PERFORMANCE-ORIENTED COMPUTING

Optimization (1) - Memory

“Memory Optimization”
as interpreted by stable diffusion

GOALS

- Understand the **performance-relevant aspects** of the **memory hierarchy** of the most common modern architectures
- Develop an **awareness for the cost of latency**
  - And how this affects different applications / phases of an application
- Understand **reuse distance** and how to optimize for it
- Realize the cost of **dynamic memory management**
  - And understand your options for dealing with it

THE MEMORY HIERARCHY

OVERVIEW

- There isn't really ***the*** one memory hierarchy that describes all relevant modern architectures
  - But there are some common factors
- We'll first discuss the **commonalities**, then dive into the most **important differentiating factors**

THE BASICS

Almost universally, you will find:

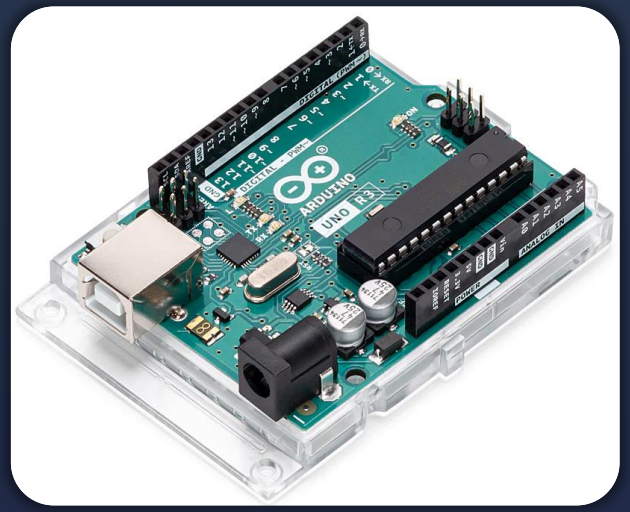
- Small, fast memory
  - Registers (HW and ISA, in that order)
  - L1 data and instruction caches
  - Scratchpad memories
- Something(s) in between
  - L2 / L3 / (L4?) data caches
  - Lower levels attached to cores
  - Higher level attached to memory subsystem
- Large, slow memory
  - DRAM
  - NVME SSDs
  - Anything attached to SATA

DIFFERENCES BETWEEN ARCHITECTURES

- While all relevant architectures share the basic *small=fast* and *large=slow* principle, there are significant differences:
  - How deep is the memory hierarchy?
  - Which layers are shared among cores or even functional units?
  - What level of user control is there?
  - What is the ratio between available memory latency/bandwidth and compute resources?

MEMORY HIERARCHY DEPTH

Generally, memory hierarchy becomes **deeper** with more **architectural sophistication** and higher **performance goals** (especially in CPUs)



Arduino Uno with **Atmega 328P**

No Cache, SRAM memory

Intel **Core i7 5775C**

4x32kB L1 data cache

4x32kB L1 inst. cache

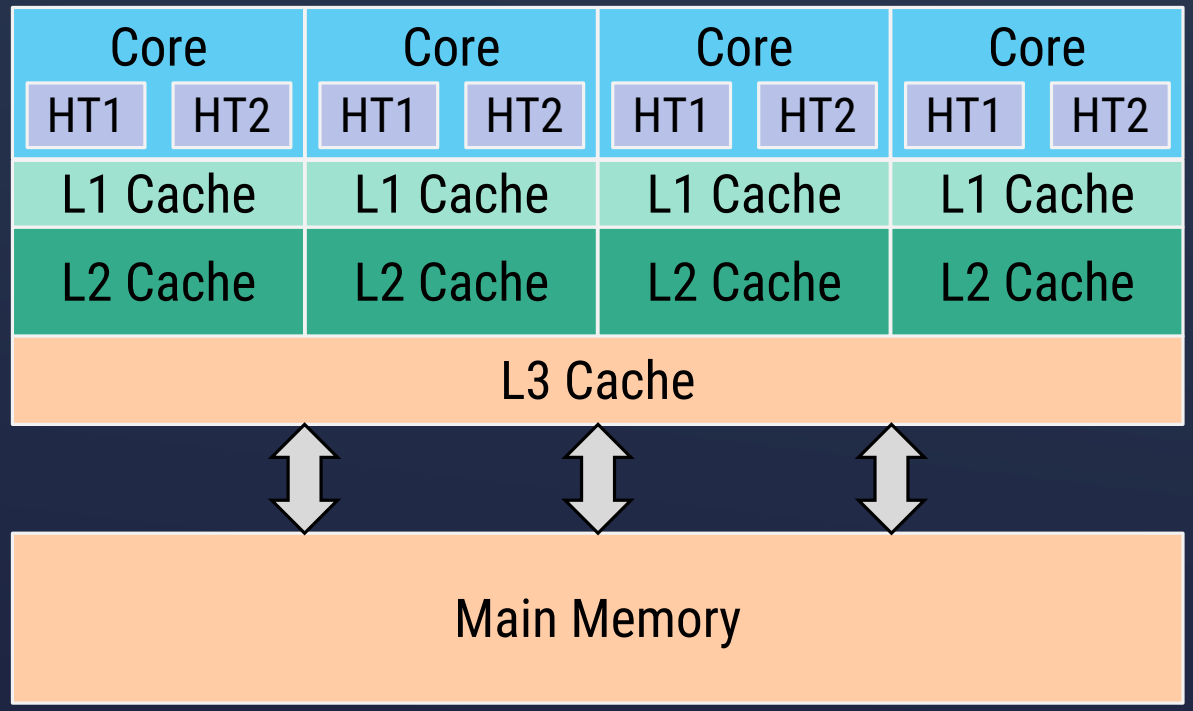
4x256kB L2 cache

6144 kB L3 cache

128 MB L4 cache

SHARED CACHE LAYERS

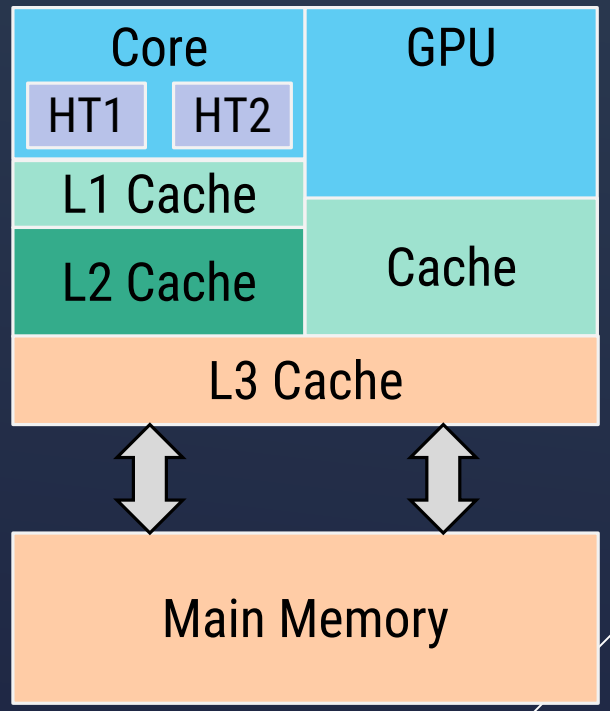
- Cache may be for a single core / functional units, or shared across multiple ones



SHARED CACHE LAYERS

- Cache may be for a single core / functional units, or shared across multiple ones

- On Systems-on-chip / APUs or other integrated architectures, higher cache levels might also be shared between distinct functional units
  - E.g. L3 or L4 cache shared between CPU cores and GPU cores
  - Consequences for performance of data sharing in heterogeneous HW



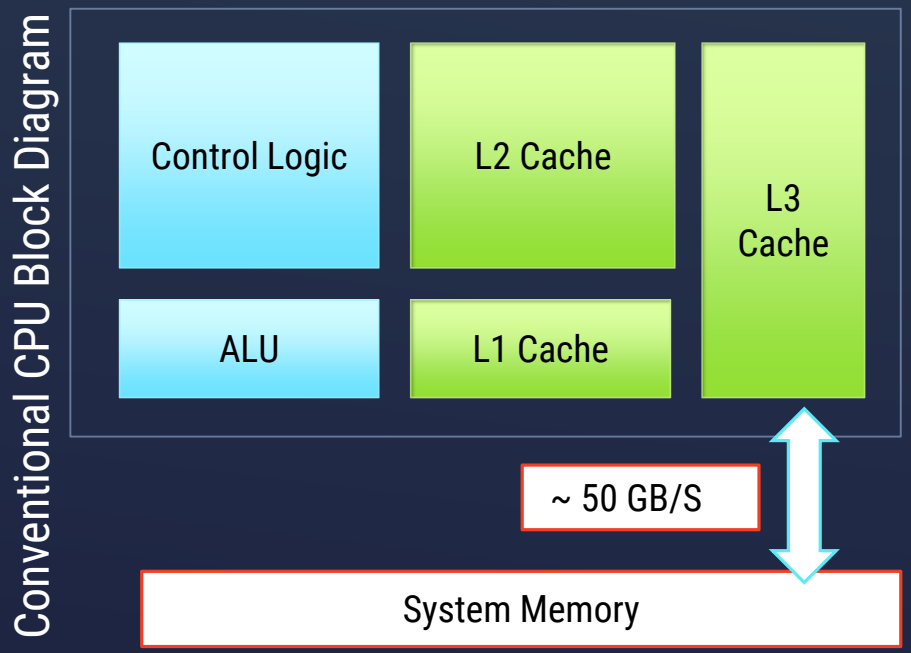
USER CONTROL OVER CACHE

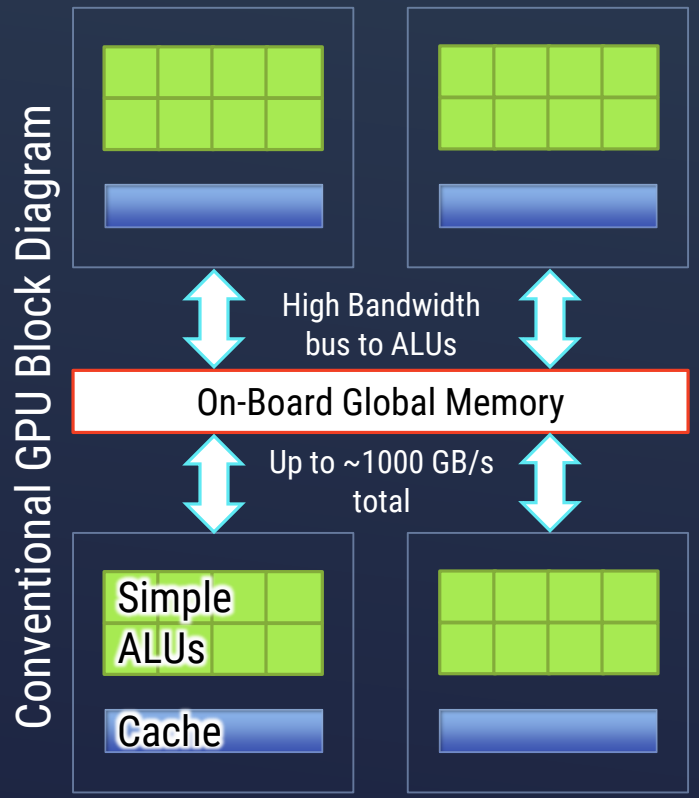
- There are different degrees of user control over small, fast memories across different hardware architectures:
  1. **No manual control**
     - Fully hardware controlled cache
  2. **Partial manual control**
     - E.g. cache, but can manually prefetch and/or disable caching for some parts
  3. **Fully manual control**
     - Manually addressed shared storage on GPUs, or scratchpad memory in architectures like Cell

- **Trade-off** between hardware complexity, speed, and ease of programming!

ALU THROUGHPUT VS MEMORY ACCESS

- Different types of architectures aiming at different applications might have **very different focus** between **compute** and **memory/cache** resources





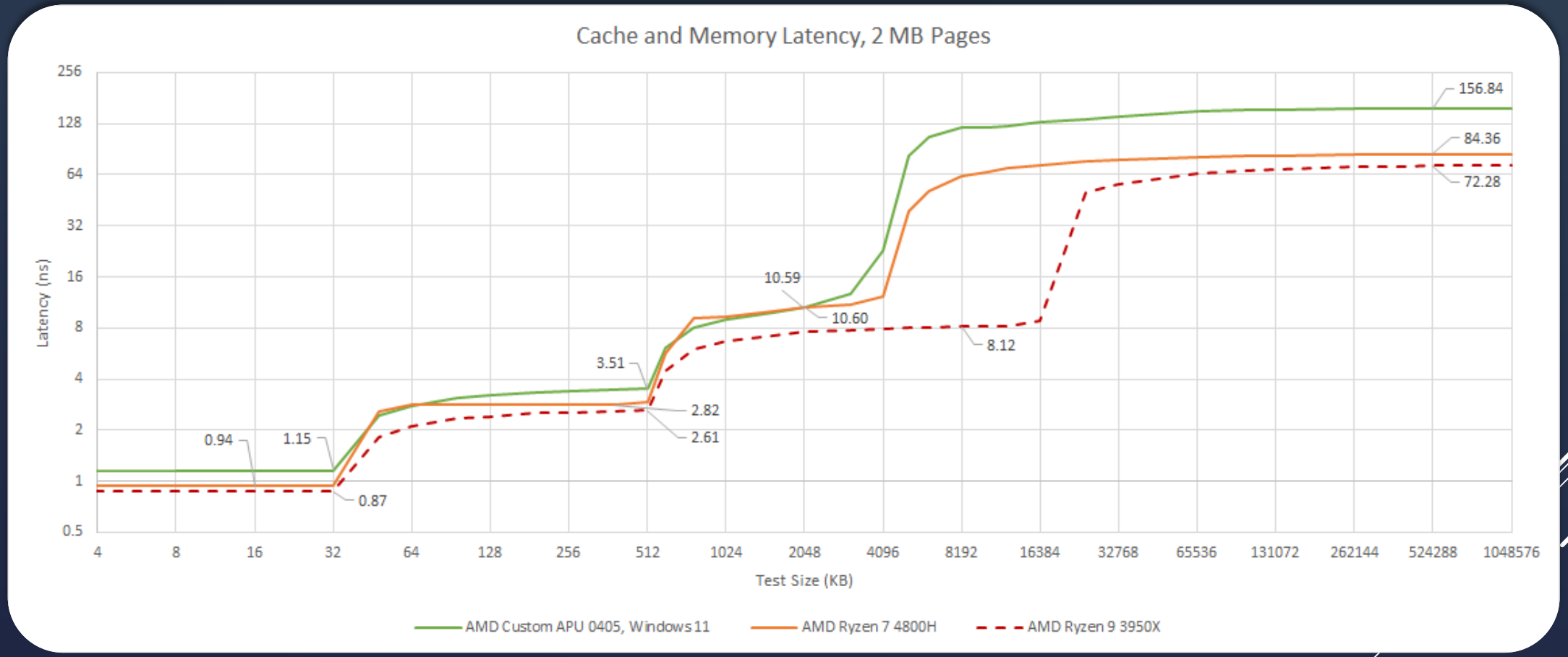
THE COST OF LATENCY

BACKGROUND

- **Why** do we have such deep and diverse memory hierarchies?
  - With all the **complexities** they bring to both hardware design and software performance optimization/engineering
- Because, on modern hardware, the time it takes to *retrieve* data from main memory is ***enormous*** compared to the time it takes to *process* that same amount of data!

  → We **need** effective caching to have any chance of keeping our ALUs fed and actively working on a problem

MAGNITUDE OF LATENCY DIFFERENCES



→ ~ **factor 10+** between L1 and L3, and factor 10 between that and main memory

Source: chipsandcheese.com *"Van Gogh, AMD's Steam Deck APU"*

THE DEVELOPMENT OF LATENCY OVER TIME

- A good resource: https://colin-scott.github.io/personal_website/research/interactive_latency.html
  - Exact numbers aren't really reliable, but it's useful for the general ballpark

- Some clear **trends** to note:
  - We only see significant CPU latency reductions until ~2005, where we get to the ~1ns level
  - From there onwards, the most significant changes are SSD-related

IMPACT OF LATENCY

- ***When*** does latency manifest as a large performance problem on modern CPUs?
  - Whenever the **cache hierarchy and prefetching** HW subsystems **cannot do their job** properly/fully
  - And further execution **depends** on those accesses
  - → *CPU is waiting for memory access before it can continue doing useful work*
- How can we tell?
  - HW counters indicating lots of stalled cycles and/or cache misses

LATENCY-FOCUSED OPTIMIZATION

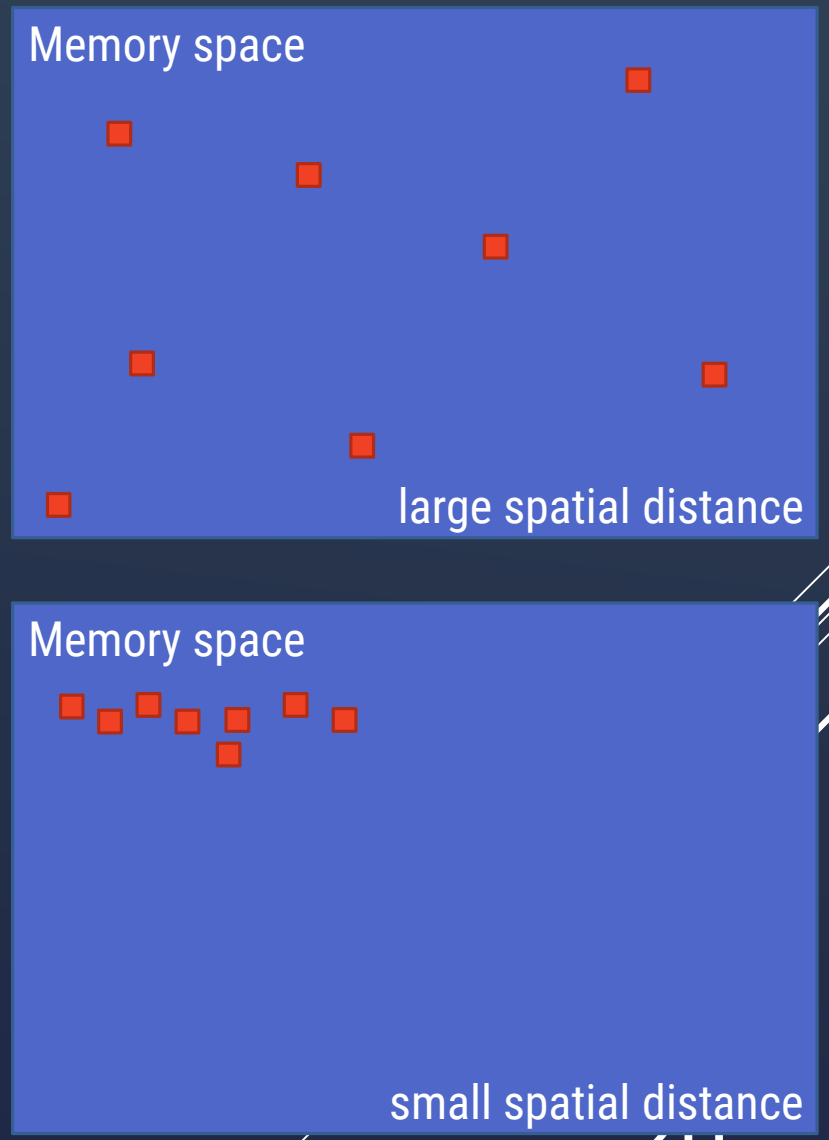
Optimization for latency can take different approaches, depending on the application or algorithm phase:

1. **Latency elimination**
   - Try to **restructure** data/control flow or data structure layout to reduce the number/influence of uncached accesses
   - We'll discuss examples of this regarding the behaviour of some data structures in a later chapter
2. **Latency hiding**
   - Accept the latency of individual accesses, but increase the degree of **parallelism** in order to give the functional units something to do while the memory access is happening
   - Common approach on GPUs, but also applicable to some extent on CPUs – even within a single instruction stream due to OOE!

REUSE DISTANCE

INTRODUCTION

- In order to understand how much caching (automatic or [semi-]manual) can potentially improve the performance of a given phase/algorithm of our application, we consider **reuse distances**
  - **Spatial reuse distance:**
    - How much distance there is **spatially** in the address space between memory locations accessed sequentially
  - **Temporal reuse distance:**
    - How much time (e.g. cycles) passes before the same memory addresses are accessed again



WHY REDUCE REUSE DISTANCE?

Reducing spatial and/or temporal reuse distance increases the efficiency of modern hardware:

- All levels of **caches** are more effective with low distances
- Improving spatial distance, in particular when also making accesses more **regular**, might also increase the effectiveness of **prefetching**

→ Can improve not only latency, but also **bandwidth**!

REUSE DISTANCE OPTIMIZATION

- Some **data structures** are **inherently** better at keeping reuse distances low than others – we will discuss this more in the data structure chapter
- However, in many cases we can also optimize the **order** of operations/accesses to reduce reuse distance and improve cache effectiveness
- One relatively simple example of this which we will discuss is dense linear algebra, but it's also applicable in **any other context** where we have a **choice on ordering**

EXAMPLE

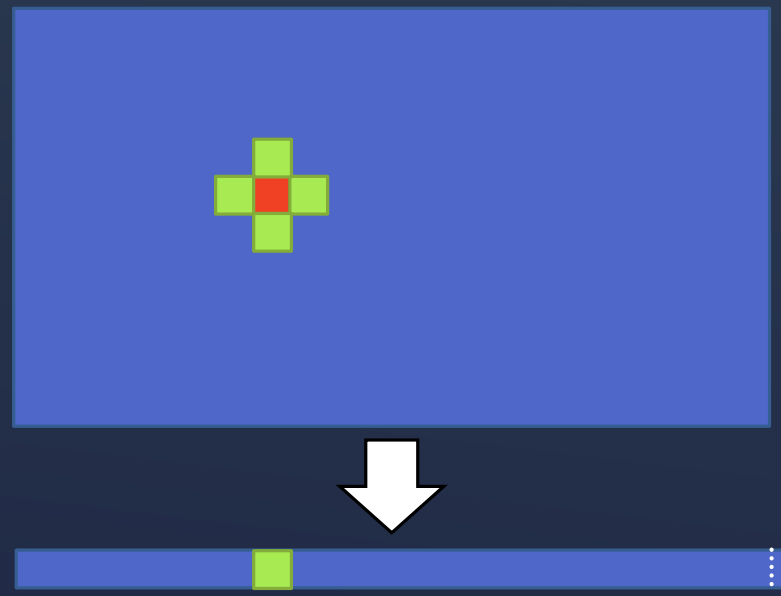
- Imagine a basic loop like the following:

```
for(int x=0; x<W; ++x) {
  for(int y=0; y<H; ++y) {
    out[x][y] = 0.2 * ( in[x][y]
              + in[x+1][y] + in[x-1][y]
              + in[x][y+1] + in[x][y-1] );
  }
}
```

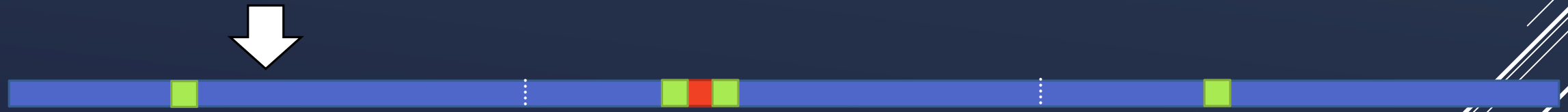
- What is our reuse distance for each of these accesses?
- How can we optimize it?

EXAMPLE

Semantically



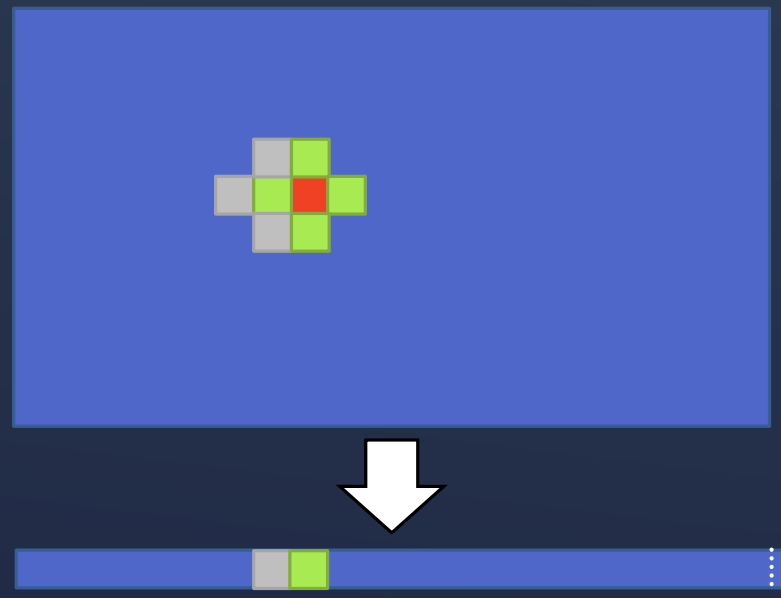
- Vertical offsets have large in-memory spatial distance



Linearized in memory

EXAMPLE

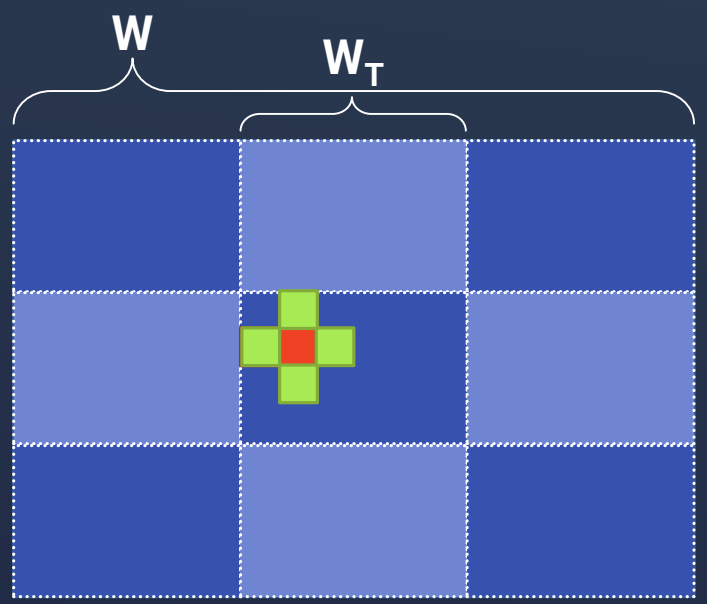
Semantically



- Vertical offsets have large in-memory **spatial** distance
- **Temporal** reuse is not bad for center line at $2/3$, but ***only after W steps*** for upper/lower elements

Linearized in memory

TILING



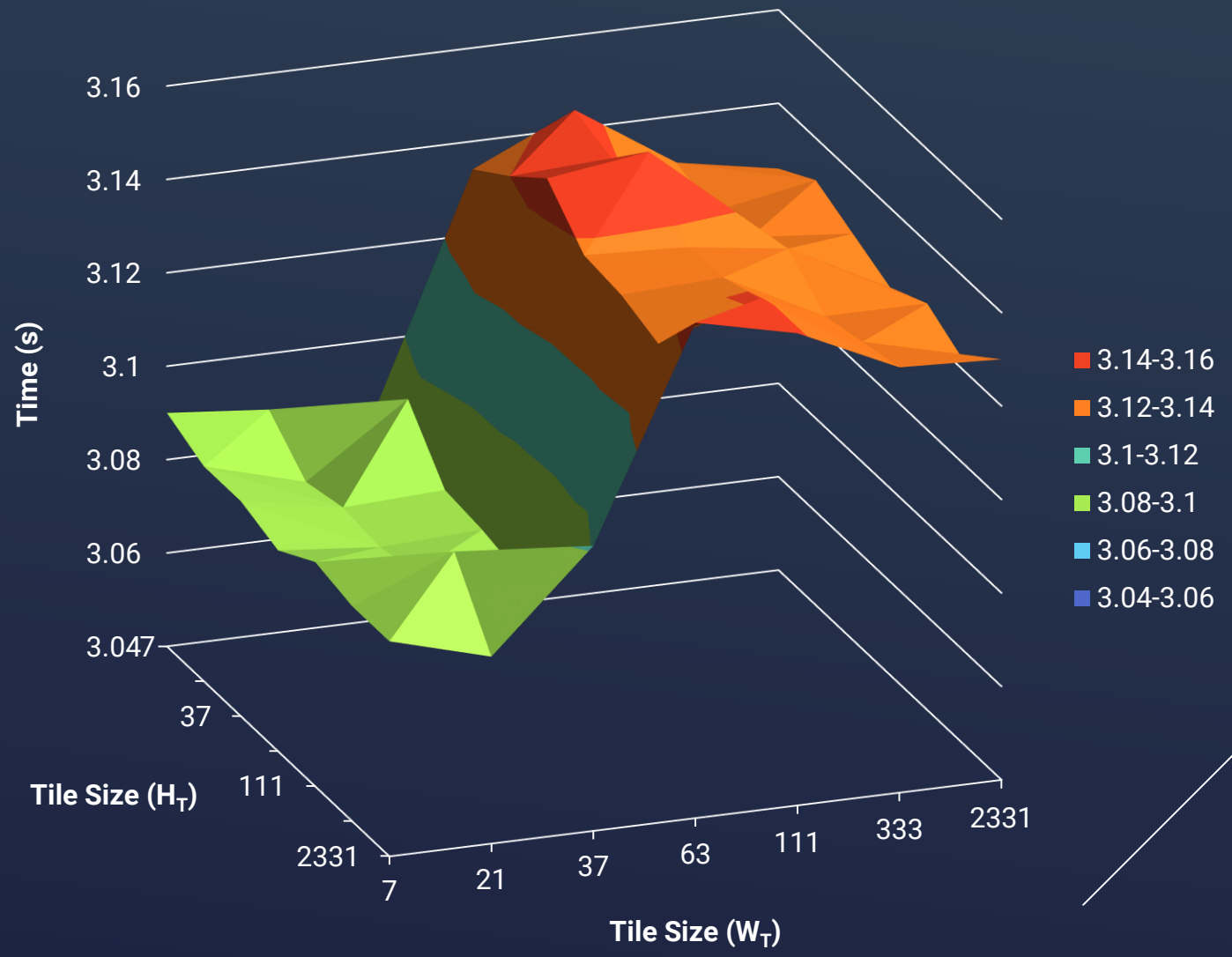
- By subdividing the loop iteration space, we can **reduce the temporal reuse distance** of the off-row elements to the width of the tile $\mathbf{W_T}$
- This general principle is called (loop) **tiling**

```
for(int tx=0; tx<nTX; ++tx) {
  for(int ty=0; ty<nTY; ++ty) {
    for(int x=tx*WT; x<tx*WT+WT; ++x) {
      for(int y=ty*HT; y<ty*HT+HT; ++y) {
        out[x][y] = 0.2 * ( in[x][y]
           + in[x+1][y] + in[x-1][y]
           + in[x][y+1] + in[x][y-1] );
      }
    }
  }
}
```

(simplified, assumes perfect tiling)

TILING EVALUATION

- On LCC2



TILING

- Note that while a basic 5 point stencil is well suited to **explaining the idea** of tiling, it's not a good example to measure anything meaningful on modern hardware
  - The access pattern is very regular, so automatic CPU **prefetching** will work very well
  - There just isn't a lot of data reuse in general

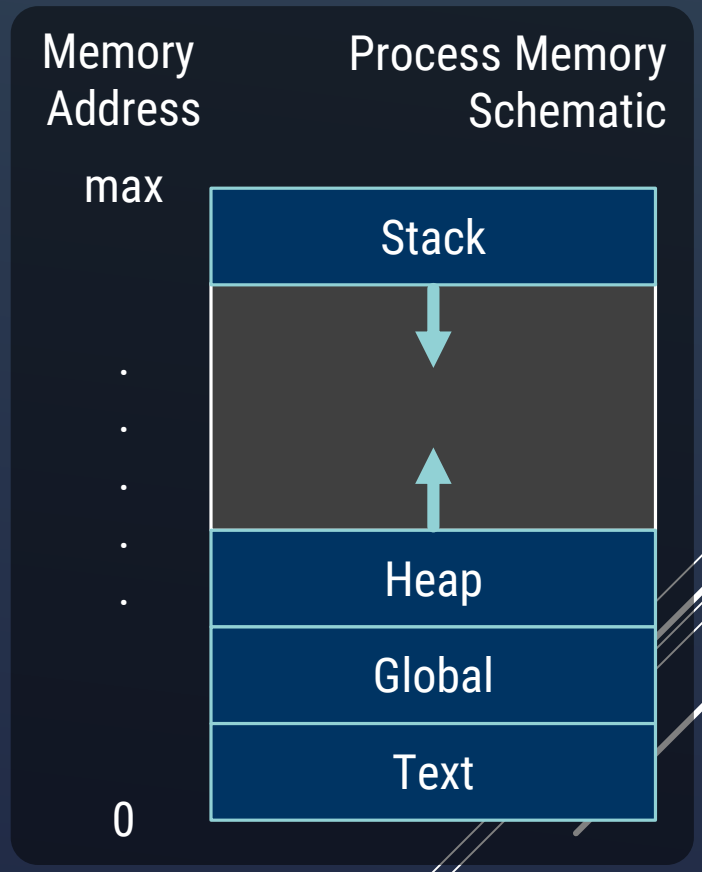
→ The evaluation result on the previous page is based on slightly more complicated code

DYNAMIC MEMORY MANAGEMENT

BASIC REMINDER

- We generally use **3 types** of memory (in the languages we are primarily interested in in this course):
  - **Global** memory (i.e. globals, statics)
    - Usually not viable for our performance optimization
  - **Stack** memory (i.e. local variables)
    - Fast but limited capacity
  - **Heap** memory (i.e. what `malloc` gives us)

→ Of these, we use the **heap** and **stack** for dynamic data



STACK MEMORY

- The stack is about as fast as dynamic memory use gets
  - Reserving / freeing space just amounts to a **single local integer operation**
  - Furthermore, since we are always using it, stack memory lines are extremely likely to be in cache
- However, there are significant restrictions
  - The amount of stack space we have is limited
  - Data lifetimes have to follow a LIFO/FILO (stack) pattern

➔ **Always** prefer to use the Stack *when you can*!

HEAP MEMORY

- The vast majority of the larger data structures we are going to work on are going to be stored in **heap memory**
- Heap memory is managed (based on OS pages) by a **memory allocator**
- The memory allocator itself manages some sophisticated data structures
  - → Allocating heap memory can be **expensive** (in terms of performance)

MEMORY ALLOCATORS

- Since memory allocation can be a **significant performance factor**, a lot of work has gone into memory allocation strategies
- Two main **categories** of allocators:
  1. **General** memory allocators
     - Can deal with **arbitrary patterns** of allocation
     - Used e.g. to implement `malloc`
  2. **Specialized** memory allocators
     - Optimized for **specific patterns** at the cost of generality
     - Can achieve higher peak performance

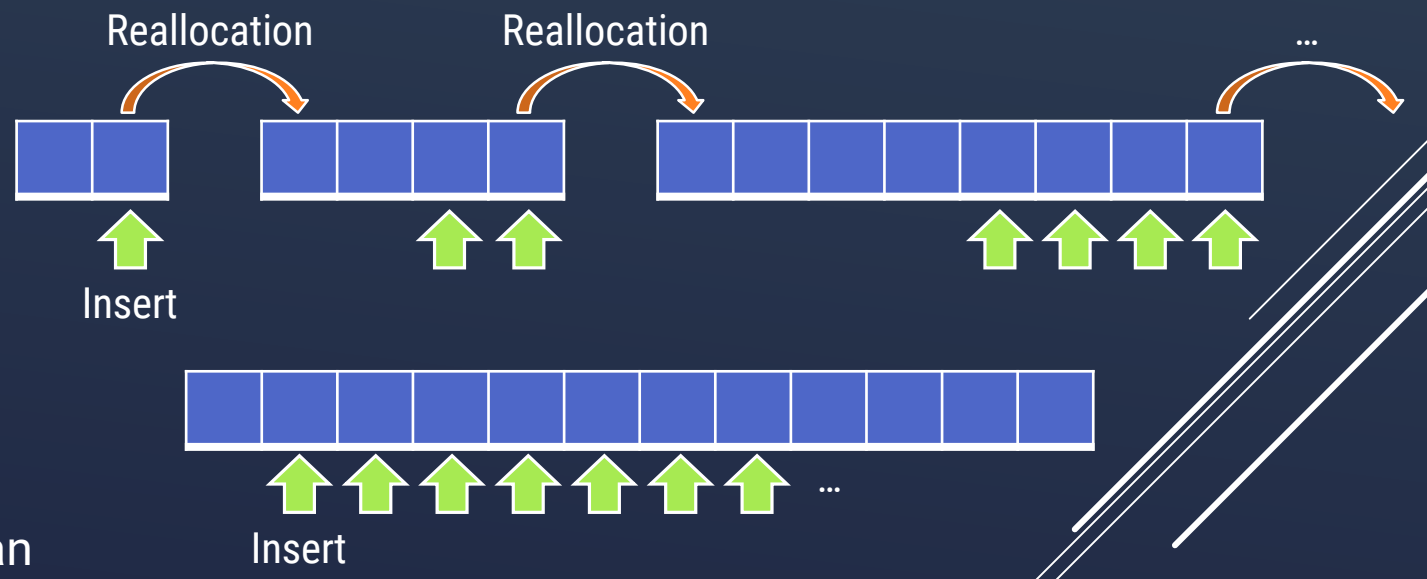
HOW TO OPTIMIZE MEMORY MANAGEMENT

If we determine that memory allocation / management is a significant performance factor in our application, we have a few potential strategies:

- **Reduce** the number of **individual allocations** / deallocations required by our algorithm or data structures
- Use a **specialized memory allocator** for *specific* performance-intensive parts or data structures in our program
- Select a **different general memory allocator**, or change its configuration

REDUCING ALLOCATIONS

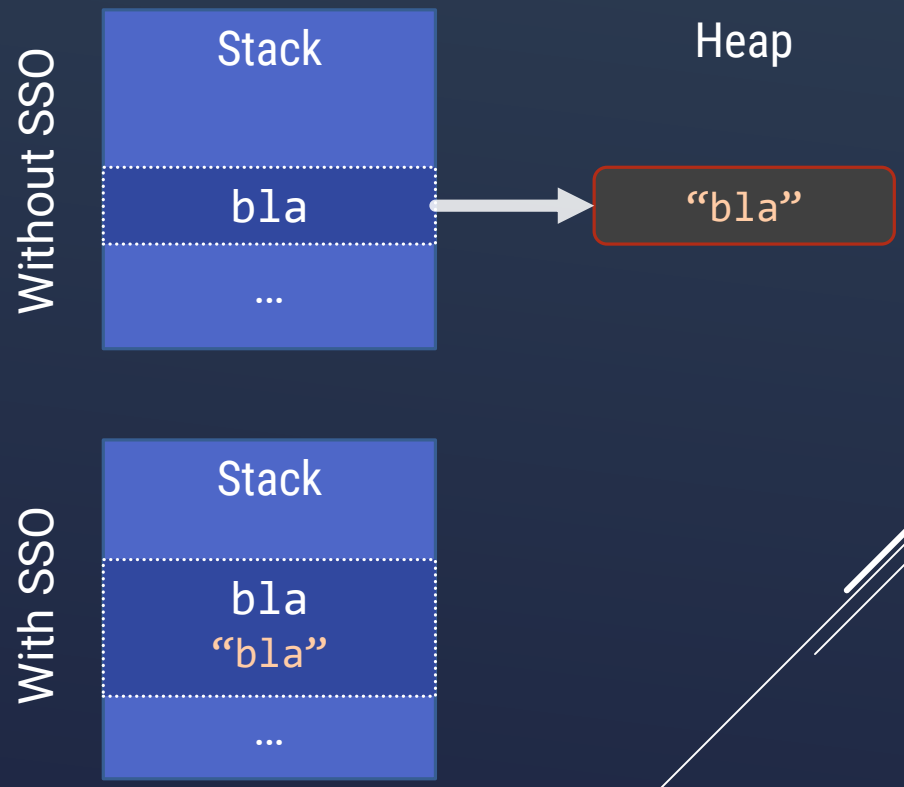
- This is highly specific to individual applications / algorithms / data structures
- However, one common and highly effective strategy is **using a-priori knowledge about the size** of data structures
  - I.e. if we know that we will put 1429 elements in a data structure, reserve space for those beforehand rather than letting it (re-)allocate on insertion



REDUCING ALLOCATIONS

```
{
  string bla = “bla”;
  // …
}
```

- Another option is to use (a more general form of) the **small string optimization (SSO)**
  - In short, if we have an additional memory indirection in our data structure, *reserve some space for small elements from the beginning*, and only do a second allocation when that space is/becomes insufficient
  - Can even use the space you would otherwise use for the pointer!
  - Also applicable to similar situations (e.g. small lists, graph connections, …)



SPECIALIZED ALLOCATORS

- General memory allocators have to deal with **arbitrary requests**
  - In terms of size, frequency, lifetime, interleaving, threads…
  - This is convenient, but **limits the absolute performance** potential of even the most optimized general allocators
- If we have more restrictive patterns and/or data structures, we might benefit from using a **more specialized allocator**

FIXED-SIZE BLOCKS

A common specialization is **for fixed-size objects** (e.g. of a specific type)

- Knowing that the size of all allocated objects is the same greatly **simplifies the implementation** of the allocation strategy
- Also might **reduce storage requirements**, especially for small objects (no need to store extra meta-information e.g. about size)
- It also completely **eliminates** (*internal*) **fragmentation**
  - *Note:* we are specifically talking about using a **specialized** allocator here (fixed-size block allocation for general memory is also efficient in time, but can cause lots of internal fragmentation)

ARENA ALLOCATION

Two possible scenarios:

1. We know **a priori** that we will need a **specific amount** of memory
2. We know that we are starting some task now, and that **all** allocations performed **within that task** can be **freed at the same time**

Of course, **both** is ideal.

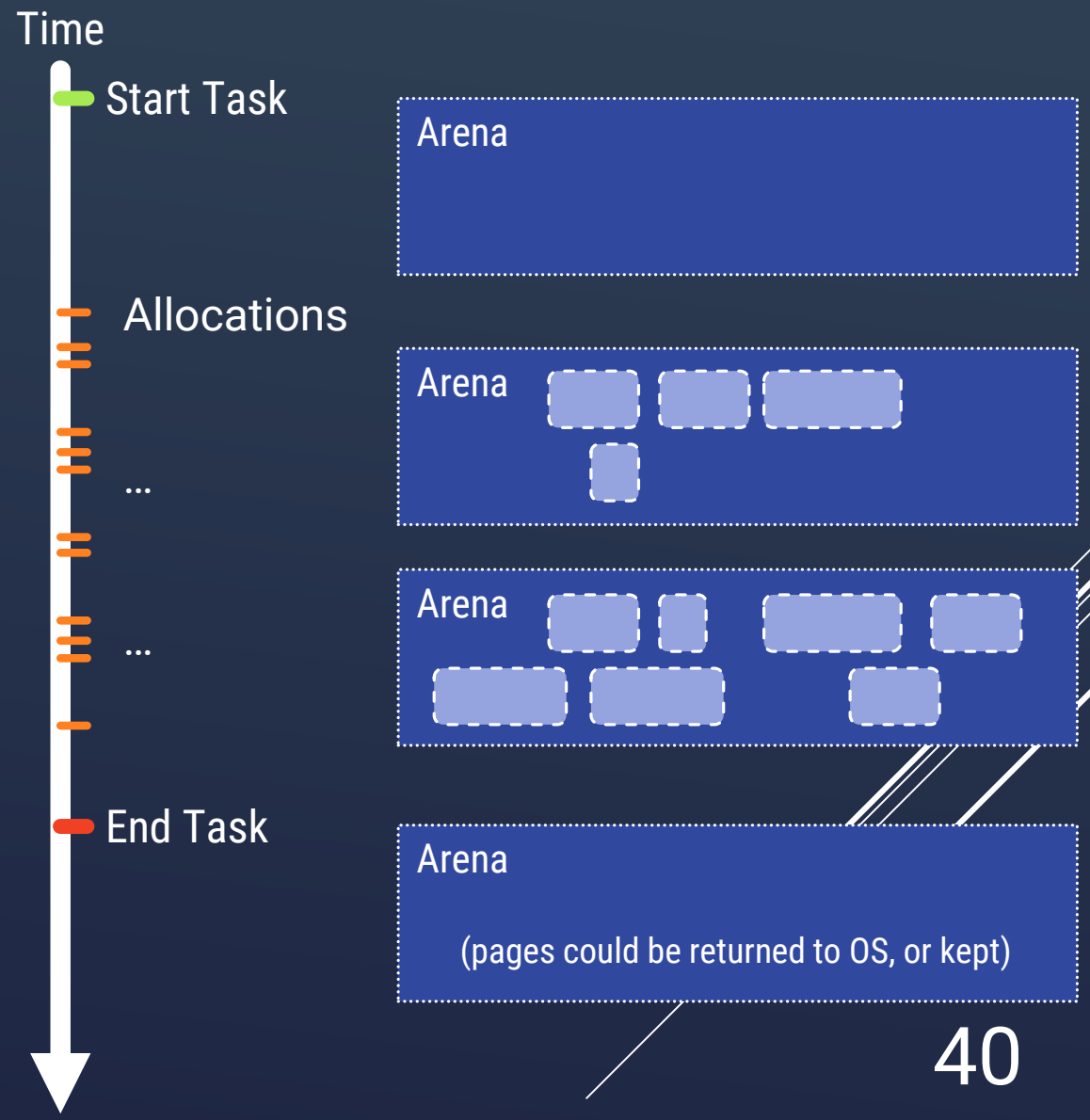
ARENA ALLOCATION

Basic principle:

1. Allocate **a large chunk** (“arena”) of memory **at the start**
2. Throw **all of it away** in one fell swoop when the current task is complete

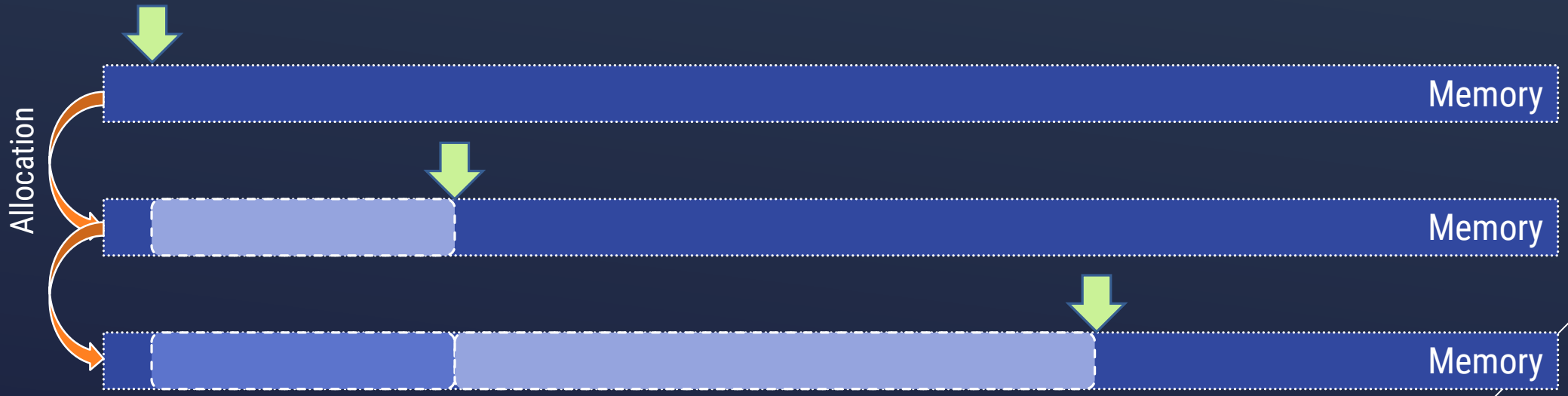
→ We don’t need any **metainformation** to free memory

→ We don’t need to **allocate OS pages** (latency!) outside the initial allocation



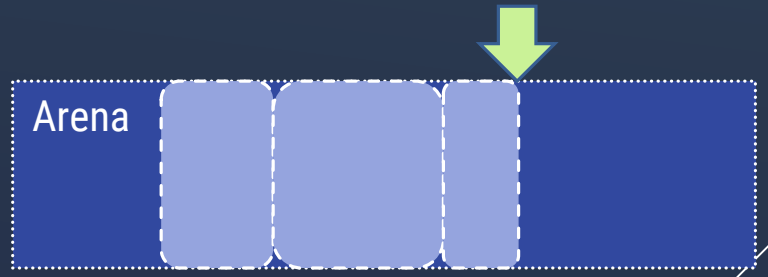
BUMP ALLOCATION

- Perhaps the **simplest** form of memory allocation
  - Allocations simply **increase ("bump") a pointer**
  - Potential to be **as cheap as stack allocation**!
- **Disadvantages**:
  - When to re-use freed memory?
  - → More complex strategies defeat the purpose



BUMP ALLOCATION IN AN ARENA

- By using a bump allocator to perform allocations *in an arena*, we get the best of both worlds:
  - **Cheapest possible allocation** (pointer bump)
  - **Cheapest possible freeing** (pointer reset)



- Of course, this requires that we
  1. Have functional units/tasks **suitable for arena allocation**
  2. Do **not require memory re-use** of allocations **within** such a task

- Examples:
  - Short-lived allocations **inside a single frame** in a game
  - Allocations associated with processing **one request** in a web server

SPECIALIZED ALLOCATORS – SUMMARY

- If we have **specific patterns** in our program that fit one of the **simple** specialized allocators, there might be significant performance gains
- More complex specialized allocators are usually **not worth it**
  - Also hard to write and maintain, and potentially brittle

- Another option if you don't have suitable allocation patterns, but know that your performance is affected by allocator performance:

➔ Use a **better general allocator** and/or **tune** its configuration

GENERAL MEMORY ALLOCATORS

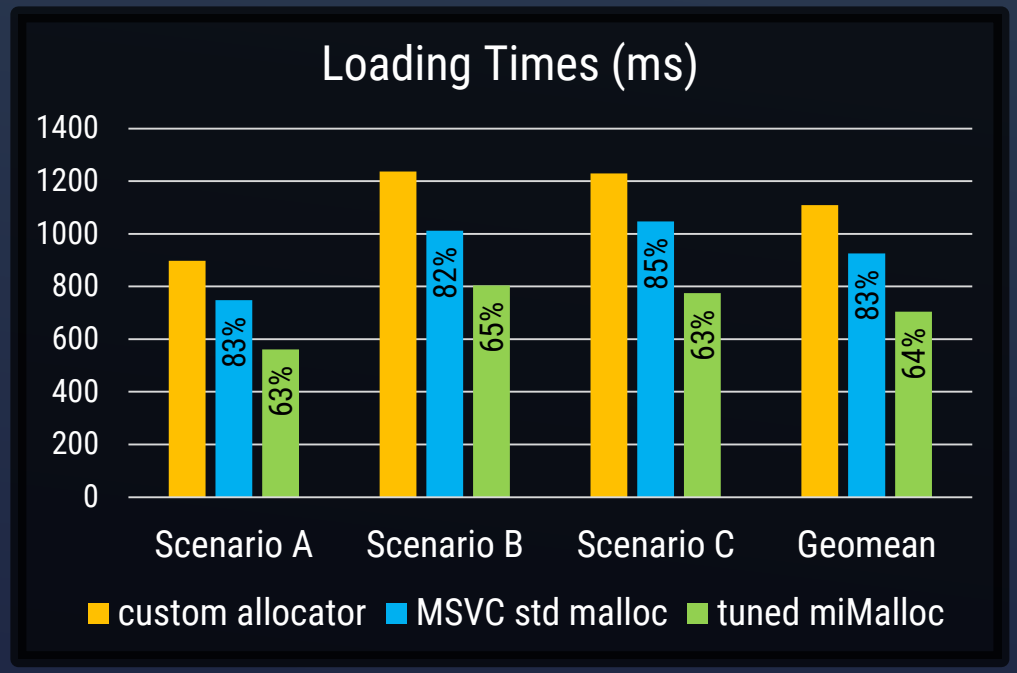
Need to deal with (among other things)

- **Different sized allocations**
  - Should be fast for small ones, but not too much overhead on large ones
- **Different threads/cores**
  - Allocations that stay on one core should be fast, but also frequent migration (allocation on thread A, use/free on B) shouldn't be an issue
- Tradeoffs between **fragmentation** and **overhead**
- **Hardware**-specific concerns (i.e. cache, …)

➔ ***Difficult*.** Luckily other people have spent time on this. A lot!

GENERAL MEMORY ALLOCATORS - STRATEGIES -

- To optimize your program, it's basically never a good choice to write your own **general** memory allocator
- Allow me to demonstrate:
  - Data is from a commercial game
  - Time to load 3 different scenarios
  - Only change is the memory allocator!



GENERAL MEMORY ALLOCATORS - STRATEGIES -

- To optimize your program, it's basically never a good choice to write your own **general** memory allocator
  - However, it's still useful to know the basics of what they do

Two primary strategies:

- **Per-thread management of small allocations**
  - → Prevents synchronization overhead for these common allocations
- Multiple **fixed-block freelist** allocators for different size categories
  - → Trades memory efficiency for speed (viable for small sizes)

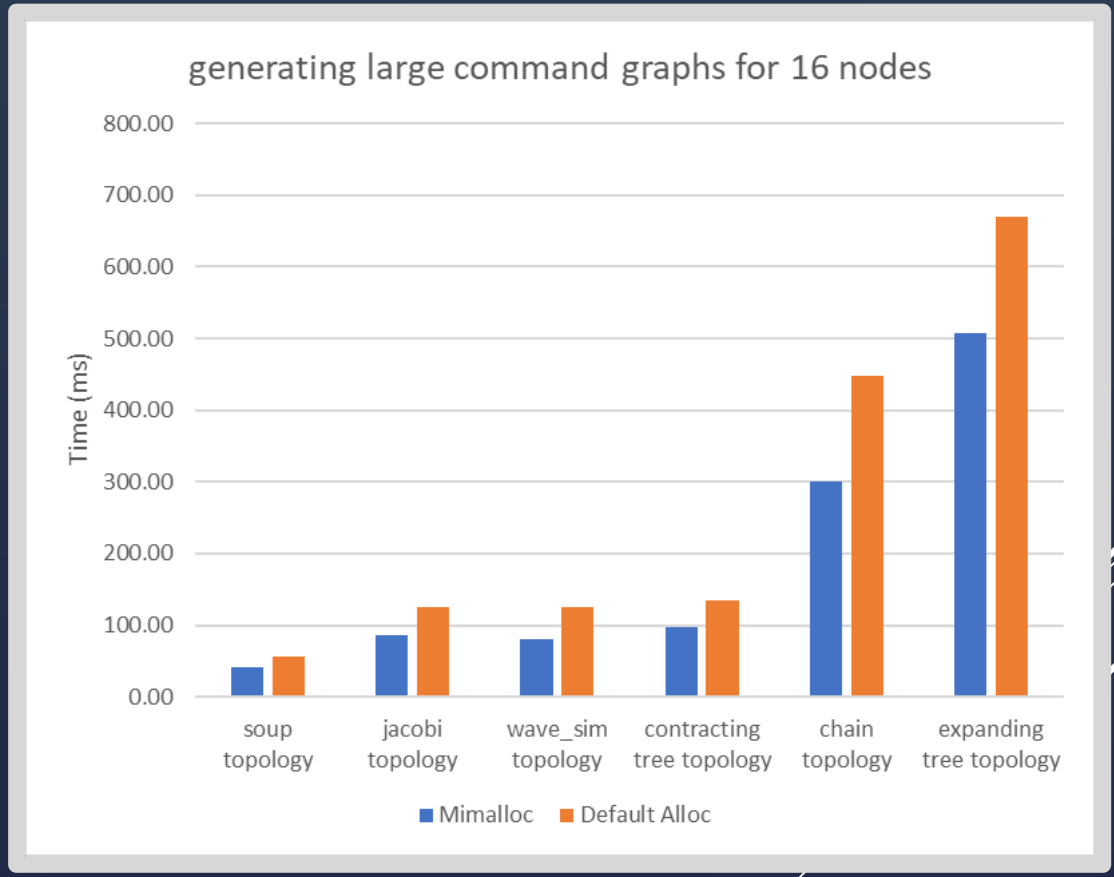
GENERAL MEMORY ALLOCATOR OPTIONS

Here are some options I've worked with before

- Google **tcmalloc** (https://github.com/google/tcmalloc)
  - Well established; relatively large code base, no longer state of the art
- Mattias Jansson's **rpmalloc** (https://github.com/mjansson/rpmalloc)
  - Compact, fast, easy to integrate; significant memory overhead
- Microsoft **mimalloc** (https://github.com/microsoft/mimalloc)
  - Also very fast, relatively compact and easy to integrate

MIMALLOC INTEGRATION

- There are a few options to integrate mimalloc, but all of them are **relatively simple**
  - No large changes in code needed, **negligible impact on maintenance**!
- Recent example: https://github.com/celerity/celerity-runtime/pull/170



CONCLUSION

SUMMARY

- **The Memory Hierarchy**
  - Caches & depth
  - User control
  - Differences between HW
- **The cost of latency**
  - Magnitude
  - Latency mitigation strategies
- **Reuse Distance**
  - Temporal / spatial
  - Optimization approaches
  - Tiling
- **Dynamic Memory Management**
  - Stack vs. Heap
  - Reducing allocations
    - Pre-sizing; SSO
  - Specialized allocators
    - Fixed block; Arena; Bump
  - General allocators

QUESTIONS ?