DOI: 10.14529/jsfi210202

Porting and Optimizing Molecular Docking onto the SX-Aurora TSUBASA Vector Computer

Leonardo Solis-Vasquez[1], *Erich Focht*[2], *Andreas Koch*[1]

In computer-aided drug design, the rapid identification of drugs is critical for combating diseases. A key method in this field is molecular docking, which aims to predict the interactions between two molecules. Molecular docking involves long simulations running compute-intensive algorithms, and thus, can profit a lot from hardware-based acceleration. In this work, we investigate the performance efficiency of the SX-Aurora TSUBASA vector computer for such simulations. Specifically, we present our methodology for porting and optimizing AutoDock, a widely-used molecular docking program. Using a number of platform-specific code optimizations, we achieved executions on the SX-Aurora TSUBASA that are in average 3.6× faster than on modern 128-core CPU servers, and up to a certain extent, competitive to V100 and A100 GPUs. To the best of our knowledge, this is the *first* molecular docking implementation for the SX-Aurora TSUBASA.

Keywords: application porting, performance optimization, molecular docking, AutoDock, vector computing, SX-Aurora.

Introduction

In recent years, the NEC SX-Aurora TSUBASA computer system has been introduced to the High Performance Computing (HPC) landscape. Besides its core technologies, i.e., vector-based processing and high memory bandwidth (1.53 TB/s), the SX-Aurora TSUBASA offers a programming framework based on standard C/C++, which eases the porting of existing programs. Simulations in computational dynamics, electromagnetism, and other fields have been accelerated on this platform [6, 13, 21], and thus, the SX-Aurora TSUBASA has become an alternative accelerator platform in HPC.

The applicability of the SX-Aurora TSUBASA in other scientific areas is yet to be investigated. An example is the field of computer-aided drug design, which leverages compute-intensive molecular docking simulations. Basically, molecular docking predicts close-distance interactions of two molecules: the receptor and the ligand, both of known three-dimensional structure. A receptor models a biological target, while a ligand acts as a drug candidate. Identifying new ligands can be done by screening large databases of small molecules, aiming to find those that interact favorably with a given receptor [9]. This process, called virtual screening, typically requires thousands of molecular docking executions. However, it enables using only promising (and fewer!) ligands in the subsequent costly and slow wet lab experiments. Software tools for molecular docking have become relevant at combating diseases. One of these is the widely-used AutoDock, which has been used as the docking engine in world-wide community grid projects such as *Fight-AIDS@Home* [1] as well as *OpenPandemics: COVID-19* [3]. In algorithmic terms, AutoDock explores several spatial geometrical arrangements between a receptor and a ligand (i.e., poses) using nested loops and divergent control flows. Moreover, AutoDock computes a score for each pose, and in turn, performs millions of score evaluations per execution.

In this paper, we present our methodology for efficiently porting and optimizing AutoDock onto the SX-Aurora TSUBASA. The code developed in this work, termed AutoDock-Aurora, has been ported from an existing OpenCL version of AutoDock. For achieving higher performance,

[1]Technical University of Darmstadt, Darmstadt, Germany
[2]NEC Deutschland GmbH, Stuttgart, Germany

platform-specific coding styles (e.g., loop pushing, data compression, predication) as well as optimization practices (e.g., those based on compiler technologies) were applied. With AutoDock-Aurora, we aim to expand the applicability of the SX-Aurora TSUBASA to solving a wider range of scientific problems. The organization of this paper is as follows. Section 1 provides background information on the target platform and application under analysis. Section 2 describes our porting and optimization methodology. Section 3 discusses the results of evaluating AutoDock-Aurora on the SX-Aurora TSUBASA as well as on modern high-end GPUs and CPUs. Section 4 reviews the related work. Finally, our conclusions and future work are presented.

1. Background

1.1. SX-Aurora Vector Engine

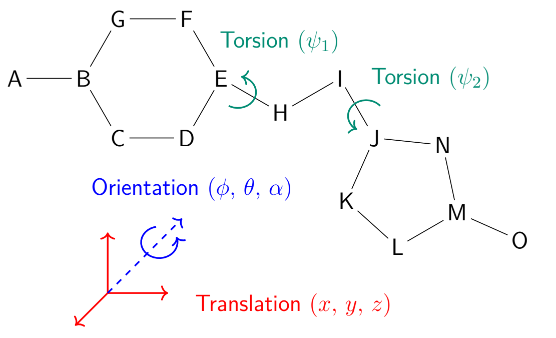
The SX-Aurora TSUBASA Vector Engine (VE) is an accelerator in the shape of a full profile dual-slot PCI card. The VE1 and VE2 generation processors have 6 $\times$ 8 GB HBM2 stacks for a total of 48 GB RAM with up to 1.53 TB/s memory bandwidth. The regular VEs that we used have only eight cores connected to a 16 MB Last Level Cache (LLC) through a fast 2D network-on-chip. Each core consists of a scalar processing unit with RISC instruction set, out-of-order execution, L1 and L2 caches, that is attached to a vector processing unit with 64 long vector registers of 256 $\times$ 64 bit words and several vector execution units. Unlike normal SIMD or SIMT architectures, the vector units are implemented as a combination of 32 $\times$ 64 bit wide SIMD units with 8-cycle deep pipelines. A vector length register controls the number of elements processed in vector operations, and 16 vector mask registers enable predication.

The VE's normal mode has uniform memory access (UMA), with all cores being able to access and use any part of the LLC and HBM2 memory. It can be reconfigured to *partitioned* mode with non-uniform memory access (NUMA), where cores are split into two equally sized groups, and by default access only their segment of LLC and HBM2 memory. NUMA mode reduces memory port and memory network conflicts, and can bring performance benefits for certain classes of programs.

VEs need to run inside a normal Linux system usually called Vector Host (VH). The VH runs the VE Operating System (VEOS) which manages VE resources, schedules processes, and manages physical and virtual memory of VEs. Programs can run natively on the VEs, with system calls offloaded to the VH and processed on behalf of their user. They 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. Native VE programs can call functions that run on the VH, in a pattern called reverse offloading.

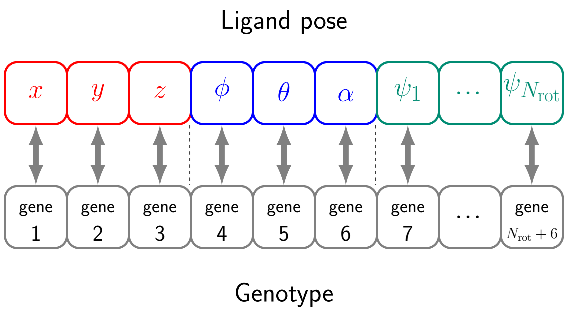
Vector Engine Offloading (VEO) [8] is a programming model which executes the main program on the VH and offloads kernels to the VEs. While the API somewhat resembles OpenCL, it differs from it due to the SIMD/vector nature of kernels, and due to their ability to execute almost any Linux system call. VEO is the lowest-level API for accelerator style programming and is the technique used for this project.

Alternative hybrid programming approaches include VEDA [4], which implements a CUDA-alike device API on top of VEO, neoSYCL [12], OpenMP target offloading [5] integrated with the LLVM compiler, HAM [20], and NEC Hybrid MPI.



G
F
A
B
E
C
D
Torsion (ψ_1)
H
I
Torsion (ψ_2)
J
N
K
M
L
O
Orientation (ϕ, θ, α)
Translation (x, y, z)

(a)



(b)

Figure 1. (a) Degrees of freedom of a theoretical ligand composed of atoms A, B, C, . . . , O. Bonds between atoms are depicted as connecting lines. Each rotatable bond such as E–H and I–J corresponds to a torsion, i.e., rotation of affected ligand atoms around the rotatable-bond axis. (b) Mapping between a ligand pose (a set of degrees of freedom) and a genotype (set of genes). The number of rotatable bonds in a ligand is denoted as N_{rot}

1.2. Molecular Docking

Molecular docking consists of solving an optimization problem that explores the poses adopted by a ligand with respect to a receptor. It is based on the *lock and key* concept by Fischer [7], in which a *perfect* binding between a ligand and receptor occurs when they have *exactly complementary* geometric shapes. The two main aims of molecular docking are: first, to predict the ligand poses within a certain binding site of a receptor; and second, to estimate the affinity of their corresponding interactions. As shown in Fig. 1a, a ligand pose can be represented by the degrees of freedom (i.e., translation, rotations, and torsions) experienced during interaction. As such representation typically involves many degrees of freedom, the docking optimization problem suffers from a combinatorial explosion. To cope with that, molecular docking performs a systematic exploration via heuristic *search methods* (e.g., genetic algorithms, simulated annealing, etc.), which are assisted by *scoring functions* that estimate the binding affinity. Extensive discussions on the categories of search methods and scoring functions are available in [15, 29].

Particularly, AutoDock [16] is one of the most-cited software tools in molecular docking. It is implemented in C++, and provided as open source. The main computation engine in AutoDock is a Lamarckian Genetic Algorithm (LGA), which hybridizes two methods: a Genetic Algorithm and a Local Search.

A Genetic Algorithm (GA) is inspired by the Darwinian evolution theory, and hence, it maps the docking search into a biological evolution process. In this context, each of the ligand's degrees of freedom corresponds to a *gene*. A ligand pose, composed of the entire set of degrees of freedom, corresponds to an *individual*, which in turn is represented by its *genotype* (i.e., full set of genes) as shown in Fig. 1b. Individuals experience genetic modifications such as *crossover* and *mutation*. Moreover, the population of individuals undergoes a *selection* procedure that chooses the *stronger* ones for the next generation. An individual's strength is quantified with its score, which is evaluated with a scoring function.

In the context of molecular docking, a score enhancement implies a *minimization* of its value. In other words, the lower the scores, the stronger the ligand-receptor interactions. In AutoDock, the Local Search (LS) aims to improve the scores of the poses already generated via the GA. For that purpose, AutoDock subjects a population subset of randomly-chosen individuals ($\mathrm{LS}_{\mathrm{rate}}$, default: 6 %) to the method of Solis-Wets [24]. Basically, this is an adaptive-iterative method that takes a genotype as input, and generates a new one by adding small changes (constrained random amount) to each of the input genes. Then, the scores of these two genotypes are computed and compared. If the score is not minimized, a second genotype is generated by subtracting (instead of adding) small changes to the input genes. Afterwards, a second score comparison is performed. The termination criterion is adapted at runtime according to the number of successful or failed attempts at minimizing the score. In each generation, the poses which could actually be improved by the LS are then re-introduced into the LGA population.

Furthermore, AutoDock uses the scoring function (SF) in Eq. 1, which computes the binding affinity as a semi-empirical free-energy force field (kcal/mol) [10]. The first terms involve the summation of four interaction types (Van der Waals, hydrogen bonding, electrostatics, and desolvation) over all the ligand and receptor atoms. The fifth term represents the (unfavorable) loss of ligand entropy upon binding due to the N_{rot} rotatable bonds. All terms are characterized by constant weights (W_{vdw}, W_{hb}, W_{el}, W_{ds}, W_{rot}) and look-up tables (A_{ij}, B_{ij}, C_{ij}, D_{ij}, S, V), as well as by other parameters. Most importantly, the score is determined by the interatomic distance r_{ij}. The value of r_{ij} is calculated at runtime from the atomic coordinates of atoms i and j, and depends entirely on a genotype (generated either via GA or LS), that in turn encodes a respective ligand movement.

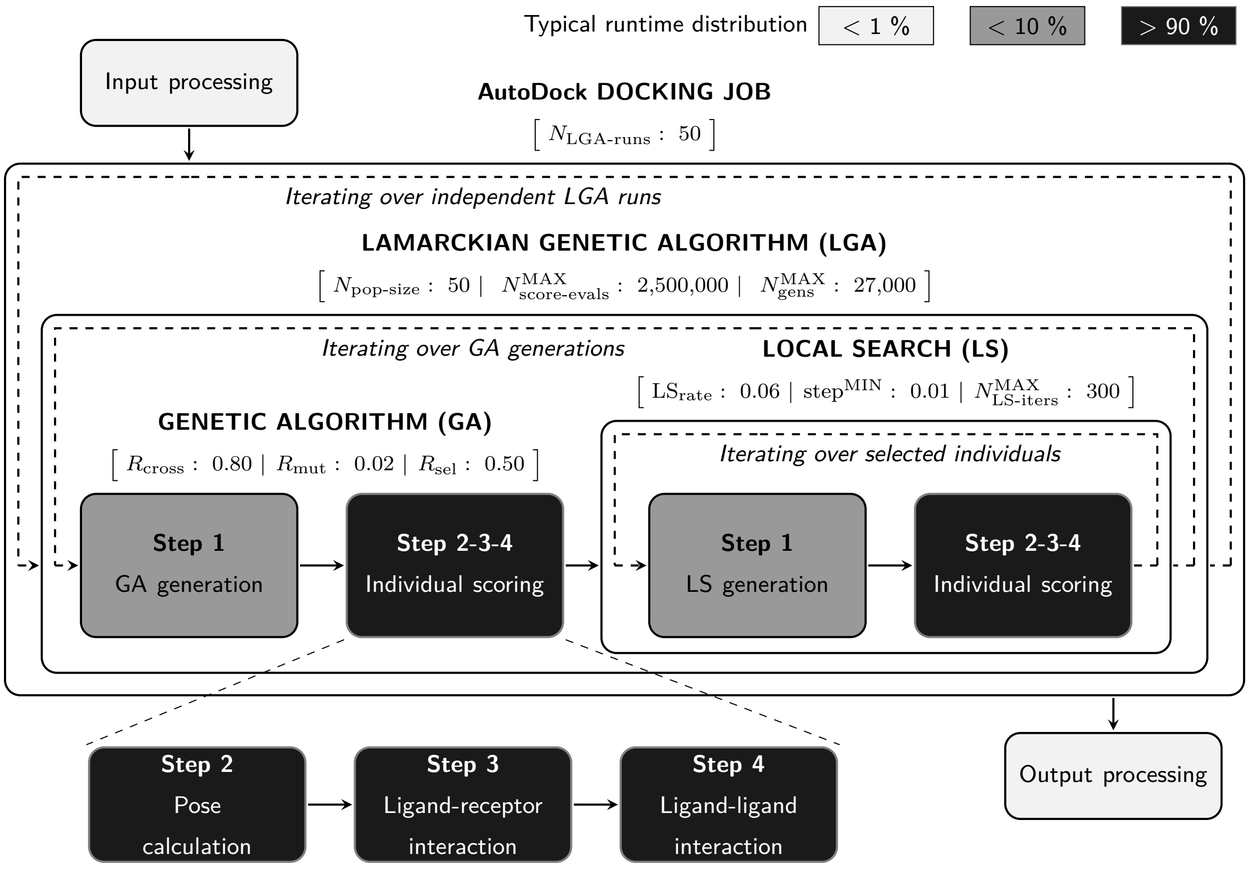
$$\begin{aligned}\text{SF} = \sum_{i,j} \Bigg[& W_{\text{vdw}} \Big(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \Big) + W_{\text{hb}}\, E(t) \Big(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \Big) + \\ & W_{\text{el}} \Big(\frac{q_i q_j}{\epsilon(r_{ij}) r_{ij}} \Big) + W_{\text{ds}} \Big(S_i V_j + S_j V_i \Big) e^{\frac{-r_{ij}^2}{2\sigma^2}} \Bigg] + W_{\text{rot}}\, N_{\text{rot}} \end{aligned} \tag{1}$$

The block diagram in Fig. 2 depicts the functionality of AutoDock, along with the default values of LGA parameters. The generation of genotypes via GA is parameterized with the ratios of crossover (R_{cross}), mutation (R_{mut}), and selection (R_{sel}), while the termination of LS is controlled by the maximum number of iterations ($N_{\text{LS-iters}}^{\text{MAX}}$), as well as by the minimum change step (step^{MIN}). A single LGA-run optimizes the scores of a population (of $N_{\text{pop-size}}$ individuals) until reaching the maximum number of score evaluations ($N_{\text{score-evals}}^{\text{MAX}}$) or generations ($N_{\text{gens}}^{\text{MAX}}$), whichever comes first. An AutoDock docking job consists of the execution of several LGA runs, typically $N_{\text{LGA-runs}} = 50$, which are completely independent from each other. Furthermore, evaluating the score of an individual involves three steps. First, the generated pose (expressed as genotype) is transformed into its corresponding atomic coordinates. Then, intermolecular (ligand-receptor) and intramolecular (ligand-ligand) interactions are calculated using Eq. 1. Note that the interactions between receptor atoms are not calculated, as this molecule is treated as rigid [27].

2. Methodology

2.1. Porting

As reported in Section 4, AutoDock has been ported to other accelerators such as GPUs and FPGAs. In this work, we used ocladock-fpga [27] (OpenCL implementation for FPGAs) as the starting point of our development for SX-Aurora. Compared to AutoDock-GPU [23] (OpenCL implementation for GPUs/CPUs), ocladock-fpga is intuitively close to the programming model of the VE and thus should allow for easier code porting.



Typical runtime distribution
< 1 %
< 10 %
> 90 %
Input processing
AutoDock DOCKING JOB
[$N_{\text{LGA-runs}}$: 50]
Iterating over independent LGA runs
LAMARCKIAN GENETIC ALGORITHM (LGA)
[$N_{\text{pop-size}}$: 50 | $N^{\text{MAX}}_{\text{score-evals}}$: 2,500,000 | $N^{\text{MAX}}_{\text{gens}}$: 27,000]
Iterating over GA generations
LOCAL SEARCH (LS)
[LS_{rate} : 0.06 | step^{MIN} : 0.01 | $N^{\text{MAX}}_{\text{LS-iters}}$: 300]
GENETIC ALGORITHM (GA)
[R_{cross} : 0.80 | R_{mut} : 0.02 | R_{sel} : 0.50]
Iterating over selected individuals
Step 1 GA generation
Step 2-3-4 Individual scoring
Step 1 LS generation
Step 2-3-4 Individual scoring
Step 2 Pose calculation
Step 3 Ligand-receptor interaction
Step 4 Ligand-ligand interaction
Output processing

Figure 2. AutoDock block diagram [26] with default values of LGA parameters

For instance, like any other OpenCL/CUDA program for GPUs, AutoDock-GPU follows a SIMT programming style, where data to be processed is accessed through a grid of threads indexes, e.g., those obtained via the OpenCL `get_global_id()/get_local_id()` built-in functions. In contrast, each of the component kernels of ocladock-fpga was coded as a *single-threaded task*. This implementation approach keeps most of the loop structures shown in Fig. 2 intact, and thus, allows porting such loops with only minor effort, as well as allowing the use of vectorization for loop-level acceleration.

In AutoDock-Aurora, we use the same host and device code partitioning already defined in ocladock-fpga. Thus, we adopt the VEO programming model, where the overall program management is assigned to the host, and the independent LGA runs are offloaded onto the VE. Regarding the host code, we kept most of it intact, except for the calls to OpenCL APIs that we replaced with their VEO counterparts. For adapting the device code, we removed all language-specific qualifiers, so that OpenCL kernels were transformed into plain standard C++ functions. In particular, the baseline code in ocladock-fpga uses OpenCL pipes (*on-chip* FIFO-like structures) to pass data between kernels without resorting to any external memory. Hence, we removed the calls to OpenCL `read_pipe()/write_pipe()`, and replaced them with required function calls. While the porting just described might appear to require a little effort, it was not a trivial task. In fact, the non-determinism (due to randomness) in the GA heuristics was a major cause of masked errors. Consequently, we had to spend significant development times verifying AutoDock-Aurora's functionality, so that the resulting ligand poses and scores actually reach the expected level of convergence.

In an initial optimization pass, we followed the compiler hints, as well as the NEC performance tuning guidelines [18]. Examples of code optimizations applied here include the removal of data dependencies, and the usage of more suitable data types (e.g., four-byte `int` instead of single-byte `char`) for index and loop-control variables. As a result, we achieved a full vectorization of the time-consuming functions computing the ligand-receptor and ligand-ligand scores (Fig. 2). Furthermore, by adding the directive `#pragma omp parallel for` to the outermost loop of the device code, we were able to parallelize the independent LGA runs, and hence, to distribute them among the eight VE cores.

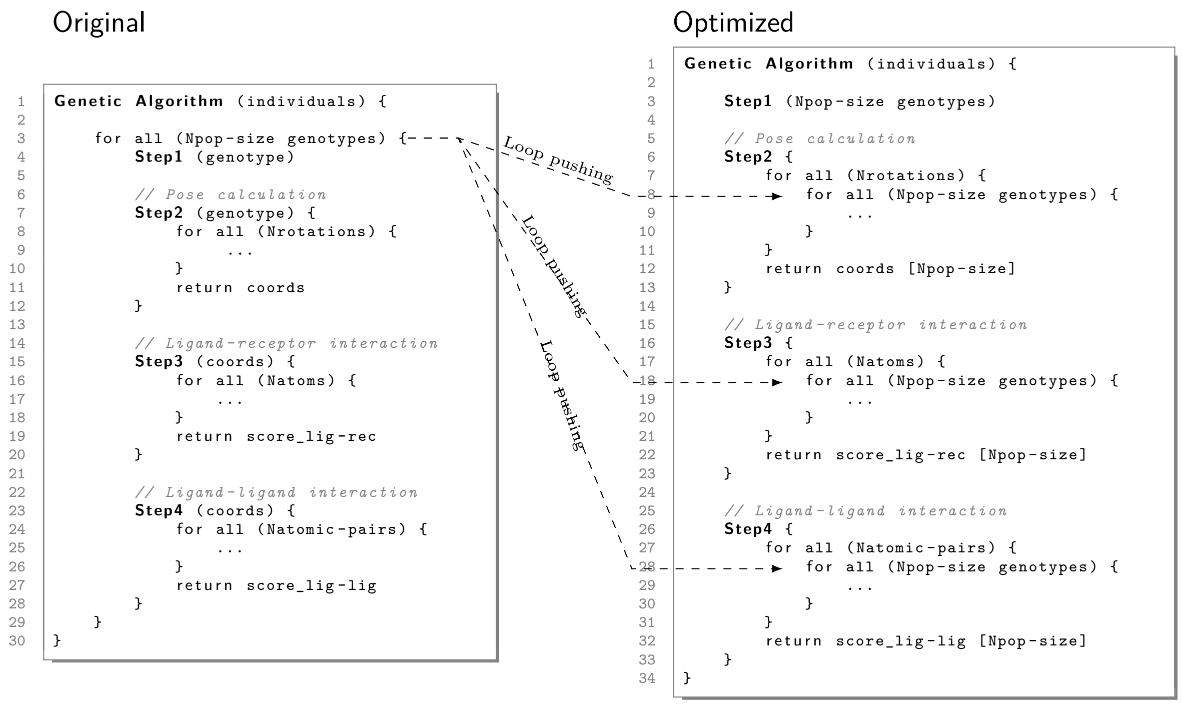
2.2. Optimization

While the ported version already ran correctly, was vectorized and parallelized, its performance was not quite satisfactory yet, executing *slower* by a factor of $\sim 2.2\times$ compared to the host CPU. Each thread of the OpenCL-derived SIMT code being mapped to one VE core was using the vector pipes only for the innermost loops, which are generally quite short. They iterate over the number of atoms of the ligand, or over the number of rotational degrees of freedom of the ligand. For the examples tested, both loop lengths were of the order of magnitude O(10), leading to vector lengths of just 1/10th (or even less!) of the maximum vector length of the VE.

The main optimization approach for increasing the vector lengths of the device kernels was to switch from mapping one SIMT thread to one VE *core*, to mapping one SIMT thread to one *vector lane*. As depicted in Fig. 2, there is no obvious outer loop in the LGA that starts with a genetic population and evolves it while selecting the best scoring individuals. Step 2, Step 3, Step 4 (individual scoring) are the most computationally intensive parts of the algorithm, and can be rewritten so that they handle a large number of individuals at once, in parallel. For each of the individuals, the operations in the scoring functions are basically the same. Thus, we can express this convergent code as either an outer loop over individuals calling the three steps inside, or as three functions which handle, each, the *entire* set of individuals. The loop over individuals can be *pushed* into each of the functions in such a way that it then becomes the innermost, data parallel, and easily vectorizable loop (Fig. 3). For optimal performance, this well-known *loop-pushing* technique must be paired with changes in the data layout, such that the vectorized code accesses data with unit-strides as much as possible.

The Local Search part of the LGA algorithm is computationally divergent code because each of the genetic individuals in the population evolves differently, can mutate with various parameters into different directions, or might already have converged. We were able to perform the *loop-pushing* optimization within the Local Search part by using predication, and compressing the data for the non-converged part of the population (Fig. 4). This aims to keep the compute-intensive scoring functions working with unit-stride accesses and without additional predication. During Local Search, already-converged individuals are removed from the computation, thus reducing the length of the innermost loop. A large population size is thus beneficial for performance, because it increases the average loop length, and thus, the performance during Local Search.

In order to vectorize the Local Search code performing the generation of individuals according to Solis-Wets [24], we needed to replace the linear congruential random generator that was originally employed. The reason for this is that each of the generated random values in the aforementioned scheme depends on the previous one, i.e., $X_{n+1} = f(X_n)$, thus hindering vectorization/parallelization. Instead, we used a 64-bit Mersenne Twister pseudorandom generator [19] implemented in the NEC NLC libraries.



```
Genetic Algorithm (individuals) {

    for all (Npop-size genotypes) {
        Step1 (genotype)

        // Pose calculation
        Step2 (genotype) {
            for all (Nrotations) {
                ...
            }
            return coords
        }

        // Ligand-receptor interaction
        Step3 (coords) {
            for all (Natoms) {
                ...
            }
            return score_lig-rec
        }

        // Ligand-ligand interaction
        Step4 (coords) {
            for all (Natomic-pairs) {
                ...
            }
            return score_lig-lig
        }
    }
}
```

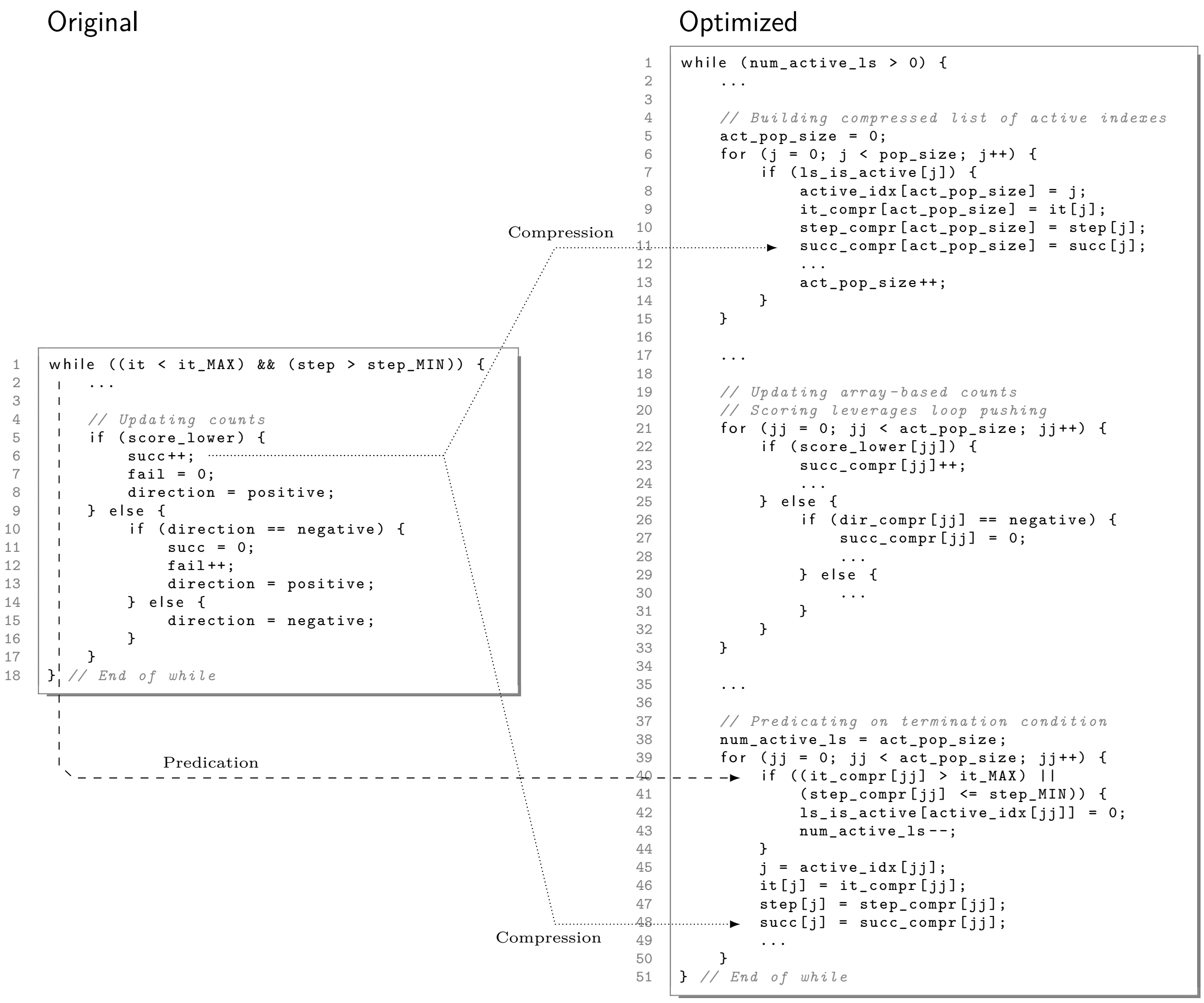
```
Genetic Algorithm (individuals) {

    Step1 (Npop-size genotypes)

    // Pose calculation
    Step2 {
        for all (Nrotations) {
            for all (Npop-size genotypes) {
                ...
            }
        }
        return coords [Npop-size]
    }

    // Ligand-receptor interaction
    Step3 {
        for all (Natoms) {
            for all (Npop-size genotypes) {
                ...
            }
        }
        return score_lig-rec [Npop-size]
    }

    // Ligand-ligand interaction
    Step4 {
        for all (Natomic-pairs) {
            for all (Npop-size genotypes) {
                ...
            }
        }
        return score_lig-lig [Npop-size]
    }
}
```

Figure 3. Optimization in Genetic Algorithm (GA): pushing the outer loop into the three components of the scoring function (Step 2, Step 3, Step 4)

Previous work ocladock-fpga [27] has shown that reducing the numerical precision from double to single floating-point does not impact the ability of the Genetic Algorithm to localize minimal energy configurations. This has been further exploited in AutoDock-GPU [23] by replacing the single precision functions like `expf()`, `sinf()`, `sqrtf()` by `native_exp()`, `native_sin()`, `native_sqrt()`, etc. ..., which provide less numerically-accurate implementations, but with higher performance. We followed this path here as well, but replaced only the single precision functions `sqrtf()` and `expf()` by simplified implementations, yielding reduced-accuracy results, but also requiring fewer floating-point operations.

Furthermore, vectorization of single precision computations on the SX-Aurora VE can be done in two ways: (1) by using vector instructions with up to 256 single-precision elements that are located either in the upper or lower half of the vector register; or (2) by using *packed* vector instructions where each 64-bit vector element of a vector register contains two 32-bit float entities. The later case is called *packed vectorization*, and effectively allows vector lengths of up to 512, with twice the performance of (1).

Unfortunately, packed vectorization has limitations and is more complex to implement in a compiler backend. The vector length must be even, predicated packed vector instructions need two vector mask registers instead of one, and memory access requires several instructions instead of one to account for possible misalignments. Moreover, all operations inside a packed vectorized loop must be executed in single-precision packed mode, otherwise the stream of vector instructions could be disturbed and the performance benefit is lost. Therefore, double-to-float casts, or calls to double-precision mathematical builtin functions (e.g., `ceil()` instead of `ceilf()`), would break packed vectorization. Recent work on the LLVM-VE project [2] has focused on enabling and improving packed vectorization on the VE, and AutoDock's *energy_ia.c* (ligand-ligand) scoring function was used as a benchmark for the progress. Since LLVM-VE's performance using packed vectorization (2) exceeds that of the NEC ncc compiler using the traditional approach (1) for the



```
while ((it < it_MAX) && (step > step_MIN)) {
    ...

    // Updating counts
    if (score_lower) {
        succ++;
        fail = 0;
        direction = positive;
    } else {
        if (direction == negative) {
            succ = 0;
            fail++;
            direction = positive;
        } else {
            direction = negative;
        }
    }
} // End of while
```

```
while (num_active_ls > 0) {
    ...

    // Building compressed list of active indexes
    act_pop_size = 0;
    for (j = 0; j < pop_size; j++) {
        if (ls_is_active[j]) {
            active_idx[act_pop_size] = j;
            it_compr[act_pop_size] = it[j];
            step_compr[act_pop_size] = step[j];
            succ_compr[act_pop_size] = succ[j];
            ...
            act_pop_size++;
        }
    }

    ...

    // Updating array-based counts
    // Scoring leverages loop pushing
    for (jj = 0; jj < act_pop_size; jj++) {
        if (score_lower[jj]) {
            succ_compr[jj]++;
            ...
        } else {
            if (dir_compr[jj] == negative) {
                succ_compr[jj] = 0;
                ...
            } else {
                ...
            }
        }
    }

    ...

    // Predicating on termination condition
    num_active_ls = act_pop_size;
    for (jj = 0; jj < act_pop_size; jj++) {
        if ((it_compr[jj] > it_MAX) ||
            (step_compr[jj] <= step_MIN)) {
            ls_is_active[active_idx[jj]] = 0;
            num_active_ls--;
        }
        j = active_idx[jj];
        it[j] = it_compr[jj];
        step[j] = step_compr[jj];
        succ[j] = succ_compr[jj];
        ...
    }
} // End of while
```

Figure 4. Optimization in Local Search (LS): usage of predication and compression. In the optimized code, predication updates the number of active individuals. An example of compression-based optimization is the replacement of the `succ` scalar variable with the `succ_compr[ ]` array counterpart. In both cases, the number of successful search attempts is counted. In the optimized code, however, the array compresses data for all active individuals

performance-critical function *energy_ia.c*, we compiled this function with LLVM-VE, while using ncc for the rest of the code.

3. Results and Discussion

For validating the docking functionality, we selected a total of 31 ligand-receptor inputs from the list used for validation in [14]. Table 1 shows a dataset subset. The maximum number of rotatable bonds (N_{rot}) in any of our inputs is eight, as recommended when using the Solis-Wets method as Local Search [23].

For profiling executions on the VE, we used the PROGINFO and FTRACE [17] utilities, both providing a large set of performance counters as well as derived performance metrics. The former provides program execution information, while the latter focuses on functions and user regions. As discussed in Section 2.2, the first major optimization was based on loop pushing. Table 2 compares relevant execution metrics for the 1ig3 input, and is used here to analyze the performance impact of this technique. First, the real time represents the wall-clock elapsed time, while the user time

Table 1. Subset of ligand-receptor inputs with their respective number of rotatable bonds (N_{rot}) and atoms (N_{atom})

Input	1ac8	1hnn	1yv3	1owe	1p62	1n46	1ig3	1t46	2bm2	1mzc
N_{rot}	0	2	2	3	4	5	6	6	7	8
N_{atom}	8	18	23	27	22	28	21	40	33	38

accounts for the time spent by all eight cores in the VE. Since the independent LGA runs are distributed among all VE cores via `#pragma omp parallel for` (Section 2.1), the user time is $\sim 8\times$ that of the real time. Moreover, it can be noted that both real and user times were improved by a factor of $\sim$21.3, while the execution time for vector instructions was reduced $\sim 4.9\times$.

Interestingly, the number of all instruction executions (Inst. Count) was reduced $\sim 51\times$, while the FLOP count is almost the *same*. The program does roughly the same number of floating point operations because it computes the same problem as before, but many of the formerly scalar loops are now vectorized with large vector length (> 200), thus the instruction count reduction in the order of $50\times$. The formerly shorter vector loops with the average vector length of 72 are now executed in longer loops, with the average length of 217, reducing the number of vector instructions approximately by a factor of $3.8\times$.

The streamlined vectorized execution, visible in the heavily increased vector operation ratio from 66.4 % to 99.2 %, and the grown average vector length of 217, lead to an increase of the number of overall operations per second (MOPS), and floating-point operations per second (MFLOPS) by $12.5\times$ (1/0.08) and $20.4\times$ (1/0.049), respectively. While the optimal average vector length would be 256, the value could not be reached in practice due to code divergence in the Local Search section. A further significant improvement of the changed code is shown in the reduction of the Level 1 Cache Miss time, from 44.2 s to 2.1 s. A L1 Cache Miss can only occur in scalar code, and the lowered value reflects the significantly reduced number of scalar instructions executed in the optimized version.

Table 3 summarizes the impact of further optimizations as relative time changes of the total evaluation. Experiments were performed on one VE using the 1ig3 input with $N_{\text{LGA-runs}} = 100$. The relative time change was computed as the percentage of the time change compared to the unoptimized case: $100 \times (t_{\text{optimized}} - t_{\text{unoptimized}})/t_{\text{unoptimized}}$. The first line reflects the gains described by Tab. 2 that lead to the reduction of the compute time by 95.3 %. The following lines show further optimizations of the loop pushing code.

When the VE was switched to NUMA partitioned mode, the multi-process code benefits from the reduced memory network contention and CPU port conflicts. As this problem only impacts the ligand-receptor energy calculation that is dominated by indirect memory accesses, the impact is small (–6.9 %) and depends a lot on the specific problem. As NUMA measurements force us to use two processes (on a VE) with 4 cores each, the values in Tab. 3 are compared against similar runs on non-NUMA VEs. Using two processes in non-NUMA mode has a higher overhead than running just one process with 8 cores. Since the NUMA benefits are in the range of this overhead, 2×4 cores with NUMA and 1×8 cores UMA timings are practically the same. A significant improvement could be achieved by the packed vectorization, that in turn, could only be achieved with the LLVM-VE compiler using the RegionVectorizer RV. This change almost doubled the execution speed of the ligand-ligand interaction scoring function, and led to an evaluation time reduction of 36.2 %. At this point, it is important to indicate that, we opted to use a single input

Table 2. Execution metrics of AutoDock-Aurora for the 1ig3 input, before and after applying loop pushing. Information was obtained using NEC PROGINF [17]

Metric	Optimization: loop pushing		Ratio
	Before	After	Before / After
Real Time [sec]	307.5	14.5	21.3
User Time [sec]	2,458.1	115.0	21.4
Vector Time [sec]	510.2	104.0	4.91
Inst. Count	5,085,000,001,257	98,888,607,313	51.4
Vec. Inst. Count	120,865,697,285	32,136,492,289	3.76
FLOP Count	4,982,577,754,822	4,826,280,301,843	1.03
MOPS	6,012.0	75,174.3	0.08
MOPS (Real)	48,082.1	597,857.0	0.08
MFLOPS	2,027.0	41,960.7	0.048
MFLOPS (Real)	16,211.5	333,711.3	0.049
Avg. Vec. Length	71.5	216.9	0.33
V. Op. Ratio [%]	66.4	99.2	0.67
L1 Cache Miss [sec]	44.2	2.1	20.4

(i.e., 1ig3) in order to provide a simple but yet reasonable analysis. Using the full dataset for such analysis would be ideal, but not strictly necessary. The reason is that, e.g., for ligands with larger N_{rot} and N_{atom}, the length of compute-intensive loops is in turn larger, and thus, we can safely expect larger benefits than those for 1ig3 in Tab. 3. Finally, the use of reduced-precision replacements for `sqrtf` and `expf` inside the ligand-ligand interaction scoring function led to a gain of 25.4 % with LLVM-VE, but to a time loss (slowdown!) with the ncc compiler, a sign that the NEC ncc compiler provides fast implementations for these functions.

Table 3. Overview of improvements obtained through various optimizations on one VE using the 1ig3 input. Larger negative values for the relative time change are better. All optimizations below the *loop pushing* line show additional gains (or losses) on top of this vectorized code

Optimization	Rel. time change
Loop pushing and vectorized random generator	–95.3 %
VE in NUMA partitioned mode	–6.9 %
Packed vectorization of *energy_ia.c* with llvm-ve	–36.2 %
Reduced precision `sqrtf` and `expf` with ncc	29.5 %
Reduced precision `sqrtf` and `expf` with llvm	–25.4 %

Finally, the execution runtimes of AutoDock-Aurora are compared against those of AutoDock-GPU [23], the state-of-the-art OpenCL-based implementation of AutoDock for GPUs/CPUs. Table 4 lists the accelerator devices equipping the systems employed. We used version v1.1 of AutoDock-GPU, in order to ensure a fair comparison, aiming for having equivalent functionality in both GPU and VE implementations. For all executions of both AutoDock-Aurora and AutoDock-GPU, we set the $\mathrm{LS}_{\mathrm{rate}}$ to 100 % instead of the default 6 %, as real-world experiments with

AutoDock-GPU typically use the highest practical $\mathrm{LS}_{\mathrm{rate}}$ value. In this configuration, all members of a population undergo Local Search, and thus, the program has higher chances to produce more-accurate molecular predictions.

Table 4. Technical characteristics of the SX-Aurora VEs, GPUs and CPUs used in the evaluation: base clock frequency (Freq), number of cores (Ncores), FP32 performance (Perf), memory bandwidth (MemBW). GPUs are composed of independent Streaming Multiprocessors (SM). Both CPU platforms posses two sockets each

Characteristics	**SX-Aurora**		**GPU**			**CPU**	
	10B	**20B**	**RTX2070**	**V100**	**A100**	**EPYC 7502**	**EPYC 7742**
Freq [GHz]	1.4	1.6	1.61	1.23	0.76	2.5	2.25
Ncores	8	8	2560	5120	6912	32 × 2	64 × 2
Perf [TFLOPS]	4.3	4.9	9.1	14.1	19.5	2.6	4.6
MemBW [GB/s]	1220	1530	448	897	1555	204.8 × 2	204.8 × 2
L1 Cache	32 kB (SPU I$) 32 kB (SPU O$)		64 kB (per SM)	128 kB (per SM)	192 kB (per SM)	96 kB (per core)	96 kB (per core)
L2 Cache	256 kB (SPU) 128 kB (VPU)		4 MB (shared)	6 MB (shared)	40 MB (shared)	512 kB (per core)	512 kB (per core)
L3 Cache	16 MB LCC (shared)		-	-	-	128 MB (shared)	256 MB (shared)

As shown in Fig. 5, larger population sizes increase the performance on the VE, but do not impact it as much on any other architectures. We attribute the different performance behavior of AutoDock-GPU to the workload distribution strategy used in OpenCL. In AutoDock-GPU, the population size directly affects the number of spawned OpenCL work-groups ($N_{\mathrm{WG}} = N_{\text{pop-size}} \times N_{\text{LGA-run}}$), but has no impact on the execution time of a score evaluation. As described in Section 1.2, the LGA terminates when the number of score evaluations reaches an upper bound (i.e., $N^{\mathrm{MAX}}_{\text{score-evals}}$). As a consequence, processing larger populations requires *fewer* iterations per LGA run, and thus compensates for the seemingly bigger workload imposed by the need for more individuals to be processed. The slight increase of runtimes on GPUs/CPUs for the larger populations is likely due to the synchronization overhead introduced by the additional OpenCL work-groups. On the other hand, in AutoDock-Aurora, larger populations positively impact the performance of the pushed-in loops, enabled by their longer vector lengths (Section 2.2). Since the purpose of molecular docking with genetic algorithms is to test larger number of genetic individuals, we see no disadvantage for other architectures when choosing a large population size that is optimal for the VE.

For the sake of clarity, Fig. 6 shows only the results when using $N_{\text{pop-size}} = 2048$. It can be observed that the VE 20B significantly outperforms the dual socket state-of-the-art AMD EPYC nodes by average factors of 6.5× (= 44.3/6.8) and 3.6× (= 25.1/6.8).

In terms of the underlying semiconductors, the VEs are at the same 16nm-node as the V100 GPU, but have a lower peak performance of 2.8× (= 14.1/4.9, Tab. 4) compared to the GPU. The performance of our implementation is slower than the V100 by a factor of 2.4× (= 6.8/2.8, Fig. 6) and thus comes very close to the theoretical peak performance ceiling. Comparing to the A100 GPU, while its peak only increased by ∼1.4× (= 19.5/14.1, Tab. 4) over the V100, the A100's average execution time shows much better performance than the latter: ∼2.5× (= 3.8/1.5). We

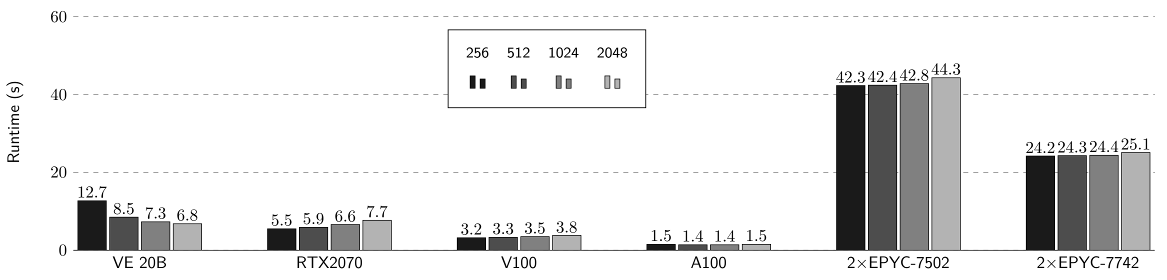


Figure 5. Geometric mean of execution runtimes over 31 inputs, comparing the impact of the chosen population size: $N_{\text{pop-size}} = \{256, 512, 1024, 2048\}$. AutoDock-Aurora was executed on the VE 20B, while AutoDock-GPU v1.1 on the GPUs and CPUs. In all executions: $N_{\text{LGA-runs}} = 100$, $\text{LS}_{\text{rate}} = 100$ %. Other parameters were left at default values

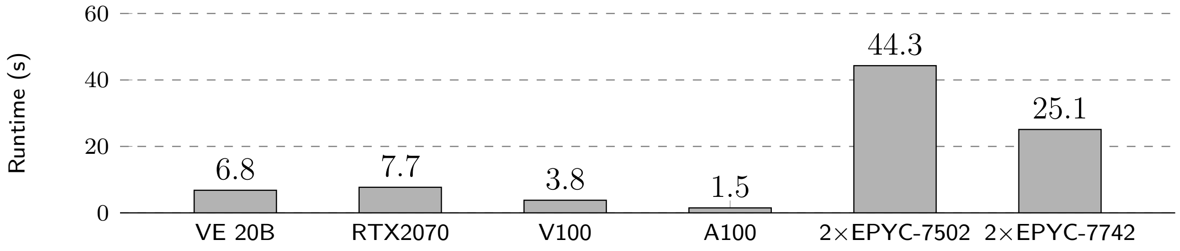


Figure 6. Geometric mean of execution runtimes over 31 inputs. AutoDock-Aurora was executed on the VE 20B, while AutoDock-GPU v1.1 on the GPUs and CPUs. In all executions: $N_{\text{popsize}} = 2048$, $N_{\text{LGA-runs}} = 100$, $\text{LS}_{\text{rate}} = 100$ %. Other parameters were left at default values

assume that the higher performance and bandwidth capabilities of the A100 are the main reason for the fastest executions on this platform.

As reported in [23], AutoDock-GPU achieves faster executions on GPUs than on CPUs, which is attributed to the more suitable mapping of OpenCL elements onto the underlying hardware. On CPUs, each OpenCL work-group is executed by a single CPU core, and thus, work-items (threads) are executed serially [11]. On GPUs, work-groups and work-items are executed in parallel by the fine-grain GPU streaming multiprocessors.

4. Related Work

Studies on benchmarking the performance of the SX-Aurora using various applications are reported as follows. Komatsu et al. used standard benchmarks and a tsunami numerical simulation code [13]. These authors introduced a performance model based on different *Byte per FLOP* (B/F) rates to analyze the bottleneck causes in applications. Onodera et al. optimized the Himeno benchmark, which solves the Poisson's equation using the Jacobi iteration method. The vector systems used in Onodera et al.'s evaluation were configured with a single and up to eight VEs [21]. While experiments in [13] and [21] were performed on SX-Aurora Type 10B, Egawa et al. used *second-generation* Type 20B devices to benchmark their optimization strategies on different scientific applications [6]. Moreover, Egawa et al. refined the model introduced by Komatsu et al. in [13]. This newly-proposed model considers a *possible* peak performance (FLOP/s), which can be determined by the FMA instruction rate of applications and sustained memory bandwidth (B/s), instead of employing their *peak* values.

Furthermore, Takizawa et al. proposed an OpenCL-like offload programming framework for SX-Aurora [28]. As the OpenCL execution model (originally designed for GPUs) does not fit well for all compute architectures (e.g., FPGAs, VEs), this framework allows the usage of different programming languages for implementing the host and device code. Particularly, the host code can be written in OpenCL C/C++, while the device code in standard C++. In general, using such a framework would potentially reduce the porting effort from existing OpenCL code as in our case. However, we opted to implement both host and device code in standard C++, by explicitly invoking VEO APIs to interact with the VE, and writing vectorizable loops. In this manner, we avoided relying on external APIs, and exploited the compiler's capability of automatic vectorization.

To the best of our knowledge, our work here on AutoDock-Aurora is the *first one* leveraging vector computing for molecular docking. The closest studies related to ours are the hardware-accelerated implementations reported in [25]. From these studies, accelerator devices such as CPUs, GPUs, and even FPGAs, have been used in single computing nodes. Moreover, parallelization strategies are not only based on the docking algorithm under analysis, but also on hardware-specific aspects of the target device. Here, we report previous studies describing AutoDock.

Regarding GPUs, AutoDock has its official release rebranded as AutoDock-GPU. It has been originally written in OpenCL [23], and afterwards, ported from the original OpenCL to CUDA [14]. This CUDA implementation was used as the docking engine for COVID-19 research on the Summit supercomputer, where OpenCL was not supported. Furthermore, AutoDock has been ported to FPGAs as well, where various efforts differ in the design approaches they adopted. Pechan et al. followed the *traditional* development (for FPGAs) by describing the docking functionality in terms of low-level transfers between hardware registers and synchronous logic design [22]. On the other hand, Solis-Vasquez et al. followed a *high-level* design approach based on OpenCL to develop ocladock-fpga [27]. In contrast to AutoDock-GPU, which parallelizes over multiple *data items* (i.e., genotypes), ocladock-fpga executes multiple *tasks* concurrently. Furthermore, it relies on pipelined hardware logic and custom memory hierarchies. In terms of performance, ocladock-fpga on an Arria-10 FPGA runs $\sim 2\times$ faster than the original AutoDock on a CPU, but it is still significantly slower compared to AutoDock-GPU. Hence, it is not being deployed for realistic docking problems.

Conclusions and Future Work

In this work, we have ported and optimized AutoDock onto the SX-Aurora TSUBASA platform. The molecular docking application consists of a Genetic Algorithm coupled with a Local Search part, which has a divergent flow with frequent calls to compute-intense score evaluations. To the best of our knowledge, this is the *first* molecular docking and Genetic Algorithm implementations for the SX-Aurora TSUBASA.

Regarding the porting process, the programming framework provided by NEC enabled a smooth experience. However, the optimization was much more involved, requiring the combination of a number of different strategies. Of these, loop pushing improved the performance significantly, but required more code refactoring in the Local Search part. Basically, the original SIMT nature of the Local Search, treating individuals in different threads, is now expressed in an explicit, vectorizable loop. Regarding the floating-point arithmetic, the computations offloaded to the VE were expressed using single-precision only, and in turn, were vectorized in *packed mode* for higher performance. Furthermore, we explored mixed compilation, employing both LLVM-VE and NEC compilers for the scoring-function components. As a result, our implementation is in average $3.6\times$ faster than 128-core CPU servers, while being competitive to V100 and A100 GPUs.

As a future work, we will incorporate alternative methods for the Local Search part. Namely, ADADELTA [30], which compared to the Solis-Wets method examined here performs more complex computations, but yields molecular predictions of higher quality [23]. Moreover, we plan to analyze the achieved efficiency using performance models, e.g., those based on Byte/FLOP ratios, as proposed in [6, 13].

References

1. FightAIDS@Home. `https://www.worldcommunitygrid.org/research/faah/overview.do` (2021), accessed: 2021-06-01
2. LLVM-VE project GitHub repository. `https://github.com/sx-aurora-dev/llvm-project` (2021), accessed: 2021-05-31
3. OpenPandemics: COVID-19. `https://www.worldcommunitygrid.org/research/opn1/overview.do` (2021), accessed: 2021-06-01
4. VEDA GitHub repository. `https://github.com/SX-Aurora/veda` (2021), accessed: 2021-05-19
5. Cramer, T., Römmer, M., Kosmynin, B., *et al.*: OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine. In: Wyrzykowski, R., Deelman, E., Dongarra, J.J., *et al.* (eds.) Parallel Processing and Applied Mathematics - 13th Int. Conf., PPAM 2019, Bialystok, Poland, Sept. 8-11, 2019, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 12043, pp. 237–249. Springer (2019). `https://doi.org/10.1007/978-3-030-43229-4_21`
6. Egawa, R., Fujimoto, S., Yamashita, T., *et al.*: Exploiting the potentials of the second generation SX-Aurora TSUBASA. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2020, Atlanta, GA, USA, Nov. 12, 2020. pp. 39–49. IEEE (2020). `https://doi.org/10.1109/PMBS51919.2020.00010`
7. Fischer, E.: Einfluss der Configuration auf die Wirkung der Enzyme. II. Berichte der deutschen chemischen Gesellschaft 27(3), 3479–3483 (1894). `https://doi.org/10.1002/cber.189402703169`
8. Focht, E.: VEO and PyVEO: Vector Engine Offloading for the NEC SX-Aurora Tsubasa. In: Resch, M.M., Kovalenko, Y., Bez, W., *et al.* (eds.) Sustained Simulation Performance 2018 and 2019. pp. 95–109. Springer (2020). `https://doi.org/10.1007/978-3-030-39181-2_9`
9. Halperin, I., Ma, B., Wolfson, H., Nussinov, R.: Principles of docking: An overview of search algorithms and a guide to scoring functions. Proteins: Struct., Funct., Bioinf. 47(4), 409–443 (2002). `https://doi.org/10.1002/prot.10115`
10. Huey, R., Morris, G.M., Olson, A.J., Goodsell, D.S.: A semiempirical free energy force field with charge-based desolvation. J. Comput. Chem. 28(6), 1145–1152 (2007). `https://doi.org/10.1002/jcc.20634`

11. Kaeli, D., Mistry, P., Schaa, D., Zhang, D.P.: Heterogeneous Computing with OpenCL 2.0. Morgan Kaufmann, 3 edn. (2015)

12. Ke, Y., Agung, M., Takizawa, H.: neoSYCL: a SYCL implementation for SX-Aurora TSUBASA. In: Hwang, S., Yeom, H.Y. (eds.) HPC Asia 2021: The Int. Conf. on High Performance Computing in Asia-Pacific Region, Virtual Event, Republic of Korea, Jan. 20-21, 2021. pp. 50–57. ACM (2021). `https://doi.org/10.1145/3432261.3432268`

13. Komatsu, K., Momose, S., Isobe, Y., *et al.*: Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In: SC18: Int. Conf. for High Performance Computing, Networking, Storage and Analysis. pp. 685–696. IEEE (2018). `https://doi.org/10.1109/SC.2018.00057`

14. LeGrand, S., Scheinberg, A., Tillack, A.F., *et al.*: GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research. In: BCB '20: 11th ACM Int. Conf. on Bioinformatics, Computational Biology and Health Informatics, Virtual Event, USA, Sept. 21-24, 2020. pp. 43:1–43:10. ACM (2020). `https://doi.org/10.1145/3388440.3412472`

15. Liu, J., Wang, R.: Classification of Current Scoring Functions. J. Chem. Inf. Model. 55(3), 475–482 (2015). `https://doi.org/10.1021/ci500731a`

16. Morris, G.M., Goodsell, D.S., Halliday, R.S., *et al.*: Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. J. Comput. Chem. 19(14), 1639–1662 (1998). `https://doi.org/10.1002/(SICI)1096-987X(19981115)19:14<1639::AID-JCC10>3.0.CO;2-B`

17. NEC: PROGINF/FTRACE User Guide. `https://www.hpc.nec/documents/sdk/pdfs/g2at03e-PROGINF_FTRACE_User_Guide_en.pdf` (2018), accessed: 2021-05-31

18. NEC: SX-Aurora TSUBASA Performance Tuning Guide. `https://www.hpc.nec/documents/guide/pdfs/AuroraVE_TuningGuide.pdf` (2020), accessed: 2021-05-29

19. Nishimura, T.: Tables of 64-bit Mersenne Twisters. ACM Trans. Model. Comput. Simul. 10(4), 348–357 (2000). `https://doi.org/10.1145/369534.369540`

20. Noack, M., Focht, E., Steinke, T.: Heterogeneous active messages for offloading on the NEC SX-Aurora TSUBASA. In: IEEE Int. Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019. pp. 26–35. IEEE (2019). `https://doi.org/10.1109/IPDPSW.2019.00014`

21. Onodera, A., Komatsu, K., Fujimoto, S., *et al.*: Optimization of the Himeno benchmark for SX-Aurora TSUBASA. In: Wolf, F., Gao, W. (eds.) Benchmarking, Measuring, and Optimizing - Third BenchCouncil Int. Symposium, Bench 2020, Virtual Event, Nov. 15-16, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12614, pp. 127–143. Springer (2020). `https://doi.org/10.1007/978-3-030-71058-3_8`

22. Pechan, I., Fehér, B., Bérces, A.: FPGA-based acceleration of the AutoDock molecular docking software. In: Proc. of the 6th Conf. on Ph.D. Research in Microelectronics Electronics. pp. 1–4. IEEE (2010), `https://ieeexplore.ieee.org/document/5587139`

23. Santos-Martins, D., Solis-Vasquez, L., Tillack, A.F., *et al.*: Accelerating AutoDock4 with GPUs and Gradient-Based Local Search. J. Chem. Theory Comput. 17(2), 1060–1073 (2021). `https://doi.org/10.1021/acs.jctc.0c01006`

24. Solis, F.J., Wets, R.J.B.: Minimization by Random Search Techniques. Math. Oper. Res. 6(1), 19–30 (1981). `https://doi.org/10.1287/moor.6.1.19`

25. Solis-Vasquez, L.: Accelerating Molecular Docking by Parallelized Heterogeneous Computing - A Case Study of Performance, Quality of Results, and Energy-Efficiency using CPUs, GPUs, and FPGAs. Ph.D. thesis, Technical University of Darmstadt, Germany (2019). `https://doi.org/10.25534/tuprints-00009288`

26. Solis-Vasquez, L., Koch, A.: A performance and energy evaluation of opencl-accelerated molecular docking. In: McIntosh-Smith, S., Bergen, B. (eds.) Proc. of the 5th Int. Workshop on OpenCL, IWOCL 2017, Toronto, Canada, May 16-18, 2017. pp. 3:1–3:11. ACM (2017). `https://doi.org/10.1145/3078155.3078167`

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

28. Takizawa, H., Shiotsuki, S., Ebata, N., Egawa, R.: An OpenCL-like offload programming framework for SX-Aurora TSUBASA. In: 20th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, Dec. 5-7, 2019. pp. 282–288. IEEE (2019). `https://doi.org/10.1109/PDCAT46702.2019.00059`

29. Wang, Z., Sun, H., Yao, X., *et al.*: Comprehensive evaluation of ten docking programs on a diverse set of protein–ligand complexes: the prediction accuracy of sampling power and scoring power. Phys. Chem. Chem. Phys. 18(18), 12964–12975 (2016). `https://doi.org/10.1039/C6CP01555G`

30. Zeiler, M.D.: ADADELTA: An Adaptive Learning Rate Method. arXiv abs/1212.5701 (2012)