# Optimized BLAS: an Example by Using Block Algorithms I

- Let's test the matrix multiplication
- A C program:

```
#define n 3000
double a[n][n], b[n][n], c[n][n];

int main()
{
    int i, j, k;
    for (i=0;i<n;i++)
```

# Optimized BLAS: an Example by Using Block Algorithms II

```
        for (j=0;j<n;j++) {
            a[i][j]=1; b[i][j]=1;
        }

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

# Optimized BLAS: an Example by Using Block Algorithms III

```
}
```

- Results:

```
cjlin@linux1:~$ gcc -O3 mat.c; time ./a.out
real 1m24.909s
user 1m24.534s
sys 0m0.193s
```

- We do the same task on Matlab
- To remove the effect of multi-threading, use

```
matlab -singleCompThread
```

# Optimized BLAS: an Example by Using Block Algorithms IV

- Results:

```
cjlin@linux1:~$ matlab -singleCompThread
>> n = 3000;
>> A = randn(n,n); B = randn(n,n);
>> tic; C = A*B; toc
Elapsed time is 1.708523 seconds.
```

- An issue about timing is elapsed time versus CPU time

# Optimized BLAS: an Example by Using Block Algorithms V

```
>> A = randn(n,n); B = randn(n,n);
>> t = cputime; C = A*B; t = cputime -t

t =

    1.3000
```

They are similar if no other jobs are running on this machine.

- Results of using multi-threading (the default of MATLAB)

# Optimized BLAS: an Example by Using Block Algorithms VI

```
cjlin@linux1:~$ matlab
>> n = 3000;
>> A = randn(n,n); B = randn(n,n);
>> tic; C = A*B; toc
Elapsed time is 0.426942 seconds.
>> A = randn(n,n); B = randn(n,n);
>> t = cputime; C = A*B; t = cputime -t

t =
```

# Optimized BLAS: an Example by Using Block Algorithms VII

```
        5.1200
```

- We see that under the same setting of using a single thread, Matlab is much faster than a code written by ourselves.
- Why ?
- Optimized BLAS: an implementation that takes the advantage of memory hierarchies
- Data locality is exploited
- Use the highest level of memory as possible

# Optimized BLAS: an Example by Using Block Algorithms VIII

- Block algorithms: a way to transfer sub-matrices between different levels of storage

  They localize operations to achieve good performance

# Memory Hierarchy I

CPU

↓

Registers

↓

Cache

↓

Main Memory

↓

Secondary storage (Disk)

- $\uparrow$: increasing in speed
- $\downarrow$: increasing in capacity

# Memory Management I

- We assume that the computer has only two layers of memory
  - main memory
  - secondary memory
- Page fault: an operand is not available in main memory and must be transported from secondary memory
- When moving things between layers, due to initialization cost, we move a continuous segment of data (called a page) instead of a single value

# Memory Management II

- Usually if a page is moved to the main memory, it overwrites page least recently used
- An example: $C = AB + C$, $n = 1,024$
- Assumption: a page 65,536 doubles = 64 columns
- 16 pages for each matrix

  48 pages for three matrices
- Assumption: available memory 16 pages, matrices access: column oriented

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

# Memory Management III

column oriented: 1 3 2 4

row oriented: 1 2 3 4

- access each row of $A$: 16 page faults, $1024/64 = 16$
- Approach 1:

```
for i =1:n
  for j=1:n
    for k=1:n
      c(i,j) = a(i,k)*b(k,j)+c(i,j);
    end
  end
end
```

# Memory Management IV

We use a matlab-like syntax here

- At each (i,j): each row a(i, 1:n) causes 16 page faults

  Total: $1024^2 \times 16$ page faults
- at least 16 million page faults
- Approach 2:

# Memory Management V

```
for j =1:n
  for k=1:n
    for i=1:n
      c(i,j) = a(i,k)*b(k,j)+c(i,j);
    end
  end
end
```

- For each $j$, access all columns of $A$
  $A$ needs 16 pages, but $B$ and $C$ take spaces as well
  So $A$ must be read for every $j$

# Memory Management VI

- For each $j$, 16 page faults for $A$

  $1024 \times 16$ page faults

  $C, B$ : 16 page faults
- What if we implement this approach in C?
- Code:

```
#define n 3000
double a[n][n], b[n][n], c[n][n];

int main()
{
    int i, j, k;
```

# Memory Management VII

```
for (i=0;i<n;i++)
   for (j=0;j<n;j++) {
      a[i][j]=1; b[i][j]=1;
      c[i][j]=0;
   }

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

# Memory Management VIII

```
}
```

- Results:

```
cjlin@linux1:~$ gcc -O3 mat1.c; time ./a.out
real 4m20.247s
user 4m19.761s
sys 0m0.154s
```

- Why is it even slower?
- C is row-oriented instead of column-oriented
- Thus we had implemented Approach 2 first and then Approach 1