Serial Performance

PPCES 2019

Tim Cramer

12.03.2019

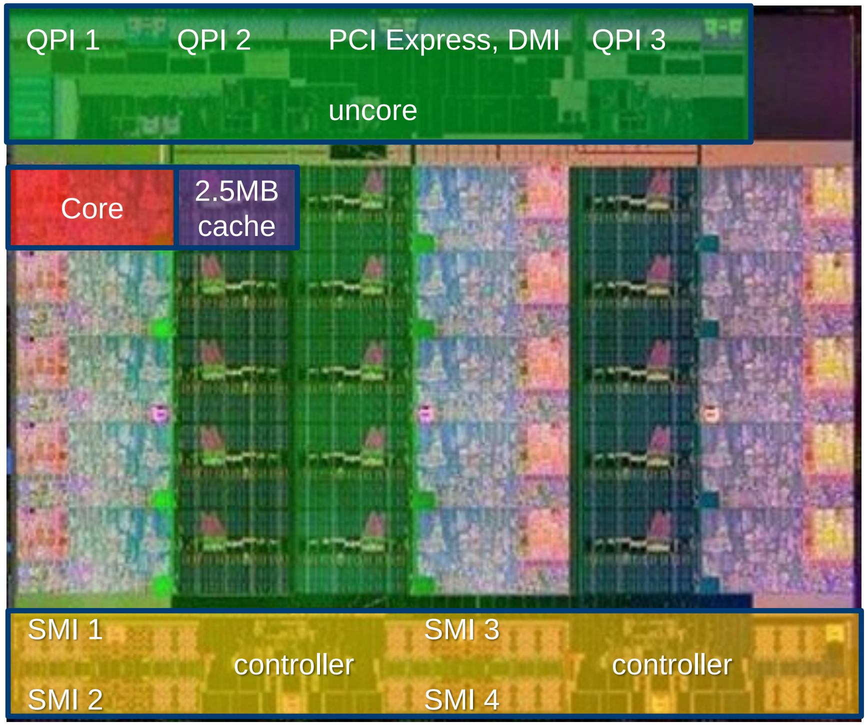
Really? Serial Performance?

- **Question:**

  Why do you guy talk about serial performance in a parallel computing course?

- **Answer:**

  Because it matters. **(as we will see…)**

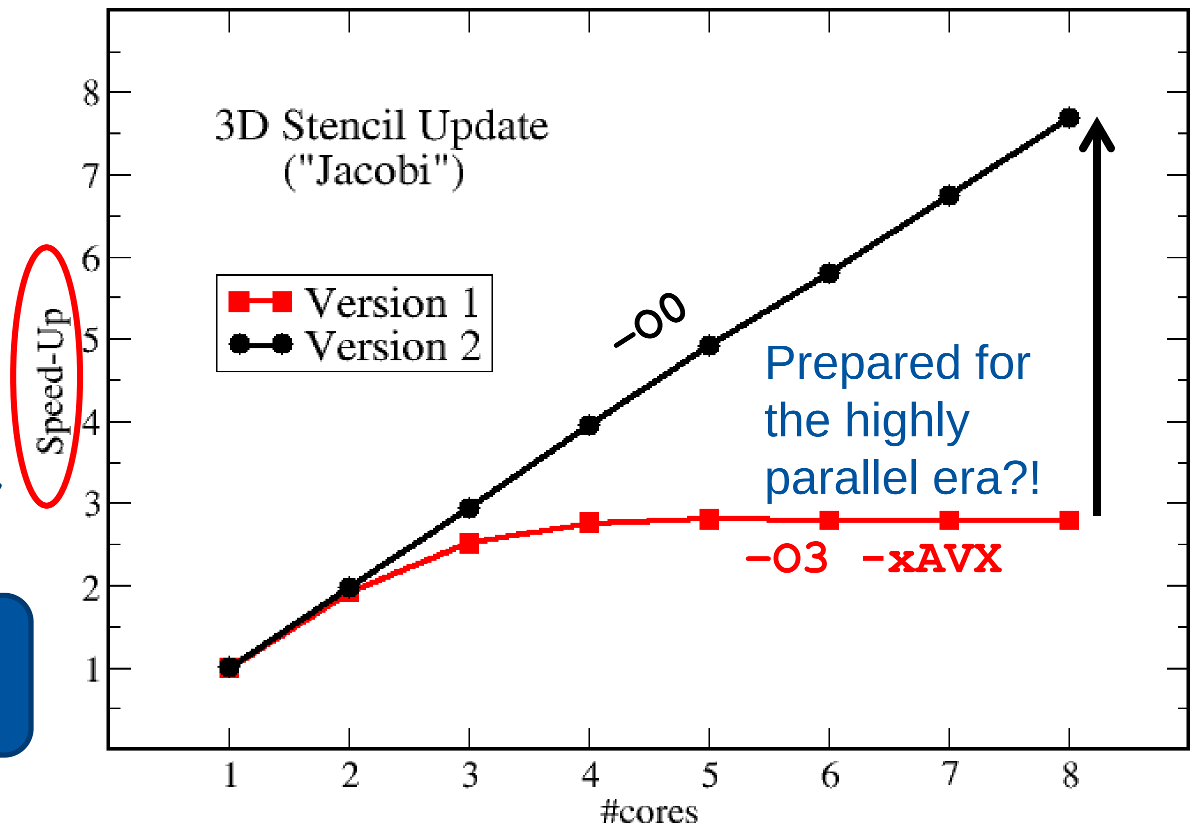


Scalability Myth: Code scalability is the key issue

parallel program

Changing only the compile options makes this code scalable on an 8-core chip

Parallel program is X times faster than serial program.

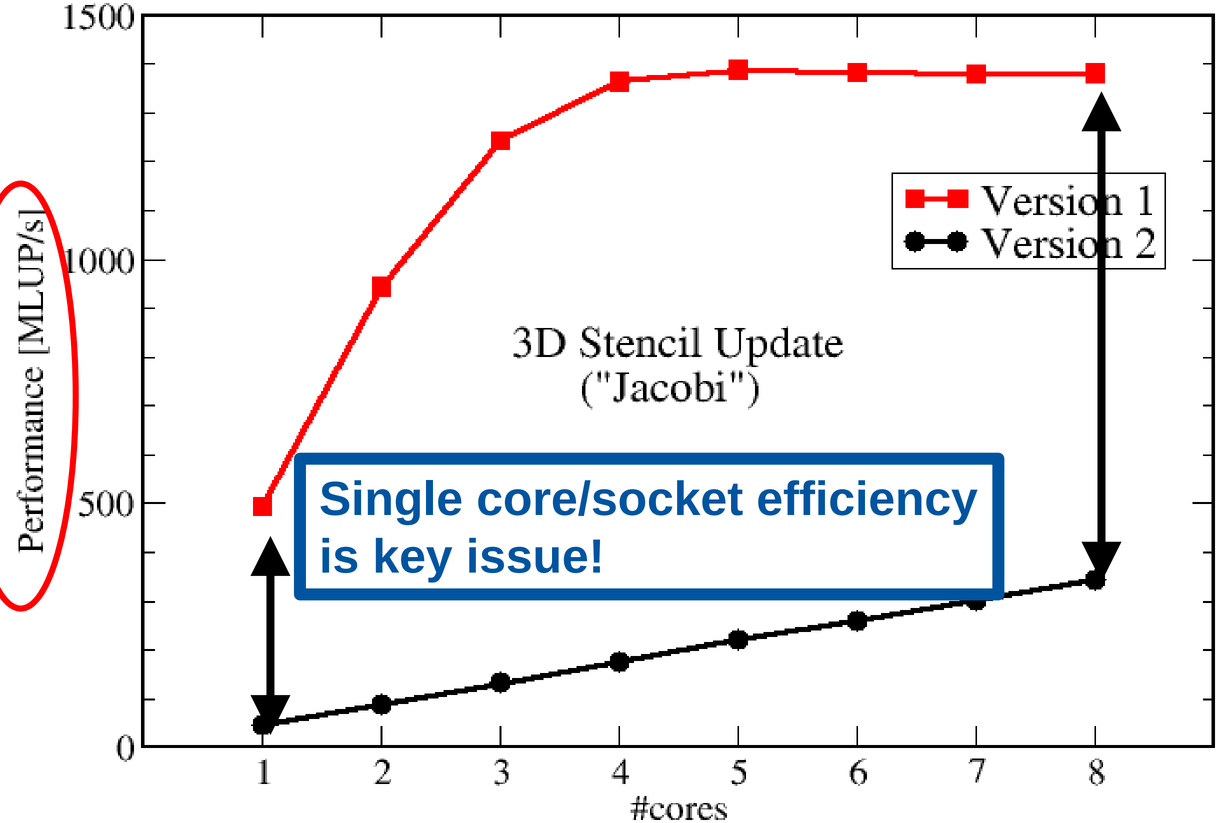


Courtesy of Erlangen Regional Computing Center (RRZE)

Scalability Myth: Code scalability is the key issue

parallel program

Absolute performance



Single core/socket efficiency is key issue!

Courtesy of Erlangen Regional Computing Center (RRZE)

Contents

- **Improving Serial Performance**
  - → General serial performance optimizations
    - → Compiler optimization
    - → Examples for code optimization
  - → Memory Access
    - → Calculation of cache-optimized matrix norms

Contents

- **Improving Serial Performance**
  - → General serial performance optimizations
    - → Compiler optimization
    - → Examples for code optimization
  - → Memory Access
    - → Calculation of cache-optimized matrix norms

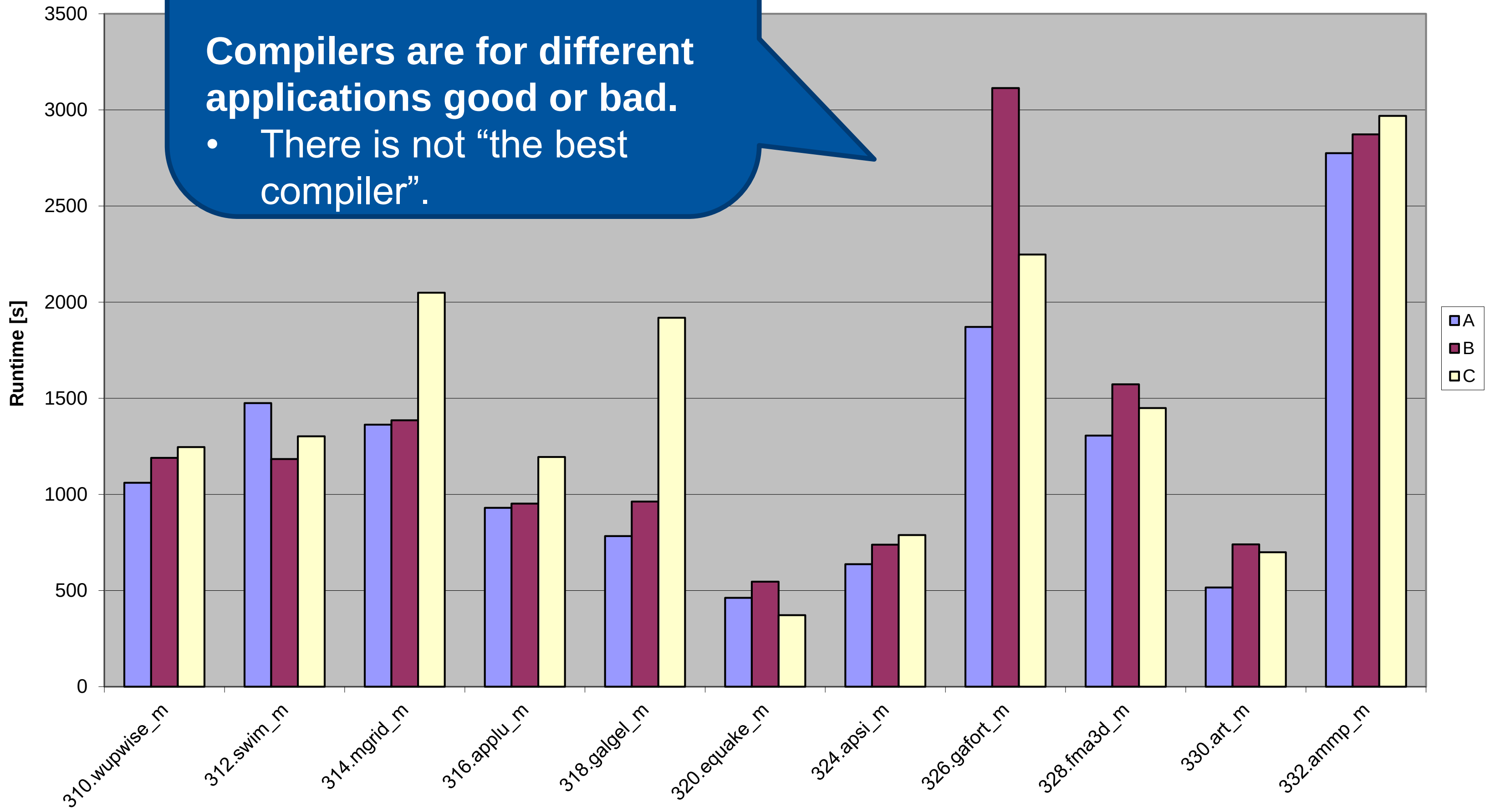
Compiler impact

- **Compiler option can have a great impact on program performance**
  - → Compiler based optimization has varying influence on code, but are in general beneficial
- **Every modern compiler has command line switches to enable or disable certain optimization patterns**
- **Check different compilers for more performance potential**
- **Do not rely on the compiler to identify every optimization potential**
  - → Design your code in the most economical way in terms of performance
- **Compilers can be surprisingly intelligent and foolish at the same time**

Compiler comparison with SPEC OMPM2001

Different compilers can deliver different performances.
- Compiler switch may help.

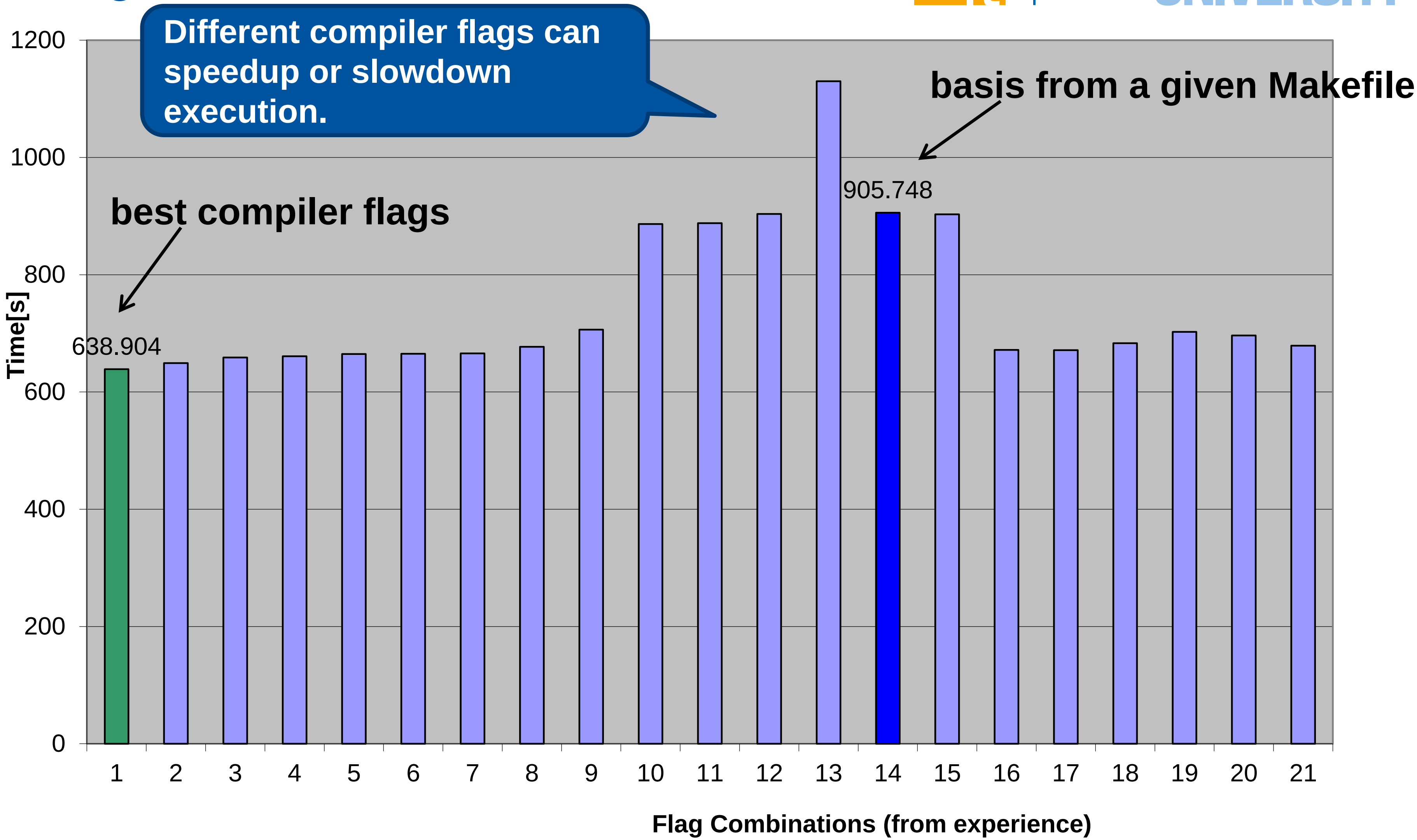
Compilers are for different applications good or bad.
- There is not “the best compiler”.



General compiler optimization levels

- **Compilers offers a set of standard optimization options**
  - → -O0, -O1, .., -O3, …
  - → Not standardized → See the compiler manual for further information
- **For debugging purposes compile with –O0 (no optimizations at all)**
  - → Higher level optimizations may include mixing of source lines, elimination of redundant variables and operations, rearrangement of arithmetic expressions
  - → Debugging is difficult if code is optimized
- **Modern architectures have wider registers**
  - → Use flags to ensure vectorization
  - → Example Intel Compiler: `-x<code>`. "`code`" indicates a feature set, e.g.: `SSE2, SSE3, SSE4.2, AVX, CORE-AVX2, COREAVX512` (now program runs only on hardware with the chosen feature set)
  - → Use `-ax<code>` to include alternative code into executable In
- **In our environment we offer flags like `$FLAGS_FAST`. You can use this flags to build your program (use module system!)**

Performance impact of compiler flags



Source: Ulf Markwardt, TU Dresden

Compiler Optimization Reports

- → Compilers offer options to generate annotated source code listings or logs that describe the optimizations that could be applied in more detail
- → Supplies additional information about register usage and spilling, superscalar operations, pipeline utilization and speculative executions
- → Example: Intel Compiler

  `-qopt-report[=n]`

  - → This option tells the compiler to generate a collection of optimization report files, one per object (i.e., file)
  - → `n` is detail degree (should be between 0 and 5)

Library of numerical Routines (LAPACK, …, FFT, … Intel MKL)

- **If a code uses some numerical routines, you may find this routines in BLAS, LAPACK, etc.**
- **Several libraries implement this numerical routines**
  - → Optimized for Intel processors: Intel Math Kernel Library
  - → Optimized for AMD processors: AMD Core Math Library
- **Depending on the hardware platform you can link the program against one of the libraries**

Contents

- **Improving Serial Performance**
  - → General serial performance optimizations
    - → Compiler optimization
    - → Examples for code optimization
  - → Memory Access
    - → Calculation of cache-optimized matrix norms

Do less work – Simple loops

- **Do less work**
  - → Rearrange code such that less work than before is being done
- **Many programs can benefit from small code changes that despite their trivial complexity can significantly increase performance**
- **Loop example**
  - → Often, programs do more work than required

C/C++

```
for(i=0; i<N; ++i)
{
  if( data[i] % 10 == 0 )
  {
    flag=true;
  }
}
```

Fortran

```
do i=1,N
  if(mod(data(i), 10)==0)
   flag=.true.
  end if
end do
```

Do less work – Simple loops

- If the condition induces no side effects, the loop may break after the flag got set to true the first time:

C/C++

```
for(i=0; i<N; ++i)
{
  if( data[i] % 10 == 0 )
  {
    flag=true;
    break;
  }
}
```

Fortran

```
do i=1,N
  if(mod(data(i), 10)==0)
   flag=.true.
   exit
  end if
end do
```

- If one other element in the dataset fits to the condition, it has no further effect since flag is already set to true. Therefore processing further elements is redundant and waste of computational resources

Avoid expensive operations

- **Some implementations just translate the formulas into code without respect to performance issues**
  - → Good, but often “expensive” operations (e.g. sin(x))
- **Performance optimization by replacing expensive operations by cheaper alternatives**
  - → Keep in mind that performance optimization bears the slight danger of changing numerics or even results
- **A common example:**

```
while(condition)
{
  [...]
  int x = (someval % 10);
  double s = sin(x);
}
```

Tabulating

- It can be profitable to consider e.g. the input range of expensive functions (such as trigonometric, e.g. sin, cos, tan, exp,…)
- Optimization technique is called tabulating

Table setup (executed once):

```
for(x = 0; x < 10; ++x)
{
  sin_table[x] = sin(x);
}
```

```
while(condition)
{
  […]
  int x = (someval % 10);
  double s = sin_table[x];
}
```

Tabulating

- **Table lookup is done at virtually no costs compared to the execution of the sine-function**
- **Lookup-tables can, depending on their size, fit into the L1 Cache and have very few CPU cycles of access time**
- **Tabulating can not be applied when the input range to function can not be isolated**

Eliminate common subexpressions

- **If parts of complex expressions can be precalculated, they should not be explicitly calculated in e.g. a loop construct**
- **In case of loops this optimization is called loop invariant code motion:**

```
for(i = 0; i < N; ++i)
{
  a[i] = a[i]+s+r*sin(x);
}
```

```
tmp = s+r*sin(x);
for(i = 0; i < N; ++i)
{
  a[i] = a[i]+tmp;
}
```

- **Compiler can detect this situation in principle**
  - → If the compiler needs to apply associativity rules it may refrain from doing so
  - → You may need to help the compiler

Avoid branches

- **Branches may prevent the compiler from applying loop unrolling or SIMD vectorization (especially in loops)**
  - → Avoid branches whenever possible
- **Processor does speculative execution**
  - → But mispredicted branches are costly
- **An example:**

```
for(i = 0; i < N; ++i)
{
  for(j = 0; j < N; ++j)
  {
    if(i < j)
      a[i][j] = a[i][j]+1;
    else if(i > j)
      a[i][j] = a[i][j]-1;
  }
}
```

j →

i ↓

–	+1	+1	+1	+1
-1	–	+1	+1	+1
-1	-1	–	+1	+1
-1	-1	-1	–	+1
-1	-1	-1	-1	–

Avoid branches

- In certain situations loop nests may be transformed so that all conditional statements vanish:

```
for(i = 0; i < N; ++i)
{
 for(j = 0; j < N; ++j)
 {
  if(i < j)
   a[i][j] = a[i][j]+1;
  else if(i > j)
   a[i][j] = a[i][j]-1;
 }
}
```

→

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

- Clearly the second variant has a bigger optimization potential

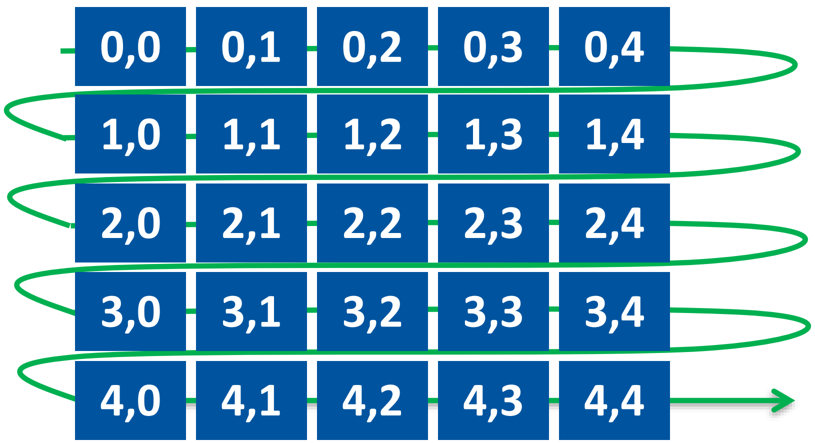
Contents

- **Improving Serial Performance**
  - → General serial performance optimizations
    - → Compiler optimization
    - → Examples for code optimization
  - → Memory Access
    - → Calculation of cache-optimized matrix norms

Memory Access

- **Cache: Fast (but small) buffers near the processor**
- **Caches are organized in “cache lines”**
- **Typically length of a cache lines: 64 byte (8 double precision values)**
- **Using for example one double value results in loading a whole cache line**
- **So it is profitable to really use all of the data of such a chunk once it resides in a cache (spatial locality)**
- **And it is even better to reuse data residing in a cache in a timely manner (temporal locality)**

Memory layout for C/C++



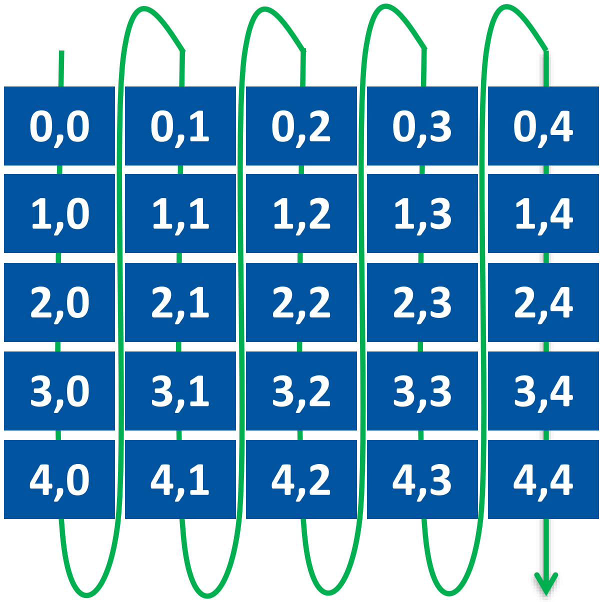
Row by Row Ordering or
row major ordering

Rows are stored consecutively
in memory.

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

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

Memory layout for Fortran



Column by Column Ordering or column major ordering

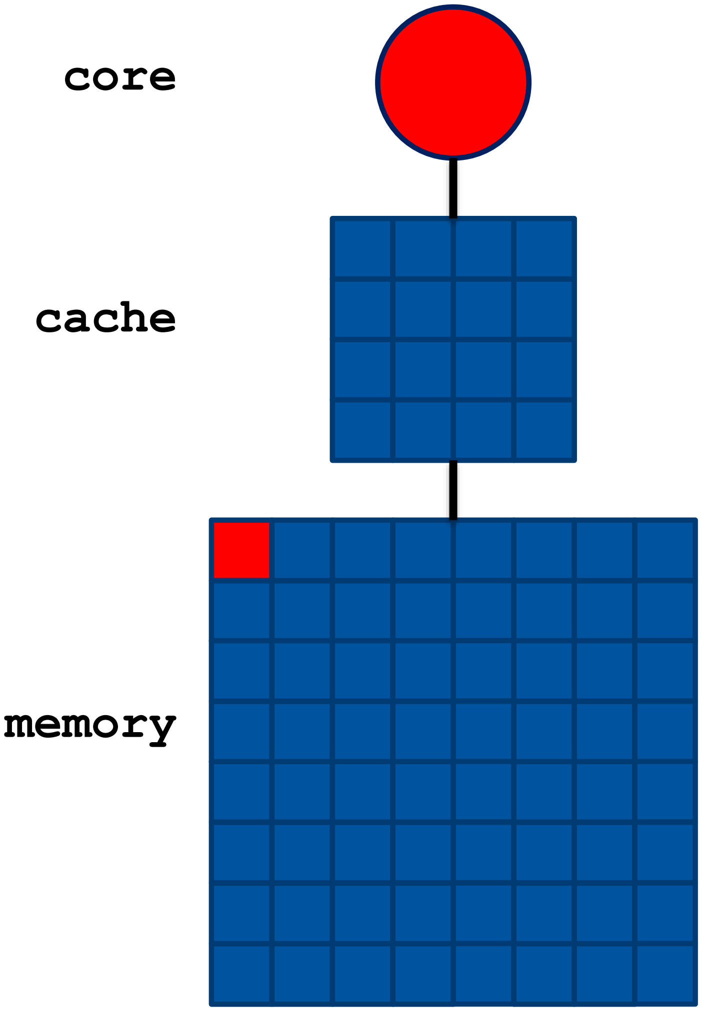
Columns are stored consecutively in memory

```
DO i=1,N
  DO j=1,N
    a(j,i) = i*j;
  END DO
END DO
```

```
DO j=1,N
  DO i=1,N
    a(j,i) = i*j;
  END DO
END DO
```

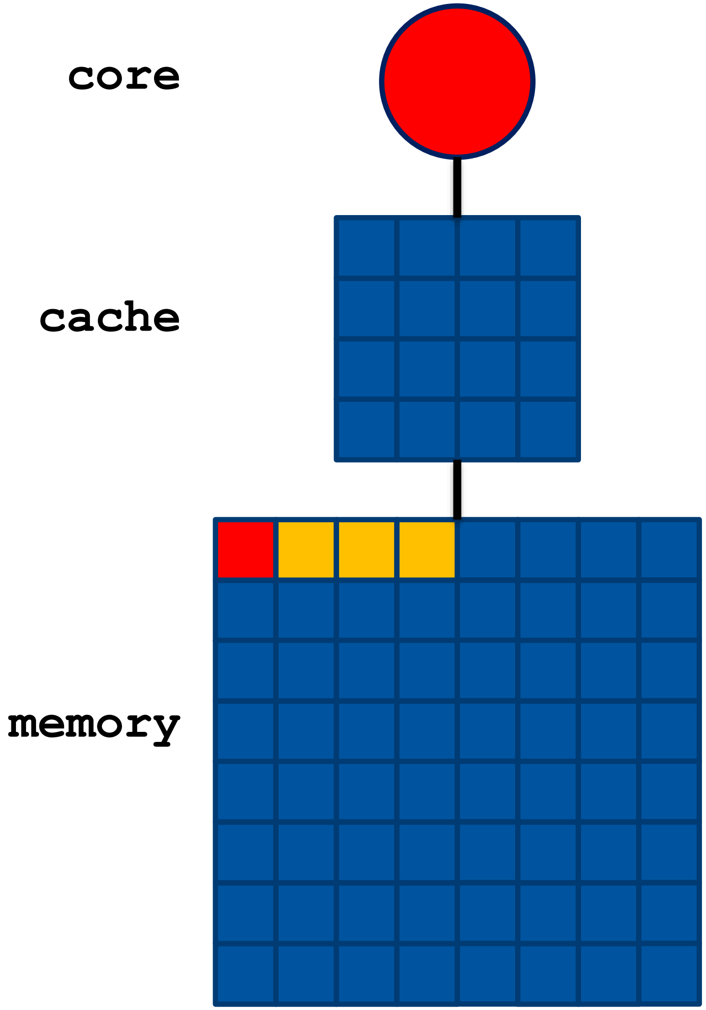
Memory Access

- **For reading or writing one element in the memory first one complete cache line must be loaded into the cache**



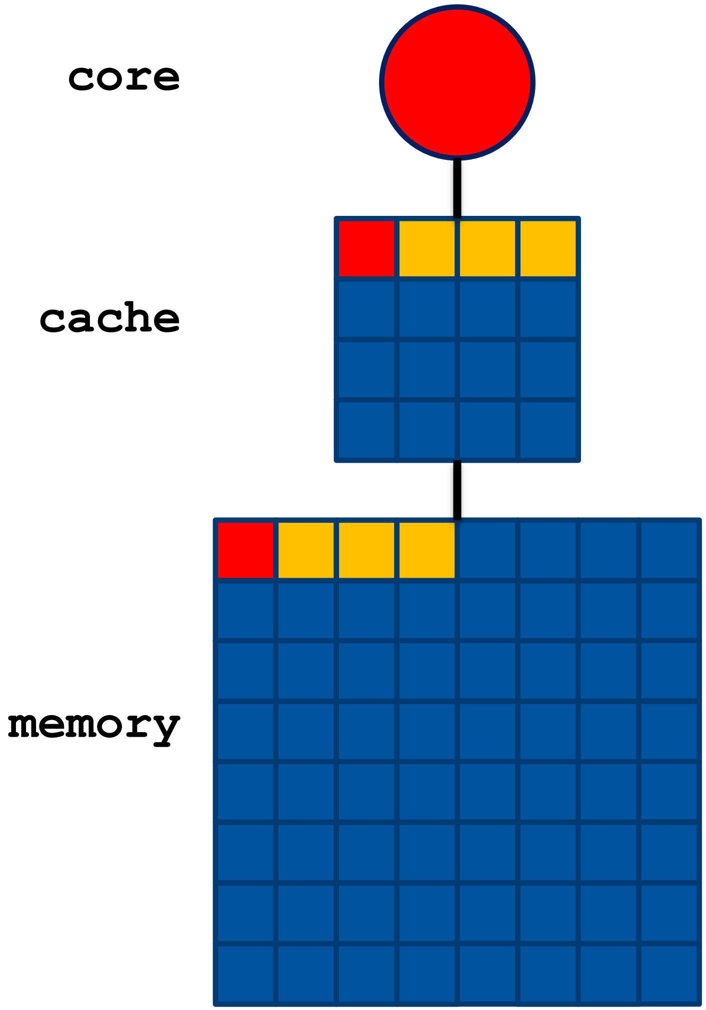
Memory Access

- **For reading or writing one element in the memory first one complete cache line must be loaded into the cache**



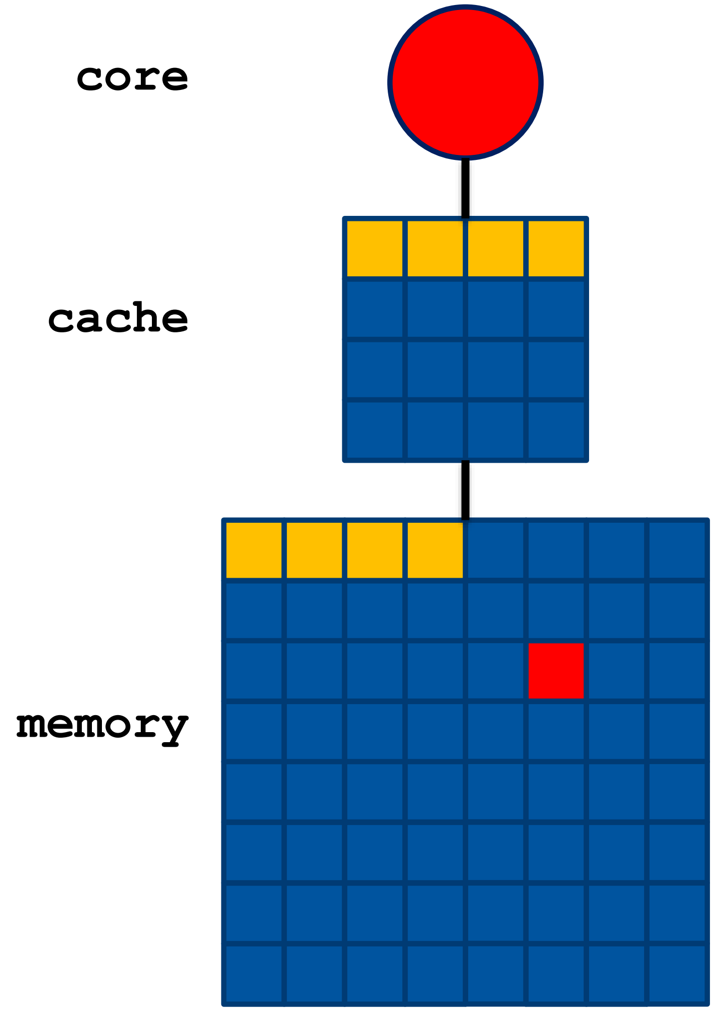
Memory Access

- **For reading or writing one element in the memory first one complete cache line must be loaded into the cache**
- **Now the processor can read and write the element**



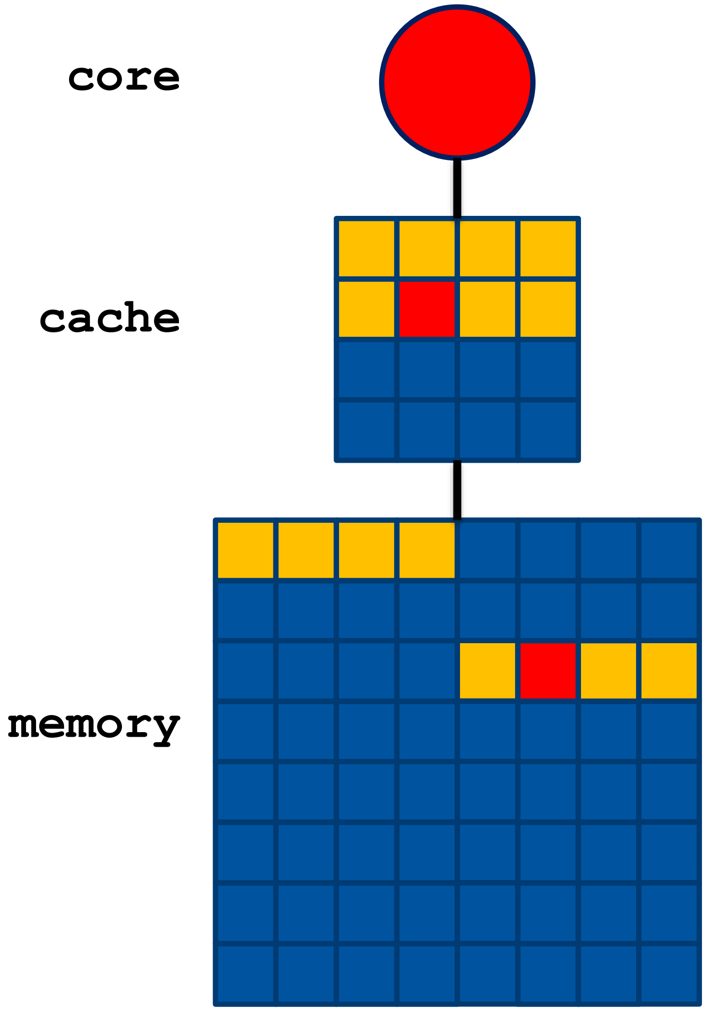
Memory Access

- **For reading or writing one element in the memory first one complete cache line must be loaded into the cache**
- **Now the processor can read and write the element**
- **If the next element is outside the loaded cache line a next one must be loaded to the cache**



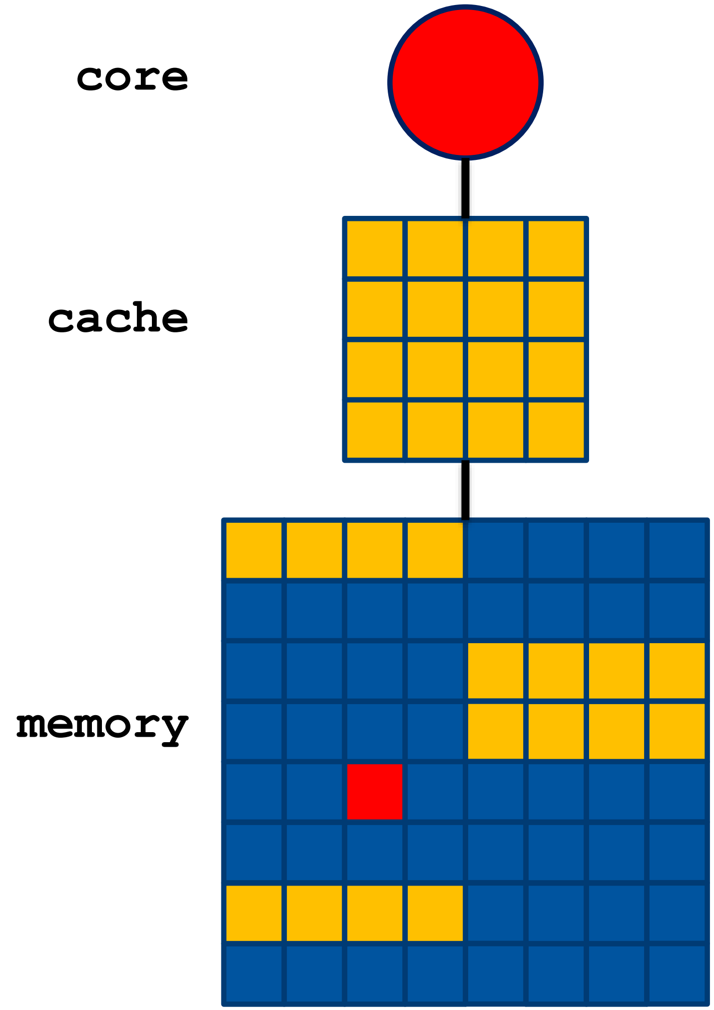
Memory Access

- For reading or writing one element in the memory first one complete cache line must be loaded into the cache
- Now the processor can read and write the element
- If the next element is outside the loaded cache line a next one must be loaded to the cache



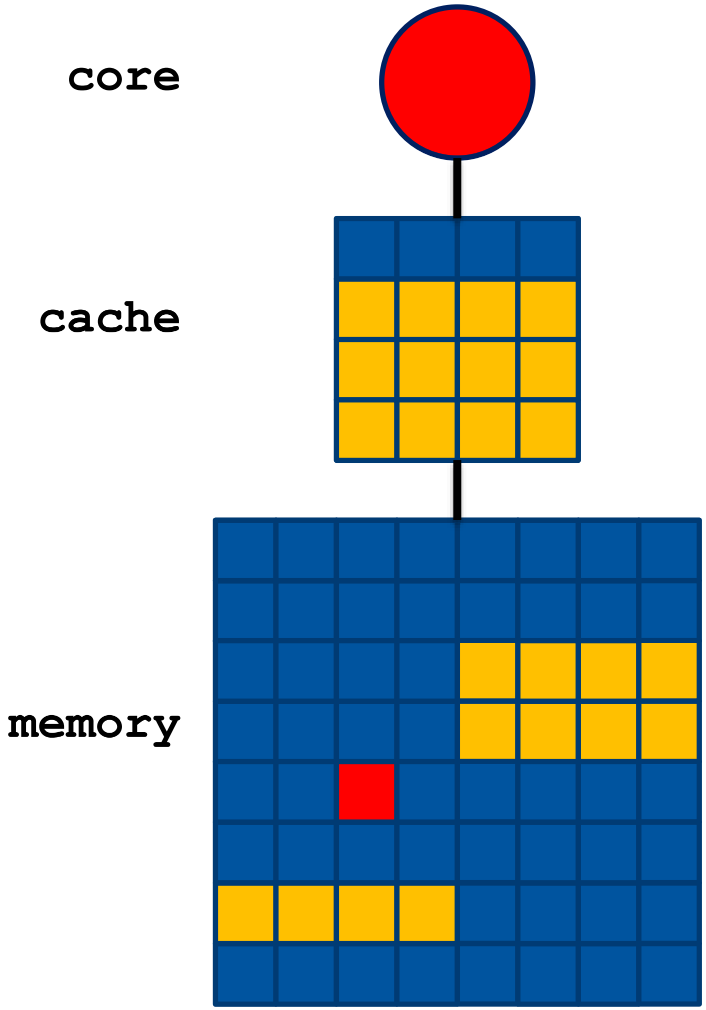
Memory Access

- For reading or writing one element in the memory first one complete cache line must be loaded into the cache
- Now the processor can read and write the element
- If the next element is outside the loaded cache line a next one must be loaded to the cache
- If the cache is full and a new cache line should be loaded an old one must be dropped



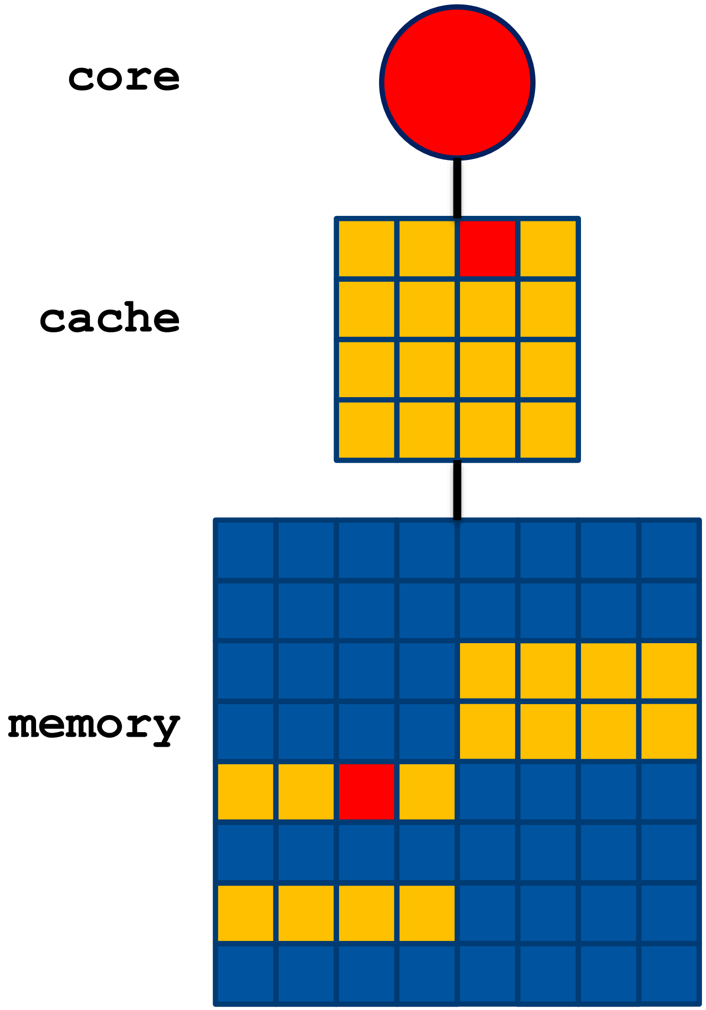
Memory Access

- **For reading or writing one element in the memory first one complete cache line must be loaded into the cache**
- **Now the processor can read and write the element**
- **If the next element is outside the loaded cache line a next one must be loaded to the cache**
- **If the cache is full and a new cache line should be loaded an old one must be dropped**



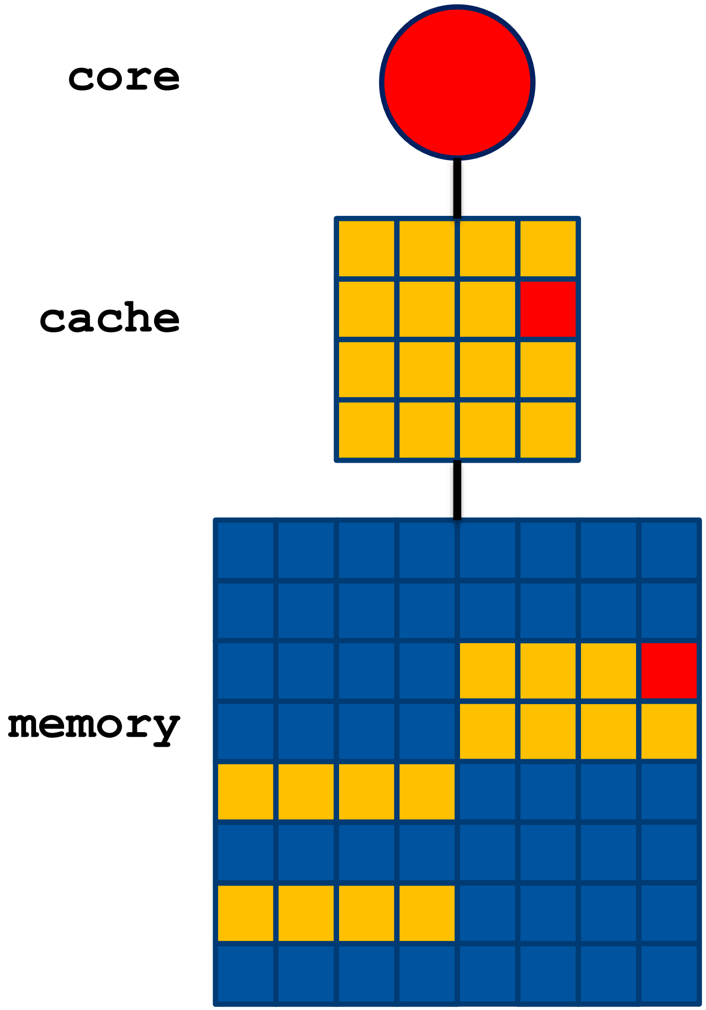
Memory Access

- For reading or writing one element in the memory first one complete cache line must be loaded into the cache
- Now the processor can read and write the element
- If the next element is outside the loaded cache line a next one must be loaded to the cache
- If the cache is full and a new cache line should be loaded, an old one must be dropped



Memory Access

- **For reading or writing one element in the memory first one complete cache line must be loaded into the cache**
- **Now the processor can read and write the element**
- **If the next element is outside the loaded cache line a next one must be loaded to the cache**
- **If the cache is full and a new cache line should be loaded, an old one must be dropped**
- **Accessing an element concerning to an already loaded cache line works with out accessing the memory**



Contents

- **Improving Serial Performance**
  - → General serial performance optimizations
    - → Compiler optimization
    - → Examples for code optimization
  - → Memory Access
    - → Calculation of cache-optimized matrix norms

Example: Norm calculation

Norm calculation of a matrix

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

$$\|A\|_1 := \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{ij}| \qquad (\text{“maximum column sum”})$$

$$\|A\|_\infty := \max_{i=1,\ldots,n} \sum_{j=1}^{n} |a_{ij}| \qquad (\text{“maximum row sum”}).$$

Algorithms

→ `norm_max()`

→ `norm1_v1() /* naive implementation */`

→ `norm1_v2() /* spatial locality */`

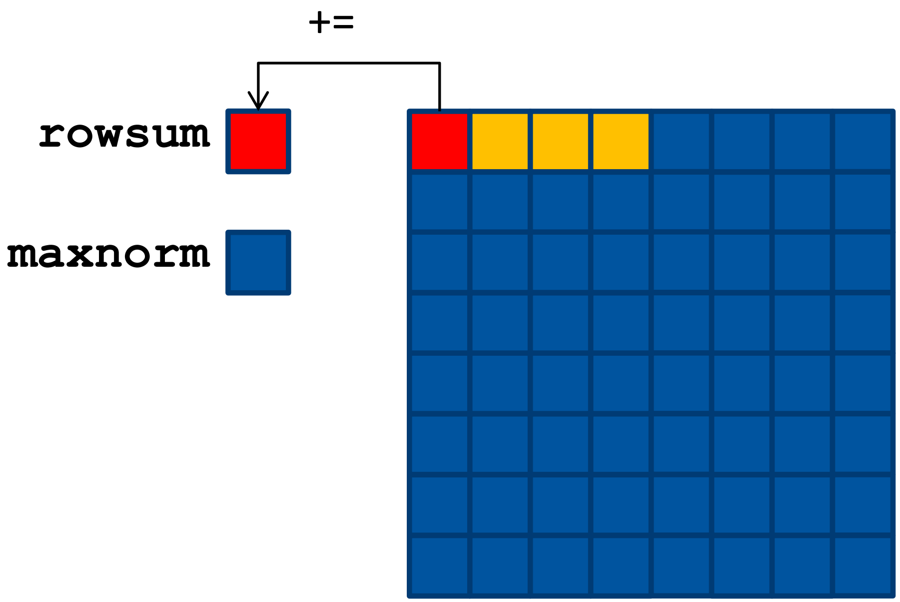
Norm calculation of a matrix
norm_max()

```
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;
}
```

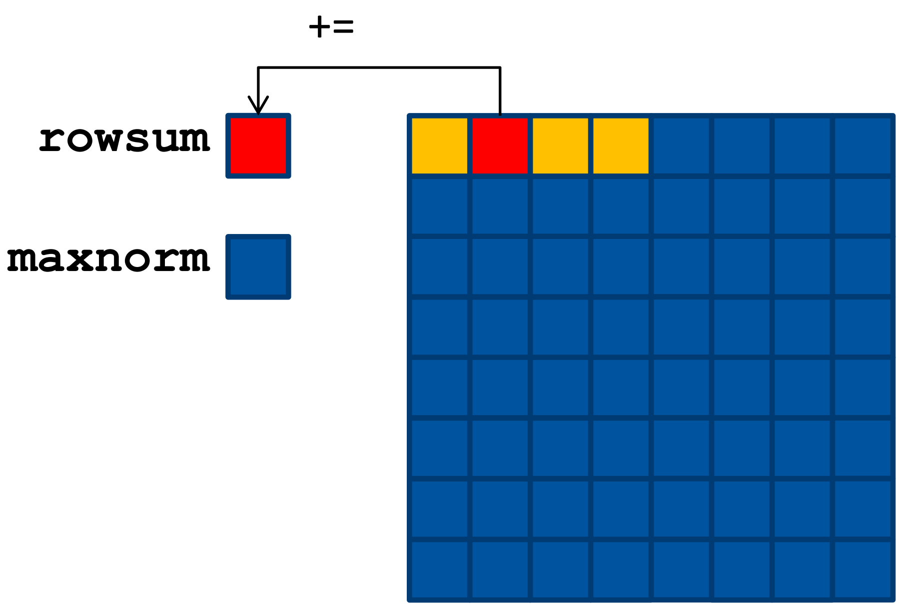
Execution Time CLAIX18 (n=10000)

Algorithm	Mflops
norm_max	1318

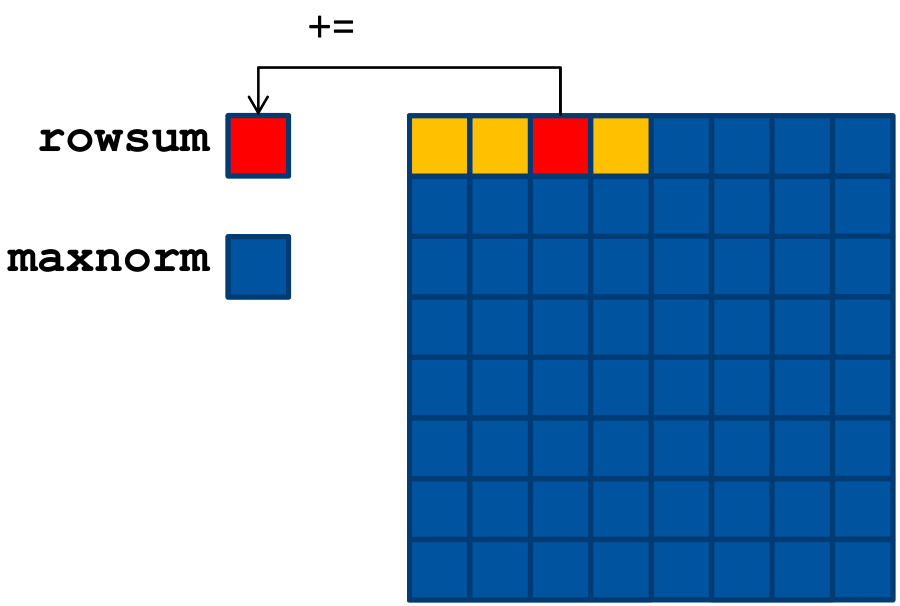
Norm calculation of a matrix
norm_max()



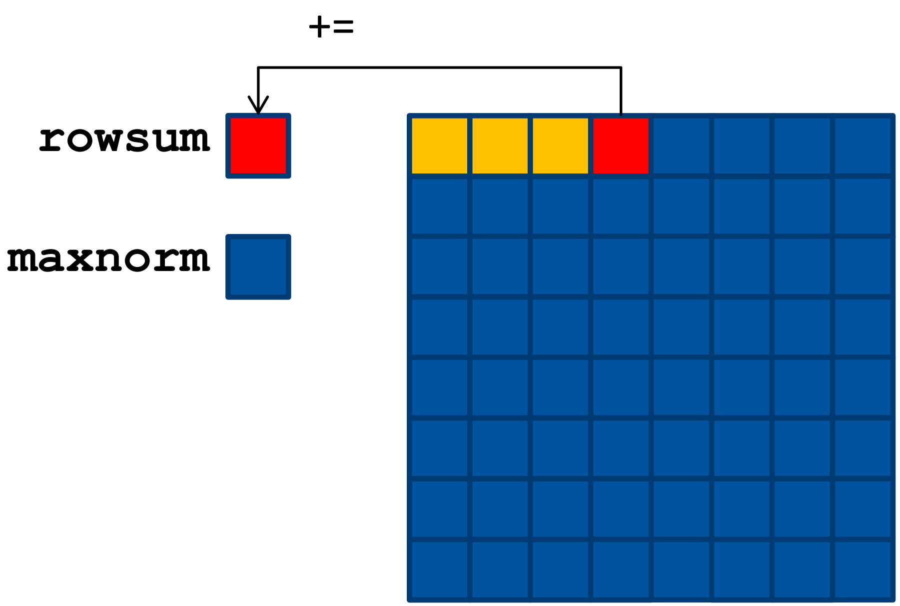
Norm calculation of a matrix
norm_max()



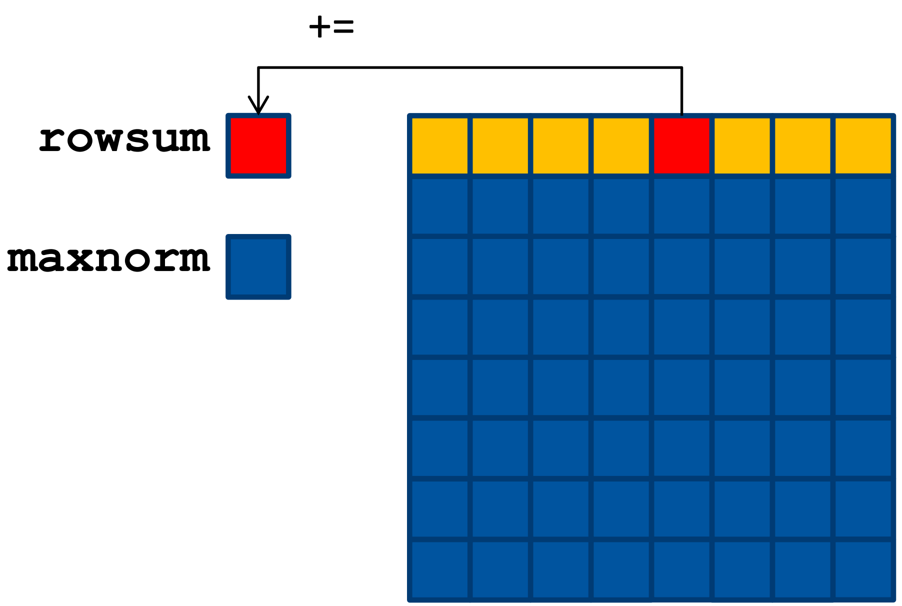
Norm calculation of a matrix
norm_max()



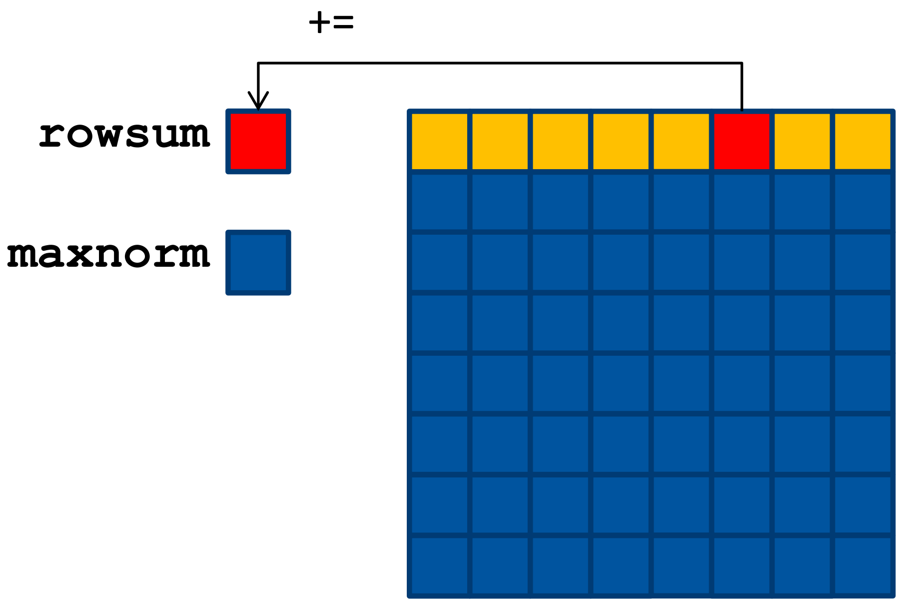
Norm calculation of a matrix
norm_max()



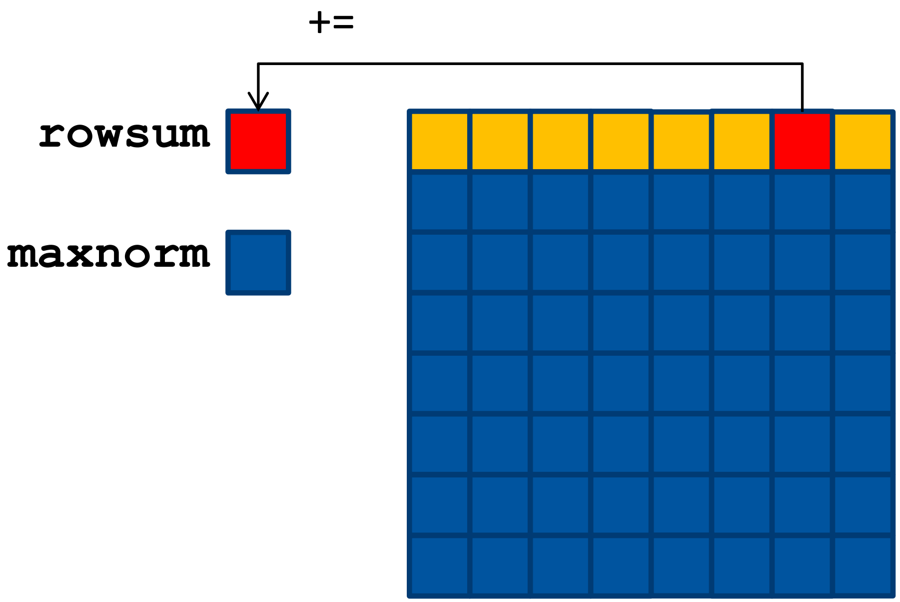
Norm calculation of a matrix
norm_max()



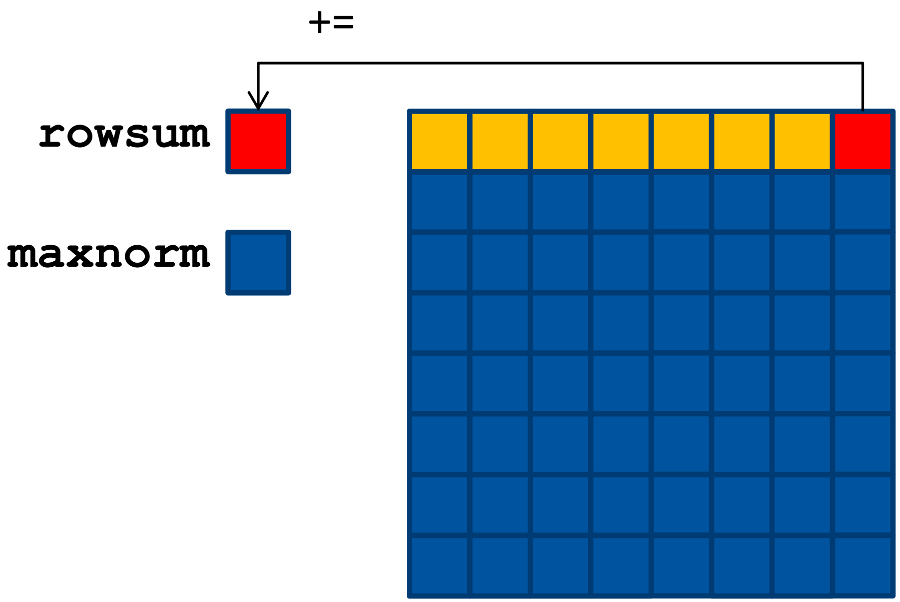
Norm calculation of a matrix
norm_max()



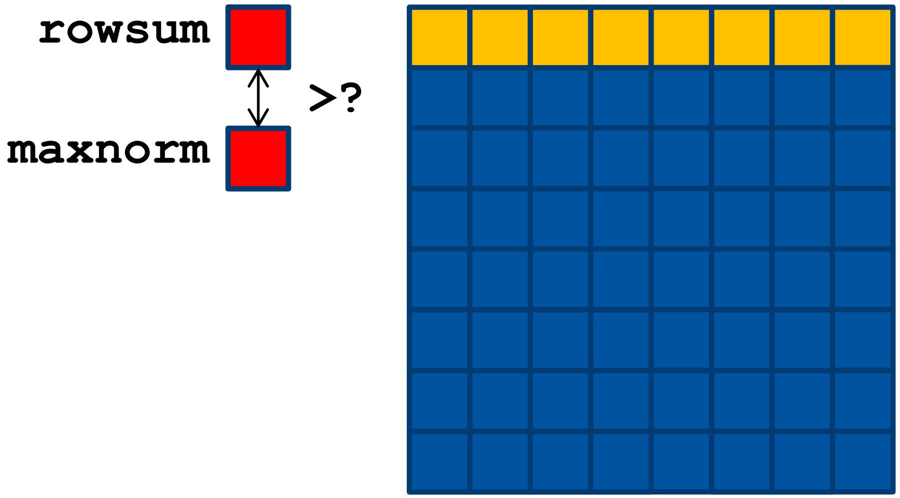
Norm calculation of a matrix
norm_max()



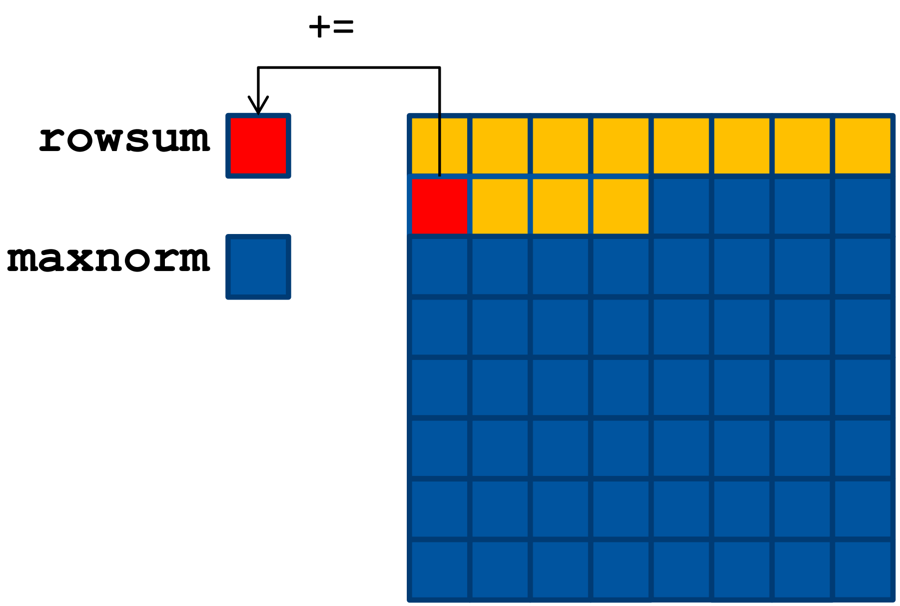
Norm calculation of a matrix
norm_max()



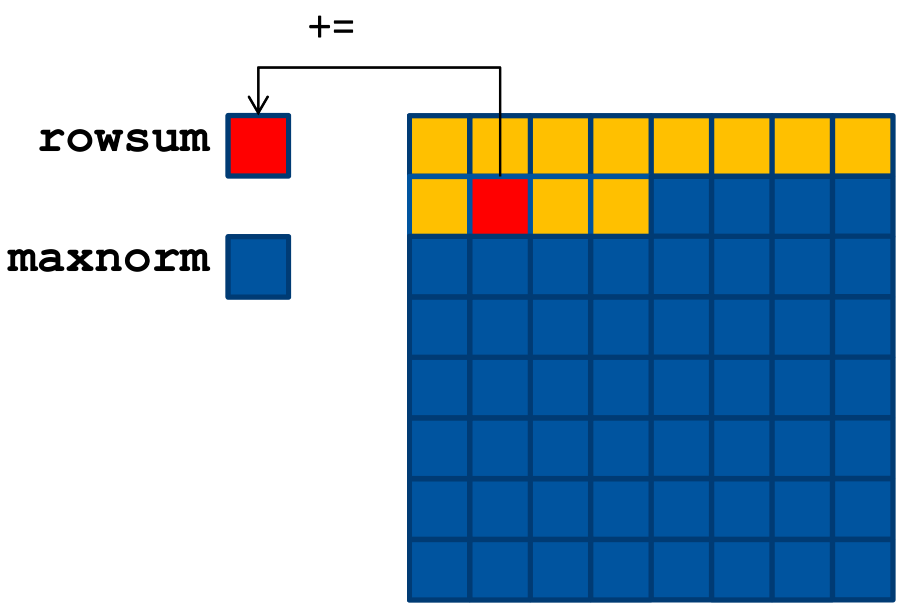
Norm calculation of a matrix
norm_max()



Norm calculation of a matrix
norm_max()



Norm calculation of a matrix
norm_max()



Norm calculation of a matrix
norm1_v1()

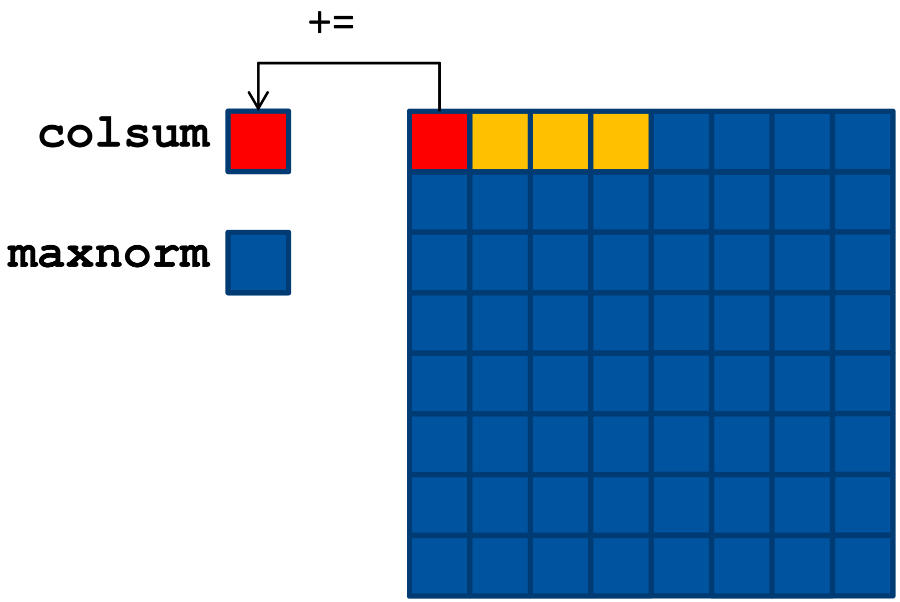
```
double norm1_v1(double** const A, const int n)
{
    double colsum=0., max_norm=-1.;
    for (int j=0; j<n; ++j)
    {
        colsum=0.;
        for (int i=0; i<n; ++i)
            colsum += abs(A[i][j]);
        if (colsum>max_norm)
            max_norm=colsum;
    }
    return max_norm;
}
```

3.5 times slower for quite similar algorithm.

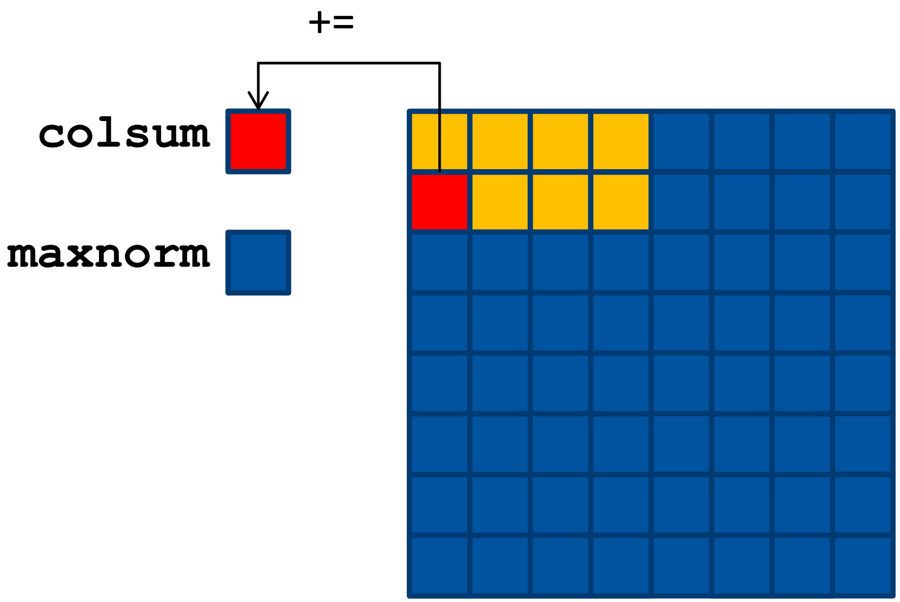
Execution Time CLAIX18 (n=10000)

Algorithm	Mflops
norm_max	1318
norm1_v1	392

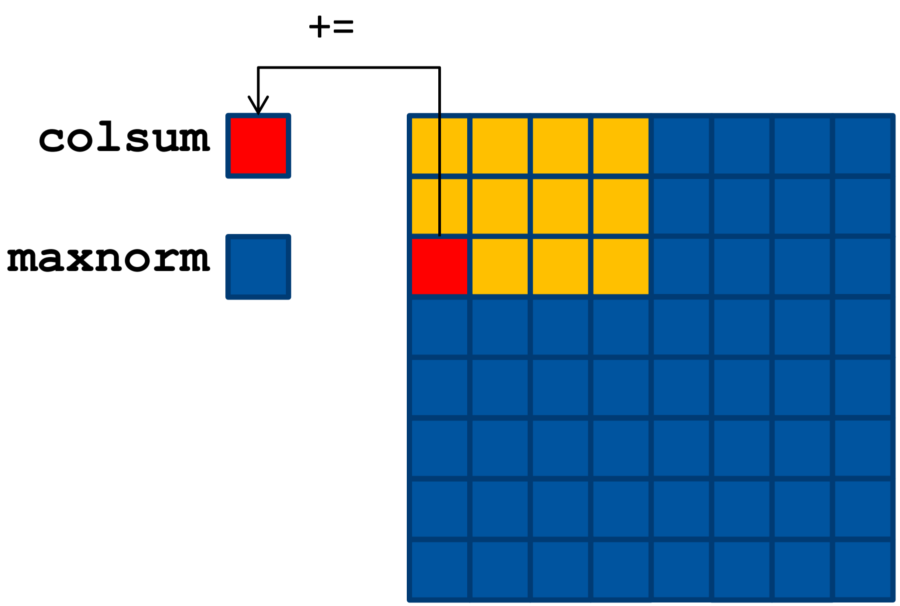
Norm calculation of a matrix norm1_v1()



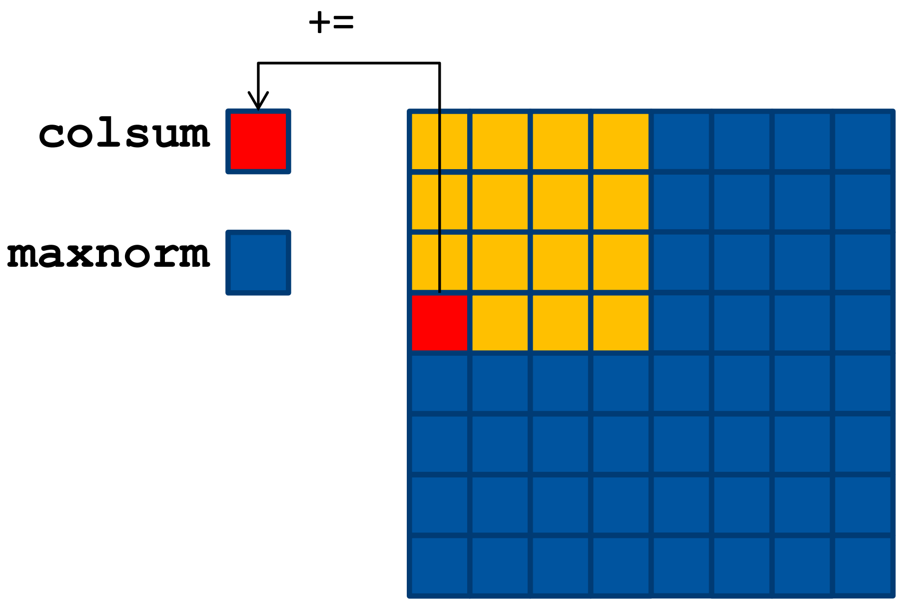
Norm calculation of a matrix
norm1_v1()



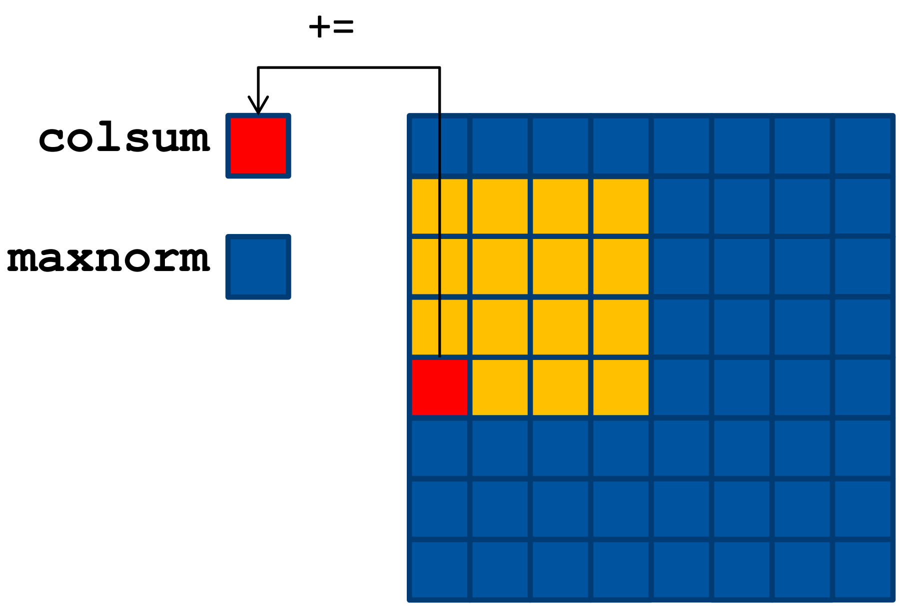
Norm calculation of a matrix
norm1_v1()



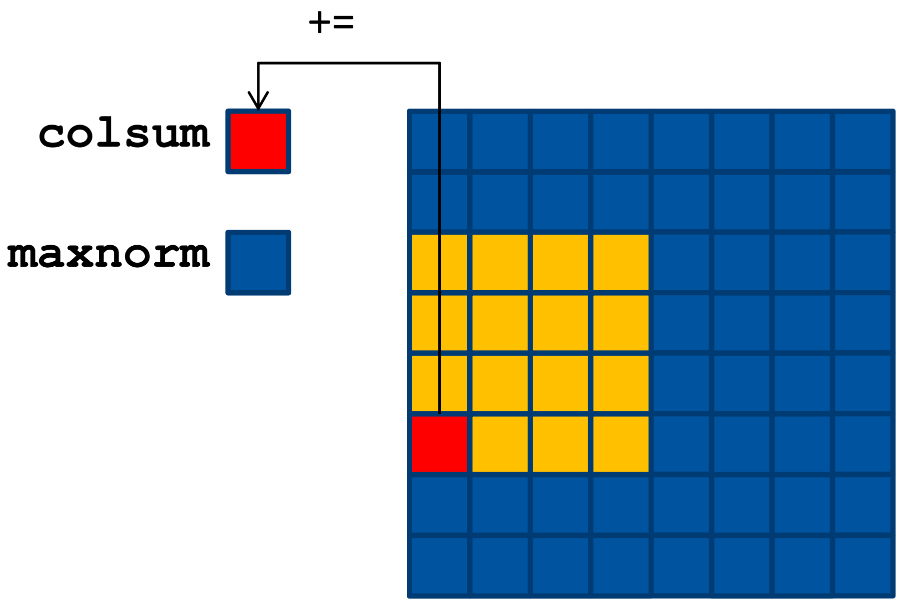
Norm calculation of a matrix
norm1_v1()



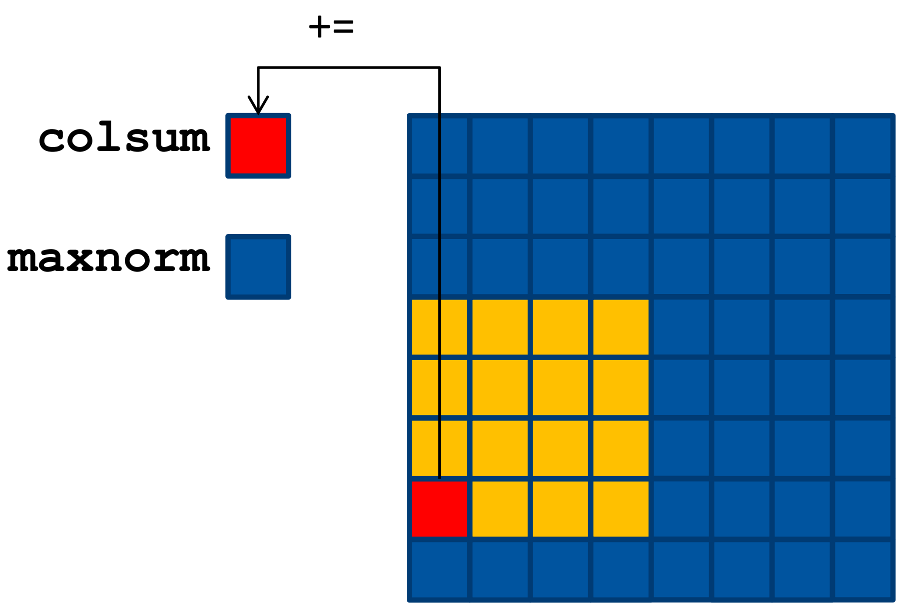
Norm calculation of a matrix
norm1_v1()



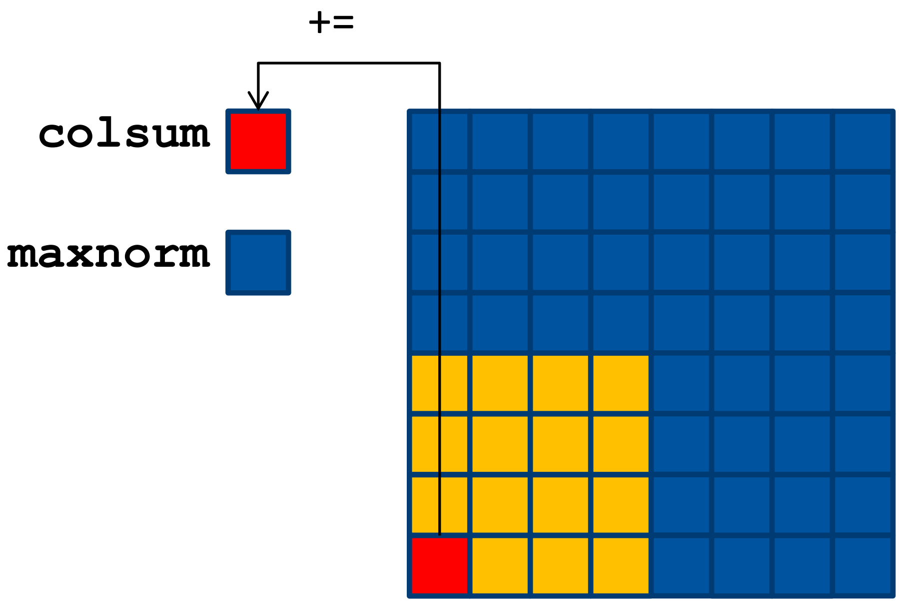
Norm calculation of a matrix
norm1_v1()



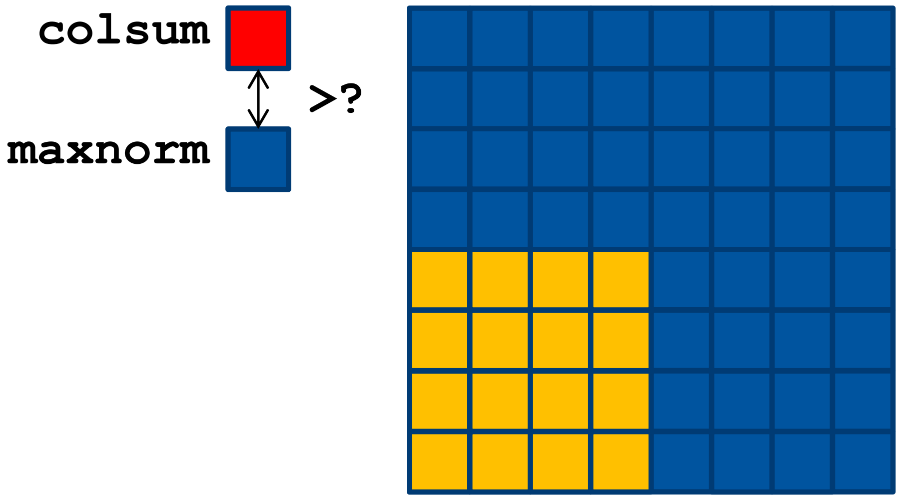
Norm calculation of a matrix
norm1_v1()



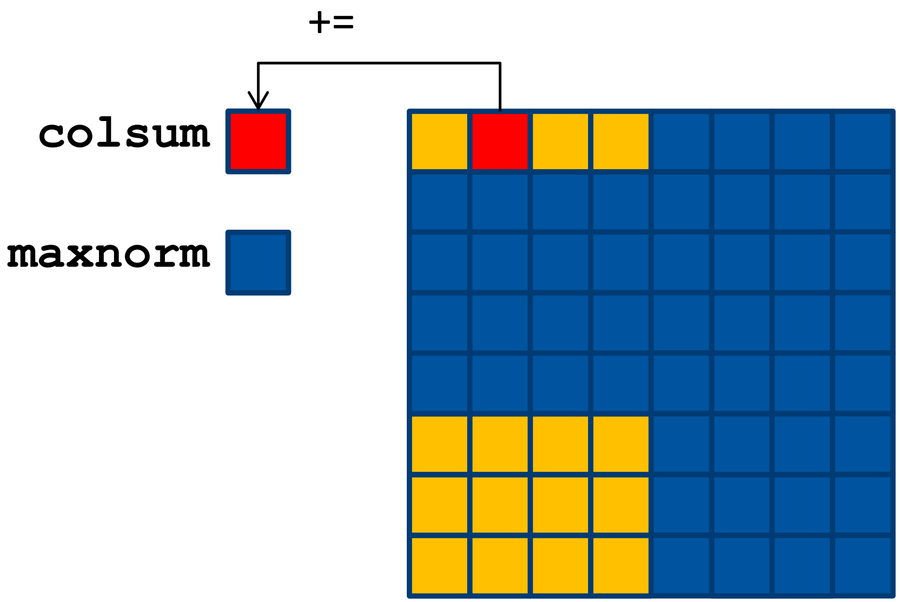
Norm calculation of a matrix norm1_v1()



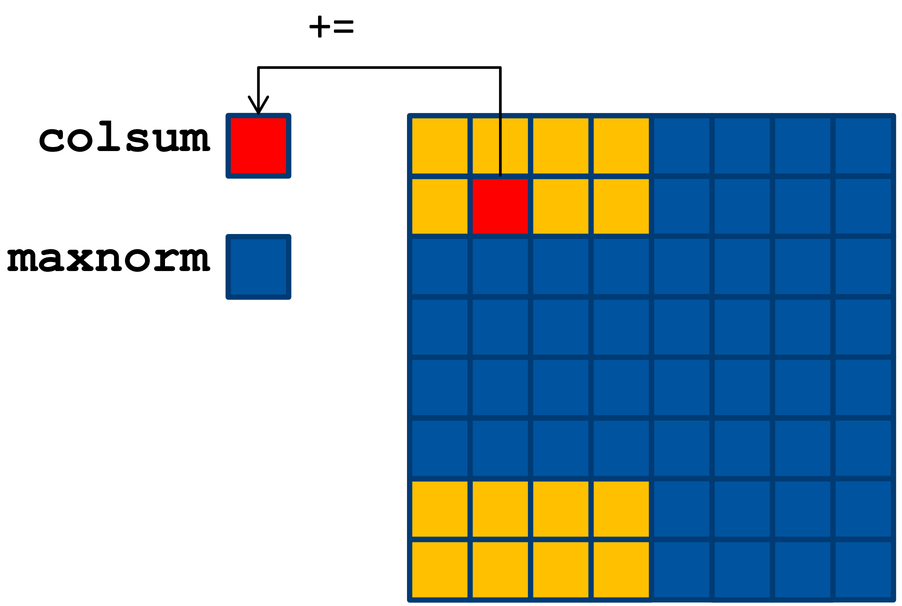
Norm calculation of a matrix
norm1_v1()



Norm calculation of a matrix
norm1_v1()



Norm calculation of a matrix norm1_v1()



Norm calculation of a matrix norm1_v2() 1/2

```
double norm1_v2(double** const A, const int n)
{
/* You need an auxiliary array for the column sums */
    double *colsums= new double[n];
    double max_norm=-1.;

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

...
```

Norm calculation of a matrix
norm1_v2() 2/2

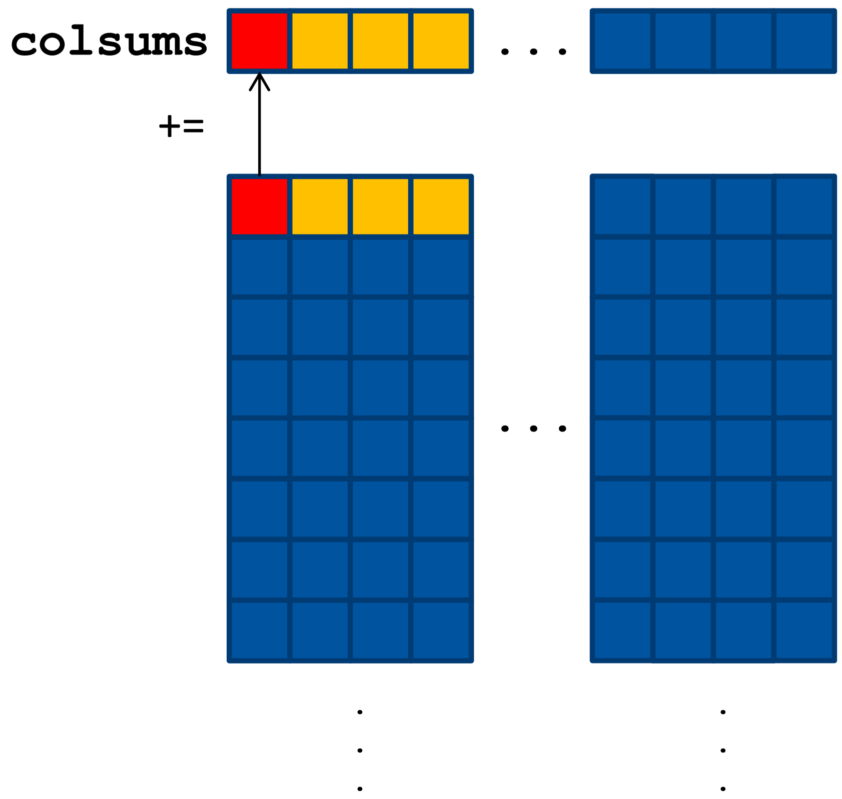
```
...

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

/* 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;
}
```

3 times fast for just switching the loop order.

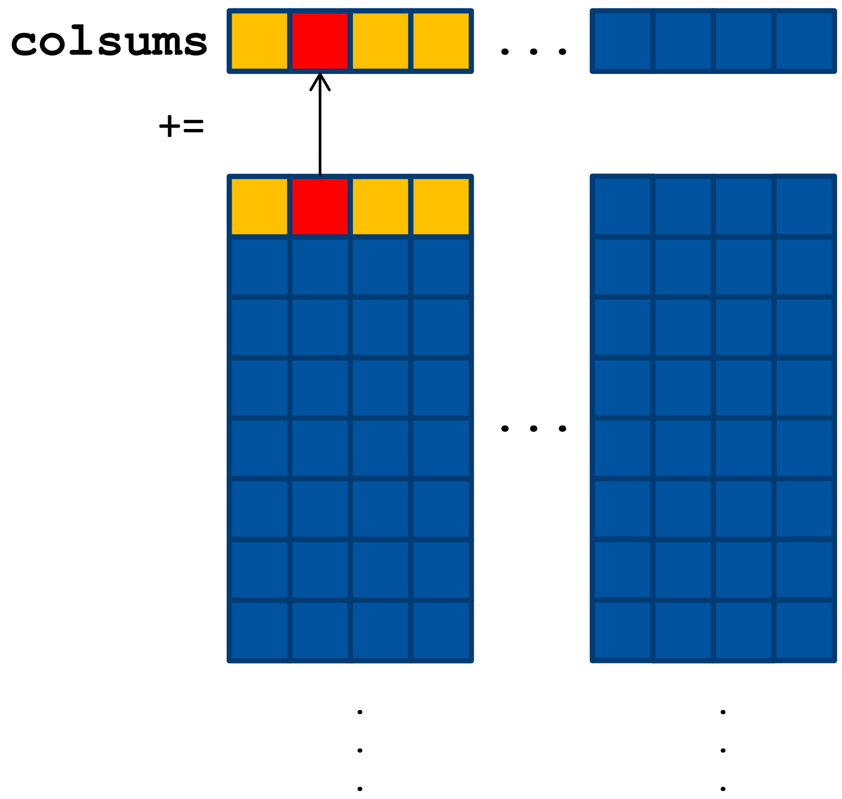
Execution Time CLAIX18 (n=10000)

Algorithm	Mflops
norm_max	1318
norm1_v1	392
norm1_v2	1189

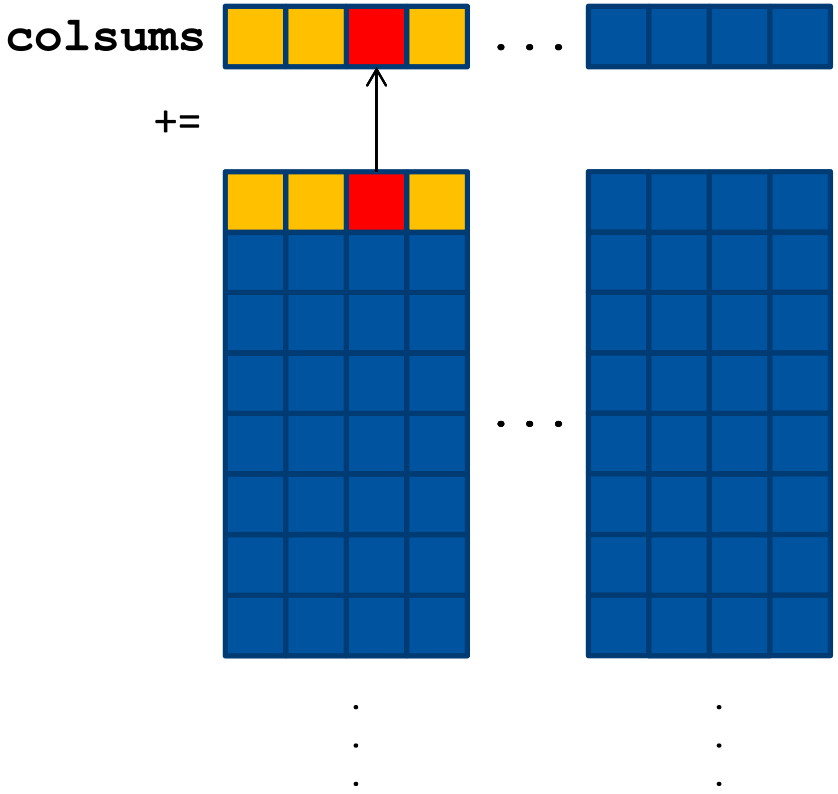
Norm calculation of a matrix
norm1_v2()



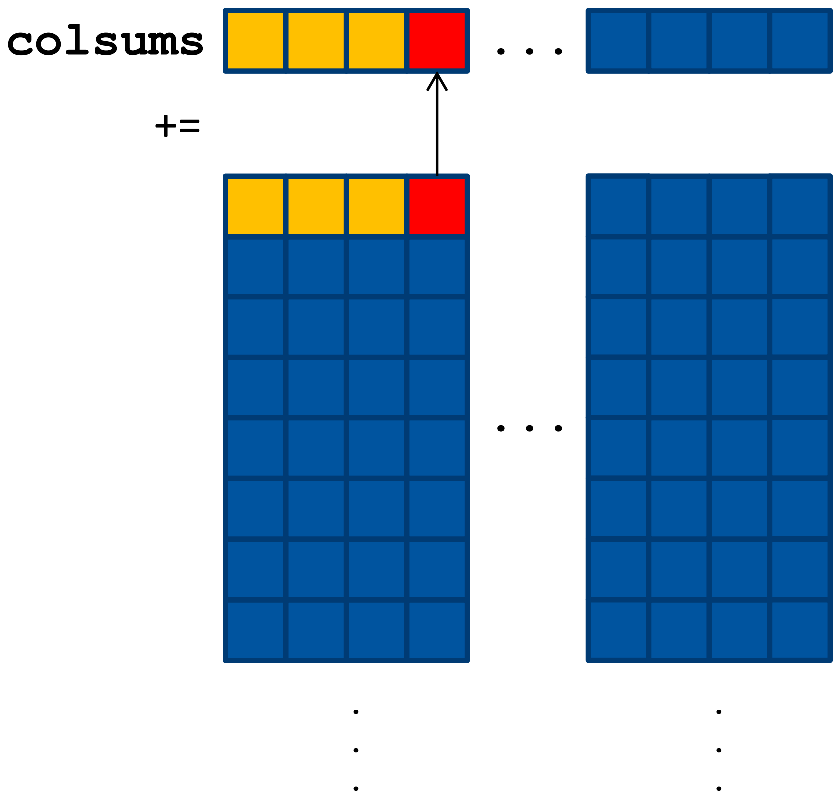
Norm calculation of a matrix
norm1_v2()



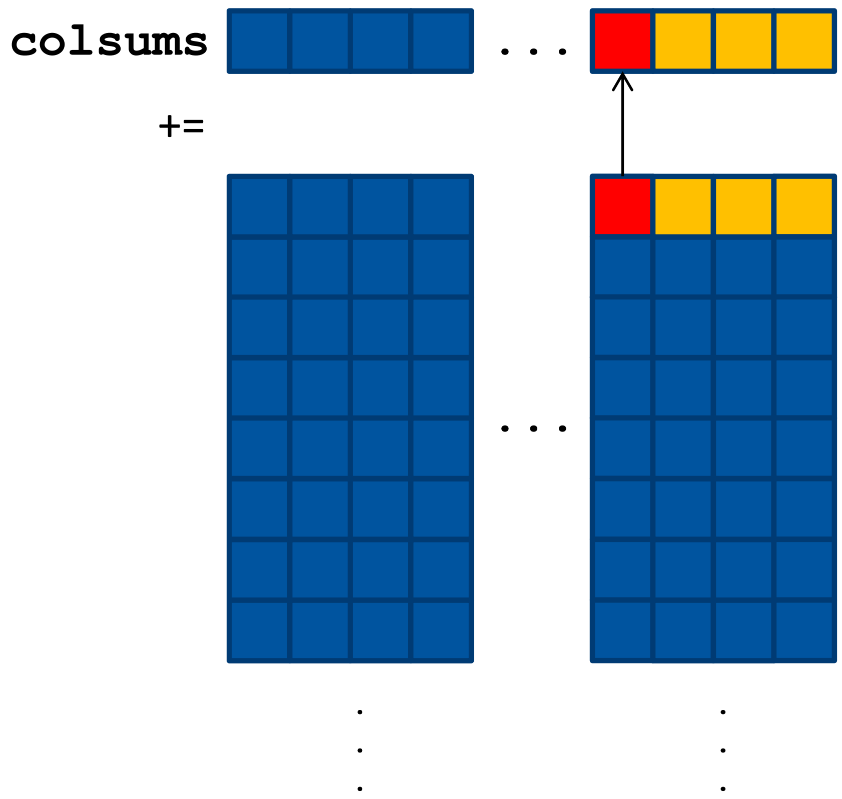
Norm calculation of a matrix norm1_v2()



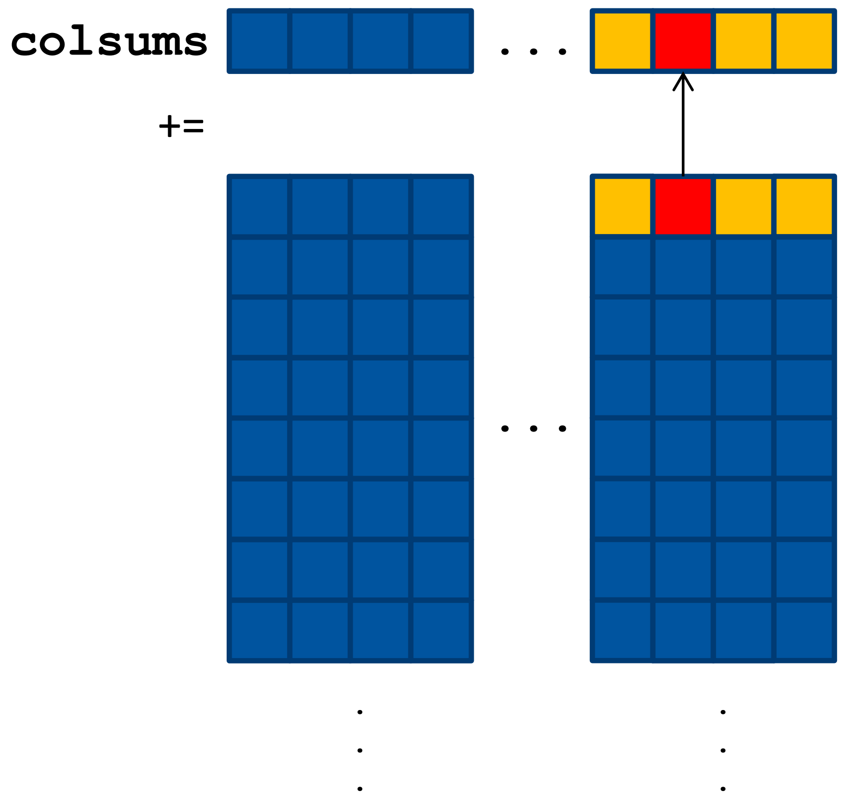
Norm calculation of a matrix
norm1_v2()



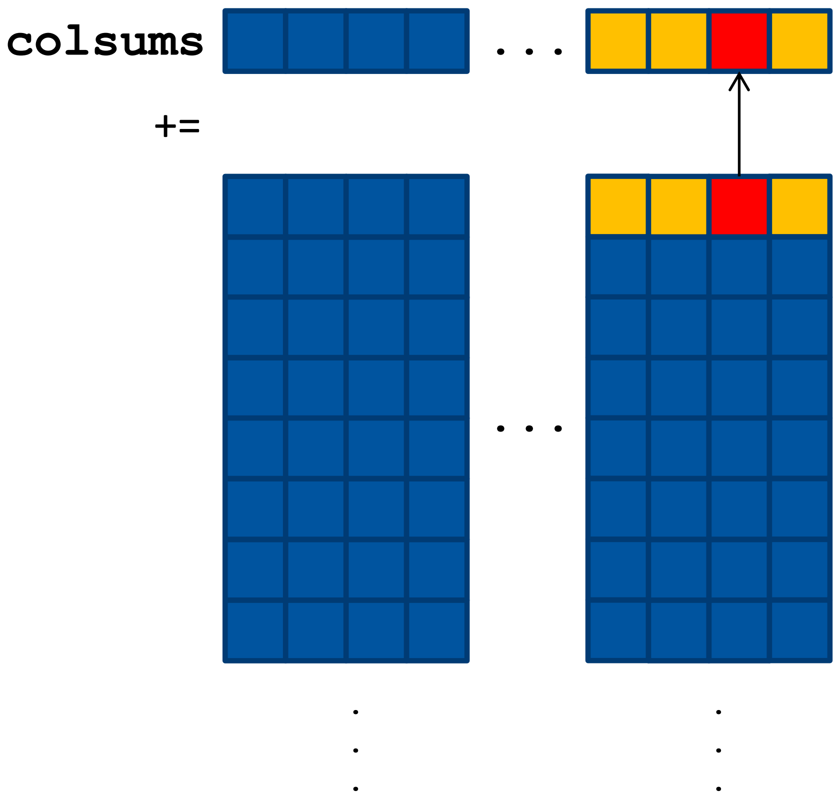
Norm calculation of a matrix norm1_v2()



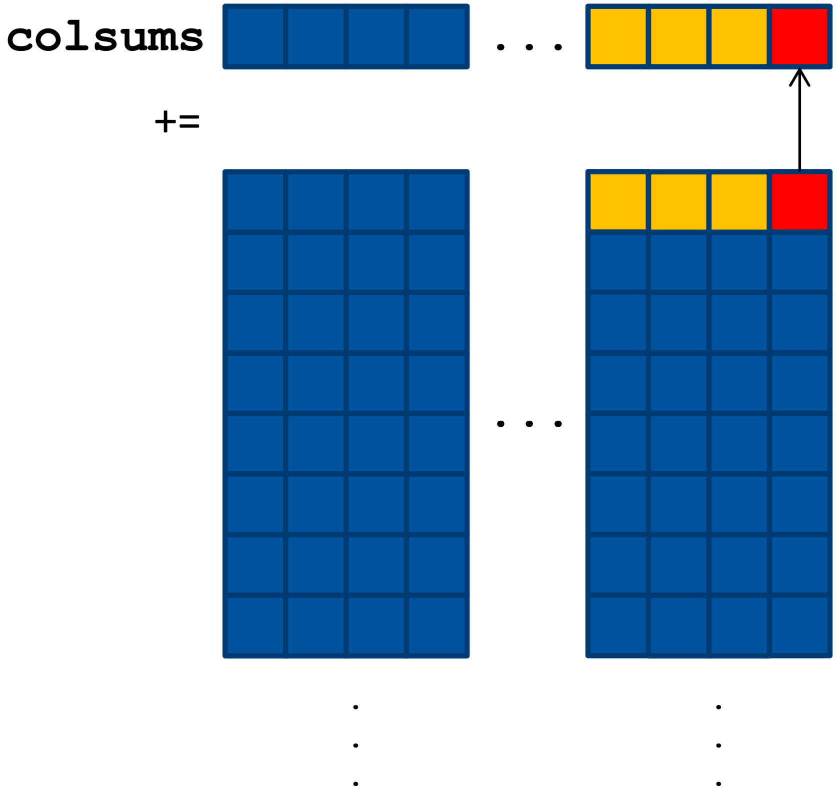
Norm calculation of a matrix norm1_v2()



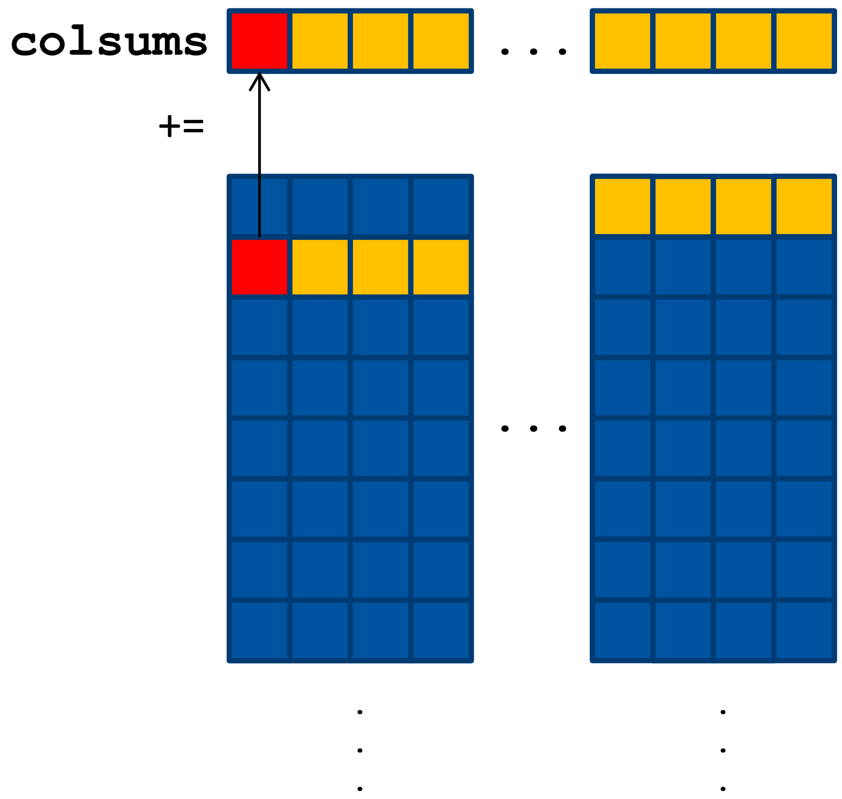
Norm calculation of a matrix
norm1_v2()



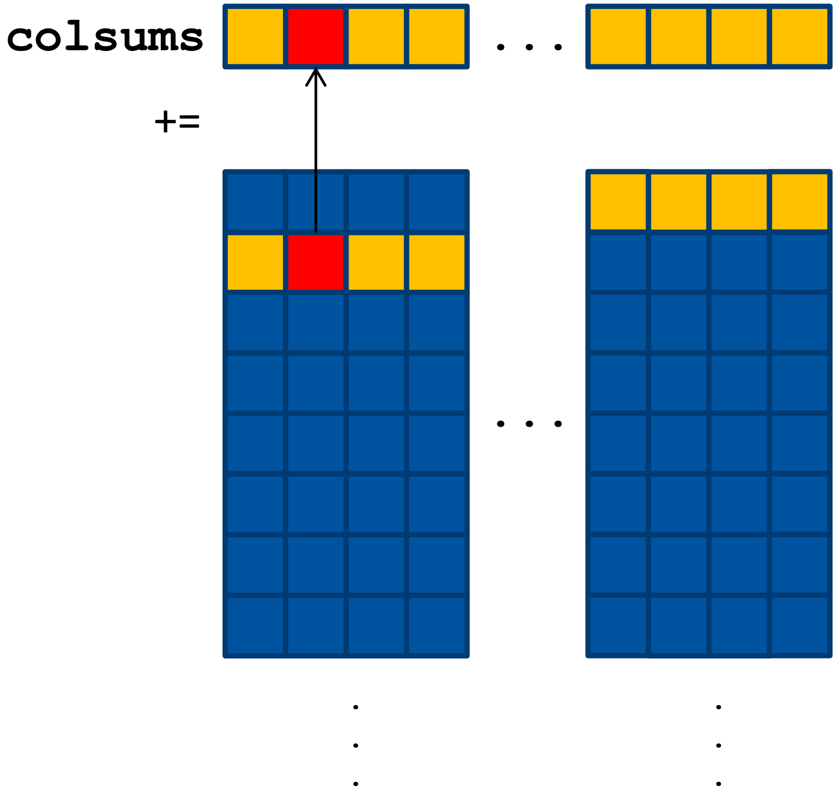
Norm calculation of a matrix
norm1_v2()



Norm calculation of a matrix norm1_v2()



Norm calculation of a matrix
norm1_v2()



Summary & Conclusion

- **Serial Performance is important**
  - → Low hanging fruits: Try different compiler & flag combinations
  - → Carefully check you algorithms at hotspots
    - → Do less work
    - → Avoid expensive operations
    - → Eliminate common subexpressions
    - → Avoid branches
    - → Use SIMD instruction sets
  - → Memory Access
    - → Remember that caches are organized in cache lines
    - → Check order of memory accesses for better cache behavior

Thank you for your attention.

Any questions?