EGEMV on Consumer Intel SoCs

By: Ethan Steere

Supervised By: Kevin Skadron

Introduction

Advances in machine learning have produced neural networks capable of providing massive value to end users. Generative Pre-Trained Transformers (GPTs) are particularly useful, but in a naive FP32 weight format are incredibly expensive to run. On consumer hardware, FP32 inference is simply not possible except for very small, low-utility models. Neural networks, through herculean efforts (Lin et al., 2024; Badri et al., 2023, Frantar et al., 2023; Egiazarian et al., 2024; Tseng et al., 2024), can be compressed to 4 bit integers, eighth precision, with tolerable loss in quality. However, these new methods do not have well optimized implementations like BLIS, OpenBLAS, and Intel oneMKL provide for FP32 (Zee et al., 2015; OpenMathLib). This investigation will look at different strategies for mapping quantized matrix-vector multiplies to CPU SIMD and integrated graphics on Intel SoCs.

Motivation

Adoption of digital technology has been a catastrophe for privacy (Greenwald, 2013). Legal protections, such as the 4th amendment, have so far failed to mitigate this (Volz, 2024). Only architectural defenses built into consumer technology that make surveillance impossible will reverse this ghastly phenomenon. Multi tenant networked devices are not securable against nation state adversaries (Larin, 2023). Any *real* improvement in privacy is a feature complete replacement with no networking at all. LLMs are the biggest privacy innovation since PGP because they can power a device for offline and un-surveillable Q&A. Such a device needs an inference library that can produce reasonable quality generations at reasonable speed on a low

power (Laptop Grade) CPU-SoC. This paper tries to improve the state of art, GGML, to bring this theoretical device closer to reality.

Background

Issues With SGEMV For Efficient NN Inference

Single Precision General Matrix Vector Multiply (SGEMV) performs a matrix vector multiplication on 2 FP32 arrays. There are many fantastic implementations like BLIS, OpenBLAS, CLBlast (integrated graphics), and the proprietary Intel Math Kernel Library, now part of Intel oneAPI. It would be ideal if these libraries could be used to inference neural networks efficiently.

Of course, I tried that first. Andrej Karpathy wrote a library, llama2.c, that can inference small llama GPTs. Built for simplicity and instruction, `llama2.c` uses a naive c SGEMV implementation relying on compiler auto vectorization and OpenMP for speed. Clearly one of the highly optimized libraries should be able to improve performance. Shockingly, use of MKL slowed down token generation by 25%. Experimental details and data are in Appendix A.

This seems like a puzzling result, but makes sense when memory bandwidth is considered. All DRAM systems can only transfer a finite amount of data to the chip per unit time. Intel VTune benchmarked my 16GB of DDR4 at 50 GB/s. Running MKL GEMM showed 200 GFLOPS was possible in practice, although the theoretical FLOP throughput is 300 GFLOPS[1].

An SGEMV invocation `(m, n) @ (n,)` will require `m * n * 4` bytes from memory and `m * n * 2` FLOPS[2]. The runtime can be estimated with

[1] 16 FLOPS/CORE/CYCLE * 4 cores * 4.7 GHz ≈ 300 GFLOPS
[2] I don't include the input in the loads figure, since this is small compared to the matrix and will remain in cache. Otherwise there are `m * n * 4 + n * 4` byte loads.

```
min(bandwidth_in_bytes / 4mn, max_flops / 2mn)
```

SGEMV is compute bound when …

```
max_flops = bandwidth_in_bytes / 2
```

Memory bandwidth must be twice the available FLOPS in order to achieve full utilization of available compute. In order for FP32 to get 200 GFLOPS of throughput I'd need 400 GB/s, 8x more than I have. With 50GB/s, my system is limited to 25 GFLOPS of SGEMV throughput. No amount of optimization magic from Math Kernel Library can mitigate this. Thus, a naive implementation is able to match it closely.

Quantization and EGEMV

Slow inference is unacceptable, so we need to either reduce the number of byte loads, or reduce the number of floating point operations. I think that both vectors of attack will need to be pushed to the limit to match GPU powered websites like chat.openai.com. This paper focuses on reducing the number of byte loads with Quantization.

Quantization is the process of representing high precision values in a compressed integer format. This paper targets a quantization output where weights are 4 bits each with size 128 groups along output channels sharing a scale (FP32) and a zero value (4 bits). Because 4 bits is one eighth of 32 bits (single precision), I call this routine Eighth Precision General Matrix Vector Multiply, *EGEMV*. Technically, this produces 4.28125[3] bits per weight, but is the best that the field can deliver, for now.

The process of determining this representation with minimal computational effort and with minimal impact on quality is its own field. GPTQ (Franter et al., 2023), AWQ (Lin et al.,

[3] `(128 * 4 + 4 + 32)/128 = 4.28125`

2024), and HQQ (Badri et al., 2023) all produce this output, and I'm hopeful that even faster and higher quality techniques will come in the future. There are other quantization methods that achieve <4 bits per weight through the use of codebooks (Tseng et al., 2024; Egiazarian et al., 2024), but do not lower memory traffic due to the need to reload codewords multiple times for different channels. In the event that grouped output channel quantization becomes obsolete, high level lessons of this paper may be useful for writing NN ops for those new techniques.

Equation 0: Unquantized Multiply Accumulate

```
out(F32) += weight(F32)*input(F32)
```

Equation 1: Quantized Multiply Accumulate

```
out(F32) += (weight(u4)-zero(u4))(F32)*scale(F32)*input(F32)
```

*zero and scale are shared among 128 weight values

Activation Quantization vs. Weight Quantization

As the name indicates, Activation-Aware *Weight* Quantization converts weights from `FP32` to `uint4`. Activations, as shown in equations 0 and 1, are still in `FP32`. However, these activations can also be held in a compressed format. The reason for this is not for conserving memory bandwidth, although that's an added benefit, but instead for the ability to use integer math in registers. Rather than each matrix vector multiply taking FP32 input and producing FP32 output to be passed to the next layer, you instead quantize the FP32 output values on the fly to i8 before passing to the next layer. This is referred to as W4A8 as opposed to the above W4A32.

Equation 3: **W4A8 Multiply Accumulate**

```
out(F32) += ((weight(u4)-zero(u4))(i8)*in(i8))*
scale(F32)*in_scale(F32)
```

*zero and scale are shared among 128 weight values

*in_scale is shared along 128 input values

Memory Layout

We have 3 matrices for every input FP32 weight matrix.

1. Weights: `unsigned int4`
2. Zeros: `unsigned int4`, (Group Size: 128 Weight Values)
3. Scales: `FP32`[4] , (Group Size: 128 Weight Values)

Ordering of values is important for achieving optimal performance. Unlike engineers working on Netlib compliant BLAS libraries, I have control over the layout of the input matrices. This control can be leveraged to ensure that no unnecessary loads take place and that any cache lines loaded from DRAM are fully used before eviction. This memory layout should allow for an optimal implementation on both the CPU and iGPU.

Figure 1 shows a fully row major representation of the 3 matrices. In order to use all loaded 64 byte cache lines, the routine must process 128 zero values per row and (128*128) 16384 weight values per row before moving on to the next row. Most input matrices from small language models are not this large.[5]

[4] AWQ allows for FP16. FP32 is more convenient due to native support in C++.
[5] Llama FFN `down_proj` is (4096, 14336).

Figure 1: Row Major Layout. All values are listed left to right in memory.

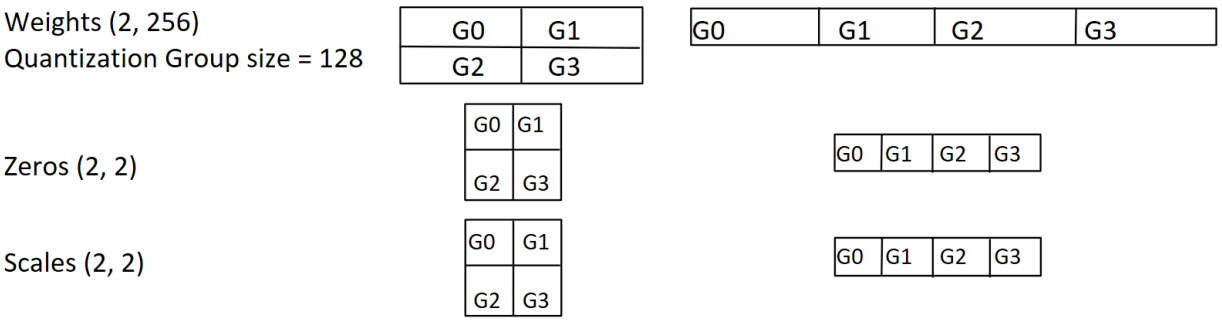


Figure 2 shows an array of structs representation. There is only one array; The scale and zero value are inlined with the weights. This is ideal for implementation simplicity and correctness. However, there is no way to arrange this such that the byte width of the struct is divisible by 64. By using this representation, we are resigned to load-and-abandon because we will always load cache lines with no intention of use. GGML, the most popular consumer device inference library, uses an array of structs representation for all of its quantization methods. Benchmarks against GGML are shown later. (It works pretty well).

Figure 2: Array of Structs. Group Scale and Zero are inlined

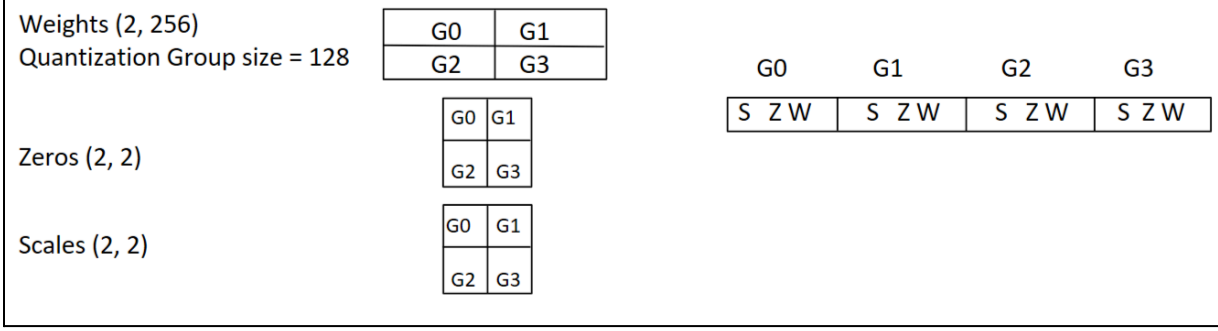


Figure 3 shows a column major representation of the 3 matrices. In order to use all loaded 64 byte cache lines, one simply has to process 128 weight values per row for 128 rows. This allows for far more flexibility in the weights partitioning for processing on heterogeneous devices. The issue with this approach is that the weights are not able to take advantage of AVX512-VNNI instructions for `int8` multiply-accumulate. The weights are contiguous top to bottom and thus a row-wise access is an expensive gather-scatter access.

Figure 3: Column Major Layout. All values are listed top to bottom in memory

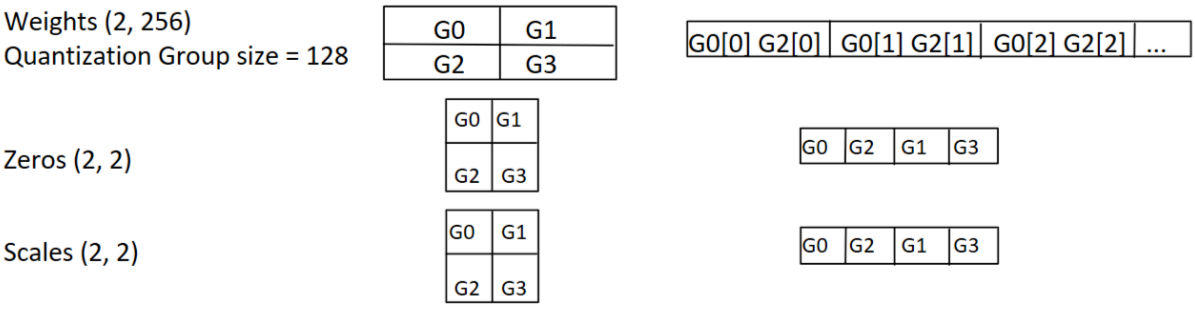
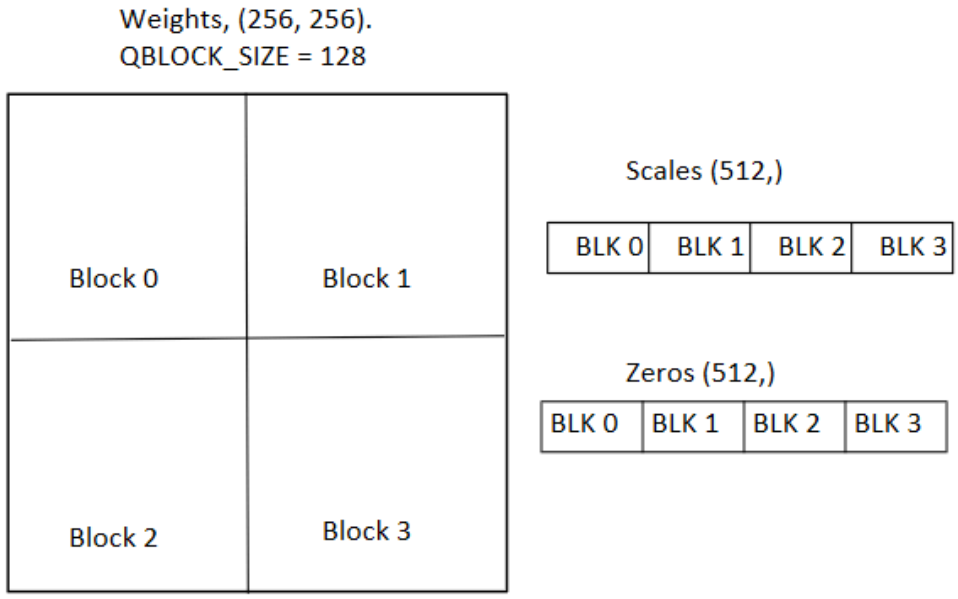


Figure 4 shows the final memory layout, Row Major Block Contiguous, that I used in the final EGEMV functions. It orders the weights row major for fast row-wise accesses. For implementation simplicity, It treats the scales and zeros as arrays rather than matrices. Every 2D quantization block (128, 128) (one cache line/64 bytes in each direction) has a `block_id` assigned left to right. By offsetting the global array pointer for scales and zeros by `block_id * 128` the values for each row group of that block are contiguous. One can iterate through these 128x128 blocks in any order and be sure to use any loaded cache lines from memory.

Figure 4: Row Major Weights, Block Contiguous Scales and Zeros.



SIMD-Aware Weight Interleaving

AWQ (Lin et al., 2024) suggested that weights be interleaved to avoid a shuffle operation. AVX512 has a hard time shuffling bits across 128 bit lanes, so it's advantageous to avoid it somehow. I do not want the AWQ suggestion of reordering weights, because it would require different models for different accelerators. If one has SIMD-aware weights then the iGPU, should it be advantageous to use, would have to reorder them manually at high cost. Rather than reorder the weights, the activations can be reordered without cost during quantization. The weights on disk stay in canonical order, and instead the inputs are reordered to match the order resulting from the AND and shift operations for unpacking.

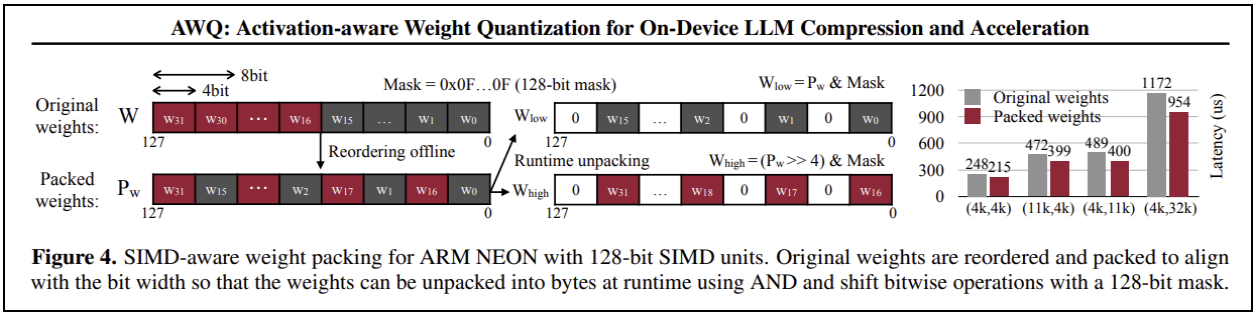
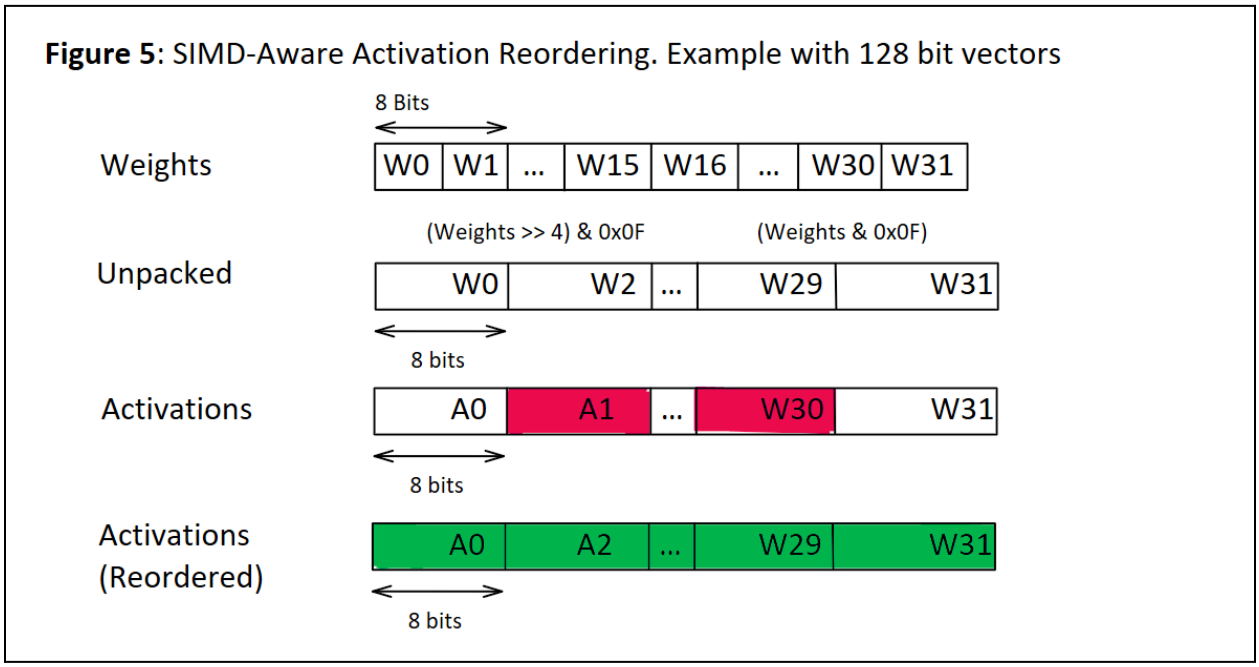


Figure 4. SIMD-aware weight packing for ARM NEON with 128-bit SIMD units. Original weights are reordered and packed to align with the bit width so that the weights can be unpacked into bytes at runtime using AND and shift bitwise operations with a 128-bit mask.

From the AWQ Paper, Pg. 6.



EGEMV on the CPU

On the CPU one wants to use SIMD units to get optimal performance. I chose to use `<immintrin.h>` in C++ to write the function.

Objectives:

1. Input Vector Reuse: The input vector working should stay in the L1 Cache.

2. Weight Cache Efficiency: Similar to input, anything loaded from memory should be used to avoid reloading.
3. Zero Cache Efficiency: No Load-And-Abandon.
4. Scale Cache Efficiency: No Load-And-Abandon.
5. Pipeline Depth: If two many registers are declared, then the processor won't be able to leverage instruction level parallelism. Worse, the program may dump data to the stack to increase instruction level parallelism artificially.
6. Minimize Floating Point Math: FP math is slow and increases energy usage causing throttling. Doing 16 FLOPs in one AVX512 instruction is more efficient than 16 sequential FLOPs, just faster.
7. Minimize write traffic to DRAM for intermediate results.

To minimize DRAM write traffic for intermediate results, I want to iterate as long as possible to amortize the cost over as many values as possible. Iterate too long, and the input may not be able to stay in the cache forcing expensive reloads from DRAM. This introduces a tunable parameter `ITER_LENGTH`. It may be advantageous to process `N_ROWS` per iteration along the input activation. Higher `N_ROWS` limits the value of `ITER_LENGTH`, but also cuts the number of loads between L1 and registers for input values by a factor of `N_ROWS`. `N_ROWS` accumulation registers are needed for each row. The right balance is unclear, and unfortunately I was not able to do a scientifically sound parameter sweep. I chose the `N_ROWS=1` and `ITER_LENGTH=16384`.

EGEMV on the iGPU: Contributed from my CS6501 Course Project

The tradeoffs for the iGPU are similar to the CPU. I chose to use OpenCL kernels compiled at runtime from source code in an OpenCL string.

Objectives

1. Input Vector Reuse: The input vector should stay on-chip.
2. Weights, Scales, and Zeros should not be loaded twice.

Trade Offs:

1. Register Pressure: More parallelism hides memory latency, but a high number of in-flight requests may trample each other in cache.
2. Atomic Memory Accesses: Atomic memory actions are very expensive, but can offer cooperation between work groups (Thread Blocks in CUDA).
3. Explicit/Implicit vectorization: OpenCL provides many vector types and associated operations that could be used to ensure that the shift operations are being executed efficiently. However, this does not give the scheduler as much flexibility which have performance consequences.

I based my investigation on a similar investigation called *FastGEMV* (Wang) which focused on Nvidia GPUs and was written in CUDA. *FastGEMV* states that a GPU GEMV implementation consists of two high level steps: (1) accumulating partial results per thread, and (2) accumulating these partial results into the output vector. A hidden piece (3) that's important, specifically for Iris Pro Graphics, is the number of rows and columns given to each thread. This was not explored in depth in the *FastGEMV* writeup.

Accumulation of the partial results inside each thread is simple, but the extent to which the operations are vectorized explicitly was up to me. Although it was unthinkable coming from AVX I decided to not to use vector types for anything except loading values, hoping the scheduler would optimally map the expressed math to available SIMD units. The *FastGEMV* code did not use any vector types in CUDA for mathematical operations.

Inter-Thread accumulation of generated partial results was a key concern for *FastGEMV*, but for this investigation it was not. *FastGEMV* used warp reduction and then a shared memory reduction if needed. Each work group (CUDA Thread Block) would do an entire row to avoid atomic operations completely, thus I did not attempt to use them. Thread partitioning constraints prevented me from using sub group reductions (CUDA warp reductions).

My device was only able to have work groups of size 256 and total work items (`256, 256, 256`). It was able to have `16777216` total work items, but in each dimension, the value could not exceed 256. This made the thread partitioning very confusing and led me to select a retrospectively suboptimal partitioning scheme. The first dimension would always have 2 work items, representing two halves of the row to be processed in parallel. The second dimension was always 64 representing 64 2 row blocks in each quantization block. The third dimension selects the output block. This introduces decent parallelism while staying under the 128 threads per work group limit. The limit of 256 in each dimension is respected.

Due to the fact that each work group is only dividing rows in half, the accumulation of partial results is not difficult in shared memory. No parallel reduction is necessary. If I had more time, I would use the first two dimensions as a composite row selector and add the remaining threads along each row. By having each row be its own work group, sub group reductions like *FastGEMV* demonstrated can be applied. Further, I would hold more data in registers to prevent excessive interleaving. I have an unconfirmed suspicion that most of the bandwidth is being

wasted in this way. Unfortunately I did not have time to implement this but it's something I look forward to trying in the future.

Benchmarks and Results

Testing Setup

All benchmarks were run on a Samsung Galaxy Book 360 with an Intel i7-1165G7 Tigerlake CPU and an Intel ® Iris ® Xe Graphics iGPU. Benchmarks were built with the `build.zig` in the source repo[6] for Windows Subsystem for Linux. I used OpenHardwareMonitor to control for the CPU Package Temperature between trials.

LLAMA FFN Burst

I decided to benchmark the popular LLAMA Feed Forward Block. This consists of three matrix multiplications and 2 quantization passes on intermediate results. The FFN was run for 200 iterations. This was meant to be a temperature controlled small burst where throttling was minimal. I observed that even despite these controls, variance and error is high. I would say GGML is still King, but that my CPU routine was competitive. The iGPU routine was too far off to chalk up the difference to error.

Table 1: GGML

	10 Trial Average
Total (s)	0.42
ms / Iteration	2.08
GFLOPS	169.73
Bandwidth (lower bound) (GB/s)	47.74

[6] https://github.com/e253/capstone

Table 2: Capstone CPU

	10 Trial Average	Change
Total (s)	0.42	2.25%
ms / Iteration	2.12	2.25%
GFLOPS	166.05	-2.17%
Bandwidth (lower bound) (GB/s)	44.31	-7.17%

Table 3: Capstone iGPU

	10 Trial Average	Change
Total (s)	1.43	243.86%
ms / Iteration	7.14	243.86%
GFLOPS	49.36	-70.92%
Bandwidth (lower bound) (GB/s)	13.21	-72.33%

LLAMA FFN Burndown

This trial simulates a ~500 token generation with 15000 iterations of the LLAMA FFN.

Table 4: GGML

Total (s)	29.27
ms / Iteration	1.95
GFLOPS	180.53
Bandwidth (lower bound) (GB/s)	50.77
Max Temp (Celsius)	95

Table 5: Capstone CPU

Total (s)	39.03
ms / Iteration	2.60
GFLOPS	135.40
Bandwidth (lower bound) (GB/s)	36.23
Max Temp (Celsius)	95

Table 6: Capstone iGPU

Total (s)	103.71
ms / Iterations	6.91
GFLOPS	50.96
Bandwidth (lower bound) (GB/s)	13.64
Max Temp (Celsius)	85

Conclusion

The results were a bit disappointing, but confirm an important point: model size determines performance. Similar to how Math Kernel Library was unable to speed up llama2.c due to memory boundedness, I was unable to assist here with optimizations that come from a compute bound mindset. My hypothesis about speeding up EGEMV with better cache efficiency overestimated the slowdown caused by struct splitting cache lines. There may be an opportunity to do the FLOPS at a lower clock speed with less energy consumption. I don't have a confident result for the iGPU, due to the fact that my implementation is subpar.

Next Steps

I think that codebook quantization methods are a key part of achieving fast inference with consumer memory. Codebook quantization allows for far better compression than the now trivial output channel scaling used in this paper. Rather than having a zero and scale for part of a row, each row of values is composed through the addition of many “codes”. Different row blocks share a small number of codes and thus the total memory footprint of the matrix can be as much as 16 times less than the equivalent FP32 representation. However, codebooks suffer from a data dependent access pattern. The total memory footprint is improved by 2x over output channel quants used in this paper, however the memory traffic and inference speed cannot fully benefit, for now.

I’m also interested in looking into implementations with Apple AMX instructions which may allow for better efficiency. Apple Silicon is far more efficient than x86 and offers a plethora of platforms for experimentation including ARM NEON SIMD, Metal GPU Kernels, Neural Engine through CoreML (or a patched kernel), and the undocumented AMX instruction set extension for matrix SIMD.

Implementation Tips

This section may be more useful than the rest of the paper. I am attracted to Rust for its ease of downstream use of third party libraries. Due there only being one compiler, `rustc`, it builds reliably across all architectures and operating systems. However, the borrow checker cannot guarantee safety for complex updates to an array in parallel. Use of pointers is eschewed and this is a huge impediment to productivity when doing linear algebra operations.

C++ is the clear choice but does come with difficult builds and difficult use of downstream dependencies, at least when using the traditional Clang/MSVC build system. Zig, a

language and build system, takes care of these headaches. It can fetch dependencies and statically links a tree-shaken, built-from-source Musl/MingW LibC. This made the development experience quite nice. The source repository for this investigation demonstrates how you can use Zig to make C++ dev better.

Appendix A: llama2.c Benchmark Details

```c
void matmul(float* xout, float* x, float* w, int n, int d) {
    // W (d,n) @ x (n,) -> xout (d,)
    // by far the most amount of time is spent inside this little function
    #ifdef mkl
        cblas_sgemv(CblasRowMajor, CblasNoTrans, d, n, 1.0, w, n, x, 1, 0.0, xout, 1);
    #else
        int i;
        #pragma omp parallel for private(i)
        for (i = 0; i < d; i++) {
            float val = 0.0f;
            for (int j = 0; j < n; j++) {
                val += w[i * n + j] * x[j];
            }
            xout[i] = val;
        }
    #endif
}
```

I replaced Andrej's `matmul` routine with that in the picture above and ran generations with 4 OMP threads on his stories110M LLM.

cmd: `OMP_NUM_THREADS=4 ./run stories110M.bin -n 512`

References

Badri, H., & Shaji, A. (2023, November). Half-Quadratic Quantization of Large Machine Learning Models. https://mobiusml.github.io/hqq_blog/

Egiazarian, V., Panferov, A., Kuznedelev, D., Frantar, E., Babenko, A., & Alistarh, D. (2024, February 6). *Extreme compression of large language models via additive quantization*. arXiv.org. https://arxiv.org/abs/2401.06118

Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2023, March 22). *GPTQ: Accurate post-training quantization for generative pre-trained transformers*. arXiv.org. https://arxiv.org/abs/2210.17323

Greenwald, G. (2013, June 11). *Edward Snowden: The whistleblower behind the NSA surveillance revelations*. The Guardian. https://www.theguardian.com/world/2013/jun/09/edward-snowden-nsa-whistleblower-surveillance

Larin, B. (2023, December 27). *Operation triangulation: The last (hardware) mystery*. Operation Triangulation: The last (hardware) mystery. https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., & Han, S. (2024, April 23). *AWQ: Activation-aware weight quantization for LLM compression and Acceleration*. arXiv.org. https://arxiv.org/abs/2306.00978

OpenMathLib. (n.d.). *OpenMathLib/OpenBLAS: OpenBLAS is an optimized Blas library based on gotoblas2 1.13 BSD version.* GitHub. https://github.com/OpenMathLib/OpenBLAS

Tseng, A., Chee, J., Sun, Q., Kuleshov, V., & De Sa, C. (2024, February 6). *Quip#: Even better LLM quantization with Hadamard incoherence and lattice codebooks*. arXiv.org. https://arxiv.org/abs/2402.04396

Volz, D. (2024, April). *Senate passes spying bill, rejecting privacy concerns*. Wall Street Journal. https://www.wsj.com/politics/national-security/senate-passes-fisa-spying-bill-882e7df4

Wang, S. (n.d.). *Wangsiping97/FastGEMV: High-speed GEMV kernels, at most 2.7x speedup compared to Pytorch Baseline.* GitHub. https://github.com/wangsiping97/FastGEMV

Zee, F. G. V., & Geijn, R. A. van de. (2015, June 1). Blis: A framework for rapidly instantiating Blas functionality. https://dl.acm.org/doi/10.1145/2764454?cid=81314495332