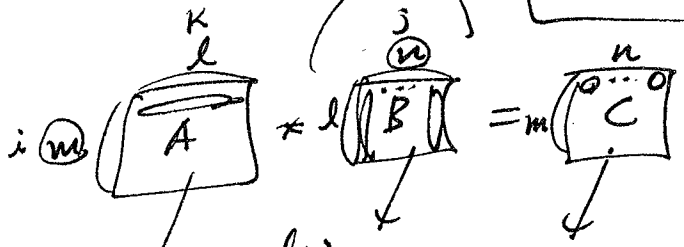
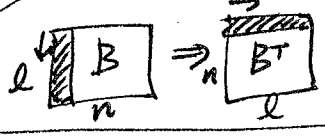
2018 book

Listing 3.1 matrix mult

$A * B = C$



(a lot of cache misses ⇒ B^T way accem (less cache misses & spatial locality)



↓ stored in linear array (row-major)

row-major layout (C++)

@ $A_{i,j} = (i * \#col) + j$ (l)

— naive mult

```
for (i=0 ; i<m ; i++)
    for (j=0; j<n ; j++)
        Acc=0;
        for (k=0; k<l : k++)
            Acc += A[i][k] * B[k][j]; — dot product
        C[i][j]=Acc ;
```

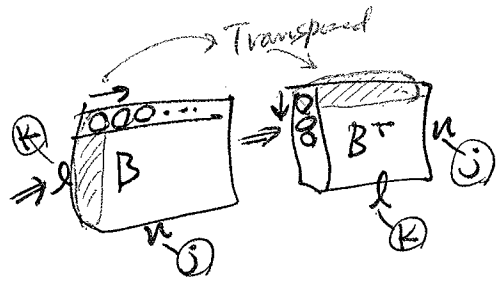
$= A[i*l+k]$ (#cols) $B[k*n+j]$ (#cols)

$C[i*n+j]$ ← row-major layout

— transpose $B \rightarrow (B^T)$ and mult.

```
for (k=0; k<l; k++)
    for (j=0; j<n; j++)
        BT[j*l+k] = B[k*n+j];
        BT[j][k]
```

(#cols) (#cols) $= B[k][j]$



↓

↓

— transposed_mult

```
for (i=0; i<m; i++)
    for(j=0; j<n; j++)
    Acc=0;
    for (k=0; k<l; k++)
        Acc += A[i*l+k] * BT[j*l+k];
```

A[i][k]

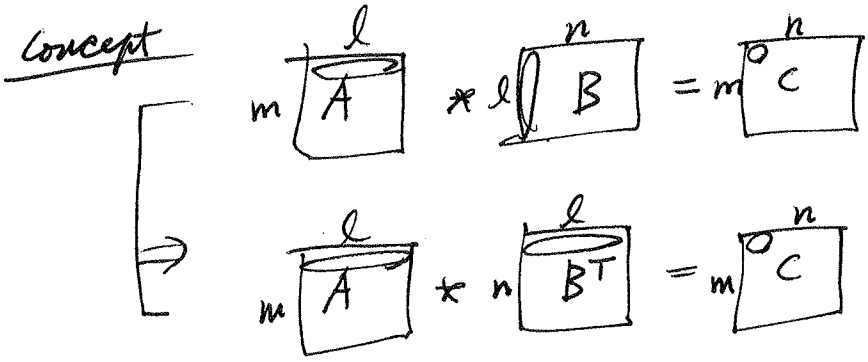
BT[j][k] — row-major access (less cache miss penalty)

≡ B[k][j] — col-major access (more penalty)

(Same entry)

---

concept



performance

— on i7, $m=n=l=2^{13}$

vs. [naive mult — 5559.46 sec

transposed_mult — 496.66 sec

transpose : 0.752 sec

mult : 497.91 sec