JST CREST研究領域
「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」

TOHOKU UNIVERSITY

進化的アプローチによる超並列複合システム向け開発環境の創出

2016年12月26日 ATTA2016
東京大学 山上会館

研究代表　滝沢 寛之

東北大学大学院情報科学研究科

主たる共同研究者

須田 礼仁 (東京大学)
高橋 大介 (筑波大学)
江川 隆輔 (東北大学)

The Xevolver Project※1

※1 進化的アプローチによる超並列複合システム向け開発環境の創出

アプリ資産の次世代システムへの円滑な移行を支援

アプリが**特定のシステム構成を仮定**して開発されていることが**問題**

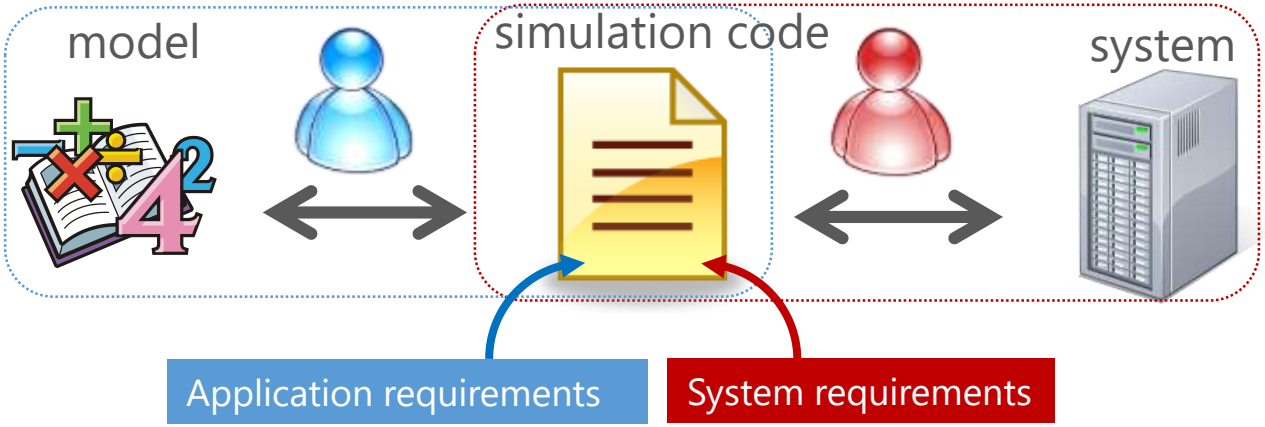
- **既存アプリで用いられているコード最適化手法調査**
  - 江川 隆輔 (東北大)
    - システム依存のコード最適化パターンを明確化
- **将来必要となるシステム依存最適化手法の考察**
  - 須田 礼仁 (東京大) and 高橋 大介 (筑波大)
    - 新世代システム向けのアルゴリズムや実装技術
- **システム依存性の統一的表現方法の確立**
  - 滝沢 寛之 (東北大) (代表)
    - アプリケーションからシステム依存性を切り離して表現

背景

- **HPCアプリケーション開発 = チームワーク**
  - **アプリ開発者 (= 計算科学)**
    - 正しいシミュレーション結果を得ることが目的

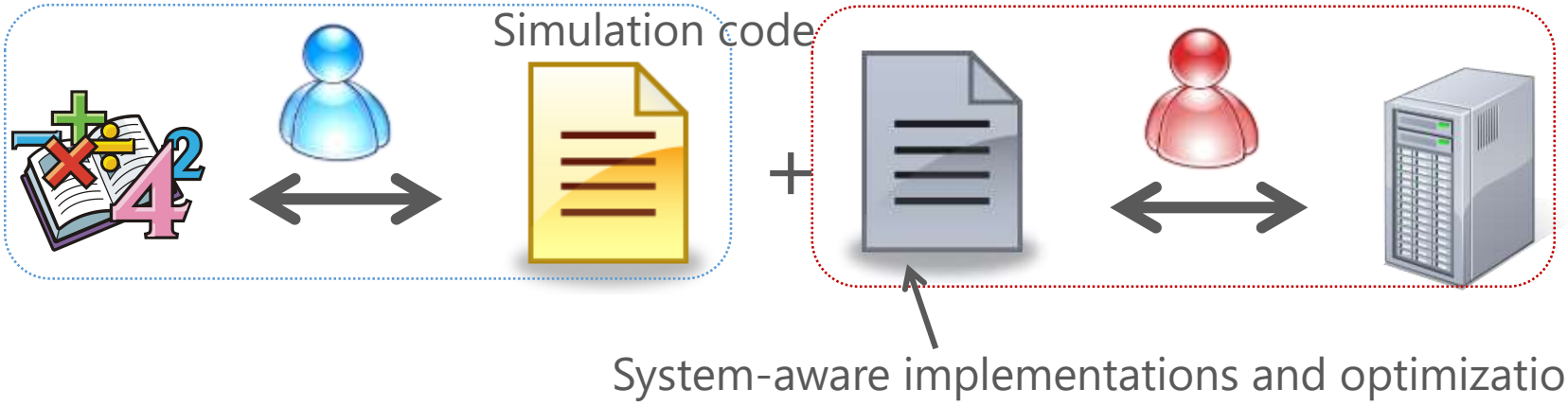
    → シミュレーションモデルとプログラムとの関係が主な関心事
  - **性能エンジニア (= 計算機科学/工学)**
    - シミュレーションを高速化することが目的

    → プログラムと計算システムとの関係が主な関心事



本研究の目的

・システム依存性をアプリコードから分離して表現・管理



システム依存性を抽象化する技術

- コンパイラ最適化
- ライブラリ＝共通インタフェースで各システムに最適化された実装を利用
- 標準化されたプログラミングモデル・言語 = MPI, OpenMP, OpenACC …

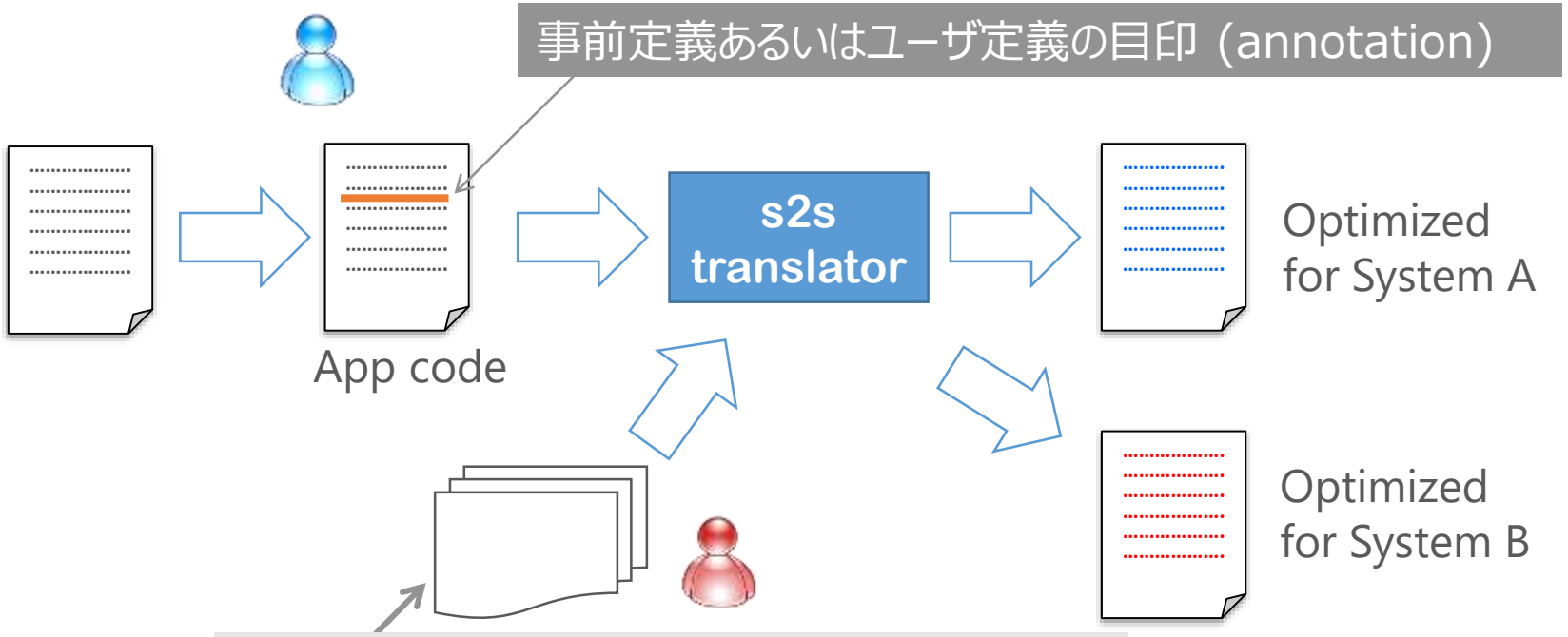
抽象化技術をすべてを適切に利用しても**アプリ特有あるいはシステム特有の理由**でアプリコードの修正が依然として必要

→ アプリコードの直接的修正を回避する方法を確立 ＝ 高い性能可搬性の実現

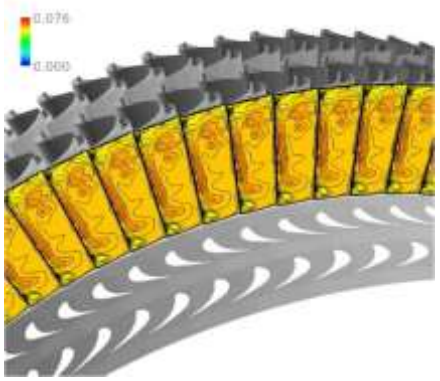
Xevolver フレームワーク

任意のコード修正を置き換えるためには**多種多様なコード変換**が必要
=いくつかの変換ルールを事前に準備して組合せるだけでは対応不可

→ Xevolver：ユーザ定義コード変換のためのフレームワーク



コード変換パターンの記述例



数値タービン (東北大 山本悟研究室)

- Fortranで記述された実用アプリ
  - 長年の開発の歴史があるコード
- NEC SX-9向けに最適化
  - 最内ループの並列性を最大限利用
- 類似の構造を持つ**44**のカーネルループ
  - GPU向けのOpenACCコンパイラには カーネルループを並列化不可

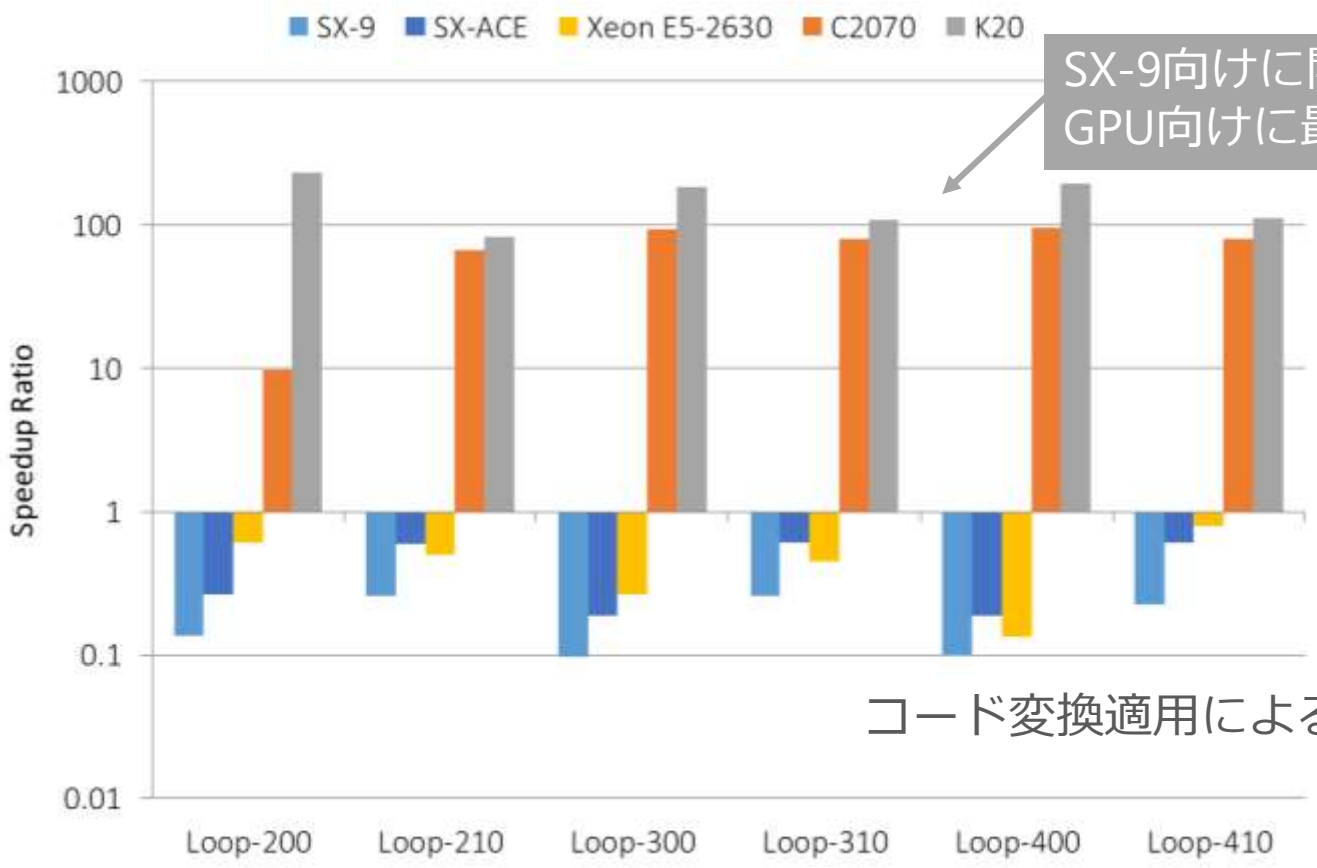
→ 44のループをすべて修正する必要性

```
program nt_opt
!$xev tgen var(i1,i2,i3,i4,i5,i6,if) stmt
!$xev tgen list(body) stmt
!$xev tgen var(lstart,lend,II2,IIF) exp
!$xev tgen condef(has_doi) contains stmt begin
  DO I=II2,IIF
!$xev tgen stmt(if)
!$xev tgen stmt(body)
  END DO
!$xev tgen end
!$xev tgen list(stmt_with_doi) stmt cond(has_doi)
!$xev tgen src begin
  DO L=lstart,lend
!$xev tgen stmt(stmt_with_doi)
  END DO
!$xev end tgen src
!$xev tgen dst begin
  DO I=1,inum
    DO L = lstart, lend
      IF (I .GE. IS(L) .AND. I .LE. IT(L)) THEN
        EXIT
      END IF
!$xev tgen stmt(if)
!$xev tgen stmt(body)
    END DO
  END DO
!$xev end tgen dst
end program nt_opt
```

変換前のコードパターン (vector-friendly)

変換後のコードパターン (GPU-friendly)

ユーザ定義変換による最適化



SX-9向けに開発されたアプリを
GPU向けに最適化する変換を表現

コード変換適用による性能変化のグラフ

手作業によるコード修正を**コード変換パターン**として表現

→ GPUで実行時には変換、 SXで実行時には無変換
＝GPUとSXの両方で高性能を達成可能

コード変換活用の事例研究

- **性能リファクタリング支援**
  - 大規模コード中のパターン検索 (Wang@IJNC)
- **性能可搬性向上**
  - OpenMP/OpenACCのカスタマイズ (Xiao@ATMG'14)
- **アプリケーション特有の最適化**
  - 数値タービン (Takizawa@SC'13 poster/HiPC'14)
  - ナノ粒子形成 (Takizawa@HiPC'14)
  - ステンシル計算用ディレクティブPACC (Kato@SC'14 Poster)
  - ステンシル計算のMPI通信隠ぺい (Hayashi@LHAM'16)
  - MSSG (Komatsu@LHAM'15)
- **アプリケーション全体に対する一括変換**
  - XevGMP (Hishinuma@CSE'16)
- **データレイアウト変換**
  - MSSGのOpenACC対応 (Takizawa@HiPC'14)
  - XSLTでAoS-to-SoA変換を実装(Yamada@LHAM'15)
  - TgenでAoS-to-SoA変換を実装(Takizawa@SSP'16)
- **自動チューニング可能化**
  - OpenTunerとの連携 (Takizawa@SC'16 poster)

既存のアプリケーション資産

- **膨大なアプリケーション資産(legacy code)**
  将来のシステムでも動くかもしれないが**高性能は期待できない**
  - 低抽象度のプログラミング言語 (e.g. C and Fortran)
    
    アプリコードの大半はすでにアプリケーション開発者によって書かれている
    
    性能エンジニアにとっては(アプリ全体を書き直さない限り)プログラミング言語を選択する余地はない
  - 長い開発の歴史
    - 数多くのプログラマは開発に参加してきた歴史
      - 全体像を把握している人は誰もいないかも・・・
      - 性能への影響が大きい部分がコード全体に散らばっている
  - アプリ資産は重要かつ高信頼かつ有用
    - だからこそ長い間維持・保守されてきた
    - 正しい計算結果が得られることが(これまでの実績から)わかっている

  →  アプリ開発者は**アプリ資産の大規模な修正を嫌う**傾向

＝ 今日の前提：なるべく元のコードをそのまま使いたい

自動チューニングの重要性

- **将来のHPCシステムのための性能最適化**
  - HPCシステムの大規模化・複雑化・多様化
    - それぞれのシステムには、異なる並列化手法、プログラミングモデル、言語などが必要となる可能性
  - HPCアプリケーションも大規模化・複雑化・多様化
    - それぞれのアプリケーションには、異なるアルゴリズム、性能最適化技法、保守方針などが必要となる可能性

さまざまな選択肢を試行錯誤で選択する必要
試行錯誤の作業を自動化したい = **自動チューニング (AT)**

自動チューニングの典型的な手順

① **対象コードに「性能のつまみ」 (パラメータ)があると仮定**
- コードの性能はパラメータ調整により変化

② **パラメータを調整する**

③ **性能を評価する**

④ **十分な性能に到達するまで②と③を繰り返す**
- 効率よく最適(準最適)な解に到達するようにパラメータ探索することがAT研究の重要課題の一つ

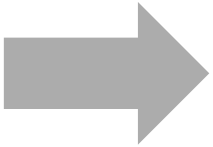
一般的なアプリに「性能のつまみ」があるか？
多くの場合はNo
→ コードにつまみをつける必要 = **AT可能化**

AT可能なコード

シンプルなオリジナルコード

```
DO i=0,n
  DO j=0,n
    sum = c(j,i)
    DO k=0,n
      sum = sum+a(k,i)*b(j,k)
    END DO
    c(j,i) = sum
  END DO
END DO
```

AT可能化！



```
DO i1=0,n,BLOCK_SIZE1
  DO j1=0,n,BLOCK_SIZE2
    DO k1=0,n,BLOCK_SIZE3
      DO i=i1,n+BLOCK_SIZE1
        DO j=j1,j1+BLOCK_SIZE2
          sum = c(j,i)
          DO k=k1,k1+BLOCK_SIZE3
            sum = sum+a(k,i)*b(j,k)
          DO END
          c(j,i) = sum
        END DO
      END DO
    END DO
  END DO
END DO
```

自動チューニングには適切な BLOCK_SIZE*の値を効率よく見つけることを期待

- ✓ アプリ開発者は複雑化したAT可能なコードを保守しなければならないのか？
- ✓ 性能エンジニアはアプリコードをどうやってAT可能化すればよいのか？
  （万能な方法はないので、個々のアプリに合わせた対応が必要）

ATとコード変換との出会い

[1] Ansel et al.@PACT2014
[2] Takizawa et al.@HiPC2014

- **OpenTuner[1] = Autotuning framework**
  - 対象コードがAT可能になっていれば、そのパラメータを効率よく見つけることができる
- **Xevolver[2] = Code transformation framework**
  - コード変換ルールをアプリから分離して記述することで、オリジナルコードを維持しつつAT可能化できる

組み合わせることによって**アプリ資産の保守性を維持**しつつ
自動チューニング技術を活用することができる

自動チューニング + コード変換

- **自動チューニング(AT)の目的**
  - 試行錯誤によるパラメータ調整を自動化
  ＝性能最適化作業の一部を自動化
- **自動チューニングの問題点**
  - ほとんどのアプリはATできるよう書かれていない
  ＝ATできるようにコードを修正する必要
    - コードの複雑化
    - コンパイル時に決めなければならないパラメータの増加
- **コード変換との連携**
  - アプリをAT可能にするための情報を分離して表現

コード変換 + 自動チューニング

- **コード変換の目的**
  - システム依存の最適化情報をアプリから分離して記述(変換レシピ)
- **コード変換の問題点**
  - システムの数だけ変換レシピが必要
  - 変換レシピを定義するための労力
- **自動チューニングとの連携**
  - 似たようなシステムのレシピを共通化
  - 適切な変換の探索を自動化

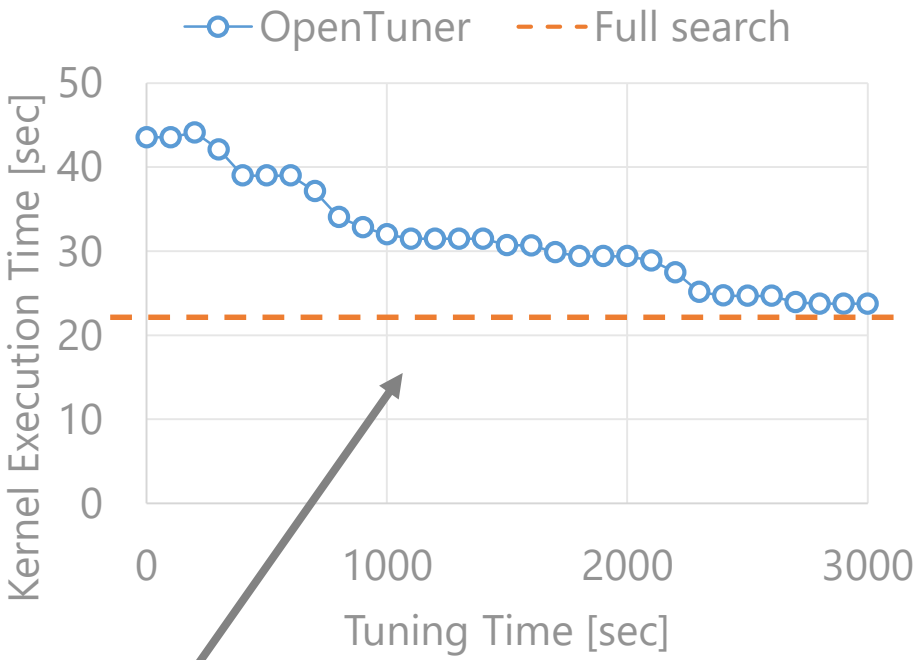
チューニング時間削減効果

- **自動チューニングの恩恵**
  - 姫野ベンチマークの自動チューニングの例

Experimental Setup

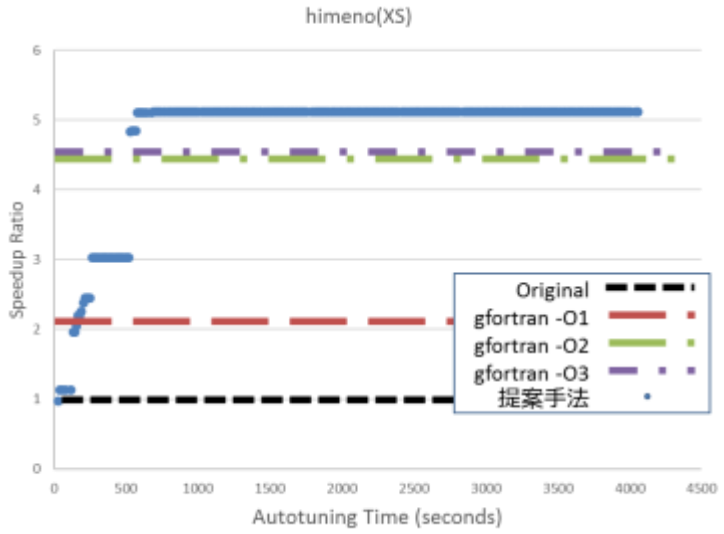
System
- CPU : Intel(R) Xeon(R) CPU E5-2630@2.30GHz
- Mem : 8 Gbytes
- OS : CentOS 6.4
- Compiler: GNU Fortran 4.7

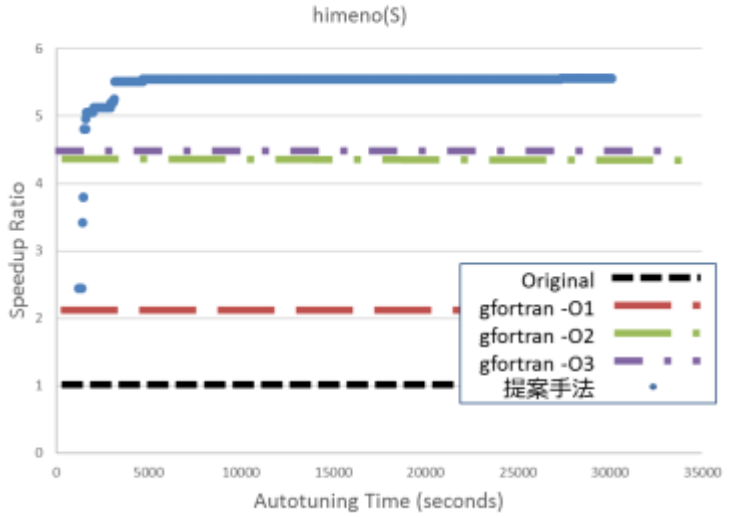
Tuning parameters
- Loop blocking, loop collapse, or no loop optimization (original).
  - ✓ For loop blocking, the block size is also determined.
- Discrete arrays (original) or an array of structures.
- 426 compiler options
  - ✓ -O0/-O1/-O2/-O3
  - ✓ -fexpections, -fwrapv, -funsafe-math-optimizations
  - ✓ -funroll-loops, … etc

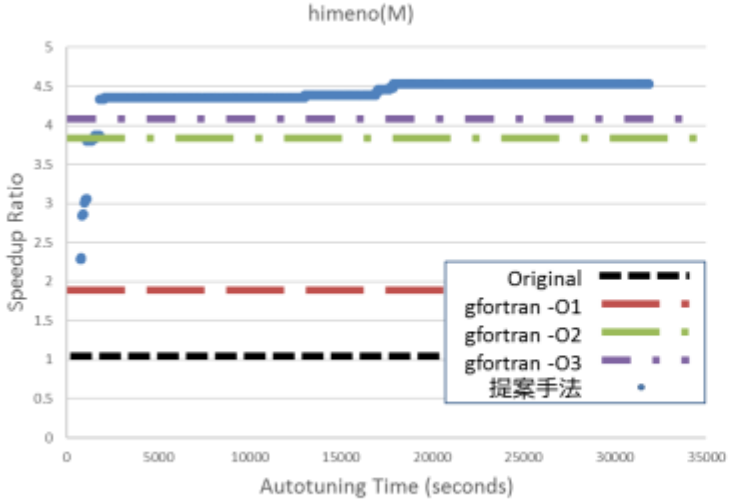


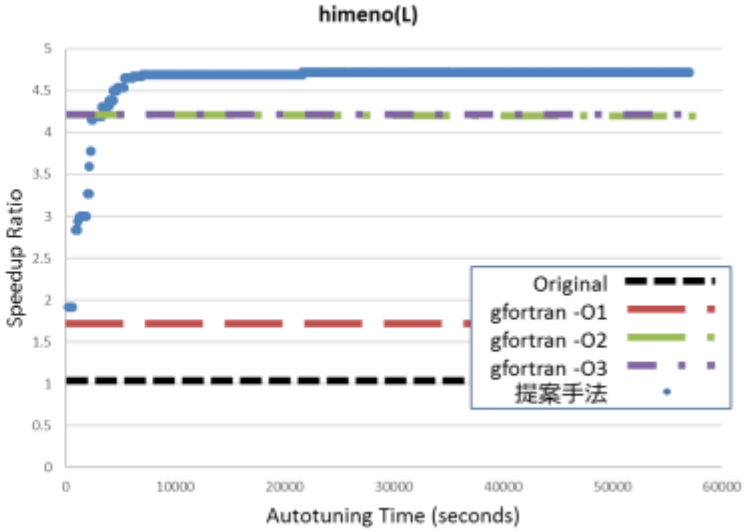
全探索すると **71,944** 秒かかるパラメータ探索問題
OpenTuner はおよそ**3,000** 秒 (4.2%)で同等の性能を達成

性能向上に関する評価


himeno(XS)
Speedup Ratio
Autotuning Time (seconds)
Original
gfortran -O1
gfortran -O2
gfortran -O3
提案手法


himeno(S)
Speedup Ratio
Autotuning Time (seconds)
Original
gfortran -O1
gfortran -O2
gfortran -O3
提案手法


himeno(M)
Speedup Ratio
Autotuning Time (seconds)
Original
gfortran -O1
gfortran -O2
gfortran -O3
提案手法


himeno(L)
Speedup Ratio
Autotuning Time (seconds)
Original
gfortran -O1
gfortran -O2
gfortran -O3
提案手法

AT可能な姫野カーネル

- AT可能なコードは**複雑になりがち**
  - より多くの最適化技法を考慮するためにはより多くの「性能のつまみ」が必要

  → 元はシンプルなカーネルでも複雑化

Original

```
do loop=1、nn
    gosa= 0.0
    do k=2、kmax-1
       do j=2、jmax-1
          do i=2、imax-1
             s0=a(I、J、K、1)*p(I+1、J、K) &
                  +a(I、J、K、2)*p(I、J+1、K) &
                  +a(I、J、K、3)*p(I、J、K+1) &
                  +b(I、J、K、1)*(p(I+1、J+1、K)-p(I+1、J-1、K) &
                                 -p(I-1、J+1、K)+p(I-1、J-1、K)) &
                  +b(I、J、K、2)*(p(I、J+1、K+1)-p(I、J-1、K+1) &
                                 -p(I、J+1、K-1)+p(I、J-1、K-1)) &
                  +b(I、J、K、3)*(p(I+1、J、K+1)-p(I-1、J、K+1) &
                                 -p(I+1、J、K-1)+p(I-1、J、K-1)) &
                  +c(I、J、K、1)*p(I-1、J、K) &
                  +c(I、J、K、2)*p(I、J-1、K) &
                  +c(I、J、K、3)*p(I、J、K-1)+wrk1(I、J、K)
             ss=(s0*a(I、J、K、4)-p(I、J、K))*bnd(I、J、K)
             GOSA=GOSA+SS*SS
             wrk2(I、J、K)=p(I、J、K)+OMEGA *SS
          enddo
       enddo
    enddo
```

Auto-tunable

Unmaintainable?

組み合わせの効果に関する考察

- **生産性**
  - コード変換ルール ：合計**102** 行
    - ループ構造変換のルール 51行
    - データ構造変換のルール 51行
  - 姫野ベンチAT可能化のコード修正行数： 合計**185** 行
    - カーネル部分は **6.5** 倍も長くなっている
- **AT+コード変換の組み合わせが実現すること**
  - アプリ開発者から見える**オリジナルコードは不変**
    - 姫野ベンチのように比較的小規模なコードでも修正行数はルールの行数よりも多い
    - 一般的な実アプリであればより多くの行を修正する必要
  - AT可能化のコード変換ルールがあれば、**個々のシステム向けの専用コード変換ルールは不要になる**かもしれない
    - システムの違いをATが吸収してくれる可能性

アプリ資産の保守性とAT可能性を両立

まとめ

- **ATとコード変換の運命の出会い**
  - ATは一つのコードを複数システムに適応可能
    - 似たようなシステムであれば同一のコード変換ルールを使えるため、コード変換ルール数を削減可能
  - コード変換はアプリコードの複雑化を回避可能
    - アプリ開発者はオリジナルコードだけを保守可能

- **今後の課題**
  - ATとコード変換は**それぞれ独自に開発**されてきた
    - 相互に情報をやりとりするインタフェースの欠如
    - ATしたいパラメータをそれぞれの設定ファイルに記載

  → 連携のためのインタフェースの必要性

興味を持った人は？

- **Xevolverを試してみてください**
  - 詳しくは http://xev.arch.is.tohoku.ac.jp