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 (IA³)
November 15, 2021
Contents

- Introduction
- Background
- Development
- Evaluation
- Concluding Remarks
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
SX-Aurora TSUBASA Vector Engine

- Device Characteristics
Overall System

x86 Vector Host (VH)

PCle

SX-Aurora TSUBASA Vector Engine (VE)
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

Receptor + Ligand = Complex

Pose 1

Pose 2
Molecular Docking

Receptor + Ligand → Complex

Pose 1

Score 1 (kcal/mol)

Pose 2

Score 2 (kcal/mol)
**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 $\rightarrow$ individual

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

- Genotype
  - Composed of set of genes

- (Pose) variable $\leftrightarrow$ gene

\[
\text{Pose} = \{x, y, z, \phi, \ldots\}
\]

\[
\text{Genotype} = \{\text{gene1, gene2, gene3, gene4, \ldots}\}
\]
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

- Crossover
-Mutation
-Selection

GA → LS
Lamarckian Genetic Algorithm (2/4)

Local Search (LS)

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

![Diagram](image-url)
Lamarckian Genetic Algorithm (3/4)

GA → LS → Solis-Wets → ADADELT → Superior molecular predictions → Legacy
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

\[
\text{for each } LGA\text{-run in } N_{LGA\text{-runs}}^{\text{TOTAL}} \text{ do}
\]

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

GA (population)

LS (population)
Lamarckian Genetic Algorithm

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)

Independent LGA runs

User-defined termination criteria
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

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

Calculating atomic coordinates

Receptor-Ligand score

Ligand-Ligand score
Solis Wets
Local Search
Function $S_W (\text{genotype})$

\[
\text{while } (N_{\text{LS-iter}} < N_{\text{LS-iter}}^{\text{MAX}}) \text{ and } (\text{step} > \text{step}^{\text{MIN}}) \text{ do} \\
\quad \delta = \text{create-delta (step)} \\
\quad \text{// new-genotype1} \\
\quad \text{for each gene in } N_{\text{genes}} \text{ do} \\
\quad \quad \text{new-gene1} = \text{gene} + \delta \\
\quad \text{if } SF (\text{new-genotype1}) < SF (\text{genotype}) \text{ then} \\
\quad \quad \text{genotype} = \text{new-genotype1} \\
\quad \quad \text{success}++; \text{ fail} = 0 \\
\quad \text{else} \\
\quad \quad \text{// new-genotype2} \\
\quad \quad \text{for each gene in } N_{\text{genes}} \text{ do} \\
\quad \quad \quad \text{new-gene2} = \text{gene} - \delta \\
\quad \quad \text{if } SF (\text{new-genotype2}) < SF (\text{genotype}) \text{ then} \\
\quad \quad \quad \text{genotype} = \text{new-genotype2} \\
\quad \quad \quad \text{success}++; \text{ fail} = 0 \\
\quad \quad \text{else} \\
\quad \quad \quad \text{success} = 0; \text{ fail}++ \\
\quad \text{step} = \text{update-step (success, fail)}
\]
Function $SW$ (genotype)

```
while ($N_{LS-\text{iter}} < N^{\text{MAX}}_{LS-\text{iter}}$) and (step > step^{\text{MIN}}) do
    delta = create-delta (step)
    // new-genotype1
    for each gene in $N_{\text{genes}}$ 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 $N_{\text{genes}}$ 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 \( S^w (\text{genotype}) \)

\[
\text{while } (N_{\text{LS-iter}} < N_{\text{MAX}}^\text{LS-iter} \text{ and } (\text{step} > \text{step}^{\text{MIN}})) \text{ do }
\]

delta = create-delta (\text{step})

// new-genotype1

\[
\text{for each gene in } N_{\text{genes}} \text{ do }
\]

\[
\text{new-gene1 = gene + delta}
\]

\[
\text{if SF (new-genotype1) < SF (genotype) then }
\]

\[
\text{genotype = new-genotype1}
\]

\[
\text{success++; fail = 0}
\]

\[
\text{else}
\]

// new-genotype2

\[
\text{for each gene in } N_{\text{genes}} \text{ do }
\]

\[
\text{new-gene2 = gene - delta}
\]

\[
\text{if SF (new-genotype2) < SF (genotype) then }
\]

\[
\text{genotype = new-genotype2}
\]

\[
\text{success++; fail = 0}
\]

\[
\text{else}
\]

\[
\text{success = 0; fail++}
\]

\[
\text{step = update-step (success, fail)}
\]
Function $Sw$ (genotype)

while ($N_{LS-\text{iter}} < N_{LS-\text{MAX}}$) and (step $> step_{MIN}$) do
  delta = create-delta (step)
  \ \ // new-genotype1
  for each gene in $N_{\text{genes}}$ 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 $N_{\text{genes}}$ 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)
Function $S^w$ (genotype)

while ($N_{LS-\text{iter}} < N_{LS-\text{MAX}}$) and ($step > step^{\text{MIN}}$) do
    delta = create-delta (step)
    // new-genotype1
    for each gene in $N_{\text{genes}}$ 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 $N_{\text{genes}}$ 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)
ADADELTA
Local Search
Function AD \((\text{genotype})\)

\[
\text{gradient} = \text{GC} \ (\text{genotype})
\]

\[
\text{while } (N_{\text{LS-}	ext{iters}} < N_{\text{LS-}	ext{iters}}^{\text{MAX}}) \text{ do}
\]

\[
\text{new-genotype} = \text{update-rule} \ (\text{genotype, gradient})
\]

\[
\text{if SF} \ (\text{new-genotype}) < \text{SF} \ (\text{genotype}) \text{ then}
\]

\[
\text{genotype} = \text{new-genotype}
\]

\[
\text{gradient} = \text{GC} \ (\text{genotype})
\]
Function \texttt{AD} (\textit{genotype})
\begin{align*}
\text{gradient} &= \text{GC} (\text{genotype}) \\
\textbf{while} \ (N_{\text{LS-ites}} < N_{\text{LS-ites}}^{\text{MAX}}) \textbf{ do} \\
\text{new-genotype} &= \text{update-rule} (\text{genotype}, \text{gradient}) \\
\textbf{if} \ SF (\text{new-genotype}) < SF (\text{genotype}) \textbf{ then} \\
\text{genotype} &= \text{new-genotype} \\
\text{gradient} &= \text{GC} (\text{genotype})
\end{align*}
Function $\text{AD (genotype)}$

\[
\begin{align*}
\text{gradient} &= \text{GC (genotype)} \\
\text{while } (N_{\text{LS-}\text{iters}} < N_{\text{LS-}\text{iters}}^{\text{MAX}}) \text{ do} \\
&\quad \text{new-genotype} = \text{update-rule (genotype, gradient)} \\
&\quad \text{if } SF (\text{new-genotype}) < SF (\text{genotype}) \text{ then} \\
&\quad \quad \text{genotype} = \text{new-genotype} \\
&\quad \text{gradient} = \text{GC (genotype)}
\end{align*}
\]
Function AD (genotype)

\[
\text{gradient} = \text{GC (genotype)}
\]

while \(N_{LS-\text{iters}} < N_{LS-\text{iters}}^{\text{MAX}}\) do

new-genotype = update-rule (genotype, gradient)

if SF (new-genotype) < SF (genotype) then

\[\text{genotype} = \text{new-genotype}\]

\[\text{gradient} = \text{GC (genotype)}\]

Gradients

(instead of +/- ops)

(more compute intense)
Development
Optimization

- Parallelization
- Vectorization
- Improving Vector-based Mapping
- Loop Pushing
Parallelization
**Function AutoDock-GPU**

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

**Function AutoDock-VE**

```plaintext
#pragma omp parallel for schedule (static, 1)
for each LGA-run in \( N_{LGA-runs}^{TOTAL} \) do
  while \((N_{score-evals} < N_{score-evals}^{MAX}) \) and \((N_{gens} < N_{gens}^{MAX}) \) do
    GA (population)
    LS (population)
```
Vectorization

- NEC compiler
  - Automatic vectorization

- Pseudorandom number generator
  - Initially employed
    - Linear Congruential Generator: $X_{n+1} = f(X_n)$
  - Dependence hinders vectorization
    - Replaced with
      - Built-in NEC Numeric Library
        - Collection functions
        - Mersenne-Twister (fully vectorized)
Vectorization

- NEC compiler
  - Automatic vectorization

- Pseudorandom number generator
  - Initially employed
    - Linear Congruential Generator: \( X_{n+1} = f(X_n) \)
    - 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: $X_{n+1} = f(X_n)$
    - 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?

Function $GA$ (population)

\[
\text{for each genotype in } N_{\text{pop-size}} \text{ do}
\]

Function $SF$ (genotype)

\[
\text{for each rot-item in } N_{\text{rot-list}} \text{ do}
\]
\[
\quad \text{PoseCalculation}
\]
\[
\text{for each lig-atom in } N_{\text{atom}} \text{ do}
\]
\[
\quad \text{InterScore}
\]
\[
\text{for each intra-pair in } N_{\text{intra-contrib}} \text{ do}
\]
\[
\quad \text{IntraScore}
\]
How vector pipes are leveraged?

```
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
```

Vector pipes leveraged only by *innermost* loops!

Some are *SHORT* loops!

Inner loops
How vector pipes are leveraged?

```
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
```

**Inner loops’ upper bounds**

<table>
<thead>
<tr>
<th>Input molecule</th>
<th>Nrot-list</th>
<th>Natom</th>
<th>Nintra_contrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>1u4d</td>
<td>23</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>1yv3</td>
<td>31</td>
<td>23</td>
<td>88</td>
</tr>
<tr>
<td>3er5</td>
<td>711</td>
<td>108</td>
<td>5,111</td>
</tr>
</tbody>
</table>
How vector pipes are leveraged?

Function GA (population)

for each genotype in $N_{\text{pop-size}}$
do

Function SF (genotype)

for each rot-item in $N_{\text{rot-list}}$ do
PoseCalculation

for each lig-atom in $N_{\text{atom}}$ do
InterScore

for each intra-pair in $N_{\text{intra-contrib}}$ do
IntraScore

Max. vec. length (VE): 256

<table>
<thead>
<tr>
<th>Input molecule</th>
<th>Nrot-list</th>
<th>Natom</th>
<th>Nintra_contrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>1u4d</td>
<td>23</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>1yv3</td>
<td>31</td>
<td>23</td>
<td>88</td>
</tr>
<tr>
<td>3er5</td>
<td>711</td>
<td>108</td>
<td>5,111</td>
</tr>
</tbody>
</table>

Large molecules can fill up the vector pipes.
How vector pipes are leveraged?

Function GA (population)

for each genotype in $N_{\text{pop-size}}$

Function SF (genotype)

for each rot-item in $N_{\text{rot-list}}$

PoseCalculation

for each lig-atom in $N_{\text{atom}}$

InterScore

for each intra-pair in $N_{\text{intra-contrib}}$

IntraScore

Max. vec. length (VE): 256

<table>
<thead>
<tr>
<th>Input molecule</th>
<th>Nrot-list</th>
<th>Natom</th>
<th>Nintra_contrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>1u4d</td>
<td>23</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>1yv3</td>
<td>31</td>
<td>23</td>
<td>88</td>
</tr>
<tr>
<td>3er5</td>
<td>711</td>
<td>108</td>
<td>5,111</td>
</tr>
</tbody>
</table>

$23 / 256 < \frac{1}{10}$ th

$108 / 256 < \frac{1}{2}$
Improving Vector-based Mapping

OpenCL thread → VE core

OpenCL thread → vector lane

Loop Pushing
Loop Pushing in GA (1/3)

Function GA (population)

\[ \text{for each genotype in } N_{\text{pop-size}} \text{ do} \]

Function SF (genotype)

\[ \text{for each rot-item in } N_{\text{rot-list}} \text{ do} \]
\[ \quad \text{PoseCalculation} \]

\[ \text{for each lig-atom in } N_{\text{atom}} \text{ do} \]
\[ \quad \text{InterScore} \]

\[ \text{for each intra-pair in } N_{\text{intra-contrib}} \text{ do} \]
\[ \quad \text{IntraScore} \]

- Call to functions obstructs vectorization
- Loop length within function is inefficient
Loop Pushing in GA (2/3)

Function GA (population)

for each genotype in $N_{\text{pop-size}}$ do

Function SF (genotype)

for each rot-item in $N_{\text{rot-list}}$ do

 PoseCalculation

for each lig-atom in $N_{\text{atom}}$ do

 InterScore

for each intra-pair in $N_{\text{intra-contrib}}$ do

 IntraScore

Function GA-VE (population)

for each rot-item in $N_{\text{rot-list}}$ do

for each genotype in $N_{\text{pop-size}}$ do

 PoseCalculation

for each lig-atom in $N_{\text{atom}}$ do

 for each genotype in $N_{\text{pop-size}}$ do

 InterScore

for each intra-pair in $N_{\text{intra-contrib}}$ do

 for each genotype in $N_{\text{pop-size}}$ do

 IntraScore
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

```python
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
```

L. Solis-Vasquez, E. Focht, A. Koch
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

GA evolves in a regular manner

All active members are processed in a GA iteration
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 \( SF(\text{new-genotype1}) < SF(\text{genotype}) \) then
    genotype = new-genotype1
    success++; fail = 0
else
    // new-genotype2
    for each gene in \( N_{\text{genes}} \) do
        new-gene2 = gene - delta
    if \( SF(\text{new-genotype2}) < SF(\text{genotype}) \) then
        genotype = new-genotype2
        success++; fail = 0
    else
        success = 0; fail++

Solis-Wets (with loop pushing)

while \( N_{\text{LS-iter}} > 0 \) do
    // Building compressed list of active indexes
    \( \text{pop}_{\text{size}}^{\text{active}} = 0 \)
    for each \( j \) in \( N_{\text{pop-size}} \) do
        if \( \text{LS}_{\text{active}}[j] \) then
            \( \text{idx}_{\text{active}}[\text{pop}_{\text{size}}^{\text{active}}] = j \)
            \( N_{\text{compressed}}^{\text{active}}[\text{pop}_{\text{size}}^{\text{active}}] = N_{\text{LS-iter}}[j] \)
            \( \text{success}_{\text{compressed}}[\text{pop}_{\text{size}}^{\text{active}}] = \text{success}[j] \)
            \( \text{pop}_{\text{size}}^{\text{active}} ++ \)
        ...

Loop Compression

Scalar → array
Predication
Solis-Wets (original)

**Function** $SW$ (genotype)

```
while ($N_{LS-\text{Iters}} < N_{MAX}$) and (step > step$^{\text{MIN}}$) do
    delta = create-delta (step)
    // new-genotype1
    for each gene in $N_{genes}$ do
        new-gen1 = 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_{active} = pop$_{size}^{active}$
for each $ii$ in pop$_{size}^{active}$ do
    if ($N_{LS-\text{Iters}}[jj] > N_{MAX}$) or (step$^{\text{compressed}}[jj] <= step^{\text{MIN}}$) then
        LS$_{active}[^{idx}_{active}[jj]] = 0$
        N$_{active} -= j$
        $j = ^{idx}_{active}[jj]$
        $N_{LS-\text{Iters}}[j] = N^{\text{compressed}}_{LS-\text{Iters}}[jj]$
        step$[j] = $step$^{\text{compressed}}[jj]$
        success$[j] = $success$^{\text{compressed}}[jj]$

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]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Before</th>
<th>After</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Time [sec]</td>
<td>1,382.2</td>
<td>40.0</td>
<td>~34x</td>
</tr>
<tr>
<td>Vector Time [sec]</td>
<td>2,217.6</td>
<td>280.2</td>
<td>~8x</td>
</tr>
</tbody>
</table>

Input molecule: 1hfs
## Impact of Loop Pushing (2/3) [Solis-Wets]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Before</th>
<th>After</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOPS</td>
<td>8,348.6</td>
<td>185,805.3</td>
<td>~22x</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>3,556.7</td>
<td>128,005.2</td>
<td>~36x</td>
</tr>
</tbody>
</table>

Input molecule: 1hfs
## Impact of Loop Pushing (3/3)

**[Solis-Wets]**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. Vector Length</td>
<td>195.4</td>
<td>214.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Optimal: 256)</td>
</tr>
<tr>
<td>Vector Operation Ratio [%]</td>
<td>75.3</td>
<td>99.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Optimal: 100%)</td>
</tr>
</tbody>
</table>
Comparison vs. GPUs and CPUs

- Impact of Population Size
- Best Results
## Hardware Devices

<table>
<thead>
<tr>
<th>SX-Aurora TSUBASA</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE 20B</td>
<td>V100</td>
<td>A100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EPYC 7713 (2 x 64 cores)</td>
</tr>
</tbody>
</table>
# Device Characteristics (1/2)

<table>
<thead>
<tr>
<th>SX-Aurora TSUBASA</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE 20B</td>
<td>V100</td>
<td>A100</td>
</tr>
<tr>
<td><strong>Process Size [nm]</strong></td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td><strong>Transistor Density [billions/mm²]</strong></td>
<td>0.009</td>
<td>0.025</td>
</tr>
</tbody>
</table>
### Device Characteristics (1/2)

<table>
<thead>
<tr>
<th></th>
<th>SX-Aurora TSUBASA</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VE 20B</td>
<td>V100</td>
</tr>
<tr>
<td>Process Size [nm]</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Transistor Density [billions/mm²]</td>
<td>0.009</td>
<td>0.025</td>
</tr>
</tbody>
</table>

Wrt. VE:
- V100: 2.7x
- A100: 7.2x

Higher transistor density
## Device Characteristics (2/2)

<table>
<thead>
<tr>
<th></th>
<th>SX-Aurora TSUBASA</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VE 20B</td>
<td>V100</td>
<td>A100</td>
</tr>
<tr>
<td>Process Size</td>
<td>16 [nm]</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>Transistor</td>
<td>0.009 [billions/mm$^2$]</td>
<td>0.025</td>
<td>0.065</td>
</tr>
<tr>
<td>Density</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perf [TFLOPS]</td>
<td>4.9</td>
<td>14.1</td>
<td>19.5</td>
</tr>
<tr>
<td>BW [GB/s]</td>
<td>1530</td>
<td>897</td>
<td>1555</td>
</tr>
</tbody>
</table>
# Device Characteristics (2/2)

<table>
<thead>
<tr>
<th></th>
<th>SX-Aurora TSUBASA</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VE 20B</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Process Size</strong></td>
<td>16 [nm]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transistor Density</strong></td>
<td>0.009 [billions/mm²]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Perf [TFLOPS]</strong></td>
<td>4.9</td>
<td>14.1</td>
<td>19.5</td>
</tr>
<tr>
<td><strong>BW [GB/s]</strong></td>
<td>1530</td>
<td>897</td>
<td>1555</td>
</tr>
</tbody>
</table>

VE’s main strength
Impact of Population Size [Solis-Wets]
Impact of Population Size [Solis-Wets]

Average runtimes (20 inputs)
Impact of Population Size [Solis-Wets]

Population Size $\uparrow$

VE runtime $\downarrow$

$2.3x$ ($= 51.7 / 22.2$)
Impact of Population Size [Solis-Wets]

Population Size ↑↑

GPU/CPU runtime ↔
Impact of Population Size [Solis-Wets]

VE faster than 128-core CPU: ~3x
(= 65.1 / 22.2)
Impact of Population Size [Solis-Wets]

![Bar chart showing docking runtime for different population sizes: VE 20B, V100, A100.]

- 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]

VE faster than 128-core CPU: 4x
(= 140.6 / 34.8)
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

![Graph showing docking runtime for different processors with Solis-Wets and ADADELTAL local search.

- VE 20B: 25.2 s
- V100: 30.1 s
- A100: 6.0 s
- 2×EPYC-7713: 30.3 s
- 2×EPYC-7713: 3.0 s
- 2×EPYC-7713: 20.2 s
- 2×EPYC-7713: 63.3 s
- 2×EPYC-7713: 122.9 s]
Population Size: 1024

VE faster than CPU:
2.5x (Solis-Wets)
4x (ADADELTA)
Population Size: 1024

ANKLE-X slower than A100:
- 8.4x (Solis-Wets)
- 1.4x (ADADELTA)
Population Size: 1024

VE slower than V100: 4.2x (Solis-Wets)
Population Size: 1024

Local Search

Solis-Wets
ADADELTA

Similar runtimes: VE & V100
(ADADELTA)
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