#include "gemm.h"

#include "utils.h"

#include "im2col.h"

#include "dark\_cuda.h"

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <float.h>

#include <string.h>

#include <stdint.h>

#ifdef \_WIN32

#include <intrin.h>

#endif

#if defined(\_OPENMP)

#include <omp.h>

#endif

#define TILE\_M 4 // 4 ops

#define TILE\_N 16 // AVX2 = 2 ops \* 8 floats

#define TILE\_K 16 // loop

#ifdef \_\_cplusplus

#define PUT\_IN\_REGISTER

#else

#define PUT\_IN\_REGISTER register

#endif

void gemm\_bin(int M, int N, int K, float ALPHA,

char \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i,j,k;

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

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

char A\_PART = A[i\*lda+k];

if(A\_PART){

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

C[i\*ldc+j] += B[k\*ldb+j];

}

} else {

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

C[i\*ldc+j] -= B[k\*ldb+j];

}

}

}

}

}

float \*random\_matrix(int rows, int cols)

{

int i;

float\* m = (float\*)xcalloc(rows \* cols, sizeof(float));

for(i = 0; i < rows\*cols; ++i){

m[i] = (float)rand()/RAND\_MAX;

}

return m;

}

void time\_random\_matrix(int TA, int TB, int m, int k, int n)

{

float \*a;

if(!TA) a = random\_matrix(m,k);

else a = random\_matrix(k,m);

int lda = (!TA)?k:m;

float \*b;

if(!TB) b = random\_matrix(k,n);

else b = random\_matrix(n,k);

int ldb = (!TB)?n:k;

float \*c = random\_matrix(m,n);

int i;

clock\_t start = clock(), end;

for(i = 0; i<10; ++i){

gemm\_cpu(TA,TB,m,n,k,1,a,lda,b,ldb,1,c,n);

}

end = clock();

printf("Matrix Multiplication %dx%d \* %dx%d, TA=%d, TB=%d: %lf ms\n",m,k,k,n, TA, TB, (float)(end-start)/CLOCKS\_PER\_SEC);

free(a);

free(b);

free(c);

}

void gemm(int TA, int TB, int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float BETA,

float \*C, int ldc)

{

gemm\_cpu( TA, TB, M, N, K, ALPHA,A,lda, B, ldb,BETA,C,ldc);

}

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

// XNOR bitwise GEMM for binary neural network

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

static inline unsigned char xnor(unsigned char a, unsigned char b) {

//return a == b;

return !(a^b);

}

// INT-32

static inline uint32\_t get\_bit\_int32(uint32\_t const\*const src, size\_t index) {

size\_t src\_i = index / 32;

int src\_shift = index % 32;

unsigned char val = (src[src\_i] & (1 << src\_shift)) > 0;

return val;

}

static inline uint32\_t xnor\_int32(uint32\_t a, uint32\_t b) {

return ~(a^b);

}

static inline uint64\_t xnor\_int64(uint64\_t a, uint64\_t b) {

return ~(a^b);

}

static inline uint32\_t fill\_bit\_int32(char src) {

if (src == 0) return 0x00000000;

else return 0xFFFFFFFF;

}

static inline uint64\_t fill\_bit\_int64(char src) {

if (src == 0) return 0x0000000000000000;

else return 0xFFFFFFFFFFFFFFFF;

}

void binary\_int32\_printf(uint32\_t src) {

int i;

for (i = 0; i < 32; ++i) {

if (src & 1) printf("1");

else printf("0");

src = src >> 1;

}

printf("\n");

}

void binary\_int64\_printf(uint64\_t src) {

int i;

for (i = 0; i < 64; ++i) {

if (src & 1) printf("1");

else printf("0");

src = src >> 1;

}

printf("\n");

}

/\*

void gemm\_nn\_custom\_bin\_mean(int M, int N, int K, float ALPHA\_UNUSED,

unsigned char \*A, int lda,

unsigned char \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int \*count\_arr = xcalloc(M\*N, sizeof(int));

int i, j, k;

for (i = 0; i < M; ++i) { // l.n - filters [16 - 55 - 1024]

for (k = 0; k < K; ++k) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

char a\_bit = get\_bit(A, i\*lda + k);

for (j = 0; j < N; ++j) { // out\_h\*out\_w - one channel output size [169 - 173056]

char b\_bit = get\_bit(B, k\*ldb + j);

count\_arr[i\*ldc + j] += xnor(a\_bit, b\_bit);

}

}

}

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

float mean\_val = mean\_arr[i];

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

C[i\*ldc + j] = (2 \* count\_arr[i\*ldc + j] - K) \* mean\_val;

}

}

free(count\_arr);

}

\*/

/\*

void gemm\_nn\_custom\_bin\_mean\_transposed(int M, int N, int K, float ALPHA\_UNUSED,

unsigned char \*A, int lda,

unsigned char \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int \*count\_arr = xcalloc(M\*N, sizeof(int));

int i, j, k;

for (i = 0; i < M; ++i) { // l.n - filters [16 - 55 - 1024]

for (j = 0; j < N; ++j) { // out\_h\*out\_w - one channel output size [169 - 173056]

for (k = 0; k < K; ++k) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

char a\_bit = get\_bit(A, i\*lda + k);

char b\_bit = get\_bit(B, j\*ldb + k);

count\_arr[i\*ldc + j] += xnor(a\_bit, b\_bit);

}

}

}

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

float mean\_val = mean\_arr[i];

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

C[i\*ldc + j] = (2 \* count\_arr[i\*ldc + j] - K) \* mean\_val;

}

}

free(count\_arr);

}

\*/

/\*

void gemm\_nn\_custom\_bin\_mean(int M, int N, int K, float ALPHA\_UNUSED,

unsigned char \*A, int lda,

unsigned char \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int \*count\_arr = xcalloc(M\*N, sizeof(int));

int i;

#pragma omp parallel for

for (i = 0; i < M; ++i) { // l.n - filters [16 - 55 - 1024]

int j, k, h;

for (k = 0; k < K; ++k) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

const char a\_bit = get\_bit(A, i\*lda + k);

uint64\_t a\_bit64 = fill\_bit\_int64(a\_bit);

int k\_ldb = k\*ldb;

for (j = 0; j < N; j += 64) { // out\_h\*out\_w - one channel output size [169 - 173056]

if ((N - j > 64) && (k\_ldb % 8 == 0)) {

uint64\_t b\_bit64 = \*((uint64\_t \*)(B + (k\_ldb + j) / 8));

uint64\_t c\_bit64 = xnor\_int64(a\_bit64, b\_bit64);

//printf("\n %d \n",\_\_builtin\_popcountll(c\_bit64)); // gcc

printf("\n %d \n", \_\_popcnt64(c\_bit64)); // msvs

int h;

for (h = 0; h < 64; ++h)

if ((c\_bit64 >> h) & 1) count\_arr[i\*ldc + j + h] += 1;

//binary\_int64\_printf(a\_bit64);

//binary\_int64\_printf(b\_bit64);

//binary\_int64\_printf(c\_bit64);

}

else {

for (; j < N; ++j) { // out\_h\*out\_w - one channel output size [169 - 173056]

char b\_bit = get\_bit(B, k\_ldb + j);

if (xnor(a\_bit, b\_bit)) count\_arr[i\*ldc + j] += 1;

}

}

}

}

}

if (mean\_arr) {

//int K\_2 = K / 2;

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

float mean\_val = mean\_arr[i];

//float mean\_val2 = 2 \* mean\_val;

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

C[i\*ldc + j] = (2 \* count\_arr[i\*ldc + j] - K) \* mean\_val;

//C[i\*ldc + j] = (count\_arr[i\*ldc + j] - K\_2) \*mean\_val2;

}

}

}

else {

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

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

C[i\*ldc + j] = count\_arr[i\*ldc + j] - K / 2;

}

}

}

free(count\_arr);

//getchar();

}

\*/

/\*

void gemm\_nn\_custom\_bin\_mean\_transposed(int M, int N, int K, float ALPHA\_UNUSED,

unsigned char \*A, int lda,

unsigned char \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int i;

#pragma omp parallel for

for (i = 0; i < M; ++i) { // l.n - filters [16 - 55 - 1024]

int j, k, h;

float mean\_val = mean\_arr[i];

for (j = 0; j < N; ++j) { // out\_h\*out\_w - one channel output size [169 - 173056]

int count = 0;

for (k = 0; k < K; k += 64) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

uint64\_t a\_bit64 = \*((uint64\_t \*)(A + (i\*lda + k) / 8));

uint64\_t b\_bit64 = \*((uint64\_t \*)(B + (j\*ldb + k) / 8));

uint64\_t c\_bit64 = xnor\_int64(a\_bit64, b\_bit64);

#ifdef WIN32

int tmp\_count = \_\_popcnt64(c\_bit64);

#else

int tmp\_count = \_\_builtin\_popcountll(c\_bit64);

#endif

if (K - k < 64) tmp\_count = tmp\_count - (64 - (K - k)); // remove extra bits

count += tmp\_count;

//binary\_int64\_printf(c\_bit64);

//printf(", count = %d \n\n", tmp\_count);

}

C[i\*ldc + j] = (2 \* count - K) \* mean\_val;

}

}

}

\*/

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

// is not used

/\*

void transpose\_32x32\_bits\_my(uint32\_t \*A, uint32\_t \*B, int lda, int ldb)

{

unsigned int x, y;

for (y = 0; y < 32; ++y) {

for (x = 0; x < 32; ++x) {

if (A[y \* lda] & ((uint32\_t)1 << x)) B[x \* ldb] |= (uint32\_t)1 << y;

}

}

}

\*/

#ifndef GPU

uint8\_t reverse\_8\_bit(uint8\_t a) {

return ((a \* 0x0802LU & 0x22110LU) | (a \* 0x8020LU & 0x88440LU)) \* 0x10101LU >> 16;

}

uint32\_t reverse\_32\_bit(uint32\_t a)

{

// unsigned int \_\_rbit(unsigned int val) // for ARM //\_\_asm\_\_("rbit %0, %1\n" : "=r"(output) : "r"(input));

return (reverse\_8\_bit(a >> 24) << 0) |

(reverse\_8\_bit(a >> 16) << 8) |

(reverse\_8\_bit(a >> 8) << 16) |

(reverse\_8\_bit(a >> 0) << 24);

}

#define swap(a0, a1, j, m) t = (a0 ^ (a1 >>j)) & m; a0 = a0 ^ t; a1 = a1 ^ (t << j);

void transpose32\_optimized(uint32\_t A[32]) {

int j, k;

unsigned m, t;

//m = 0x0000FFFF;

//for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j)) {

// for (k = 0; k < 32; k = (k + j + 1) & ~j) {

// t = (A[k] ^ (A[k + j] >> j)) & m;

// A[k] = A[k] ^ t;

// A[k + j] = A[k + j] ^ (t << j);

// }

//}

j = 16;

m = 0x0000FFFF;

for (k = 0; k < 32; k = (k + j + 1) & ~j) { swap(A[k], A[k + j], j, m); }

j = 8;

m = 0x00ff00ff;

for (k = 0; k < 32; k = (k + j + 1) & ~j) { swap(A[k], A[k + j], j, m); }

j = 4;

m = 0x0f0f0f0f;

for (k = 0; k < 32; k = (k + j + 1) & ~j) { swap(A[k], A[k + j], j, m); }

j = 2;

m = 0x33333333;

for (k = 0; k < 32; k = (k + j + 1) & ~j) { swap(A[k], A[k + j], j, m); }

j = 1;

m = 0x55555555;

for (k = 0; k < 32; k = (k + j + 1) & ~j) { swap(A[k], A[k + j], j, m); }

// reverse Y

for (j = 0; j < 16; ++j) {

uint32\_t tmp = A[j];

A[j] = reverse\_32\_bit(A[31 - j]);

A[31 - j] = reverse\_32\_bit(tmp);

}

}

void transpose\_32x32\_bits\_reversed\_diagonale(uint32\_t \*A, uint32\_t \*B, int m, int n)

{

unsigned A\_tmp[32];

int i;

#pragma unroll

for (i = 0; i < 32; ++i) A\_tmp[i] = A[i \* m];

transpose32\_optimized(A\_tmp);

#pragma unroll

for (i = 0; i < 32; ++i) B[i\*n] = A\_tmp[i];

}

void transpose\_8x8\_bits\_my(unsigned char \*A, unsigned char \*B, int lda, int ldb)

{

unsigned x, y;

for (y = 0; y < 8; ++y) {

for (x = 0; x < 8; ++x) {

if (A[y \* lda] & (1 << x)) B[x \* ldb] |= 1 << y;

}

}

}

unsigned char reverse\_byte\_1(char a)

{

return ((a & 0x1) << 7) | ((a & 0x2) << 5) |

((a & 0x4) << 3) | ((a & 0x8) << 1) |

((a & 0x10) >> 1) | ((a & 0x20) >> 3) |

((a & 0x40) >> 5) | ((a & 0x80) >> 7);

}

unsigned char reverse\_byte(unsigned char a)

{

return ((a \* 0x0802LU & 0x22110LU) | (a \* 0x8020LU & 0x88440LU)) \* 0x10101LU >> 16;

}

static unsigned char lookup[16] = {

0x0, 0x8, 0x4, 0xc, 0x2, 0xa, 0x6, 0xe,

0x1, 0x9, 0x5, 0xd, 0x3, 0xb, 0x7, 0xf, };

unsigned char reverse\_byte\_3(unsigned char n) {

// Reverse the top and bottom nibble then swap them.

return (lookup[n & 0b1111] << 4) | lookup[n >> 4];

}

void transpose8rS32\_reversed\_diagonale(unsigned char\* A, unsigned char\* B, int m, int n)

{

unsigned x, y, t;

x = y = 0;

// Load the array and pack it into x and y.

//x = (A[0] << 24) | (A[m] << 16) | (A[2 \* m] << 8) | A[3 \* m];

//y = (A[4 \* m] << 24) | (A[5 \* m] << 16) | (A[6 \* m] << 8) | A[7 \* m];

t = (x ^ (x >> 7)) & 0x00AA00AA; x = x ^ t ^ (t << 7);

t = (y ^ (y >> 7)) & 0x00AA00AA; y = y ^ t ^ (t << 7);

t = (x ^ (x >> 14)) & 0x0000CCCC; x = x ^ t ^ (t << 14);

t = (y ^ (y >> 14)) & 0x0000CCCC; y = y ^ t ^ (t << 14);

t = (x & 0xF0F0F0F0) | ((y >> 4) & 0x0F0F0F0F);

y = ((x << 4) & 0xF0F0F0F0) | (y & 0x0F0F0F0F);

x = t;

B[7 \* n] = reverse\_byte(x >> 24); B[6 \* n] = reverse\_byte(x >> 16); B[5 \* n] = reverse\_byte(x >> 8); B[4 \* n] = reverse\_byte(x);

B[3 \* n] = reverse\_byte(y >> 24); B[2 \* n] = reverse\_byte(y >> 16); B[1 \* n] = reverse\_byte(y >> 8); B[0 \* n] = reverse\_byte(y);

}

/\*

// transpose by 8-bit

void transpose\_bin(char \*A, char \*B, const int n, const int m,

const int lda, const int ldb, const int block\_size)

{

//printf("\n n = %d, ldb = %d \t\t m = %d, lda = %d \n", n, ldb, m, lda);

int i;

#pragma omp parallel for

for (i = 0; i < n; i += 8) {

int j;

for (j = 0; j < m; j += 8) {

int a\_index = i\*lda + j;

int b\_index = j\*ldb + i;

//transpose\_8x8\_bits\_my(&A[a\_index/8], &B[b\_index/8], lda/8, ldb/8);

transpose8rS32\_reversed\_diagonale(&A[a\_index / 8], &B[b\_index / 8], lda / 8, ldb / 8);

}

for (; j < m; ++j) {

if (get\_bit(A, i\*lda + j)) set\_bit(B, j\*ldb + i);

}

}

}

\*/

#endif

// transpose by 32-bit

void transpose\_bin(uint32\_t \*A, uint32\_t \*B, const int n, const int m,

const int lda, const int ldb, const int block\_size)

{

//printf("\n n = %d (n mod 32 = %d), m = %d (m mod 32 = %d) \n", n, n % 32, m, m % 32);

//printf("\n lda = %d (lda mod 32 = %d), ldb = %d (ldb mod 32 = %d) \n", lda, lda % 32, ldb, ldb % 32);

int i;

#pragma omp parallel for

for (i = 0; i < n; i += 32) {

int j;

for (j = 0; j < m; j += 32) {

int a\_index = i\*lda + j;

int b\_index = j\*ldb + i;

transpose\_32x32\_bits\_reversed\_diagonale(&A[a\_index / 32], &B[b\_index / 32], lda / 32, ldb / 32);

//transpose\_32x32\_bits\_my(&A[a\_index/32], &B[b\_index/32], lda/32, ldb/32);

}

for (; j < m; ++j) {

if (get\_bit((const unsigned char\* const)A, i \* lda + j)) set\_bit((unsigned char\* const)B, j \* ldb + i);

}

}

}

static inline int popcnt\_32(uint32\_t val32) {

#ifdef WIN32 // Windows MSVS

int tmp\_count = \_\_popcnt(val32);

#else // Linux GCC

int tmp\_count = \_\_builtin\_popcount(val32);

#endif

return tmp\_count;

}

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

#if (defined(\_\_AVX\_\_) && defined(\_\_x86\_64\_\_)) || (defined(\_WIN64) && !defined(\_\_MINGW32\_\_))

#ifdef \_WIN64

#include <intrin.h>

#include <ammintrin.h>

#include <immintrin.h>

#include <smmintrin.h>

#if defined(\_MSC\_VER) && \_MSC\_VER <= 1900

static inline \_\_int32 \_mm256\_extract\_epi64(\_\_m256i a, const int index) {

return a.m256i\_i64[index];

}

static inline \_\_int32 \_mm256\_extract\_epi32(\_\_m256i a, const int index) {

return a.m256i\_i32[index];

}

#endif

static inline float \_castu32\_f32(uint32\_t a) {

return \*((float \*)&a);

}

static inline float \_mm256\_extract\_float32(\_\_m256 a, const int index) {

return a.m256\_f32[index];

}

#else // Linux GCC/Clang

#include <x86intrin.h>

#include <ammintrin.h>

#include <immintrin.h>

#include <smmintrin.h>

#include <cpuid.h>

static inline float \_castu32\_f32(uint32\_t a) {

return \*((float \*)&a);

}

static inline float \_mm256\_extract\_float32(\_\_m256 a, const int index) {

switch(index) {

case 0:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 0));

case 1:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 1));

case 2:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 2));

case 3:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 3));

case 4:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 4));

case 5:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 5));

case 6:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 6));

case 7:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 7));

default:

return \_castu32\_f32(\_mm256\_extract\_epi32(\_mm256\_castps\_si256(a), 0));

}

}

void asm\_cpuid(uint32\_t\* abcd, uint32\_t eax)

{

uint32\_t ebx = 0, edx = 0, ecx = 0;

// EBX is saved to EDI and later restored

\_\_asm\_\_("movl %%ebx, %%edi;"

"cpuid;"

"xchgl %%ebx, %%edi;"

: "=D"(ebx),

"+a"(eax), "+c"(ecx), "=d"(edx));

abcd[0] = eax;

abcd[1] = ebx;

abcd[2] = ecx;

abcd[3] = edx;

}

#endif

#ifdef \_WIN32

// Windows

#define cpuid(info, x) \_\_cpuidex(info, x, 0)

#else

// GCC Intrinsics

void cpuid(int info[4], int InfoType) {

\_\_cpuid\_count(InfoType, 0, info[0], info[1], info[2], info[3]);

}

#endif

// Misc.

static int HW\_MMX, HW\_x64, HW\_RDRAND, HW\_BMI1, HW\_BMI2, HW\_ADX, HW\_PREFETCHWT1;

static int HW\_ABM; // Advanced Bit Manipulation

// SIMD: 128-bit

static int HW\_SSE, HW\_SSE2, HW\_SSE3, HW\_SSSE3, HW\_SSE41, HW\_SSE42, HW\_SSE4a, HW\_AES, HW\_SHA;

// SIMD: 256-bit

static int HW\_AVX, HW\_XOP, HW\_FMA3, HW\_FMA4, HW\_AVX2;

// SIMD: 512-bit

static int HW\_AVX512F; // AVX512 Foundation

static int HW\_AVX512CD; // AVX512 Conflict Detection

static int HW\_AVX512PF; // AVX512 Prefetch

static int HW\_AVX512ER; // AVX512 Exponential + Reciprocal

static int HW\_AVX512VL; // AVX512 Vector Length Extensions

static int HW\_AVX512BW; // AVX512 Byte + Word

static int HW\_AVX512DQ; // AVX512 Doubleword + Quadword

static int HW\_AVX512IFMA; // AVX512 Integer 52-bit Fused Multiply-Add

static int HW\_AVX512VBMI; // AVX512 Vector Byte Manipulation Instructions

// https://stackoverflow.com/questions/6121792/how-to-check-if-a-cpu-supports-the-sse3-instruction-set

void check\_cpu\_features(void) {

int info[4];

cpuid(info, 0);

int nIds = info[0];

cpuid(info, 0x80000000);

unsigned nExIds = info[0];

// Detect Features

if (nIds >= 0x00000001) {

cpuid(info, 0x00000001);

HW\_MMX = (info[3] & ((uint32\_t)1 << 23)) != 0;

HW\_SSE = (info[3] & ((uint32\_t)1 << 25)) != 0;

HW\_SSE2 = (info[3] & ((uint32\_t)1 << 26)) != 0;

HW\_SSE3 = (info[2] & ((uint32\_t)1 << 0)) != 0;

HW\_SSSE3 = (info[2] & ((uint32\_t)1 << 9)) != 0;

HW\_SSE41 = (info[2] & ((uint32\_t)1 << 19)) != 0;

HW\_SSE42 = (info[2] & ((uint32\_t)1 << 20)) != 0;

HW\_AES = (info[2] & ((uint32\_t)1 << 25)) != 0;

HW\_AVX = (info[2] & ((uint32\_t)1 << 28)) != 0;

HW\_FMA3 = (info[2] & ((uint32\_t)1 << 12)) != 0;

HW\_RDRAND = (info[2] & ((uint32\_t)1 << 30)) != 0;

}

if (nIds >= 0x00000007) {

cpuid(info, 0x00000007);

HW\_AVX2 = (info[1] & ((uint32\_t)1 << 5)) != 0;

HW\_BMI1 = (info[1] & ((uint32\_t)1 << 3)) != 0;

HW\_BMI2 = (info[1] & ((uint32\_t)1 << 8)) != 0;

HW\_ADX = (info[1] & ((uint32\_t)1 << 19)) != 0;

HW\_SHA = (info[1] & ((uint32\_t)1 << 29)) != 0;

HW\_PREFETCHWT1 = (info[2] & ((uint32\_t)1 << 0)) != 0;

HW\_AVX512F = (info[1] & ((uint32\_t)1 << 16)) != 0;

HW\_AVX512CD = (info[1] & ((uint32\_t)1 << 28)) != 0;

HW\_AVX512PF = (info[1] & ((uint32\_t)1 << 26)) != 0;

HW\_AVX512ER = (info[1] & ((uint32\_t)1 << 27)) != 0;

HW\_AVX512VL = (info[1] & ((uint32\_t)1 << 31)) != 0;

HW\_AVX512BW = (info[1] & ((uint32\_t)1 << 30)) != 0;

HW\_AVX512DQ = (info[1] & ((uint32\_t)1 << 17)) != 0;

HW\_AVX512IFMA = (info[1] & ((uint32\_t)1 << 21)) != 0;

HW\_AVX512VBMI = (info[2] & ((uint32\_t)1 << 1)) != 0;

}

if (nExIds >= 0x80000001) {

cpuid(info, 0x80000001);

HW\_x64 = (info[3] & ((uint32\_t)1 << 29)) != 0;

HW\_ABM = (info[2] & ((uint32\_t)1 << 5)) != 0;

HW\_SSE4a = (info[2] & ((uint32\_t)1 << 6)) != 0;

HW\_FMA4 = (info[2] & ((uint32\_t)1 << 16)) != 0;

HW\_XOP = (info[2] & ((uint32\_t)1 << 11)) != 0;

}

}

int is\_avx() {

static int result = -1;

if (result == -1) {

check\_cpu\_features();

result = HW\_AVX;

if (result == 1) printf(" Used AVX \n");

else printf(" Not used AVX \n");

}

return result;

}

int is\_fma\_avx2() {

static int result = -1;

if (result == -1) {

check\_cpu\_features();

result = HW\_FMA3 && HW\_AVX2;

if (result == 1) printf(" Used FMA & AVX2 \n");

else printf(" Not used FMA & AVX2 \n");

}

return result;

}

// https://software.intel.com/sites/landingpage/IntrinsicsGuide

void gemm\_nn(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i, j, k;

if (is\_avx() == 1) { // AVX

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

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

float A\_PART = ALPHA\*A[i\*lda + k];

\_\_m256 a256, b256, c256, result256; // AVX

a256 = \_mm256\_set1\_ps(A\_PART);

for (j = 0; j < N - 8; j += 8) {

b256 = \_mm256\_loadu\_ps(&B[k\*ldb + j]);

c256 = \_mm256\_loadu\_ps(&C[i\*ldc + j]);

// FMA - Intel Haswell (2013), AMD Piledriver (2012)

//result256 = \_mm256\_fmadd\_ps(a256, b256, c256);

result256 = \_mm256\_mul\_ps(a256, b256);

result256 = \_mm256\_add\_ps(result256, c256);

\_mm256\_storeu\_ps(&C[i\*ldc + j], result256);

}

int prev\_end = (N % 8 == 0) ? (N - 8) : (N / 8) \* 8;

for (j = prev\_end; j < N; ++j)

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

}

}

else {

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

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

PUT\_IN\_REGISTER float A\_PART = ALPHA \* A[i \* lda + k];

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

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

/\* // SSE

\_\_m128 a128, b128, c128, result128; // SSE

a128 = \_mm\_set1\_ps(A\_PART);

for (j = 0; j < N - 4; j += 4) {

b128 = \_mm\_loadu\_ps(&B[k\*ldb + j]);

c128 = \_mm\_loadu\_ps(&C[i\*ldc + j]);

//result128 = \_mm\_fmadd\_ps(a128, b128, c128);

result128 = \_mm\_mul\_ps(a128, b128);

result128 = \_mm\_add\_ps(result128, c128);

\_mm\_storeu\_ps(&C[i\*ldc + j], result128);

}

int prev\_end = (N % 4 == 0) ? (N - 4) : (N / 4) \* 4;

for (j = prev\_end; j < N; ++j){

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

\*/

}

}

}

}

void gemm\_nn\_fast(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i;

#pragma omp parallel for

for (i = 0; i < (M / TILE\_M)\*TILE\_M; i += TILE\_M)

{

int j, k;

int i\_d, k\_d;

for (k = 0; k < (K / TILE\_K)\*TILE\_K; k += TILE\_K)

{

for (j = 0; j < (N / TILE\_N)\*TILE\_N; j += TILE\_N)

{

// L1 - 6 bits tag [11:6] - cache size 32 KB, conflict for each 4 KB

// L2 - 9 bits tag [14:6] - cache size 256 KB, conflict for each 32 KB

// L3 - 13 bits tag [18:6] - cache size 8 MB, conflict for each 512 KB

\_\_m256 result256;

\_\_m256 a256\_0, b256\_0; // AVX

\_\_m256 a256\_1, b256\_1; // AVX

\_\_m256 a256\_2;// , b256\_2; // AVX

\_\_m256 a256\_3;// , b256\_3; // AVX

\_\_m256 c256\_0, c256\_1, c256\_2, c256\_3;

\_\_m256 c256\_4, c256\_5, c256\_6, c256\_7;

c256\_0 = \_mm256\_loadu\_ps(&C[(0 + i)\*ldc + (0 + j)]);

c256\_1 = \_mm256\_loadu\_ps(&C[(1 + i)\*ldc + (0 + j)]);

c256\_2 = \_mm256\_loadu\_ps(&C[(0 + i)\*ldc + (8 + j)]);

c256\_3 = \_mm256\_loadu\_ps(&C[(1 + i)\*ldc + (8 + j)]);

c256\_4 = \_mm256\_loadu\_ps(&C[(2 + i)\*ldc + (0 + j)]);

c256\_5 = \_mm256\_loadu\_ps(&C[(3 + i)\*ldc + (0 + j)]);

c256\_6 = \_mm256\_loadu\_ps(&C[(2 + i)\*ldc + (8 + j)]);

c256\_7 = \_mm256\_loadu\_ps(&C[(3 + i)\*ldc + (8 + j)]);

for (k\_d = 0; k\_d < (TILE\_K); ++k\_d)

{

a256\_0 = \_mm256\_set1\_ps(ALPHA\*A[(0 + i)\*lda + (k\_d + k)]);

a256\_1 = \_mm256\_set1\_ps(ALPHA\*A[(1 + i)\*lda + (k\_d + k)]);

a256\_2 = \_mm256\_set1\_ps(ALPHA\*A[(2 + i)\*lda + (k\_d + k)]);

a256\_3 = \_mm256\_set1\_ps(ALPHA\*A[(3 + i)\*lda + (k\_d + k)]);

b256\_0 = \_mm256\_loadu\_ps(&B[(k\_d + k)\*ldb + (0 + j)]);

b256\_1 = \_mm256\_loadu\_ps(&B[(k\_d + k)\*ldb + (8 + j)]);

// FMA - Intel Haswell (2013), AMD Piledriver (2012)

//c256\_0 = \_mm256\_fmadd\_ps(a256\_0, b256\_0, c256\_0);

//c256\_1 = \_mm256\_fmadd\_ps(a256\_1, b256\_0, c256\_1);

//c256\_2 = \_mm256\_fmadd\_ps(a256\_0, b256\_1, c256\_2);

//c256\_3 = \_mm256\_fmadd\_ps(a256\_1, b256\_1, c256\_3);

//c256\_4 = \_mm256\_fmadd\_ps(a256\_2, b256\_0, c256\_4);

//c256\_5 = \_mm256\_fmadd\_ps(a256\_3, b256\_0, c256\_5);

//c256\_6 = \_mm256\_fmadd\_ps(a256\_2, b256\_1, c256\_6);

//c256\_7 = \_mm256\_fmadd\_ps(a256\_3, b256\_1, c256\_7);

result256 = \_mm256\_mul\_ps(a256\_0, b256\_0);

c256\_0 = \_mm256\_add\_ps(result256, c256\_0);

result256 = \_mm256\_mul\_ps(a256\_1, b256\_0);

c256\_1 = \_mm256\_add\_ps(result256, c256\_1);

result256 = \_mm256\_mul\_ps(a256\_0, b256\_1);

c256\_2 = \_mm256\_add\_ps(result256, c256\_2);

result256 = \_mm256\_mul\_ps(a256\_1, b256\_1);

c256\_3 = \_mm256\_add\_ps(result256, c256\_3);

result256 = \_mm256\_mul\_ps(a256\_2, b256\_0);

c256\_4 = \_mm256\_add\_ps(result256, c256\_4);

result256 = \_mm256\_mul\_ps(a256\_3, b256\_0);

c256\_5 = \_mm256\_add\_ps(result256, c256\_5);

result256 = \_mm256\_mul\_ps(a256\_2, b256\_1);

c256\_6 = \_mm256\_add\_ps(result256, c256\_6);

result256 = \_mm256\_mul\_ps(a256\_3, b256\_1);

c256\_7 = \_mm256\_add\_ps(result256, c256\_7);

}

\_mm256\_storeu\_ps(&C[(0 + i)\*ldc + (0 + j)], c256\_0);

\_mm256\_storeu\_ps(&C[(1 + i)\*ldc + (0 + j)], c256\_1);

\_mm256\_storeu\_ps(&C[(0 + i)\*ldc + (8 + j)], c256\_2);

\_mm256\_storeu\_ps(&C[(1 + i)\*ldc + (8 + j)], c256\_3);

\_mm256\_storeu\_ps(&C[(2 + i)\*ldc + (0 + j)], c256\_4);

\_mm256\_storeu\_ps(&C[(3 + i)\*ldc + (0 + j)], c256\_5);

\_mm256\_storeu\_ps(&C[(2 + i)\*ldc + (8 + j)], c256\_6);

\_mm256\_storeu\_ps(&C[(3 + i)\*ldc + (8 + j)], c256\_7);

}

for (j = (N / TILE\_N)\*TILE\_N; j < N; ++j) {

for (i\_d = i; i\_d < (i + TILE\_M); ++i\_d)

{

for (k\_d = k; k\_d < (k + TILE\_K); ++k\_d)

{

PUT\_IN\_REGISTER float A\_PART = ALPHA\*A[i\_d\*lda + k\_d];

C[i\_d\*ldc + j] += A\_PART\*B[k\_d\*ldb + j];

}

}

}

}

for (k = (K / TILE\_K)\*TILE\_K; k < K; ++k)

{

for (i\_d = i; i\_d < (i + TILE\_M); ++i\_d)

{

PUT\_IN\_REGISTER float A\_PART = ALPHA\*A[i\_d\*lda + k];

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

C[i\_d\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

}

}

}

for (i = (M / TILE\_M)\*TILE\_M; i < M; ++i) {

int j, k;

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

PUT\_IN\_REGISTER float A\_PART = ALPHA\*A[i\*lda + k];

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

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

}

}

}

void gemm\_nn\_bin\_32bit\_packed(int M, int N, int K, float ALPHA,

uint32\_t \*A, int lda,

uint32\_t \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int i;

#pragma omp parallel for

for (i = 0; i < M; ++i) { // l.n

int j, s;

float mean\_val = mean\_arr[i];

//printf(" l.mean\_arr[i] = %d \n ", l.mean\_arr[i]);

for (s = 0; s < K; ++s) // l.size\*l.size\*l.c/32 or (l.size\*l.size\*l.c)

{

PUT\_IN\_REGISTER uint32\_t A\_PART = A[i\*lda + s];

\_\_m256i a256 = \_mm256\_set1\_epi32(A\_PART);

for (j = 0; j < N - 8; j += 8)

{

\_\_m256i b256 = \*((\_\_m256i\*)&B[s\*ldb + j]);

\_\_m256i xor256 = \_mm256\_xor\_si256(a256, b256); // xnor = xor(a,b)

\_\_m256i all\_1 = \_mm256\_set1\_epi8((char)255);

\_\_m256i xnor256 = \_mm256\_andnot\_si256(xor256, all\_1); // xnor = not(xor(a,b))

// waiting for - CPUID Flags: AVX512VPOPCNTDQ: \_\_m512i \_mm512\_popcnt\_epi32(\_\_m512i a)

\_\_m256 count = \_mm256\_setr\_ps(

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 0)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 1)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 2)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 3)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 4)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 5)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 6)),

popcnt\_32(\_mm256\_extract\_epi32(xnor256, 7)));

\_\_m256 val2 = \_mm256\_set1\_ps(2);

count = \_mm256\_mul\_ps(count, val2); // count \* 2

\_\_m256 val32 = \_mm256\_set1\_ps(32);

count = \_mm256\_sub\_ps(count, val32); // count - 32

\_\_m256 mean256 = \_mm256\_set1\_ps(mean\_val);

count = \_mm256\_mul\_ps(count, mean256); // count \* mean\_val

\_\_m256 c256 = \*((\_\_m256\*)&C[i\*ldc + j]);

count = \_mm256\_add\_ps(count, c256); // c = c + count

\*((\_\_m256\*)&C[i\*ldc + j]) = count;

}

for (; j < N; ++j) // out\_h\*out\_w;

{

PUT\_IN\_REGISTER uint32\_t B\_PART = B[s\*ldb + j];

uint32\_t xnor\_result = ~(A\_PART ^ B\_PART);

int32\_t count = popcnt\_32(xnor\_result); // must be Signed int

C[i\*ldc + j] += (2 \* count - 32) \* mean\_val;

}

}

}

}

void convolution\_2d\_old(int w, int h, int ksize, int n, int c, int pad, int stride,

float \*weights, float \*input, float \*output)

{

//const int out\_h = (h + 2 \* pad - ksize) / stride + 1; // output\_height=input\_height for stride=1 and pad=1

//const int out\_w = (w + 2 \* pad - ksize) / stride + 1; // output\_width=input\_width for stride=1 and pad=1

int fil;

// filter index

#pragma omp parallel for // "omp parallel for" - automatic parallelization of loop by using OpenMP

for (fil = 0; fil < n; ++fil) {

//int i, f, j;

int chan, y, x, f\_y, f\_x;

// channel index

for (chan = 0; chan < c; ++chan)

// input - y

for (y = 0; y < h; ++y)

// input - x

for (x = 0; x < w; ++x)

{

int const output\_index = fil\*w\*h + y\*w + x;

int const weights\_pre\_index = fil\*c\*ksize\*ksize + chan\*ksize\*ksize;

int const input\_pre\_index = chan\*w\*h;

float sum = 0;

// filter - y

for (f\_y = 0; f\_y < ksize; ++f\_y)

{

int input\_y = y + f\_y - pad;

// filter - x

for (f\_x = 0; f\_x < ksize; ++f\_x)

{

int input\_x = x + f\_x - pad;

if (input\_y < 0 || input\_x < 0 || input\_y >= h || input\_x >= w) continue;

int input\_index = input\_pre\_index + input\_y\*w + input\_x;

int weights\_index = weights\_pre\_index + f\_y\*ksize + f\_x;

sum += input[input\_index] \* weights[weights\_index];

}

}

// l.output[filters][width][height] +=

// state.input[channels][width][height] \*

// l.weights[filters][channels][filter\_width][filter\_height];

output[output\_index] += sum;

}

}

}

void convolution\_2d(int w, int h, int ksize, int n, int c, int pad, int stride,

float \*weights, float \*input, float \*output, float \*mean)

{

//const int out\_h = (h + 2 \* pad - ksize) / stride + 1; // output\_height=input\_height for stride=1 and pad=1

//const int out\_w = (w + 2 \* pad - ksize) / stride + 1; // output\_width=input\_width for stride=1 and pad=1

int i;

#if defined(\_OPENMP)

static int max\_num\_threads = 0;

if (max\_num\_threads == 0) {

max\_num\_threads = omp\_get\_max\_threads();

//omp\_set\_num\_threads( max\_num\_threads / 2);

}

#endif

//convolution\_2d\_old(w, h, ksize, n, c, pad, stride, weights, input, output);

\_\_m256i all256\_sing1 = \_mm256\_set\_epi32(0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000);

for (i = 0; i < ksize\*ksize\*n\*c; i+=8) {

\*((\_\_m256\*)&weights[i]) = \_mm256\_and\_ps(\*((\_\_m256\*)&weights[i]), \_mm256\_castsi256\_ps(all256\_sing1));

}

//for (i = 0; i < w\*h\*c; i += 8) {

//\*((\_\_m256\*)&input[i]) = \_mm256\_and\_ps(\*((\_\_m256\*)&input[i]), \_mm256\_castsi256\_ps(all256\_sing1));

//}

//\_\_m256i all256\_last\_zero = \_mm256\_set1\_epi32(0xFFFFFFFF);

//all256\_last\_zero.m256i\_i32[7] = 0;

\_\_m256i all256\_last\_zero =

\_mm256\_set\_epi32(0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0x0);

\_\_m256i idx256 = \_mm256\_set\_epi32(0, 7, 6, 5, 4, 3, 2, 1);

//\_\_m256 all256\_sing1 = \_mm256\_set1\_ps(0x80000000);

\_\_m256 all256\_one = \_mm256\_set1\_ps(1);

\_\_m256i all256i\_one = \_mm256\_set1\_epi32(1);

///\_\_m256i src256 = \_mm256\_loadu\_si256((\_\_m256i \*)(&src[i]));

///\_\_m256i result256 = \_mm256\_and\_si256(src256, all256\_sing1); // check sign in 8 x 32-bit floats

int fil;

// filter index

#pragma omp parallel for // "omp parallel for" - automatic parallelization of loop by using OpenMP

for (fil = 0; fil < n; ++fil) {

int chan, y, x, f\_y, f\_x;

float cur\_mean = fabs(mean[fil]);

\_\_m256 mean256 = \_mm256\_set1\_ps(cur\_mean);

// channel index

//for (chan = 0; chan < c; ++chan)

// input - y

for (y = 0; y < h; ++y)

// input - x

for (x = 0; x < w-8; x+=8)

{

int const output\_index = fil\*w\*h + y\*w + x;

float sum = 0;

\_\_m256 sum256 = \_mm256\_set1\_ps(0);

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

int const weights\_pre\_index = fil\*c\*ksize\*ksize + chan\*ksize\*ksize;

int const input\_pre\_index = chan\*w\*h;

// filter - y

for (f\_y = 0; f\_y < ksize; ++f\_y)

{

int input\_y = y + f\_y - pad;

//\_\_m256 in = \*((\_\_m256\*)&input[input\_pre\_index + input\_y\*w]);

if (input\_y < 0 || input\_y >= h) continue;

//\_\_m256 in = \_mm256\_loadu\_ps(&input[input\_pre\_index + input\_y\*w + x - pad]);

// filter - x

for (f\_x = 0; f\_x < ksize; ++f\_x)

{

int input\_x = x + f\_x - pad;

//if (input\_y < 0 || input\_x < 0 || input\_y >= h || input\_x >= w) continue;

int input\_index = input\_pre\_index + input\_y\*w + input\_x;

int weights\_index = weights\_pre\_index + f\_y\*ksize + f\_x;

//if (input\_y < 0 || input\_y >= h) continue;

//sum += input[input\_index] \* weights[weights\_index];

\_\_m256 in = \*((\_\_m256\*)&input[input\_index]);

\_\_m256 w = \_mm256\_set1\_ps(weights[weights\_index]);

//\_\_m256 w\_sign = \_mm256\_and\_ps(w, \_mm256\_castsi256\_ps(all256\_sing1)); // check sign in 8 x 32-bit floats

\_\_m256 xor256 = \_mm256\_xor\_ps(w, in);

//printf("\n xor256\_1 = %f, xor256\_2 = %f \n", xor256.m256\_f32[0], xor256.m256\_f32[1]);

//printf("\n in = %f, w = %f, xor256 = %f \n", in.m256\_f32[0], w\_sign.m256\_f32[0], xor256.m256\_f32[0]);

//\_\_m256 pn1 = \_mm256\_and\_ps(\_mm256\_castsi256\_ps(all256i\_one), xor256);

//sum256 = xor256;

sum256 = \_mm256\_add\_ps(xor256, sum256);

//printf("\n --- \n");

//printf("\n 0 = %f, 1 = %f, 2 = %f, 3 = %f, 4 = %f, 5 = %f, 6 = %f, 7 = %f \n", in.m256\_f32[0], in.m256\_f32[1], in.m256\_f32[2], in.m256\_f32[3], in.m256\_f32[4], in.m256\_f32[5], in.m256\_f32[6], in.m256\_f32[7]);

if (f\_x < ksize-1) {

//in = \_mm256\_permutevar8x32\_ps(in, idx256);

//in = \_mm256\_and\_ps(in, \_mm256\_castsi256\_ps(all256\_last\_zero));

}

}

}

}

// l.output[filters][width][height] +=

// state.input[channels][width][height] \*

// l.weights[filters][channels][filter\_width][filter\_height];

//output[output\_index] += sum;

sum256 = \_mm256\_mul\_ps(sum256, mean256);

//printf("\n cur\_mean = %f, sum256 = %f, sum256 = %f, in = %f \n",

// cur\_mean, sum256.m256\_f32[0], sum256.m256\_f32[1], input[input\_pre\_index]);

//\_\_m256 out = \*((\_\_m256\*)&output[output\_index]);

//out = \_mm256\_add\_ps(out, sum256);

//\*((\_\_m256\*)&output[output\_index]) = out;

\*((\_\_m256\*)&output[output\_index]) = sum256;

//\_mm256\_storeu\_ps(&C[i\*ldc + j], result256);

}

}

}

// http://graphics.stanford.edu/~seander/bithacks.html

// https://stackoverflow.com/questions/17354971/fast-counting-the-number-of-set-bits-in-m128i-register

// https://arxiv.org/pdf/1611.07612.pdf

static inline int popcnt128(\_\_m128i n) {

const \_\_m128i n\_hi = \_mm\_unpackhi\_epi64(n, n);

#if defined(\_MSC\_VER)

return \_\_popcnt64(\_mm\_cvtsi128\_si64(n)) + \_\_popcnt64(\_mm\_cvtsi128\_si64(n\_hi));

#elif defined(\_\_APPLE\_\_) && defined(\_\_clang\_\_)

return \_mm\_popcnt\_u64(\_mm\_cvtsi128\_si64(n)) + \_mm\_popcnt\_u64(\_mm\_cvtsi128\_si64(n\_hi));

#else

return \_\_popcntq(\_mm\_cvtsi128\_si64(n)) + \_\_popcntq(\_mm\_cvtsi128\_si64(n\_hi));

#endif

}

static inline int popcnt256(\_\_m256i n) {

return popcnt128(\_mm256\_extractf128\_si256(n, 0)) + popcnt128(\_mm256\_extractf128\_si256(n, 1));

}

static inline \_\_m256i count256(\_\_m256i v) {

\_\_m256i lookup =

\_mm256\_setr\_epi8(0, 1, 1, 2, 1, 2, 2, 3, 1, 2,

2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2, 3,

1, 2, 2, 3, 2, 3, 3, 4);

\_\_m256i low\_mask = \_mm256\_set1\_epi8(0x0f);

\_\_m256i lo = \_mm256\_and\_si256(v, low\_mask);

\_\_m256i hi = \_mm256\_and\_si256(\_mm256\_srli\_epi32(v, 4), low\_mask);

\_\_m256i popcnt1 = \_mm256\_shuffle\_epi8(lookup, lo);

\_\_m256i popcnt2 = \_mm256\_shuffle\_epi8(lookup, hi);

\_\_m256i total = \_mm256\_add\_epi8(popcnt1, popcnt2);

return \_mm256\_sad\_epu8(total, \_mm256\_setzero\_si256());

}

static inline int popcnt256\_custom(\_\_m256i n) {

\_\_m256i val = count256(n);

//return val.m256i\_i64[0] +

//val.m256i\_i64[1] +

//val.m256i\_i64[2] +

//val.m256i\_i64[3];

return \_mm256\_extract\_epi64(val, 0)

+ \_mm256\_extract\_epi64(val, 1)

+ \_mm256\_extract\_epi64(val, 2)

+ \_mm256\_extract\_epi64(val, 3);

}

static inline void xnor\_avx2\_popcnt(\_\_m256i a\_bit256, \_\_m256i b\_bit256, \_\_m256i \*count\_sum) {

\_\_m256i c\_bit256 = \_mm256\_set1\_epi8((char)255);

\_\_m256i xor256 = \_mm256\_xor\_si256(a\_bit256, b\_bit256); // xnor = not(xor(a,b))

c\_bit256 = \_mm256\_andnot\_si256(xor256, c\_bit256); // can be optimized - we can do other NOT for wegihts once and do not do this NOT

\*count\_sum = \_mm256\_add\_epi64(count256(c\_bit256), \*count\_sum); // 1st part - popcnt Mula's algorithm

}

// 2nd part - popcnt Mula's algorithm

static inline int get\_count\_mula(\_\_m256i count\_sum) {

return \_mm256\_extract\_epi64(count\_sum, 0)

+ \_mm256\_extract\_epi64(count\_sum, 1)

+ \_mm256\_extract\_epi64(count\_sum, 2)

+ \_mm256\_extract\_epi64(count\_sum, 3);

}

// 5x times faster than gemm()-float32

// further optimizations: do mean-mult only for the last layer

void gemm\_nn\_custom\_bin\_mean\_transposed(int M, int N, int K, float ALPHA\_UNUSED,

unsigned char \*A, int lda,

unsigned char \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int i;

#if defined(\_OPENMP)

static int max\_num\_threads = 0;

if (max\_num\_threads == 0) {

max\_num\_threads = omp\_get\_max\_threads();

//omp\_set\_num\_threads(max\_num\_threads / 2);

}

#endif

//#pragma omp parallel for

//for (i = 0; i < M; ++i)

#pragma omp parallel for

for (i = 0; i < (M/2)\*2; i += 2)

{ // l.n - filters [16 - 55 - 1024]

float mean\_val\_0 = mean\_arr[i + 0];

float mean\_val\_1 = mean\_arr[i + 1];

int j, k;

//\_\_m256i all\_1 = \_mm256\_set1\_epi8(255);

//for (j = 0; j < N; ++j)

for (j = 0; j < (N/2)\*2; j += 2)

{ // out\_h\*out\_w - one channel output size [169 - 173056]

//int count = 0;

const int bit\_step = 256;

\_\_m256i count\_sum\_0 = \_mm256\_set1\_epi8(0);

\_\_m256i count\_sum\_1 = \_mm256\_set1\_epi8(0);

\_\_m256i count\_sum\_2 = \_mm256\_set1\_epi8(0);

\_\_m256i count\_sum\_3 = \_mm256\_set1\_epi8(0);

for (k = 0; k < K; k += bit\_step) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

\_\_m256i a\_bit256\_0 = \_mm256\_loadu\_si256((\_\_m256i \*)(A + ((i + 0)\*lda + k) / 8));

\_\_m256i b\_bit256\_0 = \_mm256\_loadu\_si256((\_\_m256i \*)(B + ((j + 0)\*ldb + k) / 8));

\_\_m256i a\_bit256\_1 = \_mm256\_loadu\_si256((\_\_m256i \*)(A + ((i + 1)\*lda + k) / 8));

\_\_m256i b\_bit256\_1 = \_mm256\_loadu\_si256((\_\_m256i \*)(B + ((j + 1)\*ldb + k) / 8));

xnor\_avx2\_popcnt(a\_bit256\_0, b\_bit256\_0, &count\_sum\_0);

xnor\_avx2\_popcnt(a\_bit256\_0, b\_bit256\_1, &count\_sum\_1);

xnor\_avx2\_popcnt(a\_bit256\_1, b\_bit256\_0, &count\_sum\_2);

xnor\_avx2\_popcnt(a\_bit256\_1, b\_bit256\_1, &count\_sum\_3);

//count += popcnt256(c\_bit256);

//binary\_int64\_printf(c\_bit64);

//printf(", count = %d \n\n", tmp\_count);

}

int count\_0 = get\_count\_mula(count\_sum\_0);

int count\_1 = get\_count\_mula(count\_sum\_1);

int count\_2 = get\_count\_mula(count\_sum\_2);

int count\_3 = get\_count\_mula(count\_sum\_3);

const int f1 = (K % bit\_step == 0) ? 0 : (bit\_step - (K % bit\_step));

count\_0 = count\_0 - f1; // remove extra bits (from empty space for align only)

count\_1 = count\_1 - f1;

count\_2 = count\_2 - f1;

count\_3 = count\_3 - f1;

C[i\*ldc + (j + 0)] = (2 \* count\_0 - K) \* mean\_val\_0;

C[i\*ldc + (j + 1)] = (2 \* count\_1 - K) \* mean\_val\_0;

C[(i + 1)\*ldc + (j + 0)] = (2 \* count\_2 - K) \* mean\_val\_1;

C[(i + 1)\*ldc + (j + 1)] = (2 \* count\_3 - K) \* mean\_val\_1;

}

int i\_d;

for (i\_d = 0; i\_d < 2; ++i\_d)

{

float mean\_val = mean\_arr[i + i\_d];

for (j = (N / 2) \* 2; j < N; j += 1)

{ // out\_h\*out\_w - one channel output size [169 - 173056]

const int bit\_step = 256;

\_\_m256i count\_sum = \_mm256\_set1\_epi8(0);

for (k = 0; k < K; k += bit\_step) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

\_\_m256i a\_bit256\_0 = \_mm256\_loadu\_si256((\_\_m256i \*)(A + ((i + i\_d + 0)\*lda + k) / 8));

\_\_m256i b\_bit256\_0 = \_mm256\_loadu\_si256((\_\_m256i \*)(B + ((j + 0)\*ldb + k) / 8));

xnor\_avx2\_popcnt(a\_bit256\_0, b\_bit256\_0, &count\_sum);

}

int count = get\_count\_mula(count\_sum);

const int f1 = (K % bit\_step == 0) ? 0 : (bit\_step - (K % bit\_step));

count = count - f1; // remove extra bits (from empty space for align only)

C[(i + i\_d)\*ldc + j] = (2 \* count - K) \* mean\_val;

}

}

}

for (i = (M / 2) \* 2; i < M; i += 1)

{

float mean\_val = mean\_arr[i];

int j, k;

for (j = 0; j < N; j += 1)

{ // out\_h\*out\_w - one channel output size [169 - 173056]

const int bit\_step = 256;

\_\_m256i count\_sum = \_mm256\_set1\_epi8(0);

for (k = 0; k < K; k += bit\_step) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

\_\_m256i a\_bit256\_0 = \_mm256\_loadu\_si256((\_\_m256i \*)(A + ((i + 0)\*lda + k) / 8));

\_\_m256i b\_bit256\_0 = \_mm256\_loadu\_si256((\_\_m256i \*)(B + ((j + 0)\*ldb + k) / 8));

xnor\_avx2\_popcnt(a\_bit256\_0, b\_bit256\_0, &count\_sum);

}

int count = get\_count\_mula(count\_sum);

const int f1 = (K % bit\_step == 0) ? 0 : (bit\_step - (K % bit\_step));

count = count - f1; // remove extra bits (from empty space for align only)

C[i\*ldc + j] = (2 \* count - K) \* mean\_val;

}

}

}

//From Berkeley Vision's Caffe!

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

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

int channels, int height, int width,

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

{

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

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

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

int c;

// optimized version

if (height\_col == height && width\_col == width && stride == 1 && pad == 1)

{

#pragma omp parallel for

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

int h, w;

int w\_offset = c % ksize;

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

int c\_im = c / ksize / ksize;

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

for (w = pad; w < width\_col - pad - 4; w+=8) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

//data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

\_\_m256 src256 = \_mm256\_loadu\_ps((float \*)(&data\_im[im\_col + width\*(im\_row + height\*c\_im)]));

data\_col[col\_index + ldb\_align \* 0] = \_mm256\_extract\_float32(src256, 0);// src256.m256\_f32[0];

data\_col[col\_index + ldb\_align \* 1] = \_mm256\_extract\_float32(src256, 1);// src256.m256\_f32[1];

data\_col[col\_index + ldb\_align \* 2] = \_mm256\_extract\_float32(src256, 2);// src256.m256\_f32[2];

data\_col[col\_index + ldb\_align \* 3] = \_mm256\_extract\_float32(src256, 3);// src256.m256\_f32[3];

data\_col[col\_index + ldb\_align \* 4] = \_mm256\_extract\_float32(src256, 4);// src256.m256\_f32[4];

data\_col[col\_index + ldb\_align \* 5] = \_mm256\_extract\_float32(src256, 5);// src256.m256\_f32[5];

data\_col[col\_index + ldb\_align \* 6] = \_mm256\_extract\_float32(src256, 6);// src256.m256\_f32[6];

data\_col[col\_index + ldb\_align \* 7] = \_mm256\_extract\_float32(src256, 7);// src256.m256\_f32[7];

//\_mm256\_storeu\_ps(&data\_col[col\_index], src256);

}

for (; w < width\_col - pad; ++w) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

int col\_index = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

}

}

{

w = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

int col\_index = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

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

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

}

}

{

w = width\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

int col\_index = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

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

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

}

}

{

h = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

int col\_index = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

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

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

}

}

{

h = height\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

int col\_index = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

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

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

}

}

}

}

else {

#pragma omp parallel for

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

int h, w;

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 = (h \* width\_col + w)\*ldb\_align + c; // transposed & aligned

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

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

}

}

}

}

}

//From Berkeley Vision's Caffe!

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

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

int channels, int height, int width,

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

{

int c;

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

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

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

// optimized version

if (height\_col == height && width\_col == width && stride == 1 && pad == 1 && is\_fma\_avx2())

{

#pragma omp parallel for

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

int h, w;

int w\_offset = c % ksize;

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

int c\_im = c / ksize / ksize;

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

for (w = pad; w < width\_col-pad-8; w += 8) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

//data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

\_\_m256 src256 = \_mm256\_loadu\_ps((float \*)(&data\_im[im\_col + width\*(im\_row + height\*c\_im)]));

\_mm256\_storeu\_ps(&data\_col[col\_index], src256);

}

for (; w < width\_col - pad; ++w) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

}

}

{

w = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

{

w = width\_col-1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

{

h = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

{

h = height\_col-1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

}

}

else {

//printf("\n Error: is no non-optimized version \n");

im2col\_cpu(data\_im, channels, height, width, ksize, stride, pad, data\_col);

}

}

//From Berkeley Vision's Caffe!

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

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

int channels, int height, int width,

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

{

int c;

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

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

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

// optimized version

if (height\_col == height && width\_col == width && stride == 1 && pad == 1 && is\_fma\_avx2())

{

int new\_ldb = bit\_align;

#pragma omp parallel for

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

int h, w;

int w\_offset = c % ksize;

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

int c\_im = c / ksize / ksize;

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

for (w = pad; w < width\_col - pad - 8; w += 8) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

\_\_m256 src256 = \_mm256\_loadu\_ps((float \*)(&data\_im[im\_col + width\*(im\_row + height\*c\_im)]));

\_mm256\_storeu\_ps(&data\_col[col\_index], src256);

}

for (; w < width\_col - pad; ++w) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

}

}

{

w = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

}

}

{

w = width\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

}

}

{

h = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

}

}

{

h = height\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

}

}

}

}

else {

printf("\n Error: is no non-optimized version \n");

//im2col\_cpu(data\_im, channels, height, width, ksize, stride, pad, data\_col); // must be aligned for transpose after float\_to\_bin

// float\_to\_bit(b, t\_input, src\_size);

// transpose\_bin(t\_input, \*t\_bit\_input, k, n, bit\_align, new\_ldb, 8);

}

}

//From Berkeley Vision's Caffe!

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

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

int channels, int height, int width,

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

{

int c;

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

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

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

// optimized version

if (height\_col == height && width\_col == width && stride == 1 && pad == 1 && is\_fma\_avx2())

{

\_\_m256i all256\_sing1 = \_mm256\_set\_epi32(0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000);

\_\_m256 float\_zero256 = \_mm256\_set1\_ps(0.00);

int new\_ldb = bit\_align;

#pragma omp parallel for

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

int h, w;

int w\_offset = c % ksize;

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

int c\_im = c / ksize / ksize;

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

for (w = pad; w < width\_col - pad - 8; w += 8) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//\_\_m256i src256 = \_mm256\_loadu\_si256((\_\_m256i \*)(&data\_im[im\_col + width\*(im\_row + height\*c\_im)]));

//\_\_m256i result256 = \_mm256\_and\_si256(src256, all256\_sing1); // check sign in 8 x 32-bit floats

//uint16\_t mask = \_mm256\_movemask\_ps(\_mm256\_castsi256\_ps(result256)); // (val >= 0) ? 0 : 1

//mask = ~mask; // inverse mask, (val >= 0) ? 1 : 0

\_\_m256 src256 = \_mm256\_loadu\_ps((float \*)(&data\_im[im\_col + width\*(im\_row + height\*c\_im)]));

\_\_m256 result256 = \_mm256\_cmp\_ps(src256, float\_zero256, \_CMP\_GT\_OS);

uint16\_t mask = \_mm256\_movemask\_ps(result256); // (val > 0) ? 0 : 1

uint16\_t\* dst\_ptr = (uint16\_t\*)&((uint8\_t\*)data\_col)[col\_index / 8];

\*dst\_ptr |= (mask << (col\_index % 8));

}

for (; w < width\_col - pad; ++w) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

float val = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

if (val > 0) set\_bit((unsigned char\* const)data\_col, col\_index);

}

}

{

w = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\* const)data\_col, col\_index);

}

}

{

w = width\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\* const)data\_col, col\_index);

}

}

{

h = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\* const)data\_col, col\_index);

}

}

{

h = height\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\* const)data\_col, col\_index);

}

}

}

}

else {

printf("\n Error: is no non-optimized version \n");

//im2col\_cpu(data\_im, channels, height, width, ksize, stride, pad, data\_col); // must be aligned for transpose after float\_to\_bin

// float\_to\_bit(b, t\_input, src\_size);

// transpose\_bin(t\_input, \*t\_bit\_input, k, n, bit\_align, new\_ldb, 8);

}

}

void activate\_array\_cpu\_custom(float \*x, const int n, const ACTIVATION a)

{

int i = 0;

if (a == LINEAR)

{}

else if (a == LEAKY)

{

if (is\_fma\_avx2()) {

\_\_m256i all256\_sing1 = \_mm256\_set\_epi32(0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000);

\_\_m256 all256\_01 = \_mm256\_set1\_ps(0.1F);

for (i = 0; i < n - 8; i += 8) {

//x[i] = (x[i]>0) ? x[i] : .1\*x[i];

\_\_m256 src256 = \_mm256\_loadu\_ps(&x[i]);

\_\_m256 mult256 = \_mm256\_mul\_ps((src256), all256\_01); // mult \* 0.1

\_\_m256i sign256 = \_mm256\_and\_si256(\_mm256\_castps\_si256(src256), all256\_sing1); // check sign in 8 x 32-bit floats

\_\_m256 result256 = \_mm256\_blendv\_ps(src256, mult256, \_mm256\_castsi256\_ps(sign256)); // (sign>0) ? src : mult;

\_mm256\_storeu\_ps(&x[i], result256);

}

}

for (; i < n; ++i) {

x[i] = (x[i]>0) ? x[i] : .1\*x[i];

}

}

else {

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

x[i] = activate(x[i], a);

}

}

}

void float\_to\_bit(float \*src, unsigned char \*dst, size\_t size)

{

size\_t dst\_size = size / 8 + 1;

memset(dst, 0, dst\_size);

size\_t i;

//\_\_m256i all256\_sing1 = \_mm256\_set\_epi32(0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000);

\_\_m256 float\_zero256 = \_mm256\_set1\_ps(0.0);

for (i = 0; i < size; i+=8)

{

//\_\_m256i src256 = \_mm256\_loadu\_si256((\_\_m256i \*)(&src[i]));

//\_\_m256i result256 = \_mm256\_and\_si256(src256, all256\_sing1); // check sign in 8 x 32-bit floats

//uint32\_t mask = \_mm256\_movemask\_ps(\_mm256\_castsi256\_ps(result256)); // (val >= 0) ? 0 : 1

////mask = ~mask; // inverse mask, (val >= 0) ? 1 : 0

\_\_m256 src256 = \_mm256\_loadu\_ps((float \*)(&src[i]));

\_\_m256 result256 = \_mm256\_cmp\_ps(src256, float\_zero256, \_CMP\_GT\_OS);

uint32\_t mask = \_mm256\_movemask\_ps(result256); // (val > 0) ? 0 : 1

dst[i / 8] = mask;

}

}

static inline void transpose4x4\_SSE(float \*A, float \*B, const int lda, const int ldb)

{

\_\_m128 row1 = \_mm\_loadu\_ps(&A[0 \* lda]);

\_\_m128 row2 = \_mm\_loadu\_ps(&A[1 \* lda]);

\_\_m128 row3 = \_mm\_loadu\_ps(&A[2 \* lda]);

\_\_m128 row4 = \_mm\_loadu\_ps(&A[3 \* lda]);

\_MM\_TRANSPOSE4\_PS(row1, row2, row3, row4);

\_mm\_storeu\_ps(&B[0 \* ldb], row1);

\_mm\_storeu\_ps(&B[1 \* ldb], row2);

\_mm\_storeu\_ps(&B[2 \* ldb], row3);

\_mm\_storeu\_ps(&B[3 \* ldb], row4);

}

void transpose\_block\_SSE4x4(float \*A, float \*B, const int n, const int m,

const int lda, const int ldb, const int block\_size)

{

int i;

#pragma omp parallel for

for (i = 0; i < n; i += block\_size) {

int j, i2, j2;

//int max\_i2 = (i + block\_size < n) ? (i + block\_size) : n;

if (i + block\_size < n) {

int max\_i2 = i + block\_size;

for (j = 0; j < m; j += block\_size) {

//int max\_j2 = (j + block\_size < m) ? (j + block\_size) : m;

if (j + block\_size < m) {

int max\_j2 = j + block\_size;

for (i2 = i; i2 < max\_i2; i2 += 4) {

for (j2 = j; j2 < max\_j2; j2 += 4) {

transpose4x4\_SSE(&A[i2\*lda + j2], &B[j2\*ldb + i2], lda, ldb);

}

}

}

else {

for (i2 = i; i2 < max\_i2; ++i2) {

for (j2 = j; j2 < m; ++j2) {

B[j2\*ldb + i2] = A[i2\*lda + j2];

}

}

}

}

}

else {

for (i2 = i; i2 < n; ++i2) {

for (j2 = 0; j2 < m; ++j2) {

B[j2\*ldb + i2] = A[i2\*lda + j2];

}

}

}

}

}

void forward\_maxpool\_layer\_avx(float \*src, float \*dst, int \*indexes, int size, int w, int h, int out\_w, int out\_h, int c,

int pad, int stride, int batch)

{

const int w\_offset = -pad / 2;

const int h\_offset = -pad / 2;

int b, k;

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

#pragma omp parallel for

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

int i, j, m, n;

for (i = 0; i < out\_h; ++i) {

//for (j = 0; j < out\_w; ++j) {

j = 0;

if(stride == 1 && is\_avx() == 1) {

for (j = 0; j < out\_w - 8 - (size - 1); j += 8) {

int out\_index = j + out\_w\*(i + out\_h\*(k + c\*b));

\_\_m256 max256 = \_mm256\_set1\_ps(-FLT\_MAX);

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

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

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

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

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

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

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

if (!valid) continue;

\_\_m256 src256 = \_mm256\_loadu\_ps(&src[index]);

max256 = \_mm256\_max\_ps(src256, max256);

}

}

\_mm256\_storeu\_ps(&dst[out\_index], max256);

}

}

else if (size == 2 && stride == 2 && is\_avx() == 1) {

for (j = 0; j < out\_w - 4; j += 4) {

int out\_index = j + out\_w\*(i + out\_h\*(k + c\*b));

//float max = -FLT\_MAX;

//int max\_i = -1;

\_\_m128 max128 = \_mm\_set1\_ps(-FLT\_MAX);

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

//for (m = 0; m < size; ++m)

m = 0;

{

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

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

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

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

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

if (!valid) continue;

\_\_m256 src256 = \_mm256\_loadu\_ps(&src[index]);

\_\_m256 src256\_2 = \_mm256\_permute\_ps(src256, (1 << 0) | (3 << 4));

\_\_m256 max256 = \_mm256\_max\_ps(src256, src256\_2);

\_\_m128 src128\_0 = \_mm256\_extractf128\_ps(max256, 0);

\_\_m128 src128\_1 = \_mm256\_extractf128\_ps(max256, 1);

\_\_m128 src128 = \_mm\_shuffle\_ps(src128\_0, src128\_1, (2 << 2) | (2 << 6));

max128 = \_mm\_max\_ps(src128, max128);

}

}

\_mm\_storeu\_ps(&dst[out\_index], max128);

}

}

for (; j < out\_w; ++j) {

int out\_index = j + out\_w\*(i + out\_h\*(k + c\*b));

float max = -FLT\_MAX;

int max\_i = -1;

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

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

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

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

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

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

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

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

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

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

}

}

dst[out\_index] = max;

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

}

}

}

}

}

#else // AVX

int is\_avx() {

return 0;

}

int is\_fma\_avx2() {

return 0;

}

void gemm\_nn(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i, j, k;

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

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

PUT\_IN\_REGISTER float A\_PART = ALPHA \* A[i \* lda + k];

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

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

}

}

}

void gemm\_nn\_fast(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i, j, k;

#pragma omp parallel for

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

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

PUT\_IN\_REGISTER float A\_PART = ALPHA\*A[i\*lda + k];

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

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

}

}

}

void gemm\_nn\_bin\_32bit\_packed(int M, int N, int K, float ALPHA,

uint32\_t \*A, int lda,

uint32\_t \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int i;

#pragma omp parallel for

for (i = 0; i < M; ++i) { // l.n

int j, s;

float mean\_val = mean\_arr[i];

//printf(" l.mean\_arr[i] = %d \n ", l.mean\_arr[i]);

for (s = 0; s < K; ++s) // l.size\*l.size\*l.c/32 or (l.size\*l.size\*l.c)

{

//PUT\_IN\_REGISTER float A\_PART = 1\*a[i\*k + s];

PUT\_IN\_REGISTER uint32\_t A\_PART = A[i \* lda + s];

for (j = 0; j < N; ++j) // out\_h\*out\_w;

{

//c[i\*n + j] += A\_PART\*b[s\*n + j];

PUT\_IN\_REGISTER uint32\_t B\_PART = B[s \* ldb + j];

uint32\_t xnor\_result = ~(A\_PART ^ B\_PART);

//printf(" xnor\_result = %d, ", xnor\_result);

int32\_t count = popcnt\_32(xnor\_result); // must be Signed int

C[i\*ldc + j] += (2 \* count - 32) \* mean\_val;

//c[i\*n + j] += count\*mean;

}

}

}

}

void convolution\_2d(int w, int h, int ksize, int n, int c, int pad, int stride,

float \*weights, float \*input, float \*output, float \*mean)

{

const int out\_h = (h + 2 \* pad - ksize) / stride + 1; // output\_height=input\_height for stride=1 and pad=1

const int out\_w = (w + 2 \* pad - ksize) / stride + 1; // output\_width=input\_width for stride=1 and pad=1

//int i, f, j;

int fil;

// filter index

#pragma omp parallel for // "omp parallel for" - automatic parallelization of loop by using OpenMP

for (fil = 0; fil < n; ++fil) {

int chan, y, x, f\_y, f\_x;

// channel index

for (chan = 0; chan < c; ++chan)

// input - y

for (y = 0; y < h; ++y)

// input - x

for (x = 0; x < w; ++x)

{

int const output\_index = fil\*w\*h + y\*w + x;

int const weights\_pre\_index = fil\*c\*ksize\*ksize + chan\*ksize\*ksize;

int const input\_pre\_index = chan\*w\*h;

float sum = 0;

// filter - y

for (f\_y = 0; f\_y < ksize; ++f\_y)

{

int input\_y = y + f\_y - pad;

// filter - x

for (f\_x = 0; f\_x < ksize; ++f\_x)

{

int input\_x = x + f\_x - pad;

if (input\_y < 0 || input\_x < 0 || input\_y >= h || input\_x >= w) continue;

int input\_index = input\_pre\_index + input\_y\*w + input\_x;

int weights\_index = weights\_pre\_index + f\_y\*ksize + f\_x;

sum += input[input\_index] \* weights[weights\_index];

}

}

// l.output[filters][width][height] +=

// state.input[channels][width][height] \*

// l.weights[filters][channels][filter\_width][filter\_height];

output[output\_index] += sum;

}

}

}

static inline int popcnt\_64(uint64\_t val64) {

#ifdef WIN32 // Windows

#ifdef \_WIN64 // Windows 64-bit

int tmp\_count = \_\_popcnt64(val64);

#else // Windows 32-bit

int tmp\_count = \_\_popcnt(val64);

tmp\_count += \_\_popcnt(val64 >> 32);

#endif

#else // Linux

#if defined(\_\_x86\_64\_\_) || defined(\_\_aarch64\_\_) // Linux 64-bit

int tmp\_count = \_\_builtin\_popcountll(val64);

#else // Linux 32-bit

int tmp\_count = \_\_builtin\_popcount(val64);

tmp\_count += \_\_builtin\_popcount(val64 >> 32);

#endif

#endif

return tmp\_count;

}

void gemm\_nn\_custom\_bin\_mean\_transposed(int M, int N, int K, float ALPHA\_UNUSED,

unsigned char \*A, int lda,

unsigned char \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int i;

#pragma omp parallel for

for (i = 0; i < M; ++i) { // l.n - filters [16 - 55 - 1024]

int j, k;

float mean\_val = mean\_arr[i];

for (j = 0; j < N; ++j) { // out\_h\*out\_w - one channel output size [169 - 173056]

int count = 0;

for (k = 0; k < K; k += 64) { // l.size\*l.size\*l.c - one filter size [27 - 9216]

uint64\_t a\_bit64 = \*((uint64\_t \*)(A + (i\*lda + k) / 8));

uint64\_t b\_bit64 = \*((uint64\_t \*)(B + (j\*ldb + k) / 8));

uint64\_t c\_bit64 = xnor\_int64(a\_bit64, b\_bit64);

int tmp\_count = popcnt\_64(c\_bit64);

if (K - k < 64) tmp\_count = tmp\_count - (64 - (K - k)); // remove extra bits

count += tmp\_count;

//binary\_int64\_printf(c\_bit64);

//printf(", count = %d \n\n", tmp\_count);

}

C[i\*ldc + j] = (2 \* count - K) \* mean\_val;

}

}

}

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

int channels, int height, int width,

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

{

printf("\n im2col\_cpu\_custom\_transpose() isn't implemented without AVX \n");

}

//From Berkeley Vision's Caffe!

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

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

int channels, int height, int width,

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

{

im2col\_cpu(data\_im, channels, height, width, ksize, stride, pad, data\_col);

return;

int c;

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

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

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

// optimized version

if (height\_col == height && width\_col == width && stride == 1 && pad == 1)

{

#pragma omp parallel for

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

int h, w;

int w\_offset = c % ksize;

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

int c\_im = c / ksize / ksize;

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

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

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

}

for (; w < width\_col - pad; ++w) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

}

}

{

w = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

{

w = width\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

{

h = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

{

h = height\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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);

}

}

}

}

else {

//printf("\n Error: is no non-optimized version \n");

im2col\_cpu(data\_im, channels, height, width, ksize, stride, pad, data\_col);

}

}

//From Berkeley Vision's Caffe!

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

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

int channels, int height, int width,

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

{

int c;

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

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

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

// optimized version

if (height\_col == height && width\_col == width && stride == 1 && pad == 1)

{

int new\_ldb = bit\_align;

#pragma omp parallel for

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

int h, w;

int w\_offset = c % ksize;

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

int c\_im = c / ksize / ksize;

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

for (w = pad; w < width\_col - pad - 8; w += 1) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

float val = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

if (val > 0) set\_bit((unsigned char\*)data\_col, col\_index);

}

for (; w < width\_col - pad; ++w) {

int im\_row = h\_offset + h - pad;

int im\_col = w\_offset + w - pad;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

float val = data\_im[im\_col + width\*(im\_row + height\*c\_im)];

if (val > 0) set\_bit((unsigned char\*)data\_col, col\_index);

}

}

{

w = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\*)data\_col, col\_index);

}

}

{

w = width\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\*)data\_col, col\_index);

}

}

{

h = 0;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\*)data\_col, col\_index);

}

}

{

h = height\_col - 1;

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

int im\_row = h\_offset + h;

int im\_col = w\_offset + w;

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

int col\_index = c \* new\_ldb + h \* width\_col + w;

//data\_col[col\_index] = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

float val = im2col\_get\_pixel(data\_im, height, width, channels, im\_row, im\_col, c\_im, pad);

if (val > 0) set\_bit((unsigned char\*)data\_col, col\_index);

}

}

}

}

else {

printf("\n Error: is no non-optimized version \n");

//im2col\_cpu(data\_im, channels, height, width, ksize, stride, pad, data\_col); // must be aligned for transpose after float\_to\_bin

// float\_to\_bit(b, t\_input, src\_size);

// transpose\_bin(t\_input, \*t\_bit\_input, k, n, bit\_align, new\_ldb, 8);

}

}

void activate\_array\_cpu\_custom(float \*x, const int n, const ACTIVATION a)

{

int i;

if (a == LINEAR)

{

}

else if (a == LEAKY)

{

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

x[i] = (x[i]>0) ? x[i] : .1\*x[i];

}

}

else {

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

x[i] = activate(x[i], a);

}

}

}

void float\_to\_bit(float \*src, unsigned char \*dst, size\_t size)

{

size\_t dst\_size = size / 8 + 1;

memset(dst, 0, dst\_size);

size\_t i;

char\* byte\_arr = (char\*)xcalloc(size, sizeof(char));

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

if (src[i] > 0) byte\_arr[i] = 1;

}

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

// dst[i / 8] |= byte\_arr[i] << (i % 8);

//}

for (i = 0; i < size; i += 8) {

char dst\_tmp = 0;

dst\_tmp |= byte\_arr[i + 0] << 0;

dst\_tmp |= byte\_arr[i + 1] << 1;

dst\_tmp |= byte\_arr[i + 2] << 2;

dst\_tmp |= byte\_arr[i + 3] << 3;

dst\_tmp |= byte\_arr[i + 4] << 4;

dst\_tmp |= byte\_arr[i + 5] << 5;

dst\_tmp |= byte\_arr[i + 6] << 6;

dst\_tmp |= byte\_arr[i + 7] << 7;

dst[i / 8] = dst\_tmp;

}

free(byte\_arr);

}

static inline void transpose\_scalar\_block(float \*A, float \*B, const int lda, const int ldb, const int block\_size)

{

int i;

//#pragma omp parallel for

for (i = 0; i<block\_size; i++) {

int j;

for (j = 0; j<block\_size; j++) {

B[j\*ldb + i] = A[i\*lda + j];

}

}

}

void transpose\_block\_SSE4x4(float \*A, float \*B, const int n, const int m,

const int lda, const int ldb, const int block\_size)

{

int i;

#pragma omp parallel for

for (i = 0; i < n; i += block\_size) {

int j, i2, j2;

for (j = 0; j < m; j += block\_size) {

int max\_i2 = i + block\_size < n ? i + block\_size : n;

int max\_j2 = j + block\_size < m ? j + block\_size : m;

for (i2 = i; i2 < max\_i2; ++i2) {

for (j2 = j; j2 < max\_j2; ++j2) {

B[j2\*ldb + i2] = A[i2\*lda + j2];

}

}

}

}

}

void forward\_maxpool\_layer\_avx(float \*src, float \*dst, int \*indexes, int size, int w, int h, int out\_w, int out\_h, int c,

int pad, int stride, int batch)

{

int b, k;

const int w\_offset = -pad / 2;

const int h\_offset = -pad / 2;

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

#pragma omp parallel for

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

int i, j, m, n;

for (i = 0; i < out\_h; ++i) {

for (j = 0; j < out\_w; ++j) {

int out\_index = j + out\_w\*(i + out\_h\*(k + c\*b));

float max = -FLT\_MAX;

int max\_i = -1;

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

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

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

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

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

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

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

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

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

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

}

}

dst[out\_index] = max;

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

}

}

}

}

}

#endif // AVX

// 32 channels -> 1 channel (with 32 floats)

// 256 channels -> 8 channels (with 32 floats)

void repack\_input(float \*input, float \*re\_packed\_input, int w, int h, int c)

{

const int items\_per\_channel = w \* h;

int chan, i;

for (chan = 0; chan < c; chan += 32)

{

for (i = 0; i < items\_per\_channel; ++i)

{

int c\_pack;

for (c\_pack = 0; c\_pack < 32; ++c\_pack) {

float src = input[(chan + c\_pack)\*items\_per\_channel + i];

re\_packed\_input[chan\*items\_per\_channel + i \* 32 + c\_pack] = src;

}

}

}

}

void transpose\_uint32(uint32\_t \*src, uint32\_t \*dst, int src\_h, int src\_w, int src\_align, int dst\_align)

{

//l.bit\_align - algined (n) by 32

//new\_ldb - aligned (k) by 256

int i;

//#pragma omp parallel for

for (i = 0; i < src\_h; i += 1) // l.size\*l.size\*l.c;

{

int j;

for (j = 0; j < src\_w; j += 1) // out\_h\*out\_w;

{

((uint32\_t \*)dst)[j\*dst\_align / 32 + i] = ((uint32\_t \*)src)[i\*src\_align + j];

}

}

}

void gemm\_nn\_bin\_transposed\_32bit\_packed(int M, int N, int K, float ALPHA,

uint32\_t \*A, int lda,

uint32\_t \*B, int ldb,

float \*C, int ldc, float \*mean\_arr)

{

int i;

#pragma omp parallel for

for (i = 0; i < M; ++i) { // l.n

int j, s;

float mean\_val = mean\_arr[i];

for (j = 0; j < N; ++j) // out\_h\*out\_w;

{

float val = 0;

for (s = 0; s < K; ++s) // l.size\*l.size\*l.c/32 or (l.size\*l.size\*l.c)

{

PUT\_IN\_REGISTER uint32\_t A\_PART = ((uint32\_t\*)A)[i\*lda + s];

PUT\_IN\_REGISTER uint32\_t B\_PART = ((uint32\_t\*)B)[j \* ldb + s];

uint32\_t xnor\_result = ~(A\_PART ^ B\_PART);

int32\_t count = popcnt\_32(xnor\_result); // must be Signed int

val += (2 \* count - 32) \* mean\_val;

}

C[i\*ldc + j] += val;

}

}

}

void convolution\_repacked(uint32\_t \*packed\_input, uint32\_t \*packed\_weights, float \*output,

int w, int h, int c, int n, int size, int pad, int new\_lda, float \*mean\_arr)

{

int fil;

// filter index

#pragma omp parallel for

for (fil = 0; fil < n; ++fil) {

float mean\_val = mean\_arr[fil];

int chan, y, x, f\_y, f\_x; // c\_pack

// channel index

for (chan = 0; chan < c / 32; ++chan)

//for (chan = 0; chan < l.c; chan += 32)

//for (c\_pack = 0; c\_pack < 32; ++c\_pack)

// input - y

for (y = 0; y < h; ++y)

// input - x

for (x = 0; x < w; ++x)

{

int const output\_index = fil\*w\*h + y\*w + x;

float sum = 0;

// filter - y

for (f\_y = 0; f\_y < size; ++f\_y)

{

int input\_y = y + f\_y - pad;

// filter - x

for (f\_x = 0; f\_x < size; ++f\_x)

{

int input\_x = x + f\_x - pad;

if (input\_y < 0 || input\_x < 0 || input\_y >= h || input\_x >= w) continue;

// normal

//float input = state.input[(chan + c\_pack)\*l.w\*l.h + input\_y\*l.w + input\_x];

//float weight = l.weights[fil\*l.c\*l.size\*l.size + (chan + c\_pack)\*l.size\*l.size + f\_y\*l.size + f\_x];

// packed

//float input = re\_packed\_input[chan\*l.w\*l.h + (input\_y\*l.w + input\_x) \* 32 + c\_pack];

//float weight = l.weights[fil\*l.c\*l.size\*l.size + chan\*l.size\*l.size + (f\_y\*l.size + f\_x) \* 32 + c\_pack];

//sum += input \* weight;

//float input = re\_packed\_input[chan\*l.w\*l.h + (input\_y\*l.w + input\_x) \* 32 + c\_pack];

//float weight = l.weights[fil\*l.c\*l.size\*l.size + chan\*l.size\*l.size + (f\_y\*l.size + f\_x) \* 32 + c\_pack];

//uint32\_t bit1 = input > 0;

//uint32\_t bit2 = weight > 0;

//uint32\_t count = (~(bit1 ^ bit2)) & 1;

//float result = (2 \* (float)count - 1) \* mean\_val;

//printf("\n mul = %f, bit1 = %d, bit2 = %d, count = %d, mean = %f, result = %f ", input\*weight, bit1, bit2, count, mean\_val, result);

//sum += result;

uint32\_t input = ((uint32\_t \*)packed\_input)[chan\*w\*h + input\_y\*w + input\_x];

//uint32\_t weight = ((uint32\_t \*)l.align\_bit\_weights)[fil\*l.c\*l.size\*l.size/32 + chan\*l.size\*l.size + f\_y\*l.size + f\_x];

uint32\_t weight = ((uint32\_t \*)packed\_weights)[fil\*new\_lda / 32 + chan\*size\*size + f\_y\*size + f\_x];

uint32\_t xnor\_result = ~(input ^ weight);

int32\_t count = popcnt\_32(xnor\_result); // mandatory Signed int

sum += (2 \* count - 32) \* mean\_val;

}

}

// l.output[filters][width][height] +=

// state.input[channels][width][height] \*

// l.weights[filters][channels][filter\_width][filter\_height];

output[output\_index] += sum;

}

}

}

void gemm\_nt(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i,j,k;

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

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

PUT\_IN\_REGISTER float sum = 0;

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

sum += ALPHA\*A[i\*lda+k]\*B[j\*ldb + k];

}

C[i\*ldc+j] += sum;

}

}

}

void gemm\_tn(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i,j,k;

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

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

PUT\_IN\_REGISTER float A\_PART = ALPHA \* A[k \* lda + i];

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

C[i\*ldc+j] += A\_PART\*B[k\*ldb+j];

}

}

}

}

void gemm\_tt(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i,j,k;

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

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

PUT\_IN\_REGISTER float sum = 0;

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

sum += ALPHA\*A[i+k\*lda]\*B[k+j\*ldb];

}

C[i\*ldc+j] += sum;

}

}

}

void gemm\_cpu(int TA, int TB, int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float BETA,

float \*C, int ldc)

{

//printf("cpu: %d %d %d %d %d %f %d %d %f %d\n",TA, TB, M, N, K, ALPHA, lda, ldb, BETA, ldc);

if (BETA != 1){

int i, j;

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

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

C[i\*ldc + j] \*= BETA;

}

}

}

is\_avx(); // initialize static variable

if (is\_fma\_avx2() && !TA && !TB) {

gemm\_nn\_fast(M, N, K, ALPHA, A, lda, B, ldb, C, ldc);

}

else {

int t;

#pragma omp parallel for

for (t = 0; t < M; ++t) {

if (!TA && !TB)

gemm\_nn(1, N, K, ALPHA, A + t\*lda, lda, B, ldb, C + t\*ldc, ldc);

else if (TA && !TB)

gemm\_tn(1, N, K, ALPHA, A + t, lda, B, ldb, C + t\*ldc, ldc);

else if (!TA && TB)

gemm\_nt(1, N, K, ALPHA, A + t\*lda, lda, B, ldb, C + t\*ldc, ldc);

else

gemm\_tt(1, N, K, ALPHA, A + t, lda, B, ldb, C + t\*ldc, ldc);

}

}

}

#ifdef GPU

#include <math.h>

void gemm\_ongpu(int TA, int TB, int M, int N, int K, float ALPHA,

float \*A\_gpu, int lda,

float \*B\_gpu, int ldb,

float BETA,

float \*C\_gpu, int ldc)

{

cublasHandle\_t handle = blas\_handle();

cudaError\_t stream\_status = (cudaError\_t)cublasSetStream(handle, get\_cuda\_stream());

CHECK\_CUDA(stream\_status);

cudaError\_t status = (cudaError\_t)cublasSgemm(handle, (TB ? CUBLAS\_OP\_T : CUBLAS\_OP\_N),

(TA ? CUBLAS\_OP\_T : CUBLAS\_OP\_N), N, M, K, &ALPHA, B\_gpu, ldb, A\_gpu, lda, &BETA, C\_gpu, ldc);

CHECK\_CUDA(status);

}

void gemm\_gpu(int TA, int TB, int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float BETA,

float \*C, int ldc)

{

float \*A\_gpu = cuda\_make\_array(A, (TA ? lda\*K:lda\*M));

float \*B\_gpu = cuda\_make\_array(B, (TB ? ldb\*N : ldb\*K));

float \*C\_gpu = cuda\_make\_array(C, ldc\*M);

gemm\_ongpu(TA, TB, M, N, K, ALPHA, A\_gpu, lda, B\_gpu, ldb, BETA, C\_gpu, ldc);

cuda\_pull\_array(C\_gpu, C, ldc\*M);

cuda\_free(A\_gpu);

cuda\_free(B\_gpu);

cuda\_free(C\_gpu);

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

void time\_gpu\_random\_matrix(int TA, int TB, int m, int k, int n)

{

float \*a;

if(!TA) a = random\_matrix(m,k);

else a = random\_matrix(k,m);

int lda = (!TA)?k:m;

float \*b;

if(!TB) b = random\_matrix(k,n);

else b = random\_matrix(n,k);

int ldb = (!TB)?n:k;

float \*c = random\_matrix(m,n);

int i;

clock\_t start = clock(), end;

for(i = 0; i<32; ++i){

gemm\_gpu(TA,TB,m,n,k,1,a,lda,b,ldb,1,c,n);

}

end = clock();

printf("Matrix Multiplication %dx%d \* %dx%d, TA=%d, TB=%d: %lf s\n",m,k,k,n, TA, TB, (float)(end-start)/CLOCKS\_PER\_SEC);

free(a);

free(b);

free(c);

}

void time\_ongpu(int TA, int TB, int m, int k, int n)

{

int iter = 10;

float \*a = random\_matrix(m,k);

float \*b = random\_matrix(k,n);

int lda = (!TA)?k:m;

int ldb = (!TB)?n:k;

float \*c = random\_matrix(m,n);

float \*a\_cl = cuda\_make\_array(a, m\*k);

float \*b\_cl = cuda\_make\_array(b, k\*n);

float \*c\_cl = cuda\_make\_array(c, m\*n);

int i;

clock\_t start = clock(), end;

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

gemm\_ongpu(TA,TB,m,n,k,1,a\_cl,lda,b\_cl,ldb,1,c\_cl,n);

cudaDeviceSynchronize();

}

double flop = ((double)m)\*n\*(2.\*k + 2.)\*iter;

double gflop = flop/pow(10., 9);

end = clock();

double seconds = sec(end-start);

printf("Matrix Multiplication %dx%d \* %dx%d, TA=%d, TB=%d: %lf s, %lf GFLOPS\n",m,k,k,n, TA, TB, seconds, gflop/seconds);

cuda\_free(a\_cl);

cuda\_free(b\_cl);

cuda\_free(c\_cl);

free(a);

free(b);

free(c);

}

void test\_gpu\_accuracy(int TA, int TB, int m, int k, int n)

{

srand(0);

float \*a;

if(!TA) a = random\_matrix(m,k);

else a = random\_matrix(k,m);

int lda = (!TA)?k:m;

float \*b;

if(!TB) b = random\_matrix(k,n);

else b = random\_matrix(n,k);

int ldb = (!TB)?n:k;

float \*c = random\_matrix(m,n);

float \*c\_gpu = random\_matrix(m,n);

memset(c, 0, m\*n\*sizeof(float));

memset(c\_gpu, 0, m\*n\*sizeof(float));

int i;

//pm(m,k,b);

gemm\_gpu(TA,TB,m,n,k,1,a,lda,b,ldb,1,c\_gpu,n);

//printf("GPU\n");

//pm(m, n, c\_gpu);

gemm\_cpu(TA,TB,m,n,k,1,a,lda,b,ldb,1,c,n);

//printf("\n\nCPU\n");

//pm(m, n, c);

double sse = 0;

for(i = 0; i < m\*n; ++i) {

//printf("%f %f\n", c[i], c\_gpu[i]);

sse += pow(c[i]-c\_gpu[i], 2);

}

printf("Matrix Multiplication %dx%d \* %dx%d, TA=%d, TB=%d: %g SSE\n",m,k,k,n, TA, TB, sse/(m\*n));

free(a);

free(b);

free(c);

free(c\_gpu);

}

int test\_gpu\_blas()

{

/\*

test\_gpu\_accuracy(0,0,10,576,75);

test\_gpu\_accuracy(0,0,17,10,10);

test\_gpu\_accuracy(1,0,17,10,10);

test\_gpu\_accuracy(0,1,17,10,10);

test\_gpu\_accuracy(1,1,17,10,10);

test\_gpu\_accuracy(0,0,1000,10,100);

test\_gpu\_accuracy(1,0,1000,10,100);

test\_gpu\_accuracy(0,1,1000,10,100);

test\_gpu\_accuracy(1,1,1000,10,100);

test\_gpu\_accuracy(0,0,10,10,10);

time\_ongpu(0,0,64,2916,363);

time\_ongpu(0,0,64,2916,363);

time\_ongpu(0,0,64,2916,363);

time\_ongpu(0,0,192,729,1600);

time\_ongpu(0,0,384,196,1728);

time\_ongpu(0,0,256,196,3456);

time\_ongpu(0,0,256,196,2304);

time\_ongpu(0,0,128,4096,12544);

time\_ongpu(0,0,128,4096,4096);

\*/

time\_ongpu(0,0,64,75,12544);

time\_ongpu(0,0,64,75,12544);

time\_ongpu(0,0,64,75,12544);

time\_ongpu(0,0,64,576,12544);

time\_ongpu(0,0,256,2304,784);

time\_ongpu(1,1,2304,256,784);

time\_ongpu(0,0,512,4608,196);

time\_ongpu(1,1,4608,512,196);

return 0;

}

#endif

void init\_cpu() {

is\_avx();

is\_fma\_avx2();

}