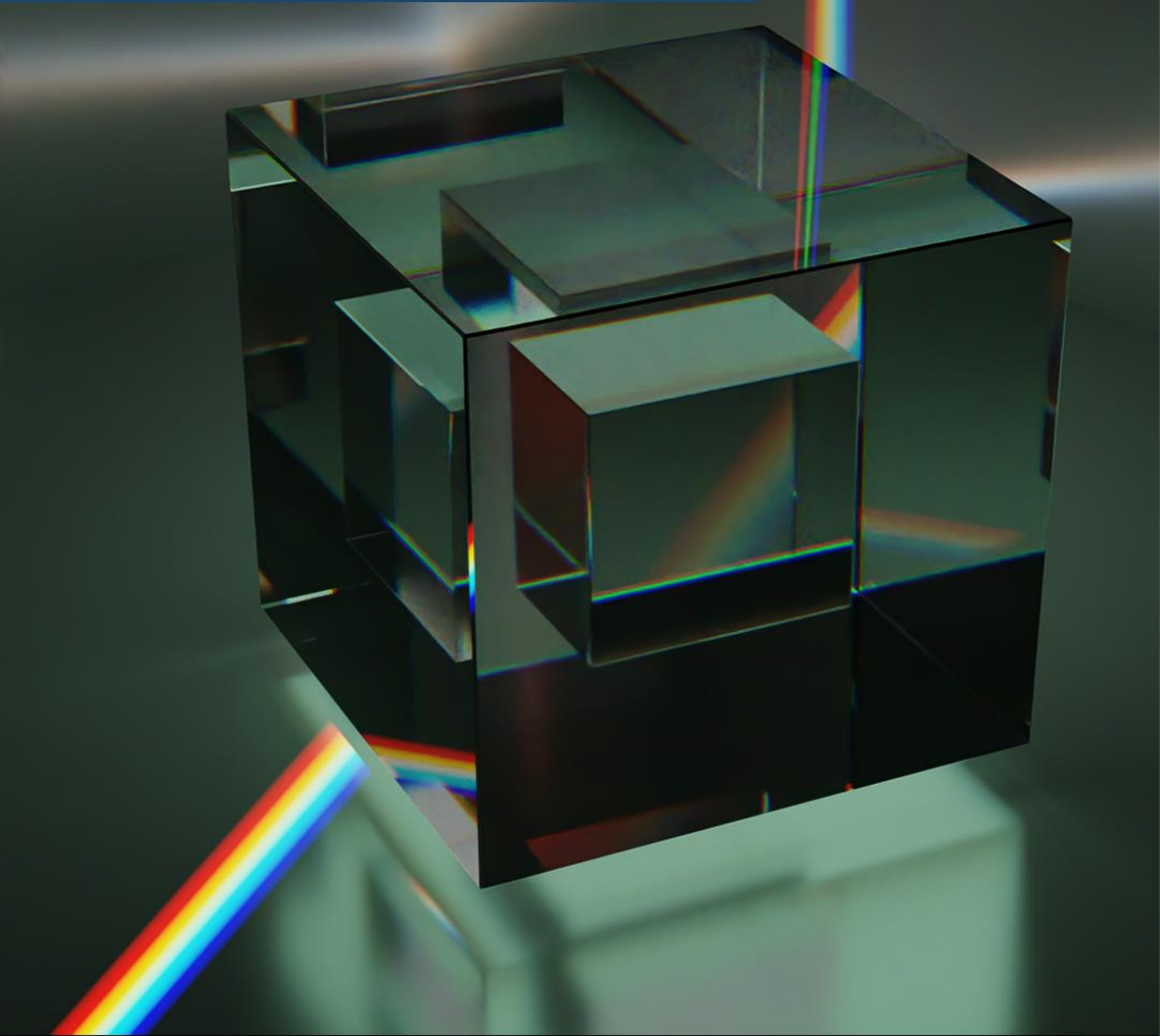
DECX

A High Performance Scientific computing library

Hardwares Supported by Intel ® and Nvidia ®

Git Repository Available : https://github.com/param0037/DECX.git

By Wayne Anderson on 16/04/2023



Author : Wayne Anderson

user manual

April 2021

Introduction

1. DECX

Computer science has been developing quickly in the past few decades. People tried to search for solutions for complex problems using computers, such as simulations, image processing, digital signal processing, etc. Nowadays, artificial intelligence is getting more and more engaged in many fields. As the processed data becomes larger, the computational ability of hardware should be increased to guarantee that the program will be done within a limited duration or even in real-time processing. Besides, the performance of software should be increased as well. Many high-performance computational and data processing libraries are constructed, such as OpenBlas, OpenCV, TensorFlow, PyTorch, etc.

DECX is a high-performance scientific computation library, several modules are included: core component, linear algebra (BLAS), digital signal processing (DSP), computer vision (CV), and neural network operators (NN). Each module is supported by both common host platforms (Intel® and AMD®) and Nvidia® devices that support CUDA and can be compiled separately to different binaries. The library is completely open-source and free. All the source codes, including APIs, documents and samples are available on GitHub [1]. The repository link is given by:
https://github.com/param0037/DECX.git.

2. Targeted Users

The targeted users are identified in a wide range. Anyone who is engaged in relative fields can be one of them. They can be university students, who are busying dealing with their coursework assignments; they can also be software companies, waiting for a quick foundation for the further developments. For students, DECX is friendly for beginners. DECX hides all the complex logic and mathematical algorithms for the users but provides some simple interfaces. Any learners including students can learn how it works easily and integrate this subprogram into their work. For companies, DECX is free and open-source, which saves considerable cost. The library is designed in modules, which is convenient for developers to customize and make it suitable for their products.

3. Structure of Article

The aim of this manual is introducing DECX, the fast, scientific computation library, and its APIs. The article first gives a brief account on the project DECX and the project background and outcomes. In the first chapter, the components and their dependent relationships are clarified. According to the content introduced, users are able to customize this library to their works. The following chapter introduces some key designs used in DECX internal realizations, which gives users a better understanding on how DECX works underneath the binary files. In this way, users are able to cooperate with DECX efficiently and create their program with a higher performance. The remaining parts of the article mainly introduces all the interfaces of DECX, including interface functions, classes, enumerators, etc. In this part, considering the beginners who have not studied advanced mathematics, mathematic principles of some APIs, matrix multiplication, filters, for example, are also introduced.

Warnings

1. Mac OS has ended its cooperation with Nvidia, which means no suitable driver is provided anymore, DECX is not supporting and can't support Mac OS.

2. DECX can only call one GPU on the machine. If the machine has multiple GPUs, these GPUs might not be able to work cooperatively. Thus, DECX may not be a good choice for context with multiple GPOUs. DECX is for those personal not-too-large projects. Besides that, the accuracy of the results DECX APIs calculate may not be as high as other famous libraries like TensorFlow, PyTorch, etc. Hopefully there will be enough time for the developer, which is Mr. Wayne Anderson, to make further developments.

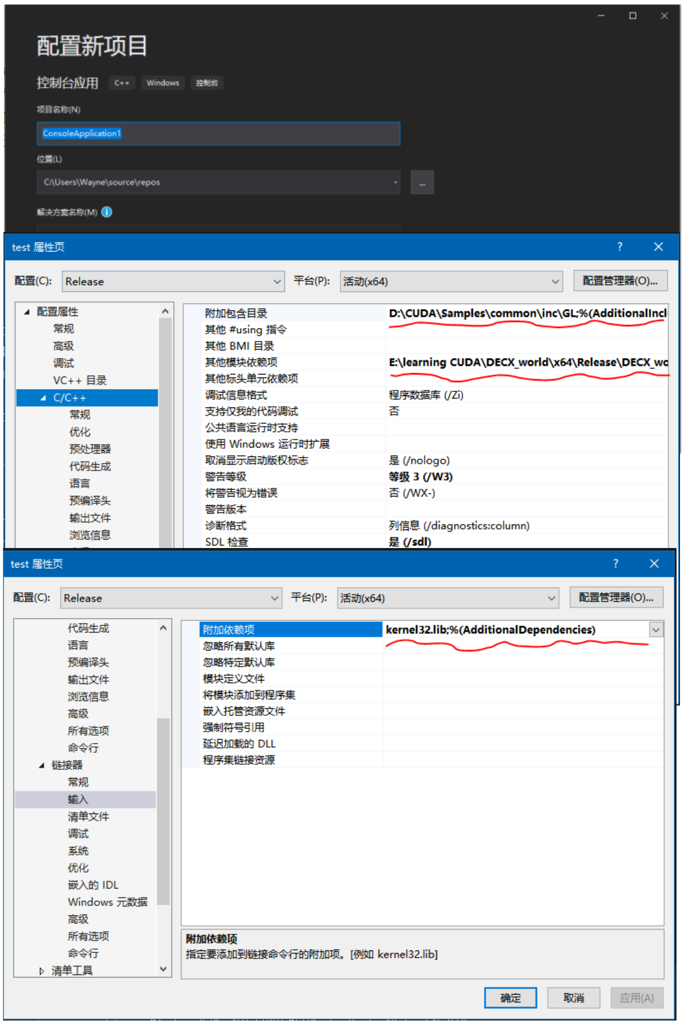
3. The computability of devices DECX supports varies from 3.0 to 7.5. However, Tensor Core is not supported yet, For the platform with computability higher than 7.5, there is no guarantee that DECX function properly. Since no test has been conducted on these platforms yet.

Table of Contents

1.Installing

1.Windows

1. As I mentioned before, this library can only be used under C++11 and above, in use of MSVC compiler toolchain. So, you need to install Microsoft Visual Studio before installation.
2. After installation of Microsoft Visual Studio, create a new win32 console project.
3. Open the project properties menu. Select the directory of the static link library "../DECX/bin/x64/DECX_world.lib" on the sheet "Additional Dependency". Then select the directory of dynamic link library "../DECX/include" on the sheet "Additional Including".
4. Open linker->input sheet. Then select the directory of the static link library "../DECX/bin/x64/DECX_world.lib" on the sheet "Additional Dependency". Close the property menu. Additionally, you can include this static link library by coding "#pragma comment(lib, "lib directory")"
5. Finally, copy "DECX/bin/x64/



项目名称(N): ConsoleApplication1

位置(L): C:\Users\Wayne\source\repos

配置(C): Release 平台(P): 活动(x64)

附加包含目录	D:\CUDA\Samples\common\inc\GL;%(AdditionalIncl
其他 #using 指令	
其他 BMI 目录	
其他模块依赖项	E:\learning CUDA\DECX_world\x64\Release\DECX_wo
其他标头单元依赖项	
调试信息格式	程序数据库 (/Zi)
支持仅我的代码调试	否
公共语言运行时支持	
使用 Windows 运行时扩展	
取消显示启动版权标志	是 (/nologo)
警告等级	等级 3 (/W3)
将警告视为错误	否 (/WX-)
警告版本	
诊断格式	列信息 (/diagnostics:column)
SDL 检查	是 (/sdl)

配置(C): Release 平台(P): 活动(x64)

附加依赖项	kernel32.lib;%(AdditionalDependencies)
忽略所有默认库	
忽略特定默认库	
模块定义文件	
将模块添加到程序集	
嵌入托管资源文件	
强制符号引用	
延迟加载的 DLL	
程序集链接资源	

附加依赖项
指定要添加到链接命令行的附加项。[例如 kernel32.lib]

2. Linux

1. Download the files.

2.Compiling

2.1 Windows

2.1.1 Compile using MS Visual Studio

After downloading or cloning the source codes of DECX, you can start compiling on your own platform. You have to download Microsoft Visual Studio and install it first since on Windows, Nvidia compiler only work with MSVC. After that, you should install all the required libraries need in compilation. These libraries are listed below:

2.1.2 Compile using CMake

1. Install Microsoft compiler MSVC of version above 16 by installing MS Visual Studio.
2. Install Nvidia CUDA development toolkit. Version 10.2 or above is recommended.
3. Make sure that MSVC and NVCC (CUDA compiler) are properly configured.
4. Install CMake. Version around 3.24 is recommended. Since the other versions of CMake might be incompatible with the codes in CMakeLists.txt.
5. Open Windows cmd.exe and enter the directory where the source codes locate.
6. Enter command “config”, and wait until the configuration complete.
7. Enter command “build”, and wait until the compiling finish.
8. Six dynamically link libraries (.dll files) and statically linkage files (.lib files) can be found at “project_dir/bin/x64/release”.

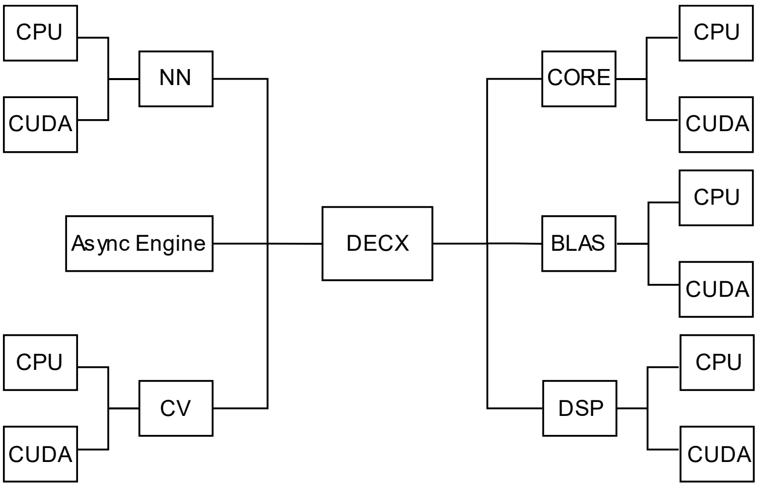
2.2 Linux

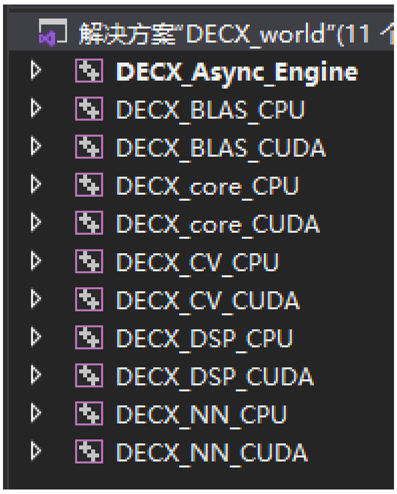
The project is recommended to be compiled via CMake. The following steps are:

1. Install GNU compiler Mingw. Make sure it supports C++14 syntax.
2. Install Nvidia CUDA development toolkit. Version 10.2 or above is recommended.
3. Make sure that MSVC and NVCC (CUDA compiler) are properly configured.
4. Install CMake. Version around 3.24 is recommended. Since the other versions of CMake might be incompatible with the codes in CMakeLists.txt.
5. Open terminal and enter the directory where the source codes locate.
6. Enter command “./config.sh”, and wait until the configuration complete.
7. Enter command “./build.sh”, and wait until the compiling finish.
8. Six shared objects (.so files) can be found at “project_dir/bin /release”.

3. Internal Design

3.1 Structure Flowgraph





解决方案"DECX_world"(11

- DECX_Async_Engine
- DECX_BLAS_CPU
- DECX_BLAS_CUDA
- DECX_core_CPU
- DECX_core_CUDA
- DECX_CV_CPU
- DECX_CV_CUDA
- DECX_DSP_CPU
- DECX_DSP_CUDA
- DECX_NN_CPU
- DECX_NN_CUDA

The figure shows the outline structure of DECX

DECX has several components and some of them are optional. Users can customize the installation and dependencies when they develop applications based on DECX. Core (including _CPU term and _CUDA term) and asynchronous engine are the core of DECX if user works on both CPUs and CUDA-embedded devices. Instead, only DECX_core_CPU is required when working with CPU platform (no CUDA-embedded device).

1. DECX_core_CPU
   DECX_cpre_CPU is the most essential component, which manages the memory pools (Please refer to chapter3.3 *Memory management* for more details), threading (Please refer to chapter3.4 *Thread management* for more details), etc. The host-related classes (Please refer to chapter4 *Objects and classes* for more details) are also defined and implemented in this module. This component is compulsory for users to install DECX.
2. DECX_core_CUDA
   This term is necessary if the CUDA based functions are used (Please refer to chapter3.5 *Naming rules* for more details).
3. DECX_BLAS_CPU
   CPU based linear algebra functions are provided in this component.
4. DECX_BLAS_CUDA
   CUDA based linear algebra functions are provided in this component.

The dependencies of each component is given in figure 3.1.1.

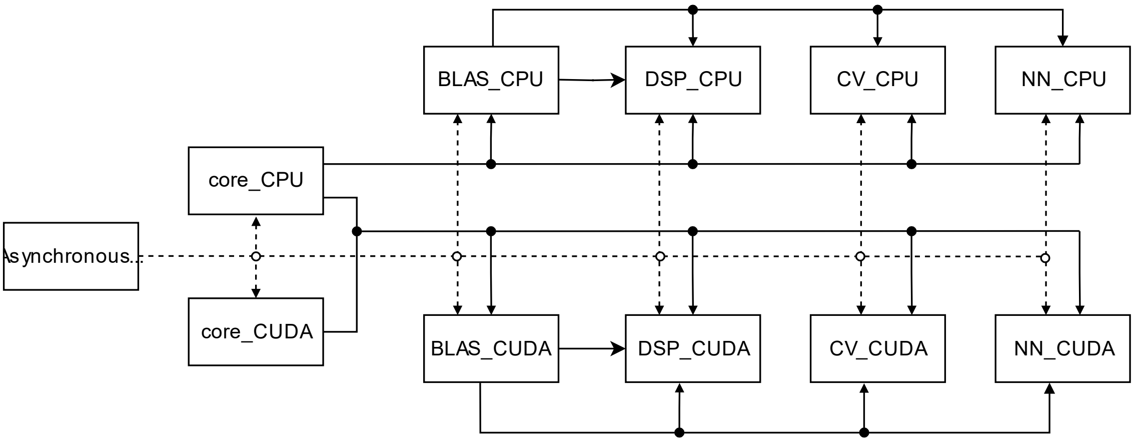


Figure 3.1.1 Dependencies of each component

3.2 Nvidia® CUDA

CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

CUDA is designed to work with programming languages such as C, C++, and Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphics programming. CUDA-powered GPUs also support programming frameworks such as OpenMP, OpenACC and OpenCL; and HIP by compiling such code to CUDA.

CUDA was created by When it was first introduced, the name was an acronym for Compute Unified Device Architecture,[5] but Nvidia later dropped the common use of the acronym.

3.3 Memory Management

The structure of memory pools is shown in Figure 3.3.1.

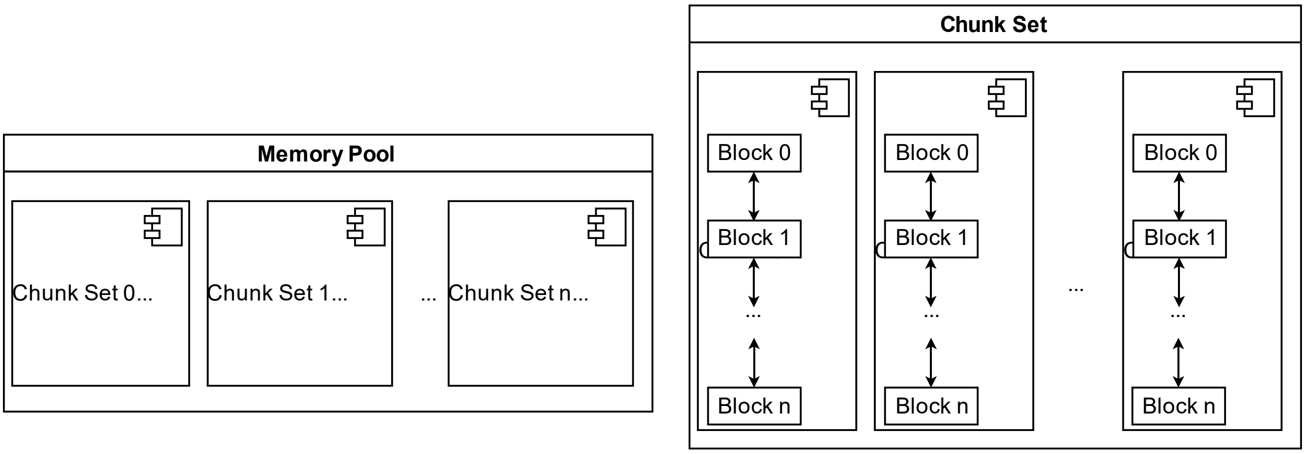


Figure 3.3.1 Structure of memory pools

Memory pools are necessary for the library. There will always be the case when users frequently create and release large memory spaces, Matrix, vector, for example. Memory pools help recycle and reuse the memory spaces and prevent calling system memory allocation APIs (malloc(), cudaDeviceMalloc(), etc.) frequently. Since these APIs are time-costing, with the help of memory pools, large data blocks are able to allocate and deallocate flexibly.

There are three types of memory pools in DECX, classified by memory types (pageable memory and page-locked memory on host, device memory on GPU). Each memory pool has the same structure and is divided into three ranks shown by three red rectangles in different shapes. The three memory pools are created when the program starts.

In each block structure, the number of references is recorded When it decreases to zero, this memory block will be automatically labeled as idle and deallocate. For example, when user is requesting a soft copy of a data block, a reference of it will be created by increasing the reference number by 1. System returns a new pointer pointing towards the same address. Deallocating one of the references will not deallocate the block, unless there is no reference to this data block (reference number is zero).

In DECX, for a complete compilation, there are two memory pools, which are on host memory and device memory respectively. The two memory pools are realized in DECX_core_CPU and DECX_core_CUDA, respectively. They are independent. On the platforms which have only central processing unit (CPU), only host memory pool is used. On the platforms equipped with CUDA supported devices, memory pools of both host and device are used.

3.4 Thread Management

Thread management is accomplished by thread pool. Similar to the memory pools, thread pools create the required number of threads from the very beginning of the program, and hold them if no task is coming. If there are tasks required to be processed, the threads will be invoked. Operating systems provides their basic threading APIs, for example, thread creation, joining the threads, etc. However, calling these APIs is also expensive. In the functional modules of this library, threads are frequently forked and joined, which costing lots of time if using the preliminary APIs for threading. Hence, thread pool is essential in DECX.

Figure 3.4.1 shows the structure of the thread pool in DECX.

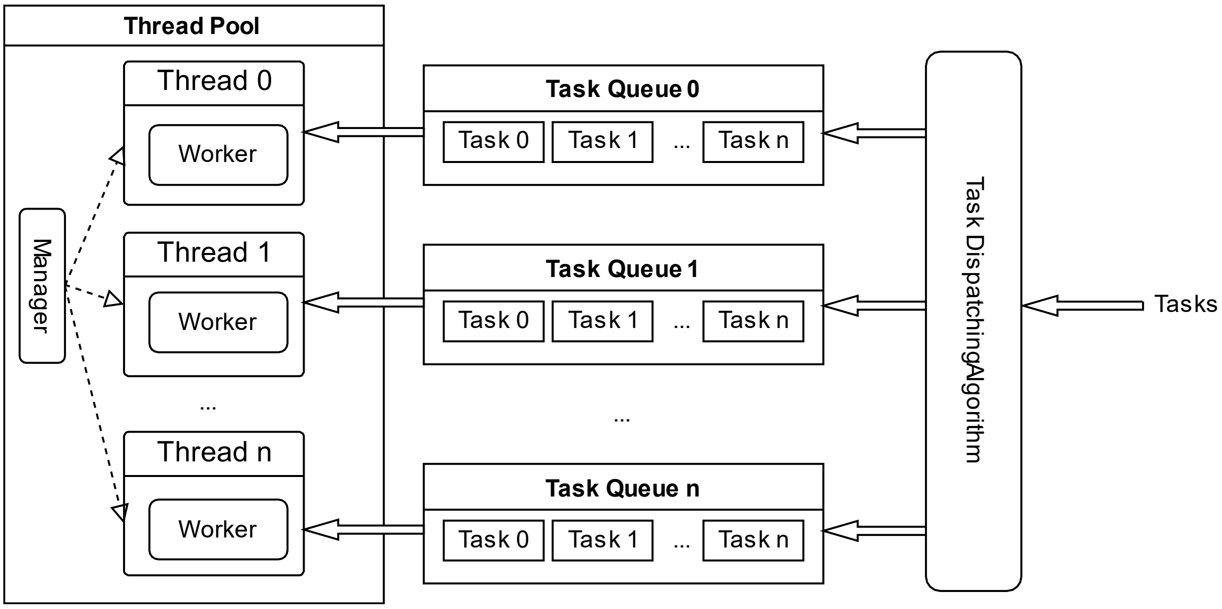


Figure 3.4.1 Structure of thread pool

For the thread pool in DECX, lock-free design is used to improve the performance. Each thread gets tasks from their own task queue, since no read or write address overlapping exists between threads. Since no lock is required. To equalize the load of each thread, the upcoming tasks are dispatched by a task dispatching algorithm, which always searches the thread which has minimum number of tasks waiting in its queue.

In the thread pool, all the threads are managed by a manager function running on the main thread of the program. The manager function helps dispatching tasks to each thread by running the task dispatching algorithm. The manager also keeps listening to the requirements from outer domain. Such as creating a new thread or destroying a thread. Additionally, the main thread is also responsible for waking up other sub threads if necessary, for example, when their task queue is no longer empty.

For each thread, a worker function is maintained since it is created. The worker function receives the tasks from the task queue. When the queue is empty, the worker will be sleeping. When the empty queue loads tasks from the task dispatching function, the corresponding thread will be invoked by the manager of the thread pool.

3.5 Naming Rules

DECX provides interfaces in C++ environment. Hence, different namespaces are used in the header files to identify different functions. Table 3.5.1 provides representations of each namespace.

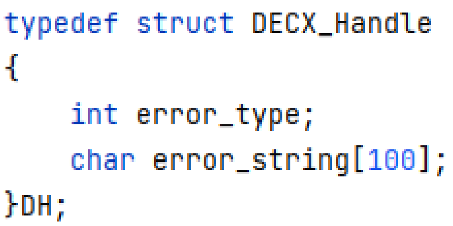
1st namespace	statement	2nd namespace	statement	3rd namespace	Statement
de::	All the interfaces (including classes, structure, enumerate, etc.) are clarified underneath this namespace.	**N/A**	General functions and objects.	**cpu::**	CPU based functions.
				cuda::	CUDA based functions.
		signal::	Functions and objects related to digital signal processing.	**cpu::**	CPU based functions.
				cuda::	CUDA based functions.
		vis::	Functions and objects related to computer vision.	**cpu::**	CPU based functions.
				cuda::	CUDA based functions.
		nn::	Functions and objects related to neural network.	**cpu::**	CPU based functions.
				cuda::	CUDA based functions.

Table 3.5.1 representations of each namespace

4. Core Functions

5. Objects and Classes

5.1. Handle



```
typedef struct DECX_Handle
{
    int error_type;
    char error_string[100];
}DH;
```

This structure is open to users, and contains the runtime errors messages, including error type and error string. The error string describes the error briefly.

Member *error_type* denotes the types of errors or warnings. See chapter 4.2 (*Error or warning types*) for more information.

Member *error_string* records the short description of a specific type of error or warning.

5.2. Error or warning types

The error types are defined in an enumerate class and each type is mapped to a specific hex number.

Please notice that the red words denote errors, the green word denotes successful condition, and the yellow words denote warnings.

1 DECX_SUCCESS

Number 0x00. No error occurs. The error code is 0x00. However please notice that warning might occur even process returns a handle with error type of **DECX_SUCCESS**.

2 DECX_FAIL_not_init

Number 0x01. User does not initialize DECX information before calling any of the APIs. Initialization includes CPU information and CUDA runtime information. Please call de::InitCuda() and de::InitCPUInfo() before using any function in DECX.

3 DECX_FAIL_FFT_error_length

Number 0x02. The length of vector or the dimensions of matrix in fast Fourier transform should be able to divided into one of the factors including 2,3, 5 and 7. Otherwise, process returns a handle with error type of **DECX_FAIL_FFT_error_length**.

4 DECX_FAIL_DimsNotMatching

Number 0x03. The dimensions of the input object are not matched the requirements.

5 DECX_FAIL_Complex_comparing

Number 0x04. Comparison between two complex numbers is illegal.

6 DECX_FAIL_ConvBadKernel

Number 0x05. The dimension of the convolutional kernel is illegal. E.g., The width and height of the kernel in 2D convolution should be odd numbers.

7 DECX_FAIL_CVNLM_BadArea

Number 0x06. In Non-local Means filter (NLM), the dimensions of searching windows and template windows should meet the requirements. Please refer to chapter *introduction on NLM* for more information.

8 DECX_FAIL_ErrorFlag

Number 0x07. The flags need to be indicated in, e.g., matrix multiplication on half-precision floating point matrices indicating the calculation accuracy, and in extension indicating the extending type. Process return error in type of **DECX_FAIL_ErrorFlag** if user indicates the illegal flag.

9 DECX_FAIL_ALLOCATION

Number 0x08. Process return error in type of **DECX_FAIL_ALLOCATION** as long as the failure occurs on any of the memory allocation, which means that all types of memory allocations are included.

10 DECX_FAIL_CUDA_STREAM

Number 0x09. Process return error in type of **DECX_FAIL_CUDA_STREAM** as long as the failure occurs on accessing CUDA stream.

11 DECX_FAIL_CLASS_NOT_INIT

Number 0x0A. Process return error in type of **DECX_FAIL_CLASS_NOT_INIT** if user input an uninitialized object if initialization is needed. An uninitialized object is the object constructed via default constructor.

12 DECX_FAIL_TYPE_MOT_MATCH

Number 0x0B. Process return error in type of **DECX_FAIL_TYPE_MOT_MATCH** if the data type of input object is not matched the requirement.

13 DECX_FAIL_INVALID_PARAM

Number 0x0C. Process return error in type of **DECX_FAIL_INVALID_PARAM** if the input parameter is not matched the requirement.

14 DECX_FAIL_IMAGE_LOAD_FAILED

Number 0x0D. This error type is only considered when loading an image from disk. Process returns error if failure occurs on file loading, e.g., file does not exist.

15 DECX_FAIL_CUDA_EVENT

Number 0x0E. Process return error in type of **DECX_FAIL_CUDA_EVENT** as long as the failure occurs on accessing CUDA event.

16 MULTIPLE_REFERENCES

Number 0x0F. Please note that this flag is not the error but warning. When deallocate a memory that is referred multiple times, process returns error in type of **MULTIPLE_REFERENCES.** Please make sure that the other objects that refer to the same memory block are useless, or the valuable data will be lost.

17 CPU_HYPER_THREADING

Number 0x10. When the number of the concurrent threads exceed that of hardware concurrency, process returns error in type of **CPU_HYPER_THREADING.** Please avoid hyper-threading the CPU in case of the performance loss.

18 MEMCPY_DIFFERENT_TYPES

Number 0x11. No error will occur if the sizes of each element of host object and device object are the same. However, if the data types are different, this warning will be yielded since the data will be reinterpreted as other type after copying.

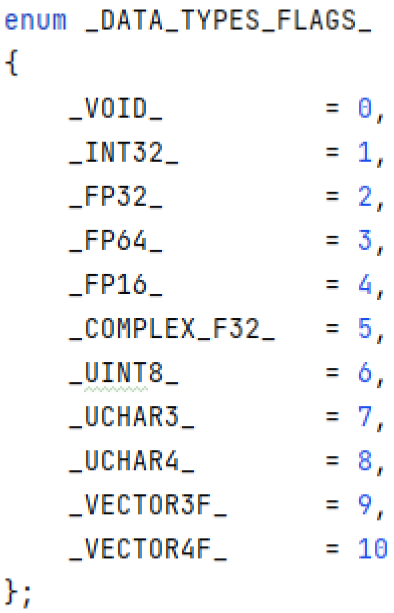
19 DECX_FAIL_MEMCPY_OVERRANGED

Number 0x12. Process returns **DECX_FAIL_MEMCPY_OVERRANGED** if the indicated copy size(s) exceed(s) the source object.

20 DECX_FAIL_UNSUPPORTED_TYPE

Number 0x13. Process returns **DECX_FAIL_UNSUPPORTED_TYPE** if the type of input object(s) is(are) unsupported.

5.3. Supported Data types



```
enum _DATA_TYPES_FLAGS_
{
    _VOID_          = 0,
    _INT32_         = 1,
    _FP32_          = 2,
    _FP64_          = 3,
    _FP16_          = 4,
    _COMPLEX_F32_   = 5,
    _UINT8_         = 6,
    _UCHAR3_        = 7,
    _UCHAR4_        = 8,
    _VECTOR3F_      = 9,
    _VECTOR4F_      = 10
};
```

1 _VOID_ : Not representing any type, only used in initialization of an object by calling its default constructor.

2 _INT32_: 32-bit integer, following IEEE 754 standard.

3 _FP32_: 32-bit floating point number, following IEEE 754 standard.

4 _FP64_: 64-bit floating point number, following IEEE 754 standard.

5 _FP16_: 16-bit floating point number, as known as “half”, following IEEE 754-2008 standard.

6 _COMPLEX_F32_: The complex number consists of two 32-bit floating point numbers, aligned to 2-bytes.

7 _UINT8_: 8-bit unsigned integer, simple binary storing.

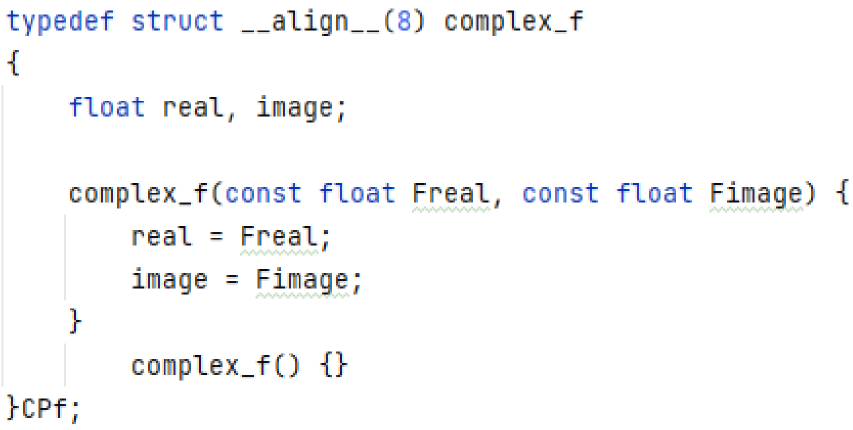
8 _UCHAR3_: The object consists of three 8-bit unsigned integers, but is aligned to 4-bytes.

9 _UCHAR4_: The object consists of four 8-bit unsigned integers, aligned to 4-bytes.

10 _VECTOR3F_: The object consists of three 32-bit floating point numbers, aligned to 16-bytes.

11 _VECTOR4F_: The object consists of four 32-bit floating point numbers, aligned to 16-bytes.

5.4. 32-bit floating point complex number

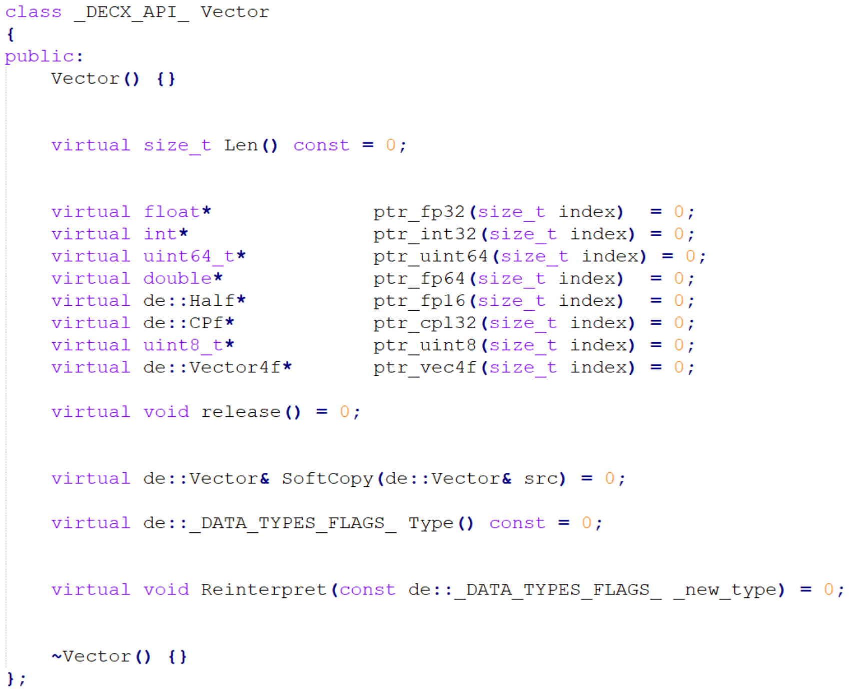


```
typedef struct __align__(8) complex_f
{
    float real, image;

    complex_f(const float Freal, const float Fimage) {
        real = Freal;
        image = Fimage;
    }
        complex_f() {}
}CPf;
```

The complex structure consists of two 32-bit floating point numbers, taking up 8 bytes of space. Member *real* denotes the real part of the complex number, while member *image* denotes the imaginary part. All the members are visible to user. There are two constructors provided. Please notice that the default constructor is not recommended, since it won’t initialize any of the members.

5.5. Vector



```
class _DECX_API_ Vector
{
public:
    Vector() {}


    virtual size_t Len() const = 0;


    virtual float*              ptr_fp32(size_t index)  = 0;
    virtual int*                ptr_int32(size_t index) = 0;
    virtual uint64_t*           ptr_uint64(size_t index)  = 0;
    virtual double*             ptr_fp64(size_t index)  = 0;
    virtual de::Half*           ptr_fp16(size_t index)  = 0;
    virtual de::CPf*            ptr_cpl32(size_t index) = 0;
    virtual uint8_t*            ptr_uint8(size_t index) = 0;
    virtual de::Vector4f*       ptr_vec4f(size_t index) = 0;

    virtual void release() = 0;


    virtual de::Vector& SoftCopy(de::Vector& src) = 0;

    virtual de::_DATA_TYPES_FLAGS_ Type() const = 0;


    virtual void Reinterpret(const de::_DATA_TYPES_FLAGS_ _new_type) = 0;


    ~Vector() {}
};
```

1. Introduction

Vector is a 1D array in both prospective of physical memory and abstraction layout. In the project DECX, de::Vector is only a fundamental class that is visible to users. In source codes, a class that inherits de::Vector is created in the internal namespace named decx. In that class, I designed a data structure to store data for de::Vector. Considered that access the memory in vectorization help accelerate memory access, when user call the constructing function in de::Vector, system will automatically allocate a memory in length of N, where N can be divided by vectorized numbers(2, 4, 8, 16 and so on, e.g.) into integers, where N >= required length.

2. Data Structure

Figure 4.5.1 shows the memory layout of de::Vector.

Data layout on RAM

As the figure shown above, the number of active elements is defined by the users, the margin is full of inactive elements (usually the zeros, in case of affecting the calculation results). Thus, according to the figure, the total length of the data space of de::Vector is 20 in prospective of RAM.

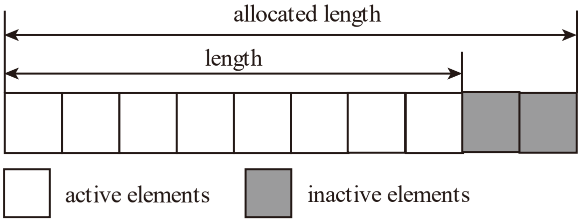


Figure 4.5.1 memory layout of de::vector

3. Supporting Types:

All the data types mentioned in chapter 6.1.4 are supported. See chapter 6.1.4 *Supported Data Types* for more information.

4. Member functions:

```
virtual size_t Len() const = 0
```

This function returns the length of vector. No parameter is needed. Note: the returned length is not the length of the internal physical memory.

```
virtual float* ptr_fp32(size_t index) = 0
```

This function returns a 32-bit floating point pointer of the indicated index. Parameter *index* is the location of the element in the vector. Note, no error will be printed if the index is out of range, accessing the pointer in this case might crash the process.

```
virtual int* ptr_int32(size_t index) = 0
```

This function returns a 32-bit integer pointer of the indicated index. Parameter *index* is the location of the element in the vector. Note, no error will be printed if the index is out of range, accessing the pointer in this case might crash the process.

```
virtual double* ptr_fp64(size_t index) = 0
```

This function returns a 64-bit floating point pointer of the indicated index. Parameter *index* is the location of the element in the vector. Note, no error will be printed if the index is out of range, accessing the pointer in this case might crash the process.

```
virtual de::Half* ptr_fp16(size_t index) = 0
```

This function returns a 16-bit floating point pointer of the indicated index. Parameter *index* is the location of the element in the vector. Note, no error will be printed if the index is out of range, accessing the pointer in this case might crash the process.

```
virtual de::CPf* ptr_cpl32(size_t index) = 0
```

This function returns a pointer in type of de::Complex_f of the indicated index. Parameter *index* is the location of the element in the vector. Note, no error will be printed if the index is out of range, accessing the pointer in this case might crash the process.

```
virtual uint8_t* ptr_uint8(size_t index) = 0
```

This function returns a pointer in type of 8-bit unsigned integer of the indicated index. Parameter *index* is the location of the element in the vector. Note, no error will be printed if the index is out of range, accessing the pointer in this case might crash the process.

```
virtual void release() = 0
```

Quickly release the vector, by unregistering the memory flag in the corresponding memory pool (see chapter 5.3 *Memory management* for more information). No internal free() or delete[] is called.

```
virtual de::Vector& SoftCopy(de::Vector& src) = 0
```

Soft copy the object to another one. No memory copying is used but only pointer mapping, by increasing the reference number in the memory pool (see chapter 5.3 *Memory management* for more information).

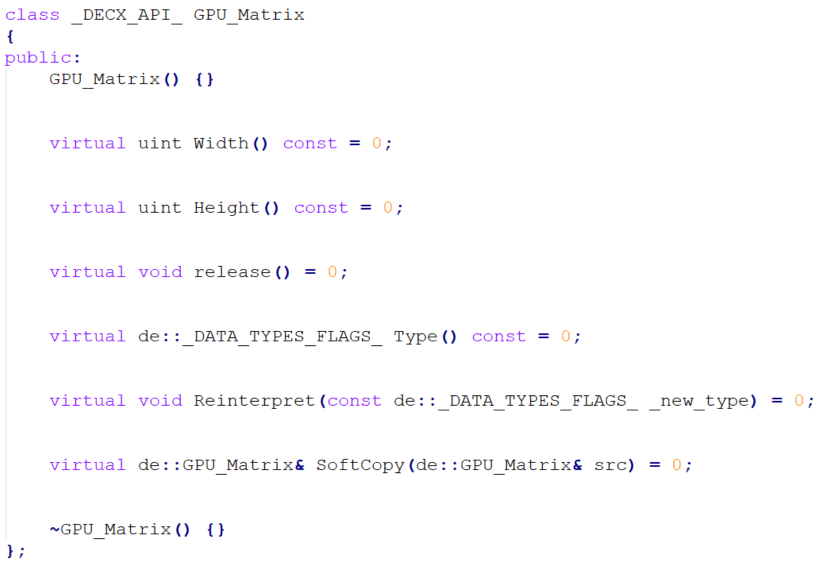
```
virtual de::_DATA_TYPE_FLAGS_ Type() const = 0
```

Get the data type label of vector. The definition of all labels is introduced in chapter 4.3 *Supported Data Types*.

```
virtual void Reinterpret(const de::_DATA_TYPE_FLAGS_ _new_type) = 0
```

Reinterpret the data type without changing any of the data, including its space length. This process only changes the data type label.

5.6. GPU_Vector



```
class _DECX_API_ GPU_Matrix
{
public:
    GPU_Matrix() {}

    virtual uint Width() const = 0;

    virtual uint Height() const = 0;

    virtual void release() = 0;

    virtual de::_DATA_TYPES_FLAGS_ Type() const = 0;

    virtual void Reinterpret(const de::_DATA_TYPES_FLAGS_ _new_type) = 0;

    virtual de::GPU_Matrix& SoftCopy(de::GPU_Matrix& src) = 0;

    ~GPU_Matrix() {}
};
```

1. Introduction:

The class GPU_Vector is basically the same as Vector. However, its pointer is referred to device (GPU) memory, where is inaccessible in host domain (non-CUDA kernel). Hence, no access for the data is provided. To read or write the device memory, function de::Memcpy() or de::MemcpyLinear() should be called to transfer data between host and device. See chapter XXX for more information.

2. Data Structure

The data storage and physical memory layout are the same as 4.6 de::Vector. See chapter 4.6 for more information. But please notice that the physical memory is located on device (GPU) memory.

3. Supporting Types:

All the data types mentioned in chapter 6.1.4 are supported. See chapter 6.1.4 for more information.

4. Member functions:

```
virtual size_t Len() const = 0
```

This function returns the length of vector. No parameter is needed. Note: the returned length is not the length of the internal physical memory.

```
virtual void release() = 0
```

Quickly release the vector, by unregistering the memory flag in the corresponding memory pool (see chapter 5.3 *Memory management* for more information). No free() or delete[] is called.

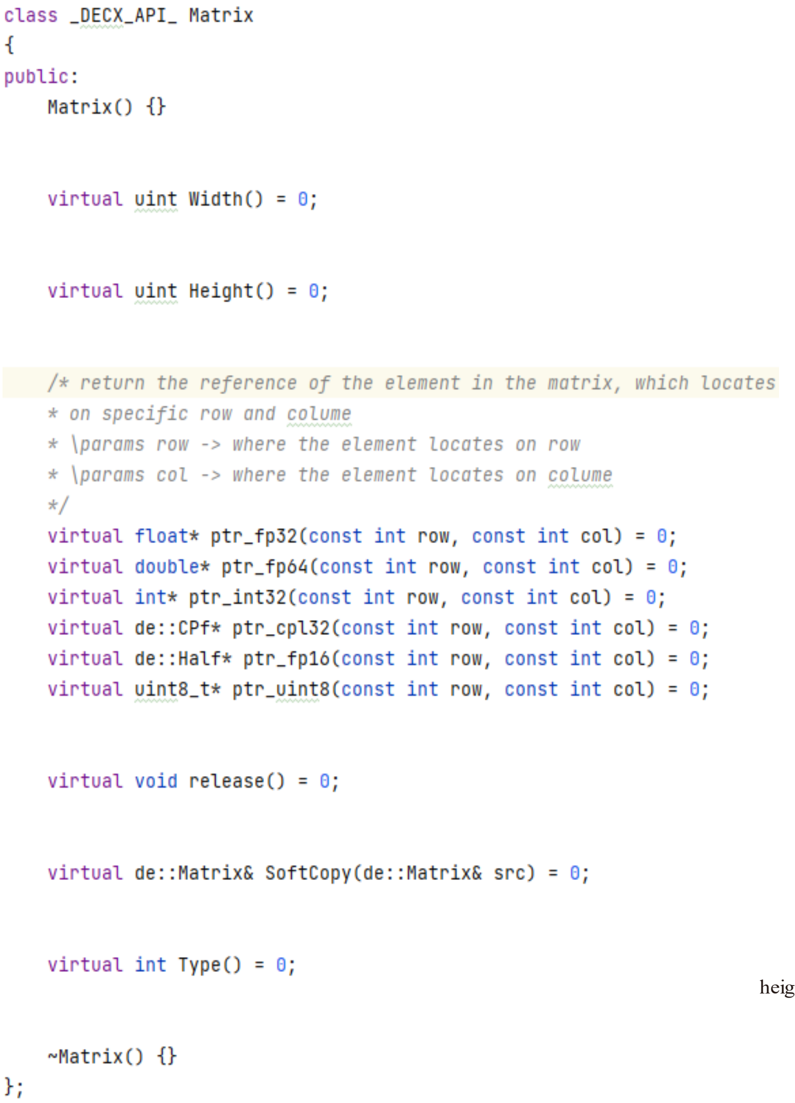
```
virtual de::GPU_Vector& SoftCopy(de::GPU_Vector& src) = 0
```

Soft copy the object to another one. No memory copying is used but only pointer mapping, by increasing the reference number in the memory pool (see chapter 5.3 *Memory management* for more information).

```
virtual int Type() const = 0
```

Get the data type label of vector. The definition of all labels is introduced in chapter 4.3 *Supported Data Types*.

5.7. Matrix



```
class _DECX_API_ Matrix
{
public:
    Matrix() {}


    virtual uint Width() = 0;


    virtual uint Height() = 0;


    /* return the reference of the element in the matrix, which locates
    * on specific row and colume
    * \params row -> where the element locates on row
    * \params col -> where the element locates on colume
    */
    virtual float* ptr_fp32(const int row, const int col) = 0;
    virtual double* ptr_fp64(const int row, const int col) = 0;
    virtual int* ptr_int32(const int row, const int col) = 0;
    virtual de::CPf* ptr_cpl32(const int row, const int col) = 0;
    virtual de::Half* ptr_fp16(const int row, const int col) = 0;
    virtual uint8_t* ptr_uint8(const int row, const int col) = 0;


    virtual void release() = 0;


    virtual de::Matrix& SoftCopy(de::Matrix& src) = 0;


    virtual int Type() = 0;


    ~Matrix() {}
};
```

The width of the active area denotes the width of the matrix. The physical width, also named as the pitch, is extended to some length so that the address of each row can be aligned and accelerations methods such as vectorization can be performed.

1. introduction

Matrix is a 2D data array in the prospective of abstraction layout. However, the physical address of each element is linear and consecutive (including the inactive elements, which will be introduced later). In project DECX, this class is only the fundamental class inherited by the invisible and internal class in the source codes. This class is visible to users. Please notice that it is a pure virtual class. And all functions are abstracted as virtual functions.

2. Data Structure

The following figure (4.7.1) shows the memory layout of de::Matrix.

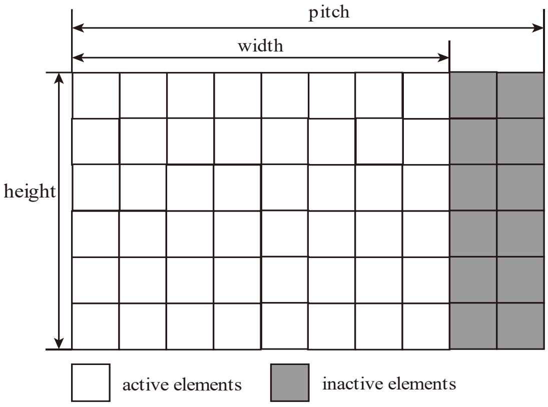


Figure 4.7.1 memory layout of de:;Matrix

3. Supporting Types:

All the data types mentioned in chapter 6.1.4 are supported. See chapter 6.1.4 for more information.

4. coordinate system for Matrix:

The coordinate system for matrix is shown in the figure below. X-axis lies on the width direction, and Y-axis lies on the height direction. The origin is located on the left and upper corner of the matrix, which is also the beginning address in physical memory of the matrix data array.

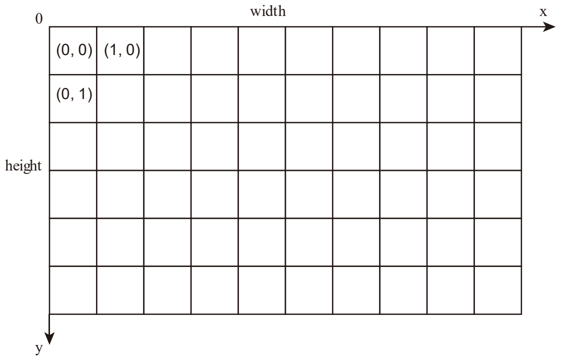


Figure xx coordinate system of 2D array in DECX

4. Member functions:

```
virtual uint Width() const = 0
```

Returns the width of the matrix. No parameter is needed. Note: the returned length is not the width of the internal physical memory (see *data storage* for more information).

```
virtual uint Height() const = 0
```

Returns the height of the matrix. No parameter is needed.

```
virtual float* ptr_fp32(const int row, const int col) = 0
```

Returns the pointer of 32-bit floating point number. Parameter *row* indicates the row order of the pointer position, which is y coordinate as well. (See chapter 4.7.4 *coordinate system for Matrix* for more information).

```
virtual double* ptr_fp64(const int row, const int col) = 0
```

Returns the pointer of 64-bit floating point number. Parameter *row* indicates the row order of the pointer position, which is y coordinate as well. (See chapter 4.7.4 *coordinate system for Matrix* for more information).

```
virtual int* ptr_int32(const int row, const int col) = 0
```

Returns the pointer of 32-bit integer. Parameter *row* indicates the row order of the pointer position, which is y coordinate as well. (Please refer to chapter 4.7.4 *coordinate system for Matrix* for more information).

```
virtual de::CPf* ptr_cpl32(const int row, const int col) = 0
```

Returns the pointer of complex number consists of two 32-bit floating point numbers. Parameter *row* indicates the row order of the pointer position, which is y coordinate as well. (See chapter 4.7.4 *coordinate system for Matrix* for more information).

```
virtual de::Half* ptr_fp16(const int row, const int col) = 0
```

Returns the pointer of 16-bit floating point number. Parameter *row* indicates the row order of the pointer position, which is y coordinate as well. (See chapter 4.7.4 *coordinate system for Matrix* for more information).

```
virtual uint8_t* ptr_uint8(const int row, const int col) = 0
```

Returns the pointer of 8-bit unsigned integer. Parameter *row* indicates the row order of the pointer position, which is y coordinate as well. (See chapter 4.7.4 *coordinate system for Matrix* for more information).

```
virtual void release() = 0
```

Quickly release the vector, by unregistering the memory flag in the corresponding memory pool (see chapter 5.3 *Memory management* for more information). No free() or delete[] is called.

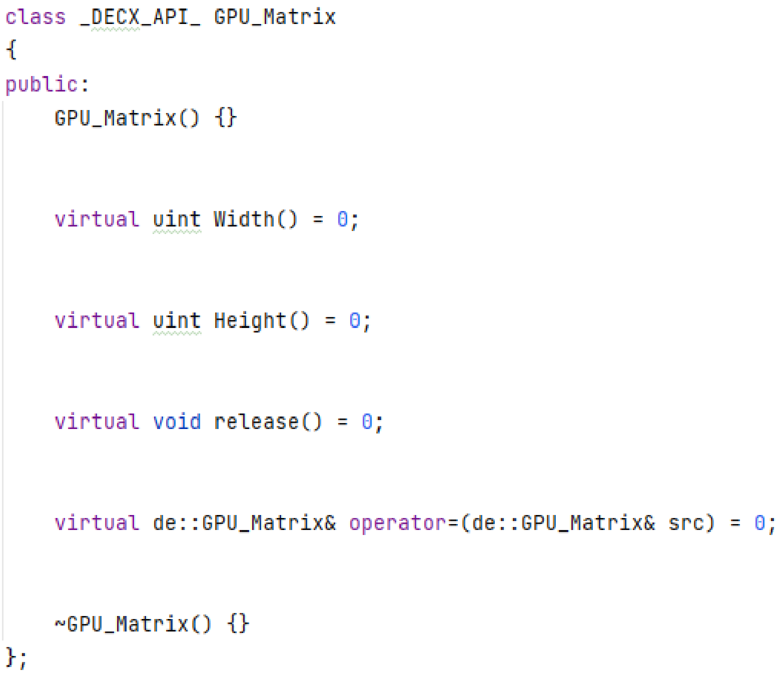
```
virtual de::Matrix& SoftCopy(de::Matrix& src) = 0
```

Soft copy the object to another one. No memory copying is used but only pointer mapping, by increasing the reference number in the memory pool (see chapter 5.3 *Memory management* for more information).

```
virtual int Type() const = 0
```

Get the data type label of vector. The definition of all labels is introduced in chapter 4.3 *Supported Data Types*.

5.8. GPU_Matrix



```
class _DECX_API_ GPU_Matrix
{
public:
    GPU_Matrix() {}


    virtual uint Width() = 0;


    virtual uint Height() = 0;


    virtual void release() = 0;


    virtual de::GPU_Matrix& operator=(de::GPU_Matrix& src) = 0;


    ~GPU_Matrix() {}
};
```

1. introduction

GPU_Matrix is a 2D data array in the prospective of abstraction layout. However, the physical address of each element is linear and consecutive (including the inactive elements, which will be introduced later). In project DECX, this class is only the fundamental class inherited by the invisible and internal class in the source codes. This class is visible to users. Please notice that it is a pure virtual class. And all functions are abstracted as virtual functions.

2. data storage

GPU_Matrix has the same physical memory layout as that of the object Matrix. Please refer to chapter 4.7 for more information.

3. Supporting Types:

All the data types mentioned in chapter 6.1.4 are supported. Please refer to chapter 6.1.4 for more information.

4. Member functions:

```
virtual uint Width() const = 0
```

Returns the width of the matrix. No parameter is needed. Note: the returned value is not the width of the internal physical memory (see *data storage* for more information).

```
virtual uint Height() const = 0
```

Returns the height of the matrix. No parameter is needed.

```
virtual void release() = 0
```

Quickly release the vector, by unregistering the memory flag in the corresponding memory pool (see chapter 5.3 *Memory management* for more information). No free() or delete[] is called.

```
virtual int Type() const = 0
```

Get the data type label of vector. The definition of all labels is introduced in chapter 4.3 *Supported Data Types.*

```
virtual de::GPU_Matrix& SoftCopy(de::GPU_Matrix& src) = 0
```

Soft copy the object to another one. No memory copying is used but only pointer mapping, by increasing the reference number in the memory pool (see chapter 5.3 *Memory management* for more information).

5.9. MatrixArray

5.10. GPU_MatrixArray

5.11. Tensor

5.12. GPU_Tensor

5.13. TensorArray

5.14. GPU_TensorArray

6. Basic process

6.1. Extension

1. Introduction

Extension process expands a data array on both sides of its each dimension, but keeps the original data unchanged. For example, a matrix with width and height of 1920 and 1080 respectively, is extended by a, b, c, and d on direction of top, bottom, left, and right respectively. The width and height of the output matrix will be $1920 + c + d$ and $1080 + a + b$, respectively.

After extending the data array, blank spaces are generated as well. Constant extension and reflected extension are two methods filling the blanks in the new array.

2. Table of supported types

Y: Supported.

N: Not supported.

N/A: (Temporarily) not available

TYPES		Vector	GPU_Vector	Matrix	GPU_Matrix	Tensor	GPU_Tensor
FP32	CPU	Y	N/A	Y	N/A	N/A	N/A
	CUDA	N/A	N/A	N/A	N/A	N/A	N/A
FP16	CPU	Y	N/A	Y	N/A	N/A	N/A
	CUDA	N/A	N/A	N/A	N/A	N/A	N/A
UINT8	CPU	Y	N/A	Y	N/A	N/A	N/A
	CUDA	N/A	N/A	N/A	N/A	N/A	N/A
FP64	CPU	Y	N/A	Y	N/A	N/A	N/A
	CUDA	N/A	N/A	N/A	N/A	N/A	N/A
INT32	CPU	N	N/A	N	N/A	N/A	N/A
	CUDA	N	N/A	N	N/A	N/A	N/A
CPf	CPU	N	N/A	N	N/A	N/A	N/A
	CUDA	N	N/A	N	N/A	N/A	N/A

Table 5.1.1 table of supported types of extension

3. APIs

Extend flags:

```
enum extend_label {
       _EXTEND_NONE_       = 0,
       _EXTEND_REFLECT_    = 1,
       _EXTEND_CONSTANT_   = 2,  };
```

Three methods provided for the filling of the blank areas.

1. _EXTEND_NONE_ and _EXTEND_CONSTANT_

For constant extension, a constant value is specified by user and all the blank spaces are filled up with the value. For example, a vector **v** is given:

$$\boldsymbol{v} = [1, 2, \dots n-2, n-1, n] \tag{5.3.1}$$

Assume **v** is extended by 2 elements using constant or zero(none) extension on both sides and produces $\boldsymbol{v}'$. The values of $\boldsymbol{v}'$ are:

$$\boldsymbol{v}' = [0, 0, 1, 2, \dots n-2, n-1, n, 0, 0] \tag{5.3.2}$$

2. _EXTEND_REFLECT_:

For reflected extension, the extended areas are the reflected version of the original data. Assume vector **v** mentioned in 5.3.1 is extended by 2 elements using reflected extension on both sides and produces $\boldsymbol{v}'$. The values of $\boldsymbol{v}'$ are:

$$\boldsymbol{v}' = [3, 2, 1, 2, \dots n-2, n-1, n, n-1, n-2] \tag{5.1.3}$$

APIs:

```
_DECX_API_ de::DH de::cpu::Extend(de::Vector& src, de::Vector& dst, const uint32_t left, const uint32_t right,
 const int border_type, void* val);
_DECX_API_ de::DH de::cpu::Extend(de::Matrix& src, de::Matrix& dst, const uint32_t left, const uint32_t right,
 const uint32_t top, const uint32_t bottom, const int border_type, void* val);
```

Provided by: DECX_BLAS_CPU and DECX_BLAS_CUDA.

Parameters:

src: The input vector or matrix to be extended. If the input object is not initialized, process returns handle with error of **DECX_FAIL_CLASS_NOT_INIT**.

dst: The output vector or matrix of the extension. No initialization is required, the properties of output object will be automatically set.

left: The number of elements the object extends on its left.

right: The number of elements the object extends on its right.

top: The number of elements the matrix extends on its top.

bottom: The number of elements the matrix extends on its bottom.

Val: The value used to fill up the blank when method **_EXTEND_CONSTANT_** is used

Return:

Process returns the handle carrying the runtime status, such as errors or warnings.

6.2. Type Converting

1. Introduction

DECX provides functions for type casting. Several types of conversions are supported, as shown in table 6.2.1.

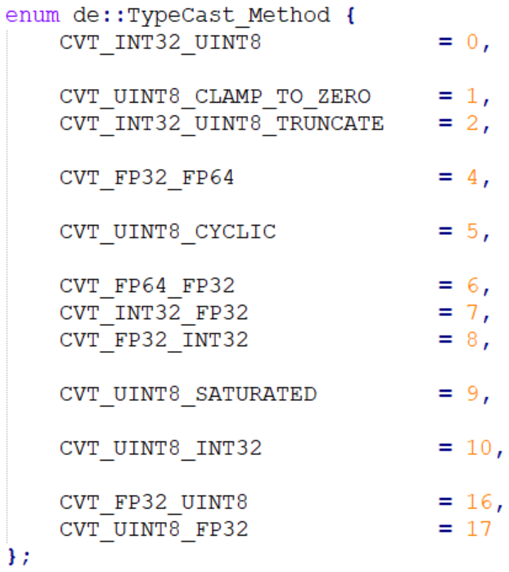
Conversions		Methods Flag(s)	Statements
From	To		
_FP32_	_FP64_	CVT_FP32_FP64	Convert 32-bit floating point to 64-bit one.
_FP64_	_FP32_	CVT_FP64_FP32	Convert 64-bit floating point to 32-bit one.
_FP32_	_UINT8_	CVT_FP32_UINT8 | CVT_UINT8_CYCLIC OR CVT_FP32_UINT8 | CVT_UINT8_CYCLIC | CVT_UINT8_CLAMP_TO_ZERO	The range of 8-bit unsigned int is [0,255]. However, the range of 32-bit floating point is much larger than that. For the floating-point value within [0,255], the conversion acts normally (The fractional part is abandoned). For the number x, which is larger than 255, the converted number returns to 0. If the number is less than 0, the converted result will be clamped to 0.
		CVT_FP32_UINT8 | CVT_UINT8_SATURATED OR CVT_FP32_UINT8 | CVT_UINT8_SATURATED | CVT_UINT8_CLAMP_TO_ZERO	For saturated conversion, for the floating-point value within [0,255], the conversion acts normally (The fractional part is abandoned). For the number x, which is larger than 255, the converted number is equal to 255. If the number is less than 0, the converted result will be clamped to 0.
INT32	_UINT8_	CVT_UINT8_INT32 | CVT_INT32_UINT8_TRUNCATE	Truncate the lowest 8 bits of 32-bit integer to get the converted unsigned 8-bit integer.
		CVT_UINT8_INT32 | CVT_INT32_UINT8_TRUNCATE | CVT_UINT8_CLAMP_TO_ZERO	Truncate the lowest 8 bits of 32-bit integer to get the converted unsigned 8-bit integer. For the negative 32-bit integer, the converted number will be clamped to 0.
		CVT_INT32_UINT8 | CVT_UINT8_CYCLIC OR CVT_INT32_UINT8 | CVT_UINT8_CYCLIC | CVT_UINT8_CLAMP_TO_ZERO	The same as conversions between 32-bit floating point number to 8-bit unsigned integers, for the number larger than 255 (maximum of 8-bit unsigned integer), the converted results returns to 0. The number less than 0 will be clamped to 0.
		CVT_INT32_UINT8 | CVT_UINT8_SATURATED	The same as conversions between 32-bit floating point number to 8-bit unsigned integers, for the

		OR CVT_INT32_UINT8 | CVT_UINT8_SATURATED | CVT_UINT8_CLAMP_TO_ZERO	number larger than 255 (maximum of 8-bit unsigned integer), the converted results is clamped to 255. The number less than 0 will be clamped to 0.
UINT8	_FP32_	CVT_UINT8_FP32	Convert unsigned 8-bit integer to 32-bit floating point.
UINT8	_INT32_	CVT_UINT8_INT32	Convert unsigned 8-bit integer to 32-bit signed integer.

Table 6.2.1 Supported conversion methods

2. APIs

The conversion method flags are provided as below:



```
enum de::TypeCast_Method {
    CVT_INT32_UINT8                 = 0,

    CVT_UINT8_CLAMP_TO_ZERO         = 1,
    CVT_INT32_UINT8_TRUNCATE        = 2,

    CVT_FP32_FP64                   = 4,

    CVT_UINT8_CYCLIC                = 5,

    CVT_FP64_FP32                   = 6,
    CVT_INT32_FP32                  = 7,
    CVT_FP32_INT32                  = 8,

    CVT_UINT8_SATURATED             = 9,

    CVT_UINT8_INT32                 = 10,

    CVT_FP32_UINT8                  = 16,
    CVT_UINT8_FP32                  = 17
};
```

All the flags are given in an enumerator under the primary namespace named “de” in the header file named *type_cast.h*.

The meanings and usages of the flags are given in table 6.2.1. If multiple flags are used to indicate the conversion method, please use operator *or* (“|”) only as the connection of the flags.

The functions are provided as below:

```
_DECX_API_ de::DH de::cpu::TypeCast(de::Vector& src, de::Vector& dst, const int cvt_method);

_DECX_API_ de::DH de::cpu::TypeCast(de::Matrix& src, de::Matrix& dst, const int cvt_method);

_DECX_API_ de::DH de::cpu::TypeCast(de::MatrixArray& src, de::MatrixArray& dst, const int cvt_meth
od);

_DECX_API_ de::DH de::cpu::TypeCast(de::Tensor& src, de::Tensor& dst, const int cvt_method);
```

7. Linear algebra

7.1. GEMM

1. Introduction

General matrix multiplication (GEMM) multiplies two matrix and adds the result with the third matrix. In matrix form, given a L by M matrix **A**, N by L matrix **B**, and N by M matrix **C**, N by M matrix **D** is denoted as the result of GEMM, each element of **D** is given by:

$$\boldsymbol{D}(i,j) = \sum_{k=0}^{L-1} \boldsymbol{A}(k,i) * \boldsymbol{B}(j,k) + \boldsymbol{C}(i,j) \tag{6.1.1}$$

Where $i \in [0, M-1], j \in [0, N-1]$, and $i \in N^{+}, k \in N^{+}$.

GEMM is important and widely used in many applications, such as neural networks and simulations. Since it is important to implement its high-speed version.

2. Table of supported types

Y: Supported.

N: Not supported.

N/A: (Temporarily) not available

TYPES	FP32		FP16		UINT8		FP64		INT32		CPf	
	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA
Matrix	Y	Y	N	Y	N	N	Y	N	N	N	Y	Y
GPU_Matrix	Y	Y	N	Y	N	N	Y	N	N	N	Y	Y

Table 5.1.1 table of supported types of extension

3. APIs

CUDA GEMM flags:

```
enum de::GEMM_properties {
       HALF_GEMM_DIRECT    = 0,
       HALF_GEMM_ACCURATE  = 1 };
```

1. HALF_GEMM_DIRECT:

   Directly compute the FMA (fast multiply and add) result of fp16, no conversion to fp32 for higher precision during the accumulating stage.

2. HALF_GEMM_ACCURATE:

After loading the fp16 data to register to ensure the global memory accessing throughput, kernel then converts the data to fp32, and executes FMA on fp32, improving the accuracy but slowing down the calculation. Finally, kernel converts the data back to fp16 (ignoring the overflow), storing the result back to global memory.

CUDA part:

```
_DECX_API_ de::DH de::cuda::GEMM(de::Matrix& A, de::Matrix& B, de::Matrix& dst);
_DECX_API_ de::DH de::cuda::GEMM(de::Matrix& A, de::Matrix& B, de::Matrix& C, de::Matrix& dst);
_DECX_API_ de::DH de::cuda::GEMM(de::GPU_Matrix& A, de::GPU_Matrix& B, de::GPU_Matrix& dst, const int flag);
_DECX_API_ de::DH de::cuda::GEMM(de::GPU_Matrix& A, de::GPU_Matrix& B, de::GPU_Matrix& C, de::GPU_Matrix& dst,
 const int flag);
```

CPU part:

```
_DECX_API_ de::DH de::cpu::GEMM(de::Matrix& A, de::Matrix& B, de::Matrix& dst);
_DECX_API_ de::DH de::cpu::GEMM(de::Matrix& A, de::Matrix& B, de::Matrix& C, de::Matrix& dst);
_DECX_API_ void de::cpu::GEMM_Async(de::Matrix& A, de::Matrix& B, de::Matrix& dst, de::DecxStream& S);
_DECX_API_ void de::cpu::GEMM_Async(de::Matrix& A, de::Matrix& B, de::Matrix& C, de::Matrix& dst, de::DecxStre
am& S);
```

Parameters:

A, B, C: The input matrices. If the input object is not initialized, process returns handle with error of **DECX_FAIL_CLASS_NOT_INIT**. According to the definition of matrix multiplication, assuming that matrix A is L_1 by M, and matrix B is L_2 by N, if $L_1 \neq L_2$, or the dimensions of matrix C is not N by M, process returns **DECX_FAIL_DimsNotMatching.**

dst: The result matrix of the matrix multiplication. No initialization is required, the properties of output object will be automatically set.

Flag: Only used when the input matrix A and B are both carrying fp16 data, indicating the fp16 calculation behaviors. Users can set the value of this flag to trade-off between throughputs and precision.

S: The asynchronous stream that the process binds to. Only used in asynchronous callings.

Return:

Process returns the handle carrying the runtime status, such as errors or warnings.

4. Benchmark results

7.2. Convolution

7.2.1. Filters

1. Introduction

Filters, in DECX context, denote the time-domain 1D and 2D convolutions. For 2D time-domain filter, assuming that **A** is the source matrix to be convoluted, **B** is the kernel and **C** is the output matrix.

$$\boldsymbol{C}(i,j)=\sum_{n=1}^{H_K}\sum_{m=1}^{W_K}A\left(i-\frac{H_k}{2}+n,j-\frac{W_K}{2}+m\right)\boldsymbol{B}(n,m) \tag{6.2.1}$$

Where, H_k and W_K denote the height and the width of the kernel, respectively. However, the indexes are out of range on the borders. Two methods are introduced for the border calculations.

1. _EXTEND_NONE_

   The borders are ignored. Considered the equation 6.2.1, the restriction will be:

$$i\epsilon N^+, i\epsilon\left[1,H-\frac{H_K}{2}\right]; j\epsilon N^+, j\epsilon\left[1,W-\frac{W_K}{2}\right] \tag{6.2.2}$$

2. _EXTEND_CONSTANT_

   The height and width of source matrix will be added by $\frac{H_K}{2}$, and $\frac{W_K}{2}$, respectively. The additional spaces will be filled with zeros. And a normal convolution ignoring the borders is conducted on the extended matrix after.

3. _EXTEND_REFLECT_

   The height and width of source matrix will be added by $\frac{H_K}{2}$, and $\frac{W_K}{2}$, respectively. The additional spaces will be fill with the values that are reflections of the source matrix. See chapter 5.1 (Extension) for more information. And a normal convolution ignoring the borders is conducted on the extended matrix after.

Sometimes, 2D filter is applied on multiple channels, for example, red, green, and blue channel in image processing. In this case, de::MatrixArray is used to represent a matrix with multiple channels. For single kernel 2D filter, the channel of the kernel is 1, and all channels of the source matrix use the same kernel information to perform the convolution. In mathematic form, given the source N by M matrix **A**, with K channels, the process of 2D filter on multiple channels using a single channel kernel B is given by:

$$\boldsymbol{C}(i,j,k)=\sum_{n=1}^{H_K}\sum_{m=1}^{W_K}A\left(i-\frac{H_k}{2}+n,j-\frac{W_K}{2}+m,k\right)\boldsymbol{B}(n,m) \tag{6.2.3}$$

Where **C** is the destinated matrix with the same channel number as **A**.

For the case when the kernel **B** also has multiple channels:

$$\boldsymbol{C}(i,j,k)=\sum_{n=1}^{H_K}\sum_{m=1}^{W_K}A\left(i-\frac{H_k}{2}+n,j-\frac{W_K}{2}+m,k\right)\boldsymbol{B}(n,m,k) \tag{6.2.5}$$

Please notice that kernel **B** must have the same number of channels as **A** and **C**. The advantage of using multi-channel 2D filter over calling2D filter multiple times is that DECX will overlap kernel execution and data transfer to save time. For the APIs running on CPUs, the threads will not synchronize until all they go through the channels. However, a thread synchronization will take place every time after a 2D filter is called. Moreover, there is only one initialization and configuration for all the convolutions on channels, which prevents configuring for the same convolution processes to save time.

2. Supported Types

The supported types of 2D filters is shown in Figure 6.2.1.

TYPES	FP32		FP16		UINT8		FP64		INT32		CPf	
	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA
Matrix	Y	Y	N	Y	Y	Y	Y	N	N	N	N	N
GPU_Matrix	Y	Y	N	Y	Y	Y	Y	N	N	N	N	N

Figure 6.2.1 Supported types of filter 2D

3. Warnings

The input dimension of kernel should not exceed 33x33 when passing a _FP32_ matrix to CUDA 2D filter APIs. The reason is, when executing the kernels, DECX loads all the necessary values to shared memory [], and the maximum size supported by shared memory has been reached when the filter size is 33x33. Considering that larger filter sizes are not commonly used practically, DECX does not provide extra kernels for those cases.

4. APIs

CUDA part:

```
_DECX_API_ de::DH de::cuda::Filter2D(de::Matrix& src, de::Matrix& kernel, de::Matrix& dst,  const int conv_flag, const int accu_flag, const de::_DATA_TYPES_FLAGS_ output_type);

_DECX_API_ de::DH de::cuda::Filter2D(de::GPU_Matrix& src, de::GPU_Matrix& kernel, de::GPU_Matrix& dst, const int conv_flag, const int accu_flag, const de::_DATA_TYPES_FLAGS_ output_type);

_DECX_API_ void Filter2D_Async(de::Matrix& src, de::Matrix& kernel, de::Matrix& dst, const int conv_flag, const int accu_flag, const de::_DATA_TYPES_FLAGS_ output_type, de::DecxStream& S);

_DECX_API_ void Filter2D_Async(de::GPU_Matrix& src, de::GPU_Matrix& kernel, de::GPU_Matrix& dst, const int conv_flag, const int accu_flag, const de::_DATA_TYPES_FLAGS_ output_type, de::DecxStream& S);
```

CPU part:

```
_DECX_API_ de::DH Filter2D(de::Matrix& src, de::Matrix& kernel, de::Matrix& dst, const int flag,
    const de::_DATA_TYPES_FLAGS_ _output_type);

_DECX_API_ void Filter2D_Async(de::Matrix& src, de::Matrix& kernel, de::Matrix& dst, const int con
    v_flag, const de::_DATA_TYPES_FLAGS_ output_type, de::DecxStream& S);
```

Parameters:

src: The input matrix. If the input object is not initialized, process returns handle with error of `DECX_FAIL_CLASS_NOT_INIT`. If the type of the source matrix is not supported (See chapter *Supported Type*) for more information.

kernel: The filter kernel. If the input object is not initialized, process returns handle with error of `DECX_FAIL_CLASS_NOT_INIT`. If the type of the source matrix is not supported (See chapter *Supported Type*) for more information.

dst: The result matrix of the 2D filter. No initialization is required, the properties of output object will be automatically set.

Flag: Only used when the input matrix A and B are both carrying fp16 data, indicating the fp16 calculation behaviors. Users can set the value of this flag to trade-off between throughputs and precision.

Output_type: Indicates the output data type. Only works when the type of input matrix is `de::_DATA_TYPE_FLAGS_::_UINT8_`. The supported output types are `de::_DATA_TYPE_FLAGS_::_UINT8_` and `de::_DATA_TYPE_FLAGS_::_FP32_`. If the other type flag is indicated, process returns handle containing the error message of `DECX_FAIL_ErrorFlag`.

S: The asynchronous stream that the process binds to. Only used in asynchronous callings.

Return:

Process returns the handle carrying the runtime status, such as errors or warnings.

7.2.2. NN-type convolutions

1. Introduction

2D convolution on multiple channel matrix or tensor is common on neural network, such as CNN (Convolutional Neural Network) []. The difference from multi-channel 2D filter is that, the source matrix is in de::Tensor form rather de::Matrix. In this form, the third dimension, where the channel represents, is denoted by index k (Please see chapter 4.11 *Tensor* and 4.12 *GPU_Tensor* for more information). The kernel **B** should have 4D data structure, which is de::TensorArray. Each tensor in **B** should have the same channel number as **A**, which means that their third dimension is the same numerically. The destinated tensor **B** should be a tensor, with the same channel as the number of tensors in **B**.

In mathematical representation, assuming that **A** denotes the source tensor, with dimensions$[M, N, K]$, **B** denotes the kernel, with dimensions $[M_B, N_B, K_B, H_B]$, **C** denotes the destinated tensor with dimensions $[M_C, N_C, K_C]$.

$$\boldsymbol{C}(i,j,k) = \sum_{x=1}^{M_B}\sum_{y=1}^{N_B}\sum_{l=1}^{K_B} \boldsymbol{A}\left(i - \frac{H_k}{2} + n, j - \frac{W_K}{2} + m, l\right) \boldsymbol{B}(x,y,l) \tag{6.2.6}$$

Where, $K = K_B$, $K_C = H_B$ are required. The borders are handling in the same way as that in 2D filter.

2. Supported types

The supported types of 2D filters is shown in Figure 6.2.2.

TYPES	FP32		FP16		UINT8		FP64		INT32	
	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA
Matrix	Y	Y	N	Y	N	N	N	N	N	N
GPU_Matrix	Y	Y	N	Y	N	N	N	N	N	N

Figure 6.2.2 Supported types of NN-type convolution

3. APIs

CPU part

```
_DECX_API_ de::DH de::cpu::Conv2D(de::Tensor& src, de::TensorArray& kernel, de::Tensor& dst, const de::Point2D strideXY, const int conv_flag);

_DECX_API_ de::DH de::cpu::Conv2D_Async(de::Tensor& src, de::TensorArray& kernel, de::Tensor& dst, const de::Point2D strideXY, const int conv_flag, de::DecxStream& S);
```

CUDA part:

```
_DECX_API_ de::DH de::cuda::Conv2D(de::GPU_Tensor& src, de::GPU_TensorArray& kernel,  de::GPU_Tensor& dst,  const de::Point2D strideXY, const int flag, const int accu_flag);

_DECX_API_ de::DH de::cuda::Conv2D_Async(de::GPU_Tensor& src, de::GPU_TensorArray& kernel,  de::GPU_Tensor& dst, const de::Point2D strideXY, const int flag, const int accu_flag, de::DecxStream& S);
```

Parameters:

src: The input tensor. If the input object is not initialized, process returns handle with error of **`DECX_FAIL_CLASS_NOT_INIT`**. If the type of the source matrix is not supported (See chapter *Supported Type*) for more information.

Kernel: The convolution kernel, which is in type of tensor array. If the input object is not initialized, process returns handle with error of `DECX_FAIL_CLASS_NOT_INIT`. If the type of the source matrix is not supported (See chapter *Supported Type*) for more information.

dst: The result tensor of the NN-type convolution. No initialization is required, the properties of output object will be automatically set.

Flag: Only used when the input matrix A and B are both carrying fp16 data, indicating the fp16 calculation behaviors. Users can set the value of this flag to trade-off between throughputs and precision.

Output_type: Indicates the output data type. Only works when the type of input matrix is `de::_DATA_TYPE_FLAGS_::_UINT8_`. The supported output types are `de::_DATA_TYPE_FLAGS_::_UINT8_` and `de::_DATA_TYPE_FLAGS_::_FP32_`. If the other type flag is indicated, process returns handle containing the error message of `DECX_FAIL_ErrorFlag`.

S: The asynchronous stream that the process binds to. Only used in asynchronous callings.

Return:

Process returns the handle carrying the runtime status, such as errors or warnings.

7.3. Dot product

7.4. Add

7.5. Subtract

7.6. Multiply

7.7. Divide

7.8. Mathematical functions

7.9. Operators of complex number

8. Digital Signal Processing

8.1. FFT and IFFT

1. Introduction

Fast Fourier transform is one of the most important algorithms in digital signal processing, which transforms the signals from time domain to frequency domain. Given a discrete signal $x(n)$ with length N, assuming that the transformed frequency domain signal is $X(k)$. Discrete 1D Fourier transform is given by.

$$X(k)=\sum_{n=0}^{N-1}x(n)e^{-\frac{j2\pi nk}{N}} \tag{8.1.1}$$

For a 2D time domain (or special domain) signal $x(i,j)$, similar to the case in 1D, the transformed frequency domain signal is given by:

$$X(u,v)=\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}x(i,j)e^{-j2\pi\left(\frac{iu}{M}+\frac{jv}{N}\right)} \tag{8.1.2}$$

Where, j is imaginary unit.

2. APIs

8.2. Frequency domain filter

9. Computer Vision

9.1. Image processors

9.1.1. Image loading

1. Introduction

DECX provides functions for loading images from disks. The decoder is based on SDL library []. Common formats of images, such as jpg, png, bmp, etc.

2. API

```
_DECX_API_ de::DH de::vis::ReadImage(const char* img_path, de::Matrix& src);
```

Parameters:

img_path: The absolute path of the image. If the path is invalid or file cannot be found, process returns a handle contains error information of **de:: DECX_FAIL_IMAGE_LOAD_FAILED**.

src: The matrix used for containing the pixels data. No initialization is needed. The data type of returned matrix will be set to **de::_UCHAR3_** if the image is detected to be in RGB format. If alpha channel is detected, the returned image will be set to **de::_UCHAR4_**.

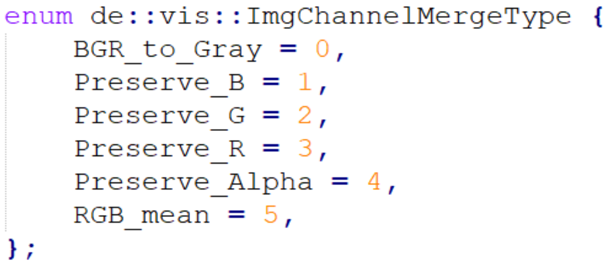
Return:

Process returns the handle carrying the runtime status, such as errors or warnings.

9.1.2. Channel Merging

1. Introduction

DECX provides function for users to manipulate image channels. An enumerator contains different channel operations is also provided.



```
enum de::vis::ImgChannelMergeType {
    BGR_to_Gray = 0,
    Preserve_B = 1,
    Preserve_G = 2,
    Preserve_R = 3,
    Preserve_Alpha = 4,
    RGB_mean = 5,
};
```

Figure 9.1.1 Channel operations

9.2. Filters

9.2.1. Gaussian Filter

1. Introduction

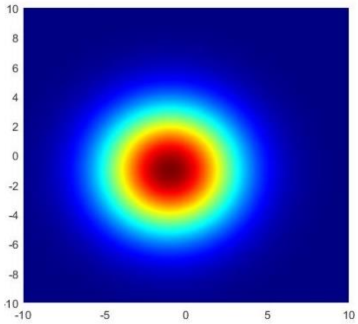
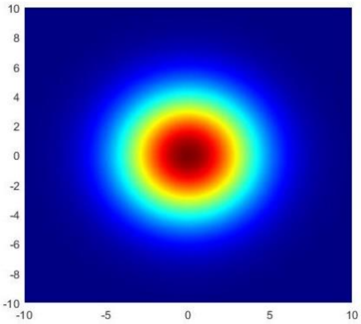
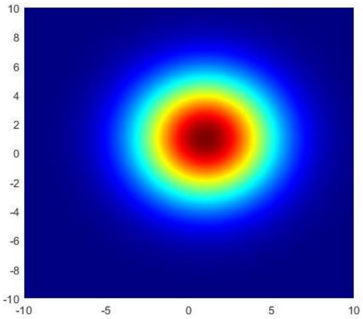
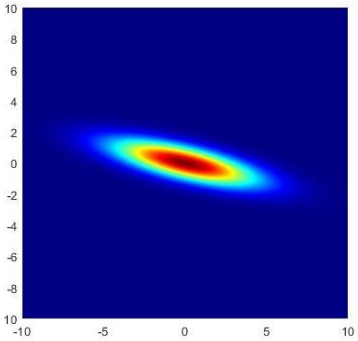
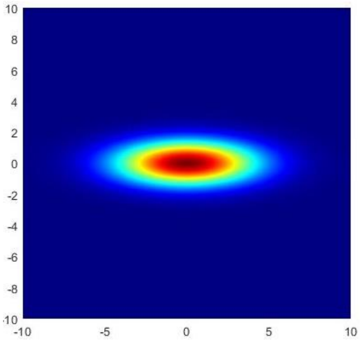
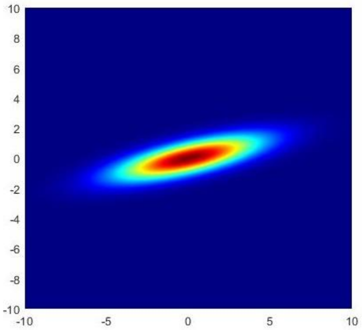
Gaussian filter on image is a 2D convolution process, please refer to chapter7.2.1 *Filters* for more information. However, the filter kernel is a normalized 2D Gaussian function, given by:

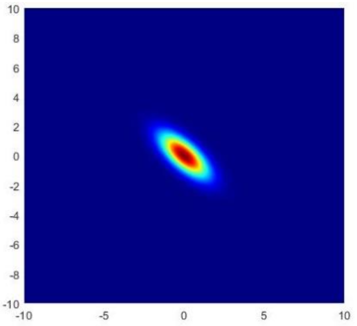
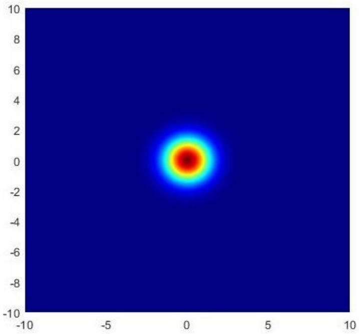
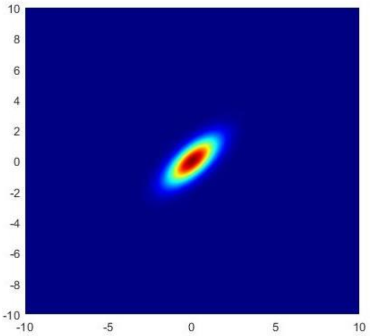
$$f(x,y) = (2\pi\sigma_1\sigma_2)^{-1}\exp\left[-0.5\left(\frac{(x-\mu_1)^2}{\sigma_1^2}+\frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right] \tag{9.2.1}$$

Which can be separated as product of two 1D Gaussian distributions with independent variables x and y:

$$\begin{aligned} f(x,y) &= \left(\sqrt{2\pi}\sigma_1\right)^{-1}\exp\left(-\frac{1}{\sqrt{2}}\left(\frac{(x-\mu_1)^2}{\sigma_1^2}\right)\right)\times\left(\sqrt{2\pi}\sigma_2\right)^{-1}\exp\left(-\frac{1}{\sqrt{2}}\left(\frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right) \\ &= Gauss1D(x)Gauss1D(y) \end{aligned} \tag{9.2.2}$$

Different 2D Gaussian kernels generated by different parameters are shown below.

<table>
<tr><th></th><th>$\mu_1 = -1\ \mu_2 = -1$</th><th>$\mu_1 = 0\ \mu_2 = 0$</th><th>$\mu_1 = 1\ \mu_2 = 1$</th></tr>
<tr><td>$\sigma_1 = 3$
$\sigma_2 = 3$</td>
<td>
</td>
<td>
</td>
<td>
</td></tr>
<tr><td>$\sigma_1 = 3$
$\sigma_2 = 1$</td>
<td>
</td>
<td>
</td>
<td>
</td></tr>
</table>

$\sigma_1 = 1$ $\sigma_2 = 1$			

9.2.2. Bilateral filter

9.3. Canny Edge detectors

1. Introduction

1. Sobel Operator

Sobel operator is given by:

$$\text{Sobel X: } \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \text{ Sobel Y: } \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

2. Scharr Operator

Scharr operator is given by:

$$\text{Scharr X: } \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} \text{ Scharr Y: } \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

Canny edge detector [] first convolutes the image using Sobel or Scharr operator and obtains the gradient angle, gradients amplitudes of x and y directions. The gradients are compared with two thresholds that one is smaller, another one is larger, called lower threshold and higher threshold, respectively. The pixels whose gradients are larger than the higher threshold will be preserved (set to 255, pure white) in the output image; for those whose gradients are smaller than the lower threshold will be abandoned (set to 0, pure black); for those whose gradients are between the two thresholds, they are compared further in another complicate algorithm, which is not introduced in this article.

Canny edge detector is widely used in edge detection of images. Since it is fast and can be done in real-time with the help of modern CPU. Since the calculation process is much faster compared to the data transfer, no CUDA version of API of Canny edge detector is provided in DECX.

Canny edge detector in DECX ignores the border of the image, with bandwidth of 1 pixel. This simplification is acceptable since the border of 1 pixel width does not contain important information.

2. APIs

```
_DECX_API_ de::DH de::vis::cpu::Find_Edge(de::Matrix& src, de::Matrix& dst, const float _L_threshold, const float _H_threshold, const int method);
```

Parameters:

src: The input image (matrix with data type of de::_UINT8_). If the input object is not initialized, process returns handle with error of **`DECX_FAIL_CLASS_NOT_INIT`**. The input image must be grayscale, colorful images (de::_UCHAR3_ or de::_UCHAR4_) is not supported.

_L_threshold: The lower threshold for Canny edge detector.

_H_threshold: The higher threshold for Canny edge detector.

Method: The operator of Canny edge detector, either **`de::DE_SOBEL`** (Sobel) or **`de::DE_SCHARR`** (Scharr).

Return:

Process returns the handle carrying the runtime status, such as errors or warnings.

10. 2D and 3D fields

11. References