SIMD Advantage Profiling

ECSE 4320

Ben Herman

Content

Experiment Setup

Test Parameter	Description
Simd	Runs with compiler auto-vectorization (SIMD instructions like fadd/fmul/fmla). Otherwise `-fno-slp-vectorize -fno-vectorize` is used.
Saxpy	Computes `out = in1 * a + in2`.
Dot Product	Computes `out += in1 * in2`.
Elementwise Multiply	Computes `out = in1 * in2`.
Stencil	Computes `out = in1*c1 + in2*c2 + in3*c3`.
Use Double	Switches all computation from using `floats` (32-bit) to using `doubles` (64-bit).
Missalignment	Forces deliberately misaligned memory allocation by offsetting pointers.
Odd Size	Adds 1 to array size so the size is not a divisible two.
Stride	Different memory access stride patterns (1, 2, 4, 8). We make sure that the total number of operations is still the same.

Compilation:
- Compiler: Clang
- C++ version: -std=c++17
- Optimizations:
  - -O3
  - -ffast-math (Allows reordering of floating point operations which improves SIMD usage)
- : ensures compatibility with modern C++ features used in the code.

Number of Runs and Array Sizes:
- Array sizes range from 2^9 to 2^22 elements.
- Each array size is tested 20 times.

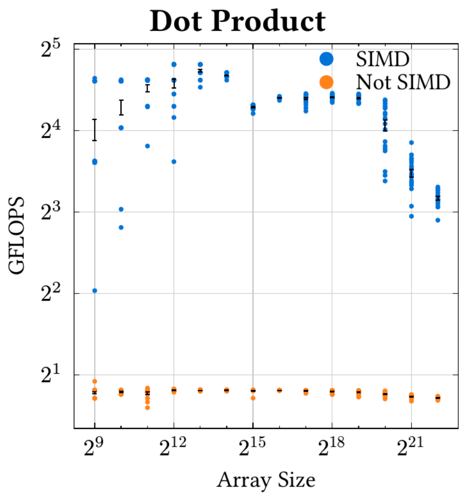
Timing Measurement:
- Execution time is measured using std::chrono::high_resolution_clock.

SIMD Detection:
- If SIMD instructions were used is done by checking if ASM produced with objdump contains instructions like "`fadd.`", "`fmul.`", or "`fmla.`"

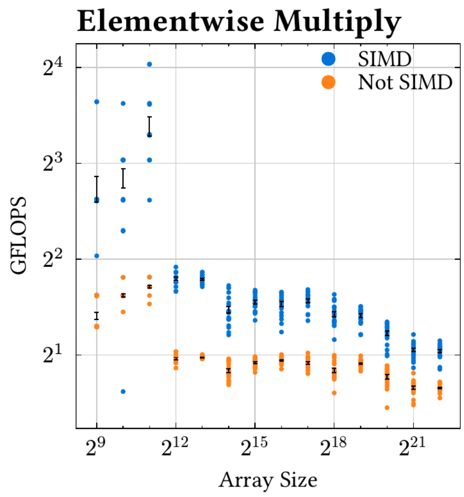
Conditions:
- Model: M2 Mac
- OS: Sequoia 15.6
- Powersource: Wall outlet
- Ram: 16 GB
- Room Temperature: ~65° Fahrenheit

Scalar vs Auto-Vectorized

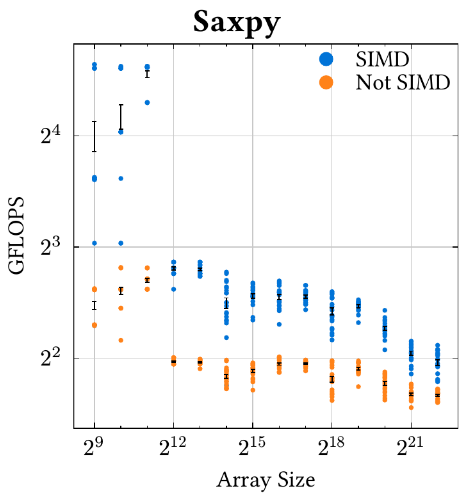
When using SIMD instructions the code did more GFLOPS than the when not using SIMD instructions. The Dot Product test did the most GFLOPS till the array got bigger than 2^{21} which then the Stencil test did the most GFLOPS. The Elementwise Multiply test did the least GFLOPS with the Saxpy test doing roughly 2 times the number of GFLOPS.



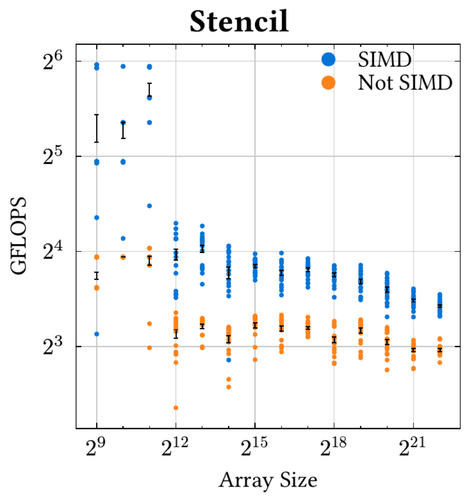
Speedup: 11.454397112940903



Speedup: 1.9083927683610362



Speedup: 2.063606664991188



Speedup: 2.019751276027687

Locality Sweep

The Scalar vs Auto-Vectorized test is also good for looking at the how array size effects the speed. For array sizes under 2^{12} the data is very noisy due to the small anount of work.

With no SIMD the GFLOPS is most constant with a very slight trend downward which is probably due to the cache not being quite fast enough to keep up with the CPU.

With SIMD the GFLOPS has changes in the same way for all tests. First its very noisy due to the small work load. Then at 2^{12} (16,384 bytes) it starts looking stable. From 2^14 (65,536 bytes) to 2^20 (4,194,304 bytes) GFLOPS decreases slowly. Right after that the graph drops a lot.

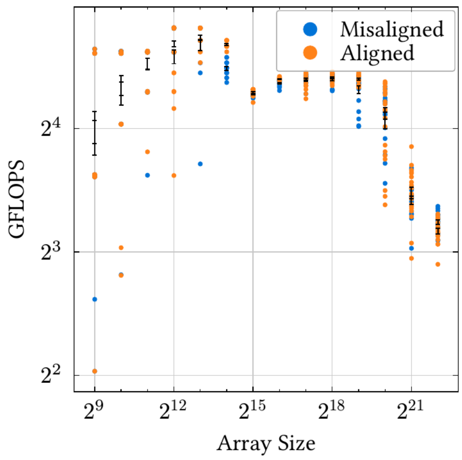
This first drop happens when the data is larger than L1 cache which 64 KB. The second larger drop it happens when the data is larger than L2 cache 4 MB.

Alignment & Tail Handling

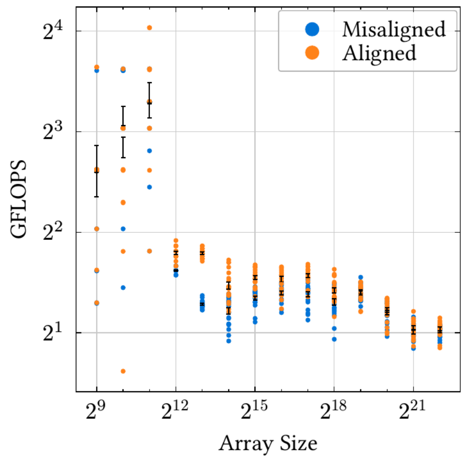
This is split between shifting the starting pointer and changing the length of the array. In both cases the compiler was still able to use SIMD.

When the array did not start at the start of the allocations the performance was slightly worse but orverall the missalignment did not effects the speed very much.

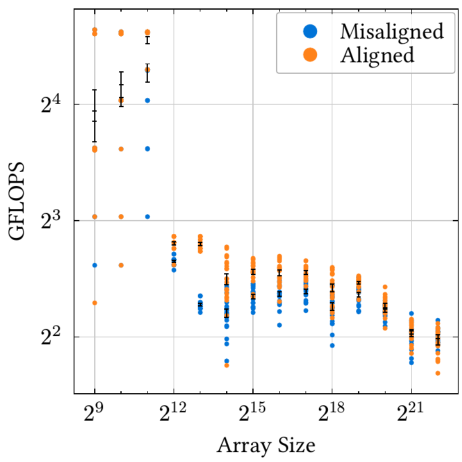
Dot Product: Misaligned vs Aligned



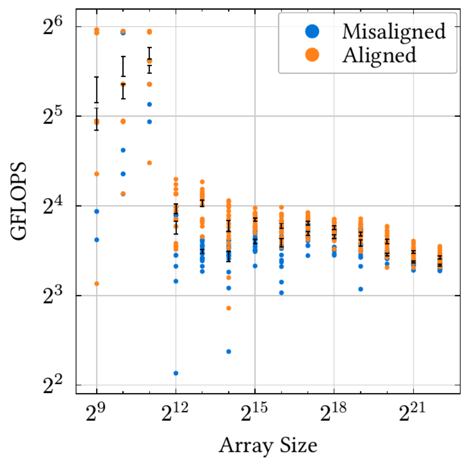
Elementwise Multiply: Misaligned vs Aligned



Saxpy: Misaligned vs Aligned

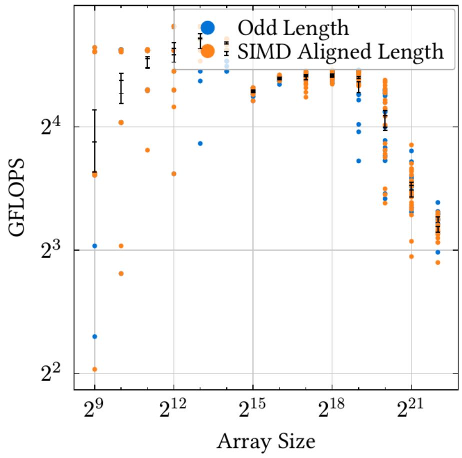


Stencil: Misaligned vs Aligned

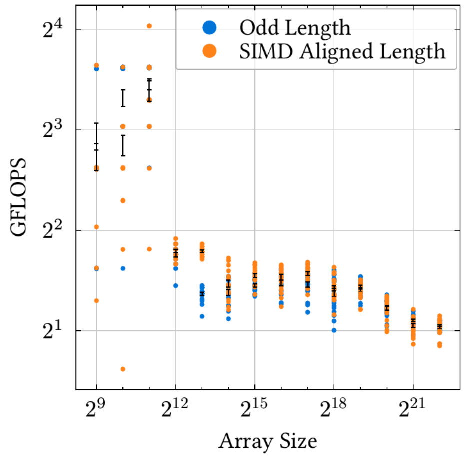


When the arrays length was not a multiple of the SIMD register's width (4 floats, 16 bytes) the speed was not effected noticeably. This is because the compiler was able to just put a small piece of code that corrected the last elements not fitting in the SIMD register.

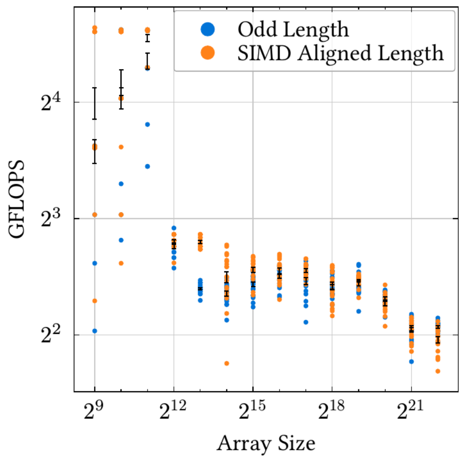
Dot Product:
Odd Length vs SIMD Aligned Length



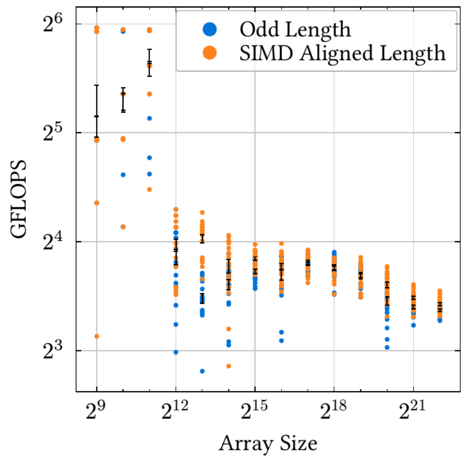
Elementwise Multiply:
Odd Length vs SIMD Aligned Length



Saxpy:
Odd Length vs SIMD Aligned Length



Stencil:
Odd Length vs SIMD Aligned Length

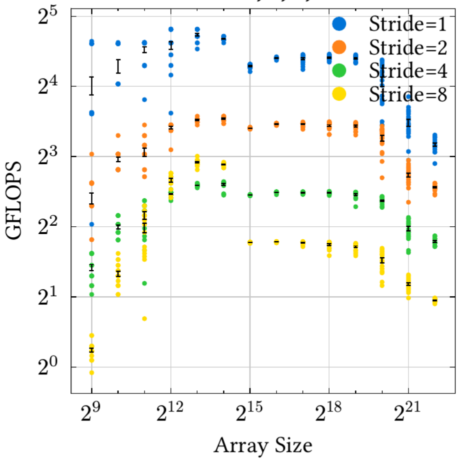


Stride / Gather Effects

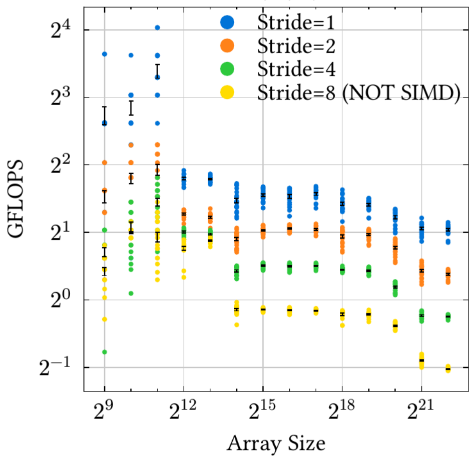
In all cases we see that increasing the stride decrease the effectivnness of using SIMD instructions. I belive this is because the cpu has to first align the floats and then run them through SIMD instructions.

Its also worth noting that when taking Strides of 8 the Elementwise Multiply and Saxpy did not compile with SIMD instructions. This may be because the compile thought the the alignment would cost more than the SIMD instructions would save. We can see this in the data because the amount of GFLOP/S loss when to a stride of 8 in Elementwise Multiply and Saxpy is similar to the loss in Dot Product and Stencil.

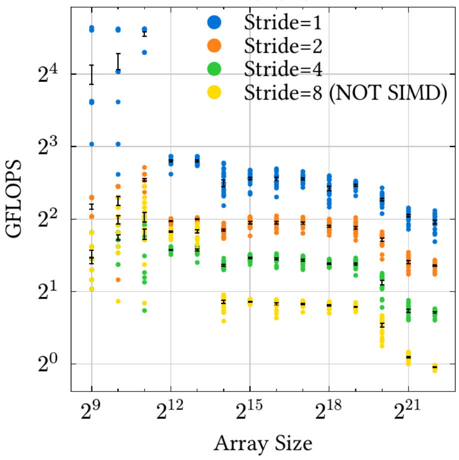
Dot Product:
Strides:1,2,4,8



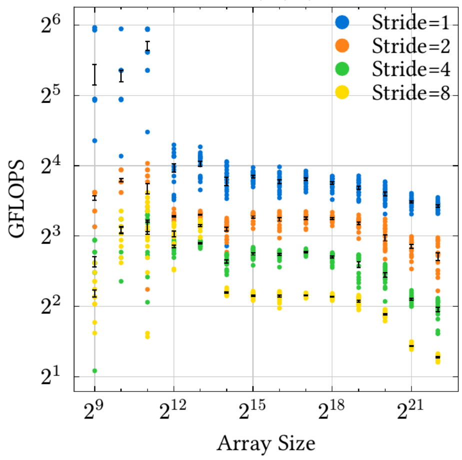
Elementwise Multiply:
Strides:1,2,4,8



Saxpy:
Strides:1,2,4,8

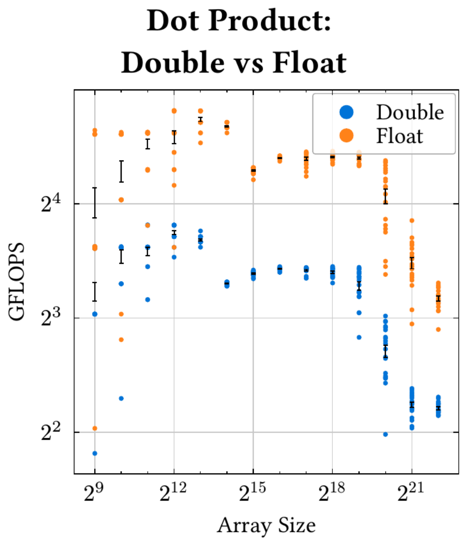


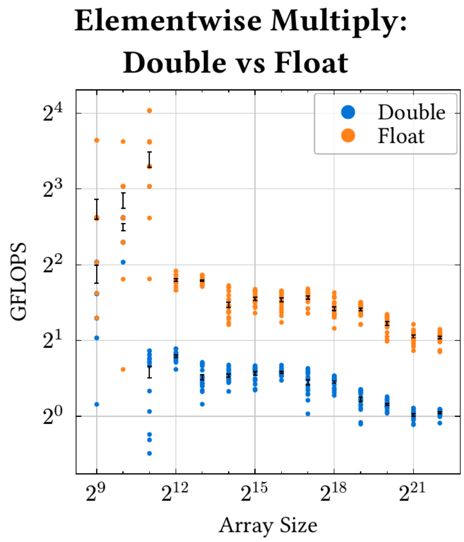
Stencil:
Strides:1,2,4,8

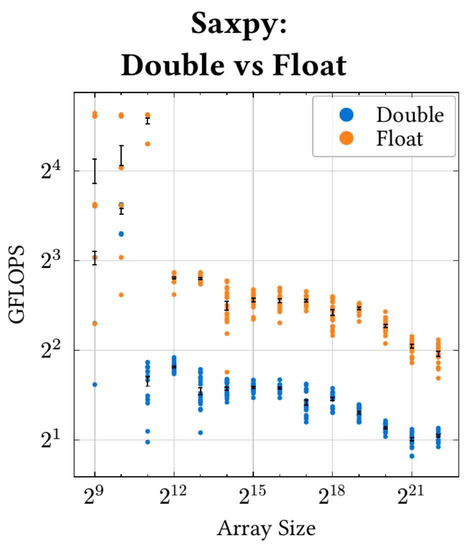


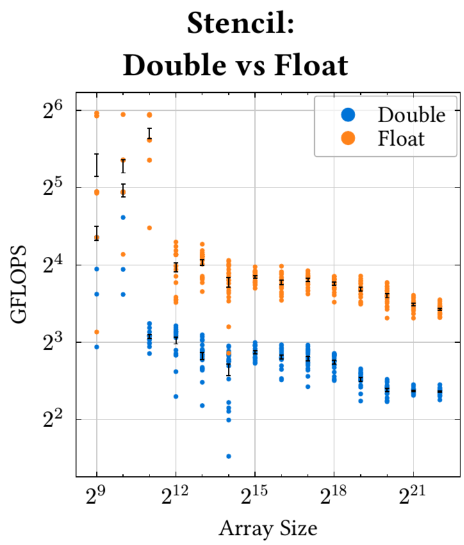
Data Type Comparison

Using doubles instead of floats was twice as slow. This make sense because double SIMD instructions only operate on 2 numbers at a time opposed to float SIMD instructions with operate on 4 numbers at a time.









Vectorization Verification

To do verify that the code is using SIMD instructions I dumped the assembly to a file with the command `objdump -d file`. Then I used python to search through the file for strings like (`fmul.` `fadd.` `fmla.`). These strings are a floating point operation followed by a dot. The actual instructions are `fmul.4` or `fmul.2` which are for single precision and double precision floating point respectively.

To show this working we have a example from the files
"`dump_...-DDOT_PRODUCT_-DSTRIDE=1_(512, 4194304)_SIMD`"
and
"`dump_...-DDOT_PRODUCT_-DSTRIDE=1_-fno-slp-vectorize_-fno-vectorize_(512, 4194304)`"

The first file does end with "`_SIMD`" which signals that there was SIMD instructions found in the assembly.

The second file does not end with "`_SIMD`" which means there where no SIMD instructions found. This makes sense because I added the flags "`-fno-slp-vectorize`" and "`-fno-vectorize`" which stops the compiler from using any SIMD instructions.

Looking in the first file we see SIMD instructions.

```
...
1000006d4: acc24d32     ldp q18, q19, [x9], #0x40
1000006d8: 4e24ce00     fmla.4s v0, v16, v4
1000006dc: 4e25ce21     fmla.4s v1, v17, v5
1000006e0: 4e26ce42     fmla.4s v2, v18, v6
1000006e4: 4e27ce63     fmla.4s v3, v19, v7
1000006e8: f100416b     subs  x11, x11, #0x10
1000006ec: 54fffee1     b.ne  0x1000006c8 <__Z4testIfEvj+0xf4>
1000006f0: 4e20d420     fadd.4s v0, v1, v0
1000006f4: 4e22d461     fadd.4s v1, v3, v2
1000006f8: 4e20d420     fadd.4s v0, v1, v0
1000006fc: 6e20d400     faddp.4s  v0, v0, v0
100000700: 7e30d802     faddp.2s  s2, v0
100000704: eb17011f     cmp x8, x23
...
```

Looking in the second file we see no SIMD instructions.

```
...
1000006b8: d29ad2a8     mov x8, #0xd695             ; =54933
1000006bc: f2bd04c8     movk  x8, #0xe826, lsl #16
1000006c0: f2c5c168     movk  x8, #0x2e0b, lsl #32
1000006c4: f2e7c228     movk  x8, #0x3e11, lsl #48
1000006c8: 9e670101     fmov  d1, x8
1000006cc: 1e610800     fmul  d0, d0, d1
1000006d0: 90000020     adrp  x0, 0x100004000 <_strlen+0x100004000>
1000006d4: f9400c00     ldr x0, [x0, #0x18]
1000006d8: 94000035     bl  0x1000007ac <_strlen+0x1000007ac>
1000006dc: 910033e8     add x8, sp, #0xc
...
```

This shows that the code is doing what we expected and that we are actually using SIMD instructions when we are marking data as SIMD.

Roofline Interpretation