Know Your Hardware: CPU Memory Hierarchy

Alexander Titov

About me

Alexander Titov

- CPU Hardware Architect
- 10+ years of C++ experience (CPU simulation)
- Teaching Computer Architecture and Design

alexander.titov@atitov.com

alexander-titov-cpu

Performance Optimizations Rules

1. Measure it!
2. Do educated decisions
3. Do not reinvent the wheel
4. Optimize in the following order
   - General algorithm (e.g., `O(NlogN)` vs. `O(N^2)`)
   - Memory allocation, copy vs. move, etc.
   - **Data organization in memory and access patterns**
   - Code footprint
   - Branches
   - …

Why Is Memory So Important?

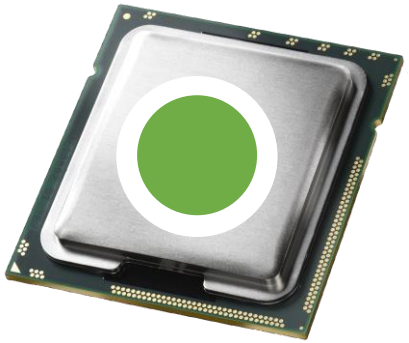
- Simplified steps to execute an instruction:
  1. Read instruction from **memory**
  2. Decode it
  3. Read inputs from **memory**
  4. Execute instruction
  5. Write result to **memory**

- **CPU works a lot with Memory**

Is Memory Fast?

- Simplified steps to execute an instruction:
  1. Read instruction from **memory**
  2. Decode it
  3. Read inputs from **memory**
  4. Execute instruction
  5. Write result to **memory**

~0.5 ns



CPU

Is Memory Fast?

- Simplified steps to execute an instruction:
  1. Read instruction from **memory**
  2. Decode it
  3. Read inputs from **memory**
  4. Execute instruction
  5. Write result to **memory**

~0.5 ns

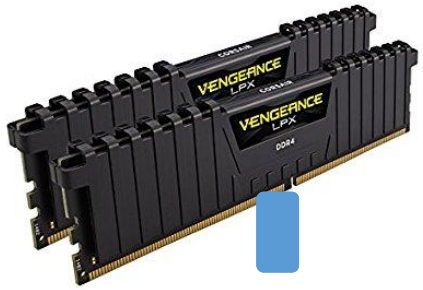
CPU

Is Memory Fast? – No

~100 ns

⏩ 4x

~0.5 ns



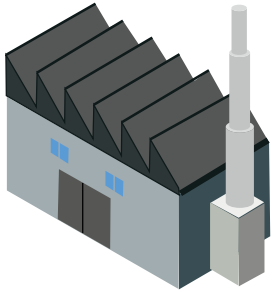
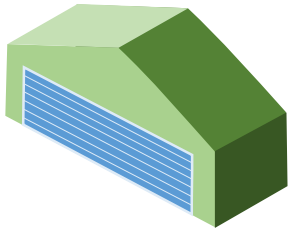
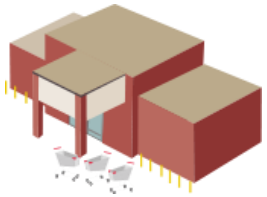
Memory

CPU

Memory is very slow.

CPU would wait 99% of the time for Memory response.

Cache Hierarchy in Real Life

	3rd level cache	2nd level cache	1st level cache	
				
Factory	Warehouse	Large store	Small shop	Home

Cache Hierarchy

Memory

3rd level cache

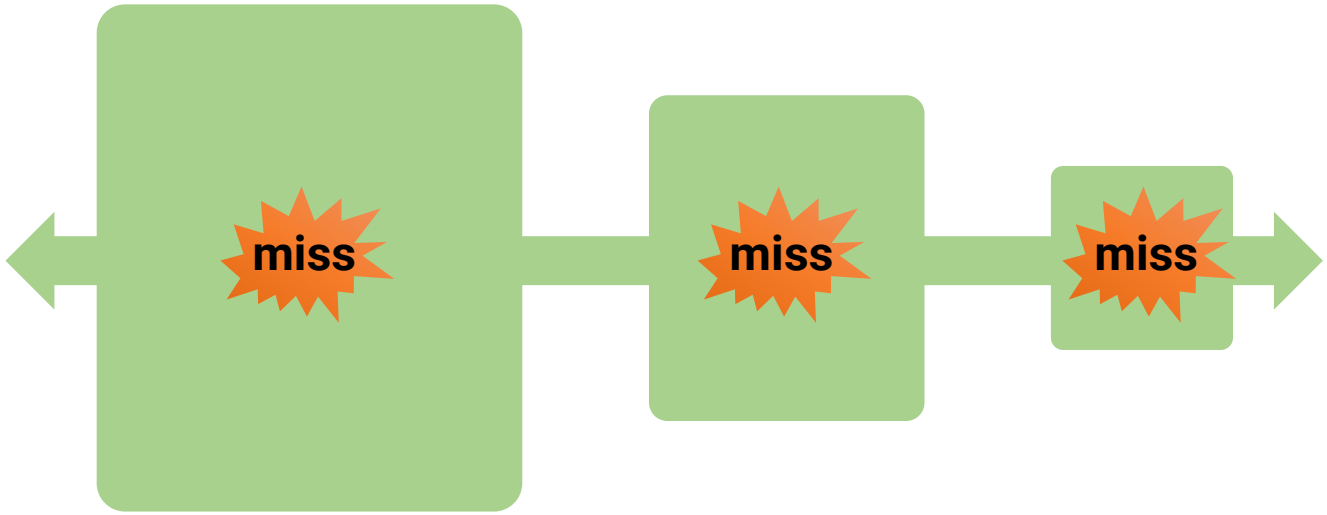
2nd level cache

1st level cache

CPU

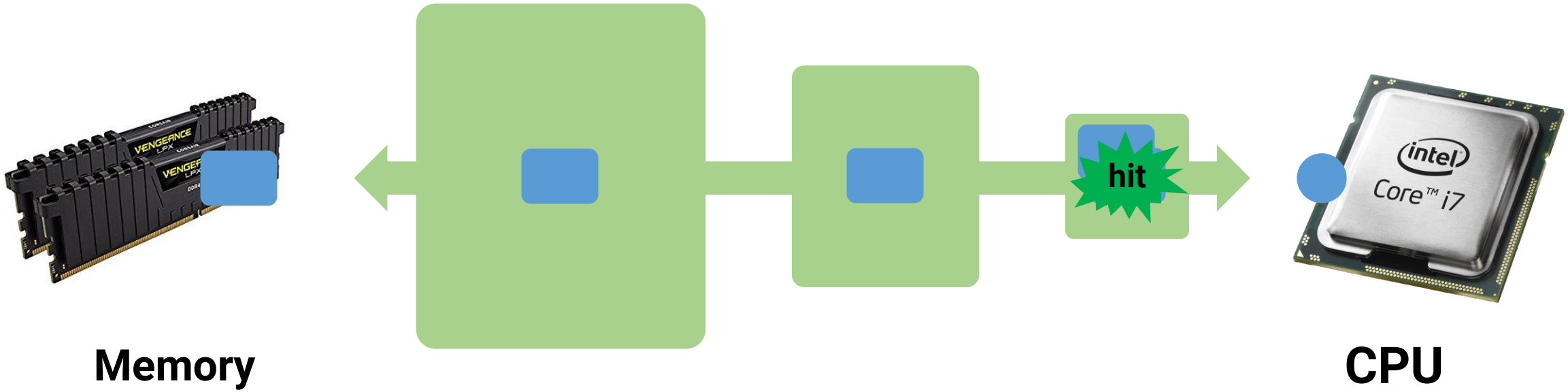
Cache Hierarchy In Action

Memory



CPU

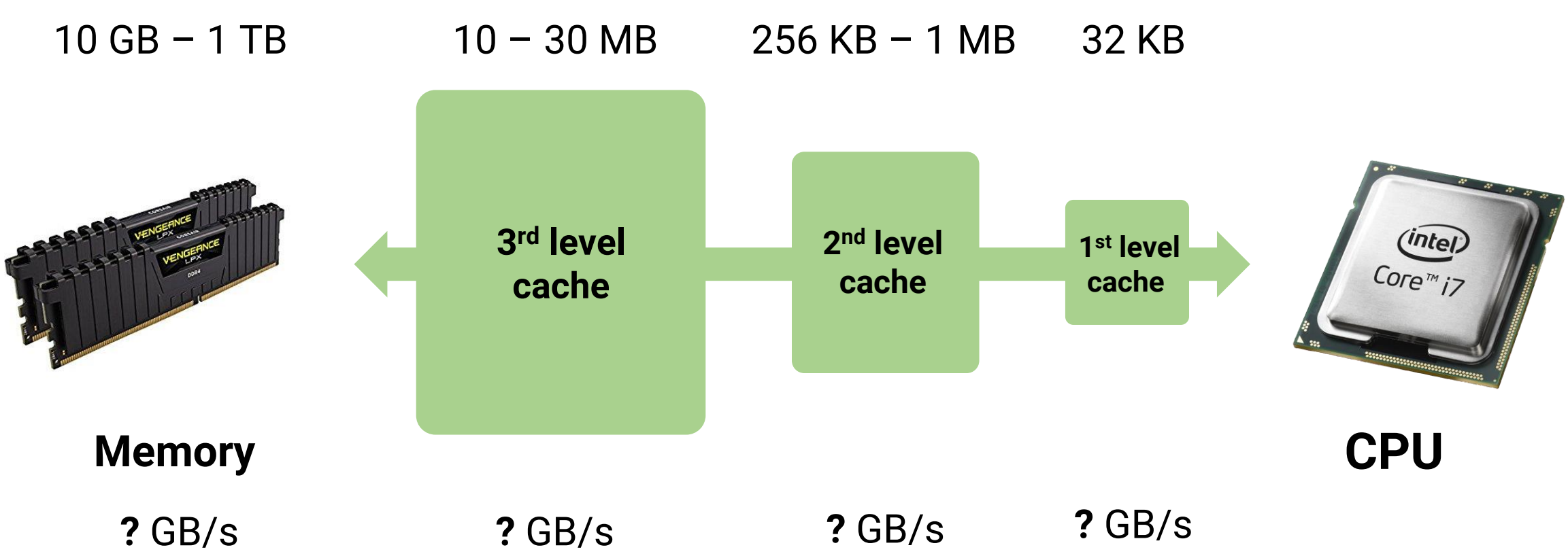
Cache Hierarchy In Action



Locality principle:

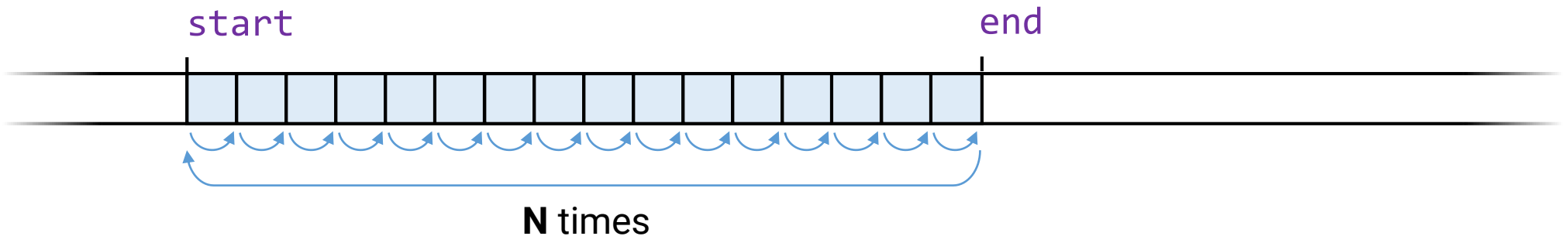
the same data is requested several times in a short period of time

Cache Hierarchy



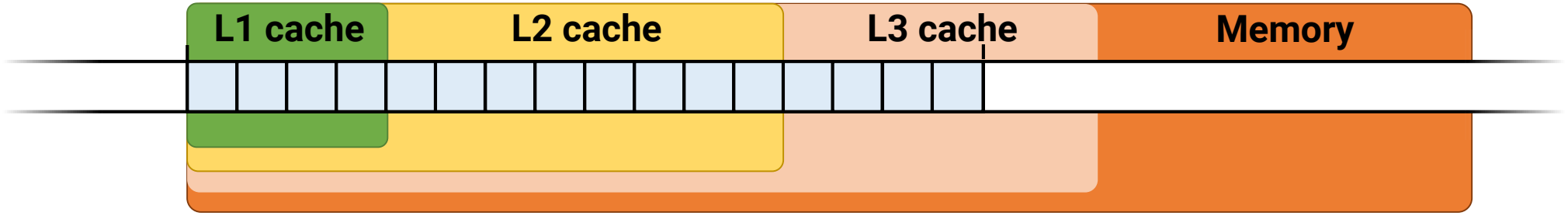
Experiment: Defining Cache Hierarchy

- Benchmark structure

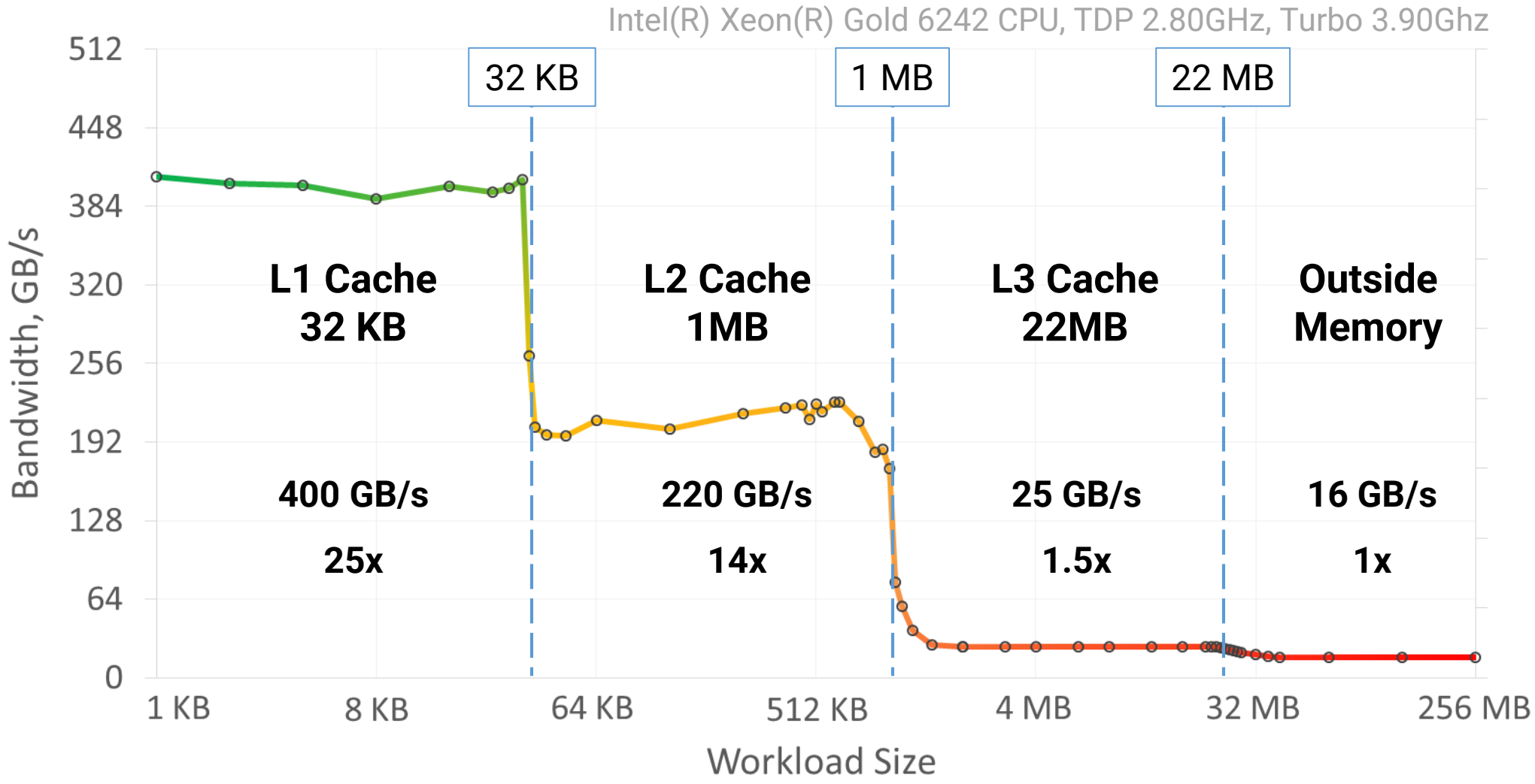


Experiment: Defining Cache Hierarchy

- Benchmark structure



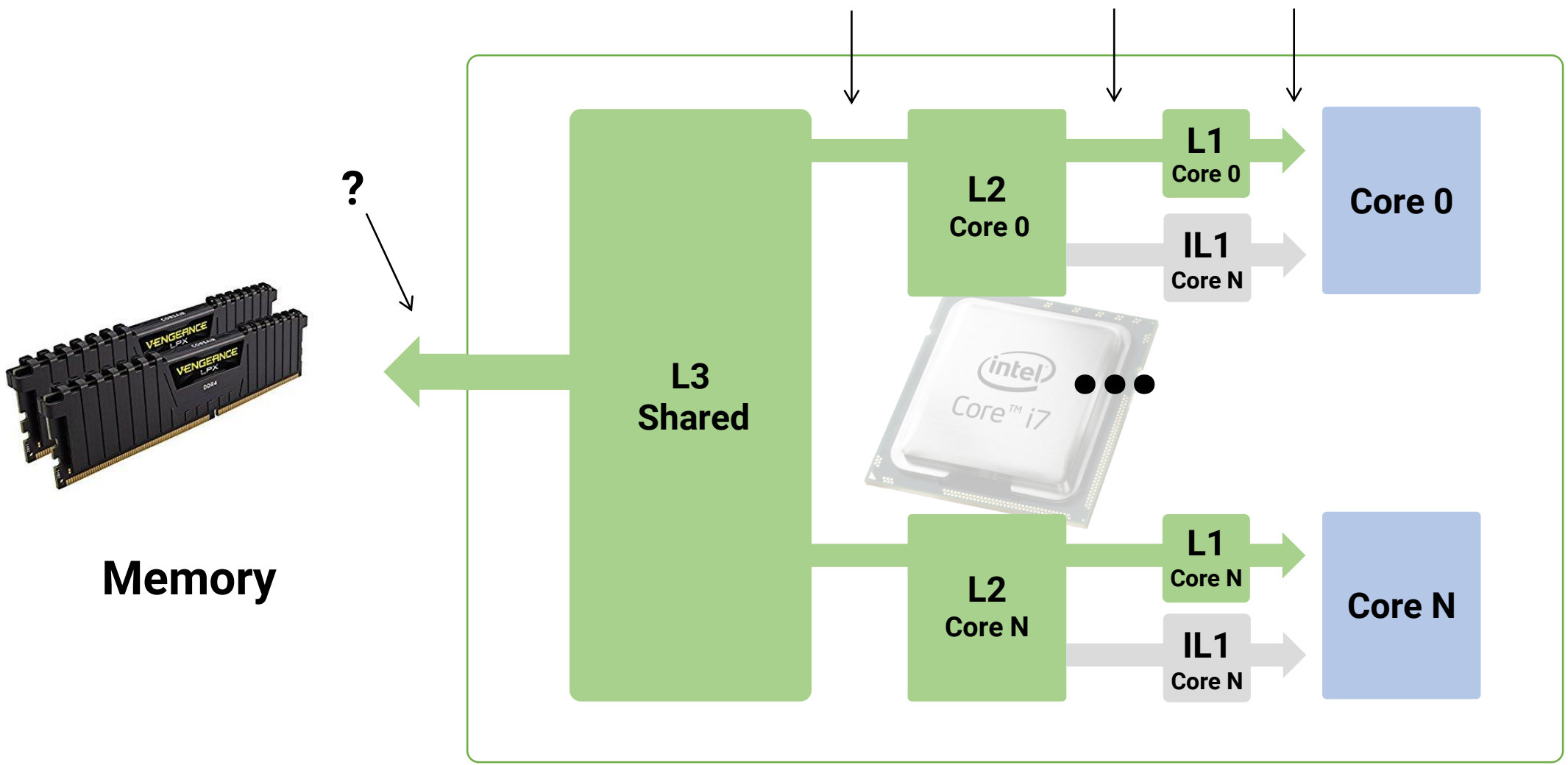
Experiment: Defining Cache Hierarchy



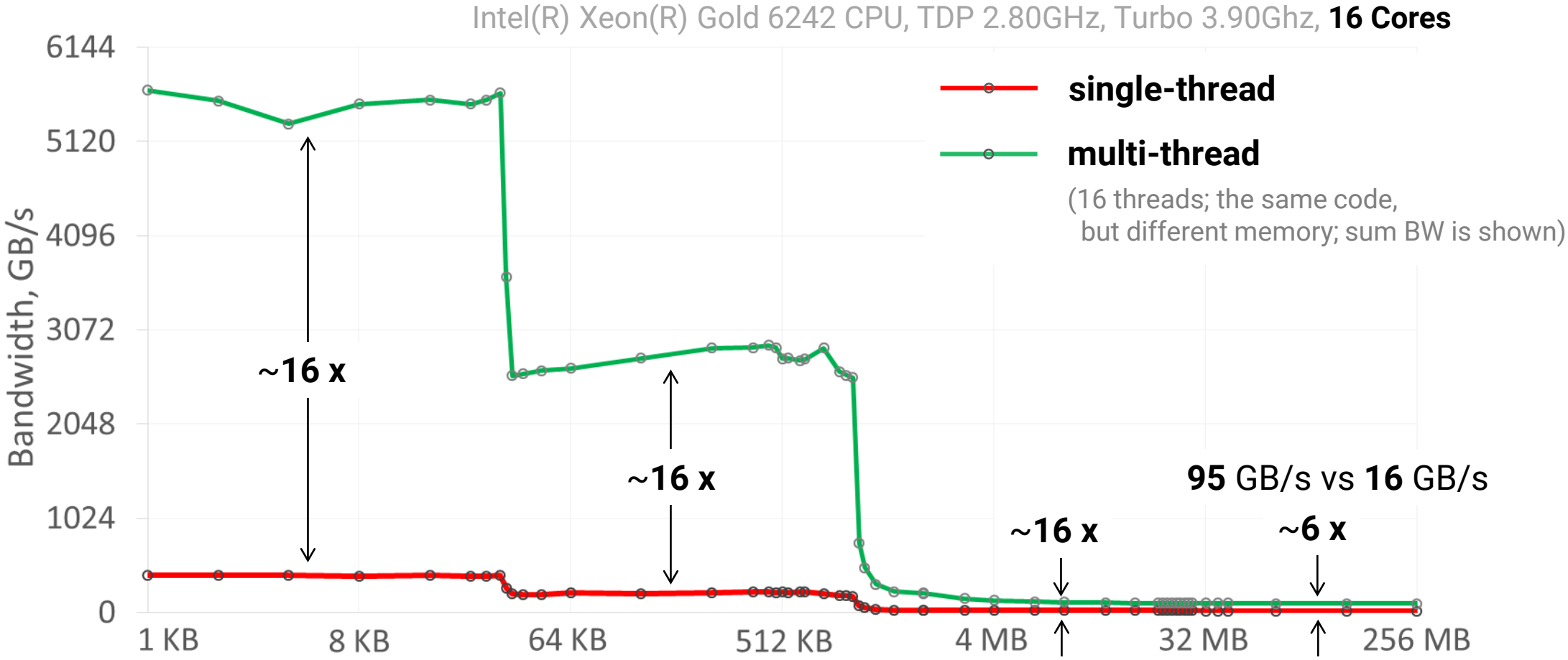
Multi-Core CPU

Memory

Multi-Core CPU



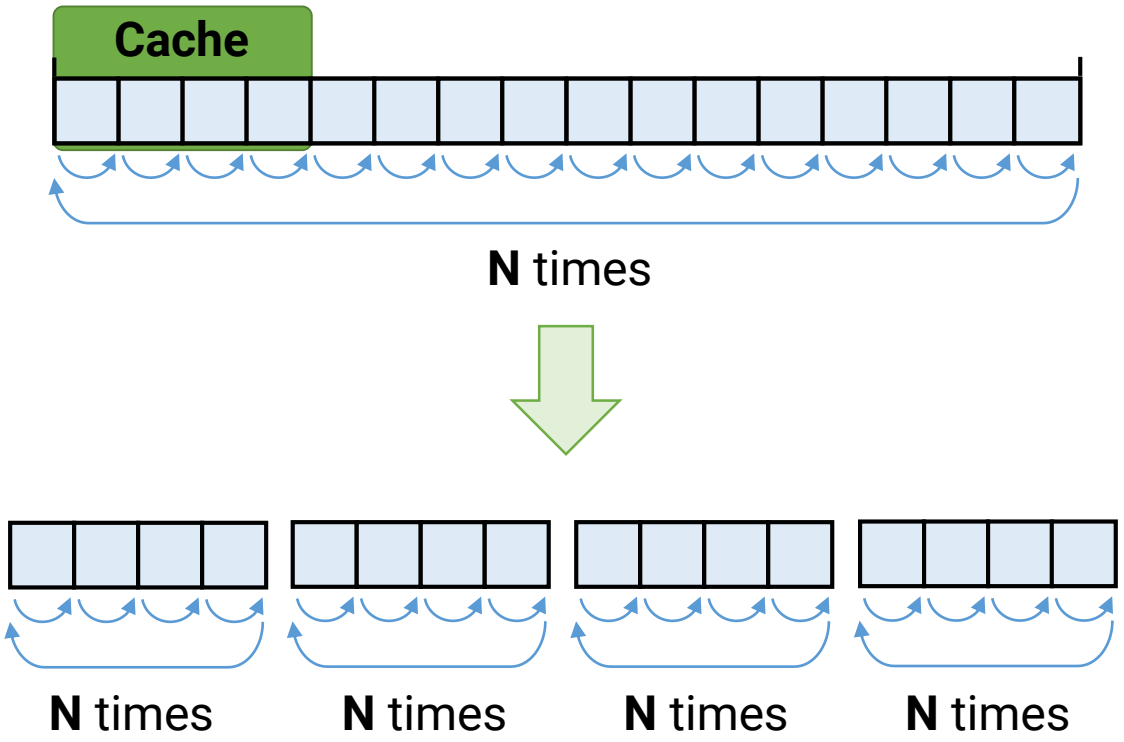
Experiment: Single-Core vs. Multi-Core



- **Multi-threaded application receives not only N x cache space and BW, but large Memory BW too**

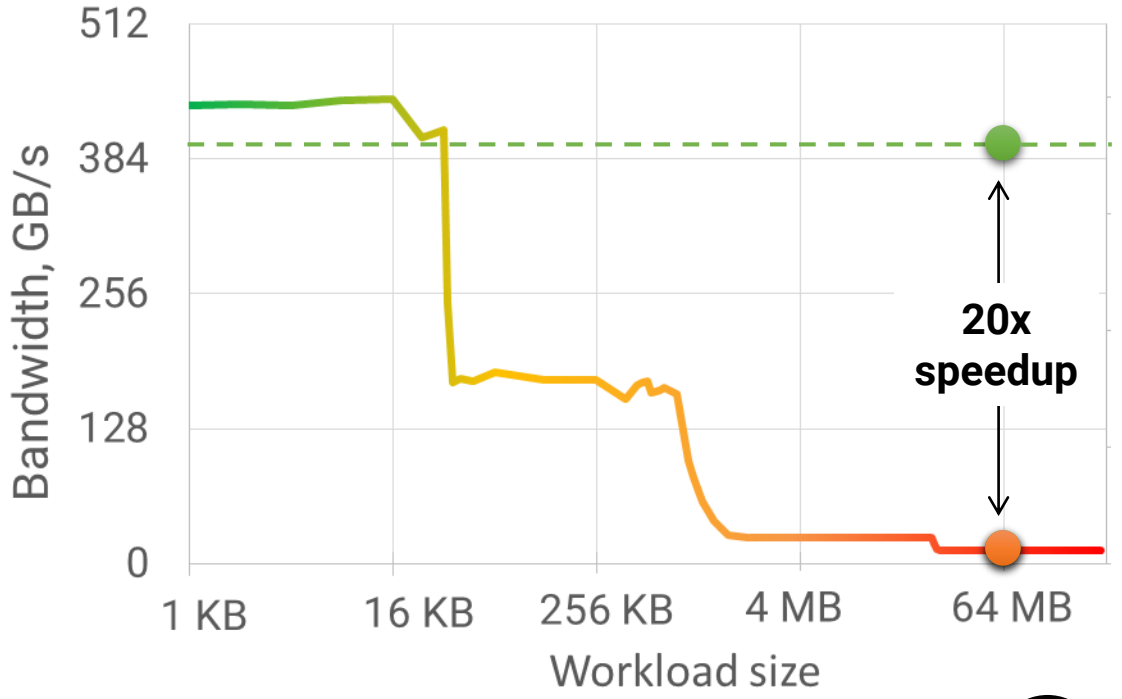
Tip: Divide and Conquer

- Split large workload in parts that fit in the cache → improve locality



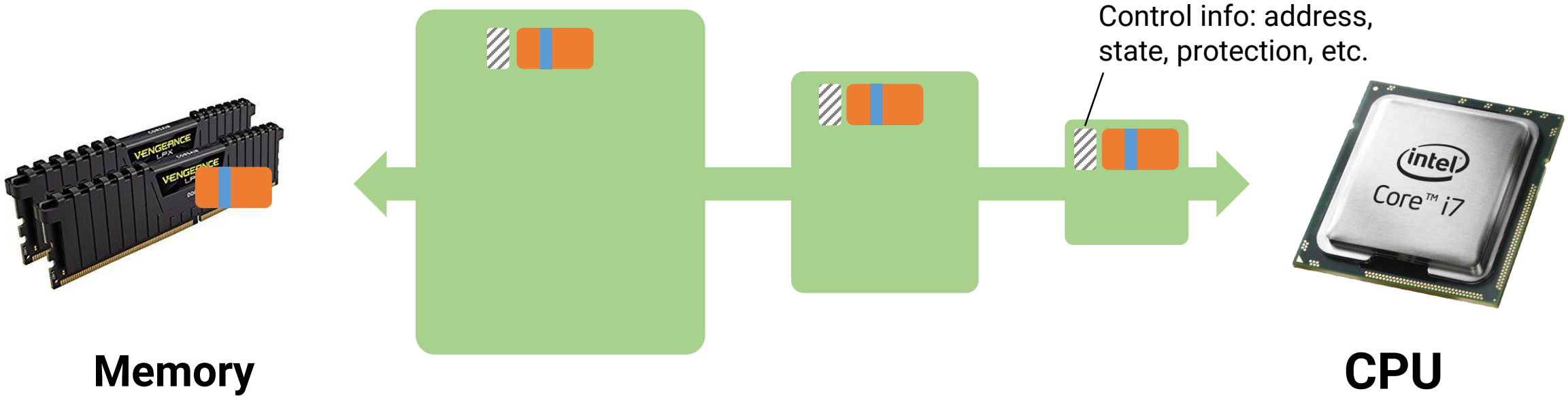
Example:

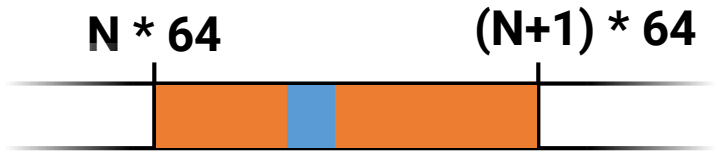
64 MB x 1000 times → 4096 parts x (16 KB x 1000 times)



Memory line

- **All data transfers are performed in aligned chunks of 64 B (memory line)**
  - If the rest of the line is not used, it occupies space and BW in vain



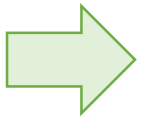


- **The better line utilization → the better performance**

Tip: Split Warm and Cold Data

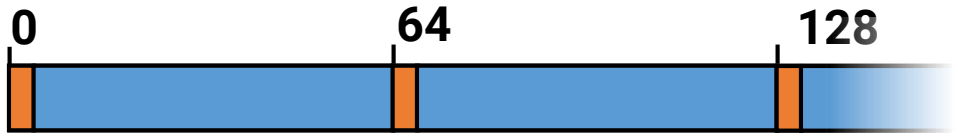
- Extract fields that are accessed more often (warm) into a separate object

```
struct Mixed {
    int warm;
    int cold[15];
};

Mixed mixed_array[N];
```

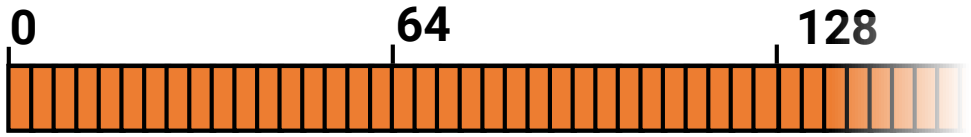


```
struct Warm {
    int warm;
};

Warm warm_array[N];
Cold cold_array[N];
```

```
struct Cold {
    int cold[15];
};
```



6% efficiency



100% efficiency

- Decrease occupied cache space and memory bandwidth by **16x times**!

Tip: Dense Data Packing

- By default, C++ has sparse data packing
  - Each field is aligned by its size (to prevent line splits)

```
struct _Sparse {
    bool b0;
    uint8_t padding0[3];
    int i;
    bool b1;
    uint8_t padding0[7];
    double d;
};
```

sizeof(Sparse) == **24**


```
struct LessSparse {
    bool b0;
    bool b1;
    int i;
    double d;
};
```

sizeof(LessSparse) == **16**


```
#pragma pack(push)
#pragma pack(1)
struct Dense {
    bool b0;
    int i;
    bool b1;
    double d;
};
#pragma pack(pop)
```

sizeof(Dense) == **14**

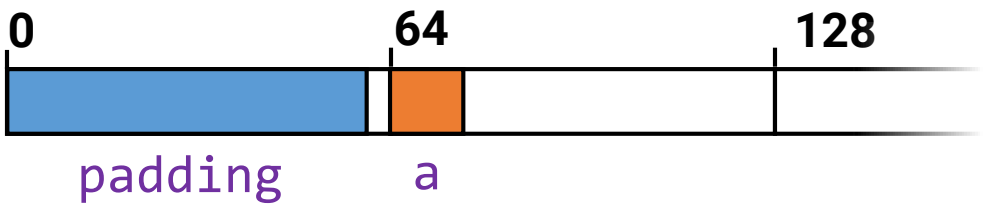
Pitfall: Split Atomic

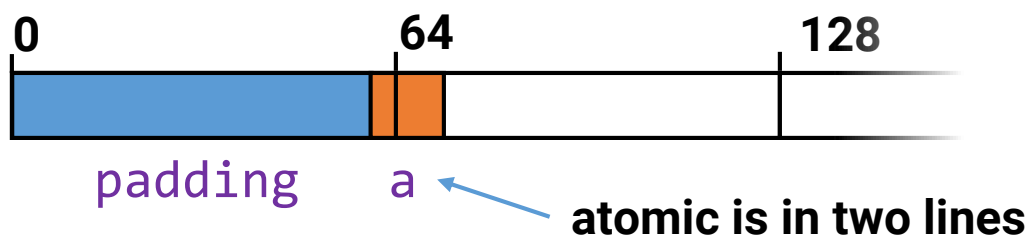
- Dense packing can be dangerous. E.g., lead to split line atomics.

```
struct alignas(64) Atomic
{
    uint8_t padding[63];
    atomic<uint64_t> a;
};
```

vs.

```
#pragma pack(push)
#pragma pack(1)
struct alignas(64) SplitAtomic
{
    uint8_t padding[63];
    atomic<uint64_t> a;
};
#pragma pack(pop)
```





- A split line atomic is **~300x** slower that an usual atomic!

What Next?

- **Not discussed**: virtual memory (TLB), prefetching, coherency, cache conflicts, memory types (UC, WC, etc.), cache inclusion/exclusion, memory ordering, replacement policies, false sharing, thread cross-trashing, …
- **Courses:** MIPT, Princeton, Technion, Berkley CS152/CS61C, UW CSE378, MIT 6.004, CMU 18-447
- **Books:** Computer Organization and Design, Computer Architecture, Modern Processor Design
- **Docs:** Intel Architectures Optimization Reference Manual
- **Tools:** Google Benchmark, godbolt.org / quick-bench.com, Intel VTune Amplifier

Many thanks!

Questions welcome :)

Alexander Titov

 alexander.titov@atitov.com

 alexander-titov-cpu