Fast matrix multiplication; Cache usage

Assignment #2

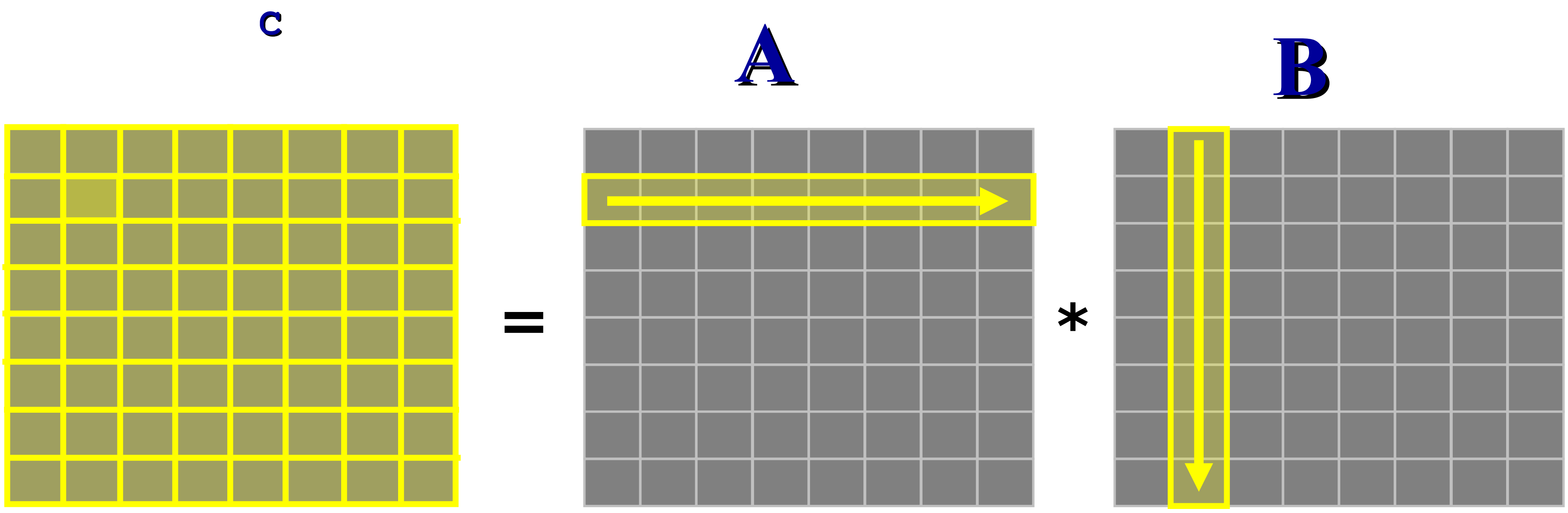
Multiplication of 2D Matrices

1. Simple algorithm

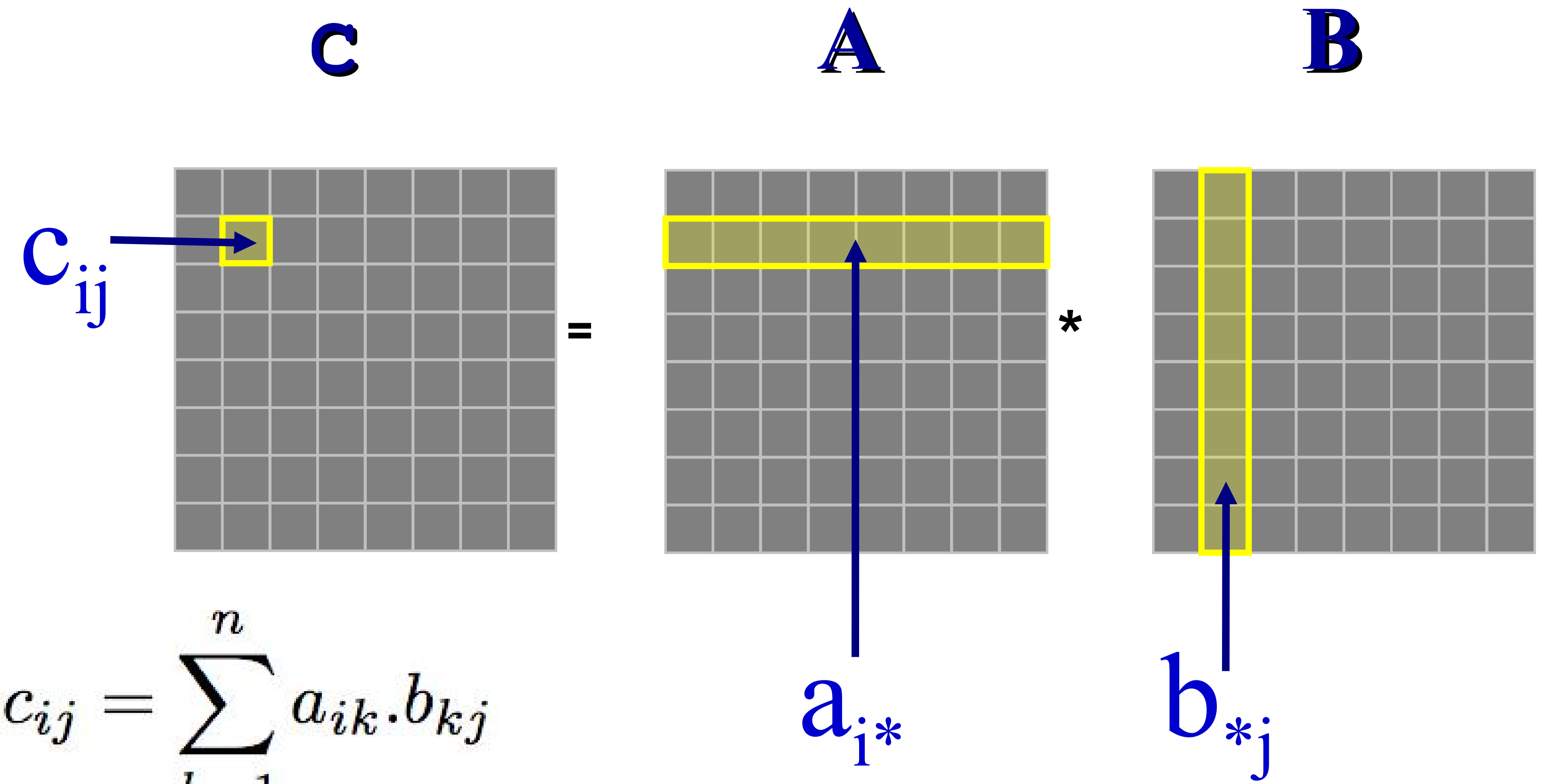
1. Time and Performance Measurement

2. Simple Code Improvements

Matrix Multiplication



Matrix Multiplication



$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

The simplest algorithm

Assumption: the matrices are stored as 2-D NxN arrays

```
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];
```

Advantage: code simplicity

Disadvantage: performance (?)

First Improvement

```
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];
```

c[i][j]:

- Requires address (pointer) computations
- is constant in the k-loop

First Performance Improvement

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

Performance Analysis

`clock_t clock()` - Returns the processor time used by the program since the beginning of execution, or -1 if unavailable.

`clock()/CLOCKS_PER_SEC` – the time in seconds (approx.)

```
#include <time.h>
clock_t t1,t2;
t1 = clock();
mult_ijk(a,b,c,n);
t2 = clock();
printf("Running time = %f seconds\n",
     (double)(t2 - t1)/CLOCKS_PER_SEC);
```

The running time

The simplest algorithm:

```
nova18% ./test
Enter matrix size and selected algorithm:
900 1
Running time = 24.870000 seconds
nova19%
```

After the first optimization:

```
nova14% ./test
Enter matrix size and selected algorithm:
900 2
Running time = 21.160000 seconds
nova15%
```

15% reduction

Profiling

- **Profiling allows you to learn where your program spent its time.**
- **This information can show you which pieces of your program are slower and might be candidates for rewriting to make your program execute faster.**
- **It can also tell you which functions are being called more often than you expected.**

gprof

- **gprof is a profiling tool ("display call graph profile data")**
- **Using gprof:**
  - compile and link with **-pg** flag (**gcc -pg**)
  - Run your program. After program completion a file name **gmon.out** is created in the current directory. **gmon.out** includes the data collected for profiling
  - Run **gprof** (e.g. "**gprof test**", where **test** is the executable filename)
- **For more details – man gprof.**

gprof outputs (gprof test gmon.out | less)

The total time spent by the function WITHOUT its descendents

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 59.12     81.33    81.33        1    81.33    81.33  mult_ijk_v1
 40.64    137.25    55.92        1    55.92    55.92  mult_ijk_v2
  0.12    137.42     0.17        2     0.09     0.09  fill_matrix_2D
  0.00    137.42     0.00        3     0.00     0.00  get_matrix_space_2D

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.
```

Cache Memory

Cache memory (CPU cache)

- A **temporary** storage area where **frequently accessed data** can be stored for **rapid access**.
- Once the data is stored in the cache, future use can be made by **accessing the cached copy** rather than re-fetching the original data, so that the **average access time is lower**.

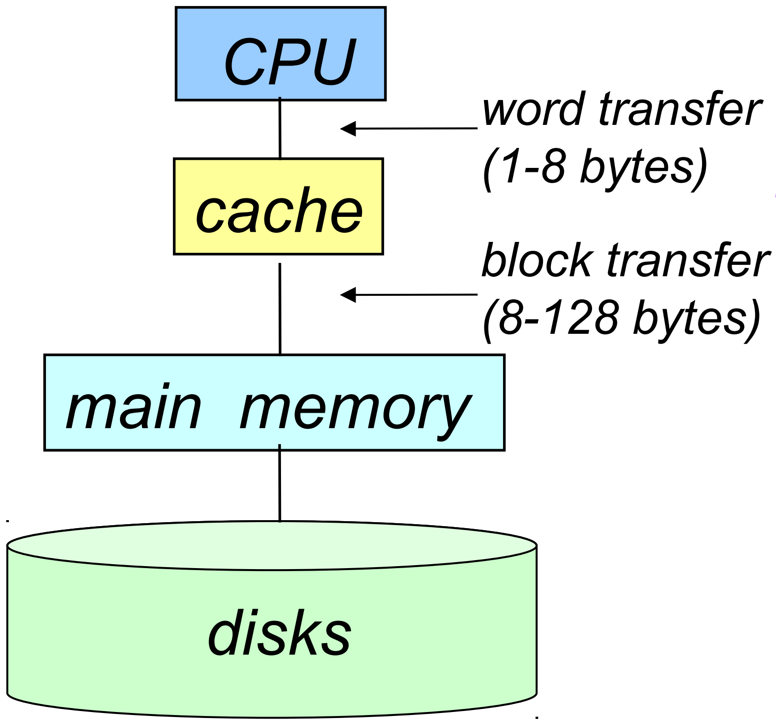
Memory Hierarchy

CPU Registers
500 Bytes
0.25 ns

Cache
16K-1M Bytes
1 ns

Main Memory
64M-2G Bytes
100ns

Disk
100 G Bytes
5 ms



- ↘ access time
- ↘ size of transfer unit
- ↗ cost per bit
- ↘ capacity
- ↗ frequency of access

- ➢ The memory cache is closer to the processor than the main memory.
- ➢ It is smaller, faster and more expensive than the main memory.
- ➢ Transfer between caches and main memory is performed in units called cache blocks/lines.

Types of Cache Misses

1. ***Compulsory misses:*** **caused by first access to blocks that have never been in cache (also known as *cold-start misses*)**

2. ***Capacity misses:*** **when cache cannot contain all the blocks needed during execution of a program. Occur because of blocks being replaced and later retrieved when accessed.**

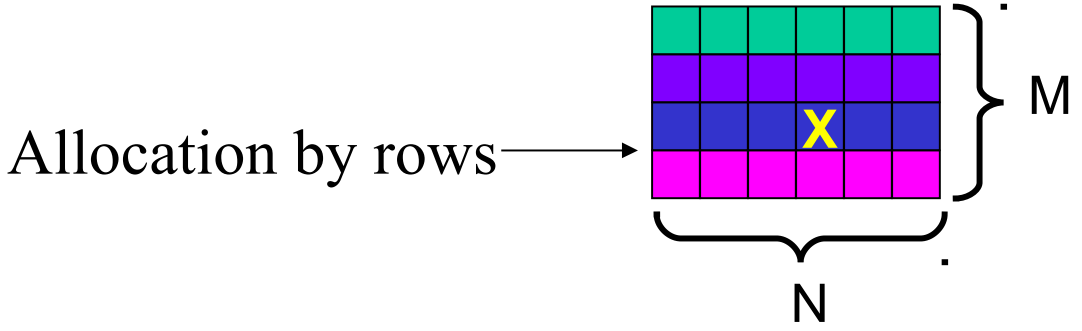
3. ***Conflict misses:*** **when multiple blocks compete for the same set.**

Main Cache Principles

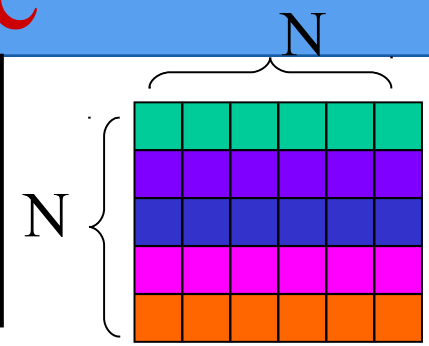
- **Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.**
  - LRU principle: Keep last recently used data
- **Spatial Locality (Locality in Space): If an item is referenced, close items (by address) tend to be referenced soon.**
  - Move blocks of contiguous words to the cache

Improving Spatial Locality:

Loop Reordering for Matrices Allocated by Row



Writing Cache Friendly Code



Assumptions: 1) block-size= β words (word = int)
2) N is divided by β
3) The cache cannot hold a complete row / column

```
int sumarrayrows(int a[N][N])
{
  int i, j, sum = 0;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
    return sum;
}
```

Accesses successive elements:
a[0][0],...,a[0][N-1],a[1][0],..

Miss rate = $1/\beta$

```
int sumarraycols(int a[N][N])
{
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      sum += a[i][j];
  return sum;
}
```

no spatial locality!

Accesses distant elements:
a[0][0],...,a[N-1][0],a[0][1],...

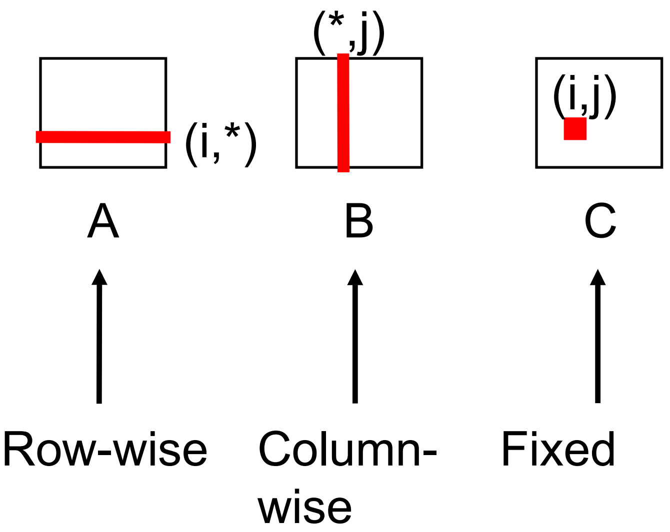
Miss rate = 1

Matrix Multiplication (ijk)

every element of A and B is accessed N times

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

Inner loop:



- Misses per Inner Loop Iteration (i=j=0):

A	B	C
$1/\beta$	1.0	1.0

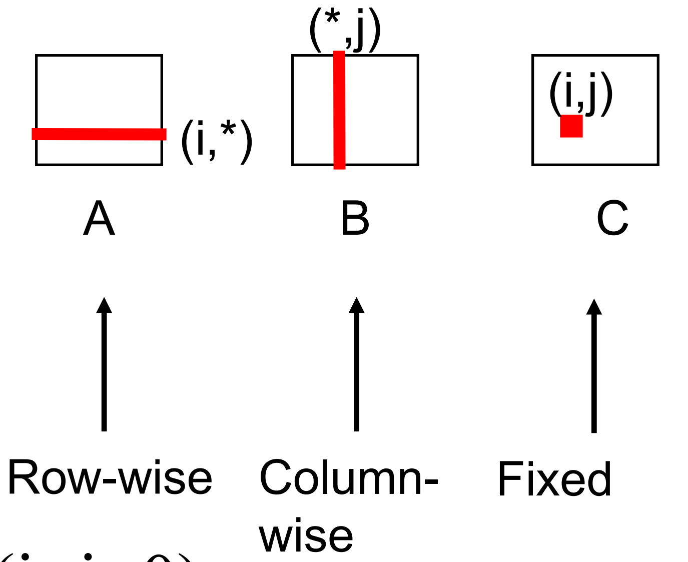
*what happens when i=0 and j=1?

Matrix Multiplication (jik)

every element of A and B is accessed N times

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

Inner loop:



- Misses per Inner Loop Iteration (i=j=0):

A	B	C
1/ β	1.0	1.0

*what happens when j=0 and i=1?

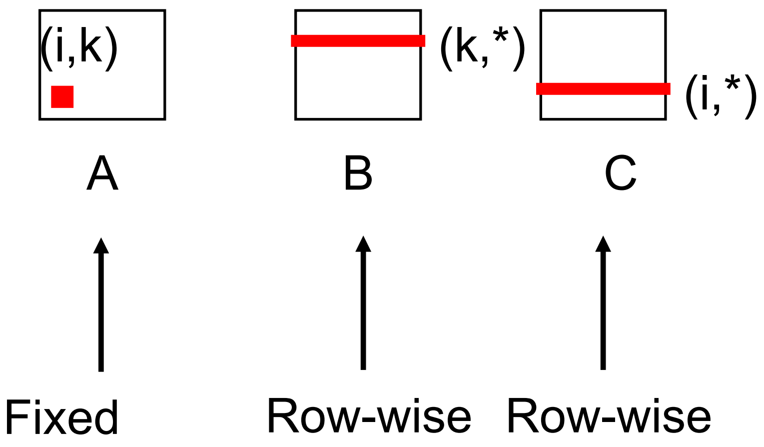
Matrix Multiplication (kij)

every element of B and C is accessed N times

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

  }
}
```

Inner loop:



- <u>Misses per Inner Loop Iteration (i=j=0):</u>

<u>A</u>	<u>B</u>	<u>C</u>
1.0	$1/\beta$	$1/\beta$

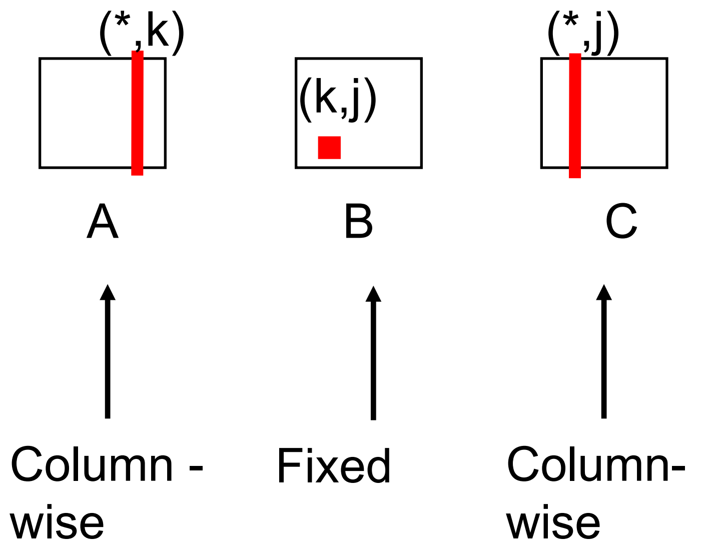
*what happens when k=0 and i=1?

Matrix Multiplication (jki)

every element of A and C is accessed N times

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

Inner loop:



- Misses per Inner Loop Iteration (i=j=0):

A	B	C
1.0	1.0	1.0

*what happens when j=0 and k=1?

Summary: misses ratios for the first iteration of the inner loop

ijk (& jik): (β+1)/(2β) >1/2 kij: 1/β jki: 1

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

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

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

Cache Misses Analysis

- Assumptions about the cache:

    β = Block size (in words)

    - The cache cannot hold an entire matrix
    - The replacement policy is LRU (Least Recently Used).
- Observation: for each loop ordering one of the matrices is scanned n times.

The inner indices point on the matrix scanned n times

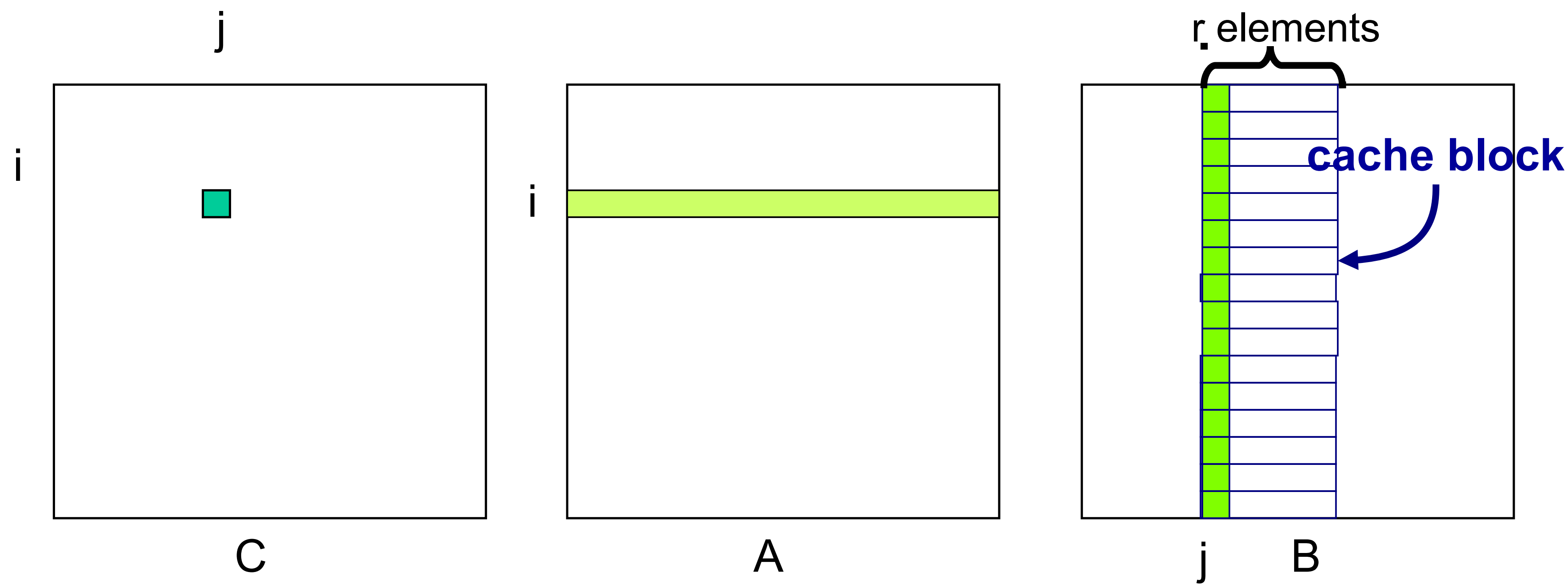
$$\Sigma_i \Sigma_j \Sigma_k C[i][j] += A[i][k] * B[k][j]$$

➔ Lower bound = $\underbrace{2n^2/\beta}_{\text{one read of 2 matrices (compulsory misses)}} + \underbrace{n^3/\beta}_{\text{n sequential reads of one matrix (compulsory + capacity misses)}} = \Theta(n^3/\beta)$

- For further details see the tutorial on the website.

Improving Temporal Locality: Blocked Matrix Multiplication

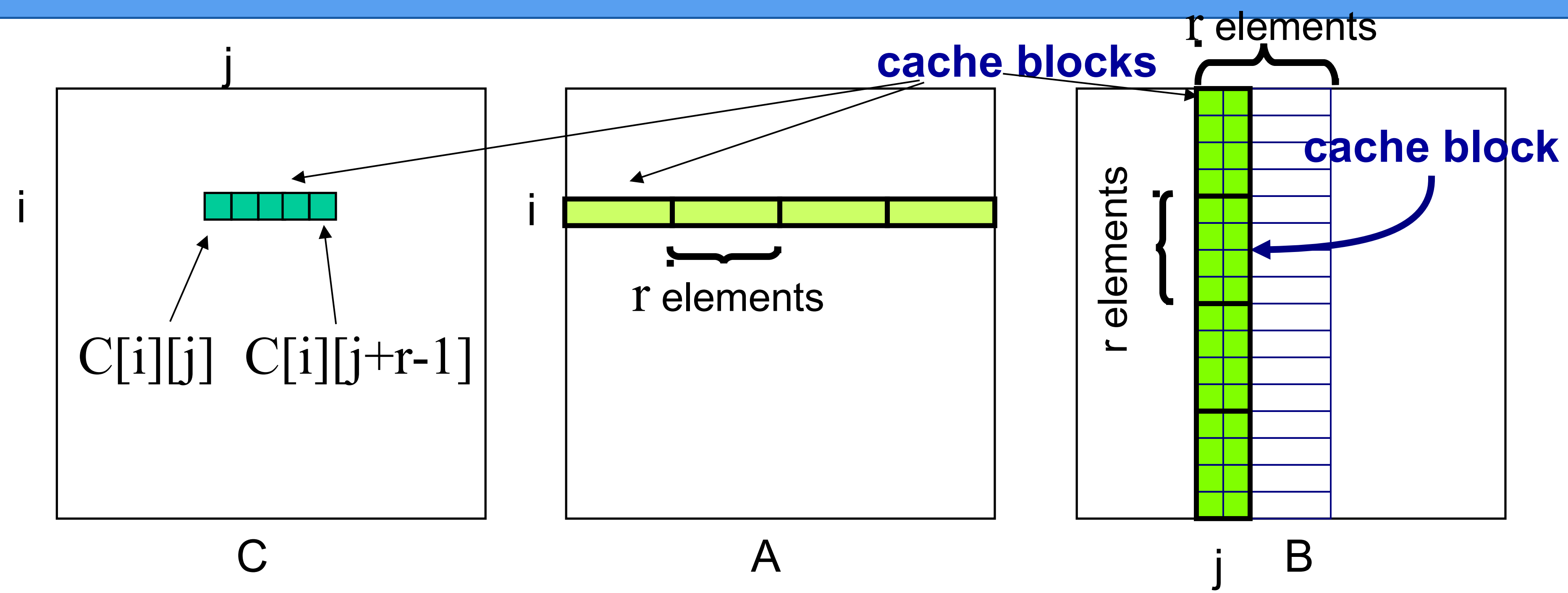
"Blocked" Matrix Multiplication



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

Key idea: reuse the other elements in each cache block as much as possible

"Blocked" Matrix Multiplication

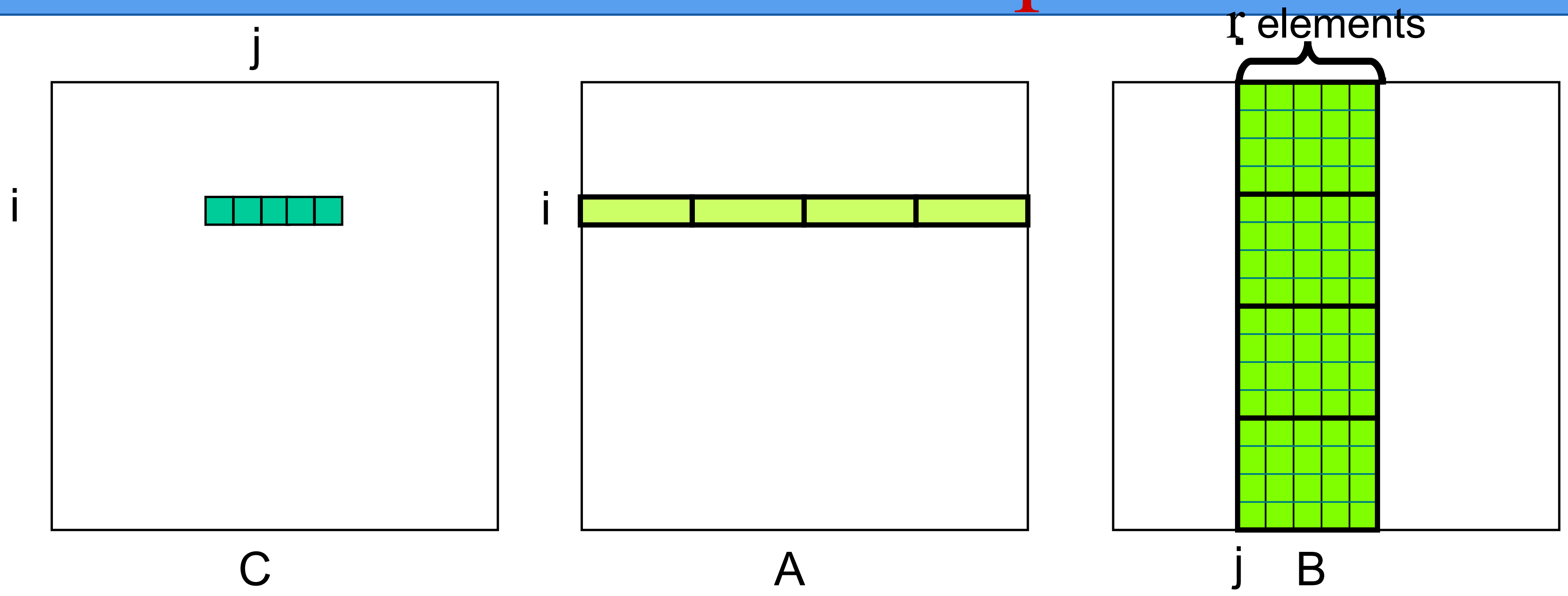


The blocks loaded for the computation of C[i][j] are appropriate for the computation of C[i,j+1]...C[i,j+ r-1]

- compute the first r terms of C[i][j],...,C[i][j+r -1]
- compute the next r terms of C[i][j],...,C[i][j+r -1]

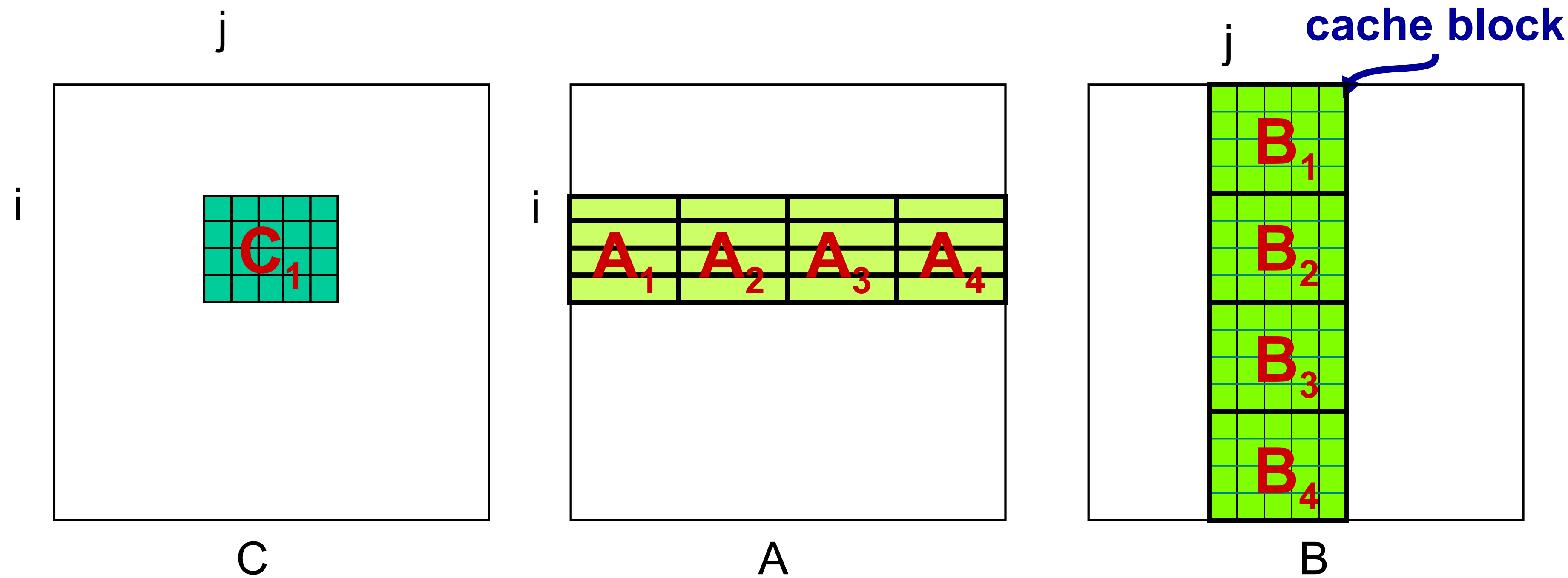
.....

"Blocked" Matrix Multiplication



Next improvement:
Reuse the loaded blocks of B for the computation of next (r -1) subrows.

“Blocked” Matrix Multiplication



Order of the operations:

Compute the first r terms of C1)=A_1*B_1)

Compute the next r terms of C1 (=A_2*B_2)

. . .

Compute the last r terms of C1 (=A_4*B_4)

“Blocked” Matrix Multiplication

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

N = 4 *r

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} * B_{k2}$$

- **Main Point:** each multiplication operates on small “block” matrices, whose size may be chosen so that they fit in the cache

Blocked Algorithm

- **The blocked version of the i-j-k algorithm is written simply as**

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

r x r matrix addition

r x r matrix multiplication

  - r = block (sub-matrix) size (Assume r divides N)
  - X[i][j] = a sub-matrix of X, defined by block row i and block column j

Maximum Block Size

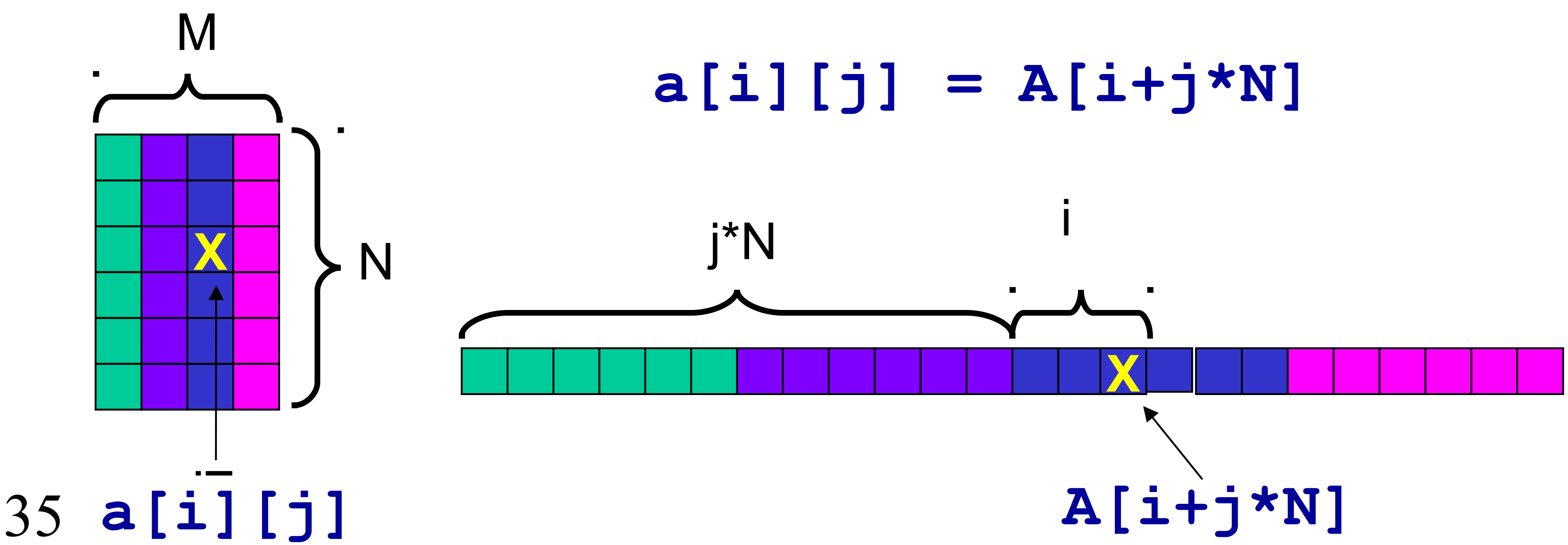
- **The blocking optimization works only if the blocks fit in cache.**
- **That is, 3 blocks of size r x r must fit in memory (for A, B, and C)**
- **M = size of cache (in elements/words)**
- **We must have: $3r^2 \approx M$, or $r \approx \sqrt{(M/3)}$**
- **Lower bound = $(r^2/\beta)\ (2(n/r)^3 + (n/r)^2) =$**

  $(1/\beta)(2n^3/r + n^2) = \Theta(n^3/(r\beta)) = \Theta(n^3/(\beta\sqrt{M}))$
- **Therefore, the ratio of cache misses ijk-blocked vs. ijk-unblocked: $1:\sqrt{M}$**

Home Exercise

Home exercise

- **Implement the described algorithms for matrix multiplication and measure the performance.**
- **Store the matrices as arrays, organized by columns!!!**



Question 2.1: mlpl

- **Implement all the 6 options of loop ordering (ijk, ikj, jik, jki, kij, kji).**
- **Run them for matrices of different sizes.**
- **Measure the performance with clock() and gprof.**
- **Plot the running times of all the options (ijk, jki, etc.) as the function of matrix size.**
- **Select the most efficient loop ordering.**

Question 2.2: block_mlpl

- **Implement the blocking algorithm.**
  - Use the most efficient loop ordering from 1.1.
- **Run it for matrices of different sizes.**
- **Measure the performance with `clock()`**
- **Plot the running times in CPU ticks as the function of matrix size.**

User Interface

- `Input`
  - Case 1: 0 or negative
  - Case 2: A positive integer number followed by values of two matrices (separated by spaces)

- `Output`
  - Case 1: Running times
  - Case 2: A matrix, which is the multiplication of the input matrices (C=A*B).

Files and locations

- **All of your files should be located under your home directory** **`/soft-proj09/assign2/`**
- **Strictly follow the provided prototypes and the file framework (explained in the assignment)**

The Makefile

```
mlpl:  allocate_free.c matrix_manipulate.c multiply.c mlpl.c
   gcc -Wall -pg -g -ansi -pedantic-errors allocate_free.c
   matrix_manipulate.c multiply.c mlpl.c -o mlpl

block_mlpl:  allocate_free.c matrix_manipulate.c multiply.c
   block_mlpl.c
   gcc -Wall -pg -g -ansi -pedantic-errors allocate_free.c
   matrix_manipulate.c multiply.c block_mlpl.c -o block_mlpl
```

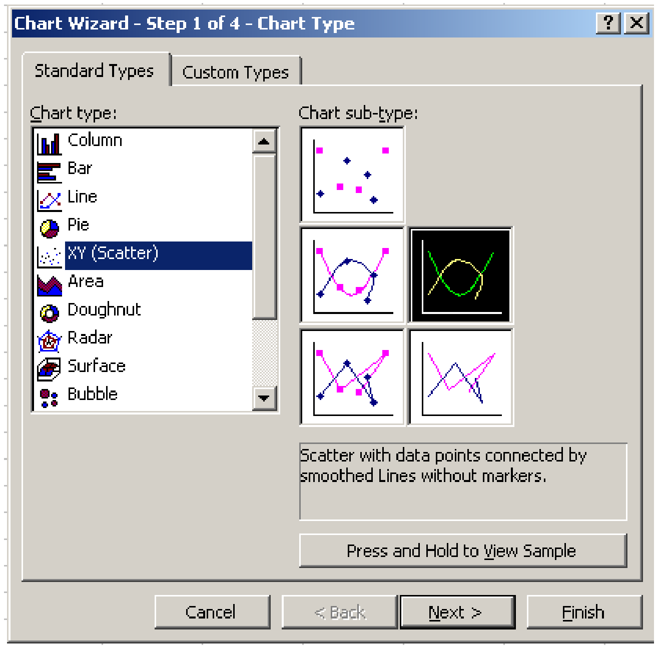
Commands:

`make mlpl` – will create the executable mlpl for 2.1

`make block_mlpl` - will create the executable block_mlpl for 2.2

Plotting the graphs

1. **Save the output to `*.csv` file.**
2. **Open it in Excel.**
3. **Use the Excel's "Chart Wizard" to plot the data as the XY Scatter.**
   - X-axis – the matrix sizes.
   - Y-axis – the running time in CPU ticks.



Final Notes

Arrays and Pointers:

The expressions below are equivalent:

```
int *a
int a[]
```

Good Luck in the Exercise!!!