#ifdef \_\_cplusplus

extern "C" {

#endif

int cuda\_debug\_sync = 0;

int gpu\_index = 0;

#ifdef \_\_cplusplus

}

#endif // \_\_cplusplus

#ifdef GPU

#include "dark\_cuda.h"

#include "utils.h"

#include "blas.h"

#include "assert.h"

#include <stdlib.h>

#include <time.h>

#include <cuda.h>

#include <stdio.h>

#pragma comment(lib, "cuda.lib")

#ifdef CUDNN

#ifndef USE\_CMAKE\_LIBS

#pragma comment(lib, "cudnn.lib")

#endif // USE\_CMAKE\_LIBS

#endif // CUDNN

#if defined(CUDNN\_HALF) && !defined(CUDNN)

#error "If you set CUDNN\_HALF=1 then you must set CUDNN=1"

#endif

void cuda\_set\_device(int n)

{

gpu\_index = n;

cudaError\_t status = cudaSetDevice(n);

if(status != cudaSuccess) CHECK\_CUDA(status);

}

int cuda\_get\_device()

{

int n = 0;

cudaError\_t status = cudaGetDevice(&n);

CHECK\_CUDA(status);

return n;

}

void \*cuda\_get\_context()

{

CUcontext pctx;

CUresult status = cuCtxGetCurrent(&pctx);

if(status != CUDA\_SUCCESS) fprintf(stderr, " Error: cuCtxGetCurrent() is failed \n");

return (void \*)pctx;

}

void check\_error(cudaError\_t status)

{

cudaError\_t status2 = cudaGetLastError();

if (status != cudaSuccess)

{

const char \*s = cudaGetErrorString(status);

char buffer[256];

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

snprintf(buffer, 256, "CUDA Error: %s", s);

#ifdef WIN32

getchar();

#endif

error(buffer);

}

if (status2 != cudaSuccess)

{

const char \*s = cudaGetErrorString(status2);

char buffer[256];

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

snprintf(buffer, 256, "CUDA Error Prev: %s", s);

#ifdef WIN32

getchar();

#endif

error(buffer);

}

}

void check\_error\_extended(cudaError\_t status, const char \*file, int line, const char \*date\_time)

{

if (status != cudaSuccess) {

printf("CUDA status Error: file: %s() : line: %d : build time: %s \n", file, line, date\_time);

check\_error(status);

}

#if defined(DEBUG) || defined(CUDA\_DEBUG)

cuda\_debug\_sync = 1;

#endif

if (cuda\_debug\_sync) {

status = cudaDeviceSynchronize();

if (status != cudaSuccess)

printf("CUDA status = cudaDeviceSynchronize() Error: file: %s() : line: %d : build time: %s \n", file, line, date\_time);

}

check\_error(status);

}

dim3 cuda\_gridsize(size\_t n){

size\_t k = (n-1) / BLOCK + 1;

size\_t x = k;

size\_t y = 1;

if(x > 65535){

x = ceil(sqrt(k));

y = (n-1)/(x\*BLOCK) + 1;

}

//dim3 d = { (unsigned int)x, (unsigned int)y, 1 };

dim3 d;

d.x = x;

d.y = y;

d.z = 1;

//printf("%ld %ld %ld %ld\n", n, x, y, x\*y\*BLOCK);

return d;

}

static cudaStream\_t streamsArray[16]; // cudaStreamSynchronize( get\_cuda\_stream() );

static int streamInit[16] = { 0 };

cudaStream\_t get\_cuda\_stream() {

int i = cuda\_get\_device();

if (!streamInit[i]) {

//printf("Create CUDA-stream \n");

cudaError\_t status = cudaStreamCreate(&streamsArray[i]);

//cudaError\_t status = cudaStreamCreateWithFlags(&streamsArray[i], cudaStreamNonBlocking);

if (status != cudaSuccess) {

printf(" cudaStreamCreate error: %d \n", status);

const char \*s = cudaGetErrorString(status);

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

status = cudaStreamCreateWithFlags(&streamsArray[i], cudaStreamDefault);

CHECK\_CUDA(status);

}

streamInit[i] = 1;

}

return streamsArray[i];

}

static cudaStream\_t streamsArray2[16]; // cudaStreamSynchronize( get\_cuda\_memcpy\_stream() );

static int streamInit2[16] = { 0 };

cudaStream\_t get\_cuda\_memcpy\_stream() {

int i = cuda\_get\_device();

if (!streamInit2[i]) {

cudaError\_t status = cudaStreamCreate(&streamsArray2[i]);

//cudaError\_t status = cudaStreamCreateWithFlags(&streamsArray2[i], cudaStreamNonBlocking);

if (status != cudaSuccess) {

printf(" cudaStreamCreate-Memcpy error: %d \n", status);

const char \*s = cudaGetErrorString(status);

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

status = cudaStreamCreateWithFlags(&streamsArray2[i], cudaStreamDefault);

CHECK\_CUDA(status);

}

streamInit2[i] = 1;

}

return streamsArray2[i];

}

#ifdef CUDNN

cudnnHandle\_t cudnn\_handle()

{

static int init[16] = {0};

static cudnnHandle\_t handle[16];

int i = cuda\_get\_device();

if(!init[i]) {

cudnnCreate(&handle[i]);

init[i] = 1;

cudnnStatus\_t status = cudnnSetStream(handle[i], get\_cuda\_stream());

CHECK\_CUDNN(status);

}

return handle[i];

}

void cudnn\_check\_error(cudnnStatus\_t status)

{

#if defined(DEBUG) || defined(CUDA\_DEBUG)

cudaDeviceSynchronize();

#endif

if (cuda\_debug\_sync) {

cudaDeviceSynchronize();

}

cudnnStatus\_t status2 = CUDNN\_STATUS\_SUCCESS;

#ifdef CUDNN\_ERRQUERY\_RAWCODE

cudnnStatus\_t status\_tmp = cudnnQueryRuntimeError(cudnn\_handle(), &status2, CUDNN\_ERRQUERY\_RAWCODE, NULL);

#endif

if (status != CUDNN\_STATUS\_SUCCESS)

{

const char \*s = cudnnGetErrorString(status);

char buffer[256];

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

snprintf(buffer, 256, "cuDNN Error: %s", s);

#ifdef WIN32

getchar();

#endif

error(buffer);

}

if (status2 != CUDNN\_STATUS\_SUCCESS)

{

const char \*s = cudnnGetErrorString(status2);

char buffer[256];

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

snprintf(buffer, 256, "cuDNN Error Prev: %s", s);

#ifdef WIN32

getchar();

#endif

error(buffer);

}

}

void cudnn\_check\_error\_extended(cudnnStatus\_t status, const char \*file, int line, const char \*date\_time)

{

if (status != CUDNN\_STATUS\_SUCCESS) {

printf("\n cuDNN status Error in: file: %s() : line: %d : build time: %s \n", file, line, date\_time);

cudnn\_check\_error(status);

}

#if defined(DEBUG) || defined(CUDA\_DEBUG)

cuda\_debug\_sync = 1;

#endif

if (cuda\_debug\_sync) {

cudaError\_t status = cudaDeviceSynchronize();

if (status != CUDNN\_STATUS\_SUCCESS)

printf("\n cudaError\_t status = cudaDeviceSynchronize() Error in: file: %s() : line: %d : build time: %s \n", file, line, date\_time);

}

cudnn\_check\_error(status);

}

#endif

cublasHandle\_t blas\_handle()

{

static int init[16] = {0};

static cublasHandle\_t handle[16];

int i = cuda\_get\_device();

if(!init[i]) {

cublasCreate(&handle[i]);

cublasStatus\_t status = cublasSetStream(handle[i], get\_cuda\_stream());

CHECK\_CUDA((cudaError\_t)status);

init[i] = 1;

}

return handle[i];

}

static float \*\*pinned\_ptr = NULL;

static size\_t pinned\_num\_of\_blocks = 0;

static size\_t pinned\_index = 0;

static size\_t pinned\_block\_id = 0;

static const size\_t pinned\_block\_size = (size\_t)1024 \* 1024 \* 1024 \* 1; // 1 GB block size

static pthread\_mutex\_t mutex\_pinned = PTHREAD\_MUTEX\_INITIALIZER;

// free CPU-pinned memory

void free\_pinned\_memory()

{

if (pinned\_ptr) {

int k;

for (k = 0; k < pinned\_num\_of\_blocks; ++k) {

cuda\_free\_host(pinned\_ptr[k]);

}

free(pinned\_ptr);

pinned\_ptr = NULL;

}

}

// custom CPU-pinned memory allocation

void pre\_allocate\_pinned\_memory(const size\_t size)

{

const size\_t num\_of\_blocks = size / pinned\_block\_size + ((size % pinned\_block\_size) ? 1 : 0);

printf("pre\_allocate... pinned\_ptr = %p \n", pinned\_ptr);

pthread\_mutex\_lock(&mutex\_pinned);

if (!pinned\_ptr) {

pinned\_ptr = (float \*\*)calloc(num\_of\_blocks, sizeof(float \*));

if(!pinned\_ptr) error("calloc failed in pre\_allocate() \n");

printf("pre\_allocate: size = %Iu MB, num\_of\_blocks = %Iu, block\_size = %Iu MB \n",

size / (1024\*1024), num\_of\_blocks, pinned\_block\_size / (1024 \* 1024));

int k;

for (k = 0; k < num\_of\_blocks; ++k) {

cudaError\_t status = cudaHostAlloc((void \*\*)&pinned\_ptr[k], pinned\_block\_size, cudaHostRegisterMapped);

if (status != cudaSuccess) fprintf(stderr, " Can't pre-allocate CUDA-pinned buffer on CPU-RAM \n");

CHECK\_CUDA(status);

if (!pinned\_ptr[k]) error("cudaHostAlloc failed\n");

else {

printf(" Allocated %d pinned block \n", pinned\_block\_size);

}

}

pinned\_num\_of\_blocks = num\_of\_blocks;

}

pthread\_mutex\_unlock(&mutex\_pinned);

}

// simple - get pre-allocated pinned memory

float \*cuda\_make\_array\_pinned\_preallocated(float \*x, size\_t n)

{

pthread\_mutex\_lock(&mutex\_pinned);

float \*x\_cpu = NULL;

const size\_t memory\_step = 512;// 4096;

const size\_t size = sizeof(float)\*n;

const size\_t allocation\_size = ((size / memory\_step) + 1) \* memory\_step;

if (pinned\_ptr && pinned\_block\_id < pinned\_num\_of\_blocks && (allocation\_size < pinned\_block\_size/2))

{

if ((allocation\_size + pinned\_index) > pinned\_block\_size) {

const float filled = (float)100 \* pinned\_index / pinned\_block\_size;

printf("\n Pinned block\_id = %d, filled = %f %% \n", pinned\_block\_id, filled);

pinned\_block\_id++;

pinned\_index = 0;

}

if ((allocation\_size + pinned\_index) < pinned\_block\_size && pinned\_block\_id < pinned\_num\_of\_blocks) {

x\_cpu = (float \*)((char \*)pinned\_ptr[pinned\_block\_id] + pinned\_index);

pinned\_index += allocation\_size;

}

else {

//printf("Pre-allocated pinned memory is over! \n");

}

}

if(!x\_cpu) {

if (allocation\_size > pinned\_block\_size / 2) {

printf("Try to allocate new pinned memory, size = %d MB \n", size / (1024 \* 1024));

cudaError\_t status = cudaHostAlloc((void \*\*)&x\_cpu, size, cudaHostRegisterMapped);

if (status != cudaSuccess) fprintf(stderr, " Can't allocate CUDA-pinned memory on CPU-RAM (pre-allocated memory is over too) \n");

CHECK\_CUDA(status);

}

else {

printf("Try to allocate new pinned BLOCK, size = %d MB \n", size / (1024 \* 1024));

pinned\_num\_of\_blocks++;

pinned\_block\_id = pinned\_num\_of\_blocks - 1;

pinned\_index = 0;

pinned\_ptr = (float \*\*)realloc(pinned\_ptr, pinned\_num\_of\_blocks \* sizeof(float \*));

cudaError\_t status = cudaHostAlloc((void \*\*)&pinned\_ptr[pinned\_block\_id], pinned\_block\_size, cudaHostRegisterMapped);

if (status != cudaSuccess) fprintf(stderr, " Can't pre-allocate CUDA-pinned buffer on CPU-RAM \n");

CHECK\_CUDA(status);

x\_cpu = pinned\_ptr[pinned\_block\_id];

}

}

if (x) {

cudaError\_t status = cudaMemcpyAsync(x\_cpu, x, size, cudaMemcpyDefault, get\_cuda\_stream());

CHECK\_CUDA(status);

}

pthread\_mutex\_unlock(&mutex\_pinned);

return x\_cpu;

}

float \*cuda\_make\_array\_pinned(float \*x, size\_t n)

{

float \*x\_gpu;

size\_t size = sizeof(float)\*n;

//cudaError\_t status = cudaMalloc((void \*\*)&x\_gpu, size);

cudaError\_t status = cudaHostAlloc((void \*\*)&x\_gpu, size, cudaHostRegisterMapped);

if (status != cudaSuccess) fprintf(stderr, " Can't allocate CUDA-pinned memory on CPU-RAM \n");

CHECK\_CUDA(status);

if (x) {

status = cudaMemcpyAsync(x\_gpu, x, size, cudaMemcpyDefault, get\_cuda\_stream());

CHECK\_CUDA(status);

}

if (!x\_gpu) error("cudaHostAlloc failed\n");

return x\_gpu;

}

float \*cuda\_make\_array(float \*x, size\_t n)

{

float \*x\_gpu;

size\_t size = sizeof(float)\*n;

cudaError\_t status = cudaMalloc((void \*\*)&x\_gpu, size);

//cudaError\_t status = cudaMallocManaged((void \*\*)&x\_gpu, size, cudaMemAttachGlobal);

//status = cudaMemAdvise(x\_gpu, size, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);

if (status != cudaSuccess) fprintf(stderr, " Try to set subdivisions=64 in your cfg-file. \n");

CHECK\_CUDA(status);

if(x){

//status = cudaMemcpy(x\_gpu, x, size, cudaMemcpyHostToDevice);

status = cudaMemcpyAsync(x\_gpu, x, size, cudaMemcpyDefault, get\_cuda\_stream());

CHECK\_CUDA(status);

}

if(!x\_gpu) error("Cuda malloc failed\n");

return x\_gpu;

}

void \*\*cuda\_make\_array\_pointers(void \*\*x, size\_t n)

{

void \*\*x\_gpu;

size\_t size = sizeof(void\*) \* n;

cudaError\_t status = cudaMalloc((void \*\*)&x\_gpu, size);

if (status != cudaSuccess) fprintf(stderr, " Try to set subdivisions=64 in your cfg-file. \n");

CHECK\_CUDA(status);

if (x) {

status = cudaMemcpyAsync(x\_gpu, x, size, cudaMemcpyDefault, get\_cuda\_stream());

CHECK\_CUDA(status);

}

if (!x\_gpu) error("Cuda malloc failed\n");

return x\_gpu;

}

void cuda\_random(float \*x\_gpu, size\_t n)

{

static curandGenerator\_t gen[16];

static int init[16] = {0};

int i = cuda\_get\_device();

if(!init[i]){

curandCreateGenerator(&gen[i], CURAND\_RNG\_PSEUDO\_DEFAULT);

curandSetPseudoRandomGeneratorSeed(gen[i], time(0));

init[i] = 1;

}

curandGenerateUniform(gen[i], x\_gpu, n);

CHECK\_CUDA(cudaPeekAtLastError());

}

float cuda\_compare(float \*x\_gpu, float \*x, size\_t n, char \*s)

{

float\* tmp = (float\*)xcalloc(n, sizeof(float));

cuda\_pull\_array(x\_gpu, tmp, n);

//int i;

//for(i = 0; i < n; ++i) printf("%f %f\n", tmp[i], x[i]);

axpy\_cpu(n, -1, x, 1, tmp, 1);

float err = dot\_cpu(n, tmp, 1, tmp, 1);

printf("Error %s: %f\n", s, sqrt(err/n));

free(tmp);

return err;

}

int \*cuda\_make\_int\_array(size\_t n)

{

int \*x\_gpu;

size\_t size = sizeof(int)\*n;

cudaError\_t status = cudaMalloc((void \*\*)&x\_gpu, size);

if(status != cudaSuccess) fprintf(stderr, " Try to set subdivisions=64 in your cfg-file. \n");

CHECK\_CUDA(status);

return x\_gpu;

}

int \*cuda\_make\_int\_array\_new\_api(int \*x, size\_t n)

{

int \*x\_gpu;

size\_t size = sizeof(int)\*n;

cudaError\_t status = cudaMalloc((void \*\*)&x\_gpu, size);

CHECK\_CUDA(status);

if (x) {

//status = cudaMemcpy(x\_gpu, x, size, cudaMemcpyHostToDevice);

cudaError\_t status = cudaMemcpyAsync(x\_gpu, x, size, cudaMemcpyHostToDevice, get\_cuda\_stream());

CHECK\_CUDA(status);

}

if (!x\_gpu) error("Cuda malloc failed\n");

return x\_gpu;

}

void cuda\_free(float \*x\_gpu)

{

//cudaStreamSynchronize(get\_cuda\_stream());

cudaError\_t status = cudaFree(x\_gpu);

CHECK\_CUDA(status);

}

void cuda\_free\_host(float \*x\_cpu)

{

//cudaStreamSynchronize(get\_cuda\_stream());

cudaError\_t status = cudaFreeHost(x\_cpu);

CHECK\_CUDA(status);

}

void cuda\_push\_array(float \*x\_gpu, float \*x, size\_t n)

{

size\_t size = sizeof(float)\*n;

//cudaError\_t status = cudaMemcpy(x\_gpu, x, size, cudaMemcpyHostToDevice);

cudaError\_t status = cudaMemcpyAsync(x\_gpu, x, size, cudaMemcpyHostToDevice, get\_cuda\_stream());

CHECK\_CUDA(status);

}

void cuda\_pull\_array(float \*x\_gpu, float \*x, size\_t n)

{

size\_t size = sizeof(float)\*n;

//cudaError\_t status = cudaMemcpy(x, x\_gpu, size, cudaMemcpyDeviceToHost);

cudaError\_t status = cudaMemcpyAsync(x, x\_gpu, size, cudaMemcpyDeviceToHost, get\_cuda\_stream());

CHECK\_CUDA(status);

cudaStreamSynchronize(get\_cuda\_stream());

}

void cuda\_pull\_array\_async(float \*x\_gpu, float \*x, size\_t n)

{

size\_t size = sizeof(float)\*n;

cudaError\_t status = cudaMemcpyAsync(x, x\_gpu, size, cudaMemcpyDefault, get\_cuda\_stream());

check\_error(status);

//cudaStreamSynchronize(get\_cuda\_stream());

}

int get\_number\_of\_blocks(int array\_size, int block\_size)

{

return array\_size / block\_size + ((array\_size % block\_size > 0) ? 1 : 0);

}

int get\_gpu\_compute\_capability(int i)

{

typedef struct cudaDeviceProp cudaDeviceProp;

cudaDeviceProp prop;

cudaError\_t status = cudaGetDeviceProperties(&prop, i);

CHECK\_CUDA(status);

int cc = prop.major \* 100 + prop.minor \* 10; // \_\_CUDA\_ARCH\_\_ format

return cc;

}

void show\_cuda\_cudnn\_info()

{

int cuda\_version = 0, cuda\_driver\_version = 0, device\_count = 0;

CHECK\_CUDA(cudaRuntimeGetVersion(&cuda\_version));

CHECK\_CUDA(cudaDriverGetVersion(&cuda\_driver\_version));

fprintf(stderr, " CUDA-version: %d (%d)", cuda\_version, cuda\_driver\_version);

if(cuda\_version > cuda\_driver\_version) fprintf(stderr, "\n Warning: CUDA-version is higher than Driver-version! \n");

#ifdef CUDNN

fprintf(stderr, ", cuDNN: %d.%d.%d", CUDNN\_MAJOR, CUDNN\_MINOR, CUDNN\_PATCHLEVEL);

#endif // CUDNN

#ifdef CUDNN\_HALF

fprintf(stderr, ", CUDNN\_HALF=1");

#endif // CUDNN\_HALF

CHECK\_CUDA(cudaGetDeviceCount(&device\_count));

fprintf(stderr, ", GPU count: %d ", device\_count);

fprintf(stderr, " \n");

}

#else // GPU

#include "darknet.h"

void cuda\_set\_device(int n) {}

#endif // GPU