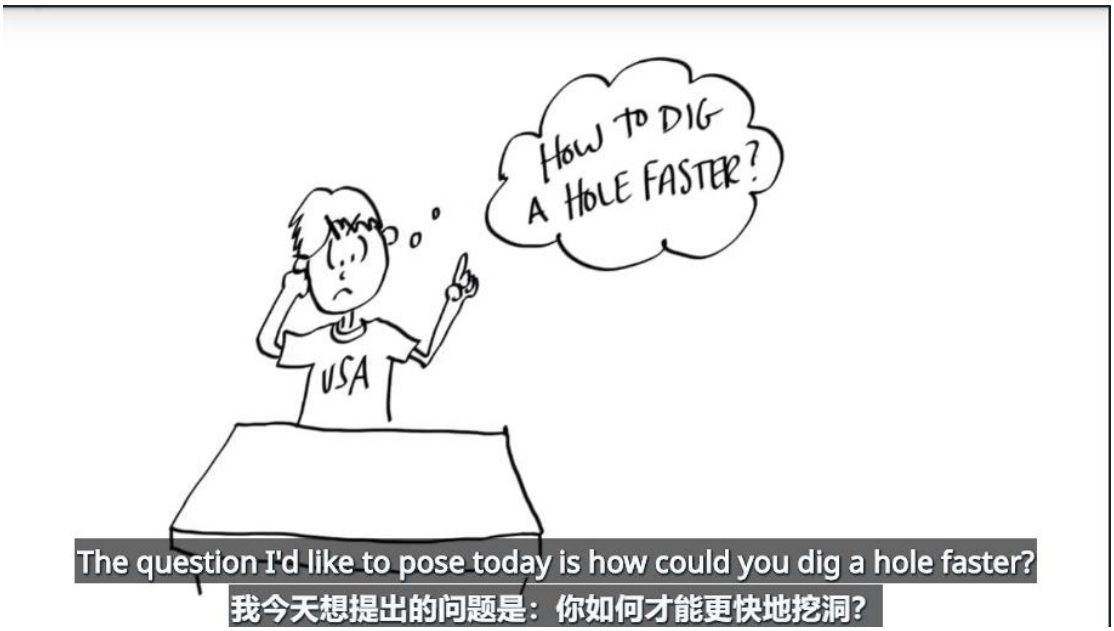
Intro to Parallel Programming

https://classroom.udacity.com/courses/cs344/lessons/55120467/concepts/658304810923

Noted by 林庆泓

Lesson 1 - The GPU Programming Model

1.4.Digging Holes

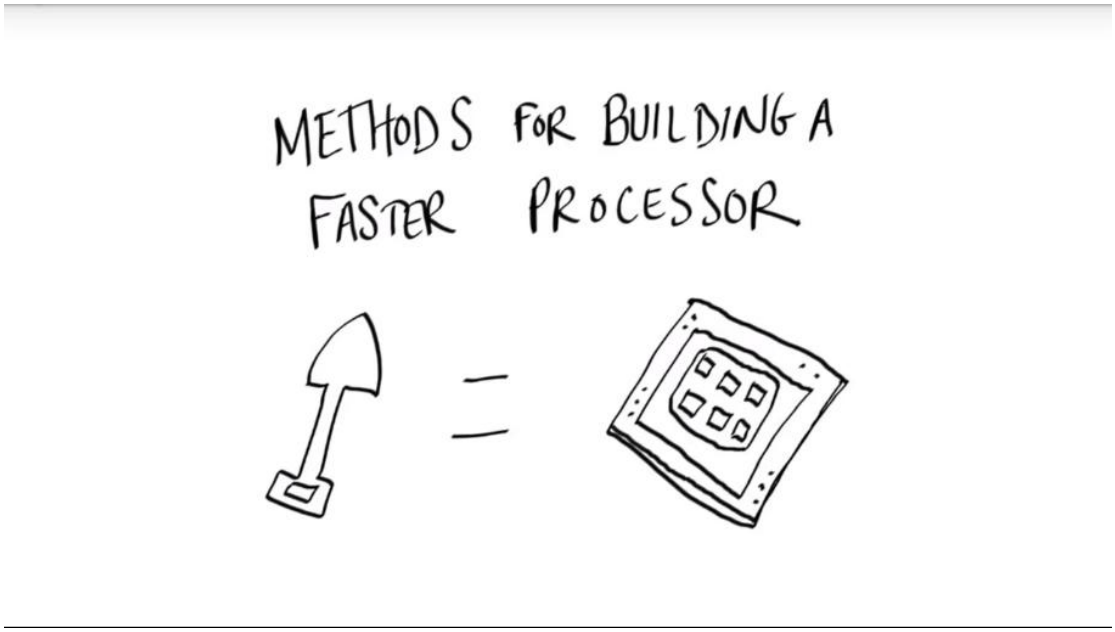


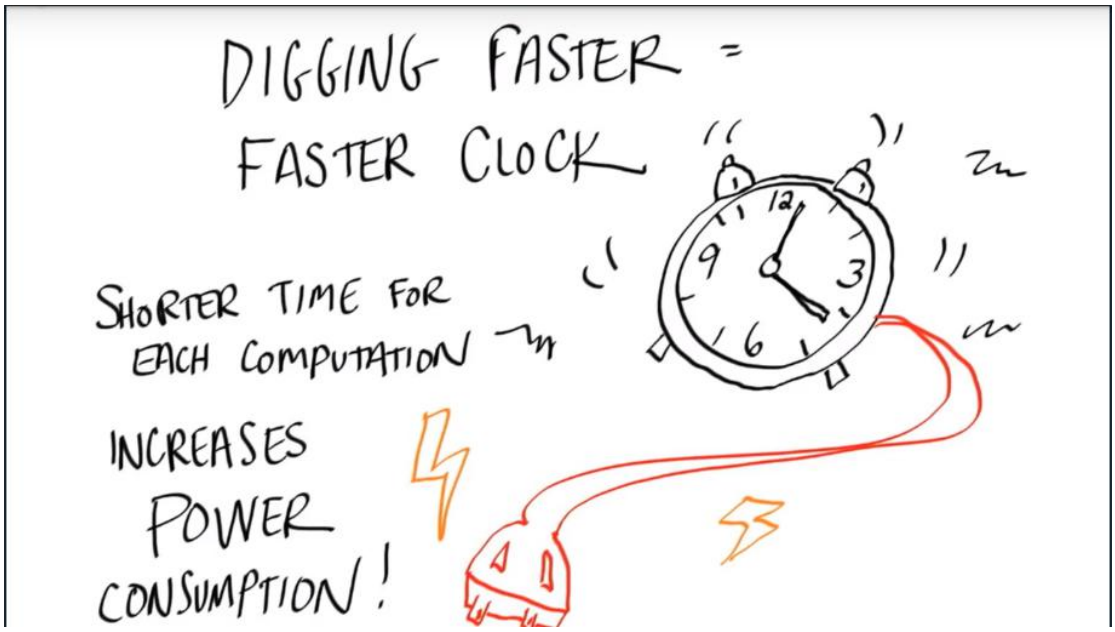
- 挖得更快

- 买一把效率更高的铲子

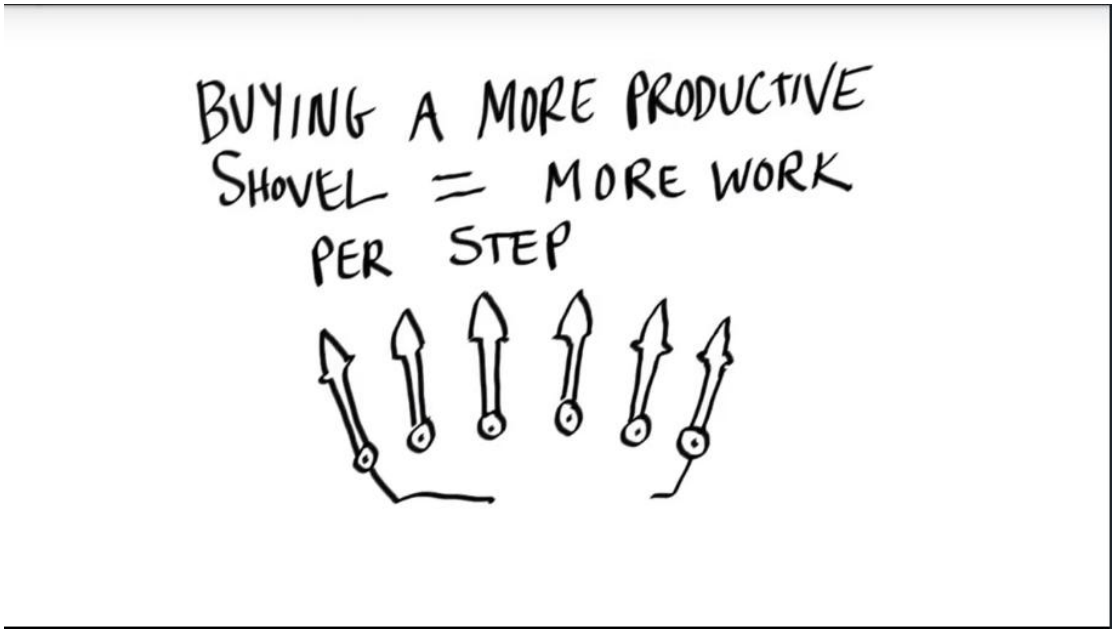
- 雇佣更多挖掘者

• 不仅仅是讨论挖洞，构造处理器也同理

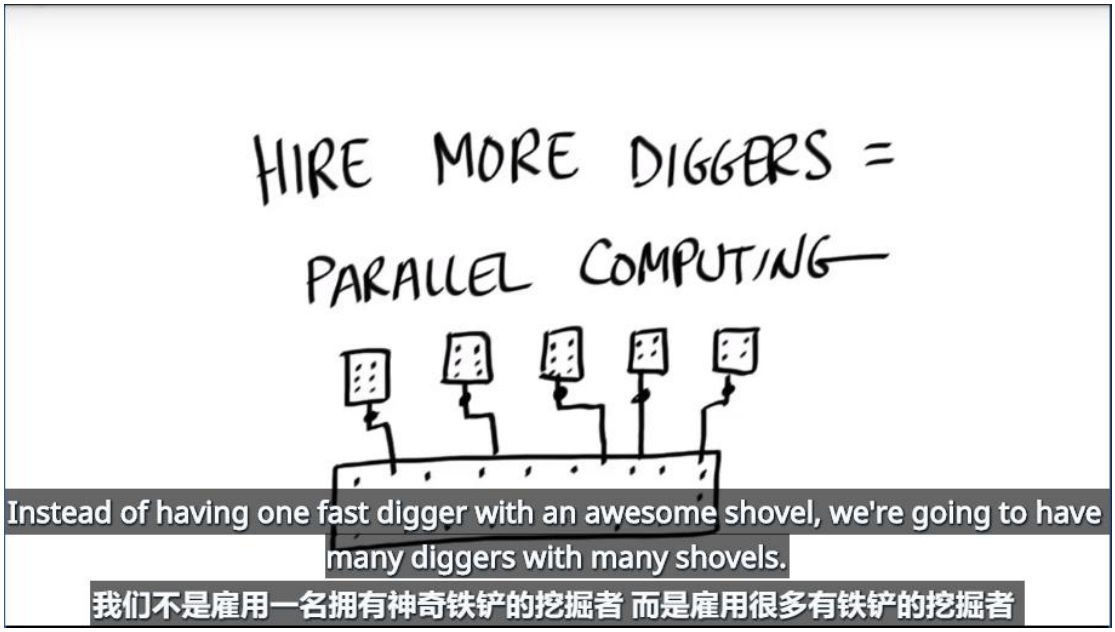




• 更快挖掘是让处理器以更快的时钟频率运行，在一次计算的每一步花费较短的时间。但在现代处理器中，提高时钟频率也会增加功耗。而我们受一块芯片上的功耗限制。

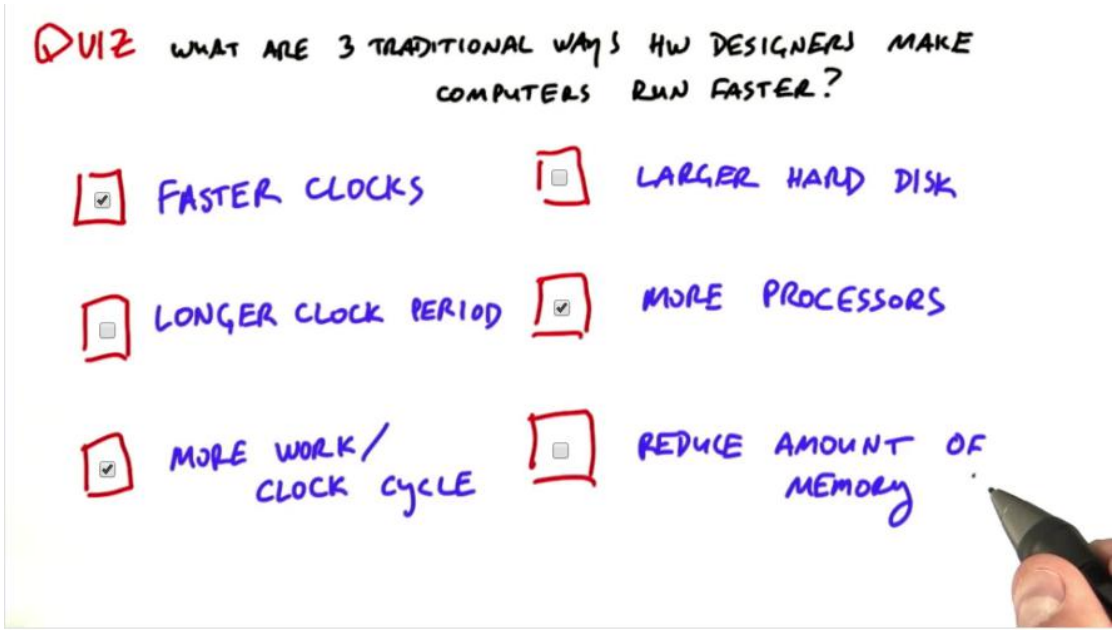


•当讨论购买更有效率的铲子，其实是要求处理器在每个步骤、每个时钟周期做更多的工作。



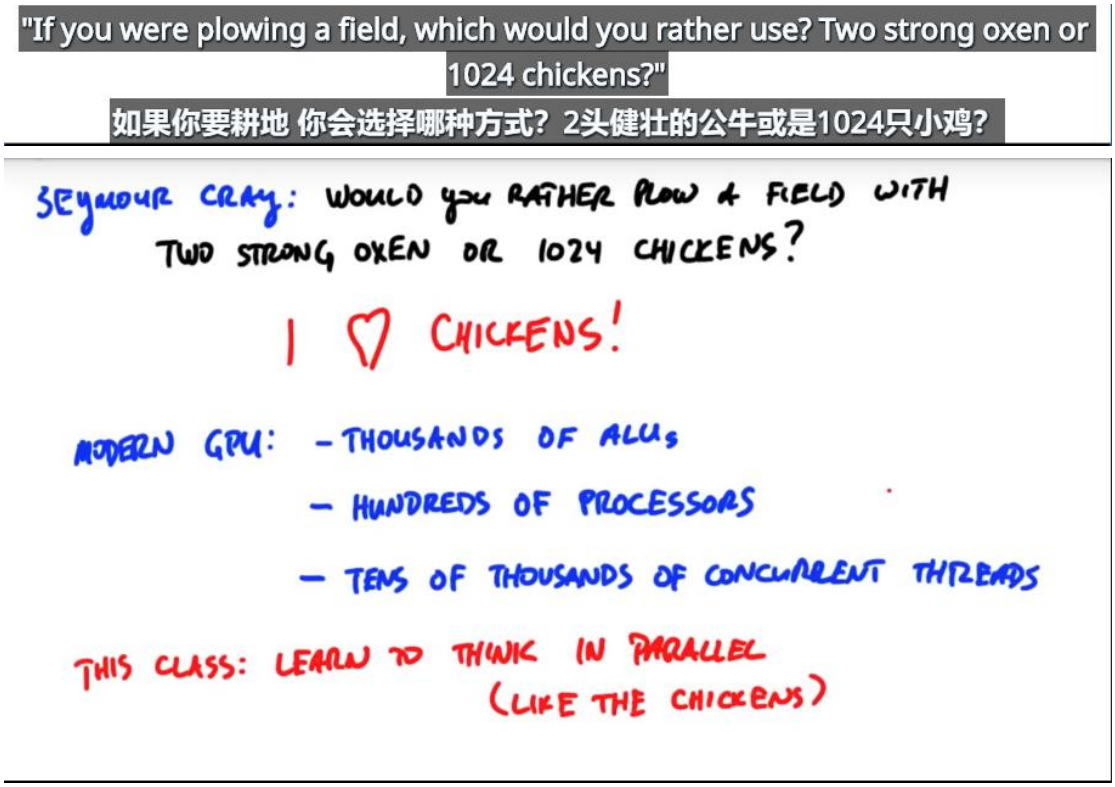
• 当谈到雇佣更多的挖掘者时，某种程度指的是并行计算。

1.5.How To Make Computers Run Faster



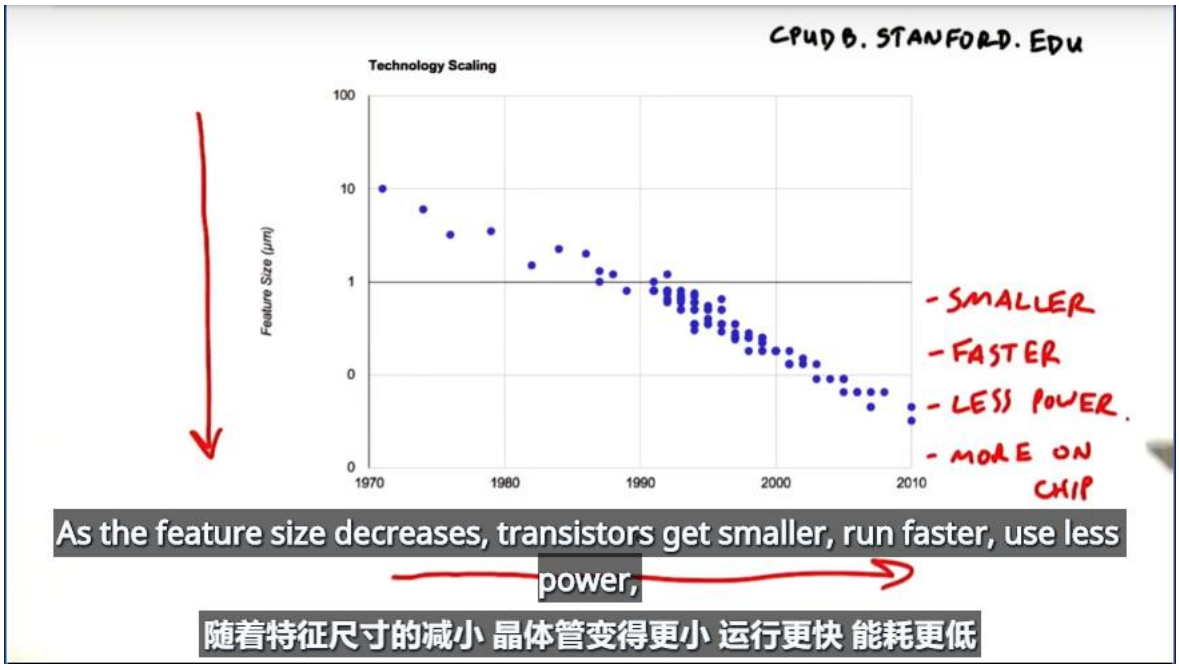
• 更快的时钟（更有效率的挖掘者）、每个时钟进行更多工作（超级铁铲）、更多处理器（雇佣更多挖掘者）。

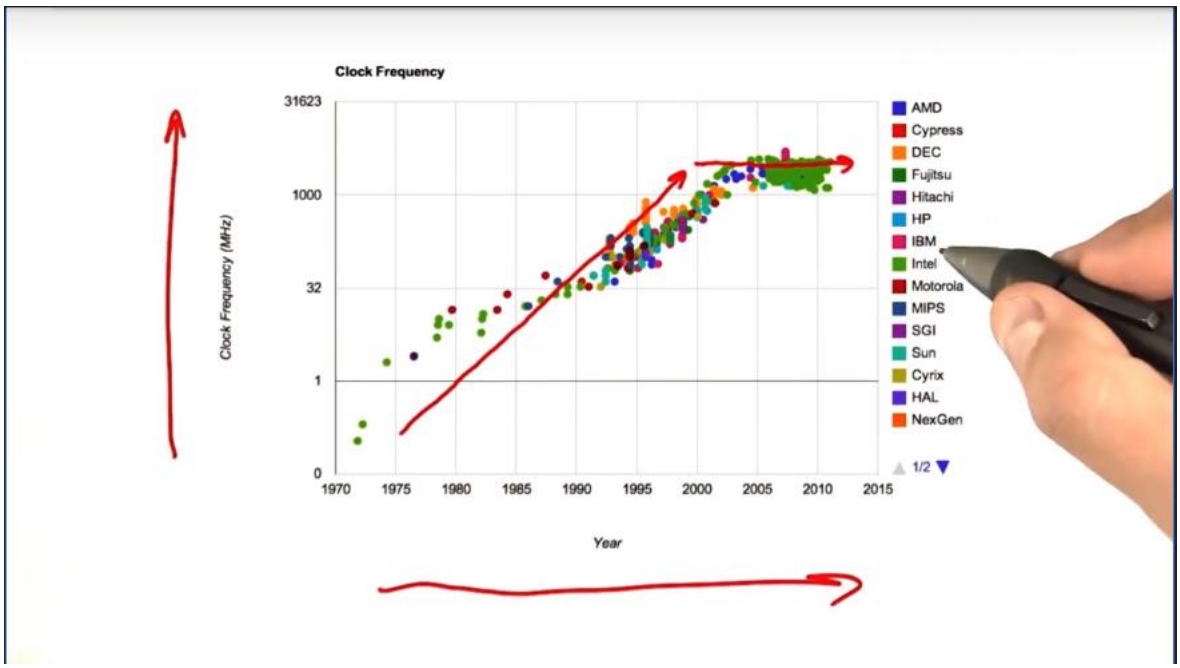
1.6.Chickens or Oxen?



• GPU: 图像处理单元上的通用编程（general purpose programmability on the graphics processing unit）

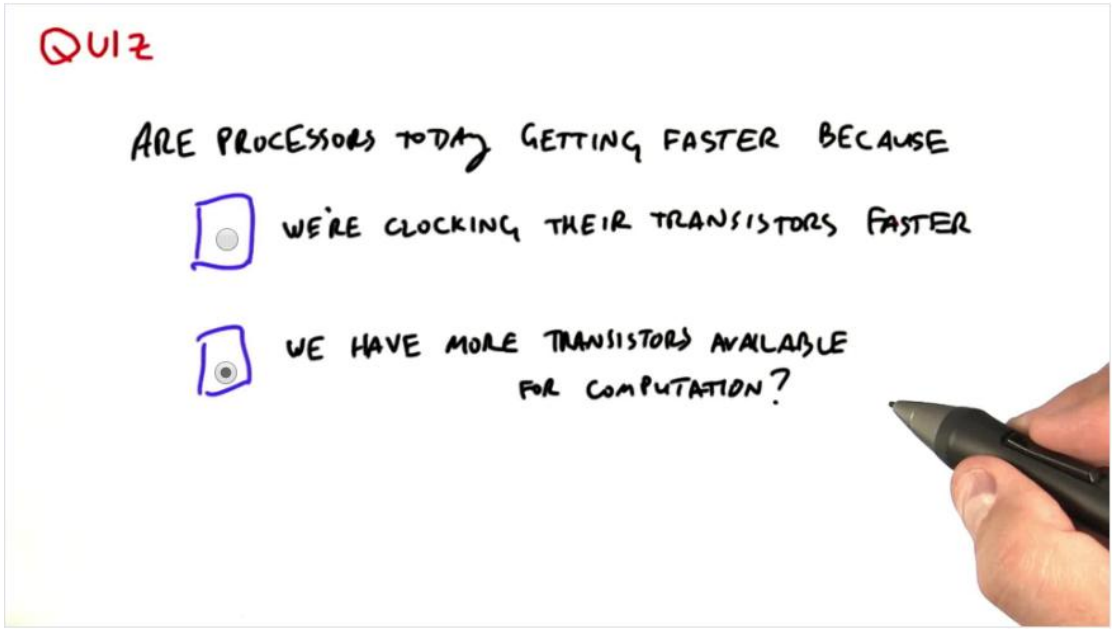
1.7.CPU Speed Remaining Flat





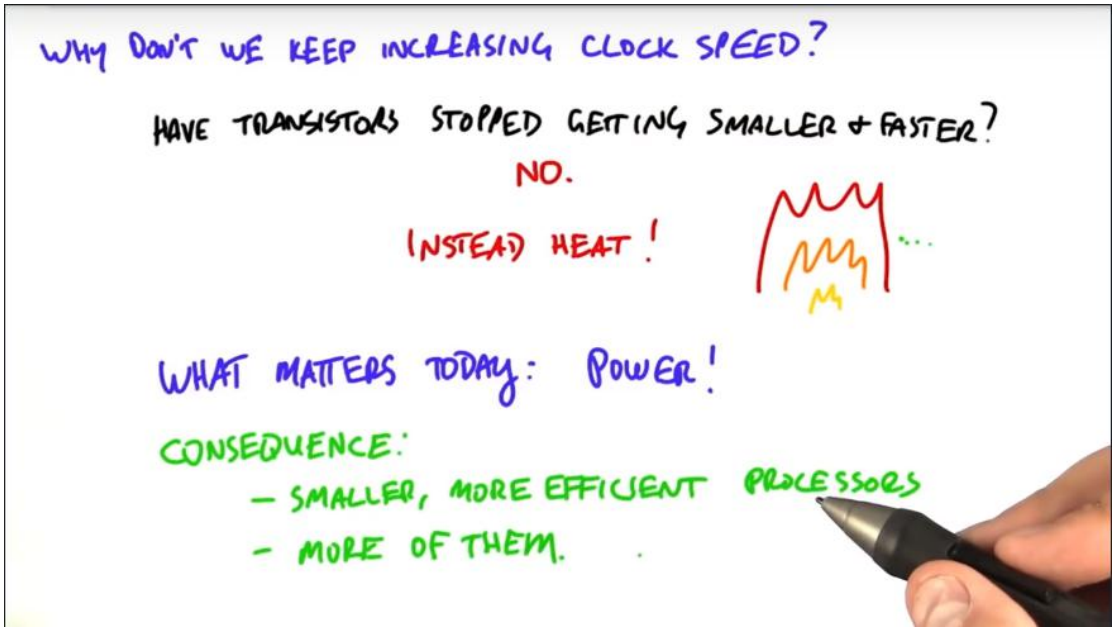
• 随着时间推移，时钟速度提升，但近几年却保持不变。

1.8.How Are CPUs Getting Fastter?



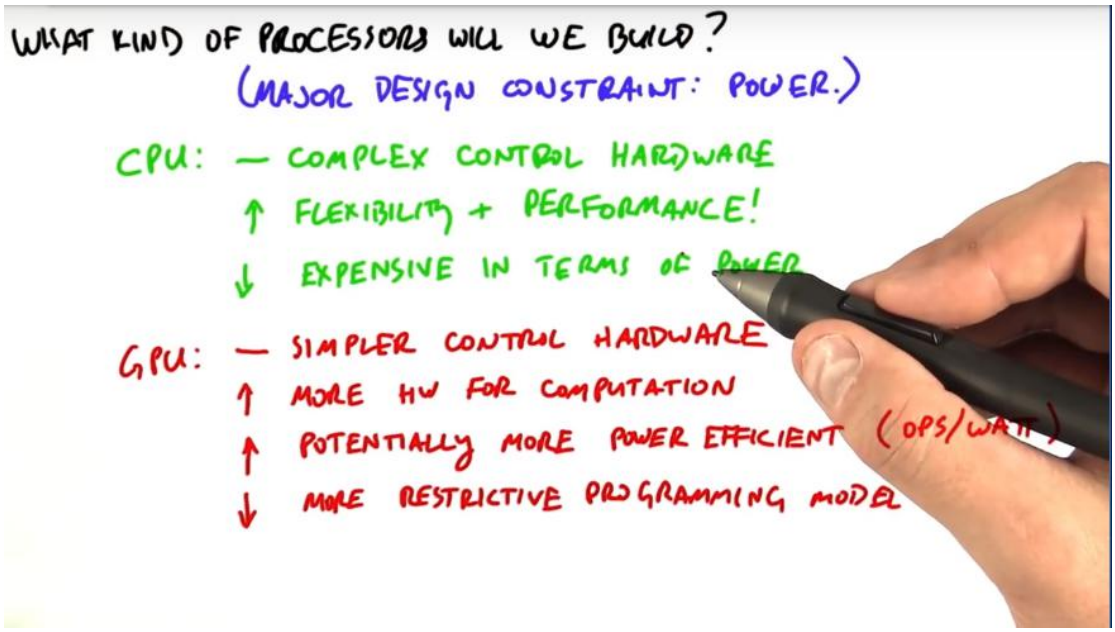
• 处理器一代比一代快是因为有更多的晶体管进行计算而不是晶体管运行的更快。

1.9.Why We Cannot Keep Increasing CPU Speed



• 电力变成了最主要的驱动因素。过去使单个处理器变得越来越快，但我们无法使其保持冷却。

1.10.What Kind of Processors Are We Building



• 假设电力是束缚

• CPU:central processing unit

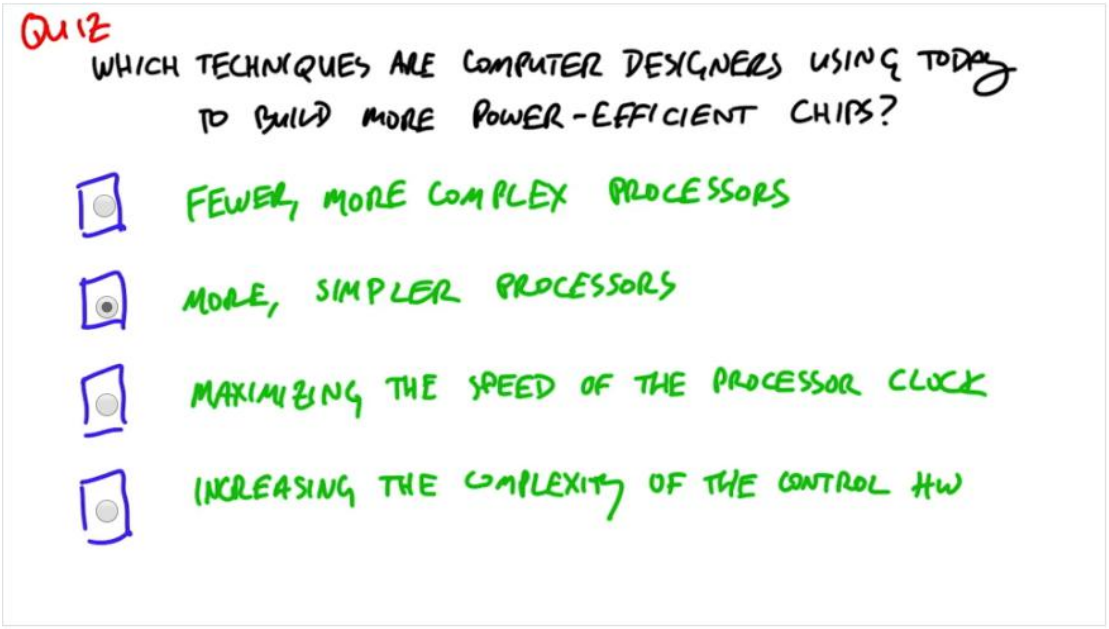
性能灵活、控制硬件复杂、电力和设计复杂度代价昂贵

于是采用简单结构支持更多数据路径计算

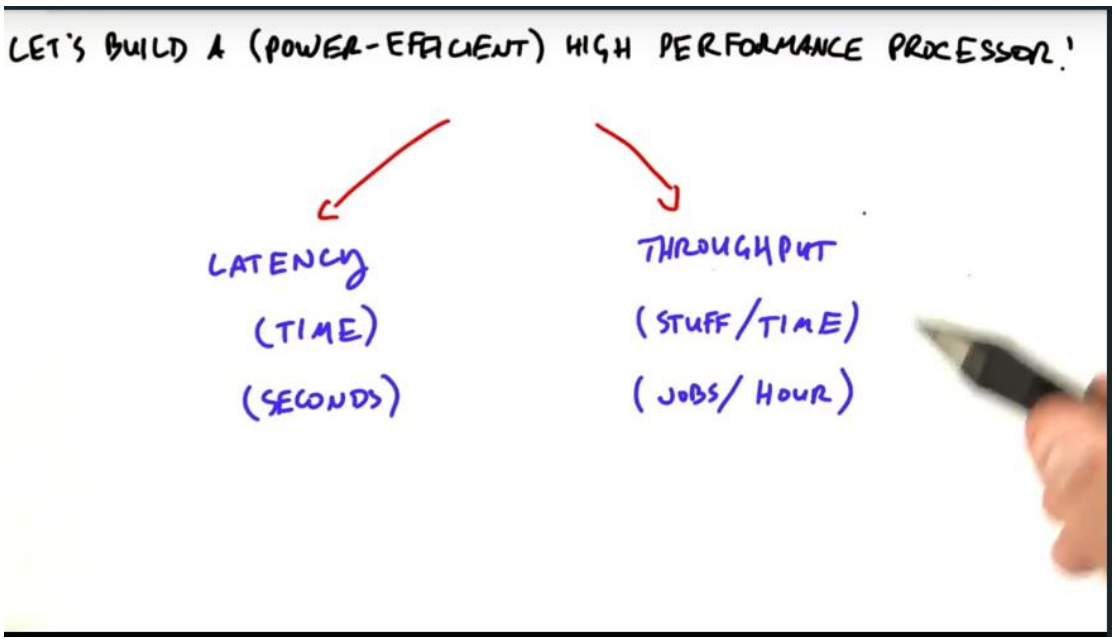
• GPU:graphics processing unit

计算单元小而简单，能效高

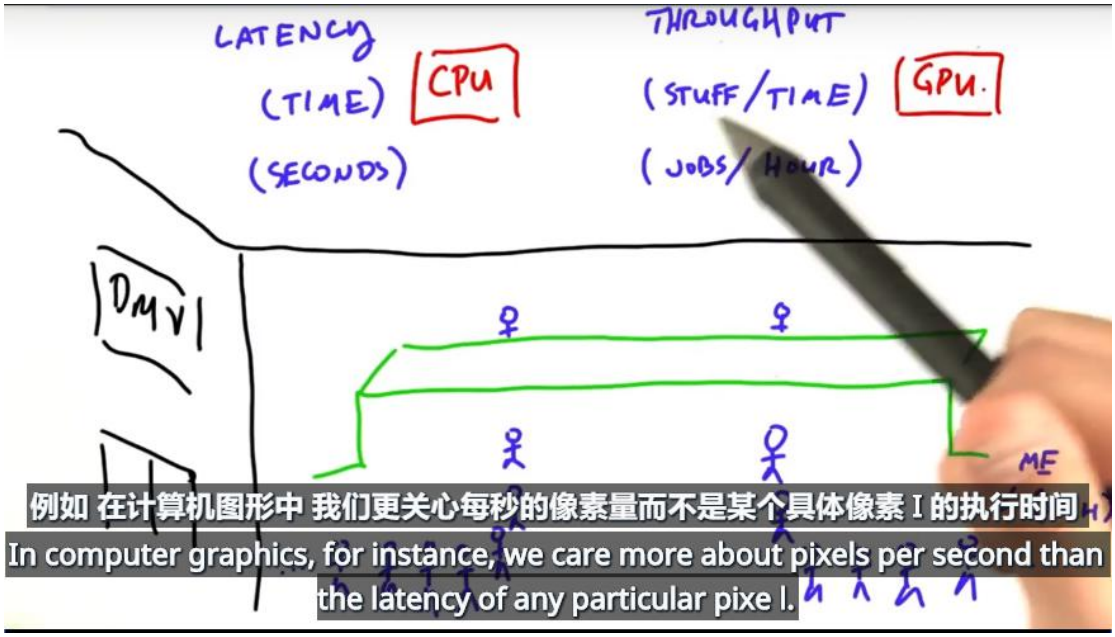
1.11.Techniques To Building Power-efficient Chips



1.12.Building A Power Efficient Processor

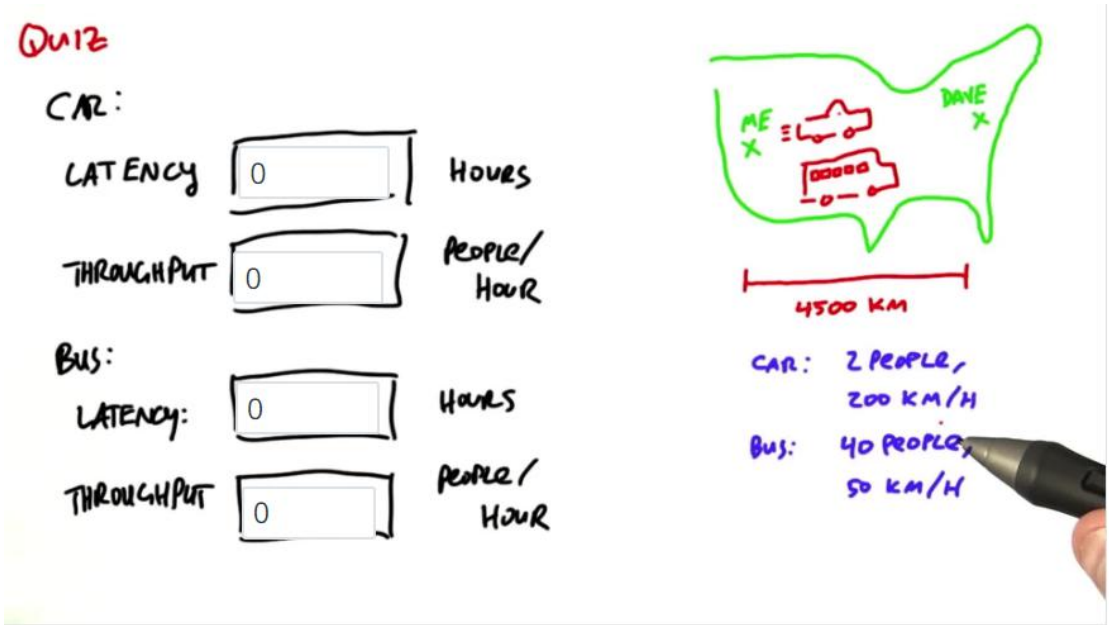


• 如何考虑能耗最佳
一是最大程度缩小执行时间、另一个是吞吐量（单位时间完成的任务量）



• CPU 优化执行时间、GPU 优化吞吐量

1.13.Latency vs Bandwidth

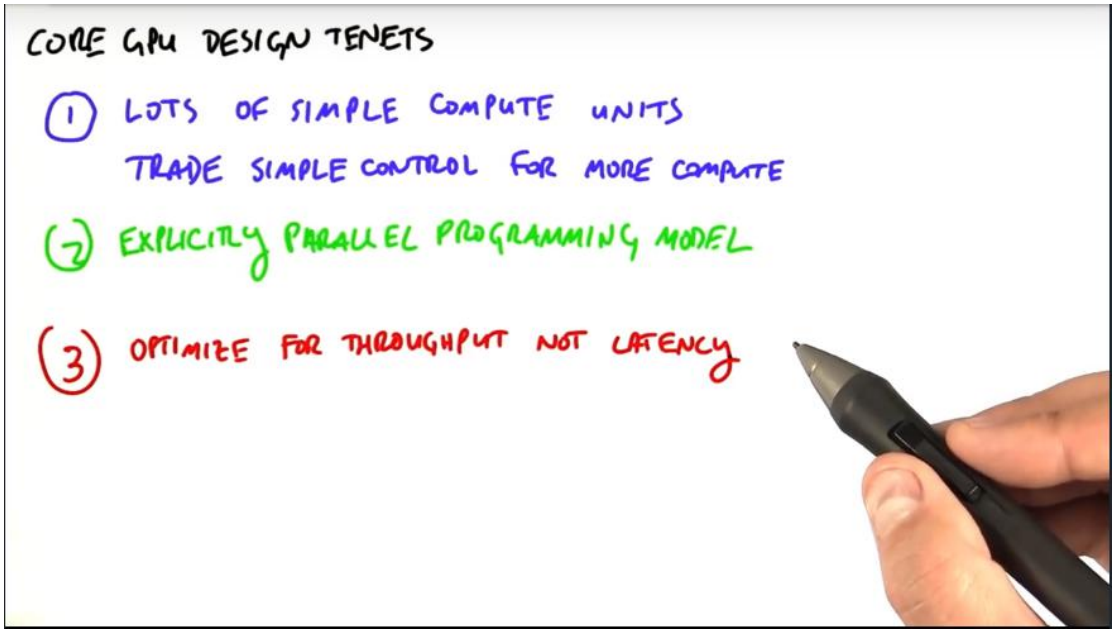


• 时间延迟、吞吐量

CAR ENCY=4500KM/200KM/H = 22.5H、THROUGHPUT=22.5H/2=0.089

LATENCY=4500KM/50KM/H=90H、THROUGHPUT=90H/40=0.45

1.14.Core GPU Design Tenets



1、GPU 有很多简单计算单位，可在一起执行大量计算。

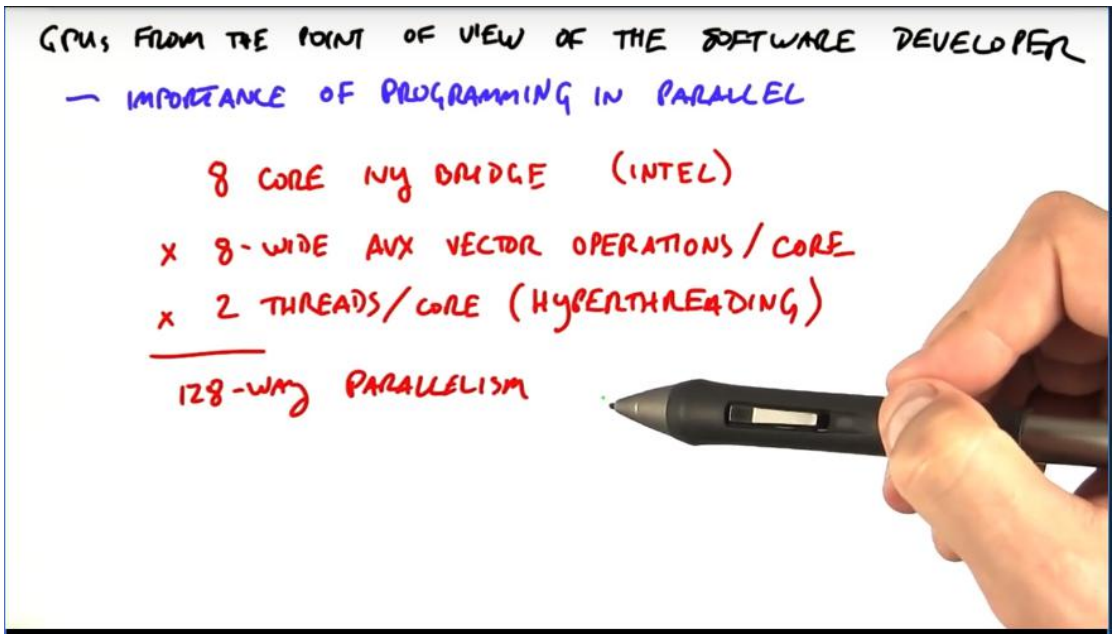
GPU 愿意牺牲控制交换计算，选择更简单的控制复杂性和更多的计算能力。

2、GPU 有一个显示并行编程模型。

3、GPU 为吞吐量进行优化而不是延迟。

GPU 愿意接受任何单一的个体计算增加的延迟，以换取每秒进行更多的计算。

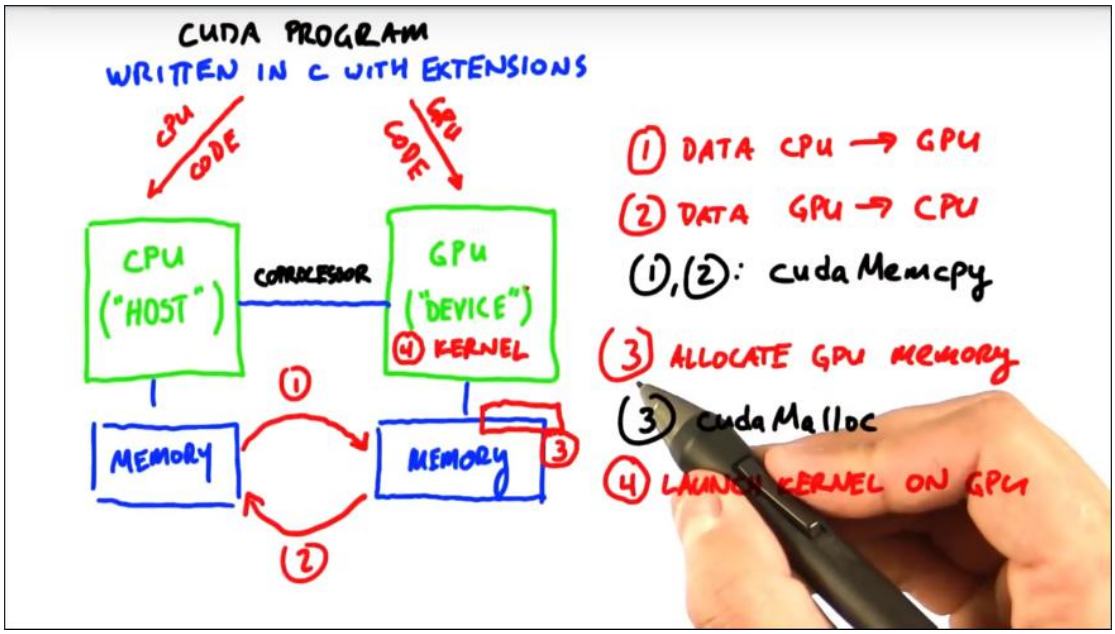
1.15.GPU from the Point of View of the Developer



• 购买一个 8 核的英特尔 Ivy Bridge 处理器，我们看到它有 8 核，每核有 8 宽位 AVX 矢量操作，每核支持两个同时运行的线程，相乘得到 128 路并行度。

1.16.CUDA Program Diagram

• 被称作异构型计算机有两种，



• CUDA 编程模型允许我们用一个程序对两个处理器进行编程。

在 CPU 上运行的是主机，在 GPU 上运行 CUDA 名称是设备。

• CUDA 假设设备 GPU 是主机 CPU 的协同处理器，假设主机和设备有各自分开的内存。CPU 和 GPU 都有各自的 DRAM 形式的专用物理内存。GPU 内存通常是性能很高的内存块。

• CPU 占主导地位，运行主程序向 GPU 发送指示告诉它做什么

1、把 CPU 内存中的数据移到 GPU 内存

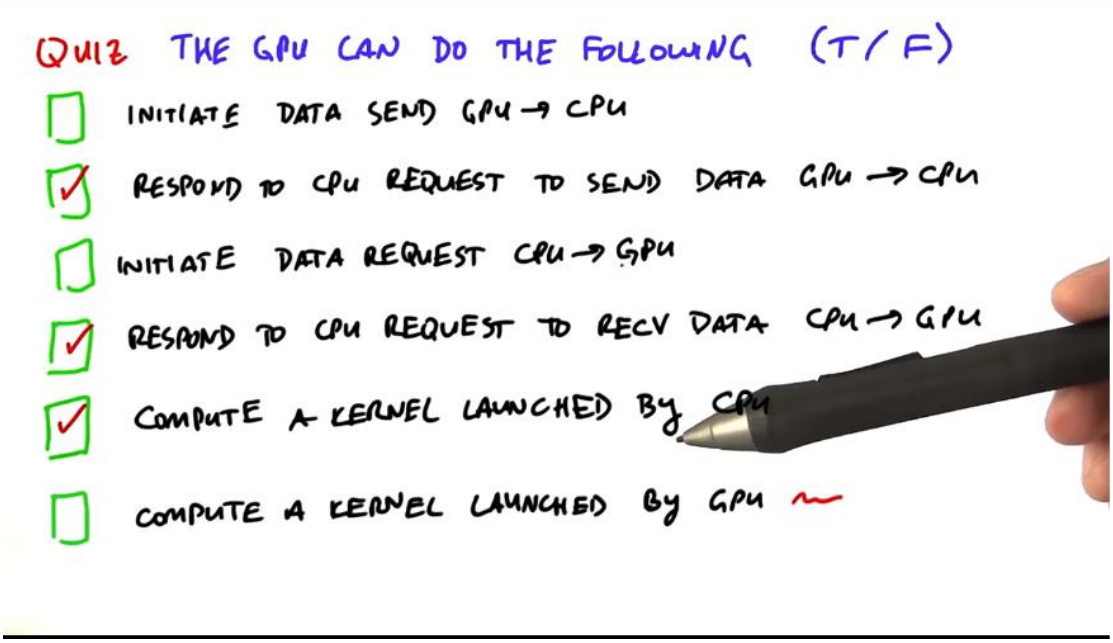
2、把数据从 GPU 移回 CPU（cudaMemcpy）

3、在 GPU 上分配内存（C 语言中是 CMalloc，对应 CUDA 命令是 cudaMalloc）

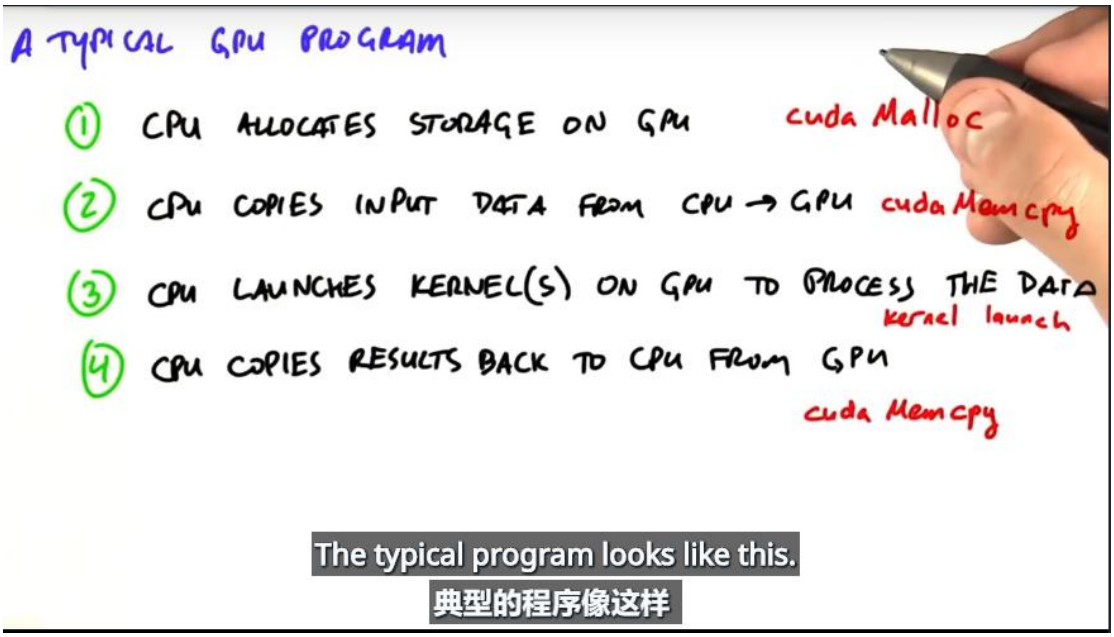
4、在 GPU 上调用以内存方式计算的程序（内核）

• 主机在设备上启动内核

1.17.What Can GPU Do in CUDA

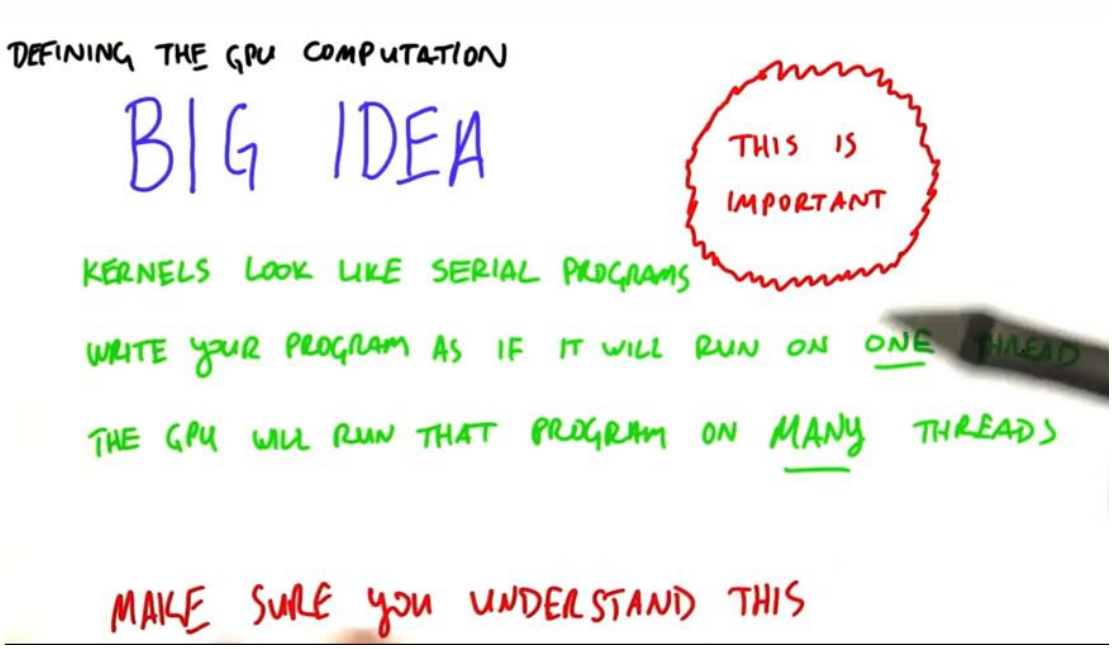


1.18.A CUDA Program



1、CPU 在 GPU 上分配存储
2、CPU 把某个输入数据从 CPU 复制到 GPU
3、CPU 调用某些内核来监视这些在 GPU 上处理这个数据的内核
4、CPU 把结果从 GPU 复制回 CPU
2 与 4 需要在 CPU 和 GPU 间来回传送数据

1.19.Defining the GPU Computation

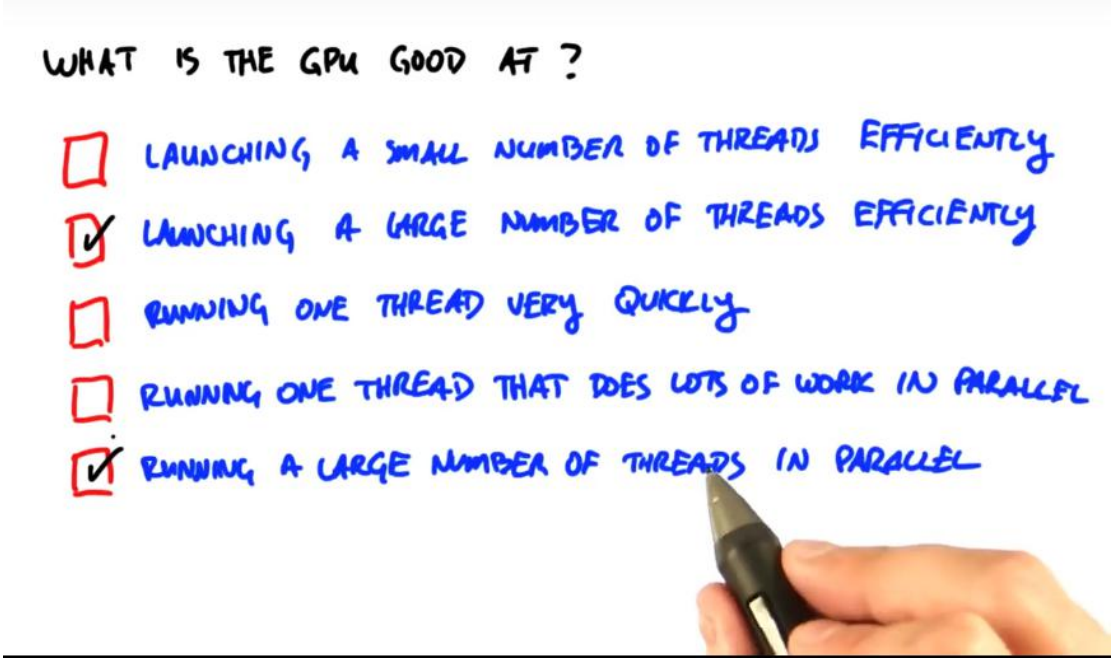


• 分配和转移数据是简单的，我们关注实际发生在 GPU 上的计算。

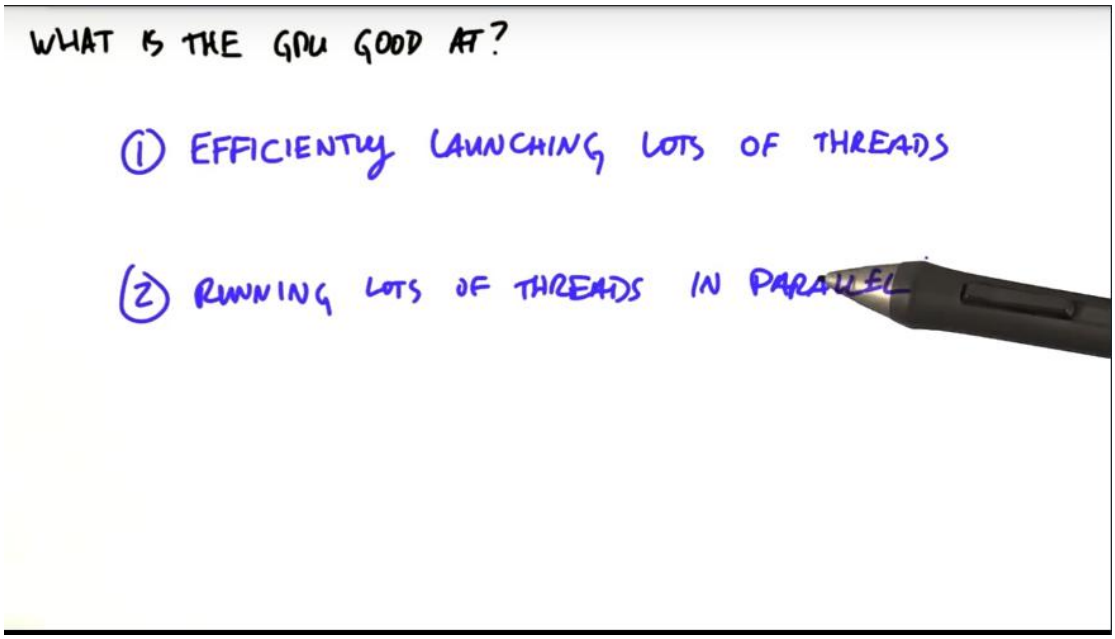
•我们将计算架构为一系列的一个或多个内核，GPU 有很多并行计算单元，当你编写内核时，这些内核需要利用该硬件并行性。

• 编写一个内核，编写看起来像一个串行程序的代码，好像它在一个线程上运行。然后从 CPU 调用内核时，告诉它要启动多少个线程。每一个线程都将运行此内核。

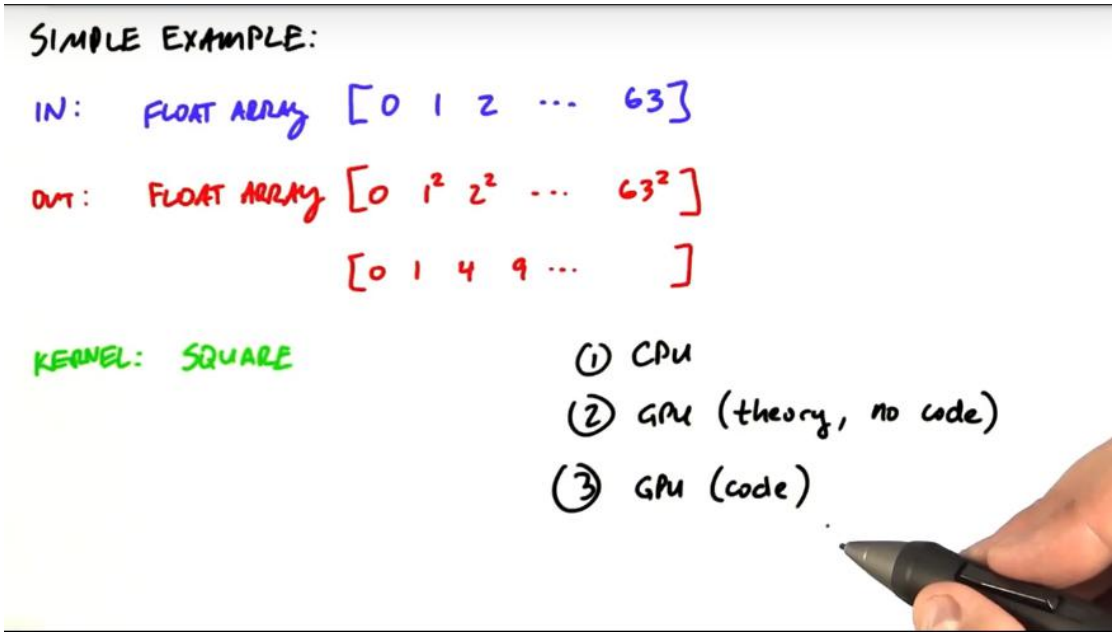
写一个内核，然后告诉 GPU 是 OK 的，当开始运行此内核时，启动 10 万个线程，每个线程将运行此内核代码。



1.20.What the GPU Is Good At

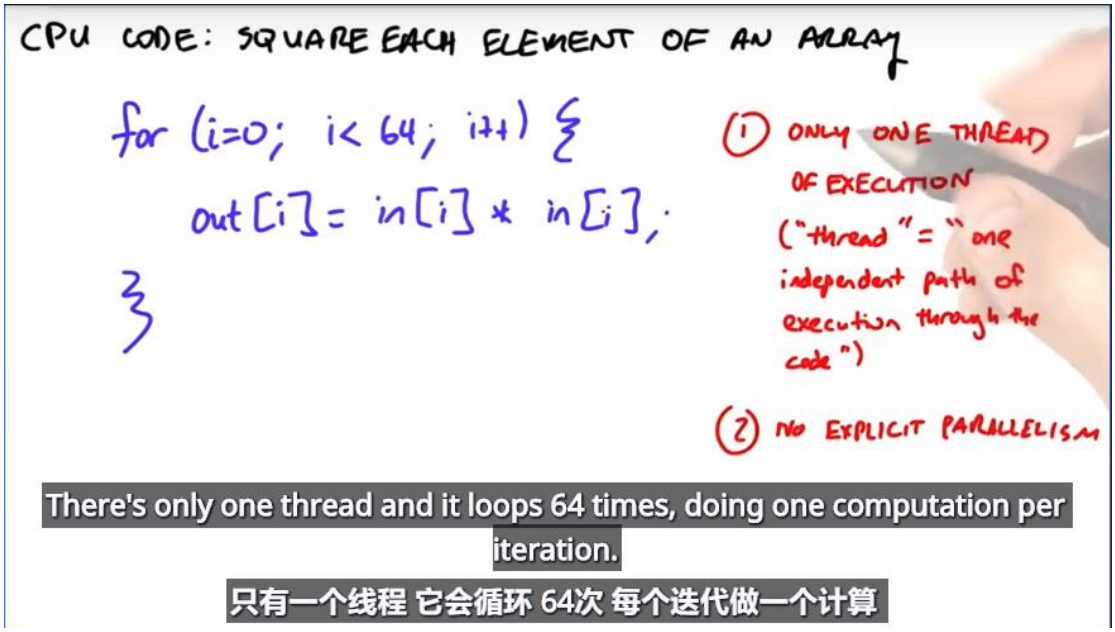


• GPU 擅长一是高效运行多线程、二是同时并行运行多线程



• 分别在 CPU 和 GPU 上运行该例子

1.21.Squaring A Number on the CPU

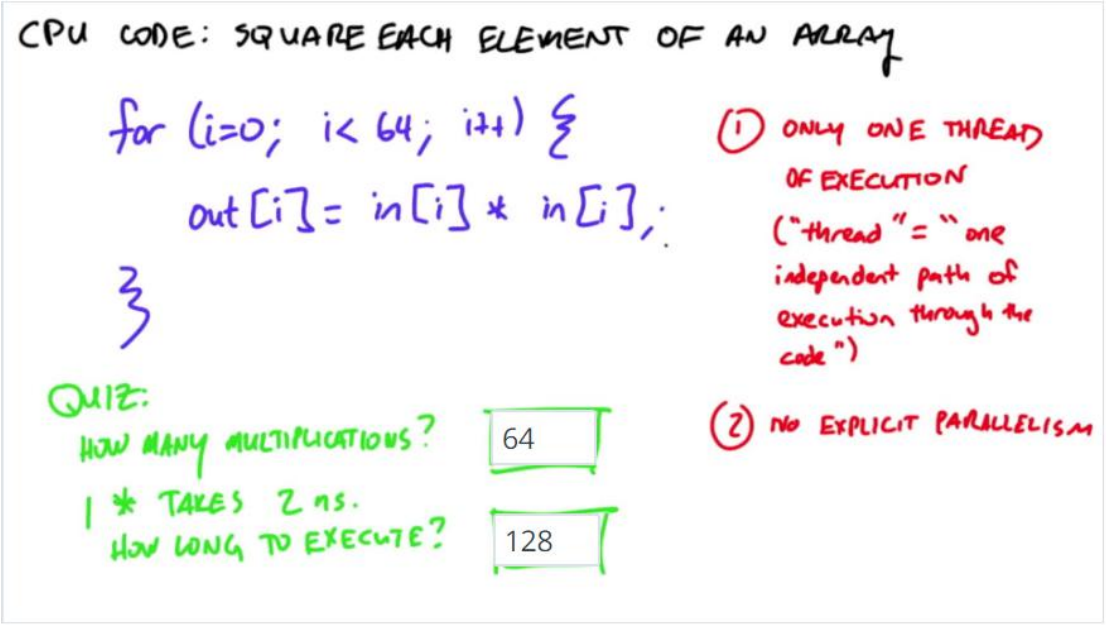


• 两个注意点

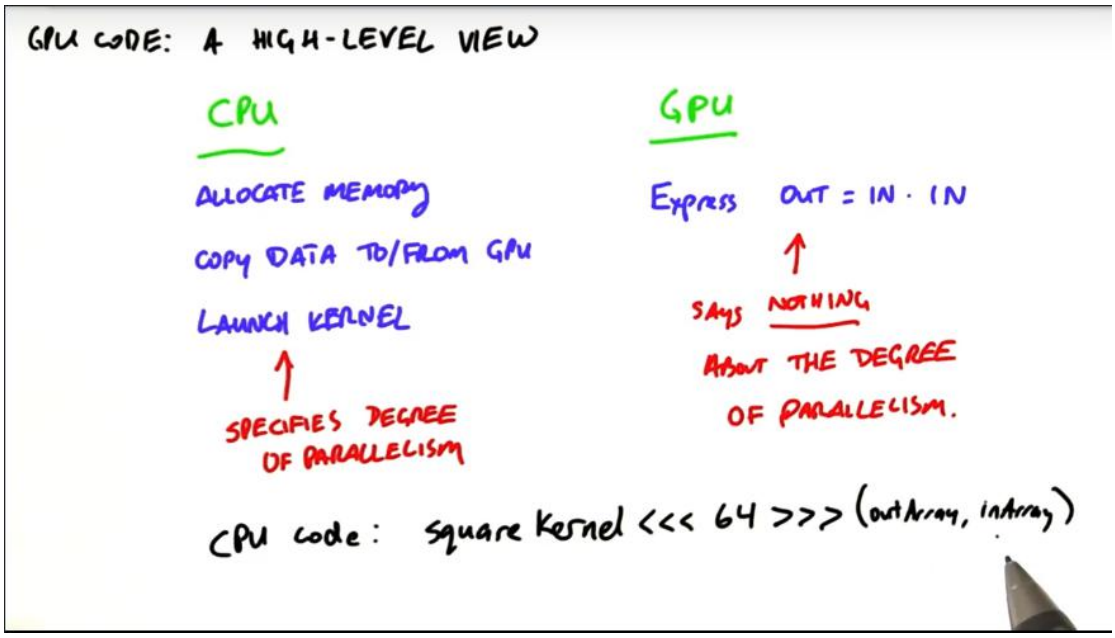
1、我们只有一个线程的执行，而且线程显示遍历所有它的输入。这里将线程定义为代码的一个独立执行路劲。

2、此代码没有显示并行度，这是串行代码

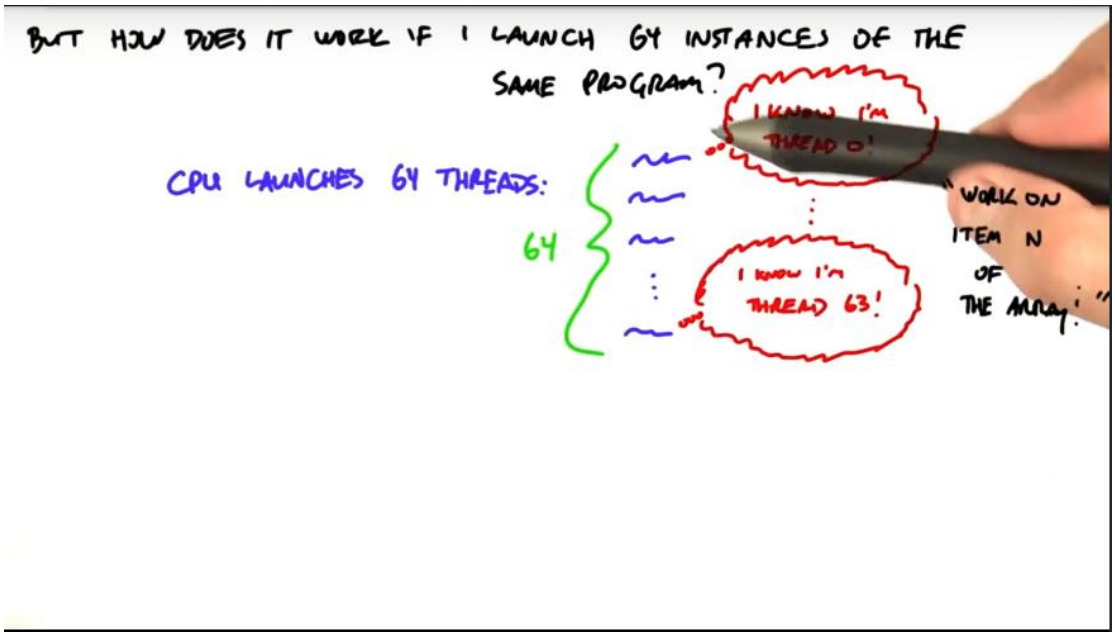
1.22.Calculation Time on the CPU



1.23.GPU Code A High Level View

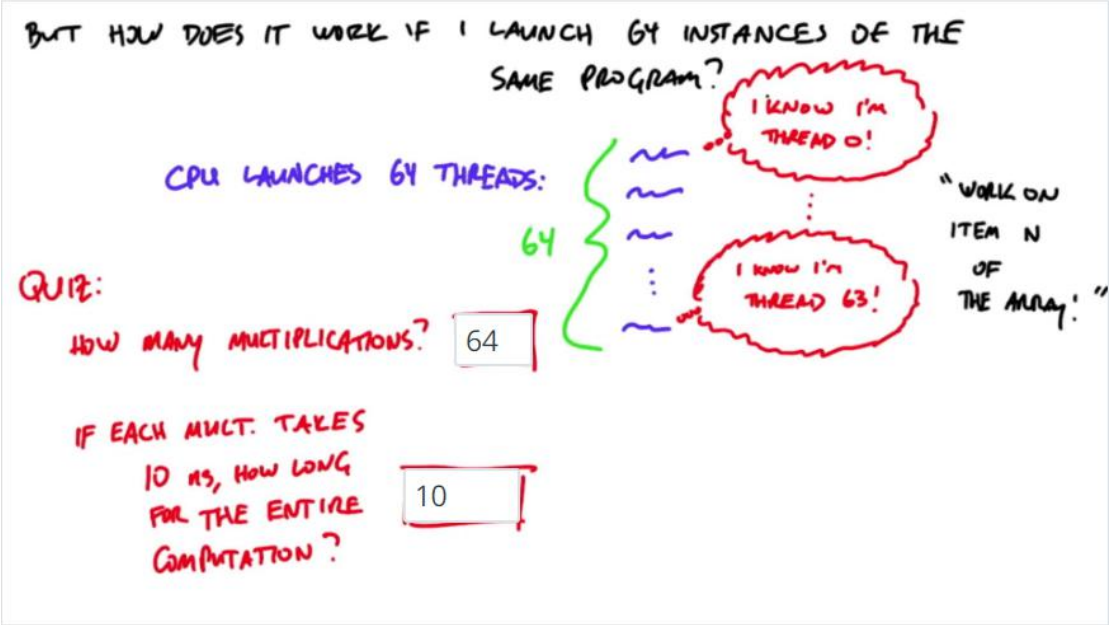


• 为 GPU 编写内核，串行程序，未体现并行度。
• CPU 分配内存、将数据复制到 GPU 和从 GPU 复制数据、**启动内核**（表达线程并行度）
• 在 64 个线程启动一个称为平方的内核，64 个内核实例中的每一个将执行 64 次平方运算中的其中一次。



• 启动的 64 个线程，每个线程知道自己是哪个，把它称为线程索引。分配线程 n 处理数组的第 n 个元素。
• **总结** 你编写你的内核，它每次将在一个线程上运行。然后启动许多线程，每个线程独立运行哪个内核。

1.24.Calculation Time on the GPU



1.25.Squaring Numbers Using CUDA Part 1

```
#include <stdio.h>

__global__ void square(float * d_out, float * d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f * f;
}

int main(int argc, char ** argv) {
  const int ARRAY_SIZE = 64;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  // generate the input array on the host
  float h_in[ARRAY_SIZE];
  for (int i = 0; i < ARRAY_SIZE; i++) {
    h_in[i] = float(i);
  }
  float h_out[ARRAY_SIZE];

  // declare GPU memory pointers
  float * d_in;
  float * d_out;
```

```
23
24   // allocate GPU memory
25   cudaMalloc((void **) &d_in, ARRAY_BYTES);
26   cudaMalloc((void **) &d_out, ARRAY_BYTES);
27
28   // transfer the array to the GPU
29   cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpy???);
30
31   // launch the kernel
32   square<<<1, ARRAY_SIZE>>>(d_out, d_in);
33
34   // copy back the result array to the CPU
35   cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpy???);
36
37   // print out the resulting array
38   for (int i = 0; i < ARRAY_SIZE; i++) {
39     printf("%f", h_out[i]);
40     printf(((i % 4) != 3) ? "\t" : "\n");
41   }
42
43   // free GPU memory allocation
44   cudaFree(d_in);
45   cudaFree(d_out);
46
47   return 0;
48 }
```

RUN CODE unsubmitted

```
$
$
$ less square.cu
$ nvcc -o square square.cu
$ ./square
0.000000        1.000000        4.000000        9.000000
16.000000       25.000000       36.000000       49.000000
64.000000       81.000000       100.000000      121.000000
144.000000      169.000000      196.000000      225.000000
256.000000      289.000000      324.000000      361.000000
400.000000      441.000000      484.000000      529.000000
576.000000      625.000000      676.000000      729.000000
784.000000      841.000000      900.000000      961.000000
1024.000000     1089.000000     1156.000000     1225.000000
1296.000000     1369.000000     1444.000000     1521.000000
1600.000000     1681.000000     1764.000000     1849.000000
1936.000000     2025.000000     2116.000000     2209.000000
2304.000000     2401.000000     2500.000000     2601.000000
2704.000000     2809.000000     2916.000000     3025.000000
3136.000000     3249.000000     3364.000000     3481.000000
3600.000000     3721.000000     3844.000000     3969.000000
$
```

1.26.Squaring Number Using CUDA Part 2

```
14   float h_in[ARRAY_SIZE];
```

```
21   float * d_in;
22   float * d_out;
```

- 主机 CPU 上的数据以 h_开头，设备 GPU 上的数据以 d_开头。

```
25   cudaMalloc((void **) &d_in, ARRAY_BYTES);
26   cudaMalloc((void **) &d_out, ARRAY_BYTES);
```

- 告诉 CUDA 数据实际在 GPU 上 而不在 CPU 上，看下面两行。

cudaMalloc 分配 GPU 上的数据，声明指针和字节数。

```
29   cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpy???);
```

• 从 CPU 数组 h_in 复制数据到 GPU 的数组 d_in。

cudaMemcpy 四个参数有目标地址、源地址和字节、**转移方向**（CUDA 内存主机到设备、CUDA 内存设备到主机以及 CUDA 内存设备到设备。

1.27.Copy to Host or Copy to Device

```
28  // transfer the array to the GPU
29  cudaMemcpy(d_in, h_in, ARRAY_BYTES, ???);
30
31  // launch the kernel
32  cube<<<1, ARRAY_SIZE>>>(d_out, d_in);
33
34  // copy back the result array to the CPU
35  cudaMemcpy(h_out, d_out, ARRAY_BYTES, ???);
```

1 cudaMemcpyHostToDevice

2 cudaMemcpyDeviceToHost

• Your choices for each of the two boxes are:
cudaMemcpyHostToDevice cudaMemcpyDeviceToHost cudaMemcpyDeviceToDevice

1.28.Squaring Numbers Using CUDA Part 3

```
28  // transfer the array to the GPU
29  cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
30
31  // launch the kernel
32  square<<<1, ARRAY_SIZE>>>(d_out, d_in);
33
34  // copy back the result array to the CPU
35  cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
36
37  // print out the resulting array
38  for (int i = 0; i < ARRAY_SIZE; i++) {
39    printf("%f", h_out[i]);
40    printf(((i % 4) != 3) ? "\t" : "\n");
41  }
```

• **Cuda 启动运算符<<< >>>**

该语句表示对一个有 64 个元素的块启动称作 square 的内核。传递内核参数的是两个指针 d 出和 d 入。CPU 在 GPU 的数据上 64 个线程启动内核的 64 个副本

1.29.Squaring Numbers Using CUDA 4

・内核本身

```
__global__ void square(float *d_out, float *d_in) {
    int idx = threadIdx.x;
    float f = d_in[idx];
    d_out[idx] = f * f;
}
```

・**__global__** 通过它 cuda 知道此代码是一个内核而非 CPU 代码；**Void** 意味着内核不返回值，它将输出写入到参数列表指定的指针；**输出和输入指针**需分配在 GPU 上，否则程序会彻底奔溃。

・CUDA 具有一个内置变量称作线程索引 **threadIDX**，告诉一个块中的每个线程它们的索引。ThreadIDX 是 c 结构，实际由 3 名成员.x、.y 和.z 结构称作 dim3

・启动 64 个线程后，线程的第一个实例 threadIDX.x 将返回零，第二个实例返回 1 到最后一个返回 63

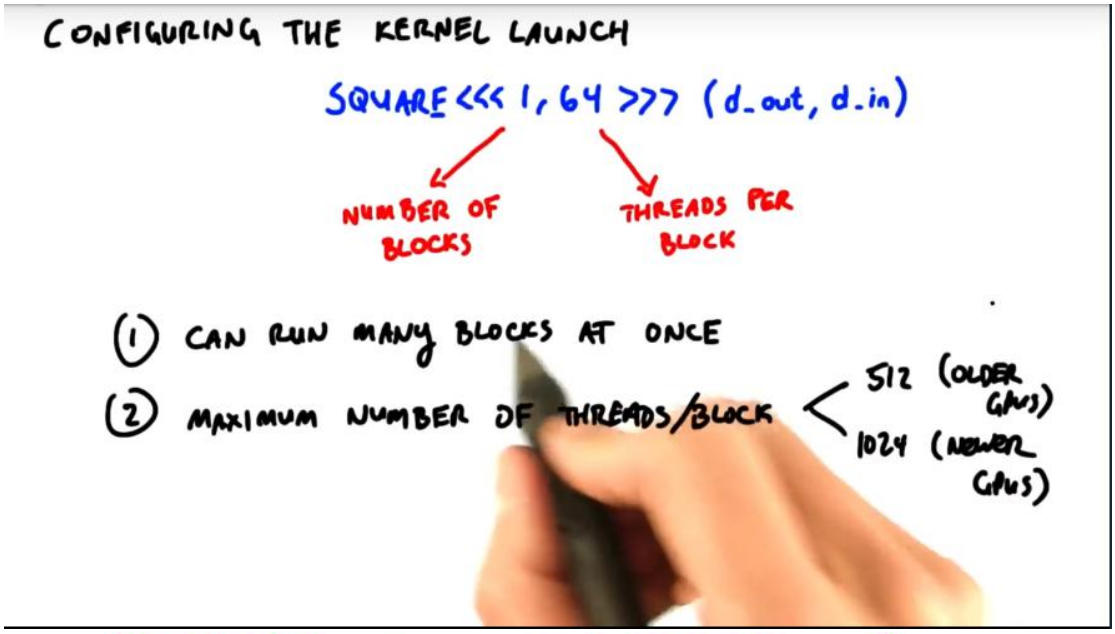
・内核中实际做什么？

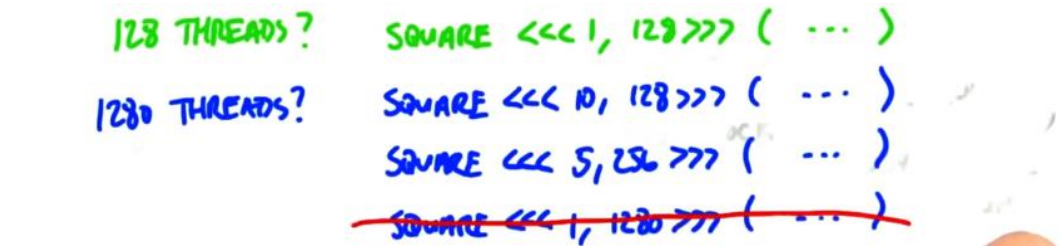
对每一个线程 首先从全局内存读取对应于此线程索引的数组元素，将它存储到浮点型变量 f，然后求 f 的平方，将值写回到全局内存。

1.30.Cubing Numbers Using CUDA

```
#include <stdio.h>

__global__ void cube(float * d_out, float * d_in){
    int idx = threadIdx.x;
    float f = d_in[idx];
    d_out[idx] = f*f*f;
}
int main(int argc, char ** argv) {
    const int ARRAY_SIZE = 96;
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

    // generate the input array on the host
    float h_in[ARRAY_SIZE];
    for (int i = 0; i < ARRAY_SIZE; i++) {
        h_in[i] = float(i);
    }
    float h_out[ARRAY_SIZE];

    // declare GPU memory pointers
    float * d_in;
    float * d_out;

    // allocate GPU memory
    cudaMalloc((void**) &d_in, ARRAY_BYTES);
    cudaMalloc((void**) &d_out, ARRAY_BYTES);

    // transfer the array to the GPU
    cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

    // launch the kernel
    cube<<<1, ARRAY_SIZE>>>(d_out, d_in);

    // copy back the result array to the CPU
    cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

    // print out the resulting array
    for (int i =0; i < ARRAY_SIZE; i++) {
        printf("%f", h_out[i]);
        printf(((i % 4) != 3) ? "\t" : "\n");
    }

    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

1.31.Configuring the Kernel Launch Parameters Part 1

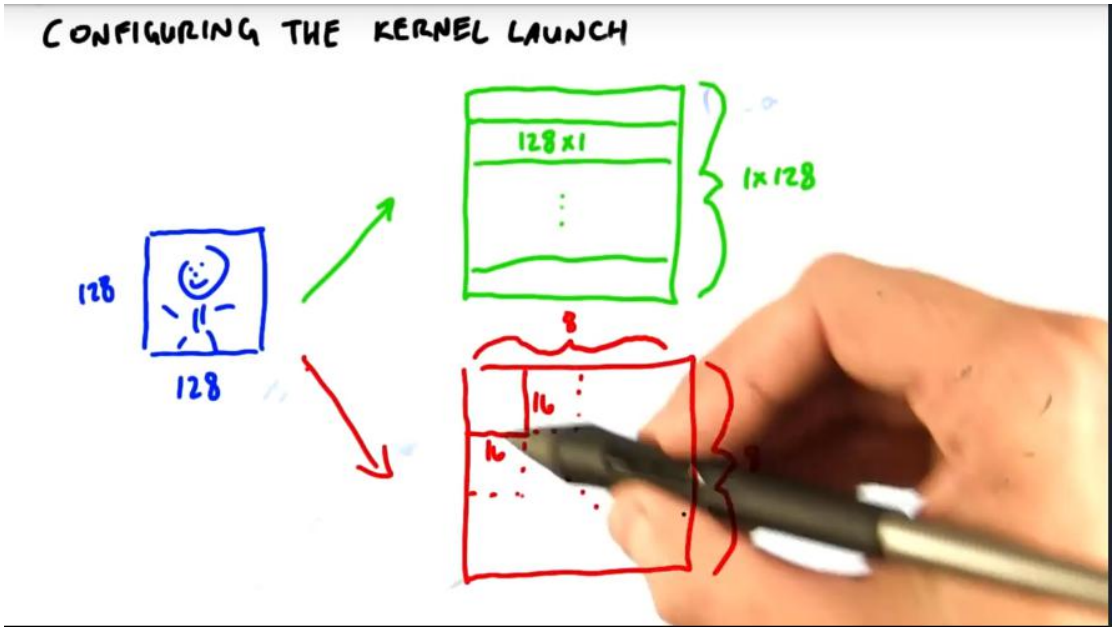




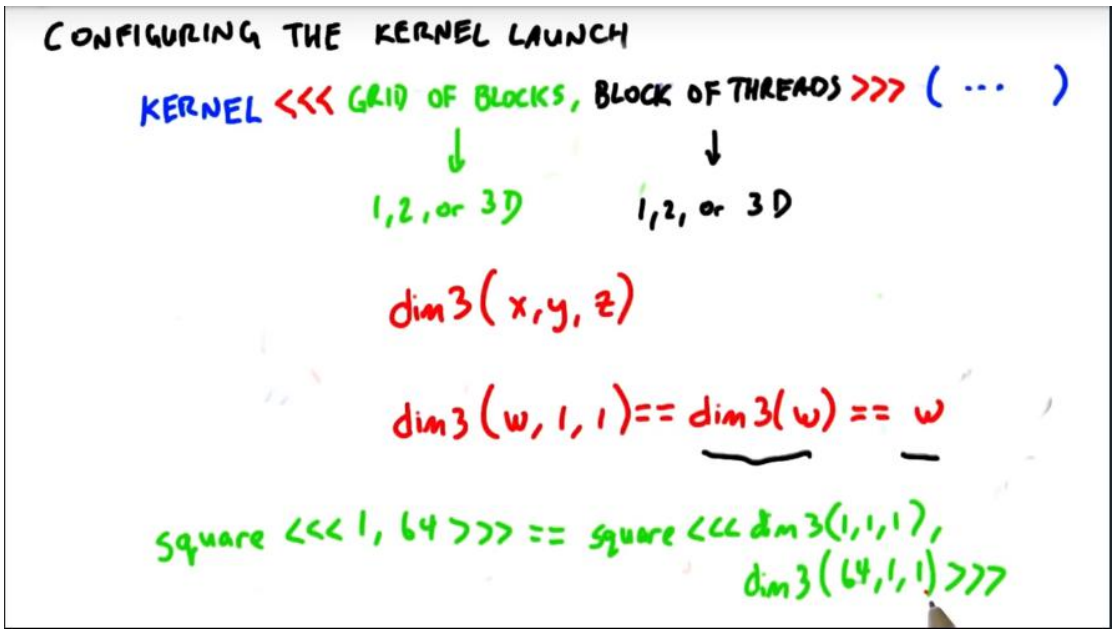
- <<<块数，每个块中线程数>>>

  硬件一有能力在同一时间运行多个块、二每个块有一个它能支持大的线程的最大数量。选择最有意义的线程和线程块的搭配。

- CUDA 还支持 2、3 维的线程块。



• 可以选择 y 维度 128 块，每块在 x 维度是一个有 128 线程的块；或者改为 8x8 的网络块，每块有 16x16 个线程。

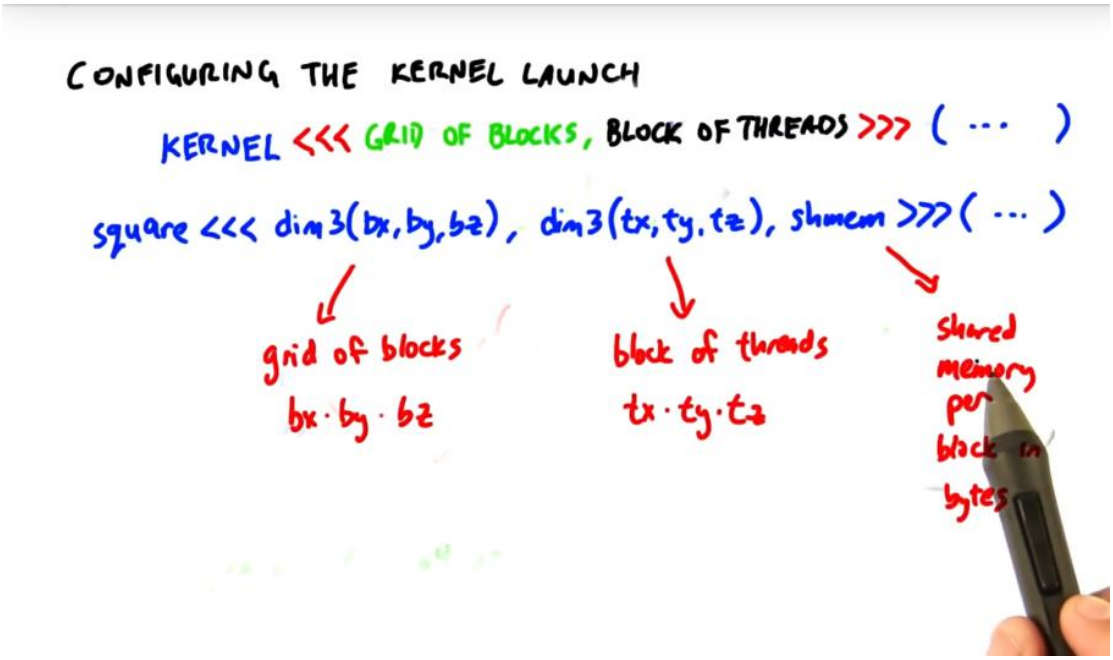


• 块和每块的线程数都可以是一维、二维或三维的。

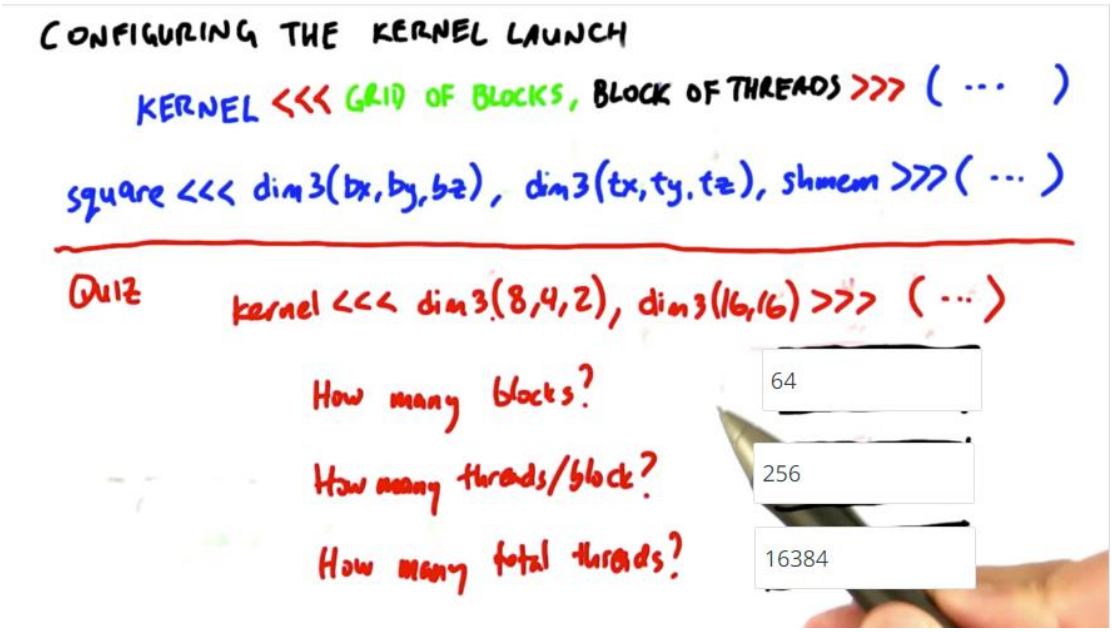
我们借助 **dim3** 结构定义三维，若不指定 y 或 z，他们的默认值为 1

square<<<1,64>>> == square<<<dim3(1,1,1), dim(64,1,1)>>>

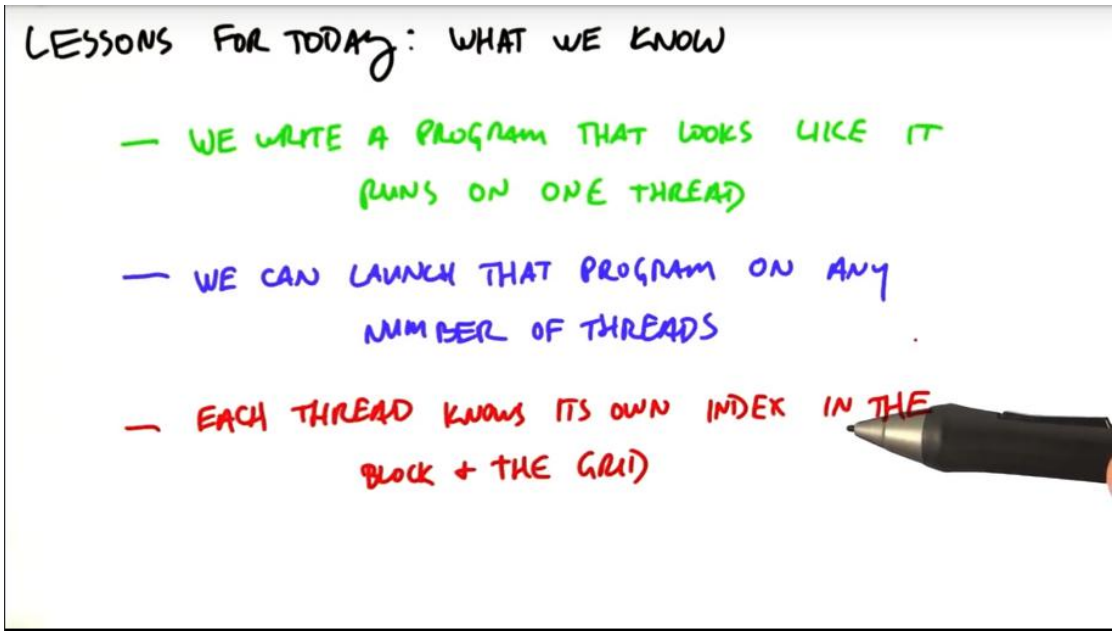
1.32.Configuring the Kernel Launch Parameters 2



• 第三个参数是以字节表示的每个线程块分配的共享内存量。



1.33.What We Know So Far



• 总结

1、当我们写一个内核程序时，该程序看起来就像在一个线程上运行

2、当我们启动程序时，我们从 CPU 代码启动这个内核确定该程序运行的范围，因此程序能够以我们所定义的任意线程数来运行

3、最终在内核程序内，那些线程的每一个都知道它自己的索引以及它自己的线程块，还有这些线程块的网络。

1.34.Map

MAP

- SET OF ELEMENTS TO PROCESS [64 FLOATS]
- FUNCTION TO RUN ON EACH ELEMENT ["SQUARE"]

MAP (ELEMENTS, FUNCTION)

GPUS ARE GOOD AT MAP

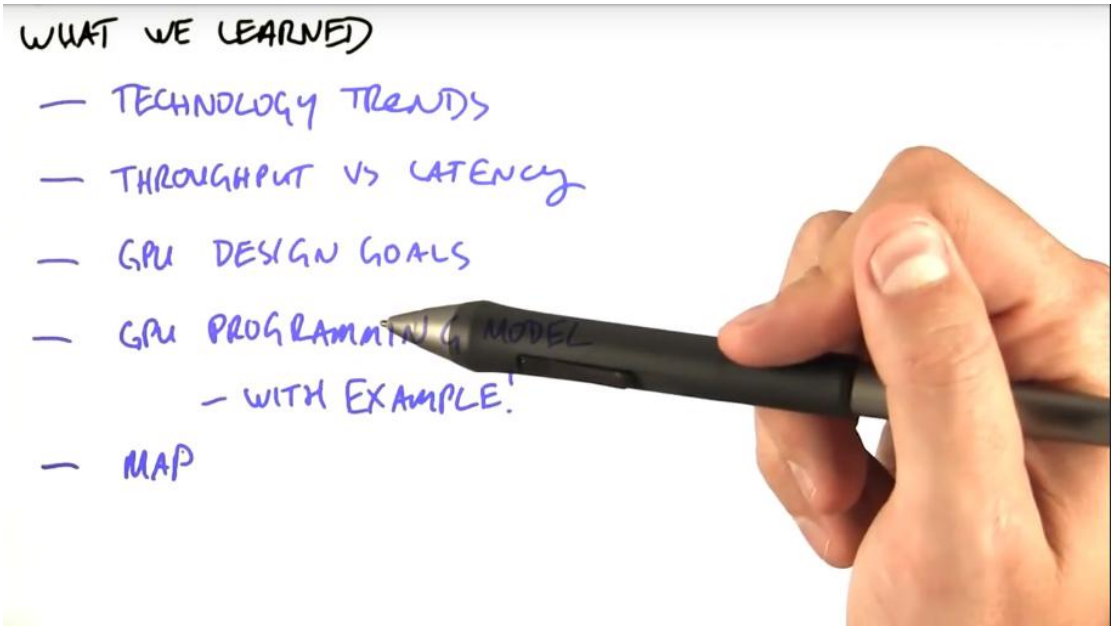
— GPUS HAVE MANY PARALLEL PROCESSORS

— GPUS OPTIMIZE FOR THROUGHPUT

QUIZ CHECK THE PROBLEMS THAT CAN BE SOLVED USING MAP.

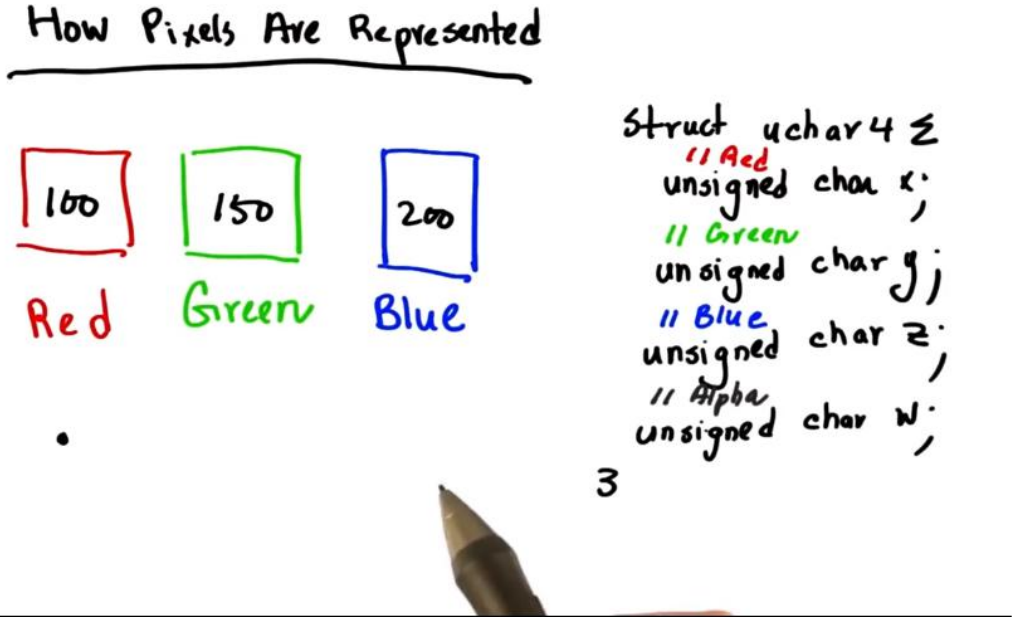
- [ ] SORT AN INPUT ARRAY
- [ ] ADD ONE TO EACH ELEMENT IN AN INPUT ARRAY
- [ ] SUM UP ALL ELEMENTS IN AN INPUT ARRAY
- [ ] COMPUTE THE AVERAGE OF AN INPUT ARRAY

1.35.Summary of Lesson 1



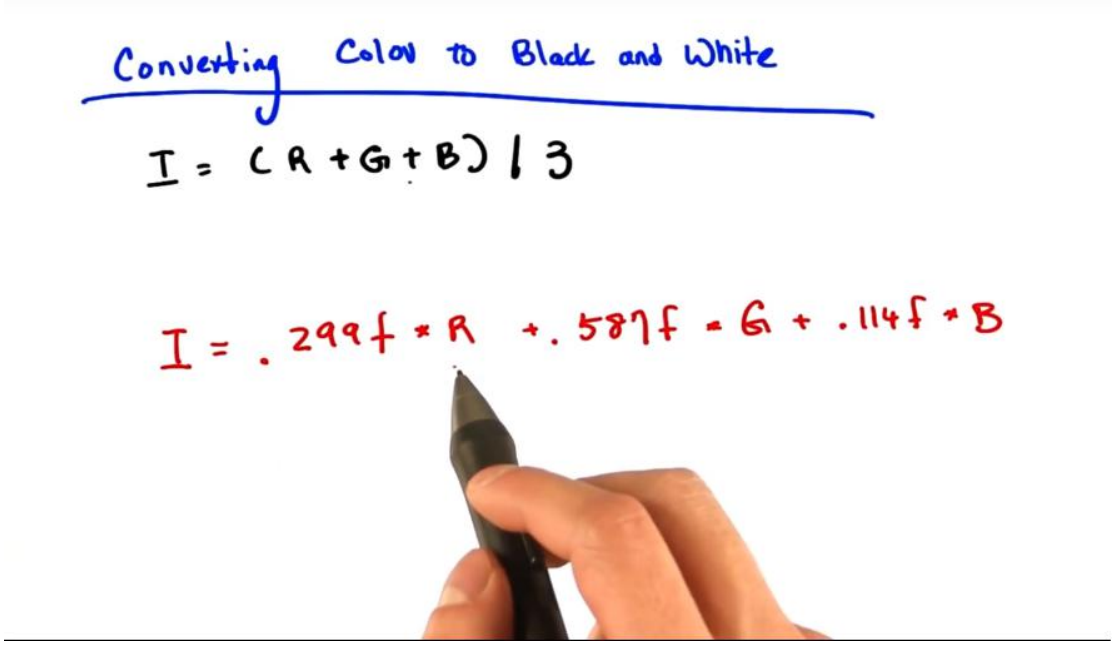
• 使用 CUDA 来输入一个彩色图像然后以灰度输出图像，涉及对该图像中的每个像素进行一次映射操作：把以红、绿和蓝输入的颜色转换为一个单一的亮度值。

1.37.Problem Set 1



• 像素在 CUDA 中如何表示？

每个像素由一个无符号的 char4 结构表示，这种结构由四个无符号的 char 组件，分别叫作 x（红）、y（绿）、z（蓝）和 w（透明）。



• 如何将彩色图像转化为黑白图像。（人眼对绿更敏感）

```
#include "reference_calc.cpp"
#include "utils.h"
#include <stdio.h>

__global__
void rgba_to_greyscale(const uchar4* const rgbaImage,
                       unsigned char* const greyImage,
                       int numRows, int numCols)
{
  //TODO
  //Fill in the kernel to convert from color to greyscale
  //the mapping from components of a uchar4 to RGBA is:
  // .x -> R ; .y -> G ; .z -> B ; .w -> A
  //
  //The output (greyImage) at each pixel should be the result of
  //applying the formula: output = .299f * R + .587f * G + .114f * B;
  //Note: We will be ignoring the alpha channel for this conversion
  int x = threadIdx.x;
  int y = threadIdx.y;

  uchar4 rgb = rgbaImage(x*numCols+y);
  unsigned char R = rgb.x;
  unsigned char G = rgb.g;
  unsigned char B = rgb.b;

  greyImage[x*numCols+y] = .299f * R + .587f * G + .114f * B;

  float greyImage = .299f * rgbaImage.x + .587f * rgbaImage.y + .114f * rgbaImage.z;
  //First create a mapping from the 2D block and grid locations
  //to an absolute 2D location in the image, then use that to
  //calculate a 1D offset
}

void your_rgba_to_greyscale(const uchar4 * const h_rgbaImage, uchar4 * const d_rgbaImage,
                            unsigned char* const d_greyImage, size_t numRows, size_t numCols)
{
  //You must fill in the correct sizes for the blockSize and gridSize
  //currently only one block with one thread is being launched
  const dim3 blockSize(1, 1, 1);  //TODO
  const dim3 gridSize( numRows, numCols, 1);  //TODO
  rgba_to_greyscale<<<gridSize, blockSize>>>(d_rgbaImage, d_greyImage, numRows, numCols);

  cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
}

```