Fast Matrix Transpose

SIG Math Fall 2025

Real life situation

- This is a real life problem that I worked on at my job last year. So for anyone who is not fully convinced that math can help with programming, this should be a good example!
- The project I was working on essentially did analysis on video input. To do this, the computer would read input from a camera, then do some analysis on the retrieved frames.

Problem

- The camera gives us a frame as soon as it comes in, so the generated output is an array of images (which are 2d arrays), i.e. arr[t][y][x].
- However, OpenCV would like each pixel to have its data across time up front, so it is a 2d array of pixel values across time (which are 1d arrays), i.e. arr[x][y][t].
- The resulting problem is a 3D matrix transpose -- given an array with indices t, y, x, create a new array that has the same data but with indices x, y, t.

Other details

- There are some other constraints to the problem too:
- Width refers to x, height refers to y, and length refers to t.
- x, y, t can all be different. Suppose you have 128 frames of a 1280x720 picture, then x = 1280, y = 720, t = 128.
- In general, the width and height will be around 1000 or so, while the length could be in the hundreds.
- All the values we're working with are 1-byte integers. This is due to how the camera and OpenCV work but also makes things much, much simpler.

Initial thoughts

- On the surface, this seems like a 3D matrix transpose problem, which is far less common than the 2D matrix transpose that is typically analyzed.
- However, look at the index swap: [x, y, t] goes to [t, y, x]. The y remains in the same place. Thus, all elements with y value 15 are sent to another element of y value 15.
- Thus we're essentially doing a 2D matrix transpose y times, where the transpose sends [x, t] to [t, x].
- As a result, we'll be studying how to do a 2D matrix transpose fast. Once that's decided it's easy to scale it up to our 3D usecase.

The naive approach

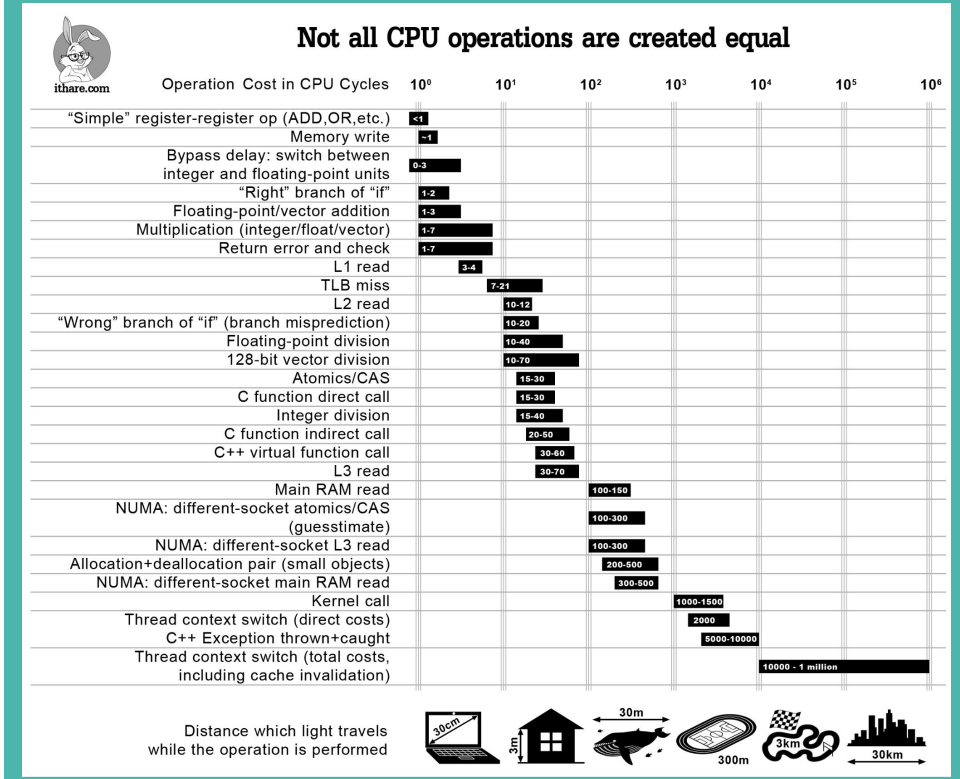
- Suppose you want to transpose a matrix the simple way. You might write:

```
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        output[x][y] = input[y][x];
    }
}
```

- This looks very, very simple -- there's not much you can remove or change to make it faster, right?

My favorite image

- I love this image so much because having an understanding of it already helps you so much in writing performant code.
- Notice the difference between cache memory reads and main memory reads. **Main RAM reads are very, very slow** compared to cached memory reads, so we should try to avoid them as much as possible.



CPU cache analogy

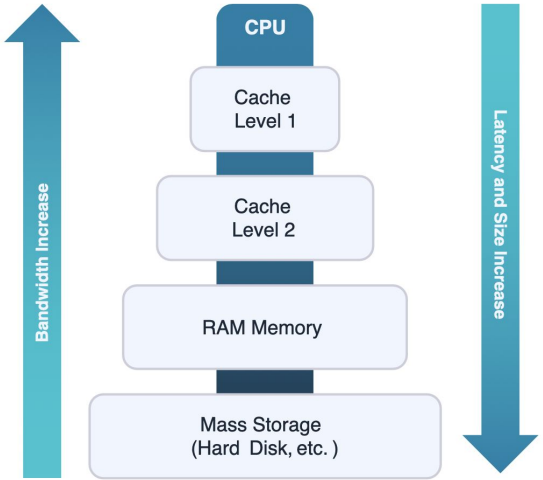
- Imagine you are a window cleaner who is working on a skyscraper.
- You might need to use things like hammers, brushes, cleaner fluid, etc. When you are hanging off a building, the fastest things to use are the tools in your hand. It's very fast, but your hands can only carry so much. When you need to get a less frequently used tool, you have to go all the way down to the ground, grab it, and go all the way back up.
- This is obviously very inefficient, so you decide to bring a backpack with you that carries some less commonly used tools. When you're on the building, grabbing items from your backpack is much faster than grabbing items from the ground, but the backpack only has limited space.

CPU caches

- CPU caches work a similar way. The RAM you have in your computer is very large, but very slow to access. (equivalent to going to the ground)
- In addition to your RAM, there are L1, L2, and sometimes L3 caches on your computer. L1 might be equivalent to your pockets and L2 is equivalent to your backpack.
    - L1 cache is generally 32-128kb large, while L2 can be up to 1MB large.
    - L1 is ~4x slower than L2, which is ~4x slower than L3, which is ~4x slower than RAM.
- Any time you try to read a piece of memory, your **CPU looks in the L1 cache first, then the L2 cache, then main RAM**. It then places that memory in all the caches to be retrieved later.
    - When the CPU finds a piece of memory in the cache, that is called a **cache hit**, otherwise it's referred to as a **cache miss**.

Cache lines

- When your CPU retrieves memory, it doesn't just receive one byte at a time but instead retrieves a 64-byte block called a **cache line**.
- For instance, if I retrieve byte 0, bytes 1-63 are also pulled into the cache.
- Your computer only has capacity for a certain number of cache lines.
    - For example, a 32kb L1 cache would have 512 cache lines (32768 / 64 = 512)
- If you use all of the cache lines, the oldest used one is evicted.
- There are a lot of subtle details to how this works (cache associativity is a big one), but it's not super relevant for this presentation.



Caching in the transpose

- Now that we understand caching, let's look at the matrix transposition algorithm again.
- Each iteration of the loop requires two memory operations, a read and a write. The read is simpler to analyze. Note that the **reads happen in order**, so byte 0 is read, followed by byte 1, followed by byte 2, etc. This has two implications:
    - When we read byte 0 from main memory, it puts bytes 1-63 into the L1 cache.
    - Because the reads happen in order, after we read byte 0 we proceed to read bytes 1-63. Those bytes are already in the cache so we don't have to read from main memory again, which is much faster.

Caching in the transpose

- Let's look at the write now. Note the order of the indices is reversed. Let's say that the width is 1000, then we would be reading bytes 0, 1000, 2000, 3000, etc.
  - Now we are not taking advantage of the cache. Reading byte 0 pulls bytes 1-63 into the cache, but we jump ahead to byte 1000 instead, triggering another main memory read.
  - By the time we loop around and get back to byte 1, that cache line has already been evicted, so we need to grab it again.

```
for (int y = 0; y < height; y++) {
     for (int x = 0; x < width; x++)
{
          output[x][y] = input[y][x];
     }
}
```

Diagram

- Here is an animation depicting what is going on. Some notes to help you understand what is happening:
  - The input array is on the left and the output array is on the right. Green elements represent processed cells, red elements represent unprocessed cells, blue is the current cell.
- Highlighted cells represent cells that are currently in a cache line, with the brightness corresponding to how recently it was acquired (as it fades out, the cache line gets older and is eventually evicted).

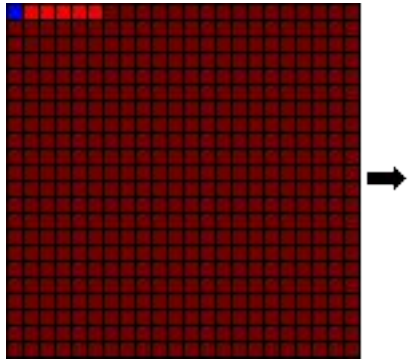
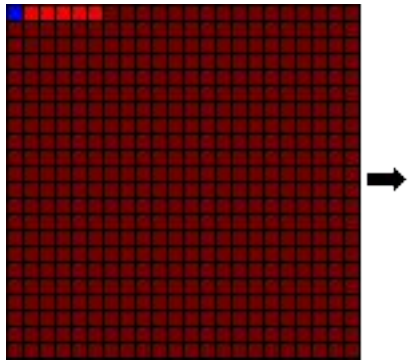


Diagram (cont.)

- Now that we understand what's going on, let's analyze it a bit:
  - On the input side, we are almost always reading in a cache line. The blue square is almost always moved to a bright red square, indicating it's reading a value in a cache line. This is fast and good!
  - On the output side, we are never reading from a cache line! The blue square is always moving to a dark red square, indicating it's not in the cache. The cache is in some sense perpendicular to the array access, which is very slow.



An easy solution

- Since the writes are out of order, why don't we just swap the for loops so the writes become in order?
- Obviously, this will make the reads become out of order. So it's not an easy solution.
- But it does actually generate an improvement! Out-of-cache reads are less bad than out-of-cache writes, so doing the writes in order does cause a slight improvement.
- However, it's clear we still haven't solved the issue.

```
for (int y = 0; y < height; y++) {
     for (int x = 0; x < width; x++)
{
          output[x][y] = input[y][x];
     }
}
```

```
for (int y = 0; y < height; y++) {
     for (int x = 0; x < width; x++)
{
          output[y][x] = input[x][y];
     }
}
```

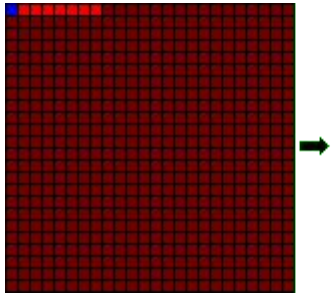
Applying mathematical insight

- In the past few slides we've been focusing on memory cache effects and the "real-world" aspect to matrix transpose. However, there are still ways that math can help us improve our algorithm!
- One important property of the transpose is that it **preserves 2D data locality**. That means if 2 elements are close together before the transpose, they will be close together after the transpose.
  - An example of a nonlocal transformation is squaring, since a cell's value depends on the other values in its row and column.
- Data locality is very powerful because it means **we can split the matrix into blocks and process one block at a time**. This is actually a result you might learn in linear algebra:

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pq} \end{bmatrix}^T = \begin{bmatrix} A_{11}^T & A_{21}^T & \dots & A_{p1}^T \\ A_{12}^T & A_{22}^T & \dots & A_{p2}^T \\ \vdots & \vdots & \ddots & \vdots \\ A_{1q}^T & A_{2q}^T & \dots & A_{pq}^T \end{bmatrix},$$

Applying mathematical insight (cont.)

- Let's split our matrix into small blocks, either 8x8, 16x16, or 32x32, and transpose one block at a time. By staying within the same block, we ensure that the elements don't go out of the cache, allowing us to avoid main memory reads as much as possible.
    - The specific block size that is the fastest is processor dependent, and is based on factors like cache line size and count. When dealing with performance, it's always important to know your hardware and profile your code in the real world.
- The specific implementation is actually recursive -- we keep splitting the grid until the block is smaller than our target size (16x16 in my case), and then perform the transpose on the small-enough block.

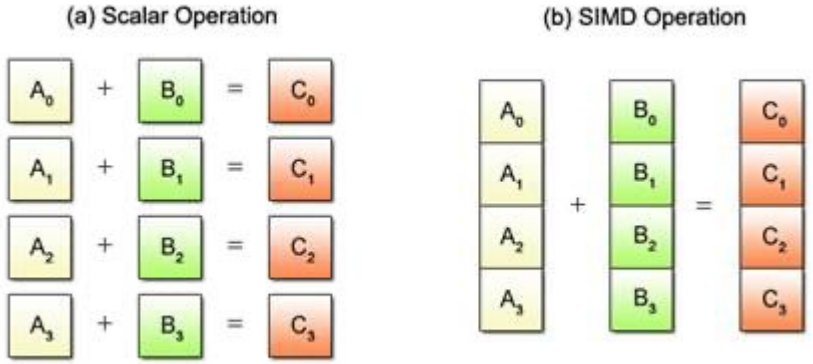


Enter SIMD

- In the original days of computing, processor instructions worked on **one byte at a time**. You might see assembly instructions like "add r1 r2" or "mul r4 3".
- As computers became more powerful, these **processor instructions started to work on multiple bytes** -- first 2 (16 bit), then 4 (32 bit), and now 8 (64 bit). This allowed math with large integers or floating point values to be much faster!
- In the 2000s, processors evolved from being able to work with a single 64-bit value to being able to **work with multiple 64-bit values in a vector**.
- The name for this is **SIMD** -- **S**ingle **I**nstruction **M**ultiple **D**ata. Programmers realized they were often doing the same computation on large arrays, so being able to work with vectors would be beneficial!

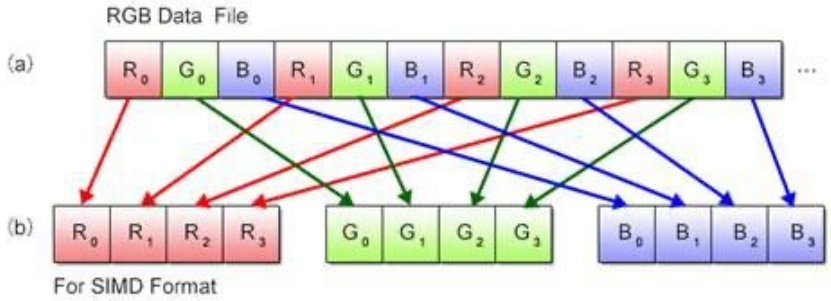
SIMD explained

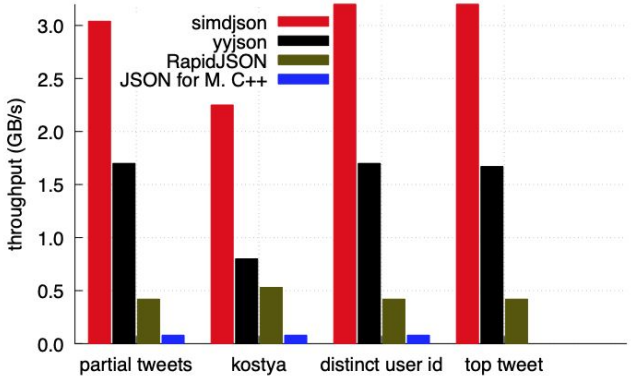
- The workflow for SIMD is roughly the same as normal. Take addition as an example:
  - Normal workflow: load 2 values from memory, call the add instruction, store the result
  - SIMD workflow: load 2 vectors (containing several values each), call the simd add instruction, store the result.
- The caveat with SIMD is it's like working with larger integers – the bytes *must* be contiguous.



SIMD applications

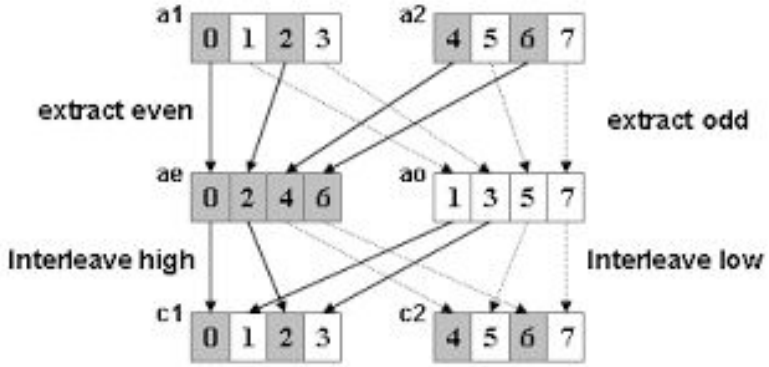
- SIMD can be used for a lot of things, such as splitting an RGB image into its color channels.
- Recently, SIMDJson was released which parses JSON dozens of times faster than previous algorithms. SIMD is often found in 3D graphics due to its performance requirements and the presence of vector math.





SIMD operations

- There are a lot of operations you can do with SIMD. We've talked about addition, but two we are looking at are extract and interleave:
  - Extract – extracts every other element from two vectors and puts it into a new vector
  - Interleave – takes the top or bottom half of two vectors, and puts them in a new vector alternating the order.
- Note that since we are taking 2 vectors as input and only outputting 1 vector, we can only process half the input.

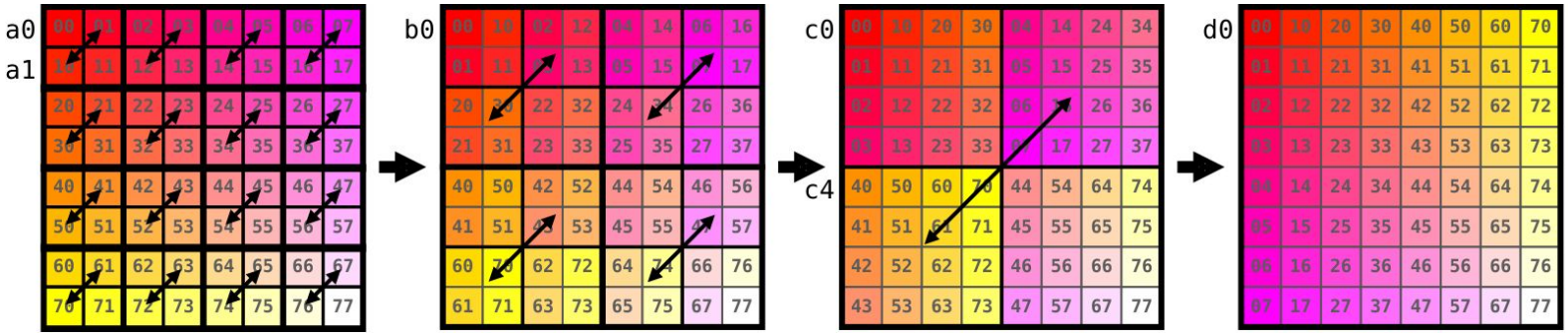


Applying SIMD to the transpose

- Recall our current algorithm loads and stores one byte per operation. This doesn't take advantage of 64-bit registers, let alone vector registers! So let's try to adapt our algorithm to process multiple elements at once.
- The most commonly supported "wide register" is the 128-bit register, which is equivalent to 16 bytes. Recall that SIMD only works on **contiguous bytes**, so it makes the most sense to use SIMD to transpose a 16x16 matrix.
- However, it's not obvious how do do this efficiently. If the first row contains 16 elements, each of those 16 elements needs to go to a different register! Naively this would take 16 * 16 = 256 operations, which is just the same as the non-SIMD approach. So we must find a better way.

Our SIMD algorithm

- The solution is to use block decomposition again. Recall most SIMD operations operate on two registers at a time.
- First, we reinterpret the 16x16 matrix as a 2x2 matrix of block size 8, and perform a transpose. Then, we reinterpret it as a 4x4 matrix of block size 4, and perform a transpose of that. Finally, we interpret it as an 8x8 matrix of block size 2, and perform a transpose of that.



Benefits of this algorithm

- We're only working with 2 rows at a time. When transposing with block size 2, the first row of blocks fits neatly into 2 SIMD registers.
- For block size 4, note that elements in row n go to elements in row n + 2, and vice versa. So we can work with rows in pairs again, fitting them into 2 SIMD registers.
- In general, the total amount of SIMD operations is 16 * 4 = 64 operations (plus 32 for loading and storing the data), which is much smaller than the 256 we would need naively.
- The result is an algorithm that is significantly faster than the non-SIMD version!

Wrap up

- Matrix transpose is an problem that seems super trivial, but having good knowledge of both mathematics and computer hardware can lead to a 10x speedup!
- It’s important to know your hardware – SIMD is insanely underutilized despite having been around for decades.
- If you are interested in this type of optimization problem solving, there are plenty of jobs that require someone that can integrate deep knowledge of math and programming.