Exercise 2

Introduction to High-Performance Computing
WS 2019

Julian Miller
Tim Cramer
Sandra Wienke
contact@hpc.rwth-aachen.de

Recap: O(N^3)/O(N^2) – Example

- **Examples for O(N^3)/O(N^2) algorithms are complex**

- **To simplify, we use an O(N^2)/O(N) algorithm to illustrate possible optimization:**

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

b is loaded from memory N times

a is loaded once per inner loop

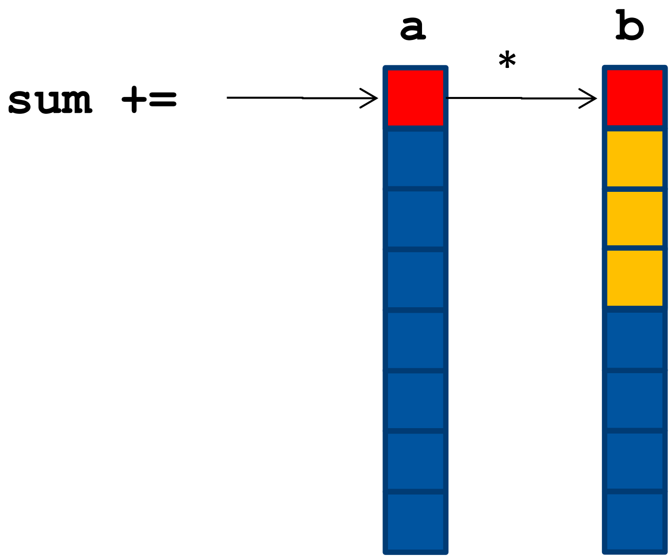
Thus memory traffic amounts to N(N+1) words.

- **Loop unroll & jam will reduce this to N(N/m + 1)**

  → with m-way unrolling

Recap: O(N^3)/O(N^2) – Example

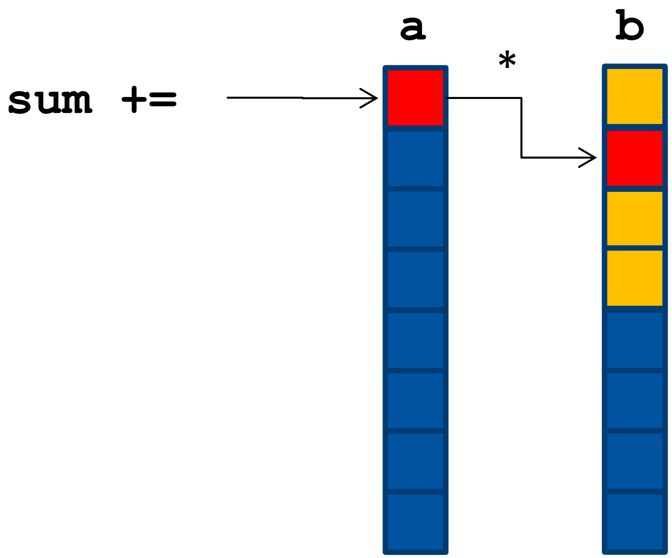
```
for(i = 0; i < N; ++i )
{
  for(j=0; j < N; ++j)
  {
    sum += a[i] * b[j];
  }
}
```



Julian Miller | IT Center der RWTH Aachen University

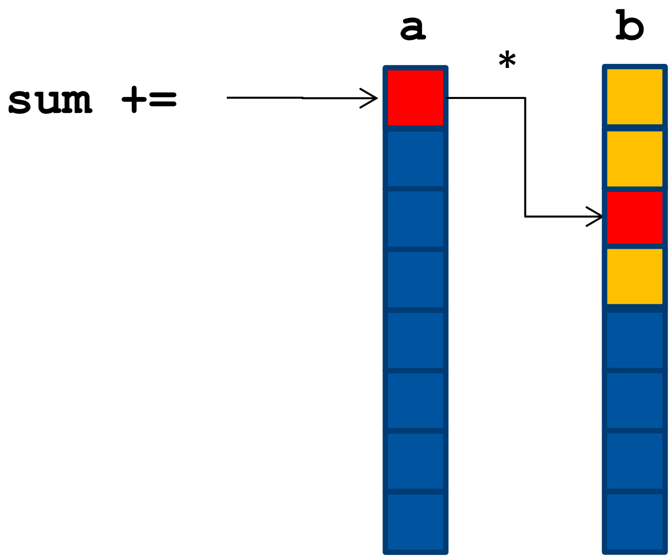
Recap: O(N^3)/O(N^2) – Example

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



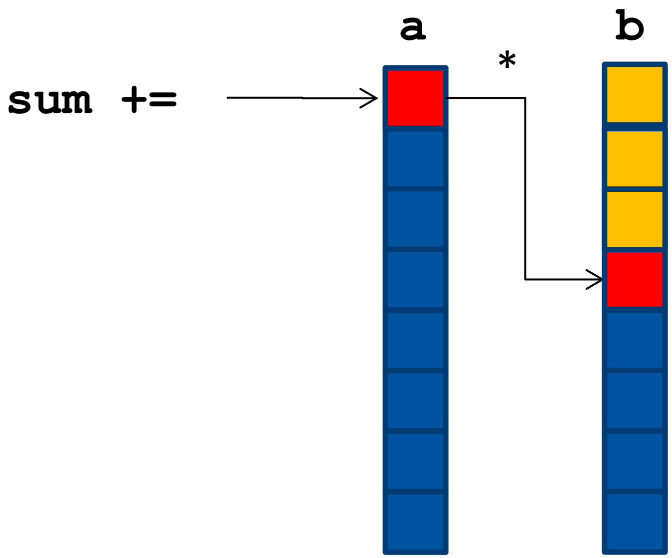
Recap: O(N^3)/O(N^2) – Example

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



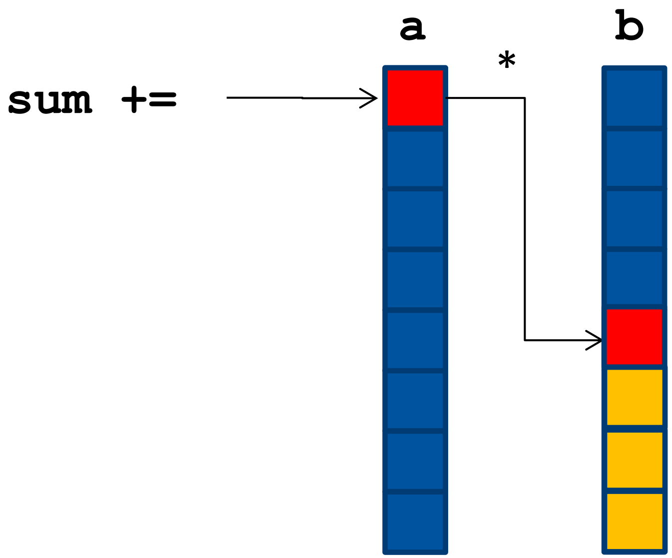
Recap: O(N^3)/O(N^2) – Example

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



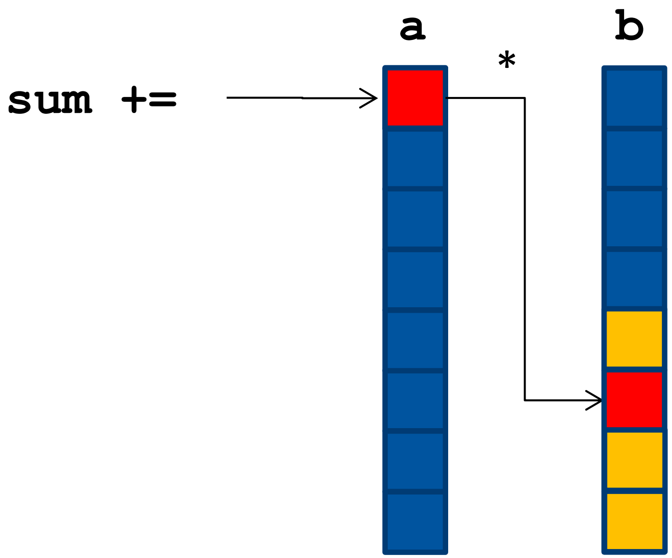
Recap: O(N^3)/O(N^2) – Example

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

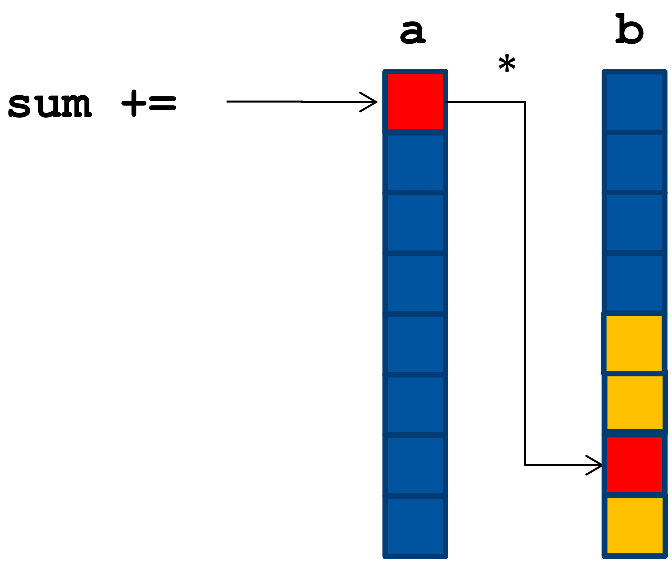


Recap: O(N^3)/O(N^2) – Example

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

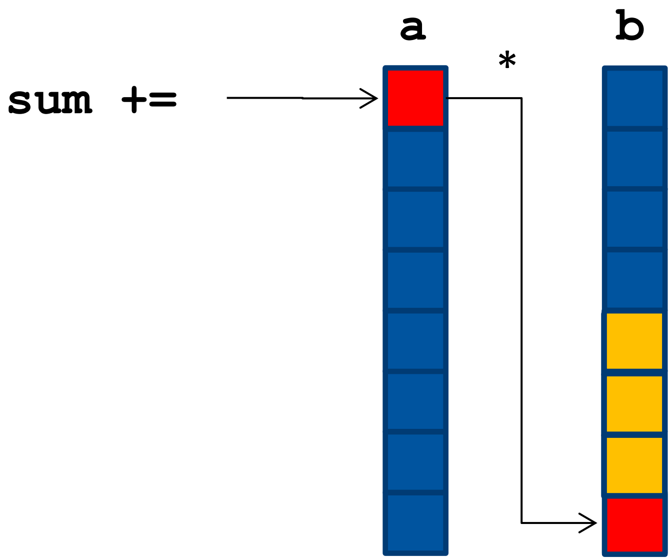


Recap: O(N^3)/O(N^2) – Example




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

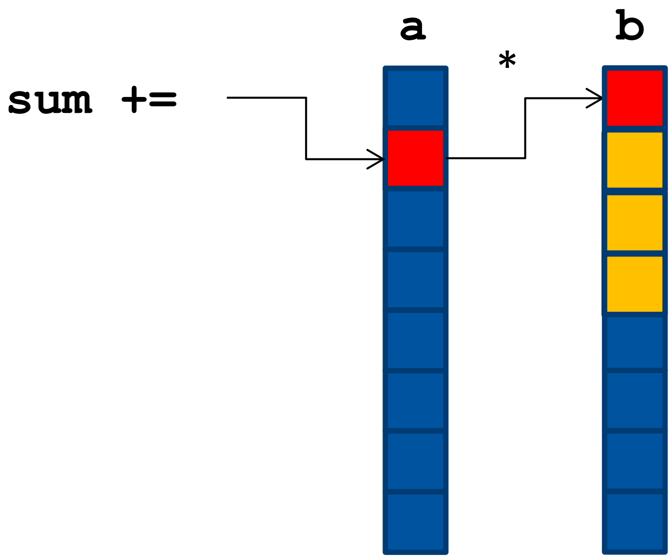
Recap: O(N^3)/O(N^2) – Example




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

Recap: O(N^3)/O(N^2) – Example

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

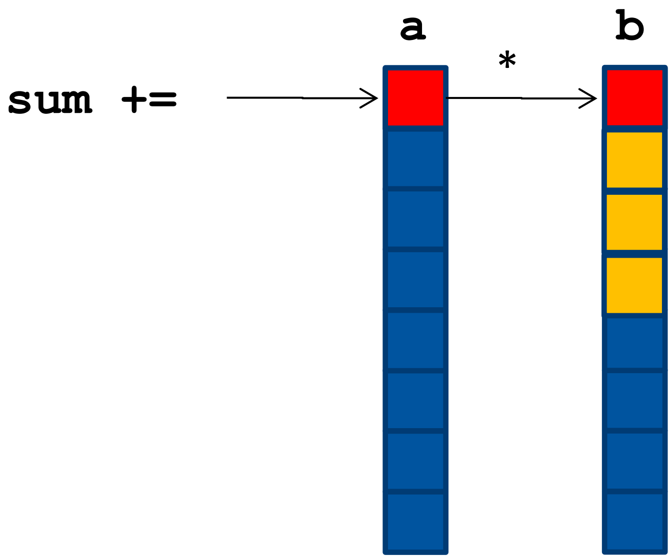


Recap: O(N^3)/O(N^2) – Example

- **Blocking the inner loop:** →
  - → With blocking factor B
- **Introduces two effects**
  - → Array b is only loaded once from memory now
    - → as long as factor B is small enough that parts loaded from b fit into the cache and stay there as long as needed
  - → Array a is loaded from memory N/B times instead of once
- **Effective memory traffic: N(N/B+1)**
- **Blocking is the method of choice here, as the blocking factor can be increased to higher values than the unrolling factor**

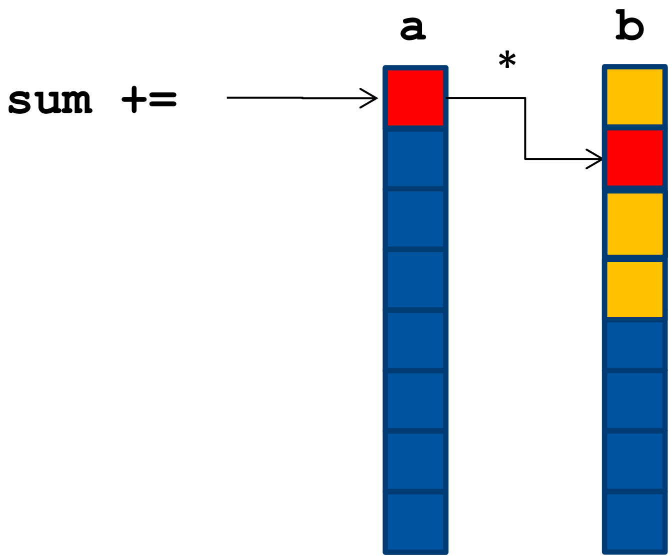
```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Recap: O(N^3)/O(N^2) – Example



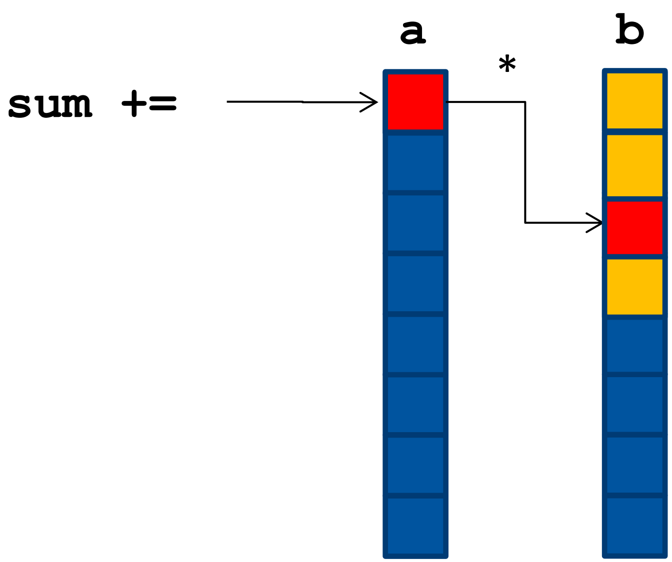

```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Recap: O(N^3)/O(N^2) – Example



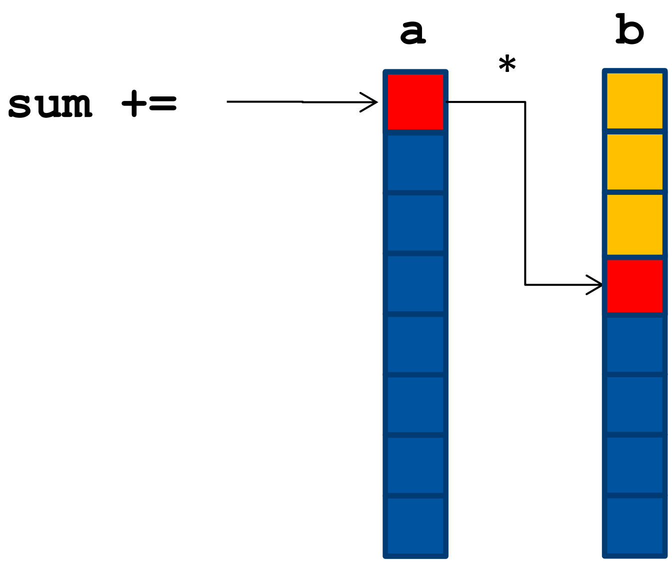

```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Recap: O(N^3)/O(N^2) – Example



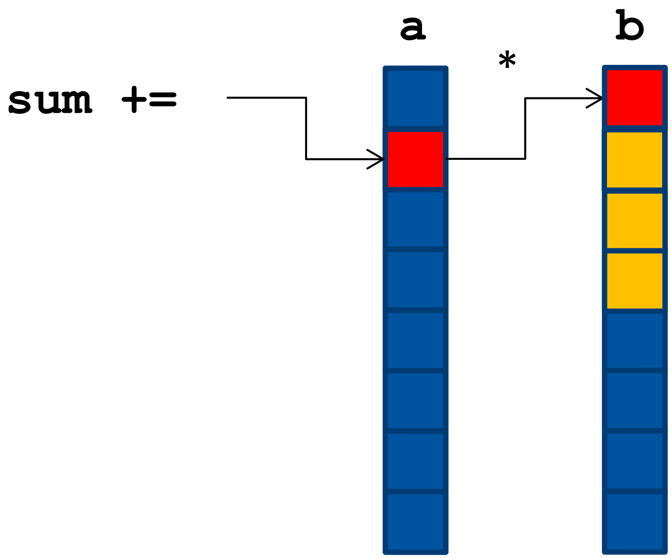

```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Recap: O(N^3)/O(N^2) – Example



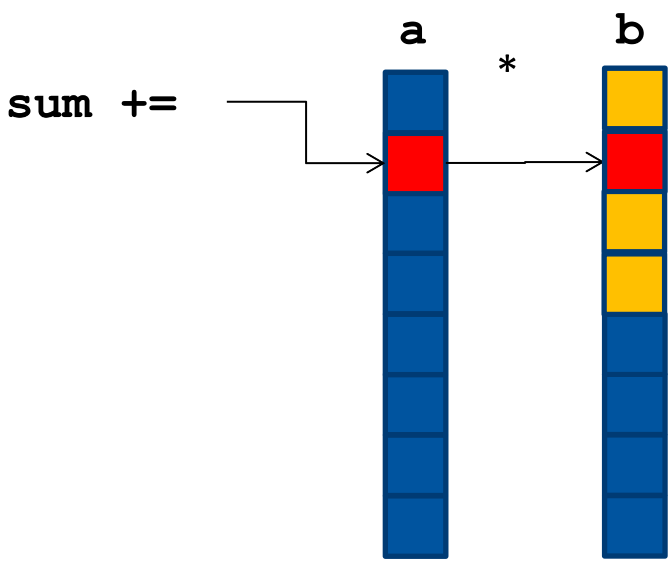

```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Recap: O(N^3)/O(N^2) – Example




```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Recap: O(N^3)/O(N^2) – Example




```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Exercise Tasks

1. **Norm calculation of a matrix**
2. Performance analysis tools
3. Balance Metric

1. Norm calculation of a matrix

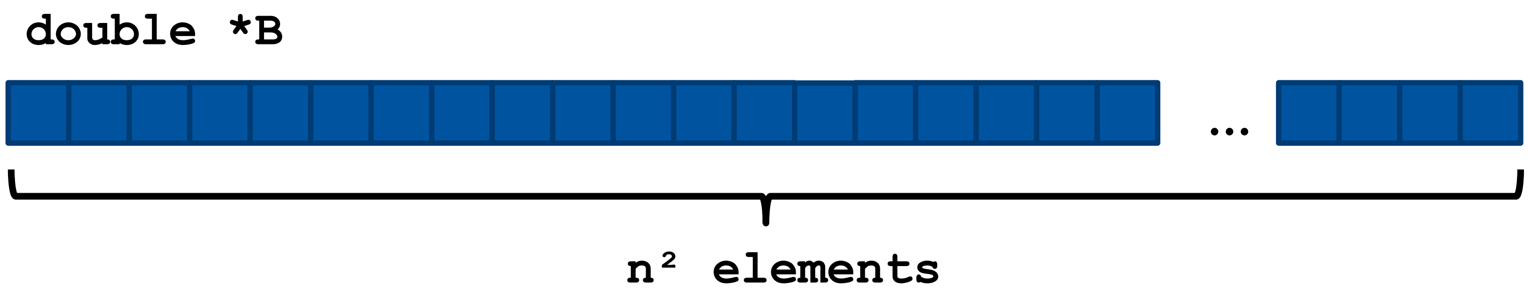
Problem 1. Norm calculation of a matrix
Let $A = (a_{ij})_{i,j=1,\dots,n} \in \mathbb{R}^{n \times n}$ be a real matrix. The norms $\|\cdot\|_1$ and $\|\cdot\|_\infty$ are defined by:

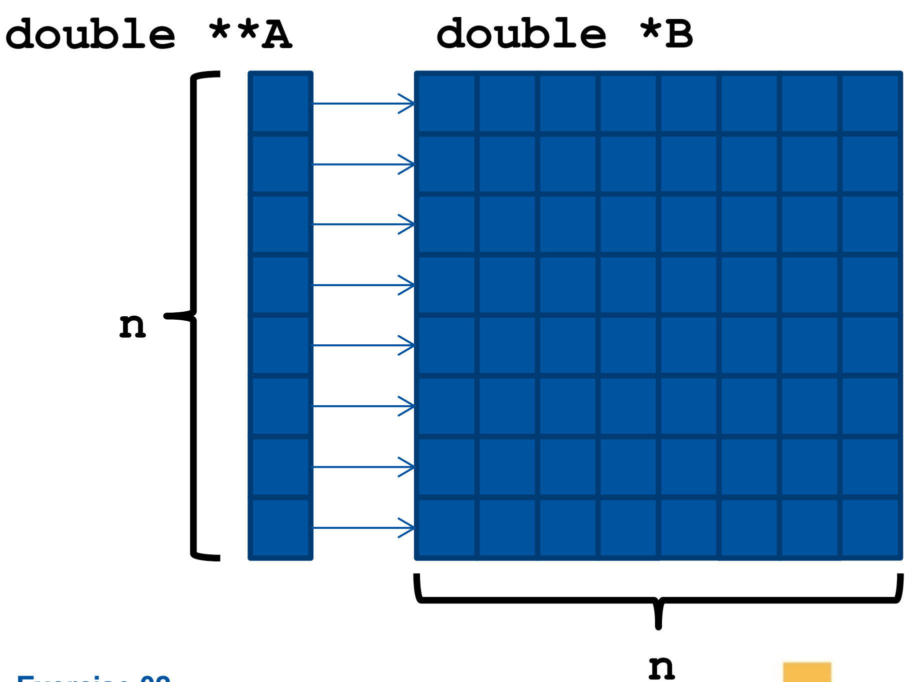
$$\begin{aligned}
\|A\|_1 &:= \max_{j=1,\dots,n} \sum_{i=1}^{n} |a_{ij}| \qquad (\text{“maximum column sum”}) \\
\|A\|_\infty &:= \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{ij}| \qquad (\text{“maximum row sum”}).
\end{aligned}$$

This exercise may be done with pen and paper or using the C++ code template in `exercise2.tar.gz`

1. Norm calculation of a matrix
Preparation: Allocate memory for the matrix

- **Idea: Store matrix as a 1D-block in memory**



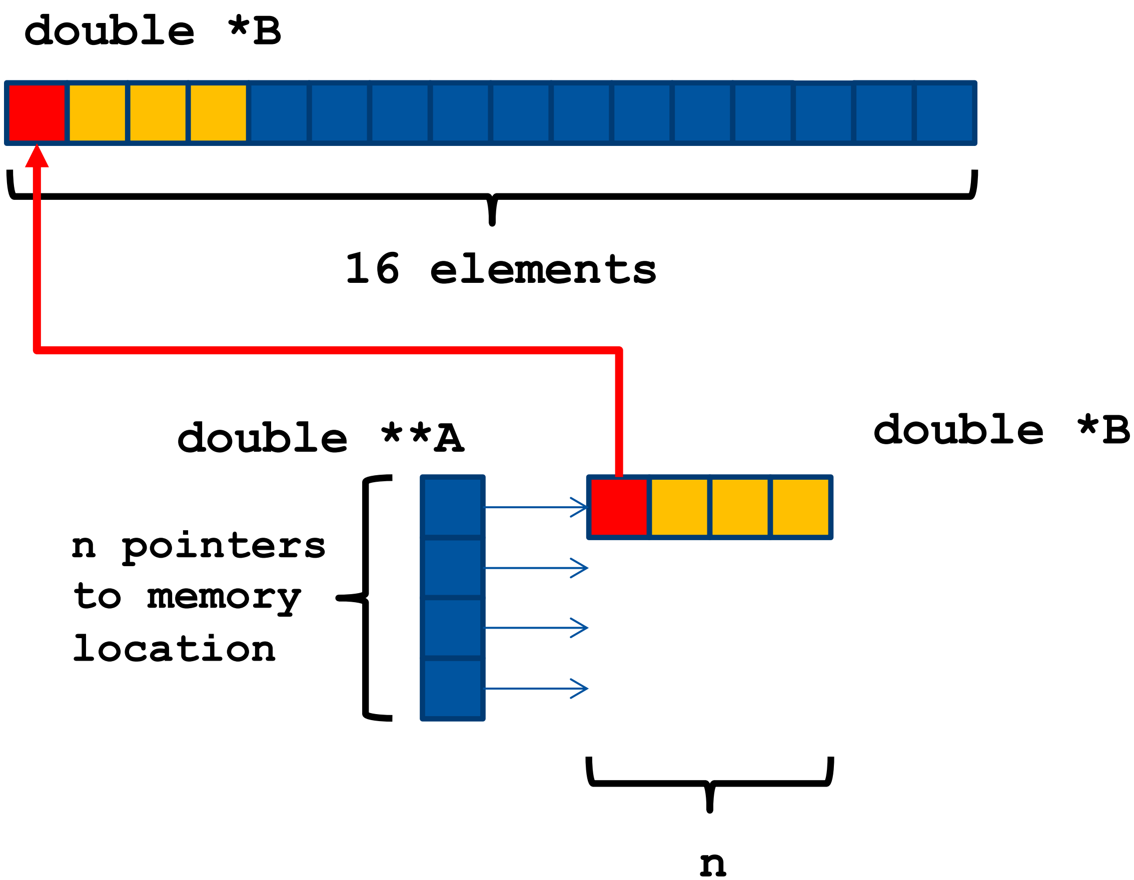


Same starting address: B=*A

Access elements by
A[x][y] or B[x*n+y]

1. Norm calculation of a matrix
Preparation: Allocate memory for the matrix

- **Example: n = 4 ➔ 4x4 matrix**

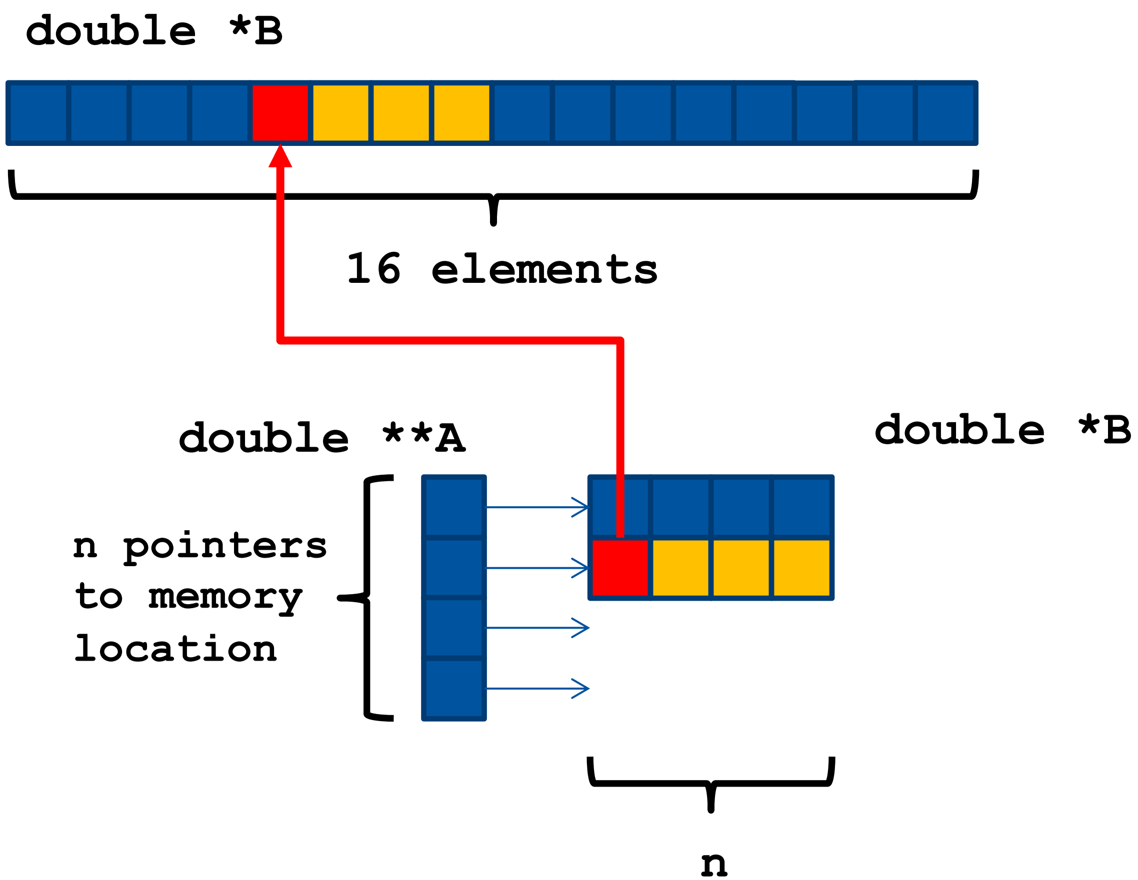


Same starting address: B=*A

Access elements by
A[x][y] or B[x*n+y]

1. Norm calculation of a matrix
Preparation: Allocate memory for the matrix

- **Example: n = 4 ➔ 4x4 matrix**

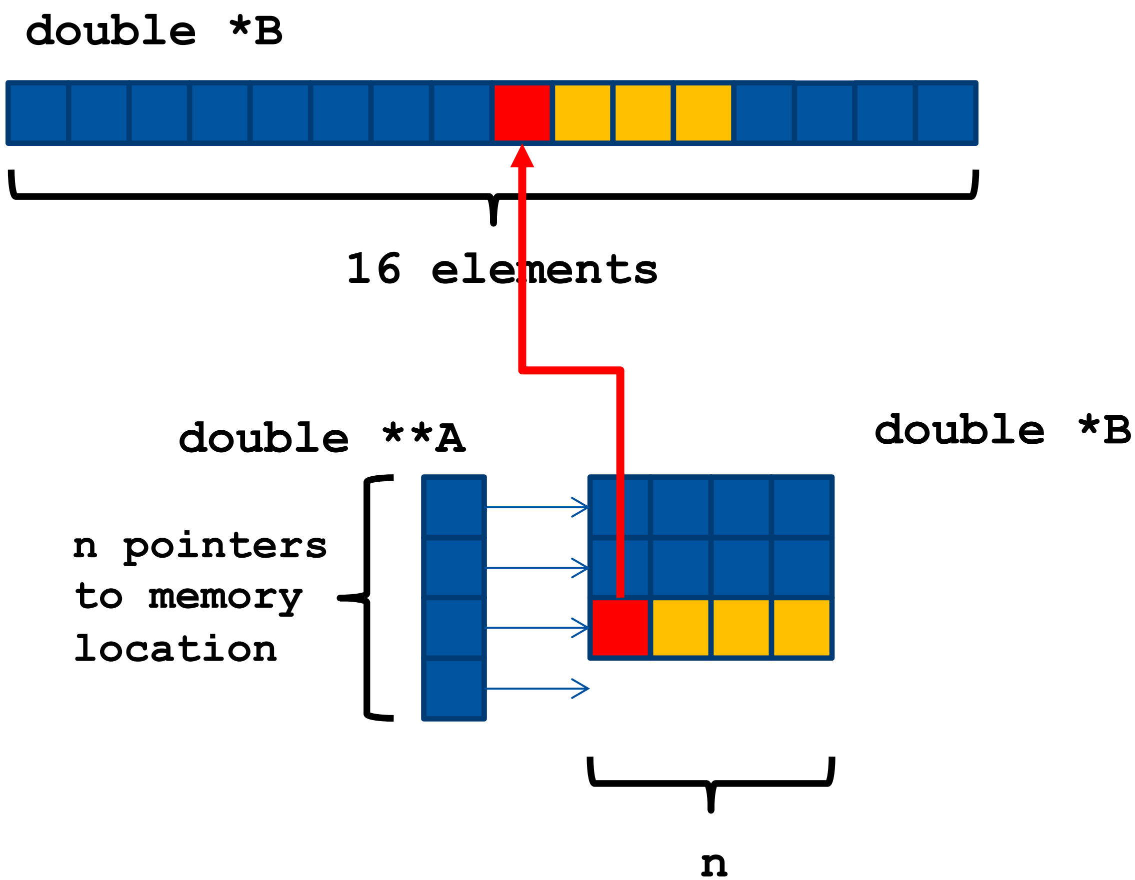


Same starting address: B=*A

Access elements by
A[x][y] or B[x*n+y]

1. Norm calculation of a matrix
Preparation: Allocate memory for the matrix

- **Example: n = 4 ➔ 4x4 matrix**

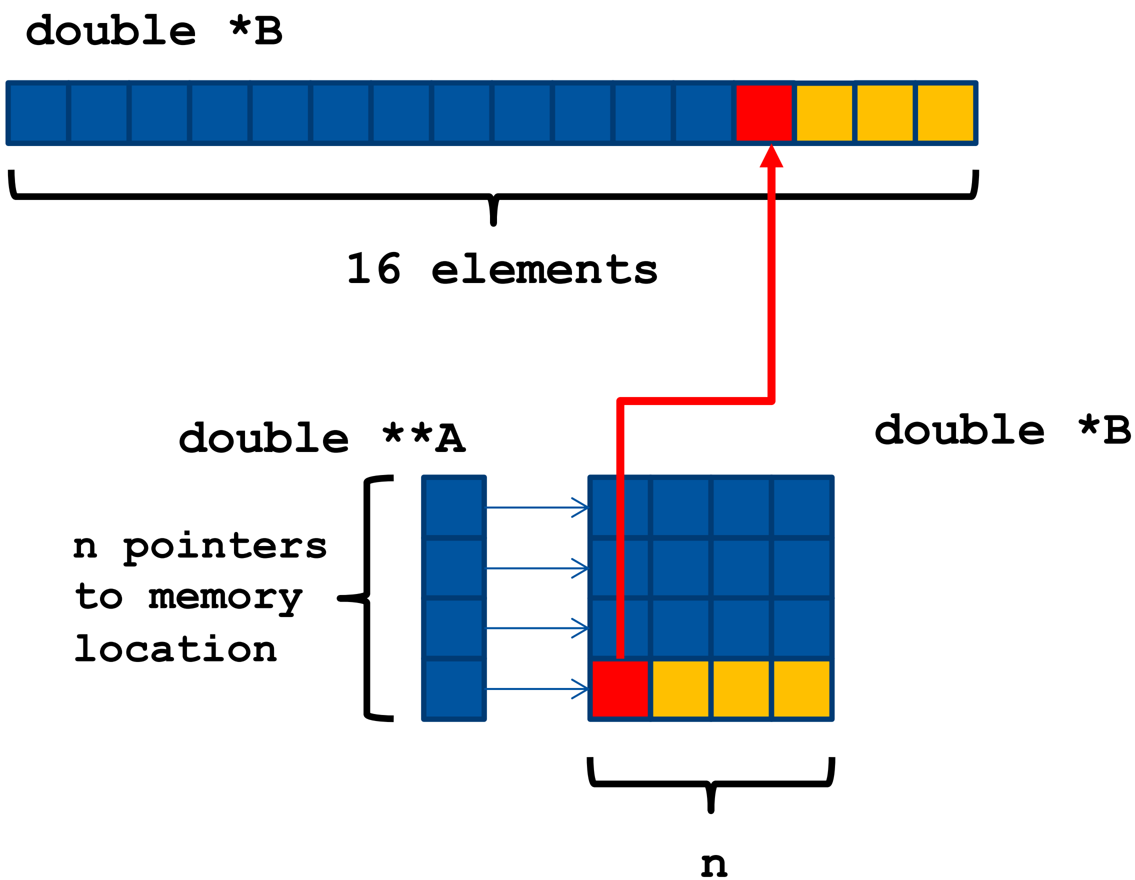


Same starting address: B=*A

Access elements by
A[x][y] or B[x*n+y]

1. Norm calculation of a matrix
Preparation: Allocate memory for the matrix

- **Example: n = 4 ➔ 4x4 matrix**



Same starting address: B=*A

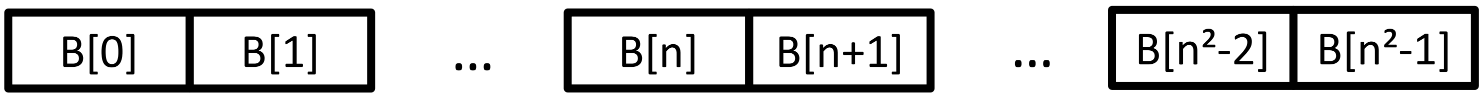
Access elements by
A[x][y] or B[x*n+y]

Livedemo

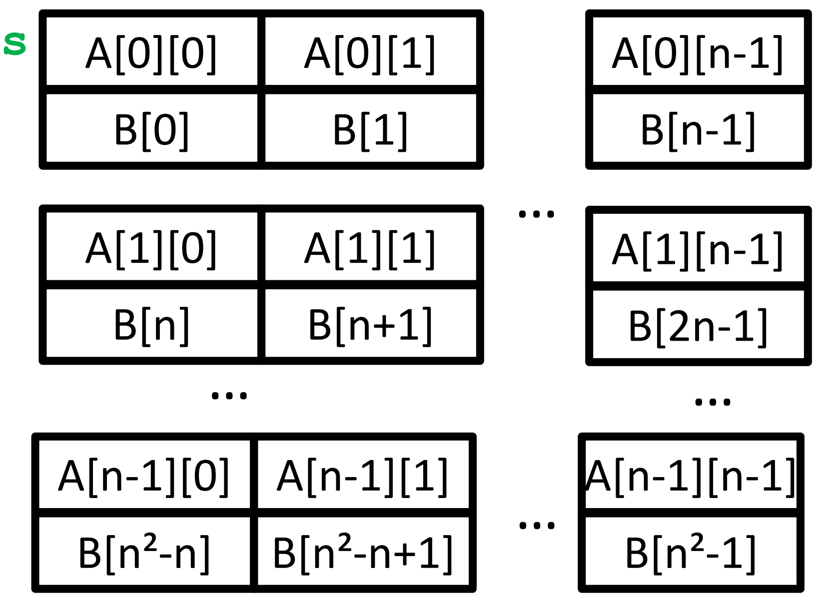
1. Norm calculation of a matrix

- **Preparation: Allocate memory for the matrix**

```
/* Allocate the complete matrix in one block */
    double *B = new double[(long)n*n];
```




```
/* Allocate memory to store the address
   of the first element of each row */
    double **A = new double*[n];

/* Place one pointer on each row
   of the matrix in the array A */
    for (int i=0; i<n; ++i)
        A[i] = &(B[(long)i*n]);
```



1. Norm calculation of a matrix
a) norm_max()

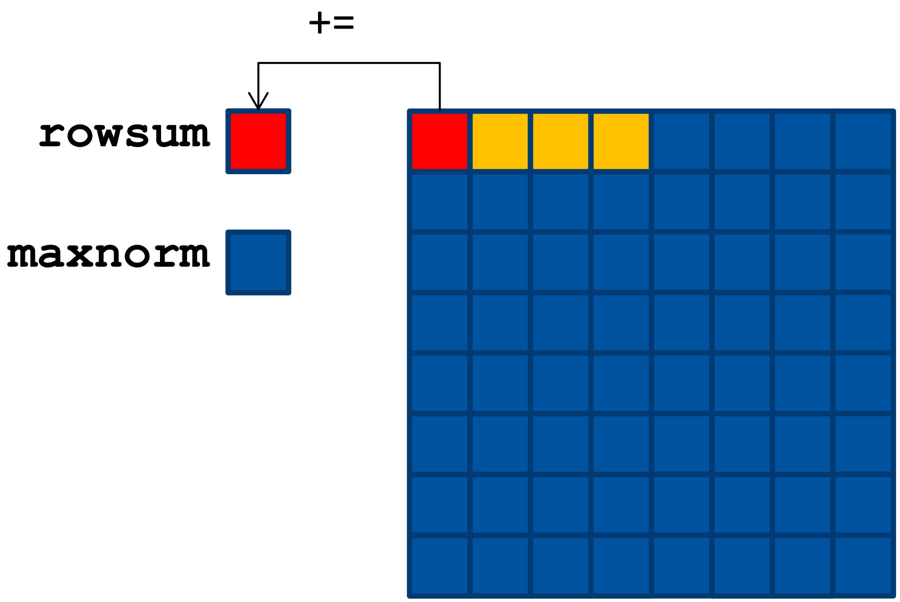
a) Design an algorithm to compute the norm $\|A\|_\infty$ by passing through the memory consecutively.

Livedemo

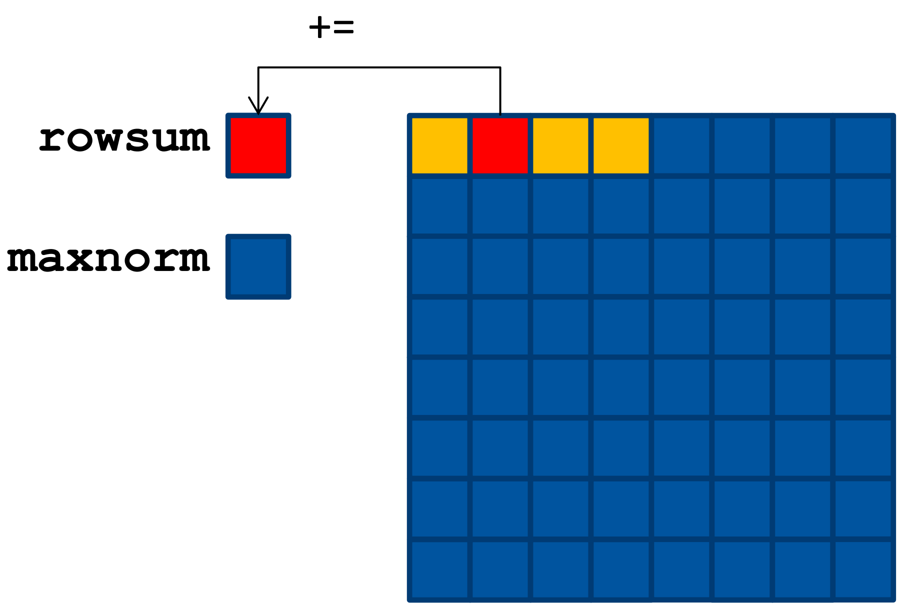
1. Norm calculation of a matrix
a) norm_max()

```
double norm_max(double** const A, const int n) {
    double rowsum = 0., max_norm = -1.;
/* TODO Implement the max norm using 2 for loops
        and the function double abs(double x) */
    for (int i = 0; i < n; ++i) {
        rowsum = 0.;
        for (int j = 0; j < n; ++j)
            rowsum += abs(A[i][j]);
        if (rowsum > max_norm)
            max_norm = rowsum;
    }
    return max_norm;
}
```

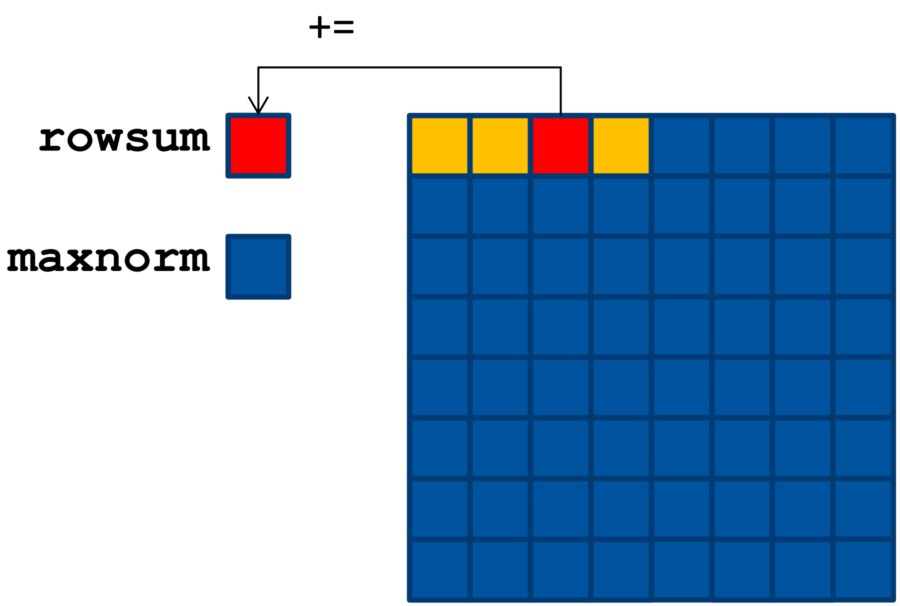
1. Norm calculation of a matrix
a) norm_max()



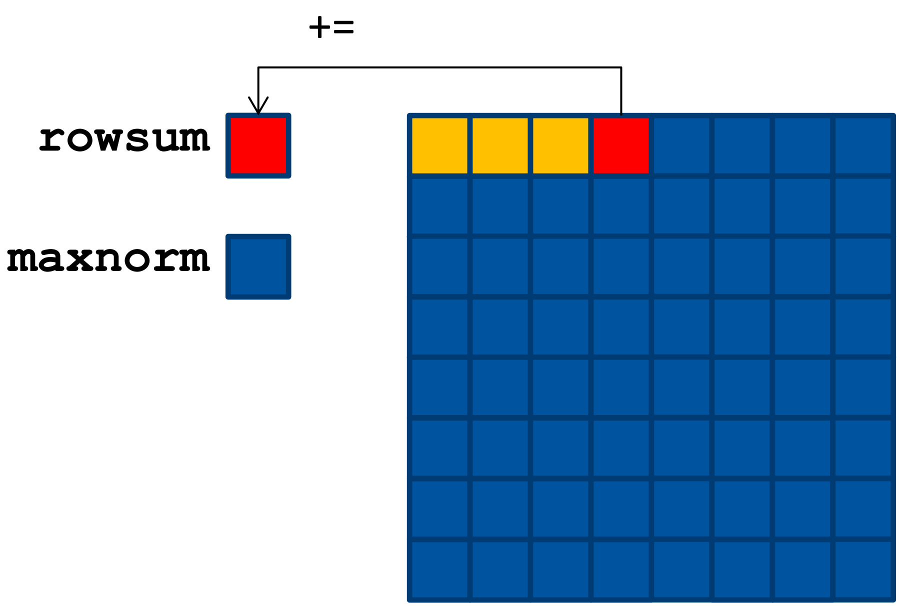
1. Norm calculation of a matrix
a) norm_max()



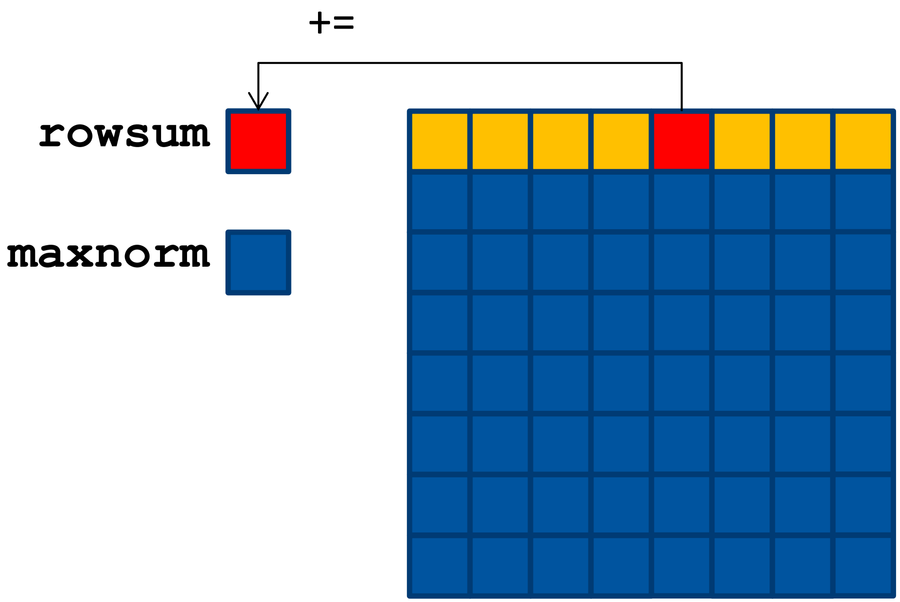
1. Norm calculation of a matrix
a) norm_max()



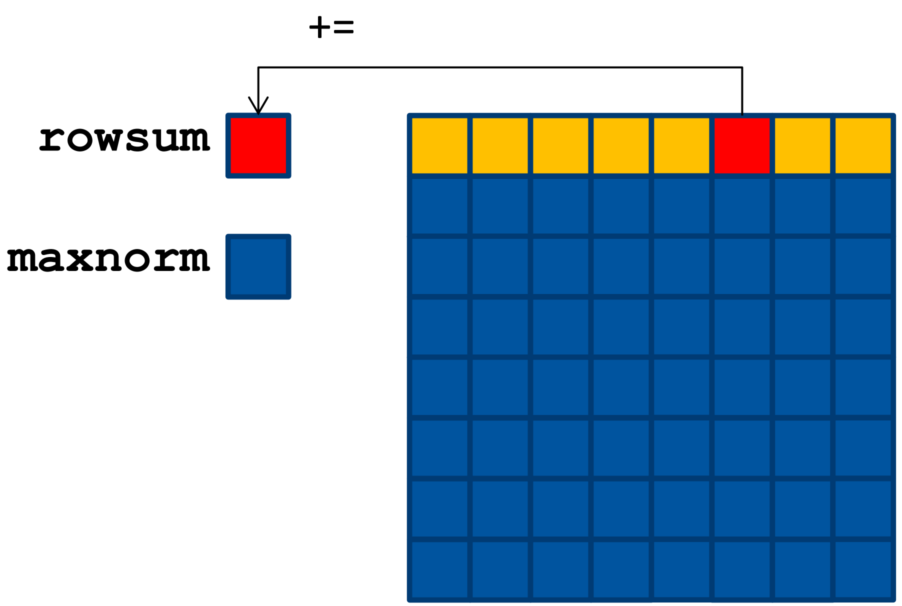
1. Norm calculation of a matrix
a) norm_max()



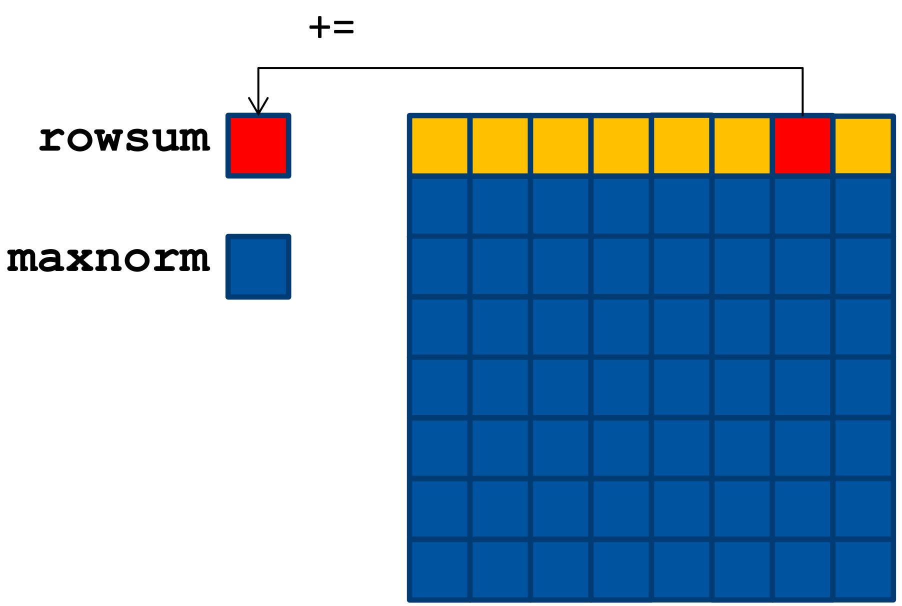
1. Norm calculation of a matrix
a) norm_max()



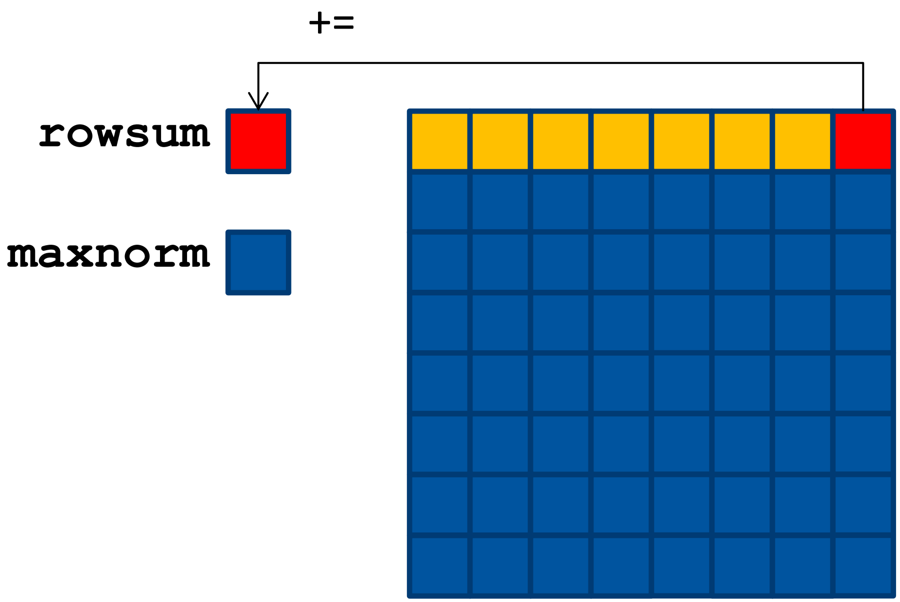
1. Norm calculation of a matrix
a) norm_max()



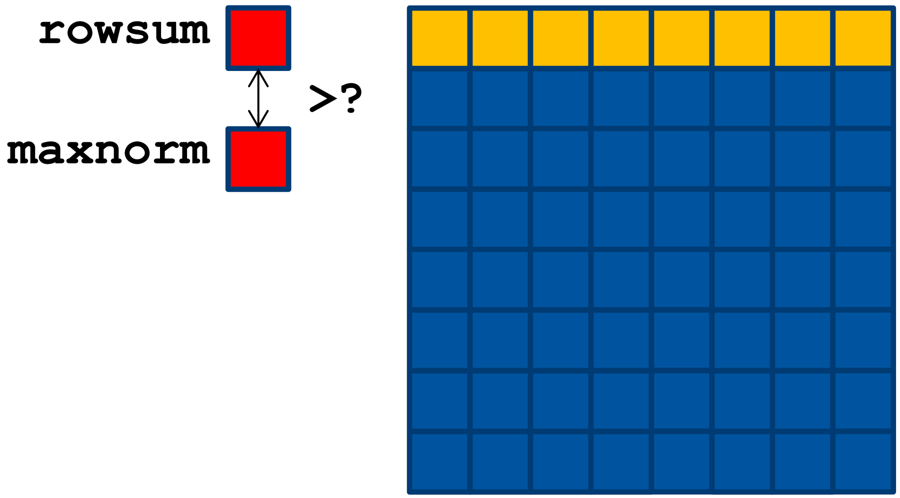
1. Norm calculation of a matrix
a) norm_max()



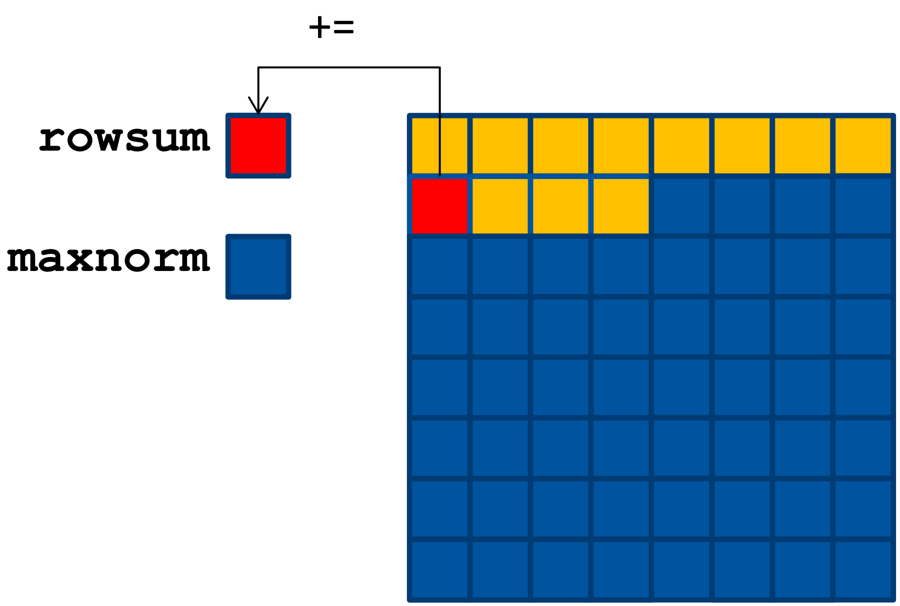
1. Norm calculation of a matrix
a) norm_max()



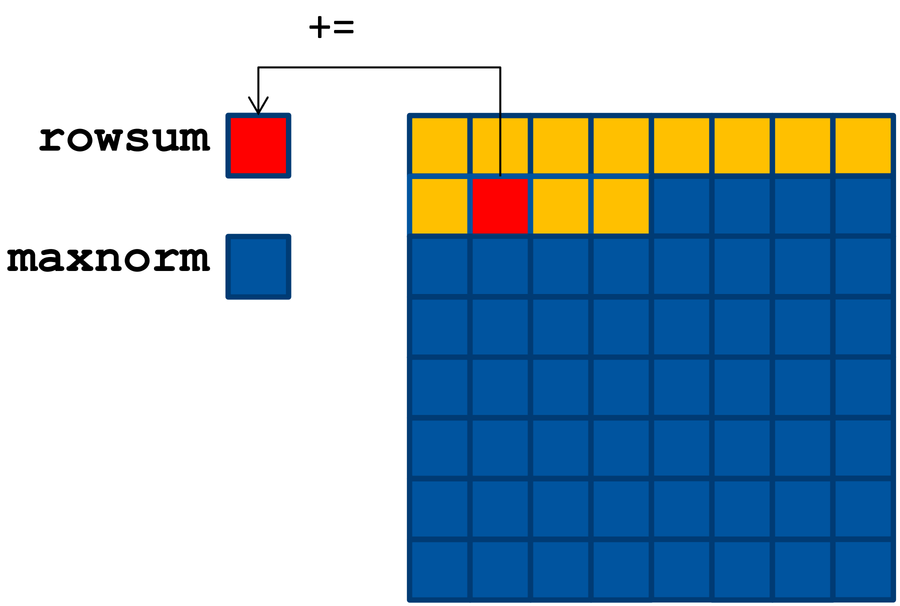
1. Norm calculation of a matrix
a) norm_max()



1. Norm calculation of a matrix
a) norm_max()



1. Norm calculation of a matrix
a) norm_max()



1. Norm calculation of a matrix
b) Calculate the time $T^{(1)}(n)$

b) Calculate the cache-miss-ratio ($\#cachemiss/\#memoryaccess$) and the time $T^{(1)}(n)$ of that algorithm in terms of T_A, T_M, T_C, l, and c where

T_A	time for an arithmetic operation
T_M	time for accessing the main memory
T_C	time for accessing the cache
l	size of a cache line (in elements)
c	size of the cache (in elements).

Hint: Auxiliary variables are stored in registers. Assume that the computation of $|\cdot|$ and the comparison of two real numbers both take time T_A. Assume for all calculations that the matrix doesn't fit to the cache ($c \ll n \cdot n$) and the cache is initially cold (empty).

1. Norm calculation of a matrix
b) Calculate the time $T^{(1)}(n)$

- n^2 **elements (the whole matrix) must be added together and every element must be examined by `abs()`:** $2n^2\,T_A$
- **Comparing the row sums:** nT_A
- **Loading the matrix elements can be divided in two different actions:**
  → Loading the data from the main memory (always a complete cache line) :
  $$\frac{n^2}{l}T_M$$
  → Loading data from the cache (cache hit because of "cache locality"):
  $$(n^2 - \frac{n^2}{l})\;T_C$$
- $T^{(1)}\,(n) = (2n^2 \;+\; n)\;T_A + (\frac{n^2}{l}\;)T_M + \;(n^2 - \frac{n^2}{l})\;T_C$
- $T^{(1)}\,(n) = (2n^2 + n)\;T_A + \frac{n^2}{8}180\;T_A + 35(n^2 - \frac{n^2}{8})\;T_A$
- $\boldsymbol{T^{(1)}\,(n) = (55{,}125\,n^2 \;+\; n)\,T_A}$
- $\boldsymbol{Cache - miss - ratio = (n^2/l)/n^2 = 1/l}$

$$\boxed{\begin{aligned} T_M &= 180\;T_A \\ T_C &= 35\;T_A \\ l &= 8 \end{aligned}}$$

1. Norm calculation of a matrix
c) norm1_col_wise()

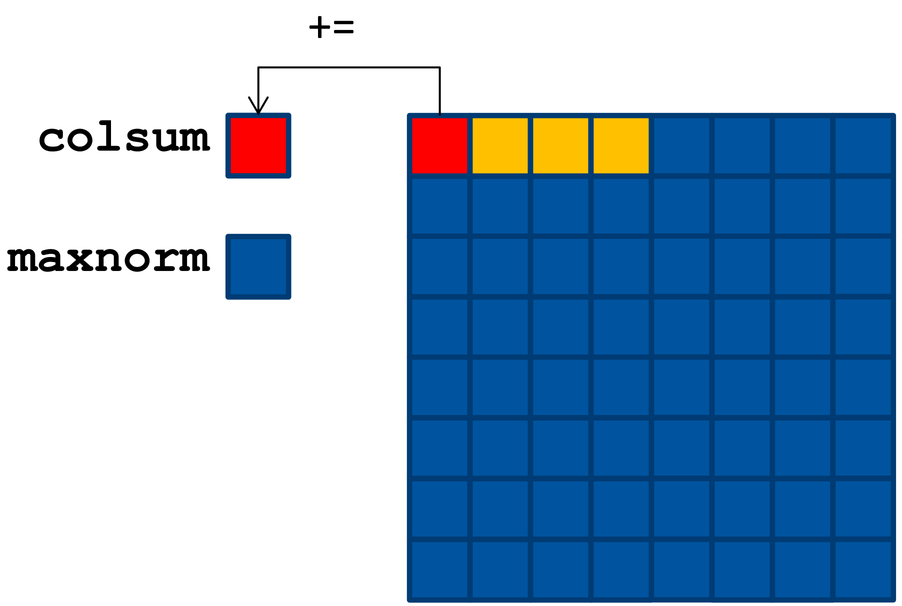
c) By switching the order of the two `for` loops in $\boxed{\text{a}}$), the algorithm now computes $\|A\|_1$. What time $T^{(2)}(n)$ does this algorithm take? Calculate the cache-miss-ratio.

Livedemo

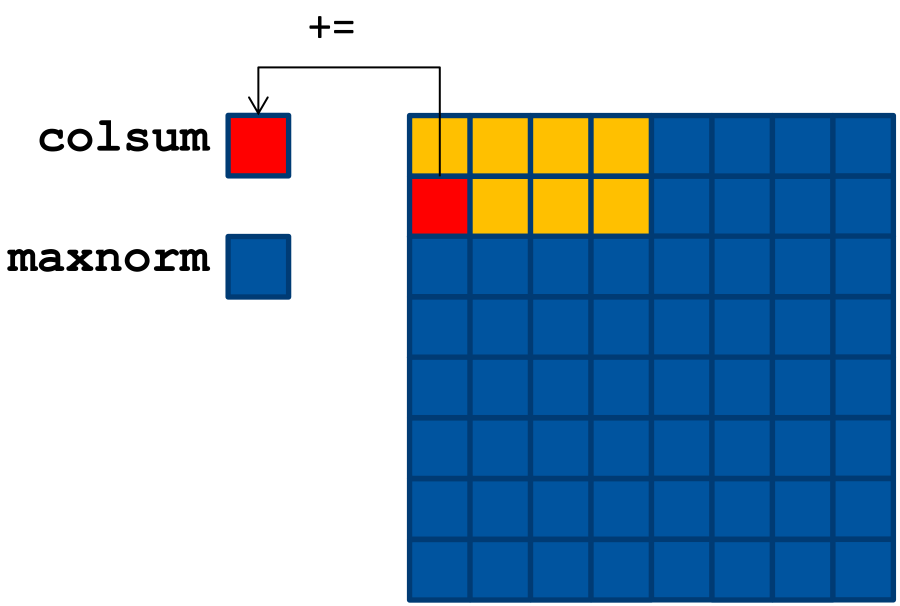
1. Norm calculation of a matrix
c) norm1_col_wise()

```
double norm1_col_wise(double** const A, const int n) {
   double colsum = 0., max_norm = -1.;
/* TODO Implement the one norm by switching
        the 2 for loops from the max norm */
    for (int j = 0; j < n; ++j) {
        colsum = 0.;
        for (int i = 0; i < n; ++i)
            colsum += abs(A[i][j]);
        if (colsum > max_norm)
            max_norm = colsum;
    }
    return max_norm;
}
```

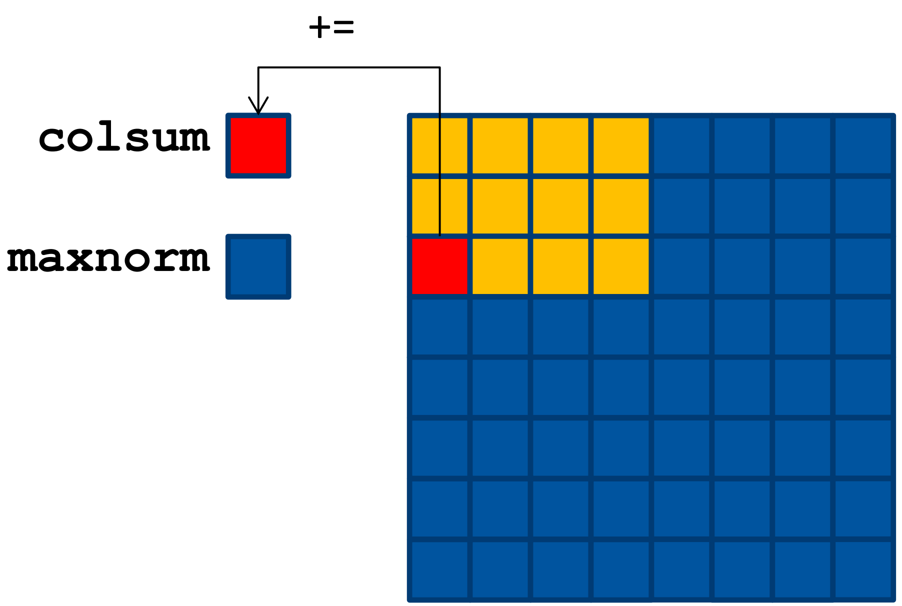
1. Norm calculation of a matrix
c) norm1_col_wise()



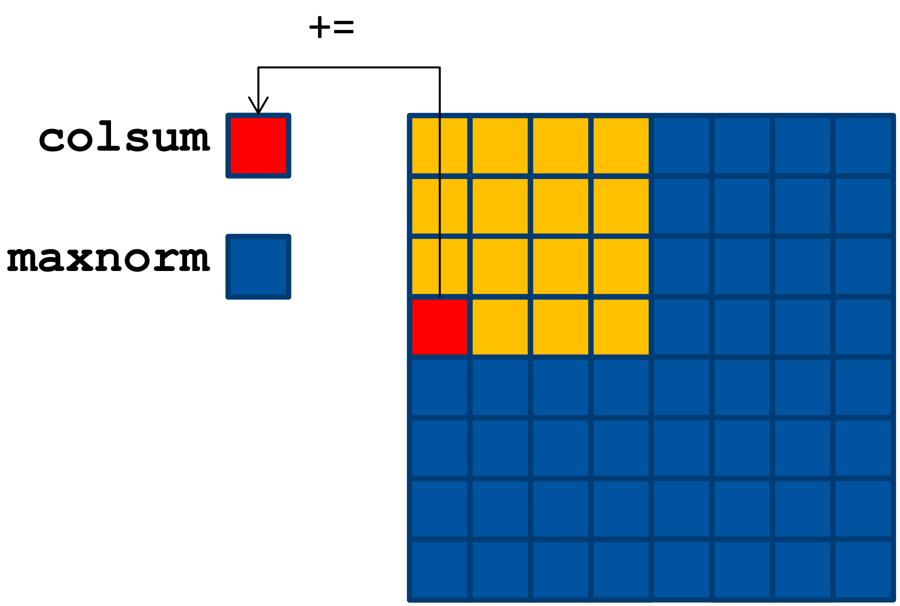
1. Norm calculation of a matrix
c) norm1_col_wise()



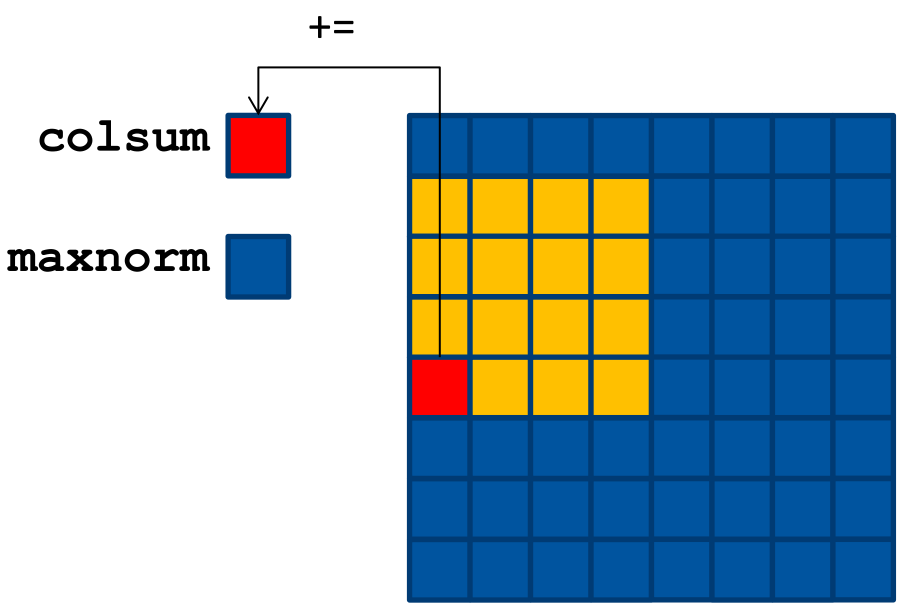
1. Norm calculation of a matrix
c) norm1_col_wise()



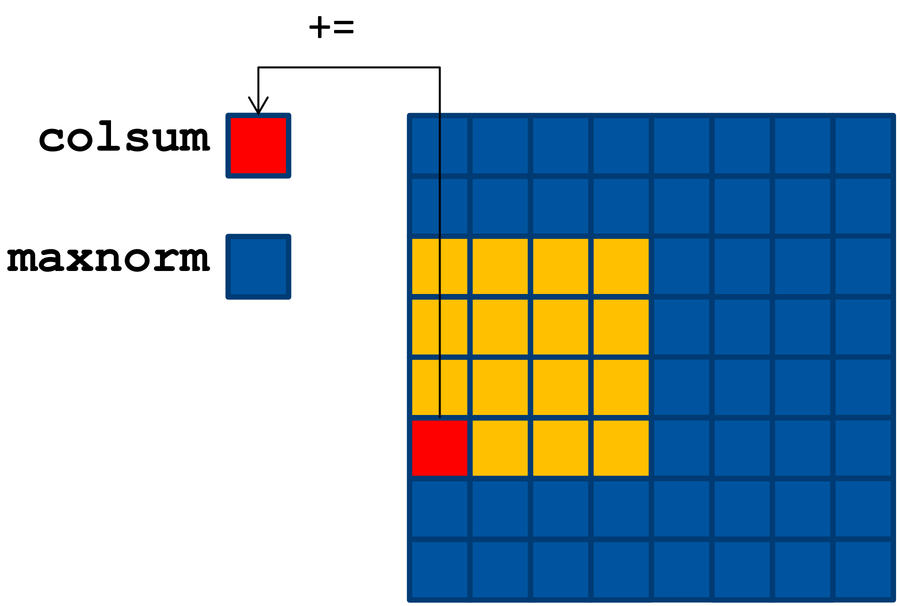
1. Norm calculation of a matrix
c) norm1_col_wise()



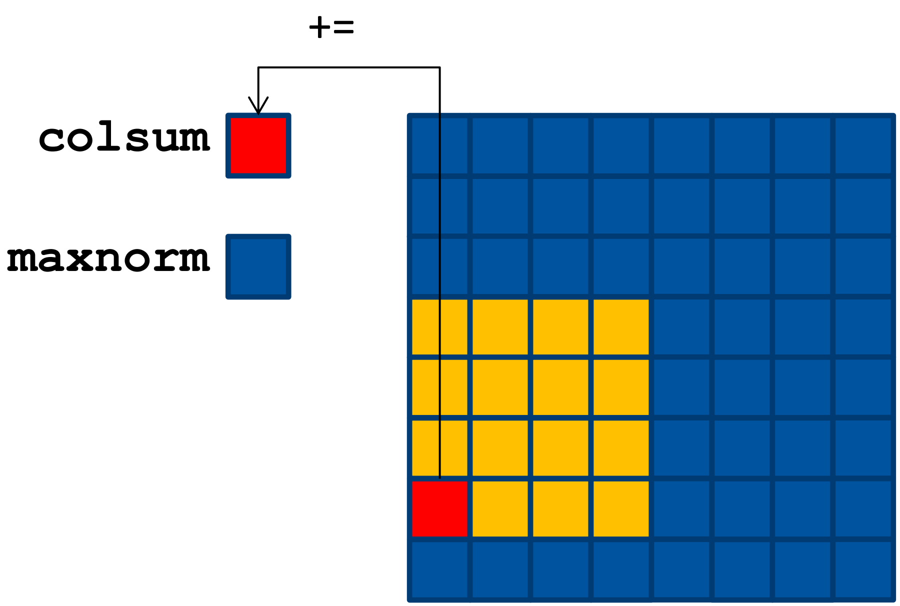
1. Norm calculation of a matrix
c) norm1_col_wise()



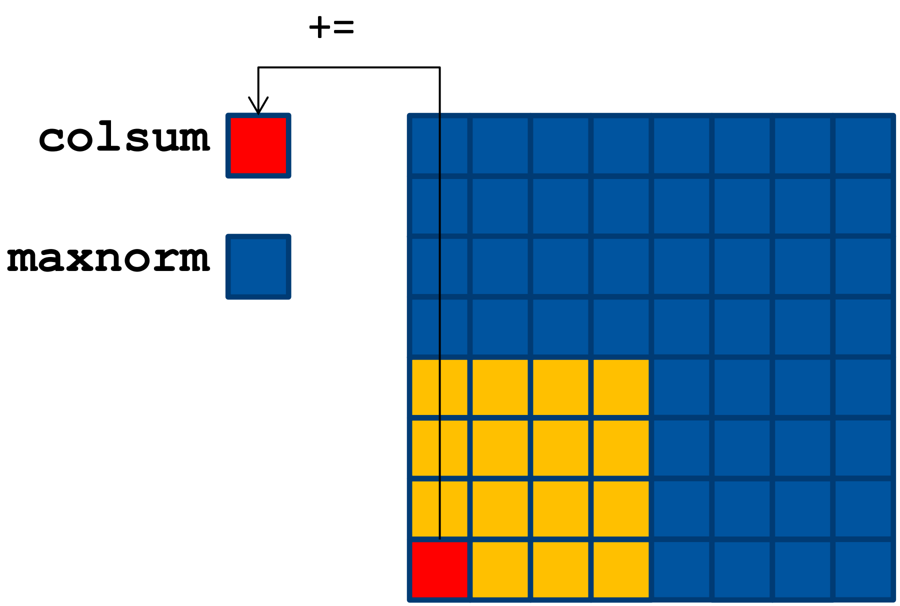
1. Norm calculation of a matrix
c) norm1_col_wise()



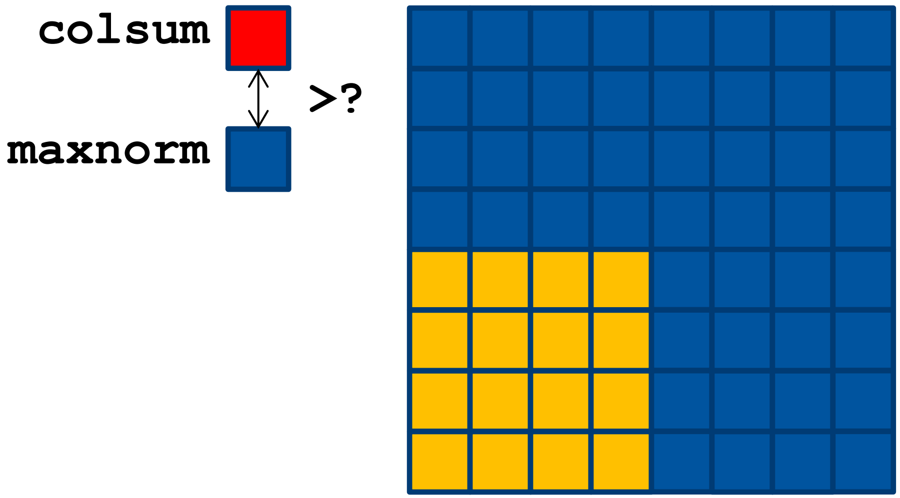
1. Norm calculation of a matrix
c) norm1_col_wise()



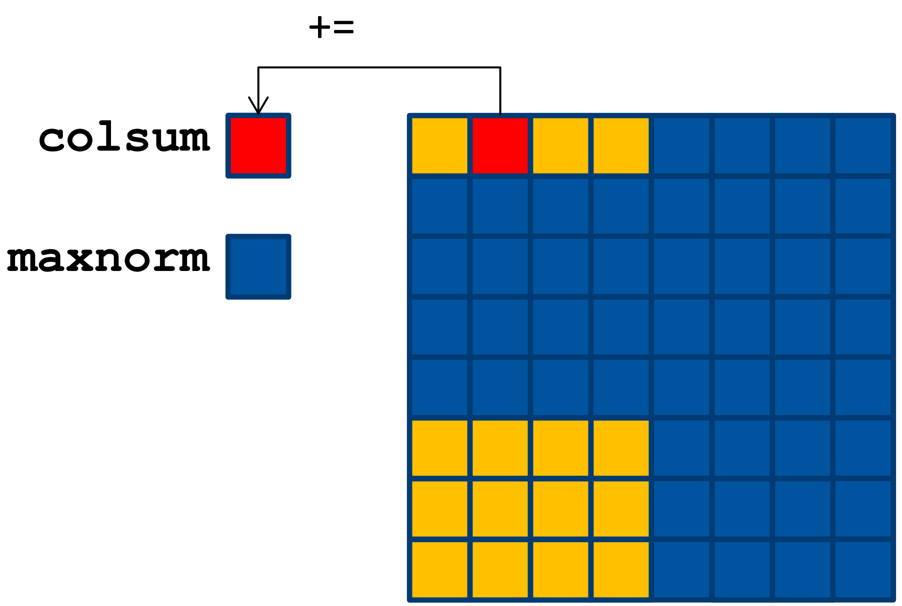
1. Norm calculation of a matrix
c) norm1_col_wise()



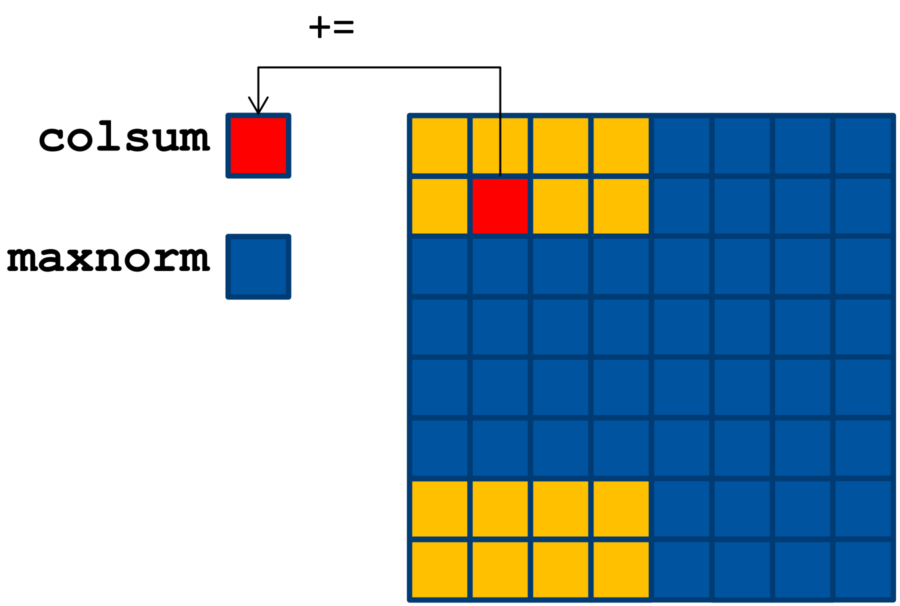
1. Norm calculation of a matrix
c) norm1_col_wise()



1. Norm calculation of a matrix
c) norm1_col_wise()



1. Norm calculation of a matrix
c) norm1_col_wise()



1. Norm calculation of a matrix
c) Time for $T^{(2)}$

- Time for arithmetic operations does not change

- Case $\boldsymbol{n \cdot l < c}$ ("Matrix fits into the cache"):

  → $T^{(2b)}(n) = T^{(1)}(n) = (2n^2 + n)\, T_A + \frac{n^2}{l}\, T_M + \left(n^2 - \frac{n^2}{l}\right) T_C$

  → $\boldsymbol{Cache - miss - ratio = \frac{\frac{n^2}{l}}{n^2} = \frac{1}{l}}$

- Cases $\boldsymbol{n > c}$ and $\boldsymbol{n \cdot l > c}$ ("Matrix does not fit into the cache"):

  → No cache hit: Every value has to be loaded from the main memory

  → $T^{(2)}(n) = (2n^2 + n)\, T_A + n^2\, T_M$

  → $\boldsymbol{Cache - miss - ratio = \frac{n^2}{n^2} = 1}$

1. Norm calculation of a matrix
d) norm1_row_wise()

d) Design an algorithm to compute $\|A\|_1$ with consecutive memory access and calculate its runtime $T^{(3)}(n)$.

Hint: Use an auxiliary array for the column sums.

Livedemo

1. Norm calculation of a matrix
d) norm1_row_wise()

```
double norm1_row_wise(double** const A, const int n)
{
/* TODO You need an auxiliary array
        for the column sums */
    double *colsums = new double[n];
    double max_norm = -1.;

/* TODO The auxiliary array must be initialised */
    for (int i = 0; i < n; ++i)
        colsums[i] = 0;

...
```

Hint: O(n) → We neglect this for T later.

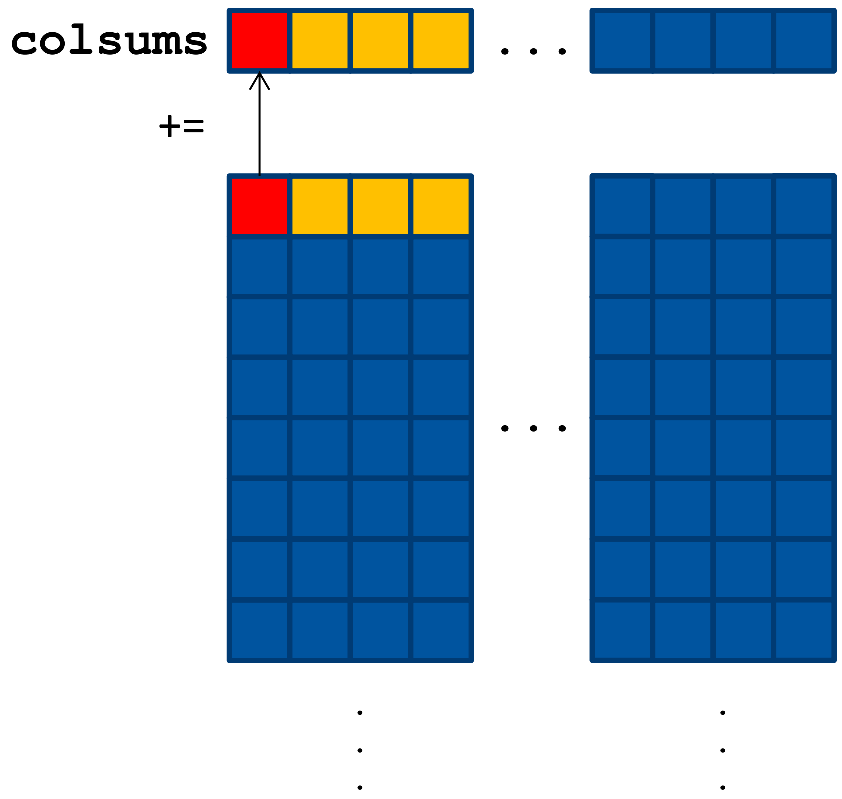
Exercise 02
Julian Miller | IT Center der RWTH Aachen University

1. Norm calculation of a matrix
d) norm1_row_wise()

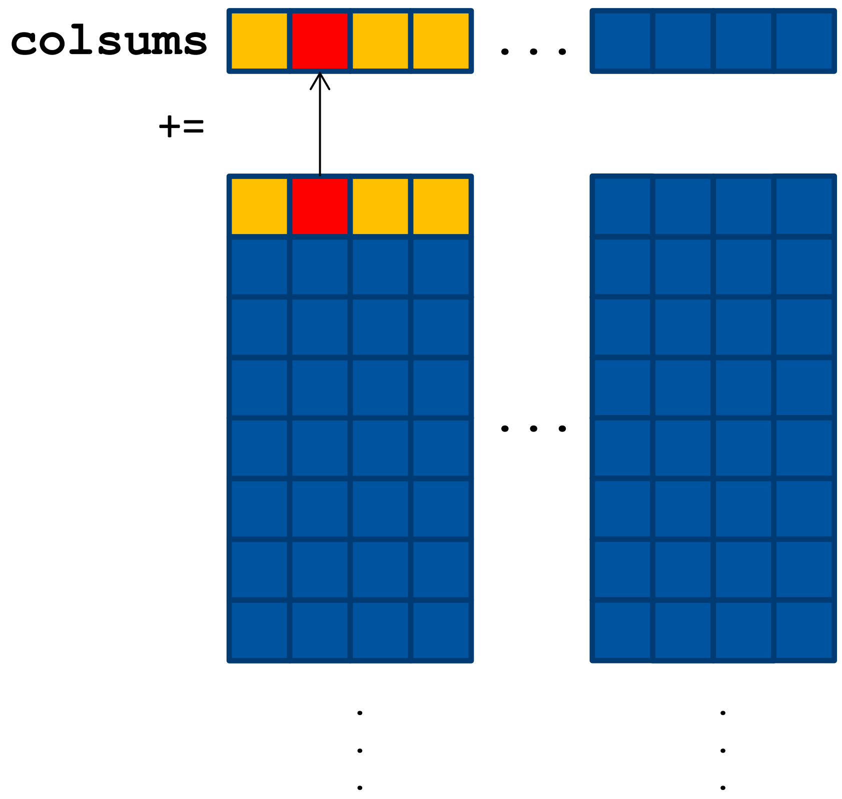
```
...

/* TODO Now compute the column sums with
        consecutive memory access */
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            colsums[j] += abs(A[i][j]);

/* TODO Find the largest column sum */
    for (int i = 0; i < n; ++i)
        if (colsums[i] > max_norm)
            max_norm = colsums[i];

    delete[] colsums;
    return max_norm;
}
```

Hint: O(n) → We neglect this for T later.

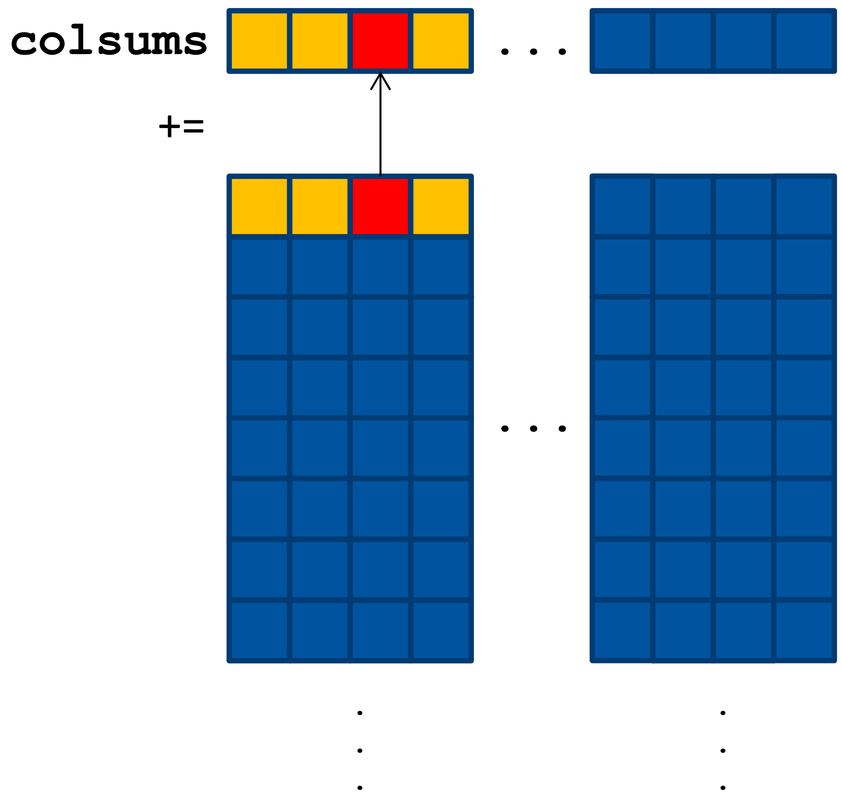
1. Norm calculation of a matrix
d) norm1_row_wise()



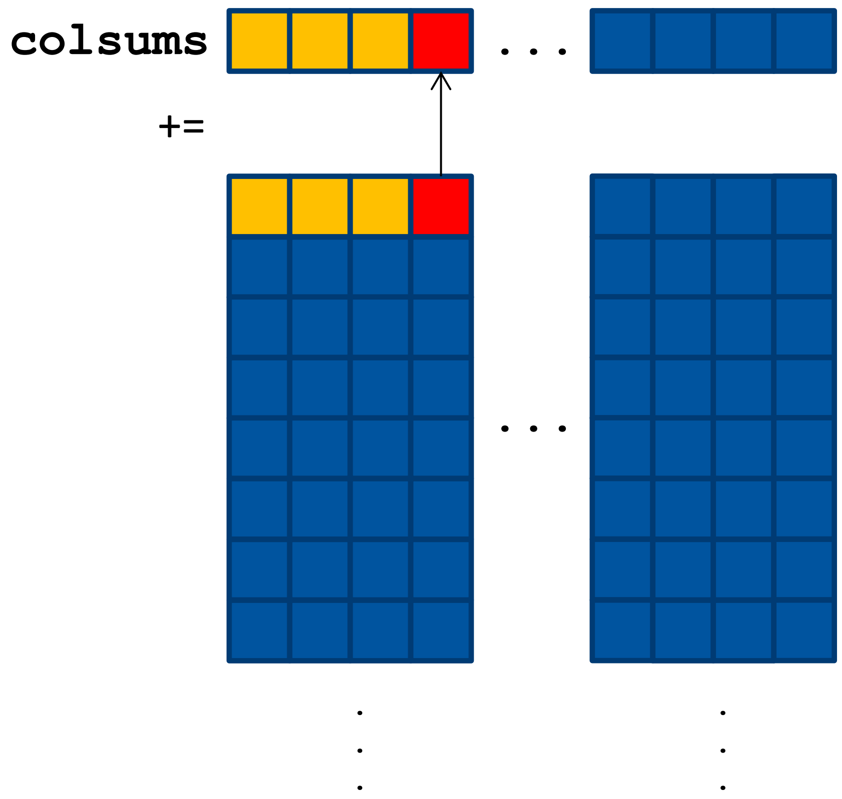
1. Norm calculation of a matrix
d) norm1_row_wise()



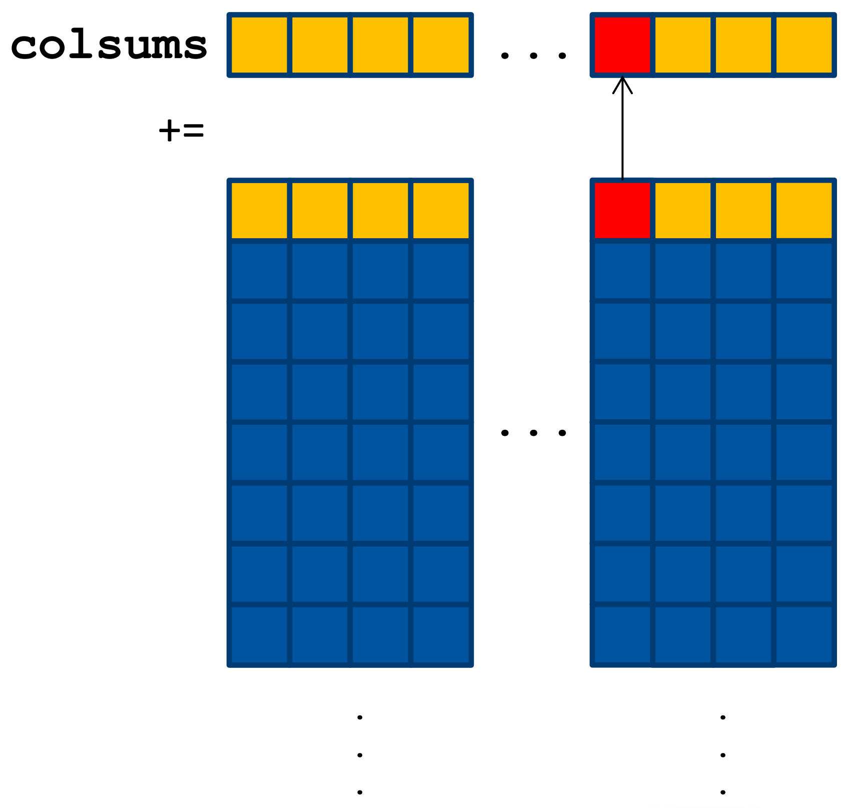
1. Norm calculation of a matrix
d) norm1_row_wise()



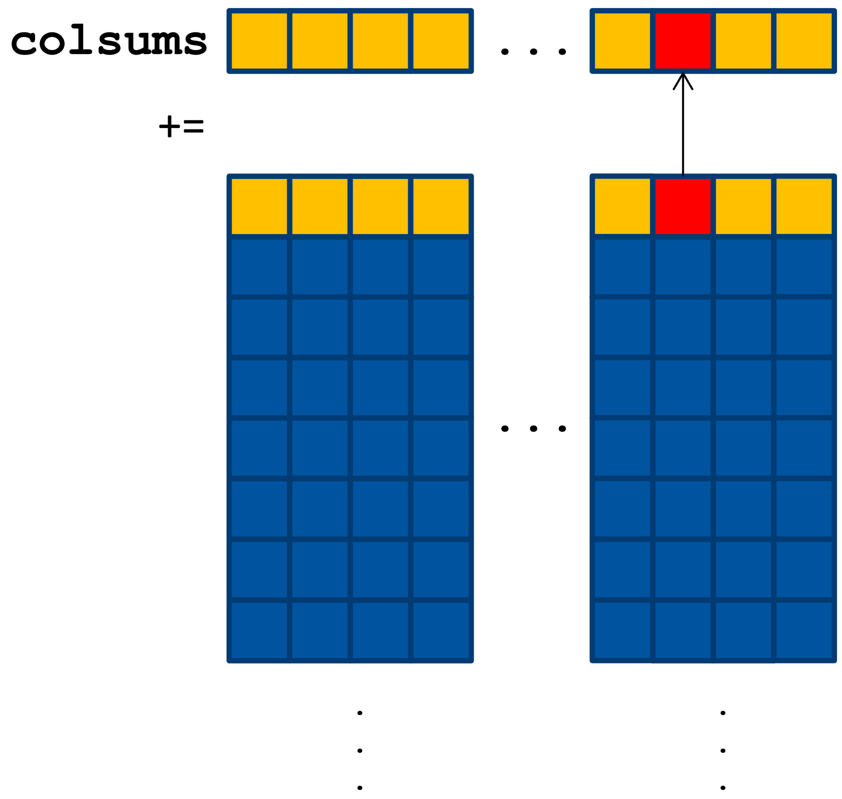
1. Norm calculation of a matrix
d) norm1_row_wise()



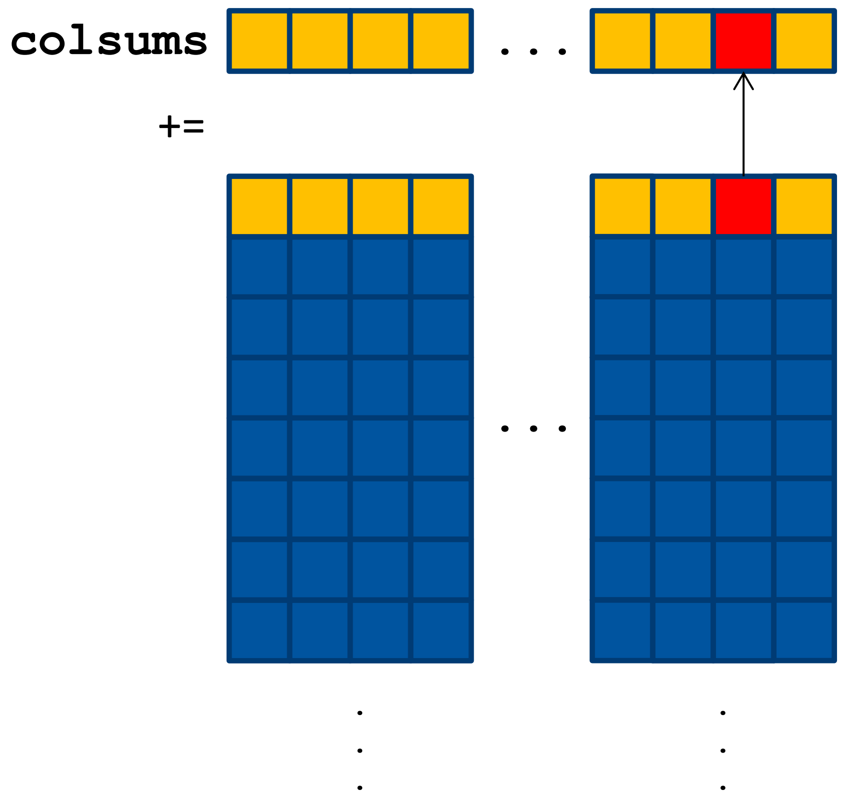
1. Norm calculation of a matrix
d) norm1_row_wise()



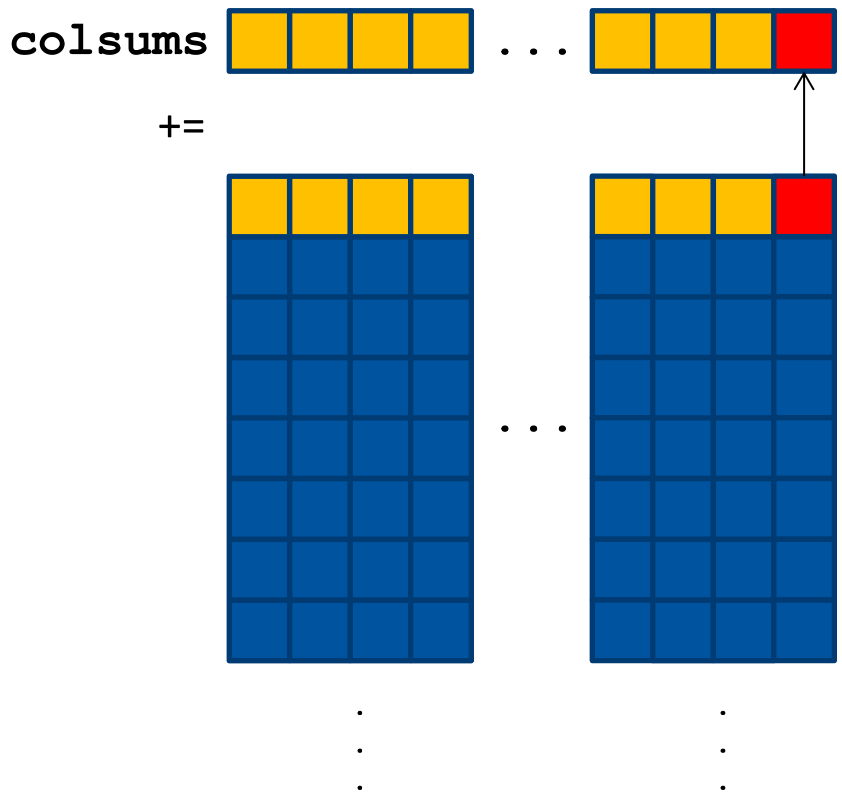
1. Norm calculation of a matrix
d) norm1_row_wise()



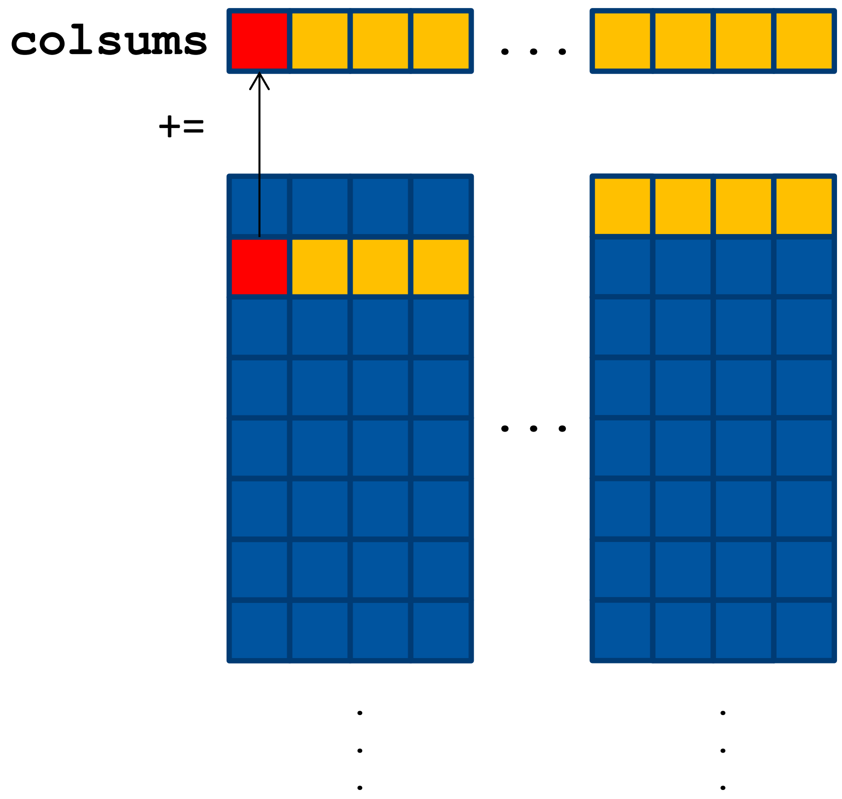
1. Norm calculation of a matrix
d) norm1_row_wise()



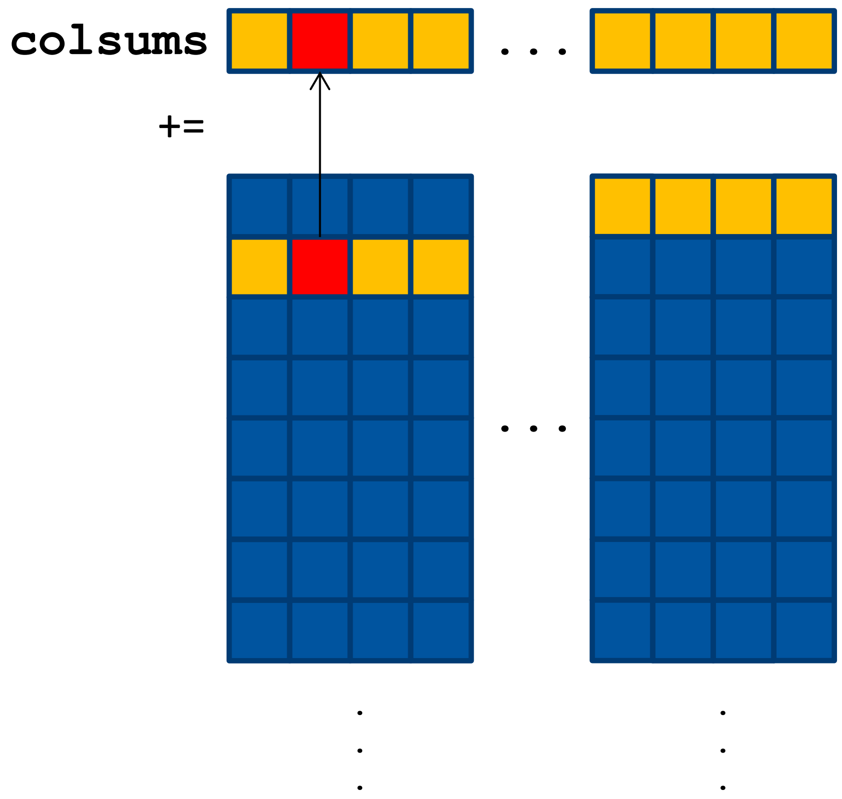
1. Norm calculation of a matrix
d) norm1_row_wise()



1. Norm calculation of a matrix
d) norm1_row_wise()



1. Norm calculation of a matrix
d) norm1_row_wise()



1. Norm calculation of a matrix
d) Time for $T^{(3)}$

- Assumption: Auxiliary array is much bigger than the cache
- Time for arithmetic operations does not change
- Additional memory access to auxiliary array:

$$\frac{n^2}{l} T_M + \left(n^2 - \frac{n^2}{l}\right) T_C$$

Hint: We neglect the initialization of the auxiliary array and the comparison, because they are small compared to this term (O(n)). If you want to respect them you need to add $\frac{n}{l} T_M + \left(n - \frac{n}{l}\right) T_C$.

- $T^{(3)}(n) = T^{(1)} + \frac{n^2}{l} T_M + \left(n^2 - \frac{n^2}{l}\right) T_C =$

$$(2n^2 + n)\, T_A + \frac{n^2}{l} T_M + \left(n^2 - \frac{n^2}{l}\right) T_C + \frac{n^2}{l} T_M + \left(n^2 - \frac{n^2}{l}\right) T_C =$$

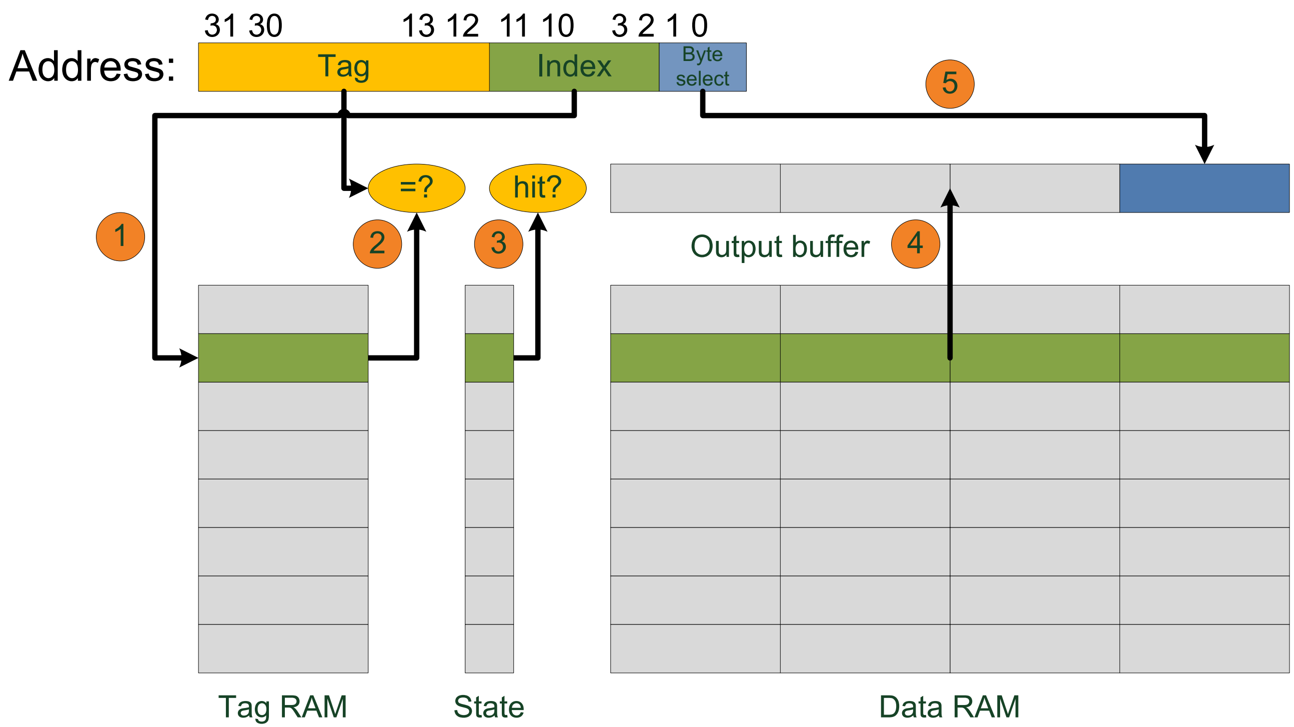
$$\boldsymbol{(2n^2 + n) T_A + 2\left[\frac{n^2}{l} T_M + \left(n^2 - \frac{n^2}{l}\right) T_C\right]}$$

1. Norm calculation of a matrix
e) Cache mapping

e) What is the worst case scenario regarding cache-miss-ratio for the algorithm of d) when we assume a direct-mapped cache?

Recap: Cache organization

- **Direct Mapped Cache:**
  → Part of the address selects one entry in the tag RAM for comparison



1. Index selects one cache line
2. Check if selected tags are equal (no => miss)
3. Check if cache line is valid (no => miss)
4. Copy data to output buffer
5. Select required part from cacheline (if data is valid)

1. Norm calculation of a matrix
e) Cache mapping

- **Worst case:**

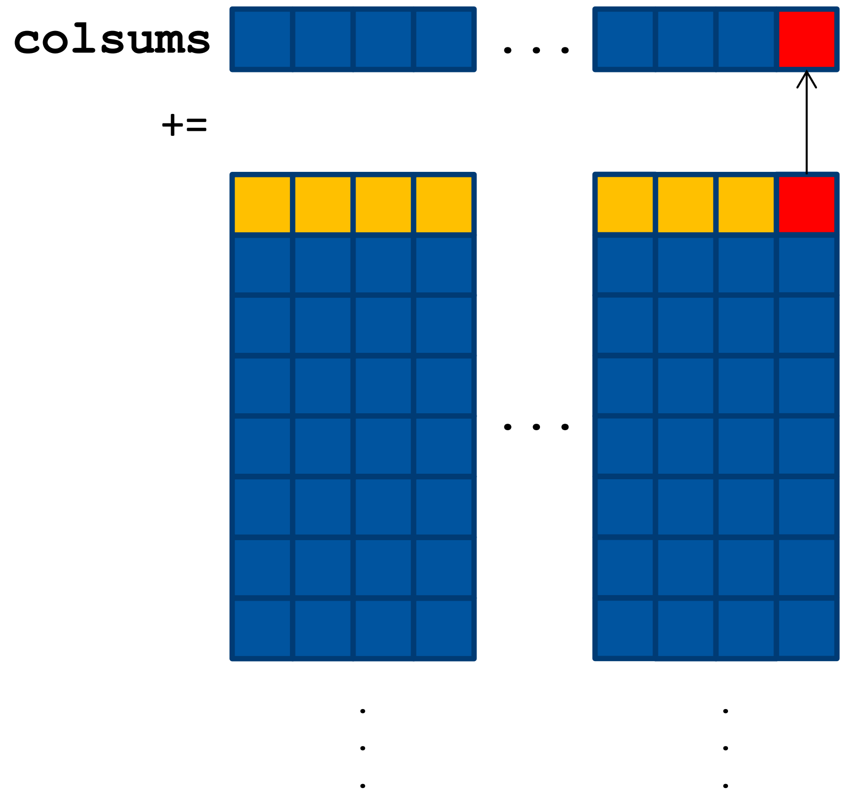
  colsums[i] and A[x][i] have same index

  $(colsums[i] \equiv A[x][i]\ mod\ c)$

- **colsums[i] += A[x][i]**

  → LOAD of A[x][i] evicts colsums[i] from cache

  → LOAD/STORE of colsums[i] has always cache miss



1. Norm calculation of a matrix
f) Calculate the Speed-up

f) Calculate the speed-up of the algorithm of d) with respect to the one of c).

$Hint$: $Speedup = \frac{T^{(2)}(n)}{T^{(3)}(n)}$

Assume that the following relations hold for an imaginary processor type:

$$\begin{aligned} T_M &= 180\, T_A \\ T_C &= 35\, T_A \\ l &= 8. \end{aligned}$$

(optional) Develop a block-version of the algorithm of d) and formulate its runtime $T^{(4)}(n)$ (cf. slide 302 f.).

1. Norm calculation of a matrix
f) Calculate the speed-up

- $T^{(2)}(n) = (2n^2 + n)\,T_A + n^2\,T_M$

- **If the auxiliary array is much bigger than the cache**
- $T^{(3)}(n) = (2n^2 + n)T_A + 2\left[\frac{n^2}{l}T_M + \left(n^2 - \frac{n^2}{l}\right)T_C\right]$

- **In the blocked version the used part of the auxiliary array always fits into the cache**
- $T^{(4)}(n) = (2n^2 + n)T_A + \frac{n^2}{l}T_M + \left(n^2 - \frac{n^2}{l}\right)T_C + n^2T_C$

1. Norm calculation of a matrix
f) Calculate the speed-up

- `norm1_col_wise()`: $T^{(2)}(n) = (182\,n^2 + n)T_A$
- `norm1_row_wise()`: $T^{(3)}(n) = (108.25\,n^2 + n)T_A$
- `norm1_blocking()`: $T^{(4)}(n) = (90.125\,n^2 + n)T_A$

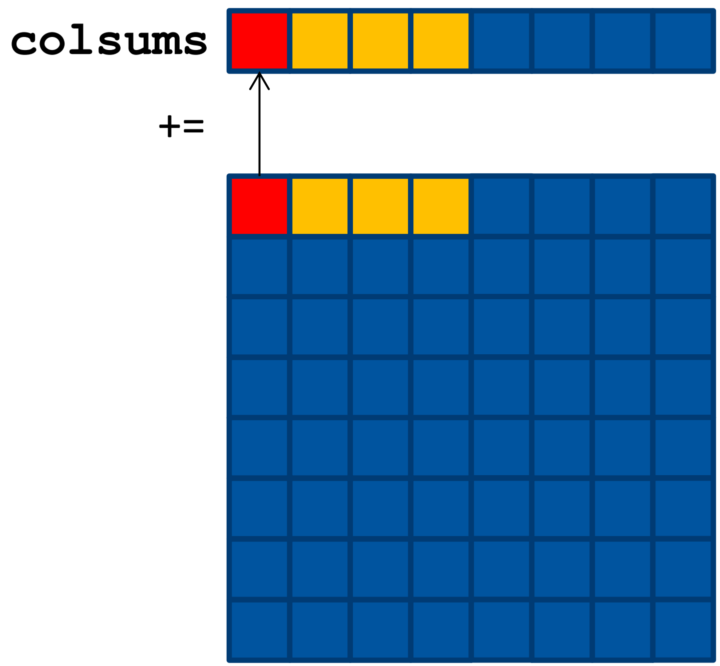
- **Speed-up of the different Versions of `norm1_*()`**

for big n

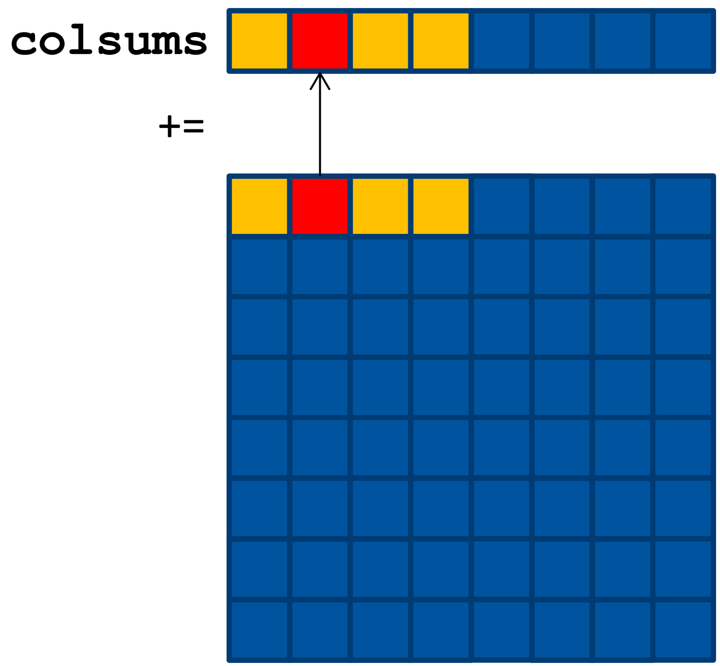
$$\rightarrow S(n) = \frac{T_{base}(n)}{T_{opt}(n)} = \lim_{n\to\infty}\left(\frac{a \cdot n^2 + n}{b \cdot n^2 + n}\right) = \lim_{n\to\infty}\left(\frac{a + \frac{1}{n}}{b + \frac{1}{n}}\right) = \frac{a}{b}$$

Algorithms	Speed-up
$T^{(2)} / T^{(3)}$	1.68
$T^{(2)} / T^{(4)}$	2.02
$T^{(3)} / T^{(4)}$	1.20

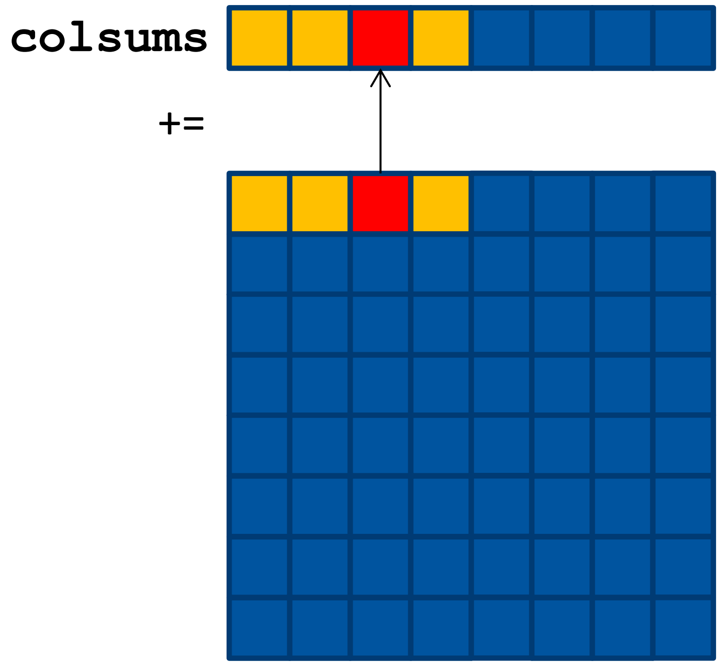
1. Norm calculation of a matrix (optional) norm1_blocking()



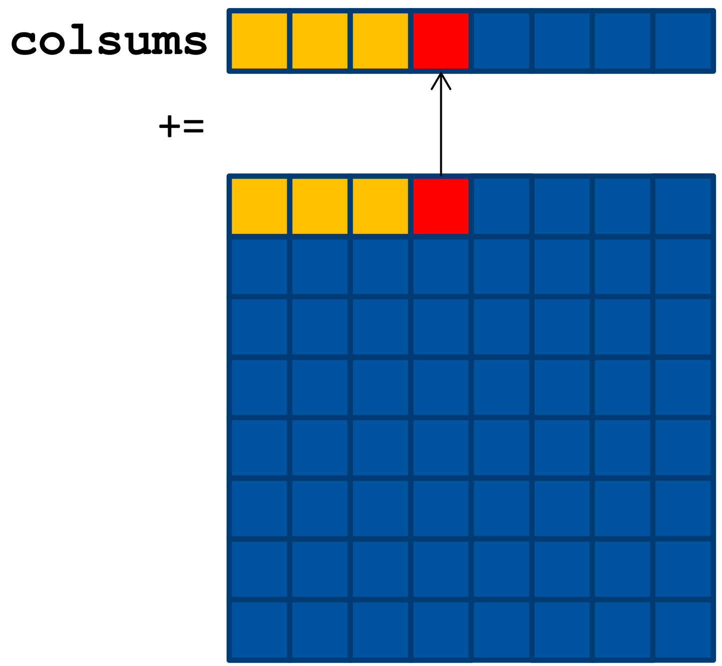
1. Norm calculation of a matrix (optional) norm1_blocking()



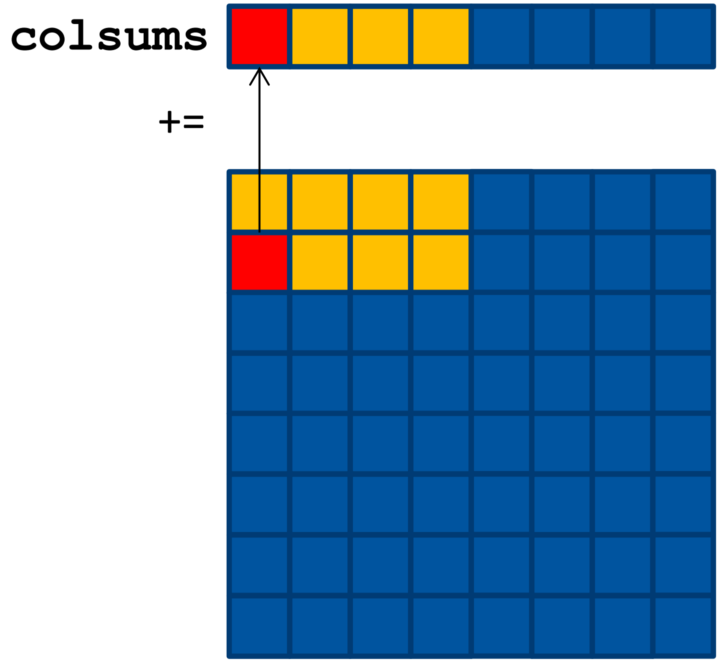
1. Norm calculation of a matrix (optional) norm1_blocking()



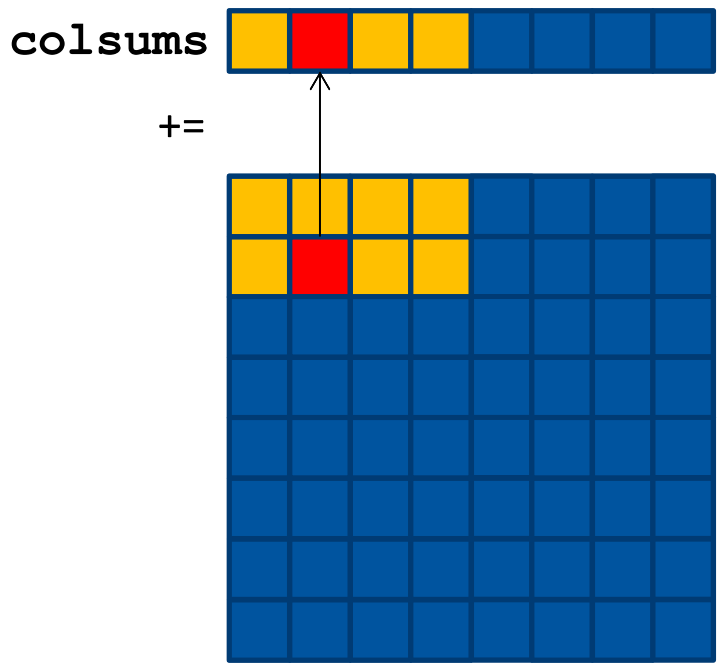
1. Norm calculation of a matrix (optional) norm1_blocking()



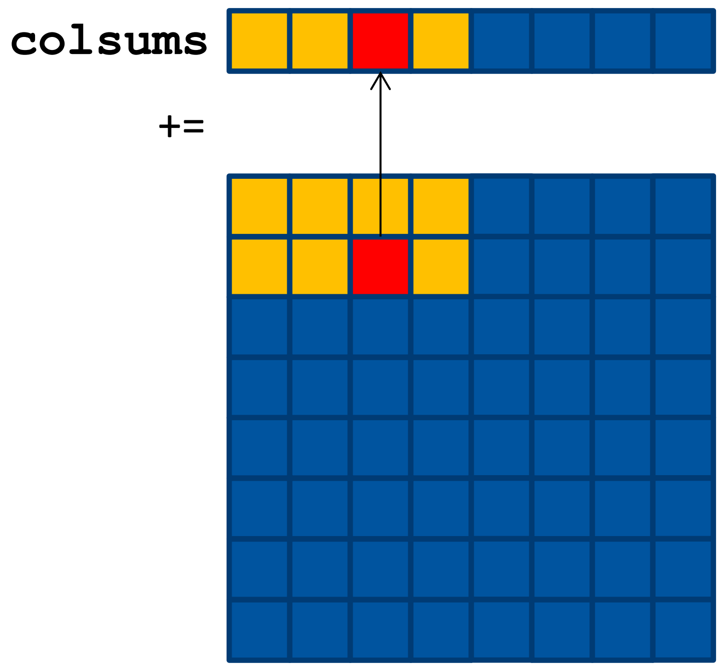
1. Norm calculation of a matrix (optional) norm1_blocking()



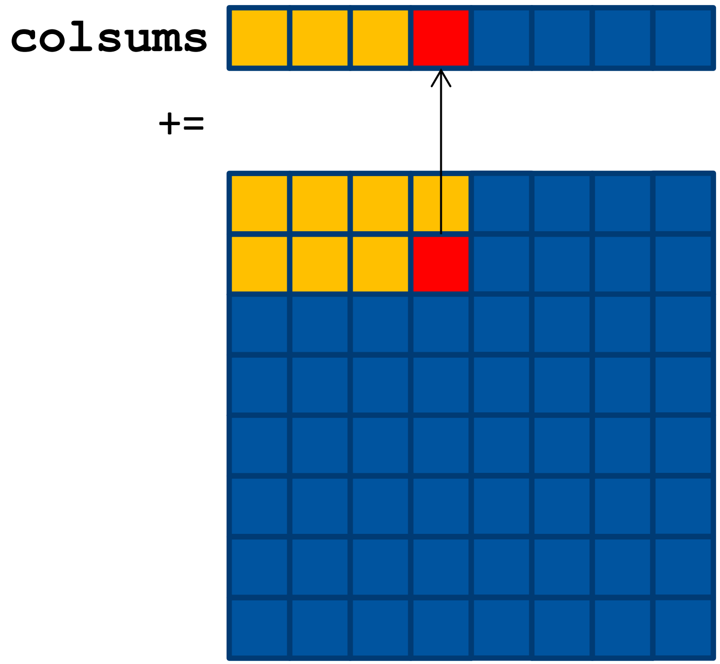
1. Norm calculation of a matrix (optional) norm1_blocking()



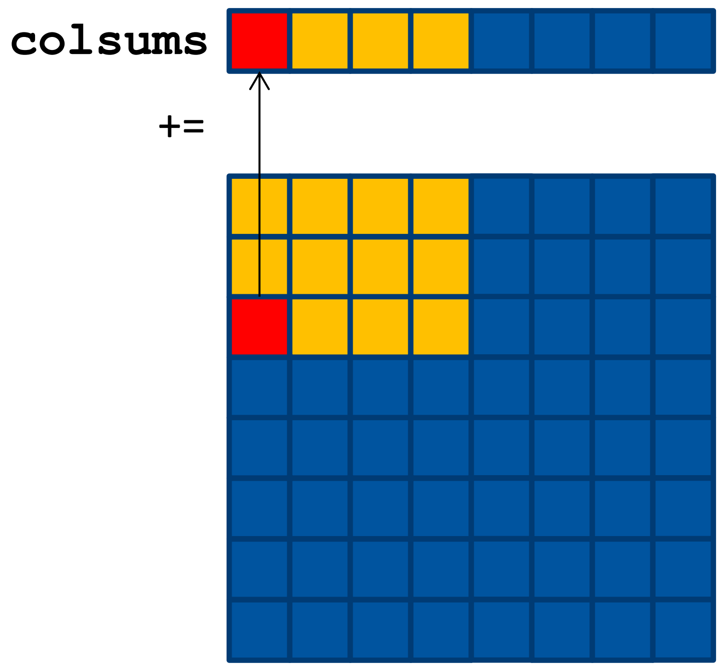
1. Norm calculation of a matrix (optional) norm1_blocking()



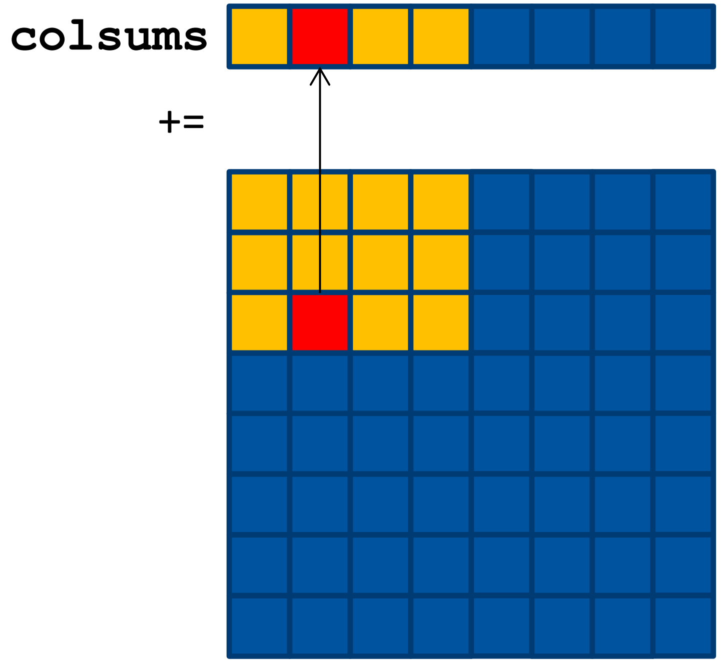
1. Norm calculation of a matrix (optional) norm1_blocking()



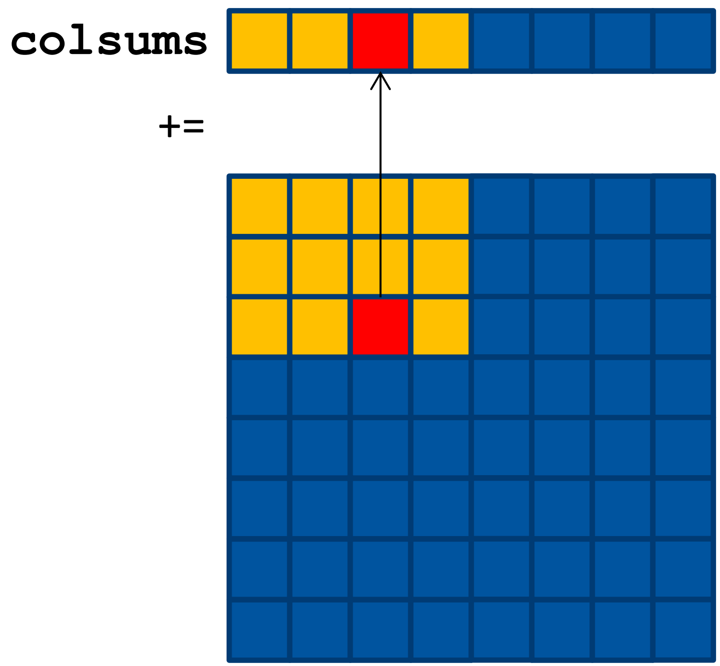
1. Norm calculation of a matrix (optional) norm1_blocking()



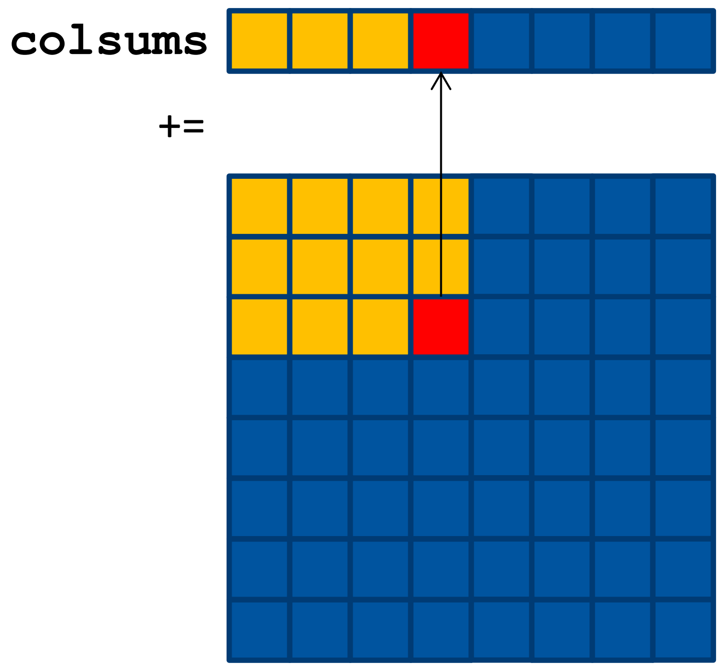
1. Norm calculation of a matrix (optional) norm1_blocking()



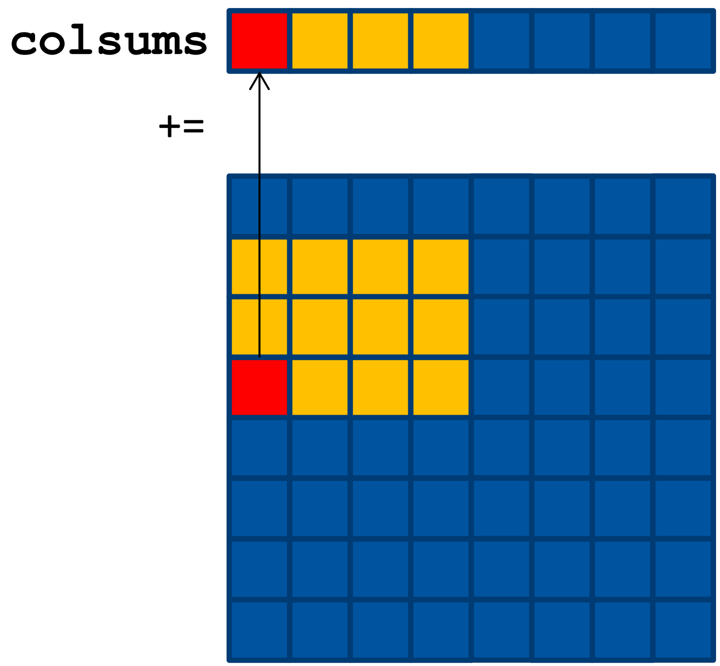
1. Norm calculation of a matrix (optional) norm1_blocking()



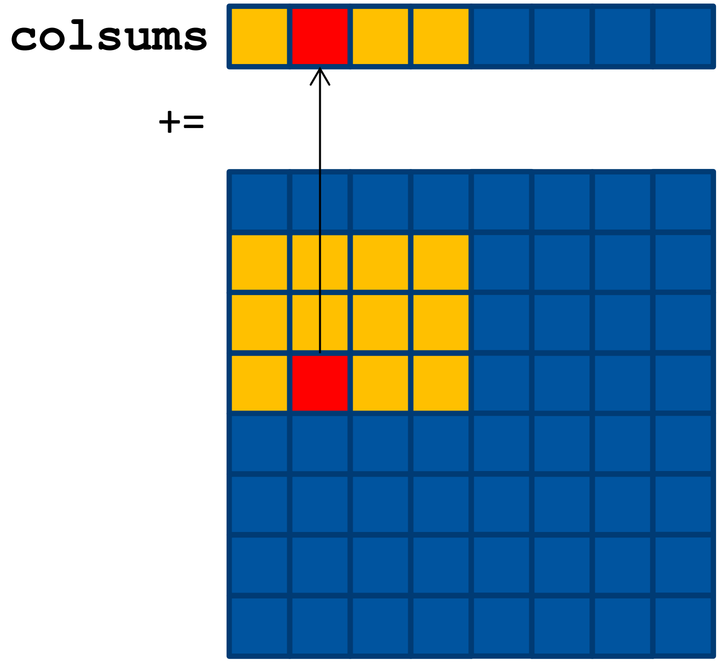
1. Norm calculation of a matrix (optional) norm1_blocking()



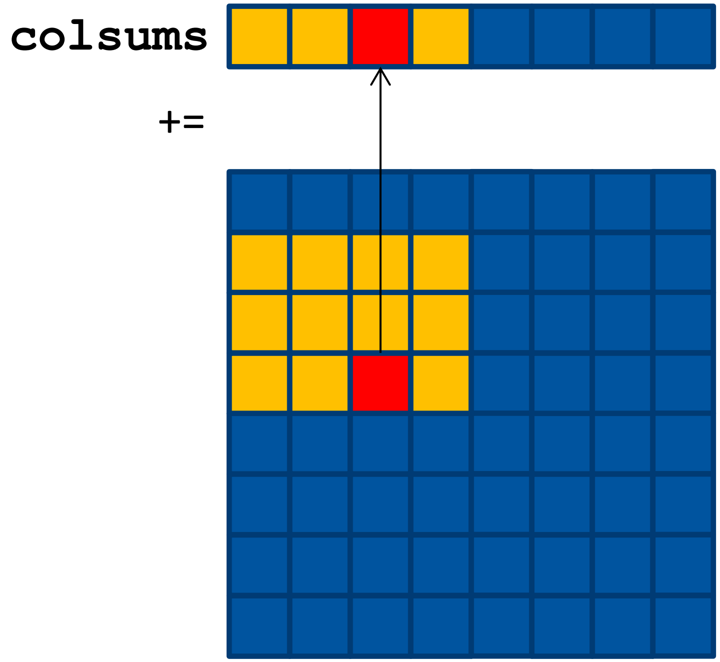
1. Norm calculation of a matrix (optional) norm1_blocking()



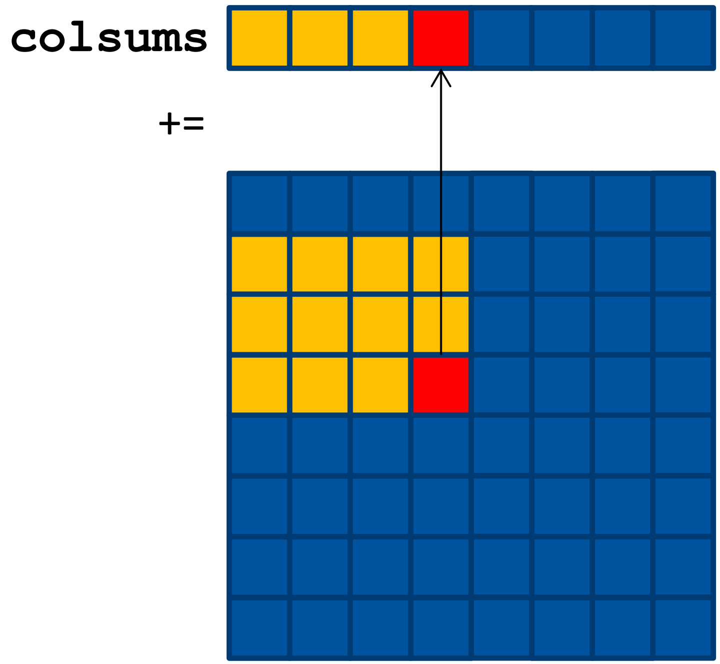
1. Norm calculation of a matrix (optional) norm1_blocking()



1. Norm calculation of a matrix (optional) norm1_blocking()



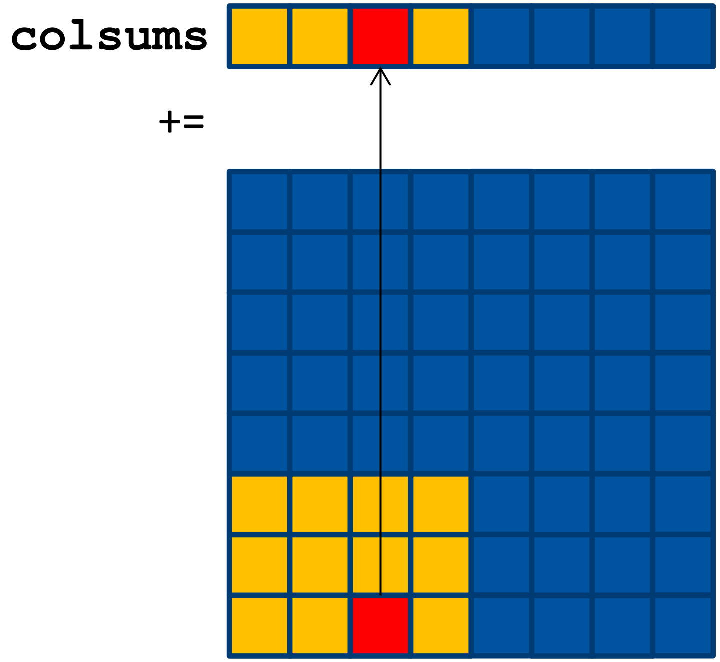
1. Norm calculation of a matrix (optional) norm1_blocking()



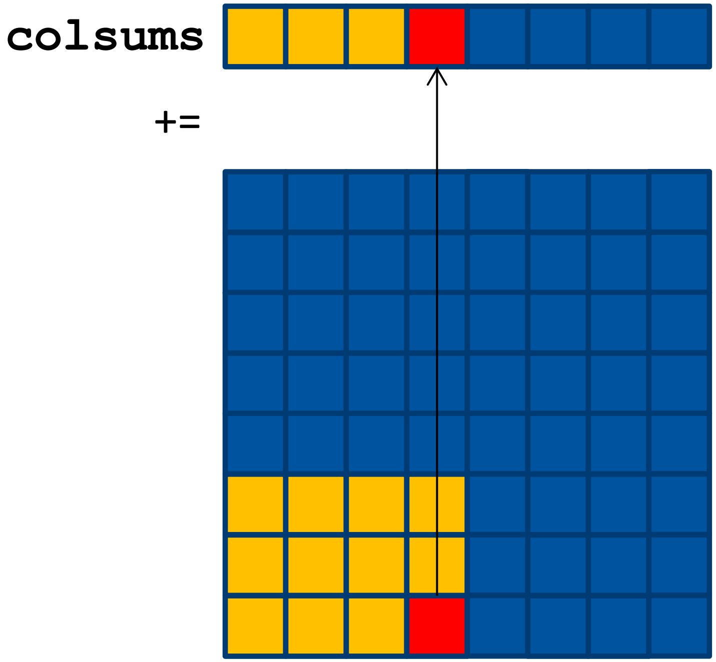
1. Norm calculation of a matrix (optional) norm1_blocking()

. . .

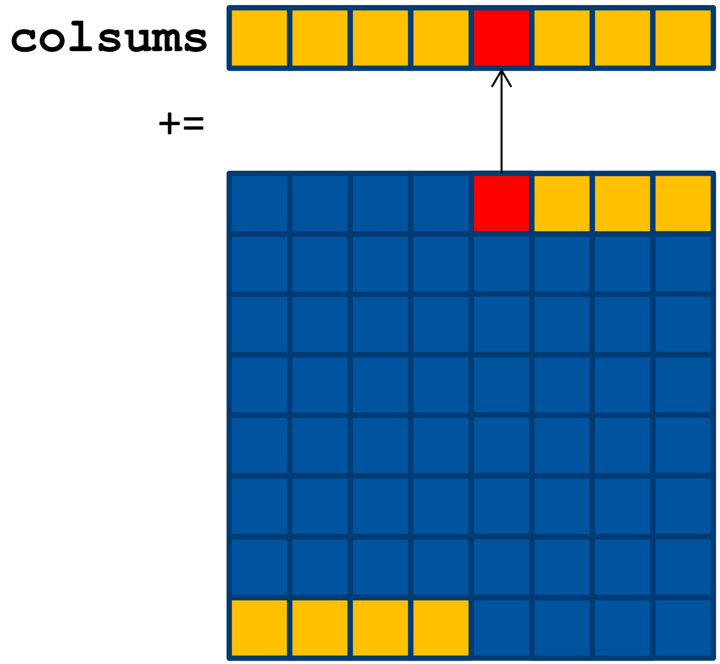
1. Norm calculation of a matrix (optional) norm1_blocking()



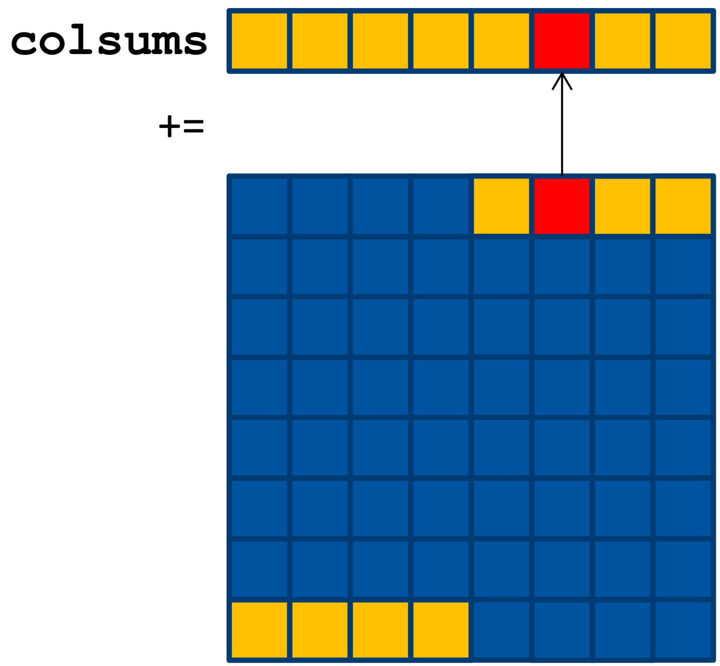
1. Norm calculation of a matrix (optional) norm1_blocking()



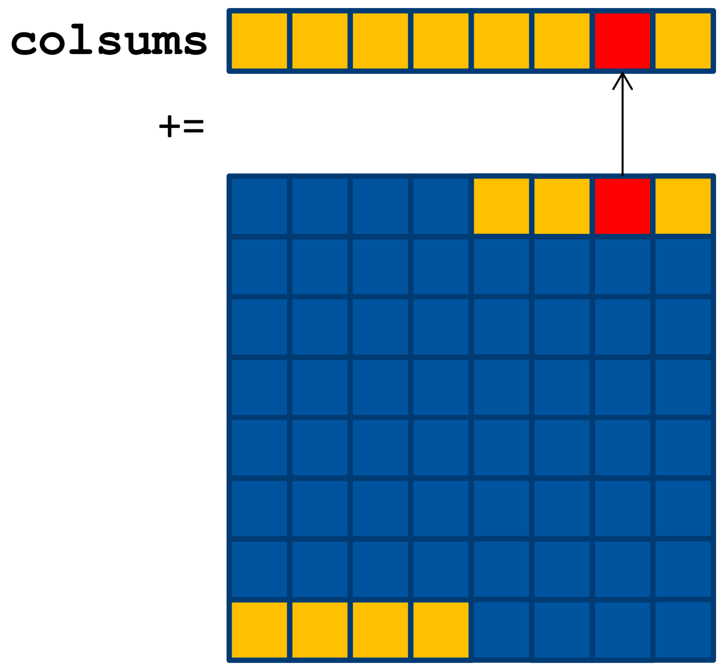
1. Norm calculation of a matrix (optional) norm1_blocking()



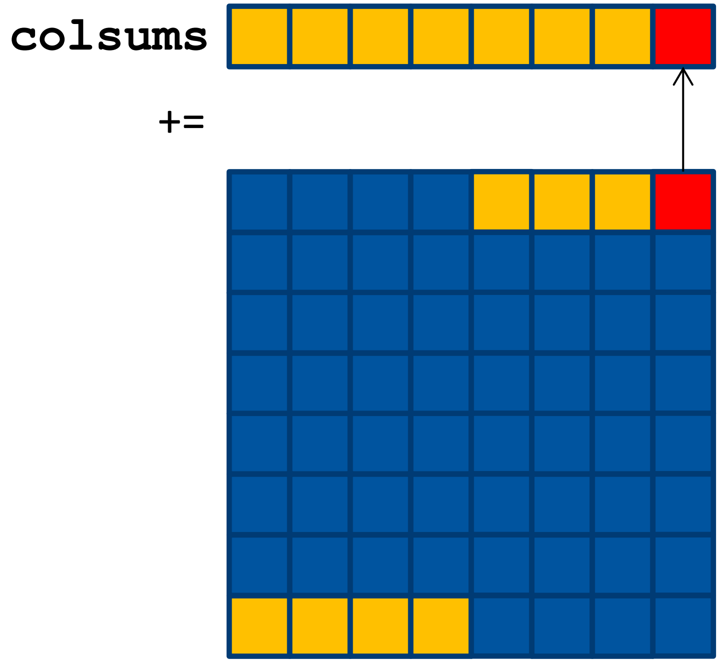
1. Norm calculation of a matrix (optional) norm1_blocking()



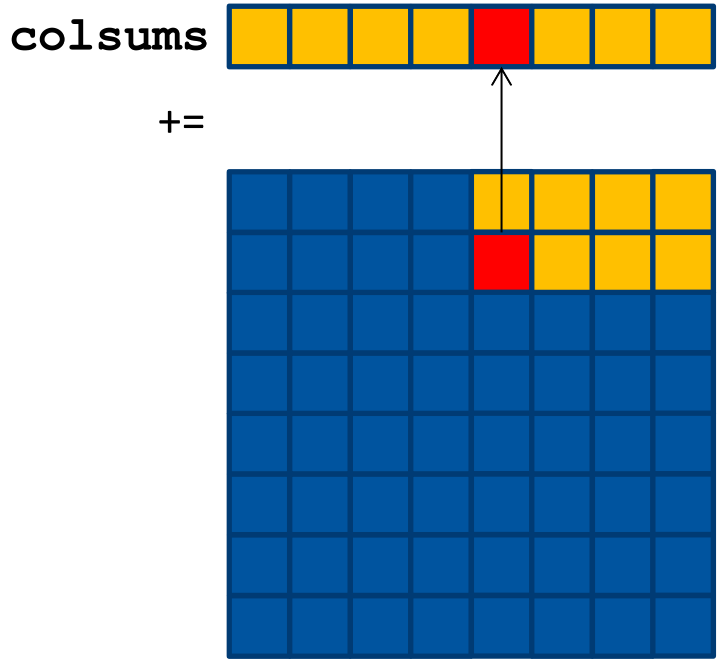
1. Norm calculation of a matrix (optional) norm1_blocking()



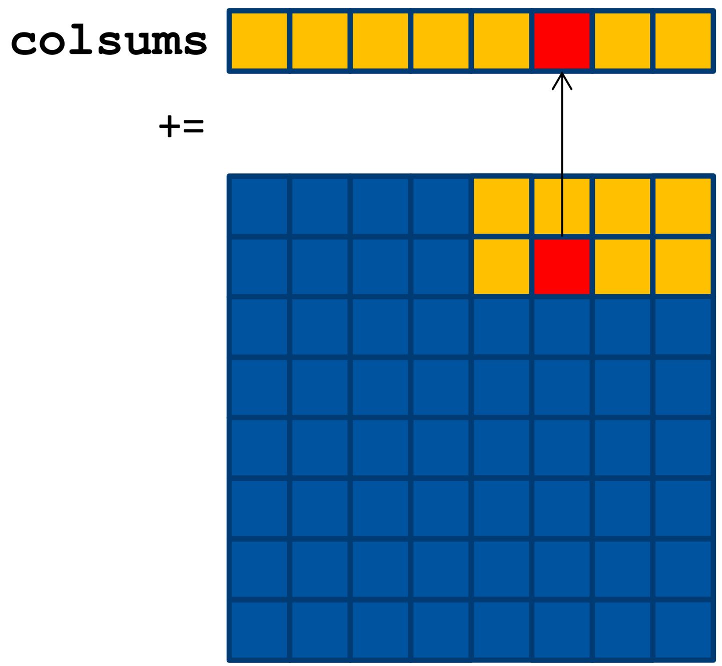
1. Norm calculation of a matrix (optional) norm1_blocking()



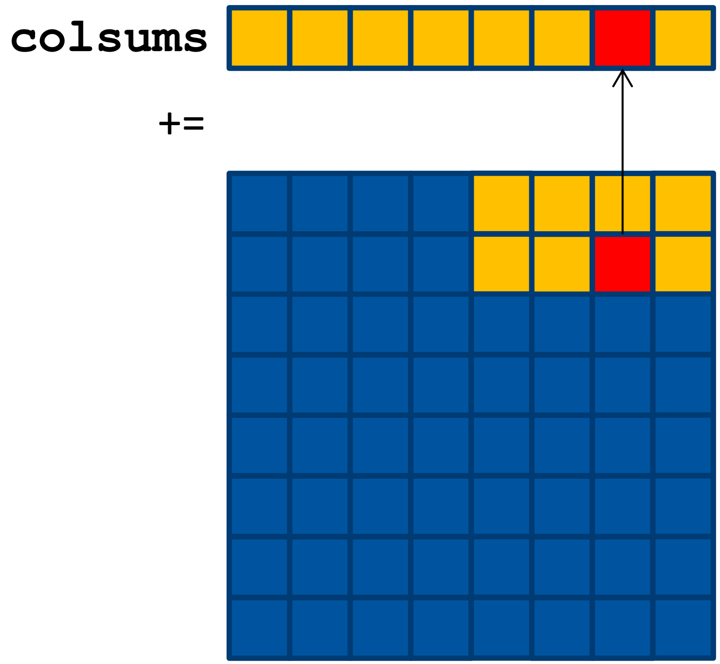
1. Norm calculation of a matrix (optional) norm1_blocking()



1. Norm calculation of a matrix (optional) norm1_blocking()

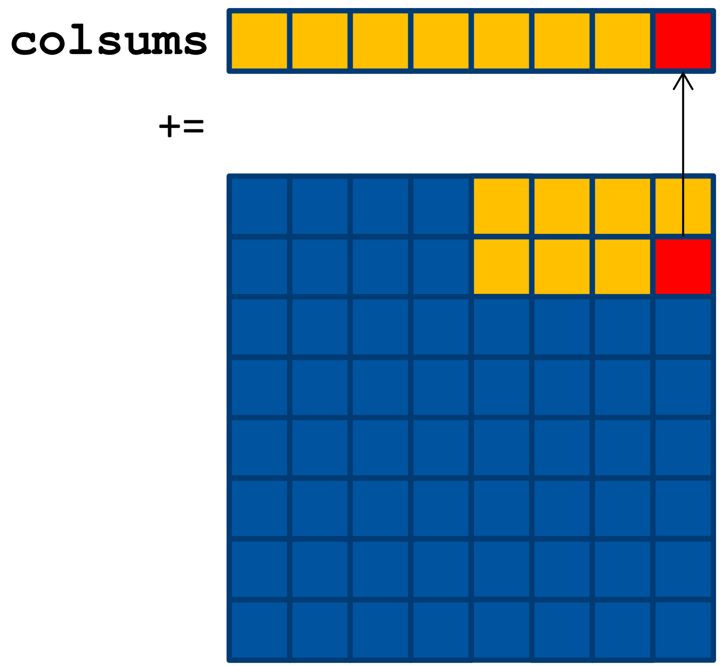


1. Norm calculation of a matrix (optional) norm1_blocking()

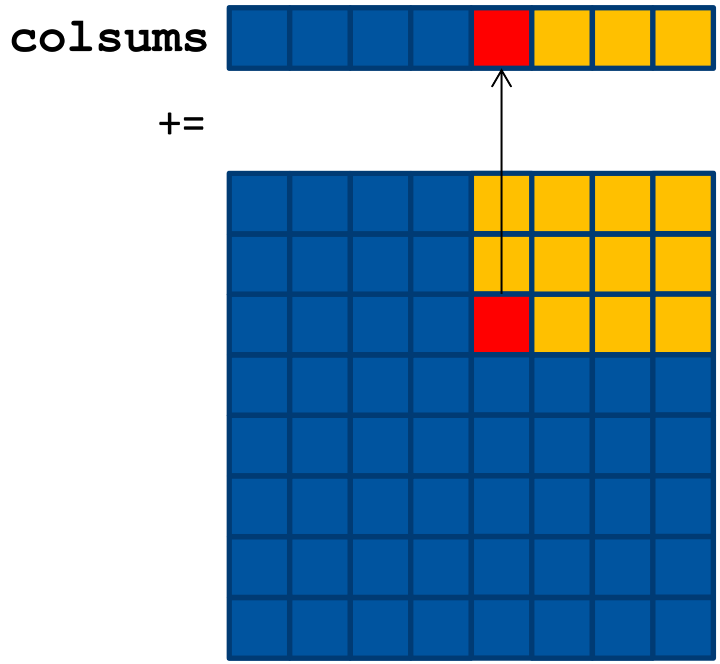


Exercise 02
Julian Miller | IT Center der RWTH Aachen University

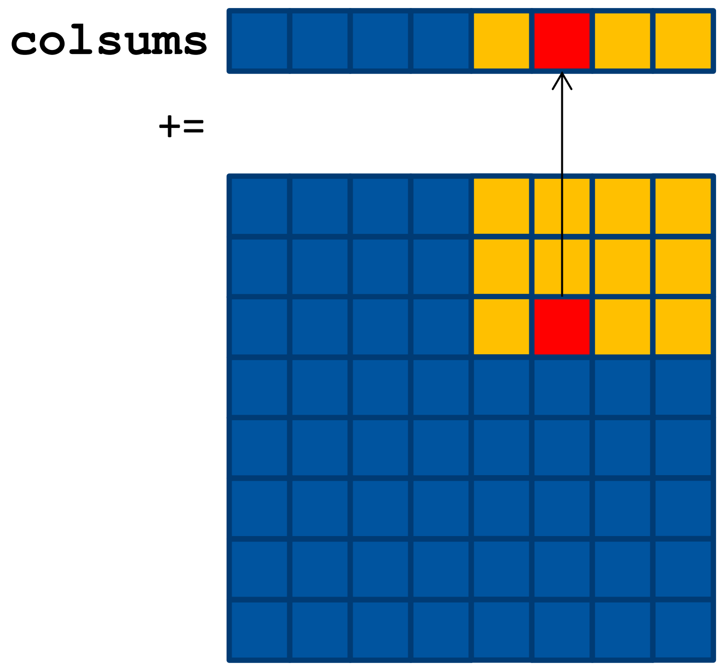
1. Norm calculation of a matrix (optional) norm1_blocking()



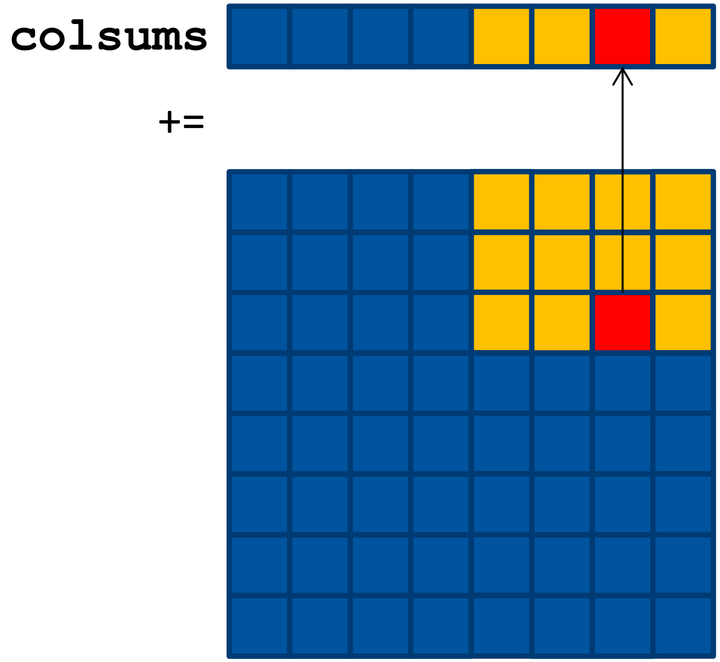
1. Norm calculation of a matrix (optional) norm1_blocking()



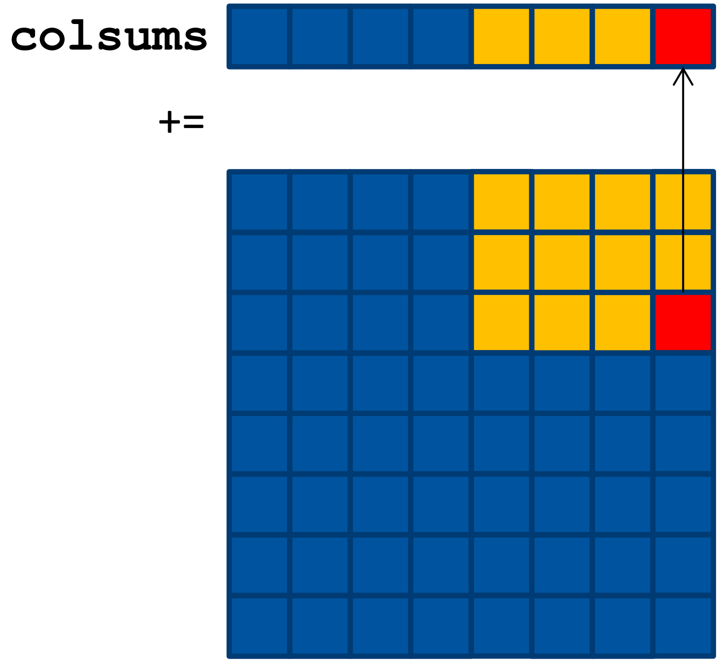
1. Norm calculation of a matrix (optional) norm1_blocking()



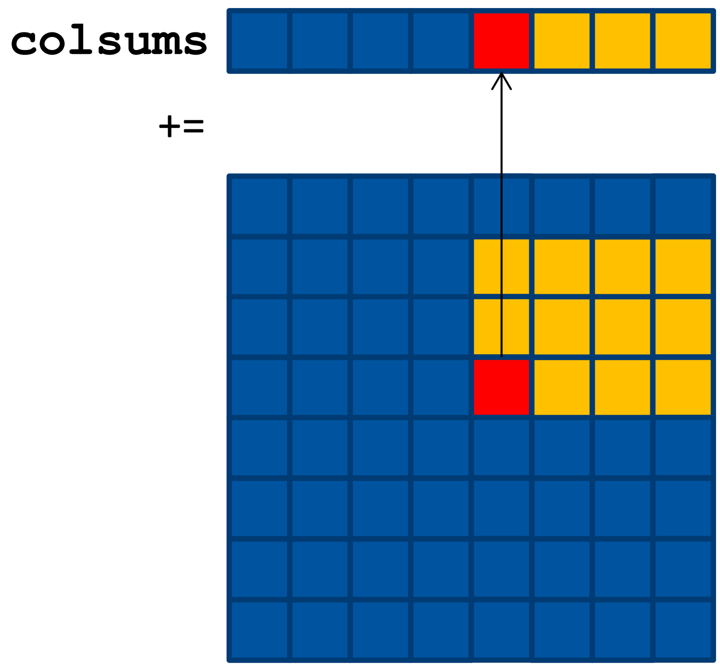
1. Norm calculation of a matrix (optional) norm1_blocking()



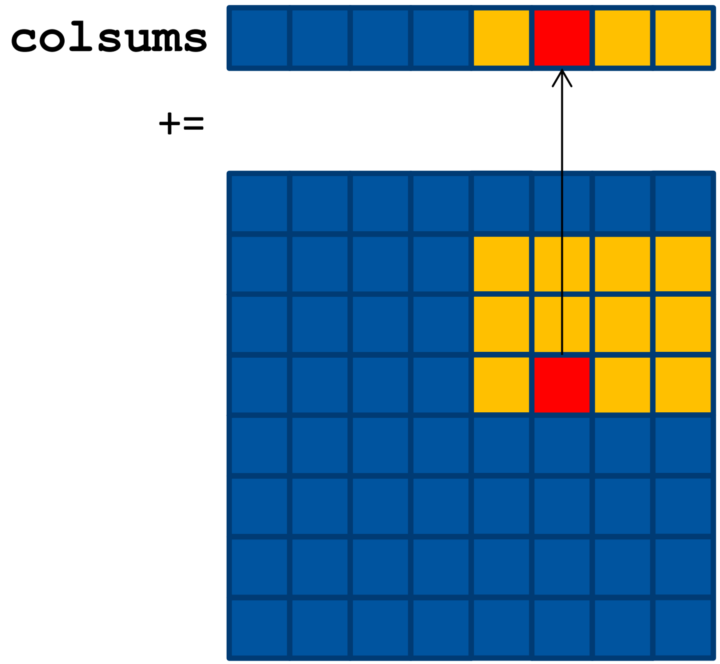
1. Norm calculation of a matrix (optional) norm1_blocking()



1. Norm calculation of a matrix (optional) norm1_blocking()



1. Norm calculation of a matrix (optional) norm1_blocking()



1. Norm calculation of a matrix (optional) norm1_blocking()

```
double norm1_block(double** const A, const int n, const int l){
    double *colsums = new double[n]; double max_norm = -1.;

/* TODO Copy the initialization of the auxiliary array and
computation of the column sums. Implement the blocking with a
surrounding for loop. */
    for (int c = 0; c < n; c += l) {
        int runto = std::min(c + l, n);
        for (int j = c; j < runto; ++j)
            colsums[j] = 0.;
        for (int i = 0; i < n; ++i)
            for (int j = c; j < runto; ++j)
                colsums[j] += abs(A[i][j]);
    }
...
```

1. Norm calculation of a matrix (optional) norm1_blocking()

```
...

/* TODO Find the largest column sum */
    for (int i = 0; i < n; ++i)
        if (colsums[i] > max_norm)
            max_norm = colsums[i];

    delete[] colsums;
    return max_norm;
}
```

Exercise Tasks

1. Norm calculation of a matrix
2. **Performance analysis tools**
3. Balance Metric

2. Performance analysis tools

Problem 2. Performance analysis tools
If you decided to solve Problem 1 by pen and paper, use the provided binaries in the `binary` directory in `exercise2.tar.gz`. Otherwise use your own code.

Problem 2.1. Profiling with gprof
You may run `make gprof` or execute `binary/norm.gprof.exe` and inspect the result with `gprof binary/norm.gprof.exe | less`
What is the time for the various algorithms according to gprof?
Try the target gprof2: `make gprof2` or execute `binary/norm.gprof2.exe` and inspect the result with `gprof binary/norm.gprof2.exe | less`
What is the difference of the outputs (You may have a look at the gprof targets in the Makefile)? What is the calculation time for the algorithms?

Livedemo

2.2 Hardware Counter

Problem 2.2. Hardware Performance Counter
Modern processor architectures have special-purpose registers to store the counts of hardware-related activities. These so called hardware performance counters can be used for low-level performance analysis or tuning. For accessing the hardware performance counters you can use the tool likwid[1] which is installed on the RWTH Compute Cluster.
Note 1: For this task you can use your own implementation of a), c), and d) from the exercise *norm calculation of a matrix* and build them with the Intel compiler (`make single` in the provided template) or use the provided binaries (norm_max.exe, norm_1_col.exe, and norm_1_row.exe).
Note 2: Use the special tuning node `login18-t.hpc.itc.rwth-aachen.de` for your measurements. Make sure that you are the only user on the system (e.g., with the `w` command) to obtain reliable results.

1. Examine the different programs with hardware counters using `likwid-perfctr`:
   ```
   module load likwid
   likwid-perfctr -C <processor id> -g <performance group> norm_max.exe
   ```
   Use the performance group `MEM_DP` for your investigations.

2.2 Hardware Counter

- **HW performance counter: Special purpose registers for the count of HW events**
- **Can be used for low-level performance analysis or tuning**
- **The tool Likwid can be used**
  - → Counters might be hard to interpret
  - → Likwid uses derived metric (e.g., memory bandwidth, data volume, MFlop/s)
  - → Use login18-t for your tests

Ignore the warning that you are not in the likwid group.
Using likwid like described should work nevertheless.
If not → mail to contact@hpc

2.2 Hardware Counter
1) Using `likwid-perfctr`

- **Load the likwid module**

```
$ module load likwid
```

- **Pin the program to one processor and examine the hardware counters of this processor with likwid**

```
$ likwid-perfctr -C <processor id> -g <performance group>
norm_max.exe
```

- **Use the performance group `MEM_DP`**

- **`N=32768`**
  → `$ likwid-perfctr -C 6 -g MEM_DP norm_max.exe 32768`

2.2 Hardware Counter
1) Using `likwid-perfctr`

- **How does this look like?**
  - → Part I: Counters

```
--------------------------------------------------------------------------------
Group 1: MEM_DP
+------------------------------------------+---------+--------------+
|                   Event                  | Counter |    Core 6    |
+------------------------------------------+---------+--------------+
|             INSTR_RETIRED_ANY            |  FIXC0  |   4181301685 |
|           CPU_CLK_UNHALTED_CORE          |  FIXC1  |  13407652411 |
|           CPU_CLK_UNHALTED_REF           |  FIXC2  |   7798376880 |
|              PWR_PKG_ENERGY              |   PWR0  |     306.8376 |
|              PWR_DRAM_ENERGY             |   PWR3  |      83.8334 |
| FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE |   PMC0  |       327680 |
|    FP_ARITH_INST_RETIRED_SCALAR_DOUBLE   |   PMC1  |      1146938 |
| FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE |   PMC2  |   1342504960 |
| FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE |   PMC3  |            0 |
|               CAS_COUNT_RD               | MBOX0C0 |    230072867 |
|               CAS_COUNT_WR               | MBOX0C1 |     49141767 |
|               CAS_COUNT_RD               | MBOX1C0 |    230072655 |
|               CAS_COUNT_WR               | MBOX1C1 |     49142741 |
|               CAS_COUNT_RD               | MBOX2C0 |    230070771 |
|               CAS_COUNT_WR               | MBOX2C1 |     49142542 |
|               CAS_COUNT_RD               | MBOX3C0 |        20616 |
|               CAS_COUNT_WR               | MBOX3C1 |        21615 |
|               CAS_COUNT_RD               | MBOX4C0 |        20943 |
|               CAS_COUNT_WR               | MBOX4C1 |        21603 |
|               CAS_COUNT_RD               | MBOX5C0 |        21311 |
|               CAS_COUNT_WR               | MBOX5C1 |        21625 |
+------------------------------------------+---------+--------------+
```

2.2 Hardware Counter
1) Using `likwid-perfctr`

- **How does this look like?**
  - → Part II: Metrics

```
+-----------------------------------+------------+
|              Metric               |   Core 6   |
+-----------------------------------+------------+
|        Runtime (RDTSC) [s]        |     5.3026 |
|        Runtime unhalted [s]       |     6.4029 |
|            Clock [MHz]            |  3600.1816 |
|                CPI                |     3.2066 |
|             Energy [J]            |   306.8376 |
|             Power [W]             |    57.8650 |
|          Energy DRAM [J]          |    83.8334 |
|           Power DRAM [W]          |    15.8097 |
|             DP MFLOP/s            |  1013.0461 |
|           AVX DP MFLOP/s          |  1012.7062 |
|           Packed MUOPS/s          |   253.2383 |
|           Scalar MUOPS/s          |     0.2163 |
|  Memory read bandwidth [MBytes/s] |  8331.2911 |
|  Memory read data volume [GBytes] |    44.1779 |
| Memory write bandwidth [MBytes/s] |  1780.1463 |
| Memory write data volume [GBytes] |     9.4395 |
|    Memory bandwidth [MBytes/s]    | 10111.4375 |
|    Memory data volume [GBytes]    |    53.6173 |
|       Operational intensity       |     0.1002 |
+-----------------------------------+------------+
```

2.2 Hardware Counter

2. Analysis of the results. Likwid derives different metric from the counted events. Answer the following questions:

   - Let be $n = 2^{15} = 32768$. Determine the read date volume D_{read} for the computation of $\|A\|_\infty$? What is reported by likwid? Why do these values differ?
   - Which floating point performance (in `MFLOP/s`) is reported by likwid? Why is it not as high as reported by the norm application?
   - Which memory bandwidth is reported by likwid? How can you assess whether the value is reasonable?

2.2 Hardware Counter

- **Let be n = 32768. Determine the read data volume D_{read} for the computation of $\|A\|_\infty$**
  - → D_{read} = n∗n ∗ 8 Byte = 8.59 GB
- **What is reported by likwid?**
  - → 44.18 GB
- **Why do these values differ?**
  - → Code in main():

```
for (int i = 0; i < num_tests; ++i) {
    n1 = norm_max(A, n);
}
```

  - → Computation is executed num_tests = 5 times.
    - → 5 * 8.59 GB = 42.95 GB
    - → Note: Some compiler (e.g., clang) optimize the code such that the loop is only executed once (might lead to wrong interpretation of the performance)

Exercise 02
Julian Miller | IT Center der RWTH Aachen University

2.2 Hardware Counter

- **Which floating point performance is reported by likwid?**
  - → 1012.87 Mflop/s
- **Why is it not as reported by the norm application?**
  - → Application reports 1650 Mflop/s
  - → Reason: likwid measures the complete application (including matrix allocation, initialization and error checking)
  - → Solution 1: Use likwid marker API to only measure the kernel
  - → Solution 2: Make more repetition (num_tests variable) to make kernel more dominant
    - →num_tests = 100: 1594.87 Mflops/s

2.2 Hardware Counter

- **Which memory bandwidth is reported by likwid?**
  - → 10110.85 MB/s
- **How can you assess whether the value is reasonable?**
  - → STREAM benchmark for one thread: 13842.9 MB/s (Triade)
  - → Not optimal, but reasonable
  - → Measured for complete application (including matrix allocation, initialization and error checking)
  - → Value for num_tests = 100: 12896.04 MB/s

Exercise Tasks

1. Norm calculation of a matrix
2. Performance analysis tools
3. **Balance Metric**

3. Performance Modeling

- **Observation:**
  We reached a performance of 1.6 Gflop/s on a given architecture.

- **Question:**
  Is that a good or a bad performance?

- **Answer:**
  Can be given an adequate performance model.

3.1 Machine Balance

- **Reference machine: 2-socket Intel Skylake CPU**
  - → 48 cores in total
  - → 2.1 GHz (base) clock frequency
  - → AVX-512 registers (512 bits)
  - → 2 operations per cycle (FMA)
  - → 2 FMA units per core
  - → Cache sizes: L1 → 32 KiB, L2 → 1 MiB, L3 → 33 MiB
  - → Sustainable main mem. bandwidth gained by Stream benchmark

See lecture slide Part 4, page 12 f.

- **Compute the machine balance B_m of this architecture (doubles).**

- **Solution**
  - → $B_m = \frac{b_s}{P_{max}}$

b_s:	achievable bandwidth over slowest data path [words/s]
P_{max}:	peak amount of floating-point operations per seconds [Flop/s]

3.1 Machine Balance

- **Achievable bandwidth [words/s]**

  → $b_s = \frac{STREAM\ BANDWIDTH}{8\ B} = \frac{170\ GB/s}{8\ B} = 21.25\ GWords/s$

- **Peak amount of floating-point operations**

  → $P_{max} = \#Cores\ *\ Frequency\ *\ SIMD\_OPs$

  → $SIMD\_OPs = 2\ *\ 2\ *\ 8\ Flop = 32\ Flops$

  →2 operations (multiply + add)

  →8 doubles per AVX-512 register (512 bit / 64 bit = 8, length of double: 8 Byte)

  →2 FMAs

b_s:	achievable bandwidth over slowest data path [words/s]
P_{max}:	peak amount of floating-point operations per seconds [Flop/s]

- **Solution**

  → $B_m = \frac{b_s}{P_{max}}$

$$= \frac{\frac{170\ GB/s}{8\ B}}{48\ \cdot 2.1\ GHz \cdot 32\ Flop} = \frac{21.25\ GWords/s}{3225.6\ GFlop/s} \approx 0.0067\ \frac{Words}{Flop}$$

3.1 Machine Balance

- **With respect to the norm calculation of a matrix, which performance limits could be adapted?**

b_s:	achievable bandwidth over slowest data path [words/s]
P_{max}:	peak amount of floating-point operations per seconds [Flop/s]

- **Solution**

→ $\boldsymbol{B_m = \frac{b_s}{P_{max}}}$

→ The norm matrix computation is a serial application!

→ Previously computed P_{max} and b_s can never be reached

→ Use single-core values for new B_m:

See lecture slide Part 4, page 12 f.

$$B_m = \frac{\frac{14\ GB/s}{8\ B}}{1\ \cdot 2.1 \cdot 32\ GFlop/s} = \frac{1.75\ GWords/s}{67.2\ GFlop/s} \approx 0.026\frac{Words}{Flop}$$

→ Use this performance limit in the following

3.2 Code Balance

See lecture slide Part 4, page 14 f.

- **Compute the code balance B_c of the $|A|_\infty$ computation.**
- **Assumptions**
  - → Access to main memory is the slowest data path
  - → Floating-point comparison (a < b?) = 1 is NOT a floating-point operation
  - → Ignore the (statistical) sign flipping in the abs-function
- **Which elements are located in registers, caches and main memory?**

3.2 Code Balance

- $\|A\|_\infty$ computation

1 FP operations

1 load
(A needs 32,768²*8 B =8 GB in memory, rowsum in register)

```
double norm_max(double** const A, const int n)
{   double rowsum=0., max_norm=-1.;
    for (int i=0; i<n; ++i)
    {   rowsum=0.;
        for (int j=0; j<n; ++j)
            rowsum += abs(A[i][j]);
        if (rowsum>max_norm)
            max_norm=rowsum;
    }
    return max_norm;
}
```

Ignore comparison

```
double abs(double x) {
   return x<0 ? -x : x;
}
```

- **Solution**

→ Code balance: $B_c = \frac{data\ transfers(LOAD,STORE)}{arithmetic\ operations} \frac{[Words]}{[Flop]}$

→ $B_c = \frac{1}{1} \frac{Word}{Flop}$

3.3 Lightspeed

- **Compute the (relative) lightspeed *l* of the $|A|_\infty$ computation on an Intel Skylake architecture**
  → How do you interpret this value?
  → In general, how can a lightspeed value be improved by the application developer?

See lecture slide Part 4, page 17 f.

- **Solution**
  → Lightspeed $\boldsymbol{l = min(1, \frac{B_m}{B_c})}$ with $\boldsymbol{B_m = 0.026\,\frac{Words}{Flop}}$, $\boldsymbol{B_c = 1\,\frac{Word}{Flop}}$
  → $\boldsymbol{l = min(1, \frac{0.026}{1}) = 0.026}$
  → 2.6 % of machine's peak performance possible for this code
  → Machine balance cannot be influenced; only code balance can be influenced by improving the transfer/computation ratio (e.g. by better cache usage)

3.3 Lightspeed

- **Compute the lightspeed *P* for absolute performance in GFlop/s!**

- **Solution**

$\rightarrow P = l \cdot P_{max} = min\left(P_{max}, \frac{b_s}{B_c}\right)$ with $b_s = 1.75\ \frac{G\ Words}{s},\quad B_c = 1\ \frac{Word}{Flop}$

$\rightarrow P = min(67.2\ GFlop/s,\ \frac{1.75\ G\ words/s}{1\ word/Flop})$

$= min(67.2\ GFlop/s,\ 1.75\ GFlop/s) = 1.75\ GFlop/s$

3.3 Lightspeed

- **Compare the measured performance [GFlop/s] (from the previously-done likwid VIEW run) with the computed theoretical performance**
  - → How close is the code to what it can reach at maximum?
  - → Why can there be a difference between theoretical and experimentally-gathered results?
- **Solution**
  - → Theoretical performance limit: $\boldsymbol{P = 1.75\ GFlop/s}$
  - → Performance analysis: ~ 1.6 GFlops/s
  - → Balance metric uses certain assumptions (that usually cannot hold completely), e.g.
    - → Data transfer & arithmetical operations overlap perfectly
    - → Perfect use of FMA (here no multiplications needed, not relevant if memory bound)
    - → Only the slowest data path is/ can be modeled (others are infinitely fast)
    - → Latency is ignored

Real code achieves 91% of theoretical peak performance of the code

Exercise 02
Julian Miller | IT Center der RWTH Aachen University