Exploring Performance of Cache-Aware Tiling Strategies in MLIR Infrastructure

Mingyu Chen  
University of Science and Technology of China  
China

Hongbo Rong  
Intel  
USA

Yu Zhang  
University of Science and Technology of China  
China

Jianhui Li  
Intel  
USA

ABSTRACT
MLIR is a rising tool for many fields, such as deep learning or performance portability. Based on the expectations of the abilities of MLIR, we explore the potential of MLIR for generating high-performance code on the CPU. We adopt the GEMM algorithm used in BLIS framework for loop tiling in a cache-aware way by MLIR. After manual optimization, we compare the performance of optimized program on single-core with oneDNN matmul benchmark, and present analysis of performance gap at assembly code level. The results show that our program reaches 80% of performance of oneDNN program.

1 INTRODUCTION
In this work we explore the potential of MLIR to generate high-performance code on the CPU. We implement the algorithm used in BLIS framework by MLIR and perform manual optimization on our MLIR-written program. The final experiment results show MLIR rivals the expert-tuned library in single-core performance. Here we use oneDNN benchmark as the representative of expert-tuned libraries. Furthermore, we compare both assembly code to explain the performance gap between our program and oneDNN at the assembly code level.

The main contributions of this paper are:
- We implement an MLIR-written GEMM program for single-core execution on CPU. After optimization, its performance can reach 80% of performance of expert-tuned library.
- We make attempts to analyze the reasons why there is a performance gap between our program and oneDNN at the assembly code level.

2 APPROACH
2.1 Design
We choose general matrix-matrix multiplication as the start point for testing the ability of MLIR. It is simple but important in many domains. For simplicity, we restrict ourselves to the case of single-core implementation of GEMM in single precision floating-point number. The single-precision FP is used here since it is more commonly used in deep learning applications.

To implement high-performance GEMM, we need a good enough algorithm to ensure that our program has a high performance-ceiling. Here we use the algorithm adopted by the BLIS framework.

To our knowledge, PlaidML[4] and Polygeist[7] are the only two frontends have potential to generate high-performance code.

2.2 Optimization
2.2.1 Tiling and Packing. We use the same cache tiling strategies as the one used in paper[6]. Its ingenuity lies in that it employs the tiled loop nest to keep the data portions to be accessed in the cache, and replaces the useless data by arranging the distance in memory between data pieces. This strategy is characterized by five tiling parameters: \( m_r, k_c, n_c, m_r, n_r \). There are two ways to calculate these parameters in the above work: empirical value or analytical model. We will discuss more details about them in section 3.

In conjunction with tiling, we pack the input matrices into continuous buffer for better access. It’s known that accessing consecutive memory locations is usually faster than nonconsecutive memory accesses.

2.2.2 Vectorization. The vector dialect in MLIR provides an interface to use vector types and operations on vector types for better hardware utilization. We use operations of vector dialect within the microkernel for two reasons: 1) We use the load and store operations of vector dialect to make arithmetic operations within microkernel only operate (vector) registers; 2) These operations are eventually lowered to the micro-ops of hardware ISA, which can improve the utilization of hardware vector arithmetic units.

3 PRELIMINARY RESULTS
Our experiments presented in this section are conducted on the Intel(R) Xeon Gold 5220R CPU with maximum frequency at 4.0 GHz. This CPU supports AVX-512 instruction set and has 1 AVX-512 FMA unit. Therefore, the theoretical peak performance of this
Table 1: The comparison between different benchmarks by VTune

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Analytically Derived</th>
<th>Empirically Derived</th>
<th>OneDNN Matmul Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Frequency</td>
<td>3.5 GHz</td>
<td>3.6 GHz</td>
<td>3.5 GHz</td>
</tr>
<tr>
<td>CPI</td>
<td>1.44</td>
<td>0.69</td>
<td>0.645</td>
</tr>
<tr>
<td>Memory Bound</td>
<td>57.9%</td>
<td>30.0%</td>
<td>8.9%</td>
</tr>
</tbody>
</table>
| Fig. 1: GEMM on a single core of Intel Xeon Gold 5220R, M=4096, N=4096, K=4800

CPU is 128 GFLOPS (single precision). For expert-tuned library, we use the matmul benchmark of oneDNN for comparison.

3.1 Evaluation

3.1.1 Evaluation of MLIR-written GEMM. Fig.1 illustrates the performance of our MLIR program and oneDNN program. Here we use two sets of parameters obtained by empirical value and analytical model respectively. The "Analytically Derived" column represents the performance of program using analytical model, and the "Empirically Derived" column is the performance of program using empirical value. In our machine, the parameters obtained by analytical model is different from the ones obtained empirically, which results in the performance gap.

Table 1 illustrates the difference in some metrics between our program and oneDNN benchmark by VTune. Besides performance, the "Empirically Derived" version has lower memory bound than "Analytically Derived". However, its performance only reaches 80% of GFLOPS of oneDNN benchmark. Its memory bound is higher than oneDNN benchmark, which results from accessing L3 Cache and DRAM.

In fact, we also evaluate the linalg.matmul operation provided by MLIR. Since its performance is rather worse than ours, limited to the space, we don’t illustrate it in this workshop paper.

3.1.2 Analysis at Assembly Level. To further analyse the code generation of MLIR compiler, we dump the assembly code from the generated executable file. The Fig. 3 shows the assembly code of the inner most loop of our MLIR-written GEMM program. The vfmadd micro-ops indicate our program implement vectorization successfully. In fact, the code generation for AVX-512 can be found at LLVM discourse[1].

We also get part of oneDNN hotspot assembly code by VTune, as shown in Fig. 2. We compare both and make the following observations:

- Our program doesn’t use all AVX-512 registers(%zmm);
- The operands of some vfmadd micro-ops are not all register;
- There are some non-FP micro-ops within the microkernel of our program, which may lower the performance of our program.

These problems may arise from the utilization of vector registers and cache or the packing procedure. Compared with the assembly code of oneDNN benchmark, there are some parts in our code slice hard to find the underlying logic. In a word, current assembly code cannot be said to be good.

3.1.3 Summary. The above experiment results show there is a performance gap between our program and oneDNN benchmark. There are many possible reasons, such as inefficient source code, aggressive optimization strategies of compiler or bad memory access etc. We are still in analysis.

4 CONCLUSION

We are one of the early work exploring the HPC potential of MLIR. Although there is still a performance gap, we think MLIR is potential to be developed to be a tool for high-performance computing. With the support of LLVM on multiple backends, MLIR is also high-level enough for portable performance.
REFERENCES


