Exploring Performance of Cache-Aware Tiling Strategies in MLIR Infrastructure

Mingyu Chen
University of Science and Technology of China
China

Yu Zhang
University of Science and Technology of China
China

Hongbo Rong
Intel
USA

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[6] 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[3] benchmark.

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 exper-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[2].

To our knowledge, PlaidML[4] and Polygeist[7] are the only two frontends have potential to generate high-performance code. PlaidML provides a C++/Python embedded domain-specific language, which can be used as a frontend to emit MLIR code and JIT compile and execute it. It employs self-designed pxa dialect as parallel extensions for affine dialect and calls the LIBXSMM[5] library for better implementation of BLAS kernel. But these mechanism is hard to upstream to MLIR infrastructure and lower the portability of generated code. Polygeist, as a compilation workflow, provides a C/C++ frontend for MLIR. Its optimization depends on what polyhedral tools can do. Therefore, we implement and optimize MLIR code manually and choose to use *mlir-cpu-runner* to compile it.

While using higher-level abstraction can maintain the loop structure in the program, we choose the memref/scf dialect as the basis of our program for hardware-aware programming.

The subsequent sections is an introduction to the manual optimization performed on our program.

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, and 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

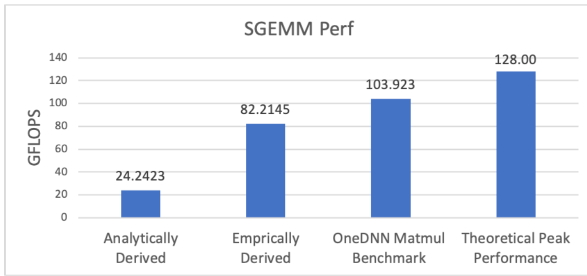

Figure 1: GEMM on a single core of Intel Xeon Gold 5220R, M=4096, N=4096, K=4800

	Analytically Derived	Empirically Derived	OneDNN Matmul Benchmark
GFLOPS	24.2423	82.2145	103.923
CPU Frequency	3.5 GHz	3.6 GHz	3.5GHz
CPI	1.447	0.693	0.645
Memory Bound	57.9%	30.0%	8.0%

Table 1: The comparison between different benchmarks by VTune

```
0x7f56902c6550      vbroadcastss zmm0, dword ptr [r11+0x4b0c]
0x7f56902c655a      prefetcht0   zmmword ptr [r10+0x740]
0x7f56902c6562      vfmadd231ps  zmm27, zmm7, zmm0
0x7f56902c6568      vfmadd231ps  zmm26, zmm6, zmm0
0x7f56902c656e      vfmadd231ps  zmm25, zmm5, zmm0
0x7f56902c6574      vfmadd231ps  zmm24, zmm4, zmm0
0x7f56902c657a      vbroadcastss zmm0, dword ptr [r11+0x960c]
0x7f56902c6584      prefetcht0   zmmword ptr [r10+0x780]
0x7f56902c658c      vfmadd231ps  zmm23, zmm7, zmm0
0x7f56902c6592      vfmadd231ps  zmm22, zmm6, zmm0
0x7f56902c6598      vfmadd231ps  zmm21, zmm5, zmm0
0x7f56902c659e      vfmadd231ps  zmm20, zmm4, zmm0
```

Figure 2: Code slice of oneDNN matmul at assembly level

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 set of parameters obtained by empirical value and analytical model mentioned in section 2.2 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". version. 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.

```
4022c0: c5 7b 10 6c 13 f8      vmovsd -0x8(%rbx,%rdx,1),%xmm13
4022c6: 62 52 7d 48 18 f5      vbroadcastss %xmm13,%zmm14
4022cc: 62 71 7c 48 10 7f ff   vmovups -0x40(%rdi),%zmm15
4022d3: 62 e1 7c 48 10 07      vmovups (%rdi),%zmm16
4022d9: 62 f2 05 58 b8 7c 13   vfmadd231ps -0x18(%rbx,%rdx,1){1to16},%zmm15,%zmm7
4022e0: fa
4022e1: 62 72 05 58 b8 44 13   vfmadd231ps -0x14(%rbx,%rdx,1){1to16},%zmm15,%zmm8
4022e8: fb
4022e9: 62 52 7d 48 16 ed      vpermps %zmm13,%zmm0,%zmm13
4022ef: 62 72 05 58 b8 4c 13   vfmadd231ps -0x10(%rbx,%rdx,1){1to16},%zmm15,%zmm9
4022f6: fc
4022f7: 62 72 05 58 b8 54 13   vfmadd231ps -0xc(%rbx,%rdx,1){1to16},%zmm15,%zmm10
4022fe: fd
4022ff: 62 52 05 48 a8 f3      vfmadd213ps %zmm11,%zmm15,%zmm14
402305: 62 52 05 48 a8 ec      vfmadd213ps %zmm12,%zmm15,%zmm13
40230b: c5 7b 10 64 13 10      vmovsd 0x10(%rbx,%rdx,1),%xmm12
402311: 62 52 7d 48 18 dc      vbroadcastss %xmm12,%zmm11
402317: 62 f2 7d 50 b8 3c 13   vfmadd231ps (%rbx,%rdx,1){1to16},%zmm16,%zmm7
40231e: 62 72 7d 50 b8 44 13   vfmadd231ps 0x4(%rbx,%rdx,1){1to16},%zmm16,%zmm8
402325: 01
402326: 62 72 7d 50 b8 4c 13   vfmadd231ps 0x8(%rbx,%rdx,1){1to16},%zmm16,%zmm9
40232d: 02
40232e: 62 72 7d 50 b8 54 13   vfmadd231ps 0xc(%rbx,%rdx,1){1to16},%zmm16,%zmm10
402335: 03
402336: 62 52 7d 48 16 e4      vpermps %zmm12,%zmm0,%zmm12
40233c: 62 52 7d 40 a8 de      vfmadd213ps %zmm14,%zmm16,%zmm11
402342: 62 52 7d 40 a8 e5      vfmadd213ps %zmm13,%zmm16,%zmm12
402348: 48 83 ef 80            sub    $0xffffffffffffff80,%rdi
40234c: 48 83 c2 30            add    $0x30,%rdx
402350: 48 81 fa 18 18 00 00   cmp    $0x1818,%rdx
402357: 0f 85 63 ff ff ff      jne    4022c0 <blis_gemm+0x770>
```

Figure 3: Code slice of MLIR-written microkernel at assembly level

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

[1] Aart Bik. visited in Feb. 2023. Case Study Docs on Vector Dialect CPU Codegen. https://discourse.llvm.org/t/case-study-docs-on-vector-dialect-cpu-codegen/1674/9

[2] Uday Bondhugula. 2020. High performance code generation in mlir: An early case study with gemm. *arXiv preprint arXiv:2003.00532* (2020).

[3] Github. visited in Feb. 2023. oneDNN. https://github.com/oneapi-src/oneDNN

[4] Github. visited in Feb. 2023. PlaidML. https://github.com/plaidml/plaidml

[5] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[6] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, 2 (2016), 1–18.

[7] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.