Memory Hierarchies

- One of the great triumphs of computer design
- Effect is a large, fast memory
- Reality is a series of progressively larger, slower, cheaper stores, with frequently accessed data automatically staged to faster storage (cache, main storage, disk)
- Programmer/compiler typically treats it as one large store. Bug or feature?
Memory Issues (review)

- Byte load/store is often slower than whole (physical) word load/store
  - Unaligned access is often extremely slow
- Temporal locality: accesses to recently accessed data will usually find it in the (fast) cache
- Spatial locality: accesses to data near recently used data will usually be fast
  - “near” = in the same cache block
- But – alternating accesses to blocks that map to the same cache block will cause thrashing
Data Alignment

- Data objects (structs) often are similar in size to a cache block ($\approx 8$ words)
  - Better if objects don’t span blocks
- Some strategies
  - Allocate objects sequentially; bump to next block boundary if useful
  - Allocate objects of same common size in separate pools (all size-2, size-4, etc.)
- Tradeoff: speed for some wasted space
Instruction Alignment

- Align frequently executed basic blocks on cache boundaries (or avoid spanning cache blocks)
- Branch targets (particularly loops) may be faster if they start on a cache line boundary
- Try to move infrequent code (startup, exceptions) away from hot code
- Optimizing compiler should have a basic-block ordering phase (& maybe even loader)
Loop Interchange

- Watch for bad cache patterns in inner loops; rearrange if possible
- Example
  
  ```
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < p; k++)
        a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
  ```
  
  - `b[i,j+1,k]` is reused in the next two iterations, but will have been flushed from the cache by the k loop
Loop Interchange

Solution for this example: interchange j and k loops

\[
\text{for } (i = 0; i < m; i++) \\
\text{for } (k = 0; k < p; k++) \\
\text{for } (j = 0; j < n; j++) \\
\text{a}[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
\]

- Now b[i,j+1,k] will be used three times on each cache load
- Safe here because loop iterations are independent
Loop Interchange

- Need to construct a data-dependency graph showing information flow between loop iterations.
- For example, iteration \((j,k)\) depends on iteration \((j',k')\) if \((j',k')\) computes values used in \((j,k)\) or stores values overwritten by \((j,k)\).
  - If there is a dependency and loops are interchanged, we could get different results – so can’t do it.
Blocking

- Consider matrix multiply
  
  ```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      c[i,j] = 0.0;
      for (k = 0; k < n; k++)
        c[i,j] = c[i,j] + a[i,k]*b[k,j]
    }
  ```

- If a, b fit in the cache together, great!
- If they don’t, then every b[k,j] reference will be a cache miss
- Loop interchange (i<->j) won’t help; then every a[i,k] reference would be a miss
Blocking

- Solution: reuse rows of A and columns of B while they are still in the cache
- Assume the cache can hold $2^c n$ matrix elements ($1 < c < n$)
- Calculate $c \times c$ blocks of C using c rows of A and c columns of B
Blocking

- Calculating $c \times c$ blocks of $C$
  
  for (i = i0; i < i0+c; i++)
    
    for (j = j0; j < j0+c; j++) {
      
      c[i,j] = 0.0;
      
      for (k = 0; k < n; k++)
        
        c[i,j] = c[i,j] + a[i,k]*b[k,j]
  
  
}
Blocking

- Then nest this inside loops that calculate successive $c \times c$ blocks

```c
for (i0 = 0; i0 < n; i0 += c)
    for (j0 = 0; j0 < n; j0 += c)
        for (i = i0; i < i0 + c; i++)
            for (j = j0; j < j0 + c; j++) {
                c[i, j] = 0.0;
                for (k = 0; k < n; k++)
                    c[i, j] = c[i, j] + a[i, k] * b[k, j]
            }
```
Parallelizing Code

- There is a long literature about how to rearrange loops for better locality and to detect parallelism

- Some starting points
  - Latest edition of *Dragon book*, ch. 11
  - Allen & Kennedy *Optimizing Compilers for Modern Architectures*
  - Wolfe, *High-Performance Compilers for Parallel Computing*