中国科学院大学计算机学院专业选修课

GPU架构与编程

第五课：CUDA编程（四）

赵地
中科院计算所
2025年秋季学期

讲授内容

- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

讲授内容：Related Programming Models: OpenCL

① OpenCL Data Parallelism Model

② OpenCL Device Architecture

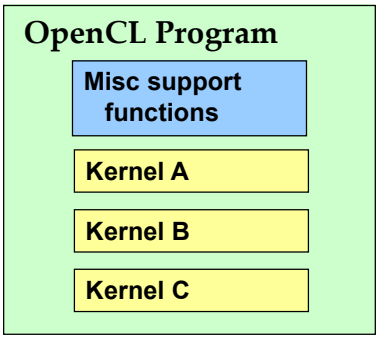
③ OpenCL Host Code

Background

- OpenCL was initiated by Apple and maintained by the Khronos Group (also home of OpenGL) as an industry standard API
  - For cross-platform parallel programming in CPUs, GPUs, DSPs, FPGAs,…
- OpenCL draws heavily on CUDA
  - Easy to learn for CUDA programmers
- OpenCL host code is much more complex and tedious due to desire to maximize portability and to minimize burden on vendors

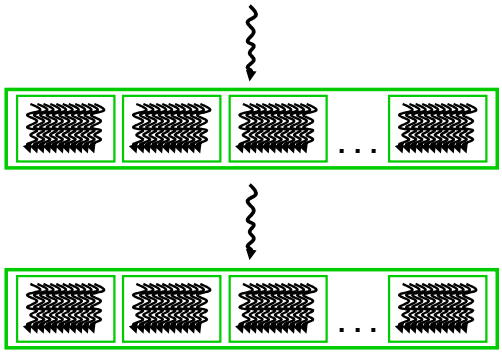
OpenCL Programs

- An OpenCL “program” is a C program that contains one or more “kernels” and any supporting routines that run on a target device
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device



OpenCL Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code



Mapping between OpenCL and CUDA data parallelism model concepts

OpenCL Parallelism Concept	CUDA Equivalent
host	host
device	device
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

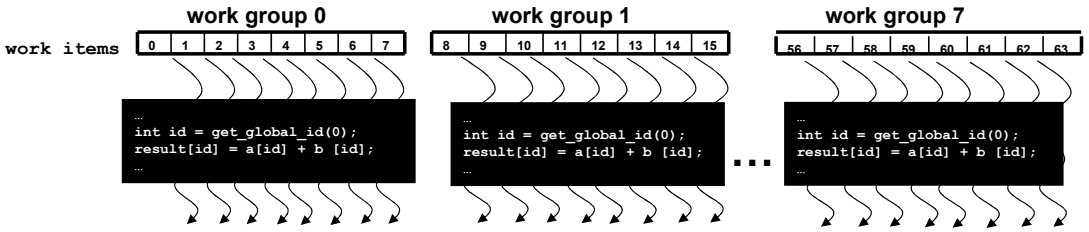
OpenCL Kernels

- Code that executes on target devices
- Kernel body is instantiated once for each work item
  - An OpenCL work item is equivalent to a CUDA thread
- Each OpenCL work item gets a unique index

```
__kernel void  vadd(__global const float *a,
                                  __global const float *b,
                          __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```

Array of Work Items

- An OpenCL kernel is executed by an array of work items
  - All work items run the same code (SPMD)
  - Each work item can call get_global_id() to get its index for computing memory addresses and make control decisions



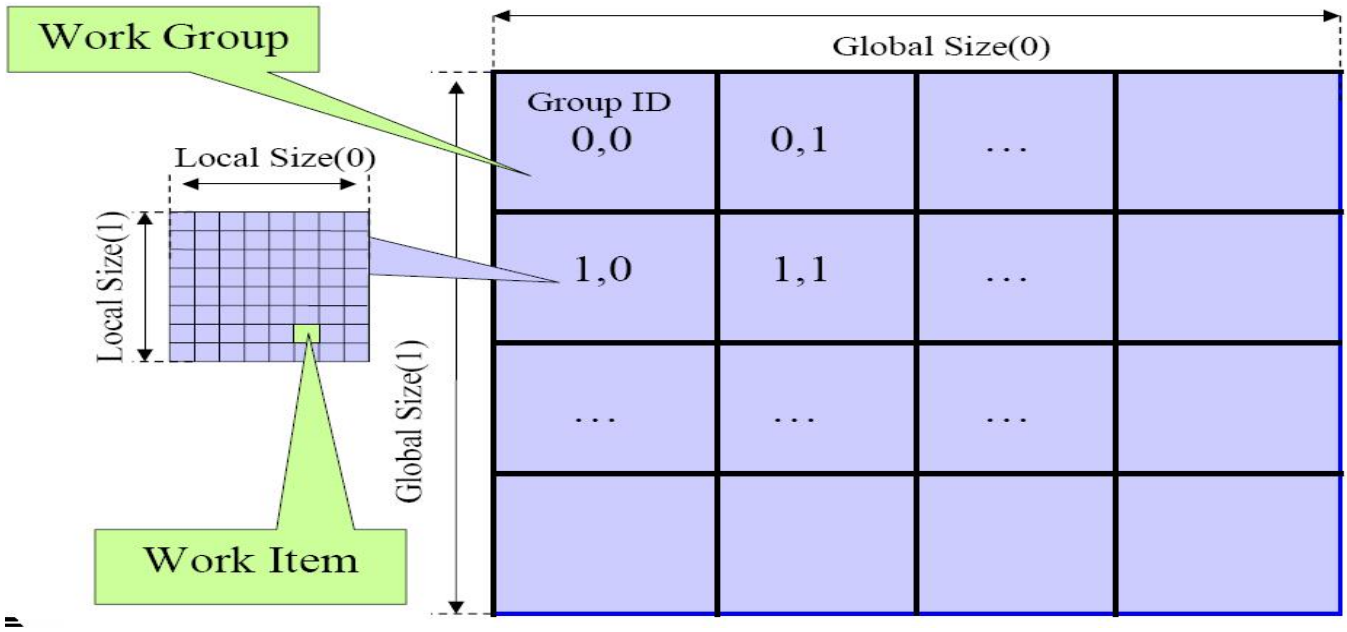
Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via shared memory and barrier synchronization
  - Work items in different work groups cannot cooperate
- OpenCL counter part of CUDA Thread Blocks

OpenCL Dimensions and Indices

OpenCL API Call	Explanation	CUDA Equivalent
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x*blockDim.x+threadIdx.x
get_local_id(0)	local index of the work item within the work group in the x dimension	threadIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x*blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

Multidimensional Work Indexing



OpenCL Data Parallel Model Summary

- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by terminating the kernel

讲授内容：Related Programming Models: OpenCL

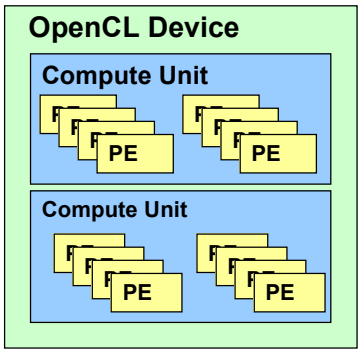
① OpenCL Data Parallelism Model

② OpenCL Device Architecture

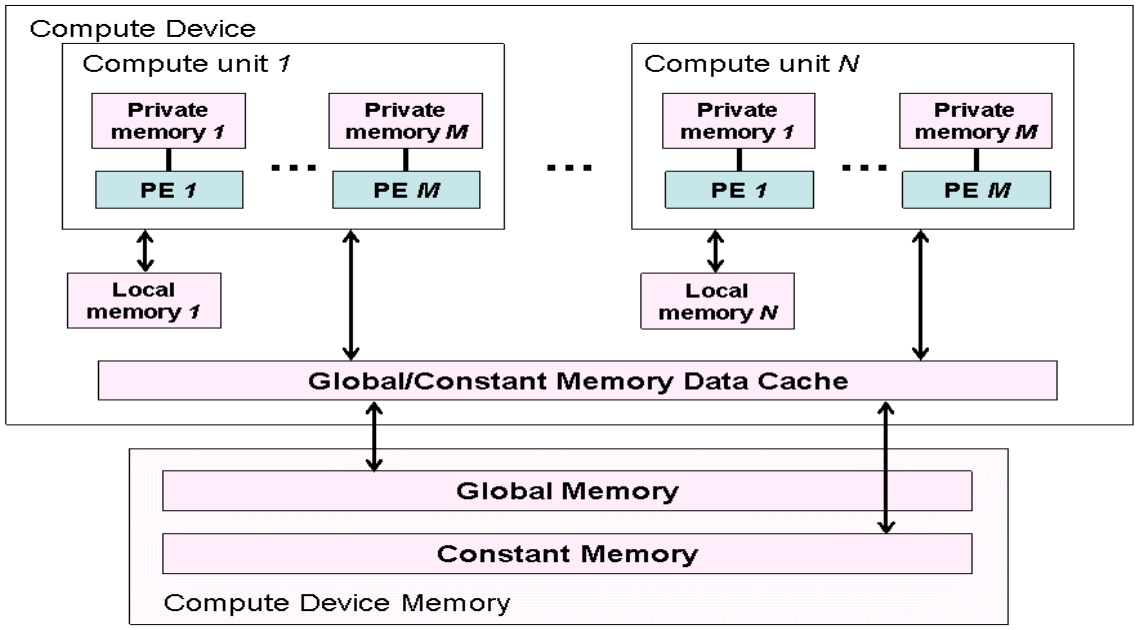
③ OpenCL Host Code

OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each device contains one or more “compute units”, i.e. cores, Streaming Multicprocessors, etc...
- Each compute unit contains one or more SIMD “processing elements”, (i.e. SP in CUDA)



OpenCL Device Architecture

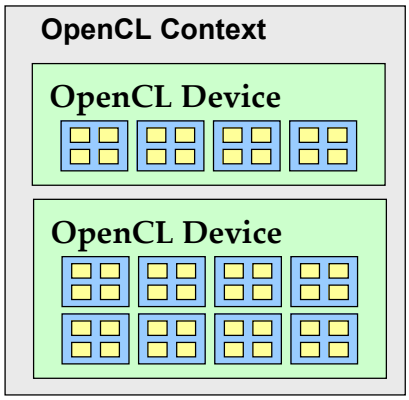


OpenCL Device Memory Types

Memory Type	Host access	Device access	CUDA Equivalent
global memory	Dynamic allocation; Read/write access	No allocation; Read/write access by all work items in all work groups, large and slow but may be cached in some devices.	global memory
constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items.	constant memory
local memory	Dynamic allocation; no access	Static allocation; shared read-write access by all work items in a work group.	shared memory
private memory	No allocation; no access	Static allocation; Read/write access by a single work item.	registers and local memory

OpenCL Context

- Contains one or more devices
- OpenCL device memory objects are associated with a context, not a specific device



讲授内容：Related Programming Models: OpenCL

① OpenCL Data Parallelism Model

② OpenCL Device Architecture

③ OpenCL Host Code

OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory copy transfers are associated with a command queue (thus a specific device)

OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;

cl_context clctx = clCreateContextFromType(0,
CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);


size_t parmsz;

clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL,
&parmsz);


cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);

clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz,
cldevs, NULL);


cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0],
0, &clerr);
```


OpenCL Kernel Compilation: vadd

```
const char* vaddsrc =
   "__kernel void vadd(__global float *d_A, __global float *d_B,
__global float *d_C, int N) { \n"   […etc and so forth…]


cl_program clpgm;

clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc, NULL,
&clerr);


char clcompileflags[4096];

sprintf(clcompileflags, "-cl-mad-enable");

clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL,
NULL);

cl_kernel clkern = clCreateKernel(clpgm, "vadd", &clerr);
```

OpenCL kernel source code as a big string

Gives raw source code string(s) to OpenCL

Set compiler flags, compile source, and retrieve a handle to the “vadd” kernel

OpenCL Device Memory Allocation

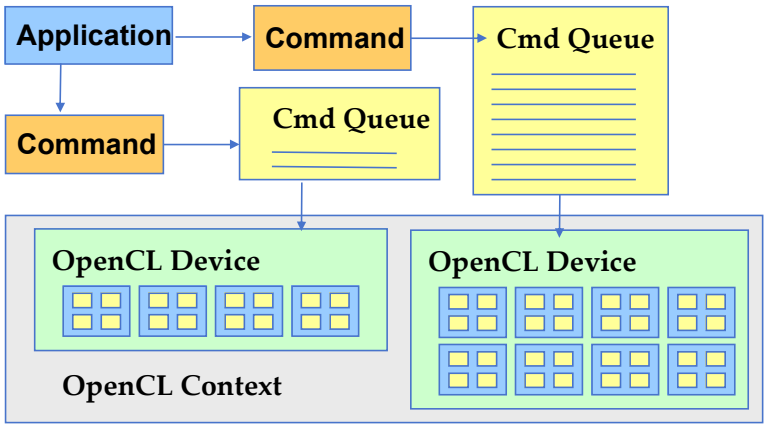
- clCreateBuffer();
  - Allocates object in the device Global Memory
  - Returns a pointer to the object
  - Requires five parameters
    - OpenCL context pointer
    - Flags for access type by device (read/write, etc.)
    - Size of allocated object
    - Host memory pointer, if used in copy-from-host mode
    - Error code
- clReleaseMemObject()
  - Frees object
    - Pointer to freed object

OpenCL Device Memory Allocation (cont.)

- Code example:
  - Allocate a 1024 single precision float array
  - Attach the allocated storage to d_a
  - "d_" is often used to indicate a device data structure

```
VECTOR_SIZE = 1024;
cl_mem d_a;
int size = VECTOR_SIZE* sizeof(float);

d_a = clCreateBuffer(clctx, CL_MEM_READ_ONLY, size, NULL,
  NULL);
…
clReleaseMemObject(d_a);
```

OpenCL Device Command Execution



OpenCL Host-to-Device Data Transfer

- `clEnqueueWriteBuffer();`
  - Memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size (in bytes) of written data
    - Source host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

OpenCL Device-to-Host Data Transfer

- `clEnqueueReadBuffer();`
  - Memory data transfer to host
  - requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

OpenCL Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - a is in host memory and d_a is in device memory

```
clEnqueueWriteBuffer(clcmdq, d_a, CL_FALSE, 0,
       mem_size, (const void * )a, 0, 0, NULL);

clEnqueueReadBuffer(clcmdq, d_result, CL_FALSE, 0,
       mem_size, (void * ) host_result, 0, 0, NULL);
```

OpenCL Host-Device Data Transfer (cont.)

- clCreateBuffer and clEnqueueWriteBuffer can be combined into a single command using special flags.
- Eg:
- d_A=clCreateBuffer(clctxt,
- CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, mem_size, h_A, NULL);
  - Combination of 2 flags here. CL_MEM_COPY_HOST_PTR to be used only if a valid host pointer is specified.
  - This creates a memory buffer on the device, and copies data from h_A into d_A.
  - Includes an implicit clEnqueueWriteBuffer operation, for all devices/command queues tied to the context clctxt.

Device Memory Allocation and Data Transfer for vadd

```
float *h_A = …,   *h_B = …;
    // allocate device (GPU) memory
   cl_mem d_A, d_B, d_C;
   d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A,
NULL);
   d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B,
NULL);
   d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY,
 N *sizeof(float), NULL, NULL);
```

Device Kernel Configuration Setting for vadd

```
clkern=clCreateKernel(clpgm, "vadd", NULL);
  …
  clerr= clSetKernelArg(clkern, 0,
sizeof(cl_mem),(void *)&d_A);
  clerr= clSetKernelArg(clkern, 1,
sizeof(cl_mem),(void *)&d_B);
  clerr= clSetKernelArg(clkern, 2,
sizeof(cl_mem),(void *)&d_C);
  clerr= clSetKernelArg(clkern, 3, sizeof(int),
&N);
```


Device Kernel Launch and Remaining Code for vadd

```
cl_event event=NULL;
clerr= clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL,
   Gsz, Bsz, 0, NULL,  &event);
clerr= clWaitForEvents(1, &event);
clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0,
   N*sizeof(float), h_C, 0, NULL, NULL);
clReleaseMemObject(d_A);
clReleaseMemObject(d_B);
clReleaseMemObject(d_C);
}
```

讲授内容

- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

讲授内容：Related Programming Models: OpenACC

① Introduction to OpenACC

② OpenACC Subtleties

OpenACC

- The OpenACC Application Programming Interface provides a set of
  - compiler directives (pragmas)
  - library routines and
  - environment variables

  that can be used to write data parallel Fortran, C and C++ programs that run on accelerator devices including GPUs and CPUs

OpenACC Pragmas

- In C and C++, the #pragma directive is the method to provide to the compiler information that is not specified in the standard language.
  - These pragmas extend the base language

Vector Addition in OpenACC

```
void VecAdd(float * __restrict__ output, const
     float * input1, const float * input 2, int
     inputLength)

{

 #pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLengt
h]),  copyout(output[0:inputLength])

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

        output[i] = input1[i] + input2[i];

    }

}
```


Simple Matrix-Matrix Multiplication in OpenACC

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.      for (int j=0; j<Nw; j++) {
7.          float sum = 0;
8.          for (int k=0; k<Mw; k++) {
9.               float a = M[i*Mw+k];
10.               float b = N[k*Nw+j];
11.               sum += a*b;
12.          }
13.          P[i*Nw+j] = sum;
14.      }
15.  }
16. }
```

Some Observations (1)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.      for (int j=0; j<Nw; j++) {
7.          float sum = 0;
8.          for (int k=0; k<Mw; k++) {
9.              float a = M[i*Mw+k];
10.              float b = N[k*Nw+j];
11.              sum += a*b;
12.          }
13.          P[i*Nw+j] = sum;
14.      }
15.  }
16. }
```

The code is almost identical to the sequential version, except for the two lines with #pragma at line 3 and line 5.

Some Observations (2)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.      for (int j=0; j<Nw; j++) {
7.          float sum = 0;
8.          for (int k=0; k<Mw; k++) {
9.              float a = M[i*Mw+k];
10.              float b = N[k*Nw+j];
11.              sum += a*b;
12.          }
13.          P[i*Nw+j] = sum;
14.      }
15.  }
16. }
```

The #pragma at line 3 tells the compiler to generate code for the ‘i’ loop at line 4 through 15 so that the loop iterations are executed at the first level of parallelism on the accelerator.

Some Observations (3)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.      for (int j=0; j<Nw; j++) {
7.          float sum = 0;
8.          for (int k=0; k<Mw; k++) {
9.              float a = M[i*Mw+k];
10.             float b = N[k*Nw+j];
11.             sum += a*b;
12.         }
13.         P[i*Nw+j] = sum;
14.     }
15. }
16. }
```

The copyin() clause and the copyout() clause specify how the compiler should arrange for the matrix data to be transferred between the host and the accelerator.

Some Observations (4)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw])
    copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.      for (int j=0; j<Nw; j++) {
7.          float sum = 0;
8.          for (int k=0; k<Mw; k++) {
9.              float a = M[i*Mw+k];
10.             float b = N[k*Nw+j];
11.             sum += a*b;
12.         }
13.         P[i*Nw+j] = sum;
14.     }
15. }
16. }
```

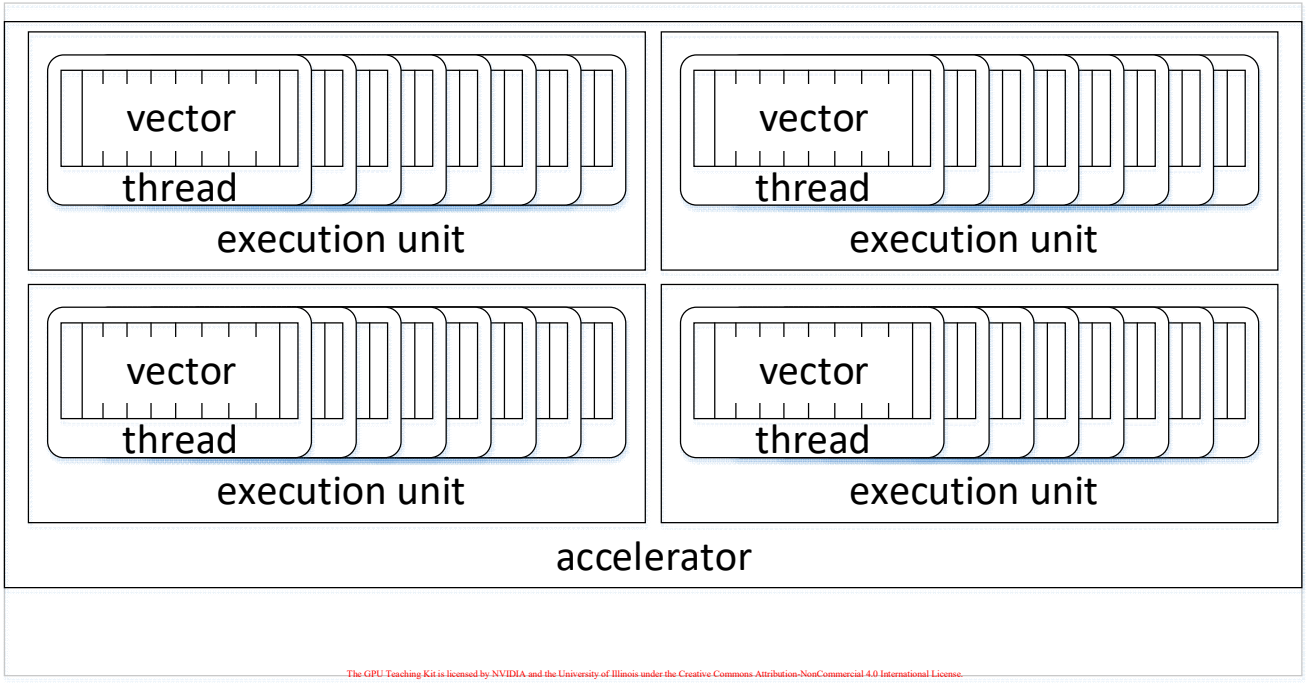
Motivation

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  - leave most of the details in generating a kernel, memory allocation, and data transfers to the OpenACC compiler.
- OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

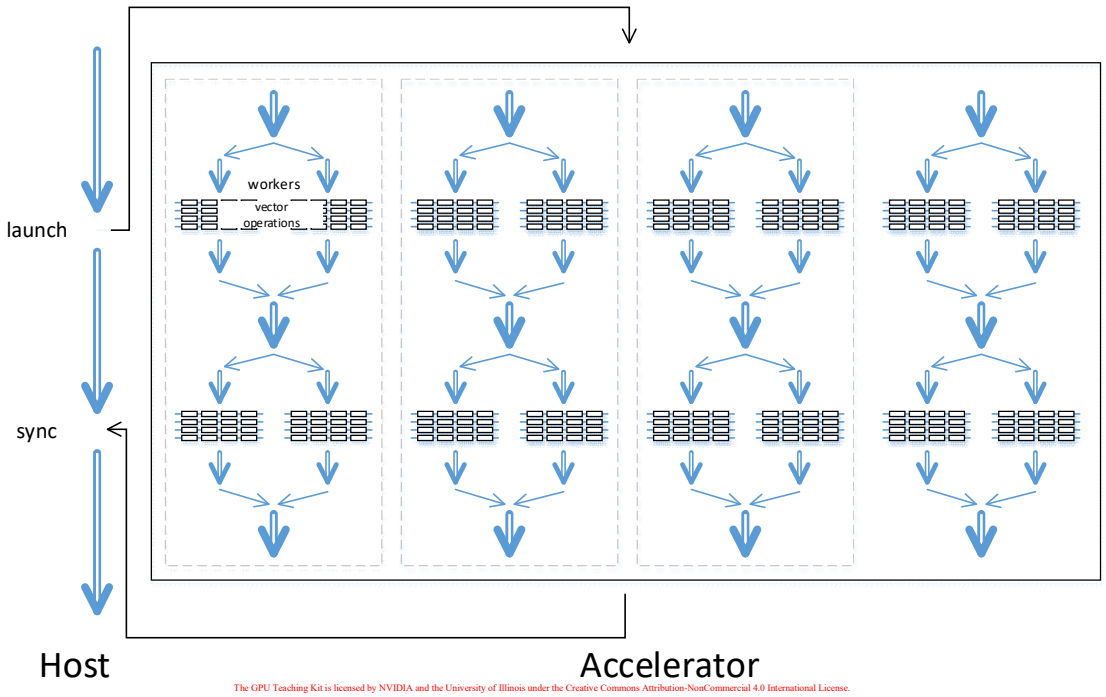
Frequently Encountered Issues

- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
  - The uncertainty is much less so for CUDA or OpenCL programs

OpenACC Device Model



OpenACC Execution Model



讲授内容：Related Programming Models: OpenACC

① Introduction to OpenACC

② OpenACC Subtleties

Parallel vs. Loop Constructs

```
#pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw])
copyout(P[0:Mh*Nw])
for (int i=0; i<Mh; i++) {
...
}
```

is equivalent to:

```
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
{
    #pragma acc loop
    for (int i=0; i<Mh; i++) {
        ...
    }
}
```

(a parallel region that consists of a single loop)

More on Parallel Construct

```
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
    a = 23;
}
```

1024*32 workers will be created. a=23 will be executed redundantly by all 1024 gang leads

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of workers in each gang
  - Equivalent to CUDA blocks and threads

What Does Each "Gang Loop" Do?

```
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        …
    }
}
```

```
#pragma acc parallel
num_gangs(1024)
{
#pragma acc loop gang
    for (int i=0; i<2048; i++) {
        …
    }
}
```

Worker Loop

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
      #pragma acc loop gang
      for (int i=0; i<2048; i++) {
            #pragma acc loop worker
            for (int j=0; j<512; j++) {
                  foo(i,j);
            }
      }
}
```

1024*32=32K workers will be created, each executing 1M/32K = 32 instance of foo()

A More Substantial Example

- Statements 1, 3, 5, 6 are redundantly executed by 32 gangs

```
#pragma acc parallel num_gangs(32)
{
   Statement 1;
   #pragma acc loop gang
   for (int i=0; i<n; i++) {
      Statement 2;
   }
。。。
}
```

```
#pragma acc parallel num_gangs(32)
{
   。。。
    Statement 3;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
       Statement 4;
    }
    Statement 5;
    if (condition) Statement 6;
}
```

A More Substantial Example

- The iterations of the n and m for-loop iterations are distributed to 32 gangs
- Each gang could further distribute the iterations to its workers
  - The number of workers in each gang will be determined by the compiler/runtime

A More Substantial Example

```
#pragma acc parallel
num_gangs(32)
{
    Statement 1;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 2;
    }
. . .
}
```

```
#pragma acc parallel num_gangs(32)
{
. . .
    Statement 3;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 4;
    }
    Statement 5;
    if (condition) Statement 6;
}
```

Avoiding Redundant Execution

- Statements 1, 3, 5, 6 will be executed only once
- Iterations of the n and m loops will be distributed to 32 workers

```
#pragma acc parallel num_gangs(1) num_workers(32)
{
    Statement 1;
    #pragma acc loop worker
    for (int i=0; i<n; i++) {
        Statement 2;
    }
    Statement 3;
    #pragma acc loop worker
    for (int i=0; i<m; i++) {
        Statement 4;
  }
  Statement 5;
   if (condition)  Statement 6;
}
```


Kernel Regions

- Kernel constructs are descriptive of programmer intentions
  - The compiler has a lot of flexibility in its use of the information
- This is in contrast with Parallel, which is prescriptive of the action for the compile follow

```
#pragma acc kernels
{
    #pragma acc loop gang(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop gang(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }
    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

Kernel Regions

- Code in a kernel region can be broken into multiple CUDA/OpenCL kernels
- The i, j, k loops can each become a kernel
  - The k-loop may even remain as host code
- Each kernel can have a different gang/worker configuration

```
#pragma acc kernels
{
    #pragma acc loop gang(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop gang(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }
    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

讲授内容

- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

讲授内容：Multi-GPU: OpenMP

① OpenMP

② Multi GPU Introduction I

③ Multi GPU Introduction II

④ Multi GPU patterns with OpenMP & Cooperative Groups

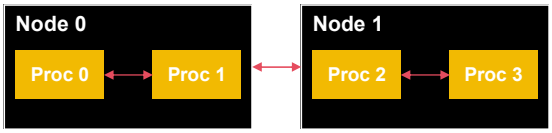
OpenMP

- A parallel programming model based on the concepts of multithreading and shared memory
- OpenMP programs consists on annotations written into the serial code, called directives
- Needs a compiler with OpenMP support, which will translate the directives to the actual parallel code
- Highly portable across systems
- Limited to a single computer, however it can handle several processors

Differences between process and threaded parallelism

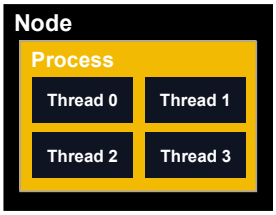
Process parallelism

- Uses local memory forcing each parallel worker to have its own memory space.
  - It needs explicit data sharing routines creating communication overhead.
- It's controlled by a main process.
- It can scale to multiple computers.
- Example: MPI (see Module 18)



Threaded parallelism

- Uses shared memory, enabling all parallel workers to access the same memory space.
- It's controlled by a main thread, with all threads running on a single process.
- It can't scale beyond a single computer.
- Example: OpenMP



OpenMP parallel regions

- OpenMP is a directive based parallel programming model, meaning we annotate the code to introduce parallelism.
- The directive to create a parallel region is:
  - #pragma omp parallel
  - This pragma is added right before the block of code to parallelize.
  - This pragma runs the instructions inside the code block in each parallel thread.

–Example parallel behavior of calling a function inside a parallel region:

```
#pragma omp parallel
{
   foo();
}
```

Visualization of the behavior of the parallel region

Parallelizing a “for” loop with OpenMP

- OpenMP is aware that in many cases “for” loops are a source for introducing parallelism, that is why it has special directives for loops
- To parallelize a for loop exists the directive:
  - #pragma omp parallel for
  - This pragma is added right before the for loop to parallelize.
  - An OpenMP compiler will identify the pragma and create a program where the loop is executed in parallel.

–Serial version of the vector sum algorithm:

```
for(int i = 0; i < n; ++i)  {
  c[i] = a[i] + b[i];
}
```

–Parallel OpenMP version of the vector sum algorithm:

```
#pragma omp parallel for
for(int i = 0; i < n; ++i)  {
  c[i] = a[i] + b[i];
}
```


#pragma omp parallel for

- Each variable used inside the loop has a kind determining if it’s accessible by other threads: shared, private, firstprivate, lastprivate
- Shared variables are accessible by every thread in the region
- Private variables are local to each thread and are not initialized
- Firstprivate behave similarly to private variables but are initialized with the value before the parallel region
- Unless otherwise stated, every variable outside the loop will be considered shared and every inside the loop will be considered private

OpenMP Synchronization

- By default every parallel region will wait until the region is finished before executing other instructions. To change this behavior use the nowait option when declaring the parallel region
- #pragma omp barrier specifies a synchronization point equivalent to __syncthreads() in CUDA, the pragma is not associated to a block of code

```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
  ...
#pragma omp barrier
  c[i] += i >= 1 ? c[i - 1] : 0;
}
```


OpenMP Sections

- Critical sections:
  - Are executed by only one thread at a time
  - Are associated to a block of code
  - #pragma omp critical
- Single sections:
  - Are executed by only one thread of the team
  - Are associated to a block of code
  - #pragma omp single

```
int a = 0;
int b = 0;
#pragma omp parallel num_threads(4)
{
#pragma omp critical
  a += 1 ;
#pragma omp single
  b += 1;
}
```

- a final value is 4, because each thread executed the section one at a time
- b final value is 1, because only one thread executed the section

OpenMP API

- OpenMP in addition to the directives have an API, to control or obtain certain runtime parameters.
- As is the case with CUDA, OpenMP allows users to set the number of threads, get the number of threads and uniquely identify each thread.

- omp_set_num_threads(int):
  - Sets the desired number of threads to be used by the OpenMP runtime on subsequent parallel regions.
- omp_get_num_threads():
  - Returns the number of threads being used in the parallel region.
- omp_get_thread_num():
  - Returns a numeric identifier for the calling thread, with different threads having different identifiers.

Matrix Vector Multiplication OpenMP example

```
void tileMultiply(double* A, double* x, double y, int rows, int cols, int tileSize){
    int rid = omp_get_thread_num() * tileSize;              // Get the first row in the tile to compute
    for(int r = rid; r < min(rows, rid + tileSize); ++r ) {   // Iterate through the rows in the tile
        double sum = 0;
        for(int c = 0; c < cols; ++c)
            sum += A[r * cols + c] * x[c];                  // Accumulate the dot product between r-
row and x
        y[r] = sum;                                                 // Store the result into the
result
    }
}
…
#pragma omp parallel                                        // Create a parallel region
{
    int tnum = omp_get_num_threads();               // Get the number of threads executing the region
    int tileSize = (rows + tnum - 1) / tnum;          // Calculate the number of rows in a tile
    tileMultiply(A, x, t, rows, cols, tileSize);         // Dispatch the function calculating the
multiplication
}
```

Compiling an OpenMP program

- OpenMP compilers typically will ignore the OpenMP pragmas unless we specify the OpenMP flag.
- To use the API functions you need to include in the appropriate file:
  - #include <omp.h>
- Examples:
  - GCC & Clang: to compile an OpenMP annotated source file we use:
    - gcc <file to compile> -fopenmp
    - clang <file to compile> -fopenmp
    - Additionally if there are multiple OpenMP libraries, you may specify which version to use, for example –lgomp uses the GNU implementation.
  - NVCC: to compile an OpenMP annotated source file we use:
    - nvcc <options> -Xcompiler -fopenmp

讲授内容：Multi-GPU: OpenMP

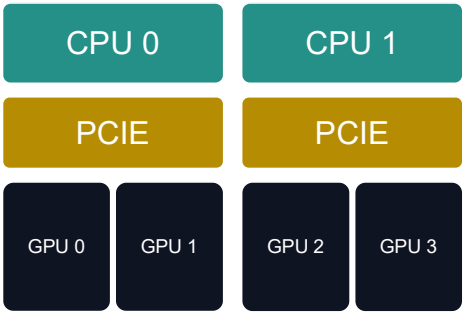
① OpenMP

② Multi GPU Introduction I

③ Multi GPU Introduction II

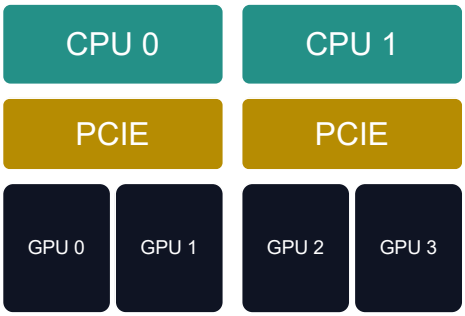
④ Multi GPU patterns with OpenMP & Cooperative Groups

Schematic of a Multi GPU system



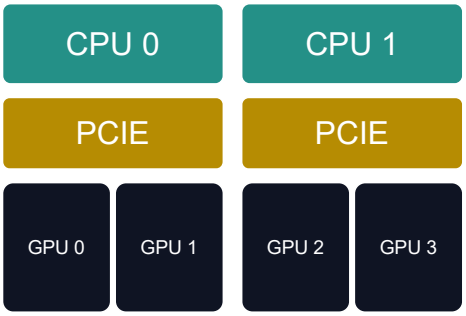
- The figure represents a system with 2 CPU's and 4 GPU's.
- GPU's are numbered from 0 to n-1, where n is the number of GPU's.
- The CUDA driver always starts with a default active device.
- There are two broad types of Multi GPU communication:
  - Through the PCIE bus
  - Through NVLINK

CUDA host API calls for Multi GPU's



- cudaSetDevice()
  - Set GPU device to use for device code execution on the active host thread.
  - Requires one parameter:
    - An int with the device id number
  - This function doesn't affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn't affect previous async calls.

CUDA host API calls for Multi GPU's



- cudaGetDevice()
  - Get GPU device being currently used by the active host thread.
  - Requires one parameter:
    - An int pointer to store the device id
- cudaGetDeviceCount()
  - Get the number of CUDA-capable devices in the system.
  - Requires one parameter:
    - An int pointer to store the device count

CUDA host API calls for Memory allocation with Multiple GPU's

To allocate or associate memory with a specific device using non-Managed CUDA-API calls, it's necessary to call cudaSetDevice() before doing the allocation call.

- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object in terms of bytes
- cudaHostAlloc()
  - Allocates pinned memory on the host
  - Three parameters
    - Address of pointer to the allocated memory
    - Size of the allocated memory in bytes
    - Host Alloc flags

CUDA host API calls for Memory allocation with Multiple GPU's Unified Memory

- If the flag cudaDevAttrConcurrentManagedAccess is set in all devices, then it's not necessary to call cudaSetDevice before the cudaMallocManaged call.
- If the flag is not set but devices can access each others memory, then calling cudaSetDevice before the cudaMallocManaged call will establish the context for the managed memory on the active device.
  - With other devices accessing the data via PCIE at reduced bandwidth.

CUDA runtime calls affected by cudaSetDevice

- If cudaSetDevice() was called before a kernel launching call, the kernel will execute in the active device.
  - It's crucial that every non managed memory being used in the kernel resides in the active device, otherwise an error will occur.
- If cudaSetDevice() was called before a cudaStreamCreate(), then the stream will be associated with the active device.
- The synchronization functions: cudaDeviceSynchronize(), cudaStreamSynchronize() are also affected by cudaSetDevice(), synchronizing tasks only for the active device on the active host thread

Putting it all together, vecAdd

```
    float *m_A0, float *m_B0, *m_A1, float *m_B1, int n;
    int size = n * sizeof(float);
    cudaSetDevice(0);                                          // Will set the active device to 0
    cudaMalloc((void**) &m_A0, size);                  // Will allocate memory on device 0
    cudaMalloc((void**) &m_B0, size);                  // Will allocate memory on device 0
    cudaSetDevice(1);                                          // Will set the active device to 1
    cudaMalloc((void**) &m_A1, size);                  // Will allocate memory on device 1
    cudaMalloc((void**) &m_B1, size);                  // Will allocate memory on device 1


    // Memory initialization on the Host and memory transfers
   cudaSetDevice(0);                                           // Set the device for kernel
execution
   vecAdd<<<gridDim, blockDim>>>(m_A0,m_B0);


   cudaSetDevice(1);                                           // Set the device for kernel
execution
   vecAdd<<< gridDim, blockDim>>>(m_A1,m_B1);


    cudaFree(m_A0); cudaFree(m_B0);
    cudaFree(m_A1); cudaFree(m_B1);
```


讲授内容：Multi-GPU: OpenMP

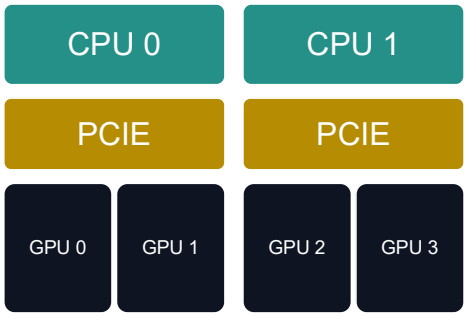
① OpenMP

② Multi GPU Introduction I

③ Multi GPU Introduction II

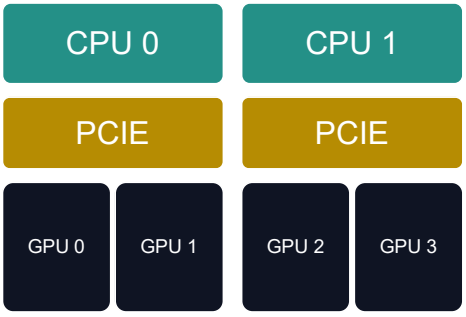
④ Multi GPU patterns with OpenMP & Cooperative Groups

Memory transfers in a Multi GPU setup



- Involves transferring memory regions from one device to another, e.g. GPU0 to GPU1.
- There are three ways to do it:
  - Fully explicit memory transfers using cudaMemcpyPeerAsync, which requires the specification of the peer devices.
  - Partially explicit memory transfers using cudaMemcpy, relying on the unified address system.
  - Implicit peer memory access performed by the driver, without the need of explicit transfers.
- Not all three possibilities are available in every system.

Explicit peer memory transfers CUDA host API functions



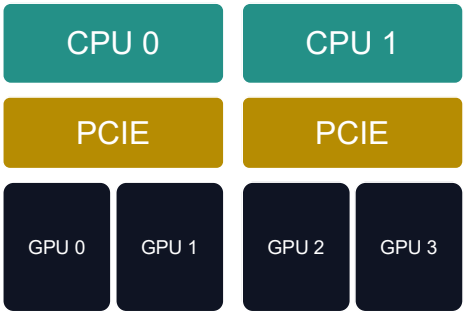
- cudaMemcpyPeerAsync()
  - Six parameters
    - Pointer to destination region on the destination device
    - Destination device id
    - Pointer to source region on the source device
    - Source device id
    - Number of bytes copied
    - CUDA stream

  Transfer between devices is asynchronous

Example: peer transfer cudaMemcpyPeerAsync

```
float *A0, *A1;
int size;

cudaSetDevice(0);  // Set active device to 0
cudaMalloc((void**) &A0, size);// Allocate memory on device 0
cudaSetDevice(1); // Set active device to 1
cudaMalloc((void**) &A1, size);  // Allocate memory on device 1

 // Initialize region A0 on device 0
cudaMemcpyPeerAsync(A1, 1, A0, 0, size, stream); // Copy the data on A0 on device 0 to the region
A1 on device 1

cudaSetDevice(1);  // Set the device for kernel execution
kernel<<<gridDim, blockDim, 0, stream>>>(A1); // Perform computations on A1

cudaFree(A0);  // Free A0 region
cudaFree(A1);  // Free A1 region
```

Explicit peer memory transfers CUDA host API functions

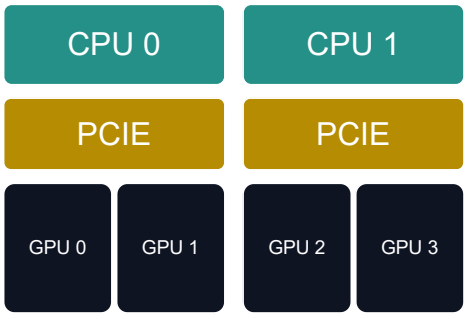


- If the flag cudaDevAttrUnifiedAddressing is set to 1, then you may copy regions between devices using the traditional cudaMemcpy API function, setting the copy kind to cudaMemcpyDefault.
- To check if the flag is set you can use the API function:
  - cudaDeviceGetAttribute()

Example: peer transfer cudaMemcpy

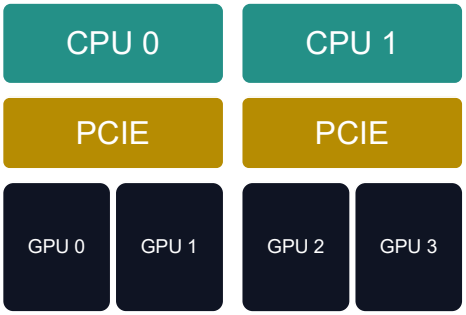
```
float *A0, *A1;

int size;

int unifiedAddr_flag0 = 0;

int unifiedAddr_flag1 = 0;
cudaSetDevice(0);// Set active device to 0
cudaMalloc((void**) &A0, size);// Allocate memory on device 0
cudaSetDevice(1);// Set active device to 1

cudaMalloc((void**) &A1, size);// Allocate memory on device 1

// Initialize region A0 on device 0

cudaDeviceGetAttribute(unifiedAddr_flag0, cudaDevAttrUnifiedAddressing, 0); // Check if unified
addressing is available on dev 0

cudaDeviceGetAttribute(unifiedAddr_flag1, cudaDevAttrUnifiedAddressing, 1); // Check if unified
addressing is available on dev 0

if( unified_addressing_flag0 == 1 && unified_addressing_flag1 == 1 )

 cudaMemcpy(A1, A0, size, cudaMemcpyDefault);  // Copy the data on A0 on device 0 to the region A1 on
device 1

else

 // Throw error indicating the copy couldn't be performed
```

Implicit peer memory access



- To query if implicit peer memory access are enabled use:
- cudaDeviceCanAccessPeer:
  - Three parameters:
    - Int pointer to place to store the flag.
    - Device id of device trying to access peer
    - Device id of peer device
  - This call is not symmetric, meaning that if the canAccess flag is set to 1 for deviceA and deviceB, it may not be set to 1 for deviceB and deviceA.

Enabling and disableing implicit peer memory access



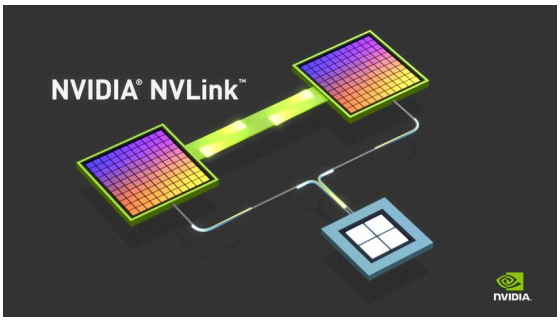
- cudaDeviceEnablePeerAccess:
  - Two parameters:
    - Device id to enable access to from the current active device.
    - Int flag set to 0.
  - This call is not symmetric.
  - Returns error cudaErrorInvalidDevice if not possible.
- cudaDeviceDisablePeerAccess:
  - One parameter:
    - Device id to disable access to from the current device.
  - This call is not symmetric.

Example: implicit peer access

```
float *ptrA;  // Pointer to memory region on device devA
int devA;
int devB;
int BcanAccessA = 0;
cudaError_t error;
cudaDeviceCanAccessPeer(&BcanAccessA, devB, devA);  // Check if devB can access devA memory.
cudaSetDevice(devB);  // Set the current active device to devB
if(BcanAccessA == 0)
 error = cudaDeviceEnablePeerAccess(devA, 0);  // Enable peer accesses to devA memory
if(error == cudaSuccess) {
  kernel<<<gridDim, blockDim, 0, stream>>>(ptrA);  // Access ptrA on device devA from device devB
}
cudaDeviceDisablePeerAccess(devA); // Disable peer access to devA, this call is not needed.
```

NVLINK

- Is a proprietary interconnect technology developed by NVIDIA
- Provides higher memory bandwidth communication between GPUS than PCIE communication
- NVLINK on the Tesla V100 delivers a 300 GB/s communication data rate, whereas the typical PCIE 3.0 link delivers only 32 GB/s



讲授内容：Multi-GPU: OpenMP

① OpenMP

② Multi GPU Introduction I

③ Multi GPU Introduction II

④ Multi GPU patterns with OpenMP & Cooperative Groups

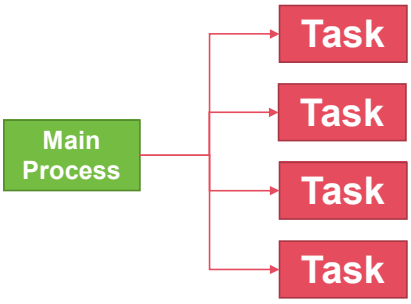
Common parallel patterns in a Multi-GPU environment

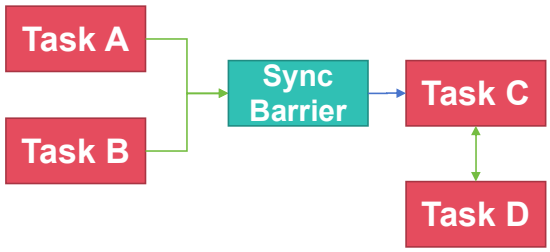
Batch processing:

- Execute the same independent task multiple times with different data.

Cooperative patterns:

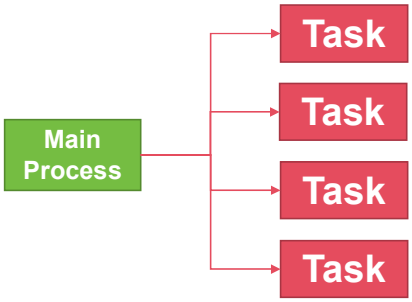
- Tasks need to cooperate between each other to collectively reach a goal.





Batch processing

- Is an embarrassingly parallel pattern.
- With enough data it is usual we can achieve 100% usage of the compute resources.
- It's common on video and image processing applications, where we need to apply the same operation to lots of different data.

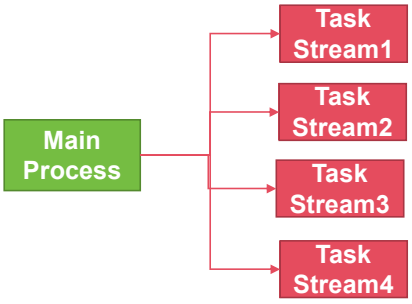


Batch processing

Typical operations to accomplish batch processing:

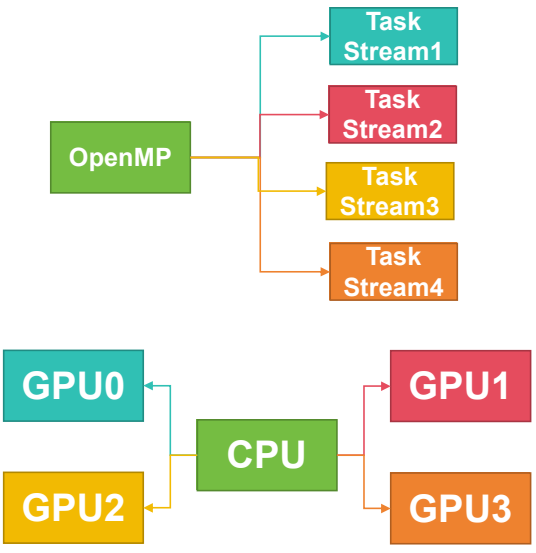
1. Get number of available devices.
2. Considering the number of devices and number of desired tasks allocate, initialize and copy the memory need it by the algorithm.
3. Create CUDA streams for each of the tasks to be executed concurrently.
4. Launch in parallel the kernel.

* Remember to set the device at the beginning of each group of operations.



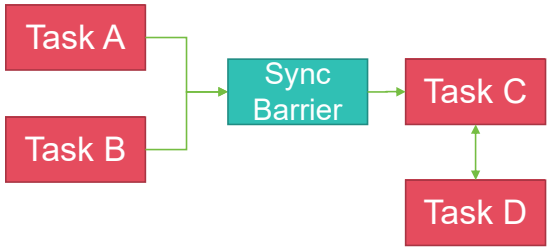
Batch processing example with OpenMP

```
  int deviceCount;
  cudaGetDeviceCount(& deviceCount);
  std::vector<cudaStream_t> streams(deviceCount);
#pragma omp parallel for num_threads(deviceCount)
  for(int dev = 0; dev < deviceCount; ++dev) {
      cudaSetDevice(dev);
      cudaStreamCreate(&streams[i]);
      // Allocate, initialize and transfer memory
  }
#pragma omp parallel for num_threads(deviceCount)
  for(int dev = 0; dev < deviceCount; ++dev) {
      cudaSetDevice(dev);
      kernel<<<gridDim, blockDim,
streams[i]>>>(…);
  }
```



Cooperative patterns

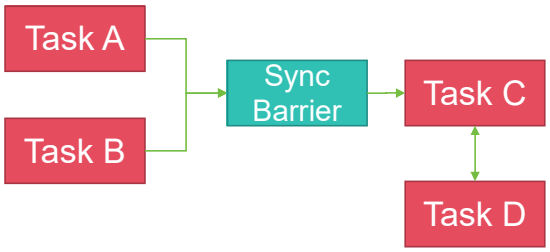
- Might have unavoidable syncing points causing tasks to wait and thus wasting compute resources.
- In some cases even when massive amounts of input data it might not reach 100% resource usage.
- It's common on applications with steps to reach a goal like iterative algorithms.



Cooperative patterns

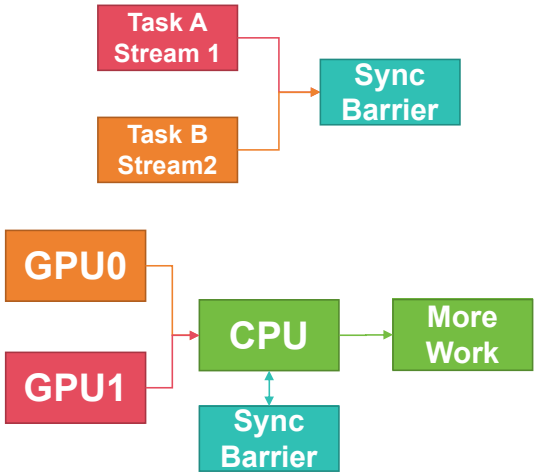
- With cooperative patterns there is no single fit solution like with batch processing.
- Thus the process consists in a loop of:

1. Launching the code in parallel.
2. Profiling it.
3. Analyzing and removing bottlenecks.



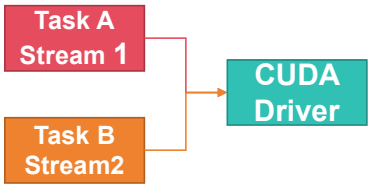
Syncing patterns on different streams with OpenMP

```
  std::vector<cudaStream_t> streams;
  // Initialization of the streams on each device.
#pragma omp parallel
{
  // Launch the different kernels on the streams.
#pragma omp for num_threads(streams.size())
  for(auto& stream : streams)
      cudaStreamSynchornize(stream);
#pragma omp barrier
}
```



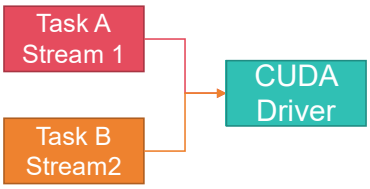
Multi-GPU Syncing patterns with Cooperative Groups

- Cooperative Groups is a C++-CUDA high level abstraction to perform syncing across different parallel granularities (Threads, Blocks, Grids, and Devices).
- Multi-GPU syncing with cooperatives groups requires:
  - Devices with the exact same compute capability.
  - Compute capability of 6 or higher.
  - Executing the same kernel across all devices.



Multi-GPU Syncing patterns with Cooperative Groups

- To enable the use of Cooperative Groups we need to include the file cooperative_groups.h and use the namespace cooperative_groups.
- The kernels needs to be compiled using separate compilation and then linked with the –rdc=true flag.
- You also need to ensure that MPS is disabled and CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH is set in the device properties using cuDeviceGetAttribute the API function.



Multi-GPU Syncing patterns with Cooperative Groups

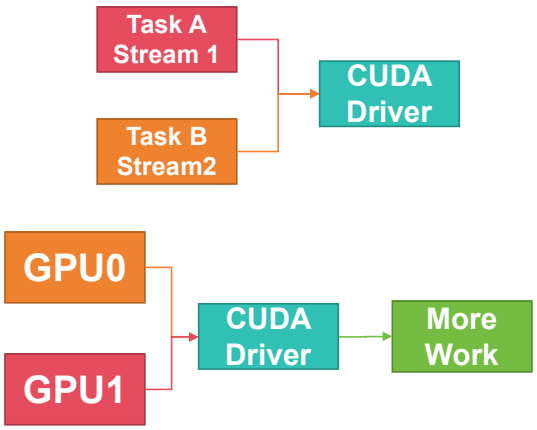
Launching a kernel with Multi-GPU syncing and Cooperative Groups requires using the API function:

- cudaLaunchCooperativeKernelMultiDevice:
  - The first parameter is an array of CUDA_LAUNCH_PARAMS, where except for kernel params and the stream, all other fields needs be the same.
  - The number of devices to use.

```
typedef struct CUDA_LAUNCH_PARAMS_st {
  CUfunction function;       // Kernel to launch.
  unsigned int gridDimX;    // Grid dimensions.
  unsigned int gridDimY;
  unsigned int gridDimZ;
  unsigned int blockDimX; // Block dimensions.
  unsigned int blockDimY;
  unsigned int blockDimZ;
  unsigned int sharedMemBytes; // Shared memory
size.
  CUstream hStream;          // Stream to perform
the work
  void **kernelParams;      // Kernel parameters
} CUDA_LAUNCH_PARAMS;
```

Multi-GPU Syncing patterns with Cooperative Groups

```
#include <cooperative_groups.h>
using namespace cooperative_groups;
void __global__ kernel(...) {
    // Work
    multi_grid_group multi_grid =
this_multi_grid();
    multi_grid.sync();
   // Work
}
//  Work
cudaLaunchCooperativeKernelMultiDevice
(...);
```



讲授内容

- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

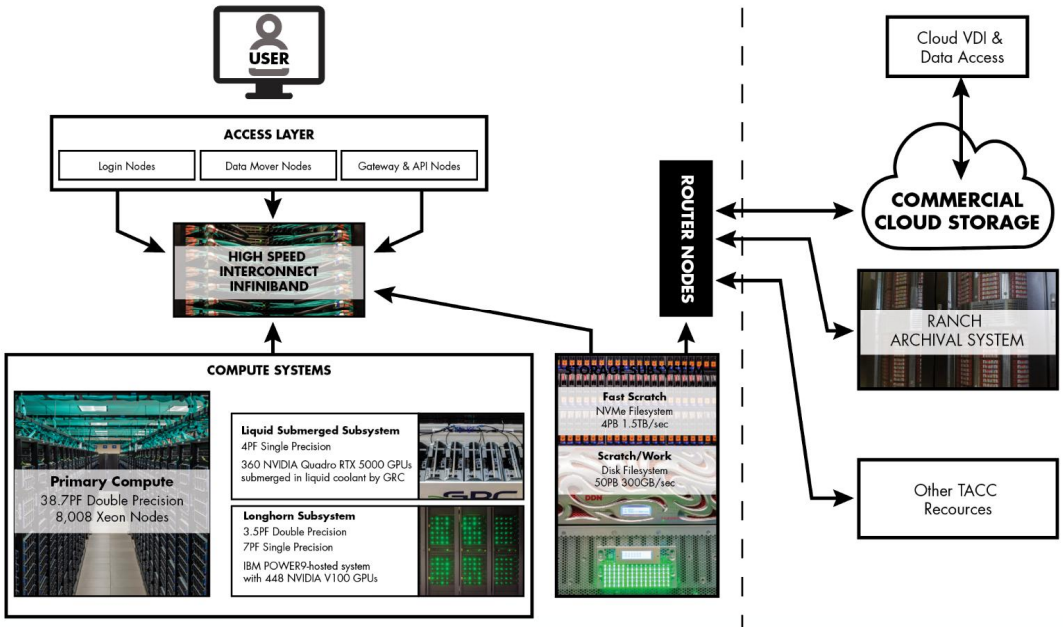
讲授内容：Related Programming Models: MPI

① Warps and SIMD Hardware

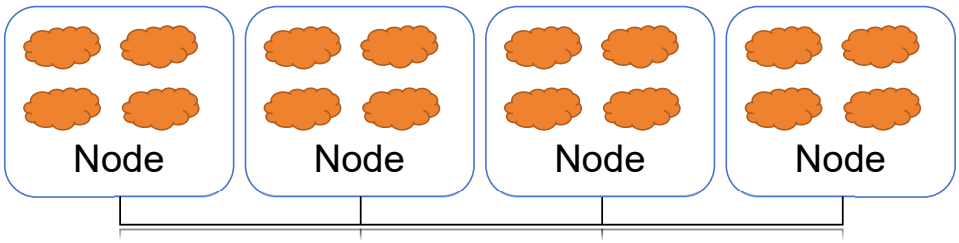
② Performance Impact of Control Divergence

③ Overlapping Computation with Communication

Frontera – Installed at TACC 9/2019



MPI – Programming and Execution Model



Many processes distributed in a cluster

Each process computes part of the output

Processes communicate with each other

Processes can synchronize

MPI Initialization, Info and Sync

```
int MPI_Init(int *argc, char ***argv)
```

Initialize MPI

```
MPI_COMM_WORLD
```

MPI group with all allocated nodes

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Rank of the calling process in group of comm

```
int MPI_Comm_size (MPI_Comm comm, int *size)
```

Number of processes in the group of comm

Vector Addition: Main Process

```
int main(int argc, char *argv[]) {
  int vector_size = 1024 * 1024 * 1024;
  int pid=-1, np=-1;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if(np < 3) {
    if(0 == pid) printf("Need 3 or more processes.\n");
    MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
  }
```

Vector Addition: Main Process

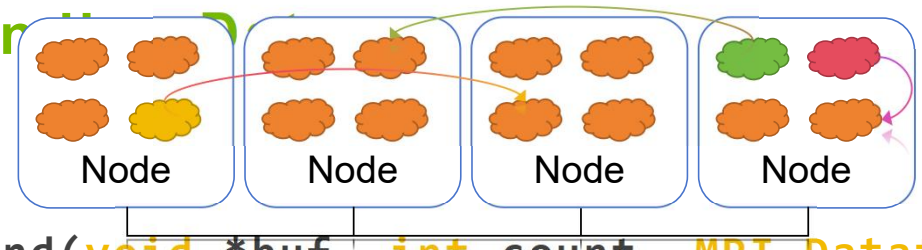
```
  if(pid < np - 1)
    compute_node(vector_size / (np - 1));
  else
    data_server(vector_size);

  MPI_Finalize();
  return 0;
}
```

MPI Sending Data

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

- **Buf**: Initial address of send buffer (choice)
- **Count**: Number of elements in send buffer (nonnegative integer)
- **Datatype**: Datatype of each send buffer element (handle)
- **Dest**: Rank of destination (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)

MPI Sending Data




```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

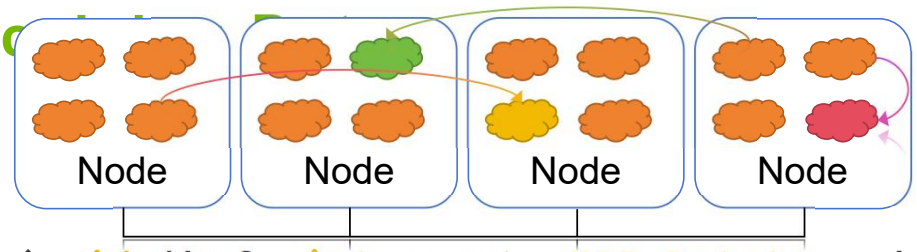
- **Buf**: Initial address of send buffer (choice)
- **Count**: Number of elements in send buffer (nonnegative integer)
- **Datatype**: Datatype of each send buffer element (handle)
- **Dest**: Rank of destination (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)

MPI Receiving Data

```
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

- **Buf**: Initial address of receive buffer (choice)
- **Count**: Maximum number of elements in receive buffer (integer)
- **Datatype**: Datatype of each receive buffer element (handle)
- **Source**: Rank of source (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)
- **Status**: Status object (Status)

MPI Rec




```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- **Buf**: Initial address of receive buffer (choice)
- **Count**: Maximum number of elements in receive buffer (integer)
- **Datatype**: Datatype of each receive buffer element (handle)
- **Source**: Rank of source (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)
- **Status**: Status object (Status)

Vector Addition: Server Process (I)

```
void data_server(unsigned int vector_size) {
  int np, num_nodes = np - 1, first_node = 0, last_node = np - 2;
  unsigned int num_bytes  = vector_size * sizeof(float);
  float *input_a = 0, *input_b = 0, *output = 0;

  /* Set MPI Communication Size */
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  /* Allocate input data */
  input_a = (float *)malloc(num_bytes);
  input_b = (float *)malloc(num_bytes);
  output = (float *)malloc(num_bytes);
  if(input_a == NULL || input_b == NULL || output == NULL) {
    printf("Server couldn't allocate memory\n");
    MPI_Abort( MPI_COMM_WORLD, 1 );
  }
  /* Initialize input data */
  random_data(input_a, vector_size , 1, 10);
  random_data(input_b, vector_size , 1, 10);
```

Vector Addition: Server Process (II)

```
  /* Send data to compute nodes */
  float *ptr_a = input_a;
  float *ptr_b = input_b;
  for(int process = 1; process < last_node; process++) {
    MPI_Send(ptr_a, vector_size / num_nodes, MPI_FLOAT,
        process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_a += vector_size / num_nodes;

    MPI_Send(ptr_b, vector_size / num_nodes, MPI_FLOAT,
        process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_b += vector_size / num_nodes;
  }
  /* Wait for nodes to compute */
  MPI_Barrier(MPI_COMM_WORLD);
```

Vector Addition: Server Process (III)

```
    /* Wait for previous communications */
    MPI_Barrier(MPI_COMM_WORLD);
    /* Collect output data */
    MPI_Status status;
    for(int process = 0; process < num_nodes;
process++) {
      MPI_Recv(output + process * num_points /
num_nodes,
          num_points / num_comp_nodes, MPI_REAL, process,
          DATA_COLLECT, MPI_COMM_WORLD, &status );
    }
    /* Store output data */
    store_output(output, dimx, dimy, dimz);
    /* Release resources */
    free(input_a);
    free(input_b);
    free(output);
}
```


Vector Addition: Compute Process (I)

```
void compute_node(unsigned int vector_size ) {
  int np;
  unsigned int num_bytes = vector_size * sizeof(float);
  float *input_a, *input_b, *output;
  MPI_Status status;

  MPI_Comm_size(MPI_COMM_WORLD, &np);
  int server_process = np - 1;
  /* Alloc host memory */
  input_a = (float *)malloc(num_bytes);
  input_b = (float *)malloc(num_bytes);
  output = (float *)malloc(num_bytes);
  /* Get the input data from server process */
  MPI_Recv(input_a, vector_size, MPI_FLOAT, server_process,
       DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
  MPI_Recv(input_b, vector_size, MPI_FLOAT, server_process,
       DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
```

MPI Barriers

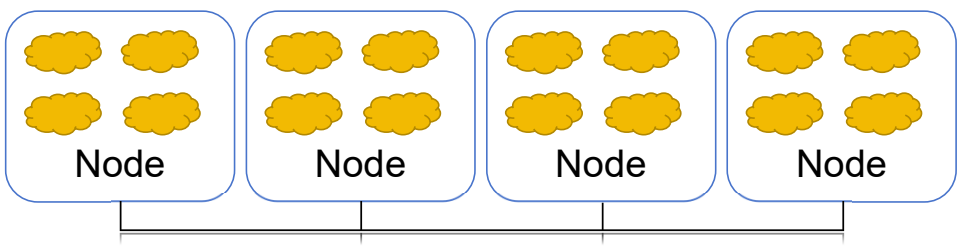
- int MPI_Barrier (MPI_Comm comm)
  - Comm: Communicator (handle)
- Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.

MPI Barriers

- Wait until all other processes in the MPI group reach the same barrier
  - All processes are executing Do_Stuff()
  - Some processes reach the barrier and the wait in the barrier until all reach the barrier

Example Code

```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```



Vector Addition: Compute Process (II)

```
    /* Compute the partial vector addition */
    for(int i = 0; i < vector_size; ++i) {
      output[i] = input_a[i] + input_b[i];
    }
    /* Report to barrier after computation is done*/
    MPI_Barrier(MPI_COMM_WORLD);
    /* Send the output */
    MPI_Send(output, vector_size, MPI_FLOAT,
        server_process, DATA_COLLECT, MPI_COMM_WORLD);
    /* Release memory */
    free(input_a);
    free(input_b);
    free(output);
}
```

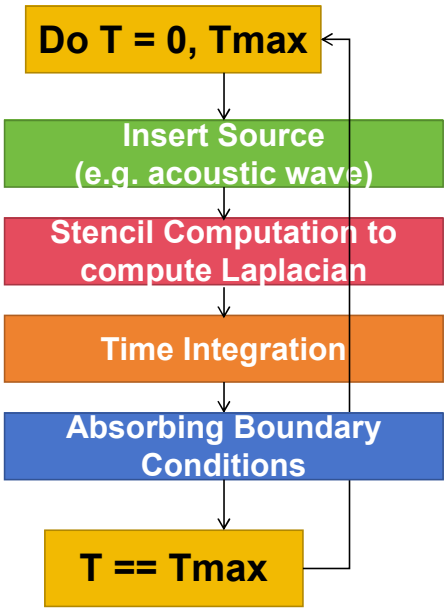

讲授内容：Related Programming Models: MPI

① Warps and SIMD Hardware

② Performance Impact of Control Divergence

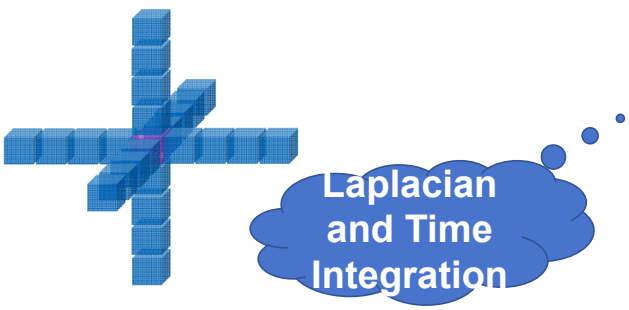
③ Overlapping Computation with Communication

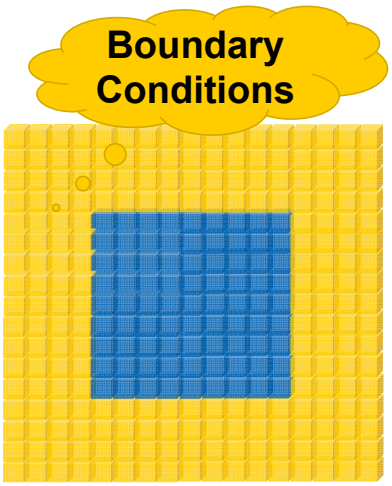
A Typical Wave Propagation Application



Review of Stencil Computations

- Example: wave propagation modeling
- $\nabla^2 U - \frac{1}{v^2}\frac{\partial U}{\partial t} = 0$
- Approximate Laplacian using finite differences





Wave Propagation: Kernel Code

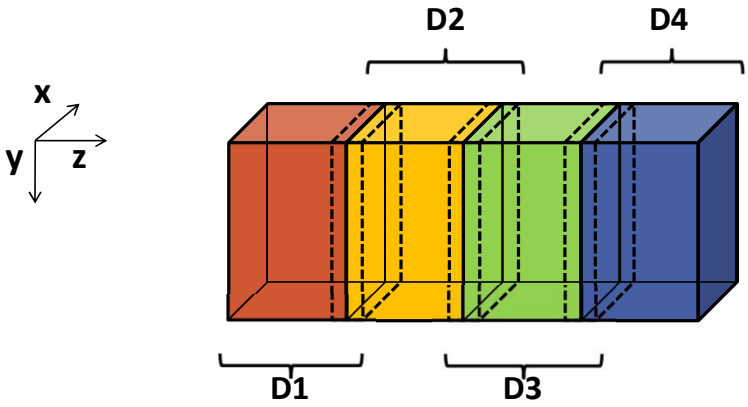
```
/* Coefficients used to calculate the laplacian */
__constant__ float coeff[5];

__global__ void wave_propagation(float *next, float *in,
            float *prev, float *velocity, dim3 dim)
{
   unsigned x = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned y = threadIdx.y + blockIdx.y * blockDim.y;
   unsigned z = threadIdx.z + blockIdx.z * blockDim.z;

   /* Point index in the input and output matrixes */
   unsigned n = x + y * dim.z + z * dim.x * dim.y;

   /* Only compute for points within the matrixes */
   if(x < dim.x && y < dim.y && z < dim.z) {

      /* Calculate the contribution of each point to the laplacian */
      float laplacian = coeff[0] + in[n];
```

Wave Propagation: Kernel Code

```
      for(int i = 1; i < 5; ++i) {
         laplacian += coeff[i] *
                      (in[n - i] + /* Left */
                      in[n + i] + /* Right */
                      in[n - i * dim.x] + /* Top */
                      in[n + I * dim.x] + /* Bottom
*/
                      in[n - i * dim.x * dim.y] + /*
Behind */
                      in[n + i * dim.x * dim.y]);  /*
Front */
      }

      /* Time integration */
      next[n] = velocity[n] * laplacian + 2 * in[n]
- prev[n];
   }
}
```

Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies



Wave Propagation: Main Process

```
int main(int argc, char *argv[]) {
    int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Nedded 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
```

Stencil Code: Server Process (I)

```
void data_server(int dimx, int dimy, int dimz, int nreps)
{
    int np, num_comp_nodes = np - 1, first_node = 0,
last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(float);
    float *input=0, *output = NULL, *velocity = NULL;
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    velocity = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL || velocity == NULL)
{
        printf("Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data and velocity */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    random_data(velocity, dimx, dimy ,dimz , 1, 10);
```

Stencil Code: Server Process (II)

```
/* Calculate number of shared points */
int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
float *input_send_address = input;

/* Send input data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, first_node,
      DATA_DISTRIBUTE, MPI_COMM_WORLD );
send_address += dimx * dimy * (dimz / num_comp_nodes - 4);

/* Send input data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
   MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
         DATA_DISTRIBUTE, MPI_COMM_WORLD);
   send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send input data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, last_node,
      DATA_DISTRIBUTE, MPI_COMM_WORLD);
```

Stencil Code: Server Process (II)

```
float *velocity_send_address = velocity;

/* Send velocity data to compute nodes */
for(int process = 0; process < last_node + 1; process++) {
    MPI_Send(send_address, edge_num_points, MPI_FLOAT, process,
           DATA_DISTRIBUTE, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_comp_nodes; process++)
    MPI_Recv(output + process * num_points / num_comp_nodes,
        num_points / num_comp_nodes, MPI_FLOAT, process,
        DATA_COLLECT, MPI_COMM_WORLD, &status );
}
```


Stencil Code: Server Process (III)

```
    /* Store output data */
    store_output(output, dimx,
  dimy, dimz);

    /* Release resources */
    free(input);
    free(velocity);
    free(output);
  }
```

Stencil Code: Compute Process (I)

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_ghost_points = 4 * dimx * dimy;
    unsigned int num_ghost_bytes  = num_ghost_points * sizeof(float);

    int left_ghost_offset   = 0;
    int right_ghost_offset  = dimx * dimy * (4 + dimz);

    float *input = NULL, *output = NULL, *prev = NULL, *v = NULL;

    /* Allocate device memory for input and output data */
    gmacMalloc((void **)&input,  num_bytes);
    gmacMalloc((void **)&output, num_bytes);
    gmacMalloc((void **)&prev, num_bytes);
    gmacMalloc((void **)&v, num_bytes);
```

Stencil Code: Compute Process (II)

```
MPI_Status status;
int left_neighbor  = (pid > 0) ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;
int server_process = np - 1;

/* Get the input data from server process */
float *rcv_address = input + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
      DATA_DISTRIBUTE, MPI_COMM_WORLD, &status );

/* Get the velocity data from server process */
rcv_address = h_v + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
      DATA_DISTRIBUTE, MPI_COMM_WORLD, &status );
```

讲授内容：Related Programming Models: MPI

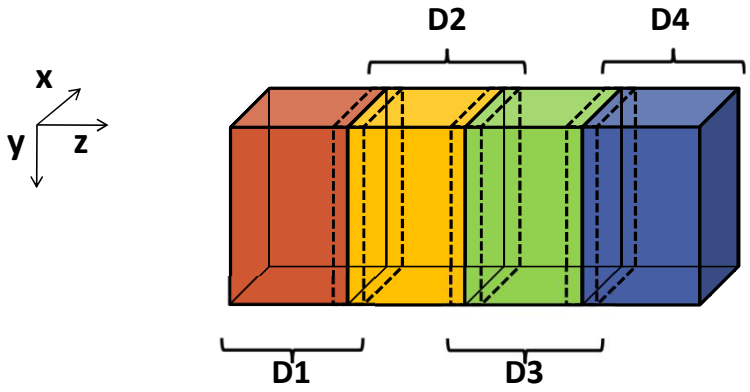
① Warps and SIMD Hardware

② Performance Impact of Control Divergence

③ Overlapping Computation with Communication

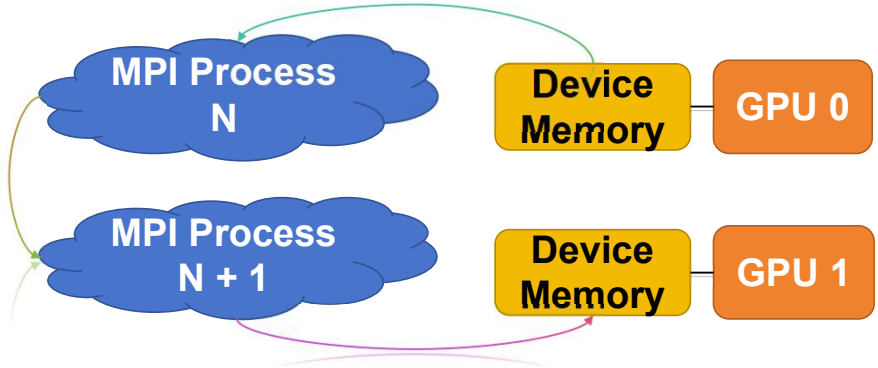
Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies



CUDA and MPI Communication

- Source MPI process:
  - cudaMemcpy(tmp,src, cudaMemcpyDeviceToHost)
  - MPI_Send()
- Destination MPI process:
  - MPI_Recv()
  - cudaMemcpy(dst, src, cudaMemcpyDeviceToDevice)



Data Server Process Code (I)

```
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np,
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    num_comp_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(float);
    float *input=0, *output=0;
        /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL) {
        printf("server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    /* Calculate number of shared points */
    int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
    int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
    float *send_address = input;
```

Data Server Process Code (II)

```
/* Send data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
      0, MPI_COMM_WORLD );

send_address += dimx * dimy * (dimz / num_comp_nodes - 4);
/* Send data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
   MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
         0, MPI_COMM_WORLD);
   send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
      0, MPI_COMM_WORLD);
```


Compute Process Code (I).

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
     MPI_Comm_size(MPI_COMM_WORLD, &np);
    int server_process = np - 1;

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_halo_points = 4 * dimx * dimy;
    unsigned int num_halo_bytes  = num_halo_points * sizeof(float);

    /* Alloc host memory */
    float *h_input  = (float *)malloc(num_bytes);
        /* Alloca device memory for input and output data */
    float *d_input = NULL;
    cudaMalloc((void **)&d_input,  num_bytes );
    float *rcv_address = h_input + num_halo_points * (0 == pid);
    MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status );
    cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```

Stencil Code: Kernel Launch

```
void launch_kernel(float *next, float *in, float
*prev, float *velocity,
                int dimx, int dimy, int dimz)
{
   dim3 Gd, Bd, Vd;

   Vd.x = dimx; Vd.y = dimy; Vd.z = dimz;

   Bd.x = BLOCK_DIM_X; Bd.y = BLOCK_DIM_Y; Bd.z =
BLOCK_DIM_Z;

   Gd.x = (dimx + Bd.x - 1) / Bd.x;
   Gd.y = (dimy + Bd.y - 1) / Bd.y;
   Gd.z = (dimz + Bd.z - 1) / Bd.z;

   wave_propagation<<<Gd, Bd>>>(next, in, prev,
velocity, Vd);
}
```

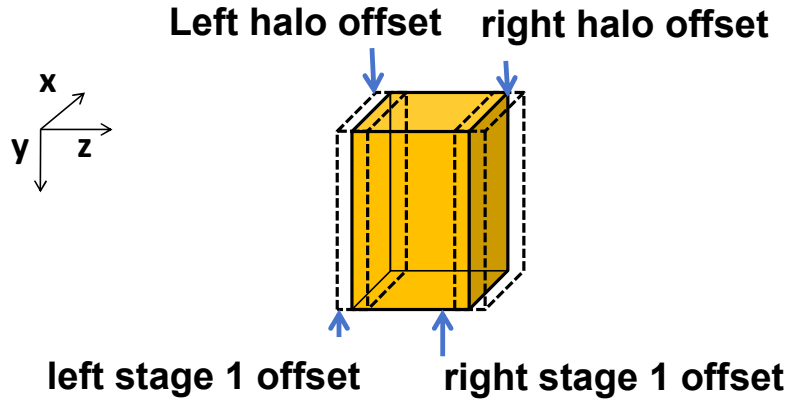

MPI Sending and Receiving Data

- int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
  - Sendbuf:Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

Compute Process Code (II)

```
float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_output = (float *)malloc(num_bytes);
cudaMalloc((void **)&d_output, num_bytes );

float *h_left_boundary = NULL, *h_right_boundary = NULL;
float *h_left_halo = NULL, *h_right_halo = NULL;

/* Alloc host memory for halo data */
cudaHostAlloc((void **)&h_left_boundary, num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_boundary,num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_left_halo,     num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_halo,    num_halo_bytes, cudaHostAllocDefault);

 /* Create streams used for stencil computation */
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
```


Device Memory Offsets Used for Data Exchange with Neighbors



Compute Process Code (III)

```
MPI_Status status;
int left_neighbor  = (pid > 0)     ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

/* Upload stencil cofficients */
upload_coefficients(coeff, 5);

int left_halo_offset   = 0;
int right_halo_offset  = dimx * dimy * (4 + dimz);
int left_stage1_offset  = 0;
int right_stage1_offset = dimx * dimy * (dimz - 4);
int stage2_offset       = num_halo_points;

MPI_Barrier( MPI_COMM_WORLD );
for(int i=0; i < nreps; i++) {
    /* Compute boundary values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset,
        d_input + left_stage1_offset, dimx, dimy, 12, stream0);
    launch_kernel(d_output + right_stage1_offset,
        d_input + right_stage1_offset, dimx, dimy, 12, stream0);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                dimx, dimy, dimz, stream1);
```

Compute Process Code (IV)

```
    /* Copy the data needed by other nodes to the
host */
    cudaMemcpyAsync(h_left_boundary, d_output +
num_halo_points,
         num_halo_bytes, cudaMemcpyDeviceToHost,
stream0 );
    cudaMemcpyAsync(h_right_boundary,
          d_output + right_stage1_offset +
num_halo_points,
          num_halo_bytes, cudaMemcpyDeviceToHost,
stream0 );
    cudaStreamSynchronize(stream0);
```

Syntax for MPI_Sendrecv()

- int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
  - Sendbuf:Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

Compute Process Code (V)

```
    /* Send data to left, get data from right */
    MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
             left_neighbor,  i, h_right_halo,
             num_halo_points, MPI_FLOAT, right_neighbor, i,
             MPI_COMM_WORLD, &status );
    /* Send data to right, get data from left */
    MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
             right_neighbor, i, h_left_halo,
             num_halo_points, MPI_FLOAT, left_neighbor,  i,
             MPI_COMM_WORLD, &status );

    cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
             num_halo_bytes, cudaMemcpyHostToDevice, stream0);
    cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
             num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
    cudaDeviceSynchronize();
    float *temp = d_output;
    d_output = d_input; d_input = temp;
}
```

Compute Process Code (VI)

```
    /* Wait for previous communications */
    MPI_Barrier(MPI_COMM_WORLD);

    float *temp = d_output;
    d_output = d_input;
    d_input = temp;

    /* Send the output, skipping halo points */
    cudaMemcpy(h_output, d_output, num_bytes,
               cudaMemcpyDeviceToHost);
    float *send_address = h_output + num_ghost_points;
    MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
          server_process, DATA_COLLECT, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    /* Release resources */
    free(h_input); free(h_output);
    cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
    cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
    cudaFree( d_input ); cudaFree( d_output );
}
```


Data Server Code (III)

```
   /* Wait for nodes to compute */
   MPI_Barrier(MPI_COMM_WORLD);

   /* Collect output data */
   MPI_Status status;
   for(int process = 0; process < num_comp_nodes; process++)
      MPI_Recv(output + process * num_points / num_comp_nodes,
         num_points / num_comp_nodes, MPI_REAL, process,
         DATA_COLLECT, MPI_COMM_WORLD, &status );

   /* Store output data */
   store_output(output, dimx, dimy, dimz);

   /* Release resources */
   free(input);
   free(output);
}
```

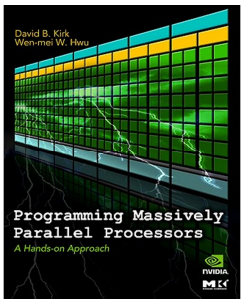
More on MPI Message Types

- Point-to-point communication
  - Send and Receive
- Collective communication
  - Barrier
  - Broadcast
  - Reduce
  - Gather and Scatter

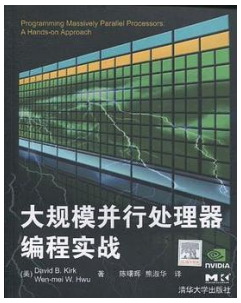
讲授内容

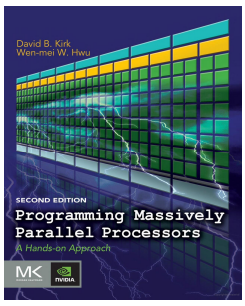
- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

参考书：CUDA编程

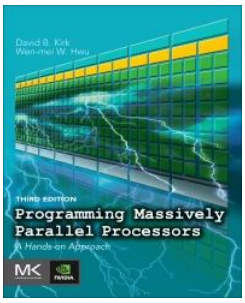


2010

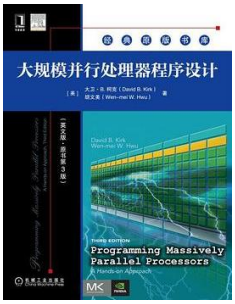


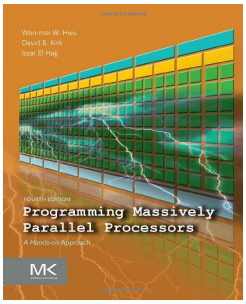


2012



2017





2022

✓**Third Edition:**

https://www.sciencedirect.com/book/9780128119860/programming-massively-parallel-processors

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach;

Contents

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach;

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach;

讲授内容

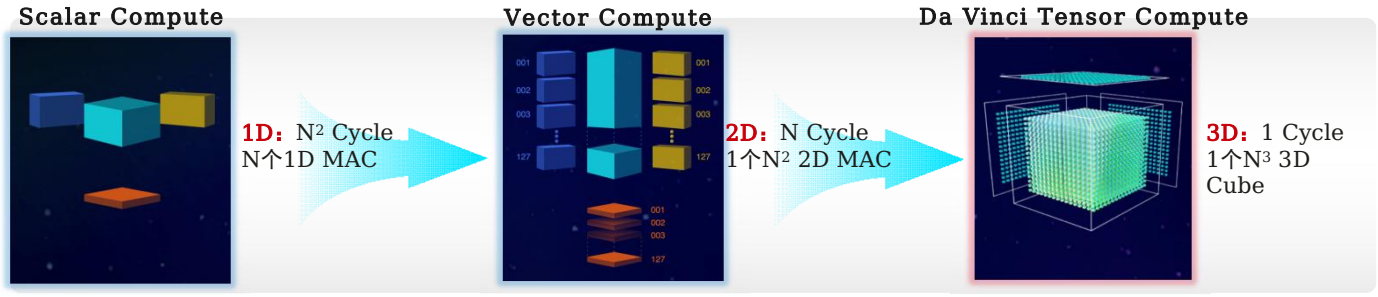
- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

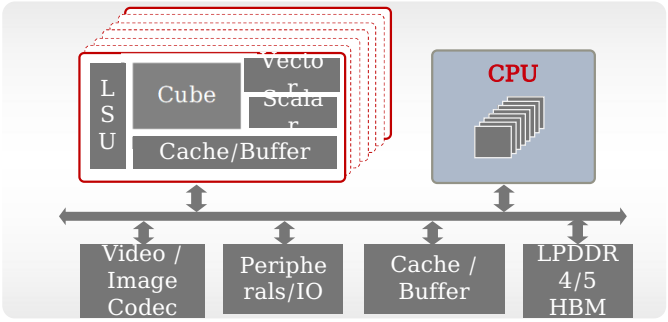
讲授内容：华为CANN编程（一）

- 华为AI全栈解决方案介绍
- 昇腾AI处理器硬件架构

昇腾AI：持续打造极致性能、极简易用的全场景人工智能平台

芯片层：基于Da Vinci AI技术架构





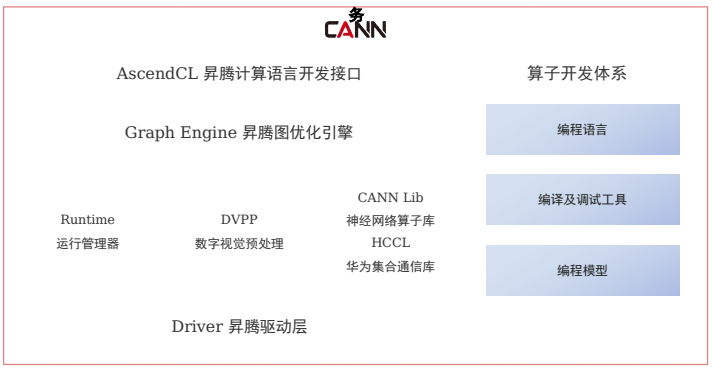
3D Cube：16^3三维弹性立方体

- 高算力：可在一个时钟周期内完成4096个FP16 MAC运算。
- 高效：支持几十毫瓦IP到几百瓦芯片，适应端、边和云的平滑架构扩展。

HUAWEI

异构计算架构CANN：开发体系开放易用，软硬协同释放极致性能

向上提供统一编程接口及友好开发体系，支持用户快速构建基于昇腾平台的AI应用和业务



向下使能昇腾处理器的并行加速能力，释放硬件澎湃性能

操作系统

Atlas 系列硬件

释放澎湃算力

任务自动流水、算子深度融合等优化技术，持续保持性能领先

1400+ 高性能算子

80% 主流算子支持动态Shape

使能高效开发

升级编程语言，提供更为简单的计算和内存抽象、自动流水同步、更简便的调试手段，降低开发者编码成本

原生C/C++开发 | 业界通用的编程范式

2人周 开发自定义算子

生态开放兼容

兼容主流 AI框架，接入主流开源社区提供昇腾算力支持，助力社区共建

6+	90%+	900+
支持主流框架	支持主流算子	提供优选模型

HUAWEI

昇思MindSpore：兼容多样性算力，使能AI模型创新

全场景覆盖

- 南北向解耦，支持多样性算力
- 统一中间表达MindIR，端边云场景模型部署零转换
- 模型自适应压缩，框架自定义裁剪

全自动并行

- 一行代码实现串行算法并行化，并行代码量降低80%
- 并行策略业界最多，自动搜索最佳并行策略，降低系统调优时间60%

全流程极简

- 动静态图统一，兼顾模型开发效率与执行效率
- 可视化模型调优，快速溯源异常Tensor
- 400+高性能模型，开箱即用

HUAWEI

结合行业Know-how，提供行业SDK，加速客户业务开发

MindX：昇腾应用使能

MindX DL 深度学习使能	MindX Edge 智能边缘使能	MindX SDK	ModelZoo 昇腾亲和模型库
快速搭建训练平台	快速完成边缘AI部署与推理	行业应用开发套件	900+高性能预训练模型 200+参考设计

MindX SDK：沉淀行业知识，使能行业应用 极简开发

基于CANN接口封装，结合行业场景业务流，将各环节接口和数据管理接口封装到插件，通过插件编排像搭积木一样编排好推理应用，提升开发效率

插件化开发模式
推理流程“插件化”，每个插件提供一种功能
通过组装不同的插件，灵活适配推理业务流程

丰富的插件库
用户可根据业务需求组合解码、抠图、缩放、模型推理、数据序列化等插件

2人月
传统应用开发方式

2人天
基于SDK开发方式

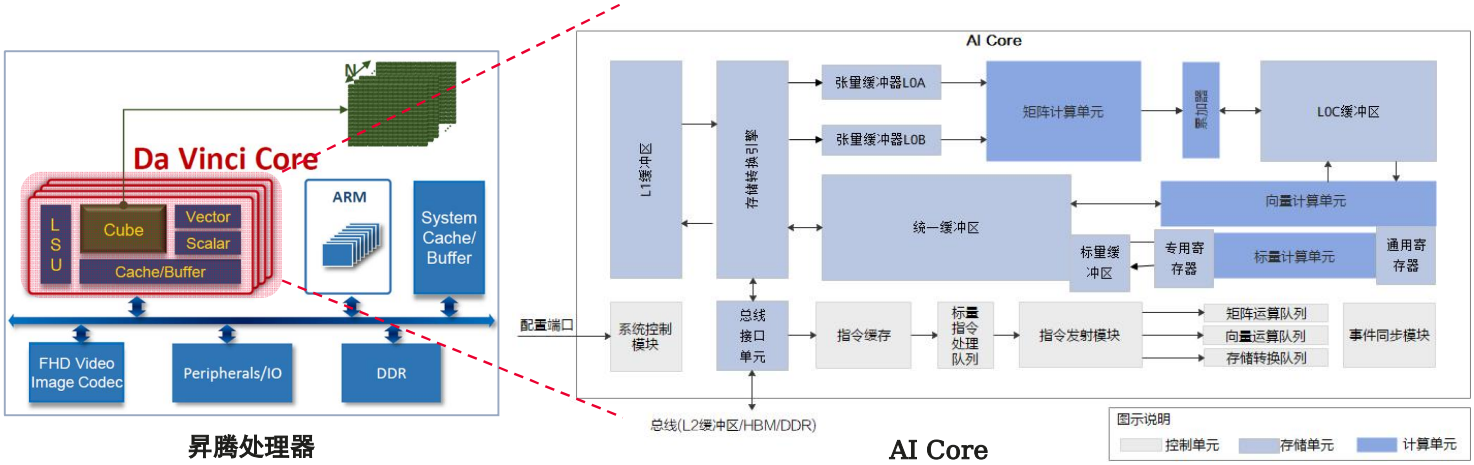
已上线 即将发布

HUAWEI

讲授内容：华为CANN编程（一）

- 华为AI全栈解决方案介绍
- 昇腾AI处理器硬件架构

昇腾处理器计算核心——AI Core

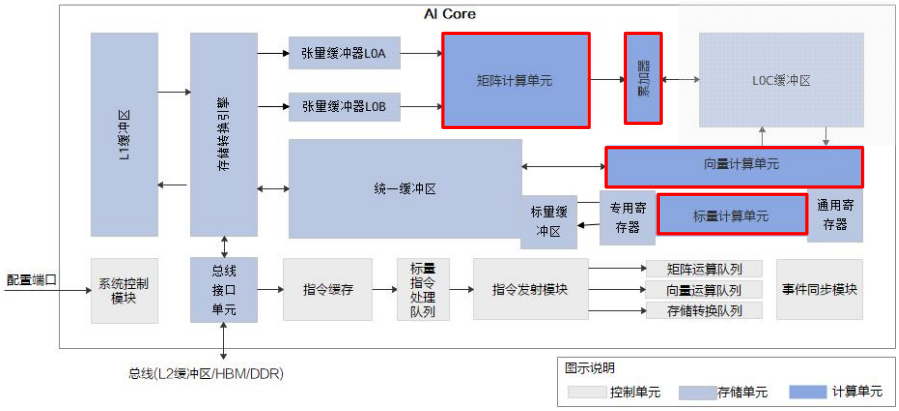


AI Core是昇腾AI处理器的计算核心，采用华为自研的达芬奇架构，通常也被叫做DaVinci Core。

达芬奇架构主要部分：

- **计算单元：** 包含三种基础计算资源（矩阵计算单元、向量计算单元、标量计算单元）
- **存储系统：** AI Core的片上存储单元和相应的数据通路构成了存储系统。
- **控制单元：** 整个计算过程提供了指令控制，相当于AI Core的司令部，负责整个AI Core的运行。

AI Core：计算单元



- **矩阵计算单元（Cube Unit）：**

  矩阵计算单元和累加器主要完成矩阵相关运算。一拍完成一个fp16的 16x16与16x16矩阵乘（4096）； 如果是int8输入，则一拍完成 16*32 与 32*16 矩阵乘（8192）。

- **向量计算单元（Vector Unit）：**

  实现单向量或双向量之间的计算，功能覆盖各种基本的计算类型和许多定制的计算类型，主要包括FP16/FP32/Int32/Int8等数据类型的计算。一拍可以完成两个128长度fp16类型的向量相加/乘， 或者64个fp32/int32类型的向量相加/乘。

- **标量计算单元（Scalar Unit）：**

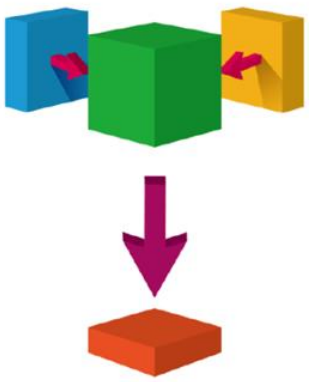
  负责标量的计算，相当于一个微型CPU，控制整个AI Core的运行，完成整个程序的循环控制、分支判断，可以为Cube/Vector提供数据地址和相关参数的计算，以及基本的算术运算。

- **累加器：**

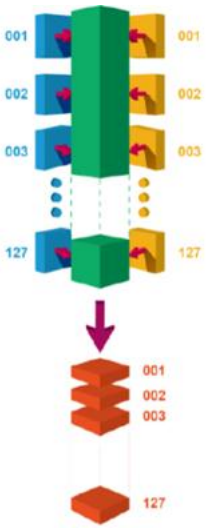
  把当前矩阵乘的结果与前次计算的中间结果相加， 可以用于完成卷积中加bias操作。

AI Core：计算单元 —— 加速原理

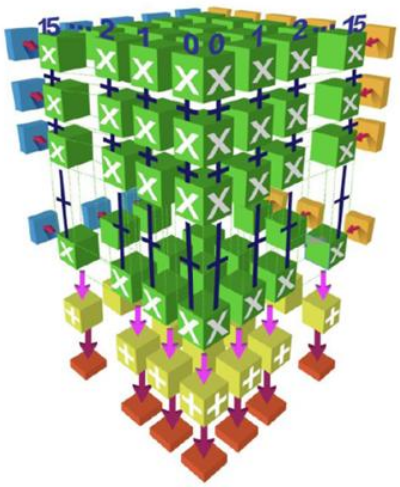
向量单元
全柔性计算



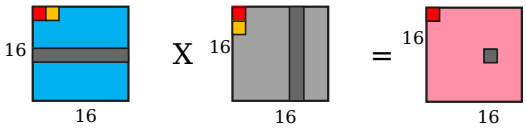
标量单元
高效、多样运算



矩阵单元
高密度计算



AI Core：计算单元 —— Cube Unit




```
float a[16][16], b[16][16], c[16][16];

CPU:
   for(int i=0; i < 16; i++)
      for(int j=0; j < 16; j++)
         for(int k=0; k < 16; k++) {
            c[i][j] += a[i][k] * b[k][j]
         }
Vector:
   for(int i=0; i < 16; i++)
      for(int j=0; j < 16; j++) {
         c[i][j] = a[i][:] *+ b[:][j]
      }
CUBE:
   CUBE:   c[:][:] = a[:][:] x
b[:][:]
```

算力密度高

Cycle=16*16*16*2=8192
DataNum per cycle:
Rd 2, Wr 1

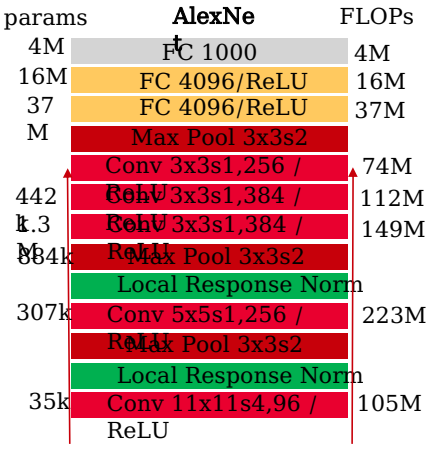
Cycle=16*16=256
DataNum per cycle:
Rd 2*16, Wr 16

Cycle=1
DataNum per cycle:
Rd 2*16*16,Wr:16*16

灵活

上图示例为一个矩阵a和另一个矩阵b之间的乘法运算c=a*b。

在不同的计算单元中实现该矩阵乘，其复杂度和运算效率差别很大。



AlexNet模型每层每秒浮点运算次数及参数数量

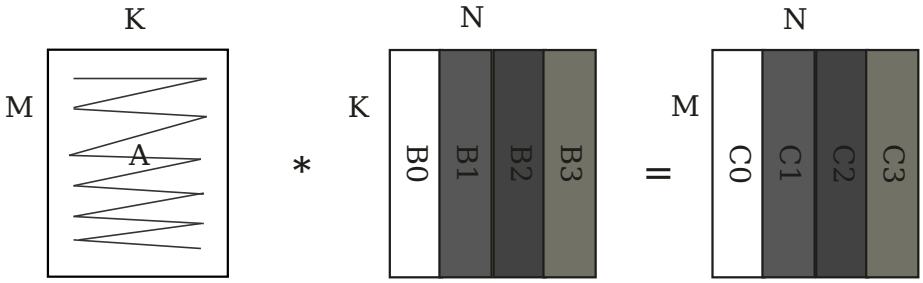
CNN经典模型的内存，计算量和参数数量对比

	AlexNet	VGG16	Inception-v3
模型内存（MB）	>200	>500	90-100
参数（百万）	60	138	23.2
计算量（百万）	720	15300	5000

从计算量角度，99%以上计算都是矩阵乘。

HUAWEI

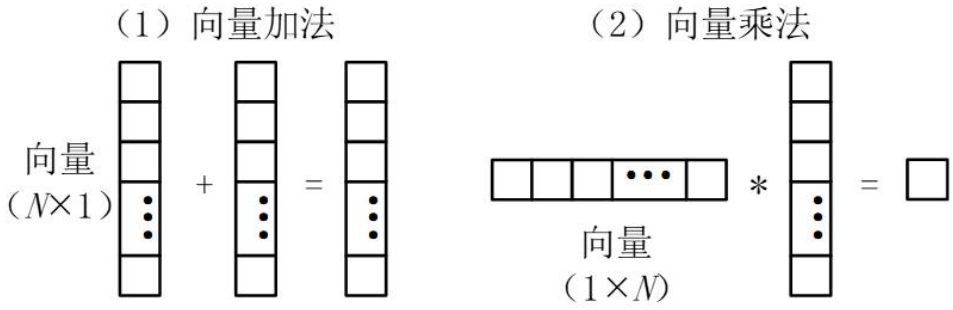
AI Core：计算单元 —— Cube Unit（矩阵分块计算）



- 一般在矩阵较大时，由于芯片上计算和存储资源有限，往往需要对矩阵进行分块平铺处理（Tiling）。受限于片上缓存的容量，当一次难以装下整个矩阵B时，可以将矩阵B划分成为B0、B1、B2和B3等多个子矩阵。而每一个子矩阵的大小都可以适合一次性存储到芯片上的缓存中并与矩阵A进行计算从而得到结果子矩阵。这样做的目的是充分利用数据的局部性原理，尽可能的把缓存中的子矩阵数据重复使用完毕并得到所有相关的子矩阵结果后再读入新的子矩阵开始新的周期。如此往复可以依次将所有的子矩阵都一一搬运到缓存中，并完成整个矩阵计算的全过程，最终得到结果矩阵C。

HUAWEI

AI Core：计算单元——Vector Unit



- AI Core中的向量计算单元主要负责完成向量相关的运算，能够实现向量与标量，或双向量之间的计算，功能覆盖各种基本和多种定制的计算类型，主要包括FP32、FP16、INT32和INT8等数据类型的计算。
- 如上图所示，向量计算单元可以快速完成两个FP16类型的向量相加或者相乘。向量计算单元的源操作数和目的操作数通常都保存在输出缓冲器中。对向量计算单元而言，输入的数据可以不连续，这取决于输入数据的寻址模式。

AI Core：计算单元 —— Scalar Unit

- 标量计算单元负责完成AI Core中与标量相关的运算。它相当于一个微型CPU，控制整个AI Core的运行。标量计算单元可以对程序中的循环进行控制，可以实现分支判断，其结果可以通过在事件同步模块中插入同步符的方式来控制AI Core中其它功能性单元的执行流水。它还为矩阵计算单元或向量计算单元提供数据地址和相关参数的计算，并且能够实现基本的算术运算。其它复杂度较高的标量运算则由专门的AI CPU通过算子完成。

- 在标量计算单元周围配备了多个通用寄存器（General Purpose Register，GPR）和专用寄存器（Special Purpose Register，SPR）。这些通用寄存器可以用于变量或地址的寄存，为算术逻辑运算提供源操作数和存储中间计算结果。专用寄存器的设计是为了支持指令集中一些指令的特殊功能，一般不可以直接访问，只有部分可以通过指令读写。

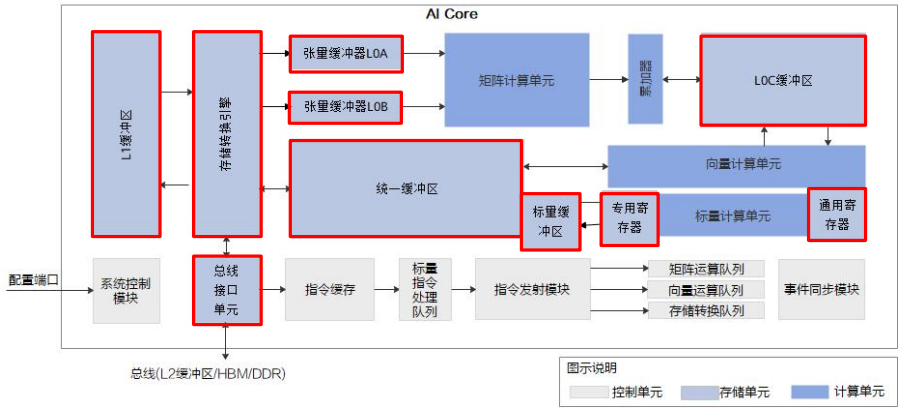
AI Core：计算单元 —— Scalar Unit

标量计算单元负责完成AI Core中与标量相关的运算。它相当于一个微型CPU，控制整个AI Core的运行。标量计算单元可以对程序中的循环进行控制，可以实现分支判断，其结果可以通过在事件同步模块中插入同步符的方式来控制AI Core中其它功能性单元的执行流水。它还为矩阵计算单元或向量计算单元提供数据地址和相关参数的计算，并且能够实现基本的算术运算。其它复杂度较高的标量运算则由专门的AI CPU通过算子完成。

在标量计算单元周围配备了多个通用寄存器（General Purpose Register，GPR）和专用寄存器（Special Purpose Register，SPR）。这些通用寄存器可以用于变量或地址的寄存，为算术逻辑运算提供源操作数和存储中间计算结果。专用寄存器的设计是为了支持指令集中一些指令的特殊功能，一般不可以直接访问，只有部分可以通过指令读写。

AI Core：存储系统

存储单元和相应的数据通路，构成了AI Core的存储系统。



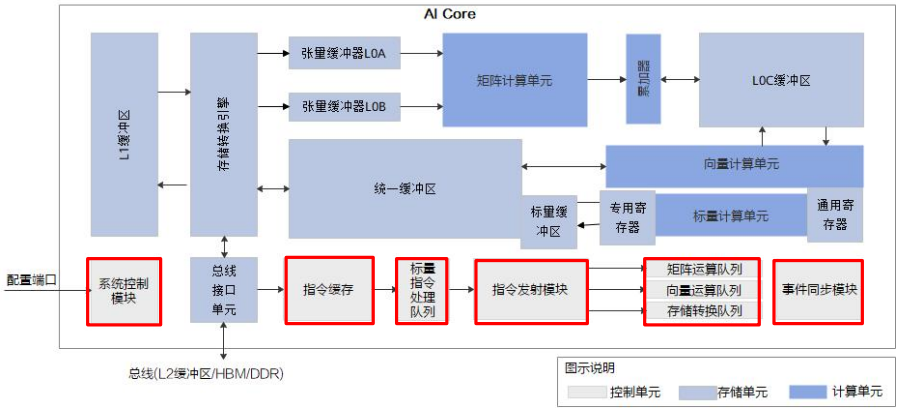
- **存储转换引擎：**
  负责AI Core内部数据在不同Buffer之间的读写管理及一些格式转换的操作，比如填充（padding）、转置（transpose）等。
  存储转换引擎还可以通过总线接口直接访问AI Core之外的更低层级的缓存。

- **缓冲区：**
  包含L1缓冲区、L0A/L0B缓冲区、L0C缓冲区、统一缓冲区、标量缓冲区。
  缓冲区可用来**暂时保留需要频繁重复使用的输入数据**，不需要每次都通过总线接口到AI Core的外部读取，从而在减少总线上数据访问频次的同时也降低了总线上产生拥堵的风险，达到节省功耗、提高性能的效果；
  缓冲区可用来**存放神经网络中每层计算的中间结果**，从而在进入下一层计算时方便的获取数据。相比较通过总线读取数据的带宽低，延迟大，通过缓冲区可以大大提升计算效率；

- **寄存器：**
  AI Core中的各类寄存器资源主要是标量计算单元在使用。

AI Core：控制单元

控制单元主要组成部分为系统控制模块、指令缓存、标量指令处理队列、指令发射模块、矩阵运算队列、向量运算队列、存储转换队列和事件同步模块。



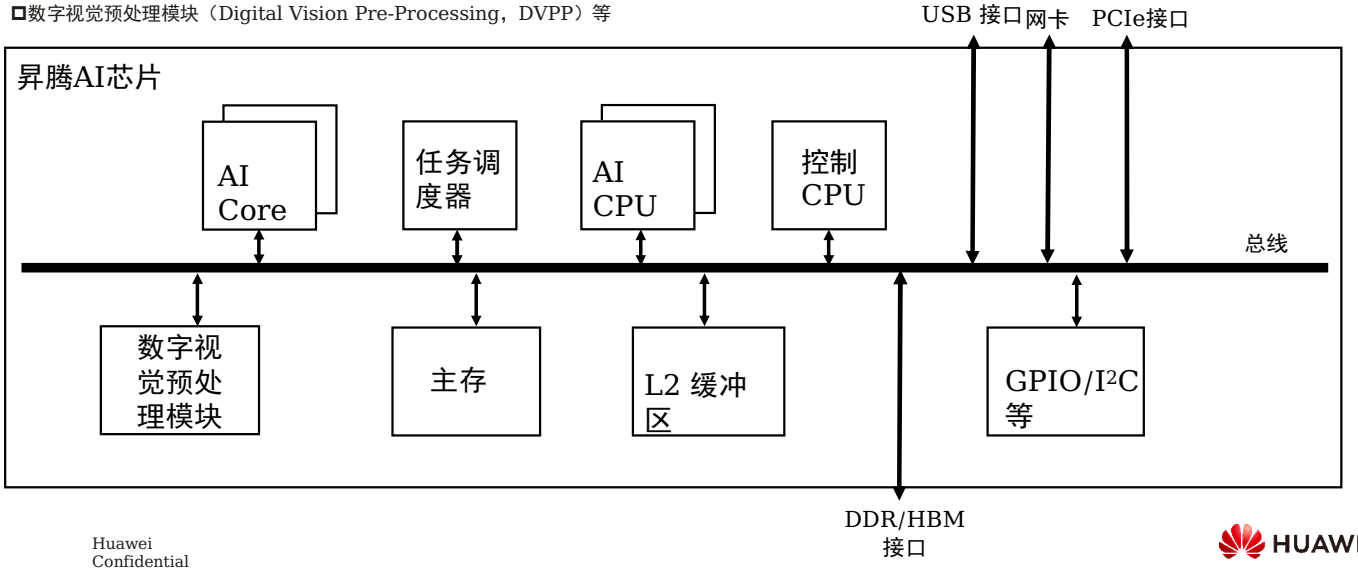
- **系统控制模块：**
  控制任务块（AI Core最小任计算任务粒度）的执行进程，在任务块执行完成后，系统控制模块会进行中断处理和状态申报。如果执行过程出错，会把执行的错误状态报告给任务调度器；
- **指令缓存：**
  在指令执行过程中，可以提前预取后续指令，并一次读入多条指令进入缓存，提升指令执行效率；
- **标量指令处理队列：**
  指令被解码后便会被导入标量队列中，实现地址解码与运算控制，这些指令包括矩阵计算指令、向量计算指令以及存储转换指令等；
- **指令发射模块：**
  读取标量指令队列中配置好的指令地址和参数解码，然后根据指令类型分别发送到对应的指令执行队列中，而标量指令会驻留在标量指令处理队列中进行后续执行；
- **指令执行队列：**
  指令执行队列由矩阵运算队列、向量运算队列和存储转换队列组成，不同的指令进入相应的运算队列，队列中的指令按进入顺序执行；
- **事件同步模块：**
  时刻控制每条指令流水线的执行状态，并分析不同流水线的依赖关系，从而解决指令流水线之间的数据依赖和同步的问题。

HUAWEI

昇腾AI处理器逻辑架构（AI Inference SoC）

Ascend 310处理器的主要架构组成：

- AI计算引擎（包括AI Core和AI CPU）
- 芯片系统控制CPU（Control CPU）
- 多层级的片上系统缓存（Cache）或缓冲区（Buffer）
- 数字视觉预处理模块（Digital Vision Pre-Processing，DVPP）等



HUAWEI

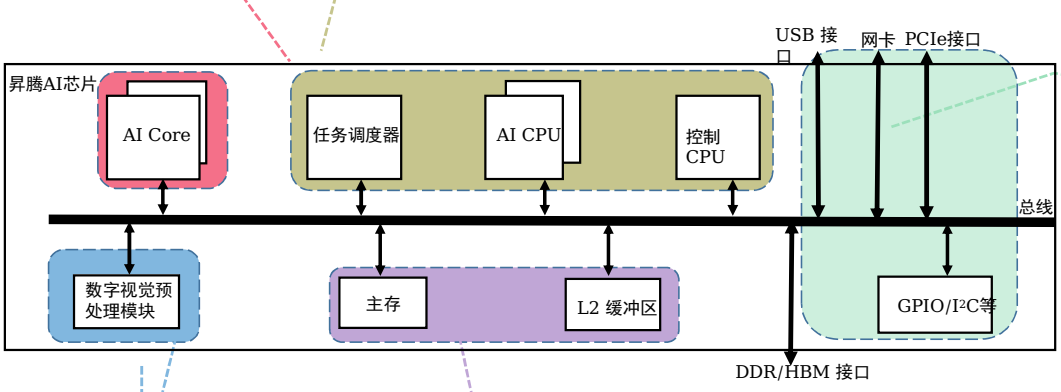
昇腾AI处理器逻辑架构（AI Inference SoC）

AI Core

昇腾AI芯片的计算核心，负责执行矩阵、向量、标量计算密集的算子任务，采用达芬奇架构。Ascend 310集成了2个AI Core。

ARM CPU核心

集成了8个ARM A55。其中一部分部署为**AI CPU**，负责执行不适合跑在AI Core上的算子（承担非矩阵类复杂计算）；一部分部署为专用于控制芯片整体运行的**控制CPU**。两类任务占用的CPU核数可由软件根据系统实际运行情况动态分配。此外，还部署了一个专用CPU作为任务调度器（**Task Scheduler**，TS），以实现计算任务在AI Core上的高效分配和调度；该CPU专门服务于AI Core和AI CPU，不承担任何其他的事务和工作。



对外接口

支持PCIE3.0、RGMII、USB3.0等高速接口、以及GPIO、UART、I2C、SPI等低速接口。

DVPP

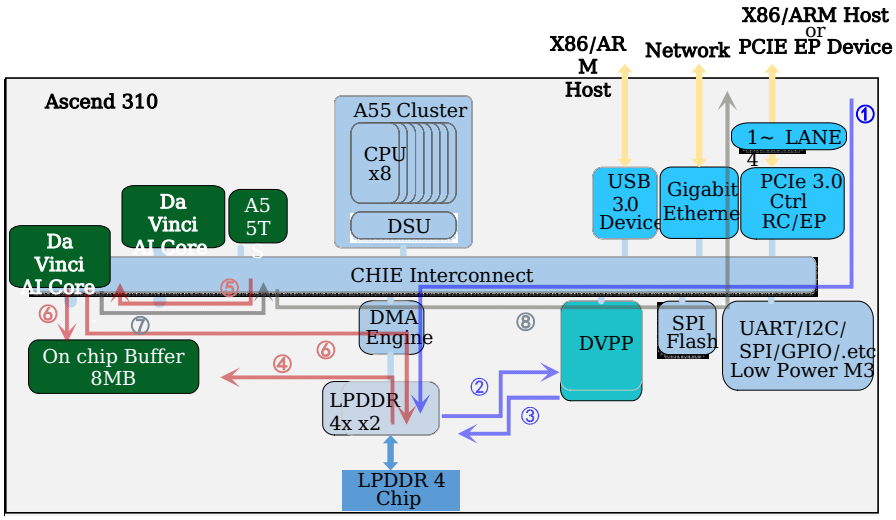
数字视觉预处理子系统，完成图像视频的编解码。

Cache & Buffer

SOC片内有层次化的memory结构，AI core内部有两级memory buffer，SOC片上还有8MB L2 buffer，专用于AI Core、AI CPU，提供高带宽、低延迟的memory访问。

HUAWEI

数据流图参考 —— 以物体识别推理应用为例



1、Camera数据采集和处理

① 摄像头传入压缩视频流，通过PCIE存储至DDR内存中。

② DVPP将压缩视频流读入缓存。

③ DVPP经过预处理，将解压缩的帧写入DDR内存。

2、对数据进行推理

④ 任务调度器（TS）向直接存储访问引擎（DMA）发送指令，将AI资源从DDR预加载到片上缓冲区。

⑤ 任务调度器（TS） 配置AI Core以执行任务。

⑥ AI Core工作时，它将读取特征图和权重并将结果写入DDR或片上缓冲区。

3、物体识别结果输出

⑦ AI Core完成处理后，发送信号给任务调度器（TS），任务调度器检查结果，如果需要会分配另一个任务，并返回步骤④ 。

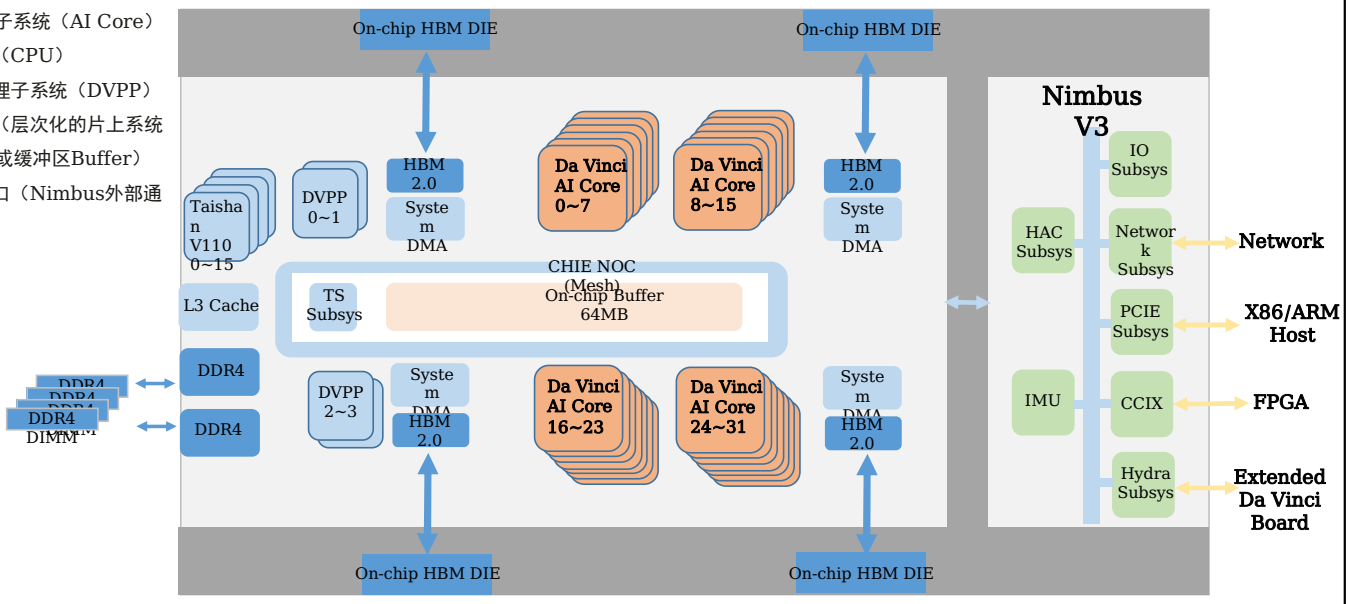
⑧ 当最后一个AI任务完成，任务调度器（TS）会将结果报告给Host。

HUAWEI

昇腾AI处理器逻辑架构（AI Training SoC）

Ascend 910处理器的主要架构组成：

- AI数据处理子系统（AI Core）
- 计算子系统（CPU）
- 图像视频处理子系统（DVPP）
- 存储子系统（层次化的片上系统缓存Cache或缓冲区Buffer）
- 低速外设接口（Nimbus外部通信模块）



HUAWEI

昇腾AI处理器逻辑架构（AI Training SoC）

CPU子系统

集成16个TaishanV110 Core。这些Taishan Core一部分部署为AI CPU，承担部分AI计算功能；一部分部署为Ctrl CPU，负责整SoC的控制功能。

TS CPU（Task Scheduler）

一个独立的4核A55 Cluster，负责任务调度，把算子任务切分之后，通过硬件调度器分发给AI Core或AI CPU。

DVPP

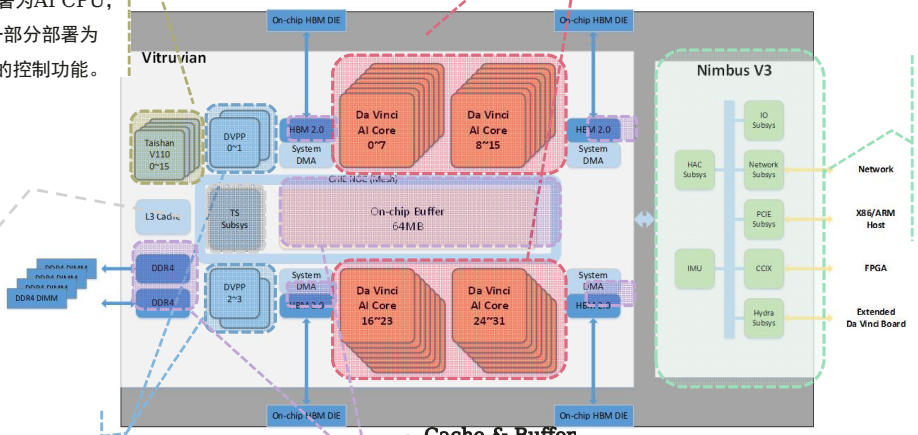
数字视觉预处理子系统，完成图像视频的编解码等预处理操作。

AI Core

昇腾AI芯片的计算核心，主要负责执行矩阵、向量计算密集的算子任务，采用达芬奇架构。 Ascend 910集成了32个AI Core。

Nimbus

提供x16 PCIe 4.0接口，和Host CPU对接，提供100G NIC（支持ROCE V2协议）用于跨服务器传递数据；集成1个A53 CPU核，执行启动、功耗控制等硬件管理任务。



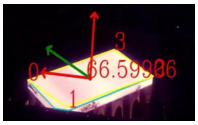
Cache & Buffer

片内有层次化的memory结构，AI Core内部有两级memory buffer，SOC片上还有64MB L2 buffer，专用于AI Core、AI CPU，提供高带宽、低延迟的memory访问。

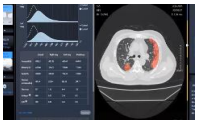
HUAWEI

通过硬件架构创新提升硬件工程能力，算力持续领先

Atlas 推理系列硬件



智能转钢系统
转钢准确率可达100%



医疗辅助诊断
阅片效率提升50倍

智能变电站
实时远程智能巡视

高速自由流
通行效率提升5倍

输电智能巡检
巡检效率，提升80倍

中心推理

Atlas 800 推理服务器

Atlas 300I Pro 推理卡

Atlas 300I 推理卡

Atlas 300V Pro 视频卡

边缘计算

Atlas 500 Pro 智能边缘服务器

Atlas 500 智能小站

端侧智能

Atlas 200 AI 加速模块

Atlas 训练系列硬件

Atlas 300T 训练卡

单卡算力**业界领先**
320 TFLOPS FP16

Atlas 800 训练服务器

最强算力密度
算力密度达到业界1.7倍

Atlas 900 PoD

超高能效比
20.48 PFLOPS/43 kW

Atlas 900 AI集群

业界首个全液冷AI集群
PUE<1.1
训练速度保持业界领先

人工智能计算中心
算力集群赋能产业集群

HUAWEI

Atlas 300I Pro

Atlas 300I Pro 推理卡：业界超强算力推理卡

- 140 TOPS INT8，典型功耗72W
- 24GB超大内存 | 204.8GB/s内存带宽，支持ECC
- PCIe 4.0 x16 Gen4.0，半高半长标卡

超强算力

单卡提供最大140 TOPS INT8算力
领先业界主流产品**2.12倍**

超高能效

单卡实现能效比达1.94 TOPS/W
领先业界主流产品**2.06倍**

安全启动

通过数字签名对待加载运行的软件（或固件）的完整性进行强校验，是构建可信的起点

互联网

内容审核　搜索推荐

智慧政务、智慧金融

票据识别　信息录入　档案资料　身份证实名认证　电子签名识别

其他行业

实时黑名单对比报警　语音分类

HUAWEI

Atlas 300T

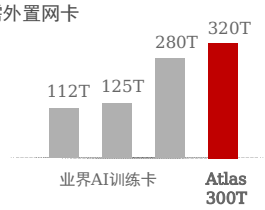
Atlas 300T 训练卡：业界最强算力训练卡

(型号：9000)

型号：9000

深度学习训练 | 天文探索
石油勘探 | 自动驾驶训练

- **多样算力选择：**
  - 320 TFLOPS FP16 (Pro)
  - 280 TFLOPS FP16
  - 220 TFLOPS FP16 (Lite)
- 32 GB HBM内存 + 16GB DDR4内存
- 支持PCIe 4.0x16，标准全高3/4长PCIe卡，适用于通用服务器
- 芯片直出 2 * 100G RoCE网口，无需外置网卡



最强算力

- 内置32个达芬奇AI Core，提供最大320 TFLOPS FP16性能，算力达到业界标卡巅峰

高集成度

- **AI算力、通用算力、I/O能力三合一：**芯片集成32个华为达芬奇AI Core + 16个TaiShan核 + 2 * 100GE RoCE v2网卡

最快带宽

- 支持PCIe 4.0和 2*100G RoCE高速接口，出口总带宽**56.5 Gb/s**，较业界1.8倍
- 无需外置网卡，训练数据和梯度同步效率提升**10%-70%**

HUAWEI

Atlas 800

Atlas 800 推理服务器产品亮点

(型号：3000)

Atlas 800 推理服务器

型号：3000

形态	2U AI服务器
CPU	2 * 鲲鹏920
CPU内存	32个DDR4内存插槽，最高2933 MT/s
AI加速卡	最大支持8个Atlas 300 AI加速卡 满足640路人/车/物视频智能分析
AI算力	704 TOPS INT8
本地存储	25*2.5 SAS/SATA 12*3.5 SAS/SATA 8*2.5 SAS/SATA+12x2.5 NVMe
PCIe	最多支持9个PCIe4.0 PCIe接口，其中1个为RAID扣卡专用的PCIe扩展槽位，另外8个为标准的PCIe扩展槽位

高性能

超强算力: 高性能鲲鹏920处理器
大内存容量：8通道内存技术，支持32个DDR4内存插槽
超强AI加速：支持8*Atlas 300 AI加速卡，满足多场景推理

开放易用

云边协同：统一开发、统一运维、安全升级，打造极致开发、使用体验
分层开放：提供丰富SDK，支持算子自定义开发及CPU开放，满足用户多样需求

灵活适配

灵活的IO扩展：支持灵活插卡和标准的智能网卡，实现丰富的网络配置
分级存储：支持大容量存储硬盘和ES3000 V5 NVMe SSD

安全可靠

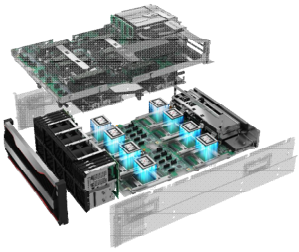
安全、可供应：采用华为全自研计算芯片，整机器件全国产化
可靠与质量保障：硬盘三级减振、45° C 散热等高可靠设计

HUAWEI

Atlas 800

Atlas 800 AI服务器：算力最强AI训练服务器

(型号：9000)



Atlas 800 AI服务器
型号：9000

深度学习训练 | AI超算
分布式训练平台

- 2P FLOPS FP16
- 4U服务器，支持4*Kunpeng 920+8*昇腾 910
- 32*DDR4+8*2.5"硬盘
- 8*100GE+1*100GEx2/25GEx4
- 5.5 kW最大整机功耗，支持风冷和液冷两种散热方式

最强算力密度

4U高度提供2P FLOPS FP16超强算力，算力密度达到业界2倍

超高能效

支持风冷和液冷两种散热方式，提供2P FLOPS/5.5 kW超高能效比，达到业界1.6倍，满足企业机房部署和集群高密部署不同场景需求。

高速网络带宽

8*100G RoCE v2高速接口，带宽达到业界2倍，芯片间跨服务器互联时延缩短10~70%

HUAWEI

讲授内容

- Related Programming Models: OpenCL
- Related Programming Models: OpenACC
- Multi-GPU: OpenMP
- Related Programming Models: MPI
- 教材总结
- 华为CANN编程（一）
- 华为CANN编程（二）

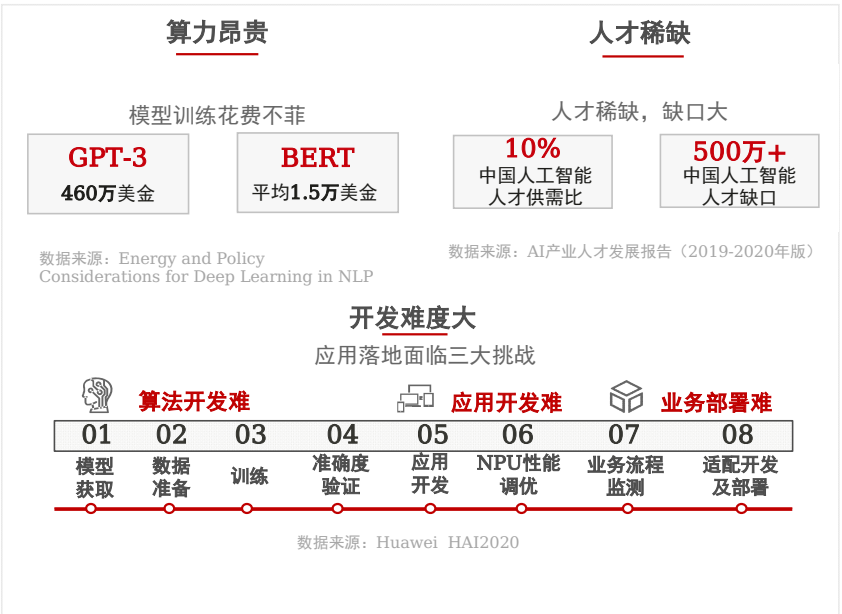
讲授内容：华为CANN编程（二）

- ➢昇腾AI基础软硬件平台
- ➢昇腾AI异构计算架构CANN
- ➢CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- ➢Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

AI深刻推动社会发展，同时挑战也随之而来

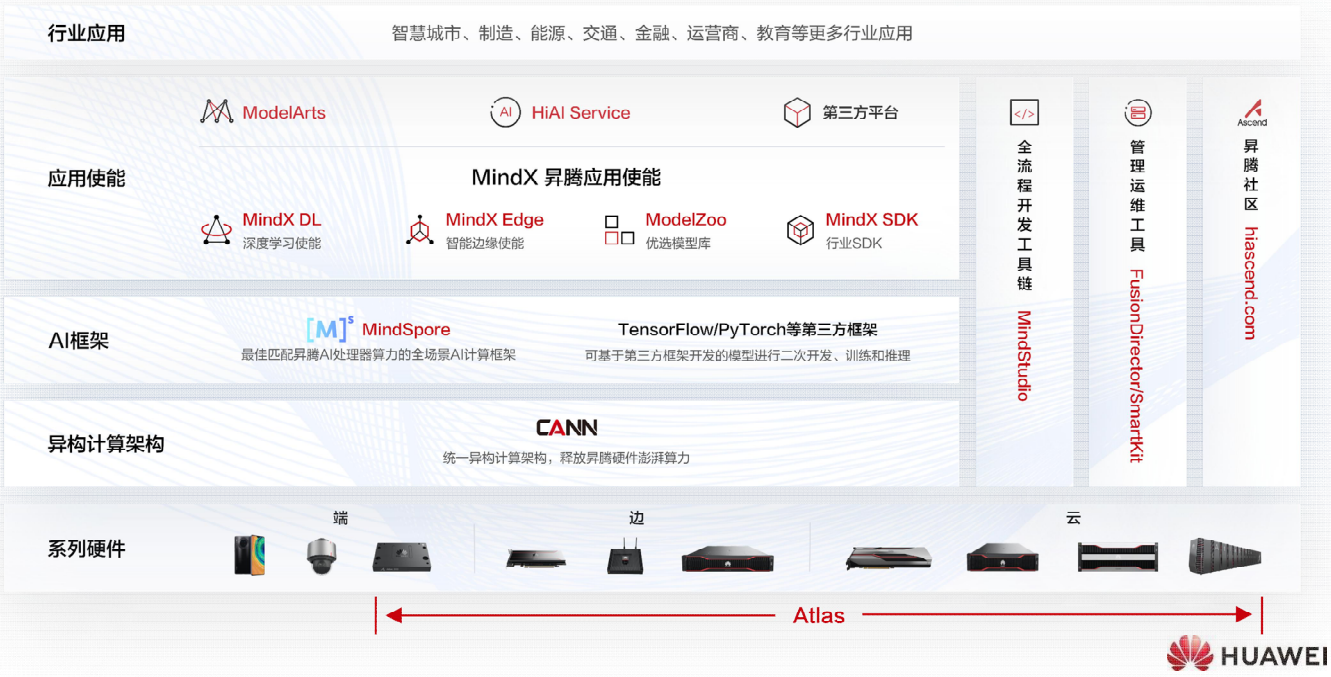
人工智能是一种新的通用目的技术（GPT）

人工智能发展挑战



HUAWEI

发展昇腾计算产业，打造最强竞争力的昇腾AI基础软硬件平台



Atlas系列硬件，使能全场景AI

AI计算中心
算力集群赋能
产业集群

深度学习平台
一站式AI模型
训练平台

全目标结构化
安全便捷的
道路和园区

医疗辅助诊断
阅片效率提升
助力智慧医院

智能变电站
实时远程
智能巡视

高速自由流
通行效率提升
助力智慧出行

输电智能巡检
巡检效率提升
远程可管可视

数据中心

Atlas 900
AI 集群

Atlas 800 9000
训练服务器

Atlas 300T
训练卡

Atlas 800 3000/3010
推理服务器

Atlas 300I
推理卡

智能边缘

Atlas 500 Pro
智能边缘服务器

Atlas 500
智能小站

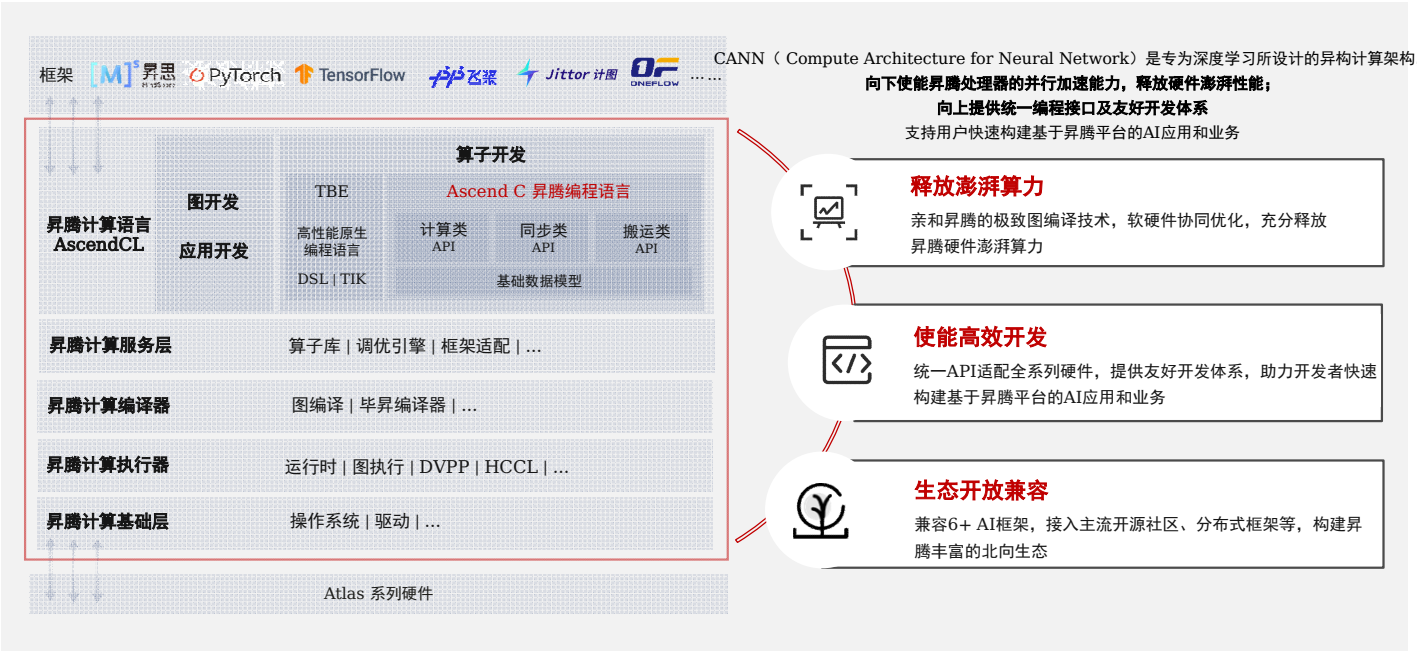
Atlas 200
AI 加速模块

昇腾AI处理器系列（针对推理/训练场景）

讲授内容：华为CANN编程（二）

- 昇腾AI基础软硬件平台
- 昇腾AI异构计算架构CANN
- CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

异构计算架构CANN：开发体系开放易用，软硬协同释放极致性能



HUAWEI

AscendCL 昇腾计算语言开发接口

昇腾计算开放编程框架，封装底层昇腾计算服务接口，提升编程易用性

应用开发接口

提供深度学习推理计算、图像预处理、单算子加速计算能力，实现对昇腾硬件算力的调用。

使用场景：

- 开发应用
- 供第三方框架调用
- 供第三方开发lib库

优势：

- 高度抽象
- 向后兼容
- 零感知芯片

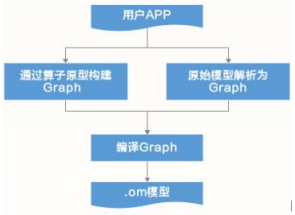
卡通图像生成

图开发

提供统一网络构图接口，支持多框架，支持用户在昇腾芯片上快速部署神经网络业务。

支持的构图方式：

- 通过算子原型构图
- 通过Parser解析为IR图

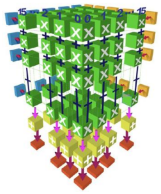


算子开发

采用C++和一组类库API实现算子编程方法，帮助开发者快速写出高效的流水计算，发挥昇腾硬件强大算力

Ascend C优势：

- 编程模型屏蔽硬件差异，编程范式提高开发效率
- 多层级API封装，兼顾易用与高效
- 孪生调试，模拟NPU行为，可先在CPU调试



HUAWEI

昇腾计算服务层

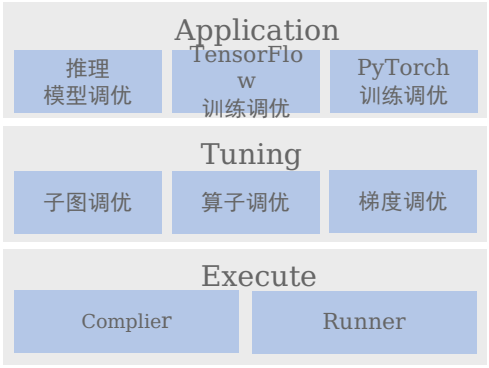
昇腾算子库AOL

算子在硬件上的加速计算构成了加速神经网络的基础和核心

- NN（Neural Network）
- BLAS（Basic Linear Algebra Subprograms）
- DVPP（Digital Video Pre-Processor）
- AIPP（AI Pre-Processing）
- HCCL（Huawei Collective Communication Library）

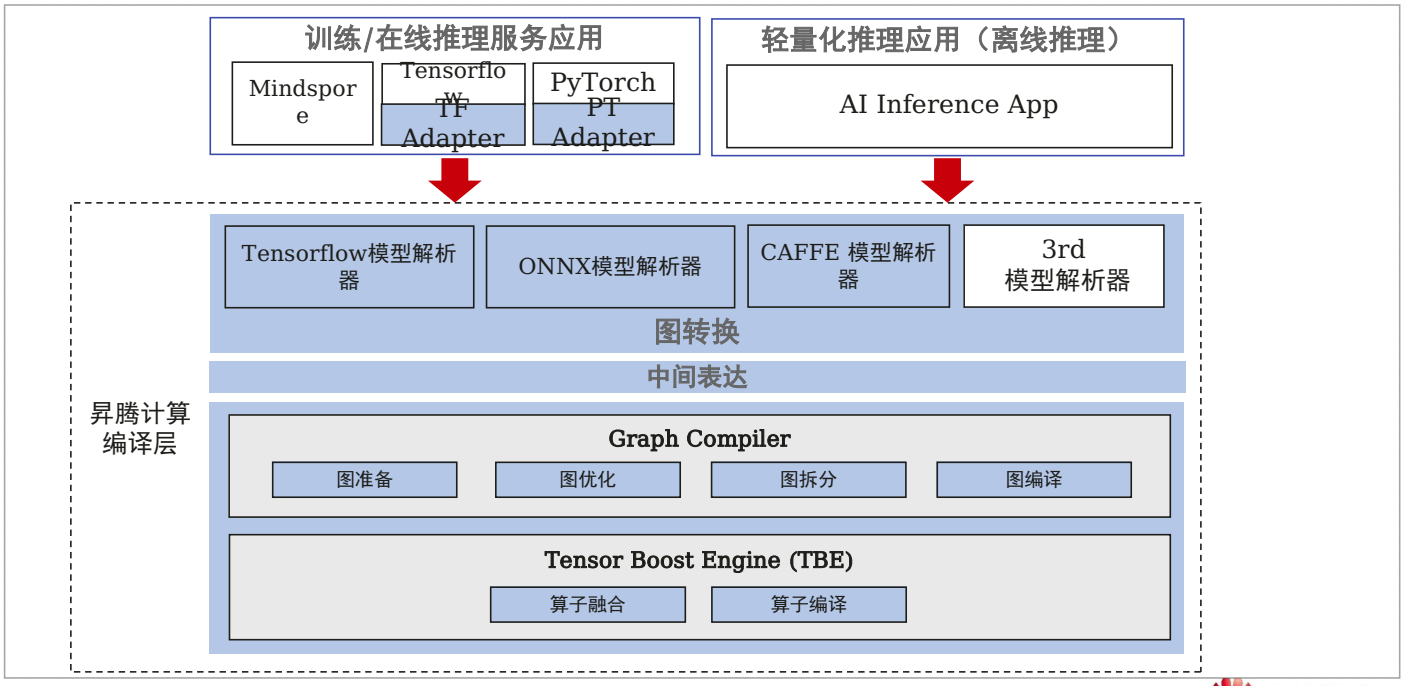
昇腾调优引擎AOE

AOE用于在推理、训练等场景对模型、算子、子图等进行调优，充分利用硬件资源，不断提升网络的性能



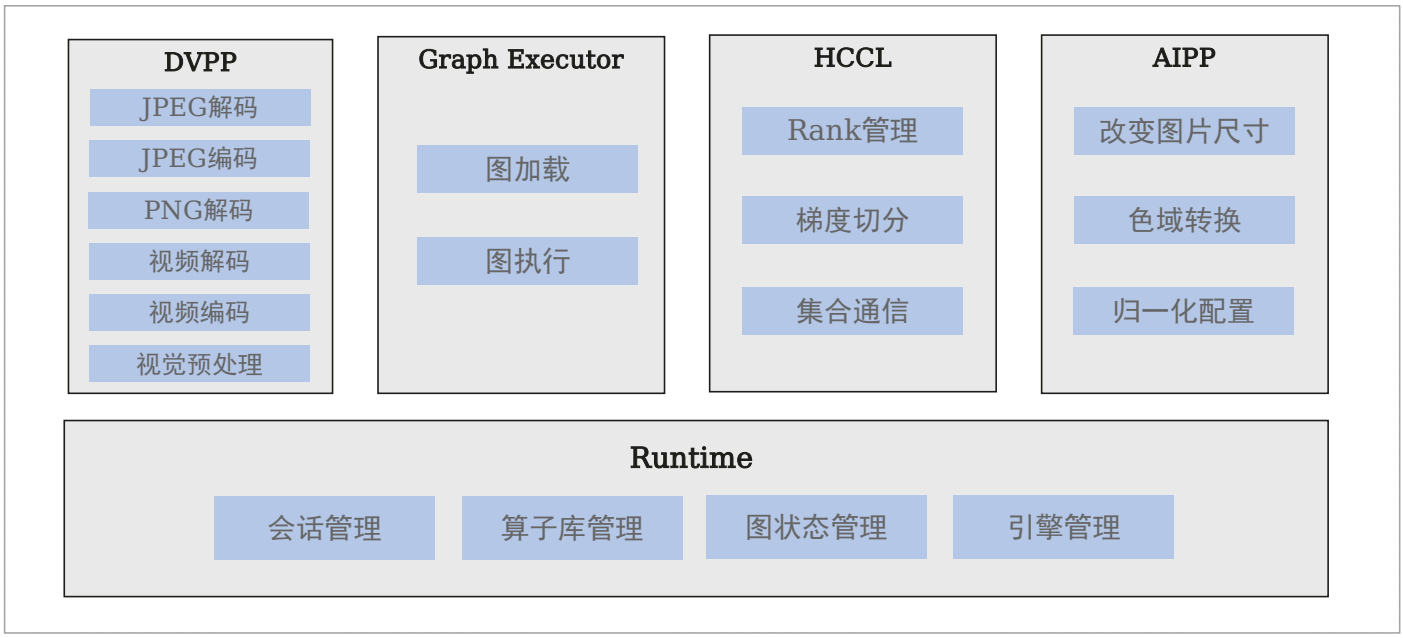
HUAWEI

昇腾计算编译层



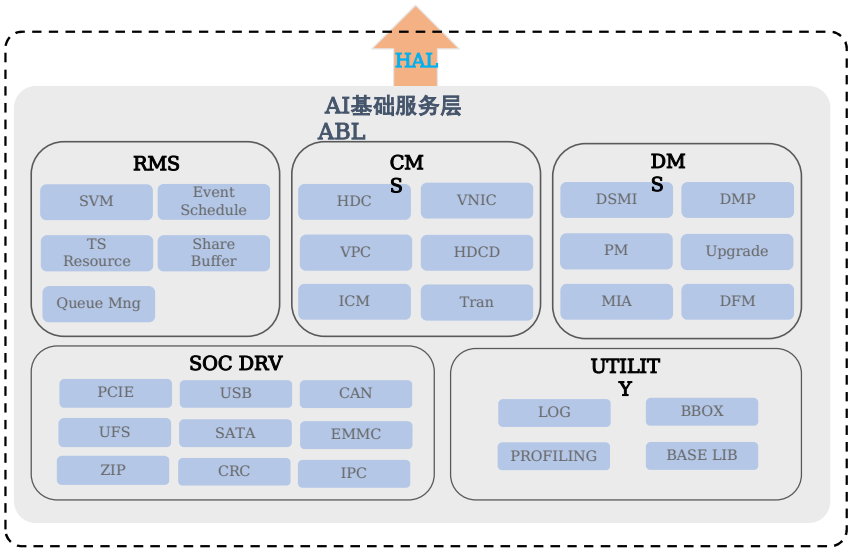
HUAWEI

昇腾计算执行层



HUAWEI

昇腾计算基础层



五个组件：

RMS：资源管理，负责管理与调度昇腾设备的计算、显存等关键资源

CMS：通信管理，负责提供片内、片间高效通信

DMS：设备管理，负责对昇腾设备进行配置、切分、升级、故障检测等管理

DRV：芯片IP驱动，负责使能硬件

UTILITY：公共服务，负责提供基础库和系统维测能力

五大竞争力：

高性能：微秒级确定性调度、数据零拷贝等技术打造高性能数据面

高可信：五道安全防线构建昇腾解决方案可信底座

归一化：一套架构-接口-代码支撑多芯、多板、多场景

弹性：端/边/云灵活适应，虚机/容器/裸金属快速部署，算力细粒度按需切分**开源开放**：Ascend社区、Linux社区开源两步走，构建昇腾基础生态

HUAWEI

讲授内容：华为CANN编程（二）

- 昇腾AI基础软硬件平台
- 昇腾AI异构计算架构CANN
- CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

核心能力一：模型迁移&训练

全面支持业界主流模型，完善工具包实现模型下载即用

针对训练场景下模型开发，CANN提供完善的工具包，提升开发和训练效率

开源模型 **下载** → 自动化模型 **迁移** → 高度工具化 **调测** → 智能性能 **调优** → 在昇腾上 **运行**

ModelZoo
适配多款业界主流AI框架

网络迁移工具
来自不同AI框架的模型，可自动迁移转化，无需手工修改代码

精度比对工具
一键整网数据比对
高效排查精度误差

AOE工具
自动调优算子和模型

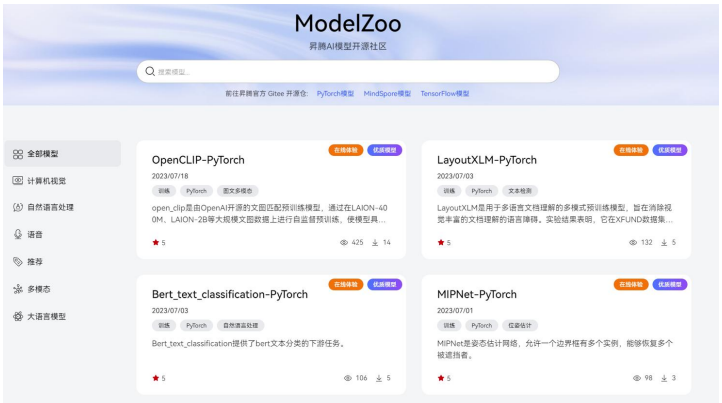
预置优化库
开机即获得最优性能

HUAWEI

核心能力一：模型迁移&训练

模型获取 → 模型迁移 → 精度调优 → 性能调优 → 固化运行

可通过昇腾AI模型开源社区ModelZoo等途径，获取针对不同应用场景、来自不同主流框架的AI模型



链接：

https://www.hiascend.com/software/modelzoo

视觉分割　目标检测　自然语言处理　……

昇思 MindSpore　PyTorch　TensorFlow

……

CANN　通过CANN异构计算架构，可以将来自不同框架的开源模型迁移至昇腾AI处理器执行训练，释放硬件澎湃算力

HUAWEI

将基于PyTorch的训练脚本迁移到昇腾AI处理器上执行训练，共有自动迁移、工具迁移、手工迁移三种方式：

◆ 自动迁移（推荐方式）

仅需开发者在训练脚本中引入转换库，在训练脚本运行时，会自动将PyTorch训练脚本中的CUDA接口更换为昇腾AI处理器支持的接口，再进行训练，操作简洁。

◆ 工具迁移

通过脚本迁移工具，自动将原始训练脚本中的CUDA接口替换为昇腾AI处理器支持的接口，生成新的训练脚本及迁移报告，训练时直接运行转换后的新脚本。

◆ 手工迁移

需要开发者对训练脚本、NPU的接口支持情况、GPU与NPU的代码异同点等均有一定了解，然后手动进行训练脚本的代码修改，使它能在昇腾AI处理器上成功执行训练。

（本方法不展开做讲解，可自行学习：
https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/70RC1alpha001/ptmoddevg/ptmigr/ptmigr_0011.html）

HUAWEI

模型获取 模型迁移 精度调优 性能调优 固化运行

核心能力一：模型迁移&训练

◆ **PyTorch模型迁移——自动迁移（推荐方式）**

1、在训练脚本中添加：

```
from torch_npu.contrib import
transfer_to_npu
```

2、执行训练脚本时，会自动将其中不支持的接口替换，再进行训练：

```
# torch.cuda.*
patch_cuda()

# torch.*
device_wrapper(torch, torch_fn_white_list)

# torch.Tensor.*
device_wrapper(torch.Tensor, torch_tensor_fn_white_list)
torch.Tensor.cuda = torch.Tensor.npu
torch.cuda.DoubleTensor = torch.npu.FloatTensor

# torch.nn.Module.*
device_wrapper(torch.nn.Module, torch_module_fn_white_list)
torch.nn.Module.cuda = torch.nn.Module.npu

# torch.distributed.init_process_group
torch.distributed.init_process_group = wrapper_hccl(torch.distributed.init_process_group)
```

①该方法仅支持PyTorch1.8.1以上版本

②可以通过查看脚本运行后是否生成了权重文件（.pth.tar），来判断模型是否迁移训练成功

HUAWEI

◆ PyTorch模型迁移——工具迁移

1、安装依赖：

```
pip3 install pandas
pip3 install libcst
pip3 install jedi
```

2、进入迁移工具所在路径：

```
cd CANN软件安装目录/ascend-toolkit/latest/tools/ms_fmk_transplt/
```

3、执行脚本迁移任务：

```
./pytorch_gpu2npu.sh -i 原始脚本路径 -o 脚本迁移结果输出路径 -v 原始脚本框架版本 [-r 自定义规则json文件路径] [-s] [-sim] [-m] [distributed -t 目标模型变量名 -m 训练脚本的入口文件]
```

必选参数：
-i：要进行迁移的原始脚本文件所在文件夹路径
-o：脚本迁移结果文件输出路径
-v：待迁移脚本的PyTorch版本

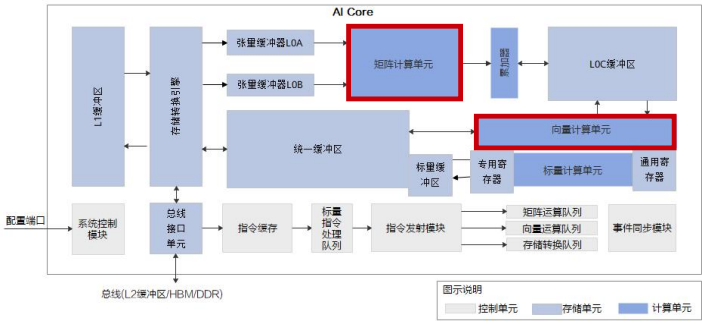
4、进入脚本迁移结果文件输出路径，查看结果文件：

```
├── xxx_msft/xxx_msft_multi     // 迁移结果输出目录
│   ├── 训练脚本文件存储目录    // 与迁移前目录结构一致
│   ├── msFmkTranspltlog.txt    // 脚本迁移过程日志文件
│   ├── cuda_op_list.csv        //分析出的cuda算子列表
│   ├── unknown_api.csv         //支持情况存疑的API列表
│   ├── unsupported_api.csv     //不支持的API列表
│   ├── change_list.csv         // 修改记录文件
│   ├── run_distributed_npu.sh  // 多卡启动shell脚本
│   ├── ascend_function         // 如果启用了自动替换不支持API功能，生成包含等价算子的目录
```

模型获取 | 模型迁移 | 精度调优 | 性能调优 | 固化运行

核心能力一：模型迁移&训练

自 动 混 合 精 度



- 昇腾处理器计算核心中的矩阵计算单元CUBE，仅支持FP16，所以：
  ①对于FP32的矩阵运算会调用向量计算单元Vector，性能差距明显；
  ②卷积算子由于Vector不支持会强制走FP16，可能带来梯度消失的问题；

- 由于PyTorch的默认浮点数精度为FP32，因此在迁移完成后、训练开始前，需开启混合精度。

1、导入torch_npu中的AMP模块：

```
from torch_npu.npu import amp
```

2、在模型、优化器定义之后，定义AMP功能中的GradScaler：

```
model = CNN().to(device)
train_dataloader = DataLoader(train_data, batch_size=batch_size)
# 定义DataLoader
loss_func = nn.CrossEntropyLoss().to(device)    # 定义损失函数
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)    # 定义优化器
scaler = amp.GradScaler()    # 在模型、优化器定义之后，定义GradScaler
```

3、在训练代码中添加AMP功能相关的代码开启AMP：

```
for epo in range(epochs):
    for imgs, labels in train_dataloader:
        imgs = imgs.to(device)
        labels = labels.to(device)
        with amp.autocast():
            outputs = model(imgs)    # 前向计算
            loss = loss_func(outputs, labels)    # 损失函数计算
        optimizer.zero_grad()
        # 进行反向传播前后的loss缩放、参数更新
        scaler.scale(loss).backward()    # loss缩放并反向传播
        scaler.step(optimizer)    # 更新参数（自动unscaling）
        scaler.update()    # 基于动态Loss Scale更新loss_scaling系数
```

精度调优

为什么要进行精度调优？

- 同一模型从GPU(或CPU)移植到NPU中可能存在精度下降问题
- 同一模型进行迭代(模型、算子或设备迭代)时可能存在精度下降问题

如何定位精度问题？

- 对比NPU芯片中的算子计算数值与GPU(CPU)芯片中的算子计算数值，进行问题定位
- 对比相同模型在迭代前后版本的算子计算数值，进行问题定位

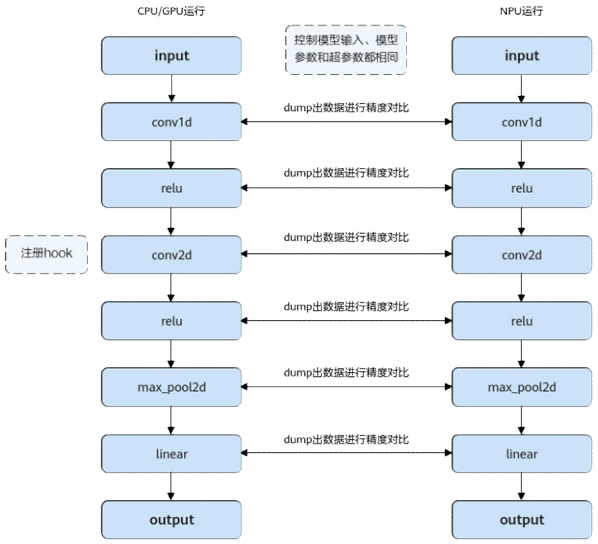
在同一模型或算子调试过程中，开发者定位算子相关的计算精度问题往往费时费力。因此CANN推出了**精度比对工具ptdbg_ascend**，通过在PyTorch模型中注入hook，帮助开发者高效排查存在的计算精度误差，帮助精准定位问题、分析原因并进行优化调整。

HUAWEI

模型获取 模型迁移 精度调优 性能调优 固化运行

核心能力一：模型迁移&训练

精度调优

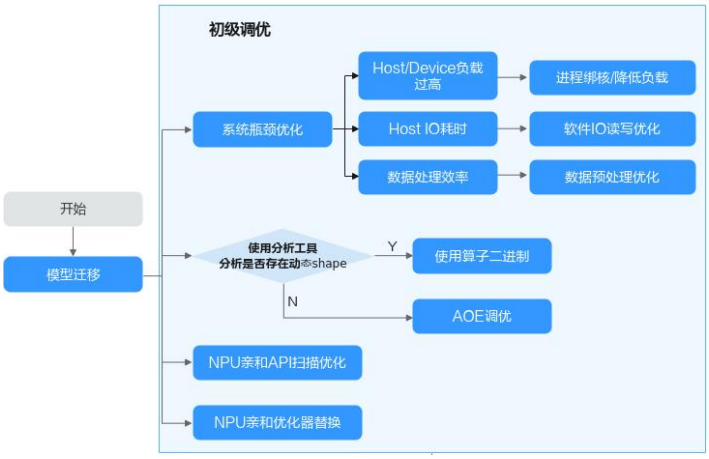


ptdbg_ascend通过在模型中注入hook，跟踪对比计算图中算子的前向传播与反向传播时的输入与输出。

```
from torch_npu.hooks import set_dump_path, seed_all,
register_acc_cmp_hook
from torch_npu.hooks.tools import compare

# 对该网络进行hook注入和数据dump
module = ModuleOp()
register_acc_cmp_hook(module) # 对模型注入forwar和backward的hooks
seed_all()
x = torch.randn(2, 2)

# cpu上计算，dump数据
set_dump_path("./cpu_module_op.pkl")
out = module(x)
loss = out.sum()
loss.backward()

# npu上计算，dump数据
set_dump_path("./npu_mpdule_op.pkl")
module.npu()
x = x.npu()
out = module(x)
loss = out.sum()
loss.backward()

# 对比dump出的数据精度，生成csv文件
compare("./npu_module_op.pkl", "./cpu_module_op.pkl",
"./module_result.csv")
```

初级性能调优

模型迁移过后，如果存在性能不达标问题，需进行性能调优，CANN提供了各种调优工具，帮助开发者快速完成模型性能提升。



需要优化性能时，可以先并行使用以下通用调优方法进行初级调优：

- 系统瓶颈调优：通过Linux系统提供的top、perf和hdparm等工具来定位相应的系统资源瓶颈，基于实际问题通过降低负载、进程绑核、数据处理优化、软件IO读写优化、升级高性能硬件、数据预处理优化等方式进行优化
- PyTorch Analyse分析工具：存在动态shape使用算子二进制调优，不存在动态shape使用AOE调优
- NPU亲和API扫描：将原生API调用手动替换为指定的亲和API以提高模型性能
- NPU亲和优化器替换：针对部分优化器做昇腾AI处理器亲和性优化修改，使性能提升

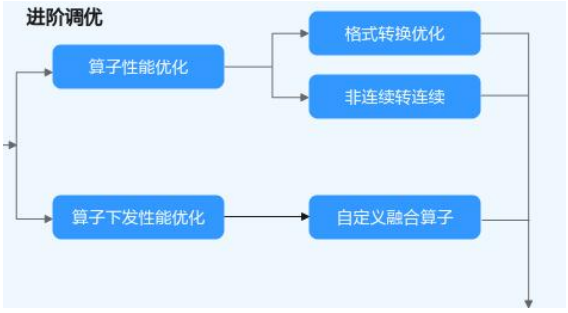
HUAWEI

模型获取 模型迁移 精度调优 性能调优 固化运行

核心能力一：模型迁移&训练

进阶性能调优

初阶调优完成之后，再次执行模型训练，并使用Profiler工具进行数据采集和分析，评估训练性能是否达标，若性能达标则调优结束，若性能未达标则考虑在基本调优的基础上进行进阶调优。



根据对训练数据的采集和分析，定位性能问题：

- 算子性能问题
  - ✓ 格式转换优化
  - ✓ 非连续转连续化
- 算子下发慢问题
  - ✓ 自定义算子融合

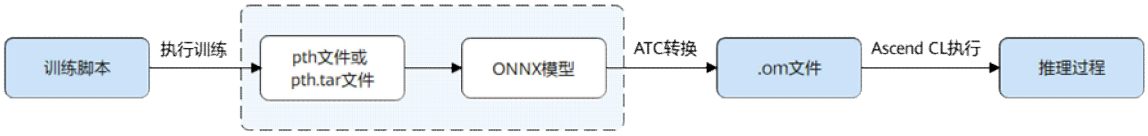
如果通过以上方式皆无法使性能达标，则可以联系华为工程师求助

HUAWEI

模型获取 | 模型迁移 | 精度调优 | 性能调优 | 固化运行

核心能力一：模型迁移&训练

模型训练完成后，用户可以通过PyTorch提供的接口保存模型文件（pth文件和pth.tar文件）用于在线推理；或将模型导出ONNX模型，然后通过ATC工具将其转换为适配昇腾AI处理器的.om文件用于离线推理。



在线推理：

在AI框架内执行推理，相比于离线推理场景，可以快速将基于PyTorch框架做推理的应用运行在昇腾AI处理器上，无需进行模型转换，适用于数据中心推理场景。

离线推理：

使用AI框架训练好的模型，通过ATC工具将其转换成昇腾AI处理器支持的离线模型，然后加载并执行推理。相较于在线推理场景，模型运行速度更快，可以部署在更小的设备上。

HUAWEI

讲授内容：华为CANN编程（二）

- 昇腾AI基础软硬件平台
- 昇腾AI异构计算架构CANN
- CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

推理应用开发

提供推理全栈工具，助力开发者实现推理应用

针对推理业务开发流程，CANN提供完备的工具链和统一接口，提升开发和运行效率

1.模型压缩 → 2.模型编译优化 → 3.开发部署

AMCT
通过量化压缩，实现模型的瘦身

ATC & AOE
完成模型编译，支持自动调优

AscendCL
统一开发接口，调用多样算力

HUAWEI

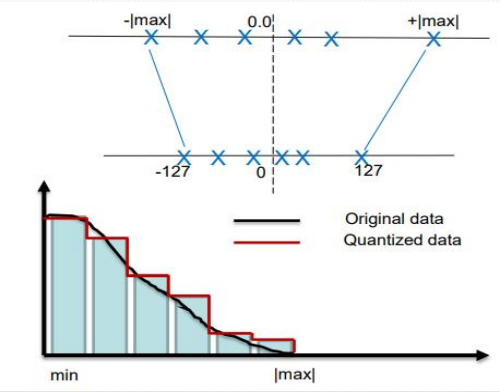
推理应用开发

AMCT模型压缩——量化

提供模型压缩工具AMCT，支持量化、通道稀疏、张量分解，降低模型的数据量和计算量，提升计算性能

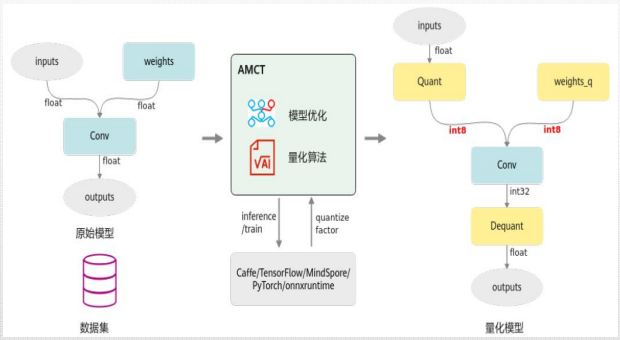
什么是量化：
将模型的权重（weight）和数据（activation）从浮点转换为整型，生成更加轻量化的网络模型，减少数据和计算量。

量化计算原理：
将数据从浮点域映射到整形域



主要量化特性：
- 支持对多种主流框架的模型进行量化
- 支持训练后量化、量化感知训练
- 支持前端框架输出的QAT模型
- 支持基于精度的自动量化
- 提供命令行方式和python API方式

AMCT量化处理流程：



HUAWEI

推理应用开发

AMCT模型压缩——张量分解和稀疏

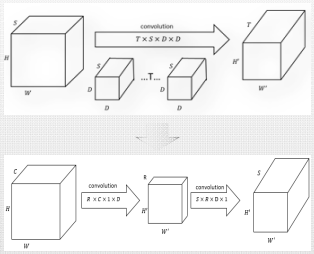
提供模型压缩工具AMCT，支持量化、通道稀疏、张量分解，降低模型的数据量和计算量，提升计算性能

什么是张量分解：
分解卷积tensor，将一个大卷积转化成两个级联的小卷积，从而降低存储空间和计算量。张量分解处理后，需要重训练以保证模型精度。

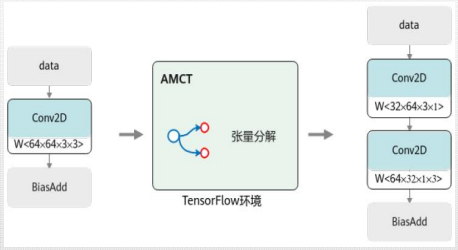
张量分解特性：
- 针对昇腾AI处理器架构，选择亲和的卷积层进行分解
- 支持TensorFlow/PyTorch/Caffe等前端框架

什么是稀疏：
通过结构剪枝，对模型中的部分算子裁剪部分权重和参数，从而得到参数量和计算量更小的网络模型。稀疏需结合重训练处理以保证模型精度。

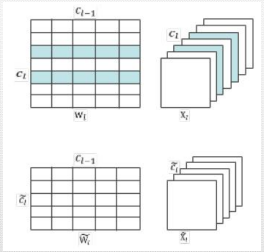
稀疏特性：
- 支持通道稀疏，裁剪掉重要性相对较低的通道
- 支持TensorFlow/PyTorch等前端框架



张量分解原理



AMCT张量分解流程



稀疏计算原理

HUAWEI

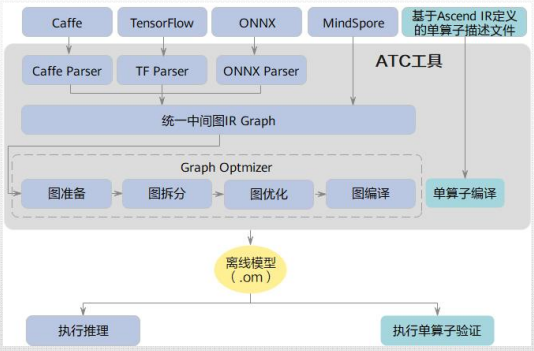
推理应用开发

ATC模型编译

提供模型转换工具ATC，支持将第三方框架模型转换成适配昇腾AI处理器的om模型

为什么需要ATC：
对于开源框架的网络模型（如Caffe、TensorFlow等），不能直接在昇腾AI处理器上做推理，需要先使用ATC工具将开源框架的网络模型转换为适配昇腾AI处理器的om模型。

功能架构：



主要特性：
- 在模型推理场景下，支持将第三方框架模型转换成适配昇腾AI处理器的om模型

```
atc --framework=0 --soc_version=${soc_version}
--model=$HOME/mod/resnet50.prototxt
--weight=$HOME/mod/resnet50.caffemodel
--output=$HOME/module/out/caffe_resnet50
```

- 在单算子执行场景下，支持将Ascend IR定义的单算子描述文件(json格式)转换为适配昇腾AI处理器的om模型

```
atc --singleop=$HOME/singleop/gemm.json --
output=$HOME/singleop/out/op_model --
soc_version=${soc_version}
```

- 支持模型的自动调优，进行算子调度优化、权重数据重排、内存使用优化等操作

HUAWEI

AOE智能调优

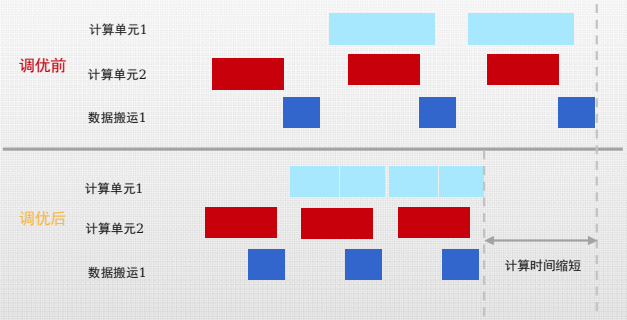
提供智能调优工具AOE，支持算子计算过程的自动寻优，提升整网计算性能

什么是自动调优：

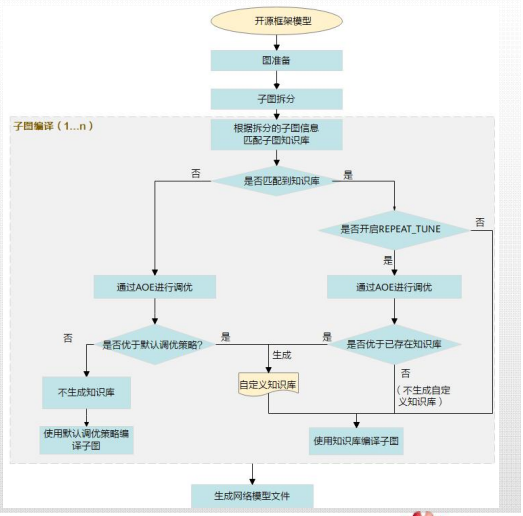
对算子计算过程开展调度、切分的自动搜索寻优，将计算和搬运充分并行，从而提升硬件利用率和网络计算性能。

智能调优的主要特性：

- 支持整网调优（集成到ATC工具）、支持单算子调优
- 支持并行加速
- 调优成果作为知识库固化，并支持优化知识库的迁移与合并



AOE智能调优流程



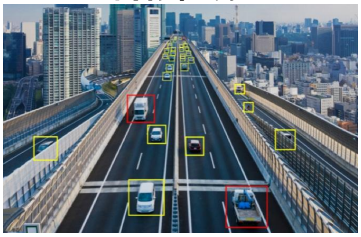
HUAWEI

推理应用开发

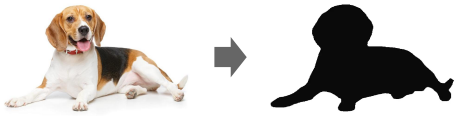
推理应用开发

在深度神经网络模型训练优化完成之后，我们可以基于此模型、使用ACL提供的API库开发深度神经网络应用，用于实现目标检测、图像分类等功能。

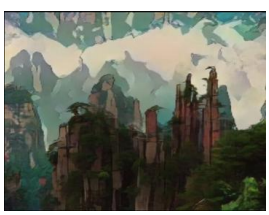
目标检测



图像分割



图像卡通化



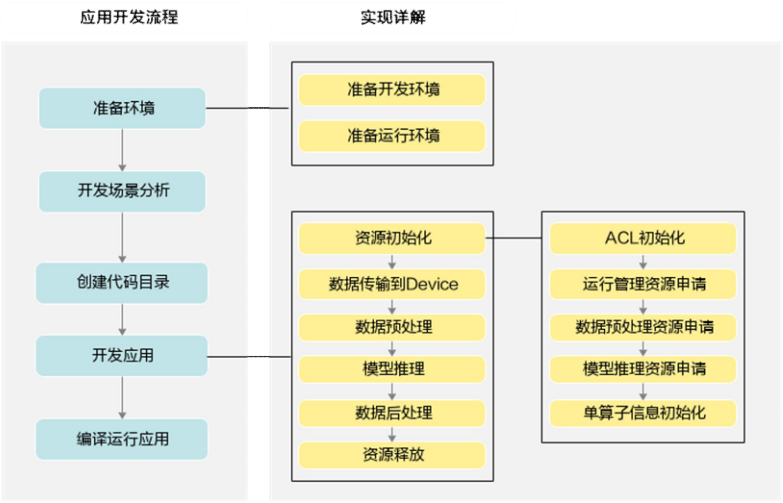
HUAWEI

推理应用开发流程

一．准备环境：
包括开发环境和运行环境。

二．开发场景分析：
根据开发场景分析涉及哪些功能（例如，数据传输、模型推理等）的开发，确定功能后，再明确涉及的命令或接口。

三．创建代码目录：
在开发应用前，您需要先创建目录，存放代码文件、编译脚本、测试图片数据、模型文件等。



四．开发应用：

1. 资源初始化：包括ACL初始化、运行管理资源申请、数据预处理资源申请、模型推理资源申请、单算子信息初始化等。
2. 将数据从Host传输到Device。
3. 数据传输到Device后，若需要抠图、缩放等操作，还需要进行数据预处理，输出YUV420 SP格式的图片，作为模型推理的输入。
4. 执行模型推理。
5. 若需要处理模型推理的结果，还需要进行数据后处理，例如对于图片分类应用，通过数据后处理从推理结果中查找最大置信度的类别标识。请参见数据后处理（调用单算子），回传结果到Host。
6. 所有数据处理结束后，需及时释放运行管理资源，再调用acl.finalize接口实现ACL去初始化。

五．编译运行应用：包括模型转换、运行应用

HUAWEI

推理应用开发---目录创建

在开发应用前，需要先创建目录，存放代码文件、编译脚本、测试图片数据、模型文件等。如下仅作为参考实例：

```
├App名称
├── model            // 该目录下存放模型文件
|   ├── xxxxxx

├── data
|   ├── xxx.jpg      // 测试数据

├── inc              // 该目录下存放声明函数的头文件
|   ├── xxx.h

├── out              // 该目录下存放输出结果

├── src
|   ├── xxx.json     // 系统初始化的配置文件
|   ├── CMakeLists.txt // 编译脚本
|   ├── xxx.cpp      // 实现文件
```

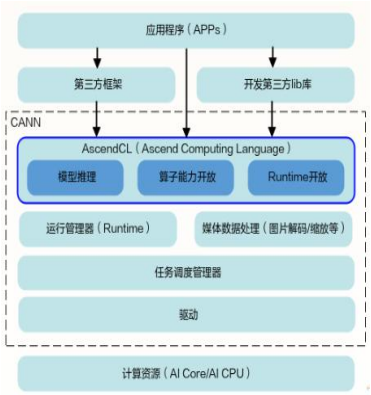
在进行推理应用开发前，可以参考学习CANN样例仓中的实际样例，例如，在下方图片分类应用的文件目录结构中，并未出现.h和.json文件，出现了用于图片预处理的.py文件，因此具体的目录及文件需要根据实际的应用场景和开发内容来确定。

```
resnet50_firstapp
├── data
|   ├── dog1_1024_683.jpg              // 测试图片，需按下文的指导获取图片，放到该目录下

├── model
|   ├── resnet50.caffemodel            // ResNet-50网络的预训练模型文件（*.caffemodel）
|                                      // 需按下文的指导获取图片，放到该目录下
|   ├── resnet50.prototxt              // ResNet-50网络的模型文件（*.prototxt）
|                                      // 需按下文的指导获取图片，放到该目录下

├── script
|   ├── transferPic.py                 // 将测试图片预处理为符合模型要求的图片
|                                      // 包括将*.jpg转换为*.bin，同时将图片从1024*683的分辨率缩放为224*224

├── src
|   ├── CMakeLists.txt                 // 编译脚本
|   ├── main.cpp                       // 主函数，图片分类功能的实现文件
```

https://gitee.com/ascend/samples/tree/master/cplusplus/level2_simple_inference/1_classification/resnet50_firstapp

HUAWEI

推理应用开发---应用开发接口AscendCL



AscendCL（Ascend Computing Language）是一套用于在昇腾平台上开发深度神经网络推理应用的C语言API库，提供包括：

- Device管理
- Context管理
- Stream管理
- 内存管理
- 模型加载与执行
- 算子加载与执行
- 媒体数据处理

等C语言API库供用户开发深度神经网络应用，用于实现目标检测、图像分类等功能。用户可以直接调用AscendCL提供的接口开发应用、可以通过第三方框架调用ACL接口、可以使用ACL封装实现第三方lib库。

- Host：Host指与Device相连接的X86服务器、ARM服务器，会利用Device提供的NN（Neural-Network ）计算能力，完成业务。
- Device：Device指安装了昇腾AI处理器的硬件设备，为Host提供NN计算能力。若存在多个Device，多个Device之间的内存资源不能共享。
- Context：管理了所有对象的生命周期（包括Stream、Event、设备内存等），不同Context的Event是完全隔离的，无法建立同步等待关系。
- Stream：用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序在Device上执行。基于Stream的kernel执行和数据传输能够实现Host运算操作、Host与Device间的数据传输、Device内的运算并行。
- Event：支持调用AscendCL接口同步Stream之间的任务，包括同步Host与Device之间的任务、同一个Device上的多个任务。

HUAWEI

推理应用开发---代码框架

资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

```
int main()
{
    // 1.定义一个资源初始化的函数，用于AscendCL初始化、运行管理资源申请（指定计算设备）
    InitResource();
    // 2.定义一个模型加载的函数，加载图片分类的模型，用于后续推理使用
    const char *modelPath = "../model/resnet50.om";
    LoadModel(mdoelPath);
    // 3.定义一个读图片数据的函数，将测试图片数据读入内存，并传输到Device侧，用于后续推理使用
    const char *picturePath = "../data/dog1_1024_683.bin";
    LoadPicture(picturePath);
    // 4.定义一个推理的函数，用于执行推理
    Inference();
    // 5.定义一个推理结果数据处理的函数，用于在终端上屏显测试图片的top5置信度的类别编号
    PrintResult();
    // 6.定义一个模型卸载的函数，卸载图片分类的模型
    UnloadModel();
    // 7.定义一个函数，用于释放内存、销毁推理相关的数据类型，防止内存泄露
    UnloadPicture();
    // 8.定义一个资源去初始化的函数，用于AscendCL去初始化、运行管理资源释放（指定计算设备）
    DestroyResource();
}
```

参考代码：
https://gitee.com/ascend/samples/blob/master/cplusplus/level2_simple_inference/1_classification/resnet50_firstapp/src/main.cpp

推理应用开发---资源初始化

资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

使用AscendCL接口开发应用时，必须先调用aclInit接口进行AscendCL初始化，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。
申请运行管理资源时，需按照Device、Context、Stream的顺序依次申请，其中，创建Context、Stream的方式分为隐式创建和显式创建，适用场景有所不同：
- 隐式创建Context和Stream：适合简单、无复杂交互逻辑的应用，但缺点在于，在多线程编程中，每个线程都使用默认Context或默认Stream，默认Stream中任务的执行顺序取决于操作系统线程调度的顺序。
- 显式创建Context和Stream：**推荐显式**，适合大型、复杂交互逻辑的应用，且便于提高程序的可读性、可维护性。

```
// AscendCL初始化、运行管理资源申请（指定计算设备）
void InitResource()
{
        aclError ret = aclInit(nullptr);
        ret = aclrtSetDevice(deviceId_);
}
```

aclinit接口的入参是配置文件所在路径，在配置文件中可包含用于精度比对、性能分析的配置：

```
// 此处的..表示相对路径，相对可执行文件所在的目录，例如，编译出来的可执行文件存放在out目录下，此处的..就表示out目录的上一级目录
const char *aclConfigPath = "../src/acl.json";
aclError ret = aclInit(aclConfigPath);
```

该入门级应用无需配置文件，则传入nullptr

Device、Context、Stream申请：

```
aclrtSetDevice(deviceId);
aclrtCreateContext(context, deviceId);
aclrtCreateStream(stream);
```

该入门级应用不涉及复杂异步任务的管理，没有使用Context、Stream的场景，因此未进行显式创建

推理应用开发---数据传输&处理

资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

数据传输的关键接口调用流程：
1、申请内存：Host上内存申请可使用接口aclrtMallocHost，Device上内存申请可使用接口aclrtMalloc或aclrtMallocHost
2、将数据读入内存：由用户自行管理数据读入内存的实现逻辑
3、通过内存复制实现数据传输：同步内存复制使用接口aclrtMemcpy（对于Host内的数据传输、Device内的数据传输、Host与Device之间的数据传输，可以调用内存复制的接口实现，也可以直接通过指针传递数据。）

```
// 申请内存，使用C/C++标准库函数将测试图片读入内存
void ReadPictureTotHost(const char *picturePath)
{
        string fileName = picturePath;
        ifstream binFile(fileName,
ifstream::binary);
        binFile.seekg(0, binFile.end);
        pictureDataSize = binFile.tellg();
        binFile.seekg(0, binFile.beg);
        aclError ret =
aclrtMallocHost(&pictureHostData,
pictureDataSize);
        binFile.read((char*)pictureHostData,
pictureDataSize);
        binFile.close();
}
```

```
// 申请Device侧的内存，再以复制内存的方式将内存中的图片数据传输到Device
void CopyDataFromHostToDevice()
{
        aclError ret =
aclrtMalloc(&pictureDeviceData,
pictureDataSize,
ACL_MEM_MALLOC_HUGE_FIRST);
        ret =
aclrtMemcpy(pictureDeviceData,
pictureDataSize, pictureHostData,
pictureDataSize,
ACL_MEMCPY_HOST_TO_DEVICE);
}
```

调用函数将测试图片数据读入Host内存，再复制到Device内存，用于执行推理

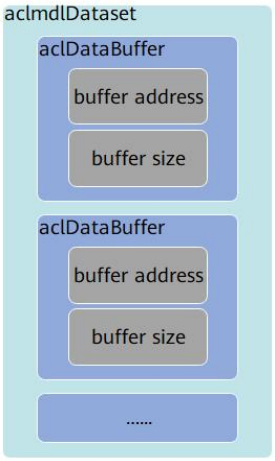
通过aclmdlLoadFromFile接口从文件加载离线模型数据

推理应用开发---模型推理

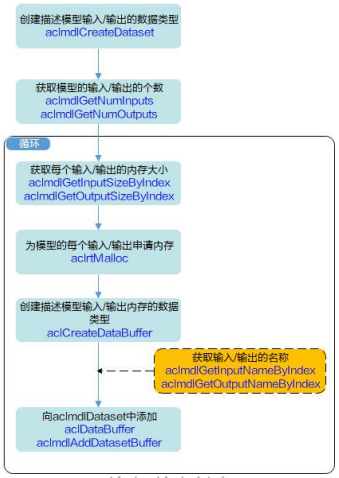
资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

在调用AscendCL接口进行模型推理前，需要先按照规定的数据类型准备输入和输出数据：

- 使用aclmdlDesc类型的数据描述模型基本信息，例如输入/输出的个数、数据类型、Format、维度信息等。
- 使用aclDataBuffer类型的数据来描述每个输入/输出的内存地址、内存大小。
- 使用aclmdlDataset类型的数据描述模型的输入、输出数据集。



aclmdDataset与aclDataBuffer



输出/输出创建

```
//参考代码中仅有一个输入和一个输出
aclmdlDataset *inputDataSet;
aclDataBuffer *inputDataBuffer;
aclmdlDataset *outputDataSet;
aclDataBuffer *outputDataBuffer;
aclmdlDesc *modelDesc;
```

当模型存在多个输入、输出时，向aclmdlDataset中添加aclDataBuffer时，为避免顺序出错，可以先获取输入、输出的名称，再根据输入、输出名称所对应的index顺序来添加。

HUAWEI

推理应用开发

推理应用开发---模型推理

资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

输入数据大小：测试图片数据大小

输出数据大小：未知，因此需要使用aclmdlGetDesc接口从模型描述中获取信息

准备符合要求的输入、输出数据之后，将模型ID、输入和输出传入aclmdlExecute接口执行推理

```
// 准备模型推理的输入数据结构
void CreateModelInput()
{
        // 创建aclmdlDataset类型的数据，描述模型推理的输入
        inputDataSet = aclmdlCreateDataset();
        inputDataBuffer = aclCreateDataBuffer(pictureDeviceData,
pictureDataSize);
        aclError ret = aclmdlAddDatasetBuffer(inputDataSet, inputDataBuffer);
}
// 准备模型推理的输出数据结构
void CreateModelOutput()
{
        // 创建并获取模型描述信息
        modelDesc =  aclmdlCreateDesc();
        aclError ret = aclmdlGetDesc(modelDesc, modelId);
        // 创建aclmdlDataset类型的数据，描述模型推理的输出
        outputDataSet = aclmdlCreateDataset();
        // 获取模型输出数据需占用的内存大小，单位为Byte
        outputDataSize = aclmdlGetOutputSizeByIndex(modelDesc, 0);
        // 申请输出内存
        ret = aclrtMalloc(&outputDeviceData, outputDataSize,
ACL_MEM_MALLOC_HUGE_FIRST);
        outputDataBuffer = aclCreateDataBuffer(outputDeviceData,
outputDataSize);
        ret = aclmdlAddDatasetBuffer(outputDataSet, outputDataBuffer);
}
```

```
// 执行推理
void Inference()
{
        CreateModelInput();
        CreateModelOutput();
        aclError ret =
aclmdlExecute(modelId, inputDataSet,
outputDataSet);
}
```

HUAWEI

推理应用开发---数据后处理

资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

模型推理的输出将存放在申请的内存中，由于示例模型的输出是测试图片属于1000种物体类别的置信度，全部打印出来过于冗长，可以通过自定义函数，只在终端打印测试图片前五置信度的类别编号。先通过aclrtMallocHost申请Host内存，再通过aclrtMemcpy将Device上的推理结果拷贝到Host，最后进行排序和显示。

```
// 在终端上屏显测试图片的top5置信度的类别编号
void PrintResult()
{
        aclError ret = aclrtMallocHost(&outputHostData, outputDataSize);
        ret = aclrtMemcpy(outputHostData, outputDataSize, outputDeviceData, outputDataSize,
ACL_MEMCPY_DEVICE_TO_HOST);
        float* outFloatData = reinterpret_cast<float *>(outputHostData);

        map<float, unsigned int, greater<float>> resultMap;
        for (unsigned int j = 0; j < outputDataSize / sizeof(float);++j)
        {
                resultMap[*outFloatData] = j;
                outFloatData++;
        }

        int cnt = 0;
        for (auto it = resultMap.begin();it != resultMap.end();++it)
        {
                if(++cnt > 5)
                {
                        break;
                }
                printf("top %d: index[%d] value[%lf] \n", cnt, it->second, it->first);
        }
}
```


推理应用开发---资源释放

资源初始化
数据传输&处理
模型推理
数据后处理
资源释放

①模型卸载

在模型推理结束后，需要通过aclmdlUnload接口卸载模型，并销毁aclmdlDesc类型的模型描述信息、释放模型运行的工作内存和权值内存。

```
// 卸载模型
void UnloadModel()
{
        aclmdlDestroyDesc(modelDesc);
        aclmdlUnload(modelId);
}
```

②内存释放

模型卸载后，需要释放推理过程中申请的输入输出内存。

```
// 释放内存、销毁推理相关的数据类型，防止内存泄露
void UnloadPicture()
{
        aclError ret = aclrtFreeHost(pictureHostData);
        pictureHostData = nullptr;
        ret = aclrtFree(pictureDeviceData);
        pictureDeviceData = nullptr;
        aclDestroyDataBuffer(inputDataBuffer);
        inputDataBuffer = nullptr;
        aclmdlDestroyDataset(inputDataSet);
        inputDataSet = nullptr;

        ret = aclrtFreeHost(outputHostData);
        outputHostData = nullptr;
        ret = aclrtFree(outputDeviceData);
        outputDeviceData = nullptr;
        aclDestroyDataBuffer(outputDataBuffer);
        outputDataBuffer = nullptr;
        aclmdlDestroyDataset(outputDataSet);
        outputDataSet = nullptr;
}
```

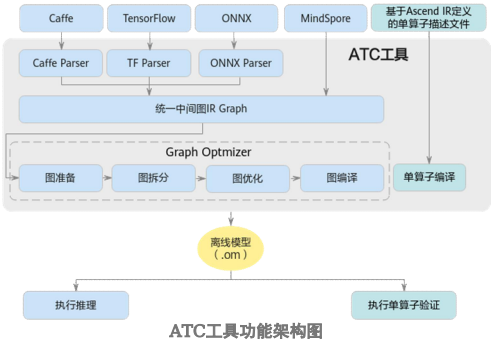
③运行资源释放&AscendCL去初始化

所有数据处理结束之后，需要一次释放运行管理资源。在确定完成了AscendCL的所有调用之后，或者进程退出之前，需调用aclFinalize接口实现AscendCL去初始化。

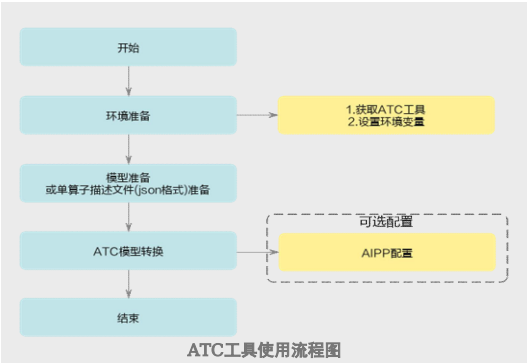
```
// AscendCL去初始化、运行管理资源释放（指定计算设备）
void DestroyResource()
{
        aclError ret = aclrtResetDevice(deviceId_);
        aclFinalize();
}
```

推理应用开发---编译运行应用

ATC工具：将开源框架的网络模型（如Caffe、TensorFlow等）以及单算子Json文件，转换成昇腾AI处理器支持的离线模型。转换过程主要包括模型读取，算子映射，算子融合，图优化等步骤。在过程中可以实现算子调度的优化、权值数据重排、内存使用优化等，可以脱离设备完成模型的预处理。



ATC工具功能架构图



ATC工具使用流程图

ATC工具使用流程：

- 使用ATC工具之前，请先在开发环境安装ATC软件包。
- 准备要进行转换的模型或单算子描述文件，并上传到开发环境。
- 使用ATC工具进行模型转换，在配置相关参数时，根据实际情况选择是否进行AIPP配置。

① 开源框架网络模型经过Parser解析后，转换为中间态IR Graph。

② 中间态IR经过图准备，图拆分，图优化，图编译等一系列操作后，转成适配昇腾AI处理器的离线模型。

③ 转换的离线模型上传到板端环境，通过AscendCL接口加载模型实现推理。也可以将开源框架网络模型转好的离线模型转json文件查看，也可以直接将开源框架网络模型通过ATC转json文件。

④ Ascend IR定义的单算子描述文件（json格式）通过ATC工具进行单算子编译后，转成适配昇腾AI处理器的单算子离线模型，然后上传到板端环境，通过AscendCL接口加载单算子模型文件用于验证单算子功能。

HUAWEI

推理应用开发

推理应用开发---编译运行应用

模型放入model目录

执行模型转换

测试图片放入data目录

创建编译脚本

设置环境变量

启动编译并生成可执行文件

运行应用

执行以下命令进行模型转换，将原始模型转换为昇腾AI处理器能识别的*.om模型文件（以Caffe框架模型为例）
atc --model=model/*.prototxt --weight=model/*.caffemodel --framework=0 --output=model/* --soc_version=Ascend310
--model：原始网络模型文件路径与文件名（*.prototxt）
--weight：原始网络模型权重文件路径与文件名（*.caffemodel）
--framework：原始网络模型框架类型（0: Caffe, 1: MindSpore, 3: TensorFlow, 5:ONNX）
--output：存放转换后的离线模型的路径以及文件名
-- soc_version：指定模型转换时昇腾AI处理器的版本

样例提供编译脚本*CMakeLists.txt*，仅需修改参数，无需自己创建

export DDK_PATH=${INSTALL_DIR}，CANN软件安装目录
export NPU_HOST_LIB=${INSTALL_DIR}/{arch-os}/devlib，arch表示操作系统架构，{os}表示操作系统

cmake ../../../src -DCMAKE_CXX_COMPILER=g++ -DCMAKE_SKIP_RPATH=TRUE
“../../../src”表示CMakeLists.txt文件所在的目录

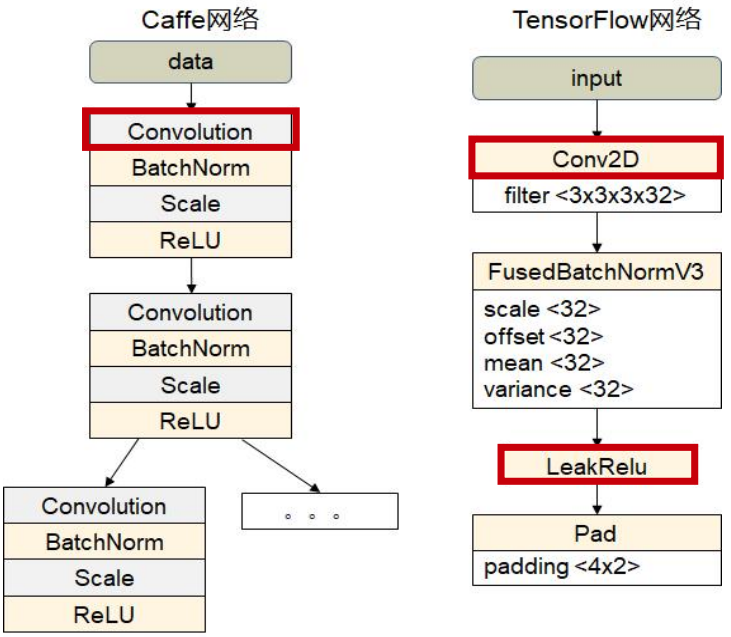
HUAWEI

讲授内容：华为CANN编程（二）

- 昇腾AI基础软硬件平台
- 昇腾AI异构计算架构CANN
- CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

什么是算子 — 算子在神经网络中的含义

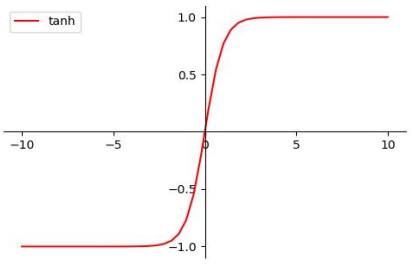
深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称OP）。在网络模型中，算子对应层或者节点的计算逻辑，通常网络模型都是由一个或多个算子拼接组成。



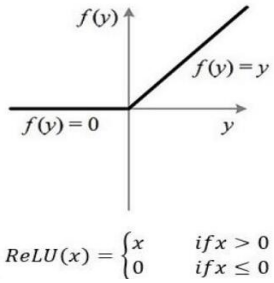
什么是算子 — 算子的数学含义

在数学领域，一个函数空间到函数空间上的映射O：X→Y，都称为算子。广义的讲，对任何函数进行某一项操作都可以认为是一个算子，比如微分算子，不定积分算子等。

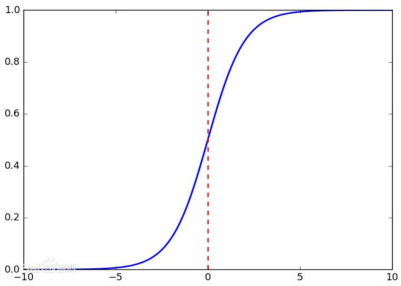
常见算子： tanh、ReLU、sigmoid等。



Tanh函数图像



ReLU函数图像



Sigmoid函数图像

什么场景需要开发自定义算子

一般情况下，开发者无需自己开发算子，但若遇到以下场景，开发者需要考虑进行自定义算子的开发：

- 训练场景或推理场景下，遇到了不支持的算子
- 网络调优时，发现某个算子性能较差，想重新开发一个高性能算子替换性能较差的算子
- 需要定制特殊算法功能的算子

例如，针对一个分类应用，我们想从分类模型的推理结果中查找可能性最大的前5个标识，则可以实现一个查找最大值的算子（例如ArgMax），后续就可以直接通过AscendCL接口调用此算子实现对推理结果的后处理。

算子基本概念 — 总览

进行算子开发前，首先需要了解算子相关的基本概念。

- **算子名称（Name）**

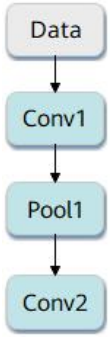
  算子名称用于标志网络中的某个算子，同一网络中算子的名称需要保持唯一。如右图所示Conv1，Pool1，Conv2都是此网络中的算子名称，其中Conv1与Conv2算子的类型为Convolution，表示分别做一次卷积运算。

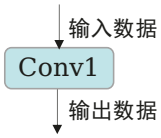
- **算子类型（Type）**

  网络中每一个算子根据算子类型进行算子实现的匹配，相同类型算子的实现逻辑相同。在一个网络中同一类型的算子可能存在多个，例如右图中名称为Conv1的算子与Conv2算子的类型都为Convolution。

- **数据容器（Tensor）**

  Tensor是存储算子输入数据与输出数据的容器，如右图所示，算子在网络中执行时，需要输入数据，算子执行完后，也会有对应的数据输出。这种承载算子数据的容器定义为张量（Tensor）。





HUAWEI

算子基本概念 — Tensor

张量（Tensor）是存储算子输入数据与输出数据的容器，而张量描述符（TensorDesc）是对输入数据与输出数据的描述，张量描述符的数据结构包含如下属性：

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需要保持唯一。
形状（shape）	Tensor的形状，比如（10,）或者（1024, 1024）或者（2, 3, 4）等。 形式：(i1, i2,...in)，其中i1到in均为正整数
数据类型（dtype）	指定Tensor对象的数据类型。 例如：float16, float32, int8, int16, int32, uint8, uint16, bool等。 不同计算操作支持的数据类型不同。
数据排布格式（format）	数据的物理排布格式，定义了解读数据的维度。

HUAWEI

算子基本概念 — Shape

下面分别介绍张量描述符中的形状和数据排布格式。

■ **形状（shape）**

张量的形状，以(D0, D1, ... ,Dn-1)的形式表示，D0到Dn是任意的正整数。

如形状(3,4)表示第一维有3个元素，第二维有4个元素，是一个3行4列的矩阵数组。

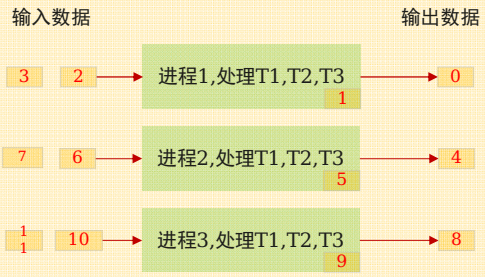
在形状的小括号中有多少个数字，就代表这个张量是多少维的张量。形状的第一个元素要看张量最外层的中括号中有几个元素，形状的第二个元素要看张量中从左边开始数第二个中括号中有几个元素，依此类推。

例如：

张量	形状
1	(0,)
[1,2,3]	(3,)
[[1,2],[3,4]]	(2,2)
[[[1,2],[3,4]], [[5,6],[7,8]]]	(2,2,2)

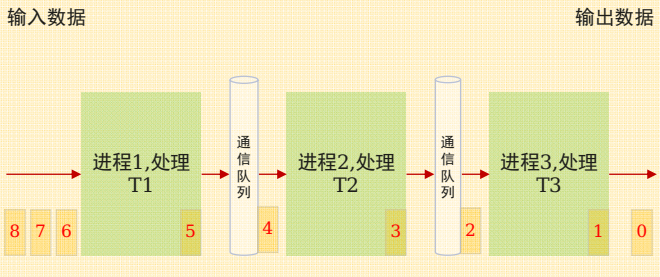
HUAWEI

并行计算中两种常见方法：单程序多数据（ SPMD ）和流水线并行



SPMD 数据并行计算原理

- 启动一组进程，它们运行相同的程序
- 把待处理数据切分，把切分后数据分片分发给不同进程处理
- 每个进程对自己的数据分片进行3个任务T1、T2、T3的处理



流水线并行原理

- 启动一组进程
- 对数据进行切分
- 每个进程都处理所有的数据切片，对输入数据分片只做一个任务的处理

HUAWEI

CANN算子是AI应用的基石

从算子到AI应用

算子库 → AI算法 → AI应用

算子开发过程

算子开发方式 **分析**	算子代码逻辑 **编写**	算子调度 **自动化**	算子编译 **优化**	在昇腾上 **运行**
分析算子数学逻辑 确定算子开发方式	Python/C++编码 简单易上手	Auto Schedule自动调度 降低开发难度	AOE智能调优 实现编译优化	

算子开发场景

模型中存在CANN不支持的算子	已有算子的计算逻辑不满足要求	已有算子的性能不满足要求

HUAWEI

已有算子的计算逻辑不满足要求

什么是Ascend C 算子

Ascend C是CANN针对算子开发场景推出的编程语言，通过**多层接口抽象、自动并行计算、孪生调试**等关键技术，极大提高算子开发效率，助力AI开发者低成本完成算子开发和模型调优部署。使用Ascend C编程语言开发的算子我们称之为Ascend C算子。

使用Ascend C开发自定义算子的优势：

- ✓ C/C++原语编程，最大化匹配用户的开发习惯
- ✓ 编程模型屏蔽硬件差异，编程范式提高开发效率
- ✓ 多层级API封装，从简单到灵活，兼顾易用与高效
- ✓ 孪生调试，CPU侧模拟NPU侧的行为，可优先在CPU侧调试

HUAWEI

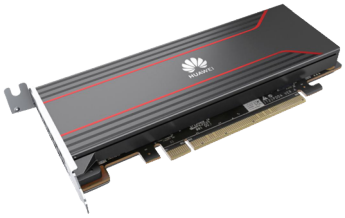
通过华为AI加速卡（NPU），可以实现大规模神经网络计算加速

Host

服务器

PCIE

Device



AI加速卡(NPU)

AICORE核心0 | AICORE核心1 | ...

GlobalMem(DDR内存)

AICORE是NPU卡的计算核心，NPU内部有多个AICORE。每个AICORE相当于多核CPU的一个核心

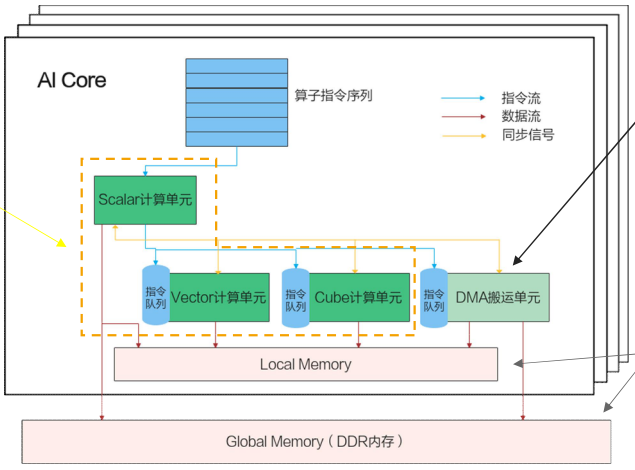
HUAWEI

AI Core内部并行计算架构抽象（1）

Ascend C 算子运行在AI Core上，AI Core是昇腾AI处理器中的**计算核心**。一个AI处理器内部有多个AI Core，AI Core中包含**计算单元、存储单元、搬运单元**等核心组件

计算单元包括了三种基础计算资源

- **Scalar计算单元：**执行地址计算、循环控制等标量计算工作，并把向量计算、矩阵计算、数据搬运、同步指令发射给对应单元执行
- **Cube计算单元：**负责执行矩阵运算
- **Vector计算单元：**负责执行向量运算



AI Core内部并行计算架构抽象示意图

搬运单元负责在Global Memory和Local Memory之间搬运数据，包含搬运单元MTE2（Memory Transfer Engine，数据搬入单元），MTE3（数据搬出单元）

存储单元为AI Core的内部存储，统称为Local Memory；
与此相对应，AI Core的外部存储称之为Global Memory；

HUAWEI

AI Core内部并行计算架构抽象（2）

异步指令流

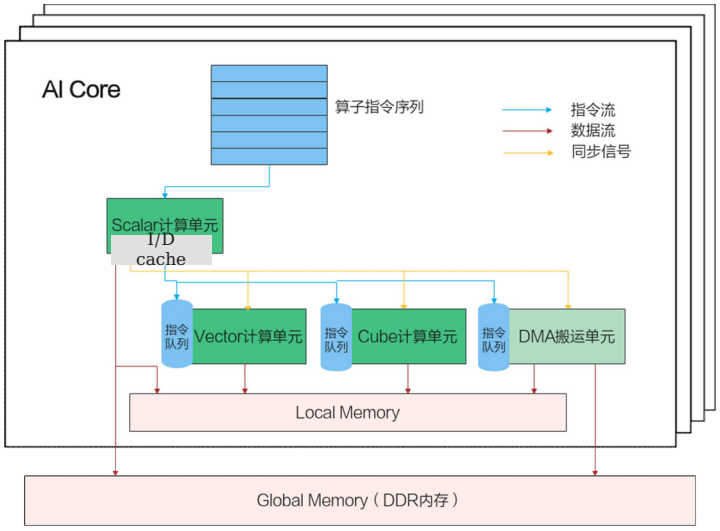
Scalar计算单元读取指令序列，并把向量计算、矩阵计算、数据搬运指令发射给对应单元的指令队列，向量计算单元、矩阵计算单元、数据搬运单元异步的并行执行接收到的指令

同步信号流

指令间可能会存在依赖关系，为了保证不同指令队列间的指令按照正确的逻辑关系执行，Scalar计算单元也会给对应单元下发同步指令

计算数据流

DMA搬入单元把数据搬运到Local Memory，Vector/Cube计算单元完成数据计算，并把计算结果写回Local Memory，DMA搬出单元把处理好的数据搬运回Global Memory

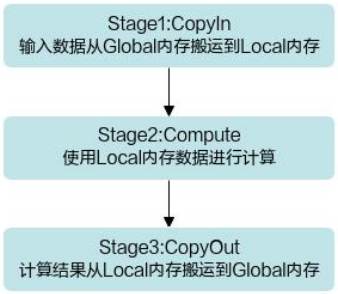


AI Core内部并行计算架构抽象示意图

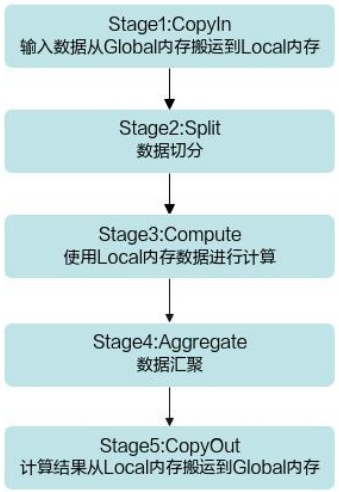
HUAWEI

编程范式——流水线式编程范式

Ascend C编程范式是一种流水线式的编程范式，把算子核内的处理程序，分成多个**流水任务**，通过队列（Queue）完成**任务间通信和同步**，并通过统一的**内存管理模块**（Pipe）管理任务间通信内存。



矢量编程基本任务设计



矩阵编程基本任务设计

HUAWEI

矢量编程 —— double buffer机制

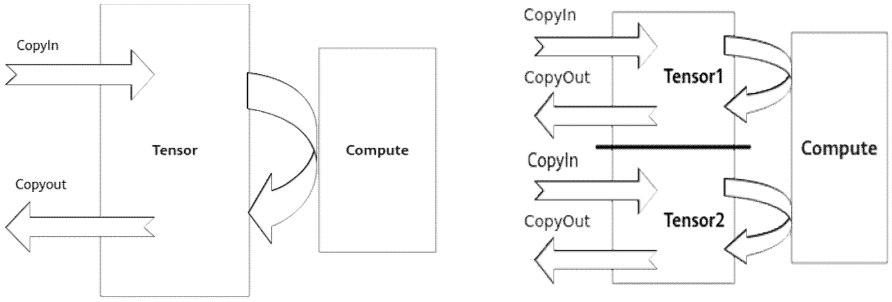
double buffer通过将**数据搬运与矢量计算并行执行**以隐藏数据搬运时间并降低矢量指令的等待时间，最终提高矢量计算单元的利用效率。

1个Tensor同一时间只能进行搬入、计算和搬出三个流水任务中的一个，其他两个流水任务涉及的硬件单元则处于Idle状态

如果将待处理的数据一分为二，比如Tensor1、Tensor2

- 当矢量计算单元对Tensor1进行Compute时，Tensor2可以执行CopyIn的任务
- 当矢量计算单元对Tensor2进行Compute时，Tensor1可以执行CopyOut的任务
- 当矢量计算单元对Tensor2进行CopyOut时，Tensor1可以执行CopyIn的任务

由此，数据的进出搬运和矢量计算之间实现并行，硬件单元闲置问题得以有效缓解



Double Buffer下的数据搬运与Vector计算过程

HUAWEI

多层级API封装

Ascend C算子采用标准C++语法和一组类库API进行编程，类库API主要包含以下几种：

- **计算类API**，包括标量计算API、向量计算API、矩阵计算API，分别实现调用Scalar计算单元、Vector计算单元、Cube计算单元执行计算的功能。
- **数据搬运API**，上述计算API基于Local Memory数据进行计算，所以数据需要先从Global Memory搬运至Local Memory，再使用计算接口完成计算，最后从Local Memory搬出至Global Memory。执行搬运过程的接口称之为数据搬移接口，比如DataCopy接口。
- **内存管理API**，用于分配管理内存，比如AllocTensor、FreeTensor接口。
- **任务同步API**，完成任务间的通信和同步，比如EnQue、DeQue接口。该类型API，开发者无需关注内部实现逻辑，使用简单的API接口即可完成。

Ascend C将API分为0-3级，随着级别增高，API使用的自由度降低，易用性增强。您可以根据需要选择合适的API，使用最通俗易懂的高级接口快速搭建算子逻辑，使用自由灵活的低级接口进行复杂的逻辑实现和性能调优。以矢量计算类API为例：

接口级别	接口说明
0级	功能灵活的计算API，充分发挥硬件优势，支持对每个操作数的Block stride，Repeat stride，Mask的操作。Block stride，Repeat stride，Mask参数的详细介绍请参见0级接口通用参数说明。
1级	slice计算API，解决多维数据中的切片计算问题。**该版本暂不支持1级接口。**
2级	针对源操作数的连续数据进行计算并连续写入目的操作数，解决一维tensor的连续计算问题。
3级	运算符重载，支持+, -, *, /, |, &, <, >, <=, >=, ==, !=，实现2级指令的简化表达。

HUAWEI

孪生调试

Ascend C提供**孪生调试**方法，即在cpu侧创建一个npu的模型并模拟它的计算行为，用来进行业务功能调试。相同的算子代码可以在cpu域调试精度，npu域调试性能。

CPU域调试

1、运行使能ASSERT，初步拦截算子指令或框架使用错误，如参数超限，指令数据地址重叠，该芯片不支持的指令等

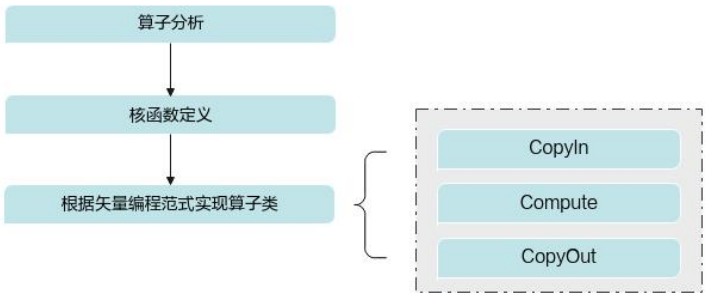
2、可使用**gdb**单步调试算子计算精度，也可以在代码中直接编写printf(...)来观察数值的输出。

NPU域调试

可使用工具获取真实芯片上profiling数据，进行性能精细调优

HUAWEI

Ascend C算子开发—算子开发流程



矢量算子实现流程

HUAWEI

矢量编程 —— 算子分析

算子类型（OpType）	Add			
算子输入	name	shape	data type	format
	x	(8, 2048)	half	ND
	y	(8, 2048)	half	ND
算子输出	z	(8, 2048)	half	ND
核函数名	add_custom			

■ **明确算子的数学表达式及计算逻辑**

Add算子的数学表达式为：$\vec{z} = \vec{x} + \vec{y}$，计算逻辑：输入数据需要先搬入到片上存储，然后使用计算接口完成两个加法运算，得到最终结果，再搬出到外部存储

■ **明确输入和输出**

Add算子有两个输入：$\vec{x}$ 与 $\vec{y}$，输出为$\vec{z}$。输入数据类型为*half*，输出数据类型与输入数据类型相同。输入支持固定shape(8，2048)，输出shape与输入shape相同。输入数据排布类型为*ND*

■ **确定算子实现所需接口**

涉及内外部存储间的数据搬运，使用数据搬移接口：*DataCopy*实现

涉及矢量计算的加法操作，使用矢量双目指令：*Add*实现

使用*LocalTensor*存放数据，使用Queue管理队列，会使用到*EnQue*（将Tensor放入队列）、*DeQue*（将Tensor从队列取出）等接口。

■ **确定核函数名称和参数**

自定义核函数名，如*add_custom*。根据输入输出，确定核函数有3个入参*x*，*y*，*z*

x，*y*为输入在Global Memory上的内存地址，*z*为输出在Global Memory上的内存地址

HUAWEI

Ascend C 算子样例详解

Ascend C算子开发—核函数定义

核函数（Kernel Function）是Ascend C算子设备侧实现的入口。在核函数中，需要为在一个核上执行的代码规定要进行的数据访问和计算操作，当核函数被调用时，多个核都执行相同的核函数代码，具有相同的参数，并行执行。

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

核函数写法比较固定，一般为**extern “C” __global__ __aicore__ void 自定义核函数名(输入1，输入2……，输出1，输出2……)** 的形式，

HUAWEI

Ascend C算子开发—核函数调用

核函数的调用语句是C/C++函数调用语句的一种扩展，使用内核调用符<<<...>>>这种语法形式，来规定核函数的执行配置。内核调用符仅可在NPU侧编译时调用，CPU侧编译无法识别该符号。

核函数的调用是异步的，核函数的调用结束后，控制权立刻返回给主机端，可以调用以下函数来强制主机端程序等待所有核函数执行完毕。

```
#ifndef __CCE_KT_TEST__
// call of kernel function
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y,
uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
#endif
```

HUAWEI

Ascend C 算子样例详解

Ascend C算子开发—算子实现类

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd(){}
    // 初始化函数，完成内存初始化相关操作
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z){
        // 初始化函数实现
    }
    // 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操作
    __aicore__ inline void Process(){
        // 算子核心处理函数实现
    }
private:
    // 搬入函数，完成CopyIn阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyIn(int32_t progress){
        // CopyIn函数实现
    }
    // 计算函数，完成Compute阶段的处理，被核心Process函数调用
    __aicore__ inline void Compute(int32_t progress){
        // Compute函数实现
    }

    // 搬出函数，完成CopyOut阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyOut(int32_t progress){
        // CopyOut函数实现
    }

private:
    TPipe pipe;  //Pipe内存管理对象
    TQue<QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;  //输入数据Queue队列管理对象，QuePosition为VECIN
    TQue<QuePosition::VECOUT, BUFFER_NUM> outQueueZ;  //输出数据Queue队列管理对象，QuePosition为VECOUT
    GlobalTensor<half> xGm, yGm, zGm;  //管理输入输出Global Memory内存地址的对象，其中xGm, yGm为输入，zGm为输出
};
```

HUAWEI

算子实现类 — 常量

使用**多核并行计算**，需要将数据切片，获取到每个核实际需要处理的在Global Memory上的内存偏移地址

数据整体长度*TOTAL_LENGTH*为8* 2048，平均分配到8个核上运行，每个核上处理的数据大小*BLOCK_LENGTH*为2048。

*block_idx*为核的逻辑ID，*(__gm__ half*)x + block_idx * BLOCK_LENGTH*即索引为*block_idx*的核的输入数据在Global Memory上的内存偏移地址

```
constexpr int32_t TOTAL_LENGTH = 8 * 2048;                                // 静态输入shape
constexpr int32_t USE_CORE_NUM = 8;                                       // 运算核个数
constexpr int32_t BLOCK_LENGTH = TOTAL_LENGTH / USE_CORE_NUM;             // 每个核上运算数据的元素个数
constexpr int32_t TILE_NUM = 8;                                           // 每个核上运算数据切分次数
constexpr int32_t BUFFER_NUM = 2;                                         // double buffer机制，如果数据量较小可以改为1，避免负优化
constexpr int32_t TILE_LENGTH = BLOCK_LENGTH / TILE_NUM / BUFFER_NUM;     // 每次参与运算的数据元素个数
```


算子实现类 — Init

初始化函数主要完成以下内容：

1、设置输入输出Global Tensor的Global Memory内存地址

2、通过Pipe内存管理对象为输入输出Queue分配内存

```
// 初始化函数，完成内存初始化相关操作
__aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z){
    // 根据唯一block_id计算出当前核参与运算的数据内存地址起始位
    GM_ADDR xGmOffset = x + BLOCK_LENGTH * GetBlockIdx();
    GM_ADDR yGmOffset = y + BLOCK_LENGTH * GetBlockIdx();
    GM_ADDR zGmOffset = z + BLOCK_LENGTH * GetBlockIdx();

    // 使用GlobalTensor存放当前核上运算数据
    xGm.SetGlobalBuffer((__gm__ half*)xGmOffset, BLOCK_LENGTH);
    yGm.SetGlobalBuffer((__gm__ half*)yGmOffset, BLOCK_LENGTH);
    zGm.SetGlobalBuffer((__gm__ half*)zGmOffset, BLOCK_LENGTH);
    // 为指定的输入输出Queue分配内存，内存大小为每次参与运算的数据元素个数 * 该类型数据占用字节数
    pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(inQueueY, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(outQueueZ, BUFFER_NUM, TILE_LENGTH * sizeof(half));
}
```

算子实现类 — Process

在Process函数内，我们循环调用矢量算子的CopyIn，Compute，CopyOut三个流水任务，对分配到该运算核上的数据进行运算

```
// 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操作
__aicore__ inline void Process(){
    // 循环次数 = 当前核运算数据元素个数 / 每次参与运算的数据元素个数
    constexpr int32_t loopCount = TILE_NUM * BUFFER_NUM;
    for (int32_t i = 0; i < loopCount; i++) {
        CopyIn(i);
        Compute(i);
        CopyOut(i);
    }
}
```

Ascend C 算子样例详解

算子实现类 — CopyIn

CopyIn函数主要完成以下内容：

1、使用DataCopy接口将GlobalTensor数据拷贝到LocalTensor。

2、使用EnQue将LocalTensor放入VecIn的Queue中。

```
__aicore__ inline void CopyIn(int32_t progress){
    // 从输入Queue分配LocalTensor，用于搬入数据
    LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
    LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
    DataCopy(xLocal, xGm[progress * TILE_LENGTH], TILE_LENGTH);
    DataCopy(yLocal, yGm[progress * TILE_LENGTH], TILE_LENGTH);
    // 把存有输入数据的LocalTensor放入Queue，完成数据搬入操作
    inQueueX.EnQue(xLocal);
    inQueueY.EnQue(yLocal);
}
```

算子实现类 — Compute

Compute函数主要完成以下内容：

1、使用DeQue从VecIn中取出LocalTensor。

2、使用Ascend C接口Add完成矢量计算。

3、使用EnQue将计算结果LocalTensor放入到VecOut的Queue中。

4、使用FreeTensor释放不再使用的LocalTensor。

```
__aicore__ inline void Compute(int32_t progress){
    // 从输入Queue中取出输入数据
    LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    // 从输出Queue分配LocalTensor，用于搬出数据
    LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    // 通过调用API完成运算
    Add(zLocal, xLocal, yLocal, TILE_LENGTH);
    // 把存有输出数据的LocalTensor放入输出Queue
    outQueueZ.EnQue<half>(zLocal);
    // 释放输入LocalTensor
    inQueueX.FreeTensor(xLocal);
    inQueueY.FreeTensor(yLocal);
}
```

Ascend C 算子样例详解

算子实现类 — CopyOut

CopyOut函数主要完成以下内容：

1、使用DeQue接口从VecOut的Queue中取出LocalTensor。

2、使用DataCopy接口将LocalTensor拷贝到GlobalTensor上。

3、使用FreeTensor将不再使用的LocalTensor进行回收。

```
__aicore__ inline void CopyOut(int32_t progress){
    // 从输出Queue取出LocalTensor
    LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
    // 根据传入的progress 和每次参与运算的数据元素个数计算出当前循环的数据地址，通过DataCopy把LocalTensor结果数据放入
    DataCopy(zGm[progress * TILE_LENGTH], zLocal, TILE_LENGTH);
    // 释放输出LocalTensor
    outQueueZ.FreeTensor(zLocal);
}
```

Add算子类实现

*CopyIn*任务：将Global Memory上的输入Tensor *xGm*和*yGm*搬运至Local Memory，分别存储在*xLocal*, *yLocal*
*Compute*任务：对*xLocal*, *yLocal*执行加法操作，计算结果存储在*zLocal*中
*CopyOut*任务：将输出数据从*zLocal*搬运至Global Memory上的输出Tensor *zGm*中

CopyIn，*Compute*任务间通过*VECIN*队列*inQueueX*，*inQueueY*进行通信和同步
Compute，*CopyOut*任务间通过*VECOUT*队列*outQueueZ*进行通信和同步

*pipe*内存管理对象对任务间交互使用到的内存、临时变量使用到的内存统一进行管理

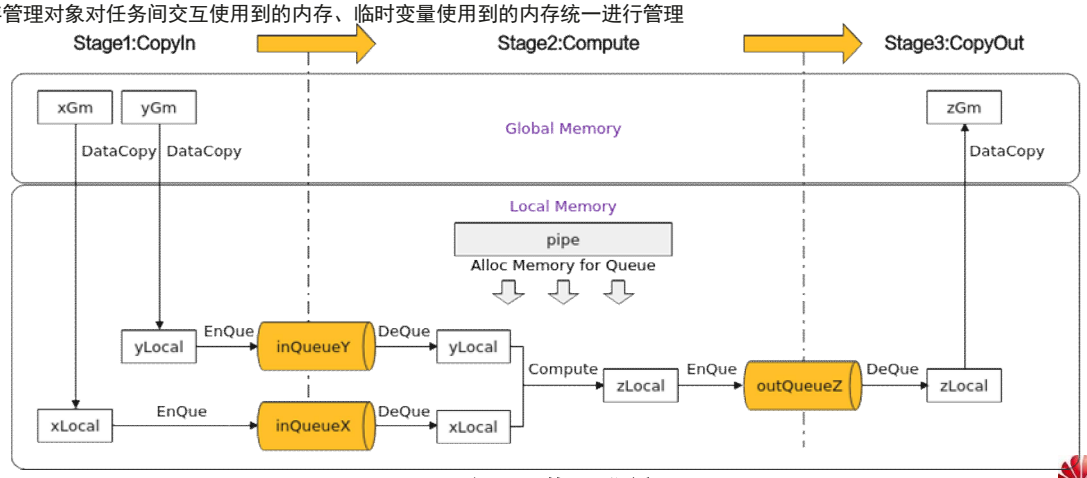


图8 矢量Add算子开发流程

算子调试 — CPU域调试

```
int32_t main(int32_t argc, char* argv[])
{
    // 输入或输出的元素个数
    const size_t inputOutputNum = 8 * 2048;
    // 需要运算数据的内存大小
    size_t inputOutputByteSize = inputOutputNum * sizeof(uint16_t);
    // 参与运算的计算核数
    uint32_t blockDim = 8;
    // 分配内存给输入输出
    uint8_t* x = (uint8_t*)tik2::GmAlloc(inputOutputByteSize);
    uint8_t* y = (uint8_t*)tik2::GmAlloc(inputOutputByteSize);
    uint8_t* z = (uint8_t*)tik2::GmAlloc(inputOutputByteSize);
    // 初始化测试数据数组，为了方便查看结果，此处只赋值了前4个数据
    uint16_t input_x [inputOutputNum] = {1,2,3,4};
    uint16_t input_y [inputOutputNum] = {5,6,7,8};
    // 把测试数据放入分配的内存
    x = (uint8_t *)input_x;
    y = (uint8_t *)input_y;
    // 调用核函数运算
    ICPU_RUN_KF(add_custom, blockDim, x, y, z);
    // 把得到的结果转为原始数据类型
    uint16_t *result = (uint16_t *)z;
    // 打印出结果，此处我们只查看前4个数据运算结果
    for(size_t i = 0; i < 4; i++){
        std::cout<<"第["<< i <<"]个输入x = "<< input_x[i] <<"  输入y = "<< input_y[i]<<" 结果 = "<< result[i];
    }
    // 释放输入输出内存
    tik2::GmFree((void *)x);
    tik2::GmFree((void *)y);
    tik2::GmFree((void *)z);
    return 0;
}
```

```
第[0]个输入x = 1 输入y = 5 结果 = 6
第[1]个输入x = 2 输入y = 6 结果 = 8
第[2]个输入x = 3 输入y = 7 结果 = 10
第[3]个输入x = 4 输入y = 8 结果 = 12
```

算子调试 — NPU域调试

```
int32_t main(int32_t argc, char* argv[])
{
    size_t inputByteSize = 8 * 2048 * sizeof(uint16_t);  // uint16_t represent half
    size_t outputByteSize = 8 * 2048 * sizeof(uint16_t);  // uint16_t represent half
    uint32_t blockDim = 8;
    CHECK_ACL(aclInit(nullptr));
    aclrtContext context;
    int32_t deviceId = 0;
    CHECK_ACL(aclrtSetDevice(deviceId));
    CHECK_ACL(aclrtCreateContext(&context, deviceId));
    aclrtStream stream = nullptr;
    CHECK_ACL(aclrtCreateStream(&stream));
    uint8_t *xHost, *yHost, *zHost;
    uint8_t *xDevice, *yDevice, *zDevice;
    // 使用ACL接口分配Host上内存
    CHECK_ACL(aclrtMallocHost((void**)(&xHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&yHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&zHost), outputByteSize));
    // 使用ACL接口分配device上内存
    CHECK_ACL(aclrtMalloc((void**)&xDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&yDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&zDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // 初始化测试数据数组，为了方便查看结果，此处只赋值了前4个数据
    uint16_t input_x [inputOutputNum] = {4,3,2,1};
    uint16_t input_y [inputOutputNum] = {5,6,7,8};
    // 把测试数据放入分配的Host内存
    xHost = (uint8_t *)input_x;
    yHost = (uint8_t *)input_y;
    // Host侧数据搬到Device
    CHECK_ACL(aclrtMemcpy(xDevice, inputByteSize, xHost, inputByteSize, ACL_MEMCPY_HOST_TO_DEVICE));
    CHECK_ACL(aclrtMemcpy(yDevice, inputByteSize, yHost, inputByteSize, ACL_MEMCPY_HOST_TO_DEVICE));
    // 调用测试函数运算
    add_custom_do(blockDim, nullptr, stream, xDevice, yDevice, zDevice);
    CHECK_ACL(aclrtSynchronizeStream(stream));
    // Device运算结果搬出到Host侧内存
    CHECK_ACL(aclrtMemcpy(zHost, outputByteSize, zDevice, outputByteSize, ACL_MEMCPY_DEVICE_TO_HOST));
    uint16_t *result = (uint16_t *)zHost;
    // 打印出结果，此处我们只查看前4个数据运算结果
    for(size_t i = 0; i < 4; i++){
        std::cout<<"第["<< i <<"]个输入x = "<< input_x[i] <<"  输入y = "<< input_y[i]<<" 结果 = "<< result[i];
    }
    CHECK_ACL(aclrtFree(xDevice));
    CHECK_ACL(aclrtFree(yDevice));
    CHECK_ACL(aclrtFree(zDevice));
    CHECK_ACL(aclrtFreeHost(xHost));
    CHECK_ACL(aclrtFreeHost(yHost));
    CHECK_ACL(aclrtFreeHost(zHost));
    CHECK_ACL(aclrtDestroyStream(stream));
    CHECK_ACL(aclrtDestroyContext(context));
    CHECK_ACL(aclrtResetDevice(deviceId));
    CHECK_ACL(aclFinalize());
    return 0;
}
```

```
};

extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

NPU侧调试时，算子实现文件内需要添加调试函数

```
#ifndef __CCE_KT_TEST__
// call of kernel function
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y, uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
#endif
```

```
第[0]个输入x = 4 输入y = 5 结果 = 9
第[1]个输入x = 3 输入y = 6 结果 = 9
第[2]个输入x = 2 输入y = 7 结果 = 9
第[3]个输入x = 1 输入y = 8 结果 = 9
```

HUAWEI

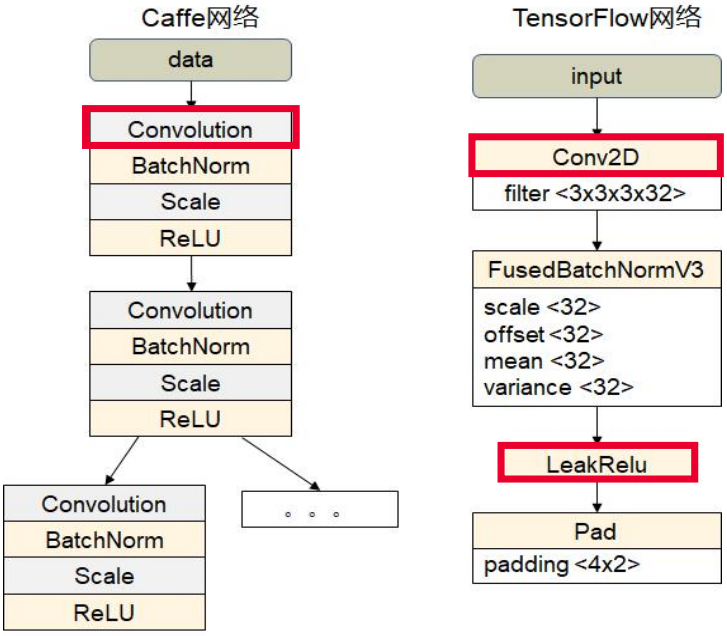
讲授内容：华为CANN编程（二）

- 昇腾AI基础软硬件平台
- 昇腾AI异构计算架构CANN
- CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

算子基本概念

什么是算子 — 算子在神经网络中的含义

深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称OP）。在网络模型中，算子对应层或者节点的计算逻辑，通常网络模型都是由一个或多个算子拼接组成。



257

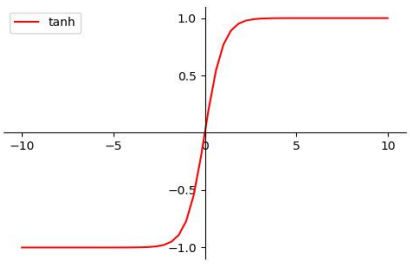
HUAWEI

算子基本概念

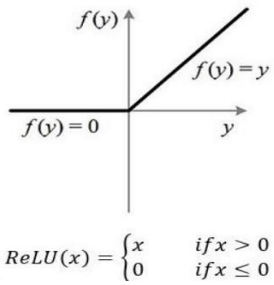
什么是算子 — 算子的数学含义

在数学领域，一个函数空间到函数空间上的映射O：X→Y，都称为算子。广义的讲，对任何函数进行某一项操作都可以认为是一个算子，比如微分算子，不定积分算子等。

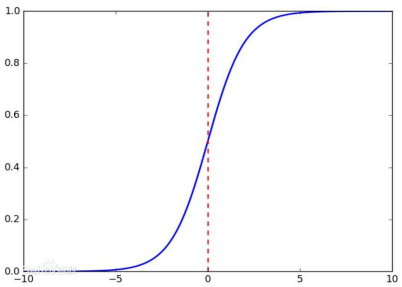
常见算子： tanh、ReLU、sigmoid等。



Tanh函数图像



ReLU函数图像



Sigmoid函数图像

258

HUAWEI

什么场景需要开发自定义算子

一般情况下，开发者无需自己开发算子，但若遇到以下场景，开发者需要考虑进行自定义算子的开发：

- 训练场景或推理场景下，遇到了不支持的算子
- 网络调优时，发现某个算子性能较差，想重新开发一个高性能算子替换性能较差的算子
- 需要定制特殊算法功能的算子

例如，针对一个分类应用，我们想从分类模型的推理结果中查找可能性最大的前5个标识，则可以实现一个查找最大值的算子（例如ArgMax），后续就可以直接通过AscendCL接口调用此算子实现对推理结果的后处理。

算子基本概念 — 总览

进行算子开发前，首先需要了解算子相关的基本概念。

- **算子名称（Name）**

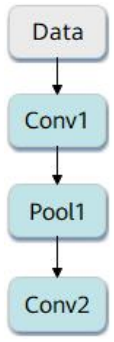
  算子名称用于标志网络中的某个算子，同一网络中算子的名称需要保持唯一。如右图所示Conv1，Pool1，Conv2都是此网络中的算子名称，其中Conv1与Conv2算子的类型为Convolution，表示分别做一次卷积运算。

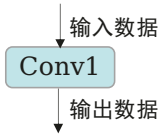
- **算子类型（Type）**

  网络中每一个算子根据算子类型进行算子实现的匹配，相同类型算子的实现逻辑相同。在一个网络中同一类型的算子可能存在多个，例如右图中名称为Conv1的算子与Conv2算子的类型都为Convolution。

- **数据容器（Tensor）**

  Tensor是存储算子输入数据与输出数据的容器，如右图所示，算子在网络中执行时，需要输入数据，算子执行完后，也会有对应的数据输出。这种承载算子数据的容器定义为张量（Tensor）。





算子基本概念 — Tensor

张量（Tensor）是存储算子输入数据与输出数据的容器，而张量描述符（TensorDesc）是对输入数据与输出数据的描述，张量描述符的数据结构包含如下属性：

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需要保持唯一。
形状（shape）	Tensor的形状，比如（10,）或者（1024, 1024）或者（2, 3, 4）等。 形式：(i1, i2,...in)，其中i1到in均为正整数
数据类型（dtype）	指定Tensor对象的数据类型。 例如：float16, float32, int8, int16, int32, uint8, uint16, bool等。 不同计算操作支持的数据类型不同。
数据排布格式（format）	数据的物理排布格式，定义了解读数据的维度。

算子基本概念 — Shape

下面分别介绍张量描述符中的形状和数据排布格式。

- **形状（shape）**

张量的形状，以(D0, D1, ... ,Dn-1)的形式表示，D0到Dn是任意的正整数。

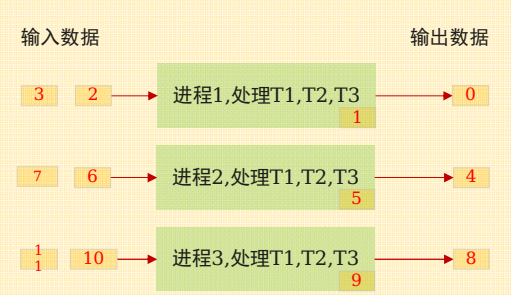
如形状(3,4)表示第一维有3个元素，第二维有4个元素，是一个3行4列的矩阵数组。

在形状的小括号中有多少个数字，就代表这个张量是多少维的张量。形状的第一个元素要看张量最外层的中括号中有几个元素，形状的第二个元素要看张量中从左边开始数第二个中括号中有几个元素，依此类推。

例如：

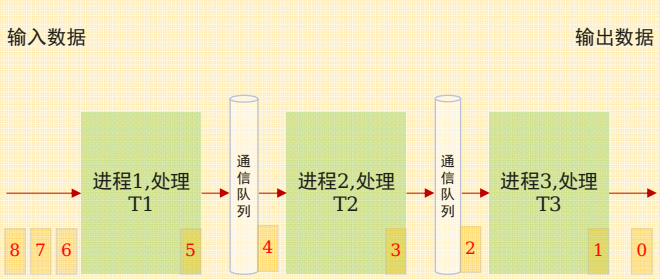
张量	形状
1	(0,)
[1,2,3]	(3,)
[[1,2],[3,4]]	(2,2)
[[[1,2],[3,4]], [[5,6],[7,8]]]	(2,2,2)

并行计算中两种常见方法：单程序多数据（ SPMD ）和流水线并行



SPMD 数据并行计算原理

- 启动一组进程，它们运行相同的程序
- 把待处理数据切分，把切分后数据分片分发给不同进程处理
- 每个进程对自己的数据分片进行3个任务T1、T2、T3的处理



流水线并行原理

- 启动一组进程
- 对数据进行切分
- 每个进程都处理所有的数据切片，对输入数据分片只做一个任务的处理

HUAWEI

讲授内容：华为CANN编程（二）

- ➢昇腾AI基础软硬件平台
- ➢昇腾AI异构计算架构CANN
- ➢CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- ➢Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解

CANN算子是AI应用的基石

已有算子的计算逻辑不满足要求

Ascend C算子编程基础

什么是Ascend C 算子

Ascend C是CANN针对算子开发场景推出的编程语言，通过**多层接口抽象、自动并行计算、孪生调试**等关键技术，极大提高算子开发效率，助力AI开发者低成本完成算子开发和模型调优部署。使用Ascend C编程语言开发的算子我们称之为Ascend C算子。

使用Ascend C开发自定义算子的优势：

- ✓ C/C++原语编程，最大化匹配用户的开发习惯
- ✓ 编程模型屏蔽硬件差异，编程范式提高开发效率
- ✓ 多层级API封装，从简单到灵活，兼顾易用与高效
- ✓ 孪生调试，CPU侧模拟NPU侧的行为，可优先在CPU侧调试

通过华为AI加速卡（NPU），可以实现大规模神经网络计算加速

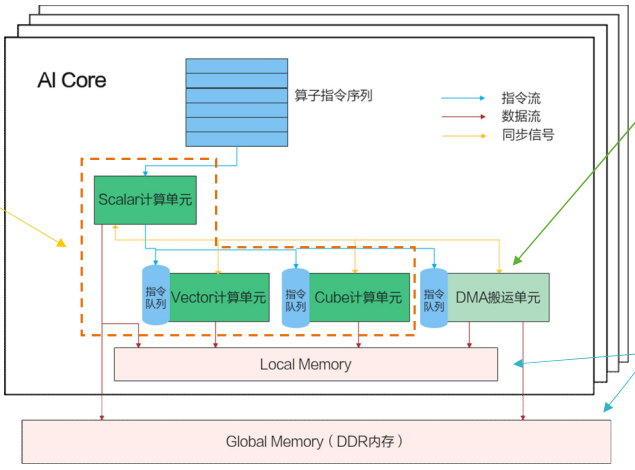
AICORE是NPU卡的计算核心，NPU内部有多个AICORE。每个AICORE相当于多核CPU的一个核心

AI Core内部并行计算架构抽象（1）

Ascend C 算子运行在AI Core上，AI Core是昇腾AI处理器中的**计算核心**。一个AI处理器内部有多个AI Core，AI Core中包含**计算单元、存储单元、搬运单元**等核心组件

计算单元包括了三种基础计算资源

- **Scalar计算单元：**执行地址计算、循环控制等标量计算工作，并把向量计算、矩阵计算、数据搬运、同步指令发射给对应单元执行
- **Cube计算单元：**负责执行矩阵运算
- **Vector计算单元：**负责执行向量运算



AI Core内部并行计算架构抽象示意图

搬运单元负责在Global Memory和Local Memory之间搬运数据，包含搬运单元MTE2（Memory Transfer Engine，数据搬入单元），MTE3（数据搬出单元）

存储单元为AI Core的内部存储，统称为Local Memory；
与此相对应，AI Core的外部存储称之为Global Memory；

AI Core内部并行计算架构抽象（2）

异步指令流

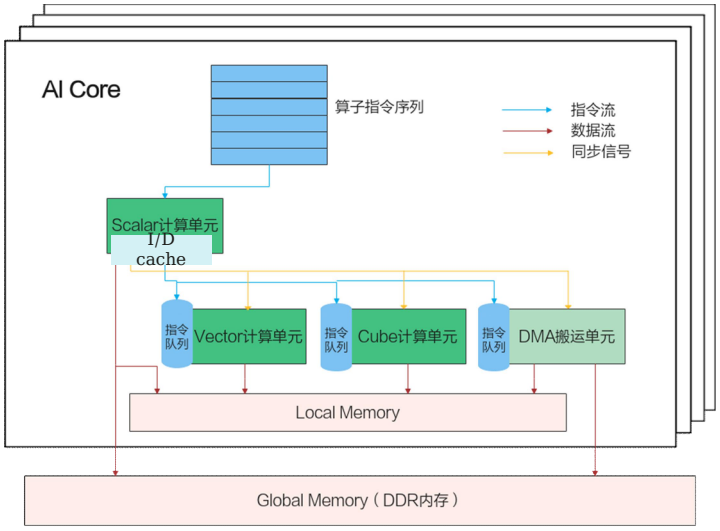
Scalar计算单元读取指令序列，并把向量计算、矩阵计算、数据搬运指令发射给对应单元的指令队列，向量计算单元、矩阵计算单元、数据搬运单元异步的并行执行接收到的指令

同步信号流

指令间可能会存在依赖关系，为了保证不同指令队列间的指令按照正确的逻辑关系执行，Scalar计算单元也会给对应单元下发同步指令

计算数据流

DMA搬入单元把数据搬运到Local Memory，Vector/Cube计算单元完成数据计算，并把计算结果写回Local Memory，DMA搬出单元把处理好的数据搬运回Global Memory



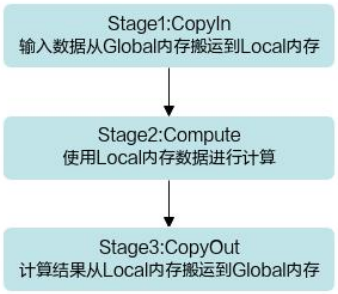
AI Core内部并行计算架构抽象示意图

269

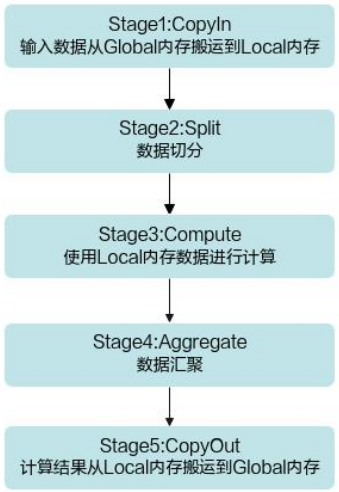
HUAWEI

编程范式——流水线式编程范式

Ascend C编程范式是一种流水线式的编程范式，把算子核内的处理程序，分成多个**流水任务**，通过队列（Queue）完成**任务间通信和同步**，并通过统一的**内存管理模块**（Pipe）管理任务间通信内存。



矢量编程基本任务设计



矩阵编程基本任务设计

270

HUAWEI

矢量编程 —— double buffer机制

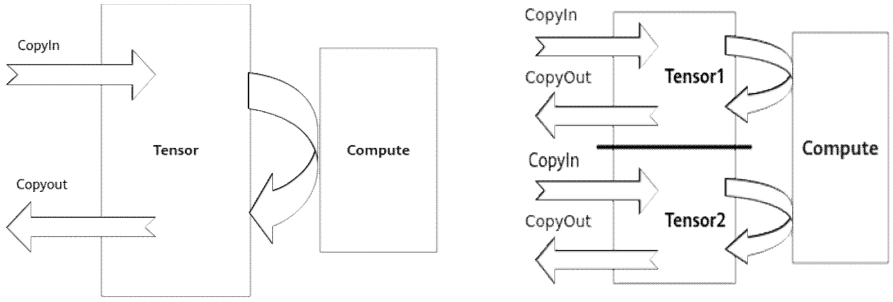
double buffer通过将**数据搬运与矢量计算并行执行**以隐藏数据搬运时间并降低矢量指令的等待时间，最终提高矢量计算单元的利用效率。

1个Tensor同一时间只能进行搬入、计算和搬出三个流水任务中的一个，其他两个流水任务涉及的硬件单元则处于Idle状态

如果将待处理的数据一分为二，比如Tensor1、Tensor2

- 当矢量计算单元对Tensor1进行Compute时，Tensor2可以执行CopyIn的任务
- 当矢量计算单元对Tensor2进行Compute时，Tensor1可以执行CopyOut的任务
- 当矢量计算单元对Tensor2进行CopyOut时，Tensor1可以执行CopyIn的任务

由此，数据的进出搬运和矢量计算之间实现并行，硬件单元闲置问题得以有效缓解



Double Buffer下的数据搬运与Vector计算过程

多层级API封装

Ascend C算子采用标准C++语法和一组类库API进行编程，类库API主要包含以下几种：

- **计算类API**，包括标量计算API、向量计算API、矩阵计算API，分别实现调用Scalar计算单元、Vector计算单元、Cube计算单元执行计算的功能。
- **数据搬运API**，上述计算API基于Local Memory数据进行计算，所以数据需要先从Global Memory搬运至Local Memory，再使用计算接口完成计算，最后从Local Memory搬出至Global Memory。执行搬运过程的接口称之为数据搬移接口，比如DataCopy接口。
- **内存管理API**，用于分配管理内存，比如AllocTensor、FreeTensor接口。
- **任务同步API**，完成任务间的通信和同步，比如EnQue、DeQue接口。该类型API，开发者无需关注内部实现逻辑，使用简单的API接口即可完成。

Ascend C将API分为0-3级，随着级别增高，API使用的自由度降低，易用性增强。您可以根据需要选择合适的API，使用最通俗易懂的高级接口快速搭建算子逻辑，使用自由灵活的低级接口进行复杂的逻辑实现和性能调优。以矢量计算类API为例：

接口级别	接口说明
0级	功能灵活的计算API，充分发挥硬件优势，支持对每个操作数的Block stride，Repeat stride，Mask的操作。Block stride，Repeat stride，Mask参数的详细介绍请参见0级接口通用参数说明。
1级	slice计算API，解决多维数据中的切片计算问题。**该版本暂不支持1级接口。**
2级	针对源操作数的连续数据进行计算并连续写入目的操作数，解决一维tensor的连续计算问题。
3级	运算符重载，支持+, -, *, /, |, &, <, >, <=, >=, ==, !=，实现2级指令的简化表达。

孪生调试

Ascend C提供**孪生调试**方法，即在cpu侧创建一个npu的模型并模拟它的计算行为，用来进行业务功能调试。相同的算子代码可以在cpu域调试精度，npu域调试性能。

CPU域调试

1、运行使能ASSERT，初步拦截算子指令或框架使用错误，如参数超限，指令数据地址重叠，该芯片不支持的指令等

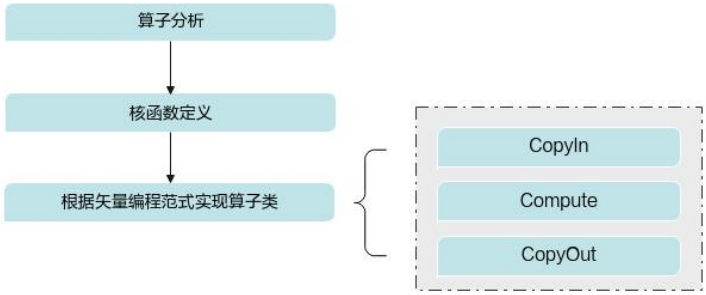
2、可使用gdb单步调试算子计算精度，也可以在代码中直接编写printf(...)来观察数值的输出。

NPU域调试

可使用工具获取真实芯片上profiling数据，进行性能精细调优

讲授内容：华为CANN编程（二）

- 昇腾AI基础软硬件平台
- 昇腾AI异构计算架构CANN
- CANN关键能力
  - ✓模型迁移&训练
  - ✓推理应用开发
  - ✓算子开发
- Ascend C 算子开发入门
  - ✓算子基本概念
  - ✓Ascend C算子编程基础
  - ✓Ascend C算子样例讲解



矢量算子实现流程

矢量编程 —— 算子分析

算子类型（OpType）	Add			
算子输入	name	shape	data type	format
	x	(8, 2048)	half	ND
	y	(8, 2048)	half	ND
算子输出	z	(8, 2048)	half	ND
核函数名	add_custom			

■ **明确算子的数学表达式及计算逻辑**

Add算子的数学表达式为：$\vec{z} = \vec{x} + \vec{y}$，计算逻辑：输入数据需要先搬入到片上存储，然后使用计算接口完成两个加法运算，得到最终结果，再搬出到外部存储

■ **明确输入和输出**

Add算子有两个输入：$\vec{x}$ 与 $\vec{y}$，输出为$\vec{z}$。输入数据类型为*half*，输出数据类型与输入数据类型相同。输入支持固定shape(8，2048)，输出shape与输入shape相同。输入数据排布类型为*ND*

■ **确定算子实现所需接口**

涉及内外部存储间的数据搬运，使用数据搬移接口：*DataCopy*实现

涉及矢量计算的加法操作，使用矢量双目指令：*Add*实现

使用*LocalTensor*存放数据，使用Queue管理队列，会使用到*EnQue*（将Tensor放入队列）、*DeQue*（将Tensor从队列取出）等接口。

■ **确定核函数名称和参数**

自定义核函数名，如*add_custom*。根据输入输出，确定核函数有3个入参*x*，*y*，*z*

x，*y*为输入在Global Memory上的内存地址，*z*为输出在Global Memory上的内存地址

Ascend C算子开发—核函数定义

核函数（Kernel Function）是Ascend C算子设备侧实现的入口。在核函数中，需要为在一个核上执行的代码规定要进行的数据访问和计算操作，当核函数被调用时，多个核都执行相同的核函数代码，具有相同的参数，并行执行。

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

核函数写法比较固定，一般为**extern “C” __global__ __aicore__ void 自定义核函数名(输入1，输入2……，输出1，输出2……)** 的形式，

277

HUAWEI

Ascend C算子开发—核函数调用

核函数的调用语句是C/C++函数调用语句的一种扩展，使用内核调用符<<<...>>>这种语法形式，来规定核函数的执行配置。内核调用符仅可在NPU侧编译时调用，CPU侧编译无法识别该符号。

核函数的调用是异步的，核函数的调用结束后，控制权立刻返回给主机端，可以调用以下函数来强制主机端程序等待所有核函数执行完毕。

```
#ifndef __CCE_KT_TEST__
// call of kernel function
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y,
uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
#endif
```

278

HUAWEI

Ascend C算子开发—算子实现类

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd(){}
    // 初始化函数，完成内存初始化相关操作
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z){
        // 初始化函数实现
    }
    // 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操作
    __aicore__ inline void Process(){
        // 算子核心处理函数实现
    }
private:
    // 搬入函数，完成CopyIn阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyIn(int32_t progress){
        // CopyIn函数实现
    }
    // 计算函数，完成Compute阶段的处理，被核心Process函数调用
    __aicore__ inline void Compute(int32_t progress){
        // Compute函数实现
    }

    // 搬出函数，完成CopyOut阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyOut(int32_t progress){
        // CopyOut函数实现
    }

private:
    TPipe pipe;  //Pipe内存管理对象
    TQue<QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;  //输入数据Queue队列管理对象，QuePosition为VECIN
    TQue<QuePosition::VECOUT, BUFFER_NUM> outQueueZ;  //输出数据Queue队列管理对象，QuePosition为VECOUT
    GlobalTensor<half> xGm, yGm, zGm;  //管理输入输出Global Memory内存地址的对象，其中xGm，yGm为输入，zGm为输出
};
```


算子实现类 — 常量

使用**多核并行计算**，需要将数据切片，获取到每个核实际需要处理的在Global Memory上的内存偏移地址

数据整体长度*TOTAL_LENGTH*为8* 2048，平均分配到8个核上运行，每个核上处理的数据大小*BLOCK_LENGTH*为2048。*block_idx*为核的逻辑ID，*(__gm__ half*)x + block_idx * BLOCK_LENGTH*即索引为*block_idx*的核的输入数据在Global Memory上的内存偏移地址

```
constexpr int32_t TOTAL_LENGTH = 8 * 2048;                            // 静态输入shape
constexpr int32_t USE_CORE_NUM = 8;                                   // 运算核个数
constexpr int32_t BLOCK_LENGTH = TOTAL_LENGTH / USE_CORE_NUM;         // 每个核上运算数据的元素个数
constexpr int32_t TILE_NUM = 8;                                       // 每个核上运算数据切分次数
constexpr int32_t BUFFER_NUM = 2;                                     // double buffer机制，如果数据量较小可以改为1，避免负优化
constexpr int32_t TILE_LENGTH = BLOCK_LENGTH / TILE_NUM / BUFFER_NUM; // 每次参与运算的数据元素个数
```

算子实现类 — Init

初始化函数主要完成以下内容：

1、设置输入输出Global Tensor的Global Memory内存地址

2、通过Pipe内存管理对象为输入输出Queue分配内存

```
// 初始化函数，完成内存初始化相关操作
__aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z){
    // 根据唯一block_id计算出当前核参与运算的数据内存地址起始位
    GM_ADDR xGmOffset = x + BLOCK_LENGTH * GetBlockIdx();
    GM_ADDR yGmOffset = y + BLOCK_LENGTH * GetBlockIdx();
    GM_ADDR zGmOffset = z + BLOCK_LENGTH * GetBlockIdx();

    // 使用GlobalTensor存放当前核上运算数据
    xGm.SetGlobalBuffer((__gm__ half*)xGmOffset, BLOCK_LENGTH);
    yGm.SetGlobalBuffer((__gm__ half*)yGmOffset, BLOCK_LENGTH);
    zGm.SetGlobalBuffer((__gm__ half*)zGmOffset, BLOCK_LENGTH);
    // 为指定的输入输出Queue分配内存，内存大小为每次参与运算的数据元素个数 * 该类型数据占用字节数
    pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(inQueueY, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(outQueueZ, BUFFER_NUM, TILE_LENGTH * sizeof(half));
}
```


算子实现类 — Process

在Process函数内，我们循环调用矢量算子的CopyIn，Compute，CopyOut三个流水任务，对分配到该运算核上的数据进行运算

```
// 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操作
__aicore__ inline void Process(){
    // 循环次数 = 当前核运算数据元素个数 / 每次参与运算的数据元素个数
    constexpr int32_t loopCount = TILE_NUM * BUFFER_NUM;
    for (int32_t i = 0; i < loopCount; i++) {
        CopyIn(i);
        Compute(i);
        CopyOut(i);
    }
}
```

算子实现类 — CopyIn

CopyIn函数主要完成以下内容：

1、使用DataCopy接口将GlobalTensor数据拷贝到LocalTensor。

2、使用EnQue将LocalTensor放入VecIn的Queue中。

```
__aicore__ inline void CopyIn(int32_t progress){
    // 从输入Queue分配LocalTensor，用于搬入数据
    LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
    LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
    DataCopy(xLocal, xGm[progress * TILE_LENGTH], TILE_LENGTH);
    DataCopy(yLocal, yGm[progress * TILE_LENGTH], TILE_LENGTH);
    // 把存有输入数据的LocalTensor放入Queue，完成数据搬入操作
    inQueueX.EnQue(xLocal);
    inQueueY.EnQue(yLocal);
}
```


算子实现类 — Compute

Compute函数主要完成以下内容：

1、使用DeQue从VecIn中取出LocalTensor。

2、使用Ascend C接口Add完成矢量计算。

3、使用EnQue将计算结果LocalTensor放入到VecOut的Queue中。

4、使用FreeTensor释放不再使用的LocalTensor。

```
__aicore__ inline void Compute(int32_t progress){
    // 从输入Queue中取出输入数据
    LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    // 从输出Queue分配LocalTensor，用于搬出数据
    LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    // 通过调用API完成运算
    Add(zLocal, xLocal, yLocal, TILE_LENGTH);
    // 把存有输出数据的LocalTensor放入输出Queue
    outQueueZ.EnQue<half>(zLocal);
    // 释放输入LocalTensor
    inQueueX.FreeTensor(xLocal);
    inQueueY.FreeTensor(yLocal);
}
```

算子实现类 — CopyOut

CopyOut函数主要完成以下内容：

1、使用DeQue接口从VecOut的Queue中取出LocalTensor。

2、使用DataCopy接口将LocalTensor拷贝到GlobalTensor上。

3、使用FreeTensor将不再使用的LocalTensor进行回收。

```
__aicore__ inline void CopyOut(int32_t progress){
    // 从输出Queue取出LocalTensor
    LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
    // 根据传入的progress 和每次参与运算的数据元素个数计算出当前循环的数据地址，通过DataCopy把LocalTensor结果数据放入
    DataCopy(zGm[progress * TILE_LENGTH], zLocal, TILE_LENGTH);
    // 释放输出LocalTensor
    outQueueZ.FreeTensor(zLocal);
}
```


Ascend C 算子样例详解

Add算子类实现

*CopyIn*任务：将Global Memory上的输入Tensor *xGm*和*yGm*搬运至Local Memory，分别存储在*xLocal*, *yLocal*

*Compute*任务：对*xLocal*, *yLocal*执行加法操作，计算结果存储在*zLocal*中

*CopyOut*任务：将输出数据从*zLocal*搬运至Global Memory上的输出Tensor *zGm*中

CopyIn，*Compute*任务间通过*VECIN*队列*inQueueX*，*inQueueY*进行通信和同步

Compute，*CopyOut*任务间通过*VECOUT*队列*outQueueZ*进行通信和同步

*pipe*内存管理对象对任务间交互使用到的内存、临时变量使用到的内存统一进行管理

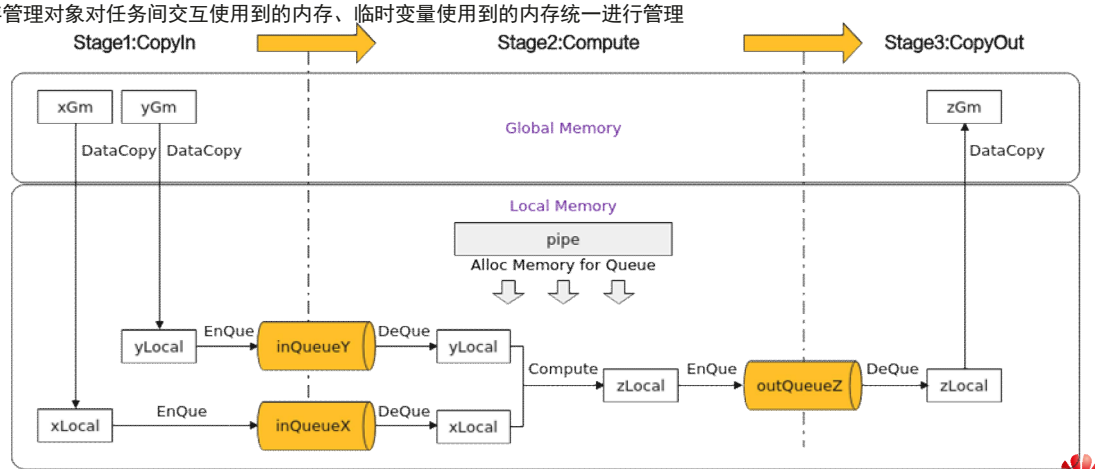


图8 矢量Add算子开发流程

算子调试 — CPU域调试

```
int32_t main(int32_t argc, char* argv[])
{
    // 输入或输出的元素个数
    const size_t inputOutputNum = 8 * 2048;
    // 需要运算数据的内存大小
    size_t inputOutputByteSize = inputOutputNum * sizeof(uint16_t);
    // 参与运算的计算核数
    uint32_t blockDim = 8;
    // 分配内存给输入输出
    uint8_t* x = (uint8_t*)tik2::GmAlloc(inputOutputByteSize);
    uint8_t* y = (uint8_t*)tik2::GmAlloc(inputOutputByteSize);
    uint8_t* z = (uint8_t*)tik2::GmAlloc(inputOutputByteSize);
    // 初始化测试数据数组，为了方便查看结果，此处只赋值了前4个数据
    uint16_t input_x [inputOutputNum] = {1,2,3,4};
    uint16_t input_y [inputOutputNum] = {5,6,7,8};
    // 把测试数据放入分配的内存
    x = (uint8_t *)input_x;
    y = (uint8_t *)input_y;
    // 调用核函数运算
    ICPU_RUN_KF(add_custom, blockDim, x, y, z);
    // 把得到的结果转为原始数据类型
    uint16_t *result = (uint16_t *)z;
    // 打印出结果，此处我们只查看前4个数据运算结果
    for(size_t i = 0; i < 4; i++){
        std::cout<<"第["<< i <<"]个输入x = "<< input_x[i] <<"  输入y = "<< input_y[i]<<" 结果 = "<< result[i];
    }
    // 释放输入输出内存
    tik2::GmFree((void *)x);
    tik2::GmFree((void *)y);
    tik2::GmFree((void *)z);
    return 0;
}
```

```
第[0]个输入x = 1 输入y = 5 结果 = 6
第[1]个输入x = 2 输入y = 6 结果 = 8
第[2]个输入x = 3 输入y = 7 结果 = 10
第[3]个输入x = 4 输入y = 8 结果 = 12
```


算子调试 — NPU域调试

```
int32_t main(int32_t argc, char* argv[])
{
    size_t inputByteSize = 8 * 2048 * sizeof(uint16_t);  // uint16_t represent half
    size_t outputByteSize = 8 * 2048 * sizeof(uint16_t);  // uint16_t represent half
    uint32_t blockDim = 8;
    CHECK_ACL(aclInit(nullptr));
    aclrtContext context;
    int32_t deviceId = 0;
    CHECK_ACL(aclrtSetDevice(deviceId));
    CHECK_ACL(aclrtCreateContext(&context, deviceId));
    aclrtStream stream = nullptr;
    CHECK_ACL(aclrtCreateStream(&stream));
    uint8_t *xHost, *yHost, *zHost;
    uint8_t *xDevice, *yDevice, *zDevice;
    // 使用ACL接口分配Host上内存
    CHECK_ACL(aclrtMallocHost((void**)(&xHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&yHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&zHost), outputByteSize));
    // 使用ACL接口分配device上内存
    CHECK_ACL(aclrtMalloc((void**)&xDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&yDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&zDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // 初始化测试数据数组，为了方便查看结果，此处只赋值了前4个数据
    uint16_t input_x [inputOutputNum] = {1,2,3,4};
    uint16_t input_y [inputOutputNum] = {5,6,7,8};
    // 把测试数据放入分配的Host内存
    xHost = (uint8_t *)input_x;
    yHost = (uint8_t *)input_y;
    // Host侧数据搬到Device
    CHECK_ACL(aclrtMemcpy(xDevice, inputByteSize, xHost, inputByteSize, ACL_MEMCPY_HOST_TO_DEVICE));
    CHECK_ACL(aclrtMemcpy(yDevice, inputByteSize, yHost, inputByteSize, ACL_MEMCPY_HOST_TO_DEVICE));
    // 调用测试函数运算
    add_custom_do(blockDim, nullptr, stream, xDevice, yDevice, zDevice);
    CHECK_ACL(aclrtSynchronizeStream(stream));
    // Device运算结果搬出到Host侧内存
    CHECK_ACL(aclrtMemcpy(zHost, outputByteSize, zDevice, outputByteSize, ACL_MEMCPY_DEVICE_TO_HOST));
    uint16_t *result = (uint16_t *)zHost;
    // 打印出结果，此处我们只查看前4个数据运算结果
    for(size_t i = 0; i < 4; i++){
        std::cout<<"第["<< i <<"]个输入x = "<< input_x[i] <<"  输入y = "<< input_y[i]<<" 结果 = "<< result[i];
    }
    CHECK_ACL(aclrtFree(xDevice));
    CHECK_ACL(aclrtFree(yDevice));
    CHECK_ACL(aclrtFree(zDevice));
    CHECK_ACL(aclrtFreeHost(xHost));
    CHECK_ACL(aclrtFreeHost(yHost));
    CHECK_ACL(aclrtFreeHost(zHost));
    CHECK_ACL(aclrtDestroyStream(stream));
    CHECK_ACL(aclrtDestroyContext(context));
    CHECK_ACL(aclrtResetDevice(deviceId));
    CHECK_ACL(aclFinalize());
    return 0;
}
```

```
};

extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}

#ifndef __CCE_KT_TEST__
// call of kernel function
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y, uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
#endif
```

NPU侧调试时，算子实现文件内需要添加调试函数

```
第[0]个输入x = 4 输入y = 5 结果 = 9
第[1]个输入x = 3 输入y = 6 结果 = 9
第[2]个输入x = 2 输入y = 7 结果 = 9
第[3]个输入x = 1 输入y = 8 结果 = 9
```


THANKS