Vol.2017-HPC-160 No.34
2017/7/28

Polyhedralコンパイラを用いたタイリングパラメータ自動調整ツールのメニーコア環境での評価

幸 朋矢[1,a)]　佐藤 幸紀[1]　遠藤 敏夫[1]

概要：ループ最適化手法の一つであるループタイリングは、近年普及が進む深いメモリ階層を持つメニーコア環境においてデータの局所性を高める手段として重要性が高い。しかしながら、この最適化による性能向上率を左右するタイリングパラメータの調整に関しては現在の Polyhedral コンパイラでは考慮されておらず、このパラメータをアプリや実行環境に合わせて最適化することが更なる性能向上のために望まれている。我々は、まず数種類のカーネル関数に対しタイリングパラメータを変化させた際に性能も同時に変化することを実験により確認した。その知見から最適パラメータに近い結果を出力をする自動調整ツールを開発し Knights Landing 環境において評価を行ったところ、ロードバランスが改善され、さらにそこから最大約 1.4 倍の性能向上が得られた。

1. はじめに

近年の計算機システムでは、コア単体の計算速度だけではなく並列処理やデータの転送速度、消費電力などを考慮した異種アーキテクチャが混在するヘテロ環境がスタンダードとなっている。それぞれのアーキテクチャは特殊かつ独立したメモリ構造を持ち、さらに単一デバイス内で複数のメモリ階層を持つものもあり、システム全体の性能を十分に引き出すためにはこれらのメモリ階層に対応したチューニングをする必要がある。このメモリ階層構造は年々深化し複雑化が進んでいるため、これらのチューニングを手作業で行うことは難しく自動化が望まれている。

ここで代表的なアクセラレータのひとつとして intel の Xeon Phi を例に挙げる。これはメニーコアで並列計算を得意としながらもコアの中身が x86_64 アーキテクチャのため、従来のプログラムからの移行が比較的容易なデバイスとして注目されている。その第 2 世代である Knight Landing(以下 KNL) ではメインプロセッサとして OS ブートから計算まで全て Xeon Phi 内で完結することができ、その利用がさらに容易となった。しかし従来の CPU コードを単純な修正のみでそのまま KNL にて実行するとその性能を十分に引き出せないことも多い。KNL の性能を正しく発揮するためにはそのアーキテクチャの特徴を考慮した最適化を行う必要がある。

多重メモリ階層において性能を十分に引き出すためには、データの局所性について考慮する必要がある。その中でも特に、ループ最適化の手法のひとつであるタイリングが有用である。これはループのスケジューリングを調整することによりデータの局所性を高め、性能を向上させるものである。さらに、ループ最適化の手法として Polyhedral モデルによるループタイリングが近年注目を集めている。これは Polyhedral モデルに適用可能なループ構造でありさえすれば、非常に柔軟なタイリングが静的に自動で計算可能というものである。これによりループタイリングを幅広いカーネルにおいて簡単に適用することができ、その最適化性能を限界近く引き出すことが可能となる。ループ最適化を自動で行う Polyhedral コンパイラは現在いくつか存在するが、タイリングパラメータについてはあまり考慮されていない。そのため、実行時のシステムのメモリ階層構造や実行するアプリ次第で性能向上率は大きく変わるというのが現状である。

本報告では、まず Polyhedral コンパイラにおいてタイリングパラメータを変化させた時に性能向上率も同時に変化することを、3 つのカーネルをピックアップしパラメータを網羅的に実験することにより確認する。ここでカーネルごとにタイリングパラメータ変化の特色が異なることが分かる。この知見を生かし、シンプルな Hill climbing 法を取り入れたタイリングパラメータ自動調整ツール PATT を設計、開発する。このツールを KNL 環境にて評価し、デフォルトのタイリングパラメータに対する速度向上率を見ながらその理由について考察する。

[1] 東京工業大学　学術国際情報センター
[a)] yuki.t.ab@m.titech.ac.jp

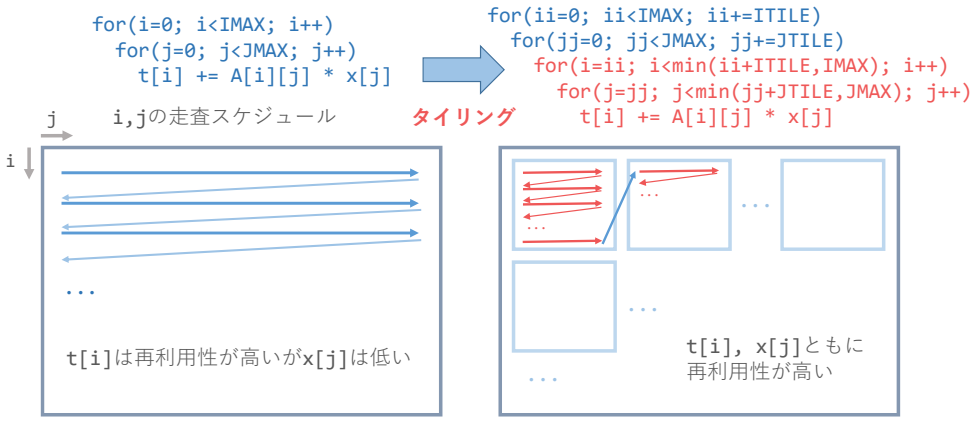

図 1: タイリングのイメージ

2. タイリングと Polyhedral コンパイラ

ここではループ最適化手法のひとつであるタイリングの概要と、Polyhedral コンパイラのひとつである Polly の紹介を行う。

2.1 タイリングの仕組み

タイリングはループ最適化の手法のひとつである。ループのスケジュールをデータの局所性が高まるように変化させることによりデータの再利用性を上げ、メモリ階層ごとのデータ転送回数を減らし、結果として性能が向上することを狙う。例として単純な行列ベクトル積計算の 2 重ループを挙げたものが**図 1** となる。元々の i,j の走査スケジュールでは t[i] は常にキャッシュに存在するが、x[j] は一定時間を超えるとキャッシュから溢れる。i-1 の時に x[j] のデータを取得していたとしても、i の時にキャッシュに存在しなければ下部のキャッシュもしくはメモリからデータを取得しなければならなくなる。この時メモリ階層を跨ぐデータ転送が発生し、性能低下の要因となる。タイリング処理を施すと、i の時に x[j] を取得した場合、i+1 の時もまだ x[j] のデータがキャッシュ内に残っている可能性が高くなる。これによりデータ転送コストを回避することができ、全体の実行速度が上がる。このタイルサイズはキャッシュやホストメモリやアクセラレータのデバイスメモリなど、システム全体のメモリ階層構造に応じて適切な値を設定することが望ましい。しかし各デバイスのメモリ構造はそれぞれ特殊なアーキテクチャをしている上、ワーキングセットサイズなども考慮に入れると、その最適値を理論的に割り出すことは難しい。それぞれのシステム環境やカーネルにおいてそれぞれ最適なタイルサイズがあるが、それを導出するアプローチは様々である。

2.2 Polyhedral モデルと Polly

ループ最適化をより効率的に行うための数学的モデルとして、Polyhedral モデルというものがある。これはループ構造を詳細に分析し、計算結果が変わらない範囲でスケジューリングを変化させられるものである。この時、ハードウェアを考慮に入れた効率的なスケジューリングを選択することで実行性能を上げることが可能となる。

この Polyhedral モデルを利用してループ最適化を行う代表的な Polyhedral コンパイラとして、Polly[1] と Pluto[2] というそれぞれオープンソースのツール 2 つが挙げられる。Pluto は C ソースコードレベルで変換を行い、Polly は LLVM-IR レベルで変換を行う。昨今ポータビリティ性が高く最適化効率も良い LLVM の需要が高まってきているという背景から、今回の研究における Polyhedral コンパイラとしては Polly を採用することとした。

ここで Polly の動作を簡単に説明する。Polly は LLVM-IR を入力とし、まずループの正規化を行い、ユーザー指定のオプションに応じた最適化処理を施し、LLVM-IR を出力する。この時オプションを適切に設定することで、タイリングだけではなくパラレル化、ベクトル化といった最適化処理も同時に行うことができる。Polly は Polyhedral モデルを用いているためスケジューリングの自由が高く、より柔軟なタイリングが可能となっている。

Polly にはタイルサイズ指定のオプションが存在するが、それを指定しなければデフォルトのタイルサイズである 32 が常に使用される。この 32 というパラメータは各ループにつきそれぞれ 32 でタイリングされるという意味である。例えば依存関係のない 3 重ループであればタイリングにより 6 重ループとなり、その時のタイルサイズ、つまり内側の 3 重ループの範囲が 32,32,32 となる。このデフォルトタイルサイズを以降 all32 と表現することにする。尚、Pluto も L1 キャッシュサイズに合わせるというオプションの場合、デフォルトは all32 となる。all32 はどのようなケースでもある程度の性能を出すとして採用されているが、必ずしも最適なパラメータとはなっていない。そのためユーザーが明示的にタイルサイズを指定することでさらなる性

表 1: 実験環境 (バイナリ実行環境)

プロセッサ	Intel Xeon Phi Processor 7210
物理コア数	64
動作周波数	1.3 GHz (1 コアあたり)
OS	CentOS 7.3.1611
コンパイラ	LLVM 5.0.0svn (+Clang+Polly) git 2017/4/24
メモリ	96GB(DDR4)+16GB(MCDRAM フラットモード)

能向上を狙える可能性が残っているが、事前に最適なパラメータを知っていなければその効果的な指定も難しい。

3. タイリングパラメータ調整による性能変化の調査

ここではタイリングパラメータを変化させることで本当に性能が変化するのか、そして変化するのであればどのような傾向があるのかを網羅的な実験により調査する。

3.1 実験方法

実験環境として、Xeon Phi の第 2 世代である Knight Landing 搭載マシンを採用した。これによりメニーコアな環境においてタイリングによる参照局所性向上の効果の調査と、x86_64 バイナリがそのまま動作するため Polly と LLVM にてバイナリを生成し測定するという一連の流れを実現した。詳細な実験環境は表 1 にまとめた。

評価用ベンチマークソフトとして Polybench[3] を採用した。このベンチには Polly が効果的な最適化を施すことができるカーネルが多く含まれており、カーネルもシンプルで比較的解析しやすく、今回の実験に最適だと判断した。また、ステンシル計算カーネルとして別途 himeno カーネルを用意した。これは姫野ベンチマーク [4] を polybench の様式に合わせ、さらに時間ループをカーネル関数の外に配置するように手動修正したものとなる。時間ループがカーネル関数の中にあると polly がタイリング処理を行わないことが分かっており、今回はタイリングの効果を見たいということでこのような処置を施した。

Polly の最適化処理としてパラレル化、ベクトル化、タイリングを行うように、-polly-parallel, -polly-vectorizer=polly, -polly-opt-isl オプションを設定した。さらに llc 実行時には-mattr=+avx2 オプションを設定した。その他のオプションについては説明を省略する。時間計測は 64 スレッドで行い、taskset コマンドにて各物理コアにバインドしてある。また、1 回の実行に要する時間が極端に短いと時間計測結果にブレが生じるため、main 関数内でカーネル関数を数回反復することで実行に時間がかかるような修正を全ベンチマークに施してある。

最初はバイナリのビルドも KNL で行っていたが、KNL の 1 コアは非常に低速で全体としてビルド時間がボトルネックとなるという現象が見られた。そのためバイナリのビルドのみ高速な Xeon マシンで行い、そのバイナリを KNL マシンにて実行するという手法を取っている。

今回タイリングパラメータごとの性能変化を確認するための実験を行うが、数パターンのパラメータを試してみた時点で性能が変化するということは分かっていた。しかしその定量的な考察やパラメータの傾向などを調査するためより網羅的な実験を行う必要があった。今回の実験では、各タイリングパラメータでひとつずつ Polly を通した後バイナリを生成し、時間測定をする。その結果をタイリングパラメータをベースに整理し、ヒートマップにて可視化を行う。予備評価 [5] ではキャッシュミス数や TLB ミス数、TotalAccess 数など各種プロファイルも同時に取っていたが、それらのデータから最速パラメータを割り出すことは難しいという結論に至り、今回の実験では時間計測のみを行った。

3.2 実験結果ヒートマップ

図 2 が実験結果のヒートマップである。2 重ループの代表として atax カーネルを、3 重ループの代表として gemm カーネルをピックアップした。問題サイズは LARGE サイズを指定した。色はバイナリ実行の実測時間となっており、青が時間が小さい=速い、赤が時間がかかる=遅いということになっている。グラデーションをより見やすくするため、最も赤い色の点はその時間以上かかった場合も含むようにヒートマップを作成している。横軸は最内ループの指定タイルサイズであり、左から右へ大きくなる。縦軸は最内のひとつ外側のループの指定タイルサイズであり、上から下へ大きくなる。atax は 2 重ループのため i と j、gemm と himeno は 3 重ループのため内側の 2 重ループである j と k を軸としている。3 重ループカーネルの i のタイルサイズについては事前に最も速かったパラメータを見つけておき、それを指定している。今回の場合、gemm の i のタイルサイズは 16、himeno は 1 となっている。1 はタイリングを行わないというパラメータである。また、このヒートマップで表示されている範囲は必ずしもループや配列のサイズと一致しない。議論したい部分だけを拡大したヒートマップを挙げていることに注意したい。

ヒートマップを眺めると、各カーネルごとにそれぞれタイリングパラメータ変化による速度差の特徴が異なることが分かる。atax の場合は i が小さく、j が大きい場合が概ね速い。gemm は青で潰れているため分かりづらいが、k=12 が最速、次点で k=8 が速い。j は 120 以上であれば概ね速い。himeno は j=1 の付近や k=512(ループサイズは 513) の付近の速度が速いことを考えると、タイリングをなるべくしないようなパラメータが速いということが分かる。

gemm の右下の赤い領域の境界部分は、タイリングされた 3 重ループのワーキングセットサイズが 500 から 600KB 前後になるタイリングパラメータであるということが分かっている。ここで、KNL では 2 つのコアの間に 1MB の

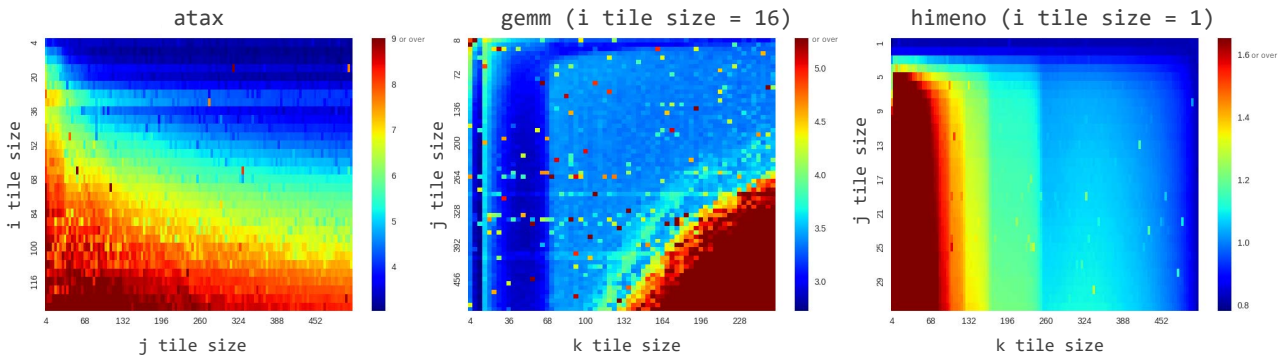

図 2: タイリングパラメータを変化させた時の実行時間を温度としたヒートマップ

L2 キャッシュが存在し、共有して使用されているという構造に着目して考察する。gemm カーネルにおいて 2 つのコアが同時に動いている時、コア両方から使用される共通データが少量存在し、大半がコアそれぞれで使用される共通しないデータであることに注意すると、550KB 前後+550KB 前後-共通データ=1MB 前後となる。よって L2 キャッシュサイズを超えるワーキングセットサイズとなるようなタイルサイズにすると性能が大きく落ちるという考察が可能となる。ちなみに Xeon マシンでも同様の実験をしたところ、こちらでは L1,L2,L3 それぞれのキャッシュサイズを超えるタイミングで速度差の境界が発生することが分かった。本研究においては特に速度の速い部分に着目したいということに加えて、gemm カーネル以外ではこれらの考察が難しいということを踏まえ、この部分の考察は以上としておく。

また、この実験の過程において、パラメータサイズをより細かく取っていくと 4 の倍数以外は遅くなる、といった傾向が数種類のカーネルでは確認できた。さらにパラメータサイズの小さい方の端、つまり 1 や 2 などでは極端に遅くなるカーネルもあった。現時点での Polly では特定のタイリングパラメータを指定すると IR 生成に失敗するケースも見られた。

3.3 高速タイリングパラメータ発見方法についての議論

今回の実験で、少なくともタイリングパラメータを変化させることで性能も大きく変わることは証明できた。しかしその理論的な裏付けとして、ワーキングセットサイズとキャッシュの関係、プリフェッチ、TLB ミス、TotalAccess 数などと様々な方向で広く浅く考察したが、どれも決め手になるものはなかった。キャッシュコンフリクトやパラレル化、ベクトル化なども考慮すると、このような様々な要因が複雑に絡み合った結果複雑な実験結果が出てきたと考えざるを得ない。

ここで、全体的に勾配があることについて言及しておく。どのヒートマップも基本的にはなだらかに性能が変化している。極地が数個しかなく、その極地が複数ある範囲も全体で見ると比較的まとまっているため、その周辺のどこかであれば最小値に近い結果となる。よって何らかの方法で繰り返し測定、探索することで最適タイリングパラメータに徐々に寄せていくことは可能である。

また、この実験の過程で 3 重ループのカーネルに関しては特にそれがパーフェクトループの場合、最外ループのタイルサイズはロードバランスと強い関係性があることが分かった。これは Polly が最外ループのみパラレル化するためである。特に小さい問題サイズかつ多いスレッド数という状況において、all32 はロードバランスが酷くなることが多く、この改善の調整をするだけで大きな速度向上が得られた。

4. タイリングパラメータ自動調整ツール PATT の開発

このような状況の中、なるべく性能が高くなるタイリングパラメータを自動的に導出する実用的なツールを開発したいと我々は考えた。前述の通り理論的アプローチは難しく、調査に時間をかけても目的を達成できるかどうかも定かではない。また様々な環境やカーネルにおいて広く使えるツールの方がより実用的であることを考えると、静的な導出より動的なアプローチの方がより汎用性があると判断した。

探索問題として見た時に、ヒートマップ結果において全体的に勾配があることに着目し、シンプルな Hill climbing 法 [6] に少しの調整を加えるだけで比較的良好な結果が得られるであろうと考えた。ひとつのタイリングパラメータの評価のために Polly を通したバイナリ生成と実行時間の測定をそれぞれ行うので全体としてそれなりに時間はかかるが、この方法ならばどのような環境やカーネルであっても同じように動作し、性能が良好なタイリングパラメータを出力することが可能となる。

ここでは我々が開発したタイリングパラメータ自動調整ツールを PATT (Polyhedral compiler based Auto Tile size opTimizer) と呼ぶこととする。このセクションでは PATT の具体的なインターフェイスと動作アルゴリズムに

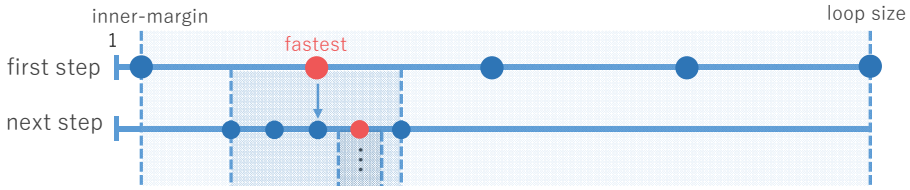

図 3: PATT の探索アルゴリズムイメージ

ついて解説する。

4.1 インターフェイス

PATT はコンパイラのように使用することができる。引数として PATT の設定ファイルをひとつだけ読み込む以外は、通常のコンパイラを使用する時のようにソースコードやオプションをそのまま記述すれば良い。ただしオプションのパーサー部分の実装は現在のところ部分的にしか為されていない。

設定ファイルにはカーネル関数を含むファイルの名前とパス、カーネル関数内の各ループのループサイズ、スレッド数、N_P の値と探索カット機能の on/off(後述)、実行したいモードを記述する。モードは、タイリングパラメータ指定なしでビルド、ユーザー指定のパラメータでビルド、パラメータを自動で調整してビルド、の 3 種類から選択する。ここから分かる通り PATT は Polly のラッパーとしても動作するように設計してあるため、タイリングパラメータ調整機能を使わなくても Polyhedral コンパイラのフロントエンドとして使える。この時 Polly に渡す各種最適化パスは出力バイナリが最も速くなるように自動的に設定される。尚、現在の実装ではループサイズをユーザー指定としているが、この情報の取得は後々自動化する予定である。

4.2 アルゴリズム

ここではタイリングパラメータ自動調整モードを指定した場合の PATT の動作について解説する。PATT の探索アルゴリズムは基本的にはシンプルな Hill climbing 法 (時間計測で評価するため実際は山を下っている) を採用しており、その一次元でのイメージは**図 3** のようになる。まずループサイズ全体において数点の測定ポイントを列挙する。ここではシンプルにループサイズ全体から等間隔で取っている。この時の測定する点の数を N_P(Number of Points) と呼ぶこととする。この点それぞれに対してバイナリを作成し、時間計測を行い、一番速かった点を今回の最速点とする。次のステップではこの点の近傍を探索範囲とし、再度 N_P 個の点を測定し、最速点を見つける。この動作を繰り返して十分に範囲が小さくなったところで探索を終了する。実際にはこの流れを 2 次元で行う。

3 次元の探索を行うと非常に時間がかかるため、最内の 2 重ループだけ 2 次元の探索とし、それ以上は 1 次元ずつ探索するとしている。つまり、元々 2 重ループであれば i,j を同時に探索し、3 重ループであればまず i を決め、次に j,k について同時に探索する。この時、3 重ループの i のタイルサイズは事前に選別した 11 種類の点を全てを測定してその中のひとつを選んでいる。

探索時、1 ステップにおいて N_P 個の点を調べるがこの時勾配があることに着目すると、ある点において速度向上が見込めなかった時点で次の点の計測を省略することができる。これにより測定点の大幅な削減ができ、最終的なタイリングパラメータ導出までの時間を縮めることが可能となる。この探索打ち切り機能は PATT の設定ファイルからユーザーが on/off を選べるようになっている。

また、ヒートマップ実験の知見を生かし、キリの悪いパラメータを排除するための alignment 定数、極端に遅くなる可能性のある端の点を排除するための inner-margin 定数をそれぞれ用意してある。例えば、alignment が 4 であれば 4 の倍数のみを調べるようになり、inner-margin が 8 であれば 8 未満のパラメータは調べないという動作になる。現在の実装では 3 重ループでは alignment=4,marign=8(i 除く)、2 重ループでは alignment=4,margin=4、それ以外では mod=8,margin=16 としてある。これは各種実験を繰り返した経験則から来る値である。

5. タイリングパラメータ自動調整ツールの評価

ここでは PATT の評価実験を行い、その結果について議論する。

5.1 実験条件

基本的な実験環境はヒートマップ実験と同じだが、今回は評価ベンチマークを増やした。polybench の中から 2 重ループカーネル (atax, gemver, gesummv, mvt, jacobi-2d) を 5 つと、3 重ループカーネルを 7 つ (gemm, 2mm, 3mm, covariance, correlation, syrk, syr2k)、総計 12 カーネルを用意した。さらに LARGE サイズ (以下 L とする) と EXTRALARGE サイズ (以下 XL とする) に加え、さらに大きい XXL サイズ (XL 問題サイズの 8〜10 倍) を作成して用意し、このそれぞれについて実験を行った。PATT の設定としては探索カット機能はオンとし、N_P は 8 とした。尚、polybench 全てのカーネルについて実験しなかった理

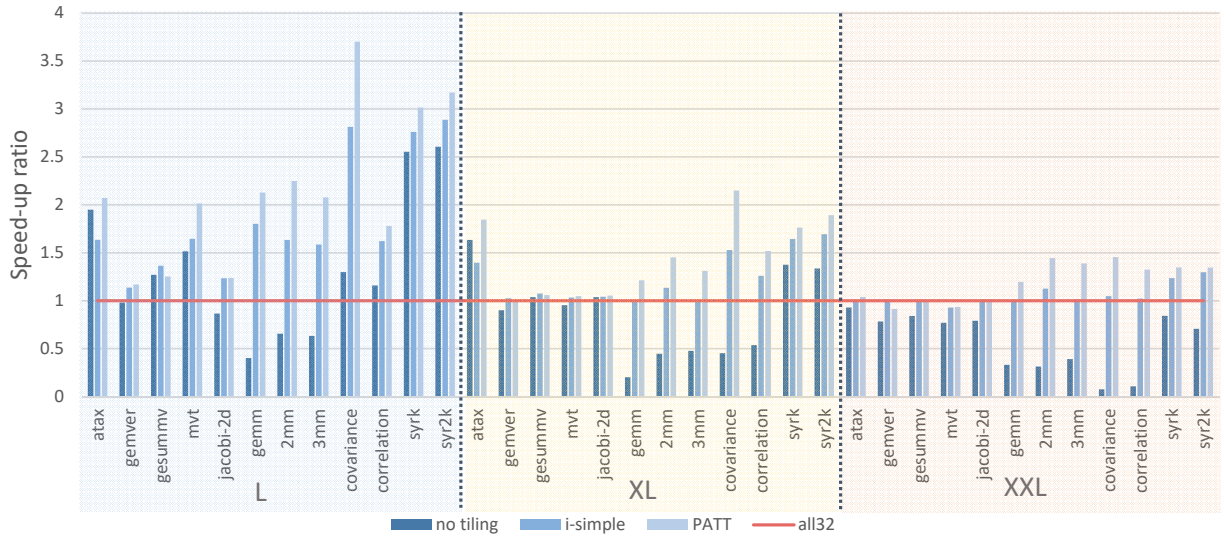

図 4: all32 を 1 とした時の各種スピードアップ

表 2: 時間計測の実測値と PATT の出力パラメータ

Kernel Info	Exe	Make [s]	Time [s]	Parameter
atax	all32	2.72	6.61	32,32
Loop Size	fastest	—	3.05	8,512
1900,2100	PATT	746.49	3.19	8,584
gemm	all32	3.93	6.24	32,32,32
Loop Size	fastest	—	2.73	16,488,12
1000,1200,1100	PATT	573.70	2.93	16,348,8
himeno	all32	7.72	30.6	32,32,32
Loop Size	fastest	—	5.87	1,1,4
257,257,513	PATT	1927.60	6.10	1,8,512

由は、そのまま Polly に通してもタイリングの最適化処理が行われないカーネルがあったためである。また、ブレを十分に防止するために各時間測定は 3 回計測をしてその中央値を取っている。

5.2 デフォルトタイリングパラメータに対する評価結果

all32 を 1 とした時の各種バイナリの性能向上率を図 4 にまとめた。all32 はタイリングパラメータを指定しないことにより自動的に Polly のデフォルトタイリングパラメータである 32 が各次元に適用されたもの、no tiling は Polly に-polly-tiling=false オプションを付けることでタイリング最適化を無効にしたもの、i-simple は PATT が出力したパラメータのうち最外ループである i のみ採用して他は全て 32 を指定したものとなっている。PATT は $N_P = 8$ とした場合の結果を載せてあるが、実験の過程においてこのパラメータを変えてもあまり性能差は出ず、全体的な傾向は同じであることを確認している。

まずタイリングの効果について考察する。no tiling と all32 を比較すると、gemm などの 3 重ループカーネルの一部ではタイリングをした場合の方が速いが、atax などでは下手にタイリングすると性能が落ちている。このような場合でも PATT によりタイルサイズを調整することで、タイリングしない場合と同等かそれ以上の性能にすることが可能であることが見て取れる。また、XXL サイズの covariance に着目すると no tiling から all32 にしただけで 13 倍近くの速度向上となっている。これはタイリング最適化が本来意図する性能向上とは別の要因の存在が予想されるが、現時点で詳細な調査はできていない。タイリングパラメータを変化させると単純にデータの参照局所性の与える影響のみが性能に反映されるのではなく、他にも要因が存在するとして結果を考察する必要がある。

次に、各速度向上率について考察する。all32 と PATT を比較した場合、最も効果があったのは L サイズの covariance で約 3.7 倍となった。全体的に見ると概ね性能向上が見られているが、XL,XXL サイズと問題サイズを大きくするに従って性能向上率が段々と落ち込んでいくことが分かる。これはロードバランスの改善の恩恵が性能向上の大きい割合を占めているためだと思われる。Polybench の L サイズのような小さい問題サイズに対し、KNL のようなメニーコア環境で多くのスレッド数で並列計算する場合、32 というタイルサイズはスレッドを十分に生かしきれないケースが多い。例えば、gemm(L サイズ) の最外ループサイズは 1000 だが、これを 32 でタイリングするとタイリング後の最外ループは 32 回のイテレーションとなり、これは 64 スレッドで並列化するには少ない。この場合はスレッド数を減らすか、タイルサイズを小さくしなければならない。特に L サイズの all32 ではこのようなロードバランスの悪さが性能に大きな悪影響を与えている。そこで i-simple という、最外ループのタイルサイズのみを調整することでロードバランスのみを改善した評価軸を用意してある。これと all32 を比較することにより、性能向上の内訳としてロードバランスの改善が大きいことと、PATT と比較することによりそこからさらに性能向上の余地があることが見て取

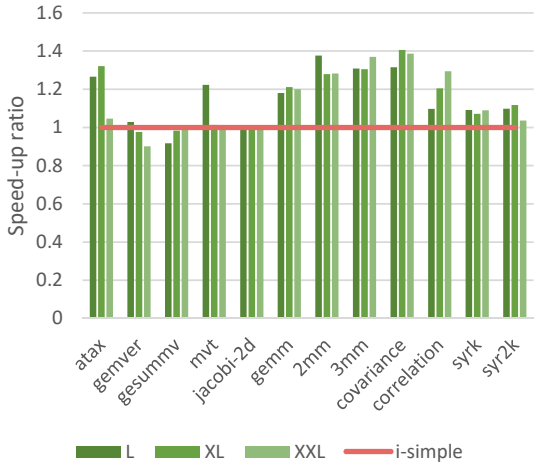

図 5: i-simple を 1 とした時の PATT のスピードアップ

れる。

表 2 にはヒートマップを取った 3 つのカーネルをピックアップし、そのループサイズとビルドに要した時間 (Make 列)、バイナリの計測時間 (Time 列)、及び各タイリングパラメータをそれぞれまとめた。fastest の行には前述したヒートマップ実験の結果から得られた最速パラメータを載せた。PATT の自動調整機能をオンにすると MakeTime が非常にかかることが見て取れるが、これは内部で何度もバイナリを作成して測定するということを繰り返しながらパラメータを探索しているためである。fastest と PATT を比較すると、出力パラメータは必ずしも一致しないものの概ね近いものとなっていて、計測時間に関してはかなり fastest に近くなっていることが分かる。

5.3 ロードバランス改善後のパラメータに対する評価結果

図 5 に、i-simple を 1 とした時の PATT の性能向上率をまとめた。この 2 つのバイナリの差としては、i は同じパラメータ、2 重ループカーネルの場合は j のみ 3 重ループカーネルの場合は j,k のパラメータが異なるということになっている。ここでは最外ループのタイリングパラメータはロードバランスに関係するとして切り分け、内側ループのタイリングパラメータ調整の恩恵について考察する。このグラフを見ると、カーネルごとに傾向が違うが、概ねサイズに依らず一定の性能向上があることが分かる。特に 3 重ループカーネルでは最大で 1.4 倍の性能向上が見られた。

ロードバランスが重要とされる場面においては、i-simple のように最外ループのタイリングパラメータ調整のみを行うだけで大きな性能向上が得られる場合が多い。そこからさらに内側ループのタイリングパラメータ調整を行うことでさらに最大で 1.4 倍程度の性能向上が見込めることをこの結果は示している。この時、特に 32 以下などの小さいタイルサイズをいくつか調べるだけで簡単に最適パラメータが見つかる可能性がある。詳細に調べたい場合は PATT のように広くパラメータを探索するとさらに最適パラメータに近付く確率が上がる。ここまでの処理は PATT であれば全て自動的に行うが、PATT でも手動であっても多くのパラメータの探索には時間がかかるため、どこまでの性能をどれだけのコストで求めたいかによって掘り下げるレベルを判断する必要がある。

6. 関連研究

Mehta らは、キャッシュサイズ等に着目して静的にタイルサイズ導出を目指す TurboTiling[7] というツールを開発している。しかし彼等の理論的アプローチによる静的なパラメータ導出モデルが幅広いケースにおいて適用できるとは限らない。一方で、我々は自動チューニングをベースとして幅広い適用範囲を目指している。

Jain らは、クラウドなどリソースが動的に変化するような環境を想定した最適化を行う ShapeShifter[8] というツールを開発している。しかしながら、リソースがある程度安定している上でメニーコアにて実行する我々の想定環境とは異っている。

[9] において白子らは、解析モデルにより経験的に探索空間を効率良く制限する手法を提案している。彼等はキャッシュヒットの上限と下限をモデリングした DL/ML モデルを利用して、探索空間の枝刈りを行っている。しかしながら彼等の手法はシングルスレッド実行での評価をベースとしており、メニーコア環境で問題となるロードバランスに関しては言及していない。

Ogilvie らは、反復的にコンパイルを行いながら最適なコンパイルのパラメータを探索するというアプローチにおいて、機械学習アルゴリズムの 1 つである Active Learning を使うことを提案している [10]。計測におけるノイズへの耐性も利点と主張している反面、現状をブラックボックスとして扱ってしまうため、自動チューニングに基づく経験的な手法とは一長一短があると考えられる。

Prajapati らは、[11] において GPU 向けコード上でタイリングを行う際のタイルサイズについて論じている。彼らの評価においては、ブロック当たりのスレッド数が重要なパラメータであると示唆している点では PATT が最外ループ i のサイズを最初に独立して決めるアイディアと通じるものがある。しかしながら、GPU のスレッドとメニーコア CPU のスレッドは粒度が大きく異なるため、異なるチューニングのアプローチが必要である。

7. まとめと今後の課題

本報告では、Polyhedral コンパイラのタイリングに着目し、そのタイリングパラメータを変化させることで性能が大きく変わることを確認し、その知見を生かしたパラメータ自動調整ツールを開発した。KNL 環境においてその評価を行った結果、タイリングパラメータ調整の重要性と有効性を示すことができた。

現在この速度向上の内訳として大きい部分を占めているロードバランスに関して、その詳細な調査を行っている。また、alignment や inner-margin といった定数値についても現在は経験則に基づき設定しており、より一般化が求められる。その他にもワーキングセットサイズとメモリ階層の関連付けができるとよりツール開発の方向性が明確になり、さらに部分的に静的なパラメータ導出も期待できるようになるため、こちらのアプローチでも調査を進めたいと考えている。

この研究を進めることでさらに PATT の精度を上げ、ユーザーにタイリングパラメータの調整を意識させることなくタイリングにおける最大限の性能を出すことを目指す。来る Polyhedral コンパイラのメニーコア環境においての実用化を想定し、PATT の実用化及びこれらの知見を生かした様々なケースでのタイリングパラメータ導出手法の提案をしていきたいと考えている。

謝辞

本研究は、 JST 、CREST の支援を受けたものである。

参考文献

[1] Grosser, T., Groesslinger, A. and Lengauer, C.: Polly - Performing polyhedral optimizations on a low-level intermediate representation, *Parallel Processing Letters*, Vol. 22, No. 04, pp. 1–28 (2012).
[2] Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer, *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pp. 101–113 (2008).
[3] Yuki, T. and Pouchet, L.-N.: PolyBench, `https://sourceforge.net/projects/polybench` (2016).
[4] Himeno benchmark: `http://accc.riken.jp/en/supercom/himenobmt/` (2017).
[5] 佐藤幸紀，　幸朋矢，遠藤敏夫ほか：透過的メモリ階層チューニングのための動的バイナリ変換機構の設計と開発，Vol. 2017, No. 35 (2017).
[6] Russell, S. J. and Norvig, P.: *Artificial Intelligence: A Modern Approach*, Pearson Education, 3 edition (2009).
[7] Mehta, S., Garg, R., Trivedi, N. and Yew, P.-C.: TurboTiling: Leveraging Prefetching to Boost Performance of Tiled Codes, *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pp. 38:1–38:12 (2016).
[8] Jain, A., Laurenzano, M. A., Tang, L. and Mars, J.: Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting, *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12 (2016).
[9] Shirako, J., Sharma, K., Fauzia, N., Pouchet, L.-N., Ramanujam, J., Sadayappan, P. and Sarkar, V.: Analytical Bounds for Optimal Tile Size Selection, *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pp. 101–121 (2012).
[10] Ogilvie, W. F., Petoumenos, P., Wang, Z. and Leather, H.: Minimizing the Cost of Iterative Compilation with Active Learning, *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pp. 245–256 (2017).
[11] Prajapati, N., Ranasinghe, W., Rajopadhye, S., Andonov, R., Djidjev, H. and Grosser, T.: Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils, *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pp. 163–177 (2017).