ETH zürich

Integrated Systems Laboratory



EmuGEMM: Fused Tensor Core Kernels for Precision Emulation in Matrix Multiplication

Denghui Lu

A. Maeder · M. Luisier · A. N. Ziogas

SC26 · Chicago · 18 November 2026

Scientific computing runs on NVIDIA GPUs



Summit ORNL

27,648 × V100 · #1 Top500, 2018



Leonardo CINECA

13,824 × A100 · #4 Top500, 2022



Alps CSCS

10,752 × GH200 · runs QuaTrEx

V100 · 2017 — **A100** · 2020 — **H100** · 2022 — 2025 →

For a decade, almost every leading science machine has been a GPU machine.

photos: ORNL, CINECA, CSCS

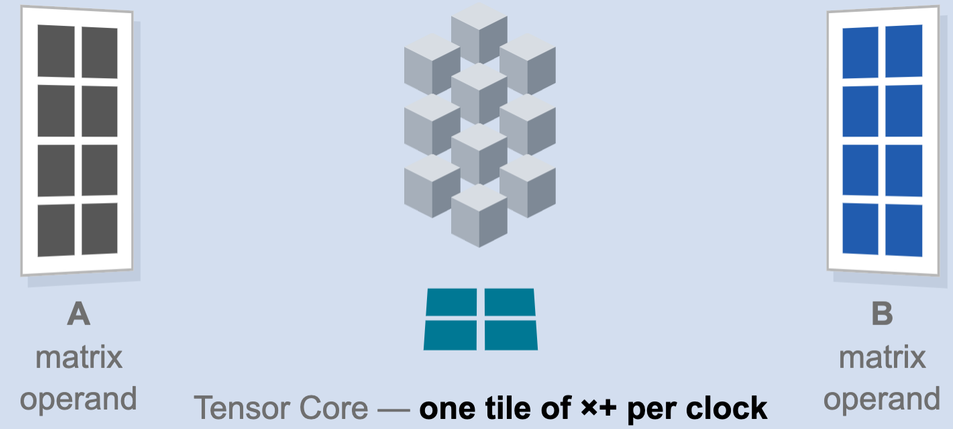
Ranks: TOP500 [1] · system data: ORNL, CINECA, CSCS

LLMs ask the GPU for a very different GEMM

Integrated Systems
Laboratory

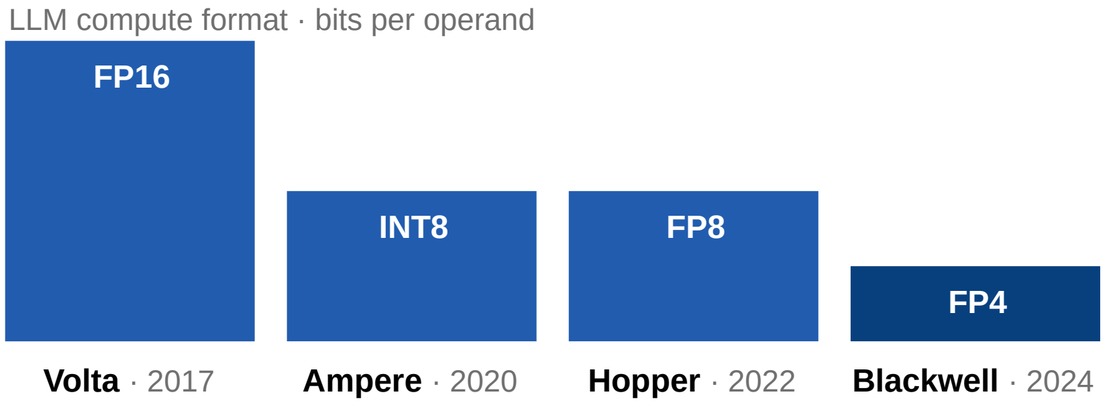
A transformer is dense GEMM

Attention and MLP layers reduce to large dense matrix multiplies — essentially the entire FLOP budget of an LLM. Same operation as simulation. **Completely different demands.**



And it is reshaping the matrix engines

Each generation leads with a smaller format — on GPUs and on every new accelerator. Low-precision GEMM is now what the silicon is built around.



Meanwhile, FP64 tensor peak **67 → 40** Tflop/s

first generation ever to go down

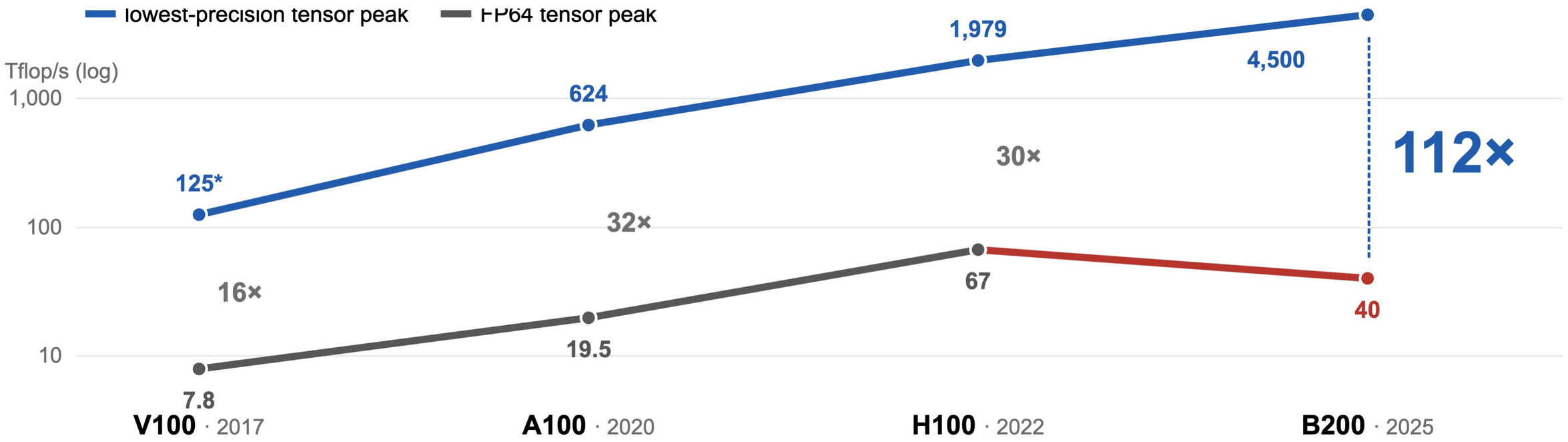
A weight tolerates noise. A wavefunction does not.

The GEMM the hardware optimises for is no longer the GEMM our solvers were written for.

Formats and peaks: NVIDIA datasheets [2]

The result: a 112× precision gap

Integrated Systems
Laboratory



Low precision: ×36 in eight years. FP64: down — the free ride is over.

MEASURED CUBLAS FP64 GEMM

58 → **36** Tflop/s −38%

NVIDIA datasheets [2], dense peaks, log scale · *V100: FP16 (no INT8 tensor path)

Measured cuBLAS DGEMM, M = N = K = 16,384: Lu et al. [4]

Can the codes you run today use those units?

Plane-wave DFT

Subspace rotations and exact-exchange builds are large dense complex GEMMs.

VASP · Quantum ESPRESSO · CPMD · ABINIT

Quantum chemistry

Tensor contractions reduce to large dense DGEMM.

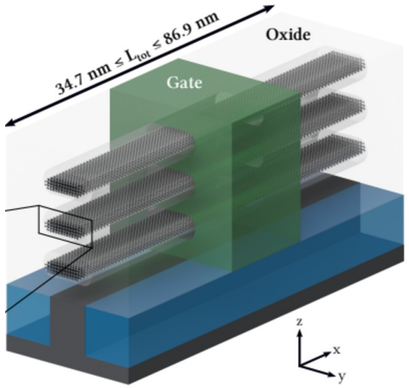
ORCA · PySCF · NWChem · TeraChem

NEGF quantum transport

Essentially the whole solver is dense complex linear algebra.

QuaTrEx · OMEN · NEMO5

? ? ?



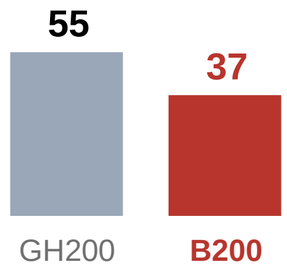
Our worked example: QuaTrEx (DFT + NEGF + GW)

42,240-atom Si nanoribbon transistor — 1.5 nm body, 5.0 nm wide, L_{tot} 34.7–86.9 nm · 1.15 Eflop/s sustained FP64 on Frontier · Gordon Bell Prize finalist, SC25.

Its hot kernel is ZGEMM — Green's functions are complex.

QuaTrEx: Vetsch et al. [3] · ZGEMM: cuBLAS, M = N = K = 16,384, measured in [4]

CUBLAS ZGEMM · TFLOP/S



−33% on the newer GPU

Emulation: rebuild FP64 from exact INT8 products

Integrated Systems
Laboratory

Cut every entry into integer slices, multiply the slices pairwise on the integer Tensor Cores — where nothing rounds — and put the pieces back together once, in FP64. **Slices kept set the accuracy; slice products run set the cost.**

1 · SLICE



Each entry → p byte slices on one exponent grid. The only place error enters.

→

2 · MULTIPLY — EXACTLY



slice pairs → INT8 GEMMs
10 of 16 carry weight

Each slice pair is one INT8 GEMM, exact in INT32 — the units getting faster.

→

3 · RECONSTRUCT



Shift each product by its weight, sum once in FP64.

Emulation: rebuild FP64 from exact INT8 products

Integrated Systems
Laboratory

Cut every entry into integer slices, multiply the slices pairwise on the integer Tensor Cores — where nothing rounds — and put the pieces back together once, in FP64. **Slices kept set the accuracy; slice products run set the cost.**

1 · SLICE



Each entry → p byte slices on one exponent grid. The only place error enters.

→

2 · MULTIPLY — EXACTLY



slice pairs → INT8 GEMMs
10 of 16 carry weight

Each slice pair is one INT8 GEMM, exact in INT32 — the units getting faster.

→

3 · RECONSTRUCT



Shift each product by its weight, sum once in FP64.

PRIOR WORK · A 14-YEAR-OLD IDEA

2012

The idea: error-free splitting

Ozaki, Ogita, Oishi, Rump [5] — split, multiply exactly, sum once.

2020 – 2024

Onto Tensor Cores

Mukunoki et al. [6] (FP16 TC, OzBLAS) → Ootomo, Ozaki, Yokota [7] (INT8, ozIMMU).

2025

Faster, modular, in cuBLAS

Uchino et al. [8, 10] (GEMMul8) · Ozaki-II [9] (CRT scheme) · cuBLAS ships Scheme-I emulation [11].

2026

EmuGEMM · this talk

Same algorithm, open source — near the full speed of the integer tensor cores.

A ready algorithm is not a solved problem — as FlashAttention [12] was for attention.

[5] Ozaki 2012 · [6] Mukunoki 2020 · [7] Ootomo 2024 · [8–10] Uchino, Ozaki 2025–26 · [11] Schwarz 2026 · [12] Dao 2022

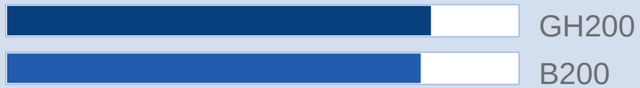
EmuGEMM: FP64-class accuracy at INT8 speed

Integrated Systems Laboratory

Same algorithm, new execution: fused integer Tensor Core kernels that rebuild high-precision GEMM from exact INT8 products. Hopper (SM90) and Blackwell (SM100), open source.

83%

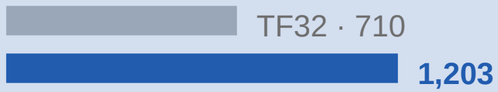
of INT8 peak on GH200



1,639 Top/s sustained · 3,654 on B200 (81%)

1.7×

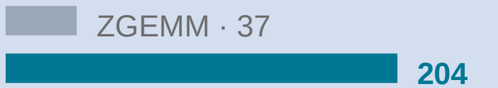
faster than cuBLAS TF32



Tflop/s on B200, 16k, comparable accuracy · 1.4× on GH200

5.5×

faster than cuBLAS ZGEMM



Tflop/s on B200 at FP64-class accuracy · 2.3× on GH200

Two schemes, one continuous precision knob

EmuGEMM-I covers TF32 to FP32; EmuGEMM-II takes over above FP32 and adds a fused 3M complex kernel. Together they Pareto-dominate cuBLAS emulation across the whole precision range.

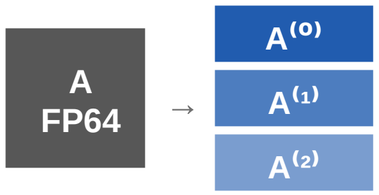
The advantage grows on Blackwell — the more the hardware favours low precision, the better emulation does.

All results: Lu et al. [4] · effective throughput, end-to-end

Scheme I, step by step: slice → multiply → reduce → reconstruct

Integrated Systems Laboratory

STEP 1 · SLICE



$$A \cong (A^{(0)} + 2^{-8}A^{(1)} + 2^{-16}A^{(2)}) \cdot S_A$$

Scale rows of A, columns of B onto one exponent grid — the scales wait outside. Example: p = 3 byte slices (FP64 needs up to 8).

STEP 2 · MULTIPI

shade = weight · dashed = skipped

INT8 × INT8 → INT32, exact

Every pair is one small integer GEMM. Pairs below the precision target (i + j ≥ p) are never computed: only 6 of 9 run here, 36 of 64 at p = 8.

STEP 3 · REDUC

Σ per anti-diagonal → INT64

Same shade = same weight = one anti-diagonal, one integer sum each. Only the p above the cut exist: $C^{(0)}$ $C^{(1)}$ $C^{(2)}$, the mantissa slices of C.

STEP 4 · RECONSTRU

$$C = (C^{(0)} + 2^{-8}C^{(1)} + 2^{-16}C^{(2)}) \cdot S_A \cdot S_B$$

Shift each slice by its weight, add once in high precision, multiply the scales back on — the only floating-point arithmetic.

FP64 in → INT8 slices → INT32 products → INT64 sums → FP64 out

Integers end-to-end between split and sum — rounding cannot enter.

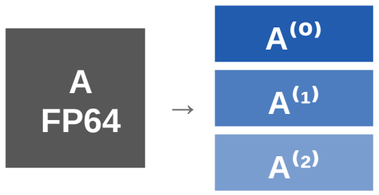
Four-step overview redrawn after NVIDIA GTC Ozaki-scheme material (DGEMM on integer Tensor Cores) — drawn for p = 3; FP64-class accuracy uses p = 8.

Scheme I: Ozaki et al. [5] · on INT8 Tensor Cores: Ootomo et al. [7] · in cuBLAS: Schwarz et al. [11]

Scheme I, step by step: slice → multiply → reduce → reconstruct

Integrated Systems Laboratory

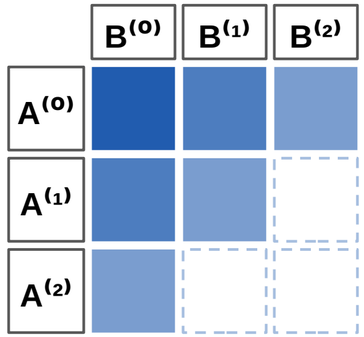
STEP 1 · SLICE



$$A \cong (A^{(0)} + 2^{-8}A^{(1)} + 2^{-16}A^{(2)}) \cdot S_A$$

Scale rows of A, columns of B onto one exponent grid — the scales wait outside. Example: p = 3 byte slices (FP64 needs up to 8).

STEP 2 · MULTIPLY



shade = weight · dashed = skipped

INT8 × INT8 → INT32, exact

Every pair is one small integer GEMM. Pairs below the precision target (i + j ≥ p) are never computed: only 6 of 9 run here, 36 of 64 at p = 8.

STEP 3 · REDUC

Σ per anti-diagonal → INT64

Same shade = same weight = one anti-diagonal, one integer sum each. Only the p above the cut exist: $C^{(0)}$ $C^{(1)}$ $C^{(2)}$, the mantissa slices of C.

STEP 4 · RECONSTRU

$$C = (C^{(0)} + 2^{-8}C^{(1)} + 2^{-16}C^{(2)}) \cdot S_A \cdot S_B$$

Shift each slice by its weight, add once in high precision, multiply the scales back on — the only floating-point arithmetic.

FP64 in → INT8 slices → INT32 products → INT64 sums → FP64 out

Integers end-to-end between split and sum — rounding cannot enter.

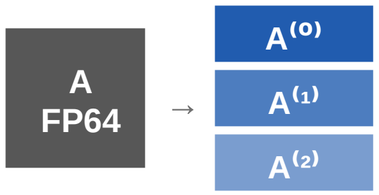
Four-step overview redrawn after NVIDIA GTC Ozaki-scheme material (DGEMM on integer Tensor Cores) — drawn for p = 3; FP64-class accuracy uses p = 8.

Scheme I: Ozaki et al. [5] · on INT8 Tensor Cores: Ootomo et al. [7] · in cuBLAS: Schwarz et al. [11]

Scheme I, step by step: slice → multiply → reduce → reconstruct

Integrated Systems Laboratory

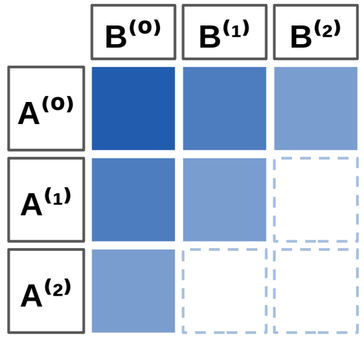
STEP 1 · SLICE



$$A \cong (A^{(0)} + 2^{-8}A^{(1)} + 2^{-16}A^{(2)}) \cdot S_A$$

Scale rows of A, columns of B onto one exponent grid — the scales wait outside. Example: p = 3 byte slices (FP64 needs up to 8).

STEP 2 · MULTIPLY

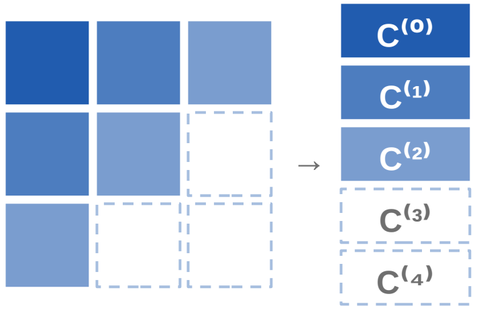


shade = weight · dashed = skipped

INT8 × INT8 → INT32, exact

Every pair is one small integer GEMM. Pairs below the precision target (i + j ≥ p) are never computed: only 6 of 9 run here, 36 of 64 at p = 8.

STEP 3 · REDUCE



Σ per anti-diagonal → INT64

Same shade = same weight = one anti-diagonal, one integer sum each. Only the p above the cut exist: $C^{(0)}$ $C^{(1)}$ $C^{(2)}$, the mantissa slices of C.

STEP 4 · RECONSTRU

$$C = (C^{(0)} + 2^{-8}C^{(1)} + 2^{-16}C^{(2)}) \cdot S_A \cdot S_B$$

Shift each slice by its weight, add once in high precision, multiply the scales back on — the only floating-point arithmetic.

FP64 in → INT8 slices → INT32 products → INT64 sums → FP64 out

Integers end-to-end between split and sum — rounding cannot enter.

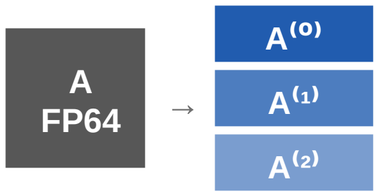
Four-step overview redrawn after NVIDIA GTC Ozaki-scheme material (DGEMM on integer Tensor Cores) — drawn for p = 3; FP64-class accuracy uses p = 8.

Scheme I: Ozaki et al. [5] · on INT8 Tensor Cores: Ootomo et al. [7] · in cuBLAS: Schwarz et al. [11]

Integrated Systems Laboratory

Scheme I, step by step: slice → multiply → reduce → reconstruct

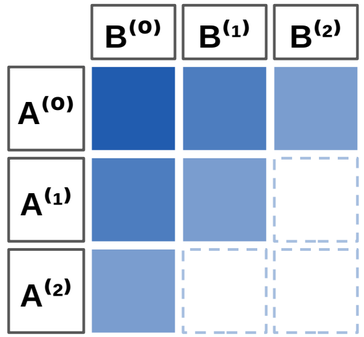
STEP 1 · SLICE



$$A \cong (A^{(0)} + 2^{-8}A^{(1)} + 2^{-16}A^{(2)}) \cdot S_A$$

Scale rows of A, columns of B onto one exponent grid — the scales wait outside. Example: p = 3 byte slices (FP64 needs up to 8).

STEP 2 · MULTIPLY

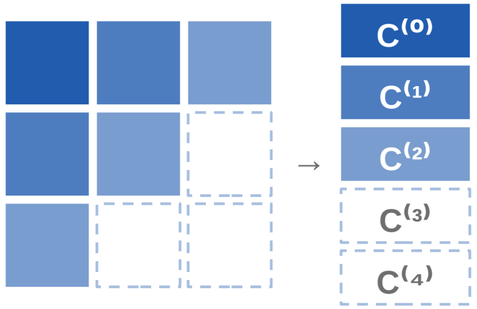


shade = weight · dashed = skipped

INT8 × INT8 → INT32, exact

Every pair is one small integer GEMM. Pairs below the precision target (i + j ≥ p) are never computed: only 6 of 9 run here, 36 of 64 at p = 8.

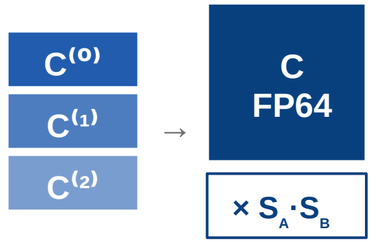
STEP 3 · REDUCE



Σ per anti-diagonal → INT64

Same shade = same weight = one anti-diagonal, one integer sum each. Only the p above the cut exist: $C^{(0)}$ $C^{(1)}$ $C^{(2)}$, the mantissa slices of C.

STEP 4 · RECONSTRUCT



$$C = (C^{(0)} + 2^{-8}C^{(1)} + 2^{-16}C^{(2)}) \cdot S_A \cdot S_B$$

Shift each slice by its weight, add once in high precision, multiply the scales back on — the only floating-point arithmetic.

FP64 in → **INT8 slices** → **INT32 products** → **INT64 sums** → **FP64 out**

Integers end-to-end between split and sum — rounding cannot enter.

Four-step overview redrawn after NVIDIA GTC Ozaki-scheme material (DGEMM on integer Tensor Cores) — drawn for p = 3; FP64-class accuracy uses p = 8.

Scheme I: Ozaki et al. [5] · on INT8 Tensor Cores: Ootomo et al. [7] · in cuBLAS: Schwarz et al. [11]

Integrated Systems
Laboratory

Scheme II: don’t slice the number — take its remainders

STEP 1 · SPLITTING

Reduce mod p coprime moduli

every entry → p one-byte residues

A B — FP64 inputs, scaled to integers

A mod m_1	B mod m_1	$m_1 = 251$
A mod m_2	B mod m_2	$m_2 = 241$
A mod m_3	B mod m_3	$m_3 = 239$

Not pieces of the number — remainders of it. Each residue fits in 8 bits; p coprime moduli in place of p slices.

STEP 2 · COMPUTATI(

One GEMM per modulus

matched pairs only — never crossed

$A^{(1)} \cdot B^{(1)} \rightarrow$ MMA $\rightarrow C^{(1)} \bmod m_1$

$A^{(2)} \cdot B^{(2)} \rightarrow$ MMA $\rightarrow C^{(2)} \bmod m_2$

$A^{(3)} \cdot B^{(3)} \rightarrow$ MMA $\rightarrow C^{(3)} \bmod m_3$

moduli never mix — only the **diagonal** exists. Scheme I needed a triangle: p(p+1)/2 GEMMs. Here: **p GEMMs — linear.**

One INT8 GEMM per modulus. INT32 accumulation is exact; the mod runs in the registers — slide 26.

STEP 3 · RECONSTRUCTI

CRT: remainders pin the number

one exact combine per element

x mod 3	= 2
x mod 5	= 3
x mod 7	= 2

⇒ x = **23** — the only value below 3·5·7 = 105

The Chinese Remainder Theorem, 3rd century: remainders under coprime moduli pin the number down **uniquely**. Keep C below $m_1 m_2 m_3$ and the reconstruction is exact — no summation of slices at all.

Reconstruction, not summation.

Scheme II: Ozaki, Uchino, Imamura [9] · GEMMul8 [10]

Integrated Systems
Laboratory

Scheme II: don’t slice the number — take its remainders

STEP 1 · SPLITTING

Reduce mod p coprime moduli

every entry → p one-byte residues

A B — FP64 inputs, scaled to integers

A mod m_1	B mod m_1	m_1 = 251
A mod m_2	B mod m_2	m_2 = 241
A mod m_3	B mod m_3	m_3 = 239

Not pieces of the number — remainders of it. Each residue fits in 8 bits; p coprime moduli in place of p slices.

→

STEP 2 · COMPUTATION

One GEMM per modulus

matched pairs only — never crossed

$A^{(1)} \cdot B^{(1)} \rightarrow$ MMA $\rightarrow C^{(1)}$ mod m_1

$A^{(2)} \cdot B^{(2)} \rightarrow$ MMA $\rightarrow C^{(2)}$ mod m_2

$A^{(3)} \cdot B^{(3)} \rightarrow$ MMA $\rightarrow C^{(3)}$ mod m_3



moduli never mix — only the **diagonal** exists. Scheme I needed a triangle: p(p+1)/2 GEMMs. Here: **p GEMMs — linear.**

One INT8 GEMM per modulus. INT32 accumulation is exact; the mod runs in the registers — slide 26.

→

STEP 3 · RECONSTRUCTI

CRT: remainders pin the number

one exact combine per element

x mod 3 = 2
x mod 5 = 3
x mod 7 = 2

⇒ x = **23** — the only value below 3·5·7 = 105

The Chinese Remainder Theorem, 3rd century: remainders under coprime moduli pin the number down **uniquely**. Keep C below $m_1 m_2 m_3$ and the reconstruction is exact — no summation of slices at all.

Reconstruction, not summation.

Scheme II: Ozaki, Uchino, Imamura [9] · GEMMul8 [10]

Scheme II: don’t slice the number — take its remainders

Integrated Systems Laboratory

STEP 1 · SPLITTING

Reduce mod p coprime moduli

every entry → p one-byte residues

A B — FP64 inputs, scaled to integers

A mod m_1	B mod m_1	m_1 = 251
A mod m_2	B mod m_2	m_2 = 241
A mod m_3	B mod m_3	m_3 = 239

Not pieces of the number — remainders of it. Each residue fits in 8 bits; p coprime moduli in place of p slices.

→

STEP 2 · COMPUTATION

One GEMM per modulus

matched pairs only — never crossed

$A^{(1)} \cdot B^{(1)} \rightarrow$ MMA $\rightarrow C^{(1)}$ mod m_1

$A^{(2)} \cdot B^{(2)} \rightarrow$ MMA $\rightarrow C^{(2)}$ mod m_2

$A^{(3)} \cdot B^{(3)} \rightarrow$ MMA $\rightarrow C^{(3)}$ mod m_3



moduli never mix — only the **diagonal** exists. Scheme I needed a triangle: p(p+1)/2 GEMMs. Here: **p GEMMs — linear.**

One INT8 GEMM per modulus. INT32 accumulation is exact; the mod runs in the registers — slide 26.

→

STEP 3 · RECONSTRUCTION

CRT: remainders pin the number

one exact combine per element

x mod 3	**= 2**
x mod 5	**= 3**
x mod 7	**= 2**

⇒ x = **23** — the only value below 3·5·7 = 105

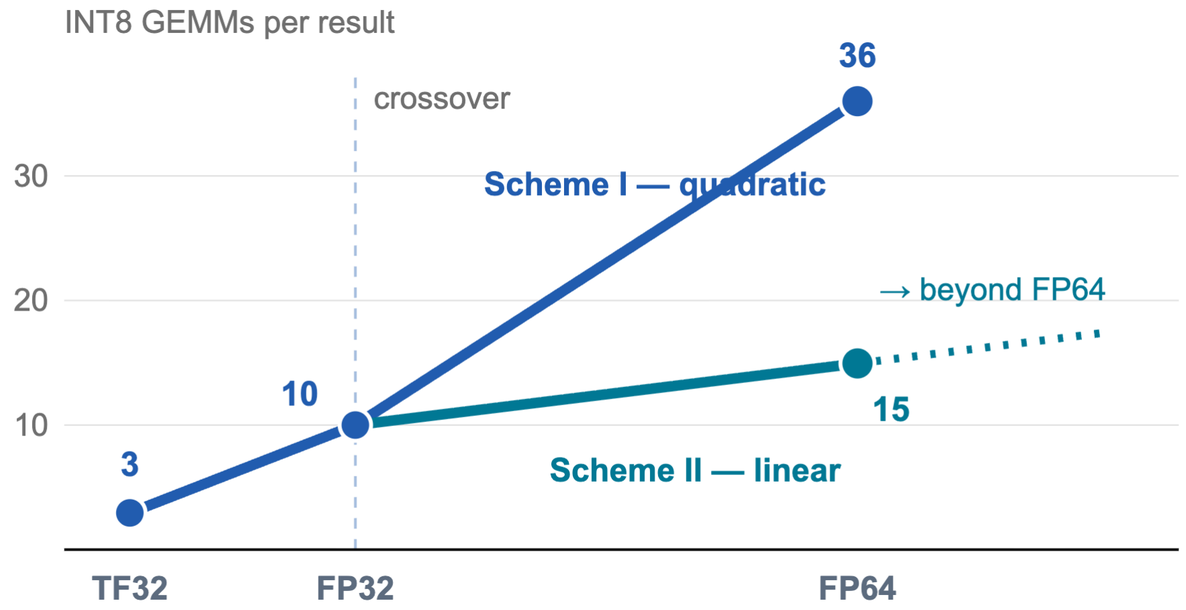
The Chinese Remainder Theorem, 3rd century: remainders under coprime moduli pin the number down **uniquely**. Keep C below $m_1 m_2 m_3$ and the reconstruction is exact — no summation of slices at all.

Reconstruction, not summation.

p residues → p exact GEMMs → one exact reconstruction. The cost is linear in p.

Two schemes, one crossover: linear in p above FP32

Integrated Systems
Laboratory



36 vs 15

INT8 GEMMs at FP64-class — 2.4× less compute

The price: reconstruction runs as a separate pass — included in every number shown. One difference: Scheme I is adaptive — add a slice, reuse every GEMM already done; Scheme II picks its moduli up front.

Below FP32: Scheme I. Above: Scheme II. One library, switched where the mathematics says.

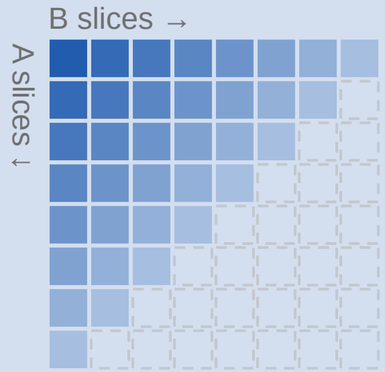
GEMM counts: Scheme I [5], Scheme II [9] · EmuGEMM design: [4]

Integrated Systems
Laboratory

The cost is the round trips, not the maths

(a) The obvious way — one kernel per tile

36 launches



solid tile = one INT8 GEMM, shade = weight $2^{-8(i+j)}$ · dashed = beyond the FP64 target, never computed **Each tile is its own kernel: its INT32 partial goes out to GMEM — 4 bytes per 1-byte operand — and comes straight back to be summed.**



GMEM 36 INT32 partials — written, then read back

72 crossings — and the only mathematics they buy is a summation

(b) The fused way — the same 36 tiles, one kernel

1 launch

chip boundary · nothing crosses until the epilogue

same 36 tiles, one launch **Products of equal weight share an accumulator in the registers; the shift-reduce runs in the epilogue — the sum never leaves the chip.**

GMEM 1 FP64 result — written once

1 write — operand traffic grows with p, not p^2

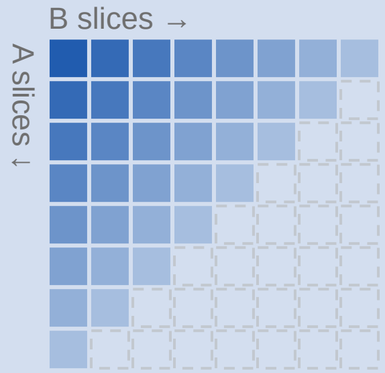
Traffic analysis: Lu et al. [4]

Integrated Systems
Laboratory

The cost is the round trips, not the maths

(a) The obvious way — one kernel per tile

36 launches



solid tile = one INT8 GEMM, shade = weight $2^{-8(i+j)}$ · dashed = beyond the FP64 target, never computed **Each tile is its own kernel: its INT32 partial goes out to GMEM — 4 bytes per 1-byte operand — and comes straight back to be summed.**

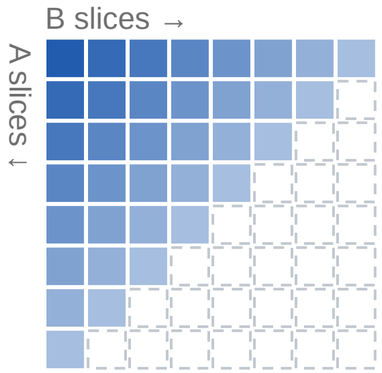
GMEM 36 INT32 partials — written, then read back

72 **crossings** — and the only mathematics they buy is a summation

(b) The fused way — the same 36 tiles, one kernel

1 launch

chip boundary · nothing crosses until the epilogue



same 36 tiles, one launch **Products of equal weight share an accumulator in the registers; the shift-reduce runs in the epilogue — the sum never leaves the chip.**

GMEM 1 FP64 result — written once

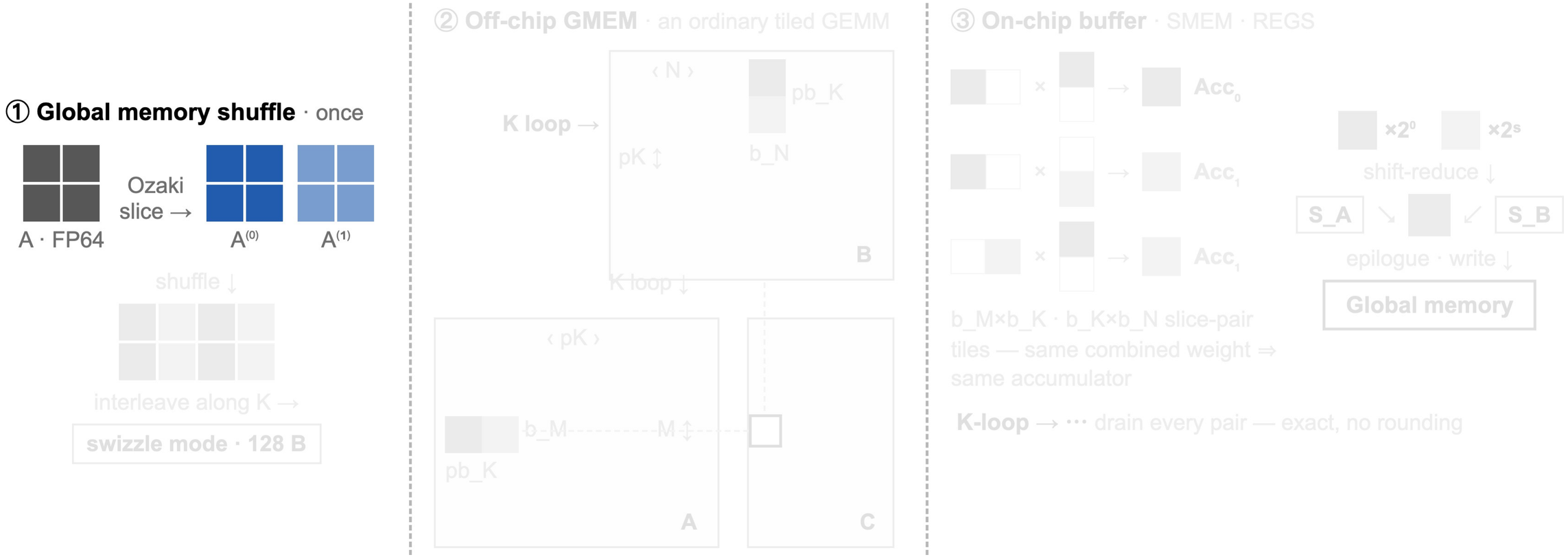
1 **write** — operand traffic grows with p, not p^2

Fusion is not an optimisation — it is the prerequisite.

at p = 8: 4.5× more work per byte moved · zero INT32 traffic

Traffic analysis: Lu et al. [4]

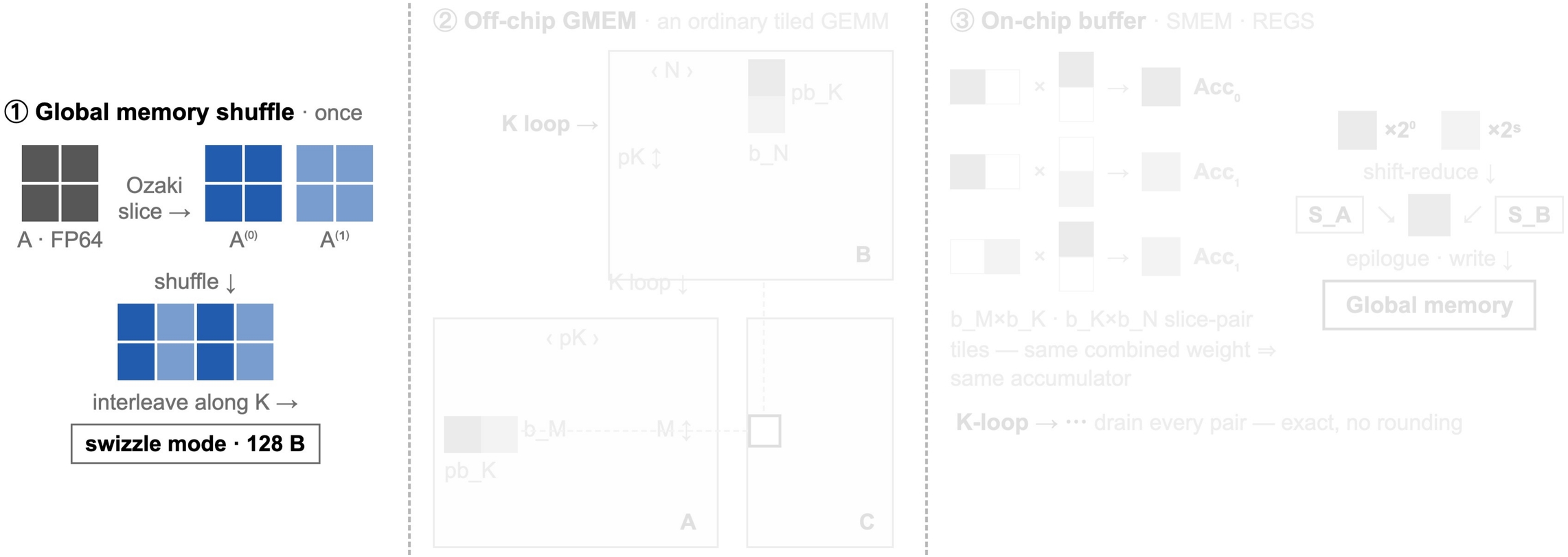
Slice, then shuffle: slices interleave along K in GMEM



One shuffle, once, in global memory — after it, all p slices travel as one matrix.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

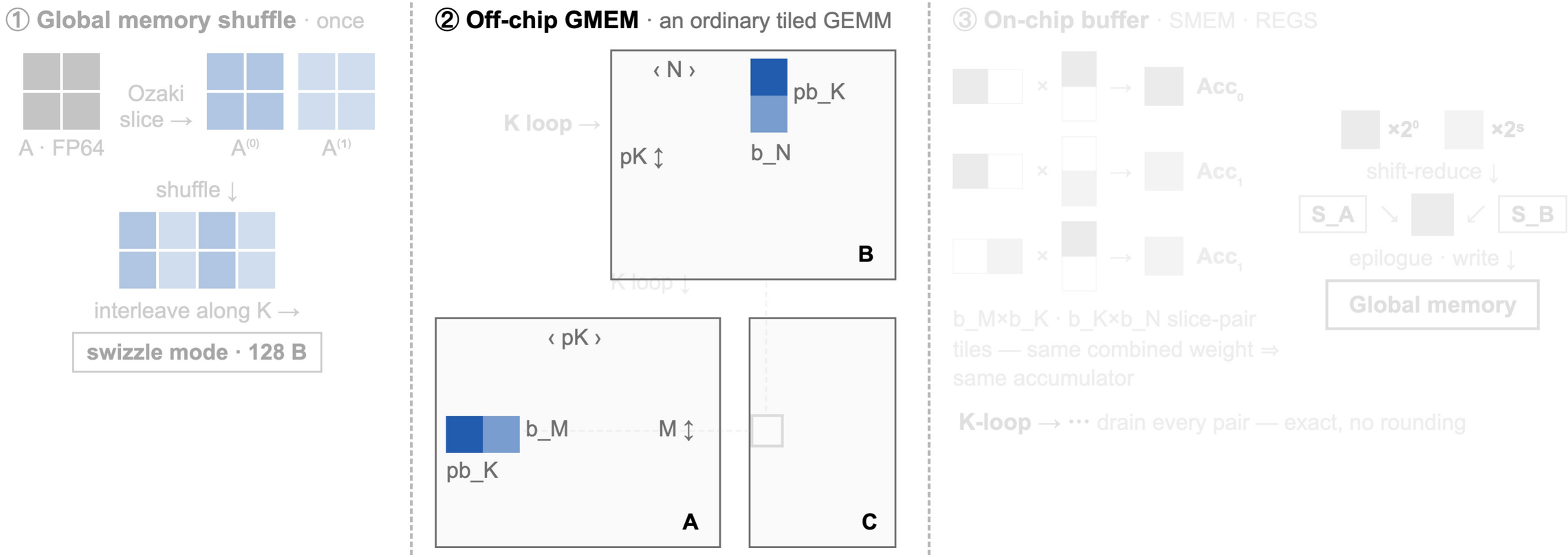
Slice, then shuffle: slices interleave along K in GMEM



One shuffle, once, in global memory — after it, all p slices travel as one matrix.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

Off-chip, it is an ordinary tiled GEMM — pK wide

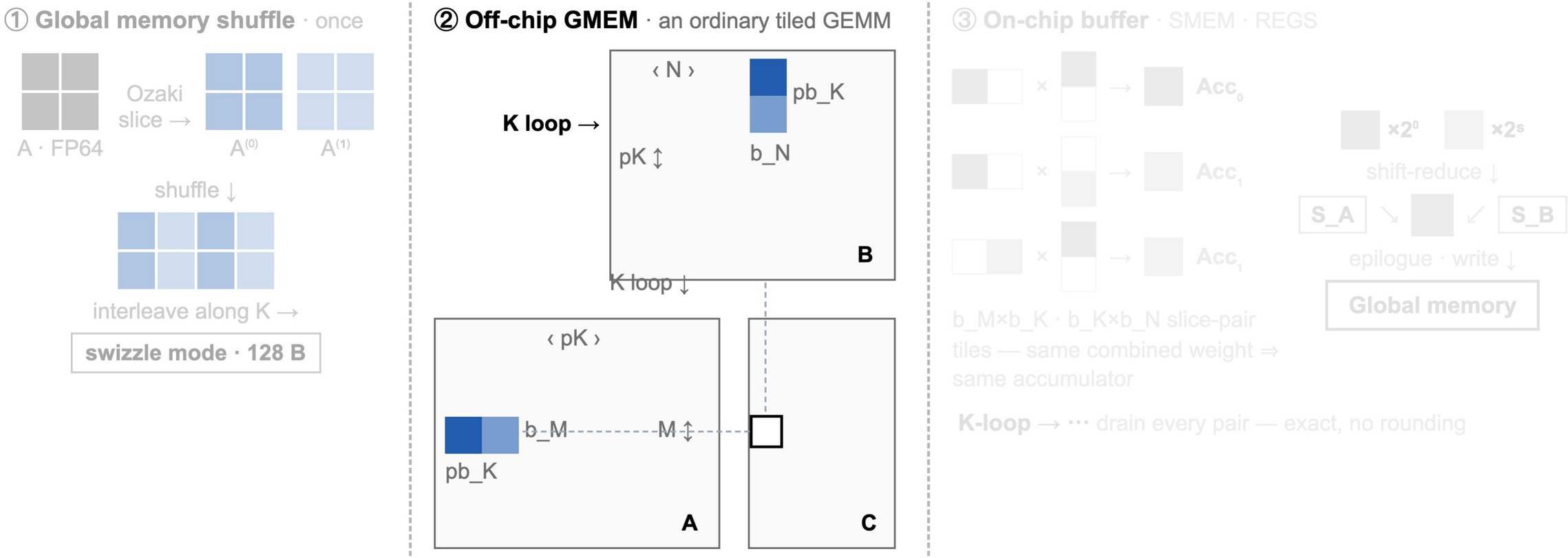


Nothing special to schedule: the emulation cost lives inside the K dimension. C is untouched until the epilogue.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

Lu et al. [4] · built on DeepGEMM [13] and CuTe / CUTLASS [14, 15]

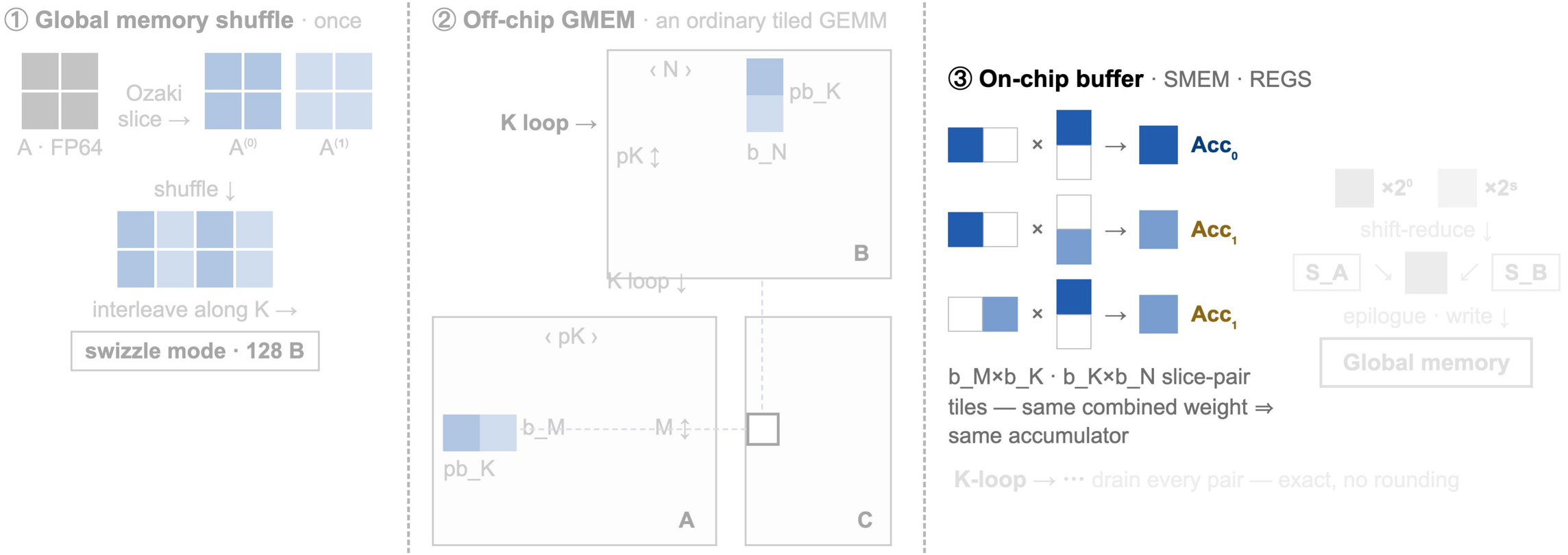
Off-chip, it is an ordinary tiled GEMM — pK wide



Nothing special to schedule: the emulation cost lives inside the K dimension. C is untouched until the epilogue.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

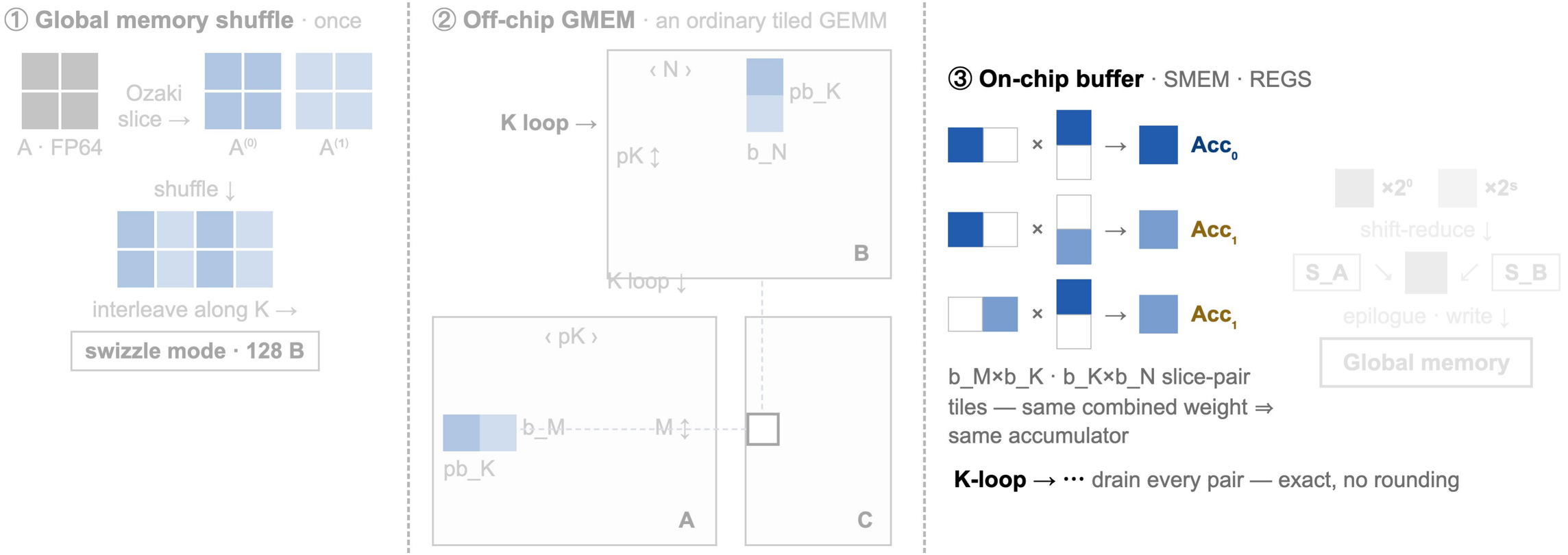
On chip: same-weight products share an accumulator



Accumulators fill exactly; one shift-reduce, scales in the epilogue, one write. Floating point happens once.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

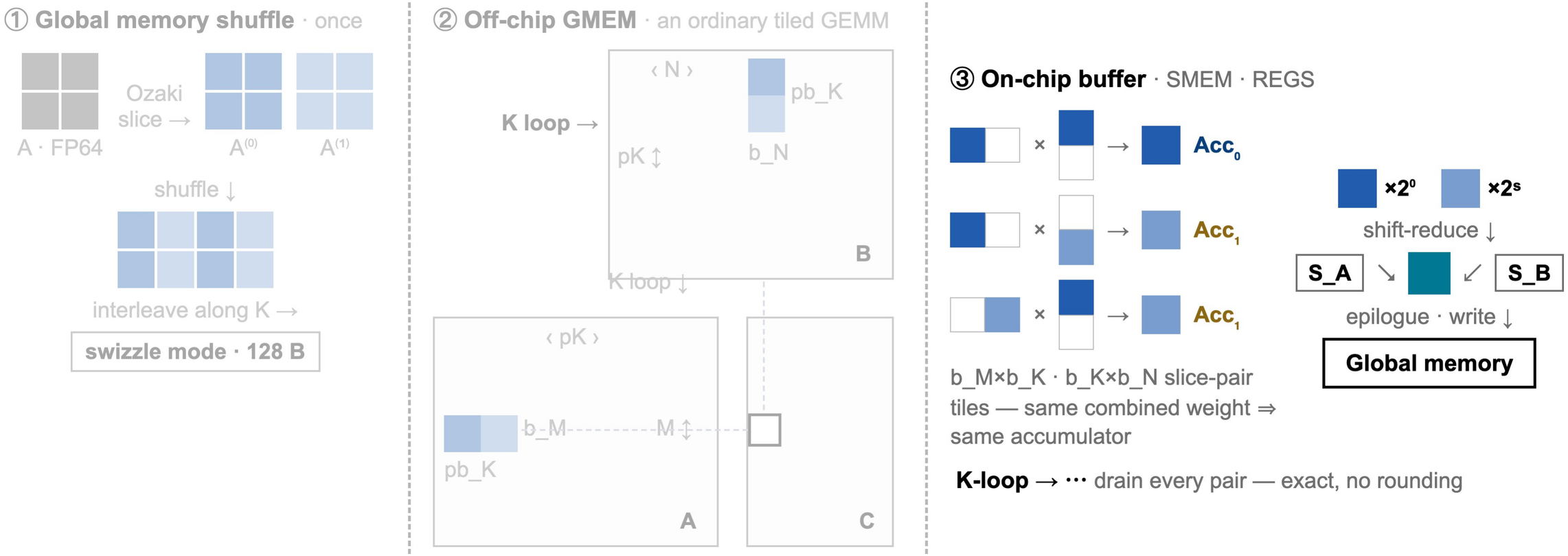
On chip: same-weight products share an accumulator



Accumulators fill exactly; one shift-reduce, scales in the epilogue, one write. Floating point happens once.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

On chip: same-weight products share an accumulator

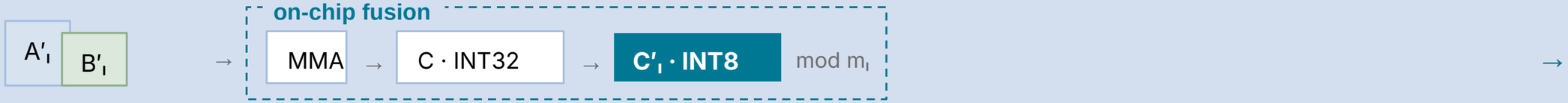


Accumulators fill exactly; one shift-reduce, scales in the epilogue, one write. Floating point happens once.

EmuGEMM-I kernel data flow — redrawn from the paper figure; original not reused.

Making Scheme II fast: reduce in the registers

(a) Real GEMM — one kernel, start to finish, on chip



the remainder costs a handful of integer instructions in the epilogue — the next tile is already on its way in · one byte out: no 4× write amplification

(b) Complex 3M GEMM — four multiplications become three

$C'_{re} = T_1 - T_2$

$C'_{im} = T_3 - T_1 - T_2$

$A'_s = A'_{re} + A'_{im}$ — the third product carries both cross terms; two integer subtractions pull them out. a quarter of the complex work, free

exactly

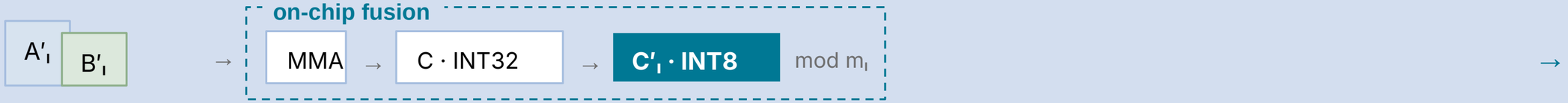
INT8 → INT32 (exact) → INT8 (mod m_l) → GMEM

Caveat, up front: CRT reconstruction is a separate kernel — all reported numbers are end-to-end.

3M for Scheme II: Uchino et al. [16] · fused kernels: Lu et al. [4]

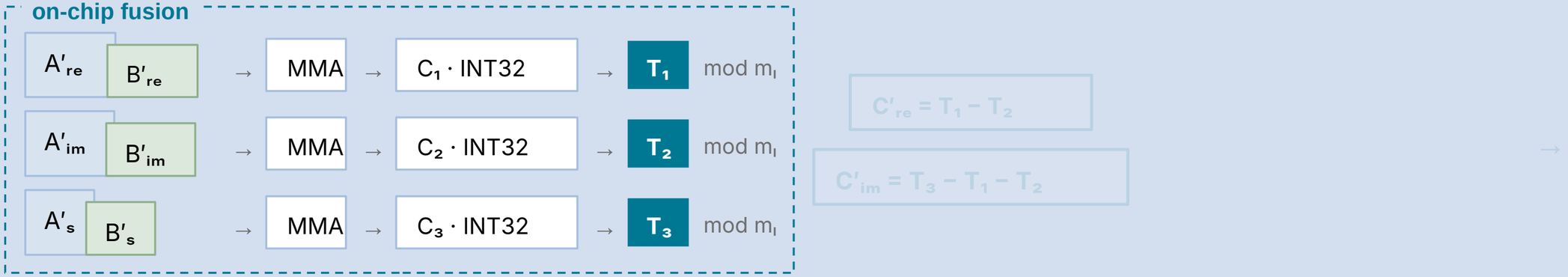
Making Scheme II fast: reduce in the registers

(a) Real GEMM — one kernel, start to finish, on chip



the remainder costs a handful of integer instructions in the epilogue — the next tile is already on its way in · one byte out: no 4× write amplification

(b) Complex 3M GEMM — four multiplications become three



A′s = A′re + A′im — the third product carries both cross terms; two integer subtractions pull them out, **exactly**

a quarter of the complex work, free

GMEM

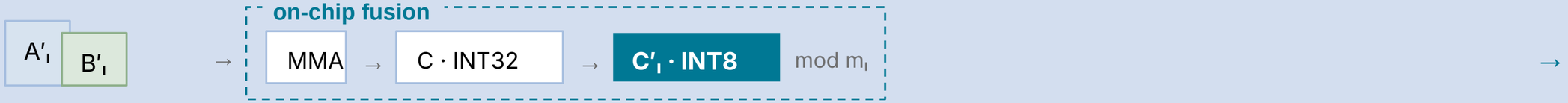
INT8 → INT32 (exact) → INT8 (mod mₗ) → GMEM

Caveat, up front: CRT reconstruction is a separate kernel — all reported numbers are end-to-end.

3M for Scheme II: Uchino et al. [16] · fused kernels: Lu et al. [4]

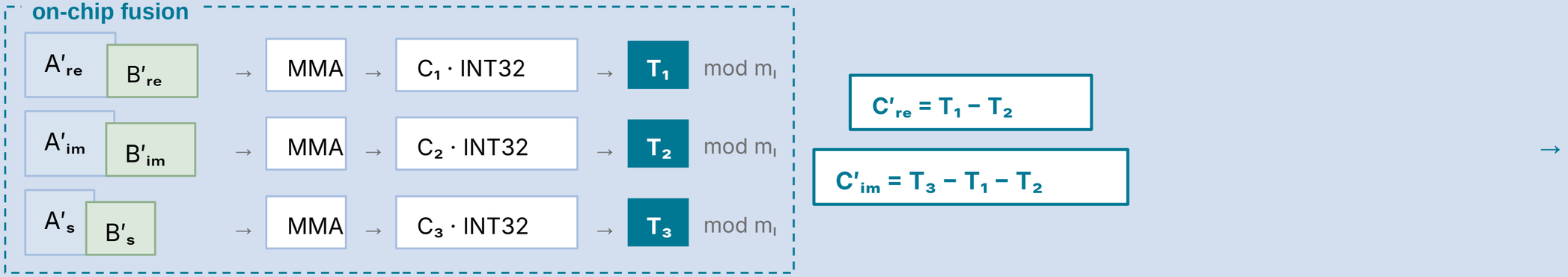
Making Scheme II fast: reduce in the registers

(a) Real GEMM — one kernel, start to finish, on chip



the remainder costs a handful of integer instructions in the epilogue — the next tile is already on its way in · one byte out: no 4× write amplification

(b) Complex 3M GEMM — four multiplications become three



$A'_s = A'_{re} + A'_{im}$ — the third product carries both cross terms; two integer subtractions pull them out, **exactly**

a quarter of the complex work, free

INT8 → INT32 (exact) → INT8 (mod m_l) → GMEM

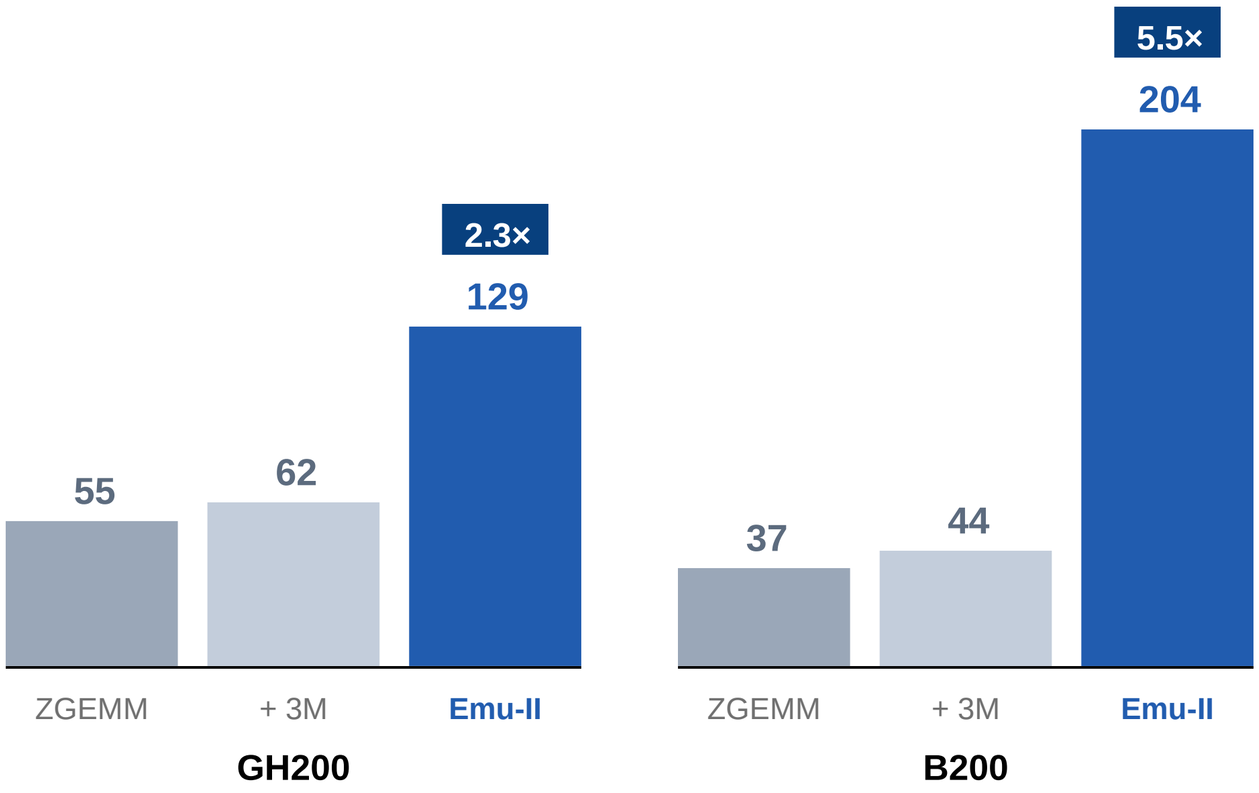
Caveat, up front: CRT reconstruction is a separate kernel — all reported numbers are end-to-end.

GMEM

3M for Scheme II: Uchino et al. [16] · fused kernels: Lu et al. [4]

Complex GEMM gains more than real — the 3M discount is free

If your workload is NEGF, Green's functions, or frequency-domain — this is your kernel.



Why complex gains more than real

3M is exact in modular arithmetic, so the 25% reduction in GEMM count is free — native 3M pays for its discount in accuracy. The more complex-valued the workload, the larger the advantage.

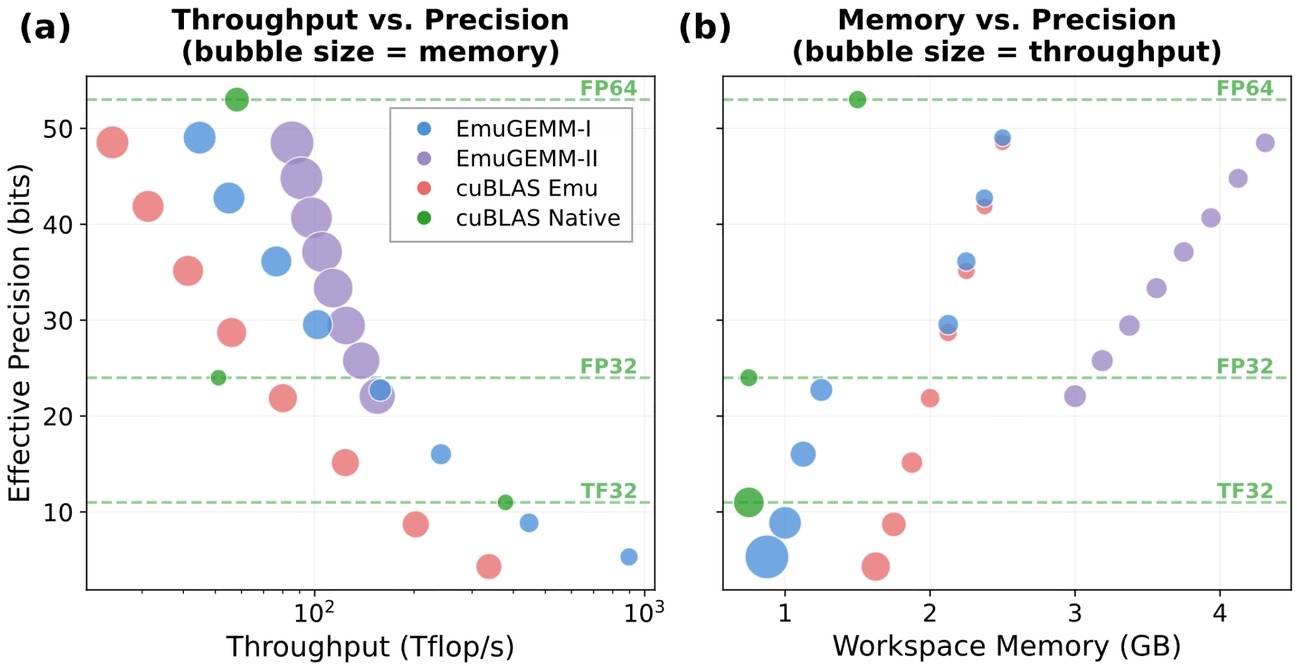
The low-precision end, too

EmuGEMM-I CGEMM at p4, 4k: 129 vs 54 for native FP32 CGEMM on GH200 — 2.4×.

Effective Tflop/s · ZGEMM at p15, M = N = K = 16,384 · cuBLAS native, with and without 3M

Data: Lu et al. [4] · GH200 and B200, CUDA 13.1

Precision is a dial: every p is a measured point



Same bits, about twice the throughput

At every slice count both EmuGEMM curves sit to the right of cuBLAS emulation — ≈ 2× at matched accuracy, 3.5× at FP64 class with Scheme II.

The crossover, measured

Scheme I and Scheme II cross at the FP32 line — one library, two engines, exactly as designed.

Kernel accuracy is not science accuracy — validate on the observables you publish. Next: QuaTrEx.

GH200 · M = N = K = 8,192 · test matrices with φ = 4.0, a deliberately wide exponent spread · effective bits = $-\log_2$ of the relative error vs a reference product · bubble size: workspace memory in (a), throughput in (b)

Data: Lu et al. [4] · cuBLAS emulation: Schwarz et al. [11]

Integrated Systems Laboratory

QuaTrEx: a four-phase loop, 98% of it block ZGEMM

G Green's functions

Assemble $M = ES - H - \Sigma^R$, then the RGF sweep: block-tridiagonal Schur recursion, forward and back

multiply · invert · multiply on $3{,}408^2$ complex blocks — only the selected blocks of G

ZGEMM | **κ up to 10^7**

→

P Polarization

Energy convolution of G with G

element-wise over orbital pairs — FFT along energy, after a data transposition

FFT

→

W Screened Coulomb

Assemble $M = I - V P^R$ and $B = V P^{\lessgtr} V^\dagger$ — block products — then the same RGF sweep

two ZGEMM-bound steps: assembly and solve

ZGEMM | **ZGEMM** | **$\kappa \lesssim 10^3$**

→

Σ GW self-energy

Energy convolution of G with W

element-wise — FFT along energy · feeds the next G

FFT

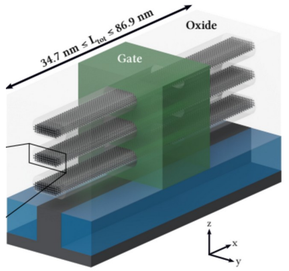
↻ repeat until the current is conserved between the contacts — hundreds of iterations

every quantity complex, $N_E \times N_{orb} \times N_{orb}$, block-tridiagonal · 10^4–10^5 energies

Where the flops go — one iteration, one GH200 24,288-atom nanoribbon · 833 Tflop · 19.7 s

G · RGF sweep	W · assembly	W · RGF sweep
244 Tflop	338 Tflop	244 Tflop

≈ 98% of the work is dense complex block multiplication at N = 3,408 — far above the emulation size threshold · boundary conditions, FFTs and the rest: 1%



Dense complex blocks, thousands of times per iteration — that is NEGF to a GEMM library.

Vetsch et al. [3], Table 4, NR-23 on Alps

NR-L, 10,560 atoms, on Alps: 1.51× throughput on 40% fewer nodes

Full simulation · GH200 nodes — 34,080 orbitals · 4,608 energies · ≥ 380 iterations

	FP64 reference	FP24 + EmuGEMM-II · p = 8 / 6	
nodes	128	77	−40%
energies per GPU	9	15	FP24 storage
Tflop/s per GPU	38.8	58.4	emulated ZGEMM
s per iteration	96.2	106.3	on 40% fewer nodes
throughput per energy point	1.00×	1.51×	

THE PHYSICS DID NOT MOVE

- **1.2×10^{-4}** relative error in I_d · budget 10^{-3}
- **8.25 µA** drain current — both ways, current conserved
- **same rate** of convergence · 1% conservation at ≈ 440 iterations

ONE GPU, ONE ITERATION · P = 6

- GH200 94.3 s → 58.3 s — **1.6×**
- B200 110.9 s → 40.8 s — **2.7×**

23% of the FP64 iteration is in kernels we did not touch — Amdahl sets the ceiling.

Same physics, same current, same convergence — on 40% fewer GPUs.

Maeder, Lu et al. [17] — Tables III–IV, Fig. 7

MALOQ: learned Hamiltonians for 20,000-atom devices

Atomistic device — STRUCTURE

Single-layer MoS_2 between amorphous HfO_2 or Al_2O_3 — melt-and-quench MD, relaxed · 10k–20k+ atoms, oxide inside the quantum domain

MLIP · MD

→

MALOQ — ELECTRONIC STRUCTURE

SO(2)-equivariant graph convolutions → Fock head → H and S blocks for any structure size — DFT accuracy without running DFT

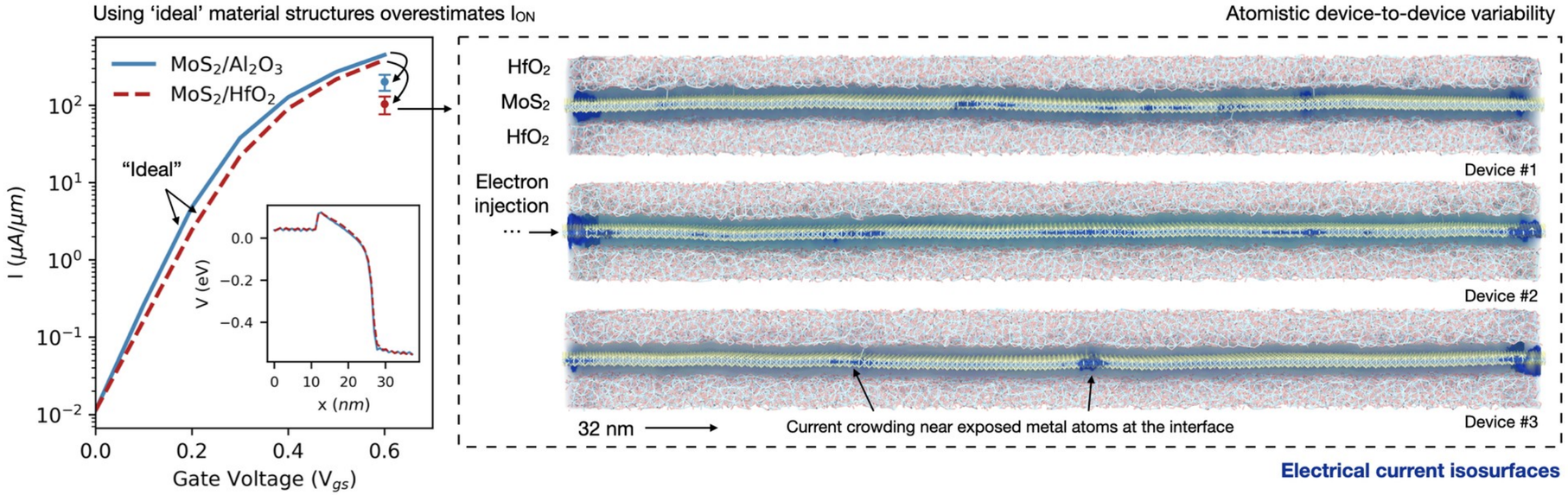
inference | GEMM-bound · low p

→

QuaTrEx — TRANSPORT

QTBM / NEGF with open boundaries → I–V, current isosurfaces, device-to-device statistics

ZGEMM · p = 8 / 6



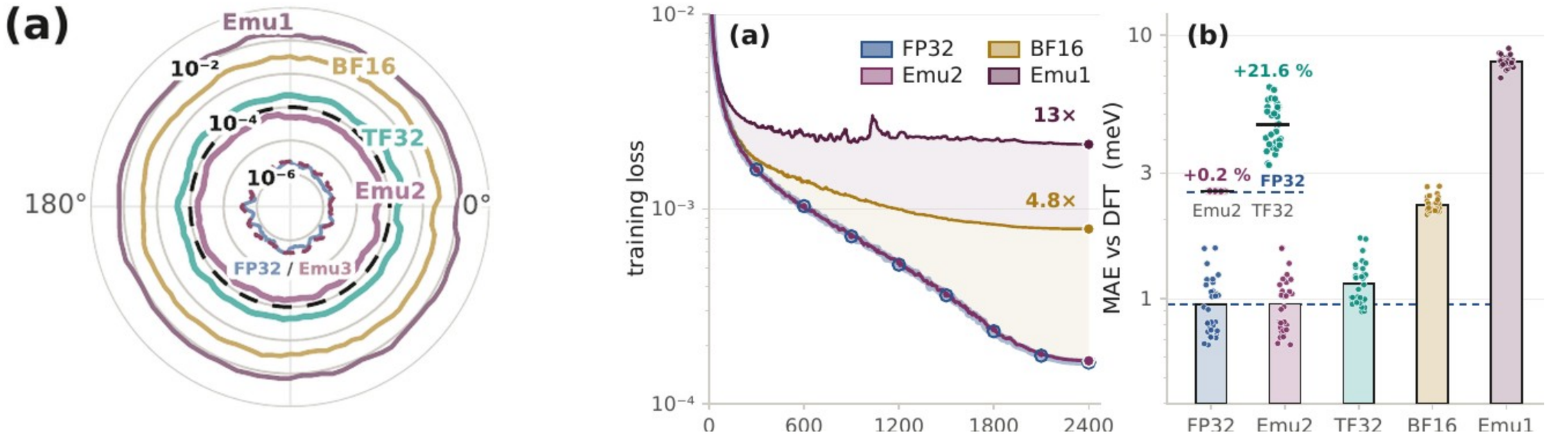
Real devices vary — statistics need ensembles, ensembles need cheap inference, and inference is GEMM.

FlashSO(2) [20] — emulation inside the model. MALOQ's SO(2) layers run on the integer Ozaki scheme: training is 3.9× faster than its FP32 version, with 58% less peak memory and no loss in accuracy.

Kaniselvan et al. [18] · MALOQ [19] · FlashSO(2) [20] · figure: M. Kaniselvan

Integrated Systems Laboratory

Fock-head equivariance vs compute precision: p = 2 clears the 10^{-4} tolerance



emu2 · p = 2 → 5.4 × 10^{-5} — under tolerance, 4× tighter than TF32, closer to FP32 than BF16 · three INT8 products

emu1 · p = 1 → 1.3 × 10^{-2} — coarser than BF16: the floor of the dial, and the model tells you where it is

One slice buys 2^8 — emu1 → emu2 measured 240×; the next slice lands under the fp64 floor

One library, both ends of the dial: p = 8 for NEGF at 10^{-3} on the current — p = 2 for a Fock head at 10^{-4}.

FlashSO(2) [20], Figs. 4a and 5a–b · MALOQ [19]

Equivariance and training: FlashSO(2) [20] · Fock-head measurement: M. Kaniselvan

TAKEAWAYS

Two things to take home

HIGH PRECISION · DFT, NEGF, TRADITIONAL SIMULATION

Fine-grained control of precision — and with it, of throughput

5.5×

cuBLAS ZGEMM on B200 at FP64-class accuracy 1.6–2.7× per GPU and iteration in QuaTrEx

- Slice count p is the dial: TF32-class → FP32 → FP64, real and complex. Each slice buys 2^8 of accuracy and costs a known share of throughput — you pick the point, per call.
- The conditioning of your problem sets p: in QuaTrEx the G system (κ up to 10^7) takes p = 8, the W system ($\kappa \lesssim 10^3$) p = 6 — the same rule for any dense GEMM in a DFT or transport loop.
- FP64-class results on silicon built for FP4 — a GEMM in, a GEMM out: QuaTrEx swapped its three ZGEMM steps, the solver untouched.

TF32 LEVEL · MACHINE-LEARNED POTENTIALS (MLIP)

At TF32 level, emulation already beats the native format

1.7×

faster than cuBLAS TF32 on B200
large matrices, comparable accuracy

- p = 2 is three INT8 products: 1.7× cuBLAS TF32 on B200 (1.4× on GH200), N = 16,384.
- FlashSO(2) [20]: emulated SO(2) layers train MALOQ 3.9× faster than its FP32 version, with 58% less peak memory and no loss in accuracy.
- Potential for MLIPs: the ladder replaces the TF32 / BF16 menu — add a slice where a model needs it; the matmuls go through unchanged.

One library, both ends of the dial: precision set per call by your problem, not by the format menu of the GPU.

arXiv 2606.25453 · SC26 · artifacts on Zenodo

CLOSING

Acknowledgements

Integrated Systems Laboratory

Swiss National Science Foundation

Grant n° 209358 — QuaTrEx

Platform for Advanced Scientific Computing

Project BoostQT

Swiss National Supercomputing Centre

Scientific support & HPC resources — projects c33 · lp16 · lp82

Preprint · arXiv 2606.25453 **Paper** · SC26, Algorithms track **Artifacts** · Zenodo — source, scripts, result logs

This work was supported by the Swiss National Science Foundation (SNSF) under grant n° 209358 (QuaTrEx) and by the Platform for Advanced Scientific Computing (PASC) in Switzerland (BoostQT). We acknowledge the scientific support and HPC resources from CSCS under projects c33, lp16, and lp82.

References

[1] TOP500 lists, June 2018 and November 2022. top500.org.

[2] NVIDIA. Tesla V100, A100, H100, GH200 and DGX B200 datasheets, 2017–2025.

[3] N. Vetsch, A. Maeder, V. Maillou, A. Winka, J. Cao, G. Kwasniewski, L. Deuschle, T. Hoefler, A. N. Ziogas, M. Luisier. Ab-initio quantum transport with the GW approximation, 42,240 atoms, and sustained exascale performance. SC ’25 (Gordon Bell Prize finalist). arXiv:2508.19138.

[4] D. Lu, A. Maeder, M. Luisier, A. N. Ziogas. EmuGEMM: Fused Tensor Core kernels for precision emulation in matrix multiplication. SC ’26. arXiv:2606.25453.

[5] 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. Numerical Algorithms 59(1), 2012.

[6] D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura. DGEMM using Tensor Cores, and its accurate and reproducible versions. ISC High Performance 2020, LNCS 12151.

[7] H. Ootomo, K. Ozaki, R. Yokota. DGEMM on integer matrix multiplication unit. IJHPCA 38(4), 2024.

[8] Y. Uchino, K. Ozaki, T. Imamura. Performance enhancement of the Ozaki Scheme on integer matrix multiplication unit. IJHPCA 39(3), 2025.

[9] K. Ozaki, Y. Uchino, T. Imamura. Ozaki Scheme II: A GEMM-oriented emulation of floating-point matrix multiplication using an integer modular technique. IJHPCA, 2026.

[10] Y. Uchino, K. Ozaki, T. Imamura. High-performance and power-efficient emulation of matrix multiplication using INT8 matrix engines. SC ’25 Workshops. Library: GEMMul8, RIKEN R-CCS.

[11] A. Schwarz, A. Anders, C. Brower, H. Bayraktar, J. Gunnels, K. Clark, R. G. Xu, S. Rodriguez, S. Cayrols, P. Tabaszewski, V. Podlozhnyuk. Guaranteed DGEMM accuracy while using reduced precision tensor cores through extensions of the Ozaki scheme. SCA/HPCAsia ’26.

[12] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. NeurIPS 2022.

[13] C. Zhao et al. DeepGEMM: clean and efficient BLAS kernel library on GPU. GitHub, 2025.

[14] V. Thakkar, P. Ramani, C. Cecka et al. CUTLASS. GitHub, 2023.

[15] C. Cecka. CuTe layout representation and algebra. arXiv, 2026.

[16] Y. Uchino, Q. Ma, T. Imamura, K. Ozaki, P. L. Gutsche. Emulation of complex matrix multiplication based on the Chinese Remainder Theorem. ISC High Performance 2026.

[17] A. Maeder, D. Lu, N. Vetsch, V. Maillou, A. Winka, J. Cao, M. Dossena, A. N. Ziogas, M. Luisier. Optimizing semiconductor device simulations through low-precision arithmetic. arXiv:2606.25595, 2026.

[18] M. Kaniselvan, M. Dossena, D. Lu, A. Maeder, N. Vetsch, A. N. Ziogas, M. Luisier. Ab initio modeling of MoS_2/oxide device interfaces with machine learned electronic structures. arXiv:2608.27533, 2026.

[19] M. Kaniselvan, A. Maeder, D. Lu, A. N. Ziogas, M. Luisier. MALOQ: Massively accelerated learning of operators for quantum transport. arXiv:2606.28911, 2026.

[20] FlashSO(2): Accelerating SO(2)-equivariant materials models through low-precision emulation. Manuscript under review, 2026.