Architecting Tensor Core-Based Reductions for Irregular Molecular Docking Kernels

Leonardo Solis-Vasquez
Technical University of Darmstadt
Darmstadt, Germany
solis@esa.tu-darmstadt.de

Andreas F. Tillack
Scripps Research
La Jolla, USA
atillack@scripps.edu

Diogo Santos-Martins
Scripps Research
La Jolla, USA
diogom@scripps.edu

Andreas Koch
Technical University of Darmstadt
Darmstadt, Germany
koch@esa.tu-darmstadt.de

Stefano Forli
Scripps Research
La Jolla, USA
forli@scripps.edu

Abstract

Tensor Cores (TCs) are specialized hardware units designed for efficient matrix multiplication and are widely utilized in deep learning workloads. However, their adoption in more irregular high-performance computing (HPC) applications remains limited. This paper presents a methodology for effectively integrating TCs into a representative HPC application: molecular docking with AutoDock-GPU. The irregular computational patterns and strict accuracy requirements of this application pose significant challenges for TC utilization. To address these, we adopt a twofold strategy: (i) accelerating sum reduction operations using TCs, and (ii) applying state-of-the-art numerical error correction (EC) techniques to maintain accuracy. Experimental evaluations on NVIDIA A100, H100, and B200 GPUs show that our CUDA-based implementation consistently outperforms the baseline while preserving algorithmic accuracy.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; **Massively parallel algorithms**; • **Applied computing** → *Chemistry*.

Keywords

Tensor cores, variable execution performance, reduction, molecular docking, AutoDock

ACM Reference Format:
Leonardo Solis-Vasquez, Andreas F. Tillack, Diogo Santos-Martins, Andreas Koch, and Stefano Forli. YYYY. Architecting Tensor Core-Based Reductions for Irregular Molecular Docking Kernels. In *Preprint: Accepted for presentation at IA3 2025 (Held with SC'25). The final version will appear in the official workshop proceedings.*. ACM, New York, NY, USA, 11 pages. https://doi.org/WW.XXXX/XXXXXX.YYYYYYY

1 Introduction

General matrix multiplication (GEMM) is a fundamental computational pattern in both high-performance computing (HPC) and deep learning applications. The growing demand for high-throughput GEMM operations—particularly for training and inference in large-scale deep neural networks that underpin emerging large language models—has driven hardware vendors to develop dedicated accelerators to support these workloads [1, 5, 32].

Several commercial hardware accelerators have been developed to optimize GEMM operations. Notable examples include NVIDIA Tensor Cores (TCs) [25], Google Tensor Processing Units (TPUs) [6], Intel Xe Matrix Extensions (XMXs) [12], and AMD Versal AI Engines [3]. Among these, NVIDIA TCs are the most widely adopted, as they are integrated into general-purpose GPU (GPGPU) architectures. To date, NVIDIA has released five generations of TCs—starting with Volta and continuing through Turing, Ampere, and Hopper [31, 32], up to the most recent Blackwell architecture [23]. Successive generations have introduced increased arithmetic throughput and support for a wider range of data formats. For example, the H100 PCIe GPU achieves up to 756 $\frac{\text{TFLOP}}{\text{s}}$ of theoretical half-precision floating-point (FP16) performance using TCs (without using their sparsity feature), representing approximately a 7.4× speedup compared to the general-purpose non-TC computing units (SMs) on the same device [24], and a 2.4× increase over the previous-generation A100 with TCs [22].

Due to their quickly growing compute capabilities, TCs have spurred various efforts, including: (a) algorithmic redesigns that exploit specific arithmetic operations supported by TCs [2], (b) development of programming guidelines for implementing custom applications [32], and (c) comprehensive benchmarking across GPU generations [31]. Despite these advancements, TCs remain intrinsically tightly specialized, limiting their applicability to a relatively narrow set of domains. Initially, only GEMM-based algorithms utilized TCs. However, in recent years, their use has expanded to include certain non-GEMM operations, notably key arithmetic primitives such as reduction and scan—commonly used in data-parallel applications [1, 5]. This suggests that, although TCs support a constrained set of compute operations, they offer compelling opportunities for broader adoption beyond deep learning, particularly in HPC applications [4].

Molecular docking represents a compelling application domain within HPC. As a key method in computer-aided drug design, it simulates close-range interactions between two molecules with known three-dimensional structures and aims to predict their binding poses (i.e., spatial arrangements) with favorable binding energy. The two interacting molecules are typically referred to as the *ligand*

SC Workshops '25, St Louis, MO, USA

ACM ISBN XXXXXXXX/YYYY/ZZ
https://doi.org/WW.XXXX/XXXXXX.YYYYYYY

(a small molecule) and the *receptor* (a macromolecule). Molecular docking is used to identify ligands with antiviral properties that bind to a receptor modeling a given biological target (e.g., a protein or nucleic acid) [11]. One of the most widely used molecular docking applications is AutoDock [9]. It explores the pose space through a systematic search composed of multiple irregular nested loops with variable upper bounds. Search refinement is guided by the score of each pose, which quantifies the strength of the molecular interaction. This score is calculated using computationally intensive models and is typically evaluated 10^6 times during the search iterations.

The high computational demands of AutoDock have attracted significant interest in parallelizing and accelerating the application. To that end, the official parallel implementation, AutoDock-GPU [29], was initially developed in OpenCL [7] and later migrated to CUDA to run on the *Summit* supercomputer for COVID-19 research [27]. Since its release, AutoDock-GPU has undergone several performance enhancements for both CPUs and GPUs [14, 15], and has also been ported to a variety of heterogeneous architectures, including FPGAs [13] and vector processors [16]. Moreover, AutoDock-GPU has served as a case study in migration experiments to alternative heterogeneous programming models. These include corresponding mini-apps implemented in HIP and Kokkos [18], as well as our recent work [17], which enables the execution of a *unified* SYCL-based AutoDock-GPU codebase on high-end GPUs from multiple vendors.

Given its complex code structure and high practical relevance, AutoDock-GPU serves as a compelling case study within the HPC domain for leveraging the compute capabilities of NVIDIA Tensor Cores (TCs) and evaluating their performance and algorithmic impact. The key contributions of this work are summarized as follows:

(1) We demonstrate how the CUDA codebase of AutoDock-GPU can be adapted to leverage NVIDIA TCs, outlining the modifications required to offload compute-intensive regions—such as sum reduction operations—to TCs, while integrating numerical error correction (EC) techniques for preserving the expected algorithmic accuracy.
(2) We evaluate the performance of our TC- and EC-enhanced CUDA implementation (TCEC), and analyze the impact of different TC generations available on the A100, H100, and B200 GPUs.

The remainder of this paper is structured as follows. Section 2 provides background on AutoDock-GPU, and Section 3 reviews prior efforts in leveraging TCs that serve as a baseline for our work. Section 4 presents our implementation, while Section 5 evaluates its performance on high-end NVIDIA GPUs. Finally, Section 6 concludes the paper with a summary.

2 Background on AutoDock-GPU

Here, we provide a relevant overview of AutoDock-GPU's functionality. Detailed descriptions of the program's fundamentals and the evolution of its codebase can be found in prior work [7, 14, 15].

2.1 Functionality Overview

AutoDock-GPU performs a systematic search based on genetic evolution heuristics, wherein each ligand pose is treated as an individual within a population. Each individual is represented by a genotype, which consists of a set of genes describing the translation, orientation, and torsion of the ligand during the docking process.

The computational core of AutoDock-GPU is an irregular Lamarckian Genetic Algorithm (LGA) that performs a hybrid search by combining a genetic algorithm (GA) with a local search (LS). These two LGA phases generate new individuals from the current population, but employ different methods. The genetic algorithm applies genetic operations—namely crossover, mutation, and selection—while the local search uses minimization techniques to further improve the score of the molecular interaction. As presented in Algorithm 1, AutoDock-GPU executes several independent LGA runs (default: $N^{\text{TOTAL}}_{\text{LGA-runs}} = 100$). Each LGA run terminates when one of the predefined upper bounds is reached—either the maximum number of score evaluations (default: $N^{\text{MAX}}_{\text{score-evals}} = 2\,500\,000$) or the maximum number of generations (default: $N^{\text{MAX}}_{\text{gens}} = 27\,000$).

Algorithm 1: Lamarckian Genetic Algorithm (LGA)

```
1 Function AutoDock-GPU
      /* Coarse-Level Parallelism */
2     par for each LGA-run in N^TOTAL_LGA-runs do
3         while (N_score-evals < N^MAX_score-evals) and (N_gens < N^MAX_gens) do
4             GA (population)
5             LS (population)
```

Scores, expressed in $\frac{\text{kcal}}{\text{mol}}$, quantify the strength (i.e., energy) of molecular interactions and are computed for every pose during both LGA phases. Algorithm 2 presents the code structure of the scoring function (SF), which comprises three components. The `PoseCalculation` phase transforms the genotypes into atomic coordinates, which are subsequently used to compute the ligand-receptor (`InterScore`) and ligand-ligand (`IntraScore`) score components. The upper bounds of the corresponding loops depend on the molecular structure of the input—specifically, the number of elements in the rotation list ($N_{\text{rot-list}}$), the number of ligand atoms (N_{atom}), and the number of intramolecular contributor pairs ($N_{\text{intra-contrib}}$).

Algorithm 2: Scoring Function (SF)

```
   /* Fine-Level Parallelism */
1  Function SF (genotype)
2      par for each rot-item in N_rot-list do
3          PoseCalculation
4      par for each lig-atom in N_atom do
5          InterScore
6      par for each intra-pair in N_intra-contrib do
7          IntraScore
```

Of the two LGA phases in AutoDock-GPU, the local search (LS) is the most time-consuming, typically accounting for more than 90% of the total execution time. Among the alternative LS methods featured in AutoDock-GPU, ADADELTA [19] achieves the

best results in terms of molecular docking quality and is therefore the focus of this work. As presented in Algorithm 3, ADADELTA generates a new genotype using the gradients of the current genotype's score (see Algorithm 3, `Line 4`). Then, if the score of the new genotype is minimized, it replaces the current genotype (see Algorithm 3, `Line 6`). ADADELTA terminates when the number of iterations reaches a predefined maximum (default: $N^{\text{MAX}}_{\text{LS-iters}} = 300$).

Algorithm 3: ADADELTA (AD) local search

```
   /* Coarse-Level Parallelism */
1  Function AD (genotype)
2      gradient = GC (genotype)
3      par while (N_LS-iters < N^MAX_LS-iters) do
4          new-genotype = update-rule (genotype, gradient)
5          if SF (new-genotype) < SF (genotype) then
6              genotype = new-genotype
7          gradient = GC (genotype)
```

A fundamental computation in ADADELTA is the gradient calculation (GC), whose code structure, presented in Algorithm 4, resembles that of the scoring function (see Algorithm 2). First, `PoseCalculation` computes the atomic coordinates, which are then used to calculate the numerical and analytical (`InterGradient` and `IntraGradient`, respectively) derivatives of the score components. At this stage, these derivatives are represented as a list of atomic contributions. However, since the overall LGA search operates on genotypes, it is necessary to convert the atom-based contributions into gene-based ones. This conversion is performed by `Gtrans`, `Grigidrot`, and `Grotbond` (see Algorithm 4, `Lines 8-10`), which are loops that execute data-dependent operations to compute the translational, orientational, and rotational components of the gradient.

Algorithm 4: Gradient Calculation (GC)

```
   /* Fine-Level Parallelism */
1  Function GC (genotype)
       /* Gradients in atomic space */
2      par for each rot-item in N_rot-list do
3          PoseCalculation
4      par for each lig-atom in N_atom do
5          InterGradient
6      par for each intra-pair in N_intra-contrib do
7          IntraGradient
       /* Convert from atomic into genetic space */
8      Gtrans // Translational gradients
9      Grigidrot // Rigid-body rotation gradients
10     Grotbond // Rotatable-bond gradients
```

2.2 Parallelization

As previously noted, AutoDock-GPU was originally developed in OpenCL [7], and later ported to CUDA [27] and SYCL [17]. These implementations are similar and all follow a *Single Instruction, Multiple Thread* (SIMT) programming model.

Table 1 illustrates how AutoDock-GPU's computations are mapped onto CUDA processing elements across different levels of parallelism. In general, AutoDock-GPU executes $N^{\text{TOTAL}}_{\text{LGA-runs}}$ independent LGA runs, indexed by $\text{Run}_{\text{ID}} = \{0, 1, 2, \ldots, N^{\text{TOTAL}}_{\text{LGA-runs}} - 1\}$. Within each LGA run, both the genetic algorithm and local search operate on a population of size Pop_{size}, with individuals indexed by $\text{Ind}_{\text{ID}} = \{0, 1, 2, \ldots, \text{Pop}_{\text{size}} - 1\}$. The core idea behind AutoDock-GPU's parallelization strategy is to *simultaneously* process individuals from different LGA runs. Based on this mapping scheme, each of the $N^{\text{TOTAL}}_{\text{LGA-runs}} \times \text{Pop}_{\text{size}}$ individuals is assigned to a distinct CUDA thread block, with the block index defined as $\text{Block}_{\text{ID}} = \text{Run}_{\text{ID}} \times \text{Pop}_{\text{size}} + \text{Ind}_{\text{ID}}$. Moreover, fine-grained tasks—such as genotype generation, scoring and gradient calculation—are executed by CUDA threads.

Table 1: Mapping of AutoDock-GPU's computations onto CUDA processing elements, with corresponding parallelization levels indicated in Algorithms 1–4

Computation	CUDA element	Parallelization level
Genetic algorithm, Local search	Kernel	Coarse
Individual	Thread block	Medium
Generation, Scoring, Gradient	Thread	Fine

3 Related Work on Accelerating AutoDock-GPU with Tensor Cores

As discussed in Section 2.1, the ADADELTA algorithm relies on gradient calculation (GC), which performs seven block-level *sum reductions* per iteration. In this context, Schieffer and Peng [8] observed a significant number of warp stalls caused by these reductions. According to their execution profiles, approximately 40% of the observed warp stalls were attributed to memory barriers, while 25% were due to shared memory instruction latency.

To mitigate the high synchronization overhead associated with warp stalls, Schieffer and Peng [8] proposed leveraging NVIDIA TCs to accelerate the aforementioned sum reductions. In the original AutoDock-GPU implementation, the seven reductions are executed sequentially, each targeting a single variable. Similarly, prior TC-based reduction methods [1, 5] also focused on reducing one variable at a time. Building on this foundation, Schieffer and Peng introduced a matrix-based, *multidimensional* reduction algorithm—outlined in Equation (1)—that *simultaneously* reduces four-element vectors, i.e., $\{x, y, z, e\}$. To this end, their work [8] defines three 16×16 matrices: A, P, and Q, as specified in Equations 2. Specifically, matrix A represents the input data to be reduced. It contains the first 64 vectors—e.g., $\{x_0, y_0, z_0, e_0\}$, ..., $\{x_{63}, y_{63}, z_{63}, e_{63}\}$—arranged in a column-major layout. Matrix P is filled with ones, while matrix Q is structured as a 16×16 matrix composed of 4×4 blocks, each of which is a 4×4 identity matrix, denoted I_4.

Equation (1) presents the algorithm proposed in [8], which consists of two main computational steps. First, the matrix product $A \times P$ performs a summation across the rows of matrix A (e.g., $\sum x_{4i} = x_0 + x_4 + \ldots + x_{60}$), as shown in Equation (3). If more than 64 vectors must be reduced, the same operation is repeated iteratively: matrix A is repopulated with the next batch of 64 vectors from the input, while matrix V accumulates intermediate results. Second, the matrix product $Q \times V$ computes the sum of every fourth element in each column of matrix V and stores the result in matrix W (e.g., $\sum x_i = \sum x_{4i} + \sum x_{4i+1} + \sum x_{4i+2} + \sum x_{4i+3}$), as shown in

Equation (4). At this stage, the first column of matrix W contains the final reduction results for the four variables $\{x, y, z, e\}$.

$$\begin{cases} V \leftarrow A \times P + V \\ W \leftarrow Q \times V \end{cases} \tag{1}$$

$$A = \begin{pmatrix} x_0 & x_4 & \dots & x_{60} \\ y_0 & y_4 & \dots & y_{60} \\ z_0 & z_4 & \dots & z_{60} \\ e_0 & e_4 & \dots & e_{60} \\ \hline x_1 & x_5 & \dots & x_{61} \\ y_1 & y_5 & \dots & y_{61} \\ z_1 & z_5 & \dots & z_{61} \\ e_1 & e_5 & \dots & e_{61} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline x_3 & x_7 & \dots & x_{63} \\ y_3 & y_7 & \dots & y_{63} \\ z_3 & z_7 & \dots & z_{63} \\ e_3 & e_7 & \dots & e_{63} \end{pmatrix}_{16\times16} \quad P = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}_{16\times16}$$

$$Q = \begin{pmatrix} I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \end{pmatrix}_{16\times16} \quad I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}_{4\times4} \tag{2}$$

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

$$W = \begin{pmatrix} \sum x_i & \sum x_i & \dots & \sum x_i \\ \sum y_i & \sum y_i & \dots & \sum y_i \\ \sum z_i & \sum z_i & \dots & \sum z_i \\ \sum e_i & \sum e_i & \dots & \sum e_i \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix}_{16\times16} \tag{4}$$

In NVIDIA TC programming, the mapping between threads and TCs follows a 32-to-1 ratio—i.e., one warp maps to a single TC unit—unlike the 1-to-1 mapping typically observed between threads and conventional FP32 SIMT cores [5]. The implementation of Equation (1) by Schieffer and Peng [8] leverages the C++-based *Warp Matrix Multiply-and-Accumulate* (WMMA) API [20], which exposes TC functionality in CUDA kernels and ensures portability across NVIDIA GPU architectures. Specifically, the WMMA API provides operations for matrix load (`load_matrix_sync`), matrix store (`store_matrix_sync`), and matrix multiply-and-accumulate (`mma_sync`). These operations work on *fragments*, where each fragment is a C++ class representing a small, fixed-size, two-dimensional matrix distributed across all threads in a warp. In their implementation, Schieffer and Peng [8] operate on fragments consisting of 16×16 FP16 elements.

Our work builds upon the foundational efforts of Schieffer and Peng [8], whose algorithm serves as our baseline. We extend their CUDA codebase to enhance both performance and algorithmic accuracy, as detailed in Section 4.

4 Algorithmic Analysis and Numerical Corrections for Tensor Cores

AutoDock-GPU performs all computations using single-precision floating-point (FP32) arithmetic. During the implementation of Schieffer and Peng's algorithm [8], we observed that utilizing TCs for GEMM operations requires converting input matrices from FP32 to FP16, thereby introducing truncation errors due to reduced numerical precision. Nonetheless, Schieffer and Peng [8] reported that their TC-based implementation produced docking scores deviating by less than 0.2% from the FP32 baseline, concluding that the observed accuracy was acceptable for their application.

However, as they did not perform an in-depth analysis of docking quality, we conducted our own more extensive, domain-specific algorithmic analysis—employing 42 ligand-receptor test cases compared to the five used in [8]. Our study revealed *significantly reduced* accuracy in the molecular predictions produced by our initial implementation of Schieffer and Peng's TC-based reduction algorithm. This analysis is based on the E_{50} metric, introduced in our prior work [7], which quantifies the number of score evaluations ($N_{\text{score-evals}}$, see Section 2) required to attain a 50% probability of finding a global minimum, i.e., the optimal score for a given ligand-receptor pair. In AutoDock-GPU, insufficient score evaluations lower the likelihood of finding this global optimum, while a sufficient number asymptotically approaches a success probability of 100%.

In [7], we defined search success using two complementary criteria: one based on the docking score and the other on the root-mean-square deviation (RMSD) from the experimentally determined native pose obtained via X-ray crystallography. According to the score-based criterion, an LGA run is considered successful if the resulting pose yields a score within 1.0 $\frac{\text{kcal}}{\text{mol}}$ of the global minimum. Under the RMSD-based criterion, an LGA run is considered successful if the predicted pose lies within 2 Å of the native pose. Figure 1 presents the results of our algorithmic analysis, comparing the E_{50} values obtained over varying numbers of score evaluations for two AutoDock-GPU versions: our initial TC-based CUDA implementation of Schieffer and Peng's algorithm (y-axis) and a reference OpenCL implementation (x-axis). In this plot, algorithmic equivalence between the two implementations would manifest as E_{50} markers lying on or near the dashed diagonal line. However, as shown in Figure 1, most markers lie above the diagonal, indicating that the TC-based CUDA implementation requires *more* score evaluations than the OpenCL reference to achieve comparable success rates. This outcome indicates a *reduction* in algorithmic accuracy introduced by Schieffer and Peng's approach.

Prior studies have investigated the numerical accuracy *degradation* associated with mixed-precision floating-point computations on TCs, and have proposed corresponding mitigation techniques [4, 28]. Among these efforts, Ootomo and Yokota introduced an error correction (EC) algorithm that improves upon existing state-of-the-art methods [10]. In particular, they identified the

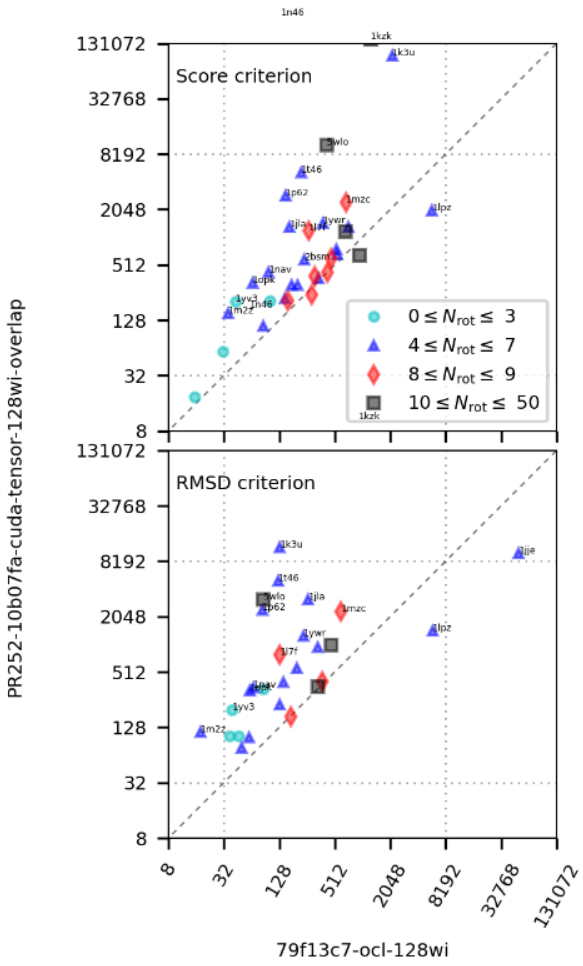

Figure 1: Comparison of E_{50} values between the initial TC-based CUDA implementation and the OpenCL reference. Results are shown for both evaluation criteria: score (top) and RMSD (bottom). Axis labels indicate the number of score evaluations required to achieve a 50% probability of success

Rounding-toward-Zero (RZ) behavior inherent to TC operations as a key contributor to numerical accuracy loss. This rounding mode is applied in the accumulator following the addition to C_{FP32}:

$$D_{\text{FP32}} \leftarrow A_{\text{FP16}} \times B_{\text{FP16}} + C_{\text{FP32}} \tag{5}$$

To avoid the effects of RZ and improve numerical accuracy, Ootomo and Yokota proposed performing the accumulation to C_{FP32} in Equation (5) *outside* the TCs, i.e., using FP32 SIMT cores for the addition instead. Figure 2 illustrates the difference between the standard accumulation method and the approach proposed by Ootomo and Yokota, which replaces *Rounding-toward-Zero* (RZ) with *Rounding-to-Nearest* (RN). Furthermore, the authors introduced two complementary algorithmic enhancements: (i) reducing the probability of underflow through input scaling, and (ii) improving the performance of EC by eliminating negligible EC terms.

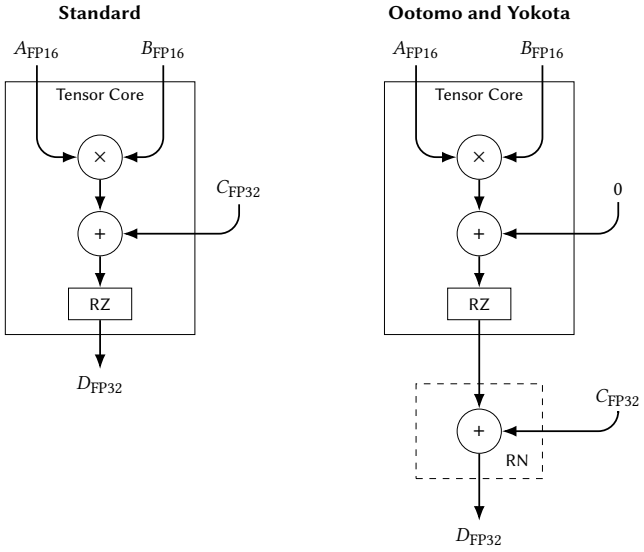

Figure 2: Comparison of TC usage methods: standard (left) vs. Ootomo and Yokota's approach (right) [10]. In the standard method, RZ is applied directly to the accumulator C_{FP32}. In contrast, Ootomo and Yokota's method—adopted in this work—avoids RZ within TCs by performing accumulation *externally* on FP32 SIMT cores using RN. This diagram is reproduced from [10] for explanatory purposes

All the enhancements proposed by Ootomo and Yokota in [10] are available through *WMMA-Extension* [26], an open-source CUDA library that encapsulates GEMM operations implemented using either NVIDIA's high-level WMMA API or the newer, low-level MMA interface. Listing 1 presents our implementation of the first half of Equation (1), which performs the matrix multiplication and accumulation operation $V_{\text{M×N}} \leftarrow A_{\text{M×K}} \times P_{\text{K×N}} + V_{\text{M×N}}$, where M = N = K = 16. In particular, Listing 1 compares the corresponding API calls between WMMA-Extension and the standard NVIDIA WMMA. The only required changes in programming effort are: (i) including the WMMA-Extension header file `tcec.hpp`, and (ii) switching between the `mtk::wmma::tcec` and `nvcuda::wmma` namespaces.

Furthermore, with regard to input datatypes for TCs, WMMA-Extension supports not only FP16 but also TensorFloat-32 (TF32), which features an 8-bit exponent and a 10-bit mantissa. Notably, TF32 shares the same exponent width as FP32 and provides a 3-bit wider exponent than FP16, thereby improving dynamic range. According to the representation accuracy experiments conducted by Ootomo and Yokota [10], TF32 achieves accuracy levels comparable to FP32 and significantly higher than FP16. Based on these findings, our TCEC implementation adopts TF32 as the input datatype (Listing 1, `Line 15`). Figure 3 presents the results of the corresponding algorithmic analysis, demonstrating that the achieved search success is now comparable to that of the OpenCL reference. A key learning of our work here is that numerically sensitive algorithms targeted to lower-precision compute elements such as TCs

Listing 1: CUDA implementations of the first half of Equation (1) using TCs. Top: TCEC based on the WMMA-Extension API, which supports GEMM and EC [26]. Bottom: baseline using NVIDIA's WMMA API for GEMM *without* EC

```cpp
#include <mma.h>
using namespace nvcuda;
using matrix_a   = wmma::matrix_a;
using matrix_b   = wmma::matrix_b;
using matrix_c   = wmma::accumulator;
using layout_ab = wmma::col_major;
wmma::layout_t layout_c  = wmma::mem_col_major;
constexpr int M = 16, N = 16, K = 16;

#ifdef 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);
#else
    /* 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);
#endif
```

do require error correction techniques like TCEC to maintain their quality-of-results.

5 Evaluation

We evaluate the performance gains of our TCEC-enhanced CUDA implementation of AutoDock-GPU, relative to the CUDA baseline that exclusively utilizes FP32 SIMT cores. Both implementations were executed on high-end NVIDIA GPUs: A100, H100, and B200. Table 2 summarizes key architectural characteristics of these devices. The evaluation uses a set of 42 ligand-receptor test cases from [30], covering molecules with up to 32 rotatable bonds (AutoDock-GPU currently supports up to $N_{\text{rot}}^{\text{MAX}} = 57$).

Table 2: Key characteristics of the NVIDIA GPU accelerators evaluated in this study. All cards are in PCIe form factor

Characteristics	A100	H100	B200
Arch. | Comp. Capability	Ampere | 8.0	Hopper | 9.0	Blackwell | 10.0
SMs $\times$ $\frac{\text{FP32 SIMT cores}}{\text{SM}}$	108 × 64	114 × 128	264 × 128
SMs $\times$ $\frac{\text{TCs}}{\text{SM}}$	108 × 4	114 × 4	264 × 4
FP32 Theo. Perf. (SM) [$\frac{\text{TFLOP}}{\text{s}}$]	19.49	51.22	80
TF32 Theo. Perf. (TC) [$\frac{\text{TFLOP}}{\text{s}}$]	155.92	378.00	1200
Memory Bandwidth [$\frac{\text{TB}}{\text{s}}$]	1.56	2.04	8.00

Note. TF32 theoretical performance assumes that the sparsity feature of TCs is not utilized. Moreover, TF32 computations in WMMA-Extension are implemented with `wmma` instructions, which do not support sparsity.

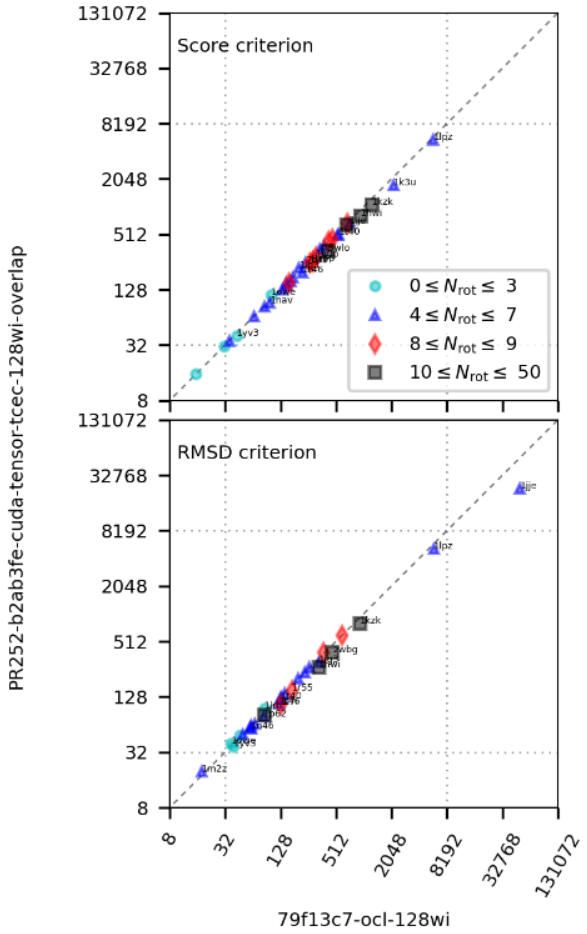

Figure 3: Comparison of E_{50} values between the TCEC CUDA implementation and the OpenCL reference. Across both criteria—score (top) and RMSD (bottom)—the two implementations require a similar number of score evaluations to reach 50% probability of success

5.1 Performance Evaluation

Our experiment involves executing simulations for the complete set of 42 test cases, each configured with $N_{\text{LGA-runs}}^{\text{TOTAL}} = 20$. For each test case, we collect the following key metrics, as listed in Table 3: the actual number of score evaluations performed ($N_{\text{score-evals}}^{\text{Actual}}$); the best score achieved with its corresponding RMSD; the best RMSD achieved with its associated score; and the docking runtime, defined as the total *program-level* LGA execution time, including both GPU kernel execution and the required host-GPU data transfers. Due to the stochastic nature of the search space traversal employed by AutoDock-GPU, the docking runtime varies across executions, and might not be a stable metric in all comparisons. To address this, we normalize the docking runtime by $N_{\text{score-evals}}^{\text{Actual}}$ (which itself exhibits variability) and use the resulting metric, expressed in microseconds per evaluation ($\frac{\mu\text{s}}{\text{eval}}$), as our primary performance indicator. Table 3 shows the metrics collected for the 7cpa test case, where the baseline

implementation yields 0.911 $\frac{\mu s}{\text{eval}}$ ($\frac{2.30\ \text{s}}{2\,523\,871\ \text{evaluations}}$), while TCEC achieves 0.791 $\frac{\mu s}{\text{eval}}$ ($\frac{1.99\ \text{s}}{2\,516\,957\ \text{evaluations}}$). This indicates that, for this test case, the TCEC implementation requires approximately 15% less time per score evaluation compared to the baseline.

Table 3: Metrics collected for the 7cpa test case executed on the A100 GPU. Both CUDA implementations use thread blocks of size 64. Runtime statistics are based on 100 execution samples

CUDA impl.	$N^{\text{Actual}}_{\text{score-evals}}$	Best score @RMSD	Best RMSD @score	Docking runtime Min | Max | Avg | Std.Dev.
Baseline	2 523 871	-21.36 $\frac{\text{kcal}}{\text{mol}}$ 1.90 Å	1.90 Å -21.36 $\frac{\text{kcal}}{\text{mol}}$	2.30 s 2.25 | 2.38 | 2.28 | 0.02
TCEC	2 516 957	-20.11 $\frac{\text{kcal}}{\text{mol}}$ 3.66 Å	2.06 Å -18.88 $\frac{\text{kcal}}{\text{mol}}$	1.99 s 1.98 | 2.10 | 2.00 | 0.02

Figure 4 presents the speedup factors achieved by both TCEC and baseline implementations for three CUDA block sizes: 64, 128, and 256 threads per block. **Absolute speedup factors** (shown as bars) represent the performance gains of each configuration—TCEC or baseline on any GPU—relative to the non-TC SM-only baseline implementation running on the A100. **Relative speedup factors** (indicated by red arrows) capture the performance gains of TCEC over its corresponding baseline on the same GPU. To give a sample interpretation: Running the 42 test-cases on a B200 using Tensor Cores and Error Correction (TCEC) runs 2.08× faster than on an A100 using non-TC SM-only code. Using the TCEC approach on the B200 gives a 35% performance gain versus just using the SMs on the same platform.

In general, larger thread block sizes and newer GPU architectures yield higher absolute speedups. An exception is observed for the baseline implementation on the H100 with 256 threads per block, where the absolute speedup is 0.90×, indicating slower performance compared to the A100 baseline. In terms of relative speedups, the highest improvement is achieved on the H100 with 256 threads per block, reaching a 1.63× speedup, equivalent to a 63% performance gain. All relative speedup factors exceed 1.00×, indicating that on each GPU, the TCEC implementation consistently outperforms its baseline SM-only counterpart, despite the overhead of Error Correction.

5.1.1 Performance Model. The previously reported speedups provide a basis for comparing the execution times across different GPU generations. Equation (6), derived from Amdahl's Law, models the predicted speedup as a function of two variables: (i) S, the theoretical acceleration factor of TCs relative to FP32 SIMT Cores, and (ii) f, the fraction of workload (ranging from 0 to 1) that is offloaded to TCs.

$$\text{Predicted Speedup}_{\text{AD-GPU}} = \frac{T_{\text{Baseline}}}{T_{\text{TCEC}}} = \frac{1}{\frac{f}{S} + (1 - f)} \tag{6}$$

The variable S is derived from Table 2 as the ratio between the theoretical throughput of TCs and that of FP32 SIMT cores. This yields acceleration factors of 8.0× (= $\frac{155.92}{19.49}$) for the A100, 7.4× (= $\frac{378.00}{51.22}$) for the H100, 15.0× (= $\frac{1200}{80}$) for the B200. The variable

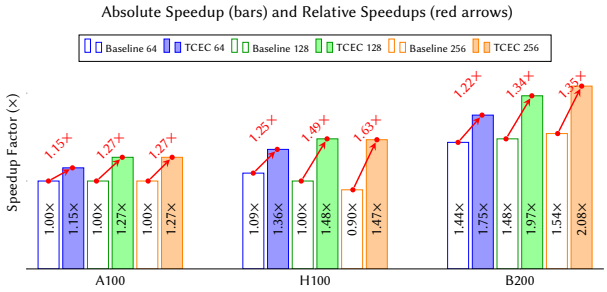

Figure 4: Absolute and relative speedup factors for three CUDA block sizes: 64, 128, and 256 threads per block. Absolute speedups represent the performance of TCEC and baseline implementations—on any GPU—relative to the baseline on the A100. Relative speedups reflect the performance gains of TCEC over its corresponding non-TC SM-only baseline on the same GPU. All speedups are computed from the aggregated performance ratios (in $\frac{\mu s}{\text{eval}}$) across the full set of 42 test cases

f represents the fraction of the workload executed on TCs, while the remaining portion $(1 - f)$ is executed on FP32 SIMT cores. Table 4 presents predicted speedups for various values of f across the evaluated GPUs. As shown, achieving approximately *half* of the theoretical maximum performance improvement requires a high utilization of TCs (i.e., $f \geq 0.9$).

Table 4: Predicted speedup factors computed using Equation (6) for selected values of TC usage (f), with $f = 0$ no time spent using TCs, $f = 1.0$ all time spent using TCs

f	**A100** (S = 8.0×)	**H100** (S = 7.4×)	**B200** (S = 15.0×)
0.0	1.00×	1.00×	1.00×
0.2	1.21×	1.20×	1.25×
0.9	3.55×	3.42×	8.57×
1.0	8.00×	7.40×	15.00×

To estimate f, we instrument AutoDock-GPU using built-in CUDA function `clock64()`, which returns the value of a 64-bit counter that increments every clock cycle on a per-Streaming Multiprocessor (SM) basis. The value of f is then computed by dividing the number of clock cycles spent in the seven sum reduction regions (which are offloaded to TCs, see Section 3) by the total number of clock cycles consumed by the entire ADADELTA kernel. Since this kernel accounts for more than 90% of the overall docking runtime, we consider the effective accelerated fraction as $f_{\text{eff}} = 0.9f$ when applying Amdahl's law.

Table 5 reports performance measurements from AutoDock-GPU executions using the 7cpa test case (characterized by medium complexity, with $N_{\text{rot}} = 15$), and thread block sizes of 64, 128, and 256. Despite its simplicity, Equation (6) yields predicted *program-level* speedups that remain closely align with measured values, both in magnitude and trend. Interestingly, the H100 achieves higher speedups than the newer and more powerful B200, as observed both for the 7cpa test case at 256 threads (Table 5) and across the full set of 42 test cases (Figure 4). We attribute this behavior to two

main factors: (i) on H100, the FP32 SIMT-only baseline execution is relatively slower, making the performance gains from TCs more pronounced; and (ii) the B200's significantly higher memory bandwidth (about 4× greater than that of the H100, as shown in Table 2), which also accelerates non-TC computations, thereby reducing the relative speedup attributable to TCs.

Table 5: Predicted and measured speedups of AutoDock-GPU for the 7cpa test case. Predicted speedups are computed using Equation (6), while measured speedups are obtained by dividing the baseline performance ratio by its TCEC counterpart (both in $\frac{s}{eval}$)

GPU	*Block Size*	f_{eff}	S	Pred. Speedup	Baseline $[\frac{s}{eval}]$	TCEC $[\frac{s}{eval}]$	Meas. Speedup
A100	*64*	0.11	8.0×	1.11×	$\frac{2.30}{2\,523\,871}$	$\frac{1.99}{2\,516\,957}$	1.15×
	128	0.14		1.14×	$\frac{2.38}{2\,513\,042}$	$\frac{1.83}{2\,509\,770}$	1.30×
	256	0.14		1.14×	$\frac{2.99}{2\,510\,114}$	$\frac{2.31}{2\,523\,529}$	1.30×
H100	*64*	0.12	7.4×	1.12×	$\frac{1.94}{2\,515\,609}$	$\frac{1.64}{2\,506\,748}$	1.18×
	128	0.14		1.14×	$\frac{2.15}{2\,515\,699}$	$\frac{1.56}{2\,508\,832}$	1.37×
	256	0.16		1.16×	$\frac{3.04}{2\,500\,704}$	$\frac{1.95}{2\,515\,840}$	1.57×
B200	*64*	0.16	15×	1.18×	$\frac{1.42}{2\,514\,822}$	$\frac{1.20}{2\,516\,053}$	1.18×
	128	0.18		1.20×	$\frac{1.65}{2\,520\,228}$	$\frac{1.13}{2\,504\,668}$	1.45×
	256	0.14		1.15×	$\frac{1.72}{2\,519\,439}$	$\frac{1.32}{2\,508\,752}$	1.30×

5.2 Execution Profiling

To demonstrate *kernel-level* performance gains across all GPUs and block-size configurations, Table 6 reports profiling metrics from a single execution of the ADADELTA kernel (test case: 7cpa) using NVIDIA Nsight Compute [21]. As expected, execution time decreases with newer hardware and is lower for TCEC compared to its corresponding baseline. The B200 shows the shortest execution time, reaching 38.7 ms at its best configuration (TCEC, 64 threads), followed by the H100 and A100 with 49.9 ms and 72.8 ms, respectively—also under TCEC with 64 threads. While not depicted here, profiling results from Nsight Compute indicate that both the TCEC and baseline implementations are compute-bound, according to the roofline model. Operational intensity (OI) increases across GPU generations, with the B200 achieving the highest OI of 3620.7 $\frac{FLOP}{Byte}$ (TCEC, 256 threads) and the highest performance of 2568.5 $\frac{GFLOP}{s}$ (TCEC, 64 threads). In terms of $\frac{GFLOP}{s}$, TCEC consistently outperforms the baseline across all evaluated block sizes and hardware platforms. Specifically, for block sizes of {64, 128, 256}, TCEC yields performance gains of {17.4%, 37.7%, 39.8%} on the A100, {27.3%, 46.4%, 74.4%} on the H100, and {30.0%, 41.1%, 40.7%} on the B200.

Fused Multiply-Add (FMA), Arithmetic Logic Unit (ALU), and Tensor Core (TC) utilization statistics quantify the number of cycles during which the corresponding hardware units are active. FMA and ALU utilization factors tend to decrease with increasing block size, particularly for the baseline implementation. Among the three GPUs, the A100 exhibits the highest FMA utilization (35.2%) under TCEC with 128 threads, while the B200 maintains moderate FMA usage but benefits more from TCs. Notably, the B200 achieves the highest TC utilization, reaching 4.7% under TCEC with 256 threads—surpassing both the A100 and H100. For all executions of the baseline implementation, TC utilization was expected to remain at 0%, as no user-level computations explicitly target these units. However, unexpectedly nonzero TC utilization—ranging from 0% to 1%—was observed on the A100 and H100. We attribute this discrepancy to differences in the versions of Nsight Compute used for profiling: v2023.3.1 for the A100, v2023.2.2 for the H100, and v2025.1.1 for the B200.

Table 6: Performance metrics from a single ADADELTA kernel execution for the 7cpa test case

CUDA block size	*64*	*128*	*256*	*64*	*128*	*256*
A100	**Baseline**			**TCEC**		
Exec. time [ms]	82.9	95.9	124.8	72.8	74.6	96.6
OI $[\frac{FLOP}{Byte}]$	1363.8	1588.7	1578.0	1390.9	1701.1	1711.8
Perf. $[\frac{GFLOP}{s}]$	1152.4	1035.0	823.9	1353.6	1425.3	1152.5
FMA Util. [%]	28.5	26.0	20.6	33.7	35.2	28.1
ALU Util. [%]	17.4	16.7	14.7	23.2	25.8	23.3
TC (FP) Util. [%]	0.8	0.9	0.9	1.5	2.7	3.3
H100	**Baseline**			**TCEC**		
Exec. time [ms]	61.6	70.6	104.7	49.9	51.1	66.5
OI $[\frac{FLOP}{Byte}]$	2735.0	2627.9	2633.0	2833.3	2573.0	2758.8
Perf. $[\frac{GFLOP}{s}]$	1583.3	1365.4	946.6	2016.7	1999.8	1651.5
FMA Util. [%]	14.3	12.3	8.7	19.4	19.7	15.9
ALU Util. [%]	17.1	15.6	12.2	25.3	26.4	24.0
TC (FP) Util. [%]	0.4	0.3	0.2	2.1	3.0	3.8
B200	**Baseline**			**TCEC**		
Exec. time [ms]	50.5	59.1	70.7	38.7	41.6	52.3
OI $[\frac{FLOP}{Byte}]$	3399.2	3318.3	3495.1	3411.2	3488.3	3620.7
Perf. $[\frac{GFLOP}{s}]$	1960.1	1641.0	1441.4	2568.5	2447.4	2028.8
FMA Util. [%]	14.1	12.5	11.0	18.4	18.8	15.9
ALU Util. [%]	16.1	15.0	14.6	23.6	24.6	21.6
TC (FP) Util. [%]	0.0	0.0	0.0	3.1	4.0	4.7

6 Conclusions

In this work, we presented a Tensor Core-accelerated implementation of AutoDock-GPU with integrated numerical error correction (TCEC) to mitigate the precision loss introduced by FP16-based GEMM operations. Building on the prior work by Schieffer and Peng, our extended algorithmic analysis revealed reduced docking accuracy in their TC-based approach. To address this, we applied a TF32-based error correction strategy using the WMMA-Extension library, which restores accuracy to levels comparable to the OpenCL reference. Experimental results on high-end NVIDIA GPUs demonstrate that TCEC consistently outperforms the FP32 SIMT-core baseline. For docking runtime across 42 molecular test cases with 256 threads per block, TCEC achieves performance gains of 27% on the A100, 63% on the H100, and 35% on the B200.

References

[1] A. Dakkak, C. Li, J. Xiong, I. Gelado, W. Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *ACM International Conference on Supercomputing*. ACM, 46 – 57. doi:10.1145/3330345.3331057

[2] A. Haidar, S. Tomov, J. Dongarra, N.J. Higham. 2018. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 603 – 613. doi:10.1109/SC.2018.00050

[3] AMD. 2025. AI Engine: Meeting the Compute Demands of Next-Generation Applications. https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/ai-engine.html

[4] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, Y. Ding. 2021. EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 278 – 291. doi:10.1145/3437801.3441599

[5] C. Navarro, R. Carrasco, R.J. Barrientos, J.A. Riquelme, R. Vega. 2021. GPU Tensor Cores for Fast Arithmetic Reductions. *IEEE Transactions on Parallel and Distributed Systems* 32, 1 (2021), 72 – 84. doi:10.1109/TPDS.2020.3011893

[6] Google Cloud. 2025. TPU architecture. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

[7] D. Santos-Martins, L. Solis-Vasquez, A.F. Tillack, M.F. Sanner, A. Koch, S. Forli. 2021. Accelerating AutoDock4 with GPUs and Gradient-Based Local Search. *J. Chem. Theory Comput.* 17, 2 (2021), 1060–1073. doi:10.1021/acs.jctc.0c01006

[8] G. Schieffer, I. Peng. 2023. Accelerating Drug Discovery in AutoDock-GPU with Tensor Cores. In *Euro-Par 2023: Parallel Processing: 29th International Conference on Parallel and Distributed Computing*. Springer-Verlag, 608 – 622. doi:10.1007/978-3-031-39698-4_41

[9] G.M. Morris, D.S. Goodsell, R.S. Halliday, R. Huey, W.E. Hart, R.K. Belew, A.J. Olson. 1998. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry* 19, 14 (1998), 1639 – 1662. doi:10.1002/(SICI)1096-987X(19981115)19:14<1639::AID-JCC10>3.0.CO;2-B

[10] H. Ootomo, R. Yokota. 2022. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications* 36, 4 (2022), 475 – 491. doi:10.1177/10943420221090256

[11] I. Halperin, B. Ma, H. Wolfson, R. Nussinov. 2002. Principles of docking: An overview of search algorithms and a guide to scoring functions. *Journal of Proteins: Structure, Function, and Bioinformatics* 47, 4 (2002), 409 – 443. doi:10.1002/prot.10115

[12] Intel. 2025. Intel Xe GPU Architecture. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/intel-xe-gpu-architecture.html

[13] L. Solis-Vasquez, A. Koch. 2018. A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software. In *5th International Workshop on FPGAs for Software Programmers (FSP)*. VDE Verlag, 1 – 10. https://ieeexplore.ieee.org/document/8470463

[14] L. Solis-Vasquez, A.F. Tillack, D. Santos-Martins, A. Koch, S. LeGrand, S. Forli. 2022. Benchmarking the performance of irregular computations in AutoDock-GPU molecular docking. *Parallel Comput.* 109 (2022), 102861. doi:10.1016/j.parco.2021.102861

[15] L. Solis-Vasquez, D. Santos-Martins, A.F. Tillack, A. Koch, J. Eberhardt, S. Forli. 2020. Parallelizing Irregular Computations for Molecular Docking. In *10th International Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 12 – 21. doi:10.1109/IA351965.2020.00008

[16] L. Solis-Vasquez, E. Focht, A. Koch. 2021. Mapping Irregular Computations for Molecular Docking to the SX-Aurora TSUBASA Vector Engine. In *11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 1 – 10. doi:10.1109/IA354616.2021.00008

[17] L. Solis-Vasquez, E. Mascarenhas, A. Koch. 2023. Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study. In *2023 International Workshop on OpenCL*. ACM. doi:10.1145/3585341.3585372

[18] M. Thavappiragasam, A. Scheinberg, W. Elwasif, O. Hernandez, A. Sedova. 2020. Performance Portability of Molecular Docking Miniapp On Leadership Computing Platforms. In *International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 36 – 44. doi:10.1109/P3HPC51967.2020.00009

[19] M.D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. *arXiv* abs/1212.5701 (2012). doi:10.48550/arXiv.1212.5701

[20] NVIDIA. 2024. Warp Matrix Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-matrix-functions

[21] NVIDIA. 2025. Nsight Compute. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[22] NVIDIA. 2025. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[23] NVIDIA. 2025. NVIDIA Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture

[24] NVIDIA. 2025. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-data-center-overview/gtc22-whitepaper-hopper

[25] NVIDIA. 2025. NVIDIA Tensor Cores - Unprecedented Acceleration for Generative AI. https://www.nvidia.com/en-us/data-center/tensor-cores

[26] H. Ootomo. 2024. wmma_extension - An extension library of WMMA API (Tensor Core API). https://github.com/wmmae/wmma_extension

[27] S. LeGrand, A. Scheinberg, A.F. Tillack, M. Thavappiragasam, J.V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez, A. Koch, S. Forli, O. Hernandez, J.C. Smith, A. Sedova. 2020. GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research. In *11th International Conference on Bioinformatics, Computational Biology and Health Informatics*. ACM. doi:10.1145/3388440.3412472

[28] S. Markidis, S.W.D. Chien, E. Laure, I.B. Peng, J.S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522 – 531. doi:10.1109/IPDPSW.2018.00091

[29] Scripps Research. 2025. AutoDock-GPU: AutoDock for GPUs and other accelerators. https://github.com/ccsb-scripps/AutoDock-GPU

[30] Scripps Research. 2025. Set of 42 protein-ligand complexes for testing search algorithms and docking runtime. https://github.com/diogomart/AD-GPU_set_of_42

[31] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, X. Chu. 2024. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 656 – 667. doi:10.1109/IPDPS57955.2024.00064

[32] W. Sun, A. Li, T. Geng, S. Stuijk, H. Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2023), 246 – 261. doi:10.1109/TPDS.2022.3217824

A Artifact Appendix

A.1 Abstract

This artifact appendix provides instructions for obtaining, compiling, and executing the AutoDock-GPU code developed in this work. It describes the procedures used to: 1) collect docking runtimes (i.e., program-level execution times accounting for GPU execution and the required host-GPU data transfers), and 2) count score evaluations. Both metrics form the basis of the performance analysis conducted on NVIDIA A100, H100, and B200 GPUs.

A.2 Artifact check-list (meta-information)

- **Algorithm:** CUDA-based parallelization of AutoDock-GPU, which offloads sum-reductions in the ADADELTA kernel to Tensor Cores.
- **Program:** AutoDock-GPU (all sources can be downloaded from GitHub), commit: `a46ab564`, size: ~102 MB.
- **Compilation:** g++ 6 or above.
- **Binary:** Source code and scripts included to generate binaries.
- **Data set:** Molecular structures prepared for AutoDock-GPU (all input files can be downloaded from GitHub), status: ready to use, size: ~1.5 GB.
- **Run-time environment:** AutoDock-GPU requires any Linux distribution supporting CUDA. We recommend Ubuntu 22.04 or 24.04. NVIDIA drivers are required. No need of root access.
- **Hardware:** We recommend NVIDIA A100, H100, and B200 GPUs.
- **Execution:** Sole user. AutoDock-GPU benchmarks on all chosen three GPUs take ~4 hours.
- **Metrics:** docking runtimes in seconds, and number of score evaluations.
- **Output:** Console indicating docking runtimes of a given experiment. Additional: docking log files (*.dlg) indicating above runtimes and number of score evaluations performed.
- **Experiments:** Execution commands (provided). Maximum allowable variation of docking runtimes: 10%.
- **How much disk space required (approximately)?:** ~50 GB.
- **How much time is needed to prepare workflow (approximately)?:** one hour.
- **How much time is needed to complete experiments (approximately)?:** 24 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GNU LGPL.

A.3 Description

A.3.1 How to access. The AutoDock-GPU source code is publicly available on GitHub at: https://github.com/ccsb-scripps/AutoDock-GPU.

A.3.2 Hardware dependencies. For reproducibility and comparable results, we recommend using NVIDIA A100, H100, or B200 GPUs. Nonetheless, AutoDock-GPU is compatible with any CUDA-capable GPU.

A.3.3 Software dependencies. We tested the implementation using CUDA Toolkit v12.3 on the A100, v12.2 on the H100, and v12.8 on the B200. All runs require the appropriate NVIDIA CUDA drivers, with installation instructions available from NVIDIA. The code has been validated on Ubuntu 22.04, which exhibited no compatibility issues. Other Linux distributions—and potentially Unix-based systems—may also be supported, although they have not been extensively tested.

A.3.4 Data sets. The molecular input data sets used for benchmarking are available at: https://github.com/diogomart/AD-GPU_set_of_42. This repository includes a curated set of 42 molecules.

A.4 Installation

(1) Clone AutoDock-GPU repository:

```
$ git clone https://github.com/ccsb-scripps/AutoDock-GPU.git
```

(2) Ensure the commit `a46ab564` is used:

```
$ cd AutoDock-GPU
$ git checkout a46ab564
```

(3) Retrieve the WMMA-Extension source code, which provides integrated support of Tensor Core (TC) operations and Error Correction (EC):

```
$ git submodule update --init --recursive
```

(4) Compile a single AutoDock-GPU binary:

```
$ make DEVICE=GPU NUMWI=<NWI> TENSOR=<STATUS>
```

where: `<NWI>` denotes the thread-block size (e.g., 64, 128, 256), and `<STATUS>` is either `ON` or any other string to enable or disable Tensor Core support, respectively. The resulting binaries are placed in the `bin/` folder and are named according to the selected thread-block size, e.g., `autodock_gpu_64wi`.

(5) Clone the data set repository:

```
$ git clone https://github.com/diogomart/AD-GPU_set_of_42
```

(6) Extract the data archives (this step is only required once):

```
$ for dir in ./data/* ; do (cd $dir && gunzip protein.*.map.
    gz); done
```

A.5 Experiment workflow

Execute the AutoDock-GPU binary (`./bin/autodock_gpu_64wi`) by specifying the following command-line arguments:

- `-ffile`: path to the receptor file
- `-lfile`: path to the ligand file
- `-nrun`: number of LGA runs
- `-lsmet ad`: enables ADADELTA local search
- `-A`: *autostop* configuration (`0`: disabled, 1: enabled)
- `-H`: *heuristics* configuration (`0`: disabled, 1: enabled)
- `-resnam`: name of the docking log output file

```
$ ls
AD-GPU_set_of_42  AutoDock-GPU
$ cd AutoDock-GPU
./bin/autodock_gpu_64wi
-ffile ../AD-GPU_set_of_42/data/7cpa/protein.maps.fld
-lfile ../AD-GPU_set_of_42/data/7cpa/rand-0.pdbqt
-nrun 100 -lsmet ad -A 0 -H 0 -resnam ad_7cpa_cuda
```

A.6 Evaluation and expected results

The results of each execution are stored in the generated *.dlg files. Inspect these files to retrieve docking runtimes and number of score evaluation performed:

```
$ grep "Run time" *.dlg
$ grep "Number of energy evaluations performed" *.dlg
```

A.7 Experiment customization

The experimental configuration can be customized by modifying the command-line arguments when executing AutoDock-GPU. For detailed instructions, refer to the README.md file in the AutoDock-GPU source code repository.

A.8 Methodology

The artifact appendix for this paper was submitted according to the guidelines at https://ctuning.org