#include "darknet.h"

#include "yolo\_v2\_class.hpp"

#include "network.h"

extern "C" {

#include "detection\_layer.h"

#include "region\_layer.h"

#include "cost\_layer.h"

#include "utils.h"

#include "parser.h"

#include "box.h"

#include "image.h"

#include "demo.h"

#include "option\_list.h"

#include "stb\_image.h"

}

//#include <sys/time.h>

#include <vector>

#include <iostream>

#include <algorithm>

#include <cmath>

#define NFRAMES 3

//static Detector\* detector = NULL;

static std::unique\_ptr<Detector> detector;

int init(const char \*configurationFilename, const char \*weightsFilename, int gpu)

{

detector.reset(new Detector(configurationFilename, weightsFilename, gpu));

return 1;

}

int detect\_image(const char \*filename, bbox\_t\_container &container)

{

std::vector<bbox\_t> detection = detector->detect(filename);

for (size\_t i = 0; i < detection.size() && i < C\_SHARP\_MAX\_OBJECTS; ++i)

container.candidates[i] = detection[i];

return detection.size();

}

int detect\_mat(const uint8\_t\* data, const size\_t data\_length, bbox\_t\_container &container) {

#ifdef OPENCV

std::vector<char> vdata(data, data + data\_length);

cv::Mat image = imdecode(cv::Mat(vdata), 1);

std::vector<bbox\_t> detection = detector->detect(image);

for (size\_t i = 0; i < detection.size() && i < C\_SHARP\_MAX\_OBJECTS; ++i)

container.candidates[i] = detection[i];

return detection.size();

#else

return -1;

#endif // OPENCV

}

int dispose() {

//if (detector != NULL) delete detector;

//detector = NULL;

detector.reset();

return 1;

}

int get\_device\_count() {

#ifdef GPU

int count = 0;

cudaGetDeviceCount(&count);

return count;

#else

return -1;

#endif // GPU

}

bool built\_with\_cuda(){

#ifdef GPU

return true;

#else

return false;

#endif

}

bool built\_with\_cudnn(){

#ifdef CUDNN

return true;

#else

return false;

#endif

}

bool built\_with\_opencv(){

#ifdef OPENCV

return true;

#else

return false;

#endif

}

int get\_device\_name(int gpu, char\* deviceName) {

#ifdef GPU

cudaDeviceProp prop;

cudaGetDeviceProperties(&prop, gpu);

std::string result = prop.name;

std::copy(result.begin(), result.end(), deviceName);

return 1;

#else

return -1;

#endif // GPU

}

#ifdef GPU

void check\_cuda(cudaError\_t status) {

if (status != cudaSuccess) {

const char \*s = cudaGetErrorString(status);

printf("CUDA Error Prev: %s\n", s);

}

}

#endif

struct detector\_gpu\_t {

network net;

image images[NFRAMES];

float \*avg;

float\* predictions[NFRAMES];

int demo\_index;

unsigned int \*track\_id;

};

LIB\_API Detector::Detector(std::string cfg\_filename, std::string weight\_filename, int gpu\_id) : cur\_gpu\_id(gpu\_id)

{

wait\_stream = 0;

#ifdef GPU

int old\_gpu\_index;

check\_cuda( cudaGetDevice(&old\_gpu\_index) );

#endif

detector\_gpu\_ptr = std::make\_shared<detector\_gpu\_t>();

detector\_gpu\_t &detector\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

#ifdef GPU

//check\_cuda( cudaSetDevice(cur\_gpu\_id) );

cuda\_set\_device(cur\_gpu\_id);

printf(" Used GPU %d \n", cur\_gpu\_id);

#endif

network &net = detector\_gpu.net;

net.gpu\_index = cur\_gpu\_id;

//gpu\_index = i;

\_cfg\_filename = cfg\_filename;

\_weight\_filename = weight\_filename;

char \*cfgfile = const\_cast<char \*>(\_cfg\_filename.c\_str());

char \*weightfile = const\_cast<char \*>(\_weight\_filename.c\_str());

net = parse\_network\_cfg\_custom(cfgfile, 1, 1);

if (weightfile) {

load\_weights(&net, weightfile);

}

set\_batch\_network(&net, 1);

net.gpu\_index = cur\_gpu\_id;

fuse\_conv\_batchnorm(net);

layer l = net.layers[net.n - 1];

int j;

detector\_gpu.avg = (float \*)calloc(l.outputs, sizeof(float));

for (j = 0; j < NFRAMES; ++j) detector\_gpu.predictions[j] = (float\*)calloc(l.outputs, sizeof(float));

for (j = 0; j < NFRAMES; ++j) detector\_gpu.images[j] = make\_image(1, 1, 3);

detector\_gpu.track\_id = (unsigned int \*)calloc(l.classes, sizeof(unsigned int));

for (j = 0; j < l.classes; ++j) detector\_gpu.track\_id[j] = 1;

#ifdef GPU

check\_cuda( cudaSetDevice(old\_gpu\_index) );

#endif

}

LIB\_API Detector::~Detector()

{

detector\_gpu\_t &detector\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

//layer l = detector\_gpu.net.layers[detector\_gpu.net.n - 1];

free(detector\_gpu.track\_id);

free(detector\_gpu.avg);

for (int j = 0; j < NFRAMES; ++j) free(detector\_gpu.predictions[j]);

for (int j = 0; j < NFRAMES; ++j) if (detector\_gpu.images[j].data) free(detector\_gpu.images[j].data);

#ifdef GPU

int old\_gpu\_index;

cudaGetDevice(&old\_gpu\_index);

cuda\_set\_device(detector\_gpu.net.gpu\_index);

#endif

free\_network(detector\_gpu.net);

#ifdef GPU

cudaSetDevice(old\_gpu\_index);

#endif

}

LIB\_API int Detector::get\_net\_width() const {

detector\_gpu\_t &detector\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

return detector\_gpu.net.w;

}

LIB\_API int Detector::get\_net\_height() const {

detector\_gpu\_t &detector\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

return detector\_gpu.net.h;

}

LIB\_API int Detector::get\_net\_color\_depth() const {

detector\_gpu\_t &detector\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

return detector\_gpu.net.c;

}

LIB\_API std::vector<bbox\_t> Detector::detect(std::string image\_filename, float thresh, bool use\_mean)

{

std::shared\_ptr<image\_t> image\_ptr(new image\_t, [](image\_t \*img) { if (img->data) free(img->data); delete img; });

\*image\_ptr = load\_image(image\_filename);

return detect(\*image\_ptr, thresh, use\_mean);

}

static image load\_image\_stb(char \*filename, int channels)

{

int w, h, c;

unsigned char \*data = stbi\_load(filename, &w, &h, &c, channels);

if (!data)

throw std::runtime\_error("file not found");

if (channels) c = channels;

int i, j, k;

image im = make\_image(w, h, c);

for (k = 0; k < c; ++k) {

for (j = 0; j < h; ++j) {

for (i = 0; i < w; ++i) {

int dst\_index = i + w\*j + w\*h\*k;

int src\_index = k + c\*i + c\*w\*j;

im.data[dst\_index] = (float)data[src\_index] / 255.;

}

}

}

free(data);

return im;

}

LIB\_API image\_t Detector::load\_image(std::string image\_filename)

{

char \*input = const\_cast<char \*>(image\_filename.c\_str());

image im = load\_image\_stb(input, 3);

image\_t img;

img.c = im.c;

img.data = im.data;

img.h = im.h;

img.w = im.w;

return img;

}

LIB\_API void Detector::free\_image(image\_t m)

{

if (m.data) {

free(m.data);

}

}

LIB\_API std::vector<bbox\_t> Detector::detect(image\_t img, float thresh, bool use\_mean)

{

detector\_gpu\_t &detector\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

network &net = detector\_gpu.net;

#ifdef GPU

int old\_gpu\_index;

cudaGetDevice(&old\_gpu\_index);

if(cur\_gpu\_id != old\_gpu\_index)

cudaSetDevice(net.gpu\_index);

net.wait\_stream = wait\_stream; // 1 - wait CUDA-stream, 0 - not to wait

#endif

//std::cout << "net.gpu\_index = " << net.gpu\_index << std::endl;

image im;

im.c = img.c;

im.data = img.data;

im.h = img.h;

im.w = img.w;

image sized;

if (net.w == im.w && net.h == im.h) {

sized = make\_image(im.w, im.h, im.c);

memcpy(sized.data, im.data, im.w\*im.h\*im.c \* sizeof(float));

}

else

sized = resize\_image(im, net.w, net.h);

layer l = net.layers[net.n - 1];

float \*X = sized.data;

float \*prediction = network\_predict(net, X);

if (use\_mean) {

memcpy(detector\_gpu.predictions[detector\_gpu.demo\_index], prediction, l.outputs \* sizeof(float));

mean\_arrays(detector\_gpu.predictions, NFRAMES, l.outputs, detector\_gpu.avg);

l.output = detector\_gpu.avg;

detector\_gpu.demo\_index = (detector\_gpu.demo\_index + 1) % NFRAMES;

}

//get\_region\_boxes(l, 1, 1, thresh, detector\_gpu.probs, detector\_gpu.boxes, 0, 0);

//if (nms) do\_nms\_sort(detector\_gpu.boxes, detector\_gpu.probs, l.w\*l.h\*l.n, l.classes, nms);

int nboxes = 0;

int letterbox = 0;

float hier\_thresh = 0.5;

detection \*dets = get\_network\_boxes(&net, im.w, im.h, thresh, hier\_thresh, 0, 1, &nboxes, letterbox);

if (nms) do\_nms\_sort(dets, nboxes, l.classes, nms);

std::vector<bbox\_t> bbox\_vec;

for (int i = 0; i < nboxes; ++i) {

box b = dets[i].bbox;

int const obj\_id = max\_index(dets[i].prob, l.classes);

float const prob = dets[i].prob[obj\_id];

if (prob > thresh)

{

bbox\_t bbox;

bbox.x = std::max((double)0, (b.x - b.w / 2.)\*im.w);

bbox.y = std::max((double)0, (b.y - b.h / 2.)\*im.h);

bbox.w = b.w\*im.w;

bbox.h = b.h\*im.h;

bbox.obj\_id = obj\_id;

bbox.prob = prob;

bbox.track\_id = 0;

bbox.frames\_counter = 0;

bbox.x\_3d = NAN;

bbox.y\_3d = NAN;

bbox.z\_3d = NAN;

bbox\_vec.push\_back(bbox);

}

}

free\_detections(dets, nboxes);

if(sized.data)

free(sized.data);

#ifdef GPU

if (cur\_gpu\_id != old\_gpu\_index)

cudaSetDevice(old\_gpu\_index);

#endif

return bbox\_vec;

}

LIB\_API std::vector<bbox\_t> Detector::tracking\_id(std::vector<bbox\_t> cur\_bbox\_vec, bool const change\_history,

int const frames\_story, int const max\_dist)

{

detector\_gpu\_t &det\_gpu = \*static\_cast<detector\_gpu\_t \*>(detector\_gpu\_ptr.get());

bool prev\_track\_id\_present = false;

for (auto &i : prev\_bbox\_vec\_deque)

if (i.size() > 0) prev\_track\_id\_present = true;

if (!prev\_track\_id\_present) {

for (size\_t i = 0; i < cur\_bbox\_vec.size(); ++i)

cur\_bbox\_vec[i].track\_id = det\_gpu.track\_id[cur\_bbox\_vec[i].obj\_id]++;

prev\_bbox\_vec\_deque.push\_front(cur\_bbox\_vec);

if (prev\_bbox\_vec\_deque.size() > frames\_story) prev\_bbox\_vec\_deque.pop\_back();

return cur\_bbox\_vec;

}

std::vector<unsigned int> dist\_vec(cur\_bbox\_vec.size(), std::numeric\_limits<unsigned int>::max());

for (auto &prev\_bbox\_vec : prev\_bbox\_vec\_deque) {

for (auto &i : prev\_bbox\_vec) {

int cur\_index = -1;

for (size\_t m = 0; m < cur\_bbox\_vec.size(); ++m) {

bbox\_t const& k = cur\_bbox\_vec[m];

if (i.obj\_id == k.obj\_id) {

float center\_x\_diff = (float)(i.x + i.w/2) - (float)(k.x + k.w/2);

float center\_y\_diff = (float)(i.y + i.h/2) - (float)(k.y + k.h/2);

unsigned int cur\_dist = sqrt(center\_x\_diff\*center\_x\_diff + center\_y\_diff\*center\_y\_diff);

if (cur\_dist < max\_dist && (k.track\_id == 0 || dist\_vec[m] > cur\_dist)) {

dist\_vec[m] = cur\_dist;

cur\_index = m;

}

}

}

bool track\_id\_absent = !std::any\_of(cur\_bbox\_vec.begin(), cur\_bbox\_vec.end(),

[&i](bbox\_t const& b) { return b.track\_id == i.track\_id && b.obj\_id == i.obj\_id; });

if (cur\_index >= 0 && track\_id\_absent){

cur\_bbox\_vec[cur\_index].track\_id = i.track\_id;

cur\_bbox\_vec[cur\_index].w = (cur\_bbox\_vec[cur\_index].w + i.w) / 2;

cur\_bbox\_vec[cur\_index].h = (cur\_bbox\_vec[cur\_index].h + i.h) / 2;

}

}

}

for (size\_t i = 0; i < cur\_bbox\_vec.size(); ++i)

if (cur\_bbox\_vec[i].track\_id == 0)

cur\_bbox\_vec[i].track\_id = det\_gpu.track\_id[cur\_bbox\_vec[i].obj\_id]++;

if (change\_history) {

prev\_bbox\_vec\_deque.push\_front(cur\_bbox\_vec);

if (prev\_bbox\_vec\_deque.size() > frames\_story) prev\_bbox\_vec\_deque.pop\_back();

}

return cur\_bbox\_vec;

}

void \*Detector::get\_cuda\_context()

{

#ifdef GPU

int old\_gpu\_index;

cudaGetDevice(&old\_gpu\_index);

if (cur\_gpu\_id != old\_gpu\_index)

cudaSetDevice(cur\_gpu\_id);

void \*cuda\_context = cuda\_get\_context();

if (cur\_gpu\_id != old\_gpu\_index)

cudaSetDevice(old\_gpu\_index);

return cuda\_context;

#else // GPU

return NULL;

#endif // GPU

}