#include "maxpool\_layer.h"

#include "convolutional\_layer.h"

#include "dark\_cuda.h"

#include "utils.h"

#include "gemm.h"

#include <stdio.h>

image get\_maxpool\_image(maxpool\_layer l)

{

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

return float\_to\_image(w,h,c,l.output);

}

image get\_maxpool\_delta(maxpool\_layer l)

{

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

return float\_to\_image(w,h,c,l.delta);

}

void create\_maxpool\_cudnn\_tensors(layer \*l)

{

#ifdef CUDNN

CHECK\_CUDNN(cudnnCreatePoolingDescriptor(&l->poolingDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->srcTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->dstTensorDesc));

#endif // CUDNN

}

void cudnn\_maxpool\_setup(layer \*l)

{

#ifdef CUDNN

CHECK\_CUDNN(cudnnSetPooling2dDescriptor(

l->poolingDesc,

CUDNN\_POOLING\_MAX,

CUDNN\_NOT\_PROPAGATE\_NAN, // CUDNN\_PROPAGATE\_NAN, CUDNN\_NOT\_PROPAGATE\_NAN

l->size,

l->size,

l->pad/2, //0, //l.pad,

l->pad/2, //0, //l.pad,

l->stride\_x,

l->stride\_y));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->srcTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, l->batch, l->c, l->h, l->w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->dstTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, l->batch, l->out\_c, l->out\_h, l->out\_w));

#endif // CUDNN

}

void cudnn\_local\_avgpool\_setup(layer \*l)

{

#ifdef CUDNN

CHECK\_CUDNN(cudnnSetPooling2dDescriptor(

l->poolingDesc,

CUDNN\_POOLING\_AVERAGE\_COUNT\_EXCLUDE\_PADDING,

CUDNN\_NOT\_PROPAGATE\_NAN, // CUDNN\_PROPAGATE\_NAN, CUDNN\_NOT\_PROPAGATE\_NAN

l->size,

l->size,

l->pad / 2, //0, //l.pad,

l->pad / 2, //0, //l.pad,

l->stride\_x,

l->stride\_y));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->srcTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, l->batch, l->c, l->h, l->w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->dstTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, l->batch, l->out\_c, l->out\_h, l->out\_w));

#endif // CUDNN

}

maxpool\_layer make\_maxpool\_layer(int batch, int h, int w, int c, int size, int stride\_x, int stride\_y, int padding, int maxpool\_depth, int out\_channels, int antialiasing, int avgpool, int train)

{

maxpool\_layer l = { (LAYER\_TYPE)0 };

l.avgpool = avgpool;

if (avgpool) l.type = LOCAL\_AVGPOOL;

else l.type = MAXPOOL;

l.train = train;

const int blur\_stride\_x = stride\_x;

const int blur\_stride\_y = stride\_y;

l.antialiasing = antialiasing;

if (antialiasing) {

stride\_x = stride\_y = l.stride = l.stride\_x = l.stride\_y = 1; // use stride=1 in host-layer

}

l.batch = batch;

l.h = h;

l.w = w;

l.c = c;

l.pad = padding;

l.maxpool\_depth = maxpool\_depth;

l.out\_channels = out\_channels;

if (maxpool\_depth) {

l.out\_c = out\_channels;

l.out\_w = l.w;

l.out\_h = l.h;

}

else {

l.out\_w = (w + padding - size) / stride\_x + 1;

l.out\_h = (h + padding - size) / stride\_y + 1;

l.out\_c = c;

}

l.outputs = l.out\_h \* l.out\_w \* l.out\_c;

l.inputs = h\*w\*c;

l.size = size;

l.stride = stride\_x;

l.stride\_x = stride\_x;

l.stride\_y = stride\_y;

int output\_size = l.out\_h \* l.out\_w \* l.out\_c \* batch;

if (train) {

if (!avgpool) l.indexes = (int\*)xcalloc(output\_size, sizeof(int));

l.delta = (float\*)xcalloc(output\_size, sizeof(float));

}

l.output = (float\*)xcalloc(output\_size, sizeof(float));

if (avgpool) {

l.forward = forward\_local\_avgpool\_layer;

l.backward = backward\_local\_avgpool\_layer;

}

else {

l.forward = forward\_maxpool\_layer;

l.backward = backward\_maxpool\_layer;

}

#ifdef GPU

if (avgpool) {

l.forward\_gpu = forward\_local\_avgpool\_layer\_gpu;

l.backward\_gpu = backward\_local\_avgpool\_layer\_gpu;

}

else {

l.forward\_gpu = forward\_maxpool\_layer\_gpu;

l.backward\_gpu = backward\_maxpool\_layer\_gpu;

}

if (train) {

if (!avgpool) l.indexes\_gpu = cuda\_make\_int\_array(output\_size);

l.delta\_gpu = cuda\_make\_array(l.delta, output\_size);

}

l.output\_gpu = cuda\_make\_array(l.output, output\_size);

create\_maxpool\_cudnn\_tensors(&l);

if (avgpool) cudnn\_local\_avgpool\_setup(&l);

else cudnn\_maxpool\_setup(&l);

#endif // GPU

l.bflops = (l.size\*l.size\*l.c \* l.out\_h\*l.out\_w) / 1000000000.;

if (avgpool) {

if (stride\_x == stride\_y)

fprintf(stderr, "avg %2dx%2d/%2d %4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", size, size, stride\_x, w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

else

fprintf(stderr, "avg %2dx%2d/%2dx%2d %4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", size, size, stride\_x, stride\_y, w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

}

else {

if (maxpool\_depth)

fprintf(stderr, "max-depth %2dx%2d/%2d %4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", size, size, stride\_x, w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

else if (stride\_x == stride\_y)

fprintf(stderr, "max %2dx%2d/%2d %4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", size, size, stride\_x, w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

else

fprintf(stderr, "max %2dx%2d/%2dx%2d %4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", size, size, stride\_x, stride\_y, w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

}

if (l.antialiasing) {

printf("AA: ");

l.input\_layer = (layer\*)calloc(1, sizeof(layer));

int blur\_size = 3;

int blur\_pad = blur\_size / 2;

if (l.antialiasing == 2) {

blur\_size = 2;

blur\_pad = 0;

}

\*(l.input\_layer) = make\_convolutional\_layer(batch, 1, l.out\_h, l.out\_w, l.out\_c, l.out\_c, l.out\_c, blur\_size, blur\_stride\_x, blur\_stride\_y, 1, blur\_pad, LINEAR, 0, 0, 0, 0, 0, 1, 0, NULL, 0, 0, train);

const int blur\_nweights = l.out\_c \* blur\_size \* blur\_size; // (n / n) \* n \* blur\_size \* blur\_size;

int i;

if (blur\_size == 2) {

for (i = 0; i < blur\_nweights; i += (blur\_size\*blur\_size)) {

l.input\_layer->weights[i + 0] = 1 / 4.f;

l.input\_layer->weights[i + 1] = 1 / 4.f;

l.input\_layer->weights[i + 2] = 1 / 4.f;

l.input\_layer->weights[i + 3] = 1 / 4.f;

}

}

else {

for (i = 0; i < blur\_nweights; i += (blur\_size\*blur\_size)) {

l.input\_layer->weights[i + 0] = 1 / 16.f;

l.input\_layer->weights[i + 1] = 2 / 16.f;

l.input\_layer->weights[i + 2] = 1 / 16.f;

l.input\_layer->weights[i + 3] = 2 / 16.f;

l.input\_layer->weights[i + 4] = 4 / 16.f;

l.input\_layer->weights[i + 5] = 2 / 16.f;

l.input\_layer->weights[i + 6] = 1 / 16.f;

l.input\_layer->weights[i + 7] = 2 / 16.f;

l.input\_layer->weights[i + 8] = 1 / 16.f;

}

}

for (i = 0; i < l.out\_c; ++i) l.input\_layer->biases[i] = 0;

#ifdef GPU

if (gpu\_index >= 0) {

if (l.antialiasing) l.input\_antialiasing\_gpu = cuda\_make\_array(NULL, l.batch\*l.outputs);

push\_convolutional\_layer(\*(l.input\_layer));

}

#endif // GPU

}

return l;

}

void resize\_maxpool\_layer(maxpool\_layer \*l, int w, int h)

{

l->h = h;

l->w = w;

l->inputs = h\*w\*l->c;

l->out\_w = (w + l->pad - l->size) / l->stride\_x + 1;

l->out\_h = (h + l->pad - l->size) / l->stride\_y + 1;

l->outputs = l->out\_w \* l->out\_h \* l->out\_c;

int output\_size = l->outputs \* l->batch;

if (l->train) {

if (!l->avgpool) l->indexes = (int\*)xrealloc(l->indexes, output\_size \* sizeof(int));

l->delta = (float\*)xrealloc(l->delta, output\_size \* sizeof(float));

}

l->output = (float\*)xrealloc(l->output, output\_size \* sizeof(float));

#ifdef GPU

CHECK\_CUDA(cudaFree(l->output\_gpu));

l->output\_gpu = cuda\_make\_array(l->output, output\_size);

if (l->train) {

if (!l->avgpool) {

CHECK\_CUDA(cudaFree((float \*)l->indexes\_gpu));

l->indexes\_gpu = cuda\_make\_int\_array(output\_size);

}

CHECK\_CUDA(cudaFree(l->delta\_gpu));

l->delta\_gpu = cuda\_make\_array(l->delta, output\_size);

}

if(l->avgpool) cudnn\_local\_avgpool\_setup(l);

else cudnn\_maxpool\_setup(l);

#endif

}

void forward\_maxpool\_layer(const maxpool\_layer l, network\_state state)

{

if (l.maxpool\_depth)

{

int b, i, j, k, g;

for (b = 0; b < l.batch; ++b) {

#pragma omp parallel for

for (i = 0; i < l.h; ++i) {

for (j = 0; j < l.w; ++j) {

for (g = 0; g < l.out\_c; ++g)

{

int out\_index = j + l.w\*(i + l.h\*(g + l.out\_c\*b));

float max = -FLT\_MAX;

int max\_i = -1;

for (k = g; k < l.c; k += l.out\_c)

{

int in\_index = j + l.w\*(i + l.h\*(k + l.c\*b));

float val = state.input[in\_index];

max\_i = (val > max) ? in\_index : max\_i;

max = (val > max) ? val : max;

}

l.output[out\_index] = max;

if (l.indexes) l.indexes[out\_index] = max\_i;

}

}

}

}

return;

}

if (!state.train && l.stride\_x == l.stride\_y) {

forward\_maxpool\_layer\_avx(state.input, l.output, l.indexes, l.size, l.w, l.h, l.out\_w, l.out\_h, l.c, l.pad, l.stride, l.batch);

}

else

{

int b, i, j, k, m, n;

int w\_offset = -l.pad / 2;

int h\_offset = -l.pad / 2;

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

for (b = 0; b < l.batch; ++b) {

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

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

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

int out\_index = j + w\*(i + h\*(k + c\*b));

float max = -FLT\_MAX;

int max\_i = -1;

for (n = 0; n < l.size; ++n) {

for (m = 0; m < l.size; ++m) {

int cur\_h = h\_offset + i\*l.stride\_y + n;

int cur\_w = w\_offset + j\*l.stride\_x + m;

int index = cur\_w + l.w\*(cur\_h + l.h\*(k + b\*l.c));

int valid = (cur\_h >= 0 && cur\_h < l.h &&

cur\_w >= 0 && cur\_w < l.w);

float val = (valid != 0) ? state.input[index] : -FLT\_MAX;

max\_i = (val > max) ? index : max\_i;

max = (val > max) ? val : max;

}

}

l.output[out\_index] = max;

if (l.indexes) l.indexes[out\_index] = max\_i;

}

}

}

}

}

if (l.antialiasing) {

network\_state s = { 0 };

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

s.input = l.output;

forward\_convolutional\_layer(\*(l.input\_layer), s);

//simple\_copy\_ongpu(l.outputs\*l.batch, l.output, l.input\_antialiasing);

memcpy(l.output, l.input\_layer->output, l.input\_layer->outputs \* l.input\_layer->batch \* sizeof(float));

}

}

void backward\_maxpool\_layer(const maxpool\_layer l, network\_state state)

{

int i;

int h = l.out\_h;

int w = l.out\_w;

int c = l.out\_c;

#pragma omp parallel for

for(i = 0; i < h\*w\*c\*l.batch; ++i){

int index = l.indexes[i];

state.delta[index] += l.delta[i];

}

}

void forward\_local\_avgpool\_layer(const maxpool\_layer l, network\_state state)

{

int b, i, j, k, m, n;

int w\_offset = -l.pad / 2;

int h\_offset = -l.pad / 2;

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

for (b = 0; b < l.batch; ++b) {

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

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

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

int out\_index = j + w\*(i + h\*(k + c\*b));

float avg = 0;

int counter = 0;

for (n = 0; n < l.size; ++n) {

for (m = 0; m < l.size; ++m) {

int cur\_h = h\_offset + i\*l.stride\_y + n;

int cur\_w = w\_offset + j\*l.stride\_x + m;

int index = cur\_w + l.w\*(cur\_h + l.h\*(k + b\*l.c));

int valid = (cur\_h >= 0 && cur\_h < l.h &&

cur\_w >= 0 && cur\_w < l.w);

if (valid) {

counter++;

avg += state.input[index];

}

}

}

l.output[out\_index] = avg / counter;

}

}

}

}

}

void backward\_local\_avgpool\_layer(const maxpool\_layer l, network\_state state)

{

int b, i, j, k, m, n;

int w\_offset = -l.pad / 2;

int h\_offset = -l.pad / 2;

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

for (b = 0; b < l.batch; ++b) {

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

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

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

int out\_index = j + w\*(i + h\*(k + c\*b));

for (n = 0; n < l.size; ++n) {

for (m = 0; m < l.size; ++m) {

int cur\_h = h\_offset + i\*l.stride\_y + n;

int cur\_w = w\_offset + j\*l.stride\_x + m;

int index = cur\_w + l.w\*(cur\_h + l.h\*(k + b\*l.c));

int valid = (cur\_h >= 0 && cur\_h < l.h &&

cur\_w >= 0 && cur\_w < l.w);

if (valid) state.delta[index] += l.delta[out\_index] / (l.size\*l.size);

}

}

}

}

}

}

}