

CS 4400

Computer Systems

LECTURE 13

More on caches

Writing cache-friendly code

Write Policy

- Writes are a special case of data access that doesn't immediately require the data
- What happens when the CPU writes data at address x ?
 - If x is in the cache, it is a ***write-hit***.
 - If x is not in the cache, it is a ***write-miss***.

Write-Hit Policy

- ***Write-through***: Main updated at the time of the hit
- ***Write-back***: Data only written when the cache block is evicted from the cache.
- What are the consequences of each policy?
- Which policy requires a ***dirty bit***?

Write-Miss Policy

- ***Fetch-on-write*** (or ***write-allocate***): Word at x is written to cache and the other words in the block are fetched from main memory.
 - Why does this work?
- ***Write-around***: Word at x is written directly to main memory.
 - What happens next time x is required?

Types of Caches

- Caches can hold
 - only instructions (*i-caches*),
 - only data (*d-caches*),
 - or both instructions and data (*unified caches*)
- Typical desktop systems have: L1 i-cache/L1 d-cache/Unified L2 cache/Unified L3 cache
- What is the advantage of separate i-/d-cache?
- What is the advantage of a unified cache?

Cache Performance Metrics

- **Miss rate:** fraction of memory references that miss
 - # of misses / total # of references
- **Hit rate:** fraction of memory references that hit
 - # of hits / total # of references OR $1 - \text{miss rate}$

Cache Performance Metrics

- ***Hit time***: time to deliver a word from cache to CPU
 - includes set selection, frame (or line) id, word selection
 - typically a few cycles for L1 caches
- ***Miss penalty***: additional time required because of a miss
 - penalty for L1 misses (served from L2) is typically 5-10 cycles
 - penalty for L2 (served from main memory) typically 25-100

Intel Ivy Bridge (& Haswell)

- L1 Data Cache Latency = 4/5 cycles for simple/complex address calculation
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 30 cycles (Haswell: 36 cycles)
- RAM Latency = 30 cycles + 53 ns (Haswell: 36 cycles + 57 ns)
- From: <http://www.7-cpu.com/cpu/IvyBridge.html> & <http://www.7-cpu.com/cpu/Haswell.html>

Impact of Size

- Larger cache capacity
 - **PRO**: increases hit rate
 - **CON**: increases the hit time and expense

Impact of Size

- Larger block size
 - **PRO**: can increase hit rate (exploiting spatial locality)
 - **CON**: decreases the number of cache frames (bad if temporal locality outweighs spatial)
 - **CON**: increases the miss penalty (larger transfer time)
- Typical compromise: **B** = 4 to 8 words

Impact of Associativity

- Higher associativity (larger values of E)
 - **PRO**: decreases cache's vulnerability to thrashing due to conflict misses
 - **CON**: increases the hit time (more tag comparisons and additional eviction state bits)
 - **CON**: increases the miss penalty (increases complexity of choosing which cache frame to evict)
- Typically $E = 1-8$ for L1-L3 caches

Writing Cache-Friendly Code

- Make the common case fast
 - Programs often spend most of their time in a few core functions which spend their time in a few loops
- Minimize the number of cache misses in each inner loop
 - Assuming that all other things are equal (such as total number of memory references), loops with better miss rates run faster

Writing Cache-Friendly Code

```
int sumvec(int v[N]) {  
    int i, sum = 0;  
  
    for(i = 0; i < N; i++)  
        sum += v[i];  
  
    return sum;  
}
```

- Compiler can cache `i` and `sum` in registers
- Stride-1 pattern is good for spatial locality
- If $B=16$, 3 out of 4 references will hit (best possible with cold cache)

Exercise: Cache-Friendly Code

```
struct {
    int x;
    int y;
} grid[16][16];

int total_x = 0, total_y = 0;
int i, j;

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++)
        total_x += grid[i][j].x;

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++)
        total_y += grid[i][j].y;
```

- Assume a DM cache with:
 - $C=1024$
 - $B=16$
 - $S=64$
- Array grid requires 2048 bytes, why?
- But, the cache can hold only half of the array.

Exercise: Cache-Friendly Code

```
struct {
    int x;
    int y;
} grid[16][16];

int total_x = 0, total_y = 0;
int i, j;

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++)
        total_x += grid[i][j].x;

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++)
        total_y += grid[i][j].y;
```

- Total number of reads?
- Number of misses?
- Miss rate?
- How can the number of misses be reduced?

Exercise: Cache-Friendly Code

```
int gridX[16][16];
int gridY[16][16];

int total_x = 0, total_y = 0;
int i, j;

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++)
        total_x += gridX[i][j];

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++)
        total_y += gridY[i][j];
```

- Total number of reads?
- Number of misses?
- Miss rate?
- How can the number of misses be reduced?

Question

```
struct {
    int x;
    int y;
} grid[16][16];

int total_x = 0, total_y = 0;
int i, j;

for(i = 0; i < 16; i++)
    for(j = 0; j < 16; j++) {
        total_x += grid[i][j].x;
        total_y += grid[i][j].y;
    }
```

- What is the miss rate?
 - B=16, C=1024, S=64
 - B=16, E=2, S=32
 - B=16, C=2048, S=128
 - B=32, E=1, C=2048

Memory References in Nested Loops

- Nested loops
 - Reuse the same word (temporal locality)
 - Use adjacent words (spatial locality)
- Innermost loop
 - Reuse causes many cache hits
- Outer loops
 - Reuses may be displaced by inner loops accesses

Example: Loop Nest

```
for(i = 0; i < N; i++)  
  for(j = 1; j < M-1; j++)  
    for(k = 0; k < P; k++)  
      A[i][j][k] = (B[i][j-1][k] + B[i][j][k] + B[i][j+1][k]) / 3;
```

- $B[i][j+1][k]$ is reused in the next two iterations of the j -loop.
 - How/Why?
- But before the next iteration of j -loop, the k -loop accesses $4 \cdot P$ array elements.
- It is possible that these accesses conflict with $B[i][j+1][k]$, causing a miss the next time it's fetched.

Loop Interchange

```
for(i = 0; i < N; i++)  
  for(k = 0; k < P; k++)  
    for(j = 1; j < M-1; j++)  
      A[i][j][k] = (B[i][j-1][k] + B[i][j][k] + B[i][j+1][k]) / 3;
```

- We can interchange the **j**-loop and the **k**-loop.
- Now **B[i][j][k]** and **B[i][j-1][k]** are highly likely to be cache hits.
- Is this loop interchange legal? (i.e., does it yield the same result?)
- Is loop interchange always legal?

Loop-Interchange Legality

```
for(i = 0; i < N; i++)
  for(j = 1; j < M-1; j++)
    for(k = 0; k < P-1; k++) {
      A[i][j][k] = (B[i][j-1][k] + B[i][j][k] + B[i][j+1][k]) / 3;
      C[i][j] = C[j-1][k+1];
    }
```

- Does interchanging the **j**-loop and the **k**-loop decrease the number of cache misses?
- Is interchanging the **j**-loop and the **k**-loop legal?
- In general, how can we tell if loop interchange is legal?
 - New value depends upon a previous value in an associative way

Example: Matrix Multiply

```
for(i = 0; i < N; i++)  
    for(j = 0; j < N; j++)  
        for(k = 0; k < N; k++)  
            C[i][j] += A[i][k] * B[k][j];
```

- Suppose $N=50$, each element is a double (8 bytes), and the cache capacity is 16 kilobytes.
- Every reference to `B[k][j]` in the innermost loop misses. All other elements of `B` map to the cache in between its use and reuse in the `i`-loop and there is no spatial locality.
- Will interchanging any of the loops help?

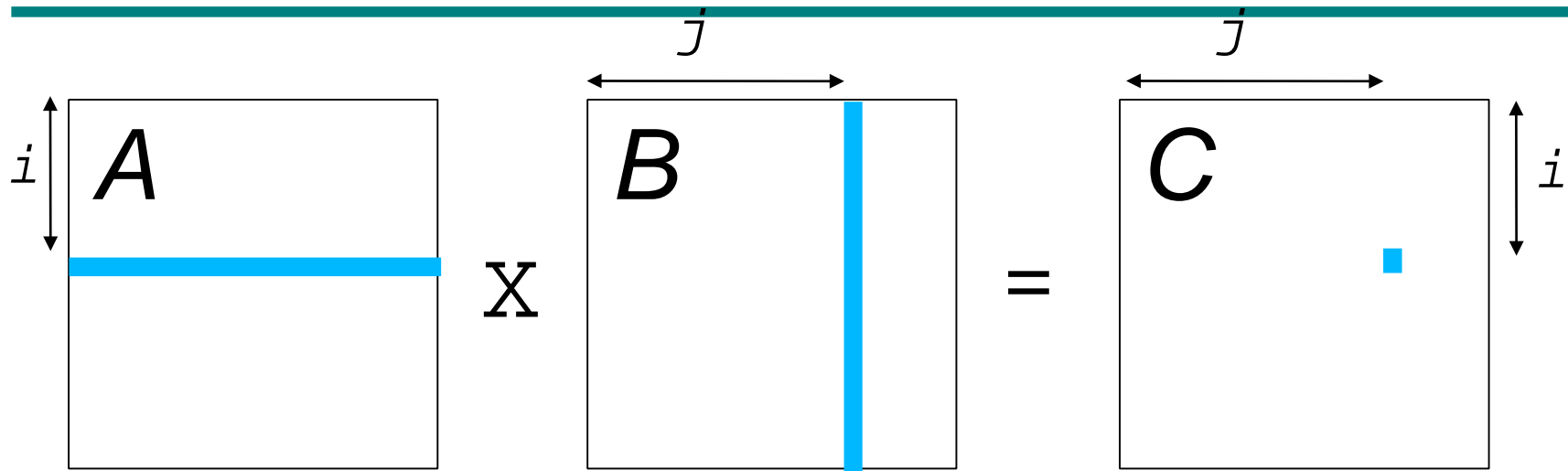
Working with Smaller Blocks

- The solution is to reuse rows of **A** and columns of **B** while they are still in the cache.
- A $c \times c$ block of **C** can be calculated from c rows of **A** and c columns of **B**.

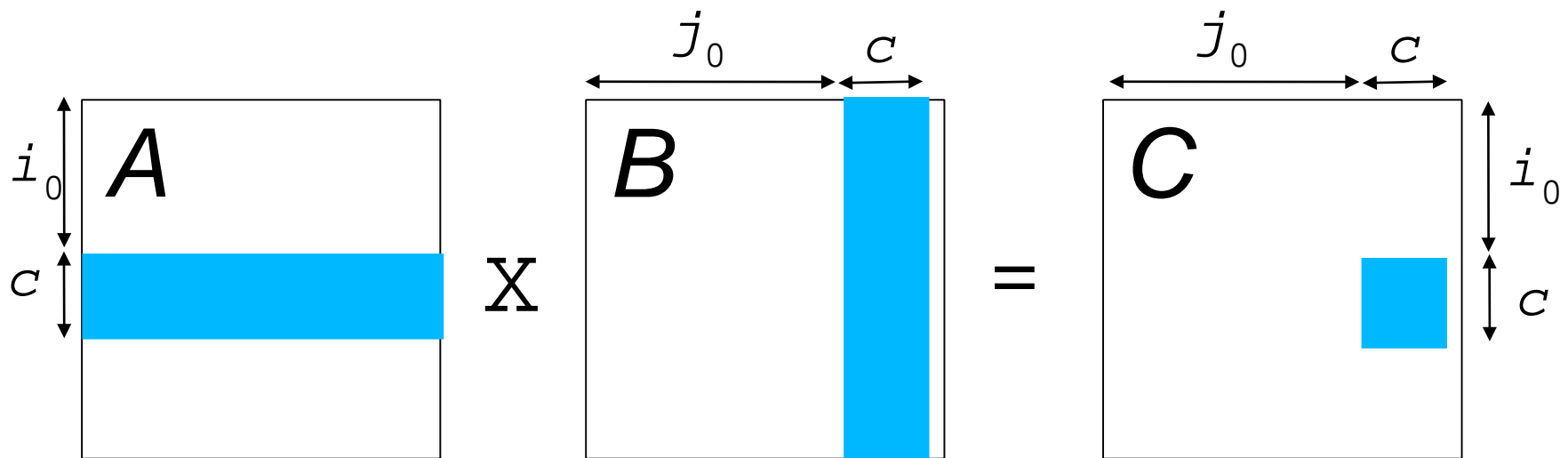
```
for(i = i_0; i < i_0 + c; i++)  
    for(j = j_0; j < j_0 + c; j++)  
        for(k = 0; k < N; k++)  
            C[i][j] += A[i][k] * B[k][j];
```

- Only $c \times N$ elements of **A** and $c \times N$ elements of **B** are used in this loop (each used c times).

Matrix-Multiply Blocks



Each element of C is computed from a row of A and a column of B .



Each $c \times c$ block of C is computed from a $c \times N$ block of A and a $N \times c$ block of B .

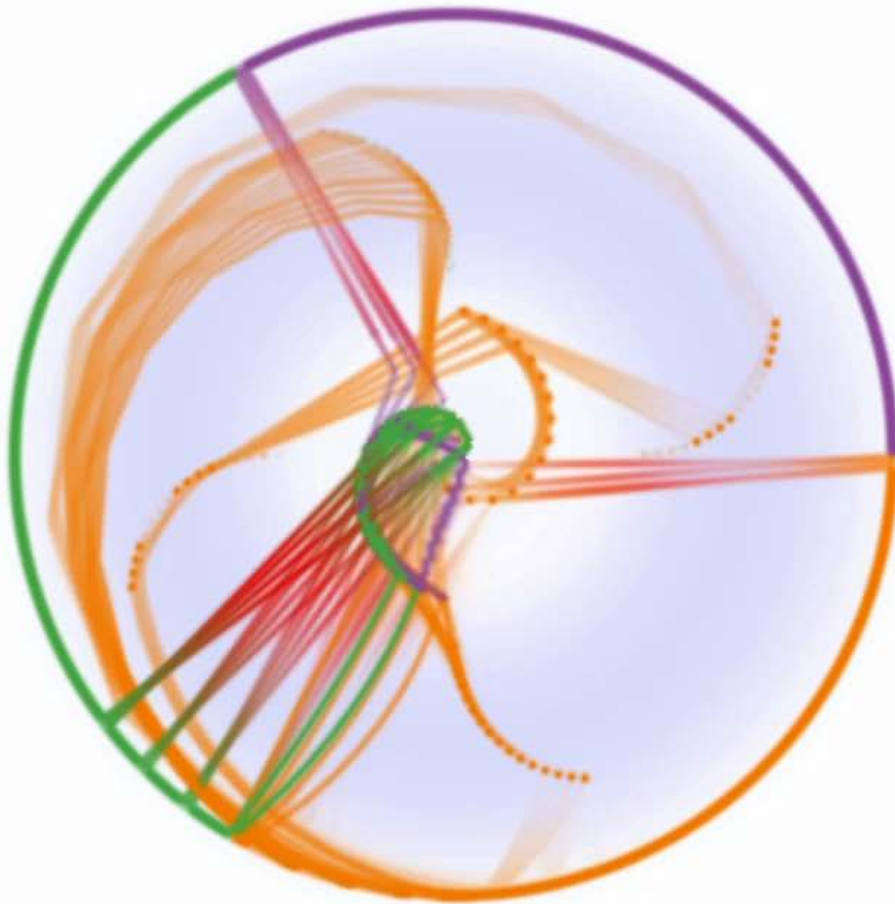
Blocking

- To compute each block of **C**, we need to set our loops for computing a single block of **C** inside some outer loops.

```
for(i_0 = 0; i_0 < N; i_0 += c)
    for(j_0 = 0; j_0 < N; j_0 += c)
        for(i = i_0; i < min(i_0 + c, N); i++)
            for(j = j_0; j < min(j_0 + c, N); j++)
                for(k = 0; k < N; k++)
                    C[i][j] += A[i][k] * B[k][j];
```

- The blocking transformation reorders computations so that all computations that use one portion (i.e., block) of data are computed before moving on to the next portion.
- How is **c** set? Should we really call the **min** function?

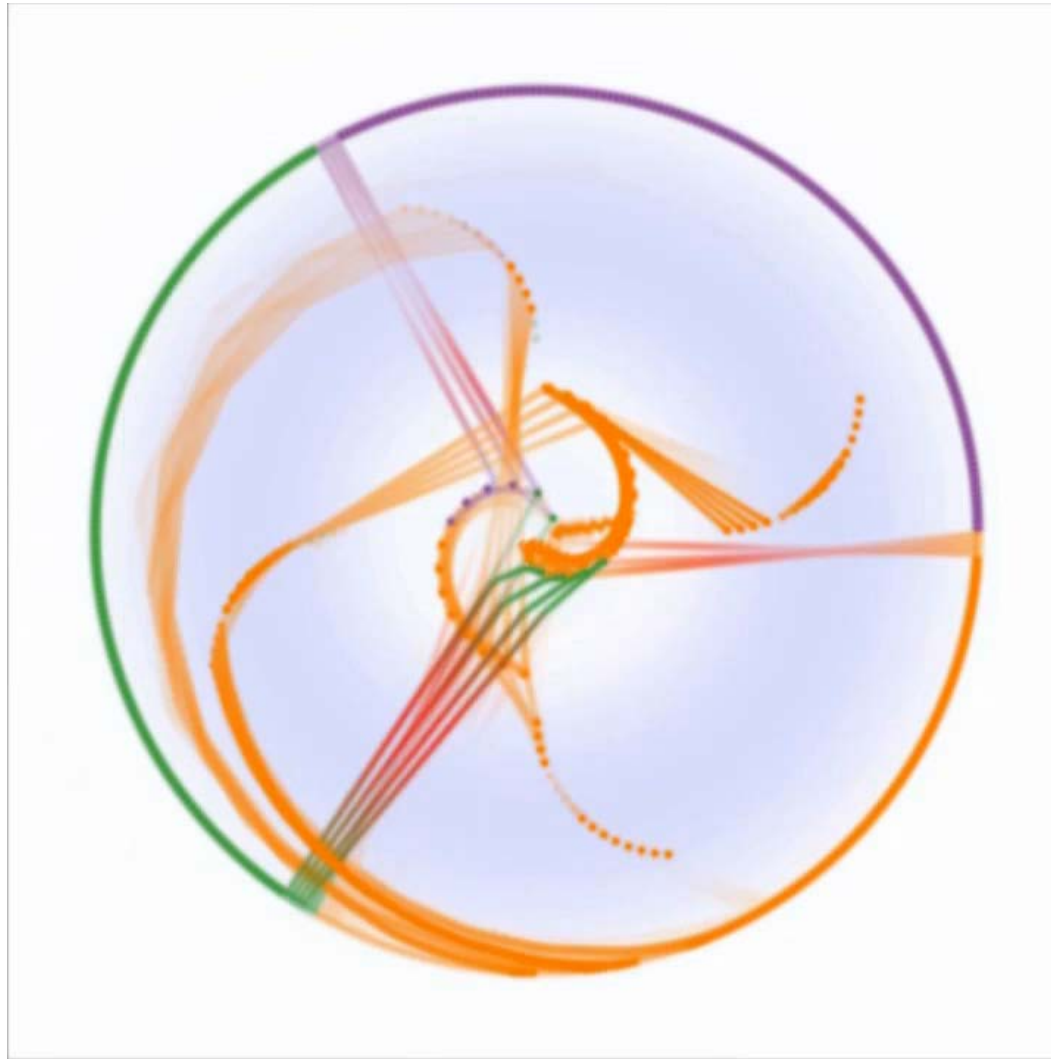
Naïve Matrix Multiply



File: matmult.cpp

```
    printf(";");  
  }  
  printf("\n");  
}  
  
void MatMult::linZero(float *A)  
{  
    for (unsigned int i = 0; i < N*N; i++)  
        A[i] = 0;  
}  
  
void MatMult::mdZero(float **A)  
{  
    for (unsigned int i = 0; i < N; i++)  
        for (unsigned int j = 0; j < N; j++)  
            A[i][j] = 0;  
}  
  
void MatMult::printLinA()  
{  
    linPrintResults(linA);  
}  
void MatMult::printLinB()  
{  
    linPrintResults(linB);  
}  
void MatMult::printLinC()  
{  
    linPrintResults(linC);  
}  
void MatMult::printMdA()  
{  
    mdPrintResults(mdA);  
}  
void MatMult::printMdB()  
{  
    mdPrintResults(mdB);  
}
```

Row Padding for Matrix Storage



Exploiting Locality

- Find code that matters using a profiler tool like Cachegrind, Intel PTU, etc.
- Focus attention on inner loops.
- Maximize spatial locality by reading data objects sequentially (in storage order).
- Maximize temporal locality by using a data object as often (and as soon) as possible once it has been read.
- Overall goal: Never load a block into the cache twice