Optimization

Lectures on Modern Scientific Programming 2016

Dániel Berényi
Wigner GPU Lab

Optimization - What? When? How?

Optimization is tailoring some parameters to a desired value
But what?

- Paid by work hours? Optimize work to take longer...
- Paid by lines of code? Maximize line breaks in the code
- Developing for an embedded system? Minimize memory usage
- Working on a HPC simulation? Tune for time...

Optimization - What? When? How?

Optimization is tailoring some parameters to a desired value
But what?

- Paid by work hours? Optimize work to take longer...
- Paid by lines of code? Maximize line breaks in the code
- Developing for an embedded system? Minimize memory usage
- Working on a HPC simulation? Tune for time

Optimization - What? When? How?

Optimization for time... What time?

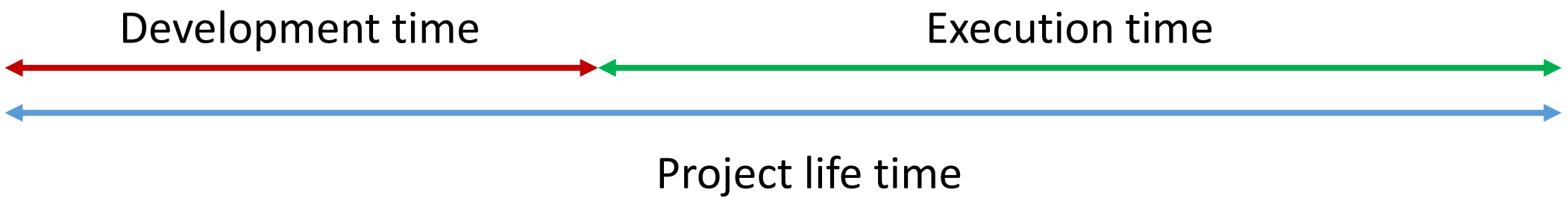
Optimization - What? When? How?

Optimization for time... What time?

Project life time

Optimization - What? When? How?

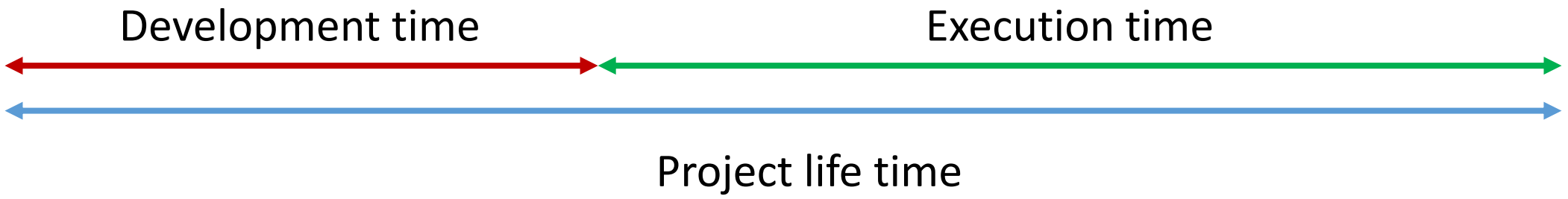
Optimization for time... What time?



Optimization - What? When? How?

Optimization for time... What time?

- How long will your program run?



Optimization - What? When? How?

Optimization for time... What time?

Only consider delving into optimizing program execution:

- If it is expected to run for a much-much longer time than the development (typically weeks, months, on a cluster of hundreds of cores)

- Or when low latency is of utmost importance

(online data processing and response)

Optimization - What? When? How?

Let’s start optimizing!

Optimization - What? When? How?

Let’s start optimizing!

- Step 1:

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Do not optimize!***

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Do not optimize!***

Premature optimization is the root of all evil <Insert scary image here>

(worse than shared mutable state!)

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Do not optimize!***
  - Verify the correctness of your code
  - Check memory management
  - Check API and library usage

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Do not optimize!***
  - Verify the correctness of your code
  - Check memory management
  - Check API and library usage

Read that goddamn manual at least ONCE

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Do not optimize!***
  - Verify the correctness of your code
  - Check memory management
  - Check API and library usage
  - Verify edge cases
  - Check dependence on environment
  - Check your floating point numbers (overflow, loss of precision, INF, NAN)

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Do not optimize!***
  - Verify the correctness of your code
  - Check memory management
  - Check API and library usage
  - Verify edge cases
  - Check dependence on environment
  - Check your floating point numbers (overflow, loss of precision, INF, NAN)
  - Check your includes and definitions especially in multiple translation units
  - Check the versions and builds of the libraries you’re linking
  - Verify argument passing to different languages, interfaces...

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Continued...***

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Continued...***
  - Verify your code in debug and non-debug (release) builds
  - Check uninitialized variables
  - Verify your code w.r.t. 32 - 64 bit width
  - Verify handling of exceptional cases, null pointers

Optimization - What? When? How?

Let’s start optimizing!

- Step 1: ***Continued...***
  - Verify your code in debug and non-debug (release) builds
  - Check uninitialized variables
  - Verify your code w.r.t. 32 - 64 bit width
  - Verify handling of exceptional cases, null pointers
  - Did I mention checking your memory management? Allocations? Deallocations?
  - Check floating point precision settings
  - Check your code with different compilers
  - Turn on more or all warnings

Optimization - What? When? How?

Let’s start optimizing!

- Step 2: ***Repeat step 1 until you are sure***

Optimization - What? When? How?

Let’s start optimizing!

- Step 2: ***Repeat step 1 until you are sure***

- ***Back up your working code!***
  - So you can compare optimized versions with the original

Optimization - What? When? How?

Let’s start optimizing!

- Step 2: ***Repeat step 1 until you are sure***

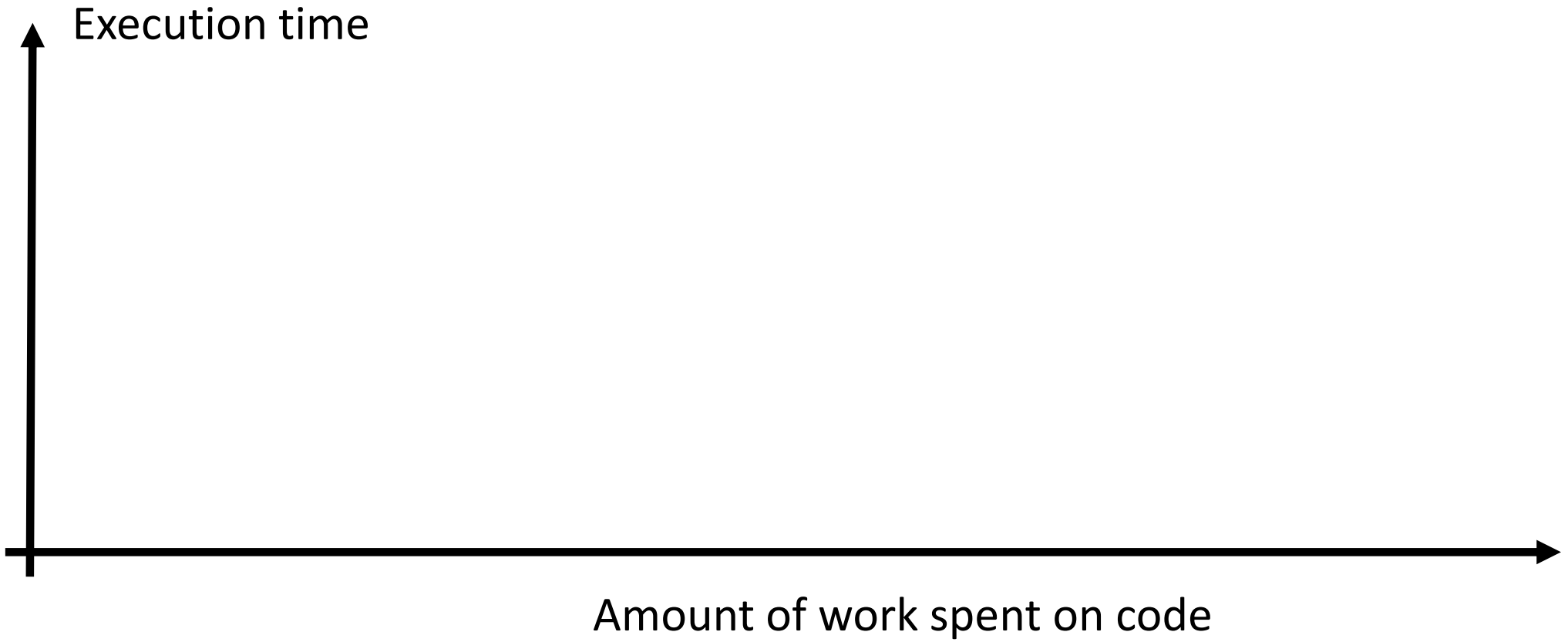
- ***Back up your working code!***
  - So you can compare optimized versions with the original

    NO! Not speed-wise

    But whether they do the same thing!

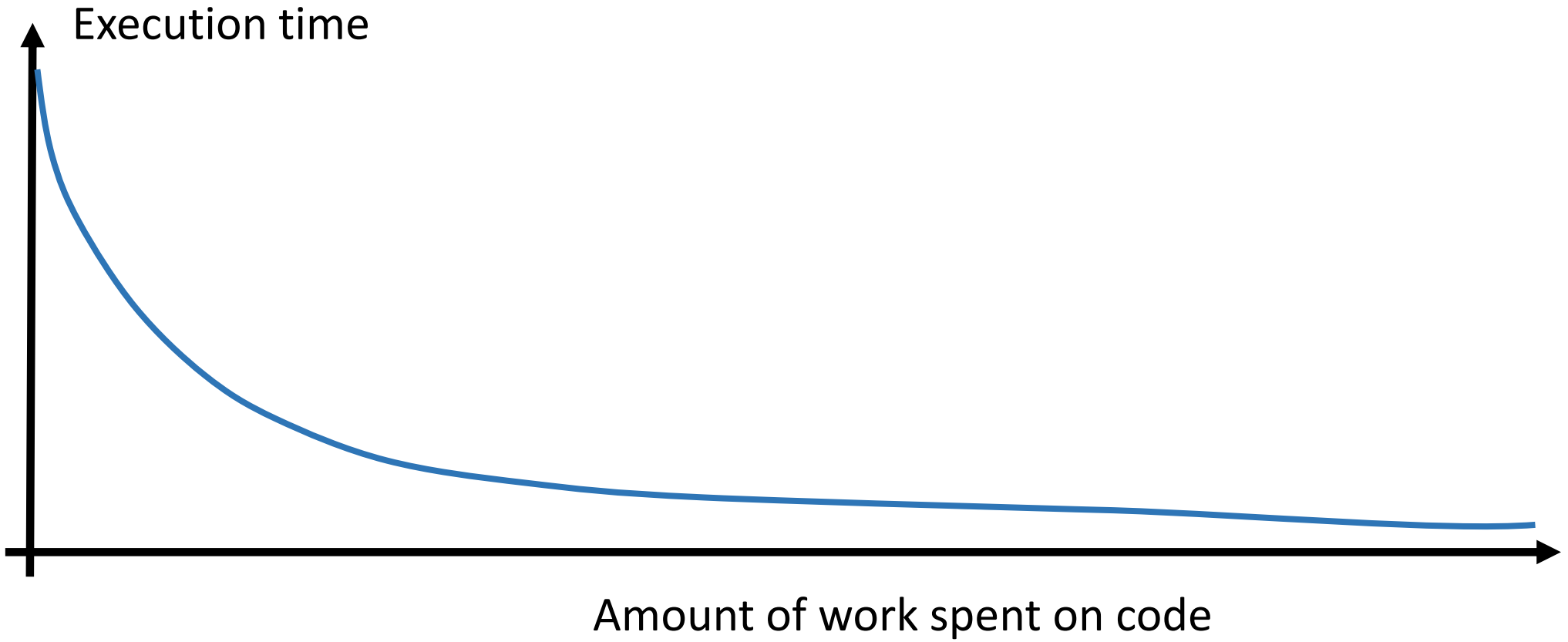
Optimization - What? When? How?

The curve of optimization gains:



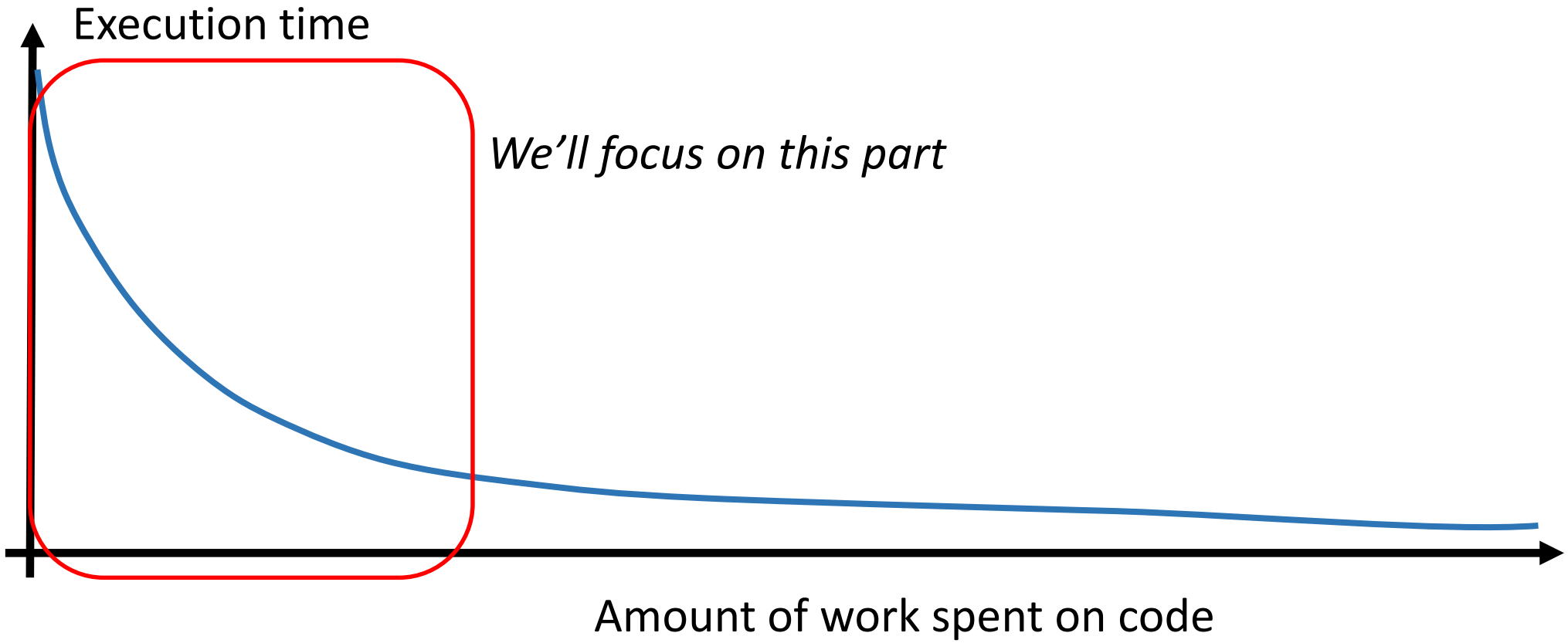
Optimization - What? When? How?

The curve of optimization gains:



Optimization - What? When? How?

The curve of optimization gains:



Time measurement

How fast is my code actually?

Time measurement

How fast is my code actually?

- One of the best and standard ways to measure time now is to use the standard library!

Time measurement

```
#include <chrono>
auto tmark(){
        return std::chrono::high_resolution_clock::now();
}

template<typename T1, typename T2>
auto delta_time( T1&& t1, T2&& t2 )
{
    return
std::chrono::duration_cast<std::chrono::nanoseconds>(t2-t1)
                                                .count()/1000.0;
}
```

Time measurement

```
auto t0 = tmark();

//fancy code to be measured

auto t1 = tmark();

std::cout << "My calculation took: "
          << delta_time(t0, t1) << " usecs.\n";
```

Time measurement

Time measurement can be tricky...

- System calls' delay depends on many factors (OS load)
- Time measurement itself is (light) a system call

Time measurement

Time measurement can be tricky...

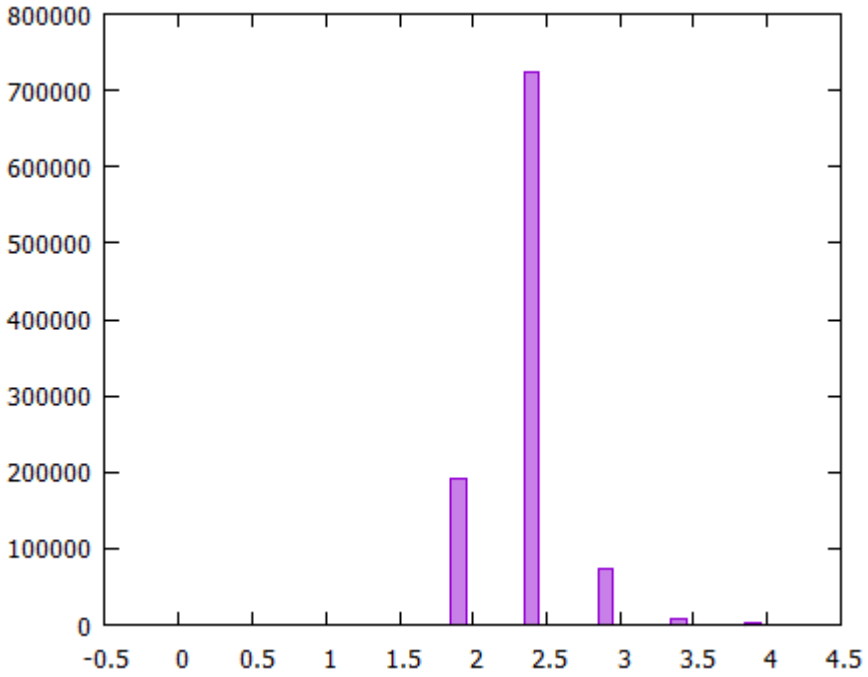
- System calls' delay depends on many factors (OS load)
- Time measurement itself is (light) a system call

- OS multi threading may put a thread aside, but measurement may not be corrected
- OS may put a thread from one core to the other measurements go frenzy

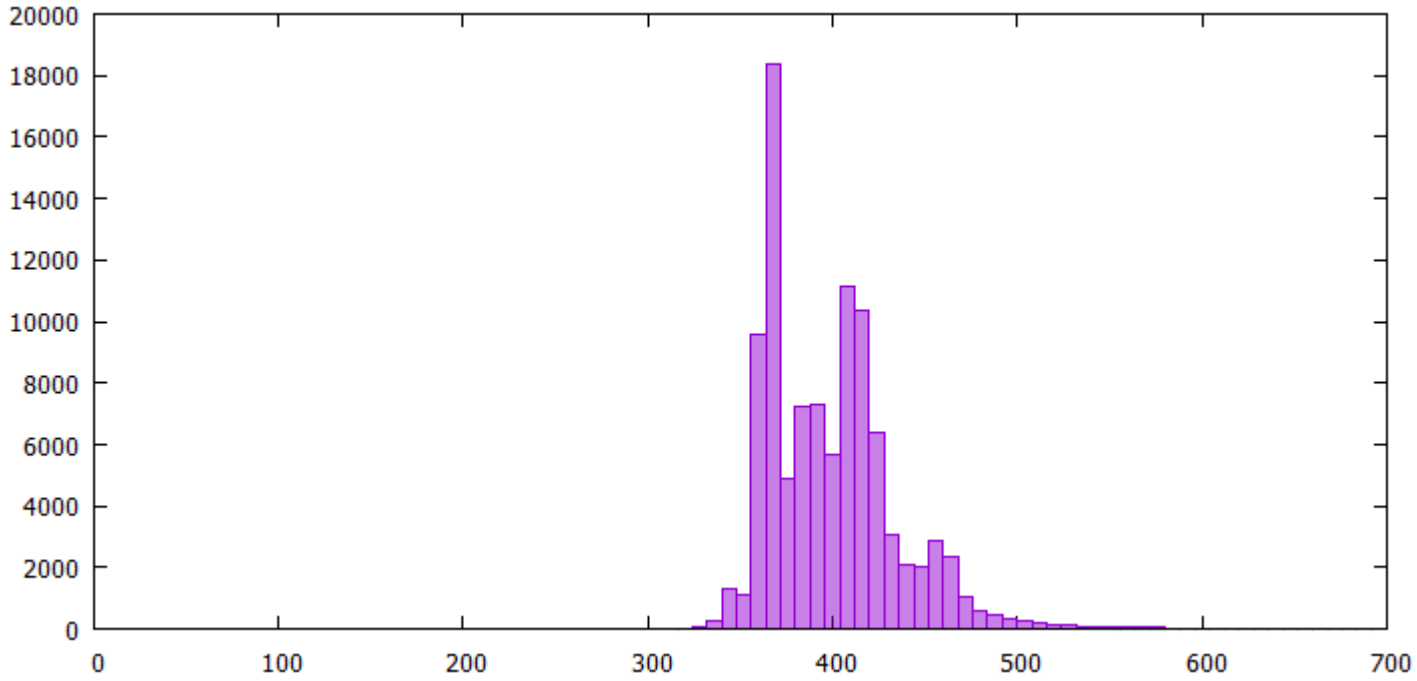
- The hardware clock underlying the measurement may do unexpected things especially on multicore systems

Time measurement

The distribution of measured times can be absolutely non-trivial:



Trigonometric sum (50 terms) [µs]



Memory allocation (300 kB) [µs]

Time measurement

- As a rule of thumb we usually measure code multiple times (tens-hundreds) and take the *minimum*
- OS and scheduling noises expected to just *increase* the execution time
- so the *minimum* should be a good estimator for the time of the actual code

Optimization - What? When? How?

Cheapest optimizations:

Optimization - What? When? How?

Cheapest optimizations:

Know your compiler

Remember:

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Taken care by the processor and the compiler

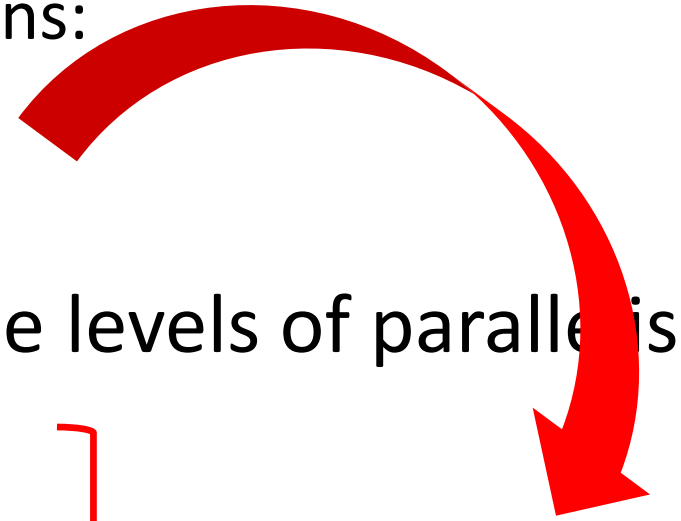
Optimization - What? When? How?

Cheapest optimizations:

Know your compiler

Remember:

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level



Taken care by the processor and the compiler

Optimization - What? When? How?

Cheapest optimizations:

- Compiler optimizations! Just a flip of switch
  - Enable optimizations (-O3)
  - Enable fast floating point (you guarantee that no NaNs Infs occur, -fast-math)
  - Enable enhanced instruction set usage (SSE, AVX, ...)
  - Enable / tune vectorization efficiency

Check your compiler's manual!

Optimization - What? When? How?

Cheapest optimizations:

- Compiler optimizations! Just a flip of switch
  - Enable optimizations (-O3)
  - Enable fast floating point (you guarantee that no NaNs Infs occur, -fast-math)
  - Enable enhanced instruction set usage (SSE, AVX, ...)
  - Enable / tune vectorization efficiency

Check your compiler's manual!

Verify program correctness! Some optimizations may alter the working of your program, especially if your are using some hacks...

Optimization – x86 instruction timings

-

Operation	Latency
Shift / Rot	1-4
AND / OR / XOR	1-4
Compare/test	1-4
Call (Ret)	5 (8)
Integer add/sub	1-8
Integer mul	3-18
Integer div	32-103

Operation	Latency
MMX	1-9
SSE Integer	1-9
SSE single + - logical	1-12
SSE single mul	3-7
SSE single div/sqrt	10-40
SSE2 double simple ops	1-12
SSE2 double mul	3-7
SSE2 double div/sqrt	14-70
SSE2 128bit int	1-10
SSE3 / SSE4	1-14

AVX figures are approximately the same as SSE

Operation	Latency
FADD/FSUB/FABS	2-6
FMUL	7-8
FDIV	23-44
FSQRT	23-44
FSIN, FCOS	160-280
FSINCOS	160-250
FPTAN	225-300
FPATAN	150-300
FSCALE	60
FYL2X/FYL2XP1	100-250

Only the ratios are important!

Optimization – high level timings

Operation	Time [ns]	Time [ms]
1 clock cycle on a 3 GHz processor	1	1e-6
L1 cache access	0.5	5e-7
Branch misprediction	5	5e-6
L2 cache access	7	7e-6
Mutex lock/unlock	25	2,5e-5
RAM access	100	0,0001
1kB data compression with Snappy	3000	0,003
1kB data transmission on a 1 Gbps network	10000	0,01
4 kB data random access read from an SSD	150000	0,15
1 MB data contiguous read from RAM	250000	0,25
Roundtrip in a datacenter	500000	0,5
1 MB data contiguous read from SSD	1e6	1
Hard disk ‚seek' (search) time	1e7	10
1 MB data contiguous read from a hard disk	2e7	20
TCP/IP packet travel time between continents	1.5e8	150

Source and video to watch

Optimization - Timings

Why is it good to roughly know the timings?

- One can approximately design the program:
  - Where different data should be located
    (most accessed in cache and memory, big or/but rarely needed on disk)
  - What expressions should be preferred in formulas

Optimization - Timings

What expressions should be preferred in formulas?

We often see people copy-pasting formulas from Mathematica

The problem is that many formulas can be very much simplified by introducing new temporary sub expressions

The compiler is not expected to know *all* math identities and cannot optimize your formulas as well as you could...

Optimization - Timings

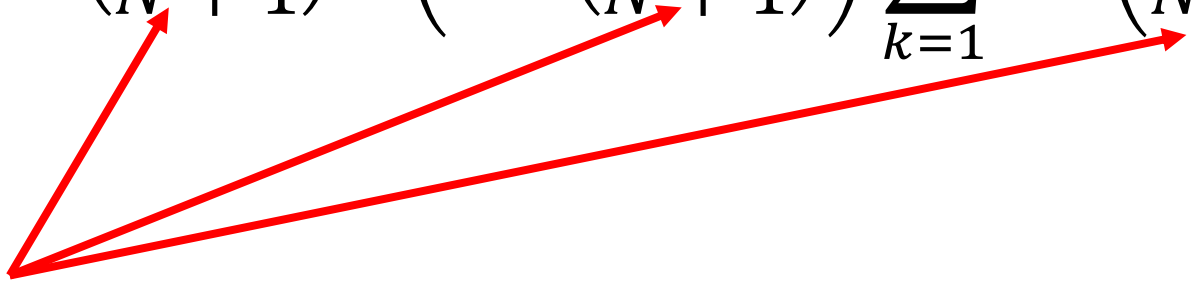
Formula:

$$\sum_{j=1}^{n} \sin\left(\frac{\pi j}{N+1}\right) f\left(\cos\left(\frac{\pi j}{N+1}\right)\right) \sum_{k=1}^{n} \sin\left(\frac{\pi k j}{N+1}\right) \frac{1-\cos(\pi \mathrm{k})}{k}$$

Optimization - Timings

Formula:

$$\sum_{j=1}^{n} \sin\left(\frac{\pi j}{N+1}\right) f\left(\cos\left(\frac{\pi j}{N+1}\right)\right) \sum_{k=1}^{n} \sin\left(\frac{\pi k j}{N+1}\right) \frac{1-\cos(\pi \mathrm{k})}{k}$$



Same factor is used multiple times, lets factor out!

Optimization - Timings

Formula:

$$\text{Let } \mathrm{q_j} = \frac{\pi j}{N+1}$$

$$\sum_{j=1}^{n} \sin(q_j)\, f\left(\cos(q_j)\right) \sum_{k=1}^{n} \sin(k q_j)\, \frac{1-\cos(\pi \mathrm{k})}{k}$$

Optimization - Timings

Formula:

Think more... cos is expensive, and here it is only evaluated at integer multiplies of π

Let $q_j = \frac{\pi j}{N+1}$

$1 - \cos(\pi k)$ = (k%2==1 ? 2.0/k : 0.0)

$$\sum_{j=1}^{n} \sin(q_j) f\left(\cos(q_j)\right) \sum_{k=1}^{n} \sin(k q_j) \frac{1-\cos(\pi k)}{k}$$

Optimization - Timings

More complex formulas may be less trivial to refactor and optimize, but you compiler will thank you!

Do not worry about temporaries, the compiler will remove them

Do worry about complex special function expressions and use identities to simplify them

Optimization – Memory!

Beginner-Intermediate optimization:

Memory

Optimization – Memory!

Beginner-Intermediate optimization:

In fact, 80% of optimizations

are dealing with data access and data organization in memory

Optimization – Memory!

Beginner-Intermediate optimization:

In fact, 80% of optimizations

are dealing with data access and data organization in memory

Accordingly:

80% of performance issues we meet are related to memory management and data access problems

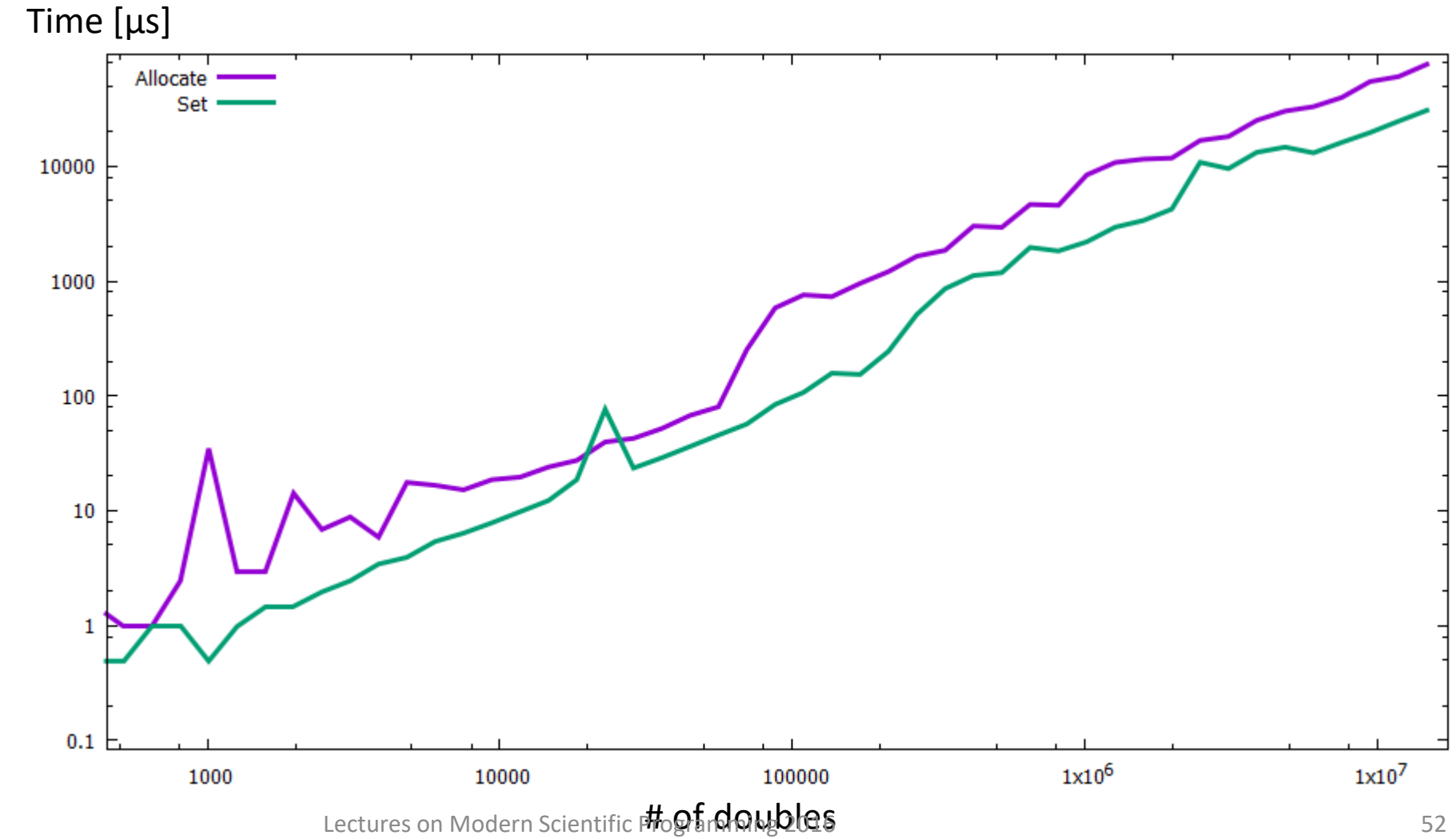
Optimization – Memory!

Question:

- Which one takes longer:
  - heap memory allocation or
  - zeroing out the allocated memory?

Optimization – Memory!

Answer:



Optimization – Memory!

Conclusion: dynamic heap memory allocation is slow...

Solution: avoid dynamic heap memory allocation☺

Optimization – Memory!

Especially avoid:

```
for(int i=0; i<N; ++i)
{




}
```

Optimization – Memory!

Especially avoid:

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


     }
}
```

Optimization – Memory!

Especially avoid:

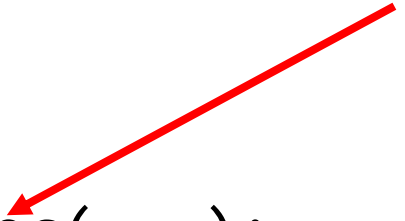
```
for(int i=0; i<N; ++i)
{
      for(int j=0; j<N; ++j)
      {
            for(int k=0; k<N; ++k)
            {
                  std::vector<double> vec(...);
            }
      }
}
```

Optimization – Memory!

Especially avoid:

```
for(int i=0; i<N; ++i)
{
      for(int j=0; j<N; ++j)
      {
            for(int k=0; k<N; ++k)
            {
                  std::vector<double> vec(...);
            }
      }
}
```

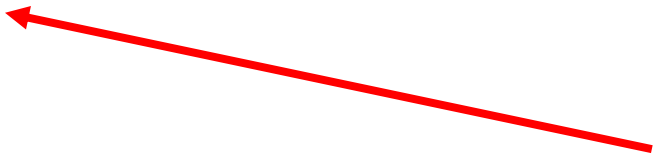
You've just payed N^3 times the allocation cost



Optimization – Memory!

```
std::vector<double> vec(...);
for(int i=0; i<N; ++i)
{
      for(int j=0; j<N; ++j)
      {
            for(int k=0; k<N; ++k)
            {

            }
      }
}
```



If possible, move the allocation out of the loop!

Optimization – Memory!

C++ standard library containers that by default use dynamical heap allocations:

- `std::vector`, `std::list`, `std::set`, ...

Containers that do not use dynamical heap allocations:

- `std::array<T, n>`

Optimization – Memory!

If you need to use dynamical heap allocation

- Try to allocate at the beginning of complex calculations

- If you do not know exactly how much memory you'll need, try to reasonably well estimate it!

Optimization – Memory!

Widely used resizable containers:

`std::vector, std::list`

Let’s compare them!

Optimization – Memory!

Widely used resizable containers:

`std::vector, std::list`

Let's compare them!

Sequential push_back:

#double	std::vector [ms]	std::list [ms]
10k	0.3	1.1
1M	32	90
10M	300	920
100M	2700	9300

Optimization – Memory!

Widely used resizable containers:

`std::vector, std::list`

Let's compare them!

Sequential push_back:

#double	std::vector [ms]	std::list [ms]	One time alloc [ms]
10k	0.3	1.1	0.05
1M	32	90	5
10M	300	920	50
100M	2700	9300	500

Optimization – Memory!

Widely used resizable containers:

`std::vector`, `std::list`

When comparing other studies (link, link), we can conclude:

The only case, when `std::list` is faster, when large amount of data need to be inserted or removed from the beginning or middle of the dataset.

Optimization – Memory!

If you're absolutely limited by allocation speed, you choices:

- Analyze the distribution of your allocations
- Write a memory manager yourself, that is tailored for those statistics
- And overload `new`/`delete` to use your memory manager and not `malloc`

Or:

- Use a pre written memory manager (tcmalloc, jemalloc)
- There exist tools, that intercept c library calls and are much better than `malloc` for frequent small allocations...

Optimization – Memory!

Data access!

Optimization – Memory!

Data access!

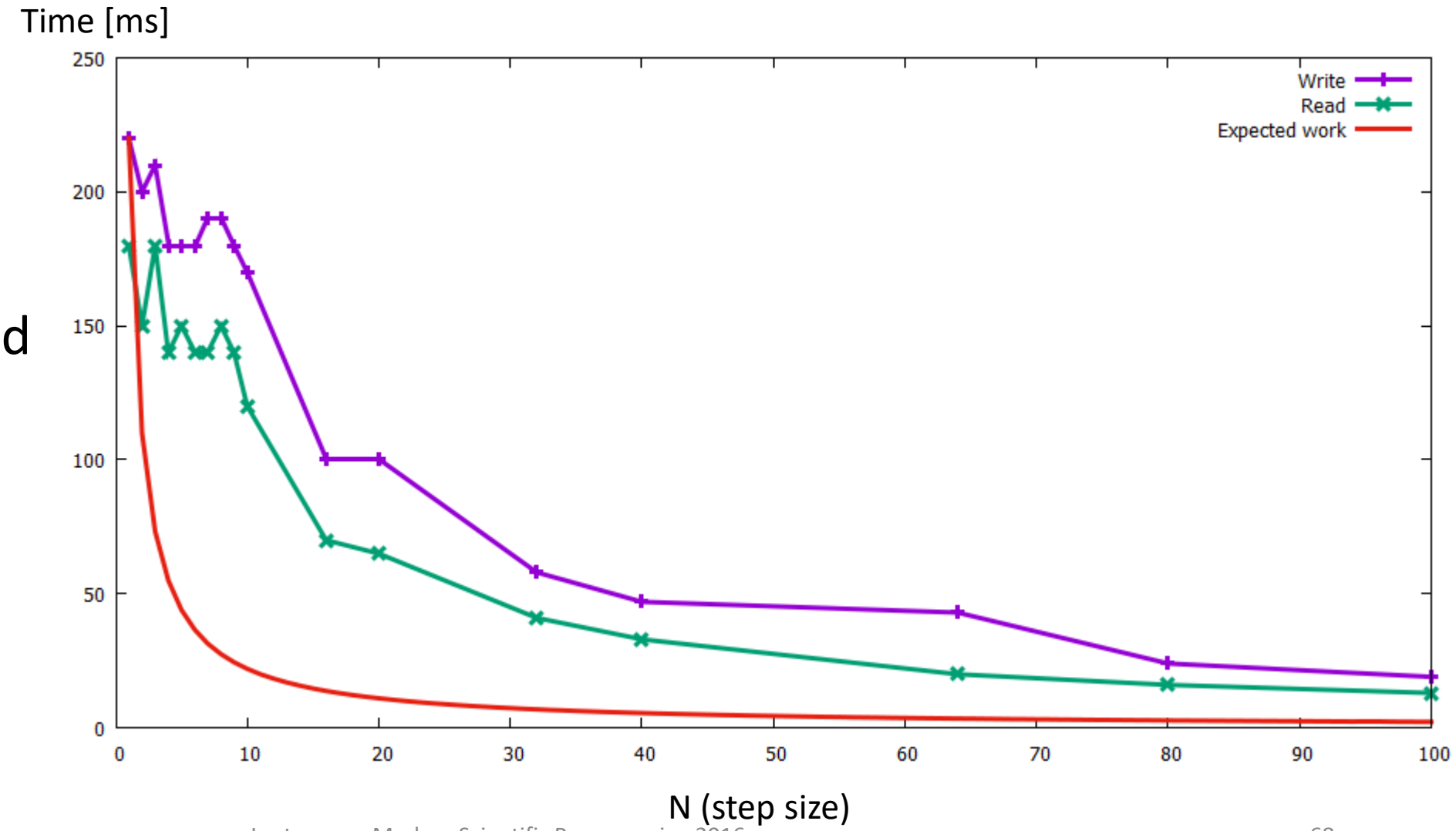
Sequential access vs gapped access (we access only every Nth element)!

Optimization – Memory!

Data access!

Sequential access vs gapped access

(we access only every Nth element)!

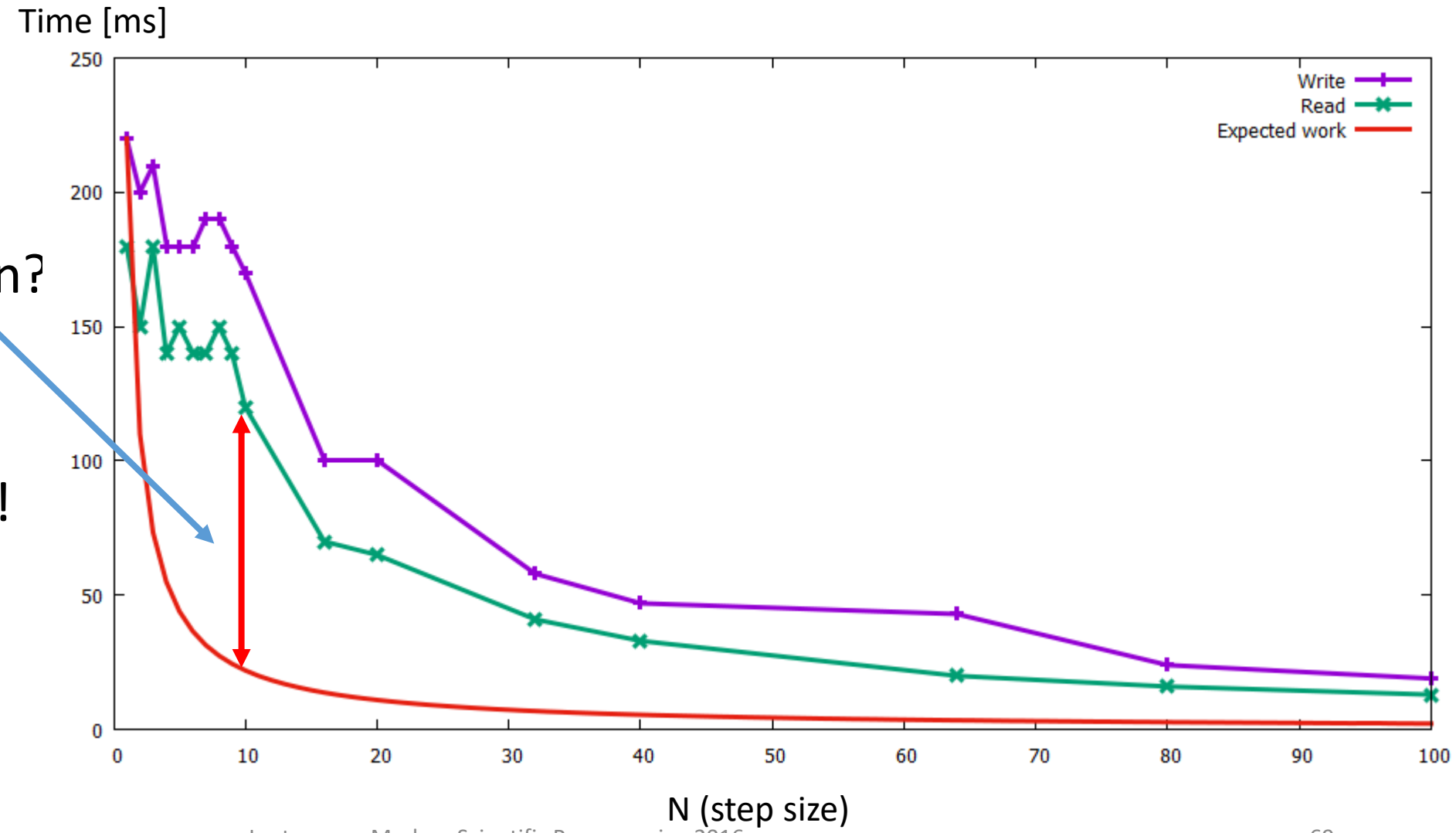


Optimization – Memory!

Data access!

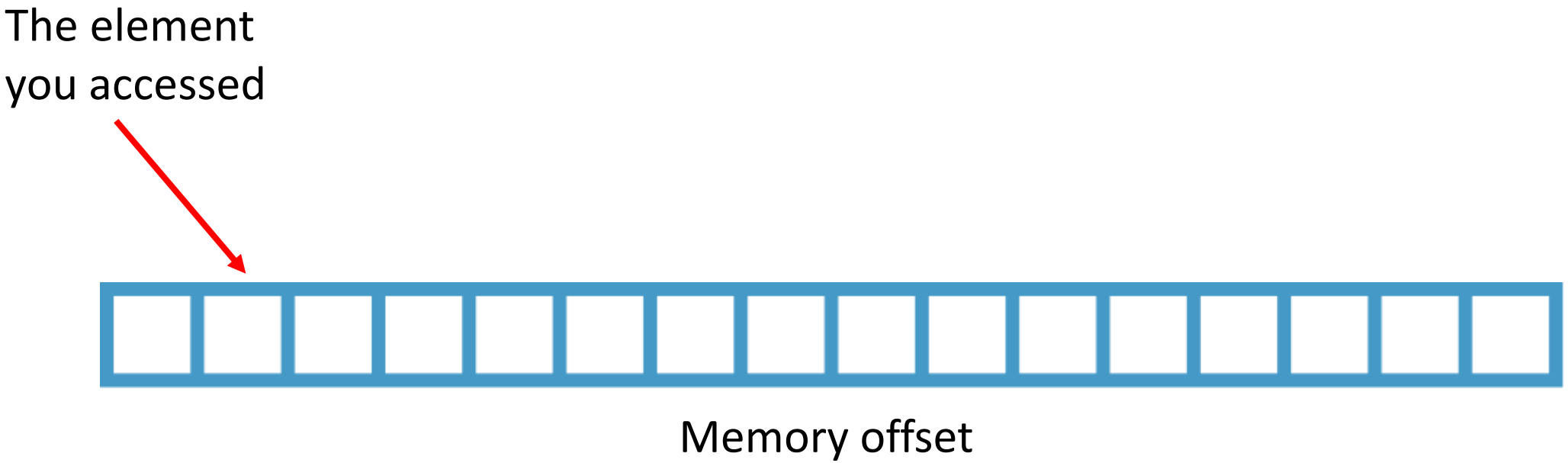
What is going on?

We are doing much less work!



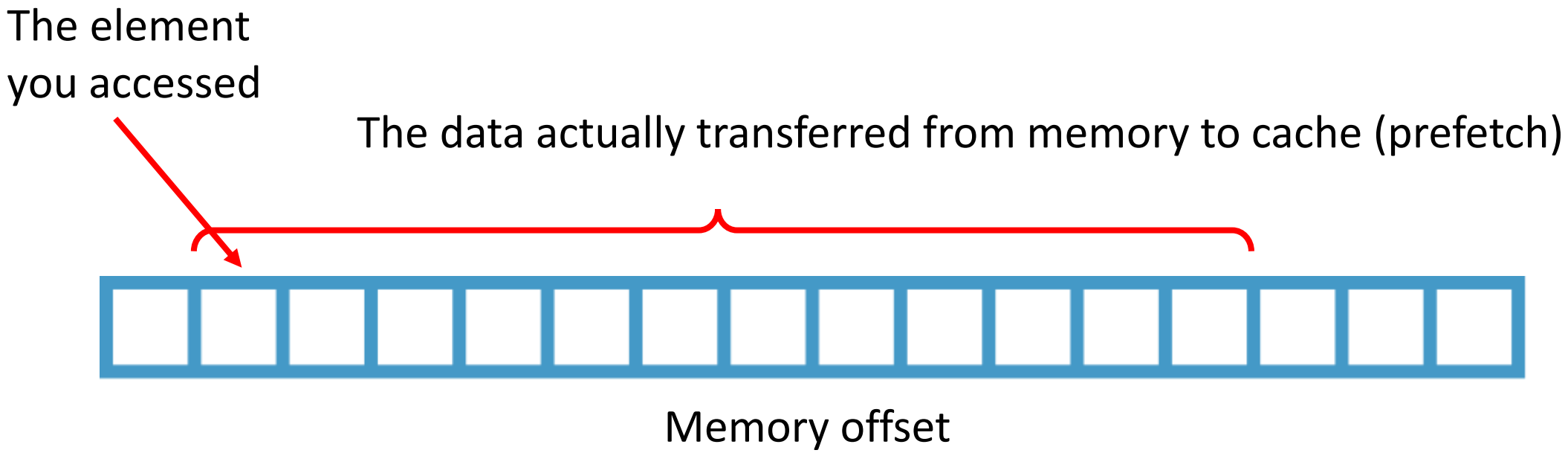
Optimization – Memory!

The memory controller and the cache is working in such a way, that the fastest operation is sequential forward access:



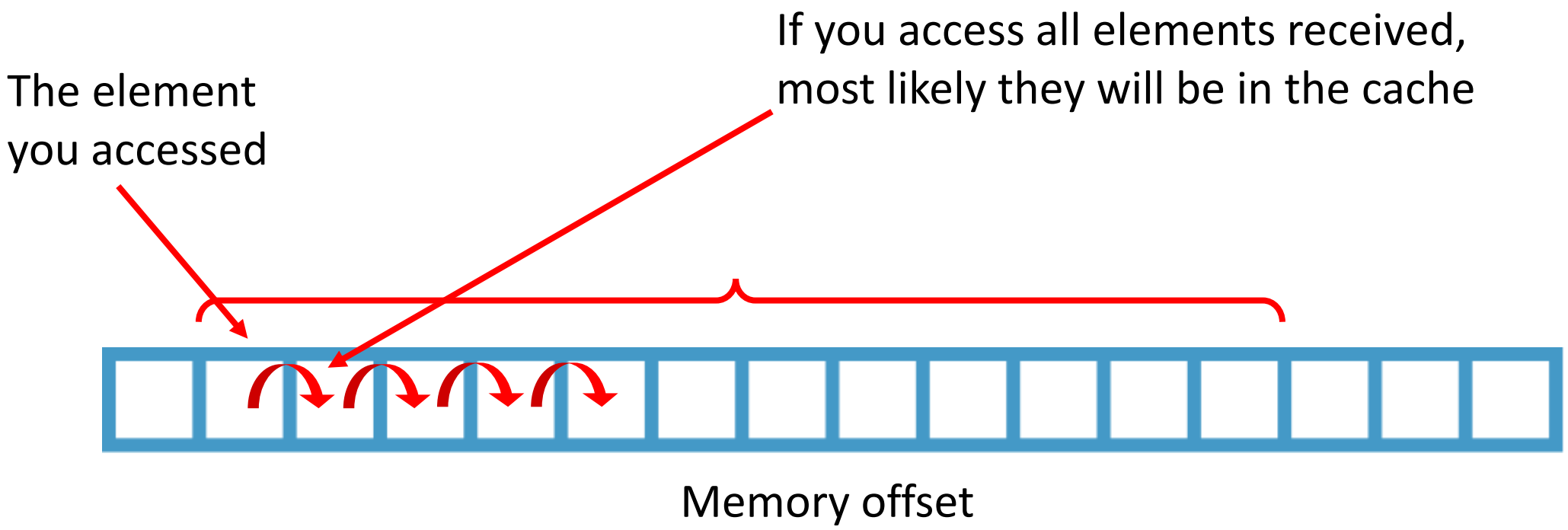
Optimization – Memory!

The memory controller and the cache is working in such a way, that the fastest operation is sequential forward access:



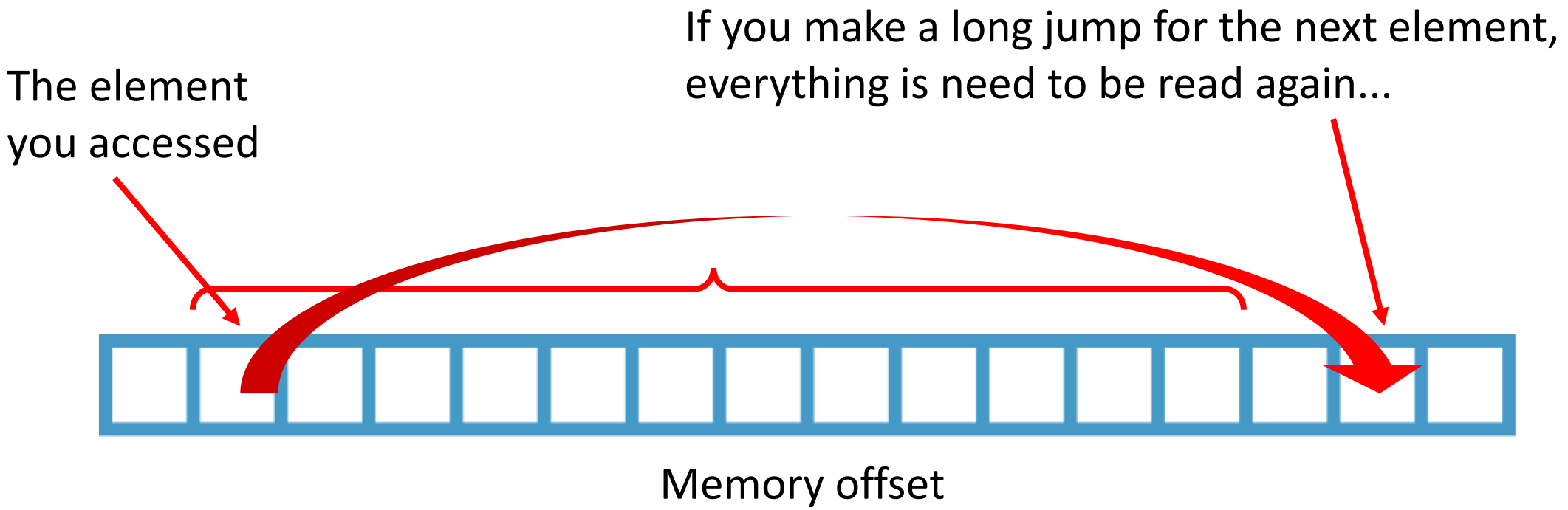
Optimization – Memory!

The memory controller and the cache is working in such a way, that the fastest operation is sequential forward access:



Optimization – Memory!

The memory controller and the cache is working in such a way, that the fastest operation is sequential forward access:



Optimization – Memory!

Typical cases of interleaved memory usage:

- Accessing only some members of structs in an array

Optimization – Memory!

Typical cases of interleaved memory usage:

- Accessing only some members of structs in an array
- Multi dimensional arrays

Optimization – Memory!

Typical cases of interleaved memory usage:

- Accessing only some members of structs in an array
- Multi dimensional arrays
- The combination of the above

Optimization – Memory!

Accessing only some members of a struct:

```
struct Particle
{
	std::array<double, 3> position;
	std::array<double, 3> velocity;
	std::array<double, 3> total_force;
	double mass, radius;
};
std::vector<Particle> particles;
```

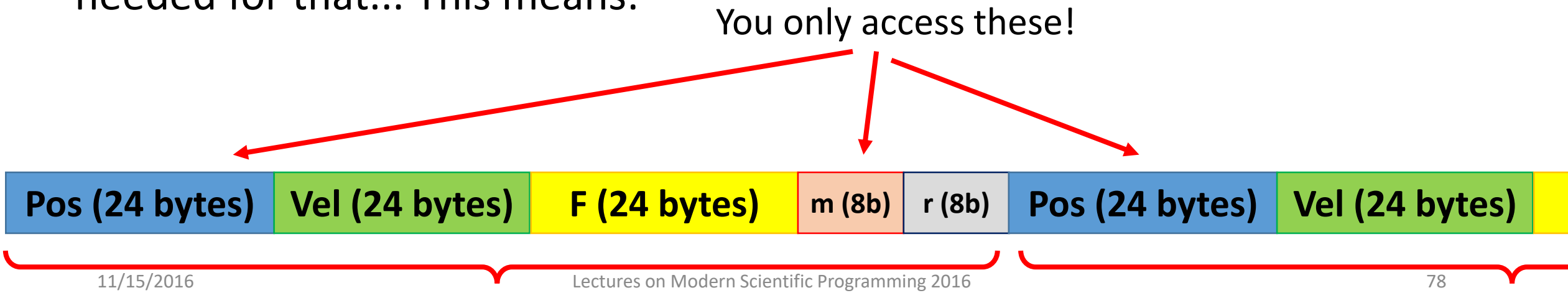
Optimization – Memory!

Accessing only some members of a struct:

Consider a force calculation...

Usually only position and mass is needed for that... This means:

```
struct Particle
{
        std::array<double, 3> position;
        std::array<double, 3> velocity;
        std::array<double, 3> total_force;
        double mass, radius;
};
std::vector<Particle> particles;
```



Optimization – Memory!

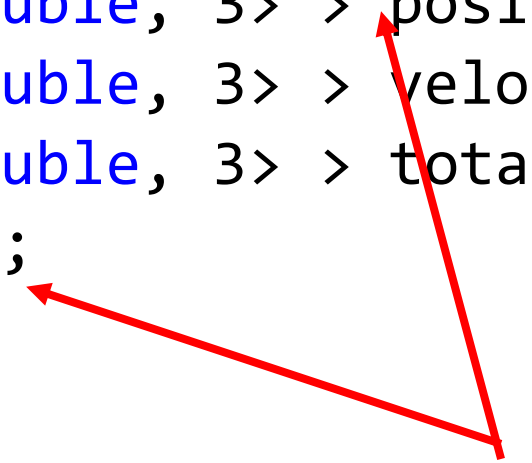
The standard solution for this is the Structures-of-Arrays arrangement:

```
struct Particles
{
      std::vector< std::array<double, 3> > positions;
      std::vector< std::array<double, 3> > velocities;
      std::vector< std::array<double, 3> > total_forces;
      std::vector<double> masses;
      std::vector<double> radii;
};
Particles particles;
```

Optimization – Memory!

The standard solution for this is the Structures-of-Arrays arrangement:

```
struct Particles
{
      std::vector< std::array<double, 3> > positions;
      std::vector< std::array<double, 3> > velocities;
      std::vector< std::array<double, 3> > total_forces;
      std::vector<double> masses;
      std::vector<double> radii;
};
Particles particles;
```



Now, these arrays will be accessed sequentially!

Optimization – Memory!

Multidimensional arrays!

- Example: naive matrix multiplication (800x800)

  with different index ordering all matrices stored as row-major:

$$C_{ik} = A_{ij} B_{jk}$$
$$D_{ik} = A_{ij} B_{kj}$$
$$E_{ik} = A_{ji} B_{jk}$$
$$F_{ki} = A_{ji} B_{jk}$$

Optimization – Memory!

Multidimensional arrays!

- Example: naive matrix multiplication

  with different index ordering all matrices stored as row-major:

$$C_{ik} = A_{ij}B_{jk} \qquad 1.93 \text{ sec}$$
$$D_{ik} = A_{ij}B_{kj} \qquad 1.17 \text{ sec}$$
$$E_{ik} = A_{ji}B_{jk} \qquad 7.00 \text{ sec}$$
$$F_{ki} = A_{ji}B_{jk} \qquad 7.04 \text{ sec}$$

Optimization – Memory!

Multidimensional arrays!

- Example: rank 3 tensor-matrix multiplication (400x400x400) with different index ordering still row-major:

$$C_{ijk} = A_{ijl}B_{kl}$$
$$D_{ijk} = A_{ijl}B_{lk}$$
$$E_{ijk} = A_{ilj}B_{kl}$$
$$F_{ijk} = A_{lij}B_{kl}$$

Optimization – Memory!

Multidimensional arrays!

- Example: rank 3 tensor-matrix multiplication (400x400x400) with different index ordering still row-major:

$$C_{ijk} = A_{ijl}B_{kl} \quad 58 \text{ sec}$$
$$D_{ijk} = A_{ijl}B_{lk} \quad 50 \text{ sec}$$
$$E_{ijk} = A_{ilj}B_{kl} \quad 58 \text{ sec}$$
$$F_{ijk} = A_{lij}B_{kl} \quad 70 \text{ sec}$$

Optimization – Memory!

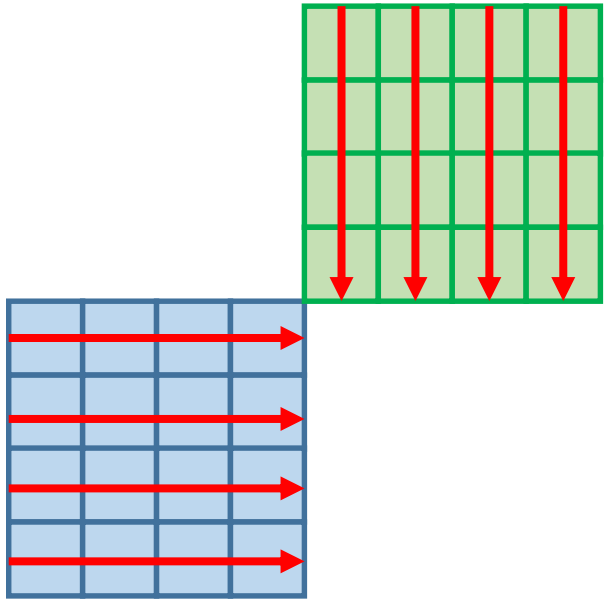
Multidimensional arrays!

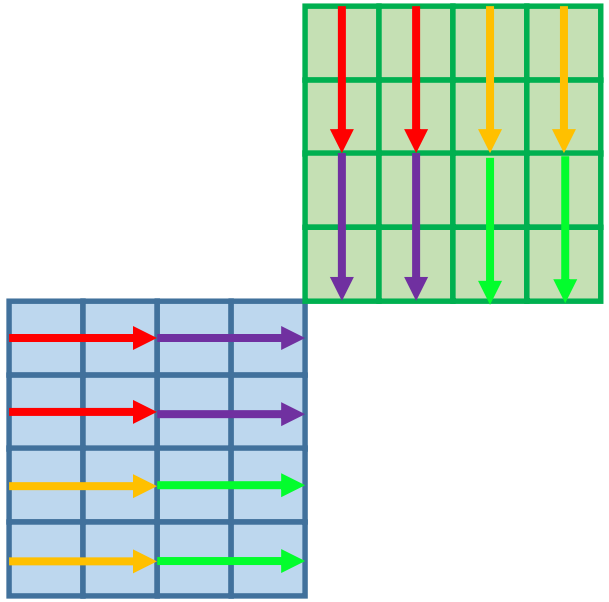
- Solution: block-wise traversal!

Optimization – Memory!

Multidimensional arrays!

- Solution: block-wise traversal!





Optimization – Memory!

Naive traversal in code:

```
for(int i=0; i<N; ++i)
{
   for(int j=0; j<N; ++j)
   {
      double sum = 0.0;
      for(int k=0; k<N; ++k){ sum += A[i*N+k] * B[k*N+j]; }
      C[i*N+j] = sum;
   }
}
```

Optimization – Memory!

Block-wise traversal in code:

```
for(int bi=0; bi<Bs; ++bi){ //block index 1
   for(int bj=0; bj<Bs; ++bj){ //block index 2
      for(int bk=0; bk<Bs; ++bk){ //block index 3

         auto i0 = bi * b; auto j0 = bj * b; auto k0 = bk * b;

         for(int i=0; i<b; ++i ){ auto ii = i0 + i;
            for(int j=0; j<b; ++j ){ auto jj = j0 + j; double sum = 0.0;
               for(int k=0; k<b; ++k ){ sum += A[i*N+k0+k] * B[(k0+k)*N+j]; }
               C[i*N+j] += sum;
            }
         }
      }
   }
}
```

Optimization – Memory!

Block-wise traversal timings:
N = 1024

$$C_{ik} = A_{ij} B_{jk}$$

Block size	Time [seconds]
1	74
2	17.1
4	5.8
8	2.7
16	2.9
32	2.7
64	**2.48**
128	2.6
256	2.6
512	15.5
1024 (equivalent to naive)	36.1

Optimization – Memory!

CAUTION: Pitfall danger!

Optimization – Memory!

Large powers of two strides can kill the cache mechanism!

Why?

- Cache is split into lines
- When data is being stored into the cache the destination line is selected by looking at the last few bits of the address

Optimization – Memory!

Large powers of two strides can kill the cache mechanism!

Why?

- Cache is split into lines

- When data is being stored into the cache the destination line is selected by looking at the **last few bits of the address**

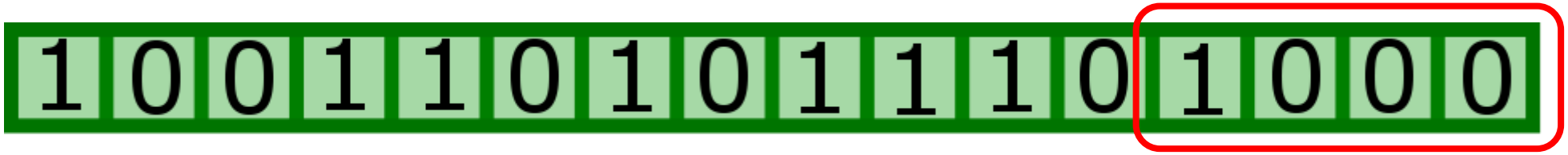
Optimization – Memory!

Large powers of two strides can kill the cache mechanism!

Why?

- Cache is split into lines

- When data is being stored into the cache the destination line is selected by looking at the **last few bits of the address**

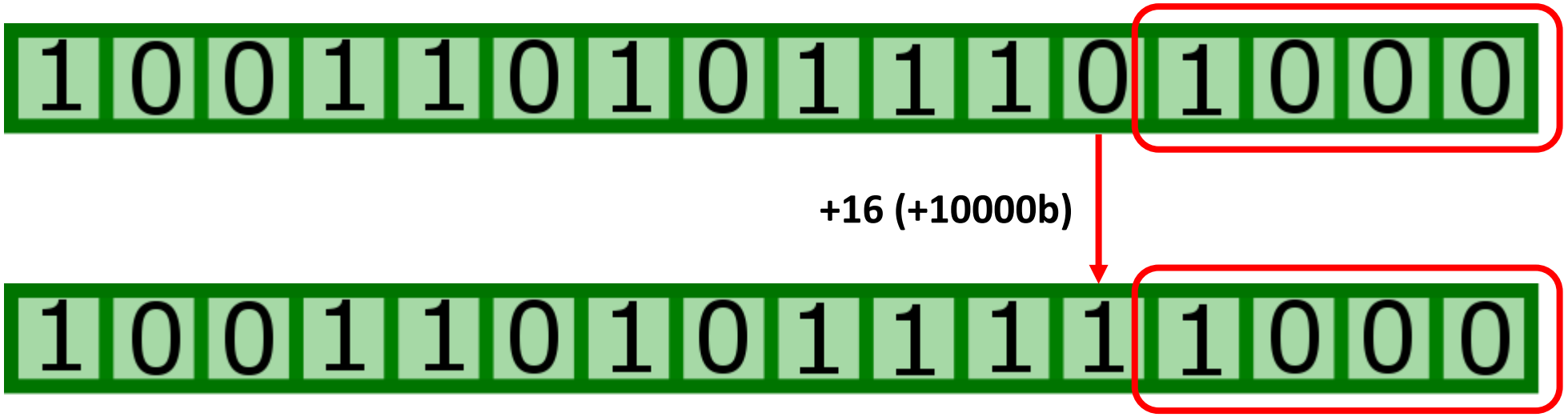


Optimization – Memory!

Large powers of two strides can kill the cache mechanism!

Why?

- If the stride in this example is 16 (binary: 10000)

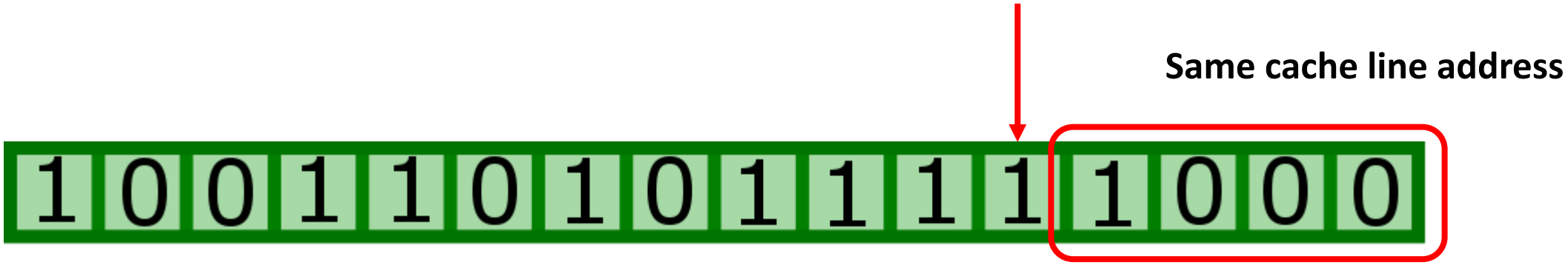


Optimization – Memory!

Large powers of two strides can kill the cache mechanism!

Why?

- If the stride in this example is 16 (binary: 10000)
- The new data will overwrite exactly the earlier one, although there would be free space in the cache!



Optimization – Memory!

Same matrix multiplication again:

N = 1024 Block size	Time [seconds]
1	74
2	17.1
4	5.8
8	2.7
16	2.9
32	2.7
64	**2.48**
128	2.6
256	2.6
512	15.5
1024	36.1

Optimization – Memory!

Same matrix multiplication again:

N = 1024 Block size	Time [seconds]
1	74
2	17.1
4	5.8
8	2.7
16	2.9
32	2.7
64	**2.48**
128	2.6
256	2.6
512	15.5
1024	36.1

N = 1023 Block size	Time [seconds]
1	10.4
3	2.8
11	1.8
13	**1.7**
33	1.8
93	2.7
341	2.9
1023	6.8

Optimization – Memory!

Same matrix multiplication again:

N = 1024 Block size	Time [seconds]
1	74
2	17.1
4	5.8
8	2.7
16	2.9
32	2.7
64	**2.48**
128	2.6
256	2.6
512	15.5
1024	36.1

N = 1023 Block size	Time [seconds]
1	10.4
3	2.8
11	1.8
13	**1.7**
33	1.8
93	2.7
341	2.9
1023	6.8

N = 1025 Block size	Time [seconds]
1	10.9
5	2.1
25	1.8
41	**1.7**
205	2.6
1025	8.5

Optimization – Memory!

Another example: Massive convolution

Image data:

- 512 x 512 x 32 channels

Kernel

- 128 different 3 x 3 x 32 channel kernels

The 512x512 images convolved with the 3x3 kernels and summed over the channels, repeated for all 128 different kernels

Optimization – Memory!

Another example: Massive convolution

Kernel storage order (row major): filter -> y -> x -> channel

Image storage order (row major): y -> x -> channel

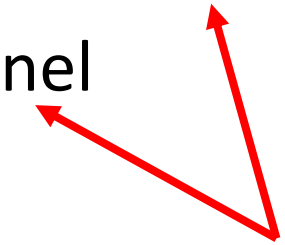
Only x direction is block grouped, because x-y could not fit in the cache at the same time.

Optimization – Memory!

Another example: Massive convolution

Kernel storage order (row major): filter -> y -> x -> channel

Image storage order (row major): y -> x -> channel



Channel should be the last, the contiguous, so the frequent summation does not suffer

Only x direction is block grouped, because x-y could not fit in the cache at the same time.

Optimization – Memory!

Another example: Massive convolution

Loop order:

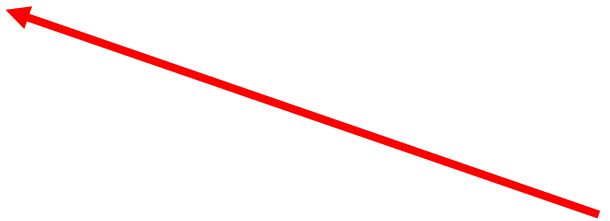
- Image y, image x block
    - Kernel (128)
        - Kernel y-x (3x3)
            - Channel (32)
                - Image x in block

Optimization – Memory!

Another example: Massive convolution

Loop order:

- Image y, image x block
  - Memcopy to temporary image array [3 x 32 x blocksize]
  - Kernel (128)
    - Kernel y-x (3x3)
      - Channel (32)
        - Image x in block



Turns out, that the image stride is so big, that it is much better to memcopy together 3 lines of the image
(3 lines of all 32 channels)

Optimization – Memory!

Short summary:

- Block traversal is much better than naive traversal
- Keep the strides small
- Keep the strides away from powers of two!

- Applies to:
  - Linear Algebra
  - Finite differences
  - Image processing

Optimization – Memory!

- Still memory access and locality

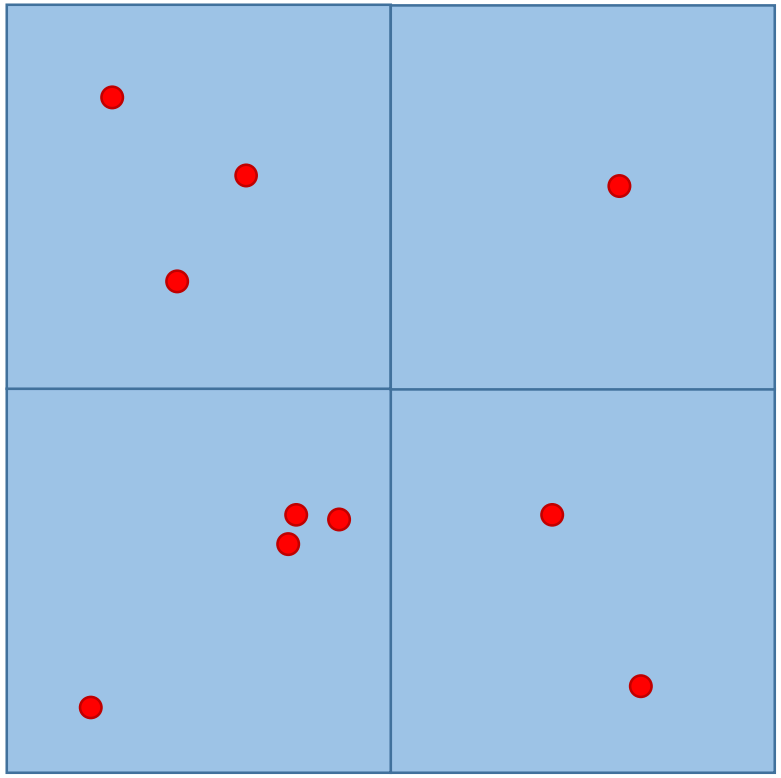
- The never ending story☺

Optimization – Memory!

- N-body simulation.
- See last year for a discussion from the beginning
- But last year we did not go into space partitioning...

Optimization – Space partitioning

- Split the simulation area into blocks
- Each block has an array of particles that are physically inside it
- Periodically update the blocks to account for particles leaving / entering



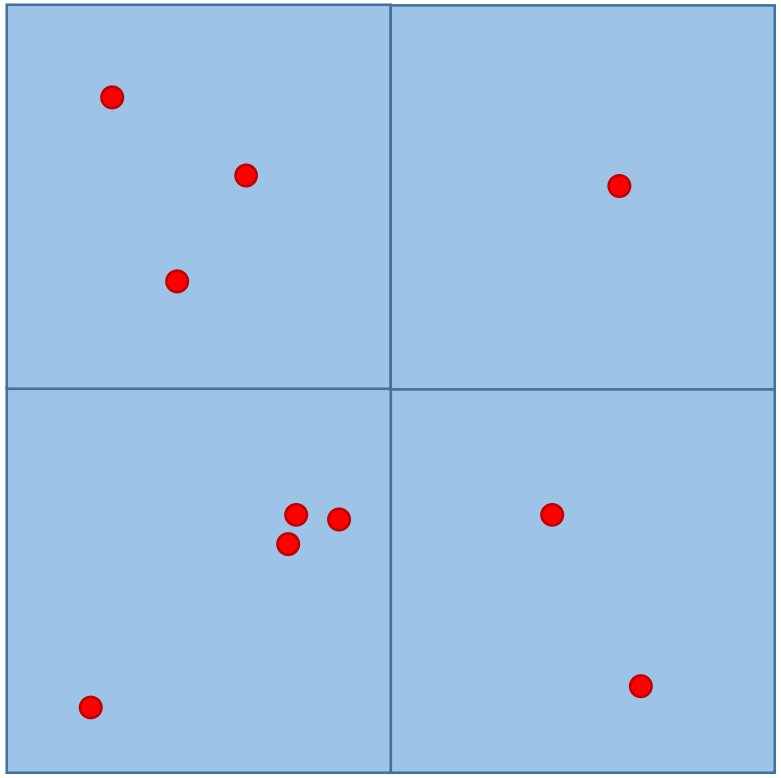
Optimization – Space partitioning

Expected improvements:

- Only calculate interaction inside blocks
- N^2 / 2 cost reduces greatly

Expected deteriorations:

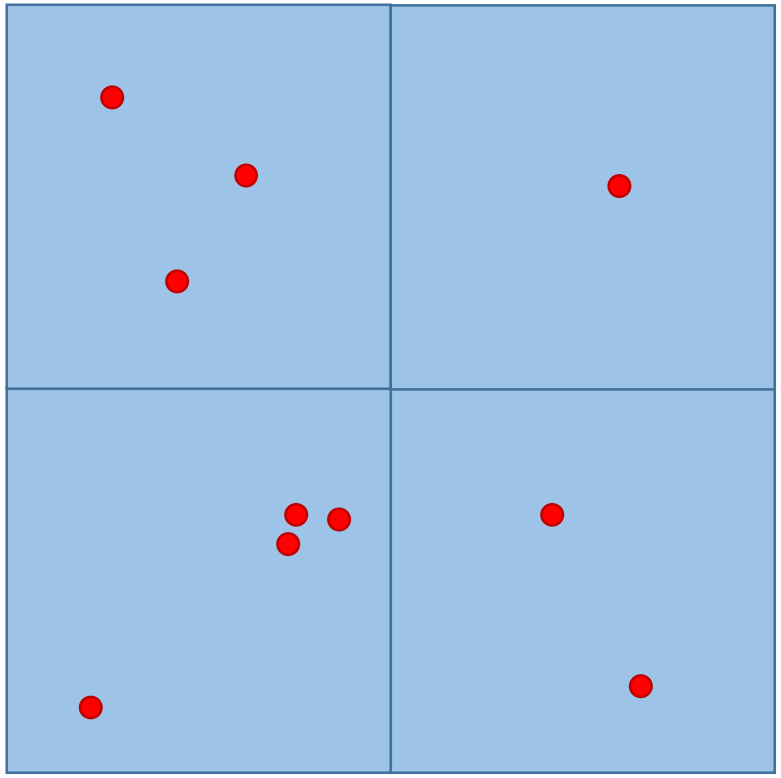
- Force calculation looses precision



Optimization – Space partitioning

But there are more improvements!

- More data locality inside the blocks
  - Better cache usage
- Easier multi-threading
  (less and more predictable data races to avoid)
  - More efficient data reductions
    (like correlation functions)



Optimization – Space partitioning

Outline of one simulation step (N_cell = 30^2, N_part = N_total / N_cell):

1. Place particles into cells (Cost: $N_{cell}N_{part}$)
2. Inter-cell pair interactions (Cost: $N_{cell}N_{part}^2/2$)
3. Exact interaction of nearest neighbor cells (Cost: $4N_{cell}N_{part}^2/2$)
4. Calculate center-of-mass data (Cost: $N_{cell}N_{part}$)
5. Center-of-mass interaction of cells (Cost: N_{cell}^2)
6. Step particles (Cost: $N_{cell}N_{part}$)

Optimization – Space partitioning

Outline of one simulation step
(N_cell = 30^2, N_part = 25k / N_cell):

1.	Place particles into cells	1 ms (2%)
2.	Inter-cell pair interactions	5.5 ms (11%)
3.	Exact interaction of nearest neighbor cells	33 ms (67 %)
4.	Calculate center-of-mass data	1.3 ms (3%)
5.	Center-of-mass interaction of cells	7.4 ms (15%)
6.	Step particles	0.8 ms (1.7%)

Total: 49 ms

Summary

This talk mainly focused on memory access optimizations

These give a large percent of performance losses that can be addressed by average amount of work and compilers wont do it automatically

Take home message:

Linear forward contiguous access

Hardware prefers spatiotemporal locality of data access