The Case for Mechanical Empathy

Ramesh Subramonian

January 20, 2026

Abstract

The article documents the author's experience playing with Andrei Karpathy's excellent one file implementation of Llama-2. The TL;DR is that there is an interesting spot on the performance-programming effort continuum achieved by collaborating with the compiler. In particular, porting selected portions of Karpathy's code to `ispc` yields a 2X performance gain.

1 Introduction

Karpathy's llama2.c project "inferences Llama-2 with one simple 700-line C file (run.c)." The code is well-written — as the author points out, it has been written with exposition and ease of understanding in mind, not necessarily performance.

Nevertheless, we took this as a starting point to see how we could improve performance without sacrificing readability. For this purpose, we ported selected portion of the code using `ispc` — Intel® Implicit SPMD Program Compiler. For those unfamiliar with this excellent project, "`ispc` is a compiler for a variant of the C programming language, with extensions for "single program, multiple data" (SPMD) programming. "

2 Modifications to Karpathy's code

2.1 Cosmetic changes

The cosmetic changes fall into the following categories

1. We broke out several of the compute intensive functions into separate files. Localizing changes helped tune and test smaller pieces without destabilizing the code.

2. Indentation styles can lead to religious wars. I chose to use my own style

3. The command line parameter `-q` was added to indicate whether weights had been quantized.

2.2 Substantive changes

The most substantive change is covered in Section 3

2.2.1 Hint, hint, ...

We provided as much guidance to the compiler as possible. For example,

1. we used the `restrict` type qualifier wherever possible. "By adding this type qualifier, a programmer hints to the compiler that for the lifetime of the pointer, no other pointer will be used to access the object to which it points. This allows the compiler to make optimizations (for example, vectorization) that would not otherwise have been possible."[1]

2. we used the `__attribute__(())` syntax in `ispc` for variable and function declarations, thereby eliminating extra code and letting the compiler know about memory alignment

3. we used the `assume()` mechanism in `ispc` where possible e.g., letting the compiler know that we have padded vectors so that they are multiples of the lane width of the vector register.

2.2.2 Aligning memory

To support vector instructions, we aligned memory on 32-byte boundaries.

1. Karpathy `mmaps` a single binary weights file for all the different kinds of weights. In contrast, we broke the single weights file into twelve separate files, one for each of the weight classes, padding the vectors to be multiples of the vector register width. For example, a 2D array with 2 rows and 3 columns such as $\begin{bmatrix} 1 & 2 & 3 \\ 10 & 20 & 30 \end{bmatrix}$ is laid out in a linear array as shown below when padded to multiples of 4

$$\{1, 2, 3, 0, 10, 20, 30, 0\}$$

2. We used `posix_memalign()` instead of `malloc()` to ensure alignment.

3. We padded all vectors, increasing memory usage somewhat, but ensuring that all vectors are properly aligned.

2.2.3 Use of OpenMP

2.2.4 Quantization

Quantization reduces memory pressure. In particular, we chose to quantize `fp32` as `uint8_t`. In doing so, each vector within a tensor was scaled independently. This means that a vector of floating point numbers of the form

[1] `https://en.wikipedia.org/wiki/Restrict`

$$\{ f_1, f_2, \ldots f_N \}$$

would be represented as unsigned 8-bit integers with two additional pieces of information

$$\min(x), \max(x), \{u_1, u_2, \ldots u_N\}$$

2.3 Nitpicks

1. Karpathy claims that " we use `calloc()` instead of `malloc()` to keep valgrind happy". I found no evidence to suport that claim.

2.4 Quantization

Karpathy chose to support quantization by writing a separate `runq.c` file. We felt that there was sufficient commonality between the original and the quantized version that a separate implementation was not justified. Our approach was

1. write a pre-processing utility to create quantized weights and scales.
2. memory map the quantized weights. For debugging purposes (not for production), the original weights are also memory mapped so that we can assess the lossiness of the compression
3. restrict the quantization to the larger weight classes, not the smaller ones
4. Since the `matmul()` function consumes most of the time, restrict the changes to that function. In particular, the only change made is to the dot product function called from `matmul()` where the lines shown in red are inserted. Note that the signature of the calling function also needed to change so that it had access to the scaling values of `offset` and `delta`.

```
export void dot_prod_qnt(
    uniform float offset, uniform float delta,
    uniform float x[], uniform uint8 y[], uniform int n, uniform float rslt[]
)
foreach ( i = 0 ... n ) {
    varying float y_i = (y[i] * delta) + offset;
    sum += x[i] * y_i;
}
rslt[0] = sum
```

5. Instead of using integer arithmetic on quantized values, we used floating point arithmetic after converting `unit32_t` back to `float` on the fly.

3 ISPC Ports

4 Performance Results

The hardware used for benchmarking was inspired by DHH's blog post `https://world.hey.com/dhh/it-s-a-beelink-baby-243fdaf1`. It was a Beelink SEi14 Mini PC, powered by Intel Ultra 9 185H (up to 5.1GHz) 16C/22T, Mini Computer with 64GB DDR5 at 5600MHz.

We used the same seed and the same prompt on the `stories42M.bin` data set to generate a mini-story. Results are reported by averaging over 20 runs after discarding the top 2 and the bottom 2 results.

We used `gcc` for both versions, with flags including `-O3 -Ofast -flto`. For `ispc`, compiler options included

```
--addressing=32 --opt=fast-math
--math-lib=fast --opt=force-aligned-memory
```

Using Karpathy's code, it runs at XXX tokens/second. Our code, using ISPC, runs at YYY tokens/second.

5 Conclusion