Vol.2022-HPC-185 No.29
2022/7/29

アスペクト指向言語を用いたHPC向けDSL作成プラットフォームへのJust-In-Timeコンパイラシステムの導入による高速化の提案と評価

石村 脩[†1,a)]　吉本 芳英[†1,b)]

概要：ドメイン特化言語 (DSL) による高速化のアプローチの問題の一つとして、DSL プラットフォーム自身の移植性の低さがあげられる。この問題を解決するため、我々はアスペクト指向プログラミング (AOP) を用いた DSL 作成プラットフォームを提案している。当プラットフォームでは、AOP を用いることで、HPC システムを利用するためのランタイムコードや最適化機構のコードをモジュール化することを可能としている。しかし、DSL で書かれたカーネルコード自体の変更を行わないため、SIMD や GPU オフローディングができない問題点が存在した。本研究では、当問題を解決するため、プラットフォーム及び AOP のアスペクトに JIT コンパイラを導入し、動的にカーネルコードを生成・実行する手法を考案し、評価を行った。

キーワード：ドメイン特化言語, アスペクト指向プログラミング, ハイパフォーマンスコンピューティング, Just-In-Time Compiler

Osamu Ishimura[†1,a)]　Yoshihide Yoshimoto[†1,b)]

1. はじめに

HPC システムが複雑化する中で、HPC アプリケーションのコードもまた複雑化している。この問題の解決のためのアプローチの一つがドメイン特化言語 DSL である。しかし、DSL を用いたアプローチでは、プログラムのもつ複雑性を DSL 処理系が担っているに過ぎず、DSL 処理系それ自体のコードは複雑なものとなる。これは DSL 処理系の保守性や移植性の低下を招く。

この問題を解決するため、我々はアスペクト指向言語を導入した、DSL 処理系構築プラットフォームを提案している [1]。当プラットフォームでは、DSL システムの階層構造に着目し、各階層向けのシステムコードをアスペクトの再利用可能なモジュールとして分離し、これらを組み合わせて DSL 処理系を作成することができる。

プラットフォームの動作モデルは、タスク並列モデルで、タスクの制御およびタスク間の通信により並列化を行っている。しかし、各タスクの実行するカーネルはエンドユーザーが DSL 処理系の提供するライブラリを利用して記述したコードがそのまま実行されるため SIMD 化対応や GPGPU 対応ができないという問題が存在した。この問題を解決するため、JIT コンパイラ及び、JIT コンパイラに対するアスペクトライブラリを導入し、SIMD 化、GPGPU 対応を行う手法を考察し、評価を行った。

本稿の構成は下記のとおりである。まず、2 章でアスペクト指向プログラミング及び本プラットフォームの概要について説明する。次に、3 章で JIT コンパイラ処理系の動作モデル、4 章で評価について述べる。そして、5 で関連研究の紹介を行い、最後に 6 章で本稿のまとめと今後の展望について述べる。

2. 背景

2.1 Aspect Oriented Programming

アスペクト指向プログラミング (AOP) は、G. Kiczales らによって提案された、オブジェクト指向プログラミング (OOP) を拡張し、OOP では依存関係の形式によって取り除くことのできない横断的関心事を分離するプログラミ

†1 現在，東京大学
Presently with The University of Tokyo
a) oishimura@is.s.u-tokyo.ac.jp
b) yosimoto@is.s.u-tokyo.ac.jp

ⓒ 2022 Information Processing Society of Japan

ングパラダイムである [2, 3]。横断的関心事の代表的な例としては Log 関数などが挙げられる。Log 関数を Log クラスとして分離したとしても、Log 関数を実行する位置の変更は、Log クラスの変更で行う事ができない。

OOP では、データと処理をオブジェクト単位にクラスとしてモジュール化する。AOP では、データと挿入される処理 (Advice) に加え、処理の挿入箇所を含めて "Aspect" と呼ぶモジュールにする。

AOP の一般的な動作モデルのひとつは AspectJ [4] や AspectC++ [5, 6] で用いられる Join Point Model (JPM) である。JPM では挿入箇所の指定に "Pointcut" と呼ばれるパターンマッチを用いる。

2.2 プラットフォームの概要

本プラットフォームは多段のタスク並列モデルをとる。一般的なタスク並列モデルによる並列化では、元のプログラムをタスクに分割することで並列化が行われる。一方、本プラットフォームでは、プログラム作成時点で計算対象領域を "Block" と呼ぶ領域に分割しており、エンドユーザーは "Block" を更新する "Subkernel" を作成する。実行時には、図 1 の様に、各タスクは、"Subkernel" と複数の "Block" を受け取り計算を行う。

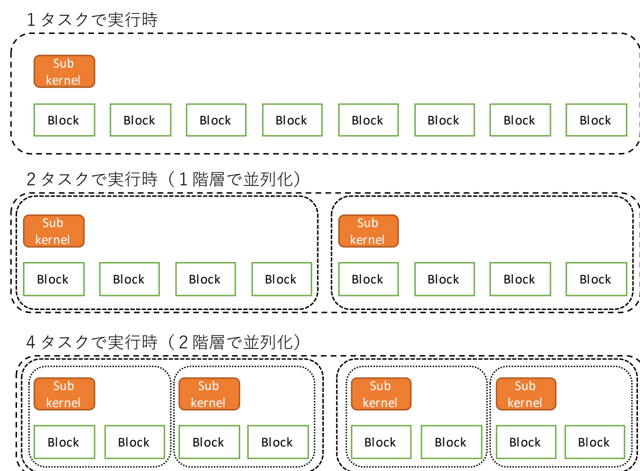

図 1: プラットフォーム動作モデル

HPC システムの階層を構成するコンポーネント毎に用意されたアスペクトモジュールは、階層の初期化・終了や、"Block" の割り付け、タスク間のデータ移動を行う。

当プラットフォームは C++及び AspectC++で作成されている。そのため、プラットフォームの開発者と DSL の開発者はこれら 2 つの言語を用いて開発を行う。エンドユーザーはプラットフォームの提供する対象とするアプリケーション向けの C++のアノテーションクラス (仮想クラス) を継承して C++でアプリケーションを作成する。

2 はプラットフォーム構成図である。

プラットフォームに関連するコードは大きく分けて 3 つ

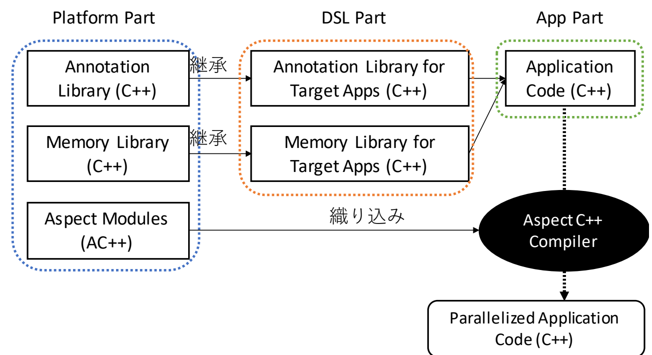

図 2: プラットフォーム構成概要

のパートに分けられる。

- Platform Part: プラットフォームの提供するライブラリ
  - Annotation Library (C++)
  - Memory Library (C++)
  - Aspect Module Library (AC++)
- DSL Part: DSL 開発者が作成する、対象アプリケーション向けのライブラリ
  - Annotation Library for Target Apps (C++)
  - Memory Library for Target Apps (C++)
- App Part: エンドユーザーが作成するアプリケーションコード
  - Application Code (C++)

Annotation Library 及び Memory Library は、Aspect Module Library 内のアスペクトが、Advice を挿入する際のアノテーション機能を提供する。これにより、DSL Part に依存せず、制御ロジックの挿入が行われる。また、"Pointcut" によるパターンマッチが想定外に行われる事を防ぐ効果もある。

Aspect Model Library は、HPC システムの各階層に対応するモジュールであり、対応するレイヤーのランタイムを管理するアスペクトを提供する。

アスペクトは、その階層初期化終了、タスクの管理、ブロックの割り付け、タスク間のデータの転送、上位レイヤーからのデータアクセス要求の処理などを行う。

エンドユーザーは作成した C++のプログラムコードを各種ライブラリとともに AspectC++コンパイラを用いて C++のコードへトランスコンパイルを行う。そして、出力されたプログラムコードを対象の計算機向けのコンパイラでコンパイルする。

2.2.1 対象アプリケーションと制約

当プラットフォームは、下記の 3 つを前提を満たすアプリケーションを対象として開発する。

前提 1: (必須) 対象アプリケーションのロジックが反復的で、各ステップの結果が前のステップの結果に何らかの算術演算（カーネルと呼ぶ）を行うことで得られる

ⓒ 2022 Information Processing Society of Japan

前提 2: 各ステップのカーネルのメモリアクセスパターンが各反復で変わらない

前提 3: メモリアクセスが空間局所性を持つ

なお、一般的な HPC アプリケーションは上記の特性を持つ。

2.2.2 データ構造

対象のアプリケーションのグローバルなデータは “Block” の木構造 (“Env”) で表現される。“Block” は Subkernel が計算するデータの単位で、計算対象のアプリケーションに応じて定義される次元を持つ固定サイズのデータ構造で、空間における配置情報をもつ。

“Block” の種類は、実際にデータを持つ実体 Block や、境界条件や通信用バッファ・静的データの保管に使われる仮想 Block が存在する。

2.2.3 メモリ管理

プラットフォームは効率化のため、複数階層でメモリ管理を行う [1]。

図 3 にプラットフォーム上のメモリの名称と構造を示す。

まず、与えられた連続的なメモリ領域 (Memory Pool (MemPool) と呼ぶ) からメモリ領域 (Virtual Memory Pool (vMemPool) と呼ぶ) を切り出す。等サイズの vMemPool を束ねたものを Virtual Memory Pools (vMemPools) と呼ぶ。vMemPools の 0 番目の vMemPool を書き込みバッファ、1 番目を vMemPool を標準の読み込みバッファとし、vMemPools 内の vMemPool の順番を入れ替えることによりマルチバッファを実現している。

そして、データはすべて vMemPools の先頭からの相対アドレスで管理している。

各ブロックは vMemPools から切り出したメモリ領域 (BufferSet と呼ぶ) をもち、データは BufferSet に保存される。

BufferSet の内部は Padding サイズ分間隔を空けて配置される固定サイズの Page の単位でアクセスの有無が管理される。そのため、分散メモリ環境等におけるデータ転送の最小の単位は Page となる。

各ページは複数のデータの配列である。Padding はキャッシュ効率の改善のために用いられる。

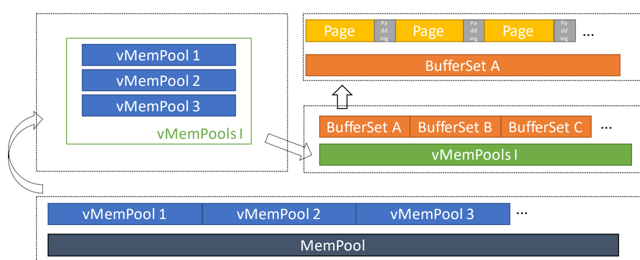

図 3: プラットフォーム上のメモリの名称と構造

2.2.4 プラットフォームの動作モデル

前提 2、前提 3 に基づいた最適化として、プラットフォームは図 4 のように動作する。まず、各 Block に対して Subkernel を実行する (DryRun)。DryRun の際のデータの書き込みはすべて破棄する。DryRun を行った際、各タスクからアクセスできないアドレスへのアクセスがあった場合、そのアドレスのデータを取得・生成し、DryRun を再実行する。アクセスできないアドレスへのアクセスがなかった場合は動作が Run に切り替わり、データの書き込みが行われる。Run の実行中は、DryRun 実行時のアクセスの記録のキャッシュの利用や、タスク間のデータのプリフェッチを行う。Run の実行中、アクセスできないアドレスへのアクセスがあった場合は、そのステップの書き込みおよびアクセスキャッシュやプリフェッチ用データを破棄し、再度 DryRun を実行する。

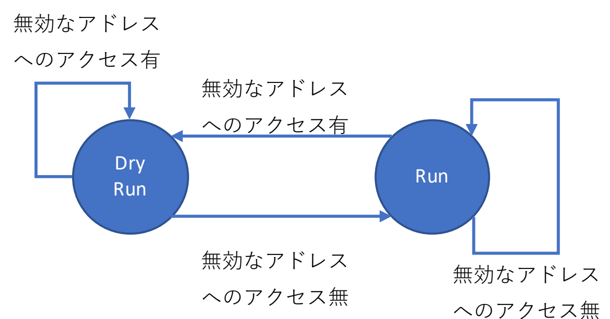

図 4: プラットフォームの各ステップの動作モデル

3. JIT コンパイラ

従来のプラットフォームでは、データ要素にアクセスする度にそのアドレスを解決する必要があり、キャッシュを用いたとしても大きな実行時のオーバーヘッドになっていた。また、コード上このアドレス解決ロジックはデータアクセスと絡みついてしまうため、SIMD 化や GPU 対応に支障がでていた。しかしプラットフォームの概要から理解できるように、環境に対して DryRun を行えば、このメモリアクセスは確定する。つまり DryRun が終了した時点で、各 Block に対して Subkernel を実行する際に行う、演算及びアクセスする BufferSet とその相対アドレスは確定し、問題のアドレス解決ロジックは不要になる。そして、このアドレス解決済みの Subkernel 処理を JIT コンパイルして以後の本番計算で用いることにすれば、アドレス解決のオーバーヘッドは取り除かれ、また Subkernel の JIT コンパイルにおいて、SIMD 化や GPU 対応を行うことができるようになる。

これを実現するため、JIT コンパイラは DryRun 時のアドレス解決情報に基づき、引数として BufferSet の先頭アドレス群を受け取り、Subkernel と等価な処理を行う関数

を作成、共有ライブラリとしてコンパイルする。

従来のプラットフォームが記録する”アクセスする BufferSet とその相対アドレス”に加え、演算をプラットフォームが取得するため、各メモリアクセス関数は直接値を返すのではなく、メモリアクセスの内容を記載したオブジェクト (メモリアクセスオブジェクト) を返す。オブジェクト間の演算は、演算子オーバーロードのシンタックスシュガーによって表現する。メモリアクセスオブジェクトへの代入毎に “Block” の持つ中間コードのリスト (Flow オブジェクト) に式を追加する。

なお JIT コンパイル自体のオーバーヘッドにより、HPC アプリ自体が小さな演算しか行わない場合は JIT コンパイルを行わない方がよいケースが存在しうる。その対応として、中間コードを JIT コンパイルせずそのまま実行する事も可能ではあるが、直接の演算と比較し大きなオーバーヘッドが生じることは避けられない。そこで、JIT を利用する場合（JIT コンパイル環境）と利用しない場合（通常環境）で Subkernel の記述方法を区別し、DSL Part の設定でどちらを利用するか切り替え可能とした。この欠点として、カーネルコードの記述が素の C++の記述とは異なってしまう。しかし、既存の埋め込み型の DSL であり、幅広く利用されている Halide [7] でも同様の手法が取られているため、この欠点は受容することとした。

JIT 用のオブジェクトとして下記を用意する。

- Constant: 定数を表現するオブジェクト
- MemoryAccess: 読み込みを表現するオブジェクト
- Assign: 書き込みを表現するオブジェクト
- Expr1: 一項演算を表現するオブジェクト
- Expr2: 二項演算を表現するオブジェクト
- Loop: ループを表現するオブジェクト
- Block: 変数のエイリアスやメモリアクセスのアイリアスを作成するオブジェクト
- Func: 関数を表現するオブジェクト
- FuncCall: 関数呼び出しを表現するオブジェクト
- Flow: オブジェクトの集合を表すオブジェクト

上記のオブジェクトの木構造で表されるロジックを中間コードと呼ぶ。また、MemoryAccess オブジェクトおよび Assign オブジェクトをまとめ、データアクセスと呼ぶ。

```
1  template <bool isDryRun>
2  void Subkernel(BLK &blk) {
3   auto c_a = C(0.8f);
4   auto c_b = C(0.1f);
5   for (AddrE i = 0; i < BLK::SIZE_X; i++) {
6    auto bl0 = GetE(LA_t{i - 1});
7    auto bl1 = GetE(LA_t{i});
8    auto bl2 = GetE(LA_t{i + 1});
9    bl1 = c_a * bl1 + c_b * (bl0 + bl2);
10  }
11 }
```

ソースコード 1: JIT 版 Subkernel コード例

3.1 境界条件の更新

境界条件のデータがステップ毎に変更される場合、JIT を利用しないケースでは境界条件を示すブロックが呼ばれるたびにデータが生成されていた。該当の機能を JIT コンパイル環境を利用して実現する手法としては、境界条件でのデータ生成を行う関数自体を共有ライブラリに組み込む方法と、事前にデータをホスト側で用意し、共有ライブラリ側では該当データへのメモリアクセスに変更する方法がある。

後者は JIT コンパイラの構造が簡潔になる一方、２点のデメリットがある。１点目は、カーネルの実行タイミングで生成すべきデータが変更されるケースに対応できない事である。しかし、プラットフォームの動作モデルによりこちらは考慮する必要がない。２点目は、事前のデータ生成処理がホストでシリアルに実行され、プログラムの実行速度の低下を招く可能性がある点である。しかし、前提２よりプログラムは空間局所性をもち、Block 内部の演算と比較し、境界条件へのアクセスは十分に小さいことが予測される。

よって当プラットフォームでは、後者の手法を取る。

3.2 中間コードの融合

上記により、Block 毎の Subkernel 相当のプログラムを作成できるが、下記の 2 点の問題がある。

- Subkernel のサイズ自体がそもそも小さく、SIMD 化を行うのに十分な並列性が取り出せるとは限らない
- 環境によっては、コンパイラの呼び出し処理が重く、複数回コンパイルすることは大きなオーバーヘッドを引き起こす

この問題を解決するため、タスクごとに、そのタスクが対象とするブロック向けの中間コードの融合を行う。

各中間コードの読み込み・書き込みオブジェクトは内部に BufferSet の ID と相対アドレスを持つ。そして、単一タスク内の中間コードに含まれるデータアクセスはタスクがアクセス可能ないずれかの vMemPools から確保された BufferSet へのアクセスである。よって、これらのオブジェクトを vMemPools の ID と相対アドレスに置き換え、Subkernel を融合することが可能である。

3.3 中間コードの最適化

中間コードが構成された時点で、Flow オブジェクトは代入回数分の Assign オブジェクトをリストとして持つ。

しかし、この中間コードからそのまま共有ライブラリのコードを作成した場合、元の Subkernel のコードに含まれ

ていた反復処理による周期性が失われたままとなり、出力されるマシンコードも複雑なものとなる。この問題を解決するため、中間コードから周期性を取り出し、再度 Loop 化する。

まず、中間コード圧縮、コンパイル時間の短縮およびコンパイラへの SIMD 化へのヒントとして Loop オブジェクトへの変換を行う。Loop 化することによる副次的な効果として、命令列が圧縮されることにより、命令キャッシュ及び命令ストリームが効率化される。Assign オブジェクト以下の木構造の変化を抽出し、木構造が同一かつ、データアクセスの差分が一定間隔であるものが一定回数以上続いたとき、それらを Loop オブジェクトに変換する処理を行う。

次に、必要に応じて、データアクセス内の保持するデータへのエイリアスの作成、ループの試行回数の変数化を Block オブジェクトの挿入によって行う。そして、作成された Block オブジェクトを、Func オブジェクト及び FuncCall オブジェクトに置き換える。

なお、上記の処理を再帰的に実行し、多重ループを構築することも可能。

3.4 データアクセス形式

中間コードの出力の際、データアクセスを C++のコードに変換するにあたって、下記の二通りのアクセス方法が考えられる。

- ポインタ方式: 対象のデータが格納されたアドレスを計算し、そのアドレスにアクセスを行う。(ソースコード 2)
- 配列方式: データの先頭のアドレスから、配列形式でアクセスを行う。(ソースコード 3)

```
uintptr_t addr_head;
uintprt_t addr;
/* some calculation */
auto v = *(float*)(void*)(addr_head+addr);
```

ソースコード 2: ポインタ方式例

```
uintptr_t addr_head;
size_t index;
auto SU__restrict * ary = *(float*)(void*)(addr_head);
/* some calculation */
auto v = ary[index];
```

ソースコード 3: 配列方式例

ソースコード中の “SU__restrict” は環境に合わせて “__restrict” もしくは “__restrict__” が定義されたマクロである。

ポインタ方式ではアプリケーションの扱うデータのデータ構造を考慮せずにアクセスする事が可能であるが、配列方式ではできないという問題がある。

利用するコンパイラによって、上記の 2 つの表記によって SIMD 化に差異が出たため、双方の JIT 用オブジェクトを実装し、切り替え可能とした。

3.5 出力コード例

ソースコード 4 に、32 点の 1 次元構造格子向けの Subkernel を、ポインタ方式で記述した場合の Windows 環境向け出力例を示す。

```
extern "C" __declspec(dllexport)
void Subkernel(void **vmps) {
 auto * SU__restrict ary0f_0=(float*)vmps[0];
 auto * SU__restrict ary0f_1=(float*)vmps[1];
 auto * SU__restrict ary1f_0=(float*)vmps[2];
 ary0f_0[0]=((0.800000f)*(ary0f_1[0]))+((0.100000f)
           *((1.000000f)+(ary0f_1[1])));
 for(int i0=0; i0<30; ++i0){
  ary0f_0[1+i0]=((0.800000f)*(ary0f_1[1+i0]))
      +((0.100000f)*((ary0f_1[i0])+(ary0f_1[2+i0])));
};
ary0f_0[31]=((0.800000f)*(ary0f_1[31]))+((0.100000f)
           *((ary0f_1[30])+(1.000000f)));
}
```

ソースコード 4: 1 次元構造格子向けの Subkernel 出力例

3.6 CUDA 対応

JIT コンパイラを前提とした、CUDA 対応のアスペクトモジュールを作成した。

JoinPoint は下記の 2 つを用意し、それぞれを上書きする Advice を用意する。

- 中間コードを受け取り、ソースコードを出力する関数
- コンパイラのパスやオプションを返す関数

中間コードからソースコードへの変換では、各 Flow を threadId が処理する形とした。条件分岐による性能劣化を抑えるため、中間コードに対して Loop 化の最適化を行い、threadId を Loop の index と見立ててコードを生成する。

ソースコード 5 に、32 点の 1 次元構造格子向けの Subkernel を、CUDA 向けに配列方式で記述した場合の出力例を示す。

```
__global__ void cudaKernel(void* dev_0
                , void* dev_1, void* dev_2){
 auto * SU__restrict ary0f_0=(float*)dev_0;
 auto * SU__restrict ary0f_1=(float*)dev_1;
 auto * SU__restrict ary1f_0=(float*)dev_2;
 int tix = threadIdx.x;
 if(tix < 1){
  int i0 = tix-0;
  ary0f_0[i0]=((0.800000f)*(ary0f_1[i0]))
   +((0.100000f)*((1.000000f)+(ary0f_1[1+i0])));
 }else  if(tix < 31){
  int i0 = tix-1;
```

```
13     aryOf_0[1+i0]=((0.800000f)*(aryOf_1[1+i0]))
14     +((0.100000f)*((aryOf_1[i0])+(aryOf_1[2+i0])));
15   }else  if(tix < 32){
16     int i0 = tix-31;
17     aryOf_0[31+i0]=((0.800000f)*(aryOf_1[31+i0]))
18      +((0.100000f)*((aryOf_1[30+i0])+(1.000000f)));
19   }
20 }
21 extern "C" void Subkernel(void **vmps) {
22   /* omited */
23   void* dev_0;
24   cudaStatus = cudaMalloc(&dev_0,100000000);
25   cudaStatus = cudaMemcpy(dev_0,vmps[0],100000000
26               ,cudaMemcpyHostToDevice);
27   /* omited */
28   cudaKernel << <1,32 >> > (dev_0,dev_1,dev_2);
29   cudaStatus = cudaGetLastError();
30   /* omited */
31   }
```

ソースコード 5: 1 次元構造格子向けの Subkernel の CUDA 向け出力例

4. 評価

4.1 ベンチマークプログラム

プラットフォーム上で 1 次元空間および 2 次元空間の構造格子向けの DSL を作成し、同 DSL を用いて作成したベンチマークの評価を行った。

ベンチマークの共通のパラメータは次の通り。

- DataType: float
- Padding Size: 0kb

現時点では、プラットフォームはメモリレイアウトを最適化する機能を持たないため、Padding が SIMD 化を行った際に性能劣化を招く。そのため、今回の性能評価では Padding サイズを 0 に固定している。

4.2 SIMD コード生成

1 次元空間構造格子向けベンチマークを実行し、生成された Subkernel のコードを、各コンパイラでコンパイルして、SIMD 命令が出力されていることを確認する。

ベンチマークのパラメータは次の通り。

- Block Size(1 次元): 16
- RegionSize(1 次元): 64
- Page Size: $sizeof(float) * 2^4$

検証に利用したコンパイラと最適化オプションは次の通り。

- 表示名: msvc
  - コンパイラ: Microsoft x86 msvc v19.29
  - 最適化オプション: /O2 /Ot /Ob1 /fp:fast
  - SIMD オプション: AVX2, AVX512
  - ターゲットアーキテクチャ: x86-64
- 表示名: gcc
  - コンパイラ: x86-64 GCC 12.1
  - 最適化オプション: -Ofast -shared -fPIC -ffast-math
  - SIMD オプション: avx2, avx512f
  - ターゲットアーキテクチャ: x86-64
- 表示名: clang
  - コンパイラ: x86-64 Clang 14
  - 最適化オプション: -Ofast -shared -fPIC -ffast-math
  - SIMD オプション: avx2, avx512f
  - ターゲットアーキテクチャ: x86-64
- 表示名: icc
  - コンパイラ: Intel x86-64 ICC 2021.5.0
  - 最適化オプション: -O3 -restrict -fPIC
  - SIMD オプション: CORE-AVX2, COMMON-AVX512
  - ターゲットアーキテクチャ: x86-64
- 表示名: FCCpx
  - コンパイラ: Fujitsu FCCpx 4.8.0
  - 最適化オプション: -Kfast
  - ターゲットアーキテクチャ: A64fx

テーブル 1 は各コンパイラで生成したアセンブリ内の SIMD 命令数である。SIMD の行はターゲットとする SIMD を示す。AVX2 用のオプションは 2、AVX512 用のオプションは 512 と省略して記載する。SIMD 化 (P) と SIMD 化 (AI) の行はそれぞれ、データアクセスがポインタ方式の場合とデータアクセスが配列方式の場合に SIMD 化が行われたかを示す。

表 1: コード生成結果

コンパイラ	msvc		gcc		clang		icc		FCCpx
SIMD	2	512	2	512	2	512	2	512	SVE
SIMD 化 (P)	×	×	○	○	×	×	○	○	×
SIMD 化 (AI)	○	○	○	○	○	○	○	○	○

icc を利用して avx512 命令を発行した場合は avx2 命令は発行されなかった。他の avx512 向けオプションを利用した場合は avx512 命令と avx2 命令が混合して発行された。

4.3 SIMD パフォーマンス評価

2 次元空間構造格子向けベンチマークを実行し、JIT 版と非 JIT 版のパフォーマンス比較を行う。

ベンチマークのパラメータは次の通り。

- Block Size(1 次元): 256
- RegionSize(1 次元): 2048
- Page Size: $sizeof(float) * 2^{11}$
- MPI Process: 4 プロセス
- ステップ数: 1000

コンパイラは icc を用いた。

- コンパイラ: Intel x86-64 ICC 19.1.3.304
- 最適化オプション: -O3 -restrict -fPIC
- SIMD オプション: COMMON-AVX512

JIT コンパイラのパラメータは次の通り。

- データアクセス: 配列方式
- Loop 化: 1 回

テスト環境は Oakbridge-CX を用いた。

4.3.1 Oakbridge-CX

- CPU: Intel Xeon Platinum 8280 × 2
- Memory (per node): 2,933 RDIMM 16 GB × 12
- Network: Intel Omni Path (12.5 GB/s)

表 2: JIT 版と非 JIT 版の実行時間比較

	非 JIT 版	JIT 版
コンパイル時間 [ms]	-	1433
実行時間 [ms]	32	12

アプリケーションの演算対象領域が小さく、且つイテレーション回数が少ないため、コンパイル時間が占める割合が非常に大きく、JIT 版のパフォーマンスが劣っている。しかし、コンパイルを除く実行時間は約 38%まで短縮している。

5. 関連研究

一般的に用いられている埋め込み型 DSL として、画像処理向けの DSL である Halide [7] が存在する。

HPC システム向けの DSL の先行研究としては Physis [8], ebb [9], Legion [10], formura [11] などが存在する。なかでも Physis は埋め込み型の DSL のモデルを採用している。

AOP を HPC アプリケーション開発に利用した先行研究は次のものが存在する。

- MPI の通信を Aspect として分離した J. Dean らの研究 [12]
- ループの並列化を Aspect を用いて行った B. Harbulot ら [13] および J. Sobral らの研究 [14]

6. 終わりに

本研究では、AOP を用いた DSL 処理系構築プラットフォームのカーネル処理に、JIT の機構の導入を行う設計を提案した。また、JIT の導入により SIMD および CUDA への対応を行った。

JIT 版を AVX512 環境で実行し、コンパイル時間を除く実行時間の高速化を確認した。

評価には記述していないが、2 次元空間構造格子向けベンチマークを実行し、JIT 版はコンパイルが現実的な時間で終了しないことも確認できた。この問題の解決のため、さらなるコードの圧縮とコンパイル時間の短縮が今後の課題となる。

他に、JIT コンパイラの拡張として下記を予定している。

- 構造体への対応
- プラットフォーム初期のメモリレイアウトをそのまま利用するのではなく、JIT コンパイラで生成された Subkernel に適したメモリレイアウトへの変換

CUDA 向け拡張では下記を予定している。

- プラットフォームへのテンポラルブロッキング対応
- メモリアクセスの検知機構の導入およびデータ転送の削減と圧縮
- GPU 間データ通信への対応

謝辞 本研究では、東京大学情報基盤センターの Oakbridge-CX 及び Wisteria/BDEC-01 を利用した。Oakbridge-CX 及び Wisteria/BDEC-01 の利用は東京大学情報理工学系研究科の計算科学アライアンスによる。

参考文献

[1] Ishimura, O. and Yoshimoto, Y.: Aspect-Oriented Programming based building block platform to construct Domain-Specific Language for HPC application (2022).

[2] Chiba, S.: アスペクト指向入門 -Java ・ オブジェクト指向から AspectJ プログラミングへ, 技術評論社 (2005).

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented programming, *The Spring Framework Reference Documentation*, Springer, Berlin, Heidelberg, pp. 220–242 (online), DOI: 10.1007/BFb0053381 (1997).

[4] : The AspectJ Project, `https://www.eclipse.org/aspectj/`.

[5] : The Home of AspectC++, `https://www.aspectc.org/`.

[6] Spinczyk, O., Gal, A. and Schröder-Preikschat, W.: AspectC++: an aspect-oriented extension to the C++ programming language, *Proceedings of the Fortieth International Conference on Tools Pacific*, Australian Computer Society, pp. 53–60.

[7] : Halide, `http://halide-lang.org/`.

[8] Naoya, M., Tatuo, N., Kento, S. and Satoshi, M.: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12 (online), DOI: 10.1145/2063384.2063398 (2011).

[9] Bernstein, G. L., Shah, C., Lemire, C., DeVito, Z., Fisher, M., Levis, P. and Hanrahan, P.: Ebb: A DSL for Physical Simulation on CPUs and GPUs, *ACM Transactions on Graphics*, Vol. 35, No. 2, pp. 1–12 (online), DOI: 10.1145/2892632 (2015).

[10] Bauer, M.: Legion: Programming Distributed Heterogeneous Architectures with Logical Regions, PhD Thesis, Stanford PhD Thesis (2014).

[11] Muranushi, T., Nishizawa, S., Tomita, H., Nitadori, K., Iwasawa, M., Maruyama, Y., Yashiro, H., Nakamura, Y., Hotta, H., Makino, J., Hosono, N. and Inoue, H.: Automatic Generation of Efficient Codes from Mathematical Descriptions of Stencil Computation, *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, Vol. 1, No. 212, pp. 17–22 (online), DOI: 10.1145/2975991.2975994 (2016).

ⓒ 2022 Information Processing Society of Japan

[12] Dean, J. S. and Mitropoulos, F. J.: An Aspect Pointcut for Parallelizable Loops, *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, New York, NY, USA, Association for Computing Machinery, p. 1619–1624 (online), DOI: 10.1145/2554850.2554917 (2014).
[13] Harbulot, B. and Gurd, J.: Separating concerns in scientific software using aspect-oriented programming, PhD Thesis, THE UNIVERSITY OF MANCHESTER (2006).
[14] Sobral, J. L., Cunha, C. A. and Monteiro, M. P.: Aspect Oriented Pluggable Support for Parallel Computing, *High Performance Computing for Computational Science - VECPAR 2006* (Daydé, M., Palma, J. M. L. M., Coutinho, Á. L. G. A., Pacitti, E. and Lopes, J. C., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 93–106 (2007).