Bringing Parallelism to the Web with River Trail

Motivation and Background

The goal of Intel Lab's River Trail project (Parallel Extensions for JavaScript*, code named River Trail) is to enable data-parallelism in web applications. In a world where the web browser is the user's window into computing, browser applications must leverage all available computing resources to provide the best possible user experience. Today web applications do not take full advantage of parallel client hardware due to the lack of appropriate programming models. River Trail puts the parallel compute power of client's hardware into the hands of the web developer while staying within the safe and secure boundaries of the familiar JavaScript programming paradigm. River Trail gently extends JavaScript with simple deterministic data-parallel constructs that are translated at runtime into a low-level hardware abstraction layer. By leveraging multiple CPU cores and vector instructions, River Trail programs can achieve significant speedup over sequential JavaScript. The River Trail API is being considered for standardization by the ECMA TC39 committee. You can to read the API documentation and follow progress on standardization at [4]. You are also invited to get involved and join the discussion on ES-Discuss [5].

An open-source prototype implementation of River Trail for Firefox* is available at [1] and instructions for installing it are available at [2]. This prototype implements a variant of the full River Trail API that compiles down to OpenCL* and can be executed on CPUs and GPUs.

This tutorial is a gentle introduction to the River Trail language extensions and API. Note that this tutorial describes the River Trail API as implemented by this prototype and *not* the strawman API being considered for standardization.

The *ParallelArray* type

The central component of River Trail is the *ParallelArray* type. ParallelArray objects are essentially *ordered collections of scalar values* .

Multi-Dimensional and Uniform

ParallelArray objects can represent multi-dimensional collections of scalars. All ParallelArray objects have a *shape* that succintly describes the dimensionality and size of the object. The shape of a ParallelArray is defined as an array of numbers in which the value of the ith element is the number of elements in the ith dimension. So a 4x5 matrix of numbers can be represented as a ParallelArray object whose shape is [4, 5]. Similarly, a 2D image in which each pixel has RGBA values can be represented as a ParallelArray object with shape [height, width, 4].

Multi-dimensional ParallelArrays are also required to be *uniform* (also called *rectangular*). That is, the length of all inner arrays in a particular dimension must be the same. For example, `<<0, 1>, <2>, <3, 4>>` would be a non-uniform ParallelArray and is not allowed.

Immutable

ParallelArrays are immutable once they are created. ParallelArrays are manipulated by invoking methods on them which produce new ParallelArray objects.

Constructors

```
 1  // Create an empty Parallel Array
 2  var pa0 = new ParallelArray();
 3  // pa0 = <>
 4
 5  // Create a ParallelArray out of a nested JS array.
 6  // Note that the inner arrays are also ParallelArrays
 7  var pa1 = new ParallelArray([ [0,1], [2,3], [4,5] ]);
 8  // pa1 = <<0,1>, <2,3>, <4.5>>
 9
10  // Create a ParallelArray from another ParallelArray
11  var pa2 = new ParallelArray(pa1);
12  // pa2 = <<0,1>, <2,3>, <4.5>>
13
14  // Create a ParallelArray from several other
15  // ParallelArrays
16  var pa3 = new ParallelArray(<0,1>, <2,3>);
17  // pa3 = <<0,1>,<2,3>>
18
19  // Create a one-dimensional ParallelArray of length
20  // 3 using the "comprehension" constructor
21  var pa6 = new ParallelArray(3,
22              function(i){return [i, i+1];});
23  // pa6 = <<0,1>, <1,2>, <2,3>>
24
25  // Create a two-dimensional ParallelArray with shape
26  // [3, 2] using the comprehension constructor
27  var pa7 = new ParallelArray([3, 2],
28              function(iv){return iv[0] * iv[1];});
29  // pa7 = <<0,0>, <0,1>, <0,2>>
30
31  // Create a ParallelArray from canvas.
32  // This creates a PA with shape [w, h, 4],
33  // corresponding to the width and height of the
34  // canvas and the RGBA values for each pixel.
35  var pa8 = new ParallelArray(canvas);
36  // pa8 = CanvasPixelArray
37
```

ParallelArray objects can be created in a variety of different ways as listed below. The constructor form in line 7 creates a new ParallelArray object `pa1` out of an existing JavaScript array. We could have also explicitly created the JavaScript array separately and used it to create `pa1` . The form in line 11 shows how to create a new ParallelArray object `pa2` from another ParallelArray object `pa1` . Since ParallelArrays are immutable, it does not matter to us at this point whether this creates a copy of `pa1` or whether it simply returns a new reference to `pa1` . We are guaranteed that both `pa1` and `pa2` will have the same structure and the same data in them for their lifetime. The constructor in line 16 returns a new ParallelArray object created by putting together two other ParallelArray objects. The arguments <0, 1> and <2, 3> each have the shape [2]. The new ParallelArray object will therefore have the shape [2, 2] - it contains two elements each of which contain 2 elements. We can also create

ParallelArray objects using a comprehension constructor shown in line 21. We specify the length of the ParallelArary we want (3), and an elemental function. As the name suggests this elemental function is invoked for every index `i` within the length of the ParallelArray, i.e., for i=0, i=1 and i=2. This function then returns a 2 element array consisting of i and i+1. After the elemental function has been invoked for every i, we have the resulting new ParallelArray <<0, 1>, <1, 2>, <2, 3>>. Note that the order in which the elemental function is called is irrelevant. We will learn more about writing these functions in the next section. The comprehension constructor can also create multi-dimensional array as shown in line 27. We simply need to supply a shape vector instead of a length and an elemental function that takes a vector index as an argument. In this case, the shape we specify is [3, 2] and the elemental function will be invoked with a 2-element vector argument iv. Finally, we can also create a new ParallelArray object directly from an HTML5 canvas object. The canvas object is used for drawing 2D shapes, pictures and video on a webpage. We will see how this is useful later when we build a video web app.

Methods

ParallelArray objects created with the above constructors come with several methods to manipulate them. These methods typically produce a new ParallelArray object (except the *reduce* method which produces a scalar value).

Map

The first method we will discuss is *map*, probably the most prominent and well known data-parallel skeleton. The *map* method expects a function as its first argument that given a single value produces a new value as its result. In the following, we will call such function *elemental function* as it is used to produce the elements of a ParallelArray object. The *map* method computes a new ParallelArray object out of an existing ParallelArray object by applying the provided elemental function to each element of the source array and storing the result in the corresponding position in the result array. Let us look at a simple example: increment

```
var source = new ParallelArray([1,2,3,4,5]);
var plusOne = source.map(function inc(v) { return v+1; });
```

First, we define a new ParallelArray object `source` that contains the numbers 1 to 5. We then call the *map* method of our source array with the function `inc` that computes the one increment of its argument. Thus, `plusOne` contains the values 2 to 6. Also note that `plusOne` has the same shape as the original array `source`. The *map* method is shape preserving.

As you may have noticed, the *map* method does not provide an index to the elemental function it calls. We refer to this as *index free* computations. Not using an index has the advantage that no indexing errors can be made. However, this added simplicity comes at the cost of expressiveness. With map, one can not inspect neighboring values, as commonly required for convolution style codes.

Combine

The *combine* method addresses this issue. Similar to map, *combine* can be used to compute a new ParallelArray object by inspecting an existing ParallelArray object's element. Other than *map*, the elemental function of *combine* is provided with access to the current index in the source array, along with a reference to the source array itself.

Let us revisit the increment example from the previous section. When using *combine*, increment can be expressed as follows:

```
var source = new ParallelArray([1,2,3,4,5]);
var plusOne = source.combine(function inc(i) { return this.get(i)+1; });
```

As before, we first produce our source array holding the values 1 to 5. We then apply the *combine* method using a slightly modified version of the `inc` function. It now expects an index `i` as argument. Furthermore, the collection, i.e., the source ParallelArray object is bound to the variable `this` within the elemental function's body. We then compute the one increment of the value at index `i`. This element is computed by calling the *get* method of the source ParallelArray object with index `i` as argument.

As this example shows, using *combine* requires more code to implement increment. However, we have gained expressiveness. As an example, consider the following implementation of *reverse*:

```
var source = new ParallelArray([1,2,3,4,5]);
var reverse = source.combine(function rev(i) {
    return this.get(this.length-i[0]-1); });
```

In the elemental function `rev` we exploit the access to the index to compute the reversed index in the source array. Note that computations are driven by the index position in the result, not the index that is read. We therefore use the expression `this.length-i[0]-1` to compute the source index of the reversed element for position `i` in the result array. This code makes use of the *length* property of the ParallelArray object that, similar to JavaScript's Array object, gives the number of elements in the array.

It is important to note here that the index `i` is not a scalar value but actually a vector of indices. In the above example, we therefore have to use `i[0]` in the computation of the source index. So far, all our examples computed on vectors and the use of an index vector in *combine* was of no help. However, ParallelArray objects in River Trail can have multiple dimensions. The *map* method always operates on the outermost dimension only, i.e., on the dimension that corresponds to the first element of the shape vector. With *combine*, the programmer can choose how deep to traverse. For this, an optional first argument to the *combine* method is used. As an example, let us generalise the above reverse operation into a transpose on matrices:

```
var source = new ParallelArray([4,4], function (iv) {
    return iv[0]*iv[1]; });
var transpose = source.combine(2, function rev(iv) {
    return this.get([this.getShape()[0]-iv[0]-1,
    this.getShape()[1]-iv[1]-1]); });
```

We use a depth of 2 and, consequently, the index vector `iv` passed to the elemental function contains two indices, corresponding to the two outer most dimensions of the source array. We also use the *getShape* method, which is the multi dimensional counterpart to *length*: It returns a vector that gives the length for each dimension of a ParallelArray object. With `this.getShape()[0]-iv[0]-1` we thus compute the index at the transposed position within the source array for the first dimension. Note here that *get* also accepts an index vector as argument.

Reduce

So far we have concentrated on parallel patterns that produce a new array out of an existing array. The *reduce* method implements a further important parallel pattern: reduction operations. As the name suggest, a reduction operation reduces the elements from an array to a single result. A good example to start with is computing the sum of all elements of an array:

```
var source = new ParallelArray([1,2,3,4,5]);
var sum = source.reduce(function plus(a,b) {
    return a+b; });
```

As the example shows, the *reduce* method expects an elemental function as first argument that, given two values as parameters, produces a new value as its result. In our example, we use `plus` which adds two values. A reduction over plus then defines the sum operation.

Note here that the reduction may be computed in any order. In particular, this means that the elemental function has to be commutative and associative to ensure deterministic results. The River Trail runtime will not check this property but results might be different between calls even on the same platform.

Scan

The *reduce* methods reduces an array into a single value. For some uses cases, it can be interesting to also store the intermediate results. One commonly used example is the prefix sum operation that, given a vector of numbers, computes another vector of numbers that at each position contains the sum of all elements of the source vector up to that position. To implement this parallel pattern, River Trail's ParallelArray features the *scan* method.

```
var source = new ParallelArray([1,2,3,4,5]);
var psum = source.scan(function plus(a,b) { return a+b; });
```

As we are still computing a sum, the elemental function has not changed. We still use the `plus` operation from the previous reduction example. However, when used with *scan* it now produces the prefix sum of the `source` array.

The same rules of parallel execution that apply to *reduce* also apply to *scan*. Although less obvious, *scan* can be computed in parallel by reordering the reduction steps. Therefore, we only guarantee a deterministic result if the elemental function is commutative and associative.

Scatter

So far we have seen *map* and *combine* that can be used to produce new arrays out of existing arrays. However, both methods are driven by result indices, i.e., they define for each index position in the result how it is to be computed. Sometimes, this mapping is difficult to specify or costly to compute. Instead, it is preferable to specify for a certain source index where it should be stored in the result array. This pattern is supported by the *scatter* method in River Trail. Here is an example:

```
var source = new ParallelArray([1,2,3,4,5]);
var reorder = source.scatter([4,0,3,1,2]);
```

We again first compute our source array `source`. In a second step, we apply the *scatter* method with a single argument: the *scatter vector* `[4,0,3,1,2]`. Thereby, we specify that the first element of `source` is to become the fifth element of the result (indexing starts with 0), the second value in `source` becomes the first in the result and so on. Overall, the above example produces the array `[2, 4, 5, 3, 1]`.

But what happens if we assign two source values to the same position of the result? As *scatter* is potentially computed in parallel, the result would be non-deterministic. Therefore, by default, the River Trail runtime will throw an exception on conflict. However, in practise, conflicts often are meaningful and can be deterministically resolved. For these scenarios, *scatter* accepts an optional second argument: the *conflict function* that, given two conflicting values, produces a single resolution.

On closer inspection, a conflict function is not enough to produce a fully defined result. If the scatter vector contains the same target index more than once, inevitably it will not fill all indices of the target array. To remedy this, we allow the programmer to specify an optional default value that will be used for all index positions that are not defined otherwise.

```
var source = new ParallelArray([1,2,3,4,5]);
var reorder = source.scatter([4,0,3,4,2], 3, function max(a, b) {
    return a>b?a:b; });
```

In the above example, the first and fourth element of the source array are both written to the fifth element of the result and no value is scattered to the second index position. However, we provide the default of `3` as second argument. Lastly, we use a maximum as the conflict resolution function. Thus, the fifth position in the result is the maximum of `1` and `4`, which is `4`. Overall we get `[2, 3, 5, 3, 4]`.

scatter has a final optional argument: the result length. By default, the length of the result will be the same as the source array's. Using a scatter index outside of the result's length will lead to a range error. To spread elements out or reduce the total number of elements, the new length has to be explicitly provided.

Putting it all together, we can write an *histogram* by means of *scatter*. A histogram computes the frequency or number of occurrences of a value with in a sample. Lets assume we have some source data that contains values from zero to five. We also need a second vector that contains weights. We will just use a vector of ones here. Here is the setup:

```
var source = new ParallelArray([1,2,2,4,2,4,5]);
var ones = source.map(function one(v) { return 1; });
```

We can then express histogram as scattering the ones we have just produced to the correct buckets, i.e., we interpret the source data as scatter indices. Conflicts are resolved by adding values, ultimately counting how many times a one has been written to a certain position. The natural default value is `0`. Lastly, we have to provide a new length, as we reduce multiple source values into a single frequency value. Here is what it looks like:

```
var hist = ones.scatter(source, 0, function plus(a,b) { return a+b;}, 6)
```

Filter

With *scatter*, we can now create new ParallelArray objects by reordering the elements of an existing ParallelArray object. However, the result still contains all elements of the source array, modulo conflicts. We are still missing a means to simply drop elements from an array. This is where the *filter* methods comes in. It expects an elemental function as its sole argument. The elemental function has the same signature as in *combine*, i.e., it gets the current index as an argument and the source object is bound to `this`. However, other than in *combine*, the elemental function in *filter* is expected to return a truth value that indicates whether the source array's element at the given index position should be included in the result. Let us look at an example:

```
var source = new ParallelArray([1,2,3,4,5]);
var even = source.filter(function even(iv) {
    return (this.get(iv) % 2) == 0; });
```

As before, we first produce a source array containing the values one to five. Next, we apply the *filter* method using `even` as elemental function, which returns true for all even elements. Thus, we remove all odd elements from the source array, leading to the result `[2, 4]`.

Parallel Video filters with River Trail

In this hands-on tutorial we will use River Trail to parallelize the computationally intensive parts of a HTML5 video application. If you haven't already, read the API description and come back. It is also a good idea to have this API description to refer to while reading through this tutorial.

Setup

Download and Install

If you have not already, download and install River Trail (see [1] and [2]). For this tutorial, you do **not** need to build the extension. You only need the River Trail distribution and the River Trail binary extension for Firefox. To be able to manipulate video from a webcam you will also need to install the Rainbow extension.

Configure

Because of Firefox's security policies, you may have to install Apache (or another webserver) and configure it so that it serves files from the River Trail directory.

Verify

To verify that extension is installed, go to the interactive shell [3]. On the last line, you should see a message saying:

`It appears that you have River Trail installed. Enabling compiled mode...`

If you see this, then the extension has been installed correctly. However if you only see something like:

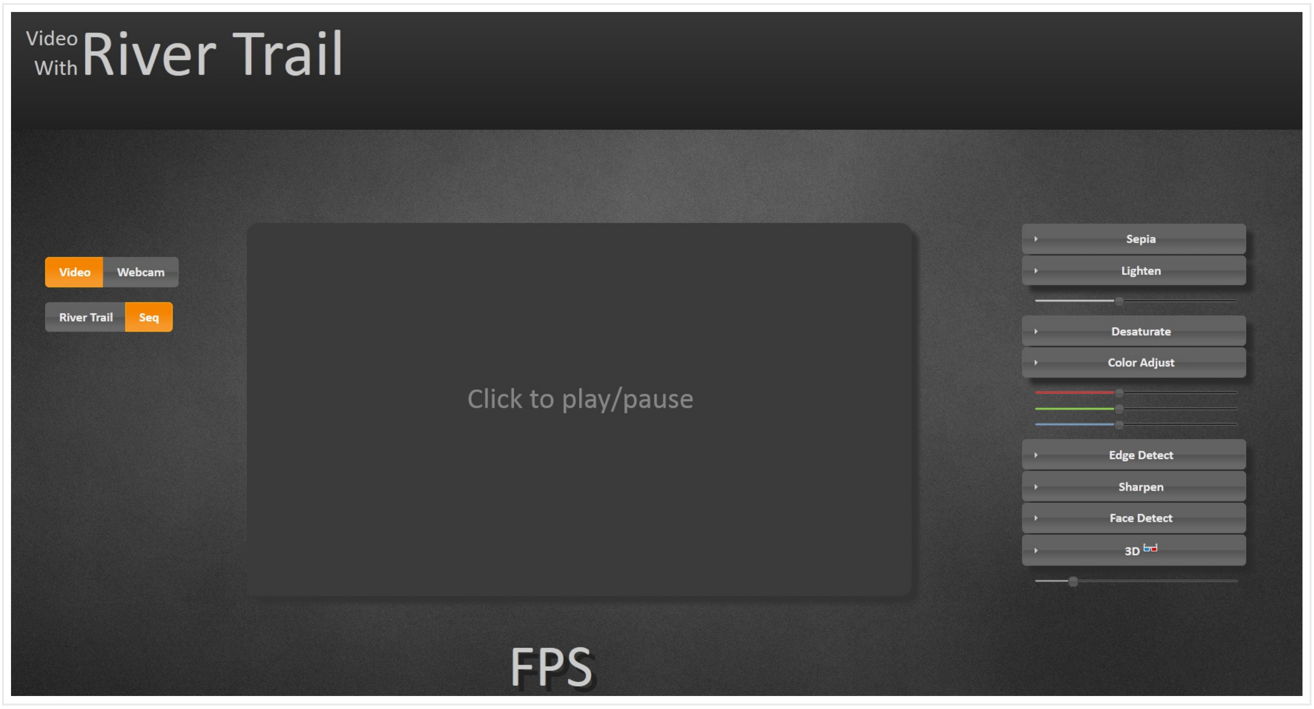
PS: This page uses the sequential library implementation of River Trail. You won't need the extension but there will be no speedup either.

then the extension isn't installed properly. See the River Trail wiki [2] for instructions on installing.

Once the extension is installed correctly, try running one of the included examples in *rivertrail/examples/*. For instance, the n-body simulation example in *idf-demo*.

The Skeleton

The directory *rivertrail/tutorial/src* contains a skeleton for the video application that you can start with. Load up `index.html` in this directory in Firefox and you should see the default screen for the application skeleton:



The large box in the center is a Canvas [6] that is used for rendering the video output. The video input is either a HTML5 video stream embedded in a video tag [7] or live video captured by a webcam. On the right of the screen you will see the various filters that can applied to this input video stream - sepia toning, lightening, desaturation etc. Click on the box in the center screen to start playback and try out these filters. To switch to webcam video, click the "Webcam" toggle in the top-left corner. The sequential JavaScript versions of the filters on the right are already implemented and in this tutorial we will implement the "parallel" versions using River Trail. Before we dive into implementation, lets look at the basics of manipulating video using the Canvas API.

Manipulating pixels on Canvas

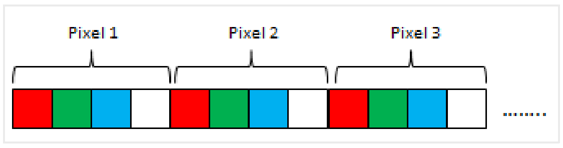
Open up `main.js` in your favorite code editor. This file implements all the functionality in this web application except the filters themselves. When you load the page, the `doLoad()` function is called after the body of the webpage has been loaded. This function sets up the drawing contexts, initializes the list of filters (or kernels) and assigns a click event handler for the output canvas. The `computeFrame()` function is the workhorse that reads an input video frame, applies all the selected filters on it to produce an output frame that is written to the output canvas context. The code below shows how a single frame from a HTML video element is drawn to a 2D context associated with a canvas element.

```
240  // main.js : computeFrame()
241  output_context.drawImage(video, 0, 0, output_canvas.width,
242      output_canvas.height);
```

After this video frame is drawn to canvas, we need to capture the pixels so that we can apply our filters. This is done by calling getImageData() on the context containing the image we want to capture.

```
248  // main.js : computeFrame(), line number 249
249  frame = input_context.getImageData(0, 0, input_canvas.width,
250      input_canvas.height);
251  len = frame.data.length;
252  w = frame.width ; h = frame.height;
```

Now we have an ImageData object [8] called `frame`. The `data` attribute of this object contains the pixel information and the "width"/"height" attributes contain the dimensions of the image we have captured. The data attribute contains RGBA values for each pixel in a row-major format. That is, for a frame with h rows of pixels and w columns, it contains a 1-dimensional array of length w * h * 4 as shown below:



So for example to get the color values of a pixel in the 100th row and 50th column in the image, we would do:

```
var red   = frame.data[100*w*4 + 50*4 + 0];
var green = frame.data[100*w*4 + 50*4 + 1];
var blue  = frame.data[100*w*4 + 50*4 + 2];
var alpha = frame.data[100*w*4 + 50*4 + 3];
```

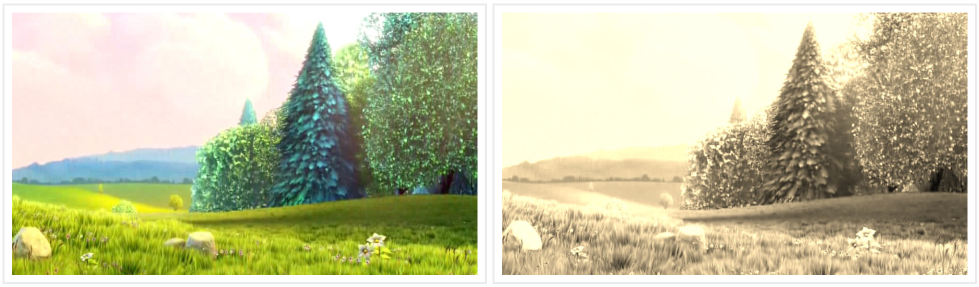
To set, for example the red value of this pixel, simply write the new value at the offset shown above in the `frame.data` buffer.

Sepia Toning

Sepia Toning [9] is a process performed on black-and-white print photographs to give them a warmer color. This filter simulates the sepia toning process on digital photographs or video. Let us first look at the sequential implementation of this filter in the function called `sepia_sequential()` in filters.js.

```
820 function sepia_sequential(frame, len, ctx) {
821     var pix = frame.data;
822     var r = 0, g = 0, b = 0;
823     for(var i = 0 ; i < len; i = i+4) {
824         r = (pix[i] * 0.393 + pix[i+1] * 0.769 + pix[i+2] * 0.189);
825         g = (pix[i] * 0.349 + pix[i+1] * 0.686 + pix[i+2] * 0.168);
826         b = (pix[i] * 0.272 + pix[i+1] * 0.534 + pix[i+2] * 0.131);
827 
828         if(r>255) r = 255;
829         if(g>255) g = 255;
830         if(b>255) b = 255;
831 
832         if(r<0) r = 0;
833         if(g<0) g = 0;
834         if(b<0) b = 0;
835 
836         pix[i] = r;
837         pix[i+1] = g;
838         pix[i+2] = b;
839     }
840     ctx.putImageData(frame, 0, 0);
841 }
```

Remember from the previous snippet that the frame.data buffer contains color values as a linear sequence of **rgba** values. The `for` loop in line 823 iterates over this buffer and for each pixel it reads the red, green and blue values (which are in `pix[i]`, `pix[i+1]` and `pix[i+2]` respectively). It computes a weighted average of these colors to produce the new red, green, blue values for that pixel. It then clamps the new red, green and blue values to [0, 255] and writes them back into the "data" buffer. When the loop is finished, we have replaced the RGB values for all the pixels with their sepia-toned values and we can now write the image back into the output context `ctx` with the `putImageData()` [10] method. The result should look like this (image on the left is the original frame, image on the right is the output):



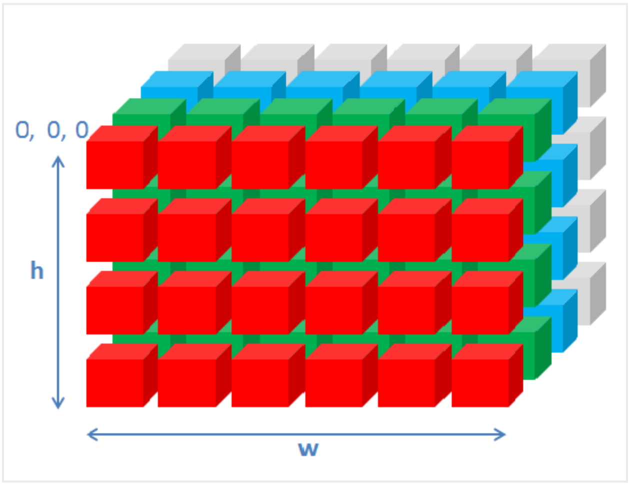
Can we make this parallel?

If you look closely at the `sepia_sequential` function above, you'll notice that each pixel can be processed independently of all other pixels as its new RGB values depend only on its current RGB values. And each iteration of the `for` loop does not produce or consume side-effects. This makes it easy to parallelize this operation with River Trail.

Recall that the *ParallelArray* type has a constructor that takes a canvas object as an argument and returns a freshly minted ParallelArray object containing the pixel data.

```
var pa = new ParallelArray(canvas);
```

This creates a 3-dimensional ParallelArray `pa` with shape [h, w, 4] that looks like the following:



So for pixel on the canvas at coordinates (x, y), pa.get(x, y, 0) will contain the red value, pa.get(x, y, 1) will contain the green value and pa.get(x, y, 2) will contain the blue value. The input ParallelArray that is given to the filter(s) (line 253, main.js):

```
253  else if (execution_mode === "parallel") {
254      stage_output = stage_input = new ParallelArray(input_canvas);
255      w = input_canvas.width; h = input_canvas.height;
256  }
```

`stage_input` and `stage_output` are ParallelArray objects that contain the input and output pixel data for each filtering "stage". Now lets look at the code that causes the filters to be applied (line 271, main.js):

```
271  if(execution_mode === "parallel") {
272    switch(filterName) {
273        case "sepia":
274        case "lighten":
275        case "desaturate":
276        case "color_adjust":
277        case "edge_detect":
278        case "sharpen":
279        case "A3D":
280        case "face_detect":
281            break;
282        default:
283    }
284    // Make this filter's output the input to the next filter.
285    stage_input = stage_output;
286  }
```

You will see that this code block is wrapped in a `for` loop that iterates over all the available filters. To implement a particular filter, we add code to produce a new ParallelArray object containing the transformed pixel data and assign it to `stage_output`. So for example, for the sepia filter, we would write:

```
272  ...
273  switch(filterName) {
274  case "sepia":
275      stage_output = /*new parallel array containing
276      transformed pixel data */;
277      break;
278  ...
```

Now, all we have to do above is produce a new ParallelArray object on the right-hand-side of the statement above. We can produce this new ParallelArray one of two ways - by using the powerful ParallelArray constructor or by using the combine method. Let us look at the constructor approach first.

Recall the the comprehension constructor has the following form:

```
var pa = new ParallelArray(shape_vector, elemental_function,
   arg1, arg2..);
```

where elemental_function is a JavaScript function that produces the value of an element at a particular index in `pa`. Recall that the input to our filter `stage_input` is a [h, w, 4] shaped ParallelArray. You can think of it as a two-dimensional ParallelArray with shape [h, w] in which each element (which corresponds to a single pixel) is itself a ParallelArray of shape [4]. The output ParallelArray we will produce will have this same shape - we will produce a new ParallelArray of shape [h, w] in which each element has a shape of [4], thereby making the ParallelArray have a final shape of [h, w, 4]. Modify line 275 to this:

```
272  ...
273  switch(filterName) {
274      case "sepia":
275          stage_output = new ParallelArray([h, w], kernelName,
276              stage_input);
277          break;
278  ...
```

The first argument [h, w] specifies the shape of the new ParallelArray we want to create. `kernelName` is a Function object pointing to the sepia elemental function (that we will talk about in a moment) and `stage_input` is an argument to this elemental function. This line of code creates a new ParallelArray object of shape [h, w] in which each element is produced by executing the function `kernelName`. This new ParallelArray is then assigned to `stage_output`.

Finally, we have to create the elemental function that produces the color values for each pixel. You can think of it as a function that when supplied indices, produces the ParallelArray elements at those indices. Create a function called `sepia_parallel` in filters.js as follows:

```
// elemental function for sepia
function sepia_parallel (indices, frame) {
}
```

The first argument `indices` is a vector of indices from the iteration space [h, w]. indices[0] is the index along the 1st dimension (from 0 to h-1) and indices[1] is the index along the 2nd dimension (from 0 to w-1). The `frame` parameter is the ParallelArray object that was passed as an argument to the constructor above.

Now let's fill in the body of the elemental function.

```
// elemental function for sepia
function sepia_parallel (indices, frame) {
    var i = indices[0];
    var j = indices[1];

    var old_r = frame[i][j][0];
    var old_g = frame[i][j][1];
    var old_b = frame[i][j][2];
    var a     = frame[i][j][3];

    var r = old_r*0.393 + old_g*0.769 + old_b*0.189;
    var g = old_r*0.349 + old_g*0.686 + old_b*0.168;
    var b = old_r*0.272 + old_g*0.534 + old_b*0.131;

    return [r, g, b, a];
}
```

In lines 3-9, we grab the indices and read the RGBA values from the input ParallelArray `frame` . then, just like in the sequential version we mix these colors in lines 11-13 and return a 4-element array consisting of the new color values for the pixel at position `i, j` . And thats it. Select the "River Trail" toggle on the top-right of the app screen and play the video. You should see the same sepia toning effect you saw with the sequential implementation.

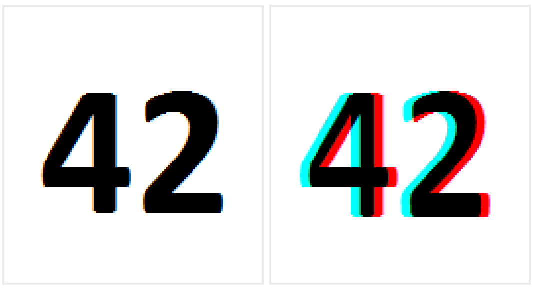
The River Trail compiler takes your elemental function and parallelizes its application over the iteration space. Note that you did not have to create or manage threads, write any non-JavaScript code or deal with race conditions and deadlocks.

Exercise

We could have also implemented the sepia filter by calling the `combine` method on the `stage_input` ParallelArray. Let us try and write that.

Stereoscopic 3D

Let's consider a slightly more complicated filter - one that transforms the input video stream into 3D in real time. Stereoscopic 3D [11] is a method of creating the illusion of depth by simulating the different images that a normal pair of eyes see (see Binocular Disparity [12]). In essence, when looking at a 3-dimensional object our eyes each see a slightly different 2D image due to the distance between them on our head. Our brain uses this difference to extract depth information from these 2D images. To implement stereoscopic 3D, we will use this same methodology - we present two 2D images each one slightly different from the other to the viewer's eyes. The difference between these images - let's call them left-eye and right-eye images; are two-fold. Firstly, the right-eye image is offset slightly to the left in the horizontal direction. Secondly, the red channel is masked off in the right-eye image and the blue and green channels are masked off in the left-eye image. The result looks something like the following (image on the left is the original, image on the right is the 3d version).



Let's look at the sequential implementation first:

```
   function A3D_sequential(frame, len, w, h, dist, ctx) {
    var pix = frame.data;
    var new_pix = new Array(len);
    var r1, g1, b1;
    var r2, g2, b2;
    var rb, gb, bb;
    for(var i = 0 ; i < len; i = i+4) {
        var k = i-(dist*4);
        if(Math.floor(j/(w*4)) !== Math.floor(i/(w*4))) j = i;
            r1 = pix[i]; g1 = pix[i+1]; b1 = pix[i+2];
            r2 = pix[k]; g2 = pix[k+1]; b2 = pix[k+2];
            var left = dubois_blend_left(r1, g1, b1);
            var right = dubois_blend_right(r2, g2, b2);

            rb = left[0] + right[0] + 0.5;
            gb = left[1] + right[1] + 0.5;
            bb = left[2] + right[2] + 0.5;

            new_pix[i] = rb;
            new_pix[i+1] = gb;
            new_pix[i+2] = bb;
            new_pix[i+3] = pix[i+3];
    }
    for(var i = 0 ; i < len; i = i+1) {
        pix[i] = new_pix[i];
    }
    ctx.putImageData(frame, 0, 0);
}
```

Don't worry about the details of the implementation just yet, just note that the structure is somewhat similar to the sepia filter. One important distinction is that while the sepia filter updated the pixel data in-place, we cannot do that here - processing each pixel involves reading a neighboring pixel. If we updated in-place, we may end up reading the updated value for this neighboring pixel. In other words, there is a write-after-read loop carried dependence here. So instead of updating in place, we allocate a new buffer `new_pix` for holding the updated values of the pixels.

Lets start implementing the parallel version. What we want to implement is an operation that reads the pixel data in the input ParallelArray object and produces new pixel data into another. So just like sepia we can use the constructor + elemental function approach.

```
...
switch(filterName) {
    case "A3D":
        stage_output = new ParallelArray([h, w], kernelName,
            stage_input, w, h);
        break;
...
```

Then create the elemental function in `filters.js` as follows:

```
// elemental function for 3D
function A3D_parallel (indices, frame, w, h, dist) {
    var i = indices[0];
    var j = indices[1];
}
```

Each pair `(i, j)` corresponds to a pixel in the output frame. Recall that each pixel such pixel in the output frame is generated by blending two images - the left-eye and right-eye images the latter being a copy of the former except shifted along the negative x-axis (i.e., to the left). Let's call the pixel `frame[i][j]` the left eye pixel. To get the right-eye pixel we will simply read a neighbor of the left eye pixel that is some distance away. This distance is given to us the the argument dist (which is updated everytime the 3D slider on the UI is moved):

```
// elemental function for 3D
function A3D_parallel (indices, frame, w, h, dist) {
    var i = indices[0];
    var j = indices[1];
    var k = j - dist;
    if(k < 0) k = j;
}
```

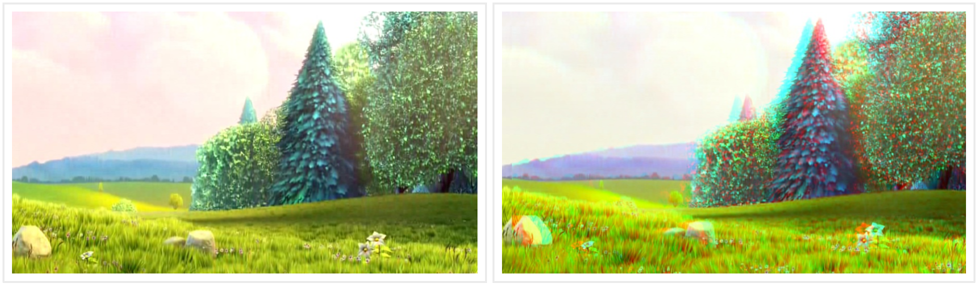
Now `frame[i][k]` is the right eye pixel. We need to guard against the fact that if the distance is large we cannot get the right eye pixel as it would be outside the frame we have. There are several approaches for dealing with this situation - for simplicity we will simply make the right eye pixel the same as the left eye pixel. Now, lets mask off the appropriate colors in each of the left and right eye pixels. We use the `dubois_blend_left/right()` functions for this. You don't have to understand the details of these functions for this tutorial; just that they take in an RGB tuple and produce new RGB tuple that is appropriately masked for the right and left eyes. For details on these functions, read about the Dubois method at [13].

```
// elemental function for 3d
function a3d_parallel (indices, frame, w, h, dist) {
    var i = indices[0];
    var j = indices[1];
    var k = j - dist;
    if(k < 0) k = j;
    var r_l = frame[i][j][0];
    var g_l = frame[i][j][1];
    var b_l = frame[i][j][2];
    var r_r = frame[i][k][0];
    var g_r = frame[i][k][1];
    var b_r = frame[i][k][2];
    var left = dubois_blend_left(r_l, g_l, b_l);
    var right = dubois_blend_right(r_r, g_r, b_r);
}
```

Now we have the separately masked and blended left and right eye pixels. We now blend these two pixels together to produce the final color values.

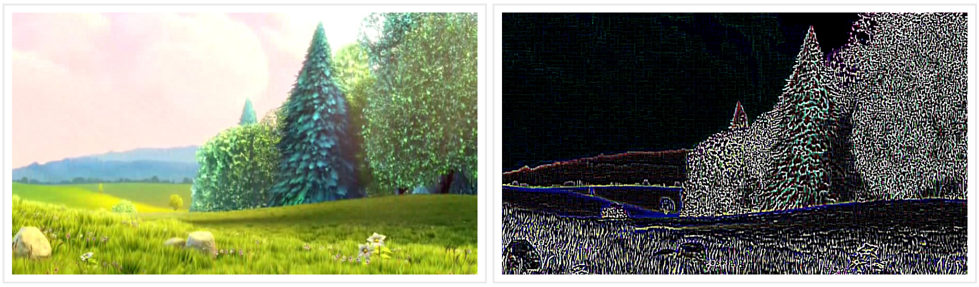
```
function a3d_parallel (indices, frame, w, h, dist) {
    var i = indices[0];
    var j = indices[1];
    var k = j - dist;
    if(k < 0) k = j;
    var r_l = frame[i][j][0];
    var g_l = frame[i][j][1];
    var b_l = frame[i][j][2];
    var r_r = frame[i][k][0];
    var g_r = frame[i][k][1];
    var b_r = frame[i][k][2];
    var left = dubois_blend_left(r_l, g_l, b_l);
    var right = dubois_blend_right(r_r, g_r, b_r);
    var rb = left[0] + right[0] + 0.5;
    var gb = left[1] + right[1] + 0.5;
    var bb = left[2] + right[2] + 0.5;
    return [rb, gb, bb, 255];
}
```

And thats it. Select "RiverTrail" execution, click play and select the 3D filter. Put on Red-Cyan 3D glasses and you should be able to notice the depth effect. Without the glasses this is how it looks (original video frame on the left, with 3D on the right):

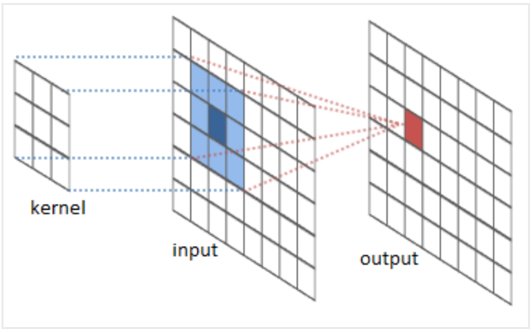


Edge Detection and Sharpening

Let's move on to something a little more complicated - edge detection [14] and sharpening [15]. Edge detection is a common tool used in digital image processing and computer vision that seeks to highlight points in the image where the image brightness changes sharply. Select the edge detection effect and click play to look at the result of the effect.

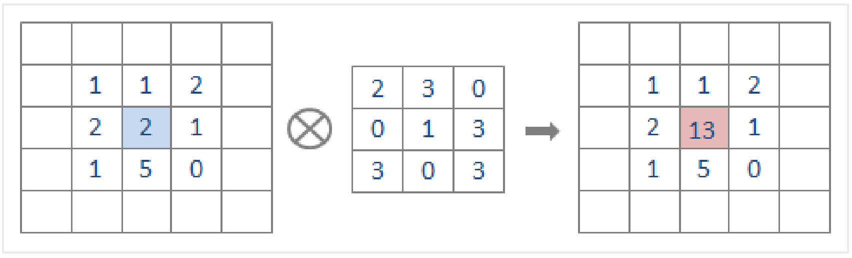


There are many diverse approaches to edge detection but we are interested in the most popular 2D discrete convolution [16] based approach.



At a high-level, discrete convolution on a single pixel in an image involves taking this pixel (shown in dark blue above) and computing the weighted sum of its neighbors that lie within some specific window to produce the output pixel (shown in dark red above). The weights and the window are described by the convolution *kernel*. This process is repeated for all the pixels to produce the final output of the convolution.

Consider a 5x5 matrix convolved with a 3x3 kernel as shown below. For simplicity, we are only interested in the input element highlighted in blue.



The weighted sum for this element is:

(1*2) + (1*3) + (2*0) +

(2*0) + (2*1) + (1*3) +

(1*3) + (5*0) + (0*3) +

= 13. The value of this element in the output matrix is therefore 13.

You can take a look at the sequential implementation of edge detection in `edge_detect_sequential` in filters.js. Don't worry about understanding it in detail yet. Let us try and implement this using RiverTrail. Make a function called `edge_detect_parallel` in filters.js.

```
function edge_detect_parallel(index, frame, w, h) {
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1], [1,2,2,2,1], [1,2,-32,2,1], [1,2,2,2,1],
        [1,1,1,1,1]];
    var kernel_width = (ekernel.length-1)/2; // will be '2' for this kernel
    var neighbor_sum = [0, 0, 0, 255];
}
```

The first two lines of the body are the same as the beginning of the parallel sepia implementation. (m, n) is now the position of a pixel in the input ParallelArray `frame`. The variable `ekernel` is the 5x5 kernel we will be using for convolution (you can copy this over from the sequential version). And we also need a 4 element array `neighbor_sum` to hold the weighted sum.

At this point we have an input frame (`frame`) and a specific pixel `(m, n)` which we will call the "input pixel". Now we need to define a "window" of neighboring pixels such that this window is centered at this input pixel. We can define such a window by using a nested loop as follows:

```
function edge_detect_parallel(index, frame, w, h) {
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1], [1,2,2,2,1], [1,2,-32,2,1], [1,2,2,2,1],
        [1,1,1,1,1]];
    var kernel_width = (ekernel.length-1)/2; // will be '2' for this kernel
    var neighbor_sum = [0, 0, 0, 255];
    for(var i = -1*kernel_width; i <= kernel_width; i++) {
        for(var j = -1*kernel_width; j <= kernel_width; j++) {
            var x = m+i; var y = n+j;
        }
    }
}
```

And there. Now we have an iteration space `(x, y)` that goes from [m-2, n-2] to [m+2, n+2] which is precisely the set of neighboring pixels we want to add up. That is, frame[x][y] is a pixel in within the neighbor window. So lets add them up with the weights from `ekernel` :

```
function edge_detect_parallel(index, frame, w, h) {
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1], [1,2,2,2,1], [1,2,-32,2,1], [1,2,2,2,1],
        [1,1,1,1,1]];
    var kernel_width = (ekernel.length-1)/2; // will be '2' for this kernel
    var neighbor_sum = [0, 0, 0, 255];
    var weight;
    for(var i = -1*kernel_width; i <= kernel_width; i++) {
        for(var j = -1*kernel_width; j <= kernel_width; j++) {
            var x = m+i; var y = n+j;
            weight = ekernel[i+kernel_width][j+kernel_width];
            neighbor_sum[0] += frame[x][y][0] * weight;
            neighbor_sum[1] += frame[x][y][1] * weight;
            neighbor_sum[2] += frame[x][y][2] * weight;
        }
    }
}
```

There is a detail we have ignored so far. What do we do with pixels on the borders of the image for whom the neighbor window goes out of the image ? There are several approaches to handle this situation - we could pad the original ParallelArray on all 4 sides so that the neighbor window is guaranteed to never go out of bounds. Another approach is to wrap around the image. For simplicity, we will simply clamp the neighbor window to the borders of the image.

```
var x = m+i; var y = n+j;
if(x < 0) x = 0; if(x > h-1) x = h-1;
if(y < 0) y = 0; if(y > w-1) y = w-1;
weight = ekernel[i+kernel_width][j+kernel_width];
```

After the loops are done, we have our weighted sum for each color in neighbor_sum which we return.

```
function edge_detect_parallel(index, frame, w, h) {
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1], [1,2,2,2,1], [1,2,-32,2,1], [1,2,2,2,1],
        [1,1,1,1,1]];
    var kernel_width = (ekernel.length-1)/2; // will be '2' for this kernel
    var neighbor_sum = [0, 0, 0, 255];
    var weight;
    for(var i = -1*kernel_width; i <= kernel_width; i++) {
        for(var j = -1*kernel_width; j <= kernel_width; j++) {
            var x = m+i; var y = n+j;
            if(x < 0) x = 0; if(x > h-1) x = h-1;
            if(y < 0) y = 0; if(y > w-1) y = w-1;
            weight = ekernel[i+kernel_width][j+kernel_width];
            neighbor_sum[0] += frame[x][y][0] * weight;
            neighbor_sum[1] += frame[x][y][1] * weight;
            neighbor_sum[2] += frame[x][y][2] * weight;
        }
    }
    return neighbor_sum;
}
```

Summary

The River Trail programming model allows programmers to utilize hardware parallelism on clients at the SIMD unit level as well as the muti-core level. With its high-level API programmers do not have to explicitly manage threads, orchestrate shared data synchronization or scheduling. Moreover, since the API is JavaScript, programmers do not have to learn a new language or semantics to use it.

To learn more about River Trail, visit the project page on Github [1].

References

[1] River Trail on Github *http://github.com/rivertrail/rivertrail*

[2] River Trail wiki *http://github.com/rivertrail/rivertrail/wiki*

[3] River Trail interactive shell *http://rivertrail.github.com/RiverTrail/interactive/*

[4] River Trail API discussion *http://wiki.ecmascript.org/doku.php?id=strawman:data_parallelism*

[5] ES-Discuss mailing list *https://mail.mozilla.org/listinfo/es-discuss*

[6] Mozilla Developer Network: Canvas Tutorial *https://developer.mozilla.org/en-US/docs/Canvas_tutorial*

[7] Mozilla Developer Network: Video Element *https://developer.mozilla.org/en-US/docs/HTML/Element/video*

[8] Mozilla Developer Network : Pixel Manipulation with Canvas *https://developer.mozilla.org/en-US/docs/HTML/Canvas/Pixel_manipulation_with_canvas*

[9] Sepia Toning on Wikipedia *http://en.wikipedia.org/wiki/Photographic_print_toning#Sepia_toning*

[10] The putImageData() method *http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#dom-context-2d-putimagedata*

[11] Stereoscopic 3D on Wikipedia *http://en.wikipedia.org/wiki/Stereoscopy*

[12] Binocular Disparity on Wikipedia *http://en.wikipedia.org/wiki/Binocular_disparity*

[13] The Dubois method *http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=941256&tag=1*

[14] Edge Detection on Wikipedia *http://en.wikipedia.org/wiki/Edge_detection*

[15] Sharpening on Wikipedia *http://en.wikipedia.org/wiki/Edge_enhancement*

[16] Convolution on Wikipedia *http://en.wikipedia.org/wiki/Convolution*