

1 Membench

The *membench* programs are run to obtain the following plots, from Code. 4, 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. 5 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.

```

1 Cache Info
2 Cache L1 Size: 32 kB
3 Cache L1 Line Size: 64 B
4
5 Cache L2 Size: 1024 kB
6 Cache L2 Line Size: 64 B
7
8 Cache L3 Size: 25344 kB
9 Cache L3 Line Size: 64 B
10
```

```

11 Cache L4 Size: 0 kB
12 Cache L4 Line Size: 0 B
13
14
15
16
17 CPU Info
18 CPU Cores: 36
19 CPU Speed: 2999.531 MHz

```

1.1 Processor Values

On the Greatlakes compute nodes, there are $36 \times$ 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 $\leq 32KB$, 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 $\leq 32KB$, 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 $\leq 512B$, 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 64KB 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 64KB 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.

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

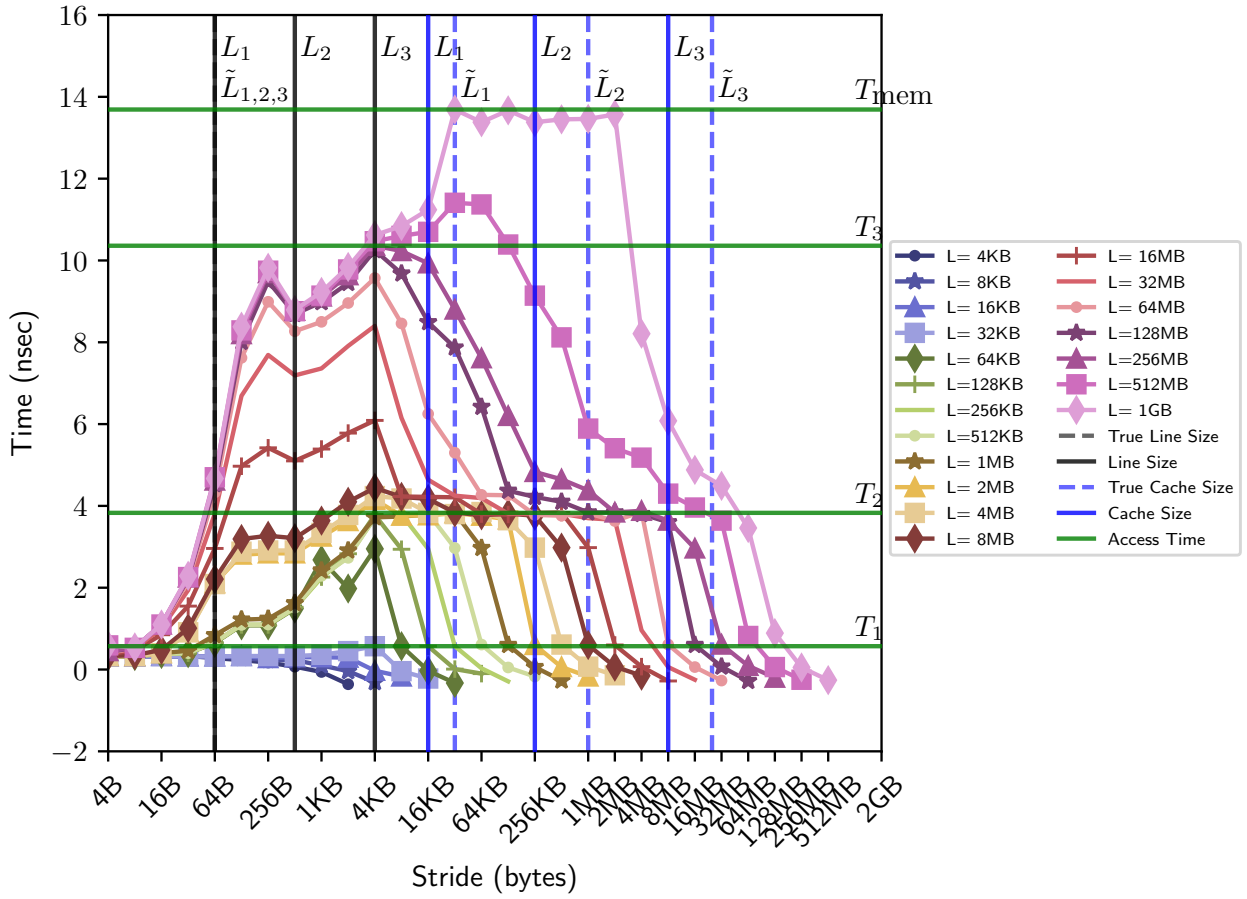


Figure 1: Memory access times for various array sizes, and stride lengths from 4B to 512MB.

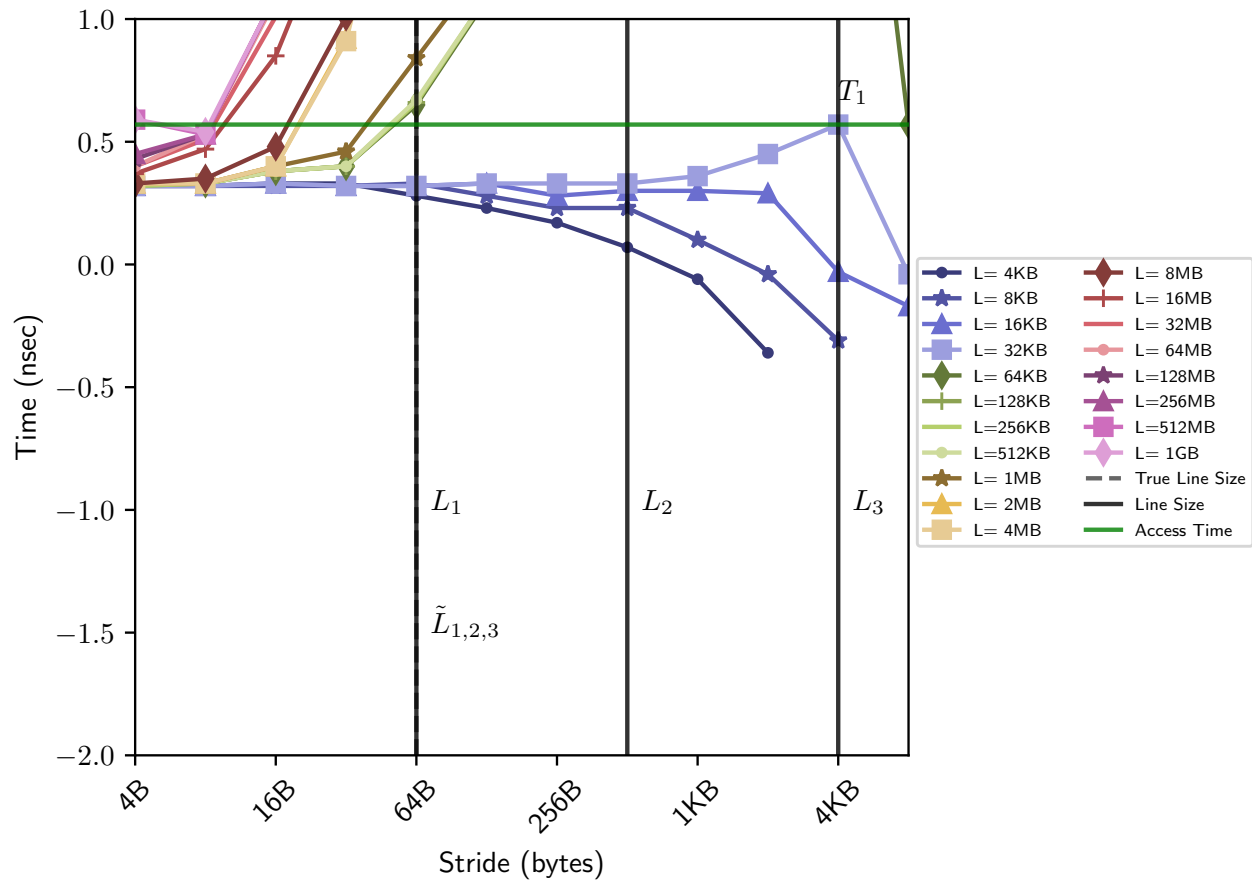


Figure 2: Memory access times for various array sizes, and stride lengths from 4B to 4KB.

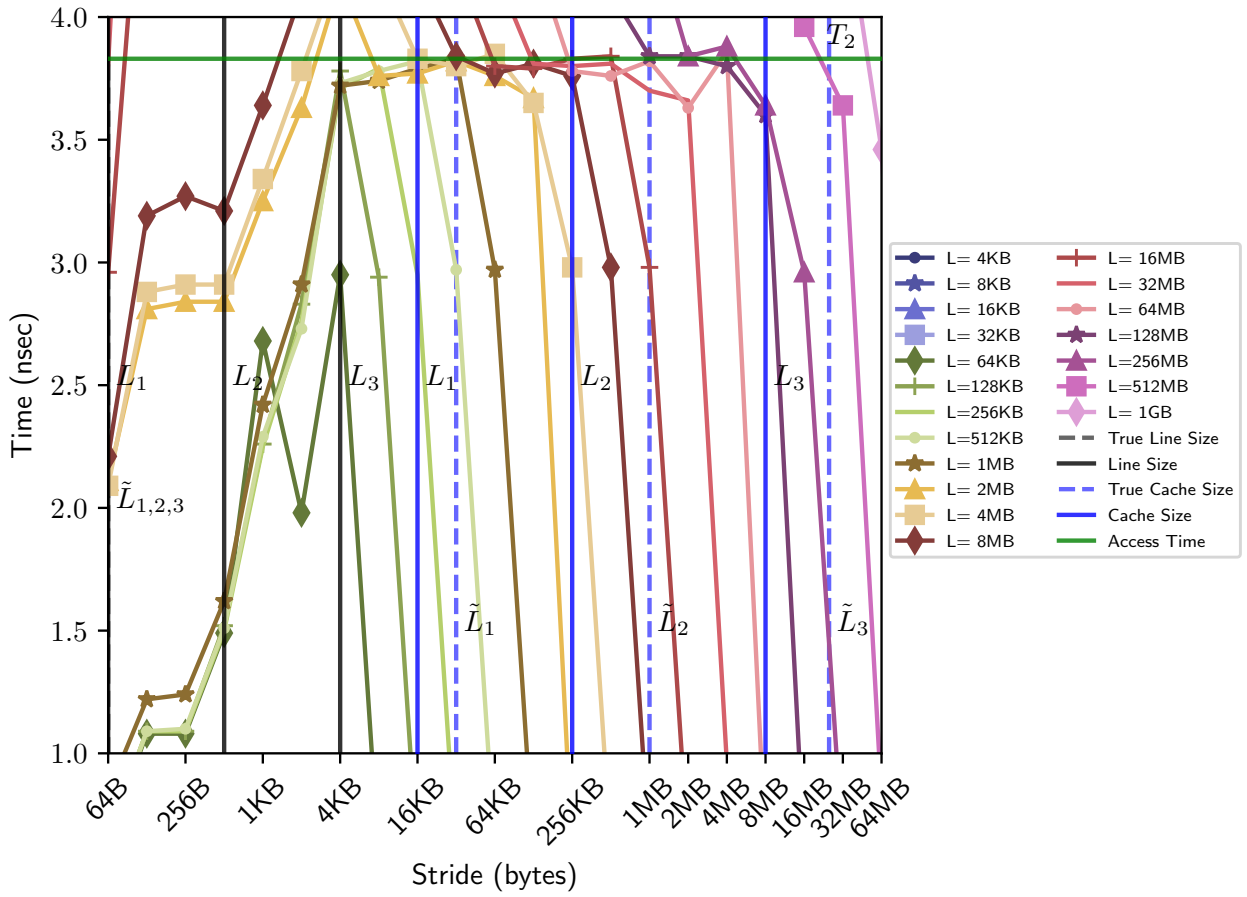


Figure 3: Memory access times for various array sizes, and stride lengths from 16B to 64MB.

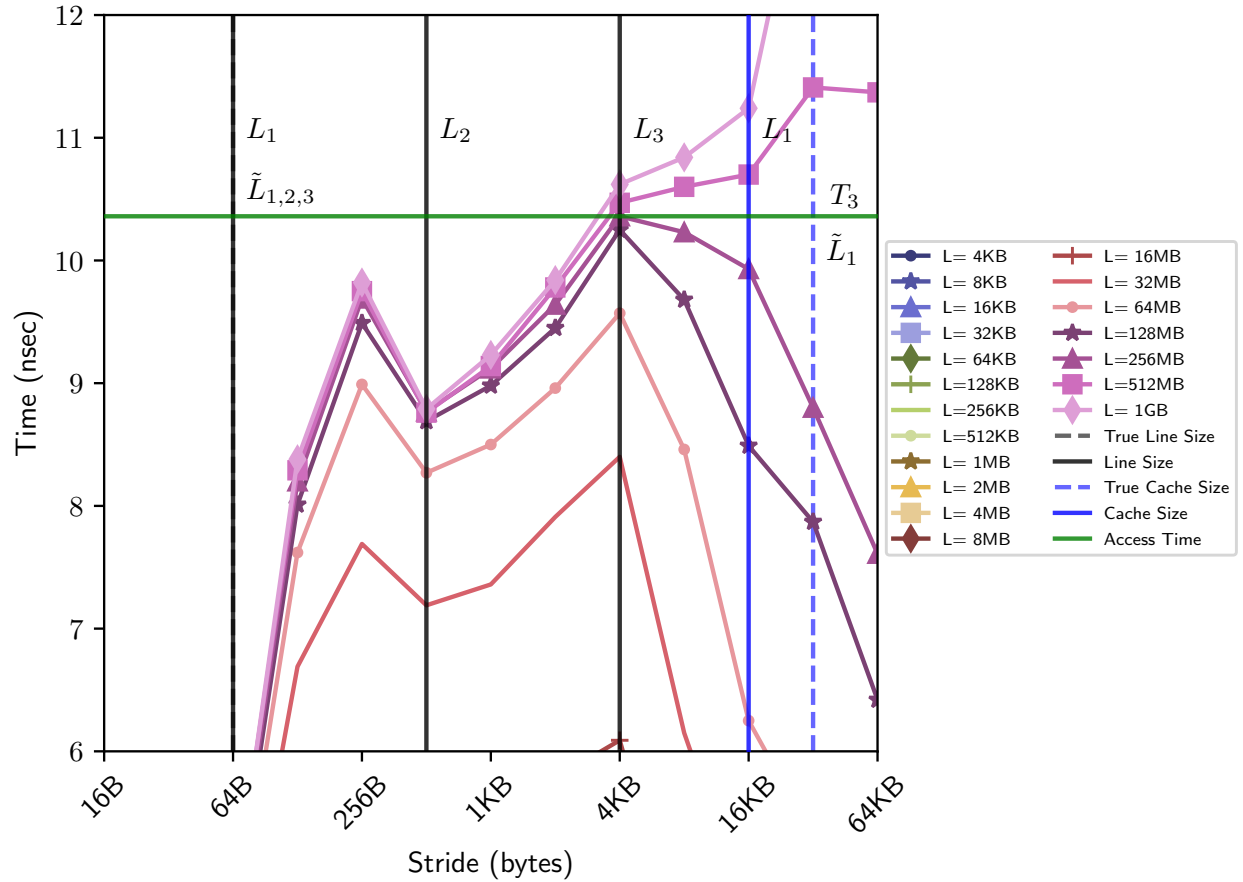


Figure 4: Memory access times for various array sizes, and stride lengths from 16B to 64KB.

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. 7 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 *vmadd132sd*.

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 *vmadd132sd*) are in the assembly instructions.

Code 2: Matrix-matrix multiplication loop assembly code

```

81      testl    %r8d, %r8d
82      je      .L29
83      movq     16(%rbp), %r12
84      leal     -1(%r8), %eax
85      leaq     8(,%r14,8), %rbx
86      xorl     %r8d, %r8d
87      leaq     8(%r12,%rax,8), %r13
88      .p2align 4,,10
89      .p2align 3
90  .L10:
91      movq     (%r12), %r14
92      xorl     %esi, %esi
93      .p2align 4,,10
94      .p2align 3
95  .L9:
96      leaq     (%r14,%rsi), %rdi
97      xorl     %eax, %eax

```



```

98      jmp      .L8
99      .p2align 4,,10
100     .p2align 3
101     .L13:
102     movq      %rdx, %rax
103     .L8:
104     movq      (%r11,%rax,8), %rdx
105     movq      (%r9,%rax,8), %rcx
106     vmulsd    (%rdi), %xmm0, %xmm1
107     addq      %rsi, %rdx
108     vmovsd    (%rdx), %xmm4
109     vfmadd132sd (%rcx,%r8), %xmm4, %xmm1
110     vmovsd    %xmm1, (%rdx)
111     leaq      1(%rax), %rdx
112     cmpq      %rax, %r10
113     jne       .L13
114     addq      $8, %rsi
115     cmpq      %rsi, %rbx
116     jne       .L9
117     addq      $8, %r12
118     addq      $8, %r8
119     cmpq      %r12, %r13
120     jne       .L10

```

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

```

1      .file      "dgemm.cpp"
2      .text
3      .p2align 4,,15
4      .globl     _Z5dgemmccjjdPKPKdS2_dPPd
5      .type      _Z5dgemmccjjdPKPKdS2_dPPd, @function
6      _Z5dgemmccjjdPKPKdS2_dPPd:
7      .LFB1538:
8      .cfi_startproc
9      pushq      %rbp
10     .cfi_def_cfa_offset 16
11     .cfi_offset 6, -16
12     movq      %rsp, %rbp
13     .cfi_def_cfa_register 6
14     pushq      %r14
15     movq      24(%rbp), %r11
16     pushq      %r13
17     pushq      %r12
18     pushq      %rbx
19     .cfi_offset 14, -24
20     .cfi_offset 13, -32
21     .cfi_offset 12, -40
22     .cfi_offset 3, -48
23     testl     %ecx, %ecx
24     je        .L27
25     testl     %edx, %edx
26     je        .L27
27     movl      %edx, %r12d
28     leal      -1(%rcx), %eax
29     movl      %edx, %ebx
30     movq      %r11, %rdi

```

```

31     shrl     $2, %r12d
32     movq     %rax, %r10
33     leaq     8(%r11,%rax,8), %r13
34     andl     $-4, %ebx
35     leal     -1(%rdx), %r14d
36     salq     $5, %r12
37     vbroadcastsd    %xmm1, %ymm3
38     .p2align 4,,10
39     .p2align 3
40 .L7:
41     movq     (%rdi), %rsi
42     cmpl     $2, %r14d
43     jbe      .L12
44     movq     %rsi, %rax
45     leaq     (%r12,%rsi), %rcx
46     .p2align 4,,10
47     .p2align 3
48 .L4:
49     vmovupd   (%rax), %xmm5
50     vinsertf128    $0x1, 16(%rax), %ymm5, %ymm2
51     addq     $32, %rax
52     vmulpd    %ymm3, %ymm2, %ymm2
53     vmovups   %xmm2, -32(%rax)
54     vextractf128    $0x1, %ymm2, -16(%rax)
55     cmpq     %rax, %rcx
56     jne      .L4
57     movl     %ebx, %eax
58     cmpl     %ebx, %edx
59     je       .L5
60 .L3:
61     movl     %eax, %ecx
62     leaq     (%rsi,%rcx,8), %rcx
63     vmulsd    (%rcx), %xmm1, %xmm2
64     vmovsd    %xmm2, (%rcx)
65     leal     1(%rax), %ecx
66     cmpl     %ecx, %edx
67     jbe      .L5
68     leaq     (%rsi,%rcx,8), %rcx
69     addl     $2, %eax
70     vmulsd    (%rcx), %xmm1, %xmm2
71     vmovsd    %xmm2, (%rcx)
72     cmpl     %eax, %edx
73     jbe      .L5
74     leaq     (%rsi,%rax,8), %rax
75     vmulsd    (%rax), %xmm1, %xmm2
76     vmovsd    %xmm2, (%rax)
77 .L5:
78     addq     $8, %rdi
79     cmpq     %r13, %rdi
80     jne      .L7
81     testl    %r8d, %r8d
82     je       .L29
83     movq     16(%rbp), %r12
84     leal     -1(%r8), %eax
85     leaq     8(,%r14,8), %rbx
86     xorl     %r8d, %r8d
87     leaq     8(%r12,%rax,8), %r13

```

```

88     .p2align 4,,10
89     .p2align 3
90 .L10:
91     movq    (%r12), %r14
92     xorl    %esi, %esi
93     .p2align 4,,10
94     .p2align 3
95 .L9:
96     leaq    (%r14,%rsi), %rdi
97     xorl    %eax, %eax
98     jmp     .L8
99     .p2align 4,,10
100    .p2align 3
101 .L13:
102    movq    %rdx, %rax
103 .L8:
104    movq    (%r11,%rax,8), %rdx
105    movq    (%r9,%rax,8), %rcx
106    vmulsd   (%rdi), %xmm0, %xmm1
107    addq    %rsi, %rdx
108    vmovsd   (%rdx), %xmm4
109    vfmadd132sd (%rcx,%r8), %xmm4, %xmm1
110    vmovsd   %xmm1, (%rdx)
111    leaq    1(%rax), %rdx
112    cmpq    %rax, %r10
113    jne     .L13
114    addq    $8, %rsi
115    cmpq    %rsi, %rbx
116    jne     .L9
117    addq    $8, %r12
118    addq    $8, %r8
119    cmpq    %r12, %r13
120    jne     .L10
121 .L29:
122    vzeroupper
123 .L27:
124    popq    %rbx
125    popq    %r12
126    popq    %r13
127    popq    %r14
128    popq    %rbp
129    .cfi_restore_state
130    .cfi_def_cfa 7, 8
131    ret
132 .L12:
133    .cfi_restore_state
134    xorl    %eax, %eax
135    jmp     .L3
136    .cfi_endproc
137 .LFE1538:
138    .size    _Z5dgemmccjjdPKPKdS2_dPPd, .-_Z5dgemmccjjdPKPKdS2_dPPd
139    .section .text.startup, "ax",@progbits
140    .p2align 4,,15
141    .type    _GLOBAL__sub_I__Z5dgemmccjjdPKPKdS2_dPPd, @function
142 _GLOBAL__sub_I__Z5dgemmccjjdPKPKdS2_dPPd:
143 .LFB2019:
144    .cfi_startproc

```

```

145     subq    $8, %rsp
146     .cfi_def_cfa_offset 16
147     movl    $_ZStL8__ioinit, %edi
148     call    _ZNSt8ios_base4InitC1Ev
149     movl    $__dso_handle, %edx
150     movl    $_ZStL8__ioinit, %esi
151     movl    _ZNSt8ios_base4InitD1Ev, %edi
152     addq    $8, %rsp
153     .cfi_def_cfa_offset 8
154     jmp     __cxa_atexit
155     .cfi_endproc
156 .LFE2019:
157     .size    _GLOBAL__sub_I__Z5dgemmccjjdPKPKdS2_dPPd,
158             .-_GLOBAL__sub_I__Z5dgemmccjjdPKPKdS2_dPPd
159     .section .init_array,"aw"
160     .align 8
161     .quad    _GLOBAL__sub_I__Z5dgemmccjjdPKPKdS2_dPPd
162     .local   _ZStL8__ioinit
163     .comm    _ZStL8__ioinit,1,1
164     .hidden  __dso_handle
165     .ident   "GCC: (GNU) 8.2.0"
166     .section .note.GNU-stack,"",@progbits

```

3 Appendix

Code 4: membench plotting script.

```

1  #!/usr/bin/env python
2  import matplotlib
3  import matplotlib.pyplot as plt
4  import numpy as np
5
6  matplotlib.rcParams['text.usetex'] = True
7  # matplotlib.rcParams['text.latex.preamble'] = [r'\usepackage{ragged2e'}]
8
9  # Routine modified from:
10     https://stackoverflow.com/questions/1094841/reusable-library-to-get-human-readable-version-of-file-size
11 def sizeof_fmt(num, suffix='B'):
12     for unit in ['', 'K', 'M', 'G', 'T']:
13         if abs(num) < 1024.0:
14             return '%3.0f%s%s' % (num, unit, suffix)
15         num /= 1024.0
16     return '%.1f%s%s' % (num, 'T', suffix)
17
18 def fmt_sizeof(fmt):
19     bases={'B':2, '':10}
20     units={k:v for k,v in zip(['', 'K', 'M', 'G', 'T'], [1,10,20,30,40])}
21     try:
22         base=fmt[-1]
23         unit=fmt[-2]
24         num = float(fmt[:-2])
25     except ValueError:
26         try:
27             base=fmt[-1]
28             unit=''
29             num = float(fmt[:-1])
30         except:
31             try:
32                 base=''
33                 unit=''
34                 num = float(fmt)
35             except:
36                 return fmt
37     # print(num, (bases.get(base,10)**units.get(unit,1)))
38     # print(num)
39     num *= (bases.get(base,10)**units.get(unit,1))
40     if int(num) == num:
41         num = int(num)
42     return num
43
44 def indexer(array, sorter, value):
45     i = np.where(sorter==value)[0]
46     return array[i]
47
48 file='membench_4'
49 cpuspeed=3e9
50
51 mldata = np.genfromtxt('%s.out'%file, usecols=(1,3,5))
52

```

```

53
54 maxunit=10
55 units=dict(zip(range(0,maxunit*4,maxunit),['','K','M','G']))
56 sizes = {'%d%s'%(b**(i),units[d],u):b**(i+d) for b,u in zip([2],[ 'B']) for d in units for i
57         in range(maxunit)}
58
59 xtlabls=[*['4B','16B','64B','256B','1KB','4KB','16KB','64KB','256KB','1MB'],
60          *['2MB','4MB','8MB','16MB','32MB','64MB','128MB','256MB','512MB','2GB']]
61 xtvals = [sizes.get(l,fmt_sizeof(l)) for l in xtlabls]
62
63
64
65
66
67 # k,l,m = '1GB','1MB','512B'
68 # print((sizes.get(k,fmt_sizeof(k)),indexer(indexer(mbddata[:,2],mbdata[:,0],sizes.get(l,
69         fmt_sizeof(l))),indexer(mbddata[:,1],mbdata[:,0],sizes.get(l,fmt_sizeof(l))),sizes.get(m,
70         fmt_sizeof(m)))[0]))
71
72 lims={
73     'all':[(xtlabls[0],xtlabls[-1]),(-2,16)],
74     'lowerleft':[( '4B','8KB'),(-2,1)],
75     'middleupper':[( '16B','64KB'),(6,12)],
76     'middle':[( '64B','64MB'),(1,4)],
77 }
78
79 heights={k:{l:-1 for l in ['h','v']} for k in lims}
80 heights['all']['v']=15
81 heights['lowerleft']['v'] = -1
82 heights['middle']['v'] = 2.5
83 heights['middleupper']['v'] = 11
84
85 lines={
86     'all':{ 'v': {
87         # **{(sizes.get(k,fmt_sizeof(k))/2,height):r'$\tilde{L}_{1,2,
88             3}\textrm{\%s}$'%k for i,k in
89             enumerate(['64B'])},
90         # **{(sizes.get(k,fmt_sizeof(k)),height):r'$L_{%d}\textrm{\%s}$'%(i+1,
91             k) for i,k in
92             enumerate(['128B'])},
93         # # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{%d}$'%(i+1) for
94             i,k in enumerate(['128B','4KB','16KB'])},
95         # **{(sizes.get(k,fmt_sizeof(k)),
96             height):r'$\tilde{L}_{%d}\textrm{\%s}$'%(i+1,k) for i,k in
97             enumerate(['32KB','1MB','24.75MB'])},
98
99         **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']-1):r'$\tilde{L}_{1,2,
100             3}$' for i,k in
101             enumerate(['64B'])},
102         **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']):r'$L_{%d}$'%(i+1)
103             for i,k in enumerate(['64B','512B','4KB'])},
104         # **{(sizes.get(k,fmt_sizeof(k)),
105             heights['all']['v']):r'$\tilde{L}_{%d}$'%(i+1) for i,k in
106             enumerate(['128B','4KB','16KB'])},
107         **{(sizes.get(k,fmt_sizeof(k)),
108             heights['all']['v']-1):r'$\tilde{L}_{%d}$'%(i+1) for i,k in
109             enumerate(['32KB','1MB','25MB'])},

```

```

93         **{(sizes.get(k,fmt_sizeof(k)),heights['all']['v']):r'$L_{%d}$'%(i+1) for
94             i,k in enumerate(['16KB','256KB','8MB'])},
95     },
96     'h': {
97         # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],mbdata[:,0],
98             sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in
99             enumerate(zip(['1GB'],['32KB'])},
100         **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],mbdata[:,0],
101             sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) if i<3 else
102             r'$T_{\textrm{mem}}$' for i,(k,l) in
103             enumerate(zip(['1GB','1GB','1GB','1GB'],['32KB','1MB','256MB','1GB'])},
104         # **{(sizes.get(k,fmt_sizeof(k)),indexer(indexer(mbddata[:,2],mbdata[:,0],
105             sizes.get(l,fmt_sizeof(l))),indexer(mbddata[:,1],mbdata[:,0],sizes.get(l,
106             fmt_sizeof(l))),sizes.get(m,fmt_sizeof(m)))[0]):r'$T_{%d}$'%(i+2) for
107             i,(k,l,m) in enumerate(zip(['1GB'],['1MB'],['512B'])},
108     },
109 },
110 'lowerleft':{'v': {
111     # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{1,2,
112     3}\|\textrm{s}$'%k for i,k in
113     enumerate(['64B'])},
114     # **{(sizes.get(k,fmt_sizeof(k)),
115     height):r'$\tilde{L}_{%d}\|\textrm{s}$'%(i+1,k) for i,k in
116     enumerate(['128B'])},
117     **{(sizes.get(k,fmt_sizeof(k)),
118     heights['lowerleft']['v']-0.5):r'$\tilde{L}_{1,2,3}$' for i,k in
119     enumerate(['64B'])},
120     **{(sizes.get(k,fmt_sizeof(k)),
121     heights['lowerleft']['v']):r'$\tilde{L}_{%d}$'%(i+1) for i,k in
122     enumerate(['64B','512B','4KB'])},
123     },
124     'h': {
125         # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],
126             mbddata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for
127             i,(k,l) in enumerate(zip(['256B'],['32KB'])},
128         **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],mbdata[:,0],
129             sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in
130             enumerate(zip(['4KB','4KB'],['32KB','1MB'])},
131     },
132 },
133 'middle':{'v': {
134     # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{1,2,
135     3}\|\textrm{s}$'%k for i,k in
136     enumerate(['64B'])},
137     # **{(sizes.get(k,fmt_sizeof(k)),
138     height):r'$\tilde{L}_{%d}\|\textrm{s}$'%(i+1,k) for i,k in
139     enumerate(['128B'])},
140     },
141     'h': {
142         # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],
143             mbddata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for
144             i,(k,l) in enumerate(zip(['256B'],['32KB'])},
145         **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],mbdata[:,0],
146             sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in
147             enumerate(zip(['4KB','4KB'],['32KB','1MB'])},
148     },
149 },
150 },
151 },
152 },
153 },
154 },
155 },
156 },
157 },
158 },
159 },
160 },
161 },
162 },
163 },
164 },
165 },
166 },
167 },
168 },
169 },
170 },
171 },
172 },
173 },
174 },
175 },
176 },
177 },
178 },
179 },
180 },
181 },
182 },
183 },
184 },
185 },
186 },
187 },
188 },
189 },
190 },
191 },
192 },
193 },
194 },
195 },
196 },
197 },
198 },
199 },
200 },
201 },
202 },
203 },
204 },
205 },
206 },
207 },
208 },
209 },
210 },
211 },
212 },
213 },
214 },
215 },
216 },
217 },
218 },
219 },
220 },
221 },
222 },
223 },
224 },
225 },
226 },
227 },
228 },
229 },
230 },
231 },
232 },
233 },
234 },
235 },
236 },
237 },
238 },
239 },
240 },
241 },
242 },
243 },
244 },
245 },
246 },
247 },
248 },
249 },
250 },
251 },
252 },
253 },
254 },
255 },
256 },
257 },
258 },
259 },
260 },
261 },
262 },
263 },
264 },
265 },
266 },
267 },
268 },
269 },
270 },
271 },
272 },
273 },
274 },
275 },
276 },
277 },
278 },
279 },
280 },
281 },
282 },
283 },
284 },
285 },
286 },
287 },
288 },
289 },
290 },
291 },
292 },
293 },
294 },
295 },
296 },
297 },
298 },
299 },
300 },
301 },
302 },
303 },
304 },
305 },
306 },
307 },
308 },
309 },
310 },
311 },
312 },
313 },
314 },
315 },
316 },
317 },
318 },
319 },
320 },
321 },
322 },
323 },
324 },
325 },
326 },
327 },
328 },
329 },
330 },
331 },
332 },
333 },
334 },
335 },
336 },
337 },
338 },
339 },
340 },
341 },
342 },
343 },
344 },
345 },
346 },
347 },
348 },
349 },
350 },
351 },
352 },
353 },
354 },
355 },
356 },
357 },
358 },
359 },
360 },
361 },
362 },
363 },
364 },
365 },
366 },
367 },
368 },
369 },
370 },
371 },
372 },
373 },
374 },
375 },
376 },
377 },
378 },
379 },
380 },
381 },
382 },
383 },
384 },
385 },
386 },
387 },
388 },
389 },
390 },
391 },
392 },
393 },
394 },
395 },
396 },
397 },
398 },
399 },
400 },
401 },
402 },
403 },
404 },
405 },
406 },
407 },
408 },
409 },
410 },
411 },
412 },
413 },
414 },
415 },
416 },
417 },
418 },
419 },
420 },
421 },
422 },
423 },
424 },
425 },
426 },
427 },
428 },
429 },
430 },
431 },
432 },
433 },
434 },
435 },
436 },
437 },
438 },
439 },
440 },
441 },
442 },
443 },
444 },
445 },
446 },
447 },
448 },
449 },
450 },
451 },
452 },
453 },
454 },
455 },
456 },
457 },
458 },
459 },
460 },
461 },
462 },
463 },
464 },
465 },
466 },
467 },
468 },
469 },
470 },
471 },
472 },
473 },
474 },
475 },
476 },
477 },
478 },
479 },
480 },
481 },
482 },
483 },
484 },
485 },
486 },
487 },
488 },
489 },
490 },
491 },
492 },
493 },
494 },
495 },
496 },
497 },
498 },
499 },
500 },
501 },
502 },
503 },
504 },
505 },
506 },
507 },
508 },
509 },
510 },
511 },
512 },
513 },
514 },
515 },
516 },
517 },
518 },
519 },
520 },
521 },
522 },
523 },
524 },
525 },
526 },
527 },
528 },
529 },
530 },
531 },
532 },
533 },
534 },
535 },
536 },
537 },
538 },
539 },
540 },
541 },
542 },
543 },
544 },
545 },
546 },
547 },
548 },
549 },
550 },
551 },
552 },
553 },
554 },
555 },
556 },
557 },
558 },
559 },
560 },
561 },
562 },
563 },
564 },
565 },
566 },
567 },
568 },
569 },
570 },
571 },
572 },
573 },
574 },
575 },
576 },
577 },
578 },
579 },
580 },
581 },
582 },
583 },
584 },
585 },
586 },
587 },
588 },
589 },
590 },
591 },
592 },
593 },
594 },
595 },
596 },
597 },
598 },
599 },
600 },
601 },
602 },
603 },
604 },
605 },
606 },
607 },
608 },
609 },
610 },
611 },
612 },
613 },
614 },
615 },
616 },
617 },
618 },
619 },
620 },
621 },
622 },
623 },
624 },
625 },
626 },
627 },
628 },
629 },
630 },
631 },
632 },
633 },
634 },
635 },
636 },
637 },
638 },
639 },
640 },
641 },
642 },
643 },
644 },
645 },
646 },
647 },
648 },
649 },
650 },
651 },
652 },
653 },
654 },
655 },
656 },
657 },
658 },
659 },
660 },
661 },
662 },
663 },
664 },
665 },
666 },
667 },
668 },
669 },
670 },
671 },
672 },
673 },
674 },
675 },
676 },
677 },
678 },
679 },
680 },
681 },
682 },
683 },
684 },
685 },
686 },
687 },
688 },
689 },
690 },
691 },
692 },
693 },
694 },
695 },
696 },
697 },
698 },
699 },
700 },
701 },
702 },
703 },
704 },
705 },
706 },
707 },
708 },
709 },
710 },
711 },
712 },
713 },
714 },
715 },
716 },
717 },
718 },
719 },
720 },
721 },
722 },
723 },
724 },
725 },
726 },
727 },
728 },
729 },
730 },
731 },
732 },
733 },
734 },
735 },
736 },
737 },
738 },
739 },
740 },
741 },
742 },
743 },
744 },
745 },
746 },
747 },
748 },
749 },
750 },
751 },
752 },
753 },
754 },
755 },
756 },
757 },
758 },
759 },
760 },
761 },
762 },
763 },
764 },
765 },
766 },
767 },
768 },
769 },
770 },
771 },
772 },
773 },
774 },
775 },
776 },
777 },
778 },
779 },
780 },
781 },
782 },
783 },
784 },
785 },
786 },
787 },
788 },
789 },
790 },
791 },
792 },
793 },
794 },
795 },
796 },
797 },
798 },
799 },
800 },
801 },
802 },
803 },
804 },
805 },
806 },
807 },
808 },
809 },
810 },
811 },
812 },
813 },
814 },
815 },
816 },
817 },
818 },
819 },
820 },
821 },
822 },
823 },
824 },
825 },
826 },
827 },
828 },
829 },
830 },
831 },
832 },
833 },
834 },
835 },
836 },
837 },
838 },
839 },
840 },
841 },
842 },
843 },
844 },
845 },
846 },
847 },
848 },
849 },
850 },
851 },
852 },
853 },
854 },
855 },
856 },
857 },
858 },
859 },
860 },
861 },
862 },
863 },
864 },
865 },
866 },
867 },
868 },
869 },
870 },
871 },
872 },
873 },
874 },
875 },
876 },
877 },
878 },
879 },
880 },
881 },
882 },
883 },
884 },
885 },
886 },
887 },
888 },
889 },
890 },
891 },
892 },
893 },
894 },
895 },
896 },
897 },
898 },
899 },
900 },
901 },
902 },
903 },
904 },
905 },
906 },
907 },
908 },
909 },
910 },
911 },
912 },
913 },
914 },
915 },
916 },
917 },
918 },
919 },
920 },
921 },
922 },
923 },
924 },
925 },
926 },
927 },
928 },
929 },
930 },
931 },
932 },
933 },
934 },
935 },
936 },
937 },
938 },
939 },
940 },
941 },
942 },
943 },
944 },
945 },
946 },
947 },
948 },
949 },
950 },
951 },
952 },
953 },
954 },
955 },
956 },
957 },
958 },
959 },
960 },
961 },
962 },
963 },
964 },
965 },
966 },
967 },
968 },
969 },
970 },
971 },
972 },
973 },
974 },
975 },
976 },
977 },
978 },
979 },
980 },
981 },
982 },
983 },
984 },
985 },
986 },
987 },
988 },
989 },
990 },
991 },
992 },
993 },
994 },
995 },
996 },
997 },
998 },
999 },
1000 },

```

```

123         **{(sizes.get(k,fmt_sizeof(k)),
124             heights['middle']['v']-0.5):r'$\tilde{L}_{1,2,3}$' for i,k in
125             enumerate(['64B'])},
126         **{(sizes.get(k,fmt_sizeof(k)),
127             heights['middle']['v']):r'$L_{%d}$'%(i+1) for i,k in
128             enumerate(['64B', '512B', '4KB'])},
129         **{(sizes.get(k,fmt_sizeof(k)),
130             heights['middle']['v']-1):r'$\tilde{L}_{%d}$'%(i+1) for i,k in
131             enumerate(['32KB', '1MB', '25MB'])},
132         **{(sizes.get(k,fmt_sizeof(k)),
133             heights['middle']['v']):r'$L_{%d}$'%(i+1) for i,k in
134             enumerate(['16KB', '256KB', '8MB'])},
135     },
136     'h': {
137         # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],
138             mbddata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for
139             i,(k,l) in enumerate(zip(['256B'], ['32KB']))},
140         **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],mbddata[:,0],
141             sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in
142             enumerate(zip(['16MB', '16MB'], ['32KB', '1MB']))},
143     },
144     'middleupper': {'v': {
145         # **{(sizes.get(k,fmt_sizeof(k)),height):r'$\tilde{L}_{1,2,3}$' for i,k in
146             enumerate(['64B'])},
147         # **{(sizes.get(k,fmt_sizeof(k)),
148             height):r'$L_{%d}$' for i,k in
149             enumerate(['128B'])},
150
151         **{(sizes.get(k,fmt_sizeof(k)),
152             heights['middleupper']['v']-0.5):r'$\tilde{L}_{1,2,3}$' for i,k
153             in enumerate(['64B'])},
154         **{(sizes.get(k,fmt_sizeof(k)),
155             heights['middleupper']['v']):r'$L_{%d}$'%(i+1) for i,k in
156             enumerate(['64B', '512B', '4KB'])},
157         **{(sizes.get(k,fmt_sizeof(k)),
158             heights['middleupper']['v']-1):r'$\tilde{L}_{%d}$'%(i+1) for i,k
159             in enumerate(['32KB', '1MB', '25MB'])},
160         **{(sizes.get(k,fmt_sizeof(k)),
161             heights['middleupper']['v']):r'$L_{%d}$'%(i+1) for i,k in
162             enumerate(['16KB', '256KB', '8MB'])},
163     },
164     'h': {
165         # **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],
166             mbddata[:,0],sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for
167             i,(k,l) in enumerate(zip(['256B'], ['32KB']))},
168         **{(sizes.get(k,fmt_sizeof(k)),np.max(indexer(mbddata[:,2],mbddata[:,0],
169             sizes.get(l,fmt_sizeof(l))))):r'$T_{%d}$'%(i+1) for i,(k,l) in
170             enumerate(zip(['16MB', '16MB', '32KB'], ['32KB', '1MB', '256MB']))},
171     },
172     },
173     # 'middle': {'v': **{(sizes.get(k,fmt_sizeof(k)), -4.5): 'L%d \LineSize: \s'%(i+1,k) for
174         i,k in enumerate(['4KB', '16KB'])}}
175 }

```



```

151
152
153 print(lines)
154
155
156 cmap = 'tab20b'
157 lengths=np.array(list(sorted(set(mbddata[:,0]))))
158
159 kwargs={}
160 kwargs['marker'] = ['.','*','^','s','d','+','']
161
162 kwargs['color'] = {k:v for k,v in zip(lengths,plt.get_cmap(cmap)(np.linspace(0, 1,
163     len(lengths))).tolist())}
164
165 kwargs['label'] = {k: 'L=%s'%(sizeof_fmt(k)) for k in lengths }
166
167
168
169
170 kwargs['lines'] = {k: {'linestyle':s,'color':c,'label':l,'alpha':a,'zorder':z} for k,l,c,s,a,z
171     in zip(
172         ['tilde{L}_{1,2,3}',
173         'tilde{L}',
174         '{L}_','L_',
175         'T_','Mem',
176         None],
177         ['True Line
178         Size','True
179         Cache
180         Size','Line
181         Size','Cache
182         Size','Access
183         Time','Access
184         Time',''],
185         ['k','b','k',
186         'b','g','g',
187         None],
188         ['--','--','-','-','-','-','-','-'],
189         [0.6,0.6,0.8,
190         0.8,0.8,0.8,
191         None],
192         [-1,-1,10,10,10,
193         10,None],
194     )}
195
196
197 for lim in lims:
198
199
200     fig,ax = plt.subplots()
201
202     # Plots
203
204     #for i in range(9,27):

```

```

187 for L in lengths:
188     #ax.plot(mbddata[istt:istt+i,1],mbdata[istt:istt+i,2],
189     ax.plot(mbddata[kwargs['inds'][L],1],mbdata[kwargs['inds'][L],2],**{k:kwargs[k][L] for k in
190     ['color','marker','label']})
191     #istt=istt+i+1
192
193 ax.set_ylabel('Time (nsec)')
194 ax.set_xlabel('Stride (bytes)')
195 ax.set_xscale('log',base=2)
196 ax.set_xticks(xtvals)
197 ax.set_xticklabels(xtlabels,rotation=45)
198 ax.set_ylim(*[l for l in lims[lim][1]])
199 ax.set_xlim(*[sizes.get(l,fmt_sizeof(l)) for l in lims[lim][0]])
200
201 # Lines
202 plotlines={k:lambda line,k=k,i=i,**kwargs: getattr(ax,'ax%sline'%k)(line[i],**kwargs) for i,k
203 in enumerate(['v','h'])}
204 annotatelines={k:lambda line,text,k=k,i=i,**kwargs:
205 getattr(plt,'annotate')(text=text,xy=line,**kwargs) for i,k in enumerate(['v','h'])}
206 for k in lines.get(lim,[]):
207     plotline=plotlines[k]
208     annotateline=annotatelines[k]
209     for line in lines[lim][k]:
210         text = lines[lim][k][line]
211         _kwargs = kwargs['lines'][None]
212         for x in kwargs['lines']:
213             if str(x) in text:
214                 _kwargs = kwargs['lines'][x]
215                 break
216         textline=list(line)
217         if 0 and (_kwargs['color'] == 'k' and _kwargs['linestyle'] == '-'):
218             textline[0] /=8
219         elif (_kwargs['color'] == 'g') and (lim=='all'):
220             textline[1] += 0.25
221         elif (_kwargs['color'] == 'g') and (lim=='lowerleft'):
222             textline[1] += 0.1
223         elif (_kwargs['color'] == 'g') and (lim=='middle'):
224             textline[0] *= 1.5
225             textline[1] += 0.06
226         elif (_kwargs['color'] == 'g') and (lim=='middleupper'):
227             textline[0] *= 1.2
228             textline[1] += 0.1
229         elif lim not in ['lowerleft']:
230             textline[0] *= 1.15
231         else:
232             textline[0] *= 1.15
233             # if lim == 'lowerleft':
234             #     textline[1] = -0.5
235
236     plotline(line,**_kwargs)
237     annotateline(textline,text)
238
239 handles,labels = ax.get_legend_handles_labels()
240 handles,labels = [h for i,(h,l) in enumerate(zip(handles,labels)) if l not in labels[:i]],[l
241 for i,(h,l) in enumerate(zip(handles,labels)) if l not in labels[:i]]
242 fig.legend(handles=handles,labels=labels,bbox_to_anchor=(0.7,0.5),loc='center
243 left',ncol=2,prop = {"size": 6},)

```

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

Code 5: CPU Info script.

```
1  #!/bin/bash
2
3  prog="/proc/cpuinfo"
4  field="cpu MHz"
5  pattern="s%.*: \([^ ]*\).*%\1%"
6  trials=5000
7
8  file=tmp1234.tmp
9  rm -f ${file}
10
11
12  for i in $(seq 1 ${trials})
13  do
14      #echo Trial: $i
15      grep "${field}" ${prog} | sed "${pattern}" >> ${file}
16      sleep 0.00000001
17  done
18
19  N=$(wc -l ${file} | sed "s:\([^ ]*\).*:\1:")
20  avg=$(paste -sd+ ${file} | bc)
21  avg=$(echo "scale=3 ; $avg / $N" | bc)
22
23  echo Avg\ (N=${N},Trials=${trials}) : ${avg}
24
25  rm ${file}
```

Code 6: membench job script.

```
1  #!/bin/bash
2
3  #SBATCH --account=ners570f20_class
4  #SBATCH --job-name=NERS570_Lab8
5  #SBATCH --partition=standard
6  #SBATCH --mail-user=mduschen@umich.edu
7  #SBATCH --mail-type=END
8  #SBATCH --nodes=1
9  #SBATCH --mem-per-cpu=8000m
10 #SBATCH          --time=01:00:00
11 #SBATCH          --ntasks-per-node=3
12
13
14 ./run.sh
```

Code 7: Matrix-Matrix multiply

```
1  #include <stdio>
2  #include <iostream>
3
4
5
6  void dgemm( char transa, char transb,
7              unsigned int m, unsigned int n, unsigned int k,
8              double alpha, const double * const * a,
9              const double * const * b,
10             double beta, double **c)
11 {
12     for(unsigned int i=0; i<n; i++){
13         for(unsigned int j=0; j<m; j++){
14             c[i][j] *= beta;
15         }
16     };
17     for(unsigned int l=0; l<k; l++){
18         for(unsigned int j=0; j<m; j++){
19             for(unsigned int i=0; i<n; i++){
20                 c[i][j] += alpha*a[i][l]*b[l][j];
21             }
22         }
23     };
24
25 }
26
```