

SysProg Programming contest 2001/2002

Prof. Gustavo Alonso
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
<http://www.inf.ethz.ch/departement/IS/iks/>

The task

- A modified form of matrix multiplication:

$$C = f(A, B)$$

$$f(A, B) :$$

$$c_{ij} = \sum_{k=0}^{N-1} \frac{j}{i+1} \cdot a_{ik} \cdot \frac{i}{j+1} \cdot b_{kj}$$

- The complicated part being that algebraic simplifications cannot be readily applied because of the way the testing is done:
 - ⌚ we test for equality of floating point numbers. Any small deviation (typical in floating point operations) results in numbers that are not equal
 - ⌚ The test being done has a type conversion problem that changes the nature of the operation being performed (the first term is an integer operation which is 0 in half of the cases)

Basic code

Matrix definition (matrix.h):

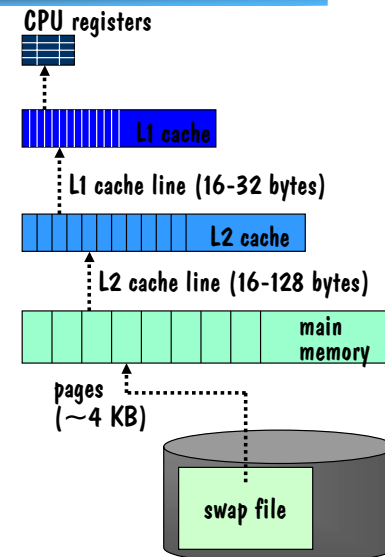
```
#define MATRIX(_a_) double _a_[N][N]
```

Basic loop (mmagic.c):

```
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] += j/(i+1)*a[i][k]*i/(j+1)*b[k][j];
  }
```

EXECUTION TIME:
482817

Caching



Cache architectures: review



□ Direct-mapped cache:

- ⌚ the memory address uniquely determines the corresponding cache line
- ⌚ two conflicting memory addresses evict each other from the cache (they have only one line for them)

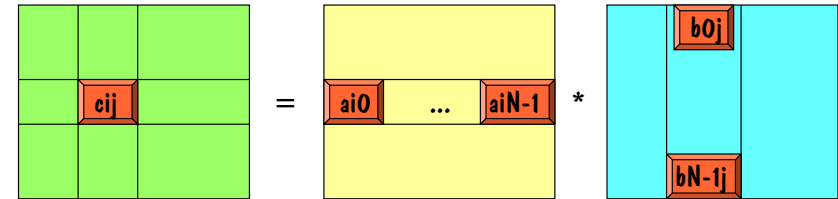
□ Fully associative:

- ⌚ a memory address can be stored anywhere in the cache
- ⌚ there are no conflicting addresses
- ⌚ expensive in terms of hardware since to find a memory address, all tags in the cache need to be searched

□ Set associative cache:

- ⌚ cache lines are divided into sets
- ⌚ memory addresses are mapped to sets (using middle bits, not the lower bits)
- ⌚ within a set, mapping is fully associative
- ⌚ two memory addresses conflict if they are mapped to the same set. However, within the same set there are several positions where the two addresses can go, thereby reducing the conflict
- ⌚ Typical set sizes are 4 (e.g., 4-way set-associative)

Matrix multiplication



$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} * b_{kj}$$

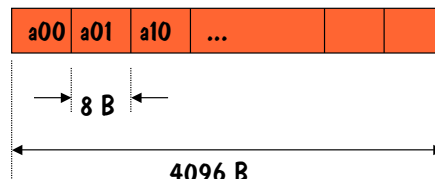
Associative caches



- In many processors, caches are set associative. This implies that a set of memory positions are mapped to the same set of cache positions

□ Typically:

- ⌚ cache line 32 bytes
- ⌚ in a 4-way set associative cache
 - addresses separated by multiples of 4096 compete for the same 4 L1 cache lines
 - addresses separated by 64 K compete for the same set of L2 cache lines



$c[i][j] += a[i][k]*b[k][j];$
the three variables are mapped to the same cache lines !!!

- Conflicts on the cache lead to thrashing behavior

Optimization 1: field padding



Matrix definition (matrix.h):

```
#define MATRIX(_a_) double _a_[N+1][N+1]
```

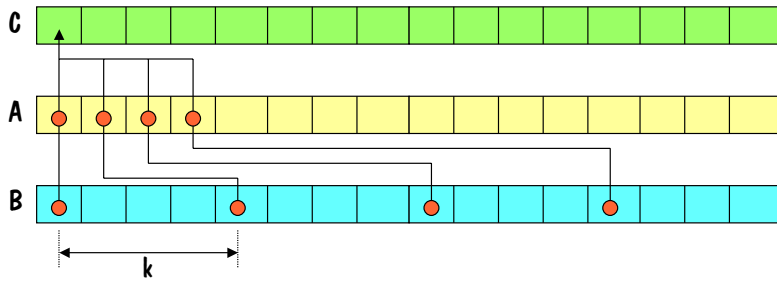


Basic loop (mmagic.c):

```
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] += j/(i+1)*a[i][k]*i/(j+1)*b[k][j];
  }
```

EXECUTION TIME:
303267

Cache behavior in 1 and 2 (simplified)



In the innermost loop, $c[i][j]$ is used several times. There is really no need to save it to memory until the innermost loop is over. For calculation purposes, it can be kept in a CPU register

Optimization 2: CPU register



Matrix definition (matrix.h):

```
#define MATRIX(_a_) double _a_[N+1][N+1]
```

Basic loop (mmagic.c):

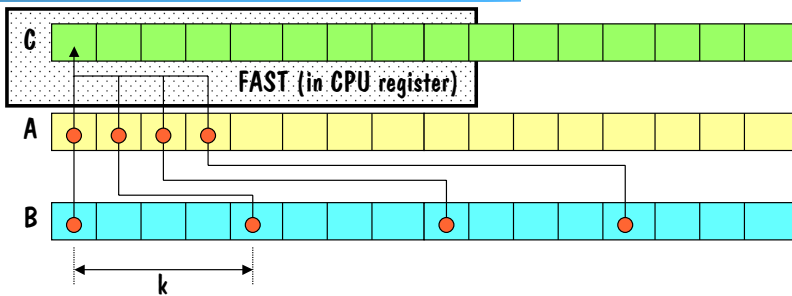
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
```



```
{
  double s = 0.0;
  for (k = 0; k < N; k++) s += j / (i+1) * a[i][k] * i / (j+1) * b[k][j];
  c[i][j] = s;
}
```

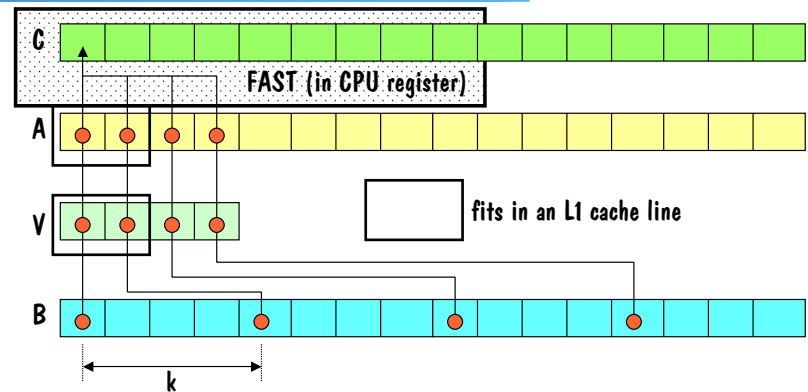
EXECUTION TIME:
258746

Cache behavior in program 3 (simplified)



In the innermost loop, every time we access a $b[k][j]$, we have a cache miss (we are going over the columns of the B matrix). The data in the B matrix could be re-arranged so that column values are adjacent and, that way, one can fetch several entries with every line cache

Sequential vs. stride access



By using an intermediate array (V) we solve two problems. V and A are not aligned with respect to each other in the cache (no conflicts for the same cache lines), and we fetch two values at a time with every cache line for both A and V.

Optimization 3: sequential access



Matrix definition (matrix.h):

```
#define MATRIX(_a_) double _a_[N+1][N+1]
```

Basic loop (mmagic.c):

```
for (j = 0; j < N; j++) {
    double v[N];
    for (i = 0; i < N; i++) v[i] = b[i][j];
    for (i = 0; i < N; i++) {
        double s = 0.0;
        for (k = 0; k < N; k++) s += j / (i+1) * a[i][k] * i / (j+1) * v[k];
        c[i][j] = s;
    }
}
```

EXECUTION TIME:
169080



Optimization 4: Loop unrolling - a



Basic loop (mmagic.c):

```
for (j = 0; j < N; j++) {
    double v[N];
    for (i = 0; i <= N; i++) v[i] = b[i][j];
    for (i = 0; i <= N; i++) {
        double s = 0.0;
        for (k = 0; k < N; k+=4) {
            s += j / (i+1) * a[i][k] * i / (j+1) * v[k];
            s += j / (i+1) * a[i][k+1] * i / (j+1) * v[k+1];
            s += j / (i+1) * a[i][k+2] * i / (j+1) * v[k+2];
            s += j / (i+1) * a[i][k+3] * i / (j+1) * v[k+3];
        }
        c[i][j] = s;
    }
}
```

EXECUTION TIME:
113196



Optimization 5: loop unrolling - b



Basic loop (mmagic.c):

```
for (j = 0; j < N; j+=2) {
    double v[N][2];
    for (i = 0; i <= N; i++) {
        v[i][0] = b[i][j]; v[i][1] = b[i][j+1];
    }
    for (i = 0; i <= N; i+=2) {
        double s00 = 0.0, s01 = 0.0, s10 = 0.0, s11 = 0.0;
        for (k = 0; k < N; k++) {
            s00 += j / (i+1) * a[i][k] * i / (j+1) * v[k][0];
            s01 += (j+1) / (i+1) * a[i][k] * i / (j+2) * v[k][1];
            s10 += j / (i+2) * a[i+1][k] * (i+1) / (j+1) * v[k][0];
            s11 += (j+1) / (i+2) * a[i+1][k] * (i+1) / (j+2) * v[k][1];
        }
        c[i][j] = s00;
        c[i][j+1] = s01;
        c[i+1][j] = s10;
        c[i+1][j+1] = s11;
    }
}
```

EXECUTION TIME:
121725



Optimization 6: partition the problem



Basic loop (mmagic.c):

```
for (j0=0; j0 < N; j0 += 64) // Partition loop into 64 x 64 matrices
    for (i0 = 0; i0 < N; i0 += 64)
        for (k0 = 0; k0 < N; k0 += 64) {

            for (i = i0; i < i0+64; i++) // Multiply each sub-matrix (op # 2)

                for (j = k0; j < k0+64; j++)
                {
                    s = 0.0;
                    for (k = j0; k < j0+64; k++)
                        s += j / (i+1) * a[i][k] * i / (j+1) * b[k][j];
                    c[i][j] += s;
                }
        }
}
```

EXECUTION TIME
(Optimization 2):
258746

EXECUTION TIME
(Optimization 6):
126141

Optimization 7: op 2 + op 4 + op 6



Basic loop (mmagic.c):

```
for (j0=0; j0 < N; j0 += 64) // Partition loops into 64 x 64 matrices
  for (i0 = 0; i0 < N; i0 += 64)
    for (k0 = 0; k0 < N; k0 += 64) {
      for (j = k0; j < k0+64; j++) { // Multiply each matrix
        for (i = j0; i < j0+64; i++) v[i] = b[i][j]; // VECTOR
        for (i = i0; i < i0+64; i++) {
          s = 0.0; // use a register
          for (k = j0; k+4 <= j0+64; k+=4) { // loop unrolling - a
            s += j / (i+1) * a[i][k] * i / (j+1) * v[k];
            s += j / (i+1) * a[i][k+1] * i / (j+1) * v[k+1];
            s += j / (i+1) * a[i][k+2] * i / (j+1) * v[k+2];
            s += j / (i+1) * a[i][k+3] * i / (j+1) * v[k+3];
          }
          c[i][j] += s;
        }
      }
    }
}
```

EXECUTION TIME
(Optimizations 2+4=6):
99789

Many more possibilities



Generic

- Combine some of the different optimizations
- Try different problem partition sizes so that each sub-problem fits in fast memory
- Look for matrix multiplication optimization algorithms (literally dozens of them)
- Look at all possible compiler optimizations (all these programs compiled with -O3 flag in gcc)
- And if everything else fails, go to the assembly code

Specific to the assignment

- Look at characteristics of the problem that may make it easier:
 - ⌚ In matrix C, everything below the diagonal is 0 (half of the work is not necessary)
 - ⌚ the first row and the first column are 0 (less work to do)
 - ⌚ one of the testing cases is with input matrices that are all 0's, if you check for this case, you will reduce the average execution time by 25 %