上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

大规模工程计算

第三课

异构加速器和访存架构

王一超

2023年10月11日

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

报告提纲

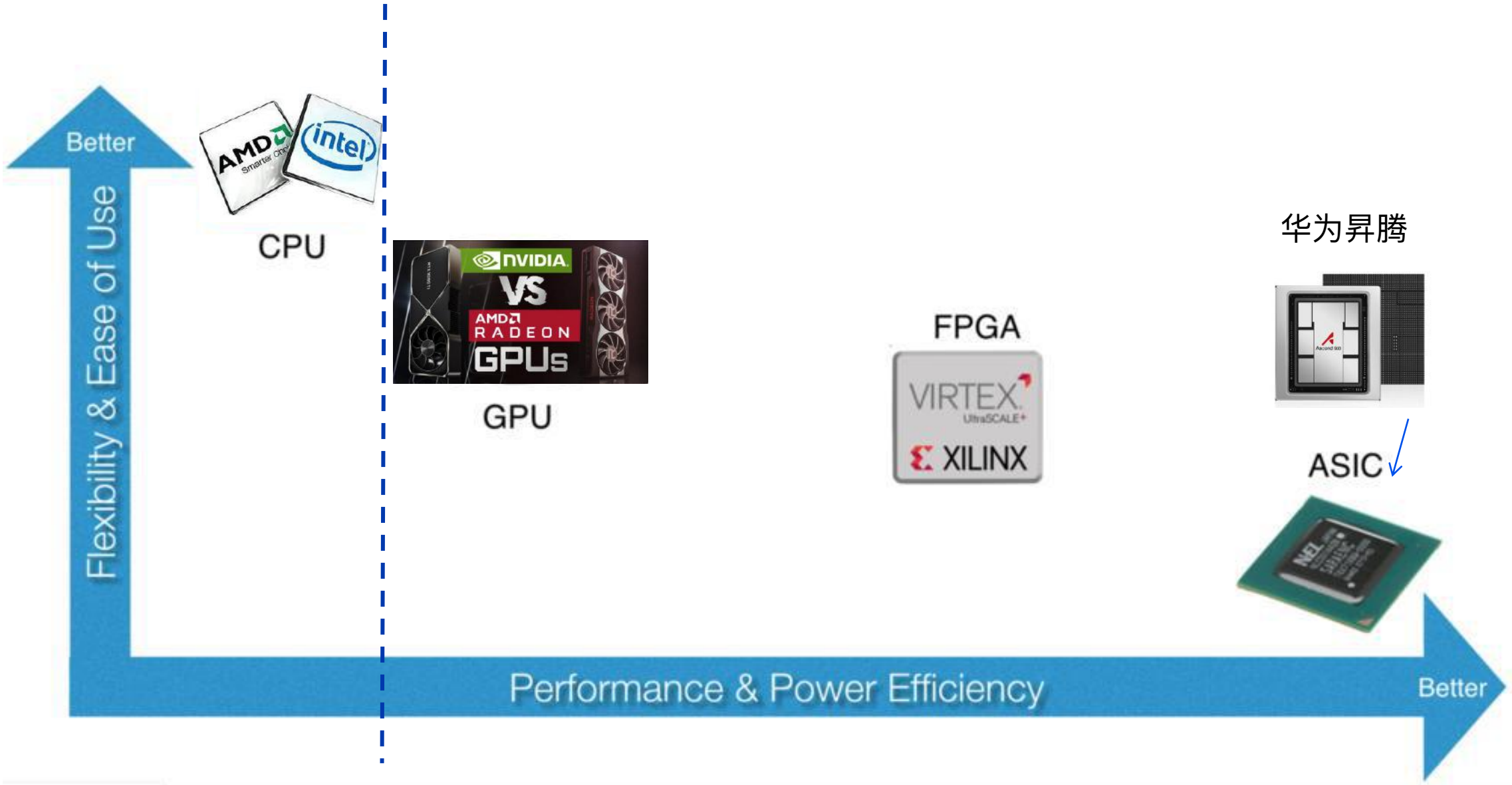
异构加速器

层次化存储

文件系统与数据管理

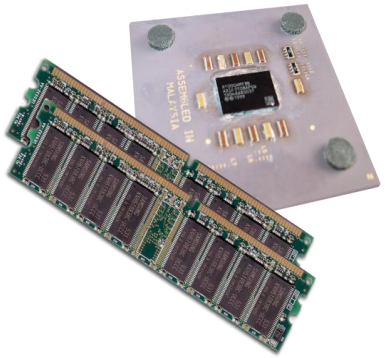
“交我算”课堂实践

Heterogeneous Computing Roadmap



Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

Heterogeneous Computing

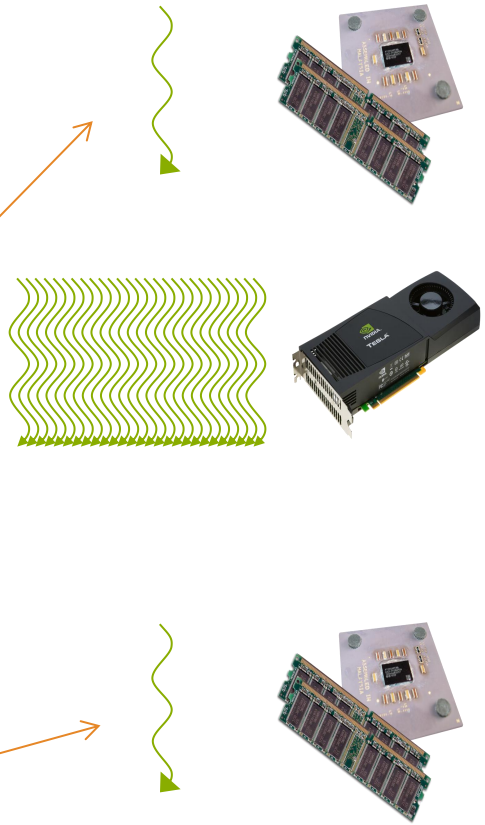
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

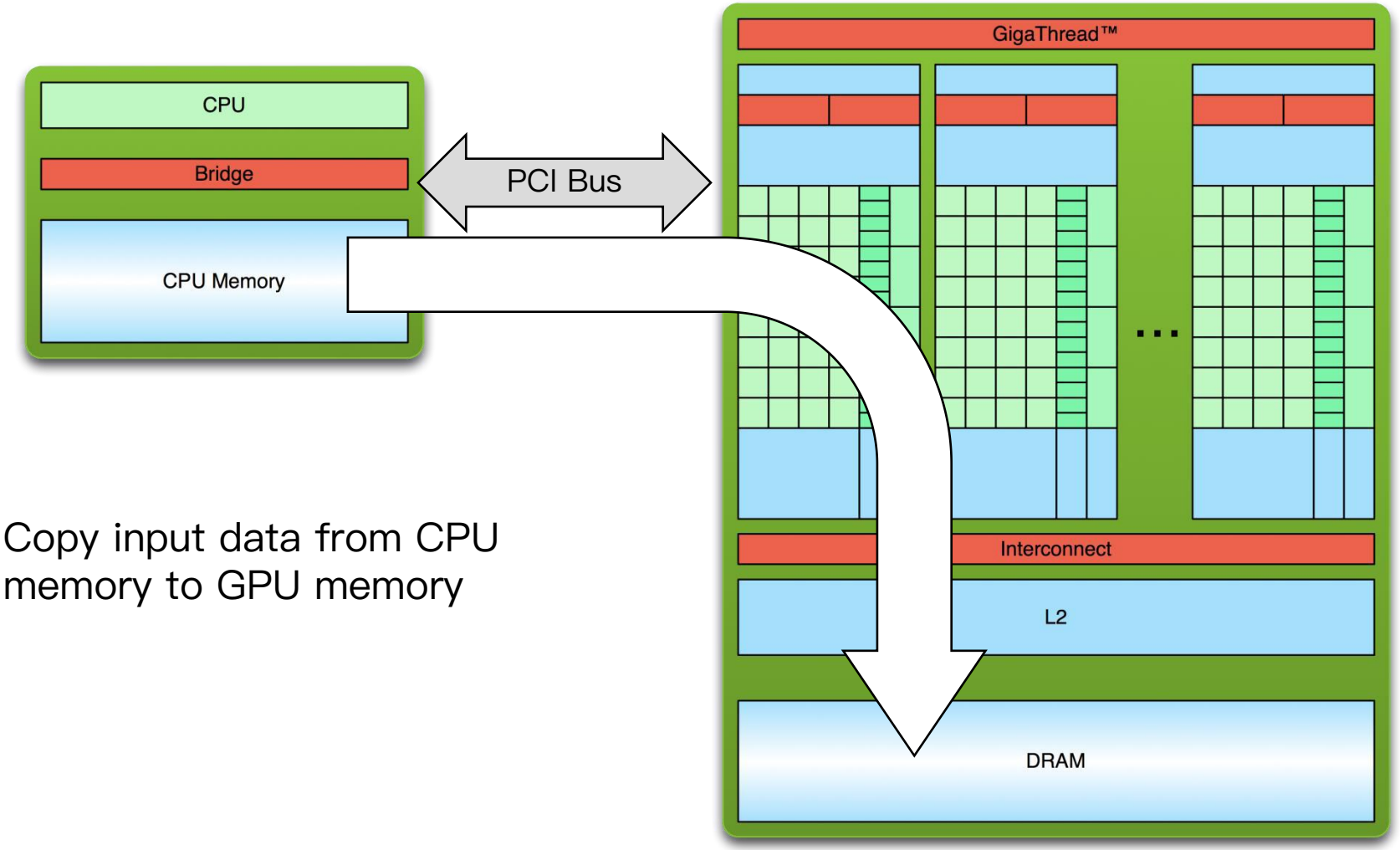
serial code

parallel code

serial code

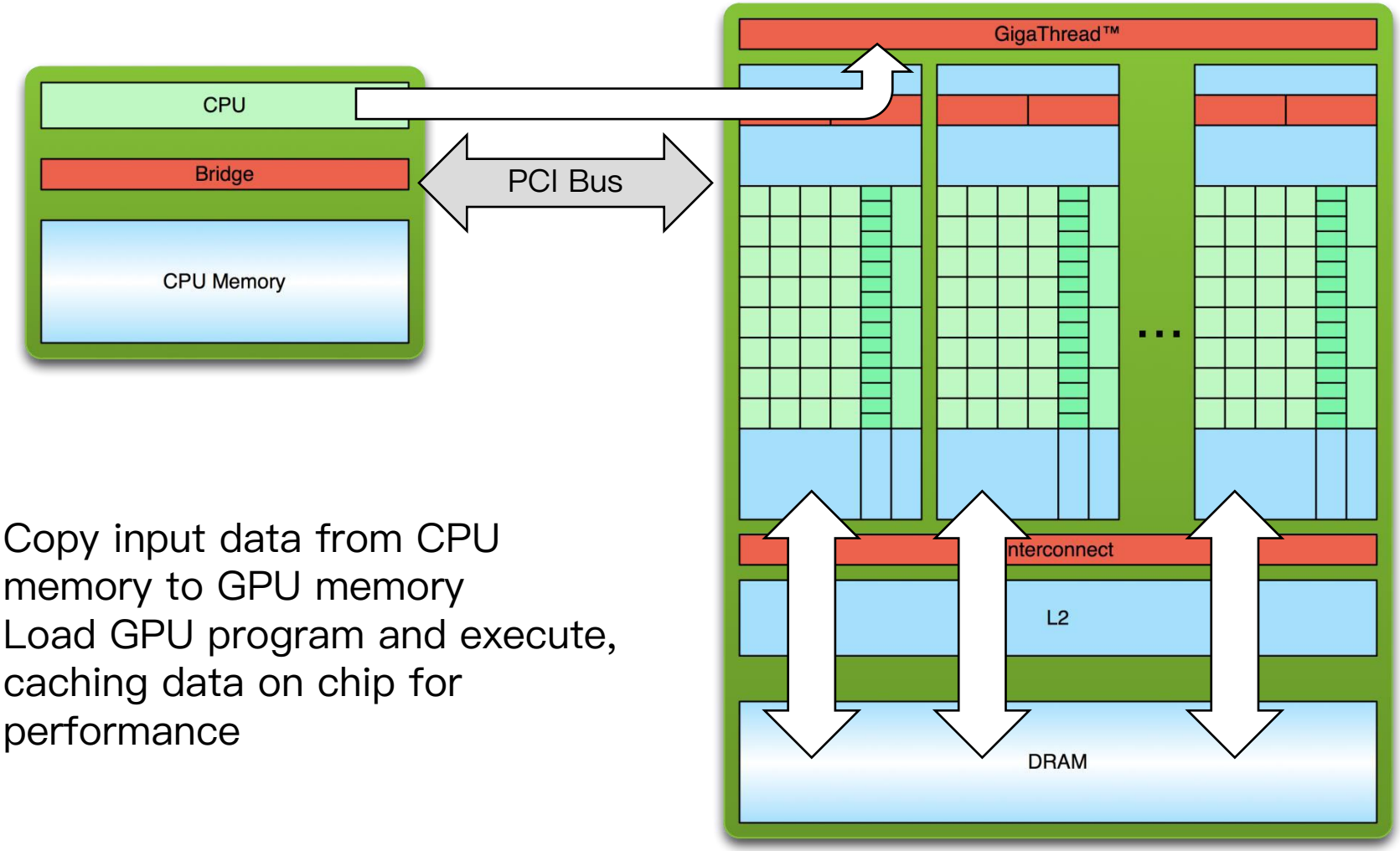


Simple Processing Flow



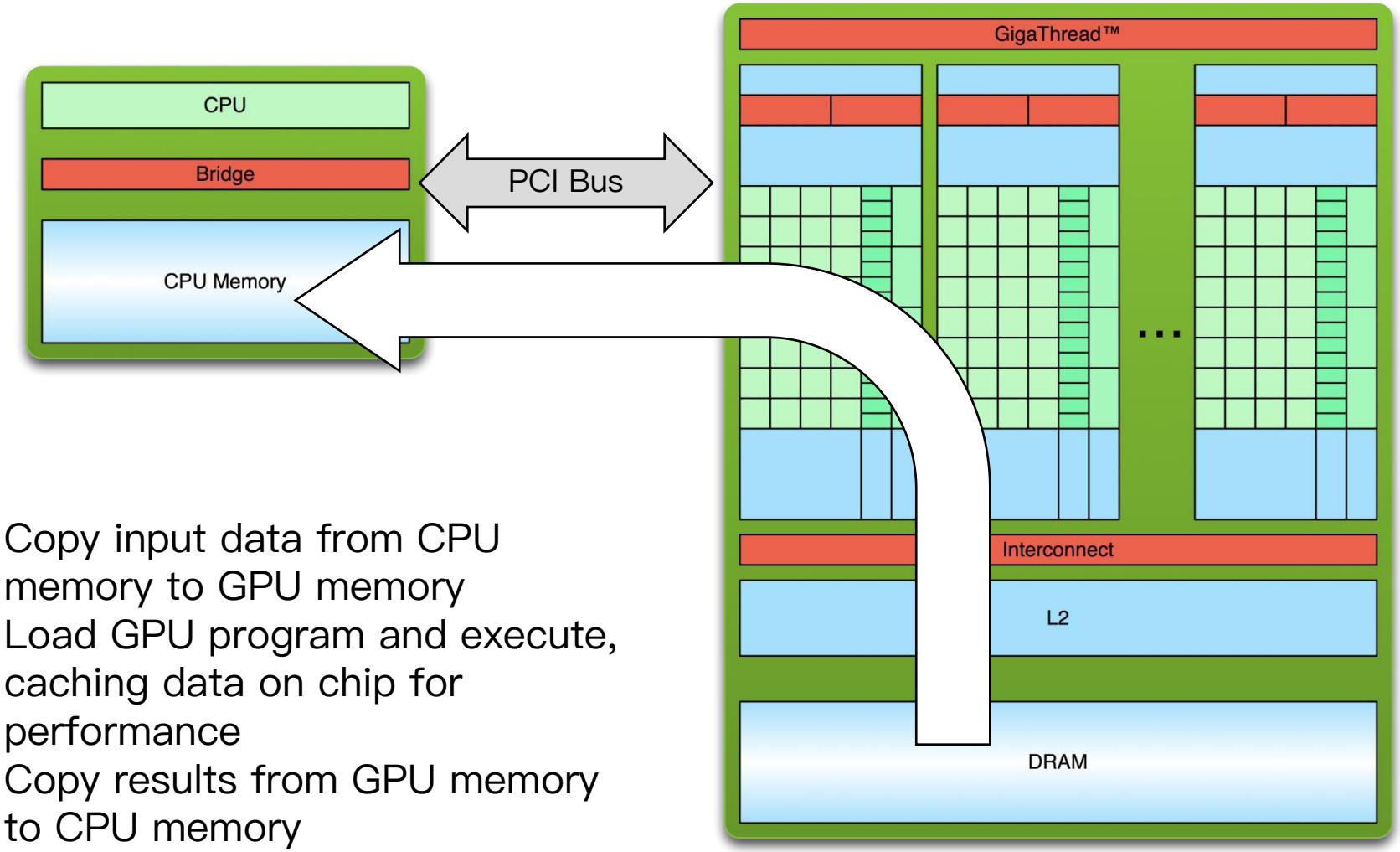
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.
cu
$ a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

- Two new syntactic elements…

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a "kernel launch"
  - We'll return to the parameters (1,1) in a moment

- That's all that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void){
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

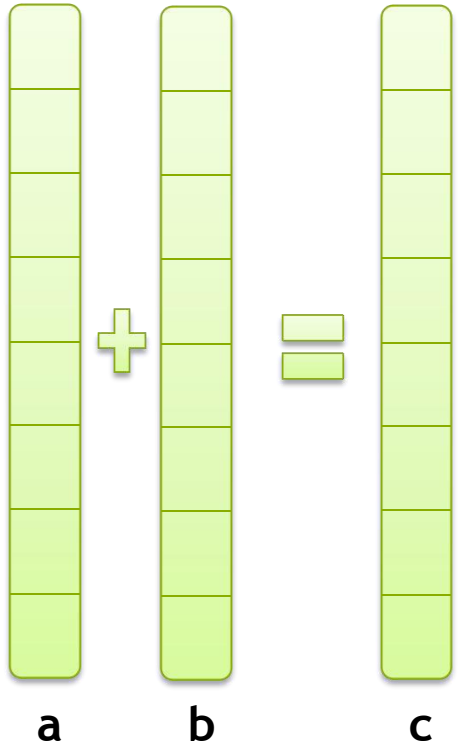
Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

- `mykernel()` does nothing, somewhat anticlimactic!

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

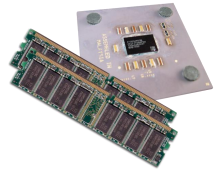
```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- Let's take a look at main()...

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b,
c

        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Graphics card	Nvidia A100 PCIe 40GB	Nvidia GeForce RTX 4090 Ti
Market (main)	Desktop	Desktop
Release date	Q2 2020	Q3 2023
Model number	A100 PCIe, Tesla A100 40GB	AD102-400-A1
GPU name	GA100	AD102
Architecture	Ampere	Ada Lovelace
Generation	Tesla Axx	GeForce 40
Lithography	7 nm	5 nm
Transistors	54,200,000,000	76,300,000,000
Bus interface	PCIe 4.0 x16	PCIe 4.0 x16
GPU base clock	765 MHz	2,355 MHz
GPU boost clock	1,410 MHz	2,625 MHz
Memory frequency	1,215 MHz	1,500 MHz
Effective memory speed	2.4 Gbps	24 Gbps
Memory size	40 GB	24 GB
Memory type	HBM2e	GDDR6X
Memory bus	5120 bit	384 bit
Memory bandwidth	1,555.0 GB/s	1,152.0 GB/s
TDP	250 W	600 W
Suggested PSU	650W ATX Power Supply #ad	1100W ATX Power Supply #ad
Multicard technology	NVLink	-

Graphics card	Nvidia A100 PCIe 40GB	Nvidia GeForce RTX 4090 Ti
Cores (compute units, SM, SMX)	108	142
Shading units (cuda cores)	6,912	18,176
TMUs	432	568
ROPs	160	192
Tensor cores	432	568
Cache memory	40 MB	96 MB
Pixel fillrate	225.6 GP/s	504.0 GP/s
Texture fillrate	609.1 GT/s	1,491.0 GT/s
Performance FP16 (half)	78.0 TFLOPS	95.4 TFLOPS
Performance FP32 (float)	19.5 TFLOPS	95.4 TFLOPS
Performance FP64 (double)	9.7 TFLOPS	1.5 TFLOPS

	H100 SXM	H800 SXM	H100 PCIe	H800 PCIe
FP64	30 TFLOPS	1 TFLOPS	24 TFLOPS	0.8 TFLOP
FP64 Tensor Core	60 TFLOPS	1 TFLOPS	48 TFLOPS	0.8 TFLOP
FP32	60 TFLOPS	67 TFLOPS	48 TFLOPS	51 TFLOPS
TF32 Tensor Core	1000 TFLOPS*	989 TFLOPS*	800 TFLOPS*	756 TFLOPS*
BFLOAT16 Tensor Core	2000 TFLOPS*	1979 TFLOPS*	1600 TFLOPS*	1513 TFLOPS*
FP16 Tensor Core	2000 TFLOPS*	1979 TFLOPS*	1600 TFLOPS*	1513 TFLOPS*
FP8 Tensor Core	4000 TFLOPS*	3958 TFLOPS*	3200 TFLOPS*	3026 TFLOPS*
INT8 Tensor Core	4000 TOPS*	3958 TFLOPS*	3200 TOPS*	3026 TFLOPS*
GPU 显存	80GB	80GB	80GB	80GB
GPU 显存带宽	3TB/s	3.35TB/s	2TB/s	2TB/s
解码器	7 NVDEC 7 JPEG	7 NVDEC 7 JPEG	7 NVDEC 7 JPEG	7 NVDEC 7 JPEG
最大热设计功耗（TDP）	700 瓦	700 瓦	350瓦	350瓦
多实例 GPU	最多 7 个 MIG，每个 10GB	最多 7 个 MIG，每个 10GB	最多 7 个 MIG，每个 10GB	最多 7 个 MIG，每个 10GB
外形规格	SXM	SXM	PCIe 双插槽 风冷式	PCIe 双插槽 风冷式
互连技术	NVLink： 900GB/s PCIe 5.0： 128GB/s	NVLink™: 400GB/s PCIe 5.0: 128GB/s	NVLink： 600GB/s PCIe 5.0： 128GB/s	NVLink: 400GB/s PCIe 5.0: 128GB/s
服务器选项	搭载 4 个或 8 个GPU 的 NVIDIA HGX™ H100 合作伙伴认证系统和NVIDIA 认证系统（NVIDIA-Certified Systems™） 搭载 8 个 GPU 的NVIDIA DGX ™ H100	搭载 8 个 GPU 的 NVIDIA HGX ™ H800 合作伙伴和NVIDIA 认证系统和搭载 8 个GPU 的 NVIDIA DGX™ H800 NVIDIA 认证系统 (NVIDIACertified Systems™)	搭载 1 至 8 个GPU 的合作伙伴认证系统及NVIDIA 认证系统	搭载 1 至 8 个 GPU 的合作伙伴和 NVIDIA 认证系统

Slurm作业运行队列

- 用户提交作业脚本时需要指定作业运行的队列

集群	节点类型	节点数	单节点核数	单节点内存	队列	允许单作业核数	可否共享	最长运行时间
π2.0	CPU节点（x86）	656个	40核	192G	small	1–20	可共享	7天
					cpu	40–24000	需独占	7天
					debug	测试节点	可共享	20分钟
	CPU节点（大内存）	3个	80核	3T	huge	6–80	可共享	2天
			192核	6T	192c6t	48–192	可共享	2天
AI平台	GPU节点	8个，每节点配16张V100卡	96核	1.45T	dgx2	最高CPU配比为1:6，GPU卡数为1–128	可共享	7天
ARM平台	CPU节点（ARM）	100个	128核	256G	arm128c256g	1–12800	可共享	3天
					debugarm	测试节点	可共享	20分钟
思源一号	CPU节点	936个	64核	512G	64c512g	1–60000	可共享	7天
					debug64c512g	测试节点	可共享	1小时
	GPU节点	23个，每节点配4张A100卡	64核	160G	a100	最高CPU配比为1:16，GPU卡数为1–92	可共享	7天

*相关内容参考：https://docs.hpc.sjtu.edu.cn/job/slurm.html

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

报告提纲

异构加速器

层次化存储

文件系统与数据管理

“交我算”课堂实践

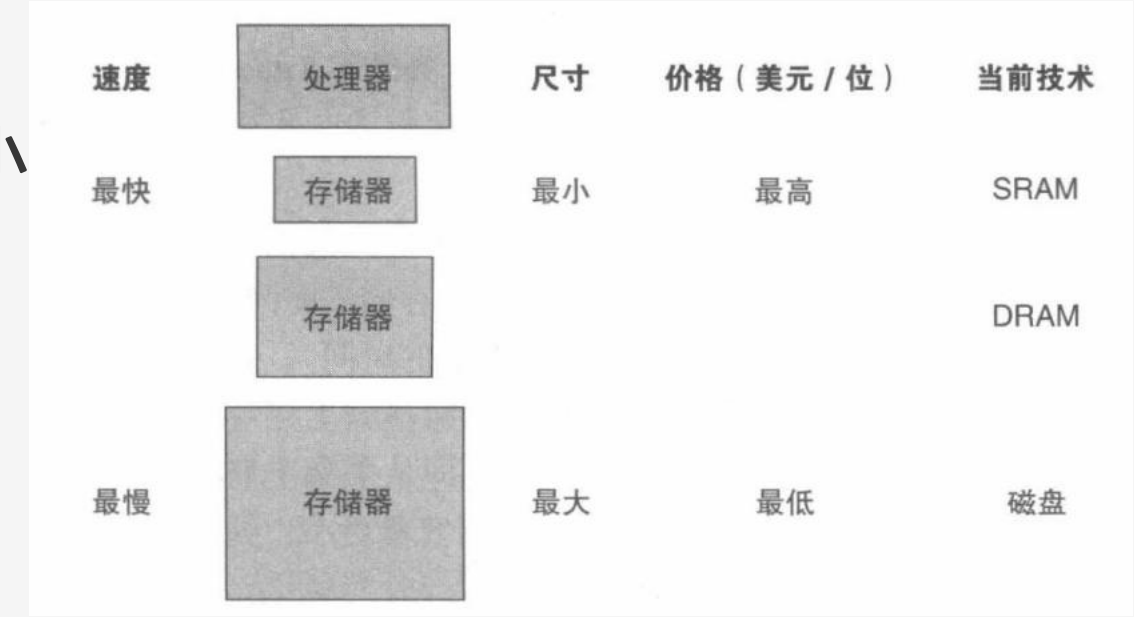
局部性原理

- 在任意一段时间内，程序都只会访问地址空间中的一小部分内容
- 时间局部性
  - 如果某个数据被访问，那么在不久的将来，它很可能再次被访问
  - 例：循环体中的指令与归纳变量（`for(i=...)`）
- 空间局部性
  - 如果某个数据被访问，那么与它地址相邻的数据可能很快也将被访问
  - 例：指令的顺序访问与数组中的数据遍历

利用局部性建立存储层次结构

存储层次结构（memory hierarchy）

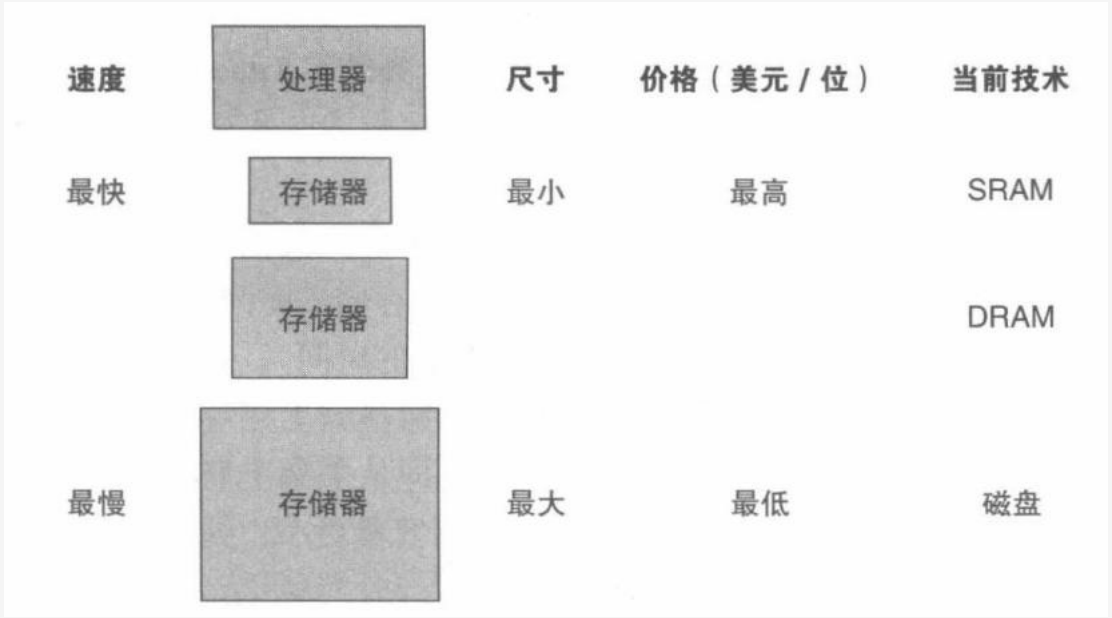
- 包括不同速度和容量的多级存储
- 存储速度越快，价格越昂贵，因此容量越小
- 速度越快的存储越靠近处理器
- 速度越慢的存储离存储器越远
- **目的：**以最低的价格提供最大容量的存储，同时访问速度与最快的存储相当



利用局部性建立存储层次结构

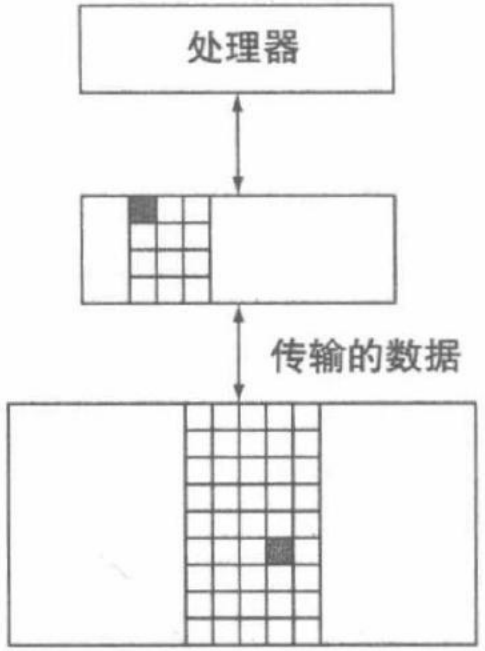
存储层次结构（memory hierarchy）

- 处理器只从最近的层次读取数据
- 存储的每一层存放哪些数据？
  - 较近层次中的数据是较远层次数据的子集
  - 最远的一层存放所有数据
  - 最近常用（及其附近）的数据放在较近的层次
- 不同层次的数据如何传输？
  - 数据只能在两个相邻层次间进行复制
  - 数据复制的最小单位是块（block）或行（line），一个块中可能包含多个字



存储器层级间的数据复制

- 层次化存储中每对层次都能被看做上层和下层
  - 上层更靠近处理器，比下层容量小、速度快
- 如果处理器要访问的数据在上层中找到
  - 命中(hit)：可从上层直接读取数据
  - 命中率(hit ratio/hit rate)：命中次数/访问次数
- 如果没能在上层找到所需数据
  - 失效(miss)：需要先访问下一层存储获取数据并复制到上层
  - 失效率(miss ratio/1-hit rate)：1-命中率
- 数据命中和失效的处理时间对性能有重要影响
  - 命中时间(hit time)：访问某个层次并命中需要的处理时间，包括判断命中或失效的时间
  - 失效损失(miss penalty)：将失效的数据块从下层复制到上层，并返回给处理器的时间



存储系统是性能的主要决定因素

- 层次化存储对局部性的利用
  - 时间局部性：将最近刚访问过的数据留在更近的存储层次中
  - 空间局部性：将包含了多个连续字的数据移动到更上层的存储中
- 层次化存储的上层速度更快，下层容量更大
  - 如果命中最上层，访问速度会很快
  - 如果失效，将访问下一级存储，容量变大但是速度变慢
- 如果命中率足够高
  - 处理器感受到的访存时间将会接近最上层存储，容量相当于最下层存储
  - 理想的存储系统状态
- 当前，相比执行运算，程序在访存上会花费大量的时间
- 存储器技术和存储系统设计非常关键

存储技术

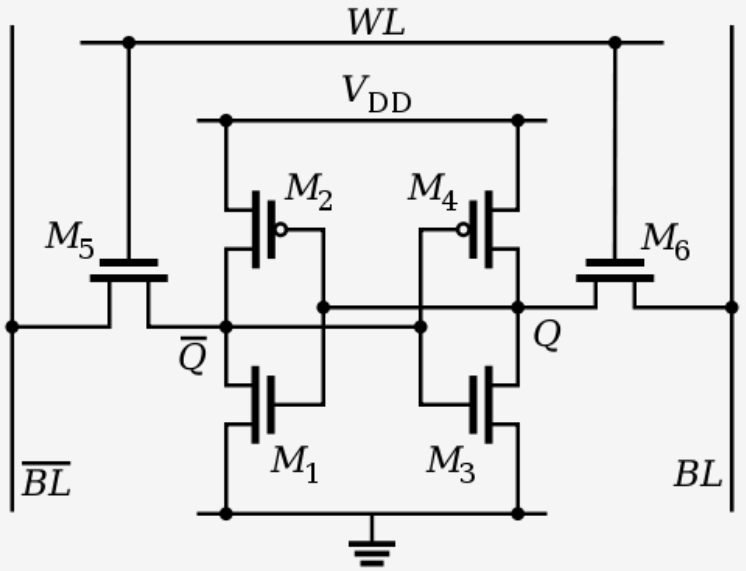
- 当今层次化存储中的四种主要技术（数据来源2012年）

存储技术	典型访问时间	单位成本（美元/GB）	常用存储器名称
SRAM半导体存储	0.5~2.5ns	500~1000	Cache，缓存
DRAM半导体存储	50~70ns	10~20	主存，内存
闪存半导体存储	5000~50000ns	0.75~1.00	固态硬盘，U盘
磁盘	5000000~20000000ns	0.05~0.10	硬盘

SRAM存储技术

静态随机访问存储（SRAM, Static Random Access Memory）

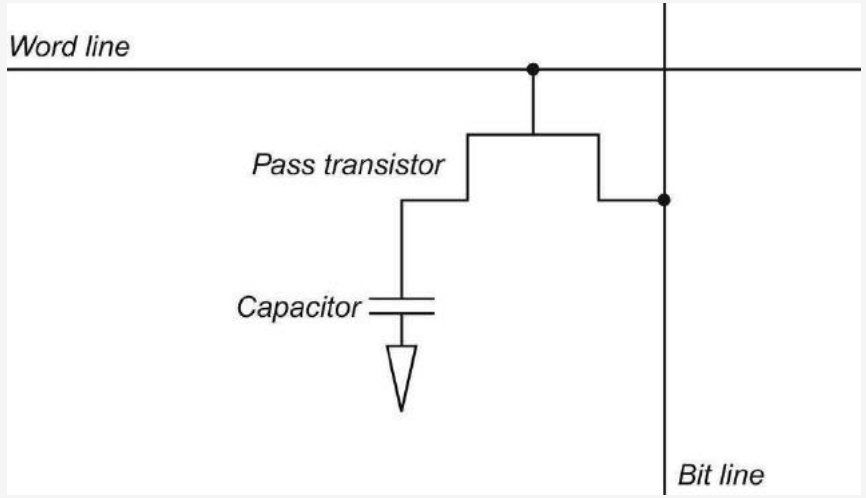
- 以电荷形式存储数据
- 由6个或8个晶体管组成
- 通电状态下可以一直保存数值，不需要进行刷新
- 读写操作的访问时间不同
- 通常被用作高速缓存
  - 过去大多是独立的SRAM芯片
  - 如今高速缓存都被集成到处理器芯片上



DRAM存储技术

动态随机访问存储（DRAM，Dynamic Random Access Memory）

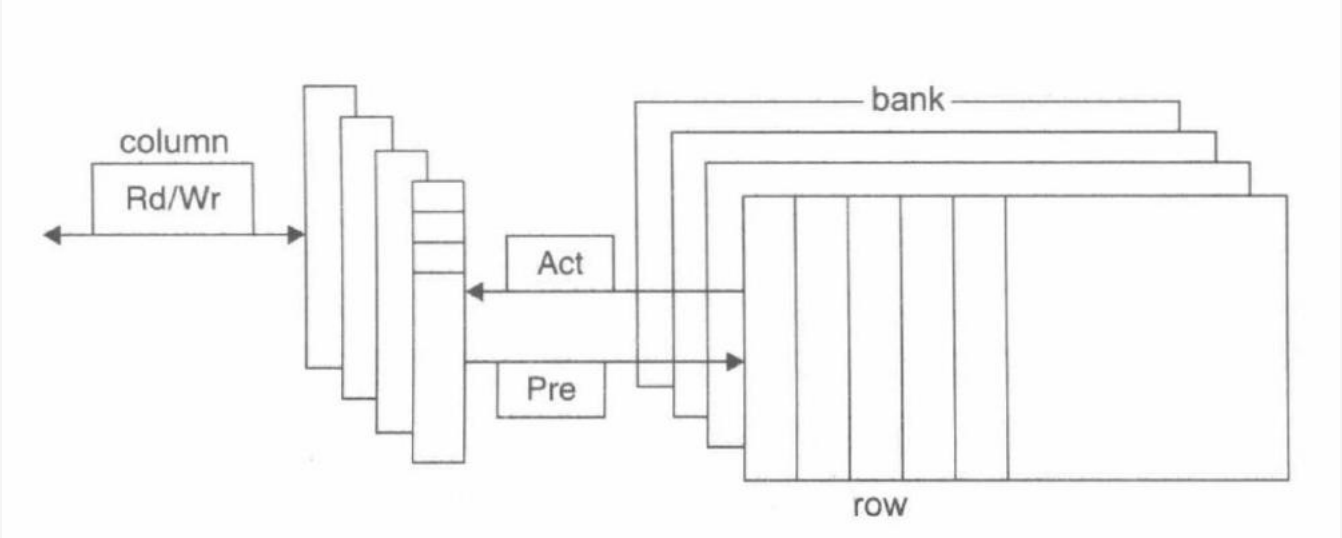
- 以电荷形式存储数据，由单个晶体管组成
- 晶体管数量比SRAM少，密度更高，价格更低廉
- 不能长久保存数据，需要进行周期性的刷新
  - 只能保持几微秒的时间
  - 刷新时，读取其中的内容并再次写回
- 通常被用作主存/内存



DRAM存储技术

DRAM的内部结构

- 以bank方式组织，每个bank包括一系列行（row）
- 预充电命令（Pre,pre-charge）可以打开或关闭某个bank
- 激活命令（Act,activate）发送行地址并将对应数据传输到缓冲区中
- 每个bank拥有一个缓冲区，缓冲区中数据通过列（column）地址向外部传输数据



DRAM存储技术

DRAM的参数趋势

生产年份	芯片容量	美元/GB	新行/列的访问时间（ns）	缓冲行的平均列访问时间（ns）
1980	64 Kibibit	1 500 000	250	150
1983	256 Kibibit	500 000	185	100
1985	1 Mebibit	200 000	135	40
1989	4 Mebibit	50 000	110	40
1992	16 Mebibit	15 000	90	30
1996	64 Mebibit	10 000	60	12
1998	128 Mebibit	4 000	60	10
2000	256 Mebibit	1 000	55	7
2004	512 Mebibit	250	50	5
2007	1 Gibibit	50	45	1.25
2010	2 Gibibit	30	40	1
2012	4 Gibibit	1	35	0.8

DRAM存储技术

先进的DRAM技术

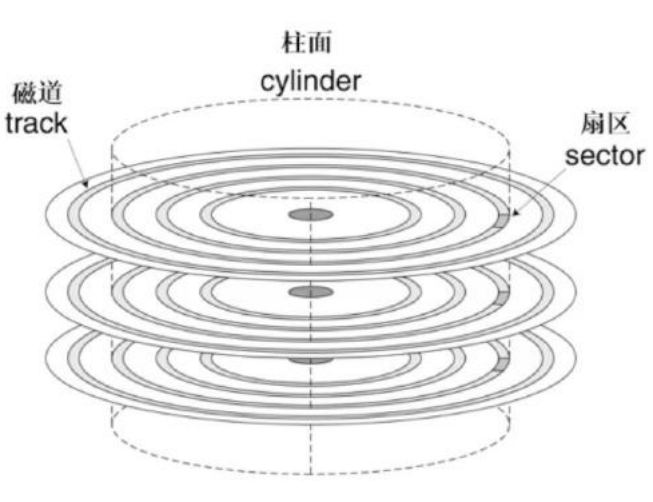
- 同步DRAM（Synchronous DRAM）
  - 给DRAM添加了时钟，消除了内存和处理器之间的同步问题
- 突发传输（burst transfer）
  - 获取一行中连续的数据，给出基址和需要获取的数据量，在每个时钟边沿到来时传输后续数据
  - 比分别提供每个数据单元的地址延迟低
- 双倍数据传输率（DDR, Double Data Rate）SDRAM
  - 在时钟的上升沿和下降沿都可以执行传输操作
- 四倍数据传输率（QDR, Quad Data Rate）SDRAM
  - 在DDR的基础上将输入和输出分离

闪存

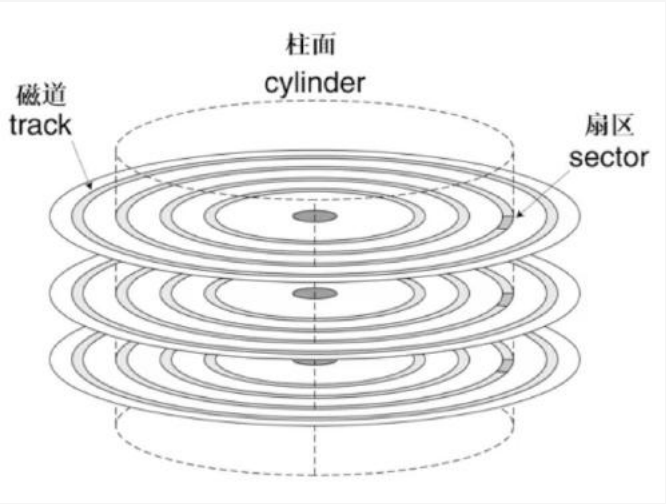
- 电可擦除的可编程只读存储器（EEPROM）
- 与磁盘相同，都是非易失性半导体存储器
- 比磁盘快100-1000倍，更小、功耗更低、更坚固
- 每GB成本更高（介于磁盘和DRAM之间）
- 写操作会对器件本身产生磨损
- 耗损均衡（wear leveling）
  - 闪存产品中包含一个控制器
  - 将发生多次写的块重新映射到较少被写的块，使操作尽量分散
  - 也可以将生产过程中出现故障的存储单元屏蔽，改善产品良率

磁盘的构造

- 由数个盘片组成，盘片围绕轴心转动
- 盘片的两面都被磁性记忆材料覆盖，类似磁带
- 由转臂通过移动进行信息的读写
- 每个盘面都配有一个磁头
- 磁道（track）：磁盘表面的同心圆
- 扇区（section）：磁道上的一段，读写信息的最小单位
  - 扇区ID
  - 数据：容量一般为512-4096字节
  - 纠错码（ECC）：用于隐藏缺陷与记录差错
  - 同步字段和间隙
- 柱面（cylinder）：不同盘片同一位置的磁道集合



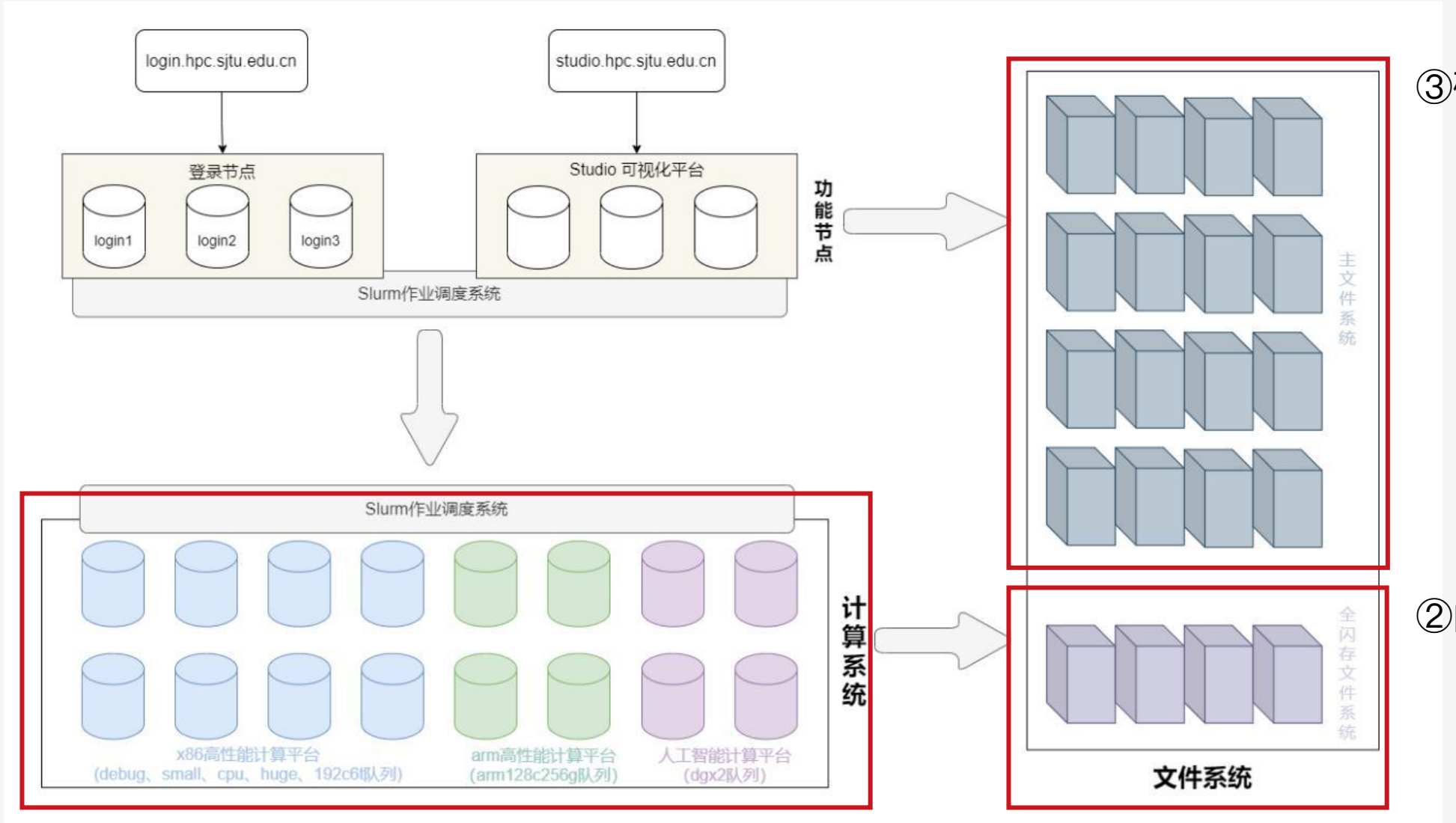
磁盘扇区的访问



- 如有其他访问正在执行，则有排队延时

1. 寻道（seek）：将磁头定位在正确的磁道上方
   - 寻道时间（seek time）：将磁头移动到所需磁道上方的时间
   - 平均寻道时间：对所有可能的寻道时间求和并计算平均值
2. 旋转延时（rotational latency）：等待所需扇区旋转到磁头下
   - 平均延时：磁盘旋转半周的时间
3. 传输时间（transfer time）：传输数据块的时间
   - 传输时间是扇区大小、旋转速度和磁道记录密度的函数

交我算集群的存储架构


login.hpc.sjtu.edu.cn
studio.hpc.sjtu.edu.cn
登录节点
login1
login2
login3
Studio 可视化平台
功能节点
Slurm作业调度系统
主文件系统
③磁盘
①内存
Slurm作业调度系统
计算系统
x86高性能计算平台
(debug、small、cpu、huge、192c6t队列)
arm高性能计算平台
(arm128c256g队列)
人工智能计算平台
(dgx2队列)
全闪存文件系统
②闪存
文件系统

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

报告提纲

异构加速器

层次化存储

文件系统与数据管理

“交我算”课堂实践

交我算平台现役的算力设施

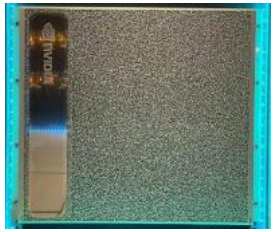
云际计算

教学 · 实践
科研 · 管理

交我算校级计算平台

云化硬件
统一调度

异构算力

云计算	人工智能计算	π2.0 超算	国产ARM 超算	思源一号超算
16,000 CPU Intel 6148	8 NVIDIA DGX-2 + 92 A100 GPU	26,240 CPU Intel 6248	12,800 CPU 鲲鹏 920	60,032 CPU Intel 8358
				

数字基座

聚合存储 65 PB

交我算集群文件系统

集群	文件系统	特点	个人目录路径	快捷环境变量
闵行集群 （π2.0+ARM+人工智能）	主文件系统	默认文件系统 登录后的默认系统 HDD盘，大容量、高可用、较高性能	/lustre/home/acct-xxxx/yyyy	$HOME
	全闪存文件系统	临时文件系统 适合作为临时工作目录 SSD盘，高性能、容量较小、安全性不高	/scratch/home/acct-xxxx/yyyy	$SCRATCH
思源一号	主文件系统	思源一号目前唯一的文件系统	/dssg/home/acct-xxxx/yyyy	$SIYUANHOME

*相关内容参考：https://docs.hpc.sjtu.edu.cn/job/slurm.html

“交我算”冷存储建设情况

20
PB

10
OceanStor 5300V5

2
元数据服务器

10
对象数据服务器

60
GB/s

2019年5月
开始建设

19年9月
交付上线

20年8月
扩容

74%
利用率

900+个
校内课题组

2000+个
校内用户

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

报告提纲

异构加速器

层次化存储

文件系统与数据管理

“交我算”课堂实践