




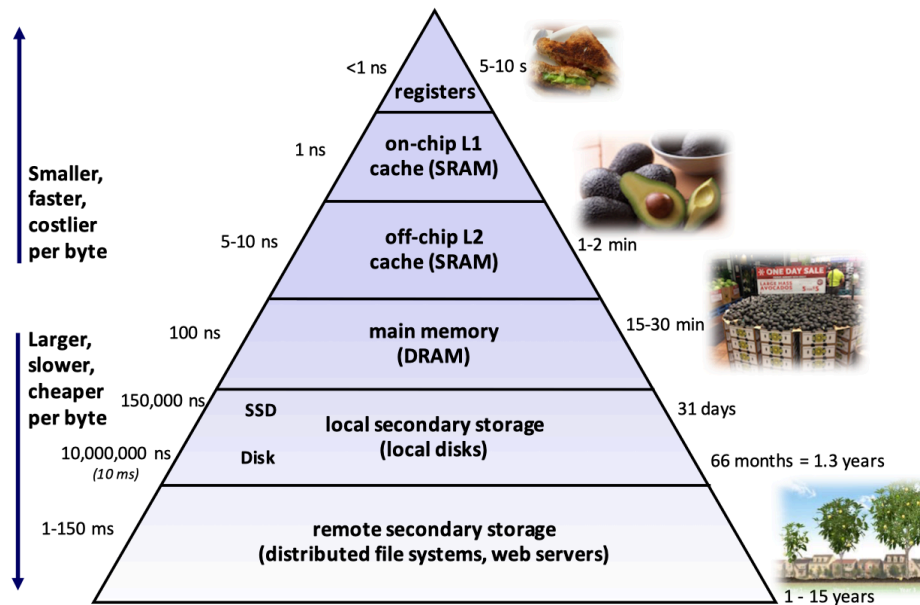
CS 211 RECITAIONS

WEEK 11

Wenjie Qiu
Teaching Assistant
Office hour: Thu noon – 1pm
wenjie.qiu@rutgers.edu

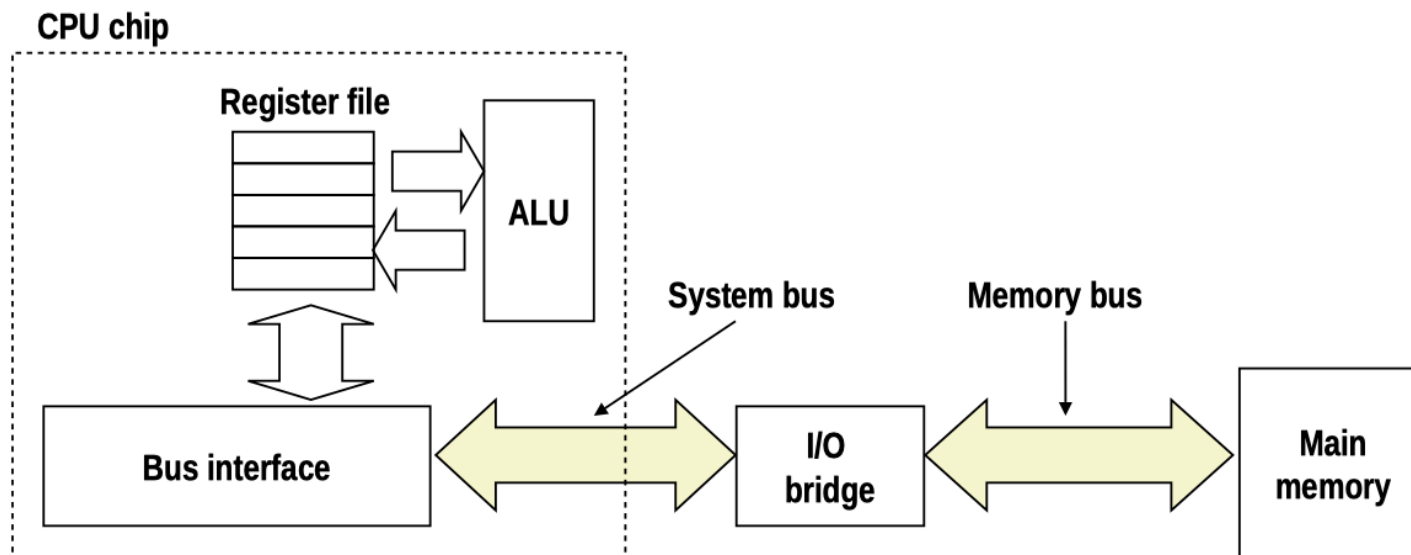


An Example Memory Hierarchy



MEMORY HIERARCHY

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.

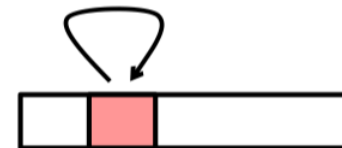


Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

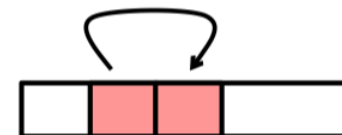
- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**

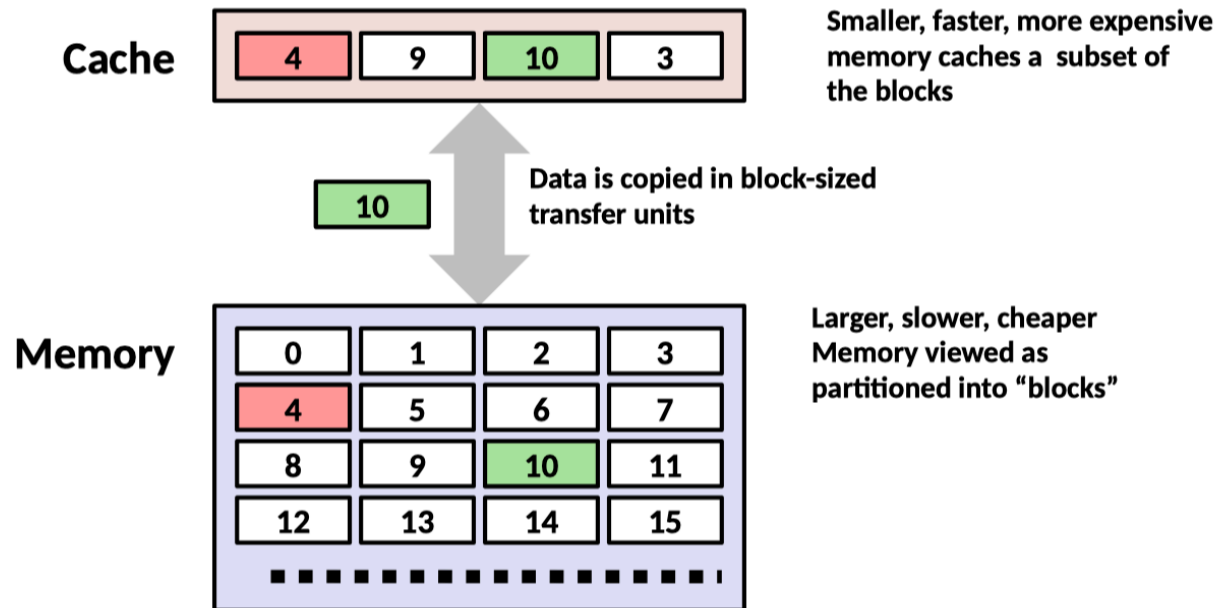
- Items with nearby addresses tend to be referenced close together in time



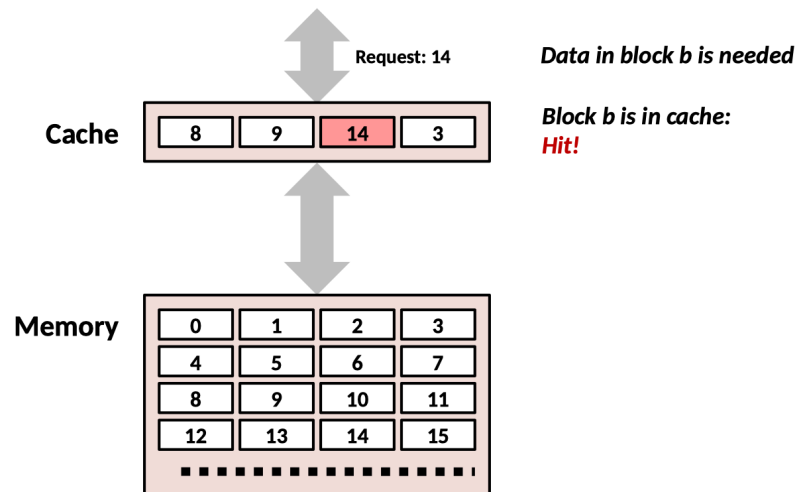
Caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- **Fundamental idea of a memory hierarchy:**
 - For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.
- **Why do memory hierarchies work?**
 - Because of locality, programs tend to access the data at level k more often than they access the data at level $k+1$.
 - Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.
- **Big Idea:** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

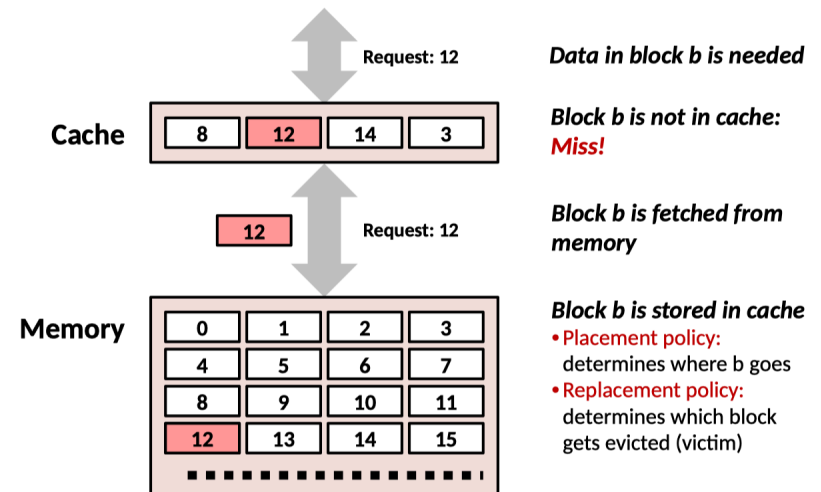
General Cache Concepts



General Cache Concepts: Hit

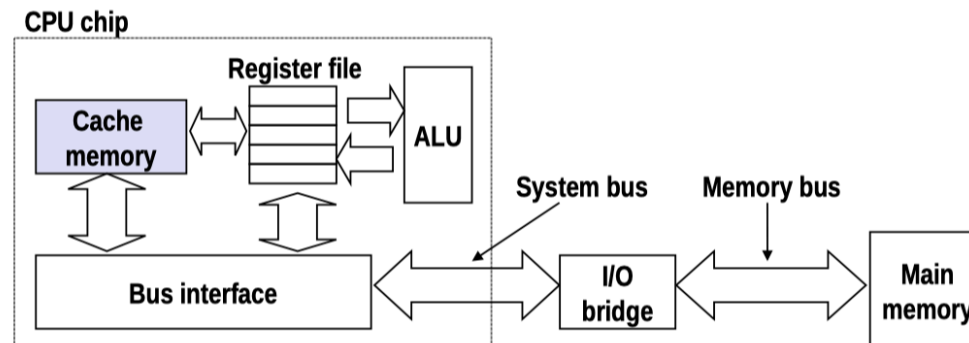


General Cache Concepts: Miss

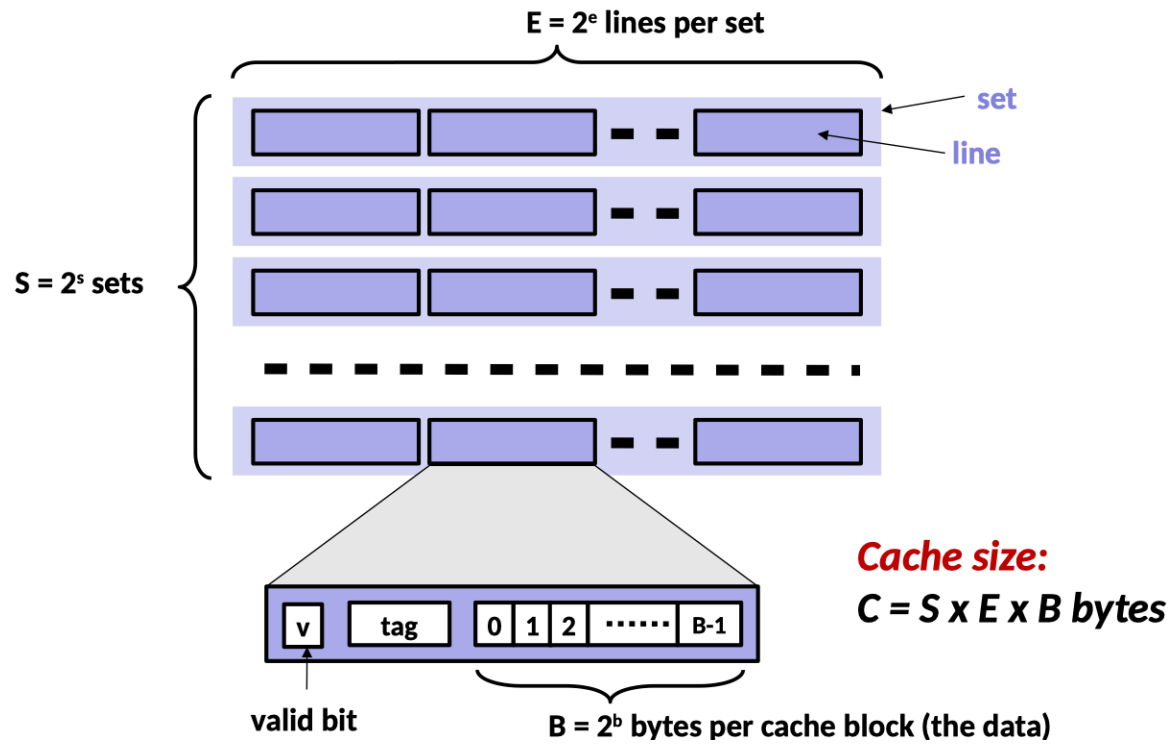


Cache Memories

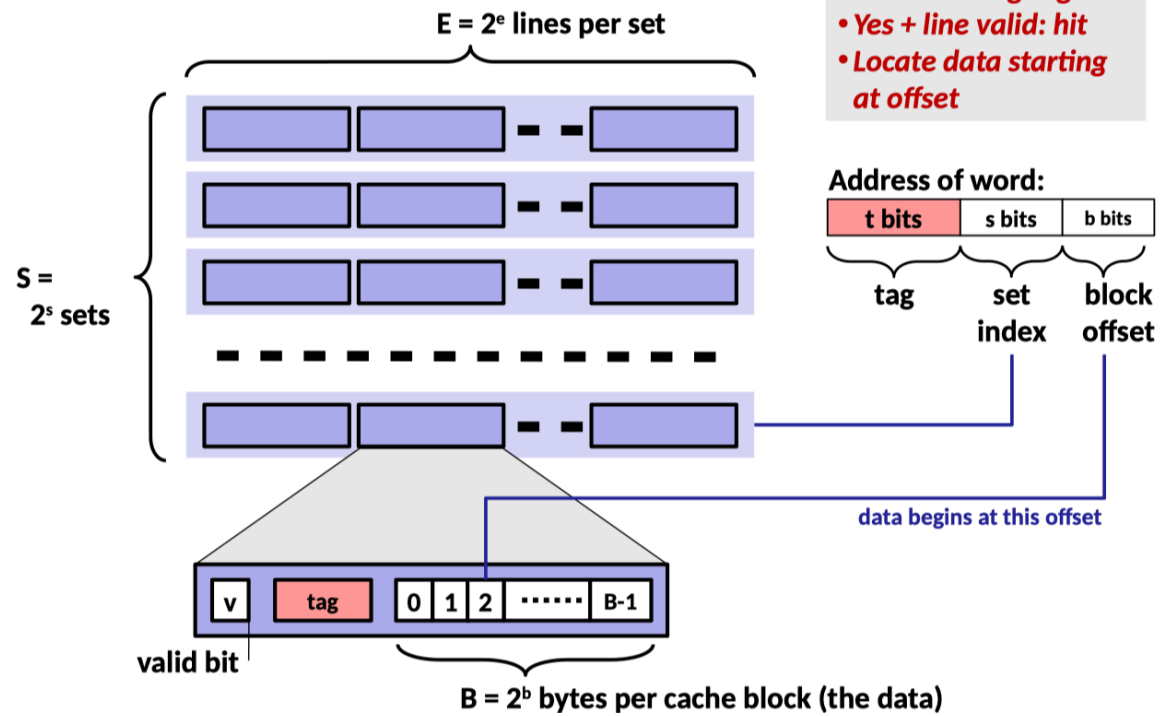
- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
 - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



General Cache Organization (S, E, B)



Cache Read



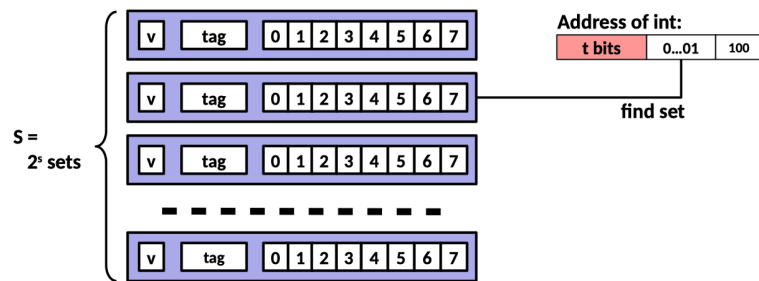
- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Cache associativity

- **Direct mapped**
 - $E = 1$ (1 line per set)
- **n-way associative**
 - n lines per set
- **Fully associative**
 - $S = 1$ (1 set containing all lines)

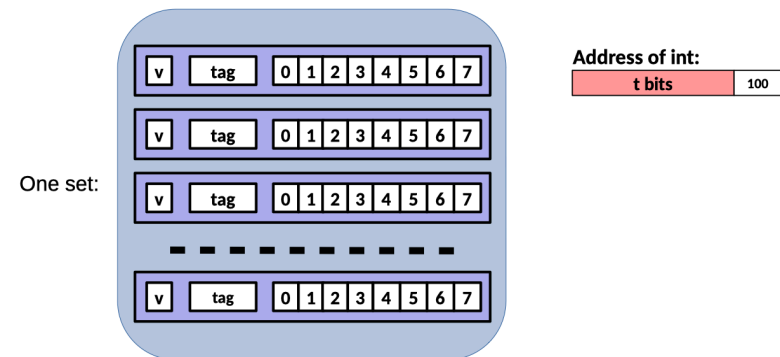
Ex.: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Ex.: Fully Associative Cache (S = 1)

One set containing all lines
Assume: cache block size 8 bytes



Cache Size

■ Suppose we're given:

- 32 KB cache
- 64 B line
- 4-way set associative
- 32-bit architecture

■ How many tag, set, block index bits?

- 64 B per block = 6 bits
- 32 KB cache / 64 B per line = 2^9 lines
- 2^9 lines / 4 lines per set = 2^7 sets
- So 6 block index bits, 7 set index bits
- $32 - 6 - 7 = 19$ tag bits

Cache Size

■ Suppose we're given:

- Read 0x2b74ce2d

■ What are the tag, set, block index?

- (19 tag, 7 set, 6 block)
- 0x2b74ce2d =

0010 1011 0111 0100 1100 1110 0010 1101

- Tag = 0010101101110100110
- Set index = 0111000 (56)
- Block index = 101101 (45)

What about writes?

- **Multiple copies of data exist:**
 - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
 - **Write-through** (write immediately to memory)
 - **Write-back** (defer write to memory until replacement of line)
 - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
 - **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location follow
 - **No-write-allocate** (writes straight to memory, does not load into cache)
- **Typical**
 - Write-through + No-write-allocate
 - Write-back + Write-allocate

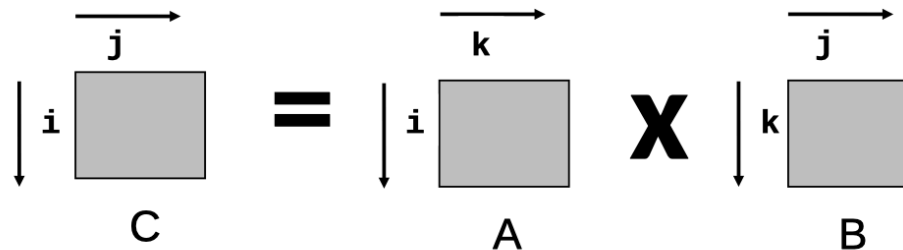
Miss Rate Analysis for Matrix Multiply

■ Assume:

- Block size = 32B (big enough for four doubles)
- Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
- Cache is not even big enough to hold multiple rows

■ Analysis Method:

- Look at access pattern of inner loop

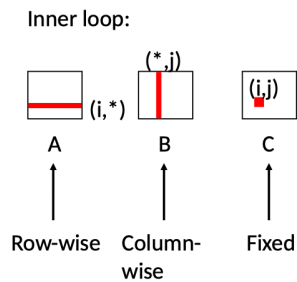


Matrix Multiplication (ijk)

```

/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

```

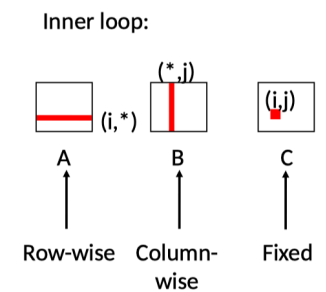


Matrix Multiplication (jik)

```

/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

```



Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {  
  for (j=0; j<n; j++) {  
    sum = 0.0;  
    for (k=0; k<n; k++)  
      sum += a[i][k] * b[k][j];  
    c[i][j] = sum;  
  }  
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (k=0; k<n; k++) {  
  for (i=0; i<n; i++) {  
    r = a[i][k];  
    for (j=0; j<n; j++)  
      c[i][j] += r * b[k][j];  
  }  
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

```
for (j=0; j<n; j++) {  
  for (k=0; k<n; k++) {  
    r = b[k][j];  
    for (i=0; i<n; i++)  
      c[i][j] += a[i][k] * r;  
  }  
}
```

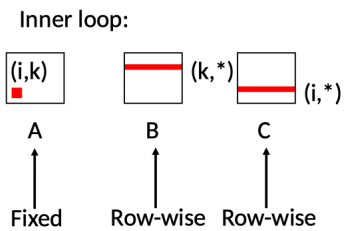
jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

