What does the image decoding attribute actually do?

Category: Blog

This page was originally created on 26-Jun-2023 and last edited on 29-Jun-2023.

Introduction

I've been trying to figure out what this attribute actually does for a while now. Not full time of course (I'm not that sad! Honestly I'm not! Really!), but every so often I read another article where this comes up, or see advice to add this to massively boost image performance and I get curious again as to what it actually does.

To start let's see what the spec says:

> In order to aid the user agent in deciding whether to perform synchronous or asynchronous decode, the `decoding` attribute can be set on `img` elements. The possible values of the `decoding` attribute are the following image decoding hint keywords:
>
> | Keyword | State | Description |
> |---|---|---|
> | `sync` | Sync | Indicates a preference to decode this image synchronously for atomic presentation with other content. |
> | `async` | Async | Indicates a preference to decode this image asynchronously to avoid delaying presentation of other content. |
> | `auto` | Auto | Indicates no preference in decoding mode (the default). |

Err... thanks for the technical explanation. But what does this actually mean in real life? Which setting should you use? Does it even matter? And if it does, why don't those clever browser engineers just set it to the best setting?

Well recently I ranted on Twitter about this (as I am often want to do!) in a long thread that really should have been a blog post. So here is that blog post.

Some misconceptions

First up let's get some misconceptions out of the way, that I see all the time about this attribute:

No, images do not block rendering of contents that follow it

No, images in your HTML are not rendering-blocking and you don't need to add this magic attribute to make the stuff below images load.

I see articles saying that in this code:

```
<p>some intro text</p>
<img src="big_and_expensive_img_to_load" />
<p>text with very important info for the user</p>
```

The bottom paragraph will not be shown until the image is loaded and decoded, and that setting `decoding=async` will magically prevent that delay:

```
<p>some intro text</p>
<img src="big_and_expensive_img_to_load" decoding="async" />
<p>text with very important info for the user</p>
```

Seriously, that's just not how browsers work! How many times have you seen a page load without images being there yet? Loads of times right? Images are not typically render-blocking and if they were the web would be a very slow and painful place to be.

I'm not going to link to the article I pulled this example from, but it's the first one that comes up when you search for "what does decoding=async do" so that's depressing. Hopefully this post will displace that if I pray to the SEO Gods enough.

No, browsers don't decode images on the main thread

Modern browsers all decode images off the main thread, and have done so for a while now, leaving it free for other stuff. And any older, or more simpler, browsers out there that do still decode on the main thread are almost certainly not going to support this attribute. So, in theory, you are not going to free up the main thread with this attribute.

Maybe this wasn't the case when the attribute was originally proposed (in which case it would have been more important than it was now), I'm not sure, but browser engineers tell me it's definitely not the case now.

However, it might *appear* like you will block the main thread... because even if it won't block the main thread itself, it might block *rendering*.

What's that you say? What's the difference? Well the main thread is where all the critical stuff happens in browsers - all your JavaScript, and also lots of browser processing to layout the page and stuff. Doing it in one place makes lots of nice things on the web work a lot simpler than if there was more parallelisation. But the downside of it is that hogging the main thread for any intensive stuff is very frowned upon and leads to serious performance issues. So if you've an expensive calculation (say decoding an image!) then ideally you don't want to do it on the main thread and want to offload it to its own thread, or if you can't do that, then chunk it up and allow other critical processes to get in on some main thread time. Browsers realised this a while back and so moved image decoding off the main thread leaving it free to run all that JavaScript you love to add to your pages, or all the other stuff.

But after you do all your lovely processing, then you will likely want to update your page. So for image decoding you'll want to display the image. If you hold up *all* rendering (which is what `decoding=sync` can do), then some might say you have effectively blocked the main thread. They're wrong as other stuff can often happen in parallel, but if the effect of that other stuff processing can't be displayed, then it can appear the main thread isn't doing what it's supposed to.

So it might be a bit of a pedantic point, but given how critical the main thread is to do other stuff, I still think it's important. Plus I'm a pedant. If decoding happened on the main thread, and then that other stuff had to happen after and only then it could be rendered it would be even slow. Anyway, I'll get back to why this often isn't as important a thing (or that it might be!), later...

However, even after all that pedantry, for images in the main screen we shall see they *can* also actually block the main thread, even if the decoding is happening off the main thread! It's a little complicated so we'll get back to this later.

No, the decoding attribute doesn't make your image decode faster or slower

Adding this attribute will *not* make images display faster (though there's some nuance here, mostly about JavaScript-inserted images). This attribute is about allowing *other* content to potentially display faster (including other images - so there's the nuance).

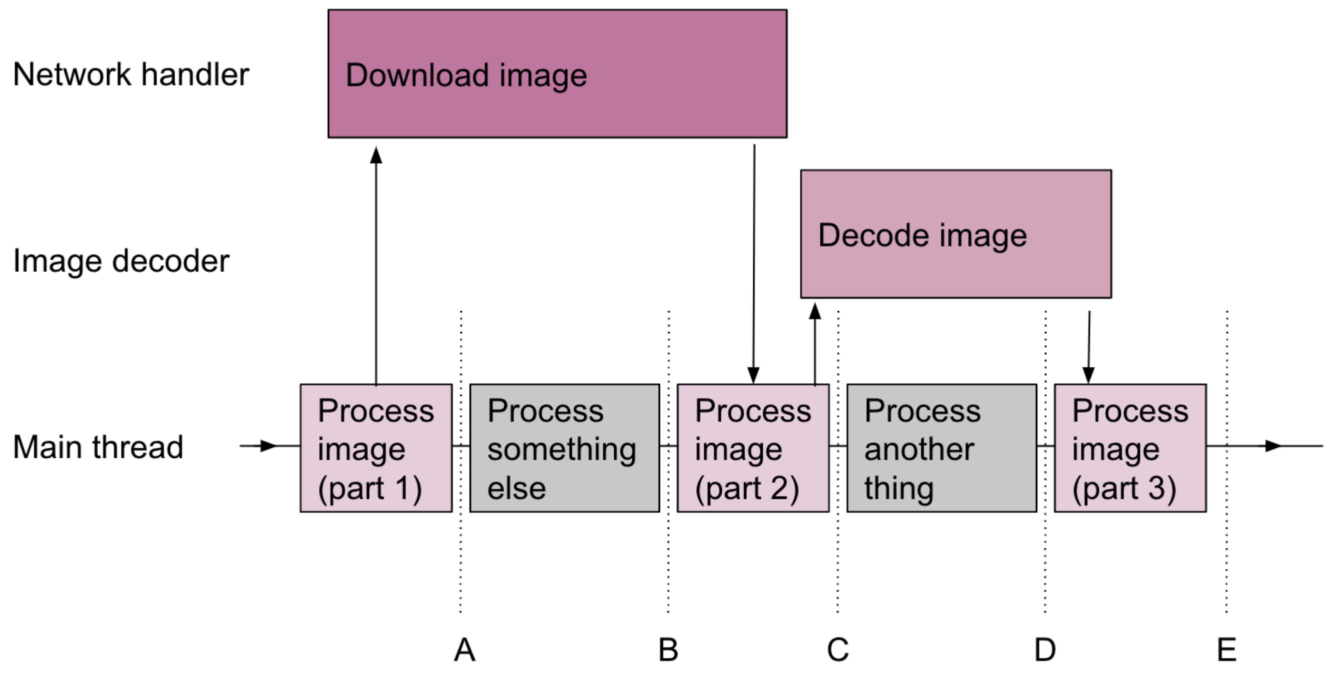
And similarly it doesn't "defer" decoding to later making images slower. *Side note: browsers actually do defer decoding until the image is in, or near the viewport. Decoded images are large, so browsers won't decode all images on a page, until it needs to. Similarly they can discard decoded images to save memory if they aren't used for a while and will re-decode them if needed again.*

The decoding time will be the same no matter what you set this to.

So what does the decoding attribute actually do?

This attribute says whether you want other work for the next render to wait for the image to be ready and to be included in it or not.

Take a look at this simplified diagram (caveat it's *very* rough and is not to scale as decoding is typically much shorter than downloading):



The parser is happily processing the document on the main thread. It sees an `<img>` element, so it needs to download that. That also doesn't happen on the main thread either btw, it just asks the Network chappy to deal with that, and happily processes some more content. After the network fetch is done, the main thread can get the image and see it needs decoded so passes it off to another decoder thread to deal with that. And it happily carries on and processes another thing. Once the decoding is done it's finally got the full image ready to render (which might involve another bit of main thread processing, or might not, depending on... things).

Rendering can happen when there is content ready to display. That can happen at **A** (we can add the empty image element), at **B** (we can display whatever was the outcome of "Process something else"), at **C** (if the image download failed we can display a broken image for example), at **D** (we can display whatever was the result of that "Process another thing"), or at **E** (We can finally display the image).

The `decoding` attribute lets us decide whether rendering can happen at **D**, or whether we should wait until **E** to render the image (that we have, but haven't decoded) at the same time as the other content that's ready to render.

The image will still take the same time to download and display (in general - nuance coming later!). The image will still use the same amount of the main thread's time. Other things can still process while all this is happening. It's all a matter of whether we should sync up the "process another thing" changes with the image display or not.

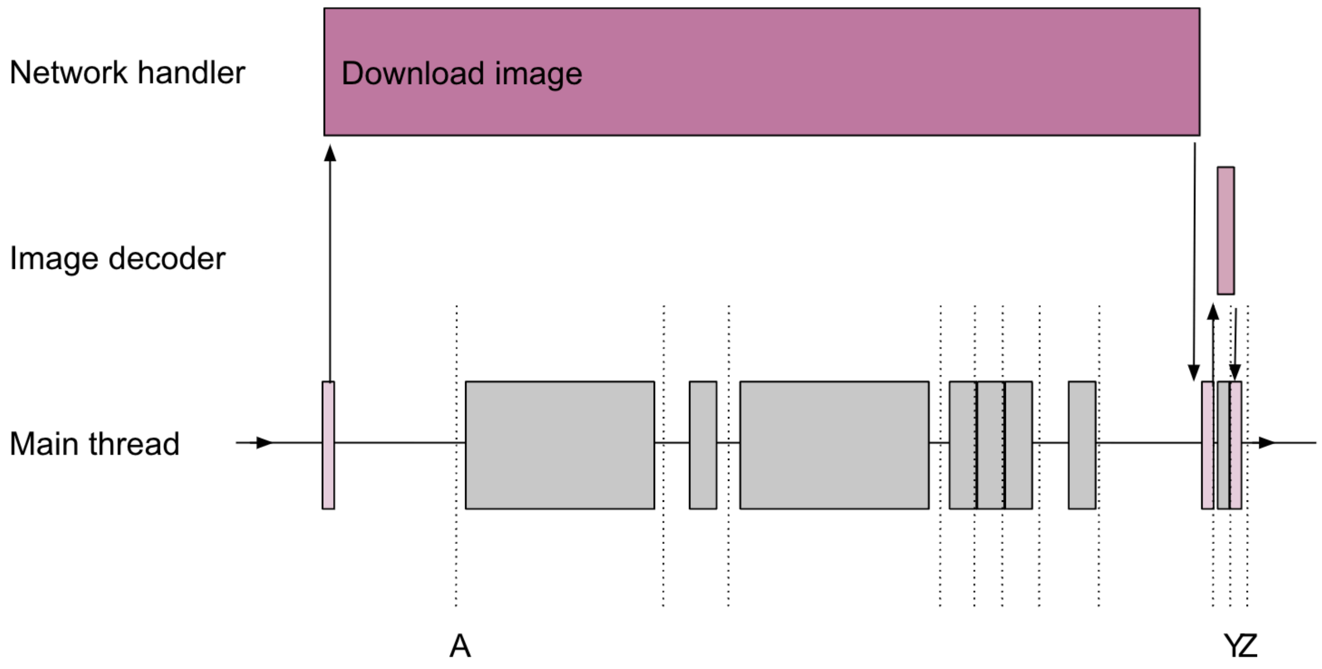
So does this make a difference as an HTML attribute?

In the above example, probably not that much. The image is semi-independent. It's being fetched from the network (or the cache) and that time isn't guaranteed. Then it's decoded. So the relationship to the content in "process something else" and "process another thing" isn't guaranteed anyway. The content that was a result of "process another thing" could be rendered while the image downloading is happening for example if that download took a bit longer. So, given there aren't any guarantees to the ordering here, syncing these updates isn't that important to be honest.

What potentially *is more important* is not to block the other content from displaying if the decoding is quite lengthy. Why shouldn't we display the results of "process another thing" earlier instead of making it wait since I've just said it's independent?

So maybe we *should* all use `decoding=async` for our images and get a slight performance boost for our other content? Look at the time difference between **D** and **E** - that's noticeable!! Stop whinging Barry and just accept this is better and all the advice is right!

Well maybe, but as I said earlier the above diagram is not to scale. In reality it usually looks much more like this:



And even that's exaggerating the scale a little. Now the difference between Y and Z is less impressive.

In fact it's even less of an issue than that, as most in-viewport images are progressively rendered - yes not just progressive JPEGs. Non-progressive images are rendered progressively top to bottom, rather than blurry to sharp. So in reality the decode time will be even smaller for these chunks of images. There also often may not even be any other content to display between each of these smaller chunks depending what else is happening.

However, for reasons we'll come back to later, that really only affects on-screen images loading at the same time as the page. Images that are already loaded (for example, being fetched from the cache), or off-screen images are not decoded immediately, meaning the progressive nature of them loading isn't as relevant. We'll return to this at the end...

So larger images, maybe it is worth it. But if you've large images that take time to decode, they also will take a lot longer time to download, so you've likely got bigger issues anyway. Those interested enough in performance to consider the `decoding` attribute really should instead spend time seeing if they can get the image smaller in my opinion!

Still, not every developer is in control of the images their users upload to the site. Or if you're a CMS or framework developer then adding this attribute *might* have some small impact, so why *not* just set it on every `<img>` element? Are there any downsides?

Are there any downsides to using `decoding=async`?

For <img> elements in the original HTML src, there don't really seem to be many downsides. I argue not as many upsides either, but if no downsides then why not just do it? This is why we see platforms like WordPress have added this by default, and also many framework image components (NextJS example) have also added this (full disclosure: I work at Google with many of the people who worked on adding these!).

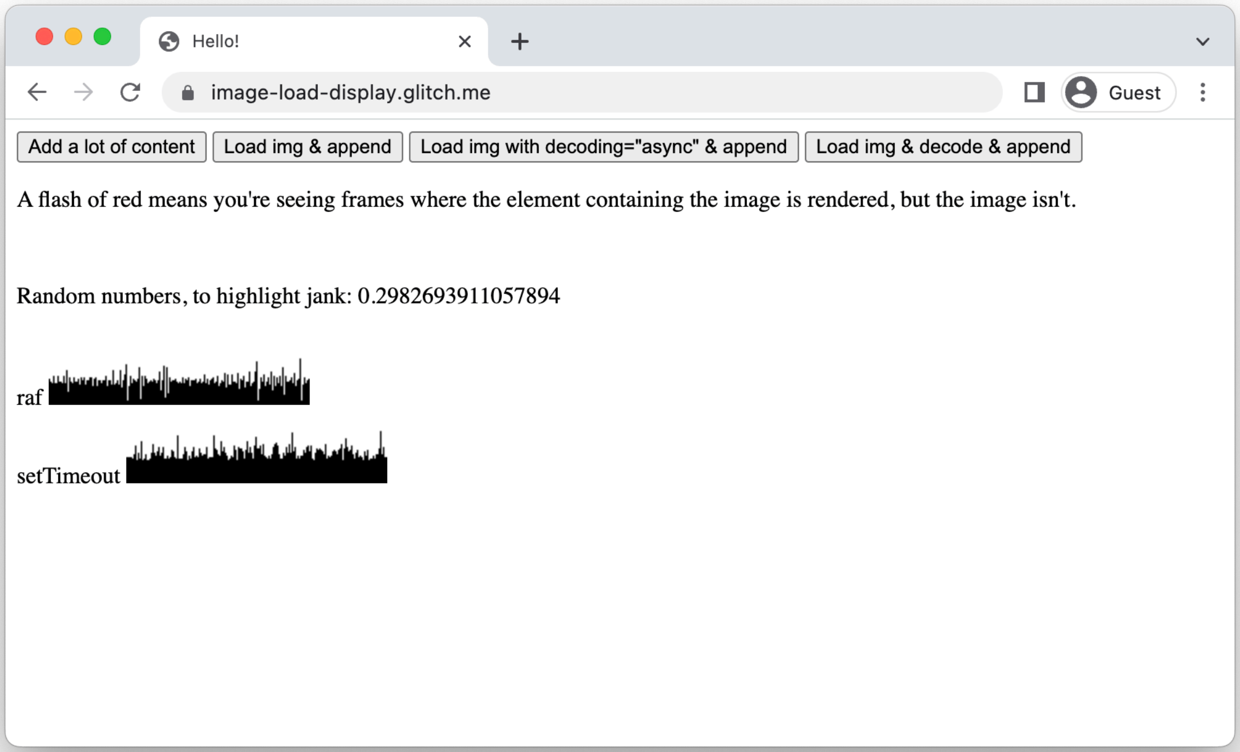
OK so why haven't browsers just changed the default anyway, even if the gains won't be noticeable in most cases? Seems obvious to just do it right? Well, many smarter people than me have argued for that, but there's a catch and, as in many things on the web, it involves our favourite little addiction on the web - JavaScript.

Images are not just in the source of HTML, they can also be dynamically added to the DOM using JavaScript. And here the differences in decoding can be more noticeable in those cases.

A demo of when you can see differences

Jake Archibald created this lovely little Glitch demo where finally we *can* see some differences. But it's a little contrived (no offence to Jake - it has to be somewhat contrived to demo this as nearly always the difference is imperceptible). It also takes a little explaining to understand what's going on. We're also going to see that both `sync` and `async` have their downsides and neither is actually that good... and that there is a better way!

So here's the Glitch:


Hello!
image-load-display.glitch.me
Guest
Add a lot of content
Load img & append
Load img with decoding="async" & append
Load img & decode & append
raf
setTimeout

A flash of red means you're seeing frames where the element containing the image is rendered, but the image isn't.

Random numbers, to highlight jank: 0.2982693911057894

There's some buttons at the top, a little explanation text and a random number generator that's continually ticking away, and also with a `raf` and `setTimeout` updating graph (that he added recently as we worked through this post!).

That random number generator and graphs below that are there to show jank - if rendering is blocked then you'll see it momentarily pauses their updates, and one or both graphs show a little spike. Would you have something that makes delayed rendering updates as obvious on your site? Maybe, maybe not - as I say it's a little contrived. But some pauses in animations or immediate feedback can be jarring even if you only notice them subconsciously. Anyway, we'll use the random number generator and the graphs to make them more obvious.

So we'll skip the first button and come back to that. The next one ("`Load img & append`") uses some JavaScript to load an image, and then add it to the DOM without using the `decoding` attribute (so it uses the browsers default for this setting - which is `sync` for Chrome and Safari and `async` for Firefox btw - see even the browsers can't agree which is the best option!).

The JavaScript is not a simple `<img>` insertion but is also a little contrived (really Jake, I'm not trying to insult you! I swear!):

```
btnAppend.addEventListener('click', async () => {
  reset();
  const img = await loadImg(getCacheBustedUrl());
  const div = document.createElement('div');
  div.className = 'img-frame';
  div.append(img);
  document.body.append(div);
});
```

So what it does is:

- Reset the demo (don't worry about this)

- Awaits the load of an image with a cache-busting URL (note I've not shown these function definitions so just trust me on what it's doing - or go look at the demo's source).

- Then it creates a `div`, with a class of `img-iframe` (which has a background colour of red btw - that will be important in a minute).

- Then it adds the image to the `div`.

- Then it adds the `div` (containing the image) to the body.

What happens with these browsers depends on your browser as they have slightly different defaults For Chrome, as it uses `sync`, so if you click the button on that, the next rendering is held back. This means two things. First the `div` and the image are not shown until the image is decoded. This prevents an empty image (yeah!), but also holds back *all* rendering - including the rendering of the unrelated random number generator. So when you use this button you'll notice a slight stalling of this updating - something we call jank. This happens even on really fast Mac Book Pros.

So this is why `sync` is bad when you have other updates you want to display.

Is async any better?

Button number two does basically the same thing, but also sets the `decoding` attribute to `async` to try to avoid this jank:

```
btnAsync.addEventListener('click', async () => {
  reset();
  const img = await loadImg(getCacheBustedUrl(), { async: true });
  const div = document.createElement('div');
  div.className = 'img-frame';
  div.append(img);
  document.body.append(div);
})
```

As you can see it is the same code except it passes `async: true` to the `loadImg` code which then sets the `decoding` attribute:

```
function loadImg(url, { async = false } = {}) {
  return new Promise(resolve => {
  const img = new Image();
  img.onload = () => resolve(img);
  if (async) {
    img.decoding = 'async';
  }
  img.src = url;
  });
}
```

Note: this is the mode Firefox works in by default so there's no difference between these buttons for Firefox. To explicitly test `sync` in Firefox (and occasionally for Safari which sometimes uses async for certain limited sets of circumstances), I remixed Jake's demo to add the `sync` button option.

So when you click this button now the random number generator does not pause momentarily - there is no jank. Yeah!!!

However, there is a flash of red that you might have noticed. That wasn't there before! What's going on? Well, since we've now said we do **not** need to wait for the image, the browser has gone ahead and drawn the `div` (which as I said about has a red background thanks to the `img-frame` class), and then a moment later it draws the image after it's decoded. It doesn't draw an empty image placeholder as the image is not empty - it's just not ready to draw yet.

So this is why async isn't always good either.

And this is why browsers haven't agreed on which value is the best. There are downsides to both!

So neither is the right answer!?!?

Yup, as always... it depends. Personally (and I'm likely very biased here!), as a default, I prefer Chrome's `sync` default. Jank is not good, but the flash of red just looked actually wrong so I see sync as a more cautious approach.

However, there is an even better way in JavaScript, and that's the third button ("`Load img & decode & append`"), which uses neither:

```
btnDecode.addEventListener('click', async () => {
  reset();
  const img = new Image();
  img.src = getCacheBustedUrl();
  await img.decode();
  const div = document.createElement('div');
  div.className = 'img-frame';
  div.append(img);
  document.body.append(div);
});
```

The key here is the `await img.decode();` line. That explicitly decodes the image (it also loads it if not already loaded, hence why we don't need a load call here). It decodes the image off the main thread as usual, but waits for that to complete in a non-blocking fashion (that's the `await` bit!). This means the decoded image is what's inserted into the DOM and so it has the best of both worlds - no decoding to be done at render time, so no jank, and no flash of incorrect content! Yeah!

So this is the best option when inserting images into a page. This solved a real problem in Google Image search where clicking on an image in the result page caused a higher resolution image to be downloaded and swapped in. On Firefox (`async` default) this caused a visible flash as the old image was removed, but the new one wasn't decoded yet. With this explicit `decode()` call, that no longer happens.

So JavaScript users, where the difference between `decoding=sync` and `async` is the most noticeable, should use neither and should instead do an explicit `decode()` call in my opinion. So `decoding=async` ain't that useful here either!

However, it should be noted that browsers will only hold the decoded image for a *very* short period of time (basically until the next frame for Chrome). Decoded images are large and therefore are memory hogs, so browsers discard them if they are not needed. This means that using this technique for off-screen images is a bit wasteful as you'll need to decode it again later.

Scrolling adds another dimension

The other button that we didn't look at in Jake's demo is the first button ("`Add a lot of content`"). What it does is add a load of content (who'd have thought!), and this is useful to show when browsers decode off-screen images.

What you sometimes notice is that first of all just adding the content adds a small jank. So it's not just image decoding - any processing if it requires enough processing can hold up rendering. In this case, if you think back to the diagram above, it would be one of the grey blocks (the browser layout process), blocking the main thread and so pushing out the random number generator's ability to get on the main thread and update.

Anyway, clicking on the second "`Load img & append`" button after the content is added, will not block the random number generator from updating as it did before without this content. This is because off-screen images are often not decoded right away. As I said above, decoding images does have a processing, and perhaps more importantly, memory cost so the browser is clever enough not to do this until it needs to render it. If you browse away from the page it will have avoided that needless processing and memory time.

However, if you then press `Page End` button (or `Command+down` if you don't have an explicit key) then you'll notice the jank showing the decoding hasn't happened until then. Similarly using the async version will show the flash of red. So ideally you'd avoid this for off screen images with the correct choice of `decoding=sync` or `async`, depending on which you prefer. Using the third option of an explicitly decode is not as useful for off screen images as the decoded image will be discarded as discussed above.

What about scrolled images in HTML source?

This brings us back out of JavaScript and right round to the original use case of `<img>` elements in the original HTML source - back to where we started. In that case the image being downloaded piecemeal and displayed progressively doesn't lead to many small decode operations. The full image may be loaded before decoding and so users would get the full hit of the decoding once it's needed.

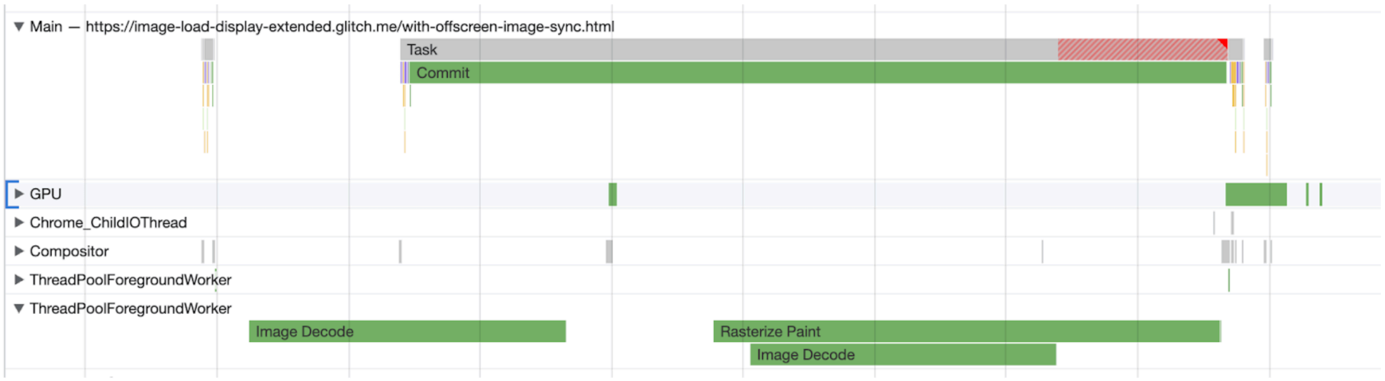
This will also likely be the case with cached images. And also lazy loaded images which hopefully will have fully downloaded in many cases before they come on-screen. Similarly browsers may also discard decoded images if they are sufficiently off-screen and re-decode them when they start to become on-screen again.

So you think that would dampen my argument about the usefullness of this attribute - surely we'd want `decoding=async` to avoid any rendering delays, however small and avoid that jank? Well, that's where it gets interesting, and to understand that we need to delve into when it does and does not block the main thread and rendering.

So does image decoding block the main thread or not?

We touched on this earlier, and the answer is not quite as obvious as you may think. Image decoding happens off the main thread so you'd think the answer would be: no, it doesn't block the main thread, but that's not strictly true!

Here's a performance trace of a page which loads an offscreen image with decoding=sync explicitly set. I let the page load let it settle down, then started the trace, and clicked and pressed `Page End`. Below is a zoomed in view of what happens:

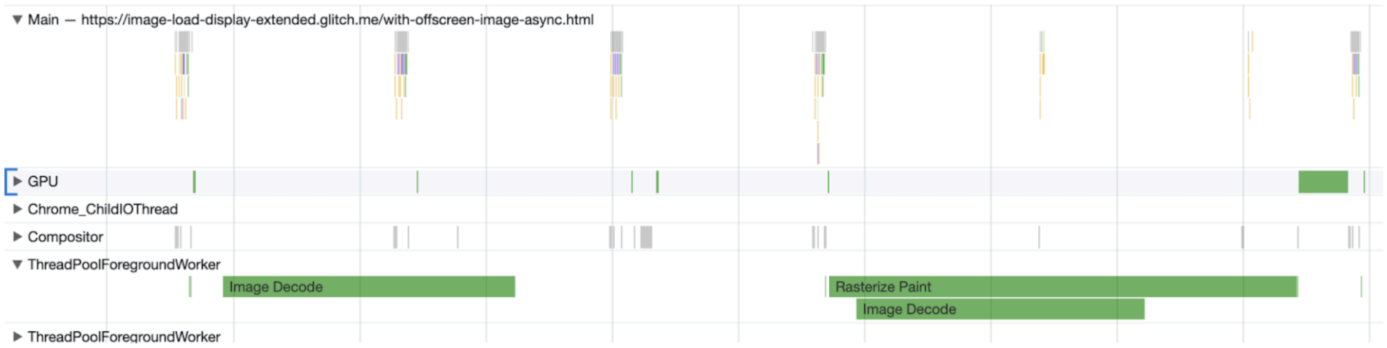

Main — https://image-load-display-extended.glitch.me/with-offscreen-image-sync.html
Task
Commit
GPU
Chrome_ChildIOThread
Compositor
ThreadPoolForegroundWorker
ThreadPoolForegroundWorker
Image Decode
Rasterize Paint
Image Decode

Hmmm, while decoding does indeed happen off the main thread, for `sync`, there is a blocking `Commit` which creates a long task, before the rasterizing paint is completed. That's not good!

So yes the image decoding is off the main thread, but the next time the browser tries to paint anything (including the counter update), the main thread is blocked as we have said to `sync`hronise the display with the decoded image. Which kinda defeats the point of decoding the image off the main thread in my opinion! Ideally it would block the paint, but not by blocking the main thread so other intensive stuff could still happen (bug raised, and a more detailed one from Michal Mocny).

Note that in this example, we are using a deliberately large image (1.4 Mb) and even then the blocking time is "only" 60ms, so for smaller images it may not even be noticeable, but then again for slower devices it could be even more noticeable. Plus I've seen a lot bigger than 1.4 Mb images on the web!!

Doing the same thing with `decoding=async` on the image doesn't show this large `Commit` and the main thread is not blocked at all:


Main — https://image-load-display-extended.glitch.me/with-offscreen-image-async.html
GPU
Chrome_ChildIOThread
Compositor
ThreadPoolForegroundWorker
Image Decode
Rasterize Paint
Image Decode
ThreadPoolForegroundWorker

What's perhaps even more interesting, is if the image is scrolled into view - rather than using `Page End` to immediately show it into view. In that case, even with `decoding=sync`, the browser decodes it ahead of time so no longer blocks with that large `Commit`. It no longer needs to sync the image (since it's still off-screen when decoding happens) so can decode with no main thread consequences. In this case `decoding=async` and `decoding=sync` basically act the exact same way.

You can repeat the same tests using Firefox and Safari (using the counters, or a `console.log` counter I added for those last two tests) and you see the same thing). It's not quite as obvious as in the performance trace in Chrome, but I'm not familiar enough with those browsers' performance profiling tools to do the same there, hence why I used those, but it seems like all three browsers handle this the same.

So are images main thread blocking? As always, it depends. Technically "no", but in reality "yes" for on-screen images.

Should browsers change the default to async?

So, using `decode=async` could help performance by not blocking other content and also by avoiding main-thread blocking `Commit` tasks, particularly for on-screen large images. So maybe we should be using it more? But if that's the case shouldn't browsers just change the default so we don't have to do this? Others, far cleverer than I, have suggested this a few times before.

However, we've shown that for the JavaScript use case that could cause unexpected flashes - something Safari/WebKit noticed before when they tried to move to a more `async` default. Persuading any site depending on the current (at least Chrome) default to move to move to explicit `decode()`, or using `decoding=sync` would be a challenge, because even if it doesn't seem like it, the browser vendors try very hard not to ship breaking changes like this without being sure it won't cause these sorts of problems.

Maybe, even if we can't change the default all the time, it's possible to use heuristics (see suggestion here from Jake) to apply `async` by default more often (for example in HTML source case, but not the JavaScript-inserted case), but as you can see from the replies there even that gets tricky.

I still come back to the fact that in most cases it will likely not be that noticeable, so not sure it's worth the effort to make a potentially breaking change. Though I'm not loving the fact the browser's differ here, which is confusing to developers.

Conclusion

So that's finally an in-depth look at what this attribute does, and how setting it *may* have some performance gains in certain circumstances. But it isn't a magic speed up that most will even notice.

For on-screen images it might help a little though you're probably often getting smaller progressive decodes a lot of the time. For off-screen images it will often decode without blocking anyway so no difference, and for JavaScript-inserted stuff there's probably a better way anyway. So, yes this attribute might help, but in many cases it won't help noticeably (if at all).

Perhaps one of the biggest longer term takeaways from this (from me at least!) is that we need to get better at documenting things we add to the web platform. I'm biased on this as it's part of my job, but if we don't explain what a feature really does and the use cases for it, then we can't really be surprised when it's misunderstood. Hopefully this post helps demystify it a bit, and we're also working on updating the documentation for this atribute on MDN.

I'm still ending up back at, it's an attribute that most developers don't need to worry about for all the reasons given above. It maybe has some small performance gain but I think in most cases it won't be noticed. It's nice that it exists for the edge cases where it matters or for the advanced people who know what it does and want to squeeze every last bit of performance out, but even in most of those cases, there are better options.

So use it on your `<img>` elements if you want. It's maybe good that image components for libraries use this, or that platforms like WordPress set it by default but if you haven't used it on your site (like I haven't on this blog), then don't expect it to magically speed up your images to a noticeable degree. Other attributes like `loading=lazy` (on offscreen images only please!), and `fetchpriority=high` (on important images only!) will have a *much larger* impact. As will ensuring your images are not so big that decoding times become a problem. So prioritise those first before worrying about this micro-optimisation.

Huge thanks to Jake Archibald who did most of the work that led to this post as well as reviewing it too for me (we miss you at Google Jake!), and also to Michal Mocny for also reviewing it. Remaining mistakes and misunderstandings in this post are mine not theirs.

What do you think?

27 Responses

Upvote Funny Love Surprised Angry Sad

2 Comments

1 Login ▾

G

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Share

Best **Newest** Oldest

Joseph Scott

2 years ago

Excellent article, thank you for the detailed write up. The level of nuance involved here does indeed make this more difficult than other traditional performance techniques. I especially appreciate the suggestions around how to do better when injecting images via JavaScript.

I think there is a typo in one part. The line `becode the rasterizing paint is completed` is probably supposed to be `because the rasterizing paint is completed`.

0 0 Reply

Barry Pollard Mod → Joseph Scott

2 years ago

Fixed to "before the rasterizing paint is completed."

1 0 Reply