Optimization Technique and Implementation Project Presentation Script

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

Professor Brandon Bass

Student: Suresh Ghimire

UC ID: 005038288

Date: August 22, 2025

Presentation Script

**Slide 1: Title Slide**

Good afternoon, my name is Suresh Ghimire. I am presenting my research project on "Optimization Technique and Implementation" for Professor Bass's Algorithms and Data Structure course.

During this presentation, I will guide you through every step of how data structure optimization can result in very real gains in performance in high-performance computing applications with some great results that might surprise you.

**Slide 2: Introduction**

When you think of high-performance computing, what do you normally think about? The most common thought is of course a parallelism - we are talking about throwing more cores at a problem, a problem being applied in parallel to multiple processors. Parallelism is definitely important but there is an equally important but overlooked concept when looking at performance in parallel applications - memory hierarchy performance.

This project is focused on data structure optimization for better data locality. If it helps to think about it in another way, whenever your CPU is consuming data, it is extremely fast at digging through it; so, if you leave the CPU waiting on memory to arrive - that power "dissolves" in a matter of seconds..

So, we set out to simply replace pointer-based data structures with contiguous arrays identifying the potential benefit of vectorization techniques and to also identify the potential benefit of pre-allocation.

The challenge was simple - let's collect some measurements to determine the actual style of performance improvements that we are capable of producing through controlled experiments.

**Slide 3: Background**

The premise of the work is built on an abundance of research. A number of studies have demonstrated that large toggles in performance can be mitigated through the optimization of both the data structure and memory access patterns.

The reasoning behind why this works is underpinned by the following science:

By replacing fragmented memory constructs with contiguous arrays - we get increased cache hits. That means instead of fetching memory with scattered pieces, you can now load chunks of data you will ultimately use.

The better your spatial and temporal locality, the less Translation Lookaside Buffer misses there will be - TLB is critical in performance of virtual memory management.

Vectorization and Single Instruction Multiple Data-friendly layouts allow your processor to operate on quantities of data and perform the same operation simultaneously, increasing overall throughput.

Pre-allocation ? You remove the cost for repeated dynamic memory allocation during runtime; and to clarify, in the case of large memory allocation, the memory is pre-allocated at the neighborhood of execution size, not too slow fat costs in request size.

All of these improvements have quantifiable performance gains - not just works.

**Slide 4: Choose Technique**

So why did we specifically choose data locality optimization?

The technique itself is simply a way to organize data contiguously in memory, improving the locality of reference. This matters because it is generally tackling one of the biggest bottlenecks in modern computing, memory wall.

Importance cannot be overstated. Bad locality means cache misses, memory stalls, and most importantly, processors sitting idle waiting for data. Good locality means fewer cache misses, and we all know that performance exponentially increases in that respect.

Our evidence base includes researchers like Yoo et al., who showed huge performance improvements with these same techniques, in 2013.

We chose this technique for three reasons.

First, it has exceptional high impact in HPC workloads, where performance is paramount.

Second, it can be used in Python natively, using NumPy - this makes it real and practical.

Third, it is directly related to data structure performance - this dovetails nicely with our course objectives.

But how do you implement this technique in practice?

**Slide 5: Implementation in python**

Now we enter the land of practicality! Our plan for implementation had three building blocks.

First: Replace python list of tuples with NumPy n-array for contiguous storage.

In reality, python lists are arrays of pointers to objects which together and separately can be scattered all around memory. Practically, with a list of tuples you are playing memory skip. NumPy n-arrays organize data in contiguous blocks of memory - and this is truly what modern processors love.

Second: Apply vectorization using NumPy broadcasting to do batch operations on n-real arrays.

Instead of writing a loop, which processes one element at a time, NumPy broadcasting allows us to perform operations over a n-array of elements in one go. The real beauty is that the underlying C implementation is highly optimized in order to leverage instruction.

Third: Use np.empty() to pre-allocate an output n-array and avoid allocations.

Dynamic allocation during computation is overhead and each time you dynamically allocate memory you are adding overhead. This problem can totally be eliminated with pre-allocating n-arrays of the correct size.

Collectively, the above three actions improve cache utilization and reduces computational overhead. What's exciting is that these optimizations are largely transparent to the programmer - you achieve greater performance without sacrificing readability in code.

**Slide 6: Strength and weakness**

Honesty is best practice, let’s analyze the above both strengths and weaknesses.

**Strengths:**

The performance gains are enormous (you will see the numbers in moment); at the same time we are seeing reduced memory utilization (which is critical in large-scale applications). The above process scales exceptionally well, and we are equipped with techniques that are language agnostic (not only for python).

**Weaknesses:**

Although you can do these techniques, you need to have a decent base knowledge of numerical computing libraries - you cannot simply drop these techniques into any codebase and hope for the best without understanding the principles.

It needs fixed dimensions. If your data is size changes considerably, then the gains might diminish.

Potentially, you can lose - for example, if you over allocate the n-array size, then you potentially waste some memory

Although, in most HPC applications, these disadvantages are far outweighed by the performance advantages.

**Slide 7 - Results and Analysis**

Alright, results - and now, we are in very interesting territory.

We tested this optimization on multiple datasets of different sizes: 200, 400, 600, 800, and 1000 data points.

The optimized approach had a speedup of up to 97 times. Let me repeat that - Ninety-seven times, compared to the baseline implementation.

That, is not a typo. We are talking about 97 seconds to compute vs. 1 second. For the end user of this codified solution, this could mean the difference between waiting hours for a result and waiting minutes.

The improved implementation provided somewhere between 20 to 30 percent savings in memory. This is very important when we talk about working with large datasets that might be able to be held in available RAM.

The important point to focus on is that these results will scale, as the size of the datasets increases, the performance improvement grows.

We were pleased with this result, as it matched well with the research done with Yoo and colleagues, in 2013, which verified both the versatility of our implementation and the broader theoretical implications.

These were not just incremental improvements. These are transformational improvements in performance.

**Slide 8: Conclusion**

So what can we conclude from this research?

Optimizing data structure for better data locality doesn't necessarily lead to marginally improved HPC performance - it leads to radically enhanced performance. The results are compelling.

Even more exciting, Python with NumPy is able to achieve performance approaching or equal to traditionally compiled languages C or Fortran. It can combine the best of both worlds, an intro for researchers without coding experience to engage in high performance computing.

These practical benefits, coupled with the scalability of these techniques, makes them especially important in the world of large-scale scientific computing. In this domain, the ability to efficiently extract information from massive datasets is typically the difference between undertaking meaningful research vs what is truly impossible.

Here is the lesson: I think it is reasonable to assume these optimizations may be as effective as a hardware upgrade. Rather than spending thousands of dollars on faster processors or more RAM , think about optimizing the data structures to achieve the functionality you need.

In a world where efficiency equals research capacity, energy expenditure, and ultimately costs, it is reasonable and justified to identify these techniques as more than just academic exercises, but as a necessity.