Modern Architecture

Code Optimization Workshop | 27.6.2022 | Jonathan Coles

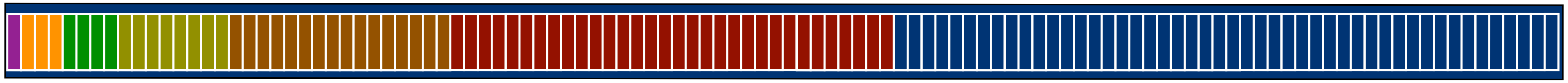
Data Storage, Data Movement, and Data Processing

How is data stored?

How is data moved between types of storage?

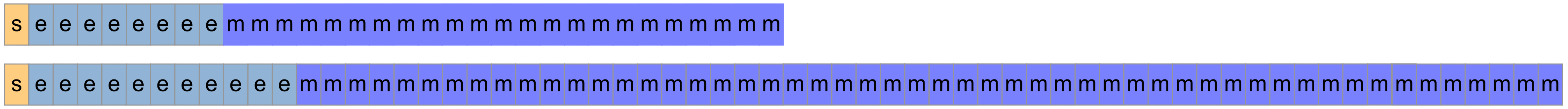
How is data processed?

System Memory and Datatypes



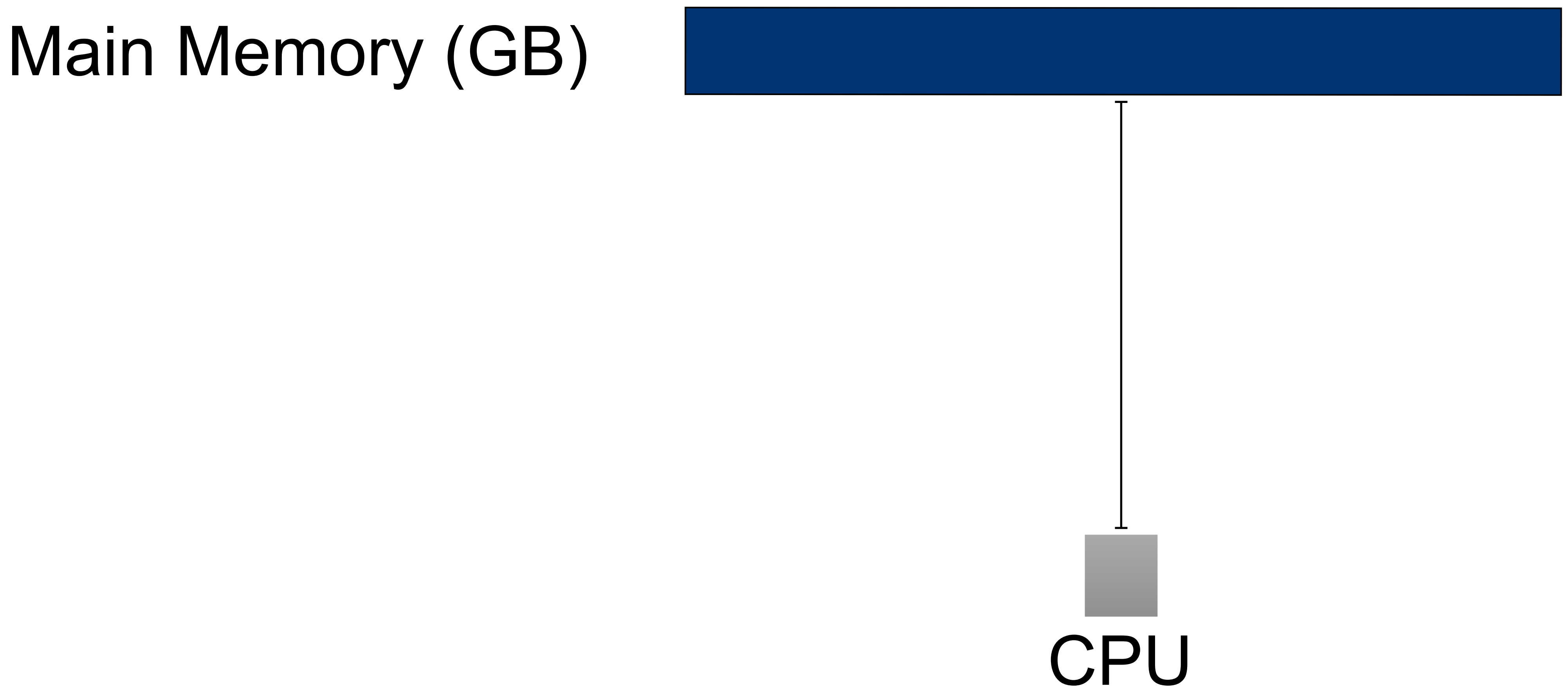
🟪	1 byte, 8 bits, 1 character
🟧	4 bytes, 32 bits, 1 integer, 1 single-precision floating-point number
🟩	8 bytes, 64 bits, 1 long (integer), 1 double-precision floating-point number
🟦	general memory

IEEE Standard for Floating-Point Arithmetic (**IEEE 754**)

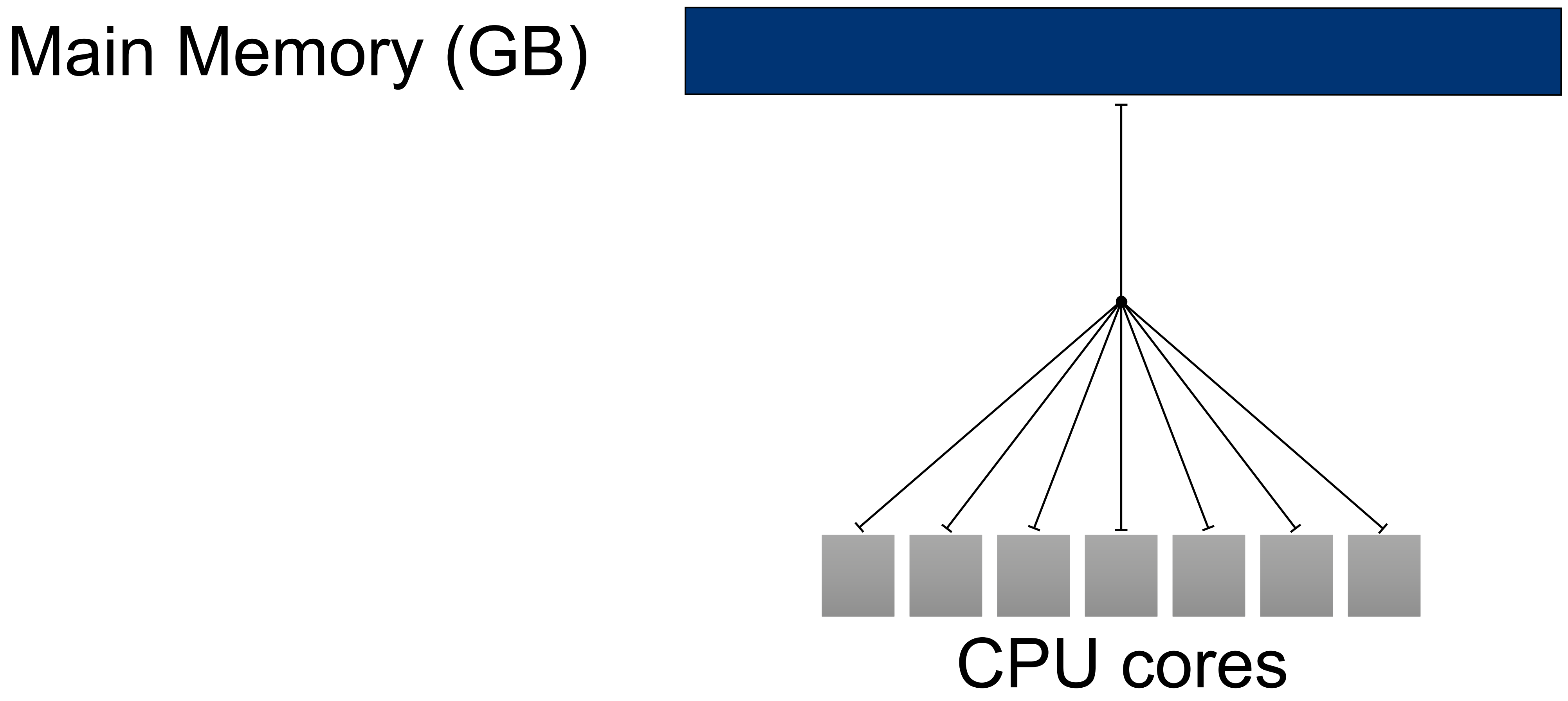

s e e e e e e e e m m m m m m m m m m m m m m m m m m m m m m m
s e e e e e e e e e e e m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m m

What Every Computer Scientist Should Know About Floating-Point Arithmetic, by David Goldberg

Memory hierarchy



Memory hierarchy

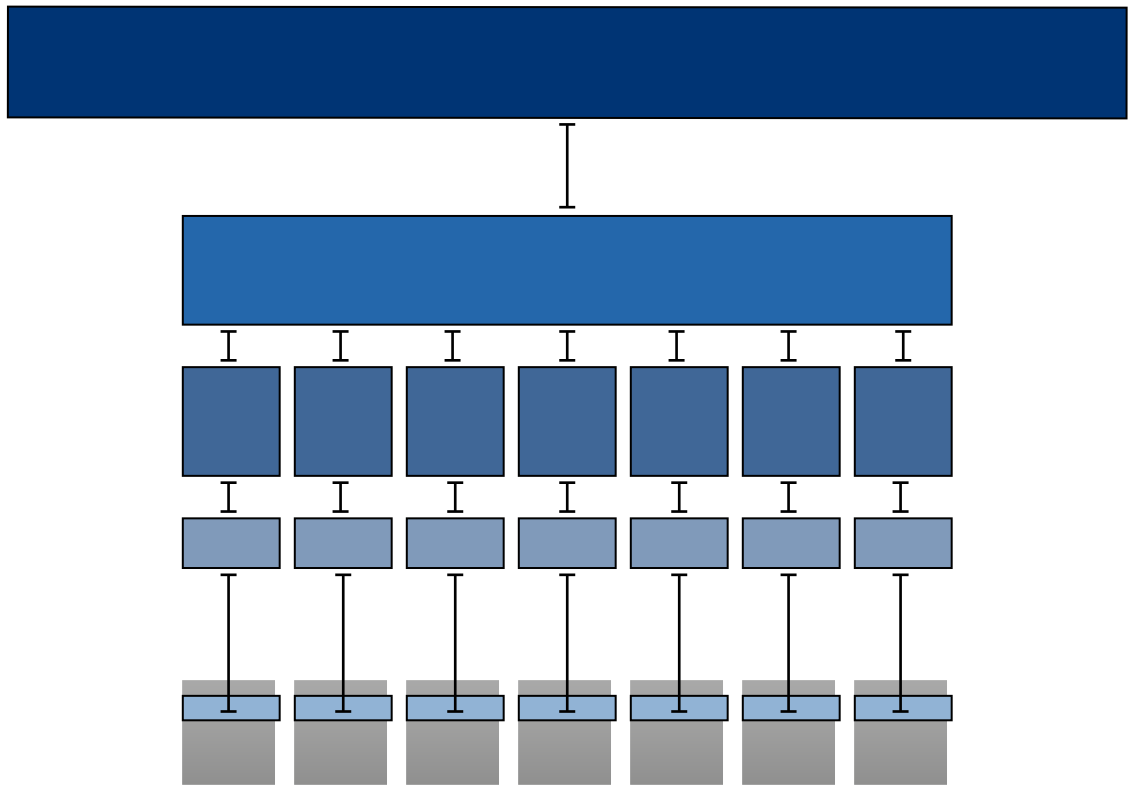


Memory hierarchy

Main Memory (GB)

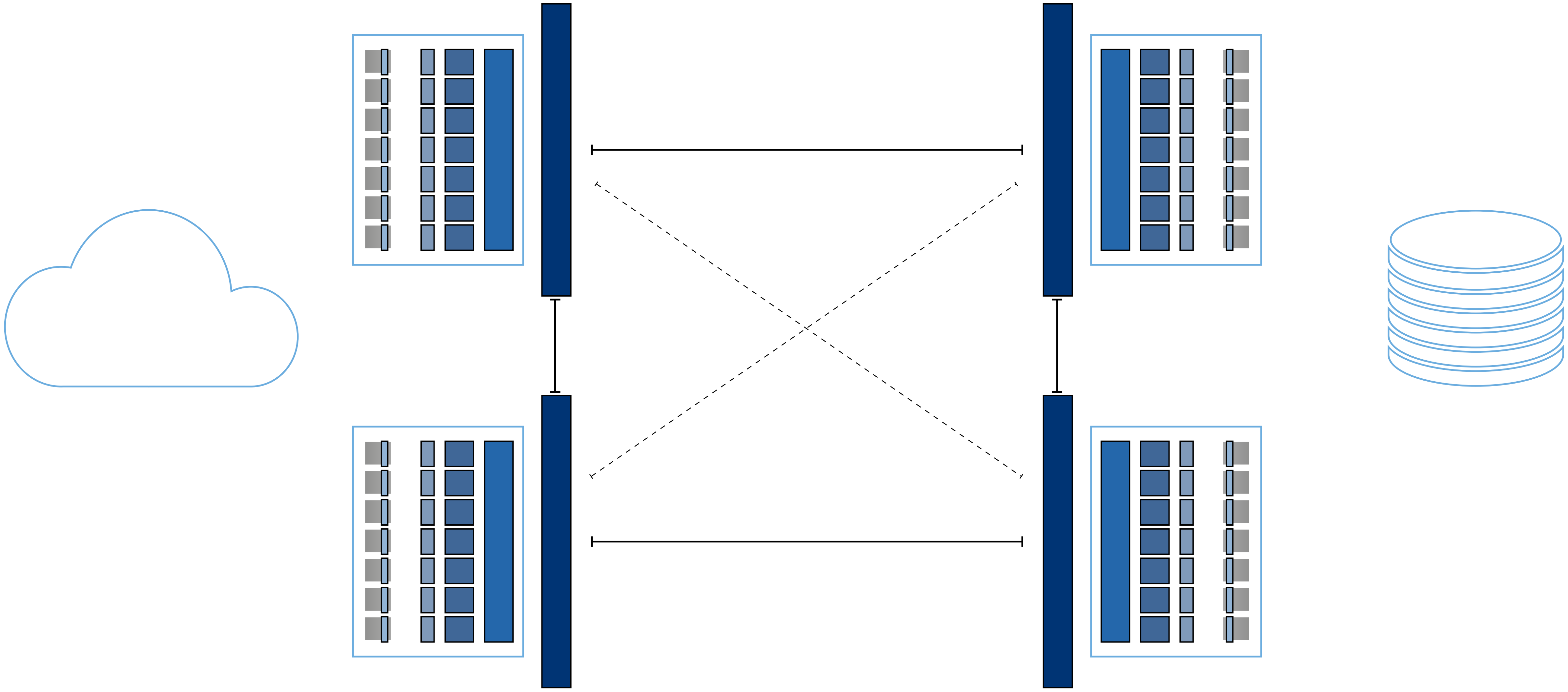
Cache
L3/LL (MB)
L2 (~MB)
L1 (KB)

Registers (B)

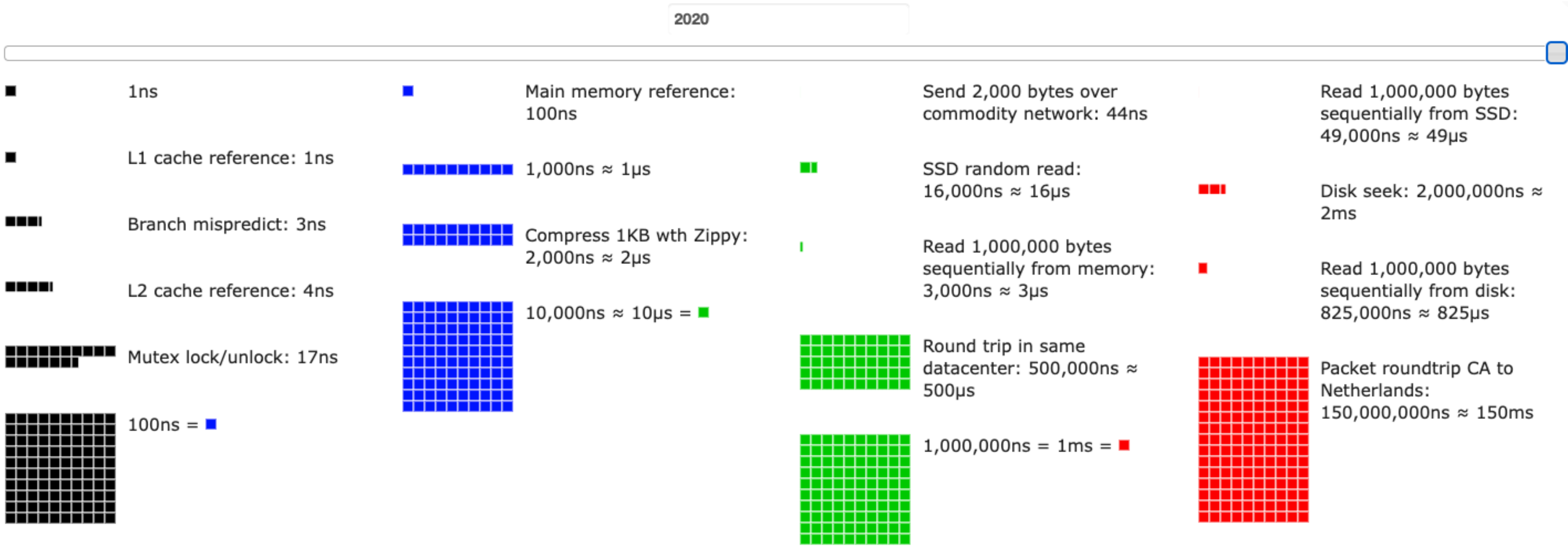


CPU cores

Memory hierarchy



Memory hierarchy



2020

1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 2,000ns ≈ 2µs

10,000ns ≈ 10µs = 🟩

Send 2,000 bytes over commodity network: 44ns

SSD random read: 16,000ns ≈ 16µs

Read 1,000,000 bytes sequentially from memory: 3,000ns ≈ 3µs

Round trip in same datacenter: 500,000ns ≈ 500µs

1,000,000ns = 1ms = 🟥

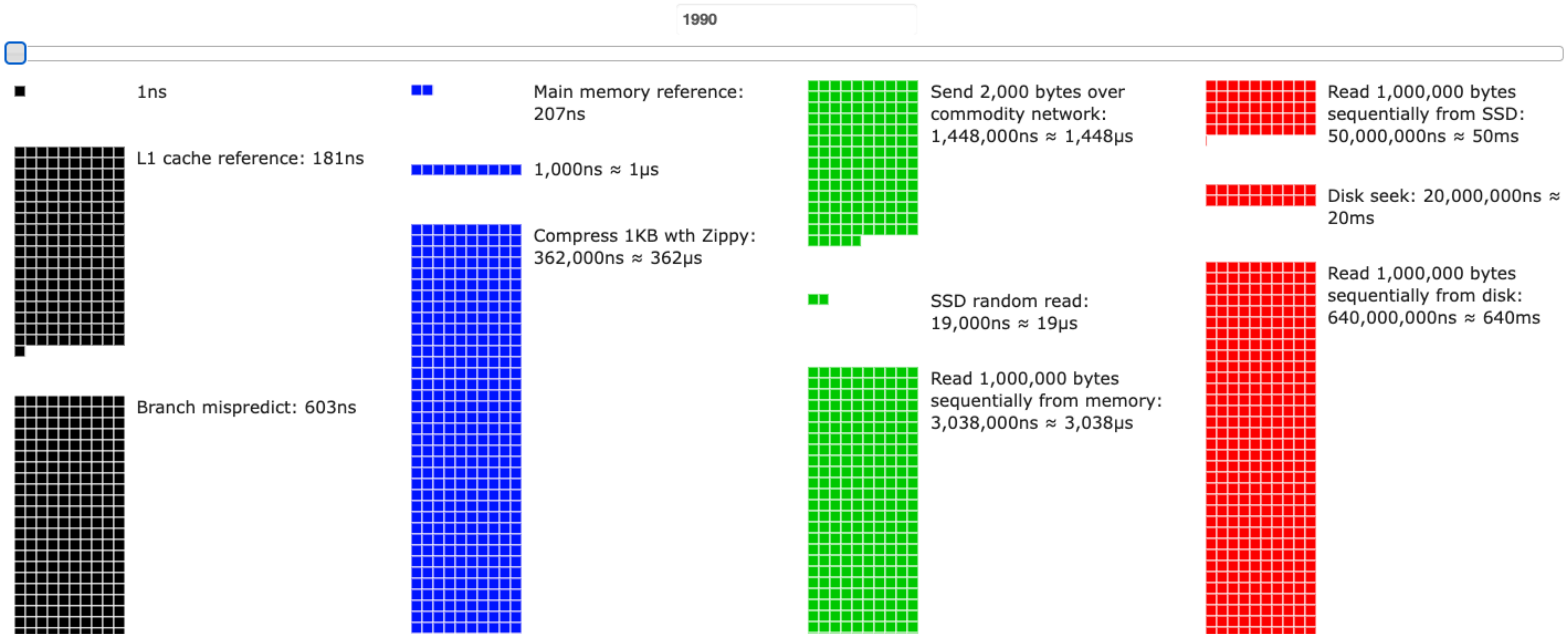
Read 1,000,000 bytes sequentially from SSD: 49,000ns ≈ 49µs

Disk seek: 2,000,000ns ≈ 2ms

Read 1,000,000 bytes sequentially from disk: 825,000ns ≈ 825µs

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

Memory hierarchy



1990

1ns

L1 cache reference: 181ns

Branch mispredict: 603ns

Main memory reference: 207ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 362,000ns ≈ 362µs

Send 2,000 bytes over commodity network: 1,448,000ns ≈ 1,448µs

SSD random read: 19,000ns ≈ 19µs

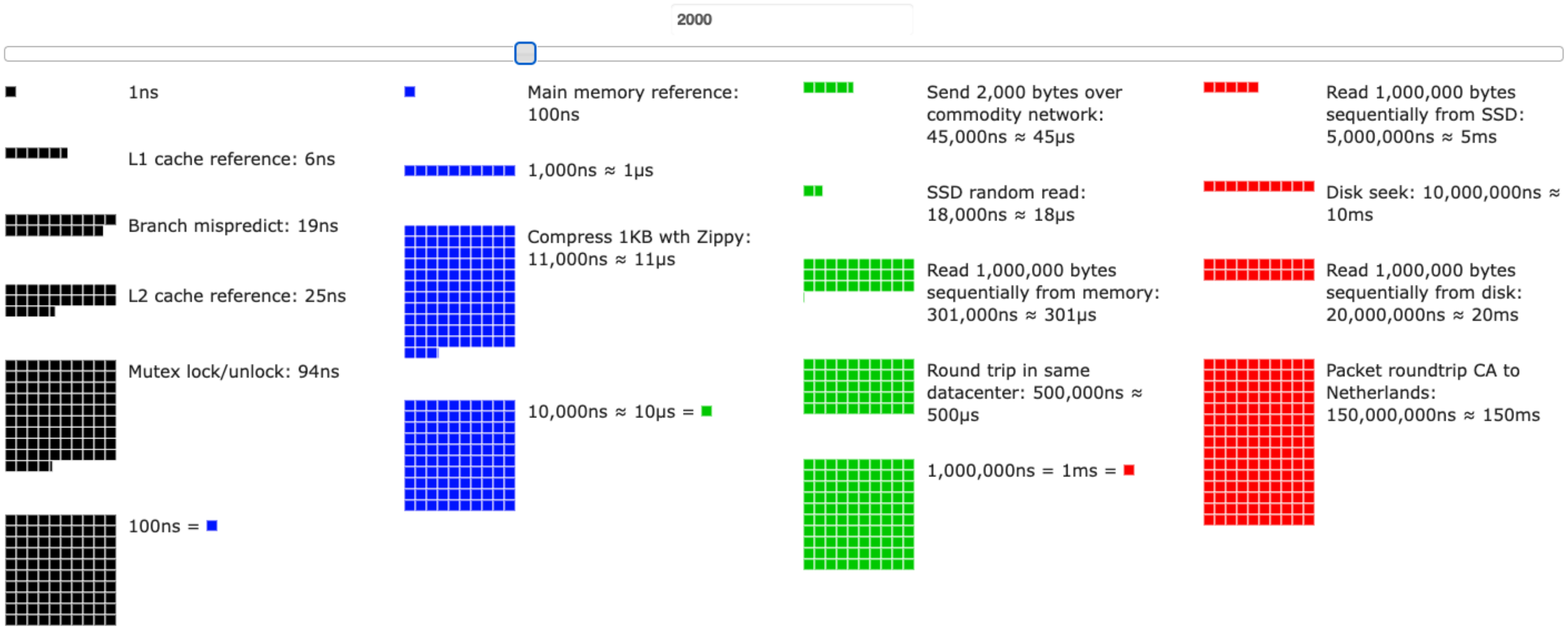
Read 1,000,000 bytes sequentially from memory: 3,038,000ns ≈ 3,038µs

Read 1,000,000 bytes sequentially from SSD: 50,000,000ns ≈ 50ms

Disk seek: 20,000,000ns ≈ 20ms

Read 1,000,000 bytes sequentially from disk: 640,000,000ns ≈ 640ms

Memory hierarchy



2000

1ns

L1 cache reference: 6ns

Branch mispredict: 19ns

L2 cache reference: 25ns

Mutex lock/unlock: 94ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1μs

Compress 1KB wth Zippy: 11,000ns ≈ 11μs

10,000ns ≈ 10μs = 🟩

Send 2,000 bytes over commodity network: 45,000ns ≈ 45μs

SSD random read: 18,000ns ≈ 18μs

Read 1,000,000 bytes sequentially from memory: 301,000ns ≈ 301μs

Round trip in same datacenter: 500,000ns ≈ 500μs

1,000,000ns = 1ms = 🟥

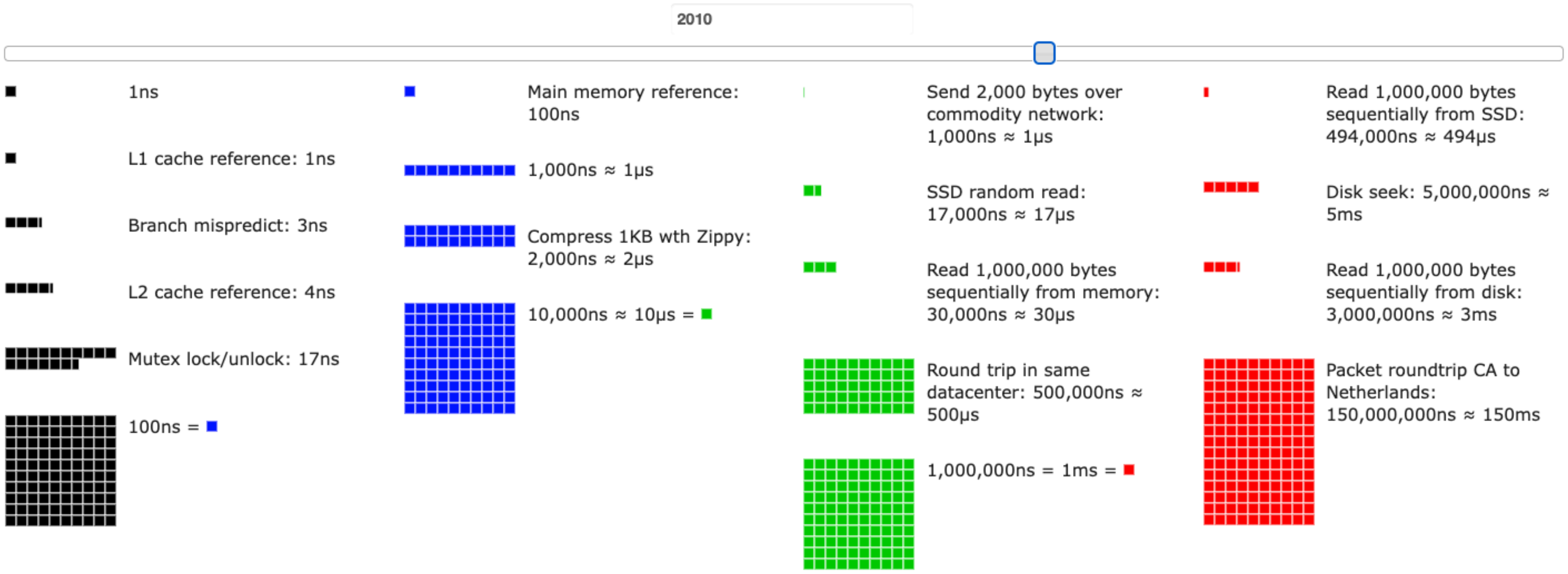
Read 1,000,000 bytes sequentially from SSD: 5,000,000ns ≈ 5ms

Disk seek: 10,000,000ns ≈ 10ms

Read 1,000,000 bytes sequentially from disk: 20,000,000ns ≈ 20ms

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

Memory hierarchy



2010

1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1μs

Compress 1KB wth Zippy: 2,000ns ≈ 2μs

10,000ns ≈ 10μs = 🟩

Send 2,000 bytes over commodity network: 1,000ns ≈ 1μs

SSD random read: 17,000ns ≈ 17μs

Read 1,000,000 bytes sequentially from memory: 30,000ns ≈ 30μs

Round trip in same datacenter: 500,000ns ≈ 500μs

1,000,000ns = 1ms = 🟥

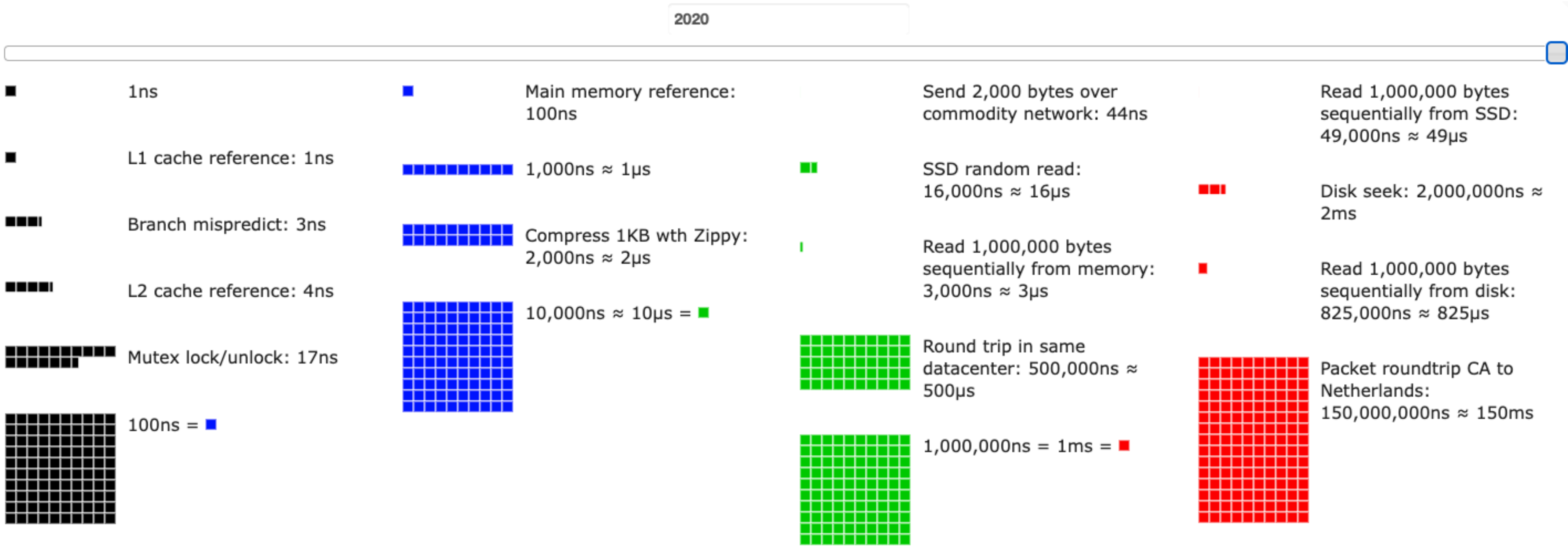
Read 1,000,000 bytes sequentially from SSD: 494,000ns ≈ 494μs

Disk seek: 5,000,000ns ≈ 5ms

Read 1,000,000 bytes sequentially from disk: 3,000,000ns ≈ 3ms

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

Memory hierarchy



2020

1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 2,000ns ≈ 2µs

10,000ns ≈ 10µs = 🟩

Send 2,000 bytes over commodity network: 44ns

SSD random read: 16,000ns ≈ 16µs

Read 1,000,000 bytes sequentially from memory: 3,000ns ≈ 3µs

Round trip in same datacenter: 500,000ns ≈ 500µs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 49,000ns ≈ 49µs

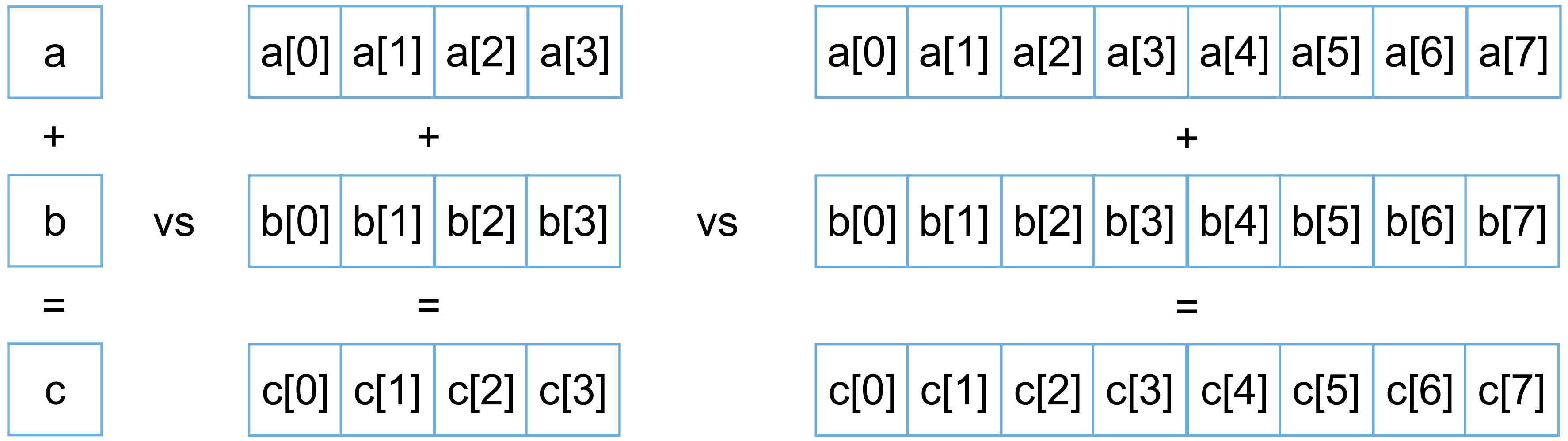
Disk seek: 2,000,000ns ≈ 2ms

Read 1,000,000 bytes sequentially from disk: 825,000ns ≈ 825µs

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

SIMD, SSE, AVX

- SIMD - Single Instruction Multiple Data
  - The same operation is applied simultaneously multiple inputs
- SSE - Streaming SIMD Extensions
  - Further iterations produced SSE2, SSE3, SSE4.1, SSE4.2
  - 128 bit XMM registers (4 floats, 2 doubles)
- AVX - Advanced Vector Extensions
  - 256 bit wide registers (YMM), supports 3 operand instructions
  - AVX-512 has 512-bit registers (ZMM)


a
+
b
=
c
vs
a[0] a[1] a[2] a[3]
+
b[0] b[1] b[2] b[3]
=
c[0] c[1] c[2] c[3]
vs
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
+
b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7]
=
c[0] c[1] c[2] c[3] c[4] c[5] c[6] c[7]

SIMD, SSE, AVX

```
#pragma omp simd aligned(a,b,c)
for (int i=0; i < N; i++)
    c[i] = a[i] * b[i];
```

```
..B1.5:
        vmovups   zmm0, ZMMWORD PTR [r11+r8*4]
        vmulps    zmm1, zmm0, ZMMWORD PTR [r10+r8*4]
        vmovups   ZMMWORD PTR [r9+r8*4], zmm1
        add       r8, 16
        cmp       r8, rdi
        jb        ..B1.5
```

SIMD, SSE, AVX

```
#pragma omp simd aligned(a,b,c)
for (int i=0; i < N; i++)
    c[i] = a[i] * b[i];
```

```
..B1.5:
        vmovups   zmm0, ZMMWORD PTR [r11+r8*4]
        vmulps    zmm1, zmm0, ZMMWORD PTR [r10+r8*4]
        vmovups   ZMMWORD PTR [r9+r8*4], zmm1
        add       r8, 16
        cmp       r8, rdi
        jb        ..B1.5
```

```
icc -O3 -xCORE-AVX512 -qopt-zmm-usage=high
```

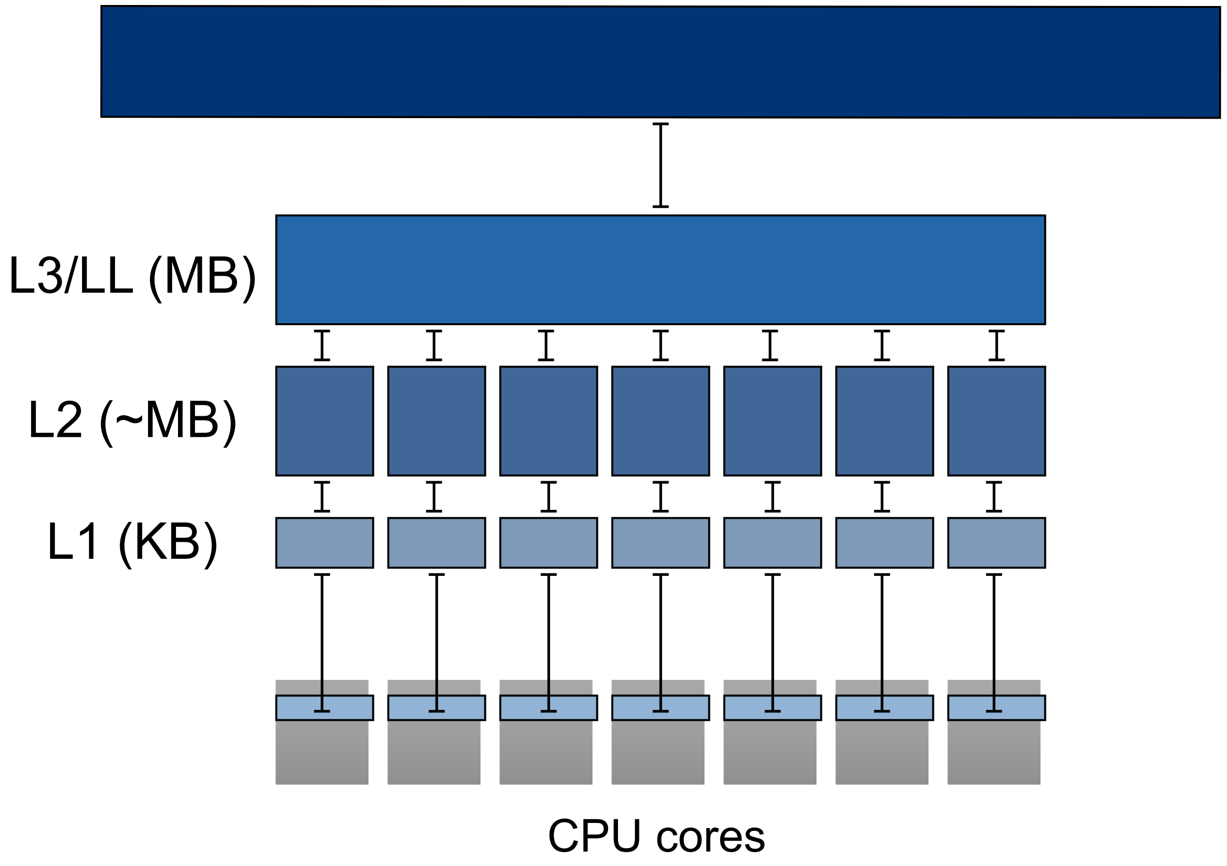
SIMD, SSE, AVX

```
#pragma omp simd aligned(a,b,c)
for (int i=0; i < N; i++)
    c[i] = a[i] * b[i];
```

- `aligned` keyword tells the compiler that the arrays have been allocated in a way that puts the address of the array on an address boundary that is optimal for loading data into a vector.
- Not necessary to achieve vectorization, but compiler can eliminate generated code to handle processing elements that occur before a natural alignment address.

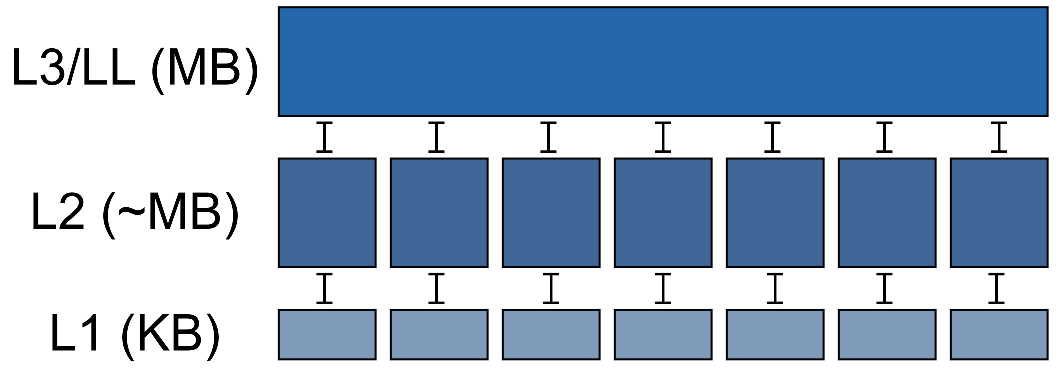
Cache

- Main memory is far from the CPU
- Access is slow compared to data processing speeds
- Cache is a smaller, but faster copy of data from main memory
- CPU manages movement of data to and from main memory and cache



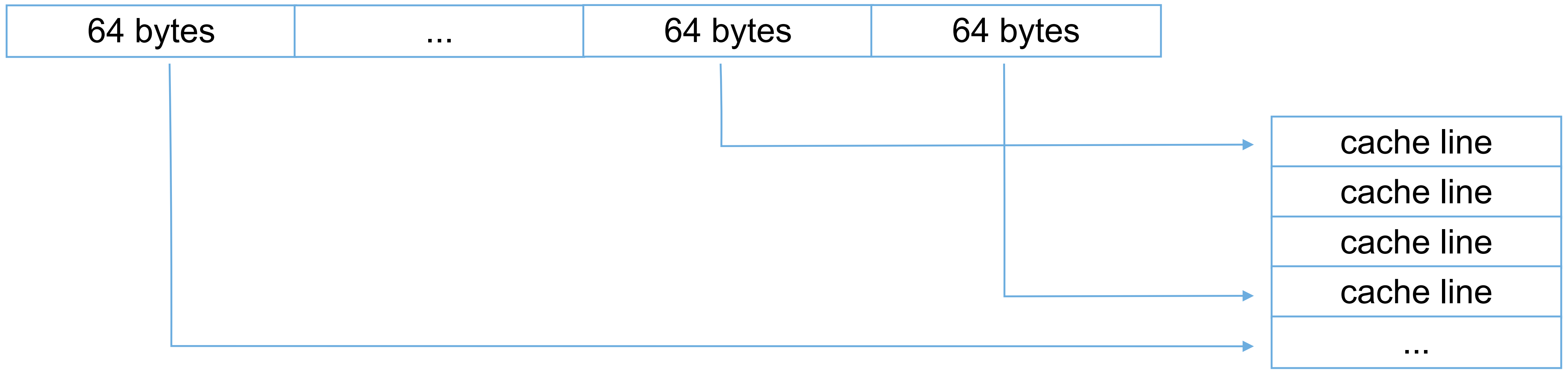
Cache

- Cache is (generally) hierarchical and duplicates data.
  - Data in L1 (level 1) is also in L2 (level 2) and L3 (level 3)
  - Data in L2 is also in L3
- Memory locations that exist in cache don't need to be copied from main memory each time they are needed.
- If a memory location is already in cache when needed that is called a cache "hit".
- If the memory location isn't in cache and needs to be copied that is a cache "miss".
- If the cache is full and a new memory location needs to be brought in, some cached data must be "evicted" and copied to the next highest level.
  - A simple eviction "policy" may be Least Recently Used (LRU)



Cache

- Copying from main memory has a high latency.
- Memory is copied 64 bytes at a time to reduce total latency.
    - **Most useful if all 64 bytes can be used once in cache!**
- 64 bytes constitutes a "cache line".
    - 16 floats, 8 doubles
- For any given cache level *n* the number of cache lines is sizeof(L*n)* / 64.
- Cache lines always aligned with memory addresses in multiples of 64 bytes.
    - Accessing a byte anywhere in a line brings in the whole line.



Cache

- Modern CPUs use *N*-way set associative cache to decide where cache lines are placed in the cache.

	way 0	way 1	way 2	way 3
set 0	cache line	cache line	cache line	cache line
set 1	cache line	cache line	cache line	cache line
set 2	cache line	cache line	cache line	cache line
set 3	cache line	cache line	cache line	cache line
...	...	...	...	...

- Memory address determines "set"
- Next available slot according to eviction policy determine "way".
- Has some implications for effective cache use. *What if all memory accesses happen on addresses that map to the same set?*

Prefetching

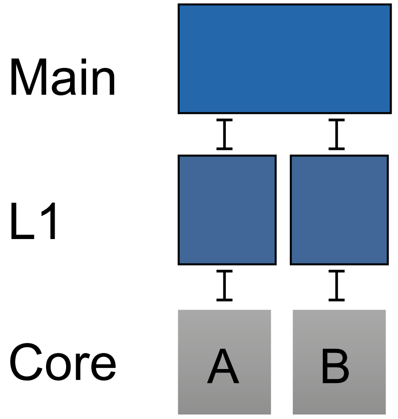
- Simple access patterns can be recognized by the CPU
- Cache lines can be brought in before they are needed.
- Overlaps computing with data copying.
- Compiler may also generate explicit prefetch instructions if it recognizes patterns.

```
for (i=0; i < N; i+=step_size)
    c[i] = a[i] * b[i];
```

False Sharing

- Cache is "coherent" on most modern systems.
- Copies of the same cache line can exist in caches of different cores.
- If a cache line is modified anywhere (any byte) by core A and then accessed by core B (any byte), the entire cache line is first copied back to main memory and then to the cache of core B before being read.
- This can lead to "false sharing" where a cache line unintentionally bounces between the caches of different cores.
- Can easily lead to a 10x drop in performance.

False Sharing



Time	Core A	Core B
1	Read memory at address 1000234	
2	**64 bytes from 1000232-1000295 loaded into cache A.**	
3	Write memory at address 1000234. Cache line marked "dirty".	
4		Read memory at address 1000280
5	**64 bytes from 1000232-1000295 written into main memory.**	
6		**64 bytes from 1000232-1000295 loaded into cache B.**
7		Write memory at address 1000280. Cache line marked "dirty".
8	Read memory at address 1000235	
9		**64 bytes from 1000232-1000295 written into main memory.**
10	**64 bytes from 1000232-1000295 loaded into cache A.**	
11	Write memory at address 1000235. Cache line marked "dirty".	

False Sharing

- Common example, manual reduction:

```
int global_array[N_THREADS];

#pragma omp parallel
{
    while (!done)
    {
        int sum=0
        for (int i=0; i < N; i++)
        {
            sum += some_data[i];
        }
        global_array[my_thread_id] = sum;
    }
}
```

False Sharing

- Possible solution: only write to different cache lines.
- Clearly wasteful. OpenMP also supports thread local variables.

```
int global_array[N_THREADS][16];

#pragma omp parallel
{
    while (!done)
    {
        int sum=0
        for (int i=0; i < N; i++)
        {
            sum += some_data[i];
        }
        global_array[my_thread_id] = sum;
    }
}
```

Hardware aligned software design

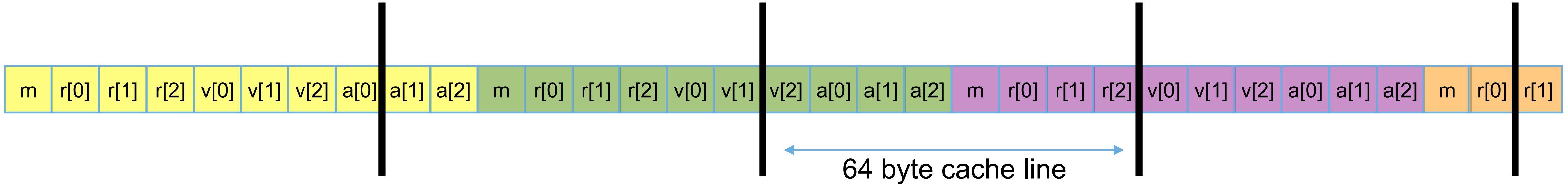
- Data access patterns and cache reuse is crucial for achieving maximum FLOPs of memory-bound applications.
- Goal is to perform maximum number of floating-point operations for each byte transferred.

Hardware aligned software design

- Clearly, not all algorithms map optimally to the underlying hardware.
- Hardware prefers simple, predictable, linear processing.
  - Compare: linear array access, random access, or linked lists.
- Understanding these principles may allow hardware features to be utilized.

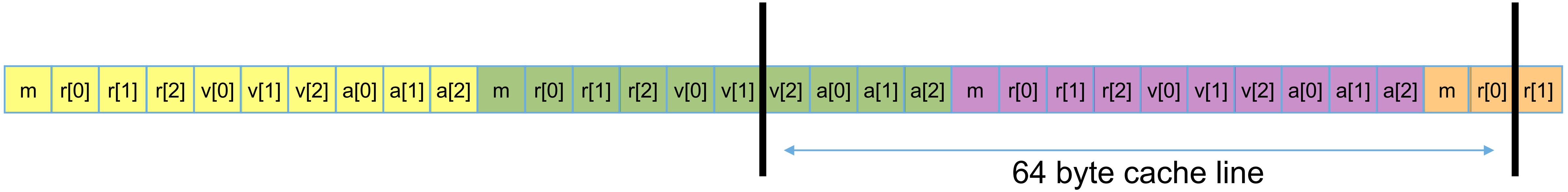
Data organization

```
struct Particle
{
  double m;
  double r[3];
  double v[3];
  double a[3];
} p[N];
```



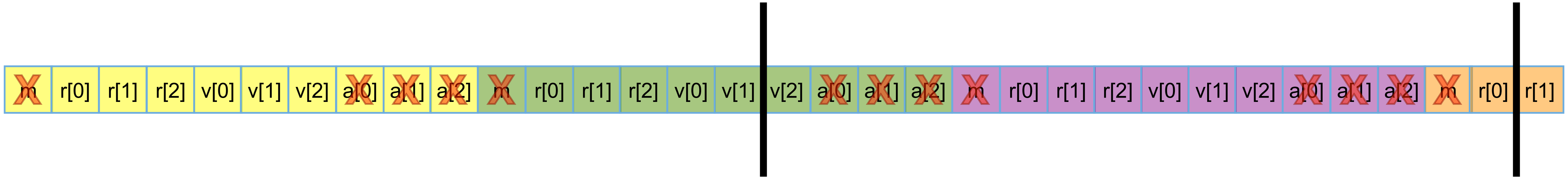
Data organization

```
struct Particle
{
  float m;
  float r[3];
  float v[3];
  float a[3];
} p[N];
```

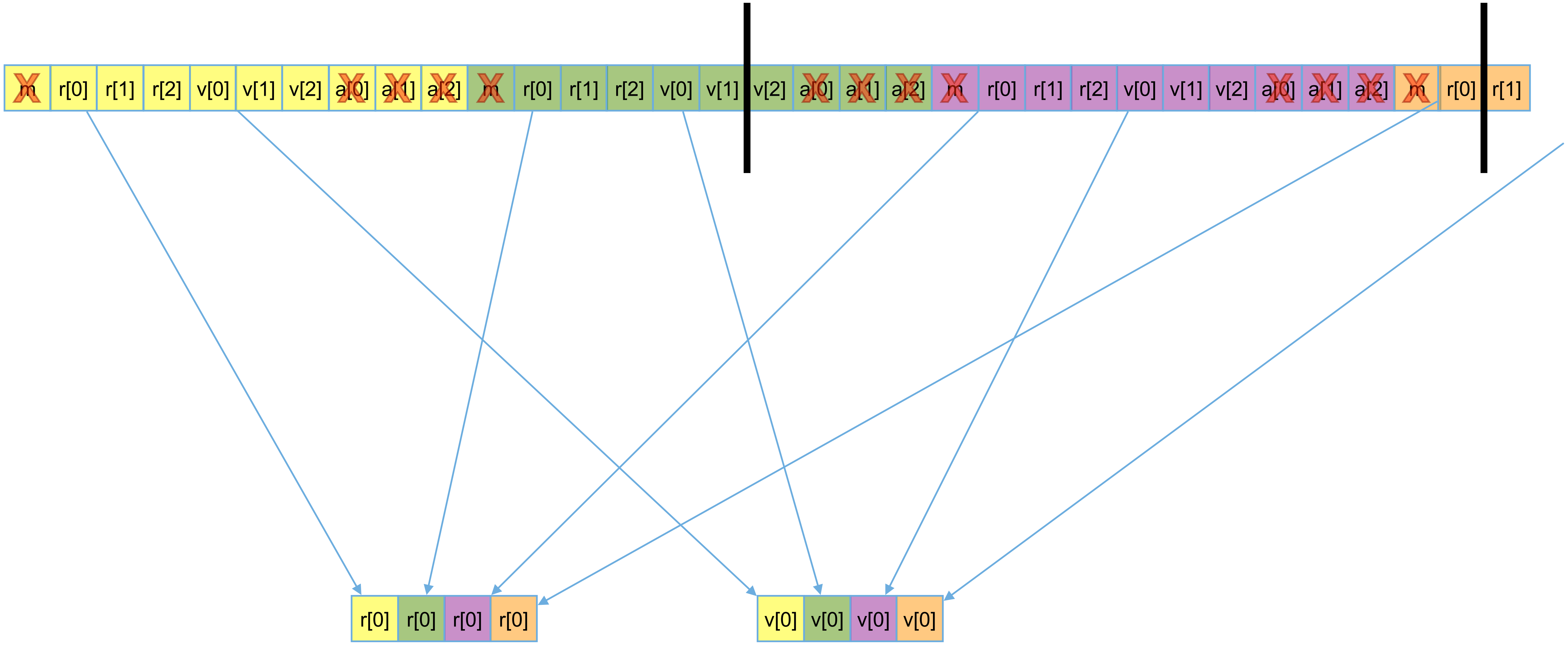


Data organization

```
void update_pos(float dt, int N, struct Particle *p)
{
   for (i=0; i < N; i++)
   {
      p[i].r[0] += p[i].v[0] * dt;
      p[i].r[1] += p[i].v[1] * dt;
      p[i].r[2] += p[i].v[2] * dt;
   }
}
```



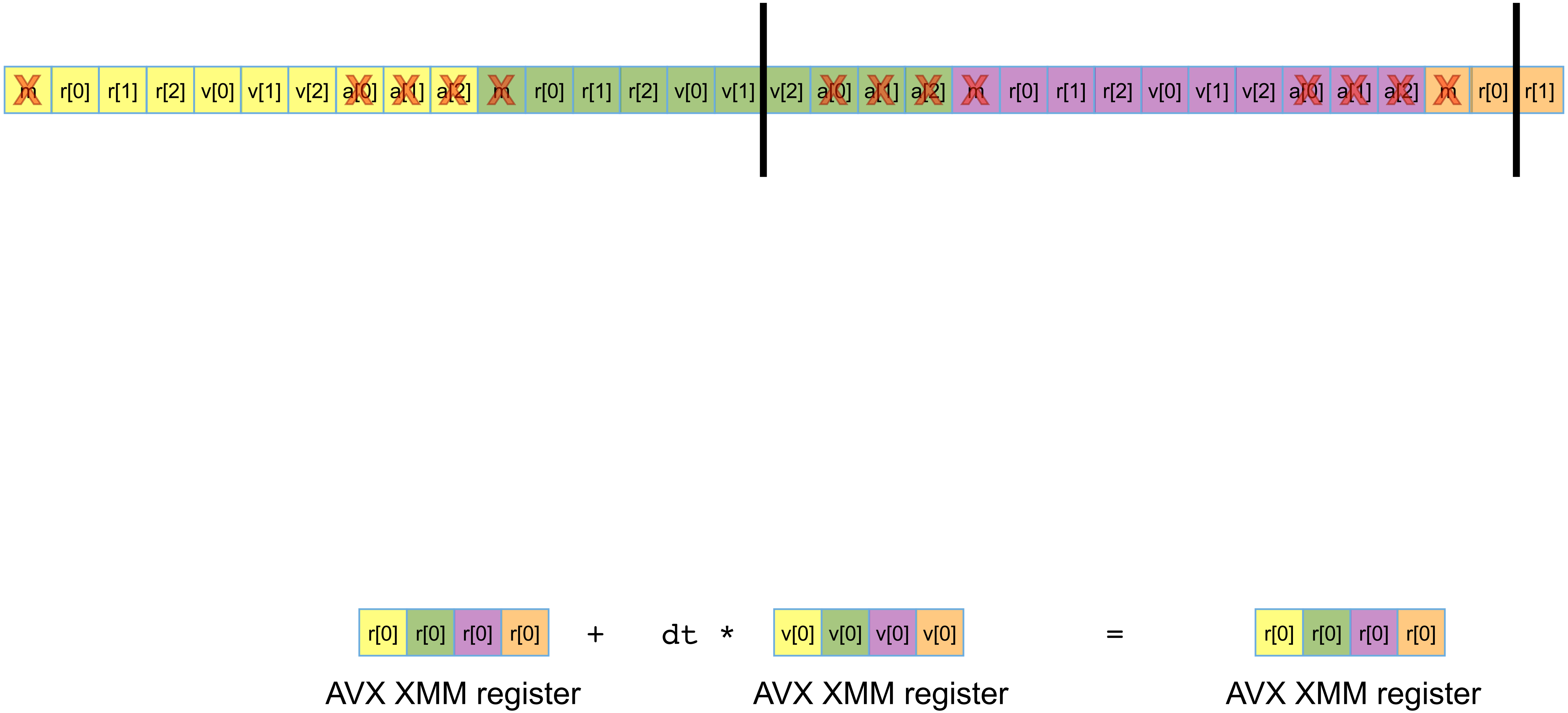
AoS Vectorization - Gather



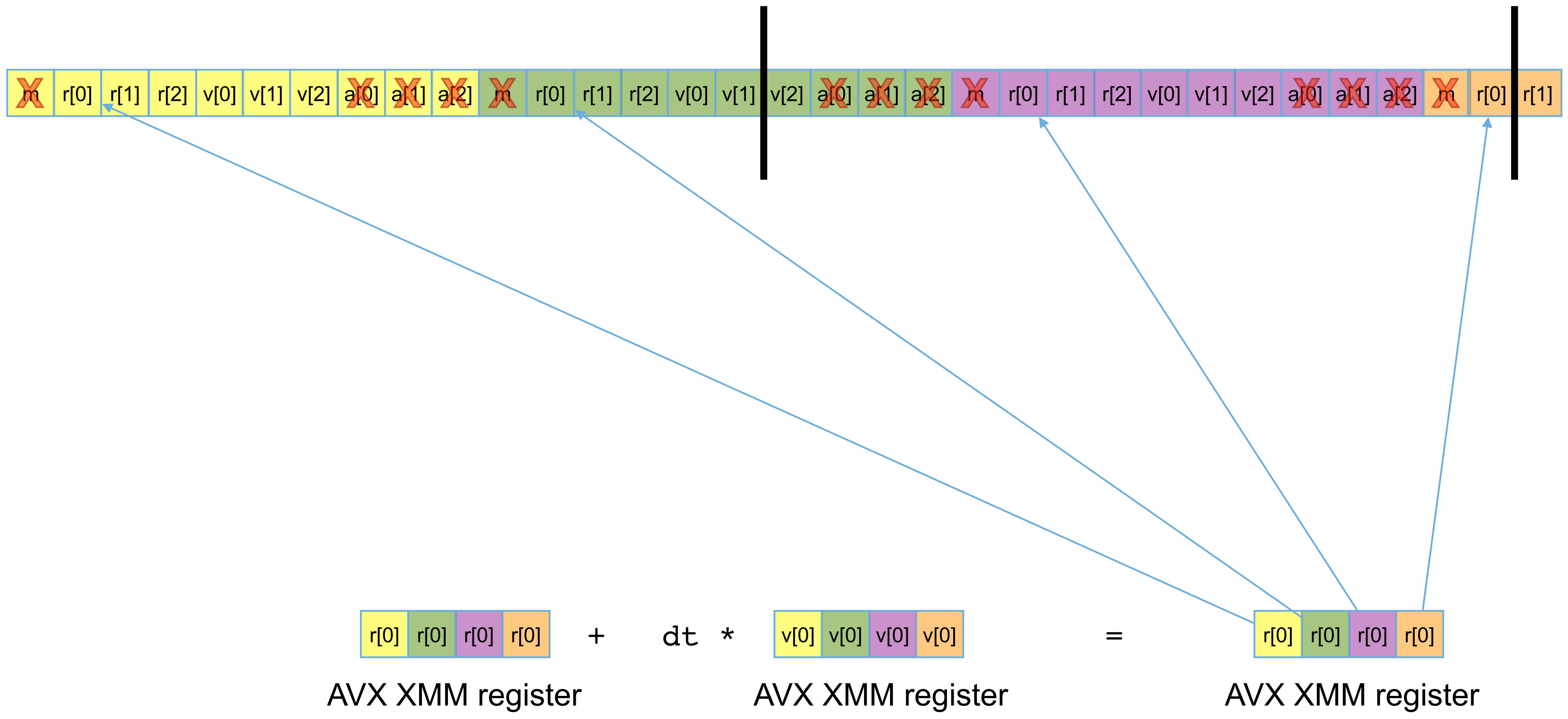
AVX XMM register

AVX XMM register

AoS Vectorization - Compute



AoS Vectorization - Scatter



Data organization

```
struct Particle
{
  float m;
  float r[3];
  float v[3];
  float a[3];
  float timestep;
  int neighbors[M];
  int id;
  ...
} p[N];
```

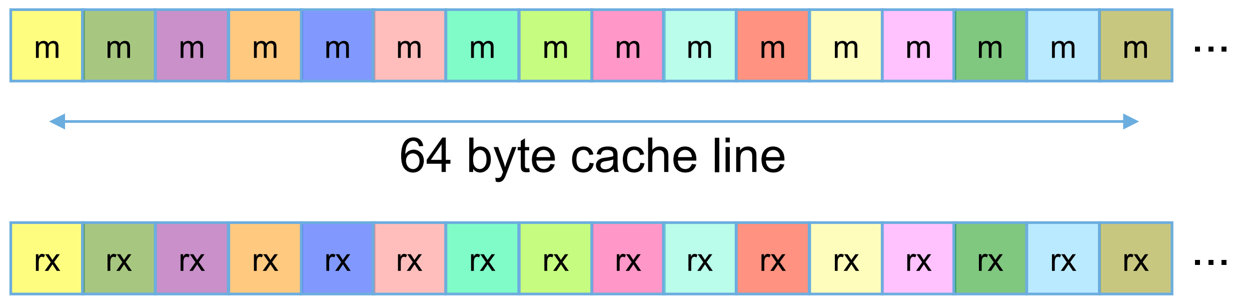
Data organization

- Array of Structures (AOS) is conceptually convenient. Groups all particle properties together.
- Often does not map well to actual usage. In any given function only a small fraction of those properties are used.
- Leads to misalignment of structure size with cache line size.

Can we do better?

Structure of Arrays (SOA)

```
struct Particles
{
  float m[N];
  float rx[N], ry[N], rz[N];
  float vx[N], vy[N], vz[N];
  float ax[N], ay[N], az[N];
} P;
```


m m m m m m m m m m m m m m m m …
64 byte cache line
rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx …

Data organization

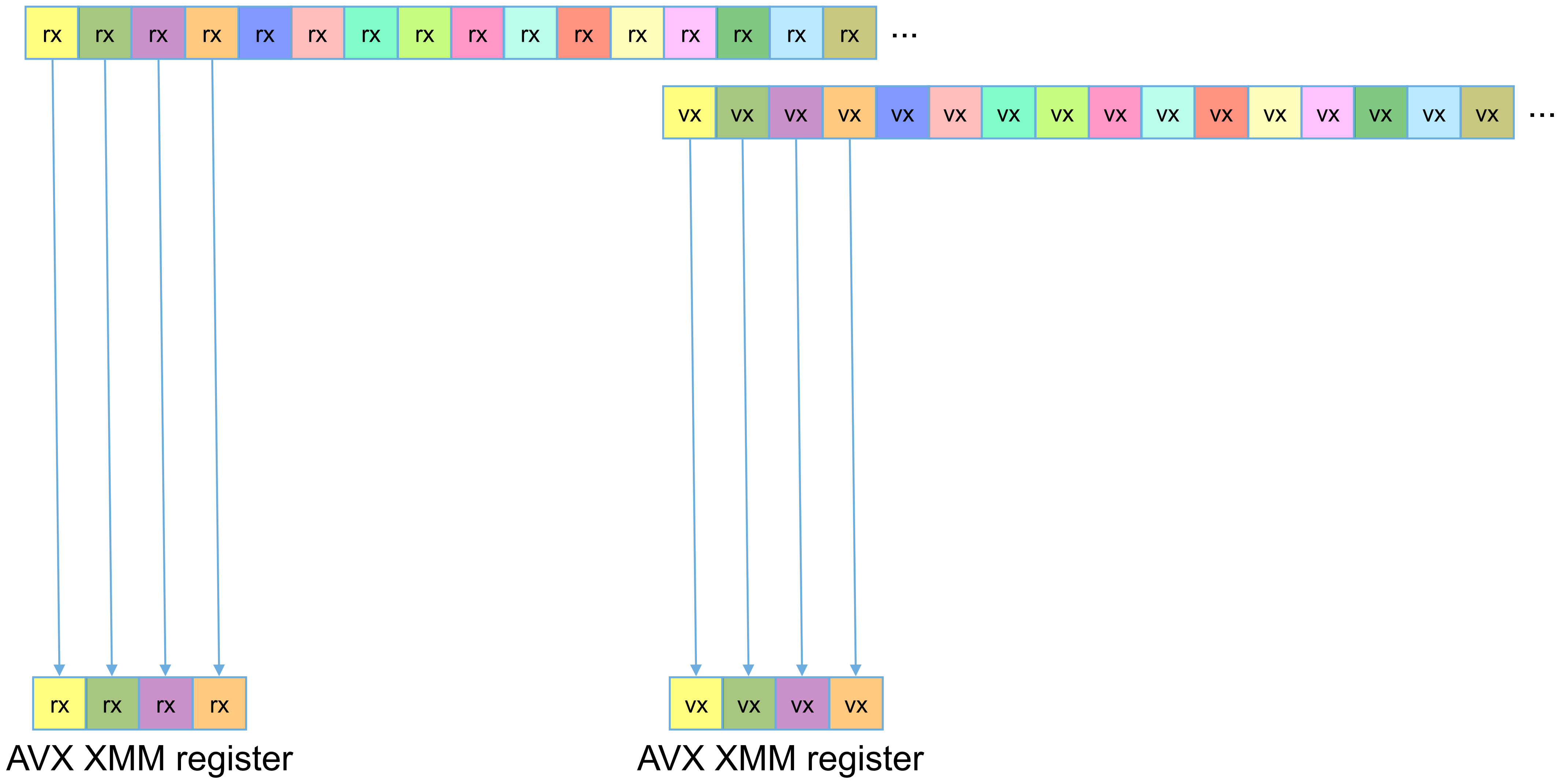
- Structure of Arrays (SOA) packs each property together.
- Only properties that are needed by a function are loaded into cache.
- Alignment is excellent. Access can be more predictable for CPU.

- More awkward to do data rearrangement (e.g., sorting)

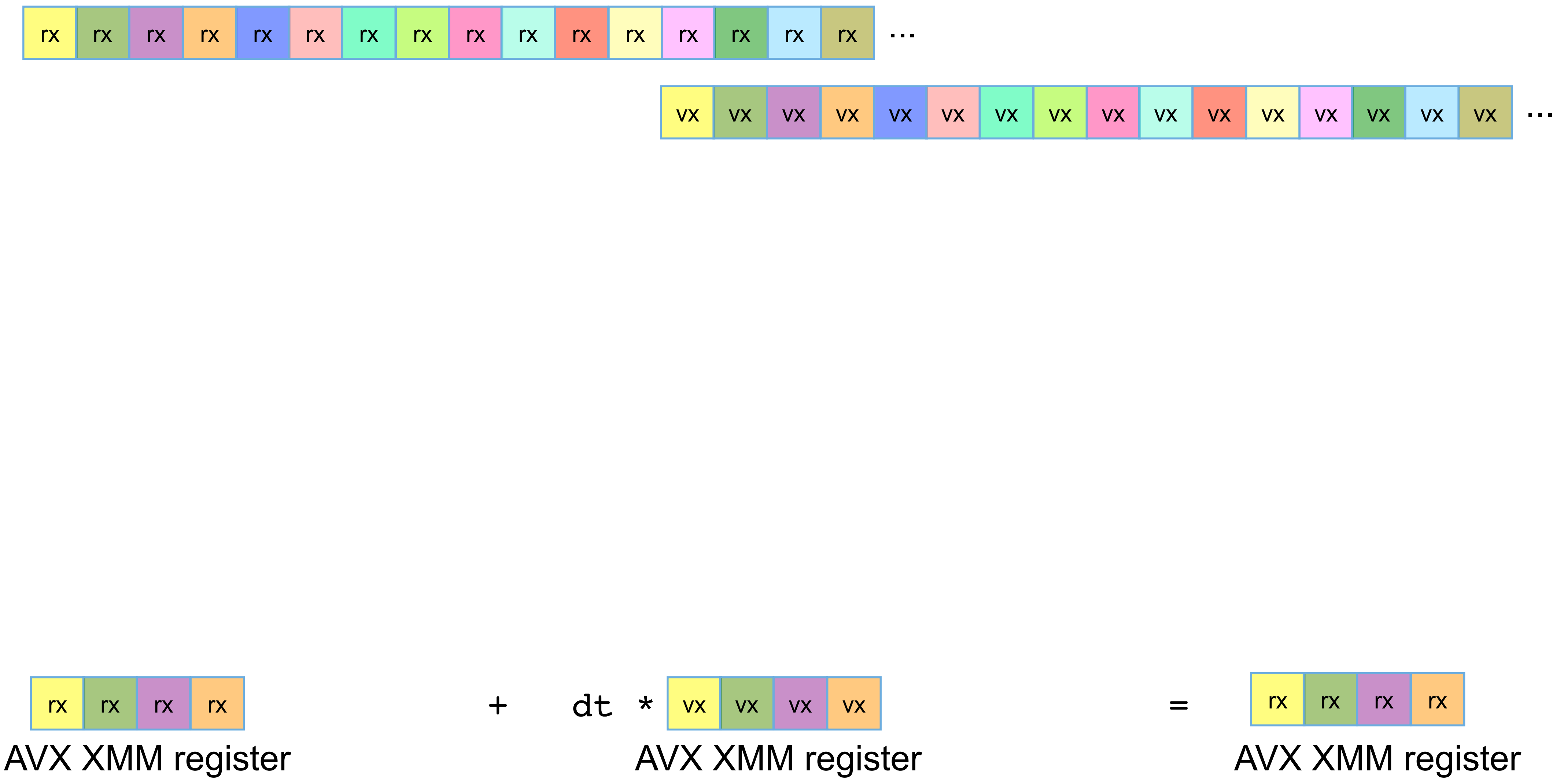
Data organization

```
void update_pos(float dt, int N, struct Particles *P)
{
    for (i=0; i < N; i++)
    {
      P.rx[i] += P.vx[i] * dt;
      P.ry[i] += P.vy[i] * dt;
      P.rz[i] += P.vz[i] * dt;
    }
}
```

SoA Vectorization - Gather

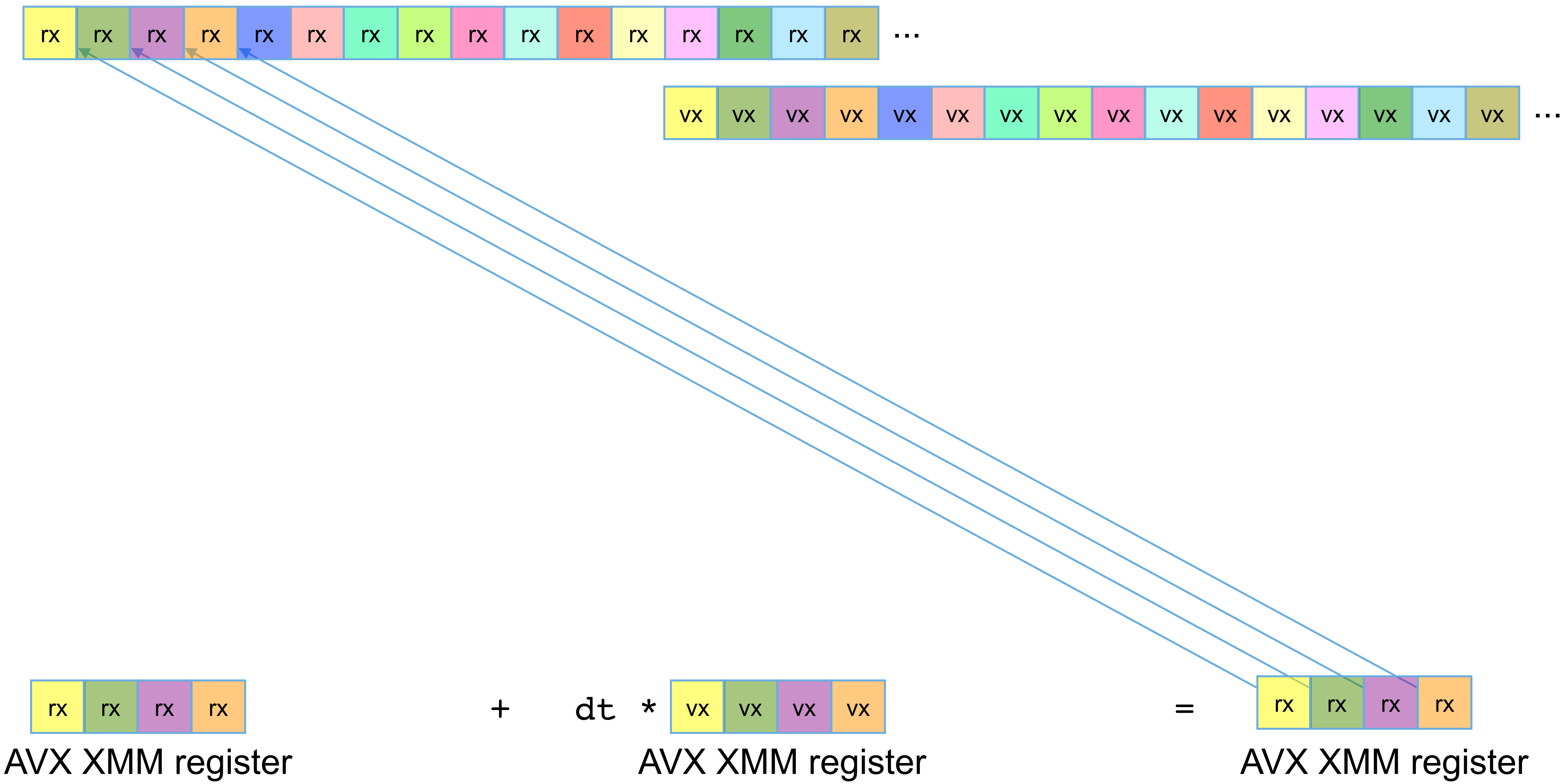

rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx ...
vx vx vx vx vx vx vx vx vx vx vx vx vx vx vx vx ...
rx rx rx rx
AVX XMM register
vx vx vx vx
AVX XMM register

SoA Vectorization - Compute



rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx …
vx vx vx vx vx vx vx vx vx vx vx vx vx vx vx vx …
rx rx rx rx
AVX XMM register
+ dt *
vx vx vx vx
AVX XMM register
=
rx rx rx rx
AVX XMM register

SoA Vectorization - Scatter


rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx rx …
vx vx vx vx vx vx vx vx vx vx vx vx vx vx vx vx …
rx rx rx rx
AVX XMM register
+ dt *
vx vx vx vx
AVX XMM register
=
rx rx rx rx
AVX XMM register

Array aliasing in C

- We have implicitly assumed that each array in SoA is distinct.
- Imagine what could happen if vx is actually a pointer to somewhere in rx?
- This is a very real possibility in C, so the compiler can not assume they are distinct.
- These loops can't be automatically vectorized.

```
for (i=0; i < N; i++)
  P.rx[i] += P.vx[i] * dt;
```

$\longrightarrow$

```
for (i=0; i < N; i++)
  P.rx[i] += P.rx[i] * dt;
```

```
P.rx[0] += P.vx[0] * dt;
P.rx[1] += P.vx[1] * dt;
P.rx[2] += P.vx[2] * dt;
P.rx[3] += P.vx[3] * dt;
```

```
P.rx[0] += P.rx[1] * dt;
P.rx[1] += P.rx[2] * dt;
P.rx[2] += P.rx[3] * dt;
P.rx[3] += P.rx[4] * dt;
```

Array aliasing in C

- We have implicitly assumed that each array in SoA is distinct.
- Imagine what could happen if vx is actually a pointer to somewhere in rx?
- This is a very real possibility in C, so the compiler can not assume they are distinct.
- These loops can't be automatically vectorized.

```
LOOP BEGIN at nb-soa-kda.c(22,5)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between vx[i] (24:9) and vz[i] (26:9)
   remark #15346: vector dependence: assumed OUTPUT dependence between vz[i] (26:9) and vx[i] (24:9)
LOOP END
```

Array aliasing in C

- `restrict` keyword tells compiler to trust you that no other array aliasing the given one :-D
- `#pragma omp simd` will also make the same assumption.

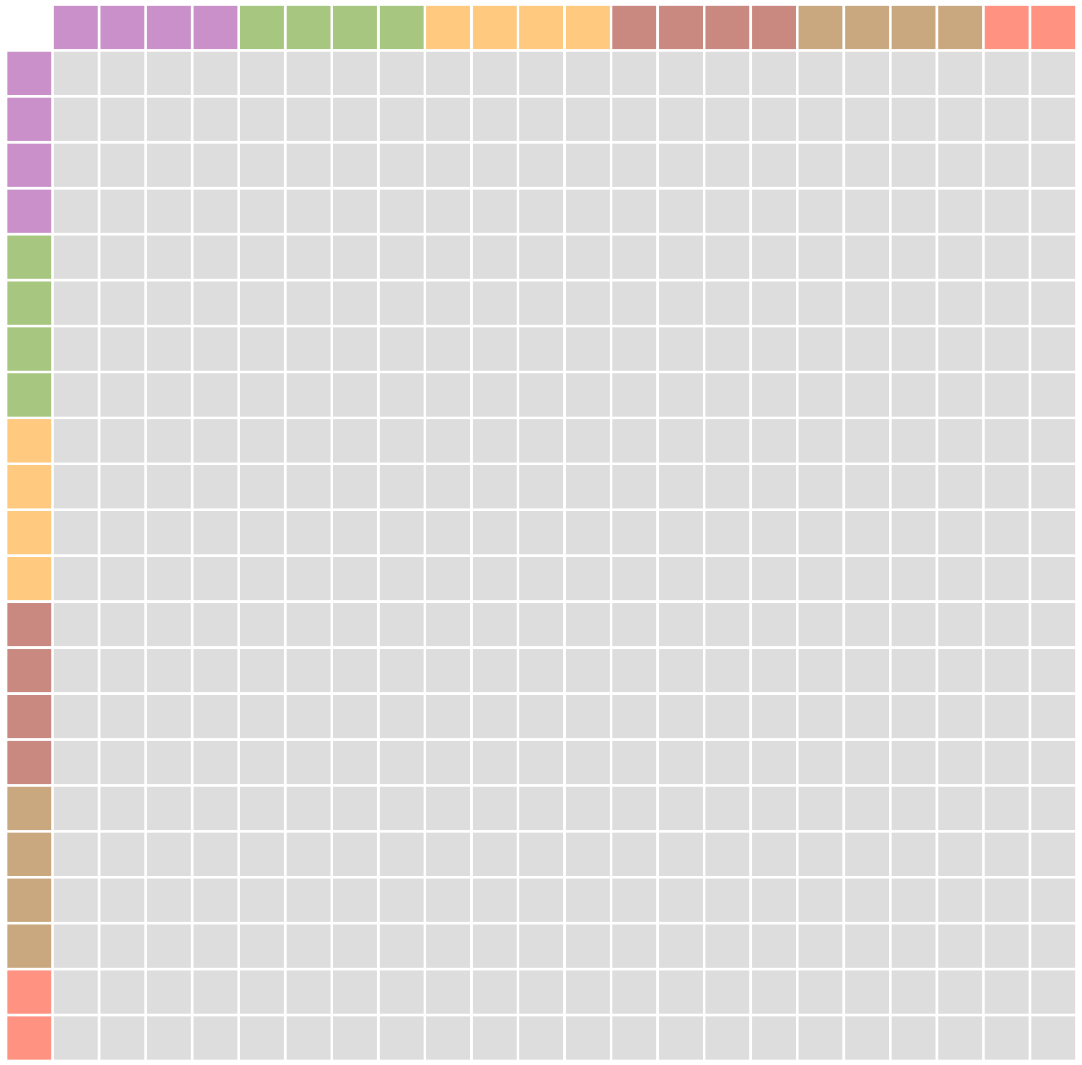
```
real * restrict vx = v->x;
real * restrict vy = v->y;
real * restrict vz = v->z;
real * restrict ax = a->x;
real * restrict ay = a->y;
real * restrict az = a->z;

#pragma omp simd
for (int i=0; i < N; i++)
{
    vx[i] += ax[i] * dt;
    vy[i] += ay[i] * dt;
    vz[i] += az[i] * dt;
}
```

Data organization

```
void update_accel(float rs, int N, struct Particles *P)
{
    for (i=0; i < N; i++)
    {
        P.ax[i] = P.ay[i] = P.az[i] = 0.0;
        for (j=0; j < N; j++)
        {
            float dx = P.rx[i] - P.rx[j];
            float dy = P.ry[i] - P.ry[j];
            float dz = P.rz[i] - P.rz[j];
            float ir = 1.0 / sqrt(dx*dx + dy*dy + dz*dz + rs*rs);
            P.ax[j] += P.m[i] * dx * ir * ir * ir;
            P.ay[j] += P.m[i] * dy * ir * ir * ir;
            P.az[j] += P.m[i] * dz * ir * ir * ir;
          }
     }
}
```

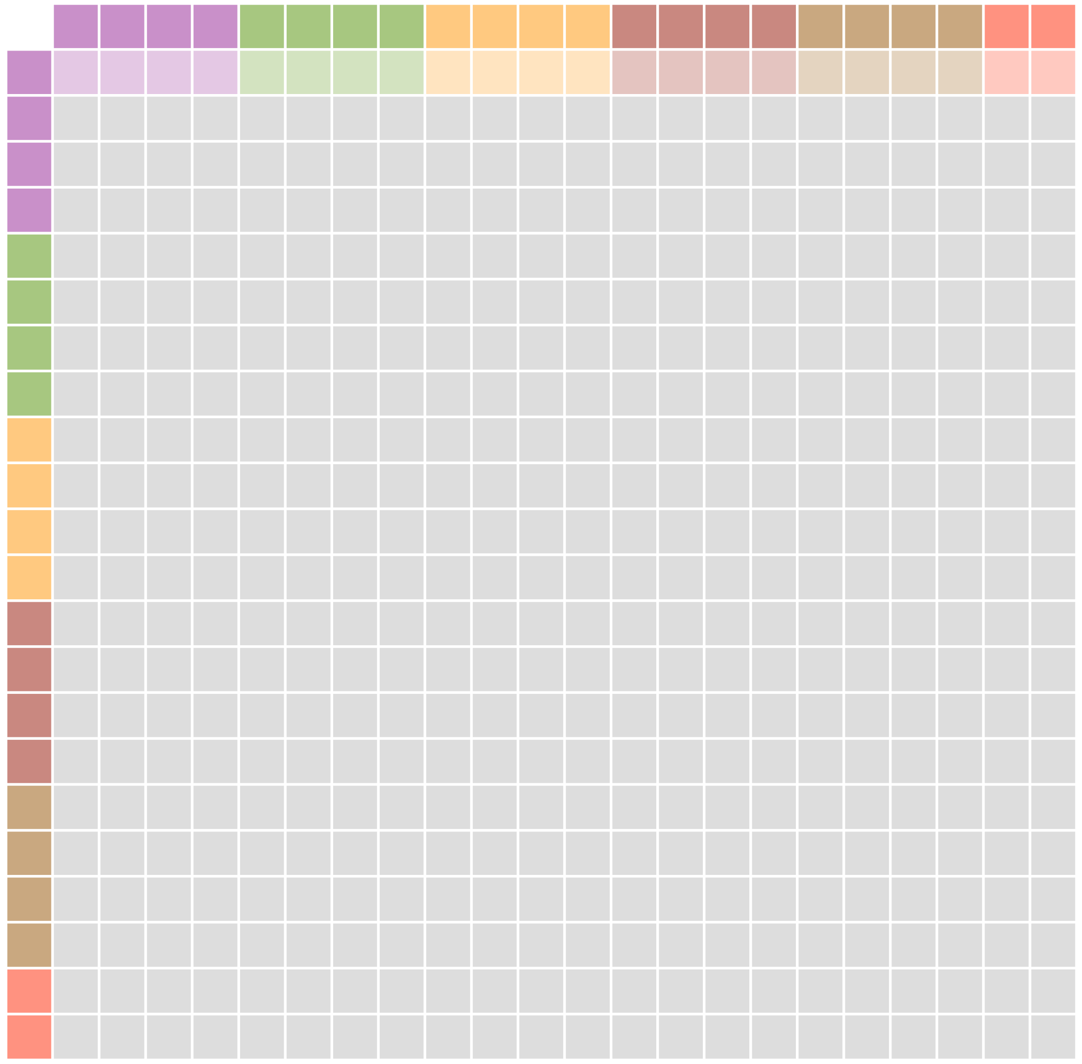
Data organization

For this example, cacheline is four elements



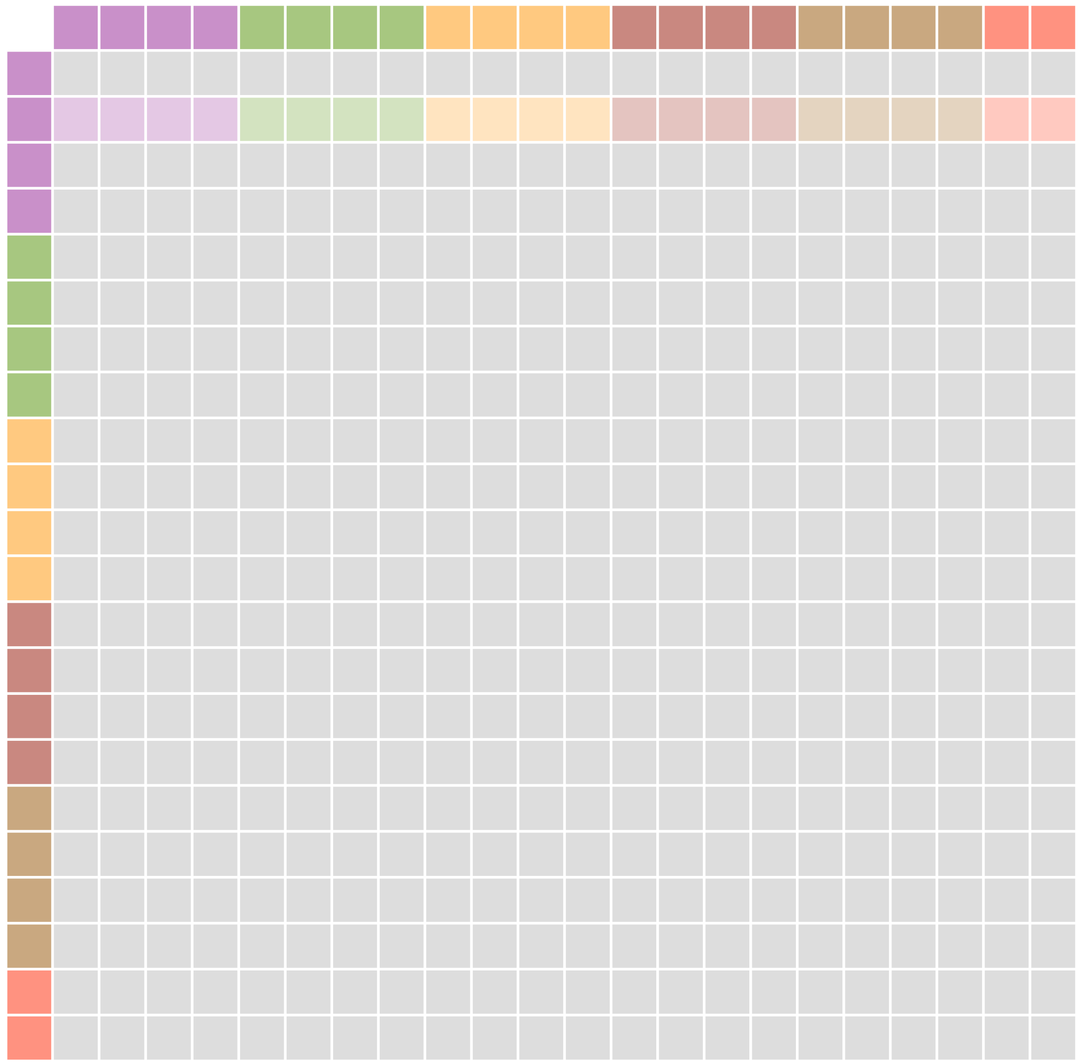
Data organization

For this example, cacheline is four elements



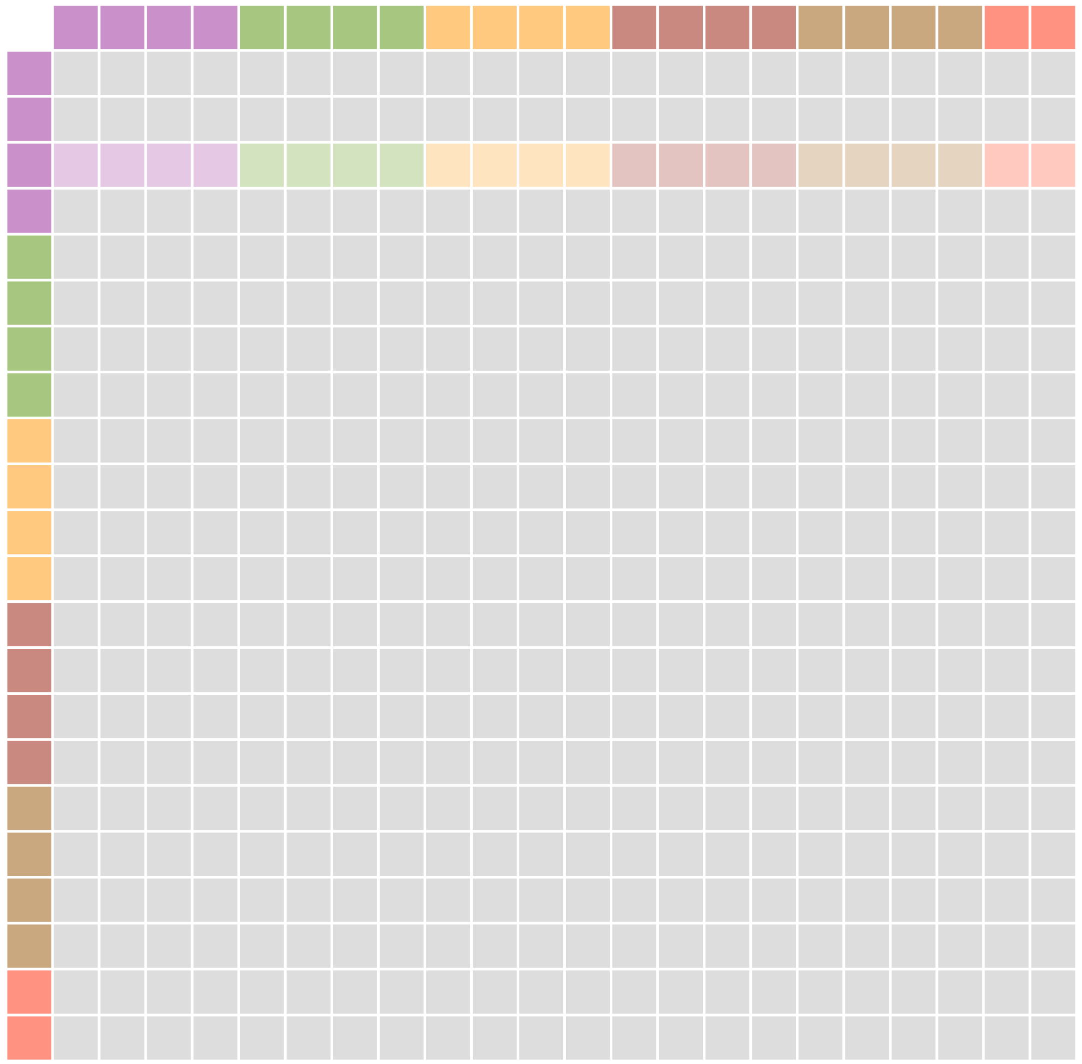
Data organization

For this example, cacheline is four elements



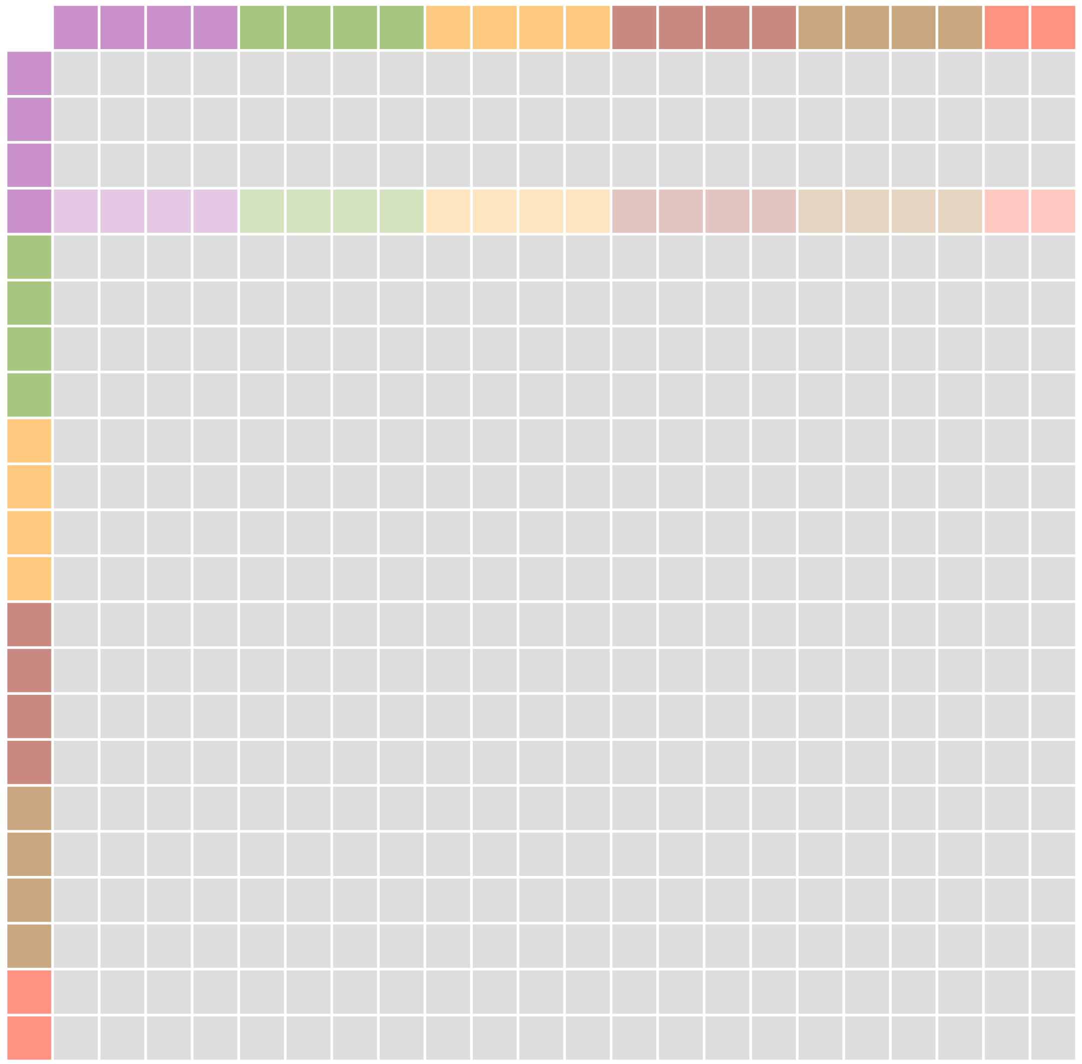
Data organization

For this example, cacheline is four elements



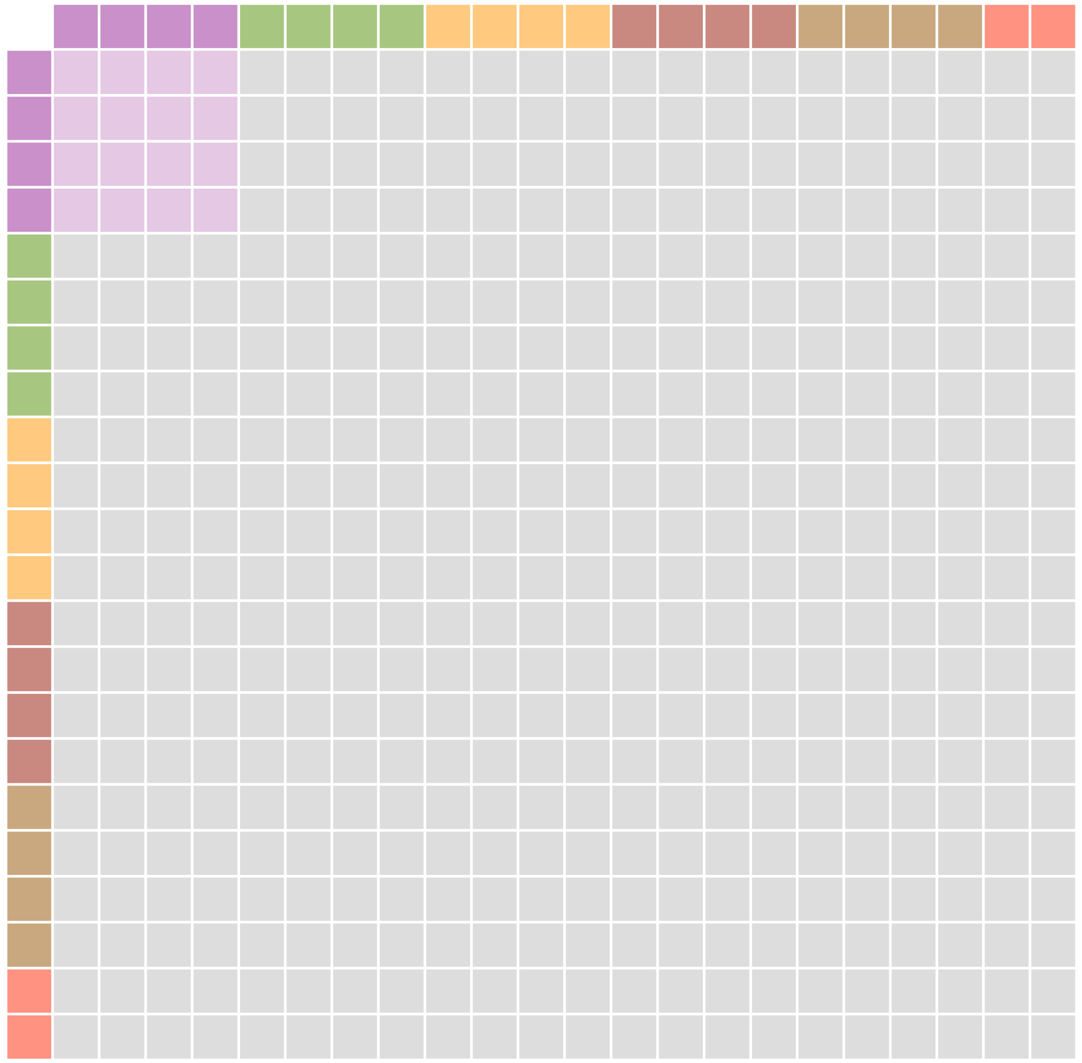
Data organization

For this example, cacheline is four elements



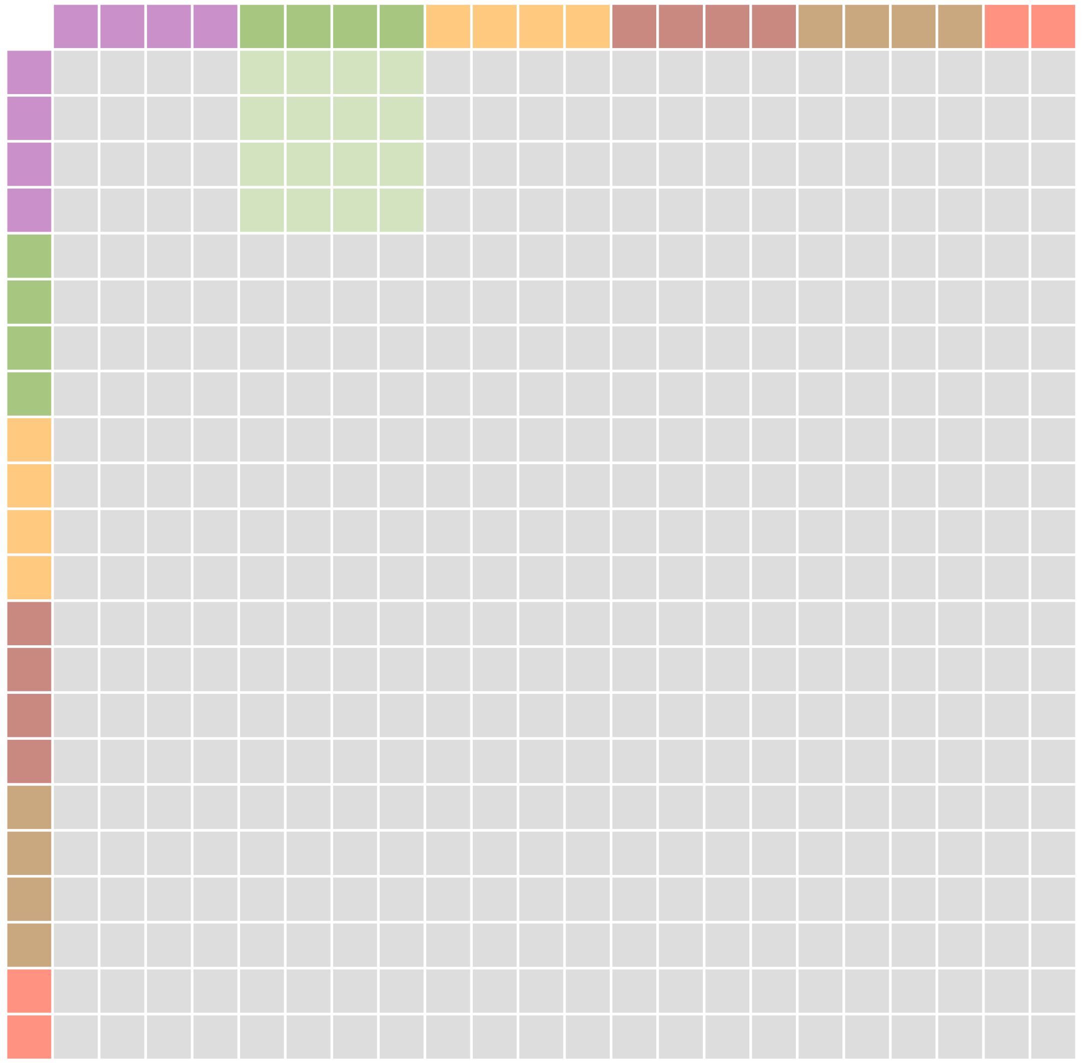
Data organization

For this example, cacheline is four elements



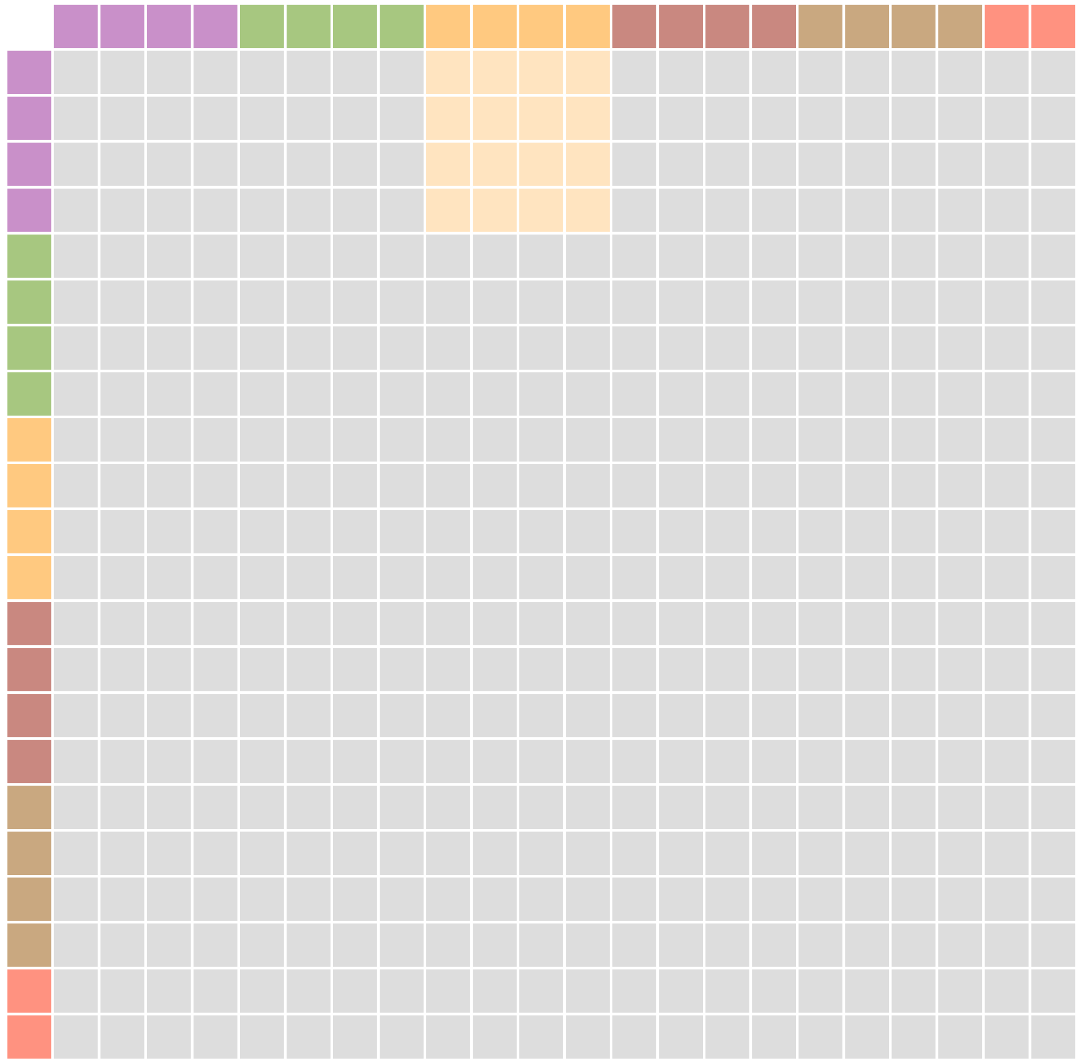
Data organization

For this example, cacheline is four elements



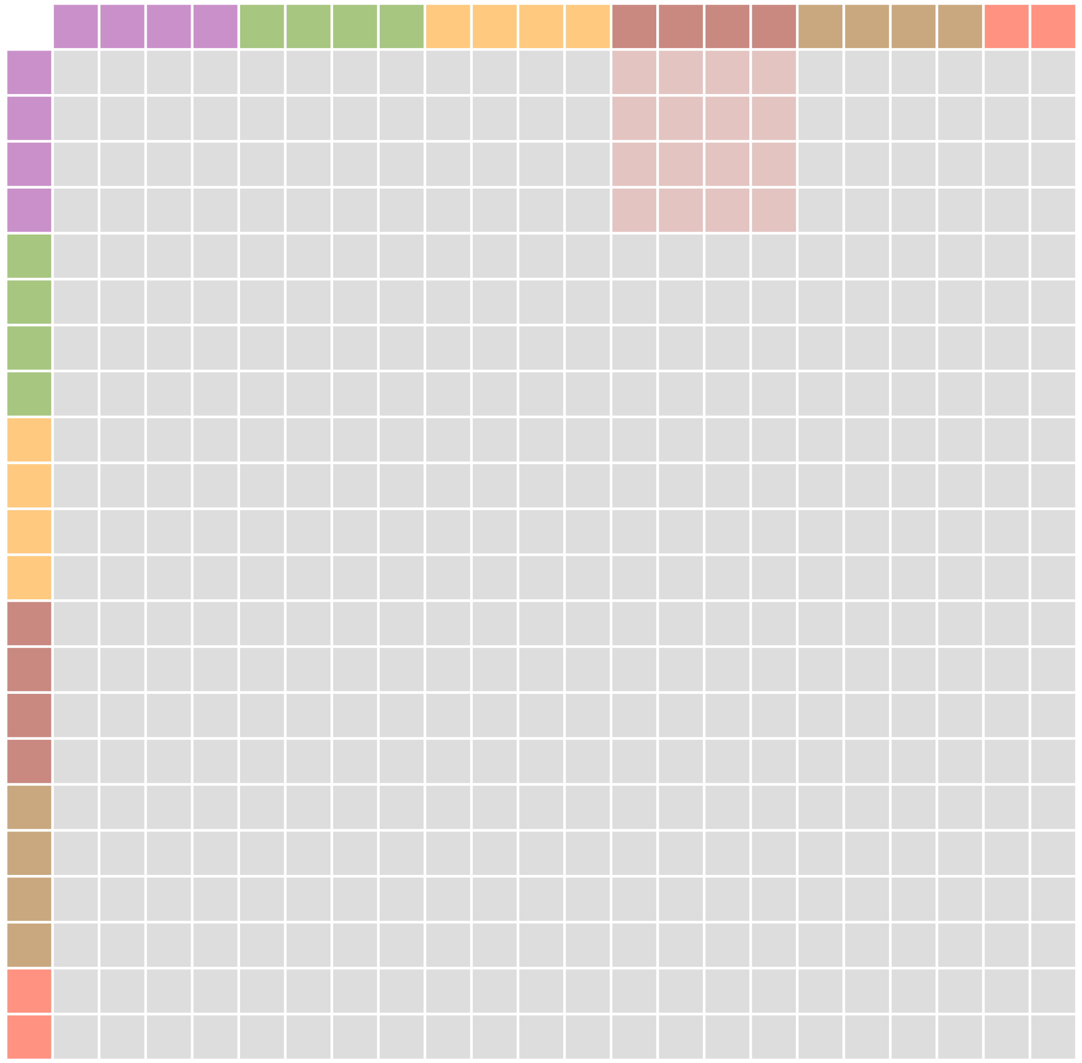
Data organization

For this example, cacheline is four elements



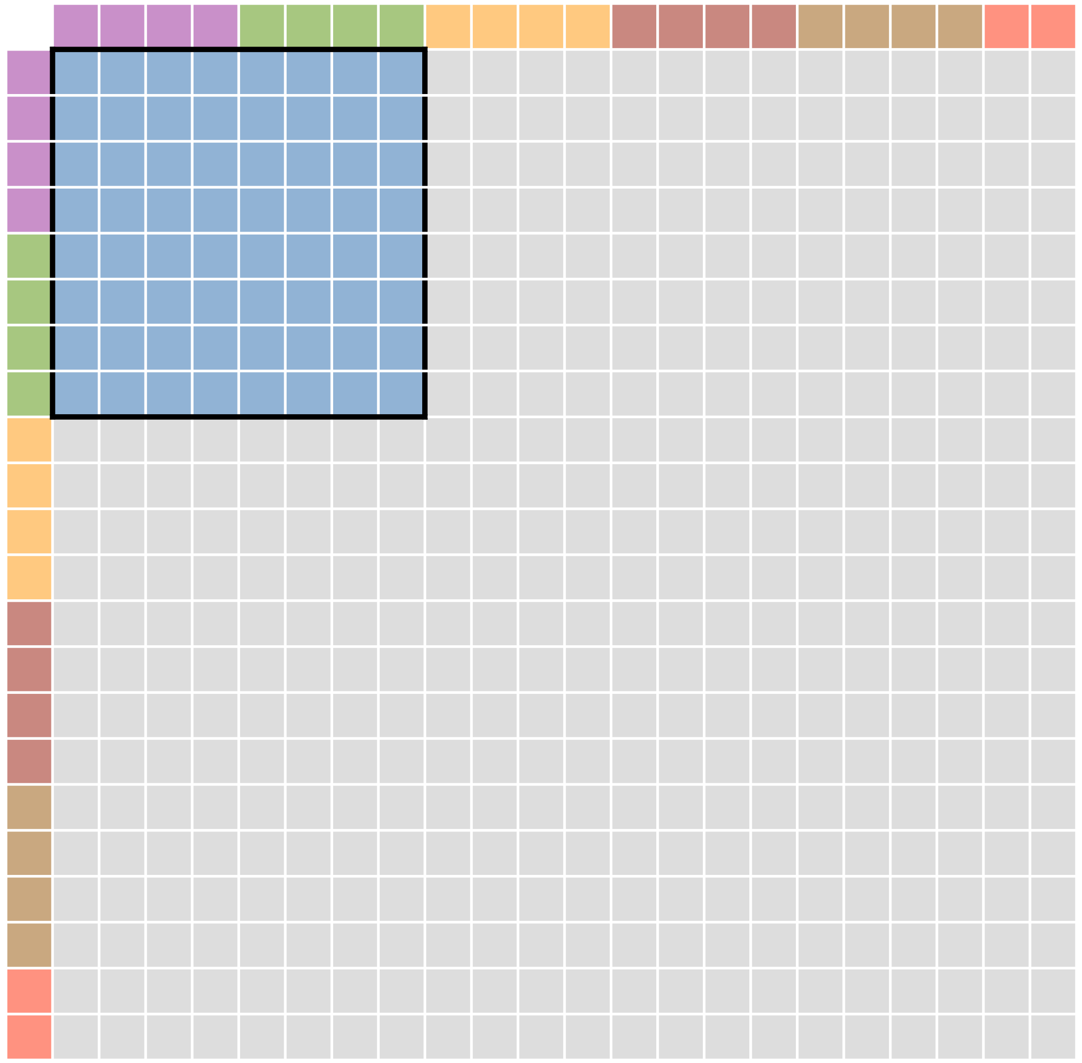
Data organization

For this example, cacheline is four elements



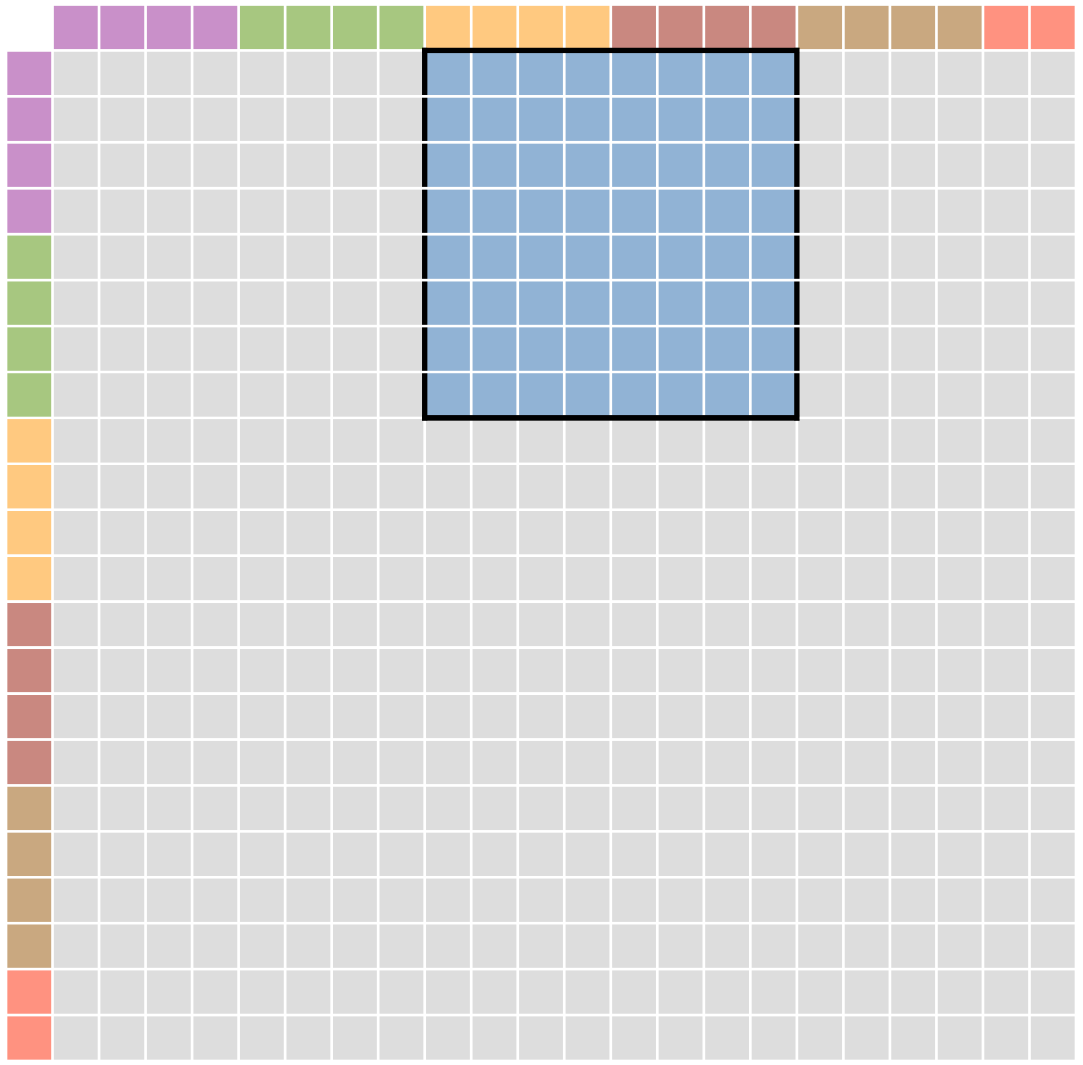
Data organization

For this example, cacheline is four elements



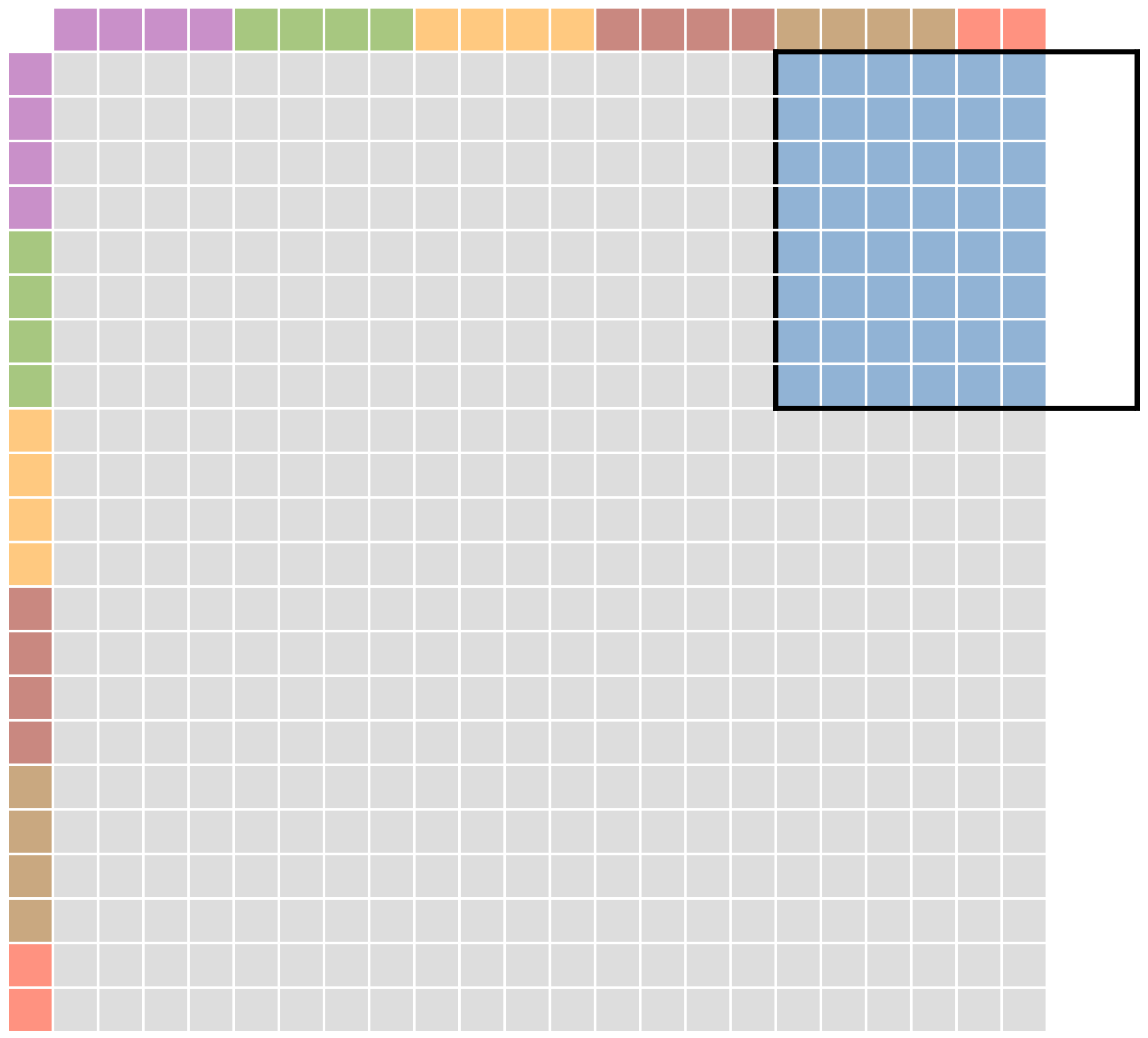
Data organization

For this example, cacheline is four elements



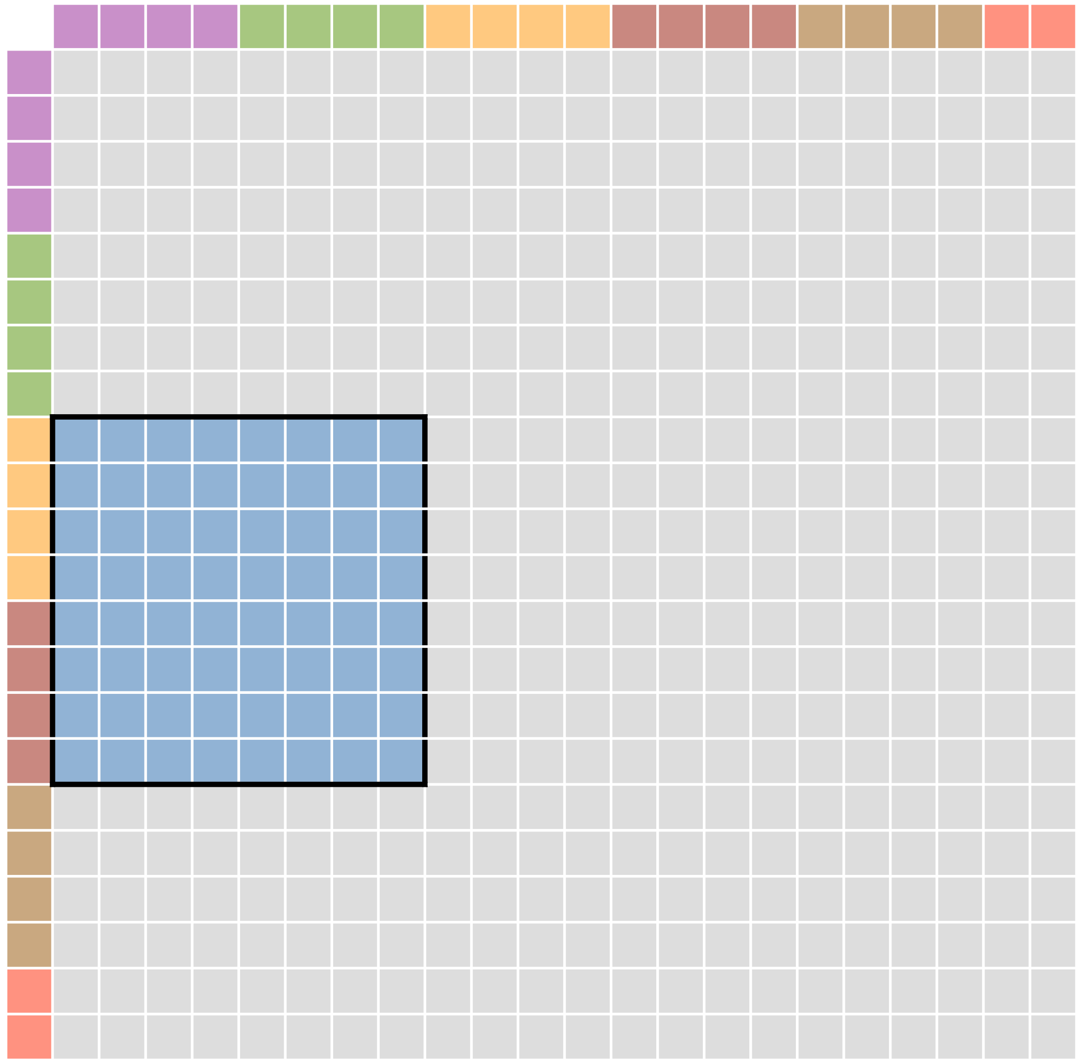
Data organization

For this example, cacheline is four elements



Data organization

For this example, cacheline is four elements

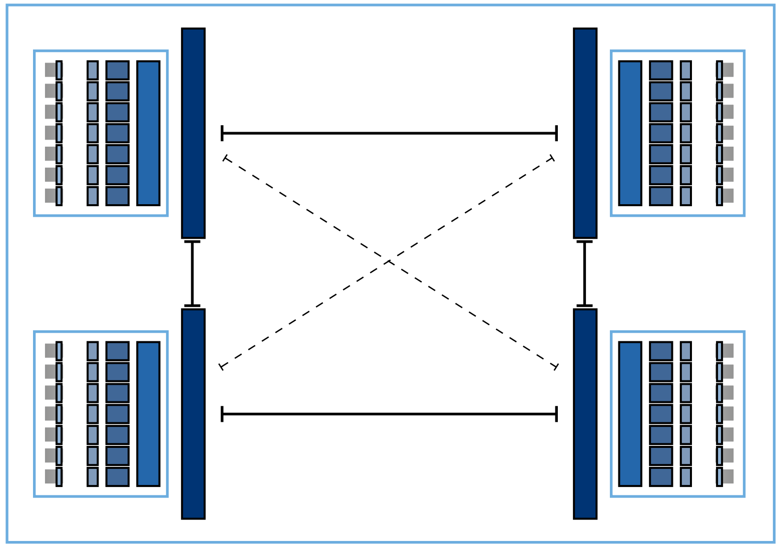


Data organization

```
void update_accel(float rs, int N, struct Particles *P)
{
    for (int ti=0; ti < N; ti += TILE_SIZE)
    {
        for (int ki=0, i=ti; ki < TILE_SIZE; ki++, i++)
            P.ax[i] = P.ay[i] = P.az[i] = 0.0;

        for (int tj=0; tj < N; tj += TILE_SIZE)
        {
            for (int ki=0, i=ti; ki < TILE_SIZE; ki++, i++)
            {
                for (int kj=0, j=tj; kj < TILE_SIZE; kj++, j++)
                {
                    float dx = P.rx[i] - P.rx[j];
                    float dy = P.ry[i] - P.ry[j];
                    float dz = P.rz[i] - P.rz[j];
                    float ir = 1.0 / sqrt(dx*dx + dy*dy + dz*dz + rs*rs);
                    P.ax[j] += P.m[i] * dx * ir * ir * ir;
                    P.ay[j] += P.m[i] * dy * ir * ir * ir;
                    P.az[j] += P.m[i] * dz * ir * ir * ir;
                }
            }
        }
    }
}
```

Non-Uniform Memory Access (NUMA)

- Each socket has its local main memory.
- Each socket can access main memory of other sockets.
- Local memory access is faster.
- Best performance when cores access memory local to their sockets.
- Memory only allocated by OS when written to, **not** when allocated via `malloc` (e.g.).
- Known as "first-touch" policy. Memory is allocated on memory local to the first core to write ("touch") to it.



Non-Uniform Memory Access (NUMA)

- Memory is organized by the hardware into pages. Typically a page size is 4096 bytes.
- Memory is not allocated all at once by first-touch per page.
- If cores on different sockets touch different pages that are part of the same array, the array will be physically allocated across different sockets' main memory.
- Logical access will be unaffected but some cores may require longer to access some elements than others.
- Best is when cores touch the pages they will need and do not access any other pages.

Non-Uniform Memory Access (NUMA)

```
void first_touch(void *s, int c, size_t nmemb, size_t size)
{
    long PAGE_SIZE = sysconf(_SC_PAGESIZE);
    char *ptr = s;
    #pragma omp parallel for schedule(static)
    for (long i=0; i < size; i += PAGE_SIZE)
        ptr[i] = 0;
}
```

Non-Uniform Memory Access (NUMA)

```
$ likwid-topology
[...]
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:          2
--------------------------------------------------------------------------------
Domain:                0
Processors:            ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 )
Distances:             10 21
Free memory:           253783 MB
Total memory:          386682 MB
--------------------------------------------------------------------------------
Domain:                1
Processors:            ( 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 )
Distances:             21 10
Free memory:           266923 MB
Total memory:          387032 MB
--------------------------------------------------------------------------------
```

Hardware vs. Software Model

- Writing optimal software is made more difficult because what is good for the hardware does not always align with how we would like to think about a problem.
- Programming languages are more often designed around making our mental model easy to express.
- Even simple "object oriented" design can map poorly onto the hardware.

lrz

Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities