DGEMM using FP64 Arithmetic Emulation and FP8 Tensor Cores with Ozaki Scheme

Daichi Mukunoki (Information Technology Center, Nagoya University)

Introduction

- **Context**: AI hardware demands are surging, often sacrificing FP64 performance for low-precision units like Tensor Cores (TCs), even on general-purpose processors.
  - ➢ E.g. Flops/s ratio of FP64 and FP16 on NVIDIA GPUs:
    A100 (2020) 1:16 -> H100 (2022) 1:30 -> B200 (2024) 1:125
- **Problem**: Many scientific computations rely on FP64, but as systems become AI-oriented, sustained performance improvements may become difficult.
- **Solution**: FP64 emulation technology is one approach to enabling FP64 workloads on AI hardware.
- **Ozaki Scheme** [1]: It enabled FP64 matrix multiplication (DGEMM) using FP16TCs [2]. Subsequently, the use of INT8TCs [3] brought a dramatic performance improvement.
- **Recent Trend**: Focus on AI performance is shifting from INT8 to FP8 or lower precision FP, which earlier Ozaki scheme implementations did not explore.
  - ➢ E.g. Throughput ratio of FP8 and INT8: GB200 1:1 -> GB300 30:1
- **Objective**: This work explores the feasibility of DGEMM by Ozaki scheme using FP8TCs and integer-based FP64 arithmetic emulation [4].

Ozaki Scheme [1]

- **Principle**: Decomposes a single high-precision GEMM into a sum of error-free low-precision GEMMs.
- **Process:**
  - **Step 1 (Slicing):** Recursively splits input matrices element-wise to ensure subsequent computations are error-free.
  - **Step 2 (Computation)**: Calculates products of sliced matrices using low-precision GEMMs without rounding errors.
  - **Step 3 (Accumulation)**: Sums the partial results to produce the final GEMM output.
- **Cost**: It requires GEMMs for the square of the number of slices, which increases depending on the following factors:
  1. inner product dimension,
  2. dynamic range of the input values,
  3. precision gap between input and accumulation formats.

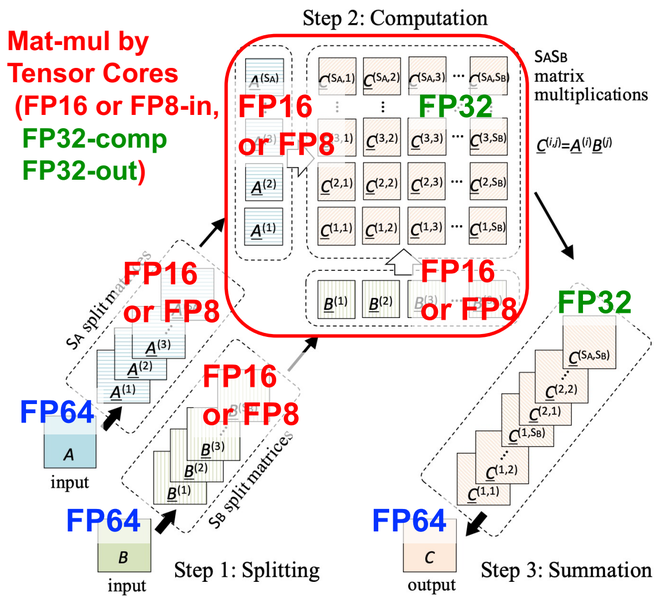

Fig. 1 Schematic of Ozaki scheme.
Note that exponent is separately computed.

Using FP16TC vs. FP8TC

- **Precision Gap & GEMM Count**: The rate at which the required number of GEMMs increases with the inner dimension (k) is determined by the precision gap between the input format (Type2) and the accumulation format (Type3).
- **Advantage of FP32 Accumulation**: TCs with FP32 accumulation provide a large precision gap relative to FP8 input. This prevents the number of GEMMs from increasing, keeping it constant even as k grows.
- **Comparison with FP16**: In contrast, FP16 input has a smaller precision gap to FP32, causing the required GEMMs to rise significantly with k
- **Throughput**: Since FP8TC often deliver 2x that of FP16TC, FP8TCs are faster overall when the ratio of the required number of GEMMs is < 2.

Tab. 1 Minimum number of GEMMs required for DGEMM using TCs with Ozaki scheme. Type2 is the input format and Type3 is the accumulation format.

Type2	FP16	FP16	FP8 (E4M3)	FP8 (E4M3)	FP6 (E3M2)	FP6 (E3M2)
Type3	FP32	FP16	FP32	FP16	FP32	FP16
$k = 8$	25	121	121	121	196	196
16	25	196	121	196	196	196
32	36	196	121	196	196	196
64	36	324	121	324	196	324
128	36	324	121	324	196	324
256	36	729	121	729	196	729
512	49	729	121	729	196	729
1024	49	2809	121	2809	196	2809
2048	64	2809	121	2809	196	2809
4096	64	–	121	–	196	–
8192	81	–	121	–	196	–
16384	81	–	121	–	196	–
32768	121	–	121	–	196	–
65536	121	–	121	–	196	–
131072	196	–	196	–	196	–
262144	196	–	196	–	196	–

FP64 Arithmetic Emulation

- **Motivation**: To examine the feasibility of execution on AI-oriented hardware lacking hardware FP64 units
- **Role in Ozaki scheme**: Slicing and accumulation processes use FP64 operations including addition, multiplication, max, comparison (a>b), and exponent scaling (scalbn).
- **Algorithm**: Processes split mantissas using 32/64-bit integer arithmetic, analogous to pen-and-paper calculation.
- **Accuracy**: Ensures bitwise identical results to hardware FP64 with round-to-nearest-even rounding.
- **Optimization**: Reduces overhead by omitting overflow/underflow handling irrelevant to the scheme.

```
typedef struct {
    uint32_t parts[4];
} uint32x4;

__device__ inline uint32x4 mul_mantissa (uint64_t a, uint64_t b) {
    uint32_t a_low = a & 0xFFFFFFFF;
    uint32_t a_high = a >> 32;
    uint32_t b_low = b & 0xFFFFFFFF;
    uint32_t b_high = b >> 32;

    uint64_t p00 = (uint64_t)a_low * b_low;
    uint64_t p01 = (uint64_t)a_low * b_high;
    uint64_t p10 = (uint64_t)a_high * b_low;
    uint64_t p11 = (uint64_t)a_high * b_high;

    uint64_t middle = p01 + p10;
    uint64_t carry = (middle < p01) ? (1ULL << 32) : 0;

    uint64_t low_result = p00 + ((middle & 0xFFFFFFFF) << 32);
    carry += (low_result < p00) ? 1 : 0;
    uint64_t high_result = p11 + (middle >> 32) + carry;

    uint32x4 result;
    result.parts[0] = low_result & 0xFFFFFFFF;
    result.parts[1] = low_result >> 32;
    result.parts[2] = high_result & 0xFFFFFFFF;
    result.parts[3] = high_result >> 32;
    return result;
}
```

Fig. 2 Part of FP64 multiplication code (mantissa part)

Evaluation

- **Environment**: NVIDIA GeForce RTX 5060 Ti (Blackwell RTX architecture), 16GB, 94.74 TFlops/s on FP8TC, 47.37 TFlops/s on FP16TCs, CUDA 12.9
- **Problem setting**: square matrices (m=n=k), elements in (1,10) (pseudo-uniform) – Corresponds to GEMM counts in **Tab. 1**.
- **Comparison**:
  - **DGEMM-FP8TC**: DGEMM using FP8TCs
  - **DGEMM-FP16TC**: DGEMM using FP16TCs
  - **-FP64emu**: using the FP64 arithmetic emulation in slicing and accumulation.

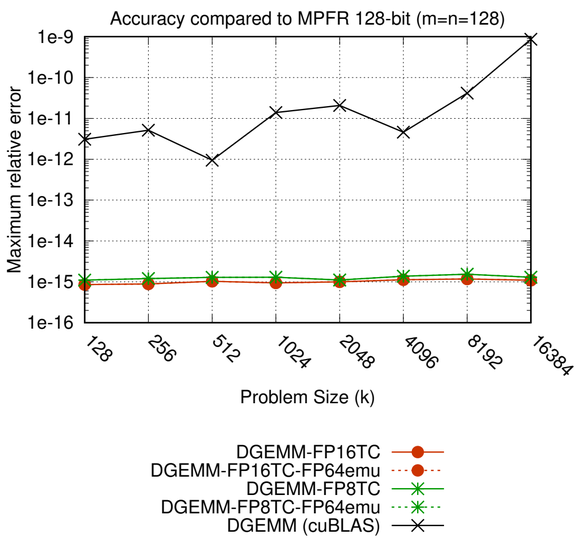

Fig. 3 Accuracy evaluation. Our implementations achieved error levels equivalent to or lower than standard FP64 GEMM.

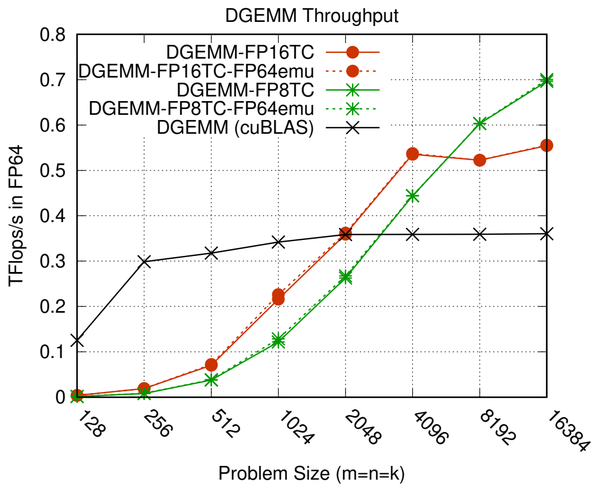

Fig. 4 DGEMM throughput. Flops/s is calculated as $2m^3$ Flops per sec. FP8TC outperforms FP16TC at large sizes ($m \geq 8192$) due to 2x hardware throughput. FP64 arithmetic emulation showed minimal performance overhead.

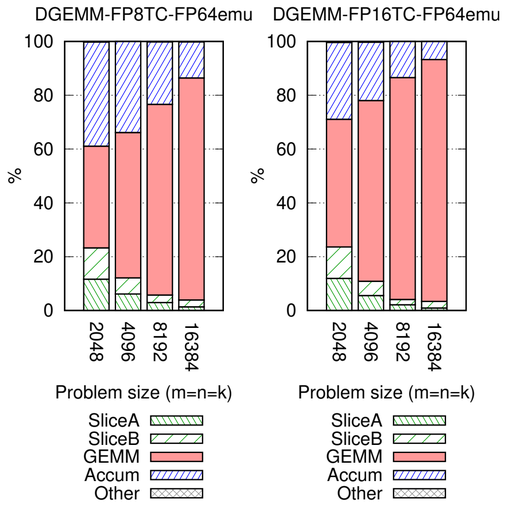

Fig. 5 Execution time breakdown. Since GEMM calculations account for the majority of execution time, FP64 arithmetic emulation does not significantly impact performance.

Conclusion

- In DGEMM emulation using the Ozaki scheme, FP8TC provides competitive performance against FP16TC, and integer-based emulation of FP64 arithmetic achieves sufficient performance.
- FP64 emulation techniques such as the Ozaki scheme are a promising approach for utilizing AI hardware in scientific computing.
- However, data representations for AI are rapidly evolving. It remains unclear whether new formats such as NVFP4 (NVIDIA), HiFloat8 (Huawei), and FP8 (UE8M0, no mantissa) are applicable to scientific workloads or FP64 emulation.
- Toward system integration for AI and scientific computing, researchers in AI, hardware, computer arithmetic (emulation technology), numerical computation, and applications must collaborate to design next-generation systems.

References

1. K. Ozaki, T. Ogita, S. Oishi, S. M. Rump, “Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications”, Numer. Algorithms 59, 1, 2012.
2. D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura, "DGEMM using Tensor Cores, and Its Accurate and Reproducible Versions", Proc. ISC 2020.
3. H. Ootomo, K. Ozaki, R. Yokota, “DGEMM on integer matrix multiplication unit”, The International Journal of High Performance Computing Applications, 2024.
4. D. Mukunoki, DGEMM using FP64 Arithmetic Emulation and FP8 Tensor Cores with Ozaki Scheme, Proc. SCA/HPCAsia 2026 WS - ExHET’26, 2026.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number JP25K24387.