

# Parallelize your program with vfThreaded-x86

## Abstract

Getting an embedded sequential C program implemented at the performance you need on a modern multicore architecture is no easy task. There are lots of ways to break up your code, and each partitioning strategy has a performance and cost impact. Up until now, you've had to do this parallelizing work by hand, which can take weeks or months. It is also risky: if you start down an infeasible path, you're not likely to realize it until you've invested a lot of work that will have to be redone. Parallelizing code is even harder if you are working with open-source or legacy code that you did not write yourself.

vfThreaded-x86 shows you only those partitioning options that are guaranteed to work, without introducing data races, preserving the semantics of your program. vfThreaded-x86 presents x86-specific estimates of the benefits and overhead for each option. Armed with that information, you can define your partitioning strategy without worrying that you might have to come back and rework it.



## 1 What you will learn

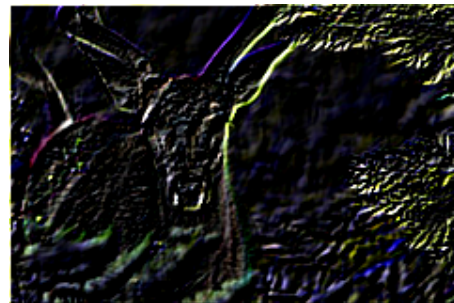
In this tutorial you will learn how to launch vfThreaded-x86, how to analyze your program, and how to explore the parallelization opportunities of your program. You will also learn how vfThreaded-x86 informs you about constructs in your program that inhibit parallelization. Finally you will learn how to parallelize your sequential code using vfThreaded-x86.

You should have received the file `emboss.zip` with this tutorial that contains program code of an application named *emboss*. It also contains the parallelized version of the original application (obtained with vfThreaded-x86) and some test images that you can pass on the command line of *emboss* to check that the program is working correctly. Details on how to build the sources on your platform are available in the included README file.

## 2 Emboss: 2D convolution on images

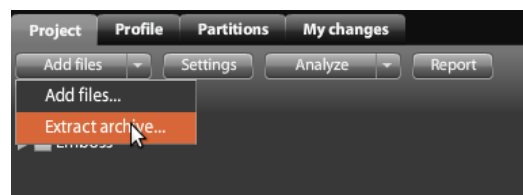
The *emboss* application is representative for a broad range of image processing algorithms. In this tutorial we analyze the parallelization options of *emboss* using vfThreaded-x86. We then obtain a parallel *emboss* implementation by executing vfThreaded-x86 recipes. The source code of the original program and the source code of the resulting parallel program are made available with this tutorial.

If you follow the instructions in the README you will be able to build the *emboss* application and run it on the example images in the TestImages folder. Both input images and output images are in the ppm format which is recognized by most popular image viewers. If you pass `TestImages/Pict_deer.ppm` to *emboss* it produces `Pict_deer_out.ppm`. The pictures should look like this:

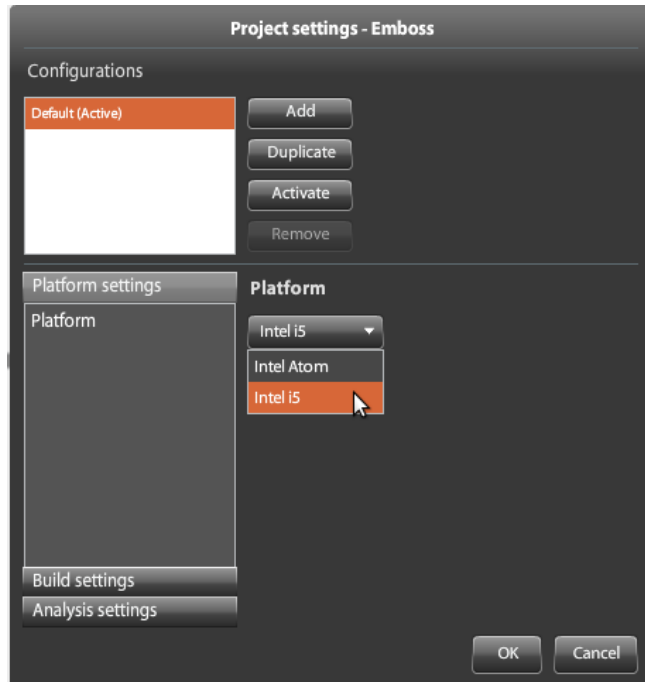


## 3 Program analysis

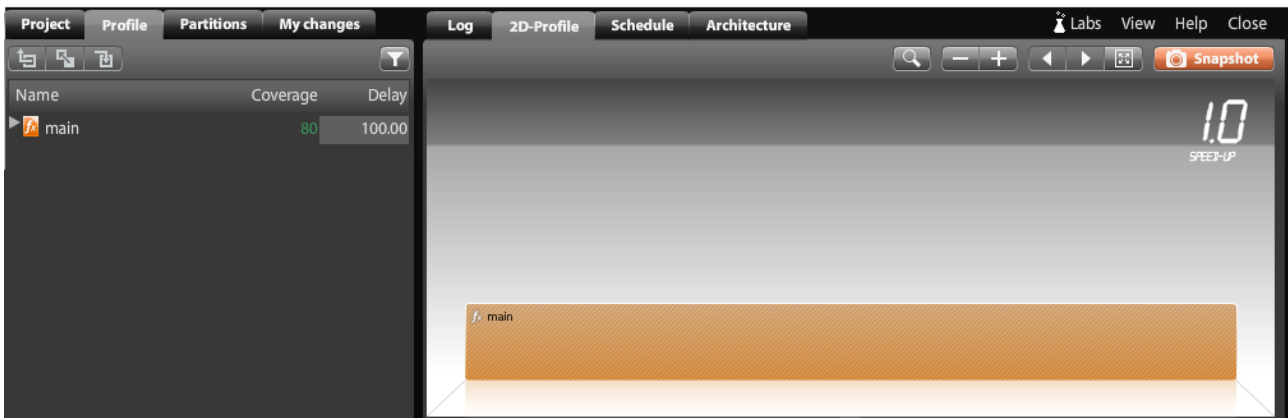
Upload the code by selecting the *Project* tab in the upper-left corner of the screen, then in the 'Add files' drop down menu select 'Extract archive ...', as shown on the right. A file browser appears and you can select the file 'emboss.zip'.



In this tutorial, the Emboss application will be mapped and analyzed on an Intel I5 platform. Since this is not the default platform for vfThreaded-x86, click the 'Settings' button on the *Project* tab and select the 'Intel I5' platform. In order for the application to find the image files navigate to 'Analysis Settings' and select 'TestImages' as its working directory.



Now press the Analyze button in the *Project* tab. This launches vfThreaded-x86's builtin compiler and linker, then executes the resulting program to create a profile. Now we are ready to explore our parallelization options. After analysis has completed, vfThreaded-x86 has switched you into the *Profile* and *2D-Profile* tabs, like this:



In the *Profile* tab to the left we see the program's top-most function invocation called `main`. The two columns next to the function name are 'Coverage' which represents the line coverage of the program's execution<sup>1</sup>, and 'Delay' which is the relative amount of time spend in this function expressed as a percentage of the overall execution time. In the *2D-Profile* at the top-right of the window we see a birds eye view on the same invocation of `main`. It is important to note that the width of the boxes in the *2D-Profile* is always scaled to match the Delay number presented in the *Profile*. Consequently, the most important functions are shown most prominently in the *2D-Profile*.

<sup>1</sup>Line coverage is a metric that shows the percentage of program lines that have been reached at least once while the program was executing

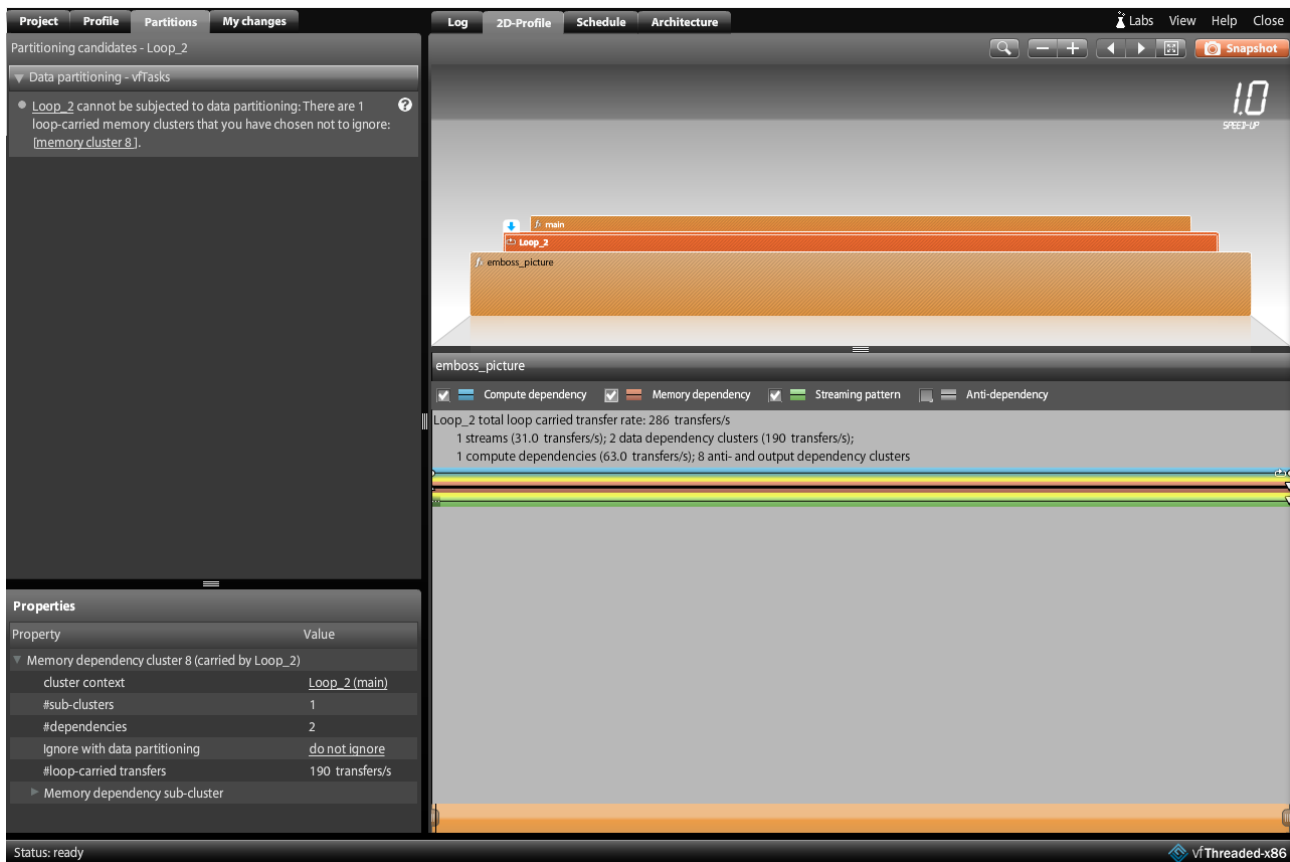
You can now open up the invocation hierarchy by either clicking the little grey triangles in the *Profile*, or by double clicking the boxes in the *2D-Profile*. After a few clicks you are looking at a screen like this:



Note that the invocation hierarchy not only contains functions, but also loops. This is an important point, because loops are at the heart of most of your parallelization opportunities. Loops are where the repetition occurs, and therefore where the concurrency is to be found.

## 4 Can we parallelize this loop?

vfThreaded-x86 can present you with the parallelization opportunities of each loop. To see them, you right-click a loop and select 'Parallelize' from the context menu. Here is what we get if we parallelize the first loop in `main`:



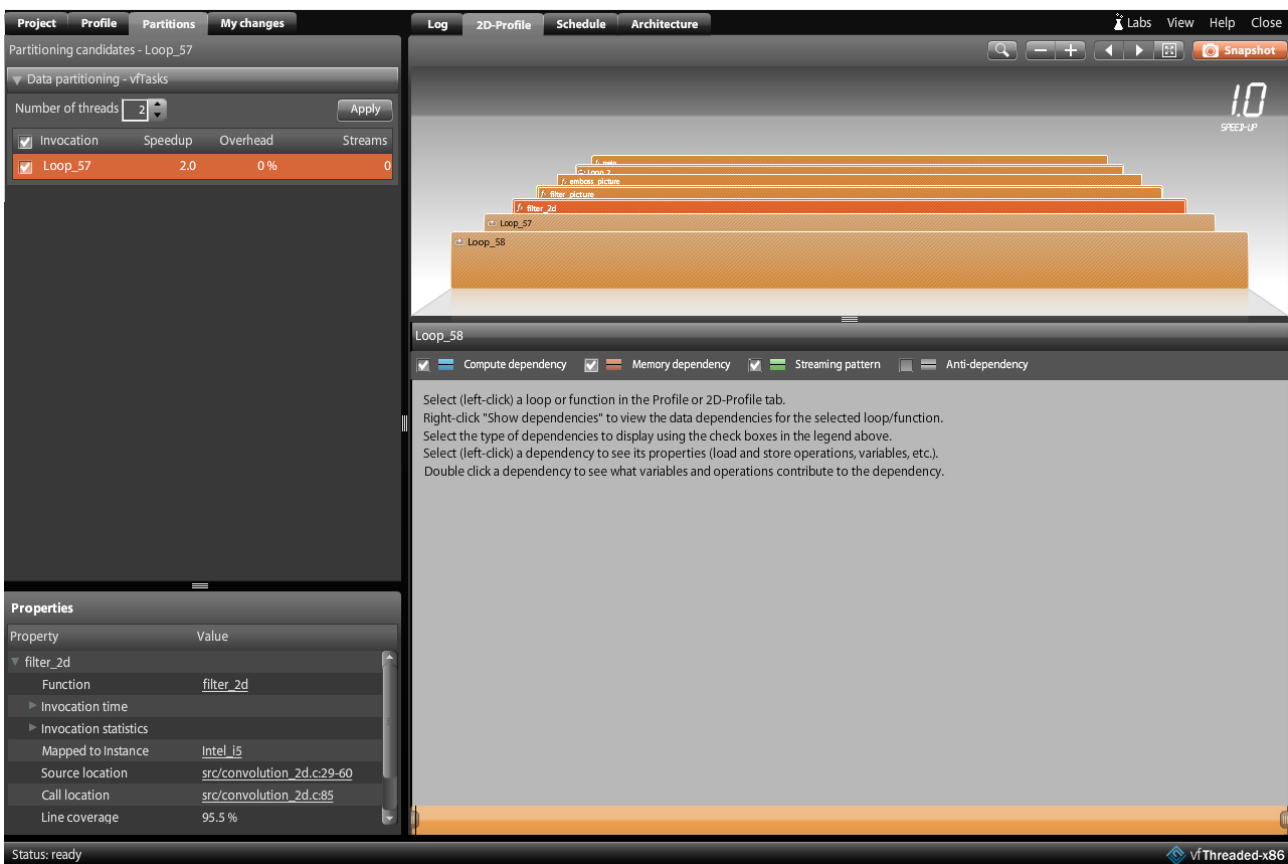
This loop processes one image at a time. You can cross-probe the corresponding source code and get an idea what this loop is about by right-clicking the loop and selecting 'View in source' from the context menu. In the *Partitions* tab vfThreaded-x86 informs us that this loop cannot be parallelized. It also gives as reason that 'Loop\_XXX cannot be subjected to data partitioning: There are 1 loop-carried memory clusters that you have chosen not to ignore: [memory cluster XXX]'. You can actually left-click this partition reject message (it is underlined) and as a result vfThreaded-x86 shows the corresponding dependency highlighted in the dependency pane (bottom-right part of the screen).

Note the Properties window in the lower-left corner describing the details of the dependency. You can click the underlined source locations to jump to the corresponding source code. From that we learn that we are dealing here with two `fprintf()` calls that are sharing the same file descriptor. vfThreaded-x86 is warning us that parallelization of this loop would cause the `fprintf()` statements to possibly execute in the wrong order, resulting in incorrect output. In other words, vfThreaded-x86 refuses to partition this loop because it would destroy the semantics of your program.

This is a very powerful capability: vfThreaded-x86 refuses to change the program semantics, it only proposes parallelizations that preserve the program behavior. All proposed parallelizations are guaranteed to be free of data races. If vfThreaded-x86 refuses to parallelize your loop you can use the detailed reject message from vfThreaded-x86 to change your program such that it can be parallelized. Alternatively you can click 'Ignore with data partitioning' to bypass the dependency, either by marking it as an induction expression or by changing its computation.

## 5 Parallelization by means of data partitioning

At this point we could try to fix the problem with the memory cluster by re-arranging the `fprintf()` statements to make the loop more amenable to parallelization, but in this tutorial we choose not to do so. Instead we drill a little deeper by double-clicking a few more times on the boxes in the *2D-Profile* until we hit the outermost loop in the `filter_2d` function where the real image processing computations are taking place. If you try to parallelize this loop (right-click the loop, select *Parallelize*) this is what you get:



Here we can achieve high speedups with data partitioning.

## 6 The schedule view

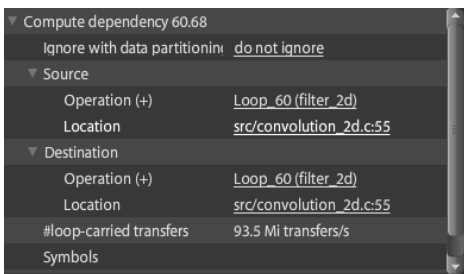
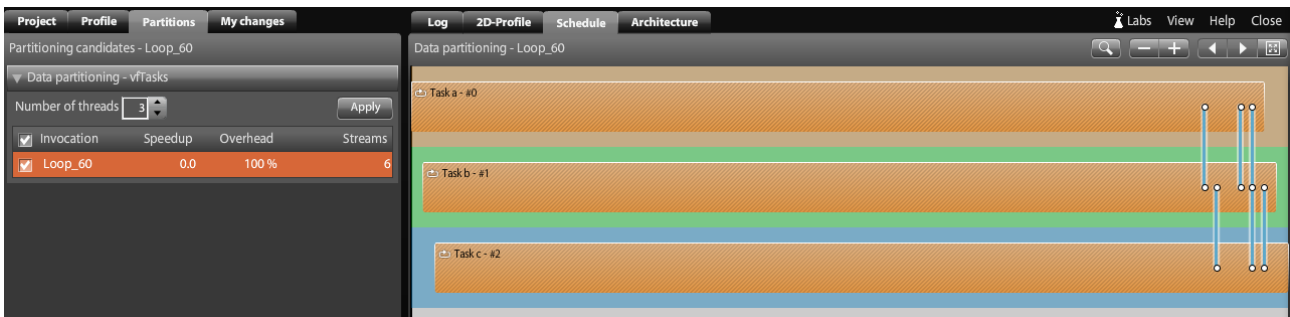
If you click on the proposed partition and then select the *Schedule* tab in the upper-right of the screen, vfThreaded-x86 displays an execution schedule for the parallel tasks that implement the loop. For example, the following picture shows the execution schedule of the outermost loop in the `filter_2d` function on 2 threads (tasks). Time flows from left to right in the *Schedule* view:



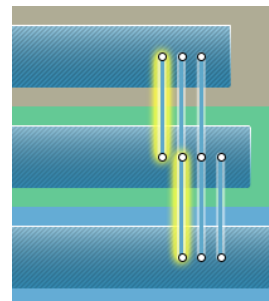
You can increase the number of parallel threads by clicking the **Number of threads**  dialogue. The *Schedule* tab immediately reflects the corresponding schedule.

A loop where all the loop body instances can execute independently of each other is called an embarrassingly parallel loop. In the *Schedule* view this is shown as tasks without any data dependencies running between them. Next we will see an example with data dependencies.

Now double-click the *2D-Profile* boxes until you arrive at the innermost loop in `filter_2d()` with the `fx` induction variable. Right-click this loop, select *Parallelize*, and set the number of threads to 3 (using the small dialogue in the *Partitions* tab). You can now verify that the *Schedule* tab shows the three horizontal terms of the convolution filter running in parallel, with the three pixel colors red, green and blue being accumulated through three compute dependencies. These compute dependencies represent the flow of the computation and they are shown as three vertical blue bars in the schedule view:



You can select a compute dependency in the schedule to see its details in the *Properties* pane (lower-left corner of the screen). In addition, the selected compute dependency is highlighted (yellow glow).



You can now click the Location hyperlinks in the *Properties* pane to see the corresponding source lines:

```

45     for (fx = -(filter->width / 2); fx <= (filter->width / 2); fx += 1)
46     {
47         int px = x + 3*fx;
48         int offlimits = px < 0 || px >= xlimit || py < 0 || py >= ylimit;
49         int pred = offlimits ? 0 : in_row[px + 0];
50         int pgrn = offlimits ? 0 : in_row[px + 1];
51         int pblu = offlimits ? 0 : in_row[px + 2];
52         int coeff = *coeff_ptr++;
53         red += pred * coeff;
54         grn += pgrn * coeff;
55         blu += pblu * coeff;
56     }
57 }
58 out_row[x + 0] = red < 0 ? 0 : red > 255 ? 255 : red * filter->gain;
59 out_row[x + 1] = grn < 0 ? 0 : grn > 255 ? 255 : grn * filter->gain;
60 out_row[x + 2] = blu < 0 ? 0 : blu > 255 ? 255 : blu * filter->gain;
61 }
62 }


```

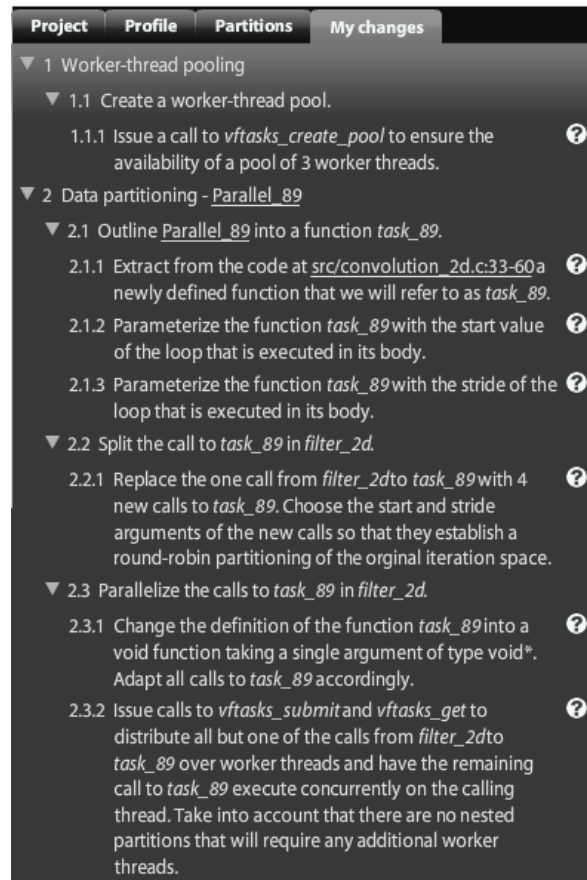
Note that vfThreaded-x86 also provides parallelization overhead for all partition candidates ('Overhead' column in the *Partitions* tab). This way you can quickly determine whether it is worth the effort to implement a certain partition on your platform. In the case of this innermost loop, the overhead is estimated at 100%, which means that no speedup will be attained when parallelizing this loop.

## 7 The parallel program

Once you have explored your parallelization options it is time to implement the parallelization of your choice. In the following example we will assume you have chosen to parallelize the outermost loop in the `filter_2d` function and that you are targeting a quad core CPU.

Switch to the *My Changes* tab. This is where vfThreaded-x86 displays an overview of all the partitioning choices you made for your program. In our case we have applied data partitioning, as shown in the screenshot on the right. The *My Changes* tab shows you a recipe that describes in detail the steps you need to take to implement the desired data partitioning. All you need to do at this point is follow the detailed instructions in the recipe. After every step you can compile your program and verify that it still works.

You can click the  buttons for detailed help with each of the steps. The big advantage of this approach is that you are in full control of the resulting parallel program. Since you are writing the code yourself, you will be able to understand the resulting program and take full ownership of it.



## 8 Summary

This tutorial has shown how to use vfThreaded-x86 to transform a sequential image processing application into an efficient parallel program that runs faster on your PC. In particular vfThreaded-x86 helps you to

1. Analyze your application, find the hot spots and understand the data dependencies;
2. Explore the parallelization opportunities and obtain realistic speedup numbers that include implementation overhead;
3. Get feedback on possible problems in your application that limit the parallelization potential;
4. Create a parallel implementation of your program that you can take full ownership of.

To learn more about vfThreaded-x86, see other examples or sign up for a free trial, visit our website at [www.vectorfabrics.com](http://www.vectorfabrics.com).