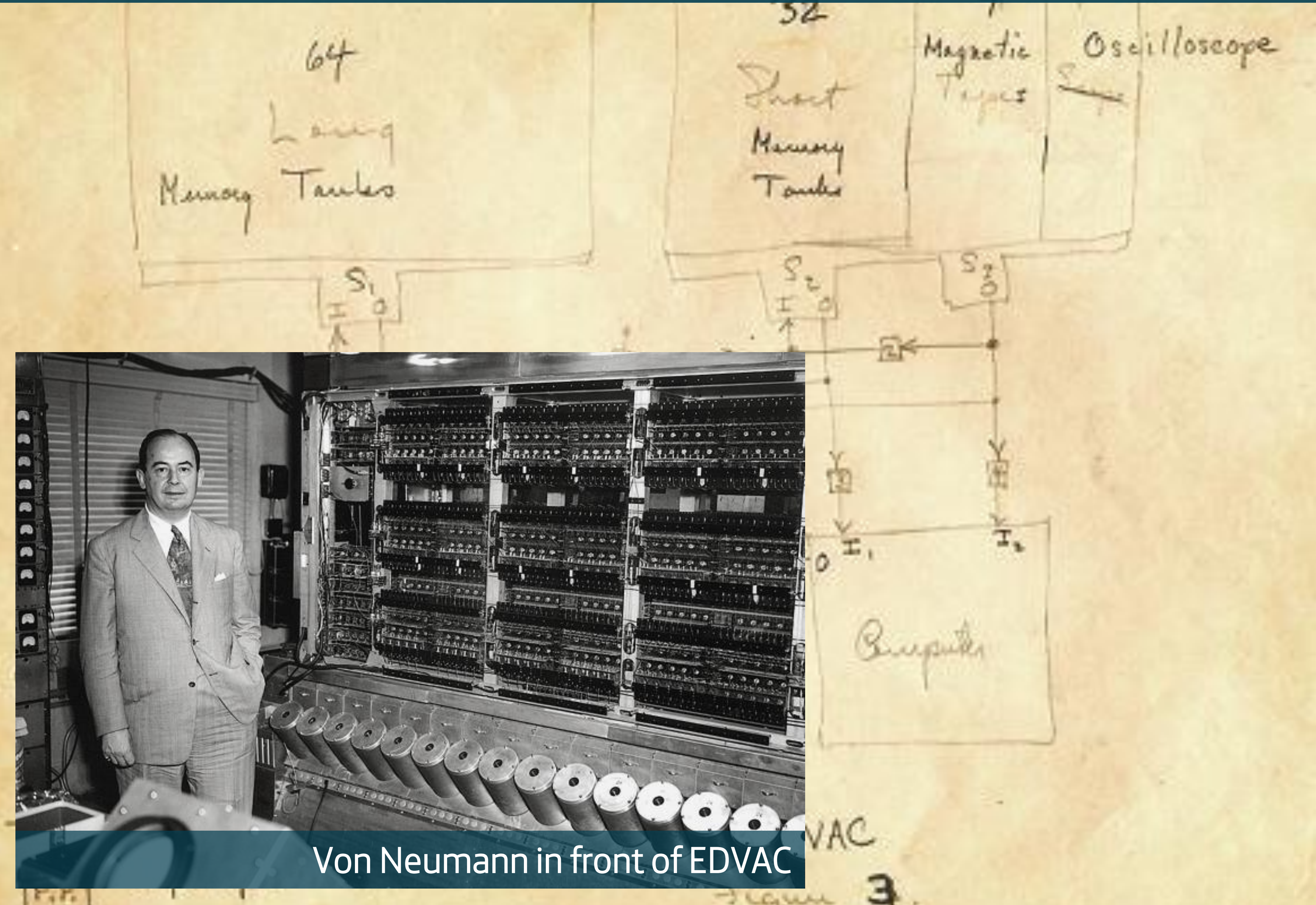
Mind the cache

using std::cpp 2015

Joaquín M López Muñoz <joaquin@tid.es>
Madrid, November 2015

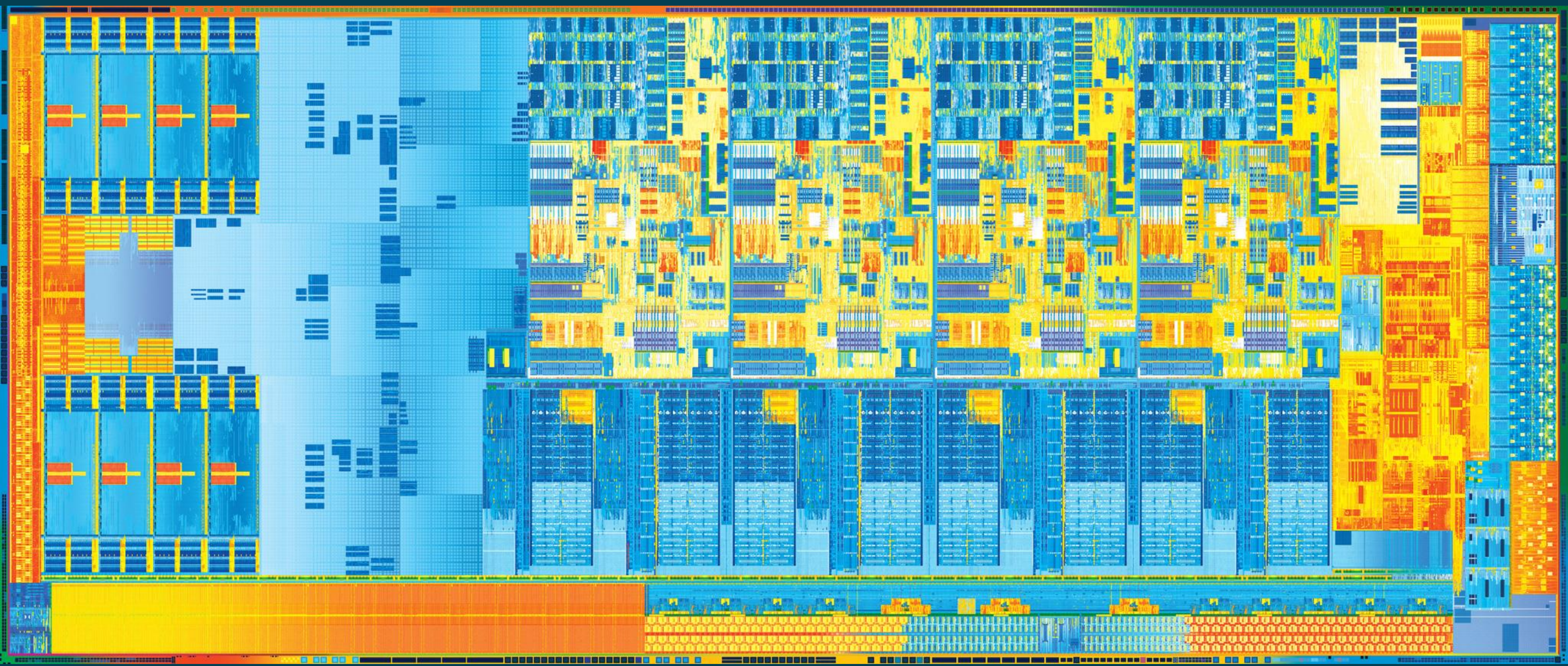
Telefónica

Our mental model of the CPU is 70 years old



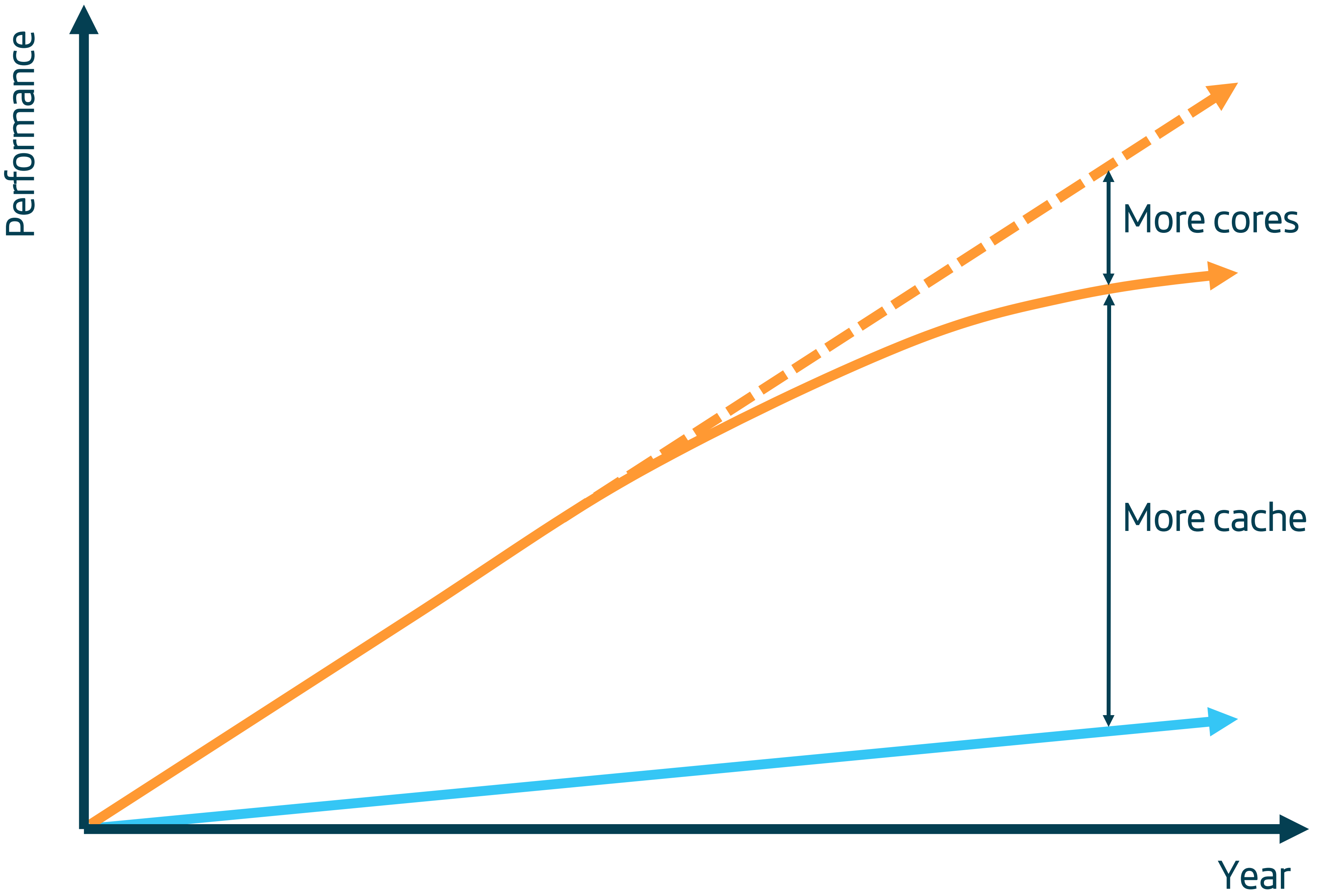
Von Neumann in front of EDVAC

Current reality looks quite different

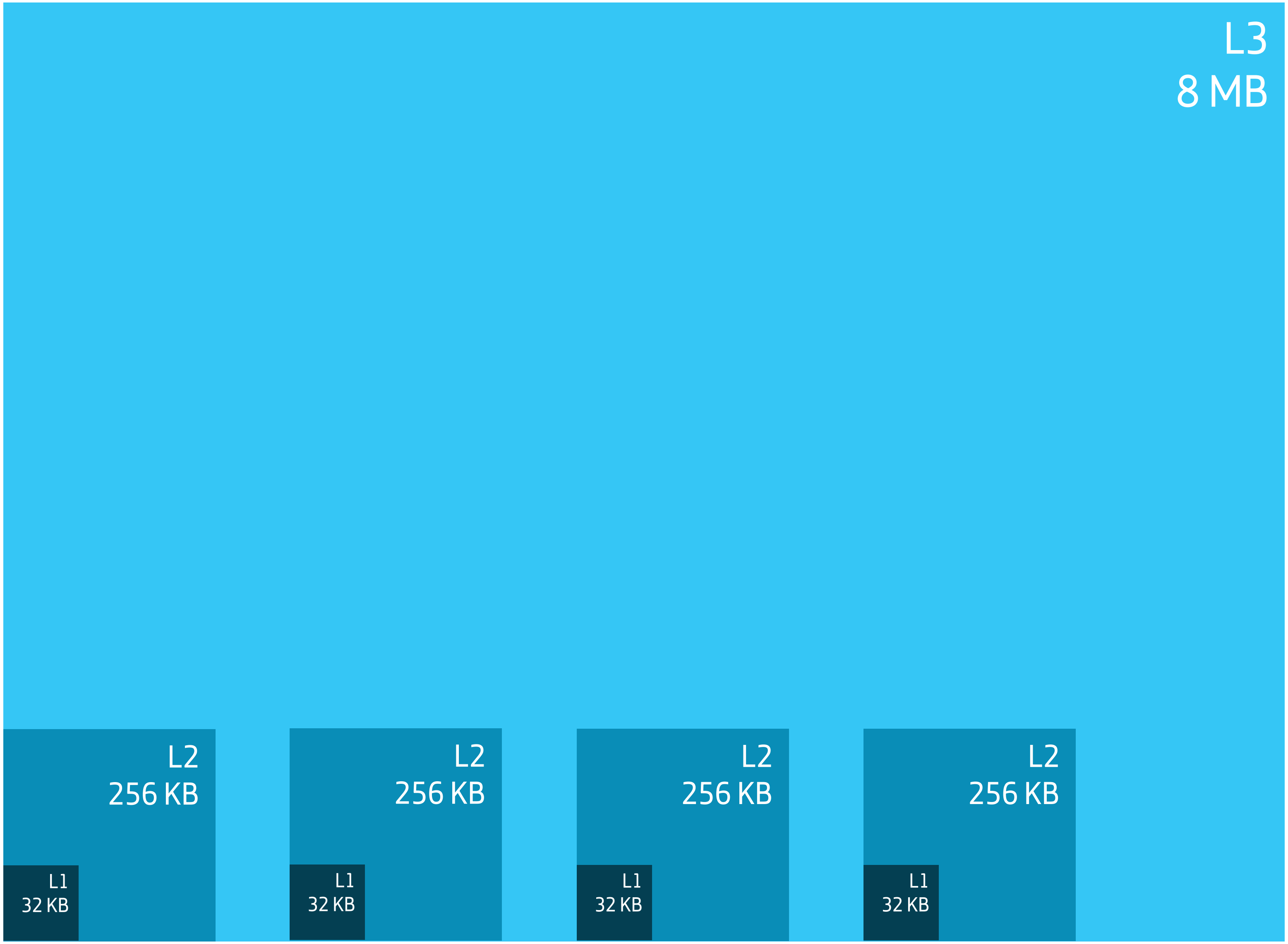


Intel Core i7-3770K (Ivy Bridge)

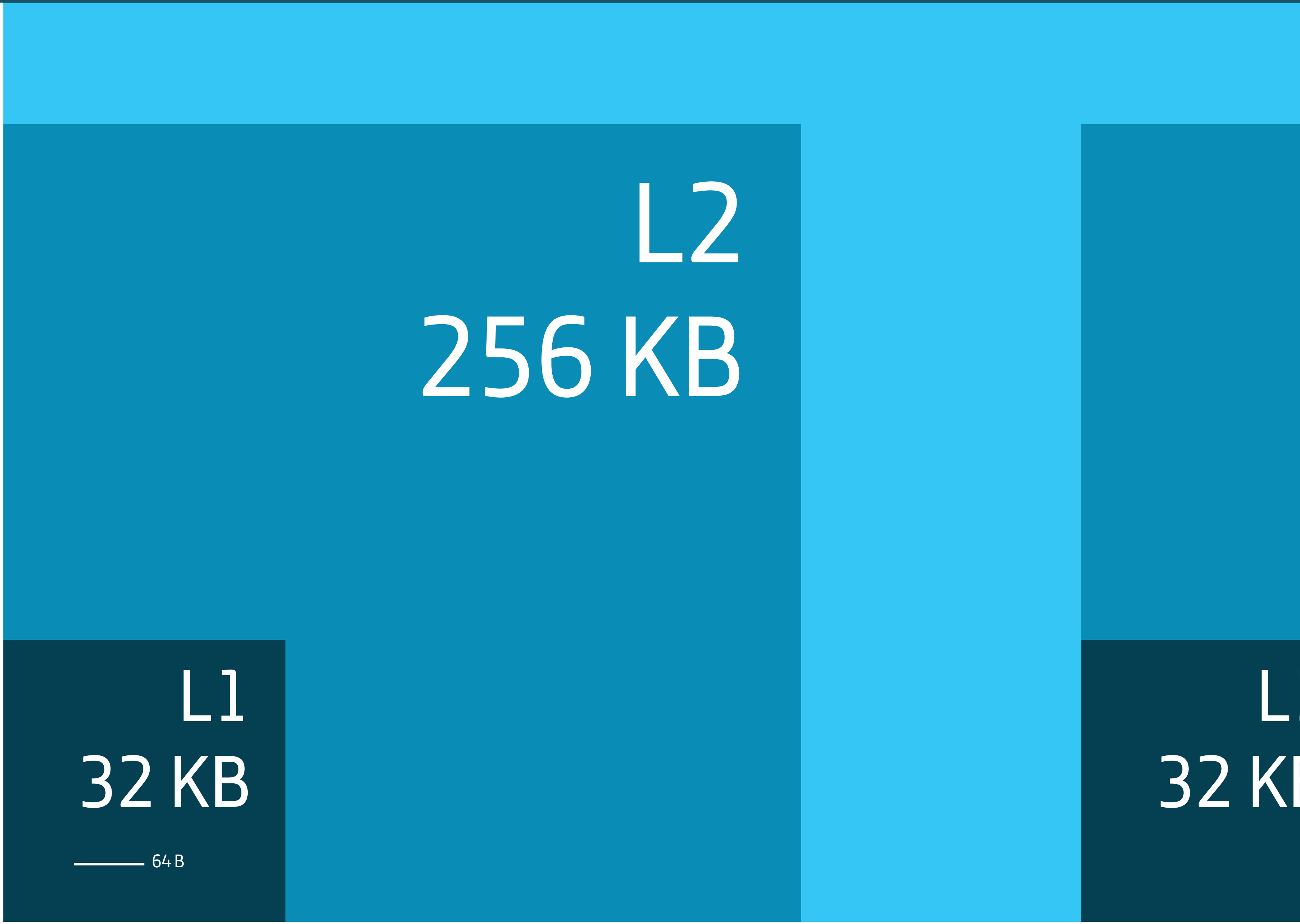
Processor-memory gap



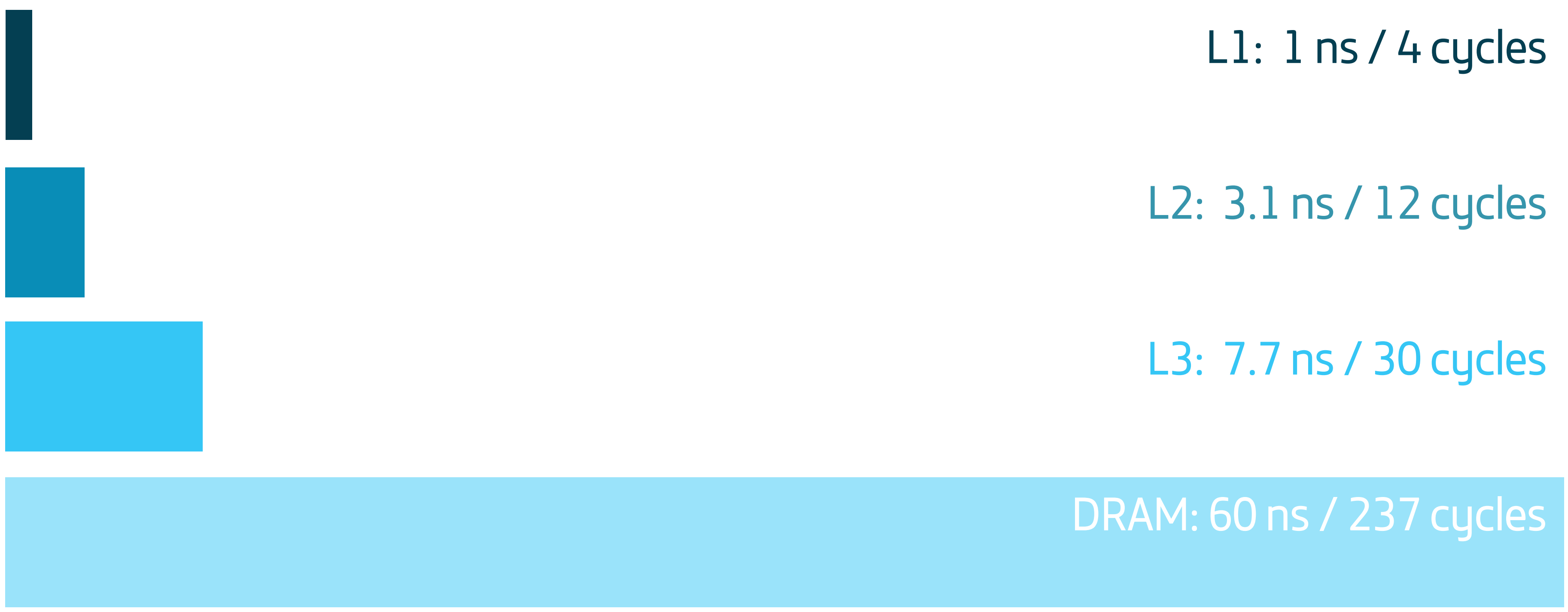
Cache hierarchy



Cache line



Access times



Measure measure measure

Our little measuring tool

```
template<typename F>
double measure(F f)
{
  using namespace std::chrono;

  static const int             num_trials=10;
  static const milliseconds    min_time_per_trial(200);
  std::array<double,num_trials> trials;
  volatile decltype(f())       res; // to avoid optimizing f() away

  for(int i=0;i<num_trials;++i){
    int                                runs=0;
    high_resolution_clock::time_point t2;

    measure_start=high_resolution_clock::now();
    do{
      res=f();
      ++runs;
      t2=high_resolution_clock::now();
    }while(t2-measure_start<min_time_per_trial);
    trials[i]=duration_cast<duration<double>>(t2-measure_start).count()/runs;
  }
  (void)(res); // var not used warn

  std::sort(trials.begin(),trials.end());
  return std::accumulate(trials.begin()+2,trials.end()-2,0.0)/(trials.size()-4)*1E6;
}
```

Profiling environment

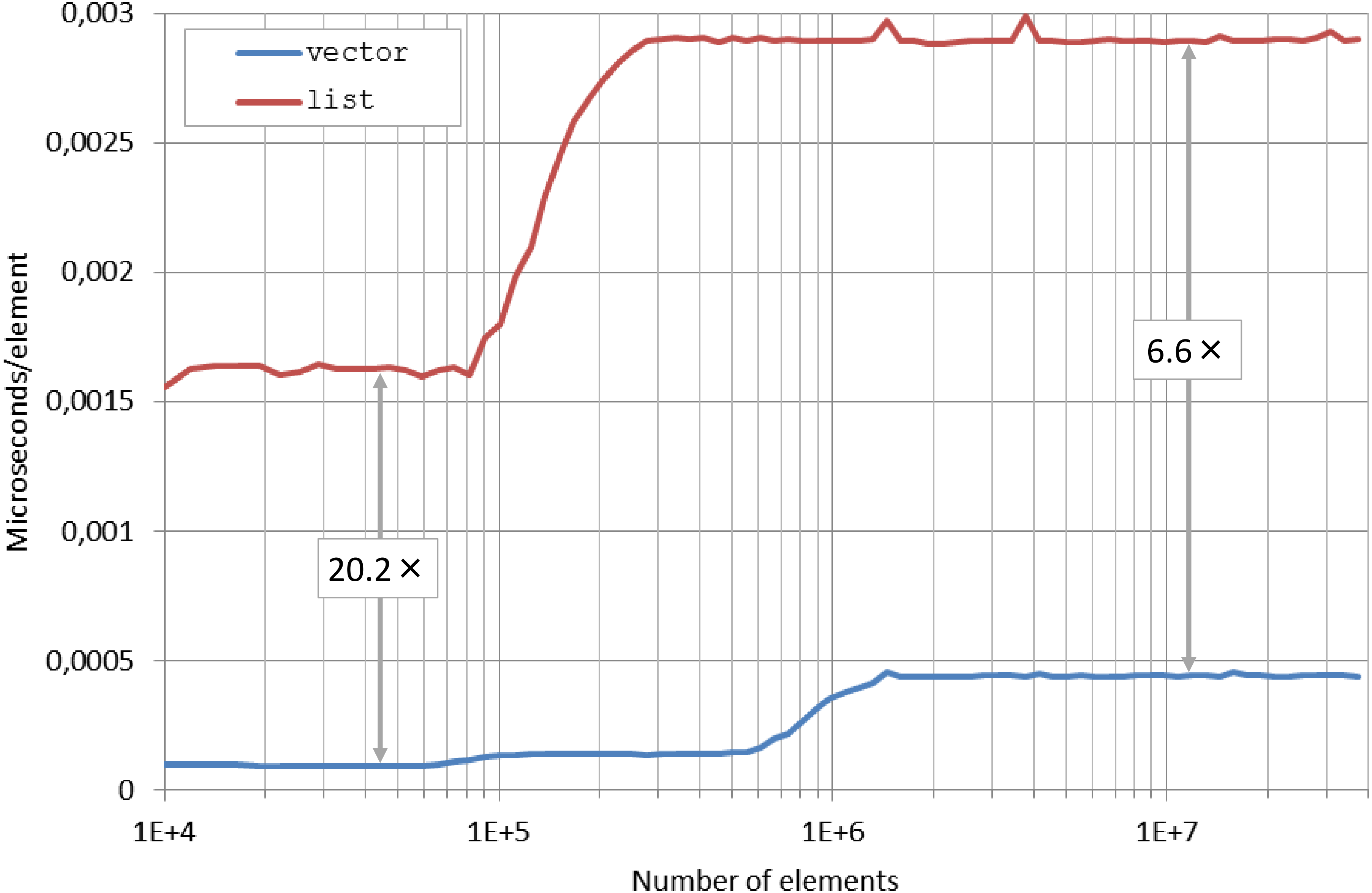
- Intel Core i5-2520M @ 2.5 GHz (Sandy Bridge)
  - L1: 2×32 KB 8-way 3 cycles
  - L2: 2×256 KB 8-way 9 cycles
  - L3: shared 3 MB 12-way 22 cycles
- 4.0 GB DRAM
- Windows 7 Enterprise
- Visual Studio Express 2015
  - x86 mode (32 bit), default release settings

Container traversal

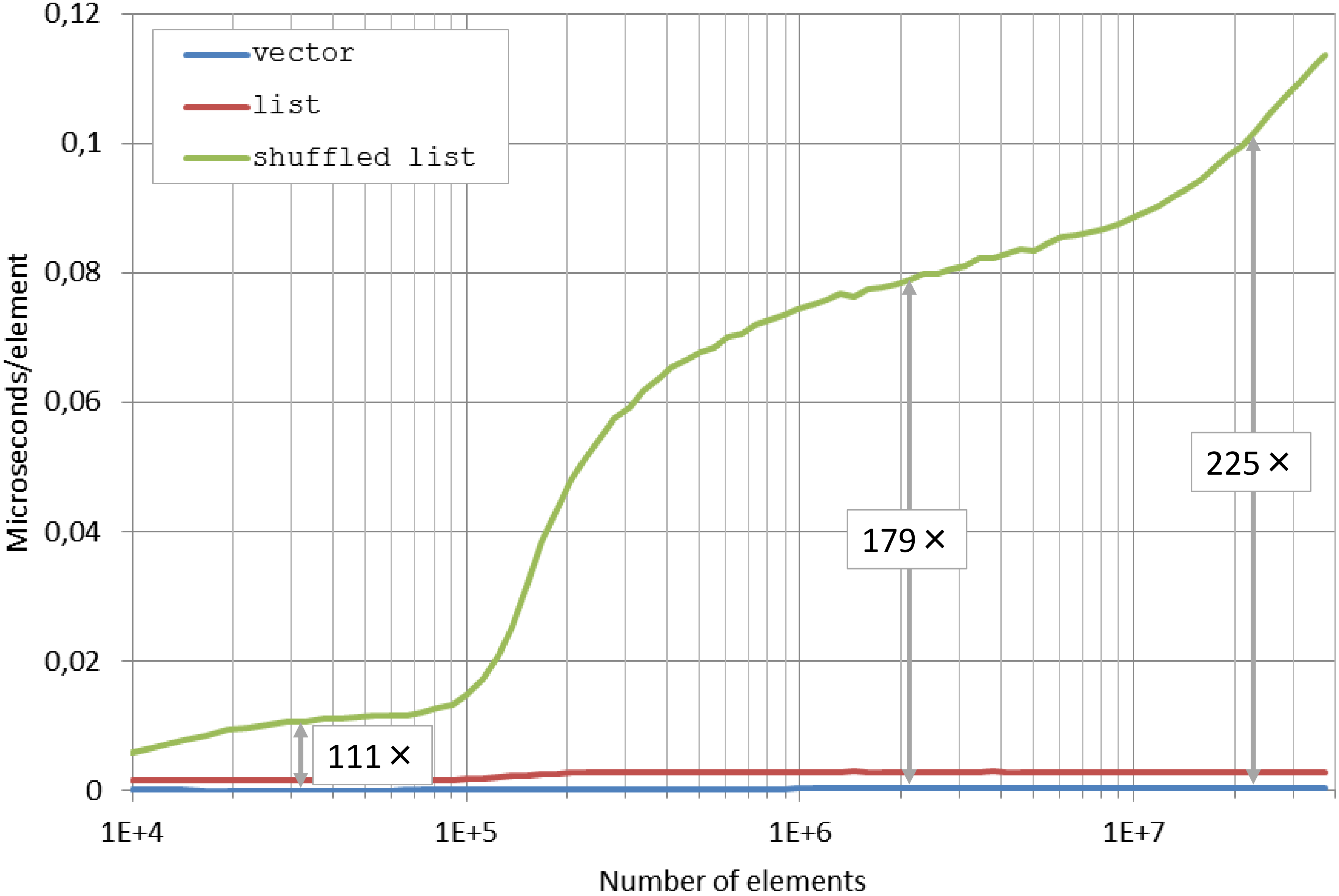
Linear traversal

```
// vector

std::vector<int> v=...;
return std::accumulate(v.begin(),v.end(),0);

// list

std::list<int> l=...;
return std::accumulate(l.begin(),l.end(),0);

// shuffled list

std::list<int> l=...; // nodes shuffled in memory
return std::accumulate(l.begin(),l.end(),0);
```

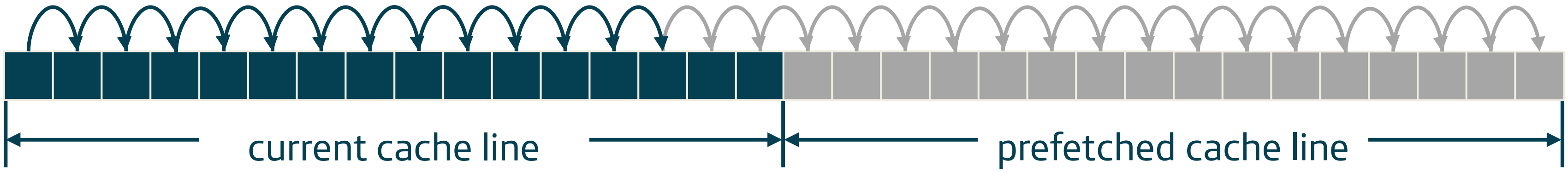
Linear traversal: results



Linear traversal: results



Linear traversal: analysis

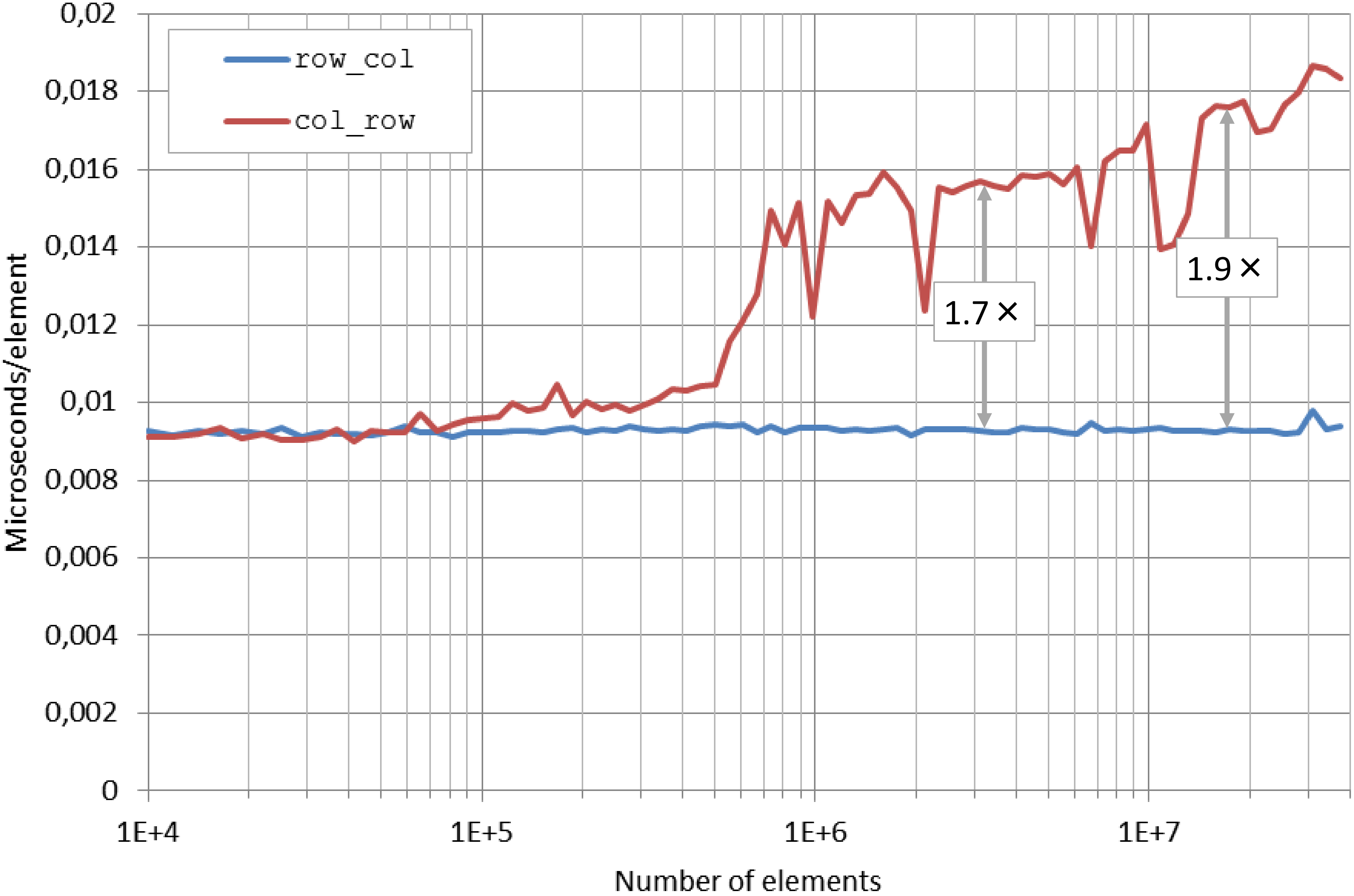


- Two cache aspects to watch out for
  - Locality
  - Prefetching
- std::vector excels at both
- std::list: sequentially allocated nodes provide some sort of non-guaranteed locality
- Shuffled nodes is the worst scenario

Matrix sum

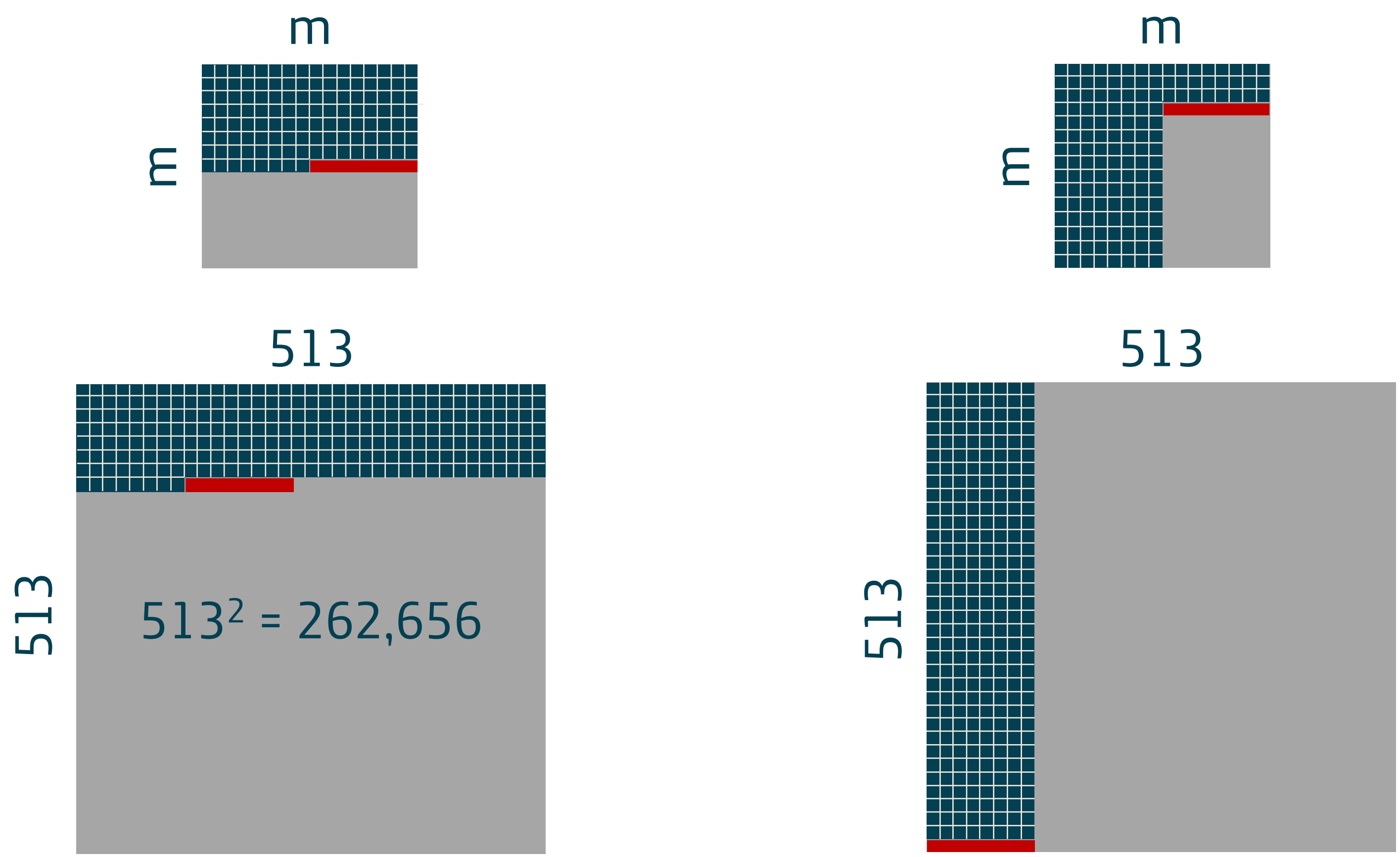
```
boost::multi_array<int,2> a(boost::extents[m][m]);

// row_col

long int res=0;
for(std::size_t i=0;i<m;++i){
  for(std::size_t j=0;j<m;++j){
    res+=a[i][j];
  }
}
return res;

// col_row

long int res=0;
for(std::size_t j=0;j<m;++j){
  for(std::size_t i=0;i<m;++i){
    res+=a[i][j];
  }
}
return res;
```

Matrix sum: results



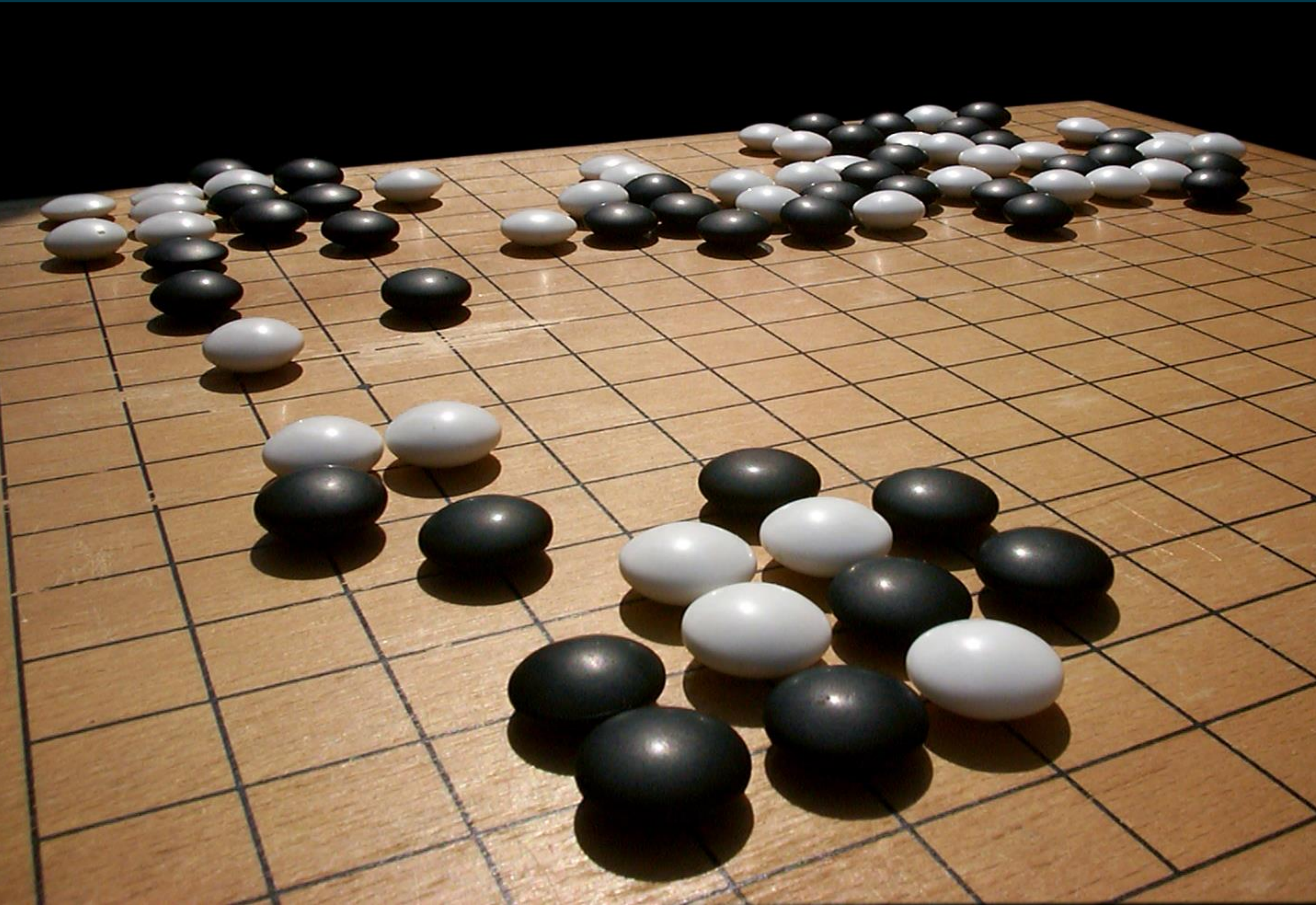
Matrix sum: analysis

- Max # of segments in L1: 32 KB / 64 B = 512



- L2 saturation @ 4097^2 = 16,785,409
- Partial cache associativity makes things actually worse

Layout tricks



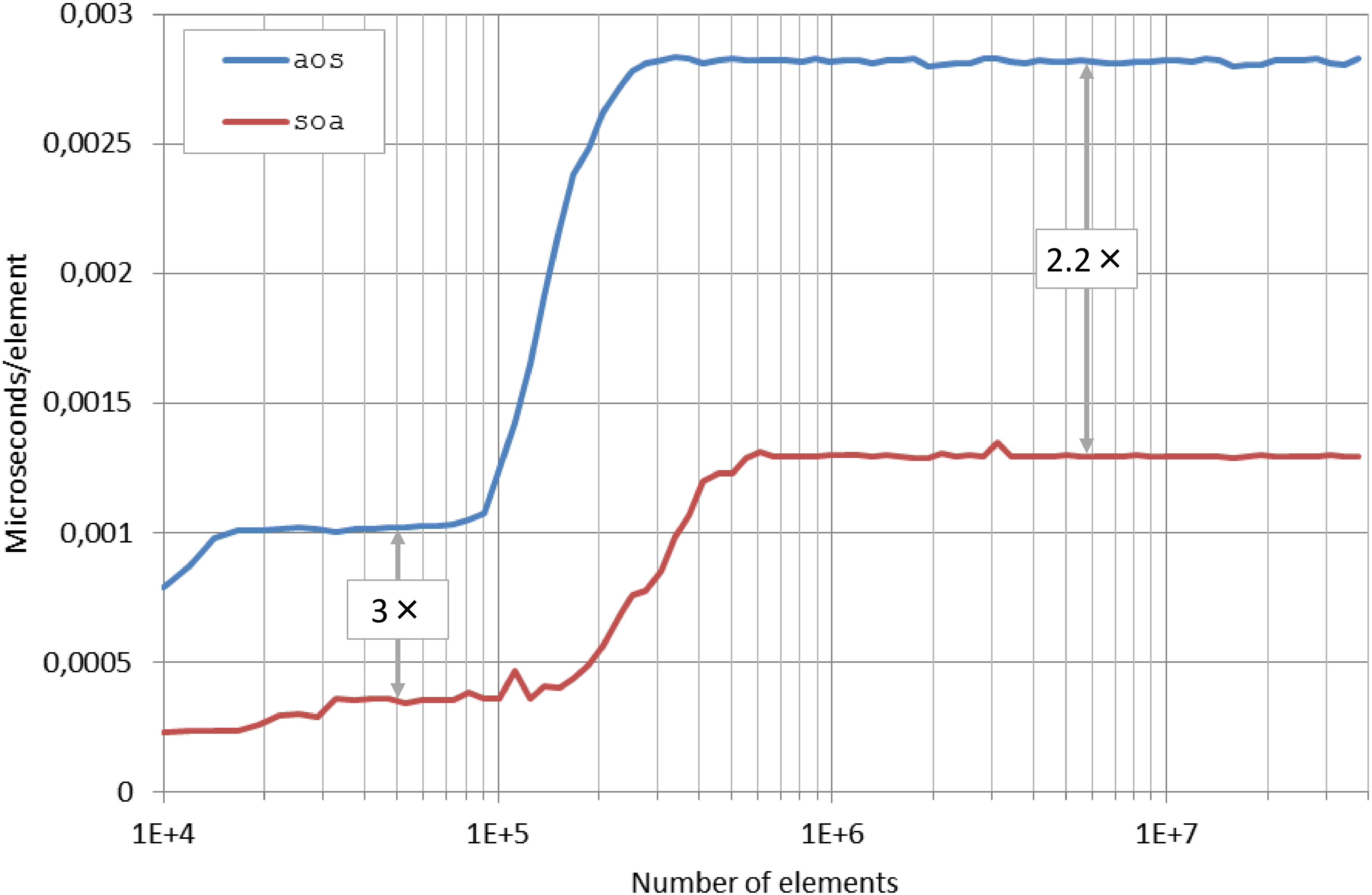
AOS vs SOA

```
// aos    array of structure

struct particle
{
  int x,y,z;
  int dx,dy,dz;
};

using particle_aos=std::vector<particle>;

auto     ps=create_particle_aos(n);
long int res=0;
for(std::size_t i=0;i<n;++i)res+=ps[i].x+ps[i].y+ps[i].z;
return res;

// soa    structure of array

struct particle_soa
{
  std::vector<int> x,y,z;
  std::vector<int> dx,dy,dz;
};

auto     ps=create_particle_soa(n);
long int res=0;
for(std::size_t i=0;i<n;++i)res+=ps.x[i]+ps.y[i]+ps.z[i];
return res;
```

Why SOA is better than AOS?

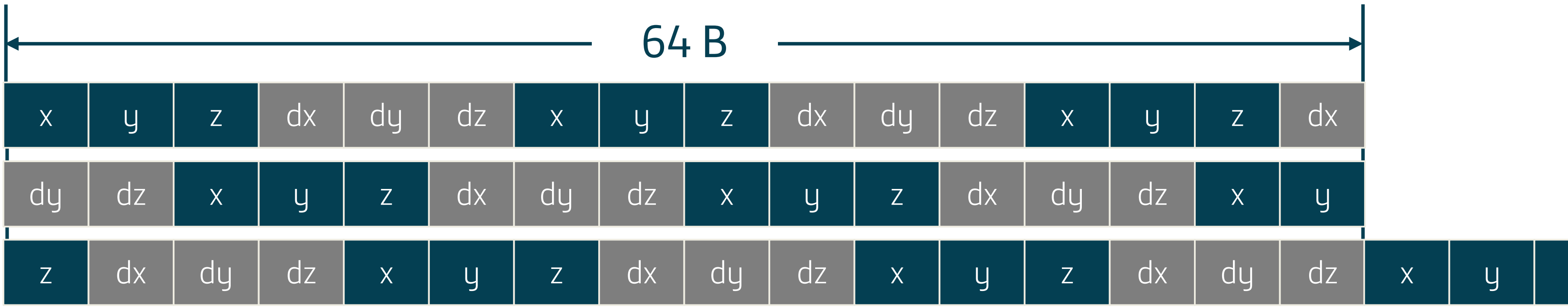
A single SIMD register can load homogeneous data, possibly transferred by a wide internal data path (e.g. 128–bit)

AOS vs SOA: results

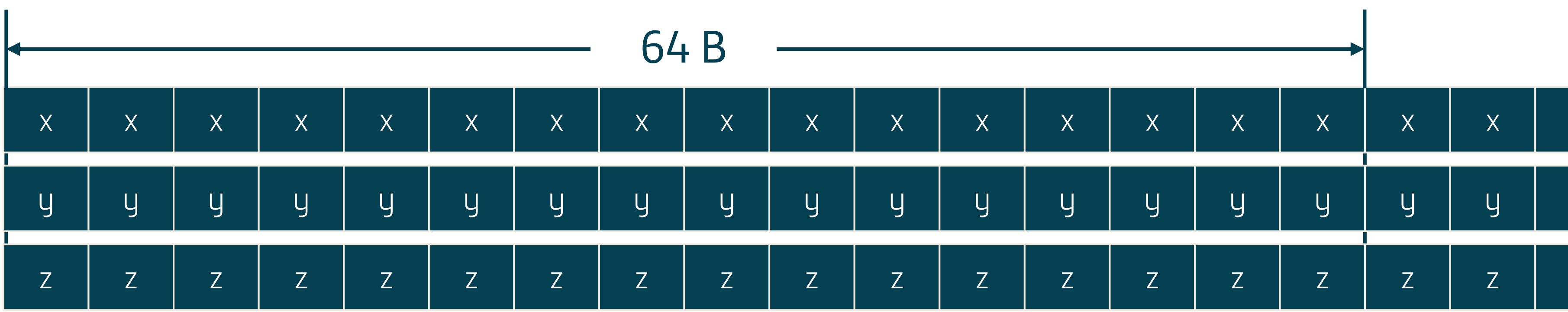


AOS vs SOA: analysis

- AOS cache load throughput: 8 *useful* values per fetch (average)



- SOA throughput: 16 useful values per fetch



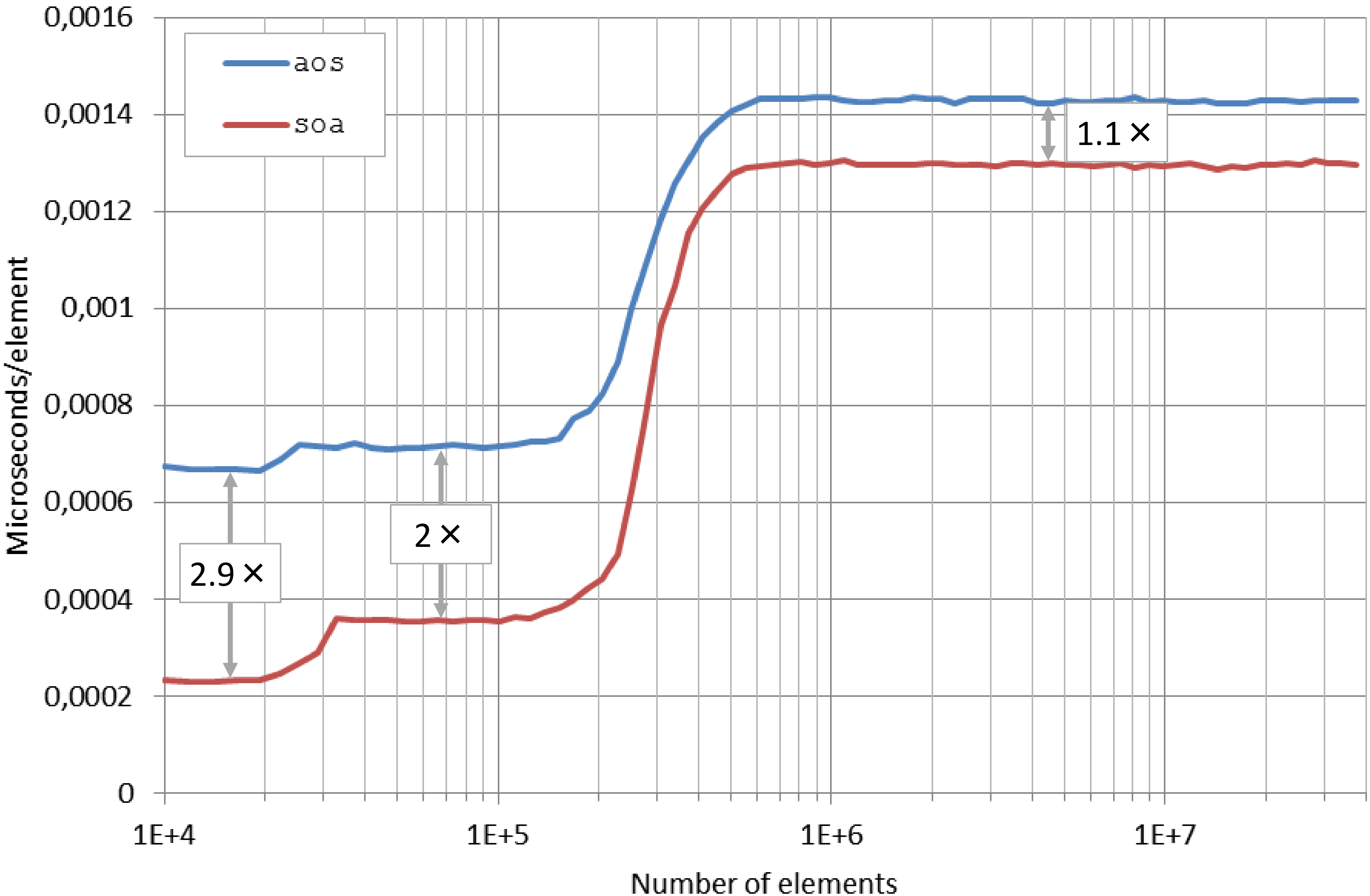
- 16/8 = 2 times faster

- Yet results are even better than this!

Compact AOS vs SOA

```
// aos

struct particle
{
  int x,y,z;
};

using particle_aos=std::vector<particle>;

// ...same as before

// soa

struct particle_soa
{
  std::vector<int> x,y,z;
};

// ...same as before
```

Compact AOS vs SOA: results



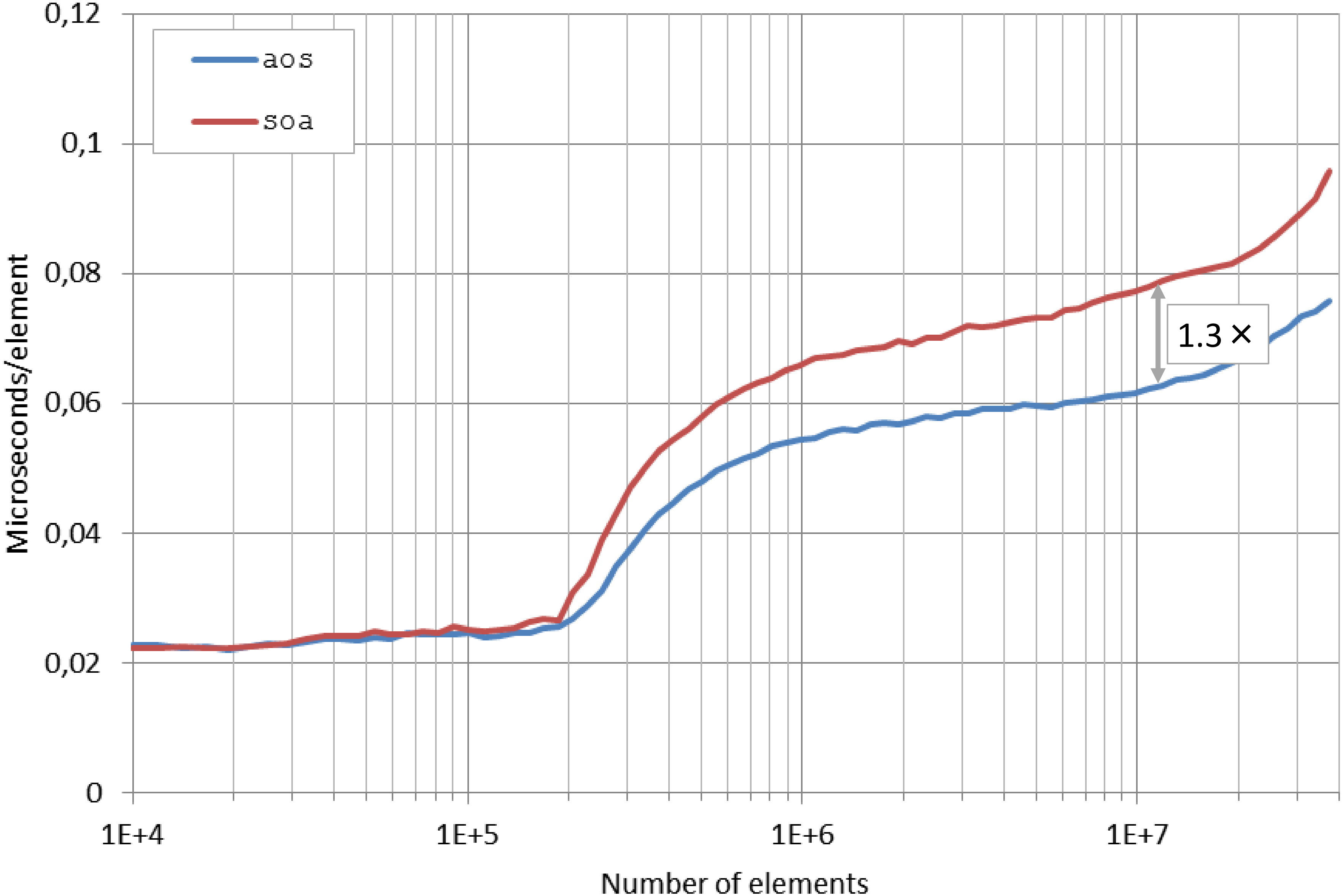
Compact AOS vs SOA: analysis

- Both AOS and SOA fetch only useful values
- Yet SOA is (diminishingly as n grows) faster
- The CPU is prefetching x, y, z *in parallel*
- L1← L2 better than L1 ← L2 ← L3 better than L1 ← ··· ← DRAM

Random access in AOS vs SOA

```
// aos

auto     ps=create_particle_aos(n);
long int res=0;
for(std::size_t i=0;i<n;++i){
  auto idx=rnd(gen);
  res+=ps[idx].x+ps[idx].y+ps[idx].z;
}
return res;

// soa

auto     ps=create_particle_soa(n);
long int res=0;
for(std::size_t i=0;i<n;++i){
  auto idx=rnd(gen);
  res+=ps.x[idx]+ps.y[idx]+ps.z[idx];
}
return res;
```

Random access in AOS vs SOA: results



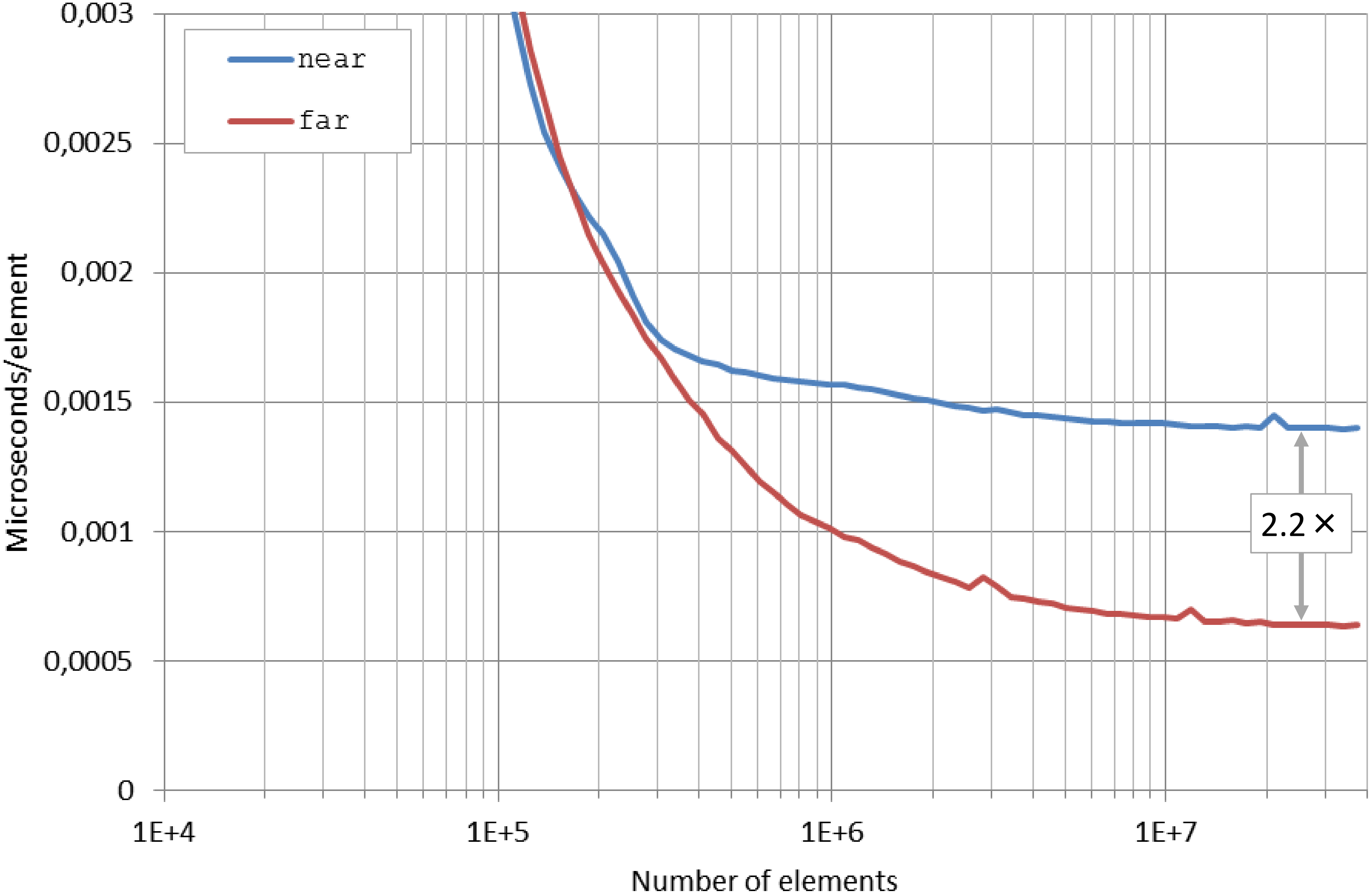
Random access in AOS vs SOA: analysis

- You're now in a position to make an educated guess ☺
- Moral: don't trust your intuition

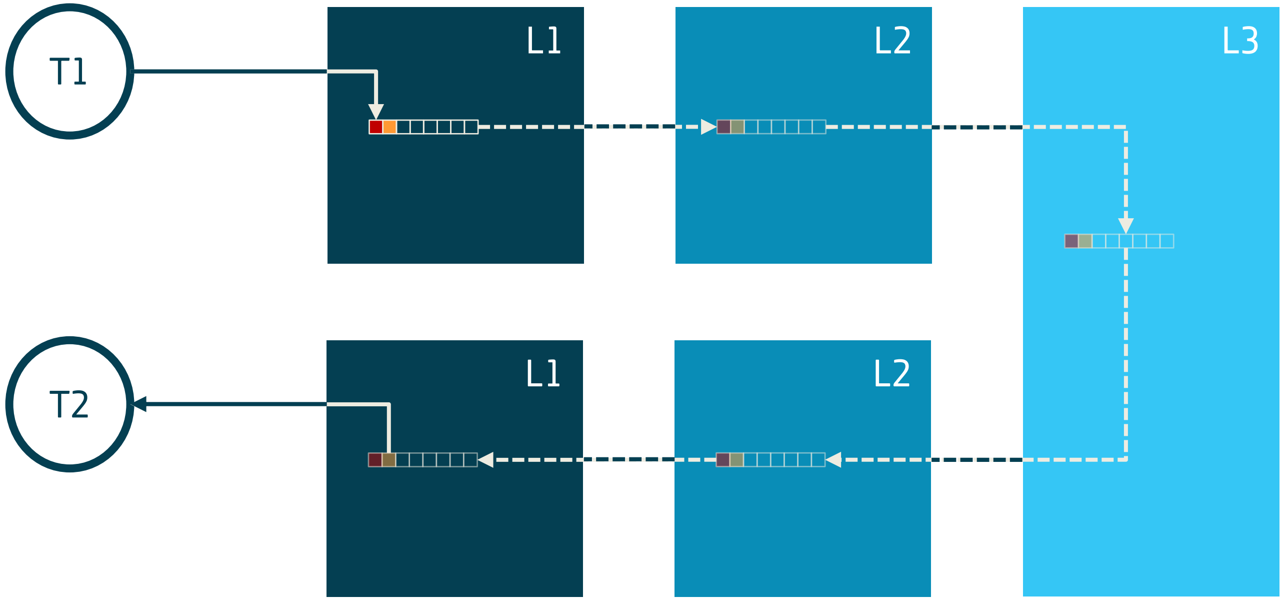
Parallel count

```
std::vector<int> v=...;

auto f=[&](int* px,int* py,int* pz,int* pw){
  auto th=[](int* p,int* first,int* last){
    *p=0;
    while(first!=last){
      int x=*first++;
      *p+=x%2;
    }
  };
  std::thread t1(th,px,v.data(),v.data()+n/4);
  std::thread t2(th,py,v.data()+n/4,v.data()+n/2);
  std::thread t3(th,pz,v.data()+n/2,v.data()+n*3/4);
  std::thread t4(th,pw,v.data()+n*3/4,v.data()+n);
  t1.join();t2.join();t3.join();t4.join();
  return *px+*py+*pz+*pw;
};

int res[49];

// near result addresses

return f(&res[0],&res[1],&res[2],&res[3]);

// far result addresses

return f(&res[0],&res[16],&res[32],&res[48]);
```

Parallel count: results



False sharing



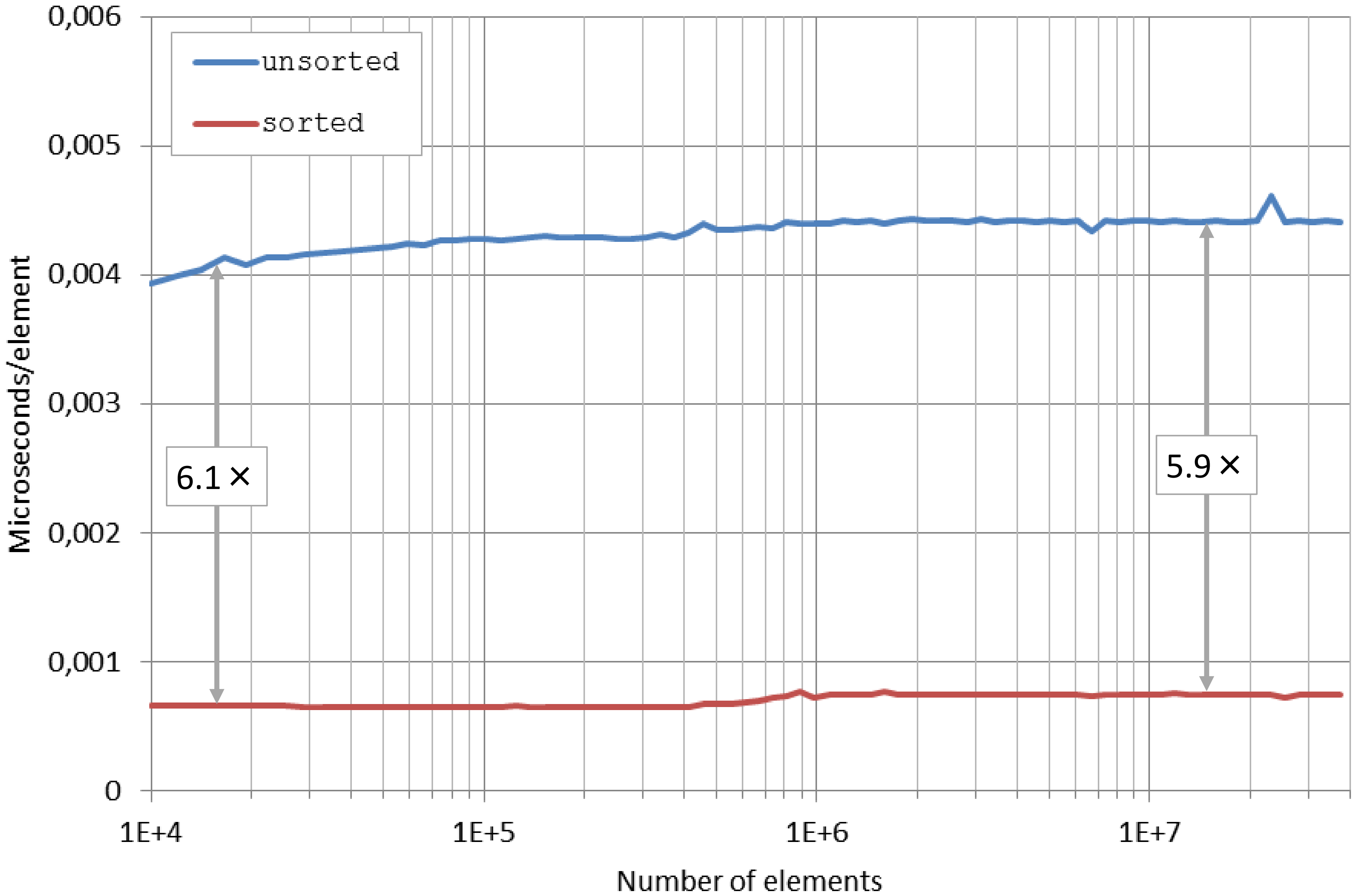
- Writing to res[0] invalidates cached res[1]
- Coherence management requires full write to DRAM
- Rule of thumb: use shared write memory *only* to communicate thread results

Branch prediction

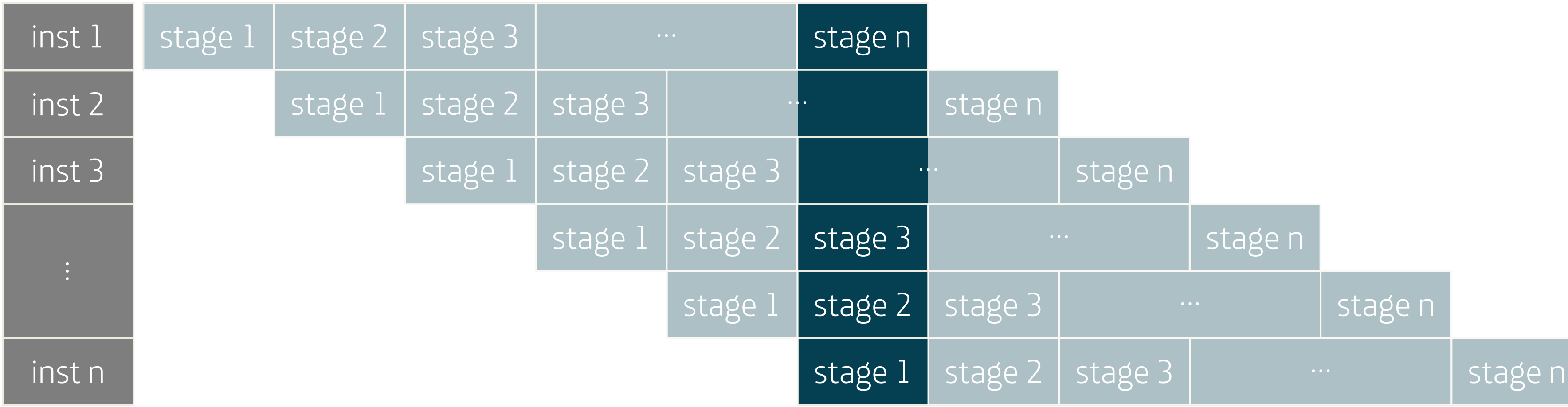
Filtered sum

```
std::vector<int> v=...;

// unsorted

long int res=0;
for(int x:v)if(x>128)res+=x;
return res;

// sorted

std::sort(v.begin(),v.end());

long int res=0;
for(int x:v)if(x>128)res+=x;
return res;
```

Filtered sum : results



Instruction pipeline



- Increases execution throughput by n (latency remains the same)
  - Sandy Bridge: 14-17 stages
  - Ivy Bridge: 14-19 stages
- Branch prediction failure invalidates partially executed queue

A particularly obnoxious case: bool-based processing

```
struct particle{
  bool active;
  int  x,y,z;
  ...
};
std::vector<particle> particles;

void render_particles(){
  for(const particle& p:particles)if(p.active)render(p);
}
```

- Remove active flag by smartly laying out objects

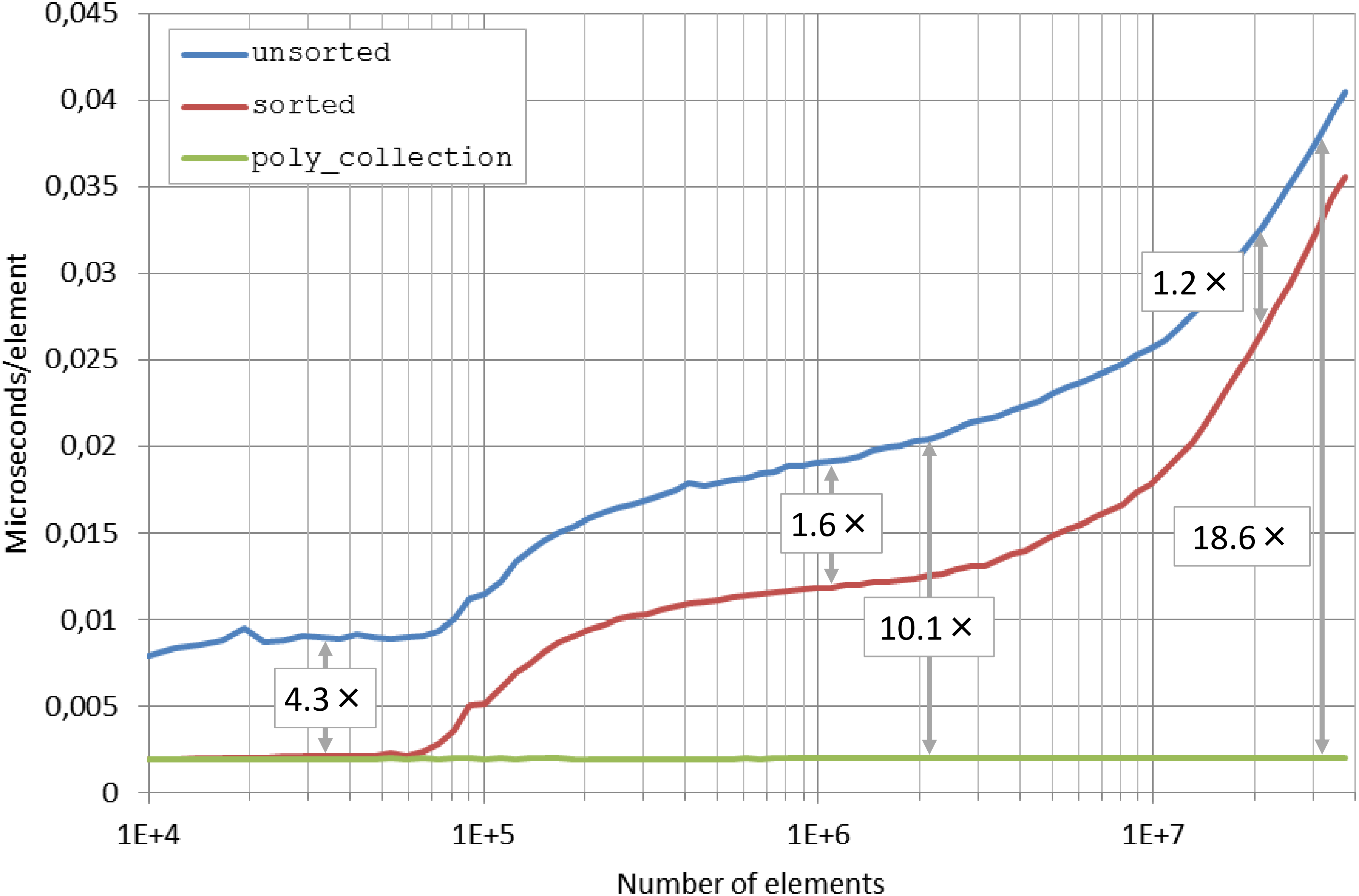
```
struct particle{
  int  x,y,z;
  ...
};
std::vector<particle> particles;
std::size_t           num_active;

void render_particles(){
  for(std::size_t i=0;i<num_active;++i)render(particles[i]);
}
```

- Better cache density, no branching, fewer elements processed

Polymorphic containers

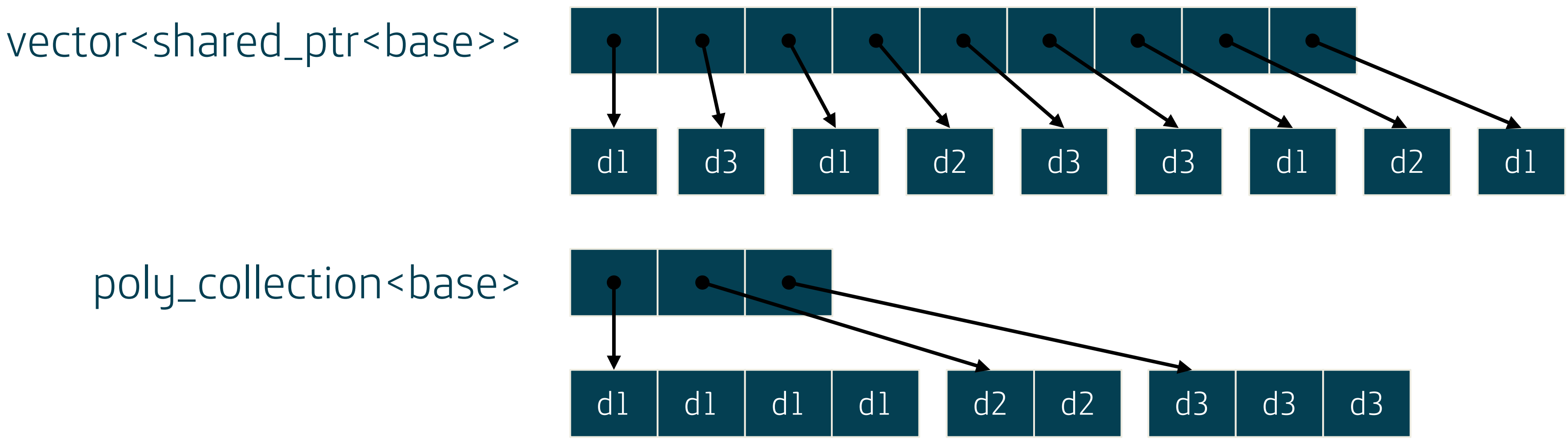
```
struct base{virtual int f()const=0; virtual ~base(){}};
struct derived1:base{virtual int f()const{return 1;};};
struct derived2:base{virtual int f()const{return 2;};};
struct derived3:base{virtual int f()const{return 3;};};
using pointer=std::shared_ptr<base>;
std::vector<pointer> v=...;

// unsorted

long int res=0;
for(const auto& p:v)res+=p->f();
return res;

// sorted

std::sort(v.begin(),v.end(),[](const pointer& p,const pointer& q){
  return std::type_index(typeid(*p))<std::type_index(typeid(*q));
});

long int res=0;
for(const auto& p:v)res+=p->f();
return res;

// poly_collection: more on this later

poly_collection<base> v=...;
v.for_each([&](const base& x){res+=x.f();});
```

Polymorphic containers: results



Polymorphic containers: analysis

- Sorting improves branch prediction even without data locality
- What is this poly_collection thing?



- Branch prediction-friendly + optimum data locality
- More info on **http://tinyurl.com/poly-collection**
- Cutting-edge compilers: look for *devirtualization*

Concluding remarks

Concluding remarks

- You can't just ignore reality if you're serious about performance
- Think about the planet: don't waste energy
- Measure measure measure
- (Almost) nothing beats traversing a vector in its natural order
  - One indirection ≈ one cache miss
- If multithreading, beware of false sharing
- SOA can get you huge speedups (and can not)
- Can you remove flags?
  - Look for *data-oriented design* if you want to know more
- Be regular: sort your objects

Mind the cache

Thank you

github.com/joaquintides/usingstdcpp2015

using std::cpp 2015
Joaquín M López Muñoz <joaquin@tid.es>
Madrid, November 2015
Telefónica