Accelerating Drug Discovery in AutoDock-GPU with Tensor Cores

Gabin Schieffer and Ivy Peng(✉)

KTH Royal Institute of Technology, Stockholm, Sweden
{gabins,ivybopeng}@kth.se

Abstract. In drug discovery, molecular docking aims at characterizing the binding of a drug-like molecule to a macromolecule. AutoDock-GPU, a state-of-the-art docking software, estimates the geometrical conformation of a docked ligand-protein complex by minimizing a scoring function. Our profiling results indicate that the current reduction operation that is heavily used in the scoring function is sub-optimal. Thus, we developed a method to accelerate the sum reduction of four-element vectors using matrix operations on NVIDIA Tensor Cores. We integrated the new reduction operation into AutoDock-GPU and evaluated it on multiple chemical complexes on three GPUs. Our results show that our method for reduction operation is 4–7 times faster than the AutoDock-GPU baseline. We also evaluated the impact of our method on the overall simulation time in the real-world docking simulation and achieved a 27% improvement on the average docking time.

Keywords: Molecular docking · AutoDock · GPU · Tensor Core · Drug Discovery

1 Introduction

The pharmacological effect of a drug is generally induced by the binding of a drug molecule to a specific protein target. Thus, characterizing the ability of binding is crucial for drug discovery. Once a target for a disease is identified, tens of millions of chemical compounds, or *ligands*, will go through high-throughput screening. For such vast search space, virtual screening that leverages computational approaches is becoming increasingly important for accelerating the process and reducing the high cost required in experimental screenings [5,12]. In particular, structure-based virtual screening software uses molecular docking tools to test a molecule drug candidate for binding a protein target (receptor). In recent COVID-19 research, high-performance virtual screening software has been used in combating the pandemic [5].

A typical molecular docking job consists of evaluating a large number of ligands, each as an independent docking task. Further distributing individual docking tasks onto high-performance computing (HPC) systems, with multi-core CPU or GPUs, can significantly accelerate docking, e.g., AutoDock-GPU reports

J. Cano et al. (Eds.): Euro-Par 2023, LNCS 14100, pp. 608–622, 2023.
https://doi.org/10.1007/978-3-031-39698-4_41

350-fold speedup over single-threaded implementation [8,12]. AutoDock is widely used in the pharmaceutical industry to characterize protein-ligand complexes. In recent efforts, AutoDock4 implements its search engine based on Lamarckian Genetic Algorithm (LGA) and is ported to GPUs. A CUDA implementation of AutoDock-GPU with enhanced workflow successfully scaled to leverage the Summit supercomputer [5].

In this work, we focus on the CUDA implementation of AutoDock-GPU as it represents the state-of-the-art of docking software on HPC systems. AutoDock-GPU predicts the geometrical conformation of a ligand-protein complex by minimizing an energy-based *scoring* function that quantifies the free energy of a given binding pose. A docking job typically have many LGA runs, each consisting of multiple iterations till reaching the max number of score evaluations or GA generations. Therefore, the scoring function is called many times, e.g., 10^6 to 10^8, in a docking job, dominating the runtime [12]. The scoring function parallelizes the computation of the energy and associated gradient values by distributing iterations across all threads in a block and computing the total energy in a block-level reduction operation. Our profiling results show that the current implementation of the reduction operation causes a significant proportion of the overall number of warp stalls in the local search kernel.

We propose a Tensor Core based reduction operation to accelerate the docking process – leveraging Tensor Core Units and reducing synchronization points. We designed a multi-dimensional reduction algorithm based on previous works [1,10]. Our design leverages compacted data layout in shared memory. By merging multiple matrix multiplications into a single one, we dramatically reduce the number of synchronization points. We implemented the new algorithm in CUDA using the Nvidia WMMA API and integrated it in the energy calculation function in AutoDock-GPU. We validated the implementation and then evaluated its performance in single kernel and overall docking time on three generations of NVIDIA GPUs, including T4, V100, and A100. The results show that our method consistently outperform the AutoDock-GPU baseline, achieving up to 6.7× and 4.7× speedup on A100 and V100, respectively. We summarize our contributions as follows:

- Our performance charaterization of the AutoDock-GPU identified the scalability bottleneck in reduction operation in scoring function
- We proposed a multi-dimension reduction operation leveraging the mixed-precision Tensor Core Units
- We provided an implementation in CUDA using WMMA API in AutoDock-GPU and validated the implementation
- We evaluated the performance within single kernel on three GPUs and achieved 4.1–6.7× speedup, and a 27% improvement on average docking time

2 Background

In this section, we introduce the computation method in molecular docking and the GPU implementation of AutoDock-GPU. We also introduce Tensor Core Unit and its programming interfaces on NVIDIA GPUs.

2.1 Computational Method in AutoDock-GPU

AutoDock [9] variants, e.g., AutoDock-Vina, AutoDock4, and AutoDock-GPU, use an energy-based scoring function to measure the quality of a given binding pose. The scoring function is a free-energy force field. It captures contributions from various physical interactions between atom pairs to associate an energy value to a ligand-receptor conformation. Recent development [12] introduces different search algorithms, such as the Solis-Wets and the ADADELTA methods, to accelerate the docking.

In the docking method in AutoDock-GPU, the target molecule is fixed. Thus, the ligand-receptor complex can be fully described by a set of variables related to the position, rotation, and internal conformation of the ligand. This set of variables, referred as *ligand pose* or *genotype*, is composed of seven dimensions, i.e., x, y, z representing the ligand's position in space, ϕ, θ, α characterizing the rotation of the ligand, and N_{rot} dimensions characterizing the torsion angles of rotatable bonds in the ligand by $\psi_1 \dots \psi_{N_{rot}}$. These variables are the input to the scoring function.

AutoDock-GPU uses a parallelized version of the original LGA [12]. The LGA uses a genetic algorithm (GA) to perform a global search, which generates several genotypes (denoted as Ω). Each genotype is then improved by a local search algorithm (LS) that minimizes the scoring function (free energy). Two commonly used local search algorithms are ADADELTA and Solis-Wets. ADADELTA [17] is a gradient-based optimization algorithm. It updates the genotype Ω at each iteration t by $\Omega_{t+1} = \Omega_t + \eta_t g_t$, where η_t depends on the history of previous update and gradient values, and g_t is the gradient of the scoring function at the point Ω_t. The computational cost of this method is dominated by the gradient calculation. AutoDock-GPU parallelizes computation of the energy value by distributing iterations across all threads in a block. Each thread computes a partial value of the total energy and a block-level reduction is used to compute the total energy value. Similarly, each thread computes a partial value of the gradient for each of the three geometrical dimensions x, y, z, as well as the torque generated by physical interactions on the ligand, which is required for the calculation of the rotation-related and torsion-related gradient values. In total, seven block-level reductions are required for each evaluation of the scoring function, during the local-search optimization process.

2.2 NVIDIA Tensor Cores

NVIDIA Tensor Cores were introduced in the Volta GPU microarchitecture, providing tremendous computing power in reduced precision [6]. NVIDIA V100 features 640 first-generation Tensor Cores and a theoretical peak performance of 125 Tflops/s in mixed precision. The Turing architecture extended Tensor Cores abilities by adding support for computation using more data types. The Tesla T4 offers 320 Tensor Cores, and provides a theoretical peak performance of 65 Tflops/s. In the Ampere architecture, the A100 GPU features 432 Tensor Cores, and provides a theoretical peak performance of 312 Tflops/s.

Tensor Core Units (TCU) are designed to perform matrix multiply-and-accumulate operations (i.e., $V \leftarrow A \cdot B + V$) in high throughput, while enforcing constraints on matrix sizes and precision. The operands of the multiplication operation must be of size 16×16 and contain half-precision elements [11]. The accumulator can use single-precision float representation.

Tensor Core operations use the *half-precision* data type, which relies on a 16-bit binary representation. This level of precision is generally sufficient for deep learning workloads, and scientific workloads resilient to precision loss can also benefit from it. However, the *half-precision* data type requires explicit conversion to the single-precision 32-bit float representation. Starting with the Ampere GPU architecture, NVIDIA added support for both *bfloat16* and *tf32* in Tensor Cores. While double-precision data type is also supported on Tensor Cores from the Ampere GPU architecture, the matrix size in this precision is limited to 8×4 for the multiplication operands, and 8×8 for the accumulator.

The WMMA API (*Warp Matrix Multiply-and-Add*) provides a limited set of functions for developers to use Tensor Cores. Codes using this API are portable across different NVIDIA GPU architecture. This API exposes functions to set up and perform multiply-and-accumulate operations on Tensor Cores. It defines a data structure named *fragment*. A fragment is an abstraction to represent a matrix. Each fragment holds the matrix metadata, i.e., the data type, the matrix size, and the type of matrix as either an operand or an accumulator. The actual matrix elements held by a fragment are spread across threads in the warp, this data-to-threads mapping is not known by the developer [1]. Instead, the WMMA API provides basic load and store functions to map generic CUDA data structures, such as arrays, to fragments. A multiply-and-accumulate operation is exposed as a function operating on fragments and requires the collaboration of all threads in a warp.

3 Performance Characterization on GPU

In this section, we first provide an overview of the runtime breakdown of a simulation and then focus on the GPU computation. We used the `7cpa` protein-ligand complex and ran with a block size of 64 threads on NVIDIA A100 GPU, using all default parameters. The profiling results were obtained with NVIDIA Nsight Systems. At high level, the runtime of a simulation is dominated by the docking time, which is GPU bound, and then I/O pre-processing [7]. In Fig. 1, NVIDIA Nsight Systems reports 90% time spent in docking.

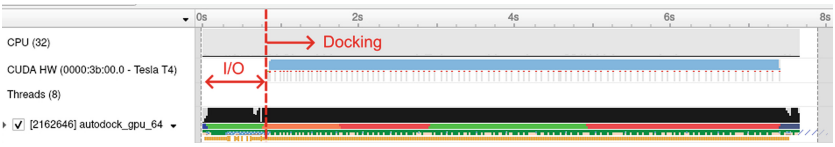

Fig. 1. Profiling results of a docking process of the `7cpa` protein-ligand complex.

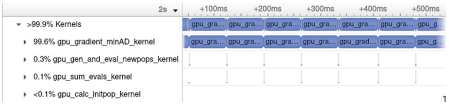

Fig. 2. The kernel launch timeline for iterations of the optimization process.

Table 1. Time breakdown in CUDA kernels

kernel name	% of total kernel runtime
gpu_calc_initprop_kernel	<0.1%
gpu_sum_evals_kernel	0.1%
gpu_gen_and_eval_newpops_kernel	0.3%
gpu_gradient_minAD_kernel	99.6%

#	Source	Live Registers	Warp Stall Sampling (All Cycles)	Warp Stall Sampling (Not-issued Cycles)	Instructions Executed
730	REDUCEFLOATSUM(torque_rot.x, pFloatAccumulator);	37	56,731	44,951	20,402,426
731	REDUCEFLOATSUM(torque_rot.y, pFloatAccumulator);	36	57,131	45,490	20,400,606
732	REDUCEFLOATSUM(torque_rot.z, pFloatAccumulator);	36	60,279	48,058	20,399,346
739	REDUCEFLOATSUM(energy, pFloatAccumulator);	36	57,268	45,362	20,401,641

Fig. 3. Profiling results of the gpu_gradient_minAD_kernel kernel.

In the docking process, the runtime is dominated by the local-search kernel, gpu_gradient_minAD. As shown in Fig. 2, the gradient-based local search dominates the docking time on GPU, i.e., 99.6% kernel time is spent in the gpu_gradient_minAD kernel (the details are described in [12]). The breakdown of GPU kernel runtime is reported in Table 1. In this kernel, seven reduction operations are performed to compute the value and gradient of the scoring function, which happens at every iteration of the gradient-descent algorithm. This reduction operation is defined as a C++ macro named REDUCEFLOATSUM (denoted as *ReduceFS* in the remainder of this paper).

We observe a large number of warp stalls in each execution of *ReduceFS* in Fig. 3, which reports four consecutive calls of *ReduceFS* macro. Moreover, these lines of code are identified among the top ten lines of code causing high numbers of warp stalls, indicating that the stalls could have a high impact on overall kernel performance. From the causes for these warp stalls returned by NVIDIA Nsight Compute, we observe that approximately 40% of warp stalls are caused by memory barriers ("membar"), related to the use of memory fence operations. Also, about 25% of warp stalls are caused by "short scoreboard", which is often caused by shared memory instruction latency.

The profiling results led us to investigate further the block-level reduction in AutoDock-GPU. We established that REDUCEFLOATSUM(value, acc) performs a block-level reduce-and-broadcast operation. Each thread provides one single-precision number *value*, which will be reduced with all other values for other threads in the block. At the end of the reduction, the result is placed back in *value*. *acc* is a pointer to a float in shared memory, which is used internally as an accumulator to perform reduction.

The current implementation mainly relies on three CUDA functions – warp shuffle functions, atomic operations, and block-level synchronizations. First, a warp-level reduction is performed through warp shuffle functions, which allow data exchange between threads within a warp without using shared memory. In

particular, the __shfl_sync function allows a thread to read a value from another thread within the same warp, in a synchronized fashion.

In the warp-level reduction algorithm, this function is called multiple times by each thread. At each call, each thread adds the value received from another thread into its local copy. By organizing communication in a tree-like pattern, five consecutive calls to __shfl_sync are sufficient for each thread to have its own local copy of the total sum across all 32 threads (a warp). This warp-level reduction algorithm is state-of-the-art [1].

After the warp-level reduction is completed, the first thread of each warp performs an atomic add of the result to a shared memory accumulator. Finally, each thread in a thread block performs a read from the accumulator in shared memory to receive the reduction result, finishing the whole operation.

Takeaway 1: Atomic operations are used for value accumulation, and could cause contention when a large number of warps is used.

As described in Sect. 2, the scoring function implementation needs to perform reduction over seven dimensions – one for the global energy value, three for the gradient calculation, and three for the torque calculation. In the current AutoDock-GPU version, this is implemented by sequentially calling the *ReduceFS* macro seven times in the scoring function kernel.

Takeaway 2: each evaluation of the scoring function repeats the block-level reduction operation seven times sequentially.

For each use of *ReduceFS*, three explicit block-level thread synchronizations are performed, which results in a total of 21 synchronizations for the seven-dimensional reduction. This could drastically reduce the parallelism of the algorithm.

Takeaway 3: Performing reduction operation on seven dimensions separately results in 21 block-level synchronizations, a potential bottleneck for scalability.

4 Methodology

In this work, we leverage Tensor Core Units (TCU) to accelerate matrix-based reduction. In [1], scan and reduction operations on an array are expressed as matrix operations and accelerated on NVIDIA Tensor Cores. This method relies on placing the elements to be reduced in a matrix, which is then multiplied by a well-chosen matrix to perform summation on the rows. A similar operation is then applied to perform summation on the columns. This line-then-column summation process effectively sum up all elements, equivalent to performing a reduction operation.

We propose an approach to replace the reduction operation in AutoDock-GPU by an implementation of a reduction method which is able to leverage Tensor Core Units. We first list the requirements that our method must meet to be used in AutoDock-GPU code. Then, we describe how we adapt and optimize the general Tensor Core-based reduction operation to meet the specific requirements in AutoDock-GPU. It is worth noting here that even though the method

and implementation proposed in this paper are tailored to a specific application, the performed operation is general. Therefore, our approach can be generalized to other applications, with reasonable adaptation efforts.

4.1 Requirements and Design Choices

The scoring function in AutoDock-GPU performs seven consecutive reductions, each time for one variable. Previous TCU-based reduction method only reduces one variable at a time. To improve the efficiency, we propose to merge the reduction operations of four variables. This change would bring two main benefits. First, the profiling results show that a single reduction operation inherently requires synchronization between threads. Thus, merging four reductions would ideally reduce the synchronization cost by four times, improving parallelism. Second, we can improve the efficiency of data movement by reducing the number of separate data transfers. As introduced in Sect. 2.2, data arrays needs to be transferred (and mapped) from shared memory to be used on TCUs. By transforming the data layout into one contiguous data layout in shared memory, this overhead can be reduced.

The mapping between matrix elements and thread registers is not consistent across different GPU architectures. For this reason, NVIDIA recommends using the exposed API functions, i.e., load_matrix_sync() to load matrices data. When this function is called, each thread copies a portion of shared memory array to its registers. The matrix data is hence spread across all threads in the warp. This process may be sub-optimal in applications where matrices elements are already initially stored in registers, since those elements would first need to be copied to shared memory and then loaded to registers while they only need to be read back from registers. For this reason, previous work [3] has reverse-engineered the memory mapping between matrix elements and corresponding thread registers. Previous TCU-based reduction method [1] chose to use this knowledge to manipulate matrix data directly in registers.

In AutoDock-GPU, matrix elements are initially stored in each thread's registers. Thus, the reverse engineered memory mapping technique could squeeze more performance. However, this technique also requires specific tuning for each architecture. Therefore, for portability across different GPUs, we chose to use the NVIDIA-recommended approach.

4.2 Matrix-Based Multi-dimensional Reduction Method

We design a method using matrix operations to perform sum reduction of a set of four-element vectors. Our method aims at computing the sum of n four-element vectors $\boldsymbol{u_i} = (x_i, y_i, z_i, e_i)$. The result is also a four-element vector, which contains on each of its coordinates the sum for each corresponding dimension, i.e., $\boldsymbol{y_i} = (\sum_i x_i, \sum_i y_i, \sum_i z_i, \sum_i e_i)$. We represent our input data as a 16×16 matrix A, containing coordinates of the first 64 vectors, organized in a column-major fashion. We also declare two 16×16 matrices – P and Q. P is a matrix filled with ones. Q is a block-matrix composed of 4×4 blocks, each being the 4×4 identity matrix I_4.

$$A = \begin{pmatrix} x_0 & x_4 & \dots & x_{60} \\ y_0 & y_4 & \dots & y_{60} \\ z_0 & z_4 & \dots & z_{60} \\ e_0 & e_4 & \dots & e_{60} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad P = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix} \qquad Q = \begin{pmatrix} I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \\ I_4 & I_4 & I_4 & I_4 \end{pmatrix}$$

We first compute the matrix product AP into V. This operation effectively performs summation on the rows. If more than 64 vectors need to be reduced, we iterate the same operation, each time with A containing elements for a new set of 64 vectors in the input dataset and accumulating the results into V. We then perform sum on every 4^{th} column in V with the matrix operation QV and save the result into W. At this point, the matrix W contains the desired result as the four first elements on the first column.

$$V \leftarrow AP = \begin{pmatrix} \sum x_{4i} & \sum x_{4i} & \dots & \sum x_{4i} \\ \sum y_{4i} & \sum y_{4i} & \dots & \sum y_{4i} \\ \sum z_{4i} & \sum z_{4i} & \dots & \sum z_{4i} \\ \sum e_{4i} & \sum e_{4i} & \dots & \sum e_{4i} \\ \sum x_{4i+1} & \sum x_{4i+1} & \dots & \sum x_{4i+1} \\ \sum y_{4i+1} & \sum y_{4i+1} & \dots & \sum y_{4i+1} \\ \sum z_{4i+1} & \sum z_{4i+1} & \dots & \sum z_{4i+1} \\ \sum e_{4i+1} & \sum e_{4i+1} & \dots & \sum e_{4i+1} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \qquad \begin{array}{l} V \leftarrow AP + V \\ \\ W \leftarrow QV \\ \\ W = \begin{pmatrix} \sum x_i & \sum x_i & \dots & \sum x_i \\ \sum y_i & \sum y_i & \dots & \sum y_i \\ \sum z_i & \sum z_i & \dots & \sum z_i \\ \sum e_i & \sum e_i & \dots & \sum e_i \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \end{array}$$

We implement our method as a CUDA __device__ function using the NVIDIA WMMA API to perform matrix operations. This function replaces four sequential uses of the *ReduceFS* macro in the energy-and-gradient calculation in AutoDock-GPU. The four elements to be reduced for each thread are first converted from float to half-precision using the CUDA half2float function, and then loaded into a contiguous data array in shared memory. The data loading is collectively performed by all threads in a block.

The accumulator V is a product of matrices A and P. Meanwhile, it is also an operand for the matrix multiplication calculating W. Then, in order to compute W using TCUs, V must be half-precision. Using single precision for accumulation in V would require to convert it to half-precision before computing W, a casting back to single precision would then be necessary. This approach requires two non-trivial conversions between two levels of precision. Instead, we choose to use half-precision for both operations.

In our implementation, two block-level synchronizations are needed in total. A first one is performed before the first WMMA API call, to ensure that values for all threads are available in shared memory before starting the reduction process. The second synchronization is performed after the last WMMA API call, to ensure that all threads in the block can read the results. Compared to the 21 synchronizations in original AutoDock-GPU, our method significantly reduces synchronization points.

Our implementation requires no memory barriers and atomic operations, unlike the current AutoDock-GPU method. Note that those operations are

responsible for a significant number of stalls (Sect. 3). In addition, the decreased amount of those contention-causing operations could improve scalability.

5 Evaluation

We evaluated our implementation on four testbeds, featuring three GPU architectures, i.e., T4, V100, and A100. We summarize their system specifications in Table 2. Docking experiments were performed using five protein-ligand complexes, referred by their four-character Protein Data Bank identifier. We used the following complexes: `1stp`, `7cpa`, `1ac8`, `3tmn`, `3ce3`. Those five complexes, which are real-world samples, are provided with AutoDock-GPU code as test samples. Three of them were chosen for their particular molecular characteristics, in order to validate various aspects of the docking implementation, in particular the gradient calculation.

5.1 Validation of the Scoring Function

Our first step is to validate the TCU-based implementation in AutoDock-GPU scoring function. For this, we leverage similar metrics defined in [12] to evaluate the correctness in LGA run and overall simulations. In particular, we compare simulation results to the baseline results to quantify the precision loss introduced by the half-precision operations on TCU.

Figure 4 presents box-and-whisker plots for the best energy value reached by the scoring function, as reported by AutoDock-GPU. As the initialization process is random, we repeat 1000 runs for each protein-ligand complexes to increase the statistical significance as in [12]. For each run, the pseudo-random number generator is initialized with the same arbitrary seed for both our code, and the original code.

Table 3 reports the absolute and relative errors in the energy value from our method and the AutoDock-GPU baseline. For both `1ac8` and `3tmn`, the best energy values show no significant variance between runs for both implementations. For `1stp`, `7cpa`, and `3ce3`, the statistical distribution produced by our code is similar to the one produced by the original code. We notice that for all tested complexes, the relative difference between the average best scores for each method is below 0.18%. This observation leads us to conclude that our method provides satisfactory results, and thus validates our approach to perform reduction in the context of AutoDock-GPU. The justification for this conclusion is two-fold. First, the result of the reduction process is used as the energy value,

Table 2. A summary of four testbeds used for evaluation

Testbed	GPU	CPU	Interconnect	GPU Memory	CPU Memory
TB1	NVIDIA Tesla T4	16 core Intel(R) Xeon(R) Gold	PCIe	16 GB RAM	576 GB DDR4
TB2	NVIDIA Tesla V100 SXM2	8 core Intel(R) Xeon(R) Gold	NVLink	32 GB HBM2	768 GB DDR4
TB3	NVIDIA Tesla V100 SXM2	16 core Intel(R) Xeon(R) Gold	NVLink	32 GB HBM2	768 GB DDR4
TB4	NVIDIA Tesla A100	32 core Intel(R) Xeon(R) Gold	NVLink	40 GB HBM2	576 GB DDR4

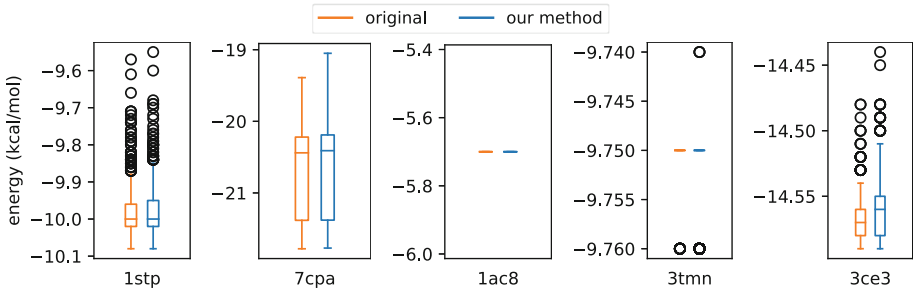

Fig. 4. Distribution of average best energy values for five protein-ligand complexes using the original code, and our method.

Table 3. Absolute difference and relative error in the best energy values and the speedup by our method compared with the baseline.

Complex	`1stp`	`7cpa`	`1ac8`	`3tmn`	`3ce3`
$\lvert E_{half} - E_{ref} \rvert$	$2.00 \cdot 10^{-5}$	$3.72 \cdot 10^{-2}$	0.0	$1.92 \cdot 10^{-3}$	$5.78 \cdot 10^{-3}$
Relative Error	<0.01%	0.2%	0.00%	0.02%	0.04%
Speedup	×1.16	×1.08	×1.22	×1.27	×1.20

thus a low difference with the reference value shows that our implementation provides a satisfactory level of accuracy for the application. Moreover, the result of the reduction process is used in further computations. Any detrimental error would thus accumulate, and the local-search algorithm would not yield satisfactory results, which is not the case in our tests.

5.2 Runtime Per Evaluation of the Scoring Function

Next, we evaluate the performance of a single evaluation function. To isolate the reduction process from the energy scoring function, we design a test kernel, where each thread in a block holds a single vector of four single-precision elements. The kernel performs a block-level reduce-and-broadcast operation over all threads. After the reduction operation, the final result is accessible by each thread in their respective local memory. We design two versions of the test kernel.

The first version uses the original AutoDock-GPU code. It first performs a warp-level reduction using warp shuffle functions, which allows to exchange data between threads without using shared memory. A block-level reduction is then performed, where the first thread of each warp adds the value it holds to a shared-memory accumulator, using an atomic operation. The value of the accumulator is then read back by all threads in the block. This three-step process is repeated for each variable that needs to be reduced. The second version of the test kernel uses our TCU-based method.

We measure the elapsed walltime for 1000 launches of each version using the CUDA Runtime API and report the average time. The only parameter

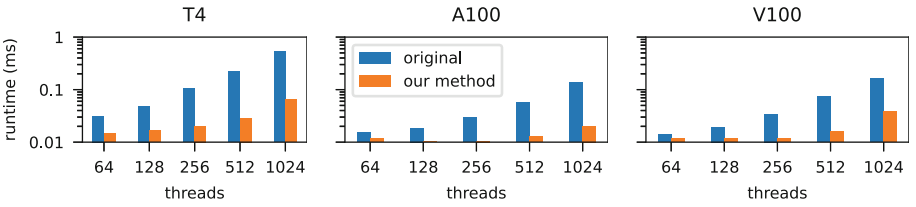

Fig. 5. Average runtime of the two versions of the test reduction kernel on three generations of NVIDIA GPUs: T4, A100, and V100.

influencing the runtime in both versions is the number of threads per block. 64 threads is the lower limit defined by our method – a 256-element matrix is used to store the values to be reduced, and each threads holds exactly four values, which results in a minimum of 64 threads to fill a single matrix. Future adaptation of the code may overcome this limitation. The upper limit of 1024 is defined by the CUDA platform [11].

Figure 5 shows the average runtime for both versions. The results show that our method consistently performs better than the AutoDock baseline for all block sizes and on all GPUs. This first observation validates the potential of our approach to perform faster block-level reduction in the context of the energy scoring function of AutoDock-GPU.

We notice that performance for both methods is significantly lower on T4 GPU than on A100 and V100. The lower performance for T4 can be explained by the lower performance Tensor Cores on T4. Performance on A100 and V100 are very similar utill the block size of 1024 threads. When using 1024 threads per block, a significant runtime difference is shown on the two GPUs – the runtime on A100 is 20 ms, which is half of the 39 ms runtime for V100. Our profiling results from NVIDIA Nsight Compute show that the test kernel achieved 100% occupancy on A100 but only 50% on V100. This low occupancy causes the device to be under-utilized. Such low theoretical occupancy indicates that the number of active threads per Streaming Multiprocessor is under the maximum achievable value because the resource requirements for the kernel are too high to be accommodated by the device. This could be, for example, the amount of available shared memory.

We evaluate the scalability of our method at increased threads per block. Figure 6 presents the speedup by our reduction method over the baseline on three GPU architectures. Figure 7 compares the execution times of local search kernel launches during a docking run, using our reduction method or the original method. We observe an increased speedup at an increased number of threads. For instance, the speedup increases from 2× at a block size of 64 on T4 to the maximum of 8.1× on 1024 threads. Overall, the speedup by our method increases linearly with the block size, up to 512 threads per block for all GPUs.

One interesting observation is that at the maximum block size of 1024 threads, the speedup on A100 increases to a maximum of 6.7× while the speedup on V100 decreases to 4.1×. Before reaching the maximum block size, speedup

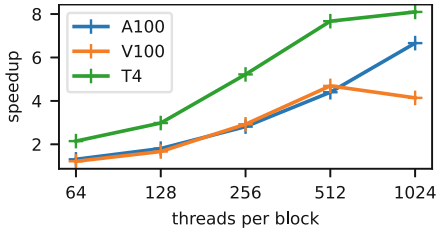

Fig. 6. Speedup of the reduction operation using our method over the AutoDock baseline on three GPUs.

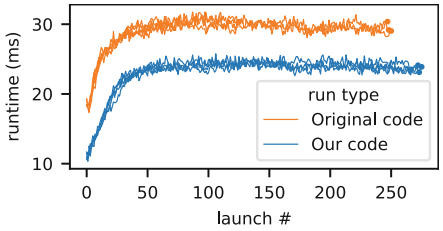

Fig. 7. Runtime of the local-search kernel, using our TCU-based method and AutoDock-GPU baseline.

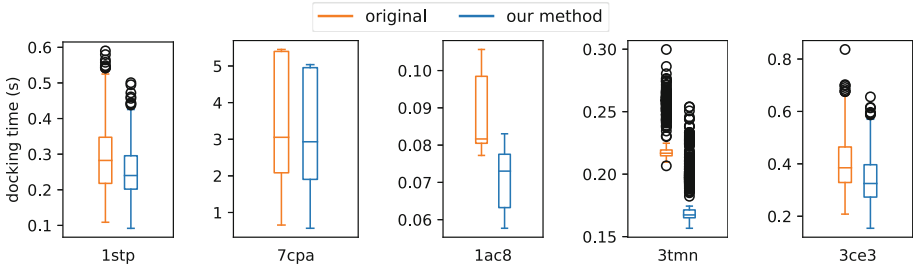

Fig. 8. Docking time on A100 for several protein-ligand complexes, using both the original code and our method.

on A100 and V100 GPUs show similar linear scalability. We investigate this and found from the runtime measurements that the amount of shared memory required when using 1024 threads per block exceeds the hardware limit on V100 GPU, thus resulting in a lower occupancy. Since the original method does not rely on shared memory, this bottleneck only affects our TCU-based method.

5.3 Impact on the Docking Time

We evaluate the contribution of our method on the overall simulation. For this, we integrated our block-level reduction method into the scoring function kernel in AutoDock-GPU. We use the docking time, a widely used figure of metric (FoM) in works on AutoDock-GPU [12,13]. The docking time is reported by AutoDock-GPU, including all docking executions and excluding the I/O operations.

Figure 8 shows the distribution of docking times for five protein-ligand complexes. Note that the docking time is significantly affected by the initial state, which is randomly chosen in AutoDock-GPU. Thus, for a fair comparison, we set the same random initialization seed for both methods. We also gather a large number of samples (1000 runs) to ensure statistical significance of the measurement. We observe that our method achieves a lower median, min, max, 25%, and 75% percentile docking time compared to the original version. This indicates

that our implementation is able to provide consistent speedup over the baseline for general cases.

Distribution of docking times for `7cpa` exhibits a larger interquartile range compared to the distribution observed for `1stp`. This difference is caused by the presence of a significant number of non-convergent runs in the experiments for `7cpa`. Non-convergent runs are observed when the search algorithm does not detect convergence, and continues until the maximum number of iterations is reached. This increased iteration count results in significantly higher docking time values for non-convergent runs when compared to convergent ones, for which the search algorithm is stopped earlier. We measured the proportion of non-convergent runs to be 61% for both versions, when using the `7cpa` complex. This indicates that our implementation does not have any impact on convergence of the search algorithm. Docking runs for other protein-ligand complexes did not exhibit non-convergent runs.

For all test cases, our implementation exhibits a lower average docking time compared to the original code. Table 3 (row 3) summarizes the speedup by our method over the original AutoDock-GPU code. We achieved a maximum $\times 1.27$ average speedup, observed for the `3tmn` complex. Speedup for the longest-running test case (`7cpa` complex) is $\times 1.08$.

6 Related Works

Molecular docking methods are widely used in drug discovery [4,9,14]. Various search techniques are used to find the best conformation between molecules [4], they rely on scoring functions that aim at evaluating the quality of a specific conformation [14]. AutoDock is a molecular docking program that relies on a genetic algorithm to find the docking conformation by minimizing a energy-based scoring function [9].

Several works have been conducted to accelerate the original AutoDock code. AutoDock Vina improved AutoDock's local-search method, and made use of multicore and multi-CPU systems to improve performance [16]. AutoDock-GPU added GPU acceleration to AutoDock by adapting the local-search method. Both OpenCL and CUDA versions have been developed. It provided up to a $\times 50$ speedup [12]. The recent addition of early stopping to AutoDock-GPU search algorithm allowed to further increase performance [13]. Once adapted for the Summit supercomputer, the CUDA version of AutoDock-GPU allowed to reach a $10\times$ speedup in a real-world docking pipeline [5]. Our work proposes a method to increase performance of the CUDA implementation of AutoDock-GPU, by using half-precision number representation in specific portions of the code.

Despite Tensor Cores being specialized in performing operations on small-size matrices, especially for deep learning applications, efforts have been made to make use of this hardware feature to accelerate other applications. For this purpose, algorithms to perform various widely-used operations on Tensor Cores have been developed, such as reduction and scan algorithms [1,10]. In our work, we adapted those methods in order to use them in AutoDock-GPU. Extensive

study of Tensor Cores characteristics have also been conducted. Benchmarking allowed to evaluate Tensor Cores performances in details [15]. The impact of using half-precision numbers for computation using Tensor Cores, and the associated accuracy loss, have also been documented and precision-refinement techniques have been developed [2,6].

7 Conclusions

In this work, we investigate a state-of-the-art GPU-accelerated molecular docking software for drug discovery – AutoDock-GPU. Our profiling results identified a core reduction operation to be sub-optimal due to a large number of synchronization points. We analyzed the specific requirements in the docking process and propose a matrix-based multi-dimensional reduction algorithm for accelerating the local search in AutoDock-GPU. We implemented our method by leveraging NVIDIA Tensor Cores and integrated it in AutoDock-GPU code. We validated our implementation and evaluated its performance on three GPUs. The results show a 4–7× speedup of the reduction operation and a 27% improvement on the average docking time for a real-world docking scenario.

Acknowledgments. This research is supported by the European Commission under the Horizon project OpenCUBE (GA-101092984).

References

1. Dakkak, A., Li, C., Xiong, J., Gelado, I., Hwu, W.M.: Accelerating reduction and scan using tensor core units. In: Proceedings of the ACM International Conference on Supercomputing, ICS 2019, pp. 46–57. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3330345.3331057
2. Haidar, A., Tomov, S., Dongarra, J., Higham, N.J.: 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, pp. 603–613. IEEE (2018)
3. Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D.P.: Dissecting the NVIDIA volta GPU architecture via microbenchmarking (2018)
4. Kitchen, D.B., Decornez, H., Furr, J.R., Bajorath, J.: Docking and scoring in virtual screening for drug discovery: methods and applications. Nat. Rev. Drug Discov. **3**(11), 935–949 (2004)
5. LeGrand, S., et al.: GPU-accelerated drug discovery with docking on the summit supercomputer: porting, optimization, and application to COVID-19 research. In: Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, BCB 2020, ACM (2020)
6. Markidis, S., Chien, S.W.D., Laure, E., Peng, I.B., Vetter, J.S.: NVIDIA tensor core programmability, performance & precision. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 522–531 (2018)
7. Markidis, S., Gadioli, D., Vitali, E., Palermo, G.: Understanding the I/O impact on the performance of high-throughput molecular docking. In: 2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW), pp. 9–14. IEEE (2021)

8. Mermelstein, D.J., Lin, C., Nelson, G., Kretsch, R., McCammon, J.A., Walker, R.C.: Fast and flexible GPU accelerated binding free energy calculations within the amber molecular dynamics package (2018)
9. Morris, G.M., et al.: Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. J. Comput. Chem. **19**(14), 1639–1662 (1998)
10. Navarro, C.A., Carrasco, R., Barrientos, R.J., Riquelme, J.A., Vega, R.: GPU tensor cores for fast arithmetic reductions. IEEE Trans. Parallel Distrib. Syst. **32**(1), 72–84 (2021)
11. NVIDIA: CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/element-types-and-matrix-sizes
12. Santos-Martins, D., Solis-Vasquez, L., Tillack, A.F., Sanner, M.F., Koch, A., Forli, S.: Accelerating AutoDock4 with GPUs and gradient-based local search. J. Chem. Theory Comput. **17**(2), 1060–1073 (2021)
13. Solis-Vasquez, L., Tillack, A.F., Santos-Martins, D., Koch, A., LeGrand, S., Forli, S.: Benchmarking the performance of irregular computations in AutoDock-GPU molecular docking. Parallel Comput. **109**, 102861 (2022)
14. Stanzione, F., Giangreco, I., Cole, J.C.: Use of molecular docking computational tools in drug discovery. Progr. Med. Chem. **60**, 273–343 (2021). https://doi.org/10.1016/bs.pmch.2021.01.004
15. Sun, W., Li, A., Geng, T., Stuijk, S., Corporaal, H.: Dissecting tensor cores via microbenchmarks: latency, throughput and numeric behaviors. IEEE Trans. Parallel Distrib. Syst. **34**(1), 246–261 (2022)
16. Trott, O., Olson, A.J.: AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. J. Comput. Chem. **31**, 455–461 (2009)
17. Zeiler, M.D.: ADADELTA: an adaptive learning rate method (2012). https://doi.org/10.48550/arXiv.1212.5701