Mapping Irregular Computations for Molecular Docking to the SX-Aurora TSUBASA Vector Engine

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

Erich Focht
NEC Deutschland GmbH
Stuttgart, Germany
erich.focht@emea.nec.com

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

***Abstract*—Molecular docking is a key method in computer-aided drug design, where the rapid identification of drug candidates is crucial for combating diseases. AutoDock is a widely-used molecular docking program, having an irregular structure characterized by a divergent control flow and compute-intensive calculations. This work investigates porting AutoDock to the SX-Aurora TSUBASA vector engine and evaluates the achievable performance on a number of real-world input compounds. In particular, we discuss the platform-specific coding styles required to handle the high degree of irregularity in both local-search methods employed by AutoDock. These Solis-Wets and ADADELTA methods take up a large part of the total computation time. Based on our experiments, we achieved runtimes on the SX-Aurora TSUBASA VE 20B that are on average 3× faster than on modern dual-socket 64-core CPU nodes. Our solution is competitive with V100 GPUs, even though these already use newer chip fabrication technology (12 nm vs. 16 nm on the VE 20B).**

***Index Terms*—variable execution performance, divergent control structures, molecular docking, AutoDock, vector computing, SX-Aurora TSUBASA**

I. Introduction

Molecular docking is a key method in computer-aided drug design. It aims to predict the close-distance interactions of two molecules, i.e., receptor and ligand, both of known three-dimensional structure. The receptor models a biological target (e.g., a protein or nucleid acid), while the ligand acts as a drug candidate. Typically, libraries containing thousand of ligands are screened in order to find those with anti-viral properties [7]. Molecular docking is used as the computational engine of the so-called *virtual screening* procedure, which selects only promising (and fewer!) ligands for subsequent wet lab experiments.

One of the most-widely used programs in molecular docking is AutoDock [5]. It performs a systematic exploration of the ligand-receptor space to predict the poses (i.e., their spatial geometrical arrangement) that are energetically strong. In algorithmic terms, AutoDock is described using nested loops with variable upper bounds and divergent control performing the molecular search, as well as time-intensive score evaluations typically invoked 10^6 times within these search iterations. In recent years, the official release of AutoDock for GPUs has been under active development. This version, renamed as AutoDock-GPU [2], has been initially implemented in OpenCL, and thereafter ported to CUDA to run on the Summit supercomputer [20]. As an example of its increasing relevance, AutoDock-GPU is nowadays being used as the docking engine in *OpenPandemics: COVID-19*, a grid-computing project aiming to study proteins from the SARS-CoV-2 virus [22].

On the other hand, while the High Performance Computing (HPC) scenario is currently dominated by multi-core CPUs and many-core GPUs, other hardware accelerator technologies exist and are worth exploring. One of these is the recently-introduced SX-Aurora TSUBASA, whose core technologies are vector-based processing and high memory bandwidth (1.53 TB/s). In addition, it offers a programming framework based on standard C/C++, which eases the porting of existing programs. Recent efforts leveraging the SX-Aurora TSUBASA in different fields [1], [9], [19] suggest that this accelerator platform is becoming a competitive alternative for HPC applications.

The most time-consuming part of AutoDock is its local-search phase. The original AutoDock implements the method proposed by Solis-Wets [4] as local-search algorithm, while AutoDock-GPU additionally incorporates the more complex ADADELTA [13] as an alternative algorithm able to predict poses of higher quality. In this paper, we present our experiences porting the irregular computations in AutoDock to the SX-Aurora TSUBASA vector engine. The code developed in this work, named AutoDock-Aurora, has been highly optimized for the SX-Aurora TSUBASA platform with vector-specific coding techniques, i.e., loop pushing, predication, and loop compression. This work discusses the porting of *both* Solis-Wets and ADADELTA methods, and thus extends our prior work in [12] that only supported Solis-Wets on the vector engine. Furthermore, we compare the performance of AutoDock-Aurora against that of AutoDock-GPU running on dual-socket 64-core CPU nodes, as well as on V100 and A100 GPUs.

II. Background

A. SX-Aurora TSUBASA Vector Computer

A system based on the SX-Aurora TSUBASA consists of a Vector Host (VH) and Vector Engine (VE). The VH is an x86 processor responsible for OS-related tasks, e.g., system calls, process scheduling, VE resource management, etc. On the other hand, the VE is a high-performance accelerator in the shape of a full-profile dual-slot PCIe card, and is responsible for the major computations of applications. The VE has eight cores, and each of these cores has a *scalar processing unit* (SPU) attached to a *vector processing unit* (VPU). The SPU employs a RISC instruction set, out-of-order execution, and L1 & L2 caches. Each VPU has 64 long vector registers, each capable of storing a vector of 16,384 bits, as well as several vector execution units. Unlike normal SIMD and SIMT architectures, the *vector execution units* are implemented as 32 $\times$ 64-bit wide SIMD units with 8-cycle deep pipelines. Full-length 16,384 bit vector operations are thus executed as a sequence of steps on the execution units, with each step processing up to 2,048 bits at once. A *vector length register* controls the number of elements processed in vector operations, while 16 *vector mask registers* enable predication.

The first- and second-generation VE processors have each six 8 GB HBM2 modules (i.e., 48 GB RAM in total), and deliver up to 1.53 TB/s of memory bandwidth. The eight cores in the VE are connected to a 16 MB Last Level Cache (LLC) through a fast 2D network-on-chip. Regarding memory accesses, the VE supports *normal* and *partitioned* modes. The normal mode has uniform memory accesses (UMA), i.e., all cores are able to access any part of the LLC and HBM2. The partitioned mode allows non-uniform memory accesses (NUMA), where cores are split into two equally-sized groups, and by default, cores access only their segment of LLC and HBM2. The use of the NUMA mode reduces memory-port and memory-network conflicts, and can bring performance benefits for certain programs.

Main programs can run natively on the VE, with system calls being offloaded to the VH. These programs behave as if running under Linux on the host, although the VE itself runs as "bare metal", without any kind of on-board operating system. As an alternative, Vector Engine Offloading (VEO) [3] is a programming model that executes the main program on the VH and offloads kernels onto the VE. VEO is based on C++ and provides host APIs resembling those of OpenCL, and thus, can be used to express kernel offloading and host-VE data movement. When using VEO, VE code still has to be written in C/C++. VEO is the lowest-level API for accelerator-style programming, and it is the technique used in this work.

B. Molecular Docking

We provide a concise description of AutoDock-GPU's functionality, more details are available in [2], [11].

Molecular docking explores the poses adopted by the ligand, in order to find those poses binding strongly to a certain region on the receptor's surface. For a systematic exploration, AutoDock-GPU maps the docking search into an evolutionary process, where each of the ligand poses is treated as an *individual* of a population. Each individual is represented by its *genotype*, which in turn, is composed of *genes*. AutoDock-GPU encodes a pose using a set of variables that describe the translation (displacement: x, y, z), orientation (rigid-body rotation angles: ϕ, θ, α), and torsion (a set of rotatable-bond angles: ψ_1, ψ_2, $\ldots$, $\psi_{N_{\text{rot}}}$) experienced by the ligand during docking. Each of these variables is considered a gene, being in total N_{genes} ($= N_{\text{rot}} + 6$) per genotype.

AutoDock-GPU executes a Lamarckian Genetic Algorithm (LGA), performing a hybrid search that combines a genetic algorithm (GA) and a local search (LS). The genetic algorithm generates new individuals (encoded as genotypes) through genetic operations such as crossover, mutation, and selection. The local search performs a score minimization procedure aiming to refine the poses produced by the genetic algorithm. The genotypes whose scores were minimized (i.e., improved) via local search, are reintroduced into the overall LGA. As indicated in Algorithm 1, AutoDock-GPU performs several independent LGA runs (default: $N_{\text{LGA}-\text{runs}}^{\text{TOTAL}}$ = 100). Moreover, each of these LGA runs terminates whenever any of their predefined upper bounds for the number of score evaluations (default: $N_{\text{score}-\text{evals}}^{\text{MAX}}$ = 2,500,000) or generations (default: $N_{\text{gens}}^{\text{MAX}}$ = 27,000) is reached.

Algorithm 1: Lamarckian Genetic Algorithm (LGA)

```
1 Function AutoDock-GPU
2     for each LGA-run in N_LGA-runs^TOTAL do
3         while (N_score-evals < N_score-evals^MAX) and (N_gens < N_gens^MAX) do
4             GA (population)
5             LS (population)
```

Both the genetic algorithm and the local search together perform millions of score evaluations in a typical LGA run. Algorithm 2 shows the code structure of the scoring function (SF) employed in AutoDock-GPU. Of the three components, the one executed first is `PoseCalculation`, which transforms the genotypes into atomic coordinates. Such coordinates are employed in the following two components: `InterScore` and `IntraScore`, which compute the ligand-receptor (intermolecular) and ligand-ligand (intramolecular) scores, respectively. The loops' upper-bounds depend on the molecular structure of the input, i.e., the number of elements in the rotation list ($N_{\text{rot}-\text{list}}$), the number of ligand atoms (N_{atom}), and the number of intramolecular contributor-pairs ($N_{\text{intra}-\text{contrib}}$).

Similarly to the original AutoDock program, AutoDock-GPU implements the method of Solis-Wets [4] as local search. Solis-Wets creates new genotypes by adding constrained *random* values (delta) to each of the initial genes composing the genotype (Algorithm 3: `line 5`). The scores of both new and initial genotypes are compared (Algorithm 3: `line 6`). In case of score improvement, then new-genotype-1 becomes the current one. Otherwise, new-genotype-2 is generated by subtracting delta (Algorithm 3: `line 11`) instead of adding it, and

Algorithm 2: Scoring Function (SF)

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

a new score comparison is performed (Algorithm 3: `line 12`). The number of successful and failed attempts is updated depending on the comparison outcome. In addition to its divergent execution, Solis-Wets also has a runtime-defined termination criterion that depends on the maximum number of iterations (default: $N_{\mathrm{LS-iters}}^{\mathrm{MAX}}$ = 300), as well as the minimum step change (default: $\mathrm{step}^{\mathrm{MIN}}$ = 0.01).

Algorithm 3: Solis-Wets (SW) local search

```
1  Function SW (genotype)
2      while (N_LS−iters < N_LS−iters^MAX) and (step > step^MIN) do
3          delta = create-delta (step)
           // new-genotype1
4          for each gene in N_genes do
5              new-gene1 = gene + delta
6          if SF (new-genotype1) < SF (genotype) then
7              genotype = new-genotype1
8              success++; fail = 0
9          else
               // new-genotype2
10             for each gene in N_genes do
11                 new-gene2 = gene - delta
12             if SF (new-genotype2) < SF (genotype) then
13                 genotype = new-genotype2
14                 success++; fail = 0
15             else
16                 success = 0; fail++
17         step = update-step (success, fail)
```

Stepping forward with respect to the original AutoDock, AUTODOCK-GPU has incorporated improved local-search methods beyond Solis-Wets. Algorithm 4 describes one of these, ADADELTA [13], which generates a new-genotype by using the gradients of the current genotype's score (Algorithm 4: `line 4`). Then, if the score of the new-genotype is improved, the latter becomes the current genotype (Algorithm 4: `line 6`). ADADELTA terminates if the number of iterations reaches a maximum (default: $N_{\mathrm{LS-iters}}^{\mathrm{MAX}}$ = 300).

Algorithm 5 describes the gradient calculation (GC) employed by ADADELTA. The code structure resembles that of the scoring function in Algorithm 2. First, the `PoseCalculation` computes the atomic coordinates, which in turn, are used for computing the numerical (`InterGradient`) and analytical (`IntraGradient`) derivatives of the corresponding score components. At this point, such derivatives are expressed as a list of atomic contributions. However, as the overall LGA search works on genotypes, it is required to convert those atom-based into gene-based contributions. This conversion is achieved with `Gtrans`, `Grigidrot`, and `Grotbond` (Algorithm 4: `lines 8-10`), which are loops performing data-dependent operations for computing the translational, orientational, and rotational components of the gradient.

Algorithm 4: ADADELTA (AD) local search

```
1  Function AD (genotype)
2      gradient = GC (genotype)
3      while (N_LS−iters < N_LS−iters^MAX) do
4          new-genotype = update-rule (genotype, gradient)
5          if SF (new-genotype) < SF (genotype) then
6              genotype = new-genotype
7          gradient = GC (genotype)
```

Algorithm 5: Gradient Calculation (GC)

```
1  Function GC (genotype)
       /* Gradients in atomic space */
2      for each rot-item in N_rot−list do
3          PoseCalculation
4      for each lig-atom in N_atom do
5          InterGradient
6      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
```

AutoDock is a compute-bound program, in which the local-search component (Solis-Wets or ADADELTA) requires ∼90 % of the total computation time. Hence, as discussed in Sections IV-B, the main benefits in performance result from optimizing both methods, which in turn, frequently call the compute-intensive scoring function and gradient calculation.

III. RELATED WORK

In recent studies, different applications have been ported onto the SX-Aurora TSUBASA to benchmark its performance. Authors in [9] used standard benchmarks and a tsunami numerical simulation code. Moreover, they introduced a performance model based on *Byte-per-FLOP* (B/F) rates to analyze the bottleneck in the selected applications. In [1], the Himeno benchmark – which solves the Poisson equation via the Jacobi iteration method – is optimized and evaluated on systems with up to eight VEs. The work in [19] refined the aforementioned model introduced in [9] to evaluate the performance of second-generation VEs running various scientific applications.

In addition to the VEO [3] model used here, other programming frameworks for the SX-Aurora TSUBASA have been developed in recent years. The work in [6] proposed an OpenCL-like programming model that allows the usage of OpenCL C/C++ for the host code, and standard C++ for device code. Other related approaches include VEDA (a CUDA-like API on top of VEO) [18], neoSYCL [23], OpenMP

target offloading (integrated with the LLVM compiler) [21], HAM [14], and NEC Hybrid MPI [17].

With regard to parallel implementations of AutoDock, AUTODOCK-GPU has originally been implemented in OpenCL [2], and ported afterwards to CUDA for COVID-19 research on the Summit supercomputer [20] (where OpenCL was not supported). Additionally, AutoDock has been ported to FPGAs as well. For the implementation of the docking engine, the work in [8] used Verilog, while the project called OCLADOCK-FPGA [10] employed OpenCL. As discussed in [11], AUTODOCK-GPU parallelizes the work over multiple *data items* (i.e., genotypes), while OCLADOCK-FPGA executes multiple *tasks* concurrently. In programming terms, each of the kernels in OCLADOCK-FPGA is single threaded. This coding style is intuitively closer to the programming model of the VE, and thus, should allow for easier porting. Regarding performance, OCLADOCK-FPGA on an Arria-10 FPGA runs $\sim$2$\times$ faster than the original AutoDock on a CPU, but it is significantly slower than AUTODOCK-GPU. Therefore, OCLADOCK-FPGA is not being deployed to solve real docking problems.

IV. PARALLELIZATION FOR SX-AURORA TSUBASA

Here, we build upon our prior effort [12], where OCLADOCK-FPGA (incorporating only Solis-Wets) was ported to the SX-Aurora TSUBASA. Since the underlying score and gradient calculations (Section II-B) have a similar code structure, the optimization techniques already applied to Solis-Wets can be extended to vectorize the more complex ADADELTA algorithm as well.

A. Porting

As discussed in [12], we used OCLADOCK-FPGA as our starting point for development. AUTODOCK-AURORA employs the same host and device code partitioning already defined in OCLADOCK-FPGA. In other words, the overall program management is assigned to the host, while the LGA optimization is offloaded onto the VE. In particular, AUTODOCK-AURORA adopts the VEO programming model, so that most of the original host code was kept intact, except for the OpenCL API calls that were replaced with their VEO counterparts. On the other hand, adapting the device code to the VE was a more-involved step. The main reason for that was the fact that the original device code, composed of several single-threaded kernels, heavily used OpenCL-specific inter-kernel communication elements known as *pipes*. When used for FPGAs, OpenCL pipes are mapped onto *on-chip* FIFO-like hardware logic that allows streaming data in/out kernels without resorting to any *off-chip* memory. For the VE port, we removed the pipes and replaced the calls to their corresponding built-in OpenCL functions, such as read_pipe() and write_pipe(), with function calls passing data via pointer arguments.

As ADADELTA was not implemented in OCLADOCK-FPGA, we use its respective implementation from AUTODOCK-GPU as a baseline for vectorization in AUTODOCK-AURORA. As a first step, we adapt the existing SIMT coding style to the VE. For that purpose, we replace the calls to built-in OpenCL functions accessing thread indexes, such as get_global_id() and get_local_id(), with standard C/C++ for loops.

At first glance, the porting effort for both Solis-Wets and ADADELTA to the VE might appear to be trivial. However, the non-determinism (due to randomness) in the LGA heuristics were a major cause of errors. Hence, we spent significant development time verifying the functional correctness of both local-search methods, so that the resulting poses reach the expected level of convergence [2] even after vectorization.

B. Optimization

The first optimization technique we employ is *parallelization*. This is realized by adding #pragma omp parallel for to the outermost loop of the LGA (Algorithm 1). By using this directive (Algorithm 6: `line 2`), we are able to distribute the independent LGA runs among the eight VE cores. Based on our tests using different input cases, faster executions are achieved with a static scheduling and chunk size of one.

Algorithm 6: Parallelized LGA for VE

```
1 Function AutoDock-VE
2     #pragma omp parallel for schedule (static, 1)
3     for each LGA-run in N_LGA-runs^TOTAL do
4         while (N_score-evals < N_score-evals^MAX) and (N_gens < N_gens^MAX) do
5             ...
```

Based on Section II-B, the Solis-Wets method requires a random number generator for creating new genotypes. OCLADOCK-FPGA employs a linear congruential generator, in which any generated random value depends on the previous one: $X_{n+1} = f(X_n)$. As this dependency hinders parallelization/vectorization, we use instead the 64-bit Mersenne Twister pseudorandom generator implemented in the NEC Numeric Library Collection (NLC) [15]. This code change is not required in ADADELTA, as that algorithm creates genotypes utilizing score gradients, rather than from random values.

At this point, the time-consuming `InterScore` and `IntraScore` functions (Algorithm 2) were fully vectorized by the compiler. However, AUTODOCK-AURORA running Solis-Wets was 2.2$\times$ slower compared to the host CPU. The reason for this low performance was due to the vector pipes being leveraged only for the *innermost* loops, which could be quite short. To visualize this, let us consider the score and gradient calculations (Algorithms 2 and 5), both consisting of inner loops with $N_{\text{rot-list}}$, N_{atom}, and $N_{\text{intra-contrib}}$ as upper bounds. Table I lists the loop lengths for some inputs used in our evaluation (Section V-A). While for complex molecules (in terms of N_{rot} and N_{atom}), loop lengths can be relatively large (e.g., `3er5`: $>$ 5,000), for small ones (e.g., `1u4d`, `1yv3`) these lengths are in the order of tens of iterations. Hence, for small molecules, AUTODOCK-AURORA *initially* leveraged vector lengths of just $\frac{1}{10}$th (or even less!) of the maximum vector length of the VE (= 256 elements, each 64-bit wide).

TABLE I
UPPER BOUNDS OF SCORE AND GRADIENT LOOPS FOR SELECTED INPUTS

ID	$N_{\text{rot-list}}$ (PoseCalculation)	N_{atom} (InterScore)	$N_{\text{intra-contrib}}$ (IntraScore)
1u4d	23	23	0
1yv3	31	23	88
3er5	711	108	5,111

From a programming perspective, the under-utilization of vector pipes described above was caused by the initial OpenCL-to-VE porting step. Our initial implementation mapped each OpenCL thread to a *VE core*. However, in order to increase the vector lengths, each OpenCL thread should be mapped instead to a *vector lane*. For that purpose, we apply the technique called *loop pushing* to the main parts of the LGA: the genetic algorithm (GA) and local search (LS). For instance, the outer loop in GA (Algorithm 7: `line 3`) can be *pushed* into the scoring function in such a way that it becomes innermost, data parallel, and easily vectorizable (Algorithm 8: `lines 5,8,11`). This technique is paired with changes in the data layout, so that the vectorized code accesses data with unit-strides as much as possible.

Algorithm 7: Before Loop Pushing: GA and SF

```
1  Function GA (population)
2      ...
3      for each genotype in N_pop-size do
4          Function SF (genotype)
               // Inner loops are detailed in Algorithm 2
5          ...
```

When applied to the GA, loop pushing requires no further adaptations in the GA's code structure, as the scoring function is invoked regularly for *all* $N_{\text{pop-size}}$ genotypes.

Algorithm 8: After Loop Pushing: GA and SF

```
1  Function GA-VE (population)
2      ...
3      Function SF (all genotypes)
4          for each rot-item in N_rot-list do
5              for each genotype in N_pop-size do
6                  PoseCalculation
7          for each lig-atom in N_atom do
8              for each genotype in N_pop-size do
9                  InterScore
10         for each intra-pair in N_intra-contrib do
11             for each genotype in N_pop-size do
12                 IntraScore
```

On the other hand, enabling loop pushing in any of the local-search methods requires *significant* adaptations. The Solis-Wets and ADADELTA algorithms are divergent. This means, that the genetic individuals in the population evolve *differently*, with some reaching convergence earlier than others. Already-converged individuals become non-active ones, and thus, are removed from the computation. To successfully support loop pushing in both methods, we resort to *predication* as well as to *loop compression* for the non-convergent part of the population. This aims to keep the underlying compute-intensive score and gradient calculations working with unit-stride accesses and without additional predication. Algorithm 9 shows the implementation of these two techniques for Solis-Wets. Predication allows, e.g., keeping track of the number of active individuals (Algorithm 9: `line 20`). On the other hand, loop compression is performed e.g., by replacing the original success scalar variable (Algorithm 3: `lines 8, 14, 16`) with the $\text{success}^{\text{compressed}}[\ ]$ array counterpart (Algorithm 9: `lines 8, 13, 23`).

Algorithm 9: Predication and Compression in SW

```
1  Function SW (all genotypes)
2      while N^active_LS-iters > 0 do
           // Building compressed list of active indexes
3          pop^active_size = 0
4          for each j in N_pop-size do
5              if LS^active[j] then
6                  idx^active[pop^active_size] = j
7                  N^compressed_LS-iters[pop^active_size] = N_LS-iters[j]
8                  success^compressed[pop^active_size] = success[j]
9                  pop^active_size++
10                 ...
           // Updating array-based counts
           // Scoring function leverages loop pushing
11         for each jj in pop^active_size do
12             if SF (new-genotype1) < SF (genotype) then
13                 success^compressed[jj]++
14                 ...
           // Predicating on termination condition
15         N^active_LS-iters = pop^active_size
16         for each jj in pop^active_size do
17             if (N^compressed_LS-iters[jj] > N^MAX_LS-iters) or
                (step^compressed[jj] <= step^MIN) then
18                 LS^active[idx^active[jj]] = 0
19                 N^active_LS-iters--
20             j = idx^active[jj]
21             N_LS-iters[j] = N^compressed_LS-iters[jj]
22             step[j] = step^compressed[jj]
23             success[j] = success^compressed[jj]
```

For ADADELTA, as discussed in [11], the scoring function (SF) and gradient calculation (GC) can be grouped together in order to leverage the data locality of the algorithm. Doing so is feasible, as both SF and GC calculate poses identically, and both share the same loop structure for their inter- and intra-molecular components. Thus, we merge both SF and GC into a single function, where `PoseCalculation` is called only once, and the initially-separated loops are replaced with equivalent fused loops (e.g., {`InterScore`, `InterGradient`} and {`IntraScore`, `IntraGradient`}). On top of this code structure, the above optimizations in Solis-Wets (loop pushing, predication, loop compression) are applied analogously in ADADELTA.

Similarly to its predecessors AUTODOCK-GPU and OCLADOCK-FPGA, the floating-point operations in AUTODOCK-

TABLE II
INPUT DATASET USED IN OUR EVALUATION

ID	1u4d	1xoz	1yv3	1owe	1oyt	1ywr	1t46	2bm2	1mzc	1r55
N_{rot}	0	1	2	3	4	5	6	7	8	9
N_{atom}	23	30	23	27	34	38	40	33	38	27

ID	5wlo	1kzk	3s8o	5kao	1hfs	1jyq	2d1o	3drf	4er4	3er5
N_{rot}	10	11	12	15	18	20	23	26	30	31
N_{atom}	46	45	44	44	54	60	44	63	93	108

AURORA are performed in single precision. Such operations can leverage *packed* vector instructions, where each 64-bit vector element (of a vector register) contains two 32-bit float entities. Therefore, on top of loop pushing, we enable *packed vectorization*, and by doing so, the vectors in AUTODOCK-AURORA can have lengths of up to 512 elements, allowing performance to be doubled.

For its memory accesses, AUTODOCK-AURORA runs by default in UMA mode. As an alternative, we tested the NUMA mode to evaluate whether there are performance benefits from the reduced memory network contention and CPU port conflicts. Running in NUMA mode forces the usage of two processes (on a VE) with four cores each. Unfortunately, the benefits of such NUMA-based multi-processing were cancelled-out by the extra overhead. In other words, executions of 2×4 cores with NUMA and 1×8 cores with UMA (default) have practically the same timings. Hence, we opt to run our experiments on the VE using the simpler UMA mode.

V. EVALUATION

A. Experimental Setup

1) Program configuration: for all experiments, we set $N_{\text{LGA}-\text{runs}}^{\text{TOTAL}}$ = 100, and $N_{\text{score}-\text{evals}}^{\text{MAX}}$ = 2,048,000. The maximum number of generations per LGA (= GA + LS) run ($N_{\text{gens}}^{\text{MAX}}$) is set to 99,999, which is larger than the default value of 27,000. The purpose of this choice is to ensure the program terminates only when the number of score evaluations actually reaches the upper bound. Section V-C discusses the performance impact of the chosen population size ($N_{\text{pop}-\text{size}}$). Moreover, in all cases, the *entire* population is subjected to both genetic algorithm and local search.

2) Dataset: for validating the docking functionality, we use the set of 20 ligand-receptor inputs from our prior work [11]. Table II indicates the number of rotatable bonds and atoms for each input. This dataset covers up to 31 rotatable bonds, which is a large range considering that AutoDock supports $N_{\text{rot}}^{\text{MAX}}$ = 32.

3) Hardware: AUTODOCK-AURORA is executed on a second-generation SX-Aurora TSUBASA VE 20B, while AUTODOCK-GPU on V100 and A100 GPUs, as well as on dual-socket 64-core CPU nodes (i.e., a total of 128 CPU cores). Note that the VE still uses 16 nm semiconductor technology and is thus one or more technology generations *behind* the chips compared against. More details are provided in Table III.

B. Performance Profiling

For profiling executions on the VE, we use the PROGINFO and FTRACE utilities [16], which provide a set of performance metrics. The first utility provides program-level information, while the second focuses on functions and user regions.

TABLE III
TECHNICAL CHARACTERISTICS OF THE EMPLOYED ACCELERATOR PLATFORMS: PROCESS SIZE, BASE CLOCK FREQUENCY (FREQ), NUMBER OF CORES (NCORES), FP32 PERFORMANCE (PERF), MEMORY BANDWIDTH (MEMBW). THE CPU PLATFORM HAS TWO SOCKETS

Characteristics	SX-Aurora	GPU		CPU
	VE 20B	***V100***	***A100***	***EPYC 7713***
Vendor	NEC	NVIDIA	NVIDIA	AMD
Process Size [nm]	16	12	7	7
Freq [GHz]	1.60	1.23	0.76	2.00
Ncores	8	5,120	6,912	64 × 2
Perf [TFLOPS]	4.9	14.1	19.5	4.1
MemBW [GB/s]	1,530	897	1,555	204.8 × 2
L1 Cache	32 kB (SPU I$) 32 kB (SPU O$)	128 kB (per SM)	192 kB (per SM)	96 kB (per core)
L2 Cache	256 kB (SPU) 128 kB (VPU)	6 MB (shared)	40 MB (shared)	512 kB (per core)
L3 Cache	16 MB LCC (shared)	-	-	256 MB (shared)

Table IV compares the execution metrics (input: 1hfs) of AUTODOCK-AURORA running Solis-Wets, using the code versions *before* and *after* applying the loop pushing technique. First, the real time represents the wall-clock elapsed time, while the user time accounts for the time spent by all cores on the VE. As described in Section IV-B, the independent LGA runs are distributed among the eight VE cores via the #pragma omp parallel for, which explains why the user time is ∼8× longer than the real time. It can be noted that, due to loop pushing, both real and user times are improved ∼34×, as well as the execution time for vector instructions (i.e., Vector Time) by a factor of ∼eight. In particular, the reduction in the instruction counts (Inst. Count and Vec. Inst. Count) by ∼54× and ∼2.4× can be explained as follows: In both versions (before and after loop pushing), AUTODOCK-AURORA solves the *same* problem, and thus, their FLOP counts are very similar. However, for the optimized version, many of the formerly scalar loops are now vectorized with large vector lengths (> 200 elements), while the formerly shorter vector loops (average length: ∼195 elements) are now executed in longer loops (average length: ∼214 iterations).

TABLE IV
EXECUTION METRICS OF AUTODOCK-AURORA RUNNING SOLIS-WETS (INPUT: 1HFS): BEFORE VS. AFTER APPLYING LOOP PUSHING

Metric	Optimization: loop pushing		Ratio
	Before	***After***	**Bef. / Aft.**
Real Time [sec]	1,382.2	40.0	34.4
User Time [sec]	11,057.3	319.3	34.6
Vector Time [sec]	2,217.6	280.2	7.9
Inst. Count	23,042,367,970,590	427,822,025,876	53.8
Vec. Inst. Count	300,955,750,861	124,250,428,847	2.4
FLOP Count	39,438,249,953,607	40,872,399,685,468	0.965
MOPS	8,348.6	185,805.3	0.045
MFLOPS	3,566.7	128,005.2	0.028
Avg. Vec. Length	195.4	214.0	0.913
V. Op. Ratio [%]	75.3	99.4	0.758
L1 Cache Miss [sec]	164.6	10.4	15.7

Regarding the throughput metrics, the number of overall operations per second (MOPS) and the number of floating-

point operations per second (MFLOPS) are increased by $\sim$22.3$\times$ ($=\frac{185{,}805.3}{8{,}348.6}$) and $\sim$35.9$\times$ ($=\frac{128{,}005.2}{3{,}566.7}$), respectively. We attribute such improvements to the now streamlined vectorized execution, which is visible in the notable improvement of the vector operation ratio (from $\sim$75 % to $\sim$94 %) as well as the average vector length (from $\sim$195 to $\sim$214 elements). Furthermore, as L1 cache misses occur only in scalar code, the significant decrease of their corresponding times (from $\sim$164 s down to $\sim$10 s) indicates that the number of scalar instructions is also reduced in the optimized version.

Table V reports the execution metrics (input: `1hfs`) of our optimized version for comparing Solis-Wets against ADADELTA. Regarding the first time metrics (real, user, vector), it can be noted that choosing ADADELTA results in $\sim$2$\times$ longer executions compared to Solis-Wets. The larger instruction and FLOP counts in ADADELTA are due to its more complex genotype generation: ADADELTA employs gradient-based calculations rather than the simple additions/subtractions in Solis-Wets (Section II). On the other hand, the MOPS and MFLOPS metrics indicate that Solis-Wets achieves higher throughput. However, ADADELTA achieves a $\sim$3.8$\times$ ($=\frac{10.4}{2.7}$) shorter L1 cache miss time, an even higher vector operation ratio (= 99.6 %), and an average vector length that is much closer to optimal (= 255.9 elements). In practice, achieving the optimal average vector length of 256 elements is not possible, due to the execution divergence in both local-search methods. Compared to Solis-Wets, ADADELTA is less irregular and more compute-intense. Consequently, AUTODOCK-AURORA is capable of leveraging larger vectors when running ADADELTA instead of Solis-Wets.

TABLE V
EXECUTION METRICS OF AUTODOCK-AURORA FEATURING LOOP PUSHING (INPUT: `1HFS`): SOLIS-WETS VS. ADADELTA

Metric	Solis-Wets	ADADELTA	Ratio AD / SW
Real Time [sec]	40.0	78.9	1.9
User Time [sec]	319.3	630.3	1.9
Vector Time [sec]	280.2	598.4	2.1
Inst. Count	427,822,025,876	529,638,863,570	1.2
Vec. Inst. Count	124,250,428,847	184,324,447,676	1.4
FLOP Count	40,872,399,685,468	74,093,567,293,801	1.813
MOPS	185,805.3	164,615.5	0.886
MFLOPS	128,005.2	117,545.9	0.918
Avg. Vec. Length	214.0	255.9	1.196
V. Op. Ratio [%]	99.4	99.6	1.002
L1 Cache Miss [sec]	10.4	2.7	0.3

C. Comparison against GPUs and CPUs

We compare the performance of AUTODOCK-AURORA against that of AUTODOCK-GPU, the state-of-the-art OpenCL-based implementation of AutoDock for GPUs/CPUs. For a fair comparison, we use the version v1.1 of AUTODOCK-GPU, in order to ensure that equivalent functionalities are run on the VE 20B and the chosen GPUs/CPU. Furthermore, we disregard the different host platforms in the systems employed for evaluation. Specifically, in our time measurements, we account only for the time spent on the LGA computation (i.e., docking runtime). This means that, for the VE 20B and GPUs, our measurements include the device-side kernel configuration and execution, plus all required host-device data movements. Host-side-only operations, such as file I/O and results processing, are not included.

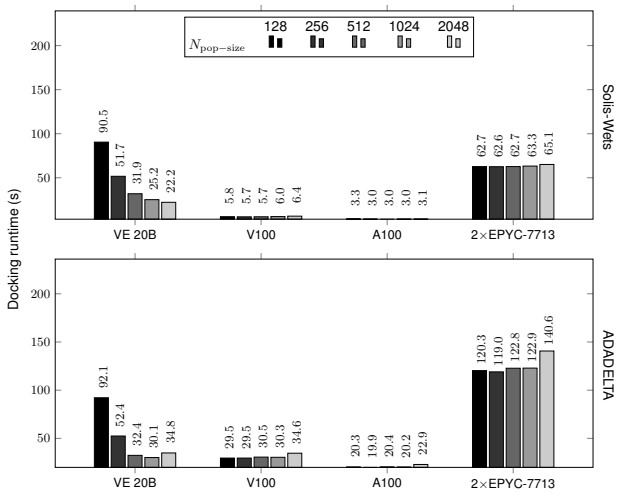

Fig. 1. Geometric mean of docking runtimes over 20 inputs, comparing the impact of different population sizes ($N_{\text{pop-size}}$).

As explained in Section IV-B, the already-converged individuals are removed from the local-search computation. Thus, the length of the innermost loops is reduced, and in turn, the performance on the VE 20B is diminished. A way to cope with that is to employ larger population sizes, which increase the vector length of *pushed-in* loops in AUTODOCK-AURORA. As shown in Figure 1, larger populations generally result in faster executions on the VE 20B. For instance, by increasing $N_{\text{pop-size}}$ from 128 to 2048, the corresponding runtimes reduce $\sim$4.1$\times$ (Solis-Wets: $\frac{90.5}{22.2}$) and $\sim$2.6$\times$ (ADADELTA: $\frac{92.1}{34.8}$). Conversely, the population sizes have less impact on any of the other devices. In fact, with respect to the VE 20B, the runtimes achieved on the GPUs/CPU – for each $N_{\text{pop-size}}$ configuration – have lower standard deviations: {28.3, 0.3, 0.1, 1.1} (Solis-Wets) and {26.0, 2.1, 1.2, 8.8} (ADADELTA) for {VE 20B, V100, A100, EPYC 7713}, respectively. In particular, the maximal runtime increase due to larger populations configured for the GPUs/CPU is $\sim$18% ($=\frac{140.6-119.0}{119.0}\times 100\%$), and occurs on the EPYC 7713 when running ADADELTA.

The performance behavior on the GPUs/CPU can be explained as follows. In AUTODOCK-GPU, the population size directly affects the workload distribution (based on the number of spawned OpenCL work-groups: $N_{\text{WG}} = N_{\text{pop-size}} \times N^{\text{TOTAL}}_{\text{LGA-runs}}$), but has no impact on the runtime of a score evaluation. Since the LGA executes until it has evaluated a given number of scores (Section V-A: $N^{\text{MAX}}_{\text{score-evals}}$ = 2,048,000), processing larger populations requires *fewer* iterations per LGA-run, as more scores are evaluated *simultaneously*. In other words, the seemingly bigger workload imposed by the need to process *more* individuals, is compensated by the *reduced number* of iterations per LGA-run. In addition,

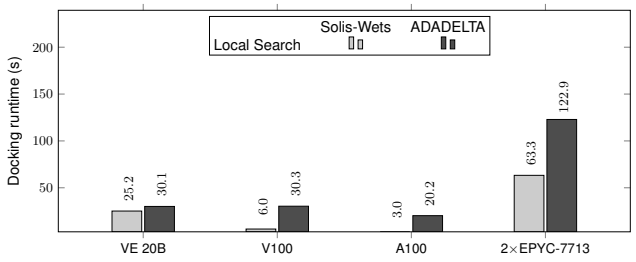

Fig. 2. Geometric mean of docking runtimes over 20 inputs, when $N_{\mathrm{pop-size}}$ = 1024, comparing overall performance per platform.

we attribute the relatively slight increase of the GPUs/CPU runtimes when processing larger populations to the synchronization overhead associated to the larger OpenCL work-group sizes. In general, as already reported in our previous work on GPUs/CPUs [2], ADADELTA executions take longer than their Solis-Wets' counterparts, regardless of the population size.

Figure 2 shows the runtimes for $N_{\mathrm{pop-size}}$ = 1024 only. Since population sizes have no significant impact on the GPUs/CPU runtimes, then we see no disadvantage in using an efficient configuration for the VE 20B in the following comparison between devices. First, the VE 20B clearly outperforms the EPYC 7713, being the former $\sim$2.5$\times$ (Solis-Wets: $\frac{63.3}{25.2}$) and $\sim$4.1$\times$ (ADADELTA: $\frac{122.9}{30.1}$) faster than the dual-socket 64-core CPU. Despite being slower by $\sim$8.4$\times$ (Solis-Wets: $\frac{25.2}{3.0}$) and $\sim$1.5$\times$ (ADADELTA: $\frac{30.1}{20.2}$) compared to the A100, the VE 20B is *only* $\sim$4.2$\times$ (= $\frac{25.2}{6.0}$) slower than the V100 when running Solis-Wets, while still achieving a slightly *faster* average execution (30.1 s vs. 30.3 s) than this latter GPU, both running ADADELTA. The reason for this performance behavior on the VE 20B is attributed to the more compute-intensive calculations of ADADELTA, which with larger populations, fill up the vector lanes more efficiently than Solis-Wets. For putting these numbers into perspective, note that the VE 20B in some cases outperforms devices that are at least one (V100) or multiple (EPYC) silicon generations ahead of it.

VI. Conclusions

In this work, we have ported the AutoDock molecular docking program to the SX-Aurora TSUBASA VE 20B. The most time-consuming component of AutoDock is the local search, which presents a high degree of irregularity. Here, both Solis-Wets and ADADELTA local-search methods have been optimized for higher performance. The most significant performance improvements on the VE 20B are achieved by applying a platform-specific coding style based on loop pushing. This technique, paired with predication and loop compression, results in time reductions of $\sim$34$\times$ (Solis-Wets) with respect to an unoptimized version. Furthermore, it can be observed that executions on the VE 20B are faster when employing larger populations in the genetic search. In terms of runtime-based performance, for a configuration of 1024 individuals per population, our implementation running ADADELTA on the VE 20B achieves a slightly faster average execution compared to that on the V100 GPU, and $\sim$2.5$\times$ (Solis-Wets) and $\sim$4.1$\times$ (ADADELTA) faster than the dual-socket 64-core EPYC 7713 CPU. Vectorized execution can thus be competitive with the SIMT approach used on GPUs. It would be very interesting to see the impact of our techniques when applied to a VE implemented in more current 7 nm semiconductor technology.

References

[1] A. Onodera et al., "Optimization of the Himeno Benchmark for SX-Aurora TSUBASA," in *Proc. Benchmarking, Measuring, and Optimizing*. Springer, 2021, pp. 127–143.

[2] D. Santos-Martins et al., "Accelerating AutoDock4 with GPUs and Gradient-Based Local Search," *J. Chem. Theory Comput.*, vol. 17, no. 2, pp. 1060–1073, 2021.

[3] E. Focht, "VEO and PyVEO: Vector Engine Offloading for the NEC SX-Aurora Tsubasa," in *Proc. Sustained Simulation Performance 2018 and 2019*. Springer, 2020, pp. 95–109.

[4] F. J. Solis et al., "Minimization by Random Search Techniques," *Math. Oper. Res.*, vol. 6, no. 1, pp. 19–30, 1981.

[5] G. Morris et al., "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function," *J. Comput. Chem.*, vol. 19, no. 14, pp. 1639–1662, 1998.

[6] H. Takizawa et al., "An OpenCL-Like Offload Programming Framework for SX-Aurora TSUBASA," in *Proc. 20th PDCAT*. ACM, 2019, pp. 282–288.

[7] I. Halperin et al., "Principles of docking: An overview of search algorithms and a guide to scoring functions," *Proteins: Struct., Funct., Bioinf.*, vol. 47, no. 4, pp. 409–443, 2002.

[8] I. Pechan et al., "FPGA-based acceleration of the AutoDock molecular docking software," in *Proc. 6th Ph.D. Research in Microelectronics & Electronics*. IEEE, 2010, pp. 1–4.

[9] K. Komatsu et al., "Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA," in *Proc. SC18*. IEEE, 2018, pp. 685–696.

[10] L. Solis-Vasquez et al., "A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software," in *Proc. 5th FSP*. VDE Verlag, 2018, pp. 1–10.

[11] ——, "Parallelizing Irregular Computations for Molecular Docking," in *Proc. 10th IA^3*. IEEE, 2020, pp. 12–21.

[12] ——, "Porting and Optimizing Molecular Docking onto the SX-Aurora TSUBASA Vector Computer," *Supercomput. Front. Innov.*, vol. 8, no. 2, pp. 27–42, 2021.

[13] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," *arXiv*, vol. abs/1212.5701, 2012.

[14] M. Noack et al., "Heterogeneous Active Messages for Offloading on the NEC SX-Aurora TSUBASA," in *Proc. IPDPSW*. IEEE, 2019, pp. 26–35.

[15] NEC, "Numeric Library Collection 2.3.0 User's Guide," 2018. [Online]. Available: https://www.hpc.nec/documents/sdk/SDK_NLC/UsersGuide/main/en/index.html

[16] ——, "PROGINF/FTRACE User Guide," 2018. [Online]. Available: https://www.hpc.nec/documents/sdk/pdfs/g2at03e-PROGINF_FTRACE_User_Guide_en.pdf

[17] ——, "Hybrid MPI," 2020. [Online]. Available: https://www.nec.com/en/global/solutions/hpc/articles/tech05.html

[18] ——, "VEDA GitHub repository," 2021. [Online]. Available: https://github.com/SX-Aurora/veda

[19] R. Egawa et al., "Exploiting the Potentials of the Second Generation SX-Aurora TSUBASA," in *Proc. 11th PMBS*. IEEE, 2020, pp. 39–49.

[20] S. LeGrand et al., "GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research," in *Proc. 11th BCB*. ACM, 2020.

[21] T. Cramer et al., "OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine," in *Proc. 13th PPAM*. Springer, 2019, pp. 237–249.

[22] World Community Grid, "OpenPandemics - COVID-19 Now Running on Machines with Graphics Processing Units," 2021. [Online]. Available: https://www.worldcommunitygrid.org/about_us/viewNewsArticle.do?articleId=693

[23] Y. Ke et al., "NeoSYCL: A SYCL implementation for SX-Aurora TSUBASA," in *Proc. HPC Asia*. ACM, 2021, pp. 50–57.

A. Artifact Appendix

A.1 Abstract

This artifact appendix provides instructions on how to retrieve, compile, and evaluate the developed AUTODOCK-AURORA code. This includes instructions on how to obtain the input data sets, as well as scripts to regenerate the *docking runtimes* achieved on the SX-Aurora TSUBASA Vector Engine (VE), which are discussed in this paper. Furthermore, these instructions allow benchmarking AUTODOCK-AURORA against AUTODOCK-GPU, a state-of-the-art OpenCL implementation for GPUs and multi-core CPUs.

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

- **Algorithm:** Molecular docking based on a Lamarckian Genetic Algorithm, which combines a genetic algorithm and a local-search method.
- **Program:** Parallelized versions of the AutoDock molecular docking software. AUTODOCK-AURORA is C/C++ vectorized version for the SX-Aurora TSUBASA Vector Engine. AUTODOCK-GPU is an OpenCL implementation for GPUs and multi-core CPUs. All sources can be downloaded from GitHub. Sizes: ∼400 MB (AUTODOCK-AURORA), ∼120 MB (AUTODOCK-GPU).
- **Compilation:** We used `g++` 4.8.5 for AUTODOCK-AURORA, and `g++` 6 (and above) for AUTODOCK-GPU.
- **Binary:** Source code and scripts included to generate binaries.
- **Data set:** Molecular structures prepared for any tool of the AutoDock suite. All files can be downloaded from GitLab, and are ready to use. Size: ∼200 MB.
- **Run-time environment:** AUTODOCK-AURORA requires a Linux distribution with NEC C++ compilers (We used Red Hat 4.8.5-44 with the `nc++` (NCC) v3.2.1). AUTODOCK-GPU requires a Linux distribution supporting OpenCL drivers (We used Ubuntu 20.04.3 LTS with NVIDIA CUDA-11 and Intel 2019 drivers). No need of root access.
- **Hardware:** We used NEC SX-Aurora TSUBASA VE 20B, as well as NVIDIA PCIe GPUs (V100 and A100) and an AMD CPU server (two sockets x 64-core EPYC 7713).
- **Execution:** Sole user. AUTODOCK-AURORA's benchmark on the NEC VE 20B takes ∼5 hours. AUTODOCK-GPU's benchmarks on both NVIDIA GPUs & AMD CPU take ∼14 hours.
- **Metrics:** Execution runtimes in seconds.
- **Output:** Console indicating execution runtime of a given experiment. Additionally, *docking log files* (.dlg) reporting execution runtime and resulting molecular pose predictions.
- **Experiments:** Bash scripts (provided). Maximum allowable variation of execution runtimes: 10%.
- **How much disk space required (approximately)?:** Maximum: 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** Two hours.
- **How much time is needed to complete experiments (approximately)?:** 20 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Lesser GNU Lesser General Public License.
- **Data licenses (if publicly available)?:** Creative Commons Attribution 4.0 International.
- **Archived (provide DOI)?:** No.

A.3 Description

A.3.1 How to access

All material is publicly available. The following is the *main* repository providing a single-entry point for reproducibility instructions:

- Reproducibility artifacts (instructions and scripts)
  `https://github.com/L30nardoSV/reproduce-ia3-2021-moldocking-vector`

The following repositories are listed here *only* for completeness. Their contents are automatically downloaded following the instructions provided in the above main repository.

- AUTODOCK-AURORA (source code)
  `https://github.com/esa-tu-darmstadt/AutoDock-Aurora`
- AUTODOCK-GPU (source code)
  `https://github.com/ccsb-scripps/AutoDock-GPU`
- Data sets
  `https://gitlab.com/L30nardoSV/ad-gpu_miniset_20`

A.3.2 Hardware dependencies

For obtaining comparable results, we recommend using the following devices as hardware accelerators:

- Vector Engine: NEC SX-Aurora TSUBASA VE 20B
- GPUs: NVIDIA PCIe-based V100 and A100
- CPU: AMD EPYC 7713 (two sockets with 64 cores each, i.e., 128 cores in total)

A.3.3 Software dependencies

NEC and NVIDIA/Intel (OpenCL) compilers/drivers are required for the above recommended hardware. Installation instructions are provided by the respective vendor.

A.4 Installation

The installation required for all experiments is described in the README.md of the aforementioned main repository, where *separate* instruction guidelines for AUTODOCK-AURORA and AUTODOCK-GPU are provided.

A.5 Experiment workflow

The first step of the instruction guidelines is to prepare input files and program binaries. Once this is complete, the second step is to run the benchmarks. The experiment workflow has been automated with bash scripts as much as possible. More details can be found in the following scripts (in the main repository): evaluate_autodock_aurora.sh and evaluate_autodock_gpu.sh.

A.6 Evaluation and expected result

Resulting docking log files (.dlg) will be stored under the following folders: results_aurora for AUTODOCK-AURORA and results_<LABEL> for AUTODOCK-GPU. Each of these docking log files will contain a set of predicted molecular poses as well as execution runtime information. The execution runtime information is listed as *docking run time* and *program run time*. Only the docking run time is used for the discussion in the paper. In addition, for streamlining the extraction of docking run times, the main repository provides a collect_results_dlg.py python script and corresponding usage guidelines.

A.7 Experiment customization

Customizing the docking configuration used in the experiments is possible by modifying the parameters in the following script (in the main repository): list_inputs.sh. For more details, please refer to the README.md in the corresponding source code repository of AUTODOCK-AURORA and AUTODOCK-GPU.

A.8 Methodology

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