Translating ML Functions to ISPC

Machine Learning Systems

Adam Brohl

December 2024

1 Introduction

With machine learning models rapidly increasing in size; reaching billions of parameters, both the hardware that runs model training and inferencing and the software require thorough optimization to run in a reasonable amount of time. This project, in particular, takes a look at model inferencing on CPUs that utilize short vector intrinsics. Many open source projects, such as llama.cpp [1] tend to use short vector intrinsics to increase performance. This, however, comes at a cost as different processor architectures have different implementations of short vector intrinsics. For example, Intel processors can support AVX2 [2], AVX2 [3], or even another implementation altogether, and ARM processors can support NEON [4], or SVE [5]. To address this issue, this project looks at the cost of replacing short vector intrinsics code in projects like llama.cpp with ISPC [6], which is a language created by Intel that uses a Single Program, Multiple Data (SPMD) methodology of constructing "serial-looking" programs that can target a wide variety of short vector implementations and determining if it is a worthwhile substitution by looking at the lines of code of both implementations, supported short vector implementations, and performance.

2 Short Vector Intrinsics

Short vector intrinsics are a parallel programming model which allows for a programmer to create short vectors of integers, floats, or boolean, and apply operations to each element in the short vector using only a single instruction, hence the term Single Instruction, Multiple Data (SIMD). These short vectors are represented by large registers that typically use somewhere between 128 and 512 bits. There are several short vector implementations; part from the few mentioned earlier, Intel processors alone may support MMX [7], AMX [8], AVX10 [9], and even more. Each of these implementations support different short vector sizes, and have their own operations for working with them. This means, however, that an application utilizing short vector intrinsics wishing to target multiple architectures will need to have code written for each short vector implementation, and distinguish at runtime what to use.

3 Auto Vectorization

An attempt to approach this problem is the use of Auto-Vectorization [10]. Auto-Vectorization allows a compiler to automatically convert serial code into its vectorized counterpart at compile time by looking at unrolled loops, and finding a short vector intrinsic to replace the unrolled statements. While this solution avoids the work of writing multiple variations for different short vector implementations, it results in delicate code, as it relies entirely on the compiler's ability to

auto-vectorize, meaning that the version of the compiler, and the way the code is written, change if the code can even be vectorized or not.

4 ISPC

Another approach to this problem, and the one this project looks at in particular, is to use ISPC [6] (Intel SPMD Program Compiler) to write code that uses special programming constructs to write code that resembles a serial program but utilizes short vectors without the fragility of Auto-Vectorization. ISPC resembles C code, and was built with interoperability to allow easy integration of it to C and C++ projects. At compilation time, ISPC has several flags which indicate which short vector implementation to target. By default, ISPC will compile to AVX2 (if possible).

When writing ISPC code, the language introduces two new types of variables that allow a programmer to write a vectorized program without actually specifying the size of the short vector. These two new types are Varying and Uniform. Uniform types behave like traditional programming variables, in which they are meant to represent a type that does not indicate a short vector. Conversely a Varying type, is a scalar value, for example a 32-bit integer, which ISPC will convert into the actual short vector at compile time. In a way, it is similar to mapping a function to a list of values. The parameter in the function to be mapped is the Varying type, and the list of values is the Uniform type.

5 Llama.cpp & GGML

Llama.cpp [1] is a library which allows for high performance model inferencing an less expensive machines, like everyday CPUs. This library is written in C/C++, and uses a library named GGML (Georgi Gerganov Machine Learning) [11] for optimized operations on tensors, which are frequently used in machine learning. For standard model inferencing, the functions of the highest execution time according to `perf` are listed in the table below.

Function Name	**Execution Time (%)**
ggml_vec_dot_q3_K_q8_K	84.23
ggml_vec_dot_q2_K_q8_K	6.30
ggml_vec_dot_f16	1.59
ggml_compute_forward_mul_mat	1.20
ggml_vec_dot_q6_K_q8_K	0.73
ggml_fp32_to_fp16_row	0.10

6 Translation Results

Property	**Value**
CPU	AMD Ryzen 7 5700U @ 4.372GHz
Model	dolphin-2.2.1-mistral-7b.Q2_K
Model Size	2.87 GiB
Model Parameters	7.24 B
Threads	8
Repetitions	25

In order to benchmark performance, the `llama-bench` tool can be used to automatically calculate the tokens / second of token generation and token processing. For prompt processing, the model was given a completely randomized prompt of 512 tokens, and for text generation, the model was asked to generate a 128-token response (from a randomized prompt).

6.1 ggml_vec_dot_q3_K_q8_K

This function, percentage wise, takes up most of the execution time of the benchmarking program. IT is therefore an ideal candidate for transling into ISPC and evaluating. This function takes in two arrays, and computes a quantized dot product. When measuring performance, its translation was used in place of the original implementation which utilized short vector intrinsics from the AVX2 instrution set. The ISPC code was also compiled to target the AVX2 instruction set. The values in the table show the 95% confidence interval.

	pp512 (t/s)	**tg128 (t/s)**
Original	[15.850, 16.830]	[9.377, 9.503]
ggml_vec_dot_q3_K_q8_K	[4.963, 5.057]	[4.316, 4.324]

As can be seen by the results, the ISPC translation of the original function shows a large decrease in the amount of tokens / second, or a decrease in the function's performance for both the prompt processing and token generation benchmark. As will be explained in more detail later on, the performance loss in this translation largely stems from the original function's dependence on short vector sizes.

6.2 ggml_vec_dot_f16

This function takes in an array of two 16-bit floats represented by unsigned integers, the length of these two arrays, and a pointer to save the result (a 64-bit float) which represents the dot product og the two input arrays. This translation is remarkably simple:

```
typedef uint16 ggml_fp16_t;
typedef double ggml_float;

export uniform ggml_float ggml_vec_dot_f16_ispc(uniform int n,
                                uniform ggml_fp16_t x[],
                                uniform ggml_fp16_t y[]
                                ) {

 uniform ggml_float sumf = 0.0;

 foreach (i = 0 ... n) {
      sumf += reduce_add((ggml_float) ((float) float16bits(x[i])
                  * (float) float16bits(y[i])));
 }

 return sumf;
}
```

As opposed to the original implementation, which included a compile time and runtime check to detect as well as an overflow loop for when the remaining number of elements to process was less

than the short vector size, this implementation requires none of those things, making it easier to maintain.

When measuring performance, this function was used in place of the original implementation which utilized the `vfmaq_f32` intrinsic and tested against the original with the `llama-bench` tool. The ISPC code was compiled to target the AVX2 instruction set. The values in the table show the 95% confidence interval.

	pp512 (t/s)	**tg128 (t/s)**
Original	[15.850, 16.830]	[9.377, 9.503]
ggml_vec_dot_f16	[13.544, 13.976]	[8.763, 8.817]

As can be seen by the results, the ISPC translation of the original function shows a decrease in the amount of tokens / second, or a decrease in the function's performance, in both of the benchmarks.

6.3 ggml_fp32_to_fp16_row

This function does not take up much of the execution time and is almost trivial, but it still demonstrates the advantage of an ISPC translation. This function simply takes in an input and output array, of 32-bit and 16-bit floats respectively, and converts them all using the AVX2 short vector implementation.

```
typedef uint16 ggml_fp16_t;

export void ggml_fp32_to_fp16_row_ispc(uniform const float x[],
                                       uniform ggml_fp16_t y[],
                                       uniform int64 n) {
 foreach (i = 0 ... n) {
  y[i] = intbits((float16) x[i]);
 }
}
```

Like the previously translated function, this one also avoids the need to add compile time or runtime checks to detect support for AVX2 instructions. There is also no need to write an overflow loop, as IPSC will also automatically handle that case.

Like the previous function, when measuring performance, this ISPC translation was used in place of the original and tested against the original with the `llama-bench` tool. Both the original and the ISPC translation utilize AVX2 instructions. The values in the table show the 95% confidence interval.

	pp512 (t/s)	**tg128 (t/s)**
Original	[15.850, 16.830]	[9.377, 9.503]
ggml_fp32_to_fp16_row	[16.605, 16.675]	[10.691, 10.949]

As can be seen by the results, the relative performance between the two implementations are inconclusive for the prompt processing benchmark, and show a slight decrease in performance for the text generation benchmark in the ISPC translation.

	pp512 (t/s)	tg128 (t/s)
Original	[15.850, 16.830]	[9.377, 9.503]
ggml_vec_dot_q3_K_q8_K	[4.963, 5.057]	[4.316, 4.324]
ggml_vec_dot_f16	[13.544, 13.976]	[8.763, 8.817]
ggml_fp32_to_fp16_row	[16.605, 16.675]	[10.691, 10.949]
All Translations	[4.827, 4.913]	[4.246, 4.254]

6.4 Combined Benchmark

The following table shows the performance of each individual translation separately and together. Results are displayed as 95% confidence intervals.

The data shows that not taking into the `ggml_vec_dot_q3_K_q8_K` translation into consideration, there is little effect on the overall performance of the benchmarking application. `ggml_vec_dot_q3_K_q8_K`, as will be explained earlier on, is not able to be translated in a way which fully utilizes the ISPC programming constructs because of the nature of its original implementation. Mainly, the `ggml_vec_dot_q3_K_q8_K` function requires the short vectors to be of a specific size, and does not modify each element of the short vector independently. Thus, a less efficient translation was written in ISPC.

7 Conclusion

In the results of the translated functions, ISPC came close to the performance of the original implementation, but at times performed less than the original. In the Llama.cpp code repository, much of the SIMD code was specifically written to ensure the compiler would generate the most optimal target possible. For example, in the `ggml_vec_dot_q3_K_q8_K` function, a majority of the code was unrolled to both make auto-vectorization more likely in the serial sections of code, and to provide better scheduling than the compiler. As was noted in the source code, writing it in a different style would result in a 2-4x performance loss for the function.

Another item of note was the inability to fully express the original `ggml_vec_dot_q3_K_q8_K` algorithm in ISPC due to its dependence on specific short vector sizes and heavy use of type punning. For example, a portion of the code had a 256-bit short vector of 32-bit integers, then applied the `_mm256_cvtepi8_epi16` intrinsic, which performs a sign extension on 8-bit integers to 16-bit integers. As ISPC uses a single varying value to represent an entire short vector, this is not possible to express with varying values in ISPC. As a result, the only way to use short vectors is to use the ISPC types which convey the short vector size at compile time in the source code which is essentially equivalent to the original implementation but in a different language. There are other drawbacks as well, such as not being able to use any of the builtin ISPC functions such as `reduce_add()`. As a result, these functions need to be implemented manually, turning a task which originally used a single instruction to as many instructions as a serial implementation.

References

[1] G. Gerganov, "GitHub - ggerganov/llama.cpp: LLM inference in C/C++ — github.com." https://github.com/ggerganov/llama.cpp, 2023. [Accessed 01-10-2024].

[2] Intel, "Intel&xAE; Advanced Vector Extensions 2 (Intel&xAE; AVX2) - 009 - ID:655258 — 12th Generation Intel&xAE; Core&x2122; Processors — edc.intel.com." https://edc.intel.com/content/www/tw/zh/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/009/intel-advanced-vector-extensions-2-intel-avx2/. [Accessed 26-09-2024].

[3] Intel, "Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Overview — intel.com." https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. [Accessed 26-09-2024].

[4] ARM, "Neon — developer.arm.com." https://developer.arm.com/Architectures/Neon. [Accessed 26-09-2024].

[5] ARM, "SVE — developer.arm.com." https://developer.arm.com/Architectures/Scalable [Accessed 26-09-2024].

[6] M. Pharr and W. R. Mark, "ispc: A spmd compiler for high-performance cpu programming," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–13, IEEE, 2012.

[7] "Details about MMX™ Technology Intrinsics — intel.com." https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/details-about-mmx-technology-intrinsics.html. [Accessed 02-12-2024].

[8] "Intel® Advanced Matrix Extensions Overview — intel.com." https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html. [Accessed 02-12-2024].

[9] "Intel® Advanced Vector Extensions 10.2 (Intel® AVX10.2) Architecture Specification — intel.com." https://www.intel.com/content/www/us/en/content-details/836199/intel-advanced-vector-extensions-10-2-intel-avx10-2-architecture-specification.html?wapkw=avx10. [Accessed 02-12-2024].

[10] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 132–143, 2006.

[11] "GitHub - ggerganov/ggml: Tensor library for machine learning — github.com." https://github.com/ggerganov/ggml. [Accessed 02-12-2024].