Designing a Browser to Benefit from Multi-core Silicon

Ekioh Ltd, Cambridge UK.
flow@ekioh.com

Abstract

This paper investigates the impact of the evolution in processor technology upon HTML browser performance, highlighting some limitations in current browser design and ways in which these limitations can be overcome. It asserts that overcoming these limitations is key to offering 4K UIs on mass-market consumer products in the very near future.

Introduction

HTML browsers are increasingly being used for application rendering and user interface (UI) presentation. The driving reasons behind this are that browsers reduce UI authoring complexity and provide a level of hardware abstraction which enables authoring to happen in parallel with hardware design.

Browser technology has also allowed the application authoring community to grow beyond embedded software engineers to include creative designers. This has led to a marked improvement in the visual quality of user interfaces and the look and feel of applications.

This flexibility and increased visual quality comes at a cost; the browser is one of the most demanding components within a device and achieving the necessary responsiveness directly drives CPU selection requirements.

Processor evolution

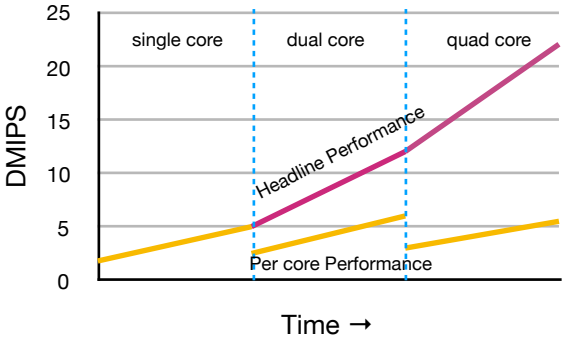
Processing power has been increasing year-on-year[1] providing comfort that if a UI design is not responsive enough on cost effective silicon today, it will be by the time that mass production hardware is selected. As the clock speed increases, the heat generated also increases. To ensure this heat is kept below acceptable limits, the voltage or process size need to decrease. This eventually becomes impractical for physical or cost reasons, but increasing the number of cores can give a boost to the overall performance. A reduction in clock speed is usual to compensate for the additional heat generated by this increase in core count.

Figure 1 outlines a harmonised view of headline processor speed compared with that of each individual core across a range of consumer electronics products.

Impact of multi-core hardware on software

Because a multi-core device is comprised of a number of independent processing cores, advertised performance can only be achieved if the task being performed can be split effectively so that each core is loaded equally. The overhead of communicating between parallel tasks and the presence of any sequential tasks that cannot be split will govern the upper limit of the overall performance gain[2]. Applications using an inherently single threaded design are unable to reap the benefits of multi-core silicon by themselves, although some benefits can be seen if multiple applications are run at the same time.

When considering how the benefits of multi-core processors impact browsers, it is necessary to delve into the browser's internal workings. Browser activity largely comprises *Networking* (where content and resources are loaded), *Parsing* (to create the Document Object Model), *Scripting*, *Layout* (where text and image placement are calculated), *Painting* (drawing images and text into offscreen buffers), and *Compositing* the offscreen buffers onto the screen.



The headline processing speed of multi-core devices benefits from increases in the number of cores and, indirectly, from reductions in process size. Year on year improvements of around 30% were achieved in headline processing speed over a five year period despite the relatively small performance improvements of each core.

Figure 1

Table 1 shows the breakdown of tasks from a sample of various types of HTML applications. It shows that scripting and layout are by far the largest consumers of time, and whilst their combined size is fairly consistent, their balance varies with the type of application.

Analysis of CPU activity for different HTML5 application types

	Networking	Parsing	Scripting	Layout	Painting	Compositing*
General Websites[3]	3%	2%	60%	32%	3%	0%
OTT Applications	2%	2%	55%	37%	4%	0%
UI Applications	1%	1%	33%	54%	11%	0%

**Compositing is a GPU function and presents negligible CPU load*

Table 1

Historically all browser activity has been contained within a single thread and so has been unable to benefit from multi-core silicon developments. More recently some browsers have allocated networking and compositing into separate processes, and introduced Web Workers. Web Workers can execute JavaScript on a separate thread, although they have limited value in many applications because they have no access to the DOM tree.

The key tasks of scripting and layout, which between them typically account for over 90% of the processor load, remain on a single thread. This presents something of a problem when the browser is used for an application or UI.

If 90% of a browser's processing load is single threaded, and therefore cannot be split across multiple cores, the benefits of the shift to multi-core silicon will not be realised. Indeed, as the average speed per core of a quad core device is considerably lower than that of an equivalent single core device, browser performance will noticeably reduce.

In the scenario where the browser is used to do multiple things at once, such as having several web pages open, each page can run on a separate core so any reduction in individual page responsiveness will be less noticeable. However, UI rendering scenarios tend to run as a single page and so single application performance is key; relying upon a traditional browser design will result in a marked decrease in responsiveness following a shift to multi-core.

In order to properly benefit from multi-core silicon, the browser must be able to keep each of the cores fully loaded in both the single page and multi-tab use cases.

Recent browser evolution has focused upon the multi-tab use case, but has done little to benefit the single page performance needed for application and UI rendering. Instead, a new approach is needed where each page uses as many cores as possible in order to benefit from their combined performance.

Several projects have investigated the use of parallelism with varying degrees of success, the most well known of which is the Servo project[4] by Mozilla Research.

Designing a new browser

Splitting the browser's activities into smaller tasks, and executing them on separate cores will reduce the overall time taken until the page is usable. Since the performance of a multithreaded browser is determined by the duration of the longest task the aim is to break these down into as many smaller tasks as possible and execute them in parallel. A first step to designing a multithreaded browser is to identify a series of tasks and implement a method for their parallel execution. An obvious candidate is image decode as decoding one JPEG has no interaction with another.

There is considerable scope for splitting layout into multiple independent parallel tasks[5]. When considering a printed page consisting of paragraphs of text, nothing in one paragraph affects the word wrapping of other paragraphs, just their vertical position. If a word were added to a paragraph the only effect would be the possible shifting up or down of the following paragraphs. This is important since each paragraph can be considered independent and word wrapped in parallel.

HTML works on the same principle. The CSS box model specification considers every object as a simple rectangular box where the position of each box can affect the position of those around it, but not their contents. Once the width of each box is determined, it is possible to lay out its contents and those of its children independently. These tasks can then be executed in parallel and spread across the available cores in a multi-core environment.

User interfaces lend themselves towards parallel layout. They tend to consist of many absolutely positioned items, and these have no knock-on effects towards other objects. For instance, in the case of a TV guide, each channel row or even each programme cell is often absolutely positioned. Even though these regularly contain little or no text, there is still a benefit to being able to lay them out in parallel, especially if animating their size or position. To achieve the ideal 60fps frame rate, scripting, layout and paint must take less than 16.6ms.

Multithreaded layout algorithm

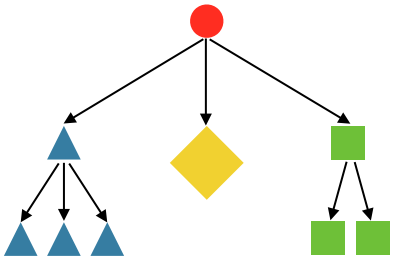
A parallel layout algorithm comprises five distinct steps:

1. Construct the render tree from the DOM tree and style rules.
2. Calculate the minimum and maximum text widths.
3. Deduce the width of all the paragraphs.
4. Lay out all the normal-flow paragraphs and other objects and calculate their height.
5. Determine the position of out-of-flow (absolutely positioned) objects.

Whilst each step typically occurs sequentially, within the steps themselves, calculations for unrelated objects can be performed in parallel.

Figure 2 shows how the calculation of the paragraph text widths can be achieved in parallel.

Calculate the paragraph widths



In this simple render tree example, the width calculations for the circular object must be performed first. The second row of objects can then be executed in parallel with no dependencies on their siblings. The triangles in the third row are calculated when their parent triangle has been calculated, and similarly the squares in the third row. The calculations for the diamond in the second row could be very complex but this does not delay the calculations of any triangle or square.

Figure 2

Browsers need to keep track of the position and size of each render object for many reasons, not least, mouse detection and JavaScript. Traditionally, each object stores this information in screen co-ordinates but, when objects are being laid out in parallel, the position of an object can not be known at the start as it depends on the position and height of any preceding object. Instead, all objects are considered to have their own co-ordinate system with their origin at 0,0.

There are several exceptions that complicate the ability to layout objects in parallel, the most frequently occurring one being the CSS property 'float'. This is usually used by content developers to force text to wrap around an image rather than appear below it. The complication is introduced because a floating image can be taller than the

paragraph it is in, and so affect the width of subsequent paragraphs. These must be laid out sequentially until the text has passed the image, then parallel layout can resume.

Utilising the GPU

Animation performance is another area which is critical to delivering a great user experience and one way to achieve this is to make full use of the GPU. GPUs are a standard component within multi-core System on a Chip (SoC) devices and are primarily designed for graphics operations. They can be used to deal with certain tasks that traditional browsers perform on the CPU such as the painting task which occurs directly after layout has occurred.

The Ekioh Technology Paper, *An Optimal Browser Design for GPU Utilisation*, examines the way traditional browsers utilise the GPU and proposes an efficient alternative approach. Using the GPU to complete the painting task delivers a small benefit in overall performance when browsing static web pages but, for dynamic UI applications, the performance improvements can be significant.

Putting theory into practice

Ekioh's Flow HTML5 browser uses the parallel multithreaded layout architecture and efficient GPU techniques described above to deliver the full potential of multi-core silicon. Designed for the consumer electronics market, Flow's use of the GPU relies on standard OpenGL ES APIs to ensure compatibility with all popular multi-core SoC solutions and supplements these with vendor-specific performance extensions where available.

By making such extensive use of the GPU, Flow is also well equipped to handle the huge increase in pixel count which will occur when user interfaces shift to 4K resolutions.

Several obstacles have been encountered during the development of Flow, and a frequent issue is that third party libraries are not often designed with multithreading as a primary concern. Most font and hardware acceleration graphics libraries limit access to one thread at a time. Flow has worked around all these limitations:

Some hardware acceleration graphics libraries have a restriction on allocating memory to a single thread. This complicates multithreaded image decode but is simply solved since web image formats store the decoded size in their header. The image header is parsed on the main thread which then allocates the required memory, allowing the decode itself to then be queued up and run on a separate thread.

Layout code needs fast, efficient, access to font metrics in order to word wrap text. The font metrics for a given font and size never change, so Flow stores the minimum efficient set of font metrics required in a specially designed, wait-free, cache. The cache is populated by one thread (protecting the font library by use of a lock), but its contents can later be read by any thread with no contention considerations.

The aim is to create many lightweight tasks so no single task can cause the browser to block. Creating operating system threads is generally quite slow, so the overhead would be significant compared to the average duration of these tasks. Rather than creating a new thread for each potentially parallel task, knowledge of the number of cores is used to create the optimum number of threads in advance. A lightweight dispatcher is then used to ensure the load is spread evenly across the pre-created threads. The operating system will then efficiently distribute these threads across the available cores.

Thread safety

Multithreaded code can be considered dangerous when each thread is able to access memory structures that are also being accessed from another thread. It is very easy to not consider every possible code path and the interactions with other threads accessing the same memory. Debugging this can be tricky.

In the case of image decode, decoding multiple images simultaneously is relatively safe since there is very little sharing of data apart from passing the final image between threads. However, multithreaded layout appears at first to be far more complex since all threads need to access the DOM tree, the styling structures and the render tree.

During the layout phase, the DOM tree and style information are largely constant. Ensuring tasks do not modify these unnecessarily avoids most threading problems. However, the render tree is where the layout code stores its calculations so must be updated in parallel. The first step of layout is the construction of this tree, and this identifies and marks every group of objects (flow) that must be processed sequentially, with each object only being a member of one flow.

Ensuring that a layout task only accesses objects from one flow and never accesses objects from other flows ensures thread safety. When processing the tree bottom-up, communication to signal that all child tasks have been completed can be achieved with an atomic counter. When this counter reaches zero, a task is scheduled to process its parent.

Using ThreadSanitizer[6] ('TSan') in a continuous integration build system over a period of several years, together with stress testing on a variety of machine loads, has ensured that Flow has matured with any threading issues being quickly addressed.

Performance testing

In a series of side-by-side performance tests comparing Ekioh Flow, Apple Safari (WebKit), Google Chrome (Blink) and Mozilla Firefox (Gecko) on a quad core MacBook Pro, Flow outperformed all three:

- In an extreme layout stress test involving the layout of over 70,000 paragraphs of text, Flow took just 4.6 seconds to complete compared with 10.9 seconds on Firefox, 17.6 on Safari and over a minute on Chrome.
- Similarly in extreme rendering stress tests, Flow achieved 60fps compared to 16.5fps on Firefox, 7.1fps on Chrome and 6.5fps on Safari.
- Running 'The Man In Blue'[7] animation benchmark (with 1000 particles), Flow continues to show a clear performance advantage achieving 54.5fps compared to 42.5fps on Safari, 29.5fps on Chrome and 16.5fps on Firefox.

Using desktop machines to gather comparison data from less extreme benchmark tests is more difficult as frame rate limiters can mask the true results. In some cases, such as 'The Man In Blue' animation benchmark, the

number of rendered objects can be increased to drive up the intensity of the test to avoid hitting the limiters. The default is 500 particles but the desktop testing above has used 1000 particles.

Taking the 'UI Applications' use case from Table 1, and extrapolating the results from the tests above, shows that Flow can be almost twice as fast as traditional browsers, as shown in Figure 3.

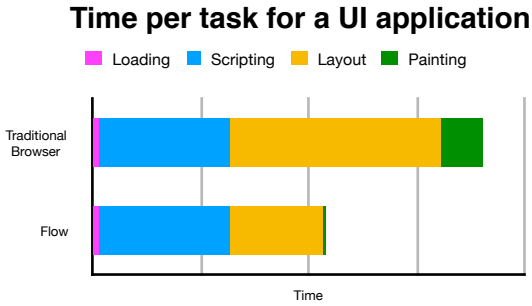


Figure 3

Testing on a quad core ARM based set top box (4 x 3K DMIPS per core) showed a similar pattern of results when compared against the box's default WebKit based browser:

- In the extreme layout stress test, Flow took 20 seconds compared 131 seconds for WebKit.
- Flow achieved 44fps in the extreme rendering stress tests compared to 5fps on WebKit.
- 'The Man In Blue' animation benchmark (with 500 particles), was somewhat closer, with Flow achieving 17fps compared to 12fps on WebKit.

Scalability testing of Flow's multithreaded layout, as shown in Figure 4, demonstrates that layout times continue to fall with each additional core and that significant benefits are achieved on both dual and quad core devices.

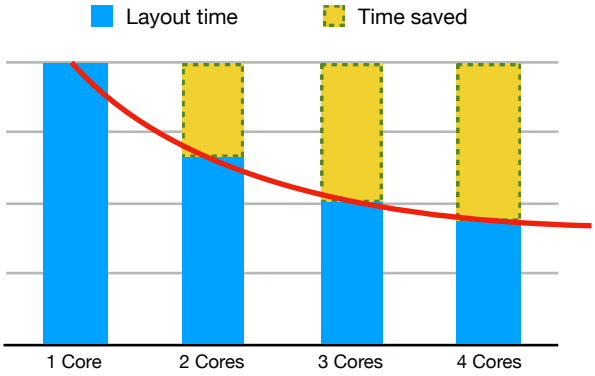


Figure 4

Summary

Flow shares many tried and tested components with the Ekioh SVG browser, ensuring millions of hours of deployment testing. Additionally, the product as a whole has been subjected to extensive automated testing ensuring that, whilst Flow is a totally new product, it comes to market deployment-ready.

Benchmarking has confirmed that Ekioh's parallel layout architecture efficiently scales with increased core count. When combined with GPU rasterisation, Flow provides a significant performance improvement over traditional single threaded browsers. This performance improvement makes 4K resolution UIs a practical possibility on high end embedded silicon today and it is projected that using Flow, the next generation of modern silicon will deliver animated full 4K UIs on mass market consumer products.

Designed initially for embedded and TV/set top box middleware applications, Flow does not yet boast the rich feature set of Ekioh's range of WebKit-based browser products. Additional features are being added as required to reflect the needs of each market adopting multi-core silicon.

About Ekioh

Ekioh designs and develops rendering software for a wide range of consumer products and resource constrained platforms. The company's products provide the high performance, compact footprint and flexibility necessary to address the wide range of application and UI rendering challenges presented by the consumer electronics and embedded systems industries. Best known for its TV and set top box solutions, Ekioh browsers are deployed in tens of millions of products in over 30 countries around the world.

Ekioh's engineering team brings together expertise in graphical systems, embedded software, set top box, TV silicon and robust software design. Using this combination of skills, fuelled by intense customer focus, Ekioh's product quality and customer support are second to none.

References

1. History of Processor Performance (Columbia University, 2012)
   http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf
2. Amdahl's law
   https://en.wikipedia.org/wiki/Amdahl's_law
3. Alexa Top 500 Global Sites
   http://www.alexa.com/topsites
4. Servo, the Parallel Browser Engine Project
   https://servo.org/
5. Fast and Parallel Webpage Layout (Leo A. Meyerovich & Rastislav Bodík)
   http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.207.1244&rep=rep1&type=pdf
6. ThreadSanitizer (TSan)
   https://clang.llvm.org/docs/ThreadSanitizer.html
7. The Man In Blue Animation Benchmark
   http://themaninblue.com/experiment/AnimationBenchmark/html/?particles=1000