UNIVERSITÀ DEGLI STUDI DI PISA

Department of
Computer Science

Master Degree
in Computer Science

Master Thesis

Parallel stereo matching with FastFlow.

Candidate:

Alessandro Ambrosano

Supervisor:
Prof. Marco Danelutto

Examiner:
Prof. Giuseppe Prencipe

Academic Year 2013/14

Contents

Chapter 1

Introduction

Image registration is a fundamental task in image processing, concerned with the establishment of correspondence between two or more pictures taken, for example, at different times, from different sensors, or from different viewpoints. Many are the systems manipulating images which at a certain point of their computation require the registration of images or some similar operation. Specific examples of such systems are target recognition, where a specifical target is matched with a real-time image of a scene or a set of images, satellite image matching for maps reconstruction or global land monitoring, alignment of medical images such as X-rays or biological images such as neural images for neural reconstruction, or 3D modeling of artworks in cultural heritage using several pictures of the object to model. When referring to the problem to solve in order to perform image registration we talk about the *correspondence* problem.

A notable particular case, which takes the name of *(computer) stereo vision* or *stereo matching*, is the registration of pairs of images used to extract 3D information from a scene. Its analogy with the *stereopsis* in the *Human Visual System* makes this problem of interest for researchers in many fields such as biology and neuroscience, besides of course computer science. This analogy affects the problem in two ways, as on one hand the problem is actually born with the purpose of emulating the Human Visual System, while on the other hand researches in biology and neuroscience of how the Human Visual System actually works helped or guided computer scientists to obtain better and better algorithms during the years.

The objective of this thesis is the study of the implementation of real time stereo matching algorithms, with a particular attention for the accuracy of the results, since usually less accurate algorithms can already achieve real time performances, and often do not show particular features that make their parallel implementation challenging.

We will discuss some stereo matching algorithms, with the purpose of showing in our small way their evolution and refinement in the course of time. Then we choose one to implement it, firstly studying the aspects of a classic sequential implementation and secondly studying its parallelization from both data parallel and stream parallel viewpoints.

In chapter 2 we show some background notions used in this thesis, specifically we will show some computer vision concepts and an overview of the FastFlow framework used during the implementation of our parallel code. A brief and focused set of stereo matching algorithms is showed in chapter 3, explaining briefly the history behind the algorithm we chose to implement. In chapter 4 we show an analysis of the ADCensus algorithm from a parallel point of view. Then we show implementation details both of sequential and parallel implementations of fastADCensus in chapter 5. In chapter 6 we show details about execution times of fastADCensus, both in its sequential and parallel implementations, and perform a comparison with stereo matching algorithms provided by the OpenCV library. Finally in chapter 7 we discuss some ideas for future development.

Chapter 2

Background

2.1 Stereo vision background

2.1.1 Formal definition

Definition 1. *An* image I *is a function*

$$I : \mathbb{N}_w \times \mathbb{N}_h \to \mathbb{N}_i^n$$

where

$$\mathbb{N}_x = \left\{ i \in \mathbb{N}^{\geq 0} \,|\, i < x \right\}$$

w is the image width in pixels, h the image height in pixels, n the number of channels and i the maximum intensity value for each channel.

Definition 2. *Given an image $I : \mathbb{N}_w \times \mathbb{N}_h \to \mathbb{N}_i^n$, a* pixel p *is an element of* $\mathbb{N}_w \times \mathbb{N}_h$.

The use of a functional definition for images comes useful because it makes simpler the explanation of further notions, and for that reason is the usual formuation used in literature. Another viable definition can be given in terms of matrices or tensors and its definition can be easily guessed, and even though it makes mathematically harder, or better, uselessly more sophisticated to explain further concepts, it is the way we will think during the actual implementation. For this reason it is worth specify that while the "access" to intensity values of an image in the functional formulation is done by cartesian coordinates, specifiying firstly the column and then the row, usually mathematical matrices, and even more frequently 2-dimensional arrays are accessed specifying firstly the row and then the column.

Definition 3. *Given two images $I_1 : \mathbb{N}_{w_1} \times \mathbb{N}_{h_1} \to \mathbb{N}_{i_1}^{n1}$ and $I_2 : \mathbb{N}_{w_2} \times \mathbb{N}_{h_2} \to \mathbb{N}_{i_2}^{n2}$, the general problem of stereo matching can be defined as the determination of a function $\Psi : \mathbb{N}_{w_1} \times \mathbb{N}_{h_1} \to \mathbb{N}_{w_2} \times \mathbb{N}_{h_2}$ such that:*

$$I_1(x, y) = I_2(\Psi(x, y)) \tag{2.1}$$

Ψ is often referred as *disparity function* or *disparity map*. Usually some constraints on w_i, h_i, i_i and n_i are given, and in many actual image formats such values are fixed or can not take many values, specifically i is usually 256 and n can be 1 or 3.

In real photo pairs often we could not and very often do not have a correspondence for some pixels, as for example when one of the two cameras can't physically see the point that the other is trying to match (*occlusion*). This formulation of the problem becomes then unsuitable, since actually the function Ψ could not exist. To circumvent this problem, as happens often when a perfect solution can not be obtained, we try to obtain the best possible solution by means of optimization problems. In this particular case, a variational formulation can be given by defining the problem as the search of a solution for a minimization problem with an objective functional E, in formulas

$$\min_f (E(f, I_1, I_2)) \tag{2.2}$$

Usually E is referred as *energy functional*, and split in two components

$$E = E_{data} + E_{smooth}$$

where E_{data} relates intensity values in the two images, and its value decreases with better pixelwise matchings, whereas E_{smooth} is a smoothness term and usually is relative to approximate gradients in the disparity map. An example could be

$$\begin{aligned} E_{data}(f, I_1, I_2) &= \sum_{(x,y)} S\left(I_1(x,y), I_2(f(x,y))\right) \\ E_{smooth}(f, I_1, I_2) &= \sum_{(x,y)} T\left(f(x,y) - f(x+1,y), f(x,y) - f(x,y+1)\right) \end{aligned}$$

where S is a function determining similarity between pixels, and T a function penalizing high gradient values, corresponding to sudden changes in the disparity map.

2.1.2 Pinhole camera model

The pinhole camera model describes the relationship between 3D points, in our case associated with objects in the real world, and the corresponding 2D projection onto the image plane.

Let us consider an "upright" camera, or in mathematical terms a camera whose optical axis[1] is collinear to one axis, let's say z[2].

Our camera consists of a cubic box with a *pinhole* at point O_C, in the middle of a face, the opposite face of the cube is called the *image plane* since there the light is projected resulting a reversed planar representation of the scene. We call the distance between the image plane and the pinhole face *focal length*, and denote it with f. In addition, we define the *virtual image plane* as the plane with distance f from the pinhole face in the opposite direction and the *principal point* as the point on the virtual image plane perpendicular to the optical axis. The projection on the virtual image plane is the same of the image plane turned upside down. By cropping such plane we obtain the actual image from the camera.

[1]The center of an optical system

[2]The choice of collinearity and of the axis is done without loss of generality, every other choice is possible after a rotation

The correspondence between a point (X_C, Y_C, Z_C) in the 3D scene thus correspond to the point

$$u = \frac{X_C f}{Z_C}, v = \frac{Y_C f}{Z_C}$$

on the virtual image plane, where X_C, Y_C and Z_C coordinates are given w.r.t. O_C. An example of a pinhole camera is shown in figure 2.1.

When taking in account multiple cameras, using just the coordinate system of a camera as a reference would result in a difficulty in the mathematical description of a scene w.r.t. the other cameras. To avoid that, we associate some *extrinsic paramethers* to every camera, which is a set of parameters that allow us pass from a camera's coordinate system to the world coordinate system. This, in mathematical terms, can be accomplished by providing a translation vector $\vec{t}$ and a rotation matrix R, so a point in the world coordinate system and a point in the camera coordinate system are related by the following formula:

$$P_W = R(P_C - \vec{t})$$

Some other parameters are *intrinsic* to the camera and often considered, some of them are:

- The focal length f, or in other camera models the parameters needed for the projective transformation.

- The parameters needed to map the image coordinate system to the projective coordinate system of the camera. In this model, given the origin point of the image $o = (o_x, o_y)$ and the width and height of a pixel h_x and h_y, we have the following relations between projective coordinates (x, y) and image coordinates (x_u, y_u):

$$\begin{aligned} x &= (x_u - o_x)h_x \\ y &= (y_u - o_y)h_y \end{aligned}$$

- Geometric distortions due to the physical parameters of the optical elements of the camera. These parameters are absent in the pinhole model but when working with actual cameras they must be taken in account.

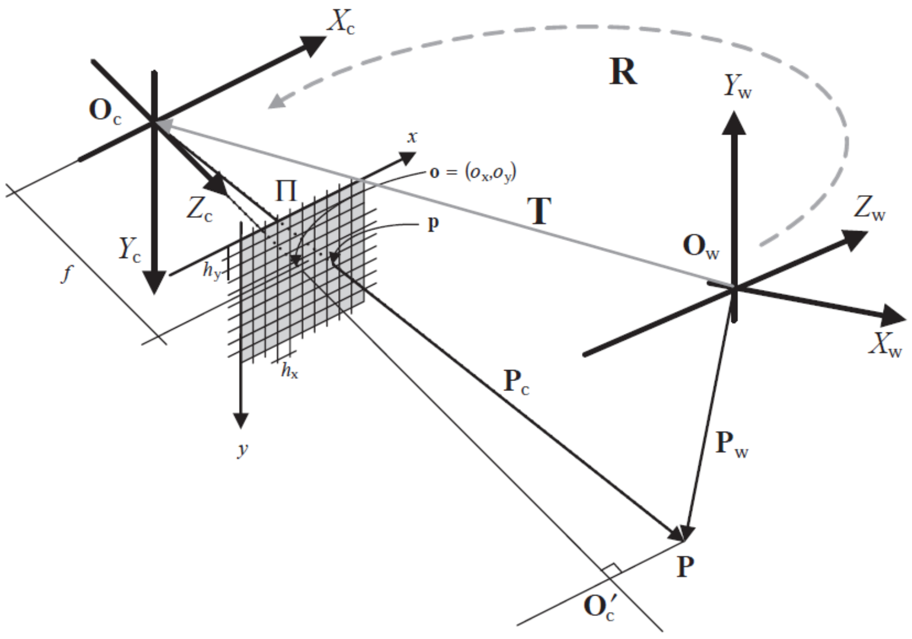


Figure 2.1: Pinhole camera

2.1.3 Epipolar geometry

When observing the same scene with two cameras from two different points of view, epipolar geometry allows us given a point in one image, limit the search of the corresponding point (if any) on the second just to one line instead of the whole image.

Let us consider two pinhole cameras at different positions acquiring the same scene. This configuration is also called *stereoscopic* image acquisition system. The cameras are composed of center points O_1, O_2 and virtual image planes Π_1 and Π_2 respectively. We call the line $\overline{O_1O_2}$ the *baseline* and the intersection of the baseline with Π_1 and Π_2 left and right *epipoles* respectively, and denote them with e_1 and e_2. Given a 3D point P, we call *epipolar plane*

Π_E the plane determined by O_1, O_2 and P. The lines where Π_E intersect with Π_1 and Π_2 are called *epipolar lines.* This example is shown in figure 2.2

The key fact about epipolar lines, which make them so important in stereo vision, is that the projection of a point P on Π_1 and Π_2 can occur only on the epipolar lines of the corresponding epipolar plane. This fact becomes useful when resolving the correspondence problem, since if we take a point on the image and know the displacement of the cameras, it is enough to search on the epipolar line associated to the point on the other image, instead of scanning the whole image. Two matrices can be built, namely the *essential matrix* E and the *fundamental matrix* F that satisfy the following formulas:

$$\begin{aligned} P_1 E P_2 &= 0 \\ p_1 F p_2 &= 0 \end{aligned}$$

for every point P from the two points of view and the corresponding projections on the virtual image planes p_i.

The knowledge of E in case of calibrated cameras, and F otherwise, is necessary for image rectification, since they encode the transformation between image planes, and thus can be used to align epipolar lines in pairs of images.

2.1.4 Algorithm categorization

The stereo matching problem has been approached with many methods which sometimes are deeply different in terms of concepts and implementations. Scharstein and Szeliski [1] give a taxonomy that fits for actual algorithms. They distinguish between *global methods* and *local methods*, and decompose the computation of a disparity map in four possible steps:

- Matching cost computation: for a pair of images I_1, I_2 a cost is assigned to every pair $(I_1(x_1, y_1), I_2(x_2, y_2))$ of pixels, a list of costs is showed later. Algorithms in this step decide how much two pixels are different.

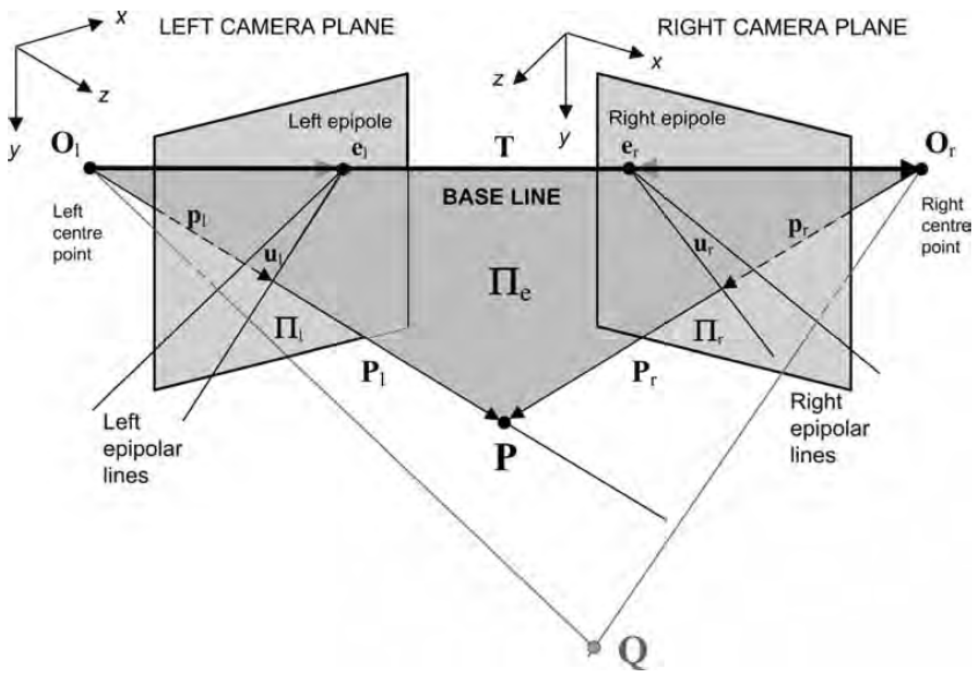


Figure 2.2: Epipolar geometry

- Cost (support) aggregation: costs in a given support region associated to the pixel pair are aggregated. They could for example be summed or correlated. In this step we need to choose a reasonable support region in order to give enough context in the decision of the match. Too small support regions give not enough context to decide the match accurately, too big and/or wrong shaped support instead can compromise the matching for taking in account too much and/or unrelated pixels. Note that exceeding pixels and unrelated pixels are two different categories: for example, when matching a generic scene, if we take a pixel corresponding to one object in one image and try to match it in the other, taking a too big or wrong shaped support region could introduce in the computation pixels belonging to some other object that in the other image is occluded (or viceversa). Pixels of this kind we say are unrelated, while exceeding pixels are pixels that belong to the same object but instead of helping in the matching process can compromise it, as can happen when the object is rotated in a particular way.

- Disparity computation / optimization: in local methods, usually this step is performed computing the cost for pairs $(I_1(\vec{x}), I_2(\vec{x}+\vec{d}))$ in the first two steps described and taking the minimizer d as the disparity value for $I_1(\vec{x})$. Global methods instead concentrate most of their work on this step, usually resolving a variational problem for a given energy functional like the one seen in section 2.1.1.

- Disparity refinement: Usually the disparity maps resulting from the three steps above is discretized, and can contain outliers in occlusion regions and depth discontinuity. If we are interested for example in 3D model reconstruction of a scene, discrete output would result in a scattered model. This could be resolved for exemple by smoothing the disparity map. Disparity outliers instead can be detected and filled with the nearest reliable disparities. This can be done for example using a technique called cross-checking, which compares the left-to-right and right-to left disparity maps to find inconsistencies.

It is worth noting that this "taxonomy" is not suitable for all stereo matching algorithms existing so far. It excludes for example multi-scale methods that use many subsamplings of the original image in coarse to fine approaches to determine disparity values. Most accurate methods so far are global, this is reasonable because global algorithms use "more information" to compute the results. However, global methods are usually time expensive, and so local methods are better suited for low latency / real time implementations. In the years, there is been a fair amount of improvements for local methods, that keep the execution time reasonably low with better accuracy in the results. We will see in chapter 3 how such methods have been improved and then in chapter 4 how to implement the best local method so far, referring to the ranking available on the Middlebury Stereo Evaluation webpage [2].

2.2 FastFlow

FastFlow [3] is a *skeleton* [4] oriented parallel programming framework. It provides efficient implementations of the most used basic configurations for the parallel computation (*parallel patterns*), such as farm, pipeline, divide & conquer, etc., and does not require the users to implement low level parallel code such as thread instantiation and synchronization / communication handling.

Writing a program with FastFlow requires the following steps:

- Creating the computation nodes by instantiating the class `ff_node` and implementing the `svc` method. The code written inside of the `svc` method is totally sequential, and can be easily adapted from possibly existing sequential code.

- Explicitly building the computation network by using a composition of parallel patterns provided, or possibly implementing a custom parallel pattern exploiting the building blocks (queues, nodes, ...).

Parallel patterns in FastFlow are not actually skeletons in a strict sense, but more exactly building blocks / templates for skeleton creation, and are less limiting than actual skeletons because they allow to create ad-hoc structures for computations.

Communication between nodes in FastFlow are implemented in a lockless fashion, by means of SPSC, SPMC and MPSC queues, and many common presets, such as emitter and collector policies in a farm, are provided, but can be customized if necessary.

FastFlow provides its own implementation of a memory allocator and deallocator with C style signatures, with `ff_malloc()`, `ff_realloc()`, `ff_free()`. Its advantages are twofolds, it is faster than the standard C allocator and implemented in a lockless fashion, resulting in better performances during parallel computations.

2.2.1 Examples

We show two simple examples of a pipeline and of a farm implemented with FastFlow. To implement a pipeline computation we just need to embed the code we want to compute in each stage of the pipeline inside a `ff_node` implementing the `svc` method, and then put them together using the `ff_pipeline` pattern. Table 2.1 shows a pipeline for the computation of two subsequent manipulations on an image, a gaussian filter followed by an image scaling.

```
#include <ff/node.hpp>
#include <ff/pipeline.hpp>

using namespace ff;

class GaussFilter : public ff_node {
public:
    void* svc(void* task) {
        // There is no input in FastFlow patterns, so the first stage
        // must take care of the task initialization too.
        image* i = load_image();
        image* i2 = compute_gauss(i);
        ff_send_out(i2);
        return GO_ON; // Keeps the node working waiting for other tasks
    }
}

class ImageScale : public ff_node {
public:
    void* svc(void* task) {
        image* i = (image*)task;
        image_scale(i, 0.5);
        return GO_ON;
    }
}

int main(int argc, char* argv[]) {
    ff_pipe pipe;

    pipe.add_stage(new GaussianFilter);
    pipe.add_stage(new ImageScale);
    pipe.run_and_wait_end();

    return 0;
}
```

Table 2.1: FastFlow pipeline example

For the farm pattern we need to do almost the same, but this time we need to implement emitter, worker and collector nodes. Table 2.2 shows a stream parallel farm for the application of a gaussian filter to a stream of images.

```
#include <ff/node.hpp>
#include <ff/farm.hpp>
#include <vector>

using namespace ff;
using namespace std;

class Emitter : public ff_node {
public:
    void* svc(void* task) {
        image** stream = load_stream();
        for (int i = 0; i < stream_size; ++i) {
            ff_send_out(stream[i]);
        }
        // Ends the execution of the node and forward the end of stream
        // signaling to all subsequent nodes
        return NULL;
    }
}

class Worker : public ff_node {
public:
    void* svc(void* task) {
        image* i = (image*)task;
        gaussian_filter(i);
        return GO_ON;
    }
}

int main(int argc, char* argv[]) {
    ff_farm<> farm;
    vector<ff_node*> workers;

    farm.add_emitter(new Emitter);
    for (int i = 0; i < nworkers; ++i) {
        vector.push_back(new Worker);
    }
    farm.add_workers(workers);
    farm.run_and_wait_end();

    return 0;
}
```

Table 2.2: FastFlow farm example

It is worth noting that the farm parallel pattern provides also scheduling and gathering policies, as for example the ordering policy for gathering, that could be useful for example when processing a stream of ordered tasks that take different times to be computed and such an order is relevant in the output as well.

Other interesting features are the possibility to compose patterns, so for example a worker of a farm could be a pipeline or viceversa, and the feedback channel for farm and pipelines, such that task output by the collector of a farm or the last stage of a pipeline are processed again by the emitter or the first stage of the pipeline respectively.

Chapter 3

Stereo matching algorithms

In this section we present a simple example of a local method with a pseudocode, then we introduce some actual algorithms describing their implementation of the steps described in section 2.1.4.
As we saw in section 2.1.3, given a point P and its projection p on the virtual image plane Π_1 of the first camera, its projection p' on Π_2 must lie on the right epipolar line. Local methods exploit this fact by limiting the pair of pixels for which they compute the cost. In fact, there is no need to compute the cost (p, p') if p' is not on the epipolar line. This useful semplification is not enough though, since epipolar lines does not correspond to horizontal lines, and due to the discrete nature of the images, this results in imprecise traversing of the epipolar line and likely in mismatches due to roundings. For this reason local methods usually assume that pair of images are preprocessed with a *rectification* algorithm, that transforms the image pair in such a way that epipolar lines correspond to image rows. In this section we will keep this assunction.

3.1 Simple algorithm

The example algorithm presented is very simple, and makes some additional assumptions:

- We will refer to I_1 as the left image and to I_2 as the right image, as we assume that the two images are taken from a slight distance one from the other with two horizontal aligned cameras.

- The two images are expressed in grayscale in a single channel.

- The disparity is represented as a scalar non-negative value, this is because the second coordinate of a disparity in a pair opf rectified images is always null. We fix also an upper bound for the disparity value d_{max}.

The four phases are implemented as follows:

- Matching cost computation: we choose to use the *square intensity difference (SD)* cost measure for pairs of pixels. Namely for a pair of pixels $p_1 = I_1(x_1, y_1)$ and $p_2 = I_2(x_2, y_2)$ we have:

$$C(p_1, p_2) = ||p_1 - p_2||^2$$

- Cost aggregation: The support window of a pixel is assumed to be square, the size of the square is a parameter of the algorithm. The aggregation is performed with a sum of the costs in the support window.

- Disparity computation: Reformulating what we said in 2.1.4, this phase is performend taking the minimizer d of the function $C(I_1(x, y), I_2(x - d, y))$.

- Disparity refinement: both cross-checking and smoothing of the disparity maps are performed.

The main algorithm consists in four phases: left-right disparity computation, right-left disparity computation, cross-checking and smoothing. A pseudocode is shown in table 3.3.

```
double compute_sd(scimage_t im1, scimage_t im2, int i1, int j1, int i2, int j2) {
    double result = 0;
```

```c
typedef struct {
    int rows, cols;
    double ** data;
} scimage_t;
```

Table 3.1: Image representation

```c
scimage_t simple_disparity(scimage_t left, scimage_t right, int disp_max) {
    scimage_t lr = disparity(left, right, disp_max, LEFT_RIGHT);
    scimage_t rl = disparity(left, right, disp_max, RIGHT_LEFT);
    cross_check(lr, rl);
    return lr;
}
```

Table 3.2: Main function

```c
    for (int i = -WIN_RADIUS; i <= WIN_RADIUS; ++i) {
        for (int j = -WIN_RADIUS; j < WIN_RADIUS; ++j) {
            result += std::pow(im1.data[i1+i][j1+j] - im2.data[i2+i][j2+j], 2);
        }
    }
    return result;
}

scimage_t disparity(scimage_t left, scimage_t right, int disp_max, int mode) {
    scimage_t result;
    // We assume same size input images
    result.cols = left.cols; result.rows = left.rows;
    result.data = allocate_matrix(left.rows, left.cols);

    for (int i = 0; i < i1.rows; ++i) {
        for (int j = 0; j < i1.cols; ++j) {
            double min, argmin;
            for (int d = 0; d <= disp_max; ++d) {
                double tmp;

                if (mode == LEFT_RIGHT) tmp = compute_sd(left, right, i, j, i-d, j);
                else tmp = compute_sd(left, right, i+d, j, i, j);

                if (tmp < min || d == 0) { min = tmp; argmin = d; }
            }
            result.data[i][j] = d;
        }
    }

    return result;
}

void cross_check(scimage_t& lr, scimage_t rl) {
    for (int i = 0; i < lr.rows; ++i) {
        for (int j = 0; j < lr.cols; ++j) {
            int d = (int)lr.data[i][j];
            if (std::abs(d - rl.data[i-d][j]) > threshold) lr.data[i][j] = -1;
        }
    }
}
```

Table 3.3: Simple local matching algorithm

Notice that in this pseudocode we don't discuss what should happen in the boundaries, and treat boundary pixels like the others. An actual implementation must worry about this and for performance reasons should do it without abuse of conditional instructions, which come natural when doing interval checking for variables.

3.2 Adaptive window with bruteforce approach

Kanade, Okutomi (1991)

The method described by Kanade and Okutomi [5] is one of the first of his kind. They use statistical arguments in order to find a way to estimate the uncertainity of a support window, given an initial disparity estimate, and to compute a disparity increment (w.r.t. the initial disparity estimate) for an arbitrary support window. Such functions are used in an iterative procedure to find pixelwise the best support region, namely the one with the least uncertainty. Since the statistical formulation is extremely generic, theoretically the window computed in the algorithm could be of any shape, but for simplicity reasons the authors use rectangular windows, updated one side at a time.

A pseudocode in C++ is shown in table 3.4.

This method has been cited for historical reason, since the idea of adaptive windows has been reused in many fashions. It is not comparable to modern algorithms in accuracy or efficiency mainly because of the following facts:

- The very first step of this algorithm is to use another algorithm to obtain a disparity estimate. In this way not only we pay the computational cost of such algorithm, but also its accuracy can affect the accuracy of the final result. We will see that this idea will be used again in other algorithms.
- The number of operations to be performed on a single pixel is too big, we need to compute many times the effect of 4 changes in the support window, and those temporary window need to be traversed entirely to obtain the uncertainity estimate. Moreover, there is no particular reason to choose a rectangular window, except the fact that checking all the possible windows would be too difficult. This results inevitably in loss of accuracy.

```
typedef struct r {
    int x, y, w, h;
    struct r (int xx, int yy, int ww, int tt): x = xx, y = yy, w = ww, h = hh {}
} rect_win_t;

scimage_t kanade_okutomi(scimage_t im1, scimage_t im2) {
    // Here we could have used simple_disparity(im1, im2, dmax)
    scimage_t disp_estimate = other_disparity_method(im1, im2, other_params);
    scimage_t result;
    result.cols = im1.cols; result.row = im1.rows;
    result.data = allocate_matrix(im1.rows, im1.cols);

    for (int i = 0; i <= im1.rows; ++i) {
        for (int j = 0; j < im1.cols; ++j) {
             // A 3x3 window centered on the pixel, considered at (0, 0)
            rect_win_t win(-1, -1, 3, 3);
            bool limit_reached = false;
            while (!limit_reached) {
                rect_win_t wxn(win.x-1, win.y, win.w+1, win.h);
                rect_win_t wxp(win.x, win.y, win.w+1, win.h);
                rect_win_t wyn(win.x, win.y-1, win.w, win.h+1);
                rect_win_t wyp(win.x, win.y, win.w, win.h+1)

                // Computes uncertainity in the four windows and returns the best
                rect_win_t best = win_arg_min(im1, wxn, wxp, wyn, wyp);

                // Here we do something to check if the limit is reached.
                // This could be window too big or local minimum reached
                // or an iteration number reached.
                if (best == win) limit_reached = true;
                else win = best;
            }
            result.data[i][j] += delta_disparity(im1, win);
        }
    }
    return result;
}
```

Table 3.4: Kanade Okutomi pseudocode

3.3 Adaptive window through pixel weighting

Yoon, Kweon (2006)

Yoon and Kweon [6] propose another adaptive window approach. Their main argument is that given a pixel p the weight that must be given to a neighboring pixel q in the aggregation phase should be higher the higher is the probability that they share the same disparity value. In formulas:

$$w(p,q) \propto \Pr(d_p = d_q)$$

Instead of relying on a initial disparity estimate, they compute support-weights for a fixed window exploiting gestalt principles. Namely, the weight for a pixel q in the neighborhood of a pixel p should decrease proportionally to both the distance (proximity) and the color "diversity" (similarity) from p.

Calling Δc_{pq} the color difference and Δg_{pq} the spatial distance, they propose to express the weight as

$$w(p,q) = k \cdot f_s(\Delta c_{pq}) \cdot f_p(\Delta g_{pq})$$

where Δg_{pq} is the euclidean distance between pixels and Δc_{pq} the euclidean distance between pixel colors expressed in the CIELab color space, whereas f_s and f_p are defined as:

$$\begin{aligned} f_s(\Delta c_{pq}) &= \exp\left(\tfrac{-\Delta c_{pq}}{\gamma_c}\right) \\ f_p(\Delta g_{pq}) &= \exp\left(\tfrac{-\Delta g_{pq}}{\gamma_p}\right) \end{aligned}$$

where γ_c and γ_p are paramethers. Thus the weight becomes:

$$w(p,q) = k \cdot \exp\left(-\left(\frac{\Delta c_{pq}}{\gamma_c} + \frac{\Delta g_{pq}}{\gamma_p}\right)\right)$$

Referring to the steps in section 2.1.4 they propose the following choices:

- As matching cost between pixels, $e(q, \bar{q}_d) = \sum_{c \in \{r,g,b\}} |I_c(q) - I_c(\bar{q}_d)|$, where $\bar{q}_d$ is the pixel at disparity d from q in the right image.

- As cost aggregation, a normalized weighted sum of matching costs. Since we want pixel comparison to be significant, both weights in the left image and in the right image are taken in account. Specifically, they are multiplied in order to obtain the final weight for a single matching cost. In formulas:

$$E(p, \bar{p}_d) = \frac{\sum_{q \in N_p, \bar{q}_d \in N_{\bar{p}_d}} w(p, q) w(\bar{p}_d, \bar{q}_d) e_0(q, \bar{q}_d)}{\sum_{q \in N_p, \bar{q}_d \in N_{\bar{p}_d}} w(p, q) w(\bar{p}_d, \bar{q}_d)} \tag{3.1}$$

- WTA selection for disparity: $d_p = arg \min_{d \in \{d_{min}, \ldots, d_{max}\}} E(p, \bar{p}_d)$

- No disparity refinement is discussed

From a computational point of view, implementing this algorithm using straightforwardly the formulas would result in some redundant computations. It can be observed immediately that $w(p, q) = w(q, p)$ so we can already say that we do at least the double of operations that we actually need. Moreover the complexity increases with the window size.

The computational problem has been firstly considered by Ju and Kang in 2009 [7], who propose an O(1) aggregation method based on integral histograms.
They simplify the work of Yoon and Kweon dropping the proximity component in weight computation and observing that

$$e(q, \bar{q}_d) \propto \Pr(d_q \neq d_{\bar{q}_d})$$

namely the error component already tells us about the significance of a pixel in the target image, so there is no need to consider the weight in the target image during the computation of $E(p, \bar{p}_d)$.
Then they define an auxiliary structure, the integral histogram of intensity differences, inspired by [8], defined by the function

$$H(x, y, b)_d = \sum_{i=0}^{x} \sum_{j=0}^{y} B\left(\left|I_1(x, y) - I_2(x - d, y)\right|, b\right)$$

where B is defined as follows:

$$B(v,b) = \begin{cases} 1 & \text{if } v \text{ belongs to bin } b \\ 0 & \text{otherwise} \end{cases}$$

and in case of multichannel image an histogram for each channel is computed. Assuming constant domain weights, that is giving the same weight to pixels in the window independently from their distance from the centre, the error formulation would become

$$E(p,\bar{p}_d) = \kappa(p)^{-1} \sum_{q \in W_p, \bar{q}_d \in W_{\bar{p}_d}} w_I(p,q) e(q,\bar{q}_d)$$

where $\kappa(p)^{-1} = \sum_{q \in W_p} w_I(p,q)$ is a normalizing factor corrseponding to the denominator in 3.1.

Using histograms comes useful because if we precompute them for any given window W_p the previous formula can be rewritten as

$$E(p,\bar{p}_d) = \kappa'(p)^{-1} \sum_{b=0}^{N} w'_I(p,b) h(b)_d$$

where N is the number of bins in the histogram h, $\kappa' = \sum_{b=0}^{N} w'_I(p,b)$ and w'_I is a variant of w_I whose second argument is an intensity value instead of a pixel, namely:

$$w'_I(p,b) = |I_1(p) - b|$$

Precomputing the histogram requires a number of operations proportional only to the image dimension and number of channels, since

$$H(x,y,b)_d = H(x-1,y,b)_d + H(x,y-1,b)_d - H(x-1,y-1,b)_d$$

$$+B(|I_1(x,y) - I_2(x,y)|, b)$$

whereas from a pixel's point of view, this operation time complexity is $O(1)$ for every disparity value.

This integral histogram is then exploited by observing that given a neighborhood N_p delimited by points (x^-, y^-) and (x^+, y^+), we have

$$h(b)_d = H(x^+, y^+, b)_d - H(x^+, y^-, b)_d - H(x^-, y^+, b)_d + H(x^-, y^-, b)_d$$

which complexity is $O(1)$ for any chosen rectangular window.

3.4 Adaptive window with geodesic distance

Hosni et al. (2009)

The work of Hosni et al. [9] can be categorized in the "adaptive window" branch, as their main proposal results in a weight of contributes of pixels in a window.

They introduce the concept of *geodesic distance* between pixels in a window, which lies on a color continuity constraint. Geodesic distance does not depend only on the intensity values of the pixel pair taken in account, but instead it searches all possible paths between them looking for the one with less color changes.

If we call c the center of our window, p the pixel considered and $\mathcal{P}_{p,c}$ all paths between c and p, we have that geodesic distance is given by:

$$D(p, c) = \min_{P \in \mathcal{P}_{p,c}} d(P)$$

where $d(P)$ give us the cost of a path. As $1-$paths from a pixel all his 8 neighbours are considered, and paths can be defined as a composition of them. Their definition for costs of such paths is given by

$$d\left(P = (p_1, \ldots, p_n)\right) = \sum_{i=2}^{n} d_C(p_i, p_{i-1})$$

and

$$d_C(p, q) = \sqrt{(p_r - q_r)^2 + (p_g - q_g)^2 + (p_b - q_b)^2}$$

In order to give high support weight to low distance pixels they use a parametric negative exponential

$$w(p, q) = exp\left(-\frac{D(p, q)}{\gamma}\right)$$

and the aggregation is the same as in other weighted methods, namely, for a window centered in c and disparity d:

$$m(c,d) = \sum_{p \in \mathcal{W}_c} w(p,c) \cdot f(p, \bar{p} - d)$$

where f could be d_C but a slightly more complex measure is chosen. The disparity value is then selected with a WTA approach. In order to avoid an expensive bruteforce approach to compute actual geodesic distances, the authors propose a way to approximate their values, using the following steps:

- Cost initialization: initially, every pixel in the support is given a cost, 0 for the center pixel and for all other pixels a large costant value.

- Forward zig-zag update: the support window is traversed from top to bottom, from right to left in a zig-zag fashion, updating the costs according to the formula

  $$C(p) := \min_{q \in K_p} C(q) + d_C(p,q)$$

  where K_p is the kernel composed by p itself and its left, upper-left, up and upper-right neighbours.

- Backward zig-zag update: the same operation is performed traversing the support window from right to left and from bottom to top using a different K'_p composed by p and the immediate neighbours not considered in the previous step.

Forward and backward updates are iterate a number of times to obtain better approximations of the actual geodesic distance, the authors propose to run the update process three times.

3.5 AD-Census

Mei et al. (2011)

Mei et al. [10] propose another local method with their own alternatives for all four steps described in 2.1.4.

Census and Rank transform Census and Rank are two nonparametric cost measures, introduced by Zabih and Woodfill [11] in 1994, lying on the corresponding transforms. They are said nonparametric because they depend only on the mutual relations between pixels in a neighbourhood.
Given a pixel p and a neighbourhood $\mathcal{N}_p$ the rank transform can be defined as

$$rank(p) = \# \left\{ q | q \in \mathcal{N}_p \wedge I(q) \geq I(p) \right\}$$

The census transform instead consists in considering a matrix M with the same dimensions of the neighbourhood and filling with the following rule:

$$M(i,j) = \begin{cases} 0 & \text{if } \mathcal{N}_p(i,j) \leq I(p) \\ 1 & \text{otherwise} \end{cases}$$

then building a bit string reading from left to right, from top to bottom the values in M, except for the position corresponding to p.
Actual matching costs are obtained by taking the difference of two rank values for the rank cost. On the other hand, to compute the census cost we need a string distance between strings of the same length, as for example the Hamming distance.
Note that both for rank and for census we need the size of all neighbourhoods to be the same in order to compute the cost consistently.

The matching cost proposed is a combination between absolute difference (AD) and census costs. They propose the following variant of AD

$$C_{AD}(p,d) = \frac{1}{3} \sum_{c \in R,G,B} |I_1(p)_c - I_2(p - (d,0))_c|$$

and the overall cost is given by

$$C(p,d) = \rho \left(C_{census}(p,d), \lambda_{census} \right) + \rho \left(C_{AD}(p,d), \lambda_{AD} \right)$$

where ρ is a robust function in c:

$$\rho(c, \lambda) = 1 - exp \left(1 - \frac{c}{\lambda} \right)$$

The aggregation phase is based on the one in [12] and consists in two steps. In the first step a "cross" is constructed for every pixel $p = (p_x, p_y)$ consisting in four pixels p^{x+}, p^{x-}, p^{y+}, p^{y-}. We explain the procedure for p^{x-} as for the others is the same. We obtain p^{x-} by moving left from p until one of the following conditions is violated:

- $D_c(p^{x-}, p) < \tau_1$ and $D_c(p^{x-}, p^{x-} + (1, 0)) < \tau_1$
- $D_s(p^{x-}, p) < L_1$
- $D_c(p^{x-}, p) < \tau_2$, if $L_2 < D_s(p^{x-}, p) < L_1$

where

$$\begin{aligned} D_s(p, q) &= |p - q| \\ D_c(p, q) &= \max_{i=R,G,B} |I(p)_i - I(q)_i| \end{aligned}$$

are the spatial and color distance respectively and L_1, L_2, τ_1, τ_2 are given constants. The support region for p is then obtained by merging all horizontal arms of the pixels on the vertical arm of p (p included). In the second step we compute the actual aggregation. The costs in a support area are accumulated for horizontal stripes, and then the intermediate results are summed. This particular choice is important because a simple implementation with integral images both avoids redundant computations and reduces the per-pixel operations to a constant amount.

For what concerns disparity selection, instead of the classical WTA approach, they suggest a scanline optimization inspired by [13]. The scanline optimization technique consists in the creation of a new cost function C_r, where r is a scanline direction. $C_r(p, d)$ is updated taking in account the aggregation cost $C_1(p, d)$ and the costs alongside r $C_r(p - r, \cdot)$ in this way:

$$\begin{aligned} C_r(p, d) &= C_1(p, d) \\ &+ \min\left(C_r(p - r, d), C_r(p - r, d \pm 1) + P_1, \min_k C_r(p - r, k) + P_2\right) \\ &- \min_k C_r(p - r, k) \end{aligned} \tag{3.2}$$

where P_1 and P_2 ($P_1 \leq P_2$) are two parameters penalizing disparity changes between neighbouring pixels. They are set according to color changes $D_1 = D_c(p, p-r)$ in the left image and $D_2 = D_c(p-d, p-d-r)$ in the right image with the following rule:

- $P_1 = \Pi_1$, $P_2 = \Pi_2$, if $D_1 < \tau_{SO}$, $D_2 < \tau_{SO}$.
- $P_1 = \Pi_1/4$, $P_2 = \Pi_2/4$, if $D_1 < \tau_{SO}$, $D_2 \geq \tau_{SO}$.
- $P_1 = \Pi_1/4$, $P_2 = \Pi_2/4$, if $D_1 \geq \tau_{SO}$, $D_2 < \tau_{SO}$.
- $P_1 = \Pi_1/10$, $P_2 = \Pi_2/10$, if $D_1 \geq \tau_{SO}$, $D_2 \geq \tau_{SO}$.

with Π_1, Π_2 constants and τ_{SO} a threshole for color difference. After performing scanline optimization in four directions, two horizontal and two vertical, the disparity value chosen is the one minimizing

$$C_2(p,d) = \frac{1}{4}\sum_r C_r(p,d)$$

Disparity refinement in this algorithm consists in a multi-step process:

- Outlier detection: outliers are first detected in the left-right disparity map D_L with consistency check. A pixel p is said outlier if $D_L(p) = D_R(p - (D_L(p), 0))$ does not hold. Outliers are then classified in occlusion and mismatches using a method proposed by Hirschmüller: for outlier p at disparity $D_L(p)$ we check the intersection of the epipolar line with D_R, if such intersection does not exist p is labelled as "occlusion" otherwise as "mismatch".
- Iterative Region Voting: for every outlier, all disparity values of reliable pixels in the support are are collected in a histogram H_p with $d_{max} + 1$ bins. Calling d_p^* the index of the bin with the higher value and $S_p =$

$\sum_{i=0}^{d_{max}} H_p(i)$ the number of reliable pixels in the support region, we choose d_p^* as disparity for p if:

$$S_p > \tau_S \wedge \frac{H_p(d_p^*)}{S_p} > \tau_H$$

with τ_S and τ_H threshold values. In order to consent the propagation of new disparity values the author suggest to iterate this process 5 times.

- Proper Interpolation: outliers that could not be corrected by the previous phase are interpolated. For every outlier, the nearest reliable pixels in 16 directions are considered. If the outlier is an occlusion, we take the minimum disparity between disparities of reliable pixels, otherwise we take the disparity of the most color-similar.

- Depth discontinuity adjustment: for every pixel on a disparity edge p two pixels p_1 and p_2 are collected from both sides of the edge. $D_L(p)$ is replaced by $D_L(p_1)$ or $D_L(p_2)$ if one of the two has smaller matching cost than $C_2(p, D_L(p))$.

- Sub-pixel enhancement: disparity values are interpolated pixelwise. Intepolation is computed as follows:

$$d^* = d - \frac{C_2(p, d_+) - C_2(p, d_-)}{2\left(C_2(p, d_+) + C_2(p, d_-) - 2C_2(p, d)\right)}$$

where $d = D_L(p)$, $d^+ = d + 1$, $d^- = d - 1$. In mathematical terms, this interpolation corresponds to the search of the minimum point of a quadratic polynomial. Namely, for every pixel p at estimated disparity d we fit C_2 with a parabola interpolating the points

$$\left(-1, C_2(p, d_-)\right), \left(0, C_2(p, d)\right), \left(1, C_2(p, d_+)\right)$$

The minimum at this point must be between -1 and 1 excluded as we already know that $C_2(p, d_-) > C_2(p, d)$ and $C_2(p, d_+) > C_2(p, d)$ by construction, but could not be exactly at 0, which would correspond to use d as disparity estimation. Since values of d correspond to pixels, this is why this is called sub pixel enhancement.

- Median filtering: the final disparity results are obtained by smoothing the interpolated disparities with a 3×3 median filter.

Chapter 4

Parallelism

In this chapter we will see how the ADCensus algorithm described in Chapter 3 can benefit of parallelism techniques. ADcensus, as all the other methods described and many other existing local algorithms, at a certain point the computation perform the same operations on every pixel (SIMD), this makes it well suitable for data parallelism techniques. However, both in data and stream parallelism we need to treat properly the Cross building stage as only source of possible load unbalancing.

4.1 Activity Graph

Considering the algorithm description in section 3.5 we can immediately split the computation in five subsequent phases: *cost initialization*, *cross building*, *cost aggregation*, *scanline optimization* and *disparity refinement*, as shown in figure 4.1

The cost initialization phase can be computed without data dependencies between pixels, so we have $w \times h$ independent activities, where w and h are the image width and the image height respectively. The same holds for the cross building phase. When computing the census cost between pairs of pixels p and $p - d$ as described in 3.5, we need previously to compute the census transform of such pixels. In order to avoid redundant computations,

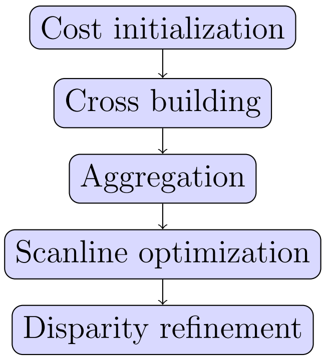


Figure 4.1: AD-Census activity graph overview.

we add an additional stage for census trasforming both images.
The aggregation step in the original algorithm is optimized using four intermediate histogram images, so this single phase also can be split as is shown in 4.2. The horizontal-vertical and vertical-horizontal normalization steps require the size in pixels of the support region of every pixel. Support regions computed merging vertically horizontal arms and viceversa have different sizes, so we need the value of one or the other depending on the current iteration. The computation can be performed with integral images similarly to the proper aggregation, for instance in HV size computation we first compute vertical integral of left-right arm sizes, and then take for every pixel its up-down interval with just a single difference. To compute VH support sizes we switch the directions.
Cost updating is iterated four times, alternating the directions of the integral images, and each iteration uses the results of the previous one, so the overall stage consists of 22 sequential phases, 2 for HV and VH support size computation and 5 for each iteration.
In the scanline optimization we need to fill $4(d_{max} + 1)$ matrices with the values of $C_2(\cdot, \cdot)$, and as we can see in equation 3.2 they are not independent. Focusing for example on the left-to-right scanline optimization, to compute C_2 for a pixel p at any disparity d we need all $d_{max} + 1$ costs for the pixel at the left of p. The same holds for the other three directions. This results in

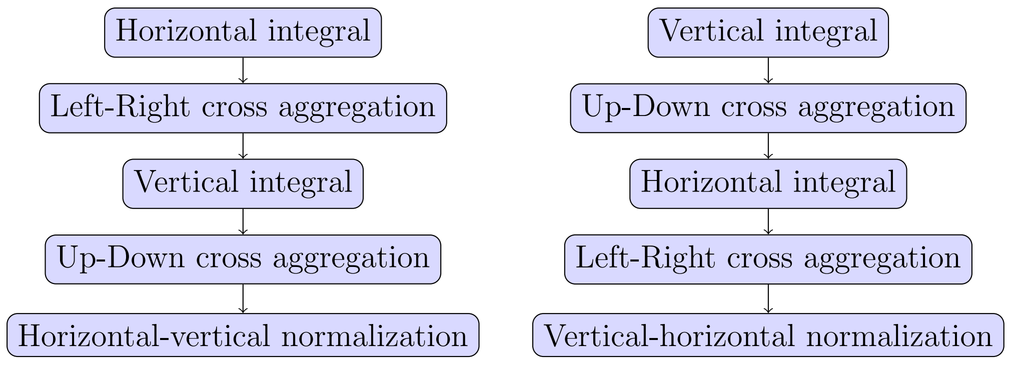

Horizontal integral
Left-Right cross aggregation
Vertical integral
Up-Down cross aggregation
Horizontal-vertical normalization
Vertical integral
Up-Down cross aggregation
Horizontal integral
Left-Right cross aggregation
Vertical-horizontal normalization

Figure 4.2: AD-Census aggregation phases: the aggregation phase is iterated four times, iterations 1 and 3 follow the chart on the left, iterations 2 and 4 the one on the right.

an activity graph locally organized as in figure 4.3.

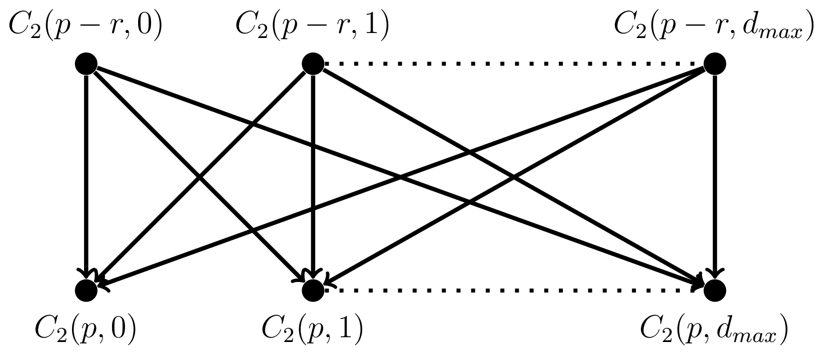


Figure 4.3: Local data dependencies in disparity computation.

An additional disparity estimation stage, can be identified, whose task is to find the minimizer of C_2 described in eq.3.5. This stage has no data dependencies between pixels.

The disparity refinement multi-step phase suggests control dependencies between the steps, as every step after outlier detection tries to correct what could not be corrected in the previous one.

All five steps can be performed pixelwise without data dependencies between pixels. For iterative region voting, in particular, we can use integral images to compute H_p and S_p.

The Depth discontinuity adjustment step requires the computation of edges in the disparity image. This can be obtained by a Sobel filter applied to the disparity image, followed by a threshold filter. Filtering phases identify two additional stages, and can be computed pixelwise without dependencies.

Balancing considerations Even though many of the phases described could be implemented in an embarassingly parallel fashion, some of them hide load balancing criticities.

- In the cross building phase, every pixel in principle could require a different amount of iterations, since we don't know when (neither if) we'll stop for a change in color intensity.
- Scanline optimization is balanced only because we choose left-right and up-down directions. A different choice would have resulted in an unbalancing in the beginning and the end of the C_2 computation.

The overall logical stages are then

- Census transform
- Cost initialization
- Cross building
- HV and VH support size computation
- Aggregation: logically we can choose to split this stage in the 4 iterations or in the 20 single steps as shown in figure 4.2.
- Scanline optimization: this stage too can be splitted in the 4 substages computed for different directions. If necessary, the 4 substages could be completed concurrently since they are independent.
- Disparity estimation
- Outlier detection

- Iterative region voting: logically divisible in its 5 iterations
- Proper interpolation
- Edge detection
- Depth discontinuity adjustment
- Sub-pixel enhancement

4.2 Data parallelism

The parallelism exploitation in the disparity computation of a single pair of images does not leave many alternatives. Since every single stage in the list is basically a loop which iterate through one or more support structures, we can then implement every stage parallelizing properly such loops.

Unbalancing in the Cross building phase can be resolved by observing that the average cost per pixel is given by

$$\mathcal{C}(\text{Cross building}) \sim 8\,\mathcal{C}(D_c)\,\mu_{arm\_length}$$

where μ_{arm_length} is the average length of a single arm of the crosses in the areas of the images over we are computing. Splitting the image in n equal parts using column ranges or row ranges would result in a significant change of the value of μ_{arm_length} in the loop portions since there are image areas, as for instance uniform backgrounds, who generate larger crosses. This can be solved using row or column interleaving, corresponding actually to vertical and horizontal subsampling. As a result the parallel portions of the loop correspond to loops in subsampled (and thus similar enough) versions of the original images, and the difference in μ_{arm_length} does not vary too much.

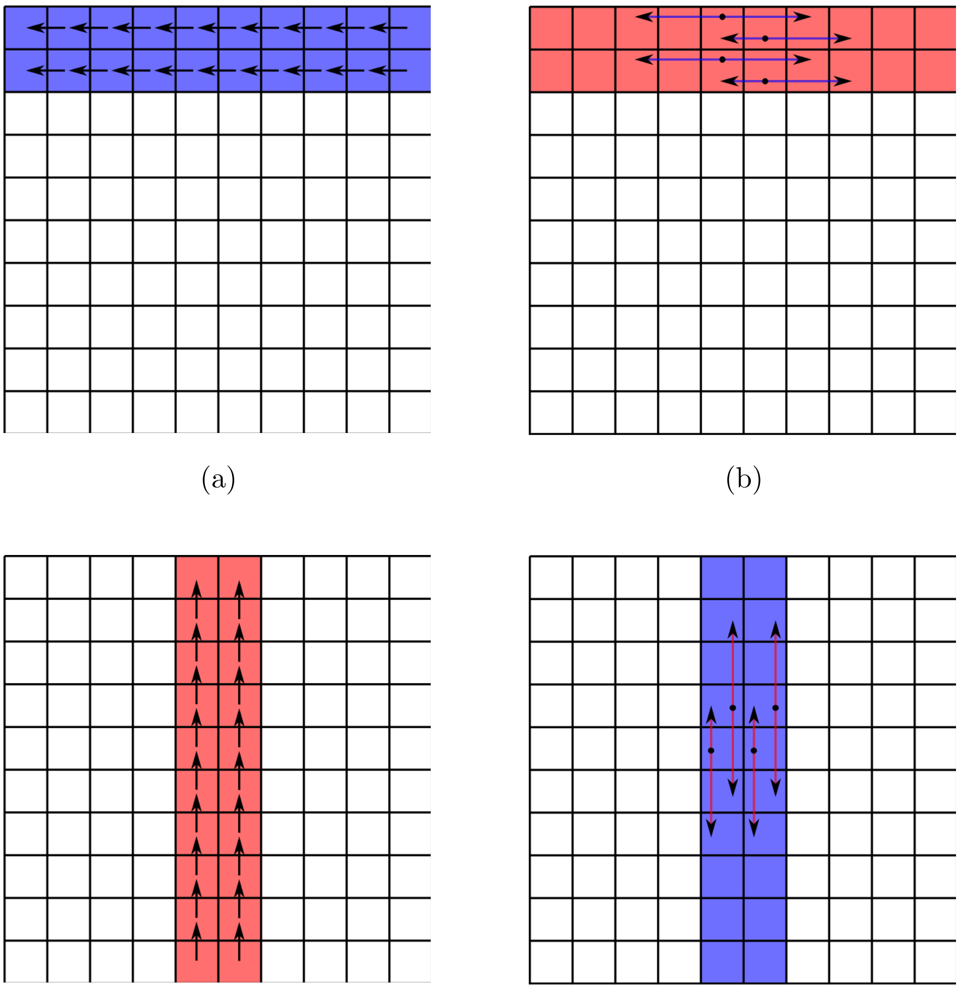
The only unbalanced stage is resolved and theoretically the computing load is actually split between the execution units, but the algorithm has

an intrinsic parallelization problem, as orthogonal integral images used in the $O(1)$ implementation of the aggregation step and orthogonal histograms used similarly in the iterative region voting turn out to be a double-edged sword. When computing integral images we use two support matrices S_1, S_2, and during every horizontal or vertical aggregation we accumulate values in one and write actual aggregations in the other. Taking as an example a HV parallel aggregation phase, accumulation in the horizontal phase is done left to right, and then the computation can be split in horizontal stripes of the support matrix S_1 (Fig. 4.4a). The left-right aggregation then can exploit locality by writing on S_2 in the same horizontal stripe (Fig. 4.4b). Specifically, for each pixel (x, y) in the stripe, during aggregation we read values in the row y of the stripe, which we have processed in the accumulation phase. At this point we start computing the vertical accumulation that this time must be done top to bottom, and we need to split S_2 in vertical stripes (Fig. 4.4d), so we need to discard most of the part of S_2 processed during horizontal aggregation ($\sim \frac{nw-1}{nw}$) and then proceed similarly to the horizontal stage.
Continuous memory discarding per se would not be a big concern, but adding the fact that we perform 1-2 operations on each element makes us pay mainly the memory manipulation. Low operation count is again an intrinsic feature of the orthogonal integral images approach, so suboptimal results can be attributed to the use of such approach.

4.3 Stream parallelism

In order to exploit parallelism in a stream of pair of images, as could be for example a pair of video streams taken from a fixed pair of cameras, we can choose between the following alternatives:

- Instantiate a farm with as many workers as needed (or as possible) and make them compute the sequential algorithm, thus reducing the aver-



(a) (b) (c) (d)

Figure 4.4: Detail of the implementation of the aggregation of costs over support regions. Colors represent different support structures, and colored arrows mean that we read data from the support structure denoted with such color.

age service time almost linearly on the number of workers, but losing the order of the processed images. In order to avoid load unbalancing on the workers we can compute an estimation of μ_{arm_length} in the emitter, by computing crosses in a subset of the image pair, and choose a proper policy for the assignment of tasks to the workers, as for example keeping an accumulator for each worker and summing there all the values of μ_{arm_length} for the tasks assigned to that worker, and then when a new task arrives assigning it to the worker with least value. This choice depends of course on the nature of the streaming.

- Instantiate a farm with data parallel workers, that is workers that are themselves farms, implementing any data parallel choice possible. This approach is a generalization to the previous one, and can be considered when the sequential code is too slow to achieve a target latency. Using this approach in fact a latency reduction can be achieved, limited only by the data parallel implementation used as a worker.

- Exploit the intrinsic pipeline structure of the algorithm to implement a low latency pipeline, as good as possible with the available hardware. Such a solution would keep the order of the images processed and thus would be suitable for example for video processing. Depending on the target architecture we can choose to proceed with a combination of the following steps:

  – Stage balancing: since the logical stages are many and have different costs, we can need to group them in order to obtain bigger stages with similar costs, or split them in substages for the same reason.

  – Stage parallelization: we split the computation of a stage/substage/stage group using farms in order to lower the latency.

We can immediately see in table 6.7 that time elapsed during Support size computation, Outlier detection and all the stages remaining after Iterative region voting is negligible, thus in order to balance service times in our pipeline we can merge them with their previous stage obtaining the stages showed in figure 4.5.

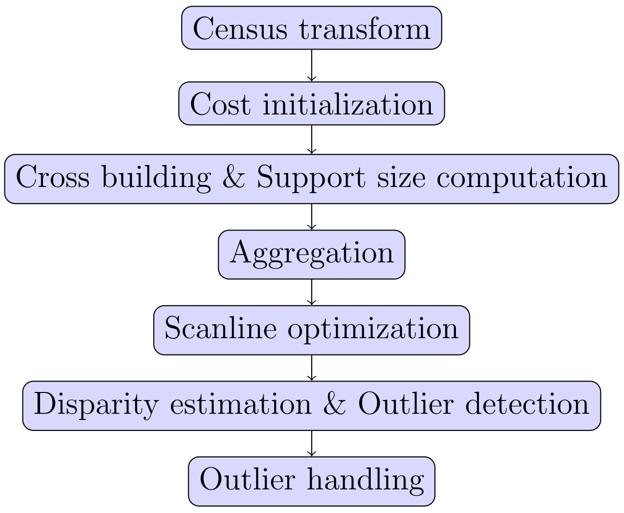


Figure 4.5: Stream parallel implementation pipeline stages.

In Census transform, Cost initialization, Cross building & Support size computation and Scanline optimization stages we can reduce the service time and latency almost linearly (view table 6.7) by implementing them with a farm.
Aggregation and Outlier handling stages must be treated differently, because as we already discussed in section 4.2, the integral based $O(1)$ implementation of the aggregation phase scales badly, and the same holds for Outlier handling because it spends most of its time in the Iterative voting substage, which has a similar $O(1)$ implementation and consequently the same issues. Even though a very slight improvement both in latency and service time can be obtained with an embarassingly parallel approach, if we are interested only in service time reduction only we can achieve significant improvement by exploiting the algorithm structure. Both Aggregation and Iterative voting stages in fact are iteration of a given substage for a number of times (four for Aggregation and five for Iterative voting), and thus we can think to reduce their service time by implementing explicitly every iteration as an independent stage. Aggregation

can be split further by exploiting the fact that both left-right and right-left aggregation are computed, and they are mutually independent.
Explicit structures for the aggregation steps are:

- Use a single execution unit, in this case we have

$$T_S = T_{aggr} \; L = T_{aggr}$$

- Use two execution units, splitting aggregation LR and aggregation RL, thus actually halving times, obtaining

$$T_S = \frac{T_{aggr}}{2} \; L = \frac{T_{aggr}}{2}$$

- Use more than two execution units, in this case we can't hope to reduce both latency and service time significantly, as we already said that splitting horizontal and vertical integrals computation between execution units does not give us an optimal performance. However we can split the aggregation step in its substeps and aggregating them in order to obtain a balanced time service. This allow us to reduce the service time $2 - 4$ times leaving the latency almost unaltered. Notice that to have a balanced result we need to add two execution units at a time.

whereas for Outlier handling as we said we can only split the computation of Iterative voting in the single iterations, this way we expect to obtain a service time reduction almost linear in the number of the execution units used.

Aggregation and Scanline optimization are the most time expensive stages in the algorithm so when implementing a stream parallel algorithm we must firstly think to spend extra available execution units to lower their service time and when possible their latency. Moreover, since the latency of these stages can be considerably high with respect to the others, we may think to merge other stages in order to free execution units to make possible a

reduction of the overall service time, especially when the number of cores of the target architecture is low.
Suppose for example that the algorithm stops just after the aggregation stage, and that we have the following times for Census transform, Cost initialization, Cross building & Support size computation and Aggregation

$$T_{census} = 150 \; T_{cost} = 550 \; T_{cross} = 400 \; T_{aggr} = 1500$$

and only 4 execution units available. Implementing every stage on a different execution unit would result in the following values for latency and service time:

$$T_S = \max\{T_{census}, T_{cost}, T_{cross}, T_{aggr}\} = T_{aggr} = 1500$$

$$L = T_{census} + T_{cost} + T_{cross} + T_{aggr} = 2600$$

this would allow us to process images at less than 1 frame per second with an overall latency of more than 2 seconds. Merging Census transform and Cost initialization in a single execution unit and using the exceeding execution unit to halve both latency and time service of Aggregation we obtain:

$$T_S = \max\left\{T_{census} + T_{cost}, T_{cross}, \frac{T_{aggr}}{2}\right\}$$

$$= \max\{700, 400, 750\} = 750$$

$$L = T_{census} + T_{cost} + \frac{T_{aggr}}{2} = 1850$$

doubling the number of frame per second we can process and reducing also the overall latency significantly.

Chapter 5

Implementation details

The code for the ADCensus algorithm has been written in C++, using the FastFlow framework described in section 2.2 for parallel implementations exploiting different features depending on the type of performance needed in the single implementations.

5.1 Phase-wise considerations

5.1.1 Census transform

Census transform is computed on 9×7 windows around every pixel, in order to both give enough information about the neighbourhood and store the transformed value of a pixel efficiently in a 64 bit wide variable.

5.1.2 Cost initialization

In this phase we need to compute the cost of pairs of pixels at disparity d for the whole (fixed) disparity range. The census cost component in the cost value is obtained by computing the Hamming distance between pairs of census values c_1, c_2 corresponding to the pixel pair considered. Hamming distance can be computed by counting the 1 bits of c_1 `xor` c_2, but looping through all 64 bits of c_1 `xor` c_2 for every reference pixel and every possible disparity value can become rather time expensive. The GNU C++ compiler provides the builtin function `__builtin_popcountll` that exploits lookup tables in order to compute the number of 1 bits in a 64 bits variable, and thus resulting in a faster computation. Another critical computation in this phase is the exponentiation, which is rather time expensive. Since best precision exponentiation values are no better than opportunely approximated ones in order to obtain more accurate disparity estimations, we use a less time-expensive approximated exponential.

5.1.3 Aggregation

The aggregation phase is computed using the integral histogram approach described in chapter 3 and specifically the implementation proposed by [12]. We use two support structures, each one $w \times h \times (DMAX + 1)$ wide, the first one being initialized during cost initialization, specifically

$$\texttt{\_\_aggr}[\texttt{0}][\texttt{i}][\texttt{j}][\texttt{d}] = \mathit{cost}\left(I_1(x,y), I_2(x-d,y)\right)$$

Then for every iteration five steps are performed:

- Every element in `__aggr[0]` is summed with the one at its left, for every value of d, visiting left to right and resulting in an horizontal integral, namely
$$\texttt{\_\_aggr}[\texttt{0}][\texttt{i}][\texttt{j}][\texttt{d}] += \texttt{\_\_aggr}[\texttt{0}][\texttt{i}][\texttt{j}-\texttt{1}][\texttt{d}]$$
- Horizontal integrals are exploited to compute the cost of each horizontal arm of every cross of every pixel, and such results are saved in the second support structure
$$\texttt{\_\_aggr}[\texttt{1}][\texttt{i}][\texttt{j}][\texttt{d}] = \texttt{\_\_aggr}[\texttt{0}][\texttt{i}][crs \to x^+][\texttt{d}] - \Delta_{aggr}$$
where
$$\Delta_{aggr} = \begin{cases} 0 & \text{if } crs \to x^- = 0 \\ \texttt{\_\_aggr}[\texttt{0}][\texttt{i}][crs \to x^- - \texttt{1}][\texttt{d}] & \text{otherwise} \end{cases}$$

- We treat in a similar way the other support structure, this way summing up to down to obtain a vertical integral image.

$$\texttt{\_\_aggr}[\texttt{1}][\texttt{i}][\texttt{j}][\texttt{d}] += \texttt{\_\_aggr}[\texttt{1}][\texttt{i}-\texttt{1}][\texttt{j}][\texttt{d}]$$

- And write finally on the first support structure the results of vertical aggregation:

$$\texttt{\_\_aggr}[\texttt{0}][\texttt{i}][\texttt{j}][\texttt{d}] = \texttt{\_\_aggr}[\texttt{1}][crs \to y^{+}][\texttt{j}][\texttt{d}] - \Delta_{aggr}$$

  where

$$\Delta_{aggr} = \begin{cases} 0 & \text{if } crs \to y^{-} = 0 \\ \texttt{\_\_aggr}[\texttt{1}][crs \to y^{-}-\texttt{1}][\texttt{j}][\texttt{d}] & \text{otherwise} \end{cases}$$

- Since support regions can have different dimensions for different pixels, in order to keep the cost value significative we normalize the overall cost obtained in the first support structure by dividing by the number of pixels in the support region, that is computed in a previous step with an analogous approach.

since the algorithm proposes to iterate four times the aggregation procedure switching between HV and VH support region every time, steps 1 and 2 become steps 3 and 4 and viceversa in the second and fourth iterations, taking care to keep alternating the support structures used (read 0, write 1, read 1, write 0, normalize).

5.1.4 Iterative voting

During this stage we need to build disparity histograms for outliers in their support region. Mei et. al explicitly specify that only non-outlier disparity must be collected in such histogram. The implementation is performed using integral images similarly to the ones used in the aggregation phase.

Specifically, we initialize the first support structure as follows:

$$\texttt{\_\_hist}[\texttt{i}][\texttt{j}][\texttt{d}] = \begin{cases} 1 & \text{if } I_1(j,i) \text{ is not an outlier} \\ 0 & \text{otherwise} \end{cases}$$

In this case we only perform HV aggregation of these disparity contributes. After the aggregation $\texttt{\_\_hist}[\texttt{i}][\texttt{j}][\texttt{d}]$ represent exactly the number of non-outlier pixels in the support region who are assigned a disparity d.

5.1.5 Proper interpolation

Interpolation consists in two steps: selection of the nearest significative neighbours along 16 directions, and choice of the disparity value from one of such neighbours following a different rule for mismatched and occlused outliers. In the first step we initialize a vector with 16×2 elements, namely

$$\texttt{\_\_dirs}[\texttt{k}] = \left(\cos\left(\frac{2k\pi}{16}\right), \sin\left(\frac{2k\pi}{16}\right)\right) \quad 0 \leq k < 16$$

then for every pixel (i,j) we visit `__outliers` as follows

$$\texttt{\_\_outliers}[\texttt{i} + \texttt{r} * \texttt{\_\_dirs}[\texttt{k}][1]][\texttt{j} + \texttt{r} * \texttt{\_\_dirs}[\texttt{k}][0]] \quad 0 \leq k < 16$$

where `r` is initially 1 and is increased until at least one visited direction does not contain an outlier, and `r * __dirs[k][*]` is rounded to the nearest integer. In this way we simulate a visit of the image by concentric circles at increasing radius r and thus the neighbours found are actually as close as possible to the considered pixel. Disparity selection is then performed as explained in chapter 3.

5.1.6 Edge detection

Edge detection is performed using a 3×3 laplacian filter.

5.2 Parallel implementations

5.2.1 Data parallel

The data parallel version of fastADCensus has been developed by implementing a FastFlow farm. For convenience the overall code has been split in functions whose inner code could be computed concurrently without any need of synchronization. Specifically, a single function is of the form: where

```
void tiny_phase(...) {
    for (int i = 0; i < image_rows; ++i) {
        for (int j = 0; j < image_cols; ++j) {
            // In some stages this loop is present as well
            for (int d = 0; d <= max_disp_lv; ++d) {
                // do something ...
            }
        }
    }
}
```

sometimes, as for example during the scanline optimization phase, some index could begin from 1, go backwards or both. Then their signature has been opportunely modified in order to implement actual concurrent code.

```
void tiny_phase(...) {}
// Becomes
void tiny_phase(int worker_id, ...) {}
```

The new parameter is then exploited in three alternative ways, showed in table 5.1:

- Horizontal stripes, used for example for horizontal aggregation and left to right or right to left scanline optimization.
- Vertical stripes, used for example for vertical aggregation.

- Horizontal subsampling, used in cross building phase to avoid unbalancing.

```
// Horizontal stripes
void tiny_phase(int worker_id, ...) {
    for (int i = (image_rows * worker_id) / nworkers;
        i < (image_rows * (worker_id + 1)) / nworkers; ++i) {
        for (int j = 0; j < image_cols; ++j) {
            // ...
        }
    }
}

// Vertical stripes
void tiny_phase(int worker_id, ...) {
    for (int i = 0; i < image_rows; ++i) {
        for (int j = (image_cols * worker_id) / nworkers;
            j < (image_cols * (worker_id + 1)) / nworkers; ++j) {
            // ...
        }
    }
}

// Horizontal subsampling
void tiny_phase(int worker_id, ...) {
    for (int i = worker_id; i < image_rows; i += nworkers) {
        for (int j = 0; j < image_cols; ++j) {
            // ...
        }
    }
}
```

Table 5.1

Finally, everything described was inserted in a FastFlow context implementing three class specializations for `ff_node`, one for the emitter, one for the

collector and one for all workers, and inserting them in a wrapped around `ff_farm`.
The emitter generates *nworkers* tasks for the first phase sending to the workers the function to be computed together with the value of `worker_id` (which is different for every task generated) and any other possible paramether for the phase function. Then every time that the collector signals him the end of a task computation it generates *nworkers* tasks for the subsequent phase. Any worker receives a task indicating a function to compute and its parameters and it executes it with the given parameters.
The collector waits for *nworkers* tasks to arrive and then signals the emitter that the current phase is ended.

5.2.2 Stream parallel

In order to obtain a stream parallel implementation we kept the use of tiny stage functions, further modified, and we used them with a different approach.
First of all, all support structures for the processing of a single image are grouped in a single big support structure representing an element of our stream. Every element in the stream need his copy of any support structure. Then we add another parameter to all the functions modifying every access

```
    void tiny_phase(stream_el* se, [int worker_id,] ...) {}
```

to any support structure inside every function in order to refer to the local structure of the single stream element. The `worker_id` parameter at this point become useful only in case we want to perform some intermediate stage in a data parallel fashion.
At this point we developed two alternative stream parallel implementations: a pipeline version and a farm of farms version, parametric in the number of data parallel workers and in their degree of parallelism.

In the first implementation, stage functions are opportunely grouped in FastFlow nodes and inserted in a FastFlow pipeline (`ff_pipeline`). In order to avoid initial latency due to memory allocations, support structures are allocated in the node that first use them, and deallocated in the last node that use them, using FastFlow allocators and deallocators.
In the second one instead, we initially allocate exactly $n_{datapar}$ tasks and then we proceed exactly as in the data parallel case, doing the following modifications:

- The emitter uses a custom load balancer, assigning computations on an element of the stream always to the same set of workers, in order to guarantee locality and simulating an actual farm of farms solution, which is not currently possible on FastFlow when the workers have feedback as in our case. Once one image has been fully processed, its corresponding task is overwritten with the first unprocessed element of the stream without memory allocations, and thus the memory footprint is constant during the whole stream processing.
- The collector uses $n_{datapar}$ independent counters instead of one, one for each data parallel worker, and update them consistently.

Chapter 6

Experimental results

In this chapter we will show a comparison the execution time of our sequential code with respect to the times publieshed by Mei et al. Then we show times and scalability results in the data parallel code, trying it both in a multicore architecture and a many core one. From a stream parallel point of view we show service time results for a simple pipeline implementation of fastADCensus on a multicore architecture, more suitable for a low parallelism degree setting like ours. Then we present another solution based on a farm with data parallel workers, making a projection of the parallelism degree necessary to achieve a given target frame rate by developing a model parametric in the frame rate value and the data parallel latency. Firstly we use this model projecting the parallelism degree necessary on the multicore architecture, that cannot be tested for a physical limitation. Then we test the model on the many core architecture, fixing a target frame rate and showing how the execution behave in practice with respect to that target. Finally we give a comparison with the existing implementation of stereo algorithms provided by OpenCV both from an accuracy point of view and a time expensiveness one.
The architectures used during the testing phase are:

- A CPU multicore architecture with 2 Intel Xeon E5-2650 @ 2.00GHz (8 cores each), 32GB RAM memory and Linux 2.6.32 OS.

- A many core architecture: an Intel Xeon Phi coprocessor 5100 with 60 cores, 8GB of RAM memory and Linux 2.6.38.8 mpss3.4.

6.1 Sequential

The sequential code was tested in the Middlebury evaluation dataset, with the disparity range suggested there, that must be equal for every algorithm in order to submit the evaluation.
Testing fastADCensus on the multicore architecture gave us better results than the ones achieved by Mei et al. Execution times are shown in table 6.1 together with their execution times, the range of disparity levels used, and the speedup achieved with fastADCensus.

	Teddy	Cones	Venus	Tsukuba
Disparity range	0-59	0-59	0-19	0-15
fastADCensus time (ms)	3578	3571	1716	927
ADCensus time (ms)	15000	15000	4200	2500
Speedup	4.19	4.20	2.45	2.70

Table 6.1: Detail of times on the sequential code for fastADCensus and the original ADCensus

The better performance of our implementation is significative since the tests of Mei et al. were executed on an Intel Core 2 Duo @ 2.20GHz, an older CPU and thus slightly faster in sequential fragments of code.

6.2 Data parallel

The data parallel implementation descripted in sec. 4.2 on the multicore achieves at best an execution time $\sim$ 44 times faster ($\sim$ 3.5 times faster with a single worker) with respect to the Mei et al.'s implementation on CPU, and $\sim$ 3.6 slower with respect to their GPU implementation on the Teddy image pair in the Middlebury data set. Table 6.7 shows a detail of times and scalability for the Teddy image pair. Results obtained can reduce considerably the latency with respect to the original implementation, even though it is not fast enough for real time processing. At best it can achieve a frame rate of 2.93 frames per second with a latency of 341 ms. It can be seen from the data that Aggregation and Iterative voting are the bottlenecks for this algorithm (see sec. 4.2).

Testing the same code on the many core architecture gave us similar results for scalability (see fig. 6.1). We have better scalability values initially for Aggregation and Iterative voting, since we can exploit better the caches and the computation is slower. These factors lead altogether to a better amortization of the memory access costs, but at a certain point (4 for Iterative voting, 16 for Aggregation) they begin to become inefficient exactly as in the multicore case, for the same reasons. Scanline Optimization start to scale suboptimally due to the branching inside the code. Census transform has a similar behaviour, but its computation is more unbalanced because of an heavy use of branches and thus starts scaling suboptimally earlier. Overall scalability is better up to 60 nodes, as we are using just one context per core, and keep improving with a worse efficiency up to 120, with a jump in performances beetween 60 and 61 when we start using two contexts per core and both the ALUs available.

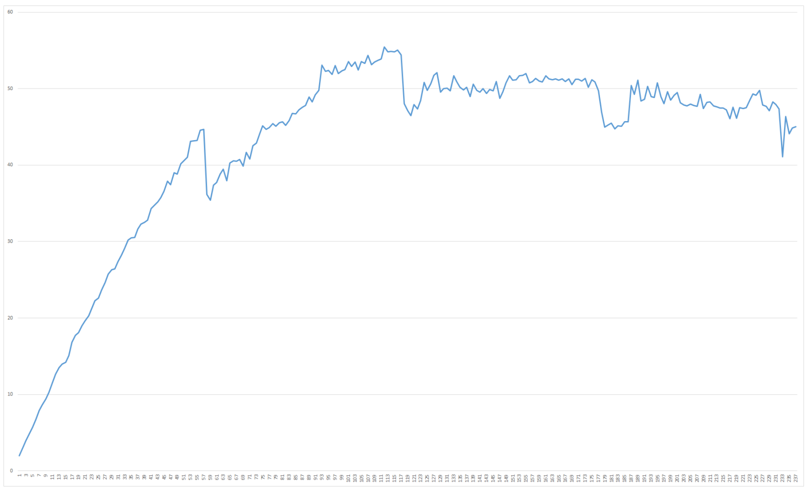


Figure 6.1: Scalability on the Xeon Phi

6.3 Stream parallel

The first stream parallel algorithm implemented is the pipeline version described in section 4.3. Cost initialization and Cross building & Support size computation stages are merged and computed with 2 nodes, one for the left image and one for the right one, Aggregation stages too uses 2 nodes, one for the LR costs and one for the LR costs, and Scanline optimization is computed with 4 nodes, one for each scanline direction, for a total of 15 nodes. It achieves a latency of $\sim 2.4s$ and service time of $\sim 510ms$ with a bottleneck in the Aggregation stage.

The second implementation developed has been the "farm of farms" one, (see sec. 4.3). This approach allow us to choose the best tradeoff between latency and and service time, and in principle it can be used to reduce arbitrarily the service time, and thus the frame rate.

Suppose we have a target number of frames per second f^*, and a farm where every worker is itself a data parallel skeleton. Its latency is parametric in its parallelism degree:

$$T_w(n_w) = T(n_w)_{datapar}$$

The service time T_S of the main farm, considering negligible the cost of the emitter and the collector, and using $n_{datapar}$ data parallel workers, is given by

$$T_S = \max\{T_e, T_c, \frac{T_w}{n_{datapar}}\} = \frac{T_w}{n_{datapar}}$$

In order to obtain the target frames per second value, we need to obtain a target service time T_S^*, of

$$T_S^* = \frac{1000}{f^*}$$

where times are indicated in milliseconds. Finally, the target number of data parallel workers $n^*_{datapar}$ can be calculated by setting $T_S = T_S^*$ obtaining

$$n^*_{datapar}(n_w) = \left\lceil \frac{f^* \times T_w(n_w)}{1000} \right\rceil = \left\lceil \frac{f^* \times T(n_w)}{1000} \right\rceil$$

	$n_w = 1$	2	3	4	5	6	7	8
n_{cores}	102	104	105	108	110	114	112	112
Latency	4061	2058	1387	1049	854	723	639	555

	nw = 9	10	11	12	13	14	15
n_{cores}	117	120	121	120	130	126	135
Latency	499	457	417	395	366	342	325

Table 6.2: Projection of the number of cores necessary to implement a farm of data parallel farms suitable to process a stream at 25fps on a Sandy Bridge architecture, and corresponding latency of a single task.

The total number of execution units n_{nodes} (for workers only) is then given by

$$n_{nodes} = n_w \times n^*_{datapar}(n_w)$$

Table 6.2 shows the overall number of cores that would be necessary in order to process a stream at a frame rate of 25 frames per second and the corresponding latency with different values for n_w for the times in table 6.7. These values are a purely theoretical projection, as at the current time we only have available 48 cores on the Sandy Bridge architecture.
In order to have a confirmation of the theoretical model, even with lower target frame rates, a many core architecture is more suitable due to his higher core count. We thus tested this solution on the Xeon Phi many core architecture with a stream of images. We used smaller images for accomplish for memory limitations and targeting a frame rate of 10fps. We computed the number of nodes and then tried to execute a farm with that number of nodes and parallelism degree n_w on a single worker, obtaining the results shown in table 6.3. Note how the ceiling operation in our model allow us to obtain actually higher frame rate values. This is easily explained: we use the smallest number of data parallel workers that is enough to obtain a frame rate of at least 10fps. The value of $\frac{f^* \times T(n_w)}{1000}$ decreases when the parallelism degree is higher, since the target frame rate f^* is fixed. If the ceiling operation for different n_w result in the same value, as it happens in the range 9-12 and again in the range 13-20, we obtain higher values for n_{nodes} the higher is n_w,

and thus better performances. The only exception in our data is $n_w = 20$, but the anomaly is due to the fact that we use more than 60 threads overall (60 for the data parallel workers + 2 for emitter and collector) and the Xeon Phi have a jump in performances between 60 and 61 (see fig. 6.1 and sec. 6.2) and the model is unsuitable in case of such discontinuities.

	$n_w = 1$	2	3	4	5	6	7	8	9	10	$n_w = 11$	12	13	14	15	16	17	18	19	20
n_{cores}	31	32	33	36	35	36	35	40	36	40	44	48	52	42	45	48	51	54	57	60
Latency	3088	1584	1071	820	661	561	495	435	392	366	373	326	301	290	267	255	233	229	214	208
Frame rate	10.00	10.18	10.45	11.29	11.01	11.23	10.03	12.25	10.26	11.58	12.73	13.81	15.16	11.95	12.46	12.75	14.63	14.79	16.14	14.22

Table 6.3: Experimental results on the Xeon Phi of a stream parallel farm obtained using the parallelism degree theorized with our model.

With the same image size, we can aim at a frame rate of 25 fps, requiring the node count showed in table 6.4. These results were not tested because, as discussed before, our model is not suitable for more than 60 threads overall.

	$n_w = 1$	2	3	4	5	6	7	8	9	10	$n_w = 11$	12	13	14	15	16	17	18	19	20
n_{cores}	78	80	81	84	85	90	91	88	90	100	110	108	104	112	105	112	102	108	114	120

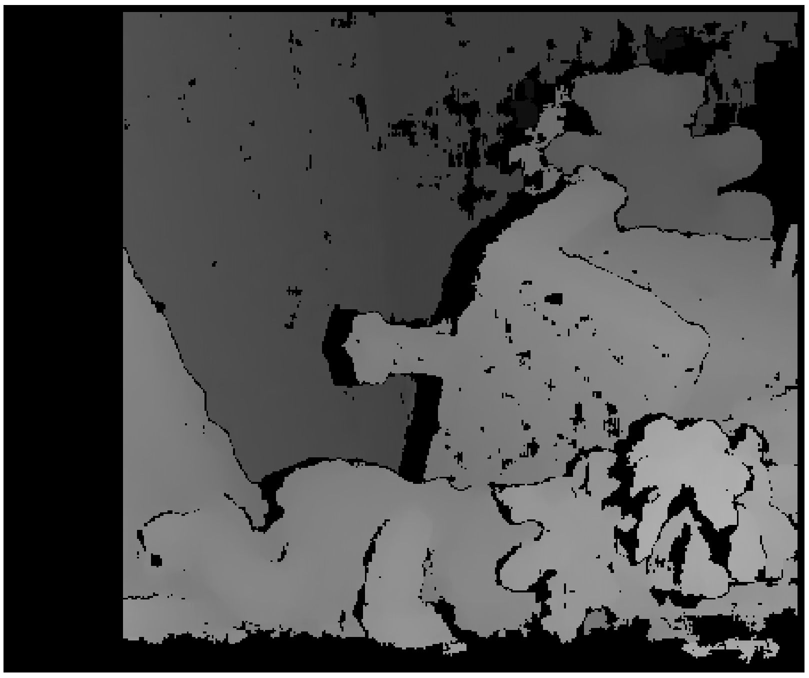
Table 6.4: Projection with our model of the number of cores needed on the Xeon Phi with a target frame rate of 25 fps. These values can not be tested because of the worsening of performance for more than 60 nodes.

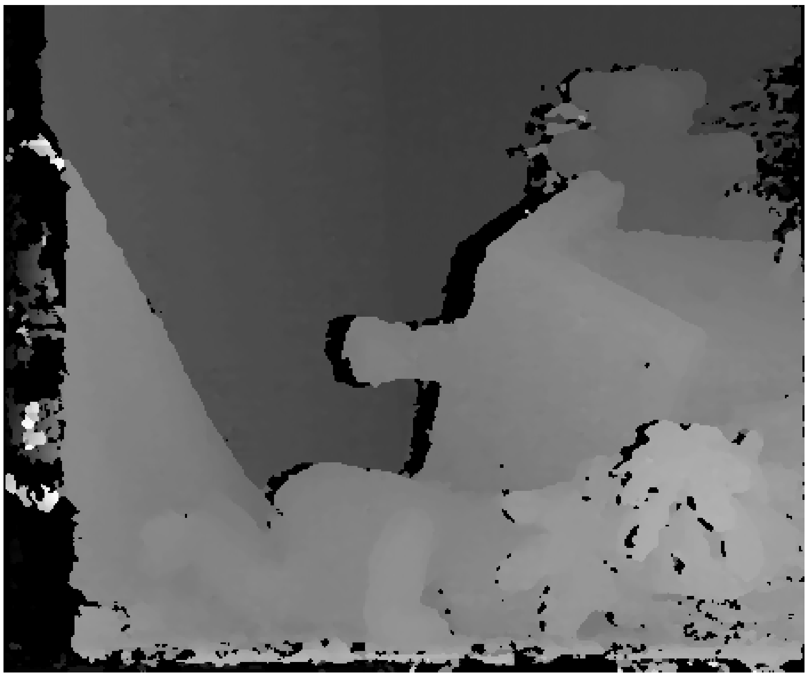
6.4 Comparison with OpenCV

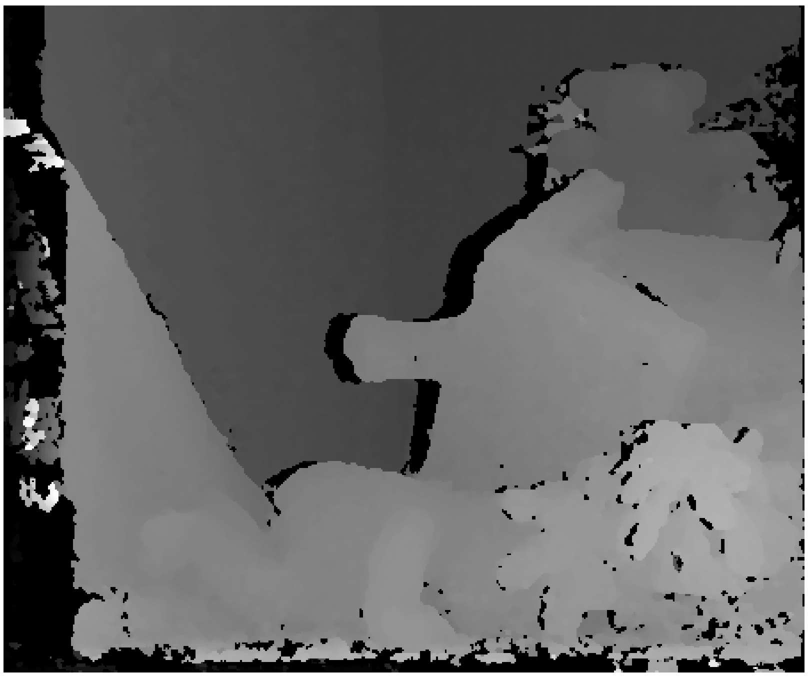
OpenCV provides different image matching algorithms based on known stereo vision results. Their performance is notably better with respect to the implementation showed, but at a substantial price in terms of accuracy. Referring again to the Teddy image pair, computing a dense disparity map for a disparity range from 0 to 59 achieves the results showed in figure 6.2. As for accuracy, the difference can be appreciated at a glance, and is confirmed by the Middlebury test as shown in table 6.5

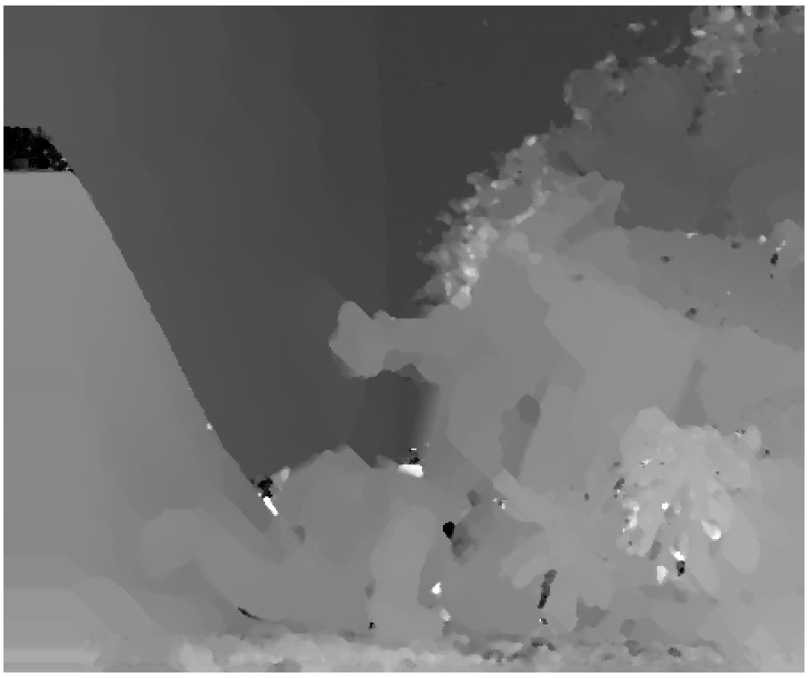
	nonocc	all	disc	Exec. time (ms)
OpenCV (bm)	28.4	35.8	45.6	22
OpenCV (sgbm)	11.4	20.5	25.9	150
OpenCV (hh)	11.8	20.9	27.2	196
OpenCV (var)	26.1	31.1	36.7	886
fastADCensus	7.57	14.7	16.5	3381

Table 6.5: Error percentage taken from the Middlebury evaluation system of stereo matching algorithms


(a) OpenCV (bm)


(b) OpenCV (sgbm)


(c) OpenCV (hh)


(d) OpenCV (var)

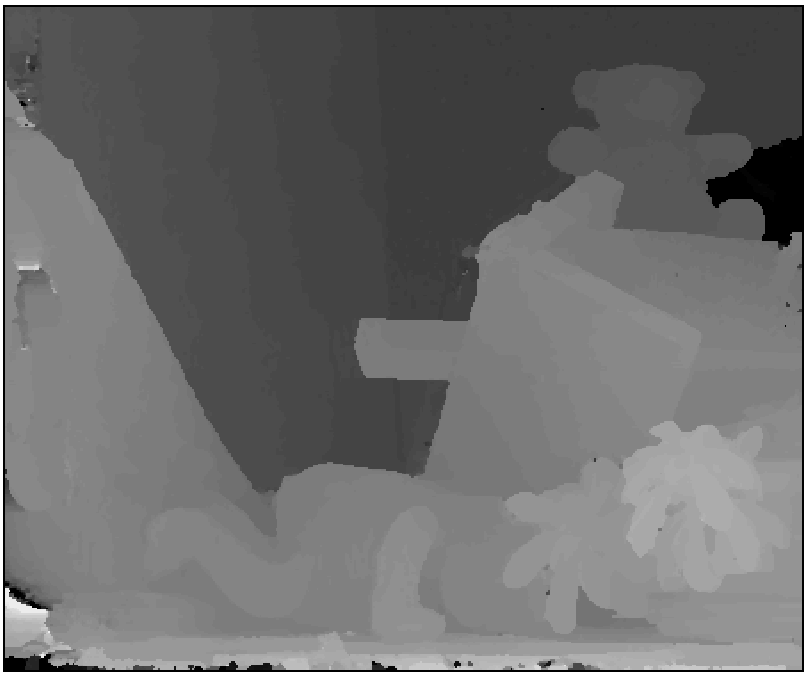

(e) fastADCensus

Figure 6.2: Results of OpenCV algorithms and our fastADCensus on the Teddy image pair

6.5 Conclusions

FastADCensus is faster than the original ADCensus CPU implementation in the Middlebury dataset, and can achieve reasonably good execution times also compared to the GPU implementation, being less than 4 times slower with respect to that implementation. It produces better results than similar routines provided by OpenCV at a price of a higher time consumption. From a data parallel point of view, good results have been achieved for scalability, and latency can be reduced up to 341ms on the multi core architecture and up to 672ms on the many core architecture for a 375×450 image pair. This corresponds respectively to 8.5 frames and 16.8 frames in a 25fps stream. The same code has been tested on the many core architecture, obtaining similar results for scalability, at an expected bigger latency. Stream parallelism is achieved with better results implementing a farm of data parallel farms instead of a pipeline. On the multi core case, the best frame rate we can achieve with the farm of farms implementation (at lowest latency) is 3.09fps, with a latency of 660ms with full scale images. This is considerably better than the pipeline implementation which achieves a latency of 4.2s and a frame rate of 1.96fps. On the many core case we used 30% scaled images (135×113 pixels) and the best frame rate achievable is 16.14fps with a latency of 214ms.

	Time (ms)							
	T(1)	T(2)	T(3)	T(4)	T(5)	T(6)	T(7)	T(8)
Census transform	127	67	47	36	29	25	22	20
Cost initialization	495	249	166	125	100	85	73	63
Cross building	261	130	87	66	52	44	36	32
Support size computation	6	6	6	6	6	4	5	6
Aggregation	1122	566	380	290	240	204	188	161
Scanline optimization	1440	731	487	367	295	246	212	186
Disparity estimation	64	34	23	16	14	10	10	8
Outlier detection & handling	546	275	191	143	118	105	93	79
TOT	4061	2058	1387	1049	854	723	639	555

	Time (ms)						
	T(9)	T(10)	T(11)	T(12)	T(13)	T(14)	T(15)
Census transform	19	16	16	14	14	13	13
Cost initialization	56	51	47	43	39	37	34
Cross building	28	26	24	22	20	18	16
Support size computation	5	6	4	6	6	6	6
Aggregation	145	131	123	113	103	102	98
Scanline optimization	166	150	135	125	115	108	100
Disparity estimation	7	6	6	6	4	4	4
Outlier detection & handling	73	71	62	66	65	54	54
TOT	499	457	417	395	366	342	325

	Scalability						
	sc(2)	sc(3)	sc(4)	sc(5)	sc(6)	sc(7)	sc(8)
Census transform	1.9	2.7	3.53	4.38	5.08	5.77	6.35
Cost initialization	1.99	2.98	3.96	4.95	5.82	6.78	7.86
Cross building	2.01	3	3.95	5.02	5.93	7.25	8.16
Support size computation	1	1	1	1	1.5	1.2	1
Aggregation	1.98	2.95	3.87	4.68	5.5	5.97	6.97
Scanline optimization	1.97	2.96	3.92	4.88	5.85	6.79	7.74
Disparity estimation	1.88	2.78	4	4.57	6.4	6.4	8
Outlier detection & handling	1.99	2.86	3.82	4.63	5.2	5.87	6.91
TOT	1.97	2.93	3.87	4.76	5.62	6.36	7.32

	Scalability						
	sc(9)	sc(10)	sc(11)	sc(12)	sc(13)	sc(14)	sc(15)
Census transform	6.68	7.94	7.94	9.07	9.07	9.77	9.77
Cost initialization	8.84	9.71	10.53	11.51	12.69	13.38	14.56
Cross building	9.32	10.04	10.88	11.86	13.05	14.5	16.31
Support size computation	1.2	1	1.5	1	1	1	1
Aggregation	7.74	8.56	9.12	9.93	10.89	11	11.45
Scanline optimization	8.67	9.6	10.67	11.52	12.52	13.33	14.4
Disparity estimation	9.14	10.67	10.67	10.67	16	16	16
Outlier detection & handling	7.48	7.69	8.81	8.27	8.4	10.11	10.11
TOT	8.14	8.89	9.74	10.28	11.1	11.87	12.5

Table 6.7: Details of time and scalability of the data parallel implementation run on the multicore architechture

Chapter 7

Conclusions

The thesis was aimed at studying state of the art stereo matching algorithms, keeping an eye both at their accuracy as given by the Middlebury ranking and their parallelization possibilities. The ADcensus algorithm has been chosen and then implemented both sequentially and in parallel, obtaining perceptible improvements in performance with respect to the authors'published results.
A theoretical model for determining the parallelism degree of a stream parallel farm capable to process a stream at any given frame rate has been developed. The goal of our thesis was to obtain a low latency / real time stream parallel implementation for fastADCensus, so we tested the model on a many core architecture in order to have a higher parallelism degree available. The model has been proved to be correct up to physical parallelism limitations, allowing us to achieve a frame rate of 16fps for 135×113 image pairs.

During this thesis, the implementation effort was dedicated to fill all the details left unsaid in the work of Mei et al. in order to obtain an actual implementation, and to all the modifications necessary to properly exploit parallelism on multi and many core architectures keeping the original code. This approach brought us to obtain a code for fastADCensus which is scalable but does not exploit explicitly vectorization. When compiled with `icc`, the code is slightly vectorized without appreciable differences ($< 2\%$) in terms of execution times with respect to the same code compiled with `g++`. In the future, we will study how and if some stages can be reformulated in order to obtain a vectorizable code, or to avoid memory dependencies or branches.

Bibliography

[1] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002.

[2] URL `http://vision.middlebury.edu/stereo/submit/`.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). *Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing*, 2011.

[4] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[5] Takeo Kanade and Masatoshi Okutomi. A stereo matching algorithm with an adaptive window: Theory and experiment. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(9):920–932, 1994.

[6] Kuk-Jin Yoon. Adaptive support-weight approach for correspondence search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):650–656, 2006.

[7] Myung-Ho Ju and Hang-Bong Kang. Constant time stereo matching. In *Machine Vision and Image Processing Conference, 2009. IMVIP'09. 13th International*, pages 13–17. IEEE, 2009.

[8] Fatih Porikli. Constant time o (1) bilateral filtering. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[9] Asmaa Hosni, Michael Bleyer, Margrit Gelautz, and Christoph Rhemann. Local stereo matching using geodesic support weights. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 2093–2096. IEEE, 2009.

[10] Xing Mei, Xun Sun, Mingcai Zhou, Haitao Wang, Xiaopeng Zhang, et al. On building an accurate stereo matching system on graphics hardware. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 467–474. IEEE, 2011.

[11] Ramin Zabih and John Woodfill. Non-parametric local transforms for computing visual correspondence. In *Computer Vision—ECCV'94*, pages 151–158. Springer, 1994.

[12] Ke Zhang, Jiangbo Lu, and Gauthier Lafruit. Cross-based local stereo matching using orthogonal integral images. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(7):1073–1079, 2009.

[13] Heiko Hirschmüller. Stereo processing by semiglobal matching and mutual information. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2):328–341, 2008.

Appendix A

Source code

In this appendix we list the full implementation of fastADCensus. Section A.1 contains the code for the business logic, almost identical to the sequential code. The core is split in 14 files, each implementing a single phase. Section A.2 contains the code used to implement the parallel version. Since the skeleton adopted is a farm, we have one file for the emitter code, one for the worker code and one for the collector code. The other files not discussed contain support structures, common values and common definitions.

A.1 Business logic

A.1.1 main.cpp

```cpp
#include <iostream>
#include <cmath>
#include <string>

#include "common.hpp"

#include "farm/emitter.hpp"
#include "farm/worker.hpp"
#include "farm/collector.hpp"

```

```
#include <vector>

using namespace std;
using namespace ff;

void __usage() {
    cout
        << "Usage: adcensus nodes nworkersDP streamsize left_image right_image"
        << endl;
}

void deserialize(const char* path, int* rows, int* cols, int* chans, uint8_t* t) {
    FILE* fd = fopen(path, "r");

    fread(rows, sizeof(int), 1, fd);
    fread(cols, sizeof(int), 1, fd);
    fread(chans, sizeof(int), 1, fd);

    for (int i = 0; i < *rows; ++i) {
        for (int j = 0 ; j < *cols; ++j) {
            fread(&t[__idx1(0, i, j)], (*chans)*sizeof(uint8_t), 1, fd);
        }
    }
    fclose(fd);
}

int main(int argc, char* argv[]) {
    if (argc != 6) {
        __usage();
        return 1;
    }

    __nnodes = atoi(argv[1]);
    __nworkers = atoi(argv[2]);
    __streamsz = atoi(argv[3]);

    __ndatapar = __nnodes / __nworkers;
    // Initializing image data
    int __rows, __cols, __chans;
    fdata = new uint8_t[2*ROWS*COLS*3];
    fdataG = new uint8_t[2*ROWS*COLS];
    deserialize(argv[4], &__rows, &__cols, &__chans, &fdata[__idx1(0, 0, 0)]);
    deserialize(argv[5], &__rows, &__cols, &__chans, &fdata[__idx1(1, 0, 0)]);

    // Converting images to grayscale
    for (int i = 0; i < __rows; ++i) {
        for (int j = 0; j < __cols; ++j) {
            uint8_t* pxl = &fdata[__idx1(0, i, j)];
            fdataG[__idx2(0, i, j)] =
                pxl[0] * 0.2126 + pxl[1] * 0.7152 + pxl[2] * 0.0722;
            pxl = &fdata[__idx1(1, i, j)];
            fdataG[__idx2(1, i, j)] =
                pxl[0] * 0.2126 + pxl[1] * 0.7152 + pxl[2] * 0.0722;
        }
    }

    // Allocating space __ndatapar tasks, every time data will be overwritten
    // in the right location
    __counters = new int[__ndatapar];
    for (int i = 0; i < __ndatapar; ++i) __counters[i] = __nworkers;
    __stream = new bigtask_t[__ndatapar];

    ff_farm<MyLoadBalancer> farm;
    SVEmitter e(farm.getlb());
    SVCollector c;

    vector<ff_node*> w;
    for (int q = 0; q < __ndatapar; ++q) {
        for (int i = 0; i < __nworkers; ++i) {
            w.push_back(new SVWorker);
        }
    }

    farm.add_emitter(&e);
    farm.add_workers(w);
    farm.add_collector(&c);

    farm.wrap_around();
    farm.run_and_wait_end();

    return 0;
}
```

A.1.2 common.hpp

```
#ifndef ADCENSUS_COMMON_HPP
#define ADCENSUS_COMMON_HPP

#include <cstdint>
#include <algorithm>

const float L_CENSUS    = 30.0;
const float L_AD     = 10.0;

const uint8_t CROSS_L1   = 34;
const uint8_t CROSS_L2   = 17;
const uint8_t CROSS_TAU1   = 20;
const uint8_t CROSS_TAU2   = 6;

const float OPT_PI1     = 1.0;
const float OPT_PI2     = 3.0;
const uint8_t OPT_TAUSO = 15;

const float ITER_TAUH   = 0.4;
const uint8_t ITER_TAUS   = 20;

#ifdef SMALL
const int ROWS = 113, COLS = 135;
#endif

```

```
#ifdef BIG
const int ROWS = 375, COLS = 450;
#endif

const int DMAX = 59;
const int DSCALE = 4;

inline static int __idx1(int lr, int r, int c, int ch = 0) {
    return (lr * (ROWS * COLS * 3) + r * (COLS * 3) + c * 3 + ch);
}

inline static int __idx1b(int lr, int r, int c, int ch) {
    return (lr * (ROWS * COLS * 3) + r * (COLS * 3) + c * 3 + ch);
}

inline static int __idx2(int lr, int r, int c) {
    return (lr * (ROWS * COLS) + r * COLS + c);
}

inline static int __idx3(int q, int lr, int r, int c, int d) {
    return (q * (2 * ROWS * COLS * (DMAX+1)) + lr *(ROWS * COLS * (DMAX+1))
        + r * (COLS * (DMAX+1)) + c * (DMAX + 1) + d);
}

inline static int __idx4(int q, int lr, int r, int c) {
    return (q * (2 * ROWS * COLS) + lr * (ROWS * COLS) + r * COLS + c);
}

inline static int __idx5(int lr, int dir, int r, int c, int d) {
    return (lr * (4 * ROWS * COLS * (DMAX+1)) + dir * (ROWS * COLS * (DMAX+1))
        + r * (COLS * (DMAX+1)) + c * (DMAX+1) + d);
}

inline static int __idx6(int lr, int r, int c, int d) {
    return (lr *(ROWS * COLS * (DMAX+1)) + r * (COLS * (DMAX+1))
        + c * (DMAX + 1) + d);
}
```

```

inline static int __idx7(int r, int c) {
    return r * COLS + c;
}

typedef struct cross_t {
  int8_t xm, xp, ym, yp;
} cross_t;

typedef enum outlier_t {
  OUT_OCCL, OUT_MISM, OUT_NONE
} outlier_t;

int __color_diff(uint8_t* p1, uint8_t* p2) {
  int r = std::abs(p1[0] - p2[0]);
  int g = std::abs(p1[1] - p2[1]);
  int b = std::abs(p1[2] - p2[2]);

  return std::max(std::max(r, g), b);
}

int __nworkers, __nnodes, __ndatapar, __streamsz;
uint8_t* fdata, * fdataG;
#endif
```

A.1.3 00_census_transform.hpp

```
#ifndef __00_CENSUS_HPP
#define __00_CENSUS_HPP

#include "../common.hpp"

void __census_init(bigtask_t* t, int worker_id, int lr) {
  int ifrom = (worker_id * t->__rows) / __nworkers;;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      t->__census_data[__idx2(lr, i, j)] = 0;
      int qfrom = std::max(-i, -4);
      int qto = std::min(4, (t->__rows - i) - 1);
      int rfrom = std::max(-j, -3);
      int rto = std::min(3, (t->__cols - j) - 1);
      for (int q = qfrom; q <= qto; ++q) {
        for (int r = rfrom; r <= rto; ++r) {
          uint8_t ig0 = t->__img_grey[__idx2(lr, i+q, j+r)];
          uint8_t ig1 = t->__img_grey[__idx2(lr, i, j)]
          if (ig0 > ig1)
            t->__census_data[__idx2(lr, i, j)] |= (1 << ((q+4)*7 + (r+3)));
        }
      }
    }
  }
}

#endif
```

A.1.4 01_cost_initialization.hpp

```
#ifndef __01_COSTINIT_HPP
#define __01_COSTINIT_HPP

#include "../common.hpp"

float __absolute_distance(bigtask_t* t, int i1, int j1, int i2, int j2) {
  float result = 0;
  for (int x = 0; x < 3; ++x) {
    result += std::abs(
      t->__img_data[__idx1b(0, i1, j1, x)] - t->__img_data[__idx1b(1, i2, j2, x)]
    );
  }
  result /= 3;
  return result;
}

```

```
static inline float
fasterpow2 (float p)
{
  float clipp = (p < -126) ? -126.0f : p;
  union { uint32_t i; float f; } v = {
  static_cast<uint32_t> ( (1 << 23) * (clipp + 126.94269504f) )
  };
  return v.f;
}

static inline float
fasterexp (float p)
{
  return fasterpow2 (1.442695040f * p);
}

void __cost_initialization(bigtask_t* t, int worker_id) {
  uint8_t cost_census;
  float cost_ad, cost_tot;

  int ifrom = (worker_id * t->__rows) / __nworkers;;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      for (int d = 0; d <= DMAX; ++d) {
        t->__aggregation[__idx3(0, 0, i, j, d)] = 2:
        t->__aggregation[__idx3(1, 0, i, j, d)] = 2;
      }
    }

    for (int j = 0; j < t->__cols; ++j) {
      int d;
      for (d = 0; d <= std::min(DMAX, j); ++d) {
        uint64_t census_string =
          t->__census_data[__idx2(0, i, j)]
          ^ t->__census_data[__idx2(1, i, j-d)];
        cost_census = __builtin_popcountll(census_string);
        cost_ad = __absolute_distance(t, i, j, i, j-d);
        cost_tot = (float)(2 - fasterexp((float)(-(float)cost_ad/L_AD))
          - fasterexp((float)(-(float)cost_census/L_CENSUS)));
        t->__aggregation[__idx3(0, 0, i, j, d)] = cost_tot;
        t->__aggregation[__idx3(1, 0, i, j-d, d)] = cost_tot;
      }
    }
  }
}

#endif
```

A.1.5 02_cross_building.hpp

```
#ifndef __02_CROSS_BUILD_HPP
#define __02_CROSS_BUILD_HPP

#include "../common.hpp"

void __cross_building(bigtask_t* t, int worker_id, int lr) {
  int8_t* vals[4];
  // Expressed as (rows, cols)
  int dirs[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

  for (int i = worker_id; i < t->__rows; i += __nworkers) {
    for (int j = 0; j < t->__cols; ++j) {
      vals[0] = &t->__crosses[__idx2(lr, i, j)].ym;
      vals[1] = &t->__crosses[__idx2(lr, i, j)].yp;
      vals[2] = &t->__crosses[__idx2(lr, i, j)].xm;
      vals[3] = &t->__crosses[__idx2(lr, i, j)].xp;
      for (int x = 0; x < 4; ++x) {
        bool go_on = true;
        int val = 0;
        while(go_on) {
          ++val;
          switch (x) {
            case 0:
```

```
          if (i-val == -1) { --val; go_on = false; }
          break;
        case 1:
          if (i+val == t->__rows) { --val; go_on = false; }
          break;
        case 2:
          if (j-val == -1) { --val; go_on = false; }
          break;
        case 3:
          if (j+val == t->__cols) { --val; go_on = false; }
          break;
        default:
          break;
      }

      if (!go_on) break;

      int i0 = __idx1(lr, i, j);
      int i1 = __idx1(lr, i+dirs[x][0]*val, j+dirs[x][1]*val);
      int cd1 = __color_diff(
        &(t->__img_data[i0]), &(t->__img_data[i1])
      );
      go_on = val < CROSS_L1;

      if (val > CROSS_L2 && val < CROSS_L1) go_on &= cd1 < CROSS_TAU2;
      else go_on &= cd1 < CROSS_TAU1;
      if (val >= 1) {
        int i0 = __idx1(lr, (i+dirs[x][0]*(val-1)), (j+dirs[x][1]*(val-1)));
        int i1 = __idx1(lr, i+dirs[x][0]*val, j+dirs[x][1]*val);
        int cd2 = __color_diff(&(t->__img_data[i0]), &(t->__img_data[i1]));

        go_on &= cd2 < CROSS_TAU1;
      }

      if (!go_on) --val;
    }

```

```
        *vals[x] = (x % 2) ? val : -val;
      }
    }
  }
}

#endif
```

A.1.6 03_support_size_computation.hpp

```
#ifndef __03_SUPPSIZE_COMP_HPP
#define __03_SUPPSIZE_COMP_HPP

#include "../common.hpp"

void __HV_supp_compute(bigtask_t* t, int worker_id, int lr) {
  if (worker_id != 0) return; // This is better computed sequentially

  for (int i = 0; i < t->__rows; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      cross_t* c = &t->__crosses[__idx2(lr, i, j)];
      t->__supp_sizes[__idx4(lr, 0, i, j)] = (c->xp - c->xm) + 1;
    }
  }

  for (int j = 0; j < t->__cols; ++j) {
    t->__supp_sizes[__idx4(lr, 1, 0, j)] =
      t->__supp_sizes[__idx4(lr, 0, 0, j)];
  }
  for (int i = 1; i < t->__rows; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      t->__supp_sizes[__idx4(lr, 1, i, j)] =
        t->__supp_sizes[__idx4(lr, 1, i-1, j)]
        + t->__supp_sizes[__idx4(lr, 0, i, j)];
    }
  }

  for (int i = 0; i < t->__rows; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      cross_t* c = &t->__crosses[__idx2(lr, i, j)];
      if (i+c->ym == 0)
        t->__supp_size_HV[__idx2(lr, i, j)] =
          t->__supp_sizes[__idx4(lr, 1, i+c->yp, j)];
      else
        t->__supp_size_HV[__idx2(lr, i, j)] =
          t->__supp_sizes[__idx4(lr, 1, i+c->yp, j)]
          - t->__supp_sizes[__idx4(lr, 1, i+c->ym-1, j)];
    }
  }
}

void __VH_supp_compute(bigtask_t* t, int worker_id, int lr) {
  if (worker_id != 0) return; // This is better computed sequentially

  for (int i = 0; i < t->__rows; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      cross_t* c = &t->__crosses[__idx2(lr, i, j)];
      t->__supp_sizes[__idx4(lr, 0, i, j)] = (c->yp - c->ym) + 1;
    }
  }

  for (int i = 0; i < t->__rows; ++i) {
    t->__supp_sizes[__idx4(lr, 1, i, 0)] =
      t->__supp_sizes[__idx4(lr, 0, i, 0)];
    for (int j = 1; j < t->__cols; ++j) {
      t->__supp_sizes[__idx4(lr, 1, i, j)] =
        t->__supp_sizes[__idx4(lr, 1, i, j-1)]
        + t->__supp_sizes[__idx4(lr, 0, i, j)];
    }
  }

  for (int i = 0; i < t->__rows; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      cross_t* c = &t->__crosses[__idx2(lr, i, j)];
      if (j+c->xm == 0)
        t->__supp_size_VH[__idx2(lr, i, j)] =
          t->__supp_sizes[__idx4(lr, 1, i, j+c->xp)];
      else
        t->__supp_size_VH[__idx2(lr, i, j)] =
          t->__supp_sizes[__idx4(lr, 1, i, j+c->xp)]
          - t->__supp_sizes[__idx4(lr, 1, i, j+c->xm-1)];
    }
  }

}

#endif
```

A.1.7 04_aggregation.hpp

```
#ifndef __04_AGGREGATION_HPP
#define __04_AGGREGATION_HPP

#include "../common.hpp"

void __horizontal(bigtask_t* t, int worker_id, int q) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int lr = 0; lr < 2; ++lr) {
    // Computing horizontal integral
    for (int i = ifrom; i < ito; ++i) {
      for (int j = 1; j < t->__cols; ++j) {
        for (int d = 0; d <= DMAX; ++d) {
          t->__aggregation[__idx3(lr, q, i, j, d)] +=
            t->__aggregation[__idx3(lr, q, i, j-1, d)];
        }
      }
    }


    // Aggregating on horizontal arms
    for (int i = ifrom; i < ito; ++i) {
      for (int j = 0; j < t->__cols; ++j) {
        for (int d = 0; d <= DMAX; ++d) {
          cross_t* c = &t->__crosses[__idx2(lr, i, j)];
          if (j+c->xm == 0)
            t->__aggregation[__idx3(lr, 1-q, i, j, d)] =
              t->__aggregation[__idx3(lr, q, i, j+c->xp, d)];
          else
            t->__aggregation[__idx3(lr, 1-q, i, j, d)] =
              t->__aggregation[__idx3(lr, q, i, j+c->xp, d)]
              - t->__aggregation[__idx3(lr, q, i, (j+c->xm)-1, d)];
        }
      }
    }
  }
}

void __vertical(bigtask_t* t, int worker_id, int q) {
  int jfrom = (worker_id * t->__cols) / __nworkers;
  int jto = ((worker_id + 1) * t->__cols) / __nworkers;
  for (int lr = 0; lr < 2; ++lr) {
    // Computing vertical integral
    for (int i = 1; i < t->__rows; ++i) {
      for (int j = jfrom; j < jto; ++j) {
        for (int d = 0; d <= DMAX; ++d) {
          t->__aggregation[__idx3(lr, q, i, j, d)] +=
            t->__aggregation[__idx3(lr, q, i-1, j, d)];
        }
      }
    }

    // Aggregating on vertical arms
    for (int i = 0; i < t->__rows; ++i) {
      for (int j = jfrom; j < jto; ++j) {
        for (int d = 0; d <= DMAX; ++d) {
          cross_t* c = &t->__crosses[__idx2(lr, i, j)];
          if (i+c->ym == 0)
            t->__aggregation[__idx3(lr, 1-q, i, j, d)] =
              t->__aggregation[__idx3(lr, q, i+c->yp, j, d)];
          else
            t->__aggregation[__idx3(lr, 1-q, i, j, d)] =
              t->__aggregation[__idx3(lr, q, i+c->yp, j, d)]
              - t->__aggregation[__idx3(lr, q, i+c->ym-1, j, d)];
        }
      }
    }
  }
}

void __HV_supp_normalize(bigtask_t* t, int worker_id, int q) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int lr = 0; lr < 2; ++lr) {
    for (int i = ifrom; i < ito; ++i) {
      for (int j = 0; j < t->__cols; ++j) {
        for (int d = 0; d <= DMAX; ++d) {
          t->__aggregation[__idx3(lr, q, i, j, d)] /=
            t->__supp_size_HV[__idx2(lr, i, j)];
        }
      }
    }
  }
}

void __VH_supp_normalize(bigtask_t* t, int worker_id, int q) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int lr = 0; lr < 2; ++lr) {
    for (int i = ifrom; i < ito; ++i) {
      for (int j = 0; j < t->__cols; ++j) {
        for (int d = 0; d <= DMAX; ++d) {
          t->__aggregation[__idx3(lr, q, i, j, d)] /=
            t->__supp_size_VH[__idx2(lr, i, j)];
        }
      }
    }
  }
}

void __aggregation_finalization(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int lr = 0; lr < 2; ++lr) {
    for (int x = 0; x < 4; ++x) {
      for (int i = ifrom; i < ito; ++i) {
        for (int j = 0; j < t->__cols; ++j) {
          for (int d = 0; d <= DMAX; ++d) {
            t->__scanline_opt[__idx5(lr, x, i, j, d)] =
              t->__aggregation[__idx3(lr, 0, i, j, d)];
          }
        }
      }
    }
  }
}

#endif
```

A.1.8 05_scanline_optimization.hpp

```
#ifndef __05_SCANLINE_HPP
#define __05_SCANLINE_HPP

#include "../common.hpp"

void __scanline_optimization(bigtask_t* t, int worker_id, int lr) {
  float p1, p2;
  float d1, d2;

  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
```

```

// Moving right
for (int i = ifrom; i < ito; ++i) {
  for (int j = 1; j < t->__cols; ++j) {
    // Computing min_d{C(p-r, d)}
    float mink = t->__scanline_opt[__idx5(lr, 0, i, j-1, 0)];
    for (int d = 1; d <= DMAX; ++d) {
      if (mink > t->__scanline_opt[__idx5(lr, 0, i, j-1, d)])
        mink = t->__scanline_opt[__idx5(lr, 0, i, j-1, d)];
    }

    d1 = d2 = OPT_TAUSO + 1;
    if (lr == 0)
      d1 = __color_diff(
        &(t->__img_data[__idx1(0, i, j)]),
        &(t->__img_data[__idx1(0, i, j-1)])
      );
    else if (lr == 1)
      d2 = __color_diff(
        &(t->__img_data[__idx1(1, i, j)]),
        &(t->__img_data[__idx1(1, i, j-1)])
      );

    for (int d = 0; d <= DMAX; d++) {
      // Computing p1, p2
      if (lr == 1 && j < t->__cols - d)
        d1 = __color_diff(
          &(t->__img_data[__idx1(0, i, j+d)]),
          &(t->__img_data[__idx1(0, i, (j+d)-1)])
        );
      else if (lr == 0 && j > d)
        d2 = __color_diff(
          &(t->__img_data[__idx1(1, i, j-d)]),
          &(t->__img_data[__idx1(1, i, (j-d)-1)])
        );

      if (d1 <= OPT_TAUSO && d2 <= OPT_TAUSO) {
        p1 = OPT_PI1; p2 = OPT_PI2;
      }
      else if ((d1 <= OPT_TAUSO && d2 > OPT_TAUSO)
        || (d1 > OPT_TAUSO && d2 <= OPT_TAUSO)) {
        p1 = OPT_PI1/4; p2 = OPT_PI2/4;
      }
      else {
        p1 = OPT_PI1/10; p2 = OPT_PI2/10;
      }
      // END

      float toadd = 0;
      if (d == 0) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d)], std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d+1)] + p1,
          mink + p2));
      } else if (d == DMAX) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d)], std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d-1)] + p1,
          mink + p2));
      } else {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d)], std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d+1)] + p1, std::min(
          t->__scanline_opt[__idx5(lr, 0, i, j-1, d-1)] + p1,
          mink + p2)));
      }

      t->__scanline_opt[__idx5(lr, 0, i, j, d)] += toadd - mink;
      t->__c2_values[__idx6(lr, i, j, d)] =
        t->__scanline_opt[__idx5(lr, 0, i, j, d)] / 4;
    }
  }
}

```

```
// Moving left
for (int i = ifrom; i < ito; ++i) {
  for (int j = t->__cols - 2; j >= 0; --j) {
    // Computing min_d{C(p-r, d)}
    float mink = t->__scanline_opt[__idx5(lr, 1, i, j+1, 0)];
    for (int d = 1; d <= DMAX; ++d) {
      if (mink > t->__scanline_opt[__idx5(lr, 1, i, j+1, d)])
        mink = t->__scanline_opt[__idx5(lr, 1, i, j+1, d)];
    }

    d1 = d2 = OPT_TAUSO + 1;
    if (lr == 0)
      d1 = __color_diff(
        &(t->__img_data[__idx1(0, i, j)]),
        &(t->__img_data[__idx1(0, i, j+1)])
      );
    else if (lr == 1)
      d2 = __color_diff(
        &(t->__img_data[__idx1(1, i, j)]),
        &(t->__img_data[__idx1(1, i, j+1)])
      );

    for (int d = 0; d <= DMAX; ++d) {
      if (lr == 1 && j < t->__cols - d - 1)
        d1 = __color_diff(
          &(t->__img_data[__idx1(0, i, j+d)]),
          &(t->__img_data[__idx1(0, i, (j+d)+1)])
        );
      else if (lr == 0 && j > d)
        d2 = __color_diff(
          &(t->__img_data[__idx1(1, i, j-d)]),
          &(t->__img_data[__idx1(1, i, (j-d)+1)])
        );

      // Computing p1, p2
      if (d1 <= OPT_TAUSO && d2 <= OPT_TAUSO) {
        p1 = OPT_PI1; p2 = OPT_PI2;
      }
      else if ((d1 <= OPT_TAUSO && d2 > OPT_TAUSO)
        || (d1 > OPT_TAUSO && d2 <= OPT_TAUSO)) {
        p1 = OPT_PI1/4; p2 = OPT_PI2/4;
      }
      else {
        p1 = OPT_PI1/10; p2 = OPT_PI2/10;
      }
      // END

      float toadd = 0;
      if (d == 0) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d)], std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d+1)] + p1,
          mink + p2));
      } else if (d == DMAX) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d)], std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d-1)] + p1,
          mink + p2));
      } else {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d)], std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d+1)] + p1, std::min(
          t->__scanline_opt[__idx5(lr, 1, i, j+1, d-1)] + p1,
          mink + p2)));
      }

      t->__scanline_opt[__idx5(lr, 1, i, j, d)] += toadd - mink;
      t->__c2_values[__idx6(lr, i, j, d)] +=
        t->__scanline_opt[__idx5(lr, 1, i, j, d)] / 4;
    }
  }
}

int jfrom = (worker_id * t->__cols) / __nworkers;
```

```
int jto = ((worker_id + 1) * t->__cols) / __nworkers;

// Moving down
for (int i = 1; i < t->__rows; ++i) {
  for (int j = jfrom; j < jto; ++j) {
    float mink = t->__scanline_opt[__idx5(lr, 2, i-1, j, 0)];
    for (int d = 1; d <= DMAX; ++d) {
      if (mink > t->__scanline_opt[__idx5(lr, 2, i-1, j, d)])
        mink = t->__scanline_opt[__idx5(lr, 2, i-1, j, d)];
    }

    d1 = d2 = OPT_TAUSO + 1;
    if (lr == 0)
      d1 = __color_diff(
        &(t->__img_data[__idx1(0, i, j)]),
        &(t->__img_data[__idx1(0, i-1, j)])
      );
    else if (lr == 1)
      d2 = __color_diff(
        &(t->__img_data[__idx1(1, i, j)]),
        &(t->__img_data[__idx1(1, i-1, j)])
      );

    for (int d = 0; d <= DMAX; ++d) {
      if (lr == 1 && j < t->__cols - d)
        d1 = __color_diff(
          &(t->__img_data[__idx1(0, i, j+d)]),
          &(t->__img_data[__idx1(0, i-1, j+d)])
        );
      else if (lr == 0 && j > d)
        d2 = __color_diff(
          &(t->__img_data[__idx1(1, i, j-d)]),
          &(t->__img_data[__idx1(1, i-1, j-d)])
        );

      // Computing p1, p2
      if (d1 <= OPT_TAUSO && d2 <= OPT_TAUSO) {
        p1 = OPT_PI1; p2 = OPT_PI2;
      }
      else if ((d1 <= OPT_TAUSO && d2 > OPT_TAUSO)
        || (d1 > OPT_TAUSO && d2 <= OPT_TAUSO)) {
        p1 = OPT_PI1/4; p2 = OPT_PI2/4;
      }
      else {
        p1 = OPT_PI1/10; p2 = OPT_PI2/10;
      }
      // END

      float toadd = 0;
      if (d == 0) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d)], std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d+1)] + p1,
          mink + p2));

      } else if (d == DMAX) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d)], std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d-1)] + p1,
          mink + p2));
      } else {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d)], std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d+1)] + p1, std::min(
          t->__scanline_opt[__idx5(lr, 2, i-1, j, d-1)] + p1,
          mink + p2)));
      }

      t->__scanline_opt[__idx5(lr, 2, i, j, d)] += toadd - mink;
      t->__c2_values[__idx6(lr, i, j, d)] +=
        t->__scanline_opt[__idx5(lr, 2, i, j, d)] / 4;
    }
  }
}
```

```

// Moving up
for (int i = t->__rows - 2; i >= 0; --i) {
  for (int j = jfrom; j < jto; ++j) {
    float mink = t->__scanline_opt[__idx5(lr, 3, i+1, j, 0)];
    for (int d = 1; d <= DMAX; ++d) {
      if (mink > t->__scanline_opt[__idx5(lr, 3, i+1, j, d)])
        mink = t->__scanline_opt[__idx5(lr, 3, i+1, j, d)];
    }

    d1 = d2 = OPT_TAUSO + 1;
    if (lr == 0)
      d1 = __color_diff(
        &(t->__img_data[__idx1(0, i, j)]),
        &(t->__img_data[__idx1(0, i+1, j)])
      );
    else if (lr == 1)
      d2 = __color_diff(
        &(t->__img_data[__idx1(1, i, j)]),
        &(t->__img_data[__idx1(1, i+1, j)])
      );

    for (int d = 0; d <= DMAX; ++d) {

      if (lr == 1 && j < t->__cols - d)
        d1 = __color_diff(
          &(t->__img_data[__idx1(0, i, j+d)]),
          &(t->__img_data[__idx1(0, i+1, j+d)])
        );
      else if (lr == 0 && j >= d)
        d2 = __color_diff(
          &(t->__img_data[__idx1(1, i, j-d)]),
          &(t->__img_data[__idx1(1, i+1, j-d)])
        );

      // Computing p1, p2
      if (d1 <= OPT_TAUSO && d2 <= OPT_TAUSO) {
        p1 = OPT_PI1; p2 = OPT_PI2;
      }
      else if ((d1 <= OPT_TAUSO && d2 > OPT_TAUSO)
        || (d1 > OPT_TAUSO && d2 <= OPT_TAUSO)) {
        p1 = OPT_PI1/4; p2 = OPT_PI2/4;
      }
      else {
        p1 = OPT_PI1/10; p2 = OPT_PI2/10;
      }
      // END

      float toadd = 0;
      if (d == 0) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d)], std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d+1)] + p1,
          mink + p2));

      } else if (d == DMAX) {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d)], std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d-1)] + p1,
          mink + p2));
      } else {
        toadd = std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d)], std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d+1)] + p1, std::min(
          t->__scanline_opt[__idx5(lr, 3, i+1, j, d-1)] + p1,
          mink + p2)));
      }

      t->__scanline_opt[__idx5(lr, 3, i, j, d)] += toadd - mink;
      t->__c2_values[__idx6(lr, i, j, d)] +=
        t->__scanline_opt[__idx5(lr, 3, i, j, d)] / 4;
    }
  }
}
```

```
}

#endif
```

A.1.9 06_disparity_estimation.hpp

```
#ifndef __06_DISPEST_HPP
#define __06_DISPEST_HPP

#include "../common.hpp"

void __disparity_estimation(bigtask_t* t, int worker_id, int lr) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      t->__disparity_estimate[__idx2(lr, i, j)] = 0;
      for (int d = 1; d <= DMAX; ++d) {
        int i0 = __idx6(lr, i, j, d);
        int i1 = __idx6(lr, i, j, t->__disparity_estimate[__idx2(lr, i, j)]);
        if (t->__c2_values[i0] < t->__c2_values[i1])
          t->__disparity_estimate[__idx2(lr, i, j)] = d;
      }
    }
  }
}

#endif
```

A.1.10 07_outlier_detection.hpp

```
#ifndef __07_OUTLDET_HPP
#define __07_OUTLDET_HPP

#include "../common.hpp"

void __outlier_detection(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      int d1 = t->__disparity_estimate[__idx2(0, i, j)];
      if (j < d1) t->__outliers[__idx7(i, j)] = OUT_OCCL;
      else {
        int d2 = t->__disparity_estimate[__idx2(1, i, j-d1)];
        if (d1 != d2) {
          t->__outliers[__idx7(i, j)] = OUT_OCCL;
          for (int d = d1; d <= std::min(DMAX, j); ++d) {
            if (t->__disparity_estimate[__idx2(1, i, j-d)] == d) {
              t->__outliers[__idx7(i, j)] = OUT_MISM;
            }
          }
        }
      }
    }
  }
}

#endif
```

A.1.11 08_iterative_voting.hpp

```
#ifndef __08_ITERVOTE_HPP
#define __08_ITERVOTE_HPP

#include "../common.hpp"

void __histogram_init(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      for (int d = 0; d <= DMAX; ++d)
        t->__voting_histograms[__idx6(0, i, j, d)] = 0;
```

```c
        if (t->__outliers[__idx7(i, j)] == OUT_NONE) {
          int idx = __idx6(0, i, j, t->__disparity_estimate[__idx2(0, i, j)]);
          t->__voting_histograms[idx] = 1;
        }
      }
    }
}

void __histogram_computation_H(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  // Computing horizontal integral
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 1; j < t->__cols; ++j) {
      for (int d = 0; d <= DMAX; ++d) {
        t->__voting_histograms[__idx6(0, i, j, d)] +=
          t->__voting_histograms[__idx6(0, i, j-1, d)];
      }
    }
  }

  // Aggregating left-right
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      cross_t* c = &t->__crosses[__idx2(0, i, j)];
      for (int d = 0; d <= DMAX; ++d) {
        if (j+c->xm == 0)
          t->__voting_histograms[__idx6(1, i, j, d)] =
            t->__voting_histograms[__idx6(0, i, j+c->xp, d)];
        else
          t->__voting_histograms[__idx6(1, i, j, d)] =
            t->__voting_histograms[__idx6(0, i, j+c->xp, d)]
            - t->__voting_histograms[__idx6(0, i, (j+c->xm)-1, d)];
      }
    }
  }
}
```

```c

void __histogram_computation_V(bigtask_t* t, int worker_id) {
  int jfrom = (worker_id * t->__cols) / __nworkers;
  int jto = ((worker_id + 1) * t->__cols) / __nworkers;
  // Computing vertical integral
  for (int i = 1; i < t->__rows; ++i) {
    for (int j = jfrom; j < jto; ++j) {
      for (int d = 0; d <= DMAX; ++d) {
        t->__voting_histograms[__idx6(1, i, j, d)] +=
          t->__voting_histograms[__idx6(1, i-1, j, d)];
      }
    }
  }

  // Aggregating up-down
  for (int i = 0; i < t->__rows; ++i) {
    for (int j = jfrom; j < jto; ++j) {
      cross_t* c = &t->__crosses[__idx2(0, i, j)];
      for (int d = 0; d <= DMAX; ++d) {
        if (i+c->ym == 0)
          t->__voting_histograms[__idx6(0, i, j, d)] =
            t->__voting_histograms[__idx6(1, i+c->yp, j, d)];
        else
          t->__voting_histograms[__idx6(0, i, j, d)] =
            t->__voting_histograms[__idx6(1, i+c->yp, j, d)]
            - t->__voting_histograms[__idx6(1, (i+c->ym)-1, j, d)];
      }
    }
  }
}

void __iterative_voting(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      if (t->__outliers[__idx7(i, j)] != OUT_NONE) {
        int sp = t->__voting_histograms[__idx6(0, i, j, 0)], dmax = 0;
        for (int d = 1; d <= DMAX; ++d) {
          sp += t->__voting_histograms[__idx6(0, i, j, d)];
          if (t->__voting_histograms[__idx6(0, i, j, d)] >
            t->__voting_histograms[__idx6(0, i, j, dmax)]) dmax = d;
        }
        float th = (float)t->__voting_histograms[__idx6(0, i, j, dmax)]/(float)sp;
        if (th > ITER_TAUH && sp > ITER_TAUS) {
          t->__voting_disparities[__idx7(i, j)] = dmax;
        }
        else t->__voting_disparities[__idx7(i, j)] = -1;
      }
    }
  }
}

void __vote_consolidation(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      if (t->__outliers[__idx7(i, j)] != OUT_NONE
        && t->__voting_disparities[__idx7(i, j)] != -1) {
        t->__disparity_estimate[__idx2(0, i, j)] =
          t->__voting_disparities[__idx7(i, j)];
        t->__outliers[__idx7(i, j)] = OUT_NONE;
      }
    }
  }
}

#endif
```

A.1.12 09_proper_interpolation.hpp

```
#ifndef __09_INTERPOLATION_HPP
#define __09_INTERPOLATION_HPP
```

```

#include "../common.hpp"

void __proper_interpolation(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      if (t->__outliers[__idx7(i, j)] != OUT_NONE) {
        t->__interpolation_disparities[__idx7(i, j)] = -1;
        int __reliables[16][2];

        bool found = false;
        for (int x = 0; x < 16; ++x) __reliables[x][0] = __reliables[x][1] = -1;

        int cnt = 1;
        while (!found) {
          for (int x = 0; x < 16; ++x) {
            int* rlb = &__reliables[x][0];
            int ii = cnt*(t->__interpolation_dirs[x][0]);
            int jj = cnt*(t->__interpolation_dirs[x][1]);
            if (i + ii >= 0 && i + ii < t->__rows
              && j + jj >= 0 && j + jj < t->__cols
              && t->__outliers[__idx7(i+ii, j+jj)] == OUT_NONE) {
              rlb[0] = i+ii;
              rlb[1] = j+jj;
              found = true;
            }
          }
          cnt++;
        }

        if (t->__outliers[__idx7(i, j)] == OUT_MISM) {
          int min = -1, cdiff = -1;
          for (int x = 0; x < 16; ++x) {
            int* rlb = &__reliables[x][0];
            if (rlb[0] != -1) {
              int cdtemp = __color_diff(&(t->__img_data[__idx1(0, i, j)]),
                &(t->__img_data[__idx1(0, rlb[0], rlb[1])]));
              int de = t->__disparity_estimate[__idx2(0, rlb[0], rlb[1])];
              if (min == -1 || cdtemp < cdiff) {
                min = de;
                cdiff = cdtemp;
              }
            }
          }
          t->__interpolation_disparities[__idx7(i, j)] = min;
        }

        else if (t->__outliers[__idx7(i, j)] == OUT_OCCL) {
          int min = -1;
          for (int x = 0; x < 16; ++x) {
            int* rlb = &__reliables[x][0];
            if (rlb[0] != -1) {
              int de = t->__disparity_estimate[__idx2(0, rlb[0], rlb[1])];
              if (min == -1 || de < min) min = de;
            }
          }
          t->__interpolation_disparities[__idx7(i, j)] = min;
        }
      }
    }
  }
}

void __interpolation_consolidation(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      if (t->__outliers[__idx7(i, j)] != OUT_NONE
        && t->__interpolation_disparities[__idx7(i, j)] != -1) {
        t->__disparity_estimate[__idx2(0, i, j)] =
          t->__interpolation_disparities[__idx7(i, j)];
      }
    }
  }
}

#endif
```

A.1.13 10_edge_detection.hpp

```
#ifndef __10_EDGEDETECT_HPP
#define __10_EDGEDETECT_HPP

#include "../common.hpp"

static int __laplace[3][3] = {
  {-1, -1, -1},
  {-1, 8, -1},
  {-1, -1, -1}
};

void __edge_detection(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    int iimin = std::max(-i, -1);
    int iimax = std::min((t->__rows-i)-1, 1);
    for (int j = 0; j < t->__cols; ++j) {
      int lapl = 0;
      int jjmin = std::max(-j, -1);
      int jjmax = std::min((t->__cols-j)-1, 1);
      for (int ii = iimin; ii <= iimax; ii++) {
        for (int jj = jjmin; jj <= jjmax; jj++) {
          lapl +=
            t->__disparity_estimate[__idx2(0, i+ii, j+jj)]
            * __laplace[ii+1][jj+1];
        }
      }
      if (lapl == 0) t->__disparity_edges[__idx7(i, j)] = 1;
      else t->__disparity_edges[__idx7(i, j)] = 0;
    }
  }
}

#endif
```

A.1.14 11_discontinuity_adjustment.hpp

```
#ifndef __11_DISCONTADJ_HPP
#define __11_DISCONTADJ_HPP

#include "../common.hpp"

void __discontinuity_adjustment(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      if (t->__disparity_edges[__idx7(i, j)]) {
        int j1 = j, j2 = j;
        while (j1 >= 0 && t->__disparity_edges[__idx7(i, j1)]) j1--;
        while (j2 < t->__cols && t->__disparity_edges[__idx7(i, j2)]) j2++;
        int d = t->__disparity_estimate[__idx2(0, i, j)];
        double c2 = t->__c2_values[__idx6(0, d, i, j)];

        if (j1 >= 0) {
          int d1 = t->__disparity_estimate[__idx2(0, i, j1)];
          double c2a = t->__c2_values[__idx6(0, d1, i, j)];
          if (c2a < c2) t->__disparity_estimate[__idx2(0, i, j)] = d1;
        }

        if (j2 < t->__cols) {
          int d2 = t->__disparity_estimate[__idx2(0, i, j2)];
          double c2b = t->__c2_values[__idx6(0, d2, i, j)];
          if (c2b < c2) t->__disparity_estimate[__idx2(0, i, j)] = d2;
        }
      }
    }
  }
}

#endif
```

A.1.15 12_sub_pixel_enhancement.hpp

```
#ifndef __12_SUBPIX_HPP
#define __12_SUBPIX_HPP

#include "../common.hpp"

void __sub_pixel_enhancement(bigtask_t* t, int worker_id) {
  int ifrom = (worker_id * t->__rows) / __nworkers;
  int ito = ((worker_id + 1) * t->__rows) / __nworkers;
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 0; j < t->__cols; ++j) {
      int d = t->__disparity_estimate[__idx2(0, i, j)];
      float num, den;
      if (d == 0 || d == DMAX) {
        num = 0; den = 1;
      }
      else {
        float c0 = t->__c2_values[__idx6(0, i, j, d-1)];
        float c1 = t->__c2_values[__idx6(0, i, j, d)];
        float c2 = t->__c2_values[__idx6(0, i, j, d+1)];
        num = c2 - c0;
        den = 2*(c0+c2-2*c1);
      }

      t->__disparity_subpix[__idx7(i, j)] =
        std::max((float)0, std::min((float)d - num/den, (float)DMAX));
    }
  }
}

#endif
```

A.1.16 13_median_filter.hpp

```
#ifndef __13_MEDIAN_FILTER
#define __13_MEDIAN_FILTER

#include "../common.hpp"
#include <cstring>

void __median_filter(bigtask_t* t, int worker_id) {
  uint8_t __median_bins[DMAX+1];
  int ifrom = 1 + ((worker_id * (t->__rows - 2)) / __nworkers);
  int ito = 1 + (((worker_id+1) * (t->__rows - 2)) / __nworkers);
  for (int i = ifrom; i < ito; ++i) {
    for (int j = 1; j < t->__cols - 1; ++j) {
      std::memset(__median_bins, 0, (DMAX+1)*sizeof(uint8_t));
      __median_bins[(uint)t->__disparity_subpix[__idx7(i-1, j-1)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i-1, j)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i-1, j+1)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i, j-1)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i, j)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i, j+1)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i+1, j-1)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i+1, j)]]++;
      __median_bins[(uint)t->__disparity_subpix[__idx7(i+1, j+1)]]++;

      int tot = 0, d = 0;
      while (tot < 5) tot += __median_bins[d++];

      t->__disparity_final[__idx7(i, j)] = d-1;
    }
  }
}

```

```
#endif
```

A.2 Skeleton code

A.2.1 emitter.hpp

```cpp
#ifndef __FARM_EMIT_HPP
#define __FARM_EMIT_HPP

#include <ff/farm.hpp>
#include <chrono>
#include <cmath>
#include <string>
#include <ff/mapping_utils.hpp>

#include "../common.hpp"

#include "task.hpp"
#include "stages.hpp"

using namespace ff;

class MyLoadBalancer: public ff::ff_loadbalancer {
private:
    inline size_t selectworker() { return victim; }
    size_t victim;
public:
    MyLoadBalancer(int max_num_workers):
        ff::ff_loadbalancer(max_num_workers) {}

    void set_victim(size_t v) { victim = v; }
};

class SVEmitter : public ff_node {
private:
    bool first = true;

    MyLoadBalancer* lb;
    int last_stream;

public:
    SVEmitter(MyLoadBalancer* lb): lb(lb) {
        last_stream = 0;
    }

    void* svc(void* task) {
        if (task == NULL) {
            int tcmax = std::min(__ndatapar, __streamsz);
            for (int tcount = 0; tcount < tcmax; ++tcount) {
                int wid = tcount * __nworkers;
                for (int i = 0; i < __nworkers; ++i) {
                    lb->set_victim(wid++);
                    ff_send_out((void*)make_task(i, 0, last_stream, tcount));
                }
                last_stream++;
            }
        }

        else {
            uint8_t worker_id;
            uint8_t stage_id;
            uint16_t stream_id;
            uint8_t datapar_id;

            read_task((uint64_t)task, worker_id, stage_id, stream_id,
                datapar_id);

            if (stages[stage_id+1].name == nullptr) {
                int newtask = last_stream + 1;
                last_stream++;
                if (newtask < __streamsz) {
                    int wid = datapar_id * __nworkers;
                    for (int i = 0; i < __nworkers; ++i) {
                        lb->set_victim(wid++);
                        ff_send_out(
                            (void*)make_task(i, 0, newtask, datapar_id)
                        );
                    }
                }

                if (stream_id == __streamsz-1) return NULL;

                return GO_ON;
            } else {
                int wid = datapar_id * __nworkers;
                for (int i = 0; i < __nworkers; ++i) {
                    lb->set_victim(wid++);
                    ff_send_out(
                        (void*)make_task(i, stage_id+1, stream_id, datapar_id)
                    );
                }
            }
        }

        return GO_ON;
    }
};

#endif
```

A.2.2 worker.hpp

```
#ifndef __FARM_WORK_HPP
#define __FRAM_WORK_HPP

#include "task.hpp"
#include "stages.hpp"

#include <ff/node.hpp>

```

```cpp
class SVWorker : public ff_node {
public:
    void* svc(void* task) {
        uint8_t worker_id;
        uint8_t stage_id;
        uint16_t stream_id;
        uint8_t datapar_id;

        read_task((uint64_t)task, worker_id, stage_id, stream_id, datapar_id);

        const stage_t* s = &(stages[stage_id]);
        if (s->tag == wtask_tag::TASK1)
            (s->callback.callback1)(&__stream[datapar_id], worker_id);
        else if (s->tag == wtask_tag::TASK2)
            (s->callback.callback2)(&__stream[datapar_id], worker_id, s->lr);
        else if (s->tag == wtask_tag::TASK3)
            (s->callback.callback3)(&__stream[datapar_id], worker_id, s->lr, s->q);
        ff_send_out(task);

        return GO_ON;
    }
};

#endif
```

A.2.3 collector.hpp

```cpp
#ifndef __FARM_COLL_HPP
#define __FRAM_COLL_HPP

#include "task.hpp"
#include <ff/node.hpp>

using namespace ff;

class SVCollector : public ff_node {
public:
    void* svc(void* task) {
        uint8_t worker_id;
        uint8_t stage_id;
        uint16_t stream_id;
        uint8_t datapar_id;

        read_task((uint64_t)task, worker_id, stage_id, stream_id, datapar_id);

        __counters[datapar_id]--;

        if (__counters[datapar_id] == 0) {
            __counters[datapar_id] = __nworkers;
            ff_send_out(task);
        }

        return GO_ON;
    }
};

#endif
```

A.2.4 task.hpp

```
#ifndef __FARM_TASK_HPP
#define __FARM_TASK_HPP

#include "../common.hpp"
#define PI 3.14159265358979323846264338327

typedef struct bigtask_t {
    uint8_t*     __img_data;
    uint8_t*     __img_grey;
    uint64_t*    __census_data;
    cross_t*     __crosses;
    float*       __aggregation;
    int*         __supp_sizes;
    int*         __supp_size_HV;
    int*        __supp_size_VH;
    float*      __scanline_opt;
    float*      __c2_values;
    int*        __disparity_estimate;
    float*      __disparity_subpix;
    float*      __disparity_final;
    outlier_t*  __outliers;
    int*        __voting_histograms;
    int*        __voting_disparities;
    float       __interpolation_dirs[16][2];
    int*        __interpolation_disparities;
    float*      __disparity_edges;
    const int   __rows = ROWS, __cols = COLS;

    bigtask_t() {
        __img_data = fdata;
        __img_grey = fdataG;
        __census_data = new uint64_t[2*ROWS*COLS];
        __crosses = new cross_t[2*ROWS*COLS];
        __aggregation = new float[2*2*ROWS*COLS*(DMAX+1)];
        __supp_sizes = new int[2*2*ROWS*COLS];
        __supp_size_HV = new int[2*ROWS*COLS];
        __supp_size_VH = new int[2*ROWS*COLS];
        __scanline_opt = new float[2*4*ROWS*COLS*(DMAX+1)];
        __c2_values = new float[2*ROWS*COLS*(DMAX+1)];
        __disparity_estimate = new int[2*ROWS*COLS];
        __disparity_subpix = new float[ROWS*COLS];
        __disparity_final = new float[ROWS*COLS];
        __outliers = new outlier_t[ROWS*COLS];
        __voting_histograms = new int[2*ROWS*COLS*(DMAX+1)];
        __voting_disparities = new int[ROWS*COLS];
        __interpolation_disparities = new int[ROWS*COLS];
        __disparity_edges = new float[ROWS*COLS];

        for (int x = 0; x < 16; ++x) {
            __interpolation_dirs[x][0] = std::sin((float)x * PI/8); // y -> rows
            __interpolation_dirs[x][1] = std::cos((float)x * PI/8); // x -> cols
        }
    }

    ~bigtask_t() {
        delete[] __census_data;
        delete[] __crosses;
        delete[] __aggregation;
        delete[] __supp_sizes;
        delete[] __supp_size_HV;
        delete[] __supp_size_VH;
        delete[] __scanline_opt;
        delete[] __c2_values;
        delete[] __disparity_estimate;
        delete[] __disparity_subpix;
        delete[] __disparity_final;
        delete[] __outliers;
        delete[] __voting_histograms;
        delete[] __voting_disparities;
        delete[] __interpolation_disparities;
        delete[] __disparity_edges;
    }

} bigtask_t;


typedef void (*callback1_t)(bigtask_t*, int);
typedef void (*callback2_t)(bigtask_t*, int, int);
typedef void (*callback3_t)(bigtask_t*, int, int, int);

typedef enum {TASK1, TASK2, TASK3} wtask_tag;
typedef union {
    callback1_t callback1;
    callback2_t callback2;
    callback3_t callback3;
} callback_t;

callback_t __callback1(callback1_t c) {
    callback_t callback;
    callback.callback1 = c;
    return callback;
}

callback_t __callback2(callback2_t c) {
    callback_t callback;
    callback.callback2 = c;
    return callback;
}

callback_t __callback3(callback3_t c) {
    callback_t callback;
    callback.callback3 = c;
    return callback;
}

typedef struct stage_t {
    wtask_tag tag;
    const char* name;
    callback_t callback;
    int count;
    int lr;
    int q;

    stage_t (wtask_tag tag, const char* name, callback_t callback, int lr, int q):
        tag(tag), name(name), callback(callback), lr(lr), q(q) {};
} stage_t;


void inline read_task(uint64_t task, uint8_t& worker_id, uint8_t& stage_id,
    uint16_t& stream_id, uint8_t& datapar_id) {
    worker_id = task & 0xff;
    stage_id = (task & 0xff00) >> 8;
    stream_id = (task & 0xffff0000) >> 16;
    datapar_id = (task &0xff00000000) >> 32;
}
```

```

uint64_t inline make_task(uint8_t worker_id, uint8_t stage_id,
    uint16_t stream_id, uint8_t datapar_id) {
    uint64_t ret = 0x7000000000000000;
    ret |= worker_id;
    ret |= stage_id * (1 << 8);
    ret |= stream_id * (1 << 16);
    ret |= datapar_id * (1 << 32);
    return ret;
}

bigtask_t* __stream;
int* __counters;
#endif
```

A.2.5 stages.hpp

```
#ifndef __STAGES_HPP
#define __STAGES_HPP

#include "task.hpp"
#include "../stages/00_census_transform.hpp"
#include "../stages/01_cost_initialization.hpp"
#include "../stages/02_cross_building.hpp"
#include "../stages/03_support_size_computation.hpp"
#include "../stages/04_aggregation.hpp"
#include "../stages/05_scanline_optimization.hpp"
#include "../stages/06_disparity_estimation.hpp"
#include "../stages/07_outlier_detection.hpp"
#include "../stages/08_iterative_voting.hpp"
#include "../stages/09_proper_interpolation.hpp"
#include "../stages/10_edge_detection.hpp"
#include "../stages/11_discontinuity_adjustment.hpp"
#include "../stages/12_sub_pixel_enhancement.hpp"
#include "../stages/13_median_filter.hpp"

void dummy(bigtask_t* b, int i){};
```

```

const stage_t stages[] = {
    stage_t(TASK2, "__census_init",
        __callback2(__census_init), 0, -1),
    stage_t(TASK2, "__census_init",
        __callback2(__census_init), 1, -1),

    stage_t(TASK2, "__cross_building",
        __callback2(__cross_building), 0, -1),
    stage_t(TASK2, "__cross_building",
        __callback2(__cross_building), 1, -1),
    stage_t(TASK2, "__HV_supp_compute",
        __callback2(__HV_supp_compute), 0, -1),
    stage_t(TASK2, "__HV_supp_compute",
        __callback2(__HV_supp_compute), 1, -1),
    stage_t(TASK2, "__VH_supp_compute",
        __callback2(__VH_supp_compute), 0, -1),
    stage_t(TASK2, "__VH_supp_compute",
        __callback2(__VH_supp_compute), 1, -1),

    stage_t(TASK1, "__cost_initialization",
        __callback1(__cost_initialization), -1, -1),

    // Aggregation
    stage_t(TASK2, "__horizontal",
        __callback3(__horizontal), 0, -1),
    stage_t(TASK2, "__vertical",
        __callback3(__vertical), 1, -1),
    stage_t(TASK2, "__HV_supp_normalize",
        __callback3(__HV_supp_normalize), 0, -1),

    stage_t(TASK2, "__vertical",
        __callback3(__vertical), 0, -1),
    stage_t(TASK2, "__horizontal",
        __callback3(__horizontal), 1, -1),
    stage_t(TASK2, "__VH_supp_normalize",
        __callback3(__VH_supp_normalize), 0, -1),

    stage_t(TASK2, "__horizontal",
        __callback3(__horizontal), 0, -1),
    stage_t(TASK2, "__vertical",
        __callback3(__vertical), 1, -1),
    stage_t(TASK2, "__HV_supp_normalize",
        __callback3(__HV_supp_normalize), 0, -1),

    stage_t(TASK2, "__vertical",
        __callback3(__vertical), 0, -1),
    stage_t(TASK2, "__horizontal",
        __callback3(__horizontal), 1, -1),
    stage_t(TASK2, "__VH_supp_normalize",
        __callback3(__VH_supp_normalize), 0, -1),

    stage_t(TASK2, "__aggregation_finalization",
        __callback2(__aggregation_finalization), -1, -1),

    // Scanline optimization
    stage_t(TASK2, "__scanline_optimization",
        __callback2(__scanline_optimization), 0, -1),
    stage_t(TASK2, "__scanline_optimization",
        __callback2(__scanline_optimization), 1, -1),
    // END Scanline optimization

    // Disparity estimation
    stage_t(TASK2, "__disparity_estimation",
        __callback2(__disparity_estimation), 0, -1),
    stage_t(TASK2, "__disparity_estimation",
        __callback2(__disparity_estimation), 1, -1),
    // END Disparity estimation

    stage_t(TASK1, "__outlier_detection",
        __callback1(__outlier_detection), -1, -1),

    // Iterative voting
    stage_t(TASK1, "__histogram_init",
        __callback1(__histogram_init), -1, -1),
    stage_t(TASK1, "__histogram_computation_H",
        __callback1(__histogram_computation_H), -1, -1),
    stage_t(TASK1, "__histogram_computation_V",
        __callback1(__histogram_computation_V), -1, -1),
    stage_t(TASK1, "__iterative_voting",
        __callback1(__iterative_voting), -1, -1),
    stage_t(TASK1, "__vote_consolidation",
        __callback1(__vote_consolidation), -1, -1),

    stage_t(TASK1, "__histogram_init",
        __callback1(__histogram_init), -1, -1),
    stage_t(TASK1, "__histogram_computation_H",
        __callback1(__histogram_computation_H), -1, -1),
    stage_t(TASK1, "__histogram_computation_V",
        __callback1(__histogram_computation_V), -1, -1),
    stage_t(TASK1, "__iterative_voting",
        __callback1(__iterative_voting), -1, -1),
    stage_t(TASK1, "__vote_consolidation",
        __callback1(__vote_consolidation), -1, -1),

    stage_t(TASK1, "__histogram_init",
        __callback1(__histogram_init), -1, -1),
    stage_t(TASK1, "__histogram_computation_H",
        __callback1(__histogram_computation_H), -1, -1),
    stage_t(TASK1, "__histogram_computation_V",
        __callback1(__histogram_computation_V), -1, -1),
    stage_t(TASK1, "__iterative_voting",
        __callback1(__iterative_voting), -1, -1),
    stage_t(TASK1, "__vote_consolidation",
        __callback1(__vote_consolidation), -1, -1),

    stage_t(TASK1, "__histogram_init",
        __callback1(__histogram_init), -1, -1),
    stage_t(TASK1, "__histogram_computation_H",
        __callback1(__histogram_computation_H), -1, -1),
    stage_t(TASK1, "__histogram_computation_V",
        __callback1(__histogram_computation_V), -1, -1),
stage_t(TASK1, "__iterative_voting",
        __callback1(__iterative_voting), -1, -1),
stage_t(TASK1, "__vote_consolidation",
        __callback1(__vote_consolidation), -1, -1),

stage_t(TASK1, "__histogram_init",
        __callback1(__histogram_init), -1, -1),
stage_t(TASK1, "__histogram_computation_H",
        __callback1(__histogram_computation_H), -1, -1),
stage_t(TASK1, "__histogram_computation_V",
        __callback1(__histogram_computation_V), -1, -1),
stage_t(TASK1, "__iterative_voting",
        __callback1(__iterative_voting), -1, -1),
stage_t(TASK1, "__vote_consolidation",
        __callback1(__vote_consolidation), -1, -1),
// END Iterative voting

// Interpolation
stage_t(TASK1, "__proper_interpolation",
        __callback1(__proper_interpolation), -1, -1),
stage_t(TASK1, "__interpolation_consolidation",
        __callback1(__interpolation_consolidation), -1, -1),
// END Interpolation

stage_t(TASK1, "__edge_detection",
        __callback1(__edge_detection), -1, -1),
stage_t(TASK1, "__discontinuity_adjustment",
        __callback1(__discontinuity_adjustment), -1, -1),
stage_t(TASK1, "__sub_pixel_enhancement",
        __callback1(__sub_pixel_enhancement), -1, -1),

stage_t(TASK1, "__median_filter",
        __callback1(__median_filter), -1, -1),

stage_t(TASK1, nullptr, __callback1(dummy), -1, -1)
};
```

```

#endif
```