

# **Optimizing AI Workloads for Enhanced Performance: A Comprehensive Framework for Comparing CPU and GPU Architectures**

In-Depth Benchmarking and Evaluation of Multithreading, OpenMP, SIMD, Quantization, Compiler Selection, and CUDA Tuning for Accelerated Machine Learning Tasks

- 
- 01 Objective
  - 02 Benchmarking Setup
  - 03 Naive Implementation
  - 04 Multithreading
  - 05 OpenMP
  - 06 Compiler & Build Tools
  - 07 SIMD
  - 08 Quantization
  - 09 Other Optimizations
  - 10 Initial vs. Final Benchmarks
  - 11 GPU-Based Optimizations
  - 12 CPU vs. GPU

## Goal

- Create a unified AI framework optimized for both CPUs and GPUs
- Evaluate the impact of various optimizations on performance across different hardware systems
- Develop a real-world machine learning model to demonstrate the framework's effectiveness

## Research Questions

- How do CPU and GPU architectures compare in AI performance?
- What optimizations most improve performance for different problem sizes?
- What are the hardware limits for performance improvements?

## Methodology

- Test on diverse hardware: NVIDIA GPUs, AMD/ Intel CPUs, and Apple Silicon
- Perform benchmarks with 12 runs, discarding the first two
- Use synthetic datasets (XL/ XXL) to evaluate optimization under different loads
- Analyze memory usage and power consumption during benchmark tests

## Hardware

CPU	Release Date	TDP (W)	Cores	Threads
AMD Ryzen 7 3800XT	07.07.2020	105	8	16
Apple M3 Pro 11-Core	30.10.2023	27	5	11
Intel Core i7 1065G7	01.06.2019	15	4	8
GPU	Release Date	TDP (W)	CUDA Cores	Threads
NVIDIA GeForce RTX 2080	20.10.2018	215	2944	1515
NVIDIA GeForce MX350	01.02.2020	20	640	1354

## Naive Implementation

## Preparations

## Select Core Functions

- Identify essential functions for AI framework (e.g., add, ReLU, matrix multiplication)

# Use MNIST Dataset

- Select MNIST for a manageable, well-understood image classification task

## Choose Training Framework

- Implement model training using TensorFlow v1, selected for its documentation and wide usage

## Export Model Weights

- Use a human-readable text format for exporting weights, avoiding complex formats like ONNX

# Naive Implementation

## Implementation

### Matrix Module

- Defines struct and handles memory management (e.g., `malloc_matrix`, `free_matrix`, `print_matrix`)

### Input/Output Module

- Defines struct and loads weights and image data (e.g., `malloc_io`, `free_io`, `io_to_matrix`, `get_value`)

### TensorFlow Module

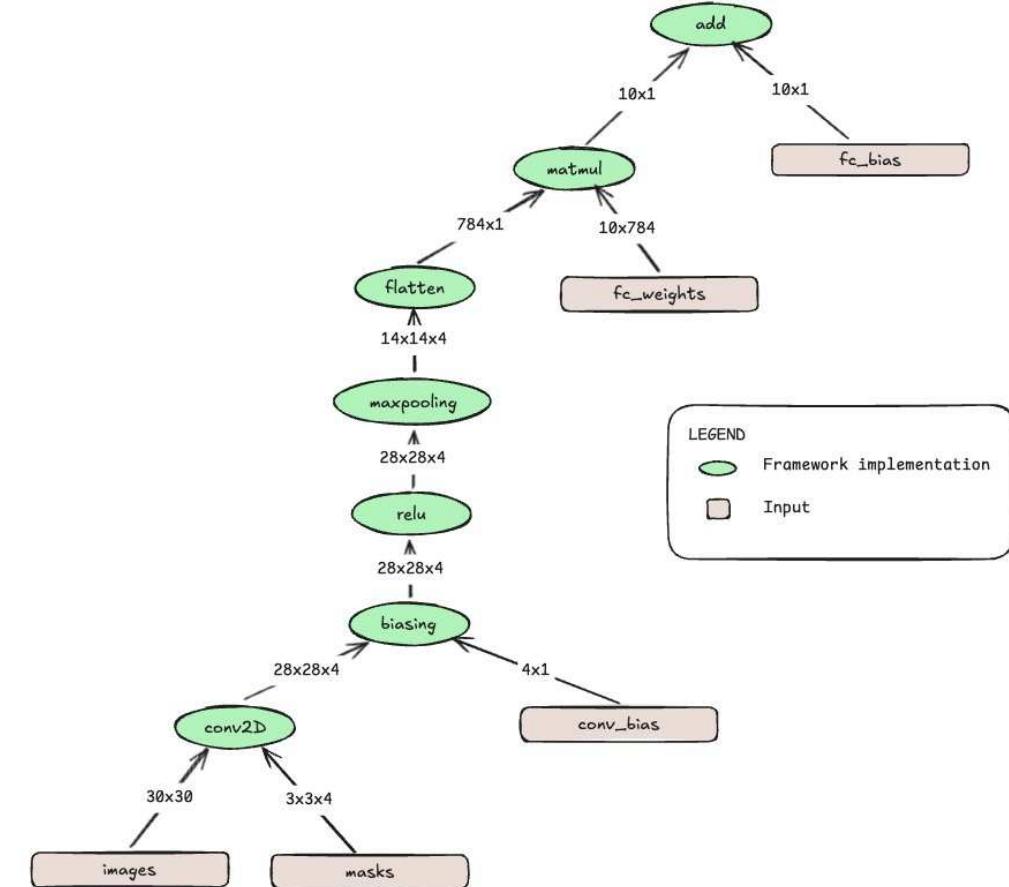
- Implements basic functions like `add`, `conv2d`, `relu`

### Timer Module

- Provides timing functions to measure performance (`start_timer`, `stop_timer`, `delta_time_us`, `delta_time_s`)

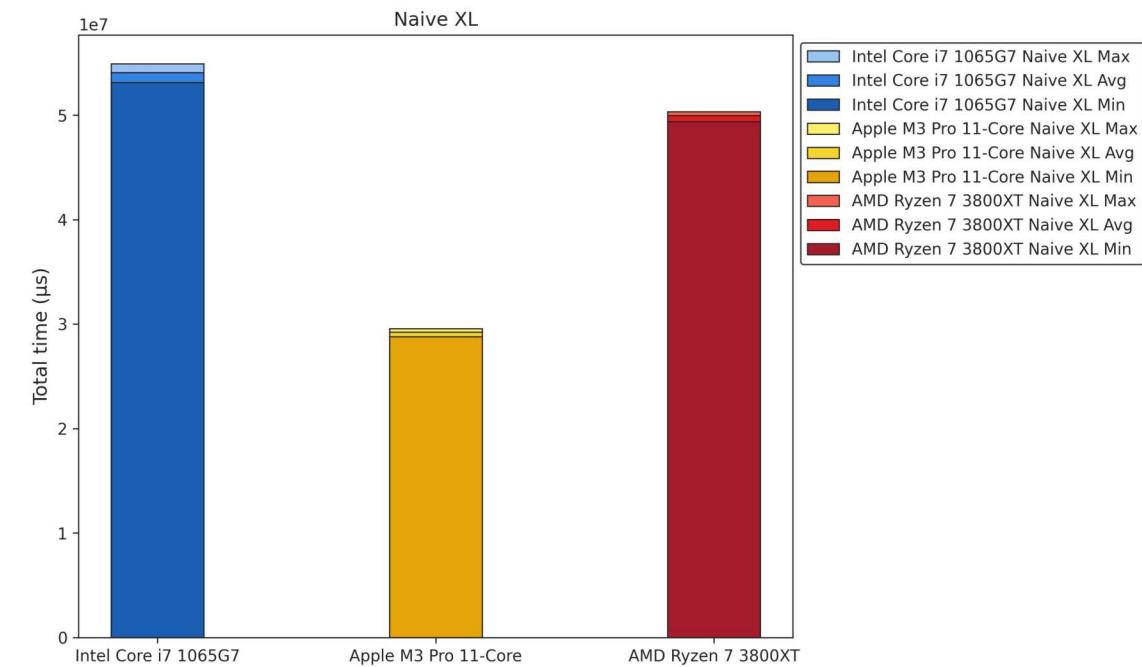
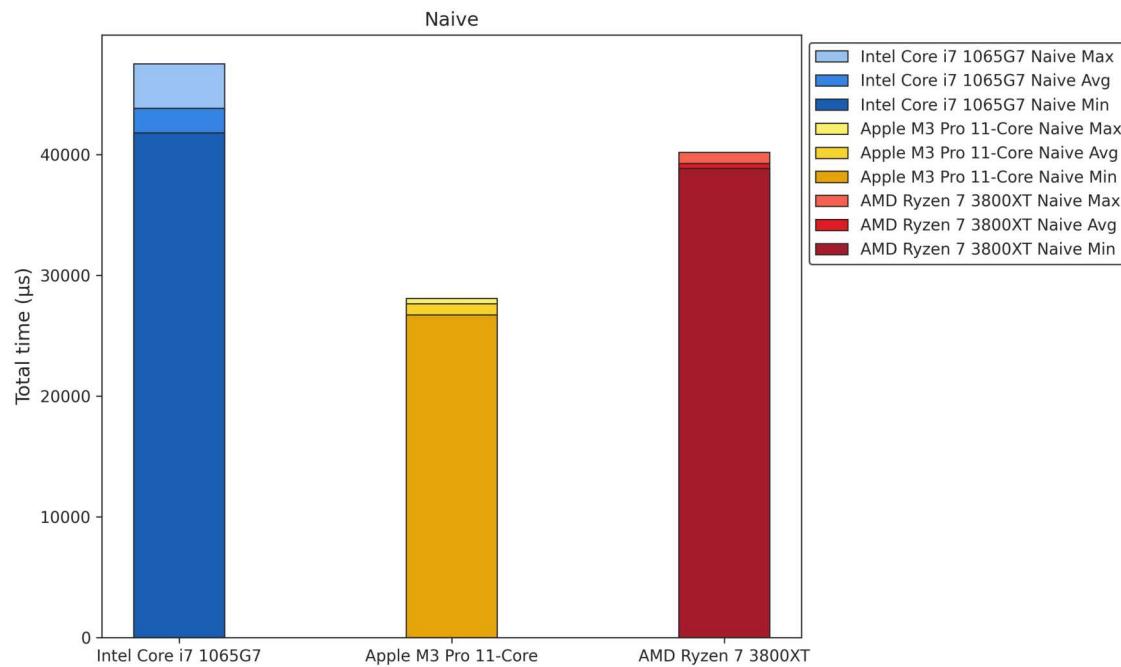
### Main Modul

- Integrates all modules, runs the AI graph, and logs performance metrics



# Naive Implementation

## Benchmarks



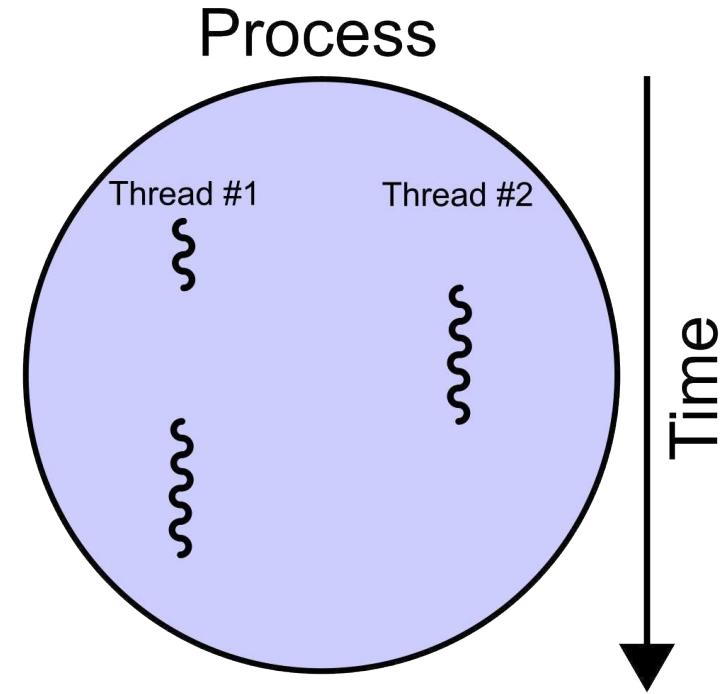
# Multithreading

### Multithreading

- **Concurrent Execution**
  - Allows multiple tasks to run simultaneously in a program
- **Increased Efficiency**
  - Utilizes CPU resources better, reducing idle time
- **Faster Performance**
  - Helps improve application responsiveness and speed

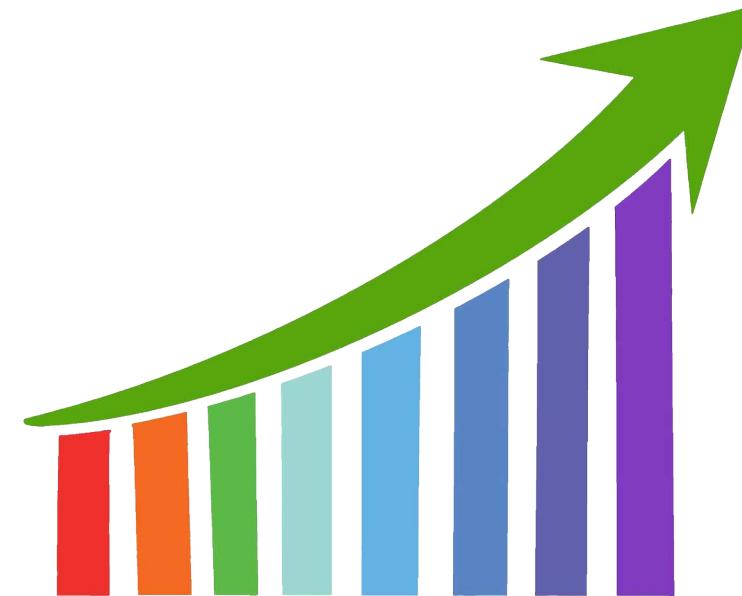
### Pthreads

- **POSIX Threads**
  - A standardized API for thread management
- **Cross-Platform**
  - Works on most Unix-like systems (Linux, macOS)
- **Thread Control**
  - Provides tools for creating, managing, and synchronizing threads



### Anticipated Benefits

- **Faster Processing**
  - Speeds up data processing
- **Better Resource Utilization**
  - Leverages multi-core processors for parallelism
- **Scalability**
  - Makes it easier to scale the system to handle large AI models and datasets



### Proof of Concept

- **Independent Thread Creation**

- Each function created and joined its own threads

### Thread Pool Implementation

- Thanks to the suggestion of Philipp Holzinger :)

- **Optimized Thread Lifecycle**

- Threads are created once at the start and joined once at the end

- **Task Management**

- A GAsyncQueue is used to manage tasks efficiently

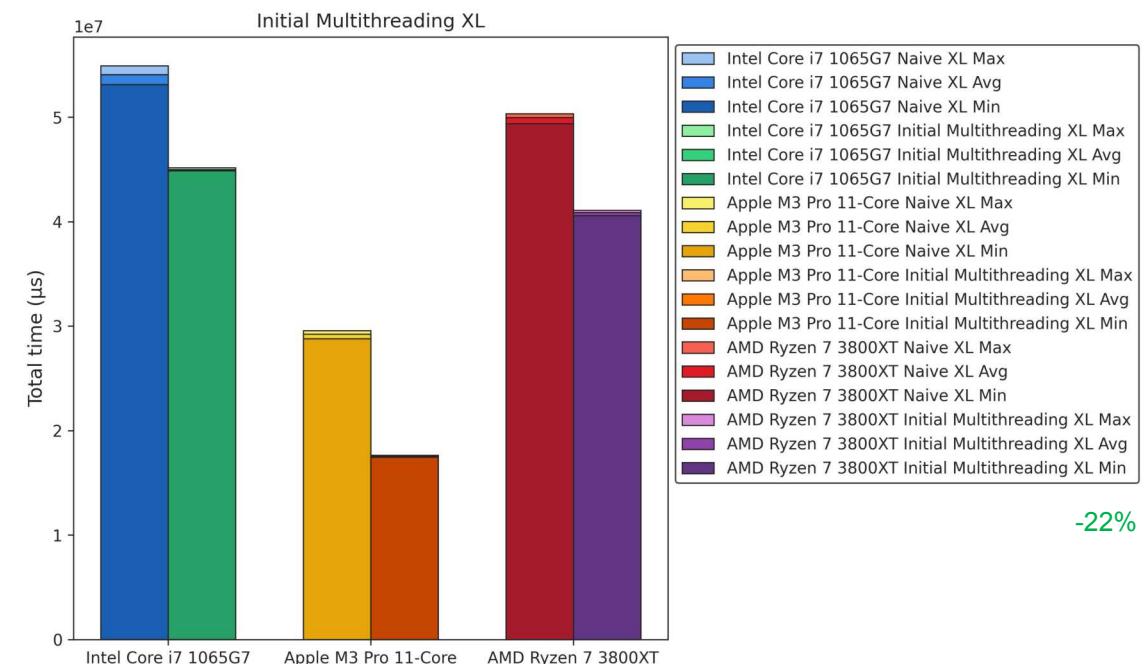
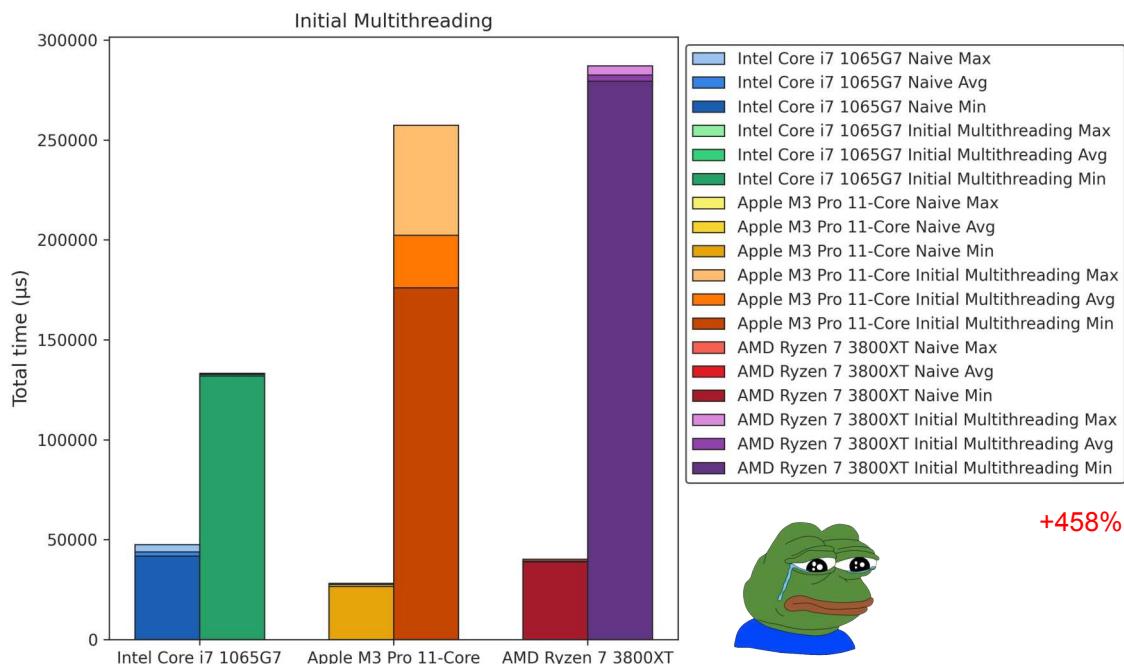
- **Poison Pill Mechanism**

- Threads continuously execute tasks until they receive a "poison pill" to terminate

```
void create_mt(long threads) {
    THREADS = threads;
    if(queue == NULL) {
        queue = g_async_queue_new();
    }
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    mt_arg *mt = malloc(THREADS * sizeof(mt_arg));
    for(long i = 0; i < THREADS; i++) {
        mt[i].idx = i;
        pthread_create(&tids[i], NULL, start_mt, &mt[i]);
    }
}
```

# Multithreading

## Benchmarks



### Smart Multithreading Implementation

- **Problem:** Overhead exceeded performance gains for some functions
- **Change:** Multithreading used only for sufficiently large tasks

### Optimized mt\_arg

- **Problem:** mt\_arg caused high memory and copy overhead
- **Change:** Reduced mt\_arg size

### Cache Optimization

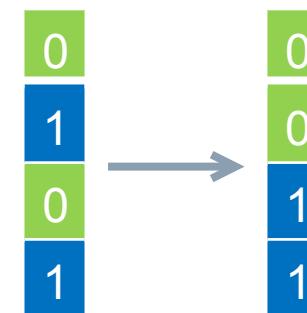
- **Problem:** Poor cache performance due to inefficient thread distribution
- **Change:** Optimized task allocation

### Dedicated Multithreading Class

- **Problem:** Image processing couldn't be multithreaded
- **Change:** Created a dedicated thread management class

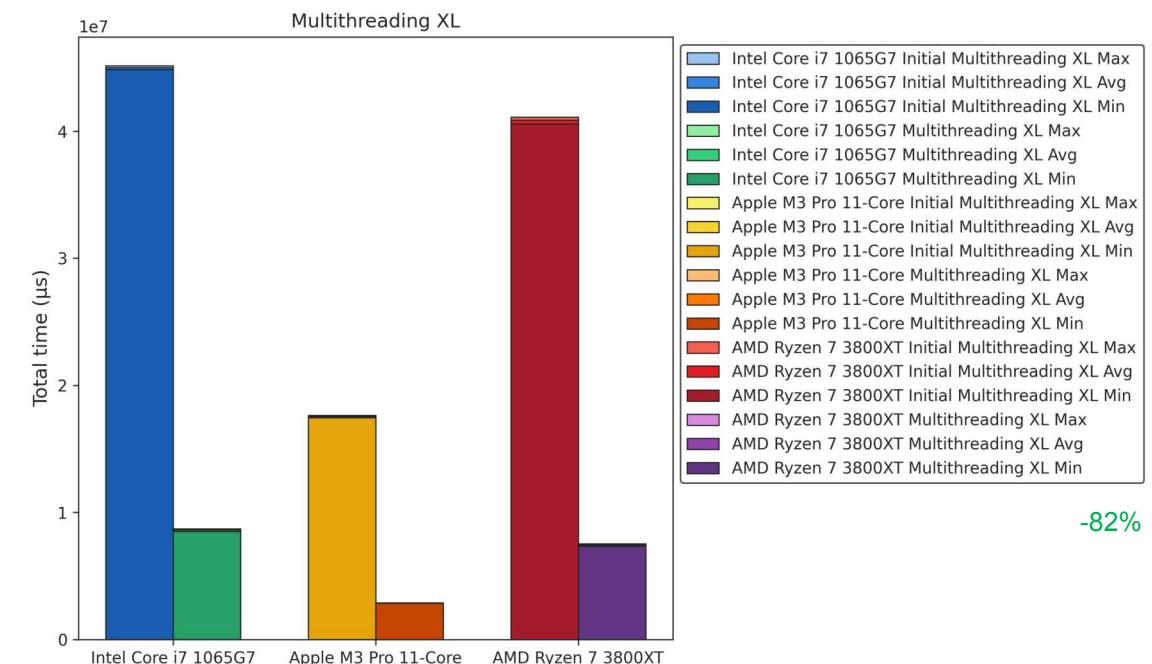
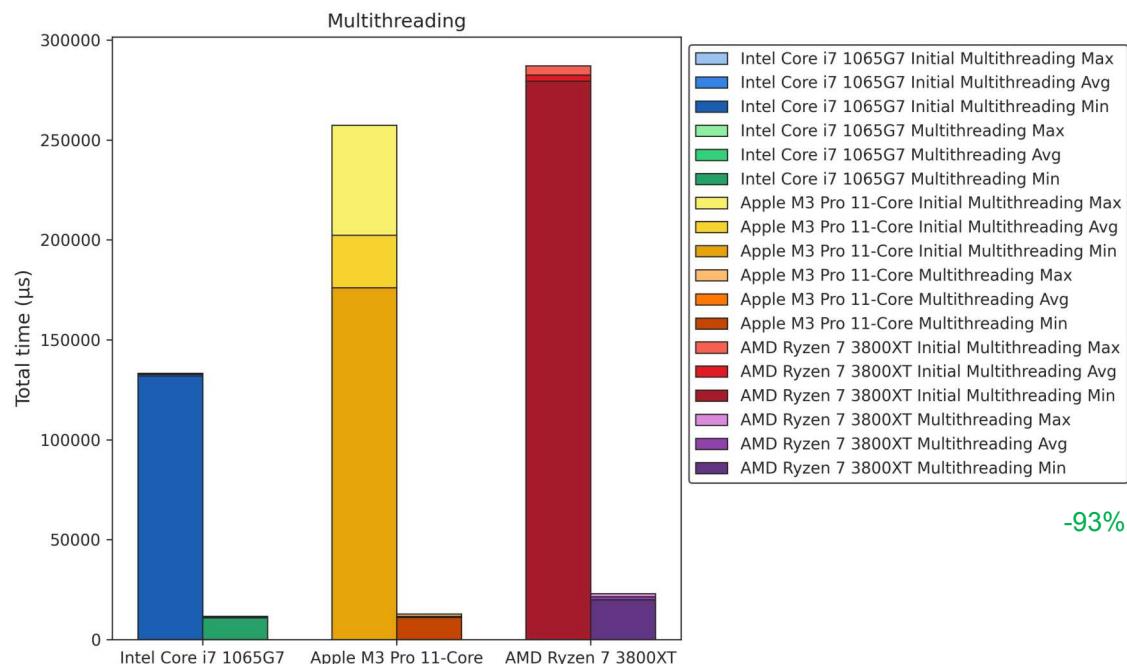
### Multithreaded Image Processing

- **Problem:** Sequential image processing was inefficient
- **Change:** Added concurrent image processing



# Multithreading

## Benchmarks



# OpenMP

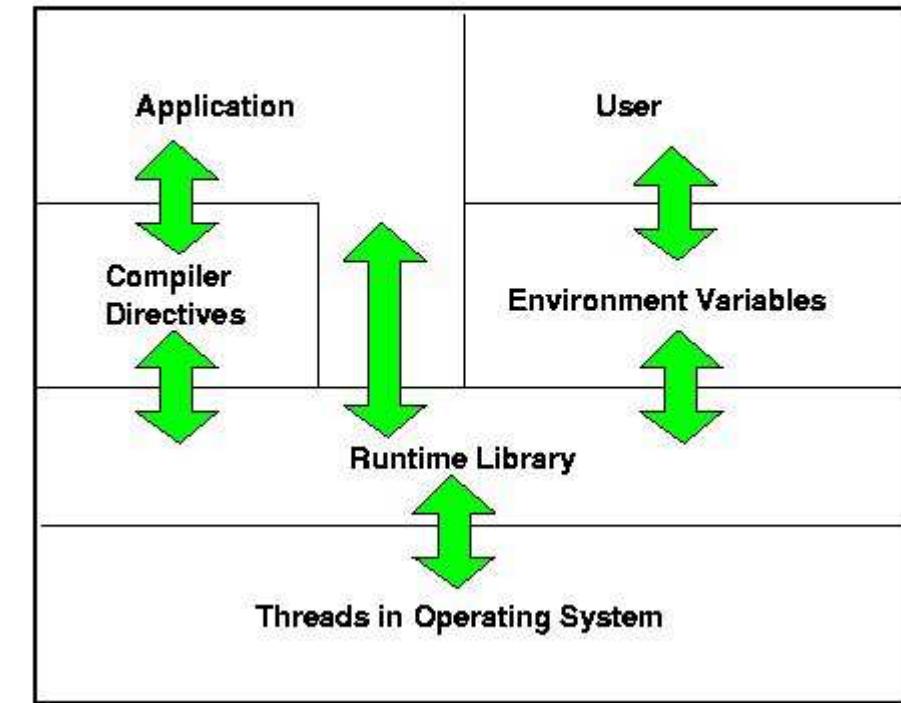
### OpenMP

- API for parallel programming in C and C++ for several CPU- and GPU-architectures
- Simplifies parallel programming by managing threading on compiler side

### Basic Function

- Uses pragmas (compiler directives) to define parallel regions in code
- Distributes work across cores based on current load
- Compiler handles most parallelization details, reducing transparency and flexibility

OpenMP Architecture



### Compiler Directives

- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp simd`

### Library Functions

- `omp_set_num_threads()`
- `omp_get_num_threads()`

### Environment Variables

- `OMP_NUM_THREADS`

```
/*
 * @param a: Input-Matrix a
 * @param b: Input-Matrix b
 * @param c: Output-Matrix c
 * @param mt: Multithreading object
 * @return: Matrix c as a result of a + b
 */
matrix *add(void *a, void *b, matrix *c, mt *instance) {
#pragma omp parallel for collapse(2)
    for(int i = 0; i < c->x; i++) {
        for(int j = 0; j < c->y; j++) {
            // ...
        }
    }
}
```

### Compiler Flags (Depending On Architecture)

- AMD: -Xcompiler -fopenmp
- Arm: -Xclang -fopenmp
- Intel: -Xcompiler -qopenmp

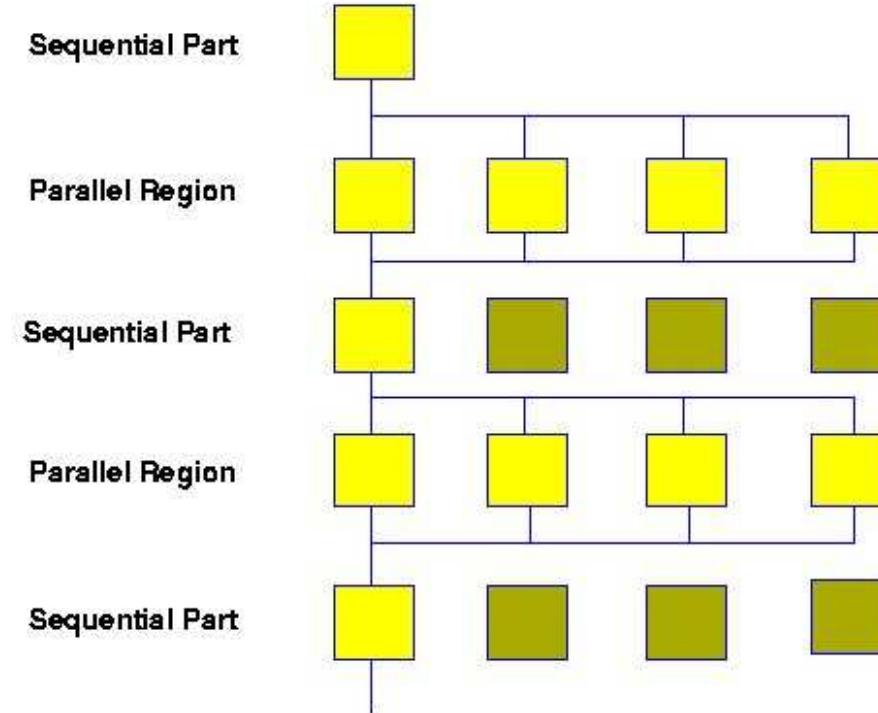
### Offload Targets

- Executes computations on devices like GPUs or specialized hardware
- Requires compiler support

### Warning

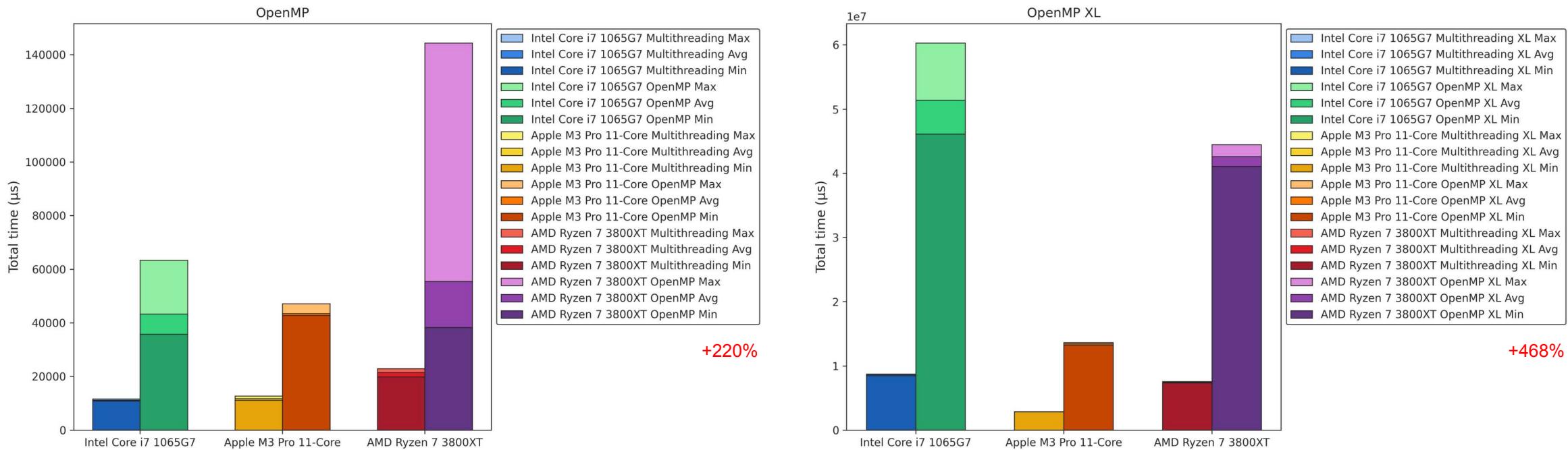
**warning:** loop not vectorized: the optimizer was unable to perform the requested transformation; the transformation might be disabled or specified as part of an unsupported transformation ordering [-Wpass-failed=transform-warning]

### OpenMP Execution Model



# OpenMP

## Benchmarks



# Compiler & Build Tools

### ICPX (Intel oneAPI DPC++/C++ Compiler)

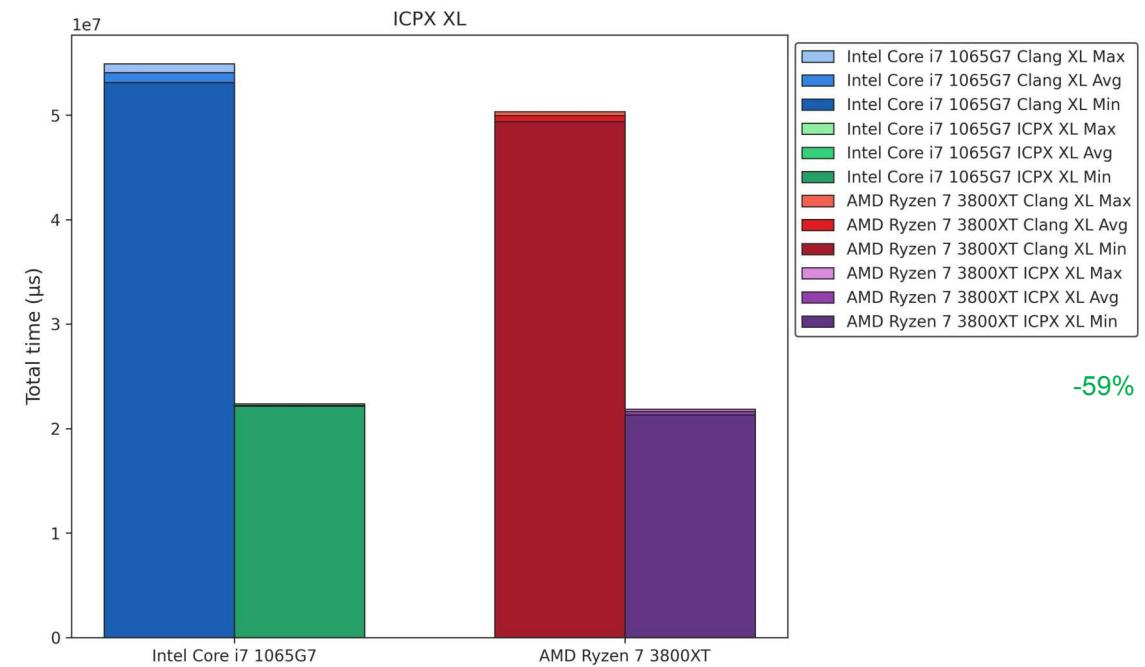
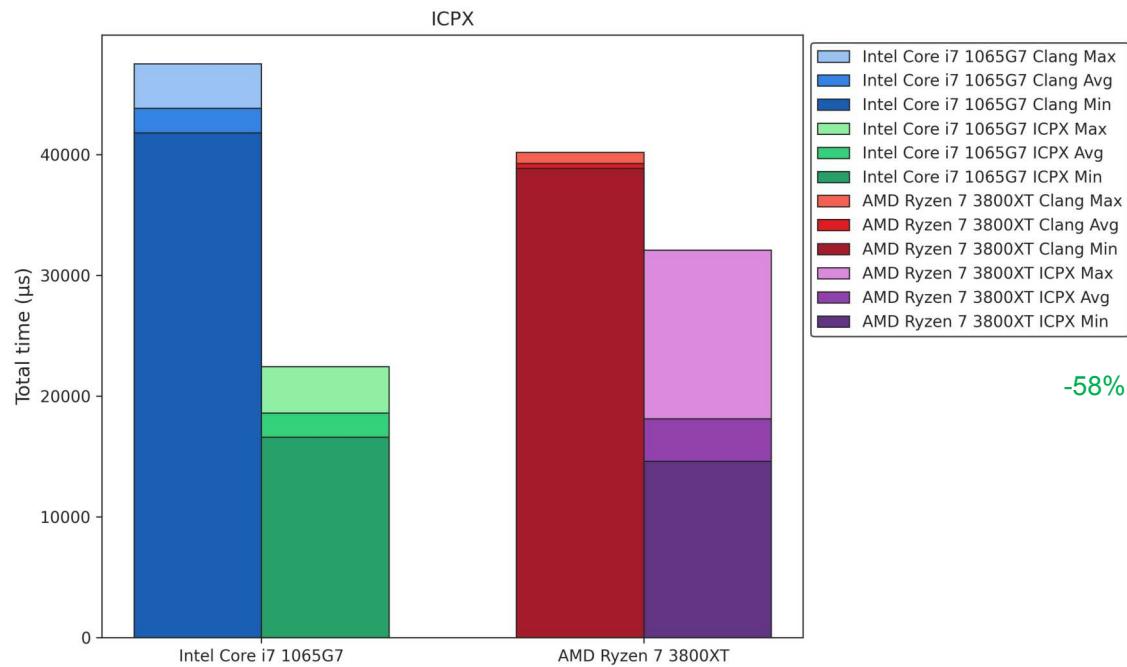
- **Former Name:** Intel C++ Compiler (ICPC)
- **Developer:** Intel
- **Primary Focus:** Optimized for Intel hardware (CPUs, GPUs, FPGAs)
- **License:** Freeware, Proprietary (under Intel's terms)
- **Supported Platforms:** Windows, Linux
- **Key Features:**
  - Supports threading with Intel oneAPI Threading Building Blocks (TBB), OpenMP, and native threads
  - Supports proprietary hardware features
  - Integration with Intel libraries for performance optimizations, such as Intel MKL and IPP

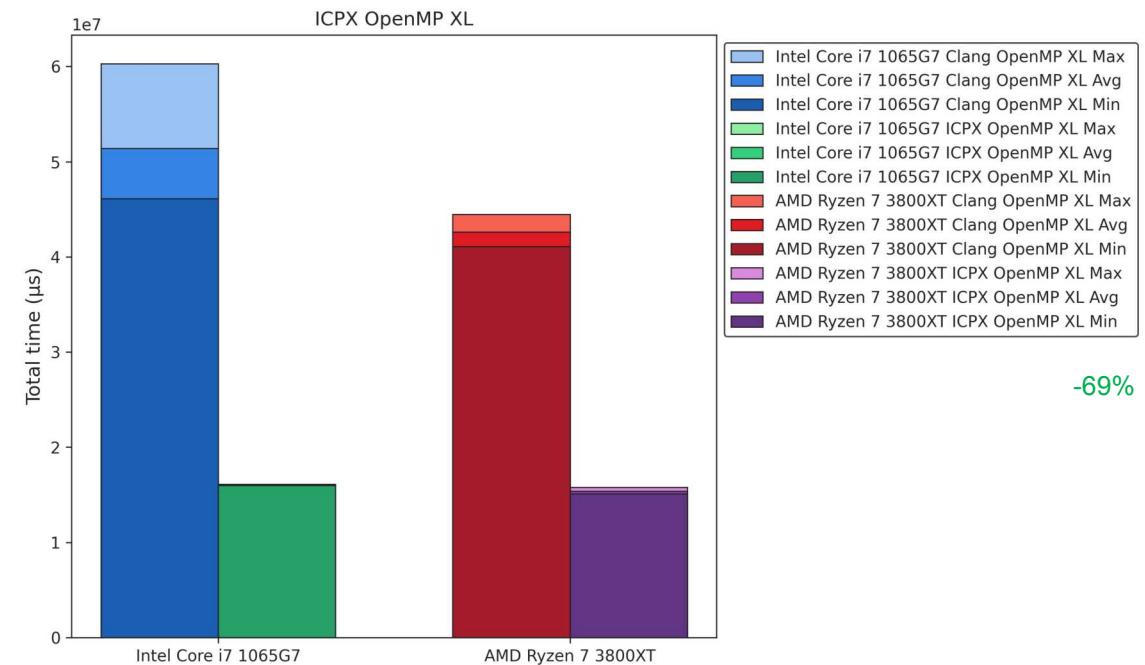
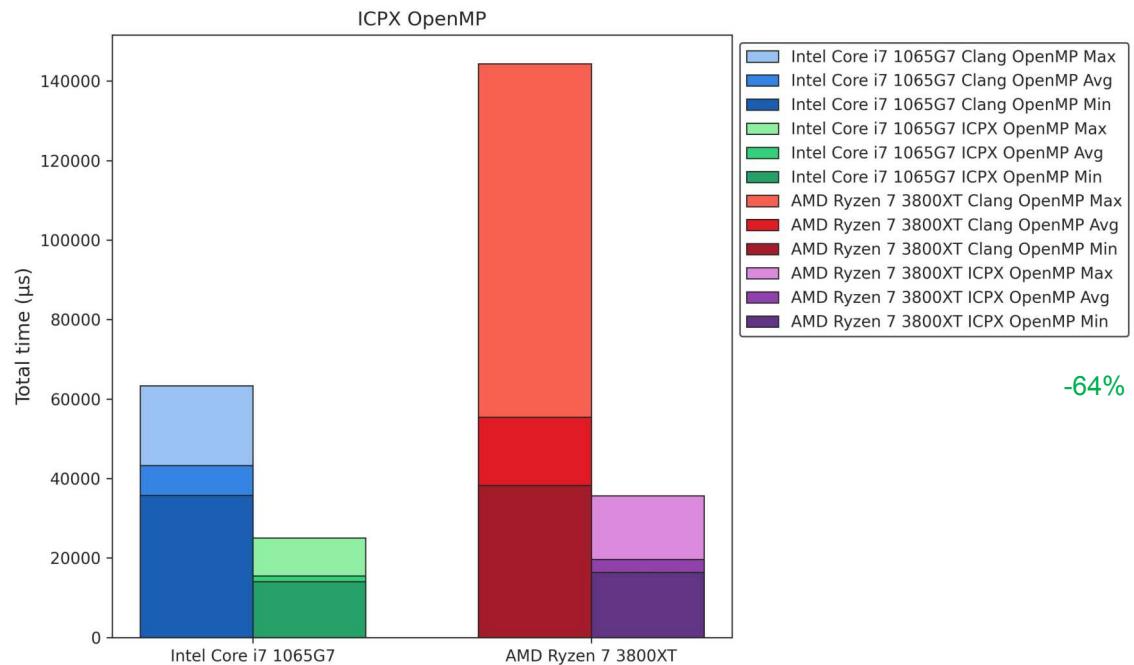


### Clang

- **Former Developer:** Apple
- **Developer:** LLVM Project
- **Primary Focus:** Supports a wide variety of architectures (x86, ARM, etc.)
- **License:** Open Source (Apache License 2.0)
- **Supported Platforms:** Windows, Linux, macOS, and more
- **Key Features:**
  - Supports OpenMP and native threads
  - Great for cross-compiling across platforms
  - Extensive tooling support for debugging and static analysis







### Reasons

- **Reduces Target Bloat**
  - Helps avoid exceeding the current 26 Makefile targets
- **Simplifies Config Selection**
  - Handles over 1024 possible CFLAG combinations
- **Improves User Experience**
  - Enhances usability for AI framework users

### Dialog

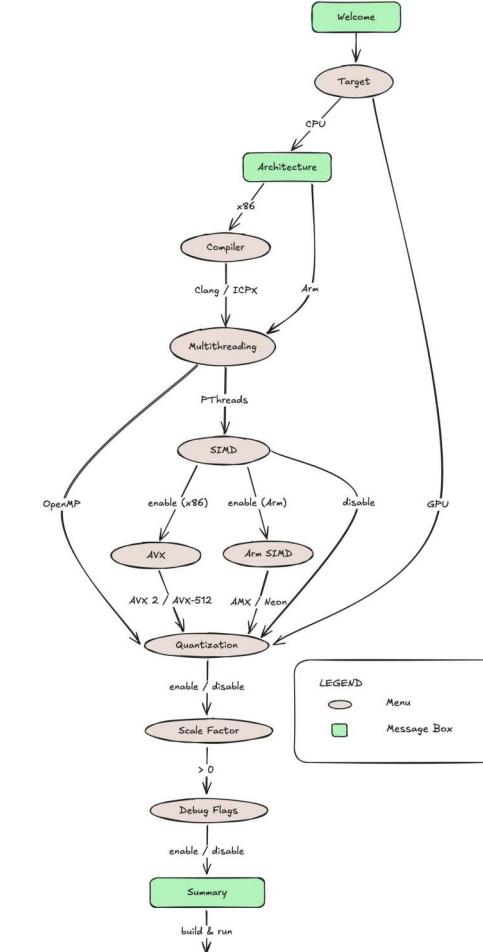
- **Create Interactive UIs**
  - Build text-based interfaces in shell scripts
- **Multiple Input Types**
  - Includes menus, checklists, and forms
- **Enhanced UX**
  - Streamlined and user-friendly command-line interface

# Build Tools

## Dialog 2/ 2



- Implemented flowchart logic in a Bash script
- Integrated the script into a Makefile target



# SIMD

### Parallel Processing

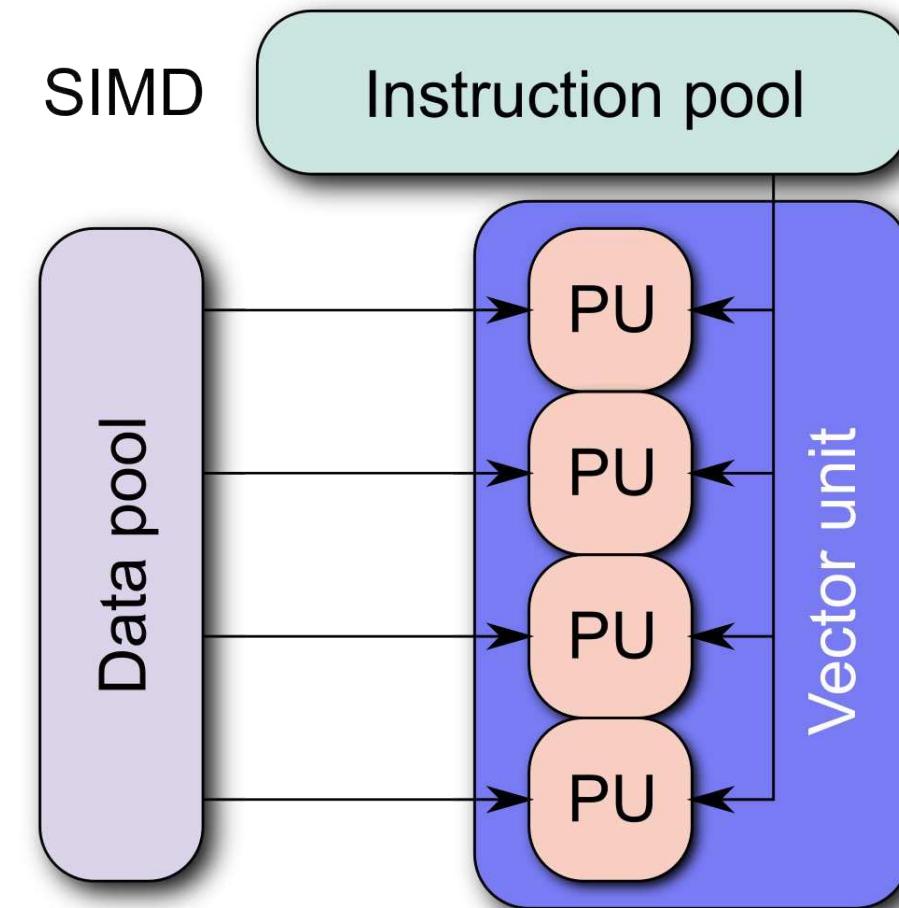
- Enables a single instruction to operate on multiple data elements at once, boosting computational efficiency

### Vectorization

- Utilizes vector registers, which can hold several data values, to perform simultaneous operations on them in parallel

### Broad CPU Support

- Different processors have their own SIMD extensions
  - AMD:** SSE and AVX2
  - Arm:** Neon
  - Intel:** SSE, AVX2, and AVX-512



### Overview

- SIMD unit for Arm processors
- Introduced in ARMv7 (currently available in ARMv9)
- Optimizes data manipulation using 32x128-bit vector registers
- Functions are available in C/ C++ via `arm_neon.h`

### Benefits

- Provides direct access to Neon instructions, eliminating the need for assembly code
- Simplifies migration between processors (AArch32 to AArch64)

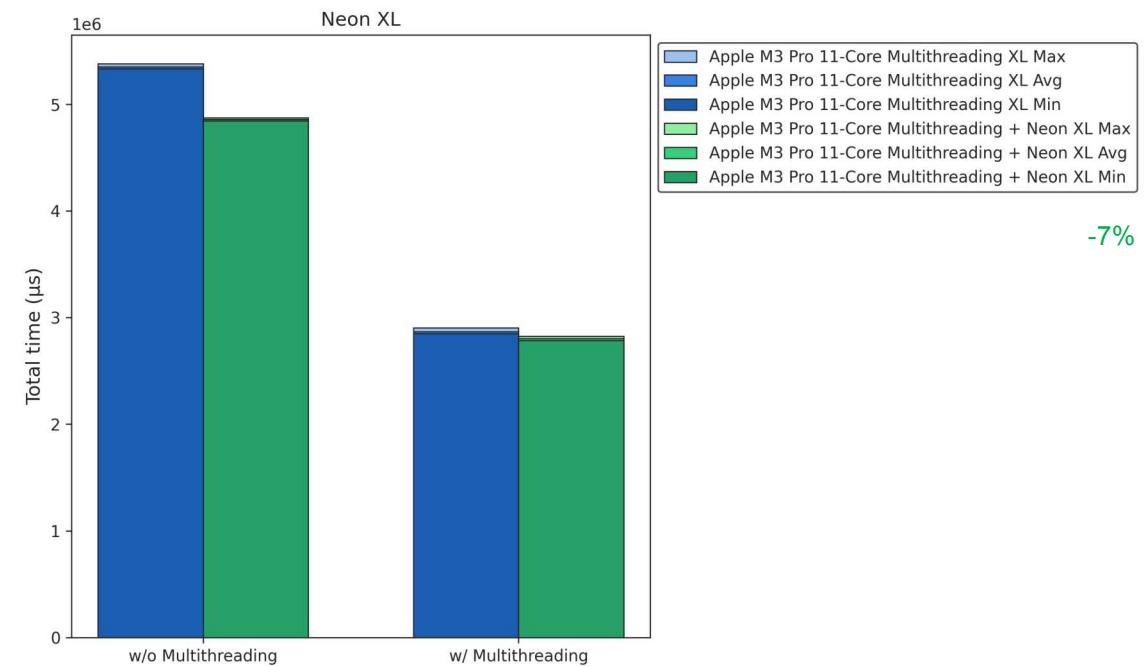
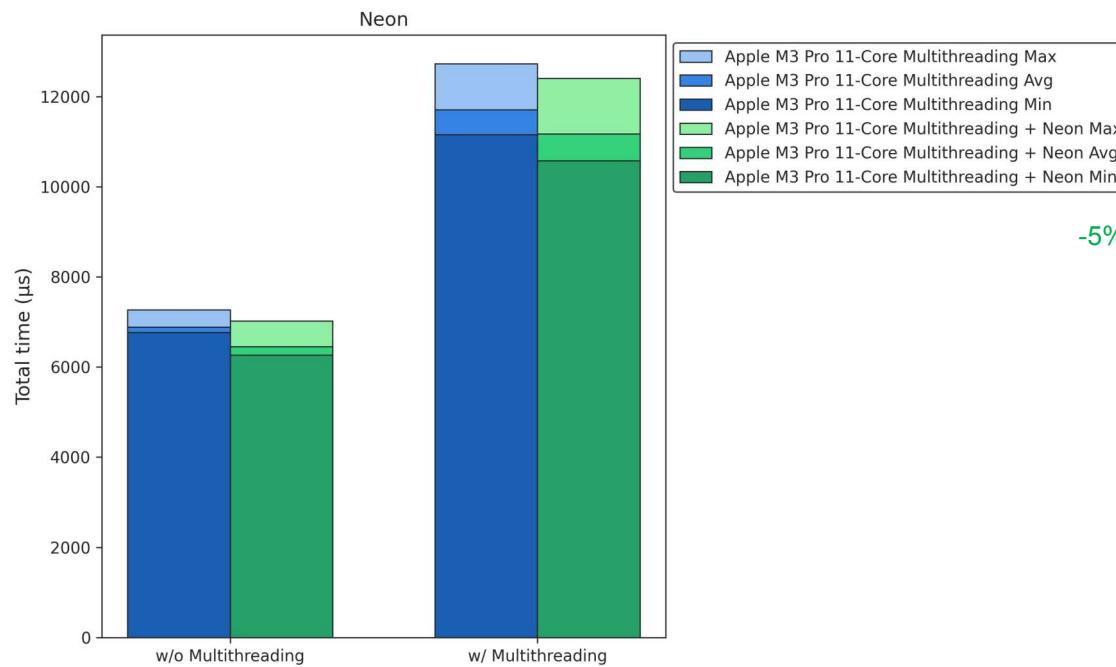
### Useful Features

- Supports various data types
- Handles unaligned data, but decreases performance
- Includes `vaddv` (Vector Add Across Vector), enabling horizontal addition across vector element - ideal for operations like `conv2d()`



# Arm Optimization

## Neon Benchmarks



### Overview

- Reverse engineering of the **AMX** (Apple Matrix Extension) chip for Apple's ARM-based Silicon (M1, M2, etc.)
- **AMX Registers**
  - **X (or Y) Registers**
    - 4 source registers for computations
    - Containing 64 floating-point values in total
  - **Z Registers**
    - Serves as the accumulator or output registers
    - Holds a 64x64 floating-point matrix

### Implementation

- `void matmul_simd(mt_arg *mt)`
- `void conv2d_simd(mt_arg *mt)`

### AMX Instructions

- **Defined in aarch64.h**
  - Implemented as macros for assembly instructions
  - Utilize bitmasks (bm) to define instruction behavior
- **Key instructions**
  - `AMX_SET()` and `AMX_CLR()`
  - `AMX_LDX(bm)` or `AMX_LDY(bm)`
  - `AMX_STZ(bm)`
  - `AMX_FMA32(bm)`

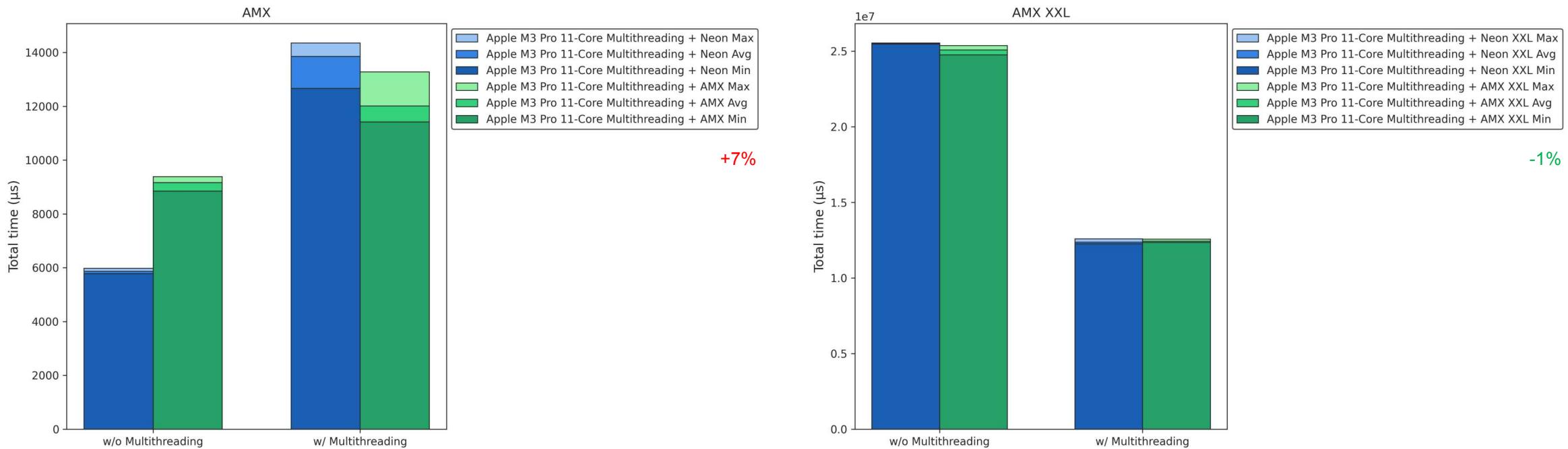
### Power Consumption

- Manually measured with powermetrics
- Average of more than ten values

Benchmark	mW
AMX	7331
Neon	7331
AMX XL	7218
Neon XL	7340

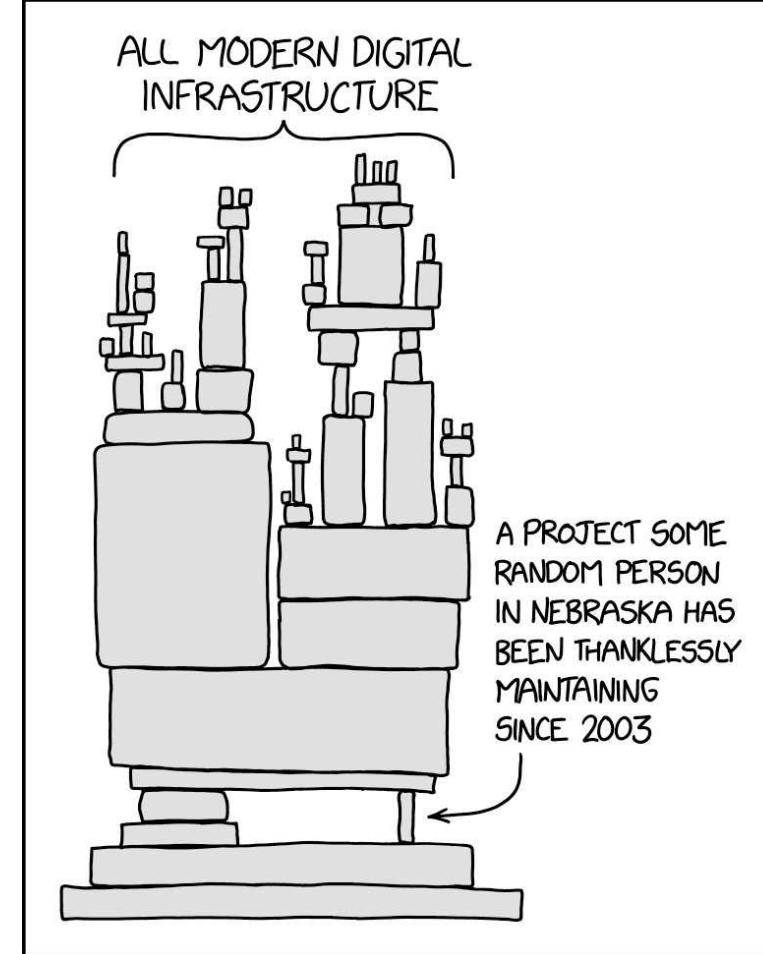
# Arm Optimization

## AMX Benchmarks



### Challenges

- Relying on a personal GitHub repository
- Lack of Documentation:
  - Difficult Implementation
  - Difficult Troubleshooting
- Limited amount of supported data types
- Limited instruction set
- Optimized for specific workloads



## SSE (Streaming SIMD Extensions)

- **128-bit vector processing:** Basic SIMD instructions for parallel processing
- **Broad compatibility:** Supported on most CPUs

## AVX2 (Advanced Vector Extensions 2)

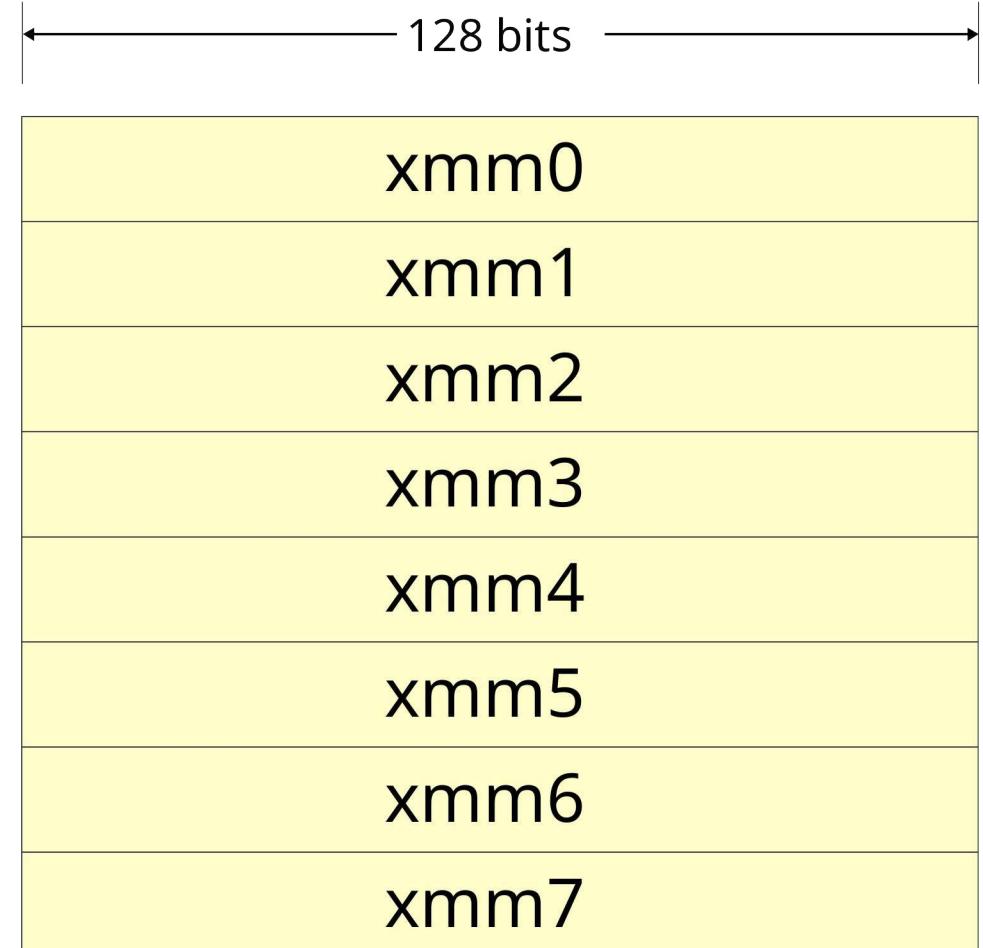
- **256-bit vectors:** Doubles the vector width for better data processing.
- **Needs breakdown for horizontal addition:** Requires splitting vectors into two

## AVX512 (Advanced Vector Extensions 512)

- **512-bit vectors:** Processes larger data sets with greater parallelism
- **reduce\_add:** Introduces a new instruction for efficient reduction by addition
- **Supports padding through masks:** Eliminates the need for fallback sequential code
- **Intel-only:** Primarily supported on Intel CPUs, not available on most AMD processors

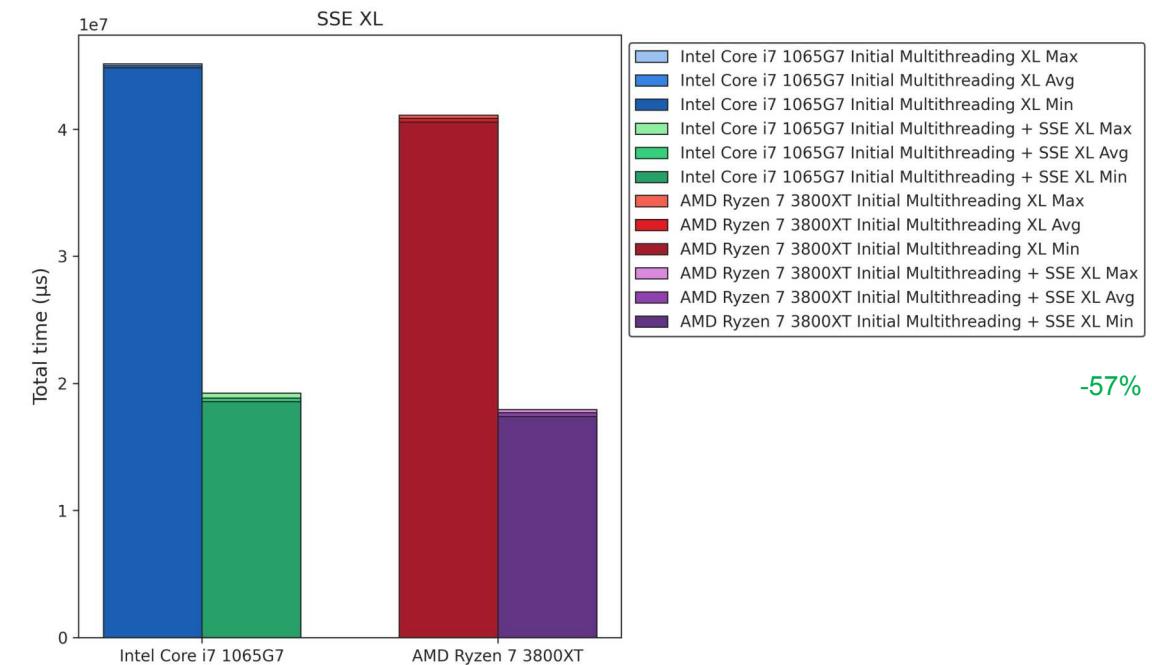
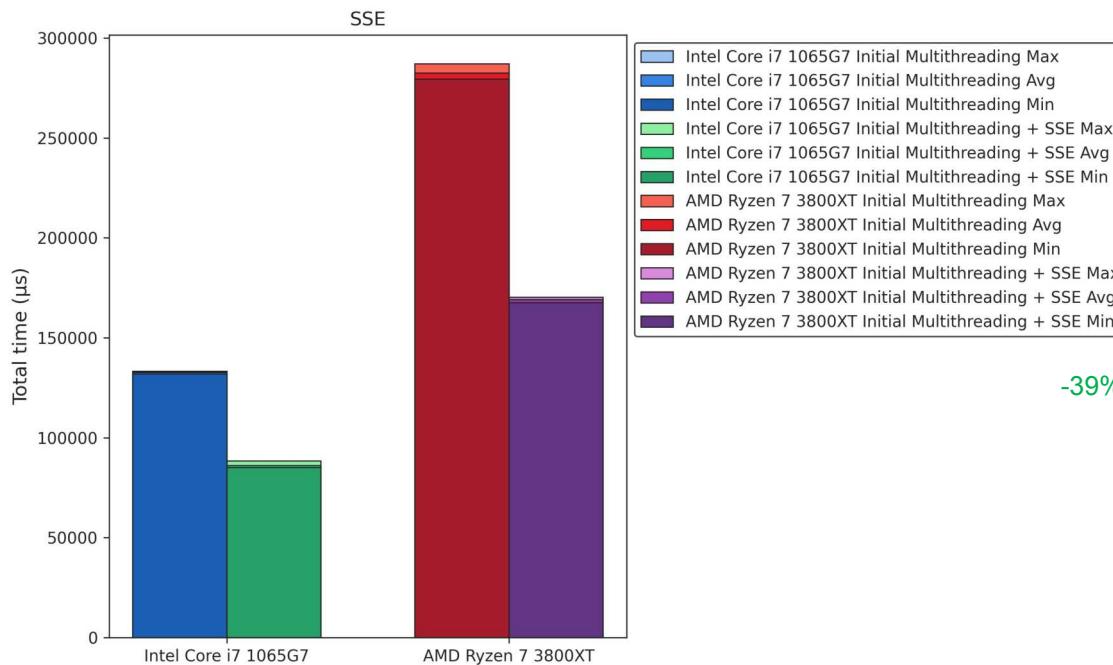
```
AVX2: (*mt->c)->m[get_idx(mt->i, mt->j, (*mt->c)->y)] =  
_mm_extract_ps(_mm_hadd_ps(_mm_hadd_ps(_mm_add_ps(_mm256_castps256_ps12  
8(c), _mm256_extractf128_ps(c, 1)), _mm_setzero_ps()), _mm_setzero_ps()), 0);
```

```
AVX512: (*mt->c)->m[get_idx(mt->i, mt->j, (*mt->c)->y)] = _mm512_reduce_add_ps(c);
```



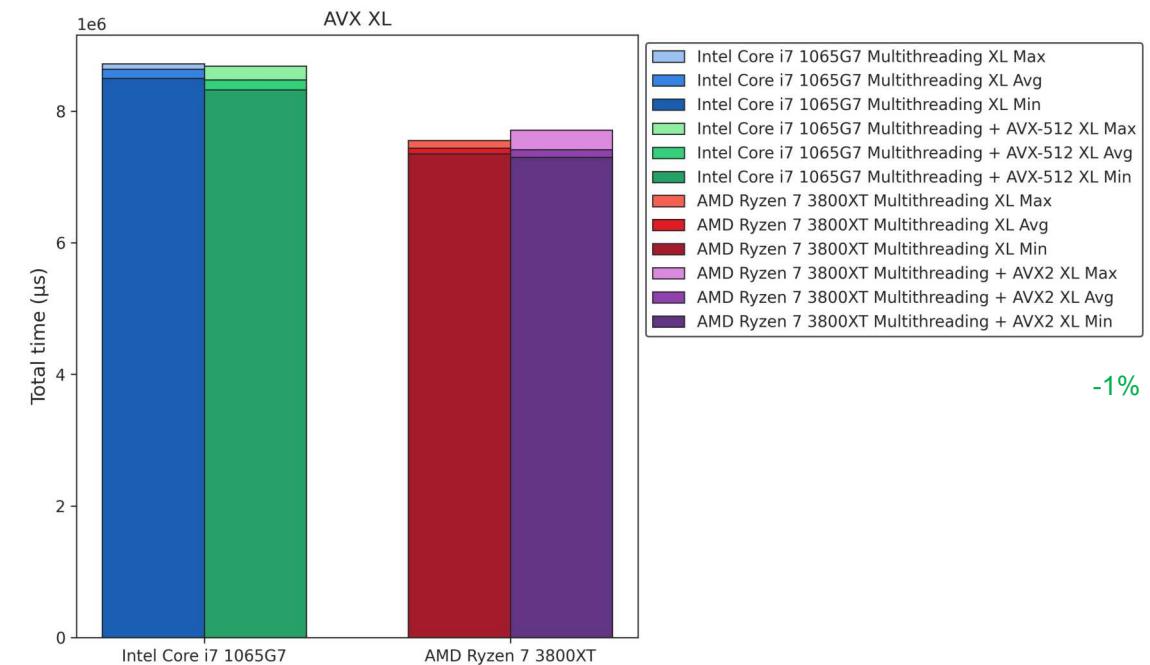
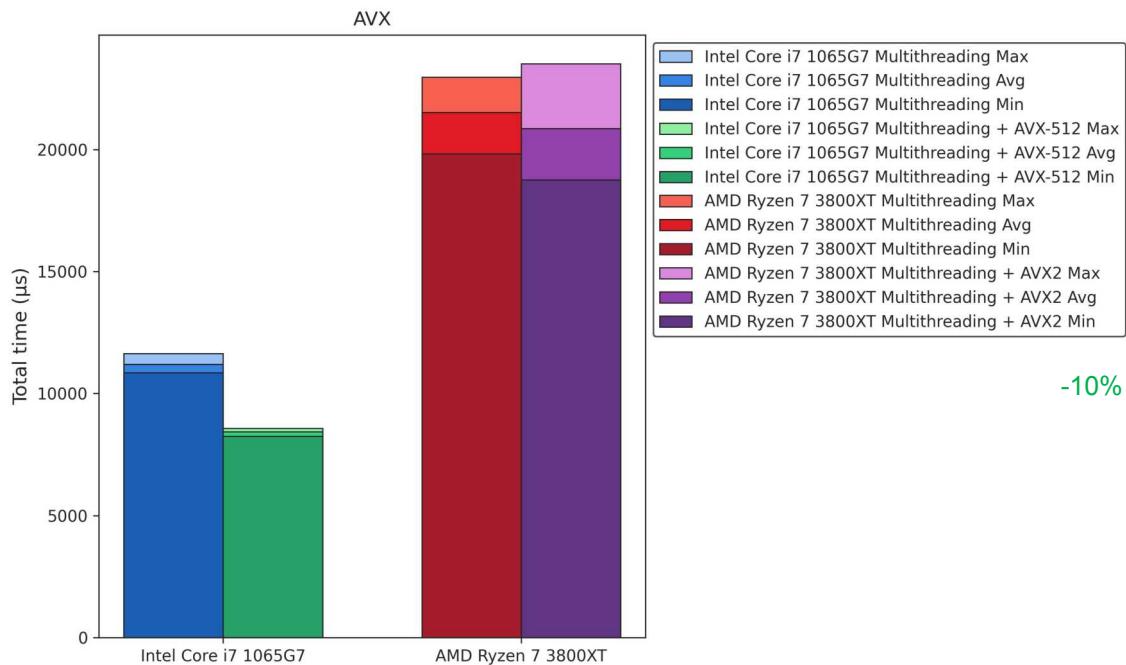
# x86 Optimization

## SSE Benchmarks



# x86 Optimization

## AVX Benchmarks



# Quantization

### Date Type Optimization

- **Change:** int8 for input values and int32 for intermediate results
- **Benefit:** Achieving higher efficiency with a minimal reduction in accuracy (decreasing by 5% to 8%, resulting in approximately 90% accuracy)

### SIMD Integration

- **Change:** Added mixed data type support in AVX2, AVX-512 and Arm Neon Code
- **Issue:** AMX does not support int8
- **Benefits:** Process even more elements in parallel, significantly boosting computing speed

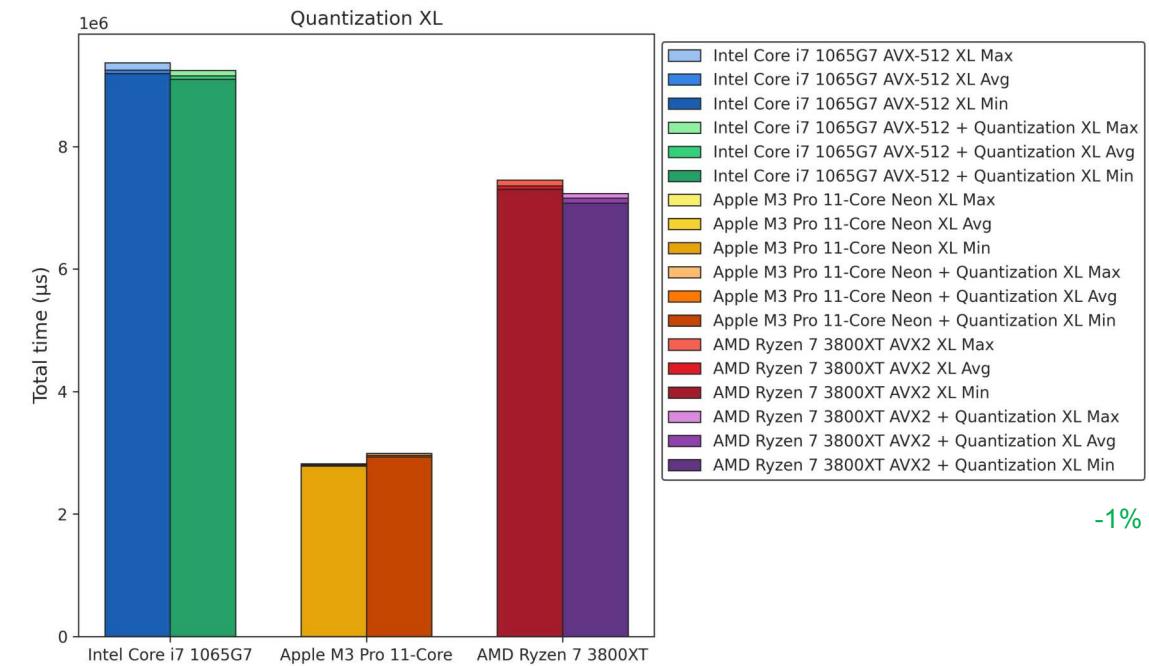
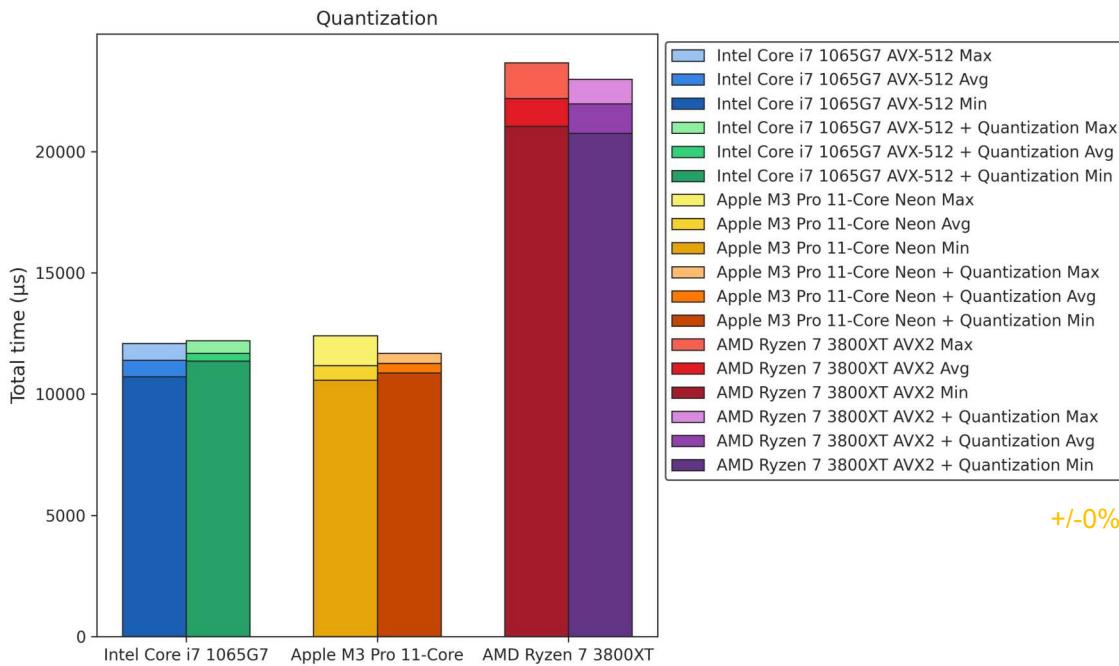
### Memory Reduction

- **Change:** Model size reduced (~15 %)
- **Issue:** Multithreading memory overhead
- **Benefit:** Frees up memory on resource-limited devices

Benchmark	Bytes
NO SIMD	2,409,961
NO SIMD_INT	2,050,771

# Quantization

## Benchmarks



# Other Optimizations

## Codebase Migration

- Migrated from C to C++ to resolve extern "C" compatibility issues
- Facilitated seamless integration with CUDA

## Performance Optimization

- Adopted flat array representation for better cache utilization and easier CUDA integration
- Optimized memory allocation by relocating memory requests outside of loops to enhance performance
  - `matrix *add(matrix *a, matrix *b);`  
→ `matrix *add(matrix *a, matrix *b, matrix *c = NULL);`

## Data Preprocessing

- Pre-transposed fc\_bias and fc\_weights to avoid repeated transposing

## Code Efficiency Improvements

- Integrated transpose directly into flatten to speed up data processing
- Removed unused functions to reduce complexity and binary size

## Code Optimization

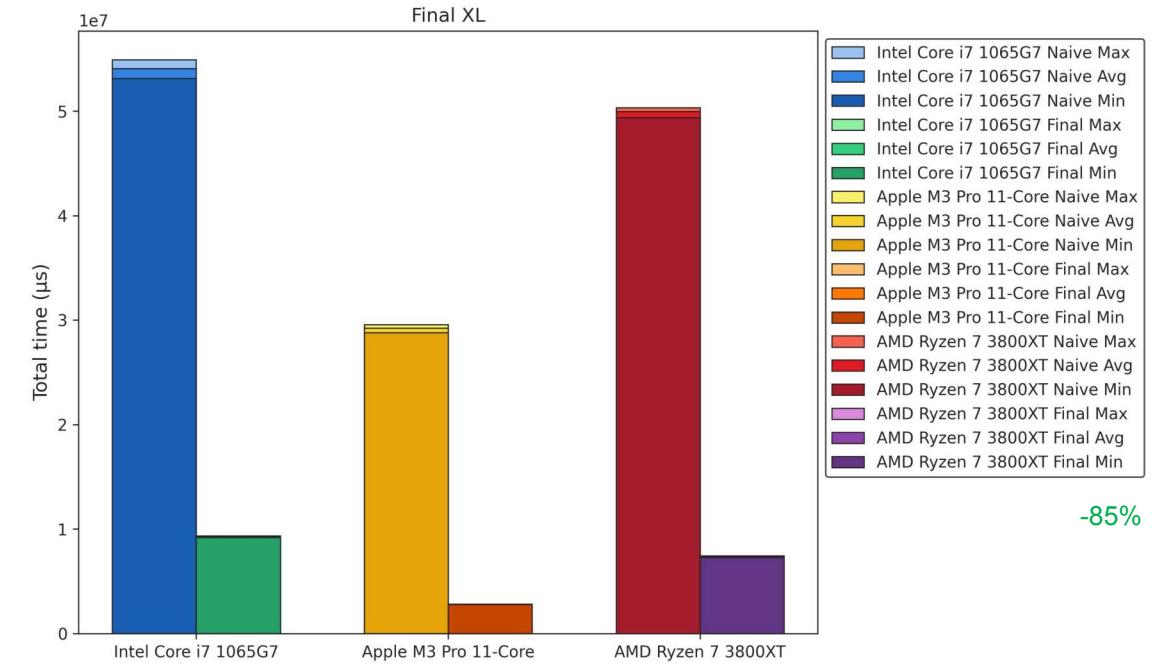
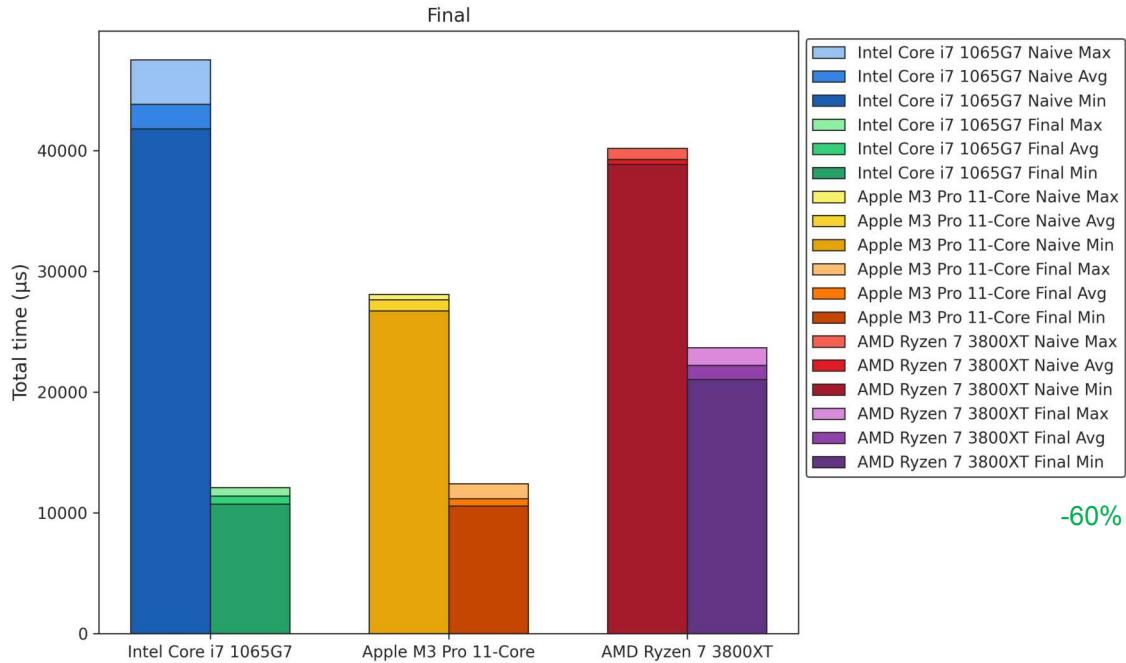
- Utilized Clang-specific and C++11 features to reduce boilerplate and enhance functionality
- Modularized parallel frameworks (NVIDIA, OpenMP, Multithreading) for easier maintenance
  - `#ifdef NVIDIA`  
`__attribute__((always_inline)) inline void run_nvidia(int i) {`

## Function Inlining

- Inlined all functions to reduce call overhead
  - `__attribute__((always_inline)) inline int get_idx(int i, int j, int y);`

# Initial vs. Final Benchmarks

# Initial vs. Final Benchmarks



# GPU-Based Optimizations

### Data Preparation

- Allocate memory on both CPU and GPU
- Copy data from CPU memory to GPU memory

### Kernel Launch

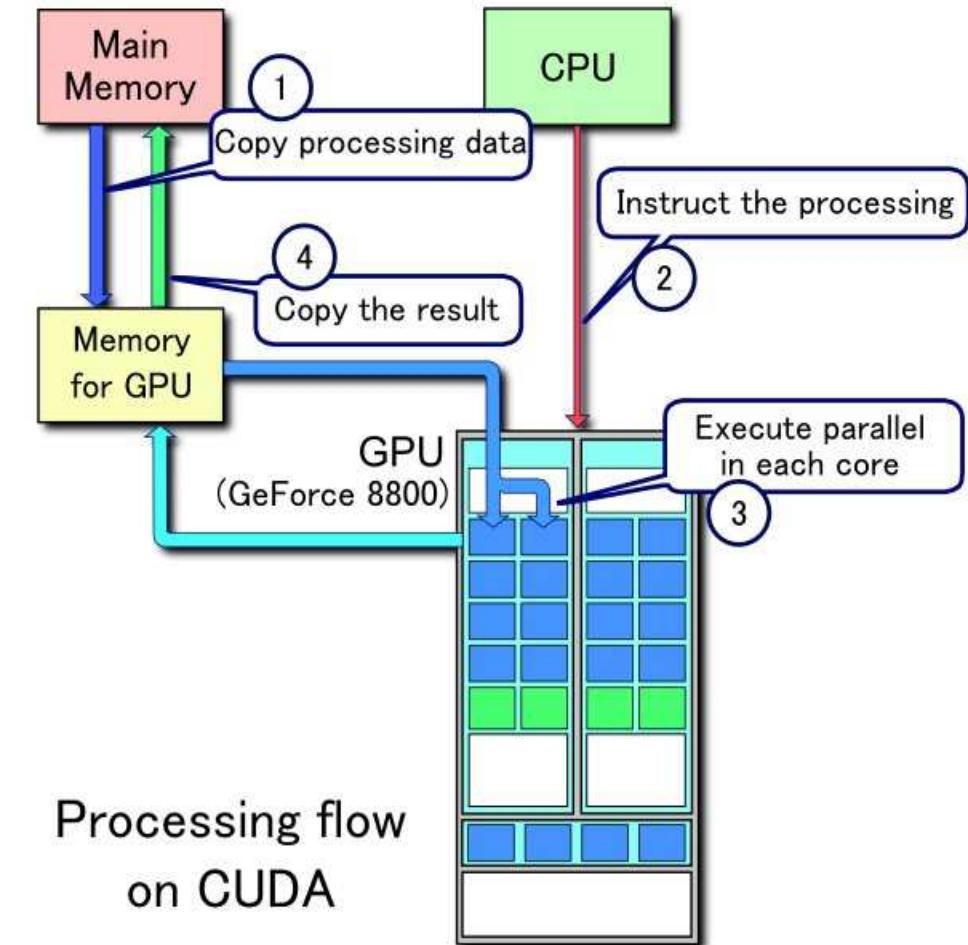
- Instruct the GPU to run a CUDA Kernel, which defines the parallel computation
- The CPU sets up the kernel parameters and sends execution commands to the GPU

### Parallel Execution

- Each GPU core executes its portion of the CUDA Kernel in parallel
- High throughput is achieved by running thousands of lightweight threads simultaneously

### Data Retrieval

- Once the GPU finishes computation, copy the results back from GPU memory to CPU memory for further processing or storage



### Definition

- CUDA Kernels are functions that execute on the GPU in parallel across many lightweight threads

### Thread Organization

- Threads are grouped into blocks, and blocks form a grid
- Each thread uses its unique IDs (threadIdx, blockIdx) to determine which part of the data it processes

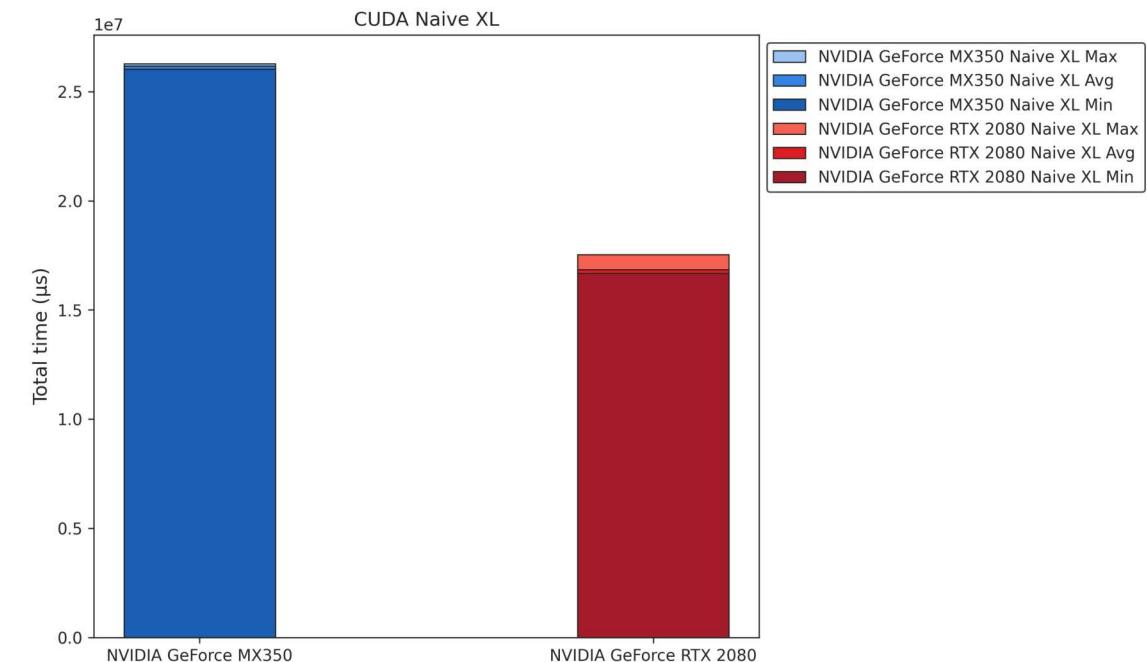
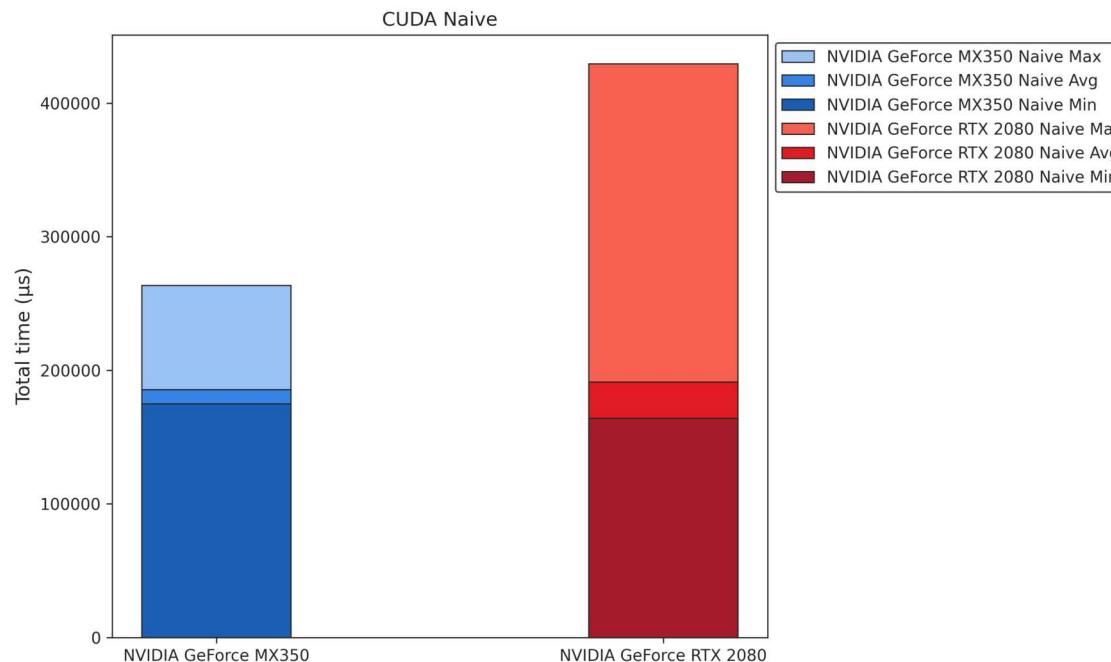
### Example Use Case: ReLU

- In this example, each thread checks one element in a matrix and sets it to 0 if it's negative
- This operation can be performed across large matrices quickly by harnessing GPU parallelism

```
__global__ void relu_kernel(DATA_TYPE *matrix, int N, int M) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    if(row < N && col < M) {  
        if(matrix[row * N + col] < 0 ){  
            matrix[row * N + col] = 0;  
        }  
    }  
}
```

# CUDA Tuning

## Benchmarks



## Reduced Memory Copy Overhead

- Moved GPU memory copies of IO-matrices outside the main loop

## Thread count

- Increased the thread count to the maximum that is available for modern GPUs (1024)

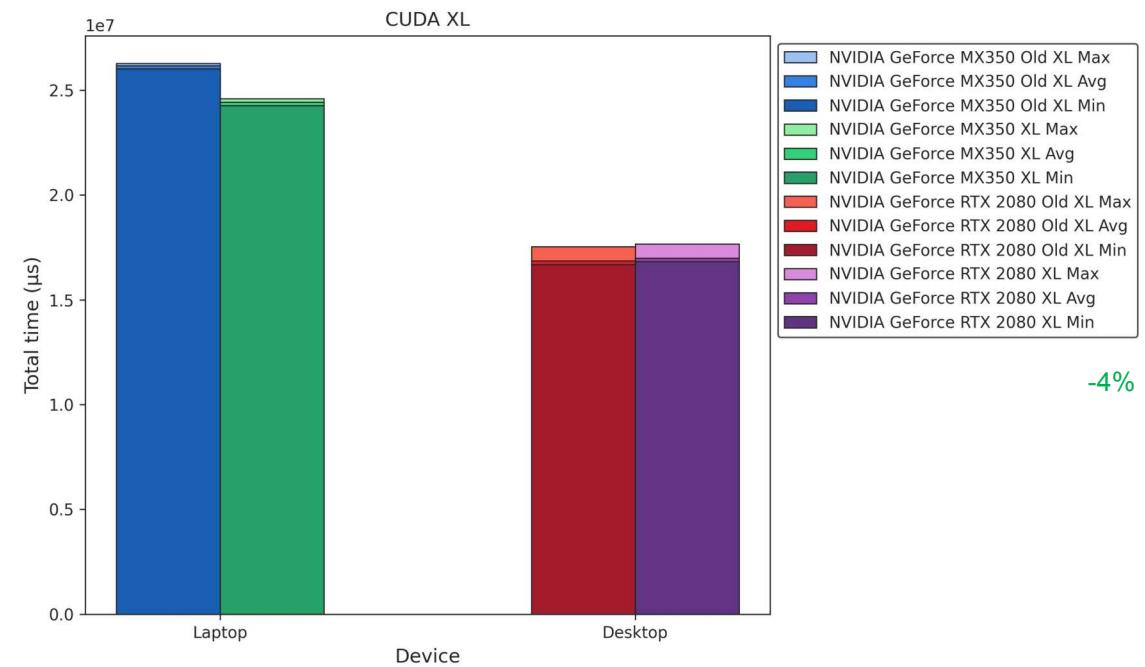
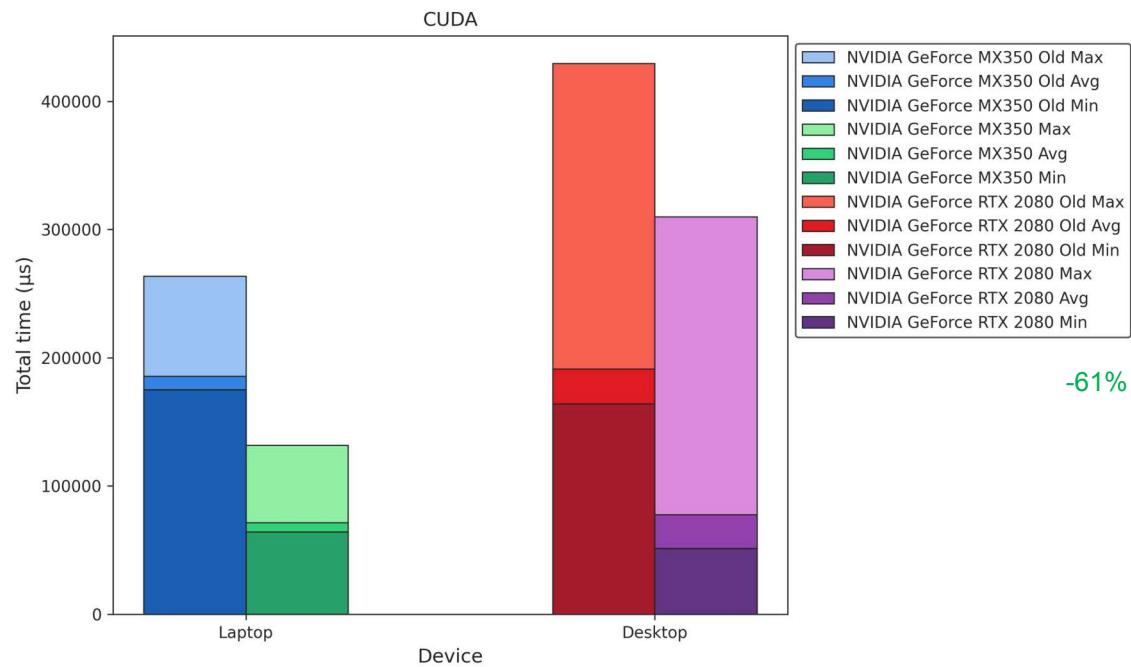
```

matrix *add(matrix *a, matrix *b, matrix *c) {
    if(c == NULL) {
        c = malloc_matrix(a->x, a->y);
    }
    // Input
    // ...
    // IO-matrix
    size_t bytes_b = K * L * sizeof(DATA_TYPE);
    DATA_TYPE *d_bias;
    cudaMalloc(&d_bias, bytes_b);
    cudaMemcpy(d_bias, b->m, bytes_b, cudaMemcpyHostToDevice);
    long threads = 32;
    dim3 block_dim(threads, threads);
    ...
    add_kernel<<<grid_dim, block_dim>>>(d_matrix, d_bias, N, M, K);
    cudaMemcpy(c->m, d_matrix, bytes_n, cudaMemcpyDeviceToHost);
    ...
}

```

# CUDA Tuning

## Benchmarks



### Eliminating Copy Overheads

- IO matrices are transferred to the GPU before computation begins
- Intermediate result matrices are pre-allocated
- Copy operations occur only at the start and end — no data is copied during computation

### Implemented Constant Memory for Convolution

- Fast Access: Data in constant memory is cached, enabling rapid access
- Efficient Bandwidth: Reduces global memory traffic, improving performance
- Consistency: Ideal for read-only data that remains constant across computations

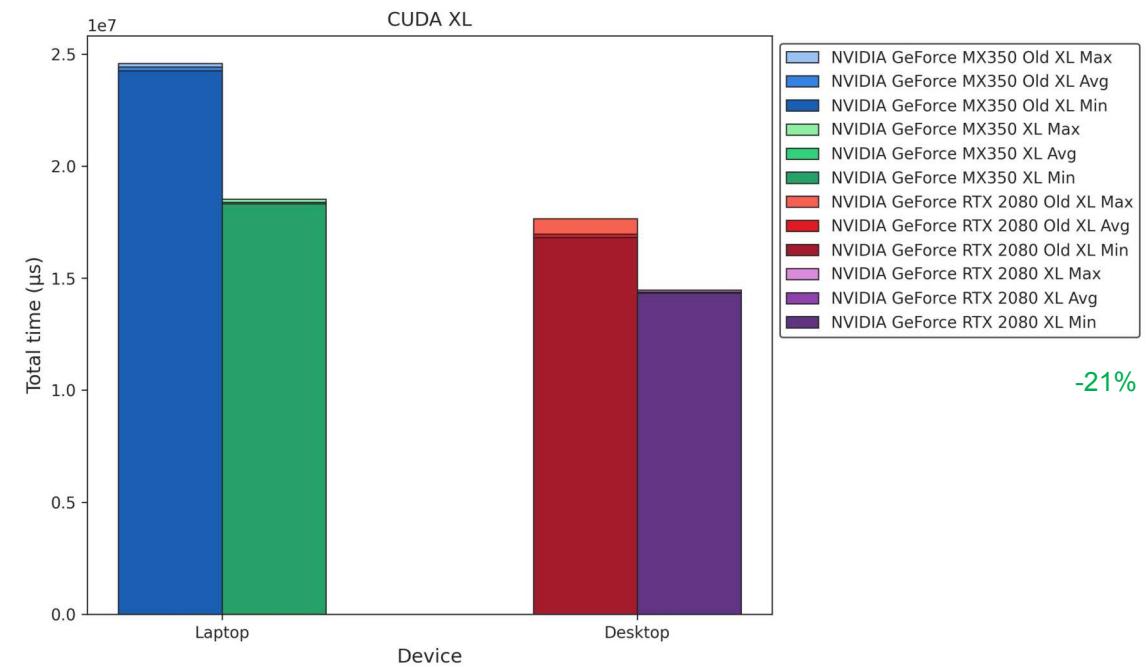
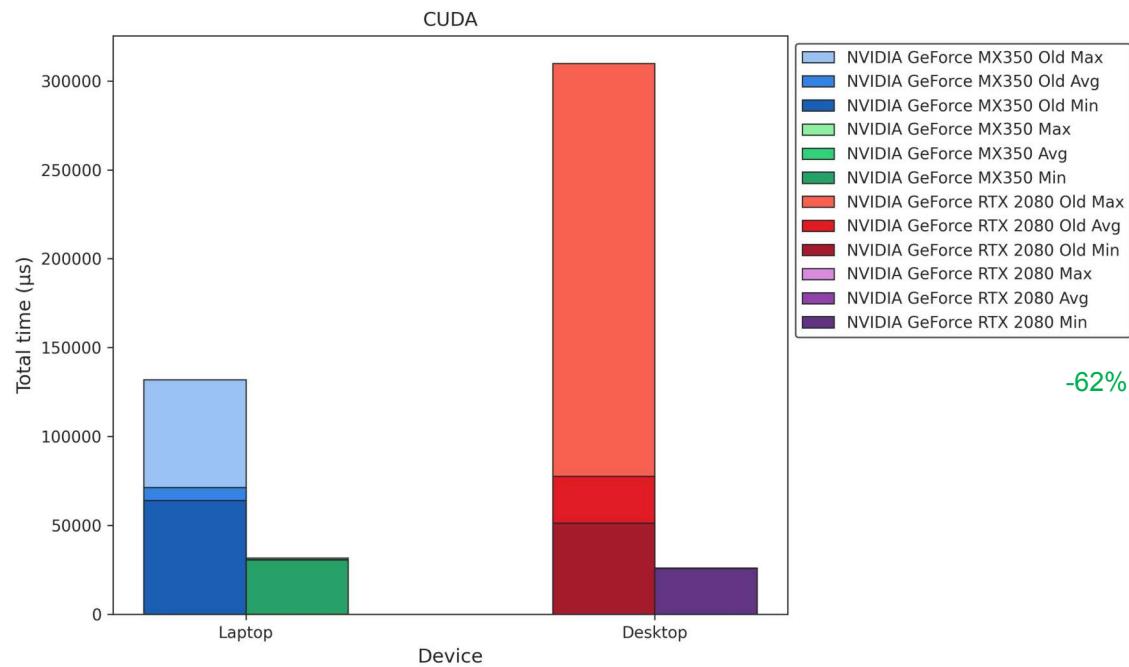
```
#define NUM_MASKS (4)
#define MASK_SIZE (3)

__constant__ DATA_TYPE c_masks[NUM_MASKS][MASK_SIZE][MASK_SIZE];

void init_const_memory(matrix **masks) {
    DATA_TYPE h_masks[NUM_MASKS][MASK_SIZE][MASK_SIZE];
    for (int i = 0; i < NUM_MASKS; i++) {
        for (int l = 0; l < MASK_SIZE; l++) {
            for (int k = 0; k < MASK_SIZE; k++) {
                h_masks[i][l][k] = masks[i]->m[l * MASK_SIZE + k];
            }
        }
    }
    cudaMemcpyToSymbol(c_masks, h_masks, sizeof(h_masks));
}
```

# CUDA Tuning

## Benchmarks

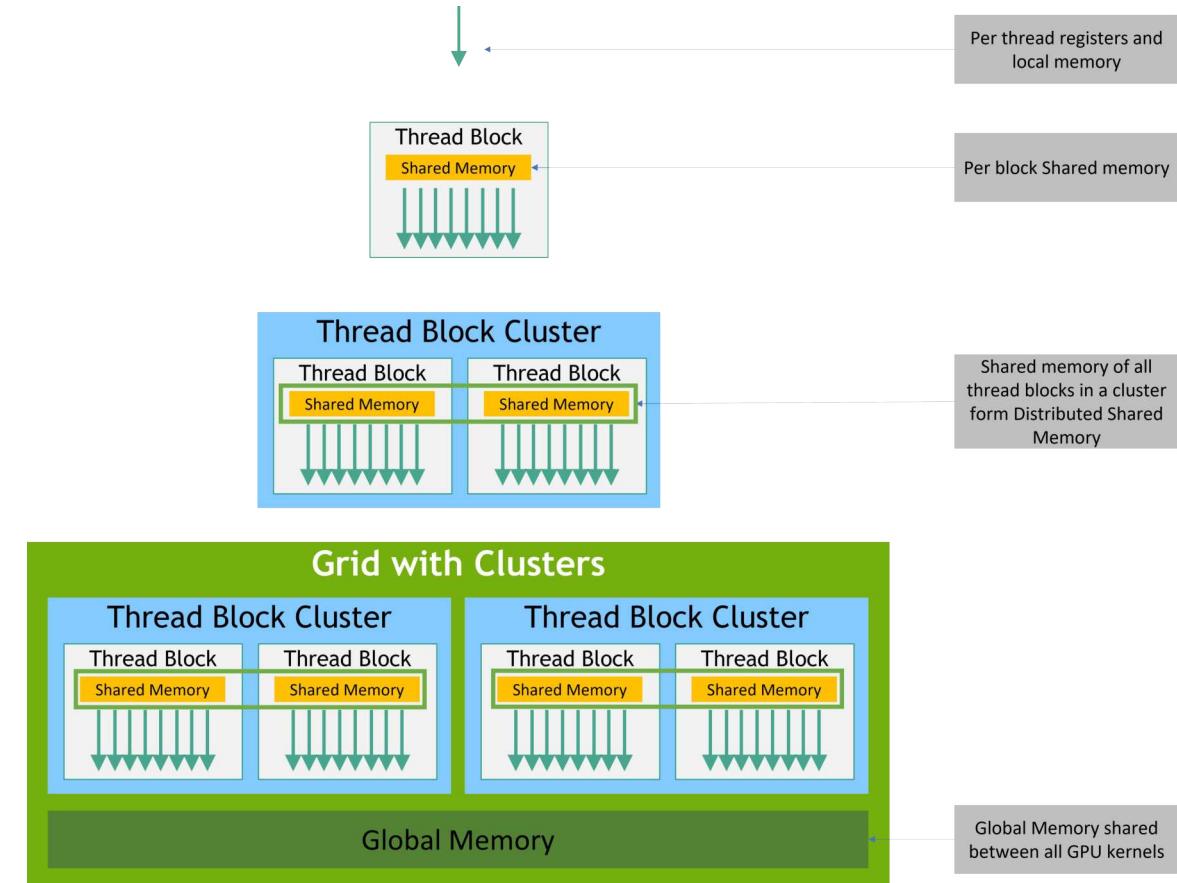


### Eliminated Transpose Overhead

- Weights are stored in a cache-optimized layout, removing the need for a transpose
- Transpose was causing a significant performance penalty

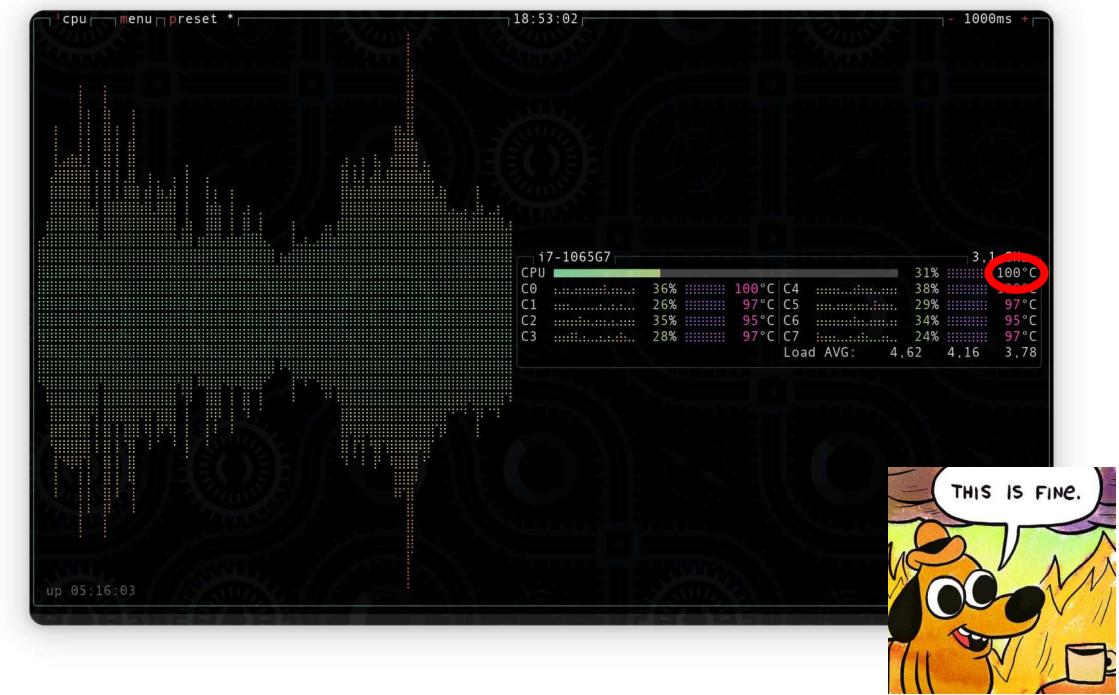
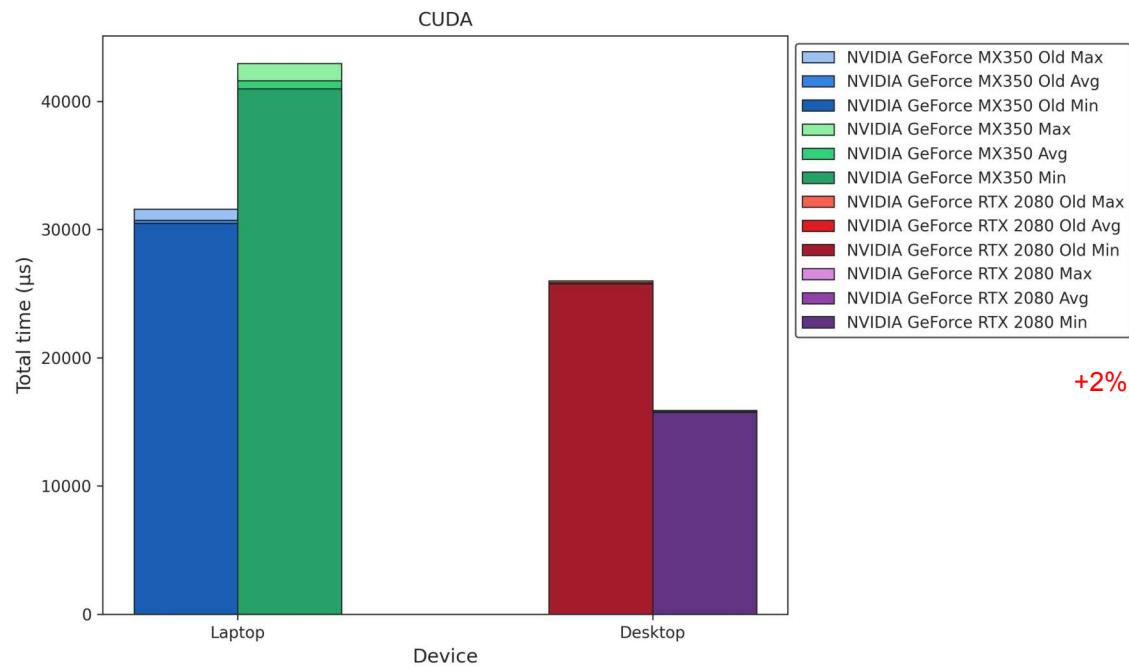
### Shared Memory for Matmul XL

- **Shared Memory**
  - A small, on-chip memory accessible by all threads in a block
  - It's much faster than global memory, making it ideal for frequently accessed data
- **Benefits**
  - Reduces global memory accesses.
  - Speeds up repeated data usage.
  - Improves performance especially for large-scale (XL) matrix multiplications.



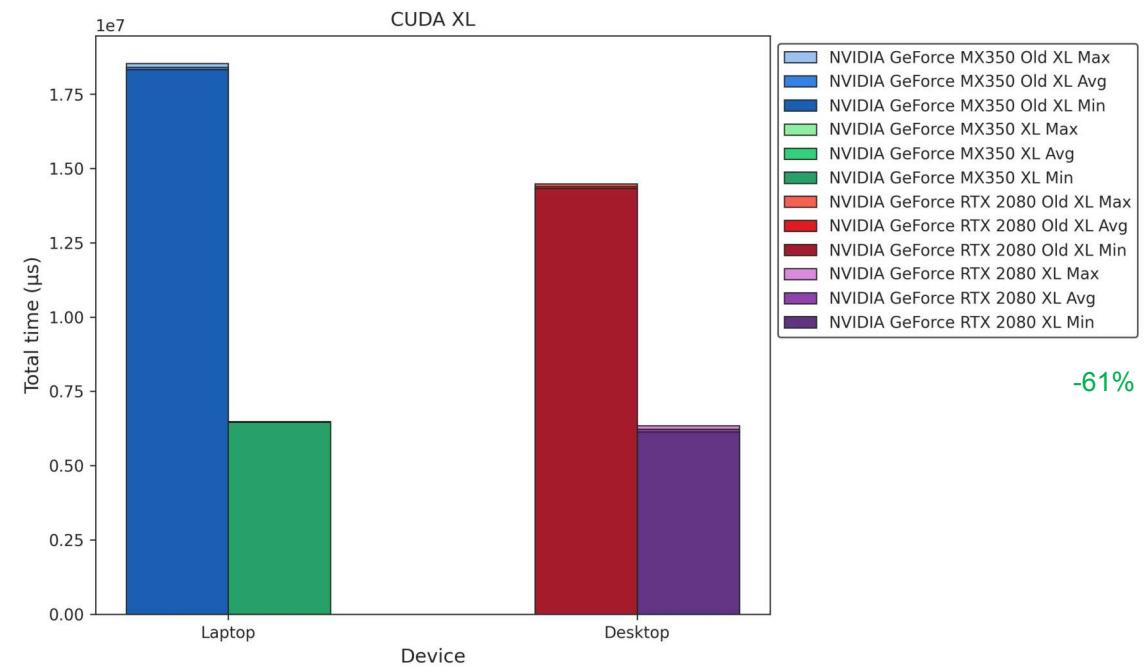
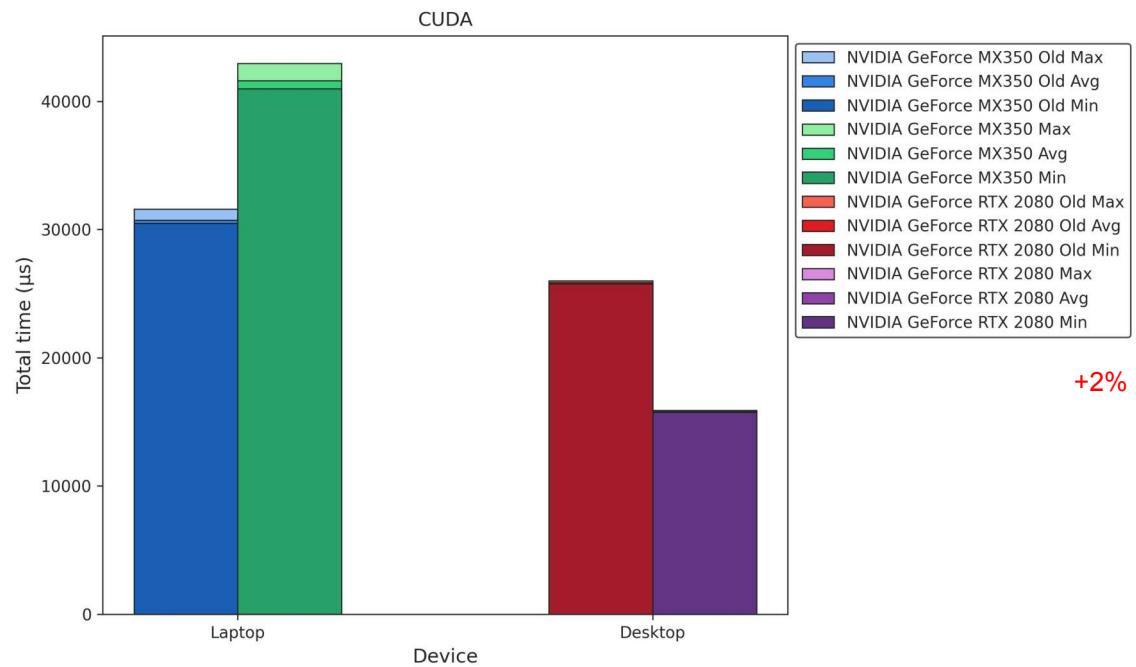
# CUDA Tuning

## Benchmarks



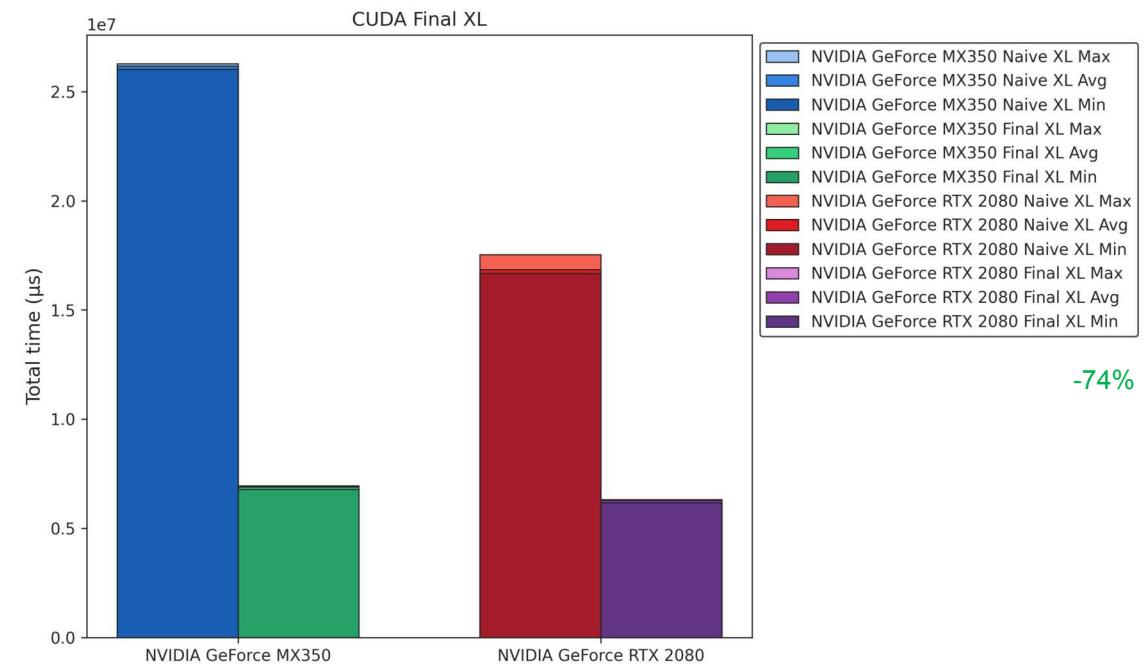
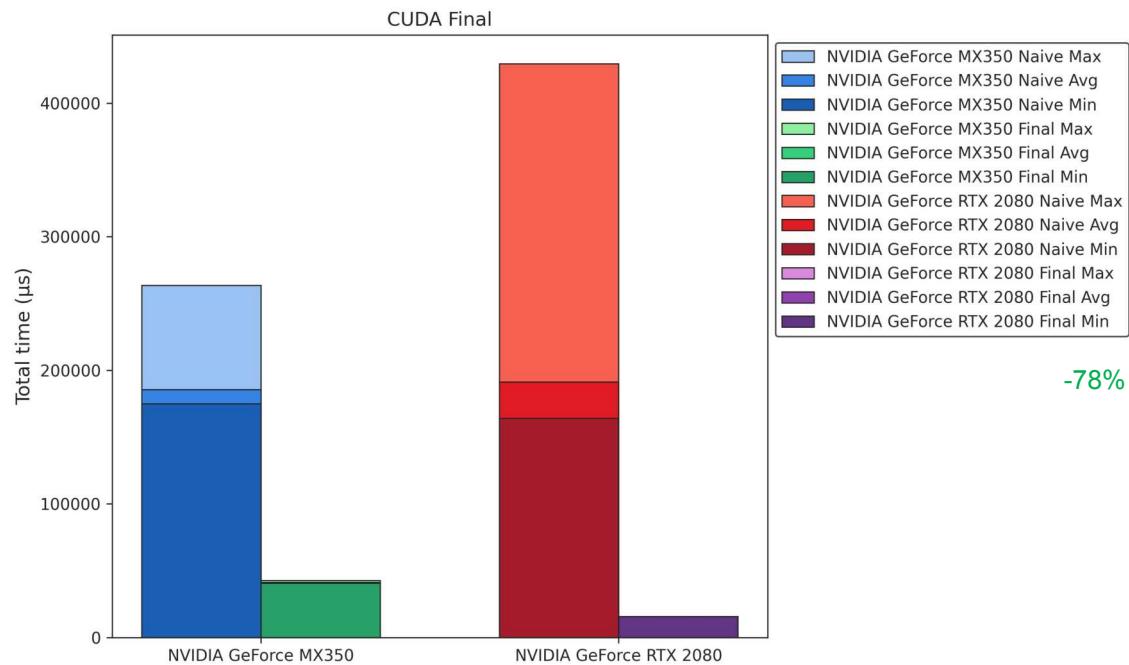
# CUDA Tuning

## Benchmarks



# CUDA Tuning

## Initial vs. Final Benchmarks



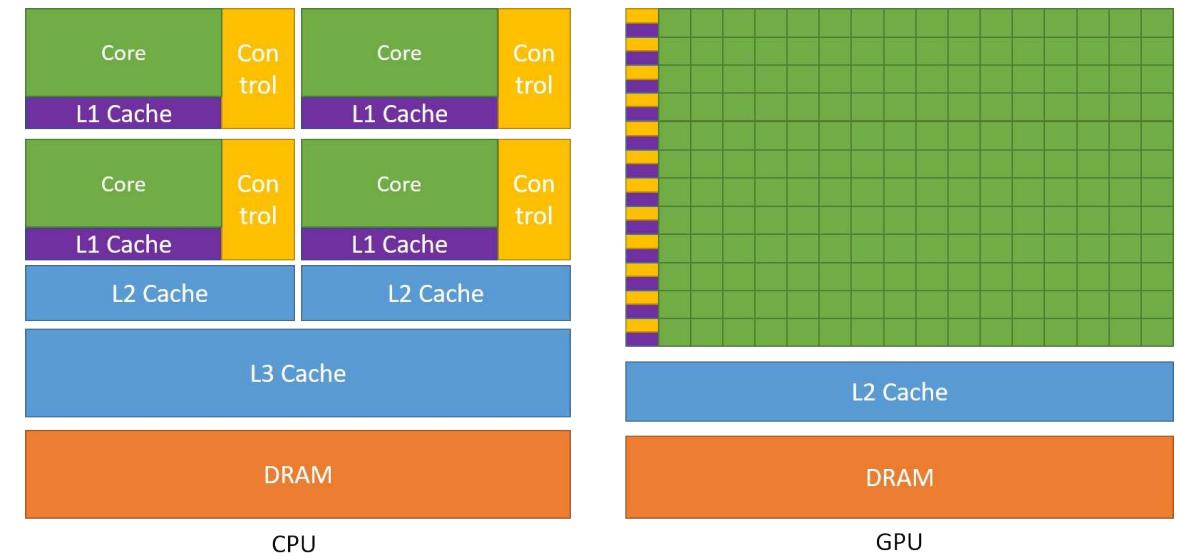
# CPU vs. GPU

## CPU

- Optimized for sequential tasks
- Many transistors dedicated to control flow and caching mechanisms
- Relies on large caches and complex flow control to handle memory latency

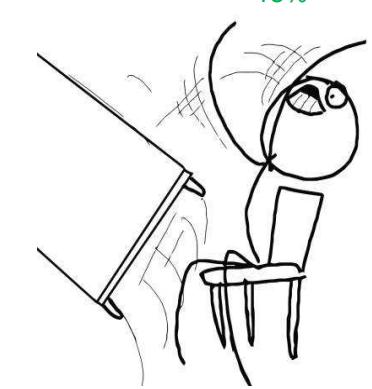
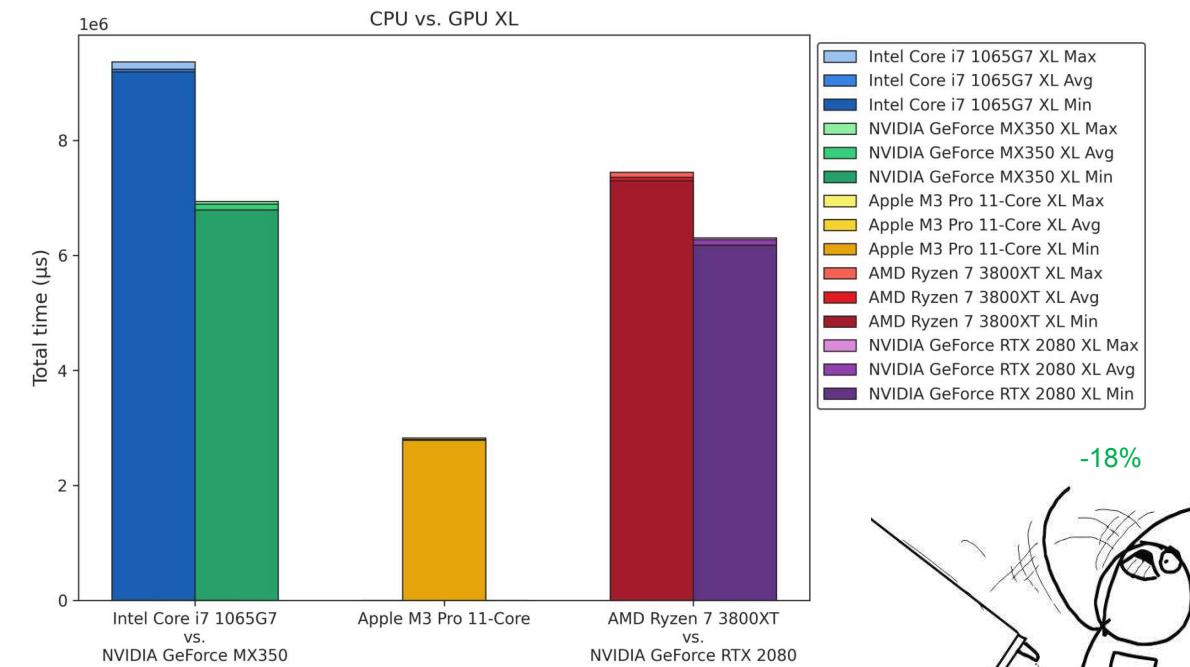
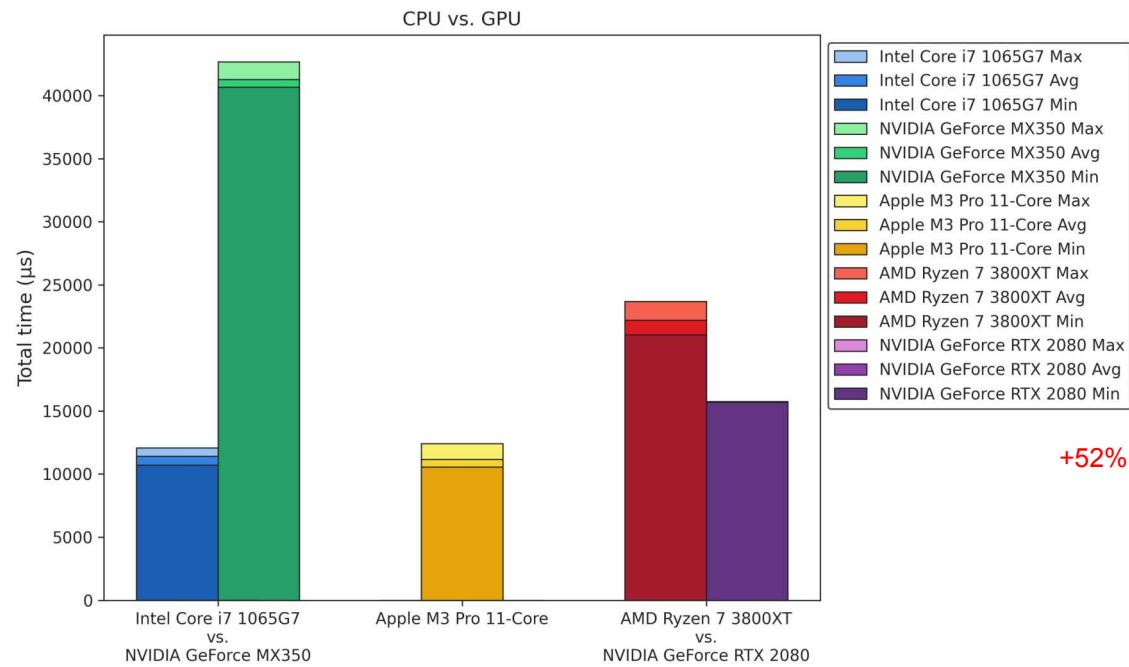
## GPU

- Optimized for parallel data processing
- Most transistors dedicated to computations
- Hides memory latency by performing more computations concurrently



# CPU vs. GPU

## Benchmarks



# Picture Credits

- p. 1: <https://www.avg.com/de/signal/cpu-vs-gpu-comparison>
- p. 9: [https://en.wikipedia.org/wiki/Multithreading\\_\(computer\\_architecture\)](https://en.wikipedia.org/wiki/Multithreading_(computer_architecture))
- p. 10: <https://pixabay.com/de/illustrations/statistiken-gesch%C3%A4ft-graph-daten-5041374/>
- p. 16, 18: <https://doku.lrz.de/openmp-shared-memory-and-device-parallelism-11481693.html>
- p. 21: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html>
- p. 21: <https://llvm.org/>
- p. 27: [https://en.wikipedia.org/wiki/Single\\_instruction,\\_multiple\\_data](https://en.wikipedia.org/wiki/Single_instruction,_multiple_data)
- p. 28: <https://documentation-service.arm.com/static/6530e5163f12c06bc0f74108>
- p. 32: <https://xkcd.com/2347/>
- p. 33: [https://en.wikipedia.org/wiki/Streaming SIMD\\_Extensions](https://en.wikipedia.org/wiki/Streaming SIMD_Extensions)
- p. 44: <https://en.wikipedia.org/wiki/CUDA>
- p. 51: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- p. 52: <https://www.nytimes.com/2016/08/06/arts/this-is-fine-meme-dog-fire.html>
- p. 56: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- p. 57: <https://imgflip.com/memegenerator/Table-Flip-Guy>