# 15-213 Recitation 7: 10/21/02

## Outline

- Program Optimization
  - Machine Independent
  - Machine Dependent
    - Loop Unrolling
    - Blocking

## Reminder

- L4 due: Thursday 10/24, 11:59pm
- Submission is *online*


**Annie Luo**

**e-mail:**

luluo@cs.cmu.edu


**Office Hours:**

Thursday 6:00 – 7:00

Wean 8402

http://www2.cs.cmu.edu/afs/cs/academic/class/15213-f02/www/L4.html

# Machine *Independent* Optimization

- Code Motion
  - move repeating code out of loop
- Reduction in Strength
  - replace costly operations with simpler ones
  - keep data in registers rather than memory
  - avoid procedure call in loop, use pointers
- Share Common sub-expressions

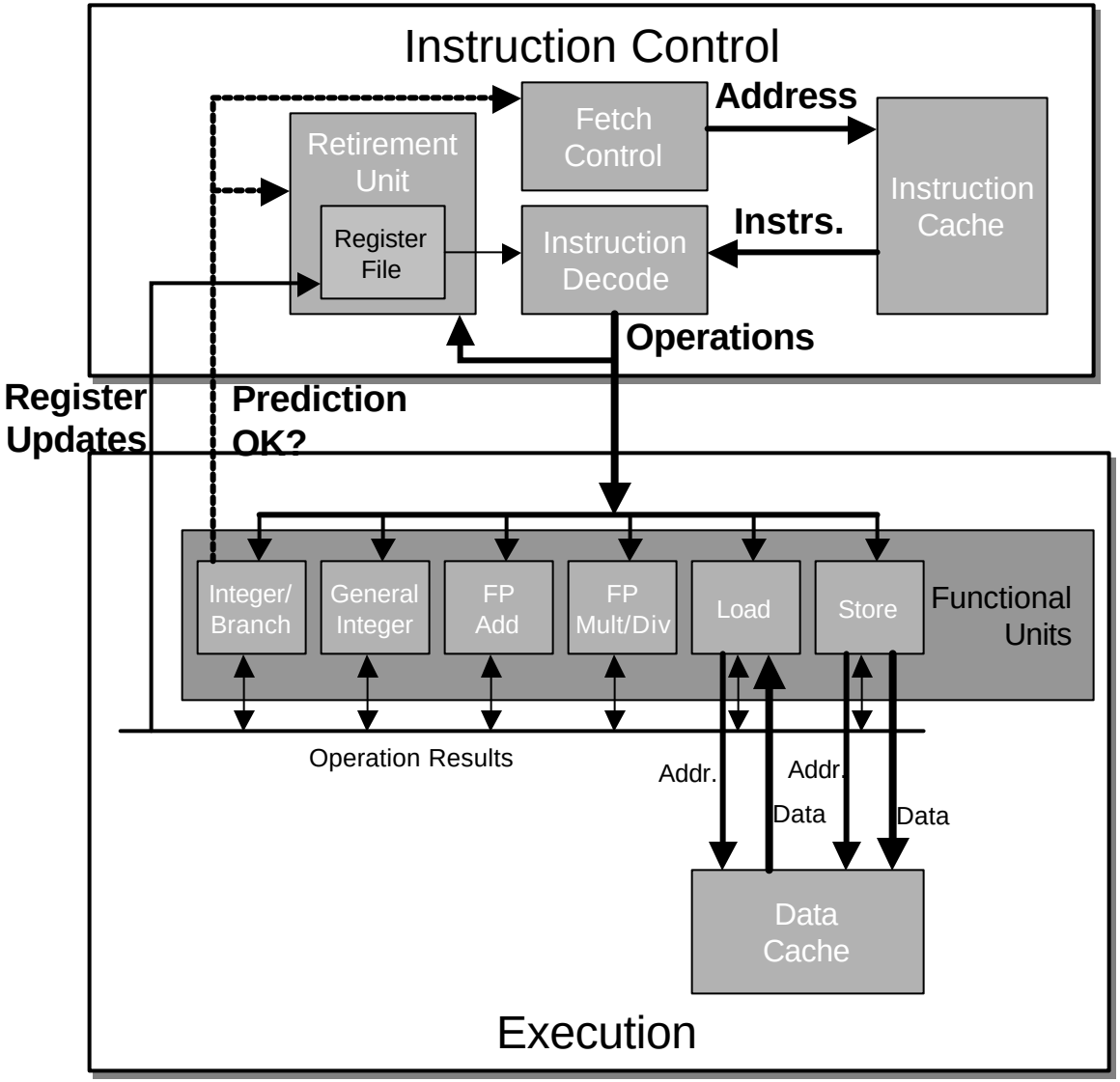
- *Procedure calls are expensive!*
- *Bounds checking is expensive!*
- *Memory references are expensive!*

# Machine *Dependent* Optimization

- Take advantage of system infrastructure

- Loop unrolling
- Blocking

# Loop Unrolling – Why We Can Do So?




- Superscalar: perform multiple operations on every clock cycle
- Out-of-order: executed instructions need not correspond to the assembly

# CPU Capability of Pentium III

- Multiple Instructions Can Execute in Parallel
  - 1 load
  - 1 store
  - 2 integer (one may be branch)
  - 1 FP Addition
  - 1 FP Multiplication or Division
- Some Instructions Take > 1 Cycle, but Can be Pipelined

| Instruction | Latency | Cycles/Issue |
|---|---|---|
| Load / Store | 3 | 1 |
| Integer Multiply | 4 | 1 |
| Integer Divide | 36 | 36 |
| Double/Single FP Multiply | 5 | 2 |
| Double/Single FP Add | 3 | 1 |
| Double/Single FP Divide | 38 | 38 |

# Loop Unrolling

```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* Combine 3 elements at a time */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+2]
           + data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

- Combine multiple iterations into single loop body
  - Perform more data operations in each iteration
- Amortize loop overhead across multiple iterations
  - Computing loop index
  - Testing loop condition
  - Make sure the loop condition NOT to run over array bounds
- Finish extras at end

# Practice Time ☺

- Work on practice problem 5.12 and 5.13

# Solution 5.12

```
void inner5(vec_ptr u, vec_ptr v, data_t *dest)
{
   int i;
   int length = vec_length(u);
   int limit = length-3;
   data_t *udata = get_vec_start(u);
   data_t *vdata = get_vec_start(v);
   data_t sum = (data_t) 0;

   /* Do four elements at a time */
   for (i = 0; i < limit; i+=4) {
        sum += udata[i] * vdata[i] + udata[i+1] * vdata[i+1] +
               udata[i+2] * vdata[i+2] + udata[i+3] * vdata[i+3];
   }

   /* Finish off any remaining elements */
   for (; i < length; i++)
        sum += udata[i] * vdata[i];
   *dest = sum;
}
```

# Solution 5.12

A. We must perform two loads per element to read values for udata and vdata. There is only one unit to perform these loads, and it requires one cycle.

B. The performance for floating point is still limited by the 3 cycle latency of the floating-point adder.

# Solution 5.13

```
void inner6(vec_ptr u, vec_ptr v, data_t *dest)
{
   int i;
   int length = vec_length(u);
   int limit = length-3;
   data_t *udata = get_vec_start(u);
   data_t *vdata = get_vec_start(v);
   data_t sum0 = (data_t) 0;
   data_t sum1 = (data_t) 0;
   /* Do four elements at a time */
   for (i = 0; i < limit; i+=4) {
        sum0 += udata[i] * vdata[i];
        sum1 += udata[i+1] * vdata[i+1];
        sum0 += udata[i+2] * vdata[i+2];
        sum1 += udata[i+3] * vdata[i+3];
   }
   /* Finish off any remaining elements */
   for (; i < length; i++)
      sum0 = sum0 + udata[i] * vdata[i];
   *dest = sum0 + sum1;
}
```

# Solution 5.13

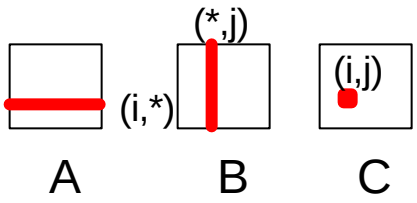
- For each element, we must perform two loads with a unit that can only load one value per clock cycle.
- We must also perform one floating-point multiplication with a unit that can only perform one multiplication every two clock cycles.
- Both of these factors limit the CPE to 2.

# Summary of Matrix Multiplication

## ijk (& jik):

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

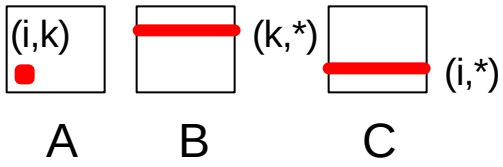
```
for (i=0; i<n; i++)  {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```




## kij (& ikj):

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

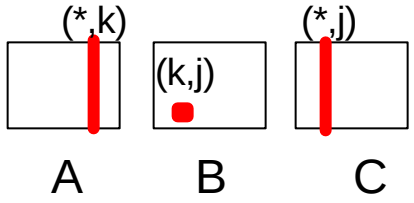
```
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```




## jki (& kji):

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

```
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;
    }
}
```



# Improve Temporal Locality by Blocking

- Example: Blocked matrix multiplication
  - "block" (in this context) does not mean "cache block"
  - instead, it means a sub-block within the matrix.
  - e.g: n = 8; sub-block size = 4 x 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

<u>Key idea:</u> Sub-blocks (i.e., $A_{xy}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

# Blocked Matrix Multiply (bijk)

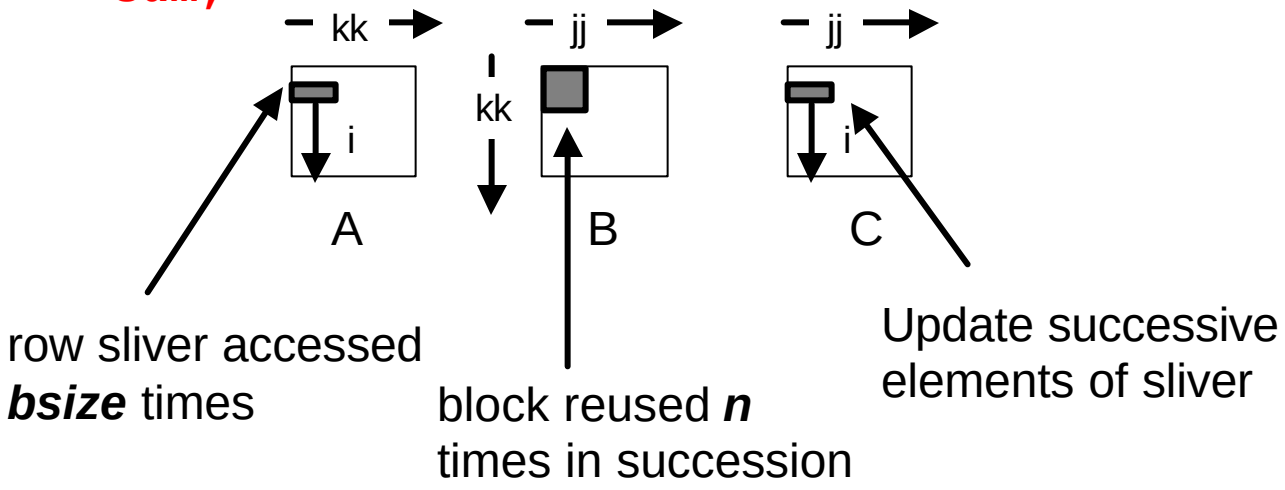
```
int en = n/bsize;
/*assume n is an integral multiple of bsize */

for(i=0; i<n; i++)
   for(j=0; j<n; j++)
       c[i][j] = 0.0;

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

# Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a *1 x bsize* sliver of *A* by a *bsize x bsize* block of *B* and accumulates into *1 x bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

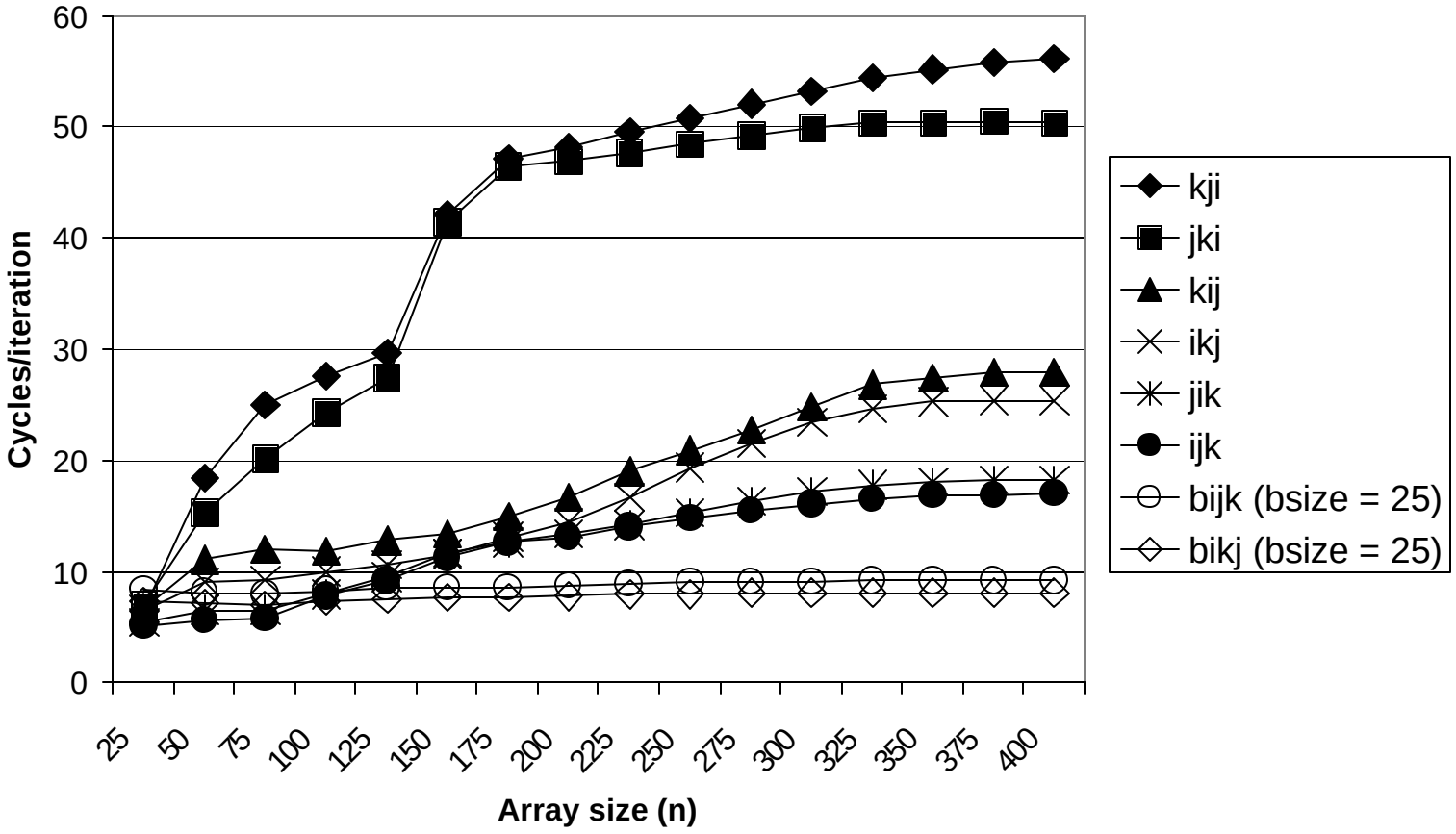
```
for (i=0; i<n; i++) {
    for(j=jj; j<jj+bsize; j++) {
        sum = c[i][j];
        for(k=kk; k<kk+bsize; k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
    }
}
```

Innermost Loop Pair



# Pentium Blocked Mat-Mul Performance

- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)
  - relatively insensitive to array size.



# Summary

- You can improve the performance of your program!
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)

- Mind writing “cache friendly code”
  - Absolute optimum performance is very platform specific: cache sizes, line sizes, etc.