



C++ COMMUNITY

# 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](mailto:alexander.titov@atitov.com)

[in](https://www.linkedin.com/in/alexander-titov-cpu) [alexander-titov-cpu](https://www.linkedin.com/in/alexander-titov-cpu)



# How CPU Works?

Outside  
World



Outside  
Memory

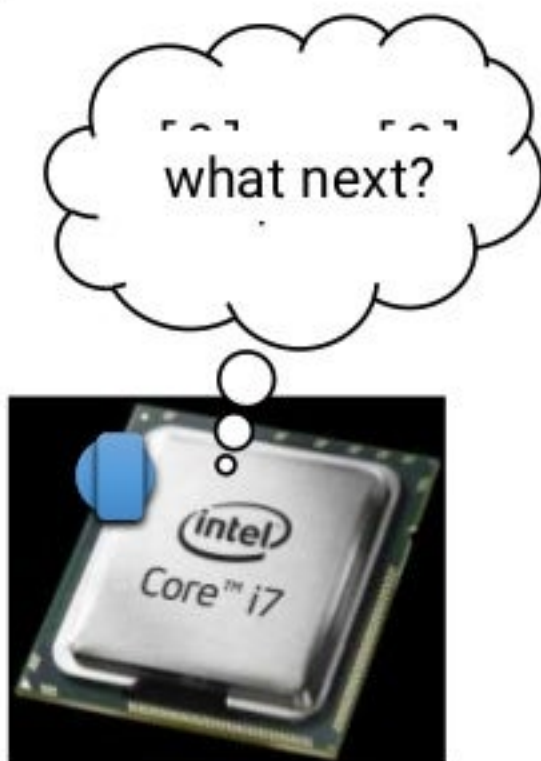


CPU

# How CPU Works?



Outside  
Memory



CPU

## Simplified CPU actions:

1. Read instruction **from MEM**
2. Decode it
3. Read inputs **from MEM**
4. Execute instruction
5. Write result **to MEM**

**Conclusion: CPU works a lot with Outside Memory**

# Is Outside Memory Fast?



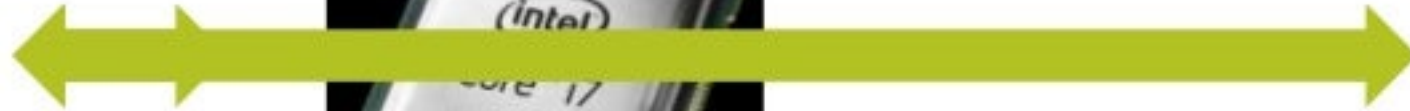
**Outside  
Memory**



**CPU**



**CPU**



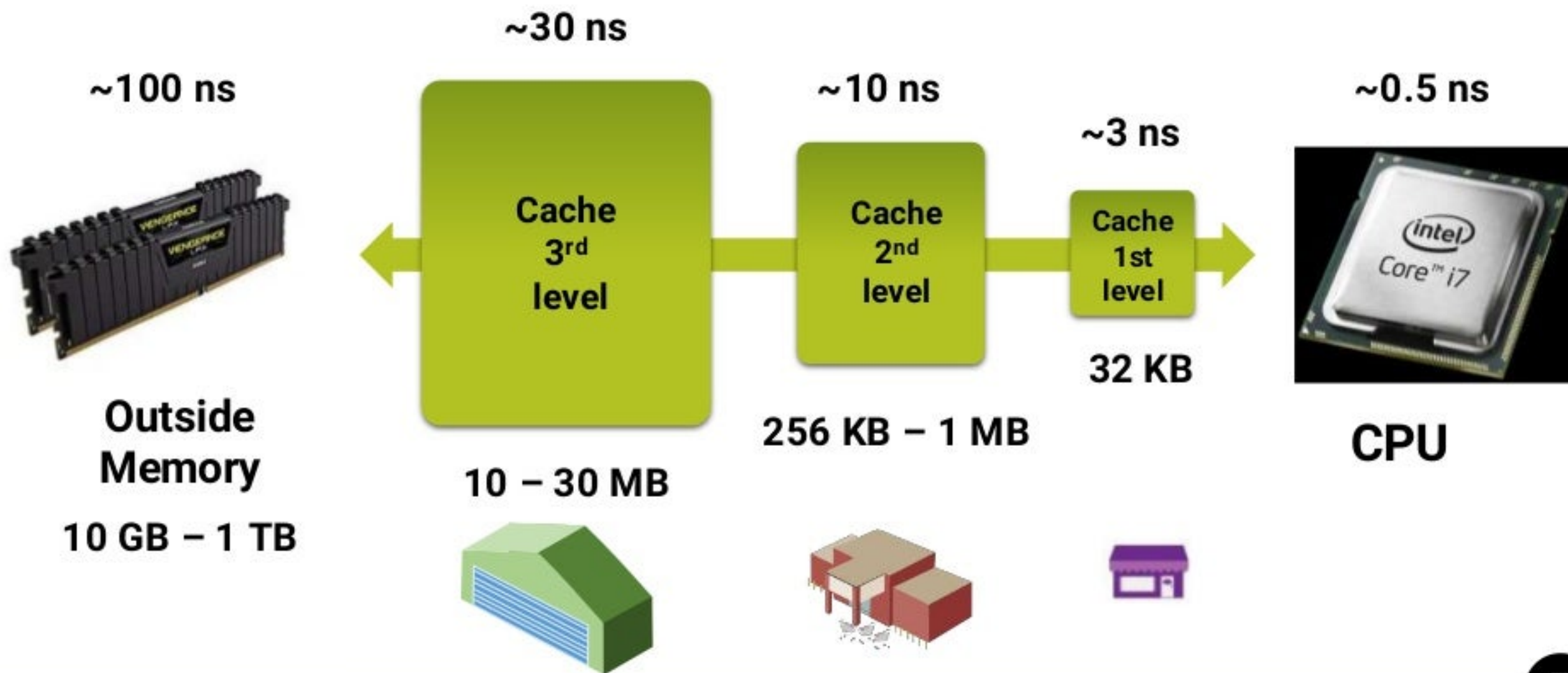


# Is Outside Memory Fast? – No



**Memory is very slow.**  
**CPU would wait 99% of the time for Memory response.**

# Cache Hierarchy



# Cache Hierarchy In Action



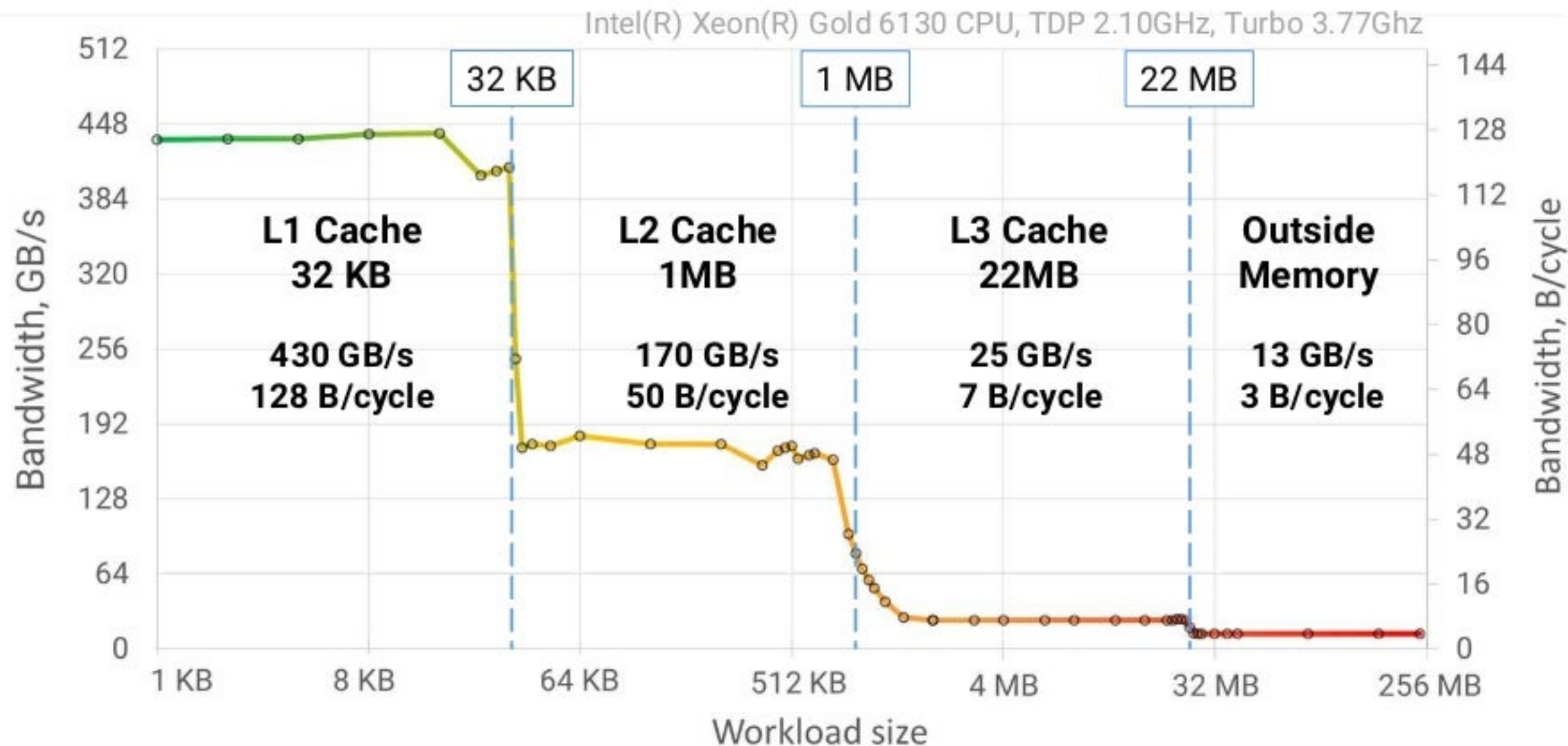


# Cache Hierarchy In Action



**Locality principle:**  
the same data is requested several times in a short period of time

# Experiment: Defining Cache Hierarchy

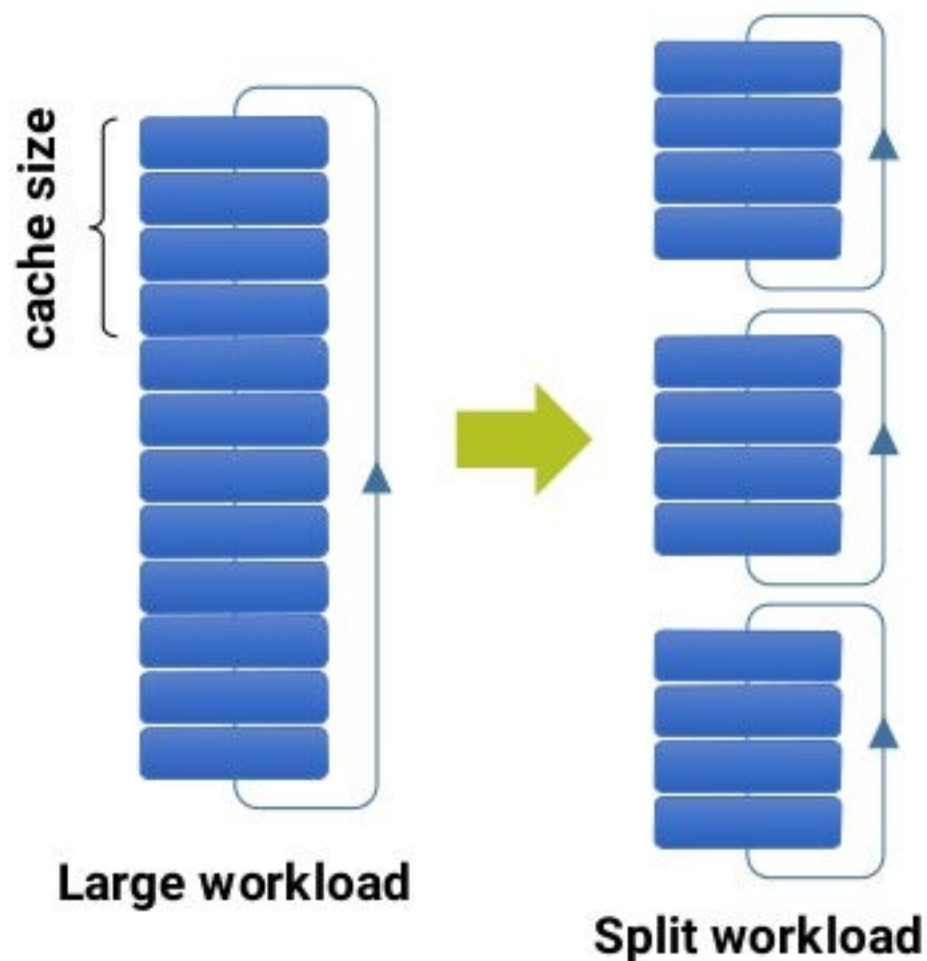


# Performance Optimizations

- Rule #1: **Measure it!**
  - Use tools: google benchmark, perf, Intel VTune Amplifier, etc.
- Rule #2: Do educated decisions
- Rule #3: Do not reinvent bicycle
  - For many tasks there are highly optimized libraries (e.g., Intel MKL)
- Rule #4: Optimize in the following order
  - General algorithm (e.g.,  $O(N \log N)$  vs.  $O(N^2)$ )
  - Memory allocation, copy vs. move, etc.
  - Data structures organization in memory and access patterns
  - Branches, code footprint
  - ...

## Tip: Divide and Conquer

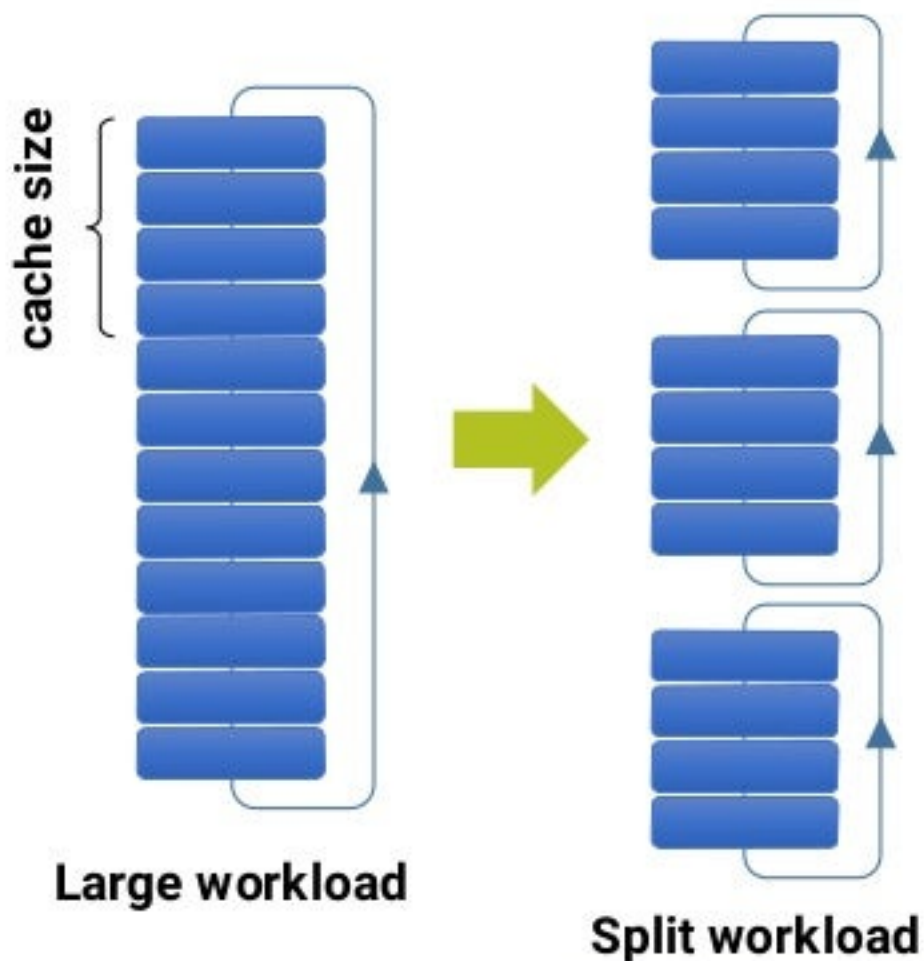
- Split large workload in parts that fit in the cache → improve locality





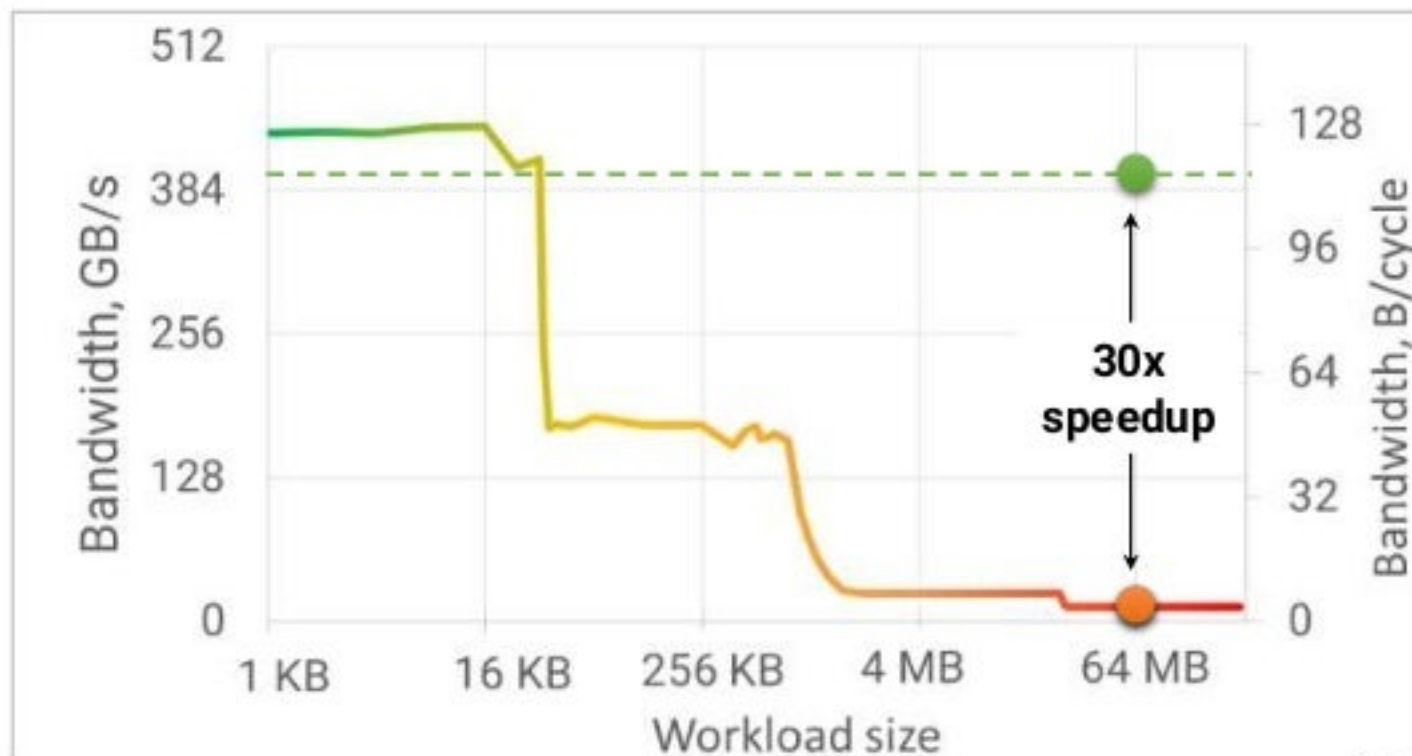
## Tip: Divide and Conquer

- 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)





## Tip: Split Warm and Cold Data

- **All data transfers are performed in aligned chunks of 64 B (line)**
  - Even if 1B is requested by instruction, the whole 64B chunk is read
- If the rest of the line is not used, it occupies space and BW in vain
- Extract fields that are accessed more often (warm) into a separate object

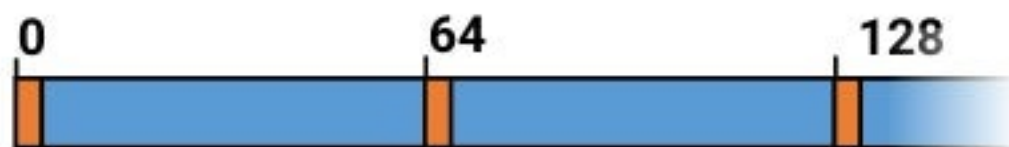
## Tip: Split Warm and Cold Data

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

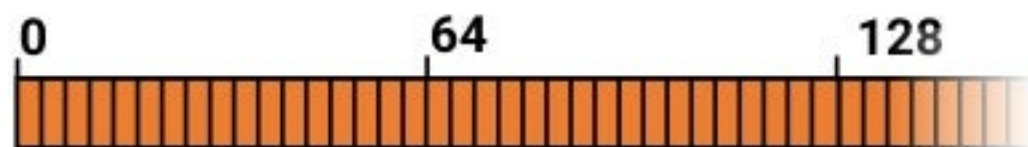
```
struct Mixed {  
    int warm;  
    int cold[15];  
};  
  
Mixed mixed_array[N];
```



```
struct Warm {  
    int warm;  
};  
  
struct Cold {  
    int cold[15];  
};  
  
Warm warm_array[N];  
Cold cold_array[N];
```



6% efficiency



100% efficiency

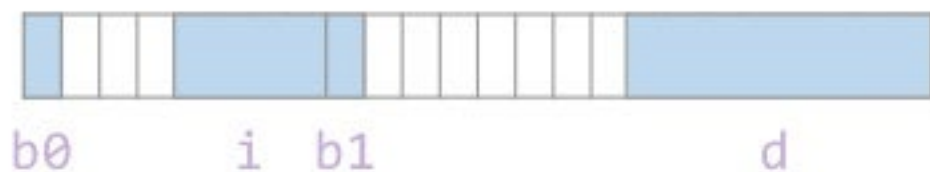
- Decrease occupied cache space and memory bandwidth by **16x times!**

## Tip: Dense Data Packing

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

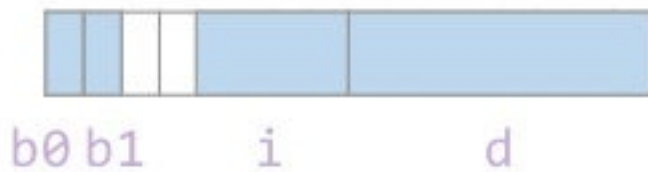
```
struct _Sparse {  
    bool b0;  
    uint8_t padding0[3];  
    int i;  
    bool b1;  
    uint8_t padding0[7];  
    double d;  
};
```

sizeof(Sparse) == **24**



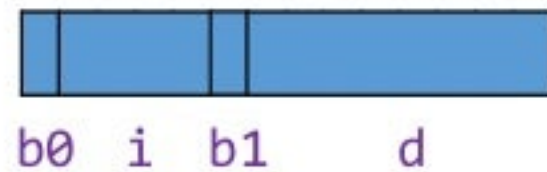
```
struct LessSparse {  
    bool b0;  
    bool b1;  
    int i;  
    double d;  
};
```

sizeof(LessSparse) == **16**



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

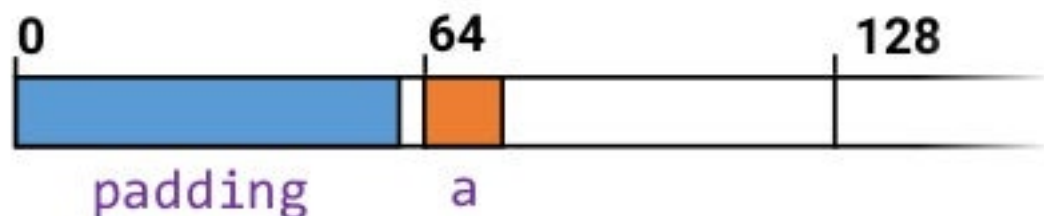
sizeof(Dense) == **14**



# Pitfall: Split Atomic

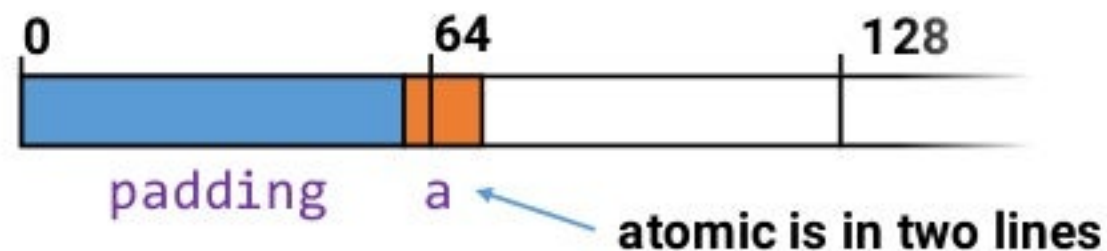
- Dense packing can be dangerous. E.g., lead to split line atomics.

```
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 than a usual atomic!



# Pitfall: False Line Sharing

- Do not place independent data written by multiple threads into one line!

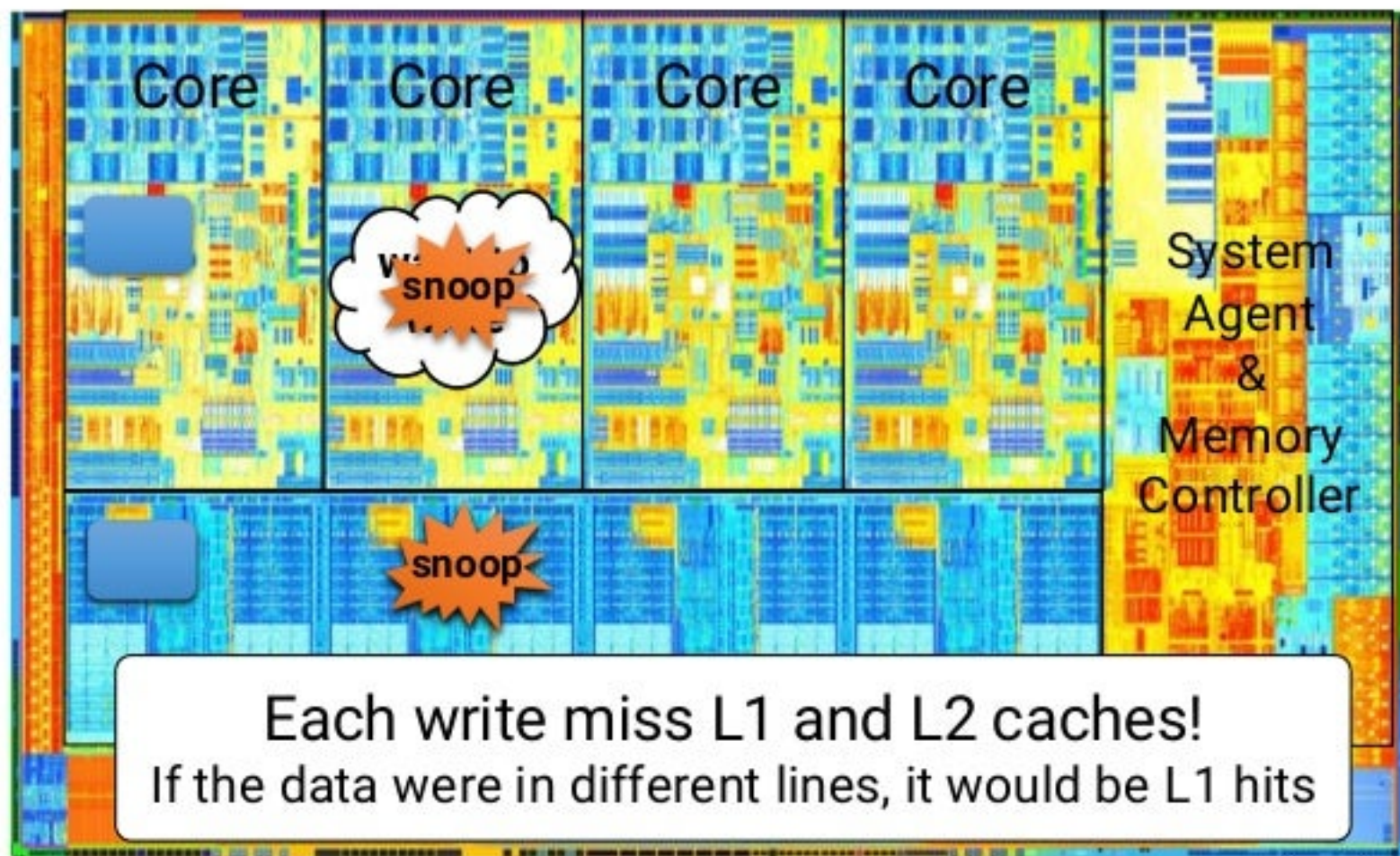
- If data is only read → OK

- Example:

```
// each thread repeatedly writes  
// only its own element  
int thread_results[4];
```



- Core must “own” a whole line to be able to write it
  - The copies in the other Cores are removed





# What Next?

- Docs:
  - [Intel Architectures Optimization Reference Manual](#)
- Courses:
  - [High-Performance Computing and Concurrency](#) by Fedor G. Pikus
- Tools:
  - [Google Benchmark](#)
  - [godbolt.org](#) / [quick-bench.com](#)
  - [Intel VTune Amplifier](#)



C++ COMMUNITY

**Many thanks!**

**Questions welcome :)**

**Alexander Titov**

 [alexander.titov@atitov.com](mailto:alexander.titov@atitov.com)

 [alexander-titov-cpu](https://www.linkedin.com/in/alexander-titov-cpu)