# Portable Code Generation and Semi-Automatic Scheduling for BLIS

Julian Bellavita<sup>1</sup>, Grace Dinh<sup>1</sup>

<sup>1</sup>University of California, Berkeley

## F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption

Accelerated matrix

Next-generation 16-core architecture

#### Powerful machine learning accelerators

A14 \*

AI Chin Landsson

## Cambrian explosion in hardware architectures





#### NVIDIA Hopper GPU Architecture Accelerates Dynamic Programming Up to 40x Using New DPX Instructions

Dynamic programming algorithms are used in healthcare, robotics, quantum computing, data science and more.

#### YouTube is now building its own video-transcoding chips

Google throws custom silicon at YouTube's massive video-transcoding workload.

by Ron Amadeo - Apr 22, 2021 10:24am PST



SambaNova

CARDINAL

SNID

20N3-PRO1 18K977 42

1888 AHW34W0100065

# What runs on these chips?

## Introduction

- Strong demand for performant matrix/tensor kernels (including BLIS) on novel architectures
- Both cost of implementation per architecture *and* number of new architectures increasing rapidly

Image source: https://semiengineering.com/how-much-will-that-chip-cost/



Cost breakdown of developing new chips.

## What makes performant kernels hard to implement?

• Tedious, difficult to debug, and requires extensive experimentation and tuning

• Hardware-specific optimizations (and knowledge of HW architecture) often required

• Few performance programming experts

Ease of implementation and portability between architectures critical!











**BLAS routine** 

SYRK

**BLAS** routine



With BLIS... implement as scaffolding + architecturespecific microkernel



With BLIS... implement as scaffolding + architecturespecific microkernel



BLIS Microkernel

Hardware-specific implementation.

#### BLIS Microkernel

# Hardware-specific implementation.

P develop - OpenBLAS / kernel /
🖿 alpha
🖿 arm
arm64
e2k
generic generic
ia64
loongarch64
inips mips
mips64
power
riscv64
imd simd
sparc sparc
<b>x86</b>
<b>x86_64</b>
arch zarch



## Hardware-specific implementation - still timeconsuming to optimize!



Image source: The Deep Learning Compiler: A Comprehensive Survey (Li et. al '20), https://arxiv.org/abs/2002.03794 15

```
for (; i < m8; i += 8) {</pre>
for (i = 0; i < m32; i += 32) {</pre>
                                                                                                                                                                                        for (; i < m16; i += 16) {</pre>
                                                                                                                          for (j = 0; j < n6; j += 6) {</pre>
                                                                                                                                                                                            for (j = 0; j < n6; j += 6) {</pre>
    for (i = 0; i < n4; i += 4) {
                                                                                                                              DECLARE_RESULT_512(0, 0);
                                                                                                                                                                                                DECLARE RESULT 512(0, 0); DECLARE RESULT
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT_512(1, 0); DECLARE_RESULT_512(2, 0); DECLARE_RESULT_512(3, 0);
                                                                                                                              DECLARE RESULT 512(0, 1);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 1); DECLARE_RESULT
        DECLARE_RESULT_512(0, 1); DECLARE_RESULT_512(1, 1); DECLARE_RESULT_512(2, 1); DECLARE_RESULT_512(3, 1);
                                                                                                                              DECLARE_RESULT_512(0, 2);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 2); DECLARE_RESULT
        DECLARE_RESULT_512(0, 2); DECLARE_RESULT_512(1, 2); DECLARE_RESULT_512(2, 2); DECLARE_RESULT_512(3, 2);
                                                                                                                              DECLARE_RESULT_512(0, 3);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 3); DECLARE_RESULT
        DECLARE_RESULT_512(0, 3); DECLARE_RESULT_512(1, 3); DECLARE_RESULT_512(2, 3); DECLARE_RESULT_512(3, 3);
                                                                                                                              DECLARE_RESULT_512(0, 4);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 4); DECLARE_RESULT
                                                                                                                              DECLARE_RESULT_512(0, 5);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 5); DECLARE_RESULT
        for (k = 0; k < K; k++) {
                                                                                                                              for (k = 0; k < K; k++) {
                                                                                                                                                                                               for (k = 0; k < K; k++) {</pre>
            LOAD A 512(0, x); LOAD A 512(1, x); LOAD A 512(2, x); LOAD A 512(3, x);
                                                                                                                                  LOAD A 512(0, x);
                                                                                                                                                                                                   LOAD_A_512(0, x); LOAD_A_512(1, x);
                                                                                                                                  BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
                                                                                                                                                                                                   BROADCAST LOAD B 512(x, 0); BROADCAS
                                                                                                                                  BROADCAST_LOAD_B_512(x, 2); BROADCAST_LOAD_B_512(x, 3);
                                                                                                                                                                                                    BROADCAST_LOAD_B_512(x, 2); BROADCAS
            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
                                                                                                                                  BROADCAST_LOAD_B_512(x, 4); BROADCAST_LOAD_B_512(x, 5);
                                                                                                                                                                                                    BROADCAST_LOAD_B_512(x, 4); BROADCAS
            BROADCAST_LOAD_B_512(x, 2); BROADCAST_LOAD_B_512(x, 3);
                                                                                                                                  MATMUL_512(0, 0);
                                                                                                                                                                                                   MATMUL_512(0, 0); MATMUL_512(1, 0);
            MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
                                                                                                                                  MATMUL_512(0, 1);
                                                                                                                                                                                                   MATMUL_512(0, 1); MATMUL_512(1, 1);
            MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(2, 1); MATMUL_512(3, 1);
                                                                                                                                  MATMUL_512(0, 2);
                                                                                                                                                                                                   MATMUL_512(0, 2); MATMUL_512(1, 2);
            MATMUL_512(0, 2); MATMUL_512(1, 2); MATMUL_512(2, 2); MATMUL_512(3, 2);
                                                                                                                                  MATMUL_512(0, 3);
                                                                                                                                                                                                   MATMUL_512(0, 3); MATMUL_512(1, 3);
            MATMUL_512(0, 3); MATMUL_512(1, 3); MATMUL_512(2, 3); MATMUL_512(3, 3);
                                                                                                                                 MATMUL_512(0, 4);
                                                                                                                                                                                                   MATMUL_512(0, 4); MATMUL_512(1, 4);
                                                                                                                                 MATMUL_512(0, 5);
                                                                                                                                                                                                    MATMUL_512(0, 5); MATMUL_512(1, 5);
                                                                                                                              }
        STORE 512(0, 0); STORE 512(1, 0); STORE 512(2, 0); STORE 512(3, 0);
                                                                                                                                                                                               }
                                                                                                                              STORE 512(0, 0);
                                                                                                                                                                                               STORE_512(0, 0); STORE_512(1, 0);
        STORE_512(0, 1); STORE_512(1, 1); STORE_512(2, 1); STORE_512(3, 1);
                                                                                                                              STORE 512(0, 1);
                                                                                                                                                                                               STORE_512(0, 1); STORE_512(1, 1);
        STORE_512(0, 2); STORE 512(1, 2); STORE_512(2, 2); STORE 512(3, 2);
                                                                                                                              STORE_512(0, 2);
                                                                                                                                                                                               STORE_512(0, 2); STORE_512(1, 2);
        STORE_512(0, 3); STORE_512(1, 3); STORE_512(2, 3); STORE_512(3, 3);
                                                                                                                              STORE_512(0, 3);
                                                                                                                                                                                               STORE_512(0, 3); STORE_512(1, 3);
                                                                                                                              STORE_512(0, 4);
                                                                                                                                                                                               STORE_512(0, 4); STORE_512(1, 4);
    for (; j < n2; j += 2) {</pre>
                                                                                                                              STORE 512(0, 5);
                                                                                                                                                                                               STORE_512(0, 5); STORE_512(1, 5);
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT_512(1, 0); DECLARE_RESULT_512(2, 0); DECLARE_RESULT_512(3, 0);
                                                                                                                                                                                           }
        DECLARE_RESULT_512(0, 1); DECLARE_RESULT_512(1, 1); DECLARE_RESULT_512(2, 1); DECLARE_RESULT_512(3, 1);
                                                                                                                          for (; j < n2; j += 2) {</pre>
                                                                                                                                                                                           for (; j < n2; j += 2) {</pre>
        for (k = 0; k < K; k++) {
                                                                                                                              DECLARE_RESULT_512(0, 0);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 0); DECLARE_RESULT
            LOAD_A_512(0, x); LOAD_A_512(1, x); LOAD_A_512(2, x); LOAD_A_512(3, x);
                                                                                                                              DECLARE_RESULT_512(0, 1);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 1); DECLARE_RESULT
                                                                                                                              for (k = 0; k < K; k++) {
            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
                                                                                                                                                                                               for (k = 0; k < K; k++) {</pre>
                                                                                                                                  LOAD A 512(0, x);
                                                                                                                                                                                                   LOAD_A_512(0, x); LOAD_A_512(1, x);
            MATMUL 512(0, 0); MATMUL 512(1, 0); MATMUL 512(2, 0); MATMUL 512(3, 0);
                                                                                                                                  BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
                                                                                                                                                                                                   BROADCAST_LOAD_B_512(x, 0); BROADCAS
            MATMUL 512(0, 1); MATMUL 512(1, 1); MATMUL 512(2, 1); MATMUL 512(3, 1);
                                                                                                                                  MATMUL_512(0, 0);
                                                                                                                                                                                                   MATMUL_512(0, 0); MATMUL_512(1, 0);
        }
                                                                                                                                  MATMUL_512(0, 1);
                                                                                                                                                                                                   MATMUL_512(0, 1); MATMUL_512(1, 1);
        STORE_512(0, 0); STORE_512(1, 0); STORE_512(2, 0); STORE_512(3, 0);
                                                                                                                              }
                                                                                                                                                                                               }
        STORE_512(0, 1); STORE_512(1, 1); STORE_512(2, 1); STORE_512(3, 1);
                                                                                                                              STORE_512(0, 0);
                                                                                                                                                                                               STORE_512(0, 0); STORE_512(1, 0);
                                                                                                                              STORE_512(0, 1);
                                                                                                                                                                                               STORE_512(0, 1); STORE_512(1, 1);
    for (; j < N; j++) {</pre>
                                                                                                                                                                                           3
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT_512(1, 0); DECLARE_RESULT_512(2, 0); DECLARE_RESULT_512(3, 0); for (; j < N; j++) {
                                                                                                                                                                                           for (; j < N; j++) {</pre>
                                                                                                                              DECLARE RESULT 512(0, 0);
                                                                                                                                                                                               DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        for (k = 0; k < K; k++) {
                                                                                                                              for (k = 0; k < K; k++) {
                                                                                                                                                                                               for (k = 0; k < K; k++) {</pre>
            LOAD A 512(0, x): LOAD A 512(1, x): LOAD A 512(2, x): LOAD A 512(3, x):
                                                                                                                                                                                                   LOAD_A_512(0, x); LOAD_A_512(1, x);
                                                                                                                                  LOAD_A_512(0, x);
            BROADCAST_LOAD_B_512(x, 0);
                                                                                                                                                                                                   BROADCAST LOAD B 512(x, 0);
                                                                                                                                  BROADCAST_LOAD_B_512(x, 0);
            MATMUL 512(0, 0); MATMUL 512(1, 0); MATMUL 512(2, 0); MATMUL 512(3, 0);
                                                                                                                                  MATMUL_512(0, 0);
                                                                                                                                                                                                   MATMUL_512(0, 0); MATMUL_512(1, 0);
        }
        STORE_512(0, 0); STORE_512(1, 0); STORE_512(2, 0); STORE_512(3, 0);
                                                                                                                              STORE_512(0, 0);
                                                                                                                                                                                               STORE_512(0, 0); STORE_512(1, 0);
```

}

for (i :	= 0; i < m32; i += 32) {
for	$(j = 0; j < n4; j += 4) $ {
	DECLARE RESULT 512(0, 0); DECLARE RESULT 512(1, 0); DECLARE RESULT 512(2, 0); DEC
	DECLARE RESULT 512(0, 1); DECLARE RESULT 512(1, 1); DECLARE RESULT 512(2, 1); DEC
	DECLARE RESULT 512(0, 2): DECLARE RESULT 512(1, 2): DECLARE RESULT 512(2, 2): DEC
	DECLARE RESULT 512(0, 3); DECLARE RESULT 512(1, 3); DECLARE RESULT 512(2, 3); DEC
	for $(k = 0; k < K; k++)$
	104D = 5, x + 1, x + 1, y + 104D = 512(1 - x) + 104D = 512(2 - x) + 104D = 512(3 - x
	COND_A_012(0, x); COND_A_012(1, x); COND_A_012(2; x); COND_A_012(0, x);
	REGARCAST LOAD B 512(x A) BROADCAST LOAD B 512(x 1)
	BROADCAST LOAD B 512(x, 2); BROADCAST LOAD B 512(x, 2);
	BROADCAST_LOAD_D_SIZ(X, Z), BROADCAST_LOAD_D_SIZ(X, S),
	MATMUH 512(0, 0): MATMUH 512(1, 0): MATMUH 512(2, 0): MATMUH 512(3, 0):
	MATMUL 512(0, 1); MATMUL 512(1, 1); MATMUL 512(2, 1); MATMUL 512(3, 1);
	MATNUE 512(0, 2); MATNUE 512(1, 2); MATNUE 512(2, 1); MATNUE 512(3, 1);
	MATHUE_512(0, 2), MATHUE_512(1, 2), MATHUE_512(2, 2), MATHUE_512(3, 2),
	MAIMUL_512(0, 3); MAIMUL_512(1, 3); MAIMUL_512(2, 3); MAIMUL_512(3, 3);
	J STORE 512(0, 0), STORE 512(1, 0), STORE 512(2, 0), STORE 512(2, 0),
	STORE_512(0, 0); STORE_512(1, 0); STORE_512(2, 0); STORE_512(3, 0);
	STORE_512(0, 1); STORE_512(1, 1); STORE_512(2, 1); STORE_512(3, 1);
	SIORE_512(0, 2); SIORE_512(1, 2); SIORE_512(2, 2); SIORE_512(3, 2);
	STORE_512(0, 3); STORE_512(1, 3); STORE_512(2, 3); STORE_512(3, 3);
}	
for	(; ] < n2; ] += 2) {
	DECLARE_RESULT_512(0, 0); DECLARE_RESULT_512(1, 0); DECLARE_RESULT_512(2, 0); DEC
	DECLARE_RESULT_512(0, 1); DECLARE_RESULT_512(1, 1); DECLARE_RESULT_512(2, 1); DEC
	for $(k = 0; k < K; k++)$ {
	LOAD_A_512(0, x); LOAD_A_512(1, x); LOAD_A_512(2, x); LOAD_A_512(3, x);
	<pre>BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);</pre>
	MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
	<pre>MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(2, 1); MATMUL_512(3, 1);</pre>
	}
	STORE_512(0, 0); STORE_512(1, 0); STORE_512(2, 0); STORE_512(3, 0);
	STORE_512(0, 1); STORE_512(1, 1); STORE_512(2, 1); STORE_512(3, 1);
}	
for	(; j < N; j++) {
	DECLARE_RESULT_512(0, 0); DECLARE_RESULT_512(1, 0); DECLARE_RESULT_512(2, 0); DEC
	for $(k = 0; k < K; k++)$ {
	LOAD_A_512(0, x); LOAD_A_512(1, x); LOAD_A_512(2, x); LOAD_A_512(3, x);
	<pre>BROADCAST_LOAD_B_512(x, 0);</pre>
	MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
	}
	STORE_512(0, 0); STORE_512(1, 0); STORE_512(2, 0); STORE_512(3, 0);
}	
3	

}

🗋 dgen

🗋 dger

🗋 dgen

🗋 dger

🗋 dger

🗋 dger 🗋 dger

nm_beta_skylakex.c	. += 8) {		for (	; i < m16; i += 16) {
nm_kernel_16x2_haswell.S	RESULT_512(0, 0);		1	<pre>for (j = 0; j &lt; n6; j += 6) {     DECLARE_RESULT_512(0, 0); DECLARE_RESULT</pre>
nm_kernel_16x2_skylakex.S	RESULT_512(0, 1); RESULT_512(0, 2);			DECLARE_RESULT_512(0, 1); DECLARE_RESULT_512(0, 2); DECLARE_RESULT_512(0, 2); DECLARE_RESULT_512(0, 2);
nm_kernel_16x2_skylakex.c	RESULT_512(0, 3);			DECLARE_RESULT_512(0, 3); DECLARE_RESULT DECLARE_RESULT_512(0, 3); DECLARE_RESULT
nm_kernel_4x4_haswell.S	RESULT_512(0, 5);			DECLARE_RESULT_512(0, 4); DECLARE_RESULT DECLARE_RESULT_512(0, 5); DECLARE_RESULT
nm_kernel_4x8_haswell.S	= 0; k < K; k++) { 0_A_512(0, x);			<pre>for (k = 0; k &lt; K; k++) {     LOAD_A_512(0, x); LOAD_A_512(1, x);</pre>
nm_kernel_4x8_sandy.S	DCAST_LOAD_B_512(x, @ DCAST_LOAD_B_512(x, 2	<pre>3); BROADCAST_LOAD_B_512(x, 1 2); BROADCAST_LOAD_B_512(x, 3</pre>	.); i);	<pre>BROADCAST_LOAD_B_512(x, 0); BROADCAS BROADCAST_LOAD_B_512(x, 2); BROADCAS</pre>
nm_kernel_4x8_skylakex.c	DCAST_LOAD_B_512(x, 4	<pre>\$     BROADCAST_LOAD_B_512(x, 5     BROADCAST_LOAD_B_512(x, 5     Section 2) </pre>	;);	BROADCAST_LOAD_B_512(x, 4); BROADCAS
nm_kernel_4x8_skylakex_2.c	UL_512(0, 0);			MATMUL_512(0, 0); MATMUL_512(1, 0);
nm_kernel_6x4_piledriver.S	UL_512(0, 1); UL_512(0, 2);			MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(0, 2); MATMUL_512(1, 2);
nm_kernel_8x2_bulldozer.S	UL_512(0, 3); UL_512(0, 4);			MATMUL_512(0, 3); MATMUL_512(1, 3); MATMUL_512(0, 4); MATMUL_512(1, 4);
nm_kernel_8x2_piledriver.S	UL_512(0, 5);			MATMUL_512(0, 5); MATMUL_512(1, 5);
nm_kernel_8x8_skylakex.c	2(0, 0);			<pre>STORE_512(0, 0); STORE_512(1, 0);</pre>
nm_ncopy_2.S	.2(0, 1); .2(0, 2);			STORE_512(0, 1); STORE_512(1, 1); STORE_512(0, 2); STORE_512(1, 2);
1m_ncopy_4.S	.2(0, 3);			STORE_512(0, 3); STORE_512(1, 3); STORE 512(0, 4); STORE 512(1, 4);
nm_ncopy_8.S	.2(0, 5);			STORE_512(0, 5); STORE_512(1, 5);
n ncopy 8 bulldozer.S	2: i += 2) {		)	for (· i < p2· i += 2) {
	RESULT_512(0, 0);			DECLARE_RESULT_512(0, 0); DECLARE_RESULT
nm_ncopy_8_skylakex.c	RESULT_512(0, 1);			DECLARE_RESULT_512(0, 1); DECLARE_RESULT
nm_small_kernel_nn_skylakex.c	: 0; k < K; k++) { )_A_512(0, x);			<pre>for (k = 0; k &lt; K; k++) {     LOAD_A_512(0, x); LOAD_A_512(1, x);</pre>
nm_small_kernel_nt_skylakex.c	DCAST_LOAD_B_512(x, 0	<pre>3); BROADCAST_LOAD_B_512(x, 1</pre>	.);	BROADCAST_LOAD_B_512(x, 0); BROADCAS
nm_small_kernel_permit_skylakex.c	UL_512(0, 1);			MATMUL_512(0, 1); MATMUL_512(1, 1);
nm_small_kernel_tn_skylakex.c	.2(0, 0);			; STORE_512(0, 0); STORE_512(1, 0);
nm_small_kernel_tt_skylakex.c	.2(0, 1);		)	STORE_512(0, 1); STORE_512(1, 1);
nm_tcopy_16_skylakex.c	l; j++) {		1	for (; j < N; j++) {
nm_tcopy_2.S	RESULT_512(0, 0); 0; k < K; k++) {			<pre>DECLARE_RESULT_512(0, 0); DECLARE_RESULT for (k = 0; k &lt; K; k++) {</pre>
nm_tcopy_4.S	_A_512(0, x); DCAST LOAD B 512(x, 0	a) •		LOAD_A_512(0, x); LOAD_A_512(1, x); BROADCAST_LOAD_B_512(x, 0):
nm_tcopy_8.S	UL_512(0, 0);			MATMUL_512(0, 0); MATMUL_512(1, 0);
nm_tcopy_8_bulldozer.S	.2(0, 0);			, STORE_512(0, 0); STORE_512(1, 0);
nm_tcopy_8_skylakex.c			}	•

How do we make writing microkernels easier?

#### Code generation

- Instead of writing hand-optimized code, have the compiler do it for you.
  - Compiler optimizations (-03, -0fast in GCC/clang, etc.)
- How do you control them or optimize them for new hardware?
  - Write your own compiler (or fork an existing one)... if you have the compiler experts to build and maintain it.



## User Scheduling



Output Code (machine code, CUDA, etc.)

#### User-schedulable compilers

- Halide (Ragan-Kelley et al. PLDI '13), TVM (Chen et al. OSDI '18), Rise/Elevate (Hagedorn et al. ICFP '20), etc.
- Not suited for BLIS/new HW application:
  - difficulty of writing HW backends
  - aimed at different domains (image processing, ML graphs)
  - not designed for library kernels (e.g. TVM prefers numpy ndarray input) or interoperability with existing flows

Exo: a user-schedulable compiler for the accelerator era

<pre>def new_sgemm():</pre>
(proc
def sgemm_full(
N: size,
M: size,
K: size,
C: f32[N, M] @ DRAM,
A: <u>f32[N, K] @ DRAM</u> ,
B: f32[K, M] @ DRAM,
):
for i in par(0, N):
for j in par(0, M):
for k in par(0, K):
C[i, j] += A[i, k] * B[k, j
return sgemm_full



#### **HW-specific schedule**

matmul\_c\_i8 = matmul\_c\_i8.split('i #0',16,['i','i\_in'], perfect=True)
matmul\_c\_i8 = matmul\_c\_i8.reorder('i\_in #0','j')
matmul\_c\_i8 = matmul\_c\_i8.split('j #0',16,['j','j\_in'], perfect=True)
matmul\_c\_i8 = matmul\_c\_i8.lift\_alloc('res : \_ #0', n\_lifts=1)
matmul\_c\_i8 = matmul\_c\_i8.lift\_alloc('res : \_ #0', n\_lifts=1, mode='col', size=16)
matmul\_c\_i8 = matmul\_c\_i8.lift\_alloc('res : \_ #0', n\_lifts=2)
matmul\_c\_i8 = matmul\_c\_i8.fission\_after('res[\_] = 0.0 #0', n\_lifts=2)
matmul\_c\_i8 = matmul\_c\_i8.fission\_after('for k in \_:\_ #0', n\_lifts=2)
matmul\_c\_i8 = matmul\_c\_i8.fission\_after('for k in \_:\_ #0', n\_lifts=2)
matmul\_c\_i8 = matmul\_c\_i8.reorder('i\_in #0','k')
matmul\_c\_i8 = matmul\_c\_i8.reorder('i\_in #0','k')

Humanwritten schedule Autotuner + crowdsourced perf data Modelbased optimizer



HW-sp	ecific sc	hedule			
<pre>matmul_c_i8 = matmul_c_i8.split('i#0',16,['i','i_in'], perfect=True) matmul_c_18 = matmul_c_18.reorder('i_in #0', j') retrul = i0 = retru</pre>					
<pre>matmut_c_is = matmut_c_is.spir(') #0 ,10,('); 'j_in'; perfect=/nue) matmut_c_is = matmut_c_is.lift_alloc('res : _ #0', n_lifts=1) matmut_c_is = matmut_c_is.lift_alloc('res : _ #0', n_lifts=1) matmut_c_is = matmut_c_is.lift_alloc('res : _ #0', n_lifts=1)</pre>					
<pre>matmul_c_i8 = matmul_c_i8.lift_alloc('res : _ #0', n_lifts=2) matmul_c_i8 = matmul_c_i8.lift_alloc('res[_] = 0.0 #0', n_lifts=2)</pre>					
<pre>matmul_c_i8 = matmul_c_i8.fission_after('for k in _:_ #0', n_lifts=2) matmul_c_i8 = matmul_c_i8.reorder('i_in #0','k')</pre>					
<pre>matmul_c_i8 = matmul_c_i8</pre>	.reorder('j_in #0','k')				
Human-	Autotuner +	Model-			
written	crowdsourced	based			
schedule	perf data	optimizer			





#### **HW backend**

```
@instr('{C_data} = _mm512_mask_fmadd_ps
def mm512_mask_fmadd_ps(
        N: size,
        A: f32[16] @ AVX512,
        B: f32[16] @ AVX512,
        C: [f32][16] @ AVX512,
):
    assert N >= 1
    assert N < 16
    assert stride(A, 0) == 1
    assert stride(B, 0) == 1
    assert stride(C, 0) == 1
    for i in par(0, 16):
        if i < N:
```

C[i] += A[i] \* B[i]





#### Simple a

#### **Optimized Code**

#### backend

gproc def sgem\_kernel\_avx512\_6x4(K: size, A: [f32][6, K] @ DRAM,

	at trazite, ou glower, c. trazite, ou glower,
	assert $K \ge 1$
	assert stride(8, 1) == 1
	assert stride(C, 1) == 1
	C_reg: R(6, 4, 16) @ AVA512 for i in par(0, 6):
	for jo in par(0, 4):
	em512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 = jo:16 = jo + 16])
	for k in parto, K):
	A_vec: R[16] @ AVX512
	em512_set1_ps(A_vec, A[i, k:k + 1])
	B vec: B[16] 8 AVX512
	mm512_loadu_ps(B_vec[0:16], B[k, 16 * jo:16 * jo + 16])
	mnSI2_fmadd_ps(A_vec, B_vec, C_reg[i, jo, 0:16])
	for 1 in par(0, 0):
	em512_storeu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 8:16])
f	bottom_panel_kernel_scheduled(#: size, K: size, A: [f32][N, K] @ DRAM,
	8: [132][K, 64] @ DRAM,
	C: [132][H, 64] @ DRAH):
	assert K >= 1
	assert stride(A, 1) == 1
	assert stride(C, 1) == 1
	if N == 1: comm kernel purch2 lodik A[0:1 0:4] B[0:4 0:64] (for out)
	else:
	sgeme_kernel_avx512_2x4(K, A[8:2, 0:K], B[0:K, 0:64], C[0:2, 0:64])
	sgemm_kernel_avx512_4x4(K, A[0:4, 0:K], B[0:K, 0:64],
	clo:4, 0:641)
	sgemm_kernel_avx512_5x4(K, A[0:5, 0:K], B[0:K, 0:64],
	for k in par(0, K):
	for i in par(0, N):
1	right_panel_kernel_scheduled(N: size, K: size, A: [f32][6, K] @ DRAM,
	8: [f32][K, N] @ DRAM, C: [f32][6, N] @ DRAM):
	assert K >= 1
	assert stride(A, 1) == 1
	assert stride(8, 1) == 1
	assert N / 16 < 4
	if N / 16 0:
	C_reg: R16, 1, 16] @ AVX512 C reg 1: R16, 16] @ AVX512
	for i in per(0, 6):
	mmSI2_maskz_loadu_ps(N, C_reg_1[i, 0:16], C[i, 0:N])
	for k in parto, K):
	A_reg2: R[16] @ AVX512
	mmS12_mask_set1_ps(N, A_reg2, A[i, k:k + 1])
	mmS12 maskz loadu ps(N, B reg2[0:16], B[k, 0:N])
	mn512_mask_fmadd_ps(N, A_reg2, B_reg2, C_reg_1[i, 0:16])
	for i in par(0, 6): mp[]] mack charge an(NC[i0,N]) C real(i0,16])
	else:
	C_reg: Rto, 2, 161 @ AVX512
	for i in par(0, 6):
	for jo in par(0, 1):
	<pre>me512_(oadv_pstC_reg[1, jo, 0:16], C[i,</pre>
	mm512_maskz_loadu_ps(N % 16, C_reg_1[i, 0:16], C[i, 16:N])
	for k in par(0, K):
	for 1 in parts, 67:

mm512\_set1\_ps(A\_reg, A[i, k:k + 1])

m512\_sot1\_ps(A\_reg, A[i, k:k + 1]) B\_reg: R[i6] # AVS12 ar532\_load\_ps(B\_reg[0:10], B[k, 16 + jo:16 + jo + 16]) ar532\_foud\_ps(A\_reg, B\_reg, C\_reg[i, jo, 0:16]) A\_reg2: R[i6] # AVS12 N/ (1 = 2); Cress B(1; 3) = 200212 Cress B(1; 3) = 200212 for 1: north, 3); for 1: no  $\begin{array}{l} \label{eq:response} \left\{ \begin{array}{l} e_{1} e_{2} i \left( ti \right) \in \left[ i \in [0, t] \right] \\ e_{1} e_{2} i \left( ti \right) e_{2} i \left( ti \right) e_{1} i \left( ti \right) e_{2} i \left( ti \right) e_{2}$ for k in par(4, 4): for j = 1 (a par(4, 7): A\_reg2: R[18] @ AVX512 ar512\_rask\_set2\_as(N % 16, A\_reg2, A[i, k:k + 1]) B\_reg2: R[16] @ AVX512 ar512\_raskz\_loadu\_ps(N % 16, B\_reg2[0:16], B[k, d(n)]  $\begin{array}{l} close \\ cress: R(s, N \ / \ i 6 \ + \ j, \ 10 \ 0 \ AV0512 \\ Cress: R(s, 10 \ 0 \ AV0512 \ ) \\ cress: R(s, 10 \ 0 \ AV0512 \ ) \\ cress: R(s, 10 \ 0 \ AV0512 \ ) \\ cress: R(s, 10 \ 0 \ AV0512 \ ) \\ ress: R(s, 10 \ A$ 

 
 A\_reg2: R(16) # AVX512
 6:16)

 ms512\_mosk\_set1\_pc(N + 16, A\_reg2, A(1, k:k + 1))
 B\_reg2: R(16) # AVX512

 ms512\_mosk\_set1\_pc(N + 16, A\_reg2, B\_reg2(b:16), B\_1(k, 16 + (N / 16):N)
 B(k, 16 + (N / 16):N)

 ms512\_mosk\_set0(s + 16, A\_reg2, B\_reg2, B\_ intervent (i, i); for i s is part(0, N / 10); mod2\_store\_part(0, i) (i = jo:15 \* jo + 16); C\_reg(1, jo =16); mod2\_store\_part(1 \* jo:16); mod2\_mod2\_store\_part(1 \* jo:16) (i = N / 16);N); c\_reg\_1(1, b:16); assert N == 1 assert X == 1 assert stride(A, 1) == 1 assert stride(A, 1) == 1 assert stride(C, 1) == 1 A1\_cache: f32(24, 35) @ DRAW\_STATIC for Ko in pair (S, 15) & (0) DRAW\_STATIC for Ko in pair (S, 15) & (0) DRAW\_STATIC for (B) in pair (S, 15) & (0) DRAW\_STATIC for (B) in pair (S, 15) & (0) DRAW\_STATIC for (B) in pair (S, 15) & (0) DRAW\_STATIC for 10 in par(0, 264): for 11 in par(0, 5 ii perte, 2007: ii in perte, 512): A1 cache[i0, i1] = A[264 + io + i0, 512 = ko + i1] for jo in par(0, N / 64): for i0 in par(0, 512): r i1 in par(0, 64):
B1\_cache[i0, i1] = B[512 + ko + i0, 64 + jo + i1] for ko in par(0, K / 512): B2\_cache: f32(512, 64) 0 DRAM\_STATIC D2\_croses (22(51), 64) = 0000\_(2571)C for 10 = in prof(5, 51) = 6 = 100 (2570) + if M % 264 > 0: for ko in par(0, K / 514) for ko in par(0, K / 514) for ko in par(0, K / 64): B3\_cache: f32[512, 64] @ DRAM\_STATIC for 10 in par(0, 512):

N % 65 > 8: for ko in por(0, K / 512): B4\_cache: f32[512, 66] 0 BBM\_STATIC for 10: in por(0, 512): for 11: in por(0, N = 66 + (N / 64)): B4\_cache(10; N = 10 = 8[512 + ko + 10, 66 + (N / 64) + 11] BT\_dttdetter, Arr BT\_dttdetter, Arr S12 + ko:512 + ko + 512], B4\_cache[8:512, 8:N - 64 + (N / 64]], (254 + (M / 264);H, B4\_cache[8:512, 8:N - 64 + (N / 64]], (254 + (M / 64);H)  $\begin{array}{l} K + 512 = 41; \\ for i \ i \ i \ m \ order \left( 5, \ M \ / \ 2501 \right) \\ for \ i \ o \ m \ order \left( 5, \ M \ / \ 2501 \right) \\ H \ (s, \ m \ order \ 125112), \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 125112, \\ for \ i \ s \ order \ 12512, \\ for \ s \ order \ 12512, \ s \ order \ s \$ genm\_above\_kernel(
 264, 64, K % 512, A[264 \* io:264 \* io + 264, 
 204, 04, K
 512, KI / 512; KI,

 512 + (K / 512); KI,

 85\_cache[0:K - 512 × (K / 512); 0:64],

 C[264 + io:264 + io + 264, 64 + jo:64 + jo + 64])
 A12 ≥ 47 for 10 in por(6), N / 26(1) R of 480 por(6), N / 26(1) R of 480 por(6), K > 512 × (K / 512)); for 41 in por(6), N = 64 × (M / 66)); B6, cache(10, 11) = 85(3 × (K / 512) + 10, 64 × (N / 64) + 11] 64 \* (N / 64) + i1] 5gess.above\_kernel( 264, N % 64, K % 512, A[264 \* io:264 \* io - 264, 512 \* (K / 512) \* (N, B6\_cache[8:K - 512 \* (K / 512), 0:N - 64 \* (N / 64)], C[264 \* io:264 \* io + 264, 64 \* (N / 64)N]) K + 512 - 8: for if N > 613 - 8: for concert f32[337, 64] g DBAM\_STATIC for concert f32[337, 64] g DBAM\_STATIC for ii in par(0, 431; for ii in par(0, 431; for ii in par(0, 431; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for ii in - 131; for ii in par(0, 131; for ii in - 131; for iin B7\_cache[0:K - 512 + (K / 512), 0:64], C[264 + (H / 264):H, 64 + 10:64 + 10 + 64]) 

for i0 in par(0, 512):
 for i1 in par(0, 64):
 B3\_cache[i0, i1] = B[512 + ko + i0, 64 + jo + i1]





#### **Optimized Code**





- Quickly implement, experiment with, and iterate on fast code targeting varied architectures.
- Generates C(++) with HW intrinsics: fully embeddable as libraries, compatible with C-based workflows
- Simple definition of HW backends separate from compiler (easily add new architectures, separate proprietary architectures from compilers)
- Allows reuse of code optimizations across multiple operations

#### Exo + undergrad + weeks = on 3 L3 BLIS routines.

## Here's how.

## 1. Extensive use of re-usable kernels

## The BLIS approach

• Cast BLAS ops in terms of reusable GEMM kernel (Goto and Van de Geijn)





- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



Modified GEPP procedure writes to varied-length panels of C



- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



Modified GEPP procedure writes to varied-length panels of C


- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



Modified GEPP procedure writes to varied-length panels of C



- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



Modified GEPP procedure writes to varied-length panels of C



- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



Modified GEPP procedure writes to varied-length panels of C



- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C





Modified GEPP procedure writes to varied-length panels of C Updating a single row looks like this



- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C





- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



Update left panel with GEBP

Middle square (diagonal) is an edge case

Do nothing for the right panel

### Vanilla SGEMM...

```
(dproc
def SGEMM(M: size, N: size, K: size, A: f32[M, K], B: f32[K, N], C: f32[M, N]):
    assert M >= 1
    assert N >= 1
    assert K >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
    for i in par(0, M):
        for j in par(0, N):
            for k in par(0, K):
                C[i, j] += A[i, k] * B[k, j]
```

### Vanilla SGEMM...

### Vanilla SGEMM...



...with scheduling instructions

### Vanilla SGEMM...

```
goroc
def SGEMM(M: size, N: size, K: size, A: f32[M, K], B: f32[K, N], C: f32[H, N]):
    assert N >= 1
    assert N >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
```

for i in par(0, M):
 for j in par(0, N):
 for k in par(0, K):
 C[i, j] += A[i, k] \* B[k, j]

### ...with **scheduling instructions**

```
def generate_GEBP(Kernel, H_blk, K_
                 GEBP = (Kernel
                         .partial eval(K=K c)
                          split('i', M_r, ['io', 'ii'], tail='cut_and_guar
                                                                        Split off edge cases
    Tile loops
                          fission_after('for jo in _:
                          fission_after('for io in .
                           reorder('ii','jo')
Reorder loops
                         .replace_all(microkernel)
                         .call_eqv(neon_microkernel, 'microkernel(_)')
                         .reorder('io', 'jo')
.stage_mem(f'B[0:{K_c},
                                                                  Substitute in optimized
                                f'{N_r}*io:{N_r}*io+{N_r}]'.
                                                                         microkernel
                  cetuco GERP
```

#### ....generates an optimized kernel....

```
// GEBPC
// N : size,
// A : [f32][64,64] @ORAM,
// B : [f32][64,N] @ORAM,
// C : [f32][64,N] @ORAM
// )
void GEBP( c_code_str_Context *ctxt, int_fast32_t N, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(N >= 1);
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
for (int jo = 0; jo < ((N) / (16)); jo++) {
float *8_strip = mallco(64 * 16 * sizeof(*8_strip));
for (int i = 0; i1 < 16; i1++) {
        B_strip[(10) * (16) + (11) * (1)] = 8.data[(10) * (B.strides[8]) + (11 + 16 * jo) * (B.strides[1])];
}</pre>
```

```
for (int io = 0; io < 16; io++) {
    neon_microkernel(ctxt,(struct exo_win_2f32){ (float*)&A.data[(4 * io) * (A.strides[0]) + (0) * (A.strides[1])], { A.strides[0], A.strides[1] } },(struct exo_win_2f32){ (float*)&B_strip[(0)</pre>
```

```
free(B_strip);
```

}

...

...generates an optimized kernel...

# ...that can be **called** or **inlined** (e.g. in SYRK)

```
void GEPP_syrk( c_code_str_Context *ctxt, int_fast32_t M, struct exo_win_2f32 A, struct exo_win_2f32 A_t, struct exo_win_2f32 C ) {
EX0_ASSUME(M >= 1);
EX0_ASSUME(A.strides[1] == 1);
EX0_ASSUME(A_t.strides[1] == 1);
EX0_ASSUME(C.strides[1] == 1);
 记r (int i = 0; i < M; i++) ┨
                           if (i == 6) {
                                                            1; jo++) {
                                   GEBP_scheduled(ctx], (struct exo_win_2f32){ (float*)&A.data[(6) * (A.strides[0]) + (0) * (A.strides[1])], { A.strides[0], A.strides[1] } }, (struct exo_win_2f32){ (float*)&A_t.
                               for (int j = 0; j < 3; j++) {</pre>
                                   for (int k = 0; k < 4; k++) {
                                      C.data[(6) * (C.strides[0]) + (j + 4) * (C.strides[1])] += A.data[(6) * (A.strides[0]) + (k) * (A.strides[1])] * A_t.data[(j + 4) * (A_t.strides[0]) + (k) * (A_t.strides[1])]
                               3
                           l else {
                               if (i == 7) {
                                                                 _____< 1; jo++) {
                                      GEBP_scheduled(ctx], (struct exo_win_2f32){ (float*)&A.data[(7) * (A.strides[8]) + (0) * (A.strides[1])], { A.strides[8], A.strides[1] } }, (struct exo_win_2f32){ (float*)&A.
                                   for (int j = 0; j < 4; j++) {</pre>
                                      for (int k = 0; k < 4; k++) {
                                          C.data[(7) * (C.strides[0]) + (j + 4) * (C.strides[1])] += A.data[(7) * (A.strides[0]) + (k) * (A.strides[1])] * A_t.data[(j + 4) * (A_t.strides[0]) + (k) * (A_t.strides[0])
                                   ે
                               } else
                                   if (i == 8) {
                                                                2; jo++) {
                                          GEBP_scheduled(ctx], (struct exo_win_2f32){ (float*)&A.data[(8) * (A.strides[0]) + (0) * (A.strides[1])], { A.strides[0], A.strides[1] } }, (struct exo_win_2f32){ (float*)&
                                       for (int j = 0; j < 1; j++) {
                                          for (int k = 0; k < 4; k++) {
                                              C.data[(8) * (C.strides[0]) + (j + 8) * (C.strides[1])] += A.data[(8) * (A.strides[0]) + (k) * (A.strides[1])] * A_t.data[(j + 8) * (A_t.strides[0]) + (k) * (A_t.strides[0])
                                   } else {
                                      if (i == 9) {
                                                                     2; jo++) {
                                              GEBP_scheduled(ctx, (struct exo_win_2f32){ (float*)&A.data[(9) * (A.strides[0]) + (0) * (A.strides[1])], { A.strides[0], A.strides[1] } }, (struct exo_win_2f32){ (float*
                                           for (int j = 0; j < 2; j++) {</pre>
                                              for (int k = 0; k < 4; k++) {
                                                   C.data[(9) * (C.strides[0]) + (j + 8) * (C.strides[1])] += A.data[(9) * (A.strides[0]) + (k) * (A.strides[1])] * A_t.data[(j + 8) * (A_t.strides[0]) + (k) * (A_t.strides[0
```

# 2. Easy Kernel Generation for Hardware

### Step 1: Hardware Backend

Mapping kernel to hardware done with **hardware intrinsics** specified by **equivalent Python code** 

HW architecture specifiable in **~1kLoC**, in **user-written file** (suitable for new/proprietary architectures)

```
# FMA
```

# Step 1: Hardware Backend



# Step 2: Code substitution

### Use code substitution

instructions (similar to how we embedded GEBP kernel earlier) in scheduling to generate HW intrinsics in code.

Transformation **formally guaranteed** to be **equivalent** to original code.

```
def generate_microkernel(kernel, N_reg, M_reg, K_blk):
        if N_reg%4:
                raise Exception(f"Error: N_req must be a multiple of 4, got {N_reg}")
        return (kernel
                .partial_eval(M_reg,N_reg)
                .partial_eval(K=K_blk)
                .reorder('j','k')
                .reorder('i','k')
                .split('j', 4, ['jo','ji'], perfect=True)
                .par_to_seg('for k in _: _')
                .stage_assn('C_reg', 'C[_] += _')
                .lift_alloc('C_reg : _', n_lifts=4)
                .double_fission('C_reg[_] = C[_]', 'C_reg[_] += _', n_lifts=4)
                .replace(neon_vld_4xf32, 'for ji in _: _ #0')
                .replace(neon_vst_4xf32, 'for ji in _: _ #1')
                .set_memory('C_reg', Neon4f)
                .stage_expr('A_vec', 'A[_,_]', memory=Neon4f)
                .stage_expr('B_vec', 'B[_,_]', memory=Neon4f)
                .replace_all(neon_vld_4xf32)
                .replace_all(neon_broadcast_4xf32)
                .replace_all(neon_vfmadd_4xf32_4xf32)
                .lift_alloc('A_vec : _', n_lifts=2)
                .fission_after('neon_broadcast_4xf32(_)', n_lifts=2)
                .lift_alloc('B_vec : _', n_lifts=2)
                .fission_after('neon_vld_4xf32(_) #1', n_lifts=2)
                .simplify())
```

# Step 2: Code substitution

### Use code substitution

instructions (similar to how we embedded GEBP kernel earlier) in scheduling to generate HW intrinsics in code.

Transformation **formally guaranteed** to be **equivalent** to original code.

```
erate microkernel(kernel, N reg. M reg. K blk)
     vaise Exception(f"Error: N reg must be a multiple of 4. got {N reg}"
     partial eval(M reg.N reg)
            , ['jo','ji'], perfect=True]
          oid neon_microkernel( c_code_str_Context *ctxt, struct exo_win_2f32 A,
                                   struct exo_win_2f32 B, struct exo_win_2f32 C ) {
        EX0_ASSUME(A.strides[1] == 1);
     rep
        EX0_ASSUME(B.strides[1] == 1);
        EX0_ASSUME(C.strides[1] == 1);
        float32x4_t C_reg[4][4];
        for (int i = 0; i < 4; i++) {
          for (int jo = 0; jo < 4; jo++) {</pre>
     .fis:
             C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
        for (int k = 0: k < 128: k++) {
           float32x4_t A_vec[4];
           for (int i = 0; i < 4; i++) {
             A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
           float32x4_t B_vec[4];
           for (int jo = 0; jo < 4; jo++) {</pre>
             B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
           for (int i = 0; i < 4; i++) {</pre>
             for (int jo = 0; jo < 4; jo++) {</pre>
               C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
        for (int i = 0; i < 4; i++) {
           for (int jo = 0; jo < 4; jo++) {</pre>
             vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
```

This generates a microkernel with vector intrinsics

# Step 2: Code substitution

# This can be **repeated** for any new piece of hardware to generate **hardwarespecific microkernels**

# 3. Simplified microkernel optimization

# Optimizing Performance

- Explore scheduling parameter space **easily** by using scheduling instructions to **programmatically generate code with optimizations**.
  - Removes tedium of implementing optimized code (e.g. hand-coding edge cases for tilings)
  - Formally verified correctness of transforms means no need to debug optimized code's functionality – faster iteration

```
Example: tuning register sizes
    generate_sgemm_microkernel(kernel=SGEMM_ N_reg=16, M_reg=4, K_blk=128)
                                                                                                 generate_sgemm_microkernel(kernel=SGEMM, N_reg=32, M_reg=4, K_blk=64)
 void neon_microkernel( c_code_str_Context *ctxt, struct exo_win_2f32 A,
                                                                                             void neon_microkernel( c_code_str_Context *ctxt, struct exo_win_2f32 A,
                                                                                                                     struct exo_win_2f32 B, struct exo_win_2f32 C ) {
                         struct exo_win_2f32 B, struct exo_win_2f32 C ) {
 EX0_ASSUME(A.strides[1] == 1);
                                                                                             EX0_ASSUME(A.strides[1] == 1);
 EX0_ASSUME(B.strides[1] == 1);
                                                                                             EX0_ASSUME(B.strides[1] == 1);
 EX0_ASSUME(C.strides[1] == 1);
                                                                                             EX0_ASSUME(C.strides[1] == 1);
 float32x4_t C_reg[4][4]:
                                                                                             float32x4_t C_reg[4][8];
 for (int i = 0; i < 4; i++) {
                                                                                             for (int i = 0; i < 4; i++) {
   for (int jo = 0; jo < 4; jo++) {</pre>
                                                                                               for (int jo = 0; jo < 8; jo++) {</pre>
     C_req[i][jo] = vld1q_f32(&0.data[(i) * (0.strides[0]) + (4 * jo) * (0.strides[1])]);
                                                                                                 C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
 for (int k = 0; k < 128; k++) {
                                                                                              for (int k = 0; k < 64; k++) {
   float32x4_t A_vec[4];
                                                                                               float32x4_t A_vec[4];
   for (int i = 0; i < 4; i++) {
                                                                                               for (int i = 0; i < 4; i++) {</pre>
     A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
                                                                                                 A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
   float32x4_t B_vec[4];
                                                                                               float32x4_t B_vec[8];
   for (int jo = 0; jo < 4; jo++) {</pre>
                                                                                               for (int jo = 0; jo < 8; jo++) {</pre>
     B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
                                                                                                 B_vec[jo] = vld1q_f32(\&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
   for (int i = 0; i < 4; i++) {</pre>
                                                                                               for (int i = 0; i < 4; i++) {</pre>
     for (int jo = 0; jo < 4; jo++) {</pre>
                                                                                                 for (int jo = 0; jo < 8; jo++) {</pre>
       C_reg[i][jo] = vmlag_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
                                                                                                   C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
 for (int i = 0; i < 4; i++) {
                                                                                             for (int i = 0; i < 4; i++) {</pre>
   for (int jo = 0; jo <_4:_jo++) {</pre>
                                                                                               for (int jo = 0; jo < 8; jo++) {</pre>
     vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
                                                                                                 vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
```

• I generated 64 microkernels using Exo

```
void gebp_edge_neon_microkernel_1x32( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
               EX0_ASSUME(A.strides[1] == 1);
        Qer EX0_ASSUME(B.strides[1] == 1);
EX0_ASSUME(C.strides[1] == 1);
               float32x4_t C_reg[64][8];
               for (int i = 0; i < 64; i++) {</pre>
                 for (int jo = 0; jo < 8; jo++) {</pre>
                    C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
                for (int k = 0; k < 64; k++) {
                 float32x4_t A_vec[64];
                 for (int i = 0; i < 64; i++) {</pre>
                    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
                  }
                 float32x4_t B_vec[8];
                 for (int jo = 0; jo < 8; jo++) {</pre>
                   B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
                 for (int i = 0; i < 64; i++) {</pre>
                    for (int jo = 0; jo < 8; jo++) {</pre>
                      C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
                    }
                for (int i = 0; i < 64; i++) {</pre>
                 for (int jo = 0; jo < 8; jo++) {</pre>
                    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
                  }
```

I generated 64 microkernels using Exo ightarrow

sts ppm.sep\_ens.ctreaser(1) = 0.1 ( ...com\_str\_Gottat ext, iterat es.sts.272 i, struct e

```
void gebp_edge_neon_microkernel_1x16( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
                       EX0_ASSUME(A.strides[1] == 1);
                       EX0_ASSUME(B.strides[1] == 1);
                      EX0_ASSUME(C.strides[1] == 1);
gener
                       float32x4_t C_reg[64][4];
                       for (int i = 0; i < 64; i++) {</pre>
                         for (int jo = 0; jo < 4; jo++) {</pre>
                           C_req[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
     C_reg[i][jo] = vld1q_f32(60.data[(i)
                       for (int k = 0; k < 64; k++) {
                         float32x4_t A_vec[64];
                         for (int i = 0; i < 64; i++) {</pre>
     for (int i = 0; i < 04; i++) {
for (int jo = 0; jo < 8; jo++) {
    C_reg[i][jo] = vmlaq_f32(C_reg[i][j
                           A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
                         float32x4_t B_vec[4];
      vstlg_f32(60.date[(i) * (0.strides[0]
                         for (int jo = 0; jo < 4; jo++) {</pre>
                           B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
                         ્ર
                         for (int i = 0; i < 64; i++) {
                           for (int jo = 0; jo < 4; jo++) {</pre>
                             C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
                           }
                      for (int i = 0; i < 64; i++) {</pre>
                         for (int jo = 0; jo < 4; jo++) {</pre>
                           vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
                         }
```

I generated 64 microkernels using Exo ightarrow

wids dms\_seq\_sens\_indexeased\_list(c\_com\_str\_Content ex:t, itrust en\_std\_272 #, itrust en

```
void gebp_edge_neon_microkernel_1x8( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
                       EX0_ASSUME(A.strides[1] == 1);
                       EX0_ASSUME(B.strides[1] == 1);
Qene EX0_ASSUME(C.strides[1] == 1);
                        float32x4_t C_reg[64][2];
    void gebp_edge_nesn_microkernet_1x32( c_c) for (int i = 0; i < 64; i++) {</pre>
                          for (int jo = 0; jo < 2; jo++) {</pre>
                            C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
                          }
     C_reg[i][jo] = vld1q_f32(6C.data[(i)
                       for (int k = 0; k < 64; k++) {
                          float32x4_t A_vec[64];
                          for (int i = 0; i < 64; i++) {</pre>
                            A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
                          float32x4_t B_vec[2];
                          for (int jo = 0; jo < 2; jo++) {</pre>
      vstlo_f32(6C,data[(i) * (C,strides[0]
                            B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
                          }
                          for (int i = 0; i < 64; i++) {</pre>
                            for (int jo = 0; jo < 2; jo++) {</pre>
                              C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
                        for (int i = 0: i < 64: i++) {</pre>
                          for (int jo = 0; jo < 2; jo++) {</pre>
                            vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
```

I generated 64 microkernels using Exo igodol

mid photosequence.storestaron1.http://dom/st victo, stroit exe.sth.2722 /, stroit exe.

void man\_peng\_aven\_diseawer\_llabid (\_com\_\_int\_\_fonted versi, intred ma\_sis\_JPI2 ), street ma\_sis\_JPI2 ), ma\_sisme(-version[] = 1); ma\_sisme(-version[] = 1

function = 0; i < 64; i++) {
 A\_vec[i] = vid[q,dup\_f12(60,data[(i) \* (0.strides[0]) + (k) \* (0.strides[1])]);
}</pre>

Tlasf25%\_5 B\_vec[4]; for (int js = 0; js < 4; jo++) { B\_vec[js] = vldig\_f25(&=.data[(k) \* (8.strides[8]) + (4 \* js) \* (8.strides[1])];

for (int i = 0; i < 64; i\*+) {
 for (int jo = 0; jo < 4; jo++) {
 C\_reg[i][jo] = vmlag\_f32(C\_reg[i][jo], A\_vec[i], B\_vec[jo]);
}</pre>

} for (int i= 0; i < 64; i++) { for (int j= 0; j< 4; j=++) { vstig\_f32(6c.deta[(i) + (0.strides[0]) + (4 + j=) + (0.strides[1])], C\_reg[i][j=]); } 200\_ASSUMP(.strides[1] = 1); RantXvi\_t\_c\_reg(46[2]); For (int i = 0; i < 44; i=) { for (int i = 0; i < 42; i=) { for (int i = 0; i < 0; 2; 2; 2; 1) { c\_reg[1][5] = vida\_cf2(62,63ta][1] \* (C.strides[0]) + (4 \* je) \* (C.strides[1])]

r (int k = 0; k < 64; k++) { float32x4\_t A\_vec[64]; for (int i = 0; i < 64; i++) { A\_vec[i] = vind\_nov\_f37(6A.data[(i) + (A.strides[0]) + (k) + (A.strides[1])]);

lost32x4\_t 8\_vec[2]; sr (int jo = 8; jo < 2; jo++) { 8.vec[i] = v(als.f22(60.data[{k} ≠ (0.strides[0]) + (4 × jo) \* (0.strides[1])]);

sr (int i = 0; i < 64; i++) {
for (int jo = 0; jo < 2; jo++) {
 C\_reg[i](jo] = vmtaq\_f32(C\_reg[i](jo], A\_vec[i], B\_vec[jo]);
 C\_reg[i](jo] = vmtaq\_f32(C\_reg[i](jo], A\_vec[i], B\_vec[jo]);
</pre>

} for (int i = 0; 1 < 64; i++) { for (int jo = 0; jo < 2; jo++) { vstlq.r32(60.data[(i) \* (0.strides[0]) + (6 \* jo) \* (0.strides[1])], C\_reg[i][jo]])

```
void gebp_edge_neon_microkernel_1x60( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
                      EX0_ASSUME(A.strides[1] == 1);
          Gener EX0_ASSUME(B.strides[1] == 1);
EX0_ASSUME(C.strides[1] == 1);
    void gebp_edge_nesn_microkernel_1x32( c_co float32x4_t C_reg[64][15];
                      for (int i = 0; i < 64; i++) {</pre>
                        for (int jo = 0; jo < 15; jo++) {</pre>
                          C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
     A_vec[i] = vldlq_dup_f32(6A.data[(i) *
                      for (int k = 0; k < 64; k++) {
                        float32x4_t A_vec[64];
     for (int i = 0; i < 64; i++) {</pre>
                          A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
      vstlo_f32(6C,data[(i) * (C,strides[8])
                        float32x4_t B_vec[15];
                        for (int jo = 0; jo < 15; jo++) {</pre>
                          B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
                        3
                        for (int i = 0; i < 64; i++) {</pre>
                          for (int jo = 0; jo < 15; jo++) {</pre>
                            C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
                        }
                      for (int i = 0; i < 64; i++) {</pre>
                        for (int jo = 0; jo < 15; jo++) {</pre>
                          vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
                        }
                      }
                      7
```

I generated 64 microkernels using Exo  $\bullet$ 

<pre>void gebp_edge_mean_microkernel_1x32( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2t EXO_ASSUME(A.strides(1) == 1);</pre>
<pre>EX0_ASSUME(0.strides[1] == 1);</pre>
EX0_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
for (int jo = 8; jo < 8; jo++) {
C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]); }
float32x4_t A_vec[64];
<pre>A_vec[1] = vldlq_dup_f52(6A.data[(1) * (A.strides[8]) + (k) * (A.strides[1])]); }</pre>
float32x4_t B_vec[8];
for (int in = 0; in < 8; in++) {
B_vec[jo] = vld1q_f32(68.data[(k) + (8.strides[8]) + (4 + jo) + (8.strides[1])]);
for (int je = 0; je < 8; je++) {
C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
for (int i = 0; i < 64; i++) {
for (int jo = 8; jo < 8; jo++) {
vstlq_r32(60.data[[1] * (0.strides[0]) + (4 * jo] * (0.strides[1])], C_reg[1][jo]);

vii peg.app.mex.iteraered\_life (\_com\_itr\_Contert +tri, itruit exg.db\_202 i, itruit exg.d

llasf25%\_f B\_vec[4]; for (int js = 0; js < 4; jo++) { B\_vec[js] = vlaia\_f25(&=.data[(k) \* (8.strides[8]) + (4 \* js) \* (8.strides[1])]

for (int i = 0; i < 64; i++) { for (int jo = 0; jo < 4; jo++) { C\_reg[i][jo] = vmlaq\_f32(C\_reg[i][jo], A\_vec[i], B\_vec[jo]);

} for (int i = 0; i < d4; i +=) { for (int jo = 0; jo < 4; jo++) { | vstig\_132(&C.deta[(i) \* (C.strides[0]) + (4 \* jo) + (C.strides[1])], C\_reg[i][jo]); } void gebp\_sdgs\_mean\_microkernel\_lx8( c\_code\_str\_Context Actxt, struct exo\_min\_2f52 Å, struct exo\_min\_2f52 Å, struct exo\_min\_2f52 Å, struct exo\_min\_2f52 Å) {
EX0\_ASSUME(A.strides[1] == 1);

200\_45309(C.Wride(1) = 1); Toattike:c=C-we(6)[0]; for (int 1 = 0; 1 < 46; i+) = 4 for (int jo = 0; jo < 2; jo+) { \_Cregi(1)[j = x,0:x, 722(6:), dta((1) + (C.strides(0)) + (4 + jo) + (C.strides(1))

or (int k = 0; k < 66; k++) {
 flast32x4\_t A\_vec[64];
 for (int i = 0; i < 64; i++) {
 A\_vec[1] + vin(a,ou,r32(6A,data[(1) + (A,strides[0]) + (k) + (A,strides[1])]);
 }
}

lost32x4\_t B\_vec[2]; sr (in: jo = 8; jo < 2; jo++) { B\_vec[i] = vials.r32(68.data[(k) \* (0.striges[0]) + (4 \* jo) \* (0.striges[1])]);

sr (int i = 8; i < 64; i++) {
for (int js = 0; js < 2; js++) {
 C\_reg[i][js] = vmlag\_f32(C\_reg[i][js], A\_vec[i], B\_vec[js]);
</pre>

}
for (int i = 0; i < 64; i++) {
 for (int jo = 0; jo < 2; jo++) {
 vstiq\_rf32(&G.dsta[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]).
 vstiq\_rf32(&G.dsta[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]).
 }
}</pre>

```
wide despectations accesses to 100 (_sSSS,ST_CANTAK +RIC, STORT BALANCE 2, STORT BALAN
```

```
void gebp_edge_neon_microkernel_1x52( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
                    EX0_ASSUME(A.strides[1] == 1);
QCN EX0_ASSUME(B.strides[1] == 1);
                    EX0_ASSUME(C.strides[1] == 1);
    void gebp_edge_neon_microkernel_1
                    float32x4_t C_reg[64][13];
                    for (int i = 0; i < 64; i++) {</pre>
                      for (int jo = 0; jo < 13; jo++) {</pre>
                         C_reg[i][jo] = vld1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])]);
     \frac{|f_{00}(int j_0 = 0; k < 64; k++)}{|g_{vecj0}| = vid_{12}(s).data} for (int k = 0; k < 64; k++) {
                      float32x4_t A_vec[64];
      C regfilliol = ymlag f32(C
                      for (int i = 0; i < 64; i++) {
                         A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
      vst10_f32(60.data[(i) * (0.st
                       float32x4_t B_vec[13];
                      for (int jo = 0; jo < 13; jo++) {</pre>
                         B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
    void gebp_edge_neon_microkernel_1
                       for (int i = 0; i < 64; i++) {
                         for (int jo = 0; jo < 13; jo++) {</pre>
                           C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);
    for (int jo = 8; jo < 15; jo++)

B_vec[jo] = vld1q_f32(68.data
     for (int i = 0; i < 64; i++) { for (int i = 0; i < 64; i++) {</pre>
                      for (int jo = 0; jo < 13; jo++) {</pre>
      C_reg[i][jo] = vmlaq_f32(C
                         vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
     vstin f32(&C.date[fi] * (C.st
```

I generated 64 microkernels using Exo  $\bullet$ 

stat spin.seg.mes.streamers.ltp1 = 0.5( = 0.500 \_cf \_cons.str ext., streat es.sts.272 %, streat es.sts.272 %,

 $\begin{array}{l} \text{vist} a pis-sequence acceleration (1) = 0; \\ \text{explosited} (1$ 

rlost25%\_t A\_vec[60]; for (int i = 0; i < 64; i++) { \_ A\_vec[i] = vlol\_dog\_for\_52(6A.dsta[(i) \* (A.strides[0]) + (k) \* (A.strides[1])]);

llasf25%\_f B\_vec[4]; for (int js = 0; js < 4; jo++) { B\_vec[js] = vlaig\_f25(8:.data[(k) \* (8.strides[8]) + (4 \* js) \* (8.strides[i])]

for (int i = 0; i < 64; i++) {
 for (int jo = 0; jo < 4; jo++) {
 C\_reg[i][jo] = vmlag\_f32(C\_reg[i][jo], A\_vec[i], B\_vec[jo]);
</pre>

void gebp\_sdgs\_neon\_microkernet\_lx8( c\_code\_str\_Context Actxt, struct exo\_min\_2f32 Å, struct exo\_min\_2f32 Å, struct exo\_min\_2f32 Å, struct exo\_min\_2f32 Å) {
EX0\_ASSUME(A.strides[1] == 1);

lost32x4\_t 8\_vec[2]; sr (int jo = 8; jo < 2; jo++) { 8.vec[i] = v(alm.f32(68.data[(k) ★ (0.strides[0]) + (4 ★ 10) ★ (0.strides[1])])

r (int i = 8; i < 64; i++) { for (int jo = 0; jo < 2; jo++) { | C\_reg[i][jo] = vmlaq\_f32(C\_reg[i][jo], A\_vec[i], B\_vec[jo]);

}
for (int i = 0; i < 64; i++) {
 for (int jo = 8; jo < 2; jo++) {
 vstiq.f32(6c.data[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]);
 vstiq.f32(6c.data[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]);
 }
}</pre>

vsis projections.attervative.list().attervation.a

ł

	void gebp_edge_neon_microkernel_1x48( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {	
	EXO_ASSUME(A.strides[1] == 1);	
laono	EX0_ASSUME(B.strides[1] == 1);	
i yene	EX0_ASSUME(0.strides[1] == 1);	
	float32x4_t C_reg[64][12];	
<pre>void gebp_edge_neon_microkernel_1x32( c, EX0_ASSUME(A.strides[1] == 1); EX0_ASSUME(8.strides[1] == 1);</pre>	for (int i = 0; i < 64; i++) {	ruct exo_win_
EX0_ASSUME(C.strides[1] == 1); float32x4_t C_reg[64][8];	for (int jo = 0; jo < 12; jo++) {	
<pre>for (int jo = 8; jo &lt; 8; jo++) {     C_reg[i][jo] = vld1q_f32(60.data[(i)) </pre>	C_reg[i][jo] = vld1q_f32(&0.data[(i) * (0.strides[0]) + (4 * jo) * (0.strides[1])]);	
	}	
float32x4_t A_vec[64]; for (int i = 8; i < 64; i++) {	}	
<pre>} float32x4_t 8_vec[8]; for (int io = 0; io &lt; 8; io++) {</pre>	for (int k = 0; k < 64; k++) {	
<pre>B_vec[jo] = vldiq_f32(6B.data[(k) + } fer (int i = 8; i &lt; 64; i++) f</pre>	float32x4_t A_vec[64];	
<pre>for (int jo = 0; jo &lt; 8; jo++) {     C_reg[i][jo] = vmlaq_f32(C_reg[i])</pre>	for (int i = 0; i < 64; i++) {	
	A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (κ) * (A.strides[1])]);	
<pre>for (int 1 = 0; 1 &lt; 64; 1++) {   for (int jo = 8; jo &lt; 8; jo++) {     vstlq_f32(60.data[(i) * (0.strides[)</pre>	}	
	float32x4_t B_vec[12];	
	for (int jo = 0; jo < 12; jo++) {	
	B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);	
void gebp_edge_neon_microkernel_1x68( c,	}	
EX0_ASSUME(A.strides[1] == 1); EX0_ASSUME(0.strides[1] == 1); EX0_ASSUME(C.strides[1] == 1);	for (int i = 0; i < 64; i++) {	
float32x4_t C_reg[64][15]; for (int i = 0; i < 64; i++) {	for (int jo = 0; jo < 12; jo++) {	
C_reg[i][jo] = vldlq_f32(60.data[(i)	C_reg[i][jo] = vmlaq_f32(C_reg[i][jo], A_vec[i], B_vec[jo]);	
<pre>} for (int k = 0; k &lt; 64; k++) {     float32x4_t A_vec[64];</pre>		
<pre>for (int i = 8; i &lt; 64; i++) {</pre>		
<pre>float32x4_t 8_vec[15]; for (int jo = 8; jo &lt; 15; jo++) {     8 vec[io] = vidio f32(60.data[(k) + </pre>		
} for (int i = 8; i < 64; i++) {	for (int i = 0; i < 64; i++) {	
C_reg[1][jo] = vmlaq_f32(C_reg[1]) }	for (int jo = 0; jo < 12; jo++) {	
	vstlq_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);	
<pre>for (int jo = 8; jo &lt; 15; jo++) {     vstlq_f32(&amp;C.data[(i) * (C.strides[()) }</pre>		

I generated 64 microkernels using Exo  $\bullet$ 

void gebp\_edge\_neon\_microkernel\_1x32( c\_code\_str\_Context wotxt, struct exo\_win\_2f32 A, struct exo\_win\_2f32 B, struct exo\_win\_2f32 C ) { C\_reg[i][jo] = vld1q\_f32(6C.data[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])]); A vec[i] = vldlo\_dup\_f32(64,data[(i) + (A,strides[0]) + (k) + (A,strides[1])]); C\_reg[i][jo] = vmlaq\_f32(C\_reg[i][jo], A\_vec[i], B\_vec[jo]); vstlo\_f32(60.data[(i) \* (0.strides[0]) + (4 \* io) \* (0.strides[1])], 0.reg[i][io]);

void gebp\_edge\_neon\_microkernel\_1x68( c\_code\_str\_Context \*ctxt, struct exo\_win\_2f32 A, struct exo\_win\_2f32 B, struct exo\_win\_2f32 G ) { B\_vec[jo] = vld1q\_f32(68.data[(k) \* (8.strides[0]) + (4 \* jo) \* (8.strides[1])]); vstin f32(60.dete[[i] \* (0.strides[0]) + (4 \* io) \* (0.strides[1])]. 0 realil[io]);

void gebp\_edge\_meon\_microkernel\_ixi6( c\_code\_str\_Context +otxt, struct exo\_win\_2f32 A, struct exo\_win\_2f32 B, struct exo\_win\_2f32 C ) { C.reg[i][io] = vldlg\_f32(60.data[(i) \* (0.strides[0]) + (4 \* io) \* (0.strides[1])]);

A\_vec[i] = vldlq\_dup\_f32(&A.data[(i) \* (A.strides[0]) + (k) \* (A.strides[1])]);

for (int jo = 0; jo < 4; jo++) {
 vstiq\_f32(&C.data[(3) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]);
}</pre>

void gebp\_edge\_neon\_microkernel\_1x8( c\_code\_str\_Context Actxt, struct exo\_win\_2f32 A, struct exo\_win\_2f32 B, struct exo\_win\_2f32 C ) { EX0 ASSUME(A.strides[1] == 1)

A\_vec[i] = vldiq\_dup\_f32(6A.data[(i) \* (A.strides[0]) + (k) \* (A.strides[1])]);

C\_reg[i][jo] = vmlaq\_f32(C\_reg[i][jo], A\_vec[i], B\_vec[jo]);

vstlo\_f32(&C.data[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]);

void mebbiedse neonimicrokernel 1852( c.code.str.Context wotxt, struct exc.win.2752 A, struct exc.win.2752 B, struct exc.win.2752 B) { vstiq\_f32(60.data[(i) \* (0.strides[8]) \* (4 \* jo) \* (0.strides[1])], 0\_reg[i][jo]);

void gebp\_edge\_meen\_microkernel\_1x48( c\_code\_str\_Context \*ctxt, struct exo\_min\_2f32 A, struct exo\_min\_2f32 B, struct exo\_min\_2f32 C ) { for (int jo = 8; jo < 12; jo++) {
 C.rea[i][jo] = v[d10\_f32(&C.dsta[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])]);
}</pre> vstlg\_f32(6C.data[(i) \* (C.strides[0]) + (4 \* jo) \* (C.strides[1])], C\_reg[i][jo]);

# Implications

- Generate procedures reusable across the level 3 BLIS: only need to optimize GEMM
- Define hardware interfaces by writing equivalent code: easier generation of HW-specific microkernels
- Easily **re-generate** microkernels with **different parameters:** simplifies tuning for hardware-dependent optimal performance

# Implications

• The workflow now becomes...



• Make BLIS even more portable with Exo!

# 70% of peak on SGEMM on AWS Graviton (with ARM Neon) in 300 lines of code
## Future Work

- Add support for various precision BLAS operations and data layouts (dgemm, cgemm)
- Implement remaining BLAS3 operations (TRMM, TRSM, SYR2K, etc.) and extend to BLAS2
- Autotuner integration
- Code generation for non-performance reasons (e.g. improved exception handling)

## Thanks for listening! Questions?

Julian Bellavita: jbellavita@berkeley.edu Grace Dinh: gnd@berkeley.edu

Thanks to: Gilbert Bernstein, James Demmel, Yuka Ikarashi, Alex Reinking, and many others