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

Leonardo Solis-Vasquez[+] Erich Focht[#] Andreas Koch[+]

(+) Technical University of Darmstadt (#) NEC Deutschland GmbH

11th Workshop on Irregular Applications: Architectures and Algorithms (IA^3)
November 15, 2021

Contents

Introduction

Research Context (1/2)

- Computer-Aided Drug Design (CADD)
  - Contributes fighting against diseases
    - AIDS
    - cancer
    - COVID-19

Research Context (1/2)

- Computer-Aided Drug Design (CADD)
  - Contributes fighting against diseases
    - AIDS
    - cancer
    - COVID-19

- Molecular docking simulations
  - Key method in CADD
  - Predict molecular interactions at short distances
  - Benefits
    - Shorten the task of identifying drug candidates
    - Reduce the overall need for costly and slow wet lab experiments

Research Context (2/2)

- Widely-used accelerators in High Performance Computing (HPC)
  - CPU
  - GPU
  - Others

Research Context (2/2)

- Widely-used accelerators in High Performance Computing (HPC)
  - CPU
  - GPU
  - Others

- SX-Aurora TSUBASA
  - Vector-based processing
  - High-memory bandwidth (1.53 TB/s)
  - Programming framework based on C++
  - Successfully used in recent studies on HPC applications

Is the SX-Aurora a competitive alternative for molecular docking ?

Our Contributions

- Investigate porting AutoDock molecular docking to Vector Engine (VE)
  - AutoDock methods are irregular and complex
    - Divergent control flow
    - Compute-intensive calculations

Our Contributions

- Investigate porting AutoDock molecular docking to Vector Engine (VE)
  - AutoDock methods are irregular and complex
    - Divergent control flow
    - Compute-intensive calculations

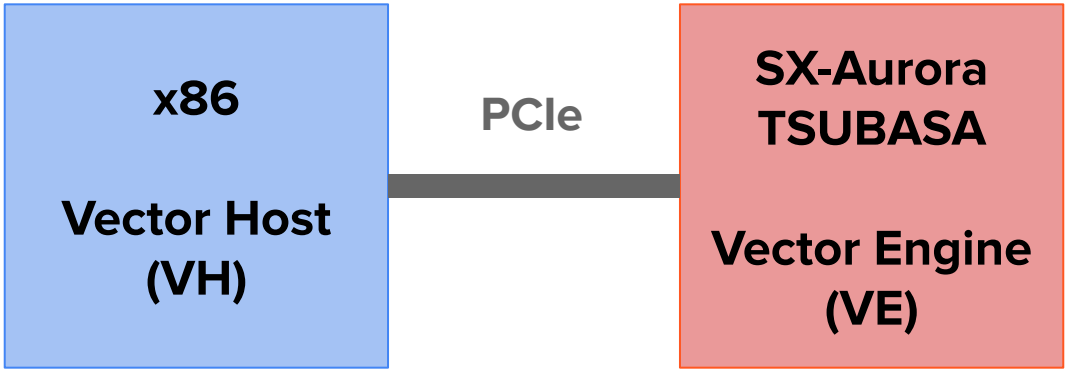
- Evaluate the achievable performance on the VE
  - Analyzing the impact of VE-specific coding techniques
  - Benchmarking against GPUs and CPUs

Background

SX-Aurora TSUBASA Vector Engine

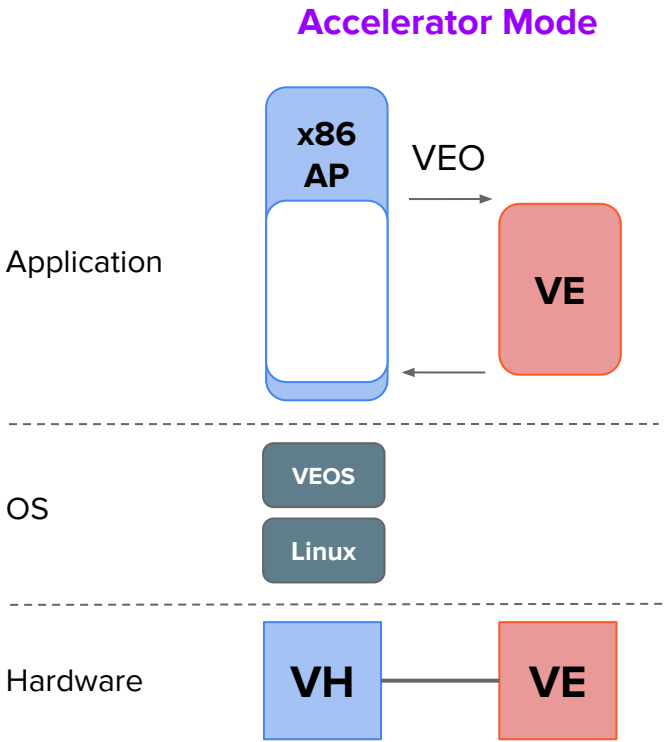
- Device Characteristics

Overall System



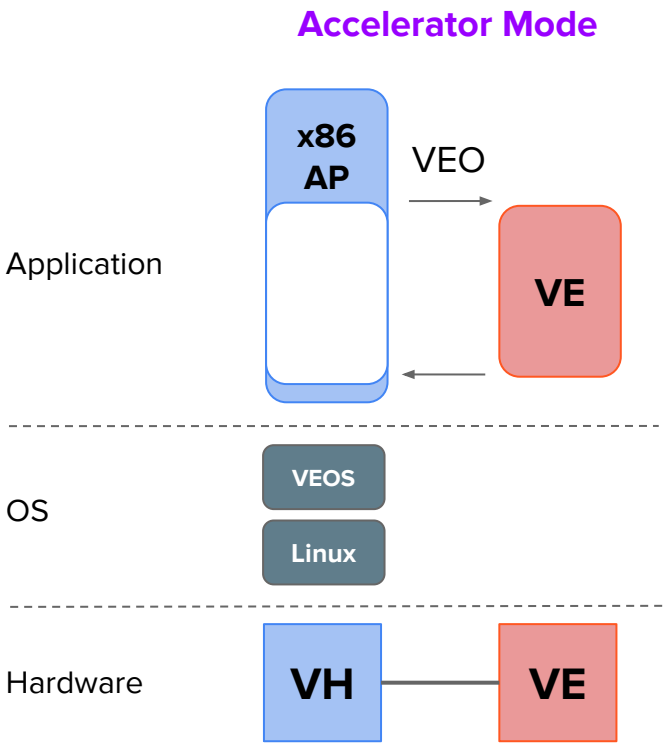
Execution Modes

We use this execution mode



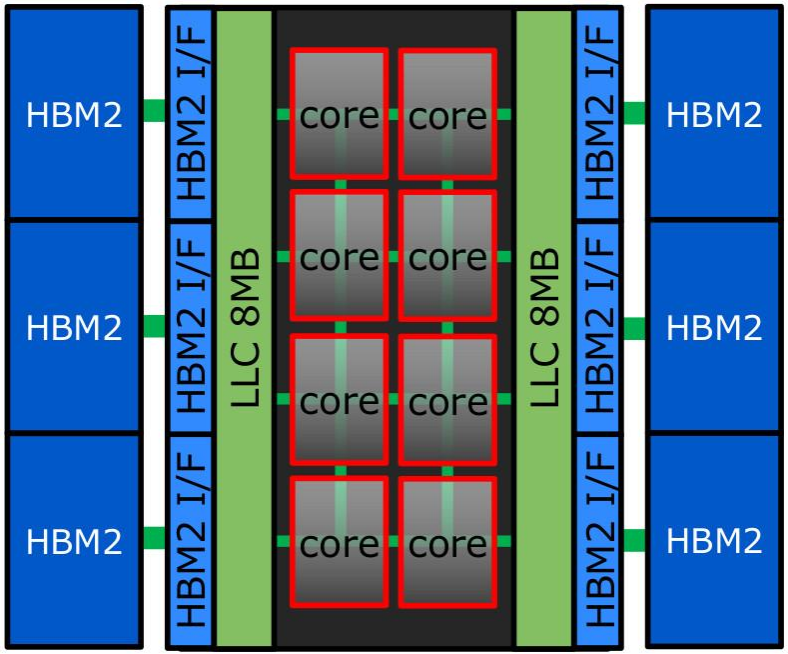
Execution Modes

- Vector Engine Offloading (VEO)
  - Programming model
  - Main program → VH
  - Compute kernels → VE
- VEO provides host APIs
  - API functions resemble those of OpenCL
- VEO can express
  - Kernel offloading
  - VH ↔ VE data movement



Vector Engine (1/2)

- Vector Engine (VE)
  - Eight cores
- Last Level Cache (LLC)
  - 16 MB
- RAM
  - 8 GB HMB2 x 6 (total: 48 GB)



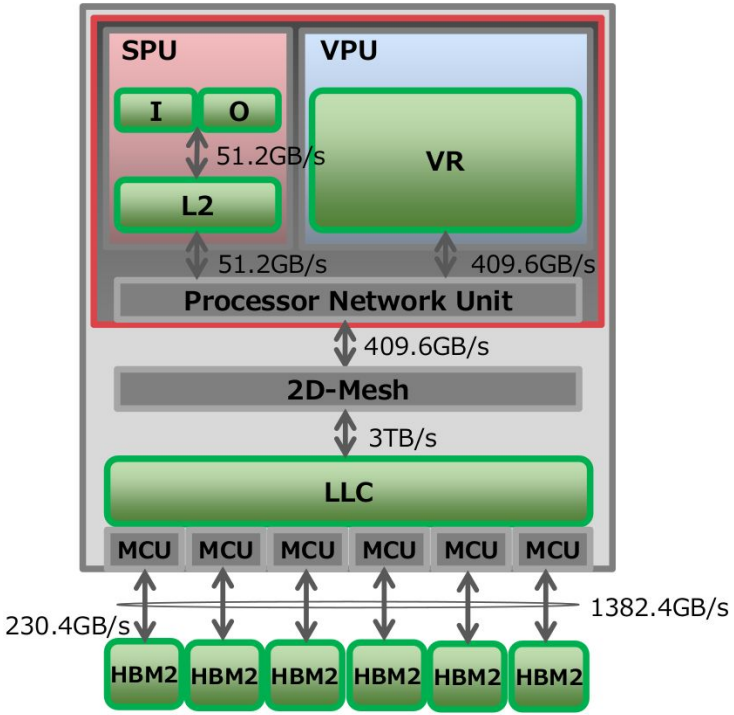
Source: NEC

Vector Engine (2/2)

- VE core
  - Scalar Processing Unit (SPU)
  - Vector Processing Unit (VPU)

- SPU
  - RISC instruction set, out-of-order
  - I cache: 32 kB
  - O cache: 32 kB
  - L2 cache: 256 kB

- VPU
  - 64 Vector Registers (VR)
  - 32 elements x 64-bit wide SIMD units
    - 8-cycle deep pipelines
  - 256 elements x 8 Byte x 64 = 128 kB

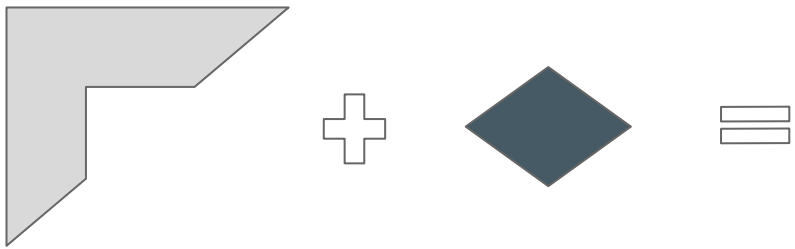


Source: NEC

Molecular Docking

- Overview
- AutoDock

Molecular Docking



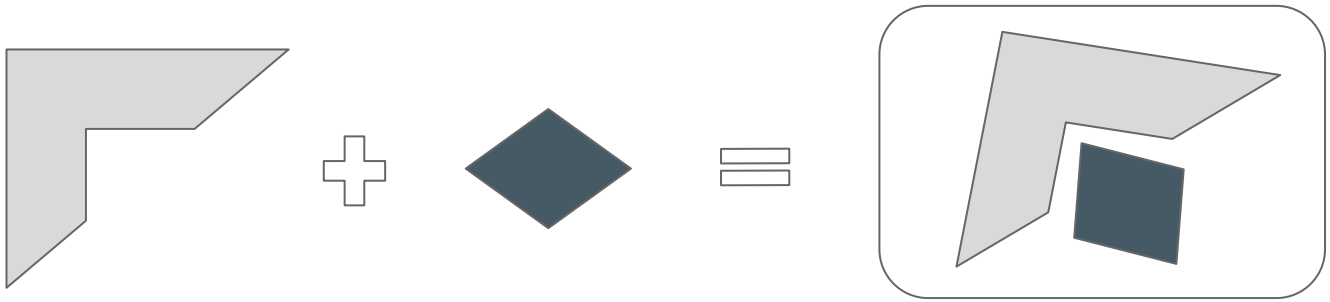
Receptor

(large molecule)

Ligand

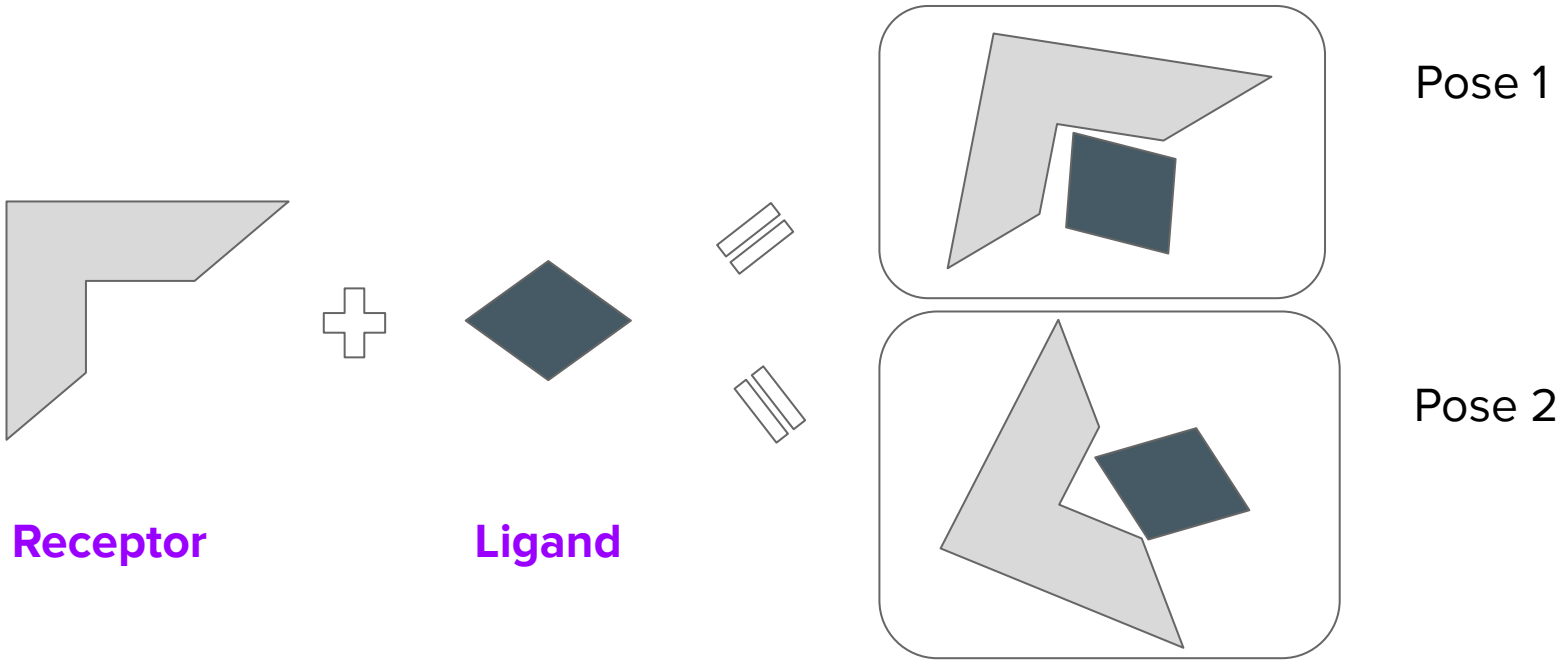
(small molecule)

Molecular Docking

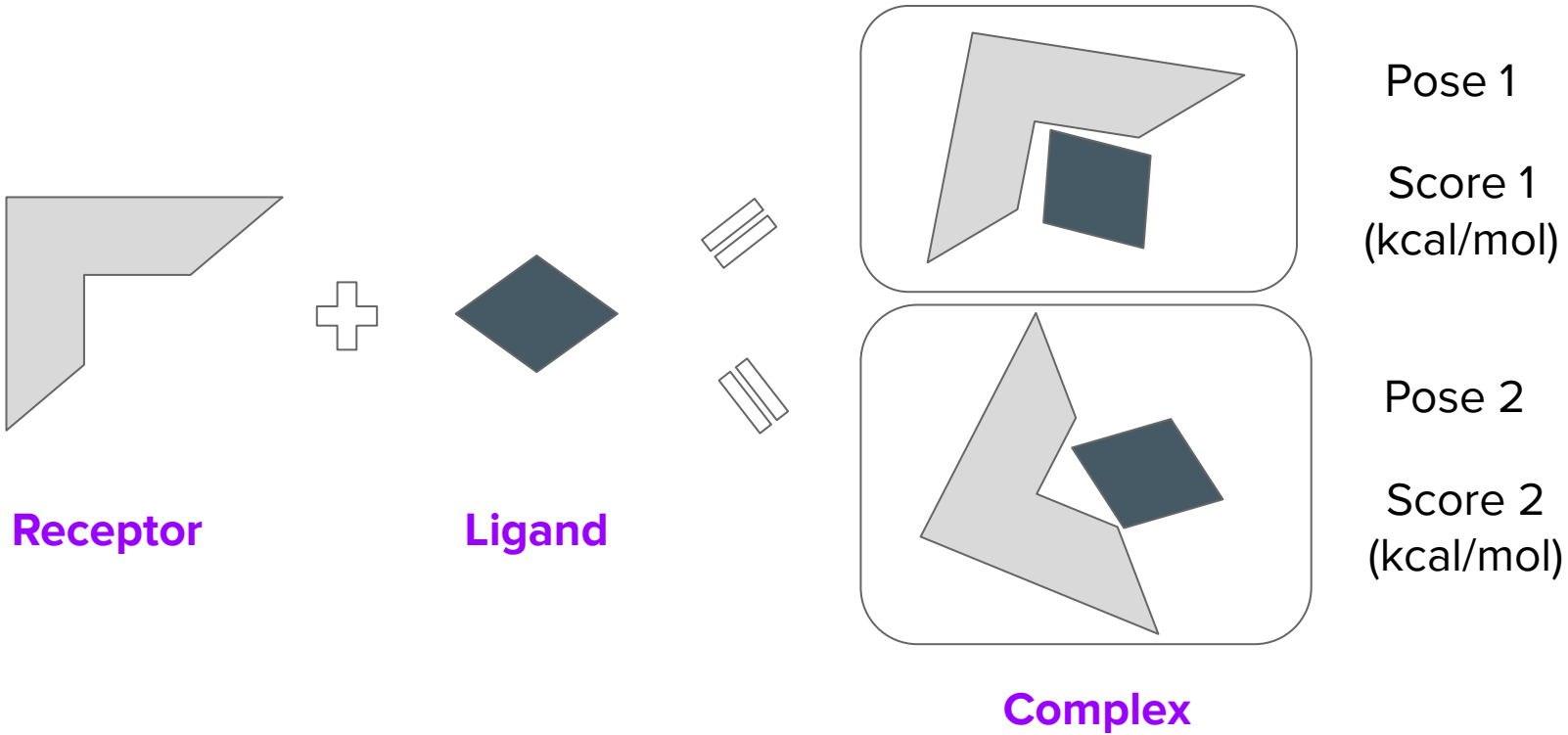


Receptor **Ligand** **Complex**

Molecular Docking



Molecular Docking



AutoDock

- Widely used
  - Open source & implemented in C++
  - Developed by Scripps Research (USA)

- Part of the AutoDock Software Suite
  - AutoDock-Vina
  - AutoDock-GPU
  - Many more ...

- Large-scale projects
  - *FightAIDS@Home*
  - *OpenPandemics: COVID-19*

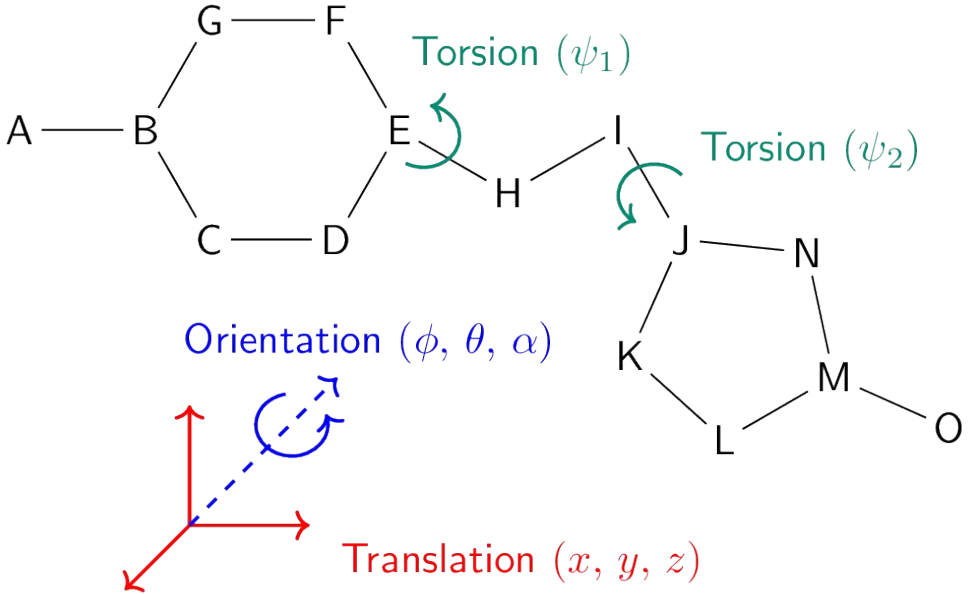
AutoDock: Receptor-Ligand docking

- Receptor
  - Large molecule
  - Treated as a rigid body

- Ligand
  - Small molecule
  - Treated as flexible

AutoDock: Receptor-Ligand docking

- Receptor
  - Large molecule
  - Treated as a rigid body
- Ligand
  - Small molecule
  - Treated as flexible
- Ligand poses
  - Encoded with variables
  - Each pose has an associated score

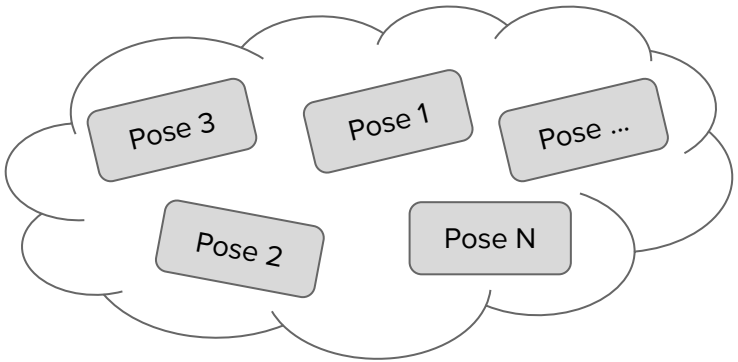


$\text{Pose}_{\text{Ligand}} = \{x,\ y,\ z,\ \phi,\ \Theta,\ \alpha,\ \Psi 1,\ \Psi 2\}$

Mapping Docking into Genetic Evolution

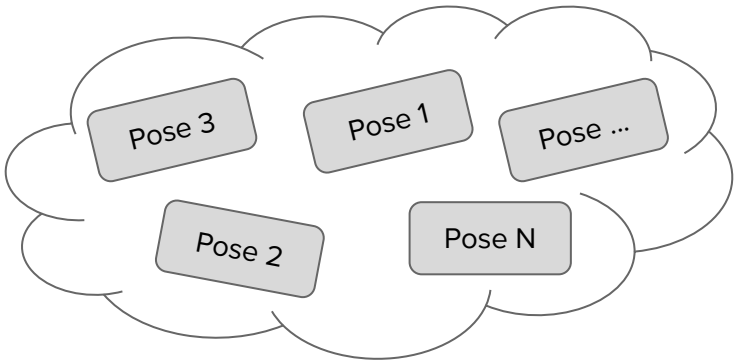
- Pose ➜ individual

- Individual
  - Member of a population
  - Represented by its genotype

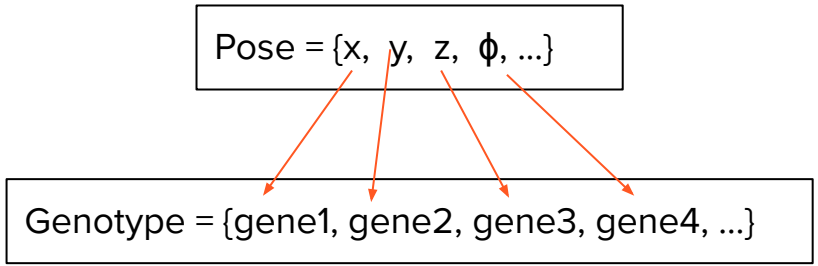


Mapping Docking into Genetic Evolution

- Pose → individual
- Individual
  - Member of a population
  - Represented by its genotype



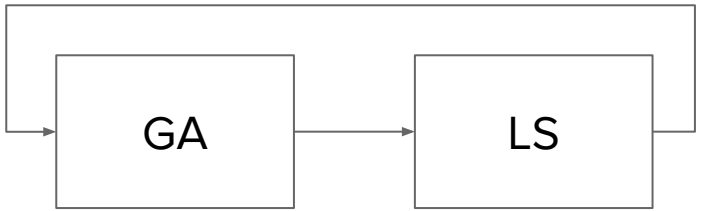
- Genotype
  - Composed of set of genes
- (Pose) variable ← → gene



Lamarckian Genetic Algorithm (1/4)

- AutoDock performs an iterative hybrid search
  - Over populations (of poses)

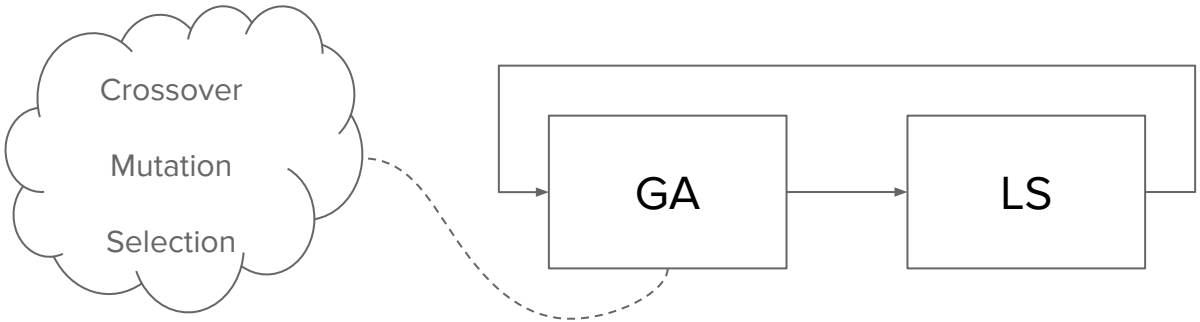
- LGA = GA + LS
  - Genetic Algorithm (GA)
  - Local Search (LS)



Lamarckian Genetic Algorithm (2/4)

Genetic Algorithm (GA)

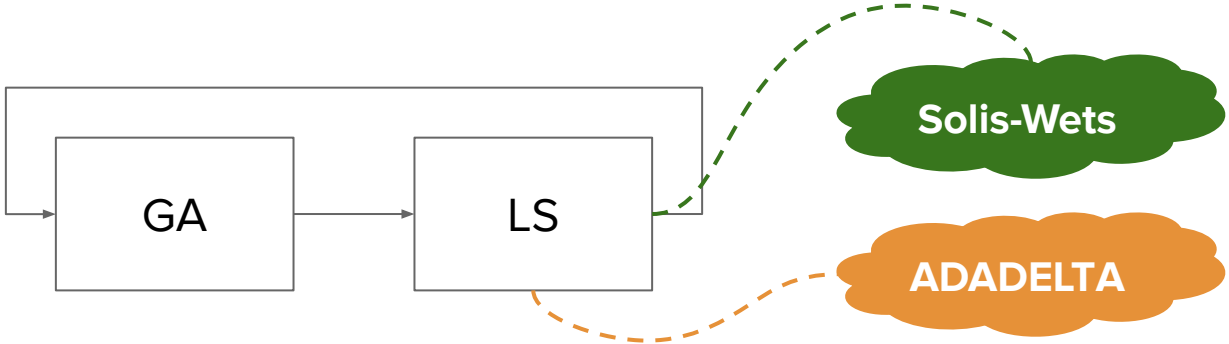
- New individuals are generated through genetic evolution
  - Genetic operations



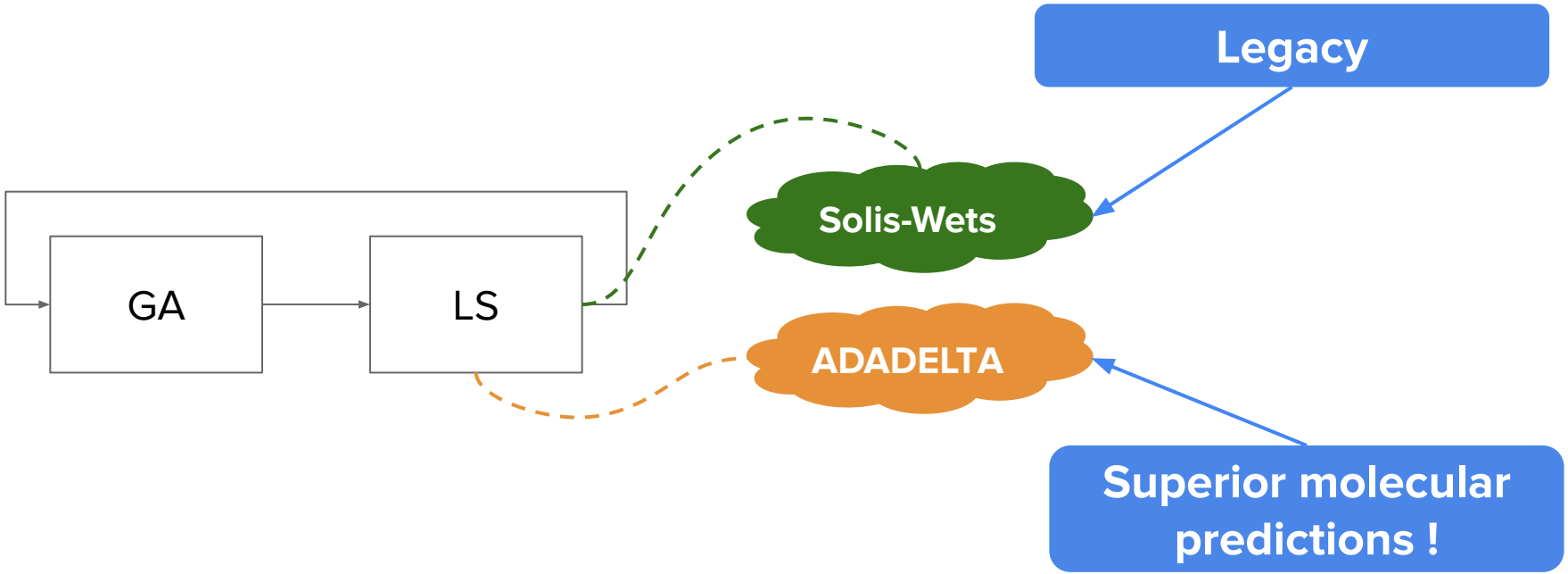
Lamarckian Genetic Algorithm (2/4)

Local Search (LS)

- Score refinement from GA poses
  - Alternative methods
    - Solis-Wets
    - ADADELTA

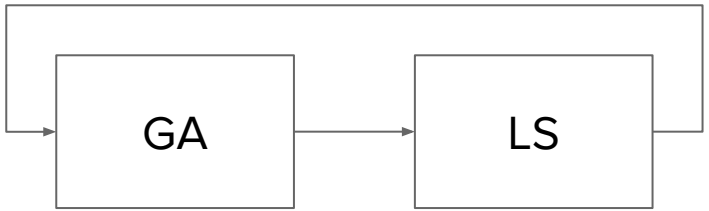


Lamarckian Genetic Algorithm (3/4)



Lamarckian Genetic Algorithm (4/4)

- AutoDock is compute bound
- Both GA and LS
  - compute-intensive score calculations
- LS
  - Driven by score optimization
  - > 90% total execution time



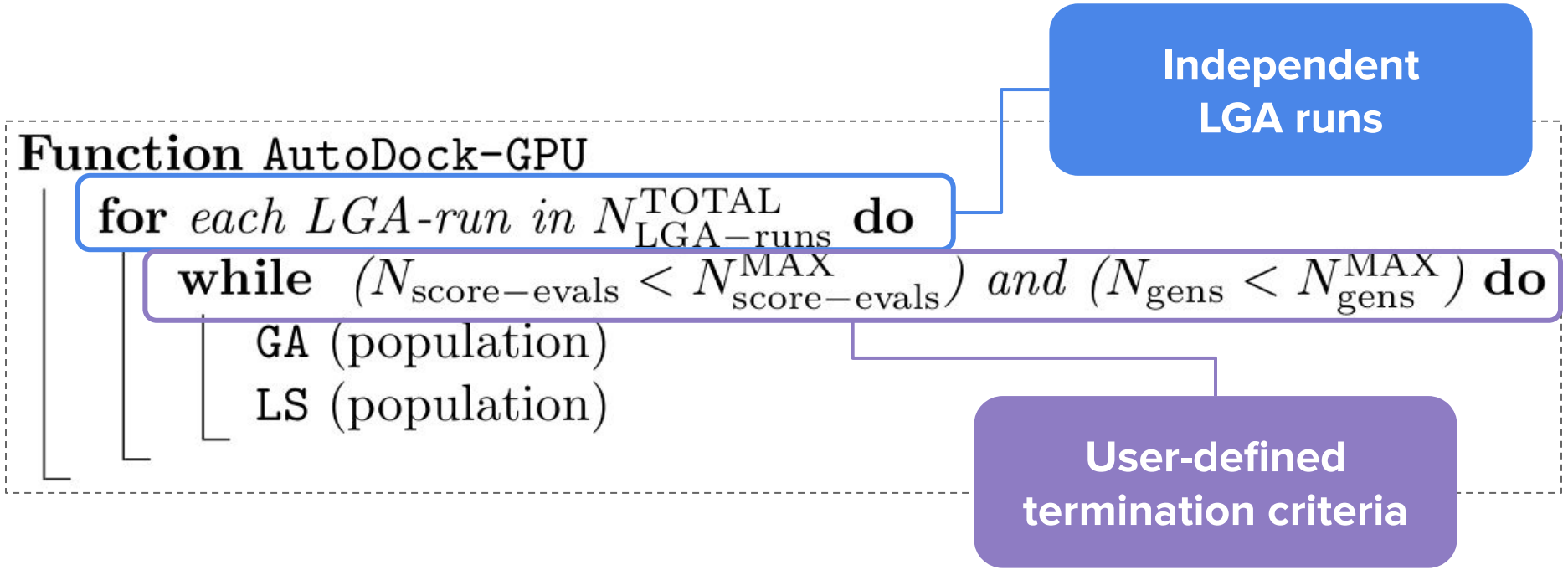
Algorithms

Lamarckian Genetic Algorithm

Function `AutoDock-GPU`

- **for** *each LGA-run in* $N^{\text{TOTAL}}_{\text{LGA-runs}}$ **do**
  - **while** $(N_{\text{score-evals}} < N^{\text{MAX}}_{\text{score-evals}})$ *and* $(N_{\text{gens}} < N^{\text{MAX}}_{\text{gens}})$ **do**
    - `GA` (population)
    - `LS` (population)

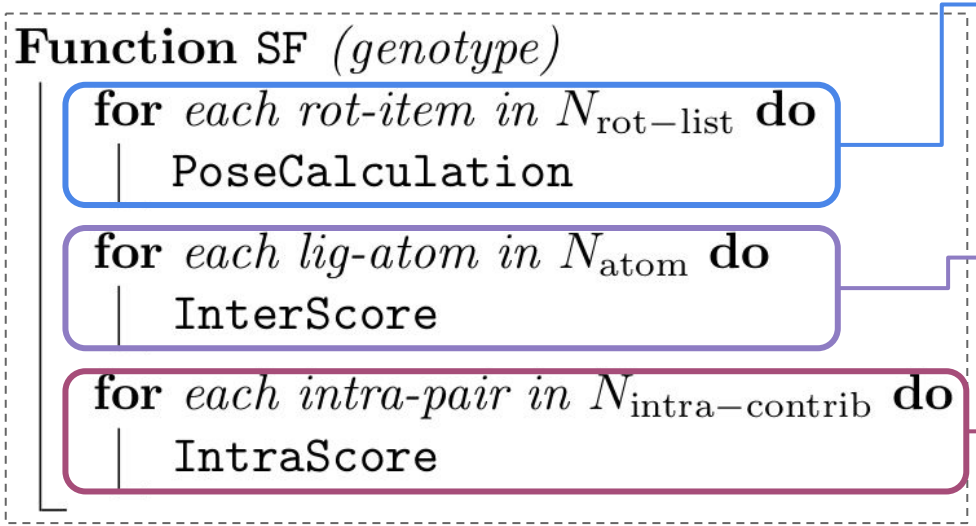
Lamarckian Genetic Algorithm



Scoring Function

```
Function SF (genotype)
    for each rot-item in N_rot−list do
        PoseCalculation
    for each lig-atom in N_atom do
        InterScore
    for each intra-pair in N_intra−contrib do
        IntraScore
```

Scoring Function

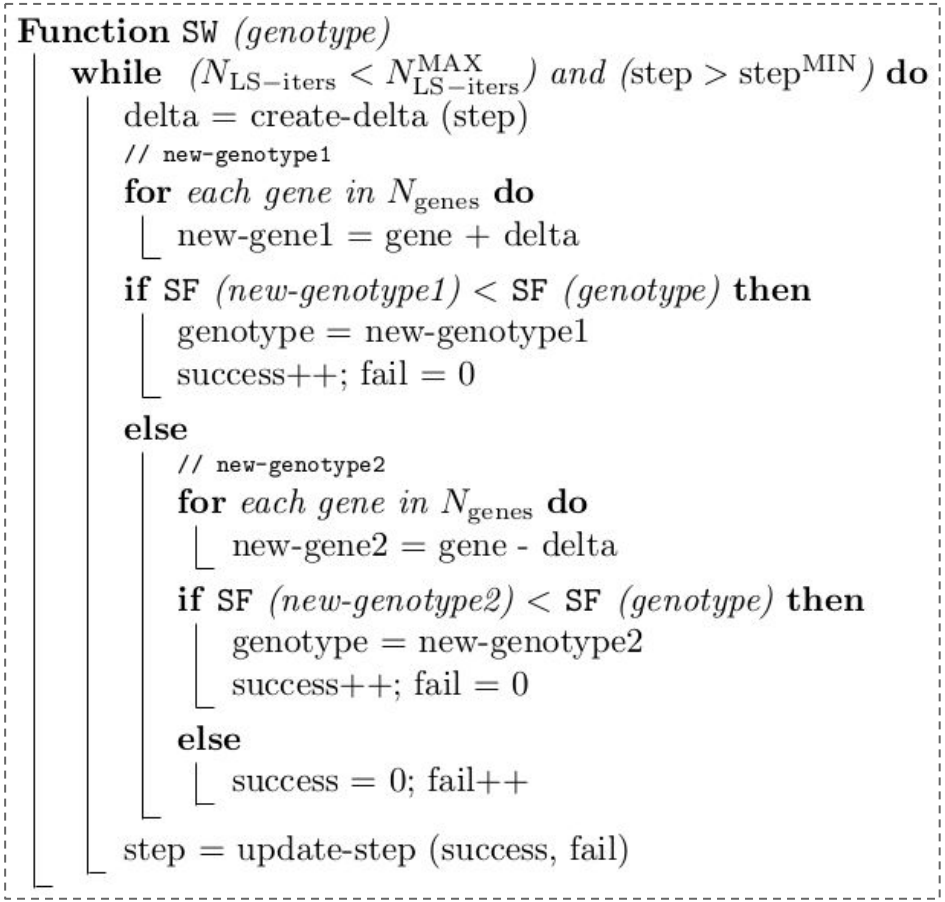


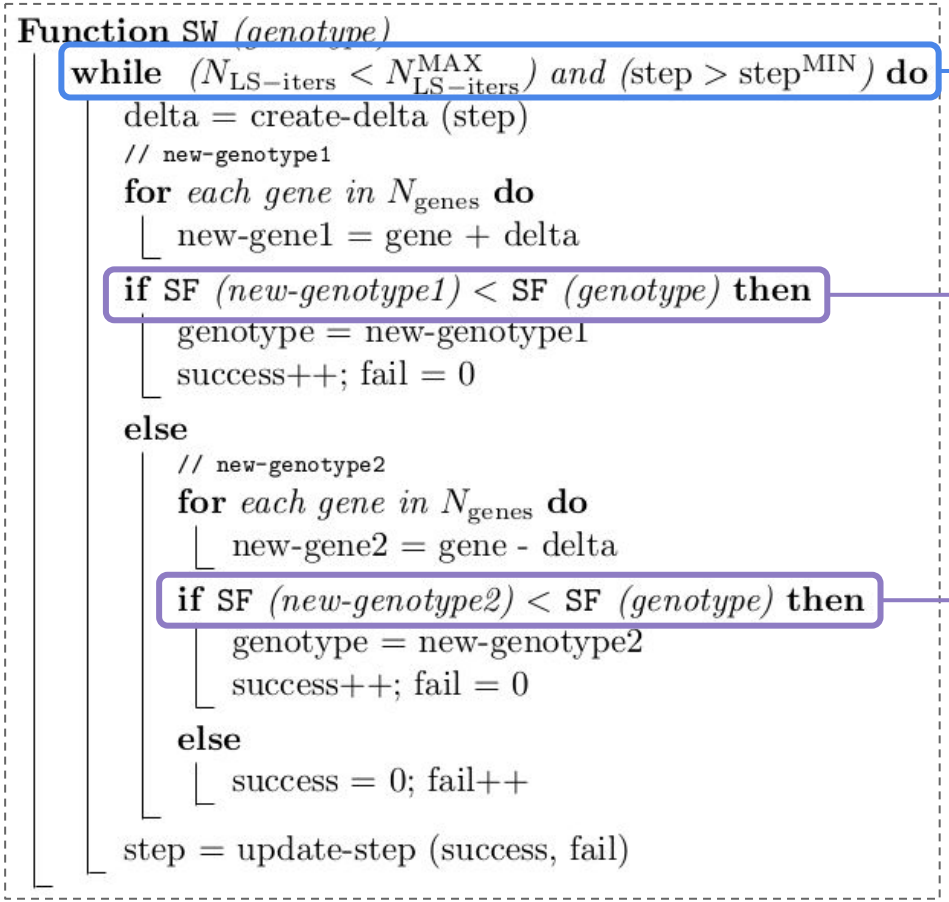
Calculating atomic coordinates

Receptor-Ligand score

Ligand-Ligand score

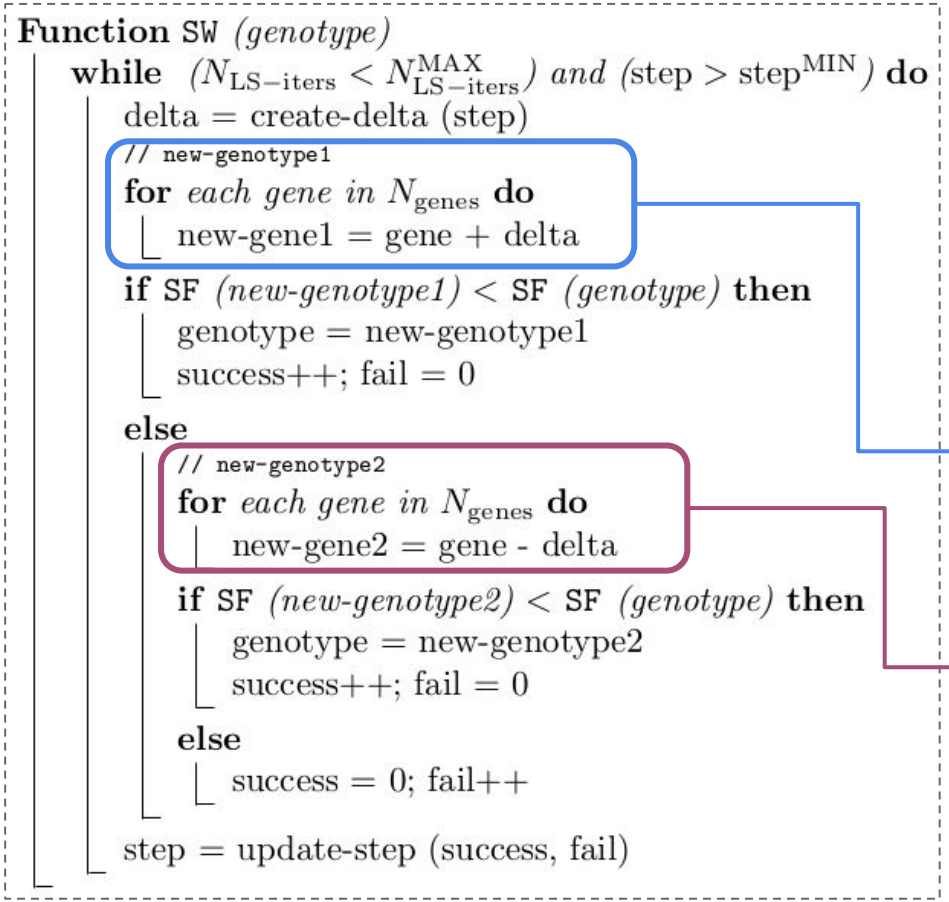
Solis Wets Local Search


Function SW (genotype)
while (N_LS−iters < N_LS−iters^MAX) and (step > step^MIN) do
delta = create-delta (step)
// new-genotype1
for each gene in N_genes do
new-gene1 = gene + delta
if SF (new-genotype1) < SF (genotype) then
genotype = new-genotype1
success++; fail = 0
else
// new-genotype2
for each gene in N_genes do
new-gene2 = gene - delta
if SF (new-genotype2) < SF (genotype) then
genotype = new-genotype2
success++; fail = 0
else
success = 0; fail++
step = update-step (success, fail)

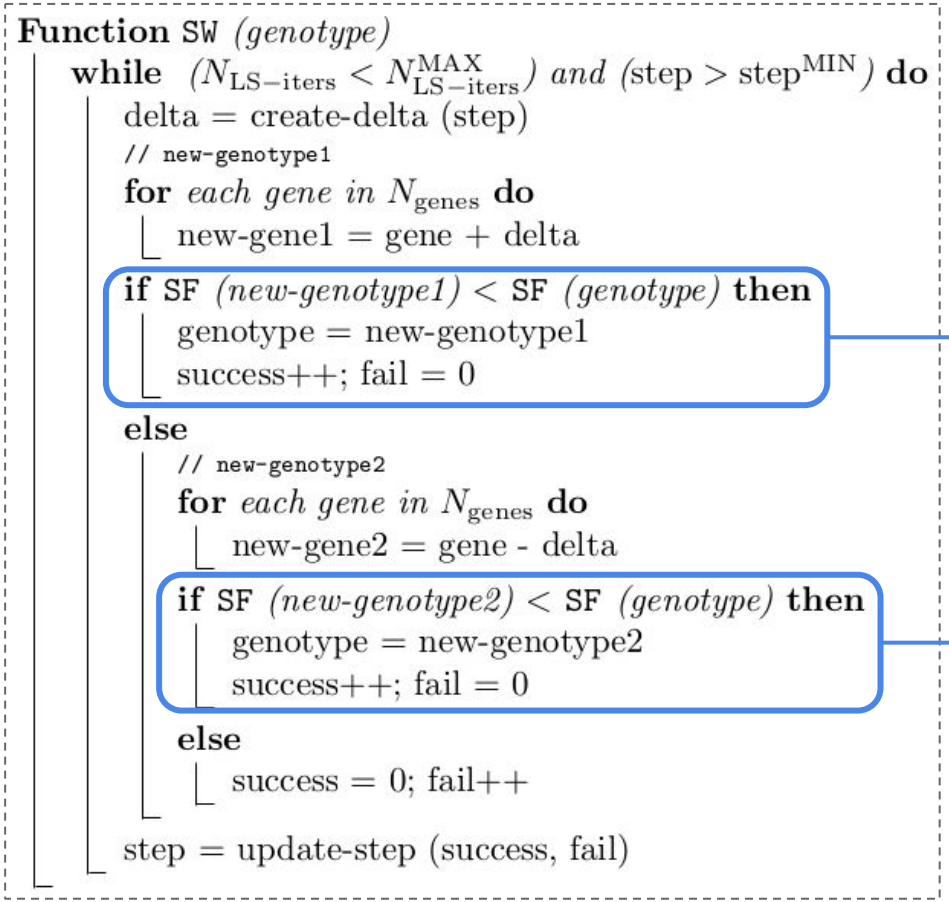


User-defined termination criteria

Time-intensive score evaluations

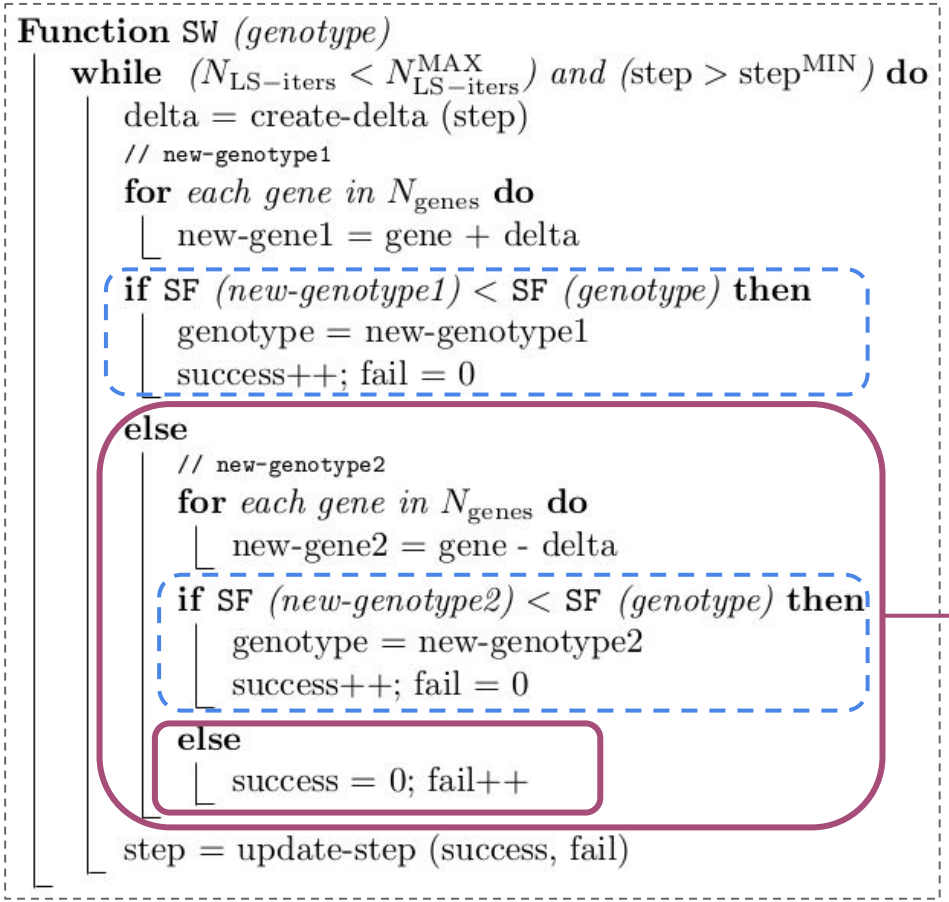


```
Function SW (genotype)
    while (N_LS−iters < N_LS−iters^MAX) and (step > step^MIN) do
        delta = create-delta (step)
        // new-genotype1
        for each gene in N_genes do
            new-gene1 = gene + delta
        if SF (new-genotype1) < SF (genotype) then
            genotype = new-genotype1
            success++; fail = 0
        else
            // new-genotype2
            for each gene in N_genes do
                new-gene2 = gene - delta
            if SF (new-genotype2) < SF (genotype) then
                genotype = new-genotype2
                success++; fail = 0
            else
                success = 0; fail++
        step = update-step (success, fail)
```



Divergent control

Score improves → SUCCESS



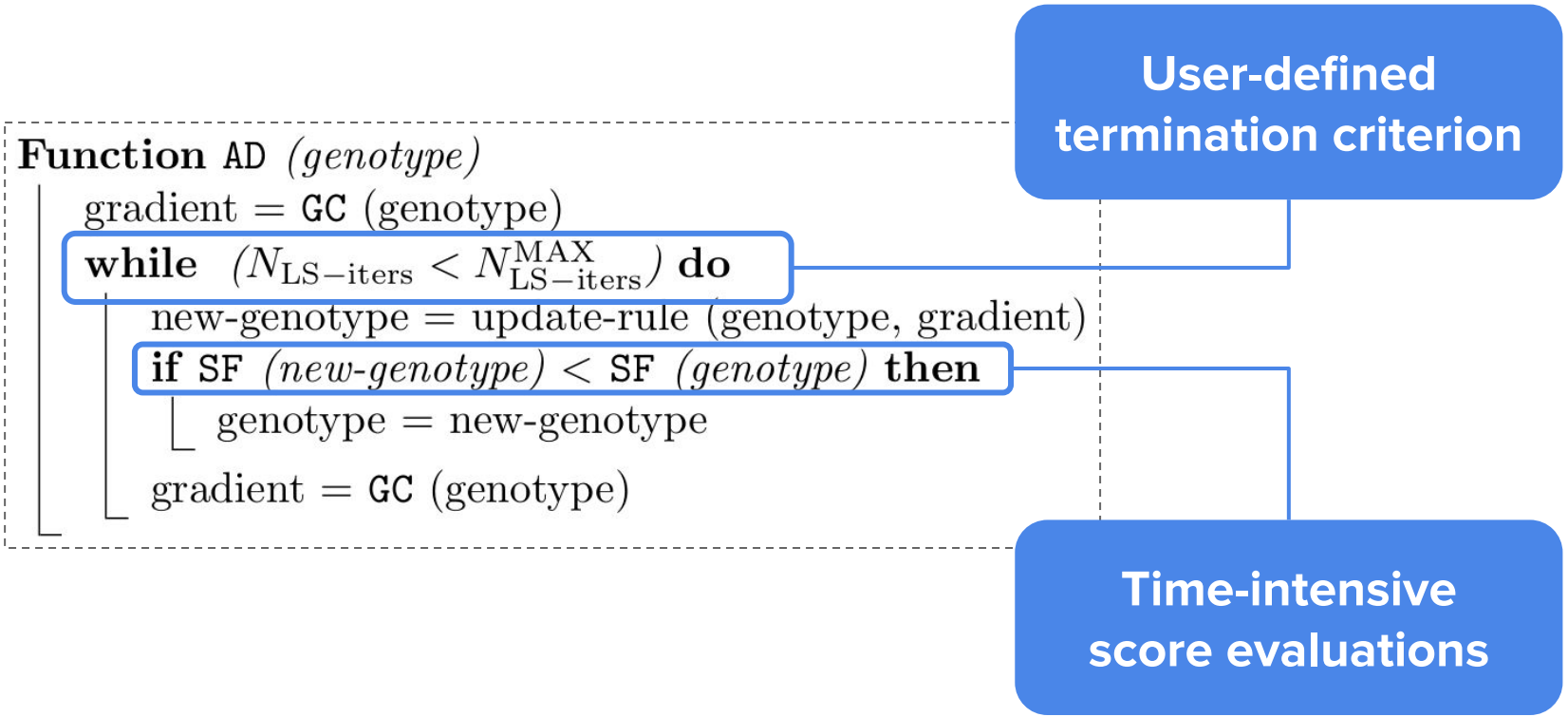
Divergent control

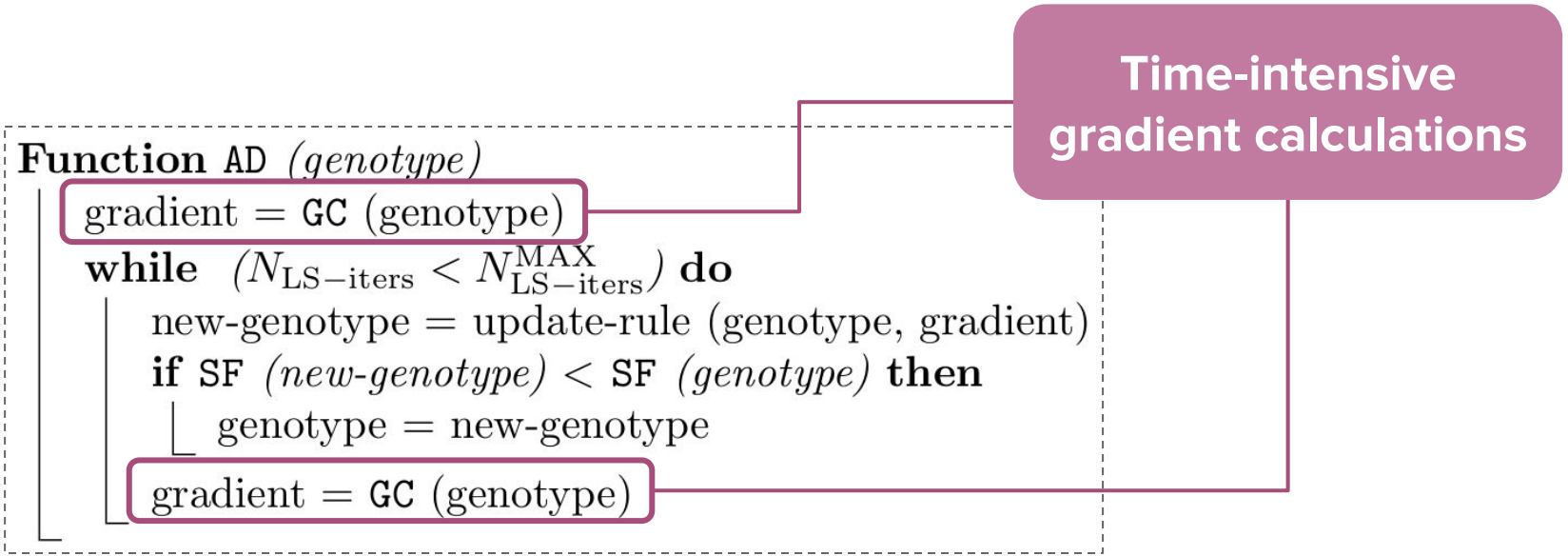
Score improves → SUCCESS

Score worsens → FAILURE

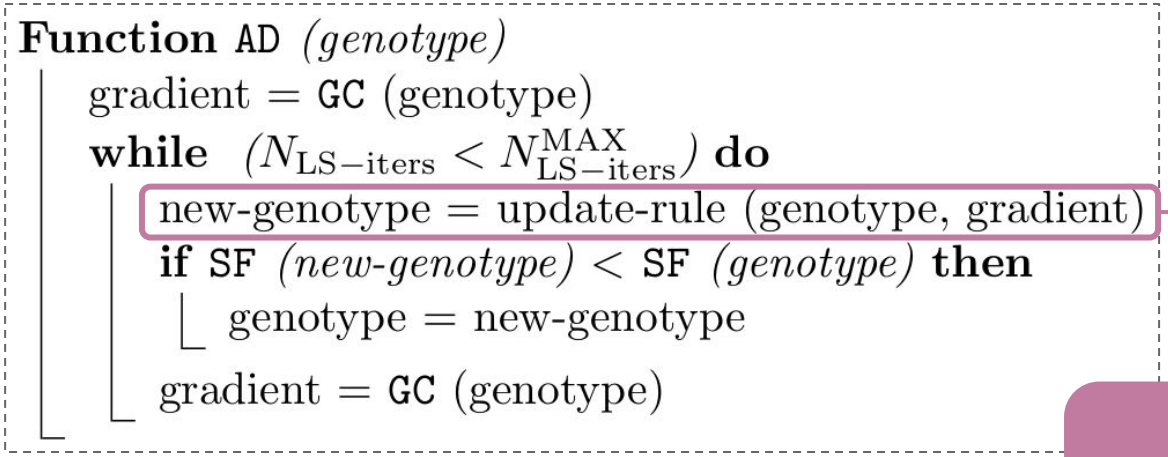
ADADELTA
Local Search

```
Function AD (genotype)
    gradient = GC (genotype)
    while (N_LS−iters < N^MAX_LS−iters) do
        new-genotype = update-rule (genotype, gradient)
        if SF (new-genotype) < SF (genotype) then
            genotype = new-genotype
        gradient = GC (genotype)
```


Function AD (genotype)
gradient = GC (genotype)
while ($N_{\mathrm{LS-iters}} < N^{\mathrm{MAX}}_{\mathrm{LS-iters}}$) do
new-genotype = update-rule (genotype, gradient)
if SF (new-genotype) < SF (genotype) then
genotype = new-genotype
gradient = GC (genotype)
User-defined termination criterion
Time-intensive score evaluations


Time-intensive gradient calculations
Function AD (genotype)
gradient = GC (genotype)
while ($N_{\text{LS-iters}} < N^{\text{MAX}}_{\text{LS-iters}}$) do
new-genotype = update-rule (genotype, gradient)
if SF (new-genotype) < SF (genotype) then
genotype = new-genotype
gradient = GC (genotype)

New genes



Gradients
(instead of +/- ops)
(more compute intense)

Development

Optimization

- Parallelization
- Vectorization
- Improving Vector-based Mapping
- Loop Pushing

Parallelization

Function `AutoDock-GPU`

for *each LGA-run in* $N_{\text{LGA-runs}}^{\text{TOTAL}}$ **do**

while $(N_{\text{score-evals}} < N_{\text{score-evals}}^{\text{MAX}})$ *and* $(N_{\text{gens}} < N_{\text{gens}}^{\text{MAX}})$ **do**

`GA` (population)

`LS` (population)

Parallelizing

Function `AutoDock-VE`

#pragma omp parallel for schedule (static, 1)

for *each LGA-run in* $N_{\text{LGA-runs}}^{\text{TOTAL}}$ **do**

while $(N_{\text{score-evals}} < N_{\text{score-evals}}^{\text{MAX}})$ *and* $(N_{\text{gens}} < N_{\text{gens}}^{\text{MAX}})$ **do**

`GA` (population)

`LS` (population)

Vectorization

- NEC compiler
  - Automatic vectorization

Vectorization

- NEC compiler
  - Automatic vectorization

- Pseudorandom number generator
  - Initially employed
    - Linear Congruential Generator: Xn+1 = f (Xn)
    - Dependence hinders vectorization
  - Replaced with
    - Built-in NEC Numeric Library Collection functions
    - Mersenne-Twister (vectorized)

Vectorization

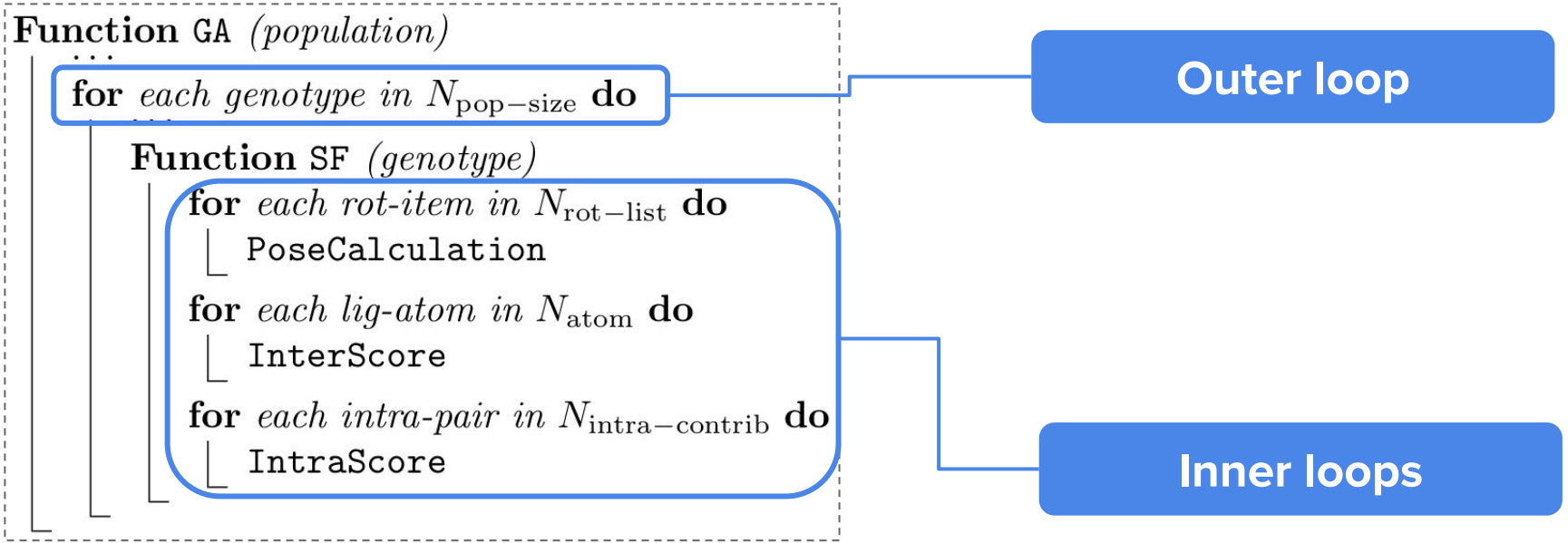
- NEC compiler
  - Automatic vectorization

- Pseudorandom number generator
  - Initially employed
    - Linear Congruential Generator: Xn+1 = f (Xn)
    - Dependence hinders vectorization
  - Replaced with
    - Built-in NEC Numeric Library Collection functions
    - Mersenne-Twister (vectorized)

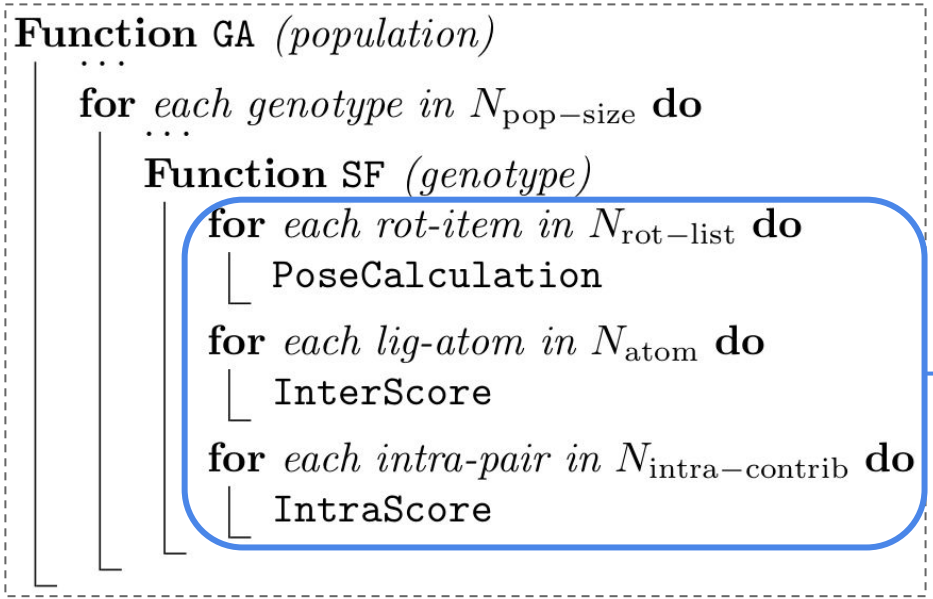
VE
2.2x slower
than host CPU !

What are the reasons for (this initial) low performance ?

How vector pipes are leveraged ?



How vector pipes are leveraged ?

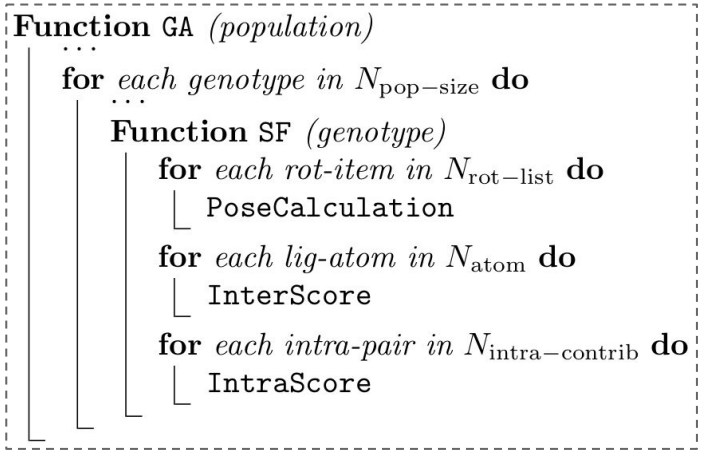


Vector pipes leveraged only by *innermost* loops !

Some are SHORT loops !

Inner loops

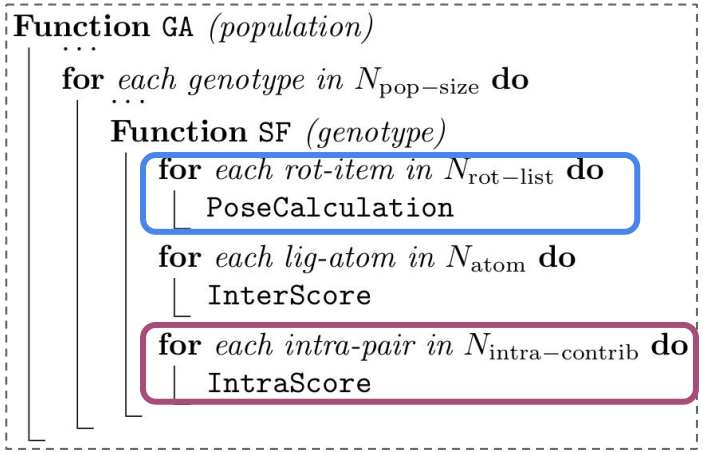
How vector pipes are leveraged ?



Inner loops' upper bounds

Input molecule	Nrot-list	Natom	Nintra_contrib
1u4d	23	23	0
1yv3	31	23	88
3er5	711	108	5,111

How vector pipes are leveraged ?

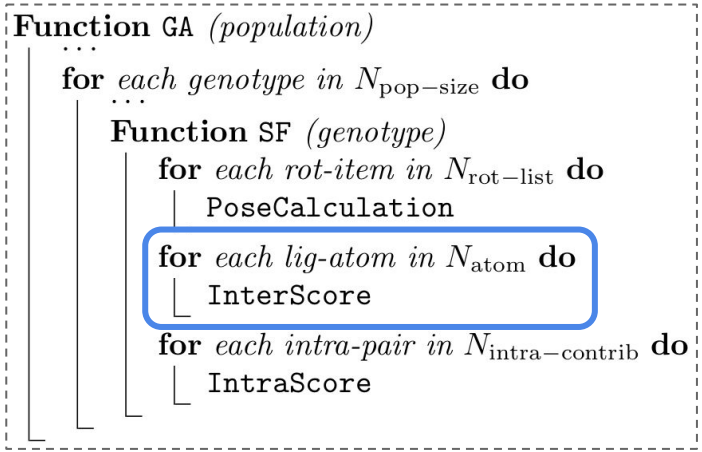


Max. vec. length (VE): 256

Input molecule	Nrot-list	Natom	Nintra_contrib
1u4d	23	23	0
1yv3	31	23	88
3er5	711	108	5,111

Large molecules can fill up the vector pipes

How vector pipes are leveraged ?



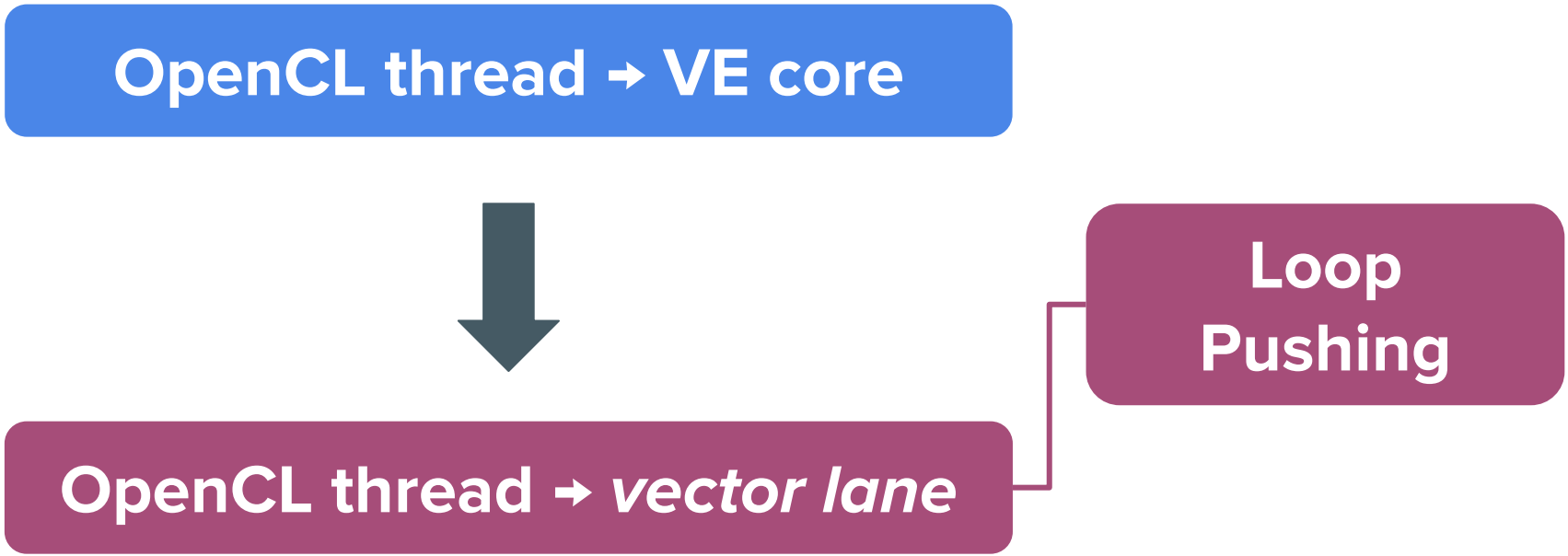
Max. vec. length (VE): 256

Input molecule	Nrot-list	Natom	Nintra_contrib
1u4d	23	23	0
1yv3	31	23	88
3er5	711	108	5,111

23 / 256 < 1/10 th

108 / 256 < 1/2

Improving Vector-based Mapping

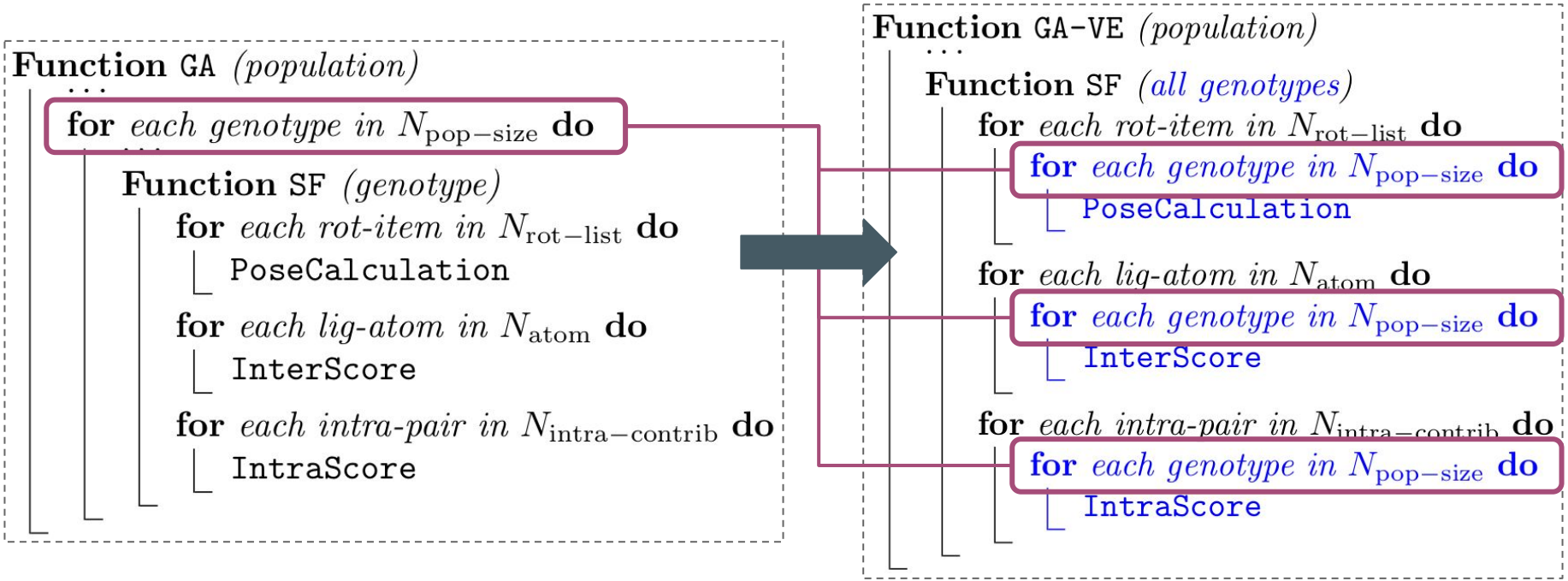


Loop Pushing in GA (1/3)

```
Function GA (population)
    ...
    for each genotype in N_pop-size do
        ...
        Function SF (genotype)
            for each rot-item in N_rot-list do
                PoseCalculation
            for each lig-atom in N_atom do
                InterScore
            for each intra-pair in N_intra-contrib do
                IntraScore
```

- Call to functions *obstructs* vectorization
- Loop length within function is inefficient

Loop Pushing in GA (2/3)



Loop Pushing in GA (3/3)

- This technique is paired with
  - Data layout changes
    - Unit-stride data accesses
    - E.g.: scalar ➔ arrays

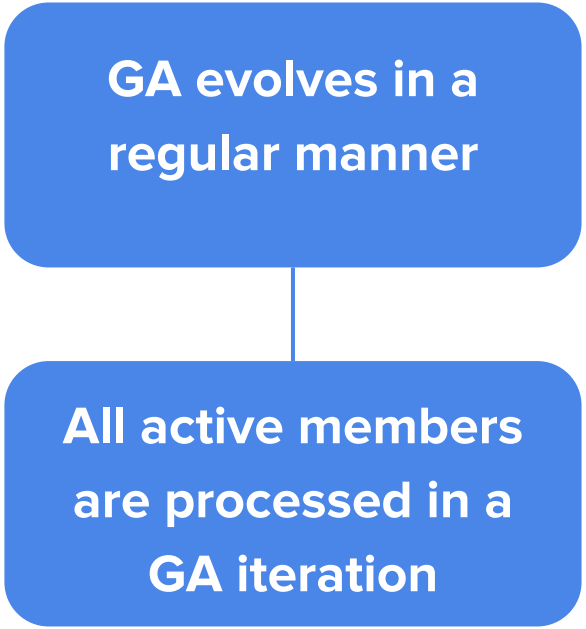
- (Initially outermost) pushed-in loop becomes
  - Innermost
  - Data parallel
  - Easily vectorizable

```
Function GA-VE (population)
    ...
    Function SF (all genotypes)
        for each rot-item in N_rot−list do
            for each genotype in N_pop−size do
                PoseCalculation
        for each lig-atom in N_atom do
            for each genotype in N_pop−size do
                InterScore
        for each intra-pair in N_intra−contrib do
            for each genotype in N_pop−size do
                IntraScore
```

Loop Pushing in LS (1/3)

- Same principle as for GA
  - However, requires *significant* adaptations

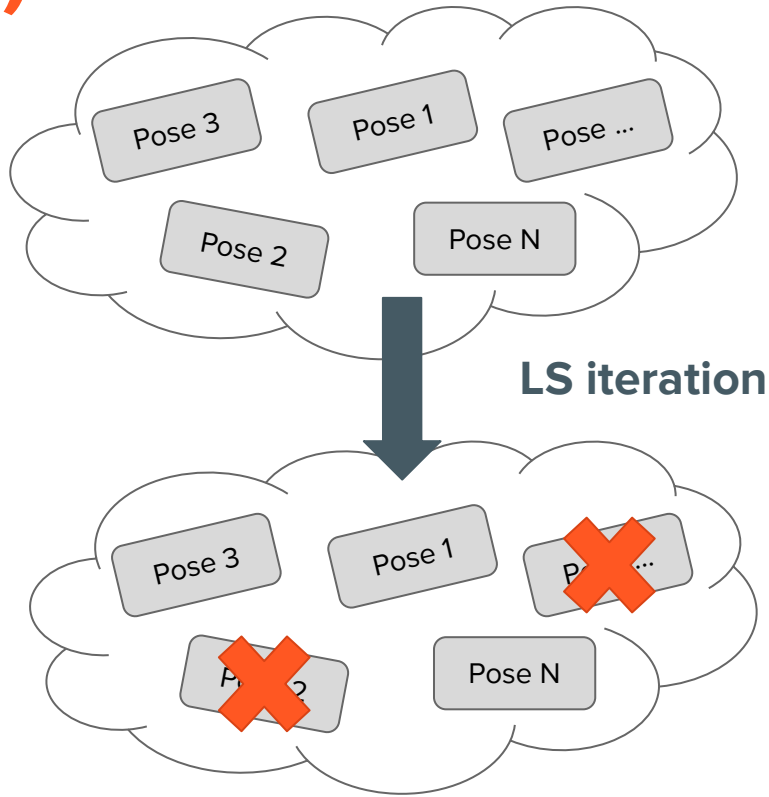
- Main difference
  - Populations in GA evolve *differently* than those in LS



Loop Pushing in LS (2/3)

- Populations in LS
  - Processed by divergent algorithms
  - Some members achieve convergence earlier than others

Already-converged members are removed from the computation



Loop Pushing in LS (3/3)

- For the non-convergent part of the population
  - Loop compression
  - Predication
- Aims to keep the score and gradient calculations …
  - … with unit-stride data accesses
  - … without additional predication

Loop compression and predication

[Solis-Wets]

Loop Compression

Solis-Wets (original)

```
if SF (new-genotype1) < SF (genotype) then
    genotype = new-genotype1
    success++; fail = 0
else
    // new-genotype2
    for each gene in N_genes do
        new-gene2 = gene - delta
    if SF (new-genotype2) < SF (genotype) then
        genotype = new-genotype2
        success++; fail = 0
    else
        success = 0; fail++
```

Solis-Wets (with loop pushing)

```
while N_LS-iters^active > 0 do
    // Building compressed list of active indexes
    pop_size^active = 0
    for each j in N_pop-size do
        if LS^active[j] then
            idx^active[pop_size^active] = j
            N_LS-iters^compressed[pop_size^active] = N_LS-iters[j]
            success^compressed[pop_size^active] = success[j]
            pop_size^active++
            ...
```

Loop Compression

Scalar → array

Predication

Solis-Wets (original)

```
Function SW (genotype)
    while (N_LS-iters < N_LS-iters^MAX) and (step > step^MIN) do
        delta = create-delta (step)
        // new-genotype1
        for each gene in N_genes do
            new-gene1 = gene + delta
        if SF (new-genotype1) < SF (genotype) then
            genotype = new-genotype1
            success++; fail = 0
```

Solis-Wets (with loop pushing)

```
// Predicating on termination condition
N_LS-iters^active = pop_size^active
for each jj in pop_size^active do
    if (N_LS-iters^compressed[jj] > N_LS-iters^MAX) or (step^compressed[jj] <=
     step^MIN) then
        LS^active[idx^active[jj]] = 0
        N_LS-iters^active--
    j = idx^active[jj]
    N_LS-iters[j] = N_LS-iters^compressed[jj]
    step[j] = step^compressed[jj]
    success[j] = success^compressed[jj]
```

Predication

To update the number of active members in LS

Evaluation

Performance Profiling

- Impact of Loop Pushing [Solis-Wets]

Impact of Loop Pushing (1/3) [Solis-Wets]

			Input molecule: 1hfs
Metric	**Before**	**After**	**Improvement**
Real Time [sec]	1,382.2	40.0	~34x
Vector Time [sec]	2,217.6	280.2	~8x

Impact of Loop Pushing (2/3) [Solis-Wets]

Input molecule: 1hfs			
Metric	**Before**	**After**	**Improvement**
MOPS	8,348.6	185,805.3	~22x
MFLOPS	3,556.7	128,005.2	~36x

Impact of Loop Pushing (3/3) [Solis-Wets]

Input molecule: 1hfs		
Metric	**Before**	**After**
Avg. Vector Length	195.4	214.0 (Optimal: 256)
Vector Operation Ratio [%]	75.3	99.4 (Optimal: 100%)

Comparison vs. GPUs and CPUs

- Impact of Population Size
- Best Results

Hardware Devices

SX-Aurora TSUBASA	GPU		CPU
VE 20B	V100	A100	EPYC 7713 (2 x 64 cores)

Device Characteristics (1/2)

	SX-Aurora TSUBASA	GPU		CPU
	VE 20B	V100	A100	EPYC 7713
Process Size [nm]	16	12	7	7
Transistor Density [billions/mm^2]	0.009	0.025	0.065	unknown

Device Characteristics (1/2)

	SX-Aurora TSUBASA	GPU	
	VE 20B	V100	A100
Process Size [nm]	16	12	7
Transistor Density [billions/mm^2]	0.009	0.025	0.065

Wrt. VE:

V100: 2.7x
A100: 7.2x

Higher transistor density

Device Characteristics (2/2)

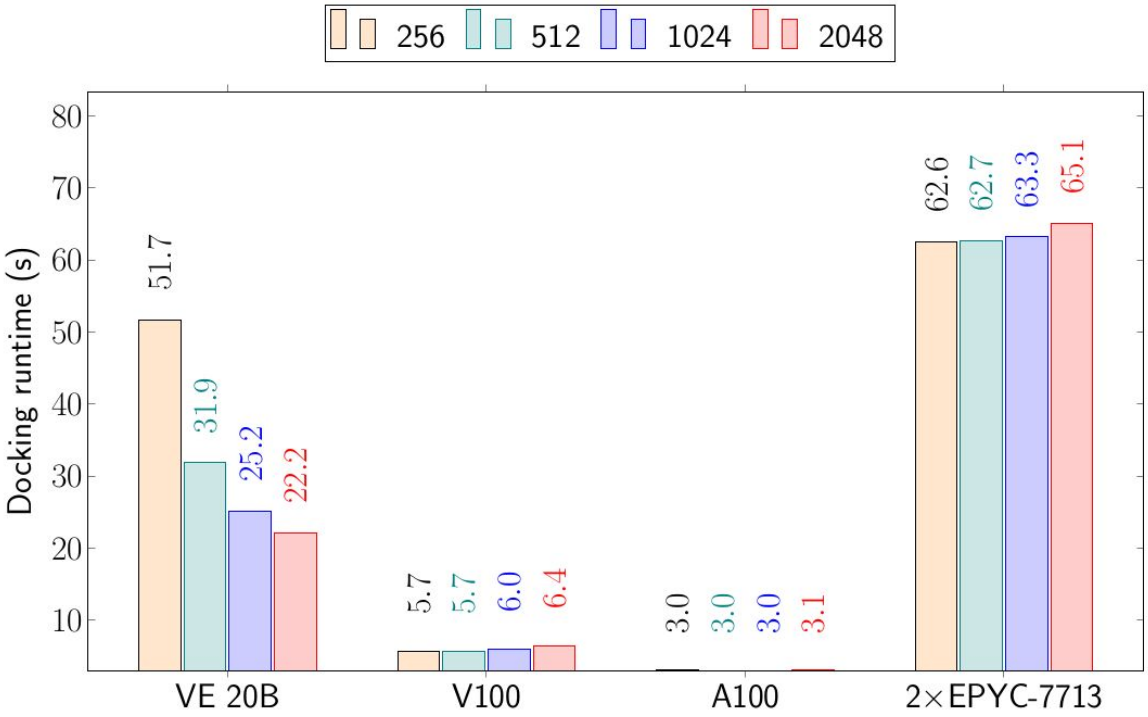
	SX-Aurora TSUBASA	GPU		CPU
	VE 20B	V100	A100	EPYC 7713
Process Size [nm]	16	12	7	7
Transistor Density [billions/mm^2]	0.009	0.025	0.065	unknown
Perf [TFLOPS]	4.9	14.1	19.5	4.1
BW [GB/s]	1530	897	1555	409.6

Device Characteristics (2/2)

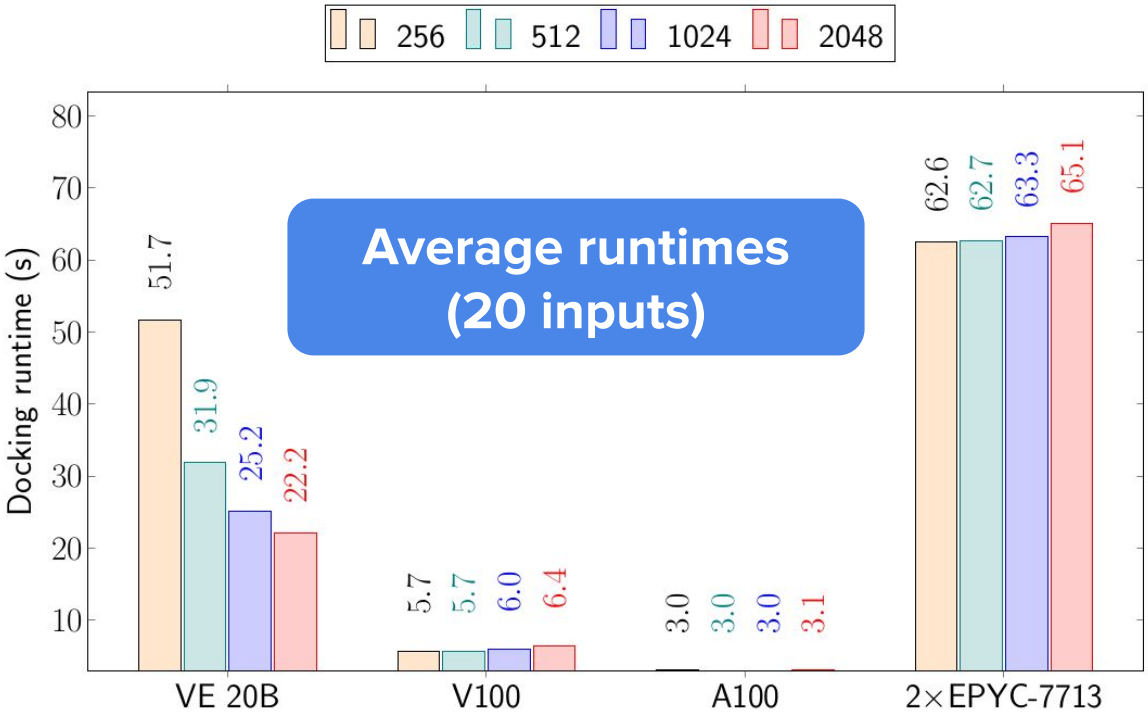
	SX-Aurora TSUBASA	GPU		CPU
	VE 20B			
Process Size [nm]	16			
Transistor Density [billions/mm^2]	0.009			
Perf [TFLOPS]	4.9	14.1	19.5	4.1
BW [GB/s]	1530	897	1555	409.6

VE's main strength

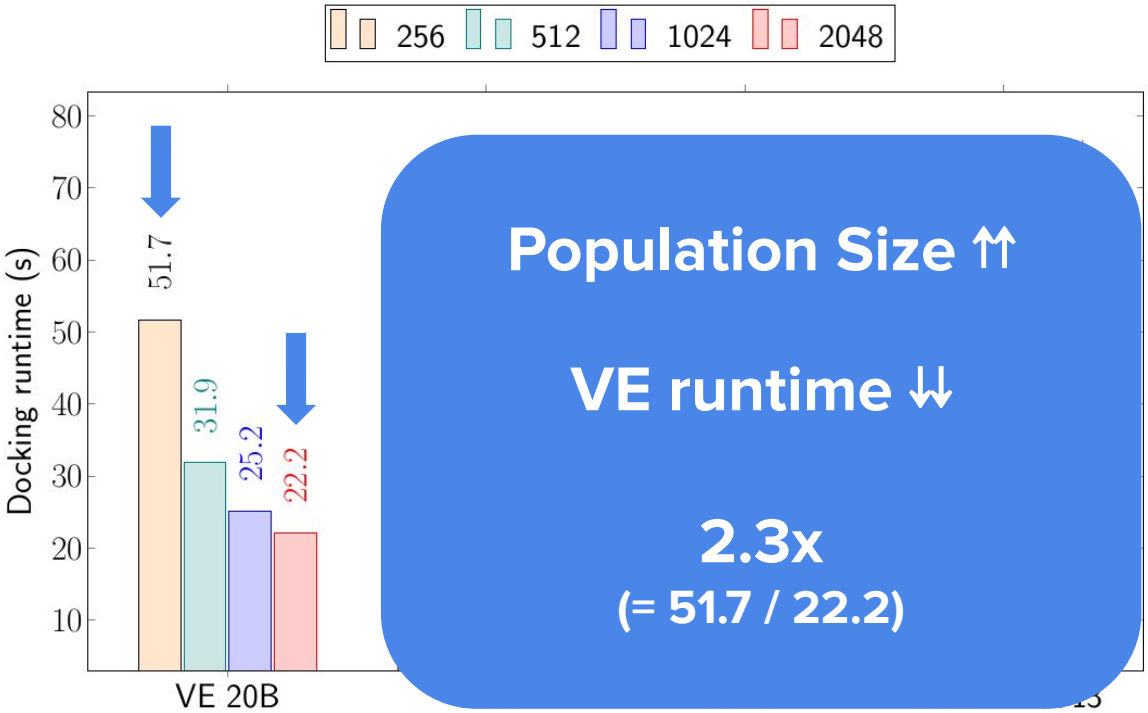
Impact of Population Size [Solis-Wets]



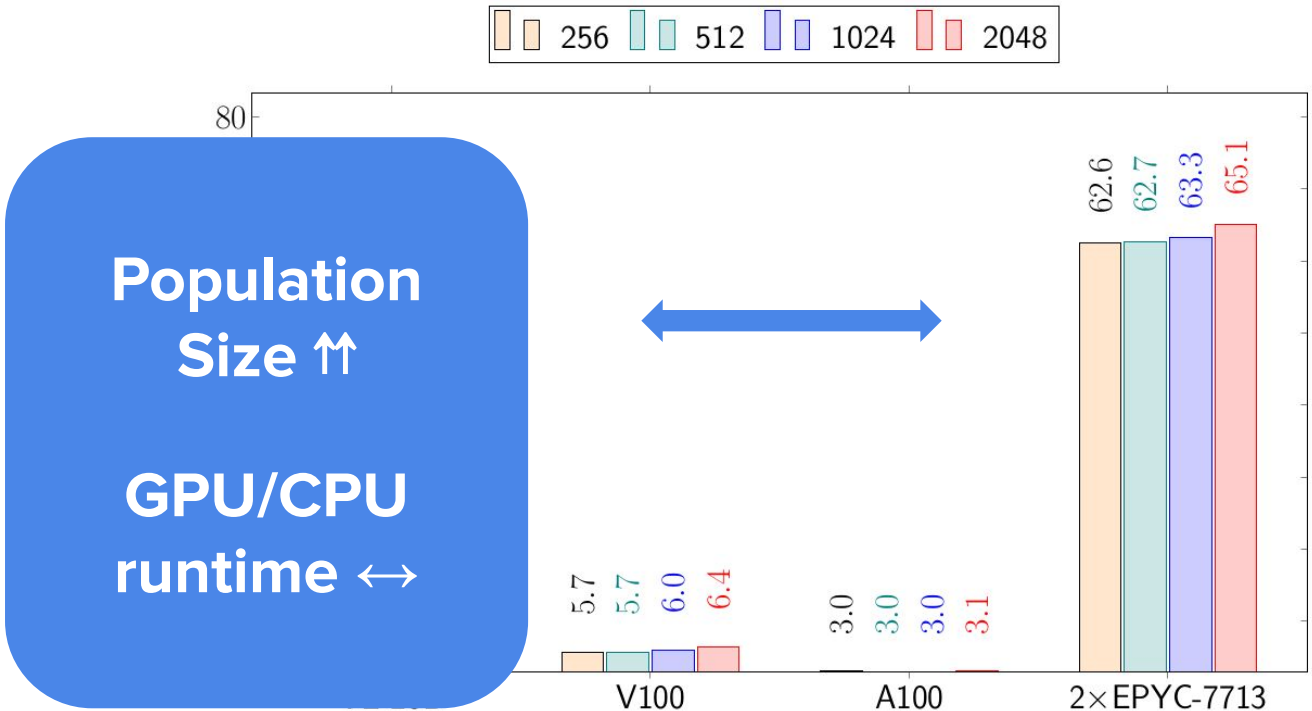
Impact of Population Size [Solis-Wets]



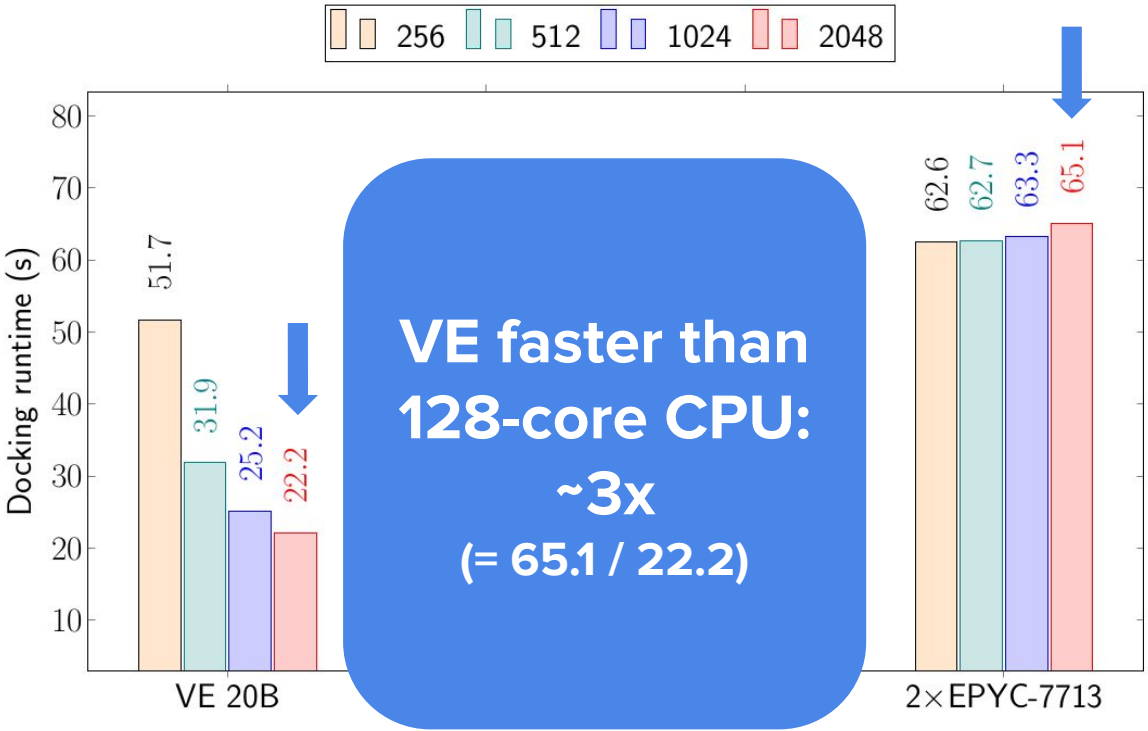
Impact of Population Size [Solis-Wets]



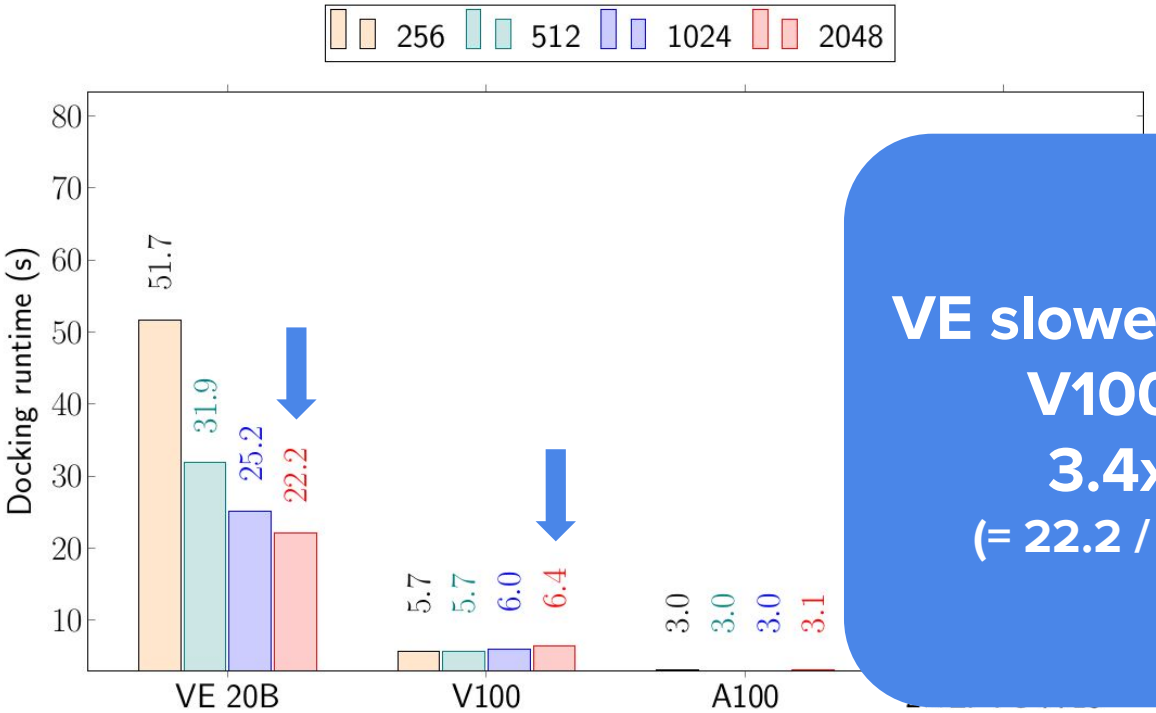
Impact of Population Size [Solis-Wets]



Impact of Population Size [Solis-Wets]

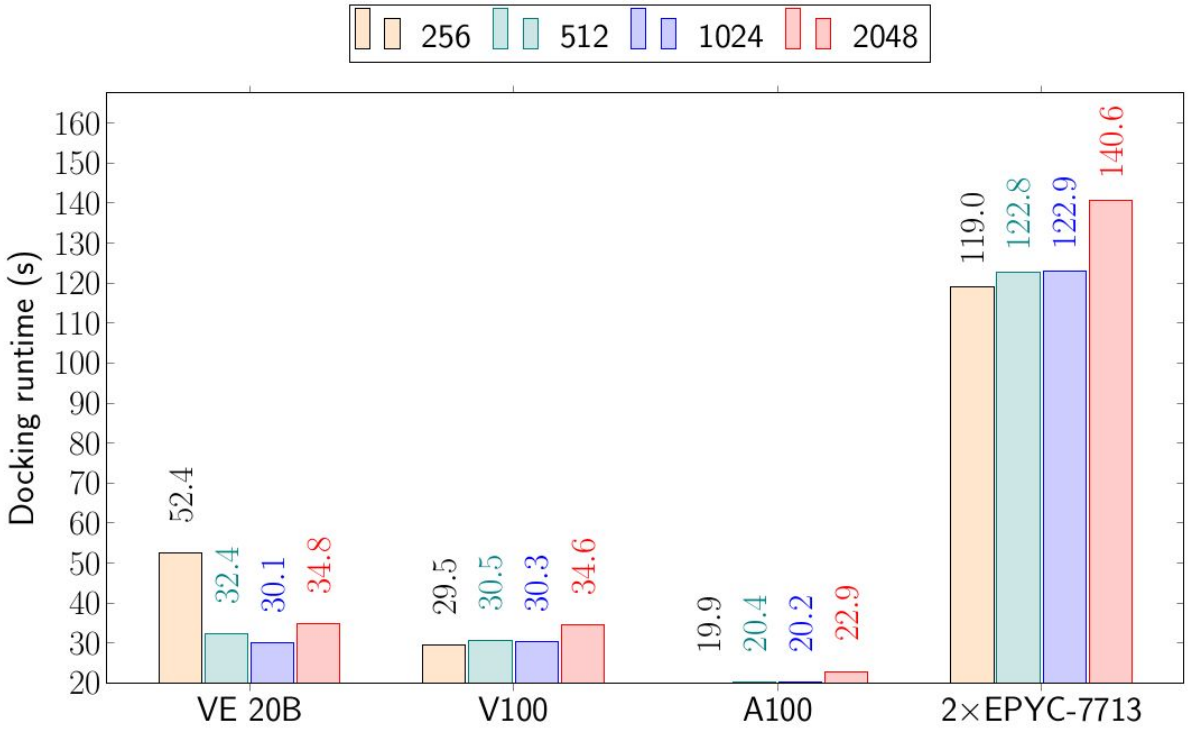


Impact of Population Size [Solis-Wets]

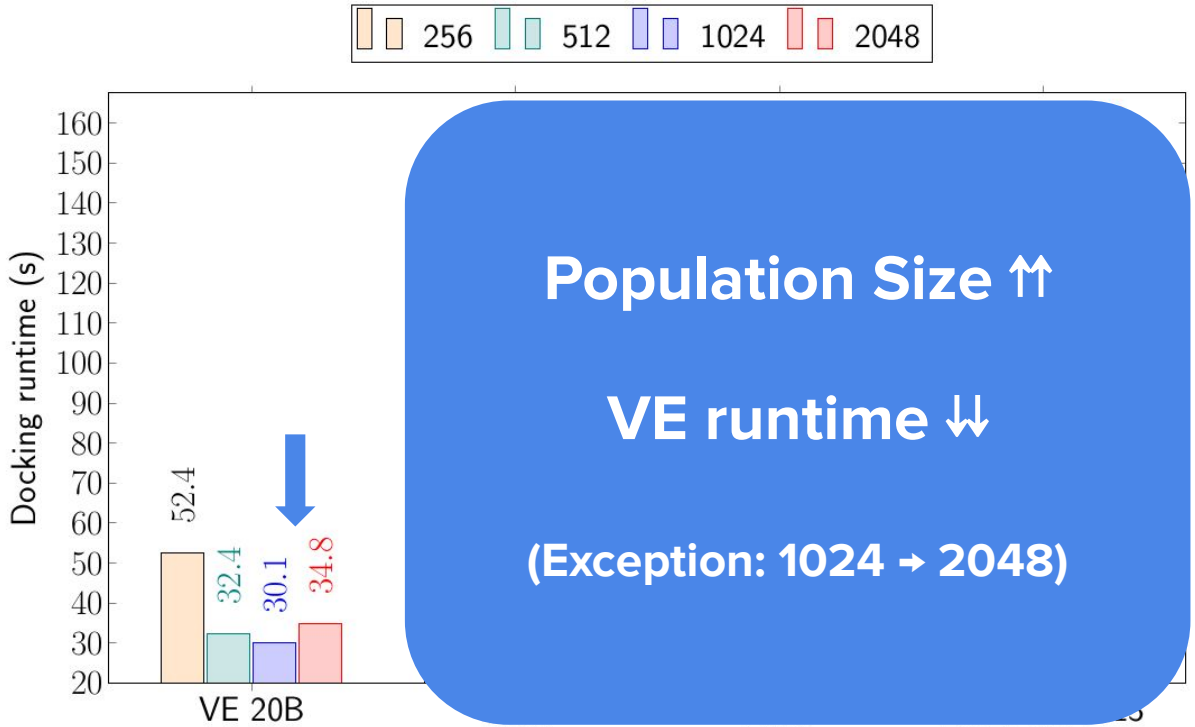


VE slower than V100: 3.4x
(= 22.2 / 6.4)

Impact of Population Size [ADADELTA]



Impact of Population Size [ADADELTA]

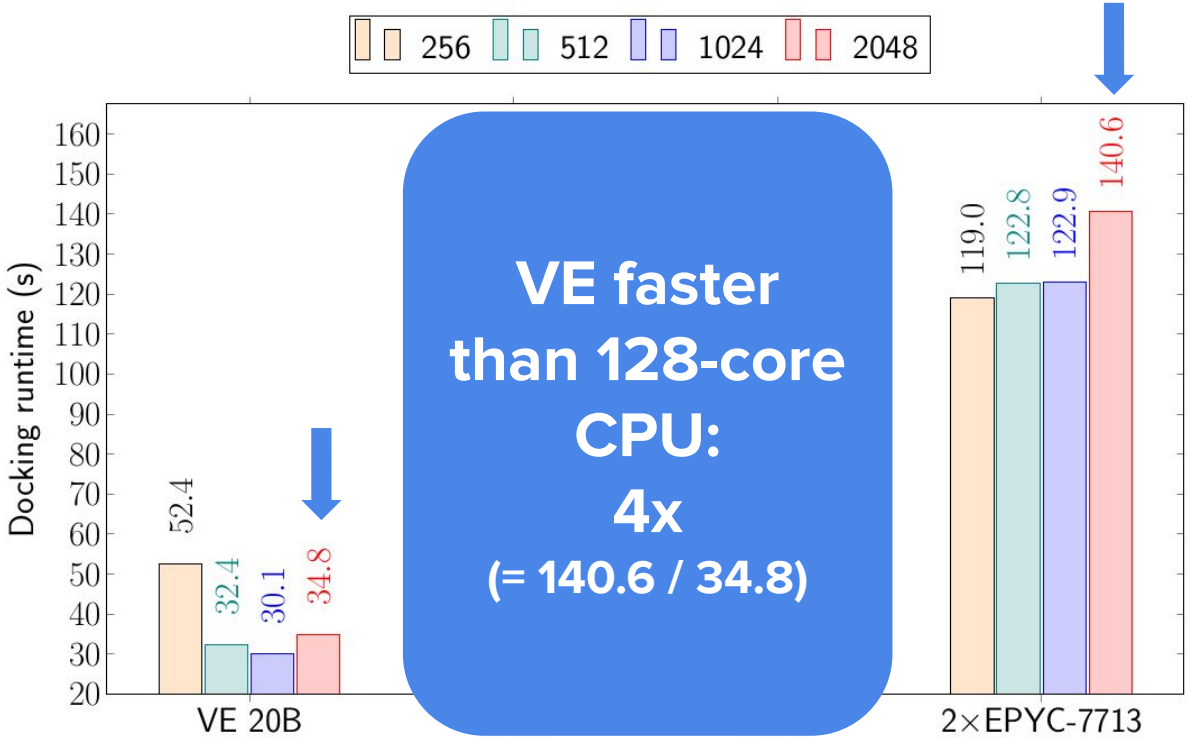


Population Size ↑↑

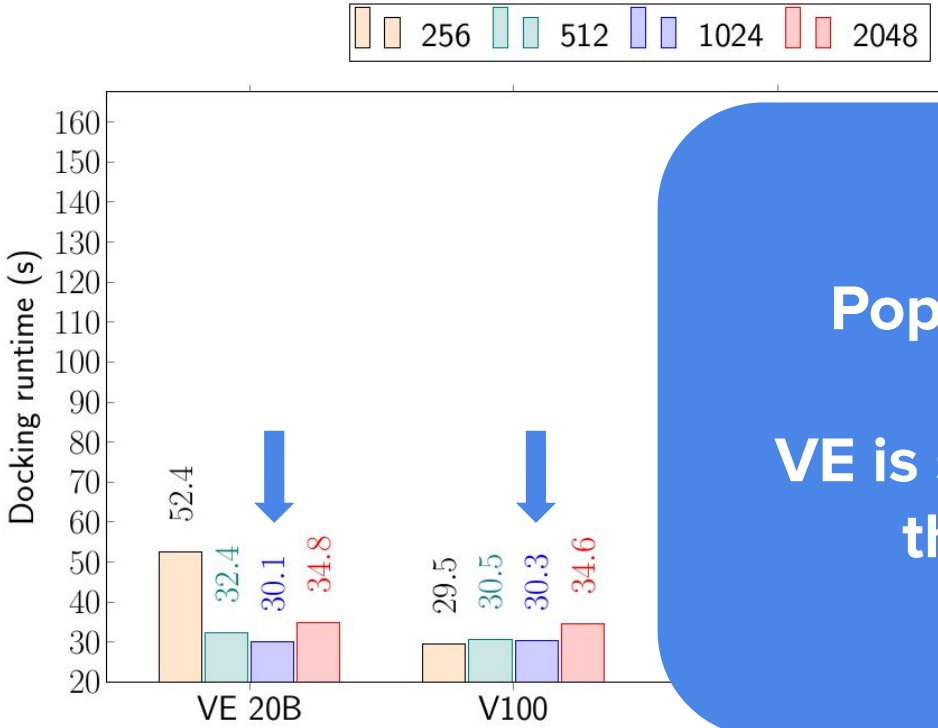
VE runtime ↓↓

(Exception: 1024 → 2048)

Impact of Population Size [ADADELTA]



Impact of Population Size [ADADELTA]

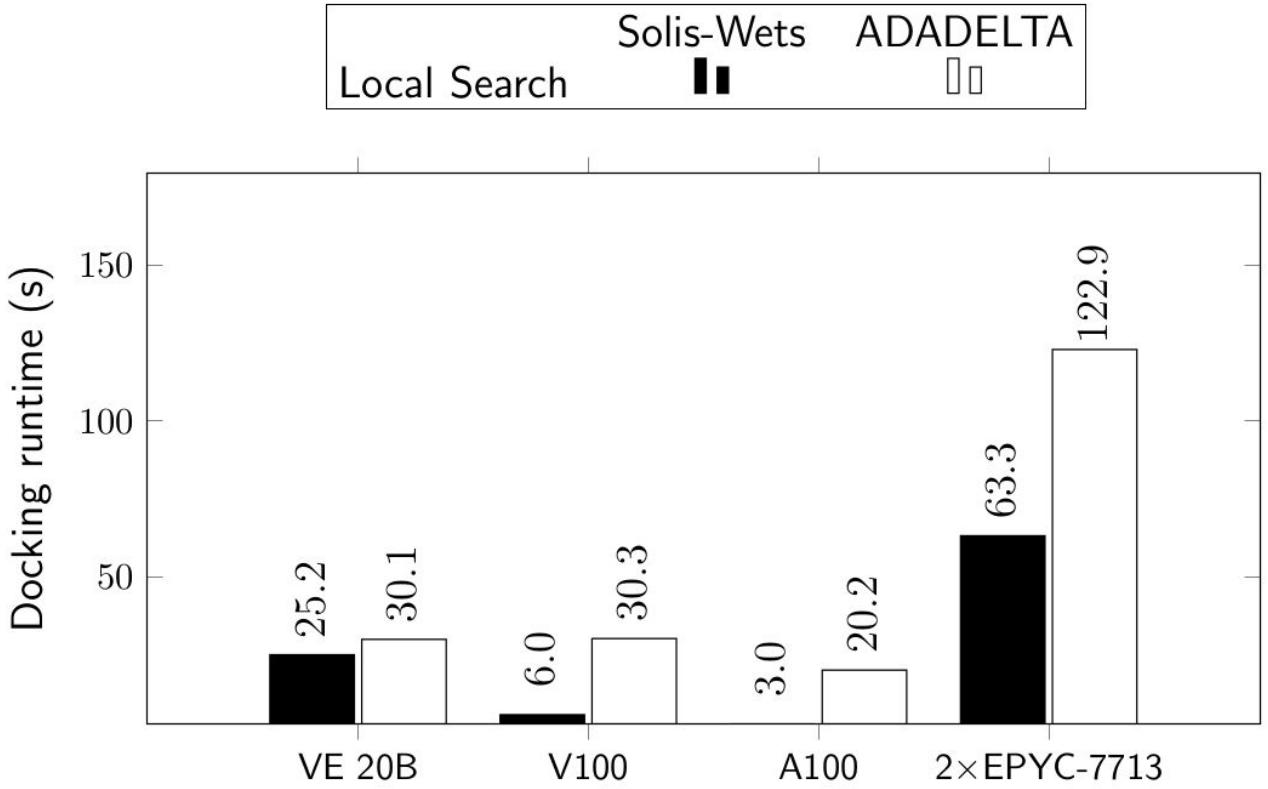


Pop. Size: 1024

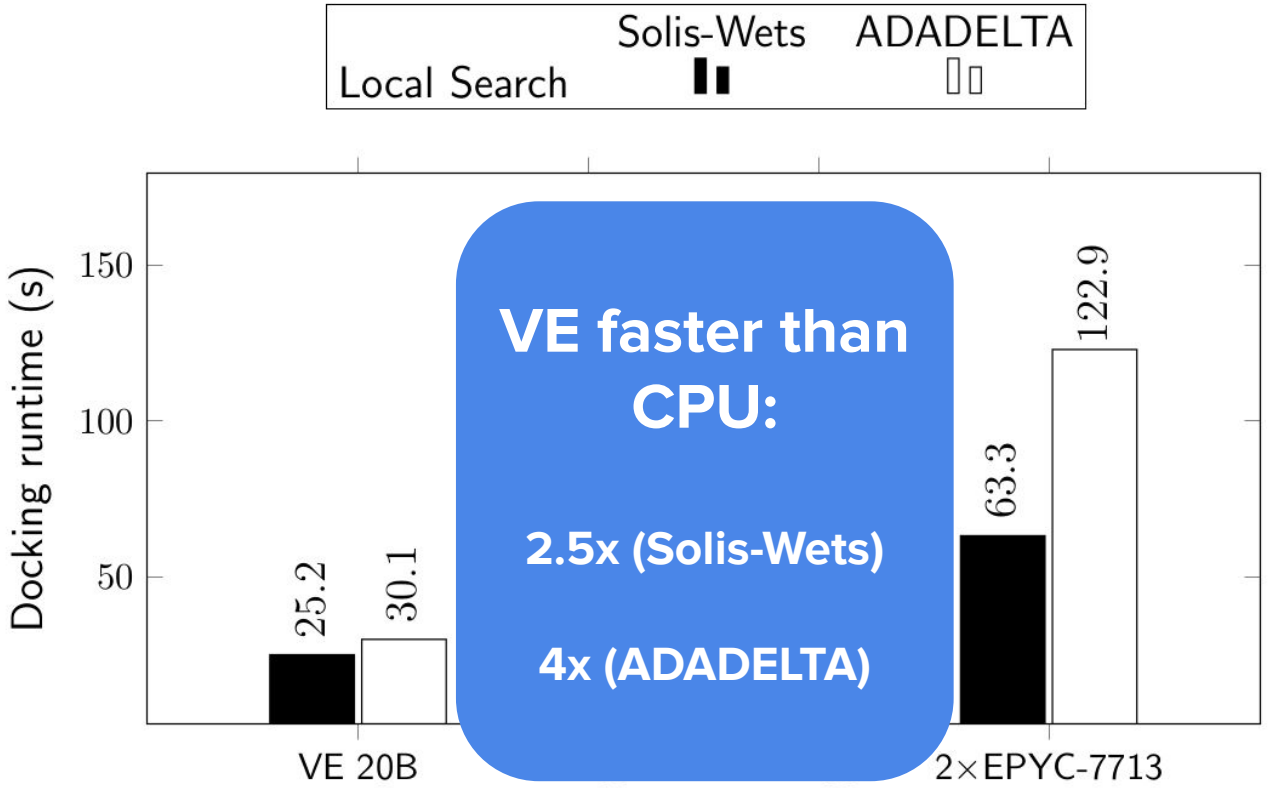
VE is slightly faster than V100

What are the best results achieved on the VE ?

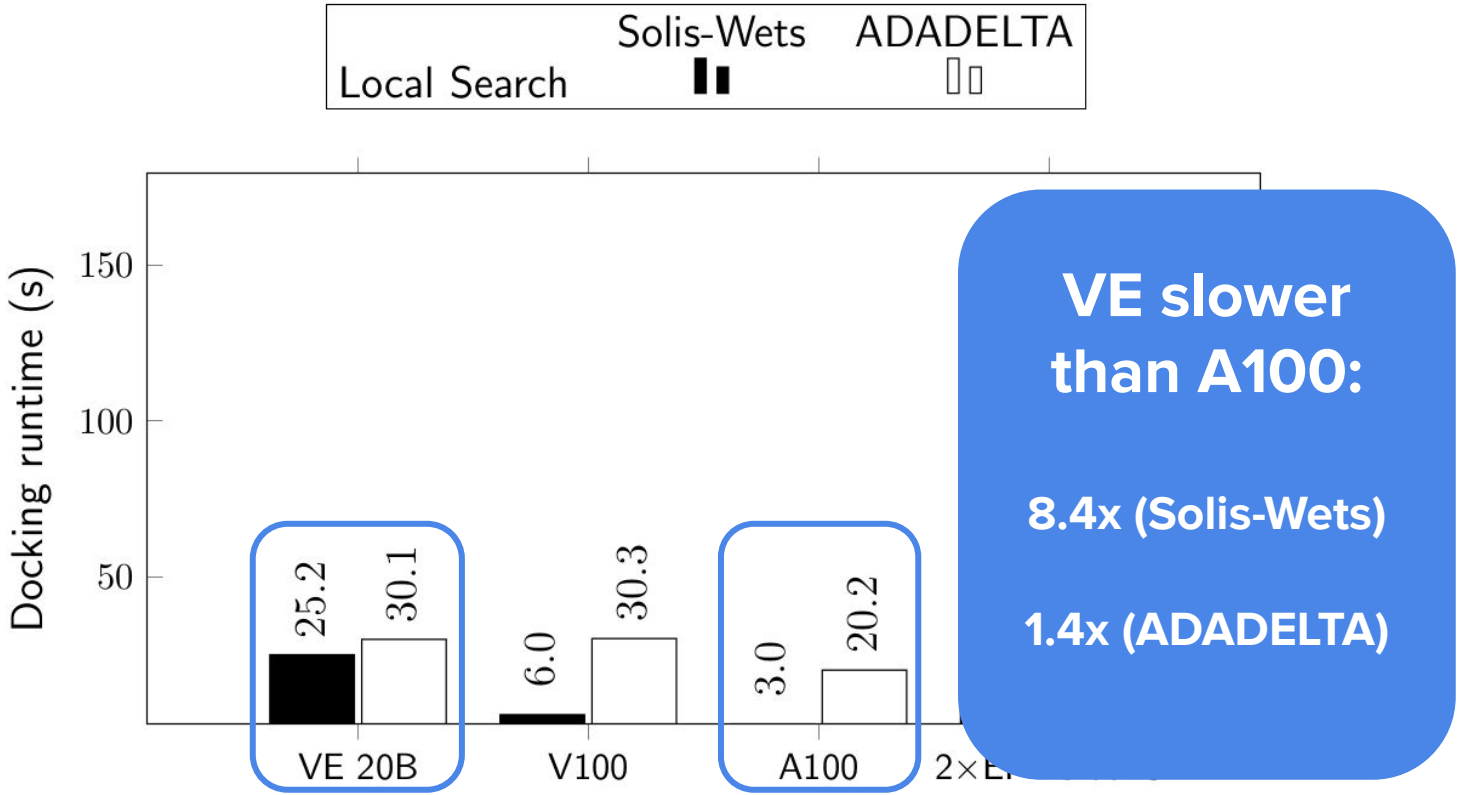
Population Size: 1024



Population Size: 1024



Population Size: 1024

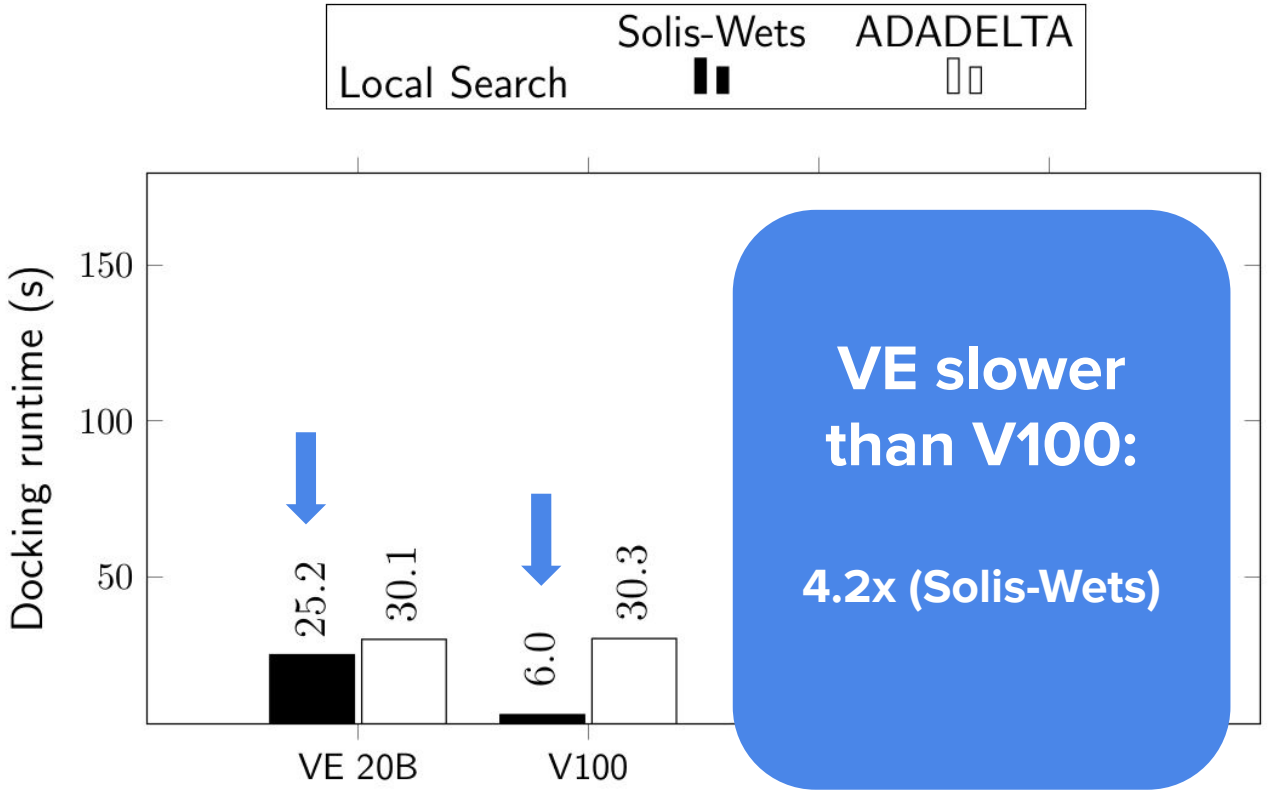


VE slower than A100:

8.4x (Solis-Wets)

1.4x (ADADELTA)

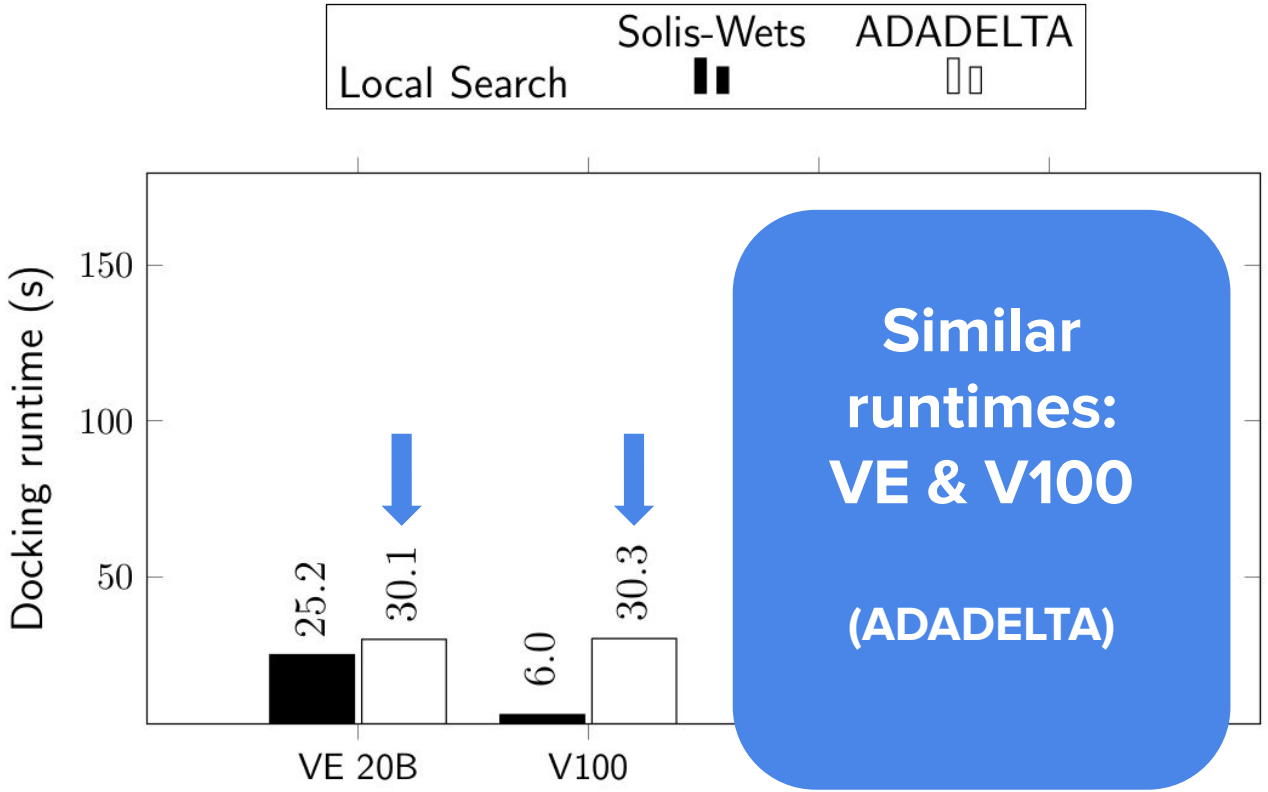
Population Size: 1024



VE slower than V100:

4.2x (Solis-Wets)

Population Size: 1024



Concluding Remarks

Summary

- AutoDock-Aurora
  - A port of AutoDock to the SX-Aurora TSUBASA
- LGA = GA + LS
  - Genetic Algorithm
  - Local Search
- Local Search
  - Bottleneck in AutoDock
  - Highly irregular
  - Available methods
    - Solis-Wets
    - ADADELTA

Conclusions

- Loop pushing
  - Increases vector lengths
  - Must be paired with
    - Loop compression
    - Predication
  - Speedup of 34x wrt. non-optimized code (Solis-Wets)
- Larger genetic populations
  - Faster executions on the VE
    - Best: population of 1024 individuals
- ADADELTA (average results)
  - Similar: VE & V100 GPU
    - V100: 2.7x higher transistor density
  - VE is 4.1x faster than 2 x 64-core EPYC 7713 CPU

AutoDock-Aurora

https://github.com/esa-tu-darmstadt/AutoDock-Aurora