

We analyze two functionally equivalent C programs. The two programs each initialize a square array a single element at a time. One program traverses the array in column major order (version 1); the second version traverses in row major order (version 2). We measure version 1 to have a higher latency across all tested array sizes compared to version 2. We posit that the explanation for this discrepancy lies with the hierarchy of memories and the principles of spatial locality. We attach all of the code required for our analysis at the end of this document.

Consider an uninitialized two-dimensional array in C. We may traverse this array and initialize each value to some integer along the way. We are able to initialize each element in any order, however we choose to compare initializing the elements in column-major and row-major order. If we initialize in column-major order we proceed down the first column and continue at the top of the second etc. In row-major order we proceed across the first row and continue at the front of the second. In the accompanying C program, two-dimensional arrays are allocated in memory in row major order. In other words, we create an array of pointers each of which point to the head of another array; each referred array is a row. This implies that adjacent elements within a row are adjacent in memory while adjacent elements within columns are not.

Now, let us consider how we may compare column-major traversal (version 1) with row-major traversal (version 2). We aim to measure the latency (wall-time) of each variant. In recognition of the fact that we carry out our experiments on a task-sharing operating system we perform two-hundred trials of each program at each array size of interest. We conduct our experiment for arrays with widths up to approximately three-thousand. To compensate for any overhead associated with calling the programs from our bench-marking script we measure the latency for calling an empty function and subtract this from our measures of interest. Figure 1 shows the results of these experiments. Clearly, version 2 (row-major) executed with much lower latency than version 1 (column-major) across all tested sizes.

So far, we have confirmed that traversing a two-dimensional array in row-major order if it has been allocated in row-major order results in a lower latency than traversing it in column-major order. Now, we attempt to explain this phenomenon. First, we consider if the assembly code associated with these two programs reveals the discrepancy. We analyze assembly code for each program as compiled by the GNU C Compiler. We

054
055
056
057
058
059
060
061
062
063
064
065
066
067
068
069
070
071
072
073
074
075
076
077
078
079
080
081
082
083
084
085
086
087
088
089
090
091
092
093
094
095
096
097
098
099
100
101
102
103
104
105
106
107

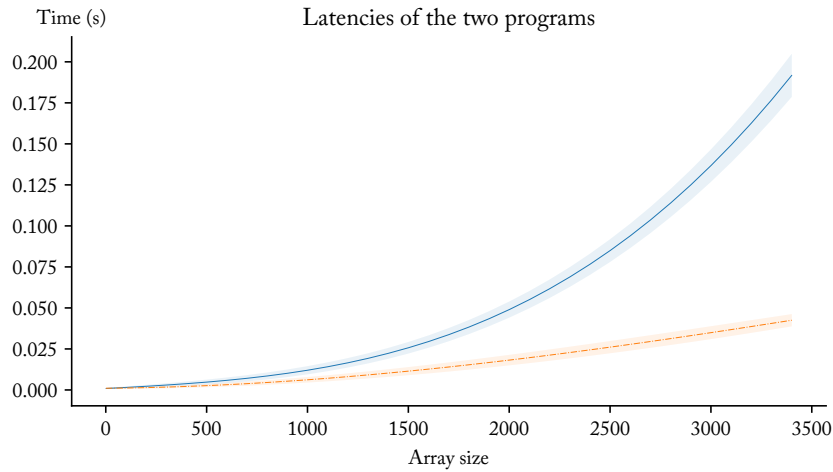


Figure 1: We compare the latencies of the two programs. Version 1 is represented by the solid blue line, while version 2 is represented by the dashed orange line. The banded regions show the range for a single standard deviation across 200 trials. The x-axis refers to the number of rows and columns for the square array under consideration. We interpolate the lines with B-splines.

notice that each program requires not only the same number of instructions but also the very same instructions. This indicates that we should expect to count the same number of cycles across executions of the programs. This analysis does not reveal the discrepancy but rather motivates us consider if some of the instructions may take a variable number of cycles to complete.

Once again analyzing the assembly code, we take note of the various types of instructions. For some, we have no reason to expect a variable number of cycles to completion; these include arithmetic instructions. On the other hand, we recognize the existence of the hierarchy of memories and suggest that data transfer instructions may require differing numbers of cycles to complete. We expect this as “nearer” parts of the hierarchy have lower access times while “further” parts have higher access times.

As we have already discussed, the two variants only differ in their memory access patterns. The row-major variant moves from an element to its neighbor in memory; we refer to the relationship between these two elements as spatial locality. Contrariwise, the column-major variant jumps between non-local elements. If our *only* memory were a random access memory we would expect locality to not effect the performance of the program. However locality does seem to effect the performance of the program. Therefore we must assume that the hierarchy of memories relies on locality for the increased performance.

108 The only conclusion we can reach is that when we show an interest in a
109 location in memory the processor must assume we are also interested other
110 locations nearby — that is we have an interest in a local neighborhood.
111 Therefore we propose that an automatic mechanism must exist that maps
112 potential regions of interest from the high-latency random access memory
113 to a low-latency memory “nearer” to the processor.
114
115
116 Thinking more specifically of our row-major program, we posit that when
117 the processor moves blocks of addresses to “nearer” memories, it has moved
118 not only the address we are interested in but also the next several as they
119 exist within a neighborhood. The column-major program does not have
120 this advantage therefore we assume it is slower because it more often needs
121 to access “further” memories.
122
123
124 To summarize: we consider the layout of our two-dimensional array in
125 memory, compare the two different paths we take through the array,
126 recognize the corresponding assembly code is virtually identical, and posit
127 that there is some unseen and automatic process occurring to lower the
128 latency of spatially local memory accesses. We recognize that this proposal
129 meshes with the concept of the hierarchy of memories and suggest that this
130 is how the performance discrepancy between the two variant programs
131 manifests itself.
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161

```
162
163
164 main.c
165
166 #include <stdlib.h>
167
168 int v1() {
169     int** array;
170     if (( array = malloc( SIZE*sizeof( int* ))) == NULL )
171         { /* error handling*/ }
172
173     for ( int i = 0; i < SIZE; i++ )
174     {
175         if (( array[i] = malloc( SIZE*sizeof(int) )) == NULL )
176             { /* error handling*/ }
177     }
178
179     for(int i = 0; i<SIZE; i++) {
180         for(int j = 0; j<SIZE; j++) {
181             array[j][i] = 0;
182         }
183     }
184
185     free(array);
186     return 0;
187 }
188
189 int v2() {
190     int** array;
191     if (( array = malloc( SIZE*sizeof( int* ))) == NULL )
192         { /* error handling*/ }
193
194     for ( int i = 0; i < SIZE; i++ )
195     {
196         if (( array[i] = malloc( SIZE*sizeof(int) )) == NULL )
197             { /* error handling*/ }
198     }
199
200     for(int i = 0; i<SIZE; i++) {
201         for(int j = 0; j<SIZE; j++) {
202             array[i][j] = 0;
203         }
204     }
205
206     free(array);
207
208     return 0;
209 }
210
211 int main() {
212     #ifdef V1
213         v1();
214     #endif
215     #ifdef V2
216         v2();
217     #endif
218 }
```

```

216
217
218 main.s
219
220     .file      1 ""
221     .section  .mdebug.abi32
222     .previous
223     .nan      legacy
224     .module   fp=32
225     .module   nooddspreg
226     .abicalls
227     .text
228     .align    2
229     .globl    v1
230     .set      nomips16
231     .set      nomicromips
232     .ent      v1
233     .type     v1, @function
234
235 v1:
236     .frame    $fp,56,$31           # vars= 16, regs= 3/0, args= 16, gp= 8
237     .mask     0xc0010000,-4
238     .fmask    0x00000000,0
239     .set      noreorder
240     .cpload   $25
241     .set      nomacro
242     addiu     $sp,$sp,-56
243     sw        $31,52($sp)
244     sw        $fp,48($sp)
245     sw        $16,44($sp)
246     move     $fp,$sp
247     movz     $31,$31,$0
248     .cprestore 16
249     li       $4,4000              # 0xfa0
250     lw       $2,%call16(malloc)($28)
251     nop
252     move     $25,$2
253     .reloc   1f,R_MIPS_JALR,malloc
254     jalr    $25
255     nop
256
257
258
259 $L3:
260     lw       $2,$24($fp)
261     nop
262     sll     $2,$2,2
263     lw       $3,$36($fp)
264     nop
265     addu    $16,$3,$2
266     li       $4,4000              # 0xfa0
267     lw       $2,%call16(malloc)($28)
268     nop
269     move     $25,$2
270     .reloc   1f,R_MIPS_JALR,malloc
271     jalr    $25
272     nop

```

```

270
271      lw      $28,16($fp)
272      sw      $2,0($16)
273      lw      $2,24($fp)
274      nop
275      addiu   $2,$2,1
276      sw      $2,24($fp)
277  $L2:
278      lw      $2,24($fp)
279      nop
280      slt     $2,$2,1000
281      bne    $2,$0,$L3
282      nop
283      sw      $0,28($fp)
284      b      $L4
285      nop
286
287  $L7:
288      sw      $0,32($fp)
289      b      $L5
290      nop
291
292  $L6:
293      lw      $2,32($fp)
294      nop
295      sll    $2,$2,2
296      lw      $3,36($fp)
297      nop
298      addu   $2,$3,$2
299      lw      $3,0($2)
300      lw      $2,28($fp)
301      nop
302      sll    $2,$2,2
303      addu   $2,$3,$2
304      sw      $0,0($2)
305      lw      $2,32($fp)
306      nop
307      addiu   $2,$2,1
308      sw      $2,32($fp)
309  $L5:
310      lw      $2,32($fp)
311      nop
312      slt     $2,$2,1000
313      bne    $2,$0,$L6
314      nop
315      lw      $2,28($fp)
316      nop
317      addiu   $2,$2,1
318      sw      $2,28($fp)
319  $L4:
320      lw      $2,28($fp)
321      nop
322      slt     $2,$2,1000
323      bne    $2,$0,$L7
324      nop
325      lw      $4,36($fp)

```

```

324         lw         $2,%call16(free)($28)
325         nop
326         move        $25,$2
327         .reloc      1f,R_MIPS_JALR,free
328 1:       jalr        $25
329         nop
330
331         lw         $28,16($fp)
332         move        $2,$0
333         move        $sp,$fp
334         lw         $31,52($sp)
335         lw         $fp,48($sp)
336         lw         $16,44($sp)
337         addiu       $sp,$sp,56
338         j          $31
339         nop
340
341         .set        macro
342         .set        reorder
343         .end        v1
344         .size       v1,.-v1
345         .align      2
346         .globl     v2
347         .set        nomips16
348         .set        nomicromips
349         .ent        v2
350         .type       v2,@function
351
352 v2:
353         .frame      $fp,56,$31           # vars= 16, regs= 3/0, args= 16, gp= 8
354         .mask       0xc0010000,-4
355         .fmask      0x00000000,0
356         .set        noreorder
357         .cpload     $25
358         .set        nomacro
359         addiu       $sp,$sp,-56
360         sw         $31,52($sp)
361         sw         $fp,48($sp)
362         sw         $16,44($sp)
363         move        $fp,$sp
364         movz        $31,$31,$0
365         .cprestore  16
366         li         $4,4000             # 0xfa0
367         lw         $2,%call16(malloc)($28)
368         nop
369         move        $25,$2
370         .reloc      1f,R_MIPS_JALR,malloc
371 1:       jalr        $25
372         nop
373
374         lw         $28,16($fp)
375         sw         $2,36($fp)
376         sw         $0,24($fp)
377         b          $L10
378         nop
379
380 $L11:
381         lw         $2,24($fp)
382         nop
383         sll        $2,$2,2

```

```

378         lw          $3,36($fp)
379         nop
380         addu         $16,$3,$2
381         li          $4,4000                # 0xfa0
382         lw          $2,%call16(malloc)($28)
383         nop
384         move        $25,$2
385         .reloc      1f,R_MIPS_JALR,malloc
386 1:        jalr       $25
387         nop
388
388         lw          $28,16($fp)
389         sw          $2,0($16)
390         lw          $2,24($fp)
391         nop
392         addiu       $2,$2,1
393         sw          $2,24($fp)
394 $L10:
395         lw          $2,24($fp)
396         nop
397         slt         $2,$2,1000
398         bne        $2,$0,$L11
399         nop
400
400         sw          $0,28($fp)
401         b          $L12
402         nop
403
404 $L15:
405         sw          $0,32($fp)
406         b          $L13
407         nop
408
409 $L14:
410         lw          $2,28($fp)
411         nop
412         sll        $2,$2,2
413         lw          $3,36($fp)
414         nop
415         addu       $2,$3,$2
416         lw          $3,0($2)
417         lw          $2,32($fp)
418         nop
419         sll        $2,$2,2
420         addu       $2,$3,$2
421         sw          $0,0($2)
422         lw          $2,32($fp)
423         nop
424         addiu       $2,$2,1
425         sw          $2,32($fp)
426 $L13:
427         lw          $2,32($fp)
428         nop
429         slt         $2,$2,1000
430         bne        $2,$0,$L14
431         nop
432
432         lw          $2,28($fp)
433         nop

```



```

432         addiu    $2,$2,1
433         sw      $2,28($fp)
434     $L12:
435         lw      $2,28($fp)
436         nop
437         slt    $2,$2,1000
438         bne    $2,$0,$L15
439         nop
440
441         lw      $4,36($fp)
442         lw      $2,%call16(free)($28)
443         nop
444         move    $25,$2
445         .reloc  1f,R_MIPS_JALR,free
446     1:     jalr   $25
447         nop
448
449         lw      $28,16($fp)
450         move    $2,$0
451         move    $sp,$fp
452         lw      $31,52($sp)
453         lw      $fp,48($sp)
454         lw      $16,44($sp)
455         addiu   $sp,$sp,56
456         j      $31
457         nop
458
459         .set    macro
460         .set    reorder
461         .end    v2
462         .size   v2,.-v2
463         .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"

```

bench.py

```

464     #!/bin/python3.6
465
466     import os
467     import numpy as np
468
469     from time import perf_counter
470
471     num_trials = 200
472
473     version = 'V2'
474
475     num_samples = 12
476
477     def get_perf_trials(cmd, num_trials=num_trials):
478         trials = []
479         for trial in range(num_trials):
480             start_time = perf_counter()
481             os.system(f'{cmd}')
482             end_time = perf_counter()
483             trials.append(end_time-start_time)
484
485         return np.array(trials)
486
487     empty_cmd = 'true'

```

```

486 call_latency = get_perf_trials(empty_cmd).mean()
487
488 for size in np.power(2, np.arange(0, num_samples, 0.25)).astype(np.int32):
489     compile_cmd = f'gcc main.c -O0 -D SIZE={size} -D {version}=1 -o {version}'
490     os.system(compile_cmd)
491
492     test_cmd = f'./{version}'
493     print(size, *(get_perf_trials(test_cmd) - call_latency), sep=',')

```

```

494 plot.py
495
496 #!/bin/python3.6
497
498 import numpy as np
499 import matplotlib.pyplot as plt
500 import matplotlib.font_manager as fm
501 import matplotlib
502
503 from scipy.interpolate import splrep, splev
504
505 def smooth(x, y):
506     tck = splrep(x, y, s=1)
507     xnew = np.arange(x.min(), x.max(), 100)
508     ynew = splev(xnew, tck, der=0)
509     return xnew, ynew
510
511 font = {'family' : 'Adobe Caslon Pro',
512        'size'    : 10}
513
514 matplotlib.rc('font', **font)
515
516 def read_data(file_name):
517     with open(file_name) as f:
518         data = []
519         for line in f.readlines():
520             data.append(np.fromstring(line, sep=','))
521
522     data = np.stack(data)
523     sizes = data[:,0].astype(np.int32)
524     times = data[:,1:]
525
526     return sizes, times
527
528 v1_sizes, v1_times = read_data('v1.data')
529 v2_sizes, v2_times = read_data('v2.data')
530
531 fig, ax = plt.subplots(1,1, figsize=(6, 3.5), dpi=900)
532 _, v1_means = smooth(v1_sizes, v1_times.mean(axis=1))
533 _, v2_means = smooth(v2_sizes, v2_times.mean(axis=1))
534 v1_sizes, v1_stddevs = smooth(v1_sizes, v1_times.std(axis=1))
535 v2_sizes, v2_stddevs = smooth(v2_sizes, v2_times.std(axis=1))
536
537 plt.plot(v1_sizes, v1_means)
538 plt.fill_between(v1_sizes, v1_means-v1_stddevs, v1_means+v1_stddevs, alpha=.1)
539
540 plt.plot(v2_sizes, v2_means, '-.')
541 plt.fill_between(v2_sizes, v2_means-v2_stddevs, v2_means+v2_stddevs, alpha=.1)

```

```
540     for line in ax.get_lines():
541         line.set_solid_capstyle('round')
542         plt.setp(line, linewidth=0.5)
543
544     h = ax.set_ylabel('Time (s)')
545     h.set_rotation(0)
546     ax.yaxis.set_label_coords(0,1.02)
547     ax.set_xlabel('Array size')
548
549     ax.spines['top'].set_visible(False)
550     ax.spines['right'].set_visible(False)
551
552     plt.title('Latencies of the two programs')
553
554     plt.tight_layout()
555     plt.savefig('plot.pdf')
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
```
