Performance analysis (and optimization) of BERT on RISC-V processors with SIMD units

Fourth International workshop on RISC-V for HPC

Francisco D. Igual

Carlos García

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

Outline

1. **Introduction**
2. **Dissecting BERT**
3. **Profiling BERT on RISC-V + RVV**
4. **Notes on optimizing BERT for RVV**
5. **Conclusions**

Introduction

Introduction

- **Transformers**
  - Introduced by Vaswani et al. in 2017
  - Widespread popularity, mainly in NLP:
    - BERT (Bidirectional Pre-trained Transformers)
    - GPT (Generative Pre-trained Transformers)
  - Also applied to computer vision tasks (classification, detection, segmentation, …)
- **Inference** in transformer-based models
  - Much cheaper and less compute-hungry than training
  - Usually run on commodity CPU-based systems. Severe time limitations
  - Cumulative impact of numerous devices –from desktop computers to wearables- running models

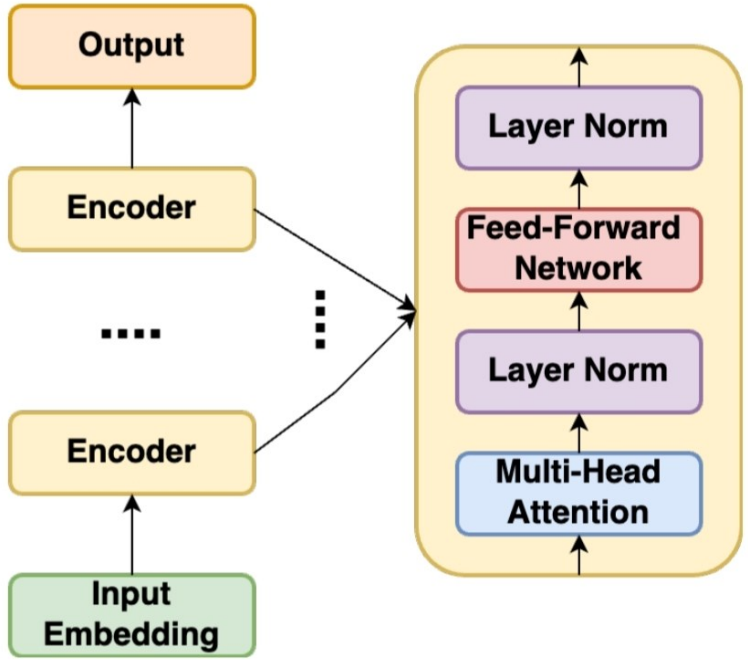
Introduction. BERT

- Transformers. In general, based on an **encoder-decoder** structure
  - Can be adapted depending on tasks that only require one of the components
  - Transformer encoders (e.g. BERT) suitable for classification
  - Transformer decoders (e.g. GPT) suitable for text generation
  - Full transformers (e.g. T5) suitable for translation/question answering
- We focus on BERT + inference
  - Useful across several NLP tasks
  - Illustrative of the potential of architectures and space for optimization in transformers
  - Inference typically deployed on low-power CPUs, typically with SIMD

Dissecting BERT

Dissecting BERT (I). General structure

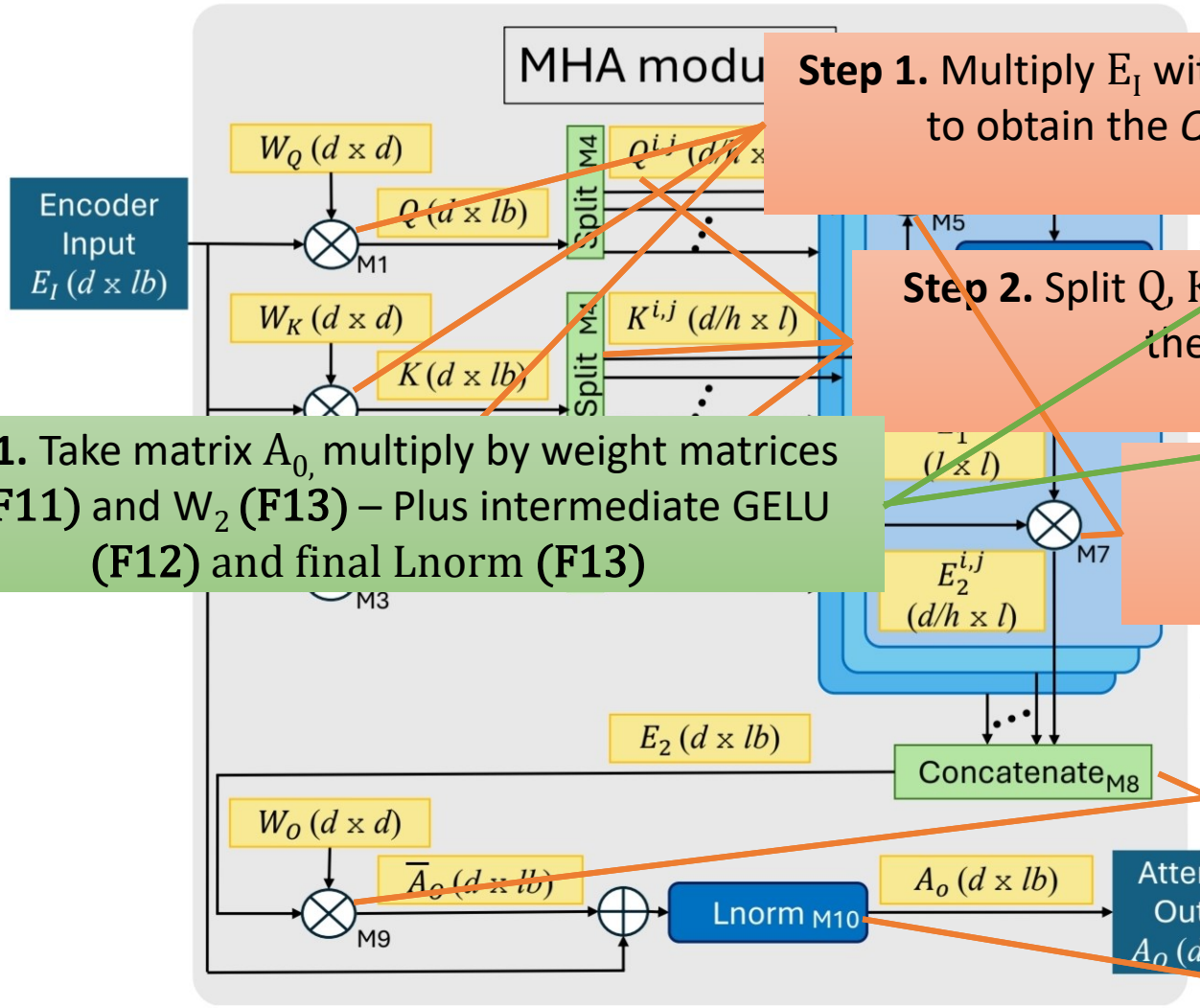
- BERT. Encoder-only transformer:
  - Input embedding
  - Several encoder layers
  - Final classification (layer adapted to specific problem)
- # of encoders depends on BERT configuration
  - BERT-tiny|base|large contain 2|12|24 encoder layers
- Each encoder layer is further decomposed into:
  - MHA (Multi-Head Attention) module
  - FFN (Feed-Forward Network) module



Dissecting BERT (II). Input embedding

- The embedding layer receives an input sentence of l tokens (words)
- Converts them to a $(d \times l)$ array
    - Each token represented as a vector of *d* embeddings
- In batch mode, it is possible to infer *b* sequences simultaneously
    - Stacking each one as a separate column of a $(d \times lb)$ input to the encoder
- E_I: encoder input, provided to the MHA module as an input

Dissecting BERT (III). MHA and FFN



Step 1. Multiply E_I with three weight matrices (W_Q, W_K, W_V) to obtain the *Query* (Q), *Key* (K) and *Value* (V).
M1, M2, M3

Step 2. Split Q, K, V into hb sub-matrices to feed the h attention heads.
M4

Step 1. Take matrix A_0, multiply by weight matrices W_1 **(F11)** and W_2 **(F13)** – Plus intermediate GELU **(F12)** and final Lnorm **(F13)**

Step 3. The hb submatrices participate in 2hb multiplications (**M5** and **M7, batched GEMM**) connected via a Softmax operation (**M6**)

Step 4. Concatenate all Softmax outputs into a new matrix E_2 **(M8)**, multiplied with weight matrix W_0 **(M9)**

Step 5. Matrix normalization to produce input to the FFN module A_0 **(M10)**

Dissecting BERT (IV). GEMM operations

			m	n	k
MHA	M1-M3.	$(Q, K, V) = (W_Q, W_K, W_V) \cdot E_I$	d	lb	d
	M4.	Split$(Q, K, V) \rightarrow (Q^{i,j}, K^{i,j}, V^{i,j})_{i=1:h}^{j=1:b}$			
		for $j = 1 : b$			
		for $i = 1 : h$			
	M5.	$\bar{E}_1^{i,j} = ((K^{i,j})^T \cdot Q^{i,j})/\sqrt{d_k}$	l	l	d/h
	M6.	$E_1^{i,j} =$Softmax $(\bar{E}_1^{i,j})$			
	M7.	$E_2^{i,j} = V^{i,j} \cdot E_1^{i,j}$	d/h	l	l
	M8.	Concatenate$(E_2^{i,j})_{i=1:h}^{j=1:b} \rightarrow E_2$			
	M9.	$\bar{A}_O = W_O \cdot E_2$	d	lb	d
	M10.	$A_O =$Lnorm $(\bar{A}_O + E_I)$			
FFN	F11.	$\bar{E}_3 = W_1 \cdot A_O$	f	lb	d
	F12.	$E_3 =$GELU $(\bar{E}_3)$			
	F13.	$\bar{E}_O = W_2 \cdot E_3$	d	lb	f
	F14.	$E_O =$Lnorm $(\bar{E}_O + A_O)$			

Param.	BERT$_B$	BERT$_L$
#Layers	12	24
d	768	1,024
h	12	16
f	3,072	4,096

Table 1. Left: Operations in the MHA and FNN modules. Right: Dimensions of BERT transformers employed in this work.

Experimental results

Platforms and experimental setup

XuantieL (C910)

- **LicheePi4a board** (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

XuantieM (C908)

- **CanMV-K230 board** (Kendryte K230 SoC)
- 1 x C908@1.6GHz
- Out-of-order superscalar
- 2 vector slices (pipelines), 128-bit (VLEN)
- RVV 1.0
- L1: 32 KiB, 2-way. L2: 0.25 MiB, 16-way

XuantieS (C906)

- **LicheeRV board** (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. L2: 0.25 MiB, 16-way

Compiler:

- GCC toolchain 10.2 (port by T-HEAD, v. 2.8.1)
- Flags: `-march=rv64imafdcv0p7_zfh_xtheadc -mabi=lp64d, and -mtune=c910|c906`

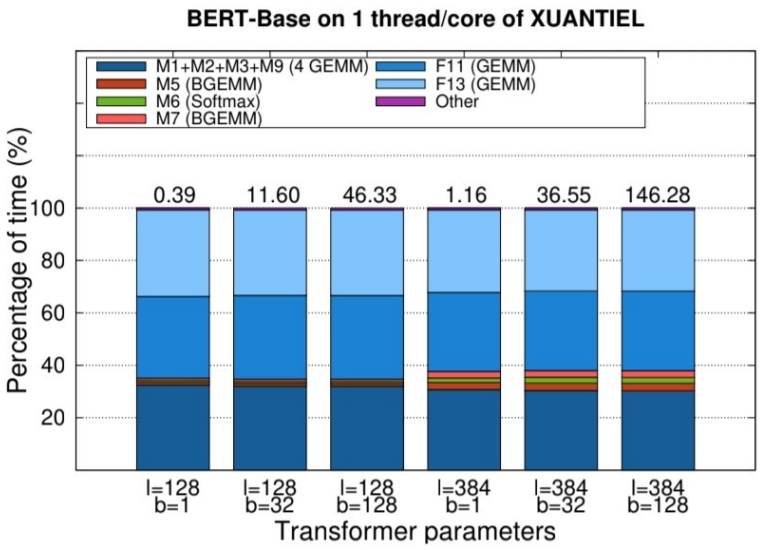
BLAS:

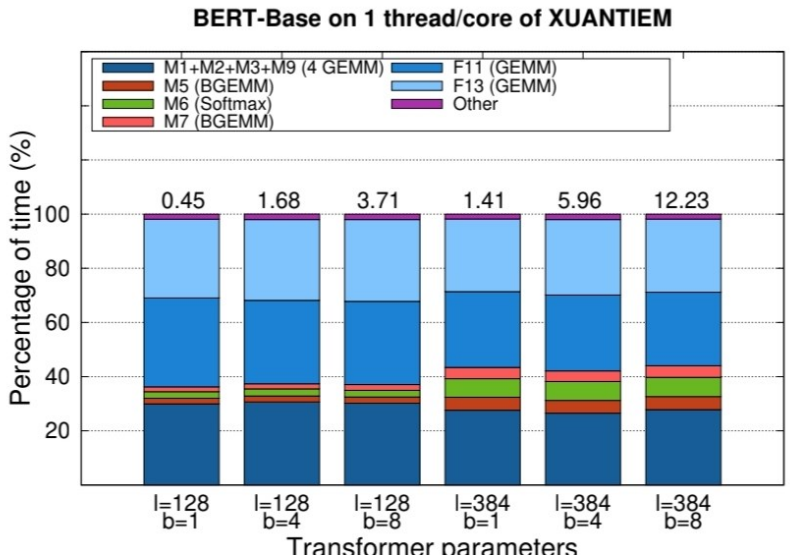
- OpenBLAS (C910/C906)
- Kendryte OpenBLAS (C908)

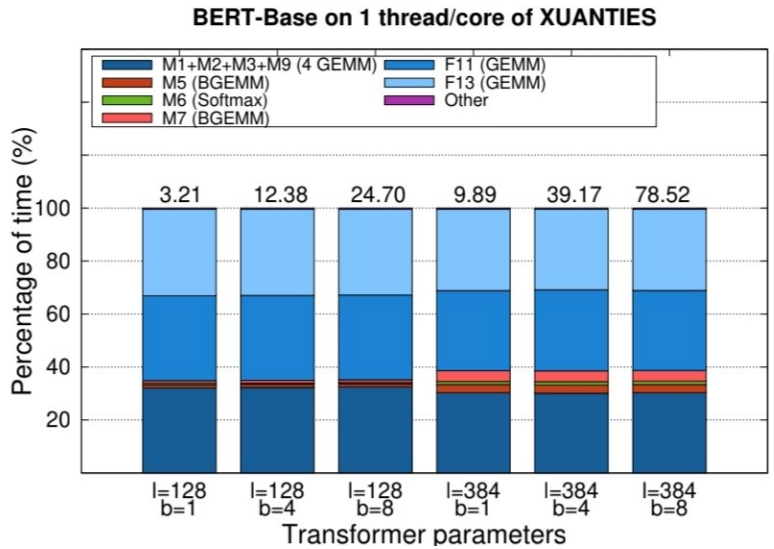
Name	Processor	Freq. (GHz)	#Cores	ISA (vector)	RAM (GB)	L1 (KB)	L2 (MB)
XuantieL	XuanTie C910	1.85	4	RVV 0.7.1	4.0 LPDDR4	64	1
XuantieM	XuanTie C008	1.60	1	RVV 1.0	0.5 LPDDR4	32	0.25
XuantieS	XuanTie C906	1.00	1	RVV 0.7.1	0.5 LPDDR4	32	0.25

Table 2. Summary of the target RISC-V processors.

BERT-Base. Performance and time breakdown

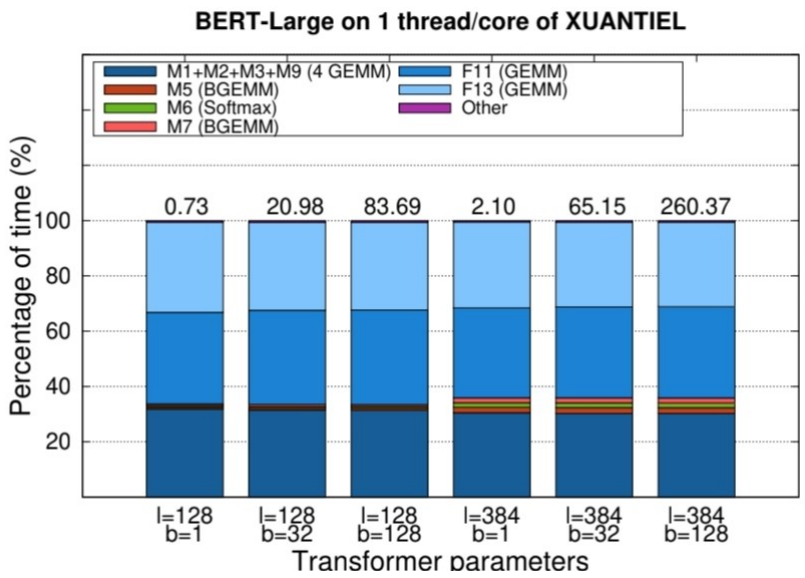


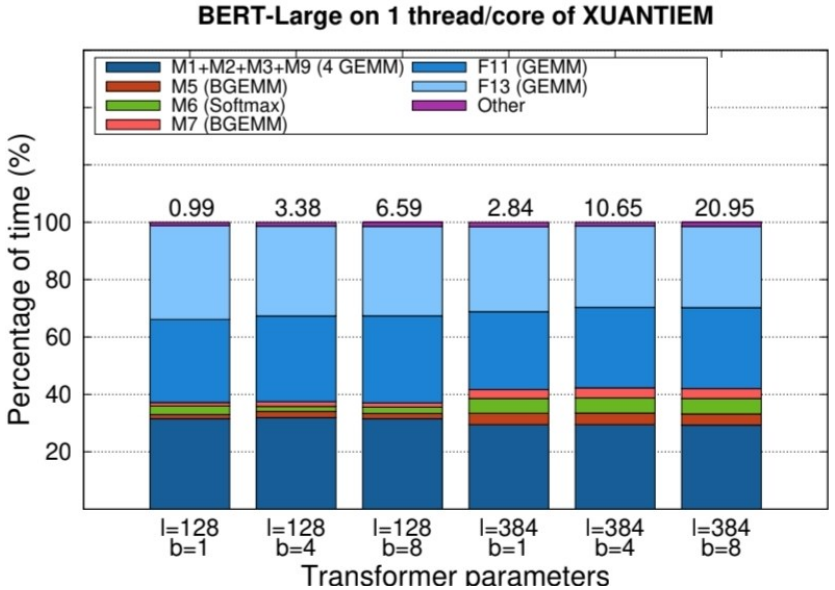


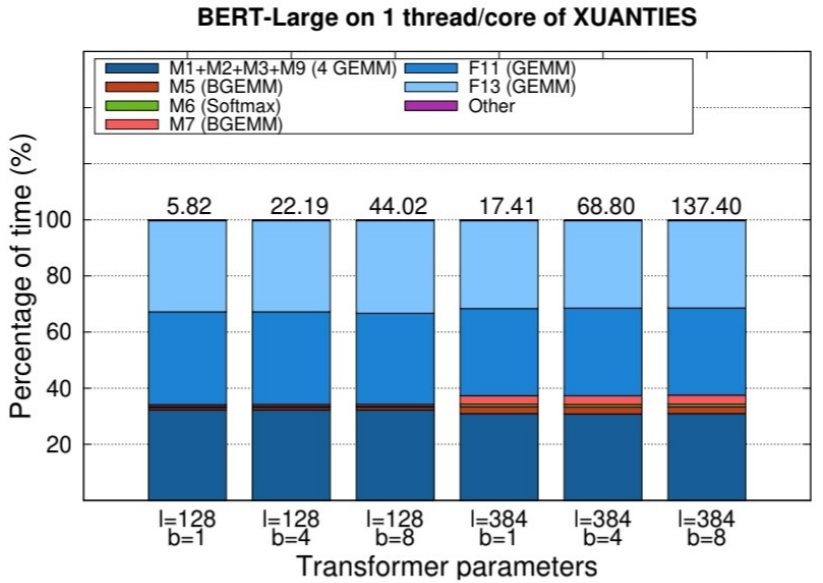


- The primary contributor to execution time is the computation of six large GEMM operations, namely **M1, M2, M3, M9, F11** and **F13**
- As b or l increase, the contribution of the two BGEMM (M5-M7) and Softmax is more prominent
- Lnorm or GELU are negligible in terms of time in all cases
- Focus only on b=1 for comparison across architectures

BERT-Large. Performance and time breakdown







- Execution time exhibits a linear growth with b
  - 4x from b=32 to b=128
- Execution time is linear for GEMM and quadratic for BGEMM and Softmax on l, resulting in a combined global effect
  - Time for GEMM operations grows by a factor of 3 from l=128 to l=384
  - Time for BGEMM operations grows by a factor of 9 from l=128 to l=384

	BERT_L $(l = 384)$			BERT_L $b = 128$		
Operation	$b = 32$	$b = 128$	*Ratio*	$l = 128$	$l = 384$	*Ratio*
M1+M2+M3+M9	19.70	78.70	3.99	26.30	78.70	2.99
M5	1.39	5.46	3.93	0.65	5.46	8.36
M6	1.10	4.37	3.97	0.48	4.37	9.07
M7	1.21	4.87	4.02	0.57	4.87	8.50
F11	21.40	85.80	4.01	28.60	85.80	3.00
F13	20.00	79.70	3.99	26.60	79.70	3.00
Other	0.35	1.47	4.20	0.49	1.47	3.00
Total	65.15	260.37	4.00	83.69	260.37	3.11

Table 3. Breakdown of execution time (in seconds) per operation on XUANTIEL.

Parallel efficiency

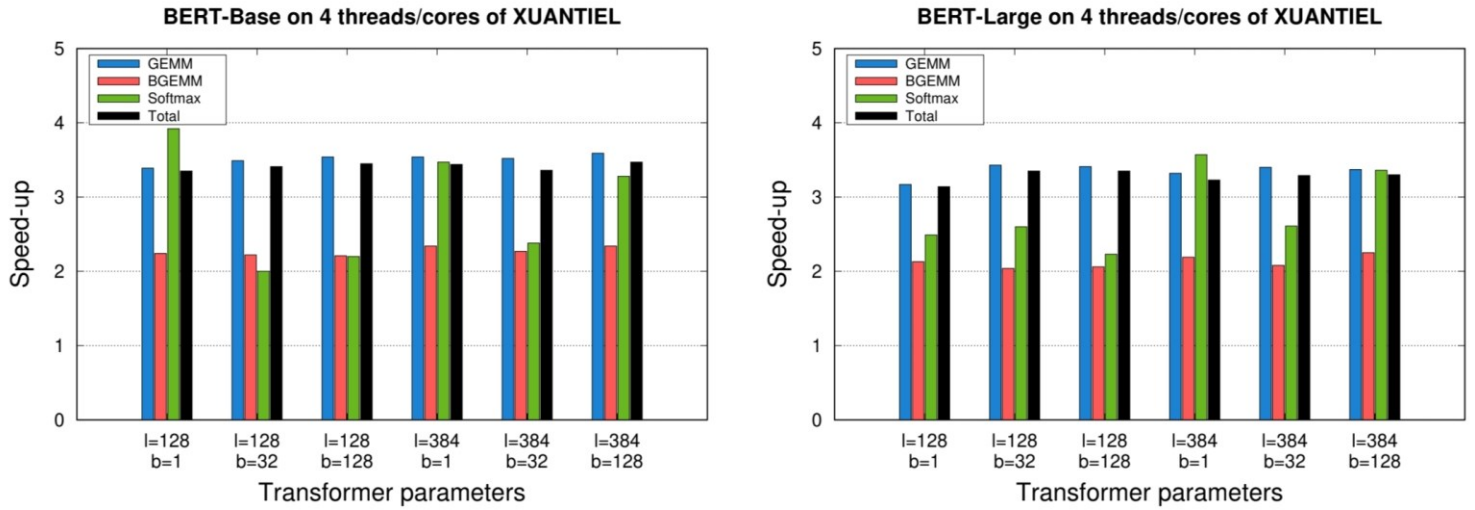


Fig. 5. Speed-up of BERT_B (left) and BERT_L (right) on XUANTIEL using 4 threads.

- GEMM offers speed-ups around 3.5 for BERT_B and 3.2 for BERT_L
- BGEMM offers speed-ups around 2 for both models (small dimensions)
- The overall improvement is that of GEMM, due to its weight in the overall computation of the transformer block

RVV. Performance comparison on RVV boards

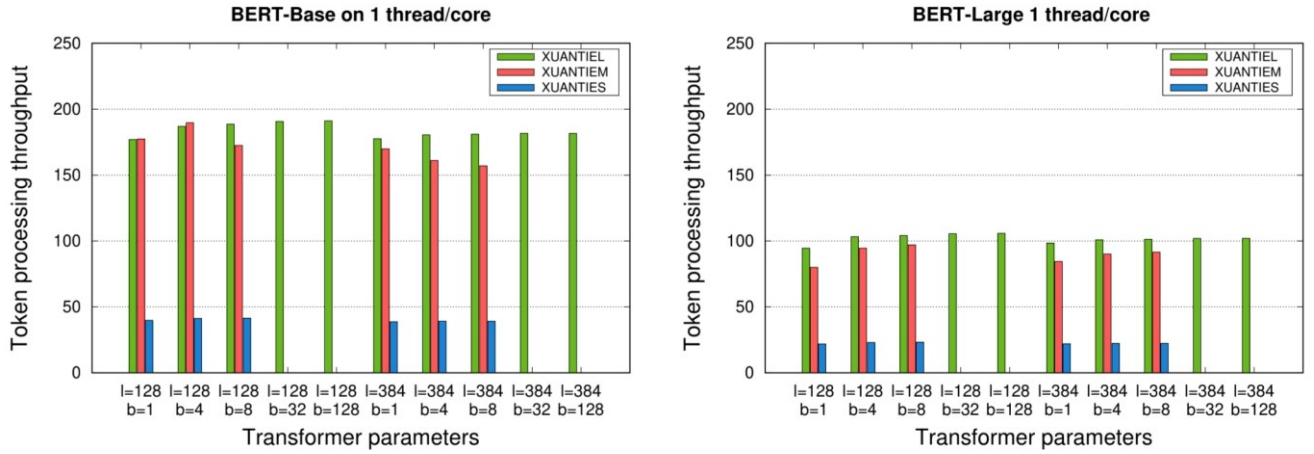


Fig. 6. Token processing throughput, normalized to processor frequency, for $BERT_B$ (left) and $BERT_L$ (right) using 1 thread.

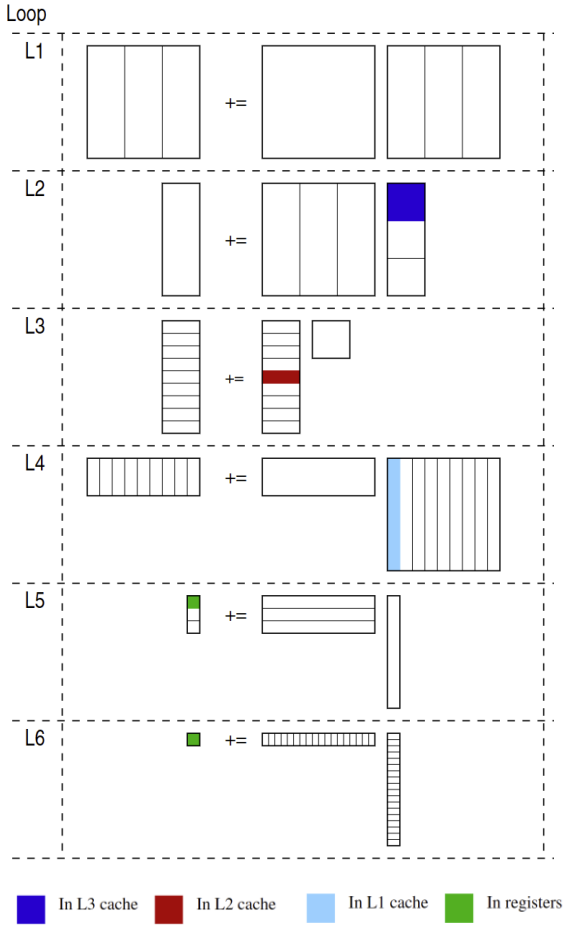
- Throughput: tokens per second
  - In our case, normalized to isolate frequency
- XuantieL is the most efficient platform in terms of throughput, followed closely by XuantieM
- Lack of memory limits the experiments with higher batch size in XuantieM and XuantieS
- Regardless of the platform, $BERT_B$ throughput is almost twice that of $BERT_L$
- Energy consumption analysis is work in progress

Notes on optimizing BERT for RISC-V

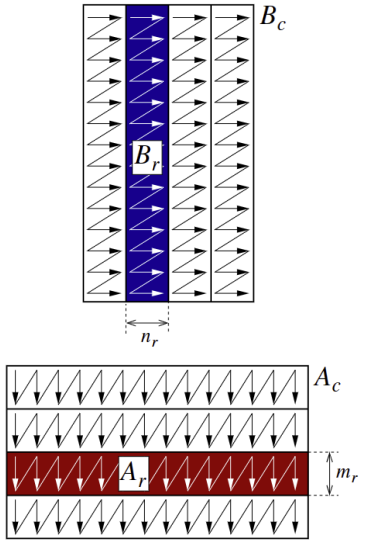
Anatomy of a high-performance GEMM

C = C + AB

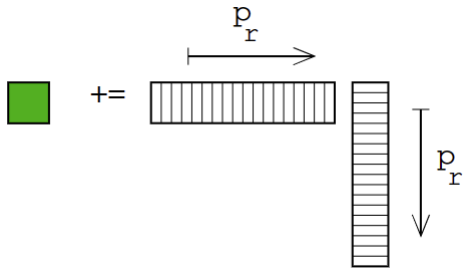
C: *mxn*; A: *mxk*; B: *kxn*




```
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

Cache Configuration Parameters (CCPs)

- **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)

GEMM micro-kernel

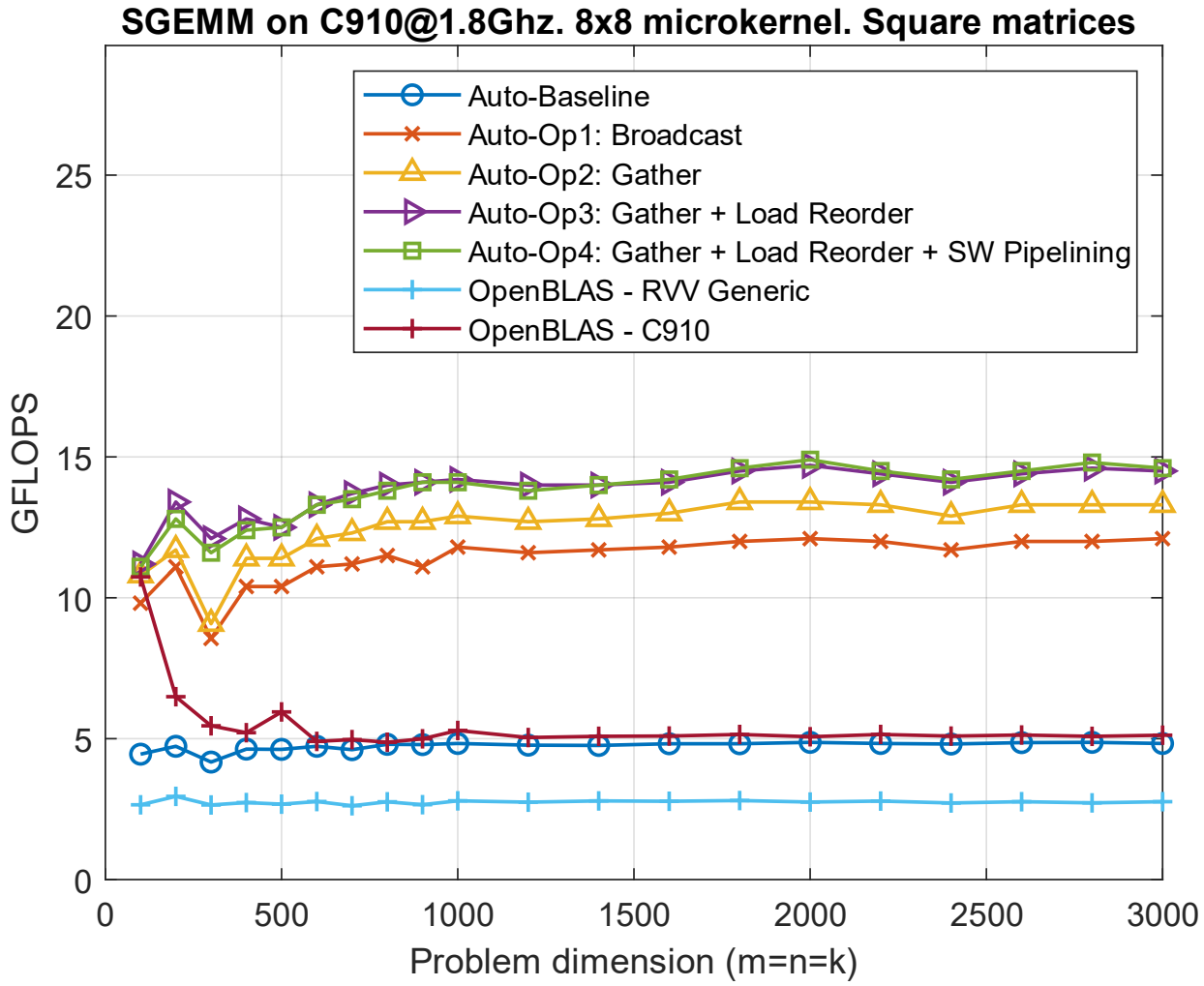
- **Typically written in assembly/intrinsics**
- **Maximize register use, flops vs. memops**
- **Generic optimizations**
  - Vectorization
  - Instruction mix/order
  - SW pipelining
  - Loop unrolling
- **Automatic generation**
  - Apache TVM [2]
  - Scripts

High-performance GEMM (Level-3 BLAS)

[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*

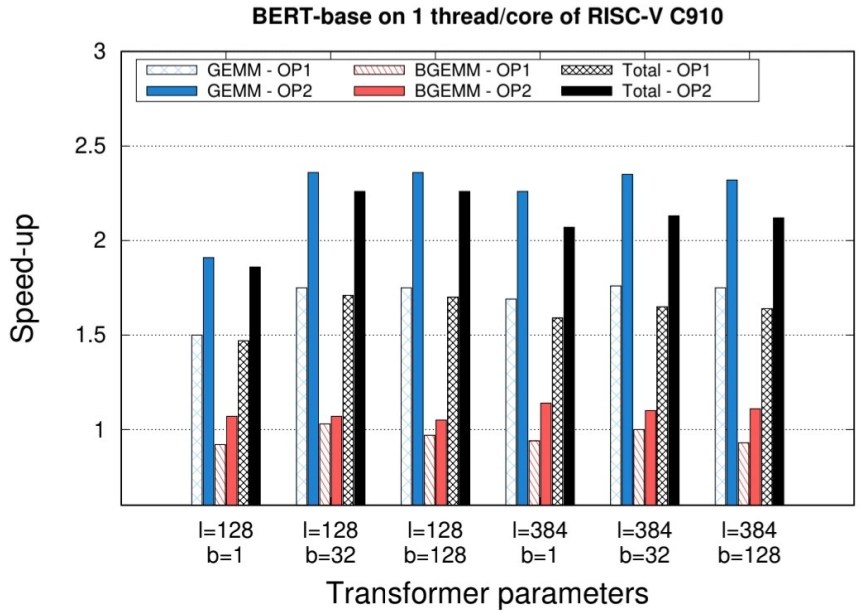
[2] *Guillermo Alaejos, Adrián Castelló, Pedro Alonso-Jordá, Francisco D. Igual, Héctor Martínez, Enrique S. Quintana-Ortí, 2023.* ***Automatic Generators for a Family of Matrix Multiplication Routines with Apache TVM.*** *CoRR abs/2310.20347 (2023)*

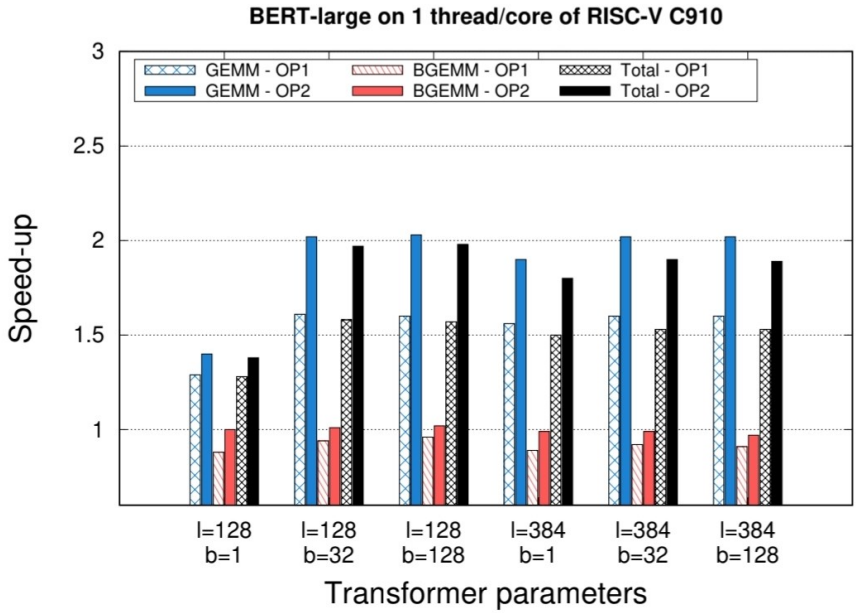
Results - C910, 8x8 microkernel, square matrices



- **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

Optimizing for single core



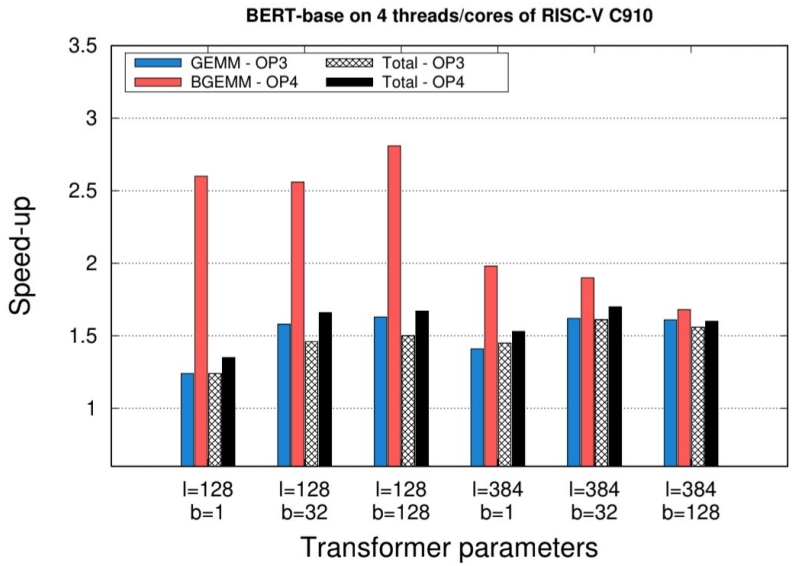


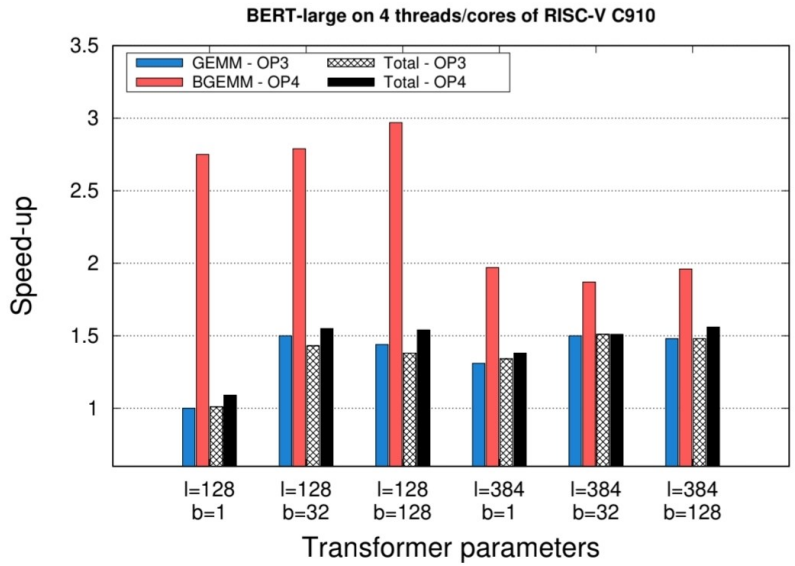
- GEMM-OP1: Selects optimal CCPs via dimension-aware analytical model on a per-GEMM basis
- GEMM-OP2: OP1 + Selects the optimal micro-kernel dimensions and automatically generates micro-kernel code

CCP selection. Tuned analytical model

	Carmel						Xuantie					
	BERT_B			BERT_L			BERT_B			BERT_L		
	m_c	n_c	k_c	m_c	n_c	k_c	m_c	n_c	k_c	m_c	n_c	k_c
M1-M3, M9	768	128	409	128	64	128	448	128	512	64	128	128
M5	128	128	64	1024	1024	128	128	128	64	448	128	512
M7	64	128	128	64	128	128	64	128	128	128	128	64
F11	1120	128	409	4096	1024	128	448	128	512	448	128	512
F13	768	128	409	1024	1120	128	448	128	512	448	128	512

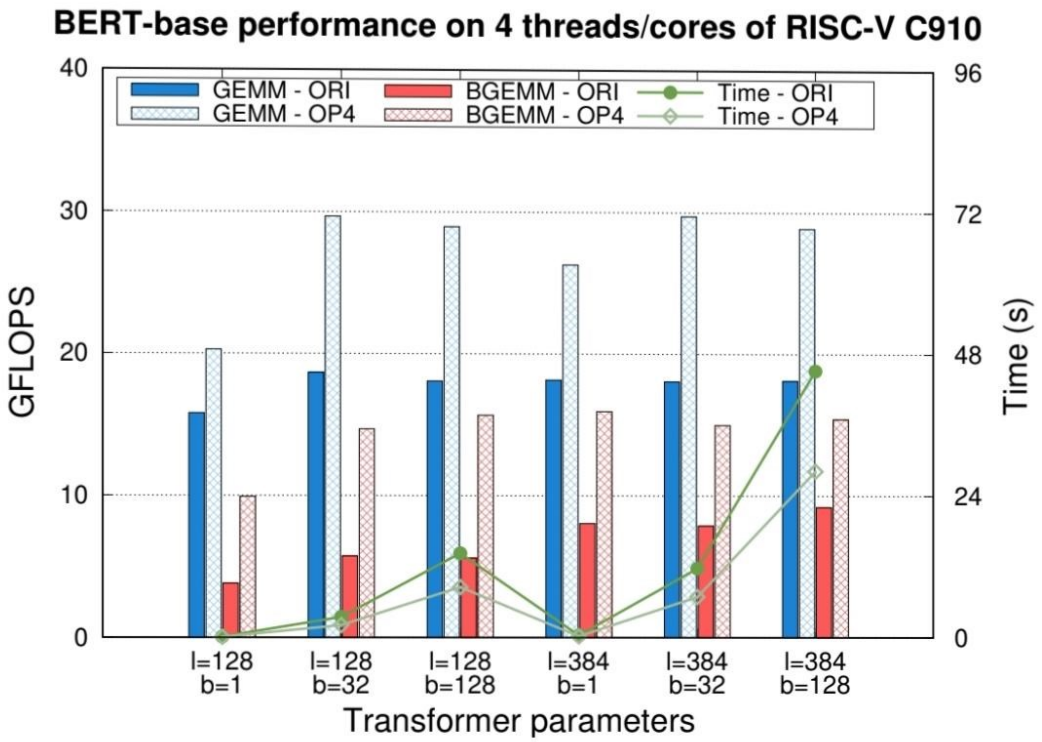
Optimizing for multi-core

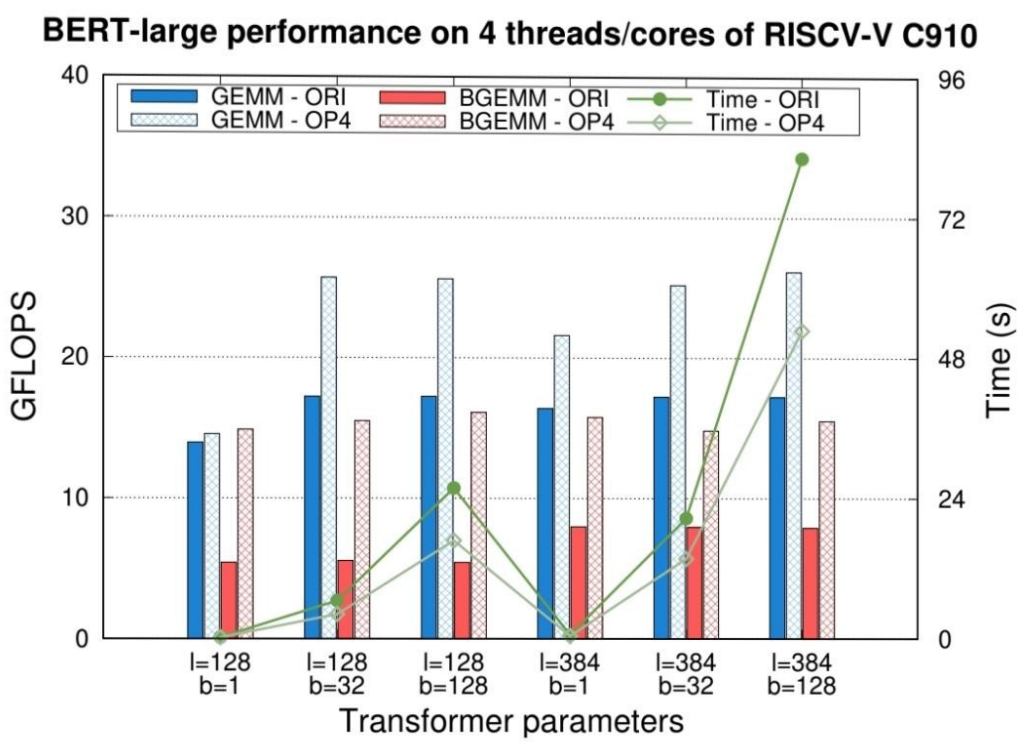




- GEMM-OP3: Replaces B3A2C0 GEMM algorithm by an A3B2C0 alternative, parallelizing loop 4 and gaining potential parallelism
- GEMM-OP4: OP3 + Collapses two loops that process the head and batch dimensions in M5-M7 and extracts parallelism externally

General optimization overview





Conclusions

Conclusions

- Provided comprehensive characterization of two representative configurations of inference with the BERT transformer
- Comparative study on three state-of-the-art RISC-V + SIMD processors
- Identified optimization challenges and performance bottlenecks
- Room for performance optimization (GEMM optimization):
  - CCP selection via refined analytical model
  - Ad-hoc microkernels
  - Specific GEMM algorithm
  - Tuned parallel versions

Performance analysis (and optimization) of BERT on RISC-V processors with SIMD units

Fourth International workshop on RISC-V for HPC

Francisco D. Igual

Carlos García

Universidad Complutense de Madrid

Héctor Martínez

Universidad de Córdoba

Sandra Catalán

Universitat Jaume I de Castelló

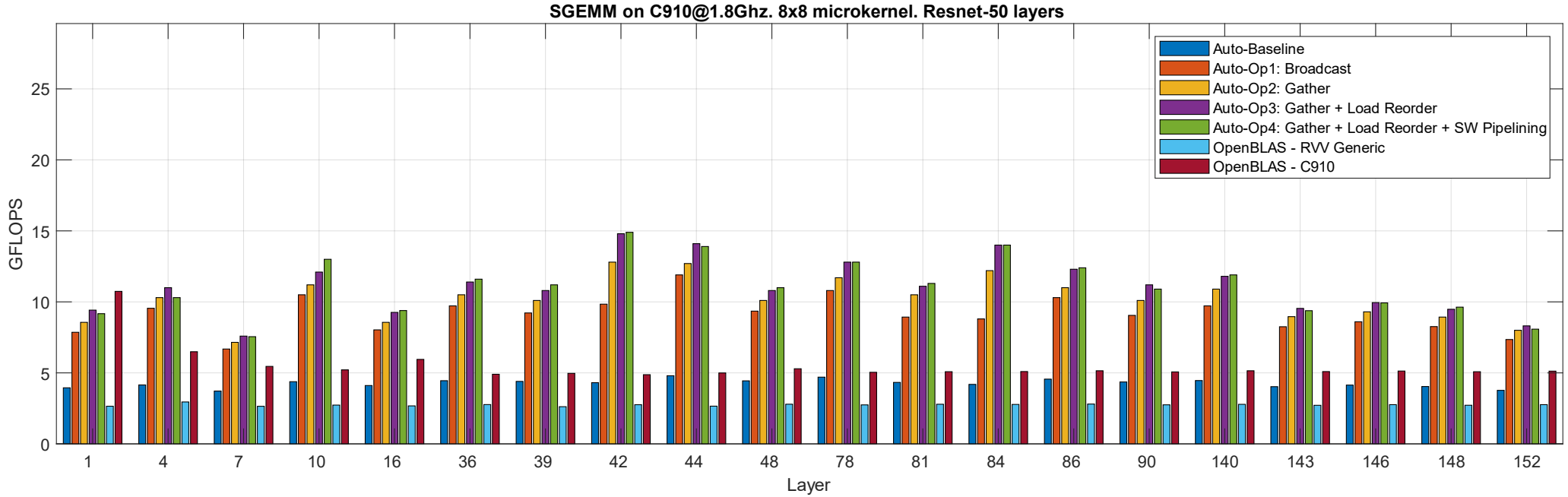
Adrián Castelló

Enrique S. Quintana-Ortí

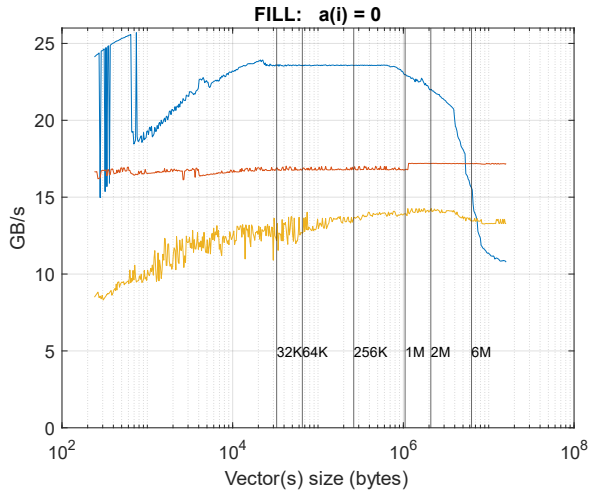
Universitat Politècnica de València

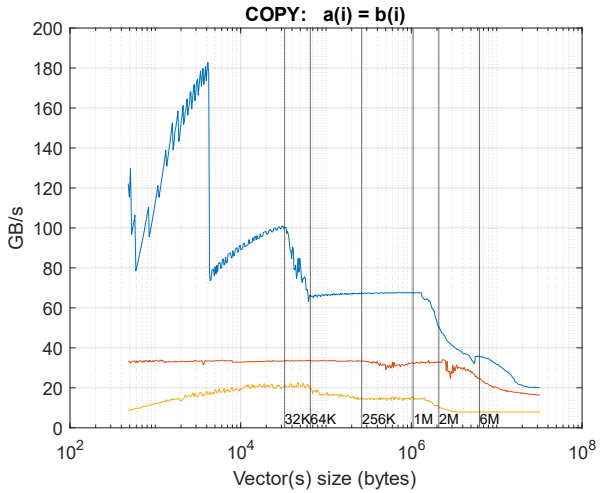
Backup slides

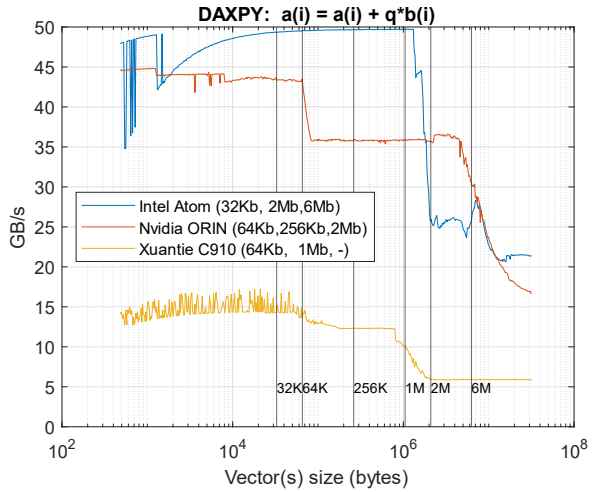
Results - C910, microkernel comparison, Resnet-50

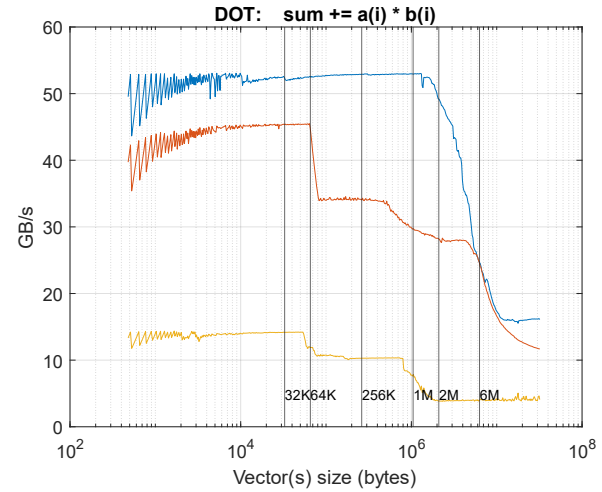


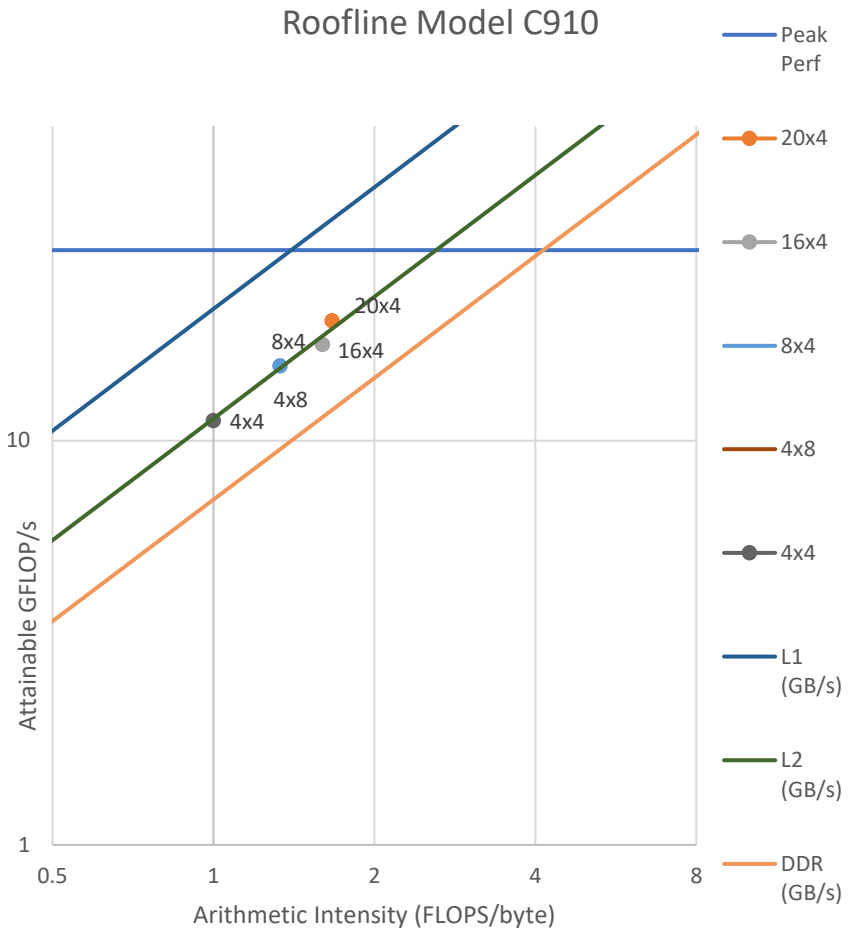
C910. STREAM – Roofline model





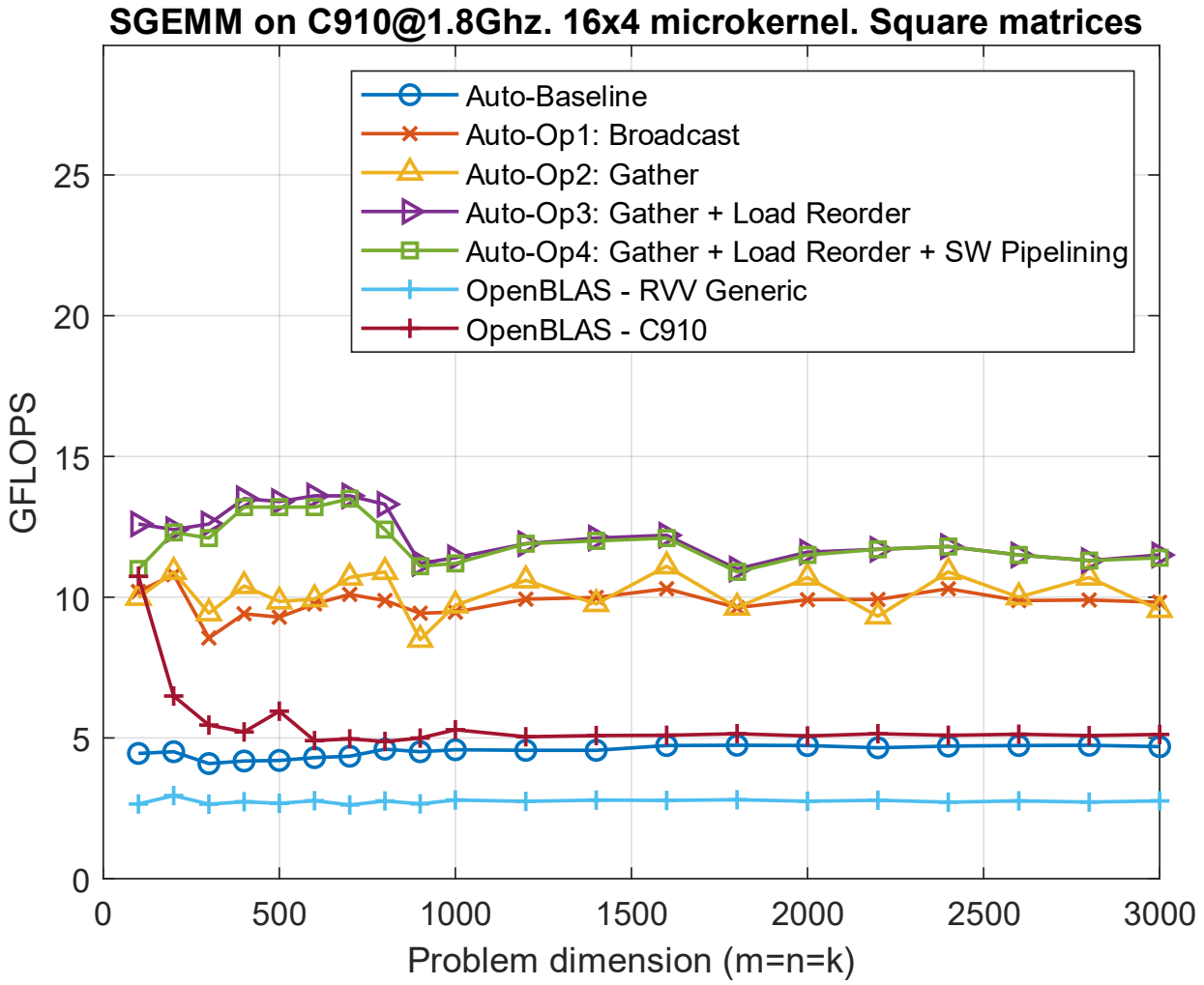




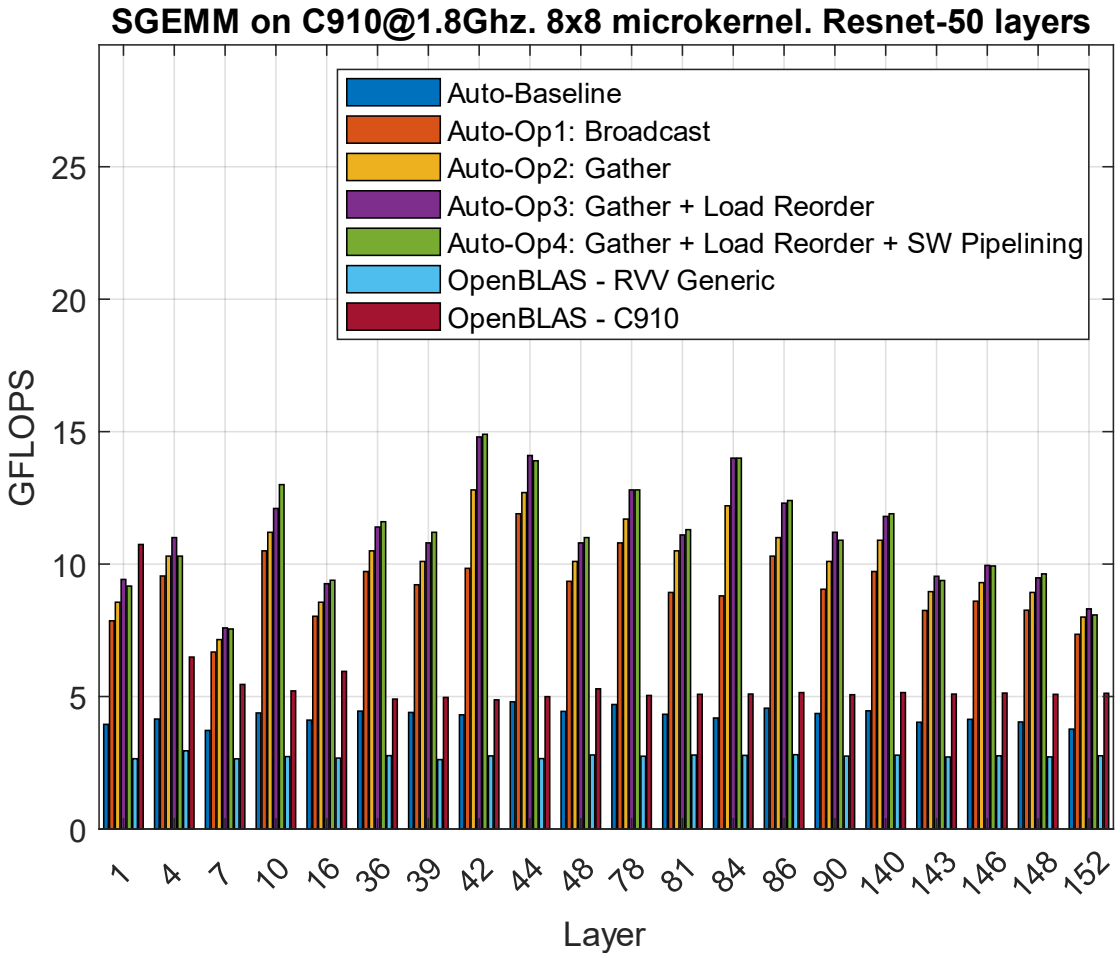


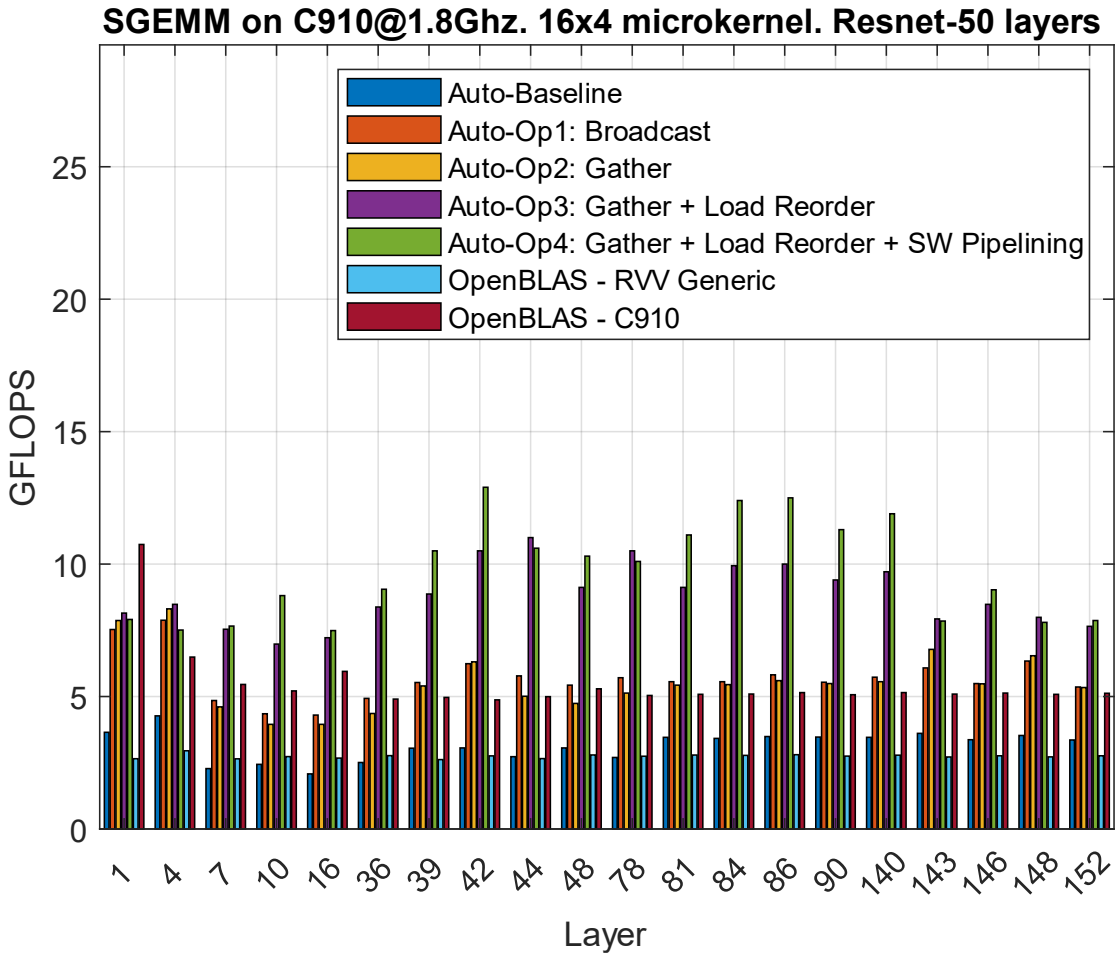
STREAM – C910 vs. Nvidia ORIN (Cortex A78AE) vs. Intel Atom (x7425E)

Results - C910, 16x4 microkernel optimizations. Square matrices

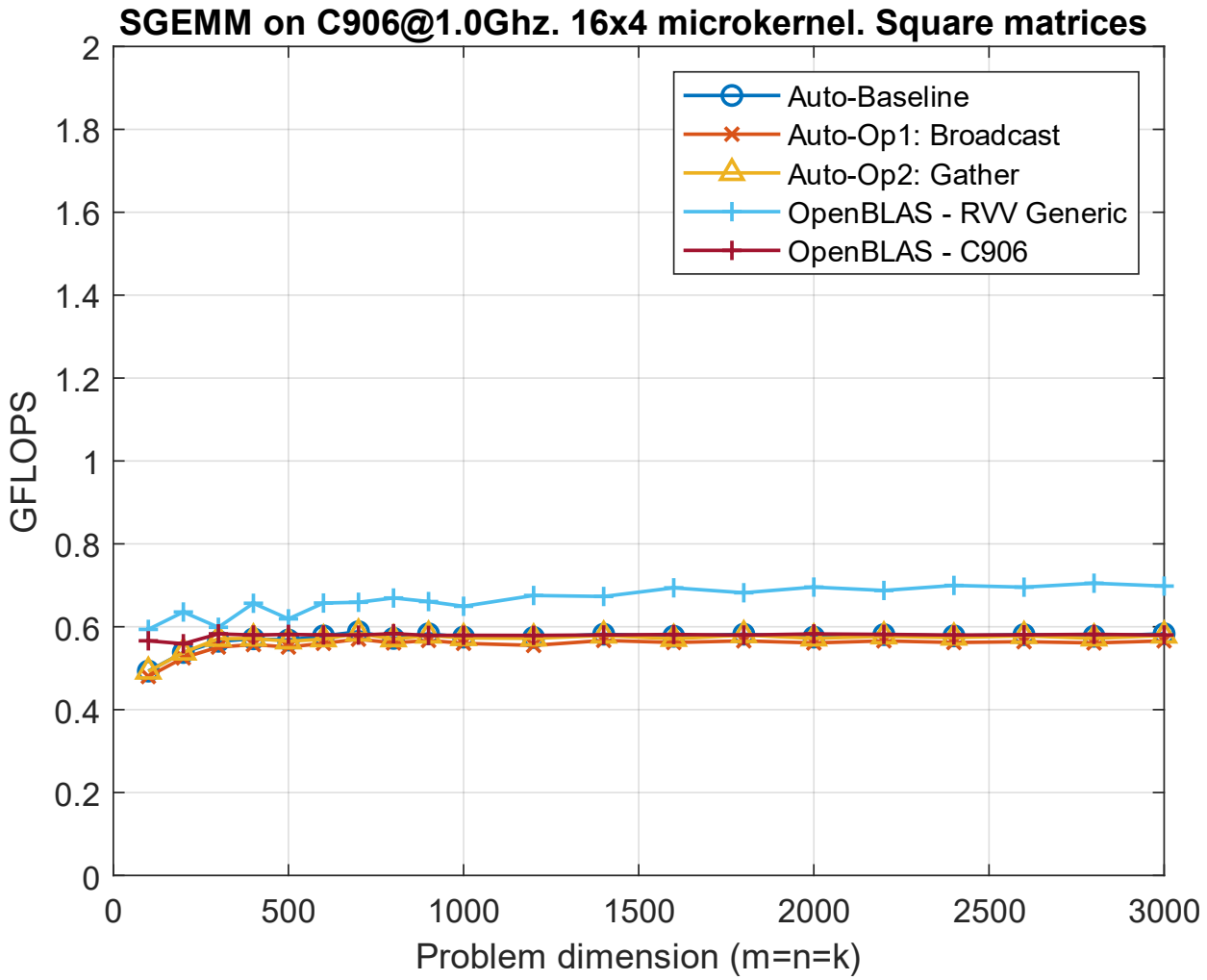


Results - C910, microkernel comparison, Resnet-50

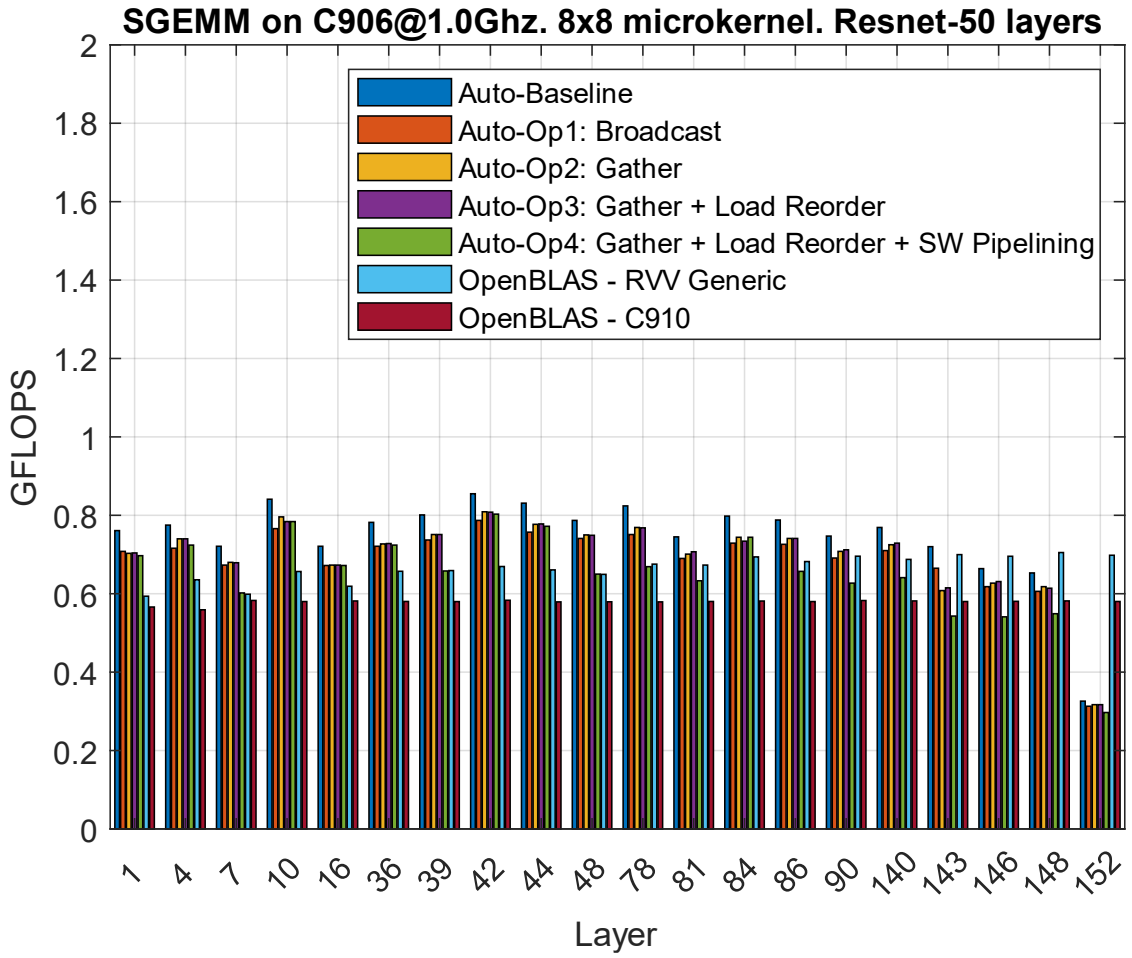


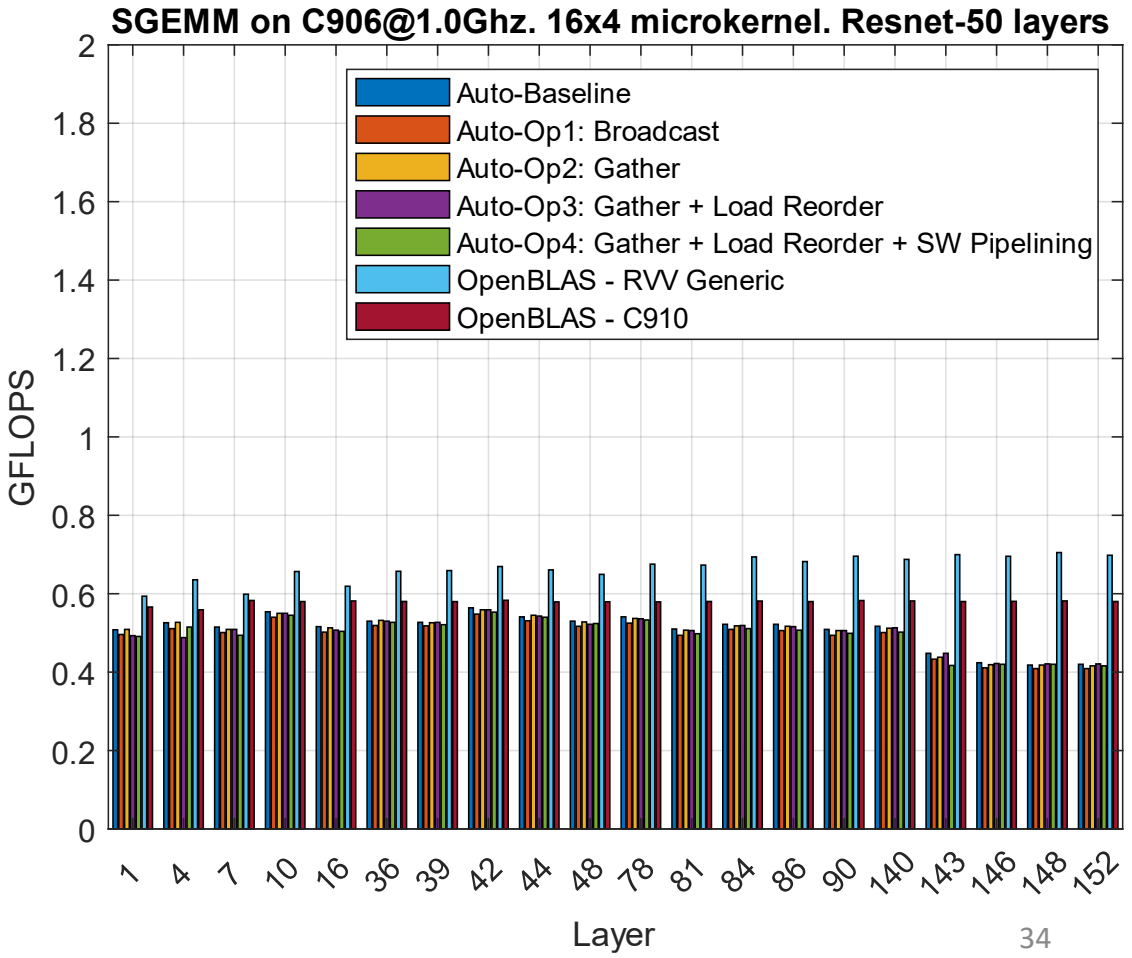


Results - C906, 16x4 microkernel optimizations. Square matrices



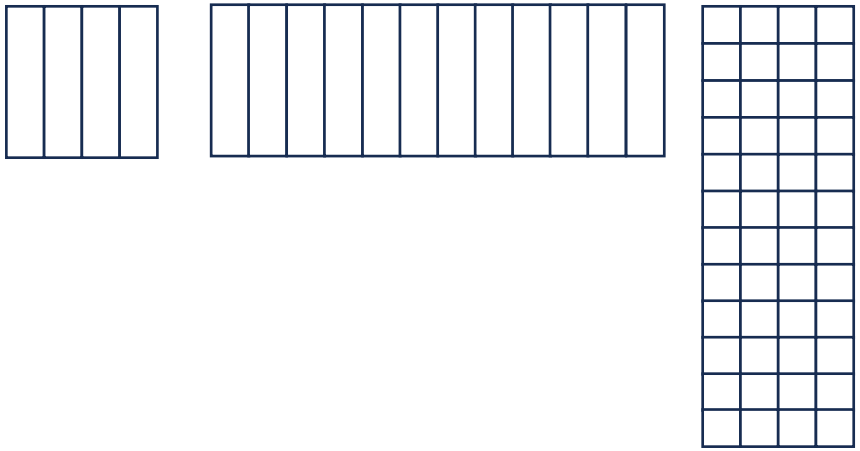
Results - C906, microkernel comparison, Resnet-50





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
```