进阶实验篇第5章

矩阵乘：面向硬件加速器的优化

第二部分：OpenCL

1 加速器的工作原理（以GPU为例）

2 OpenCL编程基础

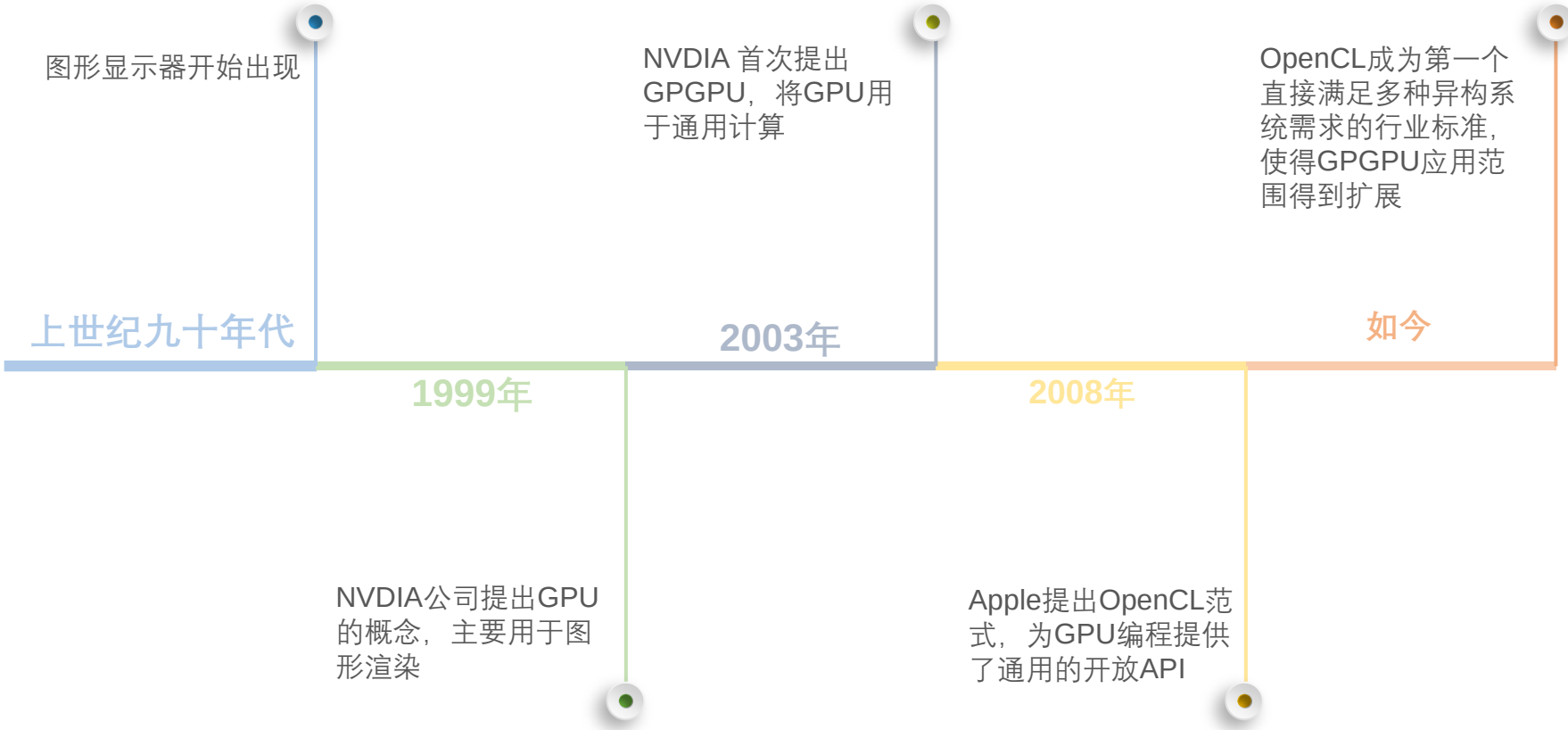
3 任务实现：基于OpenCL的矩阵乘法

4 性能分析与数据分段分析

5 延伸阅读：CXL协议

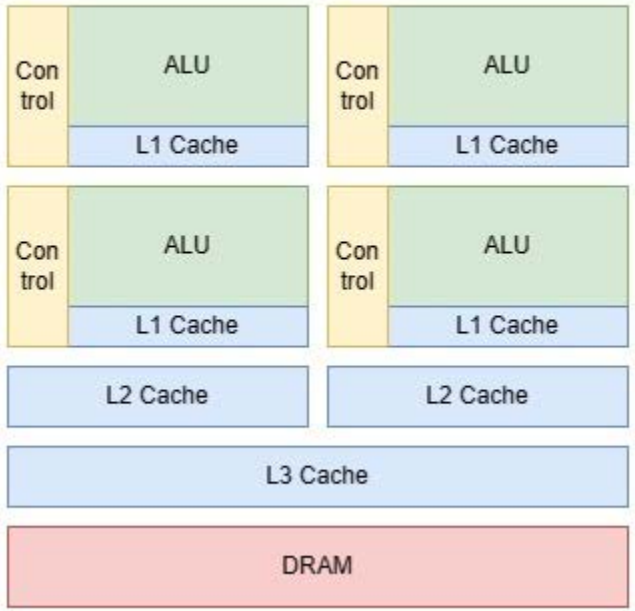
加速器的工作原理（以GPU为例）

GPU的前世今生

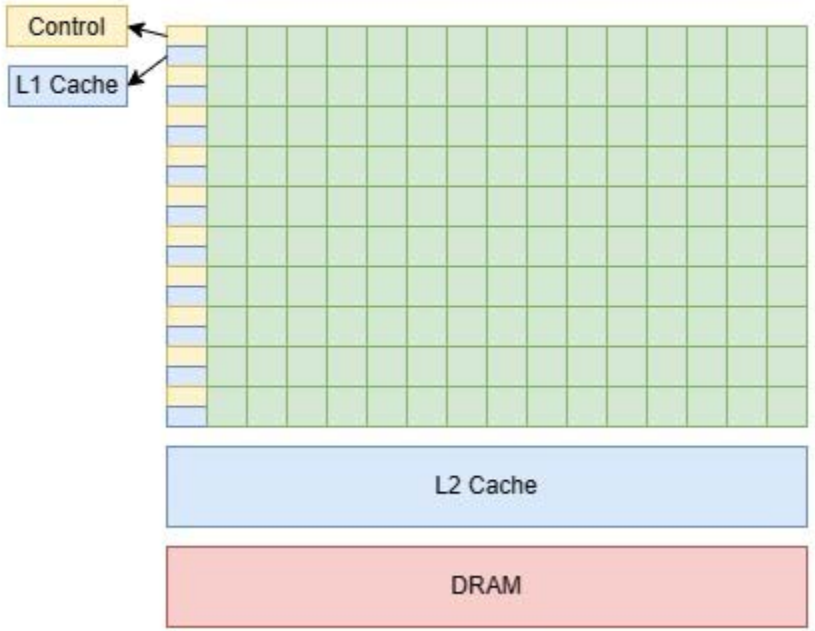

图形显示器开始出现
上世纪九十年代
NVDIA公司提出GPU的概念，主要用于图形渲染
1999年
NVDIA 首次提出GPGPU，将GPU用于通用计算
2003年
Apple提出OpenCL范式，为GPU编程提供了通用的开放API
2008年
OpenCL成为第一个直接满足多种异构系统需求的行业标准，使得GPGPU应用范围得到扩展
如今

CPU & GPU

CPU架构图



GPU架构图



- GPU的ALU数量远远多于CPU
- GPU的Cache和Control数量远远少于CPU
- 多个ALU（Core）只有一个Control，同时只能执行同样的命令

SIMT

GPU更擅长处理逻辑判断简单、数据量大的数据并行型和计算密集型任务

CPU+GPU的异构计算架构（分离式架构）

主机代码

编译

主机架构的机器码

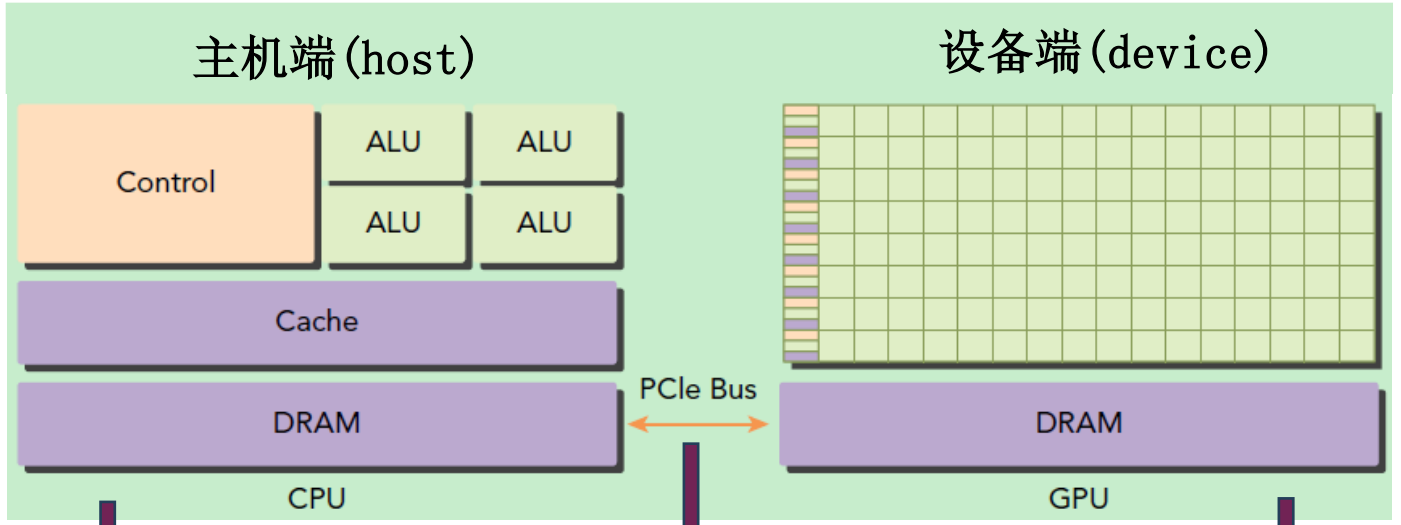
运行

运行

设备代码

编译

设备架构的机器码



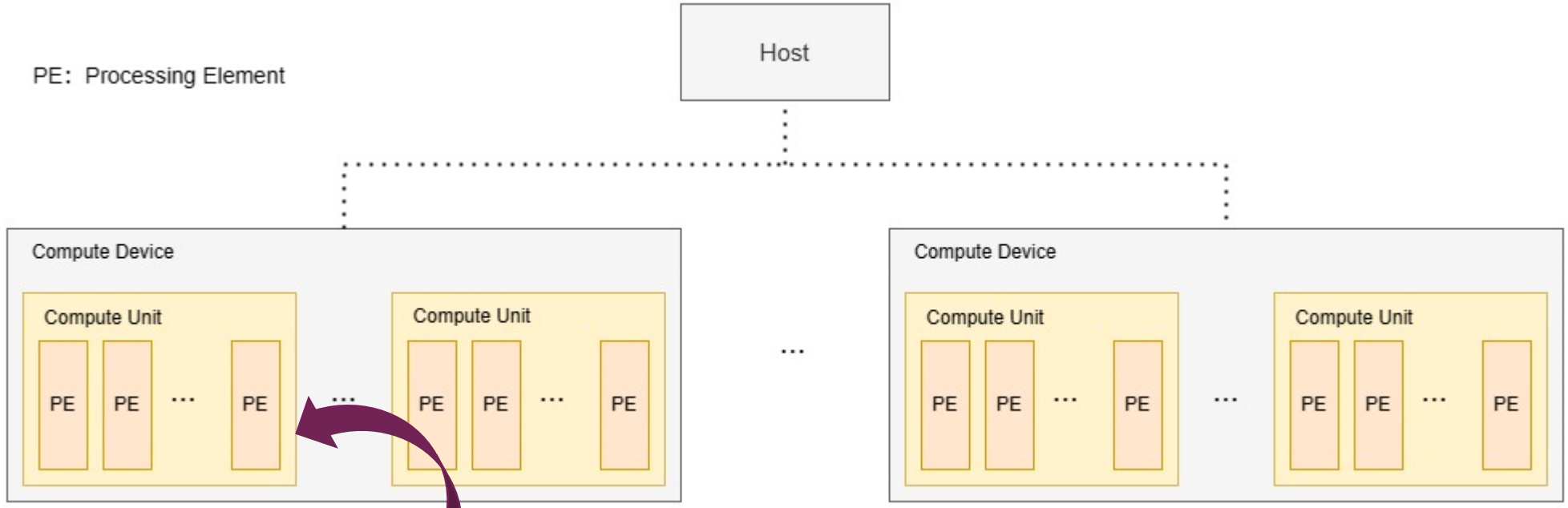
- 控制设备
- 完成数据传输等

传递指令和数据

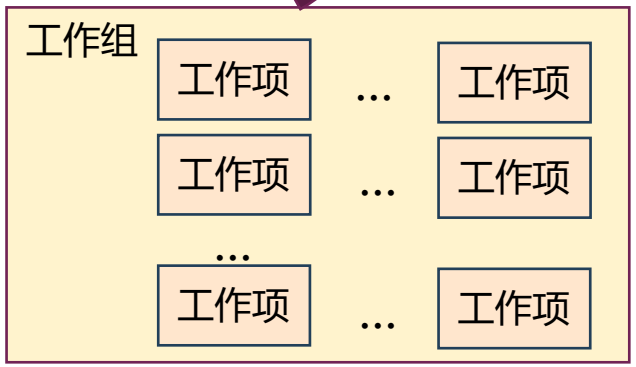
完成计算

OpenCL编程基础

OpenCL平台模型



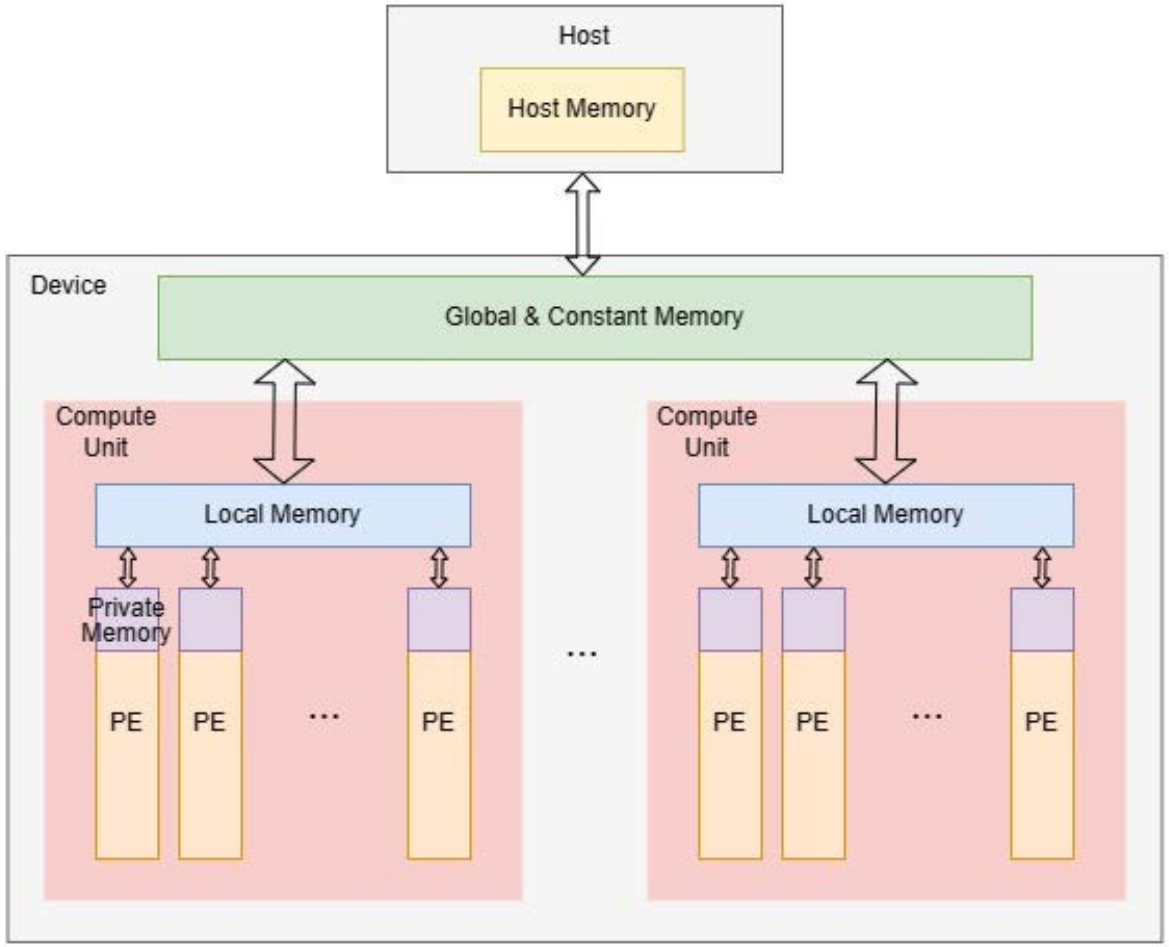
内核函数：



索引空间：

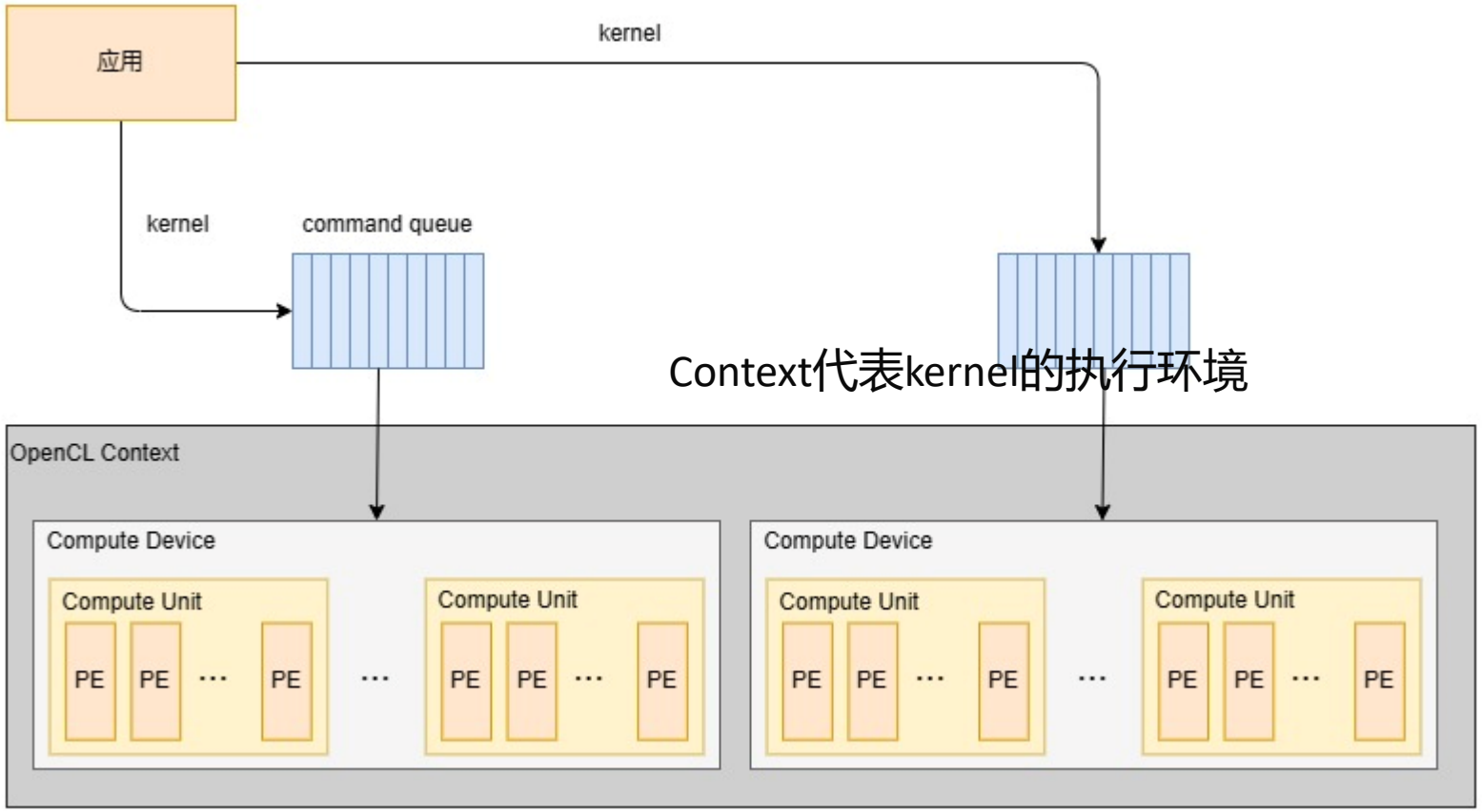
- get_global_id()
  获取工作项全局ID
- get_local_id()
  获取工作项在工作组中的局部ID
- get_group_id()
  获取工作组在索引空间的ID

OpenCL内存模型



- 主机内存
  - 仅Host端直接访问
- 全局内存
  - 设备上所有工作项可见
  - __global__限定符标识
- 常量内存
  - 允许Host进行读写
  - device只能进行读操作
  - __constant__限定符标识
- 局部内存
  - 同一工作组内工作项可进行读写
  - __local__限定符标识
- 私有内存
  - 单个工作项可见
  - __private__限定符标识

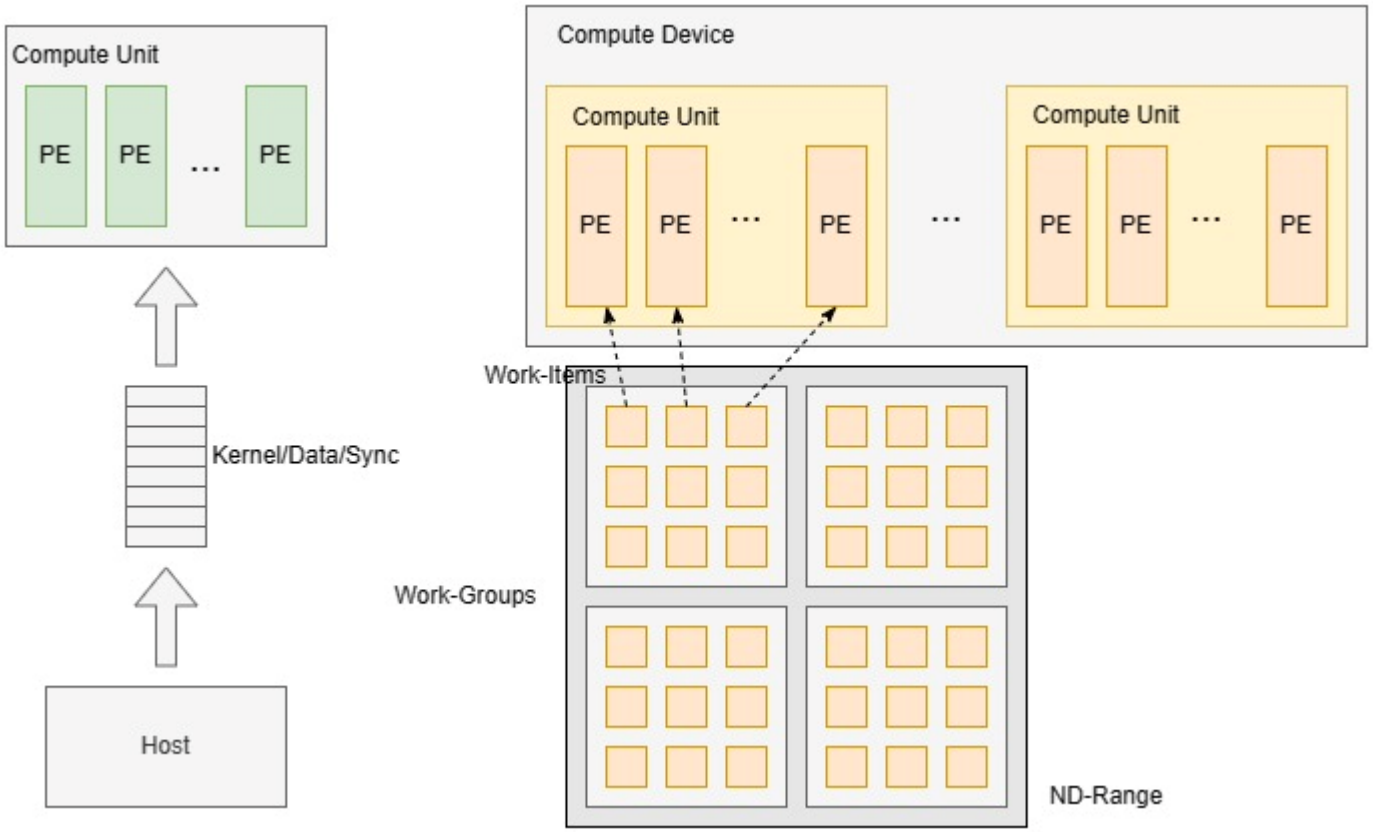
OpenCL执行模型——Context



Context：代表kernel的执行环境

- 抽象容器
- 协调host和device之间的交互
- 管理device上的内存对象
- 跟踪kernel和程序

OpenCL执行模型——NDrange



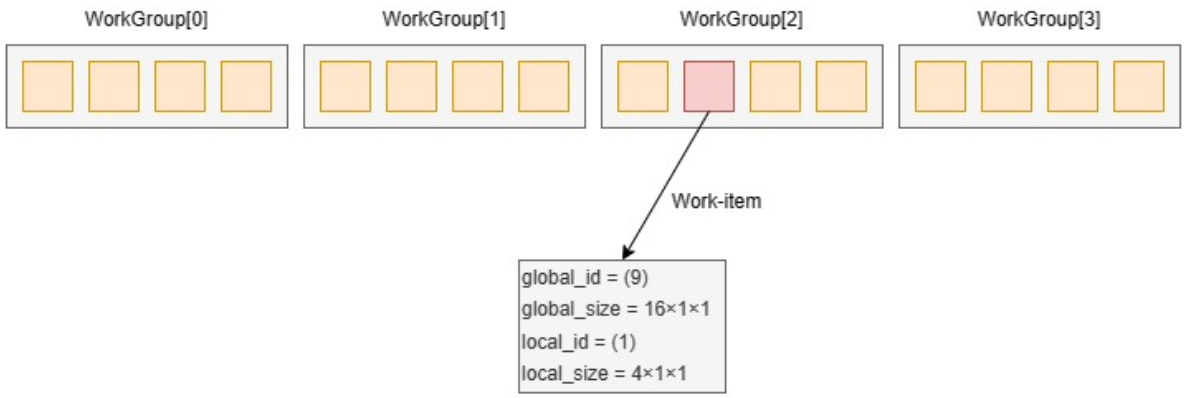
- kernel执行前，需创建一个索引空间NDRange（一维/二维/三维）
- 执行kernel的示例是 work-item
- work-item组织成 work-group
- work-group 组织成 NDRange
- NDRange映射到OpenCL Device计算单元上

OpenCL索引

寻找work-item的两种方式：

- 通过work-item的全局索引
- 先找到work-group的索引号，再根据局部索引确定

一维：

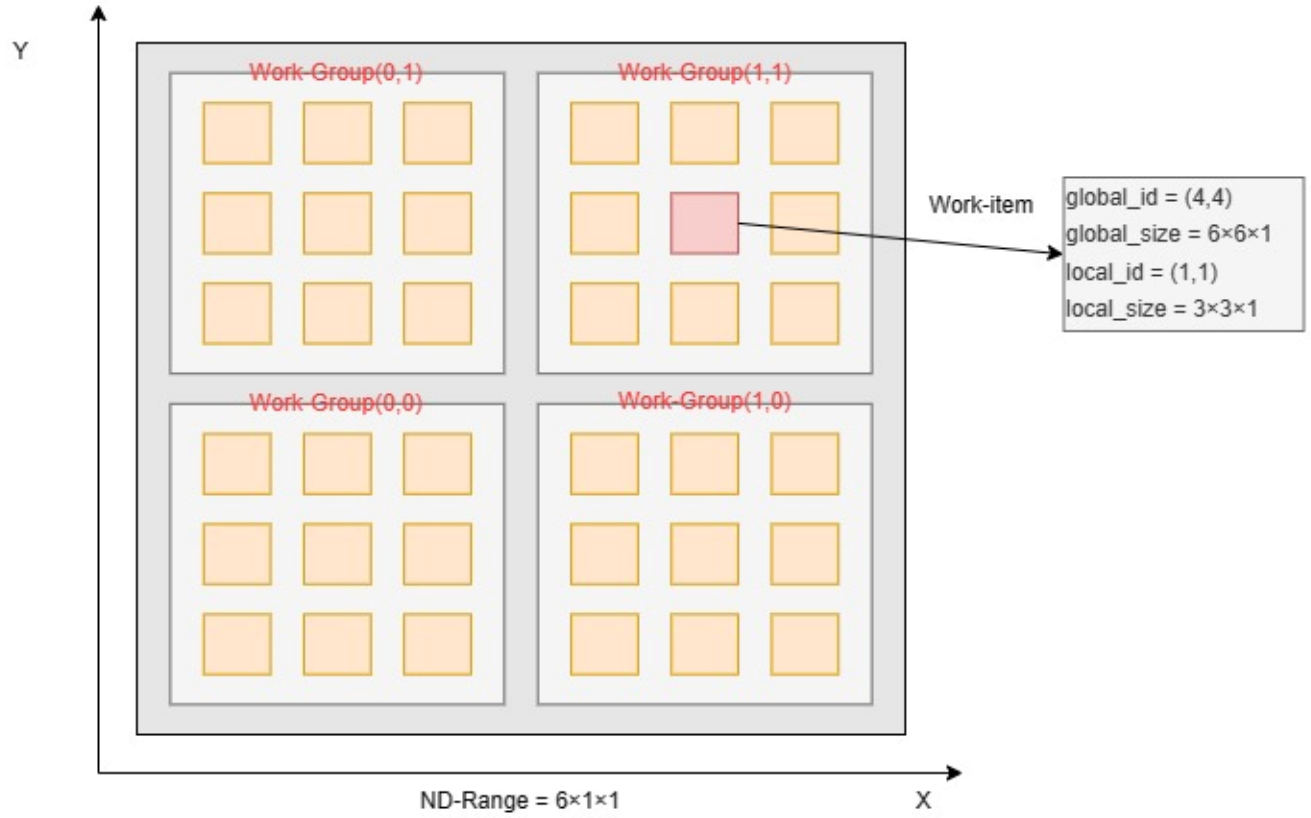


OpenCL索引

寻找work-item的两种方式：

- 通过work-item的全局索引
- 先找到work-group的索引号，再根据局部索引确定

二维：



任务实现：基于OpenCL的矩阵乘法

核函数示例

基于单处理器实现的顺序矩阵乘法：

```
void seqmatrixMul(int n, int m, int p, float* A, float* B, float* C){
    int i, j, k;
    float tmp;
    for(i = 0; i < n;i++){
        for(j = 0;j < m;j++){
            tmp = 0.0;
            for(k = 0;k < p;k++){
                // C[i][j] += A[i][k] * B[k][j];
                tmp += *(A + (i * p + k)) * *(B + (k * m + j));
            }
            *(C + (i * m + j)) = tmp;
        }
    }
}
```

映射

单个work-item ← 读取 ← A（N×P）的某行 B（P×M）的某列

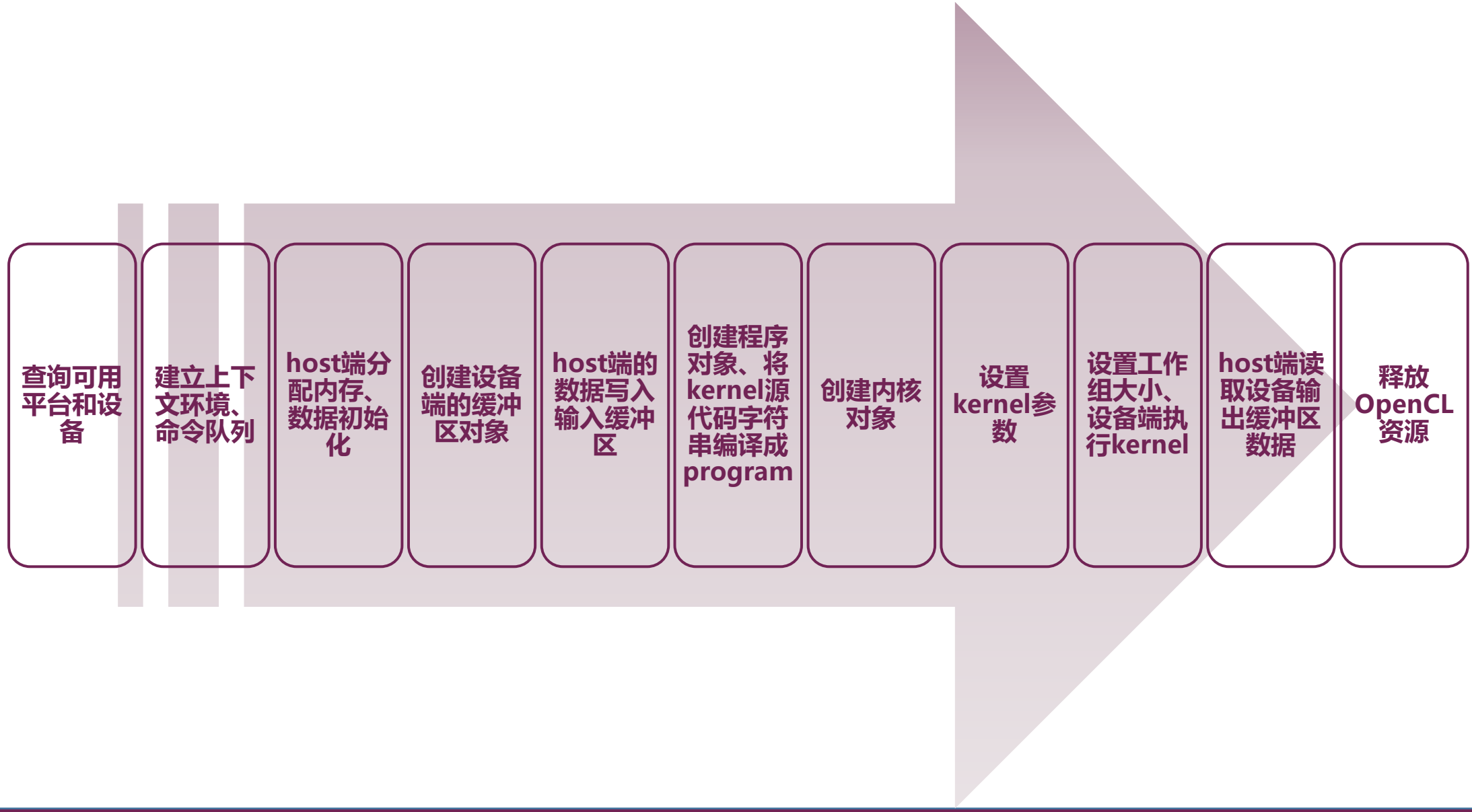
OpenCL的Kernel函数：

```
__kernel matrixMul(
const int n, const int m, const int p,
__global float* A, __global float* B, __global float* C)
{
    int k;
    int row = get_global_id(1); // 获取当前工作项在Y方向上的坐标
    int col = get_global_id(0); // 获取当前工作项在X方向上的坐标
    float tmp;
    if((row < n) && (col < m))
    {
        tmp = 0.0;
        for(k = 0;k < p;k++)
        {
            tmp += A[row * p + k] * B[k * m + col];
        }
        C[row * m + col] = tmp;
    }
}
```

二维索引空间：

- get_global_id(0) 获得工作项在水平方向上坐标
- get_global_id(0) 获得工作项在竖直方向上坐标

OpenCL主机程序执行流程



查询可用平台和设备

API介绍：

获取各个平台信息

```
cl_int clGetPlatformIDs(
        cl_uint num_entries,
        cl_platform_id* platforms,
        cl_uint* num_platforms)
```

num_entries	用于指定要获取的平台数量
platforms	返回找到的OpenCL平台对象的列表
num_platforms	返回可用的 OpenCL 平台的数量

选中可用设备

```
cl_int clGetDeviceIDs(
        cl_platform_id platform,
        cl_device_type device_type,
        cl_uint num_entries,
        cl_device_id* devices,
        cl_uint* num_devices)
```

platform	平台ID	devices	返回获取的OpenCL设备对象的数组
device_type	指定要获取的设备类型	num_devices	返回平台所连接的与device_type匹配的可用设备数量
num_entries	指定要获取的设备数量		

代码示例：

查询平台名字符串

```
cl_int err_num;
cl_uint num_platform;
cl_platform_id *platform_list;
err_num = clGetPlatformIDs(0, NULL, &num_platform);
platform_list = (cl_platform_id *)malloc(sizeof(cl_platform_id) *
num_platform);
err_num = clGetPlatformIDs(num_platform, platform_list, NULL);
```

获得可用平台列表

```
size_t size;
// 第一次调用获取字符串长度
err_num = clGetPlatformInfo(platform_list, CL_PLATFORM_NAME, 0, NULL,
&size);
char *name = (char *)malloc(sizeof(char) * size);
// 第二次调用获取平台信息到name
err_num = clGetPlatformInfo(platform_list, CL_PLATFORM_NAME,
size, name, NULL);
printf("%s=%s\n", "Platform_Name", name);
```

建立上下文环境

API介绍：

```
cl_context clCreateContext(
        const cl_context_properties* properties,
        cl_uint num_devices,
        const cl_device_id* devices,
        void (CL_CALLBACK* pfn_notify)(const char* errinfo,
            const void* private_info, size_t cb, void* user_data),
        void* user_data,
        cl_int* errcode_ret)
```

properties	用来描述上下文属性的数组
pfn_notify	由主机程序注册的回调函数
num_platforms	可用的 OpenCL 平台的数量
user_data	传递给回调函数pfn_notify的指针参数
errcode_ret	用来标识上下文的创建情况

代码示例：

创建数据的上下文

```
cl_context context = NULL;
cl_context_properties context_properties[3] =
{
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)platform_list[0],
    0
};
context = clCreateContext(context_properties, 1, &device, NULL,
NULL, &err_num);
```

建立命令队列

API介绍：

```
cl_command_queue clCreateCommandQueueWithProperties(
        cl_context context,
        cl_device_id device,
        const cl_queue_properties* properties,
        cl_int* errcode_ret)
```

context	一个有效的上下文
device	与context关联的设备
properties	一个用来描述命令队列属性的数组
errcode_ret	返回的错误码，用来标识命令队列的创建情况

代码示例：

创建命令队列

```
cl_command_queue command_queue = NULL;
cl_queue_properties queue_properties[3] =
{
    CL_QUEUE_PROPERTIES,
    CL_QUEUE_PROFILING_ENABLE,
    0
};
command_queue = clCreateCommandQueueWithProperties(
context, device, queue_properties, &err_num);
```

创建设备端缓冲区对象

API介绍：

```
cl_mem clCreateBuffer(
        cl_context context,
        cl_mem_flags flags,
        size_t size,
        void* host_ptr,
        cl_int* errcode_ret)
```

context	表示为此context对象分配buffer对象
flags	描述此buffer的属性
size	表示要申请的缓冲区对象所占的内存空间的字节数
host_ptr	指向应用程序在主机端中已经分配的数据的指针
errcode_ret	返回的错误码

将所处理的计算数据封装为内存对象

代码示例：

```
cl_mem buffer_A = clCreateBuffer(context, CL_MEM_READ_ONLY,

sizeof(float) * sizeA, NULL, &err_num);
cl_mem buffer_B = clCreateBuffer(context, CL_MEM_READ_ONLY,

sizeof(float) * sizeB, NULL, &err_num);
cl_mem buffer_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY,

sizeof(float) * sizeC, NULL, &err_num);
```

- buffer_A、buffer_B为输入缓冲区
  用来存储输入矩阵A、B
- buffer_C为输出缓冲区
  用来存储输出矩阵C。

host数据写入device输入缓冲区

API介绍：

```
cl_int clEnqueueWriteBuffer(
        cl_command_queue command_queue,
        cl_mem buffer,
        cl_bool blocking_write,
        size_t offset,
        size_t size,
        const void* ptr,
        cl_uint num_events_in_wait_list,
        const cl_event* event_wait_list,
        cl_event* event)
```

command_queue	表示此命令要入队的命令队列对象
buffer	一个有效的缓冲区对象
blocking_write	用于设置写命令是否是阻塞式操作
offset	缓冲区对象中要写入的字节数偏移量
size	表示要传输的数据的字节数
ptr	host端内存缓冲区地址
num_events_in_wait_lis	表示在执行此命令前需要完成的事件数量
event_wait_list	表示在执行此命令前需要完成的事件
event	返回的一个标识此命令的事件对象

代码示例：

主机端内存数据传入到 OpenCL buffer中

```
err_num = clEnqueueWriteBuffer(command_queue, buffer_A, CL_TRUE, 0,
                                                    sizeof(float) * sizeA, A, 0, NULL, NULL);
err_num = clEnqueueWriteBuffer(command_queue, buffer_B, CL_TRUE, 0,
                                                    sizeof(float) * sizeB, B, 0, NULL, NULL);
```

创建并构建程序对象

API介绍：

```
cl_program clCreateProgramWithSource(
        cl_context context,
        cl_uint count,
        const char** strings,
        const size_t* lengths,
        cl_int* errcode_ret)
```

context	一个有效的上下文对象
count	表示参数strings指向的数组中的字符串个数
strings	一个指针数组，指向构成设备源代码的字符串
lengths	数组存储strings中每个字符串的长度
errcode_ret	返回的错误码

```
cl_int clBuildProgram(
      cl_program program,
      cl_uint num_devices,
      const cl_device_id* device_list,
      const char* options,
      void (CL_CALLBACK* pfn_notify)(cl_program program, void*
user_data), void* user_data)
```

program	要被构建的程序对象
num_devices	参数device_list中的设备数量
num_device_list	program关联的设备对象数组
options	表示构建选项
pfn_notify	一个可由主机程序注册的回调函
user_data	传给回调函数pfn_notify的指针参数

代码示例：

```
cl_program program;
program = clCreateProgramWithSource(context, 1,
                    (const char **)&source, NULL, &err_num);
err_num = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

创建内核对象

API介绍：

```
cl_kernel clCreateKernel(
        cl_program program,
        const char* kernel_name,
        cl_int* errcode_ret)
```

program	已经成功构建设备可执行文件的程序对象
kernel_name	源代码中用__kernel限定符修饰的函数名
errcode_ret	返回的错误码

代码示例：

```
cl_kernel kernel;
kernel = clCreateKernel(program, "matrixMul", &err_num);
```

设置kernel参数

```
err_num = 0;
err_num = clSetKernelArg(kernel, 0, sizeof(int), &n);
err_num |= clSetKernelArg(kernel, 1, sizeof(int), &m);
err_num |= clSetKernelArg(kernel, 2, sizeof(int), &p);
err_num |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &buffer_A);
err_num |= clSetKernelArg(kernel, 4, sizeof(cl_mem), &buffer_B);
err_num |= clSetKernelArg(kernel, 5, sizeof(cl_mem), &buffer_C);
```

设置工作组大小并执行kernel

API介绍：

```
cl_int clEnqueueNDRangeKernel(
        cl_command_queue command_queue,
        cl_kernel kernel,
        cl_uint work_dim,
        const size_t* global_work_offset,
        const size_t* global_work_size,
        const size_t* local_work_size,
        cl_uint num_events_in_wait_list,
        const cl_event* event_wait_list,
        cl_event* event)
```

command_queue	执行内核的命令所要入队的命令队列对象
kernel	要在设备上执行的内核函数
work_dim	工作组的组织维度
global_work_offset	工作项的全局ID的偏移量
global_work_size	表示全局工作项的数量
local_work_size	表示工作组中工作项的数量
num_events_in_wait_list	表示在执行此命令前需要完成的事件数量
event_wait_list	表示在执行此命令前需要完成的事件
event	返回的一个标识此命令的事件对象

代码示例：

```
size_t global_work_size[2] = {(size_t) m, (size_t) n};
size_t local_work_size[2] = {16, 16};
err_num = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
                                        global_work_size,
local_work_size, 0, NULL, NULL);
```

读取device输出缓冲区中的数据到host端

API介绍：

```
cl_int clEnqueueReadBuffer(
        cl_command_queue command_queue,
        cl_mem buffer,
        cl_bool blocking_read,
        size_t offset,
        size_t size,
        void* ptr,
        cl_uint num_events_in_wait_list,
        const cl_event* event_wait_list,
        cl_event* event)
```

command_queue	表示读命令所要入队的命令队列对象
buffer	一个有效的缓冲区对象
blocking_read	用于设置读命令是否是阻塞式操作
offset	读取数据在缓冲区对象中偏移的字节数偏移量
size	表示要传输的数据的字节数
ptr	host端内存缓冲区地址
num_events_in_wait_list	表示在执行此命令前需要完成的事件数量
event_wait_list	表示在执行此命令前需要完成的事件
event	返回的一个标识此命令的事件对象

代码示例：

```
err_num = clEnqueueReadBuffer(command_queue, buffer_C, CL_TRUE,
0,sizeof(float) * sizeC, C, 0, NULL, NULL);
```

释放OpenCL资源

API介绍：

cl_int clReleaseKernel(cl_kernel kernel)	用于释放内核对象的clReleaseKernel()
cl_int clReleaseProgram(cl_program program)	用于释放程序对象的clReleaseProgram()
cl_int clReleaseMemObject(cl_mem memobj)	用于释放内存对象的clReleaseMemObject()
cl_int clReleaseCommandQueue(cl_command_queue command_queue)	用于释放命令队列的clReleaseCommandQueue()
cl_int clReleaseContext(cl_context context)	用于释放上下文的clReleaseContext()

代码示例：

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemObject(buffer_A);
clReleaseMemObject(buffer_B);
clReleaseMemObject(buffer_C);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
```

性能分析与数据分段分析

性能分析

获取时间
- CPU计时函数
- OpenCL中的事件对象（event）

事件 → 传递命令状态的对象

事件的状态 → 描述了相关命令的状态

使用事件代码示例：

```
cl_event prof_event;
err_num = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
                                 global_work_size, local_work_size, 0, NULL, &prof_event);
clFinish(command_queue);
cl_ulong ev_start_time = (cl_ulong)0;
cl_ulong ev_end_time = (cl_ulong)0;
size_t return_size;
err_num = clGetEventProfilingInfo(prof_event,CL_PROFILING_COMMAND_START,
                                                                  sizeof(cl_ulong), &ev_start_time, NULL);
err_num = clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
                                                                  sizeof(cl_ulong), &ev_end_time, NULL);
```

API介绍:

clFinish()函数

用于阻塞程序的执行，直到命令队列中所有的命令都执行完成

```
cl_int clGetEventProfilingInfo(
       cl_event event,
       cl_profiling_info param_name,
       size_t param_value_size,
       void* param_value,
       size_t* param_value_size_ret)
```

用于抽取设备计时数据的接口	
event	指定的事件对象
param_name	要查询的评测数据
param_value_size	参数param_value 指向内存空间的字节数
param_value	一个指针，指向返回的查询参数结果
param_value_size_ret	实际复制到param_value的数据的字节数

性能分析

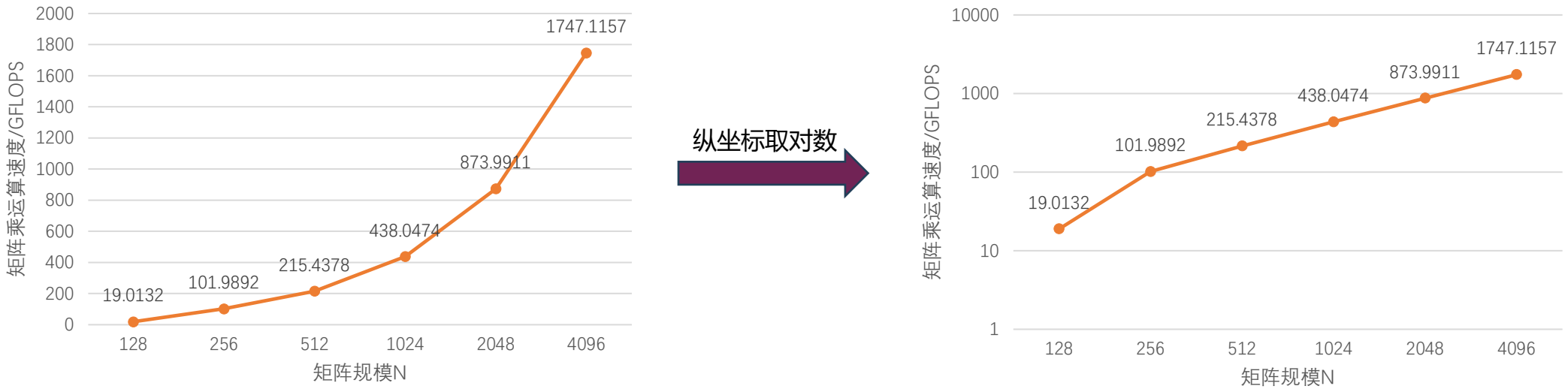
表 4 ：OpenCL 进行的不同规模矩阵乘的耗时和运算速度

矩阵规模 N	128	256	512	1024	2048	4096
矩阵乘耗时/ms	0.2206	0.329	1.246	4.9024	19.6568	78.6662
运算速度/GFLOPS	19.0132	101.9892	215.4378	438.0474	873.9911	1747.1157

OpenCL矩阵乘耗时的对数与矩阵规模N基本上成线性关系。

性能分析

画出OpenCL矩阵乘运算速度与矩阵规模之间的关系图如下：



OpenCL矩阵乘运算速度的对数与矩阵规模N基本上成线性关系

数据分段分析

CPU在调用硬件加速器

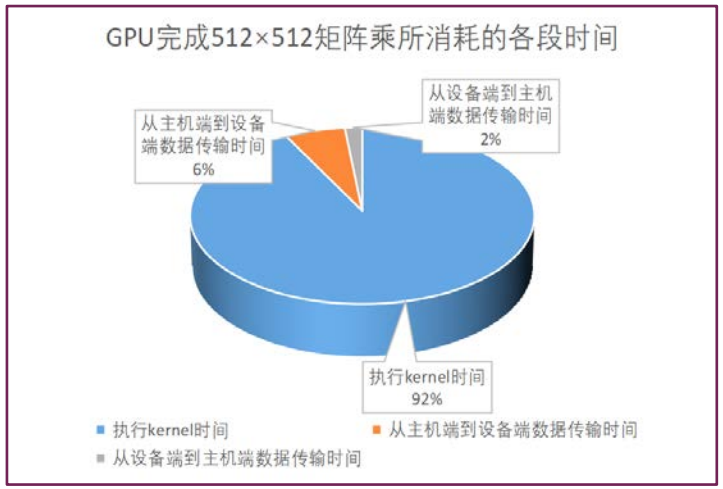
- 数据准备
- 数据传输
- 数据计算
- 数据返回

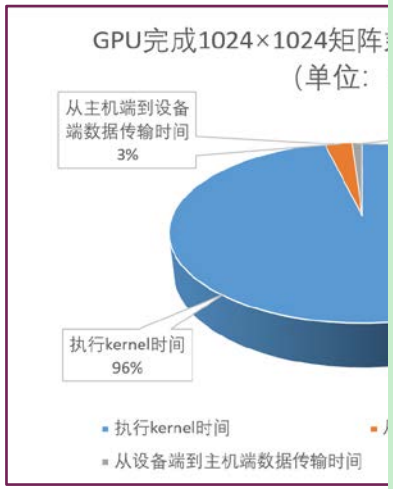
耗时？

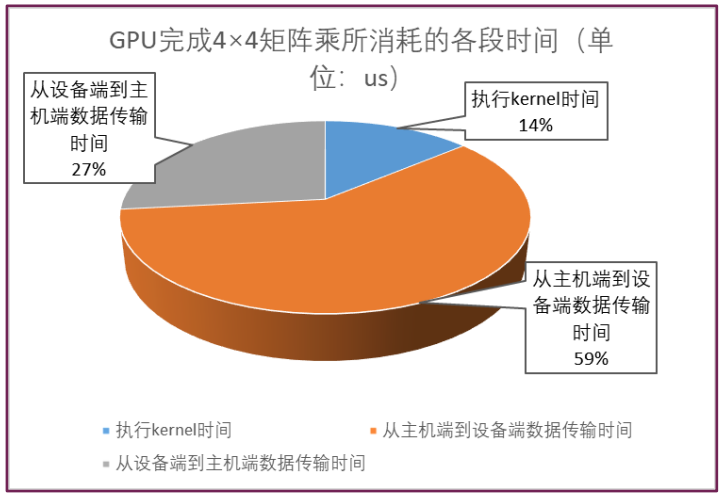
CPU定时函数的示例代码代码示例：

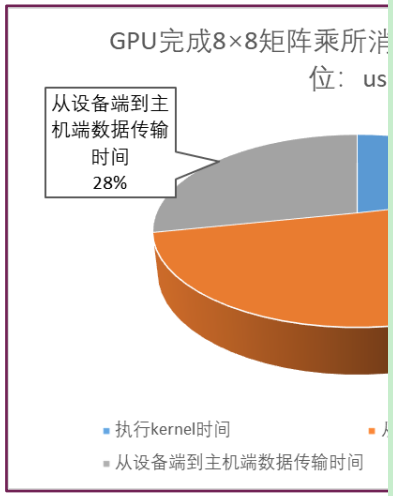
```
float esp_time_cpu_Data_Pre;
clock_t start_cpu, stop_cpu;
start_cpu = clock();// start timing
{
  //统计的代码段
  ……
}
stop_cpu = clock();// end timing
esp_time_cpu_Data_Pre = (float)(stop_cpu - start_cpu) /
CLOCKS_PER_SEC * 1000*1000;
printf("Time for preparing data: %f us\n", esp_time_cpu_Data_Pre);
```

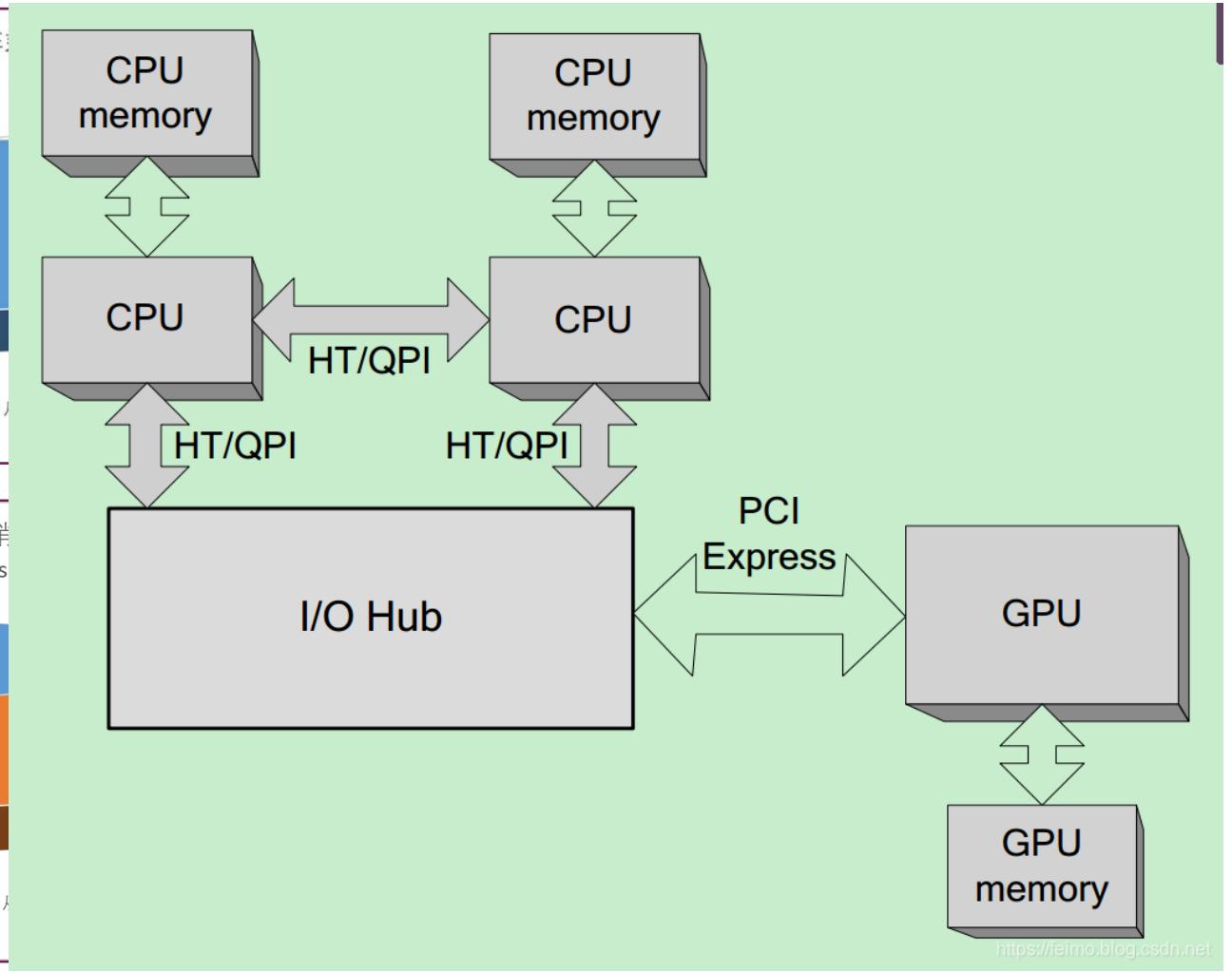
数据分段分析


GPU完成512×512矩阵乘所消耗的各段时间
从主机端到设备端数据传输时间 6%
从设备端到主机端数据传输时间 2%
执行kernel时间 92%
执行kernel时间
从主机端到设备端数据传输时间
从设备端到主机端数据传输时间


GPU完成1024×1024矩阵
（单位:
从主机端到设备端数据传输时间 3%
执行kernel时间 96%
执行kernel时间
从设备端到主机端数据传输时间


GPU完成4×4矩阵乘所消耗的各段时间（单位：us）
从设备端到主机端数据传输时间 27%
执行kernel时间 14%
从主机端到设备端数据传输时间 59%
执行kernel时间
从主机端到设备端数据传输时间
从设备端到主机端数据传输时间


GPU完成8×8矩阵乘所消
位：us
从设备端到主机端数据传输时间 28%
执行kernel时间
从设备端到主机端数据传输时间


CPU memory
CPU memory
CPU
CPU
HT/QPI
HT/QPI
HT/QPI
I/O Hub
PCI Express
GPU
GPU memory
https://feimo.blog.csdn.net

延伸阅读：CXL协议

CXL（Compute Express Link）技术

- 一种全新的互联技术标准和高速互联技术
- 以提高数据中心性能为目标
- 针对数据中心、高性能计算，人工智能等范畴
- 旨在改善传统高速串行计算机扩展总线标准系统中的通信路径
- ……

解决的问题：

- 内存墙
- IO墙

优势：

- 更快的数据传输速度
- 更低的延迟
- 更高的能耗
- 更强的可扩展性
- 更广泛的应用场景

中央处理器（Central Processing Unit，CPU）

更高效率、更快速度

- 与图形处理器（Graphics Processing Unit，GPU）
- 现场可编程逻辑门阵列（Field Programmable Gate Array，FPGA）
- …

应用：

- 高性能计算
- 存储加速
- 人工智能
- 网络加速
- AI加速
- 大规模虚拟化
- …

本章作业

程序设计

利用加速器实现矩阵乘法，并计量其性能

1. 依照4单元版本代码的示例完成8单元版本的SIMD矩阵乘法
2. 统计数据结果并计算其加速比

观察并完成实验报告

1. 运行GPU/加速器版本的代码，计量其性能和时间分配

感谢阅读