#include "im2col.h"

#include <stdio.h>

float im2col\_get\_pixel(float \*im, int height, int width, int channels,

int row, int col, int channel, int pad)

{

row -= pad;

col -= pad;

if (row < 0 || col < 0 ||

row >= height || col >= width) return 0;

return im[col + width\*(row + height\*channel)];

}

//From Berkeley Vision's Caffe!

//https://github.com/BVLC/caffe/blob/master/LICENSE

void im2col\_cpu(float\* data\_im,

int channels, int height, int width,

int ksize, int stride, int pad, float\* data\_col)

{

int c,h,w;

int height\_col = (height + 2\*pad - ksize) / stride + 1;

int width\_col = (width + 2\*pad - ksize) / stride + 1;

int channels\_col = channels \* ksize \* ksize;

for (c = 0; c < channels\_col; ++c) {

int w\_offset = c % ksize;

int h\_offset = (c / ksize) % ksize;

int c\_im = c / ksize / ksize;

for (h = 0; h < height\_col; ++h) {

for (w = 0; w < width\_col; ++w) {

int im\_row = h\_offset + h \* stride;

int im\_col = w\_offset + w \* stride;

int col\_index = (c \* height\_col + h) \* width\_col + w;

data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels,

im\_row, im\_col, c\_im, pad);

}

}

}

}

// Function uses casting from int to unsigned to compare if value of

// parameter a is greater or equal to zero and lower than value of

// parameter b. The b parameter is of type signed and is always positive,

// therefore its value is always lower than 0x800... where casting

// negative value of a parameter converts it to value higher than 0x800...

// The casting allows to use one condition instead of two.

inline static int is\_a\_ge\_zero\_and\_a\_lt\_b(int a, int b) {

return (unsigned)(a) < (unsigned)(b);

}

// https://github.com/BVLC/caffe/blob/master/src/caffe/util/im2col.cpp

void im2col\_cpu\_ext(const float\* data\_im, const int channels,

const int height, const int width, const int kernel\_h, const int kernel\_w,

const int pad\_h, const int pad\_w,

const int stride\_h, const int stride\_w,

const int dilation\_h, const int dilation\_w,

float\* data\_col)

{

const int output\_h = (height + 2 \* pad\_h -

(dilation\_h \* (kernel\_h - 1) + 1)) / stride\_h + 1;

const int output\_w = (width + 2 \* pad\_w -

(dilation\_w \* (kernel\_w - 1) + 1)) / stride\_w + 1;

const int channel\_size = height \* width;

int channel, kernel\_row, kernel\_col, output\_rows, output\_col;

for (channel = channels; channel--; data\_im += channel\_size) {

for (kernel\_row = 0; kernel\_row < kernel\_h; kernel\_row++) {

for (kernel\_col = 0; kernel\_col < kernel\_w; kernel\_col++) {

int input\_row = -pad\_h + kernel\_row \* dilation\_h;

for (output\_rows = output\_h; output\_rows; output\_rows--) {

if (!is\_a\_ge\_zero\_and\_a\_lt\_b(input\_row, height)) {

for (output\_col = output\_w; output\_col; output\_col--) {

\*(data\_col++) = 0;

}

}

else {

int input\_col = -pad\_w + kernel\_col \* dilation\_w;

for (output\_col = output\_w; output\_col; output\_col--) {

if (is\_a\_ge\_zero\_and\_a\_lt\_b(input\_col, width)) {

\*(data\_col++) = data\_im[input\_row \* width + input\_col];

}

else {

\*(data\_col++) = 0;

}

input\_col += stride\_w;

}

}

input\_row += stride\_h;

}

}

}

}

}