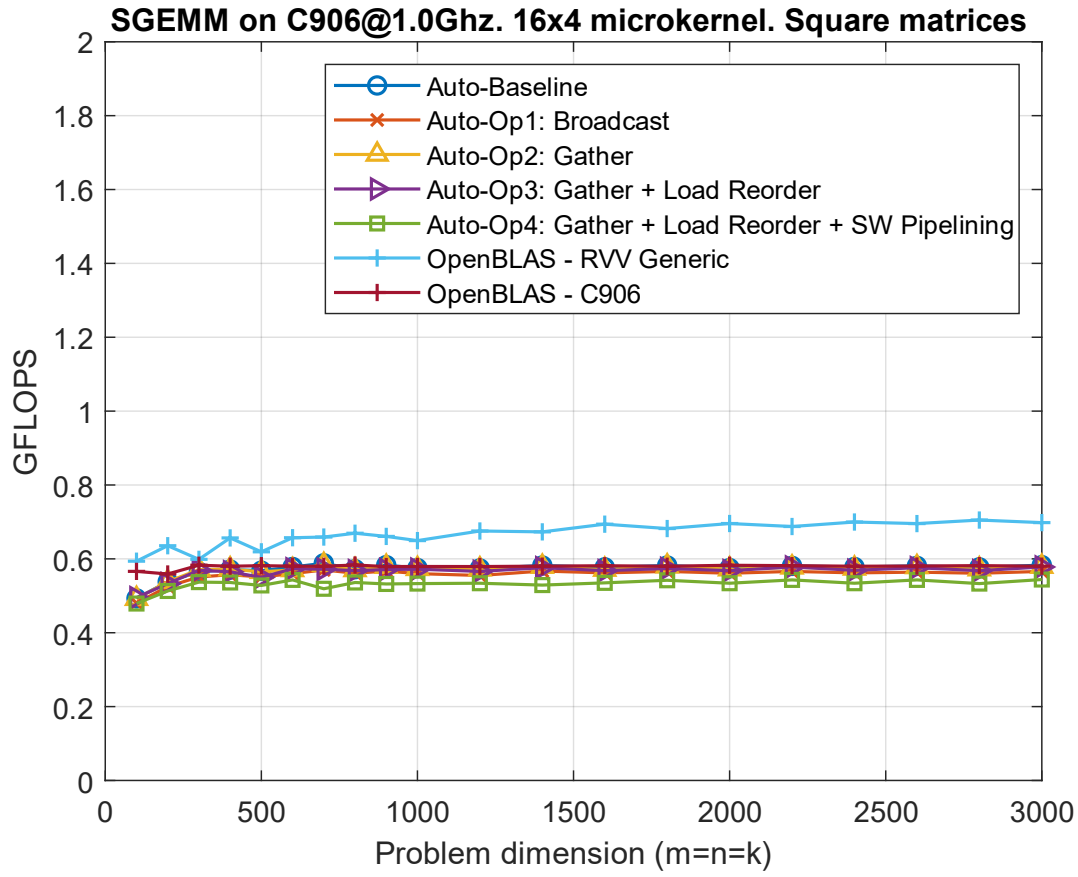
Results - C906, 8x8 microkernel, square matrices



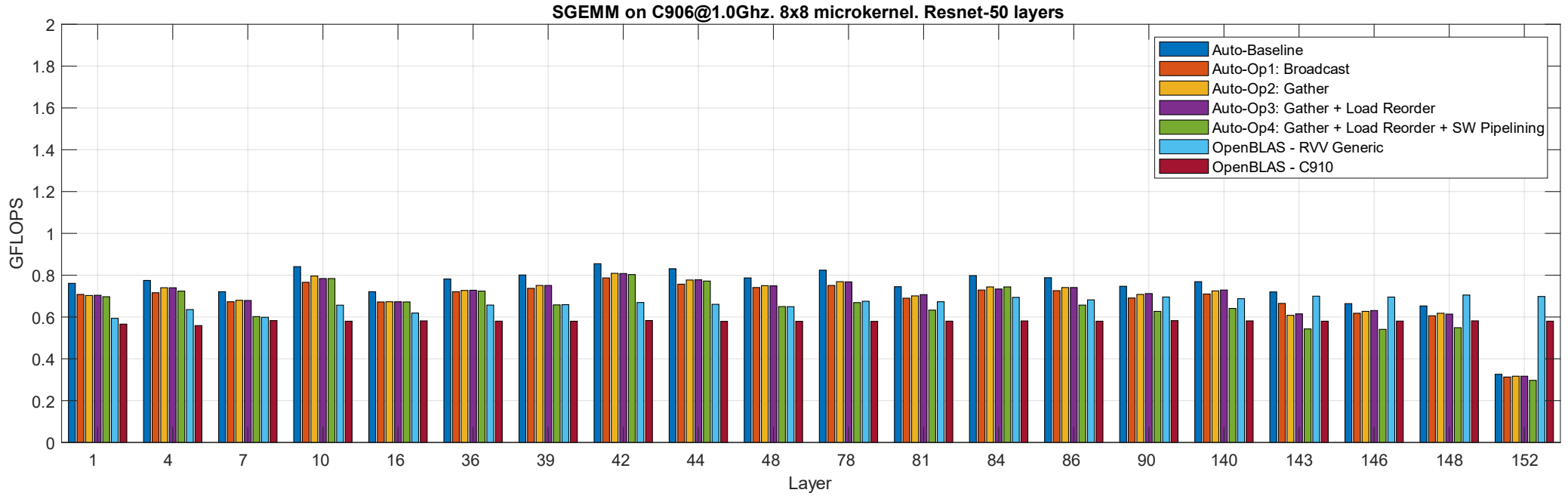
- **Auto-Baseline vs. OpenBLAS**
  - **1.32x improvement vs. OpenBLAS RVV Generic**
  - **1.51x improvement vs. OpenBLAS C910**
- **Auto-Op1 (*bcast*)**
  - 0.91x improvement vs. Auto-Baseline
- **Auto-Op2 (*gather*)**
  - 0.86x improvement vs. Auto-Baseline
- **Auto-Op3 (*load reorder*)**
  - 0.87x improvement vs. Auto-Baseline
- **Auto-Op4 (*SW pipelining*)**
  - 0.78x improvement vs. Auto-Baseline

Results - C906, 8x8 microkernel, square matrices





Results - C906, microkernel comparison, Resnet-50



Conclusions

Conclusions

- The development of micro-kernels for vector architectures is a well-structured task, with potential for automation

- Yielding a rich family of optimized micro-kernels enables the use of the most suitable depending on the underlying architecture, even when all of them implement a common ISA (in our case, RISC-V + RVV)

- Performance results for the C910/C906 demonstrate remarkable performance benefits compared with state-of-the-art BLAS implementations (OpenBLAS)

Automatic Generation of Micro-kernels for Performance Portability of Matrix Multiplication on RISC-V Vector Processors

Second International workshop on RISC-V for HPC

Francisco D. Igual

Luis Piñuel

Universidad Complutense de Madrid

Héctor Martínez

Universidad de Córdoba

Sandra Catalán

Universitat Jaume I de Castelló

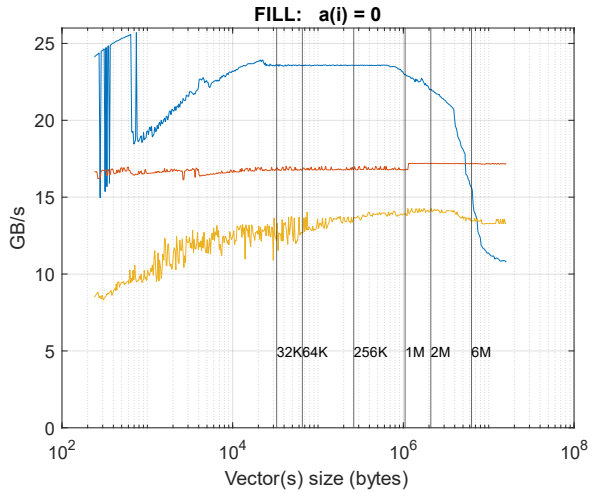
Adrián Castelló

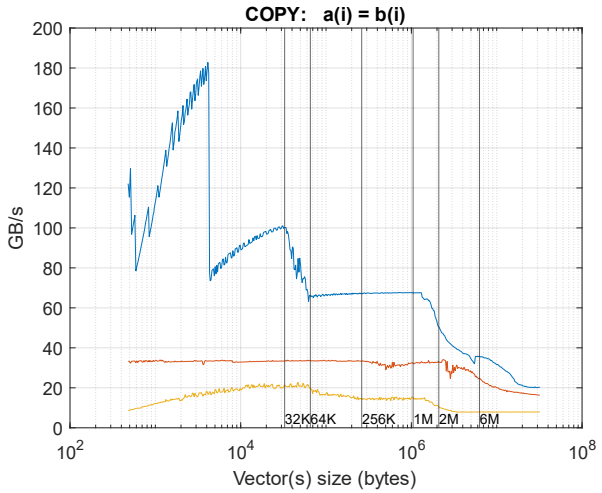
Enrique S. Quintana-Ortí

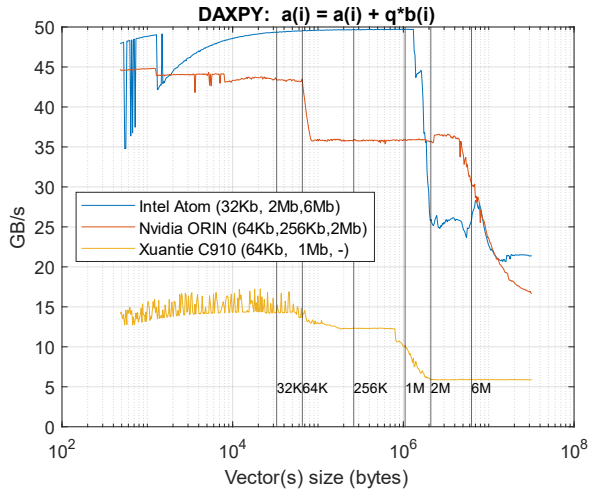
Universitat Politècnica de València

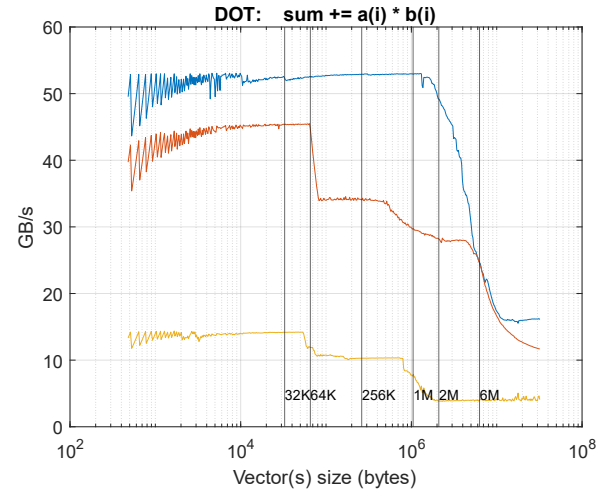
Backup slides

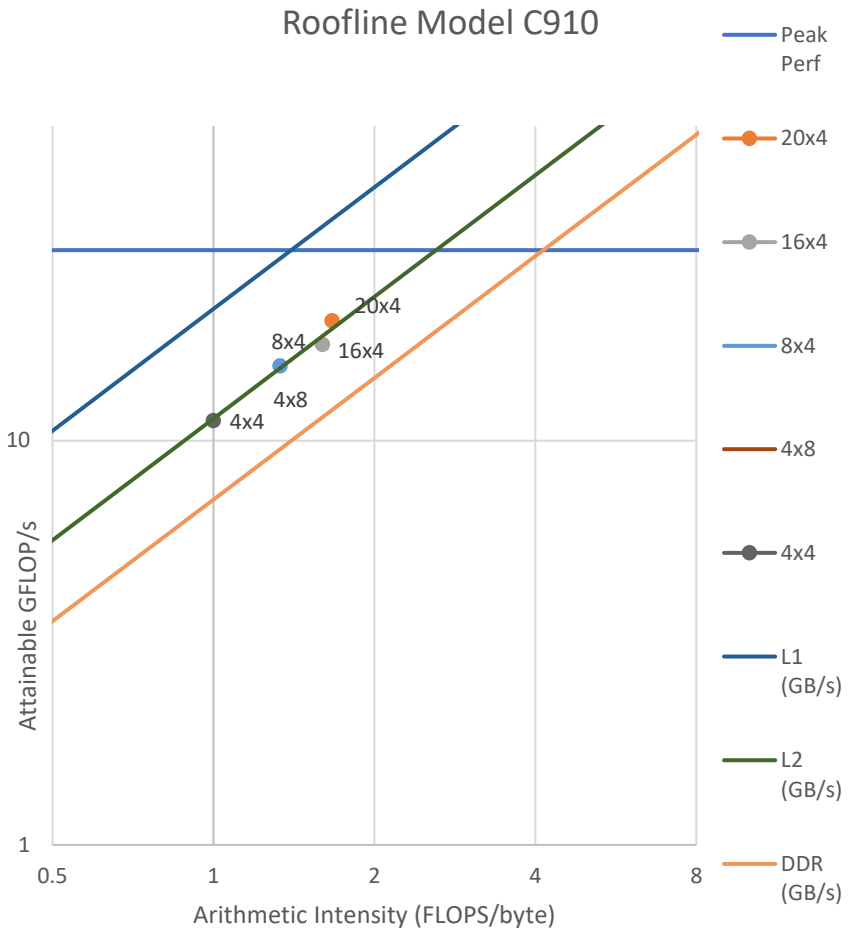
C910. STREAM – Roofline model





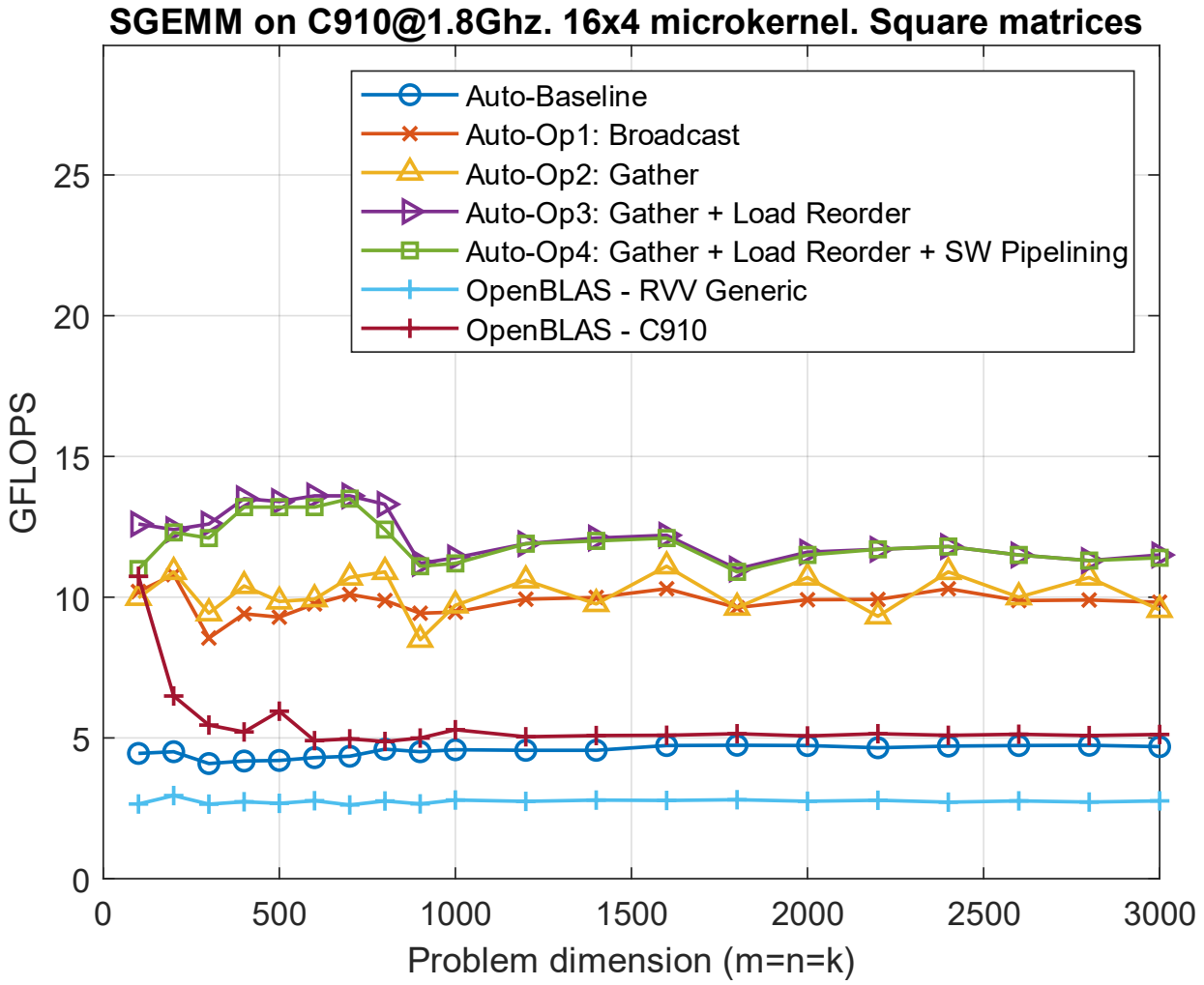




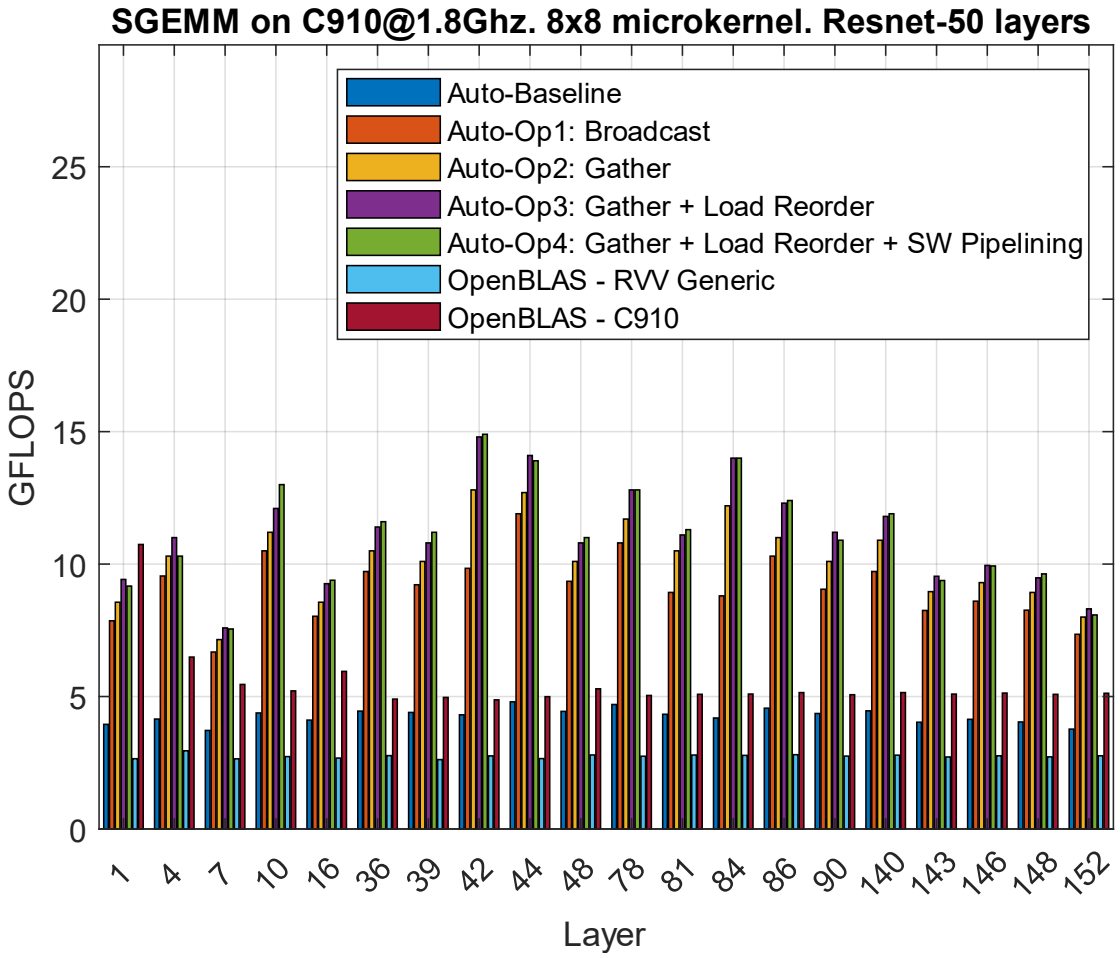


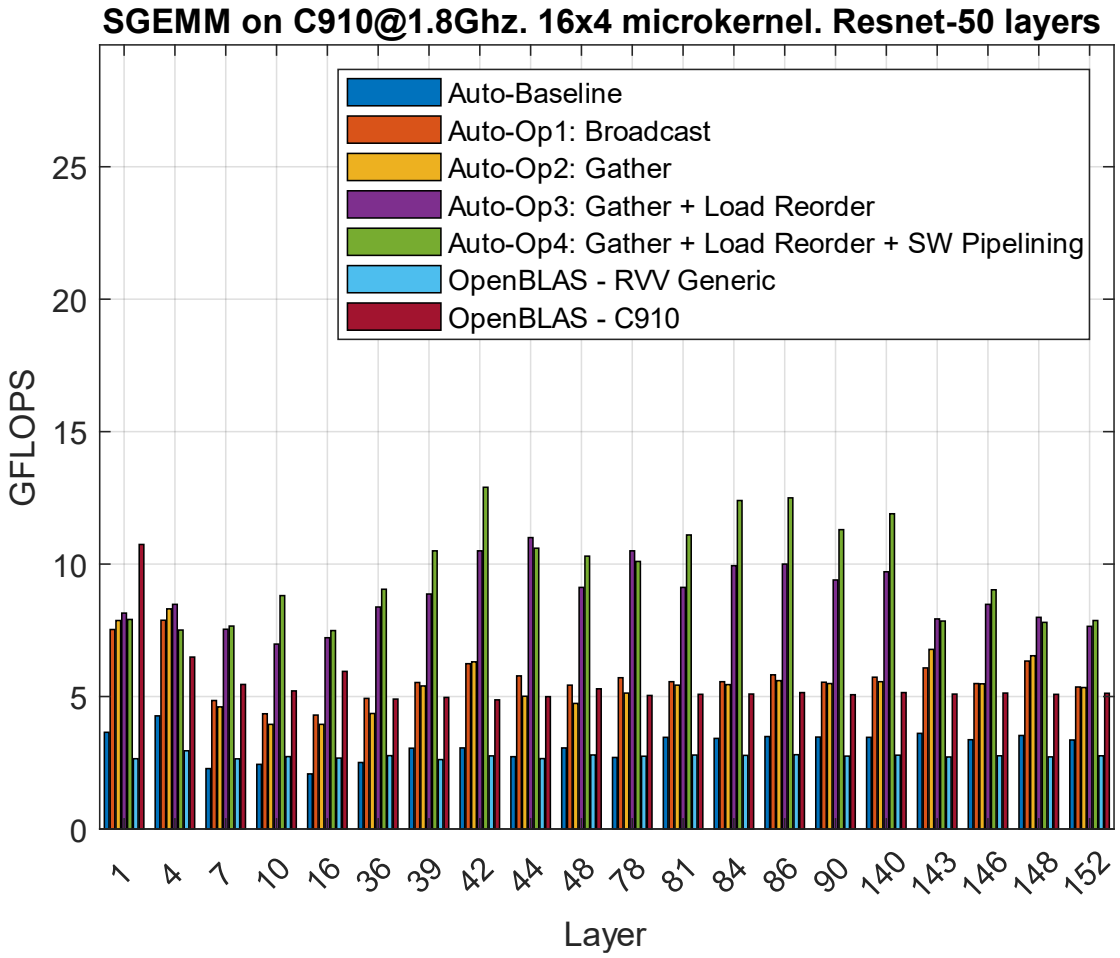
STREAM – C910 vs. Nvidia ORIN (Cortex A78AE) vs. Intel Atom (x7425E)

Results - C910, 16x4 microkernel optimizations. Square matrices

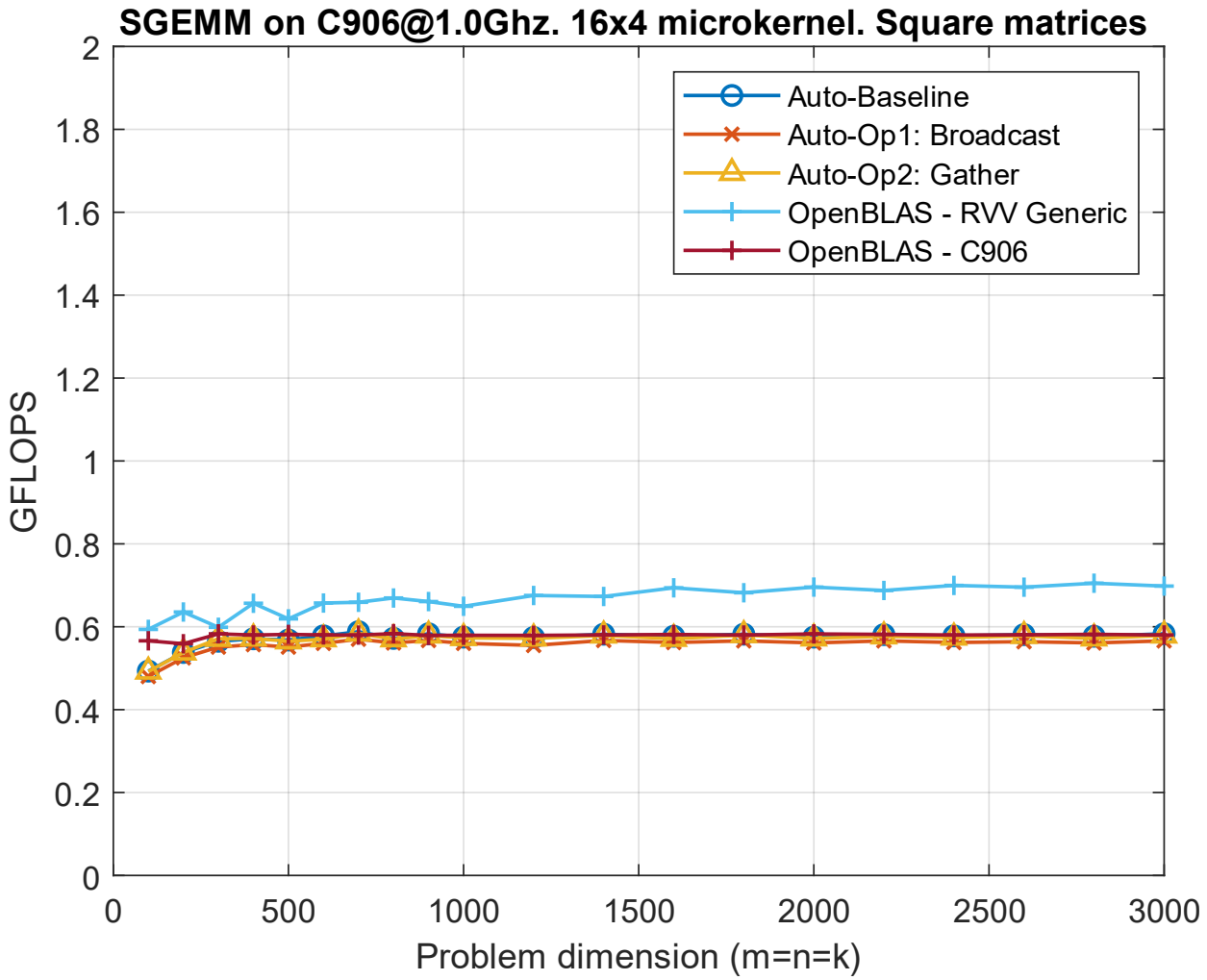


Results - C910, microkernel comparison, Resnet-50





Results - C906, 16x4 microkernel optimizations. Square matrices



Results - C906, microkernel comparison, Resnet-50