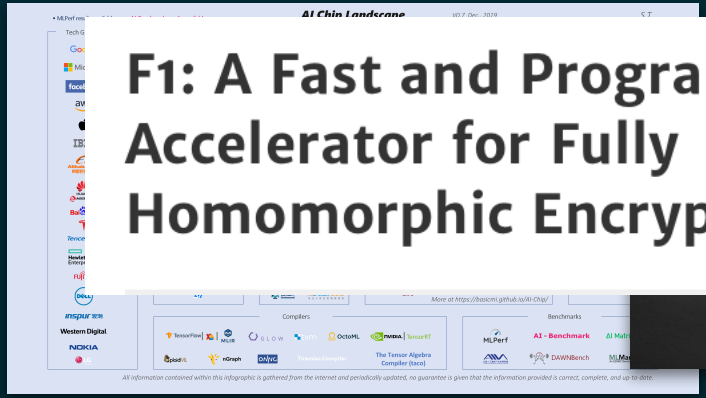
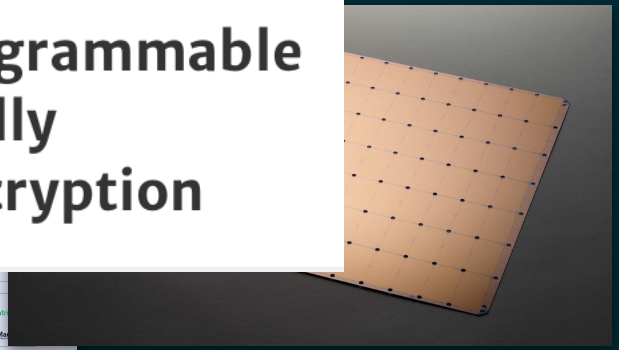
Portable Code Generation and Semi-Automatic Scheduling for BLIS

Julian Bellavita[1], Grace Dinh[1]

[1]University of California, Berkeley



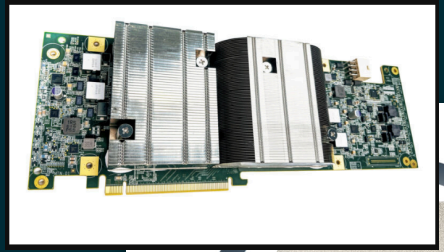
F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption

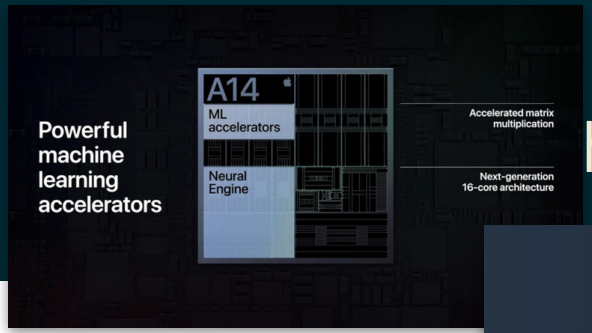


YouTube is now building its own video-transcoding chips

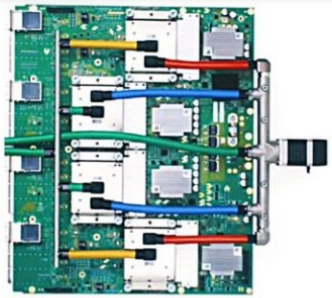
Google throws custom silicon at YouTube's massive video-transcoding workload.

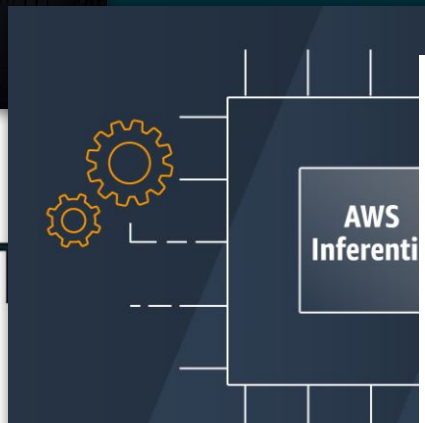
by **Ron Amadeo** - Apr 22, 2021 10:24am PST





Cambrian explosion in hardware architectures

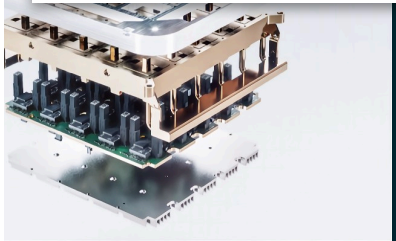




NVIDIA Hopper GPU Architecture Accelerates Dynamic Programming Up to 40x Using New DPX Instructions

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

March 22, 2022 by DION HARRIS

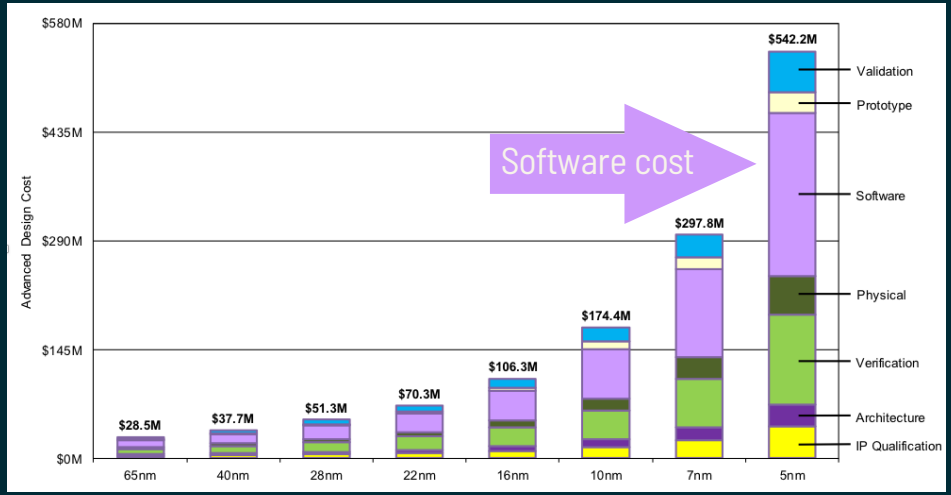


What runs on these chips?

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.

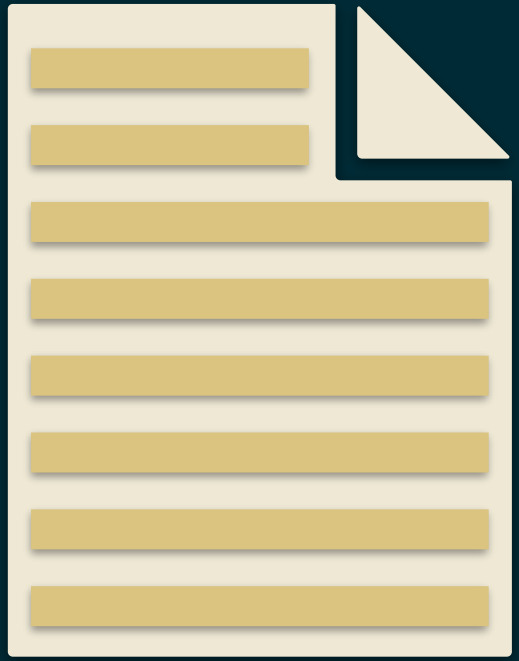
What makes performant kernels hard to implement?

- 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!

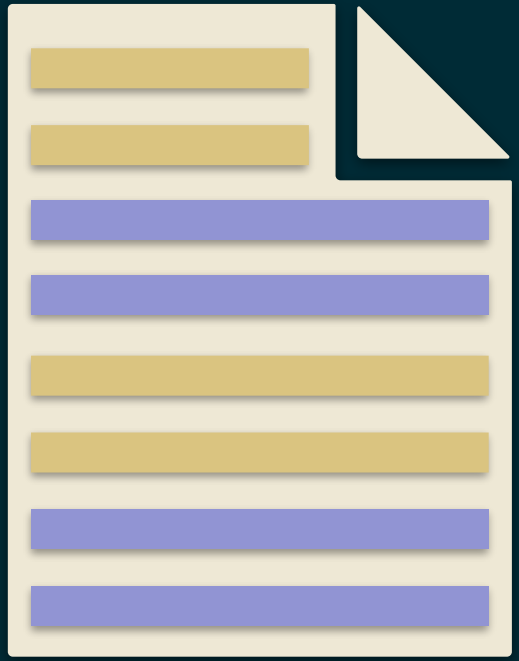
The story so far...

Performance-critical code



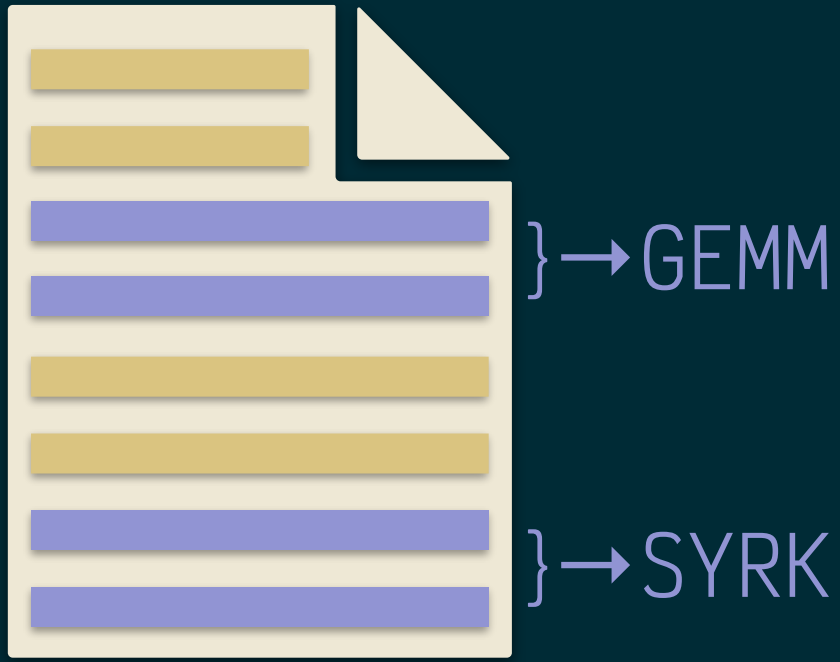
The story so far...

Performance-critical code

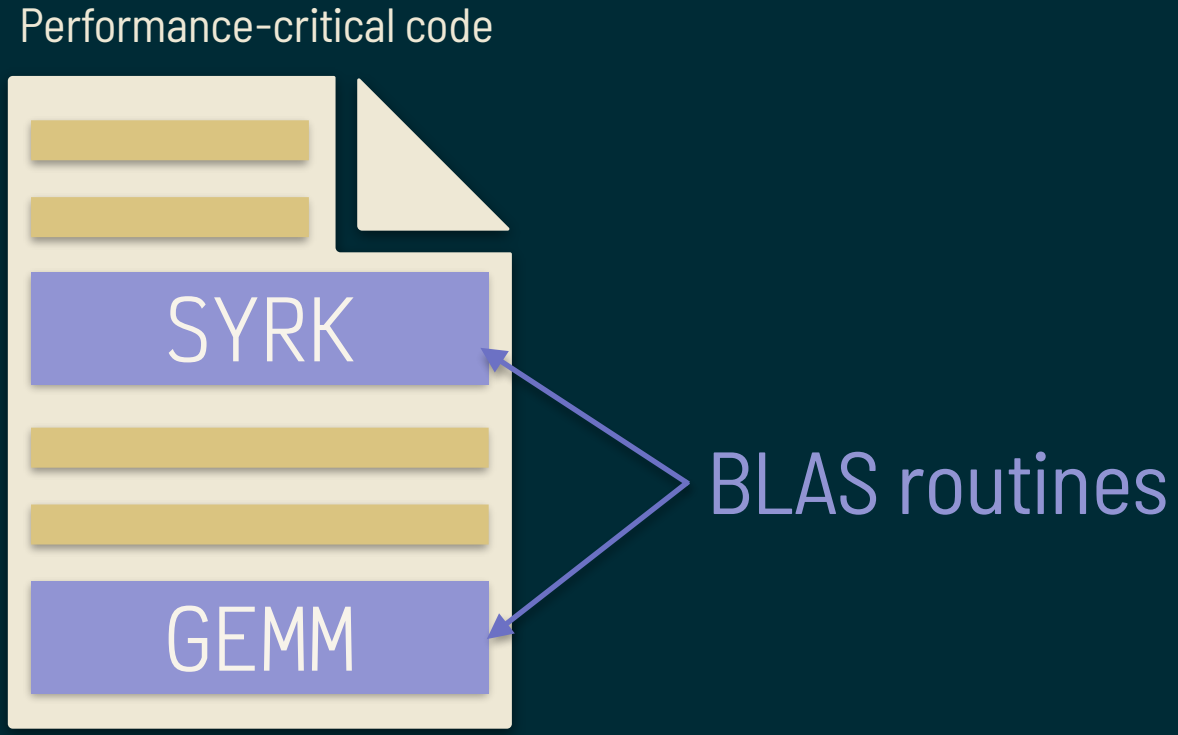


The story so far...

Performance-critical code

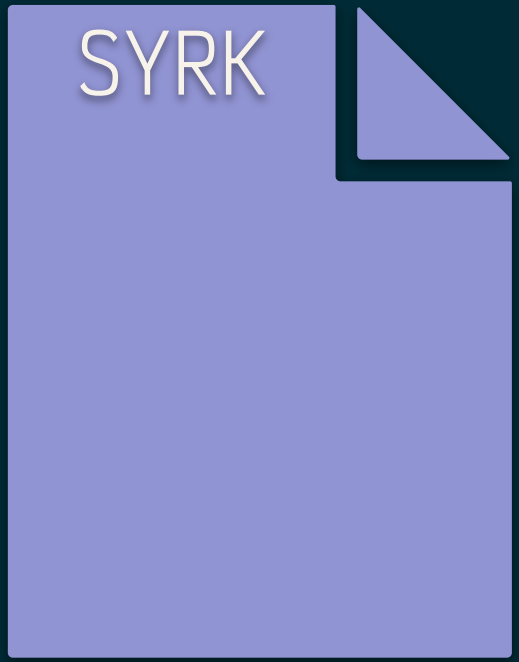


The story so far...



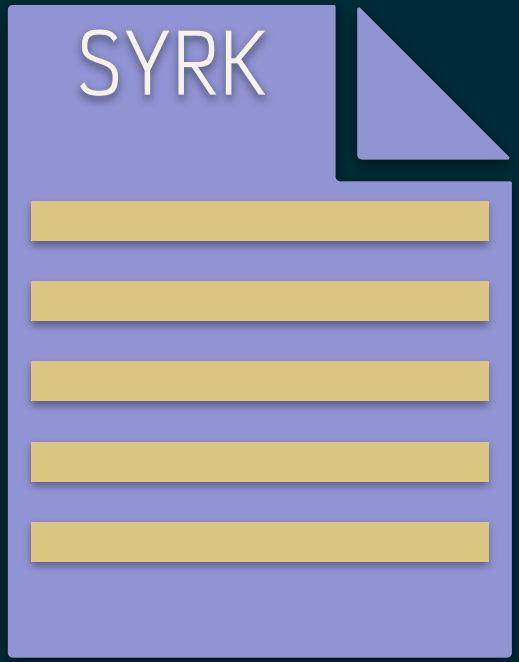
The story so far...

BLAS routine



The story so far…

BLAS routine



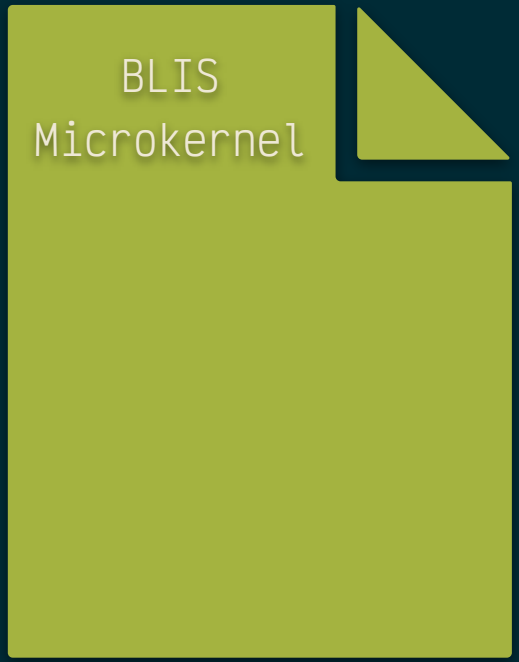
With BLIS… implement as scaffolding + architecture-specific microkernel

The story so far…

BLAS routine

With BLIS… implement as scaffolding + architecture-specific microkernel

The story so far...

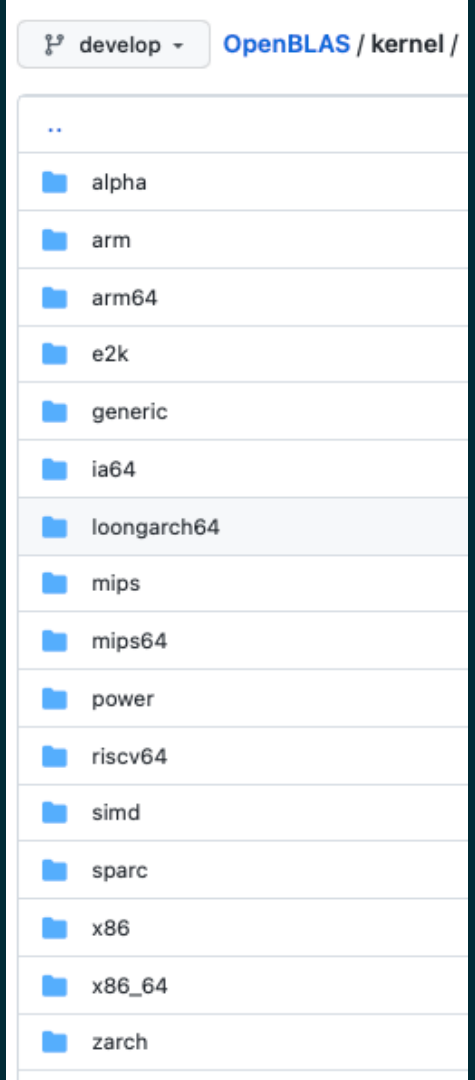


Hardware-specific implementation.

The story so far...

BLIS Microkernel

Hardware-specific implementation.



The story so far...

BLIS Microkernel

Hardware-specific implementation - still time-consuming to optimize!

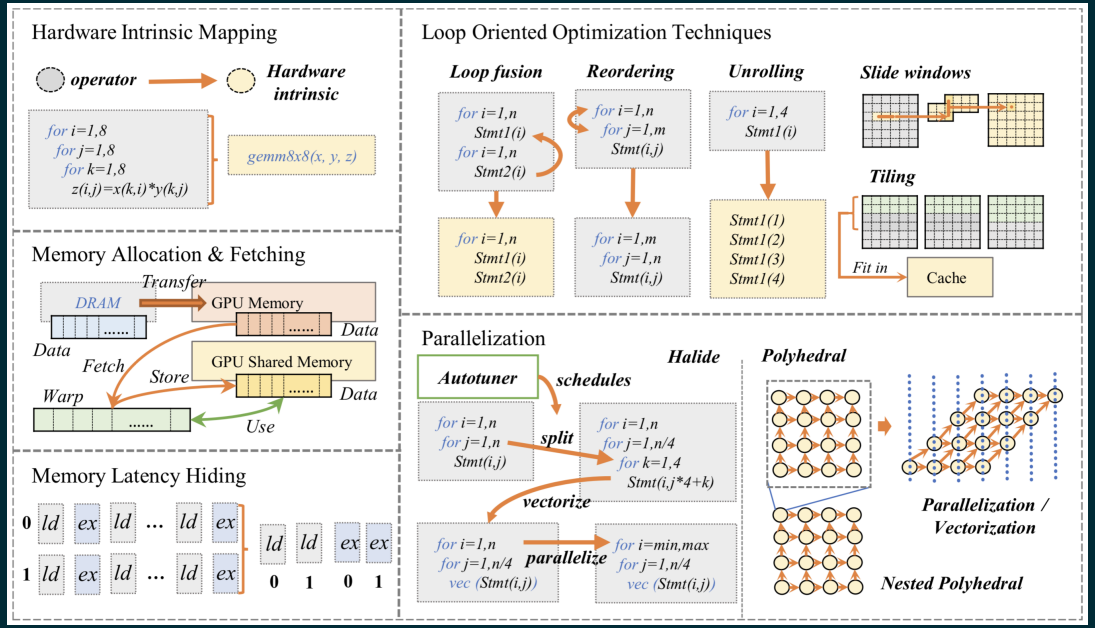


Image source: *The Deep Learning Compiler: A Comprehensive Survey* (Li et. al '20), https://arxiv.org/abs/2002.03794

```
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); 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);
        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(1, 3); DECLARE_RESULT_512(2, 3); DECLARE_RESULT_512(3, 3);

        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);

            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
            BROADCAST_LOAD_B_512(x, 2); BROADCAST_LOAD_B_512(x, 3);

            MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(2, 1); MATMUL_512(3, 1);
            MATMUL_512(0, 2); MATMUL_512(1, 2); MATMUL_512(2, 2); MATMUL_512(3, 2);
            MATMUL_512(0, 3); MATMUL_512(1, 3); MATMUL_512(2, 3); MATMUL_512(3, 3);
        }
        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, 2); STORE_512(1, 2); STORE_512(2, 2); STORE_512(3, 2);
        STORE_512(0, 3); STORE_512(1, 3); STORE_512(2, 3); STORE_512(3, 3);
    }
    for (; j < n2; j += 2) {
        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 (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);
            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
            MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(2, 1); MATMUL_512(3, 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);
    }
    for (; j < N; j++) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT_512(1, 0); DECLARE_RESULT_512(2, 0); DECLARE_RESULT_512(3, 0);
        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);
            BROADCAST_LOAD_B_512(x, 0);
            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);
    }
}
```

```
for (; i < m8; i += 8) {
    for (j = 0; j < n6; j += 6) {
        DECLARE_RESULT_512(0, 0);
        DECLARE_RESULT_512(0, 1);
        DECLARE_RESULT_512(0, 2);
        DECLARE_RESULT_512(0, 3);
        DECLARE_RESULT_512(0, 4);
        DECLARE_RESULT_512(0, 5);
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x);
            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
            BROADCAST_LOAD_B_512(x, 2); BROADCAST_LOAD_B_512(x, 3);
            BROADCAST_LOAD_B_512(x, 4); BROADCAST_LOAD_B_512(x, 5);

            MATMUL_512(0, 0);
            MATMUL_512(0, 1);
            MATMUL_512(0, 2);
            MATMUL_512(0, 3);
            MATMUL_512(0, 4);
            MATMUL_512(0, 5);
        }
        STORE_512(0, 0);
        STORE_512(0, 1);
        STORE_512(0, 2);
        STORE_512(0, 3);
        STORE_512(0, 4);
        STORE_512(0, 5);
    }
    for (; j < n2; j += 2) {
        DECLARE_RESULT_512(0, 0);
        DECLARE_RESULT_512(0, 1);
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x);
            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
            MATMUL_512(0, 0);
            MATMUL_512(0, 1);
        }
        STORE_512(0, 0);
        STORE_512(0, 1);
    }
    for (; j < N; j++) {
        DECLARE_RESULT_512(0, 0);
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x);
            BROADCAST_LOAD_B_512(x, 0);
            MATMUL_512(0, 0);
        }
        STORE_512(0, 0);
    }
}
```

```
for (; i < m16; i += 16) {
    for (j = 0; j < n6; j += 6) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        DECLARE_RESULT_512(0, 1); DECLARE_RESULT
        DECLARE_RESULT_512(0, 2); DECLARE_RESULT
        DECLARE_RESULT_512(0, 3); DECLARE_RESULT
        DECLARE_RESULT_512(0, 4); DECLARE_RESULT
        DECLARE_RESULT_512(0, 5); DECLARE_RESULT
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x);
            BROADCAST_LOAD_B_512(x, 0); BROADCAS
            BROADCAST_LOAD_B_512(x, 2); BROADCAS
            BROADCAST_LOAD_B_512(x, 4); BROADCAS

            MATMUL_512(0, 0); MATMUL_512(1, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1);
            MATMUL_512(0, 2); MATMUL_512(1, 2);
            MATMUL_512(0, 3); MATMUL_512(1, 3);
            MATMUL_512(0, 4); MATMUL_512(1, 4);
            MATMUL_512(0, 5); MATMUL_512(1, 5);
        }
        STORE_512(0, 0); STORE_512(1, 0);
        STORE_512(0, 1); STORE_512(1, 1);
        STORE_512(0, 2); STORE_512(1, 2);
        STORE_512(0, 3); STORE_512(1, 3);
        STORE_512(0, 4); STORE_512(1, 4);
        STORE_512(0, 5); STORE_512(1, 5);
    }
    for (; j < n2; j += 2) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        DECLARE_RESULT_512(0, 1); DECLARE_RESULT
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x);
            BROADCAST_LOAD_B_512(x, 0); BROADCAS
            MATMUL_512(0, 0); MATMUL_512(1, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1);
        }
        STORE_512(0, 0); STORE_512(1, 0);
        STORE_512(0, 1); STORE_512(1, 1);
    }
    for (; j < N; j++) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x);
            BROADCAST_LOAD_B_512(x, 0);
            MATMUL_512(0, 0); MATMUL_512(1, 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++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x); LOAD_A_512(2, x); LOAD_A_512(3, x);

            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
            BROADCAST_LOAD_B_512(x, 2); BROADCAST_LOAD_B_512(x, 3);

            MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(2, 1); MATMUL_512(3, 1);
            MATMUL_512(0, 2); MATMUL_512(1, 2); MATMUL_512(2, 2); MATMUL_512(3, 2);
            MATMUL_512(0, 3); MATMUL_512(1, 3); MATMUL_512(2, 3); MATMUL_512(3, 3);
        }
        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, 2); STORE_512(1, 2); STORE_512(2, 2); STORE_512(3, 2);
        STORE_512(0, 3); STORE_512(1, 3); STORE_512(2, 3); STORE_512(3, 3);
    }
    for (; j < n2; j += 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);
            BROADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
            MATMUL_512(0, 0); MATMUL_512(1, 0); MATMUL_512(2, 0); MATMUL_512(3, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1); MATMUL_512(2, 1); MATMUL_512(3, 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);
    }
    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);
            BROADCAST_LOAD_B_512(x, 0);
            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);
    }
}
```

- dgemm_beta_skylakex.c
- dgemm_kernel_16x2_haswell.S
- dgemm_kernel_16x2_skylakex.S
- dgemm_kernel_16x2_skylakex.c
- dgemm_kernel_4x4_haswell.S
- dgemm_kernel_4x8_haswell.S
- dgemm_kernel_4x8_sandy.S
- dgemm_kernel_4x8_skylakex.c
- dgemm_kernel_4x8_skylakex_2.c
- dgemm_kernel_6x4_piledriver.S
- dgemm_kernel_8x2_bulldozer.S
- dgemm_kernel_8x2_piledriver.S
- dgemm_kernel_8x8_skylakex.c
- dgemm_ncopy_2.S
- dgemm_ncopy_4.S
- dgemm_ncopy_8.S
- dgemm_ncopy_8_bulldozer.S
- dgemm_ncopy_8_skylakex.c
- dgemm_small_kernel_nn_skylakex.c
- dgemm_small_kernel_nt_skylakex.c
- dgemm_small_kernel_permit_skylakex.c
- dgemm_small_kernel_tn_skylakex.c
- dgemm_small_kernel_tt_skylakex.c
- dgemm_tcopy_16_skylakex.c
- dgemm_tcopy_2.S
- dgemm_tcopy_4.S
- dgemm_tcopy_8.S
- dgemm_tcopy_8_bulldozer.S
- dgemm_tcopy_8_skylakex.c

```
+= 8) {
j < n6; j += 6) {
RESULT_512(0, 0);
RESULT_512(0, 1);
RESULT_512(0, 2);
RESULT_512(0, 3);
RESULT_512(0, 4);
RESULT_512(0, 5);
 0; k < K; k++) {
_A_512(0, x);
ADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
ADCAST_LOAD_B_512(x, 2); BROADCAST_LOAD_B_512(x, 3);
ADCAST_LOAD_B_512(x, 4); BROADCAST_LOAD_B_512(x, 5);

UL_512(0, 0);
UL_512(0, 1);
UL_512(0, 2);
UL_512(0, 3);
UL_512(0, 4);
UL_512(0, 5);

2(0, 0);
2(0, 1);
2(0, 2);
2(0, 3);
2(0, 4);
2(0, 5);

2; j += 2) {
RESULT_512(0, 0);
RESULT_512(0, 1);
 0; k < K; k++) {
_A_512(0, x);
ADCAST_LOAD_B_512(x, 0); BROADCAST_LOAD_B_512(x, 1);
UL_512(0, 0);
UL_512(0, 1);

2(0, 0);
2(0, 1);

; j++) {
RESULT_512(0, 0);
 0; k < K; k++) {
_A_512(0, x);
ADCAST_LOAD_B_512(x, 0);
UL_512(0, 0);

2(0, 0);
```

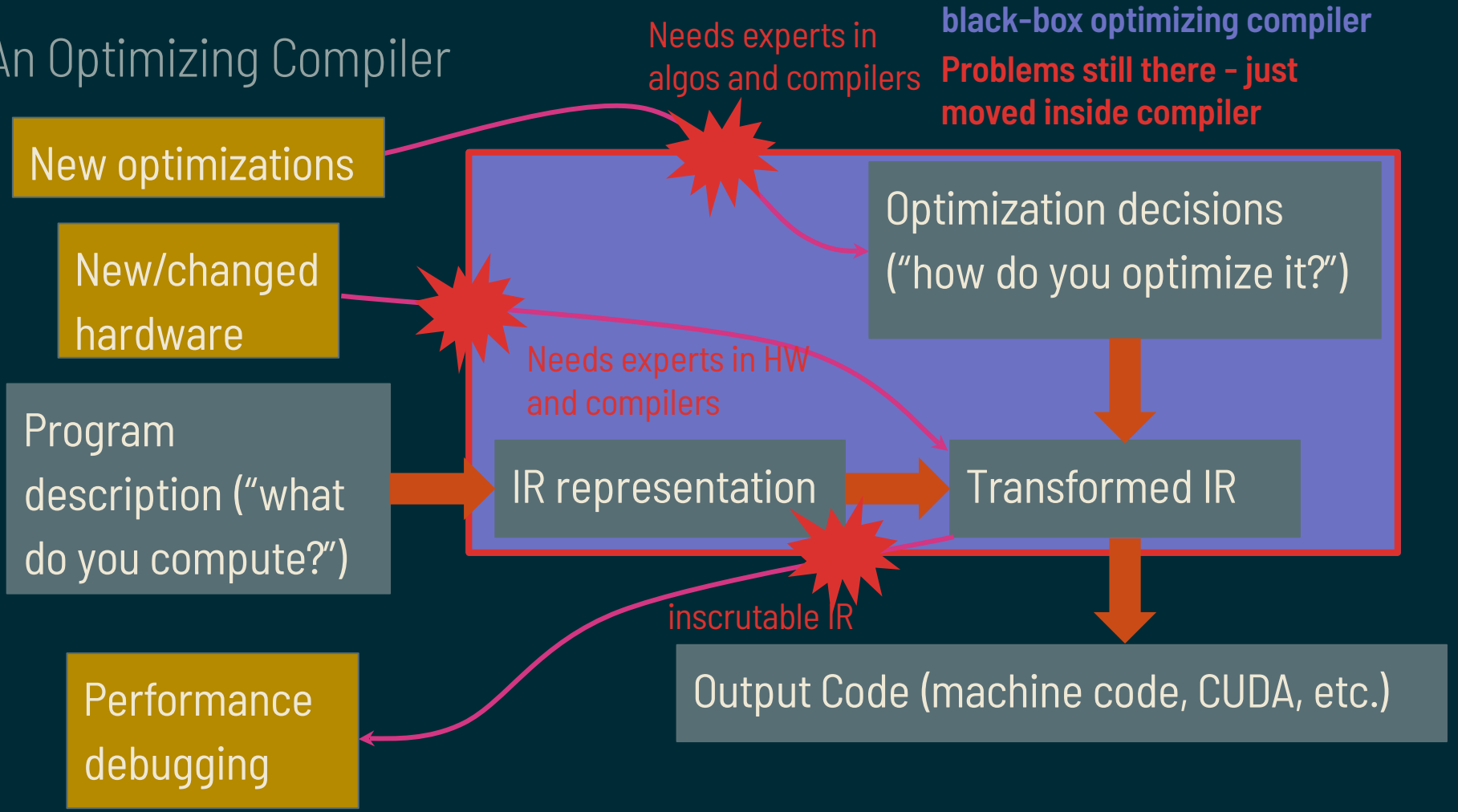
```
for (; i < m16; i += 16) {
    for (j = 0; j < n6; j += 6) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        DECLARE_RESULT_512(0, 1); DECLARE_RESULT
        DECLARE_RESULT_512(0, 2); DECLARE_RESULT
        DECLARE_RESULT_512(0, 3); DECLARE_RESULT
        DECLARE_RESULT_512(0, 4); DECLARE_RESULT
        DECLARE_RESULT_512(0, 5); DECLARE_RESULT
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x);
            BROADCAST_LOAD_B_512(x, 0); BROADCAS
            BROADCAST_LOAD_B_512(x, 2); BROADCAS
            BROADCAST_LOAD_B_512(x, 4); BROADCAS

            MATMUL_512(0, 0); MATMUL_512(1, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1);
            MATMUL_512(0, 2); MATMUL_512(1, 2);
            MATMUL_512(0, 3); MATMUL_512(1, 3);
            MATMUL_512(0, 4); MATMUL_512(1, 4);
            MATMUL_512(0, 5); MATMUL_512(1, 5);
        }
        STORE_512(0, 0); STORE_512(1, 0);
        STORE_512(0, 1); STORE_512(1, 1);
        STORE_512(0, 2); STORE_512(1, 2);
        STORE_512(0, 3); STORE_512(1, 3);
        STORE_512(0, 4); STORE_512(1, 4);
        STORE_512(0, 5); STORE_512(1, 5);
    }
    for (; j < n2; j += 2) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        DECLARE_RESULT_512(0, 1); DECLARE_RESULT
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x);
            BROADCAST_LOAD_B_512(x, 0); BROADCAS
            MATMUL_512(0, 0); MATMUL_512(1, 0);
            MATMUL_512(0, 1); MATMUL_512(1, 1);
        }
        STORE_512(0, 0); STORE_512(1, 0);
        STORE_512(0, 1); STORE_512(1, 1);
    }
    for (; j < N; j++) {
        DECLARE_RESULT_512(0, 0); DECLARE_RESULT
        for (k = 0; k < K; k++) {
            LOAD_A_512(0, x); LOAD_A_512(1, x);
            BROADCAST_LOAD_B_512(x, 0);
            MATMUL_512(0, 0); MATMUL_512(1, 0);
        }
        STORE_512(0, 0); STORE_512(1, 0);
    }
}
```

How do we make writing microkernels easier?

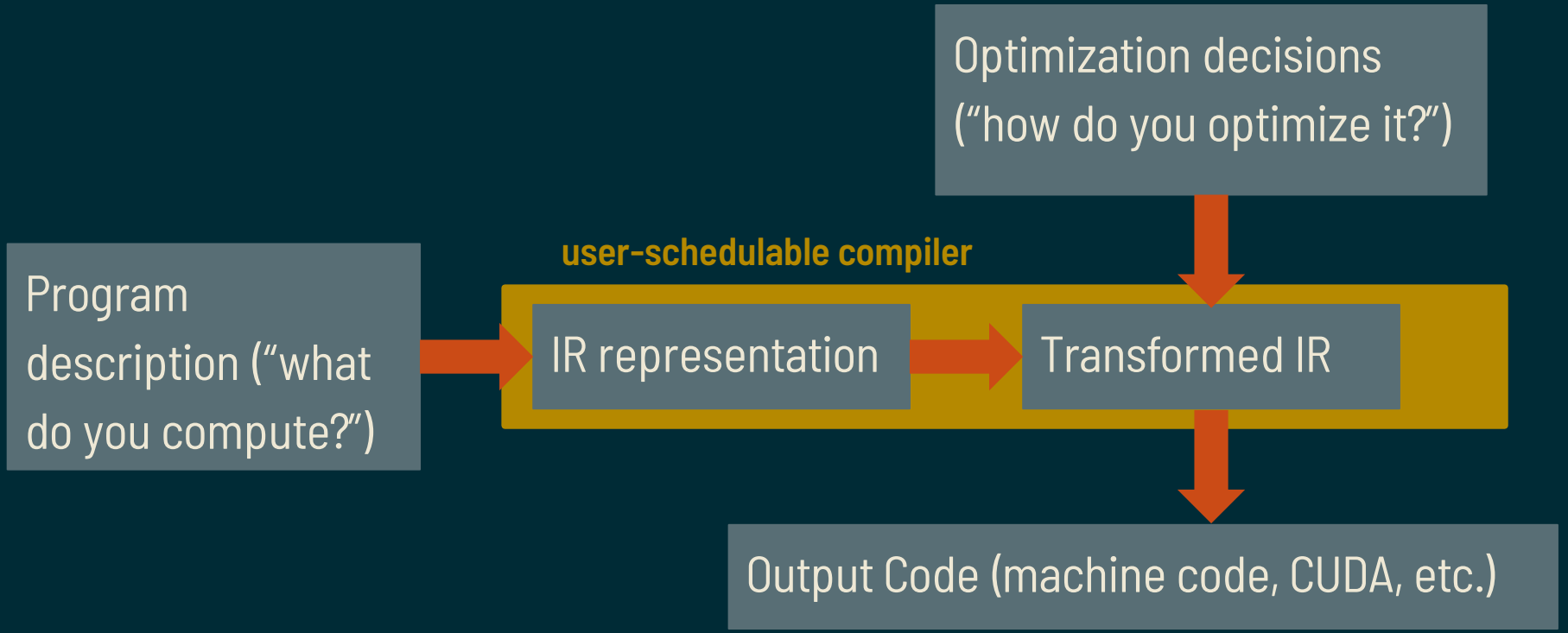
Code generation

- Instead of writing hand-optimized code, have the compiler do it for you.
  - Compiler optimizations (`-O3`, `-Ofast` 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.

An Optimizing Compiler



User Scheduling



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

Simple application code

```
def new_sgemm():
    @proc
    def sgemm_full(
        N: size,
        M: size,
        K: size,
        C: f32[N, M] @ DRAM,
        A: f32[N, K] @ DRAM,
        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.reorder('i_in #0','k')
matmul_c_i8 = matmul_c_i8.reorder('j_in #0','k')
```

Human-written schedule

Autotuner + crowdsourced perf data

Model-based optimizer

Simple application code

```
def new_sgemm():
  @proc
  def sgemm_full(
      N: size,
      M: size,
      K: size,
      C: f32[N, M] @ DRAM,
      A: f32[N, K] @ DRAM,
      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.reorder('i_in #0','k')
matmul_c_i8 = matmul_c_i8.reorder('j_in #0','k')
```

Human-written schedule

Autotuner + crowdsourced perf data

Model-based optimizer

Simple application code

```
def new_sgemm():
    @proc
    def sgemm_full(
        N: size,
        M: size,
        K: size,
        C: f32[N, M] @ DRAM,
        A: f32[N, K] @ DRAM,
        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-spec

Human-written schedule

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 application code

```
def new_sgemm():
    @proc
    def sgemm_full(
        N: size,
        M: size,
        K: size,
        C: f32[N, M] @ DRAM,
        A: f32[N, K] @ DRAM,
        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.reorder('i_in #0','k')
matmul_c_i8 = matmul_c_i8.reorder('j_in #0','k')
```

Human-written schedule

Autotuner + crowdsourced perf data

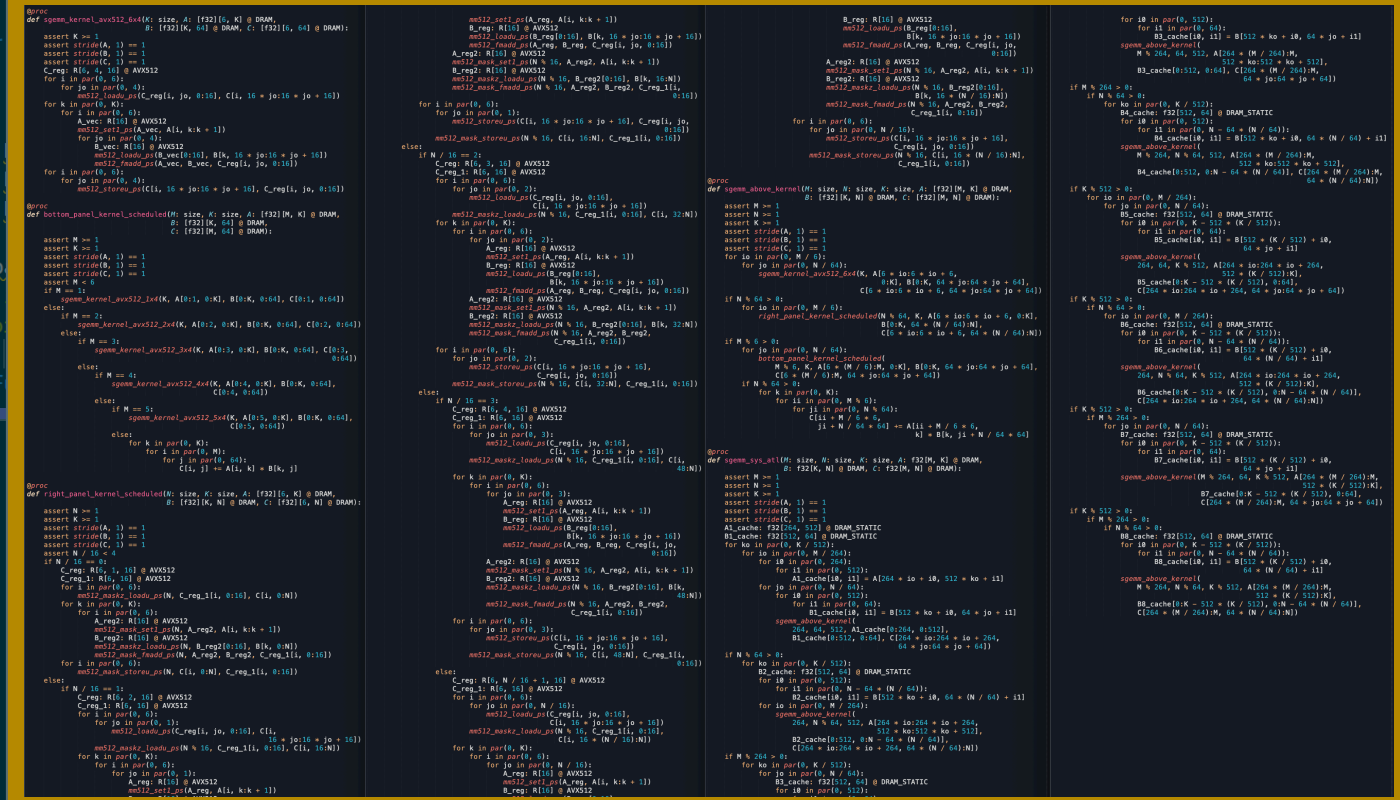
Model-based optimizer

+

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]
```

Optimized Code



Simple application code

```
def new_sgemm():
    @proc
    def sgemm_full(
        N: size,
        M: size,
        K: size,
        C: f32[N, M] @ DRAM,
        A: f32[N, K] @ DRAM,
        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.reorder('i_in #0','j')
matmul_c_i8 = matmul_c_i8.split('j #0',16,['j','j_in'], perfect=True)
matmul_c_i8.lift_alloc('res : _ #0', n_lifts=1)
matmul_c_i8.lift_alloc('res : _ #0', n_lifts=1, mode='col', size=16)
matmul_c_i8.lift_alloc('res : _ #0', n_lifts=2)
matmul_c_i8.fission_after('res[_] = 0.0 #0', n_lifts=2)
matmul_c_i8.fission_after('for k in _:_ #0', n_lifts=2)
matmul_c_i8.reorder('i_in #0','k')
matmul_c_i8.reorder('j_in #0','k')
```

Human-written schedule

Autotuner + crowdsourced perf data

Model-based optimizer

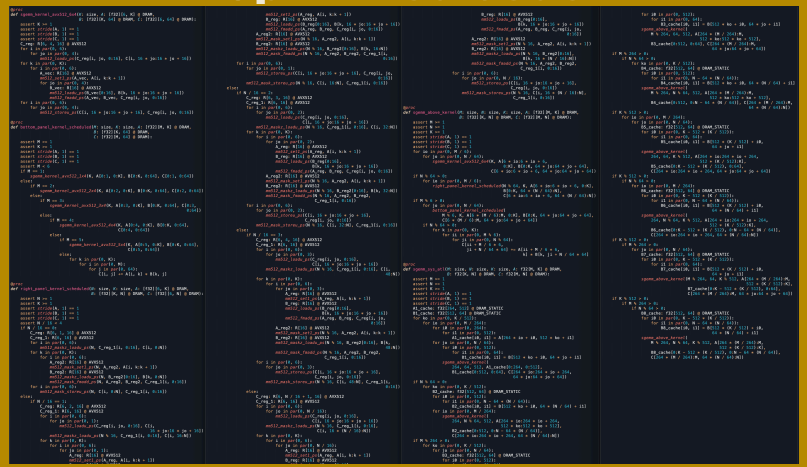
+

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]
```

=

Optimized Code



Why Exo?

- 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

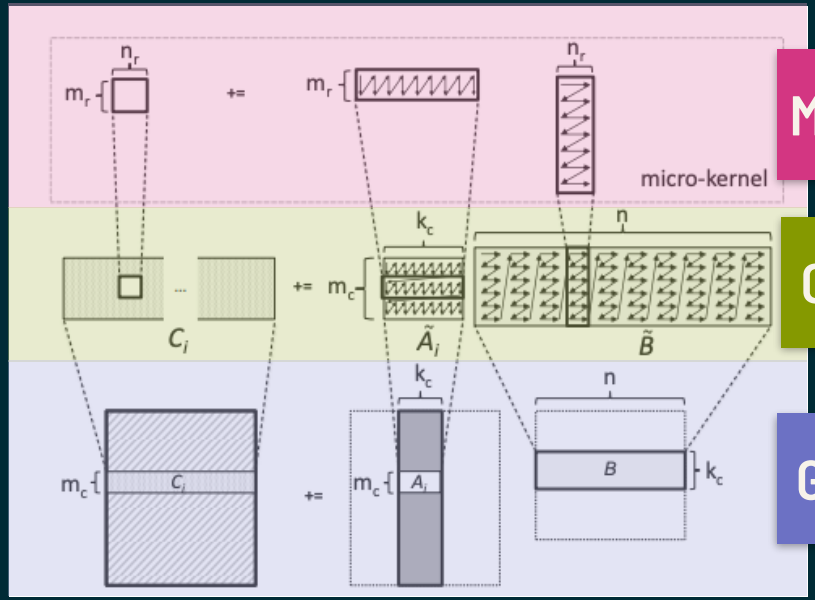
Exo + **one** undergrad + **six** weeks = **70% of peak** on **3 L3 BLIS** routines.

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)



Microkernel: $A_{reg}[m_{reg}, k_{blk}] \times B_{reg}[k_{blk}, n_{reg}]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

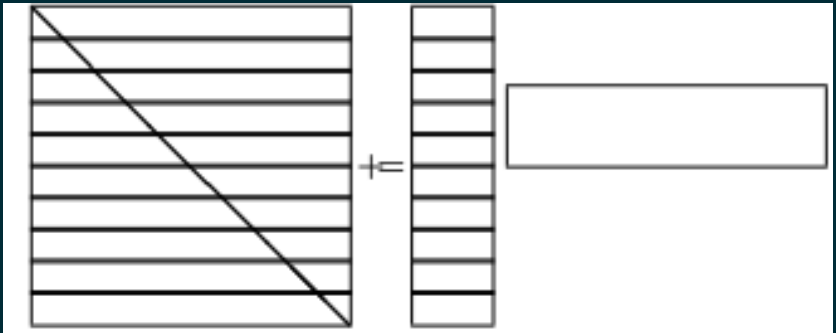
GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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



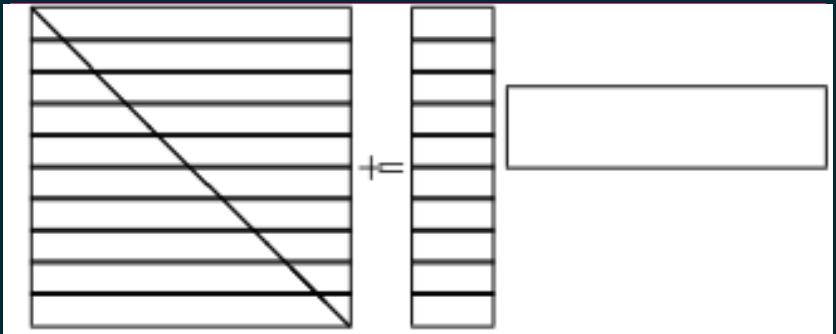
Modified GEPP procedure writes to varied-length panels of C

The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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



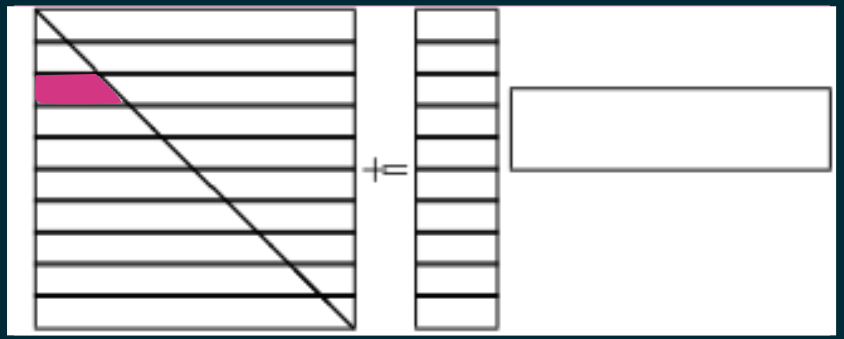
Modified GEPP procedure writes to varied-length panels of C

The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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



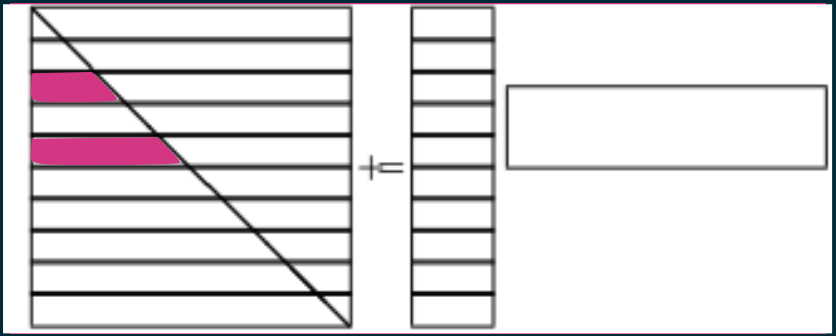
Modified GEPP procedure writes to varied-length panels of C

The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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

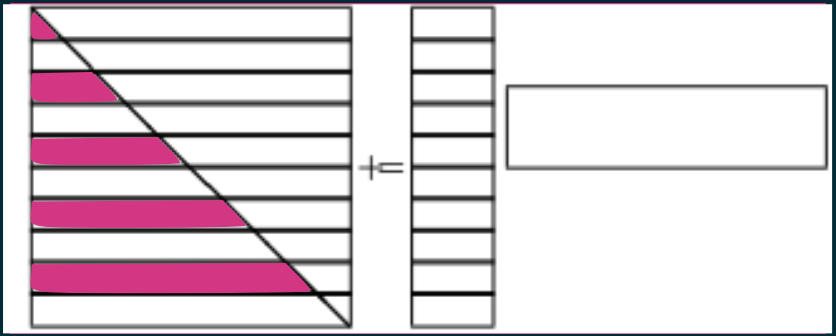


Modified GEPP procedure writes to varied-length panels of C

The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$ **GEBP**: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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



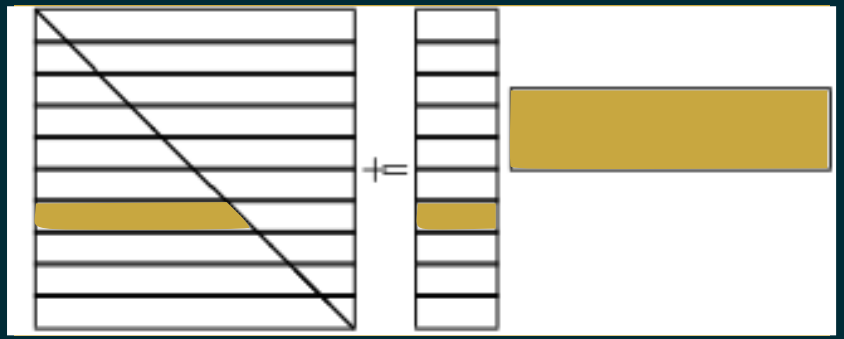
Modified GEPP procedure writes to varied-length panels of C

The BLIS approach

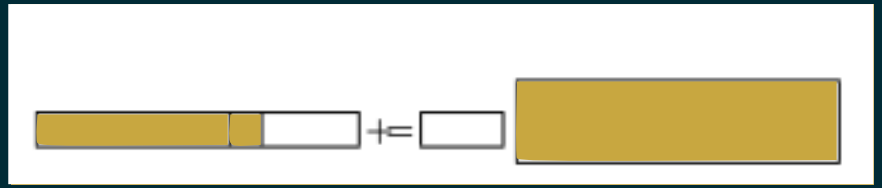
GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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



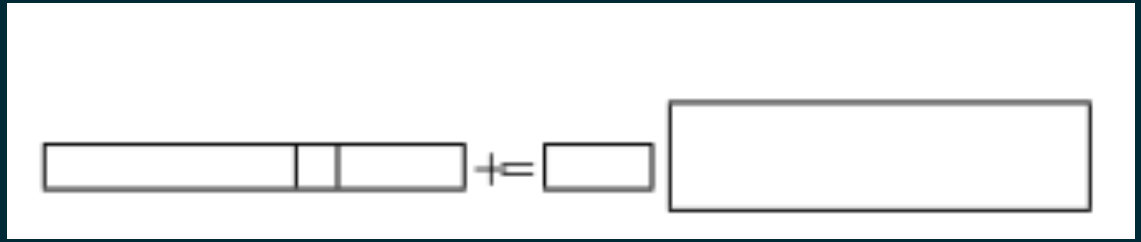
Updating a single row looks like this

The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$

GEBP: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

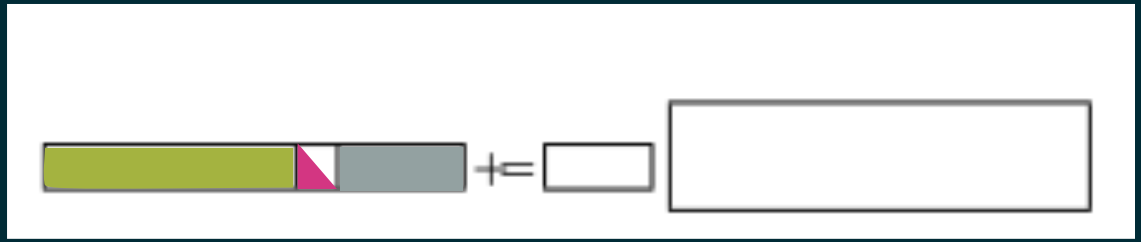
- All level 3 BLIS operations expressible in terms of GEPP, GEBP, and microkernel
- Example: SYRK (symmetric rank-K update) on lower-triangular part of C



The BLIS approach

GEPP: $A_{panel}[m, k_{blk}] \times B_{blk}[k_{blk}, n]$ **GEBP**: $A_{blk}[m_{blk}, k_{blk}] \times B_{blk}[k_{blk}, n]$

- 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

The GEBP kernel

Vanilla SGEMM...

```
@proc
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]
```

The GEBP kernel

Vanilla SGEMM...

```
@proc
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]
```

The GEBP kernel

Vanilla SGEMM...

```
@proc
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]
```

...with **scheduling instructions**

```
#Generates a GEBP kernel
def generate_GEBP(Kernel, M_blk, K_b
    GEBP = (Kernel
            #Set loop bounds to blo
            .partial_eval(M=M_c)
            .partial_eval(K=K_c)
            #Tile the block of A and the panel of B
            .split('i', M_r, ['io', 'ii'], tail='cut_and_guard')
            .split('j', N_r, ['jo', 'ji'], tail='cut_and_gua
            #Handle edge case
            .fission_after('for jo in _: _', n_lifts=2)
            .fission_after('for io in _: _', n_lifts=2)
            #reorder so loop ordering is same as microkernel
            .reorder('ii','jo')
            #insert microkernel
            .replace_all(microkernel)
            .call_eqv(neon_microkernel, 'microkernel(_)')
            #Stage strip of B
            .reorder('io', 'jo')
            .stage_mem(f'B[0:{K_c},'
                       f'{N_r}*jo:{N_r}*jo+{N_r}]',
                       'B_strip', 'for io in _:_ #0')
            .simplify())
    return GEBP
```

Refine to a specific block size

Tile loops

Split off edge cases

Reorder loops

Substitute in optimized microkernel

The GEBP kernel

Vanilla SGEMM...

```
@proc
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]
```

...with **scheduling instructions**

```
#Generates a GEBP kernel
def generate_GEBP(Kernel, M_blk, K_b
    GEBP = (Kernel
            #Set loop bounds to blo
            .partial_eval(M=M_c)
            .partial_eval(N=K_c)
            #Tile the block of A and the panel of B
            .split('i', M_r, ['io', 'ii'], tail='cut_and_guard')
            .split('j', N_r, ['jo', 'ji'], tail='cut_and_gu
            #Handle edge case
            .fission_after('for jo in _: _', n_lifts=2)
            .fission_after('for io in _: _', n_lifts=2)
            #reorder so loop ordering is same as microkernel
            .reorder('ii','jo')
            #insert microkernel
            .replace_all(microkernel)
            .call_eqv(neon_microkernel, 'microkernel(_)')
            #Stage strip of B
            .reorder('io', 'jo')
            .stage_mem(f'B[0:{K_c},'
                       f'{N_r}*jo:{N_r}*jo+{N_r}]',
                       'B_strip', 'for io in _:_ #0')
            .simplify())
    return GEBP
```

Refine to a specific kernel size

Tile loops

Split off edge cases

Reorder loops

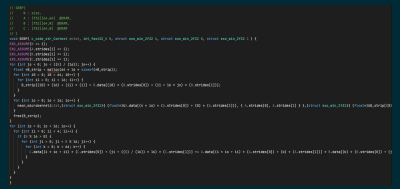
Substitute in optimized microkernel

...generates an **optimized kernel...**

```
// GEBP(
//     N : size,
//     A : [f32][64,64]  @DRAM,
//     B : [f32][64,N]  @DRAM,
//     C : [f32][64,N]  @DRAM
// )
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(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
for (int jo = 0; jo < ((N) / (16)); jo++) {
  float *B_strip = malloc(64 * 16 * sizeof(*B_strip));
  for (int i0 = 0; i0 < 64; i0++) {
    for (int i1 = 0; i1 < 16; i1++) {
      B_strip[(i0) * (16) + (i1) * (1)] = B.data[(i0) * (B.strides[0]) + (i1 + 16 * jo) * (B.strides[1])];
    }
  }
  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)
  }
  free(B_strip);
}
for (int io = 0; io < 16; io++) {
  for (int ii = 0; ii < 4; ii++) {
    if (N % 16 > 0) {
      for (int ji = 0; ji < N % 16; ji++) {
        for (int k = 0; k < 64; k++) {
          C.data[(4 * io + ii) * (C.strides[0]) + (ji + ((N) / (16)) * 16) * (C.strides[1])] += A.data[(4 * io + ii) * (A.strides[0]) + (k) * (A.strides[1])] * B.data[(k) * (B.strides[0]) + (j
        }
      }
    }
  }
}
}
```

The GEBP kernel

...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 ) {
EXO_ASSUME(M >= 1);
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(A_t.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
for (int i = 0; i < M; i++) {
  ...
        if (i == 6) {
          for (int jo = 0; jo < 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++) {
            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]
            }
          }
        } else {
          if (i == 7) {
            for (int jo = 0; jo < 1; jo++) {
              GEBP_scheduled(ctx,(struct exo_win_2f32){ (float*)&A.data[(7) * (A.strides[0]) + (0) * (A.strides[1])], { A.strides[0], A.strides[1] } },(struct exo_win_2f32){ (float*)&A_
            }
            for (int j = 0; j < 4; j++) {
              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[
              }
            }
          } else {
            if (i == 8) {
              for (int jo = 0; jo < 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.stride
                }
              }
            } else {
              if (i == 9) {
                for (int jo = 0; jo < 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++) {
                  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.stri
                  }
                }
```

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)

```
# Broadcast
@instr('{dst_data} = vld1q_dup_f32(&{src_data});')
def neon_broadcast_4xf32(
        dst: [f32][4] @ Neon4f,
        src: [f32][1] @ DRAM
):
    assert stride(dst, 0) == 1

    for i in par(0, 4):
        dst[i] = src[0]
```

```
# Load
@instr('{dst_data} = vld1q_f32(&{src_data});')
def neon_vld_4xf32(
        dst: [f32][4] @ Neon4f,
        src: [f32][4] @ DRAM
):
    assert stride(src, 0) == 1
    assert stride(dst, 0) == 1

    for i in par(0, 4):
        dst[i] = src[i]
```

```
# Store
@instr('vst1q_f32(&{dst_data}, {src_data});')
def neon_vst_4xf32(
        dst: [f32][4] @ DRAM,
        src: [f32][4] @ Neon4f
):
    assert stride(src, 0) == 1
    assert stride(dst, 0) == 1

    for i in par(0, 4):
        dst[i] = src[i]
```

```
# FMA
@instr('{dst_data} = vmulq_f32({lhs_data}, {rhs_data});')
def neon_vmul_4xf32(
        dst: [f32][4] @ Neon4f,
        lhs: [f32][4] @ Neon4f,
        rhs: [f32][4] @ Neon4f
):
    assert stride(dst, 0) == 1
    assert stride(lhs, 0) == 1
    assert stride(rhs, 0) == 1

    for i in par(0, 4):
        dst[i] = lhs[i] * rhs[i]
```

Step 1: Hardware Backend

```
# FMA
@instr('{dst_data} = vmulq_f32({lhs_data}, {rhs_data});')
def neon_vmul_4xf32(
        dst: [f32][4] @ Neon4f,
        lhs: [f32][4] @ Neon4f,
        rhs: [f32][4] @ Neon4f
):
    assert stride(dst, 0) == 1
    assert stride(lhs, 0) == 1
    assert stride(rhs, 0) == 1

    for i in par(0, 4):
        dst[i] = lhs[i] * rhs[i]
```

This HW instruction...

...operating on Neon vector registers...

...is equivalent to this Python code!

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_reg 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_seq('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.

```
def generate_microkernel(kernel, M_reg, N_reg, K_blk):
    if M_reg%4:
        raise Exception(f"Error: M_reg must be a multiple of 4, got {M_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_seq('for k in _: _')
            .sta
            .lift
            .dou
            .repl
            .repl
            .set_
            .sta
            .sta
            .repl
            .repl
            .repl
            .lift
            .fiss
            .lift
            .fiss
            .simp
```

```
void neon_microkernel( 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);
EXO_ASSUME(B.strides[1] == 1);
EXO_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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 4; 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 < 4; i++) {
  for (int jo = 0; jo < 4; jo++) {
    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 **hardware-specific 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)
```

```
void neon_microkernel( 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);
EXO_ASSUME(B.strides[1] == 1);
EXO_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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 4; 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 < 4; i++) {
  for (int jo = 0; jo < 4; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

```
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,
                       struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[4][8];
for (int i = 0; i < 4; i++) {
  for (int jo = 0; jo < 8; jo++) {
    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[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[8];
  for (int jo = 0; jo < 8; jo++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 4; i++) {
    for (int jo = 0; jo < 8; 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 jo = 0; jo < 8; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

- I generated 64 microkernels using Exo

A more extreme example

- I ger

```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 8; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 8; 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 = 0; jo < 8; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

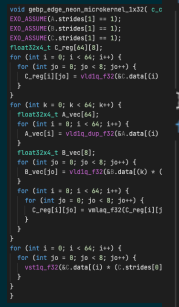
A more extreme example

- 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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 8; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 8; 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 = 0; jo < 8; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

- I gene



```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][4];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.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 < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

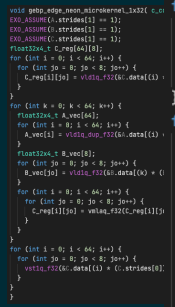
- I generated 64 microkernels using Exo

```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 8; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 8; 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 = 0; jo < 8; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][4];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.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 < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

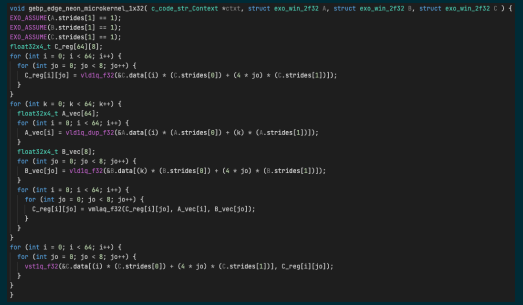
- I gener



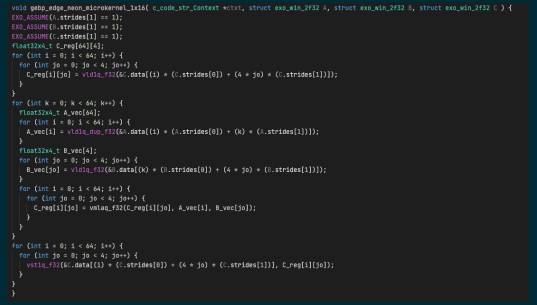
```
void gebp_edge_neon_microkernel_1x8( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][2];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 2; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; 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 = 0; jo < 2; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

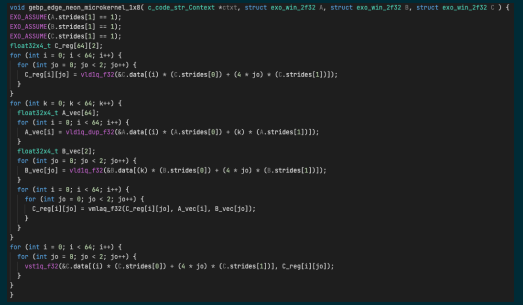
- I generated 64 microkernels using Exo



```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 8; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 8; 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 = 0; jo < 8; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```



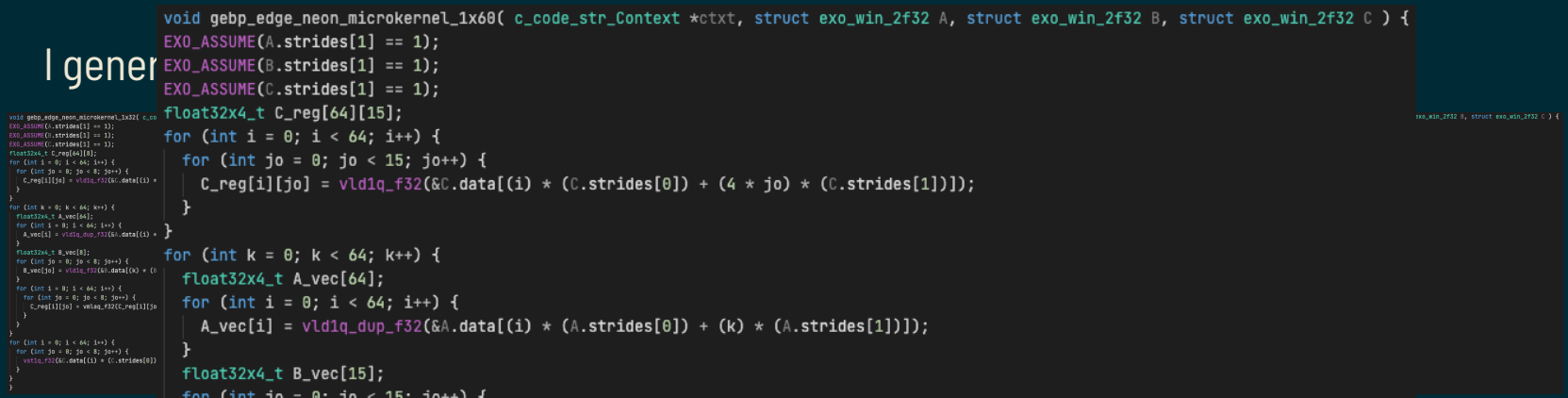
```
void gebp_edge_neon_microkernel_1x16( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][4];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.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 < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```



```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][2];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 2; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; 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 = 0; jo < 2; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

- I gener



```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][15];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 15; jo++) {
    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++) {
    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
  }
  float32x4_t B_vec[15];
  for (int jo = 0; jo < 15; jo++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 15; 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 = 0; jo < 15; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

- 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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 8; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 8; 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 = 0; jo < 8; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

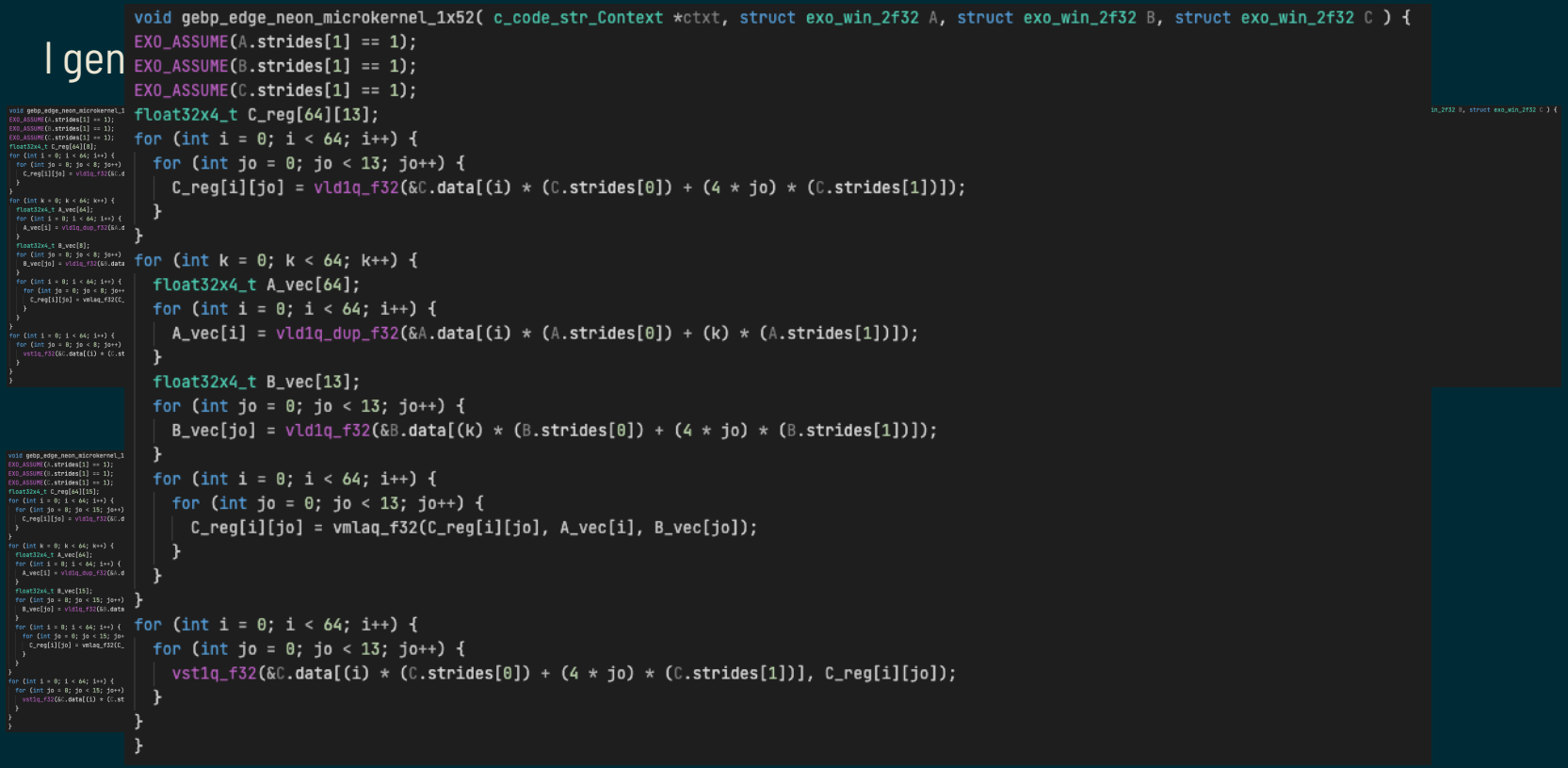
```
void gebp_edge_neon_microkernel_1x16( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][4];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.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 < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][2];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 2; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; 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 = 0; jo < 2; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][15];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 15; jo++) {
    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++) {
    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
  }
  float32x4_t B_vec[15];
  for (int jo = 0; jo < 15; jo++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 15; 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 = 0; jo < 15; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

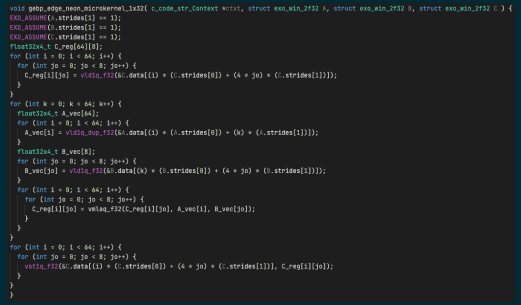
- I gen



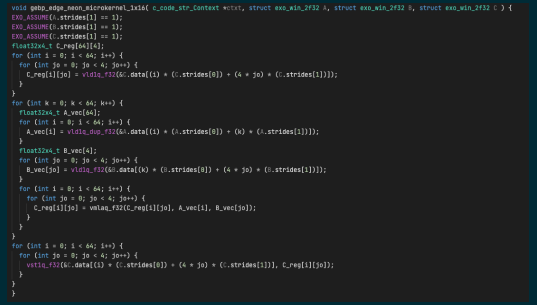
```
void gebp_edge_neon_microkernel_1x52( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][13];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 13; jo++) {
    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++) {
    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
  }
  float32x4_t B_vec[13];
  for (int jo = 0; jo < 13; jo++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 13; 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 = 0; jo < 13; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

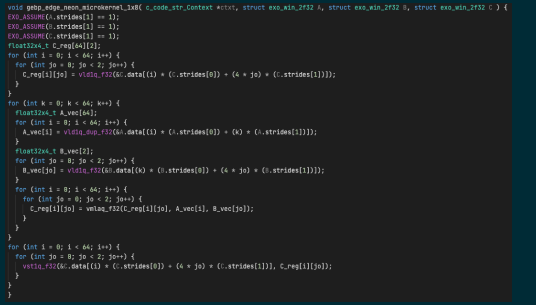
- I generated 64 microkernels using Exo



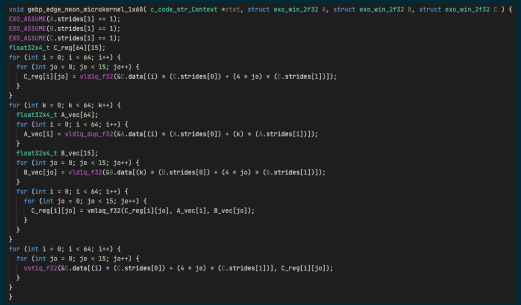
```
void gebp_edge_neon_microkernel_1x32( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][8];
for (int i = 0; i < 64; i++) {
  for (int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



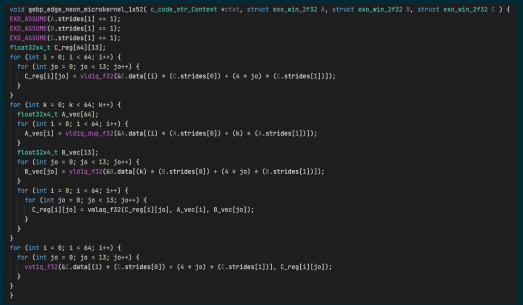
```
void gebp_edge_neon_microkernel_1x16( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][4];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.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 < 64; i++) {
  for (int jo = 0; jo < 4; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```



```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][2];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 2; jo++) {
    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++) {
    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++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; 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 = 0; jo < 2; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```



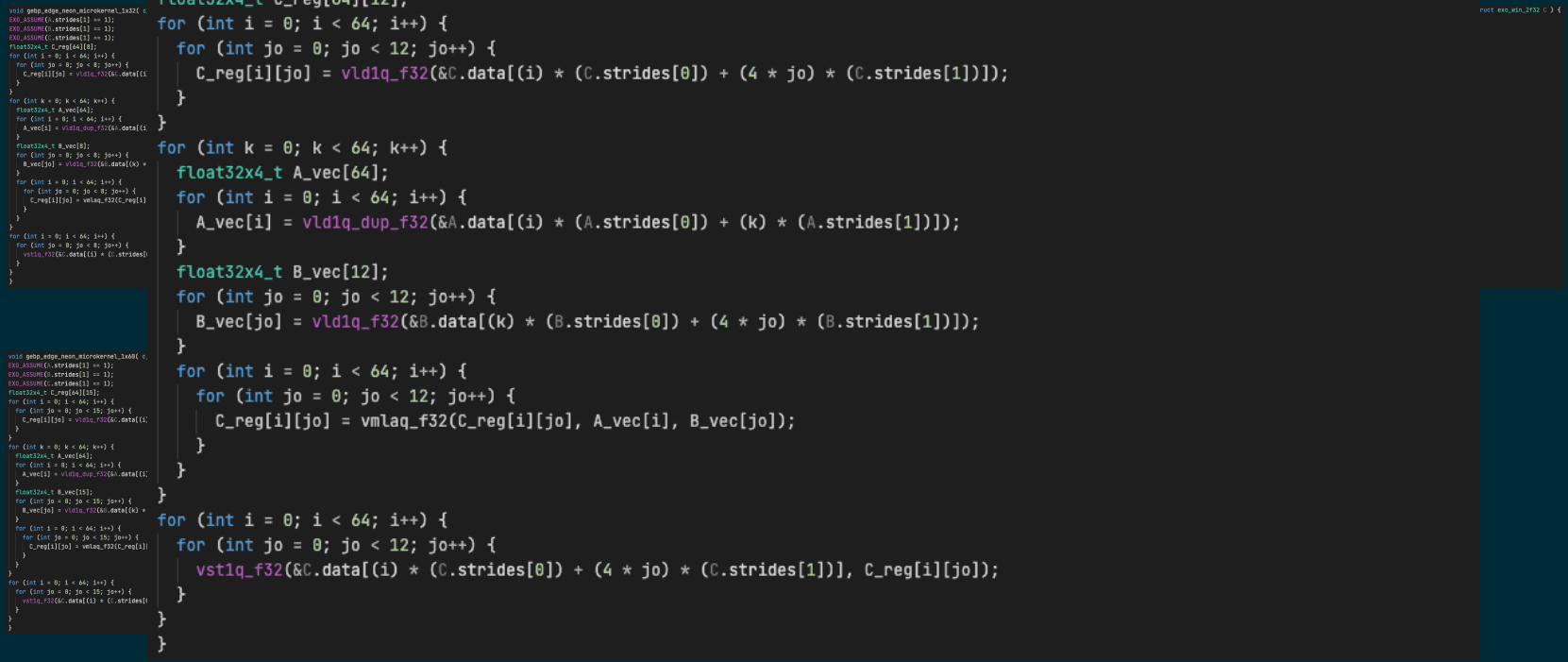
```
void gebp_edge_neon_microkernel_1x60( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][15];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 15; jo++) {
    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++) {
    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
  }
  float32x4_t B_vec[15];
  for (int jo = 0; jo < 15; jo++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 15; 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 = 0; jo < 15; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```



```
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 ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][13];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 13; jo++) {
    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++) {
    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
  }
  float32x4_t B_vec[13];
  for (int jo = 0; jo < 13; jo++) {
    B_vec[jo] = vld1q_f32(&B.data[(k) * (B.strides[0]) + (4 * jo) * (B.strides[1])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 13; 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 = 0; jo < 13; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

A more extreme example

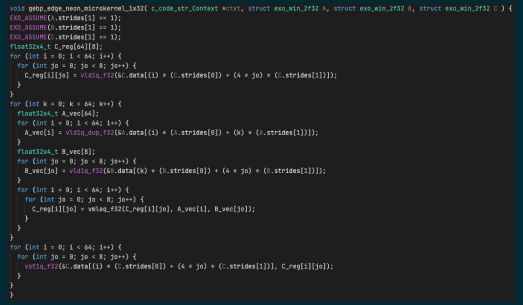
- I gene

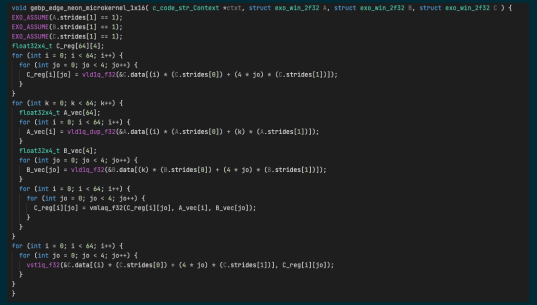


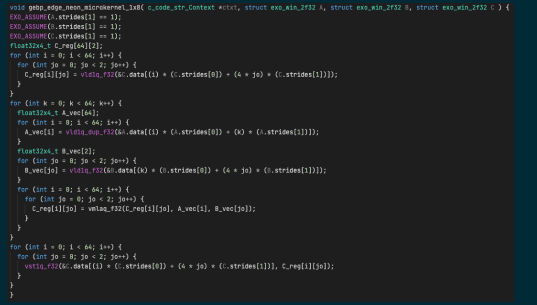
```
void gebp_edge_neon_microkernel_1x48( c_code_str_Context *ctxt, struct exo_win_2f32 A, struct exo_win_2f32 B, struct exo_win_2f32 C ) {
EXO_ASSUME(A.strides[1] == 1);
EXO_ASSUME(B.strides[1] == 1);
EXO_ASSUME(C.strides[1] == 1);
float32x4_t C_reg[64][12];
for (int i = 0; i < 64; i++) {
  for (int jo = 0; jo < 12; jo++) {
    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++) {
    A_vec[i] = vld1q_dup_f32(&A.data[(i) * (A.strides[0]) + (k) * (A.strides[1])]);
  }
  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])]);
  }
  for (int i = 0; i < 64; i++) {
    for (int jo = 0; jo < 12; 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 = 0; jo < 12; jo++) {
    vst1q_f32(&C.data[(i) * (C.strides[0]) + (4 * jo) * (C.strides[1])], C_reg[i][jo]);
  }
}
}
```

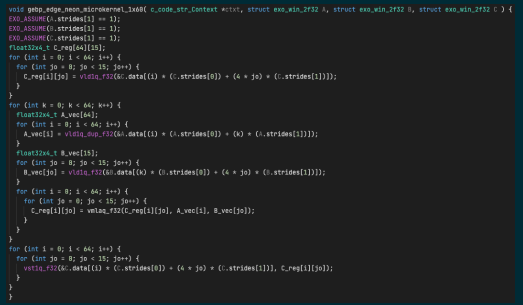
A more extreme example

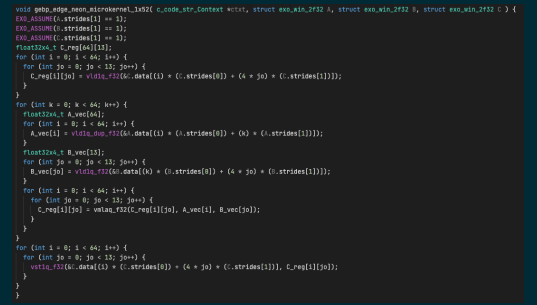
- I generated 64 microkernels using Exo

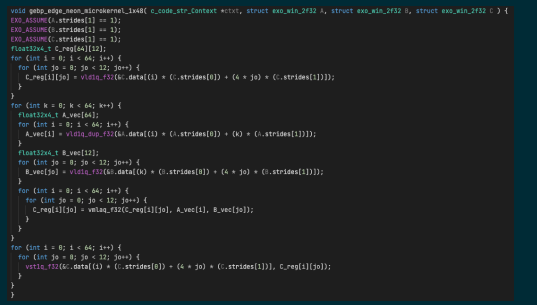












Implications

- 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