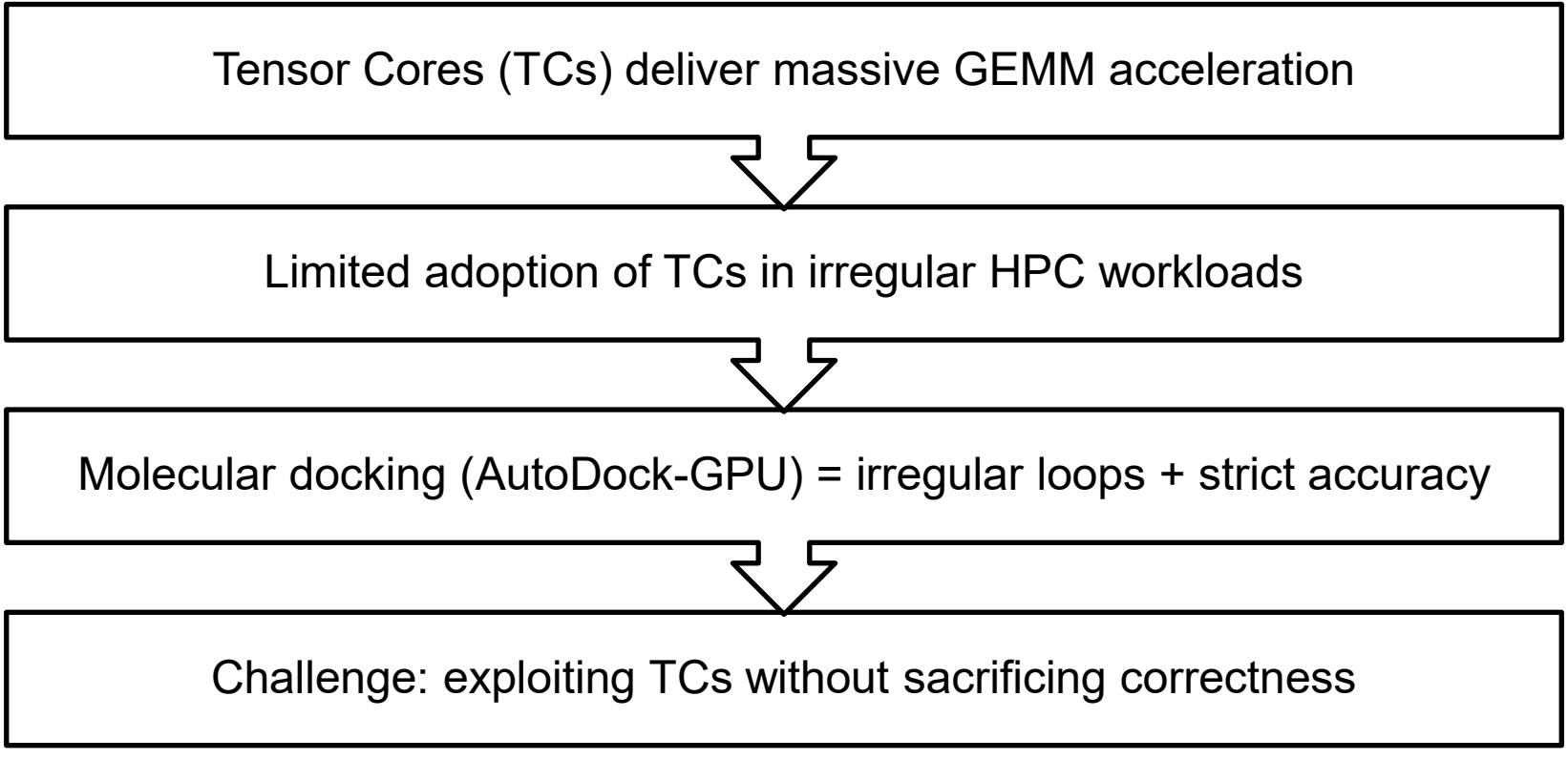
Architecting Tensor Core-Based Reductions for Irregular Molecular Docking Kernels

Leonardo Solis-Vasquez[1], Andreas F. Tillack[2], Diogo Santos-Martins[2], Andreas Koch[1], Stefano Forli[2]

1:  TECHNISCHE UNIVERSITÄT DARMSTADT

2: Scripps Research

Motivation


Tensor Cores (TCs) deliver massive GEMM acceleration
Limited adoption of TCs in irregular HPC workloads
Molecular docking (AutoDock-GPU) = irregular loops + strict accuracy
Challenge: exploiting TCs without sacrificing correctness

Our Contributions

Offload **reductions** in AutoDock-GPU molecular docking to **Tensor Cores (TCs)**

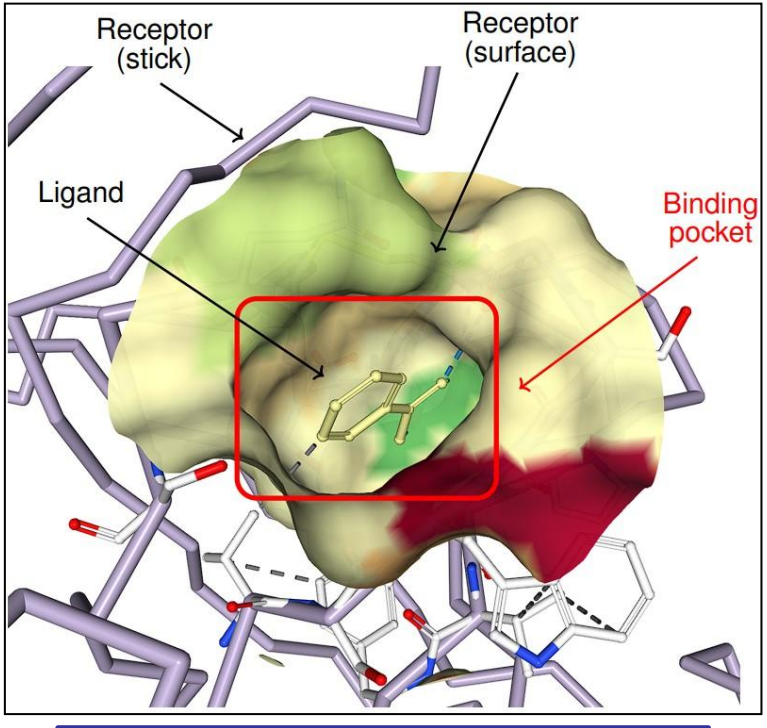
Incorporate numerical **Error Correction (EC)** methods for maintaining accuracy

Evaluate across NVIDIA GPUs:

A100, H100, B200

Molecular Docking

- Key method in computer-aided drug design
- Simulates the close-distance interaction of two molecules
- Aims to predict their binding poses in terms of energy (score)



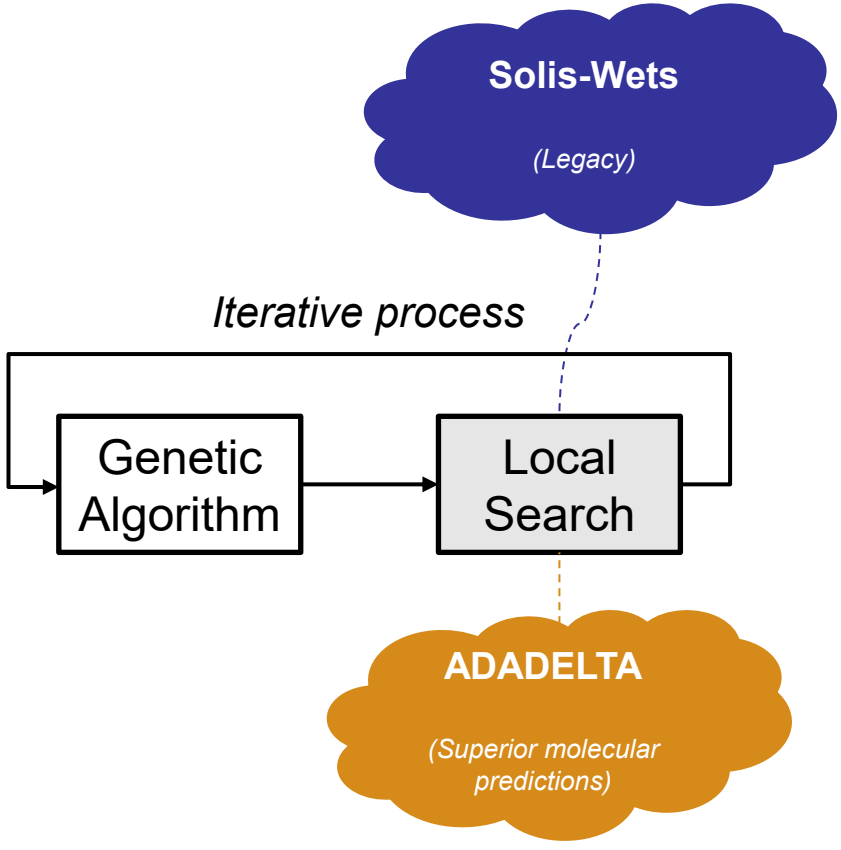
Binding between a ligand and a receptor of the “3ptb” molecular complex

AutoDock-GPU
Parallel Molecular Docking Application

AutoDock-GPU =
[Genetic Algorithm] + [Local Search]

[Local Search]
- Driven by score optimization
- Two alternative methods

ADADELTA (our focus)
- > 90% total AutoDock-GPU's execution time



Reductions in the ADADELTA kernel

ADADELTA relies on a gradient calculation (GC)

- GC performs **seven block-level sum reductions** per iteration

Reductions result in warp stalls

- 40% due to memory barriers
- 25% due to shared memory instruction latency

```
REDUCEFLOATSUM(torque_rot.x, pFloatAccumulator);
REDUCEFLOATSUM(torque_rot.y, pFloatAccumulator);
REDUCEFLOATSUM(torque_rot.z, pFloatAccumulator);
REDUCEFLOATSUM(energy, pFloatAccumulator);
REDUCEFLOATSUM(gx, pFloatAccumulator);
REDUCEFLOATSUM(gy, pFloatAccumulator);
REDUCEFLOATSUM(gz, pFloatAccumulator);
```

```
#define REDUCEFLOATSUM(value, pAccumulator) \
    if (threadIdx.x == 0) \
    { \
        *pAccumulator = 0; \
    } \
    __threadfence(); \
    __syncthreads(); \
    if (__any_sync(0xffffffff, value != 0.0f)) \
    { \
        uint32_t tgx            = threadIdx.x & cData.warpmask; \
        value                  += __shfl_sync(0xffffffff, value, tgx ^ 1); \
        value                  += __shfl_sync(0xffffffff, value, tgx ^ 2); \
        value                  += __shfl_sync(0xffffffff, value, tgx ^ 4); \
        value                  += __shfl_sync(0xffffffff, value, tgx ^ 8); \
        value                  += __shfl_sync(0xffffffff, value, tgx ^ 16); \
        if (tgx == 0) \
        { \
            atomicAdd(pAccumulator, value); \
        } \
    } \
    __threadfence(); \
    __syncthreads(); \
    value = (float)(*pAccumulator); \
    __syncthreads();
```

Schieffer & Peng (2023): Matrix-based Reduction Algorithm in AutoDock-GPU

$$\begin{cases} V \leftarrow A \times P + V \\ W \leftarrow Q \times V \end{cases}$$

Employed a matrix-based, **multi-dimensional** reduction algorithm

Offloaded reductions (in ADADELTA) to Tensor Cores

$$A = \begin{pmatrix} x_0 & x_4 & \dots & x_{60} \\ y_0 & y_4 & \dots & y_{60} \\ z_0 & z_4 & \dots & z_{60} \\ e_0 & e_4 & \dots & e_{60} \\ x_1 & x_5 & \dots & x_{61} \\ y_1 & y_5 & \dots & y_{61} \\ z_1 & z_5 & \dots & z_{61} \\ e_1 & e_5 & \dots & e_{61} \\ \vdots & \vdots & \ddots & \vdots \\ x_3 & x_7 & \dots & x_{63} \\ y_3 & y_7 & \dots & y_{63} \\ z_3 & z_7 & \dots & z_{63} \\ e_3 & e_7 & \dots & e_{63} \end{pmatrix}_{16\times16}$$

In all matrices: **half-precision** floating-point values

"A": input data to be reduced (column major)

Contains first 64 vectors: $\{x_0, y_0, z_0, e_0\}$, …, $\{x_{63}, y_{63}, z_{63}, e_{63}\}$

Schieffer & Peng (2023): Matrix-based Reduction Algorithm in AutoDock-GPU

$$\begin{cases} V \leftarrow A \times P + V \\ W \leftarrow Q \times V \end{cases}$$

Employed a matrix-based, **multi-dimensional** reduction algorithm

Offloaded reductions (in ADADELTA) to Tensor Cores

$$A = \begin{pmatrix} x_0 & x_4 & \dots & x_{60} \\ y_0 & y_4 & \dots & y_{60} \\ z_0 & z_4 & \dots & z_{60} \\ e_0 & e_4 & \dots & e_{60} \\ x_1 & x_5 & \dots & x_{61} \\ y_1 & y_5 & \dots & y_{61} \\ z_1 & z_5 & \dots & z_{61} \\ e_1 & e_5 & \dots & e_{61} \\ \vdots & \vdots & \ddots & \vdots \\ x_3 & x_7 & \dots & x_{63} \\ y_3 & y_7 & \dots & y_{63} \\ z_3 & z_7 & \dots & z_{63} \\ e_3 & e_7 & \dots & e_{63} \end{pmatrix}_{16\times16}$$

$$P = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}_{16\times16}$$

$$Q = \begin{pmatrix} I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \end{pmatrix}_{16\times16}$$

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}_{4\times4}$$

Schieffer & Peng (2023): Matrix-based Reduction Algorithm in AutoDock-GPU

$$\begin{cases} V \leftarrow A \times P + V \\ W \leftarrow Q \times V \end{cases}$$

A x P performs the summation across the rows of A:

$\sum X_{4i} = x_0 + x_4 + \ldots + x_{60}$

If more than 64 vectors must be reduced, A is repopulated with next batch

$$A \times P = \begin{pmatrix} \sum x_{4i} & \sum x_{4i} & \cdots & \sum x_{4i} \\ \sum y_{4i} & \sum y_{4i} & \cdots & \sum y_{4i} \\ \sum z_{4i} & \sum z_{4i} & \cdots & \sum z_{4i} \\ \sum e_{4i} & \sum e_{4i} & \cdots & \sum e_{4i} \\ \hline \sum x_{4i+1} & \sum x_{4i+1} & \cdots & \sum x_{4i+1} \\ \sum y_{4i+1} & \sum y_{4i+1} & \cdots & \sum y_{4i+1} \\ \sum z_{4i+1} & \sum z_{4i+1} & \cdots & \sum z_{4i+1} \\ \sum e_{4i+1} & \sum e_{4i+1} & \cdots & \sum e_{4i+1} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline \sum x_{4i+3} & \sum x_{4i+3} & \cdots & \sum x_{4i+3} \\ \sum y_{4i+3} & \sum y_{4i+3} & \cdots & \sum y_{4i+3} \\ \sum z_{4i+3} & \sum z_{4i+3} & \cdots & \sum z_{4i+3} \\ \sum e_{4i+3} & \sum e_{4i+3} & \cdots & \sum e_{4i+3} \end{pmatrix}_{16\times16}$$

Schieffer & Peng (2023): Matrix-based Reduction Algorithm in AutoDock-GPU

$$\begin{cases} V \leftarrow A \times P + V \\ W \leftarrow Q \times V \end{cases}$$

Q x V computes the sum of every fourth element in each column of matrix V:

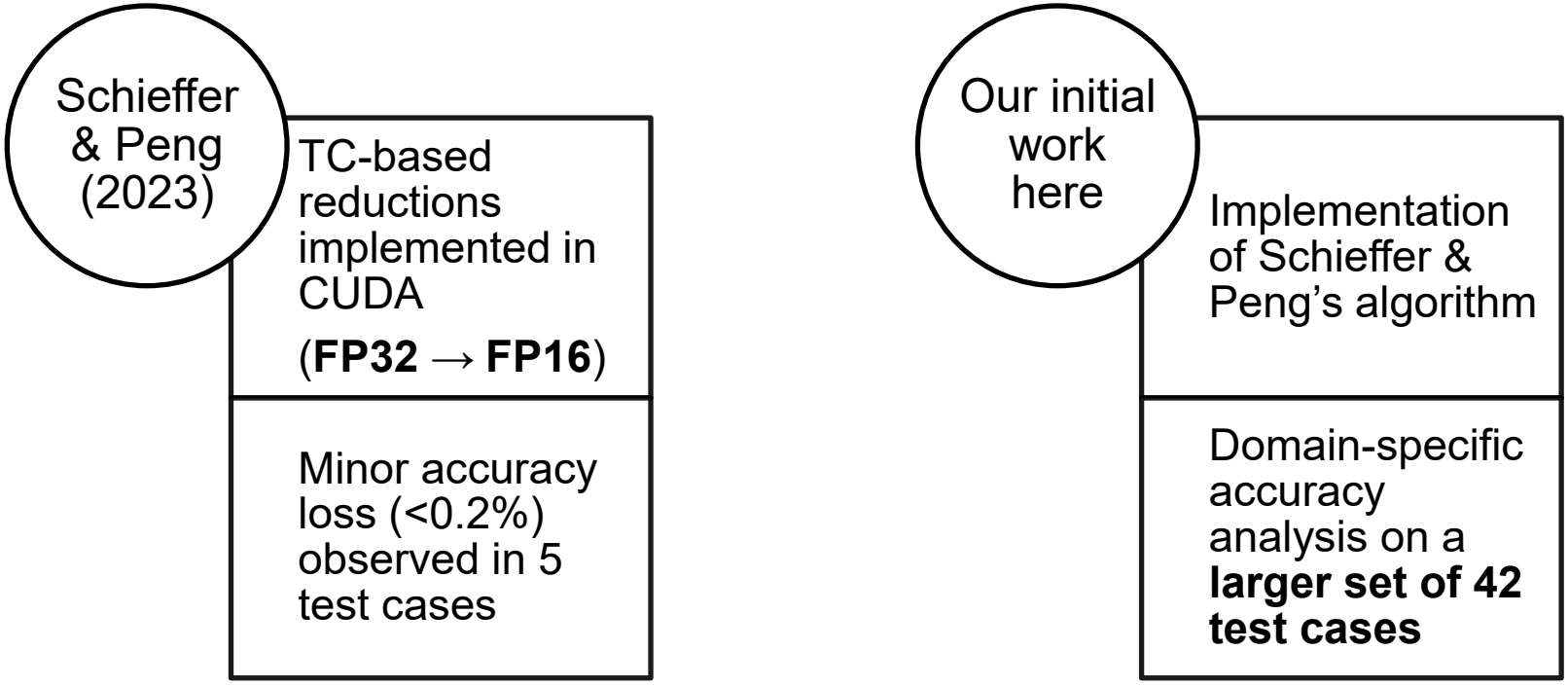
$$\sum X_i = \sum x_{4i} + \sum x_{4i+1} + \sum x_{4i+2} + \sum x_{4i+3}$$

W: its first column contains the final reduction for {x, y, z, e}

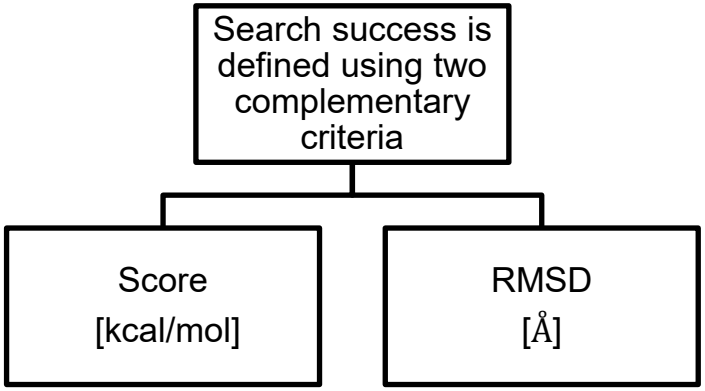
$$A \times P = \begin{pmatrix} \sum x_{4i} & \sum x_{4i} & \dots & \sum x_{4i} \\ \sum y_{4i} & \sum y_{4i} & \dots & \sum y_{4i} \\ \sum z_{4i} & \sum z_{4i} & \dots & \sum z_{4i} \\ \sum e_{4i} & \sum e_{4i} & \dots & \sum e_{4i} \\ \sum x_{4i+1} & \sum x_{4i+1} & \dots & \sum x_{4i+1} \\ \sum y_{4i+1} & \sum y_{4i+1} & \dots & \sum y_{4i+1} \\ \sum z_{4i+1} & \sum z_{4i+1} & \dots & \sum z_{4i+1} \\ \sum e_{4i+1} & \sum e_{4i+1} & \dots & \sum e_{4i+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum x_{4i+3} & \sum x_{4i+3} & \dots & \sum x_{4i+3} \\ \sum y_{4i+3} & \sum y_{4i+3} & \dots & \sum y_{4i+3} \\ \sum z_{4i+3} & \sum z_{4i+3} & \dots & \sum z_{4i+3} \\ \sum e_{4i+3} & \sum e_{4i+3} & \dots & \sum e_{4i+3} \end{pmatrix}_{16\times16}$$

$$W = \begin{pmatrix} \sum x_i & \sum x_i & \dots & \sum x_i \\ \sum y_i & \sum y_i & \dots & \sum y_i \\ \sum z_i & \sum z_i & \dots & \sum z_i \\ \sum e_i & \sum e_i & \dots & \sum e_i \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix}_{16\times16}$$

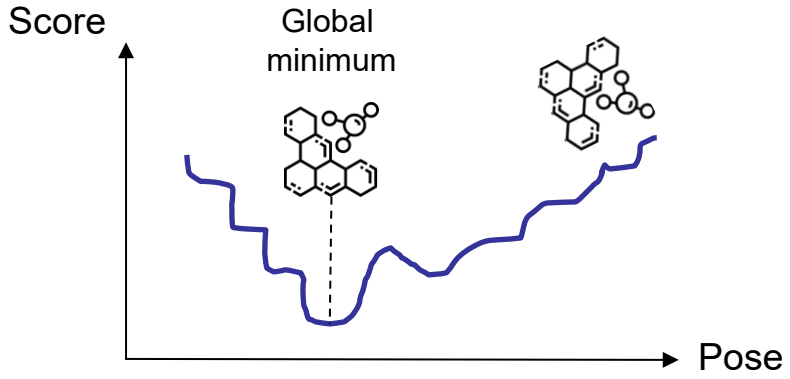
Our Algorithmic Analysis on Schieffer & Peng's Work



Accuracy Evaluation in AutoDock-GPU



RMSD: Root Mean Square Deviation



Resulting pose is deemed successful if:

$\lvert Score_{global\text{-}min} - Score_{achieved} \rvert < 1.0$ kcal/mol
$\lvert RMSD_{native\text{-}pose} - RMSD_{achieved} \rvert < 2$ Å

E_{50} metric: Half-Maximal Success Rate Evaluations

A sufficient number of score evaluations increases asymptotically the probability of finding the global minimum

$$E_{50} = 2^{a \times N_{\text{rot}} + b}$$ (our previous work in JCTC, 2021)

The number of score evaluations required to attain 50% probability of success

Analysis based on E_{50} metric: Schieffer & Peng vs. OpenCL reference

Algorithmic equivalence

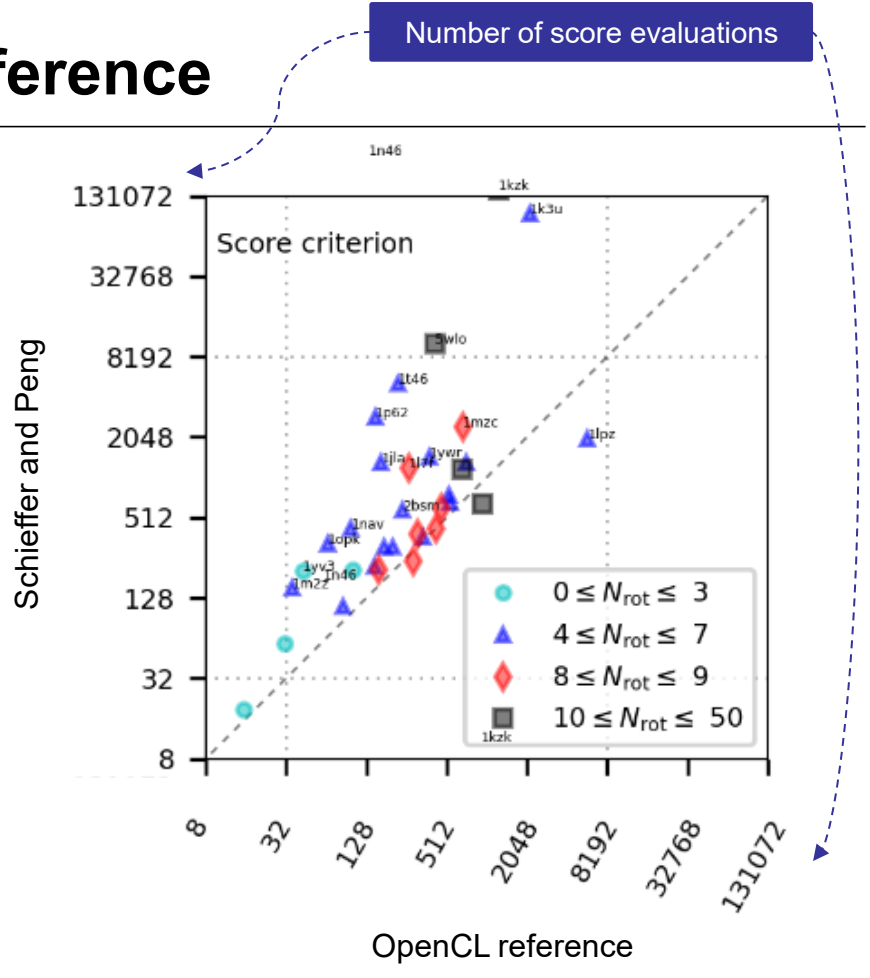
- Would manifest as E_{50} markers lying **on or near** the dashed diagonal line

Observation

- E_{50} achieved by Schieffer & Peng lie *above* the diagonal

Conclusion

- Schieffer & Peng requires **more score evaluations** (than the OpenCL reference) to achieve similar accuracy

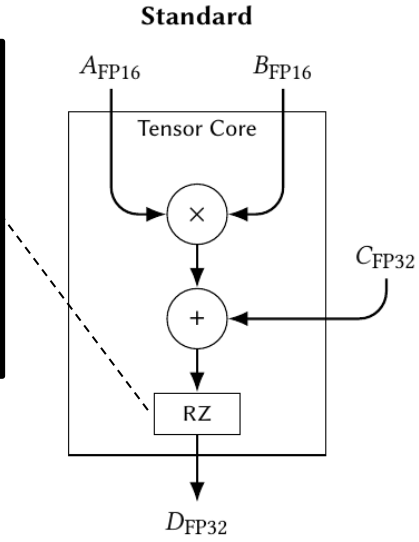


Ootomo & Yokota (2022): Error Correction (EC) Algorithm

$$D_{\text{FP32}} \leftarrow A_{\text{FP16}} \times B_{\text{FP16}} + C_{\text{FP32}}$$

Discovery

- Rounding-toward-Zero (RZ) was the *key contributor* to numerical accuracy loss



Ootomo & Yokota (2022): Error Correction (EC) Algorithm

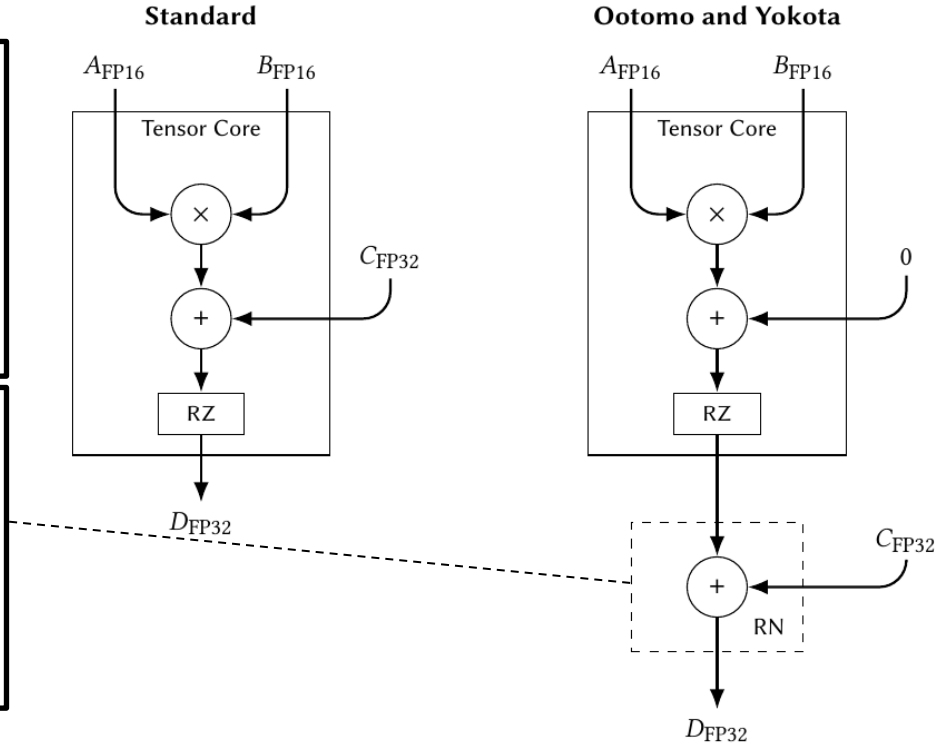
$$D_{\text{FP32}} \leftarrow A_{\text{FP16}} \times B_{\text{FP16}} + C_{\text{FP32}}$$

Discovery

- Rounding-toward-Zero (RZ) was the *key contributor* to numerical accuracy loss

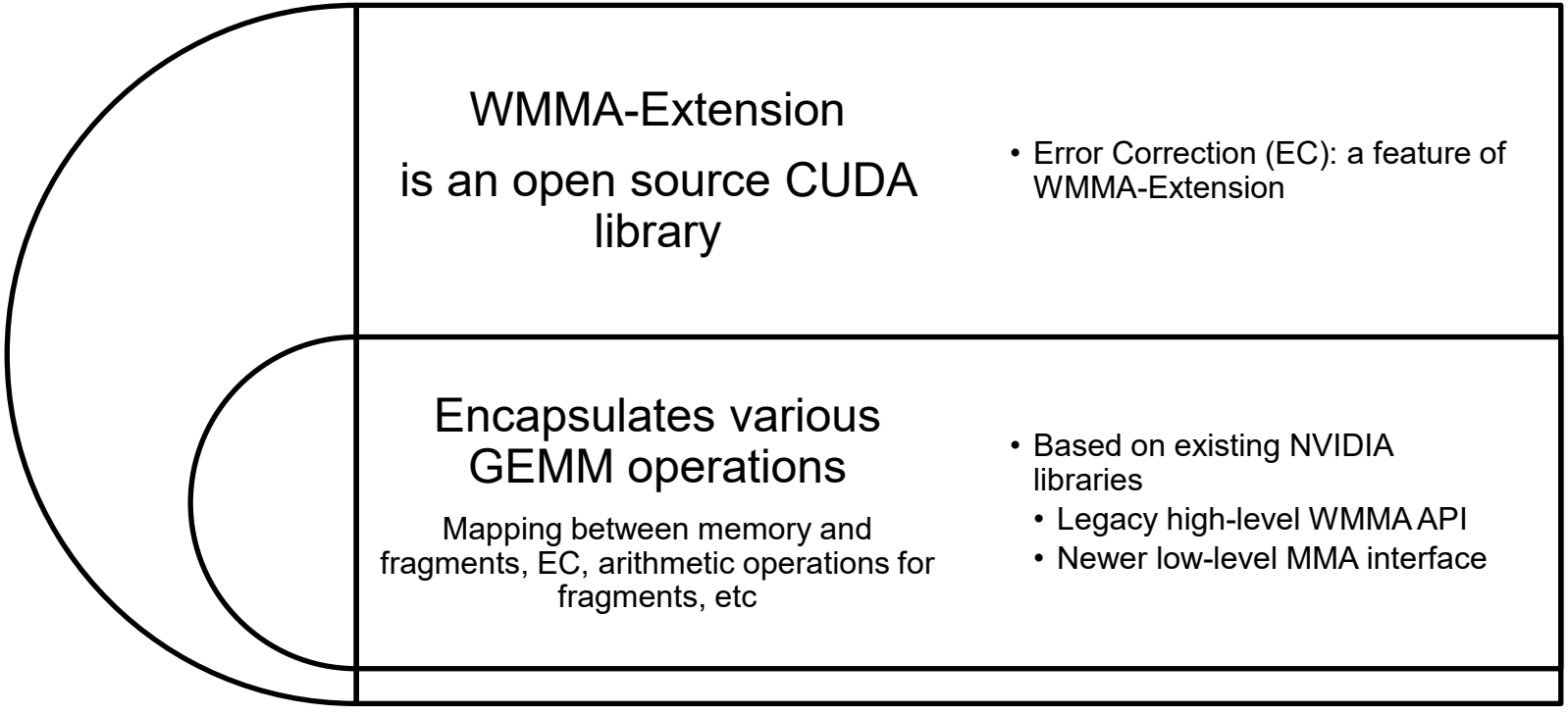
Proposal

- Performing accumulation **outside** the TCs
- Replacing RZ with **Rounding-to-Nearest (RN)**



Ootomo & Yokota (2022): WMMA-Extension

https://github.com/wmmae/wmma_extension



V ← A x P + V
CUDA Implementation

Schieffer & Peng w/o EC

```
/* Based on NVIDIA WMMA,
 * __not__ featuring Error Correction */
__shared__ __align__ (256) half tmp[TILE_SIZE];
wmma::fragment<matrix_a, M, N, K, half, layout_ab> frag_A;
wmma::fragment<matrix_b, M, N, K, half, layout_ab> frag_P;
wmma::fragment<matrix_c, M, N, K, half> frag_V;
wmma::load_matrix_sync(frag_A, datatoreduce + offset, 16);
wmma::fill_fragment(frag_P, HALF_ONE);
wmma::fill_fragment(frag_V, HALF_ZERO);
wmma::mma_sync(frag_V, frag_A, frag_P, frag_V);
wmma::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

Schieffer & Peng with EC (TCEC)

```
/* Based on WMMA Extension for FP32 GEMM using
 * Tensor Cores and Error Correction technique (TCEC) */
#include <wmma_extension/tcec/tcec.hpp>
using tf32 = wmma::precision::tf32;
__shared__ __align__ (256) float tmp[TILE_SIZE];
namespace tcec = mtk::wmma::tcec;
tcec::fragment<matrix_a, M, N, K, tf32, layout_ab> frag_A;
tcec::fragment<matrix_b, M, N, K, tf32, layout_ab> frag_P;
tcec::fragment<matrix_c, M, N, K, tf32> frag_V;
tcec::load_matrix_sync(frag_A, datatoreduce + offset, 16);
tcec::fill_fragment(frag_P, 1.0f);
tcec::fill_fragment(frag_V, 0.0f);
tcec::mma_sync(frag_V, frag_A, frag_P, frag_V);
tcec::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

V ← A x P + V
CUDA Implementation

Schieffer & Peng w/o EC

```
/* Based on NVIDIA WMMA,
 * __not__ featuring Error Correction */
__shared__ __align__ (256) half tmp[TILE_SIZE];
wmma::fragment<matrix_a, M, N, K, half, layout_ab> frag_A;
wmma::fragment<matrix_b, M, N, K, half, layout_ab> frag_P;
wmma::fragment<matrix_c, M, N, K, half> frag_V;
wmma::load_matrix_sync(frag_A, datatoreduce + offset, 16);
wmma::fill_fragment(frag_P, HALF_ONE);
wmma::fill_fragment(frag_V, HALF_ZERO);
wmma::mma_sync(frag_V, frag_A, frag_P, frag_V);
wmma::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

Schieffer & Peng with EC (TCEC)

```
/* Based on WMMA Extension for FP32 GEMM using
 * Tensor Cores and Error Correction technique (TCEC) */
#include <wmma_extension/tcec/tcec.hpp>
using tf32 = wmma::precision::tf32;
__shared__ __align__ (256) float tmp[TILE_SIZE];
namespace tcec = mtk::wmma::tcec;
tcec::fragment<matrix_a, M, N, K, tf32, layout_ab> frag_A;
tcec::fragment<matrix_b, M, N, K, tf32, layout_ab> frag_P;
tcec::fragment<matrix_c, M, N, K, tf32> frag_V;
tcec::load_matrix_sync(frag_A, datatoreduce + offset, 16);
tcec::fill_fragment(frag_P, 1.0f);
tcec::fill_fragment(frag_V, 0.0f);
tcec::mma_sync(frag_V, frag_A, frag_P, frag_V);
tcec::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

Including the WMMA-Extension header file **tcec.hpp**

Required changes in programming effort

V ← A x P + V
CUDA Implementation

Schieffer & Peng w/o EC

```
/* Based on NVIDIA WMMA,
 * __not__ featuring Error Correction */
__shared__ __align__ (256) half tmp[TILE_SIZE];
wmma::fragment<matrix_a, M, N, K, half, layout_ab> frag_A;
wmma::fragment<matrix_b, M, N, K, half, layout_ab> frag_P;
wmma::fragment<matrix_c, M, N, K, half> frag_V;
wmma::load_matrix_sync(frag_A, datatoreduce + offset, 16);
wmma::fill_fragment(frag_P, HALF_ONE);
wmma::fill_fragment(frag_V, HALF_ZERO);
wmma::mma_sync(frag_V, frag_A, frag_P, frag_V);
wmma::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

Schieffer & Peng with EC (TCEC)

```
/* Based on WMMA Extension for FP32 GEMM using
 * Tensor Cores and Error Correction technique (TCEC) */
#include <wmma_extension/tcec/tcec.hpp>
using tf32 = wmma::precision::tf32;
__shared__ __align__ (256) float tmp[TILE_SIZE];
namespace tcec = mtk::wmma::tcec;
tcec::fragment<matrix_a, M, N, K, tf32, layout_ab> frag_A;
tcec::fragment<matrix_b, M, N, K, tf32, layout_ab> frag_P;
tcec::fragment<matrix_c, M, N, K, tf32> frag_V;
tcec::load_matrix_sync(frag_A, datatoreduce + offset, 16);
tcec::fill_fragment(frag_P, 1.0f);
tcec::fill_fragment(frag_V, 0.0f);
tcec::mma_sync(frag_V, frag_A, frag_P, frag_V);
tcec::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

Including the WMMA-Extension header file **tcec.hpp**

Switching namespaces: **mtk::wmma::tcec ↔ nvcuda::wmma**

Required changes in programming effort

V ← A x P + V
Leveraging TF32 Support in WMMA-Extension

Schieffer & Peng w/o EC

```
/* Based on NVIDIA WMMA,
 * __not__ featuring Error Correction */
__shared__ __align__ (256) half tmp[TILE_SIZE];
wmma::fragment<matrix_a, M, N, K, half, layout_ab> frag_A;
wmma::fragment<matrix_b, M, N, K, half, layout_ab> frag_P;
wmma::fragment<matrix_c, M, N, K, half> frag_V;
wmma::load_matrix_sync(frag_A, datatoreduce + offset, 16);
wmma::fill_fragment(frag_P, HALF_ONE);
wmma::fill_fragment(frag_V, HALF_ZERO);
wmma::mma_sync(frag_V, frag_A, frag_P, frag_V);
wmma::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

FP16

Schieffer & Peng with EC (TCEC)

```
/* Based on WMMA Extension for FP32 GEMM using
 * Tensor Cores and Error Correction technique (TCEC) */
#include <wmma_extension/tcec/tcec.hpp>
using tf32 = wmma::precision::tf32;
__shared__ __align__ (256) float tmp[TILE_SIZE];
namespace tcec = mtk::wmma::tcec;
tcec::fragment<matrix_a, M, N, K, tf32, layout_ab> frag_A;
tcec::fragment<matrix_b, M, N, K, tf32, layout_ab> frag_P;
tcec::fragment<matrix_c, M, N, K, tf32> frag_V;
tcec::load_matrix_sync(frag_A, datatoreduce + offset, 16);
tcec::fill_fragment(frag_P, 1.0f);
tcec::fill_fragment(frag_V, 0.0f);
tcec::mma_sync(frag_V, frag_A, frag_P, frag_V);
tcec::store_matrix_sync(tmp, frag_V, 16, layout_c);
```

TF32

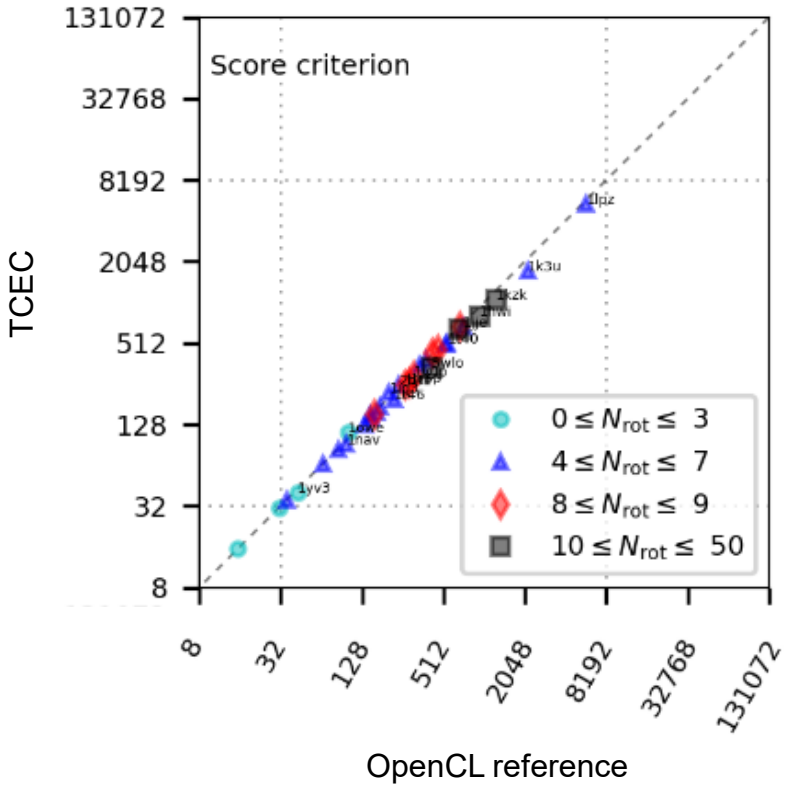
We adopt TF32 as it achieves accuracy levels (Ootomo & Yokota):

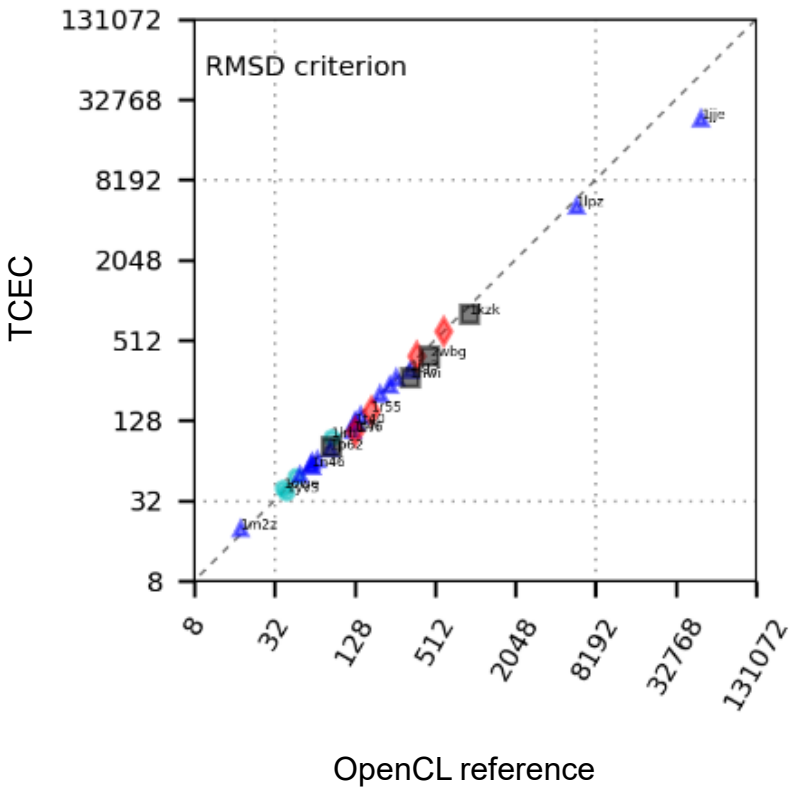
Comparable to FP32

Significantly higher than FP16

Analysis based on E_{50} metric: TCEC vs. OpenCL reference

Both TCEC and OpenCL achieve comparable accuracy in both criteria!





Evaluation Setup

Baseline:

FP32 SIMT cores (exclusively)

TCEC:

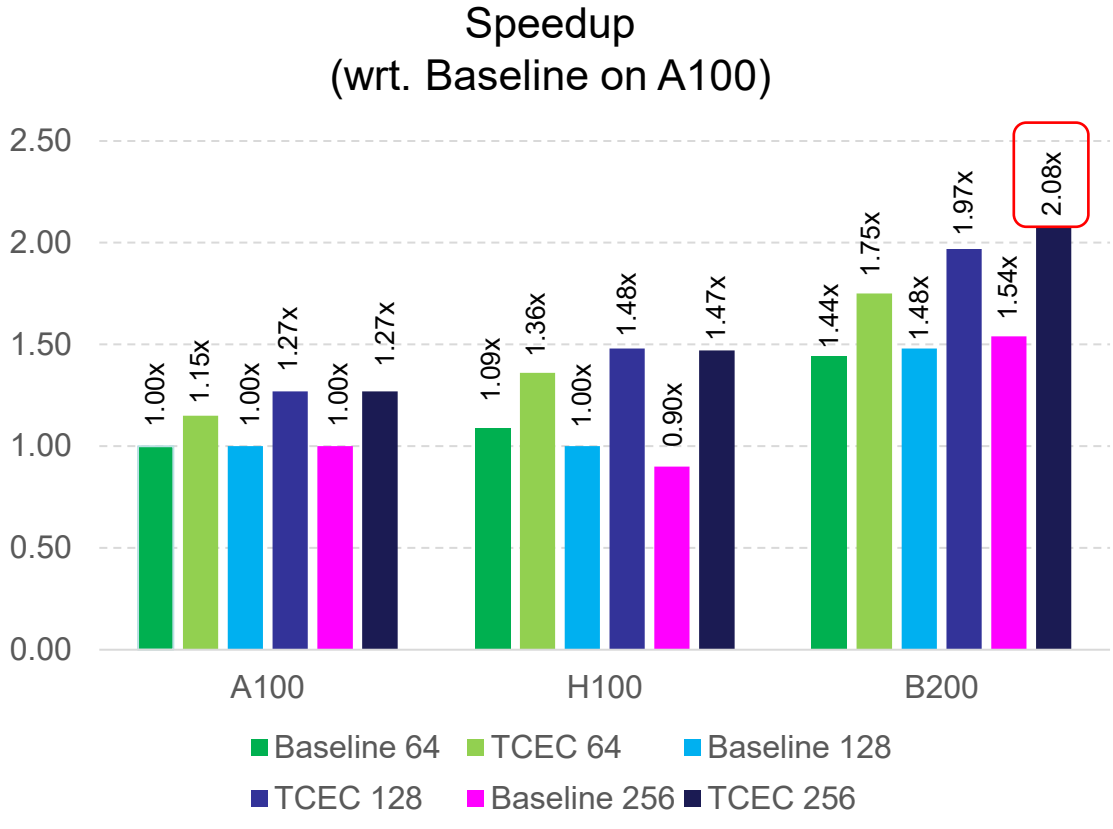
FP32 SIMT cores + **Tensor Cores** (for reductions)

Evaluation platforms:

A100, H100, B200

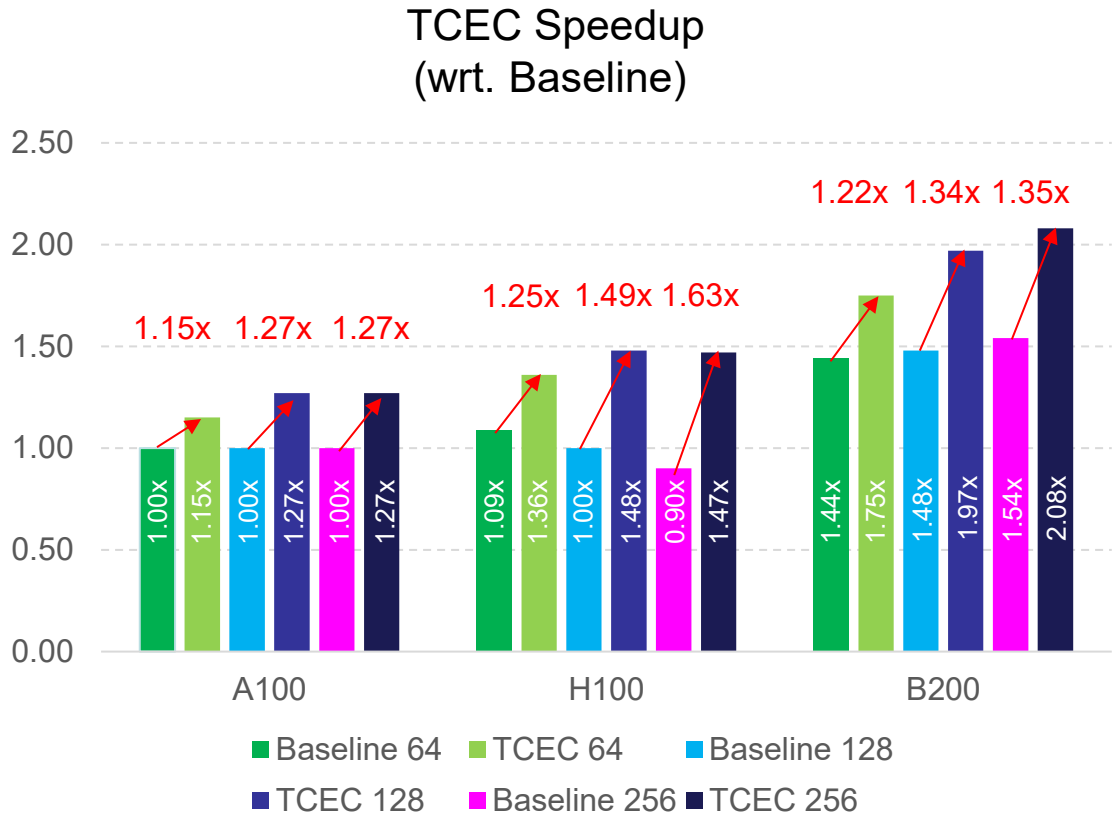
Absolute Speedups Across set of 42 test cases

Threads per block:
64 | 128 | 256



Relative Speedups Across set of 42 test cases

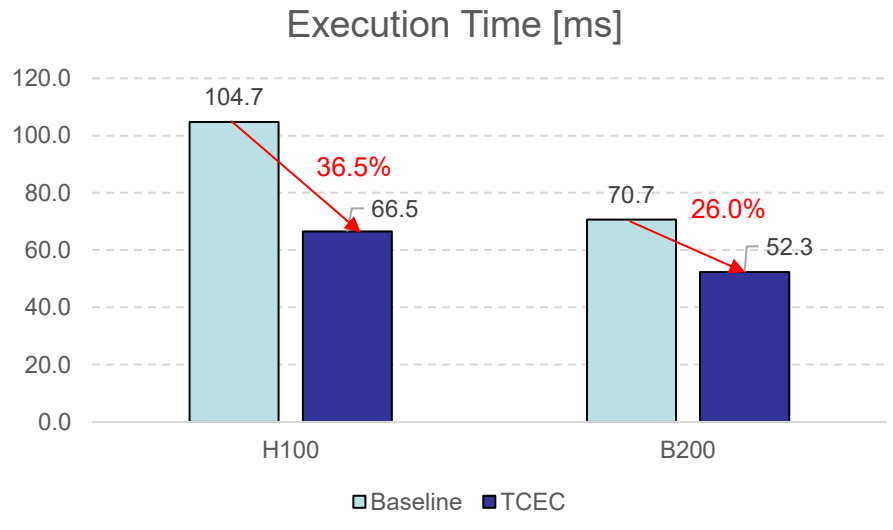
Threads per block:
64 | 128 | 256

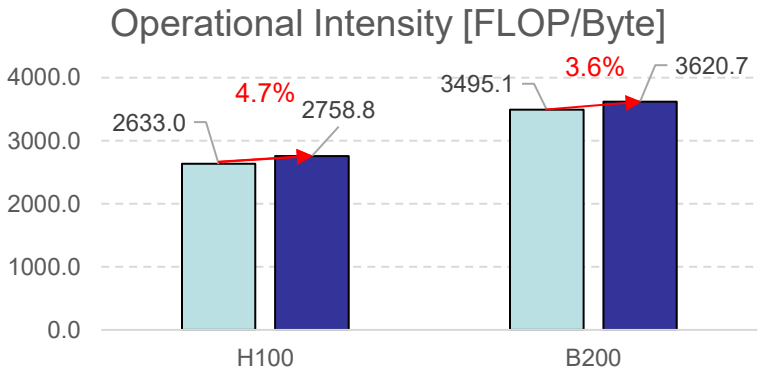


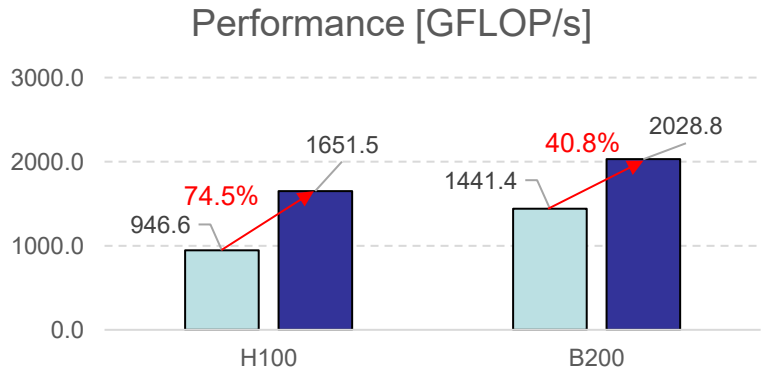
Execution Profiling of TCEC Kernel-Level Comparison (ADADELTA)

Threads per block: 256

Test case: 7cpa



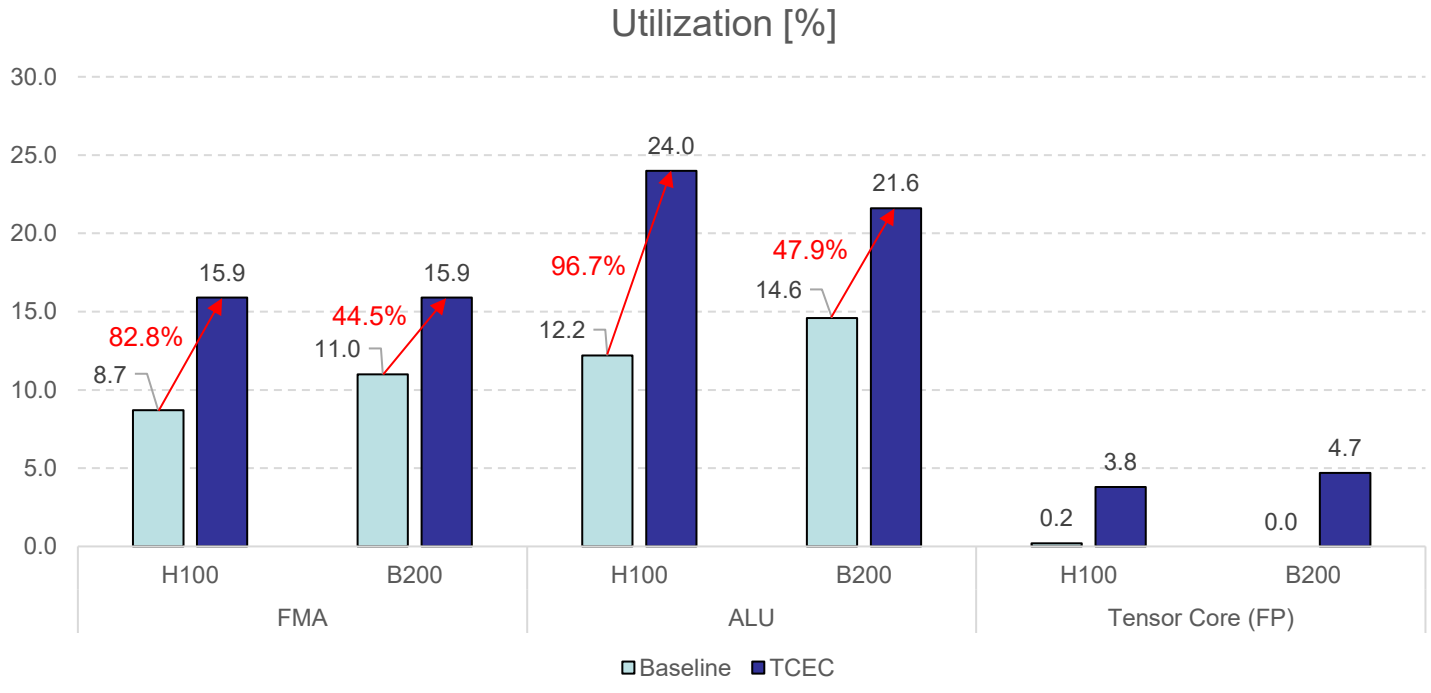




Execution Profiling of TCEC Kernel-Level Comparison (ADADELTA)

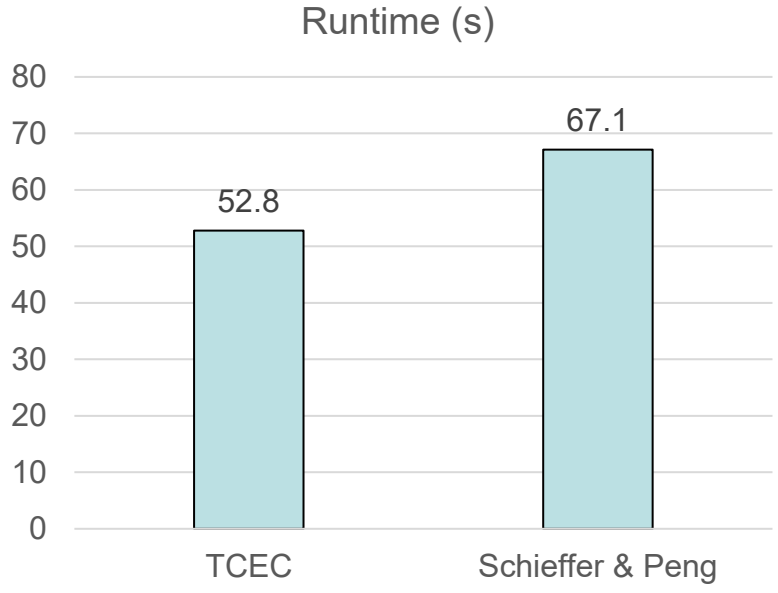
Threads per block: 256

Test case: 7cpa



TCEC vs. Schieffer & Peng (no EC) Across set of 42 test cases

Threads per block: 64



	TCEC	Schieffer & Peng (no EC)
Runtime (s)	52.81	67.07
Score Evals	105408770	105239792
µs/eval	0.50	0.64
Speedup	**1.27x**	1.00x

Summary

A Tensor Core-accelerated implementation of AutoDock-GPU

- Incorporated numerical correction (TCEC) to mitigate precision loss by FP16-based GEMM operations

Performance gains: TCEC vs. Baseline

- TCEC (TF32 on Tensor Cores) vs. Baseline (FP32 SIMT Cores)
- Set of 42 test cases | # Threads per block: 256
- A100: 27% | H100: 63% | B200: 35%

Architecting Tensor Core-Based Reductions for Irregular Molecular Docking Kernels

https://github.com/ccsb-scripps/AutoDock-GPU

Leonardo Solis-Vasquez

solis@esa.tu-darmstadt.de

Performance Model

$$\text{Predicted Speedup}_{\text{AD-GPU}} = \frac{T_{\text{Baseline}}}{T_{\text{TCEC}}} = \frac{1}{\frac{f}{S} + (1 - f)}$$

- S: Theoretical acceleration factor of Tensor Cores vs. FP32 SIMT Cores
- f: The fraction of the workload that is offloaded to TCs

Predicted speedup factors for some *f* values:

f	**A100** (S = 8.0x)	**H100** (S = 7.4x)	**B200** (S = 15.0x)
0.0	1.00x	1.00x	1.00x
0.9	3.55x	3.42x	8.57x
1.0	8.00x	7.40x	15.00x

Achieving around *half* of the theoretical maximum performance improvement requires a **high** utilization of TCs (i.e., $f \geq 0.9$)

Predicted vs. Measured Program-Level Speedups

Test case: 7cpa

GPU	# Threads per block	f_{eff}	Predicted Speedup	Measured Speedup
A100 (S = 8.0x)	128	0.14	1.14x	1.30x
	256	0.14	1.14x	1.30x
H100 (S = 7.4x)	128	0.14	1.14x	1.37x
	256	0.16	1.16x	1.57x
B200 (S = 15.0x)	128	0.18	1.20x	1.45x
	256	0.14	1.15x	1.30x

Predicted and measured program-level speedups are **closely aligned** in both magnitude and trend

Predicted vs. Measured Program-Level Speedups

Test case: 7cpa

GPU	# Threads per block	f_{eff}	Predicted Speedup	Measured Speedup
A100 (S = 8.0x)	128	0.14	1.14x	1.30x
	256	0.14	1.14x	1.30x
H100 (S = 7.4x)	128	0.14	1.14x	1.37x
	256	0.16	1.16x	1.57x
B200 (S = 15.0x)	128	0.18	1.20x	1.45x
	256	0.14	1.15x	1.30x

Block size = 256:

Speedup:
H100 > B200

On **H100**:

FP32 SIMT-only Baseline is *slower* → performance gains from TCs are more pronounced

Predicted vs. Measured Program-Level Speedups

Test case: 7cpa

GPU	# Threads per block	f_{eff}	Predicted Speedup	Measured Speedup
A100 (S = 8.0x)	128	0.14	1.14x	1.30x
	256	0.14	1.14x	1.30x
H100 (S = 7.4x)	128	0.14	1.14x	1.37x
	256	0.16	1.16x	1.57x
B200 (S = 15.0x)	128	0.18	1.20x	1.45x
	256	0.14	1.15x	1.30x

Block size = 256:

Speedup:
H100 > B200

On **B200**:

The *higher* memory bandwidth (4x vs. H100) also accelerates non-TC computations