Dustin Heither, Maximilian Achenbach, and Robert Kagan

Optimizing Deep Learning Performance:

A Hybrid CPU-GPU Framework with Multithreading, SIMD,
and Evaluation of Efficiency Metrics

Outline

2. Presentation

Multithreading

Goal

Multithreading has been implemented to optimize the performance of these functions:

tf.h:

- matrix *add(matrix *a, matrix *b);
- matrix **biasing(matrix **a, int len, matrix *b);
- matrix **conv2d(matrix *a, matrix **b, int len);
- matrix *flatten(matrix **a, int len);
- matrix **flip_kernels(matrix **a, int len);
- matrix **hyperbolic_tangent(matrix **a, int len);
- matrix *matmul(matrix *a, matrix *b);
- matrix **maxpool(matrix **a, int len);
- matrix **relu(matrix **a, int len);
- matrix *transpose(matrix *a);

→

mt.h:

- void add_mt(mt_arg *mt);
- void biasing_mt(mt_arg *mt);
- void conv2d_mt(mt_arg *mt);
- void flatten_mt(mt_arg *mt);
- void flip_kernels_mt(mt_arg *mt);
- void hyperbolic_tangent_mt(mt_arg *mt);
- void matmul_mt(mt_arg *mt);
- void maxpool_mt(mt_arg *mt);
- void relu_mt(mt_arg *mt);
- void transpose_mt(mt_arg *mt);

Multithreading

Implementation

1. **We have implemented the following struct to be passed to the pthreads:**

mt.h:

```
typedef struct mt_arg {
    long idx;
    matrix **a_ptr;
    matrix *a;
    matrix **b_ptr;
    matrix *b;
    int len;
    matrix **c_ptr;
    matrix *c;
    int m;
    void (*start_routine)(struct mt_arg *mt);
} mt_arg;
```

2. **We have developed a proof of concept:**

- Each function independently created and joined its own pthreads
- The implementation was functional
- However, performance was **suboptimal**

Multithreading

Thread pool

3. **We have implemented a thread pool:**

– Thanks to the suggeestion of Philipp Holzinger :)

– **Threads are now:**

  – Created once at the start using the function:

    – `void create_mt(long threads);`

  – Joined once at the end using the function:

    – `void join_mt();`

– **Implementation:**

  – Utilizes a GAsyncQueue for task management

  – The threads execute the followoing function:

    – `static void *start_mt(void *arg);`

– **Thread behavior:**

  – The threads wait for elements to be added to the queue

  – Upon receiving an element, they execute the provided function with its associated parameters

  – This process repeates indefinetly

  – Elements can be added to the queue with:

    – `void push_mt(mt_arg *mt);`

  – `void join_mt();` adds a termination signal to the queue by pushing the function:

    – `static void stop_mt(mt_arg *mt);`

  – This ensures that all threads exit cleanly and can be joined properly

Multithreading

Challenges

- **A race condition was identified and debugging it required significant effort:**
  - To address this, a synchronization barrier was implemented using the function:
    - **void wait_mt**();
  - This implementation required an understanding of **pthread_cond_t** and **pthread_mutex_t**
- **Testing improvements:**
  - The existing dataset (MNIST) was determined to be too small for thorough testing
  - A larger test set was created by applying a scale factor of 20, resulting in image dimensions of $(30x20)^2$
- **Concurrency research and evalutaton:**
  - Various concurrency solutions were analyzed for suitability
  - **Fork:**
    - Duplicates the calling process
    - Offers low flexibility and incurs high overhead
  - **Pthreads:**
    - The current solution being used
    - Provides full control over concurrency
  - **OpenMP:**
    - A compiler-based concurrency implememtation
    - Offers lower control
    - We plan to explore its implementation in the upcoming weeks

Hardware

Overview

CPU	Release dates	TDP (W)	Number of (performance) cores	Number of threads
AMD Ryzen 7 3800XT	7. Juli 2020	105	8	16
Apple M3 Pro 11-Core	30. Oktober 2023	27	5	11
Intel Core i7 1065G7	1. Juni 2019	15	4	8

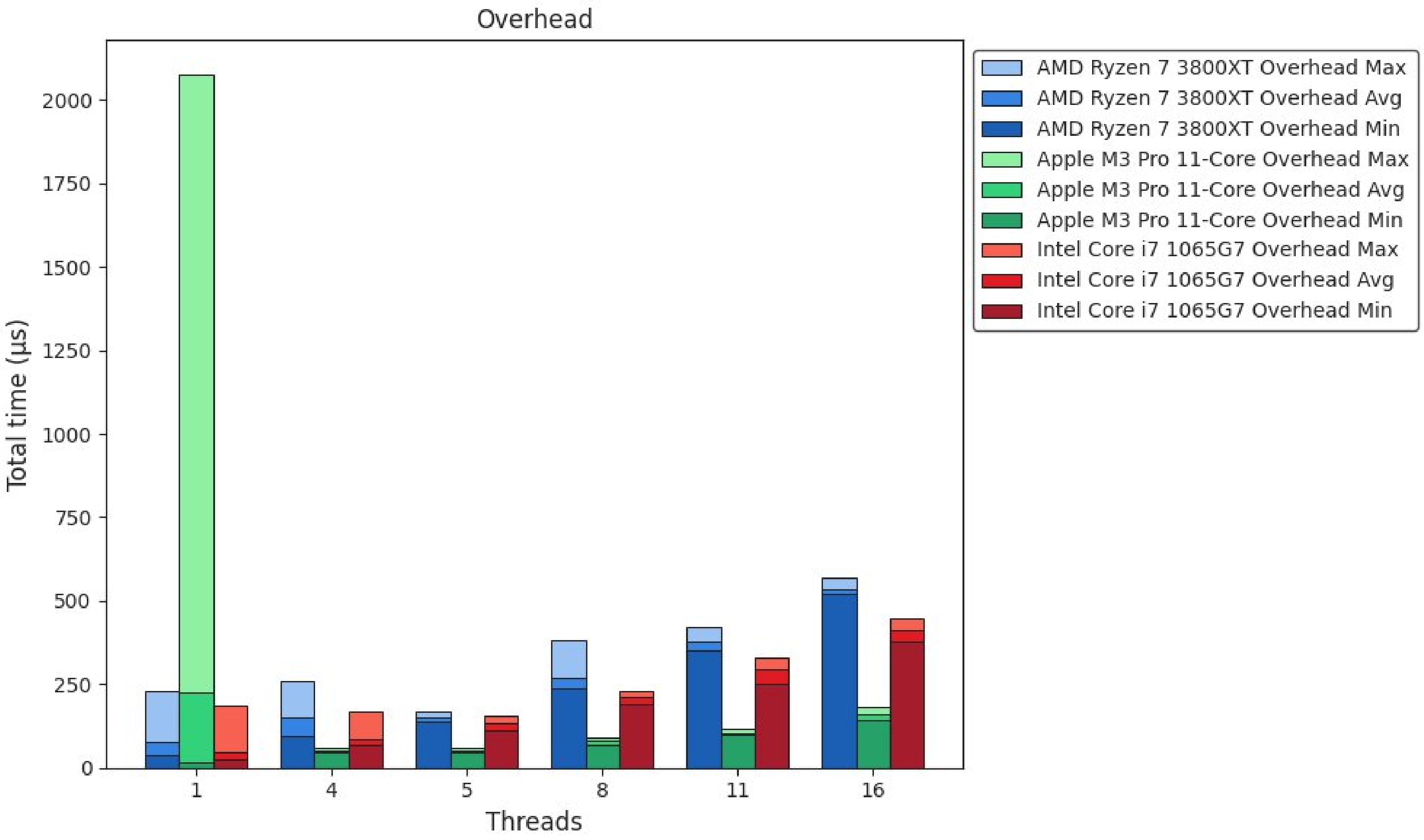
Benchmark

Overview

- **Batch size:** 1
- **Epochs:** 128
- **Overhead:**
  - In microseconds (µs)
  - Averaged over 10 runs
  - **Sum of:**
    - **void create_mt**(**long** threads);
    - **void join_mt**();
- **Performance:**
  - In microseconds (µs)
  - Averaged over 10 runs
  - Total time
- **XL:**
  - In microseconds (µs)
  - Averaged over 10 runs
  - Image dimensions of $(30x20)^2$

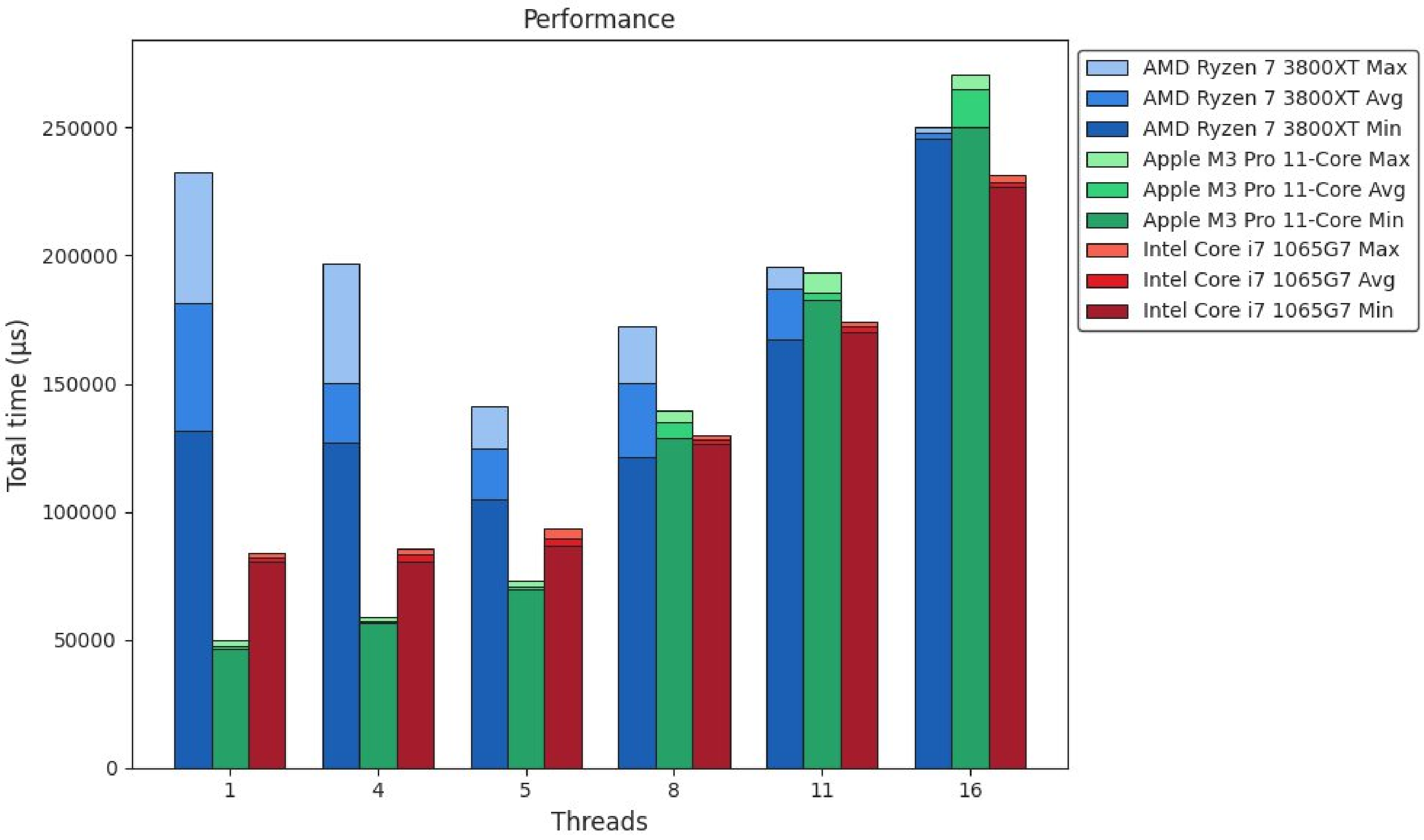
Benchmark

Overhead



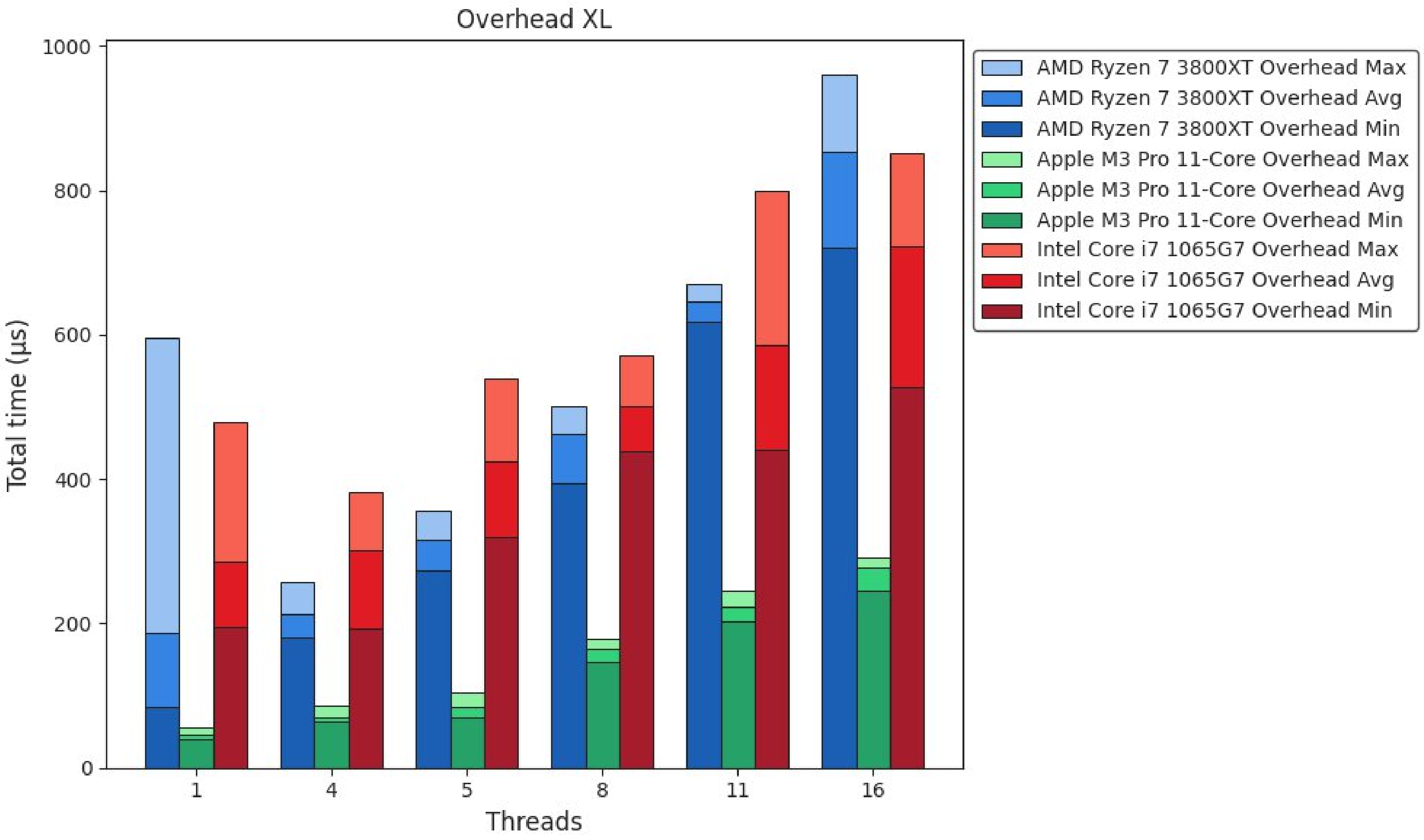
Benchmark

Performance



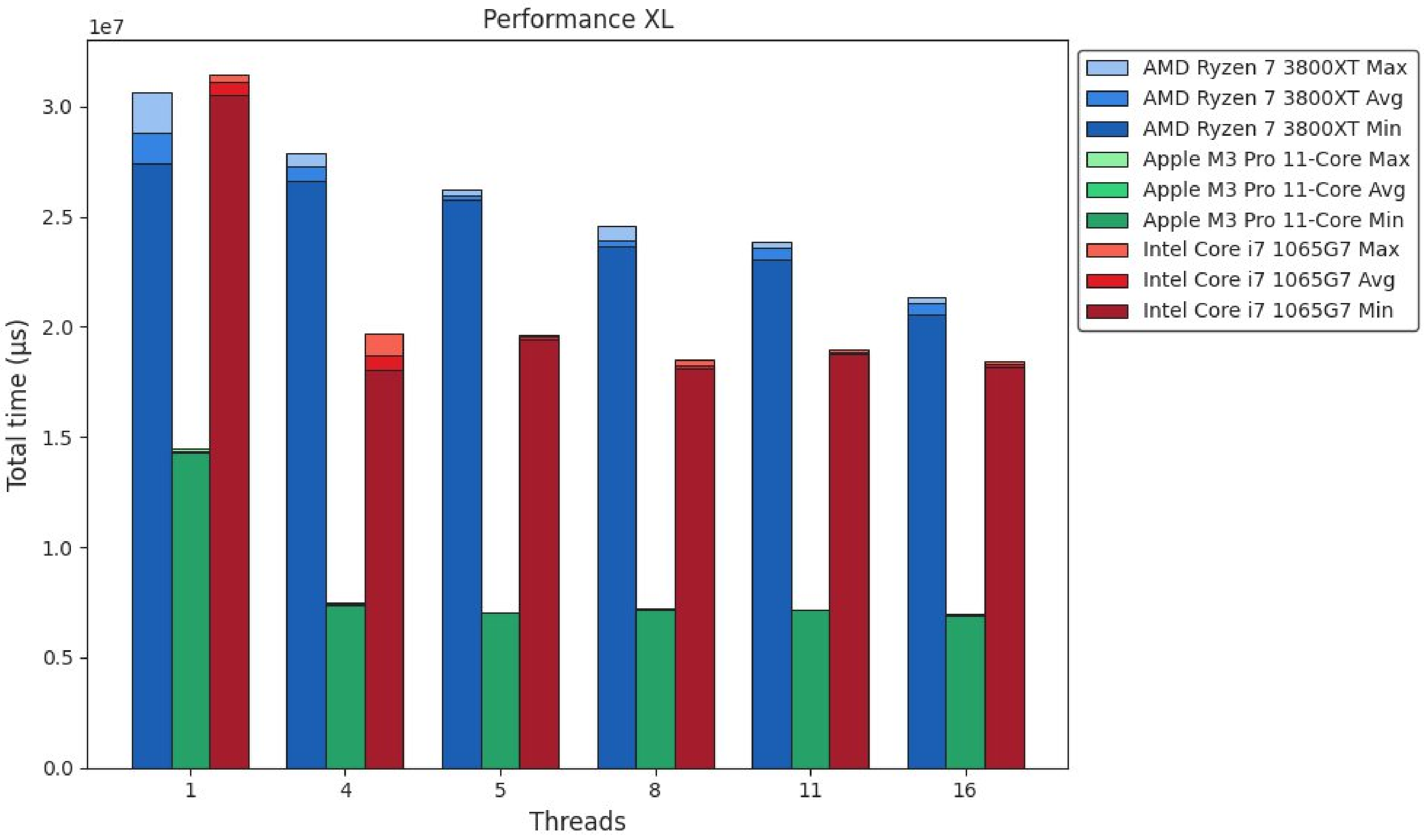
Benchmark

Overhead XL



Benchmark

Performance XL



Benchmark

ICX vs. GCC

ICX:

- Intel oneAPI DPC++/C++ Compiler
- Part of Intel oneAPI
- Formerly Intel C Compiler (ICC)
- Available for Windows and Linux
  - macOS, thus Apple, is not supported
  - → could not be benchmarked
- Supports threading via Intel oneAPI Threading Building Blocks, OpenMP, and native threads

GCC:

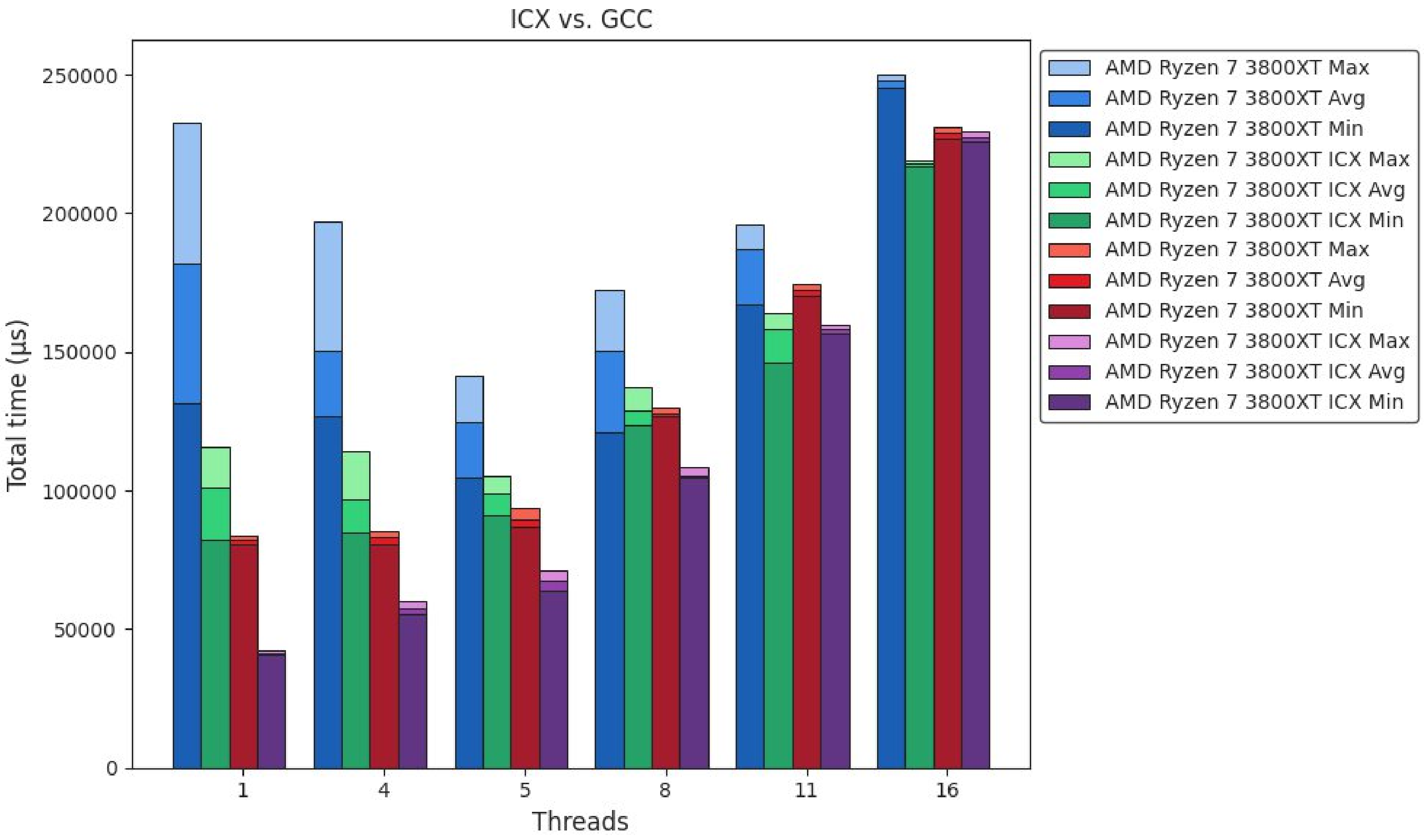
- GNU Compiler Collection
- Available for Windows, Linux, macOS, and a lot more
- Supports threading via OpenMP, native threads, and OpenACC

Both: -O3 optimization flag → turns on all optimizations

(We tested the flag for accuracy and it passed)

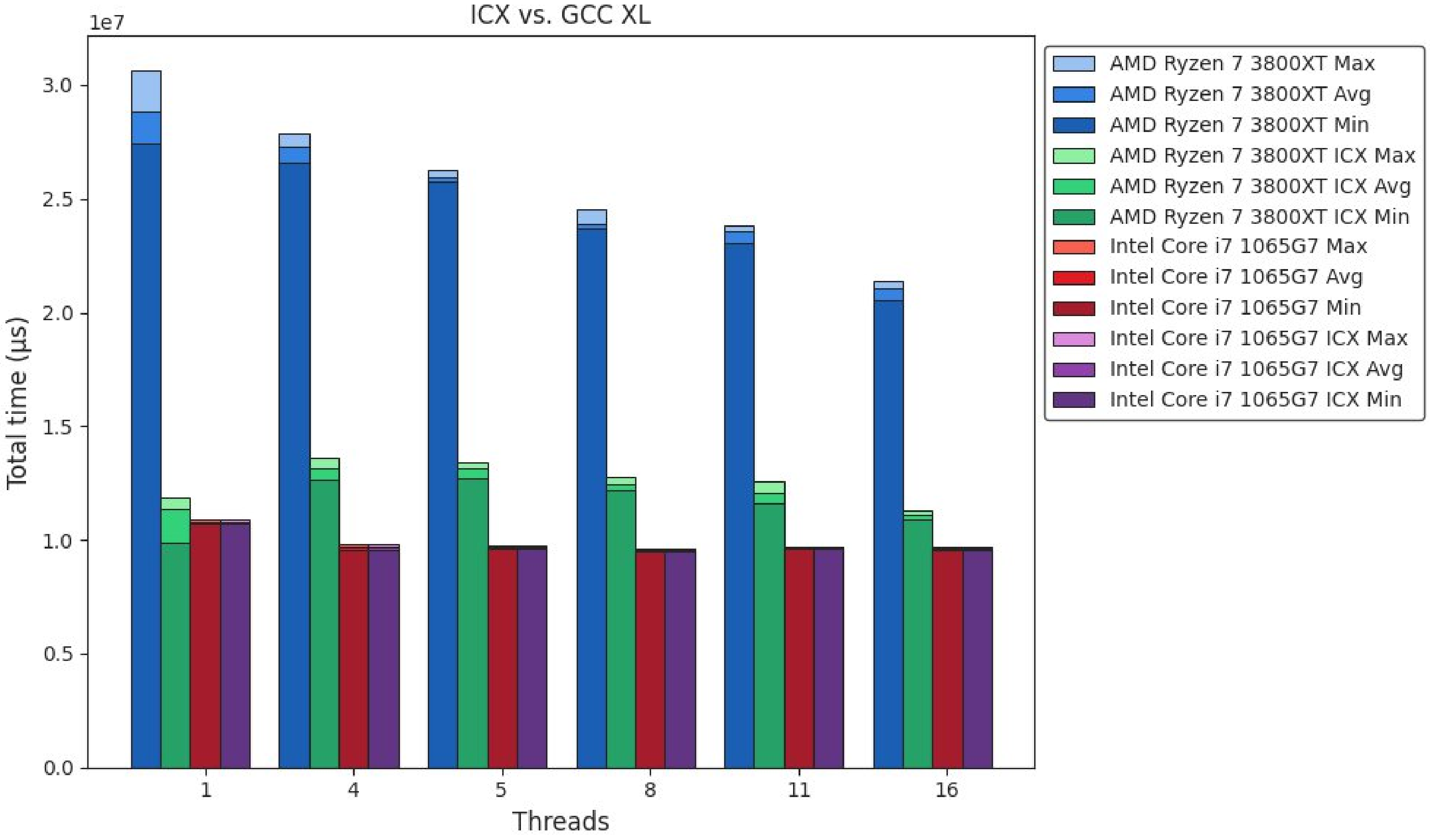
Benchmark

ICX vs. GCC



Benchmark

ICX vs. GCC XL



CUDA tuning

First steps

- Implemented matrix multiplication with CUDA in a sandboxed environment
- **First insights:**
  - For our problem size (MNIST), GPU computation takes significantly longer than CPU
    - 206.417μs vs. 2.282μs
    - Reason: Data transfer overhead between CPU and GPU memory dominates computation time
- **Difficulties encountered:**
  - Compiling CUDA C++ correctly within our framework, which is written in C
    - The extern "C" declaration in the .c files is not functioning as expected
    - Using gcc to compile the .c files
    - Using nvcc to compile the .cu files
    - Linking all components together with nvcc

Outlook

Work in progress

- Framework ✓
- Multithreading ✓
  - (Implementing OpenMP and comparing its performance to the current solution)
- ICX vs. GCC ✓
- SIMD
  - Arm Neon
  - SSE vs. AVX2 vs. AVX-512
- Quantization
- CUDA tuning
  - Matrix multiplication ✓
- (Apple M3 Pro NPU)