修士論文

Master Thesis

高い合成可能性と性能可搬性のための再帰的な
ブロッキングに基づく効率的な行列積演算の並列化

Efficient Parallelization of Matrix Multiplication Based on
Recursive Blocking for High Composability and
Performance Portability

高橋　大成

Taisei Takahahashi

学籍番号 48-206435

東京大学大学院 情報理工学系研究科 電子情報学専攻

Department of Information and Communication Engineering
Graduate School of Information Science and Technology
The University of Tokyo

指導教員　　田浦 健次朗 教授

Advisor　　Prof. Kenjiro Taura

要旨

近年の深層学習技術等の発展により，プログラムにおいて行列積演算を必要とする機会が増し，演算を行う環境も多様となっている．しかし，BLAS ライブラリを始めとする既存の線形代数演算を行うライブラリでは性能を維持したまま違う環境へ移植するコストがとても高い上，並列性がネストした状態では性能低下が引き起こされることが指摘されている．そこで，本研究では実行環境に依存しない形で実装を行うことができ，プログラム中で並列性がネストした状態が生じても性能の低下が起きにくいタスク並列という並列化に注目する．本研究は並列化した際にキャッシュ効率を損なわないような空間充填曲線に基づく行列のメモリレイアウトと高性能なタスク並列スケジューラを組み合わせることで，高性能なタスク並列な行列積についての提案を行う．本手法では並列性がネストしていない通常の並列実行状態において既存の並列化された BLAS ライブラリに対して小さな行列サイズの場合に対して 3% の性能向上を達成し，並列性がネストした状態においては既存の並列化されたライブラリよりも高い性能を得ることができた．また，問題を再帰的な分割によって処理する問題サイズを小さくしているため，実行環境に依存しない性能の可搬性を得ることができた．

目次

第 1 章

序論

1.1　導入

近年の深層学習技術を用いた人工知能分野の著しい発展や物理シミュレーションなどの高度な数値計算を必要とする研究の発展により，実験等においてコンピュータ上でソフトウェアやプログラムを動かす際に，行列やベクトルが含まれた線形代数演算を必要とする機会がとても多くなってきている．深層学習分野においては学習フェーズや推論フェーズのほとんどの計算が行列とベクトルが含まれた演算であり，この行列とベクトル間の演算が実行時間の多くの時間を占めている．このような行列積演算を始めとする行列やベクトルの基本的な計算する関数群を含むライブラリとして BLAS（Basic Linear Algebra Subprograms）ライブラリと言われるものが存在している．現在，深層学習分野などで用いられ，Python において大規模な高水準の数学関数を提供するライブラリである Numpy にも内部実装として，いくつかの代表的な BLAS ライブラリが用いられている．

また，近年の CPU 開発は CPU におけるコアのベースクロックを向上させる方向性ではなく，CPU1 ソケットあたりのコア数を増やすという方向性で CPU の性能向上が図られているというトレンドがある．そのため，CPU の性能を出来る限り引き出すためにはメニーコア環境を上手に活用したソフトウェア実装というものが必要不可欠になってきている．広く用いられている BLAS ライブラリは，マシンのメニーコアな CPU 環境を効率よく使用するために線形代数演算を並列に実行するような実装がなされているが，その並列性の実装においてマシン上にあるコア等の全てのリソースを占有して実行できることが前提とされている．そのため，並列化実装がされている BLAS ライブラリ自体を複数個並列で実行するような並列性がネストした状況や並列数がコア数を超えてしまう状況などをを考えた際，リソースマネジメントが上手く行われずにマシンの性能を十分に引き出すことができなくなる場合が存在する．

本研究では，BLAS ライブラリに含まれる代表的な演算の一つである一般的な行列同士の積の演算に焦点を当てる．そして，並列性がネストした状況下において．既存の BLAS ライブラリよりも性能低下の割合少ない分割統治法による再帰的なブロッキングを用いた行列積演算を提案する．また，BLAS ライブラリの多くは既存の様々な CPU のアーキテクチャに対してそれぞれの最適化がなされている．そのため，新たなアーキテクチャの CPU が使われるようになった場合は十分な

性能を得るために新しいアーキテクチャに対しても最適化を行う必要があり，性能可搬性を高めるためのコストが高い．本研究で提案する行列積演算手法は cache-obliviou なアルゴリズムであるため，異なるアーキテクチャの CPU に移植した場合でも高い性能可搬性を持つことがわかった．

1.2 研究背景

行列積演算を実行する際に一般的に用いられる BLAS ライブラリの概要と現在の BLAS ライブラリの使用における問題点や，BLAS ライブラリでの内部実装で用いられている行列積演算のための最適化手法について最初に説明する．その後，本研究の提案手法に用いられている分割統治法や空間充填曲線，行列積演算の並列性がネストする線形代数計算の一つとして三重対角行列の固有値分解について記述する．

1.2.1 BLAS ライブラリとは

BLAS ライブラリとは，主に BLAS(Basic Linear Algebra Subprograms)）[28][19][18] と言われる線形代数演算ルーチンを持っているライブラリのことを指す．BLAS と言われている線形代数演算のルーチン郡は主に level-1, level-2, levle-3 の 3 つのレベルに分けらている．

level-1 のカテゴリには，主にスカラーとベクトルによる演算とベクトルとベクトルによる演算に関するルーチンについて分類されている．level-2 のカテゴリにはベクトルと行列の演算に関するルーチンについて，level-3 のカテゴリには行列と行列の演算に関するルーチンについてがそれぞれ分類されている．level-1 や level-2 に属するルーチンに関しては計算の密度を上げることができないため，実装方法の違いによるで演算性能の差異は出にくい．しかしながら，level3 のルーチンである行列同士の演算に関しては実装方法などにより CPU の理論性能に迫るほどの高速化を実現できる余地があるとされている．そのため，BLAS の高速化に関する研究の多くは level-3 のルーチンである行列同士の演算に関してなされている．

この level-3 ルーチンには一般的な行列と行列の積以外にも対称行列同士やエルミート行列同士の積，エルミート行列のランク更新など様々な行列と行列における演算が存在している．その中でも一般的な行列積である GEMM(General Matrix Multiply) という演算が用いられることが多く，この GEMM というルーチンが一種の BLAS ライブラリの性能を測るベンチマークとして扱われることもある．GEMM で行われる計算については図 1.1 に概要を示す．

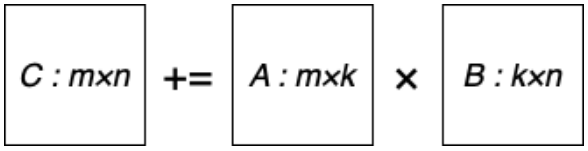


図 1.1: 典型的な行列積

1.2.2 行列積演算の一般的な高速化

行列積演算

一般的な行列積演算は簡単には以下のように 3 重のループを用いることよって実装できる．

アルゴリズム 1 $C = A \times B$ の簡単な実装

```
for i = 0 ... M do
    for j = 0 ... N do
        for k = 0 ... K do
            C[i][j] += A[i][k] × B[k][j]
        end for
    end for
end for
```

しかし，これでは単純な実装であって十分に速い実装ではない．行列積演算を高速化するためのループ交換，ループアンローリング，キャッシュブロッキングなどのいくつかのテクニックが存在している．

Strassen のアルゴリズム [41] などを除き，ほとんどの行列積を計算するアルゴリズムにおいて計算量は $O(N^3)$ で変わらない．Strassen のアルゴリズム等も計算量自体は若干少ないものの，その他の要因によって実行時間はあまり速く場合も存在する．そのような計算量のが固定の行列積の中で，変更できることは各行列の要素同士の計算の順序のみである．要素同士の計算の順番を変えることにより，演算に用いるデータの局所性が上がり性能効率の良い実装をすることができるようになる．

キャッシュとメモリの階層構造

一般的な行列積の高速化はキャッシュを有効に使うということに注目して行われることがとても多く，キャッシュの仕組みを理解するということがとても重要である．

メモリ素子技術と CPU 素子技術の差異等の原因などにより，CPU の処理速度に比べ，メモリへのアクセスにかかる時間が大きい．そのため，通常は処理に必要なデータをメモリから演算のレジスタに移動するまでの間，CPU はストールした状態となってしまい計算の効率が悪くなってしまう．

プログラムの実行において，ある命令が実行された場合はその命令で参照されたメモリ語自身や周りのメモリ語が続いて参照されやすいという参照の局所性が存在する．この参照の局所性を有効に利用するために，現在のコンピュータではメモリと CPU レジスタの間にメモリよりも小規模だがアクセス速度の高速なキャッシュを用意している．これにより，CPU はメモリから直接データを読み書きするのではなく，次に参照される可能性が高いデータを前もって入れておいたキャッシュとより高速なデータのやりとりができるようになる．メモリ階層の概略を図 1.2 に示す．

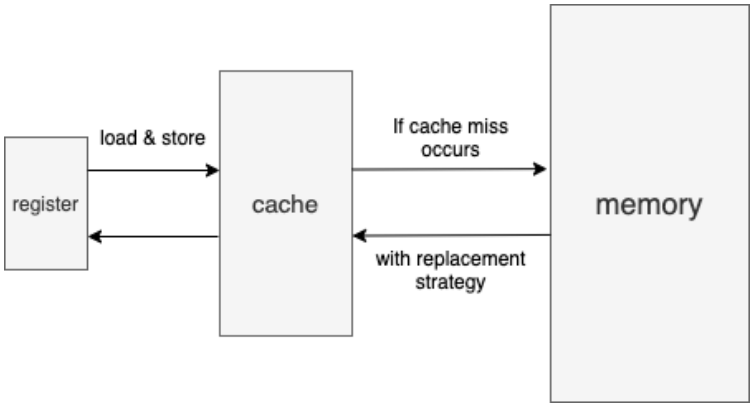


図 1.2: 計算機におけるメモリ階層

CPU はキャッシュとデータのやりとりをすることになるが，命令実行に必要なデータがキャッシュ上に存在しなかった場合は，キャッシュミスとなりアクセスに時間のかかるメモリにデータを参照しにいかねばならない．キャッシュミスを減らし，効率的なキャッシュの参照というものが，プログラムを高速化する一つの要因となる．

近年のコンピュータでは上層の 1 次キャッシュ (L1) から下層の 3 次キャッシュ (L3) まで用意されていることが多く，下層ほど大容量だがアクセスのコストが高くなる．現在主流のアーキテクチャでは, 一般的に L1 キャッシュと L2 キャッシュが各コアごとに存在し，L3 キャッシュはソケットにつき一つ，つまり全てのコアで共有されている．

より高速にプログラムを実行することを考えた場合，L1 キャッシュのみに対してなど，ある特定のレベルのキャッシュのみを考慮するだけではなく，全てのレベルのキャッシュを考慮してデータ構造を決めてプログラムの最適化をする必要がある．しかし，特定のアーキテクチャに最適化したことによる性能可搬性の低下を招く場合が多々ある．

ループ交換とループアンローリング

ループ交換とは文字通り，ループの順番を交換するという手法である．これは主に言語の使用に依存する．ここでは例として C 言語でのループ交換を考えてみる．C 言語において 2 次元配列は row-major order によってメモリに格納される．行方向の連続性を活かせ方法としては現在の i-j-k という順番のループから，i-k-j という順番のループに変更するという方法が考えられる．こうすることにより，キャッシュの空間的局所性をうまく使えるようになり，メモリアクセスの回数が減るため，プログラム自体の高速化が図れる．この走査されるパラメータの選び方についてはメモリに行列の要素をどのような順番で載せるのかによって変わる．

ループアンローリングとはアンロールという名前の通りではあるが，ループとして巻かれていたものを広げるという意味である．ループの範囲を狭めて，本来ループの中で処理されていたものの一部をあえて手動で実装するというものである．プログラム内で for 文などを用いて実装された

アルゴリズム 2 $C = A \times B$ に対するループ交換

```
for i = 0 ... M do
    for k = 0 ... K do
        for j = 0 ... N do
            C[i][j] += A[i][k] × B[k][j]
        end for
    end for
end for
```

ループが実行される際には分岐命令といわれる条件判定に関する命令が発行される．分岐命令は通常の命令と比べて処理が多い上，時間がかかってしまう．そのため，ループ回数を減らし，オーバーヘッドを減少させるという工夫が取られている．

また，ループをまたいで同じメモリにアクセスするということを減らすことができ，メモリのロードやストアの回数を減らすことも期待できる．ループを展開する数を増やしていくとオーバーヘッドは小さくなるが，上限は存在する．あまり展開しすぎるとレジスタ不足や命令のキャッシュミスを引き起こしてしまうため，プログラムを実行するアーキテクチャにあった調整が必要である．

以下に，ループ入れ替えをして i, k それぞれに対してループアンローリングを 2 段行ったものを示す．

このようなループアンローリングの段数を決めるのに対して，アーキテクチャの命令セットやレジスタの個数などの様々なパラメータを考慮して慎重に決定し，最適化が必要である必要がある．そのため，特定のアーキテクチャに最適化された BLAS ライブラリではそのアーキテクチャの適したアンローリングの設定がなされている．

アルゴリズム 3 $C = A \times B$ に対するループアンローリング

```
for i = 0 ... M, 2 do
    for k = 0 ... K, 2 do
        for j = 0 ... N do
            C[i][j] += A[i][k] × B[k][j]
            C[i][j] += A[i][k+1] × B[k+1][j]
            C[i+1][j] += A[i+1][k] × B[k][j]
            C[i+1][j] += A[i+1][k] × B[k][j]
        end for
    end for
end for
```

ブロッキング

キャッシュブロッキングというものは簡単にはキャッシュのサイズに合うように演算をある程度まとめてブロック化し，演算をする際にキャッシュを効率的に使えるようにするという手法である．キャッシュに収まるデータへのアクセスは通常のメモリアクセスと比べ格段に早いため，キャッシュにデータを入れておくことでキャッシュミスを減らすことができる上，時間的な局所性をカバーし，非常に効率的な高速化が見込まれる．

このキャッシュに関しても，L1 キャッシュや L2 キャッシュ，L3 キャッシュとった種類がある．また，それぞれのアーキテクチャごとに収まるデータサイズが違っていたり，構造的に保持しているキャッシュの種類が異なる．そのため，各アーキテクチャのキャッシュ構造やキャッシュのサイズに収まるように問題を細分化して小さいブロックにしていく必要がある．

簡単に各行列を小さい行列にブロック化をするだけでもキャッシュを有効に利用することができ，プログラムの高速化に繋がる．効率の良いブロッキングを行う場合のブロッキングの幅はキャッシュラインの構成やサイズなどを考慮して決定する必要がある．そのため，特定のアーキテクチャに対するキャッシュブロッキングの最適化を行う場合は，キャッシュのサイズやキャッシュにおけるラインのサイズ，演算器までのレイテンシ等をみる必要がある．使用頻度の高いデータの量はキャッシュのサイズに収まるようにすることや，そのデータをさらにキャッシュラインのサイズで小分けにしておくこと，レイテンシの時間は他の操作を行いオーバーラップさせることなどを考える必要が出てくる．

実際にキャッシュブロッキングを考える際には，行列をさらに小さい行列に分解し，その小さい行列同士の演算を考えていくことで演算に必要なデータサイズを小さくしている．簡単にブロックサイズ（ブロッキング幅）を 16 として，通常の行列積演算に対してブロック化を行う概要をアルゴリズム 4 に示す．

1.2.3　行列積の並列化

SIMD 並列化

SIMD というのは Simple Instruction Multiple Data の略であり，単一命令を複数のデータに対して同時に適用するというフリンの分類における並列化手法の一つである．複数のデータに対して一つの命令で処理を行うため，スループットが向上して高い性能を得ることが可能になる．

この SIMD による並列化は，SIMD 命令と呼ばれる拡張命令の実行により実現できる．比較的新しい CPU では SSE や AVX といった拡張命令セットに対応しており, ユーザーはこの拡張命令を用いることで簡単に SIMD 並列化を利用できるようになっている．

例えば，AVX-512 という拡張命令セットでは，512bit の SIMD レジスタが利用可能であり，同一の演算であれば 16 個の単精度浮動小数点演算または 8 個の倍精度浮動小数点演算が 1 コアの 1 クロックあたりに可能となる．

プログラマが明示的にアセンブリを用いて書くほかに，コンパイラによる自動ベクトル (SIMD)

アルゴリズム 4 Blocking for $C = A \times B$

```
block_size = 16
for ib = 0 ... M, block_size do
    for jb = 0 ... N, block_size do
        for kb = 0 ... K, block_size do
            for i = ib ... ib + block_size do
                for j = jb ... jb + block_size do
                    for k = kb ... kb + block_size do
                        C[i][j] += A[i][k] × B[k][j]
                    end for
                end for
            end for
        end for
    end for
end for
```

化や BLAS ライブラリ等のすでに並列最適化がなされているライブラリを用いることでこの SIMD 命令を活用することができる．

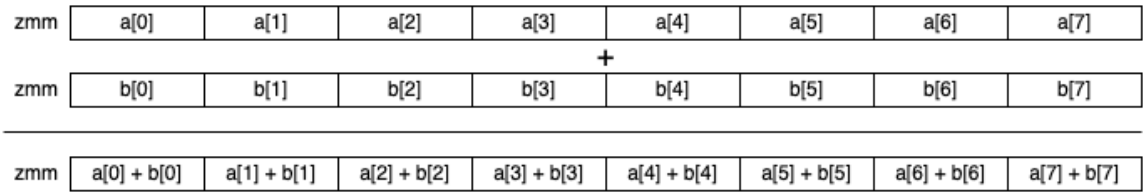


図 1.3: AVX-512 における加算命令

MIMD 並列化

MIMD というのは Multiple Instruction Multiple Data の略であり，複数の命令を複数のデータに対して同時に適用するというフリンの分類における並列化手法の一つである．複数のコアを持つ CPU において各コアがそれぞれ個別の命令を個別のデータに適用するような並列化方式である．MIMD による並列化は，CPU にコアが複数ある環境において，複数のコアを効率的に利用することが可能になる．

この MIMD による並列化は，様々な方法で実装することができる．共有メモリ型の並列計算機であれば，オープンソースの OpenMP や Intel が提供している TBB (Threading Buliding Block)[36] などのライブラリにより簡単に実現できる．例として，共有メモリ環境における OpenMP[32] では，並列化された区間ではスレッドが並列して各コアに割り当てられて個別のデー

タに対して個別の処理を同時に行えるようになっている．

近年は，1 つのチップが複数のコアを持つことがとても増えているため，このような MIMD 形式の並列化というものはさまざまなソフトウェアにとって必要不可欠となってきている．

1.2.4 BLAS ライブラリの問題点

OpenBLAS

BLAS ライブラリの一例としてオープンソースで開発が進められている OpenBLAS[7] の実装に関して簡潔に説明する．

OpenBLAS とは GotoBLAS[24] といわれる BLAS フレームワークから派生したオープンソースの BLAS ライブラリである．OpenBLAS はキャッシュブロッキングなどについて工夫されており，GotoBLAS の構造を踏襲している．

OpenBLAS における，キャッシュブロッキングをするための行列の細分化は大きく 6 つの分け方がなされている．この 6 つの分け方のうち一つの分け方について詳しく説明する．その細分化の方法を図 1.4 に示す．

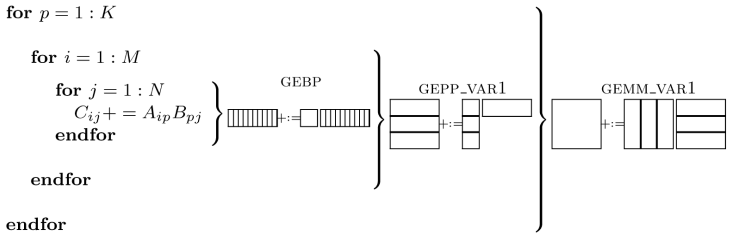


図 1.4: OpenBLAS の行列積実装 [24]

この細分化のうち一番内側で実際に演算が行われている部分をインナーカーネルとよぶ．そしてこのインナーカーネルの基本的な実装を図 1.5 に示す．

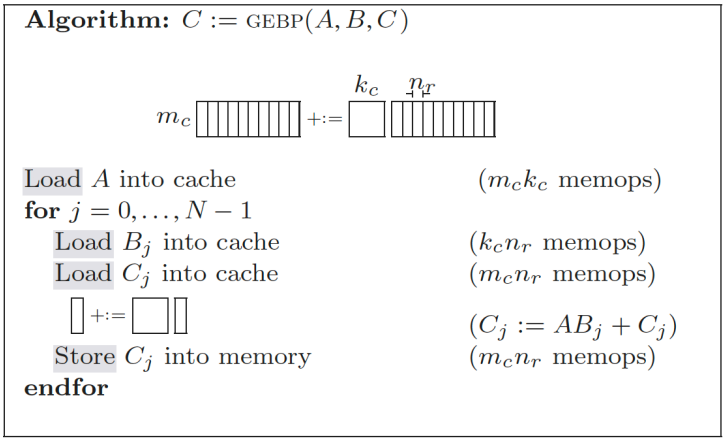


図 1.5: インナーカーネルの実装 [24]

この実装ではインナーカーネルを実行する際に小行列をキャッシュにロードしてくるようなフレームワークになっている．そのため，小行列の次元である m_c や k_c については各々の小行列がキャッシュサイズにフィットするように小ささを決める必要がある．特に，行列 A が L2 キャッシュに乗り，行列の一部である B_j と C_j が L1 キャッシュに乗るように小行列のサイズを決めるなどの工夫がされる．経験的には A が L2 キャッシュの半分ぐらいのサイズになるように調整するとより高速に動作することが知られている．

OpenBLAS ではこのような行列の細分化とループアンローリングの調整 [23][15] などはいくつかのアーキテクチャに向けて，アセンブリ言語などを用いて専門家によって高速化がされている．また，OpenBLAS では複数のコアを有効活用するために for loop などの繰り返しの処理を各コアに分割するといった並列化が OpenMP を用いて行われている．この時，OpenBLAS などの実装ではコアを全て利用可能であるといったもとで並列化及び最適化がなされている．

現在では，この OpenBLAS の構造をさらにもう一段階深くしたオープンソースの BLAS フレームワークである BLIS[52] なども登場している．

BLAS ライブラリの性能可搬性

現在，BLAS ライブラリやフレームワークは様々な形で提供されている．BLAS ライブラリやフレームワークは GPU 向けにも提供されているが，CPU 向けには特に多く提供されている．補足として，CPU とは異なるアクセラレータである GPU 向けのものとしては NVIDA が提供している CUDA を利用している cuBLAS[3] や OpenCL を利用しているオープンソースの clBLAS[2], clBALST[31] などが挙げられる．CPU 向けの BLAS ライブラリとしては，オープンソースで開発や公開がされている ATLAS[47][27],OpenBLAS[7][24],BLIS[52] や，ベンダーが自社のアーキテクチャに最適化するような形で開発して提供している AMD の ACML[1]，IBM の ESSL[4]，Intel の MKL[5] などが挙げられる．

このような幅広い形で提供されている BLAS ライブラリだが，多くの BLAS ライブラリはベンダー提供のものをはじめとして特定のアーキテクチャにしか最適されていない．そのため，新たな CPU のアーキテクチャが設計された場合や他の CPU でも利用したい場合は，新たにある BLAS ライブラリをそのアーキテクチャ向けに最適化し直すか，そのアーキテクチャに対応している他の BLAS を利用しなければならないのである．実際に日本のスーパーコンピューター富嶽が開発された当時，intel が開発したディープラーニング向けライブラリであり,BLAS ライブラリである MKL を内包する oneDNN を富嶽のアーキテクチャ向け移植するという高コストな作業が実際に行われた [8].

ここで，性能可搬性 (performance-portability) を実現するためのコストというものが現在の BLAS ライブラリにおける問題点の一つとして挙げることができる．

BLAS ライブラリの合成可能性

一般的な BLAS ライブラリはメニーコアの CPU を効率的に利用するため，行列やベクトル間の演算が分割されて複数スレッドで並列して実行されるように実装されている．この実装において，

BLAS ライブラリは自身の演算処理を行う間は計算リソースを専有することができると言う仮定で最適化がなされている．例として CPU のコア数が 16 であるような計算環境であれば，BLAS ライブラリは 16 個のスレッドを作り，求める演算を各スレッドに分けで実行できるように分割する．ここで，このような並列的な実装がなされている BLAS ライブラリがより上層のアプリケーションによって並列して呼び出されている場合を考える．その状態の概略を図 1.6 に示す．

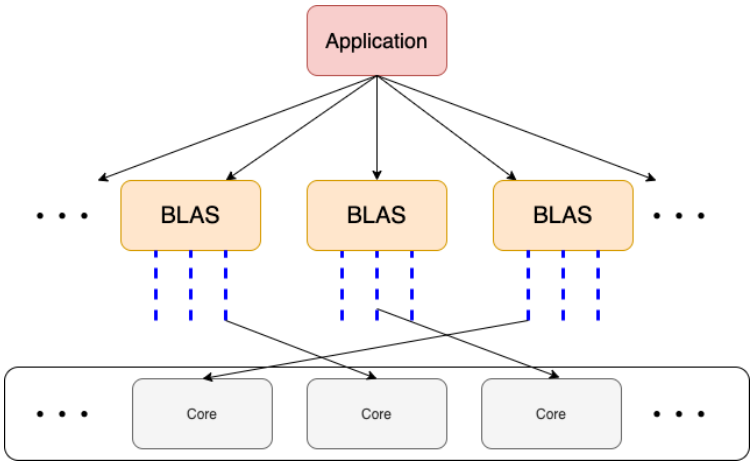


図 1.6: BLAS ライブラリの並列呼び出し

BLAS ライブラリ自体がより上位のアプリケーションなどによって並列化されている状況では，並列性がネストした状態になってしまい，BLAS ライブラリの性能を効率的に発揮することができないということが知られている．この性能低下の主な原因は，通常の OS と BLAS ライブラリだけでは計算リソースのマネジメントが貧弱であることである．具体的にはコア数以上の並列性が生まれてしまい，busy-wait な状態が発生していることと，それぞれの BLAS が全ての計算リソースを専有して実行することができるという仮定の元の最適化がなされていることなどが原因として考えられる．ここで，並列化されている BLAS ライブラリ自体を更に並列化すると言う合成可能性 (composability) も性能可搬性と並んで BLAS の問題点の一つとして挙げられる．

具体的には，三重対角行列の固有値ソルバなどの実装においてこの性能低下が見られる [51]．この三重対角行列の固有値ソルバは実対称密行列やエルミート密行列における固有値問題を解く際に用いられるソルバの中の主要なサブソルバの一つである．

三重対角行列の固有値ソルバの実装は分割統治法を用いて再帰的に行われており，再帰的に分割された小さい問題解同士をマージする際に行列積演算が複数回実行される．このマージ処理自体も並列に行われているため，行列積演算が並列に実行される状態が作られてしまう．その結果として並列性がネストした状態が作られてしまい，BLAS ライブラリの性能が効率的に発揮されなくなってしまう．

この合成可能性に関するものとしては，BLAS ライブラリ単体だけではなく並列化ライブラリ全体の問題として研究されている．この問題に関しては並列化ライブラリを対象としているため，一般的にはライブラリ自体の内部実装に関しては手を加えず，ライブラリ外の処理であるスケ

ジューリングなどに焦点を当てた研究がなされることが多い．代表的なものとして，Lithe[33] や Poli-C[9] などが挙げられる．Lithe は Pan らによって提案，開発された並列ライブラリとハードウェアの間に入るユーザーレベルスレッドを用いたリソースマネジメント用のソフトウェアである．この Lithe を挟むことによって並列化ライブラリの API 等を変更する必要なく，効率的に並列化ライブラリを合成して利用することができるようになる．そして Lithe はスレッドに関するスケジューラーやコンテキストといったものを処理する．また，並列処理でよく用いられている API である OpenMP や TBB を Lithe と一緒に使用できるように書き換えもされている．Poli-C は Anderson によって提案された，リソースのマネジメントを C 言語の言語拡張として実装されたものである．この拡張によってプログラマは並列化ライブラリを合成するだけでなく，OS によって提供される多く計算リソースを個々のポリシーの元に配分やマネジメントができるようになっている．

1.2.5 再帰的なブロッキングと並列化

本研究はこの BLAS ライブラリの問題点に対するアプローチとして，行列積演算に関して再帰的なブロッキングに基づく実装を行なった．

本研究の実装では再帰的なブロッキングを行うにあたって，分割統治法による再帰的な処理が多く用いられている．この分割統治法による実装によって，性能可搬性やタスク並列による合成可能性の向上が期待できる．

分割統治法

本研究では行列積の再帰的なブロッキング手法として分割統治法を用いた．分割統治法とは大きな問題を小さな問題に分割していき，小さい問題の個々の解を求めたのちに元々の大きな問題の解を求める方法である．プログラムにおいては，再帰的な処理によって簡単に実装することができる．この分割統治法を用いることで大きな問題を効率的に解くことが可能になる場合がある．

本研究が分割統治法を利用した理由として 2 つの理由がある．

- 小さい問題に分割した際にデータを参照の局所性が高い．
- タスク並列により並列化することが容易である．

一つ目の理由に関しては，行列積に分割統治法を用いることで，キャッシュとメモリの階層構造を効率的に活用することができ，cache obivious なアルゴリズムにすることができるという利点が挙げられる．また二つ目の理由に関しては，容易にタスク並列をすることができるようになり，メニーコア環境を活用できる上，コア数以上の並列性が生まれたり，CPU の一部のコアしか使用できない場合でも問題なく性能が得られるという利点が挙げられる．

アルゴリズム 5 Illustration of divide and conquer algorithm ($slove(P)$)

procedure SOLVE(D)
  if $size\ of\ P_k < Threshold\_size$ **then**
    $simply_solve(P_k)$
  else
    $P_1, \ldots, P_n = divide(P)$
    $A_1 = solve(P_1)$
    $\vdots$
    $A_n = solve(P_n)$
    $returncombine(A_1 \ldots A_n)$
  end if
end procedure

Cache-oblivious

cache-obliviou とは Frigo ら [21][22] によって提案されたもので，プログラム実行の際にアーキテクチャのキャッシュ構造やキャッシュサイズに対して情報をパラメータとして用いないという考え方である．cache-oblivious の対義語は cache-aware であり，cache-aware とは，CPU アーキテクチャのキャッシュサイズやキャッシュ構造などの情報を元にして，各階層のキャッシュサイズにぴったりと収まるようなデータの切り分け方や配置になるようにプログラムを設計する考え方である．

再帰的な分割統治をするアルゴリズムによって実装されることが多く，cache-oblivious アルゴリズムはキャッシュの構造を想定しながらも，具体的なサイズや遅延時間は仮定しないアルゴリズムである．分割統治をすることにより，大きな問題を再帰的に小さな問題に分割していき，その分割が進むにつれていずれは L1,L2,L3 などの階層のキャッシュに入れることができるサイズの問題に細分化される．その結果，キャッシュの構造やキャッシュサイズの情報を持っていなくても，キャッシュ内のデータで問題が解けるようになり，高い性能を得ることができる．

その概略を図 1.7 に示す．

この cache-oblivious なアルゴリズムを用いることによって，アーキテクチャのキャッシュ情報を必要としないまま効率の良いプログラムを実行することが可能になる．したがって，この cache-oblivious アルゴリズムは単体でアーキテクチャに依存せず，性能可搬性をある程度担保することができるようになると考えられる．

今後，巨大なクラスタ環境などでは，各マシンのメモリや CPU が揃っていないヘテロな環境になることが考えられ，今後 cache-aware なアルゴリズムを作ることへのコストが高くなり，cache-oblivious なアルゴリズムが有用になっていくと考えられる．

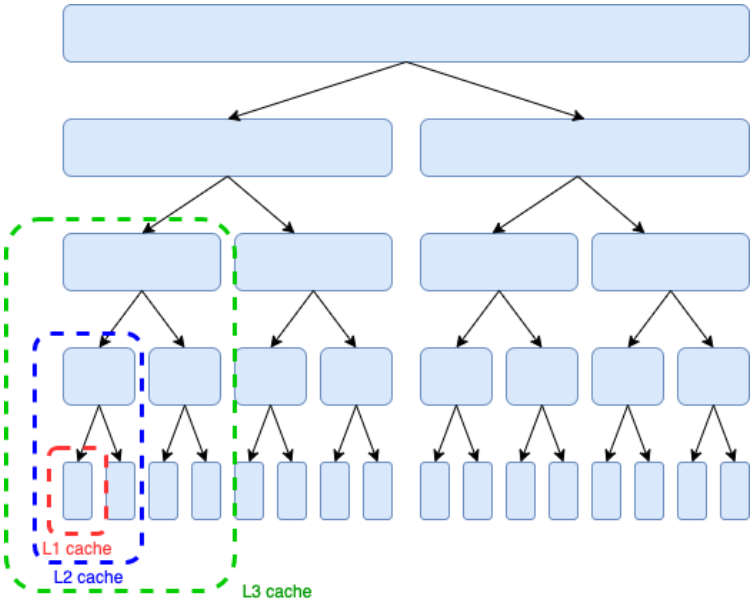


図 1.7: cache-oblivious の概要

タスク並列

タスク並列とはデータ並列と対になる言葉である．プログラムは一般的に複数のデータに対して複数の処理 (タスク) を行うが，そのプログラムを処理ごとにもデータごとにも分割することが可能である．

タスク並列とは，処理を分割し，分割された処理間に依存関係がない場合のものを集めて並列に実行するアルゴリズムを表す．また，データ並列とは処理対象の全体データがより小さいデータに分割可能であり，小さいデータの処理を行うことで，全体データへの処理も実質的に行われる場合に小さいデータへの処理を並列に実行するアルゴリズムを表す．

高速なタスク並列化を実現するためには，分割された処理の数が環境がもつ並列化可能な数を上回ることが重要であるが，分割統治法を用いたアルゴリズムは処理が再帰的に分割されていくため，タスク並列の適用が容易かつ多くの分割タスクを生成することができる．

1.2.6 タスク並列スケジューラ

プログラムをタスク並列可能な形で記述することで，タスク並列をサポートしているライブラリ等を用いて比較的容易にプログラムを並列化することができるようになる．タスク並列モデルでは一般にコア数を超える数のタスクを生成し，並列に実行していく．この生成されるタスク間には依存性があることが多く，この依存性を考慮しながらタスクをコアに並列に割り当てるというスケジューリングを行うタスク並列スケジューラーというものが存在する．

現在用いらているタスク並列をサポートしているライブラリとして，OpenMP[32] や TBB[36]

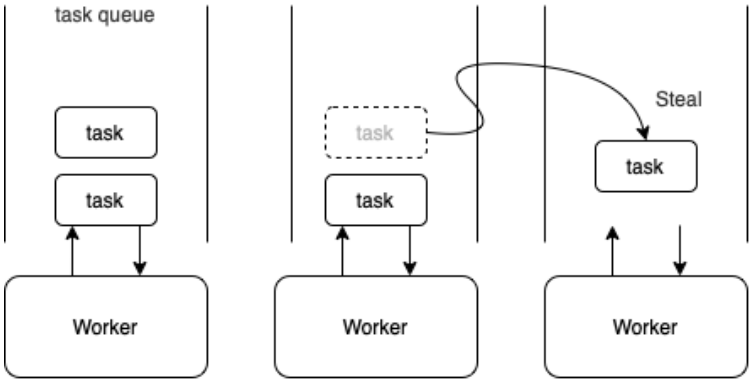


図 1.8: ワークスティーリングの概要

などが挙げられ，それぞれがタスクスケジューリングプログラムに基づいてタスクのスケジューリングを行なっている．これらのライブラリによって指定した上限のスレッド数までスレッドを生成が行われる．そしてそのスレッドに対して，最適にどんどんと生成されるタスクが割り当てられて各タスクがコアで実行される．この時，タスクの実行を各コアに割り当てるスケジューリングがタスク並列の効率を向上させる上でとても重要になる．

TBB では動的なワークスティーリング [13] が用いられており，動的なタスクの分配による負荷分散が行われている．このワークスティーリングとは図 1.8 のように，ある worker(CPU コア) が持っている自身のタスクキューにタスクがなくなった場合，別の worker のタスクキューからタスクを盗んできて行われる動的な負荷分散のことである．

このようなタスク並列における負荷分散も様々研究されており，近年のものであると MassiveThreads[30] 上に実装された ADWS[39] などが存在している．タスク並列が行える処理に関しては，タスク並列スケジューラの性能はとても重要なファクターである．そのため，一般的なタスク並列をサポートしてるライブラリだけではなく，この ADWS 等もタスク並列スケジューラとして利用した．

1.2.7 空間充填曲線を利用した行列積

2 次元平面での空間充填曲線とは，一般的に平面領域内の全ての点を通過するような曲線のことを指す．また，空間充填曲線は同じパターンが再起的に繰り返すされるようなフラクタル構造を持つ．

このような 2 次元の空間充填曲線には，2 次元配列などのものに対して 1 次元的なインデックスを付与することが可能になる．この空間充填曲線はインデックスや画像処理などの幅広い分野で利用されている．行列積演算に関する研究においても．空間充填曲線を満たす順序を利用することで性能向上を目指したものがある．

空間充填曲線の代表的なものとして，本研究でも利用したペアノ曲線とモートン曲線を紹介する．

ペアノ曲線

Bader ら [12][11] はペアノ曲線の順序をメモリレイアウトや要素間の計算順序の決定に用いた．ペアノ曲線の再帰的な構造を図 1.9 に示す．

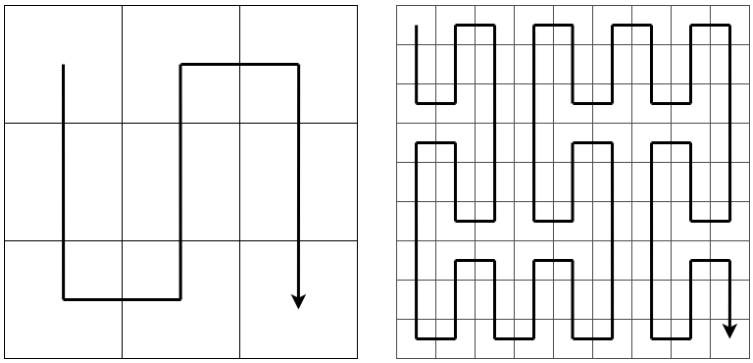


図 1.9: ペアノ曲線

ペアノ曲線は基本的には 3 の冪乗サイズの 2 次元平面に対して利用可能であり．3×3 の小さい平面に再帰的に分割して曲線が描かれる．行列積演算において，行列の各要素の積を求める順序を標準的なものからペアノ曲線の順序に沿ったものに変えることでデータの局所性が向上し，演算の性能向上が期待される．

Bader ら [10][12][11] が行なった．ペアノ曲線を用いた再帰的なブロッキングを行う行列積の性能を図 1.10 に示す．

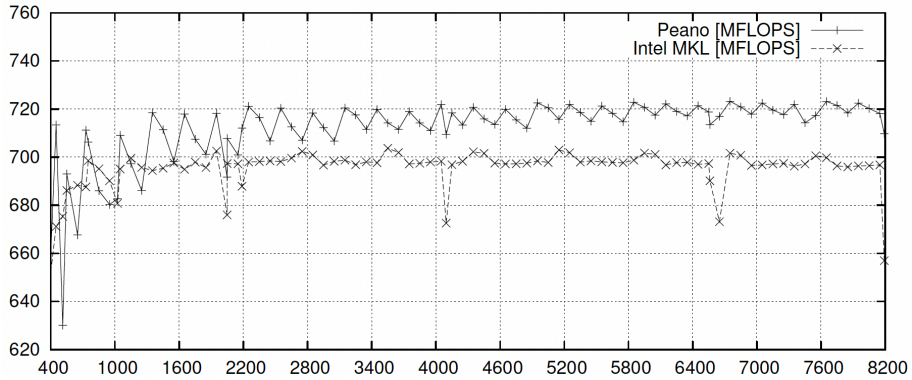


図 1.10: SIMD を用いない場合のペアノ曲線に基づく行列積と MKL7.2 の性能比較 [11]

この性能比較実験は 2005 年に行われており，SSE といった SIMD 命令は使用されていない．2005 年当時ではペアノ曲線を用いた行列積演算は Intel の MKL ライブラリよりも高速であることが報告されている．

しかし，Heinecke らが 2012 年に行なった実験 [25] ではペアノ曲線を用いた行列積演算は SSE や AVX といった SIMD 命令を利用した場合でも Intel の MKL ライブラリよりも遅いという結果が得られている．

モートン曲線

モートン曲線を利用した行列積はかなり古くから行われてきており，2002 年当時の古いプロセッサに対してモートン曲線を用いたストラッセンのアルゴリズムによる行列積なども存在する．[44] Perdacher ら [34] はモートン曲線の順序に沿って行列積や LU 分解の要素の積の計算を行うとデータの局所性が向上し，標準的な順序やその他の空間充填曲線の順序に基づいて計算される行列積よりも高い性能を得られること報告している．

また，Waker[45] はモートン曲線の順序に沿って行列のメモリレイアウトを変更し，再帰的なブロッキングを行うことで，標準的なメモリレイアウトと計算順序による行列積よりも高い性能を得られることを得られることを報告している．

モートン曲線の再帰的な構造を図 1.11 に示す．

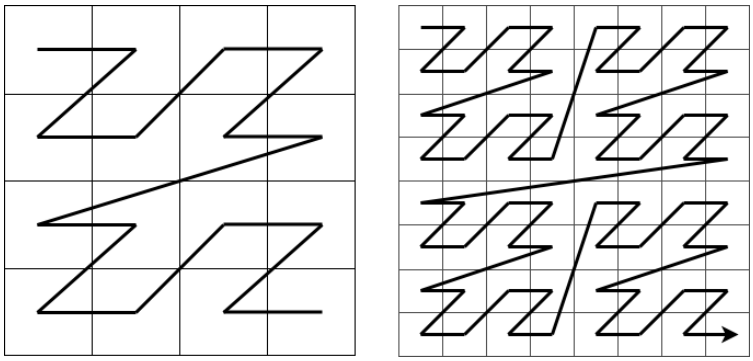


図 1.11: モートン曲線

モートン曲線は基本的には 2 の冪乗サイズの 2 次元平面に対して利用可能であるが，パディングや microcell[34] などにより任意のサイズに拡張可能である．

Perdacher ら [34] による行列積の高速化の結果を図 1.12 に示す．

この研究では，メモリレイアウトではなく計算順序つまり，行列積の解となる行列 C の各要素を求める順番をモートン曲線に基づく順序に変えたものである．実装では OpenMP[32] によるマルチコアでの並列化や AVX512 命令を用いた手動のチューニングによる SIMD 並列化が行われている．この研究ではモートン曲線の順序を用いる方が他の空間充填曲線の順序で要素を計算するよりも高速であることが報告されているが，他の BLAS ライブラリとの比較はされていない．論理性能と比較しても報告されている性能はかなり低く，BLAS ライブラリの論理性能比と比較すると提案された手法の性能は BLAS ライブラリに比べ著しく劣ってしまう．

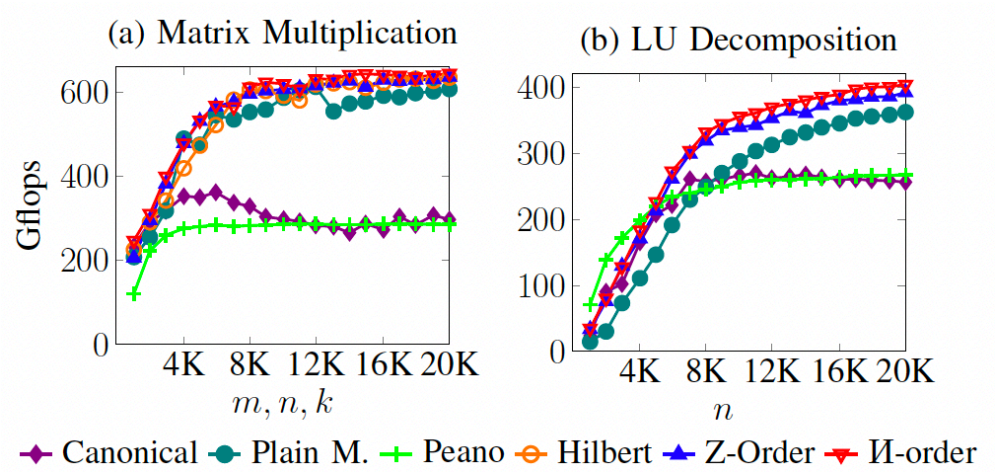


図 1.12: モートン曲線 (Z-order) を計算順序として用いた行列積と LU 分解の性能 [34]

次に Walker[45] によるモートン曲線に基づいたメモリレイアウトによる再帰的な行列積の性能を図 1.13 に示す．

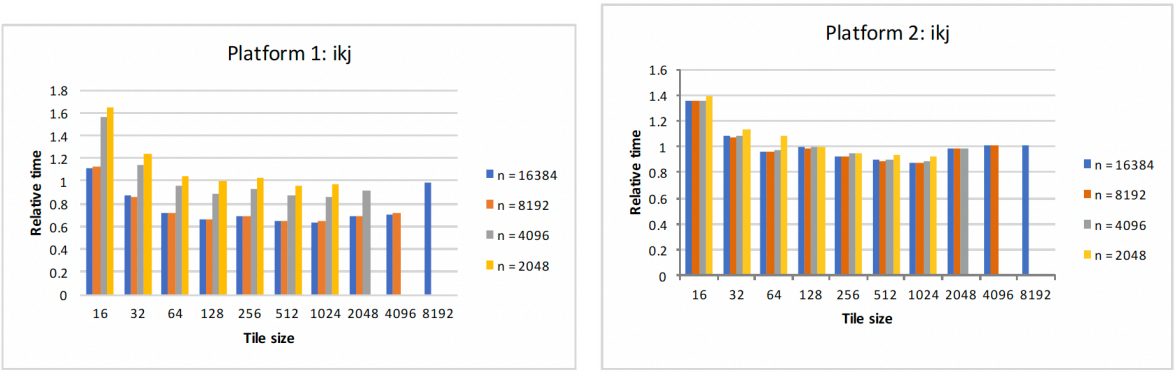


図 1.13: ループ交換を行なったカーネルを用いた比較 [45]

図 1.13 は行列のメモリレイアウトが Row-major(行に沿った順序) になってる場合の行列積演算を行なった実行時間を 1 とした際に，メモリレイアウトをモートン順序とした再帰的なブロッキングを行なった行列積演算の実行時間比を示している．どちらも Intel 社製の CPU であるが 2 つの実行環境において，モートン順序のメモリレイアウトを用いて再帰的にブロッキングしている手法はブロックサイズが小さすぎない場合には標準的な行列積よりも高速であることがわかる．この際，行列積のカーネルとしてはループ交換のみを行なった行列積カーネルを用いている．

1.2.8　空間充填曲線の生成

1.2.7 で紹介した空間充填曲線であるペアノ曲線とモートン曲線ついて具体的な曲線に基づく順序の生成方法について説明する．

ペアノ曲線

ペアノ曲線は再帰的に生成可能な空間充填曲線の 1 つであり，3 に平面が再帰的に分割されて曲線が生成される．

再帰的な曲線は 4 種類あり，P 型，Q 型，R 型，S 型と分類することができる．それぞれの型の曲線からさらに再帰的に全ての型を含んだ曲線が生成される．P 型, Q 型，R 型，S 型の再帰の様子を図 1.14 に概要を示す．

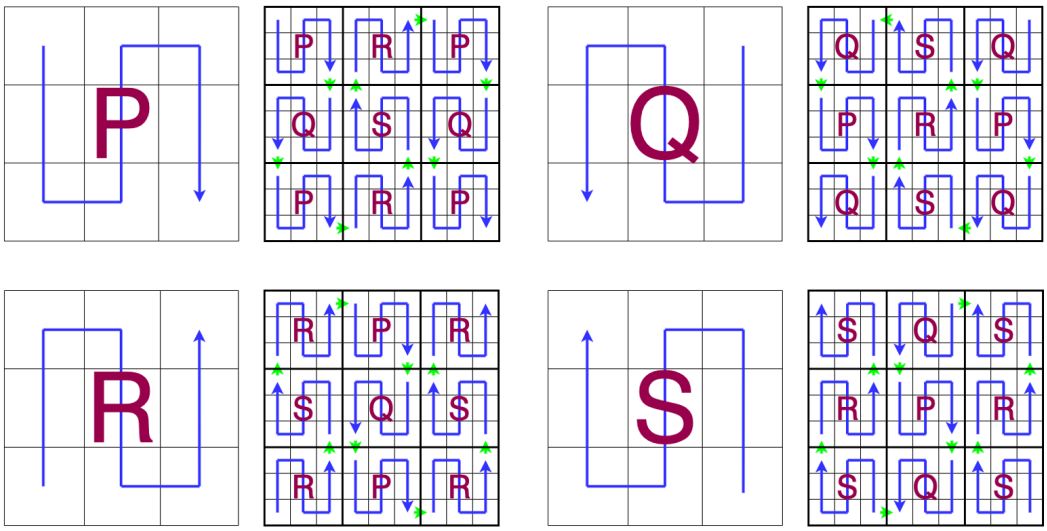


図 1.14: P 型，Q 型，R 型,S 型の外形

この P 型，Q 型，R 型，S 型の変換が再帰的に行われ，ペアノ曲線全体が生成される．ペアノ曲線の順序にメモリレイアウトを変更し, 行列を 3×3 に分割した際の各ブロックの順序は以下のように表せる．

$$\begin{bmatrix} A_0 & A_5 & A_6 \\ A_1 & A_4 & A_7 \\ A_2 & A_3 & A_8 \end{bmatrix} \begin{bmatrix} B_0 & B_5 & B_6 \\ B_1 & B_4 & B_7 \\ B_2 & B_3 & B_8 \end{bmatrix} = \begin{bmatrix} C_0 & C_5 & C_6 \\ C_1 & C_4 & C_7 \\ C_2 & C_3 & C_8 \end{bmatrix}$$

モートン曲線

モートン曲線も再帰的に生成可能な空間充填曲線の 1 つであり，2×2 に平面が再帰的に分割されて曲線が生成させる．

モートン曲線はその形状から Z 曲線とも呼ばれ，アルファベットの「Z」の文字の書き順のような順序で生成される．モートン曲線の順序は平面上の各要素の行と列の番号から計算される．

例として，2 次元平面上の点 $(i = 2, j = 1)$ を考える．この点 $(1, 2)$ のモートン曲線の順序における順番は以下の 3 つのステップによって求められる．

1. 2 次元平面の座標 i, j をそれぞれ 2 進数に変更する．
   この例の場合は, $i = 010_{(2)}$, $j = 001_{(2)}$ となる．
2. 2 進数表記の数である Z を用意し，偶数桁は i の 2 進数表記から，奇数桁は j の 2 進数表記を取り出す．
   この例の場合は，$Z = 001001_{(2)}$ となる．
3. 最後に生成された Z の値を 10 進数に戻し，この値がモートン曲線の順序における順番となる．
   この場合は，$Z = 9_{(10)}$ となる．

これらのステップにより，点 (2,1) はモートン曲線の順序で 9 番目となる点であることが求められる．

2 次元平面上の各点にこの処理を行い，モートン曲線の順序した概要を図 1.15 に示す．

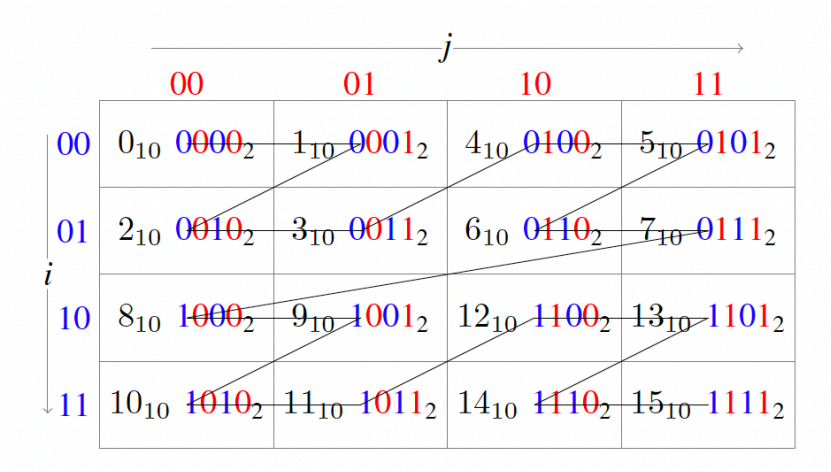


図 1.15: モートン曲線の生成方法 [34]

モートン曲線の順序にメモリレイアウトを変更し, 行列を 4×4 に分割した際の各ブロックの順序は以下のように表せる．

$$\begin{bmatrix} A_0 & A_1 & A_4 & A_5 \\ A_2 & A_3 & A_6 & A_7 \\ A_8 & A_9 & A_{12} & A_{13} \\ A_{10} & A_{11} & A_{14} & A_{15} \end{bmatrix} \begin{bmatrix} B_0 & B_1 & B_4 & B_5 \\ B_2 & B_3 & B_6 & B_7 \\ B_8 & B_9 & B_{12} & B_{13} \\ B_{10} & B_{11} & B_{14} & B_{15} \end{bmatrix} = \begin{bmatrix} C_0 & C_1 & C_4 & C_5 \\ C_2 & C_3 & C_6 & C_7 \\ C_8 & C_9 & C_{12} & C_{13} \\ C_{10} & C_{11} & C_{14} & C_{15} \end{bmatrix}$$

第 2 章

システム設計

本研究における実装は，C++ を用いた再帰的なブロッキングに基づく行列積アルゴリズムとその並列化が主である．具体的な実装内容は大きく分けて以下の 2 種類となる．

- 1 コア向け再帰的なブロッキングを用いた行列積実装
- 複数コア向けの並列化を行なった行列積実装

2 章では行列積の高速化に関する予備知識と実装したこれらの内容について詳細に説明する．なお，行列積は全て単精度の浮動小数点演算（float 型）かつ正方行列を仮定して行なった．

2.1　1 コア向けの再帰的なブロッキングを用いた行列積実装

まず初めに，1 コア向けに高速化を目的とした再帰的なブロッキングを用いた行列積の実装を行なった．この部分では，分割統治法を用いたナイーブな実装とメモリレイアウトを空間充填曲線に基づく順序に変更した実装の 2 種類を用意した．分割統治法によって再帰的に分割され，一番小さい行列積の問題が解かれる関数を `matmul_kernel` とした．この `matmul_kernel` による行列積は Intel MKL ライブラリを逐次で実行するもの利用した．行列積の逐次の 1 コア実行における SIMD の最適化や L1 キャッシュに対しての最適化に関しては，4.3.3 で紹介しているような研究により達成できると考えられるため，L1 キャッシュや SIMD 最適化は研究のスコープから除外し，可搬的であるが，最適化された理想的なカーネルとして MKL ライブラリの逐次実行を使用した．

2.1.1　ナイーブな行列積

分割統治法を用いたナイーブな行列積の実装を行なった．行列を 2×2 の小行列にそれぞれ分割し，小行列同士の行列積を再帰的に行うような実装である．解となる C の行列における各小行列の求める順序は後の並列化を考慮して，各小行列を Row-major で 2 回走査する順序をナイーブな順序とした．

ペアノ曲線に基づく行列積と比較するために，3×3 の小行列に分割した再帰的な実装も行なっ

たが，2×2 の分割の実装の擬似コードをアルゴリズム**??**に示す．また，この時に用いたメモリレイアウトは C++ において行列を確保する際の標準的な Row-major のメモリレイアウトであり，図 2.1 に示したレイアウトとなっている．

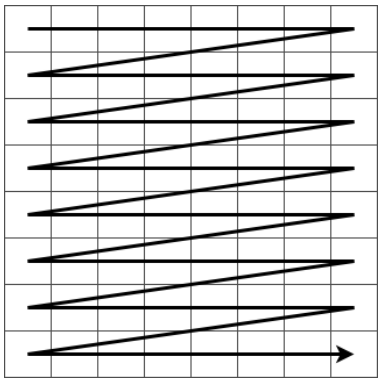


図 2.1: 標準的なメモリレイアウト

アルゴリズム 6 再帰的な行列積の実装

procedure MATMUL(A,B,C,n,ld)
  if $n < Threshold\_size$ **then**
    mamtul_kernel(A, B, C, n, ld)
  else
    matmul$(A_{00}, B_{00}, C_{00}, n/2, ld)$
    matmul$(A_{00}, B_{01}, C_{01}, n/2, ld)$
    matmul$(A_{10}, B_{00}, C_{10}, n/2, ld)$
    matmul$(A_{10}, B_{01}, C_{11}, n/2, ld)$
    matmul$(A_{01}, B_{10}, C_{00}, n/2, ld)$
    matmul$(A_{01}, B_{11}, C_{01}, n/2, ld)$
    matmul$(A_{11}, B_{10}, C_{10}, n/2, ld)$
    matmul$(A_{11}, B_{11}, C_{11}, n/2, ld)$
  end if
end procedure

2.1.2 メモリレイアウトを変更した行列積

上記で説明したナイーブな実装による行列積では，再帰的なブロッキングによって得られる小行列内で行列の各要素のデータがメモリ上で連続していない状況になってしまう．そこで，分割する小行列内で行列の要素のメモリ配置が連続するようにメモリレイアウトを変更した．本研究において，再帰的に分割された最も小さな行列は正方行列であり，行列のサイズが 2 の冪乗になるような

設定を行なった．

このメモリレイアウトの変更に関しては先行研究である Walker[45] の研究で紹介されていた手法をベースにメモリレイアウトの変更を行なった．

moton-hybrid と peano-hybrid

今回変更した，モートン曲線を利用したメモリレイアウトは morton-hybrid というメモリレイアウトである．morton-hybrid のメモリレイアウトは，行列のメモリレイアウト全てをモートン曲線の順序に変更するのではなく，2×2 で再帰的にモートン順序を決めていった際，ある一定の小さ以下の小行列に関しては標準的な Row-major な順序でのメモリレイアウトを行うというメモリレイアウト手法である．この morton-hybrid でのメモリレイアウトの概略を図 2.2 に概要を示す．

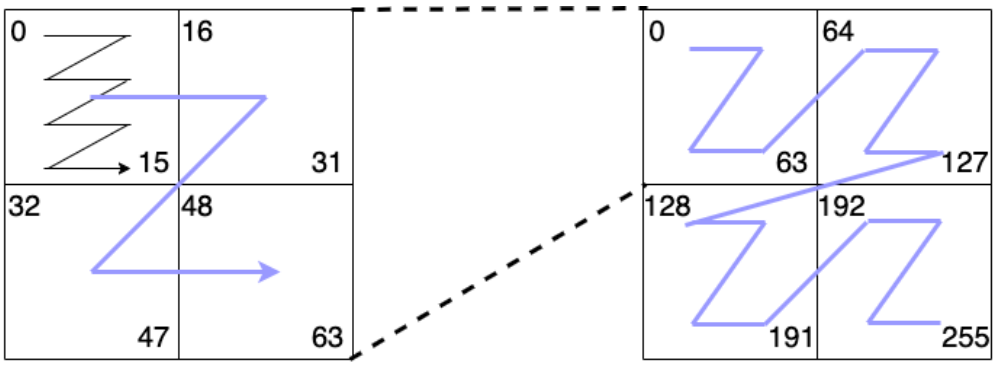


図 2.2: morton-hybrid 上の各要素の並び方

2005 年の Lorton ら [29] では morton-hybrid の形にメモリレイアウトを変更して分割した小行列ブロックの行列積に対して BLAS ライブラリを適用したものを適用した手法と，Row-major の行列に対して単純に BLAS ライブラリを適用した手法とのキャッシュ効率における比較などが報告されている．その研究においては，morton-hybrid を用いた手法と単純な BLAS ライブラリを適用した手法では実行する CPU のアーキテクチャ等の環境によってキャッシュ効率の良し悪しに反転が起きてしまっていた．しかし，2005 年前後と現在の BLAS ライブラリの実装は大きく異なっていることが想像されるため，現在に近い環境における明確な比較は存在していない．

また，ペアノ曲線に関しては morton-hybrid と同じように，小行列のブロック同士の間に注目した際はメモリ上に配置されている順序はペアノ曲線に従うが，一番小さいブロックの行列内におけるメモリレイアウトに注目した際は Row-major の順序で配置されているようなメモリレイアウトを行った．（以下，peano-hybrid）この peano-hybrid でのメモリレイアウトの概略を図 2.3 に概要を示す．

メモリレイアウトを変更する実装

morton-hybrid となるメモリレイアウトを変更するアルゴリズムの擬似コードをアルゴリズム 7 に概要を示す．

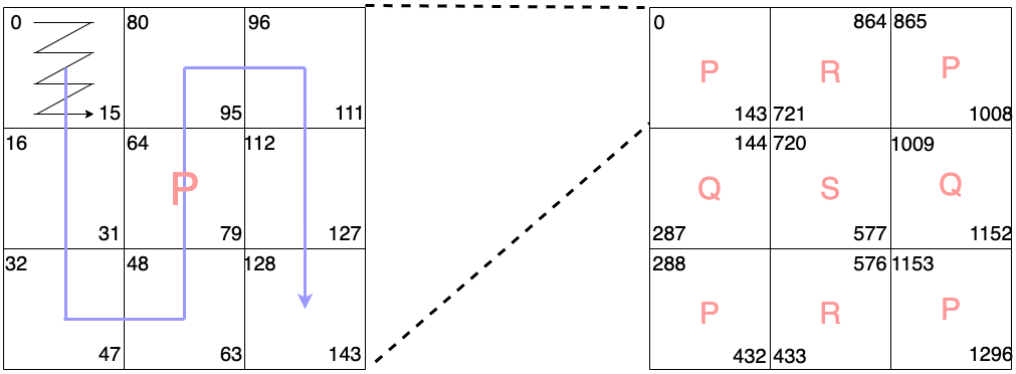


図 2.3: peano-hybrid 上の各要素の並び方

アルゴリズム **7** Memory arragement of morton-hybrid

---

procedure MORTON-HYBRID(A,n,r)
  if $r \leq 0$ **then return**
  else
    replace($A_{00}, n/2, n/2, n/2$)
    replace($A_{10}, n/2, n/2, n/2$)
    morton-hybrid($A_{00}, n/2, r-1$)
    morton-hybrid($A_{01}, n/2, r-1$)
    morton-hybrid($A_{10}, n/2, r-1$)
    morton-hybrid($A_{11}, n/2, r-1$)
  end if
end procedure

このアルゴリズムではメモリレイアウトを変更する際に，引数 r によって何段階のモートン順序を生成する再帰が行われるかを指定することができる．この引数を用いることで，任意の 2 の冪乗サイズの行列を任意の 2 の冪乗サイズの内部的には標準的なメモリレイアウトを持つ小さな行列ブロックに分割することが可能である．

この実装で用いた replace という操作は，Walker[45] の研究で概略が紹介されている unshuffle という操作に対して少々の変更を加え実装したものである．

replace という操作に関して説明する．

図 2.4 に示したように行列を行方向に 2 等分し，第 i 行の真ん中から左側の要素をまとめてベクトル a_i，右側においても同様なものをベクトル b_i とする．この時，C や C++ 言語等で行列をメモリ上に確保した場合は標準的な Row-major なメモリレイアウトとなり，メモリ上では $a_1b_1a_2b_2 \ldots a_nb_n$ という順番で連続することになる．この時，ベクトルのメモリ上に並んでいるベ

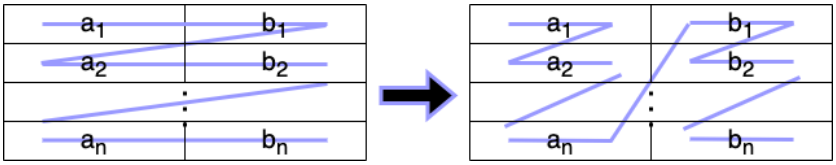


図 2.4: repalce 操作の概略

クトルの順番を以下のように変更する．

$$a_1b_1a_2b_2\ldots a_nb_n \rightarrow a_1a_2\ldots a_nb_1b_2\ldots b_n$$

この操作により，入力した行列の半分にした際の左側の小行列と右側の小行列のそれぞれの要素がメモリ上で Row-major の順序で連続し，さらには小行列同士もメモリ上で連続するようなレイアウトにすることが可能になる．この操作を再帰的に行うことで，より小さなメモリレイアウトが連続した小行列が生成可能である．本研究の C++ 言語での replace 操作の実装においては，メモリ上でのベクトル a_i や b_i の位置の交換には，`std::swap_ranges` 関数を用いて実装した．

この repalce 操作では行方向のみの 2 分割した小行列が生成されるので，morton-hybrid の順序を生成する際はアルゴリズム 7 のように，入力する行列を列方向に 2 等分してから上半分と下半分それぞれに対して repalce 操作を行なった．

また，peano-hybrid の順序を生成する際には，この replace の実装に対して少しの変更を行なった．入力された行列を行方向に 3 等分することを考え，第 i 行が連続した 3 つのベクトル a_i, b_i, c_i からなっているする．その際に，メモリ上に配置されている順番を以下のように変更する．

$$a_1b_1c_1a_2b_2c_2\ldots a_nb_nc_n \rightarrow a_1a_2\ldots a_nb_1b_2\ldots b_nc_1c_2\ldots c_n$$

これにより，入力された行列に対して行方向に 3 つに分割された小行列が生成でき，その各小行列では要素がメモリ上で連続した状態が作られる．

この repalce 操作では行方向のみの 3 分割した小行列が生成されるので，peano-hybrid の順序を生成する際は入力する行列を列方向に 3 等分してから上側，真ん中，下側のそれぞれに対して repalce 操作を行なった．しかし，これらの操作のみでは小行列内に注目した際は確かに各要素がメモリ上で連続した状況が作れているが，小行列間に注目した際は，小行列同士は標準的な順序で並んでいる状態となっており，ペアノ曲線に沿った順序を生成することができない．

そのため，peano-hybrid の順序を生成するためには小行列間の順序を変更する操作をさらに行う必要がある．小行列間の順序を標準的な順序からペアノ曲線の P 型，Q 型, R 型，S 型の順序に場合に応じて変更する必要がある．この操作は容易に実装することができる．replace 操作によって小行列内での要素はメモリ上で連続しているので，小行列間の位置の入れ替えの際に行列の最初と最後のメモリアドレスのみを気にすれば良く，小行列内の要素のメモリアドレスを機にする必要がなくなる．小行列のメモリ上での位置の入れ替えに関しても本研究では C++ 言語の `std::swap_ranges` 関数を用いて実装を行なった．

小行列に関して，標準的な順序から P 型，Q 型，R 型，S 型の順序への変換操作に関してはそれぞれ別の関数として実装を行い，再帰的に peano-hybrid を生成する関数内でそれぞれの型を指定するフラグを用意し，再帰的な生成の中で状況にあった型のレイアウトへの変更を行えるようにした．

行列積の実装

メモリレイアウトを変更した行列同士での行列積を実装するにあたり，再帰的に呼ばれる小行列同士の行列積の順序に関しても考慮する余地が存在する．morton-hybrid の順序のメモリレイアウトを利用する行列積に関してはナイーブな実装で用いられている順序におけるキャッシュ効率が良いため，同じ順序で小行列の行列積演算を実装した．つまり，疑似コード自体はアルゴリズム 6 と全く同じものとなる．peano-hybrid を利用した再帰的に行われる小行列の行列積演算順序に関してはナイーブな実装と同じ順序と，よりキャッシュを効率的に利用できると考えられる順序の 2 種類を用意した．

peano-hybrid を利用した行列積演算では一段階再帰が深くなるごとに小行列の行列積演算が 27 個生成される．その 27 個の行列積演算に関して，Bader ら [10][12][11] が提案した，より効率良くキャッシュを利用できるような順序に変更を加えた．

ペアノ曲線の順序に沿った再帰的なブロッキングでは，上記で述べたように P 型，Q 型，R 型，S 型の 4 種類の異なるメモリレイアウトを持つ小行列が生成される．そのため，小行列同士の行列積に関しては複数種類の演算が行われる．ペアノ曲線上での P 型，Q 型，R 型，S 型それぞれの存在する位置から以下の 8 種類の行列積が演算されうる．

$$PP,\ QP,\ PR,\ QR,\ RQ,\ SQ,\ RS,\ SS$$

また，これらの演算の解となる小行列の型も決まっている．

$$\begin{gathered} PP \to P,\ QP \to Q,\ PR \to R,\ QR \to S \\ RQ \to P,\ SQ \to Q,\ RS \to R,\ SS \to S \end{gathered}$$

このような 8 種類の演算に対して，各小行列のさらなる要素となる小行列のブロック間の積の計算において最適な計算順序がそれぞれ存在している．ここでは，$PP \to P$ の計算について例を挙げる．$PP \to P$ の計算は以下のように考えられる．この時，添字が増える順番にメモリ上で連続している．

$$\begin{bmatrix} A_0 & A_5 & A_6 \\ A_1 & A_4 & A_7 \\ A_2 & A_3 & A_8 \end{bmatrix} \begin{bmatrix} B_0 & B_5 & B_6 \\ B_1 & B_4 & B_7 \\ B_2 & B_3 & B_8 \end{bmatrix} = \begin{bmatrix} C_0 & C_5 & C_6 \\ C_1 & C_4 & C_7 \\ C_2 & C_3 & C_8 \end{bmatrix}$$

この場合は，図 2.5 に示した順序の計算がキャッシュの利用効率が高い．図 2.5 内の横に並んでいる 3 つの数字において，左の数字が行列 A において添字が一致する要素を表し，真ん中の数字が行列 B において添字が一致する要素，右の数字が数字が行列 C において添字が一致する要素を表す．

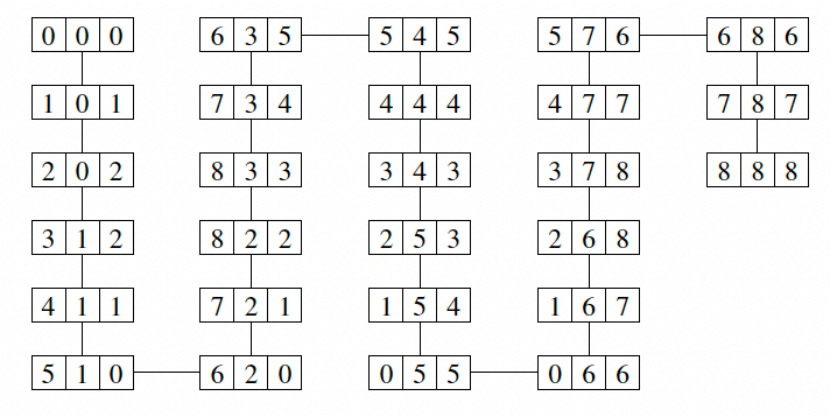


図 2.5: P 型と P 型の行列積の場合の計算順序 [11]

この順序では，次の要素同士の演算に移る際に，たった今演算に使用された 3 つの要素の内 1 つは必ず次の演算でも使用される．また残りの 2 つの直前の演算で使用された要素に関しても必ずその要素とメモリ上で連続している要素が次の演算で用いられるようになっている．これによりキャッシュの利用効率が高く，高い性能が期待される．

本研究では，上記の 8 種類全ての行列積に関してこのような順序をそれぞれ用いて実装した．

2.2 複数コア向けの MIMD 並列化を行なった行列積

2.1 では，シングルコア向けの再帰的なブロッキングを用いた高速化を目的とした設計・実装を行なった．しかし，現在主流となっている計算機の環境はマルチコアやメニーコアと呼ばれる，複数のコアを持った CPU がほとんどの割合を占めている．そのため，1 コアに特化した高速化を施してもマルチコア向けに並列化できなかったり，マルチコア向けの並列化の性能が悪かったりした場合は意味がない，このような現在の環境を考えると，マルチコアで高速に動く行列積が必要不可欠であり，再帰的なブロッキングを用いた行列積を並列化する必要がある．

2.2.1 タスク並列

一般的な行列積演算の並列化は，各行列のインデックスを走査していく for ループが多く登場し，その for ループに対して並列化が行われることが多い．

```
1 #pragma omp parallel for private (j,k)
2     for(i = 0; i < n; i++){
3         for(j = 0; j < n; j++){
4             for(k = 0; k < n; k++){
5                 c[i][j] += a[i][k] * b[k][j];
6             }
7         }
8     }
```

ソースコード 2.1: 一般的な行列積の OpenMP 並列化

ソースコード 2.1 は OpenMP[32] を用いて for ループを並列化した例である．この並列化は for ループで繰り返される処理がスレッドに分割されて処理されるようになる．このような並列化は，各演算に必要なデータが複数のスレッドに分割されて並列化されるので，データ並列の一種だと言うことができる．

それに対して，本研究で設計した行列積は分割統治法によって再帰的にブロッキングを行なって演算するプログラムであるため，このような for ループの並列化をを行うことが難しい．

本研究で用いているような再帰的に演算が増えていくプログラムに関しては，次々と生まれてくる演算を子タスクとして，タスク並列化を行うことが比較的用意である．ただし，タスク並列化を行う際は各タスク間の依存性を考慮して，並列化を行う必要がある．現在，タスク並列は OpenMP や TBB などの並列化ライブラリでサポートされている．

このようなタスク並列の形で並列化を行うことで，コア数以上に並列性が生じた場合や一部のコアでしか使用ができない場合でも性能が保てると考えられる．

2.2.2 タスク並列化の実装

morton-hybrid

まず，初めに 2.1 で設計と実装を行なった再帰的なプログラムに関して，再帰的に演算される関数に対してタスク化を行なった．

```
1  void matmul(float *a, float *b, float *c, int di, int dj, int dk, int ld) {
2    if (CUTOFF_THRESHOLD_MATMUL && (di <= CUTOFF_THRESHOLD_MATMUL || dj <=
         CUTOFF_THRESHOLD_MATMUL || dk <= CUTOFF_THRESHOLD_MATMUL)) {
3      matmul_kernel(a, b, c, di, dj, dk, ld);
4    } else {
5      #pragma omp taskgroup
6      {
7        #pragma omp task
8        matmul(a, b, c, di/2, dj/2, dk/2, ld/2);
9        #pragma omp task
10       matmul(a + 2 * (ld/2) * (ld/2), b, c + 2 * (ld/2) * (ld/2), di/2, dj/2, dk
             /2, ld/2);
11       #pragma omp task
12       matmul(a, b + (ld/2) * (ld/2), c + (ld/2) * (ld/2), di/2, dj/2, dk/2, ld/2)
             ;
13       #pragma omp task
14       matmul(a + 2 * (ld/2) * (ld/2), b + (ld/2) * (ld/2), c + 3 * (ld/2) * (ld
             /2), di/2, dj/2, dk/2, ld/2);
15     }
16     #pragma omp taskgroup
```

```
17      {
18        #pragma omp task
19        matmul(a + (ld/2) * (ld/2), b + 2 * (ld/2) * (ld/2), c, di/2, dj/2, dk/2,
              ld/2);
20        #pragma omp task
21        matmul(a + 3 * (ld/2) * (ld/2), b + 2 * (ld/2) * (ld/2), c + 2 * (ld/2) * (
              ld/2), di/2, dj/2, dk/2, ld/2);
22        #pragma omp task
23        matmul(a + (ld/2) * (ld/2), b + 3 * (ld/2) * (ld/2), c + (ld/2) * (ld/2),
              di/2, dj/2, dk/2, ld/2);
24        #pragma omp task
25        matmul(a + 3 * (ld/2) * (ld/2), b + 3 * (ld/2) * (ld/2), c+ 3 * (ld/2) * (
              ld/2), di/2, dj/2, dk/2, ld/2);
26      }
```

ソースコード 2.2: 再帰的な行列積の OpenMP task 化

ソースコード 2.2 は morton-hybrid なメモリレイアウトをした行列を再帰的に 2×2 に分割していった際に OpenMP[32] を用いてタスク並列化を行なった行列積演算を行う実際の関数である．これはアルゴリズム 6 内に出てくる 8 個の行列積演算を小タスクとして扱ったものである．この 8 個の小タスクのうち，上側 4 個と下側 4 個の間には依存関係があり，上側の 4 個の演算が終了しないと下側の 4 個の演算を始めることができない．そのため，上側 4 個と下側 4 個のタスクをそれぞれタスクグループとしてまとめる実装を行なった．同様にタスク間の依存性を考慮して再帰的に 3×3 に分割していくものも設計し実装を行なった．

この並列化はメモリレイアウトがナイーブなものと空間充填曲線に基づいたもののそれぞれについて行なった．空間充填曲線に基づいたメモリレイアウトの場合，各タスク内で行われる再帰的な行列積に必要なデータが全て連続するような設計となっているため，データの局所性が Row-major なメモリレイアウトを用いたものよりも高いことが想定される．タスク並列の実装に用いたライブラリとしては，OpenMP[32]，TBB[36]，MassiveThreads[30] の 3 種類を用いた．

peano-hybrid

どんなメモリレイアウトを用いていたとしても，2.2.2 で設計した並列化においての最小のタスクは `mamtul_kernel` といった SIMD 命令を用いて一つの小さい行列積を計算するタスクとなっている．そのため，2.1 で設計した逐次実行で peano-hybrid のメモリレイアウトを用いた際のメリットである行列積演算の順序に関しては，全く考慮されていない設計となってしまっている．

peano-hybrid のメモリレイアウトのメリットを生かすために，タスク化された再帰呼び出しが終了する小行列サイズの閾値をもう一段上げ，最小タスク内でメモリレイアウトを活用できる演算順序での行列積演算の再帰をもう一度行うような実装も用意した．これにより，peano-hybrid のメモリレイアウトを使用した際には，逐次でのデータの局所性が高まり，さらなる高速化が見込まれる．

第 3 章

評価

本章では 2 章で説明した，1 コア向けの再帰的なブロッキングを用いた行列積とそれを複数コア向けに並列化を行なった行列積の性能に関する実験結果を紹介する．

3.1 実験環境

表 3.1: マシン詳細 (Oakbridge-CX node)

OS	CentOS Linux release 7.6.1810
CPU	Intel Xeon Plutinum 8280 (Cascade Lake) × 2
Memory	192 GB (DDR4)

本研究は主に東京大学情報基盤センターの Oakbridge-CX の 1 ノードを用いて実験を行なった．実験に用いた Oakbridge-CX の一つの計算ノードの環境を表 3.1 に示す．Oakbridge-CX の計算ノードは CPU として Intel Xeon Platinum 8280 を 2 基搭載している．

複数チップを 1 つのマシン上で用いるという環境は NUMA（Non-Uniform Memory Access）環境と呼ばれ，この場合は，Xeon Platinum とメモリの 1 ットからなる NUMA ノードを 2 つ組み合わせたものがマシンの環境となっている．複数の NUMA ノードは仮想化技術によって，OS からは基本的に 1 つのプロセッサとメモリからなるマシン構成と認識される．各 NUMA ノード間のデータ等の通信はノード内の通信よりも速度が遅くなってしまうため，並列化された処理においてはスレッドやデータ自体の NUMA ノード上への割り当てが性能に影響してくる場合がある，

そのため，本研究では複数のコアを用いる実験を行う際は，`numactl` コマンドを用いて，データの割り当てに関する制限を行なった．`numactl --interleave=all` というコマンドを用いることで, プログラムで使用するデータを均等に各 NUMA ノードのメモリに分割することができるようになる．

1 コアや特定のコア数のみを用いる実験を行う際は，`taskset` コマンドを用いて，使用するコア

数に関する制限を行なった．

実験で用いた Xeon Platinum の詳細な使用を表 3.2 に示す．

表 3.2: Xeon Platinum 8280

Cores (Threads)	28 (56)
Clock	2.7 GHz (up to 4.0 GHz)
L1 cache	32KB (L1D), 32KB (L1I) / core
L2 cache	1MB /core
L3 cache	38.5MB

次に使用したソフトウェアについて説明する．並列化のスレッドライブラリとして OpenMP は OpenMP 5.0，TBB は oneTBB 2021.5.0，MassiveThreads は 0.97 を用いた．また，コンパイラとして GCC を用いるとタスク並列かつ NUMA である環境に対して十分なサポートがなされていない [43] ため，インテルの C++ 向けコンパイラである icpc version 19.1.3.304 を用いた．最適化オプションは実験全体を通じて`-O3 -march=native`とした．BLAS ライブラリとして OpenBLAS は OpenBLAS 0.3.17，Intel MKL は MKL 2020.4.304 を使用した．

また本研究では 3.5 で可搬性を確かめるために，弊研究室のサーバーである Spica を用いて実験を行なった．そのマシン環境を表 3.3 に示す．この環境での実験ではコンパイラとして GCC コンパイラを用いた．

表 3.3: マシン詳細 (spica)

OS	Ubuntu 18.04.3 LTS
CPU	Xeon E7-8890 v4 × 4
Memory	2TB (DDR4)

3.2 1 コアでの行列積

本研究で実装した，1 コアでの再帰的なブロッキングを用いた行列積の実験について評価を行う．

この評価に関しては，行列積演算を行なっている間の演算性能（GFLOPS）の値を評価指標として用いた．

また，2 で説明した再帰的なブロッキングが終了する最小の小行列サイズになった場合に実行される`matmul_kernel`関数としては Intel MKL ライブラリを用いて評価を行なった．

3.2.1 閾値に関する評価

再帰的なブロッキング処理が終了する閾値となる，最小の小行列サイズを変化させることによる性能の違いを確認した．この評価においてはナイーブな実装と morton-hybrid を用いた実装の 2 種類で実験を行なった．

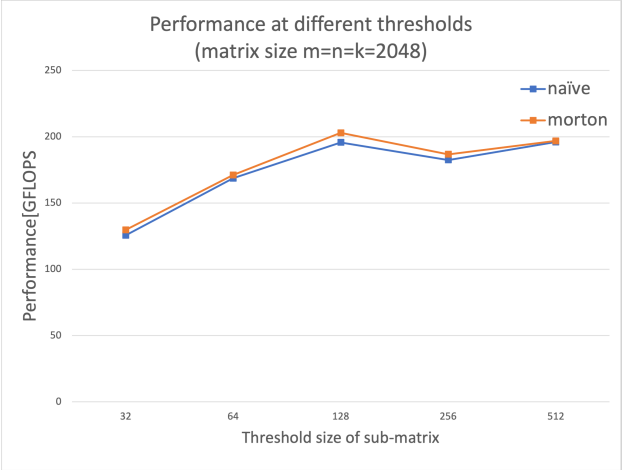


図 3.1: 最小行列サイズによる性能比較 ($m = n = k = 2048$)

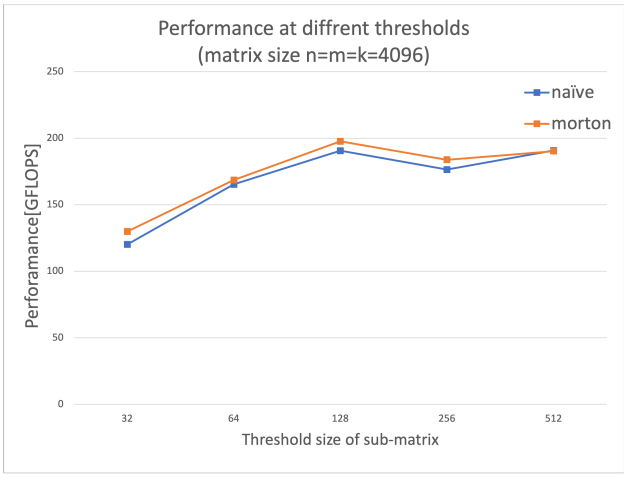


図 3.2: 最小行列サイズによる性能比較 ($m = n = k = 4096$)

図 3.1 に行列積を求める正方行列のサイズが 2048 の場合の閾値による性能比較を，図 3.2 に正方行列のサイズが 4096 の場合の閾値による性能比較の実験結果を示す．

この結果から，メモリレイアウトに関わらず再帰的なブロッキングの終了条件となる行列のサイズが 128 または 512 の場合が行列積演算において性能が出るということがわかる．

行列のサイズによるデータの大きさについて考えてみると，行列サイズが 128 の場合は行列積演算に用いるデータのサイズは $128 \times 128 \times 3 \times 4(float) = 196,608$ Byte となり 196.6[KB] である．これは環境の L2 キャッシュには載るデータサイズとなっている．また行列サイズが 512 の場合はデータサイズとして $512 \times 512 \times 3 \times 4(float) = 3145728$ Byte となり 3.14[MB] である．これは環境の L2 キャッシュに載らないデータサイズとなってしまっている．

一般的に再帰的なブロッキング等を行わず，行列積に BLAS ライブラリの GEMM を適用する場合は L1 キャッシュや L2 キャッシュのサイズや TLB に対してアセンブリなどを用いた高度な最適化が行われている．そのため，演算に必要なデータサイズが L2 キャッシュサイズを超える行列サイズ 512 の場合は，L2 サイズより上のメモリ階層に関しては高度に最適化がなされとても高い性能が出ていると考えられる．

この L2 以下に対して最適化されている状態以上に行列サイズが 128 の場合が高速に動作している．行列サイズが 128 の場合には L2 のキャッシュサイズのパラメータを用いていないにも関わらず，L2 キャッシュのサイズに対して最適化された実装を用いているよりも性能が高いということがわかる．

以降，評価においての閾値の行列サイズとして 128 を用いた．

3.2.2 BLAS ライブラリとの比較

次に，ナイーブな実装，morton-hybrid を用いた実装，peano-hybrid を用いた実装の 3 つの再帰的なブロッキングによる実装と既存の BLAS ライブラリである逐次の OpenBLAS と MKL との比較をさまざまな行列サイズの行列積演算で行なった．今回の評価において, morton-hybrid を用いた実装に関しては，再帰的なブロッキングが 2×2 の形で行われるため，対象となる行列積演算のサイズは 128×2^n とした．また peano-hybrid を用いた実装に関しても，再帰的なブロッキングが 3×3 の形に行われるため，対象となる行列積演算のサイズは 128×3^n とした．

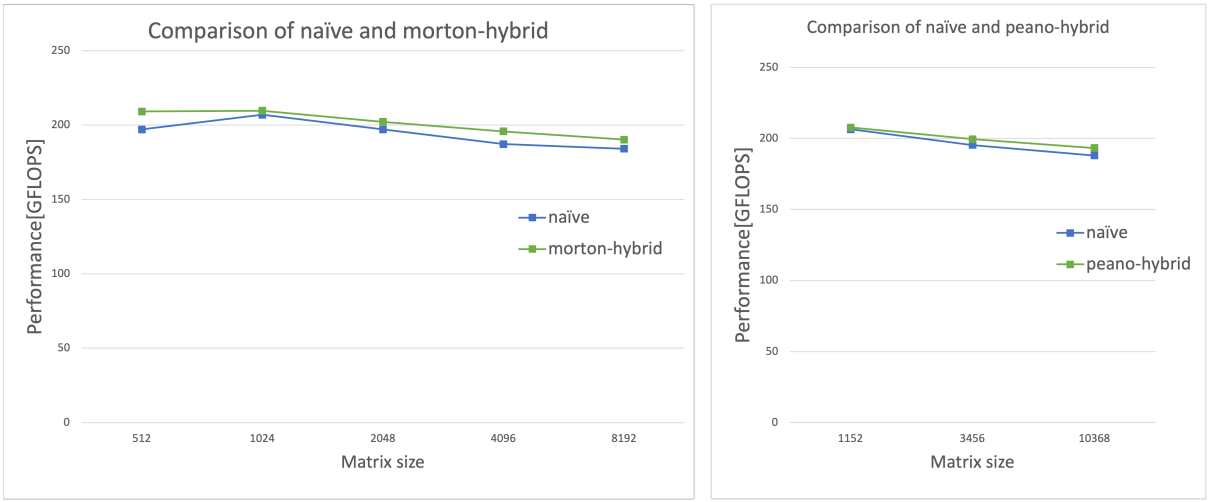


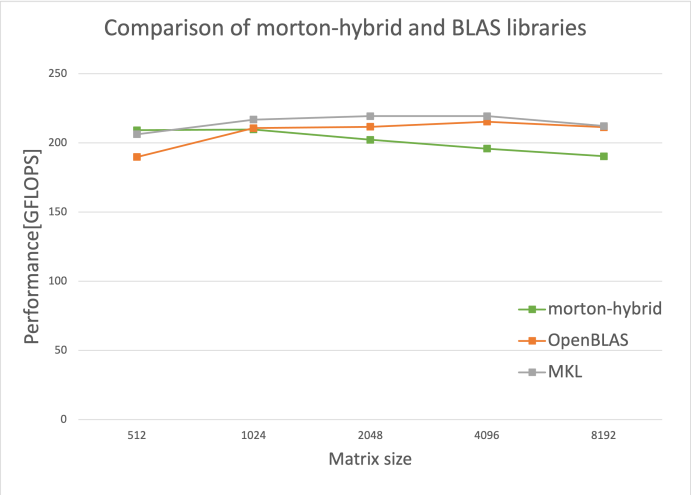
図 3.3: ナイーブな実装とメモリレイアウトを変えた実装の比較

図 3.3 にナイーブな実装とメモリレイアウトを変更した実装の性能について実験した結果を示す．

基本的には，行列のサイズが大きくなるにつれて性能が落ちる．これは再帰の回数が増えることに起因すると考えられる．

ナイーブな手法とメモリレイアウトを変えた手法では，行列積の行列のサイズが大きくなるにつれて性能の差が大きくなっている．これはデータの局所性によるものだと考えられる．ナイーブな手法では元々の行列サイズが大きくなるに従って，次の行にある要素までのメモリの配置が離れてしまう．それによって小行列をブロッキングによって生成した際にブロック内の要素間のメモリ上の距離がとても離れてしまうことにより，演算を行う際にキャッシュミスが多くなるからである．それに対して，メモリレイアウトを変えた morton-hybrid や peano-hybrid を用いた手法では，ブロック内の要素は常に連続したメモリ配置を取るようになっているため，行列のサイズが大きくなっても効率よくキャッシュを使うことができ，性能が低下しにくくなっている．

次に，このメモリレイアウトを変えた再帰的なブロッキング手法と既存の BLAS ライブラリにそのまま行列積を演算させたものを比較する．



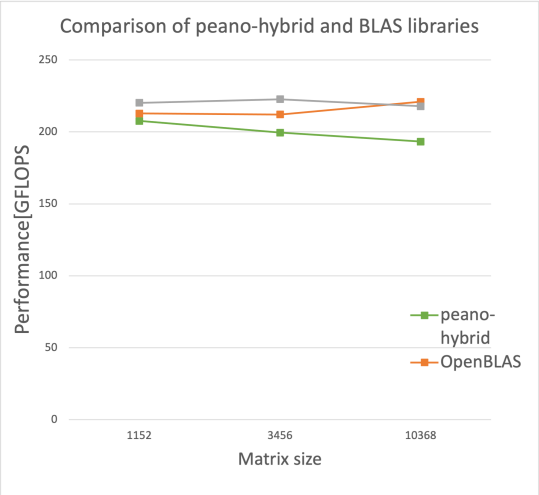


図 3.4: メモリレイアウトを変えた実装と既存の BLAS ライブラリとの比較

全体的には MKL ライブラリが一番性能が良く，ついで OpenBLAS，メモリレイアウトを変えた再帰的なブロッキングのものとなっている．既存の BLAS ライブラリは高度に最適化されており，行列サイズが大きい場合にも高い性能を発揮することができている．一方，再帰的なブロッキングに基づく手法では再帰呼び出しが多くなってしまい大きな行列では性能が落ちてしまう．

行列サイズが 512 など小さい場合には，行列が小さすぎるために BLAS ライブラリのキャッシュブロッキングなどが十分に機能していない可能性がある．それに対して，再帰的な手法では再帰回数が少ないため行列積以外のオーバーヘッドが少なく，性能を発揮できていると考えられる．

この評価では，メモリレイアウトを変更した再帰的な手法はナイーブな再帰的な実装より高速ではあるが，既存のライブラリの性能を超えることができていないということがわかった．

3.3　複数コア上での並列化された行列積

次に本研究では，メモリレイアウトを変更した再帰的なブロッキングを並列化し，複数のコア上で動かす実験及び評価を行なった．

morton-hybrid

2.2.2 で説明したような設計で，morton-hybrid を用いた再帰的なブロッキングの実装をシンプルなタスク並列の形で並列化を行なった，タスク並列化を行うにあたって OpenMP，TBB，MassiveThreads の 3 種類を用いた実装を用意した．また，評価の際に用いたスレッド数は環境のコア数と同じになるように 56 と設定した．図 3.5 に morton-hybrid に OpenMP task，Intel TBB，MassiveThreads 上に ADWS を実装したもの（myth+ADWS）の 3 種類の性能を示す．

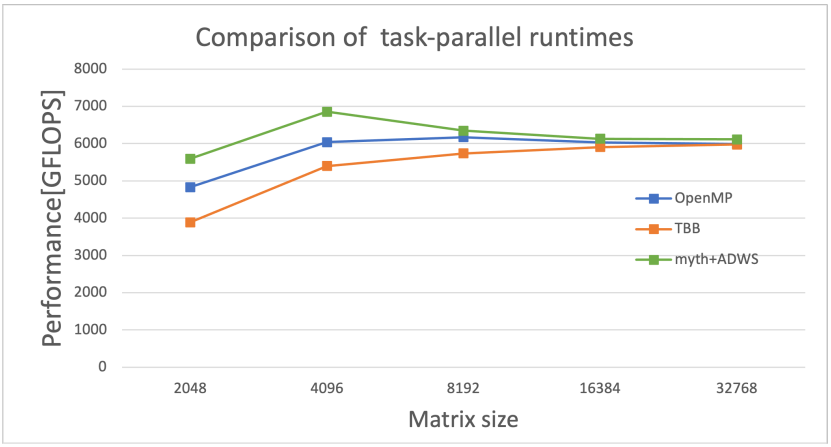


図 3.5: 各タスク並列ランタイムによる性能

morton-hybrid のメモリレイアウトで MassiveThreads と ADWS を利用したものがとても高い性能を示していおり，以下我々の手法として評価に用いていく．最大で行列サイズが 2048 の時には OpenMP による実装と比べ 16% も高く，TBB による実装と比べて 30% ほど高くなった．全ての行列サイズにおいて MassiveThreads と ADWS を利用したものが他のランタイムよりも性能が高かった．

そして，この morton-hybrid のメモリレイアウトに MassiveThreads と ADWS による実装と既存の BLAS ライブラリとの複数コアの環境での性能を比較した．既存の BLAS ライブラリとして，OpenBLAS は内部で OpenMP が使われており，Intel MKL には内部に OpenMP を使ったバージョンと TBB を使ったバージョンがある．本評価では Intel MKL についてどちらのバージョンでも性能を測定した．また，評価の際に用いたスレッド数は環境のコア数と同じになるように 56 と設定した．

図 3.6 に示されている通り，我々の手法では，2048 と 4096 の特定のサイズの行列積において既

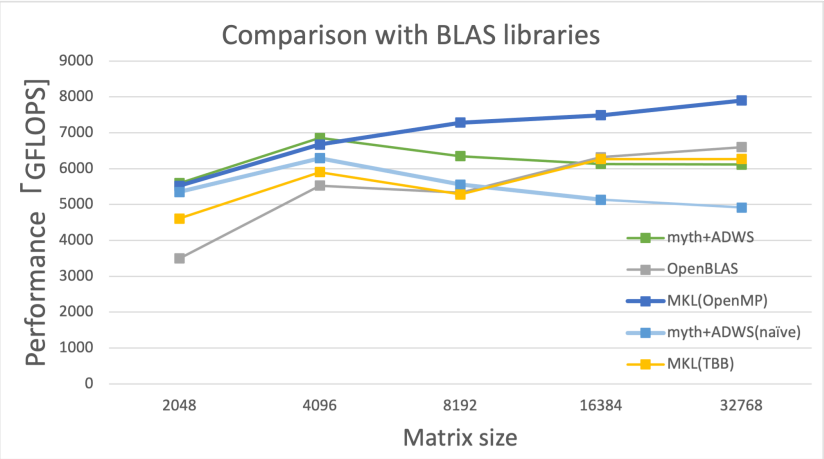


図 3.6: 既存の並列化された BLAS ライブラリとの性能比較

存の BLAS ライブラリの並列化よりも高い性能を得ることができた．行列サイズが 2048 の場合は，行列が小さいかつ環境内で使用可能なコアの数が多いため，既存の並列化された BLAS ライブラリでは並列度がコア数に比べて低くなってしまうため性能があまり出ていないと考えられる．また，タスクモデルによる並列化を行なった際は，行列サイズが 2048 と比較的小さい場合でも並列度がある程度は高くなり多くのコアを有効活用できるようになっていると考えられる．

morton-hybrid なメモリレイアウトと高性能なタスク並列スケジューラである ADWS を用いることで，並列性がネストしていない状態でも既存の並列化された BLAS ライブラリと近い性能を得ることができた．また，実行した全ての行列のサイズでは OpenBLAS に対しては高いまたは近い性能を得ることができた．

また，高性能なタスク並列スケジューラとして ADWS の後継である ML-ADWS[37] を用いて実装したものを図 3.7 に示す．

行列サイズが比較的小さい場合は ADWS の方が性能が良くなっている．これは ADWS は L3 キャッシュに対して効率の良い実装がなされており，L3 キャッシュサイズ付近のサイズの問題に関しては性能が出やすいことに起因していると考えられる．行列サイズが大きくなるに従い，ADWS による実装は性能が落ちるが，ML-ADWS は性能を維持している．そのため，行列サイズが 8192 以上の場合は ML-ADWS の方が性能が高くなっている．

このように行列積をタスク並列の形に記述できるようにすることで，様々なタスク並列スケジューラを利用することができるようになる，行列のサイズや問題の性質などによって，適切なタスク並列スケジューラを用いることで様々な問題に対して高い性能を得ることができる．

peano-hybrid

peano-hybrid を用いた再帰的なブロッキングの並列化に関しては，2.2.2 で説明した，一段階の逐次の再帰的なブロッキングアルゴリズムをタスクの分割がなされた後に入れる実装を用いて入れていない場合との比較を行なった．

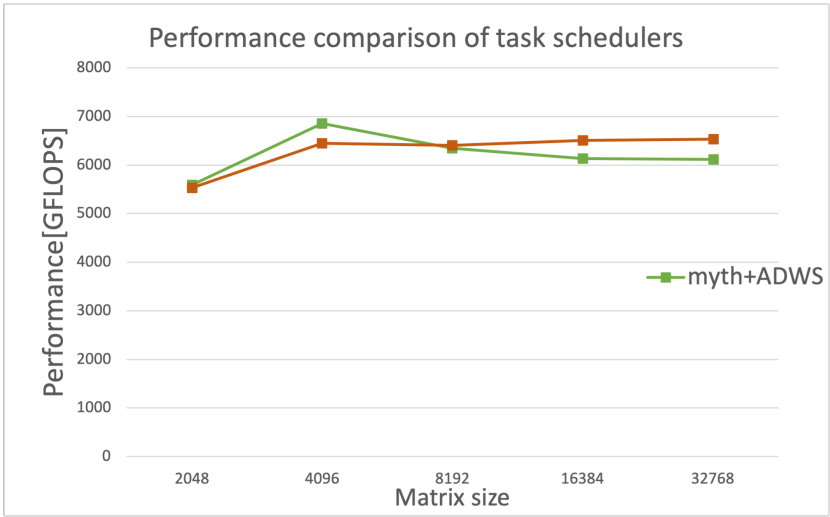


図 3.7: 行列積における ADWS と ML-ADWS の比較

図 3.8 に MassiveThreads と ADWS を用いた 2 種類の実装と既存の BLAS ライブラリの性能を示す．myth+ADWS(block) が逐次実行の部分内で再帰的なブロッキングを 1 度行う実装である．

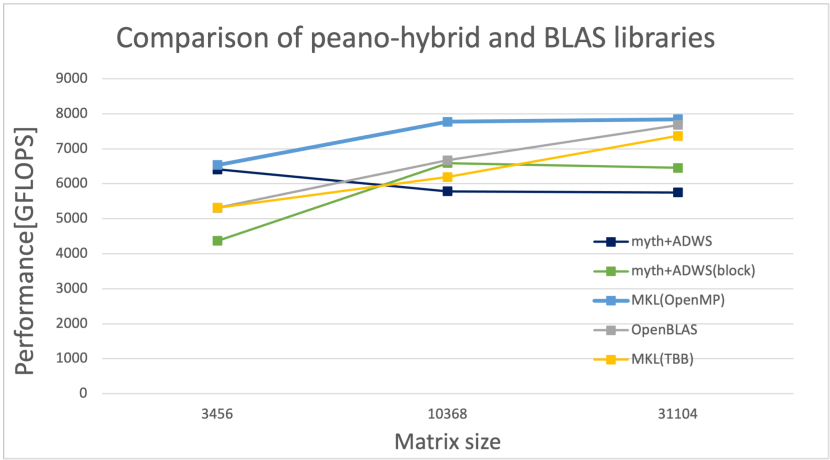


図 3.8: peano-hybrid なメモリレイアウトを用いた実装と既存の BLAS ライブラリの性能比較

行列のサイズが 10368 と 31104 と大きい場合には，逐次の処理に計算順序を決めた行列積の実装の方が性能が高くなっている．これは 2.2.2 で説明したように，ペアノ曲線を元にしたメモリレイアウトの際は，特定の計算順序を定めた方がデータの局所性が良くなるからだと考えられる．

また，行列サイズが 3456 の場合はシンプルなタスク並列の実装の方が性能が高い．これは再帰処理の一段階分を逐次処理に入れてしまったために，行列サイズが小さい場合には十分な並列性が生まれないことが原因であると考えられる．

3.4 並列性がネストした場合の行列積

本研究の目的としての合成可能性のある並列化された行列積を達成するにあたり，並列性が入れ子になっている状況での行列積演算について評価を行なった．

タスク並列が用いられている並列化プログラムは一般的に並列性がネストし，並列度がコア数を上回る状況であっても性能が低下しずらいという特徴がある．そのため，本研究で設計した行列積演算においても高い合成可能性が期待される．

本評価をするにあたり，並列化された内部で行列積演算が呼び出されるようなアプリケーションとして，三重対角行列の固有値分解の構造をベンチマークとして用いた．

3.4.1 三重対角行列の固有値ソルバ

三重対角行列は主となる対角線に加え，その上下の対角線上の要素が 0 ではない行列のことを表す．

$$\begin{bmatrix} a_1 & b_1 & & & 0 \\ c_1 & a_2 & b_2 & & \\ & c_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ 0 & & & c_{n-1} & a_n \end{bmatrix}$$

三重対角行列の固有値ソルバとは，$n \times n$ の実対称三重対角行列 A が与えられている際に，

$$A = Q\Lambda Q^T$$

を満たす直行行列 Q と対角行列 Λ を求める計算を考えるプログラムである．

この三重対角行列の固有値ソルバは実対称密行列やエルミート密行列における固有値問題を解く際に用いられるソルバの中の主要なサブソルバの一つである．

密行列における固有値問題は様々な数値計算アプリケーションの中核をなす重要な計算であるので，三重対角行列の固有値ソルバの性能向上は間接的に応用アプリケーションの性能向上に寄与する．

この三重対角行列の固有値を求めるプログラムは一般的に分割統治法によって実装されている [51]．そのプログラム内で，再帰的な分割により生成されたより小さな問題が解かれた後にマージする処理を行うのだが，その際に行列積演算 (GEMM) が 2 回呼び出される．従って，この三重対角行列を求めるプログラムがタスク並列のような形で並列に実装されていた場合は，行列積演算をする際に並列性がネストしてしまう状況が発生する．このような場合，既存の BLAS ライブラリによる行列積演算では性能が低下することがある．

3.4.2 ベンチマーク

本研究の合成可能性を評価するベンチマークとして，三重対角行列の固有値を求めるというプログラムから，分割統治法のマージの際に行列積演算が 2 回呼び出される構造を抜き出し，ベンチマークとして実装した．ベンチマークにおける分割統治はタスク並列の形で実装し，並列性がネストするような状況を設定した．ベンチマークとして実装した構造の疑似コードをアルゴリズム 8 に示す．

アルゴリズム 8 ベンチマークの構造

```
procedure BENCHMARK(A)
    if problem size < Threshold_size then
        do sequenially
    else
        spawn_task(benchmark(A_00))
        spawn_task(benchmark(A_11))
        gemm(A_00,specific matix)
        gemm(A_11,specific matix)
    end if
end procedure
```

この gemm の部分に morton-hybrid のメモリレイアウトを用いた MassiveThreads と ADWS による我々の手法の実装と既存の BLAS ライブラリの gemm 関数を用いて評価を行なった．評価するために測定するものはベンチマークに対する実行時間とした．そのため，逐次実行の部分は何も処理を行わずに，gemm の実行にかかる時間がなるべく反映されるような設計にした．

我々が実装したタスク並列な行列積は内部で MassiveThreads を用いているため，ベンチマークないのタスク並列処理系も MassiveThreads とし Ours とした．既存の BLAS ライブラリとして OpenBLAS は内部で OpenMP が用いられているため，タスク並列を作り出す処理系も OpenMP とした．また，MKL に関しては内部で OpenMP を用いてるものと TBB を用いているものがあるので，それぞれに対してベンチマークを実行した．

表 3.4: ベンチマークの実行時間比較

	Ours	Intel MKL(OpenMP)	Intel MKL (TBB)	OpenBLAS
size : 8192	0.0561 [sec]	2.2297 [sec]	0.0912 [sec]	2.2665 [sec]
size : 16384	0.4497 [sec]	11.4871 [sec]	0.6413 [sec]	17.3546 [sec]

ベンチマークに用いた行列サイズが 8192 の時，morton-hybrid のメモリレイアウトで Mas-

siveThreads と ADWS で実装する手法では OpenMP による既存の並列化された BLAS が並列に呼び出されている状況よりもとても高速であることがわかった．これは並列性がネストしてしまうことで，コア数以上に並列性が増してしまうオーバーサブスクリプションな状態となってしまう．また TBB が使われているアプリケーションでは，アプリケーション内のすべての TBB スケジューラ間でスレッドのグローバルな固定サイズプールを共有することにより、リソースのオーバーサブスクリプションを防止しようという構造があるため性能低下があまり起きていないと考えられる．

OpenMP において各スレッドは自身の処理が終わっても 200ms 程度コアを占有しながら待機しているため，コア数以上のスレッドが存在している状況ではとても実行が遅くなってしまう．そのような実装となっているため，OpenMP などを用いて並列性をネストさせた場合は性能が低下するということが指摘されており，並列性のネストした状態に対する研究が数多く行われている．[26][38].

三重対角行列の固有値ソルバの分割統治法を用いている構造をタスク並列の形で記述せず，gemm 関数の部分のみ既存の BLAS ライブラリで並列に実行した際は，行列サイズが 8192 の場合，MKL(OpenMP):0.0731 [sec]，MKL(TBB):0.0852 [sec]，OpenBLAS:0.1190 [sec] となり，並列性がネストしていなく性能低下が起きなかった場合でも我々の手法の方が性能が高いということがわかる．これは，行列サイズが 8192 程度のベンチマークではサイズの小さい行列積が多く呼び出されるが，行列サイズが小さい時には並列化のみにおいて我々の手法の方が性能が高かったためであると考えられる．

行列サイズが 16384 の場合は, 並列化がネストした状態の我々の手法は Ours:0.4497 [sec] であり，並列化がネストしていない状態の各 BLAS ライブラリにおいては MKL(OpenMP):0.4902 [sec]，MKL(TBB):0.4734 [sec]，OpenBLAS:0.6855 [sec] という結果になっている．これから，並列性がネストしていない状態では問題のサイズが大きくなるにつれて我々の手法との差が詰まってきているということがわかる．これは大きなサイズの行列積に関しては既存の BLAS ライブラリの方が性能が高いことに起因する．

Intel MKL(OpenMP) では MKL 内で並列化されるスレッド数を指定することができる．並列性がネストしたベンチマークにおける MKL 内でのスレッド数による性能比較を図 3.9 に示す．

MKL 内部の並列度が 1 や 2 の時は大きな行列積演算が 1 コアで実行されてしまうため実行時間は伸びてしまっている．

本研究で提案した手法はタスク並列の形で実装されているため，このような行列積が並列して呼ばれるような並列性がネストしたプログラムで高い性能を発揮し，高い合成可能性を持つということがわかった．

3.5 他のプラットフォームでの実験

本研究において，性能可搬性を評価するために他のアーキテクチャ上でも実験を行なった．

弊研究室にある Spica 上で複数コア向けの行列積演算を行い，性能を評価した．spica は Xeon

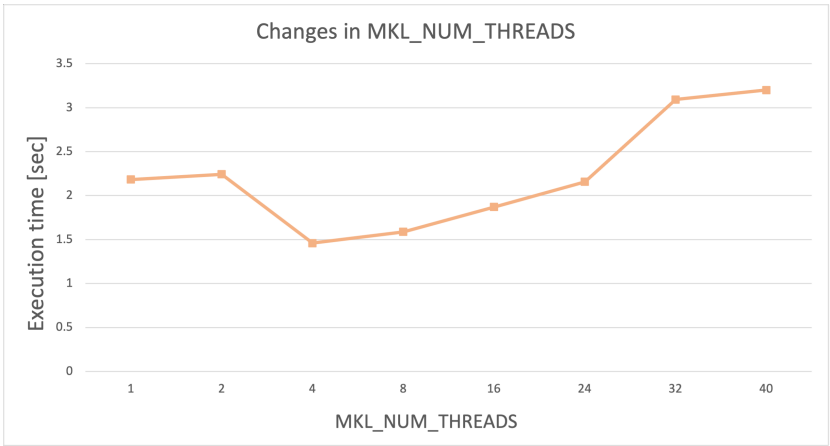


図 3.9: MKL 内の並列数によるベンチマークの実行時間比較

E7-8890 v4 を 4 基搭載した NUMA 環境である．

本評価においては Intel MKL を用いることができなかったため，再帰的なブロッキングに基づく行列積の最下層の行列積カーネルとしては OpenBLAS を逐次で用いた．この環境において，これまでの評価で高い性能が発揮されていた morton-hybrid のメモリレイアウトで MassiveThreads と ADWS を用いた実装と並列化された OpenBLAS の性能を比較し，その結果を図 3.10 に示す．

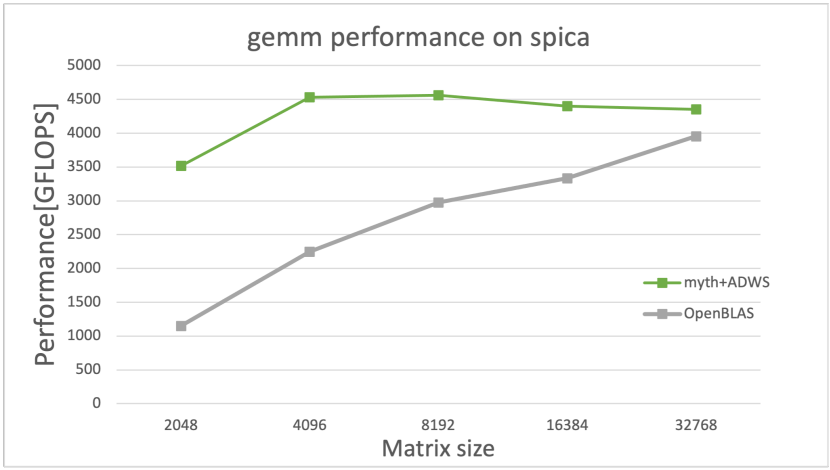


図 3.10: spica 上での性能比較

メモリレイアウトの変更と高度なタスク並列スケジューラによる実装は，このマシン上でも高い性能を発揮している．これは 4 つの NUMA ノードからなる環境であり，`numactl` コマンドによりデータが均等に配置される．そのため，行列の要素を演算に用いる際に違う NUMA ノード上の遠いメモリを参照することが多いことが，OpenBLAS で行列サイズが小さい時の性能が低くなっている要因の一つと考えられる．また，この環境ではコア数が 96 コア存在し，行列サイズが小さい場合は十分に並列化が行われていないということが考えられる．それに対し，我々のタスク並列

化による実装では，行列サイズが比較的小さい場合でも分割統治法によって高い並列性と高い性能を得ることができ，コア数の多寡に対しても性能可搬性があると言える．

再起的に分割された最下層の行列サイズはこの環境においても L2 キャッシュのサイズ以下であるため，L2 キャッシュと L3 キャッシュに関しては本研究の手法は cache-oblivious のようなアプローチになっており，高い可搬性を持っている．またキャッシュに関してだけでなく，NUMA な環境においてもとても高い可搬性を示すということがわかる．

第 4 章

関連研究

4.1 cache-oblivious な行列積演算

分割統治法を用いた再帰的なブロッキングを行う行列積演算に関しての研究は多く行われている．この手法における大きなメリットとして cache-oblivious[21][22] なアルゴリズムになることが多いということが挙げられる．

cache-oblivious というのはソフトウェアがアーキテクチャのキャッシュ構造やキャッシュサイズに対しての情報をパラメータに含まないという考え方であるそのため，少ない移植コストで高い性能可搬性を得ることが期待されている．この cache-oblivious な線形代数計算についていくつかの比較調査がなされてきた．[20][49][50]

先行研究で紹介した空間充填曲線に基づいた行列積以外に，この cache-oblivious という考え方を利用した行列積演算を紹介する．

4.1.1 メニーコアなシングルノードへの適用

シングルノードにおける cache-oblivious な行列積を利用した研究として，Butcher らの研究 [14] を紹介する．

この研究は Intel Knight Landing (KNL) Processor を代表的なアーキテクチャとするメニーコアと呼ばれる環境に対して研究されたものである．

近年の KNL をはじめとするメニーコア環境では, 今までのレジスタ, L1, L2, L3, DRAM といったメモリ階層に加えてキャッシュと DRAM の中間の性能に当たる MCDRAM（Multi-Channel DRAM）と言われる記憶領域が追加されている．この中間のメモリは KNL 等の CPU だけでなく GPU 内においても HBM に相当するものであり，この中間のメモリの有効利用というのは GPU を利用する分野においても大きな意味を持つもになる．

MCDRAM というもは 3D スタックの高速メモリであり，一般的に DRAM として用いられている DD4 規格とレイテンシはそれほど変わらないものの，4 倍のバンド幅を持つ記憶領域である．KNL プロセッサ環境ではこの MCDRAM をキャッシュとして扱うかメモリとして扱うか，また

はある一定のサイズはキャッシュとして残りはメモリとして扱うかなど幅広く設定可能である．

この研究では MCDRAM をキャッシュモードとして扱い，cache-oblivious な行列積演算を実装している．

この研究の実装では，再帰的に小行列積のタスクを分割して並列化を行なっていくのではなく，初めにハードウェア上で並列可能なスレッドの数だけ問題となる行列を分割し，各スレッドに小行列の問題を割り当てている．その後，各スレッドは逐次の分割統治法による cache-oblivious な再帰的なアルゴリズムが実行されるようになっている．

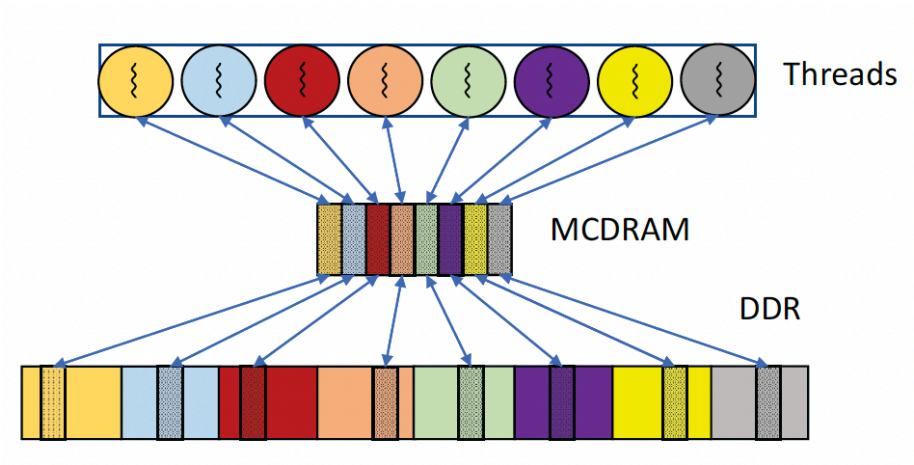


図 4.1: DRAM から各スレッドで実行されるまでのデータの流れ [14]

その際に，各スレッドの逐次の cache-oblivious アルゴリズムが進むにつれ，図 4.1 のように必ず各スレッドが処理に用いるデータサイズの合計が MCDRAM のサイズにフィットするような状況が生まれると考えられる．さらに逐次実行が進むにれて，各スレッドの扱うデータサイズがそれぞれの下層のキャッシュサイズにフィットするようになり，効率的に行列積の実行が行われるようになる．

この方法による行列積の性能と Intel MKL の比較を図 4.2 に概要を示す．

この結果の横軸は，行列積 $AB = C$ の演算において求められる行列 C を正方行列とした際の A の行と列のサイズの比を表している．どの比の演算においても，A, B, C すべての行列の要素の合計が約 40 億個になるように設定されている．

cache-obliviou と MKL の結果を比較すると全ての比率の場合で MKL の方が実行時間が短いということがわかる．したがって，この MCDRAM を利用する cache-oblivious アルゴリズムは MKL よりも性能が劣っているということになる．

MCDRAM はレイテンシは DRAM と変わらずバンド幅のみ DRAM よりも高速というものなので，computational intensity(= # of math operation/ # of memory accesses) が低い処理に対してメリットが存在する．行列積において行サイズと列サイズの比が 1 から離れるほど

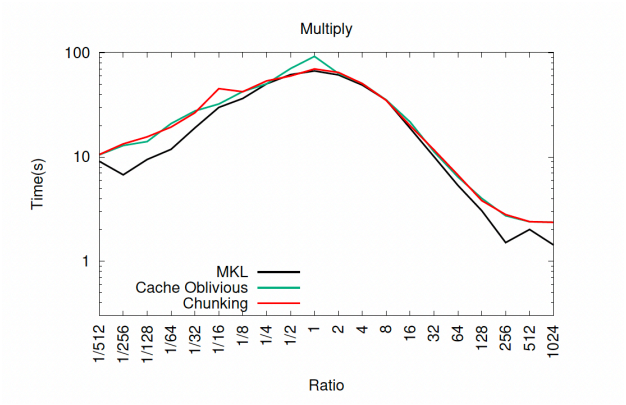


図 4.2: MKL と MCDRAM を利用した行列積の性能比較 [14]

computational intensity が小さい問題となっているので，比が 1 から遠い演算に関しては MKL の性能に近づいているものも存在している．本来，行列積は computational intensity が高い処理であるので，MCDRAM をキャッシュとして扱った場合には DRAM から MCDRAM を経由してキャッシュにデータが送られるので，MCDRAM のレイテンシの性質上ある程度データ移行が遅くなってしまうと考えられる．

この研究は MKL の性能に届かなかったものの，中間のメモリという新たなアーキテクチャに対して cache-oblivious のメリットである可搬性を活かして取り組んだ研究である．実際にある特定のサイズの問題では，MKL に近い性能を出すことが可能であった．今後は，computational intensity が低く，MCDRAM の特性を活かせる問題に対して cache-oblivious な手法を適用することが望まれる．

4.1.2 マルチノードへの適用

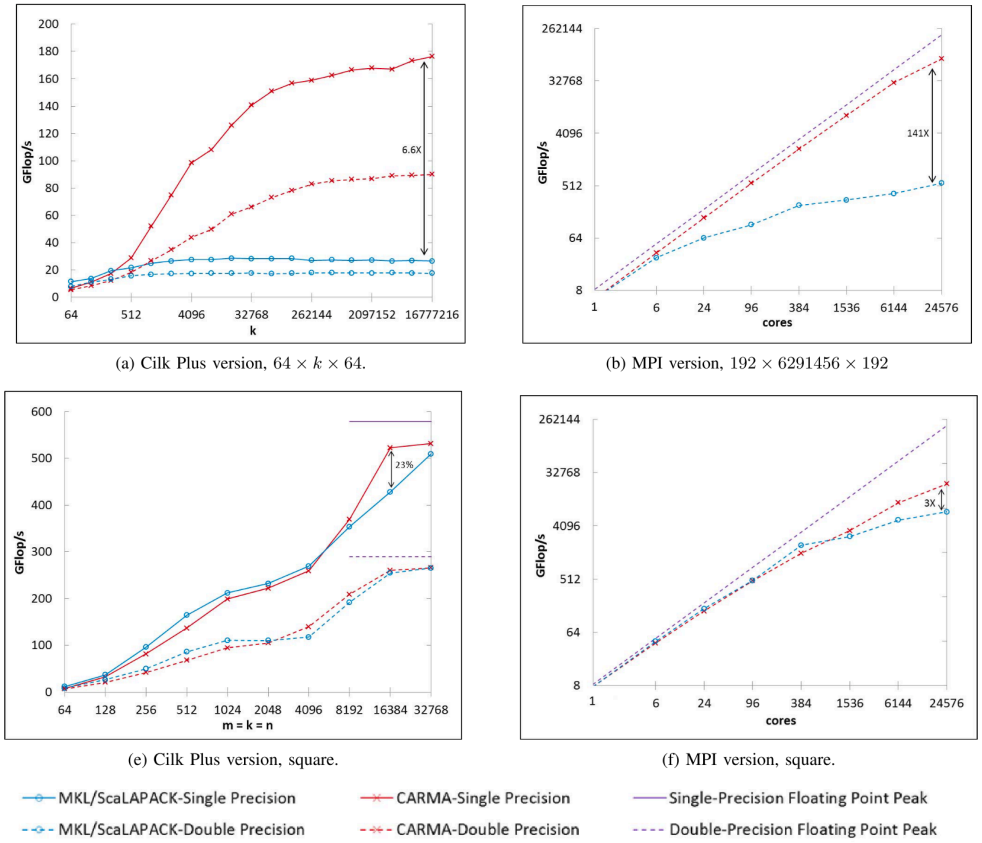
cache-obliviou な行列積演算をマルチノードへ拡張したものとして，Demmel らが提案した CARMA[17] が挙げられる．

この研究では CARMA と呼ばれるマルチノードを対象とした cache-oblivious かつ network-oblivious な行列積演算が紹介されている．また，この研究では正方行列同士の行列積がメインではなく，長方形の形をした行列同士の行列積が主に注目されている．多くの行列積に関する研究が正方行列同士の行列積に注目しているので，長方形の形をした行列に関しての行列積を扱っている点でこの研究は珍しい．

この研究では，長方形の行列を対象としているため，これまで紹介した手法のような 2×2 や 3×3 といったような平方数での行列の分割は行われない．CARMA ではまず，3 つの行列 $A(m \times k)$, $B(k \times n), C(m \times n)$ を用いた行列積 $AB = C$ の演算に登場するパラネータである，行列に関わる 3 つのサイズ m, k, n のうち最も大きいものを半分に分割し，2 つの小さな行列積の問題を生成する．

次に，生成された 2 つの行列積演算に対して，再帰的に問題を解いていく．しかし，再帰的に問題を解き始める際にメモリ環境によって 2 種類解き方に分岐する．その分岐は使用できるメモリの量で決められ，十分な量のメモリが使用できる場合は BFS（幅優先探索）に基づいた再帰的な行列積演算を行うが，十分なメモリがない場合は DFS（深さ優先探索）に基づいた再帰的な行列積演算を行うように設計されている．

BFS に基づいた再帰的な行列積とは，行列計算で使用できるプロセッサ数が P 個の時，生成された 2 つの小さな行列積演算それぞれに対して $P/2$ 個ずづのプロセッサを割り当てて演算を進めるということ行う．また，DFS に基づいた再帰的な行列積とは，行列計算で使用できるプロセッサ数が P 個の時．生成された 2 つの小さな行列積演算のうち片方の演算が P 個全てのプロセッサを使用して演算を進めるということを行う．



(a) Cilk Plus version, $64 \times k \times 64$.

(b) MPI version, $192 \times 6291456 \times 192$

(e) Cilk Plus version, square.

(f) MPI version, square.

図 4.3: 長方形の行列積と正方行列の行列積の性能 [17]

実装は，1 ノード内の並列化は Cilk Plus で並列されており，ノード間は MPI[6] を用いて実装されており，共有メモリ環境と分散メモリ環境どちらに対しても実験がなされている．

図 4.3 に性能比較を示した．行列積演算のパラメータとなる 3 つの行列のサイズ m, k, n のうち 1 つが他の 2 つと比べてかなり大きく，長方形の行列積演算となる場合は共有メモリ環境内，分散

メモリ環境のどちらでも Intel MKL の性能を上回る結果となっている．これは，再帰的に分割していくアルゴリズムは行列の形が極端でも，分割していくにつれて一般的な行列積の形に近づくので性能低下は起きにくいが，Intel MKL が極端な形の長方形の行列積がでのパフォーマンスがとても悪いことを示している．

正方行列に関しては，共有メモリ環境内，分散メモリ環境のどちらでも，行列サイズがとても大きい時のみ Intel MKL の性能を上回っており．ほとんどの場合で Intel MKL よりも性能が劣る結果となった．

この研究は，行列積の性能以外にもマルチノードにおける長方形の行列における通信コストの解析や最適化が考慮されており，長方形の行列に関する行列積演算や再帰的に分割していくアルゴリズムを分散メモリのマルチノード環境への拡張に関した研究である．

4.2 合成可能性 (並列性のネスト)

BLAS ライブラリをはじめとする並列化されたライブラリの合成可能性に関して多くの研究がされている．ここではスケジューラ側でこの問題に対処しようとしている Pan らの Lithe[33] について紹介する．

この研究は，内部で並列化されたライブラリが上位のアプリケーションによってさらに並列に呼び出しが行われた場合に起きる性能低下を問題とし，その解決を目的としている．

全てのライブラリの並列化が TBB のみによって行われているというような状況では，各 TBB のスケジューラが全体のスレッド数を共有してオーバーサブスクリプションな状況を避けるような実装がされている．しかし，並列化に用いらているライブラリが TBB と OpenMP のどちらも用いているような場合では，並列化ライブラリがそれぞれでスレッドを生成していまい，一つの物理的なコアにつき複数のスレッドが割り当てられてしまう．これらによって，スケジューリングのポリシーやキャッシュにあるデータの干渉などがスレッド間で生まれてしまう．

Lithe という考え方は Harts と呼ばれる Hardware threads，つまり一つの物理的なコアに対して一つのスレッドを生成する．この Harts はコードが実行されている最中に必要に応じて動的に生成されうる通常のスレッドと違い，コードが実行される前にランタイムに対して Harts が割り当てられる．

そして，Lithe はこのライブラリ間において Harts を交換するための標準的なメカニズムを設定している．ランタイムのインターフェースとスケジューラ呼び出しのインターフェースの 2 種類を実装している．それによって異なるスケジューラ間において自身がコントロールしている Harts をそれぞれ管理したり，他のライブラリに譲渡したりできるようにしている．

具体的には既存のスレッド並列を用いるライブラリである OpenMP や TBB などのスケジューラを Lithe の仕様に対応するように移植を行なっている．例として，OpenMP では複数のワーカーによって並列指定範囲が並列に処理されるが，Lithe 版に移植した OpenMP ではスケジューラーは利用可能な Harts に対してワーカーを多重に割り当てるようになる．

この概要を図 4.4 に示す．

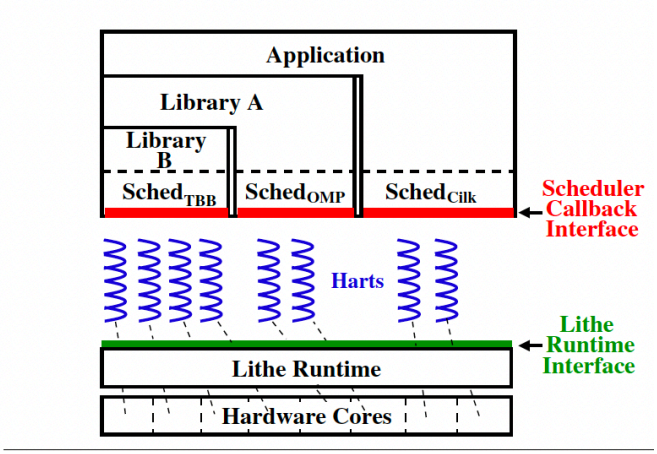


図 4.4: Lithe ランタイム [33]

このように，Lithen に対応させた OpenMP や TBB とオリジナルの OpenMP や TBB を用いて疎行列の QR 分解 [16] において比較を行なった．TBB を並列に Intel MKL を呼び出すことに使い，MKL の中では行列演算を並列化するために OpenMP が使われているという並列性がネストしている状況である．

その結果が表 4.1 である．

表 4.1: OMP と TBB のスレッド数による SPQR 分解の性能比較 [33]

		landmark	deltaX	ESOC	Rucci1
Seq OMP / Seq TBB	OMP=1, TBB=1	7.8	55.1	230.4	1352.7
Par OMP / Seq TBB	OMP=16, TBB=1	5.8	19.4	106.9	448.8
Seq OMP / Par TBB	OMP=1, TBB=16	3.1	16.8	78.7	585.8
Out-of-the-Box	OMP=16, TBB=16	3.2	15.4	73.4	271.7
Manually Tuned	OMP=5, TBB=3		12.0		
	OMP=5, TBB=16			61.1	
	OMP=8, TBB=11	2.9			
	OMP=8, TBB=16				265.6
Lithe		2.7	10.4	60.4	248.3

オリジナルの OpenMP と TBB に関して，性能が出るスレッド数をそれぞれ手動で設定した場合よりも Lithe に対応させた OpenMP と TBB てスレッド数を制御した方が良い性能を示している．

この研究のスコープは本研究よりも大きいスコープである並列化ライブラリ全体を対象としている．BLAS ライブラリに限らず並列化ライブラリ全体を対象としても，本研究の骨子である同じ処理を再帰的に分割していく処理への変更をすることで，この問題に対してもアプローチできると考えている．しかし，やはり全体的な問題として捉えた際は，この研究のようにスケジューラに関し

ての研究を行うことがより一般的である．

4.3 性能可搬性

本研究は再帰的にブロッキングによる cache-oblivious に似たアプローチを用いることで，行列積の性能可搬性を高めることを目的としていた．既存の BLAS ライブラリにおいては，cache-oblivious 以外の手法で性能可搬性を高めようとしているものがある．

4.3.1 自動チューニングによる可搬性向上

自動チューニングを目指したものとして ATLAS が挙げられる．

ATLAS はアーキテクチャの構造を踏まえて自動チューニングした BLAS の C のソースコードを生成して BLAS ライブラリとして実行できるライブラリである．ATLAS のチューニングの概要を図 4.5 に示す．

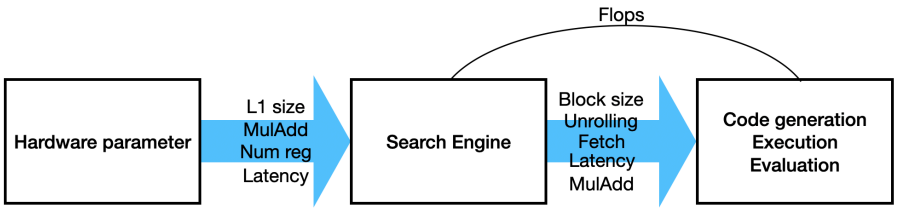


図 4.5: ATLAS におけるチューニングの流れ

ATLAS はインストール時にチューニングをし，その後は高速化された BLAS ライブラリとして使用できる．キャッシュブロッキングなどのチューニングに関して，パラメータを動的にサーチして決定するアプローチが取られている．インストール時にそのアーキテクチャの L1 キャッシュのサイズや乗加算器の有無，レジスタの個数やレイテンシなどの情報をプログラムを走らせて収集する．その後，収集した情報からブロックサイズやアンローリングの段数などのパラメータについてある程度の範囲を決定する．そして，インストール中にその決めたパラメータの範囲内で何度も C 言語のコードの生成と線形代数演算の実行を行いながら，最終的なパラメータの値を決定しインストールが終了する．

ATLAS は得られるアーキテクチャに関する情報が他の手法と比べ少なく，コード生成が C 言語の生成までで, コンパイラを一般のコンパイラを用いているため，専門家がアセンブリレベルで高速化したライブラリほど性能をあげることができない．また，ビルドするシステムとホストするシステムが違うと自動チューニングができなくなるなどの課題がある．

4.3.2 directive-based な方法による可搬性向上

directive-based なものとしては POET[35] を利用したもの [48] や AUGEM[46] などが挙げられる．

AUGEM は directive-based な手法であり，ある程度の自動チューニングと予め専門家によって書かれているテンプレートとなるアセンブリコードを組み合わせて自動で高速化を行う．AUGEM のフレームワークを図 4.6 に概要を示す．

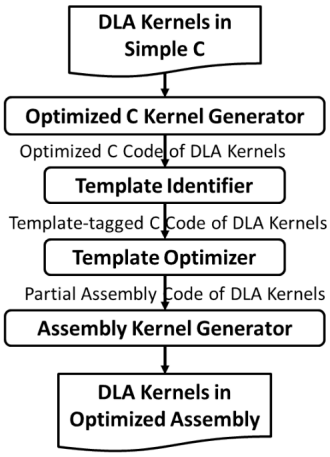


図 4.6: AUGEM フレームワーク [46]

DLA とは Dense Linear Algebra の略で密な線形代数演算のことである．AUGEM はもともと演算に関するシンプルな C 言語のコードを保持している，最初の段階としてはこの C 言語のコードをより高速化された C 言語の source-to-source で変換する．この段ではブロックサイズやループアンローリングの段数などを ATLAS と同じように実験ベースで定める．その次の段階として，その高速化された C 言語のソースの各部分が DLA でよく使われる処理のどの部分なのかを分類する．分類する際には予め用意されていた各処理の様々なテンプレートを参照し，近いテンプレートとソースの一部分をタグづけする．そして，タグづけされた C ソースを元にそのテンプレートの処理にあったアセンブリコードに置き換える．このアセンブリコードは予め専門家によって高速化されたものがいくつか用意されている．最後に，アセンブリにならなかった部分をコンパイラでアセンブリに変え，全体として演算を実行するアセンブリコードを生成する．

AUGEM は ATLAS 等と比べて良い性能を示しているが，アセンブリへのチューニングは手動でやられたものを用いており，x86 のアーキテクチャへの実装のみで，他のアーキテクチャには対応できていない．また，GEMM，GEMV，AXPY，DOT の 4 つの演算が実装されている．

4.3.3 コンパイラ最適化による可搬性向上

コンパイラ最適化によるものは LGen[40] や POCA[42] が挙げられる．

POCA はコンパイラを最適化する手法である．コンパイラを最適化することで BLIS におけるマイクロカーネルの生成を行う．そのため，この手法は BLIS や OpenBLAS のカーネルを生成する手法として一緒に用いることができる．この POCA のフレームワークを図 4.7 に示す．

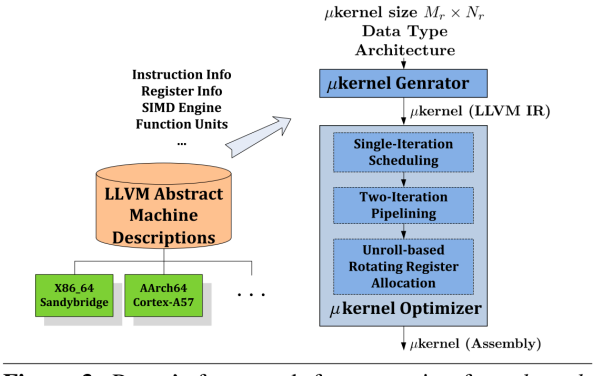
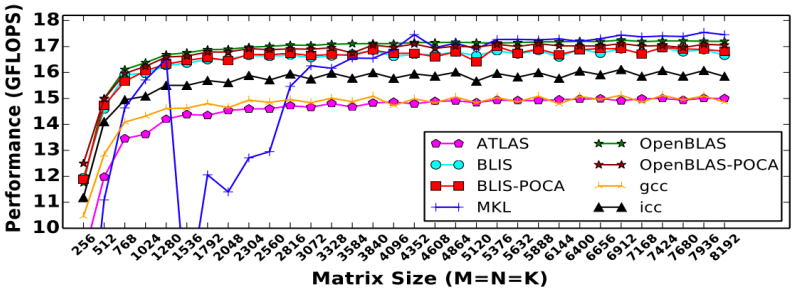


図 4.7: POCA フレームワーク [42]

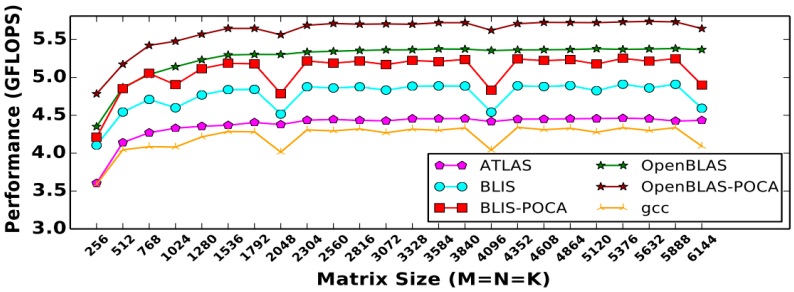
POCA においては LLVM から取得できるアーキテクチャの情報を元に演算実行するマイクロカーネルを生成する．LLVM を利用することでアーキテクチャの命令セットやレジスタ情婦，SIMD セットなど知ることができる．それらの情報を元に，SIMD などでベクトル化されたマイクロカーネルを最初に LLVM IR という中間言語で生成する．その後，その中間言語からアセンブリにコンパイルする際に 1 イテレーションに関するスケジューリングや 2 イテレーションを合わせてみた際のパイプラインなどのチューニングを行う．最後に，実行中にレジスタを全て埋めることができるようにアンローリングに関するチューニングを行ってアセンブリを出力する．

図 4.8 に既存の BLAS ライブラリや ATLAS[47]，POCA[42] を組み合わせた手法の性能を比較したものを示す．

この手法を用いて OpenBLAS のカーネルを生成するという実装は，ある程度のサイズの行列計算まではとても高い性能を示しているまた，ATLAS[47] の開発はとても古く，性能は他の BLAS ライブラリに比べて圧倒的に低くなってしまいっている．



(a) Sandybridge



(b) Cortex-A57

図 4.8: さまざまな BLAS ライブラリの GEMM の性能比較 [42]

第 5 章

結論と今後の課題

本研究では，再帰的なブロッキング手法によって行列積をタスク並列可能な形にし，キャッシュ効率の良いメモリレイアウトと高性能なタスク並列スケジューラを併用することにより，高い性能を持つ行列積のタスク並列化の提案・実装を行なった．morton-hybrid と peano-hybrid の形のメモリレイアウトを用いることにより，小行列積がタスクに分解され各スレッドで実行される時のキャッシュの効率が高くまり，分割統治を用いて再帰的にブロッキングされた行列積をタスク並列化させた場合の性能を向上させることに成功し，L2，L3 キャッシュへの高度な最適化は行なわずに，同じ逐次実行の処理を行なっており，L2，L3 キャッシュにも最適化を行う Intel MKL ライブラリにおける並列化よりも比較的小さな行列サイズで 3% 程度の性能向上を達成した．

高性能な行列積をタスク並列の形で実装することにより，並列性が入れ子になってしまった場合でも既存の BLAS ライブラリよりも性能を損なうことな行列積演算を行うことができ，並列性がネストした状態のベンチマークにおいて，MKL や OpenBLAS よりも高速に演算を行うことができた．また，分割統治法による再帰的なブロッキングを用いることで，大きい行列積の問題を小さな行列積に分割してキャッシュサイズに収まるサイズの問題を解くことで，キャッシュサイズを必要としない cache-oblivious に近い性能可搬性のある行列積演算となった．

一方，本手法においては行列積として正方行列の積のみを対象とした．また，morton-hyrid や peano-hybrid といったメモリレイアウトも 2×2 や 3×3 の分割に限定して評価を行なった．そのため，一般的な行列積演算を行うにあたり，正方行列以外の形の行列やメモリレイアウトの拡張ということが今後必要である．この問題に対しては，microcell[34] と呼ばれるモートン曲線の任意のサイズの拡張などの手法などが存在するが，今後の課題とする．

本研究においてはタスク並列を用いた行列積演算を考えたが，メモリレイアウトと高性能なタスクスケジューラを他の線形代数的な処理に用いるこで行列積以外にも合成可能性と性能可搬性がありかつ高性能な演算を実現できる可能性があると考えられる．

謝辞

田浦健次朗先生には卒業論文のテーマ決めから修士論文のご相談まで大変お世話になりました．修士課程での 2 年間は世界的な伝染病の蔓延により，長期にわたって満足に生活や研究を行うことができませんでしたが，いつも心配をしてくださり様々な相談やお話をしてくださり大変感謝申し上げます．．助教である佐藤さんには研究へのストイックな姿勢から研究活動というものを知ることができました．感謝申し上げます．また，私の研究や輪読においてアドバイスやコメントをいただきとても参考になりました．ありがとうございました．博士課程の学生である椎名俊平さんには，修士論文のテーマ決めから研究の方向性の相談やアドバイス，修士論文の執筆まで多くのサポートをしていただきました．心からの感謝を申し上げます．伝染病の影響でほとんどがリモートでの作業となりましたが，時々研究室で出会えた同期を初め，研究室のメンバーとの雑談や議論はとてもほとんどリモートの状況下で心の支えになっていました．ありがとうございました．最後に学部から大学院まで 6 年もの間，陰ながらサポートをしてくださった家族がいなければ私が無事に修士論文を書き上げることはできなかったと思います．ここに特別な感謝の意を評したいと思います．ありがとうございました．

参考文献

[1] acml_userguide.pdf.pdf. https://developer.amd.com/wordpress/media/2012/10/acml_userguide.pdf.pdf. (Accessed on 12/01/2020).

[2] clblas - clmathlibraries/clblas: a software library containing blas functions written in opencl. https://github.com/clMathLibraries/clBLAS. (Accessed on 12/01/2020).

[3] cublas :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cublas/index.html. (Accessed on 12/01/2020).

[4] Engineering and scientific subroutine library. https://www.ibm.com/support/knowledgecenter/en/SSFHY8/essl_welcome.html. (Accessed on 12/01/2020).

[5] Intel® math kernel library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. (Accessed on 12/01/2020).

[6] Open mpi: Open source high performance computing. https://www.open-mpi.org/. (Accessed on 01/22/2021).

[7] Openblas : An optimized blas library. https://www.openblas.net. (Accessed on 12/01/2020).

[8] 富岳 cpu a64fx 用ディープラーニングライブラリの深層-研究者が語る開発の軌跡- - fltech - 富士通研究所の技術ブログ. https://blog.fltech.dev/entry/2020/11/18/fugaku-onednn-deep-dive-ja. (Accessed on 04/21/2021).

[9] Z. Anderson. Efficiently combining parallel software using fine-grained, language-level, hierarchical resource management policies. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 717–736, 2012.

[10] M. Bader and C. Mayer. Cache oblivious matrix operations using peano curves. pages 521–530, 2007.

[11] M. Bader and C. Zenger. A cache oblivious algorithm for matrix multiplication based on peano's space filling curve. *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, pages 1042–1049, 2005.

[12] M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on a peano curve. *Linear Algebra and its Applications*, 417(2):301–313, 2006.

Special Issue in honor of Friedrich Ludwig Bauer.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[14] N. A. Butcher, S. L. Olivier, and P. M. Kogge. Cache oblivious strategies to exploit multi-level memory on manycore systems. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 42–51, 2020.

[15] R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001. New Trends in High Performance Computing.

[16] T. A. Davis. Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[17] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 261–272, 2013.

[18] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[19] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, 1988.

[20] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.*, 46:3–45, 03 2004.

[21] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. pages 285–297, 1999.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):1–22, 2012.

[23] K. Goto and R. van de Geijn. On reducing tlb misses in matrix multiplication, 2002.

[24] K. Goto and R. A. Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.

[25] A. Heinecke and C. Trinitis. Cache-oblivious matrix algorithms in the age of multicores and many cores. *Concurrency - Practice and Experience*, 27(9):2215–2234, 2015.

[26] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji. Bolt: Optimizing openmp parallel regions with user-level threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 29–42, 2019.

[27] Jack Dongarra, Antoine Petitet, and Clint Whaley. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2002.

[28] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[29] K. P. Lorton and D. S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *Proceedings of the 2006 Workshop on MEmory Performance: DEaling with Applications, Systems and Architectures, MEDEA '06*, 2006:5–12, 2006.

[30] J. Nakashima and K. Taura. *MassiveThreads: A Thread Library for High Productivity Languages*, volume 8665, pages 222–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 01 2014.

[31] C. Nugteren. Clblast: A tuned opencl blas library. *ACM International Conference Proceeding Series*, 2018.

[32] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[33] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with lithe. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 376–387, 2010.

[34] M. Perdacher, C. Plant, and C. Bohm. Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition. *Proceedings - 2020 IEEE International Conference on Big Data, Big Data 2020*, pages 351–360, 2020.

[35] Y. Qing. POET: a scripting language for applying parameterized source-to-source program transformations. *Software - Practice and Experience*, 39(7):701–736, 2012.

[36] J. Reinders. *Intel Threading Building Blocks*. OReilly media, Inc., USA, first edition, 2007.

[37] S. Shiina. Efficient thread scheduling strategies for nested parallel programs. In *[Unpublished master's thesis]*. University of Tokyo, 2021.

[38] S. Shiina, S. Iwasaki, K. Taura, and P. Balaji. *Lightweight Preemptive User-Level Threads*, pages 374–388. Association for Computing Machinery, New York, NY, USA, 2021.

[39] S. Shiina and K. Taura. Almost deterministic work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[40] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. *Proceedings of the 12th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2014*, pages 23–32, 2014.

[41] V. STRASSEN. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[42] X. Su, X. Liao, and J. Xue. Automatic generation of fast BLAS3-GEMM: A portable compiler approach. *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 122–133, 2017.

[43] L. T, T. Keisuke, M. Hitoshi, and S. Miku. タスクの依存性を用いた opnemp プログラムの numa 最適化. 情報処理学会研究報告, 2016-HPC-155(25):1–7, 2016.

[44] V. Valsalam and A. Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839, 2002.

[45] D. Walker. Morton ordering of 2d arrays for efficient access to hierarchical memory. *The International Journal of High Performance Computing Applications*, 32:109434201772556, 08 2017.

[46] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2013.

[47] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. pages 1–27, 1998.

[48] Q. Yi, Q. Wang, and H. Cui. Specializing Compiler Optimizations through Programmable Composition for Dense Matrix Computations. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2015-January(January):596–608, 2015.

[49] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358–386, 2005.

[50] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. pages 93–104, 01 2007.

[51] H. Yusuke. Implementation techniques of divide-and-conquer method for a manycore processor system. 情報処理学会研究報告, 2017-HPC-158(20):1–9, 2017.

[52] F. G. V. Zee, T. M. Smith, B. Marker, T. M. Low, M. Kistler, V. Austel, J. A. Gunnels, and I. Corporation. The blis framework : Experiments in portability this article discusses early results for blas-like library instantiation software. *ACM Trans. Math. Softw.*, 42(2):1–19, 2016.