#include "shortcut\_layer.h"

#include "convolutional\_layer.h"

#include "dark\_cuda.h"

#include "blas.h"

#include "utils.h"

#include "gemm.h"

#include <stdio.h>

#include <assert.h>

layer make\_shortcut\_layer(int batch, int n, int \*input\_layers, int\* input\_sizes, int w, int h, int c,

float \*\*layers\_output, float \*\*layers\_delta, float \*\*layers\_output\_gpu, float \*\*layers\_delta\_gpu, WEIGHTS\_TYPE\_T weights\_type, WEIGHTS\_NORMALIZATION\_T weights\_normalizion,

ACTIVATION activation, int train)

{

fprintf(stderr, "Shortcut Layer: ");

int i;

for(i = 0; i < n; ++i) fprintf(stderr, "%d, ", input\_layers[i]);

layer l = { (LAYER\_TYPE)0 };

l.train = train;

l.type = SHORTCUT;

l.batch = batch;

l.activation = activation;

l.n = n;

l.input\_layers = input\_layers;

l.input\_sizes = input\_sizes;

l.layers\_output = layers\_output;

l.layers\_delta = layers\_delta;

l.weights\_type = weights\_type;

l.weights\_normalizion = weights\_normalizion;

l.learning\_rate\_scale = 1; // not necessary

//l.w = w2;

//l.h = h2;

//l.c = c2;

l.w = l.out\_w = w;

l.h = l.out\_h = h;

l.c = l.out\_c = c;

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

l.inputs = l.outputs;

//if(w != w2 || h != h2 || c != c2) fprintf(stderr, " w = %d, w2 = %d, h = %d, h2 = %d, c = %d, c2 = %d \n", w, w2, h, h2, c, c2);

l.index = l.input\_layers[0];

if (train) l.delta = (float\*)xcalloc(l.outputs \* batch, sizeof(float));

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

l.nweights = 0;

if (l.weights\_type == PER\_FEATURE) l.nweights = (l.n + 1);

else if (l.weights\_type == PER\_CHANNEL) l.nweights = (l.n + 1) \* l.c;

if (l.nweights > 0) {

l.weights = (float\*)calloc(l.nweights, sizeof(float));

float scale = sqrt(2. / l.nweights);

for (i = 0; i < l.nweights; ++i) l.weights[i] = 1;// +0.01\*rand\_uniform(-1, 1);// scale\*rand\_uniform(-1, 1); // rand\_normal();

if (train) l.weight\_updates = (float\*)calloc(l.nweights, sizeof(float));

l.update = update\_shortcut\_layer;

}

l.forward = forward\_shortcut\_layer;

l.backward = backward\_shortcut\_layer;

#ifndef GPU

if (l.activation == SWISH || l.activation == MISH) l.activation\_input = (float\*)calloc(l.batch\*l.outputs, sizeof(float));

#endif // GPU

#ifdef GPU

if (l.activation == SWISH || l.activation == MISH) l.activation\_input\_gpu = cuda\_make\_array(l.activation\_input, l.batch\*l.outputs);

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

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

if (l.nweights > 0) {

l.update\_gpu = update\_shortcut\_layer\_gpu;

l.weights\_gpu = cuda\_make\_array(l.weights, l.nweights);

if (train) l.weight\_updates\_gpu = cuda\_make\_array(l.weight\_updates, l.nweights);

}

if (train) l.delta\_gpu = cuda\_make\_array(l.delta, l.outputs\*batch);

l.output\_gpu = cuda\_make\_array(l.output, l.outputs\*batch);

l.input\_sizes\_gpu = cuda\_make\_int\_array\_new\_api(input\_sizes, l.n);

l.layers\_output\_gpu = (float\*\*)cuda\_make\_array\_pointers((void\*\*)layers\_output\_gpu, l.n);

l.layers\_delta\_gpu = (float\*\*)cuda\_make\_array\_pointers((void\*\*)layers\_delta\_gpu, l.n);

#endif // GPU

l.bflops = l.out\_w \* l.out\_h \* l.out\_c \* l.n / 1000000000.;

if (l.weights\_type) l.bflops \*= 2;

fprintf(stderr, " wt = %d, wn = %d, outputs:%4d x%4d x%4d %5.3f BF\n", l.weights\_type, l.weights\_normalizion, l.out\_w, l.out\_h, l.out\_c, l.bflops);

return l;

}

void resize\_shortcut\_layer(layer \*l, int w, int h, network \*net)

{

//assert(l->w == l->out\_w);

//assert(l->h == l->out\_h);

l->w = l->out\_w = w;

l->h = l->out\_h = h;

l->outputs = w\*h\*l->out\_c;

l->inputs = l->outputs;

if (l->train) l->delta = (float\*)xrealloc(l->delta, l->outputs \* l->batch \* sizeof(float));

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

int i;

for (i = 0; i < l->n; ++i) {

int index = l->input\_layers[i];

l->input\_sizes[i] = net->layers[index].outputs;

l->layers\_output[i] = net->layers[index].output;

l->layers\_delta[i] = net->layers[index].delta;

assert(l->w == net->layers[index].out\_w && l->h == net->layers[index].out\_h);

}

if (l->activation == SWISH || l->activation == MISH) l->activation\_input = (float\*)realloc(l->activation\_input, l->batch\*l->outputs \* sizeof(float));

#ifdef GPU

cuda\_free(l->output\_gpu);

l->output\_gpu = cuda\_make\_array(l->output, l->outputs\*l->batch);

if (l->train) {

cuda\_free(l->delta\_gpu);

l->delta\_gpu = cuda\_make\_array(l->delta, l->outputs\*l->batch);

}

float \*\*layers\_output\_gpu = (float \*\*)calloc(l->n, sizeof(float \*));

float \*\*layers\_delta\_gpu = (float \*\*)calloc(l->n, sizeof(float \*));

for (i = 0; i < l->n; ++i) {

const int index = l->input\_layers[i];

layers\_output\_gpu[i] = net->layers[index].output\_gpu;

layers\_delta\_gpu[i] = net->layers[index].delta\_gpu;

}

memcpy\_ongpu(l->input\_sizes\_gpu, l->input\_sizes, l->n \* sizeof(int));

memcpy\_ongpu(l->layers\_output\_gpu, layers\_output\_gpu, l->n \* sizeof(float\*));

memcpy\_ongpu(l->layers\_delta\_gpu, layers\_delta\_gpu, l->n \* sizeof(float\*));

free(layers\_output\_gpu);

free(layers\_delta\_gpu);

if (l->activation == SWISH || l->activation == MISH) {

cuda\_free(l->activation\_input\_gpu);

l->activation\_input\_gpu = cuda\_make\_array(l->activation\_input, l->batch\*l->outputs);

}

#endif

}

void forward\_shortcut\_layer(const layer l, network\_state state)

{

int from\_w = state.net.layers[l.index].w;

int from\_h = state.net.layers[l.index].h;

int from\_c = state.net.layers[l.index].c;

if (l.nweights == 0 && l.n == 1 && from\_w == l.w && from\_h == l.h && from\_c == l.c) {

int size = l.batch \* l.w \* l.h \* l.c;

int i;

#pragma omp parallel for

for(i = 0; i < size; ++i)

l.output[i] = state.input[i] + state.net.layers[l.index].output[i];

}

else {

shortcut\_multilayer\_cpu(l.outputs \* l.batch, l.outputs, l.batch, l.n, l.input\_sizes, l.layers\_output, l.output, state.input, l.weights, l.nweights, l.weights\_normalizion);

}

//copy\_cpu(l.outputs\*l.batch, state.input, 1, l.output, 1);

//shortcut\_cpu(l.batch, from\_w, from\_h, from\_c, state.net.layers[l.index].output, l.out\_w, l.out\_h, l.out\_c, l.output);

//activate\_array(l.output, l.outputs\*l.batch, l.activation);

if (l.activation == SWISH) activate\_array\_swish(l.output, l.outputs\*l.batch, l.activation\_input, l.output);

else if (l.activation == MISH) activate\_array\_mish(l.output, l.outputs\*l.batch, l.activation\_input, l.output);

else activate\_array\_cpu\_custom(l.output, l.outputs\*l.batch, l.activation);

}

void backward\_shortcut\_layer(const layer l, network\_state state)

{

if (l.activation == SWISH) gradient\_array\_swish(l.output, l.outputs\*l.batch, l.activation\_input, l.delta);

else if (l.activation == MISH) gradient\_array\_mish(l.outputs\*l.batch, l.activation\_input, l.delta);

else gradient\_array(l.output, l.outputs\*l.batch, l.activation, l.delta);

backward\_shortcut\_multilayer\_cpu(l.outputs \* l.batch, l.outputs, l.batch, l.n, l.input\_sizes,

l.layers\_delta, state.delta, l.delta, l.weights, l.weight\_updates, l.nweights, state.input, l.layers\_output, l.weights\_normalizion);

//axpy\_cpu(l.outputs\*l.batch, 1, l.delta, 1, state.delta, 1);

//shortcut\_cpu(l.batch, l.out\_w, l.out\_h, l.out\_c, l.delta, l.w, l.h, l.c, state.net.layers[l.index].delta);

}

void update\_shortcut\_layer(layer l, int batch, float learning\_rate\_init, float momentum, float decay)

{

if (l.nweights > 0) {

float learning\_rate = learning\_rate\_init\*l.learning\_rate\_scale;

//float momentum = a.momentum;

//float decay = a.decay;

//int batch = a.batch;

axpy\_cpu(l.nweights, -decay\*batch, l.weights, 1, l.weight\_updates, 1);

axpy\_cpu(l.nweights, learning\_rate / batch, l.weight\_updates, 1, l.weights, 1);

scal\_cpu(l.nweights, momentum, l.weight\_updates, 1);

}

}

#ifdef GPU

void forward\_shortcut\_layer\_gpu(const layer l, network\_state state)

{

//copy\_ongpu(l.outputs\*l.batch, state.input, 1, l.output\_gpu, 1);

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

//shortcut\_gpu(l.batch, l.w, l.h, l.c, state.net.layers[l.index].output\_gpu, l.out\_w, l.out\_h, l.out\_c, l.output\_gpu);

//input\_shortcut\_gpu(state.input, l.batch, l.w, l.h, l.c, state.net.layers[l.index].output\_gpu, l.out\_w, l.out\_h, l.out\_c, l.output\_gpu);

//-----------

//if (l.outputs == l.input\_sizes[0])

//if(l.n == 1 && l.nweights == 0)

//{

// input\_shortcut\_gpu(state.input, l.batch, state.net.layers[l.index].w, state.net.layers[l.index].h, state.net.layers[l.index].c,

// state.net.layers[l.index].output\_gpu, l.out\_w, l.out\_h, l.out\_c, l.output\_gpu);

//}

//else

{

shortcut\_multilayer\_gpu(l.outputs, l.batch, l.n, l.input\_sizes\_gpu, l.layers\_output\_gpu, l.output\_gpu, state.input, l.weights\_gpu, l.nweights, l.weights\_normalizion);

}

if (l.activation == SWISH) activate\_array\_swish\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation\_input\_gpu, l.output\_gpu);

else if (l.activation == MISH) activate\_array\_mish\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation\_input\_gpu, l.output\_gpu);

else activate\_array\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation);

}

void backward\_shortcut\_layer\_gpu(const layer l, network\_state state)

{

if (l.activation == SWISH) gradient\_array\_swish\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation\_input\_gpu, l.delta\_gpu);

else if (l.activation == MISH) gradient\_array\_mish\_ongpu(l.outputs\*l.batch, l.activation\_input\_gpu, l.delta\_gpu);

else gradient\_array\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation, l.delta\_gpu);

backward\_shortcut\_multilayer\_gpu(l.outputs, l.batch, l.n, l.input\_sizes\_gpu, l.layers\_delta\_gpu, state.delta, l.delta\_gpu,

l.weights\_gpu, l.weight\_updates\_gpu, l.nweights, state.input, l.layers\_output\_gpu, l.weights\_normalizion);

//axpy\_ongpu(l.outputs\*l.batch, 1, l.delta\_gpu, 1, state.delta, 1);

//shortcut\_gpu(l.batch, l.out\_w, l.out\_h, l.out\_c, l.delta\_gpu, l.w, l.h, l.c, state.net.layers[l.index].delta\_gpu);

}

void update\_shortcut\_layer\_gpu(layer l, int batch, float learning\_rate\_init, float momentum, float decay, float loss\_scale)

{

if (l.nweights > 0) {

float learning\_rate = learning\_rate\_init\*l.learning\_rate\_scale;

//float momentum = a.momentum;

//float decay = a.decay;

//int batch = a.batch;

// Loss scale for Mixed-Precision on Tensor-Cores

if (loss\_scale != 1.0) {

if(l.weight\_updates\_gpu && l.nweights > 0) scal\_ongpu(l.nweights, 1.0 / loss\_scale, l.weight\_updates\_gpu, 1);

}

reset\_nan\_and\_inf(l.weight\_updates\_gpu, l.nweights);

fix\_nan\_and\_inf(l.weights\_gpu, l.nweights);

//constrain\_weight\_updates\_ongpu(l.nweights, 1, l.weights\_gpu, l.weight\_updates\_gpu);

constrain\_ongpu(l.nweights, 1, l.weight\_updates\_gpu, 1);

/\*

cuda\_pull\_array\_async(l.weights\_gpu, l.weights, l.nweights);

cuda\_pull\_array\_async(l.weight\_updates\_gpu, l.weight\_updates, l.nweights);

CHECK\_CUDA(cudaStreamSynchronize(get\_cuda\_stream()));

for (int i = 0; i < l.nweights; ++i) printf(" %f, ", l.weight\_updates[i]);

printf(" l.nweights = %d - updates \n", l.nweights);

for (int i = 0; i < l.nweights; ++i) printf(" %f, ", l.weights[i]);

printf(" l.nweights = %d \n\n", l.nweights);

\*/

//axpy\_ongpu(l.nweights, -decay\*batch, l.weights\_gpu, 1, l.weight\_updates\_gpu, 1);

axpy\_ongpu(l.nweights, learning\_rate / batch, l.weight\_updates\_gpu, 1, l.weights\_gpu, 1);

scal\_ongpu(l.nweights, momentum, l.weight\_updates\_gpu, 1);

//fill\_ongpu(l.nweights, 0, l.weight\_updates\_gpu, 1);

//if (l.clip) {

// constrain\_ongpu(l.nweights, l.clip, l.weights\_gpu, 1);

//}

}

}

void pull\_shortcut\_layer(layer l)

{

constrain\_ongpu(l.nweights, 1, l.weight\_updates\_gpu, 1);

cuda\_pull\_array\_async(l.weight\_updates\_gpu, l.weight\_updates, l.nweights);

cuda\_pull\_array\_async(l.weights\_gpu, l.weights, l.nweights);

CHECK\_CUDA(cudaPeekAtLastError());

CHECK\_CUDA(cudaStreamSynchronize(get\_cuda\_stream()));

}

void push\_shortcut\_layer(layer l)

{

cuda\_push\_array(l.weights\_gpu, l.weights, l.nweights);

CHECK\_CUDA(cudaPeekAtLastError());

}

#endif