软件系统优化

异构计算与编程

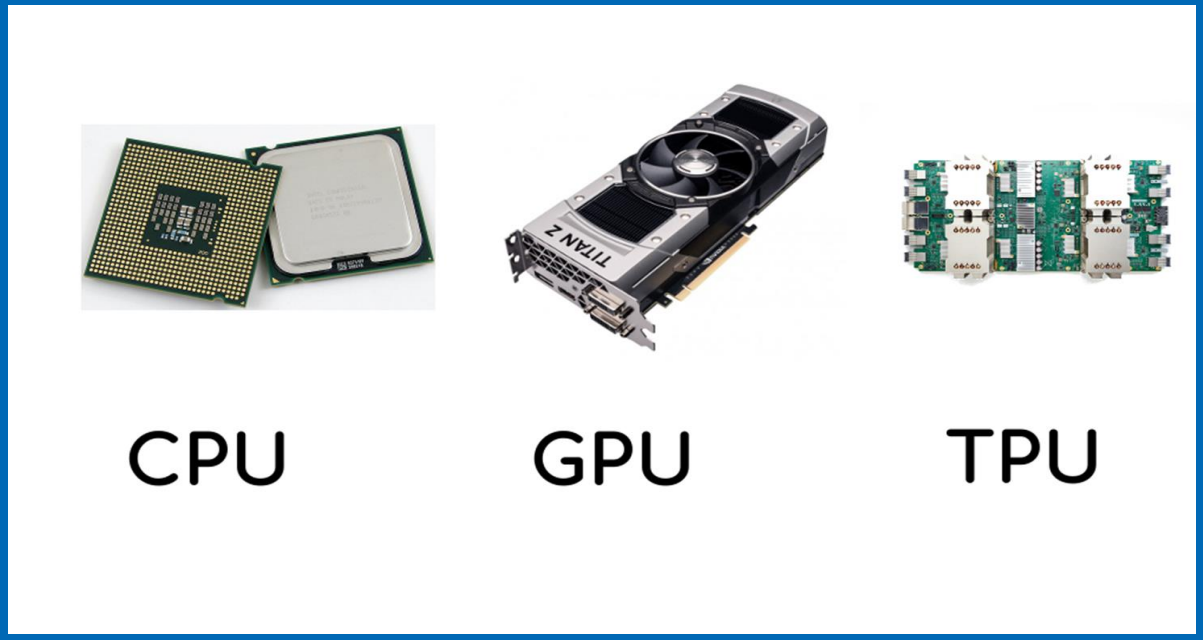
赵 鹏
英特尔数据与人工智能事业部
华东师范大学兼职副教授

intel®

课程大纲

- 异构计算概述
- 并行编程框架
  - 多核编程： Pthread/OpenMP
  - 多节点编程： MPI
  - 异构编程： DPC++
- 作业与练习

异构计算



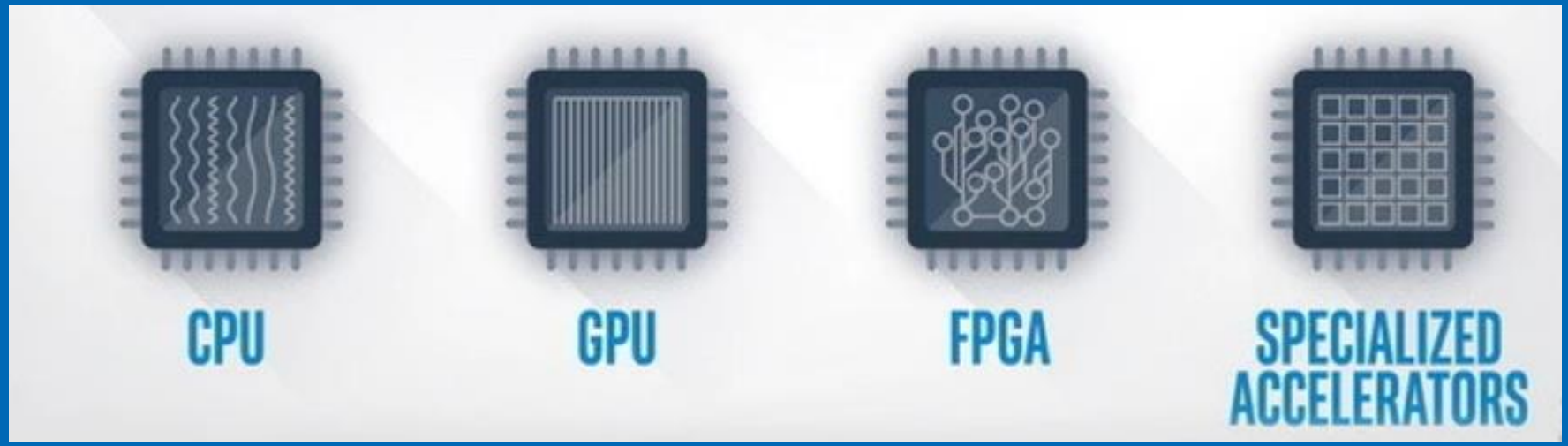
https://www.geekboots.com/story/cpu-vs-gpu-vs-tpu

简介

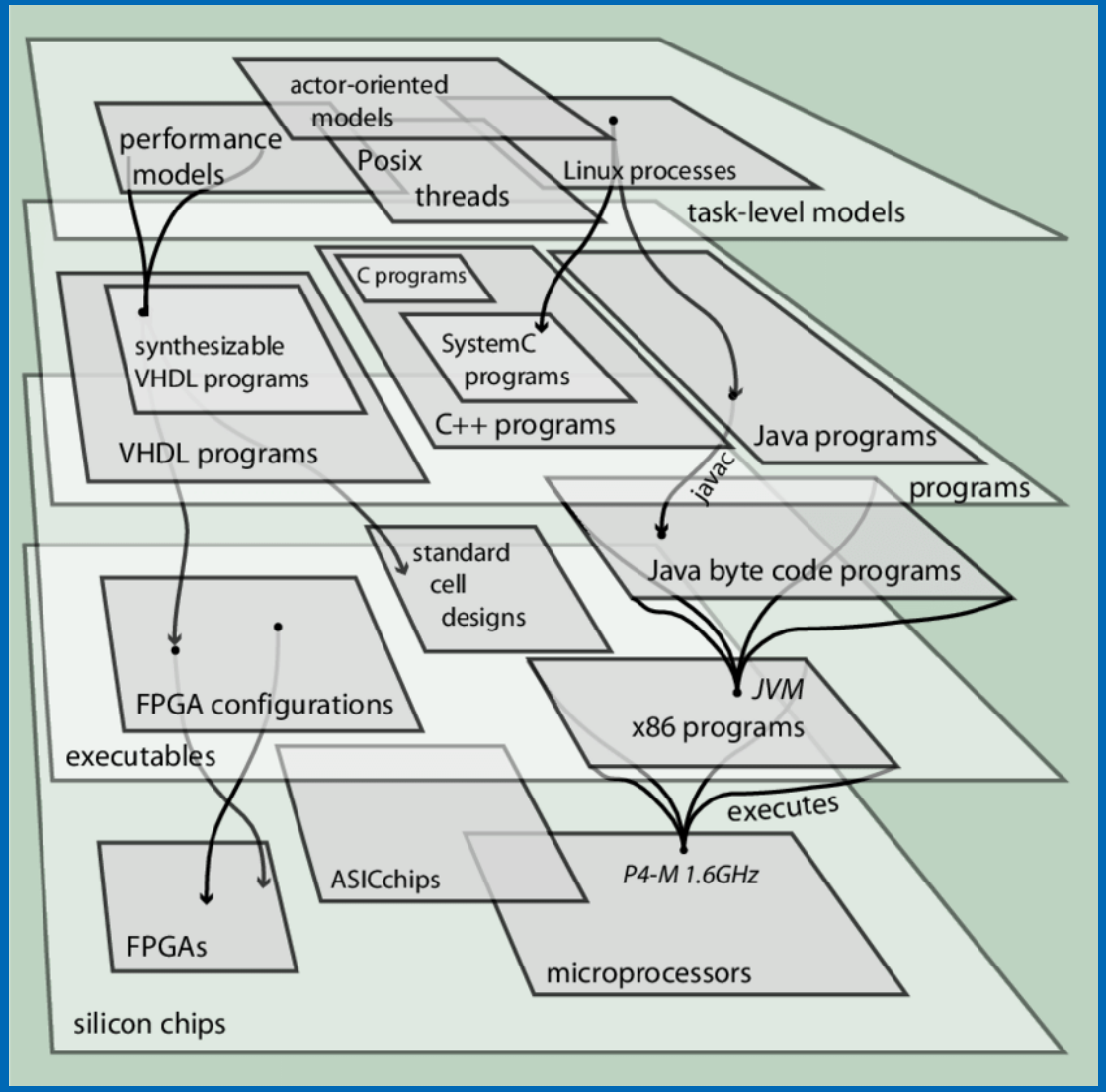
同构计算：相同类型指令集和体系架构的硬件，只是数量上的扩展，通常的同构计算指**多核计算**，即CPU中有更多的硬件核心。

异构计算：是指使用不同类型指令集和体系架构的计算单元组成系统的计算方式。

常见的计算单元类别包括CPU、GPU等协处理器、DSP、ASIC、FPGA等。

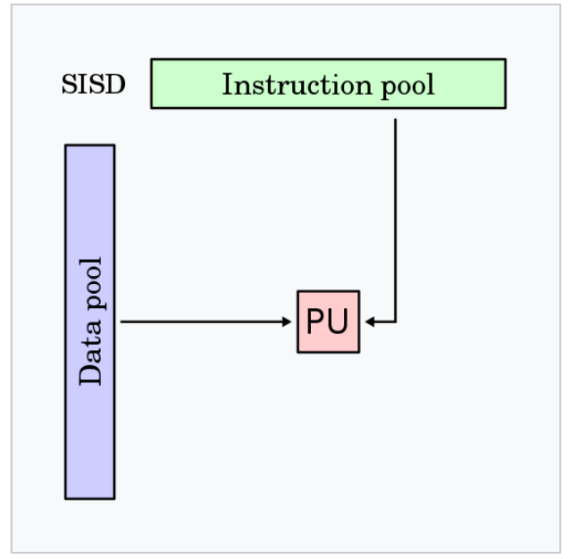


体系结构

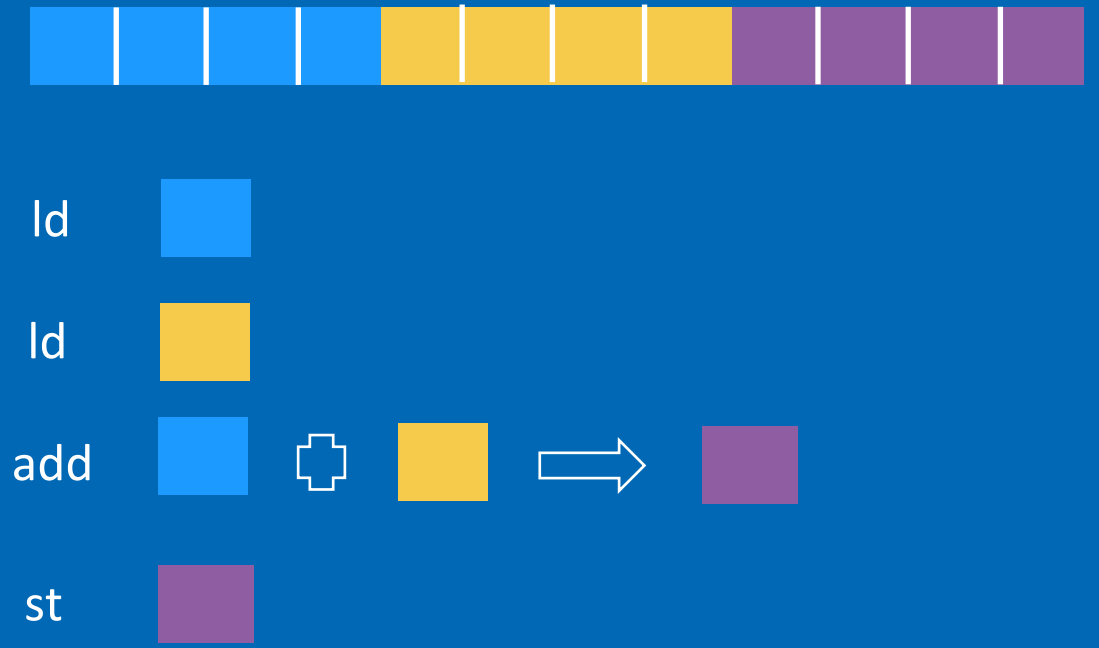

actor-oriented models
performance models
Posix threads
Linux processes
task-level models
C programs
synthesizable VHDL programs
SystemC programs
C++ programs
Java programs
VHDL programs
javac
programs
standard cell designs
Java byte code programs
FPGA configurations
JVM
x86 programs
executables
executes
ASICchips
P4-M 1.6GHz
FPGAs
microprocessors
silicon chips

异构计算硬件使用了不同的体系架构，我们首先要了解体系结构的分类方式。

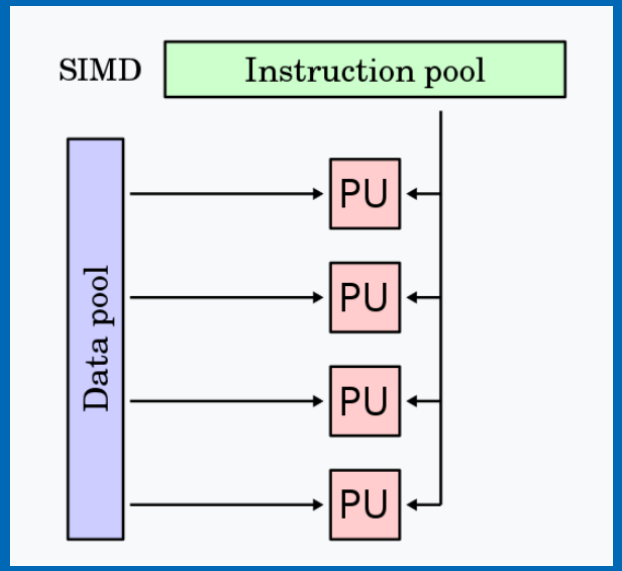
弗林（Flynn）分类法是最为广泛使用的抽象表达方法，其按照指令和数据的处理方式来分类。



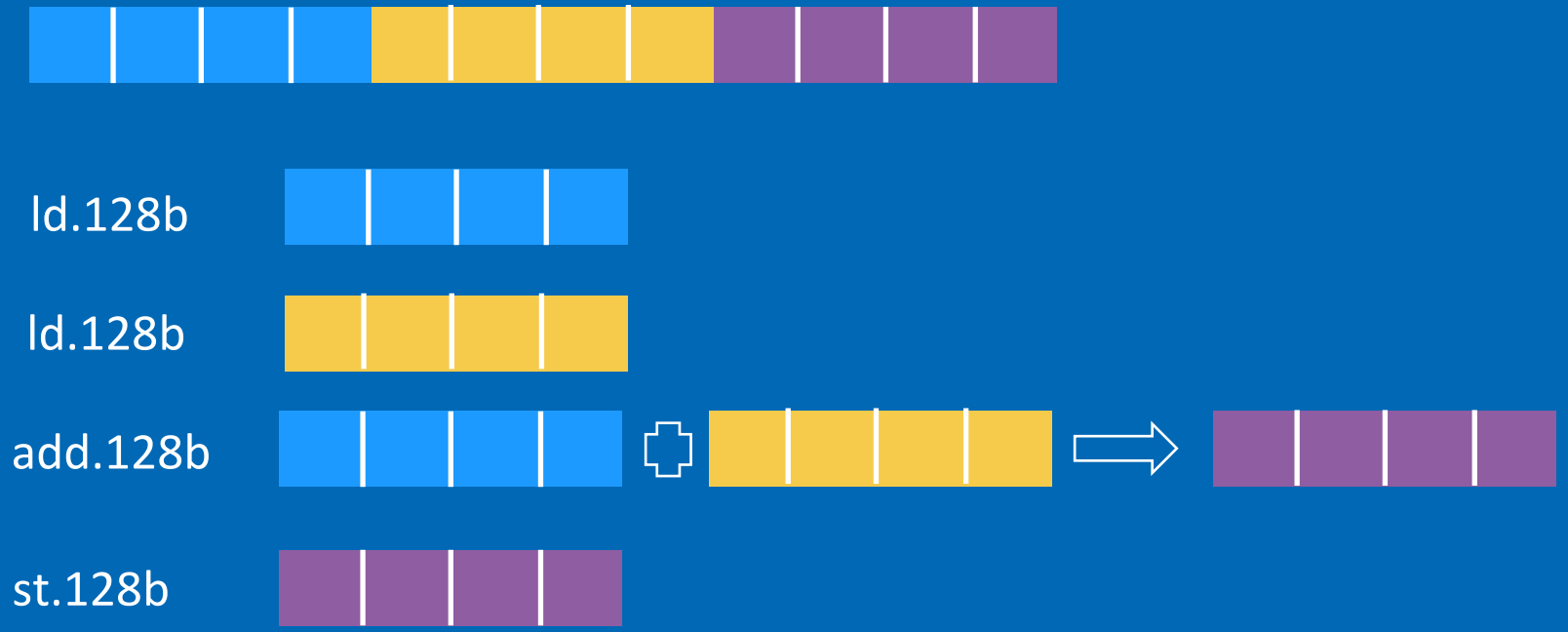
SISD： 单核CPU模式



单指令多数据（SIMD），向量化的执行模式，是现在CPU中主要的并行化方法之一，也同时广泛应用在各种加速卡上。提高数据吞吐量，但是对数据的连续性要求较高，不适合处理分散数据。

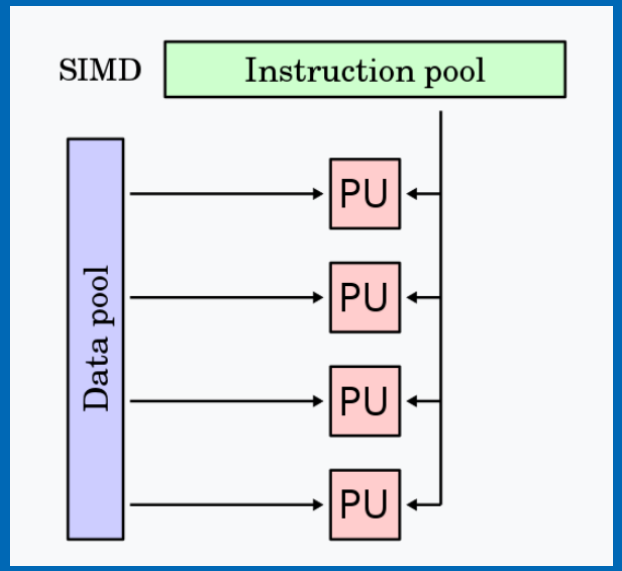


SIMD:向量化执行模式

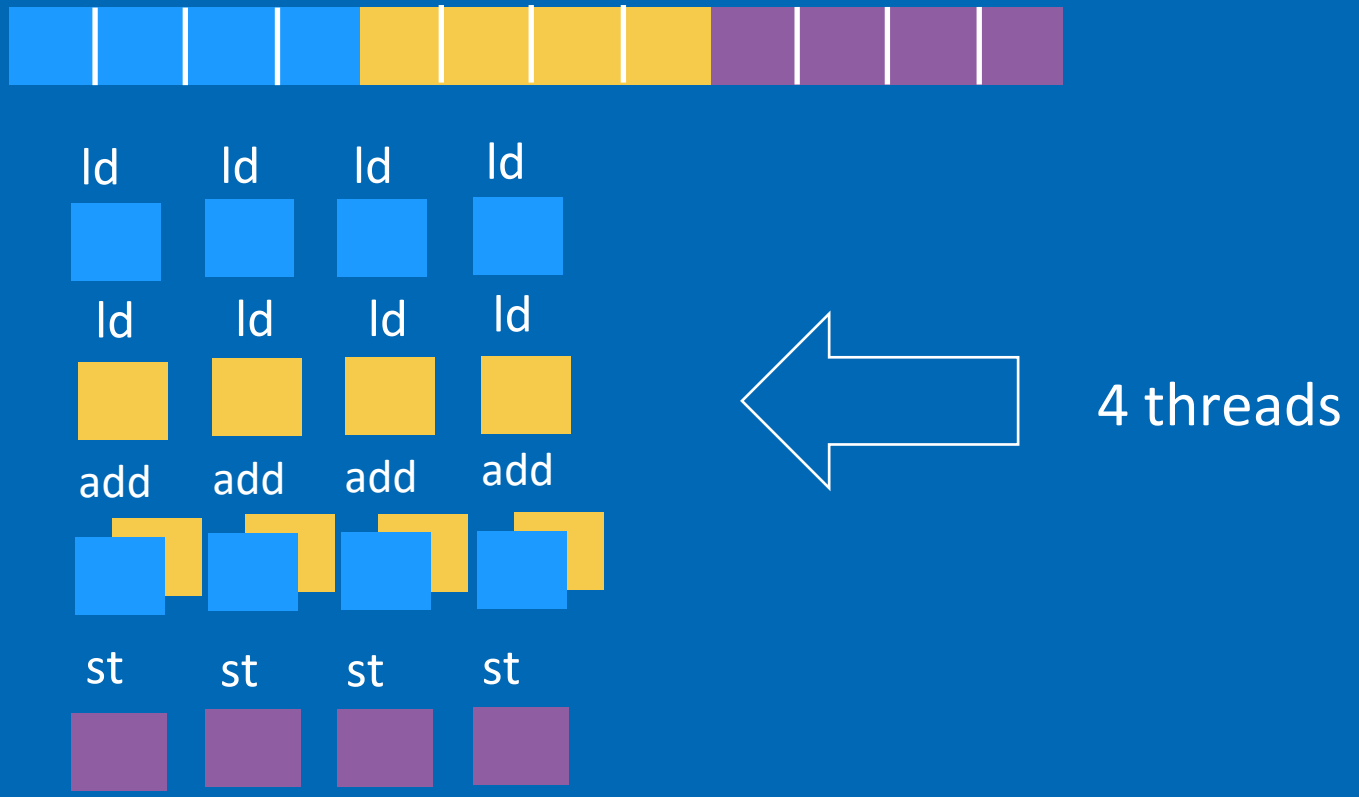


SIMD -> SIMT

单指令多线程（SIMT），一种基于线程视角的数据处理模式，每个线程处理一个数据，但是这些线程共享同一条指令。优点是多线程模式更加灵活，不需要数据的连续性，适合做数据的收集和分发（gather/scatter）操作，但是如果线程中有分支，效率会收到很大影响。

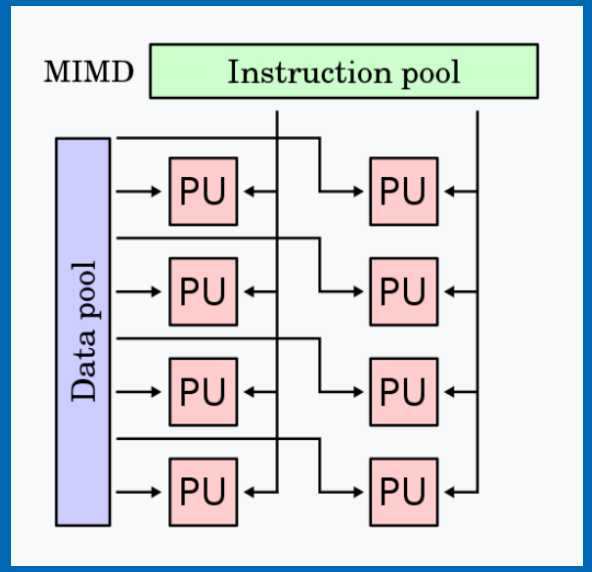


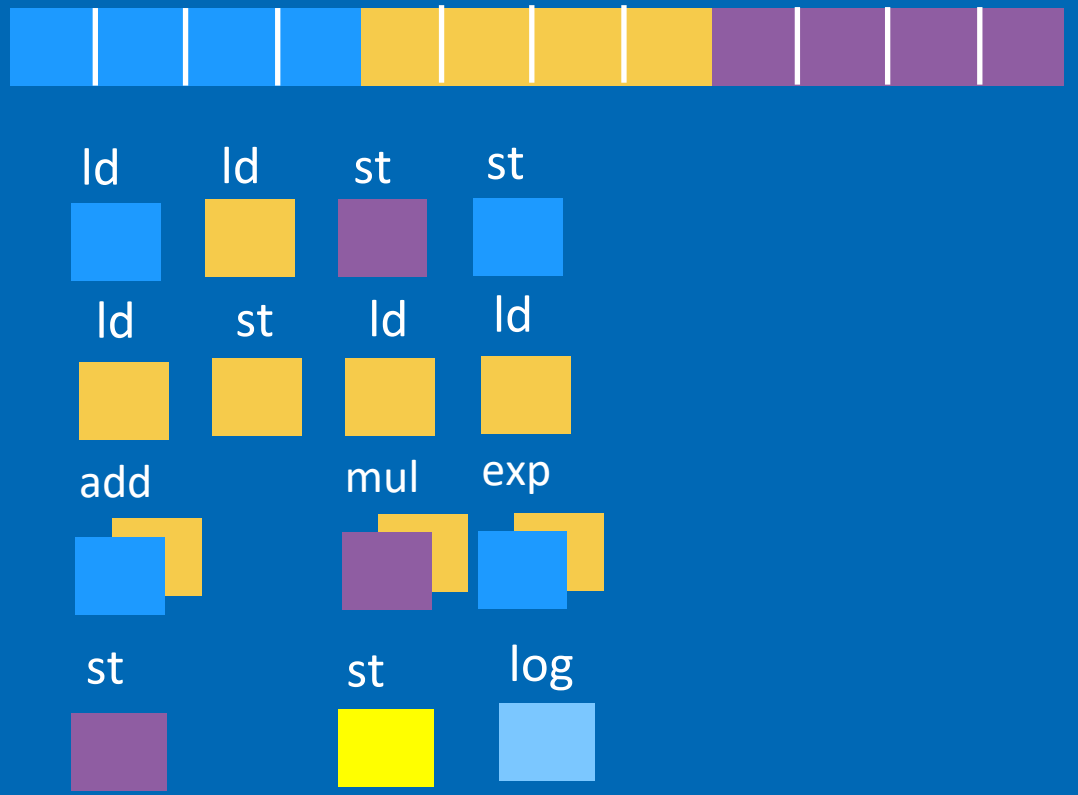
SIMT:单指令多线程



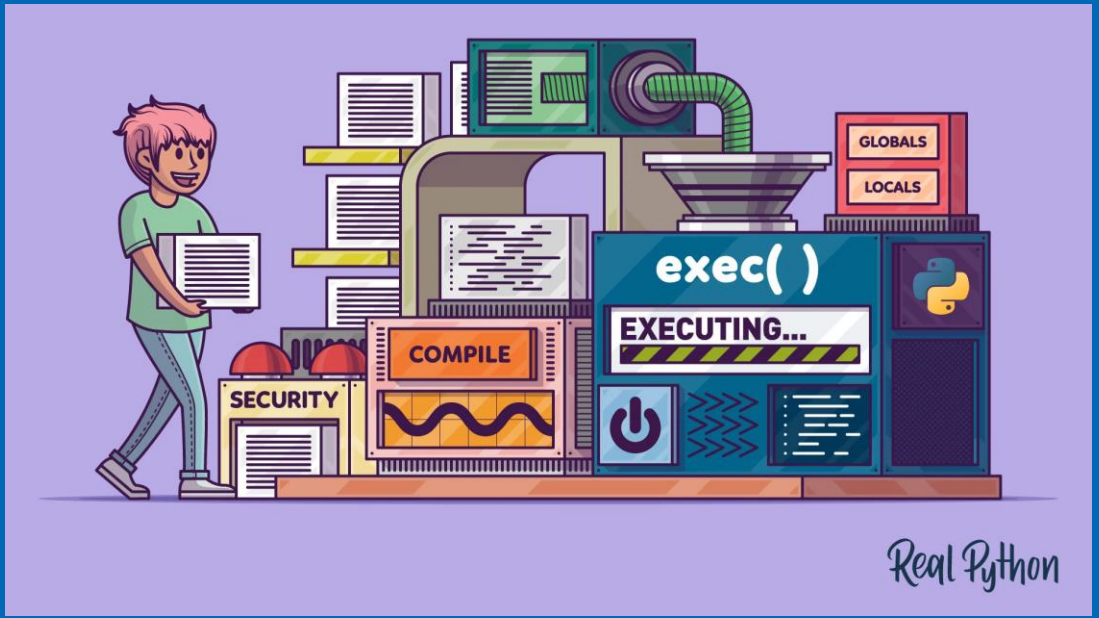
MIMD

多指令多数据流，一般用于多任务的并行模式。每一个计算单元都做各自独立的任务，并且读取各自独立的数据。其任务和数据之间都没有依赖性，所以并行程序设计方面较为简单。



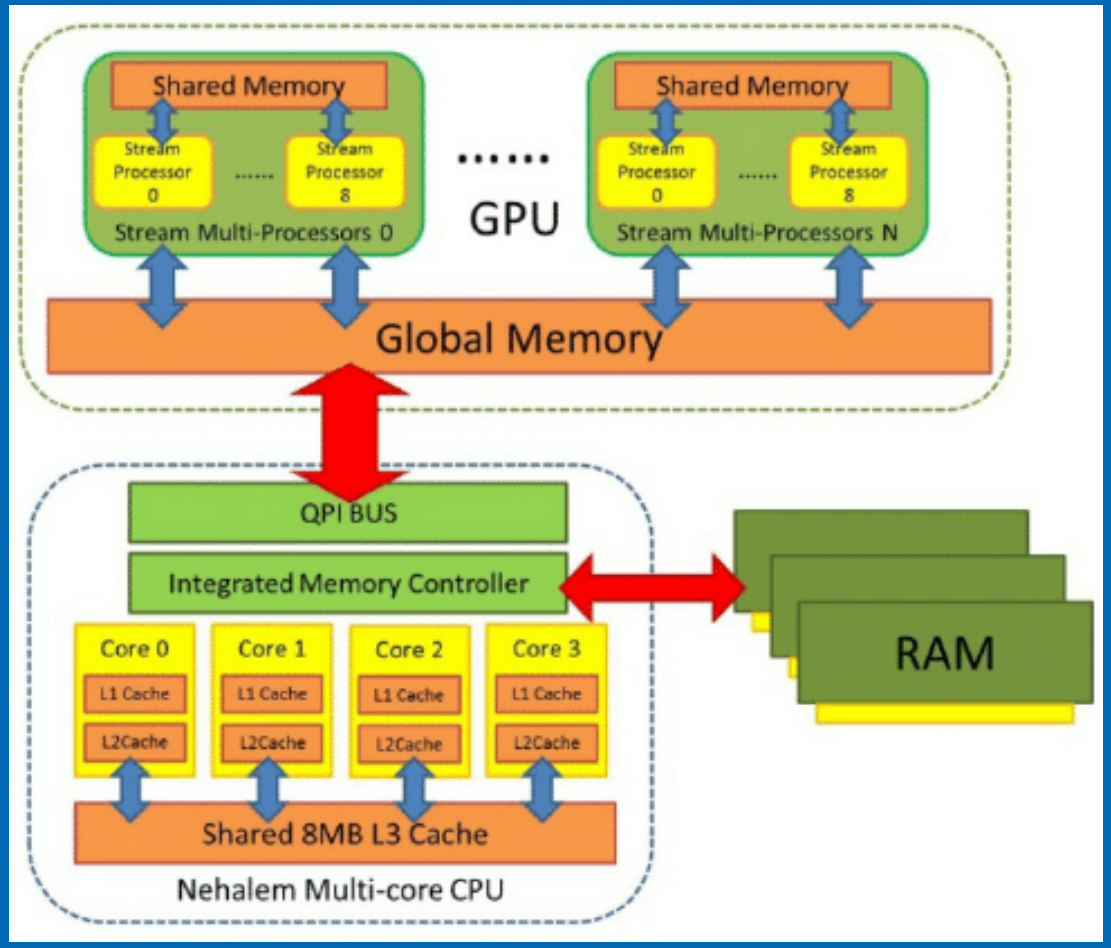


执行模型


GLOBALS
LOCALS
exec( )
EXECUTING...
COMPILE
SECURITY
Real Python

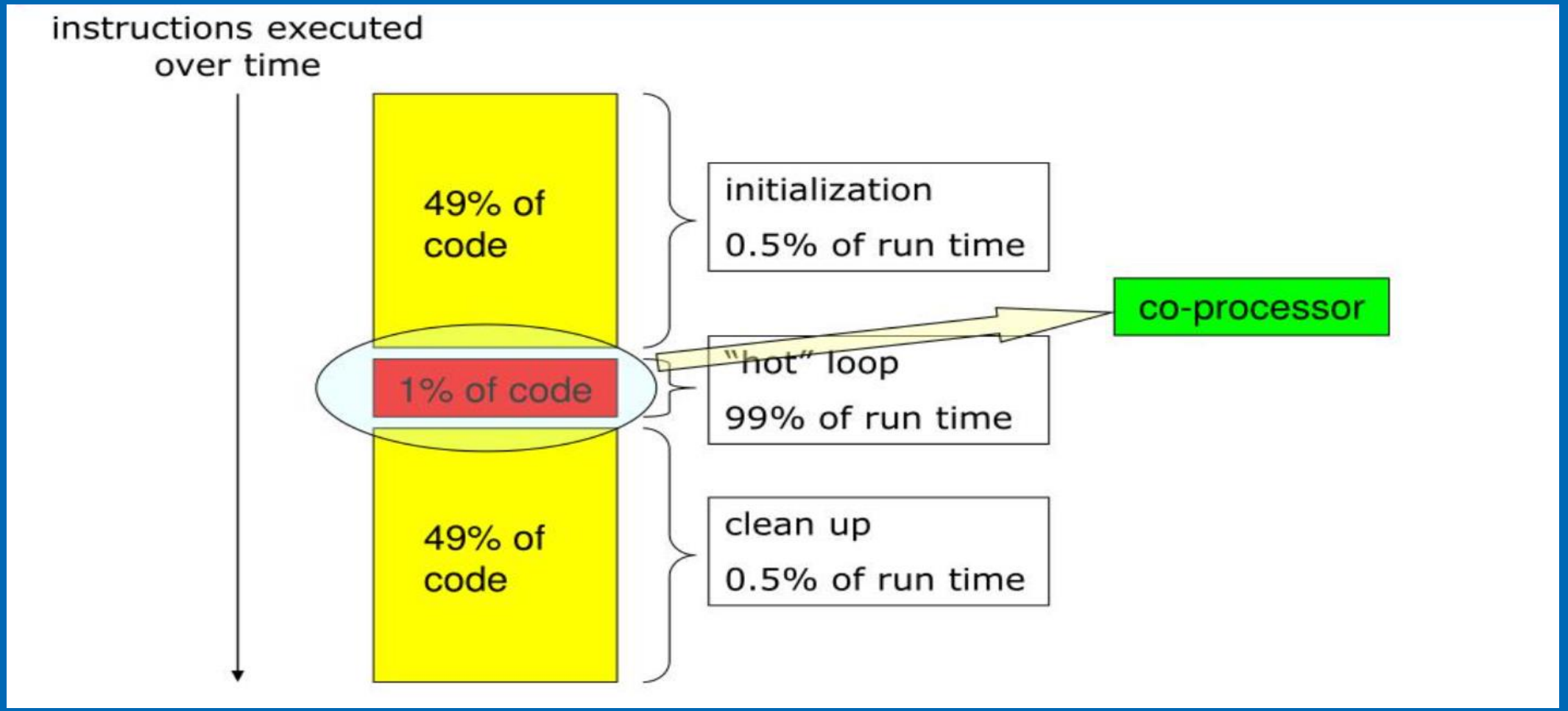
独立性

- 物理上分割，主板连接
- 独立的内存空间
- 不同的计算逻辑以及指令



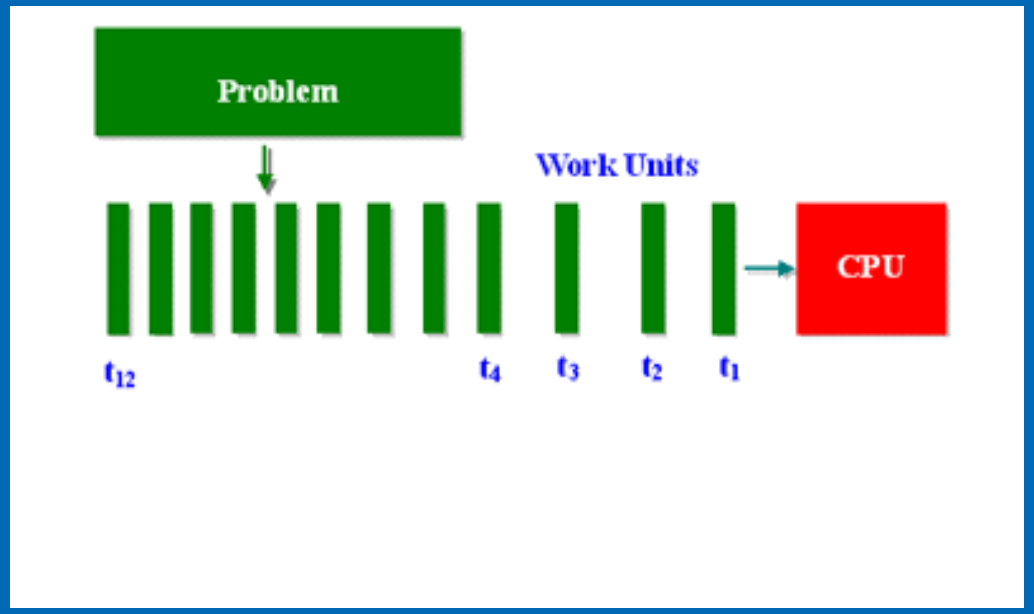
https://bmcsystbiol.biomedcentral.com/articles/10.1186/1752-0509-6-S1-S16/figures/3

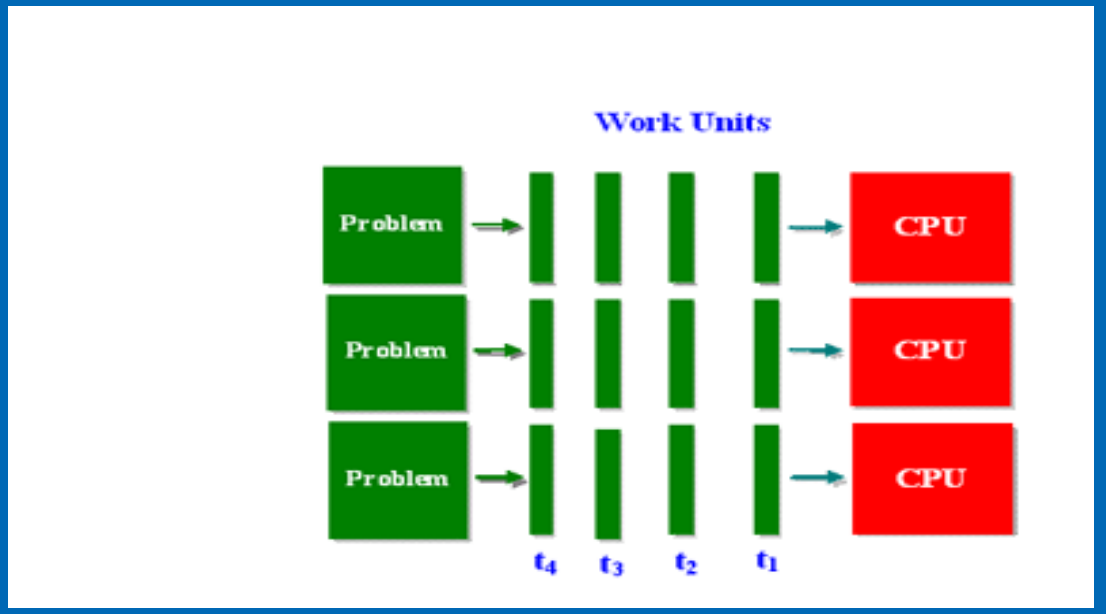
相关性


instructions executed
over time
49% of
code
1% of code
49% of
code
initialization
0.5% of run time
"hot" loop
99% of run time
clean up
0.5% of run time
co-processor

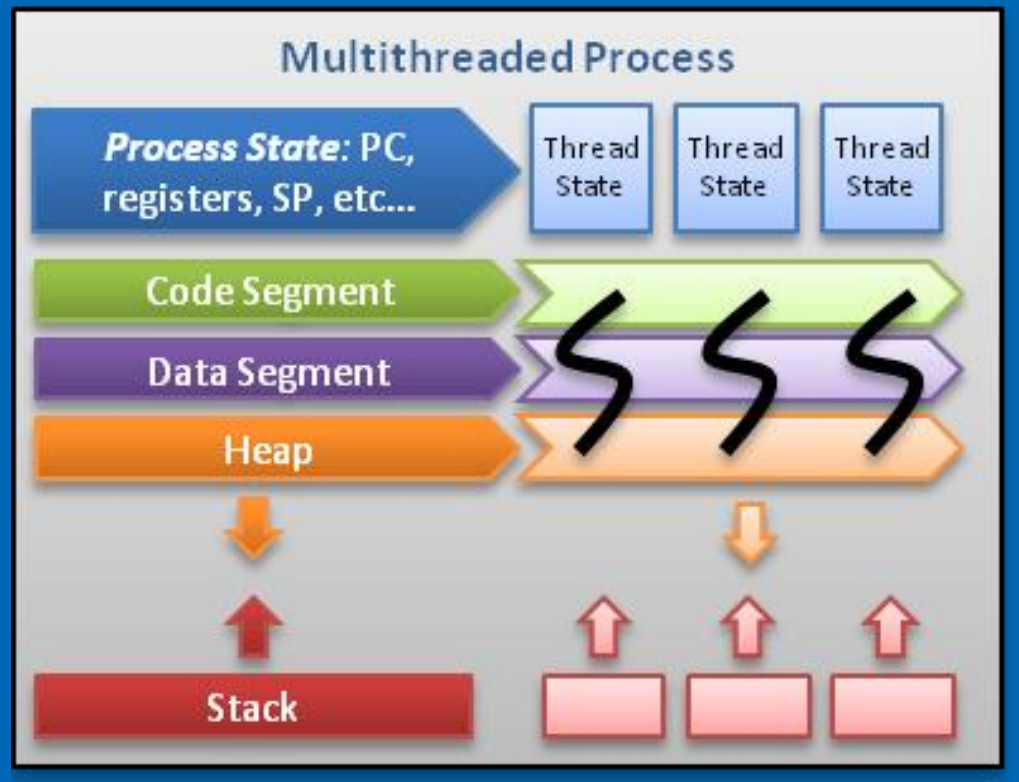
并行性

- 异构计算其实是并行计算的一个扩展，其核心内容仍然是做并行计算。
- 不同之处只是在不同的指令集和硬件架构下进行并行计算。
- 并行计算的核心是如何将任务分配到不同的计算核心。





并行编程框架


Multithreaded Process
Process State: PC, registers, SP, etc...
Thread State
Thread State
Thread State
Code Segment
Data Segment
Heap
Stack

- 多核编程： pthread，openMP
- 多节点编程： MPI
- 异构编程： CUDA，OpenCL，DPC++

多核编程

POSIX线程，简称Pthread，它定义了创建和操纵线程的一阵套API。

线程在相同的进程中共享：

- 进程中的全局变量
- 文件描述符
- 信号
- 工作目录
- 用户和组编号

线程特有的属性：

- 线程编号
- 寄存器，栈指针
- 局部变量，返回地址

Pthread *Hello World*

1. 头文件<pthread.h>
2. 设置线程数目以及独立空间
3. 定义线程函数，需void类型
4. 创建线程，pthread_create
5. 每个线程独立执行函数f
6. 等待每个线程完成，pthread_join

```
#include <pthread.h>
#include <stdio.h>

#define THREADS 4

void *f(void* id) {
   int tid = * (int*) id;
   printf("Thread %d, %ld checking in!\n", tid, pthread_self());
   return NULL;
}

int main() {

   pthread_t threads[THREADS];
   int tid[THREADS];

   for (int i = 0; i < THREADS; i++) {
      tid[i] = i;
      pthread_create(&threads[i], NULL, f, &tid[i]);
   }

    for (int i = 0; i < THREADS; i++) {
       pthread_join(threads[i], NULL);
   }

   printf("All threads finished!\n");

   return 0;
}
```

提示：

使用Pthread，开发者有很大的控制权限，非常灵活，但是也非常容易出错。

```
Thread 3, 139659471644224 checking in!
Thread 3, 139659463251520 checking in!
Thread 3, 139659446466112 checking in!
Thread 3, 139659454858816 checking in!
All threads finished!
```

```c
#include <pthread.h>
#include <stdio.h>

#define THREADS 4

void *f(void* id) {
   int tid = * (int*) id;
   printf("Thread %d, %ld checking in!\n", tid, pthread_self());
   return NULL;
}

int main() {

   pthread_t threads[THREADS];
   int tid;

   for (int i = 0; i < THREADS; i++) {
      tid = i;
      pthread_create(&threads[i], NULL, f, &tid);
   }

    for (int i = 0; i < THREADS; i++) {
       pthread_join(threads[i], NULL);
   }

   printf("All threads finished!\n");

   return 0;
}
```

Pthread：向量加

```
#define THREADS 4
#define N 100

// global data, every thread can see it
int A[N];
int B[N];
int C[N];

void *vecAdd(void* id) {
   int tid = * (int*) id;
   int tnum = N / THREADS;
   printf("Thread %d, %ld checking in!\n", tid, pthread_self());
   for(int i = tid * tnum; i < (tid +1)*tnum; i++) {
         C[i] = A[i] + B[i];
   }
   return NULL;
}
```

```
int main() {

   pthread_t threads[THREADS];
   int index[THREADS];

   int workload_per_thread = N / THREADS;
   for(int i = 0; i < N; i++) {
      A[i] = 1;
      B[i] = 2;
      C[i] = 0;
   }

   for (int i = 0; i < THREADS; i++) {
      index[i] = i;
      pthread_create(&threads[i], NULL, vecAdd, &index[i]);
   }

   for (int i = 0; i < THREADS; i++) {
       pthread_join(threads[i], NULL);
   }

}
```

OpenMP

- 一种基于编译指导的并行化方法
- 基于共享内存的轻量级并行化方法
- 无需线程的创建，同步，销毁等细粒度控制
- 开发者需要指定并行区域
- 编译器自动生成并行代码

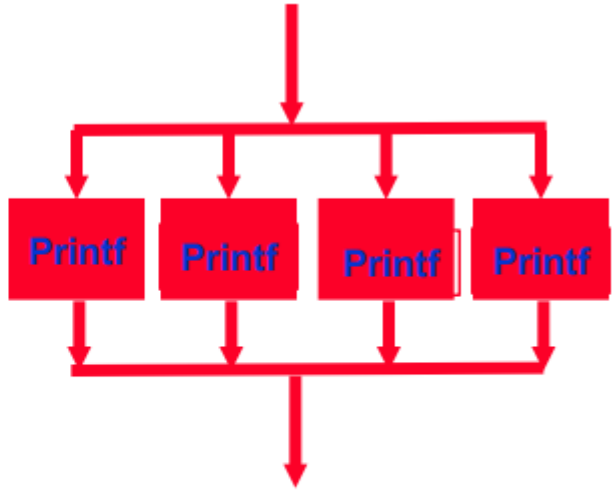
OpenMP Hello World

- #pragma 为线程创建起始标志，线程自动回收当退出当前区域
- parallel 编译指导语句，表明下面区域可以进行并行化

create

join

```
int main() {

  omp_set_num_threads(4);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```



OpenMP：向量加

- 语法结构

#pragma omp parallel [clause [clause] ...]

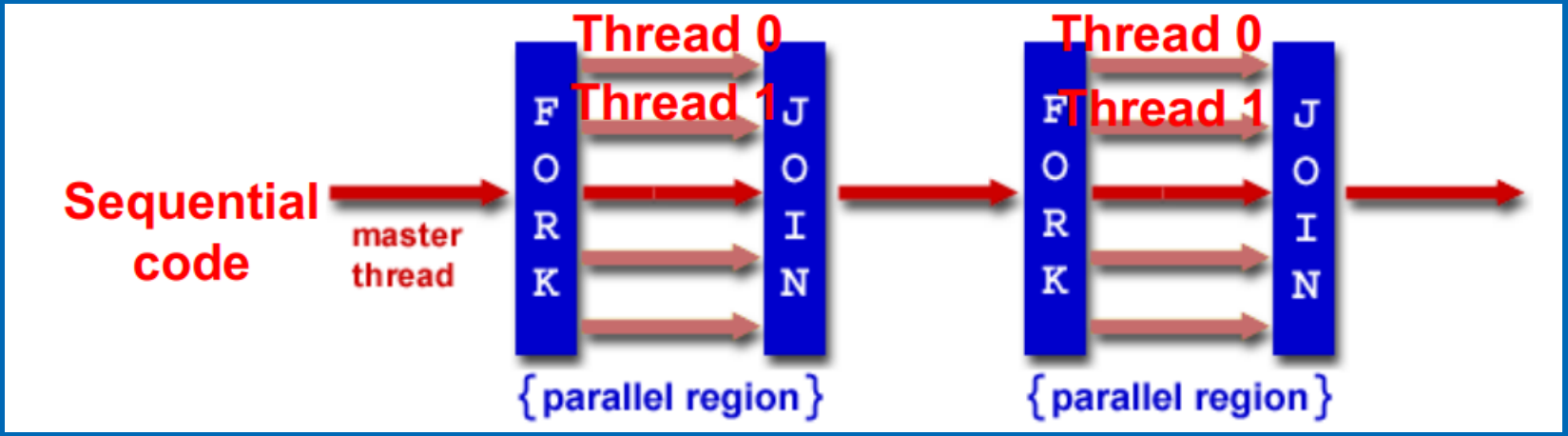
structured-block

- Clause包括：private，shared，…

```
#include <omp.h>
#include <stdio.h>

#define N 100

int main() {
   int A[N];
   int B[N];
   int C[N];

   for(int i = 0; i < N; i++) {
      A[i] = 1;
      B[i] = 2;
      C[i] = 0;
   }

#pragma omp parallel for
   for(int i = 0; i < N; i++) {
         C[i] = A[i] + B[i];
   }

   for(int i =0; i < N; i++) {
      if( C[i] != 3 ) printf("\nError in %d, %d", i, C[i]);
   }

   printf("All threads finished!\n");

   return 0;
}
```

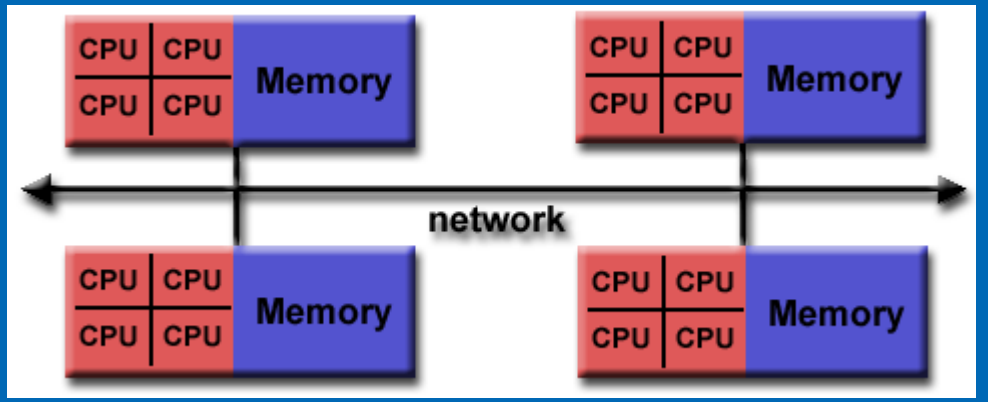
Fork-Join模型

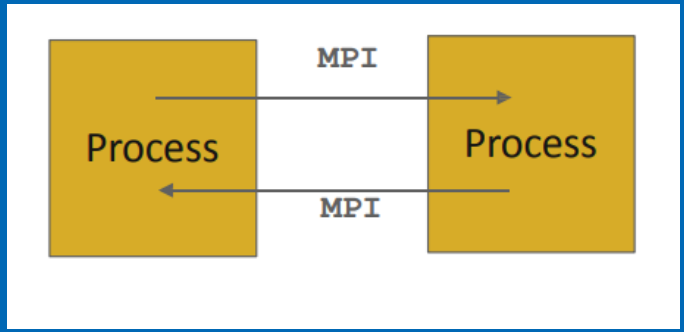
- OpenMP的编程模型是一种基于创建（fork），合并（join）的方式
- 仅在需要的地方进行并行化，线程创建和销毁的成本低

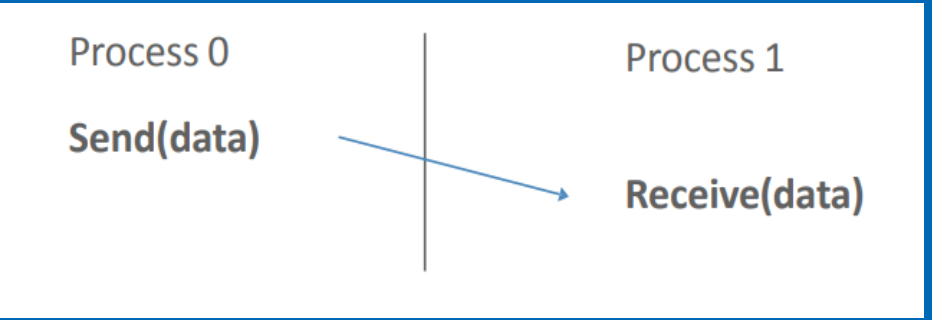


MPI: 消息传递模型

- 基于分布式内存的并行计算模式
- 进程之间进行显式的通信
- 消息发送和接收







如何运行一个MPI程序?

普通程序编译和执行:

- gcc test.c -o test
- ./test

MPI程序编译和执行:

- mpicc test.c -o test
- mpiexec –np 4 ./test

- 通过mpiexec 会创建新的进程
- Np指定number of processor

MPI Hello World

1. 头文件<mpi.h>
2. MPI初始化和结束
3. 通信域，MPI_COMM_WORLD
4. 得到当前通信域整体进程数，size
5. 得到当前进程对应的序号，rank

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Basic requirements for an MPI program

MPI send/receive

MPI_SEND(buf, count, datatype, dest, tag, comm)

MPI_RECV(buf, count, datatype, source, tag, comm, status)

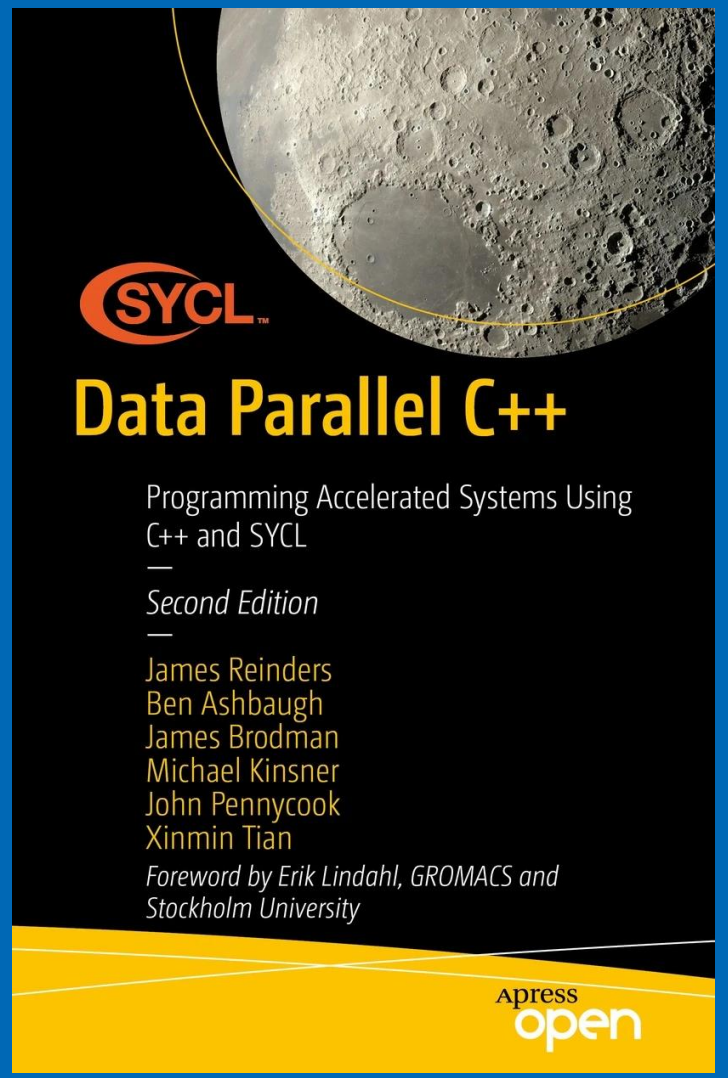
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

- MPI的工作模式就像发短信，一方发一发接收。
- 发送和接收的数据用（buf，count，datatype）描述
- tag是用户定义的类型
- Comm是定义的通讯域
- 对于SEND/RECV是阻塞型API，会一直等待到彼此之间数据传输完成

MPI：向量加

```
int main (int argc, char *argv[])
{
        // elements of arrays a and b will be added
        // and placed in array c
        int * a;
        int * b;
        int * c;

        int total_proc;  // total nuber of processes
        int rank;        // rank of each process
        int n_per_proc; // elements per process
        int n = ARRAY_SIZE;   // number of array elements
        int i;       // loop index

        MPI_Status status;   // not used in this arguably poor example
                             // that is devoid of error checking.

        // 1. Initialization of MPI environment
        MPI_Init (&argc, &argv);
        MPI_Comm_size (MPI_COMM_WORLD, &total_proc);
        // 2. Now you know the total number of processes running in parallel
        MPI_Comm_rank (MPI_COMM_WORLD,&rank);
        // 3. Now you know the rank of the current process

        // Smaller arrays that will be held on each separate process
        int * ap;
        int * bp;
        int * cp;

        // 4. We choose process rank 0 to be the root, or master,
        // which will be used to  initialize the full arrays.
        if (rank == MASTER)  {
                a = (int *) malloc(sizeof(int)*n);
                b = (int *) malloc(sizeof(int)*n);
                c = (int *) malloc(sizeof(int)*n);

                // initialize arrays a and b with consecutive integer values
                // as a simple example
                for(i=0;i<n;i++)
                        a[i] = i;
                for(i=0;i<n;i++)
                        b[i] = i;
        }

        // All processes take part in the calculations concurrently
        // determine how many elements each process will work on
        n_per_proc = n/total_proc;
        /////// NOTE:
        // In this simple version, the number of processes needs to
        // divide evenly into the number of elements in the array
        ///////////

        // 5. Initialize my smaller subsections of the larger array
        ap = (int *) malloc(sizeof(int)*n_per_proc);
        bp = (int *) malloc(sizeof(int)*n_per_proc);
        cp = (int *) malloc(sizeof(int)*n_per_proc);

        // 6.
        //scattering array a from MASTER node out to the other nodes
        MPI_Scatter(a, n_per_proc, MPI_INT, ap, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);
        //scattering array b from MASTER node out to the other node
        MPI_Scatter(b, n_per_proc, MPI_INT, bp, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);

        // 7. Compute the addition of elements in my subsection of the array
        for(i=0;i<n_per_proc;i++)
                cp[i] = ap[i]+bp[i];

        // 8. MASTER node gathering array c from the workers
        MPI_Gather(cp, n_per_proc, MPI_INT, c, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);

////////////////////// all concurrent processes are finished once they all communicate
////////////////////// data back to the master via the gather function.

        // Master process gets to here only when it has been able to gather from all processes
        if (rank == MASTER)  {
                // sanity check the result  (a test we would eventually leave out)
                int good = 1;
                for(i=0;i<n;i++) {
                        //printf ("%d ", c[i]);
                        if (c[i] != a[i] + b[i]) {
                                printf("problem at index %lld\n", i);
                                good = 0;
                                break;
                        }
                }
                if (good) {
                        printf ("Values correct!\n");
                }

        }

        // clean up memory
        if (rank == MASTER)  {
                free(a);  free(b); free(c);
        }
        free(ap);  free(bp); free(cp);

        // 9. Terminate MPI Environment and Processes
        MPI_Finalize();
```

Data Parallel C++



https://github.com/Apress/data-parallel-CPP

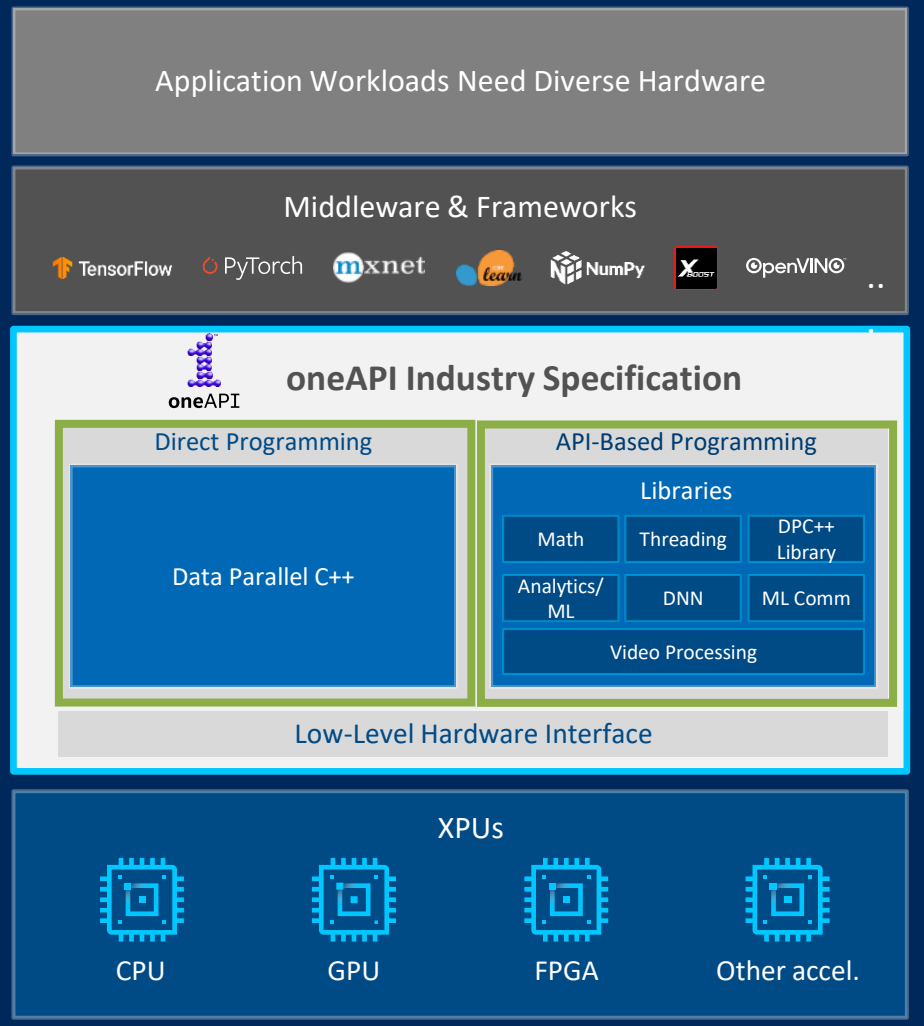
编程模型

- Low-Level Hardware Interface
- Programming Language
- High Performance Library

A cross-architecture language based on C++ and SYCL standards

Powerful libraries designed for acceleration of domain-specific functions

Low-level hardware abstraction layer



oneAPI

The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models

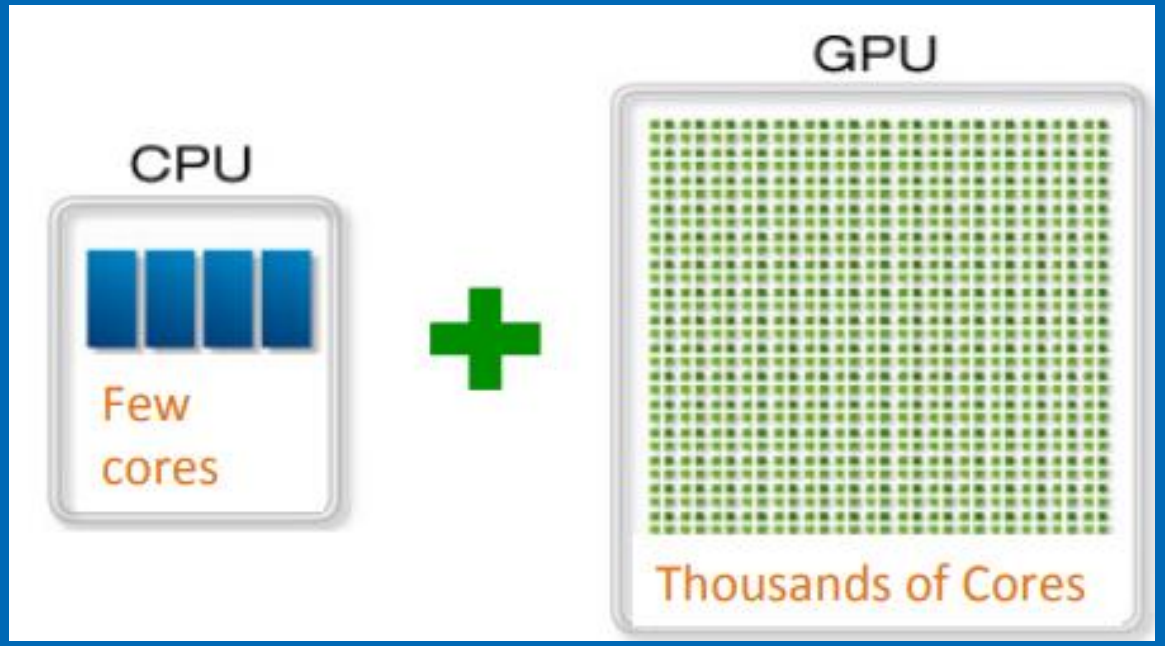
通用处理器： CPU + GPU来进行加速

CPU

- General Purpose Tasks
- Most Common App
- Compute Intensive

GPU

- Specialized tasks
- Most visual app
- Data processing in parallel



本课程重点：

- 教会同学们如何使用DPC++编写程序
- 理解串行程序和并行程序设计的区别
- 实现自己的并行算法 == 能完成作业☺

本课程将略过：

- 非关键路径的功能
- 非必要的语言特性和细节
- 各种用法的变化

Data Parallel C++ 是什么?

基于C++语言的扩展，它提供了：

- 访问硬件设备的抽象
- 数据访问的方法
- 表达并行性的方法

设备

- **设备,** 表示 OneAPI 系统中的各种硬件

  设备类是预定义的设备选择和查询的方法

  包含用于查询设备信息的成员函数，

  支持创建不同硬件， CPU/GPU/FPGA/…

- **device_selector** 类支持运行时选择特定设备

  default_selector、 cpu_selector、 gpu_selector.....

Code Example

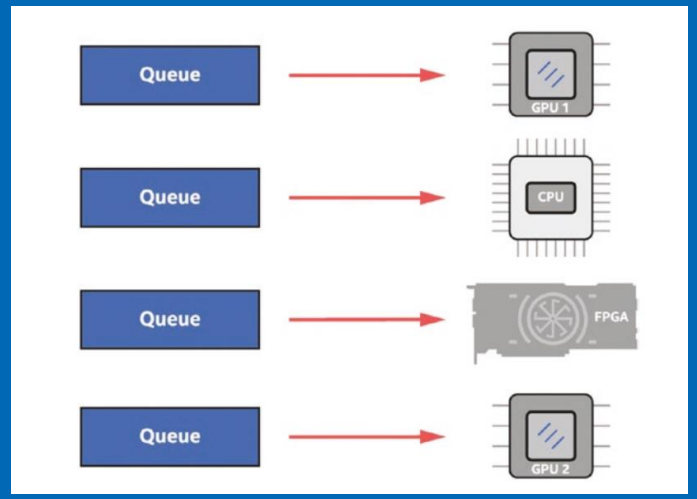
```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;
int main() {
  queue my_gpu_queue( gpu_selector{} );
}
```

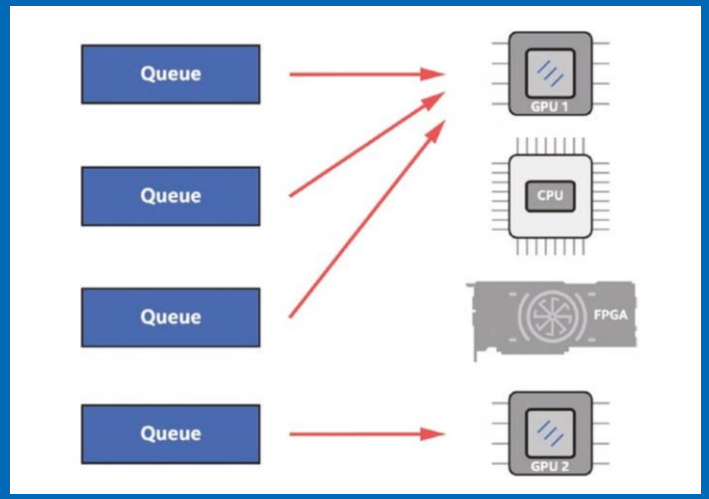
```
patric@gpu:~$ dpcpp gpu_selector.cpp
patric@gpu:~$ ./a.out
Selected GPU device: Intel(R) Iris(R) Xe MAX Graphics [0x4905]
```

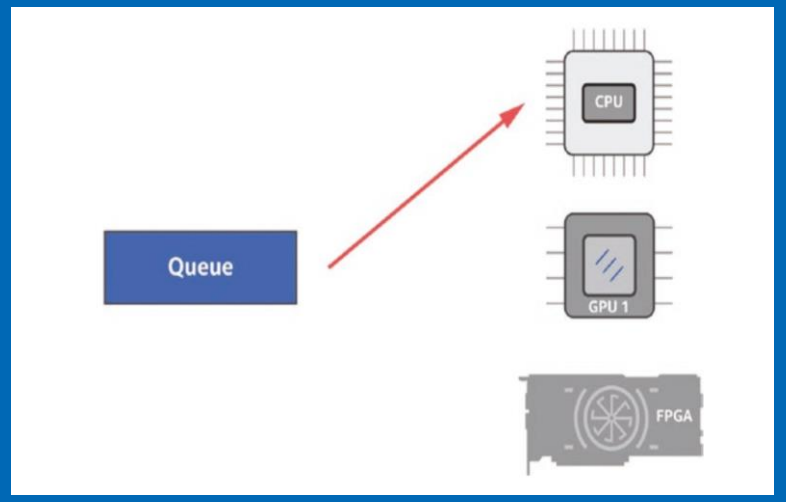
https://github.com/pengzhao-intel/oneAPI_course/blob/main/code/gpu_selector.cpp

队列

- 队列是一种将工作提交到设备的机制
- 主程序将任务推入队列，异构设备从队列中获得执行任务
- 一个队列映射到一个设备，多个队列可以映射到同一设备。





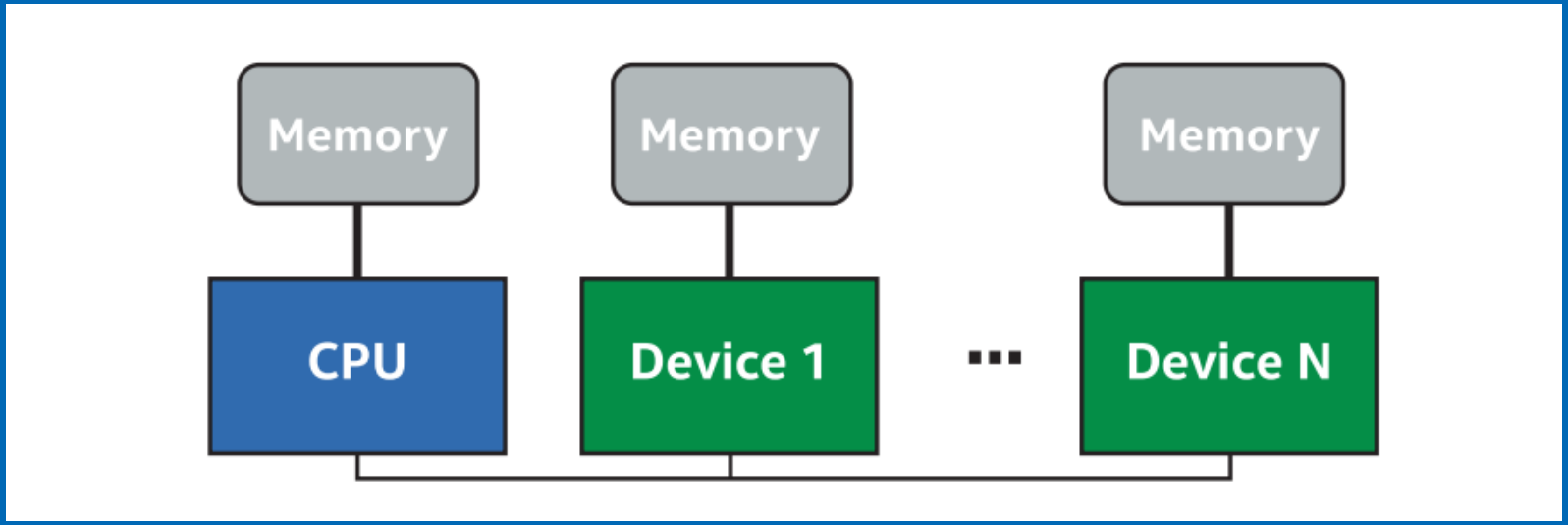


Code Example

```
int main() {
  queue my_gpu_queue( gpu_selector{} );
  std::cout << "Selected GPU device: " <<
    my_gpu_queue.get_device().get_info<info::device::name>() ";
  return 0;
}
```

数据移动

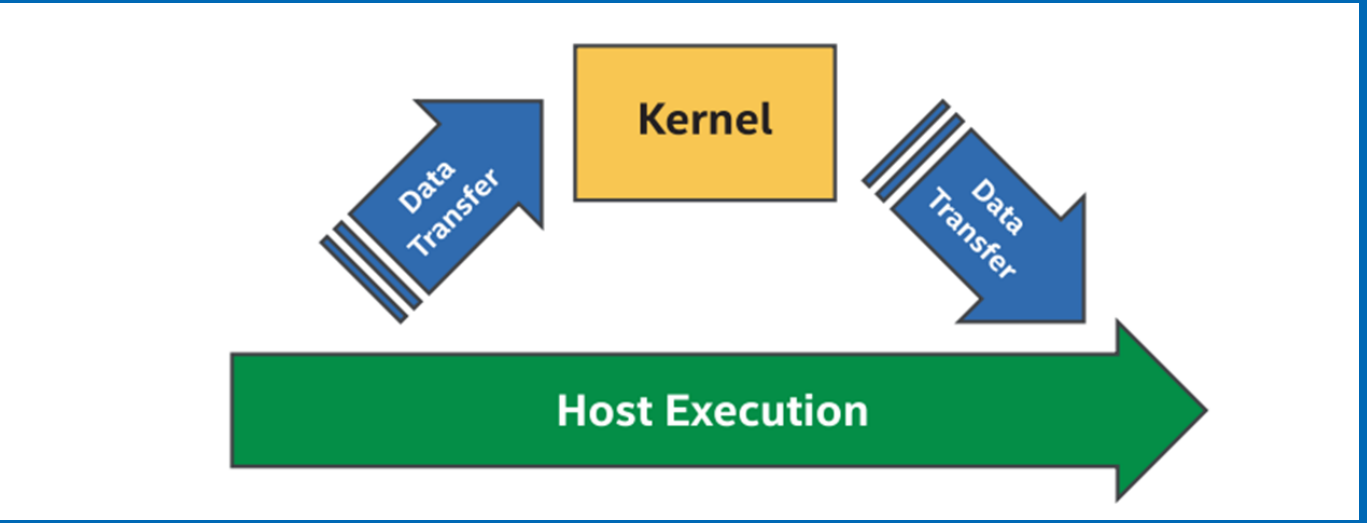
- CPU和GPU具有独立的存储空间



- 程序员需要考虑如何在不同设备之间移动数据

- 显式的数据移动，CPU->GPU，计算， GPU -> CPU

- 隐式的数据移动， 数据在被访问的时候，系统自动进行数据迁移



• 显式内存申请

Traditional C/C++

void* malloc (size_t size);

Allocate memory block

Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.

Data Parallel C++

void* malloc_host(size_t size, const sycl::queue& q);

void* malloc_device(size_t size, const sycl::queue& q);

void* malloc_shared(size_t size, const sycl::queue& q);

• 显式内存拷贝

Traditional C/C++

void * memcpy (void * destination, const void * source, size_t num);

Copy block of memory

Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.

Data Parallel C++

void * queue.memcpy (void * destination, const void * source, size_t num);

• 内存释放

Traditional C/C++

void free (void* ptr);

Deallocate memory block

A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated,

making it available again for further allocations.

Data Parallel C++

void free(void* ptr, sycl::queue& q);

Free memory allocated by sycl::malloc_device, sycl::malloc_host, or sycl::malloc_shared.

代码实例

```
constexpr int N = 10;

  int *host_mem   = malloc_host<int>(N, my_gpu_queue);
  int *device_mem = malloc_device<int>(N, my_gpu_queue);

  // Init CPU data
  for(int i = 0; i < N; i++) {    host_mem[i] = i;    }

  // Copy from host(CPU) to device(GPU)
  my_gpu_queue.memcpy(device_mem, host_mem, N * sizeof(int)).wait();

  // do some works on GPU

  // Copy back from GPU to CPU
  my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();

  printf("\nData Result\n");
  for(int i = 0; i < N; i++) {   printf("%d, ", host_mem[i]);  }
```

https://github.com/pengzhao-intel/oneAPI_course/blob/main/code/data_movement_ex.cpp

```
constexpr int N = 10;

  int *host_mem   = malloc_host<int>(N, my_gpu_queue);
  int *device_mem = malloc_device<int>(N, my_gpu_queue);
```

代码

编译：

```
patric@gpu:~/course$ icpx data_movement_ex.cpp -o data_move
```

运行输出：

```
patric@gpu:~/course$ ./data_move
Selected GPU device: Intel(R) Iris(R) Xe MAX Graphics [0x4905]

Data Result
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Task Done!
```

https://github.com/pengzhao-intel/oneAPI_course/blob/main/code/data_movement_ex.cpp

内核

- 什么地方需要并行化?

计算量最大，最耗时的地方，通常是循环内的部分

串行的执行方式

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

并行的执行方式

```
launch N kernel instances {
    int id = get_instance_id();
    c[id] = a[id] + b[id];
}
```

parallel_for

- 并行化 for-loop是并行计算的核心

  在该循环中，每个迭代应该是完全独立的，并且不分顺序。
- 并行内核使用 parallel_for 函数表示

串行的执行方式

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

使用 **parallel_for** 卸载到加速器

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    A[i] =  B[i] + C[i];
});
```

基础并行内核

基本并行内核的功能通过 range、 id 和 item 类提供。

- range 用于描述任务空间的大小
- item 代表内核函数的单个实例，向执行范围的查询属性公开其他函数
- 利用item的信息来将每个线程对应到整体的任务空间

```
h.parallel_for(range<1>(1024), [=](item<1> item){
                auto idx = item.get_id();
                auto R = item.get_range();
                // CODE THAT RUNS ON DEVICE
});
```

代码实例

```
// Copy from host(CPU) to device(GPU)
my_gpu_queue.memcpy(device_mem, host_mem, N * sizeof(int)).wait();

// do some works on GPU
// submit the content to the queue for execution
my_gpu_queue.submit([&](handler& h) {

    // Parallel Computation
    h.parallel_for(range{N}, [=](id<1> item) {
        device_mem[item] *= 2;
    });

 }); // end submit

// wait the computation done
my_gpu_queue.wait();

// Copy back from GPU to CPU
my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();
```

https://github.com/pengzhao-intel/oneAPI_course/blob/main/code/basic_parafor.cpp

代码实例

```
// Copy from host(CPU) to device(GPU)
my_gpu_queue.memcpy(device_mem, host_mem, N * sizeof(int)).wait();

// do some works on GPU
// submit the content to the queue for execution
my_gpu_queue.submit([&](handler& h) {
```

运行输出：

```
patric@gpu:~/course$ ./basic_parafor
Selected GPU device: Intel(R) Iris(R) Xe MAX Graphics [0x4905]

Data Result
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
Task Done!
```

```
my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();
```

https://github.com/pengzhao-intel/oneAPI_course/blob/main/code/basic_parafor.cpp

性能评价

https://www.constructioninsure.co.uk/high-risk-trade-insurance-most-beneficial/

Q: 我的程序快了吗?

如何衡量我们的程序是否运行的更快，更有效了?

- 时间：wall time
- 带宽：GBytes/sec
- 计算：Gflops

时间

CPU

Start timer

Code

End timer

duration = end – start

```
float duration_cpu = 0.0;
std::chrono::high_resolution_clock::time_point s, e;
s = std::chrono::high_resolution_clock::now();
// CPU code here
e = std::chrono::high_resolution_clock::now();
duration_cpu =
std::chrono::duration<float, std::milli>(e - s).count();
```

代码实例

```
// CPU computation
 printf("\n Start CPU Computation, Number of Elems = %d \n", N);

 float duration_cpu = 0.0;
 std::chrono::high_resolution_clock::time_point s, e;
 s = std::chrono::high_resolution_clock::now();

 // CPU code here
 for(int64_t i = 0; i < N; i++) {
    cpu_mem[i] = host_mem[i] * 2;
 }

 e = std::chrono::high_resolution_clock::now();
 duration_cpu =  std::chrono::duration<float, std::milli>(e - s).count();
 printf("\n End CPU Computation, Time = %lf \n", duration_cpu);
```

```
// CPU computation
 printf("\n Start CPU Computation, Number of Elems = %d \n", N);
```

代码

编译：

```
patric@gpu:~/course$ dpcpp timer.cpp -o time
```

运行输出：

```
patric@gpu:~/course$ ./time
Selected GPU device: Intel(R) Iris(R) Xe MAX Graphics [0x4905]

 Start CPU Computation, Number of Elems = 10000000
 End CPU Computation, Time = 5.333

Task Done!
```

如何测量GPU时间?

GPU

Start timer

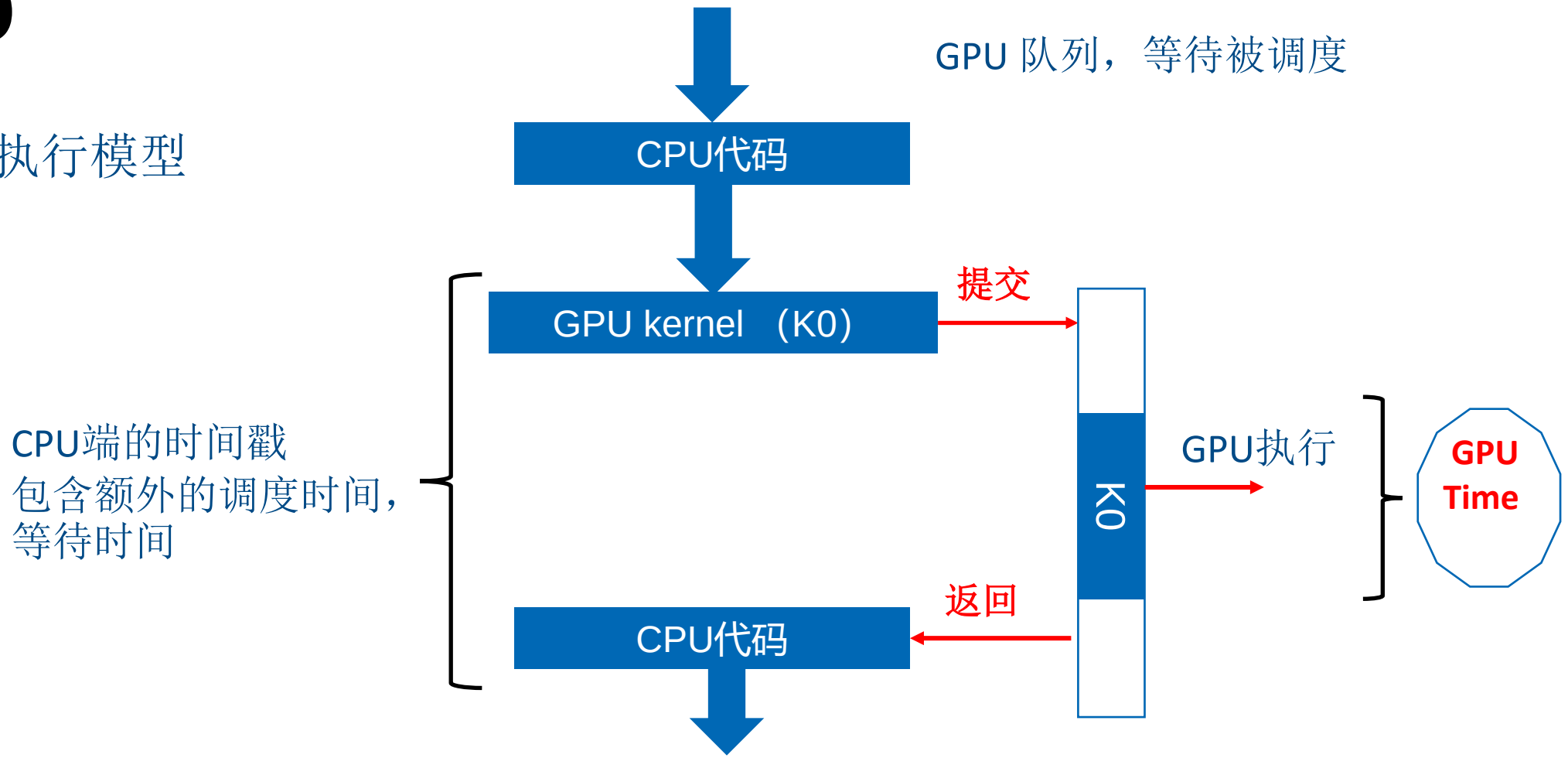
GPU Code

End timer

duration = end – start

NO

异构执行模型


GPU 队列，等待被调度
CPU代码
GPU kernel （K0）
提交
CPU端的时间戳
包含额外的调度时间，
等待时间
K0
GPU执行
GPU
Time
返回
CPU代码

GPU: add profiling in Queue

```
auto propList = cl::sycl::property_list {cl::sycl::property::queue::enable_profiling()};

queue my_gpu_queue(gpu_selector{}, propList);

// submit the content to the queue for execution
 auto event = my_gpu_queue.submit([&](handler& h) {
      // Parallel Computation
       ….
 });

 double gpu_time_ns =
  event.get_profiling_info<info::event_profiling::command_end>() -
  event.get_profiling_info<info::event_profiling::command_start>();
```

YES

从CPU的角度来看代码
执行了多长时间

```
// Copy from host(CPU) to device(GPU)
my_gpu_queue.memcpy(device_mem, host_mem, N * sizeof(int)).wait();

// do some works on GPU
// submit the content to the queue for execution
my_gpu_queue.submit([&](handler& h) {

    // Parallel Computation
    h.parallel_for(range{N}, [=](id<1> item) {
        device_mem[item] *= 2;
    });

}); // end submit

// wait the computation done
my_gpu_queue.wait();

// Copy back from GPU to CPU
my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();
```

A B C

https://github.com/pengzhao-intel/oneAPI_course/blob/main/code/timer.cpp

Compare Results

```
Selected GPU device: Intel(R) Iris(R) Xe MAX Graphics [0x4905]

Start CPU Computation, Number of Elems = 10000000

CPU Computation,        Time =    5.474415

GPU Computation, GPU Time A = 1.083385

GPU Computation, GPU Time B = 1.605373

GPU Computation, GPU Time C = 20.106045
```

Review Code Structure Again!

```
// Copy from host(CPU) to device(GPU)
my_gpu_queue.memcpy(device_mem, host_mem, N * sizeof(int)).wait();

// do some works on CPU
for(…. )   {  ….  };

// do some works on GPU
    my_gpu_queue.submit([&](handler& h)
      parallel_for( ) { ….. }
 }); // end submit
my_gpu_queue.wait();

// Copy back from GPU to CPU
my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();
```

CPU time1

GPU time1

异构计算与异步计算

```
// Copy from host(CPU) to device(GPU)
  my_gpu_queue.memcpy(device_mem, host_mem, N * sizeof(int)).wait();

  // do some works on CPU
  for(…. )   {  ….  };

  // do some works on GPU
  my_gpu_queue.submit([&](handler& h) {
        parallel_for( ) { ….. }
   }); // end submit
  my_gpu_queue.wait();

  // Copy back from GPU to CPU
  my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();
```

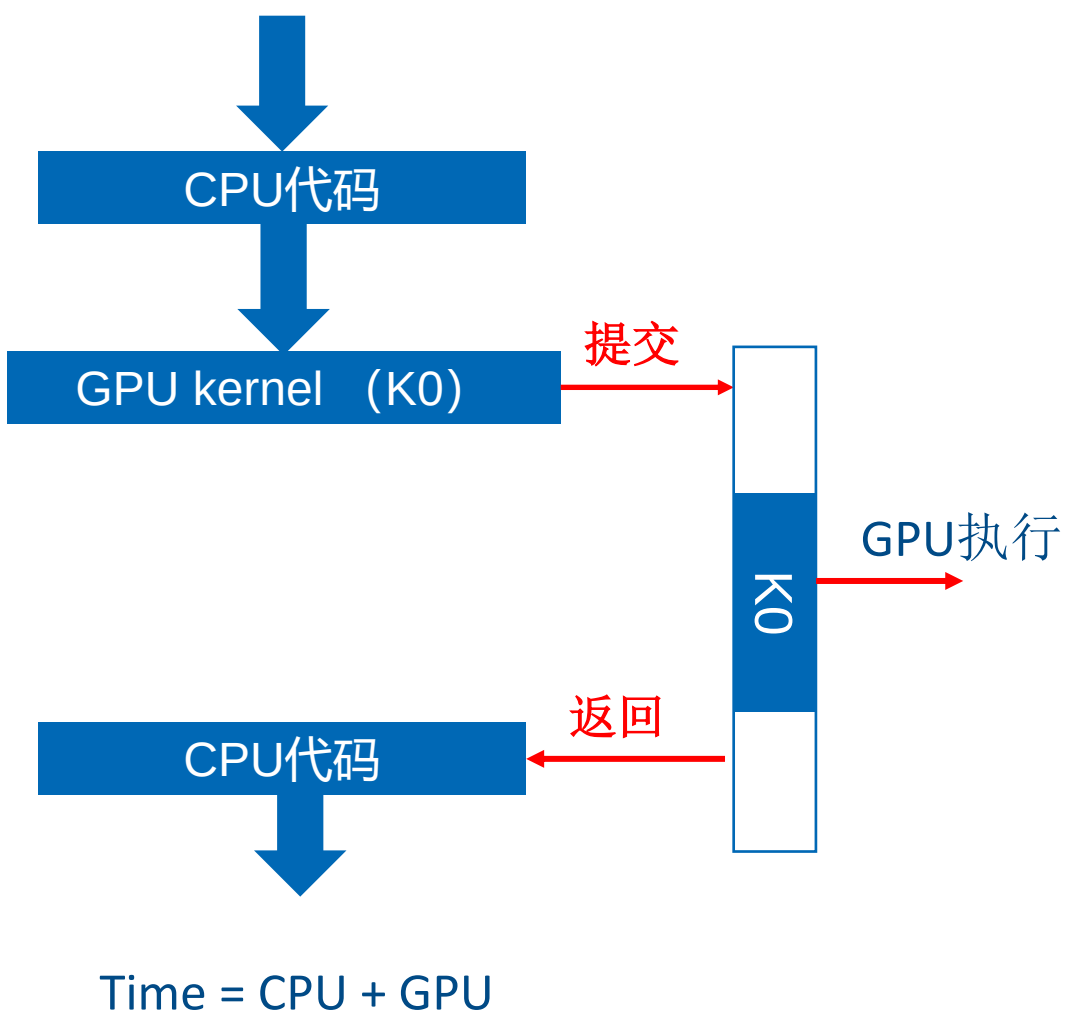
异构计算与异步计算

```
// Copy from host(CPU) to device(GPU)
```

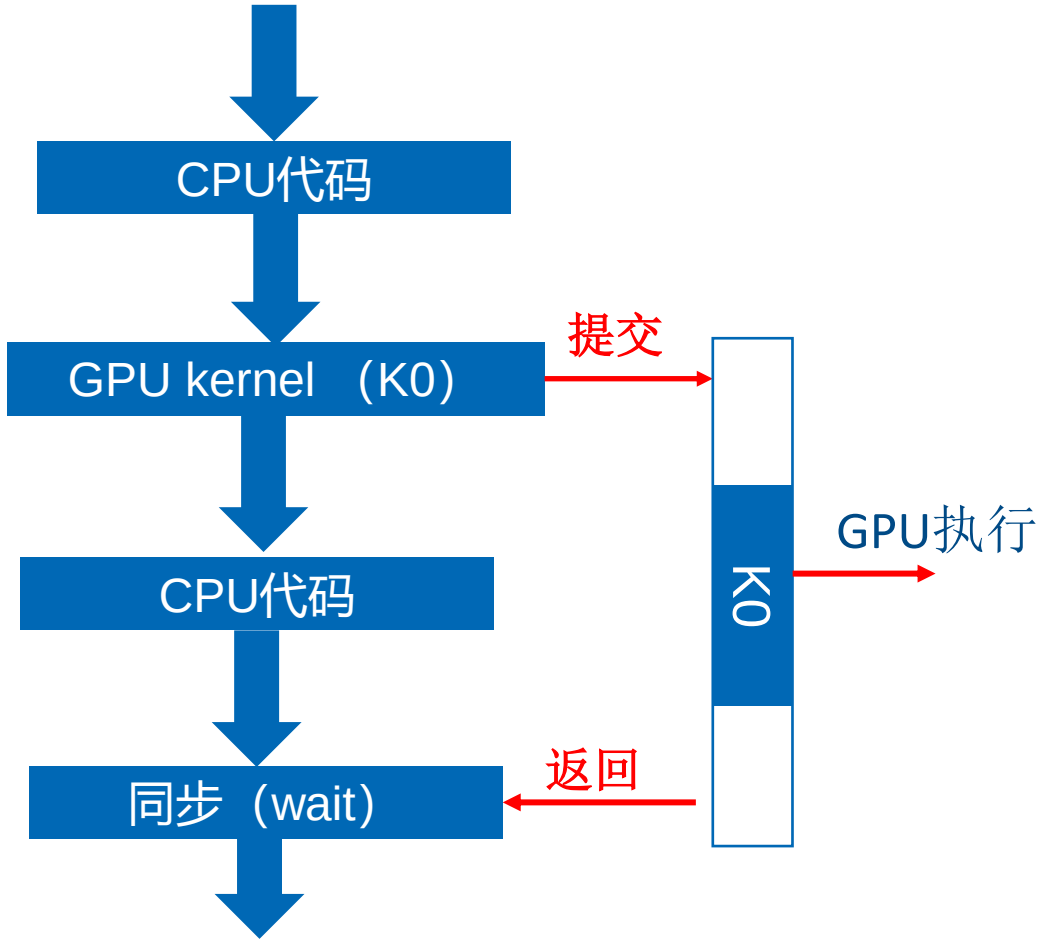
```
Selected GPU device: Intel(R) Iris(R) Xe MAX Graphics [0x4905]

Start CPU Computation, Number of Elems = 10000000

CPU Computation,   Time = 5.699950

GPU Computation,   Time = 1.552332

Total Computation, TIme = 7.252606
```

```
// Copy back from GPU to CPU
my_gpu_queue.memcpy(host_mem, device_mem, N * sizeof(int)).wait();
```

异构计算 + 异步计算



Time = CPU + GPU



Time = max(CPU, GPU)

Async Results

Selected GPU device: NVIDIA A100-PCIE-40GB

Start CPU Computation, Number of Elems = 10000000

CPU Computation, Time = 3.116137

GPU Computation, Time = 3.141288

Total Computation, TIme = 3.141403

Task Done!

小结

- 理解异构计算的核心思想
- 能够实现简单的并行内核
- 具有系统和内核层性能分析的能力


intel