向量化计算

陈宏哲@浙江大学超算队

2026-07-10

什么是向量化计算？

我将用最直白、最不绕弯子的方式告诉你：向量化计算就是把一批数据当作一个整体同时运算，而不是一个一个依次处理。

向量化计算是一个实践大于理论的方法

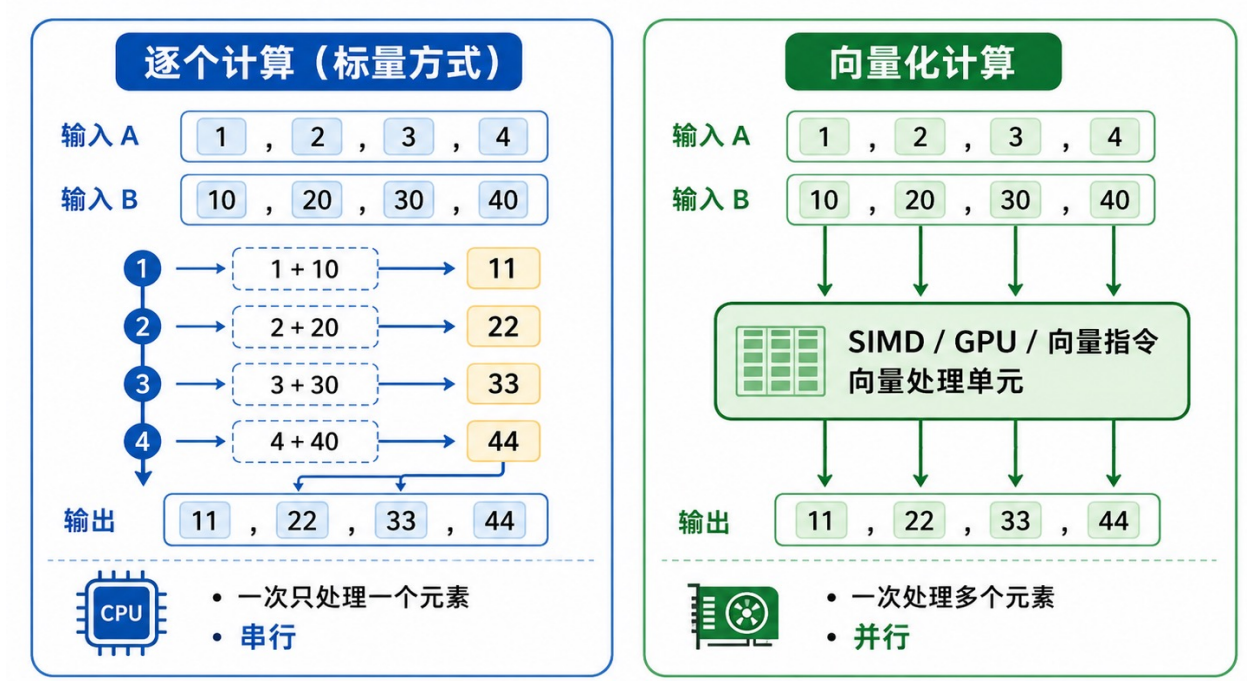
概念引入

- 标量（scalar）指的是单个数据值，比如一个整数、一个浮点数：
- a = 3
  b = 10
  a + b = 13
- 一次计算只处理一个元素，这就叫标量计算。
- 向量（vector）指的是一组同类型的数据，比如一串数字：
- A = [1, 2, 3, 4]
  B = [10, 20, 30, 40]
  A + B = [11, 22, 33, 44]
- 这里不是先算 1+10，再算 2+20，而是把整组数据交给向量处理单元，让它们同时处理多个元素。

Code Optimization - Vectorization

What is vectorization?

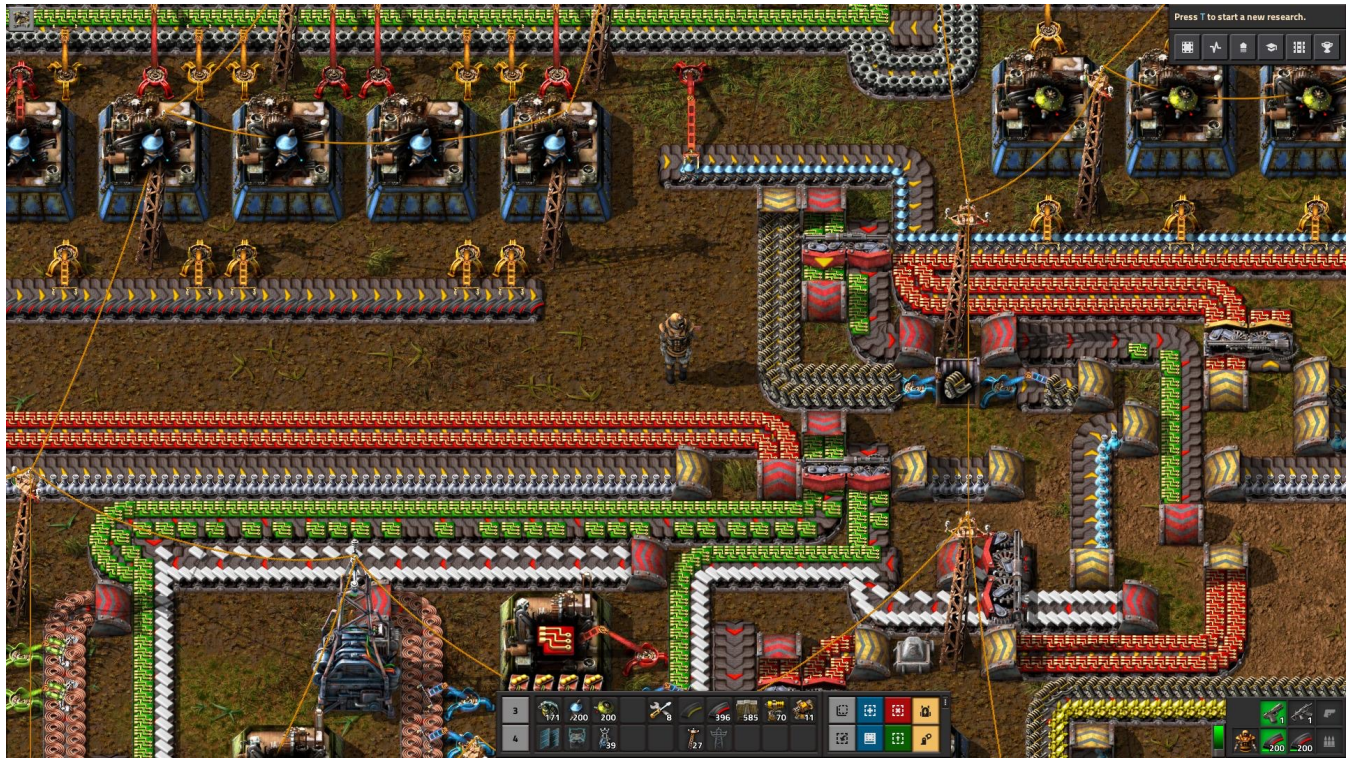
- Scaler computation: $a = 2 \cdot a$
- Vector computation: $\vec{a} = 2 \cdot \vec{a}$
  - [a, b, c, d] ⇒ [2a, 2b, 2c, 2d]



向量化计算： 把一批数据作为一个整体同时运算，而不是一个一个依次处理。

因此可以更高效地利用 CPU / GPU 的并行能力。

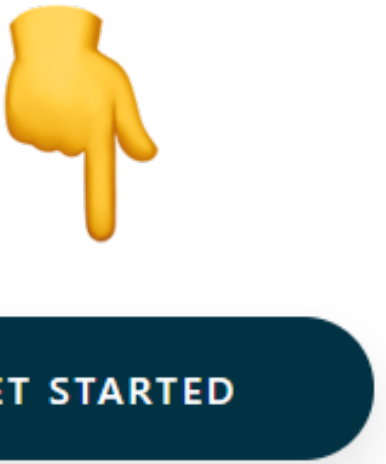
找找下面体现向量化的地方



Part-1 NumPy

NumPy 是什么？

- 使用 Python 进行**科学计算**的基础包
- 低情商表述：一个来做**矩阵运算**的库
- 高情商表述：通往**人工智能**的第一步



矩阵加法

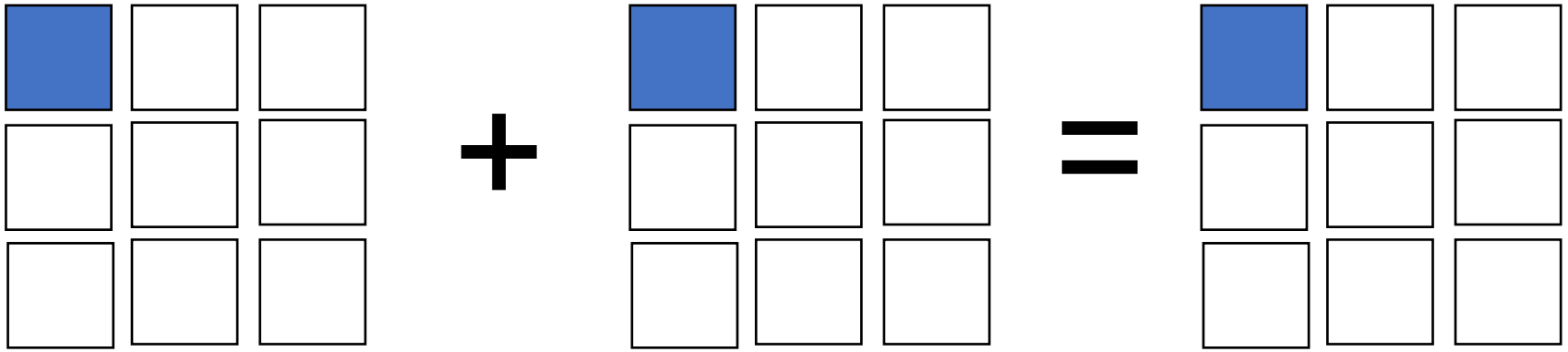
- 像这种代码就是未经向量化的

```
A = [[1,2,3],[4,5,6],[7,8,9]]
B = [[3,2,1],[6,5,4],[9,8,7]]

matrix_sum = [[0,0,0],[0,0,0],[0,0,0]]
for i in range(3):
        for j in range(3):
                matrix_sum[i][j] = A[i][j] + B[i][j]

matrix_sum
```

```
[4]  ✓ 0.5s
...  [[4, 4, 4], [10, 10, 10], [16, 16, 16]]
```

- 第一个显著特点是**大量使用 (嵌套)** for 循环
- 本质上是，如果未经优化，一次就只执行一条，造成了极大的浪费

非常简单的优化思路



- 对于 3 * 3的矩阵相加来说，答案的每个位置只和原来的两个矩阵相应坐标的值有关，和其他位置都无关
- **也就是说，一个位置的运算，和其他位置都没有关系（无依赖关系）**
- 所以如果我有九个加法单元，那么一个单元处理一个位置就可以高效并行完成。

如果是矩阵乘法呢？

```
A = [[1,2,3],[4,5,6],[7,8,9]]
B = [[3,2,1],[6,5,4],[9,8,7]]
```

[20] ✓ 0.4s

```
def matrix_mul(A,B):
        matrix_product = [[0,0,0],[0,0,0],[0,0,0]]
        for i in range(3):
                for j in range(3):
                        matrix_product[i][j] = A[i][0] * B[0][j] + A[i][1] * B[1][j] + A[i][2] * B[2][j]
        return matrix_product
```

[21] ✓ ✓ 0.4s

```
matrix_mul(A,B)
```

[22] ✓ 0.1s

```
[[42, 36, 30], [96, 81, 66], [150, 126, 102]]
```

同样可以向量化来做

- ·从输出入手，A矩阵（N*K）与B矩阵（K*M）的矩阵乘法会得到C矩阵（N*M）的结果。
- C矩阵中每一个值（C_{ij}）是由A矩阵和B矩阵中两个一维的向量相乘得到的，在计算的时候也是不相干的。
- 读的时候可能一个值会同时被**多次读，但是没有写入（没有依赖关系）**，所以是安全的。

矩阵运算大抵皆如此

- **向量化核心思想**：一次同时参与运算的不是一个值，
  而是同时多个值一起算，即一个向量
- 多个值一起算，可以是逻辑上的，也可以是实际执行上的
- 很多时候，向量化是一种思维上的抽象

NumPy

NumPy基础

GET STARTED

NumPy 基础

- 直接问 ai 就行，gpt 写 numpy 比你喝水简单
- 要学习的话可以让G老师先把代码写好，然后一行一行翻译给你这些代码和函数的意义。
- 使用Numpy的关键是找到任务里可以向量化的点。

Code is cheap. Show me your mind!

NumPy

⚠ 注意

- NumPy 本身不是“向量化”这个概念。
- 它只是提供了数组、矩阵和批量运算接口。使用 numpy 我们要做的就是把数据变成能使用向量化接口的样子。

NumPy

实战

来看看这段代码

```python
def function_2(seq: np.ndarray, sub: np.ndarray) -> np.ndarray:
    target = np.dot(sub, sub)
    candidates = np.where(
        np.correlate(seq, sub, mode='valid') == target
    )[0]

    check = candidates[:, np.newaxis] + np.arange(len(sub))
    mask = np.all(np.take(seq, check) == sub, axis=-1)

    return candidates[mask]
```

请使用 AI 工具了解这段代码的含义，并写出和它等价的非 numpy python 代码。

揭晓答案

NumPy

```python
def function_2(seq: np.ndarray, sub: np.ndarray) -> np.ndarray:
    target = np.dot(sub, sub)
    candidates = np.where(
        np.correlate(seq, sub, mode='valid') == target
    )[0]

    check = candidates[:, np.newaxis] + np.arange(len(sub))
    mask = np.all(np.take(seq, check) == sub, axis=-1)

    return candidates[mask]
```

```python
def function_1(seq: list[int], sub: list[int]) -> list[int]:
    return [
        i
        for i in range(len(seq) - len(sub) + 1)
        if seq[i:i + len(sub)] == sub
    ]
```

就这？

NumPy

- 来看个小题目：给定一个矩阵，
- 每个位置的新值是原始值加上它右上角和右下角的值
- 处于最右边的值，因为没有右上角右下角，直接当作 0

绿 = 红 + 蓝

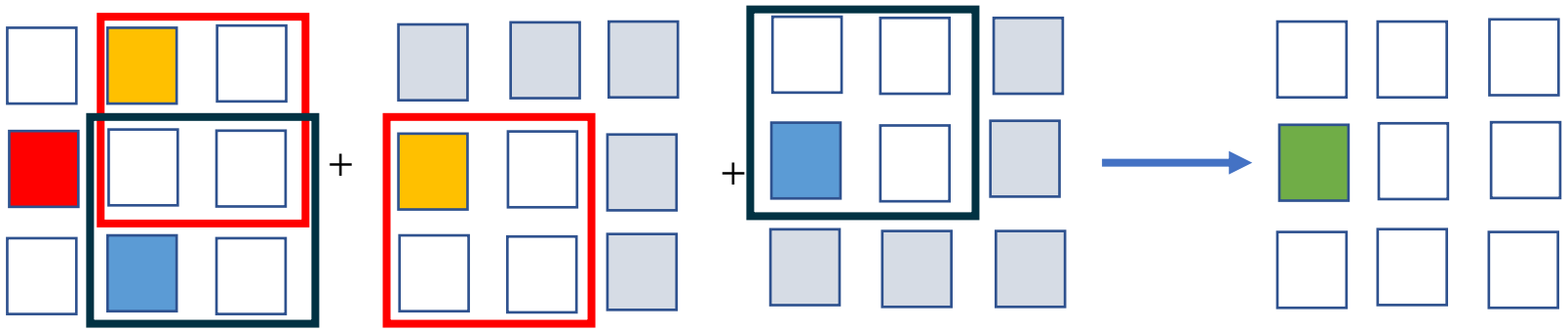
思路

算法优化 – 空间换时间

- 也就是俗称的“打表”

思路

- 空间换时间
- 对齐矩阵的格子
- 使用索引创建新的矩阵，在外面使用 pad 垫上0
- 三个矩阵相加



Code

```
a = np.arange(9).reshape(3,3)
```
[2] ✓ 0.4s

```
a
```
[3] ✓ 0.6s

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
b =np.pad(a[:-1,1:],((1,0),(0,1)))
b
```
[10] ✓ 0.6s

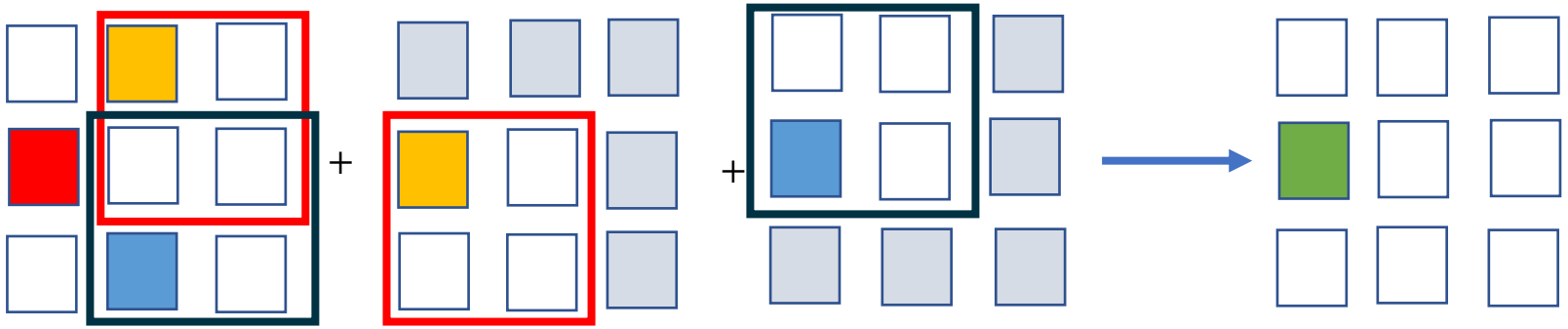
```
array([[0, 0, 0],
       [1, 2, 0],
       [4, 5, 0]])
```

```
c = np.pad(a[1:,1:],((0,1),(0,1)))
c
```
[12] ✓ 0.6s

```
array([[4, 5, 0],
       [7, 8, 0],
       [0, 0, 0]])
```

```
a + b + c
```
[13] ✓ 0.5s

```
array([[ 4,  6,  2],
       [11, 14,  5],
       [10, 12,  8]])
```



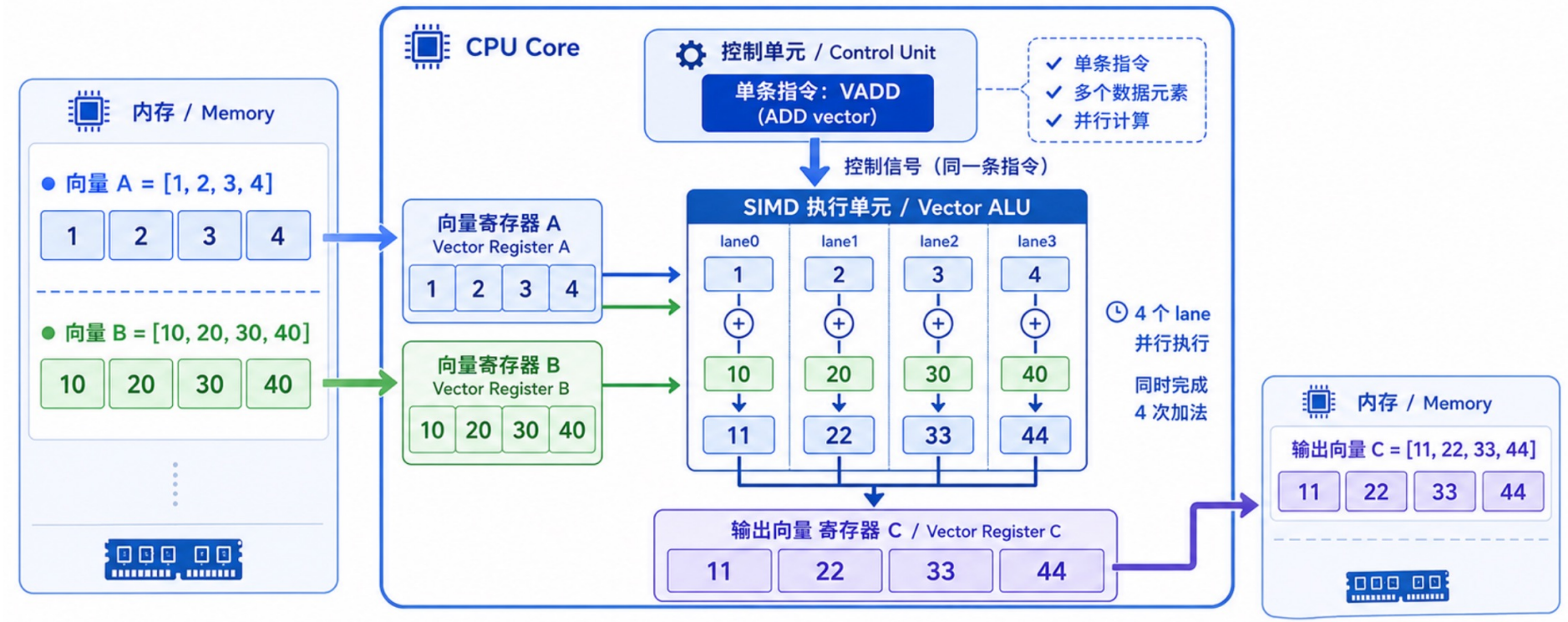
绿 = 红 + 蓝

Part-2 SIMD向量化

🧐 SIMD 是什么？

- Single Instruction Multiple Data，单指令多数据流
- 在x86架构下，SIMD一般和AVX等指令集联系在一起
- AVX指令集中提供了大量可以单指令操作多个数据单元的指令

SIMD 是什么？


内存 / Memory
向量 A = [1, 2, 3, 4]
1 2 3 4
向量 B = [10, 20, 30, 40]
10 20 30 40
CPU Core
控制单元 / Control Unit
单条指令：VADD
(ADD vector)
单条指令
多个数据元素
并行计算
控制信号（同一条指令）
向量寄存器 A
Vector Register A
1 2 3 4
向量寄存器 B
Vector Register B
10 20 30 40
SIMD 执行单元 / Vector ALU
lane0 lane1 lane2 lane3
1 2 3 4
10 20 30 40
11 22 33 44
4 个 lane
并行执行
同时完成
4 次加法
输出向量 寄存器 C / Vector Register C
11 22 33 44
内存 / Memory
输出向量 C = [11, 22, 33, 44]
11 22 33 44

数据个数=加速倍数?

- 很自然的会认为，SIMD指令同时操作n个数据，那加速比就该是n
- 真的是这样吗？🤔

数据个数=加速倍数?

- 很自然的会认为，SIMD指令同时操作2个数据，那加速比就该是2
- 事实上很复杂：内存带宽使用，解码消耗减小等等，很难说清楚，具体问题具体分析，但可以使用倍数估算。
- 当作为整体代码一部分时，情况就更加复杂了
- 甚至可能提供的AVX2指令实际上是由2条AVX指令模拟出来的，这就是我们之前提到的逻辑上的向量化。

越长越好?

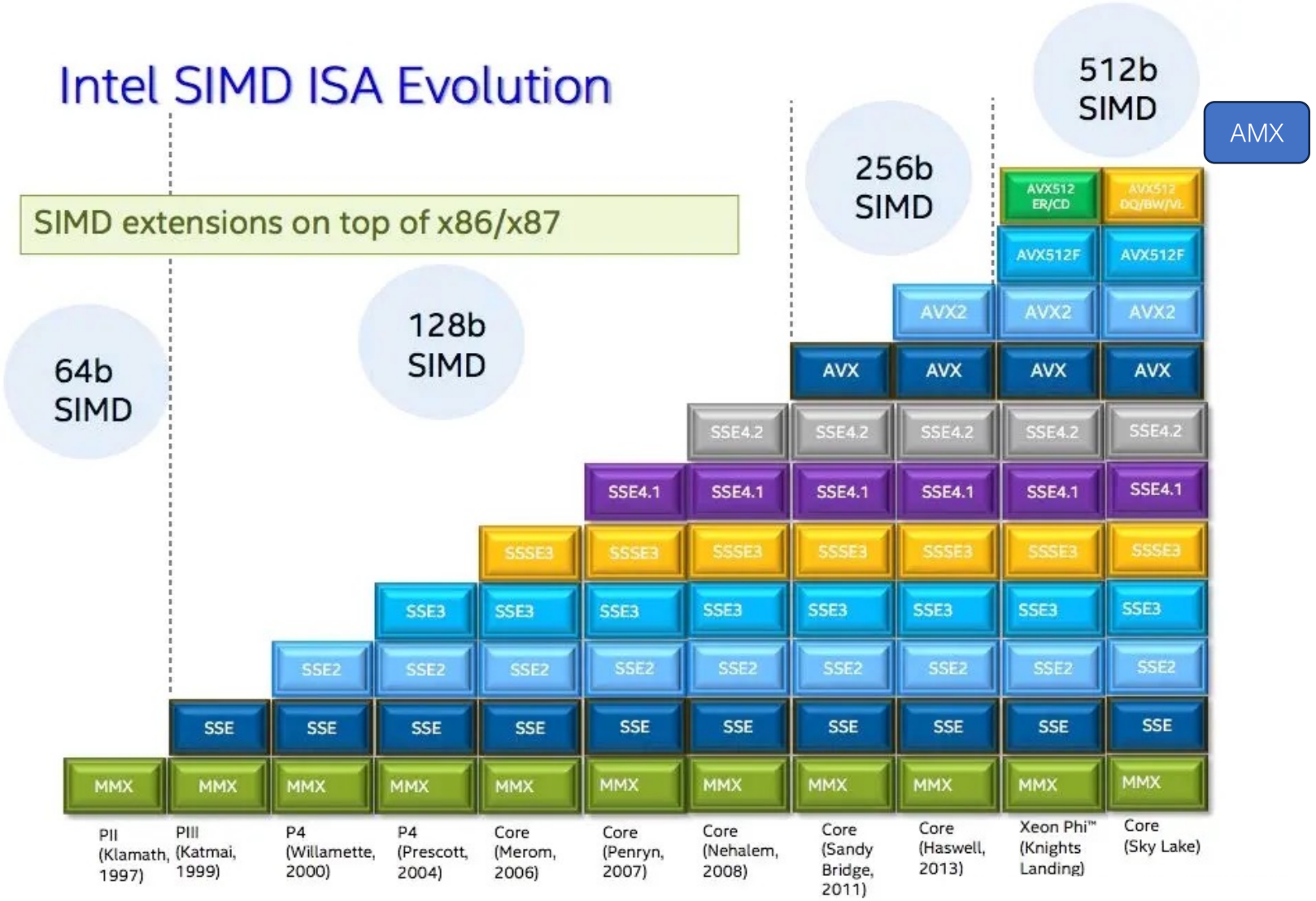
- 一条AVX2指令能处理256位的数据，一条AVX512指令一次能处理512位数据，那为什么不再长亿点？
- 一条指令干翻它10GB的数据量不香吗 🤩
- 显然这不现实 🤡

越长越好?

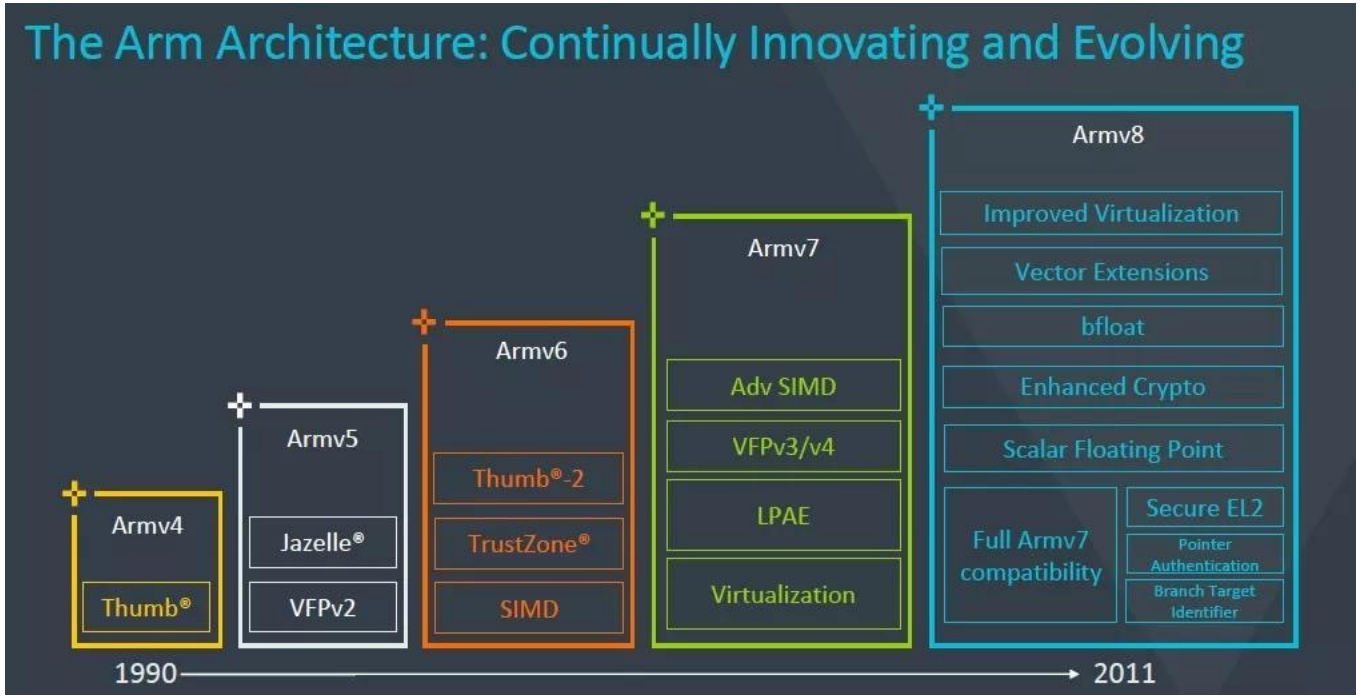
- 指令的功能是需要硬件实现的，一条指令处理8个double就意味至少需要8个double的运算单元，这意味着更大的面积，更高的成本，更多的发热
- AVX512之前被戏称为大火炉，对CPU负载高频率跑高容易不稳定
- 发热严重，过热导致降频，导致了AVX512表现不如AVX2的奇景

Intel

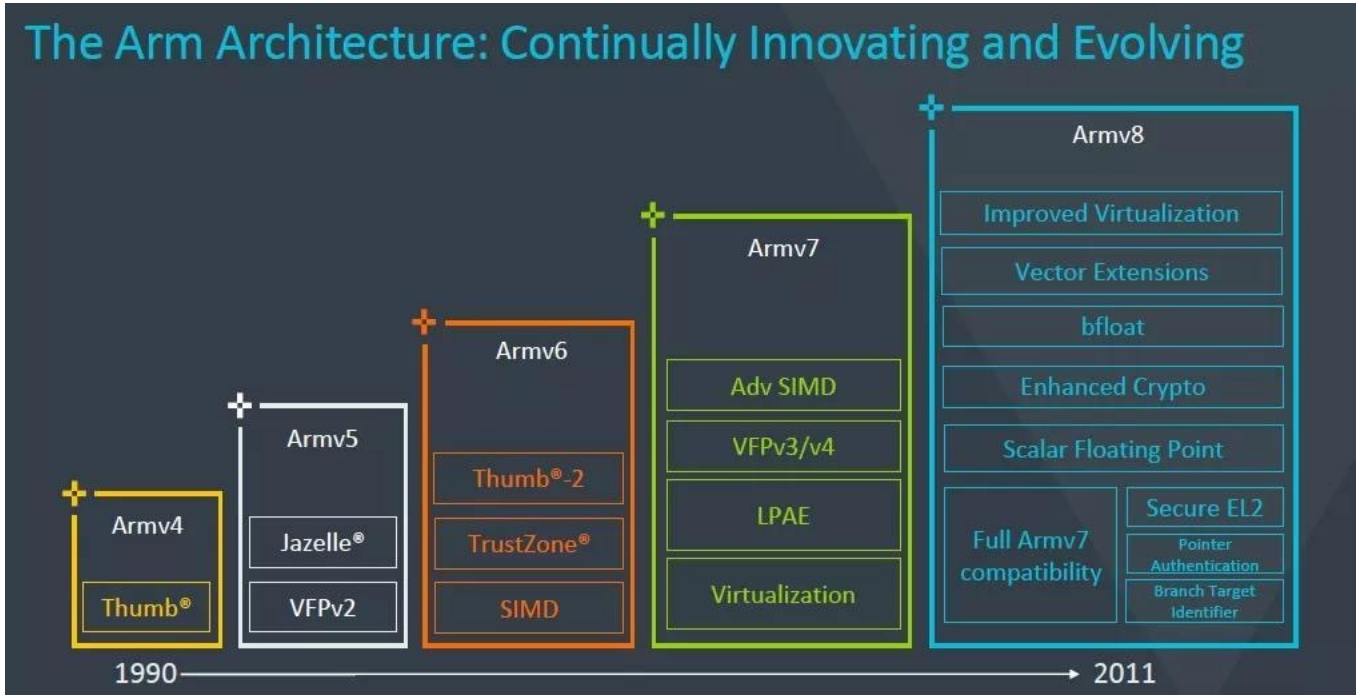
- MMX
- SSE
- SSE2
- AVX
- AVX2
- AVX512



- SVE
- SME


The Arm Architecture: Continually Innovating and Evolving
Armv4
Thumb®
Armv5
Jazelle®
VFPv2
Armv6
Thumb®-2
TrustZone®
SIMD
Armv7
Adv SIMD
VFPv3/v4
LPAE
Virtualization
Armv8
Improved Virtualization
Vector Extensions
bfloat
Enhanced Crypto
Scalar Floating Point
Full Armv7 compatibility
Secure EL2
Pointer Authentication
Branch Target Identifier
1990
2011

- SVE
- SME


The Arm Architecture: Continually Innovating and Evolving
Armv4
Thumb®
Armv5
Jazelle®
VFPv2
Armv6
Thumb®-2
TrustZone®
SIMD
Armv7
Adv SIMD
VFPv3/v4
LPAE
Virtualization
Armv8
Improved Virtualization
Vector Extensions
bfloat
Enhanced Crypto
Scalar Floating Point
Full Armv7 compatibility
Secure EL2
Pointer Authentication
Branch Target Identifier
1990
2011

为什么要手写？😱

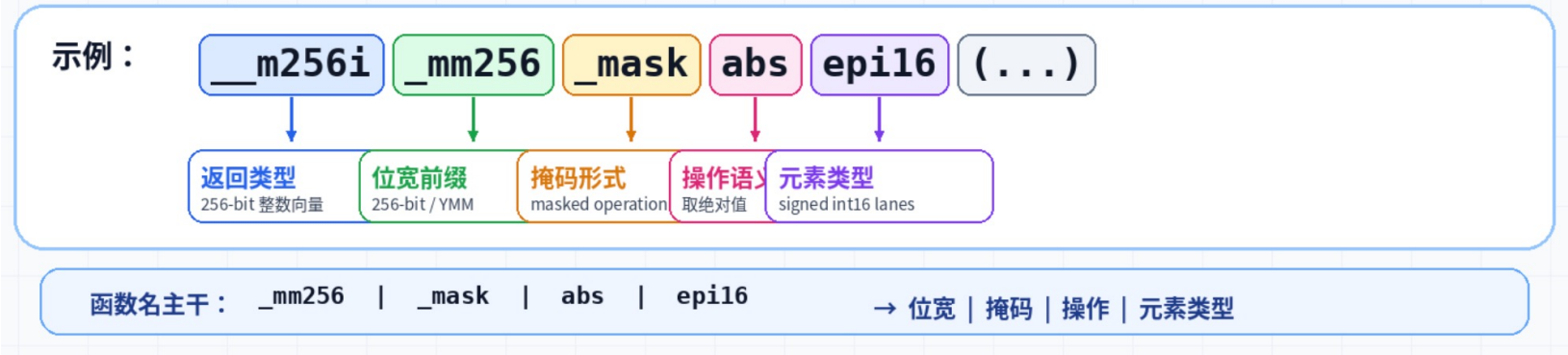
- 简单的情况（如矩阵乘法）下其实没有必要，你只要-O3 –mavx2，编译器就能做的很好
- 但如果你的代码结构复杂，循环难以界定边界，甚至还有分支，那就只能靠你自己手写了

我该怎么写？

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

- 看文档！！！
- 你需要什么
- 再看有什么

我该怎么读?



返回类型 _mm[位宽]_[mask/maskz]_[操作]_[修饰词]_[数据类型]

我该怎么读?

```
__m256i _mm256_add_epi64 (__m256i a, __m256i b)
```

这是一个针对 256 位整数计算的指令

vpaddq

Synopsis

```
__m256i _mm256_add_epi64 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vpaddq ymm, ymm, ymm
CPUID Flags: AVX2
```

需要使用这个头文件

对应的汇编指令

Description

Add packed 64-bit integers in `a` and `b`, and store the results in `dst`.

指令描述

Operation

指令等价的串行伪代码

```
FOR j := 0 to 3
        i := j*64
        dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
dst[MAX:256] := 0
```

Latency and Throughput

不同 cpu 架构的流水线延迟和吞吐

Architecture	Latency	Throughput (CPI)
Alderlake	1	0.333333333
Icelake Intel Core	1	0.33
Icelake Xeon	1	0.33
Sapphire Rapids	1	0.333333333
Skylake	1	0.33

读文档方法总结

- 命名有规则
- 看返回类型和输入类型筛选
- 指令类型_操作_单元类型
- 查找关键字
- 直接 GPT

读文档需要关注的点

- 需要的头文件
- 实际汇编指令
- lscpu可以看处理器flags
- 伪代码等效操作
- 实际架构下的性能

向量化前后

- 分支也可以向量化
- 向量化完

  可读性喂了🐶!

  我的意思

- 此处向量化的含义是

  真正的同时操作多个数据组成的向量

```
double l = m_lvec[i];
double a = m_avec[i];
double b = m_bvec[i];

double _distlab =   (l - kseedsl[n])*(l - kseedsl[n]) +
                    (a - kseedsa[n])*(a - kseedsa[n]) +
                    (b - kseedsb[n])*(b - kseedsb[n]);

double _distxy =    (x - kseedsx[n])*(x - kseedsx[n]) +
                    (y - kseedsy[n])*(y - kseedsy[n]);

//------------------------------------------------------------------------
double dist = _distlab/maxlab[n] + _distxy*invxywt;//only varying m, prettier superpixels
//double dist = _distlab/maxlab[n] + _distxy*invxywt[n];//varying both m and S
//------------------------------------------------------------------------

distlab[i] = _distlab;
distxy[i] = _distxy;

if( dist < distvec[i] )
{
    distvec[i] = dist;
    klabels[i]  = n;
}
```

```
__m256d l_vec = _mm256_load_pd(&m_lvec[i]);
__m256d a_vec = _mm256_load_pd(&m_avec[i]);
__m256d b_vec = _mm256_load_pd(&m_bvec[i]);
__m256d x_vec =
    _mm256_set_pd((double)(x + 3), (double)(x + 2),
                  (double)(x + 1), (double)(x));
__m256d y_vec = _mm256_set1_pd((double)y);
__m256d l_vec_t1 = _mm256_sub_pd(l_vec, _kseedsl_vec);
l_vec_t1 = _mm256_mul_pd(l_vec_t1, l_vec_t1);
__m256d a_vec_t1 = _mm256_sub_pd(a_vec, _kseedsa_vec);
a_vec_t1 = _mm256_mul_pd(a_vec_t1, a_vec_t1);
__m256d b_vec_t1 = _mm256_sub_pd(b_vec, _kseedsb_vec);
b_vec_t1 = _mm256_mul_pd(b_vec_t1, b_vec_t1);
__m256d _distlab_vec =
    _mm256_add_pd(l_vec_t1, a_vec_t1);
_distlab_vec = _mm256_add_pd(_distlab_vec, b_vec_t1);
__m256d x_vec_t1 = _mm256_sub_pd(x_vec, _kseedsx_vec);
x_vec_t1 = _mm256_mul_pd(x_vec_t1, x_vec_t1);
__m256d y_vec_t1 = _mm256_sub_pd(y_vec, _kseedsy_vec);
y_vec_t1 = _mm256_mul_pd(y_vec_t1, y_vec_t1);
__m256d _distxy_vec = _mm256_add_pd(x_vec_t1, y_vec_t1);
__m256d dist_vec_t1 =
    _mm256_div_pd(_distlab_vec, maxlab_vec);
__m256d dist_vec_t2 =
    _mm256_mul_pd(_distxy_vec, invxywt_vec);
__m256d dist_vec =
    _mm256_add_pd(dist_vec_t1, dist_vec_t2);

_mm256_store_pd(&distlab[i], _distlab_vec);

__m256d distvec_vec = _mm256_load_pd(&distvec[i]);
__m256d cmp_res_vec =
    _mm256_cmp_pd(dist_vec, distvec_vec, _CMP_LT_OQ);

distvec_vec = _mm256_blendv_pd(distvec_vec, dist_vec,
                               cmp_res_vec);
_mm256_store_pd(&distvec[i], distvec_vec);

__m256i permuted_vec = _mm256_permutevar8x32_epi32(
    _mm256_castpd_si256(cmp_res_vec), K_PERM_VEC);
__m128i cmp_int_vec =
    _mm256_castsi256_si128(permuted_vec);


__m128i n_vec = _mm_set1_epi32(n);
_mm_maskstore_epi32(&klabels[i], cmp_int_vec, n_vec);

x += 4;
```

基本流程

- Load需要计算的数据到向量寄存器
- 进行需要的向量化计算
- Store将向量寄存器的数据存回内存
- 和把大象塞进冰箱一样，简单吧（X 

常见问题

- 内存对齐
- 循环边界不确定——展开非整数边界
- 循环分支的开销掩盖了SIMD提升——循环展开
- 寄存器数量超限——一般默认16个256位寄存器进行考量

小结

- 如果想要熟练掌握手写SIMD向量化的优化技术
  实践才是王道
- 大多数时候自动向量化就够了
- 一定一定要在**最后**再进行手写SIMD优化

实战1

- 使用 AVX512 对以下代码进行优化。

输入：

q: [D]

K: [T, D]

V: [T, D]

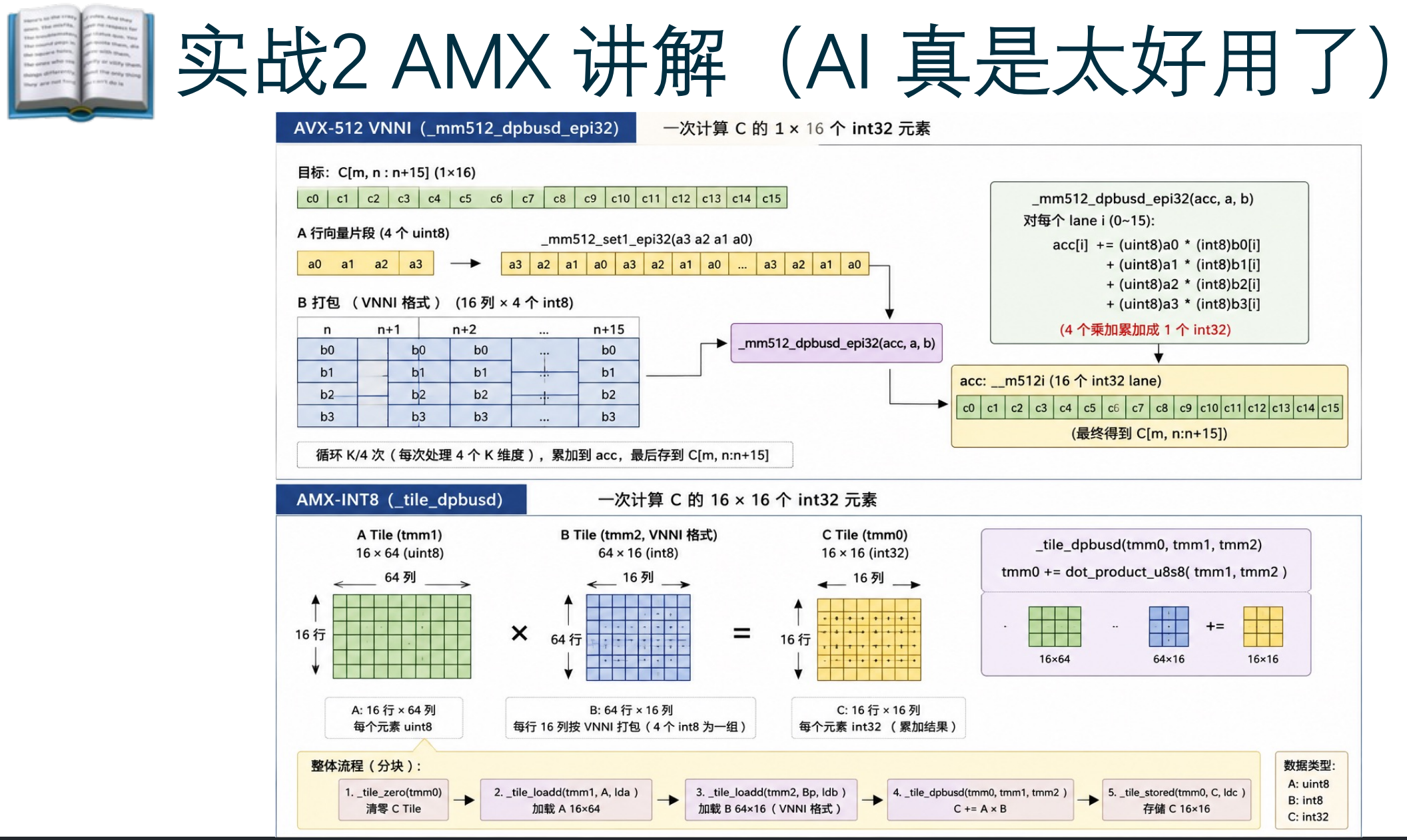
计算：

score[t] = q · K[t] / sqrt(D)

prob[t] = softmax(score)[t]

out[j] = Σ_t prob[t] * V[t][j]

实战2 AMX 讲解（AI 真是太好用了）



实战2 AMX 公式

```
for i = 0 to M step A_TILE_LINES:
     for j = 0 to N step B_TILE_COLS:
          SET C_TILE ZERO
          for k = 0 to K step B_TILE_LINES:
              LOAD A_TILES
              LOAD B_TILES
              TILE_MATMUL
          STORE C_TILES
```

1. 将A和B矩阵的形状改造成适合AMX运算的样子
2. 写好 3 重 for 循环
3. 计算好 A 和 B 矩阵的索引
4. Load、Calc and Store

实战2

- 使用 AVX512 + AMX 优化 Baseline
- 有点小难

实战注意点

- 最开始以尽可能把所有计算都向量化为目标去做
- 有多种向量化方法的时候考虑哪种访存开销最小。
- 有些时候 AMX 并不如 AVX，需要灵活使用。
- 使用 AI 工具的时候一定要回文档查询，防止造假。

SVE

https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=%5BSVE,SVE2%5D

AVX 是固定宽度向量：明确知道一次算 8 个、16 个。
SVE 是可伸缩向量：一次算几个由硬件决定，代码按“还能处理多少就处理多少”来写（SVL）。

```
for (int i = 0; i < n; i += svcntw()) {
    svbool_t pg = svwhilelt_b32(i, n);
    svfloat32_t a = svld1(pg, A + i);
    svfloat32_t b = svld1(pg, B + i);
    svfloat32_t c = svadd_f32_x(pg, a, b);
    svst1(pg, C + i, c);
}
```

_m：无效 lane 保留原值　　c = [11, 22, 3, 44]
_z：无效 lane 置零　　c = [11, 22, 0, 44]
_x：无效 lane 不关心　　c = [11, 22, ?, 44]

```
c = svadd_f32_m(pg, a, b);
pg = [1, 1, 0, 1]
a  = [1, 2, 3, 4]
b  = [10, 20, 30, 40]
```

svcntw() = 当前机器一个 SVE 向量能放多少个 32-bit 元素

命名规则：
sv + 操作名 + _数据类型 + _predicate策略

AVX里mask 是增强能力，尤其常用于尾部和条件执行。
SVE里predicate 是基本编程模型的一部分，循环控制、尾部处理、条件执行都离不开它。

被驱逐开 ARMv9 的 SME，被 S 级企业捡到～这个 SME 超规格～

- 仅作为扩展知识讲解。

1. SVMOPA 指令：矩阵乘加（Outer Product Accumulate）

`svmopa[.za] zd, pg/m, zn, zm`

- zd ：ZA 矩阵寄存器（累加结果）
- pg/m：predicate（控制有效元素）
- zn ：向量寄存器，作为矩阵 A 的一行（行向量）
- zm ：向量寄存器，作为矩阵 B 的一列（列向量）
- 作用 ：将 A(行向量) × B(列向量) 的外积，累加到 ZA 矩阵中

数学意义（外积累加）

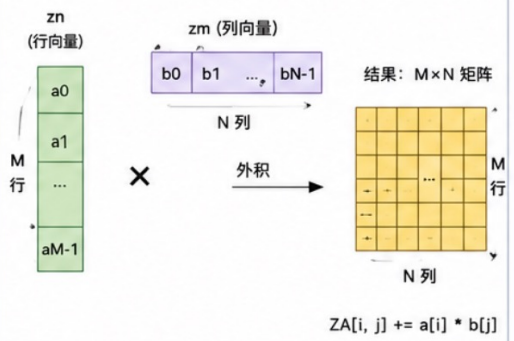
假设 zn 长度 = K，zm 长度 = K，ZA 矩阵大小为 M×N，则执行：

ZA[M×N] += zn[M×1] × zm[1×N]

即： ZA[i, j] += zn[i] * zm[j]

外积（Outer Product）：产生一个 M×N 的矩阵

2. 数据形状示意



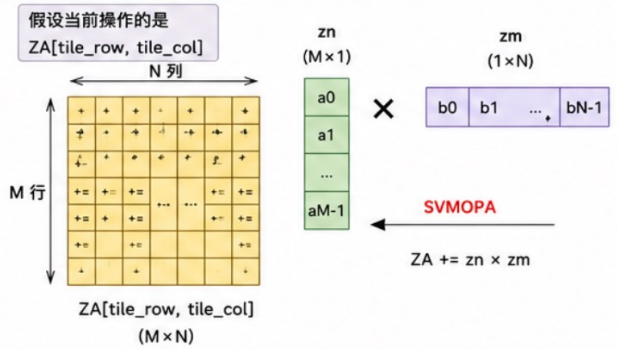
3. ZA（Z Array）Tile 寄存器

ZA 是一个二维寄存器文件，由若干 Tile 组成
每个 Tile 是 M×N 个元素
数据类型可配置（如 bf16/fp32/s32 等）

Tile(0,0)	Tile(0,1)	...	Tile(0,Tn-1)
Tile(1,0)	Tile(1,1)	...	Tile(1,Tn-1)
...	...	...	...
Tile(Tm-1,0)	Tile(Tm-1,1)	...	Tile(Tm-1,Tn-1)

Tm: tile 行数（高度） Tn: tile 列数（宽度）
每个 Tile 内部是 M×N 的矩阵元素

4. 一个 SVMOPA 指令如何更新 ZA 中的 Tile



5. 典型使用流程（GEMM 为例）

1. 配置 ZA 大小
   `smstart` -- 使能 SME
   `set_za_config(M_tile, N_tile, Tm, Tn, dtype)`
2. 初始化 ZA（清零或加载）
   `zero_za()` 或 `load_za(...)`
3. K 维度循环
   `for k in 0..K-1:`
   加载 A 的一行向量 → zn
   加载 B 的一列向量 → zm
   执行 `svmopa zd, pg/m, zn, zm` （ZA 累加）
4. 存储结果
   `store_za(...)` → C 矩阵

6. 相关寄存器概念对比

概念	SVE	SME/SME2
向量寄存器	Z0 ~ Z31	复用（Z 寄存器仍存在）
谓词寄存器	P0 ~ P15	相同（用于 mask/尾处理）
矩阵寄存器	无	ZA（Z Array） 由多个 Tile 组成

常见 SME2 矩阵乘加指令（示例）

svmopa ：外积累加（向量×向量 → ZA）
svmops ：对称外积累加
svmmla ：矩阵乘加（向量 × ZA → ZA）
svmmls ：矩阵乘减

注：具体指令后缀和数据类型（如 .b，.h，.s，.d，.bf16，.s32 等）会影响操作的元素类型和精度，本图以概念为主。

谢谢大家！