Cambricon
寒 武 纪

BANG C Language Developer Guide

Release 2.4.1

Cambricon

Feb 24, 2020

Table of Contents

List of Figures

List of Tables

1 Copyright

2 Preface

2.1 Version

This BANG C -v2.X is compatible with BANG C -v1.7, and the target platform is MLU100, MLU270, 1H8 and 1H8 mini.

Table 2.1: BANG C Doc Version

Document Name	Cambricon BANG C Language Developer Guide
Version	V2.4.1
Author	Cambricon
Date	2020.02.21

2.2 Update History

The comparison between BANG C language version and compiler, MLISA language and assembler version is as follows:

Table 2.2: Versions

BANG C Version	CNCC Version	MLISA Version	CNAS Version	Supported Arch
BANG C-v2.0.x	CNCC-v2.0.x	MLISA-v2.0.x	CNAS-v2.0.x	MLU100, 1H8, MLU270, MLU220, 1H8 mini
BANG C-v1.7.x	CNCC-v1.7.x	MLISA-v1.8.x	CNAS-v1.8.x	MLU100, 1H8, 1H8 mini
BANG C-v1.6.x	CNCC-v1.6.x	MLISA-v1.7.x	CNAS-v1.7.x	MLU100, 1H8
BANG C-v1.5.x	CNCC-v1.5.x	MLISA-v1.6.x	CNAS-v1.6.x	MLU100, 1H8
BANG C-v1.4.x	CNCC-v1.4.x	MLISA-v1.5.x	CNAS-v1.5.x	MLU100, 1H8
BANG C-v1.3.x	CNCC-v1.3.x	MLISA-v1.4.x	CNAS-v1.4.x	MLU100, 1H8
BANG C-v1.2.x	CNCC-v1.2.x	MLISA-v1.3.x	CNAS-v1.3.x	MLU100
BANG C-v1.1.x	CNCC-v1.1.x	MLISA-v1.2.x	CNAS-v1.2.x	MLU100
BANG C-v1.0.x	CNCC-v1.0.x	MLISA-v1.1.x	CNAS-v1.1.x	MLU100

2.3 About the Document

This document “Cambricon BANG C Language Developer Guide-CN-v2.x.x” provides a reference manual for Cambricon MLU (Machine Learning Unit) programming model and programming language of BANG C. The software implementation for this manual is CNCC (Cambricon Compiler Collection). For the correspondence between the version of BANG C language and the version of CNCC, please refer to the document release notes. CNCC, CNDRV, and CNML, etc. are part of CNSDK.

3 Overview

3.1 Introduction to BANG C Language

Machine learning tasks are becoming pervasive in a broad range of domains(from embedded systems to data centers). As architectures evolve toward heterogeneous multi-cores composed of a mix of cores and processors, Cambricon proposed a machine learning processors called MLU(Machine Learning Unit) realizing the best tradeoff between flexibility and efficiency.

BANG C language is a programming language proposed by the Cambricon for MLU (Machine Learning Unit) hardware. It has both terminal and cloud target platforms. It has the characteristics of general-purpose computation and machine learning high-performance computation. Its main functions include:

- Provide efficient programming interface to fully utilize Cambricon hardware features;
- Provide a common heterogeneous programming approach that allows users to extend their own applications;
- Provide users with a comprehensive and sustainably compatible high-level development language for performance optimization;
- Support for parallel programming models.

3.2 Introduction to CNCC

The structure of the CNCC (Cambricon Neuware Compiler Collection) is shown in Figure CNCC Compiler Architecture. The developer uses the BANG C language to develop a source code of the MLU side: compiles firstly the source code to MLISA assembly through the front-end CNCC, and then assembles the MLISA by the CNAS (Cambricon Neuware Assembler) to generate a binary machine code including dynamic library, static library and executable file.

Attention:

When compiling the program, the default compiler optimization level option is O0.

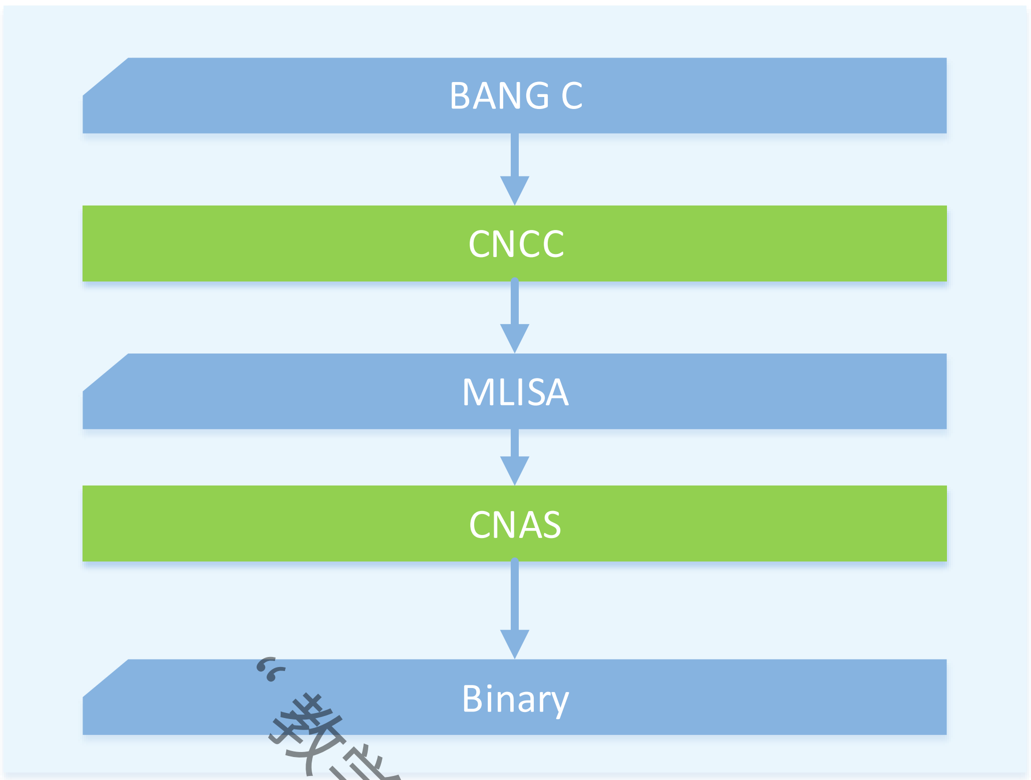

Fig. 3.1: CNCC Compiler Architecture

4 Hardware Architecture

4.1 Hardware Concepts

Table 4.1: Interpretation of MLU Hardware Concepts

Concept	Interpretation
Core	MLU270 has 16 physical Cores, MLU100 has 32 physical Cores, but 1H8 has only one physical Core.
Cluster	In MLU270 and MLU100, every 4 physical Cores form 1 Cluster, but 1H16 and 1H8 has no Cluster.
Channel	MLU270 and MLU100 has 4 DDR Channels (For single-core IP Core product forms such as 1H8 and 1H16 , each IPU Core is directly connected with memory ,unlike MLU100 and MLU270 which have a memory controller connected with multiple IPU Cores through NOC ,so channel means direct access path between IPU and memory .
GPR	Abbreviation for General Purpose Register. GPR set is a private general purpose register for each Core. GPR has no fixed data type, but it has a width of 48 bits and can store any data within 48 bits.
NRAM	Abbreviation for Neural-RAM. The on-chip RAM in each Core is used to store scalar or stream data. MLU270 has a size of 512KB and 1H8 has a size of 256KB, and part of them is CNAS reserved space. When the total size of the variable defined by the user on the NRAM is over the total platform size, it will report an error in the CNAS.
WRAM	Abbreviation for Weight-RAM. The on-chip RAM in each Core is used to store special stream data such as weights. MLU270 has a size of 1MB and 1H8 has a size of 512KB. When the total size of the variable defined by the user on the WRAM is over the total platform size, it will report an error in the CNAS.
SRAM	Abbreviation for Shared-RAM. It refers to the on-chip RAM shared by 4 Cores in one cluster of MLU270. The size is 2MB. Errors will be reported during compilation if the total size of variables on SRAM defined by users is bigger than 2MB.

GDRAMs	Abbreviation for Global-DRAMs. DDR storage with 16 Cores of MLU270 sharing the access DDR memory of the board is divided into GDRAM and LDRAM. Please refer to CNDRV manual for configuration of MLU270 GDRAM and LDRAM.[1] (Note: The data length must be an integer multiple of 32 Bytes. The CNCC will check the immediate operand in compiling.)
LDRAMs	Abbreviation for Local-DRAMs. DDR storage with each Core accessing privately DDR memory of the board is divided into GDRAM and LDRAM. Please refer to CNDRV manual for configuration of MLU270 GDRAM and LDRAM.[1] (Note: When memcpy is used to access LDRAM, the data length must be an integer multiple of 32 bytes. The CNCC will check the immediate operand in compiling.)
LLC	Abbreviation for Last Level Cache. Cache area between GDRAMs and on-chip memory, used to accelerate the reading of the same data in multi-core parallel processing. The maximum size of LLC cacheable data is 2MB. When BANG C program uses the constant pointer pointing to the GDRAM space to perform memory access, the CNCC will optimize the memory access to enable LLC. 1H8 architechture has no LLC.

4.2 MLU Server Hierarchy

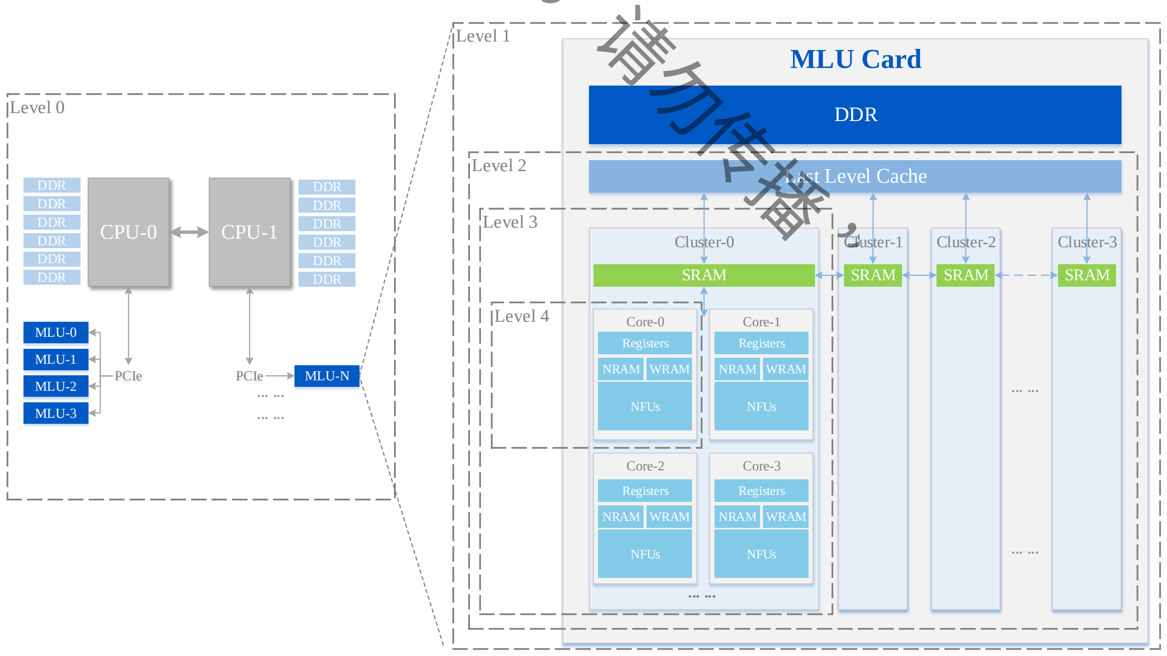

Fig. 4.1: MLU Server Hierarchy

MLU server exchange data with the Host through PCIe. The MLU server hierarchy system, with multiple cards, includes five levels: server level, card level, chip level cluster level, and core level, as

[1] DDR4 memory of MLU100 hardware can be configured with LDRAM and GDRAM size. For details, please refer to the CNDRV Cambricon MLU100 Driver Developer’ s Guide-CN-v1.x.

shown in Figure MLU Server Hierarchy .

- The Level 0 is the server level, consisting of several CPU control units, local DDR storage units, and several MLU cards interconnected by PCIe bus as the computing units of the server level.
- The Level 1 is card level. Each MLU card consists of local DDR storage units and chips as the computing units.
- The Level 2 is chip level. Each chip consists of several cluster as the computing units.
- The Level 3 is cluster level. Each cluster consists of sveral accelerating cores as control and computing units, and shared memory as storage unit.
- The Level 4 is core level. Each accelerating core consists of local storage and local processing unit array.

The MLU server hierarchy is capable of conveniently improving the computing power of the whole system by adding the number of cards, chips, clusters or cores.

4.3 MLU Chip Series Overview

Cambricon MLU200 product series include MLU220 and MLU270. Each cluster contains 4 Cores. MLU270 architechture chip has 4 Clusters and 16 IPU Cores; MLU220 has 1 Clusters and 4 IPU Cores. The other difference between the architechtures is shown on the table.

Table 4.3: MLU2XX Series Comparison

MLU	220	270
Computing Core	Cambricon IPU 1M	Cambricon IPU 1M
IPU Cores	4	16
Memory	2/4/8 GB LPDDR4x 29GB/s	16/32 GB DDR4 (ECC) 102.4 GB/s
Multimedia	16 streams video decoder (1080p30) 8 streams video encoder	48 streams video decoder (1080p30) 16 streams video encoder
Data Types	FP32, FP16, INT16, INT8, INT4	FP32, FP16, INT16, INT8, INT4
INT8	16 TOPS	128 TOPS
System Interface	1X4/2X2 PCIe Gen3	X16 PCIe Gen3
Form Factor	Multi (Module, M.2)	Low-Profile PCIe、FHHL PCIe、FHFL PCIe with fan
Process Node	TSMC 16nm	TSMC 16nm
TDP	10W/6W	70W/150W

Moreover, MLU100 architechture chip has 8 Clusters and 32 IPU Cores; 1H serials contains only one Core without cluster. On-chip NRAM size of MLU100 is 512KB, on-chip WRAM size of MLU100 is 1MB. On-chip NRAM and WRAM of 1H8 are half the size of 1H16. Nor does it have LLC storage. 1H8 CONV and MLP computation only supports the INT8 type. The user can compile the code of the platform using the compile option –bang-mlu-arch=1H16 for 1H16 and –bang-mlu-arch=1H8 for 1H8.

4.4 MLU270 Architecture

MLU270 is mainly used in the cloud, and Figure MLU270 Multi-core Processor Overall Architecture shows the architecture of MLU270. It is a multi-core processor architecture consisting of 16 Cores (C0 ~ C15), Network on Chip (NOC), and 4 DDR controllers (DDR0 ~ DDR3) in physical implementation. Each four consecutive Cores constitute a Cluster. In addition, 16 physical Cores are connected to the NOC. The NOC is connected to four DDR controllers, each containing one Channel.

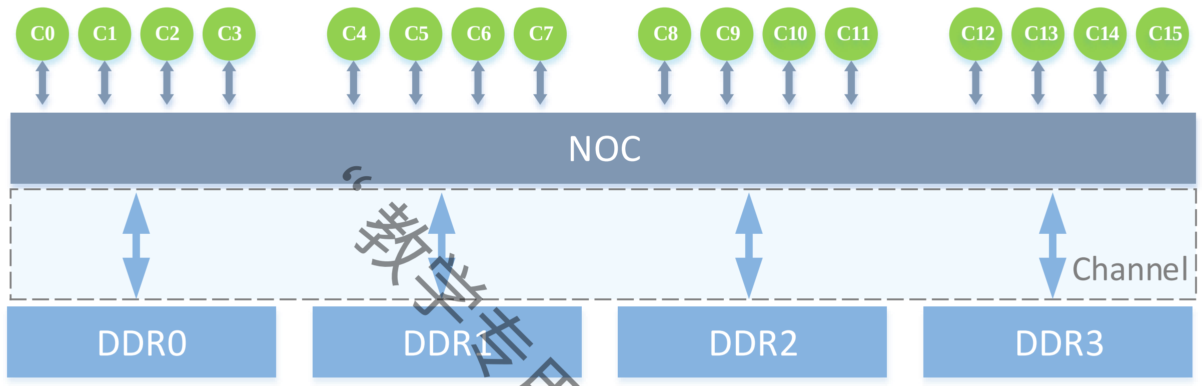

Fig. 4.2: MLU270 Multi-core Processor Overall Architecture

4.5 MLU 270 Memory Hierarchy

The memory hierarchy of the MLU270 single Channel is as following figure. Each Core is mainly composed of a functional unit (FU), a general register group (GPR), a neuron storage unit (Neural-RAM, NRAM), and a weight storage unit (Weight-RAM, WRAM). In addition, 4 physical Cores form a Cluster. And, each MLU270 has 4 Clusters. Each Core has a separate piece of memory called Local-DRAMs(LDRAMs). As shown in Fig. 4.3, four Cores in one Cluster shares Shared-RAM(SRAM) on chip. In addition, all Cores also have access to global shared memory called Global-DRAMs(GDRAMs) on DDR. LDRAMs, GDRAMs and on-chip storage channels also have a Level 1 cache (Last Level Cache, LLC), which is mainly used to buffer shared read-only data between multiple Cores, which accelerates the memory access. When the BANG C program uses the constant pointer pointing to the GDRAM space to perform memory access, the CNCC will optimize the memory access to enable LLC.

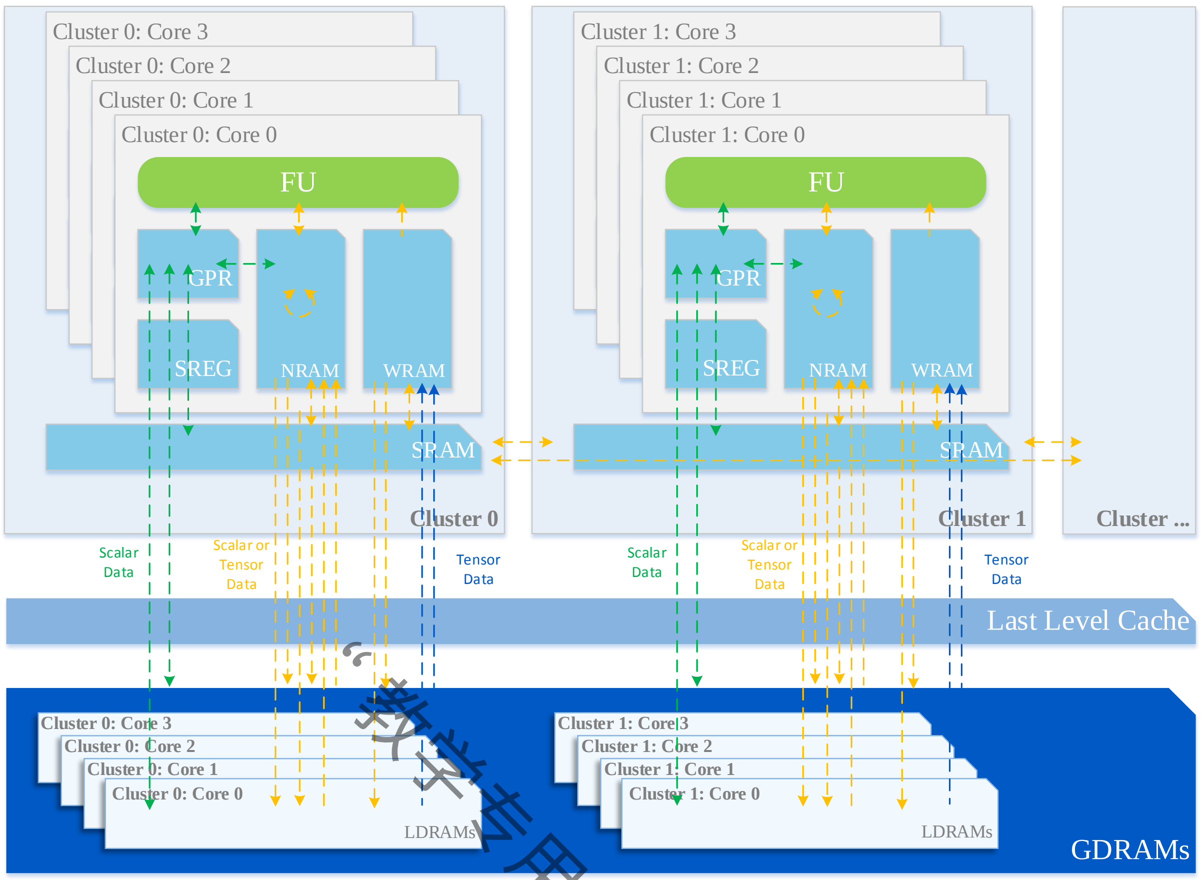

Fig. 4.3: MLU270 Memory Hierarchy

Hint:

The user can compile the code of the MLU270 platform using the compile option –bang-mlu-arch=MLU270.

5 Programming Model

Heterogeneous programming realize the use of computing units with different types of instruction set and architectures. BANG C heterogeneous programming model is based on collaborative computing between CPU and MLU, so as to break through the bottleneck of CPU development and utilize the machine learning ability of MLU, effectively solve the problems of energy consumption and scalabitlity.

Cambricon BANG C language gives consideration to both terminal and cloud target platform. This chapter introduces the parallel model and storage model of BANG C language and related concepts in combination with the concept of MLU hardware architecture.

5.1 Heterogeneous Programming

Heterogeneous computing systems are usually composed of general processors and many domain-specific processors: General processors as control devices (referred to as Host) for complex control and scheduling; domain-specific processors as sub-devices (referred to as MLU) for large-scale parallel computing and domain-specific computing tasks. Host and MLU cooperate to complete computing tasks. For a heterogeneous computing system, the original isomorphic parallel programming model is no longer applicable. Therefore, the heterogeneous parallel programming model gets increasing attention in academic and industrial circles. This chapter gives a brief introduction of Cambricon MLU heterogeneous programming.

5.1.1 Process of Compiling and Linking

Heterogeneous programming includes Host and MLU. For the Host, it mainly includes device acquisition, data/parameter preparation, execution flow creation, task description, Kernel startup, output acquisition, etc. The Entry function is the program entry on the MLU and can call the MLU functions. The MLU program uses the C/C++ language extension of the heterogeneous programming model. Binary files are compiled by the specified compiler of the MLU.

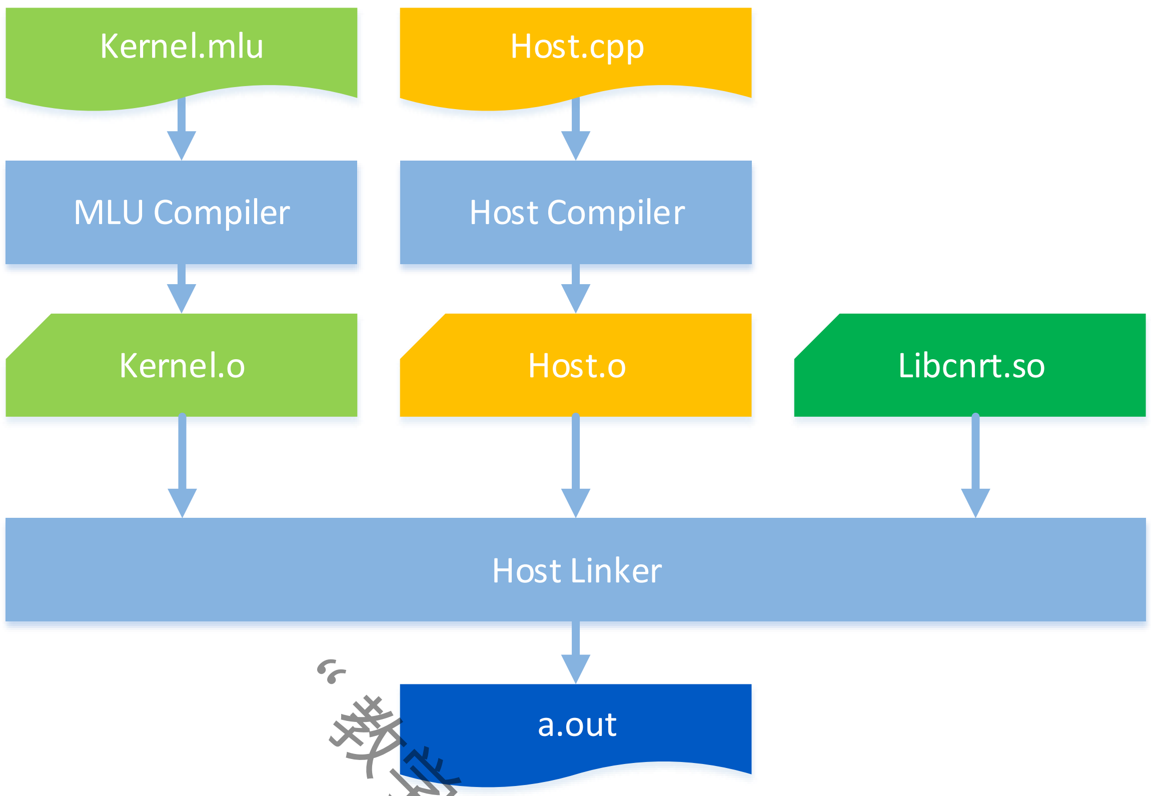

Fig. 5.1: The Compilation and Linking Process of MLU Heterogeneous Program

The compilation and linking process of MLU heterogeneous program are shown in Figure The Compilation and Linking Process of MLU Heterogeneous Program . Separate programming method is adopted, that is, the Host program and the MLU program locate in different files (i.e. the Host and Kernel files). The heterogeneous parallel program of the Host program and the MLU program need to be compiled by its own compilers.

To be more specific, the Host program is a common C/C++ program, and users can use any C/C++ compiler such as GCC, CLANG, etc. The MLU program is an extension based on C/C++ language and can be compiled by the specified compiler CNCC provided by Cambricon. The Host Linker forms an executable program by linking the two target files from Host and MLU, runtime library and other files.

The following example compiles the three files l2loss.mlu, l2loss_main.cpp, l2loss_ops.cpp into an executable program l2loss.

```
cncc -c l2loss.mlu -o l2loss.o
g++ -c l2loss_main.cpp
g++ -c l2loss_ops.cpp  -I/usr/local/cambricon/include
g++ l2loss.o l2loss_main.o l2loss_ops.o -o l2loss -L/usr/local/cambricon/lib -lcnrt
```

5.1.2 HOST Program

The HOST program is a common C/C++ program that initializes the device by calling CNRT-API, manages device memory, prepares Kernel parameters, starts Kernel, and releases resources. The main process of calling the Kernel program by the HOST program will be described below.

5.1.2.1 Header File

The HOST program needs to include the runtime header file cnrt.h, which provides a declaration of the runtime interface required for heterogeneous programming and the definition of the relevant data types used by the HOST program. For details, please refer to the CNRT related documentation[3]
.

5.1.2.2 Initialize Device

Before starting the Kernel, users need to call the CNRT interface to initialize the device, as shown in the following example:

```
cnrtInit(0);
```

5.1.2.3 Get Device

After initializing the device, users may get the device by the following method:

```
cnrtDev_t dev;
cnrtGetDeviceHandle(&dev, 0);
cnrtSetCurrentDevice(dev);
```

5.1.2.4 Prepare Input Data of MLU

The HOST side needs to prepare the input data of the MLU program and copy the input data to the specified position of the MLU.Because MLU supports some special data types, for example, half type (half-precision floating point, that is, floating point with two bytes) is not supported by C/C++ language currently, the users need to convert input data from MLU on HOST. Performing conversions, such as converting float/double data to half data and storing it in two bytes, for example, the float/double data should be converted into half data and stored in two bytes. At present, run-time provides 2 interface functions, cnrtConvertDoubleToHalf/cnrtConvertFloatToHalf, which is convenient for users to convert double/float data into half data. For details, please refer to the runtime interface[4]. It should be noted that both data conversion functions store the half data in the uint16_t data type. For details of usage method, please refer to the following example:

```
typedef uint16_t half;
half* input_half = (half*)(malloc(dims_a * sizeof(half)));
for (int i = 0; i< len; i++) {
  cnrtConvertFloatToHalf(input_half+i, input[i]);
}
```

5.1.2.5 Transmit Input Data of MLU

Since the Kernel parameter does not support arrays and cluster types, the input data of the array type needs to be explicitly copied by the user into the MLU space. The users first need to call cnrt-

[3] CNRT Cambricon Neuware Runtime Developer’ s Guide-CN-v2.x
[4] CNRT Cambricon Neuware Runtime Developer’ s Guide-CN-v2.x

Malloc to apply for a DDR space, and then call cnrtMemcpy to copy the data prepared in HOST to DDR. The following is a specific example.

```
half* mlu_input;
cnrtMalloc((void**)(&mlu_input), dims_a * sizeof(half));

cnrtMemcpy(mlu_input, input_half, dims_a * sizeof(half),
  CNRT_MEM_TRANS_DIR_HOST2DEV);
```

5.1.2.6 Prepare Kernel Parameter

The Kernel parameter only supports scalar data. When the Kernel parameter is transmitted, a cnrtKernelParamsBuffer_t should be gotten on the HOST. Then, according to the order defined by the Kernel parameter, the runtime function cnrtKernelParamsBufferAddParam will be called to push the scalar parameters into the buffer in turn. The following is a specific example.

```
cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &mlu_input, sizeof(half*));
```

5.1.2.7 Create Queue

An important concept related to the execution of the BANG C program Kernel is queue. Tasks of the same queue is executed serially, and tasks between different queues are executed in parallel. When we start the Kernel, we need to specify the queue that the Kernel executes. An example of creating a Queue is as follows:

```
cnrtQueue_t pQueue;
cnrtCreateQueue(&pQueue);
```

5.1.2.8 Specify Task Size of Kernel

The data type cnrtDim3_t provided by CNRT is configured to specify the size of the Kernel task. The task size has three dimensions of XYZ. The task size is specified with certain restrictions. Please refer to Section 2.2.2 for details. The following example shows how to specify the execution size of a task (the parallelism of the task is 1).

```
cnrtDim3_t dim;
dim.x = 1;
dim.y = 1;
dim.z = 1;
```

5.1.2.9 Specify Task Type of Kernel

The cnrtFunctionType_t is configured to specify the computational parallelism at Kernel startup, whose value can be BLOCK or UNIONn (n = 1, 2, 4, 8) type. The following is a simple example.

```
cnrtFunctionType_t ft = CNRT_FUNC_TYPE_BLOCK;
```

5.1.2.10 Start Kernel

When the Kernel starts, the function name of Kernel, the size of task, the Kernel parameter, the task type, and the queue should be transmitted to the cnrt runtime function cnrtInvokeKernel_V2 as parameters. The following is a specific example.

```
ret = cnrtInvokeKernel_V2((void *)(&L2LossKernel), dim,
  params, ft, pQueue);
```

5.1.2.11 Data Transmission from MLU to HOST

The computation result by MLU needs to be explicitly copied to the HOST by users. The users call the runtime function cnrtMemcpy and set the direction parameter to CNRT_MEM_TRANS_DIR_DEV2HOST, which can complete the data transmission from MLU to HOST. The following is an example.

```
cnrtMemcpy(output_half, mlu_output, sizeof(half),
  CNRT_MEM_TRANS_DIR_DEV2HOST);

cnrtConvertHalfToFloat(output, output_half[0]);
```

It should be noted that the half data needs to be converted to float/double data on HOST to be processed. The cnrt runtime provides two functions, cnrtConvertHalfToFloat and cnrtConvertHalfToDouble, to complete the data conversion function.

5.1.2.12 Release Resources

After Kernel is called, the relevant resources should be released. These resources mainly include the Kernel parameter buffer, queue, DDR data on MLU, and the corresponding malloc resources on HOST. Finally, users need to call cnrtDestroy to release the cnrt runtime resources. Please refer to the following example for details:

```
ret = cnrtDestroyKernelParamsBuffer(params);
ret = cnrtDestroyQueue(pQueue);
cnrtFree(mlu_input);
free(output_half);
cnrtDestroy();
```

5.1.3 MLU Program

5.1.3.1 Kernel

The MLU programming model is based on heterogenous programming model. The program that performs tasks on MLU is called Kernel. MLU can execute multiple parallel Kernels at the same time when enough resources are available. Each Kernel has an Entry function and the Entry function is used to call the Device function and Func Function. Device function and Func function can call each other. The statement of Kernel consists of Built-in Function statement and C/C++ language statement.

Entry Function

The Entry function is specified by mlu_entry in BANG C language, as in the following example.

```
__mlu_entry__ void L2LossKernel(half* input, half* output) {
 ......
}
```

Call the cnrtInvokeKernel_V2 function of CNRT-API to start the Kernel. The example code to start the l2lossKernel function on the Host is as follows.

```
ret = cnrtInvokeKernel_V2((void *)(&kernel), dim, params, ft, pQueue);
```

The current version of the BANG C language requires a .mlu file with only one mlu_entry modified function, and all device functions and data that the entry function depends on must be in the same compilation unit.

Device Function

Device function is the function type with the function modifier __mlu_device__. It is used for defining a recursive function or one calling another funtion. It has a certain function call cost (it can be determined by the compilation option whether to use inline optimization). The following is an example of Device function.

```
__mlu_device__ void CreateBox( half* box, half* anchor_,
half* delt_, int A, int W,
int H, half im_w, half im_h)
```

The Entry function can call the Device function, and the statements in the Entry and Device functions respectively use C/C++ language and the BANG C language which is the C/C++ language extension.

Func Function

Func function, with the function modifier __mlu_func__, is the Device function with inline attribute by default. When recursive function is not needed, Func function can be selected to improve performance. The func function must be in the same compilation unit as the entry function that depends on it. The __mlu_func__ modifier can be omitted, and the unmodified function defaults to the func attribute. The following example demonstrates how to start the L2LossKernel function on the Host.

```
__mlu_func__ void CreateBox(half* box, half* anchor_,
half* delt_, int A, int W,
int H, half im_w, half im_h)
```

5.1.3.2 File Name Suffix

The suffix of the program file on MLU is *.mlu, and the name of header file is similar to C language, i.e., *.h.

5.1.3.3 Header File

The MLU program must include the header file mlu.h, which contains definitions of the data types required by MLU programming, as well as declarations for function interfaces.

5.1.3.4 Program Example of Kernel

The following example defines an MLU Kernel function L2LossKernel. Kernel function is very similar to common C language function, and the main difference is that __mlu_entry__ is used to specify this function as the Kernel's entry function, or the Kernel function uses the modifier of the __nram__ address space. Please refer to the following sections for the Kernel programming specifications.

```
#include "mlu.h"
#define ONELINE 64
__mlu_entry__ void L2LossKernel(half* input, half* output, int32_t len) {
  __nram__ int32_t quotient = len / ONELINE;
  __nram__ int32_t rem = len % ONELINE;
  __nram__ half input_nram[ONELINE];
  output[0] = 0;
  for (int32_t i = 0; i < quotient; i++) {
    __memcpy(input_nram, input + i * ONELINE,
      ONELINE * sizeof(half) , GDRAM2NRAM);
    __bang_mul(input_nram, input_nram, input_nram, ONELINE);
    __bang_mul_const(input_nram, input_nram, 0.5, ONELINE);
    for (int32_t j = 0; j < ONELINE; j++) {
      output[0] += input_nram[j];
    }
  }
  if (rem != 0) {
    __memcpy(input_nram, input + quotient * ONELINE,
      ONELINE * sizeof(half), GDRAM2NRAM);
    __bang_mul(input_nram, input_nram, input_nram, ONELINE);
    __bang_mul_const(input_nram, input_nram, 0.5, ONELINE);
    for (int i = 0; i < rem; i++) {
      output[0] += input_nram[i];
    }
  }
}
```

5.2 Parallel Programming

The BANG C language provides a variety of built-in variables for uers to explicitly program in parallel. This chapter introduces the basic concepts of BANG C parallel built-in variables and parallel programming, and gives examples.

5.2.1 Task Division

For cluster and core in a single chip, users can use the built-in variables clusterDim, clusterId, coreDim, coreId to represent the dimensions and IDs of cluster and core respectively. In addition, each hardware computing core performs a task, and the user specifies the size of the Kernel task through the Dim3_t type. Task size generally has three dimensions: x, y, and z. Users can specify tasks size according to scenario requirements. The following example shows how to divide tasks into four parts in x dimension.

```
Dim3_t dim;
dim.x = 4;
dim.y = 1;
dim.z = 1;
```

The built-in variables of task include: taskDim, taskDimX, taskDimY, taskDimZ, taskIdX, taskIdY, taskIdZ, taskId. Each task is mapped to a computing core, that is, each task is executed by a core, which is the basic processing unit of MLU. A Kernel consists of multiple tasks, which are indexed in three dimensions by Dim3_t data structure.

In BANG C language, there is a Block type task for each core. Block is the basic scheduling unit of the programming model layer. It represents that the task in the Kernel will be scheduled to a single core for execution. For each Cluster, there is a Union1 type task. Two clusters form a Union2, and so on. We call this task type. The task type specifies the number of hardware cores required by a Kernel startup, that is, how many physical cores should be occupied all the time during the Kernel execution cycle. Block type task is single-core task and union type task is a multi-core parallel task.

Taking the completion of 16384 long vector addition as an example, assuming that the task type of the task is Union4, you can set dim.x = 16. Each task completes a vector addition of n = 1024, and after 16 tasks in total, you get a vector addition of dim. X * n = 16384 length. The code of vector addition on deep learning processor is shown in following example.

```
#define N 1024
__mlu_entry__add (float* x, float* y, float*z ) {
__nram__ float x_tmp [N ] ;
__nram__ float y_tmp [N ] ;
__memcpy ( x_tmp , x + taskID* N, N * sizeof (float) , GDRAM2NRAM) ;
__memcpy ( y_tmp , y + taskID* N, N * sizeof (float) , GDRAM2NRAM) ;
__bang_add ( x_tmp, x_tmp, y_tmp, N) ;
__memcpy ( z + taskID* N, x_tmp , N * sizeof (float) , NRAM2GDRAM) ;
}
```

Note:

__nram__ represents the neuron ram on the chip and__bang_add is used to complete the vector addition.

5.2.2 Parallel Model

5.2.2.1 Execution Model

MLU270 issues and executes instructions according to taskDimX as parallelism of 1 unit when starting the execution task, that is, the minimum parallel granularity is taskDimX = clusterDim * coreDim. For MLU270, coreDim = 4, users can control the parallel minimum granularity taskDimX by specifying the Union type.This restriction is shown in Table 5.1. When user specify the Union type, if the specified taskDimX is not a positive integer multiple of clusterDim * coreDim, an error will be thrown when CNRT is running .

Table 5.1: Restriction of Software and Hardware Parallel Size of MLU

Hardware Parallel Occupation			Software Parallel Restriction
Union	CLUSTER	CORE	taskDim
1	1	1*4=4	taskDimX = N * clusterDim * coreDim (N is a positive integer)
2	2	2*4=8	
4	4	4*4=16	
8	8	8*4=32	

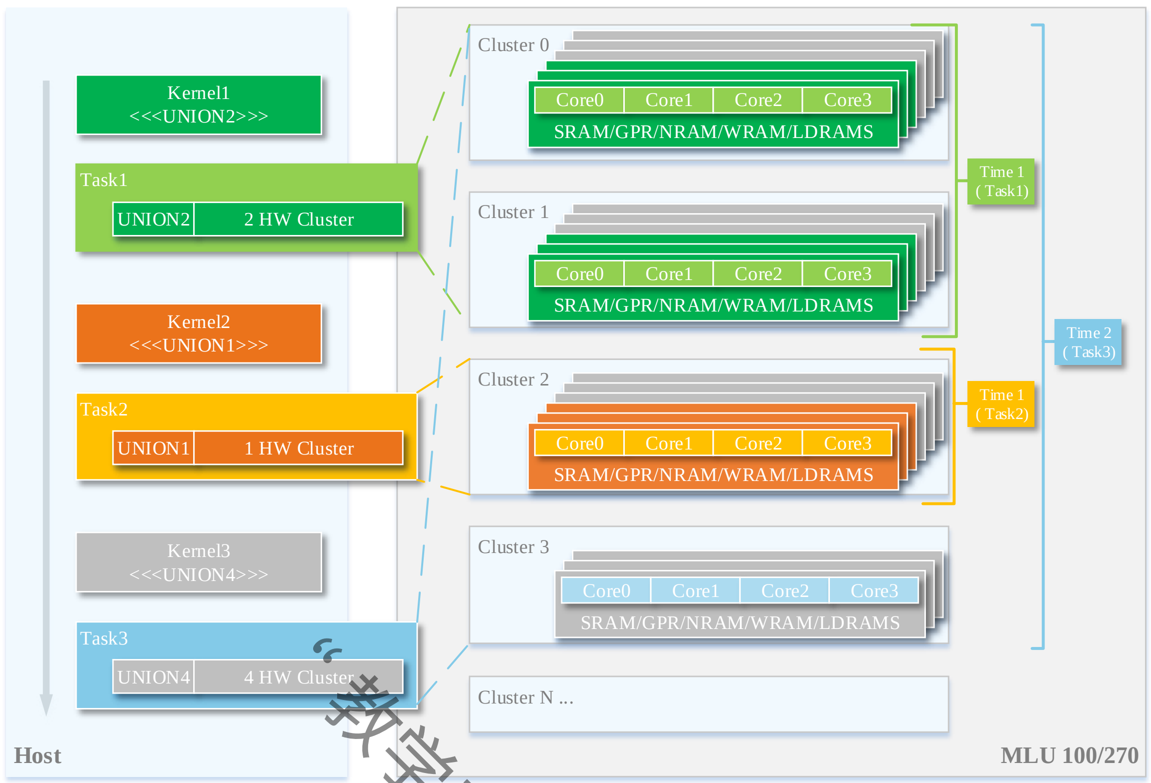

Fig. 5.2: MLU270 Parallel Executed Model

As shown in Figure MLU270 Parallel Executed Model, there are three different Kernel functions on the Host that are transmitted to the MLU270 device for execution. The specific execution process is as follows:

1. For the Kernel1, the Union type and taskDim are defined as follows:

```
cnrtFunctionType_t Union2 = CNRT_FUNC_TYPE_UNION2;
cnrtDim3_t taskDim = {8, 1, 1};  // for example
cnrtInvokeKernel_V2((void *)&kernel1, taskDim, params, Union2, pQueue);
```

The CNRT at HOST transmits Kernel1’s task size taskDim and task type Union2 to the task scheduler. The scheduler will wait until 2 hardware CLUSTERs are idle(Users get clusterId = [0 ~ 1], clusterDim = 2 in Kernel1);

2. When the scheduler finds that 2 CLUSTERs are idle, it starts a total of 8 COREs in the 2 CLUSTERs, so that each CORE starts executing the same Kernel1 code.
3. The total parallel size of Kernel1 software is taskDimX * taskDimY * taskDimZ. According to the restriction taskDimX * taskDimY * taskDimZ = 8 as shown in Table 2, the scheduler will occupy 2 CLUSTERs to start the parallel task with Union2 size at Time1 until the execution of all instructions of 8 COREs on this MLU ends.
4. When Kernel1 is issued, if there is no dependency relationship between Kernel2 and Kernel1(that is, Kernel2 and Kernel1 are not bound to a same Queue, and the two computing tasks can be issued independently without computing the interdependency between input and result),then the Host will continue to issue the Kernel2 task.
5. For the Kernel2, the Union type and taskDim are defined as follows:

```
cnrtFunctionType_t Union1 = CNRT_FUNC_TYPE_UNION1;
cnrtDim3_t taskDim = {4, 1, 1};  // for example
cnrtInvokeKernel_V2((void *)&kernel2, taskDim, params, Union1, pQueue);
```

The scheduler will wait until one hardware CLUSTER is idle, and then the task of Kernel2 will be issued. Kernel2 and Kernel1 are executed in parallel, and all are independent of each other except for GDRAMs.

6. Kernel2 is executed in the same way as Kernel1 on the hardware CLUSTER, and the total parallel size of Kernel2 software is taskDimX * taskDimY * taskDimZ = 4. The scheduler will occupy 1 CLUSTER to start the Union1 parallel task at Time1 until the execution of all instructions of 4 COREs on this MLU ends.
7. When Kernel2 is issued, if there is no dependency relationship between Kernel3 and Kernel1 or Kernel1 (for example, wait synchronization),then the Host will continue to issue the Kernel3 task(In Figure MLU270 Parallel Executed Model , it is assumed that CLUSTER[4-7] is always busy) .
8. For the Kernel3, the Union type and taskDim are defined as follows:

```
cnrtFunctionType_t Union4 = CNRT_FUNC_TYPE_UNION4;
cnrtDim3_t taskDim = {16, 1, 1};  // for example
cnrtInvokeKernel_V2((void *)&kernel3, taskDim, params, Union4, pQueue);
```

The program declares that 4 hardware CLUSTERs are required to execute. At Time2, the scheduler finds that the adjacent CLUSTER[0-3] of the logical ID is idle, and then the Union4 task of Kernel3 is issued at this time.

Note:

The above execution model is based on the assumption that cnrtInvokeKernel_V2 is called asynchronously. Currently, the interface provided by CNRT[5] is still in synchronous execution mode.

5.2.3 Parallel Built-in Variable

5.2.3.1 coreDim

The name of built-in variable of BANG C language retains keywords, and the value is equal to the number of CORE owned by 1 CLUSTER (equal to 4 on MLU270).

5.2.3.2 coreId

The name of built-in variable of BANG C language retains keywords, and the value is equal to the logical ID of each hardware CORE within CLUSTER(the range of value is [0-3] on MLU270).

[5] CNRT Cambricon Neuware Runtime Developer’s Guide-CN-v2.x

5.2.3.3 clusterDim

The name of built-in variable of BANG C language retains keywords, and the value is determined by the Union information of the task(the maximum value is 4 on MLU270 and the maximum value is 8 on MLU100).

5.2.3.4 clusterId

The name of built-in variable of BANG C language retains keywords, and the value is equal tothe logical ID of the CLUSTER where the program is running, and the range of value is [0 - clusterDim-1].

5.2.3.5 taskDimX

The name of built-in variable of BANG C language retains keywords. Before a Kernel is invoked, users need to set the logical size of this task. The logical size has three dimensions: {X, Y, Z}, and the value of taskDimX is equal to the size in the X direction.

5.2.3.6 taskDimY

The name of built-in variable of BANG C language retains keywords.Before a Kernel is invoked, users need to set the logical size of this task. The logical size has three dimensions: {X, Y, Z}, and the value of taskDimY is equal to the size in the Y direction.

5.2.3.7 taskDimZ

The name of built-in variable of BANG C language retains keywords.Before a Kernel is invoked, users need to set the logical size of this task. The logical size has three dimensions: {X, Y, Z}, and the value of taskDimZ is equal to the size in the Z direction.

5.2.3.8 taskDim

The name of built-in variable of BANG C language retains keywords. Before a Kernel is invoked, users need to set the logical size of this task. The logical size has three dimensions: {X, Y, Z}, and the value of taskDim is equal to the value of the logical size of current task after dimension reduction,that is taskDim=taskDimX*taskDimY*taskDimZ.

5.2.3.9 taskIdX

The name of built-in variable of BANG C language retains keywords, and value is equal to the task ID in the X dimension of the logical size assigned when the program is running, the range of value is[0 - taskDimX-1].

5.2.3.10 taskIdY

The name of built-in variable of BANG C language retains keywords,and value is equal to the task ID in the Y dimension of the logical size assigned when the program is running, the range of value is[0 - taskDimY-1].

5.2.3.11 taskIdZ

The name of built-in variable of BANG C language retains keywords,and value is equal to the task ID in the Z dimension of the logical size assigned when the program is running, the range of value is[0 - taskDimZ-1].

5.2.3.12 taskId

The name of built-in variable of BANG C language retains keywords, and the value is equal to the task ID assigned when the program is running, the range of value is [0 - taskDim-1]. The value of taskId is equal to the task ID of logical size after dimension reduction, that is: taskId=taskIdZ*taskDimY *taskDimX + taskIdY *taskDimX+taskIdX

5.2.3.13 Example

This table gives an example to illustrate the value of parallel built-in variables when Kernel is executed.In the example, cnrtFunctionType_t = CNRT_FUNC_TYPE_UNION2, cnrtDim3_t = {x=8, y=2, z=2}.

Table 5.2: The Parallel Variable Value of Union2 Type {X=8, Y=2, Z=2} Task Size on Time- wheel 0-3

ID	taskId				taskIdX				taskIdY				taskIdZ			
CORETime	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	8	16	24	0				0	1	0	1	0		1	
1	1	9	17	25	1											
2	2	10	18	26	2											
3	3	11	19	27	3											
4	4	12	20	28	4											
5	5	13	21	29	5											
6	6	14	22	30	6											
7	7	15	23	31	7											

Table 5.3: The Parallel Variable Value(extend) of Union2 Type {X=8, Y=2, Z=2} Task Size on Time- wheel 0-3

ID	clusterId				coreId			
CORETime	0	1	2	3	0	1	2	3
0	0				0			
1					1			
2					2			
3					3			
4	1				0			
5					1			
6					2			
7					3			

5.2.4 Task Type

When calling the cnrtInvokeKernel_V2 to start the kernel, users need to use cnrtFunctionType_t to specify the computational parallelism when the kernel starts. The value can be BLOCK or UNIONn type, which means the number of computation cores when starting the kernel.

5.2.4.1 UNION

Union is an abstraction of hardware CLUSTER from the perspective of parallel task. UNIONn indicates that n hardware CLUSTERs are occupied simultaneously in Kernel’s execution cycle. MLU270 supports the following Unionn types:

- UNION1 indicates that 1 hardware CLUSTER and 4 COREs are occupied when Kernel is executed.
- UNION2 indicates that 2 hardware CLUSTERs and 8 COREs are occupied when Kernel is executed.
- UNION4 indicates that 4 hardware CLUSTERs and 16 COREs are occupied when Kernel is executed.

5.2.4.2 BLOCK

Block is an abstraction of hardware CORE from the perspective of single-core task. Block indicates that only one hardware CORE is occupied during the execution period of Kernel (MLU270 supports scheduling of 32 different users’ Block-level tasks simultaneously).

5.2.5 QUEUE

In order to meet the requirements of serial and parallel programming design on MLU at the same time, the BANG C language runtime provides a queue mechanism at the software level, and the computation/memory copy task can be bound to a specific Queue to execute. The key idea of queue is as follows:

- The Kernel task is executed asynchronously after being sent to the Queue.
- The tasks in the same Queue are serially executed in the order in which they are sent.
- The tasks are executed in parallel between different queues.

Based on the key idea of Queue, users can put the kernel they want to execute serially into the same Queue, and put the kernel they want to execute in parallel into different Queues.Parallel scheduling of task is implemented by the scheduler. Users do not need to pay attention to the details of hardware scheduling. They only need to master the abstract Queue programming ideas and write BANG C serial/parallel programs. An example of the use of Queue is as follows:

```
cnrtQueue_t pQueue;
cnrtCreateQueue(&pQueue);

ret = cnrtInvokeKernel_V2((void*)&Kernel, dim, params, c, pQueue);
```

5.2.6 NOTIFIER

Notifier can be seen as a special computing task, which can be put into a Queue to be executed like a Kernel task. The internal Queue always follows the FIFO scheduling principle, whether the task is Notifier or Kernel. Compared to Kernel tasks, Notifier task does not need perform actual hardware operations and takes up very little execution time (us level, almost negligible). We can use Notifier to count the hardware time of the computing task. For details, please refer to the CNRT document[6].

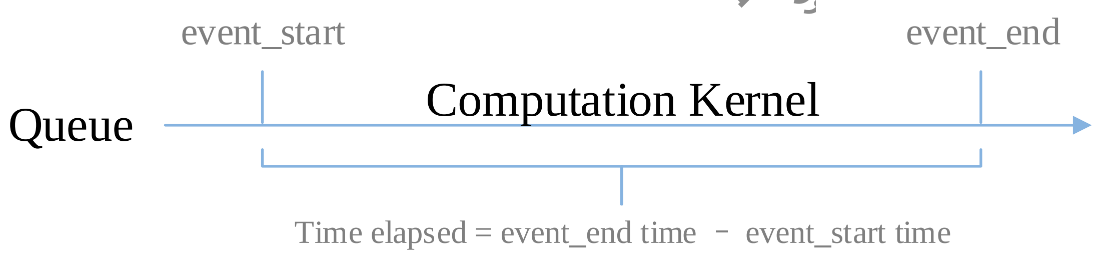

Fig. 5.3: Time Computation of Queue

An example of using the Even mechanism to count kernel execution time is as follows:

```
cnrtNotifier_t notifier_start;
cnrtNotifier_t notifier_end;
cnrtCreateNotifier(&notifier_start);
cnrtCreateNotifier(&notifier_end);
```

[6] CNRT Cambricon Neuware Runtime Developer’ s Guide-CN-v2.x

```
float timeTotal = 0.0;
struct timeval start;
struct timeval end;
gettimeofday(&start, NULL);
cnrtPlaceNotifier(notifier_start, pQueue);
ret = cnrtInvokeKernel_V2(reinterpret_cast<void *>(&ROIPoolingKernel),
  dim, params, c, pQueue);
cnrtPlaceNotifier(notifier_end, pQueue);
ret = cnrtSyncQueue(pQueue);
gettimeofday(&end, NULL);
time_use = ((end.tv_sec- start.tv_sec) * 1000000
  + (end.tv_usec- start.tv_usec)) / 1000.0;
cnrtNotifierElapsedTime(notifier_start, notifier_end, &timeTotal);
printf("Hardware Total Time: %.3f ms\n", timeTotal / 1000.0);
cnrtDestroyNotifier(&notifier_start);
cnrtDestroyNotifier(&notifier_end);
```

5.2.7 Synchronization

5.2.7.1 sync_all

__sync_all() is used to synchronize all the cores inside multiple clusters. When all the cores in all clusters reach the synchronization point, the execution will continue.The number of cores participating in the synchronization is determined by the type of the task. For example, when the task type is UNION1, only 4 cores participate in synchronization; when the task type is UNION2, the number of cores participating in the synchronization is 8, and other types are analogized. Please refer to section 6.2 for the task type.

5.2.7.2 sync_cluster

__sync_cluster() is used to synchronize all the cores inside a cluster. When all the cores in a cluster reach the synchronization point, the execution will continue.

5.2.8 Memory Access Consistency

5.2.8.1 NRAM Memory Access Consistency

Because the nram space is exclusive to a single core, the memory access consistency of parallel program is not considered. Because the single-core period of the read or write nram space is executed sequentially, the memory access consistency of single-core execution is not considered.

5.2.8.2 WRAM Memory Access Consistency

Because the wram space is exclusive to a single core, the memory access consistency of parallel program is not considered. Because the single-core period of the read or write wram space is executed sequentially, the memory access consistency of single-core execution is not considered.

5.2.8.3 LDRAM Memory Access Consistency

Because the ldram space is exclusive to a single core, the memory access consistency of parallel program is not considered. Because the single-core period of the read or write ldram space is executed sequentially, the memory access consistency of single-core execution is not considered.

5.2.8.4 SRAM Memory Access Consistency

Because the sram space is shared by multiple cores, so that developers need to ensure that multiple cores cannot concurrently read and write the same sram address.

5.2.8.5 GDRAM Memory Access Consistency

Because the gdram space is shared by multiple cores, so that developers need to ensure that multiple cores cannot concurrently read and write the same gdram address.

Attention:

The data in same row (32 bytes) of LDRAM/GDRAM cannot be modified concurrently between different tasks (cores).

5.2.9 Example

The following is an example of a Kernel program of multi-core matrix multiplication. First, the output is initialized to 0 by task 0, then the 4 tasks are synchronized and the input matrix is divided into rows and columns, and the computation is performed concurrently, and the computing result is stored in the DDR.

```
__mlu_entry__ void MatrixMulKernel(int32_t* src1, int32_t* src2, int32_t* dst) {
  if (taskId == 0) {
    __nram__ int32_t dst_nram[4][32];
    __bang_write_zero(dst_nram, 4 * 32);
    __memcpy(dst, dst_nram, 4 * 32 * sizeof(int32_t), NRAM2GDRAM);
  }
  __sync_all();
  for (int32_t j = 0; j < 32; j++) {
    for (int32_t k = 0; k < 32; k++)
      dst[taskIdX * 32 + j] += src1[taskIdX * 32 + k] * src2[k * 32 + j];
  }
}
```

An example of a Host program corresponding to multi-core matrix multiplication is as follows:

```
cnrtDev_t dev;
cnrtRet_t ret;
cnrtInit(0);
cnrtGetDeviceHandle(&dev, 0);
```

```
cnrtSetCurrentDevice(dev);
cnrtQueue_t pQueue;
cnrtCreateQueue(&pQueue);
cnrtDim3_t dim;   dim.x = 4;   dim.y = 1;  dim.z = 1;
int len1 = 4 * 32;  int len2 = 32 * 32;  int len3 = 4 * 32;
int src1[4][32];  int src2[32][32];  int dst[4][32];
INPUT(src1, src2)

cnrtFunctionType_t c = CNRT_FUNC_TYPE_UNION1;
int *mlu_pa, *mlu_pb, *mlu_pc;
cnrtMalloc(reinterpret_cast<void**>(&mlu_pa), len1 * sizeof(int));
cnrtMalloc(reinterpret_cast<void**>(&mlu_pb), len2 * sizeof(int));
cnrtMalloc(reinterpret_cast<void**>(&mlu_pc), len3 * sizeof(int));
ret = cnrtMemcpy(mlu_pa, src1, len1*sizeof(int), CNRT_MEM_TRANS_DIR_HOST2DEV);
ret = cnrtMemcpy(mlu_pb, src2, len2*sizeof(int), CNRT_MEM_TRANS_DIR_HOST2DEV);
cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &mlu_pa, sizeof(mlu_pa));
cnrtKernelParamsBufferAddParam(params, &mlu_pb, sizeof(mlu_pb));
cnrtKernelParamsBufferAddParam(params, &mlu_pc, sizeof(mlu_pb));
ret = cnrtInvokeKernel_V2(reinterpret_cast<void**>(&MatrixMulKernel),
  dim, params, c, pQueue);
ret = cnrtSyncQueue(pQueue);
int result[4][32];
ret = cnrtMemcpy(result, mlu_pc, len3 * sizeof(int),
  CNRT_MEM_TRANS_DIR_DEV2HOST);

ret = cnrtDestroyQueue(pQueue);
cnrtFree(mlu_pa);
cnrtFree(mlu_pb);
cnrtFree(mlu_pc);
cnrtDestroy();
```

5.3 CNCC Compilation Options

Table 5.4: Common Compilation Options

Common Options	Description
-E	The compiler only performs the preprocessing steps to generate the preprocessed file.
-S	The compiler only performs the steps of preprocessing and compiling to generate the MLISA assembly file.
-c	The compiler only performs the steps of preprocessing, compiling, and assembling to generate an object file in ELF format.
-o	Write output to the specified file.
-x	Specify the programming language for the input file, e.g.:bang, etc.
–target=	Specify the format of the generated target file on the MLU. The file is linked with the target file on the target HOST platform as the executable program on the target HOST, so that its value format is binary file format on the target HOST, e.g: x86_64, armv7a, etc.
–bang-mlu-arch=	Specify the MLU architecturefor the input BANG C program, e.g. MLU270, MLU100, 1H8, etc. If you use MLU200, then the output ELF file will be a fatbin file that has instructions for MLU220 and MLU270.
–bang-stack-on-ldram	Whether the stack is placed on LDRAM. The stack is placed on NRAM by default. If this option is enabled, the stack will be placed on LDRAM.
–cnas-path=	Specify the path of the assembler.

5.4 Framework Integration

There are two ways to integrate a BANG C program with a machine learning framework:

The first way is to compile the specific BANG C implementation file into a dynamic link library, and call the function in the dynamic link library to compute the operator in the framework, as shown in Figure The BANG C Program Integrates the Framework with a Dynamic Link Library .

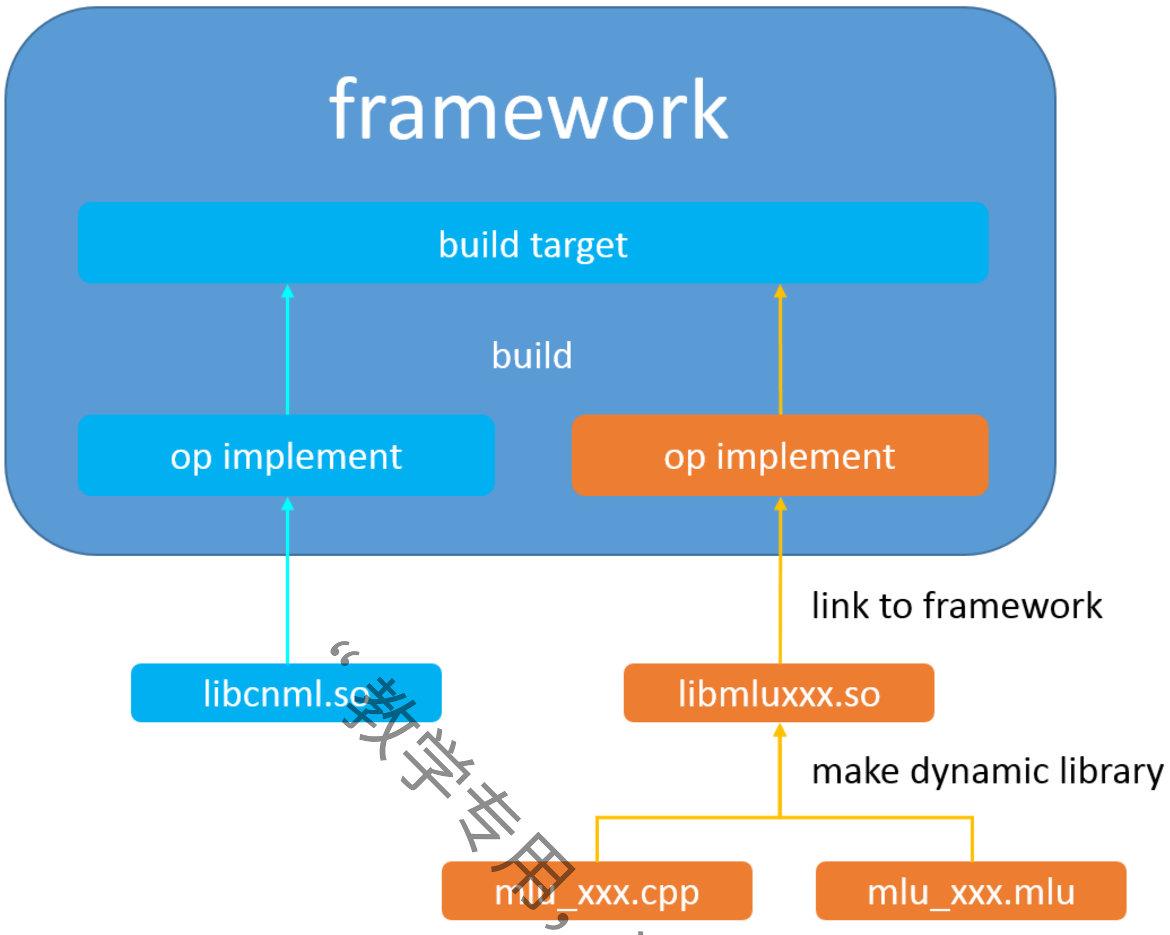

Fig. 5.4: The BANG C Program Integrates the Framework with a Dynamic Link Library

The second way is to directly integrate the BANG C source code into the machine learning framework and compile the BANG C source code, as shown in Figure The BANG C Program Integrates Framework with Source Code .

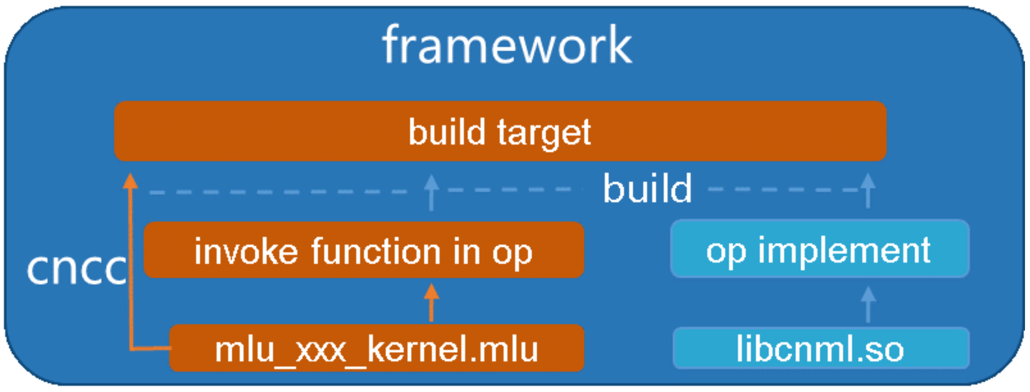

Fig. 5.5: The BANG C Program Integrates Framework with Source Code

We take the integration of upsample_layer and caffe framework as an example to explain the specific integration process. The flow chart is shown in Figure The Intergration of upsample_layer and Caffe Framework .

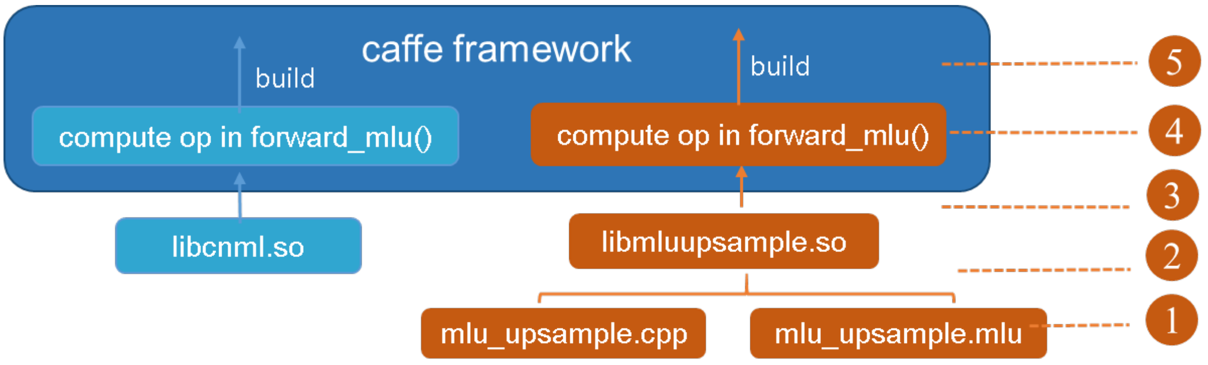

Fig. 5.6: The Intergration of upsample_layer and Caffe Framework

1. First write the corresponding operator. There are two files: mlu_upsample.cpp, mlu_upsample.mlu. Please refer to demo for specific code.
2. Compile the two files in step 1) into the dynamic link library libmluupsample.so. The compile command is shown in Makefile in the examples.
3. Create a new caffe/src/caffe/mlu_layers folder and copy libmluupsample.so in step 2) to this folder, and add a link of the dynamic link library to the caffe/src/caffe/CMakeLists file:
   (a) link_directories(./mlu_layers)
   (b) link_libraries(libmluupsample.so)
4. Modify the framework and call the function in the dynamic link library libmluupsample.so in step 2) in caffe; add a new file caffe/include/caffe/layers/mlu_upsample_layer.hpp and caffe/src/caffe/layers/ mlu_upsample_layer.cpp; declare and implement the MLUUpsample-Layer and the forward_mlu function in file caffe/src/caffe/layers/ mlu_upsample_layer.cpp.
5. Modify caffe/src/caffe/layer_factory.cpp, and add GetUpsampleLayer function. Please refer to the the example in the samples for specific code.
6. Just recompile caffe.

6 BANG C Primer

The BANG C language is the extension based on the C language. This chapter mainly introduces the extension and restriction of BANG C language in syntax.

6.1 Basic Data Type

Table 6.1: BANG C Language Supports the Following Basic Data Types

Basic Data Type	Length	Description
int8_t	1 byte	1-byte integer
uint8_t	1 byte	1-byte unsigned integer
int16_t	2 bytes	2 bytes integer
uint16_t	2 bytes	2 bytes unsigned integer
int32_t	4 bytes	4 bytes integer
uint32_t	4 bytes	4 bytes unsigned integer
half	2 bytes	half-precision floating-point data, using the IEEE-754 fp16 format
float	4 bytes	IEEE-754 fp32 format floating point type, currently only supports type conversion
char	1 byte	corresponding C Language char type
int8	1 byte	Int8 is an 8-bit fixed-point integer used to represent a floating-point vector as an 8-bit fixed-point vector. int8 and int type position together with float type scale representing floating point data[7]
int16	2 bytes	Int16 is an 16-bit fixed-point integer used to represent a floating-point vector as an 16-bit fixed-point vector. int16 and int type position together with float type scale representing floating point data[7]
bool	1 byte	corresponding C Language bool type
pointer	8 bytes	pointer type

Note: Using digit constant with floating point, if digit ends with “f”, it means that the digit is float, otherwise, it is half. For example, 0.1f is float and 0.1 is half.

[7] The actual floating point number represented by each value in the int8 type vector is:float = int8 * (2 ^ position) / scale , int8?[-128, 127], scale?[1, 2)

6.2 Key Words

6.2.1 Multi-core Parallel Key Words

The built-in variables related to multi-core parallel programming are as follows:

Table 6.2: Built-in Multi-core Parallel Variables

coreDim	coreId	clusterDim	clusterId
taskDim	taskDimX	taskDimY	taskDimZ
taskId	taskIdX	taskIdY	taskIdZ

6.2.2 Key Words of Address Space

The keywords that represent the variable address space are as follows: __nram__, __wram__, __ldram__, __mlu_shared__.

6.3 Function Modifier

The BANG C program has three types of functions, KERNEL’s entry function, Func function and Device function. The last two functions can be called by Entry function. For more information, please refer to Kernel section.

6.4 Modifier of Address Space

The storage of MLU includes GPR, NRAM/WRAM, LDRAM/GDRAM, etc. The BANG C language provides three modifiers to describe the address space of the variable, and the programmer controls the use of the address space. Local variables without any modifiers are variables in the stack space. The current implementation of CNCC defaults to placing the stack space in NRAM. When the stack space exceeds the upper limit of NRAM capacity (CNCC will report an error), the user can use the compile option –bang -stack-on-ldram to specify the stack in LDRAM.

This section introduces the basic concepts of BANG C language address space and its programming rules and precautions.

Table 6.3: Address Space Summary

Address Space	Scope	Variable Declaration	Scalar Computation	Stream Computation
NRAM	TASK	support	support	support
WRAM	TASK	support	not support	part (__conv and __ mlp, and only as their weight parameter)
SRAM	TASK	support	not support	not support
LDRAM	TASK	support	support	not support
GDRAM	KERNEL	support	support	not support

6.4.1 NRAM

The nram space is the on-chip storage of the MLU computation core. Its main function is to compute the space for storing data as a stream instruction. For example, when performing vector addition, both source data involved in the operation must be in nram and the computing result should be in nram. An example of vector addition is given below. When vector computation is executed, the source data need to be moved from gdram to nram, then vector addition should be executed on nram and a result vector will be gotten. After the operation, the result will be stored back to gdram.

The modifier is __nram__, indicating that the modified variable is a variable on the nram space.

```
__mlu_entry__ void SVAddKernel(half* src1, half* src2, half* dst) {
  __nram__ half src1_nram[128];
  __nram__ half src2_nram[128];
  __nram__ half dst_nram[128];

  __memcpy(src_nram, src1, 128 * sizeof(half), GDRAM2NRAM);
  __memcpy(dst_nram, src2, 128 * sizeof(half), GDRAM2NRAM);
  __bang_add(dst_nram, src1_nram, src2_nram, 128);
  __memcpy( dst, dst_nram, 128 * sizeof(half), NRAM2GDRAM);
}
```

The BANG C language supports scalar operations on nram spatial variables, such as defining scalars on two nrams, and some scalar operations can be performed on the two variables such as addition, subtraction, multiplication, division, and comparison. At the same time, the BANG C language also supports stream operations of nram data, such as defining 2 vectors on nram, and the 2 vectors can be added, subtracted, multiplied, and divided. nram is on-chip storage, its access efficiency is relatively high but the space is limited. For example, the MLU 100 architecture nram is only 512 KB, so BANG C programmers need to effectively use the space on the nram to improve the performance of the program.

Attention:

The address space of the stream computing variables can be nram. And the address space of the loop control variable can not be nram, it can be auto on stack.

6.4.2 WRAM

The wram space, which is the on-chip storage of the MLU computation core. wram mainly stores the weights used by the neural network computation. An example of a code that computes a convolution is given below. Before computing conv, the input data needs to be moved to nram, and the filter needs to be moved to wram.

The modifier is __wram__, indicating that the modified variable is a variable on the wram space.

```
__mlu_entry__ void ConvKernel(half* out_data,
  half* in_data,
  half* filter_data,
  int in_channel,
  int in_height,
  int in_width,
  int filter_height,
  int filter_width,
  int stride_height,
  int stride_width,
  int out_channel) {

  __nram__ half nram_out_data[OUT_DATA_NUM];
  __nram__ half nram_in_data[IN_DATA_NUM];
  __wram__ half wram_filter[FILTER_DATA_NUM];

  __memcpy(nram_in_data, in_data,
    IN_DATA_NUM * sizeof(half), GDRAM2NRAM);
  __memcpy(wram_filter, filter_data,
    FILTER_DATA_NUM * sizeof(half), GDRAM2WRAM);

  __bang_conv(nram_out_data, nram_in_data, wram_filter, IN_CHANNEL,
    IN_HEIGHT, IN_WIDTH, FILTER_HEIGHT, FILTER_WIDTH,
    STRIDE_HEIGHT, STRIDE_WIDTH, OUT_CHANNEL);

  __memcpy(out_data, nram_out_data,
    OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

Variables in the wram space can only store weights related to neural network operations, such as the weight of operations such as conv/mlp, and do not support any other type of operation.

6.4.3 SRAM

The sram space is the on-chip storage shared by the MLU computation core. Its main function is to store data as a stream instruction. An example of vector addition is given below.

```
__mlu_entry__ void SVAddKernel(half* src1, half* src2, half* dst) {
  __mlu_shared__ half src1_sram[128];
  __mlu_shared__ half src2_sram[128];
  __mlu_shared__ half dst_sram[128];
  __nram__ half src1_nram[128];
  __nram__ half src2_nram[128];
  __nram__ half dst_nram[128];

  __memcpy(src1_sram, src1, 128 * sizeof(half), GDRAM2SRAM);
  __memcpy(src2_sram, src2, 128 * sizeof(half), GDRAM2SRAM);
  __memcpy(src1_nram, src1_sram, 128 * sizeof(half), SRAM2NRAM);
  __memcpy(src2_nram, src2_sram, 128 * sizeof(half), SRAM2NRAM);
  __bang_add(dst_nram, src1_nram, src2_nram, 128);
  __memcpy(dst_sram, dst_nram, 128 * sizeof(half), NRAM2SRAM);
  __memcpy( dst, dst_sram, 128 * sizeof(half), SRAM2GDRAM);
}
```

Sram is on-chip storage, its access efficiency is relatively high but the space is limited.

Table 6.4: Values of Task Types

Task Types	dim.x	dim.y	dim.z
BLOCK	1	1	1
UNION1	4N	any value	any value
UNION2	8N	any value	any value
UNION4	16N	any value	any value
UNION8	32N	any value	any value
UNION16	64N	any value	any value

Note:

N is time slices.

Table 6.5: Share Memory Restrictions

Task Types	dim.x	dim.y	dim.z
BLOCK	not supported	not supported	not supported
UNION1	4 cores in one UNION share __mlu_shared	not shared	not shared
UNION2	4 cores in one UNION share __mlu_shared	not shared	not shared
UNION4	4 cores in one UNION share __mlu_shared	not shared	not shared
UNION8	4 cores in one UNION share __mlu_shared	not shared	not shared
UNION16	4 cores in one UNION share __mlu_shared	not shared	not shared

Table 6.6: Communication of Different Task Types

Task Types	dim.x	dim.y	dim.z
BLOCK	not supported	not supported	not supported
UNION1	not supported	not supported	not supported
UNION2	communication between clusters is supported	not supported	not supported
UNION4	communication between clusters is supported	not supported	not supported
UNION8	communication between clusters is supported	not supported	not supported
UNION16	communication between clusters is supported	not supported	not supported

Note:

The example of communication between clusters is if (clusterId == 0) __memcpy(dst, src, size, SRAM2SRAM, dst_cluster_id). UNION8 is upper bound of MLU270. UNION16 is upper bound of MLU290.

As we can see from the table above, BLOCK task type dim.x, dim.y, dim.z is 1, share memory is not supported, communication between clusters is not supported; UNION1 task type dim.x is 4N(N is time slices), so in dim.x direction, 4 cores in one UNION share __mlu_shared, as it has only one cluster, communication between clusters is not supported.

UNION2 task dim.x is 8N, 4 cores in each UNION share __mlu_shared, communication between two clusters is supported; UNION4 task dim.x is 16N, 4 cores in each UNION share __mlu_shared, communication between four clusters is supported; UNION8 task dim.x is 32N, 4 cores in each UNION share __mlu_shared, communication between eight clusters is supported;

UNION16 task dim.x is 16N, 4 cores in each UNION share __mlu_shared, communication between sixteen clusters is supported. In dim.y and dim.z direction, UNIONx task share memory is not shared, and communication between clusters is not supported.

The modifier is __mlu_shared__, indicating that the modified variable is a variable on the sram space.

The grammer rule of using keyword __mlu_shared__ is as follows:

1. in mlu_entry or mlu_global function body

```
__mlu_entry__ void kernel() {
__mlu_shared__ int array1[100];  // OK
static __mlu_shared__ int array2[100];  // OK
__mlu_shared__ int array3[100] = {0};  // error: initialization is not supported for __mlu_
↪shared__ variables
__mlu_shared__ int array4[100] = foo();  // error: initialization is not supported for __mlu_
↪shared__ variables
__mlu_shared__ half var1;  // OK
__mlu_shared__ half var2 = 1.0;  // error: initialization is not supported for __mlu_shared__␣
↪variables
}
```

2. in mlu_device or mlu_func function body

```
 __mlu_device__ void kernel1() {
 __mlu_shared__ int array1[100];  // OK
 static __mlu_shared__ int array2[100];  // OK
 __mlu_shared__ int array3[100] = {0};  // error: initialization is not supported for __mlu_
↪shared__ variables
 __mlu_shared__ int array4[100] = foo();  // error: initialization is not supported for __mlu_
↪shared__ variables
 __mlu_shared__ half var1;  // OK
 __mlu_shared__ half var2 = 1.0;  // error: initialization is not supported for __mlu_shared__␣
↪variables
}


 __mlu_func__ void kernel2() {
 __mlu_shared__ int array1[100];  // OK
 static __mlu_shared__ int array2[100];  // OK
 __mlu_shared__ int array3[100] = {0};  // error: initialization is not supported for __mlu_
↪shared__ variables
 __mlu_shared__ int array4[100] = foo();  // error: initialization is not supported for __mlu_
↪shared__ variables
 __mlu_shared__ half var1;  // OK
 __mlu_shared__ half var2 = 1.0;  // error: initialization is not supported for __mlu_shared__␣
↪variables
}
```

3. global variables declared in file domain

```
static __mlu_shared__ int array1[100] = {0};  // error: initialization is not supported for __
↪mlu_shared__ variables
static __mlu_shared__ int array2[100];  // OK
static __mlu_shared__ int array3[] = {0};  // error: initialization is not supported for __mlu_
↪shared__ variables
```

```
static __mlu_shared__ int array4[100] = foo();  // error: initialization is not supported for _
↪_mlu_shared__ variables
static __mlu_shared__ half var1;  // OK
static __mlu_shared__ half var2 = 1.0;  // error: initialization is not supported for __mlu_
↪shared__ variables

__mlu_entry__ void kernel() {
}
```

4. global variables declared cross file domain

```
extern __mlu_shared__ int array1[];  // OK
extern __mlu_shared__ int array2[100];  // error: __shared__ variable 'array1' cannot be
↪'extern'
extern __mlu_shared__ int array3[] = {0};  // warning: 'extern' variable has an initializer [-
↪Wextern-initializer]
extern __mlu_shared__ half var1;  // error: __mlu_shared__ variable 'var1' cannot be 'extern'
extern __mlu_shared__ half var2 = 1.0;  // error: __mlu_shared__ variable 'var2' cannot be
↪'extern'

__mlu_entry__ void kernel() {
}
```

6.4.4 LDRAM

The ldram space is the off-chip storage. Each MLU computation core has a separate ldram space that is primarily used to store the temporary data to be computed. The following is an example of using ldram, which performs vector addition on the 2 vectors on nram and stores the result in an ldram variable.

The modifier is __ldram__, indicating that the modified variable is a variable on the ldram space.

```
__ldram__ half ldram_tmp[LEN];
__nram__ half nram1[LEN];
__nram__ half nram2[LEN];
__nram__ half nram3[LEN];
__bang_add(nram3, nram2, nram1, LEN * sizeof(half));
__memcpy(ldram_tmp, nram3, LEN * sizeof(half), NRAM2LDRAM);
```

The BANG C language supports scalar operations on ldram spatial variables, such as defining scalars on two ldrams, and some scalar operations can be performed on the 2 variables such as addition, subtraction, multiplication, division, and comparison. The BANG C language does not support stream operations on ldram spatial data. If users want to compute the vector stored in ldram, it is necessary to explicitly move the vector data to the nram space and then perform a vector operation on the nram. Ldram is the space on the DDR, its size can be configured, please refer to the CNDRV Guide for configuration.

6.4.5 GDRAM

Similar to ldram, gdram is also off-chip storage, and all MLU computation cores share space to access gdram. BANG now provides a modifier for the gdram space.An example of declaring a gdram variable and storing it in a nram variable is given below.

The modifier is __mlu_device__, indicating that the modified variable is a variable on the gdram space.

```
__mlu_device__  half gdram_tmp[LEN];
__nram__   half nram_tmp[LEN];
__memcpy(nram_tmp, gdram_tmp, LEN * sizeof(half), GDRAM2NRAM);
```

One of the functions of the gdram space is to transfer data between HOST and MLU, such as the input data of the kernel, and the computation results of the kernel. The transfer of data between HOST and MLU is done by calling the cnrtMemcpy interface. The following example copies the kernel' s input data from HOST to gdram.

```
cnrtMemcpy(d_a, h_a_half, data_num * sizeof(half),
  CNRT_MEM_TRANS_DIR_HOST2DEV);
```

Similarly, if users want to transfer the computation result of MLU from gdram to HOST, they can refer to the following example.

```
cnrtMemcpy(mlu_result, d_c, data_num * sizeof(half),
  CNRT_MEM_TRANS_DIR_DEV2HOST);
```

The BANG C language supports scalar operations on gdram spatial variables, such as defining scalars on 2 gdrams, and some scalar operations can be performed on the two variables such as addition, subtraction, multiplication, division, and comparison. The BANG C language does not support the stream operation on gdram spatial data. If the user wants to compute the vector stored in gdram, it is necessary to explicitly move the modified vector data to the nram space and then perform vector operation on nram. The size of gdram can be configured. Please refer to the CNDRV Guide for configuration.

6.4.6 Restrictions of Address Space

Table 6.7: Restrictions of Address Space

Restriction	Description
Restriction of NRAM	When the NRAM pointer is configured as a vector to compute parameter, the position it points to is required to be offset by an integer multiple of 64 byte from the first address of the vector data.
Restriction of WRAM	The space of WRAM only stores weight data, and the data length of WRAM variables must be an integer multiple of 512 bytes.
Restriction of LDRAM Memory Access	LDRAM memory size must be an integer multiple of 32 bytes. In addition, when using the pointer of LDRAM to perform vector _memcpy, the position it points to is required to be offset by 32 byte integer times from the first address of the vector data.
Restriction of GDRAM Memory Access	GDRAM memory size must be an integer multiple of 32 bytes. In addition, when using the pointer of GDRAM to perform vector _memcpy, the position it points to is required to be offset by 32 byte integer times from the first address of the vector data.

6.5 Data Migration

6.5.1 Implicit Data Migration

The migration of implicit data is completed automatically by the compiler and the programmer do not need to participate. MLU requires that all scalar computations be performed in the GPR. When the scalar defined on ldram/gdram/nram is computed, the compiler will automatically insert the load/store instruction to move the data to the GPR for computation. After the computation is completed, the compiler will write the result on the GPR back to ldram/gdram/nram. The whole data migration is automatically completed by the compiler.

6.5.2 Explicit Data Migration

MLU requires that all stream computations be performed in nram, and the migration of stream data requires programmers to explicitly program. The BANG C language provides the __memcpy interface for data migration. The interface is defined as follows:

```
__memcpy(void *dst, void *src, int32_t Bytes, mluMemcpyDirection_t direct)
```

The direct parameter is an enumerated type that indicates the direction in which the data is migrated. The current directions supporting data migration are as follows:

Table 6.8: The Current Directions Supporting Data Migration

Parameter	Data Migration	Value of Direct
gdram2nram	from gdram to nram	GDRAM2NRAM
gdram2wram	from gdram to wram	GDRAM2WRAM
gdram2sram	from gdram to sram	GDRAM2SRAM
nram2gdram	from nram to gdram	NRAM2GDRAM
wram2gdram	from wram to gdram	WRAM2GDRAM
sram2gdram	from sram to gdram	SRAM2GDRAM
ldram2nram	from ldram to nram	LDRAM2NRAM
ldram2wram	from ldram to wram	LDRAM2WRAM
ldram2sram	from ldram to sram	LDRAM2SRAM
nram2ldram	from nram to ldram	NRAM2LDRAM
wram2ldram	from wram to ldram	WRAM2LDRAM
sram2ldram	from sram to ldram	SRAM2LDRAM
nram2nram	from nram to nram	NRAM2NRAM
sram2sram	from sram to sram	SRAM2SRAM
wram2sram	from wram to sram	WRAM2SRAM
sram2wram	from sram to wram	SRAM2WRAM
sram2nram	from sram to nram	SRAM2NRAM
nram2sram	from nram to sram	NRAM2SRAM

The Bytes of the parameter must be dividable by the following value on MLU100 and MLU270:

Table 6.9: Restrictions of Parameter Value

Parameter	MLU100	MLU270
gdram2nram	32	1
gdram2wram	512	512
gdram2sram	not support	1
nram2gdram	32	1
wram2gdram	512	512
sram2gdram	not support	1
ldram2nram	32	1
ldram2wram	512	512
ldram2sram	not support	1
nram2ldram	32	1
wram2ldram	512	512
sram2ldram	not support	1
nram2nram	32	128
sram2sram	not support	1
wram2sram	not support	512
sram2wram	not support	512
sram2nram	not support	1
nram2sram	not support	1

6.6 Using Function Call

All functions are inlined in earlier versions and function call is not supported. After MLU270, we introduce function call with dynamic stack. This chapter demonstrates this new feature.

6.6.1 Using Modifier

Function modifier or addressing space moidifier can be specified explicitly.

6.6.1.1 Explicit Function Modifier

MLU270 provides three function modifiers described in Function Modifier . For common MLU functions only __mlu_device__ and __mlu_func_ can be used.

```
__mlu_device__ int f();  // Using __mlu_device__ function modifier for uninlined function.
__mlu_func__ int f();  // Using __mlu_func__ function modifier for inlined function.
    __ldram__ int f();  // Error! Unsupported modifier.
__nram__ int f();  // Error! Unsupported modifier.
```

6.6.1.2 Explicit Parameter Pointer' s Addressing Space

Specify address space when define a funtion parameter as a pointer.

```
__mlu_func__ int f(float* p);  // Default addressing space in stack.
__mlu_func__ int f(float __nram__ *p); // Explicit nram addressing space.
```

6.6.1.3 Variable Declaration

Global variable declaration should specify addressing space and be initialized statically. Otherwise, report errors.

```
__ldram__ float a; // Global variable in ldram.
__nram__ int a = f();  // Error! Global variable should not be initialized dynamically.
```

6.6.2 Byte Alignment for Structure

Data alignment is processed by compiler automatically but can also be manipulated by programmers through #pragma pack() directive and structure attribute.

6.6.2.1 Default Behaviour

The default alignment is set according to the maximum structure member.

```
struct test_pack {
  char vchar;
  int8_t vi8;
  int16_t vi16;
  int32_t vi32;
  half vf16;
};
int pack = sizeof(struct test_pack);  // pack is 12. Alignment is 4 for the
                                      // maximum structure member is int32_t.
```

6.6.2.2 Use #pragma pack(n) Directive

The directive instructs the compiler to pack structure members with particular alignment. The actual alignment is the minimum of n and structure' s natural alignment.

```
#pragma pack(1)
  struct test_pack_1 {
    char vchar;
    int8_t vi8;
    int16_t vi16;
    int32_t vi32;
    half vf16;
  };
  int pack_1 = sizeof(struct test_pack_1);  // pack_1 is 10
#pragma pack(8)
  struct test_pack_8 {
    int8_t vi8;
    int16_t vi16;
    char vchar;
    half vf16;
    int32_t vi32;
  };
  int pack_8 = sizeof(struct test_pack_8);  // pack_8 is 12. The alignment is 4 not 8.
// Cancel custom alignment, restore default alignment.
#pragma pack()
  struct tets_pack_cancel {
    int32_t vi32;
    int8_t vi8;
    char vchar;
    half vf16;
    int16_t vi16;
  };
  int pack_cancel = sizeof(struct test_pack_cancel);  // pack_cancel is 12. Default behaviour.
......
// Push the current packing alignment value on the internal compiler stack,
// and set the current packing alignment value to 16.
#pragma pack(push, 16)
......
// Pop from internal compiler stack and set alignment.
#pragma pack(pop);
......
```

6.6.2.3 Use Structure Attribute

Programmers can use __attribute__((aligned(n))) or __attribute__((packed)) structure attribute to set or cancel alignment.

```
struct __attribute__((aligned(128)))struct_test1 {
  char m1;
  int32_t m4;
  half m2;
};
int struct_1 = sizeof(struct struct_test1); // struct_1 is 128;
// Cancel and set to the minimum possible alignment.
struct __attribute__((packed))struct_test2 {
  char m1;
  int32_t m4;
  half m2;
};
int struct_2 = sizeof(struct struct_test2); // struct_2 is 7;
```

6.6.3 Function in C++ Class

For BANG C language, function call in C++ class is under developing. It’ s declaration and implementation must be separated and should be used with caution.

6.6.3.1 Function Declaration

Function declaration must have modifier either with __mlu_func__ for inlining or __mlu_device__ for uninlining.

6.6.3.2 Function Implementation

Function implementation can be inside or outside of class body.

```
class Point {
  public:
    half area();  // Error! Function modifier must be provided.
    __mlu_func__ half area() { return w*h; };  // Correct!
    __mlu_func__ half area();  //Correct!
};

__mlu_func_ half Point::area() {  // Correct!
  return w*h;
}
```

6.6.4 Variadic Function

Variadic functions take a variable number of arguments and declared with an ellipsis in place of the last parameter.

6.6.4.1 Defining Variadic Function

Variadic function has at least one named parameter and the ellipsis is the last parameter.

```
__mlu_device__ void _sum(...);  // Error! No named parameter.
__mlu_device__ void _sum(float **p, ..., float va_test);  // Error! Ellipsis should be at last.
__mlu_device__ void _sum(float **p, float va_test, ...);  // Correct!
```

6.6.4.2 Accessing the Arguments

To access the arguments, use the following macros.

1)Use _mlu_va_start to start iterating arguments. The macro is called with two arguments. The first is a variable with the type _mlu_va_list. The second is the name of the last named parameter of the function.

```
_mlu_va_list ap;
_mlu_va_start(ap, va_test);
```

2)Use _mlu_va_ptr to retrieve an argument. Each invocation of this macro yields a pointer to the next argument. The first argument is the _mlu_va_list. The second is the type of the next argument passed to the function.

```
float **p;
int **p2;
*p = _mlu_va_ptr(ap, float);  // Return pointer to the next float type argument.
*p2 = _mlu_va_ptr(ap, int);  // Return pointer to the next int type argument.
```

3)Use _mlu_va_end to free _mlu_va_list.

```
_mlu_va_end(ap); // Free va_list.
```

6.6.5 Considerations

To choose which way to implement a function, inlining or with dynamic stack, take the following two points into considerations. Inlining is recommended.

1)Recursive function is easy to implement and produces a smaller code size while extra overhead for function calling. If performance is preferred instead of code simplicity, rewrite the function inlined.

2)Performance degradation occurs for overhead in function call. Before entering a function, parameters have to be pushed onto stack and general registers should be saved and restored later. Besides, addressing based on stack frame is indirect, which is slower than symbol addressing. Frequently used data structures and operators placed in stack for indirect addressing hurt performance.

6.6.6 Programming Example

6.6.6.1 Recursive Function Example

The following example demonstrates calling recursive fibonacci procedure in MLU kernel.

```
#include "mlu.h"
#define DATA_SIZE 32
// __mlu_device__ modifier for recursive function.
__mlu_device__ int fibonacci(int a) {
  if (a < 2) {
    return 1;
  } else {
    return fibonacci(a-1) + fibonacci(a-2);
  }
}

// size should be smaller then 33
__mlu_entry__ void kernel(int32_t *a, int size) {
  __nram__ int32_t a_tmp[DATA_SIZE];

  // generating fibonacci numbers.
  for (int i = 0; i < DATA_SIZE; i++) {
    a_tmp[i] = fibonacci(i);
  }
  __memcpy(a, a_tmp, size*sizeof(int32_t), NRAM2GDRAM);
}
```

6.6.6.2 Function in C++ Class Example

This example shows how to use function in C++ class.

```
#include "mlu.h"
#define DATA_SIZE 16
struct P {
  half w;
  half h;
};

class Point {
  public:
    half w, h;
    // Explicit funcion modifier.
    __mlu_func__ P init(half _w, half _h);
    __mlu_func__ half area();
};

// Function implementation is outside of class body.
__mlu_func__ P Point::init(half _w, half _h) {
  w = _w;
  h = _h;
  P p;
  p.w = _w;
  p.h = _h;
  return p;
}

__mlu_func__ half Point::area() {
  return w * h;
}

__mlu_device__ half getArea(Point p) {
  half t = p.area();
  return t;
}

__mlu_entry__ void kernel(half *src2) {
  Point p;
  P p1;
  p1 = p.init(*src2, 2);
  half t = getArea(p);
  if (p1.w * p1.h == t && t == p.area()) {
    printf("PASSED");
  }
}
```

6.6.6.3 Variadic Function Example

This example shows how to use variadic function.

```
#include "mlu.h"

__mlu_device__ void _sum(float **pptr1, float **pptr2, float **pptr3,
                         float va_test,
                         float var_float, ...) {
  _mlu_va_list ap;
  _mlu_va_start(ap, var_float);
  *pptr1 =_mlu_va_ptr(ap, float);
  *pptr2 =_mlu_va_ptr(ap, float);
  *pptr3 =_mlu_va_ptr(ap, float);
  _mlu_va_end(ap);
}

__mlu_entry__ void kernel(float* out1, float* out2, float* out3,
                          float g_n0, float g_n1,
                          float g_n2, float g_n3) {
  float buffer[48];
  float * p_arr1 = buffer;
  float * p_arr2 = buffer + 16;
  float * p_arr3 = buffer + 32;
  *p_arr1 = 100;
  *p_arr2 = 200;
  *p_arr3 = 300;
   _sum((float**)&p_arr1, (float**)&p_arr2, (float**)&p_arr3, 3.0,  g_n0, g_n1, g_n2, g_n3);
  __memcpy(out1, p_arr1, 16 * sizeof(float), NRAM2GDRAM);
  __memcpy(out2, p_arr2, 16 * sizeof(float), NRAM2GDRAM);
  __memcpy(out3, p_arr3, 16 * sizeof(float), NRAM2GDRAM);
}
```

6.7 Using Embedded Assembly

Embedded assembly is also supported in BANG C Language. Assembly means MLISA assembly IR(Intermediate Representation) produced by CNCC. This chapter demonstrates this feature.

6.7.1 Example

```
__mlu_entry__ void kernel(uint32_t *miss) {

  uint32_t miss_start = 0;
  uint32_t miss_stop = 0;
  __asm__ volatile(
      "mv.sreg.gpr %%perf_start, 1;\n\t"
      "mv.sreg.gpr %%perf_read, 1;\n\t"
      "mv.gpr.sreg %[miss_start], %%perf_cache_miss_num;\n\t"
      "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;\n\t"
      "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;\n\t"
      "mv.sreg.gpr %%perf_stop, 1;\n\t"
      "mv.sreg.gpr %%perf_read, 1;\n\t"
      "mv.gpr.sreg %[miss_stop], %%perf_cache_miss_num;\n\t"
      :[miss_start]"+&r"(miss_start),[miss_stop]"+&r"(miss_stop)
  );
      *miss = miss_stop - miss_start;
}
```

Embedded assembly begins with __asm__ modifier. Volatile means compiler do not optimize code, leaves the assemble code as it is. Mv.sreg.gpr and mv.gpr.sreg are assembly instruction. %%perf_start represents assemble instruction

operand, which is perf_start special reg. %[miss_start] represents gpr specified by variable miss_start, noted after ":". [miss_start]" +&r" (miss_start) "+" represents readable(input operand) and writable(output operand);

"r" represents general register. (miss_start) represents input operand C variable passed by C program and output operand C variable passed to C program. In the example above, %[miss_start] read from C miss_start variable, after executing

the instruction "mv.gpr.sreg %[miss_start], %%perf_cache_miss_num;nt", it writes %[miss_start] register, and passed the value to C miss_start variable out. Thus, after executing the whole assembly program, we can get miss_stop and miss_start.

So we can calculate the miss number by miss_stop sub miss_start in C program.

6.8 Restrictions of Language Features

The current version of the BANG C language does not support some C language features. Please refer to the table below for details.

Table 6.10: Restrictions of Language Features

Restricted C Language features	Description
Global variable	The BANG C language only supports global variables modified by NRAM/WARM/LDRAM/GDRAM.
static variable	The BANG C language does not support the static variables modified by static.
C library function	The BANG C language does not support calling standard C library function.
extern function and variable	The BANG C language support extern functions and variables declared by extern.
function pointer	The BANG C language does not support function pointer.
Pointer assignment	Pointers pointing to different address spaces in the BANG C language cannot be assigned to each other
Label address-of	The BANG C language does not support Label address-of.
function call	The BANG C language does not support entry function nesting and calling other entry functions.
function parameter	The BANG C language does not support calling entry function. struct, union and other cluster type parameters and their pointers.
multi-file programming	Programs belonging to the same kernel in the BANG C language must be in the same compilation unit.
multi-function programming in files	The BANG C language does not support writing multiple entry functions in the same compilation unit.

7 Built-in Function

The BANG C language provides a variety of built-in functions. Built-in functions are provided by compiler which user do not need to include header files when using the function. This chapter lists the built-in functions that BANG C currently supports for users to refer to.

7.1 Scalar Operation Function

7.1.1 abs

int abs (int __x);

short abs (short __x);

Calculate the absolute value of data.

- Parameters:
  <__x> The source operand
- Return Value:
  The absolute value of operand
- Remarks:
  This interface is supported on MLU100 and MLU270.

7.1.2 ceil

half ceil (half __x);

Find the minimum integer that is not smaller than the half type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The result integer
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.3 ceilf

float ceilf (float __x);

Find the minimum integer that is not smaller than the float type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value: The result integer
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.4 cosf

float cosf (float __x);

Calculate the cosine of the float type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The cosine value of the operand
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.5 floor

half floor (half __x);

Find the maximum integer that is not bigger than the half type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The result integer
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.6 floorf

float floorf (float __x);

Find the maximum integer that is not bigger than the float type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The result integer
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.7 fabsf

float fabsf (float __x);

Calculate the absolute value of the float type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The absolute value of operand
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.8 fmaxf

float fmaxf (float __x, float __y);

Compare the two float type data <__x> and <__y>, and find the larger one between them.

- Parameters:
  <__x> The first source operand
  <__y> The second source operand
- Return Value:
  The larger operand
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.9 fminf

float fminf (float __x, float __y);

Compare the two float type data <__x> and <__y>, and find the smaller one between them.

- Parameters:
  <__x> The first source operand
  <__y> The second source operand
- Return Value:
  The smaller operand
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.10 log2f

float log2f (float __x);

Calculate the logarithms of the float type data <__x> based on 2.

- Parameters:
  <__x> The source operand
- Return Value:
  The logarithms of operand based on 2.
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.11 max

unsigned short max (unsigned short __x, unsigned short __y);

unsigned int max (unsigned int __x, unsigned int __y);

short max (short __x, short __y);

int max (int __x, int __y);

float max (float __x, float __y);

Find the larger one of two numbers.

- Parameters:
  <__x> The first source operand
  <__y> The second source operand
- Return Value:
  The larger one of two operands
- Remarks:
  The float version of the interface is supported on MLU270, and not supported on MLU100. The other version of the interfaces are supported on MLU100 and MLU270.

7.1.12 min

unsigned short min (unsigned short __x, unsigned short __y);

unsigned int min (unsigned int __x, unsigned int __y);

short min (short __x, short __y);

int min (int __x, int __y);

half min (half __x, half __y);

float min (float __x, float __y);

Find the smaller one of two numbers.

- Parameters:
  <__x> The first source operand
  <__y> The second source operand
- Return Value:
  The smaller one of two operands
- Remarks:
  The float version of the interface is supported on MLU270, and not supported on MLU100. The other version of the interfaces are supported on MLU100 and MLU270.

7.1.13 powf

float powf (float __x, float __y);

Calculate the <__y> power of the float type data <__x>.

- Parameters:

<__x> The first source operand
<__y> The second source operand

- Return Value:
  The result that <__y> power of <__x>
- Remarks:
  __x is 2 in current version. When __x is not 2, mlu assertion information is reported.
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.14 round

half round (half __x);

float roundf (float __x);

Round the data <__x> by 4/5.

- Parameters:
  <__x> The source operand
- Return Value:
  The rounding result
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.15 sinf

float sinf (float __x);

Calculate the sine of the float type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The sine of operand
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.16 sqrtf

float sqrtf (float __x);

Calculate the square root of the float type data <__x>.

- Parameters:
  <__x> The source operand
- Return Value:
  The square root of operand
- Remarks:
  This interface is supported on MLU270, and not supported on MLU100.

7.1.17 truncf

float truncf (float __x);

Truncate the float type data <__x> to a integer.

- Parameters:

  <__x> The source operand
- Return Value:

  The result value
- Remarks:

  This interface is supported on MLU270, and not supported on MLU100.

7.2 Scalar Type Conversion Function

Scalar type conversion function is supported both on MLU100 and MLU270.

7.2.1 Rounding Method introduction

Table 7.1: Rounding Method of Type Conversion Instruction

Rounding Method	Description	Before Rounding	After Rounding
tz	to zero	1.5	1
tz	to zero	1.4	1
tz	to zero	-1.5	-1
oz	off zero	1.5	2
oz	off zero	1.4	2
oz	off zero	-1.5	-2
rd	round	1.5	2
rd	round	1.4	1
rd	round	-1.5	-2
dn	down	1.5	1
dn	down	1.4	1
dn	down	-1.5	-2
up	up	1.5	2
up	up	1.4	2
up	up	-1.5	-1

7.2.2 ___half2int_tz

int __half2int_tz (half a);

The half type data is converted to int type data, and ‘tz’ means the result is rounded to the 0 direction.

- Parameters:
  - <a> The source operand
- Return Value:
  - The convertion result

7.2.3 ___half2int_oz

int __half2int_oz (half a);

The half type data is converted to int type data, and ‘oz’ means the result is rounded away from the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.4 __half2int_up

int __half2int_up (half a);

The half type data is converted to int type data, and ‘up’ means the result is rounded up.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.5 __half2int_dn

int __half2int_dn (half a);

The half type data is converted to int type data, and ‘dn’ means the result is rounded down.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.6 __half2int_rd

int __half2int_rd (half a);

The half type data is converted to int type data, and ‘rd’ means the result get from rounding.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.7 __half2short_tz

short __half2short_tz (half a);

The half type data is converted to short type data, and ‘tz’ means the result is rounded to the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:

The convertion result

7.2.8 __half2short_oz

short __half2short_oz (half a);

The half type data is converted to short type data, and ‘oz’ means the result is rounded away from the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.9 __half2short_up

short __half2short_up (half a);

The half type data is converted to short type data, and ‘up’ means the result is rounded up.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.10 __half2short_dn

short __half2short_dn (half a);

The half type data is converted to short type data, and ‘dn’ means the result is rounded down.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.11 __half2short_rd

short __half2short_rd (half a);

The half type data is converted to short type data, and ‘rd’ means the result gets from rounding.

- Parameters:
  <a> The source operand
- Return Value: The convertion result

7.2.12 __float2int_tz

int __float2int_tz (float a);

The float type data is converted to int type data, and 'tz' means the result is rounded to the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.13 __float2int_oz

int __float2int_oz (float a);

The float type data is converted to int type data, and 'oz' means the result is rounded away from the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.14 __float2int_up

int __float2int_up (float a);

The float type data is converted to int type data, and 'up' means the result is rounded up.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.15 __float2int_dn

int __float2int_dn (float a);

The float type data is converted to int type data, and 'dn' means the result is rounded down.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.16 __float2int_rd

int __float2int_rd (float a);

The float type data is converted to int type data, and 'rd' means the result gets from rounding.

- Parameters:
  <a> The source operand
- Return Value:

The convertion result

7.2.17 __float2short_tz

short __float2short_tz (float a);

The float type data is converted to short type data, and 'tz' means the result is rounded to the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.18 __float2short_oz

short __float2short_oz (float a);

The float type data is converted to short type data, and 'oz' means the result is rounded away from the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.19 __float2short_up

short __float2short_up (float a);

The float type data is converted to short type data, and 'up' means the result is rounded up.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.20 __float2short_dn

short __float2short_dn (float a);

The float type data is converted to short type data, and 'dn' means the result is rounded down.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.21 __float2short_rd

short __float2short_rd (float a);

The float type data is converted to short type data, and ‘rd’ means the result gets from rounding.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.22 __float2half_tz

half __float2half_tz (float a);

The float type data is converted to half type data, and ‘tz’ means the result is rounded to the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.23 __float2half_oz

half __float2half_oz (float a);

The float type data is converted to half type data, and ‘oz’ means the result is rounded away from the 0 direction.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.24 __float2half_up

half __float2half_up (float a);

The float type data is converted to half type data, and ‘up’ means the result is rounded up.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.25 __float2half_dn

half __float2half_dn (float a);

The float type data is converted to half type data, and ‘dn’ means the result is rounded down.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.2.26 ___float2half_rd

half __float2half_rd (float a);

The float type data is converted to half type data, and 'rd' means the result gets from rounding.

- Parameters:
  <a> The source operand
- Return Value:
  The convertion result

7.3 Scalar Atomic Function

Scalar atomic function is supported on MLU270, and not supported on MLU100.

7.3.1 ___bang_atomic_add

unsigned short __bang_atomic_add (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_add (short* dst, short* src1, short src2);

unsigned int __bang_atomic_add (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_add (int* dst, int* src1, int src2);

half __bang_atomic_add (half* dst, half* src1, half src2);

float __bang_atomic_add (float* dst, float* src1, float src2);

Read address src1 value, add unsigned short value <src2> to unsigned short value <*src1>, and store the original value of <src1> in <dst>.

These three operations are performed in one atomic transaction.

That is: <dst> = <src1>; <*src1> = <*src1> + <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

- Example:

```
#include "mlu.h"
  __mlu_entry__ void kernel(unsigned short* ac, unsigned short* data, unsigned short tid) {
  unsigned short index = tid * taskDimX + taskIdX;
  unsigned short value = data[index];
  __nram__ unsigned short v;
  __bang_atomic_add(&v, &ac[value], 1);
}
```

7.3.2 __bang_atomic_and

unsigned short __bang_atomic_and (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_and (short* dst, short* src1, short src2);

unsigned int __bang_atomic_and (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_and (int* dst, int* src1, int src2);

half __bang_atomic_and (half* dst, half* src1, half src2);

float __bang_atomic_and (float* dst, float* src1, float src2);

Apply logical AND operation to the <*src1> and <src2>, store the result in <*src1>, and store the original value of <src1> in <dst>. That is: <dst> = <src1>; <*src1> = <*src1> & <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.3.3 __bang_atomic_cas

unsigned short __bang_atomic_cas (unsigned short* dst, unsigned short* src1, unsigned short src2, unsigned short src3);

short __bang_atomic_cas (short* dst, short* src1, short src2, short src3);

unsigned int __bang_atomic_cas (unsigned int* dst, unsigned int* src1, unsigned int src2, unsigned int src3);

int __bang_atomic_cas (int* dst, int* src1, int src2, int src3);

half __bang_atomic_cas (half* dst, half* src1, half src2, half src3);

float __bang_atomic_cas (float* dst, float* src1, float src2, float src3);

Compare <*src1> and <src2>. If <src2> is equal to <*src1>, store the float value <src3> in <*src1>. Store the original value of <src1> in <dst>.

That is: <dst> = <src1>; <*src1> = (<*src1> == <src2>) ? <src3> : <*src1>

- Parameters:
  - <dst> The address of destination operand
  - <src1> The address of first operand
  - <src2> The second operand
  - <src3> The third operand
- Return Value:
  - The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

- Example:

```
#include "mlu.h"

__mlu_entry__ void kernel(unsigned short* worker, unsigned short* data, unsigned short tgt,␣
↪unsigned short* res, unsigned short tid) {
 unsigned short index = tid * taskDimX + taskIdX;
 unsigned short val = data[index];

 __nram__ unsigned short v;
 __bang_atomic_cas(&v, worker, tgt, val);
 __memcpy(&res[index*4], &v, 16, NRAM2GDRAM);
}
```

7.3.4 __bang_atomic_dec

unsigned short __bang_atomic_dec (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_dec (short* dst, short* src1, short src2);

unsigned int __bang_atomic_dec (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_dec (int* dst, int* src1, int src2);

half __bang_atomic_dec (half* dst, half* src1, half src2);

float __bang_atomic_dec (float* dst, float* src1, float src2);

Compare <src1> and <src2>. If <*src1> is bigger than <src2>, or the value of <*src1> is 0, store the int value <src2> in <*src1>; otherwise, reduce <*src1> by 1. Store the original value of <src1> in <dst>. That is: <dst> = <src1>; <*src1> = (<*src1> == 0 || <*src1> > <src2>) ? <src2> : (<*src1> - 1)

- Parameters:
  - <dst> The address of destination operand
  - <src1> The address of first operand
  - <src2> The second operand

- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

- Example:

```
 __mlu_entry__ void kernel(unsigned short* worker, unsigned short* data, unsigned short tid) {
 unsigned short index = tid * taskDimX + taskIdX;
 unsigned short value = data[index];
 unsigned short maxval = 2000;

 __nram__ unsigned short v;
 __bang_atomic_dec(&v, &worker[value], maxval);
}
```

7.3.5 __bang_atomic_exch

unsigned short __bang_atomic_exch (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_exch (short* dst, short* src1, short src2);

unsigned int __bang_atomic_exch (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_exch (int* dst, int* src1, int src2);

half __bang_atomic_exch (half* dst, half* src1, half src2);

float __bang_atomic_exch (float* dst, float* src1, float src2);

Store <src1> in <dst>. Store <src2> in <*src1>. That is: <dst> = <src1>; <*src1> = <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

- Example:

```
 __mlu_entry__ void kernel(unsigned short* worker, unsigned short* data, int* res, unsigned␣
↪short tid) {
 unsigned short index = tid * taskDimX + taskIdX;
 unsigned short val = data[index];

 __nram__ unsigned short v;
 __bang_atomic_exch(&v, worker, val);
 __memcpy(&res[index*2], &v, 16, NRAM2GDRAM);
}
```

7.3.6 __bang_atomic_inc

unsigned short __bang_atomic_inc (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_inc (short* dst, short* src1, short src2);

unsigned int __bang_atomic_inc (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_inc (int* dst, int* src1, int src2);

half __bang_atomic_inc (half* dst, half* src1, half src2);

float __bang_atomic_inc (float* dst, float* src1, float src2);

Compare <*src1> and <src2>. If <*src1> is smaller than <src2>, increase <*src1> by 1; otherwise, set <*src1> to 0. Store the original value of <src1> in <dst>.

That is: <dst> = <src1>; <*src1> = (<*src1> >= <src2>) ? 0 : (<*src1> + 1)

- Parameters:
  - <dst> The address of destination operand
  - <src1> The address of first operand
  - <src2> The second operand
- Return Value:
  - The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

- Example:

```
__mlu_entry__ void kernel(unsigned short* worker, unsigned short* data, unsigned short tid) {
 unsigned short index = tid * taskDimX + taskIdX;
 unsigned short value = data[index];
 unsigned short maxval = 2000;

 __nram__ unsigned short v;
 __bang_atomic_inc(&v, &worker[value], maxval);
}
```

7.3.7 __bang_atomic_max

unsigned short __bang_atomic_max (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_max (short* dst, short* src1, short src2);

unsigned int __bang_atomic_max (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_max (int* dst, int* src1, int src2);

half __bang_atomic_max (half* dst, half* src1, half src2);

float __bang_atomic_max (float* dst, float* src1, float src2);

Take the larger value from <*src1> and <src2>, and store it in <*src1>. Store the original value of <src1> in <dst>. That is: <dst> = <src1>; <*src1> = (<*src1> > <src2>) ? <*src1> : <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.3.8 __bang_atomic_min

unsigned short __bang_atomic_min (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_min (short* dst, short* src1, short src2);

unsigned int __bang_atomic_min (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_min (int* dst, int* src1, int src2);

half __bang_atomic_min (half* dst, half* src1, half src2);

float __bang_atomic_min (float* dst, float* src1, float src2);

Take the smaller value from two float values <*src1> and <src2>, and store it in <*src1>. Store the original value of <src1> in <dst>.

That is: <dst> = <src1>; <*src1> = (<*src1> < <src2>) ? <*src1> : <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.3.9 __bang_atomic_or

unsigned short __bang_atomic_or (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_or (short* dst, short* src1, short src2);

unsigned int __bang_atomic_or (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_or (int* dst, int* src1, int src2);

half __bang_atomic_or (half* dst, half* src1, half src2);

float __bang_atomic_or (float* dst, float* src1, float src2);

Apply logical OR operation to <*src1> and <src2>, store the result in <*src1>, and store the original value of <src1> in <dst>. That is: <dst> = <src1>; <*src1> = <*src1> | <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.3.10 __bang_atomic_xor

unsigned short __bang_atomic_xor (unsigned short* dst, unsigned short* src1, unsigned short src2);

short __bang_atomic_xor (short* dst, short* src1, short src2);

unsigned int __bang_atomic_xor (unsigned int* dst, unsigned int* src1, unsigned int src2);

int __bang_atomic_xor (int* dst, int* src1, int src2);

half __bang_atomic_xor (half* dst, half* src1, half src2);

float __bang_atomic_xor (float* dst, float* src1, float src2);

Apply logical XOR operation to <*src1> and <src2>, store the result in <*src1>, and store the original value of <src1> in <dst>. That is: <dst> = <src1>; <*src1> = <*src1> ^ <src2>

- Parameters:
  <dst> The address of destination operand
  <src1> The address of first operand
  <src2> The second operand
- Return Value:
  The address of destination operand
- Remarks:

1. <dst> must point to nram address space;

2. <src1> must point to ldram/gdram address space.

7.4 Stream Operation Function

Attention:

Please align the parameters of the stream operation function. Otherwise, running the function will result in an error.

Compiler can report alignment errors, if the code of instruction’ s addressing mode is symbolic addressing or constant.

Compiler cannot check variable alignment or report errors, if the code of instruction’ s addressing mode is register indirect addressing.

Stream operation function is supported on MLU100 and MLU270. For more information about the function compatibility, please refer to Built-in Function Compatiblity between MLU100 and MLU270.

7.4.1 __bang_active_abs

void __bang_active_abs (half* dst, half* src, int elem_count);

Apply active (abs) operation on operand <src>, a vector of half type.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The range of <src> operand is (0.01, +500];
5. Error: 1.5%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.2 __bang_active_cos

void __bang_active_cos (half* dst, half* src, int elem_count);

void __bang_active_cos (float* dst, float* src, int elem_count);

Apply active (cosine) operation on operand <src>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. The range of <src> operand is [-2π, 2π];
5. Error: 2%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.3 __bang_active_exp

void __bang_active_exp (half* dst, half* src, int elem_count);

void __bang_active_exp (float* dst, float* src, int elem_count);

Apply active (exponent) operation src operand <src>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. The range of <src> operand is [-7.75, 10];
5. Error: 1%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.4 __bang_active_exp_less_0

void __bang_active_exp_less_0 (half* dst, half* src, int elem_count);

void __bang_active_exp_less_0 (float* dst, float* src, int elem_count);

Apply active (exponent) operation src operand <src> when src less than 0.

- Parameters:

<dst> The address of destination vector
<src> The address of source vector
<elem_count> Number of elements in source

- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of <src> operand is [-15.5, 0];
5. Error: 0.4%;
6. <dst> and <src> can be homologous operand.

7.4.5 __bang_active_gelu

void __bang_active_gelu (half* dst, half* src, int elem_count);

void __bang_active_gelu (float* dst, float* src, int elem_count);

Apply active (gelu) operation on operand <src>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of <src> operand is [-4, +4];
5. Error: 0.06%;
6. <dst> and <src> can be homologous operand.

7.4.6 __bang_active_gelup

void __bang_active_gelup (half* dst, half* src, int elem_count);

void __bang_active_gelup (float* dst, float* src, int elem_count);

Apply active (gelup) operation on operand <src>, which precision is higher than active(gelu).

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;

3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of <src> operand is [-4, +4];
5. Error: 0.06%;
6. <dst> and <src> can be homologous operand.

7.4.7 __bang_active_log

void __bang_active_log (half* dst, half* src, int elem_count);

void __bang_active_log (float* dst, float* src, int elem_count);

Apply active (log base e) operation on operand <src>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of <src> operand is [0.00002, 60000];
5. Error: 0.4%;
6. <dst> and <src> can be homologous operand.

7.4.8 __bang_active_recip

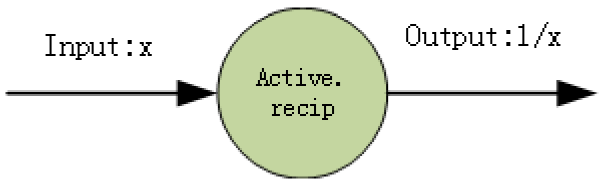

Fig. 7.1: Reciprocal Process

void __bang_active_recip (half* dst, half* src, int elem_count);

void __bang_active_recip (float* dst, float* src, int elem_count);

Apply active (reciprocal) operation on operand <src>.

Hint:

Recip means reciprocal.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. The range of <src> operand is (0.01, +500];
5. Error: 1.5%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

- Example:

```
__mlu_entry__ void kernel(half* y, half* x, int len) {
__nram__ half ny[LEN];
__nram__ half nx[LEN];

// copy data from gdram to nram
__memcpy(ny, y, LEN * sizeof(half), GDRAM2NRAM);
__memcpy(nx, x, LEN * sizeof(half), GDRAM2NRAM);

// invoke active abs option
__bang_active_recip(ny, nx, LEN);

// copy data from nram to gdram
__memcpy(y, ny, len * sizeof
(half), NRAM2GDRAM);

return;
}
```

7.4.9 __bang_active_recip_greater_1

void __bang_active_recip_greater_1 (half* dst, half* src, int elem_count);

void __bang_active_recip_greater_1 (float* dst, float* src, int elem_count);

Apply active (reciprocal) operation on operand <src> when src greater than 1.

Hint:

Recip means reciprocal.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The range of <src> operand is [1, 60000];
5. Error: 0.06%;
6. <dst> and <src> can be homologous operand.

7.4.10 __bang_active_relu

void __bang_active_relu (half* dst, half* src, int elem_count);

void __bang_active_relu (float* dst, float* src, int elem_count);

Apply active (relu) operation on operand <src>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The range of <src> operand is [-65504, +65504];
5. Error: none;
6. <dst> and <src> can be homologous operand.

7.4.11 __bang_active_rsqrt

void __bang_active_rsqrt (half* dst, half* src, int elem_count);

void __bang_active_rsqrt (float* dst, float* src, int elem_count);

Apply active (rsqrt) operation on operand <src>, i.e, 1/sqrt(src).

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;

2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of <src> operand is [0.005, 63487];
5. Error: 0.4%;
6. <dst> and <src> can be homologous operand.

7.4.12 ___bang_active_sigmoid

void __bang_active_sigmoid (half* dst, half* src, int elem_count);

void __bang_active_sigmoid (float* dst, float* src, int elem_count);

Apply active (sigmoid) operation on operand <src>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of src operand is [-65504, +65504];
5. Error: 0.1%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.13 ___bang_active_sin

void __bang_active_sin (half* dst, half* src, int elem_count);

void __bang_active_sin (float* dst, float* src, int elem_count);

Apply active (sine) operation on operand <src>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The range of src operand is [-2π, 2π];
5. Error: 2%;

6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.14 __bang_active_sqrt

void __bang_active_sqrt (half* dst, half* src, int elem_count);

void __bang_active_sqrt (float* dst, float* src, int elem_count);

Apply active (sqrt) operation on operand <src>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The range of <src> operand is [0, +65504];
5. Error: 1%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.15 __bang_active_tanh

void __bang_active_tanh (half* dst, half* src, int elem_count);

Apply active (tanh) operation on operand <src>, a vector of half type.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <elem_count> Number of elements in source
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The range of <src> operand is (0.01, +500];
5. Error: 1.5%;
6. Average relative error: within 1%. Average relative error = (Σ | CPU result - MLU result |)/(Σ | CPU result |);
7. <dst> and <src> can be homologous operand.

7.4.16 __bang_add

void __bang_add (half* dst, half* src0, half* src1, int elem_count);

void __bang_add (float* dst, float* src0, float* src1, int elem_count);

Add two vectors.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst> and <src0> can be homologous operand, <dst> and <src1> can be homologous operand.

- Example

```
#include "mlu.h"
#define DATA_SIZE 128
__mlu_entry__ void kernel(uint32_t size, half* c, half* a, half* b) {
  __nram__ half a_tmp[DATA_SIZE + DATA_SIZE];
  __nram__ half c_tmp[DATA_SIZE];
  __nram__ half b_tmp[DATA_SIZE];
  __memcpy(a_tmp + DATA_SIZE, a, size * sizeof(half), GDRAM2NRAM);
  __memcpy(b_tmp, b, size * sizeof(half), GDRAM2NRAM);

  __bang_add(c_tmp, a_tmp + DATA_SIZE, b_tmp, size);

  __memcpy(c, c_tmp, size * sizeof(half), NRAM2GDRAM);
}
```

7.4.17 __bang_add_const

void __bang_add_const (half* dst, half* src, half const_value, int elem_count);

void __bang_add_const (float* dst, float* src, float const_value, int elem_count);

Add vector src with const_value to result vector dst.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source operand vector
  - <const_value> The constant to add
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> * sizeof(type) must be dividable by 128;

2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. This interface is not supported on MLU100, and is supported on MLU270.

7.4.18 ___bang_avgpool

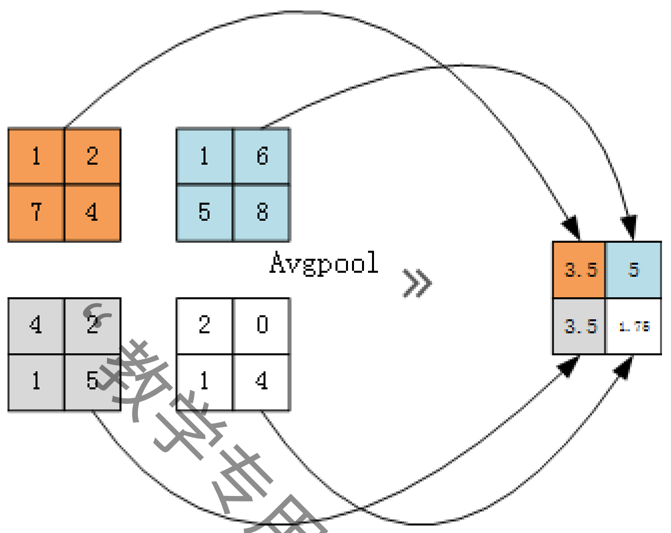

Fig. 7.2: Process of Avgpool

void __bang_avgpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width);

void __bang_avgpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_avgpool (float* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply avgpooling operation on src[height, width, channel], a three-dimensional matrix, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and in each window an avg number will be computed.

- Parameters:
  <dst> The address of destination matrix, and the matrix data order is HWC
  <src> The address of source matrix, and the matrix data order is HWC
  <channel> Input channel
  <height> The height of input map
  <width> The width of input map
  <kernel_height> The height of kernel
  <kernel_width> The width of kernnel
  <stride_x> Stride of X direction

<stride_y> Stride of Y direction

- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn't match the window size, these elemets will be discarded.
5. The result is a three-dimensional matrix with HWC data order.

- Example

```
#include "mlu.h"
#define CHANNELS 64
#define HEIGHT 9
#define WIDTH 9
#define KERNEL_HEIGHT 3
#define KERNEL_WIDTH 3
#define BOTTOM_DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))
#define TOP_DATA_COUNT \
  ((CHANNELS) * (HEIGHT / KERNEL_HEIGHT) * (WIDTH / KERNEL_WIDTH))

__mlu_entry__ void avgPoolingKernel(half* bottom_data, half* top_data,
                                    int channels, int height, int width,
                                    int pooled_height, int pooled_width) {
  __nram__ half a_tmp[BOTTOM_DATA_COUNT];
  __nram__ half b_tmp[TOP_DATA_COUNT];
  __memcpy(a_tmp, bottom_data, BOTTOM_DATA_COUNT * sizeof(half), GDRAM2NRAM);
  __bang_avgpool(b_tmp, a_tmp, CHANNELS, HEIGHT, WIDTH, KERNEL_HEIGHT, KERNEL_WIDTH);
  __memcpy(top_data, b_tmp, TOP_DATA_COUNT * sizeof(half), NRAM2GDRAM);
}
```

7.4.19 __bang_avgpool_bp

void __bang_avgpool_bp (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_avgpool_bp (float* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply avgpooling operation on src[height, width, channel], a three-dimensional matrix, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and in each window an avg number will be computed.

- Parameters:

<dst> The address of destination matrix, and the matrix data order is HWC
<src> The address of source matrix, and the matrix data order is HWC
<channel> Input channel
<height> The height of input map
<width> The width of input map

<kernel_height> The height of kernel
<kernel_width> The width of kernnel
<stride_x> Stride of X direction
<stride_y> Stride of Y direction

- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn't match the window size, these elemets will be discarded.
5. The result is a three-dimensional matrix with HWC data order.
6. Propagate direction is backforward.

- Example

```
#include "mlu.h"

__mlu_entry__ void PoolAvgKernel(half* output, half* input, int channels,
                                 int in_height, int in_width, int kernel_height,
                                 int kernel_width, int stride_x, int stride_y) {
  __nram__ half a_tmp[INPUT_COUNT];
  __nram__ half b_tmp[OUTPUT_COUNT];
  __memcpy(b_tmp, output, OUTPUT_COUNT * sizeof(half), GDRAM2NRAM);
  __memcpy(a_tmp, input, INPUT_COUNT * sizeof(half), GDRAM2NRAM);
  __bang_avgpool_bp(b_tmp, a_tmp, channels, in_height, in_width, kernel_height,
            kernel_width, stride_x, stride_y);
  __memcpy(output, b_tmp, OUTPUT_COUNT * sizeof(half), NRAM2GDRAM);
}
```

7.4.20 __bang_collect

void __bang_collect (half* dst, half* src, half* mask, int elem_count);

void __bang_collect (float* dst, float* src, float* mask, int elem_count);

Select number in one vector according to the corresponding values in another vector. The elements in <src> will be selected if corresponding elements in <mask> are not equal to zero. The result is the selected elements.

- Parameters:

<dst> The address of destination vector
<src> The address of first operand vector
<mask> The address of second operand vector
<elem_count> The elements number of vector

- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <mask> and <dst> must point to __nram__ space;

3. Offset of the position, which operand <src>, <mask>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. <dst> and <src> can be homologous operand.

- Example

```
__bang_collect
  (dst, src, mask, elem_count);
selectedNum[0] = dst[0];
selectedNum[1] = dst[1];
...
```

7.4.21 __bang_collect_bitindex

void __bang_collect_bitindex (half* dst, half* src, void* bitmask, int elem_count);

void __bang_collect_bitindex (float* dst, float* src, void* bitmask, int elem_count);

Select number in one vector according to the corresponding values in another vector. The elements in <src> will be selected if corresponding bit value in <bitmask> are not equal to zero. The result is the selected elements.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <bitmask> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <bitmask> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src>, <bitmask>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. <dst> and <src> can be homologous operand.

- Example

```
__bang_collec_bitindex
  (dst, src, bitmask, elem_count);
selectedNum[0] = dst[0];
selectedNum[1] = dst[1];
...
```

7.4.22 __bang_conv

void __bang_conv (float* dst, int16* src, int8* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (float* dst, int8* src, int8* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (float32* dst, int16* src, int16* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (half* dst, half* src, half* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output);

void __bang_conv (half* dst, int16* src, int16* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (half* dst, int16* src, int8* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (half* dst, int8* src, int8* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (int16* dst, int16* src, int16* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (int16* dst, int16* src, int8* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (int16* dst, int8* src, int8* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (float* dst, int16* src, int8* kernel, float* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (float* dst, int8* src, int8* kernel, float* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (float32* dst, int16* src, int16* kernel, float32* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (half* dst, int16* src, int16* kernel, half* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (half* dst, int16* src, int8* kernel, half* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (half* dst, int8* src, int8* kernel, half* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (int16* dst, int16* src, int16* kernel, int16* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (int16* dst, int16* src, int8* kernel, int16* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (int16* dst, int8* src, int8* kernel, int16* bias, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

void __bang_conv (float32* dst, int* src, int16* kernel, const int channal_input, const int height, const int width, const int kernel_height, const int kernel_width, const int stride_x, const int stride_y, const int channal_output, int fix_position);

Apply convolution operation on operand <src>

- Parameters:
  - <dst> The address of destination matrix, and the matrix has NHWC data order
  - <src> The address of source matrix, and the matrix has NHWC data order
  - <kernel> The address of filter matrix, and the matrix has NHWC data order
  - <bias> The address of bias matrix, and the matrix has NHWC data order
  - <channal_input> Input channel
  - <height> The height of input map
  - <width> The width of input map
  - <kernel_height> The height of filter
  - <kernel_width> The width of filter
  - <stride_x> The stride in x direction
  - <stride_y> The stride in y direction
  - <channal_output> Output channel
  - <fix_position> The sum of int16_pos,int8_pos of <src> and <kernel>
- Remarks

1. The operand <src>, <dst> and <bias> must point to __nram__ space;
2. <kernel> must point to __wram__ space;
3. <fix_position> can be immediate or register, and it's range is [-127, 127] on MLU270, range is [-63, 63] on MLU100;
4. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
5. <kernel> needs to reshape through interface cnrtFilterReshape. Please refer to below example for details;
6. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
7. <kernel> needs to reshape through interface cnrtFilterReshape. Please refer to below example

   for details;

8. <channal_input> * sizeof(type of input) must be dividable by 64 on MLU270, by 32 on MLU100;
9. <channal_output> * sizeof(type of output) must be dividable by 128 on MLU270, by 32 on MLU100;
10. Bang_conv with int input version is not recommended, because it is implemented by combination, with lower performance. __clang_bang_extra.h should be included when using this version. When channal_input*width*height > MAX_CONV_INPUT_BUFFER, MAX_CONV_INPUT_BUFFER/(width*height) should be dividable by 32.

- Example

Host:

```
...
//  prepare filter data for MLU
If (CNRT_RET_SUCCESS != cnrtFilterReshape(filter_half, ori_filter_data,
  //  CNRT_FLOAT16 is the dataType of ori_filter_data
  OUT_CHANNEL, FILTER_HEIGHT, FILTER_WIDTH, IN_CHANNEL, CNRT_FLOAT16)) {
  printf ("cnrtFliterReshapeHalf failed! \n");
  exit(-1);
}
...
//  copy data from cpu to mlu
if (CNRT_RET_SUCCESS != cnrtMemcpy(mlu_in_data, in_data_half,
  IN_DATA_NUM * sizeof(half), CNRT_MEM_TRANS_DIR_HOST2DEV)) {
  printf("cnrtMemcpy FAILED!\n");
  exit(-1);
}
if (CNRT_RET_SUCCESS != cnrtMemcpy(mlu_filter_data, filter_half,
  FILTER_DATA_NUM * sizeof(half), CNRT_MEM_TRANS_DIR_HOST2DEV)) {
  printf("cnrtMemcpy FAILED!\n");
  exit(-1);
}
...
//  Passing param
cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &mlu_out_data, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &mlu_in_data, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, & mlu_filter_data, sizeof(half*));
...
// InvokeKernel
if (CNRT_RET_SUCCESS !=
  cnrtInvokeKernel_V2((void*)&ConvKernel, dim, params, c, pQueue)) {
  printf("cnrtInvokeKernel FAILED!\n");
  exit(-1);
}
...
```

```
...
//  prepare filter data for MLU
```

```
If (CNRT_RET_SUCCESS != cnrtFilterReshape(filter_half, ori_filter_data,
  //  CNRT_FLOAT16 is the dataType of ori_filter_data
  OUT_CHANNEL, FILTER_HEIGHT, FILTER_WIDTH, IN_CHANNEL, CNRT_FLOAT16)) {
  printf ("cnrtFliterReshapeHalf failed! \n");
  exit(-1);
}
...
//  copy data from cpu to mlu
if (CNRT_RET_SUCCESS != cnrtMemcpy(mlu_in_data, in_data_half,
  IN_DATA_NUM * sizeof(int8), CNRT_MEM_TRANS_DIR_HOST2DEV)) {
  printf("cnrtMemcpy FAILED!\n");
  exit(-1);
}
if (CNRT_RET_SUCCESS != cnrtMemcpy(mlu_filter_data, filter_half,
  FILTER_DATA_NUM * sizeof(int8), CNRT_MEM_TRANS_DIR_HOST2DEV)) {
  printf("cnrtMemcpy FAILED!\n");
  exit(-1);
}
...
//  Passing param
cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &mlu_out_data, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &mlu_in_data, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, & mlu_filter_data, sizeof(half*));
...
// InvokeKernel
if (CNRT_RET_SUCCESS !=
  cnrtInvokeKernel((void*)&ConvKernel, dim, params, c, pQueue)) {
  printf("cnrtInvokeKernel FAILED!\n");
  exit(-1);
}
...
```

MLU:

```
#include "macro.h"
#include "mlu.h"
__mlu_entry__ void ConvKernel(half* out_data, int8* in_data, int8* filter_data,
  int in_channel, int in_height, int in_width, int filter_height, int filter_width,
  int stride_height, int stride_width, int out_channel) {
  __nram__ half nram_out_data[OUT_DATA_NUM];
  __nram__ int8 nram_in_data[IN_DATA_NUM];
  __nram__ int8 nram_filter[FILTER_DATA_NUM];
  __memcpy(nram_in_data, in_data, IN_DATA_NUM * sizeof(int8), GDRAM2NRAM);
  __memcpy(nram_filter, filter_data, FILTER_DATA_NUM * sizeof(int8), GDRAM2NRAM);
  __bang_conv(nram_out_data, nram_in_data, wram_filter, in_channel, in_height,
    in_width, filter_height, filter_width, stride_width, stride_height,
    out_channel, F16_INT8_INT8, 2);
  __memcpy(out_data, nram_out_data, OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

```
#include "macro.h"
#include "mlu.h"
__mlu_entry__ void ConvKernel(half* out_data, half* in_data, half* filter_data,
  int in_channel, int in_height, int in_width, int filter_height, int filter_width,
  int stride_height, int stride_width, int out_channel) {
  __nram__ half nram_out_data[OUT_DATA_NUM];
  __nram__ half nram_in_data[IN_DATA_NUM];
  __wram__ half wram_filter[FILTER_DATA_NUM];
  __memcpy(nram_in_data, in_data, IN_DATA_NUM * sizeof(half), GDRAM2NRAM);
  __memcpy(wram_filter, filter_data, FILTER_DATA_NUM * sizeof(half),
    GDRAM2WRAM);
  __bang_conv(nram_out_data, nram_in_data, wram_filter, in_channel, in_height,
    in_width, filter_height, filter_width, stride_height, stride_width, out_channel);
  __memcpy(out_data, nram_out_data, OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

7.4.22.1 Compatibility between MLU100 and MLU270

Table 7.2: Conv Data Types Supported on MLU100

Input Type	Kernel Type	Output Type
float16	float16	float16
int8	int8	float16

Table 7.3: Conv Data Types Supported on MLU270

Input Type	Kernel Type	Output Type	Bias Type
int8	int8	float16	none
int8	int8	float32	none
int8	int8	int16	none
int16	int8	float16	none
int16	int8	float32	none
int16	int8	int16	none
int16	int16	float16	none
int16	int16	float32	none
int16	int16	int16	none
int32	int16	float32	none
int8	int8	float16	float16
int8	int8	float32	float32
int8	int8	int16	int16
int16	int8	float16	float16
int16	int8	float32	float32
int16	int8	int16	int16
int16	int16	float16	float16
int16	int16	float32	float32
int16	int16	int16	int16

- Remarks
  1. Param fix_position’s value range on MLU100 is [-63, 63], on MLU270 is [-127, 127];
  2. Param channal_input * sizeof(src) on MLU100 should be dividable by 32, on MLU270 should be divided by 64;
  3. When output is 16bit, param channal_output * sizeof(half) on MLU100 should be dividable by 32, on MLU270 should be divided by 128;
  4. When output is 32bit, param channal_output * sizeof(float) on MLU100 should be dividable by 32, on MLU270 should be divided by 256;
  5. Bang_conv with int32 input version is not recommended, because it is implemented by combination, with lower performance. __clang_bang_extra.h should be included when using this version. When channal_input*width*height > MAX_CONV_INPUT_BUFFER,

MAX_CONV_INPUT_BUFFER/(width*height) should be dividable by 32.

7.4.23 __bang_count

void __bang_count (unsigned int* dst, half* src, int elem_count);

void __bang_count (unsigned int* dst, float* src, int elem_count);

Count the number of non-zero elements in the input vector.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <dst> must point to __nram__ space;
3. The __nram__ space to which the vector operand <dst> points must be at least 128 bytes on MLU270, at lease 32 bytes on MLU100;
4. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example

```
__bang_count
  (dst, src, elem_count);
unsigned int counter =
  ((unsigned int*)dst)[0];
```

7.4.24 __bang_count_bitindex

void __bang_count_bitindex (unsigned int* dst, half* src, int elem_count);

void __bang_count_bitindex (unsigned int* dst, float* src, int elem_count);

Count the number of non-zero bit value in the input vector.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <dst> must point to __nram__ space;
3. The __nram__ space to which the vector operand <dst> points must be at least 128 bytes
4. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example

```
__bang_count_bitindex(dst, src, elem_count);
unsigned int counter = ((unsigned int*)dst)[0];
```

7.4.25 __bang_cycle_add

void __bang_cycle_add (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_add (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src> into <src_elem_count> / <seg_elem_count> parts, then each element in each part ADD the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <seg> The address of second operand vector
  <src_elem_count> The elements number of <src> vector
  <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof(float) and <seg_elem_count> * sizeof(float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …).

- Example

```
#include "mlu.h"
#define N1 256
#define N2 65280
#define s1 N1
#define s2 N2
__mlu_entry__ void kernel(uint32_t size1, uint32_t size2, half* c, half* a,
                          half* b) {
  __nram__ half a_tmp[s1];
  __nram__ half b_tmp[s2];
  __nram__ half c_tmp[s2];
  __memcpy(a_tmp, a, s1 * sizeof(half), GDRAM2NRAM);
  __memcpy(b_tmp, b, s2 * sizeof(half), GDRAM2NRAM);
  __bang_cycle_add(c_tmp, b_tmp, a_tmp, size2, size1);
  __memcpy(c, c_tmp, s2 * sizeof(half), NRAM2GDRAM);
}
```

7.4.26 __bang_cycle_mul

void __bang_cycle_mul (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_mul (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src> into <src_elem_count> / <seg_elem_count> parts, then each element in each part MULTIPLY the corresponding element in <seg>, the result (low order) is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof(float) and <seg_elem_count> * sizeof(float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.4.27 __bang_cycle_sub

void __bang_cycle_sub (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_sub (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src> into <src_elem_count> / <seg_elem_count> parts, then each element in each part SUBTRACT the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof(float) and <seg_elem_count> * sizeof(float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.4.28 __bang_findfirst1

void __bang_findfirst1 (unsigned int* dst, half* src, int elem_count);

void __bang_findfirst1 (unsigned int* dst, float* src, int elem_count);

Find the first non-zero bit in the values of <src> which size is <elem_count>, and store the results in <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source operand vector
  <elem_count> The length of source operand vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).

- Example:

```
 __mlu_entry__ void kernel(uint32_t size, half* c, half* a) {
 __nram__ half a_tmp[DATA_SIZE + 64];
 __nram__ half c_tmp[DATA_SIZE + 64];
 __memcpy(a_tmp + 64, a, size * sizeof(half), GDRAM2NRAM);
 __bang_findfirst1((uint32_t*)(c_tmp + 64), a_tmp + 64, size);
 __memcpy(c, c_tmp + 64, size * sizeof(half), NRAM2GDRAM);
}
```

7.4.29 __bang_findlast1

void __bang_findlast1(unsigned int* dst, half* src, int elem_count);

void __bang_findlast1(unsigned int* dst, float* src, int elem_count);

Find the last non-zero bit in the values of <src> which size is <elem_count>, and store the results in <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source operand vector
  <elem_count> The length of source operand vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).

7.4.30 __bang_maskmove

void __bang_maskmove (half* dst, half* src, half* mask, int elem_count);

void __bang_maskmove (float* dst, float* src, float* mask, int elem_count);

Select the value in <src>, a vector of float/half type, which size is <elem_count>, according to the value of the vector <mask>, and store the result in <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source operand vector
  - <mask> The address of mask operand vector
  - <elem_count> The length of operand vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <mask> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src>, <mask> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. In fact, this instruction' s function is same with __bang_collect;
5. <dst> and <src> can be homologous operand.

- Example

..code:

```
__bang_maskmove
  (dst, src, mask, elem_count);
selectedNum[0] = dst[0];
selectedNum[1] = dst[1];
...
```

7.4.31 __bang_maskmove_bitindex

void __bang_maskmove_bitindex (half* dst, half* src, void* bitmask, int elem_count);

void __bang_maskmove_bitindex (float* dst, float* src, void* bitmask, int elem_count);

Select the value in <src> which size is <elem_count>, according to the value of the vector <bitmask>, and store the result in <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source operand vector
  - <bitmask> The address of mask operand vector
  - <elem_count> The length of operand vector
- Remarks

1. <elem_count> * sizeof(float) must be dividable by 512;
2. The operand <src>, <mask> and <dst> must point to __nram__ space;
3. Offset of the position, which operand <src>, <mask> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The value stored in <bitmask> is bit value;
5. <dst> and <src> can be homologous operand.

- Example

```
__bang_maskmove_bitindex
  (dst, src, bitmask, elem_count);
selectedNum[0] = dst[0];
```

```
selectedNum[1] = dst[1];
...
```

7.4.32 __bang_maximum

Find maximum value of each two corresponding elements in two vector that distance is <distance>.

void __bang_maximum (half* dst, half* src, int distance, int size);

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <distance> The distance between the two operand vector
  - <size> The elements number of vector
- Remarks

1. <size> * sizeof (type) must be dividable by 128
2. The operand <src> and <dst> must point to __nram__ space.
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
4. In fact, this function is equivalent to __bang_maxequal(dst, src, src+distance, size).

- Example:

```
#include "mlu.h"
#define CHANNELS 64
#define IN_HEIGHT 3
#define IN_WIDTH 3
#define INPUT_COUNT ((CHANNELS) * (IN_WIDTH) * (IN_HEIGHT))
#define OUTPUT_COUNT CHANNELS
__mlu_entry__ void PoolMaxKernel( half* output, half* input,
                                  int channels,
                                  int in_height, int in_width,
                                  int kernel_height, int kernel_width) {
  __nram__ half a_tmp[INPUT_COUNT*2];
  __nram__ half b_tmp[OUTPUT_COUNT*2];
  __memcpy(a_tmp + channels, input, INPUT_COUNT * sizeof(half), GDRAM2NRAM);
  __bang_maximum(b_tmp+channels, a_tmp+channels, in_width, CHANNELS);
  __memcpy(output, b_tmp+channels, OUTPUT_COUNT * sizeof(half), NRAM2GDRAM);
}
```

7.4.33 __bang_maxpool

void __bang_maxpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width);

void __bang_maxpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_maxpool (float* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply maxpooling operation on src[height, width, channel], a three-dimensional matrix of float type, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and the max element in each window will be selected.

- Parameters:
  <dst> The address of destination matrix, and the matrix data order is HWC
  <src> The address of source matrix, and the matrix data order is HWC
  <channel> Input channel
  <height> The height of input map
  <width> The width of input map
  <kernel_height> The height of sliding window
  <kernel_width> The width of sliding window
  <stride_x> Stride of X direction
  <stride_y> Stride of Y direction
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn’t match the window size, these elements will be discarded.
5. The result is a three-dimensional matrix with HWC data order.

- Example:

```
#include "mlu.h"
#define CHANNELS 64
#define HEIGHT 4
#define WIDTH 4
#define KERNEL_HEIGHT 2
#define KERNEL_WIDTH 2
#define BOTTOM_DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))
#define TOP_DATA_COUNT \
  ((CHANNELS) * (HEIGHT / KERNEL_HEIGHT) * (WIDTH / KERNEL_WIDTH))

__mlu_entry__ void maxPoolingKernel(half* bottom_data, half* top_data,
                                    int channels, int height, int width,
                                    int pooled_height, int pooled_width) {
  __nram__ half a_tmp[BOTTOM_DATA_COUNT];
  __nram__ half b_tmp[TOP_DATA_COUNT];
  __memcpy(a_tmp, bottom_data, BOTTOM_DATA_COUNT * sizeof(half), GDRAM2NRAM);
  __bang_maxpool(b_tmp, a_tmp, CHANNELS, HEIGHT, WIDTH, KERNEL_HEIGHT, KERNEL_WIDTH);
  __memcpy(top_data, b_tmp, TOP_DATA_COUNT * sizeof(half), NRAM2GDRAM);
}
```

7.4.34 ___bang_maxpool_index

void __bang_maxpool_index (unsigned short* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width);

void __bang_maxpool_index (unsigned short* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_maxpool_index (unsigned int* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply maxpooling operation on src[height, width, channel], a three-dimensional matrix, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and the index of the max element in each window will be selected.

- Parameters:
  <dst> The address of destination matrix, and the matrix data order is HWC
  <src> The address of source matrix, and the matrix data order is HWC
  <channel> Input channel
  <height> The height of input map
  <width> The width of input map
  <kernel_height> The height of kernel
  <kernel_width> The width of kernnel
  <stride_x> Stride of X direction
  <stride_y> Stride of Y direction
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn’t match the window size, these elemets will be discarded.
5. The result is a three-dimensional matrix with HWC data order.

- Example:

```
#include "mlu.h"
#define CHANNELS 64
#define HEIGHT 9
#define WIDTH 12
#define KERNEL_HEIGHT 5
#define KERNEL_WIDTH 4

#define BOTTOM_DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))
#define TOP_DATA_COUNT \
  ((CHANNELS) * (HEIGHT / KERNEL_HEIGHT) * (WIDTH / KERNEL_WIDTH))

__mlu_entry__ void MaxPoolIndexKernel(half* bottom_data, int16_t* top_data,
                                      int channels, int height, int width,
                                      int pooled_height, int pooled_width) {
```

```
  __nram__ half a_tmp[BOTTOM_DATA_COUNT];
  __nram__ uint16_t b_tmp[TOP_DATA_COUNT];
  __memcpy(a_tmp, bottom_data, BOTTOM_DATA_COUNT * sizeof(half), GDRAM2NRAM);

  __bang_maxpool_index(b_tmp, a_tmp, CHANNELS, HEIGHT, WIDTH,
                                    KERNEL_HEIGHT, KERNEL_WIDTH);

  __memcpy(top_data, b_tmp, TOP_DATA_COUNT * sizeof(int16_t), NRAM2GDRAM);
}
```

7.4.35 __bang_minpool

void __bang_minpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width);

void __bang_minpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_minpool (float* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply minpooling operation on src [height, width, channel], a three-dimensional matrix, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and the min element in each window will be selected.

- Parameters:
  <dst> The address of destination matrix, and the matrix data order is HWC
  <src> The address of source matrix, and the matrix data order is HWC
  <channel> Input channel
  <height> The height of input map
  <width> The width of input map
  <kernel_height> The height of sliding window
  <kernel_width> The width of sliding window
  <stride_x> Stride of X direction
  <stride_y> Stride of Y direction
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn' t match the window size, these elements will be discarded.
5. The result is a three-dimensional matrix with HWC data order.

7.4.36 __bang_minpool_index

void __bang_minpool_index (unsigned short* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width);

void __bang_minpool_index (unsigned short* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_minpool_index (unsigned int* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply minpooling operation on src[height, width, channel], a three-dimensional matrix, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and the index of the min element in each window will be selected.

- Parameters:
  - <dst> The address of destination matrix, and the matrix data order is HWC
  - <src> The address of source matrix, and the matrix data order is HWC
  - <channel> Input channel
  - <height> The height of input map
  - <width> The width of input map
  - <kernel_height> The height of sliding window
  - <kernel_width> The width of sliding window
  - <stride_x> Stride of X direction
  - <stride_y> Stride of Y direction
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn’t match the window size, these elements will be discarded.
5. The result is a three-dimensional matrix with HWC data order.

7.4.37 ___bang_mirror

void __bang_mirror (unsigned char* dst, unsigned char* src, int height, int width);

void __bang_mirror (unsigned short* dst, unsigned short* src, int height, int width);

void __bang_mirror (short* dst, short* src, int height, int width);

void __bang_mirror (half* dst, half* src, int height, int width);

void __bang_mirror (unsigned int* dst, unsigned int* src, int height, int width);

Apply mirror inversion operand on <src>, a matrix, which size is <height> * <width>, and store the result in <dst>.

- Parameters:
  - <dst> The address of destination matrix
  - <src> The address of operand matrix
  - <height> The width of <src>
  - <width> The height of <src>
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;

2. <height> and <width> must be dividable by 16;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. This interface is supported on MLU270, not supported on MLU100.

- Example:

```
#include "mlu.h"
#define WIDTH 64
#define HEIGHT 32
#define LEN WIDTH * HEIGHT

__mlu_entry__ void kernel(short* dst, short* src, int height, int width) {
  __nram__ short ny[LEN + 64];
  __nram__ short nx[LEN];

  __memcpy(nx, src, LEN * sizeof(short), GDRAM2NRAM);
  __bang_mirror(ny, nx, HEIGHT, WIDTH);
  __memcpy(dst, ny, sizeof(short) * LEN, NRAM2GDRAM);
  return;
}
```

7.4.38 __bang_mlp

void __bang_mlp (float* dst, int16* src, float* bias, int16* weight, const int height, const int width, int fix_position);

void __bang_mlp (float* dst, int16* src, float* bias, int8* weight, const int height, const int width, int fix_position);

void __bang_mlp (float* dst, int* src, float* bias, int16* weight, const int height, const int width, int fix_position);

void __bang_mlp (half* dst, half* src, half* bias, half* weight, const int height, const int width);

void __bang_mlp (half* dst, int16* src, half* bias, int16* weight, const int height, const int width, int fix_position);

void __bang_mlp (half* dst, int16* src, half* bias, int8* weight, const int height, const int width, int fix_position);

void __bang_mlp (half* dst, int8* src, half* bias, int8* weight, const int height, const int width, int fix_position);

void __bang_mlp (half* dst, int8* src, half* bias, int8* weight, const int height, const int width, int fix_position);

void __bang_mlp (int16* dst, int16* src, int16* bias, int16* weight, const int height, const int width, int fix_position);

void __bang_mlp (int16* dst, int16* src, int16* bias, int8* weight, const int height, const int width, int fix_position);

void __bang_mlp (int16* dst, int8* src, int16* bias, int8* weight, const int height, const int width, int fix_position);

Apply mlp operation, dst[1][height] = src[1][width] * weight[width][height] + bias[1][height], on operand <src>.

- Parameters:
  - <bias> The address of bias matrix
  - <dst> The address of destination matrix
  - <fix_position> The sum of int16_pos, int8_pos of <src> and <weight>
  - <height> The height of <weight>
  - <src> The address of source matrix
  - <weight> The address of weight matrix
  - <width> The width of <weight>
- Remarks

1. The operand <dst>, <src> and <bias> must point to __nram__ space;
2. <weight> must point to __wram__ space;
3. <height> must be dividable by 16 on MLU100, <height> * sizeof (half) must be dividable by 128 on MLU270;
4. <height> * sizeof (half) must be dividable by 128 on MLU270, <height> must be dividable by 16 on MLU100;
5. <width> * sizeof (type of input) must be dividable by 64 on MLU270, <width> must be dividable by 64 on MLU100;
6. <fix_position> must be immediate, it's range is [-127, 127];
7. Offset of the position, which operand <src>, <bias> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
8. <weight> needs to reshape through interface cnrtFilterReshape;
9. Mlp of int input type version is not recommended, because it is implemented by combination, with lower performance. __clang_bang_extra.h should be included when using this version.

- Example:

```
#include "mlu.h"
#define HEIGHT 64
#define WIDTH 64
#define POS 0

__mlu_entry__ void MlpKernel(half* out_data, int16_t* in_data,
                             int16_t* filter_data, half* bias_data,
                             int height, int width, int pos) {
 __nram__ half nram_out_data[HEIGHT];
 __nram__ half nram_bias_data[HEIGHT];
 __nram__ int16_t nram_in_data[WIDTH];
 __wram__ int16_t wram_filter[HEIGHT * WIDTH];

 __memcpy(nram_in_data, in_data, WIDTH * sizeof(int16_t), GDRAM2NRAM);
 __memcpy(nram_bias_data, bias_data, HEIGHT * sizeof(half), GDRAM2NRAM);
 __memcpy(wram_filter, filter_data, HEIGHT * WIDTH * sizeof(int16_t),
         GDRAM2WRAM);
```

```
  __bang_mlp(nram_out_data, nram_in_data, nram_bias_data, wram_filter,
             HEIGHT, WIDTH, POS);

  __memcpy(out_data, nram_out_data, HEIGHT * sizeof(half), NRAM2GDRAM);
}
```

7.4.38.1 Compatibility between MLU100 and MLU270

Table 7.4: mlp Data Types Supported on MLU100

dst type	src type	bias type	weight type
float16	float16	float16	float16
float16	int8	float16	int8

Table 7.5: mlp Data Types Supported on MLU270

dst type	src type	bias type	weight type
float16	int16	float16	int16
float16	int8	float16	int8
float16	fix16	float16	fix8
float32	fix16	float32	fix16
float32	fix8	float32	fix8
float32	fix16	float32	fix8
float32	int32	float32	fix16
fix16	fix16	fix16	fix16
fix16	fix8	fix16	fix8
fix16	fix16	fix16	fix8

· Remarks

1. param fix_position’ s value range on MLU100 is [-63, 63], on MLU270 is [-127, 127];
2. <height> * sizeof (half) must be dividable by 128 on MLU270, <height> is dividable by 16 on MLU100;
3. <width> * sizeof (src) must be dividable by 64 on MLU270, <width> is dividable by 64 on MLU100;
4. Mlp of int input type version is not recommended, because it is implemented by combination, with lower performance. __clang_bang_extra.h should be included when using this version.

7.4.39 __bang_mul

void __bang_mul (half* dst, half* src0, half* src1, int elem_count);

void __bang_mul (float* dst, float* src0, float* src1, int elem_count);

Multiply two vectors. The result (low order) is assigned to <dst>.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst> and <src0> can be homologous operand.

7.4.40 __bang_mul_const

void __bang_mul_const (half* dst, half* src0, half const_value, int elem_count);

void __bang_mul_const (float* dst, float* src0, float const_value, int elem_count);

Multiply a vector of float type with a given constant.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of operand vector
  <const_value> The constant to multiply
  <elem_count> The elements number of operand vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst> and <src0> can be homologous operand.

- Example:

```
#include "mlu.h"
#define DATA_SIZE 128
 __mlu_entry__ void kernel(uint32_t size, half* c, half* a, half* b) {
   __nram__ half a_tmp[DATA_SIZE];
   __nram__ half c_tmp[DATA_SIZE];
   __memcpy(a_tmp, a, size * sizeof(half), GDRAM2NRAM);
   __bang_mul_const(c_tmp, a_tmp, b[0], size);
```

```
  __memcpy(c, c_tmp, size * sizeof(half), NRAM2GDRAM);
}
```

7.4.41 __bang_pad

void __bang_pad (float* dst, float* src, int channel, int height, int width, int pad_height, int pad_width);

void __bang_pad (float* dst, float* src, int channel, int height, int width, int pad_top, int pad_bottom, int pad_left, int pad_right);

void __bang_pad (half* dst, half* src, int channel, int height, int width, int pad_height, int pad_width);

void __bang_pad (half* dst, half* src, int channel, int height, int width, int pad_top, int pad_bottom, int pad_left, int pad_right);

void __bang_pad (int* dst, int* src, int channel, int height, int width, int pad_top, int pad_bottom, int pad_left, int pad_right);

Apply zero-padding operation on operand <src>.

- Parameters:

  <channel> Size of channel
  <dst> The address of destination matrix, and the matrix has HWC data order
  <height> The height of <src>
  <pad_bottom> The bottom of pad
  <pad_height> The height of pad
  <pad_left> The left of pad
  <pad_right> The right of pad
  <pad_top> The top of pad
  <pad_width> The width of pad
  <src> The address of source matrix, and the matrix has HWC data order
  <width> The width of <src>

- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. <channel> * <width> * sizeof (float) must be dividable by 128;
3. <channel> * <width> * sizeof (half) must be dividable by 128 on MLU270, <channel> * sizeof (half) must be dividable by 128 on MLU100;
4. (<pad_height> + <pad_width>) * <channel> * sizeof (float) must be dividable by 128;
5. (<pad_height> + <pad_width>) * <channel> * sizeof (half) must be dividable by 128 on MLU270;
6. (<pad_left> + <pad_right>) * <channel> * sizeof (half) must be dividable by 128 on MLU270;
7. (<pad_height> * (<pad_width> + <pad_width> + <width>) + <pad_width>) * <channel> * sizeof (float) must be dividable by 128;
8. (<pad_height> * (<pad_width> + <pad_width> + <width>) + <pad_width>) * <channel> * sizeof (half) must be dividable by 128 on MLU270;
9. (<pad_top> * (<pad_left> + <pad_right> + <width>) + <pad_left>) * <channel> * sizeof (half) must be dividable by 128 on MLU270;

10. (<pad_bottom> * (<pad_left> + <pad_right> + <width>) + <pad_right>) * <channel> * sizeof (half) must be dividable by 128 on MLU270;
11. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯), both on MLU100 and MLU270.

- Example

```
__nram__ half
  ny[CHANNEL * (LENINPUT_H + 2 * LENPAD_H) * (LENINPUT_W + 2 * LENPAD_W)];
__nram__ half nx[CHANNEL * LENINPUT_H * LENINPUT_W];
__bang_pad(ny, nx, CHANNEL, LENINPUT_H, LENINPUT_W, LENPAD_H, LENPAD_W);
```

7.4.42 __bang_reduce_sum

void __bang_reduce_sum (half* dst, half* src, int elem_count);

void __bang_reduce_sum (float* dst, float* src, int elem_count);

Take every 128 bytes data from source operand <src> and calculate their sum. Store the result in destination operand <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> Number of elements in source
- Remarks

1. The first element in destination operand <dst> of every 128 bytes is the sum of every 128 bytes data, the other elements in destination operand <dst> of every 128 bytes is zero.
2. This instruction take every 128 bytes to calculate each time. When vector type is float, the function will take the 32 elements to calculate. When vector type is half, the function will take the 64 elements to calculate.
3. <elem_count> * sizeof(type) must be dividable by 128.

7.4.43 __bang_rotate90

void __bang_rotate90 (int8* dst, int8* src, int height, int width);

void __bang_rotate90 (char* dst, char* src, int height, int width);

void __bang_rotate90 (unsigned char* dst, unsigned char* src, int height, int width);

void __bang_rotate90 (half* dst, half* src, int height, int width);

void __bang_rotate90 (short* dst, short* src, int height, int width);

void __bang_rotate90 (unsigned short* dst, unsigned short* src, int height, int width);

void __bang_rotate90 (float* dst, float* src, int height, int width);

void __bang_rotate90 (int* dst, int* src, int height, int width);

void __bang_rotate90 (unsigned int* dst, unsigned int* src, int height, int width);

Rotate the <src>, a matrix, which size is <height> * <width>, by $\pi/2$ angle, and store the result in <dst>.

- Parameters:
  <dst> The address of destination matrix
  <src> The address of source operand matrix
  <height> The height of <src>
  <width> The width of <src>
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. The rotate direction is anti-clockwise.
3. <height> and <width> must be dividable by 16;

- Example:

```
__mlu_entry__ void kernel
 (unsigned char* dst, unsigned char* src, int height, int width) {
__nram__ unsigned char ny[LEN];
__nram__ unsigned char nx[LEN];

__memcpy
 (nx, src, LEN * sizeof
 (unsigned char), GDRAM2NRAM);

__bang_rotate90
 (ny, nx, height, width);

__memcpy
 (dst, ny, sizeof
 (unsigned char) * LEN, NRAM2GDRAM);

return;
}
```

7.4.44 __bang_rotate180

void __bang_rotate180 (int8* dst, int8* src, int height, int width);

void __bang_rotate180 (char* dst, char* src, int height, int width);

void __bang_rotate180 (unsigned char* dst, unsigned char* src, int height, int width);

void __bang_rotate180 (half* dst, half* src, int height, int width);

void __bang_rotate180 (short* dst, short* src, int height, int width);

void __bang_rotate180 (unsigned short* dst, unsigned short* src, int height, int width);

void __bang_rotate180 (float* dst, float* src, int height, int width);

void __bang_rotate180 (int* dst, int* src, int height, int width);

void __bang_rotate180 (unsigned int* dst, unsigned int* src, int height, int width);

Rotate the <src>, a matrix, which size is <height> * <width>, by π angle, and store the result in <dst>.

- Parameters:
  - <dst> The address of destination matrix
  - <src> The address of source operand matrix
  - <height> The height of <src>
  - <width> The width of <src>
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. The rotate direction is anti-clockwise.
3. <height> and <width> must be dividable by 16;

7.4.45 __bang_rotate270

void __bang_rotate270 (int8* dst, int8* src, int height, int width);

void __bang_rotate270 (char* dst, char* src, int height, int width);

void __bang_rotate270 (unsigned char* dst, unsigned char* src, int height, int width);

void __bang_rotate270 (half* dst, half* src, int height, int width);

void __bang_rotate270 (short* dst, short* src, int height, int width);

void __bang_rotate270 (unsigned short* dst, unsigned short* src, int height, int width);

void __bang_rotate270 (float* dst, float* src, int height, int width);

void __bang_rotate270 (int* dst, int* src, int height, int width);

void __bang_rotate270 (unsigned int* dst, unsigned int* src, int height, int width);

Rotate the <src>, a matrix, which size is <height> * <width>, by $3\pi/2$ angle, and store the result in <dst>.

- Parameters:
  - <dst> The address of destination matrix
  - <src> The address of source operand matrix
  - <height> The height of <src>
  - <width> The width of <src>
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. The rotate direction is anti-clockwise.
3. <height> and <width> must be dividable by 16;

7.4.46 __bang_select

void __bang_select (half* dst, half* src, half* index, int elem_count);

void __bang_select (float* dst, float* src, float* index, int elem_count);

Select number in one vector according to the corresponding values in another vector. The elements in <src> will be selected if corresponding elements in <index> are not equal to zero. The result is composed of two parts. The first part is the number of selected elements, and the second part is the selected elements.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <index> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The <src>, <index> and <dst> must point to __nram__ space;
3. Offset of the position, which <src>, <index>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst> is composed of two parts: the first 128 bytes stores the number of selected numbers on MLU270; the rest store the selected number;
5. <dst> stores the number of selected numbers in 32 bytes on MLU100;
6. <dst> and <src> can be homologous operand.

- Example:

```
__bang_select
(dst, src, index, elem_count);

// get the number of selected elements from the first line
unsigned int numberOfSelectedNumbers = *
(
(unsigned int*)dst); // dst[0]
// get the selected elements from the second line
selectedNum[0] = dst[32];
selectedNum[1] = dst[33];
...
```

```
__bang_select
(dst, src, index, elem_count);

// get the number of selected elements from the first line
unsigned short numberOfSelectedNumbers = *
(
(unsigned short*)dst); // dst[0]
// get the selected elements from the second line
selectedNum[0] = dst[16];
selectedNum[1] = dst[17];
...
```

7.4.47 __bang_select_bitindex

void __bang_select_bitindex (half* dst, half* src, void* bitindex, int elem_count);

void __bang_select_bitindex (float* dst, float* src, void* bitindex, int elem_count);

Select number in one vector according to the corresponding values in another vector. The elements in <src> will be selected if corresponding bit value in <bitindex> are not equal to zero. The result is composed of two parts. The first part is the number of selected elements, and the second part is the selected elements.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <bitindex> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 1024 on MLU270, be dividable by 64 on MLU100;
2. The <src>, <bitindex> and <dst> must point to __nram__ space;
3. Offset of the position, which <src>, <bitindex>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst> is composed of two parts: the first 128 bytes stores the number of selected numbers on MLU270; the rest store the selected number;
5. <dst> stores the number of selected numbers in 32 bytes on MLU100;
6. <dst> and <src> can be homologous operand.

- Example
  __bang_select_bitindex (dst, src, bitindex, elem_count);
  // get the number of selected elements from the first line unsigned short numberOf-SelectedNumbers = * ((unsigned short*)dst); // dst[0] // get the selected elements from the second line selectedNum[0] = dst[16]; selectedNum[1] = dst[17]; ⋯

7.4.48 __bang_square

void __bang_square (half* dst, half* src, int elem_count);

void __bang_square (float* dst, float* src, int elem_count);

Apply a square activation operation on <src>, a vector, and store the result in <dst>. That is: <dst> = <src> * <src>

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
__mlu_entry__ void kernel
 (uint32_t size, half* c, half* a) {
```

```
__nram__ half a_tmp[DATA_SIZE + 64];
__nram__ half c_tmp[DATA_SIZE];
__memcpy
 (a_tmp + 64, a, size * sizeof
 (half), GDRAM2NRAM);
__bang_square
 (c_tmp, a_tmp + 64, size);
__memcpy
 (c, c_tmp, size * sizeof
 (half), NRAM2GDRAM);
}
```

7.4.49 __bang_sub

void __bang_sub (half* dst, half* src0, half* src1, int elem_count);

void __bang_sub (float* dst, float* src0, float* src1, int elem_count);

Subtract two vectors.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst> and <src0> can be homologous operand, <dst> and <src1> can be homologous operand.

- Example:

```
#include "mlu.h"
#define DATA_SIZE 128
__mlu_entry__ void kernel(uint32_t size, float* c, float* a, float* b) {
  __nram__ float a_tmp[2 * DATA_SIZE];
  __nram__ float c_tmp[DATA_SIZE];
  __nram__ float b_tmp[DATA_SIZE];
  __memcpy(a_tmp + DATA_SIZE, a, size * sizeof(float), GDRAM2NRAM);
  __memcpy(b_tmp, b, size * sizeof(float), GDRAM2NRAM);
  __bang_sub(c_tmp, a_tmp + DATA_SIZE, b_tmp, size);
  __memcpy(c, c_tmp, size * sizeof(float), NRAM2GDRAM);
}
```

7.4.50 __bang_sub_const

void __bang_sub_const (half* dst, half* src, half const_value, int elem_count);

void __bang_sub_const (float* dst, float* src, float const_value, int elem_count);

Subtract vector src with const_value to result vector dst.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source operand vector
  - <const_value> The constant to sub
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> * sizeof(type) must be dividable by 128;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. This interface is not supported on MLU100, and is supported on MLU270.

7.4.51 ___bang_sumpool

void __bang_sumpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width);

void __bang_sumpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

void __bang_sumpool (float* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y);

Apply sumpooling operation on src[height, width, channel], a three-dimensional matrix, with sliding window [kernel_height, kernel_width] and stride [stride_x, stride_y], and in each window an sum number will be computed.

- Parameters:
  - <dst> The address of destination matrix, and the matrix data order is HWC
  - <src> The address of source matrix, and the matrix data order is HWC
  - <channel> Input channel
  - <height> The height of input map
  - <width> The width of input map
  - <kernel_height> The height of kernel
  - <kernel_width> The width of kernnel
  - <stride_x> Stride of X direction
  - <stride_y> Stride of Y direction
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
3. <channel> * sizeof (float) must be dividable by 128.
4. When sliding window in certain direction (H or W direction), if the left elements number doesn' t match the window size, these elemets will be discarded.
5. The result is a three-dimensional matrix with HWC data order.

- Example:

```
#include "mlu.h"
#define CHANNELS 64
#define IN_HEIGHT 9
#define IN_WIDTH 9
#define KERNEL_HEIGHT 3
#define KERNEL_WIDTH 3
#define STRIDE_X 1
#define STRIDE_Y 1
#define INPUT_COUNT ((CHANNELS) * (IN_WIDTH) * (IN_HEIGHT))
#define OUTPUT_COUNT                                        \
  ((CHANNELS) * ((IN_HEIGHT - KERNEL_HEIGHT) / STRIDE_Y + 1) * \
   ((IN_WIDTH - KERNEL_WIDTH) / STRIDE_X + 1))
__mlu_entry__ void PoolSumKernel(half* output, half* input, int channels,
                                 int in_height, int in_width, int kernel_height,
                                 int kernel_width, int stride_x, int stride_y) {
  __nram__ half a_tmp[INPUT_COUNT];
  __nram__ half b_tmp[OUTPUT_COUNT];
  __memcpy(a_tmp, input, INPUT_COUNT * sizeof(half), GDRAM2NRAM);
  __bang_sumpool(b_tmp, a_tmp, channels, in_height, in_width, kernel_height,
             kernel_width, stride_x, stride_y);
  __memcpy(output, b_tmp, OUTPUT_COUNT * sizeof(half), NRAM2GDRAM);
}
```

7.4.52 __bang_transpose

void __bang_transpose (half* dst, half* src, int height, int width);

void __bang_transpose (short* dst, short* src, int height, int width);

void __bang_transpose (unsigned short* dst, unsigned short* src, int height, int width);

void __bang_transpose (int8* dst, int8* src, int height, int width);

void __bang_transpose (char* dst, char* src, int height, int width);

void __bang_transpose (unsigned char* dst, unsigned char* src, int height, int width);

void __bang_transpose (float* dst, float* src, int height, int width);

void __bang_transpose (int* dst, int* src, int height, int width);

void __bang_transpose (unsigned int* dst, unsigned int* src, int height, int width);

Transpose operand src[height][width], a matrix, to dst[width][height].

- Parameters:
  - <dst> The address of destination matrix, and the matrix has WH data order
  - <src> The address of source matrix, and the matrix has HW data order
  - <height> The height of <src>
  - <width> The width of <src>
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. **<height>*sizeof** (type) and <width>*sizeof (type) must be dividable by 128 on MLU270,
   <height> and <width> must be dividable by 16 on MLU100;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
#include "mlu.h"
#define LEN 8192
#define WIDTH 128
#define HEIGHT 64
__mlu_entry__ void kernel(short* dst, short* src, int height, int width) {
  __nram__ short ny[LEN + 64];
  __nram__ short nx[LEN];

  __memcpy(nx, src, LEN * sizeof(short), GDRAM2NRAM);

  __bang_transpose(ny + 64, nx, HEIGHT, WIDTH);

  __memcpy(dst, ny + 64, sizeof(short) * LEN, NRAM2GDRAM);
}
```

7.4.53 __bang_unpool

Below graph give an example of unpool operation, where height = 5, width = 5, kernel_height = 2, kernel_width = 2, stride_height = 3, stride_width = 3, input_height = 2, input_width = 2, index = 1.

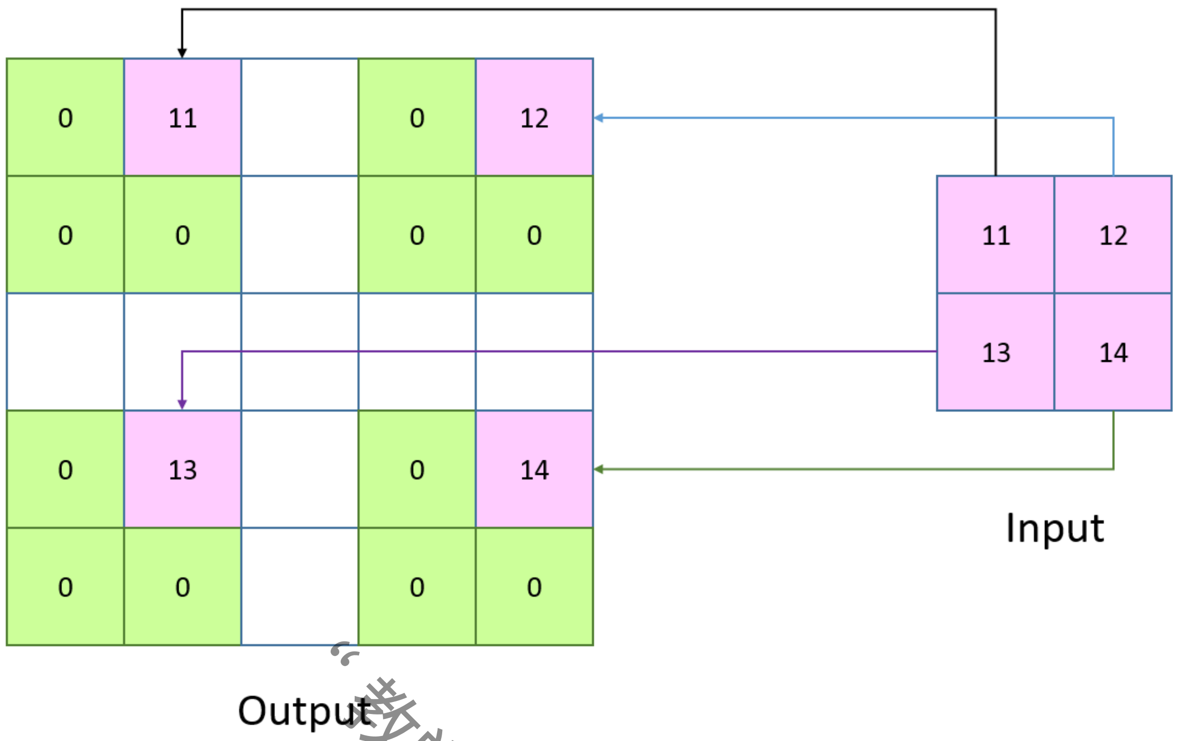

Fig. 7.3: Process of Unpool Operation

void __bang_unpool (half* dst, half* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y, int index);

void __bang_unpool (float* dst, float* src, int channel, int height, int width, int kernel_height, int kernel_width, int stride_x, int stride_y, int index);

Apply unpooling operation on operand <src>, a matrix. Every element of the <src> matrix operand corresponds to a kernel window on <dst> matrix operand. The element of <src> matrix is written into <index>’ th position in that kernel window on <dst> matrix, and other positions in the kernel window are written zero.

- Parameters:
  - <dst> The address of destination matrix
  - <src> The address of source matrix
  - <channel> Input channel
  - <height> The height of output map
  - <width> The width of output map
  - <kernel_height> The height of kernel
  - <kernel_width> The width of kernnel
  - <stride_x> X direction of stride
  - <stride_y> Y direction of stride
  - <index> Index of position in kernel to write element into
- Remarks

1. The operand <src> and <dst> must point to __nram__ space;
2. <channel> must be dividable by 64;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1,

2, ⋯);

4. The shape of <dst> matrix [out_channel, <height>, <width>] and <src> matrix [<channel>, input_height, input_width] has following restrictions:

out_channel = <channel>, <height> = (input_height – 1) * <stride_y> + <kernel_height>, <width> = (input_width – 1) * <stride_x> + <kernel_width>;

5. The <src> and <dst> matrix operands should not point to the same address;
6. The <index>' th position in kernel window means ikh' th row and ikw' th column in kernel window, where (ikh * <kernel_width> + ikw = <index>);
7. If the condition (<stride_x> >= kernel_x && <stride_y> >= kernel_y) is not satisfied, the output in <dst> matrix is undefined. If (<stride_x> > kernel_x || <stride_y> > kernel_y), there are some elements in <dst> vector that do not belong to any kernel window. These elements are not touched during computation.

- Example:

```
#include "mlu.h"
#define CHANNELS 128
#define HEIGHT 6
#define WIDTH 4
#define KERNEL_HEIGHT 3
#define KERNEL_WIDTH 2
#define STRIDE_X 2
#define STRIDE_Y 3
#define INDEX_IN_KERNEL 2
#define H_small ((HEIGHT-KERNEL_HEIGHT)/STRIDE_Y+1)
#define W_small ((WIDTH-KERNEL_WIDTH)/STRIDE_X+1)
#define TOP_DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))
#define BOTTOM_DATA_COUNT \
  ((CHANNELS) * H_small * W_small)

__mlu_entry__ void UnPoolKernel(half* top_data, int16_t* bottom_data,
                                int channels, int height, int width,
                                int kh, int kw,
                                int sx, int sy,
                                int index_in_kernel) {
  __nram__ half a_tmp[BOTTOM_DATA_COUNT];
  __nram__ half b_tmp[TOP_DATA_COUNT];
  __memcpy(a_tmp, bottom_data, BOTTOM_DATA_COUNT * sizeof(half), GDRAM2NRAM);

  // load original top data to nram, make sure if stride > kernel,
  // data outside of kernels is untouched.
  __memcpy(b_tmp, top_data, TOP_DATA_COUNT * sizeof(half), GDRAM2NRAM);

  __bang_unpool(b_tmp, a_tmp, CHANNELS, HEIGHT, WIDTH,
                               KERNEL_HEIGHT, KERNEL_WIDTH,
                               sx, sy,
                               index_in_kernel);

  __memcpy(top_data, b_tmp, TOP_DATA_COUNT * sizeof(half), NRAM2GDRAM);
}
```

7.4.54 __bang_write_zero

void __bang_write_zero (half* dst, int elem_count);

void __bang_write_zero (float* dst, int elem_count);

Write zeros in the memory space.

- Parameters:
  <dst> The address of destination vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <dst> must point to __nram__ space;
3. **Offset of the position, which operand <dst> points to, must be n * 64 bytes** (n = 0, 1, 2, ⋯).

7.4.55 __bang_rand

void __bang_rand (short* dst, int elem_count)

Generate a vector of uniformly distributed random number of short type.

- Parameters:
  <dst> The address of destination vector
  <elem_count> The elements number of operand vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <dst> must point to __nram__ space;
3. Offset of the position, which the operand <dst> points to, must be | n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
__mlu_entry__ void kernel
(int16_t *a, int size){
__nram__ short a_tmp[DATA_SIZE];
__bang_rand
(a_tmp, DATA_SIZE);
__memcpy
(a, a_tmp, size * sizeof
(int16_t), NRAM2GDRAM);
}
```

7.4.56 __bang_histogram

void __bang_histogram (int* dst, int8* src, int8* kernel, int size)

void __bang_histogram (int* dst, int16* src, int16* kernel, int size)

Generate a histogram of src according to kernel.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <kernel> The address of kernel vector
  - <elem_count> The elements number of operand vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> must point to __nram__ space;
3. The operand <kernel> must point to __wram__ space;
4. This interface is supported on MLU220, not supported on MLU100 and MLU270.

- Example

When src is 0 1 2 3 0 1 2 3 3, kernel’ s first line is 0, it selects src’ s “0” ,and generates 2; kernel’ s second line is 1, it selects src’ s “1” ,and generates 2; kernel’ s third line is 2, it selects src’ s “2” ,and generates 2; kernel’ s fourth line is 3, it selects src’ s “3” ,and generates 3; kernel’ s the other line is 0, and generates 2.

So dst is 2 2 2 3 2 2 ⋯2, dst number count is 64.

```
#include "mlu.h"
#define SRC_NUM 512
#define DST_NUM 64
#define LT_NUM 64
#define LT_SIZE 32
#define WEI_NUM ((LT_NUM) * (LT_SIZE))

__mlu_entry__ void kernel(int32_t* dst, int16_t* src, int16_t* filter, int size) {
  __nram__ int32_t dst_tmp[DST_NUM];
  __nram__ int16_t src_tmp[SRC_NUM];
  __wram__ int16_t filter_tmp[WEI_NUM];
  __memcpy(src_tmp, src, SRC_NUM * sizeof(int16_t), GDRAM2NRAM);
  __memcpy(filter_tmp, filter, WEI_NUM * sizeof(int16_t), GDRAM2WRAM);
  __bang_write_zero((half*)dst_tmp, DST_NUM * 2);
  __bang_histogram(dst_tmp, src_tmp, filter_tmp, size);
  __memcpy(dst, dst_tmp, DST_NUM * sizeof(int32_t), NRAM2GDRAM);
}
```

7.4.57 __bang_reshape_filter

void __bang_reshape_filter (half* dst, half* src, int n, int h, int w, int c)

void __bang_reshape_filter (short* dst, short* src, int n, int h, int w, int c)

void __bang_reshape_filter (unsigned short* dst, unsigned short* src, int n, int h, int w, int c)

void __bang_reshape_filter (int8* dst, int8* src, int n, int h, int w, int c)

void __bang_reshape_filter (char* dst, char* src, int n, int h, int w, int c)

void __bang_reshape_filter (unsigned char* dst, unsigned char* src, int n, int h, int w, int c)

void __bang_reshape_filter (float* dst, float* src, int n, int h, int w, int c)

void __bang_reshape_filter (int* dst, int* src, int n, int h, int w, int c)

void __bang_reshape_filter (unsigned int* dst, unsigned int* src, int n, int h, int w, int c)

Reshape the input kernel suit for MLP and CONV, n changed to n/64 along n direction, since there are 64 LTs.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of source vector
  - <n> Batch number of kernel
  - <h> Height of kernel
  - <w> Width of kernel
  - <c> Channel input of kernel
- Remarks

1. The interface is supported on MLU270;
2. The operand <src> must point to __nram__ space;
3. The operand <dst> must point to __nram__ space;
4. <n> must be dividable by 64;
5. <h> * <w> * <c> * sizeof(src) must be dividable by 128.

- Example:

```
#include "mlu.h"
#define IN_CHANNEL 64
#define IN_HEIGHT 5
#define IN_WIDTH 5
#define FILTER_HEIGHT 3
#define FILTER_WIDTH 3
#define STRIDE_HEIGHT 1
#define STRIDE_WIDTH 1
#define OUT_CHANNEL 64
#define OUT_HEIGHT ((((IN_HEIGHT) - (FILTER_HEIGHT)) / (STRIDE_HEIGHT)) + 1)
#define OUT_WIDTH ((((IN_WIDTH) - (FILTER_WIDTH)) / (STRIDE_WIDTH)) + 1)
#define OUT_DATA_NUM ((OUT_HEIGHT) * (OUT_WIDTH) * (OUT_CHANNEL))
#define IN_DATA_NUM ((IN_HEIGHT) * (IN_WIDTH) * (IN_CHANNEL))
#define FILTER_DATA_NUM \
  ((FILTER_HEIGHT) * (FILTER_WIDTH) * (IN_CHANNEL) * (OUT_CHANNEL))
__mlu_entry__ void ConvKernel(half* out_data, half* in_data, half* filter_data,
                              int in_channel, int in_height, int in_width,
                              int filter_height, int filter_width,
                              int stride_height, int stride_width,
                              int out_channel) {
```

```
  __nram__ half nram_out_data[OUT_DATA_NUM];
  __nram__ half nram_in_data[IN_DATA_NUM];
  __nram__ half nram_filter_data[FILTER_DATA_NUM];
  __nram__ half nram_filter_data_back[FILTER_DATA_NUM];
  __wram__ half wram_filter[FILTER_DATA_NUM];

  __memcpy(nram_in_data, in_data, IN_DATA_NUM * sizeof(half), GDRAM2NRAM);
  __memcpy(nram_filter_data, filter_data, FILTER_DATA_NUM * sizeof(half), GDRAM2NRAM);
  __bang_half2fix16_dn(&nram_in_data, nram_in_data, IN_DATA_NUM, -14);
  __bang_half2fix16_dn(&nram_filter_data, nram_filter_data, FILTER_DATA_NUM, -14);

  __bang_reshape_filter(nram_filter_data_back, nram_filter_data,
                        OUT_CHANNEL, FILTER_HEIGHT, FILTER_WIDTH,
                        IN_CHANNEL);
  __memcpy(wram_filter, nram_filter_data_back, FILTER_DATA_NUM * sizeof(half),
          NRAM2WRAM);

  __bang_conv(nram_out_data, (int16_t*)nram_in_data, (int16_t*)wram_filter, IN_CHANNEL, IN_
↪HEIGHT,
         IN_WIDTH, FILTER_HEIGHT, FILTER_WIDTH, STRIDE_WIDTH, STRIDE_HEIGHT,
         OUT_CHANNEL, -28);

  __memcpy(out_data, nram_out_data, OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

7.4.58 __bang_reshape_nhwc2nchw

void __bang_reshape_nhwc2nchw (half* dst, half* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (short* dst, short* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (unsigned short* dst, unsigned short* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (int8* dst, int8* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (char* dst, char* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (unsigned char* dst, unsigned char* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (float* dst, float* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (int* dst, int* src, int n, int h, int w, int c)

void __bang_reshape_nhwc2nchw (unsigned int* dst, unsigned int* src, int n, int h, int w, int c)

Reshape the kernel from NHWC to NCHW.

- Parameters:
  <dst> The address of destination vector
  <src> The address of source vector
  <n> Batch number of kernel

<h> Height of kernel
<w> Width of kernel
<c> Channel input of kernel

- Remarks

1. The interface is supported on MLU270;
2. The operand <src> must point to __nram__ space;
3. The operand <dst> must point to __nram__ space;
4. <h> * <w> * sizeof(src) must be dividable by 64;
5. <c> * sizeof(src) must be dividable by 64.

- Example:

```
#include "mlu.h"
#define N 3
#define H 4
#define W 32
#define C 128
#define DATA_NUM (N * H * W * C)
__mlu_entry__ void kernel(uint32_t n, int h, int w, int c, float* dst, float* src) {
  __nram__ float a_tmp[DATA_NUM];
  __nram__ float c_tmp[DATA_NUM];
  __memcpy(a_tmp, src, n * h * w * c * sizeof(float), GDRAM2NRAM);
  __bang_reshape_nhwc2nchw(c_tmp, a_tmp, N, H, W, C);
  __memcpy(dst, c_tmp, n * h * w * c * sizeof(float), NRAM2GDRAM);
}
```

7.4.59 __bang_reshape_nchw2nhwc

void __bang_reshape_nchw2nhwc (half* dst, half* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (short* dst, short* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (unsigned short* dst, unsigned short* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (int8* dst, int8* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (char* dst, char* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (unsigned char* dst, unsigned char* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (float* dst, float* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (int* dst, int* src, int n, int h, int w, int c)

void __bang_reshape_nchw2nhwc (unsigned int* dst, unsigned int* src, int n, int h, int w, int c)

Reshape the kernel from NCHW to NHWC.

- Parameters:

<dst> The address of destination vector
<src> The address of source vector

<n> Batch number of kernel
<h> Height of kernel
<w> Width of kernel
<c> Channel input of kernel

- Remarks

1. The interface is supported on MLU270;
2. The operand <src> must point to __nram__ space;
3. The operand <dst> must point to __nram__ space;
4. <h> * <w> * sizeof(src) must be dividable by 64;
5. <c> * sizeof(src) must be dividable by 64.

7.5 Stream Comparison Function

The half version of stream comparison function is supported on MLU100 and MLU270, the float version is supported on MLU270, and not supported on MLU100.

7.5.1 __bang_cycle_eq

void __bang_cycle_eq (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_eq (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector , into <src_elem_count> / <seg_elem_count> parts, then judge whether each element in each part and the corresponding element in <seg> are EQUAL, the result is assigned to <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <seg> The address of second operand vector
  <src_elem_count> The elements number of <src> vector
  <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
5. The address of operand <dst> and <src> can not be same.

- Example:

```
#include "mlu.h"
#define N1 256
#define N2 65280
#define s1 N1
```

```
#define s2 N2
__mlu_entry__ void kernel(uint32_t size1, uint32_t size2, half* c, half* a,
                          half* b) {
  __nram__ half a_tmp[s1];
  __nram__ half b_tmp[s2];
  __nram__ half c_tmp[s2];
  __memcpy(a_tmp, a, s1 * sizeof(half), GDRAM2NRAM);
  __memcpy(b_tmp, b, s2 * sizeof(half), GDRAM2NRAM);
  __bang_cycle_eq(c_tmp, b_tmp, a_tmp, size2, size1);
  __memcpy(c, c_tmp, s2 * sizeof(half), NRAM2GDRAM);
}
```

7.5.2 __bang_cycle_ge

void __bang_cycle_ge (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_ge (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then judge whether each element in each part is GREATER than or EQUAL to the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).
5. The address of operand <dst> and <src> can not be same.

7.5.3 __bang_cycle_gt

void __bang_cycle_gt (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_gt (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then judge whether each element in each part is GREATER than the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:

<dst> The address of destination vector
<src> The address of first operand vector
<seg> The address of second operand vector
<src_elem_count> The elements number of <src> vector
<seg_elem_count> The elements number of <seg> vector

- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
5. The address of operand <dst> and <src> can not be same.

7.5.4 __bang_cycle_le

void __bang_cycle_le (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_le (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then judge whether each element in each part is LESS than or EQUAL to the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:

<dst> The address of destination vector
<src> The address of first operand vector
<seg> The address of second operand vector
<src_elem_count> The elements number of <src> vector
<seg_elem_count> The elements number of <seg> vector

- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
5. The address of operand <dst> and <src> can not be same.

7.5.5 __bang_cycle_lt

void __bang_cycle_lt (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_lt (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then judge whether each element in each part is LESS than the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …).
5. The address of operand <dst> and <src> can not be same.

7.5.6 ___bang_cycle_ne

void __bang_cycle_ne (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_ne (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then judge whether each element in each part and the corresponding element in <seg> are NOT EQUAL, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space;
4. Offset of the position, which operand <src>, <seg>, and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …).
5. The address of operand <dst> and <src> can not be same.

7.5.7 ___bang_eq

void __bang_eq (half* dst, half* src0, half* src1, int elem_count);

void __bang_eq (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in two vectors are equal. The result element is 1, if the corresponding elements in <src0> and <src1> are equal; otherwise, the result element is

0.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).

- Example:

```
#include "mlu.h"
#define DATA_SIZE 128
__mlu_entry__ void kernel(uint32_t size, half* c, half* a, half* b) {
  __nram__ half a_tmp[DATA_SIZE + DATA_SIZE];
  __nram__ half c_tmp[DATA_SIZE];
  __nram__ half b_tmp[DATA_SIZE];
  __memcpy(a_tmp + DATA_SIZE, a, size * sizeof(half), GDRAM2NRAM);
  __memcpy(b_tmp, b, size * sizeof(half), GDRAM2NRAM);
  __bang_eq(c_tmp, a_tmp + DATA_SIZE, b_tmp, size);
  __memcpy(c, c_tmp, size * sizeof(half), NRAM2GDRAM);
}
```

7.5.8 __bang_eq_bitindex

void __bang_eq_bitindex (half* dst, half* src0, half* src1, int elem_count);

void __bang_eq_bitindex (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in two vectors are equal. The result element bit is 1, if the corresponding elements in <src0> and <src1> are equal; otherwise, the result element bit is 0.

The difference between __bang_eq and __bang_eq_bitindex is the result of __bang_eq_bitindex is bit which bit count is elem_count, the result of __bang_eq is half or float elem which elem count is elem_count.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 512;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;

3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. This interface is not supported on MLU100, and is supported on MLU270.

7.5.9 __bang_ge

void __bang_ge (half* dst, half* src0, half* src1, int elem_count);

void __bang_ge (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are larger than (including equal) the corresponding one in the other vector. The result element is 1, if the corresponding elements in <src0> are larger than corresponding elements in <src1>; otherwise, the result element is 0.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.5.10 __bang_ge_bitindex

void __bang_ge_bitindex (half* dst, half* src0, half* src1, int elem_count);

void __bang_ge_bitindex (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are larger than (including equal) the corresponding one in the other vector. The result element is 1, if the corresponding elements in <src0> are larger than corresponding elements in <src1>; otherwise, the result element is 0.

The difference between __bang_ge and __bang_ge_bitindex is the result of __bang_ge_bitindex is bit which bit count is elem_count, the result of __bang_ge is half or float elem which elem count is elem_count.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 512;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);

4. This interface is not supported on MLU100, and is supported on MLU270.

7.5.11 __bang_gt

void __bang_gt (half* dst, half* src0, half* src1, int elem_count);

void __bang_gt (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are larger than (not including equal) the corresponding one in the other vector. The result elements is 1, if the corresponding elements in <src0> are larger than corresponding elements in <src1>, otherwise, the result element is 0.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.5.12 __bang_gt_bitindex

void __bang_gt_bitindex (half* dst, half* src0, half* src1, int elem_count);

void __bang_gt_bitindex (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are larger than (not including equal) the corresponding one in the other vector. The result element is 1, if the corresponding elements in <src0> are larger than corresponding elements in <src1>; otherwise, the result element is 0.

The difference between __bang_gt and __bang_gt_bitindex is the result of __bang_gt_bitindex is bit which bit count is elem_count, the result of __bang_gt is half or float elem which elem count is elem_count.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 512;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);

4. This interface is not supported on MLU100, and is supported on MLU270.

7.5.13 __bang_le

void __bang_le (half* dst, half* src0, half* src1, int elem_count);

void __bang_le (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are smaller than (including equal) the corresponding one in the other vector. The result elements is 1, if the corresponding elements in <src0> are smaller than corresponding elements in <src1>; otherwise, the result element is 0.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …).

7.5.14 __bang_le_bitindex

void __bang_le_bitindex (half* dst, half* src0, half* src1, int elem_count);

void __bang_le_bitindex (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are smaller than (including equal) the corresponding one in the other vector. The result element is 1, if the corresponding elements in <src0> are smaller than corresponding elements in <src1>; otherwise, the result element is 0.

The difference between __bang_le and __bang_le_bitindex is the result of __bang_le_bitindex is bit which bit count is elem_count, the result of __bang_le is half or float elem which elem count is elem_count.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 512;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);

4. This interface is not supported on MLU100, and is supported on MLU270.

7.5.15 __bang_lt

void __bang_lt (half* dst, half* src0, half* src1, int elem_count);

void __bang_lt (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are smaller than (not including equal) the corresponding one in the other vector. The result element is 1, if the corresponding elements in <src0> are smaller than corresponding elements in <src1>; otherwise, the result element is 0.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.5.16 __bang_lt_bitindex

void __bang_lt_bitindex (half* dst, half* src0, half* src1, int elem_count);

void __bang_lt_bitindex (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in one vector are smaller than (not including equal) the corresponding one in the other vector. The result element is 1, if the corresponding elements in <src0> are smaller than corresponding elements in <src1>; otherwise, the result element is 0.

The difference between __bang_lt and __bang_lt_bitindex is the result of __bang_lt_bitindex is bit which bit count is elem_count, the result of __bang_lt is half or float elem which elem count is elem_count.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 512;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);

4. This interface is not supported on MLU100, and is supported on MLU270.

7.5.17 __bang_ne

void __bang_ne (half* dst, half* src0, half* src1, int elem_count);

void __bang_ne (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in two vectors are not equal. The result elements is 1 if the corresponding elements in <src0> and <src1> are not equal; otherwise, the result element is 0.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.5.18 __bang_ne_bitindex

void __bang_ne_bitindex (half* dst, half* src0, half* src1, int elem_count);

void __bang_ne_bitindex (float* dst, float* src0, float* src1, int elem_count);

Element-wise operator. Judge whether the numbers in two vectors are not equal. The result elements is 1 if the corresponding elements in <src0> and <src1> are not equal; otherwise, the result element is 0.

The difference between __bang_ne and __bang_ne_bitindex is the result of __bang_ne_bitindex is bit which bit count is elem_count, the result of __bang_ne is half or float elem which elem count is elem_count.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 512;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. This interface is not supported on MLU100, and is supported on MLU270.

7.6 Stream Logic and Bit Operation Function

The half version of stream logic and bit operation function is supported on MLU100 and MLU270, the float version is supported on MLU270, and not supported on MLU100.

7.6.1 __bang_and

void __bang_and (half* dst, half* src0, half* src1, int elem_count);

void __bang_and (float* dst, float* src0, float* src1, int elem_count);

Apply element-wise AND operation on two vectors.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
#include "mlu.h"
#define DATA_SIZE 128
__mlu_entry__ void kernel(uint32_t size, half* c, half* a, half* b) {
  __nram__ half a_tmp[DATA_SIZE + DATA_SIZE];
  __nram__ half c_tmp[DATA_SIZE];
  __nram__ half b_tmp[DATA_SIZE];
  __memcpy(a_tmp + DATA_SIZE, a, size * sizeof(half), GDRAM2NRAM);
  __memcpy(b_tmp, b, size * sizeof(half), GDRAM2NRAM);
  __bang_and(c_tmp, a_tmp + DATA_SIZE, b_tmp, size);
  __memcpy(c, c_tmp, size * sizeof(half), NRAM2GDRAM);
}
```

7.6.2 __bang_band

void __bang_band (char* dst, char* src0, char* src1, int elem_count);

Apply bit-wise AND operation on two vectors of char type.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector

- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
#include "mlu.h"
#define DATA_SIZE 128
__mlu_entry__ void kernel(uint32_t size, int* c, int* a, int* b) {
  __nram__ int a_tmp[2 * DATA_SIZE];
  __nram__ int c_tmp[DATA_SIZE];
  __nram__ int b_tmp[DATA_SIZE];
  __memcpy(a_tmp + DATA_SIZE, a, size * sizeof(int), GDRAM2NRAM);
  __memcpy(b_tmp, b, size * sizeof(int), GDRAM2NRAM);
  __bang_band((char*)c_tmp, (char*)(a_tmp + DATA_SIZE),
              (char*)b_tmp, size * 4);
  __memcpy(c, c_tmp, size * sizeof(int), NRAM2GDRAM);
}
```

7.6.3 __bang_bnot

void __bang_bnot (char* dst, char* src, int elem_count);

Apply bit-wise NOT operation on a vector of char type.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.4 __bang_bor

void __bang_bor (char* dst, char* src0, char* src1, int elem_count);

Apply bit-wise OR operation on two vectors of char type.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.5 __bang_bxor

void __bang_bxor (char* dst, char* src0, char* src1, int elem_count);

Apply bit-wise XOR operation on two vectors of char type.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.6 __bang_cycle_and

void __bang_cycle_and (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_and (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then each element in each part AND the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
#include "mlu.h"
#define N1 64
#define N2 16320
#define s1 N1
#define s2 N2
#define LEN 256
__mlu_entry__ void kernel(uint32_t size1, uint32_t size2, float* c, float* a,
                          float* b) {
  __nram__ float a_tmp[s1 + LEN];
  __nram__ float b_tmp[s2 + LEN];
  __nram__ float c_tmp[s2 + LEN];
  __memcpy(a_tmp + LEN, a, s1 * sizeof(float), GDRAM2NRAM);
  __memcpy(b_tmp + LEN, b, s2 * sizeof(float), GDRAM2NRAM);
  __bang_cycle_and(c_tmp + LEN, b_tmp + LEN, a_tmp + LEN, size2, size1);
  __memcpy(c, c_tmp + LEN, s2 * sizeof(float), NRAM2GDRAM);
}
```

7.6.7 __bang_cycle_band

void __bang_cycle_band (char* dst, char* src, char* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector of char type, into <src_elem_count> / <seg_elem_count> parts, then each element in each part apply bit-wise AND operation with the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (char) and <seg_elem_count> * sizeof (char) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.8 __bang_cycle_bor

void __bang_cycle_bor (char* dst, char* src, char* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector of char type, into <src_elem_count> / <seg_elem_count> parts, then each element in each part apply bit-wise OR operation with the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (char) and <seg_elem_count> * sizeof (char) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …).

7.6.9 __bang_cycle_bxor

void __bang_cycle_bxor (char* dst, char* src, char* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector of char type, into <src_elem_count> / <seg_elem_count> parts, then each element in each part apply bit-wise XOR operation with the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (char) and <seg_elem_count> * sizeof (char) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …).

7.6.10 __bang_cycle_maxequal

void __bang_cycle_maxequal (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_maxequal (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then each element in each part apply maxequal operation (select the maximum value) with the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of first operand vector
  - <seg> The address of second operand vector
  - <src_elem_count> The elements number of <src> vector
  - <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).

- Example:

```
#define s1 N1
#define s2 N2
#define LEN 256
__mlu_entry__ void kernel
 (uint32_t size1, uint32_t size2, half* c, half* a,half* b) {
__nram__ half a_tmp[s1 + LEN];
__nram__ half b_tmp[s2 + LEN];
__nram__ half c_tmp[s2 + LEN];
__memcpy
 (a_tmp + LEN, a, s1 * sizeof
 (half), GDRAM2NRAM);
__memcpy
 (b_tmp + LEN, b, s2 * sizeof
 (half), GDRAM2NRAM);
__bang_cycle_maxequal
 (c_tmp + LEN, b_tmp + LEN, a_tmp + LEN, size2, size1);
__memcpy
 (c, c_tmp + LEN, s2 * sizeof
 (half), NRAM2GDRAM);
}
```

7.6.11 __bang_cycle_minequal

void __bang_cycle_minequal (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_minequal (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then each element in each part apply minequal operation (select the minimum value) with the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:

<dst> The address of destination vector
<src> The address of first operand vector
<seg> The address of second operand vector
<src_elem_count> The elements number of <src> vector
<seg_elem_count> The elements number of <seg> vector

- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.12 __bang_cycle_or

void __bang_cycle_or (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_or (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then each element in each part OR the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <seg> The address of second operand vector
  <src_elem_count> The elements number of <src> vector
  <seg_elem_count> The elements number of <seg> vector
- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.13 __bang_cycle_xor

void __bang_cycle_xor (half* dst, half* src, half* seg, int src_elem_count, int seg_elem_count);

void __bang_cycle_xor (float* dst, float* src, float* seg, int src_elem_count, int seg_elem_count);

Divide <src>, a vector, into <src_elem_count> / <seg_elem_count> parts, then each element in each part XOR the corresponding element in <seg>, the result is assigned to <dst>.

- Parameters:
  <dst> The address of destination vector
  <src> The address of first operand vector
  <seg> The address of second operand vector

<src_elem_count> The elements number of <src> vector
<seg_elem_count> The elements number of <seg> vector

- Remarks

1. <src_elem_count> * sizeof (float) and <seg_elem_count> * sizeof (float) must be dividable by 128;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied;
3. The operand <src>, <seg> and <dst> must point to __nram__ space.
4. Offset of the position, which operand <src>, <seg> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.14 __bang_max

void __bang_max (half* dst, half* src, int elem_count);

void __bang_max (float* dst, float* src, int elem_count);

Find the maximum in a given vector. The result is composed of two parts. The first part is the max number, the second part is the index of the max number in <src> vector.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <dst> must point to __nram__ space;
3. The __nram__ space to which the vector operand <dst> points must be at least 128 bytes on MLU270, at least 32 bytes on MLU100;
4. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
5. The operand <dst> is composed of two elements: the first element is the max number, the second element is the index of the max number.
6. The index in second part of <dst> may be different with the result in BANG C-v1.x

- Example:

```
__bang_max
(dst, src, elem_count);

half maxNumber = dst[0];
unsigned int indexOfMaxNumber =
(
(unsigned int*)dst )[1];
```

7.6.15 __bang_maxequal

void __bang_maxequal (half* dst, half* src0, half* src1, int elem_count);

void __bang_maxequal (float* dst, float* src0, float* src1, int elem_count);

Find maximum value of each two corresponding elements in the two vector.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space.
3. Offset of the position, which operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).

7.6.16 ___bang_min

void __bang_min (half* dst, half* src, int elem_count);

void __bang_min (float* dst, float* src, int elem_count);

Find the minimum in a given vector. The result is composed of two parts. The first part is the min number, the second part is the index of the min number in <src> vector.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <dst> must point to __nram__ space;
3. The __nram__ space to which the vector operand <dst> points must be at least 128 bytes on MLU270, at least 32 bytes on MLU100;
4. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···).
5. The operand <dst> is composed of two elements: the first element is the min number, the second element is the index of the min number.
6. The index in second part of <dst> may be different with the result in BANG C-v1.x

- Example
  __bang_min (dst, src, elem_count);
  half minNumber = dst[0]; unsigned short indexOfMinNumber = ((unsigned short*)dst)[1];

7.6.17 ___bang_minequal

void __bang_minequal (half* dst, half* src0, half* src1, int elem_count);

void __bang_minequal (float* dst, float* src0, float* src1, int elem_count);

Find minimum value of each two corresponding elements in the two vector.

- Parameters:
  <dst> The address of destination vector
  <src0> The address of first operand vector
  <src1> The address of second operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space.
3. Offset of the position, which operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

- Example:

```
__mlu_entry__ void kernel
 (uint32_t size, half* c, half* a, half* b) {
 __nram__ half a_tmp[DATA_SIZE + DATA_SIZE];
 __nram__ half c_tmp[DATA_SIZE];
 __nram__ half b_tmp[DATA_SIZE];
 __memcpy
 (a_tmp + DATA_SIZE, a, size * sizeof
 (half), GDRAM2NRAM);
 __memcpy
 (b_tmp, b, size * sizeof
 (half), GDRAM2NRAM);
 __bang_minequal
 (c_tmp, a_tmp + DATA_SIZE, b_tmp, size);

 __memcpy
 (c, c_tmp, size * sizeof
 (half), NRAM2GDRAM);
}
```

7.6.18 __bang_not

void __bang_not (half* dst, half* src, int elem_count);

void __bang_not (float* dst, float* src, int elem_count);

Apply element-wise NOT operation on two vectors.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.19 __bang_or

void __bang_or (half* dst, half* src0, half* src1, int elem_count);

void __bang_or (float* dst, float* src0, float* src1, int elem_count);

Apply element-wise OR operation on two vectors.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.6.20 __bang_xor

void __bang_xor (half* dst, half* src0, half* src1, int elem_count);

void __bang_xor (float* dst, float* src0, float* src1, int elem_count);

Apply element-wise XOR operation on two vectors.

- Parameters:
  - <dst> The address of destination vector
  - <src0> The address of first operand vector
  - <src1> The address of second operand vector
  - <elem_count> The elements number of vector
- Remarks

1. <elem_count> must be dividable by 64;
2. The operand <src0>, <src1> and <dst> must point to __nram__ space;
3. Offset of the position, which the operand <src0>, <src1> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).

7.7 Stream Type Conversion Function

7.7.1 __bang_int82half

void __bang_int82half (half* dst, int8* src, int src_count, int fix_position);

void __bang_int82half (half* dst, int8* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from int8 to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1).

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 128 on MLU100 and MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-31,31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof (int8).
8. The interfaces are supported on both MLU100 and MLU270.

7.7.2 __bang_float2half_dn

void __bang_float2half_dn (half* dst, float* src, int src_count);

void __bang_float2half_dn (half* dst, float* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from float to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). ‘dn’ means the result is rounded down.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (half);
7. This interface is not supported on MLU100.

- Example:

```
#include "mlu.h"
#define DATA_SIZE 128
__mlu_entry__ void kernel(half* c, float* a, uint32_t size) {
  __nram__ float a_tmp[DATA_SIZE + DATA_SIZE];
  __memcpy(a_tmp + DATA_SIZE, a, size * sizeof(float), GDRAM2NRAM);
  __bang_float2half_dn((half*)(a_tmp + DATA_SIZE), a_tmp + DATA_SIZE, size);
  __memcpy(c, a_tmp + DATA_SIZE, size * sizeof(half), NRAM2GDRAM);
}
```

7.7.3 ___bang_float2half_oz

void __bang_float2half_oz (half* dst, float* src, int src_count);

void __bang_float2half_oz (half* dst, float* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from float to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'oz' means the result is rounded away from the 0 direction.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (half);
7. This interface is not supported on MLU100.

7.7.4 ___bang_float2half_rd

void __bang_float2half_rd (half* dst, float* src, int src_count);

void __bang_float2half_rd (half* dst, float* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from float to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'rd' means the result get from rounding.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<dst_stride> Destination address stride (bytes)
<src_stride> Source address stride (bytes)
<count> Section number, equals conversion times

- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (half);
7. This interface is not supported on MLU100.

7.7.5 ___bang__float2half__tz

void __bang_float2half_tz (half* dst, float* src, int src_count);

void __bang_float2half_tz (half* dst, float* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from float to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'tz' means the result is rounded to the 0 direction.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<dst_stride> Destination address stride (bytes)
<src_stride> Source address stride (bytes)
<count> Section number, equals conversion times

- Remarks

1. <src_count> must be dividable by 32 on MLU100, be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (half).
7. The interfaces are supported both on MLU100 and MLU270.

7.7.6 ___bang__float2half__up

void __bang_float2half_up (half* dst, float* src, int src_count);

void __bang_float2half_up (half* dst, float* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from float to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride(bytes)
  - <src_stride> Source address stride(bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (half);
7. This interface is not supported on MLU100.

7.7.7 __bang_float2int16_dn

void __bang_float2int16_dn (int16* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int16 in a given vector. The elements number in each conversion is <src_count>. 'dn' means the result is rounded down.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor(<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.8 __bang_float2int16_oz

void __bang_float2int16_oz (int16* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int16 in a given vector. The elements number in each conversion is <src_count>. 'oz' means the result is rounded away from the 0 direction.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.9 __bang_float2int16_rd

void __bang_float2int16_rd (int16* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int16 in a given vector. The elements number in each conversion is <src_count>. 'rd' means the result get from rounding.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.10 __bang_float2int16_tz

void __bang_float2shortz (int16* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int16 in a given vector. The elements number in each conversion is <src_count>. 'tz' means the result is rounded to the 0 direction.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;

5. This interface is not supported on MLU100.

7.7.11 __bang_float2int16_up

void __bang_float2int16_up (int16* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int16 in a given vector. The elements number in each conversion is <src_count>. 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.12 __bang_float2int8_dn

void __bang_float2int8_dn (int8* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int8 in a given vector. The elements number in each conversion is <src_count>. 'dn' means the result is rounded down.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.13 __bang_float2int8_oz

void __bang_float2int8_oz (int8* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int8 in a given vector. The elements number in each conversion is <src_count>. 'oz' means the result is rounded away from the 0 direction.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.14 __bang_float2int8_rd

void __bang_float2int8_rd (int8* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int8 in a given vector. The elements number in each conversion is <src_count>. 'rd' means the result get from rounding.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.15 __bang_float2int8_tz

void __bang_float2int8_tz (int8* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int8 in a given vector. The elements number in each conversion is <src_count>. 'tz' means the result is rounded to the 0 direction.

- Parameters:

<dst> The address of destination vector
<src> The address of operand vector
<src_count> The elements number in each conversion
<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;

5. This interface is not supported on MLU100.

7.7.16 __bang_float2int8_up

void __bang_float2int8_up (int8* dst, float* src, int src_count, int fix_position);

Convert element data type from float to int8 in a given vector. The elements number in each conversion is <src_count>. 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.17 __bang_half2char_dn

void __bang_half2char_dn (signed char* dst, half* src, int src_count);

Convert element data type from half to char in a given vector. The elements number in each conversion is <src_count>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
- Remarks

1. <src_count> must be dividable by 64 on MLU100, be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The interface is supported both on MLU100 and MLU270.

7.7.18 __bang_half2float

A description of a vector of half to float conversion with stride:

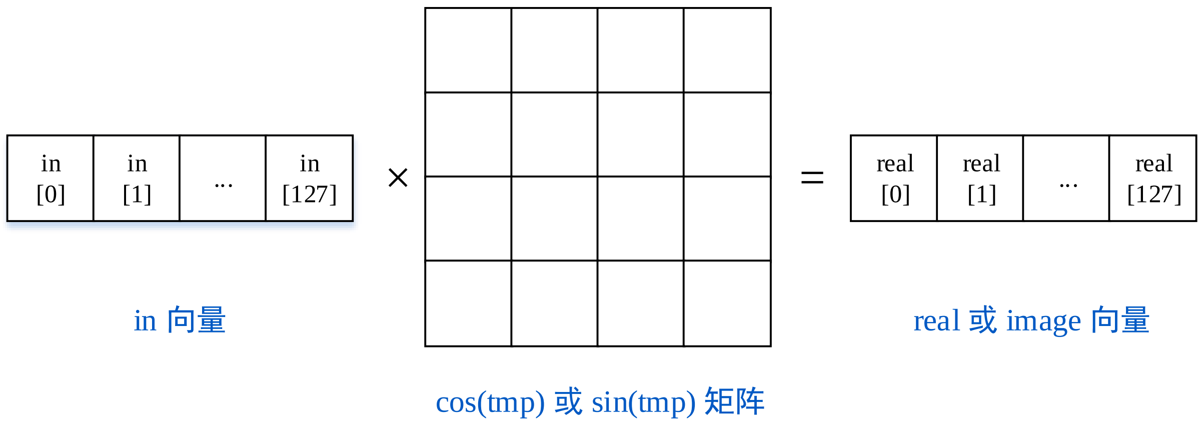

Fig. 7.4: Description of __bang_half2float Conversion

void __bang_half2float (float* dst, half* src, int src_count);

void __bang_half2float (float* dst, half* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from half to float in a given vector. The number of elements in each conversion is <src_count>, and the conversion times is (<count> + 1).

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The number of elements in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU100 and MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (float);
7. The interface is supported both on MLU100 and MLU270.

7.7.19 ___bang_half2int16_dn

void __bang_half2int16_dn (int16* dst, half* src, int src_count, int fix_position);

Convert element data type from half to int16 in a given vector. The elements number in each conversion is <src_count>. ‘dn’ means the result is rounded down.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion

<fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)

- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.20 __bang_half2int16_oz

void __bang_half2int16_oz (int16* dst, half* src, int src_count, int fix_position);

Convert element data type from half to int16 in a given vector. The elements number in each conversion is <src_count>. 'oz' means the result is rounded away from the 0 direction.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.21 __bang_half2int16_rd

void __bang_half2int16_rd (int16* dst, half* src, int src_count, int fix_position);

Convert element data type from half to int16 in a given vector. The elements number in each conversion is <src_count>. 'rd' means the result get from rounding.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.22 __bang_half2int16_tz

void __bang_half2int16_tz (int16* dst, half* src, int src_count, int fix_position);

Convert element data type from half to int16 in a given vector. The elements number in each conversion is <src_count>. 'tz' means the result is rounded to the 0 direction.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.23 __bang_half2int16_up

void __bang_half2int16_up (int16* dst, half* src, int src_count, int fix_position);

Convert element data type from half to int16 in a given vector. The elements number in each conversion is <src_count>. 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.24 __bang_half2int8_dn

void __bang_half2int8_dn (int8* dst, half* src, int src_count, int fix_position);

void __bang_half2int8_dn (int8* dst, half* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from half to int8 in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'dn' means the result is rounded down.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU100, be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-31,31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof (char).

7.7.25 __bang_half2int8_oz

void __bang_half2int8_oz (int8* dst, half* src, int src_count, int fix_position);

void __bang_half2int8_oz (int8* dst, half* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from half to int8 in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'oz' means the result is rounded away from the 0 direction.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-31,31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof (char);
8. This interface is not supported on MLU100.

7.7.26 __bang_half2int8_rd

void __bang_half2int8_rd (int8* dst, half* src, int src_count, int fix_position);

void __bang_half2int8_rd (int8* dst, half* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from half to int8 in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'rd' means the result get from rounding.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
  <dst_stride> Destination address stride (bytes)
  <src_stride> Source address stride (bytes)
  <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-31,31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof (char);
8. This interface is not supported on MLU100..

7.7.27 __bang_half2int8_tz

void __bang_half2int8_tz (int8* dst, half* src, int src_count, int fix_position);

void __bang_half2int8_tz (int8* dst, half* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from half to int8 in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'tz' means the result is rounded to the 0 direction.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
  <dst_stride> Destination address stride (bytes)
  <src_stride> Source address stride (bytes)
  <count> Section number, equals conversion times

- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-31,31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof (char)
8. This interface is not supported on MLU100.

7.7.28 __bang_half2int8_up

void __bang_half2int8_up (int8* dst, half* src, int src_count, int fix_position);

void __bang_half2int8_up (int8* dst, half* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from half to int8 in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). ‘up’ means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> / 2^<fix_position>)
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-31,31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof (char);
8. This interface is not supported on MLU100.

7.7.29 __bang_half2short_dn

void __bang_half2short_dn (short* dst, half* src, int src_count);

void __bang_half2short_dn (short* dst, half* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from half to short in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'dn' means the result is rounded down.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <dst_stride> Destination address stride (bytes)
  <src_stride> Source address stride (bytes)
  <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU100 and on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (short);
7. The interfaces are supported both on MLU100 and MLU270.

7.7.30 __bang_half2short_oz

void __bang_half2short_oz (short* dst, half* src, int src_count);

void __bang_half2short_oz (short* dst, half* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from half to short in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'up' means the result is rounded up.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <dst_stride> Destination address stride (bytes)
  <src_stride> Source address stride (bytes)
  <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (short);
7. This interface is not supported on MLU100.

7.7.31 __bang_half2short_rd

void __bang_half2short_rd (short* dst, half* src, int src_count);

void __bang_half2short_rd (short* dst, half* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from half to short in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU100 and on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (short);
7. This interface is not supported on MLU100.

7.7.32 __bang_half2short_tz

void __bang_half2short_tz (short* dst, half* src, int src_count);

void __bang_half2short_tz (short* dst, half* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from half to short in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;

3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (short);
7. This interface is not supported on MLU100.

7.7.33 __bang_half2short_up

void __bang_half2short_up (short* dst, half* src, int src_count);

void __bang_half2short_up (short* dst, half* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from half to short in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1). 'up' means the result is rounded up.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ···);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (short);
7. This interface is not supported on MLU100.

7.7.34 __bang_half2uchar_dn

void __bang_half2uchar_dn (unsigned char* dst, half* src, int src_count);

void __bang_half2uchar_dn (unsigned char* dst, half* src, half* src_addition, int src_count);

Convert element data type from half to unsigned char in a given vector. The elements number in each conversion is <src_count>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_addition> The address of operand vector in addition
  - <src_count> The elements number in each conversion

- Remarks

1. <src_count> must be dividable by 64 on MLU100, be dividable by 128 on MLU270;
2. The operand <src>, <src_addition>, <dst> must point to __nram__ space;
3. Offset of the position, which <src>, <src_addition> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. Src_addition point to addition nram space. The space length which src_addition point to must be equal to space length that src point to.
5. The difference between two version is : the version without src_addition src_addition can convert data belongs to [0, 127], the version with src_addition can convert data belongs to [0, 255].

- Example:

```
  __mlu_entry__ void Kernel(half *out_data, half* in_data, int size, int pos) {
    __nram__ half nram_in_data[DATA_NUM];
    __nram__ half nram_in_data_ad[DATA_NUM];
    __nram__ half nram_out_data[DATA_NUM];

   __memcpy(nram_in_data, in_data, DATA_NUM * sizeof(half), GDRAM2NRAM);

   __bang_half2uchar_dn((uint8_t*)nram_out_data, nram_in_data, nram_in_data_ad, size);
   __memcpy(out_data, nram_out_data, DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

7.7.35 __bang_int82half

void __bang_int82half (half* dst, int8* src, int src_count, int fix_position);

void __bang_int82half (half* dst, int8* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from int8 to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1).

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> * 2^<fix_position>)
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 128 on MLU100 and MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯).
4. Round type only supports rounding down.

5. <fix_position> must belong to [-31, 31] and be constant.
6. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
7. <dst_stride> and <src_stride> must be divided by 32;
8. <dst_stride> must not be less than <src_count> * sizeof (half);
9. The interfaces are supported both on MLU100 and MLU270.

7.7.36 __bang_int162float

void __bang_int162float (float* dst, int16* src, int src_count, int fix_position);

Convert element data type from int16 to float in a given vector. The elements number in each conversion is <src_count>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> * 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. This interface is not supported on MLU100.

7.7.37 __bang_int162half

void __bang_int162half (half* dst, int16* src, int src_count, int fix_position);

Convert element data type from int16 to half in a given vector. The elements number in each conversion is <src_count>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> * 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 64 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, …);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.38 __bang_int82float

void __bang_int82float (float* dst, int8* src, int src_count, int fix_position);

Convert element data type from int8 to float in a given vector. The elements number in each conversion is <src_count>.

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <fix_position> Scale factor (<dst> = <src> * 2^<fix_position>)
- Remarks

1. <src_count> must be dividable by 128 on MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-127,127] and be constant;
5. This interface is not supported on MLU100.

7.7.39 ___bang_short2half

void __bang_short2half (half* dst, short* src, int src_count);

void __bang_short2half (half* dst, short* src, int src_count, int dst_stride, int src_stride, int count);

Convert element data type from short to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is (<count> + 1).

- Parameters:
  - <dst> The address of destination vector
  - <src> The address of operand vector
  - <src_count> The elements number in each conversion
  - <dst_stride> Destination address stride (bytes)
  - <src_stride> Source address stride (bytes)
  - <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 64 on MLU100 and MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
5. <dst_stride> and <src_stride> must be divided by 32;
6. <dst_stride> must not be less than <src_count> * sizeof (half);
7. The interfaces are supported both on MLU100 and MLU270.

7.7.40 ___bang_uchar2half

void __bang_uchar2half (half* dst, unsigned char* src, int src_count);

void __bang_uchar2half (half* dst, unsigned char* src, int src_count, int fix_position, int dst_stride, int src_stride, int count);

Convert element data type from unsigned int8 to half in a given vector. The elements number in each conversion is <src_count>, and the conversion times is <count> + 1.

- Parameters:
  <dst> The address of destination vector
  <src> The address of operand vector
  <src_count> The elements number in each conversion
  <fix_position> Scale factor(<dst> = <src> * 2^<fix_position>)
  <dst_stride> Destination address stride (bytes)
  <src_stride> Source address stride (bytes)
  <count> Section number, equals conversion times
- Remarks

1. <src_count> must be dividable by 128 on MLU100 and MLU270;
2. The operand <src>, <dst> must point to __nram__ space;
3. Offset of the position, which operand <src> and <dst> point to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. <fix_position> must belong to [-31, 31] and be constant.
5. <dst_stride>, <src_stride>, <count> is not negative and must be constant;
6. <dst_stride> and <src_stride> must be divided by 32;
7. <dst_stride> must not be less than <src_count> * sizeof(half);
8. The interfaces are supported both on MLU100 and MLU270.

7.8 Stream Atomic Function

Stream atomic function is supported on MLU270, and not supported on MLU100.

7.8.1 __bang_atomic_add

unsigned short __bang_atomic_add (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_add (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_add (unsigned int* dst, unsigned int* src1, unsigned int* src2, unsigned int size);

int __bang_atomic_add (int* dst, int* src1, int* src2, int size);

half __bang_atomic_add (half* dst, half* src1, half* src2, int size);

float __bang_atomic_add (float* dst, float* src1, float* src2, int size);

Read vector pointer address src1 value, add vector value <src2> to value <src1>, and store the original value of <src1> in <dst>.

These three operations are performed in one atomic transaction.

That is: <dst> = <src1>; <src1> = <src1> + <src2>

- Parameters:

<dst> The address of destination operand vector
<src1> The address of first operand vector
<src2> The address of second operand vector
<size> The elements number of vector

- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

- Example:

```
#include "mlu.h"

__mlu_entry__ void kernel(int* worker, int* data, int size, int tid) {
  __nram__ int v[512];
  int cpysz = size * sizeof(int);
  if (cpysz % 16 != 0) {
    cpysz = (cpysz / 16 + 1) * 16;
  }
  __memcpy(v, data, cpysz, GDRAM2NRAM);
  __bang_atomic_add(v, worker, v, size);
}
```

7.8.2 __bang_atomic_and

unsigned short __bang_atomic_and (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_and (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_and (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_and (int* dst, int* src1, int* src2, int size);

half __bang_atomic_and (half* dst, half* src1, half* src2, int size);

float __bang_atomic_and (float* dst, float* src1, float* src2, int size);

Apply logical AND operation to the vector <src1> and <src2>, store the result in <src1>, and store the original value of <src1> in <dst>. That is: <dst> = <src1>; <src1> = <src1> & <src2>

- Parameters:
  <dst> The address of destination operand vector
  <src1> The address of first operand vector
  <src2> The address of second operand vector
  <size> The elements number of vector
- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.3 __bang_atomic_cas

unsigned short __bang_atomic_cas (unsigned short* dst, unsigned short* src1, unsigned short* src2, unsigned short* src3, int size);

short __bang_atomic_cas (short* dst, short* src1, short* src2, short* src3, int size);

unsigned int __bang_atomic_cas (unsigned int* dst, unsigned int* src1, unsigned int* src2, unsigned int* src3, int size);

int __bang_atomic_cas (int* dst, int* src1, int* src2, int* src3, int size);

half __bang_atomic_cas (half* dst, half* src1, half* src2, half* src3, int size);

float __bang_atomic_cas (float* dst, float* src1, float* src2, float* src3, int size);

Compare vector <src1> and <src2>. If <src2> is equal to <src1>, store the value <src3> in <src1>. Store the original value of <src1> in <dst>.

That is: <dst> = <src1>; <src1> = (<src1> == <src2>) ? <src3> : <src1>

- Parameters:
  - <dst> The address of destination operand vector
  - <src1> The address of first operand vector
  - <src2> The address of second operand vector
  - <src3> The address of third operand vector
  - <size> The elements number of vector
- Return Value:
  - The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.4 __bang_atomic_dec

unsigned short __bang_atomic_dec (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_dec (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_dec (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_dec (int* dst, int* src1, int* src2, int size);

half __bang_atomic_dec (half* dst, half* src1, half* src2, int size);

float __bang_atomic_dec (float* dst, float* src1, float* src2, int size);

Compare vector <src1> and <src2>. If <src1> is bigger than <src2>, or the value of <src1> is 0, store the int value <src2> in <src1>; otherwise, reduce <src1> by 1. Store the original value of <src1> in <dst>. That is: <dst> = <src1>; <src1> = (<src1> == 0 || <src1> > <src2>) ? <src2> : (<src1> - 1)

- Parameters:
  - <dst> The address of destination operand vector
  - <src1> The address of first operand vector
  - <src2> The address of second operand vector
  - <size> The elements number of vector
- Return Value:
  - The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.5 ___bang_atomic_exch

unsigned short __bang_atomic_exch (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_exch (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_exch (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_exch (int* dst, int* src1, int* src2, int size);

half __bang_atomic_exch (half* dst, half* src1, half* src2, int size);

float __bang_atomic_exch (float* dst, float* src1, float* src2, int size);

Store vector <src1> in <dst>. Store <src2> in <src1>. That is: <dst> = <src1>; <src1> = <src2>

- Parameters:
  - <dst> The address of destination operand vector
  - <src1> The address of first operand vector
  - <src2> The address of second operand vector
  - <size> The elements number of vector
- Return Value:
  - The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.6 ___bang_atomic_inc

unsigned short __bang_atomic_inc (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_inc (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_inc (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_inc (int* dst, int* src1, int* src2, int size);

half __bang_atomic_inc (half* dst, half* src1, half* src2, int size);

float __bang_atomic_inc (float* dst, float* src1, float* src2, int size);

Compare vector <src1> and <src2>. If <src1> is smaller than <src2>, increase <src1> by 1; otherwise, set <src1> to 0. Store the original value of <src1> in <dst>.

That is: <dst> = <src1>; <src1> = (<src1> >= <src2>) ? 0 : (<src1> + 1)

- Parameters:
  <dst> The address of destination operand vector
  <src1> The address of first operand vector
  <src2> The address of second operand vector
  <size> The elements number of vector
- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.7 __bang_atomic_max

unsigned short __bang_atomic_max (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_max (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_max (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_max (int* dst, int* src1, int* src2, int size);

half __bang_atomic_max (half* dst, half* src1, half* src2, int size);

float __bang_atomic_max (float* dst, float* src1, float* src2, int size);

Take the larger value from vector <src1> and <src2>, and store it in <src1>. Store the original value of <src1> in <dst>. That is: <dst> = <src1>; <src1> = (<src1> > <src2>) ? <src1> : <src2>

- Parameters:
  <dst> The address of destination operand vector
  <src1> The address of first operand vector
  <src2> The address of second operand vector
  <size> The elements number of vector
- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;

2. <src1> must point to ldram/gdram address space.

7.8.8 __bang_atomic_min

unsigned short __bang_atomic_min (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_min (short* dst, short* src1, short src2, int size);

unsigned int __bang_atomic_min (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_min (int* dst, int* src1, int* src2, int size);

half __bang_atomic_min (half* dst, half* src1, half* src2, int size);

float __bang_atomic_min (float* dst, float* src1, float* src2, int size);

Take the smaller value from two vector values <src1> and <src2>, and store it in <src1>. Store the original value of <src1> in <dst>.

That is: <dst> = <src1>; <src1> = (<src1> < <src2>) ? <src1> : <src2>

- Parameters:
  <dst> The address of destination operand vector
  <src1> The address of first operand vector
  <src2> The address of second operand vector
  <size> The elements number of vector
- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.9 __bang_atomic_or

unsigned short __bang_atomic_or (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_or (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_or (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_or (int* dst, int* src1, int* src2, int size);

half __bang_atomic_or (half* dst, half* src1, half* src2, int size);

float __bang_atomic_or (float* dst, float* src1, float* src2, int size);

Apply logical OR operation to vector <src1> and <src2>, store the result in <src1>, and store the original value of <src1> in <dst>. That is: <dst> = <src1>; <src1> = <src1> | <src2>

- Parameters:

<dst> The address of destination operand vector
<src1> The address of first operand vector
<src2> The address of second operand vector
<size> The elements number of vector

- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.8.10 __bang_atomic_xor

unsigned short __bang_atomic_xor (unsigned short* dst, unsigned short* src1, unsigned short* src2, int size);

short __bang_atomic_xor (short* dst, short* src1, short* src2, int size);

unsigned int __bang_atomic_xor (unsigned int* dst, unsigned int* src1, unsigned int* src2, int size);

int __bang_atomic_xor (int* dst, int* src1, int* src2, int size);

half __bang_atomic_xor (half* dst, half* src1, half* src2, int size);

float __bang_atomic_xor (float* dst, float* src1, float* src2, int size);

Apply logical XOR operation to vector <src1> and <src2>, store the result in <src1>, and store the original value of <src1> in <dst>. That is: <dst> = <src1>; <src1> = <src1> ^ <src2>

- Parameters:
  <dst> The address of destination operand vector
  <src1> The address of first operand vector
  <src2> The address of second operand vector
  <size> The elements number of vector
- Return Value:
  The address of destination operand vector
- Remarks:

1. <dst> and <src2> must point to nram address space;
2. <src1> must point to ldram/gdram address space.

7.9 Memory Setting Function

7.9.1 __bang_lock

void __bang_lock (int lock_id_0, int lock_id_1);

Apply and seize lock_id.

- Parameters:

<lock_id_0> The first lock id
<lock_id_1> The second lock id

- Remarks

1. For IPU_Core, lock_id_0=lock_id_1, apply and seize one lock_id;
2. For Mem_Core, lock_id_0!=lock_id_1, apply and seize two lock_ids;
3. __bang_lock is always paired with __bang_unlock;
4. __bang_lock is used before memcpy from to GDRAM, size larger than 64KB;
5. The interface is supported on MLU270, not on MLU100.

- Example

```
#include "mlu.h"
#define BUFFER_SIZE 261632
__mlu_entry__ void add_kernel(half*input,half*output,int copyin_len,int copyout_len,
                                int offseta,int offsetb,int offsetc,int tid){
    __wram__ half temp[BUFFER_SIZE];
    if (coreId != 0x80) {
      __bang_lock(0,0);
    }
    for(int i=0;i<50000;i++){
      __memcpy(temp,input,510*512*sizeof(half),GDRAM2WRAM);
    }
    if (coreId != 0x80) {
      __bang_unlock(0,0);
    }
}
```

7.9.2 __bang_unlock

void __bang_unlock (int lock_id_0, int lock_id_1);

Release lock_id.

- Parameters:
  <lock_id_0> The first lock id
  <lock_id_1> The second lock id
- Remarks

1. For IPU_Core, lock_id_0=lock_id_1, release one lock_id;
2. For Mem_Core, lock_id_0!=lock_id_1, release two lock_ids;
3. __bang_unlock is always paired with __bang_lock;
4. __bang_unlock is used after memcpy from to GDRAM, size larger than 64KB;
5. The interface is supported on MLU270, not on MLU100.

7.9.3 __gdramset

void __gdramset (char* dst, int elem_count, char value);

void __gdramset (signed char* dst, int elem_count, signed char value);

void __gdramset (unsigned char* dst, int elem_count, unsigned char value);

void __gdramset (half* dst, int elem_count, half value);

void __gdramset (short* dst, int elem_count, short value);

void __gdramset (unsigned short* dst, int elem_count, unsigned short value);

void __gdramset (float* dst, int elem_count, float value);

void __gdramset (int* dst, int elem_count, int value);

void __gdramset (unsigned int* dst, int elem_count, unsigned int value);

Set <elem_count> blocks of the __gdram__ space pointed by <dst> to the <value>.

- Parameters:
  <dst> The address of destination area
  <elem_count> The number of elements to be set
  <value> Value to be set
- Remarks

1. The operand <dst> must point to __gdram__ space;
2. <elem_count> must be constant, and must be dividable by 64;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The interface is supported on MLU270, not on MLU100.

- Example

```
#include "mlu.h"
#define LEN 1024
__mlu_entry__ void kernel(half* a, int offset, int elem_num, half val) {
  __gdramset(a, LEN / 4, val);
  __gdramset(a + LEN / 4, elem_num / 4, 2.0);
  __gdramset(a + LEN / 4 * 2, LEN / 4, val);
  __gdramset(a + LEN / 4 * 3, elem_num / 4, val);
}
```

7.9.4 __ldramset

void __ldramset (char* dst, int elem_count, char value);

void __ldramset (signed char* dst, int elem_count, signed char value);

void __ldramset (unsigned char* dst, int elem_count, unsigned char value);

void __ldramset (half* dst, int elem_count, half value);

void __ldramset (short* dst, int elem_count, short value);

void __ldramset (unsigned short* dst, int elem_count, unsigned short value);

void __ldramset (float* dst, int elem_count, float value);

void __ldramset (int* dst, int elem_count, int value);

void __ldramset (unsigned int* dst, int elem_count, unsigned int value);

Set <elem_count> blocks of the __ldram__ space pointed by <dst> to the specified <value>.

- Parameters:
  <dst> The address of destination area
  <elem_count> The number of elements to be set
  <value> Value to be set
- Remarks

1. The operand <dst> must point to __ldram__ space;
2. <elem_count> must be constant, and must be dividable by 64;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The interface is supported on MLU270, not on MLU100.

7.9.5 __memcpy

void __memcpy (void* dst, void* src, int size, mluMemcpyDirection_t dir);

void __memcpy (void* dst, void* src, int size, mluMemcpyDirection_t dir, int dst_stride, int src_stride, int count);

void __memcpy (void* dst, void* src, int size, mluMemcpyDirection_t dir, int dst_stride, int src_stride, int count, int id_dst_cluster);

Copy data from one address space <src> to another address space <dst>. The number of each copy equals <size>.

- Parameters:
  <count> Section number
  <dir> Copy direction. It must be one of: GDRAM2NRAM, NRAM2GDRAM, NRAM2LDRAM, LDRAM2NRAM, SRAM2NRAM, NRAM2SRAM, NRAM2NRAM
  <dir> Copy direction. It must be one of: GDRAM2NRAM, NRAM2GDRAM, NRAM2LDRAM, LDRAM2NRAM, WRAM2LDRAM, LDRAM2WRAM, WRAM2GDRAM, GDRAM2WRAM, NRAM2NRAM, GDRAM2GDRAM, GDRAM2LDRAM, LDRAM2GDRAM, SRAM2GDRAM, GDRAM2SRAM, SRAM2LDRAM，LDRAM2SRAM, NRAM2SRAM, SRAM2NRAM, SRAM2WRAM, WRAM2SRAM, SRAM2SRAM
  <dst_stride> Destination address stride
  <dst> The address of destination area
  <size> The number of bytes to be copied
  <src_stride> Source address stride
  <src> The address of source area
  <id_dst_cluster> Destination cluster id
- Remarks

1. <size> must be dividable by 32 when <dir> is NRAM2NRAM, GDRAM2NRAM, NRAM2GDRAM, NRAM2LDRAM, LDRAM2NRAM, GDRAM2GDRAM, GDRAM2LDRAM, or LDRAM2GDRAM on MLU100;
2. <size> must be dividable by 512 when <dir> is WRAM2SRAM, SRAM2WRAM, WRAM2LDRAM, LDRAM2WRAM, WRAM2GDRAM, or GDRAM2WRAM on MLU100 and MLU270;
3. <size> must be dividable by 1 when <dir> is SRAM2GDRAM, GDRAM2SRAM, SRAM2LDRAM，LDRAM2SRAM, NRAM2SRAM, SRAM2NRAM, SRAM2SRAM, GDRAM2NRAM, NRAM2GDRAM,

NRAM2LDRAM, LDRAM2NRAM, GDRAM2GDRAM, GDRAM2LDRAM, or LDRAM2GDRAM on MLU270;

4. <size> must be dividable by 128 when <dir> is NRAM2NRAM on MLU270;
5. Offset of the position, which operand <src> points to, must be n * 32 bytes(n = 0, 1, 2, ⋯) when <dir> is GDRAM2NRAM, LDRAM2NRAM, LDRAM2WRAM, GDRAM2WRAM, GDRAM2GDRAM, GDRAM2LDRAM, LDRAM2GDRAM on MLU100;
6. Offset of the position, which operand <dst> points to, must be n * 32 bytes(n = 0, 1, 2, ⋯) when <dir> is NRAM2GDRAM, NRAM2LDRAM, WRAM2LDRAM, WRAM2GDRAM, GDRAM2GDRAM, GDRAM2LDRAM, LDRAM2GDRAM on MLU100;
7. Offset of the position, which operand <src> points to, must be n * 32 bytes(n = 0, 1, 2, ⋯) when <dir> is GDRAM2NRAM, LDRAM2NRAM;
8. Offset of the position, which operand <dst> points to, must be n * 32 bytes(n = 0, 1, 2, ⋯) when <dir> is NRAM2GDRAM, NRAM2LDRAM;
9. Copy data with stride support between nram and gdram/ldram, between nram2nram, and between sram2sram;
10. <dst_stride>, <src_stride>, <count> is not negative and can be constant or register;
11. <dst_stride> and <src_stride> must be divided by 32 on MLU100
12. <dst_stride> and <src_stride> must be dividable by 1 on MLU270 when <dir> is SRAM2GDRAM, GDRAM2SRAM, SRAM2LDRAM， LDRAM2SRAM, be dividable by 4 on MLU270 when <dir> is GDRAM2NRAM, NRAM2GDRAM, NRAM2LDRAM, LDRAM2NRAM;
13. <dst_stride> must not be less than <size>.
14. The number of conversion is (<count> + 1).
15. The interface are supported on MLU270 and on MLU100. The interface which direction refer to SRAM are not supported on MLU100, and are supported on MLU270.
16. The interface version with id_dst_cluster is only supported on direction SRAM2SRAM.

7.9.6 __nramset

void __nramset (char* dst, int elem_count, char value);

void __nramset (signed char* dst, int elem_count, signed char value);

void __nramset (unsigned char* dst, int elem_count, unsigned char value);

void __nramset (half* dst, int elem_count, half value);

void __nramset (short* dst, int elem_count, short value);

void __nramset (unsigned short* dst, int elem_count, unsigned short value);

void __nramset (float* dst, int elem_count, float value);

void __nramset (int* dst, int elem_count, int value);

void __nramset (unsigned int* dst, int elem_count, unsigned int value);

Set <elem_count> blocks of the __nram__ space pointed by <dst> to <value>.

- Parameters:
  - <dst> The address of destination area
  - <elem_count> The number of elements to be set
  - <value> Value to be set

- Remarks

1. The operand <dst> must point to __nram__ space;
2. <elem_count> must be constant, and must be dividable by 64;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, …);

7.9.7 ___nramset_half

void __nramset_half (void* dst, int elem_count, half value);

Set <elem_count> blocks of the __nram__ space pointed by <dst> to the specified <value>.

- Parameters:
  - <dst> The address of destination area
  - <elem_count> The number of elements to be set
  - <value> Value to be set
- Remarks

1. The operand <dst> must point to __nram__ space;
2. <elem_count> must be constant, and must be dividable by 64;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The interface is supported on MLU270 and on MLU100.

7.9.8 ___nramset_int

void __nramset_int (void* dst, int elem_count, int value);

Set <elem_count> blocks of the __nram__ space pointed by <dst> to the specified <value>.

- Parameters:
  - <dst> The address of destination area
  - <elem_count> The number of elements to be set
  - <value> Value to be set
- Remarks

1. The operand <dst> must point to __nram__ space;
2. <elem_count> must be constant, and must be dividable by 32;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, …);
4. The interface is supported on MLU270 and on MLU100.

7.9.9 ___nramset_short

void __nramset_short (void* dst, int elem_count, short value);

Set <elem_count> blocks of the __nram__ space pointed by <dst> to the specified <value>.

- Parameters:
  - <dst> The address of destination area
  - <elem_count> The number of elements to be set
  - <value> value to be set
- Remarks

1. The operand <dst> must point to __nram__ space;
2. <elem_count> must be constant, and must be dividable by 64;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. The interface is supported on MLU270 and on MLU100.

7.9.10 ___sramset

void __sramset (char* dst, int elem_count, char value);

void __sramset (signed char* dst, int elem_count, signed char value);

void __sramset (unsigned char* dst, int elem_count, unsigned char value);

void __sramset (half* dst, int elem_count, half value);

void __sramset (short* dst, int elem_count, short value);

void __sramset (unsigned short* dst, int elem_count, unsigned short value);

void __sramset (float* dst, int elem_count, float value);

void __sramset (int* dst, int elem_count, int value);

void __sramset (unsigned int* dst, int elem_count, unsigned int value);

Set <elem_count> blocks of the __sram__ space pointed by <dst> to <value>.

- Parameters:
  - <dst> The address of destination area
  - <elem_count> The number of elements to be set
  - <value> Value to be set
- Remarks

1. The operand <dst> must point to __sram__ space;
2. <elem_count> must be constant, and must be dividable by 64;
3. Offset of the position, which operand <dst> points to, must be n * 64 bytes (n = 0, 1, 2, ⋯);
4. This interface is not supported on MLU100.

7.10 Synchronization Function

7.10.1 ___sync_all

void __sync_all ();

To synchronize all clusters on which the kernel executes. The interface is supported on MLU270 and on MLU100.

7.10.2 ___sync_all_ipu

void __sync_all_ipu
();

Synchronize all ipucores of clusters on which the kernel executes. The interface is supported on MLU270, not on MLU100.

7.10.3 ___sync_all_mpu

void __sync_all_mpu

();

Synchronize all memcores of clusters on which the kernel executes. The interface is supported on MLU270, not on MLU100.

7.10.4 ___sync_cluster

void __sync_cluster ();

To synchronous all cores inside the cluster on which the kernel executes. The interface is supported on MLU270 and on MLU100.

7.11 Control Flow And Debugging Function

7.11.1 ___abort

void __abort

();

Terminate current process.

7.11.2 ___assert

void __assert

(bool flag);

Make assertion in the code, program stopped when assert condition is false in time of running, doing nothing when assert condition is true.

- Parameters:
  <flag>: assertion flag.

7.11.3 exit

void exit

(int status);

To return and exit from current function.

- Parameters:
  <status>: exit status

7.11.4 printf

int printf

(const char * fmt, ⋯);

Print arguments to screen, formatted by the format string.

- Parameters:

  <fmt> : The format string

7.11.5 __breakdump_scalar

void __breakdump_scalar (type a [, type b] [, type c] [, type d] [, type e] [, type f]);

Accept scalar parameters, and dump their values into file .dumpscalar_data.

- Parameters:

  {type} The type of data apply to all parameters.It must be one of: short, int, unsigned short, unsigned int, half.The data type of all parameters can be the same or different.

  <a>, ⋯, <f> The scalar parameters to be dumped.

- Remarks:

1. __breakdump_scalar can accept at most 6 parameters.

7.11.6 __breakdump_vector

void __breakdump_vector (void *src, int size, mluBreakDumpAddrSpace_t AddrSpace);

Dump a set of user specified area values to file .dumpvector_data.

- Parameters:

  <src> The addrse of vector values to be dumped

  <size> The number of bytes of vector values to be dumped.

  <AddrSpace> The addrse space of vector values to be dumped. It must be one of: NRAM, LDRAM.

- Remarks:

1. size must be less than or equal to 1024;
2. The vector values must be stored in the NRAM addrse space if the AddrSpace is NRAM;
3. The vector values must be stored in the LDRAM addrse space if the AddrSpace is LDRAM.

- Example

```
#include "mlu.h"

__mlu_entry__ void kernel(half* out_data, half in1, half in2) {
  __ldram__ half ldram_out[16];
  __nram__ half nram_out[16];
  int flag = 1;
  for (int i = 0; i < 16; ++i) {
    ldram_out[i] = flag ? in1 + i : in2 - i;
    flag = flag ? 0 : 1;
  }
  __memcpy(nram_out, ldram_out, 16 * sizeof(half), LDRAM2NRAM);
  __memcpy(out_data, nram_out, 16 * sizeof(half), NRAM2GDRAM);
  __breakdump_vector(ldram_out, 16 * sizeof(half), LDRAM);
}
```

7.11.7 ___bang_printf

void __bang_printf (const char *fmt [, type a1] [,type a2] [, type a3] ···[, type a16]);

Similar to standard printf function, print arguments to screen, formatted by the format string.

- Parameters:
  - <fmt> The format string, which must be a literal constant string.
  - {type} The type of parameters.It must be one of: char, unsigned char, short, unsigned short, int, unsigned int, half, float, char, bool, pointer.The type of all parameters can be the same or different.
- Remarks:

1. type cannot be int8;
2. __bang_printf can at most accept 17 parameters including the format string;
3. fmt must be a literal constant string, for the description of format string, please see chapter 9.3.

7.12 CNSCCL Function

CNSCCL means Cambricon Neuware Shared-Memory Collective Communication Library. The API is supported on MLU270, not on MLU100. The API is as follows.

7.12.1 cnscclAllReduce

void cnscclAllReduce (const void** sendbuff, void** recvbuff, int count, cnscclDataType_t type, cnscclRedOp_t op);

Reduces data arrays of length count in sendbuff using op operation and leaves identical copies of the result on each recvbuff.

- Parameters:
  - <sendbuff> Pointer to the data to read from.
  - <recvbuff> Pointer to the data to write to.
  - <count> Number of elements to process.
  - <type> Type of element. cnscclDataType_t is enum { cnscclChar = 0,cnscclShort = 1,cnscclHalf = 2, cnscclInt = 3, cnscclFloat = 4, cnsccl_NUM_TYPES = 5 }.
  - <op> Operation to perform on each element. cnscclRedOp_t is enum { cnscclSum = 0,cnscclProd = 1,cnscclMax = 2,cnscclMin = 3,cnscclAnd = 4, cnscclOr = 5,cnscclXor = 6, cnscclFuncNull = 7, cnsccl_NUM_OPS= 8 }.
- Example:

```
__mlu_entry__ void deviceAllReduceTestInt(int *ptrInput,
                                          int *ptrOutput,
                                          size_t sizeInput,
                                          size_t sizeOutput,
                                          int *time,
                                          cnscclRedOp_t typeOp,
                                          cnscclDataType_t typeData)
{
  const int COUNT = SIZE_BUFFER / sizeof(int);

  __mlu_shared__ int buffSend[CLUSTER_DIM][COUNT];
  __mlu_shared__ int buffRecv[CLUSTER_DIM][COUNT];

  // 1. Load Data
  __memcpy(&buffSend[clusterId][0], ptrInput + clusterId * COUNT,
        sizeInput / clusterDim, GDRAM2SRAM);

  __sync_all();

  // 2. Call Lib CNSCCL Kernel API
  cnscclAllReduce((void*)&buffSend,
                 (void*)&buffRecv,
                 COUNT,
                 typeData,
                 typeOp);

 // 3. Store Data
 __memcpy(ptrOutput + clusterId * COUNT, &buffRecv[clusterId][0],
        sizeOutput / clusterDim, SRAM2GDRAM);
}
```

7.12.2 cnscclBroadcast

void cnscclBroadcast (void* buff, int count, cnscclDataType_t type, int root);

Copies data the count values from the root directory to all of the other devices. The root directory is the rank where data resides before the operation is started.

- Parameters:
  - <sendbuff> Pointer to the data to read from (root) or write to (non-root).
  - <count> Number of elements to process.
  - <type> Type of element.
  - <root> Rank of the root of the operation.
- Example:

```
__mlu_entry__ void deviceBroadcastTestInt(int *ptrInput,
                                          int *ptrOutput,
                                          size_t sizeInput,
                                          size_t sizeOutput,
                                       int *time,
                                       cnscclDataType_t typeData,
                                       int root)
{
  const int COUNT = SIZE_BUFFER / sizeof(int);

  __mlu_shared__ int buffRecv[COUNT];

  // 1. Load Data
  if (clusterId == root) {
    __memcpy(buffRecv, ptrInput + root * COUNT, sizeInput / clusterDim, GDRAM2SRAM);
  }

  __sync_all();

  // 2. Call Lib CNSCCL Kernel API
  cnscclBroadcast((void*)&buffRecv,
                  COUNT,
                  typeData,
                  root);

  // 3. Store Data
  __memcpy(ptrOutput + clusterId * COUNT, buffRecv,
           sizeOutput / clusterDim, SRAM2GDRAM);
}
```

7.12.3 cnscclGather

void cnscclGather (const void** sendbuff, void** recvbuff, int count, cnscclDataType_t type, int root);

Gathers count values from other MLUs into recvbuff, receiving data from rank i at offset i*count.

- Parameters:
  - <sendbuff> Pointer to the data to read from.
  - <recvbuff> Pointer to the data to write to. This should be the size of count*nranks.
  - <count> Number of elements sent per rank.
  - <type> Type of element.
  - <root> Rank of the root of the operation.
- Example:

```
__mlu_entry__ void deviceGatherTestInt(int *ptrInput,
                                       int *ptrOutput,
                                       size_t sizeInput,
                                       size_t sizeOutput,
                                       int *time,
                                       cnscclDataType_t typeData,
                                       int root)
{
```

```
const int COUNT = SIZE_BUFFER / sizeof(int);

__mlu_shared__ int buffSend[COUNT];
__mlu_shared__ int buffRecv[CLUSTER_DIM][COUNT];

// 1. Load Data
__memcpy(&buffSend, ptrInput + clusterId * COUNT,
         sizeInput / clusterDim, GDRAM2SRAM);

__sync_all();


// 2. Call Lib CNSCCL Kernel API
cnscclGather((void*)&buffSend,
              (void*)&buffRecv,
              COUNT,
              typeData,
              root);

 // 3. Store Data
if (clusterId == root) {
   __memcpy(ptrOutput, buffRecv, sizeOutput, SRAM2GDRAM);
}
}
```

7.12.4 cnscclReduce

void cnscclReduce (const void** sendbuff, void** recvbuff, int count, cnscclDataType_t type, cnscclRedOp_t op, int root);

Reduces data arrays of length count in sendbuff using op operation.

- Parameters:
  - <sendbuff> Pointer to the data to read from.
  - <recvbuff> Pointer to the data to write to.
  - <count> Number of elements to process.
  - <type> Type of element.
  - <op> Operation to perform on each element
  - <root> Rank of the root of the operation.
- Example:

```
__mlu_entry__ void deviceReduceTestInt(int *ptrInput,
                                       int *ptrOutput,
                                       size_t sizeInput,
                                       size_t sizeOutput,
                                       int *time,
                                       cnscclRedOp_t typeOp,
                                       cnscclDataType_t typeData,
                                       int root)
```

```
{
   const int COUNT = SIZE_BUFFER / sizeof(int);

   __mlu_shared__ int buffSend[COUNT];
   __mlu_shared__ int buffRecv[COUNT];

   // 1. Load Data
   __memcpy(&buffSend, ptrInput + clusterId * COUNT, sizeInput / clusterDim,
         GDRAM2SRAM);

   __sync_all();

   // 2. Call Lib CNSCCL Kernel API
   cnscclReduce((void*)&buffSend,
                (void*)&buffRecv,
                COUNT,
                typeData,
                typeOp,
                root);

   // 3. Store Data
   if (clusterId == root) {
      __memcpy(ptrOutput + root * COUNT, buffRecv, sizeOutput / clusterDim,
            SRAM2GDRAM);
   }
}
```

7.12.5 cnscclReduceScatter

void cnscclReduceScatter (const void** sendbuff, void** recvbuff, int count, cnscclDataType_t type, cnscclRedOp_t op, int root);

Reduces data arrays of length count in sendbuff using op operation and leaves the reduced result scattered over the devices so that the recvbuff on rank i will contain the i-th block of the result.

- Parameters:
  - <sendbuff> Pointer to the data to read from.
  - <recvbuff> Pointer to the data to write to.
  - <count> Number of elements to process.
  - <type> Type of element.
  - <op> Operation to perform on each element.
- Example:

```
__mlu_entry__ void deviceReduceScatterTestInt(int *ptrInput,
                                              int *ptrOutput,
                                              size_t sizeInput,
                                              size_t sizeOutput,
                                              int *time,
                                              cnscclRedOp_t typeOp,
                                              cnscclDataType_t typeData)
  {
    const int COUNT = SIZE_BUFFER / sizeof(int);

    __mlu_shared__ int buffSend[COUNT];
    __mlu_shared__ int buffRecv[COUNT/CLUSTER_DIM];

    // 1. Load Data
    __memcpy(&buffSend, ptrInput + clusterId * COUNT, sizeInput / clusterDim,
           GDRAM2SRAM);

    __sync_all();

    // 2. Call Lib CNSCCL Kernel API
    cnscclReduceScatter((void*)&buffSend,
                        (void*)&buffRecv,
                        COUNT,
                        typeData,
                        typeOp);

    // 3. Store Data
    __memcpy(ptrOutput + clusterId * COUNT, buffRecv, sizeOutput / clusterDim,
           SRAM2GDRAM);
  }
```

7.13 Built-in Function Change Description

The following table gives the change description of the deprecated built-in function interfaces. We strongly recommend that users use the new interface programming to ensure compatibility, while ensuring that CNCC continues to be compatible with the old interface that was deprecated in the two versions after the change.

For more information about the function compatibility, please refer to Built-in Function Compatiblity between MLU100 and MLU270.

Table 7.6: Built-in Function Change Description

Serial number	Deprecated Interface	Corresponding New Interface	Changed Version
1	__max	max	BANG C 1.3.0
2	__min	min	BANG C 1.3.0
3	__abs	abs	BANG C 1.3.0
4	__sv_add	__bang_add	BANG C 1.3.0

Continued on next page

Table 7.6 – continued from previous page

Serial number	Deprecated Interface	Corresponding New Interface	Changed Version
5	__sv_sub	__bang_sub	BANG C 1.3.0
6	__sv_mul	__bang_mul	BANG C 1.3.0
7	__sv_mul_const	__bang_mul_const	BANG C 1.3.0
8	__sv_eq	__bang_eq	BANG C 1.3.0
9	__sv_ne	__bang_ne	BANG C 1.3.0
10	__sv_gt	__bang_gt	BANG C 1.3.0
11	__sv_ge	__bang_ge	BANG C 1.3.0
12	__sv_lt	__bang_lt	BANG C 1.3.0
13	__sv_le	__bang_le	BANG C 1.3.0
14	__sv_and	__bang_and	BANG C 1.3.0
15	__sv_or	__bang_or	BANG C 1.3.0
16	__sv_xor	__bang_xor	BANG C 1.3.0
17	__sv_select	__bang_select	BANG C 1.3.0
18	__sv_max	__bang_max	BANG C 1.3.0
19	__sv_not	__bang_not	BANG C 1.3.0
20	__sv_count	__bang_count	BANG C 1.3.0
21	__sv_write_zero	__bang_write_zero	BANG C 1.3.0
22	__sv_cycle_add	__bang_cycle_add	BANG C 1.3.0
23	__sv_cycle_sub	__bang_cycle_sub	BANG C 1.3.0
24	__sv_cycle_mul	__bang_cycle_mul	BANG C 1.3.0
25	__sv_cycle_eq	__bang_cycle_eq	BANG C 1.3.0
26	__sv_cycle_ne	__bang_cycle_ne	BANG C 1.3.0
27	__sv_cycle_gt	__bang_cycle_gt	BANG C 1.3.0
28	__sv_cycle_ge	__bang_cycle_ge	BANG C 1.3.0
29	__sv_cycle_lt	__bang_cycle_lt	BANG C 1.3.0

Continued on next page

Table 7.6 – continued from previous page

Serial number	Deprecated Interface	Corresponding New Interface	Changed Version
30	__sv_cycle_le	__bang_cycle_le	BANG C 1.3.0
31	__sv_cycle_and	__bang_cycle_and	BANG C 1.3.0
32	__sv_cycle_or	__bang_cycle_or	BANG C 1.3.0
33	__sv_cycle_xor	__bang_cycle_xor	BANG C 1.3.0
34	__relu	__bang_active_relu	BANG C 1.3.0
35	__sigmoid	__bang_active_sigmoid	BANG C 1.3.0
36	__conv	__bang_conv	BANG C 1.3.0
37	__mlp	__bang_mlp	BANG C 1.3.0
38	__pad	__bang_pad	BANG C 1.3.0
39	__transpose	__bang_transpose	BANG C 1.3.0
40	__maxpool	__bang_maxpool	BANG C 1.3.0
41	__minpool	__bang_minpool	BANG C 1.3.0
42	__avgpool	__bang_avgpool	BANG C 1.3.0
43	__maxpool_index	__bang_maxpool_index	BANG C 1.3.0
44	__minpool_index	__bang_minpool_index	BANG C 1.3.0
45	__sv_half2_dn	__bang_half2int8_dn	BANG C 1.3.0
46	__sv_int82half	__bang_int82half	BANG C 1.3.0
47	__sv_int82half	__bang_int82half	BANG C 1.3.0
48	__sv_short2half	__bang_short2half	BANG C 1.3.0
49	__sv_half2short_dn	__bang_half2short_dn	BANG C 1.3.0
50	__unpool	__bang_unpool	BANG C 1.3.0
51	__bang_count(half * dst, half * src,int NumOfEle)	__bang_count(unsigned short * dst,half * src, int NumOfEle)	BANG C 1.4.0

Continued on next page

Table 7.6 – continued from previous page

Serial number	Deprecated Interface	Corresponding New Interface	Changed Version
52	__bang_maxpool_index(half * src, const int ic, const int ih, const int iw, const int kh, const int kw [,const int sx, const int sy])	__bang_maxpool_index(half * src, const int ic, const int ih, const int iw, const int kh, const int kw [,const int sx, const int sy])	BANG C 1.4.0
53	__bang_minpool_index(half * dst, half * src, const int ic, const int ih, const int iw, const int kh, const int kw [,const int sx, const int sy])	__bang_minpool_index(unsigned short * dst, half * src, const int ic, const int ih, const int iw, const int kh, const int kw [,const int sx, const int sy])	BANG C 1.4.0
54	__bang_half2int8_dn(half * dst, half * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	__bang_half2int8_dn(int8 * dst, half * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	BANG C 1.4.0
55	__bang_int82half(half * dst, half * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	__bang_int82half(half * dst, int8 * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	BANG C 1.4.0
56	__bang_uchar2half(half * dst, half * src, int NumOfEle [,int dststride, int srcstride, int NumOfSection])	__bang_uchar2half(half * dst, unsigned char * src, int NumOfEle [,int dststride, int srcstride, int NumOfSection])	BANG C 1.4.0
57	__bang_half2short_dn(half * dst, half * src, int NumOfEle [,int dststride, int srcstride, int NumOfSection])	__bang_half2short_dn(short * dst, half * src, int NumOfEle [,int dststride, int srcstride, int NumOfSection])	BANG C 1.4.0
58	__bang_half2int8_dn(int8 * dst, half * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	__bang_half2int8_dn(int8 * dst, half * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	BANG C 1.5.0

Continued on next page

Table 7.6 – continued from previous page

Serial number	Deprecated Interface	Corresponding New Interface	Changed Version
59	__bang_int82half(half * dst, int8 * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	__bang_int82half(half * dst, int8 * src, int NumOfEle, int pos [,int dststride, int srcstride, int NumOfSection])	BANG C 1.5.0

7.14 Built-in Function Compatiblity between MLU100 and MLU270

7.14.1 __bang_conv

For details of the difference of conv data types between MLU100 and MLU270, please refer to Compatibility between MLU100 and MLU270.

7.14.2 __bang_mlp

For details of the difference of mlp data types between MLU100 and MLU270, please refer to Compatibility between MLU100 and MLU270.

7.14.3 __bang_pad

Changed parameter restriction, for details see function description of __bang_pad.

7.14.4 Class of Cycle Instruction

Changed parameter restriction,

1. <src_elem_count> * sizeof (half) and <seg_elem_count> * sizeof (half) must be dividable by 128 on MLU270;
2. <src_elem_count> % <seg_elem_count> == 0 must be satisfied on MLU270;
3. <src_elem_count> and <seg_elem_count> must be dividable by 64 on MLU100;

7.14.5 Class of Type Conversion Instruction

Changed parameter restriction,

1. <src_count> * sizeof (mintype) must be dividable by 128 on MLU270, where mintype is smaller type in src and dst;
2. <src_count> * sizeof (srctype) must be dividable by 128 on MLU100, where mintype is type of src.

7.14.6 __bang_transpose

Changed parameter restriction,

1. <height>*sizeof(type) and <width>*sizeof (type) must be dividable by 128 on MLU270;
2. <height> and <width> must be dividable by 16 on MLU100;

7.14.7 __bang_max/__bang_min

Changed parameter restriction and behavior,

1. The __nram__ space to which the vector operand <dst> points must be at least 128 bytes on MLU270, at least 32 bytes on MLU100;
2. The index value returned by the instruction on MLU100 and MLU270 may be different.

7.14.8 Class of Select Instruction

1. __bang_select and _bang_select_bitindex: <dst> stores the number of selected numbers in 32 bytes on MLU100, 128 bytes on MLU270;
2. __bang_select_bitindex: <elem_count> must be dividable by 1024 on MLU270, be dividable by 64 on MLU100;
3. __bang_maskmove_bitindex: <elem_count> * sizeof (half) must be dividable by 512 on MLU270;
4. 1 bit of bitmask select one byte of src in maskmove_bitindex instruction, 1 bit of bitmask select one number of src in collect and select instruction.

7.14.9 __bang_maximum

This instruction is newly added on MLU270.

7.14.10 __bang_count

Changed parameter restriction,

1. The __nram__ space to which the vector operand <dst> points must be at least 128 bytes on MLU270, at lease 32 bytes on MLU100.

7.14.11 __memcpy

Changed parameter restriction. For details, see section __memcpy .

8 Performance Guide

Compared with the traditional programming language, BANG C language pays more attention to making full use of computing units of hardware. Because compared with a general processor, one of the advantages of MLU is that MLU has a larger number of parallel computing units to deal with large-scale data. For example, it is faster to calculate vector multiplication with a tensor computing unit than the scalar computing unit. This chapter introduces common optimization methods for improving the performance of BANG C language.

8.1 Maximize On-Chip Memory Computation

8.1.1 NRAM Computation

Since the memory access speed of nram is fast, it is recommended that users use nram instead of gdram/ldram for computation. The example code is as follows. On the MLU270 platform, the performance of nram version improves by about 10 times.

GDRAM Version:

```
#define LEN 8192
__mlu_entry__ void kernel(half* dst, half* src1, half *src2, int len) {
  for ( int i = 0; i < len; i++) {
    dst[i] = src1[i] * src2[i];
  }
}
```

NRAM Version:

```
#define LEN 8192
__mlu_entry__ void kernel(half* dst, half* src1, half *src2, int len) {
  __nram__ half src1_nram[LEN];
  __nram__ half src2_nram[LEN];
  __memcpy(src1_nram, src1, len * sizeof(half), GDRAM2NRAM);
  __memcpy(src2_nram, src2, len * sizeof(half), GDRAM2NRAM);
  for (int i = 0; i < len; i++) {
    src2_nram[i] = src1_nram[i] * src2_nram[i];
  }
  __memcpy(dst, src2_nram, len * sizeof(half), NRAM2GDRAM);
}
```

8.1.2 SRAM Computation

Since the shared memory sram is on chip and memory access speed is fast, it is recommended that users use sram instead of gdram/ldram for computation when storing tempory variable. The example code is as follows.

```
#define CLUSTER_DIM 1

#define CORE_DIM 4

#define TASK_DIM (CLUSTER_DIM * CORE_DIM)

#define STRIDE 1024

__mlu_entry__ void reductionKernel(float* out_sum,
                                   float* arr,
                                   int num_elems) {
 __nram__ float sum[STRIDE];
 __nram__ float val[STRIDE];
 __mlu_shared__ float partial_sum[TASK_DIM];

 int start_offset = taskId * task_stride;
 int end_offset = (taskId + 1) * task_stride;
 if (end_offset > num_elems) end_offset = num_elems;

 __nramset(sum, STRIDE, 0.0f);

 for (int task_offset = start_offset; task_offset < end_offset;
    task_offset += STRIDE) {

  __memcpy(val, arr + task_offset, STRIDE * sizeof(float),
          GDRAM2NRAM);

  __bang_add(sum, sum, val, STRIDE);
 }

 for (int i = 0; i < 32; i ++) {
  __bang_reduce_sum(val, sum + i * 32);
  sum[i] = val[0];
 }

 __bang_reduce_sum(val, sum);

 //! Store the result into the shared memory
 partial_sum[taskId] = val[0];

 __sync_cluster();

 if(taskId == 0)
 {
       for(int i = 0; i < taskDim; i ++) {
         out_sum[0] += partial_sum[i];
       }
 }
}
```

For more information about SRAM restriction, please refer to SRAM .

8.2 Maximize Parallel Computation

8.2.1 Stream Computation

Since the operation speed of stream is fast, it is recommended that users use the stream operation instead of the scalar operation. The example code is as follows. The optimization speed of stream operation is positively correlated with the length of stream data. For the code below, on the MLU270 platform, when the data length is 8192, the performance is improved by about 40 times; when the data length is 65535, the performance is improved by about 150 times.

Serial Version:

```
#define CHANNELS  8192
__mlu_entry__ void kernel(half* dst, half* src1, half *src2, int len) {
  int channels = CHANNELS;
  __nram__ half src_nram[2][CHANNELS]; // macro for constant
  __nram__ half dst_nram[CHANNELS];
  __memcpy(src_nram[0], src1, channels * sizeof(half), GDRAM2NRAM);
  __memcpy(src_nram[1], src2, channels * sizeof(half), GDRAM2NRAM);
  for (int i =0 ;I < channels; i++) {
    dst_nram[i] = __max(src_nram[0][i], src_nram[1][i]);
  }
  __memcpy(dst, dst_nram, channels * sizeof(half), NRAM2GDRAM);
}
```

Stream Version:

```
#define CHANNELS  8192
__mlu_entry__ void kernel(half* dst, half* src1, half *src2, int len) {
  int channels = CHANNELS;
  __nram__ half src_nram[2][CHANNELS]; // macro for constant
  __nram__ half dst_nram[CHANNELS];
  __memcpy(src_nram[0], src1, channels * sizeof(half), GDRAM2NRAM);
  __memcpy(src_nram[1], src2, channels * sizeof(half), GDRAM2NRAM);
  __bang_maxpool(dst_nram, src_nram, CHANNELS, 1, 2, 1, 2); // macro for constant
  __memcpy(dst, dst_nram, channels * sizeof(half), NRAM2GDRAM);
}
```

8.2.2 Multi-core Parallel

MLU270 is a multi-core heterogeneous platform that may be used for multi-core acceleration computation. Example code for dividing two arrays using multi-core in parallel is given below. When using 32 cores, the performance is improved by about 10 times compared to a single-core performace.

Serial Version:

```
#define _USE_MULTICORE_ 1
#define CORE_NUM 32
#define LEN 65536
#define PER_CORE_LEN  (LEN/CORE_NUM)  // should be multiples of 64
__mlu_entry__ void kernel(half* dst, half* src1, half *src2, int len) {
  __nram__ half src1_nram[LEN];
  __nram__ half src2_nram[LEN];
  __memcpy(src1_nram, src1, len * sizeof(half), GDRAM2NRAM);
  __memcpy(src2_nram, src2, len * sizeof(half), GDRAM2NRAM);
  for ( int i = 0; i < len; i++) {
    src2_nram[i] = src1_nram[i] / src2_nram[i];
  }
  __memcpy(dst, src2_nram, len * sizeof(half), NRAM2GDRAM);
}
```

Parallel Version:

```
#define _USE_MULTICORE_ 1
#define CORE_NUM 32
#define LEN 65536
#define PER_CORE_LEN  (LEN/CORE_NUM)  // should be multiples of 64
__mlu_entry__ void kernel(half* dst, half* src1, half *src2, int len) {
  int per_core_len = PER_CORE_LEN;
  __nram__ half src1_nram[PER_CORE_LEN]; // use macro for constant
  __nram__ half src2_nram[PER_CORE_LEN]; // use macro for constant
  __memcpy(src1_nram, src1 + taskId * per_core_len, per_core_len * sizeof(half), GDRAM2NRAM);
  __memcpy(src2_nram, src2 + taskId * per_core_len, per_core_len * sizeof(half), GDRAM2NRAM);
  for ( int i = 0; i < per_core_len; i++) {
    src2_nram[i] = src1_nram[i] / src2_nram[i];
  }
  __memcpy(dst + taskId * per_core_len, src2_nram, per_core_len * sizeof(half), NRAM2GDRAM);
}
```

8.3 Optimizing Half And Float Computation

8.3.1 Half Computation

Enable half optimization on MLU platform by specifying '-fmlu-fast-math' option can significantly improve performance for half-computation-intensive applications. The example code is shown as follows. On the MLU100 platform, the performance of fast-math version improves by about 2 times.

```
#define LEN     8192
__mlu_entry__ void kernel(half *input, int len, int *output) {
       __nram__ half src_nram[LEN];
       __nram__ int dst_nram[LEN];
       __memcpy(src_nram, input, len * sizeof(half), GDRAM2NRAM);
       for (int i = 0; i < len; i++) {
               dst_nram[i] = floor(src_nram[i]);
       }
       __memcpy(output, dst_nram, len * sizeof(int), NRAM2GDRAM);
}
```

8.4 Utilizing SRAM Communication Library

8.4.1 CNSCCL Computation

On MLU270 with multi clusters, after each cluster doing its own operation on variables on SRAM , all the data need be reduced or gathered to do some operation. To achieve best performance, CNSCCL(Cambricon Neuware Shared-Memory Collective Communication Library) lib should be called. The example code is as follows.

```
__mlu_entry__ void deviceAllReduceTestInt(int *ptrInput,
                                       int *ptrOutput,
                                       size_t sizeInput,
                                       size_t sizeOutput,
                                       int *time,
                                       cnscclRedOp_t typeOp,
                                       cnscclDataType_t typeData)
{
  const int COUNT = SIZE_BUFFER / sizeof(int);

  __mlu_shared__ int buffSend[CLUSTER_DIM][COUNT];
  __mlu_shared__ int buffRecv[CLUSTER_DIM][COUNT];

  // 1. Load Data
  __memcpy(&buffSend[clusterId][0], ptrInput + clusterId * COUNT,
        sizeInput / clusterDim, GDRAM2SRAM);

  __sync_all();

  // 2. Call Lib CNSCCL Kernel API
  cnscclAllReduce((void*)&buffSend,
                 (void*)&buffRecv,
                 COUNT,
                 typeData,
                 typeOp);

 // 3. Store Data
 __memcpy(ptrOutput + clusterId * COUNT, &buffRecv[clusterId][0],
        sizeOutput / clusterDim, SRAM2GDRAM);
}
```

8.5 Best Practice

In this section, the Discrete Fourier Transform (DFT) algorithm is taken as an example to introduce the best practice of BANG C language.

8.5.1 Process of DFT

DFT is an algorithm that transforms signals from time domain to frequency domain, and both time domain and frequency domain are discrete. DFT can find out what kinds of sine waves a signal is composed of. The result of DTP is the amplitude and phase of these sine waves. The transformation formula is as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi}{N}kn}$$

$X(k)$ is the frequency domain sequence after transformation, and $x(n)$ is the time domain sequence before transformation.

If $e^{-j\frac{2\pi}{N}kn}$ part of the above formula is replaced by Euler formula $e^{jx} = \cos\ x + j\sin x$, the following formula can be obtained:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)[\cos(-\frac{2\pi}{N}k) + j\sin(\frac{2\pi}{N}kn)] \\ &= \sum_{n=0}^{N-1}[x(n)\cos(\frac{2\pi}{N})kn - jx(n)\sin(\frac{2\pi}{N}kn)] \end{aligned}$$

The real part and the imaginary part of $X(k)$ are respectively as follows:

$$\begin{aligned} real(k) &= \sum_{n=0}^{N-1} x(n)\cos(\frac{2\pi}{N}n) \\ imag(k) &= \sum_{n=0}^{N-1} -x(n)\sin(\frac{2\pi}{N}kn) \end{aligned}$$

The amplitude of $X(k)$ is as follows:

$$Amp(k) = \sqrt{real(k)^2 + imag\ \ (k)^2}$$

It is easy to understand by using the scalar statement of BANG C language to realize this calculation logic, as the following sample.

```
#define PI 3.14159265
#define N 128
__mlu_entry__ void DFT(float*x, float*Amp){
 for (int k = 0;k<N; k++){
  float real = 0.0;
  float imag = 0.0;
  for (int n = 0; n<N; n++){
  real += x[n] * cosf(2*PI/N*k*n);
  imag += -x[n] * sinf(2*PI/N*k*n);
  }
 Amp[k] = sqrtf(real*real + imag*imag);
 }
}
```

This program is a two-layer for loop, with n as the number of cycles, and O(n2) as the time complexity. The input and output of the program are respectively an array of N elements. In addition, there are only a few temporary variables such as real and imag. Therefore, the space complexity of the program is O (n). The calculation amount of multiplication and addition in this program mainly includes "k * n", "x [n] * sin()", "x [n] * cos()", "real * real", "imag * imag", "real + =", "imag + =" and "real * real + imag * imag". Therefore, the strategies of optimization focus on these operations. The program will be optimized based on the mentioned performance optimization method.

8.5.2 On-Chip Memory Computation

The original program only uses the MLU memory GDRAM. For the on-chip memory computation, we move the input and output data to NRAM on the chip before calculation. The modified process includs the following steps: applying the NRAM space, moving the input data from GDRAM to NRAM, calculating and moving the output data from NRAM to GDRAM, as in following sample.

```
# define PI 3.14159265
# define N 128
__mlu_entry__ void DFT (float * x, float * Amp){
  __nram__ float in[N];
  __nram__ float out[N];
  __memcpy (in, x, N * sizeof (float), GDRAM2NRAM);

  for (int k = 0; k < N; k ++){
   float real = 0.0;
   float imag = 0.0;
   for (int n = 0; n < N; n ++){
    real += in[n] * cosf (2 * PI/N * k * n);
    imag += -in[n] * sinf (2 * PI/N * k * n);
  }
  out[k] = sqrtf (real * real + imag * imag);
 }
 __memcpy (Amp, out, N * sizeof (float), NRAM2GDRAM);
}
```

8.5.3 Tensor Computation

The original program uses scalar calculation statements. We optimize the statement "out [k] = sqrt (real * real + imag * imag)" into the tensor calculation statement of the BANG C language. As for the inner for loop, we will polish the algorithm optimization in the next step. The optimized code directly uses the build-in function, as shown in the following sample.

```
# define PI 3.14159265
# define N 128
__mlu_entry__ void DFT (float * x, float * Amp){
 __nram__ float in[N];
 __nram__ float out[N];
 __nram__ float real[N];
 __nram__ float imag[N];
 __memcpy (in, x, N * sizeof (float), GDRAM2NRAM);

 for (int k = 0; k < N; k ++){
  real[k] = 0.0;
  imag[k] = 0.0;
  for (int n = 0; n < N; n ++){
   real[k] += in[n] * cosf (2 * PI/N * k * n);
   imag[k] +=  in[n] * sinf (2 * PI/N * k * n);
  }
 }

 __bang_mul (real, real, real, N);
 __bang_mul (imag, imag, imag, N);
 __bang_add (out, real, imag, N);
 __bang_active_sqrt (out, out, N);
 __memcpy (Amp, out, N * sizeof (float), NRAM2GDRAM);
}
```

8.5.4 Algorithm Optimization

At the logic level of the algorithm, the optimization method is as follows:

1. Calculate sin and cos once, for the parameter of sin and cos are the same in the original algorithm, to avoid repetitive computation;
2. Calculate once outside the loop, for the first three of the five numbers "2 * PI / N * k * n" are constants, to avoid repetitive computation;
3. Take a negative number of imag for accumulation, for imag will be squared later.

The optimized code is shown as follows.

```
# define PI 3.14159265
# define N 128
__mlu_entry__ void DFT (float * x, float * Amp){
 __nram__ float in[N];
 __nram__ float out[N];
 __nram__ float real[N];
 __nram__ float imag[N];
 __memcpy (in, x, N * sizeof (float), GDRAM2NRAM);

 float con = 2 * PI/N;
 for (int k = 0; k < N; k ++){
  real[k] = 0.0;
  imag[k] = 0.0;
  for (int n = 0; n < N; n ++){
   float tmp = con * k * n;
   real[k] += in[n] * cosf (tmp);
   imag[k] += -in[n] * sinf (tmp);
  }
 }

 __bang_mul (real, real, real, N);
 __bang_mul (imag, imag, imag, N);
 __bang_add (out, real, imag, N);
 __bang_sqrt (out, out, N);
 __memcpy (Amp, out, N * sizeof (float), NRAM2GDRAM);
}
```

Real and imag calculation are still left for optimization:

- k * n in the inner for loop;
- “real [k] += in [n] * cos()” and “imag [k] += in[n] * sin()” in the inner for loop.

These two calculation are most likely to be associated with convolution. Therefore, we use __bang_conv to transform these operations.

For k * n in the two-layer for loop, N^2 results will be generated. The process of convolution is shown in Figure Convolution of real and imag Transformation .

0	0	...	0
0	1	...	127
...	...	...	...
0	127	...	127^2

Fig. 8.1: Matrix Multiplication of k*n

The function _bang_mult_const mulitplies each element of a vector by a constant, and we use it to transform 2 * PI/N.

Then the scalars sin and cos can also be replacedd with vector computation statements.

The “real[k] += in[n] * cos (tmp)” and “imag[k] += -in[n] * sin (tmp)” wrapped by the for loop can be converted into _bang_conv, and all the results of k*n are saved in this matrix. Convolution operation is illustrated in Figure Convolution of real and imag Transformation .

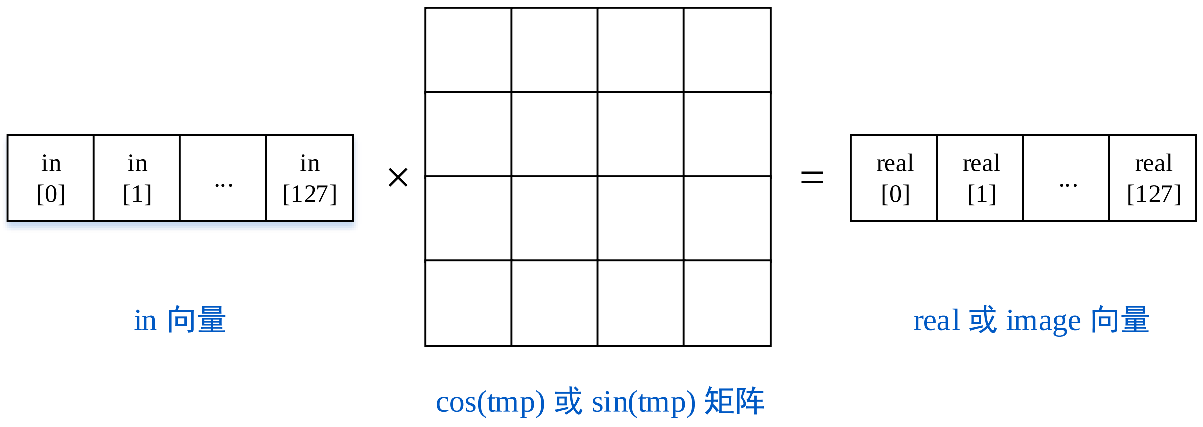

Fig. 8.2: Convolution of real and imag Transformation

The original two-layer for loop has been completely replaced by a series of transformations, as the following sample.

```
# define PI 3.14159265
# define N 128
__mlu_entry__ void DFT (float * x, float * Amp){
 __nram__ int16_t in[N];
 __nram__ float real[N];
 __nram__ float imag[N];
 __nram__ float kn[N * N];
 __nram__ float cos1[N * N];
 __nram__ float sin1[N * N];
 __nram__ int16_t cos2[N * N];
 __nram__ int16_t sin2[N * N];
 __wram__ int16_t cosw[N * N];
 __wram__ int16_t sinw[N * N];
 __nram__ float cos_res[N * N];
 __memcpy(in, x, N * sizeof(int16_t), GDRAM2NRAM);
 int idx = -1;
 for (int k = 0; k < N; k++) {
   for (int n = 0; n < N; n++) {
     kn[++idx] = k * n;
   }
 }
 __bang_mul_const(kn, kn, 2 * PI / N, N * N);

 __bang_active_sin(sin1, kn, N * N);
 __bang_active_cos(cos1, kn, N * N);
 __bang_float2int16_dn(sin1, sin1, N * N, 0);
 __bang_float2int16_dn(cos1, cos1, N * N, 0);
 __bang_reshape_filter(sin2, sin1, N, 1, 1, N);
 __bang_reshape_filter(cos2, cos1, N, 1, 1, N);
 __memcpy(sinw, sin2, N * N * sizeof(int16_t), NRAM2WRAM);
 __memcpy(cosw, cos2, N * N * sizeof(int16_t), NRAM2WRAM);
 __bang_conv(real, in, cosw, N, 1,
        1, 1, 1, 1, 1,
       N, 0);
__bang_conv(imag, in, sinw, N, 1,
       1, 1, 1, 1, 1,
       N, 0);
__bang_mul(real, real, real, N);
__bang_mul(imag, imag, imag, N);
__bang_add(imag, real, imag, N);
__bang_active_sqrt(imag, imag, N);
__memcpy(Amp, imag, N * sizeof(float), NRAM2GDRAM);
}
```

8.5.5 Constant Preprocessing

In the algorithm optimization process, the value of “k * n” matrix is fixed, which means that the value of sin and cos are also fixed. To avoid the repetitive computation, we can calculate these constants in advance and directly pass them in as parameters. The updated code is as follows.

```
# define PI 3.14159265
# define N 128
__mlu_entry__ void DFT (float * x, float * cos_mat, float *
 __nram__ float in[N];
 __nram__ float real[N];
 __nram__ float imag[N];
 __nram__ float cos_res[N * N];
 __nram__ float sin_res[N * N];

 __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);
 __memcpy(cosw, cos_mat, N * N * sizeof(float), GDRAM2NRAM);
 __memcpy(sinw, sin_mat, N * N * sizeof(float), GDRAM2NRAM);

 __bang_conv(real, in, cosw, N, 1,
           1, 1, 1, 1, 1,
           N, 0);
 __bang_conv(imag, in, sinw, N, 1,
           1, 1, 1, 1, 1,
           N, 0);

 __bang_mult(real, real, real, N);
 __bang_mult(imag, imag, imag, N);
 __bang_add(imag, real, imag, N);
 __bang_sqrt(imag, imag, N);
 __memcpy(Amp, imag, N * sizeof(float), NRAM2GDRAM);
}
```

The calculation of cos_mat and sin_mat is completed in CNRT of the Host. And then, copy the data from the Host memory into the MLU memory (GDRAM).

8.5.6 Analysis of Optimization

Based on the above optimization, DFT operation changed from 11-line code with the original scalar calculation in GDRAM to nearly 30 lines optimized code. Except for a small piece of code that can only utilize scalar computation, the code has been converted into tensor computation, and run in NRAM[8] . Although the amount of code increased, the overall performance has been greatly improved by nearly 1800 times. The performance improvement by different optimization methods is shown in Table 8.1.

Table 8.1: Performance Improvement by Different Optimization Methods

Optimization Method	Performance Improvement (Times)
Original scalar program	1 (reference performance)
On-Chip Memory Computation	1.49
Tensor Computation	10.65
Algorithm Optimization	29.27
Constant Preprocessing	1794

From the optimization examples, it is obvious that the performance tuning process is the process of keeping the data close to the arithmetic, and also the process of gradually eliminating the for loop and replacing the scalar calculation statement with the tensor calculation statement.

If we take multi-core parallelism into consideration, the optimization will involve the splitting process of computation. In summary, the performance tuning scheme focuses on making full use of hardware resources, including at least four aspects:

- making full use of near-end storage (using on-chip memory);
- utilizing of tensor arithmetic (tensor quantization);
- reducing the amount of computation and saving storage space, including algorithm optimization and constant preprocessing;
- taking advantage of multi-core parallel (splitting computing task), etc.

[8] In the current performance tuning methods, only single-core optimization has been adopted. Performance can be further improved after adopting multi-core optimization.

9 BANG C Debugging

The debugging of the program on HOST in the MLU heterogeneous program can use the CNGDB debugging method. To make it easier for users to debug KERNEL programs, the BANG C language provides a BREAKDUMP interface for users to look up MLU scalar and vector values. This chapter mainly introduces the usage method and precautions for looking up the value of variables using BREAKDUMP.

9.1 Debugging with CNGDB

CNGDB is a software debugging tool developed by Cambricon, reengineering based on open-source GDB. CNGDB can control programs running on Cambricon's hardware, providing function of monitoring program's running status and retrieving intermediate running result. CNGDB can release the burden of program debugging, improve development efficiency.

Based on the original function, CNGDB has features as follows:

1. Support debugging of BANG C program on device side and C/C++ program on host side at the same time.
2. Support debugging of BANG C single core and multi-core programs.
3. Allow setting breakpoint on MLU, single-step debugging of Cambricon's program.
4. Allow checking and modifying variables on running cores or data in memory.
5. Compatible with original function fully.

In most cases, we can use CNGDB as follows:

```
cngdb add_half-01
```

If debugging program has extra parameters, we can use CNGDB with "–args" as follows:

```
cngdb --args add_half-01 arg1 arg2
```

For more detail of using CNGDB, Cambricon-CN-GDB-User-Guide is prefered.

9.2 Debugging of Scalar

BANG C users can dump the value of a scalar using the __breakdump_scalar function.Its semantics is to dump the values of a set of user-specified scalar variables into the file.dumpscalar_data and terminate the execution of the current program on CORE. The main format is as follows:

```
__breakdump_scalar(var1) // dump the value of 1 scalar
__breakdump_scalar(var1, var2) // dump the values of 2 scalars simultaneously
__breakdump_scalar(var1, var2, var3) // dump the values of 3 scalars simultaneously
__breakdump_scalar(var1, var2, var3, var4) // dump the values of 4 scalars simultaneously
__breakdump_scalar(var1, var2, var3, var4, var5) // dump the values of 5 scalars simultaneously
__breakdump_scalar(var1, var2, var3, var4, var5, var6) // dump the values of 6 scalars␣
↪simultaneously
```

Note:

- Dump up to 6 scalar values simultaneously every time

- After the breakdump_scalar statement, the current CORE in the MLU will terminate the execution of the program.

- The environment variable DUMPMLUSCALAR needs to be set to dump data into .dumpscalar_data.

9.2.1 Example

1. Kernel Code:

```
__mlu_entry__ void kernel(half* out_data,
  half in1,
  half in2) {
  *out_data = __max(in1, in2);
  __breakdump_scalar(*out_data);
}
```

2. Host Code:

Common Host code, without any special modification;

3. Check dump results:

If the users set the DUMPMLUSCALAR environment variable before executing the program, the result will be output to.dumpscalar_data. For example:

```
$ export DUMPMLUSCALAR=1
$ ./a.out
$ ls .dumpscalar_data
.dumpscalar_data
$ cat .dumpscalar_data
MLU coreID 0: Dumping 1 Scalar...
Scalar 0 value : 5520
```

9.3 Debugging of Vector

BANG C users can DUMP the value of a vector using the __breakdump_vector function.Its semantics is to dump the values of a set of user-specified vectors into the file.dumpvector_data and terminate the execution of the current program on CORE. The format of the interface is as follows:

```
__breakdump_vector(void *src, int32_t Bytes, mluBreakDumpAddrSpace_t AddrSpace)
```

The AddrSpace parameter represents the address space where the address data to be dumped is located. Currently, the BANG C language supports the dump function for vector data on the nram/ldram/gdram space, so that the value of AddrSpace is NRAM/LDRAM/GDRAM.

Note:

- Currently supports up to 1024 bytes at a time, that is, Bytes <= 1024.

- The execution of the program on the current CORE is terminated after the __breakdump_vector function is executed.

- The result of dump will be output to the .dumpvector_data file only if the environment variable DUMPMLUVECTOR is set before the program is executed.

9.3.1 Example

1. the Kernel example of Dump LDRAM space array :

```
__mlu_entry__ void kernel(half* out_data,
  half in1,
  half in2) {
  half ldram_out[16];
  __nram__ half nram_out[16];
  for (int i = 0; i < 16; ++i) {
    ldram_out[i] = in1 + in2;
  }
  __memcpy(nram_out, ldram_out, 16 * sizeof(half), LDRAM2NRAM);
  __memcpy(out_data, nram_out, 16 * sizeof(half), NRAM2GDRAM);
  __breakdump_vector(ldram_out, 16 * sizeof(half));
}
```

2. the Kernel example of Dump NRAM space array :

```
__mlu_entry__ void kernel(half* out_data,
  half in1,
  half in2) {
```

```
  __nram__ half nram_out[16];
  for (int i = 0; i < 16; ++i) {
    nram_out[i] = in1 + in2;
  }
  __memcpy(out_data, nram_out, 16 * sizeof(half), NRAM2GDRAM);
  __breakdump_vector(nram_out, 16 * sizeof(half));
}
```

3. Check dump result:

The user sets the DUMPMLUVECTOR environment variable before executing the program, then the result will be output to .dumpvector_data. For example:

```
$ export DUMPMLUVECTOR=1
$ ./a.out
$ls .dumpvector_data
.dumpvector_data
$cat dumpvector_data
MLU coreID 0: Dumping 32 bytes vector data...
80 4e d0 54 00 4f b0 54
80 4f 90 54 00 50 70 54
40 50 50 54 80 50 30 54
c0 50 10 54 00 51 e0 53
```

9.4 Format Output

The BANG C users may print zero or more scalar values using the __bang_printf function.The semantics is to print parameters to the screen (standard output) according to the formatted string. The format of this interface is as follows:

```
__bang_printf (const char* fmt)
__bang_printf (const char* fmt, type arg1)
__bang_printf (const char* fmt, type arg1, type arg2)
__bang_printf (const char* fmt, type arg1, type arg2, type arg3)
......
__bang_printf (const char* fmt, type arg1, type arg2, type arg3,..., type arg16)
```

Note:

- The parameter fmt must be a string constant.
- Type can not be int8, the type of each parameter in __bang_printf can be different.
- In addition to format string, at most 16 parameters can be printed.
- After the __bang_printf function is executed, the execution of the program on the current CORE will not be terminated;
- The users can use the environment variable CNRT_BANG C_PRINTF_LIMIT to specify the size of the print buffer (the value is 1024 by default), if the printout exceeds the size of the buffer, only the latest print results will be retained.

- If the environment variable CNRT_BANG C_PRINTF_LIMIT is set to 0 before cnrtInvokeKernel_V2, it is processed according to the default value of 1024.

- After compilation, one __bang_printf can produce more than 30 machine instructions, with each machine instruction 64 bytes, total 1920 bytes, nearly occupying 2KB ICACHE. To get better performance, we sugguest not using __bang_printf in release mode. The code is as follows.

```
#ifdef _DEBUG
    #define PRINTF(format, ...) __bang_printf(format,
    ##__VA_ARGS__)
    #else
    #define PRINTF(format, ...)
    #endif
```

9.4.1 Description of Format String

The format of format string used by __bang_printf is same as the standard printf format string of C language, but the supported placeholders are a subset of the standard printf. For placeholders supported by __bang_printf, if the placeholder is explicitly defined in standard printf, __bang_printf prints the same result as standard printf.If this placeholder is not explicitly defined in standard printf, __bang_printf may behave differently than standard printf.For example, “%hf” in __bang_printf means that the half type is printed, however, “%hf” in standard printf means that the float/double type is printed by default.

1. Placeholder format

%[flags][width][.precision][length]type

2. Flags domain of placeholder

The Flags domian can be zero or a plurality of combinations of the“-” (minus), “+” (plus), ” “(blank), “0” (zero), and “#” (hash), and the order is not limited.

3. Width domain of placeholder

The width domain can be null or an integer value, but it cannot be a “*” (multiplication sign).

4. Precision domain of placeholder

The precision domain can be null or an integer value, but it cannot be a “*” (multiplication sign).

5. Length domain of placeholder

The Length domain can be null or one of “hh” , “h” , “l” , and “ll” .The current combination of “h” and floating point indicates that the half type is printed (such as “%hf” , “%he”), and the behavior of the combination of “hh” and floating point is undefined.

6. Type domain of placeholder

The Type domian can be any character in the “character” list of the table below. Please **note** that n and s in standard printf are not supported here.

Table 9.1: Type Domain

Character	Classification
%	percent sign
d, i, u, x, X, o	integer
f, F, e, E, g, G, a, A	floating point data
c	single character
p	pointer

7. Suggested length domain and type domain to use to print bangc types

Note:

For supported types, flag domain, width domain, and precision domain can be used as you want. Length domain and type domain are combined to tell the exact type of your data.

Table 9.2: Suggested Length Domain and Type Domain for BANG C Types

BANG C Type	Length Domain and Type Domain
int8_t	%hhd
uint8_t	%hhu
int16_t	%hd
uint16_t	%hu
int32_t	%d
uint32_t	%u
half	%hf
float	%f
char	%c
int8 (8-bit fixd point float number)	Not supported. Please convert to half or float to print.
int16(16bit fixd point float number)	Not supported. Please convert to half or float to print.
bool	%d
pointer	%p

9.4.2 Host Runtime APIs and Environment Variables Needed for __bang_printf

In order to print MLU formatted data to standard output, you need to call cnrtSyncQueue or cnrtDestroyQueue. These two functions will print MLU formatted data from the cnrtQueue parameter. If you do not call these functions, no MLU data will be printed.

Maximum number of MLU print data per task is determined by environment variable CNRT_BANG C_PRINTF_LIMIT, default value is 1024. If actuall number of MLU print data is more than CNRT_BANG C_PRINTF_LIMIT, only the newest CNRT_BANG C_PRINTF_LIMIT number of MLU print data is kept, and old data is discarded.

9.4.3 Example

1. Kernel example of __bang_printf:

```
__mlu_entry__ void kernel(half* out_data,
  half in1,
  half in2) {
```

```
  __nram__ half nram_out[16];
  for (int i = 0; i < 16; ++i) {
    nram_out[i] = in1 + in2;
    __bang_printf("nram_out[%d]=%.0hf\n", i, nram_out[i]);
  }
  __memcpy(out_data, nram_out, 16 * sizeof(half), NRAM2GDRAM);
}
```

2. Check print result:

The users do not need to set special environment variables, the kernel print information will be output following standards, for example:

```
$ ./a.out
nram_out[0]=2
nram_out[1]=1
nram_out[2]= 3
nram_out[3]=-1
nram_out[4]=102
nram_out[5]=3
nram_out[6]=20
nram_out[7]=-2
nram_out[8]=5
nram_out[9]=20
nram_out[10]=-97
nram_out[11]=6
nram_out[12]=-204
nram_out[13]=-4
nram_out[14]=5
nram_out[15]=25
```

10 Feedback

If you find any errors in the software CNCC, please send an email to compiler@cambricon.com and please provide:

- A source file that caused the error.
- CNCC version number.
- Other environmental information that may cause errors during use.
- A brief description of the problem.

We welcome your suggestions for improvements.