Hardware Acceleration

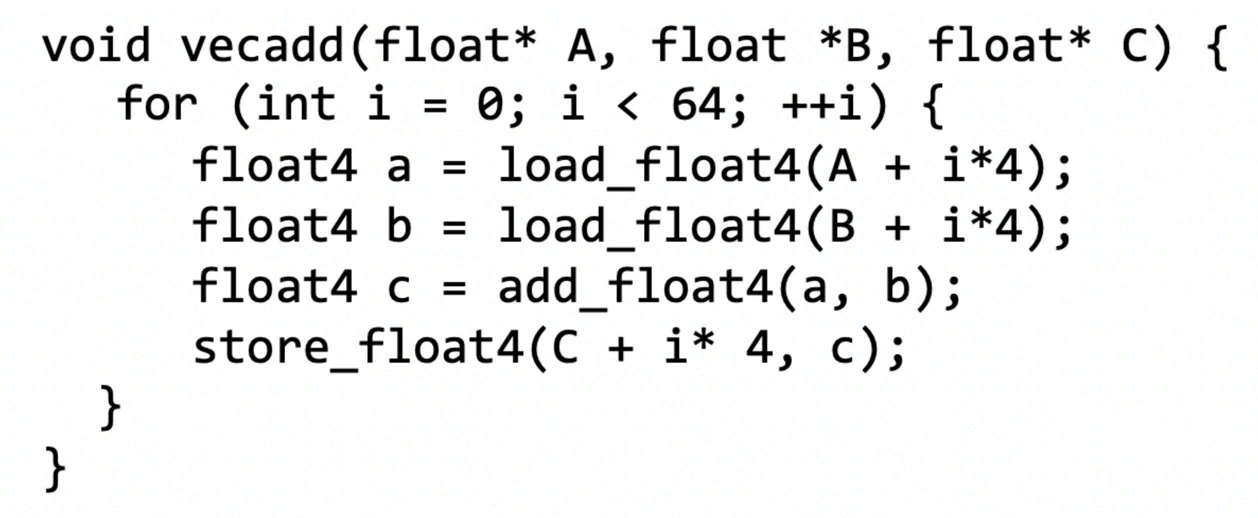
General Acceleration Techniques

本节主要介绍一些通用的加速技巧，主要针对 CPU

机器学习框架主要分为 high-level 的计算图和 low-level 的线性代数库两部分。

Vectorization

核心思想：对于不同数据执行相同操作，可以构建向量，存在相应的特殊寄存器中，达到并行计算的目的。



```
void vecadd(float* A, float *B, float* C) {
   for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C + i* 4, c);
  }
}
```

要求：内存地址对齐（16 字节 or 32 字节）

Data Layout and Strides

传统数据布局分为 Row major 和 Column major 两种

- Row major: `A[i, j] => Adata[i * A.shape[1] + j]`
- Column major: `A[i, j] => Adata[j * A.shape[0] + i]`

更一般地，有 Strides format

- Strides format: `A[i, j] => Adata[i * A.strides[0] + j * A.strides[1]]`

优点：可以在不需要拷贝的前提下实现切片和转置操作

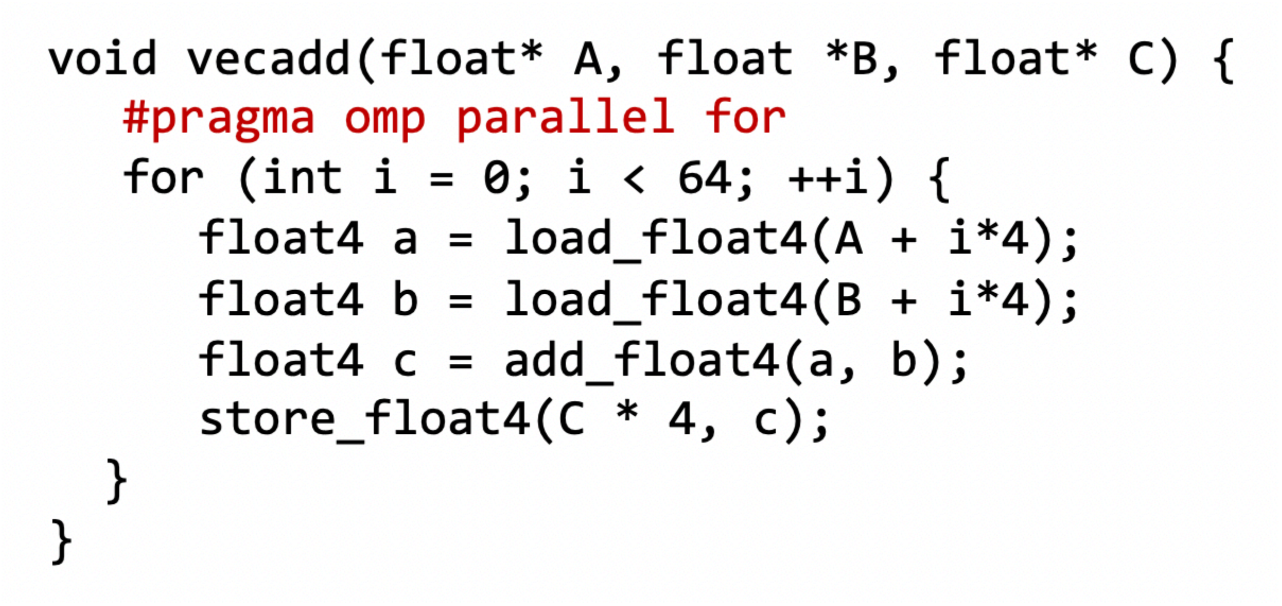
- Slice: change the begin offset and shape
- Transpose: swap the strides
- Broadcast: insert a stride equals 0

缺点：内存访问不连续，效率低

- 使向量化更困难
- 很多线性代数的操作需要首先对数组进行压缩

Parallelization

核心思想：对于多核处理器，可以对循环操作进行并行处理，如 OpenMP 库

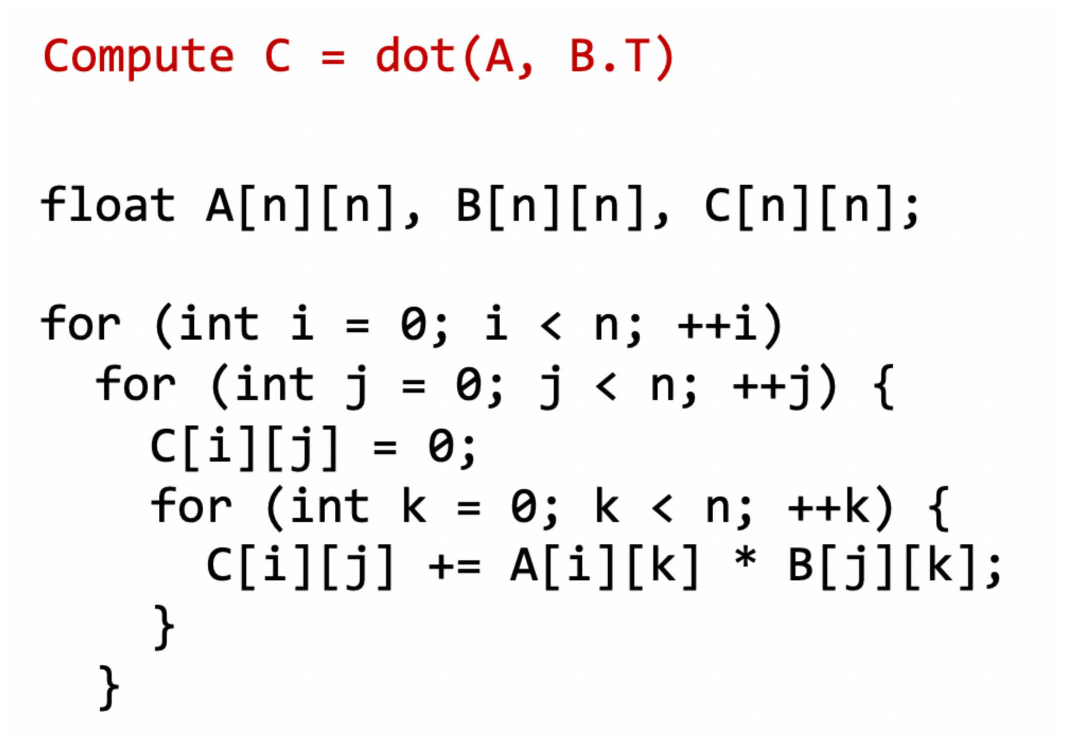


```
void vecadd(float* A, float *B, float* C) {
   #pragma omp parallel for
   for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C * 4, c);
  }
}
```

Case study: Matrix Multiplication

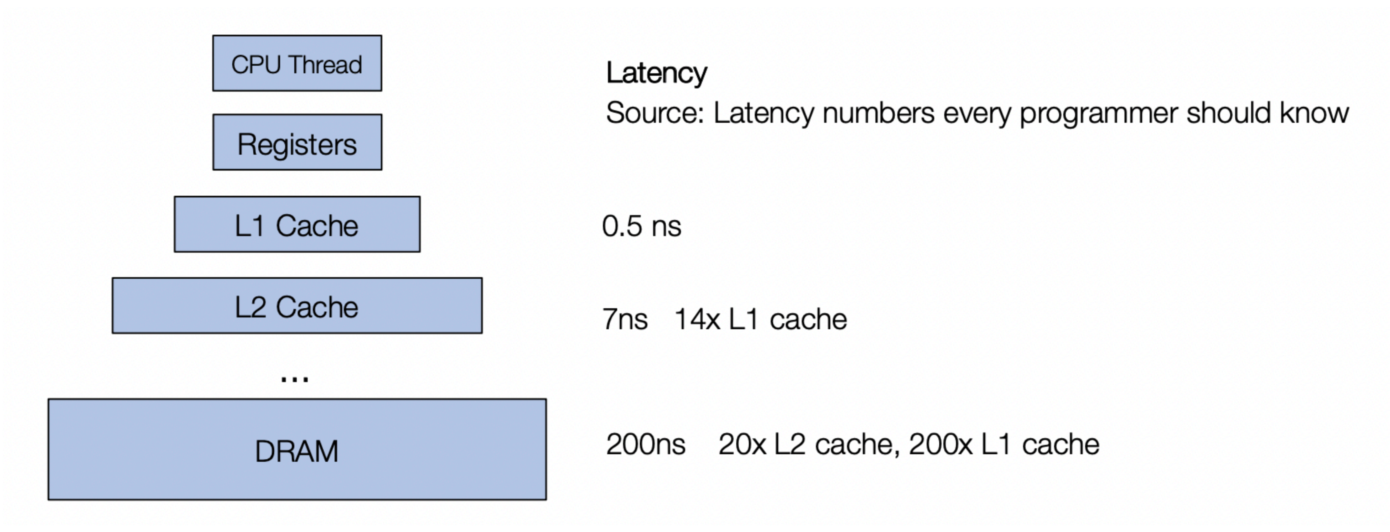
Vanilla Matrix Multiplication

最普通的矩阵乘法，时间复杂度为 $O(n^3)$



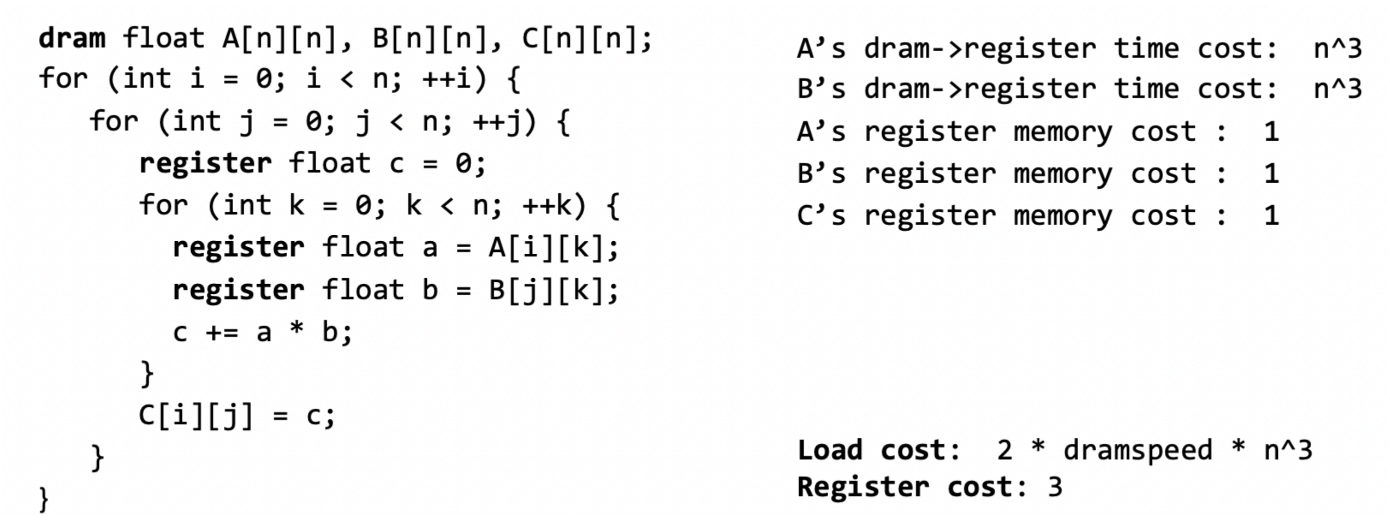
```
Compute C = dot(A, B.T)


float A[n][n], B[n][n], C[n][n];

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

可以有复杂度更低的算法，但在高性能计算库中并不一定采用更好的算法，而是根据内存架构进行优化，提升数据访问速度。


CPU Thread
Registers
L1 Cache
L2 Cache
...
DRAM
Latency
Source: Latency numbers every programmer should know
0.5 ns
7ns 14x L1 cache
200ns 20x L2 cache, 200x L1 cache

Architecture Aware Analysis

分析寄存器数量开销和数据加载开销



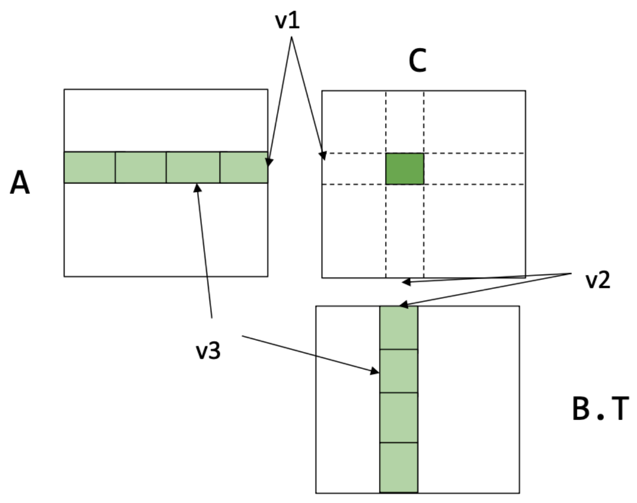
```
dram float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      register float c = 0;
      for (int k = 0; k < n; ++k) {
        register float a = A[i][k];
        register float b = B[j][k];
        c += a * b;
      }
      C[i][j] = c;
   }
}
```

```
A’s dram->register time cost:  n^3
B’s dram->register time cost:  n^3
A’s register memory cost :  1
B’s register memory cost :  1
C’s register memory cost :  1

Load cost:  2 * dramspeed * n^3
Register cost: 3
```

Register Tiled Matrix Multiplication

核心思想：将矩阵乘法计算的单位从一个元素扩展为一个小矩阵乘法（分片），从而达到更多的数据复用，减小数据加载开销。

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
   for (int j = 0; j < n/v2; ++j) {
      register float c[v1][v2] = 0;
      for (int k = 0; k < n/v3; ++k) {
        register float a[v1][v3] = A[i][k];
        register float b[v2][v3] = B[j][k];
        c += dot(a, b.T);
      }
      C[i][j] = c;
   }
}
```



```
A’s dram->register time cost:   n^3/v2
B’s dram->register time cost:   n^3/v1
A’s register memory cost:   v1*v3
B’s register memory cost:   v2*v3
C’s register memory cost:   v1*v2
```

```
load cost:  dramspeed * (n^3/v2 + n^3/v1)
Register cost: v1*v3 + v2*v3 + v1*v2
```

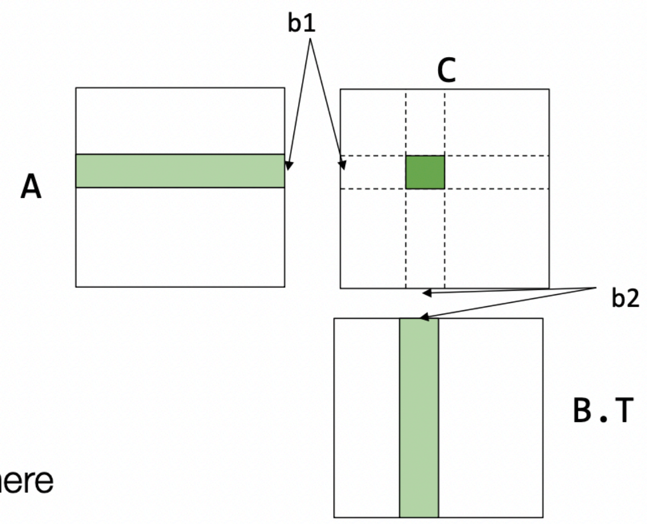
寄存器数量增加，但数据加载开销减小。

Cache Line Aware Tiling

核心思想：在 register tiling 的基础上将 cache 考虑进来，先把对应的行块和列块加载到 cache 中，再执行有 register tiling 的乘法。

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```

Sub-procedure, can apply register tiling here



```
A’s dram->l1 time cost:   n^2
B’s dram->l1 time cost:   n^3 / b1

Constraints:
 -   b1 * n + b2 * n < l1 cache size
 -   To still apply register blocking on dot
   -   b1 % v1 == 0
   -   b2 % v2 == 0
```

限制条件：加载数据不能超过 cache size, 要满足 tiling 所需的分块要求

Putting it together

```
dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1/v1][n][v1] = A[i];
   for (int j = 0; j < n/b2; ++j) {
     l1cache b[b2/v2][n][v2] = B[j];
     for (int x = 0; x < b1/v1; ++x)
       for (int y = 0; y < b2/v2; ++y) {
         register float c[v1][v2] = 0;
         for (int k = 0; k < n; ++k) {
            register float ar[v1] = a[x][k][:];
            register float br[v2] = b[y][k][:];
            C += dot(ar, br.T)
         }
       }
   }
}
```

```
load cost:

l1speed * (n^3/v2 + n^3/v1) +
dramspeed * (n^2 + n^3/b1)
```

关键点在于内存加载的复用，`a` 被复用了 `v2` 次，`b` 被复用了 `v1` 次，因此相应的数据加载开销减小。

Common Reuse Patterns

一般地，如果对数组 `A` 的访问与索引 `j` 无关，那么将 `j` 对应的维度进行 v - 分片，则可对 `A` 复用 v 次。