# SIMD Advantage Profiling

October 02, 2025

**Joyal Mathew**

## System Information

**CPU**
AMD Ryzen 5 5625U

**Clock Speed**
4.34 GHz

**Cache Sizes**

| | |
|---|---|
| **L1** | 192 KiB |
| **L2** | 3 MiB |
| **L3** | 16 MiB |

**Vector Instructions**

AVX2 (256 bit)

**Operating System**
Ubuntu 22.04.5 LTS

## Experiments

The following kernels were tested:

$$\begin{aligned} &\text{Fused Multiply Add (fma):} && y_i = a x_i + y_i \\ &\text{Dot Product (dot):} && s = \sum_i x_i y_i \\ &\text{1D Convolution (conv1d):} && y_i = a x_{i-1} + b x_i + c x_{i+1} \end{aligned}$$

Error-bars depict the 95% confidence interval for 10 runs.

Processes were run using the following to maintain consistency.

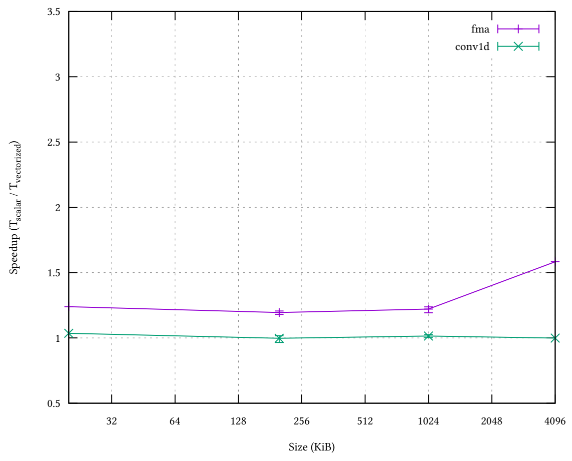
```
taskset -c 0 chrt -f 99 ..
```

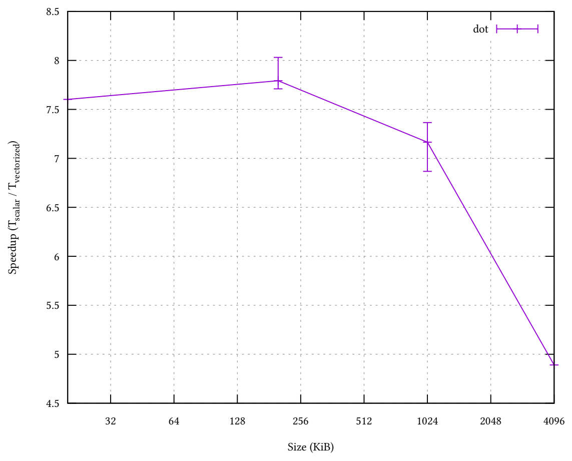
Run on logical core 0 with maximum real-time priority (FIFO scheduler).

CPU governor was set to “performance”.

### Scalar vs. Auto-Vectorized

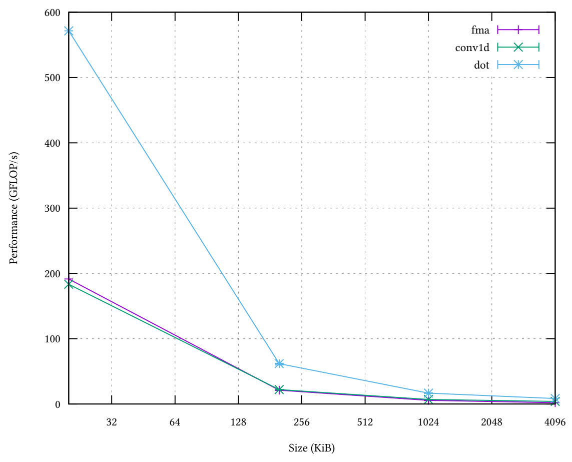
The achieved speedup of the auto-vectorized code over the scalar code for different kernel sizes is shown for each function tested. the dot kernel is depicted on a separate graph for readability.

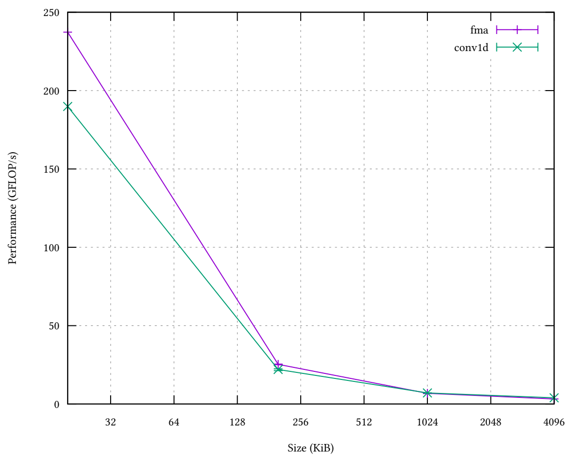


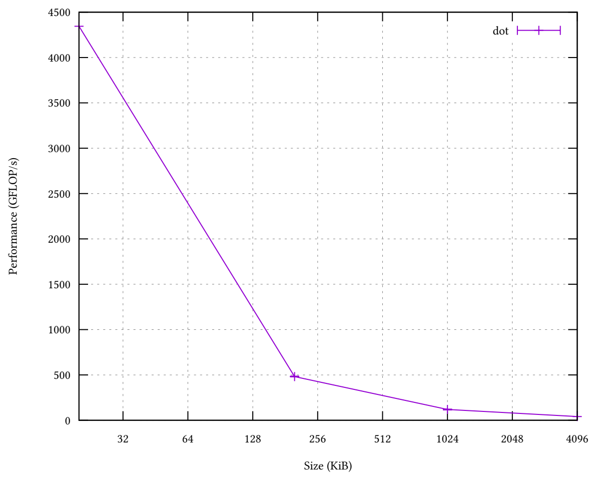


We can see that the dot kernel achieves a much larger speedup than the other two. This can be explained by the dot kernel being compute bound (reduce operation) while the other kernels are memory-bound (element-wise operation).

The follow plots and tables show scalar and vector speedup/performance for each kernel.







| Size (KiB) | 20 | 200 | 1024 | 4096 |
|---|---|---|---|---|
| GFLOP/s | 571.53 | 61.82 | 16.66 | 8.39 |

Table 1: scalar dot kernel performance

| Size (KiB) | 20 | 200 | 1024 | 4096 |
|---|---|---|---|---|
| GFLOP/s | 4344.76 | 481.76 | 119.34 | 41.03 |
| Speedup | 7.60 | 7.79 | 7.16 | 4.89 |

Table 2: vector dot kernel performance

| Size (KiB) | 20 | 200 | 1024 | 4096 |
|---|---|---|---|---|
| GFLOP/s | 191.48 | 21.19 | 5.58 | 2.03 |

Table 3: scalar fma kernel performance

| Size (KiB) | 20 | 200 | 1024 | 4096 |
|---|---|---|---|---|
| GFLOP/s | 237.28 | 25.31 | 6.81 | 3.21 |
| Speedup | 1.23 | 1.19 | 1.22 | 1.58 |

Table 4: vector fma kernel performance

| Size (KiB) | 20 | 200 | 1024 | 4096 |
|---|---|---|---|---|
| GFLOP/s | 183.38 | 22.08 | 6.95 | 3.96 |

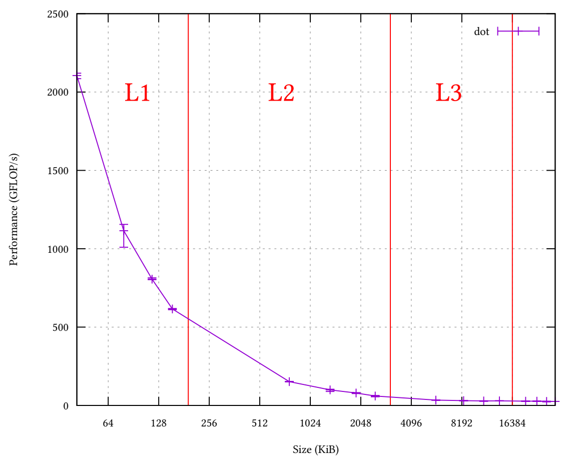
Table 5: scalar conv1d kernel performance

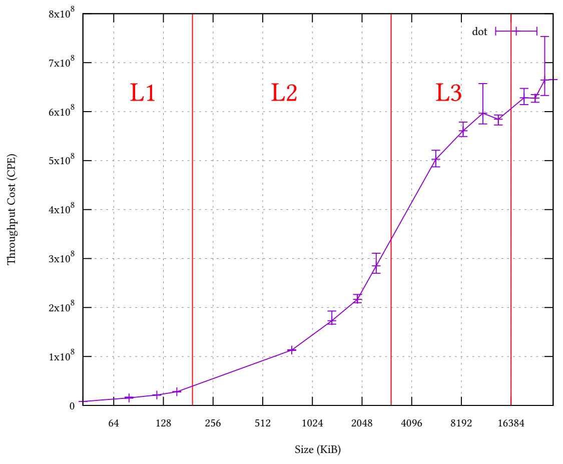
| Size (KiB) | 20 | 200 | 1024 | 4096 |
|---|---|---|---|---|
| GFLOP/s | 189.86 | 22.03 | 7.05 | 3.95 |
| Speedup | 1.03 | 0.99 | 1.01 | 0.99 |

Table 6: vector conv1d kernel performance

## Locality Sweep

Here we sweep the kernel size regime from L1 cache up to main memory. Results are for the dot kernel.

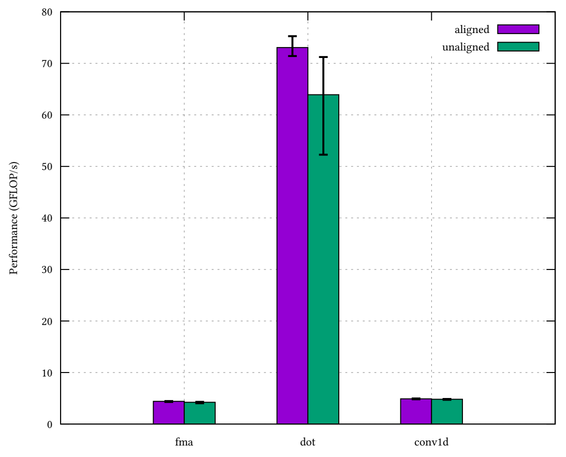




As expected, performance drops off quickly as the kernel crosses memory level boundaries. As we enter the L3 cache regime, memory accesses take long enough to make the kernel memory-bound. This causes the sharp drop-off in performance.

## Alignment & Tail Handling

Here we compare aligned vs unaligned loads for a kernel size of 2 MiB. Unaligned data was offset by 16 bytes on 32 byte boundaries.

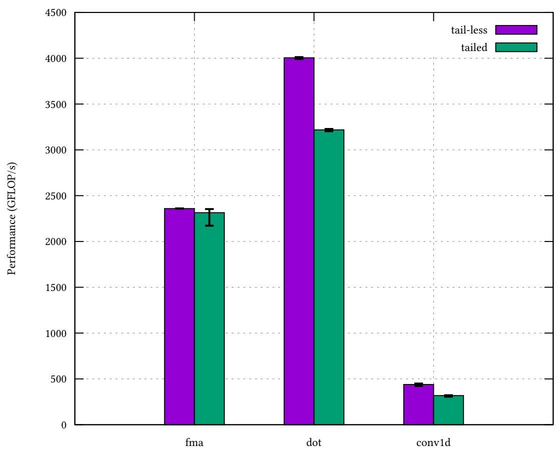


| Kernel | Aligned (GFLOP/s) | Unaligned (GFLOP/s) |
|---|---|---|
| fma | 4.41 | 4.22 |
| dot | 73.06 | 63.91 |
| conv1d | 4.89 | 4.81 |

For the dot kernel, processing aligned data is approximately 14% faster than unaligned. This is due to aligned loads being faster than unaligned loads under certain conditions. If the unaligned load crosses a cache line boundary, both cache lines need to be queried. Additionally, if the unaligned load crosses a page boundary, both pages need to be queried. This can get expensive if the cache or TLB is cold.

For the other two kernels, the unaligned cost is low. This is because they are both streaming kernels where the latency cost is low due to being completely parallelizable.

In the following experiment, we investigate the effects of tail handling. Tail-less Kernels are 32 bytes, tailed kernels are 48 bytes, and the operation is repeated 1000 times.

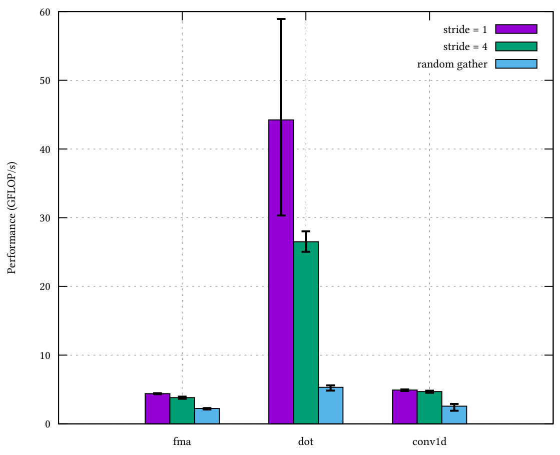


| Kernel | Tail-Less (GFLOP/s) | Tailed (GFLOP/s) |
|---|---|---|
| fma | 2359.00 | 2314.42 |
| dot | 4004.96 | 3218.33 |
| conv1d | 439.06 | 317.70 |

For all kernels, there is a slight performance loss in tailed sizes. This is because the compiler has to emit extra instructions (epilogue) to handle the tail. The compiler can also use masked vector operations but this still requires extra masking instructions.

## Stride/Gather Effects

In this experiment we compare different access patterns: sequential, strided, and gather. Stride = 4 was chosen for the strided access and random indices were chosen for gather. A 2 MiB kernel size was used.



All units are in GFLOP/s

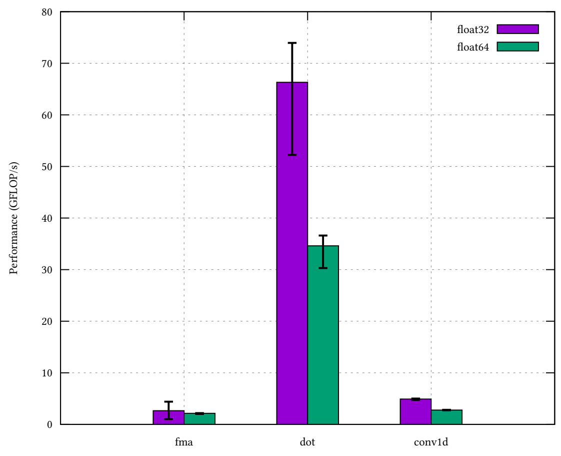
| Kernel | Unit Stride | Stride = 4 | Gather |
|---|---|---|---|
| fma | 4.39 | 3.81 | 2.23 |
| dot | 44.23 | 26.50 | 5.31 |
| conv1d | 4.92 | 4.69 | 2.56 |

Performance degrades from unit stride to stride = 4 to random gather. The effect is most potent in the dot kernel. When using a stride of 4, the cache utilization is significantly reduced. Each load populates a 64 byte cache line but only 16 bytes are actually utilized since the next element resides in the next cache line. This significantly lowers the cache hit rate and leads to decreased performance.

In the case of random gather, the effect is even worse. For each loaded cache line, only 4 bytes are expected to be utilized (32 bit float). Additionally the TLB hit rate is reduced due to loads coming from random pages.

## Data Type Comparison

In this experiment we compare the effect of using 32-bit vs 64-bit floats. Kernel size was 2 MiB.



All units are in GFLOP/s

| Kernel | float32 | float64 |
|---|---|---|
| fma | 2.65 | 2.14 |
| dot | 66.30 | 34.61 |
| conv1d | 4.91 | 2.77 |

For each kernel, more performance is achieved using 32-bit floats. This can be explained by how many vector lanes there are for each data type. For AVX2 256-bit vectors, there are 8 lanes available for 32-bit data types and only 4 lanes for 64-bit data types. This allows faster processing of 32-bit floats.

This effect is strongest in the dot kernel. Again, this can be explained by the dot kernel being reducing. When there are 8 vector lanes, it can reduce across 8 elements at once. When there are only 4 vector lanes, it is half as effective.

## Vectorization Verification

See the following assembly snipets the for the fma kernel. The auto-vectorized version uses vector instructions (prefixed with "v") and the ymm registers showing that it took advantage of AVX2. It also uses the fused multiply-add instruction (`vfmadd213ps`). The scalar version uses the default x86 `ss`-postfixed instruction with the xmm registers.

### Auto-Vectorized

```
fma_kernel:
    ; ...
.L26:
    vmovaps (%rcx,%rax), %ymm1
    vfmadd213ps (%rdx,%rax), %ymm2, %ymm1
    vmovaps %ymm1, (%rdx,%rax)
    ; ...
```

### Scalar

```
fma_kernel:
    ; ...
.L24:
    movss (%rsi,%rax,4), %xmm1
    mulss %xmm0, %xmm1
    addss (%rdx,%rax,4), %xmm1
    movss %xmm1, (%rdx,%rax,4)
    ; ...
```

We see the same for the dot kernel.

**Auto-Vectorized**

```
dot_kernel:
    ; ...
.L80:
    vmovaps (%rcx,%rax), %ymm3
    vmulps (%rdx,%rax), %ymm3, %ymm0
    addq $32, %rax
    vaddps %ymm0, %ymm1, %ymm1
    cmpq %rsi, %rax
    jne .L80
    vextractf128 $0x1, %ymm1, %xmm0
    movq %rdi, %rax
    vaddps %xmm1, %xmm0, %xmm0
    andq $-8, %rax
    vmovhlps %xmm0, %xmm0, %xmm1
    vaddps %xmm0, %xmm1, %xmm1
    vshufps $85, %xmm1, %xmm1, %xmm0
    vaddps %xmm1, %xmm0, %xmm0
    ; ...
```

**Scalar**

```
dot_kernel:
    ; ...
.L49:
    movss (%rsi,%rax,4), %xmm0
    mulss (%rdx,%rax,4), %xmm0
    addq $1, %rax
    addss %xmm0, %xmm1
    cmpq %rax, %rdi
    jne .L49
    ; ...
```

Again, vector instructions are utilized in the conv1d kernel as well.

**Auto-Vectorized**

```
conv1d_kernel:
    ; ...
.L117:
    vmulss -4(%rcx,%rax,4), %xmm0, %xmm3
    vmulss (%rcx,%rax,4), %xmm1, %xmm4
    vaddss %xmm4, %xmm3, %xmm3
    cmpq %rax, %rsi
    je .L125
    vfmadd231ss 4(%rcx,%rax,4), %xmm2,
%xmm3
    vmovss %xmm3, (%rdx,%rax,4)
    ; ...
```

**Scalar**

```
conv1d_kernel:
    ; ...
.L66:
    movss -4(%rsi,%rax,4), %xmm3
    movss (%rsi,%rax,4), %xmm5
    mulss %xmm0, %xmm3
    mulss %xmm1, %xmm5
    cmpq %rdi, %rax
    je .L73
    movss 4(%rsi,%rax,4), %xmm4
    addss %xmm5, %xmm3
    mulss %xmm2, %xmm4
    addss %xmm3, %xmm4
    movss %xmm4, (%rdx,%rax,4)
    ; ...
```