Optimizing Sliding Performance in iOS

Qin Zhao[1,2], Qi Qi[1,2](✉), Lejian Zhang[1,2], and Qiwei Shen[1,2]

[1] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, People's Republic of China
zhaoqinl92@gmail.com, {qiqi, zhanglejian, shenqiwei}@ebupt.com

[2] EBUPT Information Technology Co., Ltd, Beijing 100191, People's Republic of China

Abstract. How to improve iOS sliding performance has always been the focus of iOS application optimization. This paper analyzes the principle of AutoLayout and Frame view layout, the opportunity of network loading, CPU and GPU performance consumption during sliding process. First, we provide the appropriate solution to avoid using AutoLayout, and adjust the time of network loading by preloading to reduce the waiting time dynamically. Pre-cache and asynchronous rendering to reduce the main thread CPU consumption is implemented to reduce the main thread CPU consumption, and at the same time, GPU consumption is reduced by asynchronous rendering. Finally, verify the feasibility and effectiveness of the optimization scheme by experiments. It is verified that the percentage of the main thread CPU consumption decreases by 17.2% and FPS increases from 37 Hz to 60 Hz.

Keywords: Sliding performance · AutoLayout · Pre-cache Asynchronous · FPS

1 Introduction

The operating systems of iOS provide many UI views for users to browse more information via sliding up and down. In fact, developers can not request all resource from the Internet. In this scenario, operating systems need to load latest data after exploiting exhaustive search. Regardless of iOS or Android, network action is expensive because it costs much resource such as time, network traffic, electricity and so on. But the network situation of mobile phones is so bad in some cases that users have to wait until application receives network response which causes terrible user experience. On the other hand, after obtaining the resource from remote server, device should visualize data on the hardware screen. The CPU and GPU would finish work respectively to supply cache data for rendering on screen. The heavy load caused by CPU and GPU could lead to frame loss and set a delay response after users touching screen. There are many factors which affect the performance of CPU or GPU. AutoLayout based on Cassowary makes iOS layout simple and quick [1], while the improper way using AutoLayout or high load operation would drag the CPU. Blended layers, misaligned images and off-screen rendering are the killer of GPU, where

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
P. H. J. Chong et al. (Eds.): SmartGIFT 2018, LNICST 245, pp. 111–121, 2018.
https://doi.org/10.1007/978-3-319-94965-9_12

off-screen rendering affects dramatically because it wastes a lot of performance to rendering off-screen images. That is to say, the tardiness of network and visualize data would affect sliding performance in iOS and UE (user experience). It is important that how to discover these key points and solve the knotty problems.

This paper is committed to optimizing sliding performance to improve user experience. From discovering the issues of AutoLayout and off-screen rendering performance, this paper proposes some method, such as pre-cache, asynchronous rendering to reduce the heavy burdens of device. So that many layout and rendering sites can be resolved, to improve the sliding performance of application. Due to the similar the hardware system or foundation framework for iOS or Android platforms [2], some optimization methods mentioned in this paper can be applied in other platforms.

2 Performance Optimization

The optimizing performance of application focuses on code structure optimizing and operational performance optimizing. References of Method for Mobile User Interface Design Patterns Creation for iOS Platform [3] gives guidelines for developers work in a high level of usability quality purpose, which belongs to structure optimizing. While it cannot promote performance when application is running. And References of On-device Objective-C Application Optimization Framework for High Performance Mobile Processors proposes a methodology to tailor a given Objective-C application and its associated device-specific shared library codebase using on-device post-compilation code optimization and transformation [4] that modified runtime library of iOS to acquire a high performance. But it would lose many features of runtime library, for example, the magic feature of using JavaScriptCore and runtime replace some method when calling some object message. Maybe there are other approach optimizing performance, and developer can avoid incorrect way that drag CPU and GPU of mobile device.

2.1 Preloading

With the development of mobile communication and the wide coverage of WIFI, mobile devices have better network services. However, network request is still an expensive operation. On the one hand, it costs a good deal of traffic, and developers need to take full advantage of the returned data as much as possible. On the other hand, network response time is unpredictable, which may cause bad user experience because of long waiting time. These two points are more obvious in iOS sliding. Initialization request data should not be too much, because users will not browse related information and waste a lot of traffic. So application should load the data according to the number of pages. Usually, it is time to load when slide to the bottom of the list page with showing a load animation and making network request. The drawback is that sliding page will stop until the network responds to new data, which wastes time for users. As a result, our primary goal is preloading, that is processing network request before reaching the bottom of the list. In this way, application has obtained new data before sliding to the bottom for display. Through predicting the users' behavior, it could

effectively save the traffic, and make user use application without waiting for network requests. Developers can change the network load time despite they could not determine the network situation. In another word, the preloading method is not optimizing network request but optimizing the opportunity for network requests.

After demonstrating the correctness of preloading, the timing of preloading should be taken into account. Normally, setting the fixed threshold is a simply approach. For example, we set 0.7 as a threshold, and it will process network request when sliding at the 70% of the total height. The corresponding code is as follows (Fig. 1):

```
// threshold
CGFloat threshold = 0.7;
// current page
int currentPage = 0;
// slide loop logic
- (void)scrollViewDidScroll:(UIScrollView *) scrollView {
    // current slide offset
    CGFloat offsetHeight = scrollView.contentOffset.y + scrollView.frame.size.height;
    // current total height
    CGFloat totalHeight = scrollView.contentSize.height;
    // the ratio of current height to total height
    CGFloat ratio = offsetHeight / totalHeight;
    // exceed the threshold and make request
    if (ratio >= threshold) {
        currentPage += 1;
        // request next page data
    }
}
```

Fig. 1. Fixed threshold

The code is very simple, but in fact it would not avoid the following problem: As the number of pages increasing, the height of the list will continue growing. A fixed threshold will lead to the growth of the height of unviewed page. In order to compare the waste of network resources, we assume that each table cell has the same height. Therefor the size of the data is able to reflect the height of the view. The threshold is shown in Table 1.

Table 1. The effect of fixed threshold.

Page	TotalNum	TimeNum	Diff
1	10	7	3
2	20	14	6
3	30	21	9
4	40	28	12
5	50	35	15
6	60	42	18

Page represents the number of pages and TotalNum represents the total count of data respectively. TimeNum indicates the data which has been viewed when preloading. Diff shows the amount of data which has not been browsed. It is displayed that the amount of data which has not yet been browsed will increase as the number of

pages growing. The opportunity for preloading is kept in advance and leads to a lot of data being loaded that users would not browse. Furthermore, it causes the waste of application traffic.

To above issues, we design a new method to optimize the preloading: For each page, it is specified that 70% of the amount of new data is set as the threshold. If the sliding height exceeds this threshold, application will request new data and change the threshold again. The code is depicted in Fig. 2.

```
//preloaded when reach to 70% of the last page
CGFloat threshold = 0.7;
// current page
int currentPage = 1;
// the amount of data per page
int perPageNum = 10;
// slide loop logic
- (void)scrollViewDidScroll:(UIScrollView *) scrollView {
    // current slide offset
    CGFloat offsetHeight = scrollView.contentOffset.y + scrollView.frame.size.height;
    // current total height
    CGFloat totalHeight = scrollView.contentSize.height;
    CGFloat ratio = offsetHeight / totalHeight;

    // the amount that need to make a network request
    CGFloat needRead = itemPerPage * threshold + currentPage * itemPerPage;
    CGFloat totalItem = itemPerPage * (currentPage + 1);
    // dynamically adjust the threshold
    CGFloat newThreshold = needRead / totalItem
    // exceed the threshold and make request
    if ratio >= newThreshold {
        currentPage += 1;
        // request next page data
    }
}
```

Fig. 2. Dynamical threshold

With the growth of page amount, through adjusting the threshold, the amount of data which is not viewed remains within a stable range, and as a result network resources could be saved. Shown in Fig. 3, as the number of pages increasing, the threshold will dynamically grow to delay the preloading time.

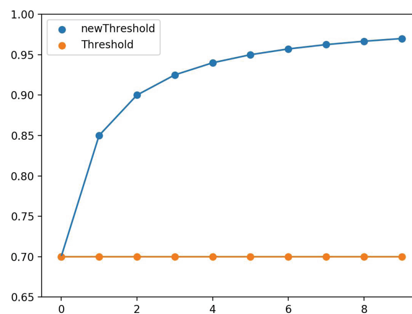

Fig. 3. The curve of dynamically threshold and fixed threshold

2.2 The Bottleneck of AutoLayout Performance

AutoLayout [7] is the implementation of the UI layout program after iOS6, which can easily solve the UI adaptation problem. Nevertheless, we should abandon this technical program to get higher sliding performance. It is because that compared with the traditional Frame layout, AutoLayout makes the design of UI convenient, but it would affect CPU performance when running application. The traditional Frame layout is to specify the location of a UI view in the parent view, which must include the coordinates of the axis x, y and the length and width of the view itself. AutoLayout is based on the Cassowary algorithm that adds a lot of constraints to the view, such as the distance to the left and top of the parent view and so on. All of these constraints are abstracted as a set of linear equations or inequalities. Finally, the operating system get the x, y coordinates and the width and height by solving the set of linear equations or inequalities. However, the calculation of linear equations requires CPU to consume. If there are a large number of views using AutoLayout, it will need CPU to solve multiple sets of linear equations at the same time. The refresh rate of iOS is 60 Hz, which is vital for optimizing sliding performance. If the CPU solves the layout for more than 16.67 ms (1/60 s = 16.67 ms), it will cause the data of this frame not to be prepared in the buffer. When V-sync signal coming, it is inevitable that FPS [8] will be declined to impress sliding performance if the required rendering data is not available when the buffer data is read.

Figure 4 is obtained by randomly generating N views on iOS10.2.1 iPhone7 and the layout modes are AutoLayout and Frame respectively. The abscissa in the graph represents the number of render views, and the ordinate represents the time to render the views.

From Fig. 2, Frame mode has better performance in all cases. For example, it takes about 11.7 ms to render when the number of views is 100, and AutoLayout needs 32.0 ms at the same circumstance. The rising speed of the curve of AutoLayout is also significantly greater than that of Frame. And according to iOS rendering frequency, we can see that when the rendering time is greater than 16.67 ms, it will certainly generate block. As seen from Fig. 3, it will produce the performance problem when generating

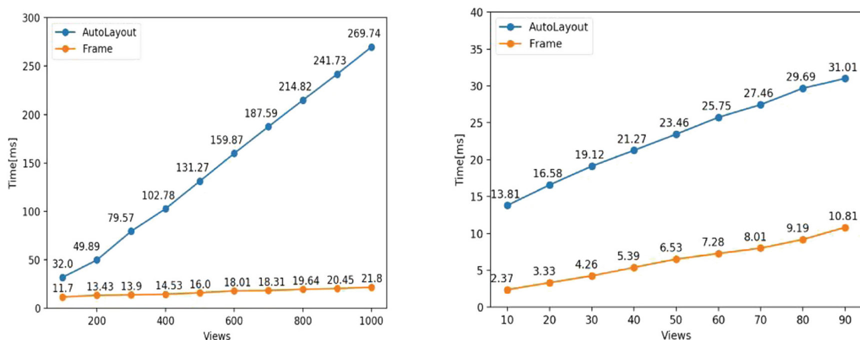

Fig. 4. AutoLayout vs. frame

approximately 20 views on the experimental device under AutoLayout mode. Relatively, Frame layout only consumes about 16 ms to generated about 500 views.

Therefore, if it needs to maintain a high FPS or high performance during the sliding process, we should not select AutoLayout as technical proposal. While Frame layout is more cumbersome, it can be compensated by efficient operation to provide better sliding performance. Obliviously the first step in optimizing sliding performance is to Frame layout.

2.3 Pre-cache

The widgets of UI Views (such as UITableView [9], UICollectionView [10]) in iOS programming exist reuse mechanism [5], which only stores the current cell displayed on the screen, regardless of all the list cell. The biggest advantage of this mechanism is to save memory space. Assuming that every data generates a cell, the memory will soon be exhausted if there are a large amount of data. However, there is a flaw in the reuse mechanism that different data have different styles. The application needs to recalculate the layout information in real time and then display it. In the process of rapid sliding, a large number of calculation has a bad impact on the performance of the main thread CPU. How to ease the pressure of the CPU during the sliding process is very important for optimizing performance. This paper employs pre-cache to solve the problem. Pre-cache is to calculate the view layout in advance and cache them. Pre-cache creates asynchronous thread to parse the layout model after getting data from network, and each model stores the information for displaying. The layout of the Cell is uniquely determined by the contents of the model, so application will calculate the layout information in advance based on the content of the model and store these data.

It will notify the main thread to refresh the UI after all data has been analyzed. Although it is necessary to continually update the UI during the rapid sliding of the list, pre-cache can reduce the operating load of the CPU during sliding because it can read UI view data directly without recalculating.

2.4 Asynchronous Drawing

Via the pre-cache processing, it has been reduced the burden of CPU during sliding to a great degree. However, we find that there are still some points to optimize after analyzing the performance loss of CPU during sliding process. The function drawInText of UILabel is responsible for rendering text that is running on the main thread. It will be not serious if the number of UILabel [11] is small, and has not become a major constraint on the impact of sliding performance. If UILabel needs to display a lot of text, it would make CPU performance degradation because it occupies a lot of resources of the main thread. We use self-defined CALayer [12] layer to ease the burden of main thread CPU by transferring the timing of rendering to asynchronous thread, using CoreText [13] Framework and asynchronous thread to draw text. The UILabel and AsyncLabel architecture of iOS is depicted in Fig. 5.

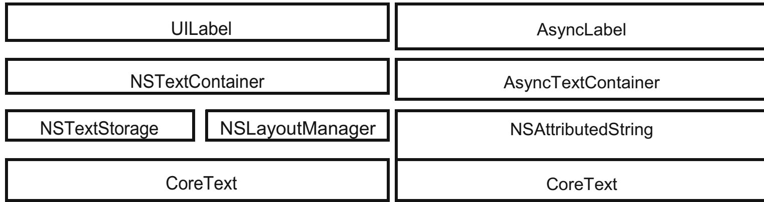

Fig. 5. UILabel architecture diagram vs AsyncLabel architecture diagram

Both AsyncLabel and UILabel are based on CoreText. And CoreText is also the basic framework for all text and image widget in iOS programming. AsyncLabel keeps the basic information of the NSAttributedString, and the NSTextStorage and NSLayoutManager layer are greatly simplified. The biggest difference between UILabel and AsyncTextContainer is the rendering time of AsyncTextContainer layer occurs on asynchronous threads, which finally renders the picture in the main thread that described by text information through the CTFrame [14], CTLine [15], CTRun [16]. Obviously, asynchronous drawing can reduce the CPU load during the sliding process further.

2.5 Asynchronous Rendering

After pre-caching, we can find that the main thread of CPU usage is relieved, but it is still not smooth during sliding. Then we turn our attention to easing the burden on GPU. Detecting FPS by the Instruments, a fantastic tool for monitoring all performance of iOS device, we find that the biggest factor which affects GPU performance is off-screen rendering. Off-screen rendering composites a part of the layer tree into a new buffer (which is off-screen, i.e. not on the screen), and then that buffer is rendered onto the screen. Generally, we do our best to avoid off-screen rendering, because it costs too much.

In the client of iOS, the business scene which can trigger off-screen rendering is to set the users' pictures as rounded corners. How to set the image circular efficiently is one of the key factors to improve the sliding performance. The traditional way to set the circular is to cover the CALayer, but it would cause off-screen rendering and consume GPU performance. Of course, if the client can get the circular image from server directly, there is no GPU rendering problem. For the same picture, different business scenes need different shapes of pictures, for example, some places require a rectangle and some place require rounded corners. So the fundamental solution is to instruct the client to handle rounded pictures locally and the essence of the problem translates into how to set the rounded image efficiently. The optimization scheme designed in this paper is using asynchronous rendering. We import the original image resource to an asynchronous thread and then use the underlying CoreGraphics in the asynchronous thread for rounded corners or other effects. The processing here does not create rounded corners by setting the CALayer, but it employs the Bezier curve to cut the original picture as a new picture resource. Finally, the processed image is passed to the main thread to display. In this case, it could improve the FPS through asynchronous rendering to a large extent.

3 Experiment and Validation

In the optimization function discussed above, the preload can prepare the data in advance, which improves the smoothness of the sliding and do not need to wait for network to respond [6]. The previous comparison has been concluded that AutoLayout affects sliding performance dramatically. Ignoring the two factors that have been identified to affect the sliding performance, we set two groups of layout to compare the optimization effect in the pre-cache, asynchronous rendering, asynchronous loading. The first set of data is not set the above optimization point, and the second set of data is on the contrary. Test environment is macOS 10.12.2 system, XCode 8.2.1, iPhone 7, iOS 10.2.1 system.

It can be seen from the above experimental data that in about 30 s of time-consuming, depicted in Figs. 6 and 7. The percentage of the main thread CPU consumption decreases by 17.2%, and at the same time, asynchronous thread CPU time-consuming increases by 19.4%. The results are consistent with the expectations

Weight	Self Weight	Symbol Name
22.94 s 73.9%	0 s	▶Main Thread 0x6a22e
1.07 s 3.4%	0 s	▶_dispatch_worker_thread3 0x6a265
627.00 ms 2.0%	0 s	▶_dispatch_worker_thread3 0x6a23d
532.00 ms 1.7%	0 s	▶_dispatch_worker_thread3 0x6a26e
517.00 ms 1.6%	0 s	▶_dispatch_worker_thread3 0x6a249
420.00 ms 1.3%	0 s	▶_dispatch_worker_thread3 0x6a2b0
365.00 ms 1.1%	0 s	▶_dispatch_worker_thread3 0x6a2c6
353.00 ms 1.1%	0 s	▶_dispatch_worker_thread3 0x6a23f
259.00 ms 0.8%	0 s	▶_dispatch_worker_thread3 0x6a2c0
151.00 ms 0.4%	0 s	▶_dispatch_worker_thread3 0x6a27a
150.00 ms 0.4%	0 s	▶_dispatch_worker_thread3 0x6a2d6
120.00 ms 0.3%	0 s	▶_dispatch_worker_thread3 0x6a2e5
50.00 ms 0.1%	0 s	▶_dispatch_worker_thread3 0x6a2e9
40.00 ms 0.1%	0 s	▶_dispatch_worker_thread3 0x6a262
35.00 ms 0.1%	0 s	▶_dispatch_worker_thread3 0x6a2d9
17.00 ms 0.0%	0 s	▶_dispatch_worker_thread3 0x6a264
17.00 ms 0.0%	0 s	▶_dispatch_worker_thread3 0x6a26b
16.00 ms 0.0%	0 s	▶_dispatch_worker_thread3 0x6a23e
7.00 ms 0.0%	0 s	▶std::__1::__thread_proxy<std::__1::tuple<com::bytedance::push::WebSocketClient::WebSocketClient 0x6a269
6.00 ms 0.0%	0 s	▶WebCore::ThreadGlobalData::~ThreadGlobalData 0x6a27e
1.00 ms 0.0%	0 s	▶WebCore::StorageThread::threadEntryPoint 0x6a289
1.00 ms 0.0%	0 s	▶asio::detail::posix_thread::func<asio::detail::resolver_service_base::work_io_service_runner>::run 0x6a26a

Fig. 6. The analysis of unoptimized application CPU consumption

Weight	Self Weight	Symbol Name
17.31 s 56.7%	0 s	▶Main Thread 0x72d23
1.72 s 5.6%	0 s	▶_dispatch_worker_thread3 0x72d48
1.61 s 5.2%	0 s	▶_dispatch_worker_thread3 0x72db4
1.51 s 4.9%	0 s	▶_dispatch_worker_thread3 0x72d71
1.31 s 4.2%	0 s	▶_dispatch_kevent_worker_thread 0x72d69
1.01 s 3.3%	0 s	▶_dispatch_worker_thread3 0x72ded
716.00 ms 2.3%	0 s	▶_dispatch_worker_thread3 0x72d84
696.00 ms 2.2%	0 s	▶_dispatch_worker_thread3 0x72d72
578.00 ms 1.8%	0 s	▶_dispatch_worker_thread3 0x72dee
536.00 ms 1.7%	0 s	▶_dispatch_worker_thread3 0x72ddd
316.00 ms 1.0%	0 s	▶_dispatch_worker_thread3 0x72d70
271.00 ms 0.8%	0 s	▶_dispatch_worker_thread3 0x72d3f
115.00 ms 0.3%	0 s	▶_dispatch_worker_thread3 0x72e00
110.00 ms 0.3%	0 s	▶_dispatch_worker_thread3 0x72d40
88.00 ms 0.2%	0 s	▶_dispatch_worker_thread3 0x72df3
18.00 ms 0.0%	0 s	▶_dispatch_kevent_worker_thread 0x72d3d
11.00 ms 0.0%	0 s	▶_dispatch_worker_thread3 0x72d5d
9.00 ms 0.0%	0 s	▶std::__1::__thread_proxy<std::__1::tuple<com::bytedance::push::WebSocketClient::WebSocketClient 0x72d6b
6.00 ms 0.0%	0 s	▶RunWebThread 0x72d7e
4.00 ms 0.0%	0 s	▶_dispatch_worker_thread3 0x72d67
1.00 ms 0.0%	0 s	▶asio::detail::posix_thread::func<asio::detail::resolver_service_base::work_io_service_runner>::run 0x72d6c
1.00 ms 0.0%	0 s	▶-[NSRunLoop run] 0x72d7f

Fig. 7. The analysis of optimized application CPU consumption

after optimization. And it is confirmed that the increasement in asynchronous thread consumption is slightly greater than the reduced consumption in main thread because the growth in the number of threads on the CPU also has a certain impact. Via pre-caching and asynchronous rendering, the main thread of the work can move to the asynchronous thread which reduces the main thread consumption.

On the other hand, we also compare the GPU performance before and after the asynchronous rendering (from the perspective of FPS), shown in Figs. 8 and 9. Asynchronous rendering mainly improves the performance of the GPU. Before optimizing, FPS is around 37 Hz during fast sliding process, and after optimization, FPS increases to 60 Hz. In the process of rapid sliding, it is very smooth in line with the expected optimization.

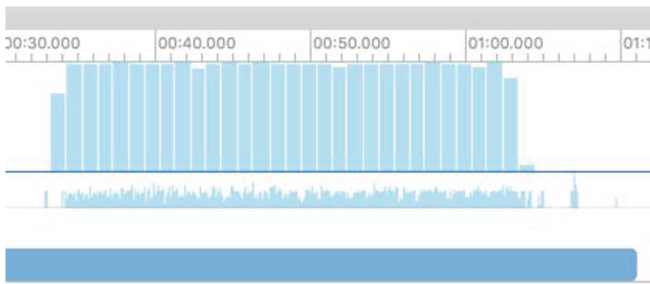

Fig. 8. Unoptimized application of FPS data with GPU

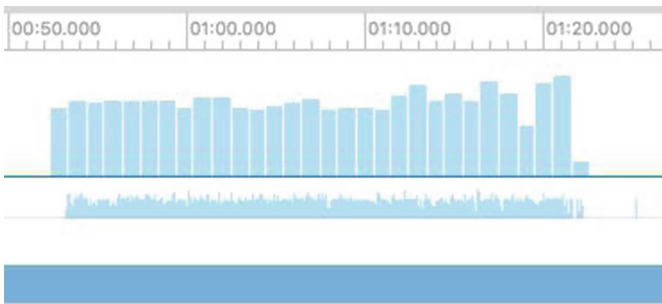

Fig. 9. Optimized application of FPS data with GPU

4 Conclusions

Sliding performance optimization in iOS is the key to providing a good user experience, especially in the new application [18]. In this paper, we constantly adjust the timing of requesting network data through dynamic preloading and get data without

user awareness to save network resources, as well as reducing the sliding process to wait for the network response time. And then from the performance of the CPU and GPU, we use Frame layout, pre-cache, asynchronous rendering to reduce CPU performance loss, and use asynchronous rendering to avoid GPU loss brought by off-screen rendering. It is found that all of these measures make the usage rate of CPU declining 30%, the usage rate of GPU declining 40%, FPS rise from 37 Hz to 60 Hz, and achieve a very smooth sliding effect. In the future, the iOS performance can be optimized by the popular machine learning mechanism [18–20].

Acknowledgement. This work was supported in part by the (1) National Natural Science Foundation of China (No. 61671079, 61771068, 61471063) (2) Beijing Municipal Natural Science Foundation (No. 4182041).

References

1. Badros, G.J., Borning, A., Stuckey, P.J.: The Cassowary linear arithmetic constraint solving algorithm. ACM Trans. Comput.-Hum. Interact. **8**(4), 267–306 (2001)
2. Novac, O.C., Novac, M., Gordan, C., Berczes, T.: Comparative study of Google Android, Apple iOS and Microsoft Windows phone mobile operating systems. In: 2017 14th International Conference on Engineering of Modern Electric Systems (EMES). Oradea, Romania, pp. 154–159 (2017)
3. Wetchakorn, T., Prompoon, N.: Method for mobile user interface design patterns creation for iOS platform. In: 2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE), Songkhla, Thailand, pp. 150–155 (2015)
4. Bournoutian, G., Orailoglu, A.: On-device Objective-C application optimization framework for high performance mobile processors. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, pp. 1–6 (2014)
5. Ferreira, P.: Reclaiming storage in an object oriented platform supporting extended C++ and Objective-C applications. In: Proceedings 1991 International Workshop on Object Orientation in Operating Systems, Palo Alto, CA, USA, pp. 100–102 (1991)
6. Gutierrez, A., Dreslinski, R.G., Wenisch, T.F.: Full-system analysis and characterization of interactive smartphone applications. In: 2011 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, pp. 81–90 (2011)
7. Develop Apple. https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/AutolayoutPG/index.html
8. WikiPedia. https://en.wikipedia.org/wiki/FPS
9. Develop Apple. https://developer.apple.com/documentation/uikit/uitableview
10. Develop Apple. https://developer.apple.com/documentation/uikit/uicollectionview
11. Develop Apple. https://developer.apple.com/documentation/uikit/uilabel
12. Develop Apple. https://developer.apple.com/reference/quartzcore/calayer
13. Develop Apple. https://developer.apple.com/documentation/coretext
14. Develop Apple. https://developer.apple.com/documentation/coretext/ctframe
15. Develop Apple. https://developer.apple.com/documentation/coretext/ctline
16. Develop Apple. https://developer.apple.com/documentation/coretext/ctrun-61n
17. Xu, P., Yin, Q., Huang, Y., Song, Y.-Z., Ma, Z., Wang, L., Xiang, T., Kleijn, W.B., Guo, J.: Cross-modal subspace learning for fine-grained sketch-based image retrieval. Neurocomputing **278**, 75–86 (2018)

18. Ma, Z., Xue, J.-H., Leijon, A., Tan, Z.-H., Yang, Z., Guo, J.: Decorrelation of neutral vector variables: theory and applications. IEEE Trans. Neural Netw. Learn. Syst. **29**(1), 129–143 (2018)
19. Liu, W., Cao, J., Yang, L., Xu, L., Qiu, X., Li, J.: AppBooster: boosting the performance of interactive mobile applications with computation offloading and parameter tuning. IEEE Trans. Parallel Distrib. Syst. **28**(6), 1593–1606 (2017)
20. Ma, Z., Rana, P.K., Taghia, J., Flierl, M., Leijon, A.: Bayesian estimation of Dirichlet mixture model with variational inference. Pattern Recogn. **47**(9), 3143–3157 (2014)