Weekly Assignment 3
Parallel Functional Programming

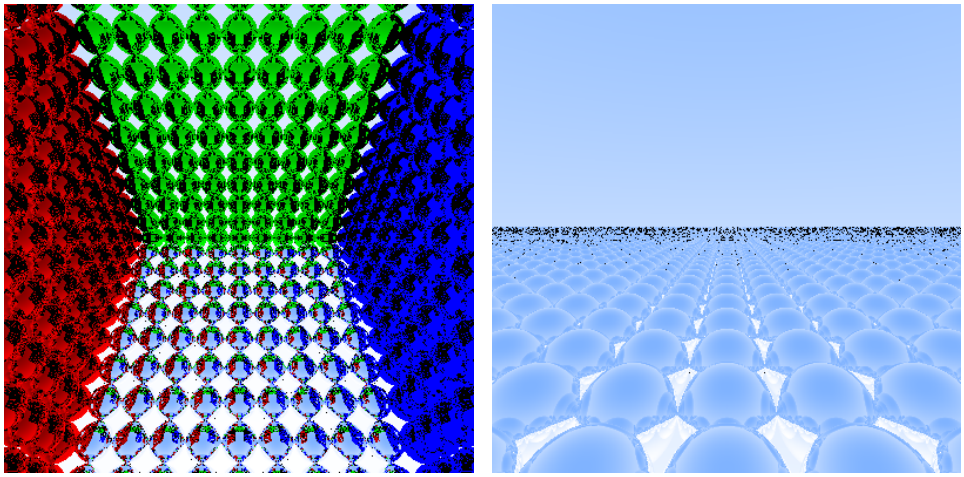
Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

December 2019

Introduction

The handin deadline is the 12th of December.

The handin is expected to consist of a report in either plain text or PDF file (the latter is recommended unless you know how to perform sensible line wrapping) of 4—5 pages, excluding any figures, along with an archive containing your source code. The report should contain instructions on how to run and benchmark your code.



(a) `rgbbox` (b) `irreg`

Figure 1: Renderings of two scenes.

Task 1: Parallelising a ray tracer in Haskell

The nice thing about teaching a university course is that if you find some hobby you enjoy, you can force your students to enjoy it along with you. Recently, I have come to enjoy computer graphics.

This task involves parallelising a provided Haskell implementation of a ray tracer, which given a scene and camera description can render an image in the PPM image format. A ray tracer is a graphics program that creates photorealistic images[1] by simulating rays fired from each pixel, calculating the pixel colour based on which objects are hit in the scene. The Internet contains plentiful material on ray tracing if you need to know more—it's a very common technique.

For our ray tracer, a scene consists of a nonzero number of coloured metallic and reflective balls. You do not need to understand the actual physics or 3D geometry used in the ray tracer.

The ray tracer consists of the following Haskell modules.

`Vec3`: Basic vector math. You do not need to modify this file.

`Scene`: Definitions of sample scenes. You do not need to modify this file, unless you want to add more scenes (the two predefined ones are not very imaginative).

[1] Or rather, better ray tracers create photorealistic images.

`Image`: A data type for images. Most importantly, defines the `mkImage` function, which constructs an image given width, height, and a function for computing the pixel value. This is a candidate for parallelisation.

`Raytracing`: The core ray tracing algorithm. You do not need to understand the vector math, but try to look for potential parallelism here.

`BVH`: A definition of a *bounding volume hierarchy* (more on this below) and axis-aligned bounding boxes (`AABB`). You will need to parallelise the `makeBVH` function.

Finally there are two programs, `pfpray.hs` which renders scenes, and `bench-pfpray.hs`. See `README.md` in the source handout on how to compile and run them.

Bounding volume hierarchies

Quoting from Wikipedia:

> A bounding volume hierarchy is a tree structure on a set of geometric objects. All geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the top of the tree.

A visualisation of the idea is shown on Figure 2. The advantage of using a BVH is that we do not need to check every object in the scene when we need to figure out which object a ray is intersecting. Instead, the BVH allows us to use the bounding box (`AABB`) of the enclosed nodes to quickly rule out entire branches of the tree, without having to look at each individual leaf (see `objsHit` in `Raytracing.hs`).

Your tasks

Ultimately, your task is to make the ray tracer *run fast*, no matter what it takes.

- Parallelise the construction of the BVH. It is a divide-and-conquer algorithm, which the lecture slides contain an example of how to handle.

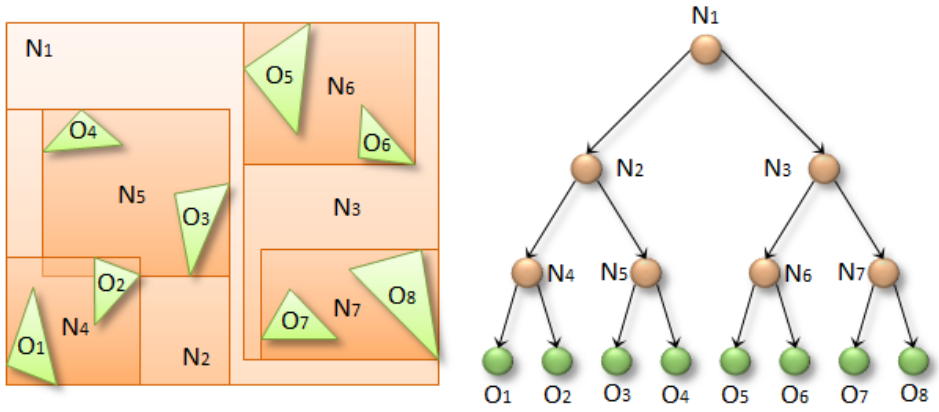


Figure 2: A visualisation of a 2D BVH taken from `https://devblogs.nvidia.com/thinking-parallel-part-ii-tree-traversal-gpu/`

- Each pixel in the image can be computed independently of all others. Parallelise the rendering. Is it better to use static or dynamic scheduling?
- There is at *least* one more source of parallelism in this program. Find one and argue why or why not you think it is worth exploiting in practice. Can you think of a pathological workload (e.g. a one-pixel image) where it might be beneficial to exploit?
- Make sure to use `bench-pfpray.hs` to benchmark your code. Add more benchmark workloads of your own to showcase things you think may be interesting. Read the `criterion` tutorial[2] to understand the output format. Show how your runtime scales as you use more cores (with the `+RTS -Nx` option). Does it scale linearly? Why or why not?

You may use either of the Parallel Haskell frameworks discussed at the lecture: the Par monad or `Control.Parallel.Strategies`.

[2] `http://www.serpentine.com/criterion/tutorial.html`

Task 2: Fusing Stencils: The Blur Example (Halide Lecture)

The file `blur-fusion.cpp` contains several of the implementations discussed in the Halide lecture. More precisely, the breadth-first scheduling, the fully/totally-fused scheduling and the sliding-window scheduling are already implemented there.

Your task is to fill in the missing code corresponding to the implementation of:

a) tiled-fusion scheduling in function `tiledFused`, and

b) sliding-window within tiles scheduling in function `tiledWindow`.

The code structure and the OpenMP parallelization is already provided to you, together with the validation. Your task is to fill in the missing parts of the code (see comments in the code itself).

After your implementation validates for both cases, please run them on (different) multicore hardware that you have access to and report the runtime of each implementation on each such hardware (e.g., which of the five schedules is the fastest, and what speedup is achieved in comparison with the breath-first scheduling).

Please note that `gpu01..4` use the same type of hardware, so report it only once for one of them. Your laptop or home desktop has different hardware characteristics, so report for those as well, etc.

Task 3: Iterative Stencil Fusion

This task refers to the following simple one-dimensional iterative stencil:

```
for(int q=0; q<ITER; q++) {
    forall(int x=0; x<DIM_X; x++) // parallel
        out[x] = ( inp[x-1]+inp[x]+inp[x+1])/3;
    tmp = inp; inp = out; out = tmp; // switch inp with out
}
```

The breadth-first scheduling is already implemented in file `it1d-stencil.cpp`, function `breadthFirst`.

Your task is to fill in the implementation of the function `tiledFused` in the same file `it1d-stencil.cpp`, which has the semantics of:

1 tiling the loop of index `x` with a tile `T`,

2 stripmining the outer loop of count `ITER` by a `FUSEDEG` factor, and

3 'interchanging' the stripmined slice inside the loop of index `x`, while adding enough redundant computation for the ghost zones to respect the original program behavior.

Assuming for simplicity that `FUSEDEG` evenly divides `ITER`, the structure of the resulted code is given below:

```
for(int qq=0; qq<ITER; qq+=FUSEDEG) {

  forall(int tx=0; tx<DIM_X; tx+=T) { // parallel

    float tile[2][-FUSEDEG .. T+FUSEDEG]; // allocated per thread

    // 1. initialize tile[0][-FUSEDEG   : FUSEDEG+T] from array 'inp'

    for(int32_t q0 = 0; q0 < FUSEDEG; q0++) {
      // 2. compute one iteration of the original stencil using
      //    the 'tile' array, which is allocated per thread:
      //    iter q0=0 computes tile[1][-FUSEDEG+1 : FUSEDEG+T-1] and
      //              reads from tile[0][-FUSEDEG   : FUSEDEG+T  ],
      //
      //    iter q0=1 computes tile[0][-FUSEDEG+2 : FUSEDEG+T-2] and
      //              reads from tile[1][-FUSEDEG+1 : FUSEDEG+T-1]
      //    ... and so on ...
    }

    // 3. copy back the result 'tile' to array 'out'
  }
  tmp = inp; inp = out; out = tmp; // double buffering
}
```

Your task is to implement the missing code, such that the program validates, and to test it on as many different multicore hardware as you have access to and report the speedup over the breadth-first scheduling for each of them.