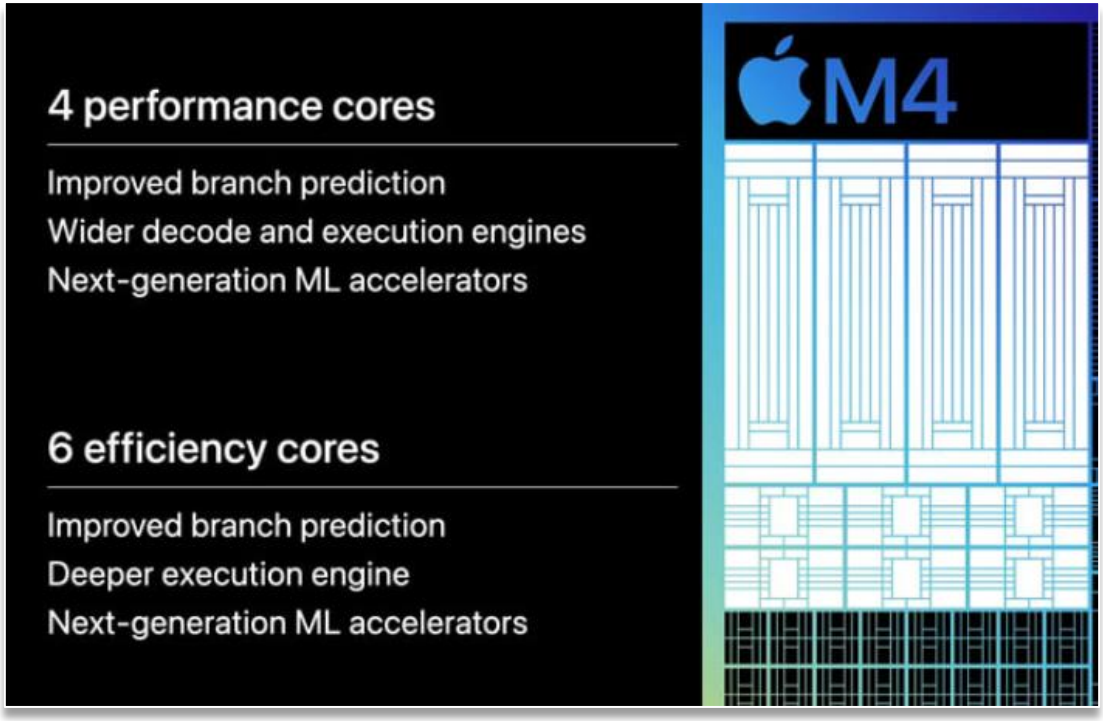
Hello SME!
Generating Fast Matrix Multiplication Kernels Using the Scalable Matrix Extension

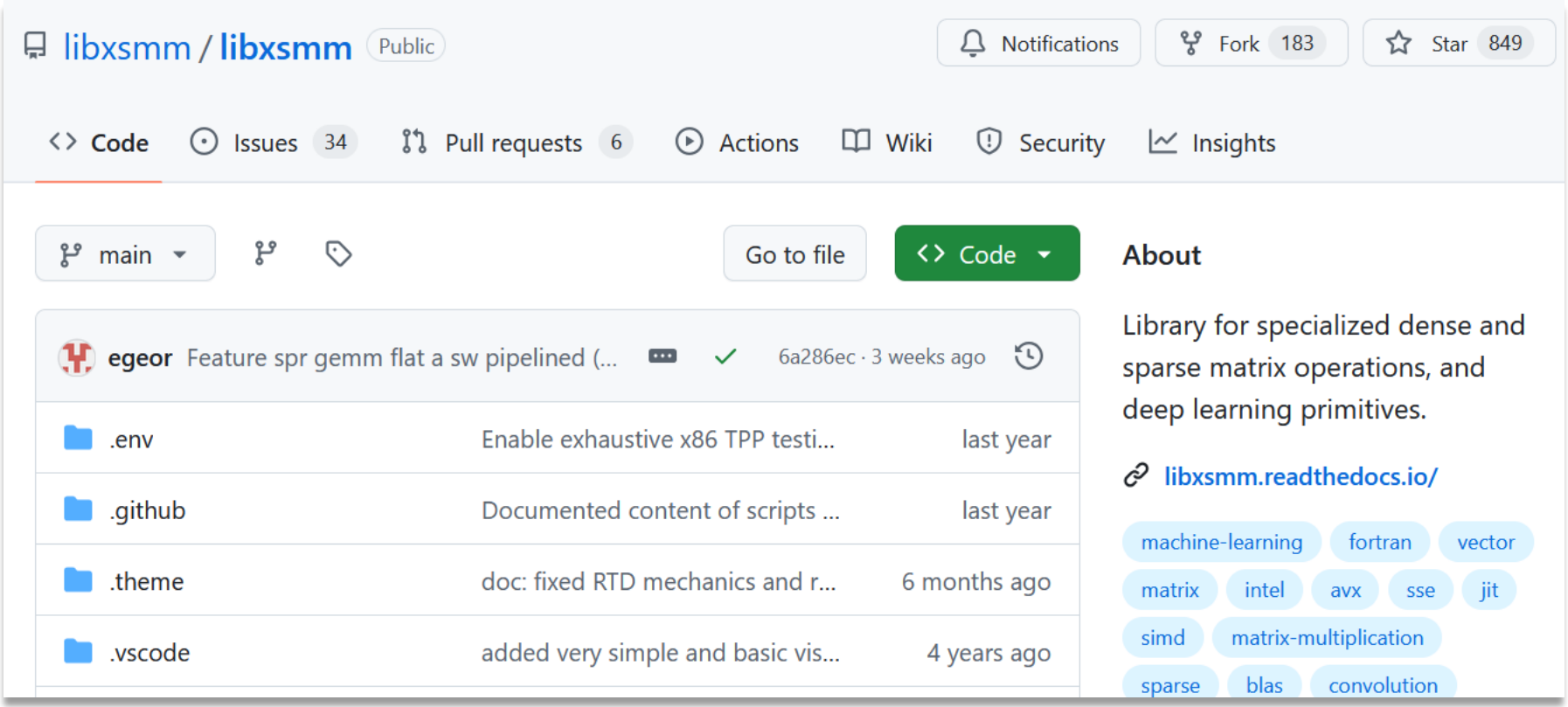
Stefan Remke, Alexander Breuer

Foto: Anne Günther

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA



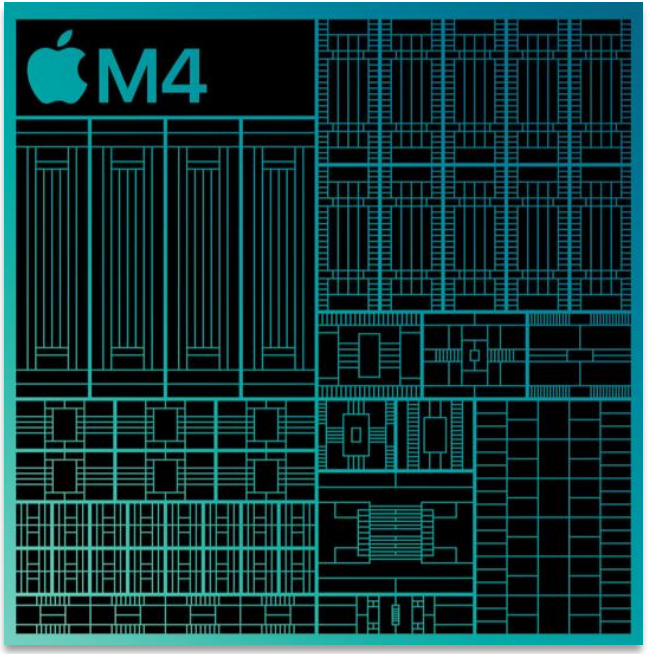
Source: https://www.apple.com/de/newsroom/2024/05/apple-introduces-m4-chip/


libxsmm / libxsmm
Public
Notifications
Fork 183
Star 849
Code
Issues 34
Pull requests 6
Actions
Wiki
Security
Insights
main
Go to file
Code
egeor Feature spr gemm flat a sw pipelined (...
6a286ec · 3 weeks ago
.env
Enable exhaustive x86 TPP testi...
last year
.github
Documented content of scripts ...
last year
.theme
doc: fixed RTD mechanics and r...
6 months ago
.vscode
added very simple and basic vis...
4 years ago
About
Library for specialized dense and sparse matrix operations, and deep learning primitives.
libxsmm.readthedocs.io/
machine-learning
fortran
vector
matrix
intel
avx
sse
jit
simd
matrix-multiplication
sparse
blas
convolution

Background

Technical Specifications

- System on a Chip (SoC)
- 3-nanometer technology
- CPU has 4 P-Cores and 6 E-Cores
- CPU supports Scalable Matrix Extension (SME)



Source: https://www.apple.com/de/newsroom/2024/05/apple-introduces-m4-chip/

Single Instruction Multiple Data (SIMD)

Vector Instructions

- Arm ASIMD/NEON
- Arm Scalable Vector Extension (SVE)
- $c \mathrel{+}= a * b$ (FMLA)

Matrix Instructions

- Arm Scalable Matrix Extension (SME)
- $C \mathrel{+}= a \otimes b$ (FMOPA)

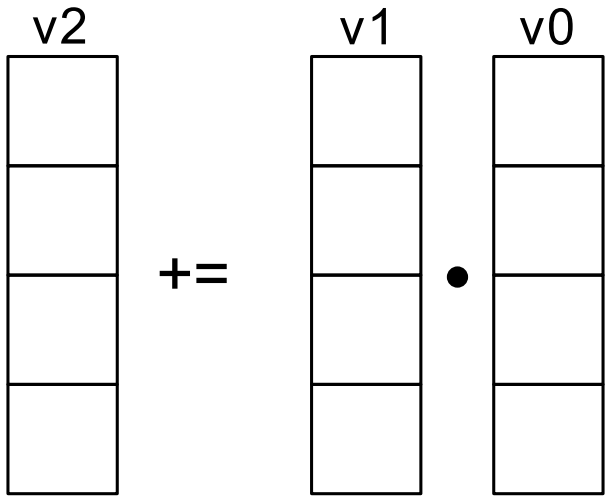
Single Instruction Multiple Data (SIMD)

Vector Instructions

- Arm ASIMD/NEON
- Arm Scalable Vector Extension (SVE)
- $c \mathrel{+}= a * b$ (FMLA)

Matrix Instructions

- Arm Scalable Matrix Extension (SME)
- $C \mathrel{+}= a \otimes b$ (FMOPA)



fmla v2.s, v1.s, v0.s

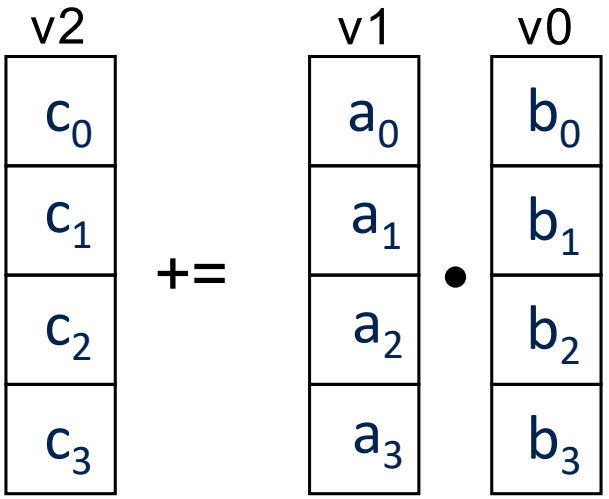
Single Instruction Multiple Data (SIMD)

Vector Instructions

- Arm ASIMD/NEON
- Arm Scalable Vector Extension (SVE)
- $c \mathrel{+}= a * b$ (FMLA)

Matrix Instructions

- Arm Scalable Matrix Extension (SME)
- $C \mathrel{+}= a \otimes b$ (FMOPA)



fmla v2.s, v1.s, v0.s

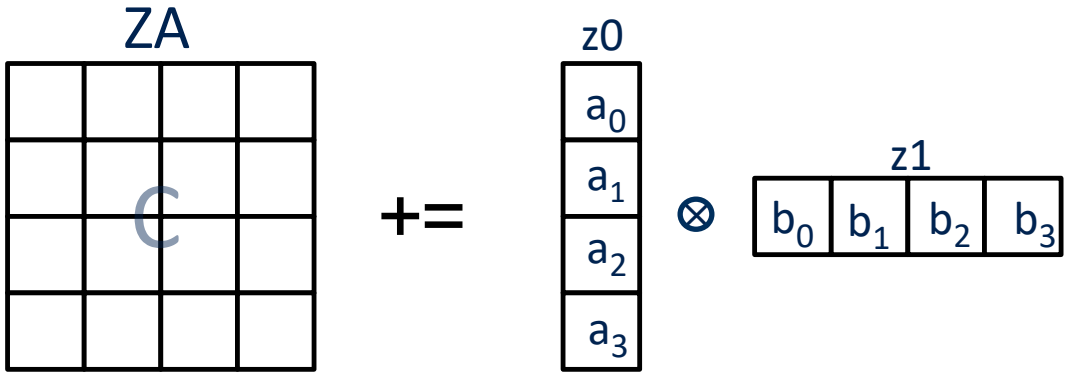
Single Instruction Multiple Data (SIMD)

Vector Instructions

- Arm ASIMD/NEON
- Arm Scalable Vector Extension (SVE)
- $c \mathrel{+}= a * b$ (FMLA)

Matrix Instructions

- Arm Scalable Matrix Extension (SME)
- $C \mathrel{+}= a \otimes b$ (FMOPA)



fmopa za0.s, p0/m, p1/m, z0.s, z1.s

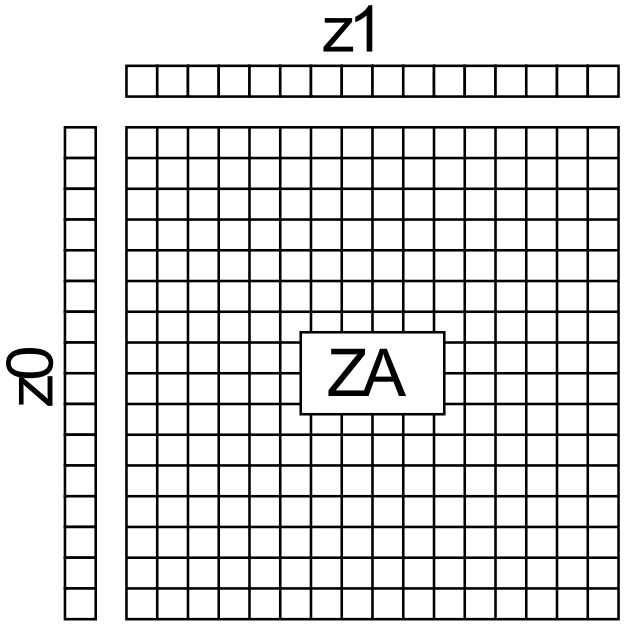
Single Instruction Multiple Data (SIMD)

Vector Instructions

- Arm ASIMD/NEON
- Arm Scalable Vector Extension (SVE)
- $c \mathrel{+}= a * b$ (FMLA)

Matrix Instructions

- Arm Scalable Matrix Extension (SME)
- $C \mathrel{+}= a \otimes b$ (FMOPA)



Microbenchmarks

- M4's SME acceleration is FP32-centric
- No speedup from FP32 to FP16/BF16
- Only 2x speedup with I8 to I32
- Streaming SVE performance is low
  → use SME2
- FP32 Multithreaded peak: 2341 GFLOPS

Instruction	Datatype		GOPS	
	In	Out	P-Core	E-Core
FMLA (Neon)	FP64	FP64	56	23
FMLA (Neon)	FP32	FP32	113	46
FMLA (Neon)	FP16	FP16	220	91
BFMMLA (Neon)	BF16	FP32	67	31
FMOPA (SME)	FP64	FP64	503	89
FMOPA (SME)	FP32	FP32	2009	357
BFMOPA (SME)	BF16	FP32	2010	357
FMOPA (SME)	FP16	FP32	2010	357
SMOPA (SME)	I16	I32	2010	357
SMOPA (SME)	I8	I32	4017	715
FMLA (SME2)	FP64	FP64	251	89
FMLA (SSVE)	FP64	FP64	16	11
FMLA (SME2)	FP32	FP32	501	179
FMLA (SSVE)	FP32	FP32	31	22

Source: https://arxiv.org/pdf/2409.18779

Microbenchmarks

- M4's SME acceleration is FP32-centric
- No speedup from FP32 to FP16/BF16
- Only 2x speedup with I8 to I32
- Streaming SVE performance is low
  → use SME2
- FP32 Multithreaded peak: 2341 GFLOPS

Instruction	Datatype		GOPS	
	In	Out	P-Core	E-Core
FMLA (Neon)	FP64	FP64	56	23
FMLA (Neon)	FP32	FP32	113	46
FMLA (Neon)	FP16	FP16	220	91
BFMMLA (Neon)	BF16	FP32	67	31
FMOPA (SME)	FP64	FP64	503	89
FMOPA (SME)	FP32	FP32	2009	357
BFMOPA (SME)	BF16	FP32	2010	357
FMOPA (SME)	FP16	FP32	2010	357
SMOPA (SME)	I16	I32	2010	357
SMOPA (SME)	I8	I32	4017	715
FMLA (SME2)	FP64	FP64	251	89
FMLA (SSVE)	FP64	FP64	16	11
FMLA (SME2)	FP32	FP32	501	179
FMLA (SSVE)	FP32	FP32	31	22

Source: https://arxiv.org/pdf/2409.18779

Microbenchmarks

- M4's SME acceleration is FP32-centric
- No speedup from FP32 to FP16/BF16
- Only 2x speedup with I8 to I32
- Streaming SVE performance is low
  → use SME2
- FP32 Multithreaded peak: 2341 GFLOPS

Instruction	Datatype		GOPS	
	In	Out	P-Core	E-Core
FMLA (Neon)	FP64	FP64	56	23
FMLA (Neon)	FP32	FP32	113	46
FMLA (Neon)	FP16	FP16	220	91
BFMMLA (Neon)	BF16	FP32	67	31
FMOPA (SME)	FP64	FP64	503	89
FMOPA (SME)	FP32	FP32	2009	357
BFMOPA (SME)	BF16	FP32	2010	357
FMOPA (SME)	FP16	FP32	2010	357
SMOPA (SME)	I16	I32	2010	357
SMOPA (SME)	I8	I32	4017	715
FMLA (SME2)	FP64	FP64	251	89
FMLA (SSVE)	FP64	FP64	16	11
FMLA (SME2)	FP32	FP32	501	179
FMLA (SSVE)	FP32	FP32	31	22

Source: https://arxiv.org/pdf/2409.18779

Microbenchmarks

- M4's SME acceleration is FP32-centric
- No speedup from FP32 to FP16/BF16
- Only 2x speedup with I8 to I32
- Streaming SVE performance is low
  → use SME2
- FP32 Multithreaded peak: 2341 GFLOPS

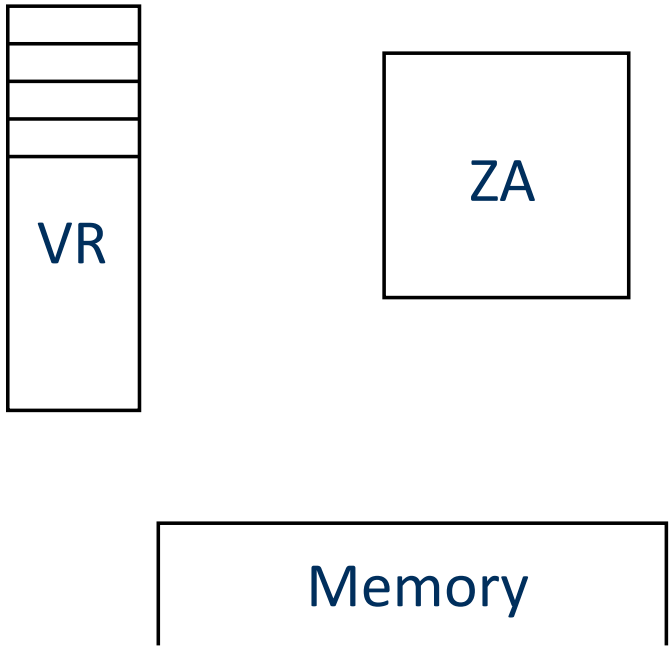
Instruction	Datatype		GOPS	
	In	Out	P-Core	E-Core
FMLA (Neon)	FP64	FP64	56	23
FMLA (Neon)	FP32	FP32	113	46
FMLA (Neon)	FP16	FP16	220	91
BFMMLA (Neon)	BF16	FP32	67	31
FMOPA (SME)	FP64	FP64	503	89
FMOPA (SME)	FP32	FP32	2009	357
BFMOPA (SME)	BF16	FP32	2010	357
FMOPA (SME)	FP16	FP32	2010	357
SMOPA (SME)	I16	I32	2010	357
SMOPA (SME)	I8	I32	4017	715
FMLA (SME2)	FP64	FP64	251	89
FMLA (SSVE)	FP64	FP64	16	11
FMLA (SME2)	FP32	FP32	501	179
FMLA (SSVE)	FP32	FP32	31	22

Source: https://arxiv.org/pdf/2409.18779

Microbenchmarks

Load to ZA tile

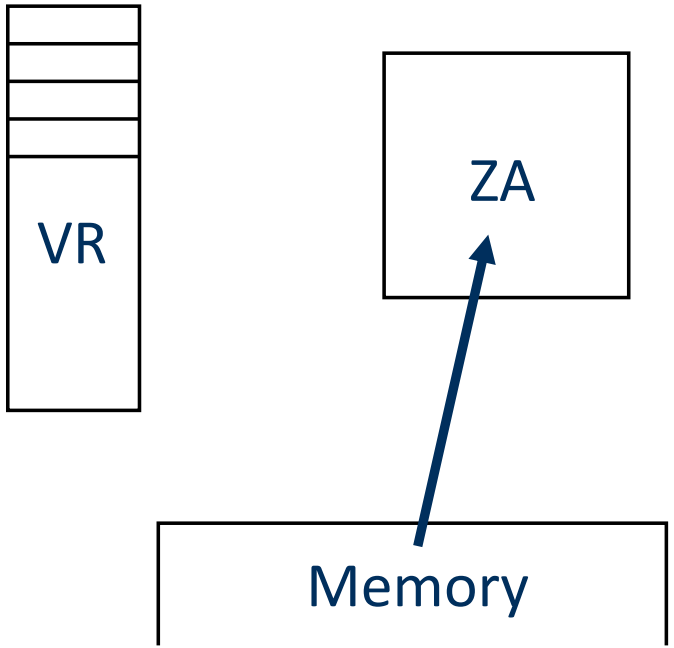
- Two strategies:
  - LDR
  - LD1W and MOVA
- LDR loads 64 byte to ZA
- LD1W loads up to 256 byte into VR and MOVA copy them to ZA



Microbenchmarks

Load to ZA tile

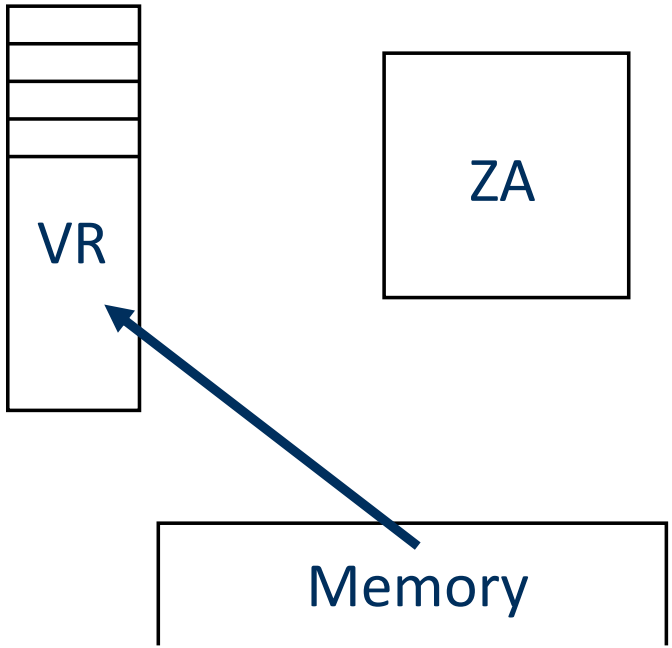
- Two strategies:
  - LDR
  - LD1W and MOVA
- LDR loads 64 byte to ZA
- LD1W loads up to 256 byte into VR and MOVA copy them to ZA



Microbenchmarks

Load to ZA tile

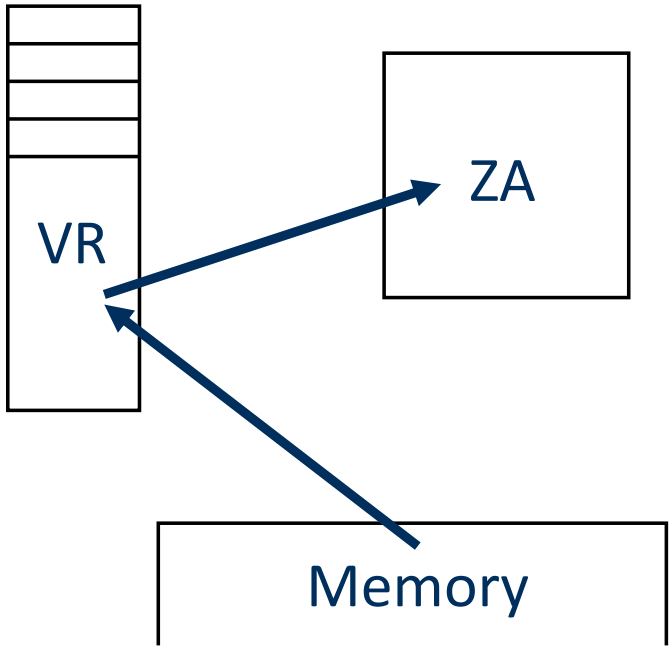
- Two strategies:
  - LDR
  - LD1W and MOVA
- LDR loads 64 byte to ZA
- LD1W loads up to 256 byte into VR and MOVA copy them to ZA



Microbenchmarks

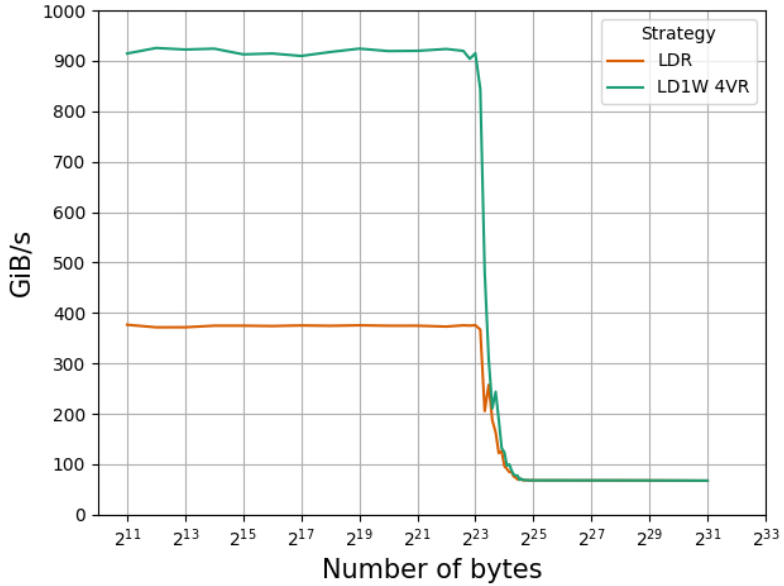
Load to ZA tile

- Two strategies:
  - LDR
  - LD1W and MOVA
- LDR loads 64 byte to ZA
- LD1W loads up to 256 byte into VR and MOVA copy them to ZA

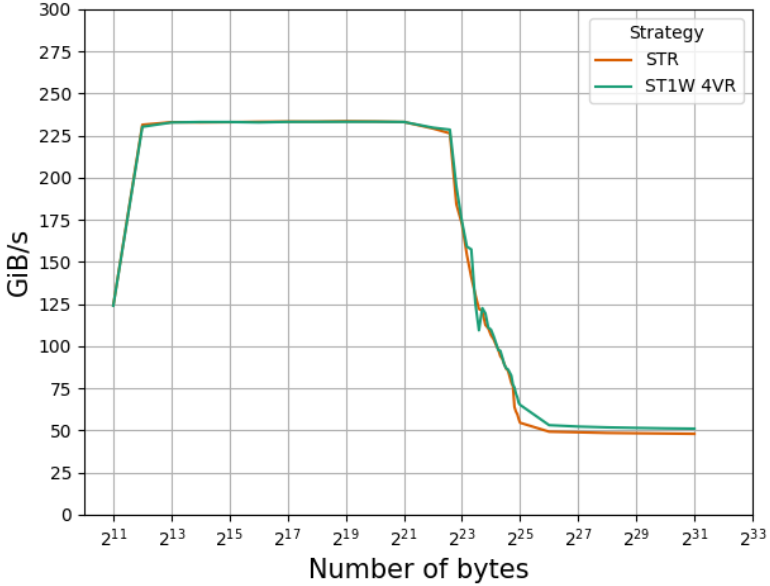


Microbenchmarks

Load to ZA tile



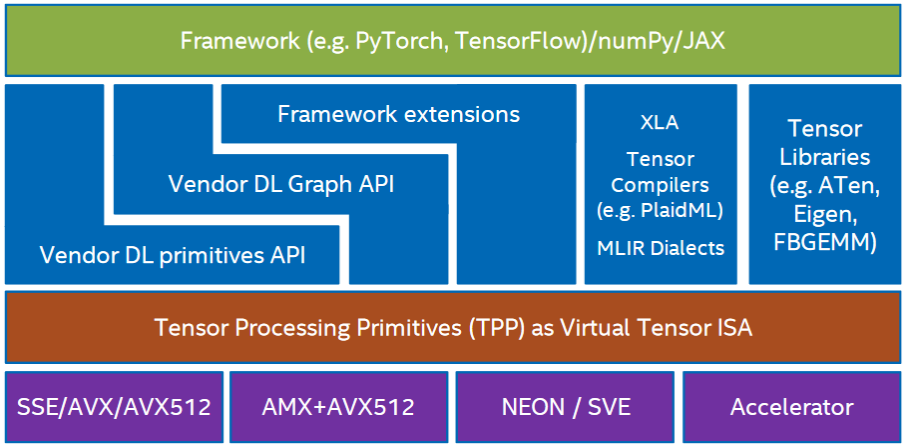
Store from ZA tile



Just-in-time Code Generation

LIBXSMM library

- Set of tensor processing primitives
- High-level user does not have to worry about hardware
- Tailor to specific microarchitecture (x86, ARM)
- Write machine code to memory and make it executable

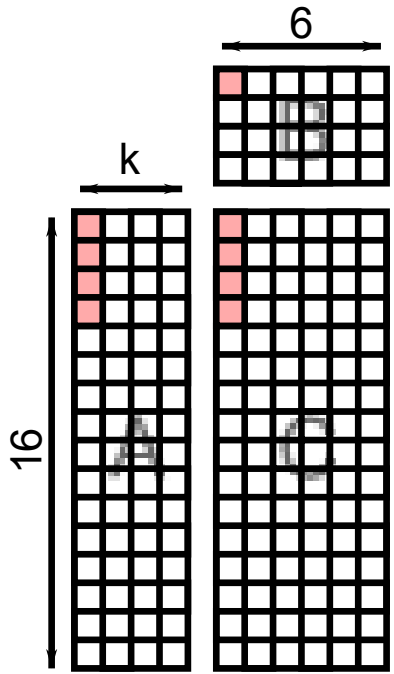


Source: https://arxiv.org/pdf/2104.05755

Just-in-time Code Generation

LIBXSMM using NEON

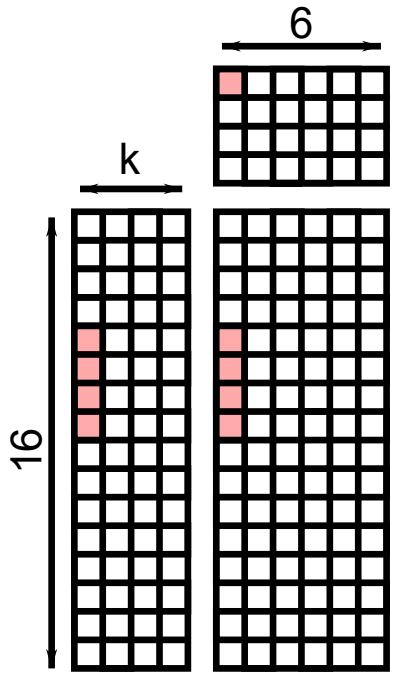
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 8 FLOPS per Instruction (FMLA)
- 24 vector registers for accumulator C
- 8 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using NEON

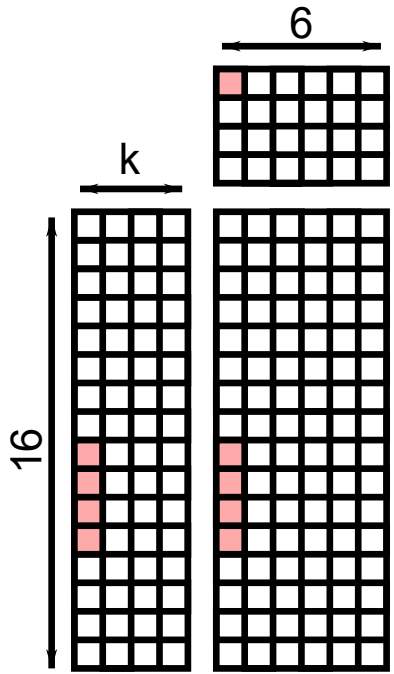
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 8 FLOPS per Instruction (FMLA)
- 24 vector registers for accumulator C
- 8 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using NEON

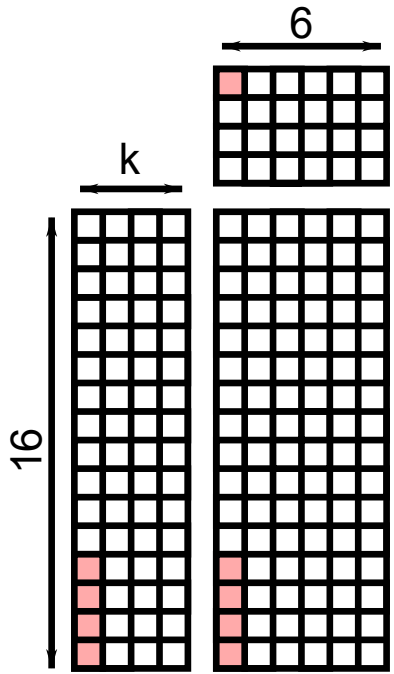
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 8 FLOPS per Instruction (FMLA)
- 24 vector registers for accumulator C
- 8 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using NEON

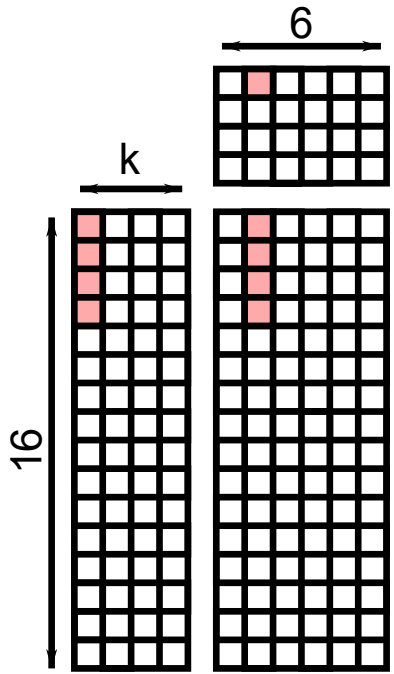
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 8 FLOPS per Instruction (FMLA)
- 24 vector registers for accumulator C
- 8 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using NEON

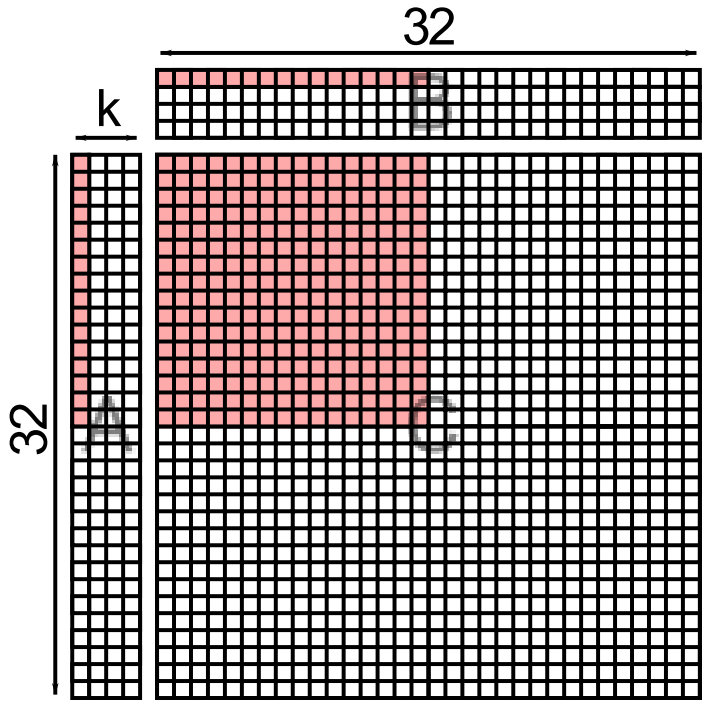
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 8 FLOPS per Instruction (FMLA)
- 24 vector registers for accumulator C
- 8 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using Scalable Matrix Extension

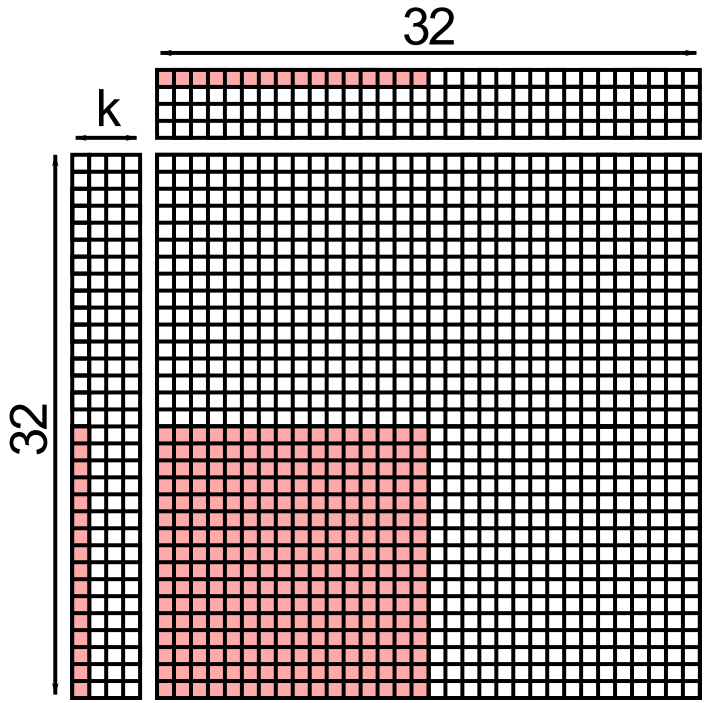
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 512 FLOPS per instruction (FMOPA)
- ZA tile holds 32x32 values
- 4 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using Scalable Matrix Extension

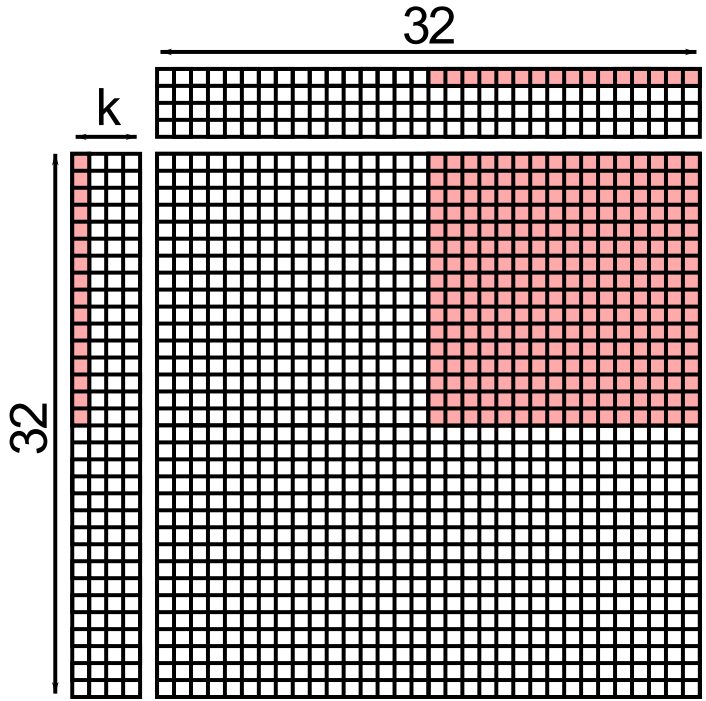
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 512 FLOPS per instruction (FMOPA)
- ZA tile holds 32x32 values
- 4 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using Scalable Matrix Extension

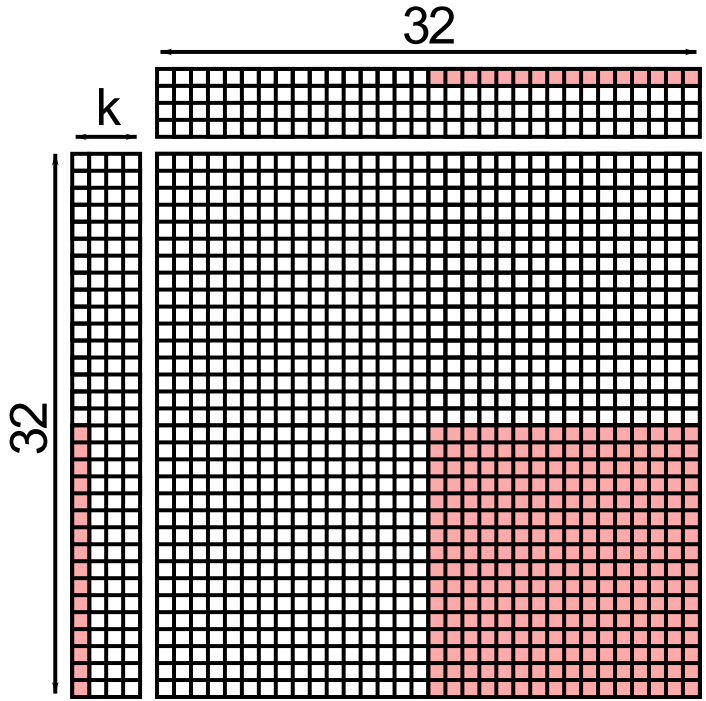
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 512 FLOPS per instruction (FMOPA)
- ZA tile holds 32x32 values
- 4 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using Scalable Matrix Extension

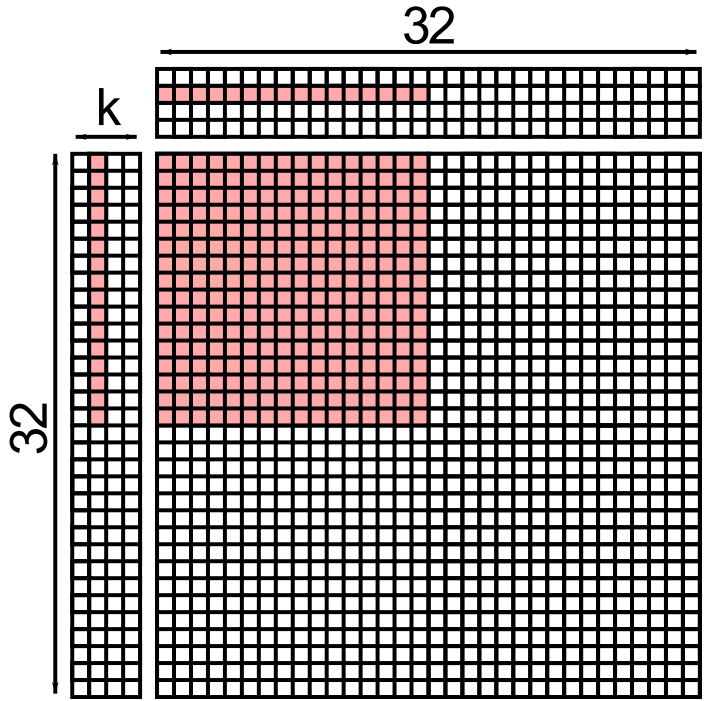
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 512 FLOPS per instruction (FMOPA)
- ZA tile holds 32x32 values
- 4 vector registers for streaming A and B



Just-in-time Code Generation

LIBXSMM using Scalable Matrix Extension

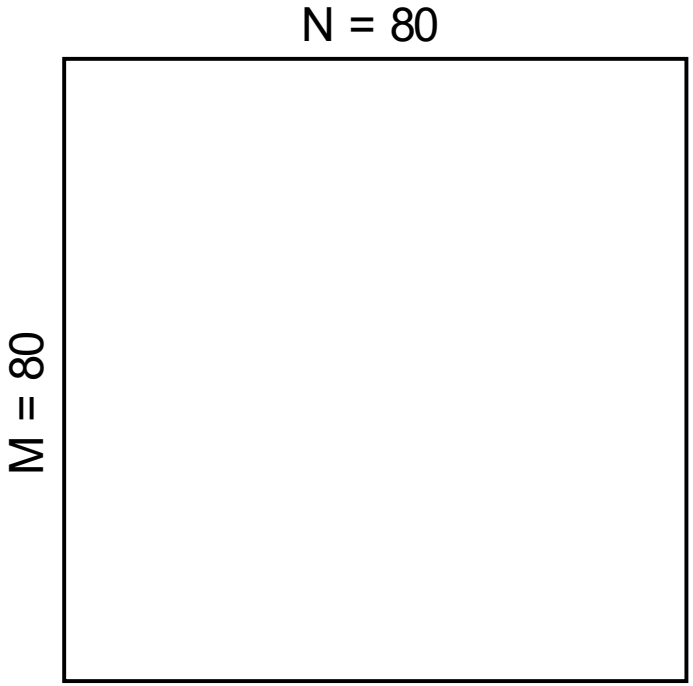
- Compute: $C \mathrel{+}= AB$
- Single Precision (FP32)
- 512 FLOPS per instruction (FMOPA)
- ZA tile holds 32x32 values
- 4 vector registers for streaming A and B



Just-in-time Code Generation

Matrix Blocking

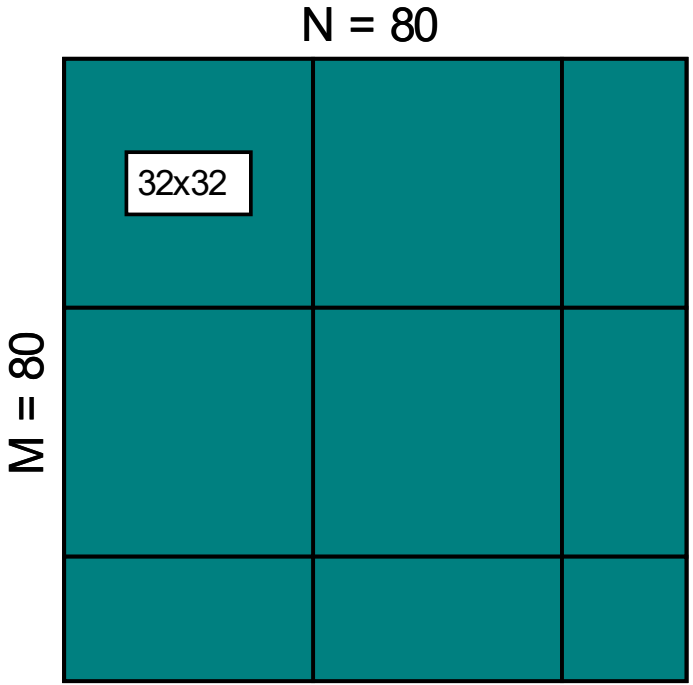
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 0



Just-in-time Code Generation

Matrix Blocking

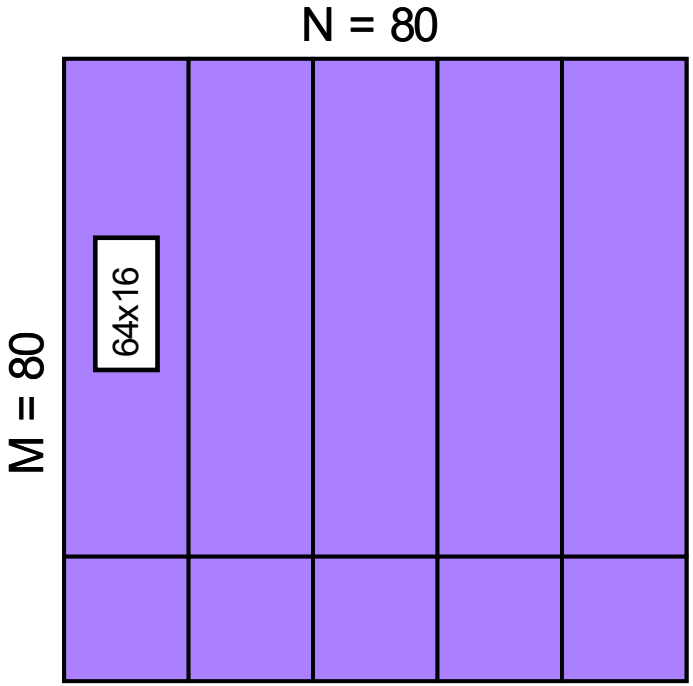
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 9



Just-in-time Code Generation

Matrix Blocking

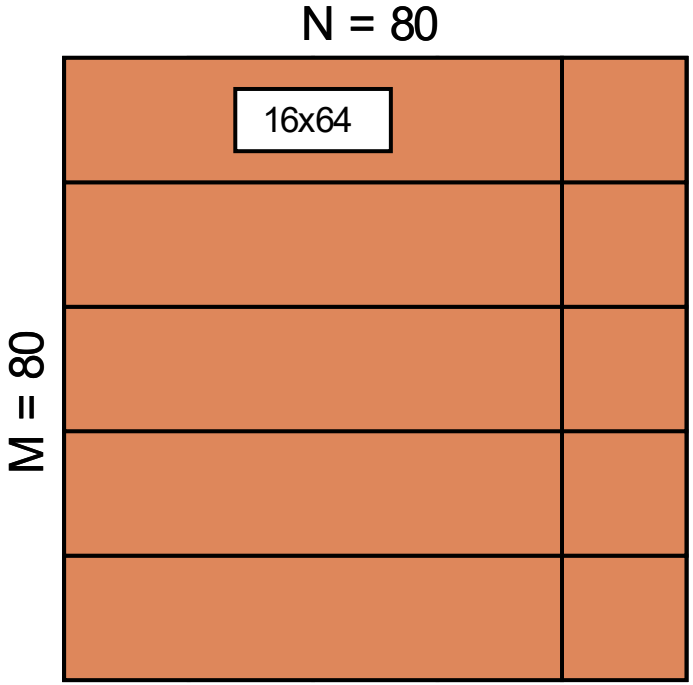
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 10



Just-in-time Code Generation

Matrix Blocking

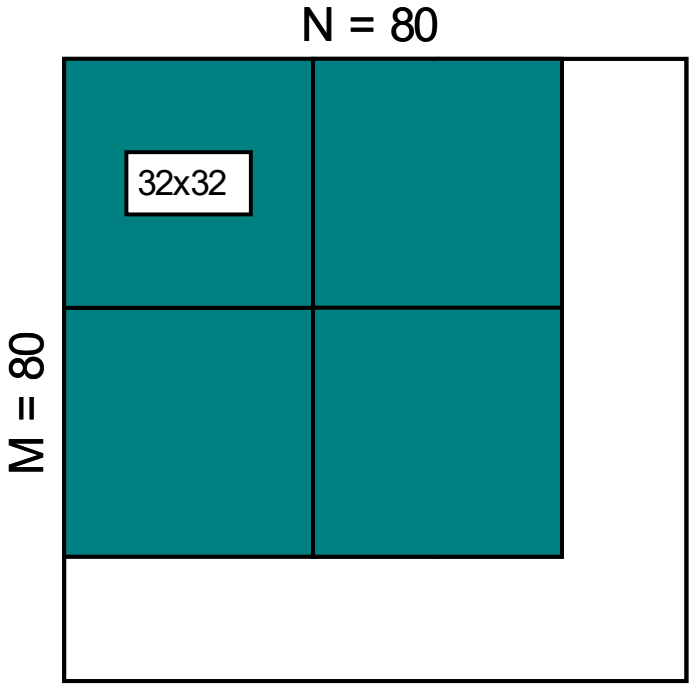
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 10



Just-in-time Code Generation

Matrix Blocking

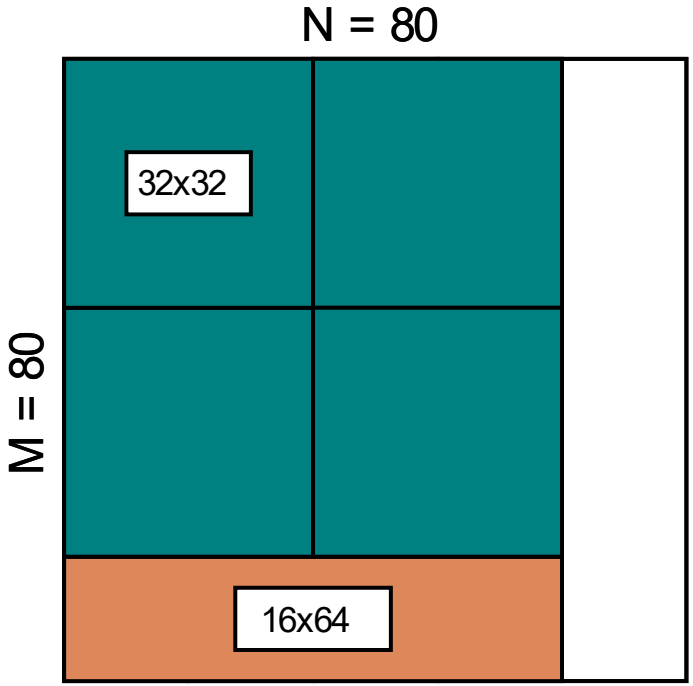
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 4



Just-in-time Code Generation

Matrix Blocking

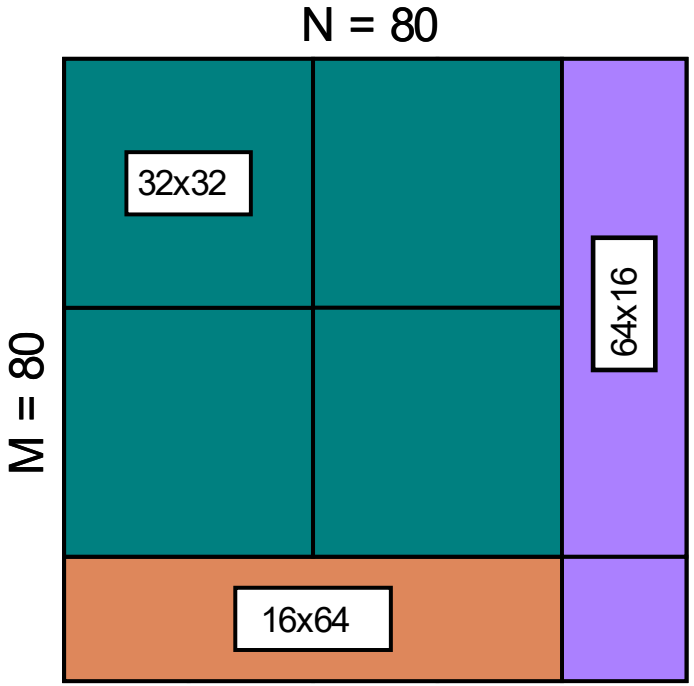
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 5



Just-in-time Code Generation

Matrix Blocking

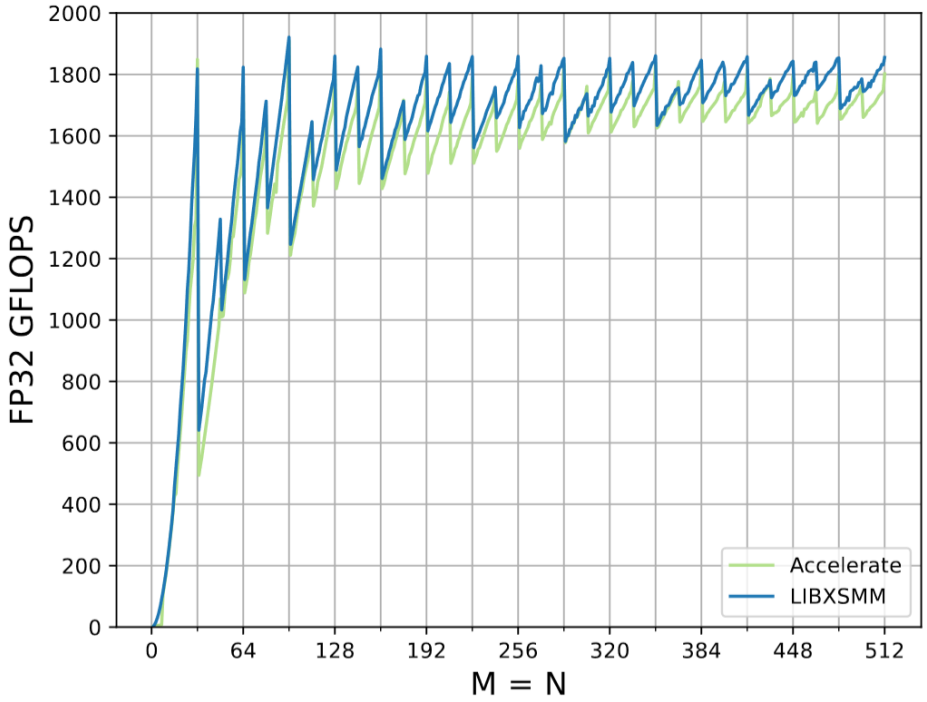
- Example: $C \in \mathbb{R}^{80x80}$
- Presicion: FP32
- 3 kernel types:
  - 32x32
  - 64x16
  - 16x64
- Kernel count: 7



Performance evaluation

Computing: $C \mathrel{+}= AB^T$

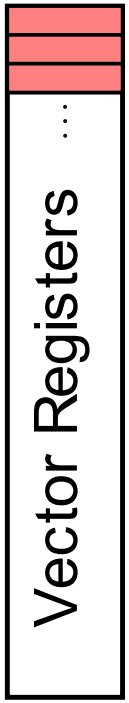
- $K = 512$
- Average speedup: 78 GFLOPS

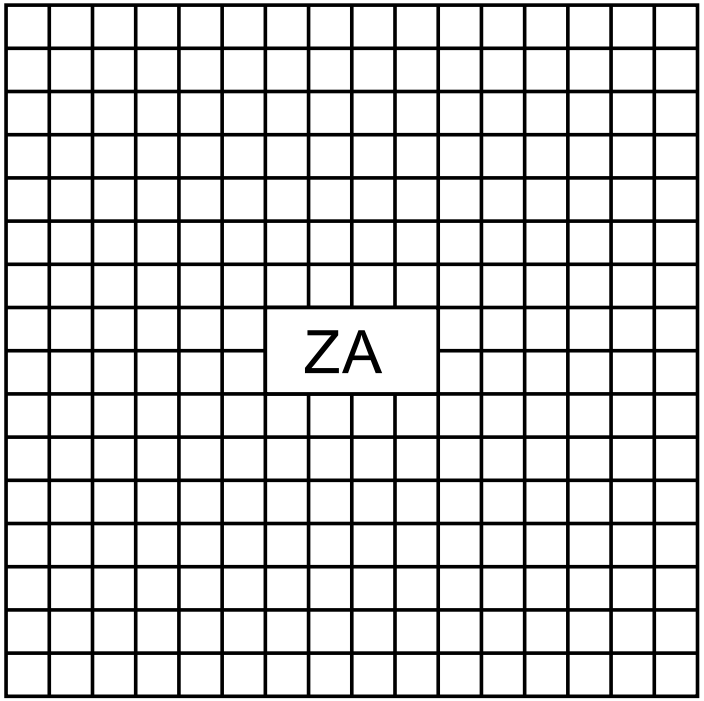


Just-in-time Code Generation

Transposing with ZA tile

- Allocate memory on stack
- Transpose B
- Store transposed B into stack
- Run normal k-loop

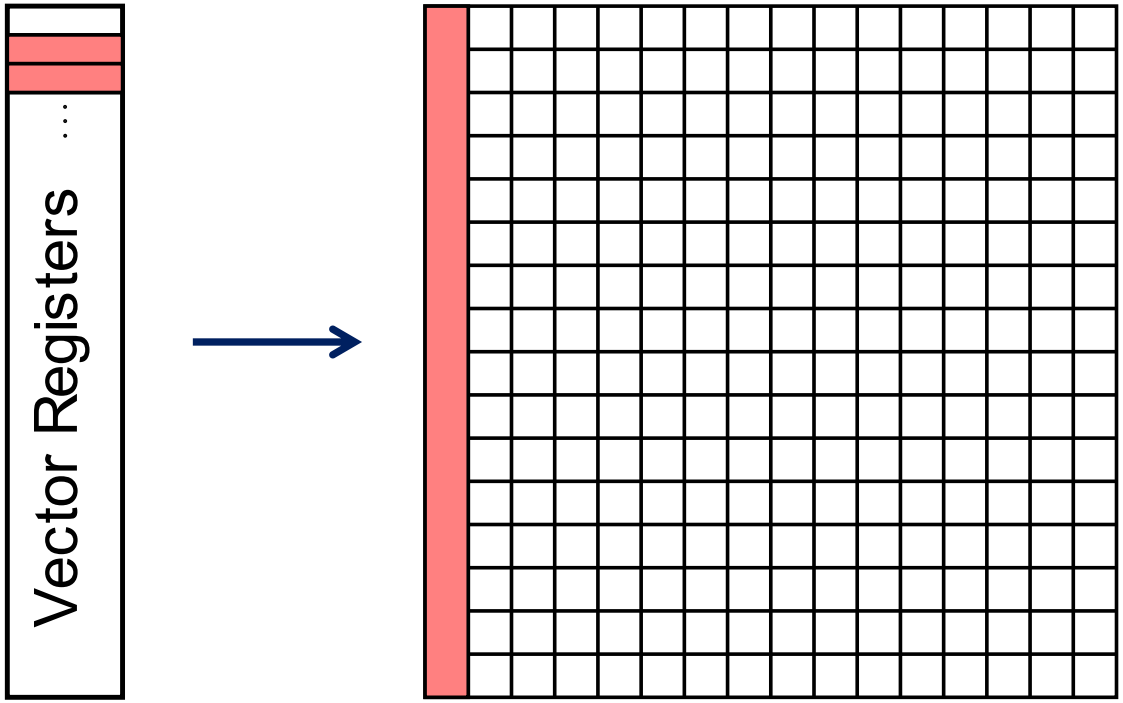




Just-in-time Code Generation

Transposing with ZA tile

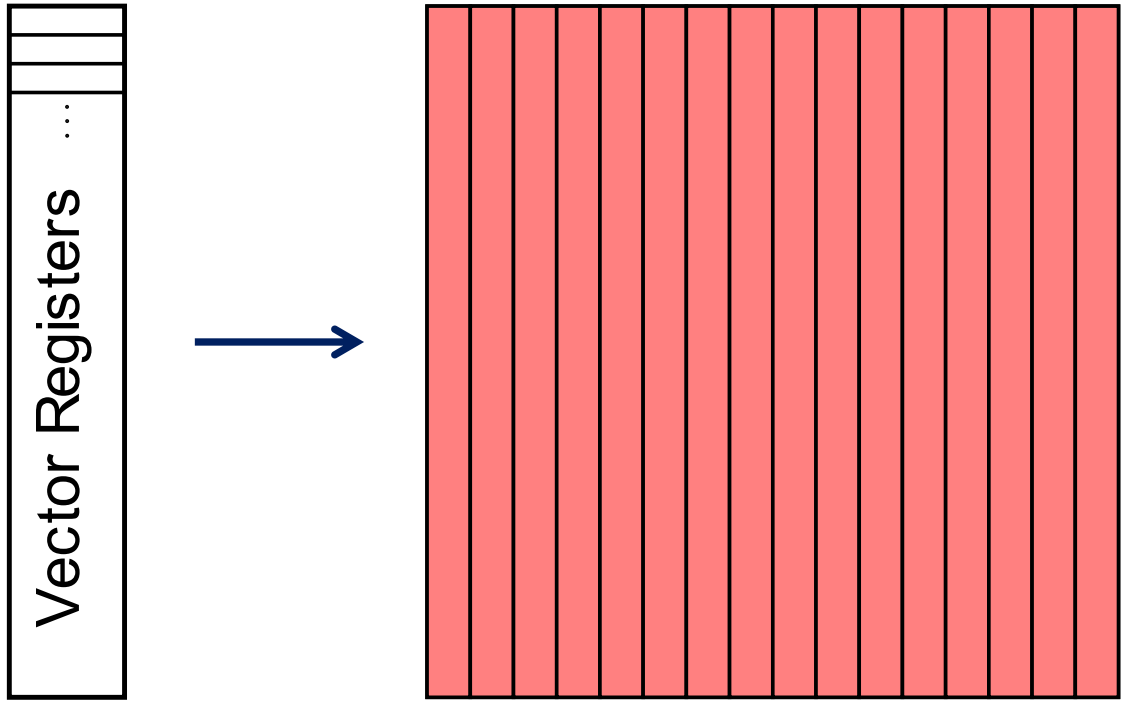
- Allocate memory on stack
- Transpose B
- Store transposed B into stack
- Run normal k-loop



Just-in-time Code Generation

Transposing with ZA tile

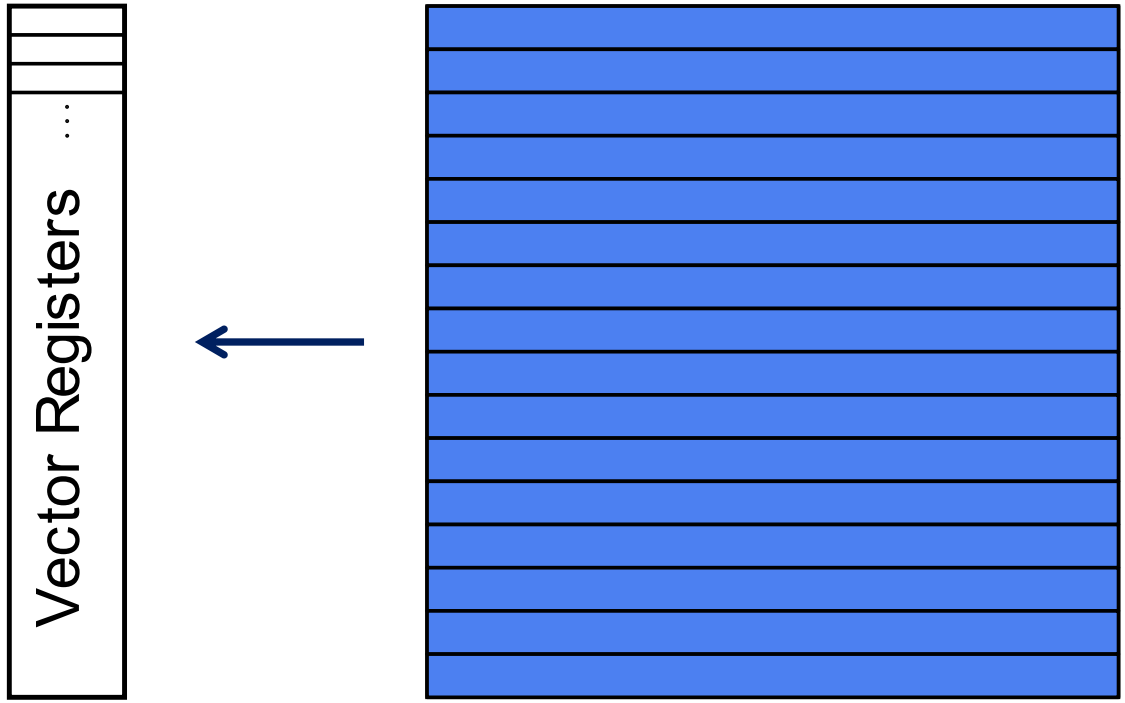
- Allocate memory on stack
- Transpose B
- Store transposed B into stack
- Run normal k-loop



Just-in-time Code Generation

Transposing with ZA tile

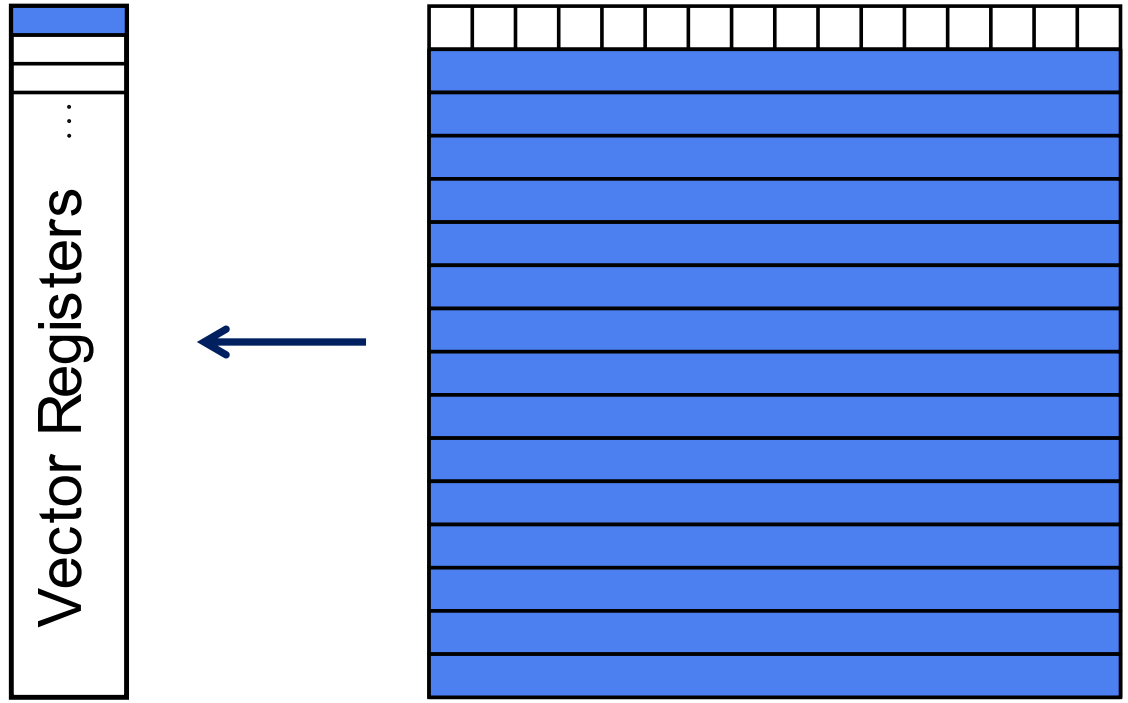
- Allocate memory on stack
- Transpose B
- Store transposed B into stack
- Run normal k-loop



Just-in-time Code Generation

Transposing with ZA tile

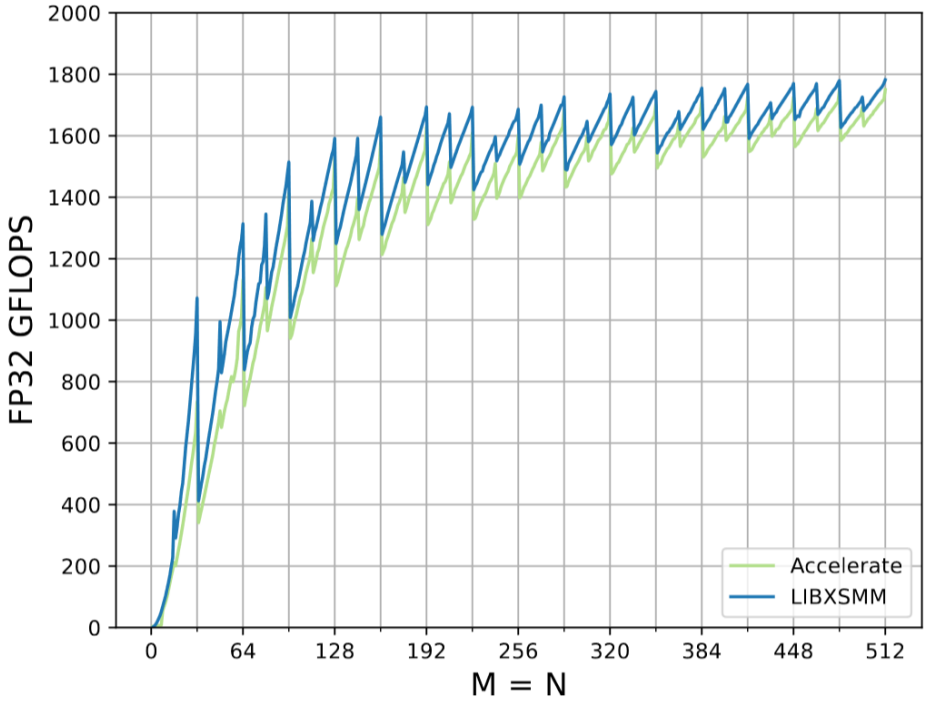
- Allocate memory on stack
- Transpose B
- Store transposed B into stack
- Run normal k-loop



Performance evaluation

Computing: $C \mathrel{+}= AB$

- $K = 512$
- Average speedup: 97 GFLOPS



Summary

- M4 first architecture supporting Scalable Matrix Extension (SME)
- M4's SME acceleration is FP32-centric
- SME has a major advantage over vector execution
- Open source LIBXSMM is significantly faster than vendor BLAS