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

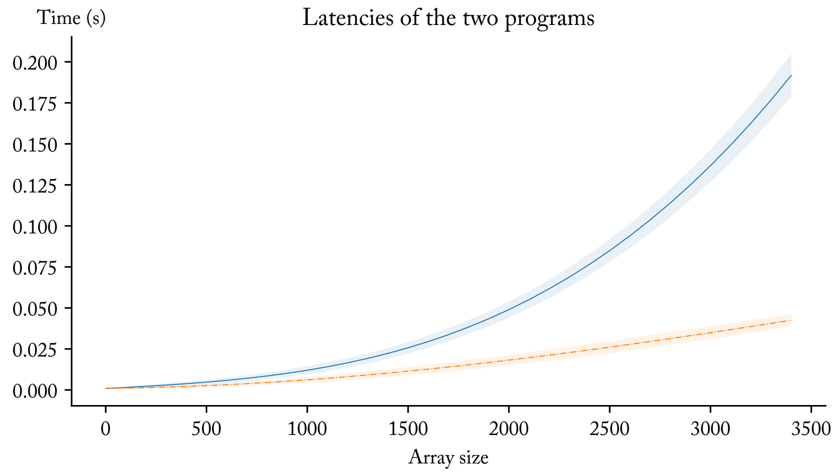

Figure 1: We compare the latencies of the two programs. Version 1 is represented by the solid blue line, while version 2 is represented by the dashed orange line. The banded regions show the range for a single standard deviation across 200 trials. The x-axis refers to the number of rows and columns for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

The only conclusion we can reach is that when we show an interest in a location in memory the processor must assume we are also interested other locations nearby — that is we have an interest in a local neighborhood. Therefore we propose that an automatic mechanism must exist that maps potential regions of interest from the high-latency random access memory to a low-latency memory "nearer" to the processor.

Thinking more specifically of our row-major program, we posit that when the processor moves blocks of addresses to "nearer" memories, it has moved not only the address we are interested in but also the next several as they exist within a neighborhood. The column-major program does not have this advantage therefore we assume it is slower because it more often needs to access "further" memories.

To summarize: we consider the layout of our two-dimensional array in memory, compare the two different paths we take through the array, recognize the corresponding assembly code is virtually identical, and posit that there is some unseen and automatic process occurring to lower the latency of spatially local memory accesses. We recognize that this proposal meshes with the concept of the hierarchy of memories and suggest that this is how the performance discrepancy between the two variant programs manifests itself.

main.c

```c
#include <stdlib.h>

int v1() {
  int** array;
  if (( array = malloc( SIZE*sizeof( int* ))) == NULL )
    { /* error handling*/ }

  for ( int i = 0; i < SIZE; i++ )
    {
      if (( array[i] = malloc( SIZE*sizeof(int) )) == NULL )
        { /* error handling*/ }
    }

  for(int i = 0; i<SIZE; i++) {
    for(int j = 0; j<SIZE; j++) {
      array[j][i] = 0;
    }
  }

  free(array);
  return 0;
}

int v2() {
  int** array;
  if (( array = malloc( SIZE*sizeof( int* ))) == NULL )
    { /* error handling*/ }

  for ( int i = 0; i < SIZE; i++ )
    {
      if (( array[i] = malloc( SIZE*sizeof(int) )) == NULL )
        { /* error handling*/ }
    }

  for(int i = 0; i<SIZE; i++) {
    for(int j = 0; j<SIZE; j++) {
      array[i][j] = 0;
    }
  }

  free(array);

  return 0;
}

int main() {
  #ifdef V1
    v1();
  #endif
  #ifdef V2
    v2();
  #endif
}
```

main.s

```
	.file	1 ""
	.section .mdebug.abi32
	.previous
	.nan	legacy
	.module	fp=32
	.module	nooddspreg
	.abicalls
	.text
	.align	2
	.globl	v1
	.set	nomips16
	.set	nomicromips
	.ent	v1
	.type	v1, @function
v1:
	.frame	$fp,56,$31		# vars= 16, regs= 3/0, args= 16, gp= 8
	.mask	0xc0010000,-4
	.fmask	0x00000000,0
	.set	noreorder
	.cpload	$25
	.set	nomacro
	addiu	$sp,$sp,-56
	sw	$31,52($sp)
	sw	$fp,48($sp)
	sw	$16,44($sp)
	move	$fp,$sp
	movz	$31,$31,$0
	.cprestore	16
	li	$4,4000			# 0xfa0
	lw	$2,%call16(malloc)($28)
	nop
	move	$25,$2
	.reloc	1f,R_MIPS_JALR,malloc
1:	jalr	$25
	nop

	lw	$28,16($fp)
	sw	$2,36($fp)
	sw	$0,24($fp)
	b	$L2
	nop

$L3:
	lw	$2,24($fp)
	nop
	sll	$2,$2,2
	lw	$3,36($fp)
	nop
	addu	$16,$3,$2
	li	$4,4000			# 0xfa0
	lw	$2,%call16(malloc)($28)
	nop
	move	$25,$2
	.reloc	1f,R_MIPS_JALR,malloc
1:	jalr	$25
	nop
```

```
        lw      $28,16($fp)
        sw      $2,0($16)
        lw      $2,24($fp)
        nop
        addiu   $2,$2,1
        sw      $2,24($fp)
$L2:
        lw      $2,24($fp)
        nop
        slt     $2,$2,1000
        bne     $2,$0,$L3
        nop

        sw      $0,28($fp)
        b       $L4
        nop

$L7:
        sw      $0,32($fp)
        b       $L5
        nop

$L6:
        lw      $2,32($fp)
        nop
        sll     $2,$2,2
        lw      $3,36($fp)
        nop
        addu    $2,$3,$2
        lw      $3,0($2)
        lw      $2,28($fp)
        nop
        sll     $2,$2,2
        addu    $2,$3,$2
        sw      $0,0($2)
        lw      $2,32($fp)
        nop
        addiu   $2,$2,1
        sw      $2,32($fp)
$L5:
        lw      $2,32($fp)
        nop
        slt     $2,$2,1000
        bne     $2,$0,$L6
        nop

        lw      $2,28($fp)
        nop
        addiu   $2,$2,1
        sw      $2,28($fp)
$L4:
        lw      $2,28($fp)
        nop
        slt     $2,$2,1000
        bne     $2,$0,$L7
        nop

        lw      $4,36($fp)
```

```
        lw      $2,%call16(free)($28)
        nop
        move    $25,$2
        .reloc  1f,R_MIPS_JALR,free
1:      jalr    $25
        nop

        lw      $28,16($fp)
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        lw      $16,44($sp)
        addiu   $sp,$sp,56
        j       $31
        nop

        .set    macro
        .set    reorder
        .end    v1
        .size   v1, .-v1
        .align  2
        .globl  v2
        .set    nomips16
        .set    nomicromips
        .ent    v2
        .type   v2, @function
v2:
        .frame  $fp,56,$31              # vars= 16, regs= 3/0, args= 16, gp= 8
        .mask   0xc0010000,-4
        .fmask  0x00000000,0
        .set    noreorder
        .cpload $25
        .set    nomacro
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        sw      $16,44($sp)
        move    $fp,$sp
        movz    $31,$31,$0
        .cprestore      16
        li      $4,4000                 # 0xfa0
        lw      $2,%call16(malloc)($28)
        nop
        move    $25,$2
        .reloc  1f,R_MIPS_JALR,malloc
1:      jalr    $25
        nop

        lw      $28,16($fp)
        sw      $2,36($fp)
        sw      $0,24($fp)
        b       $L10
        nop

$L11:
        lw      $2,24($fp)
        nop
        sll     $2,$2,2
```

```
        lw      $3,36($fp)
        nop
        addu    $16,$3,$2
        li      $4,4000                 # 0xfa0
        lw      $2,%call16(malloc)($28)
        nop
        move    $25,$2
        .reloc  1f,R_MIPS_JALR,malloc
1:      jalr    $25
        nop

        lw      $28,16($fp)
        sw      $2,0($16)
        lw      $2,24($fp)
        nop
        addiu   $2,$2,1
        sw      $2,24($fp)
$L10:
        lw      $2,24($fp)
        nop
        slt     $2,$2,1000
        bne     $2,$0,$L11
        nop

        sw      $0,28($fp)
        b       $L12
        nop

$L15:
        sw      $0,32($fp)
        b       $L13
        nop

$L14:
        lw      $2,28($fp)
        nop
        sll     $2,$2,2
        lw      $3,36($fp)
        nop
        addu    $2,$3,$2
        lw      $3,0($2)
        lw      $2,32($fp)
        nop
        sll     $2,$2,2
        addu    $2,$3,$2
        sw      $0,0($2)
        lw      $2,32($fp)
        nop
        addiu   $2,$2,1
        sw      $2,32($fp)
$L13:
        lw      $2,32($fp)
        nop
        slt     $2,$2,1000
        bne     $2,$0,$L14
        nop

        lw      $2,28($fp)
        nop
```

```
        addiu        $2,$2,1
        sw        $2,28($fp)
$L12:
        lw        $2,28($fp)
        nop
        slt        $2,$2,1000
        bne        $2,$0,$L15
        nop

        lw        $4,36($fp)
        lw        $2,%call16(free)($28)
        nop
        move        $25,$2
        .reloc        1f,R_MIPS_JALR,free
1:        jalr        $25
        nop

        lw        $28,16($fp)
        move        $2,$0
        move        $sp,$fp
        lw        $31,52($sp)
        lw        $fp,48($sp)
        lw        $16,44($sp)
        addiu        $sp,$sp,56
        j        $31
        nop

        .set        macro
        .set        reorder
        .end        v2
        .size        v2, .-v2
        .ident        "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"
```

---

bench.py

```python
#!/bin/python3.6

import os
import numpy as np

from time import perf_counter

num_trials = 200

version = 'V2'

num_samples = 12

def get_perf_trials(cmd, num_trials=num_trials):
    trials = []
    for trial in range(num_trials):
        start_time = perf_counter()
        os.system(f'{cmd}')
        end_time = perf_counter()
        trials.append(end_time-start_time)

    return np.array(trials)

empty_cmd = 'true'
```

```
call_latency = get_perf_trials(empty_cmd).mean()

for size in np.power(2, np.arange(0, num_samples, 0.25)).astype(np.int32):
    compile_cmd = f'gcc main.c -O0 -D SIZE={size} -D {version}=1 -o {version}'
    os.system(compile_cmd)

    test_cmd = f'./{version}'
    print(size, *(get_perf_trials(test_cmd) - call_latency), sep=',')
```

---

plot.py

```
#!/bin/python3.6

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
import matplotlib

from scipy.interpolate import splrep, splev

def smooth(x, y):
    tck = splrep(x, y, s=1)
    xnew = np.arange(x.min(), x.max(), 100)
    ynew = splev(xnew, tck, der=0)
    return xnew, ynew

font = {'family' : 'Adobe Caslon Pro',
        'size'   : 10}

matplotlib.rc('font', **font)

def read_data(file_name):
    with open(file_name) as f:
        data = []
        for line in f.readlines():
            data.append(np.fromstring(line, sep=','))

    data = np.stack(data)
    sizes = data[:,0].astype(np.int32)
    times = data[:,1:]

    return sizes, times

v1_sizes, v1_times = read_data('v1.data')
v2_sizes, v2_times = read_data('v2.data')

fig, ax = plt.subplots(1,1, figsize=(6, 3.5), dpi=900)
_, v1_means =  smooth(v1_sizes, v1_times.mean(axis=1))
_, v2_means =  smooth(v2_sizes, v2_times.mean(axis=1))
v1_sizes, v1_stddevs =  smooth(v1_sizes, v1_times.std(axis=1))
v2_sizes, v2_stddevs =  smooth(v2_sizes, v2_times.std(axis=1))

plt.plot(v1_sizes, v1_means)
plt.fill_between(v1_sizes, v1_means-v1_stddevs, v1_means+v1_stddevs, alpha=.1)

plt.plot(v2_sizes, v2_means, '-.')
plt.fill_between(v2_sizes, v2_means-v2_stddevs, v2_means+v2_stddevs, alpha=.1)
```

```
for line in ax.get_lines():
    line.set_solid_capstyle('round')
    plt.setp(line, linewidth=0.5)

h = ax.set_ylabel('Time (s)')
h.set_rotation(0)
ax.yaxis.set_label_coords(0,1.02)
ax.set_xlabel('Array size')

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)


plt.title('Latencies of the two programs')

plt.tight_layout()
plt.savefig('plot.pdf')
```