Microsoft

人工智能系统 System for AI

矩阵运算与计算机体系结构
Computer architecture for Matrix computation

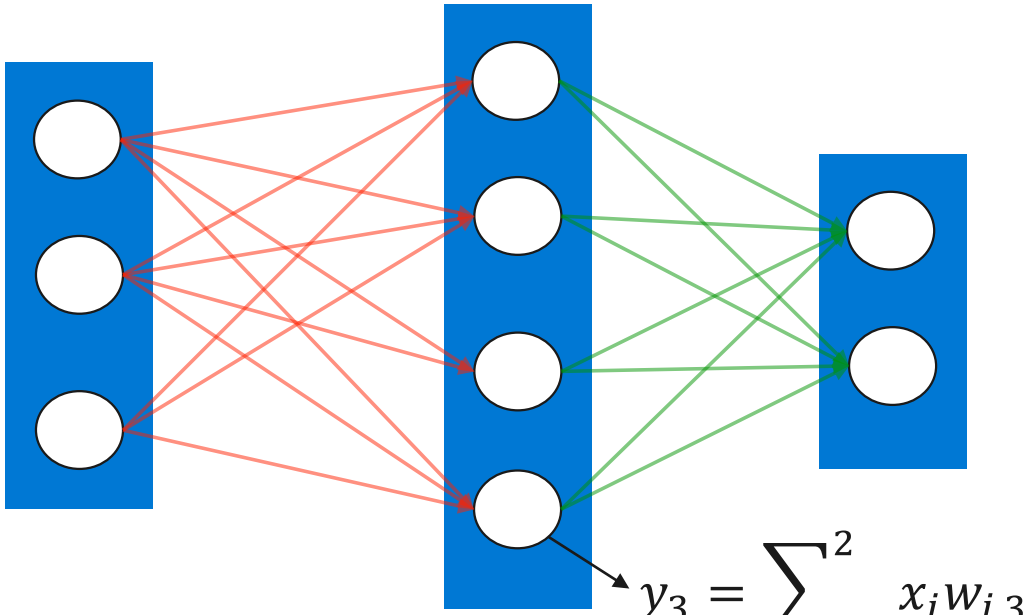
主要内容


深度学习
FC
CNN
RNN
Attention
矩阵运算
矩阵乘法
向量乘法
线性代数运算
矩阵分解
体系结构
CPU/SIMD
GPU
ASIC/TPU/IPU

深度学习常见模型的结构

- 全连接层
  - 前向全连接层
  - 多层感知模型（MLP)
- 卷积层
  - 常用于卷积神经网络中 (CNN)
  - 常用于图像任务中
- 循环网络层
  - 常用于循环神经网络中（RNN）
  - 常用在有时间序的顺序数据上（如，语音处理，自然语言处理等）
- Attention层
  - 通常实现为矩阵相乘
  - Transformer网络中的主要结构

全连接层

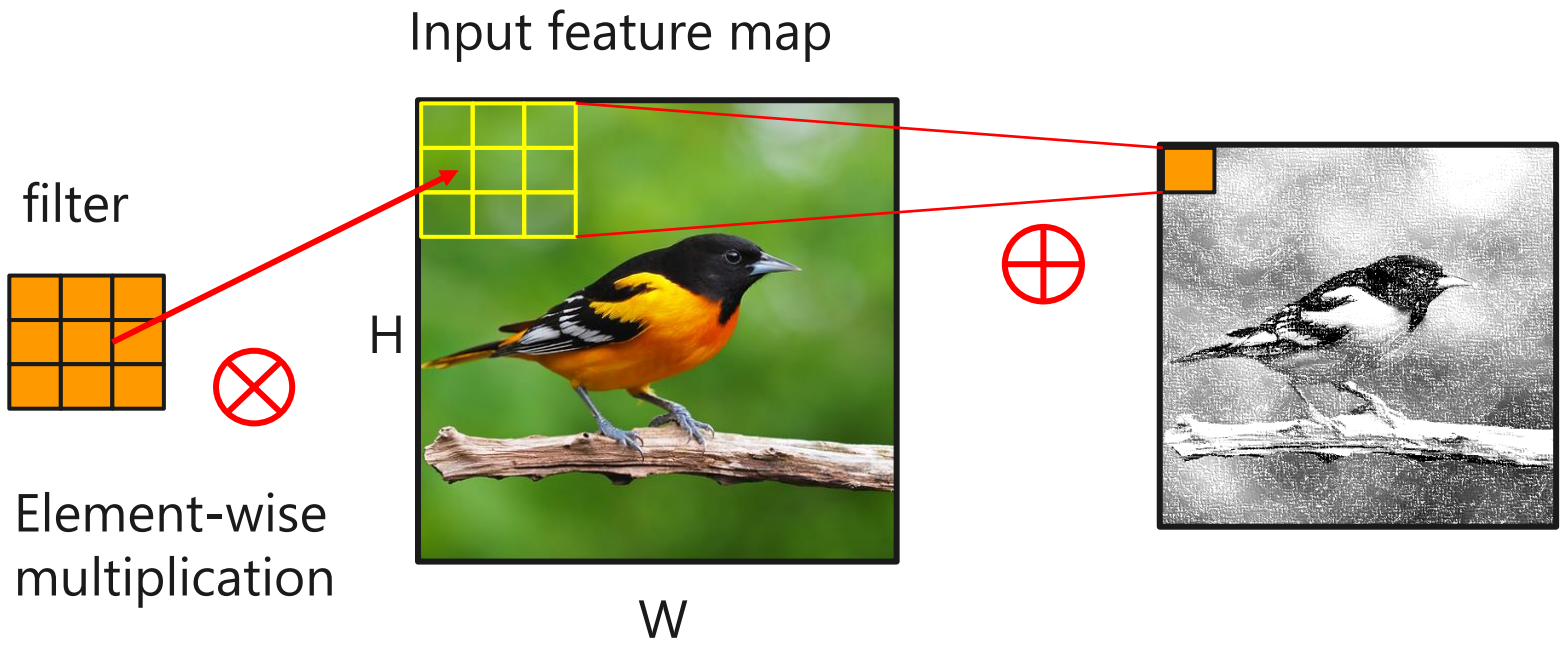


映射到矩阵运算：

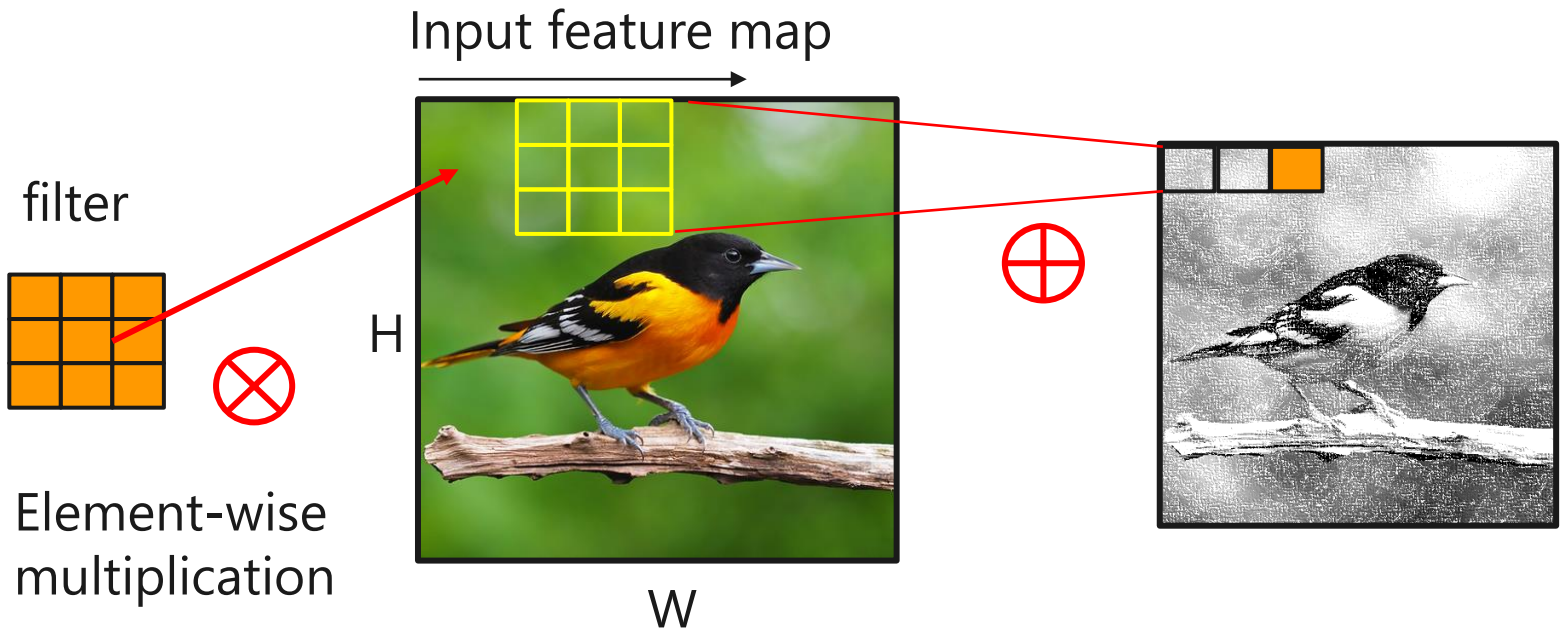
$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \quad W = \begin{bmatrix} w_{0,0} & \cdots & w_{0,2} \\ \vdots & \ddots & \vdots \\ w_{2,0} & \cdots & w_{2,3} \end{bmatrix} \quad Y = \begin{bmatrix} y_0 \\ \vdots \\ y_3 \end{bmatrix}$$

$$Y = W^T X$$

卷积层（Convolution）



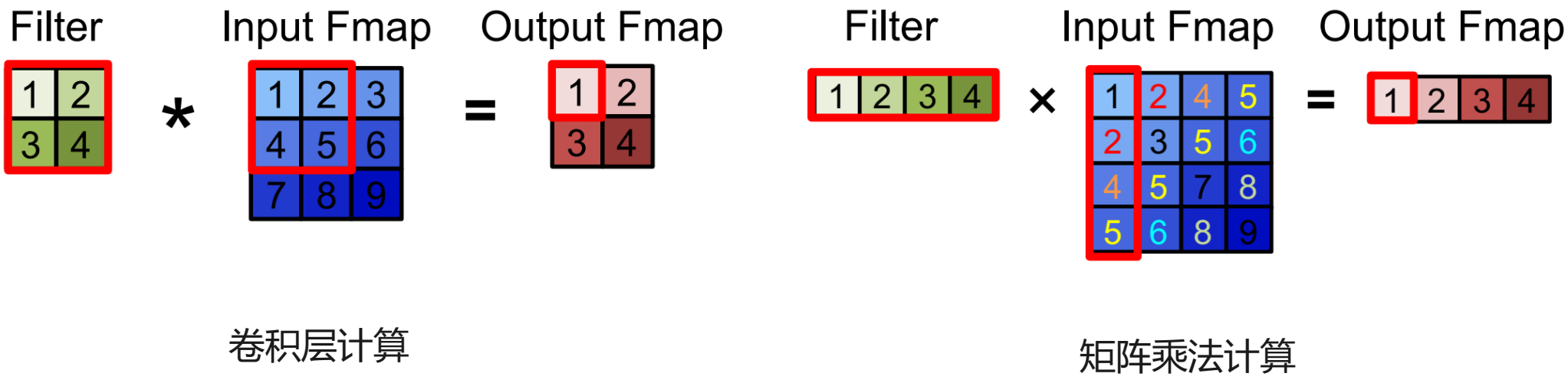
卷积层（Convolution）



Sliding Window processing

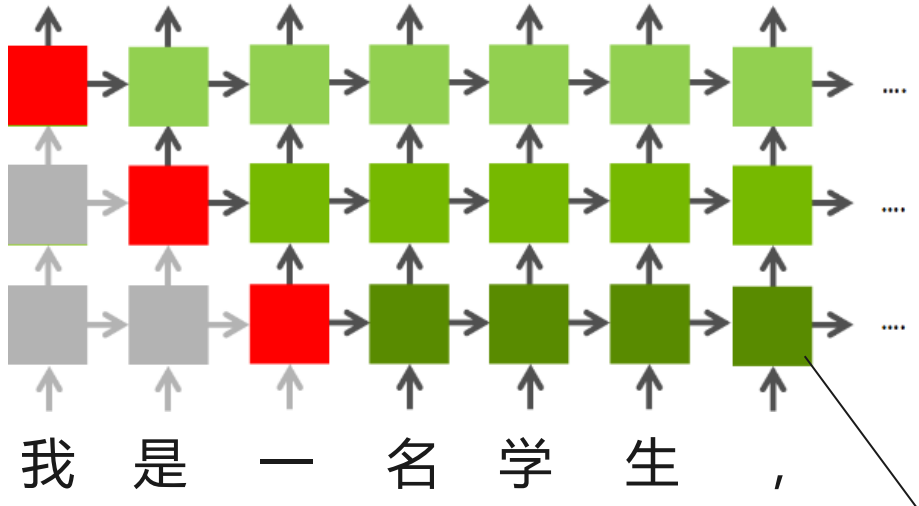
卷积层映射到矩阵运算

· 通过重组输入矩阵，卷积层计算可以被转换成矩阵乘法

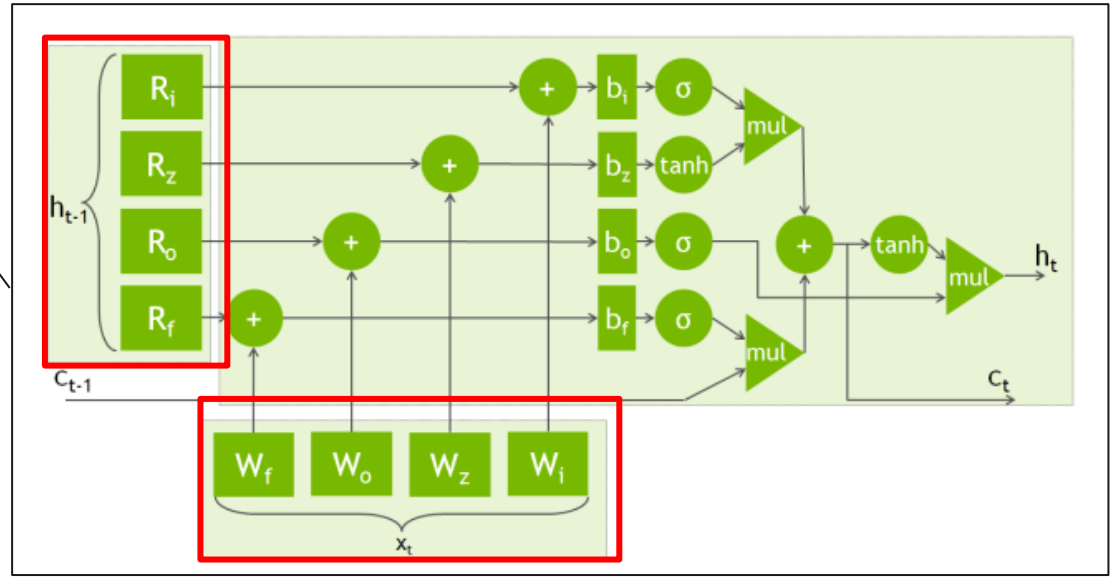


循环网络层

- LSTM循环神经网络中的矩阵运算

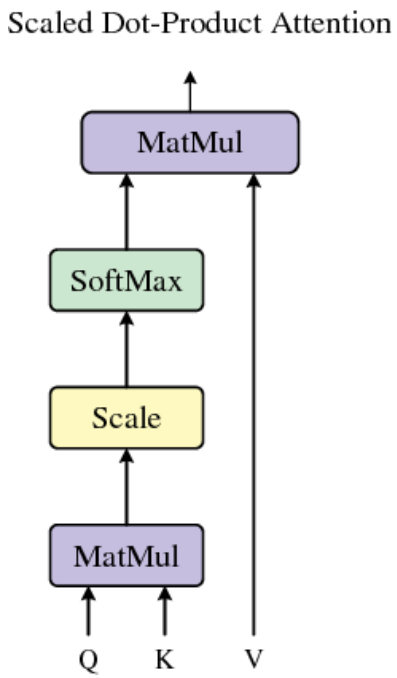


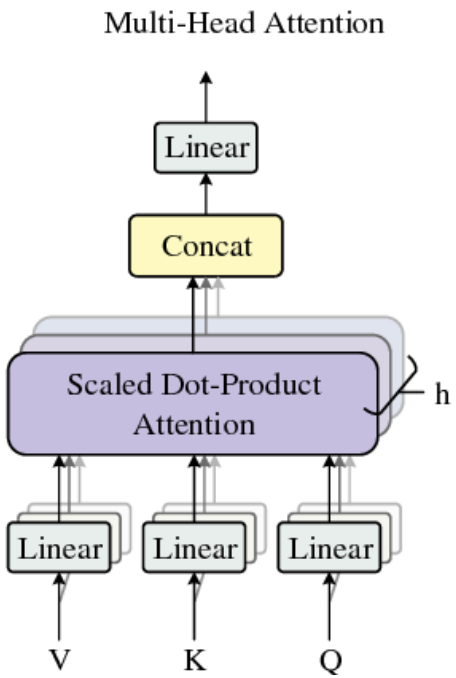
第一个cell都包括8个矩阵乘法和若干element_wise计算

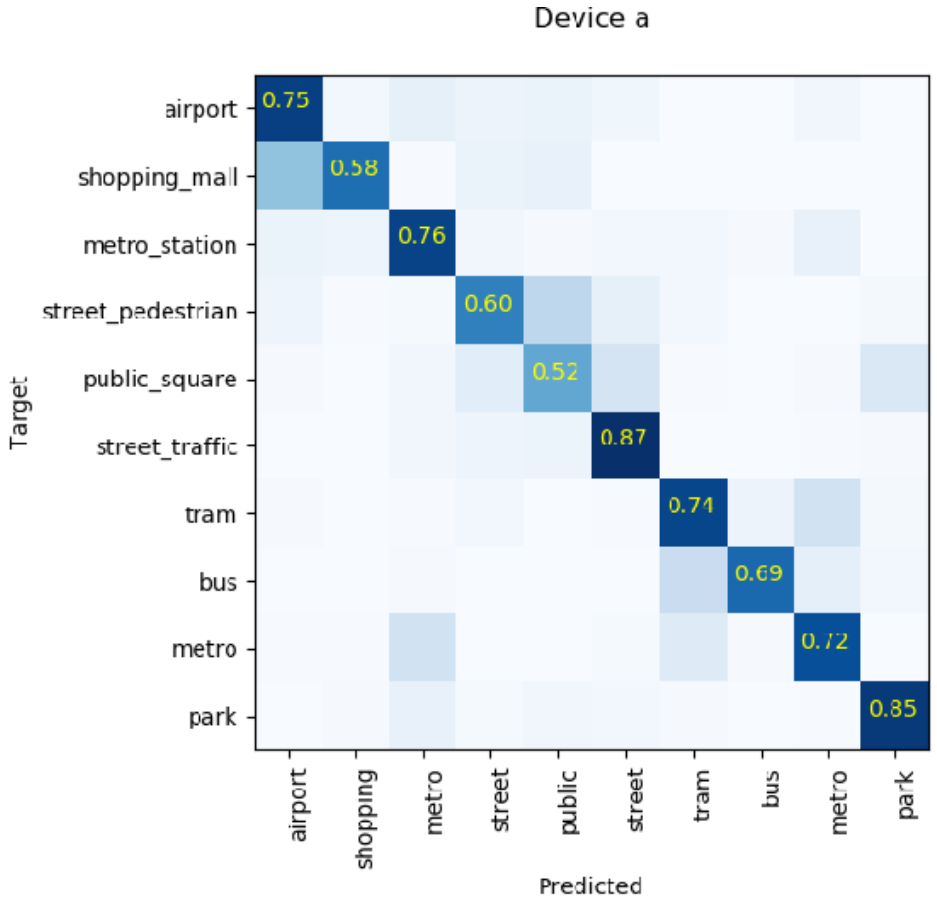


Attention网络层

- 当前主流Attention机制中的核心计算均为矩阵乘法
  - Transformer
  - BERT



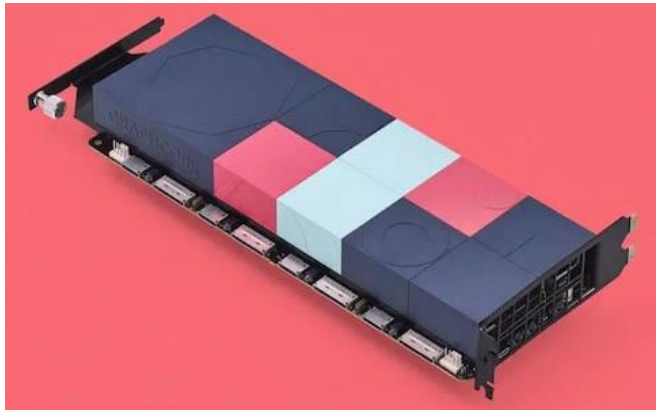


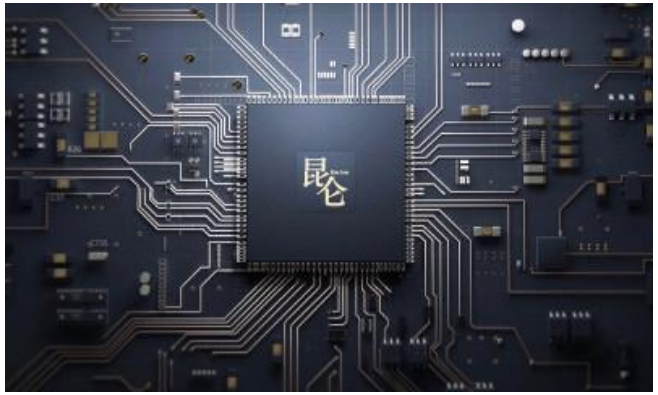


小结与思考

- 当前神经网络中使用的主流结构
  - 全连接层、卷积层、循环网络层、Attention结构
- 主流网络结构的计算均可表达成矩阵运算
  - 核心计算都可表达或转化成“**矩阵乘法**”
- 为什么神经网络结构要表示成矩阵计算呢
  - 矩阵计算表达出较好的计算并行性
  - 有成熟的矩阵计算加速硬件和软件库
    - GPU、CPU
    - MKL, CuBLAS
  - 鸡生蛋的问题

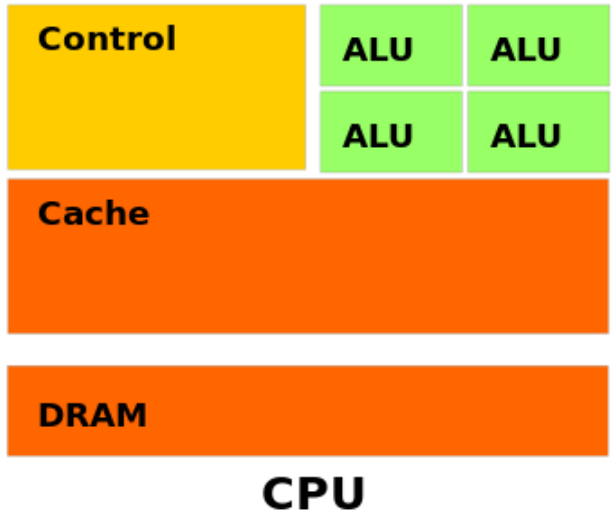
计算机体系结构与矩阵运算





CPU体系结构

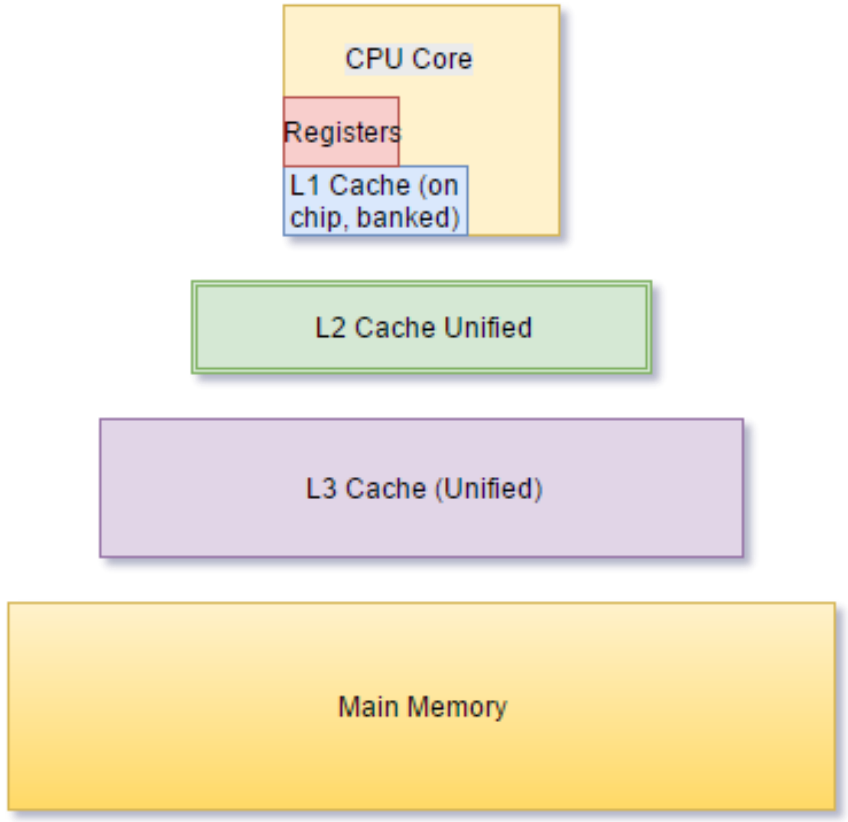
- 核心由复杂的控制单元和少量计算单元组成
  - 计算密度较低 – ALU较少
  - 控制逻辑复杂 – 较大的控制单元
- 主要面向顺序指令执行
  - 成熟的调度技术：如分支预测、推测执行、乱序执行等
  - 擅长处理单线程、控制密集型的计算任务
  - 缺点：低计算吞吐



CPU内存架构

- 采用较深的内存架构
  - 多层cache来隐藏访存延时
    - Register, L1, L2, L3, Main Memory
  - 调度上采用访存预取和各种较为成熟的预取预测机制

	Quantity	Speed	Cost
Registers	512 bytes	1	?
L1 cache	32 KB	2	?
L2 cache	512 KB	10	$200/MB
Main memory	32 MB	100	$50/MB

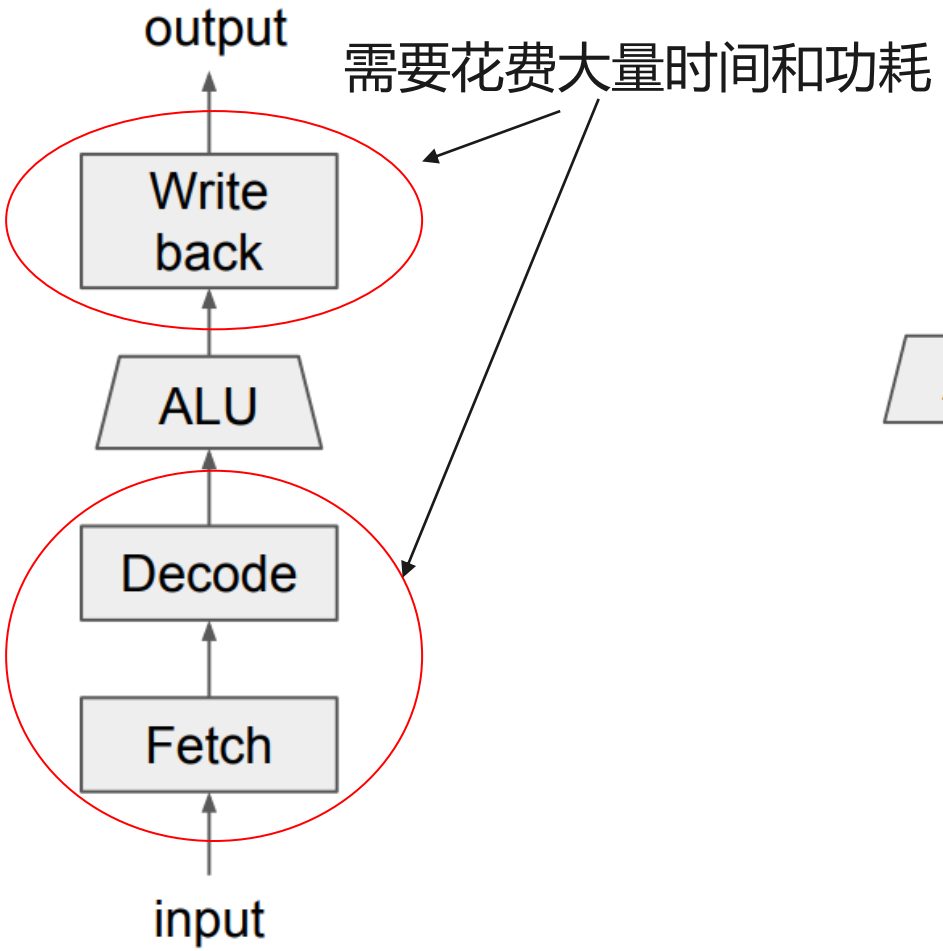


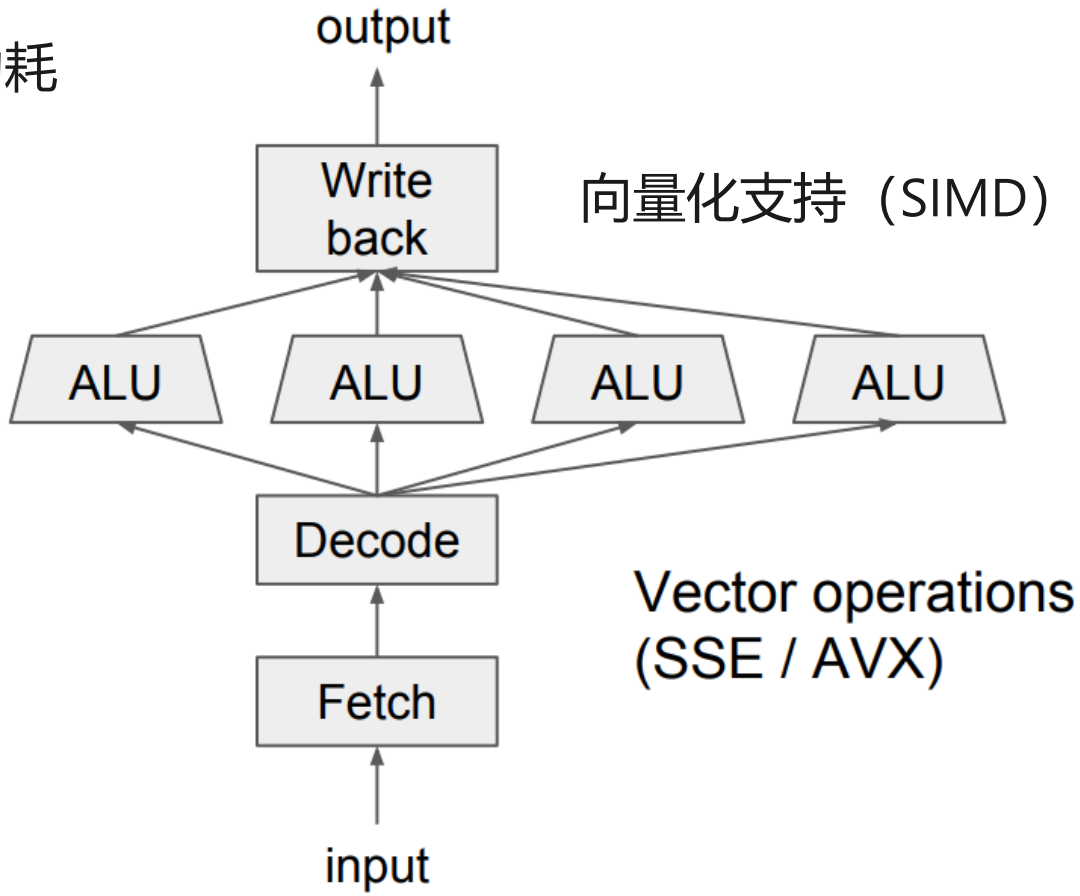
Source: https://web.stanford.edu/class/ee282h

CPU性能增长瓶颈

- 由于近年来CPU的发展达到了一些物理极限和由于功耗的限制，CPU的性能已经无法显著提升
- 新的性能提升方向：
  - **乱序执行互不依赖的指令：**当代大多CPU可以有限支持
  - **增加更多的计算核心：**依赖操作系统将应用程序高度到多核上；或者用户的程序中显示使用多线程进行计算；
  - **给单核增加向量化功能：**允许CPU在向量数据上执行相同的指令，需要用户程序中显示使用向量化批量来实现

CPU指令执行过程





Source: http://dlsys.cs.washington.edu/

SIMD (Single Instruction, Multiple Data)

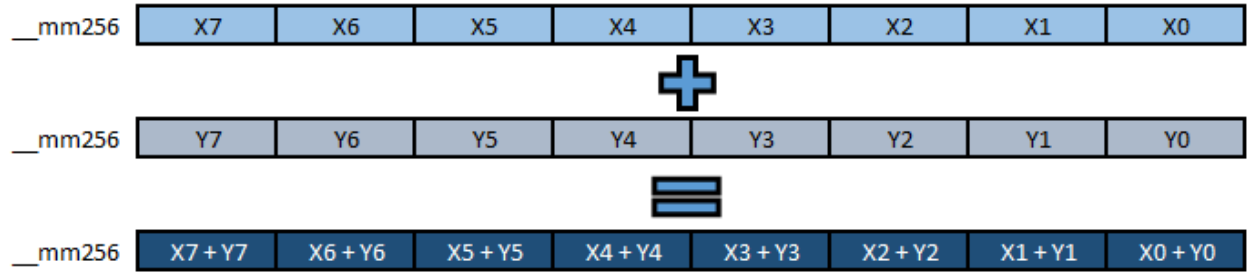
SSE Data Types (16 XMM Registers)

__m128	Float	Float	Float	Float	4x 32-bit float
__m128d	Double		Double		2x 64-bit double
__m128i	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	16x 8-bit byte
__m128i	short	short	short	short	short	short	short	short	8x 16-bit short
__m128i	int		int		int		int		4x 32bit integer
__m128i	long long				long long				2x 64bit long
__m128i	doublequadword								1x 128-bit quad

AVX Data Types (16 YMM Registers)

__mm256	Float	Float	Float	Float	Float	Float	Float	Float	8x 32-bit float
__mm256d	Double		Double		Double		Double		4x 64-bit double
__mm256i	*256-bit Integer registers. It behaves similarly to __m128i.Out of scope in AVX, useful on AVX2*								

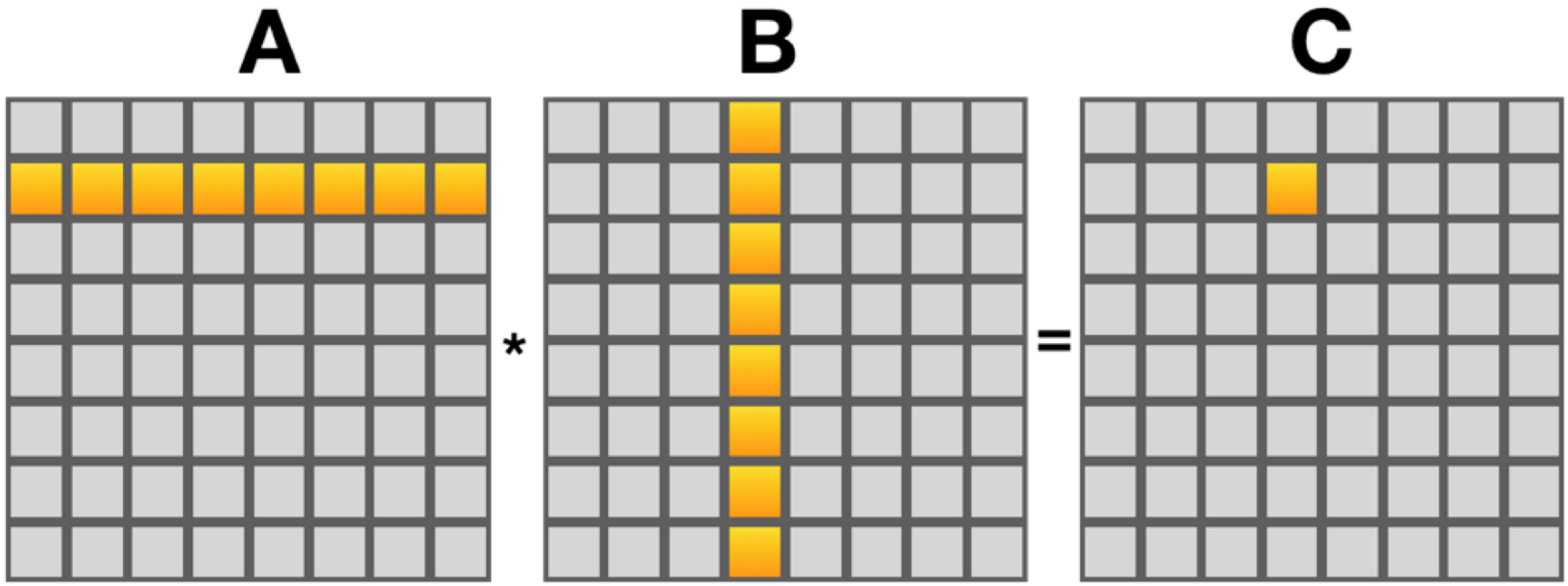
AVX Operation



https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx

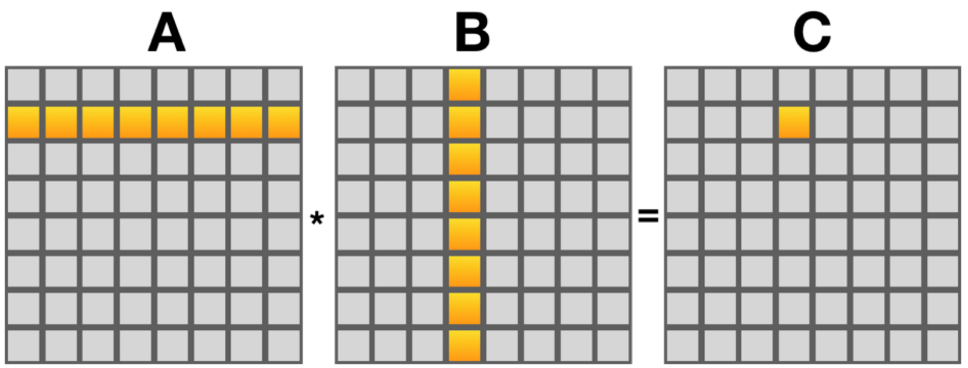
如何在CPU上高效地计算一个矩阵乘法?

- $C[m,n] = A[m,k] \times B[k,n]$
- $C[i,j] = \sum_{p=0}^{k} A[i,p] \times B[p,j]$



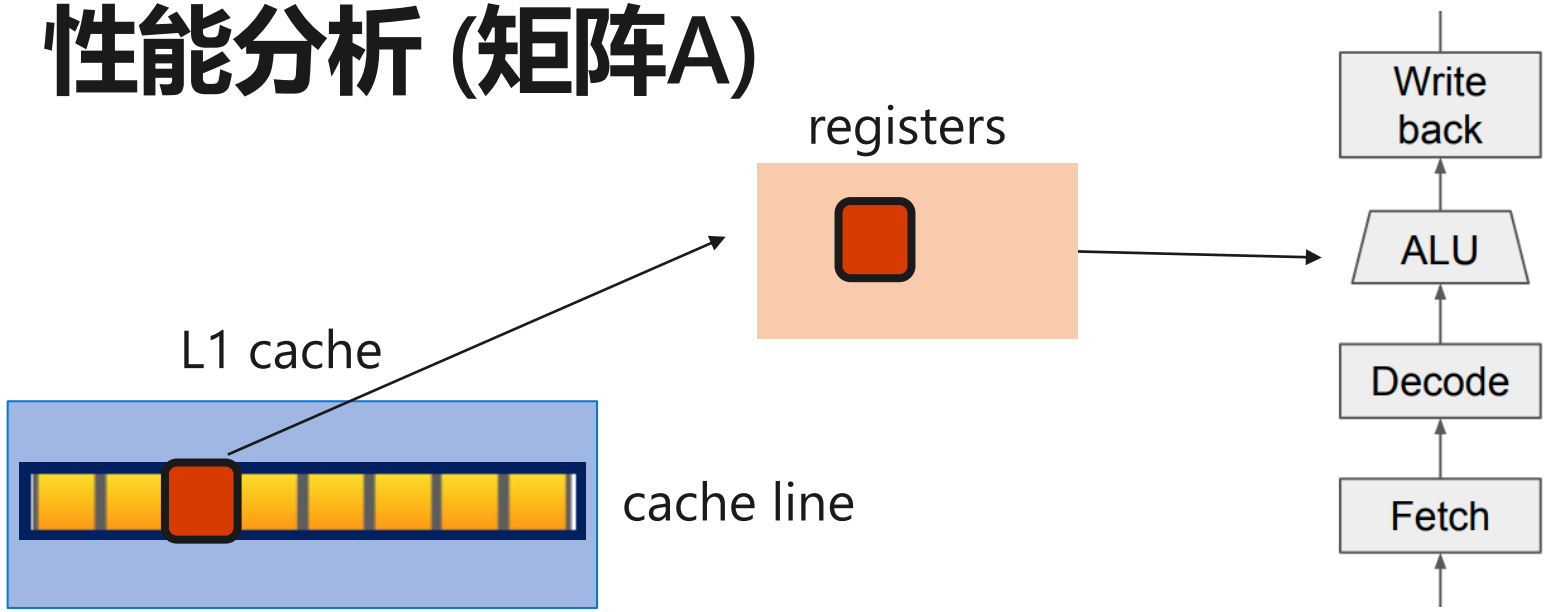
直观简洁的实现方法

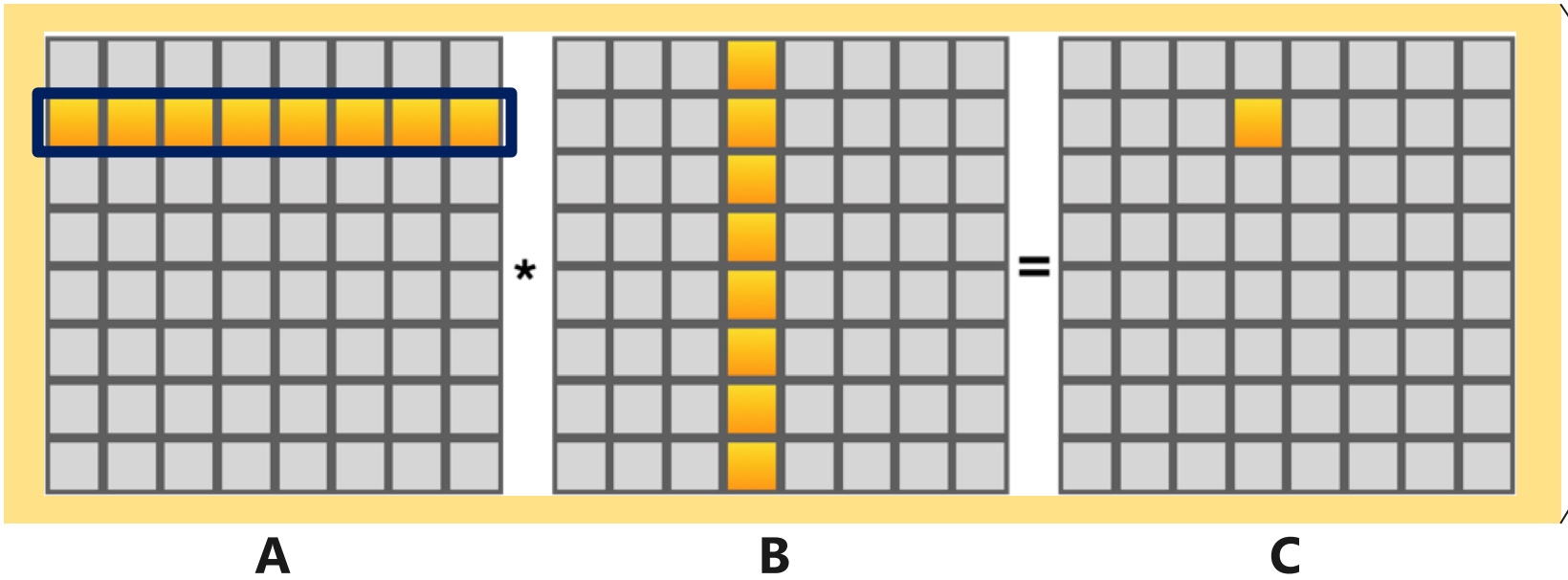
- $C[i,j] = \sum_{p=0}^{k} A[i,p] \times B[p,j]$
- 问题：这样的实现有什么缺点吗？

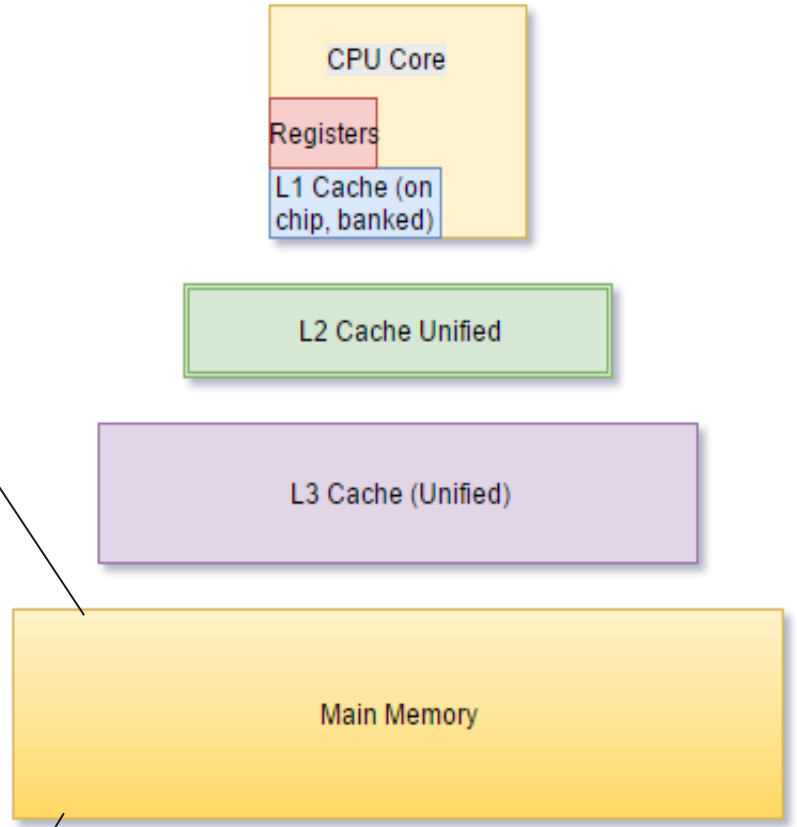



```
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    for (int p = 0; p < k; p++) {
      C(i, j) += A(i, p) * B(p, j);
    }
  }
}
```

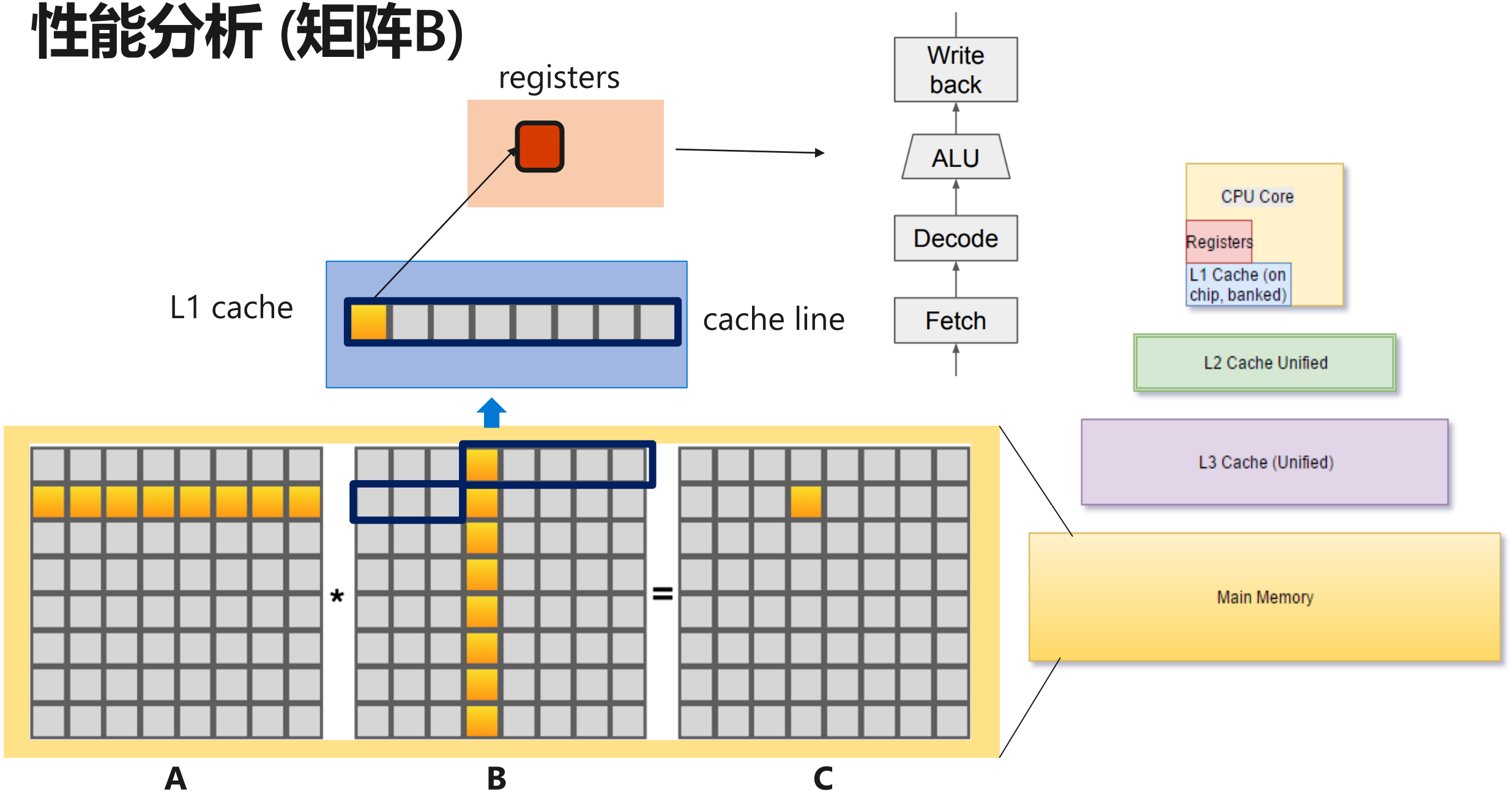
性能分析 (矩阵A)


registers
Write back
ALU
Decode
Fetch
L1 cache
cache line


*
=
A
B
C


CPU Core
Registers
L1 Cache (on chip, banked)
L2 Cache Unified
L3 Cache (Unified)
Main Memory

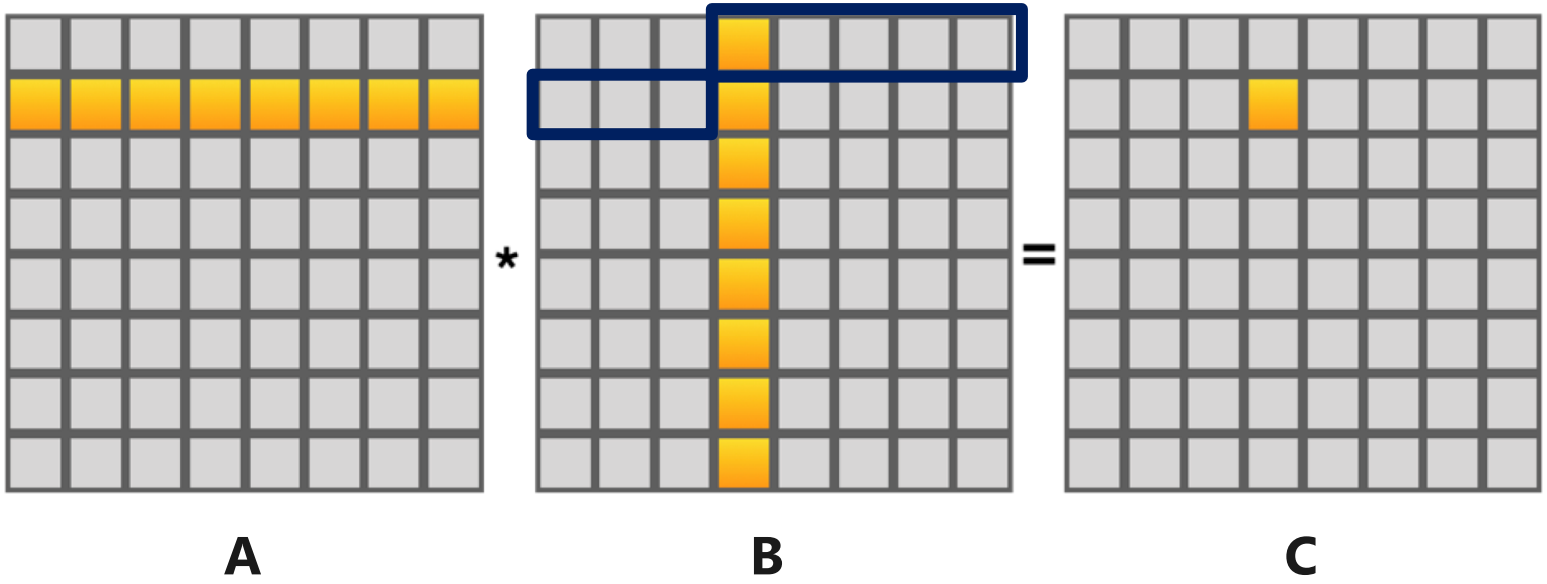
性能分析 (矩阵B)


registers
L1 cache
cache line
Write back
ALU
Decode
Fetch
CPU Core
Registers
L1 Cache (on chip, banked)
L2 Cache Unified
L3 Cache (Unified)
Main Memory
*
=
A
B
C

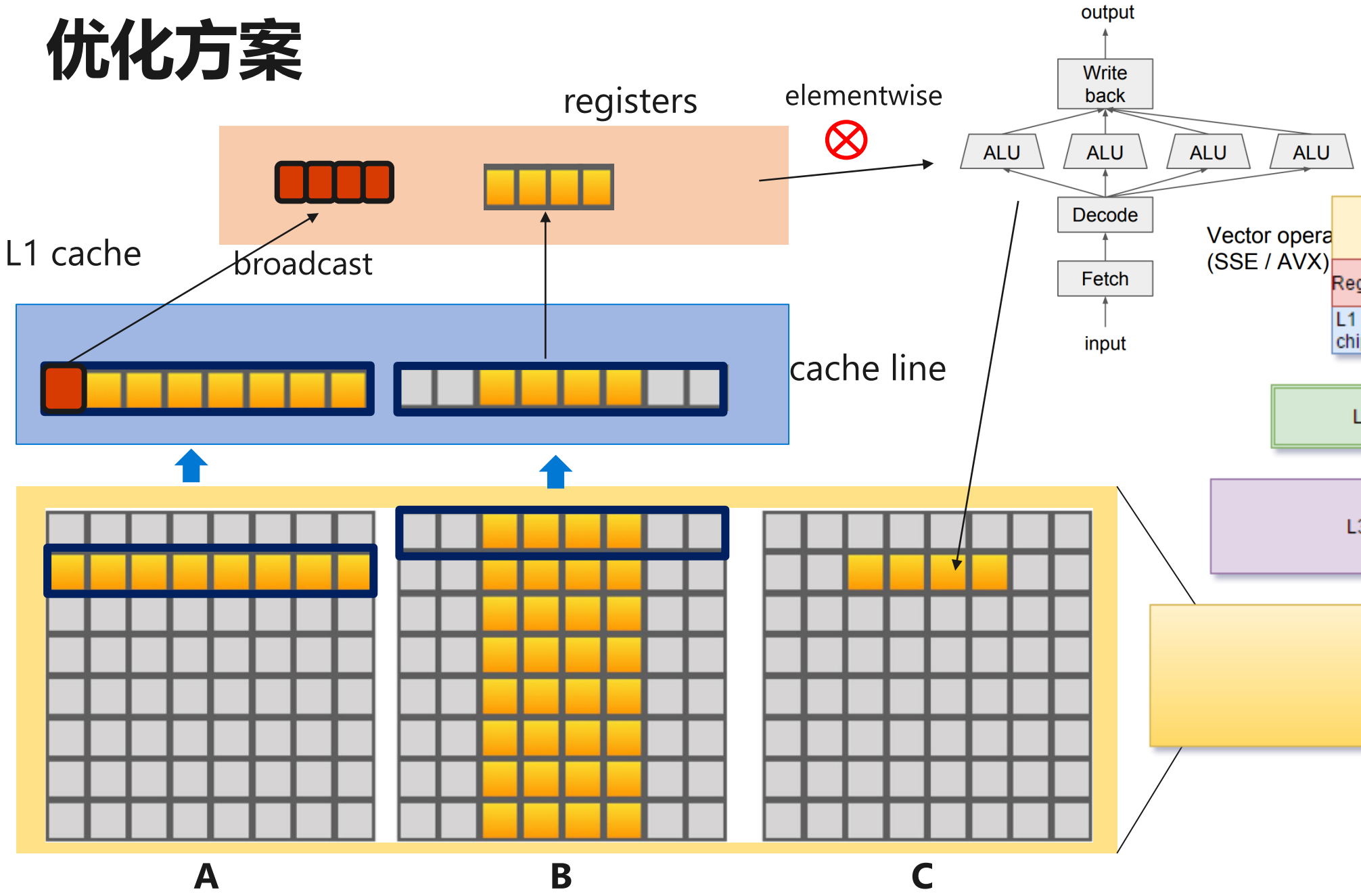
优化方案

- 针对矩阵A计算低效问题→ 使用向量化指令增加计算吞吐
- 针对矩阵B访存抵消问题→ 增加cache利用效率

问题：将B矩阵转置是否可行?

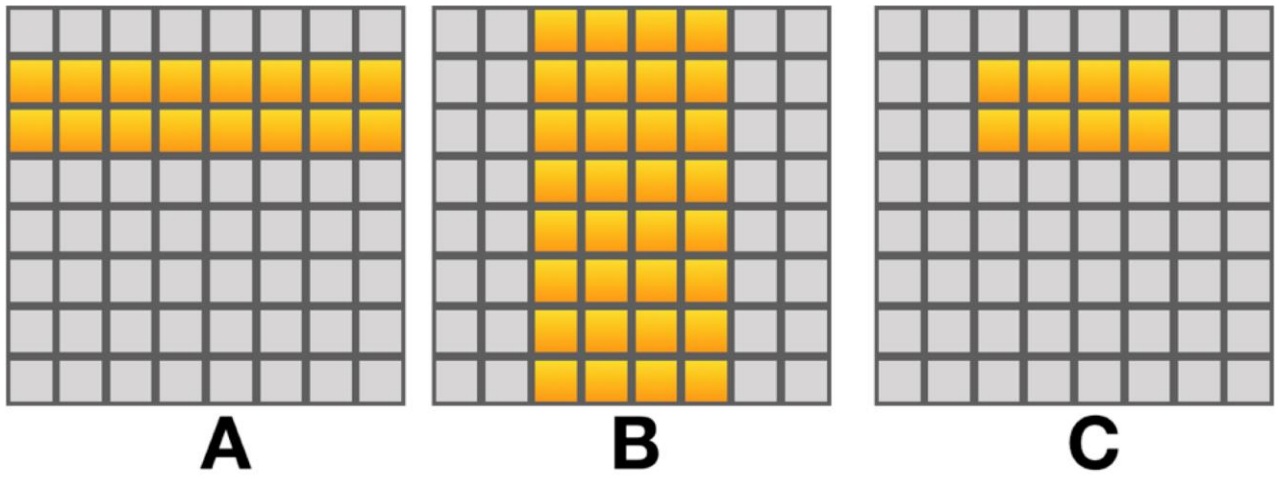


优化方案


registers
elementwise
output
Write back
ALU
ALU
ALU
ALU
Decode
Fetch
input
L1 cache
broadcast
cache line
Vector opera
(SSE / AVX)
CPU Core
Registers
L1 Cache (on chip, banked)
L2 Cache Unified
L3 Cache (Unified)
Main Memory
A
B
C

优化方案

- 考虑到register的重用，实际中往往需要将A和B矩阵划分成合适大小的块，使得最终的访问性价比最高



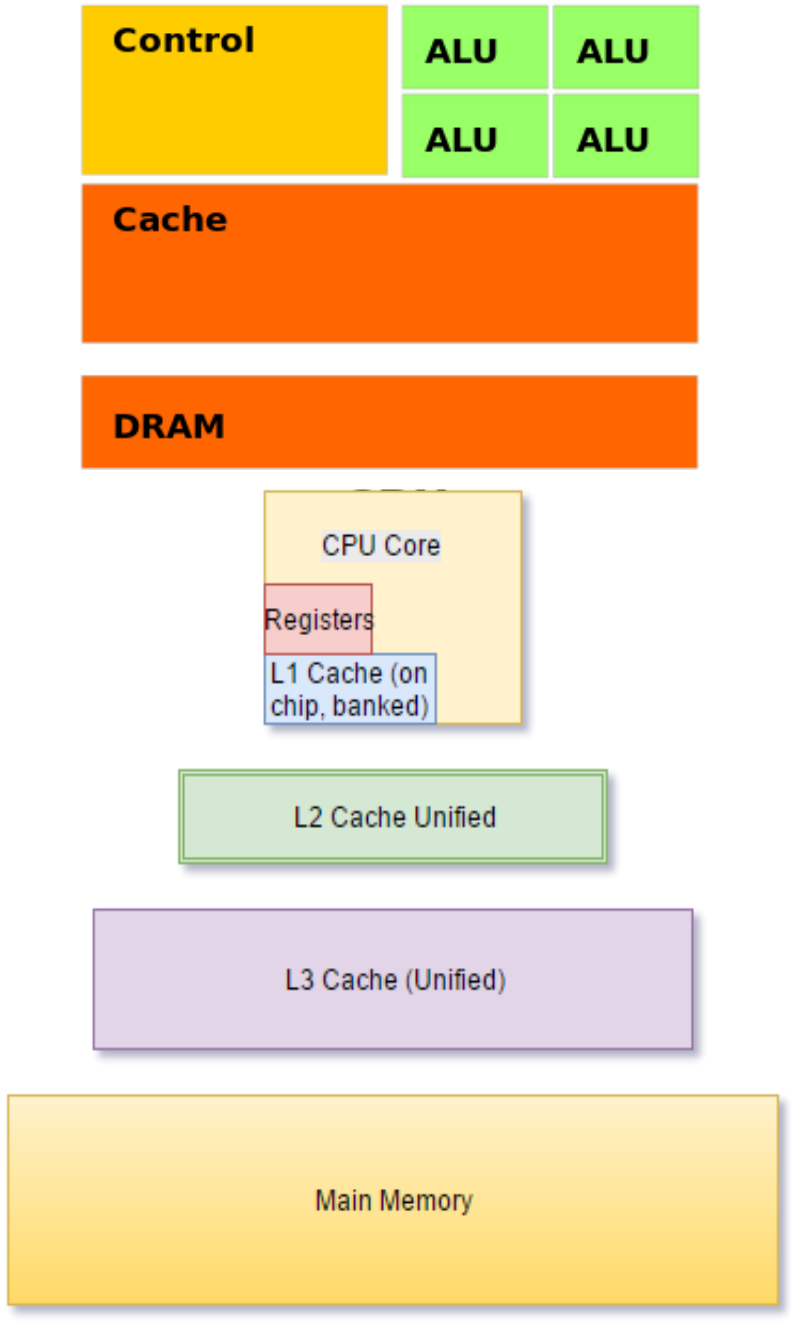

```
float8 csum[regsA][regsB] = {{0.0}};
for (int p = 0; p < k; p++) {

  // Perform the DOT product.
  for (int bi = 0; bi < regsB; bi++) {
    float8 bb = LoadFloat8(&B(p, bi * 8));
    for (int ai = 0; ai < regsA; ai++) {
      float8 aa = BroadcastFloat8(A(ai, p));
      csum[ai][bi] += aa * bb;
    }
  }
}

// Accumulate the results into C.
for (int ai = 0; ai < regsA; ai++) {
  for (int bi = 0; bi < regsB; bi++) {
    AdduFloat8(&C(ai, bi * 8), csum[ai][bi]);
  }
}
```

然而，实际的优化远比这些复杂...

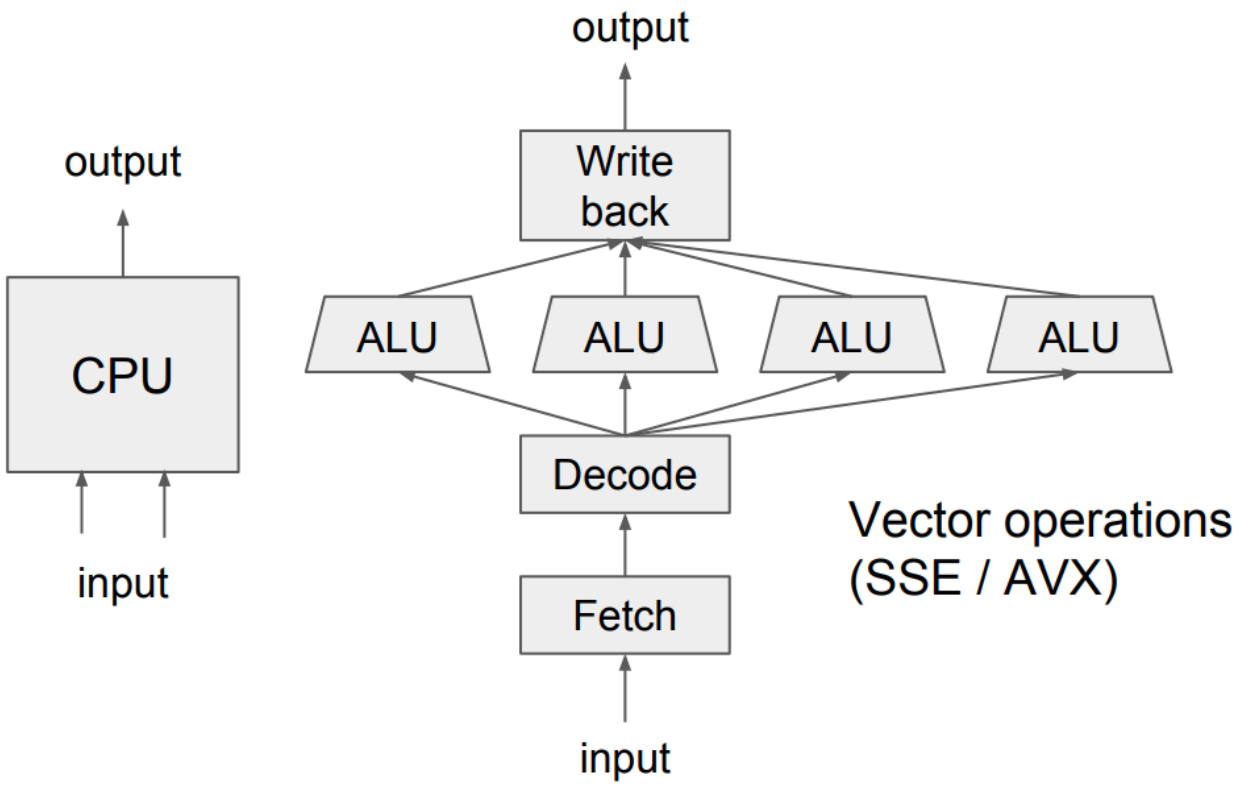
- 其它需要考虑的优化
  - 考虑多核并行计算
  - 多个FMA调度端口
  - 流水线并行
  - Tiling
  - AVX/SSE指令的使用
  - ...
- 所幸的是
  - BLAS 库
  - Intel Math Kernel Library （MKL)

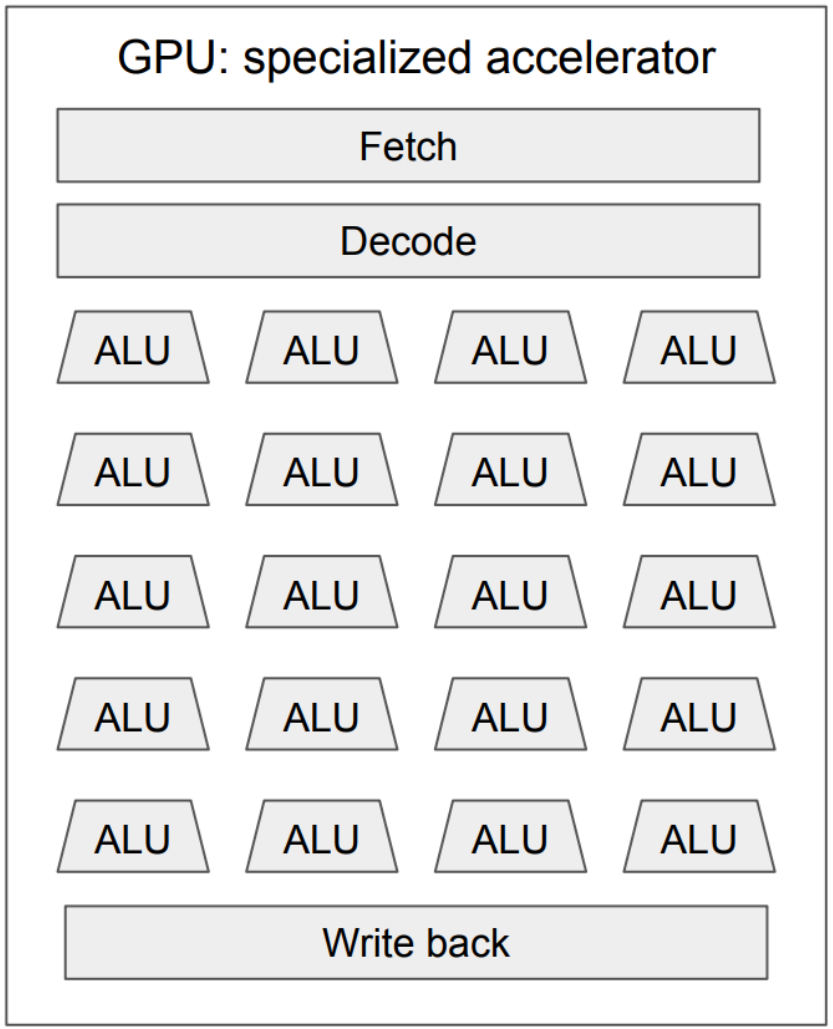
小结：CPU与矩阵运算

- CPU核心的组成
- 计算单元与控制单元
- 内存架构
- CPU上高效矩阵计算



GPU体系结构



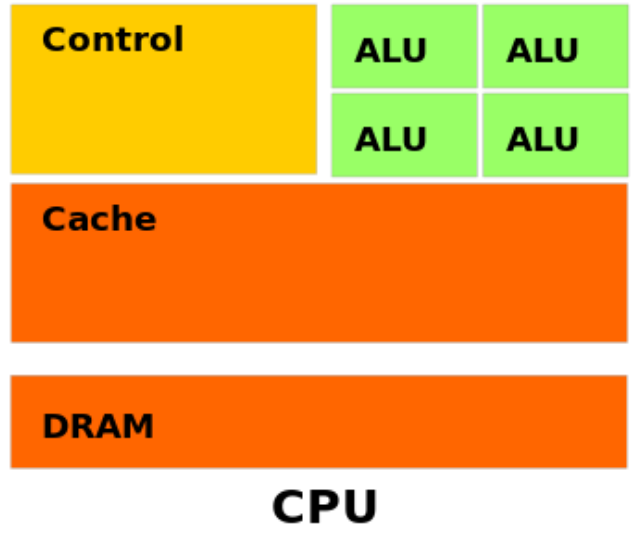


Source: http://dlsys.cs.washington.edu/

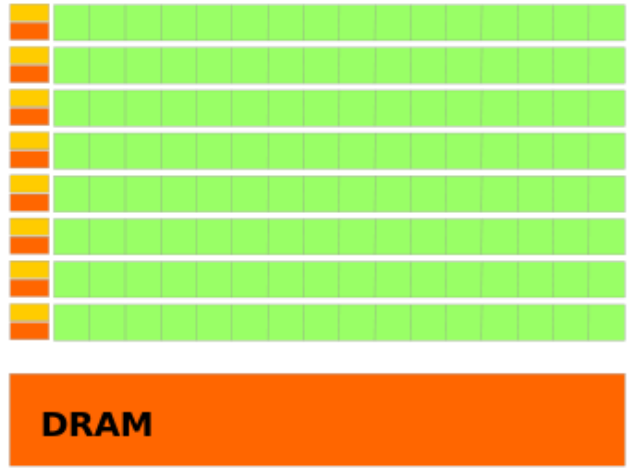
GPU体系结构

- 由上千个简单的核心（core）组成
  - 每个Core的结构非常简单（与CPU相比）
  - **不支持**分支预测、推测执行、乱序执行等

- GPU特点
  - 较高的计算密度
  - 计算与访存比较高
  - 擅长处理高度并行的计算：如图像处理，矩阵计算
  - 早期主要用与显示处理
  - 缺点：不擅长处理控制逻辑复杂的程序



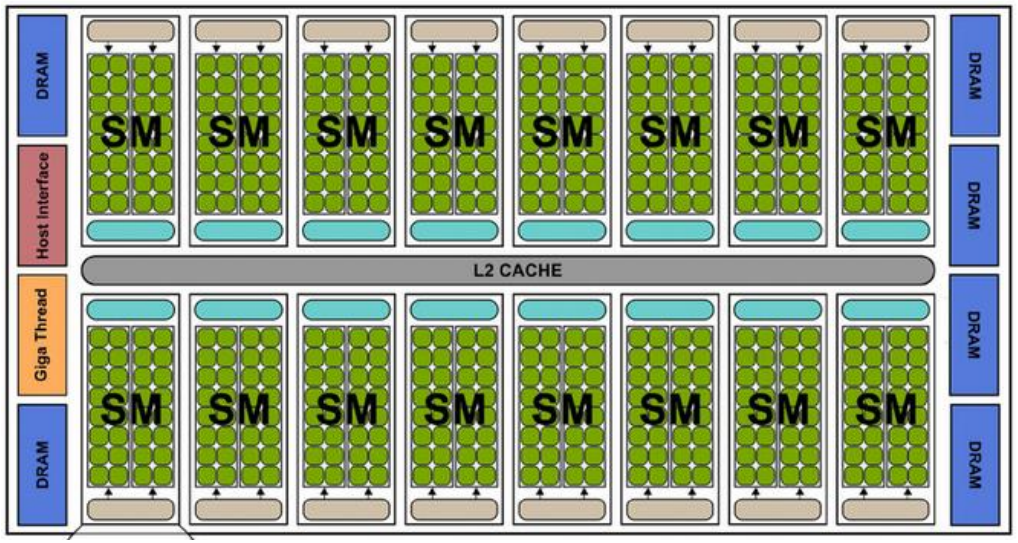
CPU

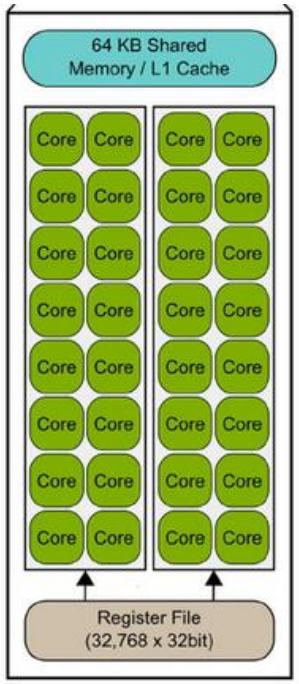


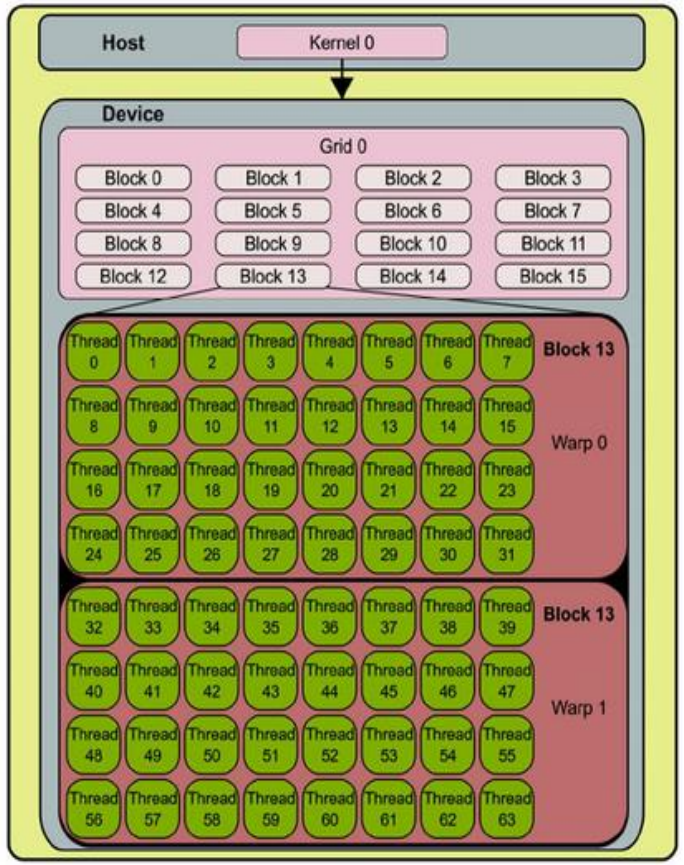
GPU

GPU执行模型

- SIMT（Single Instruction, Multiple Threads）
  - 将一组Cores组织成一个cluster
  - 在同一时间这些cores都执行相同的指令
  - 32个线程为一组构成warp, 以warp为单位调度到cores上
  - 一个warp内的所有线程执行相同指令，但操作在不同的数据上

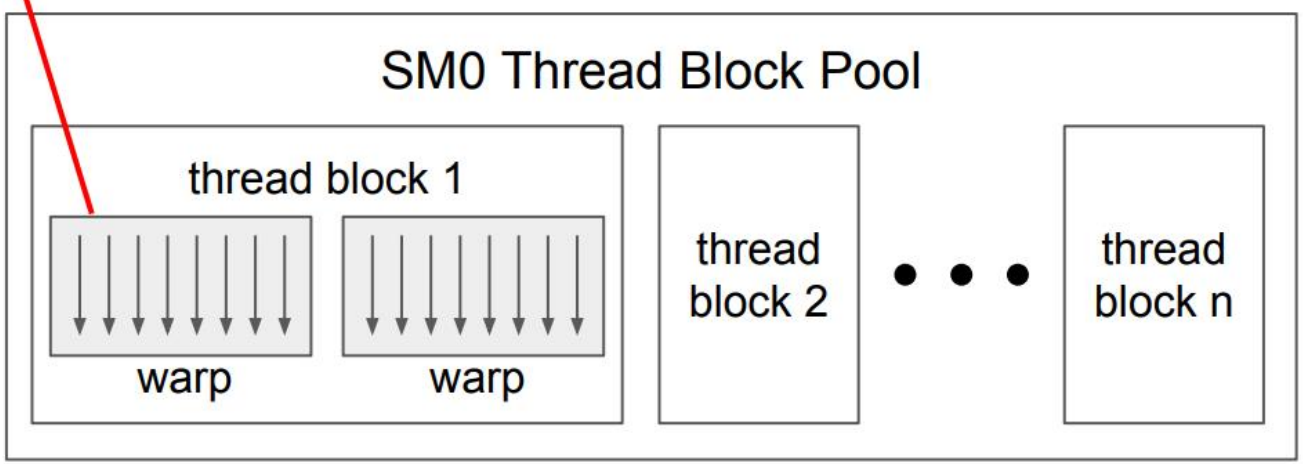




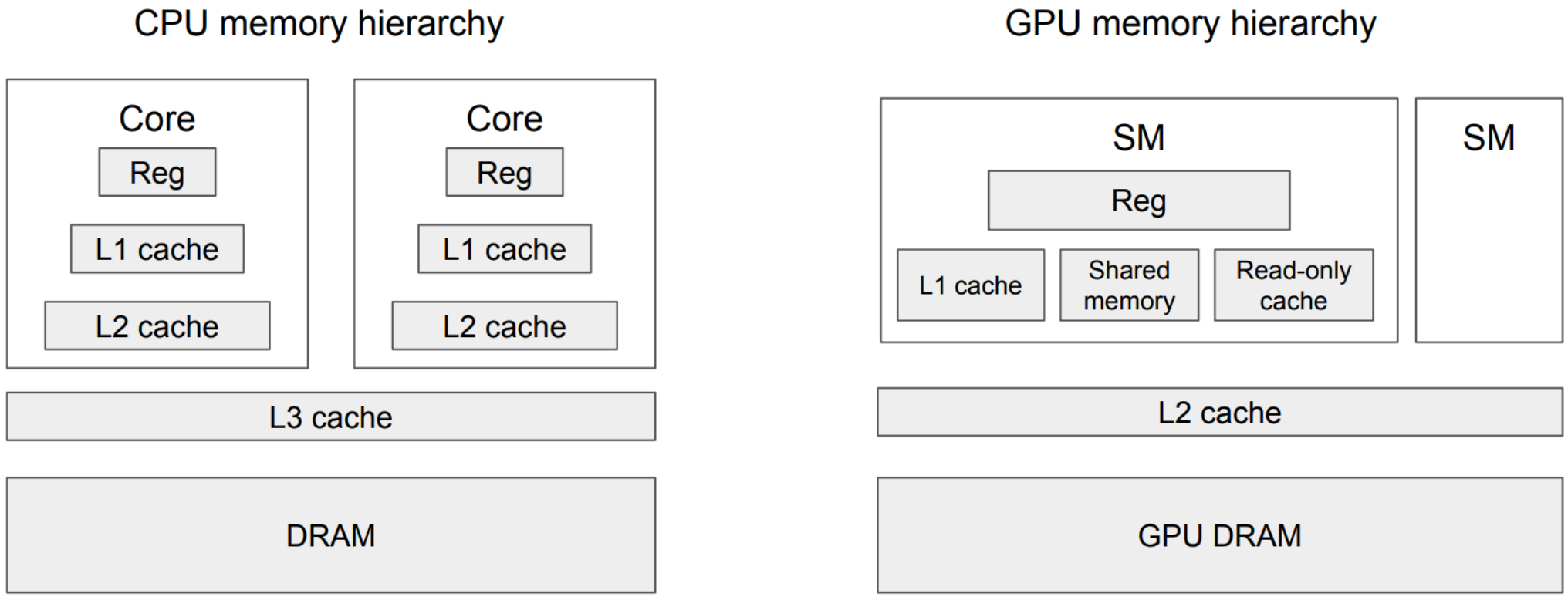


Kernel Execution

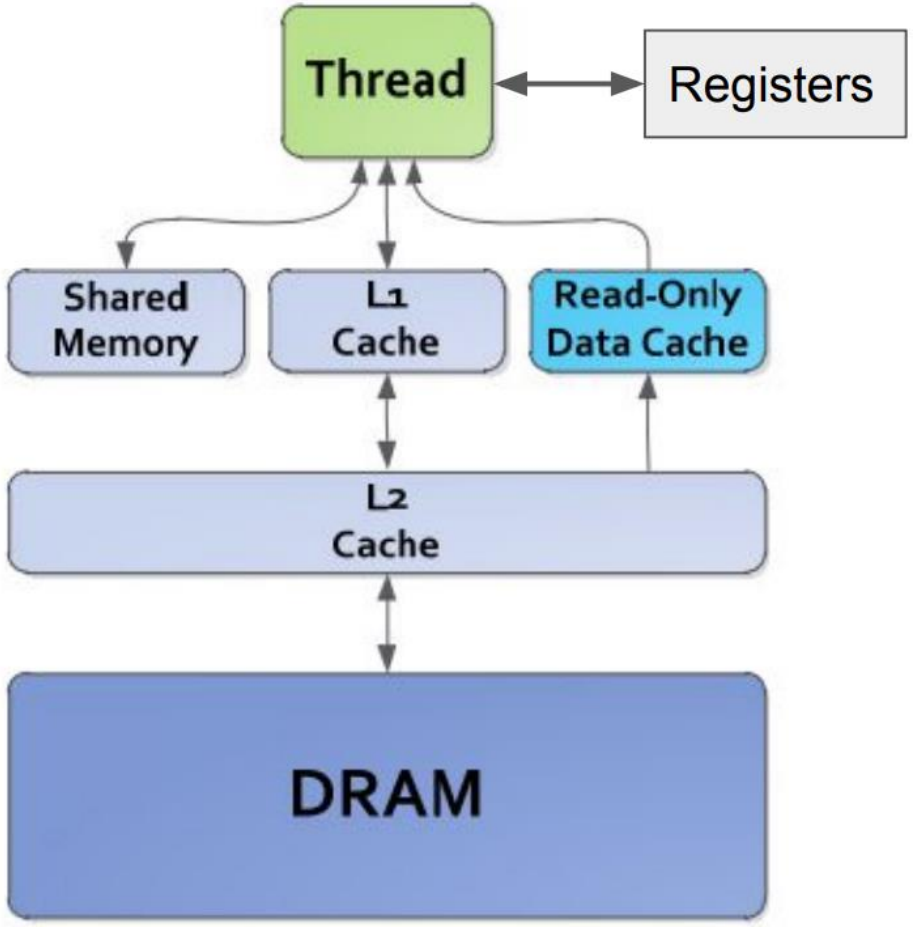
- A warp consists of 32 threads
  - A warp is the basic schedule unit in kernel execution.
- A thread block consists of 32-thread warps.
- Each cycle, a warp scheduler selects one ready warps and dispatches the warps to CUDA cores to execute.



GPU内存架构


CPU memory hierarchy
Core
Reg
L1 cache
L2 cache
Core
Reg
L1 cache
L2 cache
L3 cache
DRAM
GPU memory hierarchy
SM
Reg
L1 cache
Shared memory
Read-only cache
SM
L2 cache
GPU DRAM

GPU内存访问延迟



Registers: R 0 cycle / R-after-W ~20 cycles

L1/texture cache: 92 cycles
Shared memory: 28 cycles
Constant L1 cache: 28 cycles

L2 cache: 200 cycles

DRAM: 350 cycles

(for Nvidia Maxwell architecture)

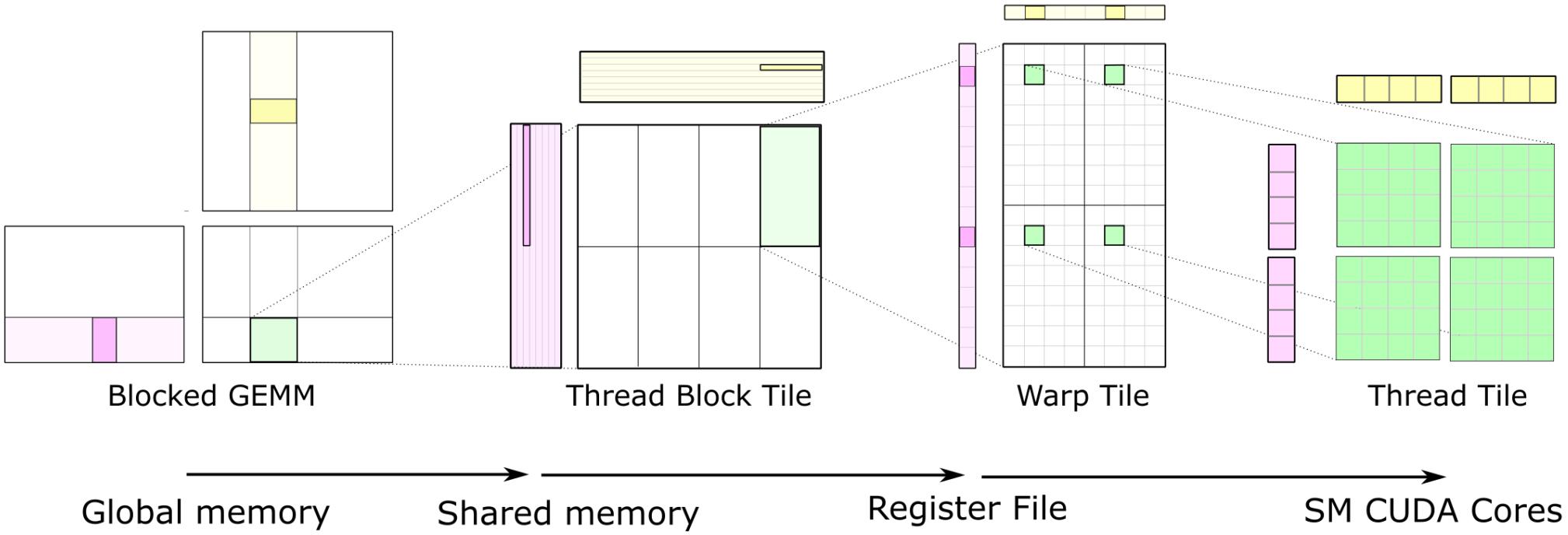
http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf

GPU结构小结

- GPU擅长处理在规则、稠密的数据上的高数据并行计算任务
- 这该类任务上，GPU比CPU可获得更高的能效比
  - 大量的算术运算单元
  - 大量Core共享指令解码单元和控制单元

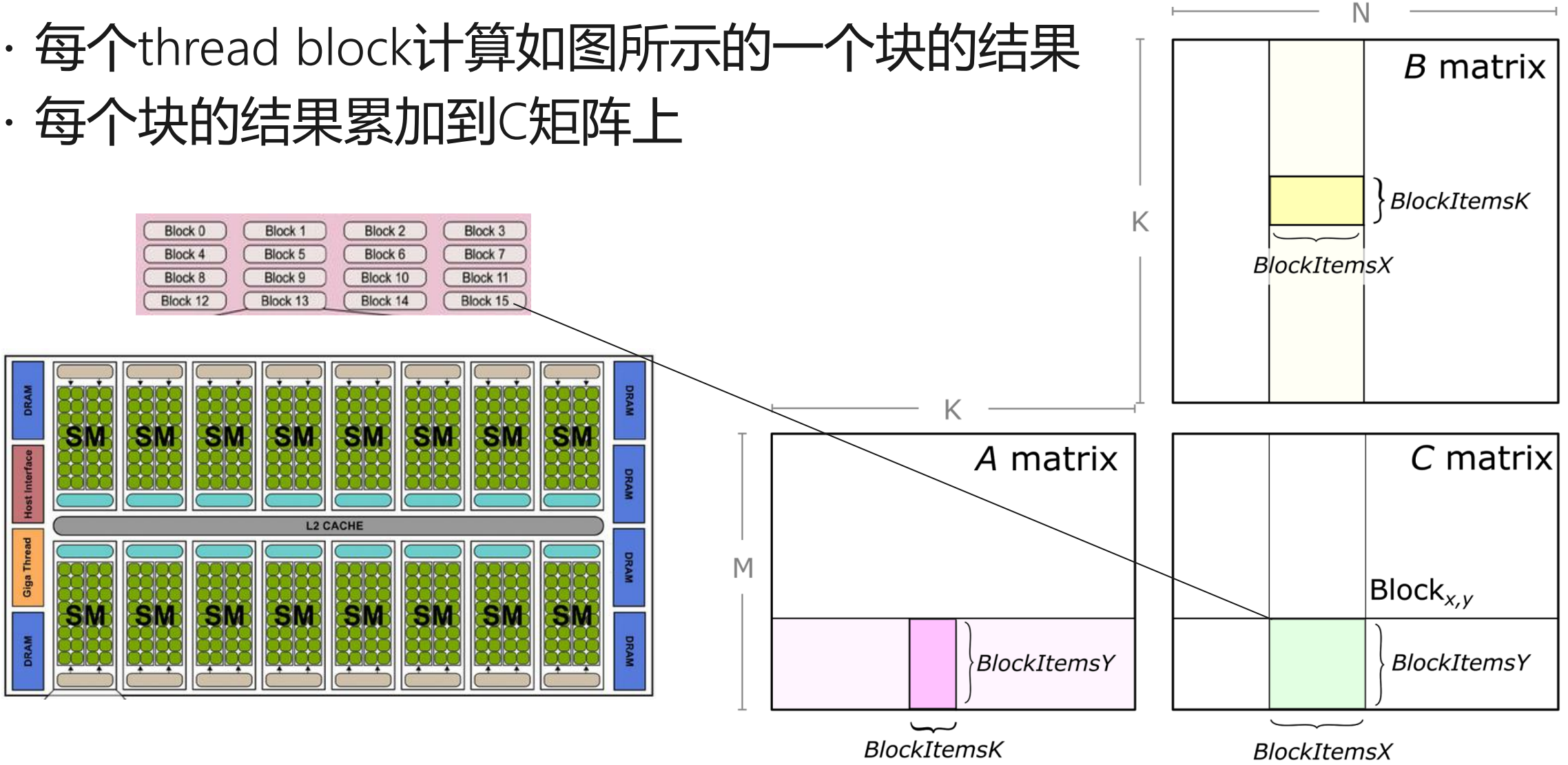
如何在GPU上高效地计算一个矩阵乘法?

- 提高访存计算比：从global memory到register的每层尽可能多的复用数据



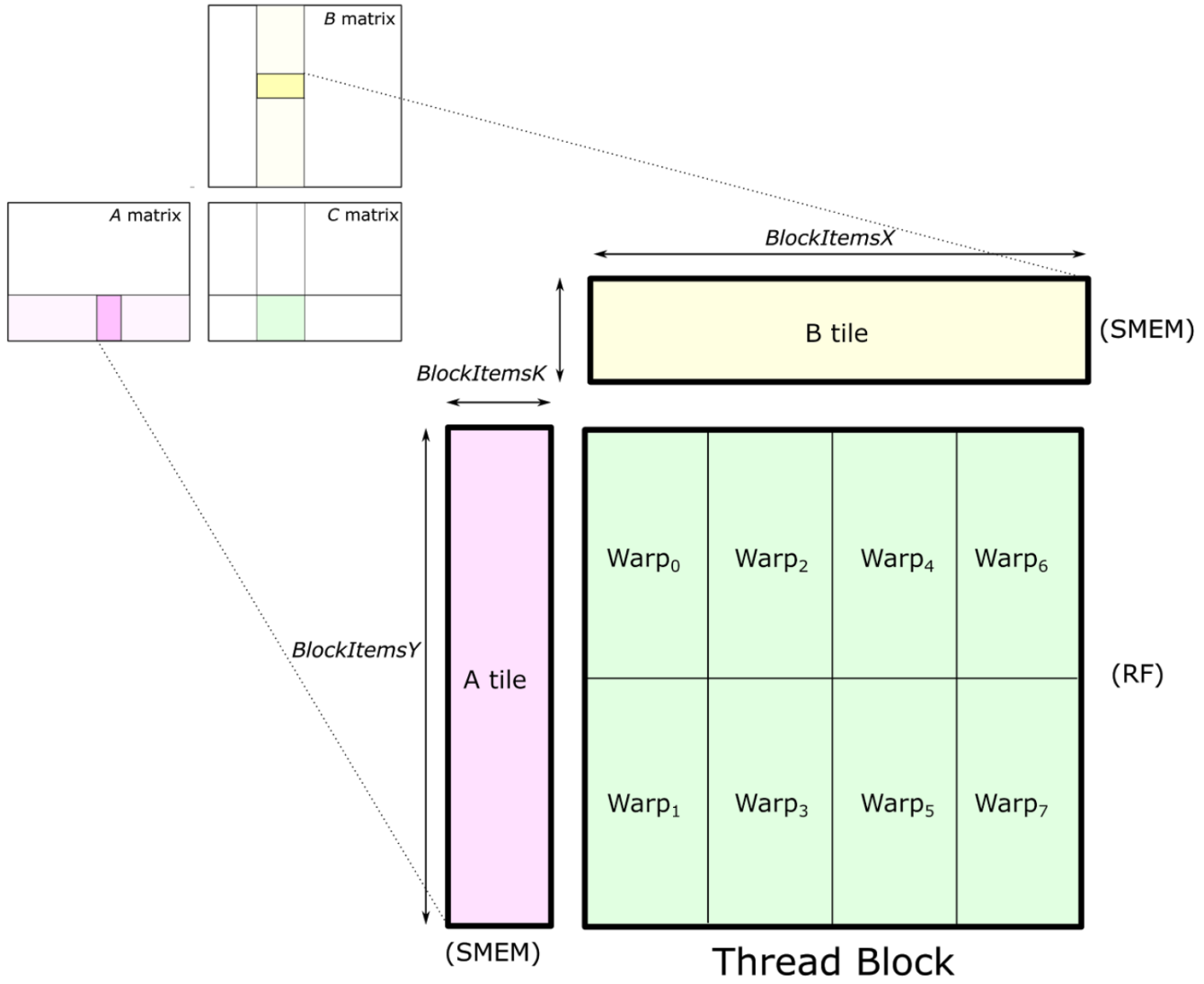
Thread Block Tile

- 每个thread block计算如图所示的一个块的结果
- 每个块的结果累加到C矩阵上



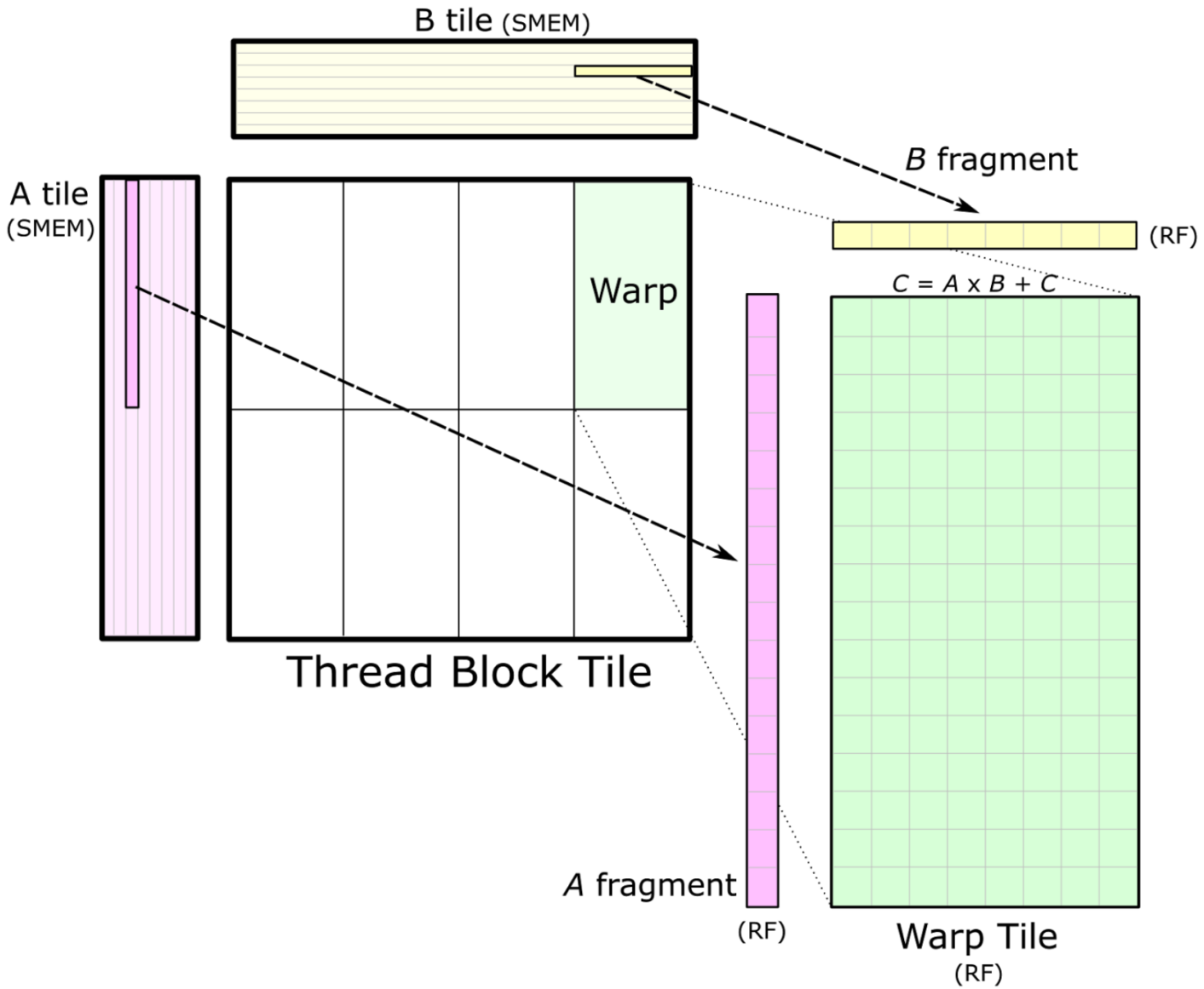
Warp Tile

- 每个thread-block中进一步划分成warps
- Tile A和B从memory中load到shared memory中
- 该SM中的所有warps都可以读取A和B
- A和B的结果需要不断累加，因此放到寄存器文件中



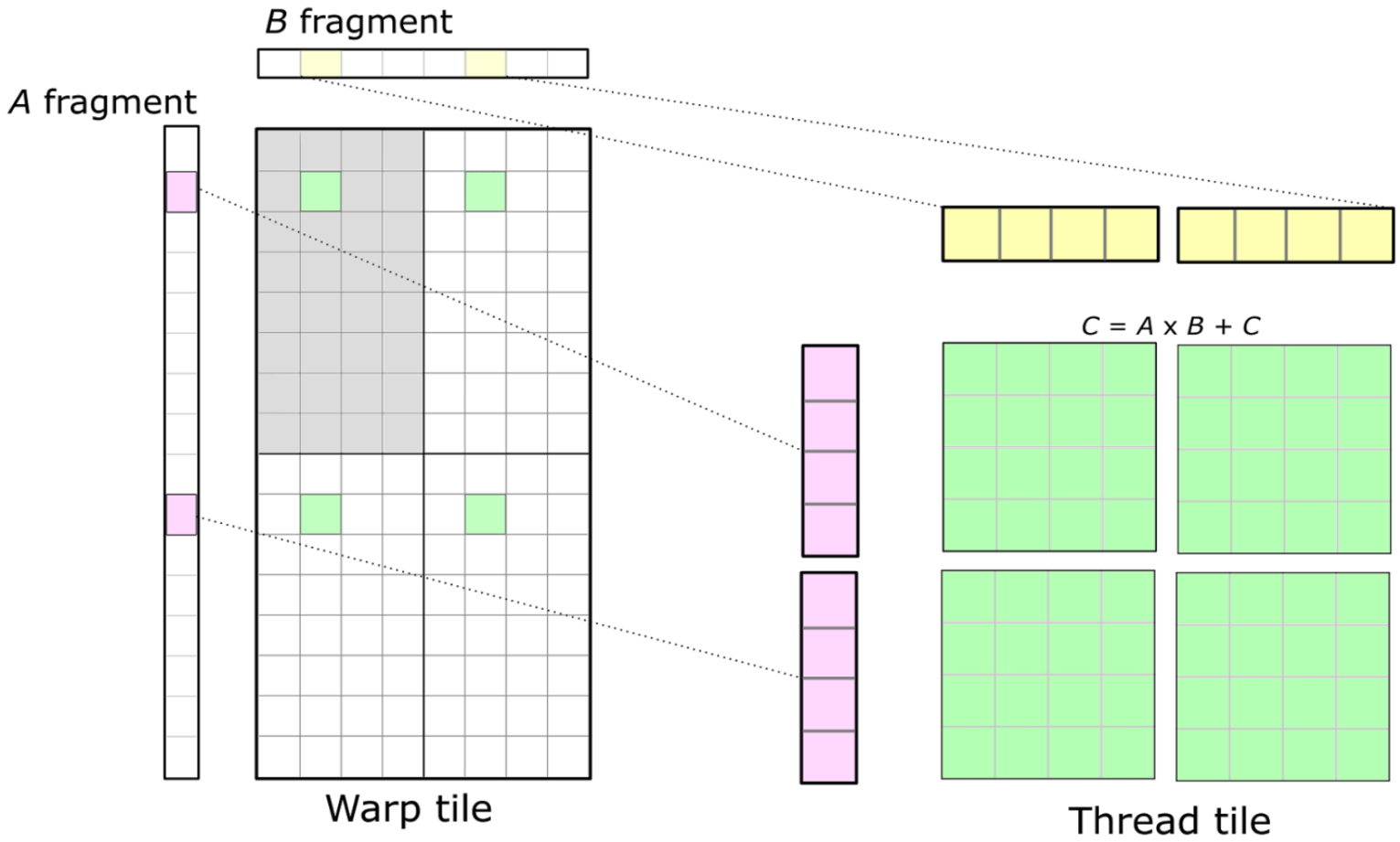
Warp Tile

- 每个warp负责tile A和B中一个分片的计算
  - 从shared memory中读取对应分片到寄存器文件中
  - 通过累加矩阵乘法计算C的每一个元素



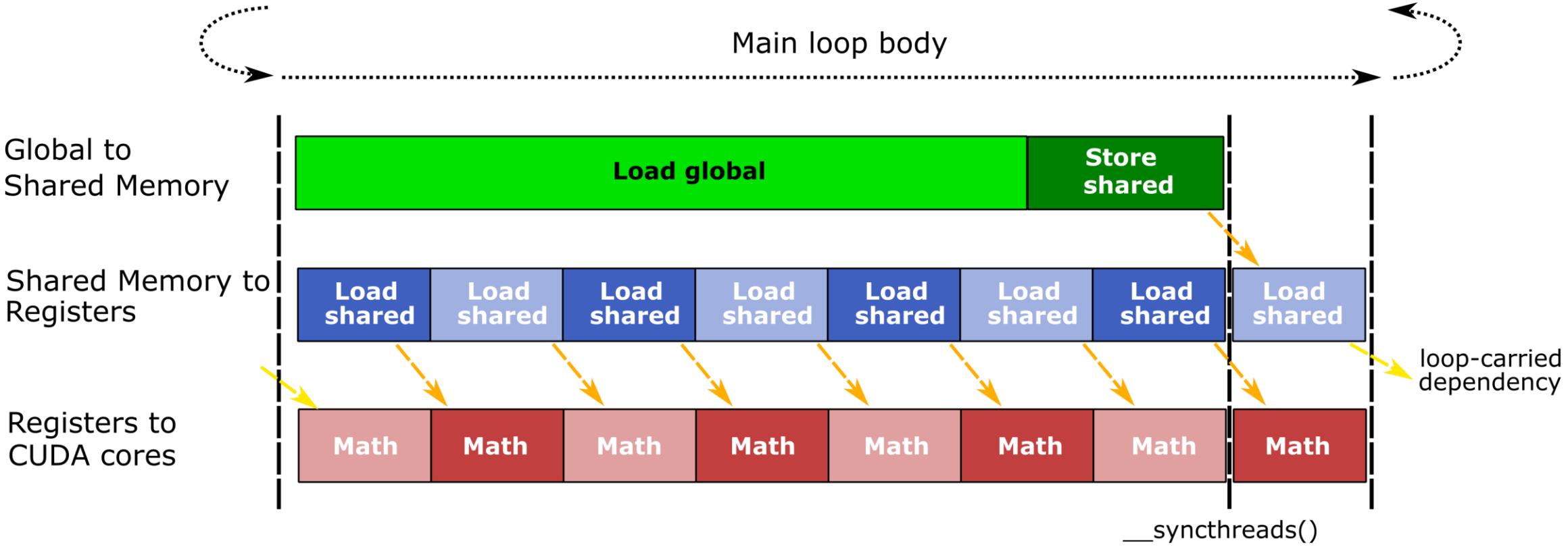
Thread Tile

- 由于线程间无法互相访问寄存器文件，因此每个线程应尽可能将寄存器的读取的数据进行重复使用



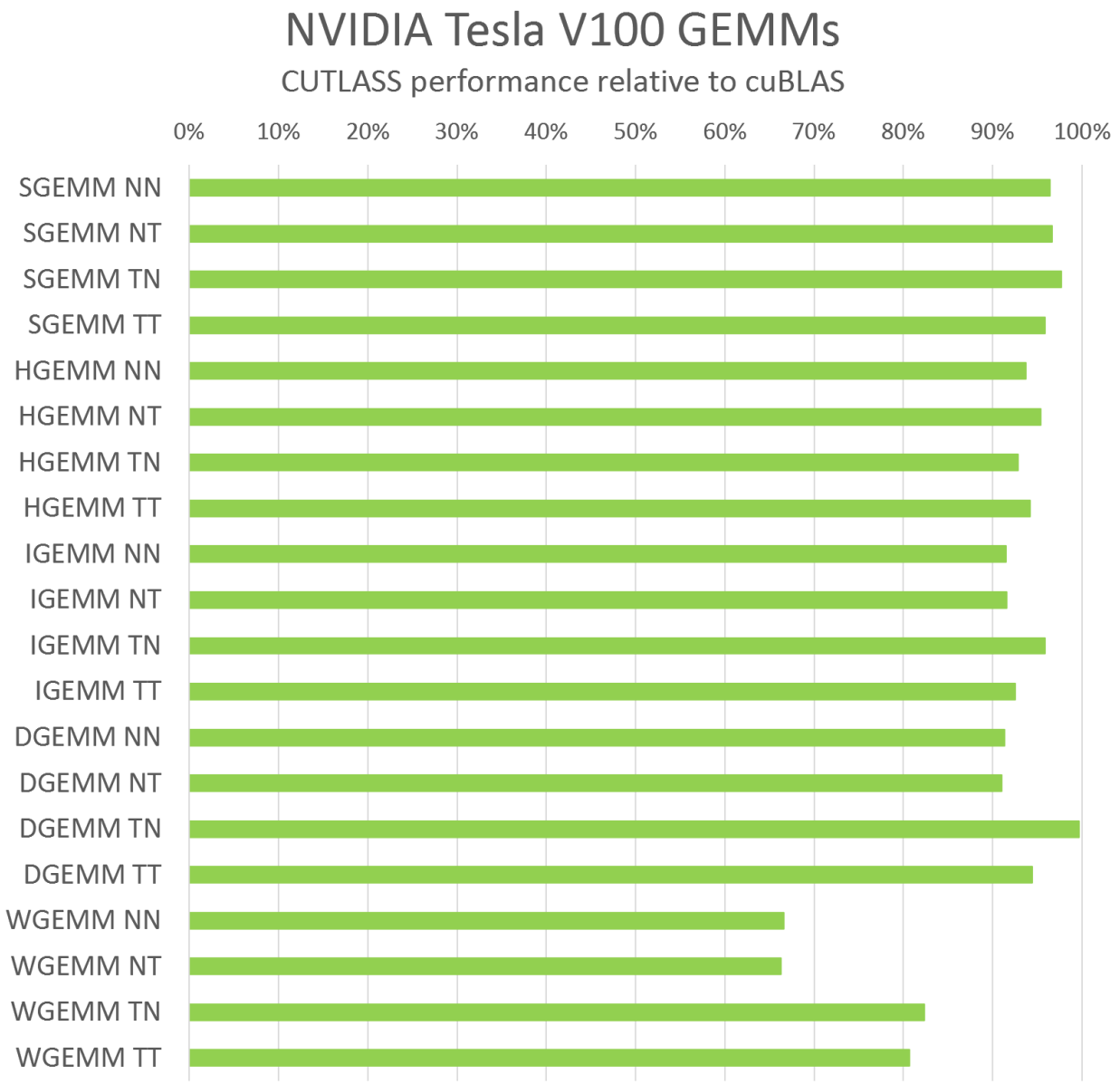
软件流水线

- 由于大量使用寄存器使得并发运行的blocks较少，访存计算比较低
- 采用软件流水线的方法隐藏访存延时



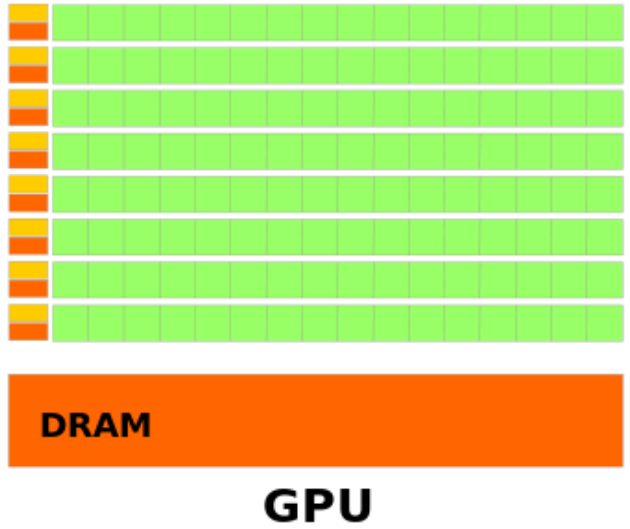
性能

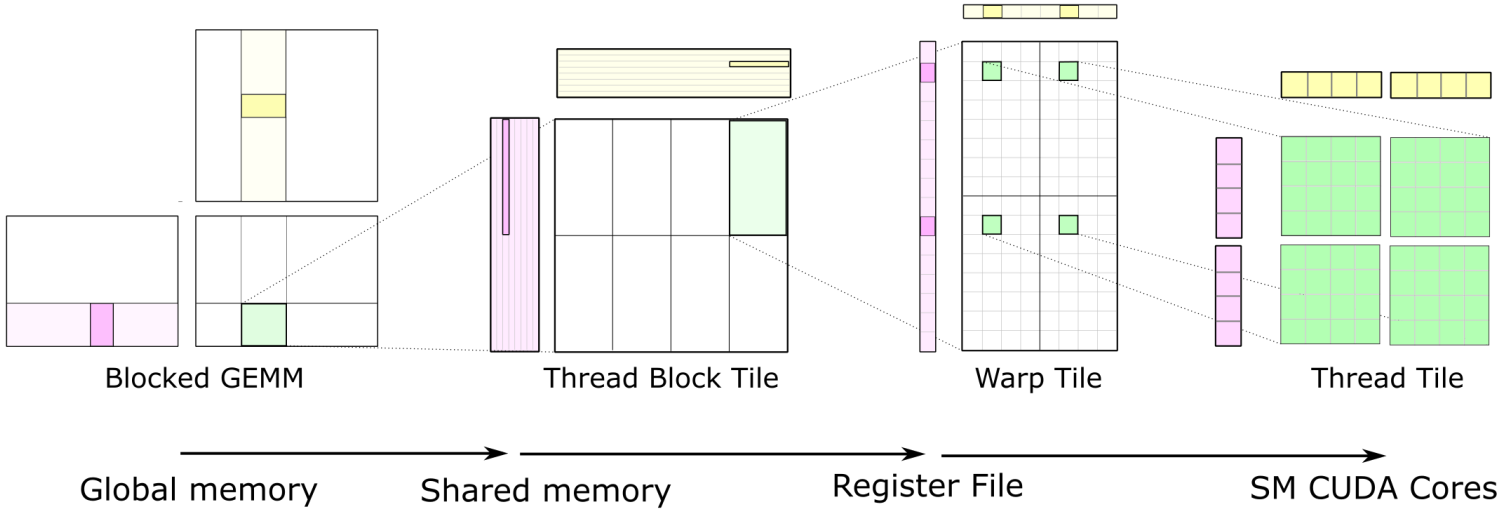
- 达到90%左右cublas性能
- 更多的优化
  - 如WMMA 指令
  - 寄存器bank冲突控制
  - 内存bank冲突控制
  - …
- 实际中
  - cuBLAS可提高较高的性能
  - 更多请阅读：https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/



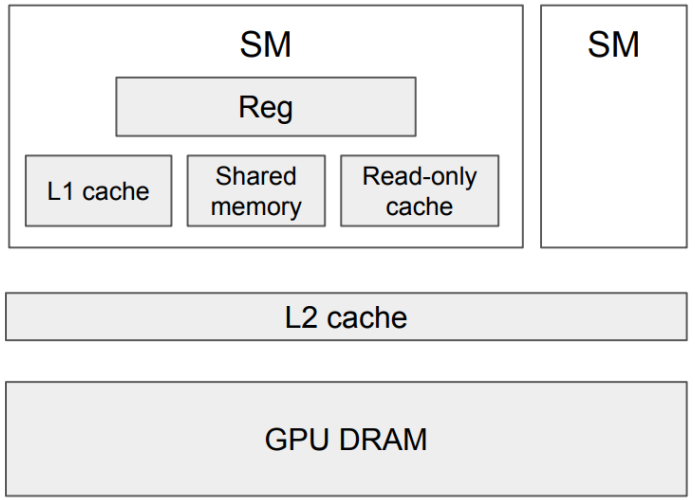
小结

- GPU体系结构
- GPU内存层次结构
- GPU执行模型
- 如何在GPU中实现高效矩阵乘法





GPU memory hierarchy

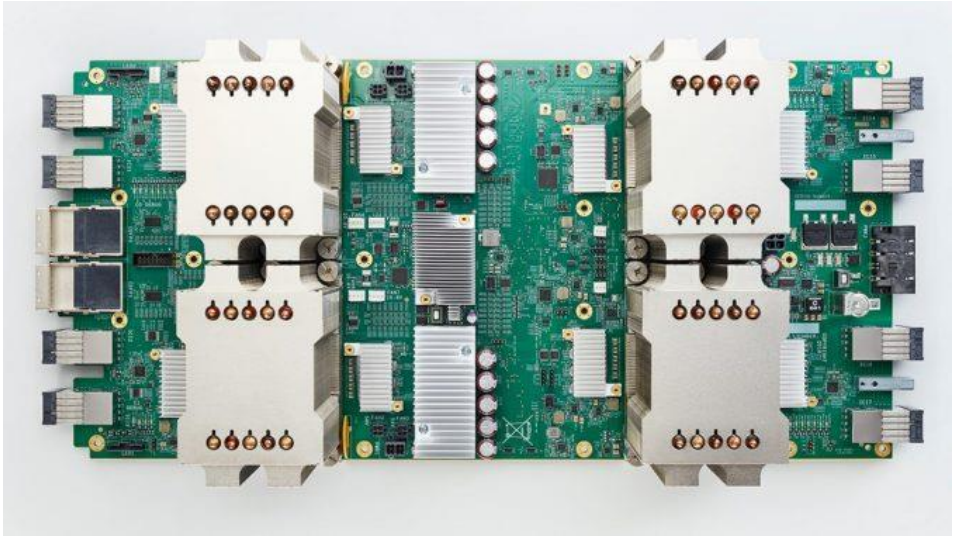


为矩阵运算设计专用芯片（ASICs）

- 观察：当前深度学习模型的计算特点：
  - 深度学习的计算本身是一种近似求解方法 → 可容忍较低计算精度
  - 当前深度学习模型的计算大量基于矩阵乘法运算 →可使用较简单的计算指令
  - 矩阵乘法在现有体系结构上都是访存瓶颈 → 增加片上内存，减少访存
  - 负载中有较大量的浮点运算→ 增加算术运算密度
- 设计思路
  - 使用低精度计算，如16bit 浮点 或8bit 定点
  - 使用简化的指令集描述关键计算指令，如CISC
  - 使用流水线计算模型来减少数据读取，如脉动阵列
  - 使用比向量比指令计算密度更高的并行方法

Tensor Processing Unit (TPU)

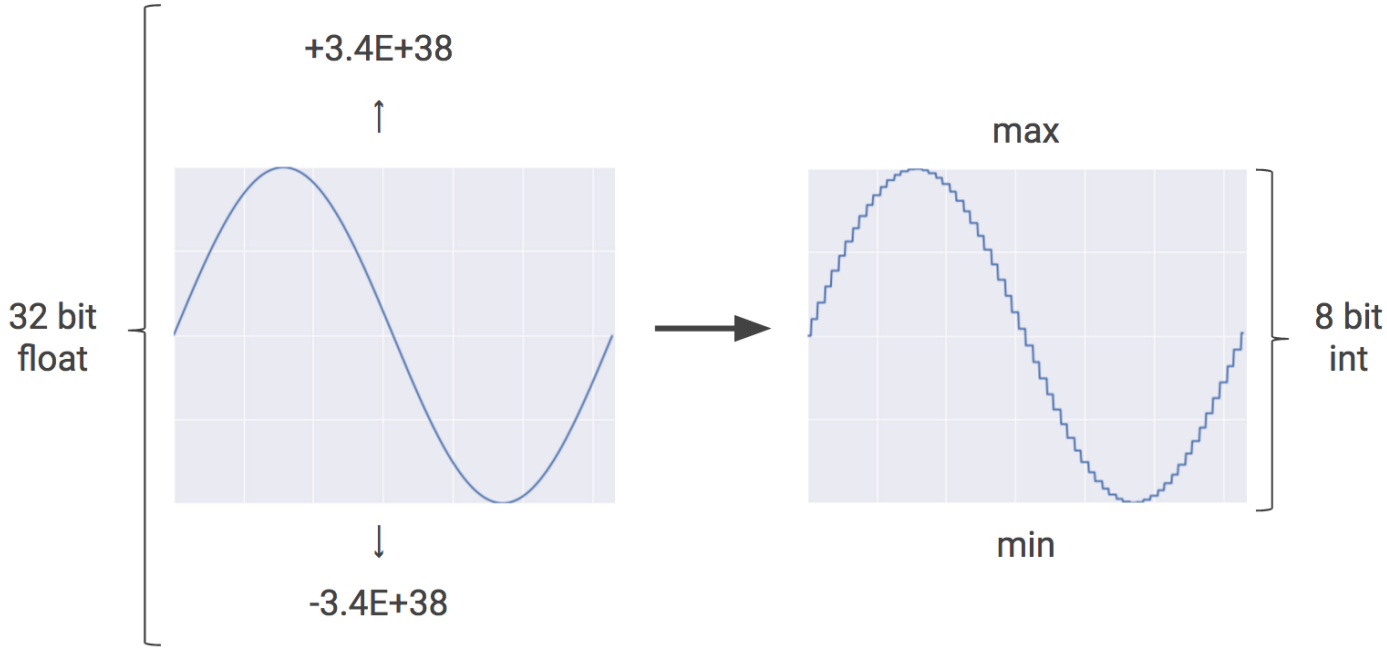
- 第一代TPU
  - 28nm process
  - runs at 700MHz
  - consumes 40W



An in-depth look at Google’s first Tensor Processing Unit (TPU)

低精度量化

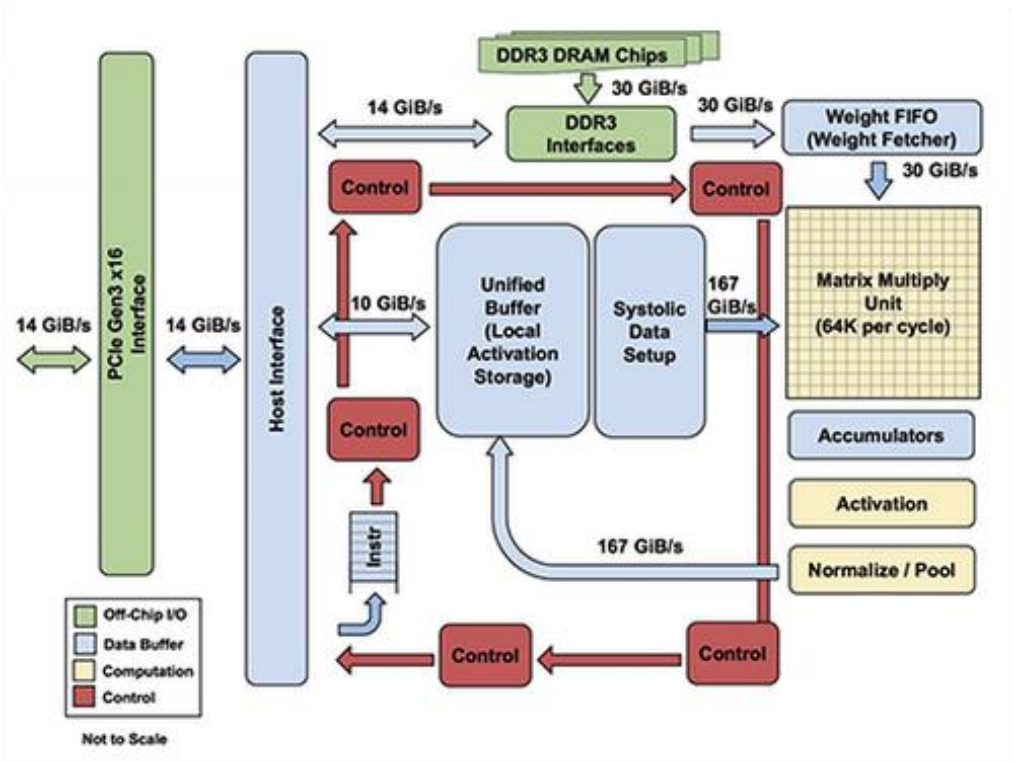
- 为降低功耗和芯片面积，一代TPU采用8bit整型作为计算数据类型
  - TPU提供65,536 8-bit 整数乘法单元
  - GPU一般提供几千个32-bit 浮点乘法单元
  - 只可用于模型推理



基于CISC的指令集

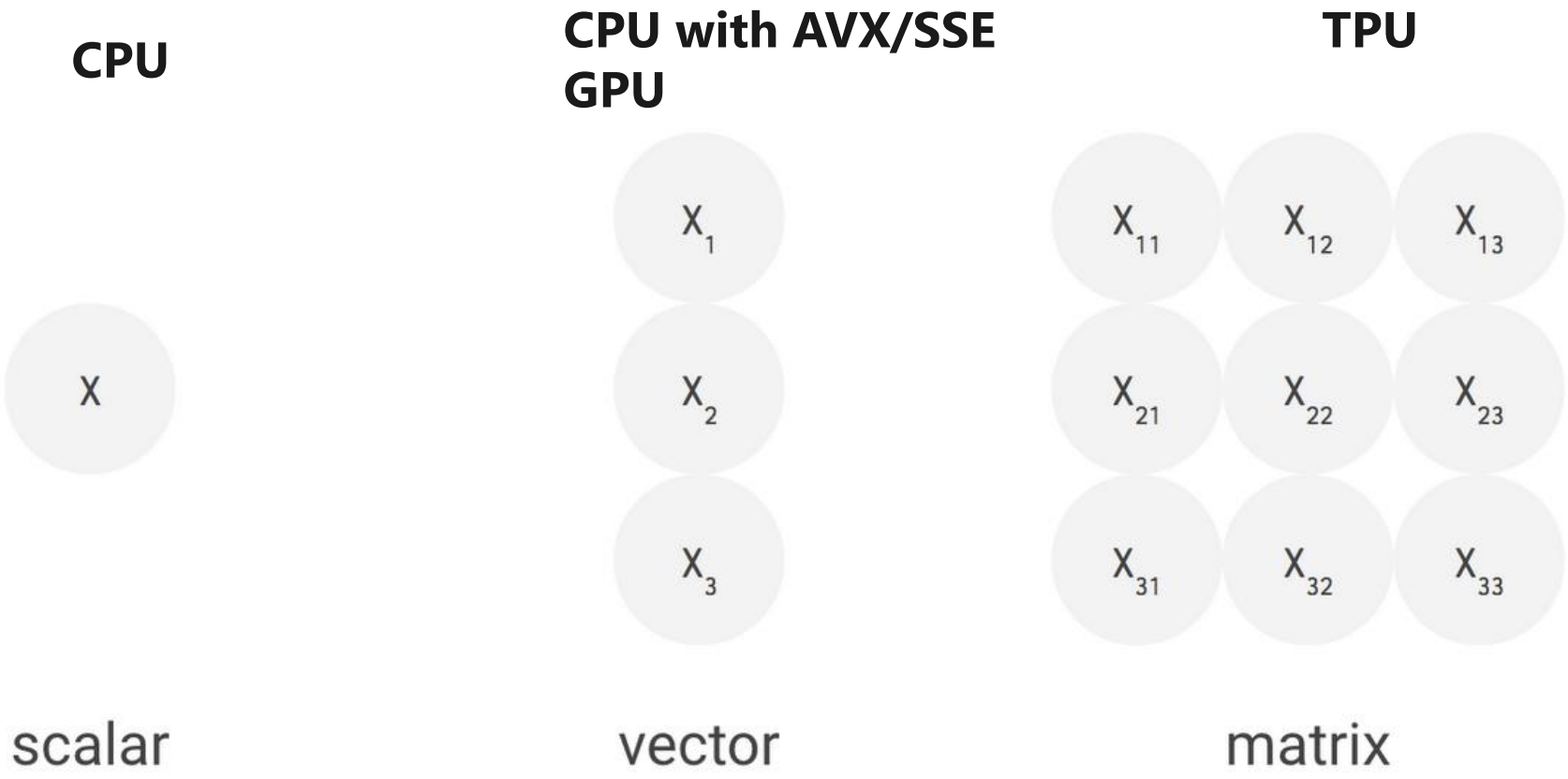
- Matrix Multiplier Unit (MXU): 65,536 8-bit 乘加单元
- Unified Buffer (UB): 24MB of SRAM
- Activation Unit (AU): 硬件激活单元

TPU Instruction	Function
Read_Host_Memory	Read data from memory
Read_Weights	Read weights from memory
MatrixMultiply/Convolve	Multiply or convolve with the data and weights,accumulate the results
Activate	Apply activation functions
Write_Host_Memory	Write result to memory



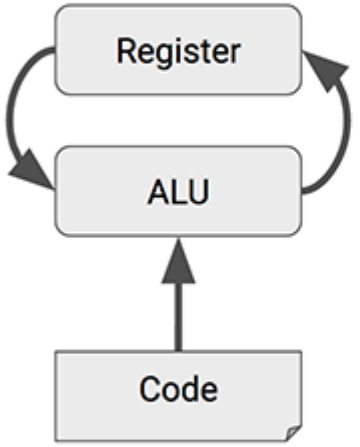
高度并行的矩阵处理单元（MXU)

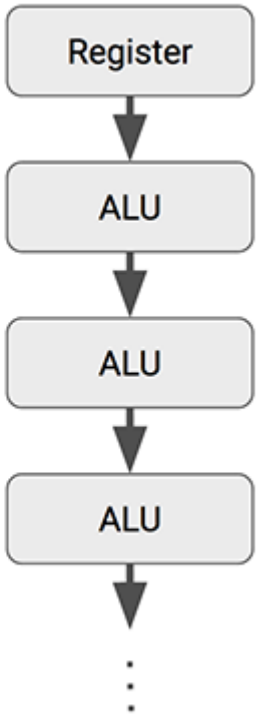
· 可以在一个时间周期计算数十万个计算



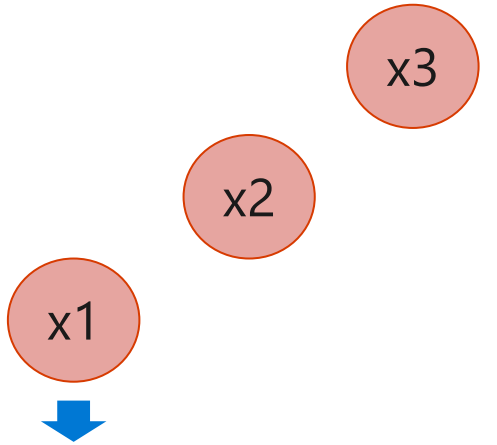
节省访存的核心：脉动阵列

- CPUs和GPUs通常需要花费大量的功耗去读取寄存器的值
- 脉动阵列的设计思想是将多个ALU运算单元串起来，从而避免每次计算都读取寄存器
- 缺点：要求计算符合特点的规则

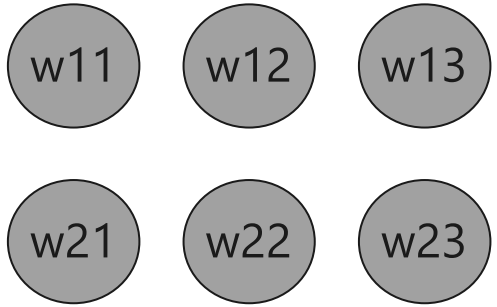




例子：矩阵乘向量



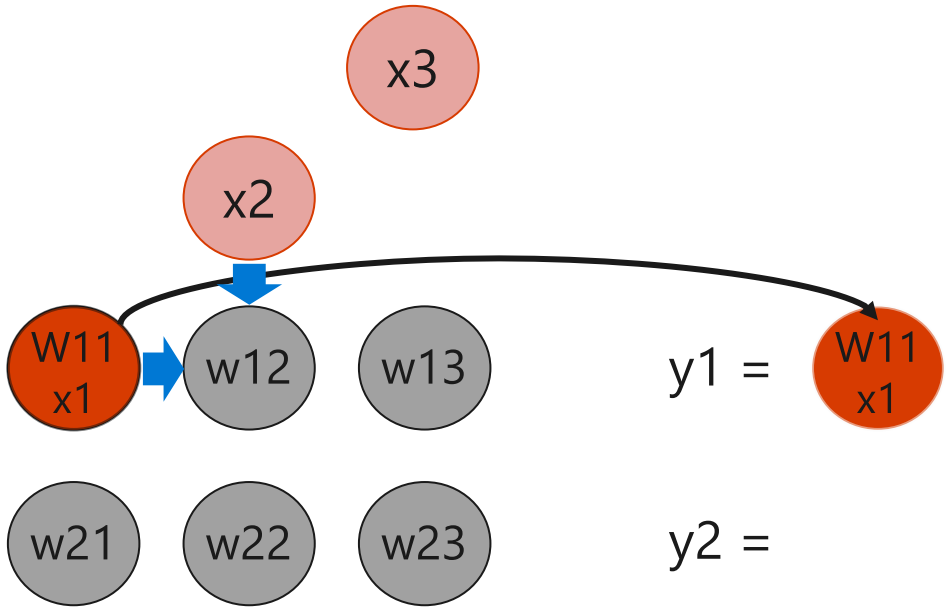
$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



y1 =

y2 =

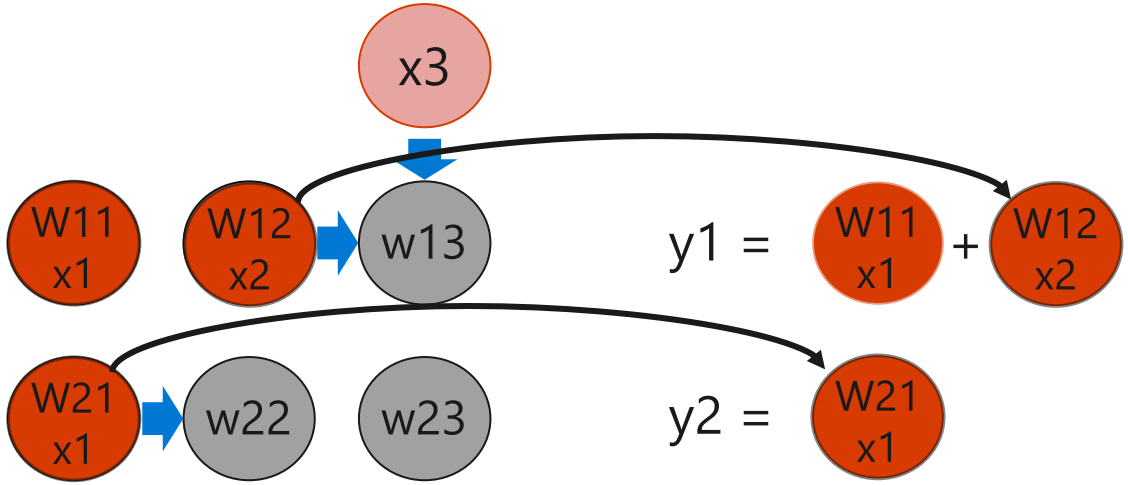
例子：矩阵乘向量



$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

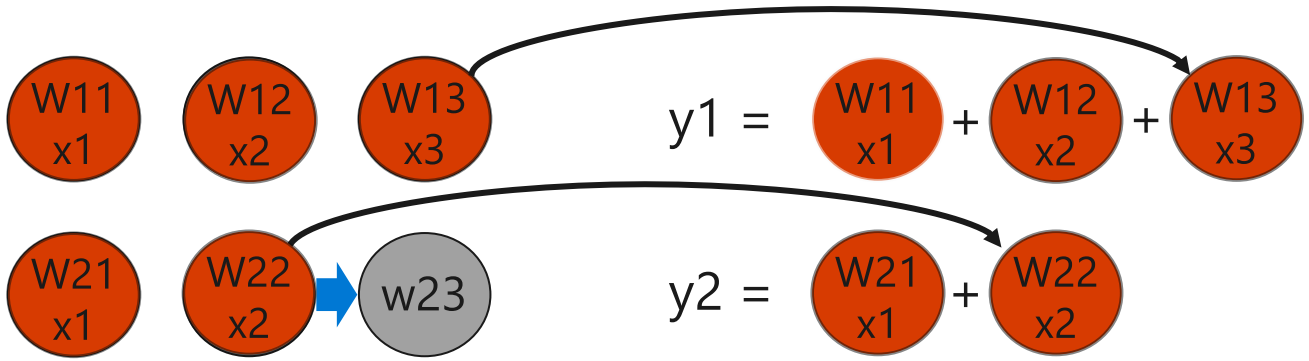
例子：矩阵乘向量

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



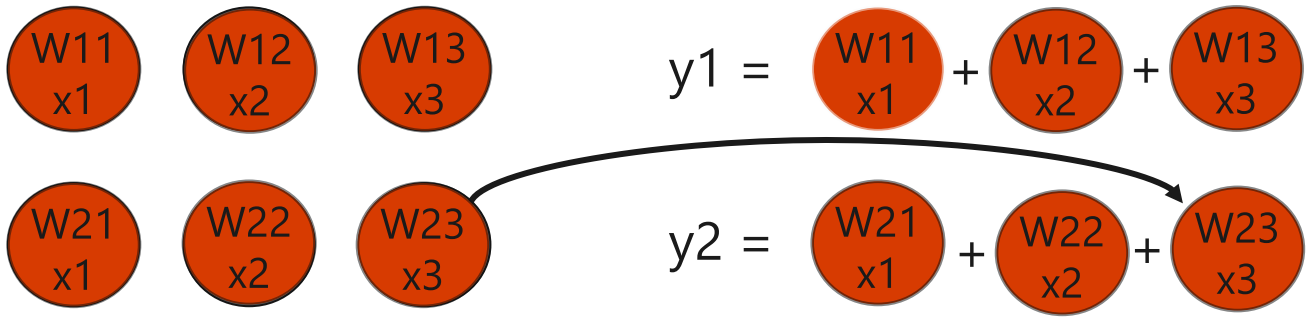
例子：矩阵乘向量

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



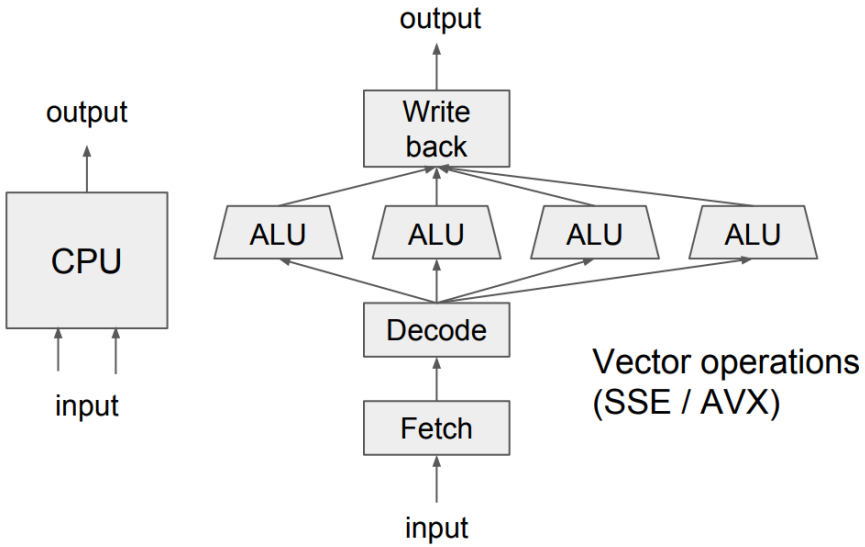
例子：矩阵乘向量

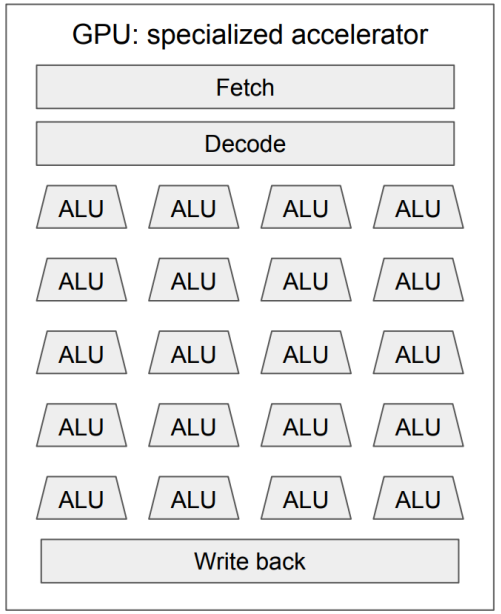
$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

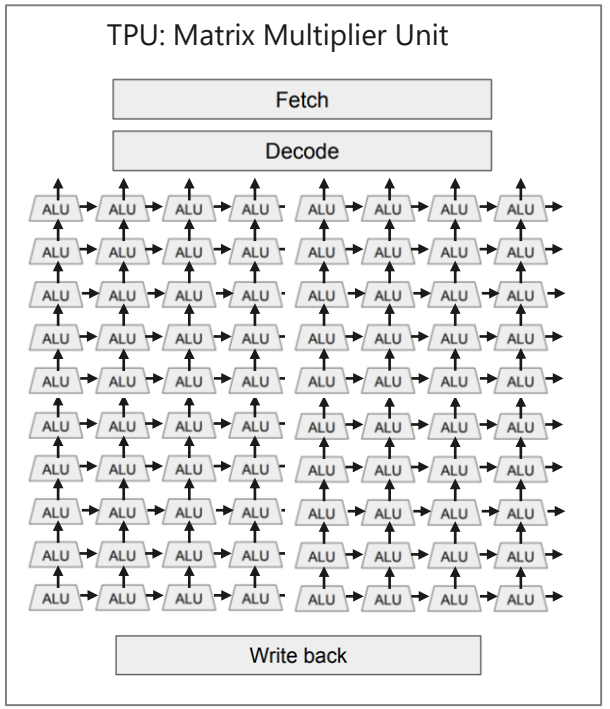


小结

- 为矩阵运算设计专用芯片（ASIC）
  - 低精度量化
  - 基于CISC的简化指令集
  - 高度并行的矩阵处理单元（MXU）
  - 节省访存的核心：脉动阵列







课后作业

· 推荐补充阅读材料

Lab 1 (for week 1, 2)

- Purpose
  - A simple throughout end-to-end AI example, from a system perspective
  - Understand the systems from debugger info and system logs
- Get ready
  - https://github.com/microsoft/ai-edu/ai-system/labs/1