CHAPTER

Introduction to CUDA 3

CHAPTER CONTENTS

INTRODUCTION

To a CUDA™ programmer, the computing system consists of a *host*, which is a traditional central processing unit (CPU), such as an Intel® architecture microprocessor in personal computers today, and one or more *devices*, which are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Because data parallelism plays such an important role in CUDA, we will first discuss the concept of data parallelism before introducing the basic features of CUDA.

3.1 DATA PARALLELISM

Many software applications that process a large amount of data and thus incur long execution times on today's computers are designed to model real-world, physical phenomena. Images and video frames are snapshots

in a simultaneous manner 以同时的方式

Rigid body physics. 刚体物理

fluid dynamics model nature forces
流体动力学模型自然力

GPU中的SM计算单元.

一个SM单元能提供的最大线程数是确定的.

一个SM单元拥有 Register File 是固定的.

但是活跃线程数不确定.

所以一个线程能用到的寄存器是动态分配的.

CUDA编译器 nvcc 提供.

Q1: 实习公司的工具链流程.

高层编译 → 底层编译 → 汇编算子 → 自研GPU

对比 nvcc 工具链. 是针对特定的 NVIDIA 平台.

当然, 和AMD平台不互通 (叶叫秦始皇)

nvcc会编译 CUDA 程序到 PTX 中间表示.

结合实际运行的硬件架构进行优化.

这种完整的 cuda 工具链就不适合自研 GPU 的情况.

∴ 会在高层使用 MLIR. 下降到 LLVM .

高层部分还不需要考虑硬件平台特定的优化. 这样.

Q1: 底层编译和汇编算子对接了什么.
传递了什么优化信息.

Q2: GPU中的 Instruction Cache 和 Warp Scheduler 关系.

Q3: CUDA 模型的工具链流程

Q4: CUDA的上下游.

Q5: 软件层面线程执行对应到硬件的关系.
<<<grid, block>>>

of a physical world where different parts of a picture capture simultaneous, independent physical events. Rigid body physics and fluid dynamics model natural forces and movements that can be independently evaluated within small time steps. Such independent evaluation is the basis of data parallelism in these applications.

As we mentioned earlier, data parallelism refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner. We illustrate the concept of data parallelism with a matrix–matrix multiplication (matrix multiplication, for brevity) example in Figure 3.1. In this example, each element of the product matrix **P** is generated by performing a dot product between a row of input matrix **M** and a column of input matrix **N**. In Figure 3.1, the highlighted element of matrix **P** is generated by taking the dot product of the highlighted row of matrix **M** and the highlighted column of matrix **N**. Note that the dot product operations for computing different matrix **P** elements can be simultaneously performed. That is, none of these dot products will affect

M x N = P

M的一行与N的一列相乘后相加的结果

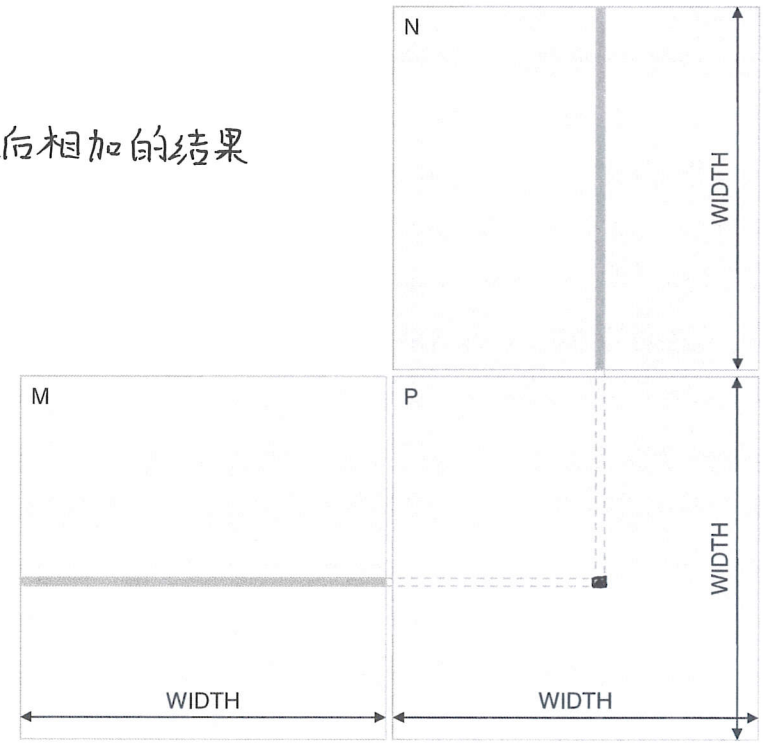


FIGURE 3.1

Data parallelism in matrix multiplication.

encompassing 包含.涵盖.

a unified source code 统一的源代码

CPU ⇒ host code

GPU ⇒ device code

nvcc来把一个CUDA程序中交给CPU和GPU的部分区分开

Q: nvcc编译器与标准C编译器.

nvcc对接了标准C的接口.把被"__global__"等修饰的函数提取出来.翻译为PTX之类的

并且如果这个kernel更适合在CPU上执行的话也可以在CPU上模拟执行.反正就是自由度高.

the results of each other. For large matrices, the number of dot products can be very large; for example, a 1000 × 1000 matrix multiplication has 1,000,000 independent dot products, each involving 1000 multiply and 1000 accumulate arithmetic operations. Therefore, matrix multiplication of large dimensions can have very large amount of data parallelism. By executing many dot products in parallel, a CUDA device can significantly accelerate the execution of the matrix multiplication over a traditional host CPU. The data parallelism in real applications is not always as simple as that in our matrix multiplication example. In a later chapter, we will discuss these more sophisticated forms of data parallelism.

3.2 CUDA PROGRAM STRUCTURE

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. A CUDA program is a unified source code encompassing both host and device code. The NVIDIA® C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. The device code is typically further compiled by the nvcc and executed on a GPU device. In situations where no device is available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU using the emulation features in CUDA software development kit (SDK) or the MCUDA tool [Stratton 2008].

The kernel functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism. In the matrix multiplication example, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix **P**. In this example, the number of threads used by the kernel is a function of the matrix dimension. For a 1000 × 1000 matrix multiplication, the kernel that uses one thread to compute one **P** element would generate 1,000,000 threads when it is invoked. It is worth noting that CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically require thousands of clock cycles to generate and schedule.

abundant 丰富，大量

be collectively 共同的，集体的

concretely 具体，明确

brevity 简洁

matrices 矩阵

Grid: All the threads.

Kernel 与 Grid 是同一个层级上的概念.

每次只能有一个 kernel 在一个 GPU 上执行. 所以串行的

但是 Grid 只是逻辑上对 Block 的组织

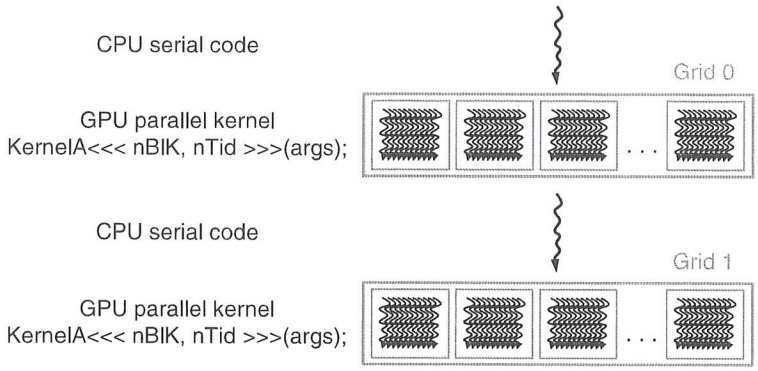


FIGURE 3.2

Execution of a CUDA program.

The execution of a typical CUDA program is illustrated in Figure 3.2. The execution starts with host (CPU) execution. When a kernel function is invoked, or *launched*, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a *grid*. Figure 3.2 shows the execution of two grids of threads. We will discuss how these grids are organized soon. When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.

3.3 A MATRIX–MATRIX MULTIPLICATION EXAMPLE

At this point, it is worthwhile to introduce a code example that concretely illustrates the CUDA program structure. Figure 3.3 shows a simple main function skeleton for the matrix multiplication example. For simplicity, we assume that the matrices are square in shape, and the dimension of each matrix is specified by the parameter `Width`.

The main program first allocates the **M**, **N**, and **P** matrices in the host memory and then performs I/O to read in **M** and **N** in Part 1. These are ANSI C operations, so we are not showing the actual code for the sake of brevity. The detailed code of the main function and some user-defined ANSI C functions is shown in Appendix A. Similarly, after completing the matrix multiplication, Part 3 of the main function performs I/O to write the product matrix **P** and to free all the allocated matrices. The details of Part 3 are also shown in Appendix A. Part 2 is the main focus of our

consecutive 连续连贯.

```
int main(void) {
1.  // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

2.  // M * N on the device
    MatrixMultiplication(M, N, P, Width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

FIGURE 3.3

A simple main function for the matrix multiplication example.

example. It calls a function, `MatrixMultiplication()`, to perform matrix multiplication on a device.

Before we explain how to use a CUDA device to execute the matrix multiplication function, it is helpful to first review how a conventional CPU-only matrix multiplication function works. A simple version of a CPU-only matrix multiplication function is shown in Figure 3.4. The `MatrixMultiplication()` function implements a straightforward algorithm that consists of three loop levels. The innermost loop iterates over variable k and steps through one row of matrix **M** and one column of matrix **N**. The loop calculates a dot product of the row of **M** and the column of **N** and generates one element of **P**. Immediately after the innermost loop, the **P** element generated is written into the output **P** matrix.

The index used for accessing the **M** matrix in the innermost loop is `i*Width+k`. This is because the **M** matrix elements are placed into the system memory that is ultimately accessed with a linear address. That is, every location in the system memory has an address that ranges from 0 to the largest memory location. For C programs, the placement of a 2-dimensional matrix into this linear addressed memory is done according to the row-major convention, as illustrated in Figure 3.5.[1] All elements of a row are placed into consecutive memory locations. The rows are then placed one after another. Figure 3.5 shows an example where a 4×4 matrix is

[1]Note that FORTRAN adopts the column–major placement approach: All elements of a column are first placed into consecutive locations, and all columns are then placed in their numerical order.

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         float sum = 0;
         for (int k  = 0; k < Width; ++k) {
            float a = M[i * width + k];
            float b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```

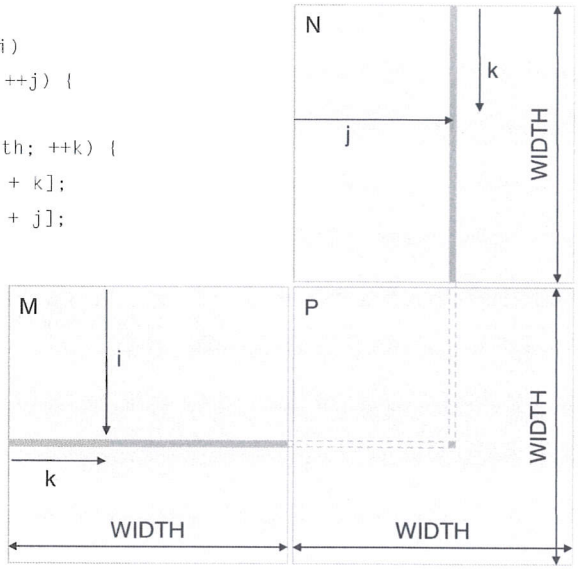


FIGURE 3.4

A simple matrix multiplication function with only host code.

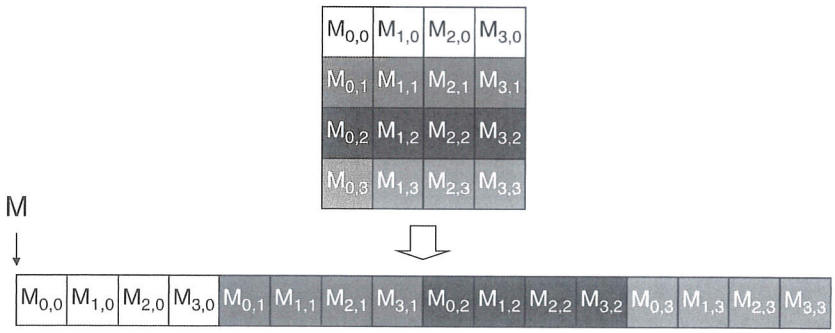


FIGURE 3.5

Placement of two-dimensional array elements into the linear address system memory.

revised function 修订后的函数.

outsourcing agent 外包代理

Tip: 因为CPU和GPU是不同的硬件设备.
∴GPU无法直接访问 Main Memory.
必须通过PCIE进行数据传输

Q.关于GPU直接从硬盘中获取数据.

GPU Direct RDMA (Remote Direct Memory Access)
绕过主机内存,直接读取数据.需要硬件支持
(也只有比较高端的支持).

直接访问硬存 ⇒ 编程复杂度 ①.传输控制 ②.一致性
并且即使直接访问硬存,也要通过PCIE传输

所以,除非是超大规模/流式数据.很难看到提升

Tip: CUDA编程必须经CPU,所以是有些不兼容的.

placed into 16 consecutive locations, with all elements of row 0 first followed by the four elements of row 1, etc. Therefore, the index for an **M** element in row *i* and column *k* is `i*Width+k`. The `i*Width` term skips over all elements of the rows before row *i*. The *k* term then selects the proper element within the section for row *i*.

The outer two (*i* and *j*) loops in Figure 3.4 jointly iterate over all rows of **M** and all columns of **N**; each joint iteration performs a row–column dot product to generate one **P** element. Each *i* value identifies a row. By systematically iterating all **M** rows and all **N** columns, the function generates all **P** elements. We now have a complete matrix multiplication function that executes solely on the CPU. Note that all of the code that we have shown so far is in standard C.

Assume that a programmer now wants to port the matrix multiplication function into CUDA. A straightforward way to do so is to modify the `MatrixMultiplication()` function to move the bulk of the calculation to a CUDA device. The structure of the revised function is shown in Figure 3.6. Part 1 of the function allocates device (GPU) memory to hold copies of the **M**, **N**, and **P** matrices and copies these matrices over to the device memory. Part 2 invokes a kernel that launches parallel execution of the actual matrix multiplication on the device. Part 3 copies the product matrix **P** from the device memory back to the host memory.

Note that the revised `MatrixMultiplication()` function is essentially an outsourcing agent that ships input data to a device, activates the calculation on the device, and collects the results from the device. The agent does so in such

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M and N to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

FIGURE 3.6

Outline of a revised host code `MatrixMultiplication()` that moves the matrix multiplication to a device.

compose 组成 构成.

pertinent data 相关数据.

bidirectional 双向

CUDA Runtime.

在Host和Device之间协调. - 组调用接口

eg. 设备内存分配 cudaMalloc().

数据搬运 cudaMemcpy() / cudaMemcpyAsync()

初始线程的运行上下文

显式分配内存

a way that the main program does not have to even be aware that the matrix multiplication is now actually done on a device. The details of the revised function, as well as the way to compose the kernel function, will serve as illustrations as we introduce the basic features of the CUDA programming model.

3.4 DEVICE MEMORIES AND DATA TRANSFER

In CUDA, the host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own dynamic random access memory (DRAM). For example, the NVIDIA T10 processor comes with up to 4 GB (billion bytes, or gigabytes) of DRAM. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 of Figure 3.6. Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed. This corresponds to Part 3 of Figure 3.6. The CUDA runtime system provides application programming interface (API) functions to perform these activities on behalf of the programmer. From this point on, we will simply say that a piece of data is transferred from host to device as shorthand for saying that the piece of data is transferred from the host memory to the device memory. The same holds for the opposite data transfer direction.

#可以理解CUDA_runtime就是一组接口.(运行平台)也可以理解成一个OS.

Figure 3.7 shows an overview of the CUDA device memory model for programmers to reason about the allocation, movement, and usage of the various memory types of a device. At the bottom of the figure, we see global memory and constant memory. These are the memories that the host code can transfer data to and from the device, as illustrated by the bidirectional arrows between these memories and the host. Constant memory allows read-only access by the device code and is described in Chapter 5. For now, we will focus on the use of global memory. Note that the host memory is not explicitly shown in Figure 3.7 but is assumed to be contained in the host.[2]

The CUDA memory model is supported by API functions that help CUDA programmers to manage data in these memories. Figure 3.8 shows the API functions for allocating and deallocating device global memory. The function `cudaMalloc()` can be called from the host code to allocate

[2]Note that we have omitted the texture memory from Figure 3.7 for simplicity. We will introduce texture memory later.

striking 突出，惊人的.

intentional 故意.

Constant Memory 常量内存. 64 KB

GPU的一种特殊内存类型，存储只读数据

CPU可以通过专用的 CUDA API 写入数据

所有的线程都可读取数据，全局共享

"__constant__"

Tip：针对这种特殊的只读内存，GPU 提供了相关的优化

广播机制

当多个线程同时访问 Constant Memory 中的一个地址时，GPU硬件会把这一堆访问优化为一次读取

从物理角度：Constant Memory 是 HBM 的一部分

从逻辑角度：特殊缓存模型，有独立的硬件缓存路径

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories

Host Memory

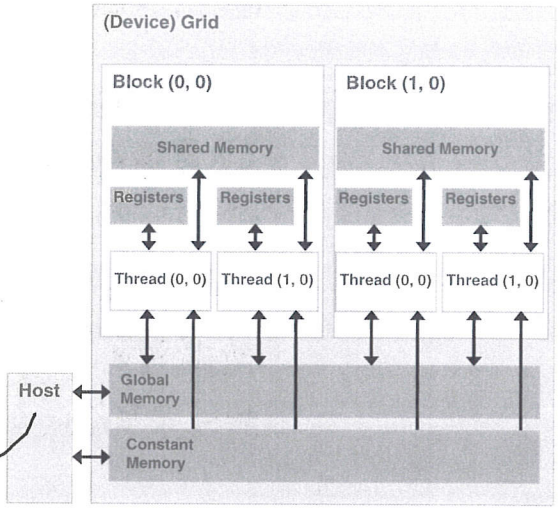


FIGURE 3.7

Overview of the CUDA device memory model.

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object

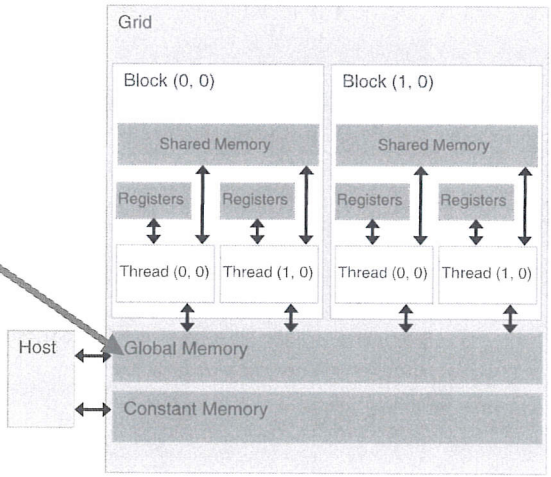


FIGURE 3.8

CUDA API functions for device global memory management.

a piece of global memory for an object. The reader should be able to notice the striking similarity between `cudaMalloc()` and the standard C runtime library `malloc()`. This is intentional; CUDA is C with minimal extensions. CUDA uses the standard C runtime library `malloc()` function to manage

be consistent with 与…保持一致

a continuation of …的延续

cudaMalloc((void**)& devicePtr, size)

Q1：为什么是指针变量地址

因为 cudaMalloc() 函数的第一个参数指向分配已后空间

也就是说 cudaMalloc() 会原地修改传参.

当参数本身是个指针的时候，就要传递指针的指针.

Tip：C/C++都是值传递哦！

Q2：为什么是[(void*)*]类型.

首先要理解的是类型和空间的大小强相关.

或者说类型这个概念就是为了计算空间大小而存在

这点在当时写 RVCC 的时候有了深刻体会.

所谓变量就是一片字节空间，类型是人为赋予的意义.

所以这里返回的是一片空间，自然可以是任何类型.

the host memory and adds `cudaMalloc()` as an extension to the C runtime library. By keeping the interface as close to the original C runtime libraries as possible, CUDA minimizes the time that a C programmer needs to relearn the use of these extensions.

The first parameter of the `cudaMalloc()` function is the address of a pointer variable that must point to the allocated object after allocation. The address of the pointer variable should be cast to `(void **)` because the function expects a generic pointer value; the memory allocation function is a generic function that is not restricted to any particular type of objects. This address allows the `cudaMalloc()` function to write the address of the allocated object into the pointer variable.[3] The second parameter of the `cudaMalloc()` function gives the size of the object to be allocated, in terms of bytes. The usage of this second parameter is consistent with the size parameter of the C `malloc()` function.

#指针的指针传递.
#第2个参数以字节单位

We now use a simple code example illustrating the use of `cudaMalloc()`. This is a continuation of the example in Figure 3.6. For clarity, we will end a pointer variable with the letter "d" to indicate that the variable is used to point to an object in the device memory space. The programmer passes the address of **Md** (i.e., `&Md`) as the first parameter after casting it to a void pointer; that is, **Md** is the pointer that points to the device global memory region allocated for the **M** matrix. The size of the allocated array will be `Width*Width*4` (the size of a single-precision floating number). After the computation, `cudaFree()` is called with pointer **Md** as input to free the storage space for the **M** matrix from the device global memory:

```
float *Md
int size = Width * Width * sizeof(float);
cudaMalloc((void**)&Md, size);
...
cudaFree(Md);
```

The reader should complete Part 1 of the `MatrixMultiplication()` example in Figure 3.6 with similar declarations of an **Nd** and a **Pd** pointer variable as

[3]Note that `cudaMalloc()` has a different format from the C `malloc()` function. The C `Malloc()` function returns a pointer to the allocated object. It takes only one parameter that specifies the size of the allocated object. The `cudaMalloc()` function writes to the pointer variable whose address is given as the first parameter. As a result, the `cudaMalloc()` function takes two parameters. The two-parameter format of `cudaMalloc()` allows it to use the return value to report any errors in the same way as other CUDA API functions.

cudaMemcpy().

```
cudaError_t cudaMemcpy(
        void* dst,
        const void* src,
        size_t count,
        cudaMemcpyKind kind );
```

只能作用在单个GPU硬件板子上

well as their corresponding `cudaMalloc()` calls. Furthermore, Part 3 in Figure 3.6 can be completed with the `cudaFree()` calls for **Nd** and **Pd**.

Once a program has allocated device global memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling one of the CUDA API functions, `cudaMemcpy()`, for data transfer between memories. Figure 3.9 shows the API function for such a data transfer. The `cudaMemcpy()` function takes four parameters. The first parameter is a pointer to the destination location for the copy operation. The second parameter points to the source data object to be copied. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the types of memory involved in the copy: from host memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory. For example, the memory copy function can be used to copy data from one location of the device memory to another location of the device memory. Please note that `cudaMemcpy()` cannot be used to copy between different GPUs in multi-GPU systems.

For the matrix multiplication example, the host code calls the `cudaMemcpy()` function to copy the **M** and **N** matrices from the host memory to the device memory before the multiplication and then to copy the **P** matrix from the device memory to the host memory after the multiplication is done.

- cudaMemcpy()
  - **Memory** data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Transfer is asynchronous

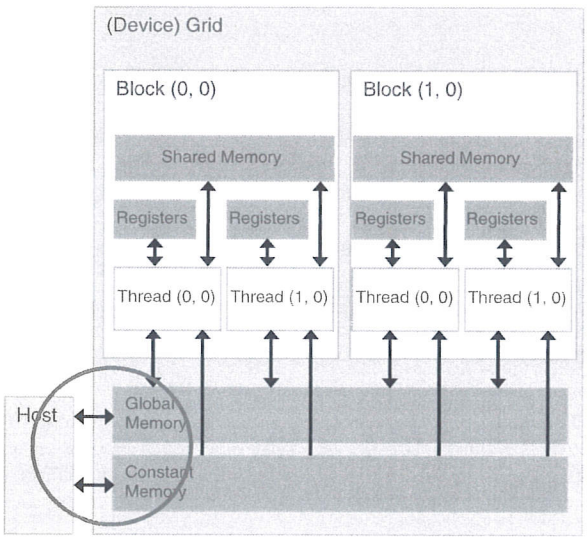


FIGURE 3.9

CUDA API functions for data transfer between memories.

symbolic constant. 符号常量

stub function. 桩函数，存根函数.

a more fleshed out version of ...
一个更完善/充实的... 版本.

Stub Function.

主机代码 (Host code) 调用
但实际在设备 (Device) 执行的函数.

Stub 词源.
软件：一种简化实现的占位函数/接

Assume that **M**, **P**, **Md**, **Pd**, and size have already been set as we discussed before; the two function calls are shown below. Note that the two symbolic constants, `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`, are recognized, predefined constants of the CUDA programming environment. The same function can be used to transfer data in both directions by properly ordering the source and destination pointers and using the appropriate constant for the transfer type:

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

To summarize, the main program in Figure 3.3 calls `MatrixMultiplication()`, which is also executed on the host. `MatrixMultiplication()`, as outlined in Figure 3.6, is responsible for allocating device memory, performing data transfers, and activating the kernel that performs the actual matrix multiplication. We often refer to this type of host code as the *stub function* for invoking a kernel. After the matrix multiplication, `MatrixMultiplication()` also copies result data from device to the host. We show a more fleshed out version of the `MatrixMultiplication()` function in Figure 3.10.

Compared to Figure 3.6, the revised `MatrixMultiplication()` function is complete in Part 1 and Part 3. Part 1 allocates device memory for **Md**,

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;

1. // Transfer M and N to device memory
  cudaMalloc((void**) &Md, size);
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMalloc((void**) &Nd, size);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

  // Allocate P on the device
  cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code - to be shown later
  ...
3. // Transfer P from device to host
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
  // Free device matrices
  cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

FIGURE 3.10

The revised `MatrixMultiplication()` function.

counterparts. of ···的对应部分.

the device counterparts of 对应GPU上内存分配么.....

Nd, and **Pd**, the device counterparts of **M**, **N**, and **P**, and transfers **M** to **Md** and **N** to **Nd**. This is accomplished with calls to the `cudaMalloc()` and `cudaMemcpy()` functions. The readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Figure 3.10. Part 2 invokes the kernel and will be described in the following text. Part 3 reads the product data from device memory to host memory so the value will be available to `main()`. This is accomplished with a call to the `cudaMemcpy()` function. It then frees **Md**, **Nd**, and **Pd** from the device memory, which is accomplished with calls to the `cudaFree()` functions.

3.5 KERNEL FUNCTIONS AND THREADING

We are now ready to discuss more about the CUDA kernel functions and the effect of invoking these kernel functions. In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Because all of these threads execute the same code, CUDA programming is an instance of the well-known single-program, multiple-data (SPMD) parallel programming style [Atallah 1998], a popular programming style for massively parallel computing systems.[4]

Figure 3.11 shows the kernel function for matrix multiplication. The syntax is ANSI C with some notable extensions. First, there is a CUDA-specific keyword "`__global__`" in front of the declaration of `MatrixMulKernel()`. This keyword indicates that the function is a kernel and that it can be called from a host functions to generate a grid of threads on a device.

In general, CUDA extends C function declarations with three qualifier keywords. The meanings of these keywords are summarized in Figure 3.12. The `__global__` keyword indicates that the function being declared is a CUDA kernel function. The function will be executed on the device and can only be called from the host to generate a grid of threads on a device. We will show the host code syntax for calling a kernel function later in Figure 3.14. Besides `__global__`, there are two other keywords that can be used in front of a function declaration. Figure 3.12 summarizes the

[4]Note that SPMD is not the same as single instruction, multiple data (SIMD). In an SPMD system, the parallel processing units execute the same program on multiple parts of the data; however, these processing units do not have to be executing the same instruction at the same time. In an SIMD system, all processing units are executing the same instruction at any instant.

Q1 __device__ 关键字. CUDA device function.

该函数or变量定义在GPU的设备内存中.

并且只能在GPU上执行的代码中调用or访问.

通过CUDA Runtime完成这段初始化

首先,明确 C/C++ ⇒ PTX(IR) ⇒ GPU二进制代码. SASS

被"__device__"修饰的函数代码,在程序启动时把已翻译好的二进制传输到GPU指令存储区.

并且因为"__device__"修饰的函数在GPU片上,无法被主机host访问到,所以主机只能通过"__global__"函数来间接调用"__device__"函数.

Tip: GPU上也可以执行递归,但执行递归的函数必须是被"__device__"修饰的函数.

当然,因为递归的特点,在GPU的执行不高效.

不过一些分治算法会合适, eg. QuickSort.

Q2. 也可同时使用的关键字 __host__, __device__

不存在一个 __host__device__ 这样的关键字

声明该函数可以在 Host / Device 端执行

编译会根据函数的被调用情况自动判断

注[2]:这个程序的索引 tx, ty 虽然是以矩阵的行/列为单位的.但是线程的并行粒度仍然是像素(单个元素).

因为线程块中的线程共享数据.单个线程无法直接访问目标数据.必须通过 threadIdx 提供的索引

Q3. Device Function 不能包含调用,也不能通过指针进行间接的函数调用.

准确说,并没有语法错误,只是极其不推荐.

因为很多情况递归深度不可控,单个线程没那么多资源.

而间接的函数调用就是指C++那种函数指针的使用.

函数指针地址只能是运行时解析的.GPU没有硬件支持.

注[1]:在生成很多线程的时候.该线程的执行到访问自己的线程位置(相对),从而访问到需要自己处理的数据,避免了循环线性获取目标数据的过程.

且线程执行以块为单位 ⇒ 数据在线程块中共享.

所以只需要 threadIdx 提供的相对位置.

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
    float Mdelement = Md[ty * Width + k];
    float Ndelement = Nd[k * Width + tx];
    Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```

FIGURE 3.11

The matrix multiplication kernel function.

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

FIGURE 3.12

CUDA extensions to C functional declaration.

meaning of these keywords. The `__device__` keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can only be called from a kernel function or another device function. Device functions can have neither recursive function calls nor indirect function calls through pointers in them. The `__host__` keyword indicates that the function being declared is a CUDA host function. A host function is simply a traditional C function that executes on the host and can only be called from another host function. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration. This makes sense, as many CUDA applications are ported from CPU-only execution environments. The programmer would add kernel functions and device functions

notable 显著/值得注意.

multidimensional 多维

Q. the porting process. (移植过程)

程序从 CPU 平台转移到 GPU 平台的过程

eg. 指定执行参数 <<< grid, block >>>

当然好的移植还需要针对并行特点进行很多优化

但是 nvcc + 参数指定，把程序执行走下来是 ok 的.

Q. threadIdx 关键字.

CUDA 的一个内置变量，三维向量类型. (x, y, z)

每个线程块由很多个线程组成，每个线程都有一个独立的 threadIdx 的值.

threadIdx 的范围就是一个线程块的范围.

一个线程块的总数 = threadIdx.x · ~.y · ~.z.

线程的索引并不直接决定硬件资源，

而是从逻辑上决定它应该处理哪些数据.

Q. 网格与线程块的层级关系.

网格中的每个线程块通过 blockIdx 标识

一个线程在全局中的索引位置，可计算

Spec ThreadIdx = ThreadIdx + BlockDim * BlockIdx.

BlockDim：一个线程块中的线程数量.

<类比基址寻址的过程>

during the porting process. The original functions remain as host functions. Having all functions default into host functions spares the programmer the tedious work of changing all original function declarations.

Note that one can use both `__host__` and `__device__` in a function declaration. This combination triggers the compilation system to generate two versions of the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function. This supports a common use when the same function source code can be simply recompiled to generate a device version. Many user library functions will likely fall into this category.

Other notable extensions of ANSI C, in Figure 3.11, are the keywords `threadIdx.x` and `threadIdx.y`, which refer to the thread indices of a thread. Note that all threads execute the same kernel code. There needs to be a mechanism to allow them to distinguish themselves and direct themselves toward the particular parts of the data structure that they are designated to work on. These keywords identify predefined variables that allow a thread to access the hardware registers at runtime that provide the identifying coordinates to the thread. Different threads will see different values in their `threadIdx.x` and `threadIdx.y` variables. For simplicity, we will refer to a thread as $Thread_{threadIdx.x,\ threadIdx.y}$. Note that the coordinates reflect a multidimensional organization for the threads. We will come back to this point soon.

各个线程通过下标来区分.

A quick comparison of Figure 3.4 and Figure 3.11 reveals an important insight for CUDA kernel functions and CUDA kernel invocation. The kernel function in Figure 3.11 has only one loop, which corresponds to the innermost loop in Figure 3.4. The readers should ask where the other two levels of outer loops go. The answer is that the outer two loop levels are now replaced with the grid of threads. The entire grid forms the equivalent of the two-level loop. Each thread in the grid corresponds to one of the iterations of the original two-level loop. The original loop variables *i* and *j* are now replaced with *threadIdx.x* and *threadIdx.y*. Instead of having the loop increment the values of *i* and *j* for use in each loop iteration, the CUDA threading hardware generates all of the *threadIdx.x* and *threadIdx.y* values for each thread.

In Figure 3.11, each thread uses its *threadIdx.x* and *threadIdx.y* to identify the row of **Md** and the column of **Nd** to perform the dot product operation. It should be clear that these indices simply take over the role of variables *i* and *j* in Figure 3.8. Note that we assigned *threadIdx.x* to the automatic C variable *tx* and *threadIdx.y* to variable *ty* for brevity in

be comprised of 由…组成

invocation 调用/引用

hazard-free 无冲突的.

Figure 3.8. Each thread also uses its *threadIdx.x* and *threadIdx.y* values to select the **Pd** element that it is responsible for; for example, $Thread_{2,3}$ will perform a dot product between column 2 of **Nd** and row 3 of **Md** and write the result into element (2,3) of **Pd**. This way, the threads collectively generate all the elements of the **Pd** matrix.

When a kernel is invoked, or *launched*, it is executed as *grid* of parallel threads. In Figure 3.13, the launch of Kernel 1 creates Grid 1. Each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example, each element of a large array might be computed in a separate thread.

grid → block → thread

Threads in a grid are organized into a two-level hierarchy, as illustrated in Figure 3.13. For simplicity, a small number of threads are shown in Figure 3.13. In reality, a grid will typically consist of many more threads. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. In Figure 3.13, Grid 1 is organized as a 2×2 array of 4 blocks. Each block has a unique two-dimensional coordinate given by the CUDA specific keywords `blockIdx.x` and `blockIdx.y`. All thread blocks must have the same number of threads organized in the same manner.

- A thread block is a batch of threads that can cooperate with each other by:
  - — Synchronizing their execution
    - For hazard-free shared memory accesses
  - — Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate

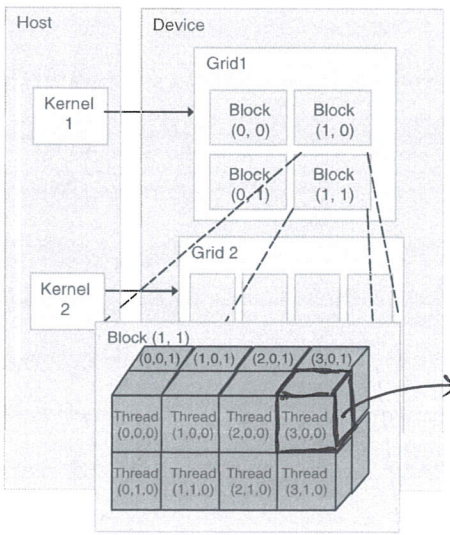


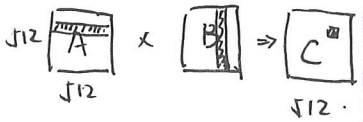
FIGURE 3.13

CUDA thread organization.

Each thread block is, in turn, organized as a three-dimensional array of threads with a total size of up to 512 threads. The coordinates of threads in a block are uniquely defined by three thread indices: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. Not all applications will use all three dimensions of a thread block. In Figure 3.13, each thread block is organized into a 4×2×2 three-dimensional array of threads. This gives Grid 1 a total of 4×16 = 64 threads. This is obviously a simplified example.

In the matrix multiplication example, a grid is invoked to compute the product matrix. The code in Figure 3.11 does not use any block index in accessing input and output data. Threads with the same `threadIdx` values from different blocks would end up accessing the same input and output data elements. As a result, the kernel can use only one thread block. The `threadIdx.x` and `threadIdx.y` values are used to organize the block into a two-dimensional array of threads. Because a thread block can have only up to 512 threads, and each thread calculates one element of the product matrix in Figure 3.11, the code can only calculate a product matrix of up to 512 elements. This is obviously not acceptable. As we explained before, the product matrix must have millions of elements in order to have a sufficient amount of data parallelism to benefit from execution on a device. We will address this issue in Chapter 4 using multiple blocks.

一个线程块只能处理两个512大小的矩阵乘



When the host code invokes a kernel, it sets the grid and thread block dimensions via *execution configuration* parameters. This is illustrated in Figure 3.14. Two `struct` variables of type `dim3` are declared. The first is for describing the configuration of blocks, which are defined as 16×16 groups of threads. The second variable, `dimGrid`, describes the configuration of the grid. In this example, we only have one (1×1) block in each grid. The final line of code invokes the kernel. The special syntax between the name of the kernel function and the traditional C parameters of the function is a CUDA extension to ANSI C. It provides the dimensions of the grid in terms of number of blocks and the dimensions of the blocks in terms of number of threads.

使用dim3类型声明了两个结构体实例.

```
// Setup the execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

FIGURE 3.14

Example of host code that launches a kernel.

heterogeneous 异构的

Q. __device__/__global__/__host__ 区别

__device__：只能在GPU执行，并且被 kernel/device 调用

__host__：CPU执行.

__global__：CPU调度在GPU运行（kernel）

3.6 SUMMARY

This chapter serves as a quick overview of the CUDA programming model. CUDA extends the C language to support parallel computing. The extensions discussed in this chapter are summarized below.

3.6.1 Function declarations

CUDA extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Figure 3.12. Using one of `__global__`, `__device__`, or `__host__`, a CUDA programmer can instruct the compiler to generate a kernel function, a device function, or a host function. If both `__host__` and `__device__` are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host. If a function declaration does not have any CUDA extension keyword, the function defaults into a host function.

3.6.2 Kernel launch

CUDA extends C function call syntax with kernel execution configuration parameters surrounded by <<< and >>>. These execution configuration parameters are only used during a call to a kernel function, or a kernel launch. We discussed the execution configuration parameters that define the dimensions of the grid and the dimensions of each block. The reader should refer to the *CUDA Programming Guide* [NVIDIA 2007] for more details regarding the kernel launch extensions as well as other types of execution configuration parameters.

3.6.3 Predefined variables

CUDA kernels can access a set of predefined variables that allow each thread to distinguish among themselves and to determine the area of data each thread is to work on. We discussed the `threadIdx` variable in this chapter. In Chapter 4, we will further discuss `blockIdx`, `gridDim`, and `blockDim` variables.[5]

[5]Note that the gridDim and blockDim variables are built-in, predefined variables that are accessible in kernel functions. They should not be confused with the user defined dimGrid and dimBlock variables that are used in the host code for the purpose of setting up the configuration parameters. The value of these configuration parameters will ultimately become the values of gridDim and blockDim once the kernel has been launched.

be referred to. 被称为/提及
参阅…资料，了解…

by no means 绝不

emphasis 强调，重点

Q4. CUDA体系 = ① + ②

① CUDA Toolkit.
- CUDA Compiler.
- Developer Tools.
  - DEBUGGERS.
  - PROFILERS.
- CUDA C++ CORE

② CUDA Driver.
- Memory Management.
- Windows & Graphics.
- Commons Lib

USER-ACCESS.	GPT. Recommender. AI models.
APPS / Frame.	TF, PyTorch. ONNX, Panda.
CUDA-X Lib.	Machine Learning / HPC.
CUDA.	CUDA 体系如上.
Platform.	Operating System.

峰值算力的计算.

$$\text{Peak FLOPS} = F_{clk} \times N_{SM} \times F_{req}$$

F_{clk}：GPU一个时钟周期内指令的执行数.
N_{SM}：GPU中的SM数量.
F_{req}：运行频率.

3.6.4 Runtime API

CUDA supports a set of API functions to provide services to CUDA programs. The services that we discussed in this chapter are `cudaMalloc()` and `cudaMemcpy()` functions. These functions allocate device memory and transfer data between the host and device on behalf of the calling program. The reader is referred to the *CUDA Programming Guide* [NVIDIA 2007] for other CUDA API functions.

Our goal for this chapter is to introduce the fundamental concepts of the CUDA programming model and the essential CUDA extensions to C for writing a simple CUDA program. The chapter is by no means a comprehensive account of all CUDA features. Some of these features will be covered in the rest of the book; however, our emphasis will be on key concepts rather than details. In general, we would like to encourage the reader to always consult the *CUDA Programming Guide* for more details on the concepts that we cover.

References and Further Reading

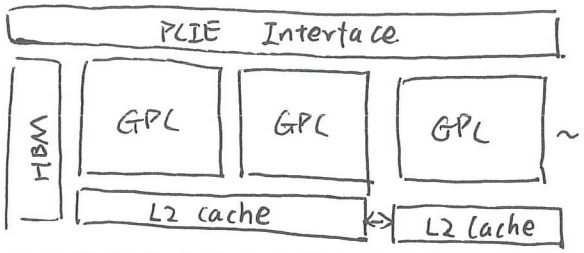
Atallah, M. J. (Ed.), (1998). *Algorithms and theory of computation handbook*. Boca Raton, FL: CRC Press.

NVIDIA. (2009). *CUDA programming guide 2.3*. Santa Clara, CA: NVIDIA.

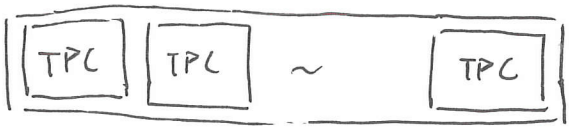
Stratton, J. A., Stone, S. S., & Hwu, W. W. (2008). MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 21st International Workshop on languages and compilers for parallel computing (LCPC)*. Canada: Edmonton.

CUDA的硬件支持. 以A100为例.

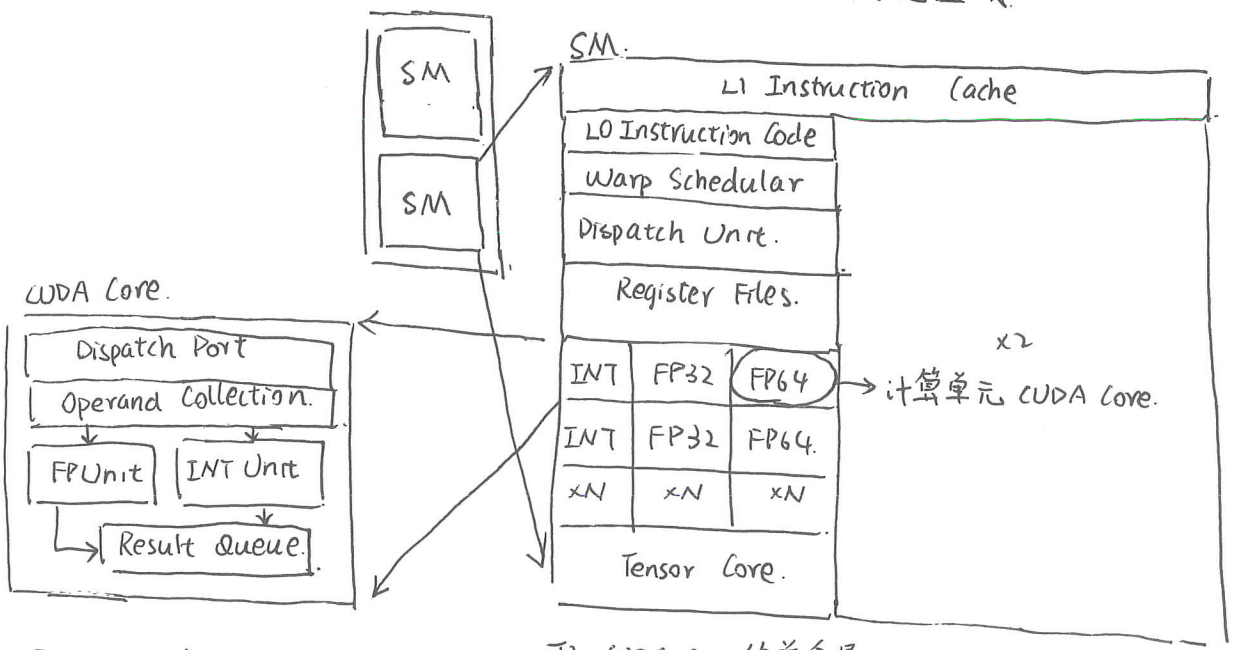
GPU由GPC / Graph Processed Cluster 组成



深入GPC, 由TPC Texture Processed Cluster 组成



深入TPC, 由SM真正的线程执行单元组成



Tip: CUDA Core的前身是 SP. Streaming Processor

Q5. 线程执行与硬件的关系.

一个线程块的线程都是放在一个SM中执行

又因为线程超配的原因, 一个SM中的计算单元不可能被全部激活.

所以Register Files会动态的分配给激活线程使用

eg. 一个总计算量K的任务.

一个SM至多并行2048个线程.

一个线程块包含256个线程 ⇒ 1SM=8Block

在硬件层面引入Warp. 控制线程的锁同步.

Tip: 一个SM中会有多个逻辑上的线程Block执行.

一个SM内部会有共享内存, 但是线程块之间是不能共享的.

同时, 在CUDA编程中, 一个CUDA kernel函数的执行. 必须显式声明它的 <<<grid, block>>>

grid: 需要把计算任务切分多个块.

block: 每个块中可以包含多少线程

参照图片的切割. 前端的像素任务会被映射到硬件上

Tip: 一个SM中线程块的分配也会受到共享内存的约束