Lab 8: Microbenchmarks and SIMD Instructions

1 Membench

The *membench* programs are run to obtain the following plots, from Code. 5, in ?? to Fig. 3 for the effect of array length and stride length on type of memory access, and time to access.

In this plot, the solid vertical lines indicate the estimated memory sizes, and dashed vertical lines indicate the true memory sizes, with black lines indicating the cache line sizes, and blue lines indicating the total cache sizes. The green solid horizontal lines indicate the approximate cache access times.

In general, it is observed that for a given range of strides, and particularly in regions with plateaus of strides, say between 64B and 4KB, greater array lengths have greater access times. These parallel plateaus can be attributed to filling and then calling the lowest possible caches/accessing the lowest possible level of memory when there are cache misses in the even lower level of memory. This accessing is constant for the given lowest possible memory level.

Here, lowest possible memory level refers to the smallest cache at which the stride is less than the cache size and or line size. At this lowest possible level of memory, there will be the least number of cache misses, as it will be assumed that the array will fill this level of cache with as many full cache lines as possible. Larger arrays will fill more cache lines at that level, up to the level being full (and disregarding space being required for the code/instructions on top of space for the data). This filling also depends on whether a cache miss prompts a cache line to be overwritten, or a new cache line to be filled with data from a higher memory level, This depends on the specific association, inclusivity/exclusivity, hierarchy and relative sizes of the different cache levels.

So for larger arrays, the number of iterations through the array is obviously greater, and must take more time, and more cache misses will occur, regardless of the relative difference in stride size to cache line size. This greater number of misses (and corresponding new cache lines being filled from higher memory levels) is likely linear in the array size. This linear increase in cache misses explains the parallel plateaus at higher accessing times for greater array sizes, even if the time to access each element within a given cache size is roughly constant.

The access times will be measured where there are significant plateaus in the access times, and the *maximum* of these plateaus will be used as an estimate for the access times. The reason there are not sharp increases between plateaus is that as the stride increases, there are less cache hits and more misses as the indexing goes beyond the cache line size, however there are also less indexing calls with greater stride. So the increase up to the next plateau, which corresponds to the memory access time of the next greater memory level, is gradual, as there is a mix of hits in different memory levels.

The following script *calc.sh* in the Code. 6 in the appendix was used to get the cpu and memory statistics in Code. 1. The average processor core speed for the 36 processors on GreatLakes to be 3000 MHz, and the cache line sizes appear to be constant at 64 Bytes. The cpu core speed will be used to calculate the number of processor clock cycles required to access the various types of memory,

$$\text{\# cycles per memory access} = \text{clock speed} \times \text{memory access time.}$$

Code 1: CPU and Memory values.

```
Cache Info
Cache L1 Size: 32 kB
Cache L1 Line Size: 64 B

Cache L2 Size: 1024 kB
Cache L2 Line Size: 64 B

Cache L3 Size: 25344 kB
Cache L3 Line Size: 64 B

```

```
Cache L4 Size: 0 kB
Cache L4 Line Size: 0 B




CPU Info
CPU Cores: 36
CPU Speed: 2999.531 MHz
```

1.1 Processor Values

On the Greatlakes compute notes, there are 36 × Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz. From repeated requests for the speed of each core, the average core speed over all cores and samples is 2.99 GHz.

1.2 L_1 Cache Values

For the L_1 cache line size, the true line size is 64B, and from the plots, particularly in Fig. 2, it can be seen that the access times do not start to initially increase until around strides of 64B, before then plateauing. This jump before plateauing indicates that strides greater than this value must start to involve more L_1 misses, and require accessing the L_2 memory.

$$L_1 \text{ Line } = 128B.$$

For the L_1 total cache size, although the true cache size is 32KB, from the plots, particularly in Fig. 2, it can be seen that the access times remain very constant, and at their minimum all way the up to 4KB for arrays with length ≤ 32KB, suggesting the entire array, or at least half of the array can be loaded into the L_1 cache. In addition, the next largest 62KB array shows increased access times for up to 32KB strides, suggesting some L_1 cache misses possibly occur, and so the L_1 cache size is likely less than 32KB.

$$16KB \leq L_1 \leq 32KB.$$

For the L_1 access time, given the quite constant access times up to 4KB strides for arrays with length ≤ 32KB, the estimated access time is therefore the maximum of the 32KB length array curve:

$$T_1 = 0.57\text{ns} = 2 \text{ cycles}.$$

1.3 L_2 Cache Values

For the L_2 cache line size, the true line size is 64B, and from the plots, particularly in Fig. 3, it can be seen that after the initial increase of access times, there is a slight plateau for strides ≤ 512B, and arrays of sizes 64KB-8MB, indicating that possibly the array is being quickly indexed in the L_2 cache with a cache line of between 64B and 512B. There is then a jump in access times, but not a huge jump for arrays up to size 8MB, indicating there are possibly still cache hits in the L_2 cache, but at different lines.

$$64B \leq L_2 \text{ Line } \leq 512B.$$

For the L_2 total cache size, although the true cache size is 1MB, from the plots, particularly in Fig. 3, it can be seen that the access times remain very constant over a large range of array sizes between 64KM to 8MB, with strides between 1KB and 256KB. This suggests lots of these array sizes can be mostly loaded into the L_2 cache on several cache lines. This suggests the L_2 total cache size to be less than 256KB (and greater than the L_1 total cache size); the point where the access times for these array sizes drops dramatically when many less array elements are indexed. The difficulty at finding tighter bounds on the L_2 cache sizes is possibly due to the L_2 cache sometimes being shared by pairs of cores, affecting the timing, depending on which cores the array is being computed on.

$$16KB \leq L_2 \leq 256KB.$$

For the L_2 access time, given the quite constant access times for array sizes between 64KM to 8MB, with strides between 1KB and 256KB, the estimated access time is therefore the maximum of the 8MB size array curve along this plateau:

$$T_2 = 3.83\text{ns} = 12 \text{ cycles}.$$

1.4 L_3 Cache Values

For the L_3 cache line size, the true line size is 64B, however from the plots, it is difficult to tell where exactly there is a distinct plateau for array indexing with strides within the size of this larger cache's lines. This may be due to the L_3 cache being typically shared between all (36) cores, and so timings may be affected depending how the computations are distributed amongst the cores. However for array sizes of 16MB to 512MB, there is somewhat a plateau between 512B and 4KB, indicating a possible range for the L_3 cache line size. Here, there may be hits due to this larger cache being allowed to store more of these larger arrays, minimising cache misses. The lack of distinct plateau is also possibly attributed to the large arrays being far larger than the cache, and there being many hits and misses while the lines are being filled from the main memory.

$$64B \leq L_3 \text{ Line } \leq 4KB.$$

For the L_3 total cache size, although the true cache size is 24.75MB, from the plots, particularly in Fig. 4, it can be seen that the plateau between 512B and 4KB, for array sizes of 16MB to 512MB, rises, and then decreases gradually, before plateauing again for strides between 256KB and 8MB. This suggests that the time is not decreasing solely due to there being less elements indexed with greater stride, but there also possibly being effects of the elements still being in the faster L_3 cache compared to the main memory. There are still hits occurring in succession in a cache that are causing this plateau at non-zero access times. There is still though great uncertainty in the exact total size of this L_3 cache.

$$256KB \leq L_3 \leq 8MB.$$

For the L_3 access time, given the two different plateaus present in the access times for the larger arrays, the access time will be estimated as the maximum of these plateaus in Fig. 4.

$$T_3 = 10.36\text{ns} = 32 \text{ cycles}.$$

1.5 Main Memory Values

Only the 1GB array sizes appear to be unable to be stored fully in any caches, and there are enough misses in the lower caches that the main memory must be accessed. It is assumed the upper plateau for the 1GB array are these memory hits, and the access time is assumed to be the maximum of this plateau. This is assumed to be a lower bound, if some of the array is in the L_3 cache, and there are some hits there, and some in the main memory.

$$T_{\text{mem}} \geq 13.69\text{ns} = 42 \text{ cycles}.$$

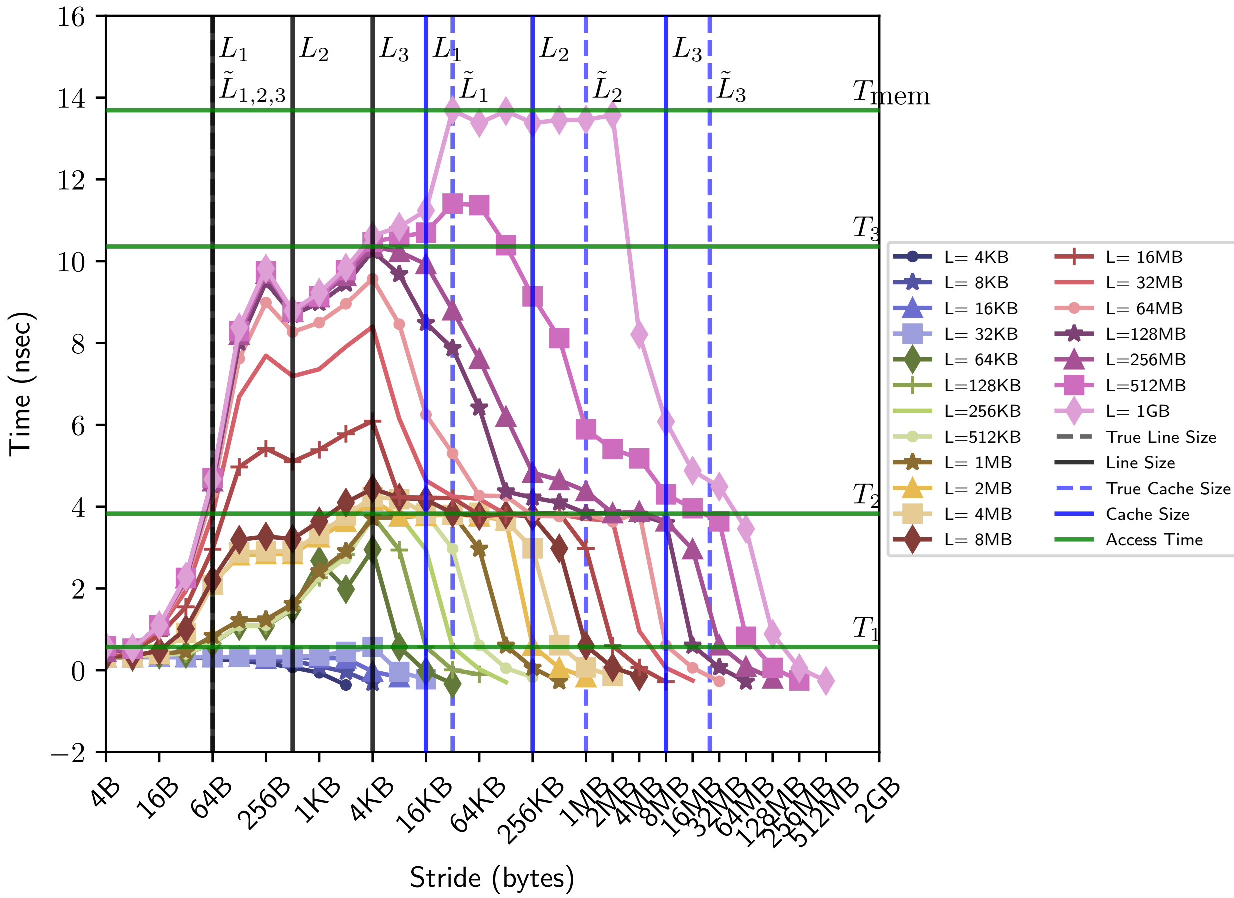

Figure 1: Memory access times for various array sizes, and stride lengths from 4B to 512MB.

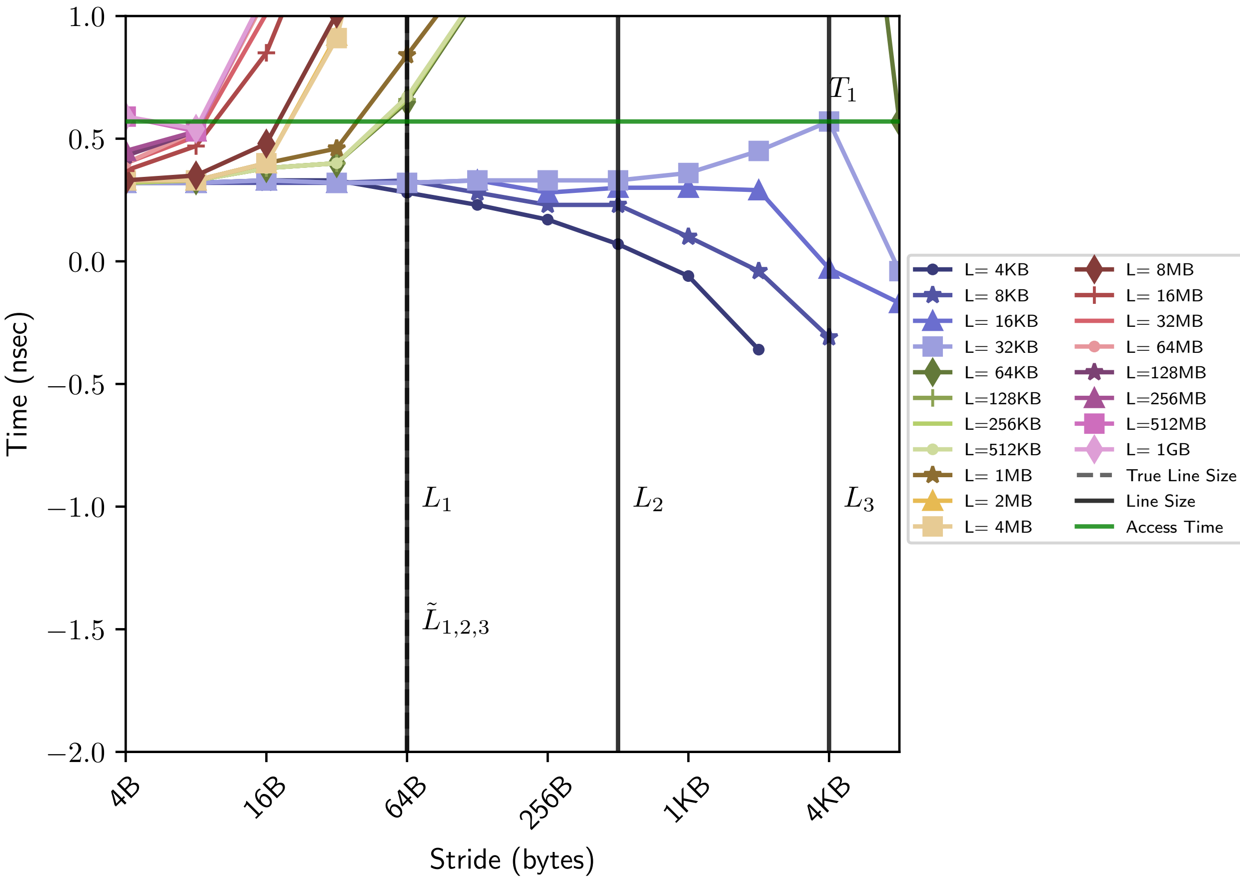

Figure 2: Memory access times for various array sizes, and stride lengths from 4B to 4KB.

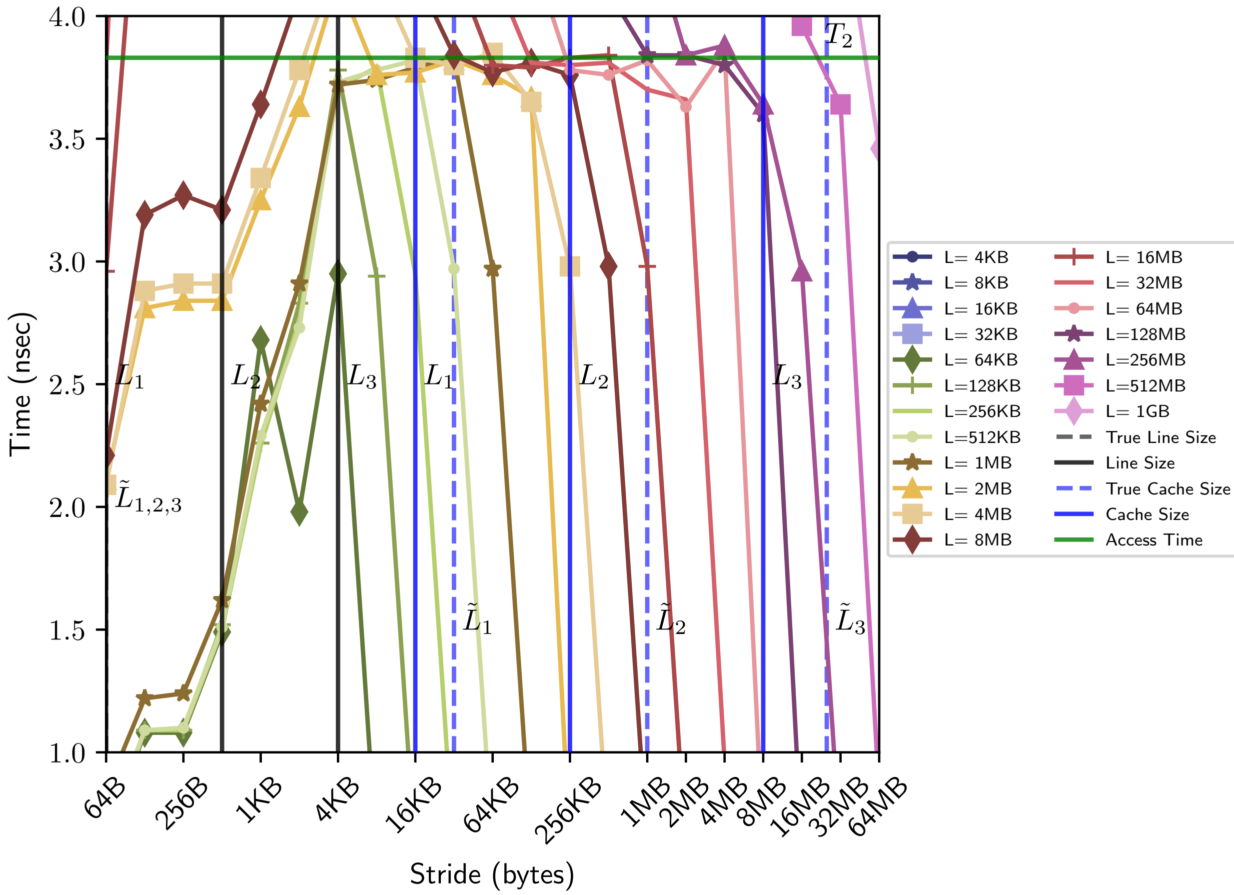

Figure 3: Memory access times for various array sizes, and stride lengths from 16B to 64MB.

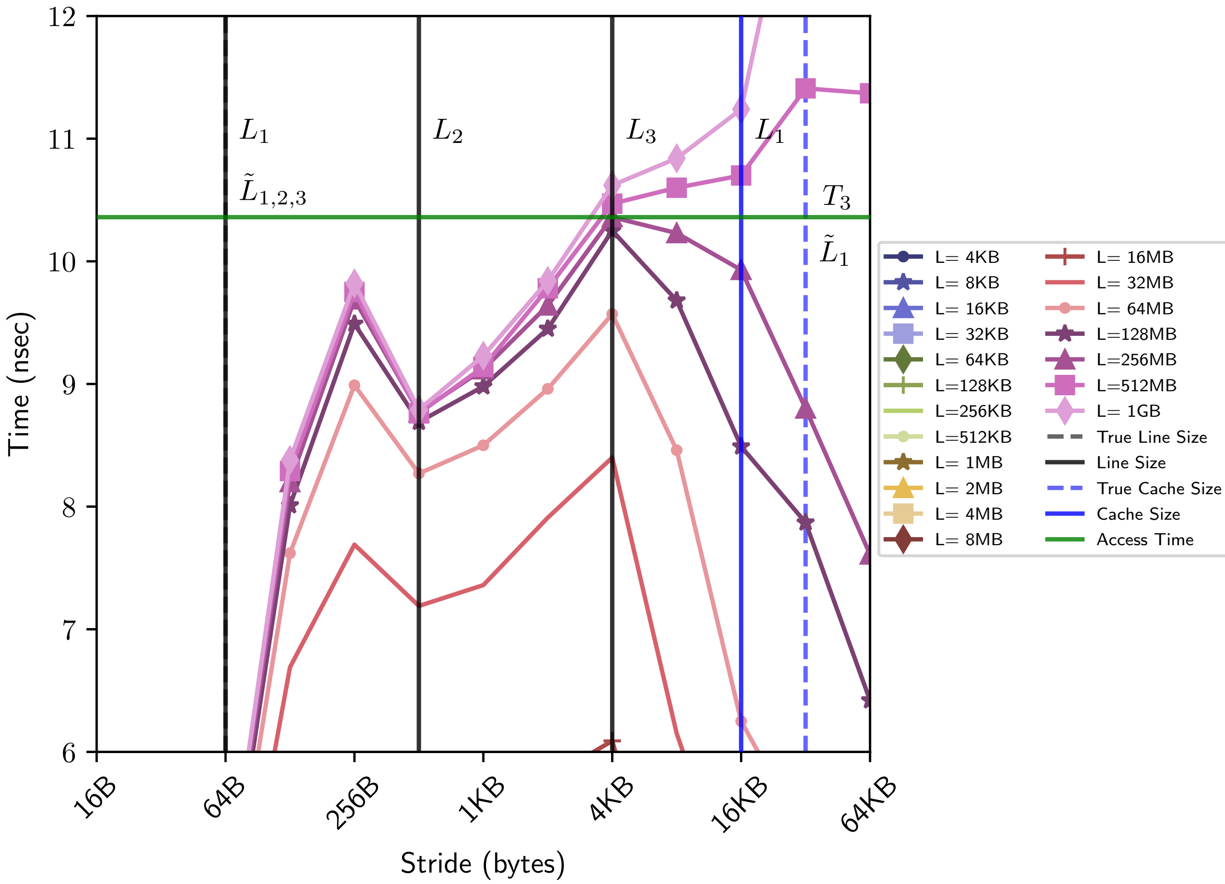

Figure 4: Memory access times for various array sizes, and stride lengths from 16B to 64KB.

2 SIMD Instructions

2.1 AVX Support

- The commands used to verify if the current machine/processor supports AVX are:

  grep -E "avx[^0-9]" /proc/cpuinfo

  and if the current machine/processor supports AVX2 are:

  grep -E "avx2" /proc/cpuinfo

  These searches will show whether the supported AVX fields are in the flags section of the processor info from /proc/cpuinfo. This command will show all processors (36 × Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz, for GreatLakes compute nodes).

- The commands used to verify if the current GNU compiler supports AVX or AVX2 by the constants/macros the compiler defines:

  gcc -mavx2 -dM -E - < /dev/null | grep "AVX" | sort

  This command will show the boolean settings for the AVX constants:
  #define __AVX__ 1
  #define __AVX2__ 1

- The commands used to verify if the current Intel compiler supports AVX or AVX2 by the constants/macros the compiler defines:

  icc -march=native -dM -E - < /dev/null | grep "AVX" | sort

  This command will show the boolean settings for the AVX constants:
  #define __AVX__ 1
  #define __AVX2__ 1
  #define __AVX512BW__ 1
  #define __AVX512CD__ 1
  #define __AVX512DQ__ 1
  #define __AVX512F__ 1
  #define __AVX512VL__ 1
  #define __AVX_I__ 1

2.2 dgemm Assembly Instructions

The command to get the assembly instructions is as follows, where the function for the naive *dgemm.cpp* in Code. 8 in the appendix is translated into assembly code, with the avx2 and fast optimizations using the commands for gnu and intel compilers:

g++ -S -Ofast -mavx2 -mfma dgemm.cpp

icc -S -Ofast -march=core-avx2 dgemm.cpp

This command (with the gnu compiler) produces the following assembly code in Code. 3. Here it can be seen in lines 81-120, there is the assembly code for the three loops (add, *add*,compute, *cmp*, jump, jne, as well as the vectorized multiply *vmulsd*, add *add*, and vectorized move *vmovsd*. The specific SIMD fused multiply-add commands are *vfmadd132sd*.

The assembly instructions are known to contain AVX2 instructions because they are writing to the AVX specific register keywords *ymm*, and AVX2 is confirmed, because fused-multiply-add commands (such as *vfmadd132sd*) are in the assembly instructions.

Considering the correct compiler options and hardware compatibility for AVX2 instructions has appeared to be verified, no modifications of the original matrix-matrix multiplication source code were necessary. The mydgemm, compiled with the above avx2 and fma options, as well as the fast optimizations, does not appear on GreatLakes to run faster, at least for matrix sizes up to 2000.

Code 2: Matrix-matrix multiplication loop assembly code

```
        testl       %r8d, %r8d
        je      .L29
        movq        16(%rbp), %r12
        leal        -1(%r8), %eax
        leaq        8(,%r14,8), %rbx
        xorl        %r8d, %r8d
        leaq        8(%r12,%rax,8), %r13
        .p2align 4,,10
        .p2align 3
.L10:
        movq        (%r12), %r14
        xorl        %esi, %esi
        .p2align 4,,10
        .p2align 3
.L9:
        leaq        (%r14,%rsi), %rdi
        xorl        %eax, %eax
        jmp     .L8
        .p2align 4,,10
        .p2align 3
.L13:
        movq        %rdx, %rax
.L8:
        movq        (%r11,%rax,8), %rdx
        movq        (%r9,%rax,8), %rcx
        vmulsd      (%rdi), %xmm0, %xmm1
        addq        %rsi, %rdx
        vmovsd      (%rdx), %xmm4
        vfmadd132sd     (%rcx,%r8), %xmm4, %xmm1
        vmovsd      %xmm1, (%rdx)
        leaq        1(%rax), %rdx
        cmpq        %rax, %r10
        jne     .L13
        addq        $8, %rsi
        cmpq        %rsi, %rbx
        jne     .L9
        addq        $8, %r12
        addq        $8, %r8
        cmpq        %r12, %r13
        jne     .L10
```

Code 3: Matrix-Matrix multiply assembly with avx2 and fast optimization

```
        .file       "dgemm.cpp"
        .text
        .p2align 4,,15
        .globl      _Z5dgemmccjjjdPKPKdS2_dPPd
        .type       _Z5dgemmccjjjdPKPKdS2_dPPd, @function
```

```
_Z5dgemmccjjjdPKPKdS2_dPPd:
.LFB1538:
        .cfi_startproc
        pushq       %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        pushq       %r14
        movq        24(%rbp), %r11
        pushq       %r13
        pushq       %r12
        pushq       %rbx
        .cfi_offset 14, -24
        .cfi_offset 13, -32
        .cfi_offset 12, -40
        .cfi_offset 3, -48
        testl       %ecx, %ecx
        je          .L27
        testl       %edx, %edx
        je          .L27
        movl        %edx, %r12d
        leal        -1(%rcx), %eax
        movl        %edx, %ebx
        movq        %r11, %rdi
        shrl        $2, %r12d
        movq        %rax, %r10
        leaq        8(%r11,%rax,8), %r13
        andl        $-4, %ebx
        leal        -1(%rdx), %r14d
        salq        $5, %r12
        vbroadcastsd        %xmm1, %ymm3
        .p2align 4,,10
        .p2align 3
.L7:
        movq        (%rdi), %rsi
        cmpl        $2, %r14d
        jbe         .L12
        movq        %rsi, %rax
        leaq        (%r12,%rsi), %rcx
        .p2align 4,,10
        .p2align 3
.L4:
        vmovupd        (%rax), %xmm5
        vinsertf128        $0x1, 16(%rax), %ymm5, %ymm2
        addq        $32, %rax
        vmulpd        %ymm3, %ymm2, %ymm2
        vmovups        %xmm2, -32(%rax)
        vextractf128        $0x1, %ymm2, -16(%rax)
        cmpq        %rax, %rcx
        jne         .L4
        movl        %ebx, %eax
        cmpl        %ebx, %edx
        je          .L5
.L3:
        movl        %eax, %ecx
        leaq        (%rsi,%rcx,8), %rcx
```

```
        vmulsd          (%rcx), %xmm1, %xmm2
        vmovsd          %xmm2, (%rcx)
        leal            1(%rax), %ecx
        cmpl            %ecx, %edx
        jbe             .L5
        leaq            (%rsi,%rcx,8), %rcx
        addl            $2, %eax
        vmulsd          (%rcx), %xmm1, %xmm2
        vmovsd          %xmm2, (%rcx)
        cmpl            %eax, %edx
        jbe             .L5
        leaq            (%rsi,%rax,8), %rax
        vmulsd          (%rax), %xmm1, %xmm2
        vmovsd          %xmm2, (%rax)
.L5:
        addq            $8, %rdi
        cmpq            %r13, %rdi
        jne             .L7
        testl           %r8d, %r8d
        je              .L29
        movq            16(%rbp), %r12
        leal            -1(%r8), %eax
        leaq            8(,%r14,8), %rbx
        xorl            %r8d, %r8d
        leaq            8(%r12,%rax,8), %r13
        .p2align 4,,10
        .p2align 3
.L10:
        movq            (%r12), %r14
        xorl            %esi, %esi
        .p2align 4,,10
        .p2align 3
.L9:
        leaq            (%r14,%rsi), %rdi
        xorl            %eax, %eax
        jmp             .L8
        .p2align 4,,10
        .p2align 3
.L13:
        movq            %rdx, %rax
.L8:
        movq            (%r11,%rax,8), %rdx
        movq            (%r9,%rax,8), %rcx
        vmulsd          (%rdi), %xmm0, %xmm1
        addq            %rsi, %rdx
        vmovsd          (%rdx), %xmm4
        vfmadd132sd             (%rcx,%r8), %xmm4, %xmm1
        vmovsd          %xmm1, (%rdx)
        leaq            1(%rax), %rdx
        cmpq            %rax, %r10
        jne             .L13
        addq            $8, %rsi
        cmpq            %rsi, %rbx
        jne             .L9
        addq            $8, %r12
        addq            $8, %r8
        cmpq            %r12, %r13
```

```
        jne        .L10
.L29:
        vzeroupper
.L27:
        popq        %rbx
        popq        %r12
        popq        %r13
        popq        %r14
        popq        %rbp
        .cfi_remember_state
        .cfi_def_cfa 7, 8
        ret
.L12:
        .cfi_restore_state
        xorl        %eax, %eax
        jmp        .L3
        .cfi_endproc
.LFE1538:
        .size        _Z5dgemmccjjjdPKPKdS2_dPPd, .-_Z5dgemmccjjjdPKPKdS2_dPPd
        .section        .text.startup,"ax",@progbits
        .p2align 4,,15
        .type        _GLOBAL__sub_I__Z5dgemmccjjjdPKPKdS2_dPPd, @function
_GLOBAL__sub_I__Z5dgemmccjjjdPKPKdS2_dPPd:
.LFB2019:
        .cfi_startproc
        subq        $8, %rsp
        .cfi_def_cfa_offset 16
        movl        $_ZStL8__ioinit, %edi
        call        _ZNSt8ios_base4InitC1Ev
        movl        $__dso_handle, %edx
        movl        $_ZStL8__ioinit, %esi
        movl        $_ZNSt8ios_base4InitD1Ev, %edi
        addq        $8, %rsp
        .cfi_def_cfa_offset 8
        jmp        __cxa_atexit
        .cfi_endproc
.LFE2019:
        .size        _GLOBAL__sub_I__Z5dgemmccjjjdPKPKdS2_dPPd,
          .-_GLOBAL__sub_I__Z5dgemmccjjjdPKPKdS2_dPPd
        .section        .init_array,"aw"
        .align 8
        .quad        _GLOBAL__sub_I__Z5dgemmccjjjdPKPKdS2_dPPd
        .local        _ZStL8__ioinit
        .comm        _ZStL8__ioinit,1,1
        .hidden        __dso_handle
        .ident        "GCC: (GNU) 8.2.0"
        .section        .note.GNU-stack,"",@progbits
```

3 In-line Assembly

With help from the internet, the following matrix-vector multiplication $y = Ax$, in-line assembly,as well as conventional c code, is shown in Code. 9 in the appendix. The fused-multiply adds assembly was not quite able to be implemented, however the procedure of writing the fused-multiply adds with avx instructions (in the assembly syntax, as opposed to the intel syntax) was attempted. There are no compiler errors, however the issue seems to be that the y value is not being assigned the computed multiplied value in the set register in *_mm_storel_pd(y, _mm256_castpd256_pd128(Ax))*.

The assembly registers for multiplies and adds currently only allow for operating on 4×4 matrices if double precision is used. So, blocks of the matrix and vector in these sizes must be passed to the assembly function *_matvec_avx(A,x,y)* in a loop. This loop can then be unrolled, to assign each part of y separately.

The program is compiled with the following command, and assembly snippet is shown in Code. 4.

gcc -Ofast -mavx2 -mfma -funroll-all-loops matvec.c

Code 4: Matrix-Vector multiplication inline assembly code

```
void _matvec_avx(const double* A,const double* x,double* y){
    asm volatile ("# avx code begin");  // looking at assembly with gcc -S
    __m256d xrow = _mm256_loadu_pd(x);

    __m256d a = _mm256_mul_pd(_mm256_loadu_pd(A), xrow);
    __m256d b = _mm256_mul_pd(_mm256_loadu_pd(A+4), xrow);
    __m256d c = _mm256_mul_pd(_mm256_loadu_pd(A+8), xrow);
    __m256d d = _mm256_mul_pd(_mm256_loadu_pd(A+12), xrow);

    // our task now is to get {sum(a), sum(b), sum(c), sum(d)}
    // This is tricky because there is no hadd instruction for avx

    // {a[0]+a[1], b[0]+b[1], a[2]+a[3], b[2]+b[3]}
    __m256d sumab = _mm256_hadd_pd(a, b);

    // {c[0]+c[1], d[0]+d[1], c[2]+c[3], d[2]+d[3]}
    __m256d sumcd = _mm256_hadd_pd(c, d);

    // {a[0]+a[1], b[0]+b[1], c[2]+c[3], d[2]+d[3]}
    __m256d blend = _mm256_blend_pd(sumab, sumcd, 0b1100);

    // {a[2]+a[3], b[2]+b[3], c[0]+c[1], d[0]+d[1]}
    __m256d perm = _mm256_permute2f128_pd(sumab, sumcd, 0x21);

    // {sum(a), sum(b), sum(c), sum(d)}
    __m256d Ax = _mm256_add_pd(perm, blend);
    // printf("yinit = %f",*y);
    _mm_storel_pd(y, _mm256_castpd256_pd128(Ax));
    // printf("yfinal = %f",*y);
    // _mm_storel_pd(y, Ax);
    asm volatile ("# avx code end");

};
```

4 Appendix

Code 5: membench plotting script.

```
#!/usr/bin/env python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

matplotlib.rcParams['text.usetex'] = True
# matplotlib.rcParams['text.latex.preamble'] = [r'\usepackage{ragged2e'}]

# Routine modified from:
  https://stackoverflow.com/questions/1094841/reusable-library-to-get-human-readable-version-of-file-size
def sizeof_fmt(num, suffix='B'):
    for unit in ['','K','M','G','T']:
        if abs(num) < 1024.0:
            return '%3.0f%s%s' % (num, unit, suffix)
        num /= 1024.0
    return '%.1f%s%s' % (num, 'T', suffix)

def fmt_sizeof(fmt):
    bases={'B':2,'':10}
    units={k:v for k,v in zip(['','K','M','G','T'],[1,10,20,30,40])}
    try:
      base=fmt[-1]
      unit=fmt[-2]
      num = float(fmt[:-2])
    except ValueError:
      try:
        base=fmt[-1]
        unit=''
        num = float(fmt[:-1])
      except:
        try:
          base=''
          unit=''
          num = float(fmt)
        except:
          return fmt
    # print(num,(bases.get(base,10)**units.get(unit,1)))
    # print(num)
    num *= (bases.get(base,10)**units.get(unit,1))
    if int(num) == num:
      num = int(num)
    return num

def indexer(array,sorter,value):
  i = np.where(sorter==value)[0]
  return array[i]


file='membench_4'
cpuspeed=3e9

mbdata = np.genfromtxt('%s.out'%file,usecols=(1,3,5))

```

```

maxunit=10
units=dict(zip(range(0,maxunit*4,maxunit),['','K','M','G']))
sizes = {'%d%s%s'%(b**(i),units[d],u):b**(i+d) for b,u in zip([2],['B']) for d in units for i in range(maxunit)}


xtlabels=[*['4B','16B','64B','256B','1KB','4KB','16KB','64KB','256KB','1MB'],
          *['2MB','4MB','8MB','16MB','32MB','64MB','128MB','256MB','512MB','2GB']]
xtvals = [sizes.get(l,fmt_sizeof(l)) for l in xtlabels]





# k,l,m = '1GB','1MB','512B'
# print((sizes.get(k,fmt_sizeof(k)),indexer(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))),indexer(mbdata[:,1],mbdata[:,0],sizes.get(l,fmt_sizeof(l))),sizes.get(m,fmt_sizeof(m)))[0]))

lims={
       'all':[(xtlabels[0],xtlabels[-1]),(-2,16)],
       'lowerleft':[('4B','8KB'),(-2,1)],
       'middleupper':[('16B','64KB'),(6,12)],
       'middle':[('64B','64MB'),(1,4)],
       }

heights={k:{l:-1 for l in ['h','v']} for k in lims}
heights['all']['v'] =15
heights['lowerleft']['v'] = -1
heights['middle']['v'] = 2.5
heights['middleupper']['v'] = 11
lines={
        'all':{'v': {
                     # **{(sizes.get(k,fmt_sizeof(k))/2,height):r'$\tilde{L}_{1,2,3}\\\textrm{%s}$'%k for i,k in enumerate(['64B'])},
                     # **{(sizes.get(k,fmt_sizeof(k)),height):r'${L}_{%d}\\\textrm{%s}$'%(i+1,k) for i,k in enumerate(['128B'])},
                     # # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{%d}$'%(i+1) for i,k in enumerate(['128B','4KB','16KB'])},
                     # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{%d}\\\textrm{%s}$'%(i+1,k) for i,k in enumerate(['32KB','1MB','24.75MB'])},

                     **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']-1):r'$\tilde{L}_{1,2,3}$' for i,k in enumerate(['64B'])},
                     **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']):r'${L}_{%d}$'%(i+1) for i,k in enumerate(['64B','512B','4KB'])},
                     # **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']):r'$\tilde{L}_{%d}$'%(i+1) for i,k in enumerate(['128B','4KB','16KB'])},
                     **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']-1):r'$\tilde{L}_{%d}$'%(i+1) for i,k in enumerate(['32KB','1MB','25MB'])},
```

```
                    **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']):r'$L_{%d}$'%(i+1) for i,k in enumerate(['16KB','256KB','8MB'])},
                    },

            'h': {
                  # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['1GB'],['32KB']))},
                  **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) if i<3 else r'$T_{\textrm{mem}}$' for i,(k,l) in enumerate(zip(['1GB','1GB','1GB','1GB'],['32KB','1MB','256MB','1GB']))},
                  # **{(sizes.get(k,fmt_sizeof(k)),indexer(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))),indexer(mbdata[:,1],mbdata[:,0],sizes.get(l,fmt_sizeof(l))),sizes.get(m,fmt_sizeof(m)))[0]):r'$T_{%d}$'%(i+2) for i,(k,l,m) in enumerate(zip(['1GB'],['1MB'],['512B']))},
                  },



            },
      'lowerleft':{'v': {
                        # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{1,2,3}\\\textrm{%s}$'%k for i,k in enumerate(['64B'])},
                        # **{(sizes.get(k,fmt_sizeof(k)),height):r'${L}_{%d}\\\textrm{%s}$'%(i+1,k) for i,k in enumerate(['128B'])},

                        **{(sizes.get(k,fmt_sizeof(k)),heights['lowerleft']['v']-0.5):r'$\tilde{L}_{1,2,3}$' for i,k in enumerate(['64B'])},
                        **{(sizes.get(k,fmt_sizeof(k)),heights['lowerleft']['v']):r'${L}_{%d}$'%(i+1) for i,k in enumerate(['64B','512B','4KB'])},


                      },
                  'h': {
                        # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['256B'],['32KB']))},
                  **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['4KB','4KB'],['32KB','1MB']))},
                      },
                  },
      'middle':{'v': {
                        # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{1,2,3}\\\textrm{%s}$'%k for i,k in enumerate(['64B'])},
                        # **{(sizes.get(k,fmt_sizeof(k)),height):r'${L}_{%d}\\\textrm{%s}$'%(i+1,k) for i,k in enumerate(['128B'])},

```

```
                                **{(sizes.get(k,fmt_sizeof(k)),heights['middle']['v']-0.5):r'$\tilde{L}_{1,2,3}$' for i,k in enumerate(['64B'])},
                                **{(sizes.get(k,fmt_sizeof(k)),heights['middle']['v']):r'${L}_{%d}$'%(i+1) for i,k in enumerate(['64B','512B','4KB'])},
                                **{(sizes.get(k,fmt_sizeof(k)),heights['middle']['v']-1):r'$\tilde{L}_{%d}$'%(i+1) for i,k in enumerate(['32KB','1MB','25MB'])},
                                 **{(sizes.get(k,fmt_sizeof(k)),heights['middle']['v']):r'$L_{%d}$'%(i+1) for i,k in enumerate(['16KB','256KB','8MB'])},
                            },
                    'h': {
                            # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['256B'],['32KB']))},
                    **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['16MB','16MB'],['32KB','1MB']))},
                            },
                    },
        'middleupper':{'v': {
                                # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{1,2,3}\\\textrm{%s}$'%k for i,k in enumerate(['64B'])},
                                # **{(sizes.get(k,fmt_sizeof(k)),height):r'${L}_{%d}\\\textrm{%s}$'%(i+1,k) for i,k in enumerate(['128B'])},

                                **{(sizes.get(k,fmt_sizeof(k)),heights['middleupper']['v']-0.5):r'$\tilde{L}_{1,2,3}$' for i,k in enumerate(['64B'])},
                                **{(sizes.get(k,fmt_sizeof(k)),heights['middleupper']['v']):r'${L}_{%d}$'%(i+1) for i,k in enumerate(['64B','512B','4KB'])},
                                **{(sizes.get(k,fmt_sizeof(k)),heights['middleupper']['v']-1):r'$\tilde{L}_{%d}$'%(i+1) for i,k in enumerate(['32KB','1MB','25MB'])},
                                 **{(sizes.get(k,fmt_sizeof(k)),heights['middleupper']['v']):r'$L_{%d}$'%(i+1) for i,k in enumerate(['16KB','256KB','8MB'])},
                            },
                    'h': {
                            # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['256B'],['32KB']))},
                    **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbdata[:,2],mbdata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in enumerate(zip(['16MB','16MB','32KB'],['32KB','1MB','256MB']))},
                            },
                    },
        # 'middle':{'v': {**{(sizes.get(k,fmt_sizeof(k)),-4.5):'L%d\\LineSize:\\%s'%(i+1,k) for i,k in enumerate(['4KB','16KB'])}}}
    }


```

```python


print(lines)


cmap = 'tab20b'
lengths=np.array(list(sorted(set(mbdata[:,0]))))

kwargs={}
kwargs['marker']  = ['.','*','^','s','d','+','']

kwargs['color'] = {k:v for k,v in zip(lengths,plt.get_cmap(cmap)(np.linspace(0, 1, len(lengths))).tolist())}
kwargs['label'] = {k:'L=%s'%(sizeof_fmt(k)) for k in lengths }

kwargs['marker'] = {k: kwargs['marker'][i%len(kwargs['marker'])] for i,k in enumerate(lengths)}
kwargs['inds'] = {k: np.where(mbdata[:,0]==k)[0] for k in lengths}



kwargs['lines'] = {k: {'linestyle':s,'color':c,'label':l,'alpha':a,'zorder':z} for k,l,c,s,a,z in zip(
                                                                                                   ['tilde{L}_{1,2,3}','tilde{L}','{L}_','L_','T_','Mem',None],
                                                                                                   ['True Line Size','True Cache Size','Line Size','Cache Size','Access Time','Access Time',''],
                                                                                                   ['k','b','k','b','g','g',None],
                                                                                                   ['--','--','-','-','-','-',None],
                                                                                                   [0.6,0.6,0.8,0.8,0.8,0.8,None],
                                                                                                   [-1,-1,10,10,10,10,None],
                                                                                                   )}

for lim in lims:


  fig,ax = plt.subplots()

  # Plots

  #for i in range(9,27):
```

```
for L in lengths:
  #ax.plot(mbdata[istt:istt+i,1],mbdata[istt:istt+i,2],
  ax.plot(mbdata[kwargs['inds'][L],1],mbdata[kwargs['inds'][L],2],**{k:kwargs[k][L] for k in ['color','marker','label']})
  #istt=istt+i+1

ax.set_ylabel('Time (nsec)')
ax.set_xlabel('Stride (bytes)')
ax.set_xscale('log',base=2)
ax.set_xticks(xtvals)
ax.set_xticklabels(xtlabels,rotation=45)
ax.set_ylim(*[l for l in lims[lim][1]])
ax.set_xlim(*[sizes.get(l,fmt_sizeof(l)) for l in lims[lim][0]])

# Lines
plotlines={k:lambda line,k=k,i=i,**kwargs: getattr(ax,'ax%sline'%k)(line[i],**kwargs) for i,k in enumerate(['v','h'])}
annotatelines={k:lambda line,text,k=k,i=i,**kwargs: getattr(plt,'annotate')(text=text,xy=line,**kwargs) for i,k in enumerate(['v','h'])}
for k in lines.get(lim,[]):
  plotline=plotlines[k]
  annotateline=annotatelines[k]
  for line in lines[lim][k]:
    text = lines[lim][k][line]
    _kwargs = kwargs['lines'][None]
    for x in kwargs['lines']:
      if str(x) in text:
        _kwargs = kwargs['lines'][x]
        break
    textline=list(line)
    if 0 and (_kwargs['color'] == 'k' and _kwargs['linestyle'] == '-'):
      textline[0] /=8
    elif (_kwargs['color'] == 'g') and (lim=='all'):
      textline[1] += 0.25
    elif (_kwargs['color'] == 'g') and (lim=='lowerleft'):
      textline[1] += 0.1
    elif (_kwargs['color'] == 'g') and (lim=='middle'):
      textline[0] *= 1.5
      textline[1] += 0.06
    elif (_kwargs['color'] == 'g') and (lim=='middleupper'):
      textline[0] *= 1.2
      textline[1] += 0.1
    elif lim not in ['lowerleft']:
      textline[0] *= 1.15
    else:
      textline[0] *= 1.15
    # if lim == 'lowerleft':
    #   textline[1] = -0.5

    plotline(line,**_kwargs)
    annotateline(textline,text)

handles,labels = ax.get_legend_handles_labels()
handles,labels = [h for i,(h,l) in enumerate(zip(handles,labels)) if l not in labels[:i]],[l for i,(h,l) in enumerate(zip(handles,labels)) if l not in labels[:i]]
fig.legend(handles=handles,labels=labels,bbox_to_anchor=(0.7,0.5),loc='center left',ncol=2,prop = {"size": 6},)
```

```
fig.subplots_adjust(right=0.7,bottom=0.15)
fig.savefig('../../figures/%s_%s.pdf'%('_'.join(file.split('_')[:-1]),lim))
```

Code 6: CPU Info script.

```
#!/bin/bash

prog="/proc/cpuinfo"
field="cpu MHz"
pattern="s%.*: \([^ ]*\).*$%\1%"
trials=5000

file=tmp1234.tmp
rm -f ${file}


for i in $(seq 1 ${trials})
do
        #echo Trial: $i
        grep "${field}" ${prog} | sed "${pattern}" >> ${file}
        sleep 0.00000001
done

N=$(wc -l ${file} | sed "s:\([^ ]*\).*:\1:")
avg=$(paste -sd+ ${file} | bc)
avg=$(echo "scale=3 ; $avg / $N" | bc)

echo Avg\(N=${N},Trials=${trials}\)  :  ${avg}

rm ${file}
```

Code 7: membench job script.

```
#!/bin/bash

#SBATCH --account=ners570f20_class
#SBATCH --job-name=NERS570_Lab8
#SBATCH --partition=standard
#SBATCH --mail-user=mduschen@umich.edu
#SBATCH --mail-type=END
#SBATCH --nodes=1
#SBATCH --mem-per-cpu=8000m
#SBATCH        --time=01:00:00
#SBATCH        --ntasks-per-node=3


./run.sh
```

Code 8: Matrix-Matrix multiply

```
#include <cstdio>
#include <iostream>



void dgemm( char transa, char transb,
              unsigned int m, unsigned int n, unsigned int k,
              double alpha, const double * const * a,
              const double * const * b,
              double beta, double **c)
{
    for(unsigned int i=0; i<n; i++){
            for(unsigned int j=0; j<m; j++){
                c[i][j] *= beta;
                }
        };
    for(unsigned int l=0; l<k; l++){
               for(unsigned int j=0; j<m; j++){
                    for(unsigned int i=0; i<n; i++){
                     c[i][j] += alpha*a[i][l]*b[l][j];
                        }
                }
        };


}
```

Code 9: Matrix-Vector multiply

```
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define ALIGN 16
#define SIMD_BLOCK (ALIGN/sizeof(double))

#define base 0


// Coords (i_{0},...,i_{dim-1}) from linear index (Naive operations)
void position(int z, const int dim, const int * shape,int * indices){
	int i,j,L;
	for (i=0;i<dim;i++){
		L = 1;
		for (j=i+1;j<dim;j++){
			L *= shape[j];
		};
		indices[i] = (z/L)%shape[i];
	};

	return;
};


// Linear order of indices (i_{0},...,i_{dim-1}),
//  with shape(N_{0},...,N_{dim-1})
int lorder(const int dim,const int * indices,const int * shape){

	int z = 0;
	int w = 1;
	int i,j;
	for (i=0;i<dim;i++){
		w = 1;
		for (j=i+1;j<dim;j++){
			w *= shape[j];
		}
		z += indices[i]*w;
	}
	return z;

};


// z order of indices (i_{0},...,i_{dim-1}) (Bitwise operations)
int zorder(const int dim, const int * indices,const int *shape){
	const int zsize=8*sizeof(int);
	int i,j;
	int z = 0;
	int x;
	for (i=0;i<zsize;i++){
		for(j=0;j<dim;j++){
```

```
                        x = (indices[dim-1-j] & (1 << i));
                        z |= (x << (i+j));
                        };
                };
        return z;
};

// z order of indices (i_{0},...,i_{dim-1}) for (2d) array (Bitwise operations)
void z_order(const int N, const int dim, double * arr){
        int i,j,z;
        int indices[dim];
        int shape[dim];
        int size = 1;
        for (i=0;i<dim;i++){
                shape[i] = N;
                size *= N;
        };
        double tmp[size];
        for(i=0;i<size;i++){
                position(i,dim,shape,indices);
                // printf("(%d,%d) -> %d\n",indices[0],indices[1],i);
                z = zorder(dim,indices,shape)+base;
                tmp[i] = arr[z];
        };
        for(i=0;i<size;i++){
                arr[i] = tmp[i];
        };
        return;
};

int size(const int dim,const int * shape){
        int size=1;
        for(unsigned int i=0;i<dim;i++){
                size *= shape[i];
        };
        return size;
};

void arr_init(const int dim,const int * shape, double * arr,const char * type){
        int N=size(dim,shape);

        for(unsigned int i=0;i<N;i++){
                if (strcmp(type,"RANDOM")==0){
                        arr[i] = rand();
                }
                else if (strcmp(type,"LINEAR")==0){
                        arr[i] = i;
                }
                else if (strcmp(type,"ONES")==0){
                        arr[i] = 1;
                }
                else if (strcmp(type,"ZEROS")==0){
                        arr[i] = 0;
                }
                else {
                        arr[i] = 0;
                };
```

```
	};
	return;
};

void arr_pad(const int n,const int m,const int N, const int M, const double * A, double * A_,double padding){
	unsigned int i,j,z,w;
	for(i=0;i<N;i++){
		for(j=0;j<M;j++){
			if ((i<n) && (j<m)){
				z = i*M+j;
				w=i*m+j;
				A_[z] = A[w];
			}
			else{
				z = i*M+j;
				A_[z] = padding;
			};
		};
	};
};


// Print out array of shape (N,M)
void printa(const int N, const int M, double const * arr, char const * label){
	const int dim = 2;
	int indices[dim];
	int shape[] = {N,M};
	int i,j;
	int a,z;
	// int label_size = strlen(label);
	// printf("label size = %d",label_size);
	// if(label_size>1){
	//		char spacing[label_size-1];
	//		for (unsigned int i=0;i<label_size-1;i++){
	//			spacing[i]=" ";
	//		};
	// }
	// else{
	//		char spacing[]="";
	// };
	char spacing[]="";
	printf("%s = [",label);
	for (i = 0; i < N; i++) {
		  for (j = 0; j < M; j++){
			indices[0] = i;
			indices[1] = j;
			z = lorder(dim,indices,shape);
			a = arr[z];
			if (j==0 && i>0){
				printf("%s",spacing);
				printf("      ");
			}
			else if (j==0 && i == 0){
				printf(" ");
			};
		    printf("%d",a);
```

```
                        if (j== (M-1) && i==(N-1)){
                                        printf(" ]\n");
                                }
                                else if (j ==(M-1)) {
                                        printf("\n");
                                }
                        else if (a < 10){
                                printf("    ");
                        }
                        else if (a < 100){
                                printf("   ");
                        }
                        else if (a < 1000){
                                printf("  ");
                        }
                        else{
                                printf(" ");
                        };
                };
        };
        printf("\n");
        return;
};

void matvec(const int n,const int m,const double * A,const double *x, double *y){
        for(unsigned int i=0;i<n;i++){
                for(unsigned int j=0;j<m;j++){
                        // std::cout << i << " " << j << " :: " << y[i] << " = " << A[i*m+j] <<
                          " * " << x[j] << std::endl;
                        y[i] += A[i*m +j]*x[j];
                }
        };
};


void _matvec_avx(const double* A,const double* x,double* y){
    asm volatile ("# avx code begin");  // looking at assembly with gcc -S
    __m256d xrow = _mm256_loadu_pd(x);

    __m256d a = _mm256_mul_pd(_mm256_loadu_pd(A), xrow);
    __m256d b = _mm256_mul_pd(_mm256_loadu_pd(A+4), xrow);
    __m256d c = _mm256_mul_pd(_mm256_loadu_pd(A+8), xrow);
    __m256d d = _mm256_mul_pd(_mm256_loadu_pd(A+12), xrow);

    // our task now is to get {sum(a), sum(b), sum(c), sum(d)}
    // This is tricky because there is no hadd instruction for avx

    // {a[0]+a[1], b[0]+b[1], a[2]+a[3], b[2]+b[3]}
    __m256d sumab = _mm256_hadd_pd(a, b);

    // {c[0]+c[1], d[0]+d[1], c[2]+c[3], d[2]+d[3]}
    __m256d sumcd = _mm256_hadd_pd(c, d);

    // {a[0]+a[1], b[0]+b[1], c[2]+c[3], d[2]+d[3]}
    __m256d blend = _mm256_blend_pd(sumab, sumcd, 0b1100);

    // {a[2]+a[3], b[2]+b[3], c[0]+c[1], d[0]+d[1]}
```

```
    __m256d perm = _mm256_permute2f128_pd(sumab, sumcd, 0x21);

    // {sum(a), sum(b), sum(c), sum(d)}
    __m256d Ax = _mm256_add_pd(perm, blend);
    // printf("yinit = %f",*y);
    _mm_storel_pd(y, _mm256_castpd256_pd128(Ax));
    // printf("yfinal = %f",*y);
    // _mm_storel_pd(y, Ax);
    asm volatile ("# avx code end");

};



void matvec_avx(const int n,const int m, const double* A,const double* x,double* y){
        // Break up multiplication into 4x4 blocks for avx instructions
        //Pad A,x,y with zeros to make multiple of 4 for SIMD
        int N = n,M=m;
        if (n%SIMD_BLOCK != 0){
                N = SIMD_BLOCK*(n/SIMD_BLOCK+1);
        };
        if (m%SIMD_BLOCK != 0){
                M = SIMD_BLOCK*(m/SIMD_BLOCK+1);
        };
        if (N<M){
                N = M;
        }
        else if(N>M){
                M = N;
        };
        const int L = N*M;
        const int S = N/SIMD_BLOCK;
        double A_[L],x_[M],y_[N];
        double y__;

        arr_pad(n,m,N,M,A,A_,0.0);
        arr_pad(m,1,M,1,x,x_,0.0);
        arr_pad(n,1,N,1,y,y_,0.0);

        // Use z-index ordering for blocks of 4
        // printa(N,M,A_,"A");
        // z_order(N,2,A_);
        // printa(N,M,A,"A");

        printf("Padded Arrays\n");
        printa(N,M,A_,"A");
        printa(M,1,x_,"x");
        printa(N,1,y_,"y");


        _matvec_avx(A_,x_,y_);


        // unsigned int i,j,z;
        // for(z=0;z<L;z+=SIMD_BLOCK){
        //          i=L/M;
        //          j=L%M;
```

```
	//		printf("z=%d,i=%d,j=%d\n",z,i,j);
	//		_matvec_avx(A_+z,x_+j,&y__);
	//		y[i] += y__;
	// };
	return;
};




int main(int argc, char** argv){
	//Shape of matrix
	const int dim = 2;
	int shape[dim];
	for (int i=dim-1;i>-1;i--){
		shape[i]=0;
		if (argc > (i+1)){
			for(unsigned int j=0;j<dim;j++){
				if (j<=i){
					shape[j] = atoi(argv[j+1]);
				}
				else{
					shape[j] = atoi(argv[i+1]);
				};
			};
			break;
		};
	};

	const int N=size(dim,shape);

	double A[N];
	arr_init(dim,shape,A,"LINEAR");



	int dim_x=1,dim_y=1;
	int shape_x[1]={shape[1]};
	int shape_y[1]={shape[0]};

	double x[shape[1]],y[shape[0]];
	arr_init(dim_x,shape_x,x,"ONES");
	arr_init(dim_y,shape_y,y,"ZEROS");


	printf("Init Arrays\n");
	printa(shape[0],shape[1],A,"A");
	printa(shape[1],1,x,"x");
	printa(shape[0],1,y,"y");

	// for(unsigned int i=0;i<m;i++){
	//		x[i] = i;
	// };
	// for(unsigned int i=0;i<n;i++){
	//		y[i] = 0.0;
	// };
	printf("Naive Mat-Vec Arrays\n");
```

```
        matvec(shape[0],shape[1],A,x,y);

        printf("Naive Result\n");
        printa(shape[0],1,y,"y");


        printf("ASM Mat-Vec Arrays\n");
        arr_init(dim_y,shape_y,y,"ZEROS");
        matvec_avx(shape[0],shape[1],A,x,y);

        printf("ASM Result\n");
        printa(shape[0],1,y,"y");

        // for(unsigned int i=0;i<n;i++){
        //         std::cout << i << " " << y[i] << std::endl;
        // };


        return 0;
};
```