東京大学
情報理工学系研究科 創造情報学専攻
修士論文

計算化学に向けた
ドメイン特化言語のための Rank 型の提案

A proposal of domain specific language
with Rank type for computational chemistry

西田 秀之
Hideyuki Nishida

指導教員　千葉 滋 教授

2020 年 1 月

概要

本研究は計算化学の High Performance Computing のためにプログラムを簡便に記述出来るドメイン特化言語（DSL）の提案を行うものである。DSL は C++ の内部 DSL として実装し、計算の本質を示すアルゴリズム部と計算機上の効率化のためのスケジューリング部を分けるデザインで言語を構築する。同様のデザインを持つ従来のシステムでは、計算化学で用いるイレギュラーな配列に対応できず、ネストの深いループ、Reduction 計算に対してコードの記述が複雑になる。本研究では、計算化学で用いるデータ構造に応じた Rank 型の概念を提案し、その問題点を解決している。Rank 型は階層的なデータ構造を表す型であり、変数宣言時にデータの階層を読み込ませることで一種のポインタとして活用することができる。これにより、コード記述の簡便化、階層毎のコードの独立化、特定の Reduction 計算に向けた指示の明文化を果たしている。

Abstract

This research aims to make domain specific language (DSL) designed to make it easier to write high-performance computing code for computational chemistry. Although the previous DSL design which devide algorithm part and scheduling part is evaluated on the basis of highly portability, it is not appropriate for the computational chemistry field, because computational chemistry often uses irregular type data structure, and deeply loop nest. In this paper, we propose a Rank type for this kind of DSL design. This new type takes on a pointer-like role, and easily expresses equation and data structure of computational chemistry.

目次

第 1 章

はじめに

1.1 序論

計算化学は計算によって化学の諸問題を解決する一分野である。計算化学によって、化学反応や構造の本質の解明・実験では観測が困難な微視的現象の理解が進められている。コンピュータ性能の向上や理論の深化に伴い、様々な計算対象を高精度で計算できるため、実験の先導的役割も担うようになってきている。2013 年には、分子動力学シミュレーションがノーベル化学賞を受賞した。計算化学では、古典力学や量子力学の原理が用いられている。古典力学では、原子や分子の相互作用する粒子について運動方程式を解くことで力を計算する。一般的に古典力学を用いる方が早く、ダイナミクスの計算にしばしば用いられる。量子力学では、電子を露わに取り扱い、高精度な物性計算を可能としている。タンパク質などの巨大で複雑な分子をシミュレーション計算する際には、古典力学と量子力学の組み合わせが用いられることもある。計算化学では、計算対象の巨大化と高精度計算に伴って計算量の増加が問題点として挙げられることがある。これは、原子数や電子数の多さそのものと組み合わせ計算による組み合わせ爆発に由来する。そのため、効率的に計算するための高速化技法が議論されているが、計算化学用アプリケーションに落としこむことは労力を要する。主な理由は以下のように挙げられる。第一に、計算の理論自体が複雑化しているため、実装そのものが難しい点がある。新しい理論の試行錯誤で既存コードを変更する場合でも、該当箇所の検索や正確な実装は慎重に行う必要がある。第二に、適切な高速化技法は用いるデータと計算式に依存してしまう点がある。例えば、並列計算においては計算対象のデータ構造に合わせた並列化を行わなければ十分な高速化は果たされない。しかし、データ構造は実行時情報であるため、本分野で頻繁に用いられる C++ や Fortran 言語などの Ahead of Time (AOT) コンパイル言語では通常は用いることができない。

高性能なコードの生産性を向上させるメリットとして、以下に例を挙げる。第一に、理論の試行錯誤が容易となり、研究の進行に繋がる。理論的な研究を行う場合にコードの書き直しによる試行錯誤がしばしば行われるため、変更する箇所をミス無く書けることが望ましい。特に想定通りの精度が得られない場合に理論そのものの問題かコードに由来する問題かは分析の必要が出てくる。第二に、計算資源の節約ができる。特に計算化学でも使われるスーパーコン

ピュータなど、計算機の使用に巨額の費用がかかるケースでは高速な計算はそのまま研究費の削減に繋がる。したがって、高性能なコードを生産するプラットフォームは計算化学研究の推進に役立つと考えられる。このような、特定ドメインにおけるコードの要求を満たす手法として、ドメイン特化言語 (Domain Specific Language, DSL) が知られている。

DSL は、対象のドメイン毎にそのインターフェイス、機能が特化された言語である。ドメインのユーザが取り扱いやすい記法でコードを記述することができ、様々な分野で使われてきた。例えば、飛躍的な発展を見せる機械学習分野では、TensorFlow と呼ばれる DSL が活用されている。DSL は基本的に汎用言語 (C 言語や Fortran 言語) を用いて開発される。その形式により、外部 DSL と内部 DSL に分類することができる。外部 DSL は、開発するアプリケーションのメインとなる言語 (ホスト言語) とは異なる構文規則を持つ言語である。内部 DSL は、ホスト言語の構文を利用して表現される言語である。内部 DSL では、ホスト言語の機能 (コンパイラなど) を利用しつつ独自の機能を持たせることができる。例えば、C++ は AOT コンパイルを行うが、DSL で独自のコンパイラを組み込むことで Just in Time(JIT) コンパイルの機能を取り入れることができるのである。DSL は特に開発要件の定義は慎重に行わなければならない。何故ならば、特定ドメインのコードを記述しやすくする目的で開発されるためにユーザビリティが指標になるからである。ユーザビリティは個人の主観に委ねられがちであり、現在明確な評価法は知られていない。そこで、DSL の開発者にはドメインに対する知識が求められ、現行アプリケーションコードにおける問題点の抽出が必要である。現状計算化学分野で DSL があまり知られていない理由のその点であると考えられ、計算化学の知識と DSL の知識を同時に持った開発者が少ないと予想される。そのほかにも、計算化学分野のユーザが新しい言語を学ぶ学習コストの高さが懸念される。そこで、DSL は本分野のユーザが使い慣れた言語の拡張であるべきであり、加えて数式をより簡単に記述できる形式が求められる。

DSL の中でも計算の本質を示すアルゴリズム部分と計算機上の最適化を示すスケジューリング部分を分離させて記述する言語デザインが存在する。画像処理用 DSL である Halide[1] に代表されるそのデザインは、異なるアーキテクチャにコードを流用する際や最適化手法を変更する際にスケジューリング部の変更のみで対応することができる。これは汎用言語で全体のコードを見直さなければならない形式よりも取り回しに優れる。例えば、C 言語で OpenMP を使う際には for 文の直前に pragma ディレクティブを挿入しなければならないが、OpenMP が使えない環境ではコード全体から該当箇所を検索して修正する必要がある。この時に for 文を間違って修正してしまうなどのミスが発生する可能性がある。一方で Halide のデザインではスケジューリング部にまとめられているため、計算の本質部分には触らずに変更することができる。このように、Halide は可搬性の高いデザインとして広く知られている。

しかし、Halide の言語デザインは、計算化学に対して容易に適応することはできない。元々画像処理用として設計されているため、計算化学に使うことは想定されていないが、使う場合を想定して述べる。主な問題点は以下のようである。第一に、アルゴリズムの記述に扱うデータの次元数分の引数を必要とするため、可読性が低くなる点がある。例えば、分子 ー Sub ー 原子の階層的データ構造が存在する場合に、原子のデータ (x 座標など) にアクセスするために分子と Sub のインデックスも必要となる。特に計算化学で頻出する組み合わせ計算の場合

には、インデックスの数が巨大化する。画像処理のように、三次元または四次元程度の計算であれば良いが、計算化学で用いることを想定すると 1 行あたりの記述量が多くなりすぎる。第二に、Reduction 計算が非自明である点がある。Halide の言語デザインでは、RDom と呼ばれる特別な型を用いて Reduction を行う。RDom は他の引数と同様にユーザが取り扱うが、暗黙的な変換を行うため一見して Reduction 計算を行うかが判断できない。Reduction の識別のために、ユーザに変数の命名規則を押し付けている形となっている。第三に、計算化学ではイレギュラーなデータ構造を用いる点がある。画像処理は、xyz 座標で代表されるように、全体が矩形的なデータ構造となっている。しかし、分子に属する原子数が異なることからも分かるように、計算化学ではイレギュラーなデータ構造を取り扱う。loop fusion などは共通の技術として用いることができるが、タイリングなどの矩形データ構造用の最適化は用いることはできない。

本研究は、計算化学の High Performance Computing のための新たなプラットフォームの創出を目指すものである。DSL 開発のライフサイクルに則って、計算化学に向けた DSL が満たすべき要件定義を行なった。特に先行研究のデザインで懸念されるユーザビリティの低下は解決すべき課題である。化学者が用いることを想定するため、計算化学で頻繁に用いられる C++ をホスト言語および生成言語とした DSL を開発する。インターフェイスは可搬性の高い Halide に基づくものとして、JIT コンパイルの利点を生かした設計を行う。本 DSL では、計算化学を分析して Halide 形式に対応するために新たに Rank 型の概念を導入した。Rank 型は階層的なデータ構造を表現するための抽象機構である。簡単のため、分子が SubSystem によって分割され、SubSystem の中に原子がある三次元の配列で説明する。このデータ構造は、分子が原子の下に属することが無いことのように各次元が上位の次元に依存している。そこで、変数宣言時にデータの階層を Rank 型に読み込ませることで、依存した上位階層を省略して記述することができる。例えば、汎用言語では、data[i][j][k] と書くところを Rank 型の変数を用いて atom と記述できる。上位の次元のインデックスである i と j は暗黙的に引き継がれる。これは汎用言語でのポインタの役割である。Rank 型の導入により、以下の利点が確認された。第一に、コード記述の簡便化がなされた。Halide で用いる型では、上位階層の情報も全て明示的に記述しなければならない。Rank 型を用いることで、下位階層のデータを用いる際には上位階層の記述を必要としないので、コードの記述量が減り、可読性の向上に繋げている。例えば原子のデータにアクセスする場合に、Halide では、data(mol, sub, atom, ”x”) などと記述しなければならないところを本 DSL では atom.attr(”x”) と書くところを Rank 型の変数を用いて atom と記述できる。これは引数が減って取り回しが良くなっているほか、実際の数式にも近い形となる。第二に、階層毎のコードの独立化がなされた。計算化学では計算手法によって、階層の追加や削減が頻繁に発生する。Rank 型によって、上位階層のデータとは独立に計算式を定義することができるため、コードの可搬性が向上している。次に本 DSL では、Reduction 計算の明文化を果たした。例えば総和計算を行う場合には、sigma 関数を使うという規約を設定している。sigma 関数では第一引数に計算式を示す型の変数、第二引数以降に Reduction を行いたい次元の Rank 型の変数を入れる。これにより Reduction 計算の明示化の他、ユーザがどこ次元で畳み込みを行いたいかを明確にしている。組み合わせ計算で頻

出するが、単一の計算式に同階層の変数を複数使用する場合、Halide の形式では畳み込みの明示化が困難であった。本 DSL では、階層構造は既知であるという前提に立つため、特定のループへの指示が明確である。本 DSL で構成したコードと Halide の仮想コードとの比較を行った。本研究で提案した Rank 型と sigma 関数を用いることでコードの可搬性・可読性を向上させている。

本 DSL は Deep Embedding と呼ばれる手法を用いて開発した。この手法では、計算式に基づく抽象構文木 (AST) を組み込んだ独自コンパイラに読み込ませることで、JIT 的にコード生成を行う。システム構成としては、3 段階コンパイルを用いる形式である。本 DSL を用いて記述されたコードは通常の C++ コンパイラでコンパイルされた後、生成した AST を独自コンパイラでコンパイルして最適化された C++ コードを生成、それを C++ コンパイラがコンパイルして最終的な結果が得られる。本 DSL は高速度計算を目指しているため、最適化機能として MPI 並列化、MP 並列化を組み込んでいる。ユーザは mpi_parallel 関数と mp_parallel 関数でそれらを用いることができ、2、3 行の変更のみでそれらの並列化を検討することができる。

本 DSL の実用性を示すために、簡易な計算に対しての実験を行なった。なお、検証のために Halide が計算化学に適応出来たと仮定しての仮想コードと比較している。Halide とのコンパイル ＋ 実行時間の比較では、本 DSL の方が遅い結果となった。読み込ませるデータの情報量は同じであるが、矩形的なデータ構造を想定されている Halide の方がより最適化がなされていると考えられる。次に、並列化の検証として本 DSL で書いたコードのスケジューリング部 2、3 行のみを変更して計算時間の比較を行なった。生成したコードは逐次実行・MP 並列・MPI 並列・ハイブリッド並列 (MP + MPI) ・ロードバランシングの五種類である。計算に用いるデータ構造やスレッド数の変更により最適化効率が変化していることが確認された。ここで重要な点は計算対象と計算機に応じた比較実験を単純な変更のみで実現可能である点である。

本研究の主な貢献は以下の通りである。

- 化学分野 (応用分野) などの user adjustable な DSL に初めてポインタ的な動作を組み込んだ。
- 計算化学のデータ構造や理論を検証して、既存研究応用の際の問題点を解決する DSL のデザインを考案した。

1.2　本論文の構成

本論文の構成内容を述べる。本論文は本章をはじめとする 6 章編制である。第 2 章では本研究と関連研究を取り巻く諸概念について説明する。今回の研究対象となる DSL の基礎知識を述べた後に対象ドメインである計算化学とインターフェイスを参考とした Halide について述べ、計算化学用 DSL を設計する際の問題点を取り上げる。第 3 章では、本研究で開発した言語と新たに導入した概念を説明する。同時に、考案した user adjustable な DSL デザイン

を提案し、既存研究との相違点を述べる。第 4 章では、本研究で開発した言語の実装について述べる。システム構成を述べた後に言語仕様をマニュアルとしてまとめ、実装した機能、設計上の制限について述べる。第 5 章では、実装した DSL を用いた実験について述べる。Halide とのコンパイル比較実験と並列化コードの実行時間計測を検証している。最後に，第 6 章で本論文のまとめと今後の課題について述べる。

第 2 章

研究背景

本研究は、計算化学に向けた新たなプラットフォームを創出することを目的としている。そこで本章では、本研究に必要となる前提知識・諸概念を説明する。2.1 節では、本研究で開発することになる DSL について説明する。2.2 節では、対象ドメインとなる計算化学について、そのソフトウェア開発の問題点と共に説明する。2.3 節では、本研究に関わる既存研究について説明する。2.4 節では、計算化学用 DSL を設計する上で、既存研究を用いる場合の問題点について説明する。

2.1 DSL

プログラミングにおけるソフトウェア開発のスタイルは従来から研究されてきた。その中に、言語指向プログラミング (Language Oriented Programming, LOP) [2] と呼ばれる開発スタイルが存在する。LOP は諸問題に対し、汎用言語 (C 言語や Fortran など) 1 つのみを用いるのではなく、その作業に適した言語を用いる開発スタイルである。LOP では、開発者がその問題に特化した言語を開発したり、その問題固有の概念を利用して汎用言語を拡張する手法が挙げられている。このスタイルで開発された、特定の領域に対して特化させたプログラミング言語はドメイン固有言語 [5] (Domain Specific Language, DSL) と呼ばれる。DSL においては、ドメインに対するユーザ要件およびユーザの望む表現手法が、ユーザが行う実装と可能な限り同一になるように開発される。

Data Base Management System (DBMS) 操作のためのの DSL である SQL [7] や機会学習用 DSL である Tensorflow [8] は広く扱われる DSL の一つである。例えば SQL では、図 2.1 で示すコードで記述される。これは、data テーブルの全てのデータを取得する文である。

汎用言語では、*演算子は様々な意味を持つ。C 言語では、計算式中に*演算子を用いる場合は掛け算として解釈されるが、ポインタを示す演算子としても用いることができる。しか

```
1  SELECT * FROM data
```

図 2.1. SQL のコード例

し、図 2.1 の SQL コードでは、”全て”と解釈される。このように、特定の文字がどの意味を持つかは言語の構文規約とそれに付随するコンパイラに依存する。DSL では、対象のドメインのユーザが望む表現手法に近い形となるようにそれぞれの演算子を定義、解釈するためのコンパイラを開発する。学術面においても、宇宙科学の国際的プロジェクトである Square Kilometre Array (SKA) 計画に専用の DSL が開発された。このように各ドメインに特化した DSL の開発は様々な分野で行われている。ドメインに適した DSL を用いることで、その領域に関しては汎用言語でコードを記述するよりも簡便に、品質の高いコードを開発することができる。

DSL は、特定領域のみに特化しているために、その領域のユーザからは理解がしやすいメリットがあり、メンテナンス性や可読性の向上に繋がる。[6] しかし、その領域に特化させるに当たって十分な機能を持っているかの検証は難しい。何故ならば、特化の十分性の検証にはドメインに纏わる広範な知識が必要であるほか、それが全ての要求を満たしているかの総当たりな検証は現実的に不可能だからである。また、ユーザにとって使いやすいかの検証も困難であり、現在定量的な評価手法は知られていない。このように、DSL は設計・開発・管理のコストがかかることにデメリットがある。さらに、新たな言語をユーザに用いてもらう都合上、学習コストがかかる。

DSL は、基本的に汎用言語によって構築および開発される。DSL に対して元になる汎用言語のことをホスト言語と呼ぶ。DSL 開発に当たっては、ライフサイクルとして 6 つのフェーズが提言されている。[3][4]

1. 決定 (decision)
2. 分析 (analysis)
3. 設計 (design)
4. 実装 (implementation)
5. 配備 (deployment)
6. 保守 (maintenance)

第一フェーズとして、DSL 開発は、DSL を新規に開発するかを判断することから開始する。汎用言語・既存 DSL で足りる場合には新規に開発する必要はない。ドメインの諸概念を判断するためには、DSL 開発者は既存手法に則ったソフトウェア開発プロセスを経験していることが望ましい。ドメインに纏わる深い知識が必要となるためである。

第二フェーズとして、対象ドメインのモデル構築を行う。まず、興味のある分野の内、どこまでをドメインにするか、スコープを定義する。次に、ドメインで用いる要素を定義する。その後、ドメイン記述を DSL で行う部分とホスト言語で行う部分に分割する。

第三フェーズとして、DSL とホスト言語の関係を洗い直して設計を行う。ホスト言語を特殊化して文法の制限を行うか、拡張して表現を広げるかを設計する。

第四フェーズとして、適切な手法を用いて実装する。ホスト言語の持つ機能の内どの程度を用いるのか、DSL を直接実行するもしくは変換するのかなどを選択する。既存のエディタを用いることも重要である。DSL 開発の労力は手法に依存するので、適切な選択が必要である。

第五フェーズとして、DSL を扱うことができる環境の整理がなされる。ユーザが実際に用いることで、性能の評価が可能となる。

第六フェーズとして、DSL の評価を元とした修正を行う。一般的に 0 から開発するよりも修正する方が容易である。しかし、ドメインまたは DSL の本質的な変更によって実装が変わることがあり得る。

DSL は、大きく 2 種類に大別することができ、それぞれ外部 DSL・内部 DSL と呼ばれる。ホスト言語とは全く別の構文を持ち、機能を全て独自に作るようなものを外部 DSL と呼ぶ。ホスト言語上で開発され、あたかも新たな言語を構成するかのように開発されたものを内部 DSL と呼ぶ。以下、内部 DSL と外部 DSL の説明を行う。

2.1.1　外部 DSL

外部 DSL はホスト言語とは異なる構文規約を持つ。外部 DSL の長所の一つが、ホスト言語の文法の制約を受けないことである。つまり、特化するドメインに対して開発者が自由に設計することができるため、最も簡単な形式を用いたドメインの表現が可能となる。しかし、短所も何点か挙げられる。その一つが字句解析器・構文解析器・評価器などは DSL 開発者が実装しなければならないことである。開発する言語が複雑になるとこれらの開発はより困難となる。加えて、現代の開発環境における外部 DSL の問題点も挙げられる。現在、汎用言語には専用の統合開発環境 (Integrated Development Environment, IDE) が存在することが多い。外部 DSL はホスト言語の構文に直接的な繋がりがないため、IDE は外部 DSL の解釈を行うことができない。この問題点は、より複雑なコードを記述する際に顕著となる。例えば、プログラマが関数名を変更するために IDE のリファクタリング機能を用いて rename する場合、IDE が自動的に共通の関数を検索して全置換を行う。しかし、外部 DSL で書かれた言語の構文規約は IDE では理解できないため、リファクタリング機能を用いることができない。ユーザが手動で全置換する場合には、ミスをする可能性が生じる。デバッグする際も同様である。外部 DSL で記述されたコードはパースの後にホスト言語へと変換されることがある。変換後のコードであればデバッガがステップイン実行などが可能であるが、元となったコードのステップイン実行はできない。

2.1.2　内部 DSL

内部 DSL はホスト言語の構文のみで実現できる DSL である。ホスト言語の機能を用いて記述法を工夫することで、対象ドメインに適した構文を表現する。内部 DSL の利点としてホスト言語のツールを流用できることが挙げられる。コンパイラを流用することができるため、独自のパーサなどを作る必要がない。IDE もホスト言語のものを使うことができる。これにより、開発コストの軽減に繋がっている。しかし、問題点も何点か挙げられる。その一つが、内部 DSL はホスト言語の文法に依存するため、構文規則やプログラミングモデルの逸脱ができない点である。仮にドメインにより適した文法が存在しても、それがホスト言語の文法的制

$$\sum_{sub}^{in\ mol} \sum_{atom1}^{in\ sub} \sum_{atom2}^{in\ sub} \sqrt{(atom1.x - atom2.x)^2 + (atom1.y - atom2.y)^2}$$

図 2.2. 計算式

```
1  calc1(atom1, atom2) = ((atom1.attr(0) - atom2.attr(0)) ** 2 +
2                (atom1.attr(1) - atom2.attr(1)) ** 2 )) ** (1 / 2);
3  calc2(sub) = sigma(calc1, atom(sub), atom(sub));
4  calc3(mol) = sigma(calc2, sub(mol));
```

図 2.3. コード例

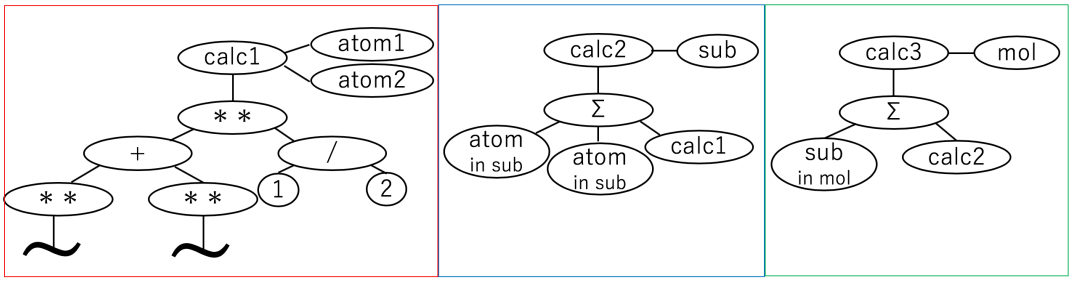

図 2.4. 生成される AST

限を超える場合には採用することはできない。その上、ホスト言語を前提としたものであるために、ホスト言語を知るユーザしか使えないことも問題点の一つである。

2.1.3 Deep Embedding

Deep Embedding と呼ばれる内部 DSL 開発手法が存在する。Deep Embedding を用いることで、計算式を記述した際に計算がその場では実行されず、計算式を AST として保持する。

例えば、計算式図 2.2 を記述した図 2.3 で示されるコードでは図 2.4 で示される AST が出力される。

この計算式は、二次元平面において二点間距離を計算し、全ての和を取る計算式である。この時、汎用言語においては外側のループから記述するところを、内側から記述していく。図 2.2 の赤、青、緑の順にコードが書かれていることが分かる。それぞれの行をコンパイラが解析する時に、実際の計算は行われていない。この 3 行では図 2.4 で示される AST がそれぞれ計算される。このように C++ コンパイラ実行時に AST を保持しておき、後に別のコンパイラを実行することで結果を得る手法を Deep Embedding と呼ぶ。

この計算式を全て読み込んだ後、最終的には図 2.5 で示される AST が生成される。

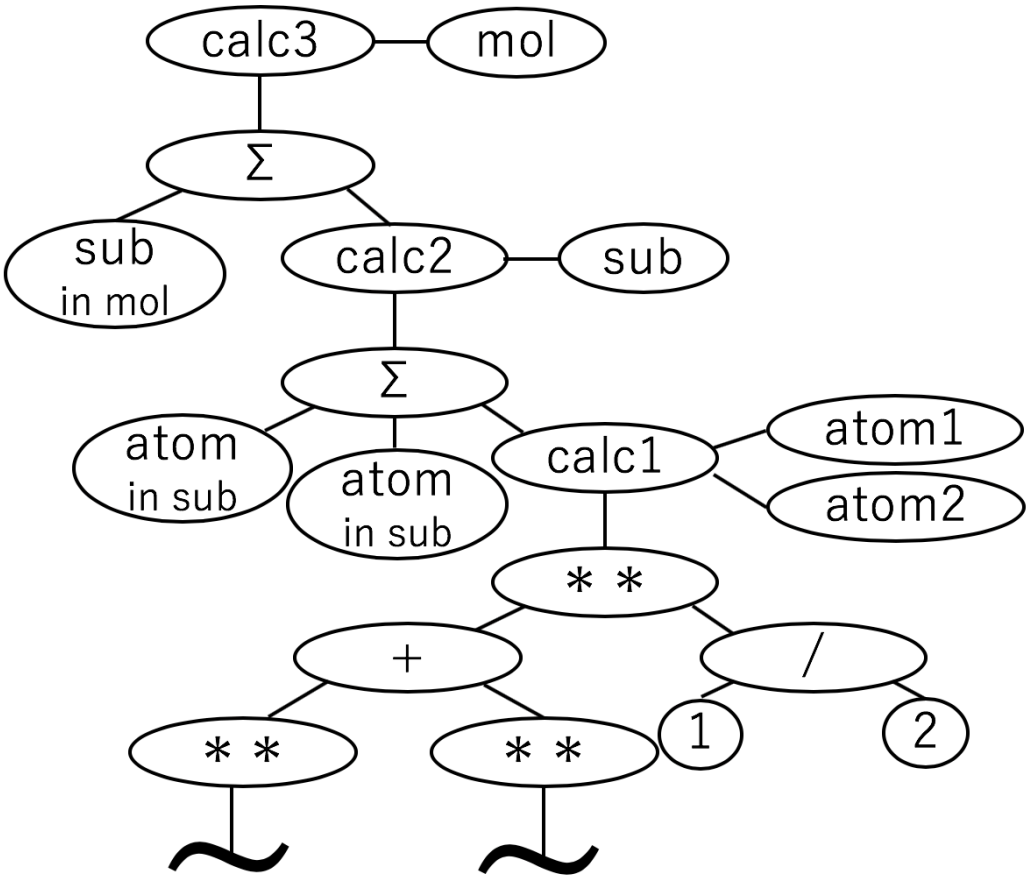

図 2.5. 最終的に生成される AST

2.1.4　オペレータオーバーロード

オペレータオーバーロードの機能を用いることで、Deep Embedding を C++ をホスト言語とする内部 DSL で実現できる。オーバーロードは特定の演算子に通常と異なる意味をもたせたい場合に用いられる C++ に備わった標準機能である。例えば通常の C++ では、+ などの二項演算子は計算結果を出力するために用いられる。しかし、Deep Embedding のために + 演算子に AST を生成する役割をもたせたい。+ 演算子を計算式を表現する Expr 型に暗黙的に変換するため、その Expr 型を引数とする二項演算子を図 2.6 のようにオーバーロードすることで、Add という AST のノードが生成される。

2.2　計算化学

計算化学はコンピュータ技術の発展に伴い、研究面・工業面の広い範囲で取り扱われている。実験的手法の補助のみならず、現在は実験では観測できない微視的な観点による物性も計

```
1  Expr operator+(Expr lhs, Expr rhs)
2  {
3      return Add::create(std::move(lhs), std::move(rhs));
4  }
```

図 2.6. オペレータオーバーロード

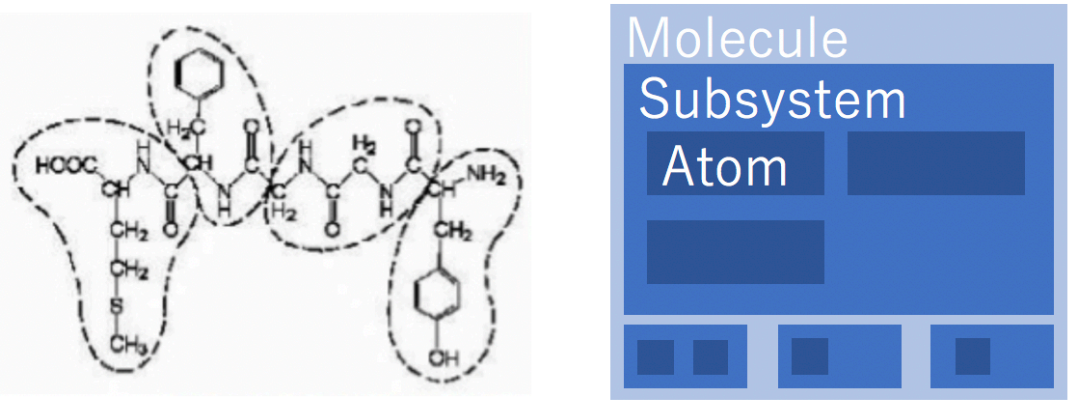

図 2.7. 計算化学における計算対象モデル

算によって予測することが可能となっている。計算化学の中でも、その精度から現在盛んに研究されている手法に量子化学計算がある。量子化学計算では、古典的手法と異なり、電子の相関を露わに取り入れている。これにより、現実の分子構造を計算で予測したり、反応経路の予測が可能となっている。しかし、量子化学計算用ソフトウェアを取り巻く問題も存在する。例えば以下の二点が挙げられる。第一に、実装難易度が高い点がある。量子化学計算では、量子論的振る舞いをする電子の状態を基底関数と呼ばれる関数を主に用いることで確率 (電子密度) を基に計算する。この基底関数を表現する計算式は複雑であり、計算式と実装コードの乖離はコードの保守性・可読性を下げる。第二に、最適化が困難である点がある。量子化学計算では計算量が電子数の累乗のオーダーで増加するため、分子サイズの偏りと行列のスクリーニングにより計算量が大きく変化し、並列化のロードバランシングが困難である。また、計算に用いる行列サイズも同様に増大するため、大規模系ではメモリの不足が頻出する。そのため、多くの量子化学計算ソフトウェアでは行列を分割してメモリにのるように調整される。これらの最適化は、計算系とハードウェアに合わせた試行錯誤を必要とする。

計算化学では、例えば図 2.7 に示すようなモデルが扱われる。

量子化学計算では電子の基底関数を元に考えるが、ここでは簡単のために古典的なモデルを C++ で考える。ある分子を表すデータ構造は階層的になっており、分子そのものを示す Molecule クラスのクラス変数として Atom クラスが存在する。Atom クラスはそれぞれの原子の種類、座標などを変数として所有している。構造最適化計算では、各原子毎の距離に基づいた力を計算することで最適化された構造を求める。しかし、計算手法を変えるとデータ

構造も変化する。例えば、Fragment Moleculer Orbital (FMO) 法 [9] などでは高速化のために、分子を SubSystem で区切る。 (分子軌道法であるために実際は別のデータ構造である) SubSystem 内の原子のみで力を計算する場合には Molecule - Atom であったデータ構造は Molecule - SubSystem - Atom に変化する。そのため、一般的な言語では、その計算手法毎にクラスの書き換え、計算アルゴリズムの書き換えが発生する。さらに、並列計算を行う場合には問題は複雑化し、Atom 単位で MP 並列化を行なっていたところを SubSystem 単位に変更するなどを考慮しなければならない。FMO 法ではさらに、ロードバランシングが問題となっている。高速化技法の一つである並列化を行う際に、それぞれのプロセッサ毎の仕事量は同じであることが望ましい。しかし、SubSystem に区切る際に、中の原子数にばらつきが生じるので、仕事量に差が出てしまう。特に、量子化学では、電子数の累乗のスケールで計算量が増えるので、この問題は深刻化する。

現在、計算化学用の DSL で広く使われているものはあまり知られていない。その理由として、主に以下の点が挙げられる。第一に、DSL にはドメインの知識が求められるため、開発が困難な点がある。化学に属する計算化学と情報学で扱われる DSL は異分野であり、それぞれの知識を同時に扱うことはあまりない。そのため、計算化学の DSL を開発するようなデベロッパーが少ないと考えられる。第二に、DSL の学習コストがかかる点がある。計算化学のユーザは C++ か Fortran 言語を主に用いている。Python によるフレームワークも存在するが、高速性を重視する場合が多いので前者が頻出する。この環境に対して全く新しい言語の導入は学習コストの点から懸念される。

2.3　既存研究

本節では、DSL の既存研究の例を挙げる。

2.3.1　Halide

画像処理用の DSL として広く用いられている DSL に、Halide がある。これは、C++ をホスト言語とした内部 DSL である。特徴的な点は主に二種類挙げられる。第一にインターフェイスが特徴的であり、計算の本質を示すアルゴリズム部と計算機上の最適化命令であるスケジューリング部を分割して記述することができる。これにより、計算機や計算対象ごとに最適化の必要がある部分を計算式とは別個にカスタマイズできるため、可搬性の高い DSL として知られている。第二にマルチプラットフォーム対応な点がある。Ahead of Time（AOT）コンパイルする言語である C++ をホスト言語としながら、Halide 独自のコンパイラを実行時に走らせることで Just in Time（JIT）コンパイラの機能を持たせており、ユーザが指定したプラットフォームへの特化したコードを実行時に生成する。Halide で主に用いられる型は五種類あり、Func、Var、Expr、Buffer、RDom とそれぞれ呼ばれる。Func 型は各画素の値を持つかを定義する型であり、=オペレータに Expr 型を渡すことでその計算式が定義される。Var 型は、Func 型で用いる変数名を定義しており、Func 型の () オペレータに渡すこと

で Func のどの次元にどの変数が相当するかを示す。Expr 型は、Func 型の計算式を定義するものであり、ユーザが明示的に用いることは少ないが、計算式は全て暗黙的に Expr 型に変換される。Buffer 型は、データの実態を示す型であり、テンプレートパラメータに規定の型を入れ込むことで複数の bit 形式に対応できる。RDom 型は、Reduction 計算を行う際に用いる型であり、変数宣言時に RDom の定義域を指定し、計算式上で用いることでループ構造を形成する。上記の型を用いてアルゴリズムを純粋関数として定義した後、コード生成を行う。Halide によるアルゴリズム記述を図 2.8 に示す。

```
1  int main() {
2    Halide::Func f;
3    Halide::Var x, y;
4    Halide::Buffer<int> output;
5
6    f(x, y) = x + y;
7    output = f.realize(100, 100);
8  }
```

図 2.8. Halide によるアルゴリズム記述

```
1  for(int y = 0; y < 100; y++){
2    for(int x = 0; x < 100; x++){
3      f[x][y] = x + y;
4    }
5  }
```

図 2.9. 等価なループ文

```
1  Halide::Func blur_3x3() {
2    Halide::Func blur_x, blur_y;
3    Halide::Var x, y, xi, yi;
4    Halide::Buffer<uint8_t> input = load_image("images/figure.png");
5
6    // algorithm part
7    blur_x(x, y) = (input(x−1, y) + input(x, y) + input(x+1, y))/3;
8    blur_y(x, y) = (blur_x(x, y−1) + blur_x(x, y) + blur_x(x, y+1))/3;
9
10   // scheduling part
11   blur_y.tile(x, y, xi, yi, 256, 32)
12   .vectorize(xi, 8).parallel(y);
13   blur_y.realize(input.width(), input.height());
14 }
```

図 2.10. Halide を用いたブラー操作

Halide は C++ の内部 DSL であるため、図 2.8 の 2~4 行目のように変数宣言を C++ と同様に行う。6 行目がアルゴリズム記述部であり、関数 f は定義域 x、y に対して x + y を値域に持つことを意味する。ここでは Func の計算パイプラインを定義しただけであり、計算処理そのものは行われていない。7 行目で realize 関数に定義域 x、y を渡すことで Halide 独自のコンパイラが動く。計算パイプラインに基づいたコードが生成・実行され、計算結果を得られる。6 行目に示した計算式と等価な C++ コードを図 2.9 に示す。単純なコードであるが、Halide は for 文を明示的に書かせないことがわかり、ミス要因の削減に繋がっている。

この Halide のシステムは、より大規模な最適化が必要な系において威力を発揮する。そこで、画像にブラー操作を行う例を図 2.10 および図 2.11 に示す。

```
1  void box_filter_3x3(const Image &in, Image &blury) {
2    __m128i one_third = _mm_set1_epi16(21846);
3    #pragma omp parallel for
4    for (int yTile = 0; yTile < in.height(); yTile += 32) {
5      __m128i a, b, c, sum, avg;
6      __m128i blurx[(256/8)*(32+2)];
7      for (int xTile = 0; xTile < in.width(); xTile += 256) {
8        __m128i *blurxPtr = blurx;
9        for (int y = -1; y < 32+1; y++) {
10         const uint16_t *inPtr = &(in[yTile+y][xTile]);
11         for (int x = 0; x < 256; x += 8) {
12             a = _mm_loadu_si128((__m128i*)(inPtr-1));
13             b = _mm_loadu_si128((__m128i*)(inPtr+1));
14             c = _mm_load_si128((__m128i*)(inPtr));
15             sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
16             avg = _mm_mulhi_epi16(sum, one_third);
17             _mm_store_si128(blurxPtr++, avg);
18             inPtr += 8;
19         }}
20       blurxPtr = blurx;
21       for (int y = 0; y < 32; y++) {
22         __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
23         for (int x = 0; x < 256; x += 8) {
24           a = _mm_load_si128(blurxPtr+(2*256)/8);
25           b = _mm_load_si128(blurxPtr+256/8);
26           c = _mm_load_si128(blurxPtr++);
27           sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
28           avg = _mm_mulhi_epi16(sum, one_third);
29           _mm_store_si128(outPtr++, avg);
30         }
31       }
32     }
33   }
34 }
```

図 2.11. C++ によるブラー操作

図 2.10 と 2.11 を比較する。まず Halide では、前述した通りに計算の本質を示すアルゴリズム部と計算機上の最適化命令であるスケジューリング部を分割して記述していることが分かる。この例では、図 2.10 の 7、8 行目がアルゴリズム部であり、11、12 行目がスケジューリング部に相当する。このインターフェイスにより、異なるアーキテクチャに流用するときでもスケジューリング部のみの変更で対応することができ、可搬性を向上させている。スケジューリングとしては、タイリング、ベクトル化、並列化を用いている。それぞれ tile、vectorize、parallel 関数を用いることで実現される。これらは Halide が画像処理に特化させるために組み込まれた primitive な関数と呼べる。一方等価な C++ コードにおいては、3 行目で示されるように計算式の途中に最適化命令が記述されている。異なるアーキテクチャに流用する際にはコード全体を見て書き換えなければならない。これは、コード全体を見直すためにメンテナンス時間が増加することや変数の指定のミスを誘発してしまう。

for 文の取り扱いに関しても相違点が認められる。Halide では for 文は明示的に書かない。Func 型の () オペレータにおける引数が for 文の定義域に相当し、Var 型を当てはめることで自動的にドメインが決定される。等価な C++ コードにおいては、通常の C++ における for 文に基づく記述がなされている。これはループの定義域の設定ミス、ブロックの閉じ忘れなどが生じる可能性がある。Halide のデザインではそのようなミスが生じない。

このように、最適化コードを汎用言語のみで書くと煩雑になりがちである。Halide はその煩雑になる要因を独自コンパイラによって吸収する機能を持っている。ユーザビリティの観点から見ても、Halide は汎用言語よりも最適化コードを書きやすい言語と言える。

また、Halide の既存研究 [12][13][14][15] で示されるように、オートチューニング機能も強力である。その分野のエキスパートが専用にチューニングしたコードと比較しても、Halide を用いれば少ない行数で高速なコードを生成できることが報告されている。このように、人間では探索しづらいチューニング可能性を探索できることも Halide の強みである。

2.3.2　PolyMage

PolyMage [10] は Halide と同様に画像処理用の DSL である。こちらはホスト言語としては Python を用いており、生成コードは C++ となる。PolyMage を用いたコードを図 2.12 に示す。

ホスト言語が違うこともあり、構文規則自体は Halide と異なる。しかし、計算パイプラインを組み立てて最適化を行う点に関しては Halide と同様である。Halide との相違点として挙げられるのは、計算のオートチューニングに動的計画法に基づいたグルーピングアルゴリズム [16] を用いている点である。Halide によるオートチューニングでは、brute force な手法が取られていた。PolyMage ではブロッキングなどの最適化可能性を評価関数を用いて探索することにより、Halide よりも高速なコードを生成可能であると報告されている。

```
1  R, C = Parameter(Int, "R"), Parameter(Int, "C")
2  I = Image(Float, "I", [R+2, C+2])
3
4  x, y = Variable("x"), Variable("y")
5  row, col = Interval(0, R+1, 1), Interval(0, C+1, 1)
6
7  c = Condition(x, '>=', 1) & Condition(x, '<=', R) & \
8  Condition(y, '>=', 1) & Condition(y, '<=', C)
9
10 Iy = Function(varDom = ([x, y], [row, col]), Float, "Iy")
11 Iy.defn = [ Case(c, Stencil(I(x, y), 1.0/12, \
12 [[-1,-2,-1], \
13 [ 0, 0, 0], \
14 [ 1, 2, 1]])) ]
```

図 2.12. PolyMage を用いたコード

```
1  val sumNbh = fun(nbh => reduce(add, 0.0f, nbh))
2  val stencil = fun( A: Array(Float, N) =>
3    map(sumNbh, slide(3, 1, pad(1, 1, clamp, A))))
```

図 2.13. Lift を用いたコード

2.3.3 Lift

Lift [11] と呼ばれる DSL が存在する。これは、前述した Halide や PolyMage とは対象ドメインが異なる。Lift は高い可搬性を持つ高速なコードを記述できることを目指して開発されている言語である。高い可搬性は基本的にインターフェイスではなく内部実装により確保される。Lift で書かれたコードはオートチューニングにより用いるアーキテクチャに最適化されたコードを出力する。Lift の構文の例を図 2.13 に示す。

このコードは 3-point Jacobi Stencil 計算の記述である。このコードは内部的には用いているアーキテクチャ用にオートチューニングされてコード生成が行われる。計算式や最適化手法には触れないが、図 2.13 によって Halide などとは異なるインターフェイスを持っていることが分かる。Lift においては、計算のストリームを組み立て、次のストリームに渡していく形の構文となっている。

2.4 問題点

本研究では、先行研究を参考に High Performance Computing が容易な計算化学用 DSL を設計することを検討する。特に前述の Halide のインターフェイスは可搬性が高くユーザチューニングが容易である。そのため、計算化学分野へと応用することを検討した。しかし、Halide をそのまま計算化学分野に適応させることは困難である。それは主に以下の点に起因

$$Distance(atom1, atom2) = \sqrt{(atom1.x - atom2.x)^2 + (atom1.y - atom2.y)^2}$$

図 2.14. 問題となる数式 1

```
1  distance(mol1, sub1, atom1, mol2, sub2, atom2) =
2    (((sys(mol1,sub1,atom1, “x”)
3          −sys(mol2,sub2,atom2, “x”))**2) +
4     ((sys(mol1,sub1,atom1, “y”)
5         −sys(mol2,sub2,atom2, “y”))**2))**(1/2);
```

図 2.15. 数式 1 を実現する Halide コード

$$\text{SumOfDistance} = \sum_{mol1, mol2 \in sys} \sqrt{(mol1.x - mol2.x)^2 + (mol1.y - mol2.y)^2}$$

図 2.16. 問題となる数式 2

```
1  distance(mol1, mol2) =
2    (((sys(mol1, “x”) − sys(mol2, ” x”)) ** 2) +
3     ((sys(mol1, “y”) − sys(mol2, ” y”)) ** 2)) ** (1 / 2);
4  sumOfDistance = distance(rdom_mol1, rdom_mol2);
```

図 2.17. 数式 2 を実現する Halide コード

する。

第一に、計算化学のデータ構造と計算が複雑な点がある。組み合わせ計算が多いことや、ループネストが深くなることで、計算式を組み立てる時に引数が非常に多くなりがちなのである。例えば、図 2.14 で示す計算式を Halide で記述すると図 2.15 となる。配列 sys の 1 次元目が mol、2 次元目が sub、3 次元目が atom に相当すると仮定する。

図 2.14 は原子間距離 (二次元平面上) を計算する数式である。図 2.15 で示すように、原子の座標データを取得する際に配列 sys に対して上位の次元である分子と Subsystem も引数として渡さなければならない。このように次元数分の引数が必要であるので、ネストの深いデータ構造を用いる場合には可読性が低くなる。同時に、この式は組み合わせ計算であるので関数 distance の引数が多いことも確認できる。n 体計算では n 倍に増えるため、ここでも記述量が多くなる。どの mol とどの sub が atom に対応するのかもユーザが管理しなければならない。さらに、検証のためにデータ構造を変化させ、Subsystem の構造がなくなった際にこのコードは使えない問題もある。

第二に、Reduction 計算が直観的ではない点がある。数式とプログラムの見た目がかけ離れるため、可読性が下がる。例えば、図 2.16 で示す計算式を Halide で記述すると図 2.17 となる。

図 2.16 は、分子間距離 (二次元平面上) を計算して計算系でその総和をとる数式である。図 2.17 で示すように、一度シグマの内部を定義した後に、Reduction 部分を定義する。Reduction は RDom 型と呼ばれる特別な型を用いて表現でき、関数の引数として用いることでその関数が暗黙的に Reduction 計算へと変換される。このように、引数の型によって振る舞いが変わるインターフェイスとなっているので、どこが Reduction であるかが明示的ではない。管理をし易くするために、変数の命名規則をユーザ側に押し付けている形となっている。

第三に、計算化学で用いるデータ構造がイレギュラーな形をしている点がある。Halide は画像処理用の DSL であるため、各次元毎に要素数が決まった矩形構造のデータ構造を想定している。例えば、画像の幅と高さは画像毎に決まった値である。RGB 値を用いるために次元数が増えたとしてもそれは変わらない。そのため、内部実装上でもイレギュラーなデータ構造には対応していない。その上、Halide では最後に realize 関数を呼ぶことで計算の実行が行われるが、その際に画像の定義域を指定する必要がある。画像処理では可能であるが、計算化学においては、その構造の複雑さ・組み合わせ計算による大きさの巨大化によりユーザに記述させることは避けることが望ましい。

第四に、扱える最適化が異なる点がある。例えば高速度計算で頻出する並列化であるが、画像処理のような矩形的データ構造においては仕事量のばらつきは生じにくい。しかし、計算化学のイレギュラーなデータ構造ではロードバランシングの問題が顕著に顕れる。例えば、上述の FMO 計算においては、分子を Subsystem で区切った場合に SubSystem 毎に Atom の数が異なるため、仕事量に差が生じる。これを解決する手法として、work stealing などの手法が存在するが、ネストが深いループでの適応は困難である。計算化学用の DSL では、ロードバランシングを調整できる機能が必要となる。加えて、キャッシュヒットの最適化も異なる。画像処理で扱うデータは固定長配列におさまるデータ構造である。それはタイリングなどの手法を用いることでキャッシュヒットを向上できる。しかし、イレギュラーなデータ構造では可変長な配列にならざるを得ないので、そのような技法を用いることは出来ない。

また、Halide の brute force な loop fusion 手法や PolyMage で使われているような動的計画法によるオートチューニングも適用することは難しい。何故ならば、イレギュラーなデータ構造を用いるために、内部要素全てを考慮する際に組み合わせ爆発が生じてしまうからである。現在の既存研究では、この内部構造も考慮したオートチューニング技法はあまり知られていない。

Lift の計算ストリームを組み立てる形を計算化学に用いる場合も問題点がある。まず、map や reduce などを用いて計算式を組み立てる形式は現実の計算式と乖離が激しい。それは、map などは配列のマッピングなどのコンピュータアーキテクチャに根ざした使われ方をされるからであり、計算化学の数式には現れない。これはコード実装におけるミスや遅延を誘発する。化学者に向けてのユーザビリティを考えると現実の計算式に近いコードを書ける方が望ましい。

PolyMage と Lift の両者に言えることであるが、化学分野で使われている C++ や Fortran とは異なる構文であることが明らかである。普段用いている言語と異なる構文は学習コストを増加させるため、DSL の使用を躊躇させる要因となる。DSL の使用促進の観点から普段用い

る言語と近い構文を持つことが望ましい。

第 3 章

計算化学に向けたドメイン特化言語の提案

本研究では、Halide における可搬性の高いインターフェイスを計算化学分野に適応することを計画した。しかし、前章で説明した問題点からそれは困難である。そこで、計算化学におけるデータ構造と計算理論を検証し、新たな言語デザインを開発する。

本研究では、計算化学に向けたユーザチューニング容易な DSL のための新たな型として Rank 型を提案する。3.1 節では、開発する DSL の基本方針について述べる。3.2 節では、提案する Rank 型の概念について述べる。3.3 節では、Reduction 計算の明示化について述べる。3.4 節では、提案した型と記法を取り入れた DSL デザインについて述べる。3.5 節では、既存デザインとの比較を行う。

3.1 DSL の基本方針

本節では、計算化学に向けた DSL 開発の基本方針について述べる。

前章で解説したように、Halide の形式をそのまま計算化学に応用することは困難である。そこで、計算化学用 DSL として望ましい要件を以下に挙げた。

- 化学の階層的でイレギュラーなデータ構造に適用できる。
- データ構造を直感的に取り扱うことができる。
- 実際の計算式に近い記述が可能である。
- 組合せ爆発や深いネストのループを簡便に記述できる。
- 明示的な最適化が可能である。

これらの要件は、大きく二つに分類できる。

第一に、内部実装上の要件に分類できる。前述の要件の 1 点目はこの分類である。Halide は、画像処理に特化させているために矩形的データ構造にしか対応していない。しかし、内部実装を変更することでインターフェイスの流用は可能である。当然用いることができる最適化は異なる。

第二に、言語インターフェイスに向けた要件に分類できる。前述の要件の 2~5 点目はこの分類である。順番に解説する。2 点目のデータ構造を直感的に取り扱うことができるとは、メンテナンス性にまつわる要件である。Halide や最適化された汎用言語のコードに共通するが、データの所在は明確化されていない。高速なコードの多くに共通するが、実体のあるデータは単一の配列に格納される。Halide でも、Func 型を一種の配列とみなすことで、同様の構造が見える。これでは、階層構造を持つ場合に直感的な形とならない。例えばある原子の座標はその原子からアクセスされるべきであるが、原子は配列の引数として用いられるのみである。この場合、座標はあたかも上位階層のデータであるかのようにアクセスされてしまう。したがって、計算化学で用いる階層的なデータ構造を抽象化した型が必要となる。3 点目の実際の計算式に近い記述が可能であるとは、コードの書きやすさと可読性にまつわる要件である。ユーザが実装するタイミングで考えると、実際の計算式に近い記述ができことが望ましい。この点は基本的な演算子をオーバーロードして記述をさせる Halide は概ね満たす。しかし、RDom 型を用いる Reduction 計算は実際の計算式とは異なる。暗黙的な変換がされるために、ユーザが直観的に判別することができない。計算式上でシグマ記号が使われる場合には、プログラムでも似た形で書けるべきである。よって、特に Reduction 計算を表す構文は工夫する必要がある。4 点目の組合せ爆発や深いネストのループを簡便に記述できるとは、可読性と可搬性の要件である。Halide や最適化された汎用言語のコードでは、用いる変数の数が多くなりがちである。関数の引数としてデータ構造全ての引数を必要とすると、可読性が著しく下がる。さらに、計算理論やデータ構造の変更に対応できない。5 点目の明示的な最適化が可能であるとは、チューニングの容易さに関する要件である。複雑な計算式において、多数の変数の中から特定のオブジェクトを指定することは難しい。特に、Halide の Var 型は計算毎に異なる役割を持つことも可能であり、ユーザは計算式ごとの変数の役割を把握していなければならない。加えて、reduction 計算においても同様であり、複数の reduction を単一の式に記述することを許しているので、どこで畳み込むかの明示化が難しい。

3.2　Rank 型

本節では、提案手法の骨子となる Rank 型のアイディアについて述べる。

3.2.1　Rank 型の概念

Halide 形式の DSL では、realize 関数を用いるまでは、実際の計算は行われない。しかし、データの読み込みは最初に行い、それを元に計算アルゴリズムを組み立てる。つまり、ユーザおよび計算式にとって用いるデータ構造は既知である。その点から、データ構造そのものをコードに取り入れることを考案した。計算化学においては、階層的でイレギュラーな形式のデータ構造になる。そのデータ構造と、用いる計算式を示す抽象機構が Rank 型である。これは汎用言語でのポインタに相当する。Rank 型により、階層的になっているデータ構造の部分構造を直接示すことが可能で、より上位に位置する階層を省略して指定することができる。例

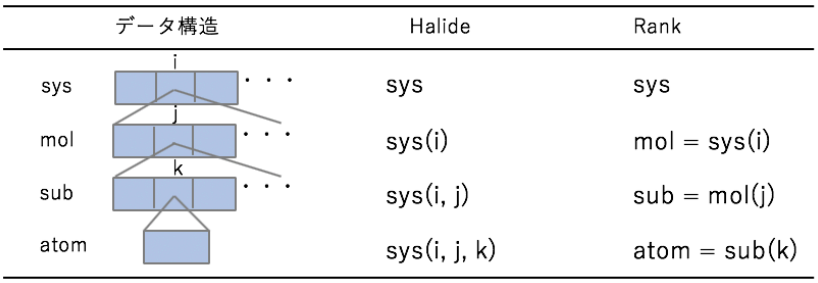

図 3.1. Rank と実データの対応関係

```
1  Func calc;
2  Data data = read_input("data.dat");
3  Rank mol, sub, atom;
4
5  data.has(mol.has(sub.has(atom)));
```

図 3.2. 変数宣言時

```
1  calc(atom1, atom2) = atom1.attr(0) + atom2.attr(0);
```

図 3.3. Rank を用いたデータアクセス

えば、図 3.1 のような対応関係が構成される。

sys - mol - sub - atom の階層的配列が存在してそれぞれ i, j, k のインデックスをしようする場合に、halide では sys 配列にインデックスを用いて指定した次元の配列にアクセスするものであった。Rank 型の導入により、それぞれの階層に変数名を当てはめることで上位階層を暗黙的に取り込むことができる。

使い方を疑似コードと共に説明する。例えば、Molecule - Subsystem - Atom の階層構造を持ったデータが存在する場合を考える。C++ では変数宣言が必要であるが、その時に追加で Rank 型の構築を行う。これを図 3.2 に示す。

図 3.2 の 1 ～3 行目は通常の C++ と同様である。*read_input* 関数でデータを読み込んでいることにより、この時点でデータ構造はユーザとコンパイラから既知の情報である。従って、5 行目において Rank の構築ができる。mol は Molecule に、sub は Subsystem に、atom は Atom の階層に紐づけられる。下位の階層は上位階層の情報を保有している。これにより、以後の計算式では Rank 型を上位の階層情報も含めた変数として用いることができる。つまり、計算化学のデータ構造、計算式に合わせた抽象機構である Rank 型は、Halide 形式のインターフェイスにポインタを導入している。

次に、Halide のように計算式を構築する場合は Rank 型を図 3.3 のように用いることができる。

図 3.3 のように、それぞれの階層が持っている要素に attr 関数を用いてアクセスすることが

$$\sum_{mol1,\, mol2 \in sys} \sqrt{(mol1.x - mol2.x)^2 + (mol1.y - mol2.y)^2}$$

図 3.4. Reduction 計算のサンプル

できる。これは、atom の 0 番目の要素にアクセスする例を示している。ここで重要なのは、実データがどこに属しているかを明示化している点にある。例えば、同様の役割を持つ Halide の形式と比較した場合を表 3.1 に示す。

表 3.1. Rank との比較

Rank 形式	Halide 形式
mol.attr(0)	sys(mol, 0)
atom.attr(0)	sys(mol, sub, atom, 0)

前述した通りに、Halide 形式では上位階層の引数も必要であるほか、第三引数に要素アクセス用の整数を入れるのかさらに下の階層を参照するのかが明確でない。加えて、Halide の記述では sys の要素にアクセスしている形に見える。Rank 型を用いることで、どの階層のデータにアクセスするかが明示的となっている上、ミスの要因も減少させている。これは、汎用言語で言う所のクラスの役割である。高速計算のために使われないことも多いクラスであるが、JIT コンパイルを行う本形式においては、インターフェイスとして用いる分には問題がない。つまり Rank 型は階層構造を持ったクラスの役割も持ち合わせている。

上記のように、Rank 型は Halide のような user adjustable な言語デザインにポインタやクラスを導入している。

3.3　Reduction 計算の明示化

本節では、Reduction 計算を行う際に取り入れるべき形式について述べる。

Halide では、RDom 型を用いて Reduction を表現するが、ユーザにとって明示的でない問題点が存在した。その問題点の解決のため、Reduction 計算を関数としてユーザが明示的に用いるデザインの導入を提案する。Reduction を行う際には、必ず一文を用いて記述する規則を取り入れる。図 3.4 で示す計算を記述する際の具体例を図 3.5 で示す。なお、この例は Rank 型の導入を前提とする。

図 3.4 は、分子の座標 (例えば重心座標、二次元平面) の距離を全分子の組み合わせで計算し、足し合わせる式である。本研究で提案する記述法では、$\sum$ の内部を一つの関数で定義した後にその総和を別の行で定義する。3.5 の 1 行目は $\sum$ の内部の計算式であり、Rank 型を用いて数式をそのままプログラムに記述している。5 行目は $\sum$ 計算を表す式であり、

```
1  distance(mol1, mol2) =
2    ((mol1.attr( “x” )− mol2.attr( “x” ))** 2 +
3     (mol1.attr( “y” )− mol2.attr( “y” ))** 2) ** (1 / 2);
4
5  sumOfDistance = sigma(distance, mol1(sys), mol2(sys));
```

図 3.5. Reduction 計算明示化のサンプル

```
1  int main() {
2    Func calc;
3    Rank mol, atom;
4    Data sys, ret;
5    sys.read_input("data.dat");
6    sys.has(mol.has(atom)));
7
8    calc(mol) = atom(mol).attr(0);
9
10   calc.mp_parallel();
11
12   ret = calc.realize();
13   return 0;
14 }
```

図 3.6. Rank 型を用いる DSL のコードサンプル

sumOfDistance 変数の計算式として sigma 関数を用いている。

3.4 DSL デザイン

本節では、Rank 型を取り入れた user adjustable な DSL の言語デザインを提案する。Rank 型の提案自体が Halide の言語デザインに基づいたものであるため、Rank を用いる DSL は Halide の形式に則る。ユーザが主に用いる型は三種類であり、それぞれ Func、Rank、Data と呼ばれる。Func は Halide と同様に計算式を表現するための型である。Rank は前節で提案した通りに階層構造を含めたポインタおよびクラス的な振る舞いができる型である。Data はインプットファイルなどの実体のあるデータを表現する型である。Rank、Data はそれぞれ Halide の Var、Buffer 型に対応している。Halide と比べて、Expr や RDom はユーザが用いることはできない。Expr は内部実装として用いることはできても、ユーザの取り扱える形にしない方が良い。Halide では、Func の計算式定義と同様な形で Expr の定義もできる。しかし、これを許すと違う役割を持つ計算式定義が二種類存在することになり、ユーザの混乱を招く自由度を下げる代わりにコードを書きやすくする DSL の基本方針である。Reduction 計算は前節で述べた通りに特別な関数に一文を用いる形式とする。本 DSL デザインでのインターフェイスは図 3.6 のようになる。

本インターフェイスは大きく四つのパートに別れている。それぞれ、変数宣言部、アルゴリ

$$\mathrm{SumOfDistance(mol)} = \sum_{sub1,sub2 \in mol} \sum_{atom1,atom2 \in sub1,sub2} (atom1.x + atom2.x)$$

図 3.7. 比較するための計算式

ズム部、スケジューリング部、実行部である。2∼4 行目は C++ と同様の変数宣言部である。5、6 行目は本 DSL のための設定部分であり、特徴的なものとなる。5 行目で決められた形式のデータファイルを読み込む。この時点で、コンパイラは計算対象のデータ構造を理解できるようになる。6 行目で has 関数を用いてを階層構造を Rank 型に記憶させる。本例においては、input のデータは二階層であり、上位階層を mol、下位階層を atom の変数に対応づけられた。8 行目の計算式定義部分では、Func 型変数の () オペレータの引数に Rank 型を用いて記述させる。=オペレータの左辺値に使われた Rank 型は右辺値にも用いることができる。さらに、その Rank 型の下位に当たる変数 (mol にとっての atom) も用いることができる。Rank 型変数内の各要素へのアクセスは、下位の Rank 型もしくは attr 関数を用いて行うことができる。この例では attr(0) とすることで、例えば x 座標へのアクセスとしている。10 行目のスケジューリング部では、計算機のための最適化命令を行う部分であり、Func 型や Rank 型のメンバ関数を呼び出す形で最適化する。最後に Func 型の realize 関数を使用することで計算の実行が行われる。

3.5　既存デザインとの比較

Halide では、そのインターフェイスを計算化学に適応させることは困難である。実際には内部実装上用いることはできないが、同デザインで実装のみを変更したと仮定して Rank 型を用いる形と比較を行う。本 DSL は計算化学が対象ドメインであるため、化学で用いる計算とする。簡単のため、分子が Subsystem によって分割され、その中に原子があるデータ構造に関して、全原子間の x 座標 (0 番目の要素) を分子毎に総和を計算することとする。その数式を図 3.7 に示す。

Rank 型を使った場合のコードを図 3.8 に、Halide によるコードを図 3.9 に、それと等価な C++ コードを 3.10 に示す。

Rank 型を取り入れた場合と Halide を比較すると、何点か相違点が挙げられる。

第一に、引数が異なっている。distance を定義する際に、左辺値として Rank 型形式では二つ、Halide では五つの引数が必要となる。これは Rank 型は上位の階層情報も保持しているためであり、atom へのアクセスに mol や sub を必要としていない。この例ではネストは浅いが、これが実際に用いるような 10 次元以上のネストになると影響が大きくなる。Halide の形式では全ての階層を引数として渡す必要があるため、ネストの深さの分だけ引数が多くなる。Rank 型を用いることで該当階層の変数のみを引数として渡せば良くなる。右辺値においても同様で、Func 型の利用時とデータアクセスの引数の数が異なる。このように、Rank 型による

```
1  int main() {
2    Func distance, sumOfDistanceSub, sumOfDistanceMol;
3    Rank mol, sub, sub1, sub2, atom1, atom2;
4    Data sys, ret;
5
6    sys.read_input("data.dat");
7    sys.has(mol.has(sub.has(atom)));
8    sub.is(sub1).is(sub2);
9    atom.is(atom1).is(atom2);
10
11   distance(atom1, atom2) = atom1.attr(0) + atom2.attr(0);
12   sumOfDistanceSub(sub1, sub2) =
13     sigma(distance, atom1(sub1), atom2(sub2));
14   sumOfDistanceMol(mol) =
15     sigma(inner, sub1(mol), sub2(mol));
16
17   ret = sumofDistanceMol.realize();
18   return 0;
19 }
```

図 3.8. Rank 型を用いる場合のコード

```
1  int main() {
2    Func outer, inner;
3    Var mol, sub, sub1, sub2, atom;
4    Buffer<double> sys, ret;
5    RDom rmol, rsub1, rsub2;
6
7    sys.read_input("data.dat");
8
9    distance(mol, sub1, atom1, sub2, atom2) =
10     sys(mol, sub1, atom1, 0) + sys(mol, sub2, atom2, 0);
11   sumOfDistance(mol) =
12     distance(mol, rdom_sub1, rdom_atom1, rdom_sub2, rdom_atom2);
13
14   ret = sumOfDistance.realize();
15   return 0;
16 }
```

図 3.9. Halide のコード

可読性の向上が分かる。Rank 型の形式はコードの可搬性も向上させている。例えば、計算理論を変えて上位のデータ構造が変わった場合に、Halide では distance 関数を定義し直す必要がある。一方で、Rank 型の計算式定義には上位階層の変数を用いる必要がないため、has 関数を用いた設定部のみの変更で対応できる。これは特に計算化学上では重要であり、前述した FMO 計算のように、概ねのデータ構造は他と同様でも途中に高速化用の階層を用意することがあるためである。

第二に、Reduction 計算の明瞭化がなされている。Halide では、RDom 型を用いて Reduc-

```
1  int main() {
2    double sum = 0;
3    for(int mol = 0; mol < mol_size; mol++){
4      for(int sub1 = 0; sub1 < sub_size[mol]; sub1++){
5      for(int sub2 = 0; sub2 < sub_size[mol]; sub2++){
6        for(int atom1 = 0; atom1 < atom_size[mol][sub1]; atom1++){
7        for(int atom2 = 0; atom2 < atom_size[mol][sub2]; atom2++){
8          sum += data[mol][sub1][atom1][0] + data[mol][sub2][atom2][0];
9        }
10       }
11     }
12     }
13   }
14   return 0;
15 }
```

図 3.10. 等価な C++ コード

tion を表現するが、RDom 型は Var 型と同様の使われ方をするために、二つの型を混同してしまう。これはミスの要因となる。本 DSL においては、RDom 型は存在せず、sigma 関数を用いて表現するために Reduction 部分が明確である。さらに、Halide の形式では RDom が二つ以上使われた際にどのループに対して畳み込みが明瞭でない。本 DSL では、Reduction に一文を用いてさらに Rank 型の指定が可能であるためにループと畳み込みの対応関係が確定する。これは計算化学で良く用いられる組み合わせ計算を行う時により顕著である。単一の計算式上に複数の同階層の変数が出現するため、Reduction の明瞭化が必要なのである。加えて、実際の数式上でシグマを用いるため、本 DSL の形式の方が数式に近い記述ができると考えられる。

第三に、データの所在が明示化されている。前述した通りに、Rank 型のクラス的振る舞いの効果である。本件でアクセスするのは Atom の 0 番目の要素である。そのアクセス手法が Rank 型を用いる場合と Halide の場合で異なっている。Rank 型では変数 atom のメンバ関数である attr の引数に番号を指定してアクセスする。一方で Halide では、data に上位階層も含めた変数を埋め込み、最後の引数として番号を指定する。これは記述の簡略化以外の役割を持っており、データの所在が明示化されていることを意味している。atom の x 座標は当然 atom に依存する要素であり、atom のメンバ変数としてアクセスするのが自然である。しかし、Halide の形式ではあたかも data の要素としての記述となっている。これは、Rank 型を導入することで、クラス構造に代表されるユーザが理解しやすい形でデータを構成する役割を持っている。

等価な C++ コードと比較すると、for 文を冗長に書く必要がなくなっていることが分かる。これは画像処理でも同様であったが、最大の相違点は要素アクセスの配列を書かなくて良い点である。C++ の for 文で、上限値に配列を用いることから分かるように、イレギュラータイプのデータ構造では、配列内配列の要素数は、上位配列の要素によって異なる。この記述にはミスが生じやすく、ネストが深い場合にはより顕著である。同様に、最適化命令 (例えば

#pragma omp parallel) も for 文と入れ混ざって記述する必要がないために、コードの最適化が容易である。

このように、Halide の Func、Var、Buffer を用いる形式ではイレギュラーなデータ構造でループの深いネストを持つ計算化学への適用の際に冗長になり過ぎてしまう。本研究で提案した Rank 型を用いることで、計算化学コードにおける可搬性、可読性を向上させており、ユーザビリティの向上が期待される。

第 4 章

実装

前章で提案した Rank 型を用いることができる DSL をフルスクラッチで開発する。ホスト言語は計算化学分野で広く用られる C++ であり、出力言語も同様とする。様々な環境で用いることを想定して C++ の STL のみを用いて実装する。実装では Deep Embedding を用いた手法で行う。アルゴリズム部の記述には計算式を表す Func 型に抽象構文木 (Abstruct Syntax Tree, AST) を代入する。スケジューリング部の記述は AST に対する最適化命令であり、最後にコンパイル用関数を呼ぶことで最適化された C++ コードを生成・実行する。4.1 節では、本 DSL のシステム構成について説明する。4.2 節では、本 DSL の構文規則と機能について説明する。4.3 節では、本 DSL に関する制限を説明する。

4.1 システム構成

4.1.1 概形

本 DSL のシステム構成を図 4.1 に示す。

本システムは三段階のコンパイル形式を採用している。まず、ユーザは C++ コードの一部のみを本 DSL で記述する。コードは C++ のコンパイラによってコンパイルをして実行ファイルが出力される。実行時に、計算式に基づいた AST を取得、ユーザが記述した並列化などの最適化命令を元に、独自のコンパイラによって C++ のライブラリへと変換する。ライブラリは C++ のコンパイラでダイナミックリンクライブラリが生成されるようにコンパイルを行

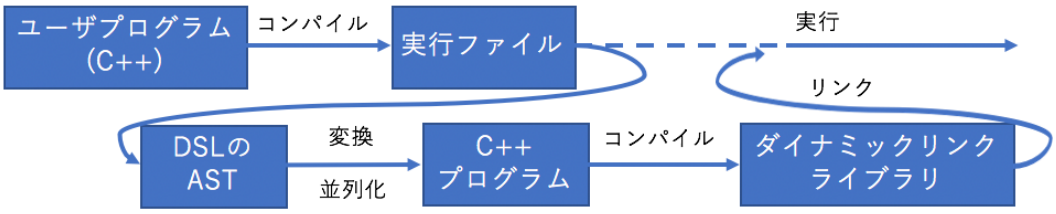

図 4.1. システム構成

い、実行中のファイルにダイナミックリンクをして計算結果を取得する。なお、コード生成環境と実行環境が異なるケースが想定されるため、ダイナミックリンクをせずに生成のみを行うサポートも取り入れる。このシステムにより、独自コンパイラを工夫することで、データ構造・計算アルゴリズムを考慮した最適化が可能となる。

4.1.2 内部実装

次にデータの実体に関する内部実装であるが、計算化学で用いるイレギュラーなデータ構造に対応するために Data 型の内部にデータの実体格納用配列および下位階層の要素数を保持している。アクセススピードを考えると int もしくは double 型の配列に全ての内部要素を入れる方が速い。しかし、計算実行時には別の形式で保存し直すために、JIT コンパイル前はそのような実装になっている。データの実体格納用配列には計算式上で attr 関数を用いてアクセスできるデータが格納される。

4.1.3 文法定義

本 DSL の記法を表 4.1 にまとめる。

本 DSL では、Func・Rank・Data 型および数字と文字列をユーザが露わに用いる。プログラムは四つの部分で構成される。

1. 宣言 (declaration)
2. アルゴリズム (algorithm)
3. スケジューリング (scheduling)
4. 実行 (realize)

宣言部は、型名と変数名および Rank 構築用関数からなる。変数宣言は通常の C++ コードと同様に行う。その後、has 関数と is 関数を用いて Rank を構築する。アルゴリズム部は Func の定義式の集まりからなる。Func の () オペレータの引数に Rank 型を用いて関数の引数とする。定義式は=オペレータの右辺値に記述する。式は項 term と +・-・*・/演算子、または sigma 構文からなる。term は Rank を引数にとった Func と Rank の attr 関数を用いた形からなる。sigma 構文は第一引数に Func、第二引数以降に Rank を指定する。演算子と同時に扱うことはできない。スケジューリング部は Func の最適化関数の集まりからなる。最適化関数の引数に条件を入れることで指定の Func が最適化される。実行部は、Data と Func の realize 関数からなる。

4.2 機能

本 DSL は高速度計算が可能なコードの生成を目指しているため、並列化命令の実装がなされている。計算化学でも扱われるハイブリッド並列への対応のため、MPI 並列と MP 並列の最適化命令を用意した。並列化コードのスニペットを、図 4.2 に、C++ でのハイブリッド並

表 4.1. 本 DSL の記法

digit ::= (”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”)+
digits ::= digit | digit ”.” digit
operator ::= ”+” — ”-” — ”*” — ”/”
type ::= ”Func” | ”Data” | ”Rank” | ”int” | ”double”
program ::= (declaration | algorithm_definition | scheduling_definition | realize_statement)
declaration ::= (type ID | rank_construct) ”;”
rank_construct ::= (has_statement | is_statement) ”;”
has_statement ::= (dataID | rankID) (”.has” ”(” (has_ statement | rankID) ”)”) ”;”
is_statement::= rankID (”.is” ”(” rankID ”)”)+ ”;”
algorithm_definition ::= assignment ”;”
assignment ::= func_statement ”=” expr
func_statement ::= funkID (”(” rankIDs ”)”)?
expr ::= $\sum$ term | sigma_statement
term ::= $\prod$ (func_statement | rank_statement) | expr | expr operator expr
rank_statement ::= rankID “.” “attr” “(” digits “)”
sigma_statement ::= “sigma” “(” funkID (“,” rankID “(” parentRankID “)”)+ “)”
rankIDs ::= rankID | rankID ”,” rankIDs
parentRankID ∈ rankIDs
scheduling_definition ::= funkID ”.” optimizeMethod “(” conditions “)” ”;”
conditions = condition | condition “,” conditions
condition ::= digits | funcID | rankID
realize ::= dataID ”=” funkID ”.realize()”

列化コードを図 4.3 に示す。なお、実際にはクラスを用いているため、出力されるコードは異なるが、簡単のために一般的な配列を用いて紹介する。この配列はイレギュラー型のデータ構造に対応しているものとする。

```
1  distance(sub1, sub2) = atom(sub1).attr(0) + atom(sub2).attr(0);
2  sumOfDistance(mol) = sigma(distance, mol(sub1), mol(sub2));
3
4  outer.mpi_parallel(4);
5  inner.mp_parallel();
```

図 4.2. 並列化コード生成用の記述

```
1  MPI_Init(&argc, &argv);
2  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
3
4  #pragma omp parallel for reduction(+:my)
5  for(int mol = mol_size / 4 * my_rank;
6        mol < mol_size/ 4 * (my_rank + 1); mol++){
7    for(int sub1 = 0; sub1 < sub_size[mol]; sub1++){
8      for(int sub2 = 0; sub2 < sub_size[mol]; sub2++){
9        for(int atom1 = 0; atom1 < atom_size[mol][sub1]; atom1++){
10         for(int atom2 = 0; atom2 < atom_size[mol][sub2]; atom2++){
11           my += data[mol][sub1][atom1][0] +
12                   data[mol][sub2][atom2][0];
13         }
14       }
15     }
16   }
17 }
18 MPI_Reduce(my, sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
19 MPI_Finalize();
```

図 4.3. C++ ハイブリッド並列化コード

4、5 行目のスケジューリング部に書かれているように、mpi_parallel 関数と mp_parallel 関数を呼び出しているのが並列化命令である。最初に C++ コンパイラが動作する際にはアルゴリズム部を元に AST が生成される。スケジューリング部の並列化命令に基づいて outer と inner に相当するノードに並列化命令が組み込まれる。次に、本 DSL のコンパイラが動作する際に該当ループを並列化したコード生成が果たされる。これにより、スケジューリング部の 2 行のみで outer に関しては MPI 並列と inner に関しては MP 並列のハイブリッド並列が実現できる。C++ のコードと比べても、記述が簡便であることが分かる。仮に outer の方を MP 並列化したい場合には、このスケジューリング部のみの変更で対応が可能である。

4.3 制限

本節では、Rank 型を用いた本 DSL における制限を述べる。

Halide との相違点でもあるが、本 DSL は対象ドメインの都合上 Rank 型を直接計算式に用いることを想定していない。Func 型の引数として使用するか、attr 関数を用いた形にすることしかできない。例えばある原子がある分子内配列の何番目に相当するかの情報を取り出すこ

とはできない。Halide では、Var 型を直接計算式に用いることで、その Var が示す番号を取得することができた。これは対象ドメインの都合上である。計算化学の場合は、立体的な位置情報はそのクラス変数として保存される。配列にされた場合の隣同士は立体的な情報とは無関係であり、配列としての隣を抜き出す必要がない。一方、画像処理においては配列の隣が画像にとっての隣であり、引数が持っている情報量が異なっているのである。

Func 型の () 演算子の引数として、Rank 型以外を用いることはできない。Halide においては、同様の引数に Expr 型を指定しており、int 型を用いることも可能であった。これはステンシル計算などを用いる画像処理においては必要な機能であるが、上記の通りにデータ構造上の隣に意味がないドメインでは不要である。

本質的な制限ではないが、本 DSL では静的な負荷分散のみを実装している。オートチューニング機能はない。並列化のオートチューニングのためには、イレギュラーなデータ構造に由来して仕事量が異なるプロセッサに対し、データの内部構造も考慮してコードを生成できるアルゴリズムが必要となる。現在そのような対象に適した手法はあまり知られていない。チューニングアルゴリズムの開発が必要である。

第 5 章

実験

本研究では、開発した DSL について検証するために実験を行なった。本章では行なった実験の内容とその結果について述べる。また、実装が異なるために参考程度であるが、Halide を用いた実験も示す。

5.1 コードコンパイルの Halide との比較

本研究の実験では、表 5.1 に示すハードウェアを用いた。

表 5.1. 実験に用いたハードウェア情報 1

	計算機
CPU	Intel Core i7
CPU クロック周波数	3.5 GHz
CPU コア数	2
CPU スレッド数	4

また、実験に用いたソフトウェアのバージョン情報を表 5.2 に示す。

表 5.2. 実験に用いたソフトウェア情報

ソフトウェア	バージョン
OS	macOS High Sierra 10.13.6
GNU コンパイラ (g++)	4.2.1
LLVM / Clang (clang++)	10.0.0

開発した DSL ではデフォルトで GNU コンパイラを用いて JIT コンパイルを行う。しかし、Halide ではバックエンドに LLVM を用いるので比較のために clang コンパイラにも対応させて実験を行った。

図 5.1 で示すコードのコンパイルを行なった。

```
1  #include "dsl.h"
2  #include <sys/time.h>
3  #include <sys/types.h>
4  using namespace DSL;
5
6  int main()
7  {
8      Func calc("calc");
9      Data dat("dat"), out("out");
10     Rank a("a"), b("b");
11
12     dat.read_input("test_data.dat");
13     dat.has(a.has(b));
14
15     calc(a) = b(a).attr(0) + b(a).attr(1);
16
17     calc.set_print(true);
18     std::cout << "calc time start" << std::endl;
19
20     struct timeval s, e;
21
22     gettimeofday(&s, NULL);
23     out = calc.realize();
24     gettimeofday(&e, NULL);
25
26     printf("time = %lf\n",
27             (e.tv_sec - s.tv_sec) +
28             (e.tv_usec - s.tv_usec) * 1.0E-6);
29
30     out.print_data();
31
32     return 0;
33 }
```

図 5.1. 簡易コード

15 行目に記述されるように、内部の 0 番目、1 番目の要素にアクセスするのみのコードである。

一方で、参考として Halide には図 5.2 で示されるコードのコンパイルを行なった。

```
1  #include "Halide.h"
2  #include <stdio.h>
3  #include <sys/time.h>
4  #include <sys/types.h>
5  using namespace Halide;
6
7  int main(int argc, char **argv) {
8
9      Func gradient("gradient");
10     Var x("x"), y("y");
11     gradient(x, y) = x + y;
12
13     struct timeval s, e;
14
15     gettimeofday(&s, NULL);
16     Buffer<int> output = gradient.realize(8, 8);
17     gettimeofday(&e, NULL);
18
19     printf("time = %lf\n",
20             (e.tv_sec - s.tv_sec) +
21             (e.tv_usec - s.tv_usec) * 1.0E-6);
22
23     return 0;
24 }
```

図 5.2. Halide の簡易コード

それぞれの 26 行目・19 行目の printf 関数によって実行時間の検証を行う。対象はそれぞれの realize 関数の実行である。これによりそれぞれの DSL の独自コンパイラおよびバックエンドコンパイラの実行時間、計算時間が得られる。これを 100 回繰り返し、その平均を取ることで結果とする。コンパイル時間を表 5.3 に示す。なお、内部実装が異なる都合上計算対象も異なるので、あくまで参考程度の記録である。

表 5.3. 実行時間

	実行時間
本 DSL	1.94 sec
Halide	0.503 sec

本 DSL の方が遅い結果となっている。この主要因としては、本 DSL では、ファイル出力によるデータの受け渡しを行い、別プロセスを立ち上げて計算を行わせることに由来すると考えられる。これは、計算化学の対象が大規模系になるにつれ、全体の計算時間に対する JIT コンパイル時間が短くなることを想定しての実装であるが、今回のように簡易的なコードにおいては、その差が顕著に表れた。

$$\text{SumOfDistance} = \sum_{mol \in sys} \sum_{atom1, atom2 \in mol} \sqrt{(atom1.x - atom2.x)^2 + (atom1.y - atom2.y)^2}$$

図 5.3. 計算式

5.2　並列化

本 DSL の実用性を示すため、並列化の実験を行なった。実験では、スケジューリング部分の最適化関数のみを書き換えて異なる並列化コードの生成を行なった。本 DSL は 3 段階のコンパイル方式を採用しており、最初に本 DSL を用いて記述されたコードを C++ コードに変換する際は表 5.2 の環境で行なった。生成したコードのコンパイルおよび実行は表 5.4 の環境で行なった。

表 5.4. 実験に用いたソフトウェア情報 2

ソフトウェア	バージョン
OS	Red Hat Enterprise Linux 7、CentOS 7
Intel コンパイラ (icpc)	18.0.1

用いたハードウェア情報は表 5.5 に示す。これは東京大学情報基盤センターの Oakforest-PACS スーパーコンピュータシステムである。

表 5.5. 実験に用いたハードウェア情報 2

	計算機
マシン名	Fujitsu PRIMERGY CX1640 M1
CPU	Intel Xeon Phi 7250
CPU クロック周波数	1.4 GHz
CPU コア数	68
理論演算性能	3.0464 TFLOPS
Memory	96 GB(DDR4) + 16 GB(MCDRAM)
インターコネクト	Intel Omni-Path ネットワーク (100 Gbps)

計算式は 5.3 に示す。多数の分子がある環境で、分子内の原子間距離の総和を計算し、最後に全分子でリダクションする。なお、実際の計算化学の計算式とは異なるが、試験として簡単な記述にしてある。計算対象は、イレギュラーなデータ構造とするため、一分子の原子数が極端に多く、他の分子は共通の原子数とする。生成したコードは逐次実行・MP 並列・MPI 並列・ハイブリッド並列 (MP + MPI)・ロードバランシングの五種類である。逐次実行コード

は、スケジューリング部の記述を行わず、DSL がデフォルトで出力するコードを用いている。MP 並列コードは、スケジューリング部に mp_parallel 関数を用いて OpenMP の共有メモリ型並列化がされている。MPI 並列コードは、mpi_parallel 関数を用いて MPI の分散メモリ型並列化がされている。ハイブリッド並列は、MP 並列と MPI 並列の組み合わせであり、外側のループに対して MPI 並列が行われ、内側のループに対して MP 並列が行われる。ロードバランシングでは、配列を並列化する際に計算量が多い (配列の要素数が多い) 部分のみを取り出しループを分割する。計算量が多い部分は全プロセッサで計算させることで高速化を図っている。計算対象及び使用プロセス数・スレッド数を表 5.6 に示す。

表 5.6. 計算対象

計算名	プロセス数	スレッド数	一分子の原子数	他分子の原子数	他分子の分子数
task1	4	4	1000000	1000	16
task2	4	4	1000000	100000	16
task3	2	8	1000000	1000	16
task4	2	8	100000	10000	2048
task5	2	8	100000	10000	512
task6	2	8	100000	10000	1024
task7	2	8	200000	10000	16
task8	2	8	400000	10000	16
task9	2	8	600000	10000	16
task10	2	8	100000	10000	16

実験結果を図 5.4 に示す。なお、各計算で計算量の違いから計算時間が大きく異なる。データをまとめるために、逐次実行を 1 とする正規化を行なっている。

図 5.4 では、色が task に対応しており、同じ色を見ることで並列化の種類による違いを見ることができる。原子数やスレッド数の変更により、最適化効率が変化していることが分かる。task4・task5・task6 で並列化コードが大きく高速化されていることから分かる通り、分散する仕事量が多い場合に並列化の効果が表れる。ほとんどのケースで mpi 並列が mp 並列よりも遅い結果が得られているが、計算式が簡単なため、分散に対するオーバーヘッドの比重が大きいことが考えられる。ロードバランシングでは、計算量が多い部分に対して集中的にプロセッサを用いるため、配列毎に計算量差が大きいケースでは高速化の度合いが大きくなる。ほとんどの計算で他のコードよりも高速に実行できる。この実験で重要な事項は、それぞれの最適化を 2, 3 行のコード変更のみで生成可能である点であり、計算対象と計算機に応じた比較実験が簡単に実行可能である。

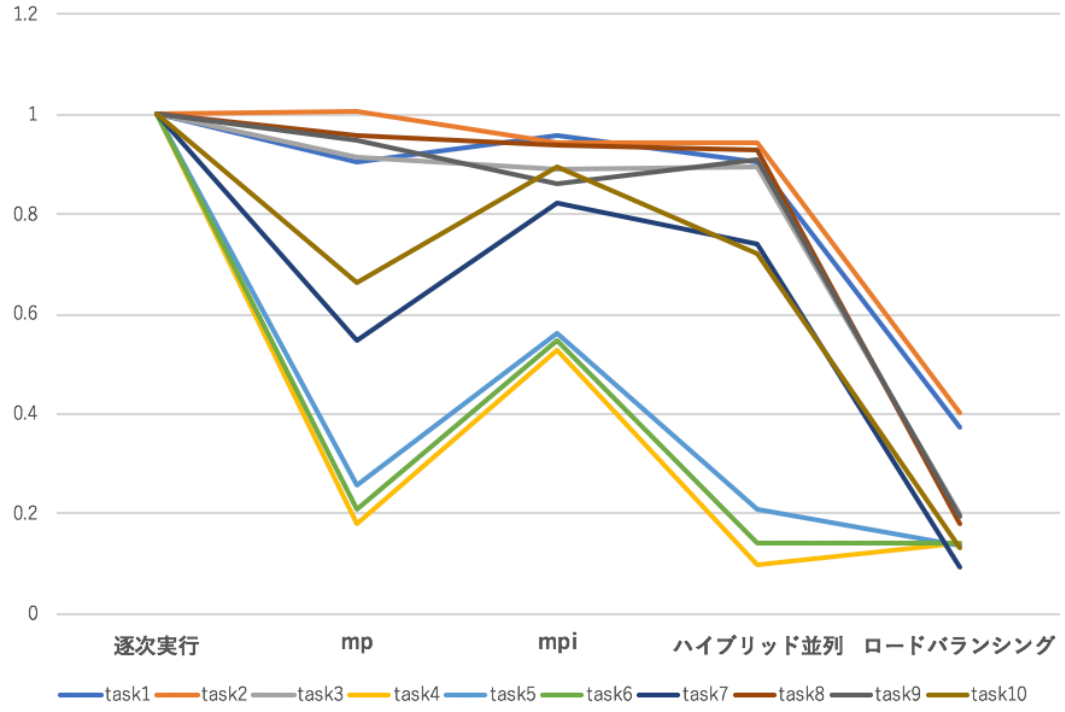
1.2
1
0.8
0.6
0.4
0.2
0
逐次実行
mp
mpi
ハイブリッド並列
ロードバランシング
task1
task2
task3
task4
task5
task6
task7
task8
task9
task10

図 5.4. 計算結果

第 6 章

まとめと今後の課題

6.1 まとめ

本研究は、計算化学の High Performance Computing のための新たなプラットフォームとして、C++ をホスト言語とした DSL の開発を目指した。計算化学では、配列の中の配列でそれぞれの要素数が異なるようなイレギュラーなデータ構造を用いる。その上、計算式の複雑化・組み合わせ計算の多さにより計算式に必要な変数の数が多くなるうえ、コードの最適化が難しい。そこで、開発する DSL のインターフェイスは既存研究の Halide を参考とし、計算化学への適応を図った。Halide は計算の本質を示すアルゴリズム部と計算機上の最適化命令であるスケジューリング部を分割したインターフェイスを用いている。しかし、Halide の既存の型では、計算化学の表現は困難である。何故ならば、そのイレギュラータイプのデータ構造を表現・内部的な最適化ができないためである。また、計算式定義時の変数の数が多くなり、コードの可読性を下げてしまう。そこで本研究では、Halide 形式で計算化学に特化した DSL のための Rank 型という新たな概念を提案した。Rank 型は汎用言語でのポインタやクラスのような役割を持つ。Halide の形式では、JIT コンパイルを行うため、データ構造や計算式は既知として扱うことができる。そこで、変数宣言時に Rank 型にデータ構造を記憶させることで、Rank 型は上位の階層構造も取り込んだ記述ができる。Rank 型を用いることで、既存研究では記述が困難であった計算化学の計算式を容易に記述することができる。Rank 型は、user adjustable な Halide の形式に、新たな言語機能を追加するものであり、ユーザビリティの向上に繋がっている。また、Reduction 計算の明示化も行なった。これにより、Halide の形式では不明瞭であった Reduction をユーザが理解しやすい形で明文化され、数式通りの記述も果たしている。

提案した DSL のデザインを実際に用いるために、内部 DSL をフルスクラッチした。DSL は計算化学でも頻繁に用いられる C++ をホスト言語および出力言語としている。システムとしては、3 段階のコンパイルを用いるものであり、途中に独自実装したコンパイラで AST からコード生成を行う。DSL は Deep Embedding と呼ばれる手法を用いて実装した。Deep Embedding では、アルゴリズム部の記述で計算式を表す型に AST を代入する。スケジューリング部の記述は AST に対する最適化命令であり、最後にコンパイル用関数を呼ぶことで最

適化された C++ コードが生成される。単調なコード生成以外の機能として並列化コード用関数の実装を行なった。mp_parallel および mpi_parallel 関数を用いることで頻繁に用いられるハイブリッド並列化を行うことができる。本 DSL が AST 生成後にコンパイルする（JIT コンパイル）することを生かして内部要素を考慮したロードバランシングが可能となっている。これにより、FMO 計算などの計算化学で問題点となる、配列毎の計算量差を考慮した最適化が可能となる。

実験として、簡易コードに対してのコンパイル時間検証を行なった。内部実装が異なるので Halide との比較は本来できないが、参考のために Halide も簡単なコードに対してコンパイル検証を行なった。開発した DSL の方が遅い結果となった。単純比較はできないが、対象ドメインが異なることによる結果だと考えている。並列化コードの速度検証は、計算機システムと計算対象データを変えて複数行なった。本 DSL を用いることにより、2,3 行の変更のみで様々な並列化コードの検証を行うことが可能となった。

6.2　今後の課題

ここで、今後の研究方針について何点か述べる。

本研究では、計算化学に向けた DSL のための言語デザインの創出を行った。追加の機能として、並列化の静的なチューニングを可能とした。しかし、計算化学への適応を考えるとより多くの拡張機能が必要となる。例えば、超大規模系の計算においては、データ全てがメモリにのるわけではないため、アウトコアアルゴリズムなどの考慮すべき事項が多い。これらを DSL の最適化命令で組み込む必要がある。また、マルチアーキテクチャの対応も行うべきである。現在は GPU なども計算分野に使われているので、サポートする必要がある。

また、本 DSL に対する定量的評価軸も考慮しなければならない。一般に言語デザインの評価は難しく、定量的な評価法はあまり知られていない。したがって、本研究の評価は困難であるが、本研究はこれからの研究の土台となると考えている。Halide などの既存 DSL では適応できないドメインのための土壌を開発した意義は大きく、独自の最適化手法の開発する道があると想定される。

例えば、DSL を実践的に用いるためにはオートチューニングの機能が必要だと考えている。本研究が対象としているようなイレギュラーなデータ構造を用いる計算の最適化はあまり知られていない。既存研究では Halide のオートチューニングにも触れられており、動的計画法によるチューニングが紹介されている。しかし、矩形構造が対象となるため、本ドメインには適応できない。イレギュラーなデータ構造用のアルゴリズムが必要となる。加えて、キャッシュヒットをあげる試みも考えている。画像処理ではタイリングに代表されるキャッシュヒット向上手段が存在するが、計算化学のようにクラスを用いる場合のキャッシュヒット向上は難しい。そこで、子の同じ要素を親に持たせるような機能の追加を考えている。例えば、Atom クラスが xyz 座標を持つとき、x を Atom のクラス変数として扱うのではなく、親クラス（例えば Molecule）のクラス変数の配列としてまとめるのである。この機能は実際の計算において一つ一つの計算式を見ると、子のクラスの同じ要素にアクセスする場合が多いことに基づいて

いる。これらの最適化は高速化のためのものであり、高速化手法であれば定量的な評価が可能である。

また、インターフェイスの拡張として、より計算化学に特化したものも考案する余地があると考えている。例えば、量子化学計算で使われるブラケット記法は C++ のオーバーロードを用いても困難である。よって、内部 DSL でありながら、ホスト言語の構文規約を超えたインターフェイスを作れる手法の開発が必要となる。

発表文献と研究活動

(1) 西田 秀之, 千葉 滋. ジャグ配列を用いる FMO 計算の効率的並列化に向けた DSL の設計, 第 21 回プログラミングおよびプログラミング言語ワークショップ, 岩手, 2019. (ポスター発表)

(2) 西田 秀之, 科学技術計算の効率的超並列化に向けた静的負荷分散を行う DSL の開発, 2019 年度　東京大学情報基盤センタースーパーコンピューティング部門若手・女性研究者研究提案課題採択

参考文献

[1] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines *PLDI*, 2013.

[2] Ward, Martin. Language Oriented Programming. Software-Concepts&Tools, 15. 147-161. 1994.

[3] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. ACM Comput. Surv., 37(4):316-344, 2005.

[4] Visser, Eelco. WebDSL: A Case Study in Domain-Specific Language Engineering. Lecture Notes in Computer Science, 5235/2008. 291-373. 10.1007/978-3-540-88643-3_7. 2007.

[5] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. In *Sigplan Notices*, Vol. 35, No. 6, pp. 26–36, 2000.

[6] R. M. Herndon, Jr. and V. A. Berzins. The realizable benefits of a language prototyping language. In *IEEE Trans. Softw. Eng.*, Vol. 14, No. 6, pp. 803–809, June 1988.

[7] Chris J Date and Hugh Darwen. *A Guide To Sql Standard*, Vol. 3. Addison-Wesley Reading, 1997.

[8] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems *CoRR* abs/1603.04467 , 2016.

[9] K. Kitaura, E. Ikeo, T. Asada, T. Nakano, M. Uebayasi. Fragment molecular orbital method: an approximate computational method for large molecules *Chem. Phys.*

Lett. 313 (3–4): 701–706. 1999.

[10] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines In *ASPLOS*, Mar 2015.

[11] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation In *International Symposium on Code Generation and Optimization, CGO*, 2017.

[12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, Frédo Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines In *SIGGRAPH*, 2012

[13] Jonathan Ragan-Kelley. Decoupling Algorithms from the Organization of Computation for High Performance Image Processing: The design and implementation of the Halide language and compiler Ph.D. dissertation, MIT, May 2014.

[14] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically Scheduling Halide Image Processing Pipelines In *SIGGRAPH*, 2016

[15] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, Jonathan Ragan-Kelley. Differentiable Programming for Image Processing and Deep Learning in Halide In *SIGGRAPH*, 2018

[16] Abhinav Jangda and Uday Bondhugula. An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines. In *Proceedings of ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, Vo. sendorf, Austria, Feb 2018 (PPoPP' 18), 15 pages.

謝辞

本論文は筆者が東京大学大学院情報理工学系研究科創造情報学専攻修士課程在籍中の研究成果をまとめたものである。本研究を進めるにあたり、支えてくださった皆様に感謝いたします。特に、指導教員である同専攻教授千葉先生には重要なご指導、ご助言を戴きました。大変感謝しております。筆者の所属する千葉研究室の同専攻助教穐山先生には様々な疑問に答えて戴き、丁寧にご指導戴きました。大変感謝しております。専攻における発表の場などでは、副査の同専攻准教授塩谷先生、同専攻准教授塚田先生、並びに同専攻教員の方々に有益なご討論ご助言を戴きました。大変感謝しております。また、同研究室で共に研究していた学生の方々との会話で研究のヒントを得る事もでき、皆さんの研究進捗は私の研究に対する励みとなりました。大変感謝しております。本研究の一部は東京大学情報基盤センター若手・女性利用者推薦によりスーパーコンピュータシステムの利用をさせて戴きました。大変感謝しております。多くの人に支えられて研究が進めることができ、人生の大きな糧となりました。これからの人生でもこの経験を活かして生きていく所存です。ここで再び深く感謝申し上げます。