High-Performance Clifford Layers in C

TEAM 75

Simon Huber, Anirudhh Ramesh, Konstantinos Chasiotis, Mikkeline Elleby

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Clifford Algebra

- Operate on multivectors of scalars, vectors, planes, volumes, etc
- We work with basis vector that represent direction in space

- Uses geometric product with a main key rule:

$$e_i \times e_i = g_i$$

- This results in blades – a geometric object
- For a dimension d, we have 2^d blades which we define how they multiply with each other via a multiplication table. For 1d for instance it looks like this:

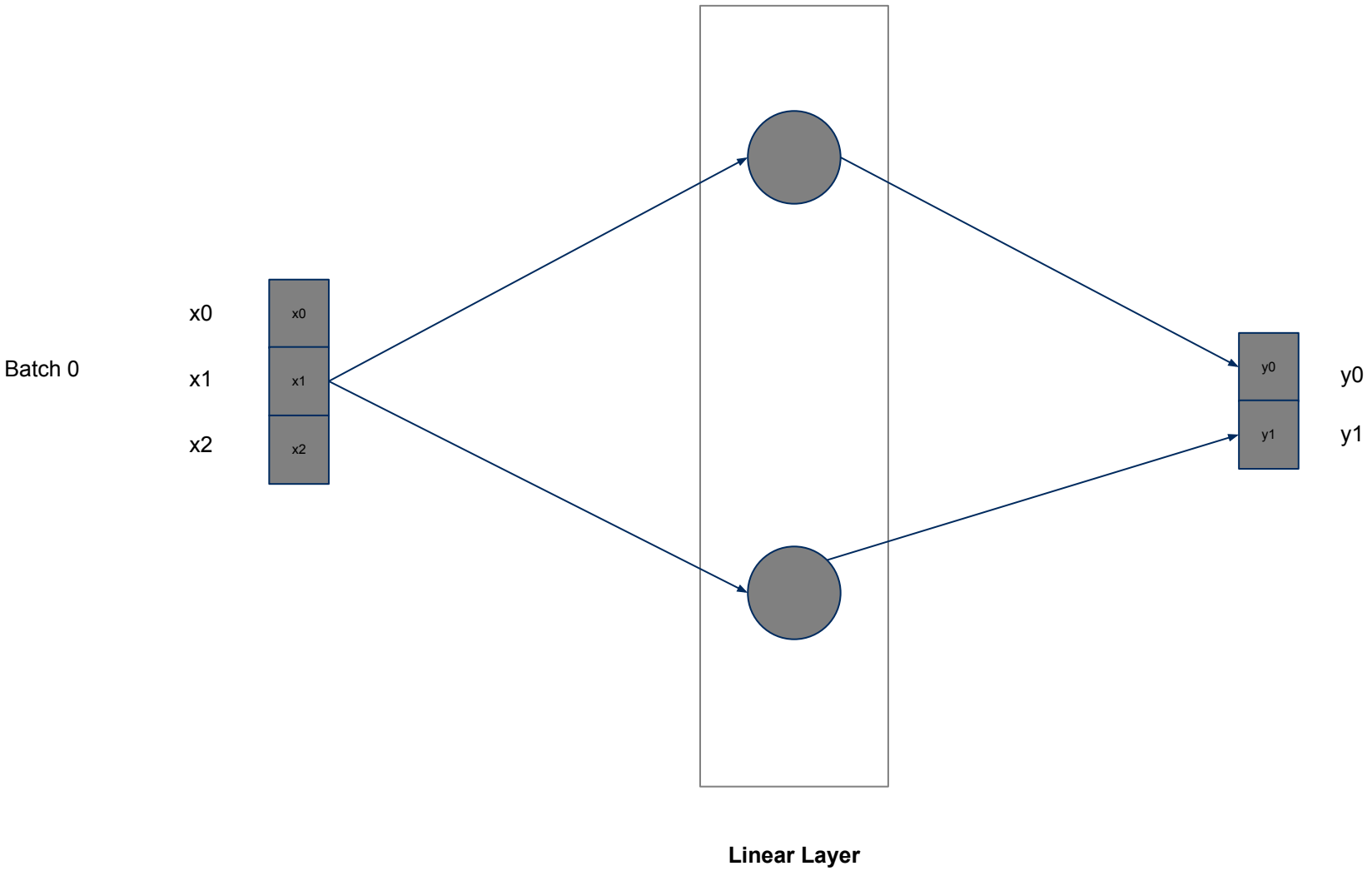
×	1	e_1
1	1	e_1
e_1	e_1	g_0

Reference Codebase

- Started with given codebase : GitHub - microsoft/cliffordlayers
- Created wrapper files for each c implementation to import the function into the original codebase
- Translated into c and focused on files for :
  - linear layer
  - multi-vector sigmoid linear units
  - group normalizations
- Extended the pytest for testing our implementation
  - Compared them with the python implementation

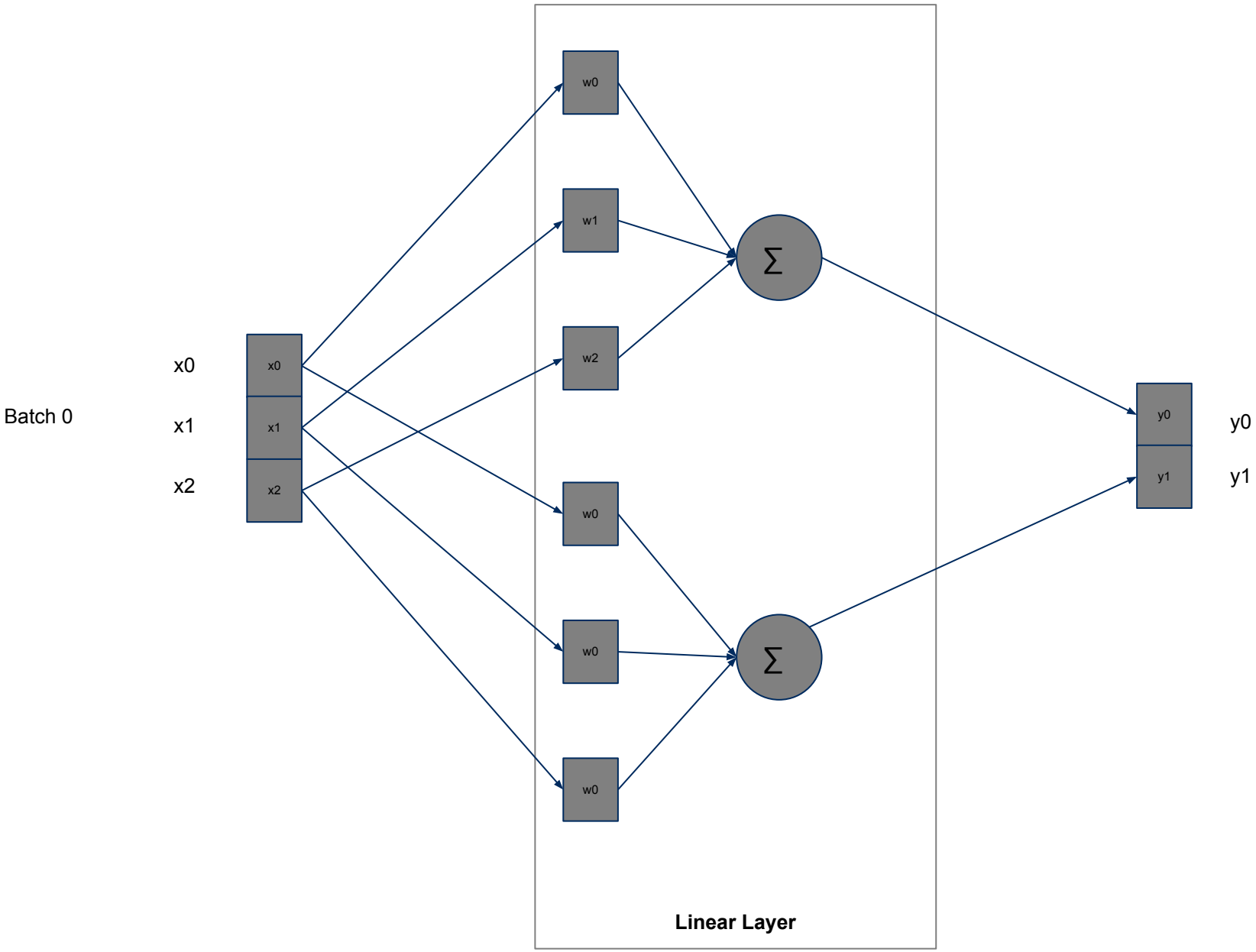
Layer: Linear Layer

Batch=1, channels_in = 3, channels_out = 2, dim=1



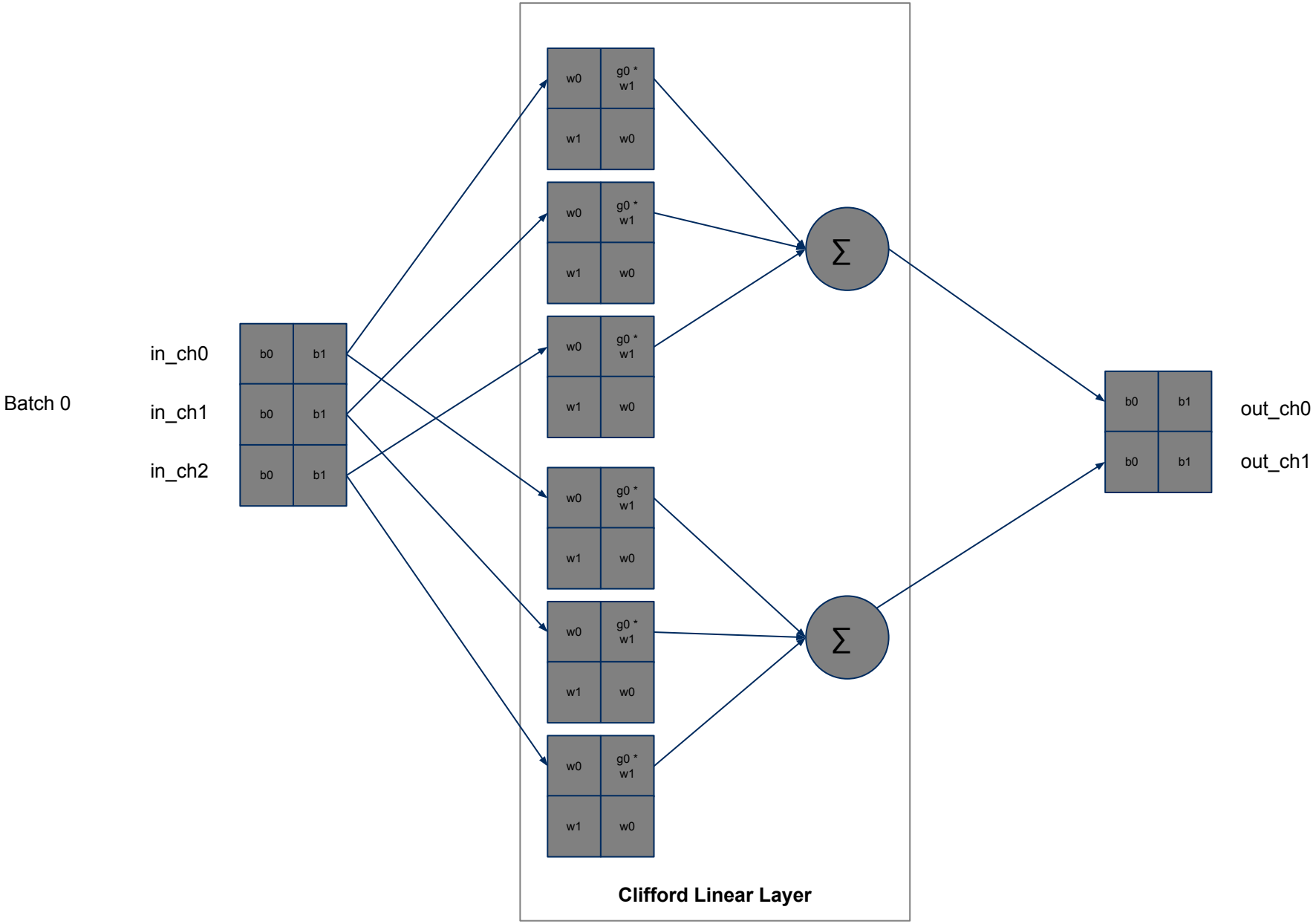
Layer: Linear Layer

Batch=1, channels_in = 3, channels_out = 2, dim=1



Layer: Clifford Linear Layer

Batch=1, channels_in = 3, channels_out = 2, dim=1

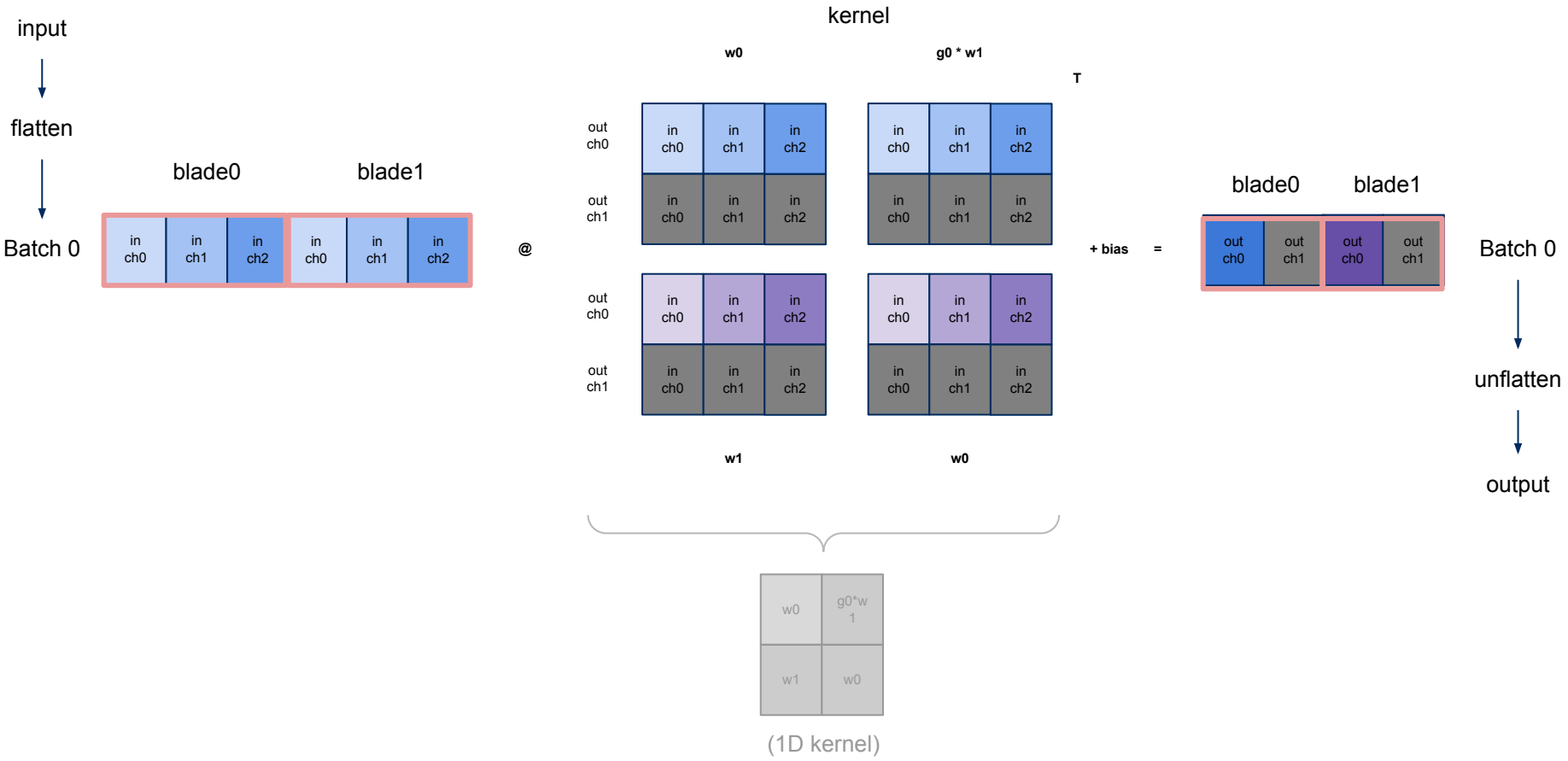


Batch=1, channels_in = 3, channels_out = 2, dim=1

Layer: Linear Layer

- **BASELINE**

Flattened input multiplied by kernels.



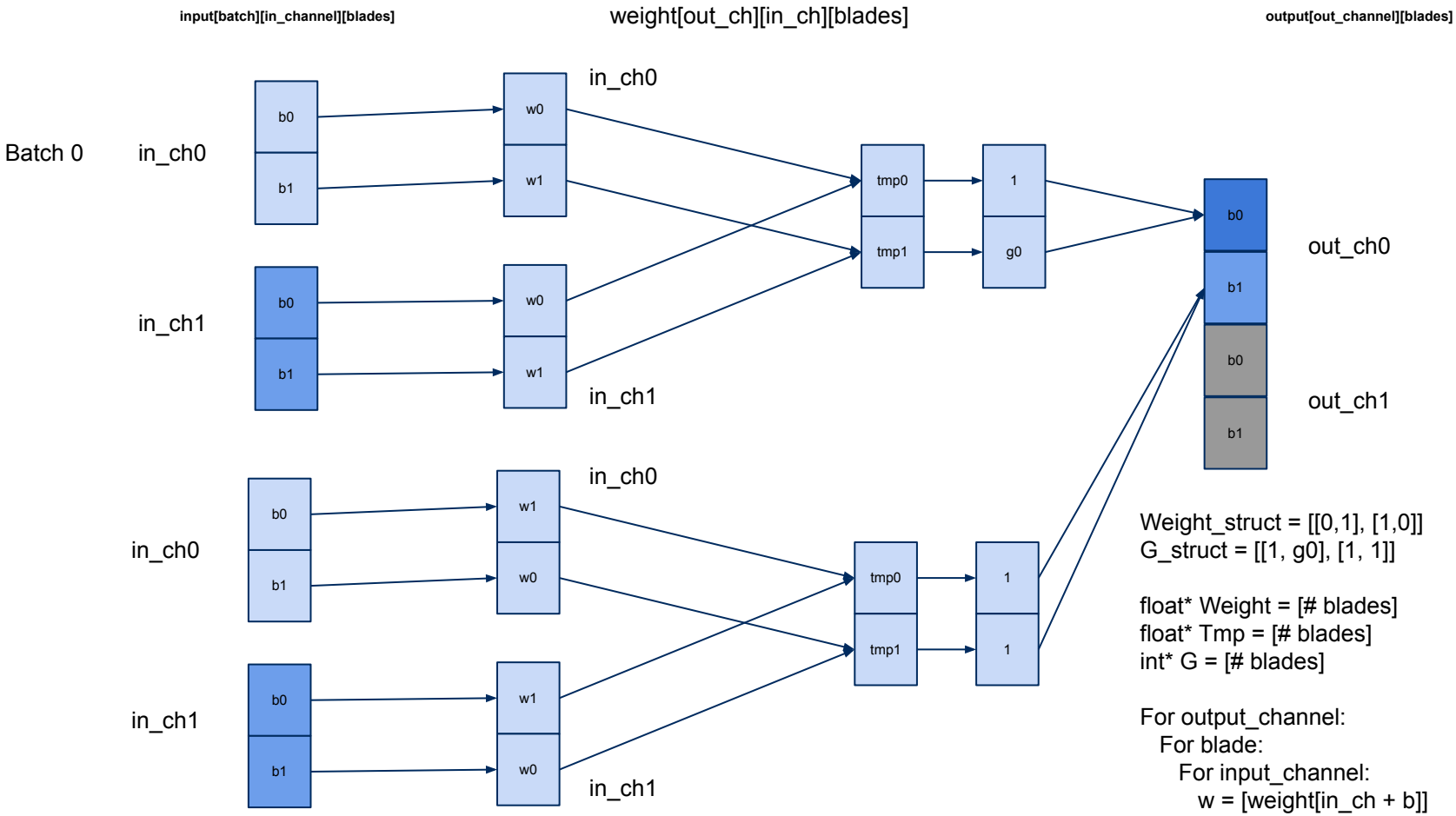
Improving

Re-use inputs
Weight[in_ch][out_ch][blades]
Have to 'random index' weights

Re-use outputs **<- probably better!**
Weight[out_ch][in_ch][blades]

In_ch -> contiguous mem accesses

Batch=1, channels_in = 3, channels_out = 2, dim=1



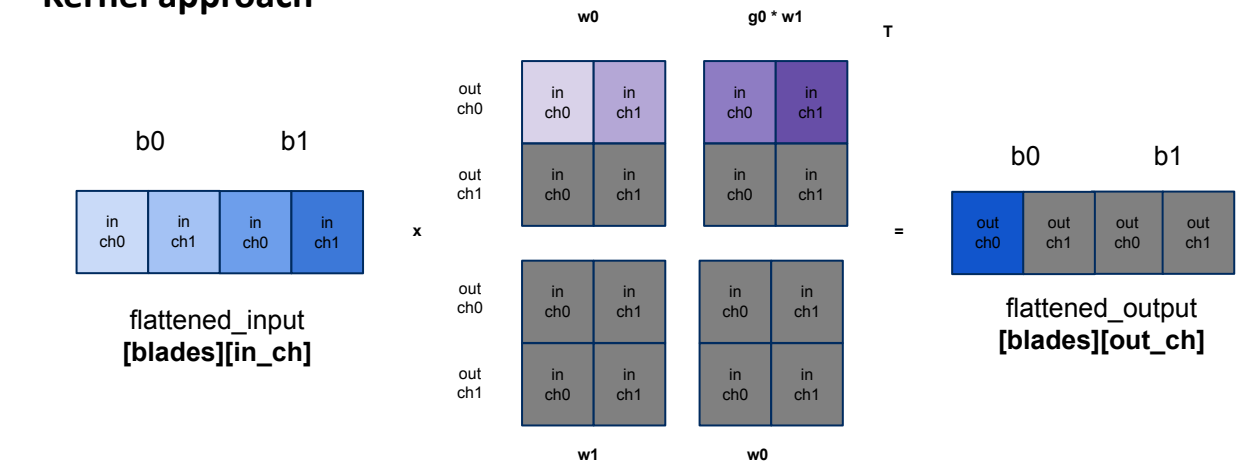
Weight_struct = [[0,1], [1,0]]
G_struct = [[1, g0], [1, 1]]

float* Weight = [# blades]
float* Tmp = [# blades]
int* G = [# blades]

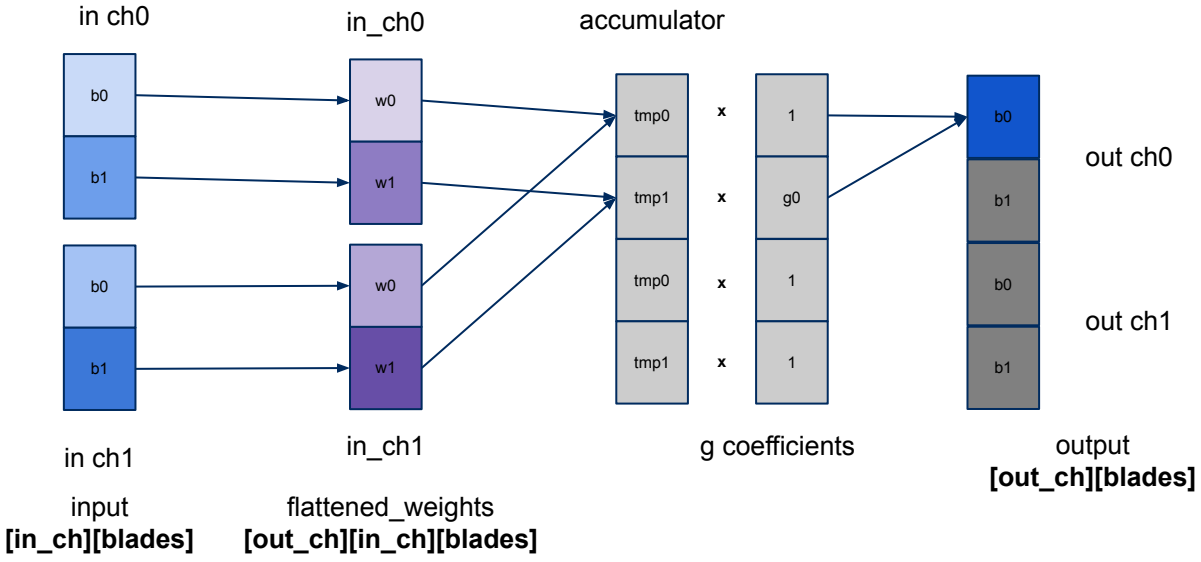
For output_channel:
  For blade:
    For input_channel:
      w = [weight[in_ch + b]]
      For blade:
        tmp[b] += in[b] * w[b]
  For blade:
    Tmp = Tmp[blade] * g[blade]
    Out[out + b] = sum_Reduce(tmp)

Problem with this? Strided weight accesses (switch between input channels)?
Blade inner performs very poorly?
When in_channels > cache line, we have cache striding => Worse than input re-ordering + non-strided accesses.

Kernel approach



No-Kernel approach



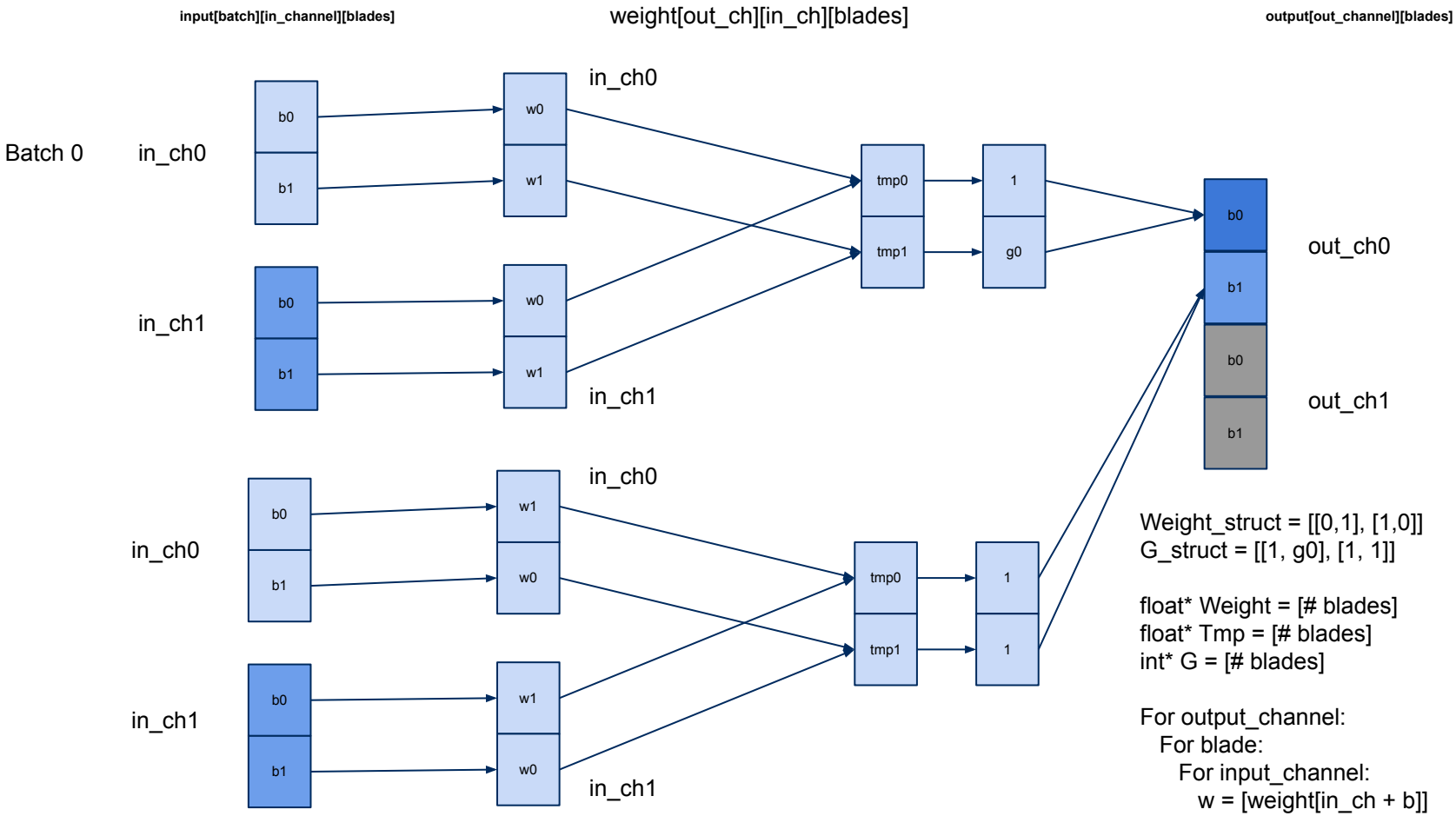
Improving

Re-use inputs
Weight[in_ch][out_ch][blades]
Have to 'random index' weights

Re-use outputs **<- probably better!**
Weight[out_ch][in_ch][blades]

In_ch -> contiguous mem accesses

Batch=1, channels_in = 3, channels_out = 2, dim=1



Weight_struct = [[0,1], [1,0]]
G_struct = [[1, g0], [1, 1]]

float* Weight = [# blades]
float* Tmp = [# blades]
int* G = [# blades]

For output_channel:
  For blade:
    For input_channel:
      w = [weight[in_ch + b]]
      For blade:
        tmp[b] += in[b] * w[b]
  For blade:
    Tmp = Tmp[blade] * g[blade]
    Out[out + b] = sum_Reduce(tmp)

Problem with this? Strided weight accesses (switch between input channels)?
Blade inner performs very poorly?
When in_channels > cache line, we have cache striding => Worse than input re-ordering + non-strided accesses.

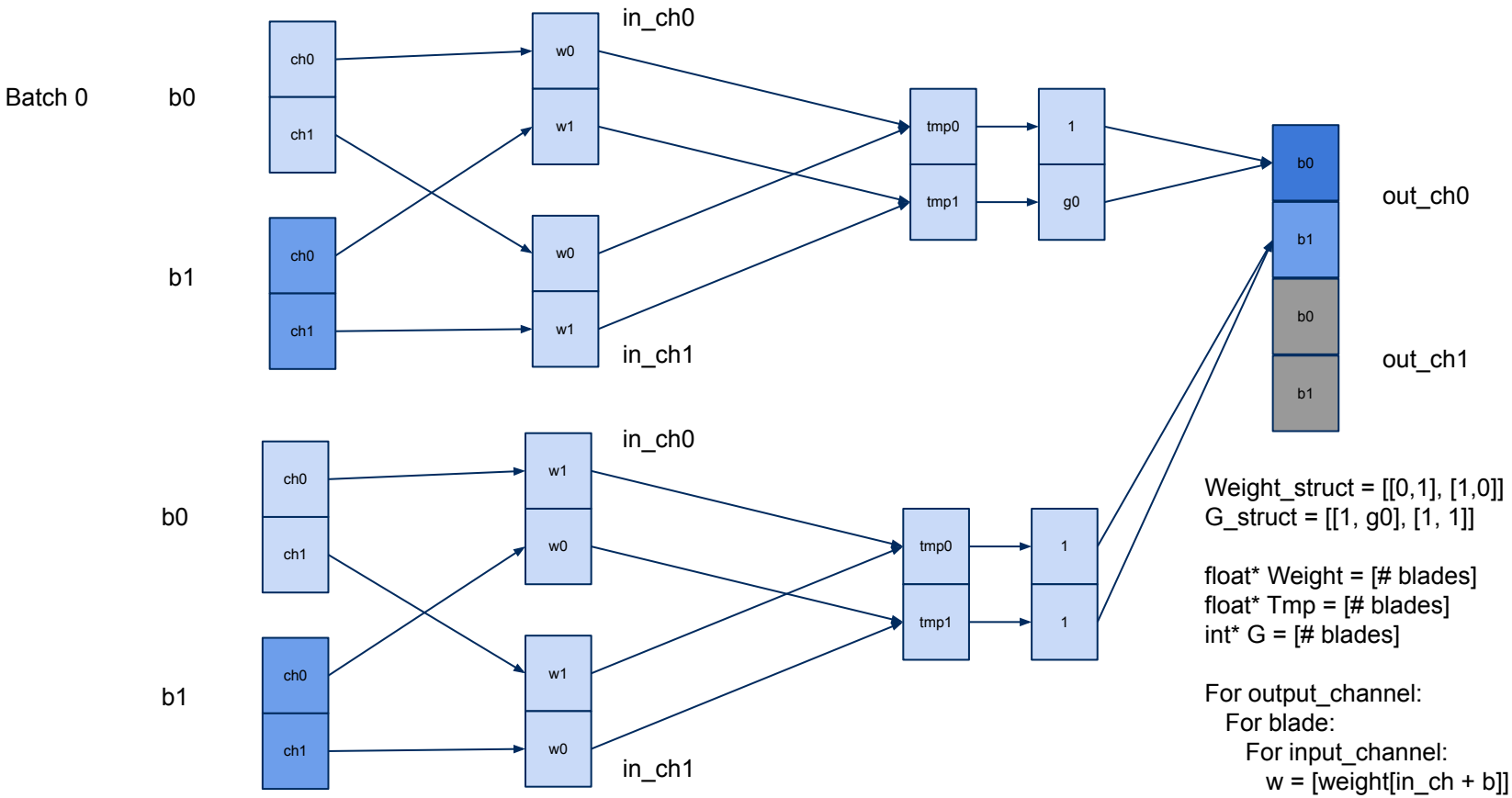
Improving

Batch=1, channels_in = 3, channels_out = 2, dim=1

So could multiply both together.
Apply AVX
Apply …
Clean up all the code
Roofline plots and stuff

weight[blades][out_ch][in_ch]

output[out_channel][blades]



Weight_struct = [[0,1], [1,0]]
G_struct = [[1, g0], [1, 1]]

float* Weight = [# blades]
float* Tmp = [# blades]
int* G = [# blades]

For output_channel:
  For blade:
    For input_channel:
      w = [weight[in_ch + b]]
      For blade:
        tmp[b] += in[b] * w[b]
    For blade:
      Tmp = Tmp[blade] * g[blade]
      Out[out + b] = sum_Reduce(tmp)

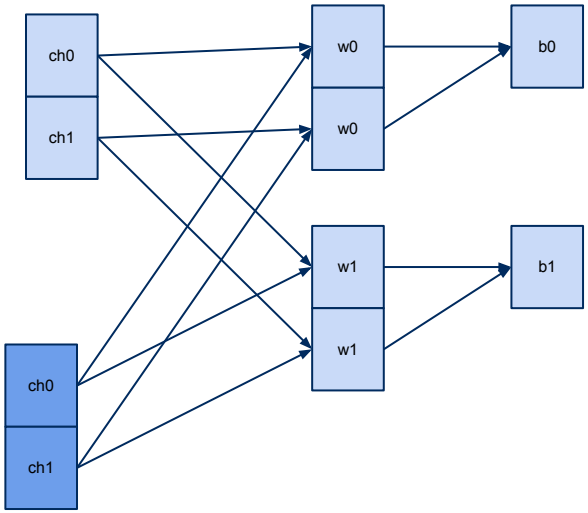
Multiply and store the output variable per input channel based on index!
AVX vectorization of this approach
Should scale without strided access issues.
Any interesting AVX optimizations? Channel-dependent however

Baseline

Different input weight per channel
input_f[batch][blades][input]

Different input weight per channel
weights[blades][output][input]

Different input weight per channel
output_f[batch][blades][output]

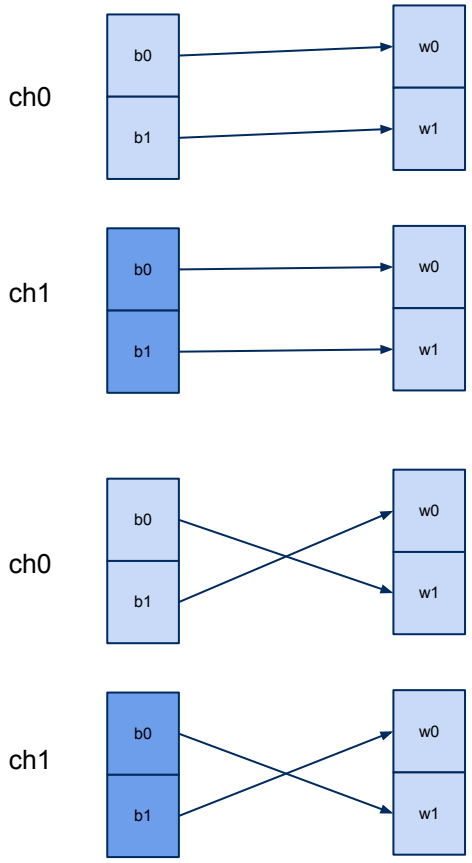


We'd get striding on the weight matrix (if inputs >> large)

No kernel w/o flattened

Different input weight per channel
input_f[batch][input][blades]

Different input weight per channel
weights_f[output][input][blade]



We'd get striding as we multiply each input. Ideal for input x
input[batch][input][blades]

- **Optimization:**

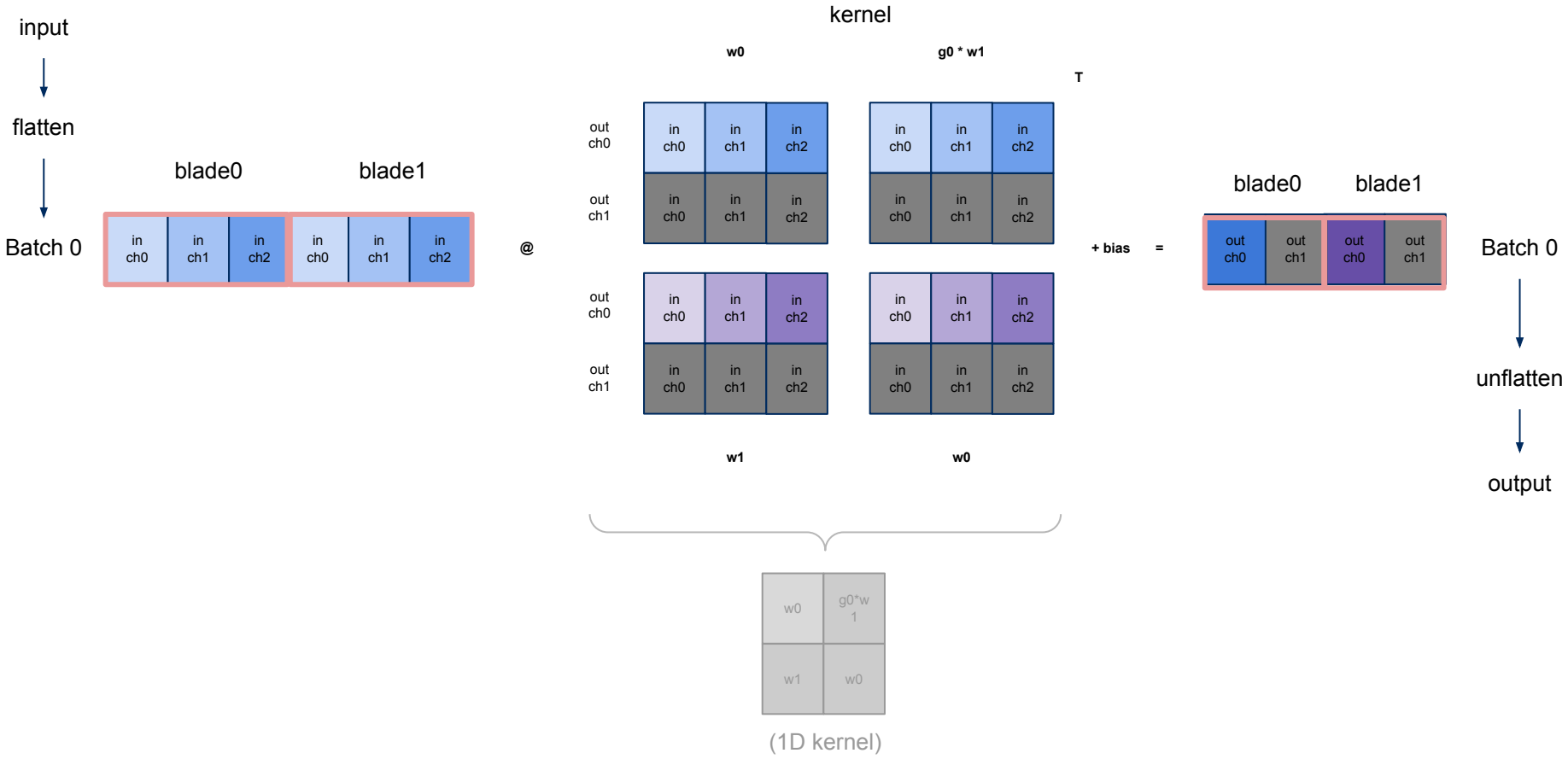
Cache the kernel (avoid reconstruction)

During kernel construction, pre-compute various multiple computations

AVX-2 vectorizations of the kernel multiplication, vector k-accumulators, FMA

=> ~3x overall scalar speed-up

=> ~8x overall vectorized speed-up



- **Optimization:**

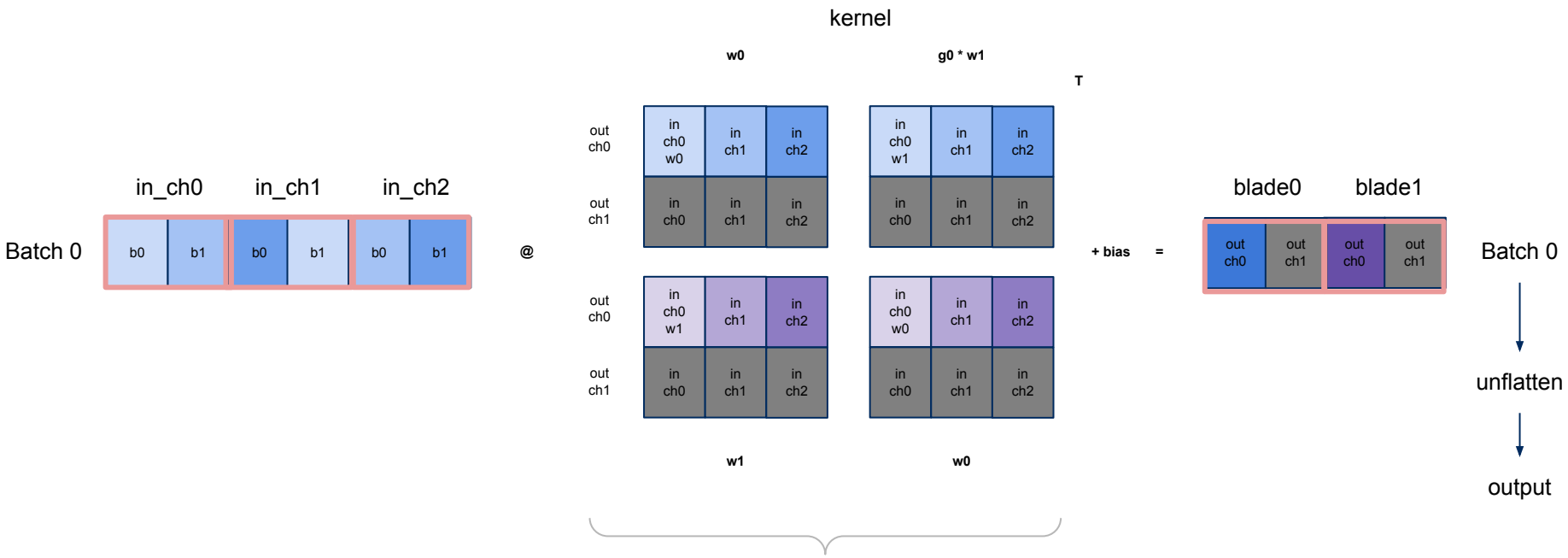
Cache the kernel (avoid reconstruction)

During kernel construction, pre-compute various multiple computations

AVX-2 vectorizations of the kernel multiplication, vector k-accumulators, FMA

=> ~3x overall scalar speed-up

=> ~8x overall vectorized speed-up



Can we remove the flattening?

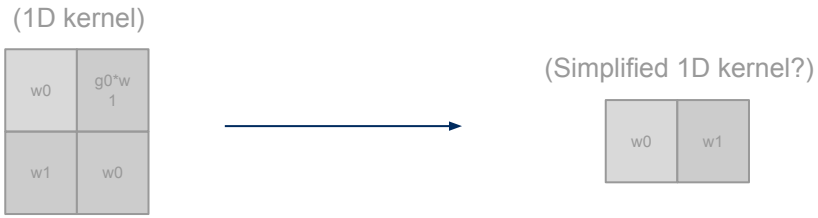
Without removing flattening?

Layer: Linear Layer

- **Further Optimization ideas (in progress)**

Currently, the kernels look as follows:

1D:
[w0 g0*w1]
[w1 w0]

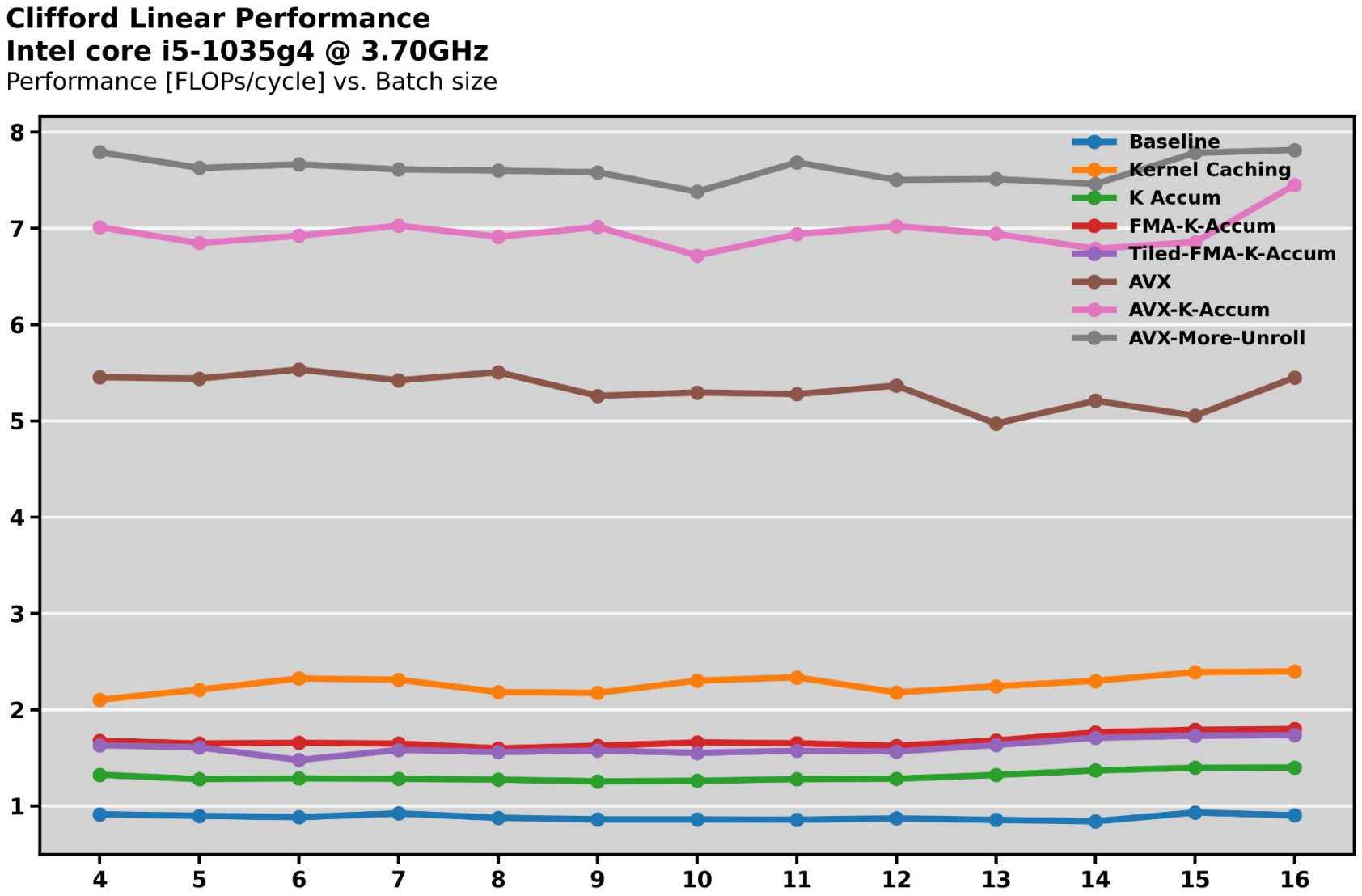


2d:
… (similar pattern)

3d:
… (similar pattern)

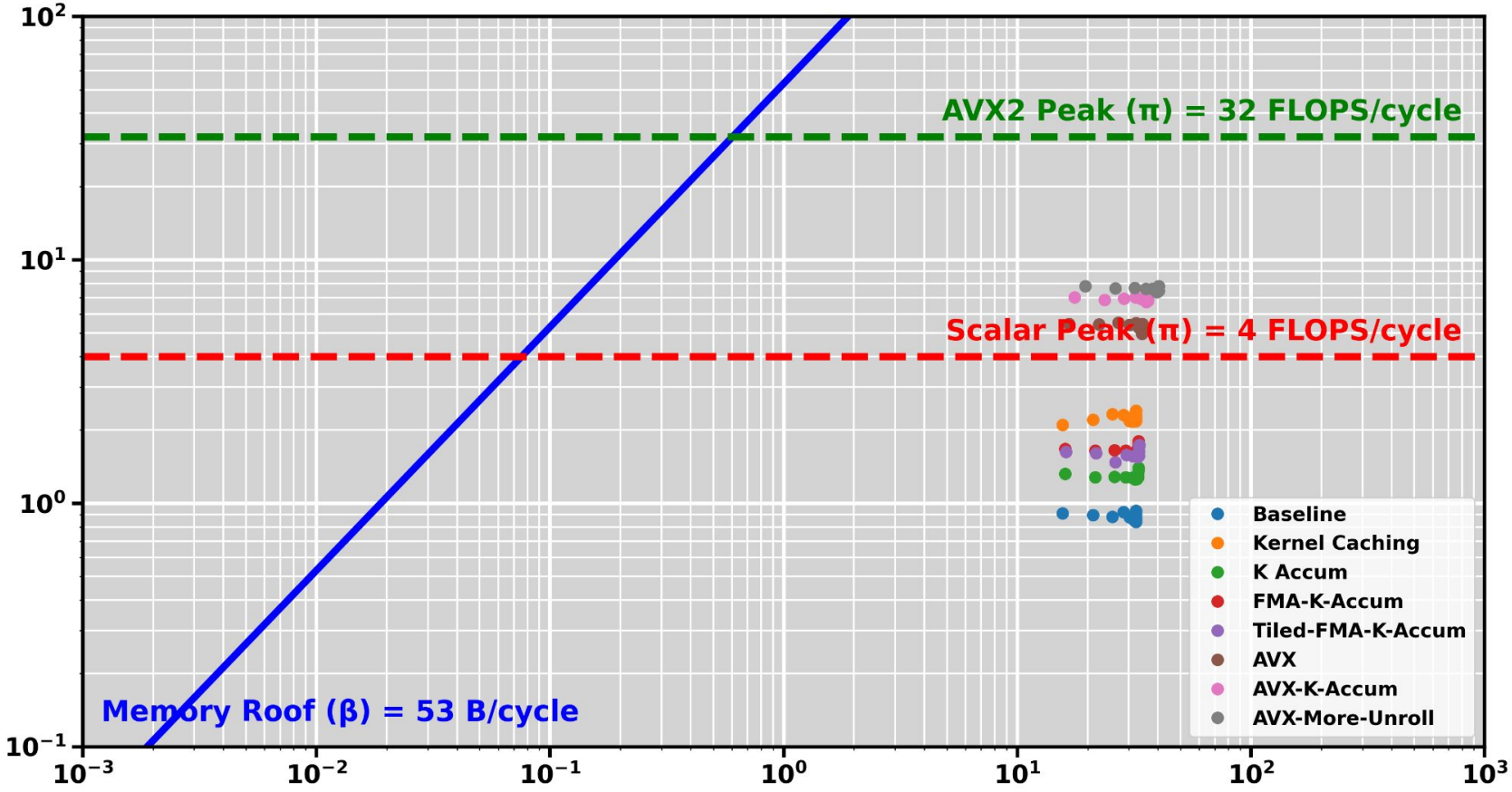
Could we halve the size of the kernels (e.g. k_opt = [w0 g0*w1])?
Then, in the multiplications we can do e.g. x @ k_opt and k_opt @ x to construct the final output.
Another idea: Construct the kernels in a way that allows no flattening?
=> Results in a smaller kernel (i.e. less memory load cycles)

Layer: Linear Layer (Performance)



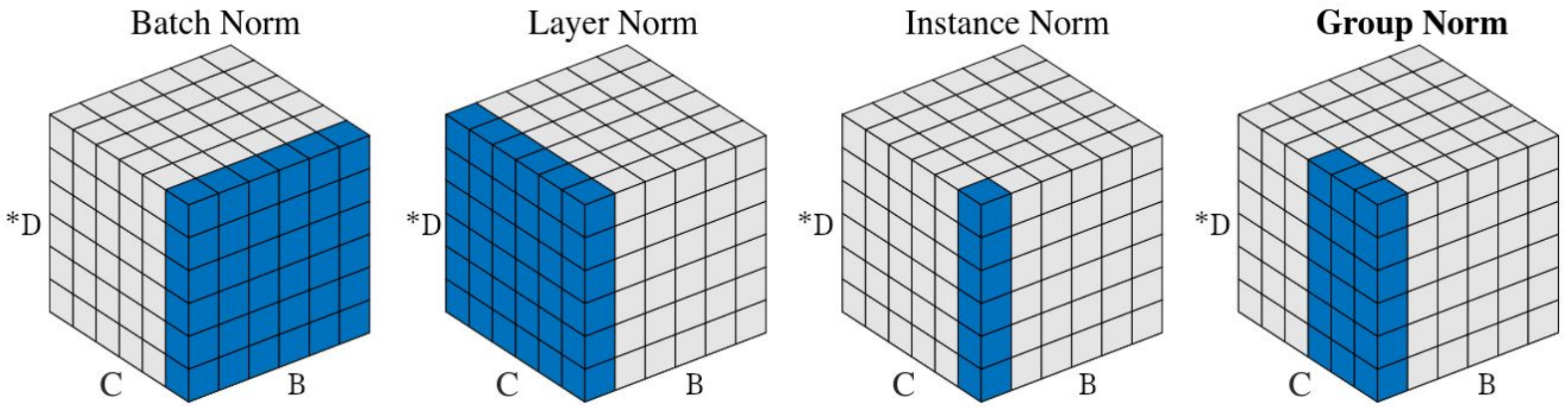
Layer: Linear Layer (Roofline)

Intel Core i5-1035G4 @ 3.70GHz
Compiler: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Performance (F/C) vs Operational Intensity (F/B)



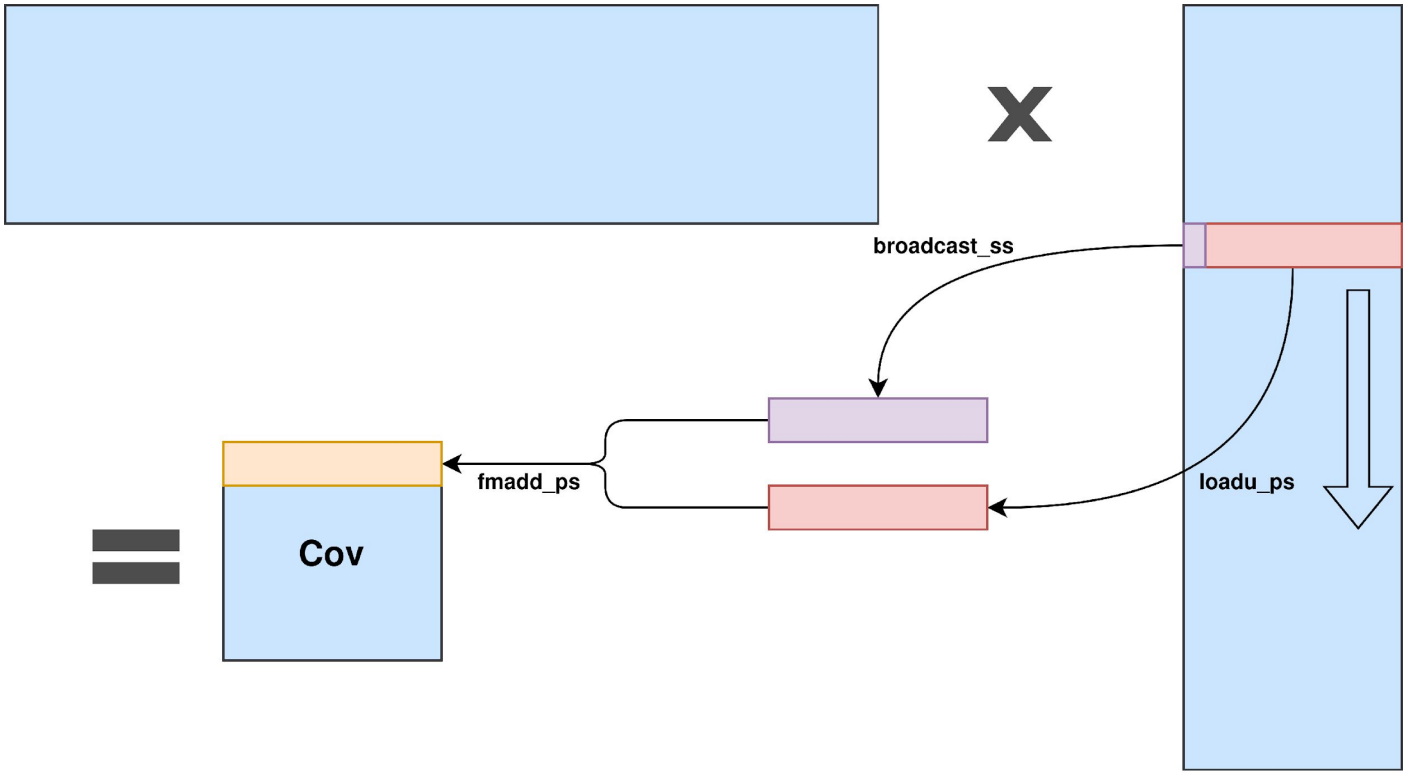
Layer: Group normalizations

- Normalizes across a group of channels
- Involves:
  - Calculating and subtracting means
  - Calculating covariance matrices
  - Finding maximums
  - Calculating Cholesky decompositions
  - Back-substitutions



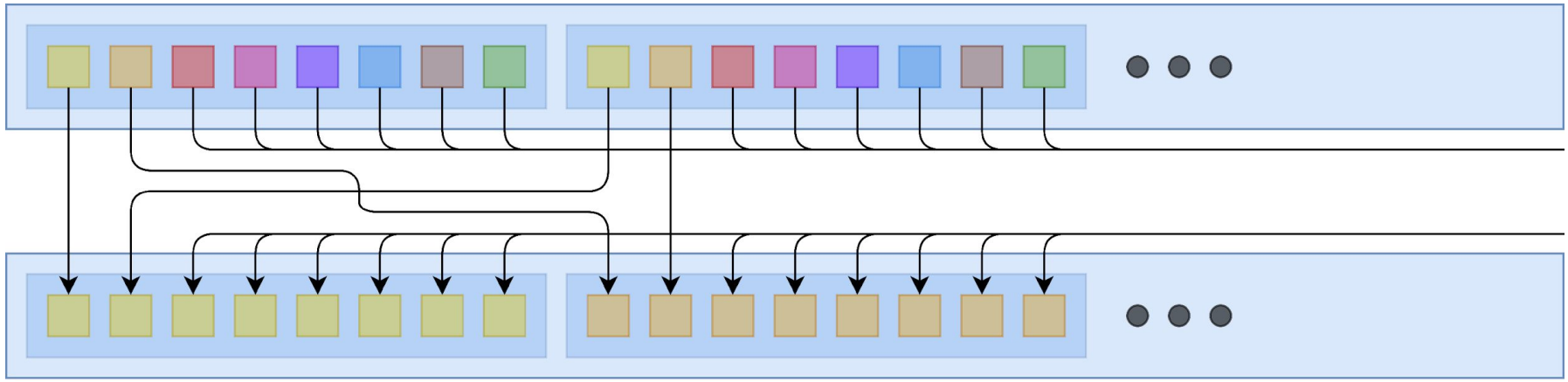
Groupnorm Covariance Optimization

- **BASELINE: Transpose into temporary array and do standard matrix multiply**
- **OPTIMIZATION: Vectorized direct calculation with AVX fmadd_ps**

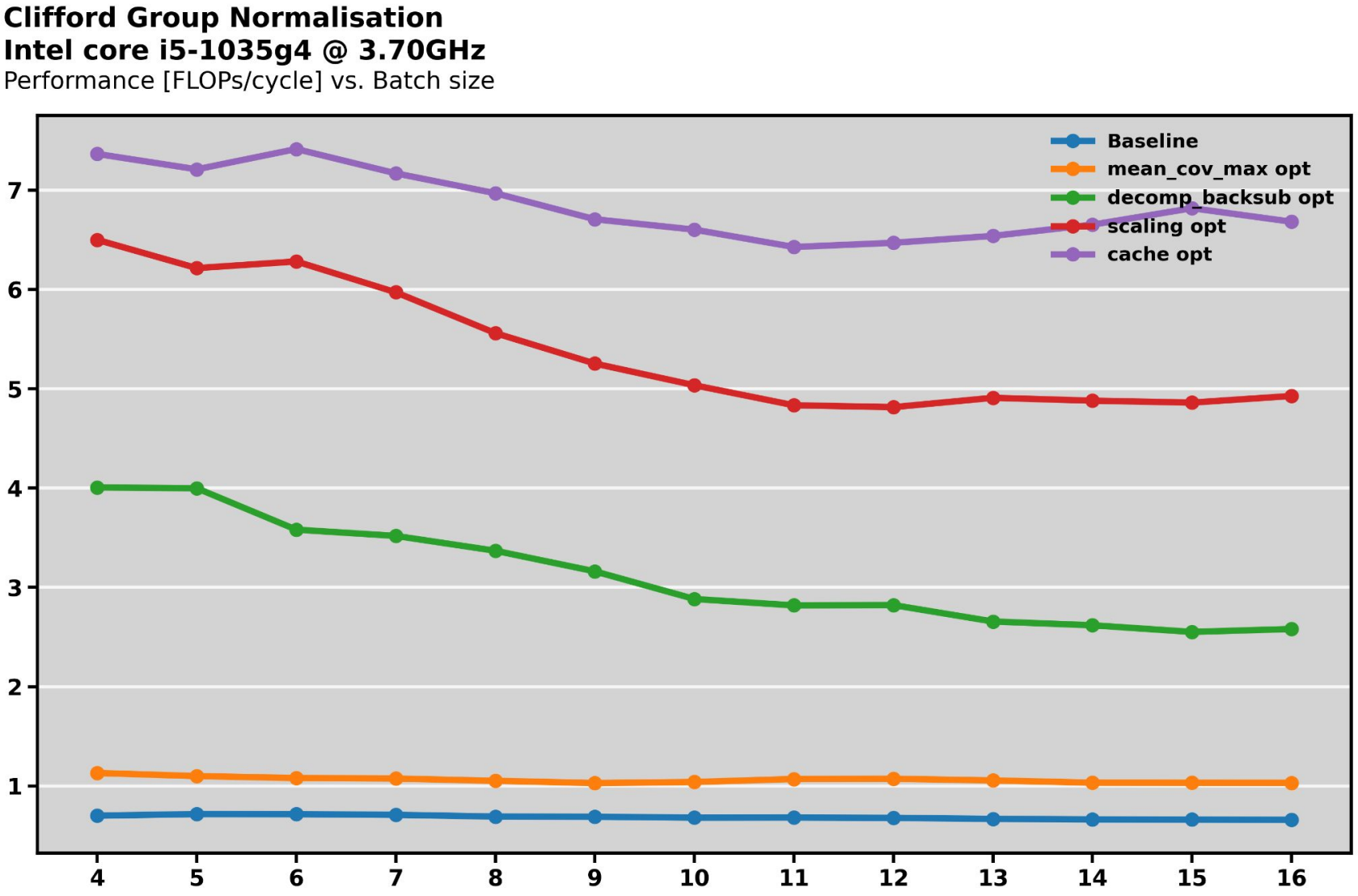


Groupnorm Backsubstitution Optimization

- Dependencies across iterations => hard to optimize single Back substitution
- BASELINE: One matrix and one vector at once
- OPTIMIZATION: One matrix and 8 vectors at once (AVX)
  - Needs memory layout restructuring

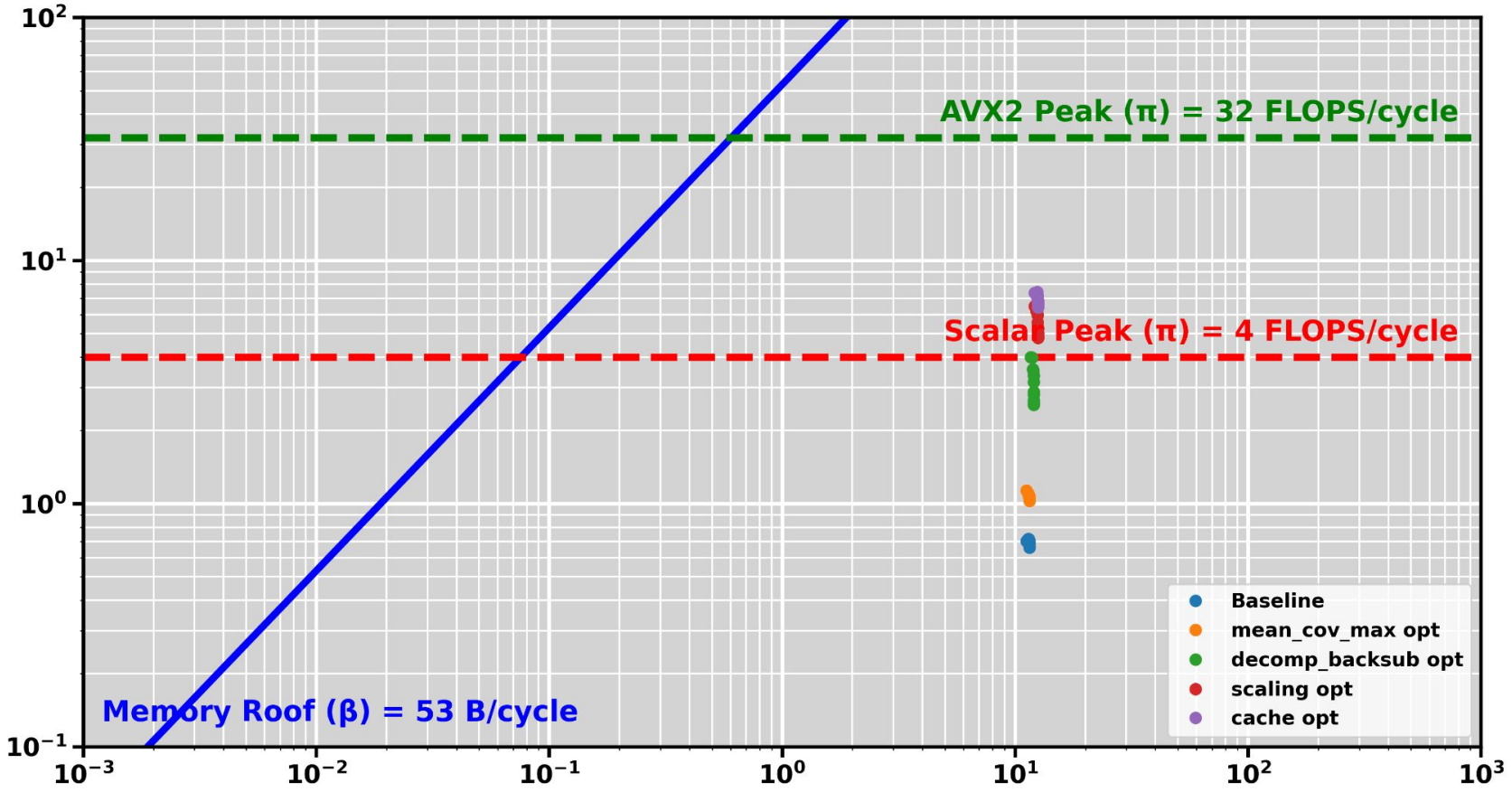


Groupnorm Performance Plots



Groupnorm Roofline

Intel Core i5-1035G4 @ 3.70GHz
Compiler: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Performance (F/C) vs Operational Intensity (F/B)



Multi-vector sigmoid linear units

1. **Pre-activation** z**:**

$$z = b_c + \sum_{j=0}^{I-1} W_{c,j} X_{b,c,p,j}$$

2. **Non-linearity** *gate*:

$$\text{gate} = \sigma(z) = \frac{1}{1+e^{-z}}$$

3. **Output** Y**:**

$$Y_{b,c,p,i} = X_{b,c,p,i} \times \text{gate}$$

- **Depth-wise 1×1 "convolution" → sigmoid → gate each blade component**

Machine perception notes helped us solve a bug ;)

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

convolution over an input signal composed of several input planes.

output value of the layer with input size (N, C_{in}, H, W) and output) can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

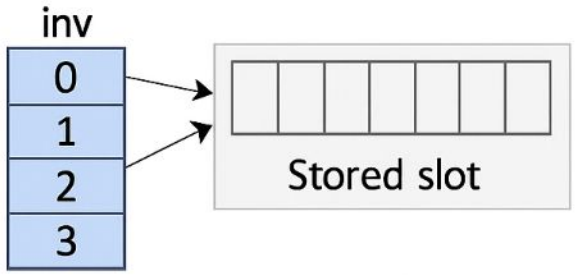
Most DL frameworks implement a correlation (and call it convolution). Annoying, but …

Indeed ☺

It doesn't matter **for learning**. In DL we **learn the kernel weights**, we don't care whether we learn a flipped kernel or not.

MV: Inverse mapping

- **Full blade space → Stored slots**
  - **We start with a large space of possible blade indices (0…all_blades–1), but only store a small subset (slots) per channel**



- **Build tiny lookup table:**

```
for (int b = 0; b < all_blades; ++b) inv[b] = -1;
for (int i = 0; i < slots; ++i) inv[input_blades[i]] = i;
```

Go from O(batch·channels·all_blade_space)

→ to O(all_blades) memory !

Note : all_blades = size of the full blade index range

MV: Fast Sigmoid via Lookup

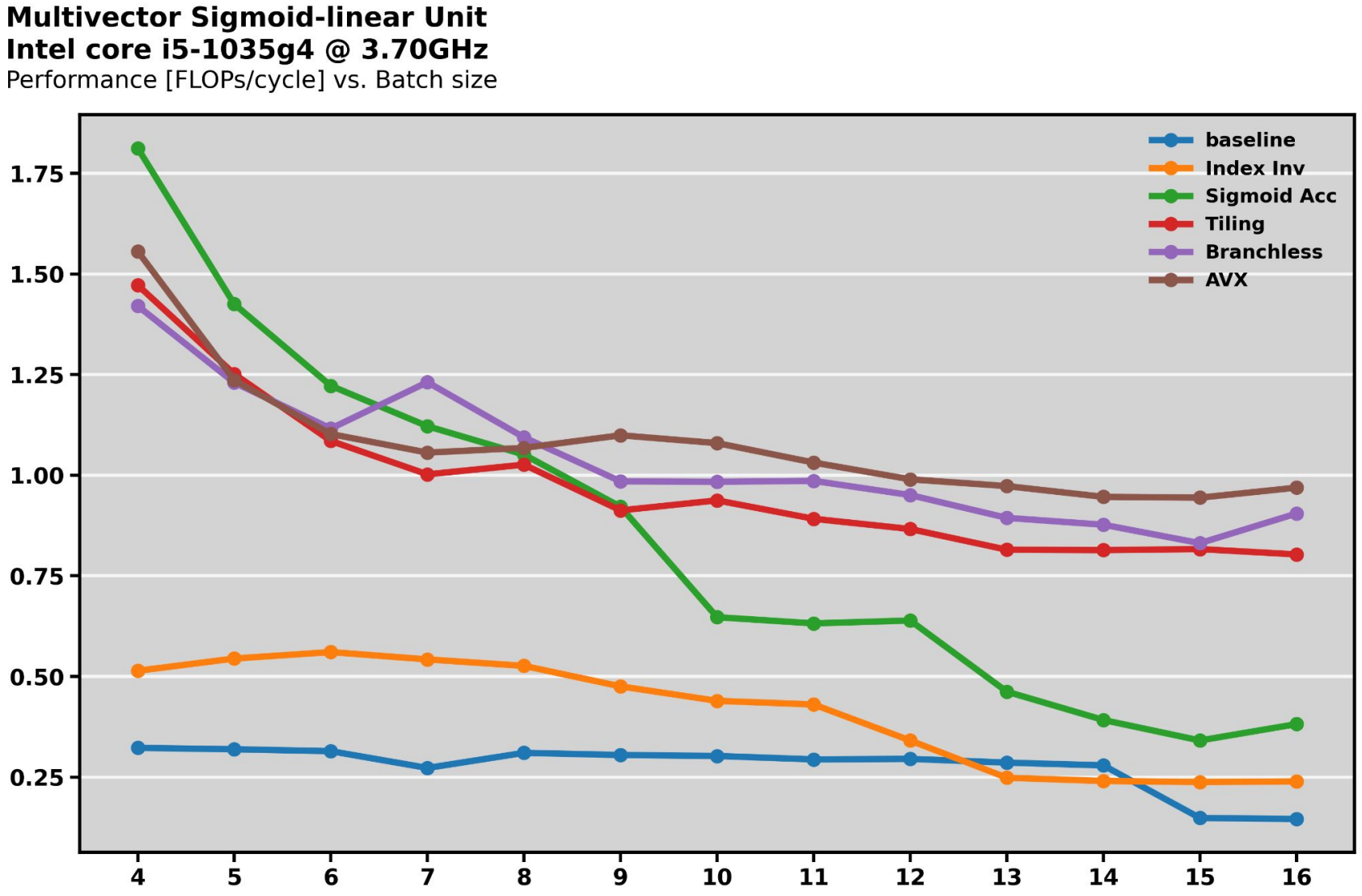
- Per-element 1/(1+expf(–z)) costs tens of cycles each
- Dominates our inner loop SO EVEN MORE EXPENSIVE !

  - Idea: Sample once by doing the computation
  - Lookup + interpolate:
    - *Clamp z to [−6,6]*

```
for (int i = 0; i < 512; ++i) {
  float x = -6.0f + 12.0f * i / 511.0f;
  sigmoid_table[i] = 1.0f / (1.0f + expf(-x));
}
```

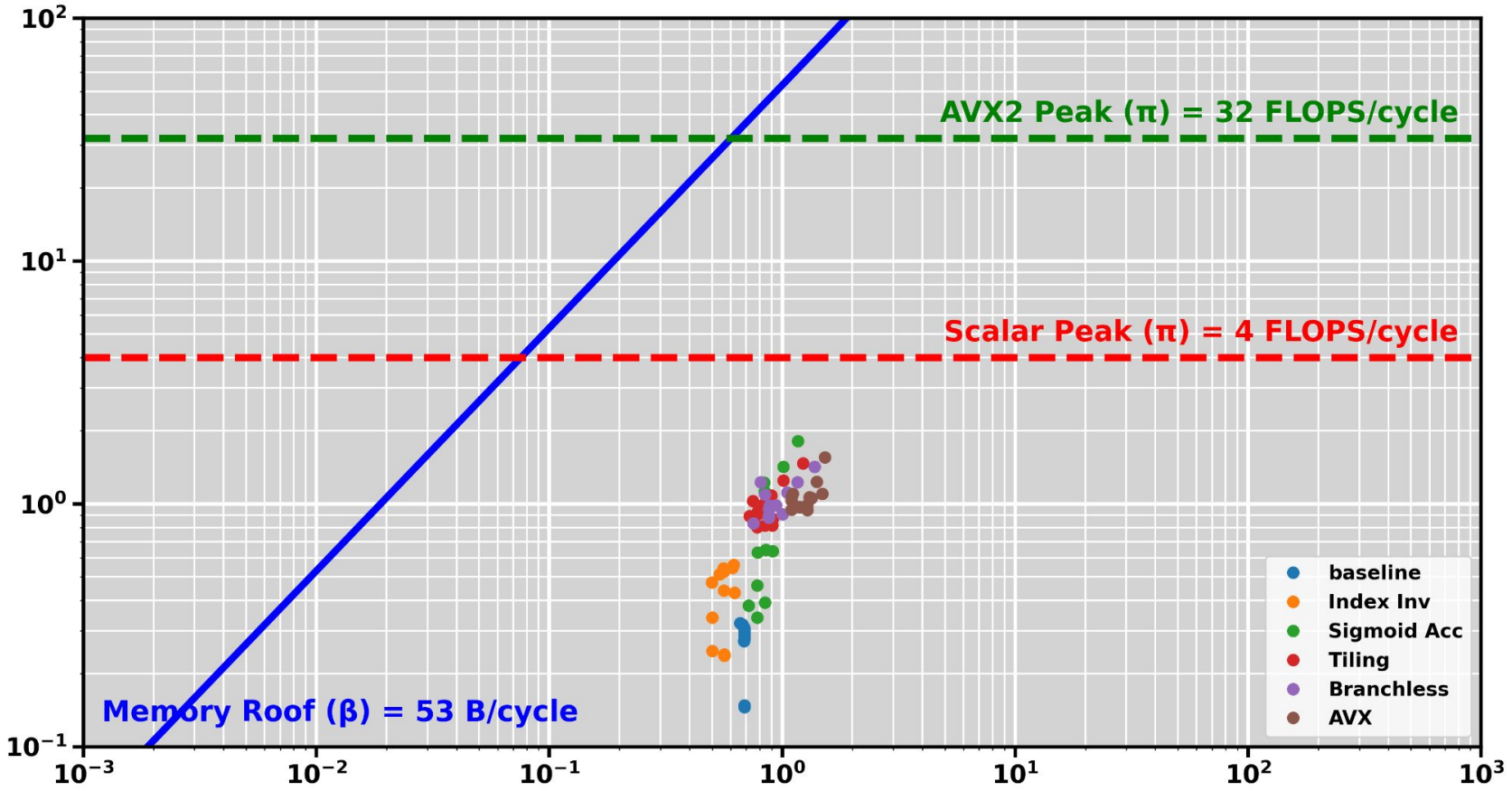
- No expf() inside the hot loop—just two loads + three ops
- Still get ~1e-5 accuracy

Layer: Activation (Performance)



Layer: Activation (Roofline)

Intel Core i5-1035G4 @ 3.70GHz
Compiler: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Performance (F/C) vs Operational Intensity (F/B)



Thank you for listening !