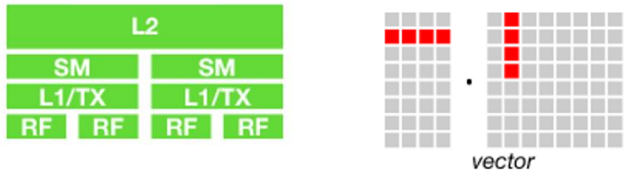
GPU Introduction

GPU简介

2025-10

三类算力芯片



来源：互联网

CPU

中央处理器（简称CPU）作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。现代的CPU一般通过多核技术来提升并行处理能力。大部分DPU可以近似为具有网络接口的多核CPU

GPGPU

图形处理器（简称GPU），是一种用于处理图像和图形或并行计算工作的处理器。目前利用并行图形处理单元来计算原本由中央处理器处理的通用计算任务，这些通用计算常常与图形处理没有任何关系

DSA（TPU/NPU/ASIC）

领域专用加速器（简称DSA），是使用专用电路结构进行一些特定场景或算法族（例如AI算法）计算的芯片级加速。DSA的计算并行度比传统的GPGPU更高，相近的概念是TPU/NPU/ASIC

三大CPU架构

x86架构是一种CPU指令集,也叫Intel指令集和IA32指令集, 是Intel公司开发的CPU架构标准（Intel/AMD）

ARM架构，曾称为先进精简指令集机器（Advanced RISC Machine），是种arm公司开发的精简指令RISC处理器架构，已开始规模部署在云计算场景（高通/苹果）

RISC-V是基于精简指令集计算(RISC)原理建立的开放指令集架构(ISA)。NVIDIA和Intel也开始进入RISC-V领域

GPGPU成为AI计算主力的原因

2012年，AlexNet团队将深度学习算法从CPU转换到GPU上，获得了更快更高的成绩。之后CUDA成为主流AI编译生态

2016年发布的P100率先在GPGPU中引入HBM，通过**近存计算架构**（Chiplet封装）提升数据吞吐速率

2017年发布的Volta GPGPU架构使用Tensor Core（一种**DSA**结构）来提升AI计算性能3-12倍

DSA大量用于大模型的训练和部署

DNN Model	*TPU v1 7/2016 (Inference)*	*TPU v3 4/2019 (Training & Inference)*	*TPU v4 Lite 2/2020 (Inference)*	*TPU v4 10/2022 (Training)*
MLP/DLRM	61%	27%	25%	24%
RNN	29%	21%	29%	2%
CNN	5%	24%	18%	12%
Transformer	--	21%	28%	57%
(BERT)	--	--	*(28%)*	*(26%)*
(LLM)	--	--	--	*(31%)*

来源：Google

国产AI芯片（DSA）难点

大部分灵活性不如GPGPU、技术代上短期难以超过NVIDIA集成的DSA（ Tensor Core ）

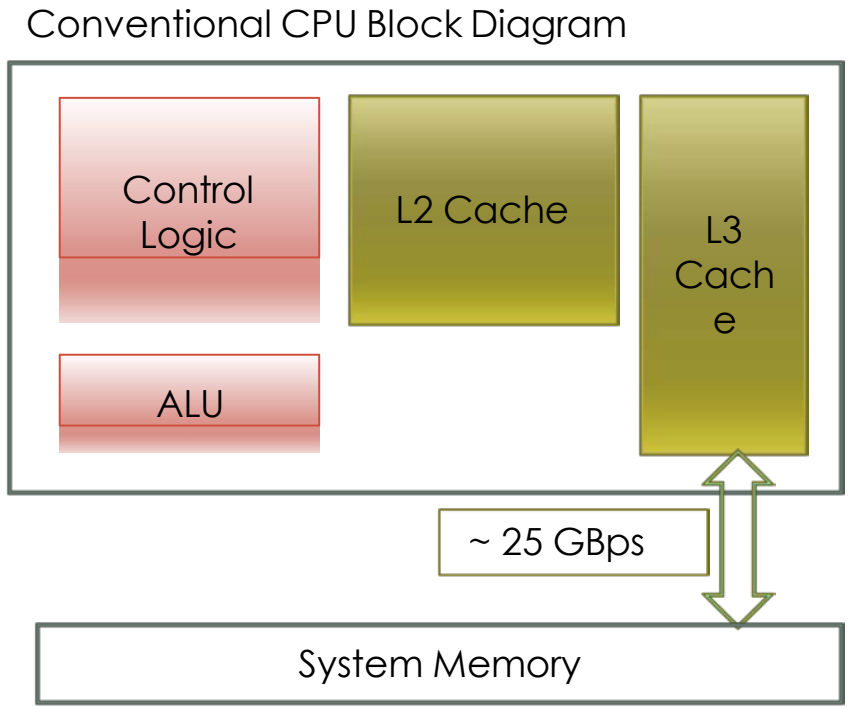
GPU VS GPGPU

- GPU全称Graphics Processing Unit，一般被称为图形处理单元。GPU被广泛用于嵌入式系统、移动电话、个人电脑、工作站和电子游戏解决方案中。现代的GPU在图像和图形处理方面十分高效，这是因为GPU被设计为高效的并行架构，使得其比通用处理器CPU在数据块上进行并行处理时更具优势

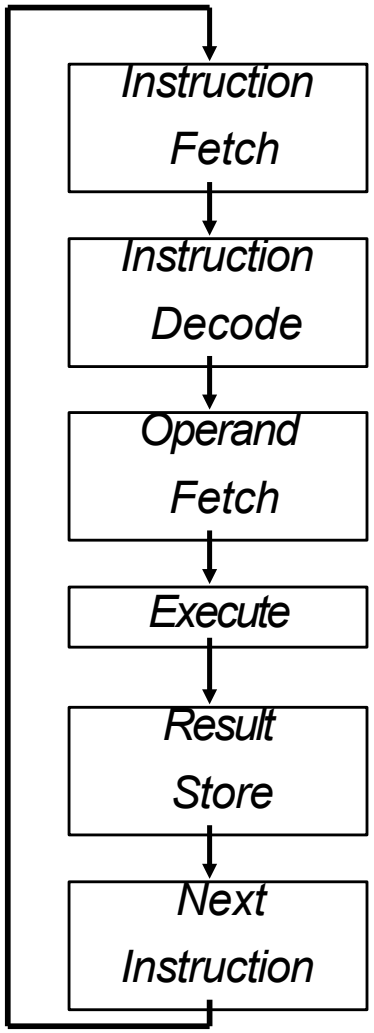
- GPGPU全称General Purpose GPU，其中第一个“GP”表示通用目的（General Purpose），第二个“GP”表示图形处理（Graphic Process），这两个“GP”搭配起来即“通用图形处理”，再加上“U”（Unit）就成为了完整的“通用图形处理器”

传统 CPU 架构：面向延迟优化控制执行

- 高时钟频（>1.5GHz）
- 核心数一般不超过64
- 较大的集中式缓存
  - 将长延迟内存访问转换为短延迟缓存访问
- 复杂的控制/复杂的 ALU
  - 分支预测以减少分支延迟
  - 数据转发以减少数据延迟
  - 减少操作延迟



处理器基本计算流程



从程序存储获取指令

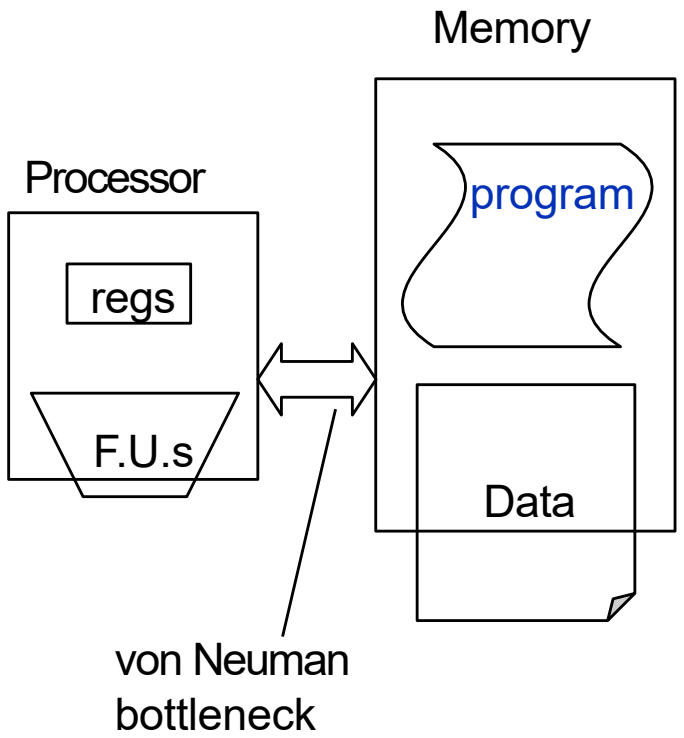
确定所需的动作和指令大小

定位并获取操作数数据

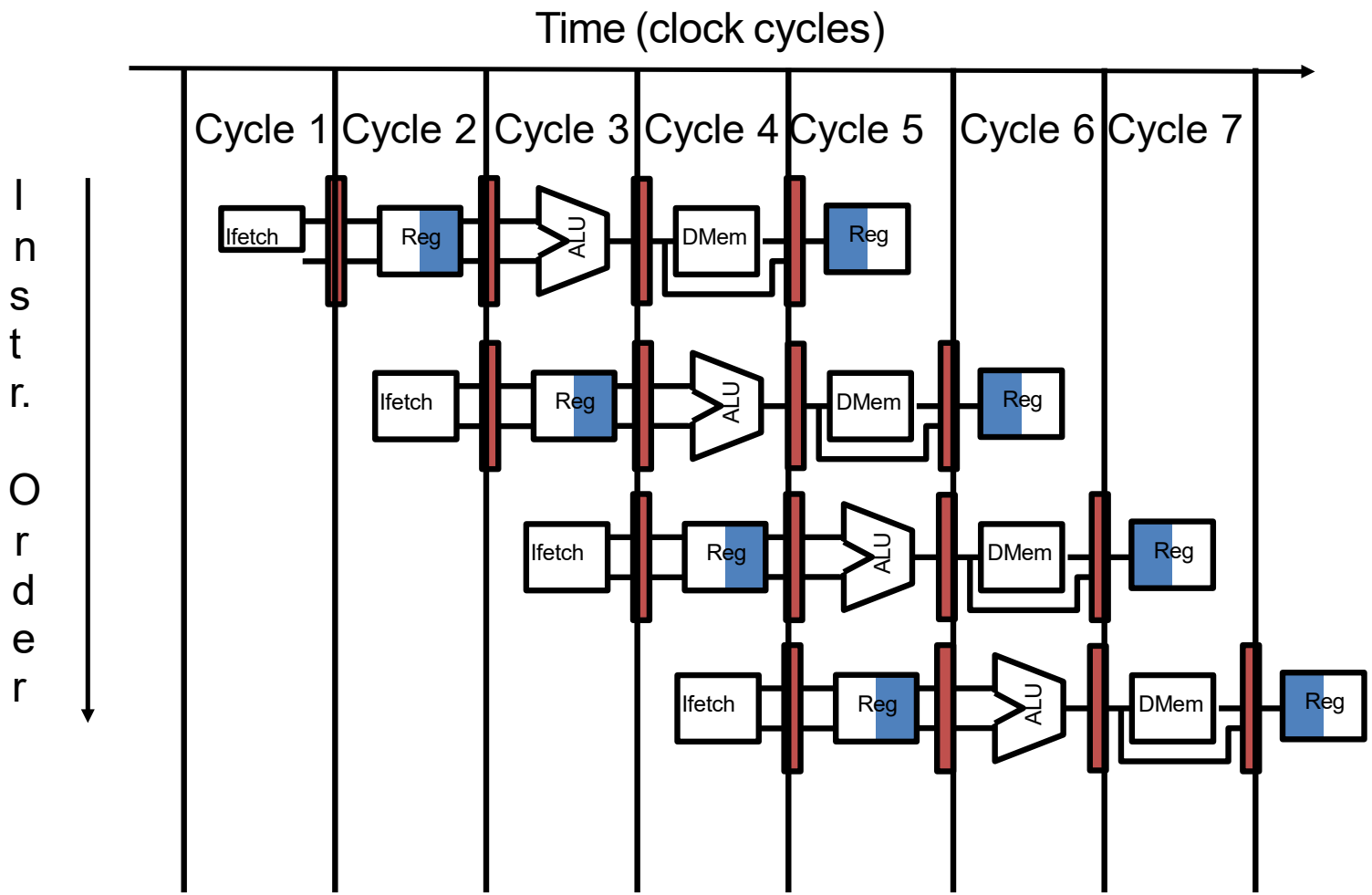
计算结果值或状态

存放结果以备后用/写回

确定后续指令

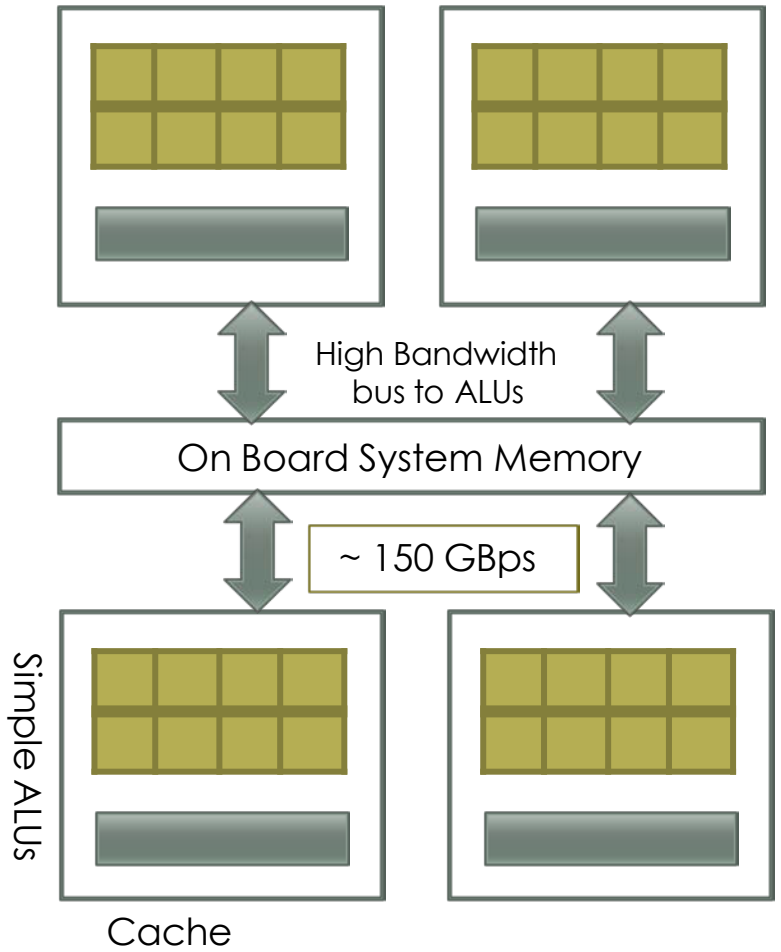


流水线原理


Time (clock cycles)
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Instr. Order
Ifetch
Reg
ALU
DMem
Reg

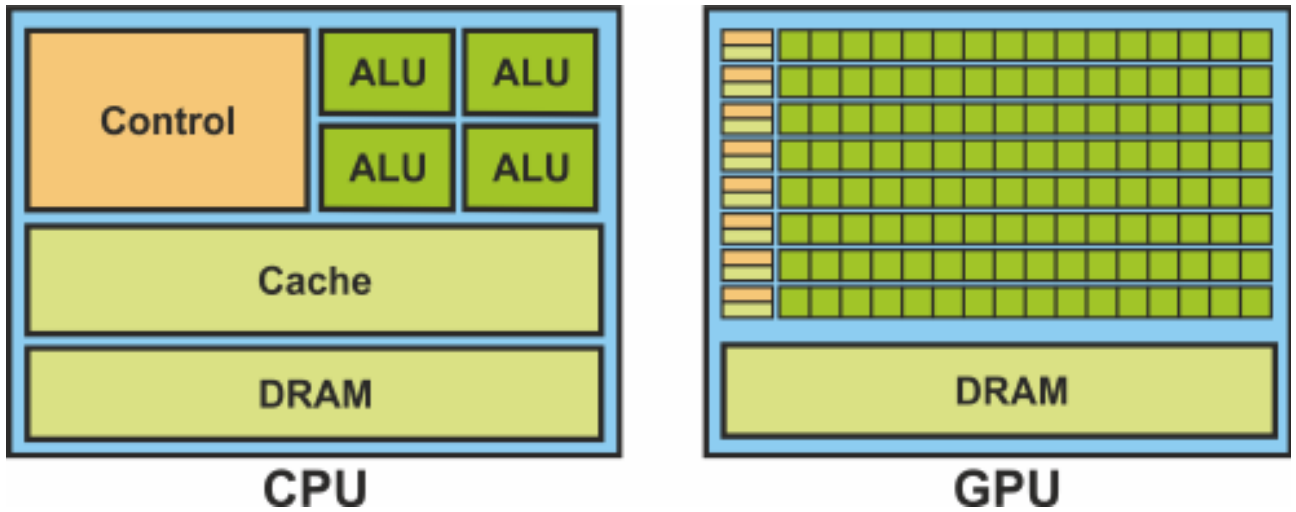
GPU架构：面向吞吐率优化

- 中等时钟频率
- 核心数一般超过1024，大量线程
- 较小缓存
  - 提高内存吞吐量
- 大量ALU/简单控制
  - 无分支预测
  - 无数据转发
  - 一般称为PE
  - 延迟高同时吞吐量高

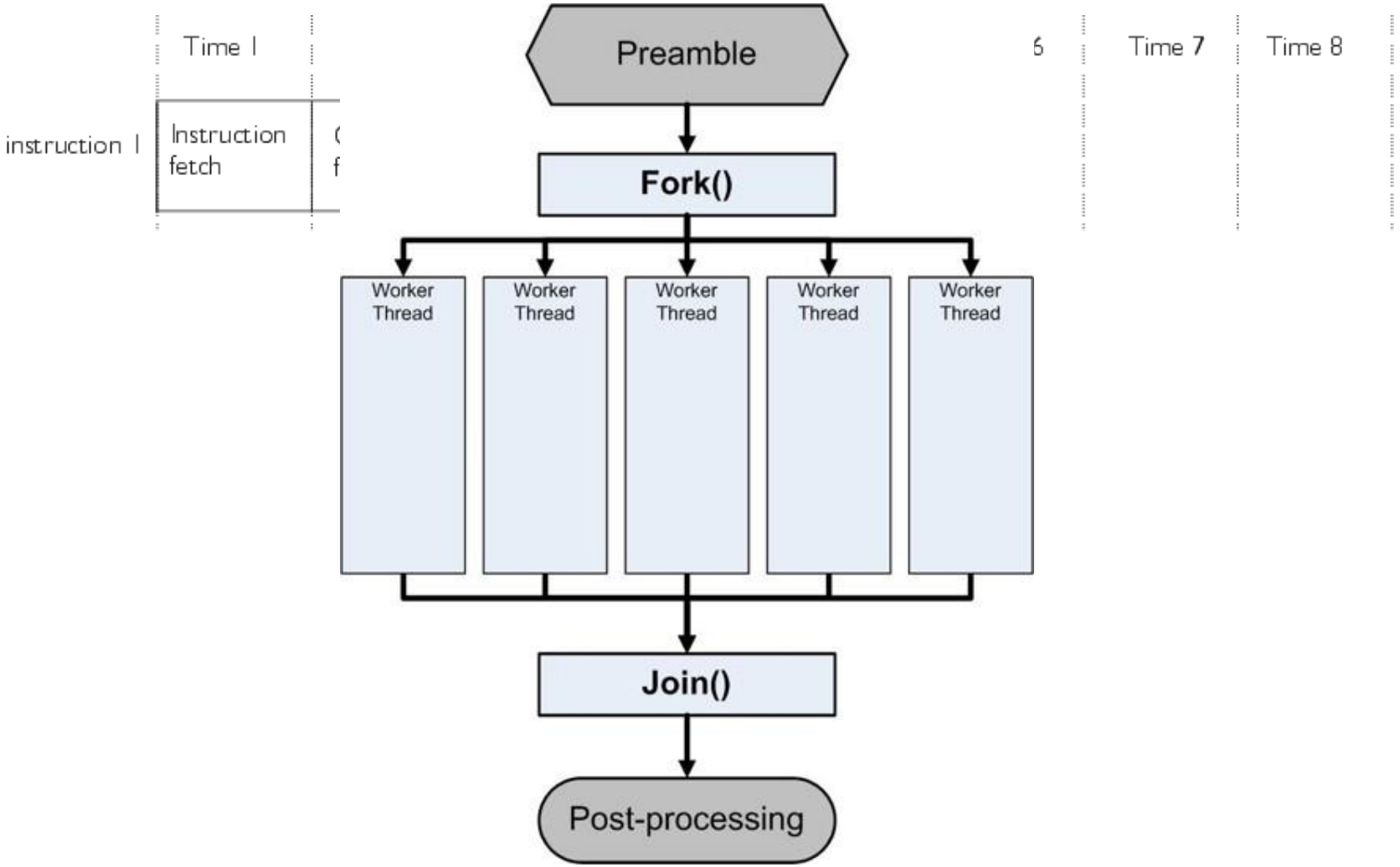


从CPU到GPU的计算范式变迁

- GPU 主要用于计算密集型、高数据并行计算
  - CPU大量的晶体管被Cache和控制电路所占用（负责控制指令和分支预测等操作），而ALU（Arithmetic Logic Unit，算术逻辑单元）占用的只是一小部分。与之相对的是GPU作为专用运算器其控制电路极其简单，单个核心对Cache需求较小，所以大部分的晶体管被用于组成各类专用计算单元和长流水线

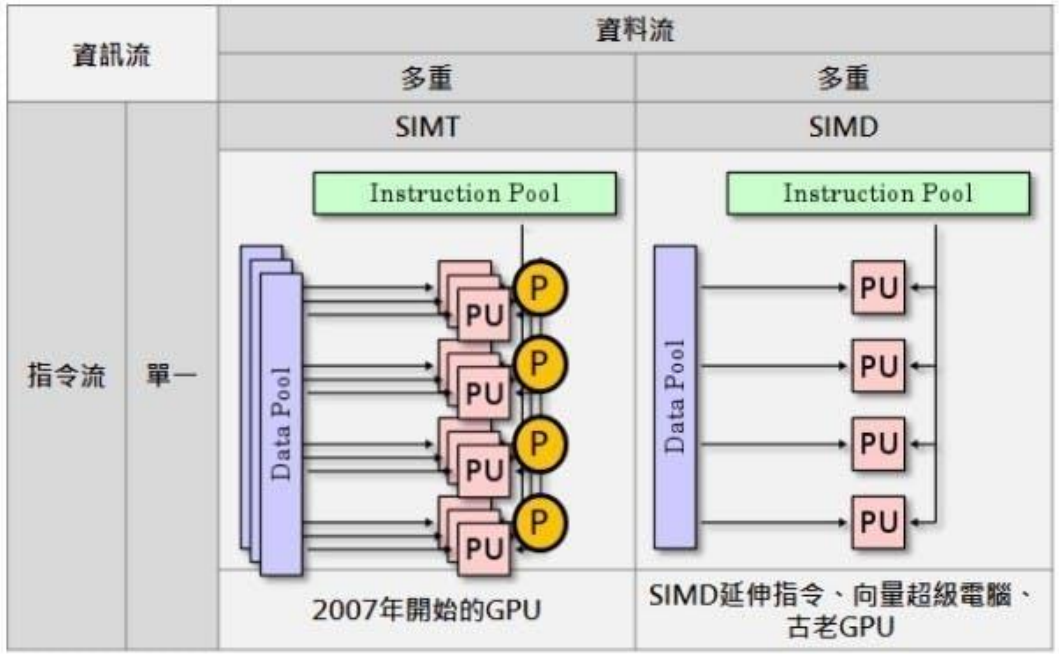


GPU的关键思路：并行化



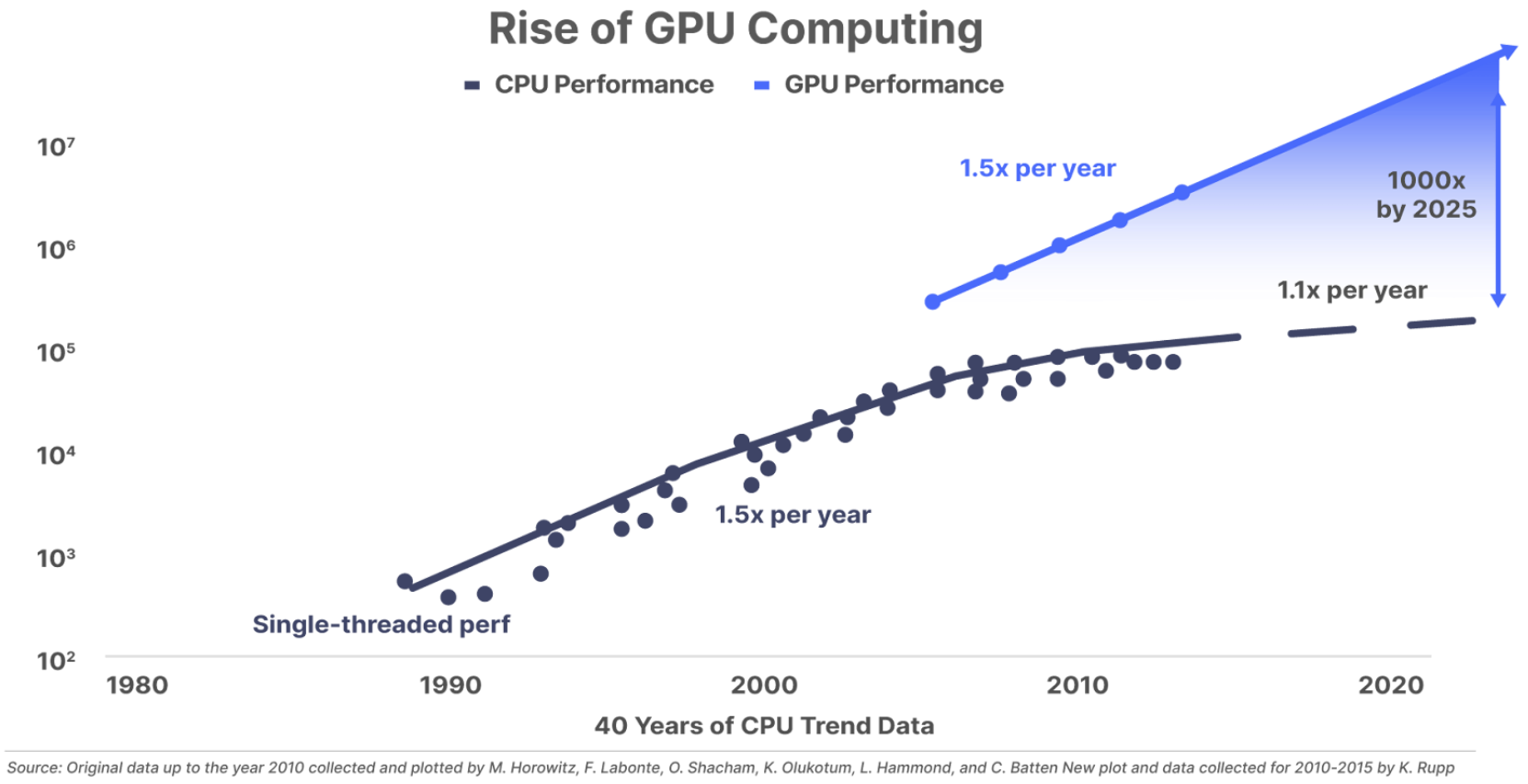
SIMD VS SIMT

- SIMD：单指令多数据
  - 通常所有执行单元共享程序计数器
- SIMT：单指令多线程
  - 每个线程有自己独立的程序计数器和寄存器状态， SIMT可以支持更多的线程(数百到数千)
  - 每个线程可以根据自己的执行情况选择不同的代码路径
- 编程灵活性
  - SIMT编程可以不需要考虑硬件特性,由编译器自动优化生成设备代码。在复杂计算的情况下，SIMT对开发人员和编译器的要求更低，更好优化。SIMD需要更规整的代码结构才能发挥优势
- 高级语言支持
  - SIMD需要在C语言中嵌入特定指令，对编程人员不友好。 SIMT的编程模型是线程，并行由硬件自动管理
- 可移植性/可调试性
  - SIMT可移植性更好，调试更简单



单核-多核-众核的算力跃迁

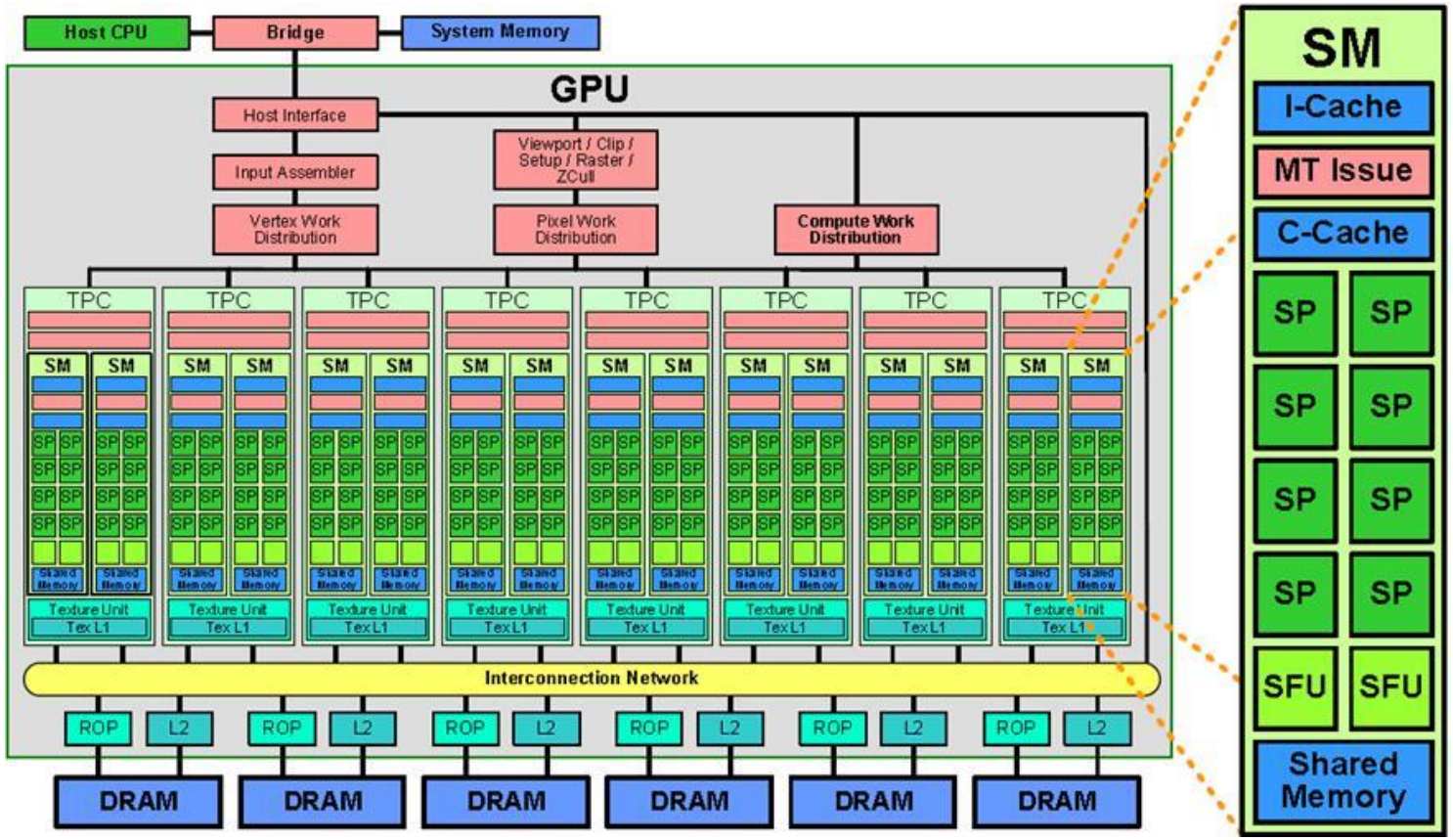
- 半导体制造工艺限制了单核处理器的性能发展
- 用空间换时间是GPU的关键思路



Source: Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotum, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

现代GPU基本架构

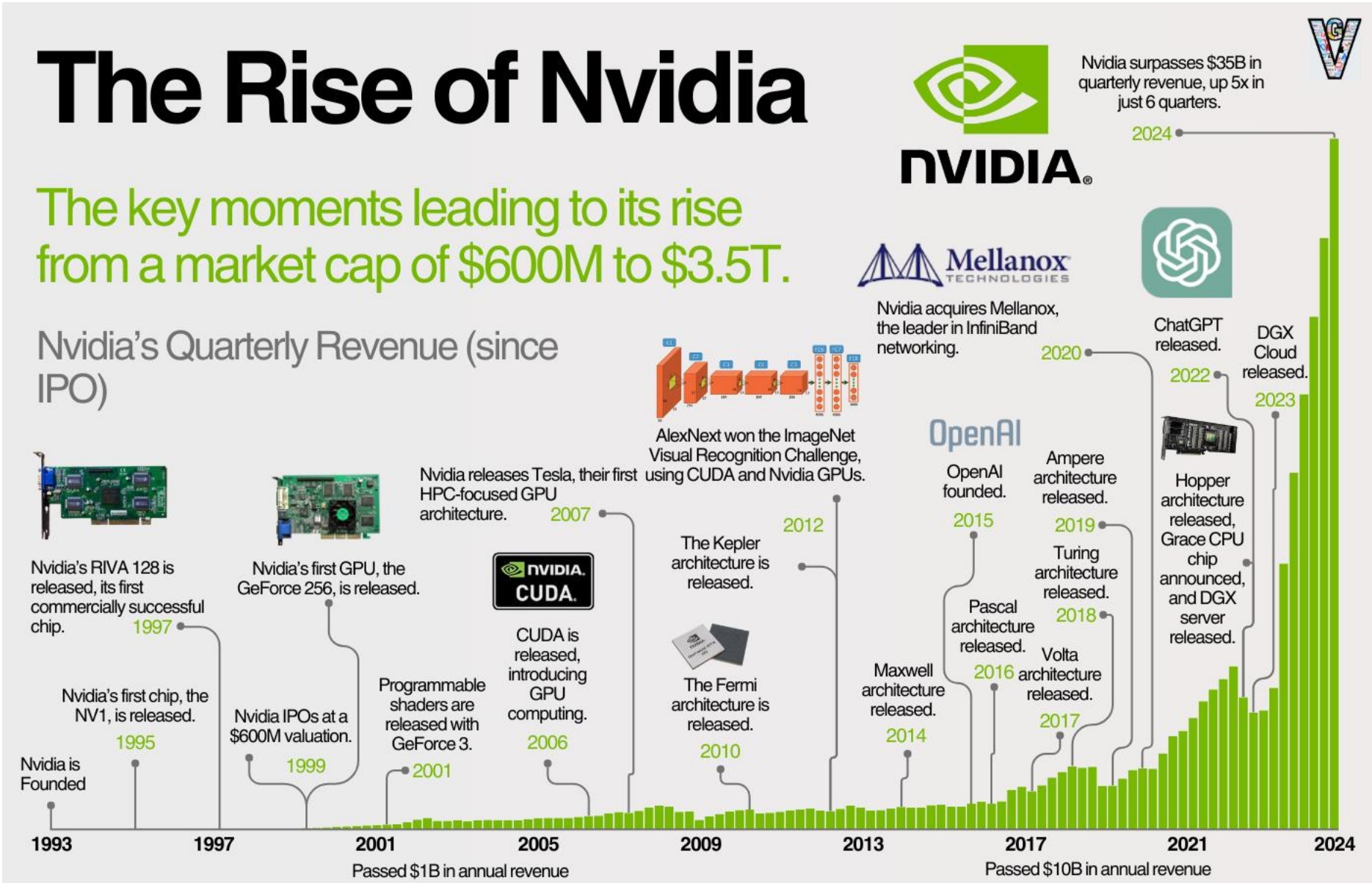
- 图形处理需要支持多条流水线，因此GPU拥有并行处理能力
- 运算单元极多（远多于CPU），与之相对的是强大的计算能力拥有更快速的显存、更大的显存位宽和显存带宽，适合高密集计算
- 在对流式数据进行并行处理上具有明显优势



NVIDIA发展史

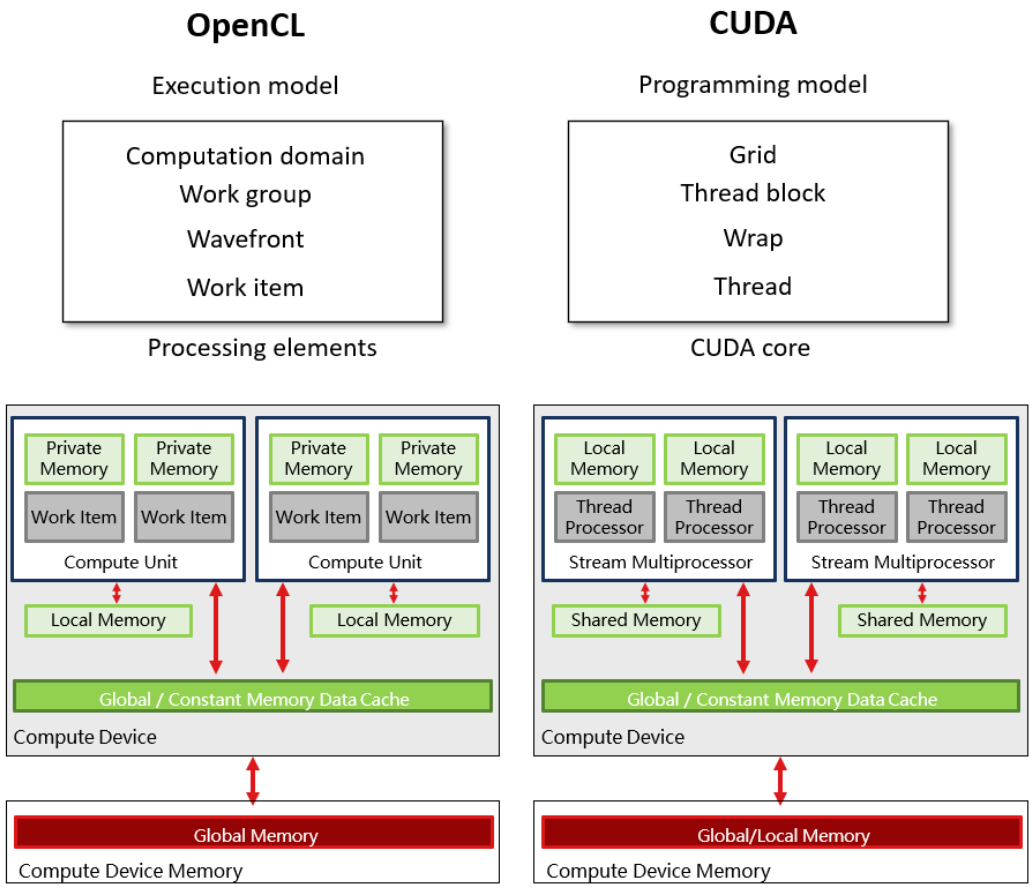
- 图形显示需求
- 挖矿需求
- 大模型计算需求

- 工艺进步
- 架构进步
- 编译器进步



CUDA VS OpenCL

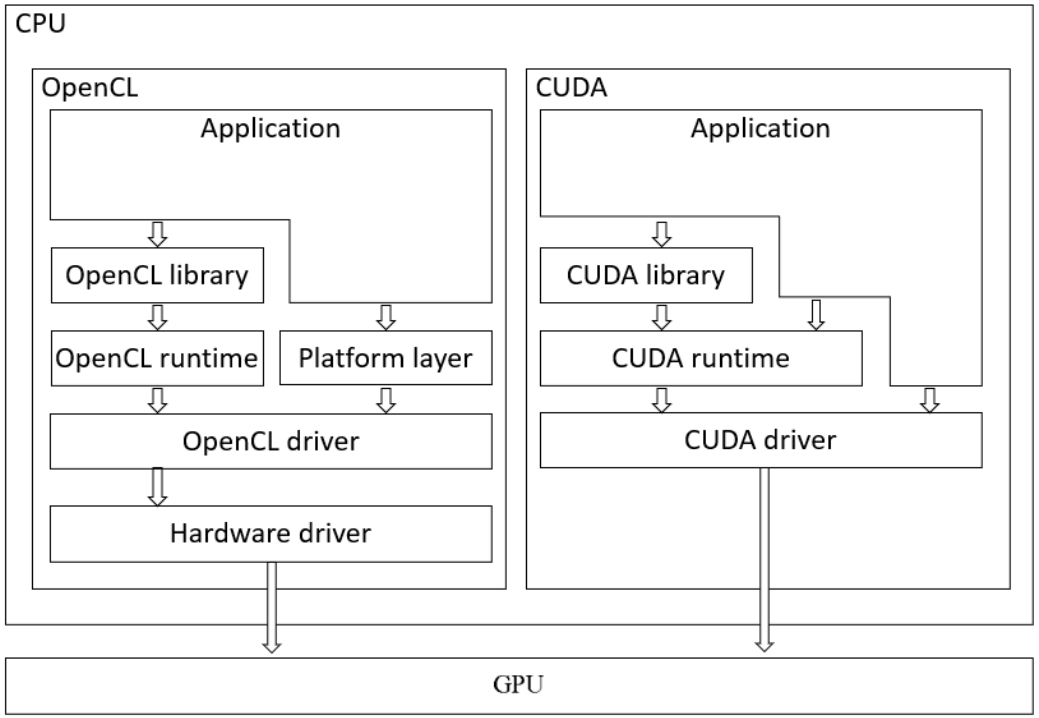
- CUDA (Compute Unified Device Architecture，计算通用设备架构) 和 OpenCL (Open Computing Language，开放计算语言) 是两种主流的GPGPU API
- OpenCL（Open Computing Language）是一个开放的标准API，具有跨平台的优势。CUDA是NVIDIA开发的并行计算API，包括Runtime API和Driver API



陈巍谈芯

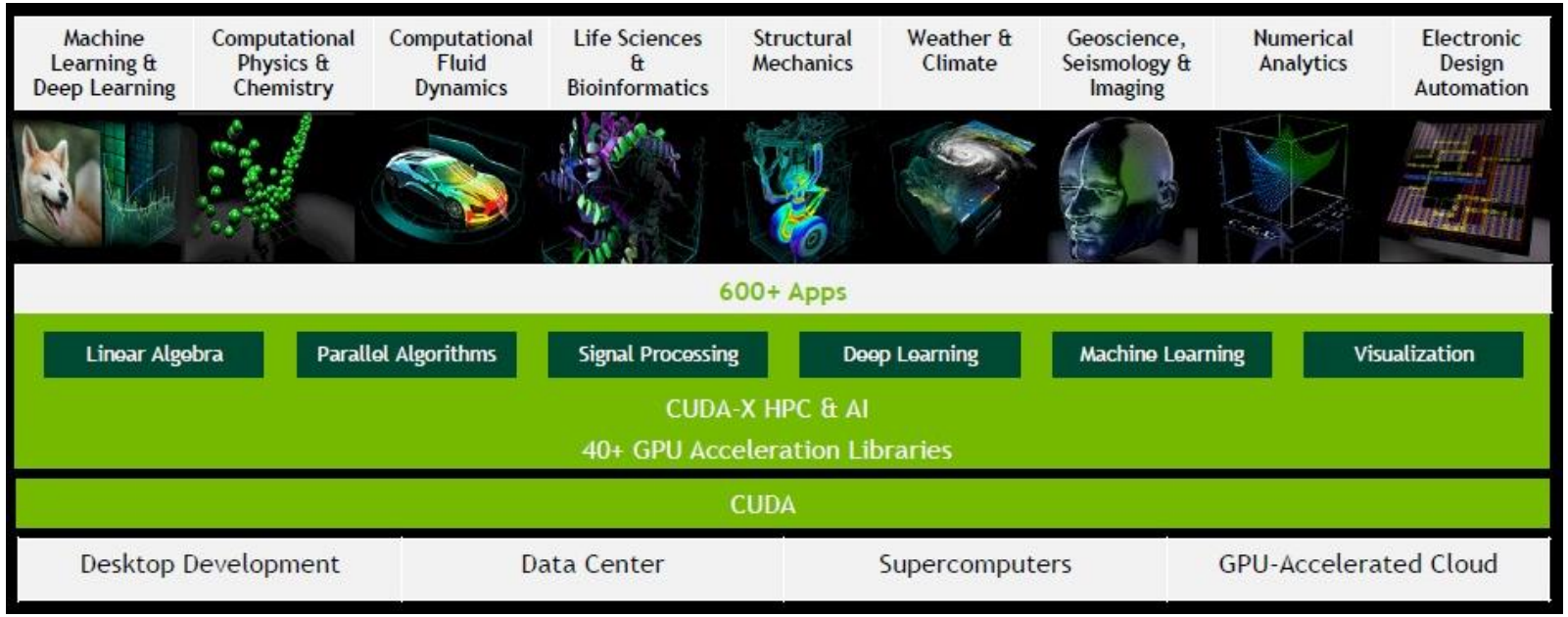
OpenCL编程模型

- OpenCL也是基于C的一个编程语言，分为Platform Layer、Runtime、Compiler三个部分：Platform Layer用来管理计算装置，提供初始化装置的界面，并建立compute contexts和work-queues
- Runtime用来管理资源，并执行程序的kernel。Compiler则是ISO C99的子集合，并加上了OpenCL特殊的语法。
- 在OpenCL的执行模型中，有所谓的Compute Kernel和Compute Program。Compute Kernel基本上类似于CUDA定义的kernel，是最基本的计算单元；而Compute Program则是Compute Kernel和内建函数的集合，类似一个动态函数库。很大程度上OpenCL与CUDA Driver API比较相像



CUDA编程模型

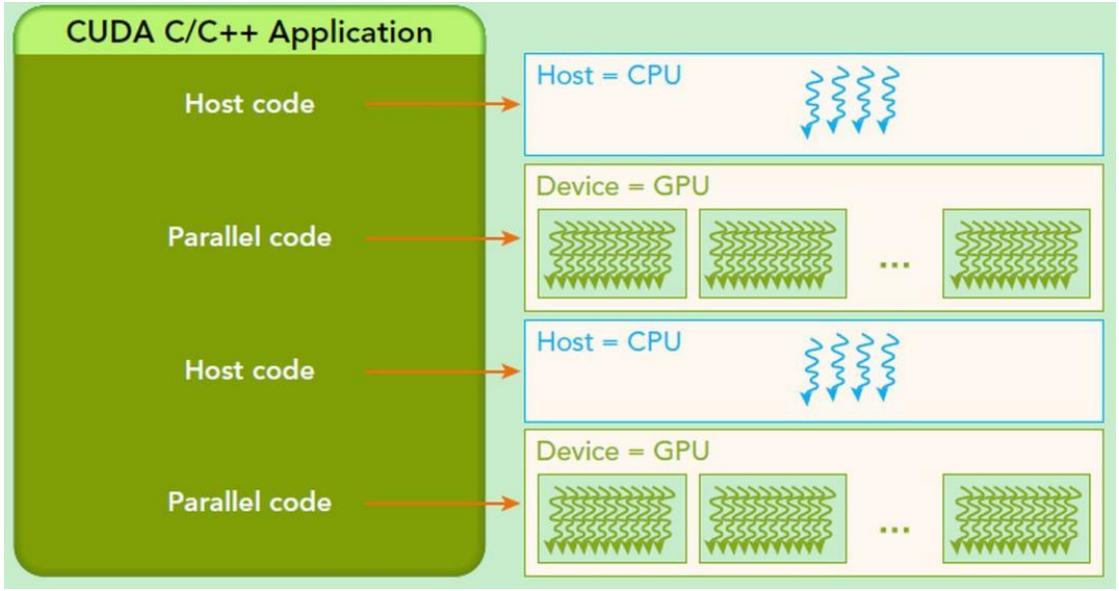
- 运行在GPU上的CUDA并行计算函数称为Kernel（内核函数）。一个完整的CUDA程序是由一系列的设备端Kernel函数并行步骤和主机端的串行处理步骤共同组成的。这些处理步骤会按照程序中相应语句的顺序依次执行，满足顺序一致性
- CUDA SDK提供的API分为CUDA Runtime API和CUDA driver API。CUDA Runtime API在CUDA Driver API的基础上进行了封装，隐藏了一些实现细节，编程更加方便



- CUDA Driver API是一种基于句柄的底层接口，可以加载二进制或汇编形式的kernel模块，指定参数并启动运算。CUDA driver API编程复杂，但有时能通过直接操作硬件的执行来实现一些更加复杂的功能

核函数

- 运行在GPU（Device）上的代码称做核（Kernel）函数，内核函数将会由大量硬件线程并行执行
- 核函数在调用时由 N 个不同的CUDA 线程并行执行 N 次，即循环被转化为并行线程，而不是像常规 C++ 函数那样仅执行一次。每个执行内核的线程都有一个唯一的线程 ID，可以通过内置变量在内核中访问
- 个典型的CUDA程序的执行步骤为：先把数据从CPU内存拷贝到GPU内存，再调用核函数对存储在GPU内存中的数据进行运算操作，最后将数据从GPU内存传送回CPU内存

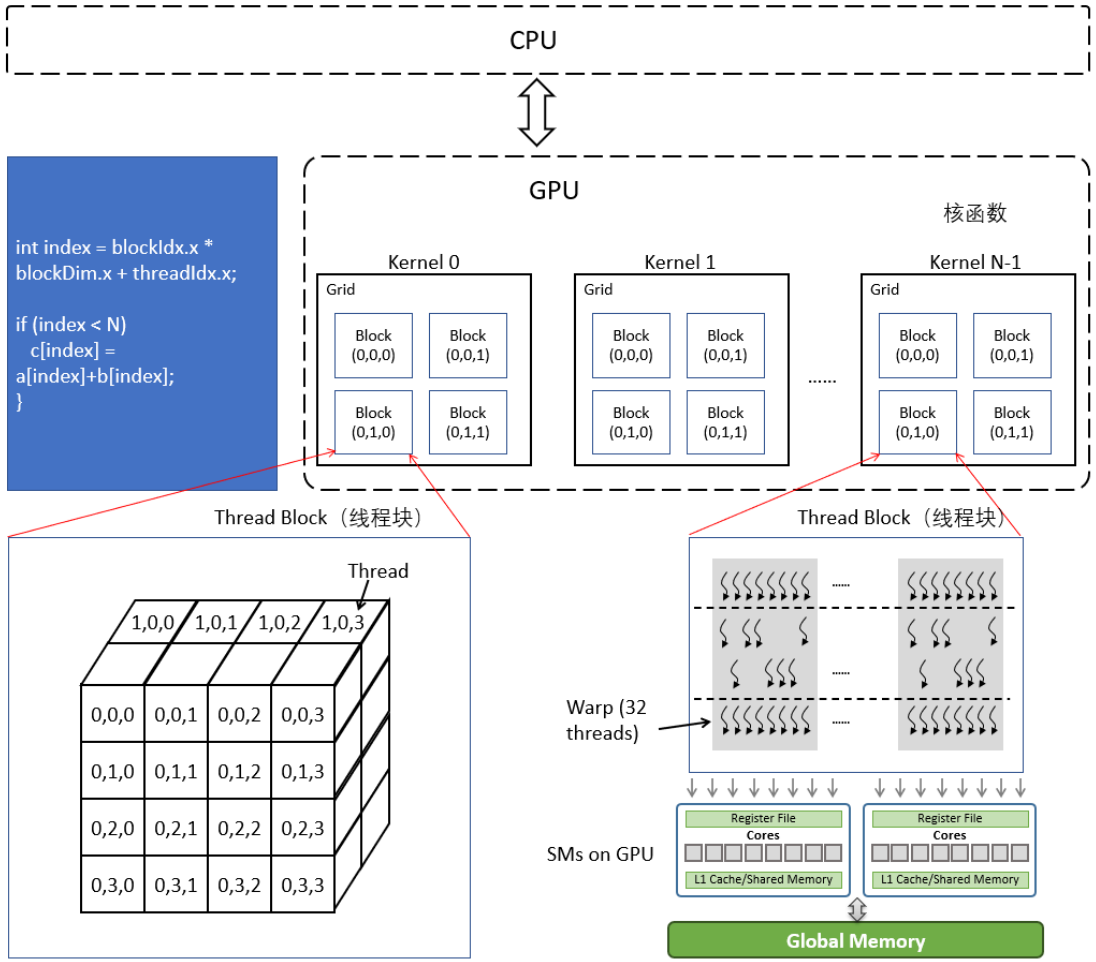



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float
{
    int i = threadIdx
    C[i] = A[i] + B[i];
}

int
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>
    ...
}
```

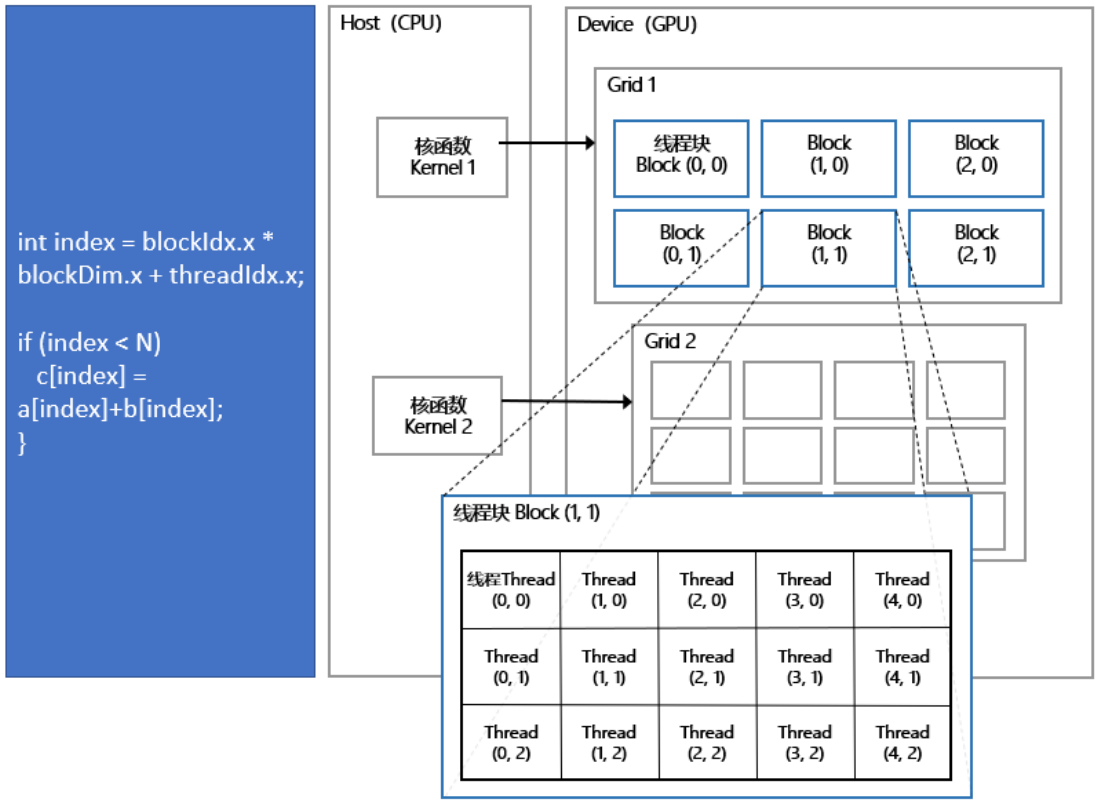
线程结构

- 主机上调用了核函数之后，程序执行进入GPU，生成大量的线程。线程执行前，核函数执行配置，确定线程块数和每个线程块中的线程数以及共享内存大小
- 线程块是一组并发线程，可以通过屏蔽同步和共享内存来相互协作，可以是多维的。而网格则是一组线程块，每个线程块可独立执行
- 每个核函数对应一组网格，而每个网格中包括多个线程块
- 线程通过以下坐标变量来区标识：
  - blockIdx，网格内的块标识
  - threadIdx ，块内的线程标识



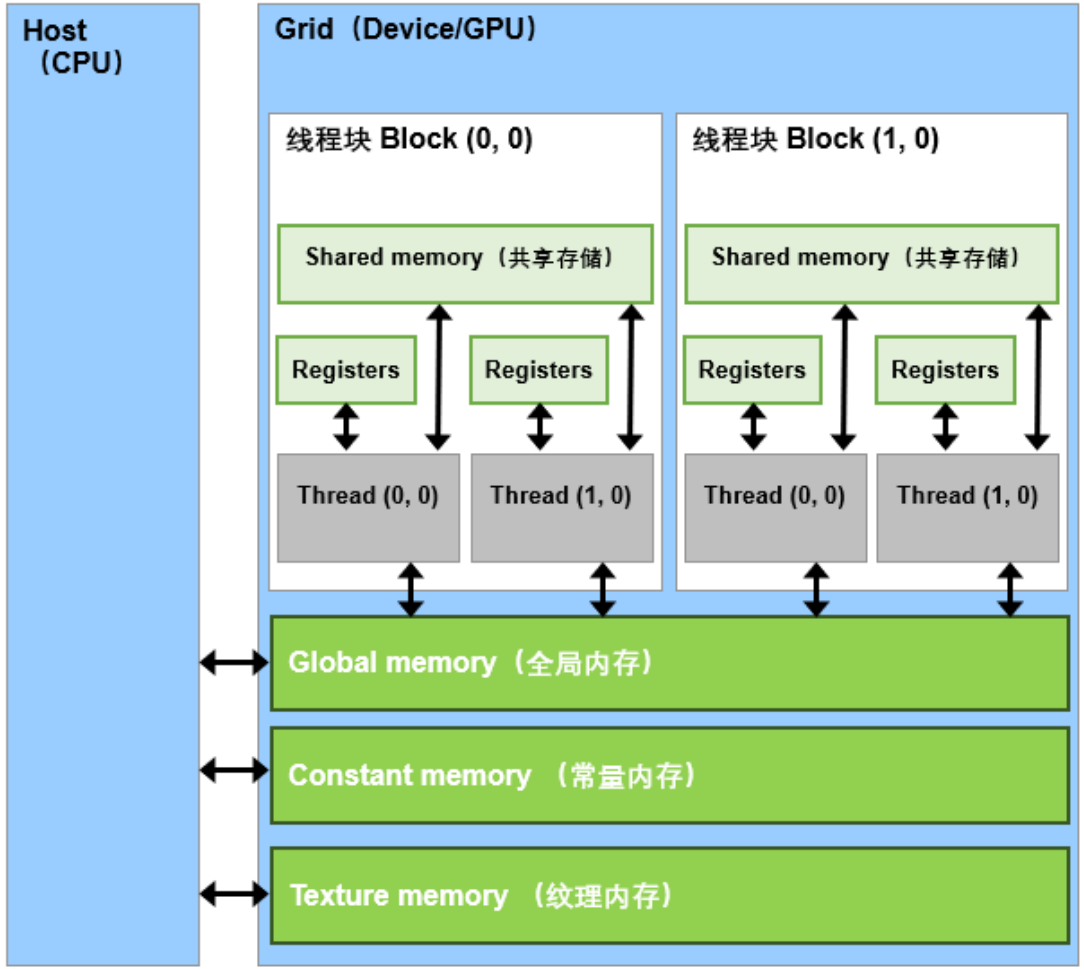
线程管理

- 由一个内核启动所产生的所有线程统称一个网格（Grid），同一网格中的所有线程共享相同的全局内存空间
- 一个网格由多个线程块（Block）构成，而一个线程块由一组线程（Thread）构成。这一组线程又可以分成多个GPU的基本执行单元——线程束（Warp），在OpenCL中对等的单元被称为Wavefront
- 线程网格和线程块从逻辑上代表了一个核函数的线程层次结构，这种组织方式可以帮助开发者有效地利用资源、优化性能



GPU存储管理

- 在编程时可手动指定变量存储的位置。具体来说，这些存储包括寄存器（Registers，本地存储）、共享存储（Shared Memory）、常量内存（Constant Memory）、全局内存（Global Memory）、本地内存（Local Memory）和纹理内存（Texture Memory）等
- GPU编程技巧
  - 尽量使用寄存器（本地存储）
  - 尽量将数据声明为局部变量
  - 当存在数据的重复利用时，可以把数据存放在共享内存里
  - 对于全局内存，尽量合并数据的访问以尽量减少设备对内存子系统再次发出访问操作的次数



清華大學深圳国际研究生院
Tsinghua Shenzhen International Graduate School

为培养下一代图灵奖获得者传递火种