PARALLELIZATION OF POLYBENCH KERNELS *symm* AND *gemver*

Xudong Jiang, Shuhao Li, Rongxing Liu, Xin Hong, Junxiao Cao

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

As a suite of benchmarks representing typical computations in diverse applications, Polybench is a useful tool for assessing computing platform and compiler performance. Our goal is to parallelize the *symm* and *gemver* kernels from Polybench using OpenMP, CUDA, and MPI with various optimization techniques, then evaluated the performance of our implementations against different baselines.

1. INTRODUCTION

Motivation. Polybench[1] is a suite of polyhedral benchmarks with static control flow for evaluating polyhedral compilers such as polly[2]. It is also useful for other compiler studies such as automatic parallelization. In this project, we focus on the parallelization of Polybench. In addition, instead of studying automatic polyhedral compiler optimization, we investigate what can be achieved by human handcraft optimization and explore effective optimization techniques. We choose 2 kernels with different characteristics to explore different aspects of optimization: *symm* which is typically compute-bound, and *gemver* which is typically memory-bound. We tried using different parallel programming paradigms and applied different optimization techniques based on the distinct characteristics of each algorithm and compared their performance and scalability to compiler automatic parallelization such as icc, polly [2] and state-of-the-art implementations such as MKL, AOCL, and OpenBLAS [3] under different configurations.

2. BACKGROUND

symm and *gemver* are basic linear algebra subprograms [4, 5] (BLAS), which are useful in scientific and engineering applications.

symm. *symm* stands for Symmetric Matrix Matrix Multiplication which is available in current BLAS [4] implementations [3]. Given a symmetric matrix A with size $M \times M$, dense matrices B and C with size $M \times N$, two scalar values α and β. The kernel needs to compute $\alpha \cdot AB + \beta \cdot C$, which is similar to the *gemm* kernel in BLAS, except that A is a symmetric matrix with a special storage format. Specifically, in Polybench, A is stored as a dense matrix in lower triangular form.

gemver. *gemver* is part of the updated BLAS [5], which is usually not included in current BLAS implementations. It contains multiple matrix-vector multiplications and matrix additions. Given two scalars α and β, seven vectors $u1$, $u2$, $v1$, $v2$, x, y, z of length N, and a matrix A with size $N \times N$, the kernel does the following computations: $A = A + u_1 v_1^T + u_2 v_2^T$, $x = \beta \cdot A^T y + z$, $w = a \cdot Ax$.

Related Works. Optimization for Level-3 BLAS such as *gemm* and *symm* has been extensively studied in academia and industry. The widely used OpenBLAS [3] is based on GotoBLAS [6], which analyzed how to effectively utilize multi-level cache hierarchy with multi-level blocking. Another work [7] provides an analytical model to find the optimum blocking parameters. While these works achieve near optimum single thread performance, other studies [8] explore the potential of parallelism by analyzing each level of the loops. Another work [9] focuses on IO optimality for distributed computation. We will try applying the optimization techniques in previous works and try to achieve comparable performance to state-of-the-art implementations such as MKL, AOCL, and OpenBLAS. On the other hand, Level-2 BLAS such as ger, gemv, and *gemver* are not a focus of previous research, so we can explore potential optimization techniques such as operator fusion, SIMD, and NUMA.

3. PROPOSED METHOD

We chose OpenMP and CUDA as the two parallel paradigms for *symm*, and OpenMP and MPI for *gemver*.

3.1. OpenMP implementation of *symm*

We apply techniques used by previous studies on Level-3 BLAS [6, 7, 8], with the awareness of the hardware-specific details such as SIMD, and CPU topology of Zen2 CPU on Euler cluster.

Multi-level blocking. We apply multi-level blocking [6] with optimal block size according to the analytical model in [7]. Specifically, we use $m_r = 6, n_r = 8$ for register level since each core in Zen2 CPU can issue 2 AVX-256 FMA with 5 cycles latency. Then we use $k_c = 256, n_c = 72$ for 32KiB L1 and 512KiB L2 cache. Finally, we use a flexible $m_c <= 4080$ for a 16MiB L3 cache shared by 4 cores.

SIMD intrinsics. The multi-level blocking [6] packed the original matrix to a buffer so that the blocked matrix access will be continuous in memory, then perform efficient register level matrix multiplication. Packing and inner matrix multiplication kernels are important for performance, so we explicitly write AVX intrinsic for these kernels. Specifically for packing, we implement a fast register level transpose by first transposing a 2x2 subblock without crossing the lane and then permuting blocks across the lanes. In addition, we perform early checks of boundary conditions at each block level to determine the relative position to the diagonal in A as early as possible.

CPU topology aware work distribution. Parallelizing multi-level blocking [6] like in [8] requires nested parallelism. However, currently, GCC OpenMP support for nested parallelism is not satisfying because it will repeatedly fork and join threads in nested parallel regions instead of using a thread pool. Therefore, instead of directly using OMP for, we set thread affinity and manually distribute work according to CPU topology so that threads sharing the same L3 cache can share the same A block.

Prepacking. Previous work [8] interleaves packing and computation. However, OpenMP does not support fine-grained barriers, and Vtune profiling results show that 26% of the runtime was wasted on synchronization. By finishing all the packing before computation, we only need 1 barrier and reduce the drawback of coarse barrier in OpenMP, which is worth the overhead of additional buffer access.

COSMA. We also tried the blocking in k-dimension as suggested in COSMA [9], but it doesn't help. Probably it is because reducing communication in a single node is not that important.

3.2. CUDA implementation of *symm*

As *symm* is only a special case of general matrix matrix multiplication(*gemm*) kernels, lots of our ideas for optimizing the kernel comes from a blog talking about optimizing CUDA *gemm* kernels[10].

We chose to implement our version of the *symm* kernel in a row-major system and matrix A is stored as lower triangular form.

Shared memory caching. Aside from global memory, GPU has another type of memory called shared memory. Shared memory is located on chip thus can achieve much lower latency and higher bandwidth than global memory. To make good use of this shared memory region, we can compute Matrix Matrix Multiplication in a blocked fashion as shown in Fig.1.

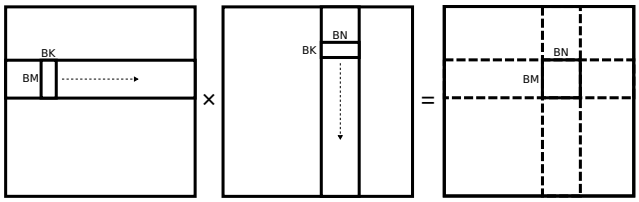


Fig. 1: Shared memory blocking for doing *gemm*

During computation, a thread block first load a block of A and B into shared memory collaboratively. After finishing loading the blocks, a small *gemm* on the loaded block is done and the result gets accumulated into the current block of C.

As we are dealing with *symm*, the loading of A's blocks into shared memory is a bit different from that of *gemm*. When loading elements that are above diagonal, we actually load the corresponding elements below the diagonal into the shared memory.

Vectorized loading. Elements from global memory can be loaded into shared memory more efficiently by using vectorized loading. CUDA provides vector data type for float as float4. Below is a sample line of code that we use for transfering elements from global memory into shared memory using vectorized loading.

```
reinterpret_cast<float4 *>(&As[curr_As_idx])[0] =
     reinterpret_cast<const float4 *>(&A[
    curr_A_idx])[0];
```

Adaptive loading of A. In a row major system, when the block we are trying to load from A is entirely below the diagonal, the block is natively stored as a row-major one. When the block is entirely over the diagonal, then the corresponding block below the diagonal is actually stored as column-major in the global memory. Therefore, to speed up the loading from global memory to shared memory, we can load the blocks of A adaptively into the shared memory as row or column major depend on whether the block is entirely below or above diagonal. For non-adaptive loading variants, we chose to always load blocks of A into shared memory as column-major. In other words, for those blocks natively stored as row-major in the global memory, we transpose them in-place during loading.

Thread tiling. We can use a single thread on GPU to compute multiple elements in the result matrix. This can reduce the total number of threads used thus reduce the overhead of launching threads. Also, we can make better use of registers, which is even after than shared memory, by using each thread to handle a micro block in the result matrix. A micro block of A and B first gets loaded into registers and a thread performs a micro matrix matrix multiplication on the blocks in registers to accumulate the result matrix.

Tensor Core. Aside from implementing matrix matrix multiplication by ourselves, we also used Tensor Core to perform the matrix matrix multiplication for us. Tensor Core can compute $C = AB + C$ on floats while sizes of A, B and C are 16×8, 8×16 and 16×16 respectively. One drawback for using Tensor Core is that Tensor Cores are actually performing float arithmetics at a reduced precision. The data type used in Tensor Cores for A and B is actually tf32 instead of float. Though having the same range as float, tf32 actually has a machine epsilon that is equal to the float16 (half) data type.

3.3. Gemver

We apply OpenMP and MPI for gemver parallization because OpenMP may provide lower overhead for a single node and MPI can provide the ability to scale to multiple nodes. The work is distributed according to rows of A. For MPI, each process explicitly receives an equal portion of A, while in OpenMP, the portion of the matrix is implicitly allocated to the NUMA nodes of the corresponding thread by the first touch policy. After the transposed matrix-vector multiplication, the intermediate x vector is reduced, and the final w vector is gathered after the non-transposed multiplication. The following discusses other optimization techniques.

SIMD. In both MPI and OpenMP versions of gemver, we explicitly wrote AVX SIMD intrinsics for vectorization. The experiments showed that using this method led to a notable improvement in performance.

Operator fusion. In both the OpenMP and MPI versions of gemver, we fused the 2 rank-1 updates with the first transposed matrix-vector multiplication, by computing elements of A on the fly, which reduces 1 read of A.

Non-blocking MPI. We also tried using non-blocking MPI calls to overlap computation and communication. However, this didn't work well on the single-node experiments, probably because there is no benefit of RDMA on a single-node while the overhead of MPI calls increases.

Backward Looping. In the OpenMP version of *gemver*, we also tried performing the second matrix-vector multiplication by iterating over the corresponding part of A in reverse order, which allows us to reuse the recently used elements of A in the cache.

4. EXPERIMENTAL RESULTS

Experimental setup. We conducted all of our experiments on the ETH Euler Cluster. We benchmarked our CPU kernels on a single computing node with up to 128 cores because of privilege limits. GPU kernels were benchmarked on an RTX 4090.

System Detail. Compute nodes in different phases of Euler have different types of AMD CPUs, for consistent measurement, we explicitly require nodes with 2 $\times$ AMD EPYC 7H12 CPU, which have an on-demand frequency governor with a range 1.5GHz - 2.6GHz, and a 3.3GHz boost frequency. There are 2 $\times$ 64 cores in total, each core can issue 2 AVX-256 FMA each cycle which allows a 16 flops/cycle per core peak performance. The L1D, L2 are 8-way associative caches with size 32KiB, and 512KiB respectively. A 16 MiB 16-way associative L3 cache is shared by 4 cores in each CCX. There are 2 $\times$ 4 NUMA nodes, each having 2 memory channels with 8B bus width, which allows a 409.6 GB/s theoretical peak bandwidth with the 3200MHz memory frequency. The operating system is CentOS 7 with Linux kernel version 3.10.0. We use OpenMPI 4.1.4 and GCC 11.4.0 compiler with -O3 -ffast-math -march=native flags.

For experiments on CPU kernels, to utilize more memory channels, we spread threads to more NUMA nodes by setting OMP_PROC_BIND=spread, except for experiments for OpenBLAS baseline which will revert to single-thread execution when this is set. We used mmap with flag MAP_PRIVATE and MAP_ANONYMOUS to allocate new memory. Before the initialization, we ran the tested kernel to touch the memory to ensure the page was allocated to the core that accesses it first.

We use CUDA 12.1.1 for GPU kernels and the building was handled by CMake 3.26.3 with flag -DCMAKE_BUILD_TYPE=Release. We confirmed that the compilation command generated by CMake in Release mode has "-O3" enabled.

Measurements. We use LibLSB 0.2.2 compiled with PAPI 7.0.1 for performance measurements. In OpenMP benchmark executions, runtime was determined by the master thread. In MPI benchmark executions, the runtime was the median of individual process runtimes in each run, and we excluded the time for initial data distribution. Experiment runtimes, for both OpenMP and MPI benchmarks, were represented by the median of all runs. We measure under warm cache configuration with warmup runs. The number of benchmark runs was adjusted to ensure a 95% confidence interval within 5% of the median runtime.

For the CUDA versions of benchmarks, the runtime was measured by the *cudaEventElapsedTime* function in milliseconds. Each version of kernels were ran for multiple profiling iterations. Within each profiling iteration, the number of runs of a kernel was determined to ensure each profiling iteration lasts at least 100ms to reduce the overhead of timing routines. The runtime of the kernel within a profiling iteration was chosen to be the average runtime in that iteration.

Baselines. For *symm*, there exist BLAS implementations including AOCL 4.1.0, MKL 2022.1.2, and OpenBLAS

0.3.26. To make MKL work efficiently on AMD CPU, we override the mkl_serv_intel_cpu_true() to skip CPU detection. AOCL and OpenBLAS with multi-threading support are compiled from source using GCC. We also tried automatic compiler parallization such as icc 2021.5.0 and polly with Clang 17.0.6. icc requires that the source codes are contained in one file for successful parallelization, while polly failed for parallizing *symm*. The *symm* in cublas library was used as our baseline for the CUDA version of our *symm* kernel.

For *gemver*, lacking a suitable existing implementation, we created a baseline using 2 ger, 2 gemv, and 1 axpy BLAS kernels. We found that only OpenBLAS provides reasonable performance. One reason may be because some BLAS libraries such as AOCL do not parallelize some level-2 BLAS kernels such as ger. We also tried icc 2021.5.0 and polly with Clang 17.0.6, which both succeeded in parallelization.

4.1. OpenMP implementation of Symm

We implemented our kernel with multi-level blocking [7] with specific block sizes. For simplicity, we currently only focus on input sizes that are multiples of our block sizes.

Comparison to Baselines. Figure 2 and Figure 3 compare the runtime and performance between our best OpenMP kernel with other baselines on various problem sizes. We achieve comparable results to AOCL and outperform MKL and OpenBLAS, which may be because MKL and OpenBLAS are not specifically optimized for Zen2 architecture. For the icc automatic parallelization baseline, only results with input size $M = 2034$ are shown here because others are too slow. Polly [2] failed to parallize *symm*. For the percentage of peak performance, the work is estimated at $2M^2N + 2MN$, and we use flops/s instead of flops/cycles for performance because different cores may have different frequencies. The peak performance is estimated using the 3.3 GHz boost frequency, which is 52.8 Gflops/s per core. We also show how performance varies with input size in Figure 5, and the performance is stable when the input size is larger than 4608.

Potential bottleneck. Some of the performance drop after 32 threads may be explained by the frequency drop as shown in Fig. 4 We instrument PAPI to measure the frequency of each core using PAPI_TOT_CYC and perf::TASK-CLOCK PAPI event for cycle and time. The violin plot shows the distribution of core frequency across all cores and measurement runs. When the number of threads is larger than 32, the probability of having a low-frequency core is higher. Since the work is equally distributed across cores, the frequency variation will cause an imbalanced runtime, and more time will be wasted waiting for slow cores. Some research [11] have investigated this issue, and this may be a potential further improvement.

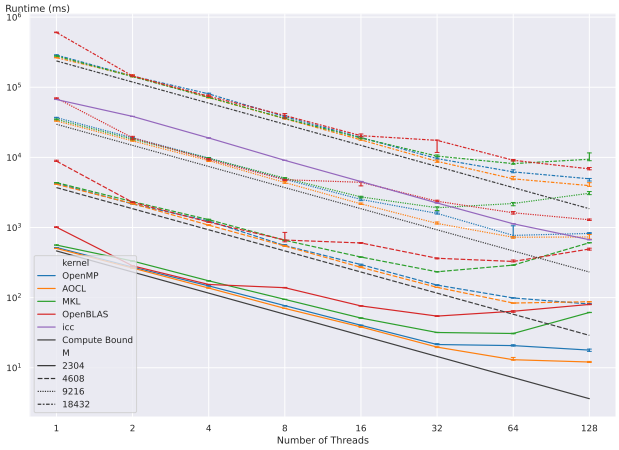


Fig. 2: Comparison of OpenMP *symm* with baselines. Runtime vs. number of threads for various input sizes. The error bar denotes the 95% nonparametric confidence interval of the median.

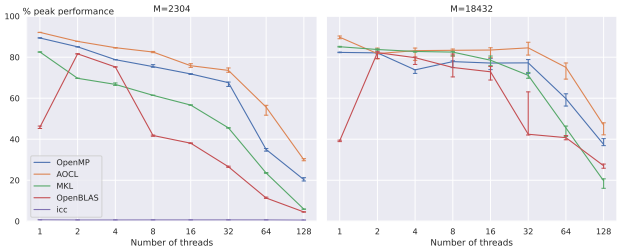


Fig. 3: Comparison of OpenMP *symm* with baselines. Performance vs. number of threads for various input sizes. The error bar denotes the 95% nonparametric confidence interval of the median.

4.2. CUDA implementation of *symm*

Performance Analysis. For a problem size s, the size of matrices are determined to be $M = N = s$, and the work is estimated to be $3s^3 + 2s^2$. Therefore, after getting the running time of the kernel in seconds, t, the performance of the kernel can be computed as $(2s^3 + 3s^2)/t$. The peak performance for single-precision computation on RTX 4090 is 82.58 TFlop/s. Then, the percentage of peak performance reached of the kernel, on RTX 4090, can be derived as $(2s^3 + 3s^2)/(t \times 82.58 \times 10^{12})$. In other words, lower running time and higher performance are two interchangable terms in this case.

Reference kernel using cublas. To compare with cublas, we implemented two reference implementations using cublas. Cublas requires the input matrices to be column-major. Line *cublas* in Fig.6b shows the performance of the cublas kernel in such column-major system. As we implemented all our kernels for row-major matrices, to do fair comparison, a

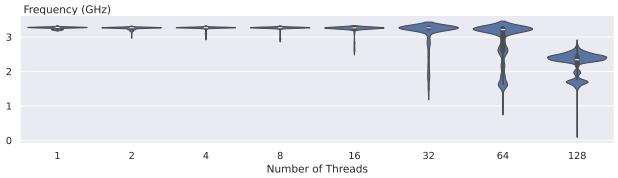


Fig. 4: Distribution of CPU frequency for different numbers of threads with input size $M = 2304$ when running our OpenMP *symm* kernel

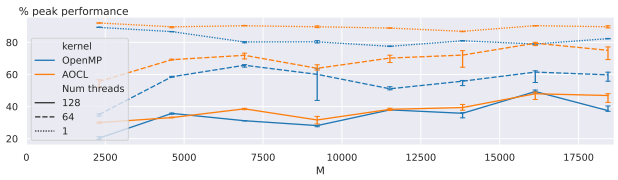


Fig. 5: Comparison of OpenMP *symm* with baselines. Performance vs. Input Size M for various number of threads. The error bar denotes the 95% nonparametric confidence interval of the median.

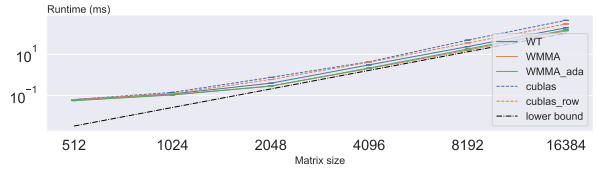
row-major variant of the reference kernel was implemented using $C^T = B^T A^T$. The performance of this row-major reference kernel is indicated as the *cublas_row* line in Fig.6b.

Results. We consider two variants of our kernels to be the best kernels that we've implemented. The runtime and the performance curves of these two kernels are named with WT and WMMA in Fig.6. The WT kernel is the kernel in which we implemented our own version of matrix matrix multiplication while the WMMA kernels are the ones we use Tensor Cores to do the actual computation.

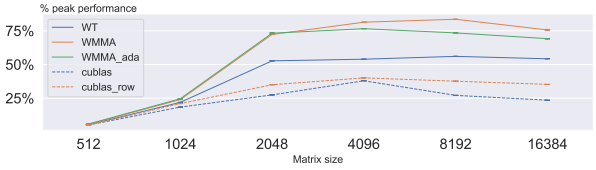
For the non Tensor Core kernel, WT, we did not implement the adaptive loading of A approach in 3.2 in it as we found that this approach barely affect the performance in the kernels that we had implemented before WT. We can see clearly from Fig.6b that we have got better performance than both variants of the cublas kernel. We manged to achieve 56% of RTX 4090's peak performance for single-precision computation with kernel WT. We believe this number can be further improved with better tuning for the kernel's hyper parameters.

For the WMMA kernels, we tried the adaptive loading of A approach as Tensor Core provides us with a native way do to matrix matrix multiplications with operands can be either row or column major. Unfortunately, the non-adaptive approch beats the performance of the adaptive one according to Fig.6b. The non-adaptive WMMA kernel achieved 84% of RTX 4090's peak performance.

We think the reasons for the adaptive loading approch did not work as expected is listed as follows. For non Tensor Core kernels, the cost of doing inplace transpose of blocks



(a) Running time of kernels on different problem sizes



(b) Percentage of peak performance reached

Fig. 6: Runtime and Percentage of peak performance reached of our best CUDA *symm* kernels on different problem sizes. Error bar denote the 95% nonparametric confidence interval of the median. Dashed lines represent the performance of cublas kernels.

of A is low comparing to the actual computation part as such transpose is done in registers. For the WMMA kernels, as the actual computations in Tensor Cores are hidden from us, we can only take guesses for the reason why the adaptive variant have worse performance. It is possible that the actual Tensor Core implementation can only handle case which operand A is column-major while operand B as row-major. If A is given as row-major, it might need to do inplace transpose for operand A every single time which leads to additional cost.

When using Nsight Compute to profile all the kernels, we found that cublas's *symm* consists of three CUDA kernels. In comparison, our kernels only consists of a single CUDA kernel, which has lower overhead comparing to the composited approach used in cublas. On the other hand, the *symm* kernel itself in cublas might not be a well-optimized one. We also tried to benchmark the *gemm* kernel using fully-filled symmetric matrix A and found that *gemm* takes only about half the time to finish comparing to *symm*.

4.3. *gemver*

We first compared our results to baselines in strong scaling and weak scaling configurations, then we showed how the performance varies with different input sizes.

We also established a runtime lower bound under memory bandwidth constraints, employing conservative data transfer estimates for our warm cache configuration. Assuming all vectors can fit in the L2 cache, and the entire L3 cache can be dedicated to a portion of matrix A, we calculated compulsory data transfer as 2 loads and 1 write of the remaining part of A. This totals $3(8 \cdot N^2 - C)$ bytes,

Threads	1	2	4	8	16	32	64	128
Bandwidth	29.4	53.8	107.7	213.1	316.7	332.0	323.6	306.1

Table 1: Peak Memory Bandwidth in GB/s. Measured by stream_mem_avx in likwid-bench. According to[12], threads are spread across NUMA nodes and CCXs to get maximum bandwidth.

with N as the input size and C as the total L3 cache size. Threads were spread across CCXs for a maximal total L3 cache. The maximum cache size available for n threads was the lesser of 512 MiB or $n * 16$ MiB. Since the theoretical peak bandwidth is far from reality we measured the peak memory bandwidth using stream_mem_avx benchmark in likwid-bench 5.3 as shown in Table 1. The runtime lower bound was then derived using the measured peak memory bandwidth and data transfer lower bound.

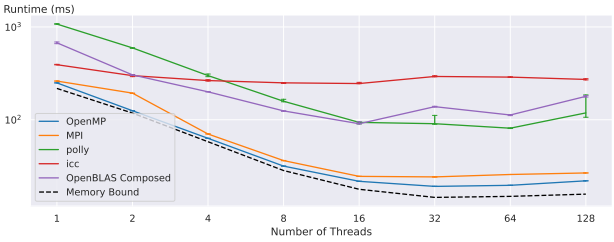


Fig. 7: Strong Scaling for *gemver* compared with baselines. The error bar denotes the 95% nonparametric confidence interval of the median. N = 16384

Strong Scaling. The Fig. 7 showed that both of our kernels surpassed the baselines, exhibiting effective strong scaling from using 1 to 16 threads or processes. The best performance was achieved with 32 threads, aligning with the peak memory bandwidth in Table 1. Using more than 32 threads will not increase total L3 cache size while the total memory bandwidth decreases. Additionally, the best kernel of the OpenMP version ran faster than that of the MPI version. In our early experiments, MPI version achieves better scaling, however, when further techniques such as NUMA is performed, OpenMP can outperform MPI in a single node because of its lower overhead.

Weak Scaling. Given the complexity of our *gemver* problem, we scaled the problem size by $\sqrt{2}$ as the number of processes or threads doubled, maintaining a consistent workload for each process or thread through out all the experiments. The specified configurations for the number of processes or threads its corresponding problem size in each experiment are as follows: 1:2048, 2:2896, 4:4096, 8:5792, 16:8192, 32:11648, 64:16384, and 128:23040. Fig.8 showed that both of our kernels outperformed the baseline, demonstrating effective weak scaling from 1 to 16 threads or processes. Then, the runtime increases since the memory bandwidth is saturating.

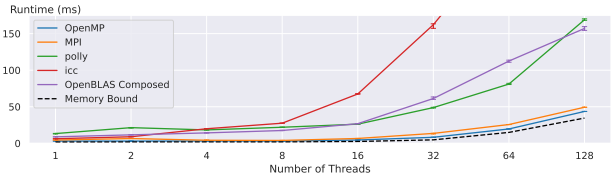


Fig. 8: Weak Scaling for *gemver* compared with baselines. The error bar denotes the 95% nonparametric confidence interval of the median

Performance. We plotted the performance for various input sizes using 8, 32, 128 threads or processes respectively, estimating the total work as $10n^2 + n$, where n represents the input size. The upper bound was derived from the runtime lower bound and the estimated work. Fig. 9 showed that both the OpenMP and MPI kernels exhibited improved performance in the initial stage as the input size increases while still fitting in the L3 cache. However, performance started to decline after reaching its peak when the input size started surpassing the capacity of the L3 cache, and the performance is bounded by the memory bandwidth bound.

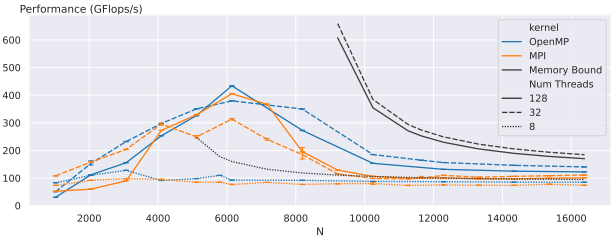


Fig. 9: Performance plot of *gemver*

5. CONCLUSIONS

We have successfully implemented *symm* kernels in CUDA with and without using Tensor Core. Both versions of kernel out-performed the available *symm* kernel in the cublas library. The OpenMP implementation of *symm* kernels demonstrated better performance compared to MKL and OpenBLAS, achieving results comparable to AOCL in our testing environment. Both OpenMP and MPI implementations of *gemver* kernels outperformed the baseline composed from OpenBLAS. However, due to hardware limitation, we did not manage to measure cache misses. Further investigation regarding cache behaviour might be carried out in the future.

6. REFERENCES

[1] Louis-Noel Pouchet and Tomofumi Yuki, "Polybench/c," .

[2] Tobias Grosser, Armin Größlinger, and Christian Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, 2012.

[3] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[4] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, mar 1990.

[5] "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, jun 2002.

[6] Kazushige Goto and Robert A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, may 2008.

[7] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, no. 2, aug 2016.

[8] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1049–1059.

[9] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler, "Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2019, SC '19, Association for Computing Machinery.

[10] Simon Boehm, "How to optimize a cuda matmul kernel for cublas-like performance: A worklog," Dec 2022.

[11] Xing Su and Fei Lei, "Hybrid-grained dynamic load balanced gemm on numa architectures," *Electronics*, vol. 7, no. 12, 2018.

[12] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg, "Memory performance of amd epyc rome and intel cascade lake sp server processors," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. Apr. 2022, ICPE '22, ACM.