Gradient bitmap calculator analysis

Used metrics:

- Execution time;
- Memory consumption during the execution (heap + stack);

  RSS - resident set size, is the portion of memory occupied by a process that is held in main memory (RAM);
- Code readability (this part is based on subjective opinion of a one programmer).

Metric	Kotlin	Go	Dart
Execution time (ms)	11.60659	0.013057	0.021
Max RSS Memory(kb)	36240	1816	12992
Minor (frame) pf	3210	188	1810
Voluntary con. switches	49	1	0
Involuntary con. switches	295	13	7

Kotlin

Language performance

Execution time:

Kotlin is significantly slower than Go and Dart. The reason of this is that Kotlin does a lot of internal checks, such as boundary checks on arrays and synchronization checks on each access to I/O. Some of the check delays come from Java libraries, which Kotlin uses.

The other thing is that Kotlin is aimed to eliminate Null Pointer Exceptions from the code. It introduces its own null-safety principle: a variable can be if and only if it is of nullable type. The Kotlin compiler does null check automatically. Using Kotlin is a safer option in terms of null safety, however it takes some time to perform such checks.

The other factor that affects Kotlin execution time in this particular code example is the be the usage of Kotlin experimental types (UByte and UByte Array). It requires additional verification: check whether the experimental types are used and whether they are marked as experimental types in the code.

In addition, Kotlin uses Mark-and-Sweep garbage collector type. It means that before the each garbage collection cycle, the execution of the program is stopped in order to prevent the allocating new objects when going through the objects tree. The normal program execution resumes after GC stops its work.

Memory management:

Kotlin is run on JVM so it uses the same memory management scheme. The stack is automatically managed by operating system rather than JVM. The interesting part for us is how the heap space is managed.

The heap memory is divided into two parts: "young generation space" and "tenured space" . The young generation space is where new objects live This space is managed by "minor GC". The young generation space is further divided into "eden" and "survivor" space. When new object is created memory is allocated in "eden" space. The "survivor" space is where objects that survived the minor GC are stored. The "tenured space" is where objects that reached the maximum tenure threshold during minor GC live. The objects which reached max tenured threshold during the minor GC, will be moved to tenured space. It is managed by "major GC".

The garbage collection in Kotlin (JVM) is generational. The minor GC is invoked when JVM is unable to allocate space for a new Object, the Eden is getting full. So the higher the allocation rate, the more frequently Minor GC is executed. The major GC is invoked when: developer explicitly call it from the program, JVM decides there is not enough tenured space as it gets filled up from minor GC cycles, during minor GC, if the JVM is not able to reclaim enough memory from the Eden or survivor spaces. As you can notice, if there are plenty of unused objects but the heap is not full or more memory is not needed the GC will not be invoked. The Kotlin program occupies a greater portion of RSS because of this GC scheme.

External functionality:

Kotlin is designed with Java interoperability in mind. Existing Java code can be called from Kotlin in a natural way. Although in this code it is not needed, it is worth to mention that many thing in Kotlin may be done by using java libraries. For example, reading from file or stdin is done using java Scanner class.

Code readability

```
@kotlin.ExperimentalUnsignedTypes
fun fillGradientLine(arr: kotlin.UIntArray, line: Int, w: Int, a: UByte, b: UByte): Unit {
    val delta = b - a
    val offset = line * w
    for (i in 0 until w) {
        var fraction = i.toFloat() / w.toFloat()
        arr[offset + i] = a.toUByte() + (delta.toFloat() * fraction).toUInt()
    }
}
@kotlin.ExperimentalUnsignedTypes
fun generateGrayGradient(h: Int, w: Int, a: UByte, b: UByte): kotlin.UIntArray{
    val size = w * h
    val arr = kotlin.UIntArray(size)
    for (i in 0 until h){
        fillGradientLine(arr, i, w, a, b)
    }
    return arr
}
@kotlin.ExperimentalUnsignedTypes
fun main() {
    val h = 50
    val w = 50
    val a: UByte = 55.toUByte()
```

```
    val b: UByte = 233.toUByte()
    val start = System.nanoTime();
    generateGrayGradient(h, w, a, b)
    val timeMilli = (System.nanoTime() - start) / 1000000.0f
    println("$timeMilli")
}
```

Code length:

Approximately the same as Go code, longer than Dart code. Since unsigned Byte type is now experimental type in Kotlin, Each function, where operations with such types are done, should be marked with "@kotlin.ExperimentalUnsignedTypes", so it makes the code longer a bit.

Separators:

By convention semicolon is not used in Kotlin. One may use it if they like but it is not recommended. so There will be less mistakes that are connected to end line typos. Functions are wrapped in curly braces.

General impression:

It is worth to mention that it became shorter than Java code and it is easier to read. Comparing to Go and Dart Kotlin code seems to be cleaner and more human-readable.

Go

Language performance

Execution time:

The Go language is the fastest among the sample ones. Memory consumption is the best (the lowest) as well. Go employs a thread-local cache to speed up small object allocations and maintains scan/noscan spans to speed up GC. This structure along with the process avoids fragmentation to a great extent making compact unnecessary during GC.

Memory management:

The Go's memory allocator was modeled upon TCMalloc (Thread-Caching Malloc), therefore there is no generational memory as in JVM. Instead, Golang uses page heap (mheap). Any data, which size cannot be calculated in compile time (dynamic data) is stored in mheap. This is the biggest block of memory, where garbage collection takes place. The resident set is divided into pages of 8 KB memory and each page is managed by one global mheap object. Large objects (greater than 32 KB) are allocated from mheap directly.

The mheap manages pages grouped into different constructs. We are interested in mspan construct. It is a double linked list that holds the address of the start page, span size class and the number of pages in the span. Like TCMalloc, Go also divides Memory Pages into a block of 67 different classes by size starting at 8 bytes up to 32 kilobytes. Each span exists twice, one for objects with pointers (scan classes) and one for objects with no pointers (noscan classes). This helps during GC: alive objects are not located in noscan spans, so they are not traversed during GC. It saves some amount of time.

Unlike many garbage collected languages, in go many objects are allocated directly in program stack. Firstly, go compiler finds objects whose lifetime is known at compile-time and allocates them on the stack rather than in heap memory. Then during compilation Go does the escape analysis to determine what can go into Stack (static data) and what needs to go into Heap

(dynamic data).

External functionality:

Go uses "fmt" package for formatted input/output operations (analogous to C's printf and scanf) and "time" package to check the system time (need this to measure the elapsed time).

Code readability

```
package main
import (
    "fmt"
    "time")

func fillGradientLine(arr []uint8, line int, w int, a uint8, b uint8) {
    delta := b - a
    offset := line * w
    for i := 0; i < w; i++ {
        fraction := float32(i) / float32(w)
        arr[offset + i] = a + uint8(float32(delta) * fraction)
    }
}
func generateGrayGradient(h int, w int, a uint8, b uint8) []uint8 {
    size := w * h
    arr := make([]uint8, size)
    for i := 0 ; i < h; i++ {
        fillGradientLine(arr, i, w, a, b)
    }
    return arr
}
func main() {
    h := 50
    w := 50
    a := uint8(55)
    b := uint8(233)
    start := time.Now()
    generateGrayGradient(h, w, a, b)
    elapsed := float32(time.Since(start).Nanoseconds()) / 1000000
    fmt.Printf("%f\n", elapsed)
}
```

Code length:

The length of the code is approximately the same as the Kotlin code, longer than Dart code.

Separators:

Like C, Go's formal grammar uses semicolons to terminate statements, but unlike in C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them.

The rule is this. If the last token before a newline is an identifier, a basic literal such as a number or string constant or one of the tokens

`break continue fallthrough return ++ -- ) }`

the lexer always inserts a semicolon after the token. This could be summarized as, “if the newline comes after a token that could end a statement, insert a semicolon”.

A semicolon can also be omitted immediately before a closing brace, so a statement such as

`go func() { for { dst <- <-src } }()`

needs no semicolons. Idiomatic Go programs have semicolons only in places such as for loop clauses, to separate the initializer, condition, and continuation elements. They are also necessary to separate multiple statements on a line, should you write code that way.

One consequence of the semicolon insertion rules is that you cannot put the opening brace of a control structure (if, for, switch, or select) on the next line.

General impression:

The code is more difficult to read than Dart or Kotlin. The variable declaration with ":=" symbol, type names, that wrapped the value such as uint(55) (*lines 25-26*), the brackets that denote array before the array type (*line 16, return type*), all these things make the code to complicated to read. In general code is "not great, not terrible". It is understandable for human and number of special symbols is low.

Dart

Language performance

Execution time:

Slower than Go but faster than Kotlin.

RSS Memory:

Less than Kotlin but more than Go.

External functionality:

Dart does not require additional imports to perform output operations, print function is built-in.

Code readability

```
1  void fillGradientLine(arr, line, w, a, b){
2      var delta = b - a;
3      var offset = line * w;
4      for(var i = 0; i < w; i++){
5          var fraction = i / w;
6          arr[offset + i] = a + (delta * fraction).floor();
7      }
8  }
9
10 List generateGrayGradient(h, w, a, b){
11     var size = h * w;
12     var arr = [];
13     for(var i = 0; i < size; i++) arr.add(0);
14     for(var i = 0; i < h; i++) fillGradientLine(arr, i, w, a, b);
15     return arr;
16 }
```

```dart

void main() {
    var h = 10;
    var w = 10;
    var a = 55;
    var b = 233;
    Stopwatch stopwatch = new Stopwatch()..start();
    generateGrayGradient(h, w, a, b);
    print(stopwatch.elapsedMicroseconds /1000); // executing time in milliseconds
}
```

Code length:

Dart code is the shortest among the sample ones (26 lines).

Separators:

The end-line semicolon is obligatory. Code blocks are wrapped into the curly braces.

General impression:

Although there are end line separators (semi columns), the code seems to be clean. It is easy to read. Kotlin and Dart are pretty similar when declaring variables. var and val in Kotlin is a nice way of doing the more ‘traditional’ var and final in Dart. const val is a bit more verbose in Kotlin compared to just const in Dart.