SIMD 演算を用いた
高精度疎行列計算ソフトウェアの高速化

筑波大学

図書館情報メディア研究科

2020 年 3 月

菱沼 利彰

SIMD 演算を用いた高精度疎行列計算ソフトウェアの高速化

菱沼 利彰

物理シミュレーションでは大規模な疎行列に対する連立一次方程式の求解が求められる．このような連立一次方程式を解くには Krylov 部分空間法とよばれる解法を用いることが一般的である．Krylov 部分空間法は連立一次方程式の近似解を求める反復法アルゴリズムの 1 つのカテゴリで，行列の性質に応じて様々なアルゴリズムがある．多くの Krylov 部分空間法はスカラ，ベクトル，疎行列に対する演算（基本演算）の組み合わせによって構成され，最も時間がかかる演算は疎行列ベクトル積（SpMV）および転置疎行列ベクトル積（TSpMV）である．Krylov 部分空間法は丸め誤差により反復回数が増大したり，近似解が求まらなくなったりする．

収束を改善する試みとして丸め誤差を低減するためにプログラム全体や誤差の蓄積しやすい一部の演算を高精度化する手法がある．高精度演算は計算時間やメモリデータ量が多く必要になることや標準的なプログラミング環境から高精度演算を行うためのソフトウェアが少ないことが問題となる．

現実的な計算時間で高精度演算を実現するために，まずは倍精度と比べて 2 倍の精度である 4 倍精度に着目する．4 倍精度演算の実現手法の 1 つである Double-Double 精度（DD）演算は 2 つの倍精度数を用いて 4 倍精度相当の数を表現し，倍精度演算 10〜20 回を組み合わせることで 4 倍精度相当の演算を実行できる．

本研究はマルチコア CPU において Krylov 部分空間法の実装に必要な DD の基本演算を高速化することを目的とした．具体的な高速化のアプローチとして，代表的な Krylov 部分空間法の実装に必要な DD の基本演算を現在のほとんどの汎用プロセッサでサポートされている FMA（Fused Multiply-Add）命令，SIMD（Single Instruction Multiple Data Streaming）命令，マルチスレッドを用いて高速化を行う．倍精度型と DD 型を組み合わせてこれらの基本演算を簡単に扱えるソフトウェアを開発する．

FMA 命令は積和演算命令の 1 つで乗算の結果を丸めずに加算に利用することで精度よく計算できる．DD には FMA 命令を用いた演算量の少ない乗算アルゴリズムがある．FMA 命令は積と和を 1 命令で行うため積と和がペアの計算であれば最大で 2 倍の性能向上が期待できる．SIMD 命令は 1 命令で複数のデータへの同一演算を同時に実行する．現在の一般的な SIMD 命令の同時処理数は倍精度データ 4 または 8 つが標準的である．例えば 4 つの倍精度データに対する演算を同時処理すれば最大で 4 倍の性能向上が期待できる．マルチスレッドは処理をスレッドという単位に分割する．スレッドをプロセッサの複数のコアに割り当てて処理を並

列化すれば最大でコア数に比例した性能向上が期待できる．

DD 演算は倍精度演算と比べてメモリデータ量が 2 倍，演算量が 10〜20 倍であるため，演算を行うために必要なメモリからの演算あたりのデータ転送量（Byte / Flop，B/F）が低い．そのため倍精度演算と比べて演算時間の性能特性，ボトルネックが異なる．高速化は計算機環境やアルゴリズムの性能特性を考慮して行う必要があるが，4 または 8 つの倍精度演算を同時実行できる SIMD 命令を用いた疎行列に対する DD 演算がどのような性能特性になるかは明らかにされていない．そのため本研究では 4 つの倍精度演算を同時処理する SIMD 命令を用いた際の DD 演算の性能特性を明らかにし，疎行列に対する DD 演算に対して性能特性に基づいた高速化手法を検討する．

誤差の蓄積しやすい変数や演算のみを高精度化する混合精度手法を用いることですべての変数に DD を用いた場合と比べてメモリデータ量やメモリアクセス量が削減できるため基本演算の高速化が見込める．一方で高精度演算を用いることによる収束の改善と計算時間の増大はトレードオフで，どの変数や演算を高精度化すればよいかの判断は疎行列の性質やアルゴリズムによって異なる．入力に応じて精度を切り替える必要があることを考えると，精度の切り替えによってプログラム全体の変更が必要になるようなインタフェースは現実的でない．

これらのことからソフトウェアの要件として次の 3 つを定め，これらの核となる基本演算の高速化方法を検討する．

要件 1 Krylov 部分空間法に必要なスカラ，ベクトル，疎行列に対する演算があること

要件 2 DD のスカラ，ベクトル，疎行列に対する演算がマルチコア CPU で高速されていること

要件 3 倍精度と DD の混合精度演算が利用できるインタフェースをもつこと

ベクトル演算をマルチスレッド化した場合，メモリ性能に制約を受けて並列化の効率を十分に引きだせないことが判明した．そのため多くの計算時間が必要と考えられる SpMV，TSpMV においてメモリへのデータ要求量の削減を検討することが必要になった．多くの Krylov 部分空間法では行列の値は書き換えられないため，行列は倍精度としても丸め誤差の影響はない．これにより行列を DD でもつ場合と比べてメモリデータ量を半分にでき，疎行列ベクトル積のメモリへのデータ要求量を約 6 %削減できる．

疎行列を表現するためのデータ構造ではメモリ使用量を節約するために零要素を記憶しない．最も一般的な疎行列の格納形式である CRS（Compressed Row Storage）形式は，非零要素を格納する配列，各行の開始位置を示すインデックス配列，列番号を示すインデックス配列の 3 つで構成される．CRS 形式の SpMV や TSpMV はインデックス配列を利用したベクトルに対する間接参照，同時処理数が 4 の SIMD 命令によって 4 の倍数でない要素に対する処

理（以下，端数処理とよぶ），SIMD 演算の結果の足し合わせが必要になる．実測により，これらの処理が計算時間全体に占める割合は SpMV で約 60〜90 %，TSpMV で約 65〜89 %であることが明らかになった．これらの処理を CRS 形式の性能劣化要因とよび，性能劣化要因をなくす方法を検討した．

CRS 形式の性能劣化要因による性能低下を解消するため，4 つのデータを連続して処理できる格納形式である BCRS（Block CRS）形式に着目した．BCRS 形式はサイズ $r \times c$ の小密行列（以下，ブロックとよぶ）を CRS 形式のように行方向に格納する格納形式で，BCRS 形式はブロックの要素値を格納する配列，ブロック行の開始位置を示すインデックス配列，ブロック列の番号を示すインデックス配列の 3 つの配列で構成される．ブロック内の要素に対して連続したアクセスを行えるが，ブロック内に零要素を含めるため演算量やデータ量が最大で $r \times c$ 倍に増加する．

ブロックのサイズが SIMD の同時演算数である 4 ($r \times c = 4$) になるのはブロックサイズ 1x4，4x1，2x2 の 3 通りである．これらの特徴を比較検討した結果，CRS 形式の性能劣化要因を改善することのできるブロックサイズは 1x4，4x1 であることがわかった．非零パターンの異なる 100 問題に対して SpMV の性能を評価した結果，BCRS4x1 形式は CRS 形式と比べて平均で約 2.61 倍，BCRS1x4 と比べて約 1.55 倍高速という結果をえた．

BCRS4x1 形式の TSpMV は CRS 形式と比べて約 3.4 倍遅くなる．これは SIMD 演算の結果の足し合わせが 4 要素ごとに必要になるためである．BCRS4x1 形式のみで SpMV と TSpMV を高速に実行するため，BCRS4x1 に対しアクセスパターンを変更して列方向への並列化を行う BCRS (C) 方式を実装した．非零パターンの異なる 100 問題に対して TSpMV の性能を評価した結果，BCRS4x1 (C) 方式は CRS 形式と比べて平均で約 5.91 倍高速という結果をえた．

これらを核として倍精度型と DD 型を組み合わせて利用できるインタフェースを C++を用いて実装し，変数の宣言部を変更するだけでプログラム全体の計算精度を変更できるインタフェースを実現した．開発したソフトウェアを用いれば DD の BiCG 法を倍精度の約 1.1〜1.3 倍の時間で実行できた．

これらの実験から，SIMD 命令を用いた DD の疎行列演算おいてボトルネックとなるのは非連続なメモリアクセスや SIMD 命令で複数演算を同時に処理しなければならないことによって生じる処理である．これに対し SIMD 命令を効率よく利用するためには SIMD 命令の同時処理数に合わせてデータをフィッティングすることが非常に有効で，性能劣化要因をなくせるようにブロックサイズやアクセスパターンを変更することで高い性能が発揮できることがわかった．

本研究の今後の課題として，分散メモリ環境への適用を行っていくことや，今回実装した基本演算の実装をより多くの環境に適用させることを挙げた．

本研究では，現在一般的なプロセッサに搭載されている FMA 命令，SIMD 命令，マルチスレッドを用いた際の DD の疎行列演算の演算特性を明らかにし，これらに基づいた実装により高速が可能であることを示した．これらを扱うソフトウェアのインタフェースを構築したことで高速化された演算を少ないプログラムの変更で簡単に扱うことができる．本研究で提案した SIMD の同時処理数に合わせてデータをフィッティングすることで高速化を行うアプローチやそれらを利用するためのソフトウェアインタフェースは，汎用的かつ実用的な高精度 Krylov 部分空間法の実現を可能にできるという点で数値計算の基盤技術として有用である．

High Precision Sparse Matrix Software Accelerated by SIMD

Toshiaki Hishinuma

In physics simulations, it is often required to solve a system of linear equations concerning a large-scale sparse matrix. Generally, Krylov subspace methods are applied to solve such linear equations simultaneously. These methods are formulated based on a combination of the scalar, vector, and sparse matrix operations. Sparse matrix and vector product (SpMV), and transposed sparse matrix and vector product (TSpMV) are the most computationally expensive operations in these operations. Solvers based on the Krylov subspace methods may sometimes diverge, stagnate, and take more iterations owing to the rounding errors.

One possible way of improving convergence is to reduce the rounding errors using high-precision arithmetics for the entire program or particular operations. However, high-precision computations require considerable amount of computation time and memory space, and only a limited number of currently available software can be successfully applied to process high-precision arithmetics.

To perform high-precision arithmetic operations in realistic computation time, the author has focused on quadruple precision that is twice precise compared with double precision. Double-double precision (DD) arithmetic that corresponds to quadruple precision arithmetic, implies using the two double-precision variables from 10 to 20 double-precision operations for each arithmetic operation.

The purpose of this thesis is to accelerate the basic operations in DD with the purpose of implementing the Krylov subspace methods on a multi-core CPU. The author has utilized the fused multiply-add (FMA) instructions, single instruction multiple data streaming (SIMD) instructions, and multi-threading, and then, developed a tuned software corresponding the sparse matrix operations that can process double precision and DD in the same time.

DD benefits from a fast multiplication algorithm using the FMA instruction that can compute an addition for results of multiplication without a rounding error. The FMA instruction simultaneously allows performing multiplication and addition jointly. Therefore, the achieved performance can be twice better at most. The SIMD instructions apply the same operation to multiple data elements simultaneously. In current general-purpose processors, the maximum number of simultaneously processed operations corresponding to the SIMD instructions is four or eight. For example, by simultaneously processing the four operations on the four double-precision data, the total computation time can be reduced to 1/4. The performance is accordingly expected to improve four-fold. In

multi-threading, a process is divided into threads and then is assigned to particular processor cores for parallelization. The performance can be expected to improve proportionally to the number of cores.

DD operation requires the twice larger memory data space and from ten to twenty double-precision operations. The ratio between the amount of the data transfer from the memory to CPU and the data required for a floating-point operation (Byte/Flop (B/F)) in DD is much lower than that corresponding to the double-precision arithmetic. A DD operation differs from a double precision operation in terms of the breakdown of computation time (it is denoted as performance characteristics or bottlenecks). Acceleration strategies need to be determined based on performance characteristics. However, it is not clear what performance characteristics can be obtained, as the performance of DD operations using the SIMD instructions processed on four or eight double-precision operations simultaneously has not yet been investigated. There is no research dedicated to the acceleration of DD arithmetic using SIMD instructions and multi-threading. In this thesis, the author aims to clarify the performance characteristics of DD operations using the SIMD instructions and multi-threading. The author has proposed a novel method to use SIMD with the purpose of achieving high performance and speeding up the SpMV operations in DD.

The mixed-precision method employs high-precision arithmetic for a part of variables and operations. This method can be used to accelerate computation as it reduces number of double-precision operations, the amount of memory data, and memory access. However, the improvement of convergence is smaller than DD. Particularly, while applying high-precision arithmetic, there is a need to identify a trade-off between the improvement in the number of iterations and an increase in computation time. Generally, a fast and low-scale computation in low precision and a high-scale computation in high precision is deemed a considerable trade-off. Selection of the variables and operations to be executed in the high-precision depends on the properties of the considered problems and algorithms. Focusing on such a usage, it is necessary that an interface can switch the desired precision with a small rewriting effort.

The author has formulated the following three requirements for software and investigated how to speed up these core operations.

Requirement 1 Implementation of scalar, vector, and sparse matrix operations is required for the Krylov subspace method.

Requirement 2 Scalar, vector, and sparse matrix operations in DD are accelerated on a multi-core CPU.

Requirement 3 Implementation of an interface that can use a combination of double precision and DD is desirable.

It should be noted that performance estimates of vector operations accelerated by multi-threading are bounded by the memory access speed. It means that the vector operations have relatively small computation resources for each data. As SpMV and TSpMV require considerable computation time, it is necessary to consider reducing the memory access. To reduce the amount of the memory data and data transfer, the author has changed the data type of matrix from DD precision to double precision. The author assumes that there is no influence of rounding error even in the case of a double-precision matrix, as many Krylov subspace methods do not need updating matrix values. By setting the matrix to be double precision, the amount of the memory data can be reduced by a half compared with that of DD, and the amount of data transfer required from the memory to execute SpMV can be decreased by approximately 6 %.

Generally, a sparse matrix does not store zero elements. The compressed row storage (CRS) format that is the most common sparse matrix storage format, comprises the three arrays: the array that stores nonzero elements, index array that indicates the starting position of each row, and index array that indicates the column number.

SpMV and TSpMV in the CRS format require indirect access to vector elements using an index array. As the number of vector elements to be processed by SIMD instructions is multiple of four or eight, in particular cases, the processing of the remainder in each row is required. Ordinary operations are needed to add four or eight elements in the SIMD register in each row. As a result, these processes consume 60 % – 90 % of the total computation time in SpMV and 65 % – 89 % in TSpMV. The author denotes these processes as performance degradation factors.

To eliminate the performance degradation factors, the author has focused on the block CRS (BCRS) format that is one of the storage formats that creates a set of size $r \times c$, small and dense matrices (denoted as a block), including zero elements. The BCRS format comprises the three arrays: the array that stores the values of the blocks, index array that indicates the starting position of the block row, and index array that indicates the number of the block columns. The blocks with all zero elements are not stored. Herein, $r \times c$ data in a block can be processed continuously. As the blocks of BCRS includes zero elements, the number of computations and amount of data increases up to $r \times c$ times.

Considering that the maximum number of simultaneous operations of SIMD is four, there are three choices of block sizes: BCRS1x4, BCRS4x1, and BCRS2x2, in which the block size becomes

four ($r \times c = 4$). BCRS1x4 and BCRS4x1 can eliminate the performance degradation factors. The performance of BCRS4x1 was 2.61 times better in terms of the computation speed compared with that of the CRS format.

To improve the performance of TSpMV in BCRS4x1, the author has proposed the BCRS4x1 (C) method that restricts column access for column-wise multi-threading. The performance of BCRS4x1 format was 5.91 times better in terms of the computation speed than compared with that of the CRS format.

The author has formulated the basic operations using these implementations as a kernel, an interface that can be used in combination with double-precision, and DD that are implemented using C++ templates. As a result, an interface that can change the arithmetic accuracy of the entire program by merely changing the declaration part of the variables has been realized. The computation time required for the biconjugate gradient method in DD using the proposed software is 1.1 to 1.3 times compared with that of double precision. The proposed approach is aimed to process four elements simultaneously using SIMD instructions. This is achieved by using the BCRS4x1 format with the block size fitted to the SIMD register. Experimental results confirm that the problem of the CRS format can be improved.

Future works related to this study will be focused on applying the implemented basic operations to a distributed memory environment, and this new implementation to many environments.

In this thesis, the author aims to clarify a bottleneck of a sparse matrix operation in DD using the FMA and SIMD instructions, and multi-threading that are currently installed in general processors. The author demonstrates that the proposed approach enables the high-performance implementation to be resolving the bottlenecks. By defining a software interface to handle them appropriately, the accelerated operations can be processed easily requiring only several program changes. In summary, the proposed approach of speeding up the computations by fitting the format of matrix data to the SIMD register and applying a suitable software interface to process them can be used to realize the practical high-precision Krylov subspace method.

目 次

表 目 次

図 目 次

第1章　序論

1.1　研究背景，目的

物理シミュレーションの核は物理現象を記述する偏微分方程式などを差分法や有限要素法などを用いて離散化することによって得られる疎行列を係数行列とする連立一次方程式の求解である．疎行列は解析領域を格子分割するなどによって得られる．解析領域を細かく分割することによって離散化誤差を減らし精度の高いシミュレーションを行えるが，行列のサイズは大きくなり求解にかかる時間が増大する．

一般的に離散化により得られる疎行列は格子点数に依存した次元数および各格子点の節点間の接続に依存した非零要素をもつ．解析領域を細かく分割しても接続関係は大きく変わらないため 1 行あたりの非零要素数はほとんど変化せず，大規模なシミュレーションでは非常に疎な行列が現れる [1]．そのため一般的に疎行列の表現にはメモリ使用量を節約するために行番号や列番号のインデックス配列を用いて零要素を記憶しない格納形式を用いる [2]．疎行列を係数とする連立一次方程式解法では疎行列の値の書き換えや挿入が頻繁に行われる LU 分解などの直接解法でなく反復解法がよく使われる．

連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ に対する Krylov 部分空間法は，初期近似解 x_0 に対応する初期残差ベクトル $r_0 = \boldsymbol{b} - Ax_0$ を用いて A のべき乗と r_0 の積の像が張る空間から近似解を探索する反復法アルゴリズムの一つのカテゴリで，行列の性質に応じて様々なアルゴリズムがある [3]．行列をベクトルに作用させ，ベクトルの更新を繰り返すことで近似解をえる．ベクトルのみを更新するため疎行列を変更する必要がなく疎行列を扱いやすい [4]．一方で Krylov 部分空間法は丸め誤差により収束が遅くなったり，収束しなくなったりする [5]．

近年，計算機の発展に伴い計算機の演算性能やメモリサイズが増大し扱われる問題の規模が大きくなってきたことから丸め誤差による収束への影響も増大している [6]．

Krylov 部分空間法は行列の性質に合わせて前処理や解法を変更することで収束を改善するアプローチをとることが一般的である [4]．これはアルゴリズムに対する理解やプログラムの書き直しが必要になる．丸め誤差を減らして収束を改善する試みとして，プログラム全体や誤差の蓄積しやすい一部の演算を基準となる IEEE Std. 754-2008 の binary64（倍精度）[7] 演算よりも高い精度（以下，高精度）にすることで収束を改善する手法がある [8–11]．これは

アルゴリズムを変えることなく適用が可能であるが，計算時間やメモリデータ量が多く必要となることや標準的なプログラミング環境から高精度演算を行うためのソフトウェアが少ないことが問題となる．

現実的な時間で丸め誤差による収束への影響を低減するために倍精度の 2 倍の精度である 4 倍精度に着目する．ソフトウェアによる 4 倍精度演算の実現手法の 1 つに Double-Double 精度（DD）演算 [12] がある．DD 演算は 2 つの倍精度数を用いて 4 倍精度相当の演算を実行する手法で，倍精度演算 10〜20 回を組み合わせることで実行できる．

本研究はマルチコア CPU において Krylov 部分空間法の実装に必要な DD の演算を高速化することを目的とした．具体的な高速化のアプローチとして，代表的な Krylov 部分空間法の実装に必要な DD の演算を対象に現在のほとんどの汎用プロセッサでサポートされている FMA（Fused Multiply-Add）命令，SIMD（Single Instruction Multiple Data Streaming）命令，マルチスレッドを用いて高速化を行う．倍精度と DD を組みわせてこれらの演算を扱えるソフトウェアを開発する．

SIMD 命令は 1 命令で複数のデータへの同一演算を同時に実行する．現在の一般的な SIMD 命令の同時処理数は倍精度データ 4 または 8 つが標準的である．複数の倍精度データに対する演算を同時処理することで最大で同時処理数分の性能向上が期待できる．

DD の高速化を行う研究として，プロセッサのもつ並列化機構を用いた高速化技術を用いて DD の基本演算を高速化する研究 [13–15] や，メモリデータ量や計算量の削減のために倍精度型と DD 型を組み合わせて使う研究 [11, 16] が行われている．

文献 [15] では，マルチコア CPU において疎行列に対する DD の計算を同時処理数が 2 の SIMD 命令を用いて高速化している．一方で現在主流である同時処理数が 4 または 8 つの SIMD 命令を用いて疎行列に対する DD の計算を高速化している研究は確認されていない．

また，DD 演算は倍精度演算と比べて演算量が 10〜20 倍，メモリデータ量が 2 倍であるため演算数とメモリとのデータ転送量の比 （以下，Byte / Flop または B/F）が倍精度演算と比べて低く，倍精度演算とは演算にかかる時間の内訳（性能特性，ボトルネック）が異なる．高速化の方針は DD 演算の性能特性に応じて決める必要があるが，どのような性能特性になるかは明らかにされていない．

そこで本研究ではマルチコア CPU において Krylov 部分空間法の実装に必要な DD の演算を高速化する目的を達成するために，DD 演算の性能特性について評価し，DD の演算を高速に行うために性能特性に基づいたメモリレイアウトや FMA 命令，SIMD 命令，マルチスレッドの有効な利用方法を明らかにする．高速化した演算を核として倍精度型と DD 型を簡単に組み合わせて簡単に扱うためのソフトウェアのインタフェースを検討し，ユーザが簡単に倍精度型と DD 型を組み合わせて Krylov 部分空間法を実装できるソフトウェアを提供する．

1.2 研究の達成要件

本節では Krylov 部分空間法を実装するために必要な演算およびマルチコア CPU における高速化手法について述べ，研究目的の達成およびソフトウェアの開発のために必要な要件を定義する．

1.2.1 対象とする演算

Krylov 部分空間法は様々なアルゴリズムが提案されている．本論文では高速化の対象とする演算を文献 [17] に掲載されている，または Intel Math Kernel Library [18] に実装されている以下の Krylov 部分空間法に必要なものを選んだ．なお，BiCG 法のアルゴリズムは 5.6 節に載せた．

- CG 法（Conjugate Gradient method）
- BiCG 法（BiConjugate Gradient method）
- GMRES 法（Generalized Minimal RESidual method）

これらを実装するために必要なスカラ，ベクトル，疎行列に対する演算を表 1.1 にまとめた．このとき N × N の疎行列に対してベクトルは N 次元の列ベクトルで，疎行列に対してベクトルを左からかけるなどの存在しない演算は “N/A” とし，演算として存在するが使用されない場合は “none” とした．

必要な演算の中で，スカラとスカラの演算は他と比較して計算時間がほとんどかからない．スカラと疎行列の演算は使用頻度が少ない．疎行列と疎行列の演算は結果が密行列となってしまうため使われない．重要になるのはスカラとベクトルの演算，ベクトルとベクトルの演算，疎行列とベクトルの積である．本論文では，Krylov 部分空間法の実装に必要なこれらの演算を基本演算とよぶ．

表 1.1: Krylov 部分空間法における演算 (*x op y* を行う場合．*op* は和または積)

x \ *y*	スカラ	ベクトル	疎行列
スカラ	+, ×	×	×
ベクトル	N/A	+, ×	×
疎行列	N/A	N/A	none

必要な基本演算の中で最も時間がかかるのは疎行列ベクトル積（SpMV）および転置疎行列ベクトル積（TSpMV）である．本研究では SpMV と TSpMV を主なターゲットとした高速化を行う．

1.2.2　高速化の方針

CPU における高速化手法は次の 2 つに分けられる．

(1) 演算器内において 1 命令で複数の演算を同時実行することによる高速化

(2) コア単位での並列化による高速化

(1) は FMA 命令や SIMD 命令による高速化手法である．FMA 命令は積の結果を丸めずに和に利用することで精度よく計算できる積和演算命令で，DD 演算には FMA 命令を用いた演算量の少ない乗算アルゴリズムがある．積と和を 1 命令で行うため積と和がペアの計算であれば 2 倍の性能向上が期待できる．SIMD 命令は 1 命令で複数のデータを同時処理する命令である．現在の一般的なアーキテクチャにおける同時処理数は倍精度データ 4 または 8 つが標準的で，複数の倍精度データを格納できるレジスタ（SIMD レジスタ）に対する SIMD 用の演算，ロード，ストア命令を用いることで同時処理数分の性能向上が期待できる．

そこで本研究では，文献 [15] で用いられている SIMD 命令の 2 倍の同時処理数をもつ同時処理数 4 つの SIMD 命令を用いて基本演算の高速化を行う．また，将来的に同時処理数が 8 つ以上の SIMD 命令に対してソフトウェアの拡張を行うことを考え，ソフトウェアの設計においては SIMD 命令の同時処理数の変更によるプログラムの修正が少なくできるソフトウェアの構成について検討する．

(2) はマルチスレッドとよばれる手法をもちいて処理をスレッドという単位に分割する．スレッドを複数のコアに割り当てて並列実行すれば最大でコア数分の性能向上が期待できる．本論文ではスレッドはコアに対し最大 1 つ割り当てて利用し，スレッド数は使用するコア数および期待できる性能向上率と等しいようにした．

4.2 節で詳しく述べるが，DD のスカラ演算のアルゴリズムには並列性がほとんどない．そのため並列化はベクトルや疎行列の要素に対して並列化を行うことになる．このことからベクトルや疎行列の要素に対して並列処理を行うためにはメモリレイアウトや格納形式が重要になる．

一般的に疎行列を表現するためのデータ構造ではメモリ使用量を節約するために零要素を記憶しない．これは非零要素のみをもつ配列と，非零要素に対応する列番号や行番号のインデックス配列を用いる．そのため SpMV や TSpMV においてインデックス配列を用いたベク

トルへの間接参照が必要になり，SIMD 命令を用いて演算をまとめて行ううえではオーバヘッドになることが予想される．本研究ではこれらの高速化手法を DD の SpMV，TSpMV に適用した場合の性能特性について評価し，効率よく計算を行うための方法を明らかにする．

1.2.3 ソフトウェアのインタフェース

メモリデータ量の削減や経産時間の短縮を目的として，プログラム全体を高精度化せず，誤差が蓄積しやすい変数や演算のみを高精度化する混合精度手法とよばれる手法がある．変数や演算に対してどの精度を選択すれば収束が改善できるかは問題やアルゴリズムによって異なる．ユーザが混合精度手法を実装することを考えると，基本演算に対して倍精度と DD を組み合わせられるインタフェースが必要になる．

複数の精度を組み合わせた入力に対応するためには，FORTRAN や C などの線形代数ライブラリでよく用いられる “型名＋演算名” のような関数の命名法によるインタフェースは変数の精度を変更するたびに関数呼び出しを書き直さなければならないため非現実的である．ユーザが高速化，メモリ削減のために複数の精度を組み合わせるためには任意の型のデータを引数として与えられ，入力に対して自動で内部のルーチンを切り替えられる関数のインタフェースが必要である．本研究では倍精度型と DD 型を組み合わせて Krylov 部分空間法を実装するために精度の指定方法およびインタフェースについて検討する．

1.2.4 ソフトウェアの開発要件

前節までに述べた課題を整理する．本研究では高速化した DD の基本演算を用いて Krylov 部分空間法を実現するために次の 3 つの要件を定義した．

要件 1 Krylov 部分空間法に必要なスカラ，ベクトル，疎行列に対する演算があること

要件 2 スカラ，ベクトル，疎行列に対する演算がマルチコア CPU で高速に実行できること

要件 3 倍精度と DD の混合精度演算が実装できるインタフェースをもつこと

これらの要件を満たすことで，ユーザがマルチコア CPU において高速かつ高精度な Krylov 部分空間法を実装するための基本演算ソフトウェアを実現する．

1.3 本論文の構成

本論文は 6 つの章から構成される．第 1 章は本章であり，本研究の背景，目的，および意義について述べ，高速化の基本的な方針や開発すべきソフトウェアの要件について定義した．

第 2 章では関連研究として DD 演算の高速化に対する研究，疎行列とベクトルに対する演算を高速化している研究，Krylov 部分空間法に対して精度を変更することで収束を改善している研究を紹介することで本研究の立ち位置を明らかにする．

第 3 章では開発するソフトウェアの設計と戦略について考察する．ユーザが Krylov 部分空間法を実装する際にどのようなプログラミング言語でどのようなインタフェースを提供するべきかを議論し，それらを達成するために必要な機能を明らかにする．

第 4 章では DD 演算のアルゴリズムとマルチコア CPU における高速化手法を述べ，それらに基づいて DD 演算を高速化する手法を提案する．

第 5 章では 4 章で提案した手法の性能を示し，提案手法による高速化が有効であることを示す．

最後に第 6 章で本研究の結論として本研究の成果をまとめる．また，付録として次の 3 つを載せた．

A 開発したソフトウェアである DD-AVX v3 の機能一覧

B DD-AVX v3 を用いて実装した BiCG 法のプログラム

D GMP を用いたプログラムの自動生成

第2章　関連研究および関連ソフトウェア

はじめに高精度演算を用いて Krylov 部分空間法の収束改善を行っている研究を紹介する．Furuichi [8] らは悪条件なストークス流れの方程式を解くために問題の条件を変えて DD の GCR 法 [4] と MINRES 法 [19] を行っている．MINRES 法に対しては高精度化の効果はほとんど得られなかったが，GCR 法に対しては有効で倍精度では解けない悪条件な問題が解けることを示している．幸谷 [9] は CGS 法 [4]，BiCG 法，BiCGSTAB 法 [4]，GPBiCG 法 [20] において倍精度では解けない問題に対して精度を 512 bit から 8192 bit に変化させることで解けることを示している．

倍精度と DD を組み合わせた混合精度 Krylov 部分空間法に対する研究として，Yamazaki ら [16, 21] は高速な CA-GMRES 法や Cholesky 分解を高速に GPU で行うために，特定の変数のみに DD 型，残りは倍精度型を用いることで倍精度よりも高速に求解できることを示している．Saito ら [22] は GCR 法に対し特定の変数のみに DD，残りの変数は倍精度で計算を行うことでいくつかの問題に対してはすべて DD で行った場合と同様の反復回数で求解できることを示している．

これらの研究から，高精度演算を用いることで解けない問題を解くことができる場合があること，Krylov 部分空間法に対する高精度化は有効であるが高速化のためには問題に応じて解法や精度を使い分ける必要があることが読み取れる．

次に DD の基本演算に対する研究を紹介する．Bailey [12] の提案した DD 演算は Dekker [23] と Knuth [24] の丸め誤差のない倍精度加算と乗算のアルゴリズムに基づいて倍精度演算を 10〜20 回行うことで 4 倍精度演算を実現する手法である．DD 演算を利用できるソフトウェアとして，Hida らは DD のスカラ型を C++のクラスとして定義 [25] し，DD のスカラ型に対して四則演算や数学関数を実装することで，倍精度型などのプログラミング言語に標準で実装されているデータ型と同じインタフェースで扱うことができる QD ライブラリ [26] を開発している．また，MATLAB [27] や Julia [28] などの高機能な数値計算向けのプログラミング言語で QD ライブラリ相当の機能を提供しているソフトウェアもある [29, 30].

CPU 以外の環境で動作する DD 演算のソフトウェアとして，Lu らは DD のスカラ型に対する四則演算を GPU で実装 [31] している．Joldes らは DD のスカラ型に対する四則演算を GPU で実装 [32] し，CAMPARY [33] として公開している．

また，IBM XL FORTRAN compiler，IBM XL C compiler，gcc for the Power series (RS6000) はそれぞれ IBM の Power CPU 向けの Fortran, C, C++のコンパイラで，“long double” 型や “REAL*16” 型として DD のスカラ型を定義し，これに対する演算を実装 [34–37] している．

このように DD のスカラ型に対する算術演算はライブラリやコンパイラという形態で様々な環境で利用できる．一方，これらはスカラに対する算術演算のソフトウェアであるためベクトルや行列への演算は扱っておらず，SIMD 化やマルチスレッド化による高速化は行われていない．

密行列とベクトルに対する演算を行うソフトウェアに BLAS [38] がある．BLAS は単精度実数型，倍精度実数型，単精度虚数型，単精度虚数型をサポートした線形代数ソフトウェアで，ベクトルや行列の先頭アドレスと次元数を与えることで内積や行列積などを行う機能を提供している．BLAS は型ごとに関数名の頭文字が異なり，複数の精度を組み合わせて入力することはできない．BLAS に準拠したインタフェースのソフトウェアが数多く開発されており [18, 39]，それぞれ独自の手法で演算を最適化，並列化している．

高精度な BLAS の開発または BLAS に含まれる演算の一部を高速化する研究として，Li らは入力を倍精度とし，内部の計算を DD とした高精度な BLAS である XBLAS [40] を開発している．密行列に対する混合精度 Krylov 部分空間法を行うことで近似解の精度が向上したことを示している [41] が並列化は行っていない．中田は任意多倍長ライブラリ GMP [42], MPFR [43] と QD ライブラリを用いて高精度な BLAS, LAPACK ライブラリを MPACK [44] として開発している．BLAS に含まれる演算を DD で実装し，OpenMP [45] によるマルチスレッド化は有効 [46] で，半正定値計画問題を高速，高精度に解くことができたことが示されている [47]．中村らは OpenCL [48] というフレームワークを用いて DD の密行列と密行列の積に対して AVX を用いて高速化し，SIMD 化による高速化が有効であることを示している [49]．Yamada らは京コンピュータで動作する QPBLAS [50] を開発し，FMA 命令の利用やマルチスレッド化が有効であることを示している [51]．Mukunoki らは GPU 向けに BLAS に含まれる演算を DD で実装し，GPU による高速化が有効であることを示している [13]．

BLAS に含まれる演算を倍精度と DD を組み合わせて行った研究として，八木らは倍精度と DD のスカラ，ベクトル，密行列に対する演算を MATLAB [27] 向けに実装し，FMA 命令の利用，4 つの倍精度演算を同時実行できる SIMD 拡張命令 AVX2 [52] を用いた SIMD 化，OpenMP によるマルチスレッド化が有効であることを示している [14]．これらを MATLAB の演算子などを用いたインタフェースで扱えるようにしたライブラリとして MuPAT [53] を開発している．

これらの論文では DD のベクトルや密行列に対する DD の演算の高速化が有効であることが示されているが，疎行列は扱っていない．密行列に対する演算で最も時間がかかる行列と

行列の積はブロック化などの最適化手法を用いることでキャッシュにあるデータを再利用でき，メモリへのアクセスを減らせるため B/F を下げることができる．そのため行列と行列の積では倍精度，DD のどちらにおいてもメモリ性能はボトルネックになりにくい．例えば文献 [13, 51] の結果では，行列と行列の積において DD は倍精度と比べて 10〜20 倍の時間がかかっており，性能比が演算量の比とほぼ等しくなっていることが読み取れる．一方で本研究でターゲットとする SpMV や TSpMV は多くの場合キャッシュにあるデータを再利用しにくいため，行列と行列の積とは演算特性が異なると考えられる．

疎行列の格納形式は非零要素の配置やアーキテクチャに応じて様々なものが提案されている [2]．疎行列に対する演算を行うソフトウェアに Sparse BLAS [54] があるが，すべての格納形式に対して統一されたインタフェースを実装することは難しいため BLAS と違い標準的なインタフェースとして定着していない．例えば CPU 向けの Intel MKL [18] と GPU 向けの NVIDIA cuSPARSE [55] を比べてもインタフェースが大きく異なり，同一ソフトウェア内でも格納形式ごとに機能が異なっているケースもある．一方で SpMV や TSpMV が疎行列に対する主要な演算であるのはどのソフトウェアも同様である．

DD の疎行列に対する演算を高速化している研究として，小武守らは行列は倍精度，ベクトルは DD とした疎行列やベクトルに対する演算に対する同時処理数が 2 つの SIMD 拡張命令 SSE2 [52] による SIMD 化や OpenMP によるマルチスレッド化が有効であることを示し [15]，これらの基本演算を核として反復解法ライブラリ Lis [56] を開発している．これは同時処理数が 4 または 8 つの SIMD 命令による高速化は行っていない．また，Lis がユーザに提供しているのは反復解法のインタフェースのみで，行列やベクトルに対する基本演算の機能はユーザから利用できない．Mukunoki ら [57] は DD の SpMV を GPU で高速化し，Krylov 部分空間法に適用している [57] ．これは GPU 向けのものであり，CPU における高速化は行われていない．

DD の疎行列に対する混合精度演算を行っている研究として，Kikkawa ら [58] は Scilab [59] 向けの MuPAT [53] を開発している．これは倍精度と DD スカラ，ベクトル，密行列，疎行列型に対する演算を Scilab 上で行えるようにしているが，並列化などは行われていない．

倍精度の SpMV における疎行列の格納形式や高速化に対する研究として，Kotakemori らは疎行列の格納形式である CRS (Compressed Row Storage), BCRS (Block CRS), DIA (Diagonal) 形式 [2] に対する SpMV のマルチスレッド化による高速化を行い，疎行列の形状やスレッド数に応じて最適な格納形式を選択することで SpMV を高速化できたことを示している [60]. Im らはレジスタのサイズに合わせて疎行列をブロック化することで SpMV の高速化が行えたことを示している [61]．Buluç らは CPU での倍精度の SpMV におけるボトルネックはメモリ性能であるとし，SIMD の同時演算数に合わせて疎行列をブロック化することで，インデッ

クスの配列などのデータサイズを減らせるため高速化できることを示している [62]．Saule らはコプロセッサである Intel Xeon Phi に搭載されている同時処理数が 8 つの SIMD 拡張命令 AVX512 [52] を用いて CRS 形式の SpMV を高速化できることを示している [63]．また，特定の分野のシミュレーションに現れる疎行列に特化した最適化として，Morita らは有限要素法で用いられる疎行列が 3 × 3 のブロック構造で現れることを利用し，疎行列を 3 × 3 の BCRS 形式でもつことで構造解析に現れる疎行列に対する SpMV が京コンピュータで効率よく計算できることを示している [64]．これは有限要素法に基づく構造解析ソフトウェアである FrontISTR [65] に実装されている．

これらは倍精度に対するもので疎行列に対する DD の演算を同時処理数が 4 または 8 つの SIMD 命令を用いて高速化を行っている例は確認されていない．また，倍精度と DD を組み合わせて疎行列やベクトルに対する基本演算を実行できるソフトウェアは Scilab 向けの MuPAT 以外は確認されていない．

SpMV や TSpMV はキャッシュにあるデータを再利用しにくいため，一般的に倍精度ではメモリ性能がボトルネックになる [62]．DD の SpMV は倍精度の場合と比べて B/F が低いため性能特性が異なることが予想される．例えば文献 [63] では，CRS 形式の倍精度 SpMV が同時演算数が 8 つの AVX512 を用いて最大で 1.5 倍程度しか高速化できていない．一方で文献 [15] では，CRS 形式の DD の SpMV が同時処理数が 2 つの SSE2 を用いてほぼ 2 倍まで高速化できている．このように倍精度と DD では SIMD 化などによる高速化の効果が異なることが推測されるが，DD の SpMV がどのような性能特性になるかは明らかになっていない．

性能特性を得るためにはプロセッサやメモリのピーク性能に対してどの程度性能が得られているか，ピーク性能に達しない場合はどのような処理に時間がかかっているかを評価する必要があるが，文献 [15] における性能評価は主に実行時間と SIMD 化による性能向上率を対象としており，疎行列に対する DD の計算の性能特性が倍精度と比べてどのように異なるかについては触れられていない．また，マルチコア CPU において疎行列に対する倍精度と DD の混合精度演算を高速化している研究はなく，利用できるソフトウェアも確認されていない．

このことからマルチコア CPU における高速化手法である FMA 命令，SIMD 命令，マルチスレッドを用いた際の疎行列に対する DD の計算の性能特性を明らかにし，性能特性に基づいてマルチコア CPU の高速化手法の効率的な利用方法を提案することや，高速化された基本演算の関数への入力として倍精度型と DD 型を組み合わせて扱うことができるソフトウェアのインタフェースを定めることは，将来のアーキテクチャでも使える高速かつ高精度な物理シミュレーションを行うための基盤技術として有用である．

第3章　ソフトウェアの設計および戦略

本章ではユーザにソフトウェアをどのように提供するかについて検討する．本章で定めたインタフェースに基づいた内部実装の高速化については 4 章で述べる．

3.1　戦略

本節では高速化した基本演算をソフトウェアとしてまとめることを考え，近年の数値計算ソフトウェアの動向を述べたうえで倍精度と DD を組み合わせて使えるインタフェースの実現に必要なことを議論する．

図 3.1 に，本研究で採用したソフトウェアの構成を示す．ソフトウェアの構成についてそれぞれ次のように定義した．

フロントエンド ユーザが倍精度型や DD 型に対する基本演算を行うためのインタフェース

バックエンド 基本演算が実装されたプログラム

カーネル関数 基本演算の核となる倍精度型と DD 型の四則演算などのプログラム

一般的に最適化や並列化はプログラムのハードウェア依存性を高める．特に SIMD 命令はハードウェアの命令セットに依存する．ハードウェア依存性は避けられないが，ソフトウェアを階層化することでハードウェア依存を限定的なものにすることは可能である．そのため SIMD 命令を用いて高速化された倍精度型と DD 型の四則演算などをカーネル関数（または SIMD カーネル関数）として実装し，バックエンドはこれらを呼び出すことでハードウェアに依存しないようにする．カーネル演算では SIMD 命令の同時演算数が異なる場合についてそれぞれ実装し，マクロ変数等で切り替えられるようにする．カーネル関数の詳細は 4.3.2 節で述べ，本章ではフロントエンド，バックエンドについて検討する．

近年，数値計算に多く使われる C, C++, Fortran などの型付けやメモリ管理に厳密なプログラミング言語で実装された高速なプログラムをバックエンドとし，低速だが高機能なインタプリタ型のプログラミング言語からバックエンドを呼び出して利用するケースが増加している [66, 67]．たとえばインタプリタ型のプログラミング言語である Python [68] 向けに開発さ

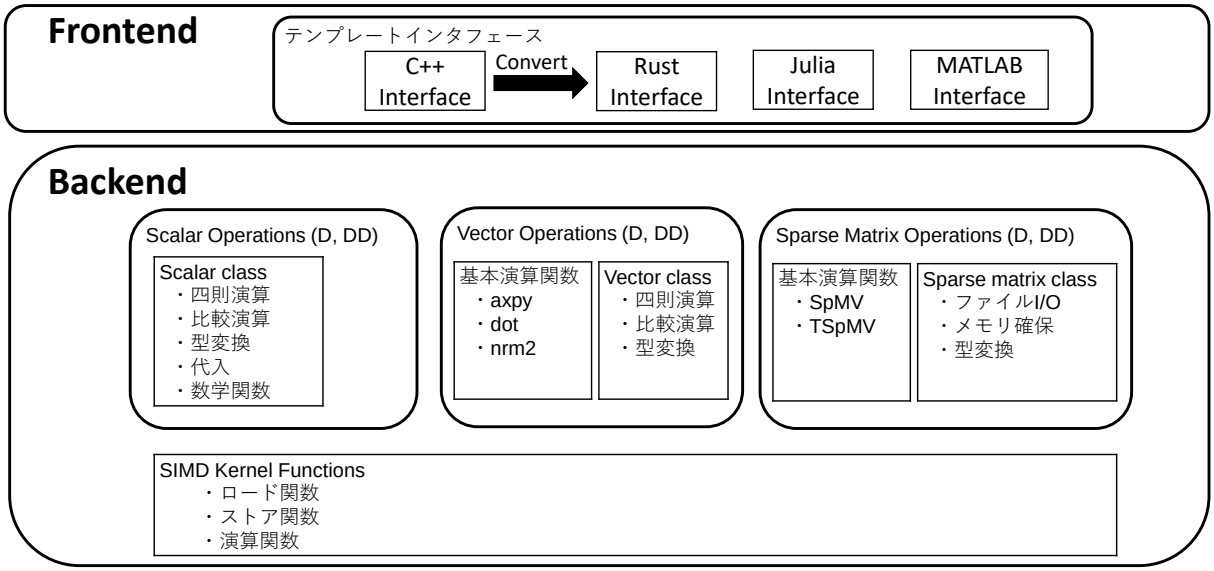

図 3.1: ソフトウェアの構成

れた Numpy [69] や Scipy [70] では，Fortran で作られた BLAS や Sparse BLAS をバックエンドとして呼び出せる．

また，数値計算に特化した MATLAB [27] や Julia [28] のような言語ではベクトルや行列の型が定義されている．Julia では行列型 A, B に対して “A * B” と書くだけで BLAS の行列積が呼び出されるなど，行列やベクトルを簡単に扱うことができる機能をサポートしている．また，Julia は言語仕様に BLAS を含んでおり行列やベクトルに対する計算のバックエンドとして BLAS を呼び出す．

本研究において主眼となる基本演算の高速化を実装するのはバックエンドおよびカーネル関数である．バックエンドは高速な C, C++, Fortran などで実装する必要があると考えられるが，フロントエンドを実装するプログラミング言語について考える必要がある．

Rust, Julia, MATLAB では従来の C, C++, Fortran などのライブラリを簡単に利用するための次のような機能を提供している．

- Rust は C, C++, Fortran のヘッダファイルから自動的にインタフェースを生成できる [71].
- Julia は C, C++, Fortran のライブラリをリンクして呼び出せる．
- MATLAB は MATLAB 用のヘッダファイルを新たに作成すれば C, C++, Fortran の関数を呼び出せる．

Rust, Julia, MATLAB にはベクトル型や疎行列型が実装されている．これは内部でオブジェ

クト指向のクラスとして実装されており，疎行列やベクトルとの演算やベクトル長の取得や動的な要素の削除や挿入などの基本操作をメンバ関数として実装しており，疎行列やベクトルを簡単に扱うことができる．そこで本論文ではターゲットとするフロントエンドとして想定するプログラミング言語を Rust, Julia, MATLAB の 3 つに定めた．

開発したソフトウェアをバックエンドとし，Rust, Julia, MATLAB の呼び出して使うことを考える．開発したバックエンドの機能やインタフェースが Rust, Julia, MATLAB のベクトル型や疎行列型のもつ機能と大きく異なる場合，フロントエンドの機能と組み合わせたり紐付けたりすることが困難になる．そのためフロントエンドのベクトル型を参考に DD のベクトル，疎行列クラスを設計することで，フロントエンドのベクトル型や疎行列型と同様のインタフェースで DD のベクトル，疎行列型を利用できるようにする．

倍精度と DD を組み合わせた混合精度のインタフェースを考えると，例えば BLAS のような C, Fortran から使われることを前提とした “型名+演算名” の関数の命名規則によるインタフェースの場合は変数の精度を変更するごとに関数名を変更しなければならないため現実的ではなく，フロントエンドからの利用も難しいと考えられる．Lis [56] では，倍精度や DD の型が独自に定義されており，“set_option()” という関数に “double” または “quad” を指定することで呼び出される Krylov 部分空間法全体の精度を変更できる．一方で複数の精度の組み合わせには対応していない．精度を設定する関数を用意して精度を切り替える方法はプログラム全体を変えずに精度を変更できるが，変数宣言と精度の指定を別に行わなければならないため，変数と型の関係がわかりにくい．そこで C++などのオブジェクト指向型の言語であれば，多重定義（オーバーロード）やテンプレートを用いることで同名の関数に対して複数の実装を定義できるため，宣言部分を変更するだけでプログラム全体の精度を変えられると考えた．

そこで本研究では，バックエンドの開発にオブジェクト指向プログラミング言語である C++を用いる．C++のオーバーロードやテンプレート機能を用いることで倍精度型と DD 型を組み合わせて利用できるバックエンドを開発する．要素の参照やコピーなどの基本操作機能をもつ倍精度および DD 型のスカラ，ベクトル，疎行列クラスを定義することで，高機能なクラスおよび型の宣言を変更するだけでプログラム全体の精度を変更できる基本演算のインタフェースを実現する．バックエンドのクラスに実装する機能は Rust, Julia, MATLAB のベクトルや疎行列型に合わせることで，変換を行った際にフロントエンドの機能との紐付けを容易にできるようにする．また，具体的なクラスや関数の一覧は付録 A に載せた．

3.2　スカラ，ベクトル，疎行列クラスの設計

本節では Rust, Julia, MATLAB の機能を参考に倍精度および DD 型のスカラ，ベクトル，疎行列クラスに必要な機能について検討する．Rust, Julia, MATLAB のベクトル，疎行列クラスがもつ機能をすべて実装することは難しい．Krylov 部分空間法の実装，解くべき疎行列の生成などに必要な機能について検討し，実装するべき機能を定義する．

3.2.1　スカラクラス（dd_real）

C++, Rust, Julia, MATLAB では倍精度のスカラ型は言語標準として利用でき，機能にほとんど差がない．そのため倍精度のスカラ型と同等の機能をもつ DD のスカラ型の実現について考える．

スカラ型は C++の倍精度型である “double” と組み合わせて同様の使い方ができることが望ましい．DD のスカラクラスとして “dd_real” を定義し，これのメンバ関数として Krylov 部分空間法に必要な次の機能を実装する．

- 倍精度型および DD 型のスカラに対する四則演算子を用いた演算
- “=” 演算子を用いた代入
- 比較演算子を用いた値の比較
- “(double)” や “(dd_real)” を用いた倍精度型と DD 型の間の型変換
- “sqrt()” などの数学関数
- 標準出力およびファイルへの入出力

3.2.2　ベクトルクラス（d_real_vector, dd_real_vector）

C++には Rust, Julia, MATLAB のように配列を行列やベクトルとして扱える型はない．C++の標準ライブラリでは配列の代わりに “std::vector” というクラスを用意している [72]．これは C の配列の機能を拡張したもので，クラスのメンバ関数として次のような基本操作機能を提供している．

- コンストラクタ，デストラクタ関数によるメモリ確保，解放
- “=” 演算子を用いたベクトルのコピー

- “size()” 関数を用いたベクトル長の取得
- “at()” 関数を用いた要素の参照
- “insert()” 関数を用いた要素の挿入

Rust, Julia, MATLAB のようにベクトルを扱うため，倍精度ベクトルクラスとして d_real_vector，DD ベクトルクラスとして dd_real_vector を定義した．これは std::vector を継承し，Krylov 部分空間法でよく使われる操作を d_real_vector, dd_real_vector のメンバ関数として実装することにした．DD は配列長や挿入などに必要な操作が倍精度と異なるため，変更が必要な関数については独自に再実装し，機能を上書き（C++では特殊化とよぶ）する．これに対し Rust, Julia, MATLAB のベクトル型に実装されている次の機能を追加実装する．Krylov 部分空間法の実装であまり使われないべき乗演算，丸め演算，集合演算，ビット演算は対象外とした．

- ベクトルの各要素に対してベクトルの各要素を四則演算
- ベクトルの全要素に対してスカラを四則演算
- ベクトルの全要素が一致または不一致かを判定
- 符号反転

Krylov 部分空間法ではスカラ α，ベクトル $\boldsymbol{x}, \boldsymbol{y}$ に対する $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$ という演算がよく現れる．これはベクトルとスカラの四則演算だけで実装する場合は $\alpha\boldsymbol{x}$ の結果を一時的なベクトルに保存する必要がある．BLAS [38] では，内部で $\alpha\boldsymbol{x}$ の結果を中間変数に保持せずに $\boldsymbol{y}$ との和に用いるなど，よく使われる演算に対する最適化がされている．このことからよく使われる基本演算は関数としてまとめることにした．CG 法，BiCG 法，GMRES 法に必要なベクトルに対する基本演算の定義を表 3.1 に示す．これらを C++のテンプレート機能を用いて倍精度型，DD 型の組み合わせに対して同一の関数呼び出しインタフェースで使えるようにする．

また，今回対象としない密行列は多次元のベクトルとみなせるため，d_real_vector と dd_real_vector に対する拡張として実装できる．

3.2.3 疎行列クラス（d_real_SpMat, dd_real_SpMat）

C++の標準ライブラリには疎行列型はない．そこで倍精度疎行列クラスとして d_real_SpMat，DD 疎行列クラスとして dd_real_SpMat を定義した．疎行列に対し必要な演算は SpMV および TSpMV である．表 3.2 に SpMV, TSpMV の関数の定義を示す．

表 3.1: Krylov 部分空間法の実装に必要なベクトルに対する演算

関数名（引数）	演算内容
void axpy(α, x, y)	$\boldsymbol{y} = \alpha\boldsymbol{x}+\boldsymbol{y}$
void axpyz(α, x, y, z)	$\boldsymbol{z} = \alpha\boldsymbol{x}+\boldsymbol{y}$
void xpay(α, x, y)	$\boldsymbol{y} = \boldsymbol{x}+\alpha\boldsymbol{y}$
void dot(x, y, val)	val = $\boldsymbol{x} \cdot \boldsymbol{y}$
void nrm2(x, val)	val = $\|\|\boldsymbol{x}\|\|$
void scale(α, x)	$\boldsymbol{x} = \alpha\boldsymbol{x}$

NOTE: α と val は倍精度または DD のスカラ，$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ は倍精度または DD のベクトル．

表 3.2: Krylov 部分空間法の実装に必要な疎行列に対する演算

関数名（引数）	演算内容
void SpMV(A, x, y)	$\boldsymbol{y} = A\boldsymbol{x}$
void TSpMV(A, x, y)	$\boldsymbol{y} = A^T\boldsymbol{x}$

NOTE: A は疎行列クラス，$\boldsymbol{x}, \boldsymbol{y}$ はベクトルクラス．

MATLAB や Julia では独自の疎行列型が定義されており，密行列とほぼ同様に扱うことができる．一方で疎行列の表現には零要素を記憶しない形式を用いることから，密行列のように行番号，列番号を用いて扱うとユーザの想定以上に処理時間がかかる場合がある．例えば行や列をベクトルとして扱ったり入れ替える処理は疎行列では処理時間が多くかかるうえに，Krylov 部分空間法ではほとんど使われない．そこで MATLAB および Julia の疎行列型を参考に，Krylov 部分空間法で想定される疎行列の操作に必要な次の機能を実装する．Krylov 部分空間法の実装であまり使われないランダム行列に対する処理，密行列との演算や密行列への変換，行や列などの並び替えや置換は対象外とした．

- ファイル I/O
- コンストラクタ，デストラクタ関数によるメモリ確保，解放
- 疎行列のコピー
- “size()” 関数を用いた行列の行数の取得
- “nnz()” 関数を用いた非零要素数の取得
- 要素の参照，更新

これらの機能を C++のオーバーロードやテンプレート機能を用いて倍精度と DD を組み合わせた入力に対して実装することで，Rust, Julia, MATLAB から混合精度の Krylov 部分空間法を実装できるソフトウェアのインタフェースを実現する．

第4章　DD演算の実装と高速化

本章では，はじめに 4.1 節でマルチコア CPU における高速化手法および今回対象としたアーキテクチャの概要を紹介する．次に 4.2 節で DD 演算のアルゴリズムや特徴について紹介する．これらに基づきベクトル演算の実装について 4.3 節，疎行列とベクトルの演算の実装について 4.4 節で検討する．

4.1　マルチコア CPU における高速化

本節ではマルチコア CPU における高速化手法の概要および使用方法について述べる．

4.1.1　プロセッサの性能を決める要因

数値計算の分野では計算機の性能指標として HPL [73] ベンチマークがよく用いられる．HPL は密行列を係数とする線形方程式を解く時間に対するベンチマークで，高速化された倍精度行列と倍精度行列の積（DGEMM）が用いられる．

DGEMM の核は倍精度の積和演算である．そのためプロセッサの性能は 1 秒間に倍精度の積または和の浮動小数点演算を何回実行できるか（Floating-Point Operations Per Second, FLOPS）で性能が比較され，多くのハードウェアベンダが実行回数を多くするための工夫をしてきた．プロセッサが理論上 1 秒間に何回計算できるかを理論性能（またはピーク性能）とよぶ．マルチコア CPU においてピーク性能を求める計算式に関係している要素は次の 3 つである．

1. 動作周波数
2. 浮動小数点演算ユニット（FPU）の同時演算数
3. コア数

ほとんどのプロセッサのピーク性能はこれらの値または有無によって決まる．すべてのプロセッサに存在する動作周波数を除けばマルチコア CPU におけるプロセッサの特徴を決める要因はコア数および FPU の同時演算数といえる．以降の節ではこれらの概要や一般的な利用方法について述べる．

4.1.2　FPU 内の並列化

FPU（Floating Point Unit）は浮動小数点数に対する処理を実行する演算器で，演算コア内に搭載されている．本論文では主に浮動小数点数に対する加算，乗算，積和を行うことのできるものを指す．演算器は命令発行ポートから命令を受け取ることでそれぞれの計算を行う．SIMD 命令や FMA 命令とよばれる命令を用いることで 1 命令で同時処理するデータ数を増加させることで計算時間を短縮することができる．それぞれについて以降で述べる．

FMA 命令

Fused Multiply-Add（FMA）命令は IEEE Std. 754-2008 [7] に規定された FMA 演算を行う積和演算の 1 つである．積和演算は $x \times y + z$ を行う演算で，FMA 演算は $x \times y$ の結果を丸めずにケチ表現を含む 106 bit の中間レジスタに保持し，加算に利用することで精度よく積和演算を行える [7]．これを 1 命令で行う命令を FMA 命令，実行する演算器を FMA 演算器とよぶ．なお，ケチ表現は正規化された仮数部の最上位ビットを省略した表現を意味する．

なお，FMA 演算は C99 や C++11 から “fma” という関数で提供されており [74] ハードウェアで FMA 命令が使えない場合はソフトウェアによる実装が利用できるが，ハードウェアによる実装と比べ 10〜20 倍の時間がかかるため，本論文では FMA 演算器が搭載されていることを前提とする．

積を行う命令と同じサイクル数で命令を実行できるため，DGEMM のような積と和の回数がほぼ等しい演算では 2 倍の性能が期待できる．FMA 命令を利用する方法は次の 3 つである．

1. アセンブリ言語を用いて FMA 命令を記述する
2. C99 や C++11 の “fma” 関数 [74] を利用する
3. コンパイル時にコンパイラに対し FMA 命令を自動生成するオプションを与える

SIMD 命令

SIMD（Single Instruction Streaming Multiple Data Streaming）は，1 命令で複数のデータに対して同時処理（SIMD 演算）を行う．SIMD 命令を用いて処理を並列化することを SIMD 化とよび，SIMD 命令で同時に処理することのできる数を SIMD 長とよぶ．

SIMD 命令を実行するための機構は一般的に複数のデータに対する演算を同時に実行できる演算器およびレジスタを用いて実装され，同時処理する複数のデータを格納できるレジスタ（SIMD レジスタ）にデータを格納し，専用の演算器（SIMD 演算器）に対して専用の命令

（SIMD 命令）を用いて複数のデータに対する同時処理を行う．SIMD 命令は SIMD レジスタに対するロード，ストア，演算用の命令群（命令セット）が型ごとに実装されている．ほとんどの SIMD 命令は倍精度などの命令と同じ時間で実行できるため，単位時間あたりのデータ処理量が向上することで SIMD 命令を用いない場合と比べて性能が同時処理数分だけ向上する．

SIMD 長が 4 の SIMD 命令を用いた SIMD 化の基本的なフローは次のようになる．

1. メモリから SIMD レジスタに対して連続な同一の型の 4 個のデータを読み込む命令（以下，ロード命令）を用いて SIMD レジスタに読み込み，
2. SIMD レジスタの 4 個のデータに対して 1 つの SIMD 命令を用いた演算を行い，
3. レジスタ内の 4 個の計算結果をメモリの連続した領域に格納する命令 （以下，ストア命令）を用いて格納する．

SIMD 命令の制約としてレジスタ内の複数のデータに対して 1 つの命令で一括して処理を行うため，レジスタ内の特定のデータのみ異なる処理を行ったり，異なる型のデータを SIMD レジスタに格納することはできない．また，異なる型のデータが格納された SIMD レジスタ同士の処理を行うことはできない．ロードやストアの対象となるデータが連続に配置されていない場合は “gather” や “scatter” とよばれる非連続領域からのデータ集約や拡散のための特殊な命令を用いるか，ロード命令やストア命令を用いるためにユーザがデータを連続した領域に再配置する必要があるため性能が低下する．

SIMD 命令を利用する方法は次の 4 つである．

1. アセンブリ言語を用いる
2. アセンブリ言語のニーモニックとほぼ 1 対 1 の組み込み関数を用いる
3. コンパイル時にコンパイラに対し SIMD 命令を自動生成するオプションを与える
4. OpenMP [75] などのフレームワークに実装されているコンパイラへの指示句を用いる

4.1.3 マルチコアによる並列化

マルチコアは 1 つのプロセッサ（またはチップ，パッケージともよばれる）内に複数のコアを搭載する技術である [76]．各コアは命令発行ユニットやレジスタ，演算器をもつ．

スレッドという単位に処理を分割し，各演算コアにスレッドを割り当てることで処理を高速化でき，均等に処理を分割できれば最大でコア数分の高速化が期待できる．本論文ではコ

アに対し最大 1 つのスレッドを割り当てることとし，複数のスレッドに処理を分割して複数のコアに処理を割り当てて並列化することをマルチスレッド化とよぶ．

マルチスレッド化を行う場合，OpenMP [45] や POSIX thread（pthread） [77] を用いるのが一般的である．pthread はスレッド生成，終了，スレッド間のデータのやりとりなどを制御することのできるライブラリである．OpenMP はマルチスレッド化を行うためのフレームワークで，プログラムの並列化箇所にコンパイラへの指示句を入れることでマルチスレッド化を行う．

4.1.4　計算機システムの Byte / Flop

計算機システムのメモリとプロセッサの性能比の指標として Byte / Flop（B/F）がある．B/F は次のように求められる．

$$B/F = \text{メモリの最大データ供給量 / プロセッサのピーク性能} \tag{4.1}$$

B/F は計算機の性能を示す絶対的な指標ではないが，アプリケーションの B/F と計算機システムの B/F を比較することでアプリケーションの性能上限を概算したり，どの程度高速化の機構を利用したプログラムを開発すべきか検討できる．

これまで述べた処理の高速化技術の利用により単位時間あたりの計算能力が向上するが，演算器が計算をするために必要なデータ量が増加する．

ほとんどの環境ではプロセッサの計算能力に対してメモリのデータ供給性能は低い．そのため計算対象のデータをキャッシュを用いて再利用できない演算を行う場合はメモリからのデータ待ちが発生し，性能はメモリ性能に制約を受ける．計算に必要なデータ量を少なくしたり，キャッシュにあるデータを再利用することでメモリへのデータ要求量を削減することが重要になる．

4.1.5　対象とするアーキテクチャ

Haswell の概要

本研究では高速化を行う対象としてマルチコア CPU で SIMD 長が 256 bit の SIMD 演算器および FMA 演算器が搭載されている Intel Haswell アーキテクチャ（以下，Haswell） [78] を用いる．

はじめに Haswell の構成を図 4.1 に示す．Haswell は 256 bit の SIMD 命令が使える FMA 演算器を 2 基搭載し，倍精度データ 4 つに対する積，和，FMA 演算を 1 命令で同時実行できる．

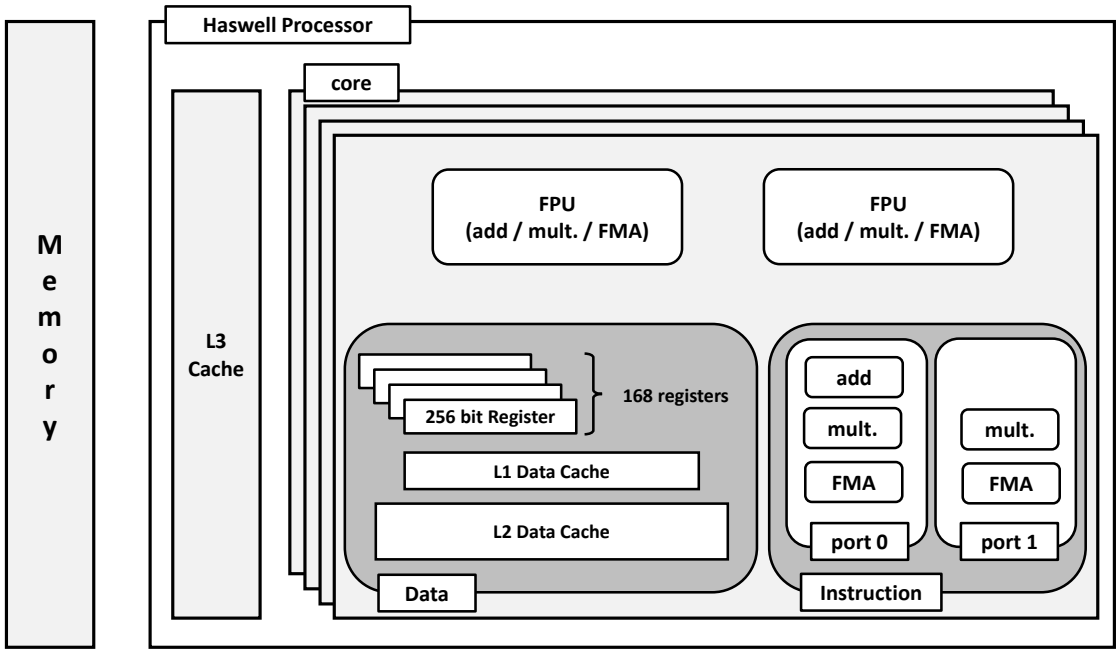

図 4.1: Haswell の構成

表 4.1: Haswell で使える SIMD 命令

	Year	ベクトル長（bit）	オペランド数	FMA 命令の利用可否
SSE2	2000-	128	2	No
AVX	2011-	256	3	No
AVX2	2013-	256	3	Yes

2〜18 のコアをもち，3 階層キャッシュモデルを採用している．各コアに搭載されている L1, L2 キャッシュとすべてのコアで共有の L3 キャッシュである．

Haswell の SIMD 命令

Haswell で使える主な SIMD の命令セットは次の 3 種類である．

- SSE2 (Streaming SIMD Extensions 2)
- AVX (Advanced Vector Extensions)
- AVX2 (Advanced Vector Extensions 2)

これらの特徴を表 4.1 に示す．SSE2 は Pentium 4（2000 年） [79] から登場した 128 bit の SIMD 拡張命令で，128 bit の xmm レジスタとよばれる SIMD レジスタに対して，1 命令で 2 つの倍精度のデータに対して同時演算ができる．

AVX は Intel Sandy Bridge アーキテクチャ（2011 年） [80] から登場した SIMD 命令で，256 bit の ymm レジスタと呼ばれる SIMD レジスタに対する SIMD 演算が行えるため，1 命令で 4 つの倍精度のデータに対して同時演算ができ，SSE2 と比べて最大で 2 倍の性能が期待できる．

AVX2 は AVX に対して FMA 命令などを追加した命令セットで，FMA 演算を行うことにより AVX と比べて最大で 2 倍の性能が期待できる [52].

また，AVX, AVX2 では，VEX プリフィックスというプリフィックス方式が採用され，SIMD 命令を用いない場合や SSE2 を用いた場合は 2 オペランドの命令しか実行できないのに対し，3 または 4 オペランドの命令が実行でき，レジスタ退避，復元処理の記述を省ける．そのため，AVX や AVX2 は，SIMD 命令を用いない場合や SSE2 を用いた場合と比べてレジスタ退避，復元処理の削減によって同時処理数の増加以上の性能向上となる可能性がある．

AVX, AVX2 が使えるアーキテクチャでは一般的な 64 bit の浮動小数点数を格納するレジスタ，xmm レジスタ，ymm レジスタに対する命令が使えるが，これらは物理的に 3 種類のレジスタが搭載されているのではなく浮動小数点数を格納するレジスタと xmm レジスタがそれぞれ ymm レジスタの下位 64，128 bit を使用し，ymm レジスタに包括されるような設計となっている．このような ymm レジスタの設計から AVX, AVX2 命令には SSE2 にはなかった次の様な制約がある．

- 同一 ymm レジスタ内の上位と下位の 128 bit 境界を越えた処理を行うことはできない．
- AVX, AVX2 命令と SSE2 命令を同一コード内で使用するとき，xmm レジスタと ymm レジスタの論理的な整合性を取るために数十サイクルかけてレジスタの値がハードウェアによって自動的にメモリに退避させられる．

C++において AVX2 を用いる場合，アセンブリ言語のニーモニックとほぼ 1 対 1 対応の組み込み関数を使用することでアセンブリ言語をプログラムに埋め込むことなく AVX2 の命令を記述できる．これは “immintrin.h” [52] を読み込み，コンパイラに対し “-mavx2” オプションを指定することで利用できる．

“immintrin.h” では倍精度数 4 つを格納するための AVX2 のレジスタ型として “__m256d” を定義しており，これをロード，ストア，演算の関数を用いて扱う．なお，“__m256d” 型の末尾の “d” は倍精度を表す．表 4.2 に本論文で用いる immintrin.h の関数を示す．

これらの関数名のうち，“mm256” の部分がデータサイズを表現し，SSE2 であれば “mm128” になる．同様に関数の末尾部分の “pd” が演算範囲とデータ型を表現しており，1 文字目の “p”

表 4.2: immintrin.h のロード，ストア，演算の関数

Intrinsics	Description
v _mm256_add_pd(v, v)	SIMD レジスタの 4 つの倍精度数同士を加算
v _mm256_sub_pd(v, v)	SIMD レジスタの 4 つの倍精度数同士を減算
v _mm256_mul_pd(v, v)	SIMD レジスタの 4 つの倍精度数同士を乗算
v _mm256_fmadd_pd(v, v, v)	SIMD レジスタの 4 つの倍精度数同士を FMA 演算
v _mm256_load_pd(mem)	4 つの倍精度数を 1 つのメモリアドレスから始まる連続した領域から SIMD レジスタにロード
v _mm256_set_pd(d, d, d, d)	4 つの倍精度数を SIMD レジスタにロード
v _mm256_broadcast_sd(d, v)	1 つの倍精度数を SIMD レジスタのすべての要素にロード
v _mm256_store_pd(mem)	SIMD レジスタの要素を 1 つのメモリアドレスに連続にストア

NOTE: v は SIMD レジスタ型，mem はメモリアドレス, d は倍精度数.

はレジスタ内のデータすべてに対する演算を意味し，“s” の場合はひとつのデータに対する演算になる．2 文字目の “d” が倍精度を意味する．

_mm256_load_pd 関数はデータのメモリ配置が連続な場合に用い，ランダムアクセスが最大 1 回発生する可能性がある．

_mm256_set_pd 関数はデータのメモリ配置が非連続な場合に用いる．1 命令中に最大 4 回のランダムアクセスが発生する可能性がある．また，_mm256_set_pd 関数はハードウェア実装はされておらず_mm256_load_pd 関数の組み合わせに分解されて処理される [52].

_mm256_broadcast_sd 関数は 1 つの倍精度浮動小数点数を SIMD レジスタのすべての要素にロードする．ランダムアクセスが最大で 1 回発生する可能性がある．

_mm256_store_pd 関数は SIMD レジスタ内の 4 つの要素を連続したメモリアドレスにストアする．

また，_mm256_fmadd_pd 関数は 4 つのオペランドをもつが AVX2 の命令は 3 オペランドである．組み込み関数では利便性のために 4 つの引数が取れるようになっているが，コンパイル時に一時変数への値の退避やオペランドの並び替えが行われて 3 オペランドの命令が生成される．

これらの関数を用いて，AVX2 の SIMD レジスタ型である “__m256d” 型に対する操作，演算を記述する．

Haswell のピーク性能

例えば Intel Sandy Bridge アーキテクチャには浮動小数点数の加算，乗算の演算器がそれぞれ 1 つ，浮動小数点数の加算，乗算の命令発行ポートがそれぞれ 1 つずつ搭載されている．この場合，1 サイクルに加算 1 回，乗算 1 回，または加算と乗算を 1 回ずつのいずれかが実行できる [80]．なお，演算器に対する命令の割り当てはハードウェアによって自動的に行われるためプログラムを開発する際に意識する必要はない．

一方で，演算器の数に対して命令発行ポートが 1 サイクルに発行できる命令数が対応していない場合がある．例えば Haswell には図 4.1 のとおり次の 2 器の演算器が搭載されている．

0 FMA / Add / Multiply ポート

1 FMA / Multiply ポート

倍精度加算，倍精度乗算，倍精度 FMA 命令が発行できる命令発行ポート 0 と，倍精度乗算，倍精度 FMA 命令のみが発行できる命令発行ポート 1 という構成である，そのため倍精度乗算，倍精度 FMA 命令は 1 サイクルに 2 回実行できるが倍精度加算は 1 回しか実行できず，加算の多いプログラムの場合は性能が低下する [78]．Haswell において FMA 命令を用いて加算と乗算を必ずペアとして実行できる場合のピーク性能は次のように計算できる．

- SIMD を用いないとき：
  “動作周波数” × “コア数” × 2 (add + mult.)
- SSE2 のとき：
  “動作周波数” × “コア数” × 2 (SIMD) × 2 (add + mult.)
- AVX のとき：
  “動作周波数” × “コア数” × 4 (SIMD) × 2 (add + mult.)
- AVX2 のとき：
  “動作周波数” × “コア数” × 4 (SIMD) × 4 (FMA × 2)

命令発行ポートの構成から，加算のみを行う場合の性能は 1 / 4, 乗算命令のみを行う場合は 1 / 2, FMA 演算に置き換えることのできない依存性のある等しい数の加算，乗算を行う場合は 1 / 2 の性能になる．

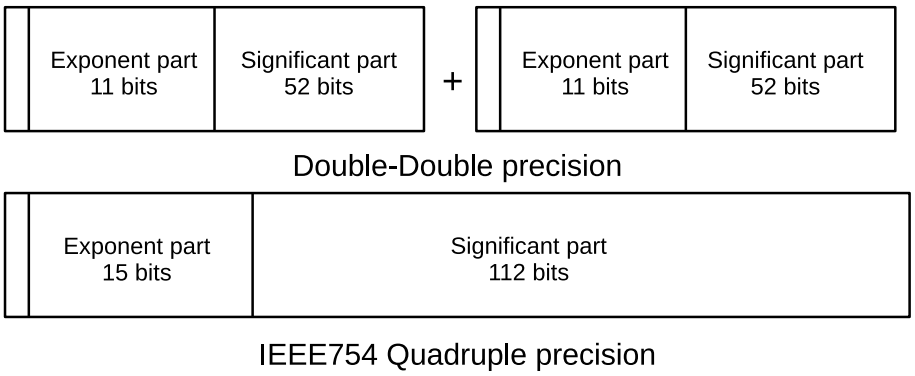

図 4.2: IEEE754 準拠の 4 倍精度型と DD 型

4.2 DD 演算

本節では，4.2.1 節で DD 演算のアルゴリズムについて紹介し，4.2.2 で DD のスカラ演算の実装について述べる．

4.2.1 DD 演算のアルゴリズム

DD 演算は Bailey が提案した “Double-Double” 精度のアルゴリズム [12] を用いて 4 倍精度演算を実行する手法である．DD 数を $a = (a_{hi}, a_{lo})$, $\frac{1}{2}\mathrm{ulp}(a_{hi}) \geq |a_{lo}|$（上位パート a_{hi} と下位パート a_{lo} は倍精度浮動小数点数）とし，倍精度浮動小数点数 2 つを用いて 4 倍精度を実現する．なお，ulp($\boldsymbol{x}$) は $\boldsymbol{x}$ の仮数部の “unit in the last place” を意味する．

DD の数は符号部 1 bit，指数部 11 bit，仮数部 104 (52×2) bit からなる．これは符号部 1 bit，指数部 15 bit，仮数部 112 bit で構成される IEEE754 準拠の 4 倍精度数と比べ指数部が 4 bit，仮数部が 8 bit 少ないが，IEEE754 準拠の 4 倍精度とくらべて高速である [10]．図 4.2 に DD の変数と IEEE 754 準拠の 4 倍精度の変数のデータ構造を示す．

次に DD 数に対する加算と乗算の実現方法について紹介する．2 つの倍精度数について加算および乗算を行うと一般に誤差が入るが倍精度数の範囲内では無誤差で評価可能という事実を用いる [23, 24, 26]. $\oplus, \ominus, \otimes$ をそれぞれ浮動小数点数同士の加算，減算，乗算とする．

加算 $s = a \oplus b$ およびその誤差 $e = a + b - (a \oplus b)$ について述べる．浮動小数点数の a, b について大小関係 $|a| \geq |b|$ がわかっているとき加算 $s = a \oplus b$ およびその誤差 $e = a + b - (a \oplus b)$ はアルゴリズム 1 の FAST-Two-Sum(a, b) を用いて評価できる．

アルゴリズム 1 FAST-Two-Sum($a,\ b$)

$s \leftarrow a \oplus b$

$e \leftarrow b \ominus (s \ominus a)$

return($s,\ e$)

大小関係が不明な場合はアルゴリズム 2 の Two-Sum($a,\ b$) を用いる.

アルゴリズム 2 Two-Sum($a,\ b$)

$s \leftarrow a \oplus b$

$v \leftarrow s \ominus a$

$e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$

return($s,\ e$)

次に乗算 $p = a \otimes b$ およびその誤差 $e = a \times b - (a \otimes b)$ はアルゴリズム 4 の Two-Prod($a,\ b$) で評価する. そこではアルゴリズム 3 の Split(a) を用いて浮動小数点数を無誤差で 2 つに分割する.

アルゴリズム 3 Split(a)

$t \leftarrow (2^{27} + 1) \otimes a$

$a_{hi} \leftarrow t \ominus (t \ominus a)$

$a_{lo} \leftarrow a \ominus a_{hi}$

return($a_{hi},\ a_{lo}$)

アルゴリズム 4 Two-Prod($a,\ b$)

$p \leftarrow a \otimes b$

$(a_{hi},\ a_{lo}) \leftarrow$ Split(a)

$(b_{hi},\ b_{lo}) \leftarrow$ Split(b)

$e \leftarrow ((a_{hi} \otimes b_{hi} \ominus p) \oplus a_{hi} \otimes b_{lo} \oplus a_{lo} \otimes b_{hi}) \oplus a_{lo} \otimes b_{lo}$

return($p,\ e$)

Two-Prod($a,\ b$) は FMA 命令が使える場合はアルゴリズム 5 の Two-Prod-FMA($a,\ b$) に書き換えられる. これは FMA 演算が $x \times y$ の結果をケチ表現を含む 106 bit の中間レジスタに保持して丸めを行わずに加算に利用できるため，乗算を誤差なしで行うことが保証されているためである [81, 82].

アルゴリズム 5 Two-Prod-FMA($a,\ b$)

$p \leftarrow -a \otimes b$

$e \leftarrow a \otimes b \oplus p$

$p \leftarrow -p$

return($p,\ e$)

これらを用いることで DD 数の加算と乗算を実現できる [26]. DD の加算 $c = a + b$ の結果を返す QuadAdd_Cray_Add($a,\ b$) をアルゴリズム 6 に DD の乗算 $c = a \times b$ の結果を返す QuadMul($a,\ b$) をアルゴリズム 7 に示す.

アルゴリズム 6 QuadAdd_Cray_Add($a,\ b$)

$(s_{hi},\ e_{hi})$ = Two-Sum(a_{hi}, b_{hi})

$e_{hi} = e_{hi} \oplus a_{lo} \oplus b_{lo}$

$(c_{hi},\ c_{lo})$ = FAST-Two-Sum$(s_{hi},\ e_{hi})$

return(**c**)

アルゴリズム 7 QuadMul($a,\ b$)

if (FMA 命令が使用できない)

$(p_{hi},\ p_{lo})$ = Two-Prod$(a_{hi},\ b_{hi})$

else

$(p_{hi},\ p_{lo})$ = Two-Prod-FMA$(a_{hi},\ b_{hi})$

endif

$p_{lo} = p_{lo} \oplus (a_{hi} \otimes b_{lo} \oplus a_{lo} \otimes b_{hi})$

$(c_{hi},\ c_{lo})$ = FAST-Two-Sum$(p_{hi},\ p_{lo})$

return(**c**)

このように DD の加算，乗算は倍精度数の四則演算の組み合わせのみで実現できる．なお，DD の加算において厳密に下位の誤差まで計算したい場合は下位パートに対する再計算が必要になる．下位の誤差を計算する方式を IEEE-Add 方式，計算しない方式は Cray-Add 方式とよばれ [25]，IEEE-Add 方式は倍精度演算が 20 回必要であるのに対し Cray-Add 方式は 11 回でよい．今回の実装では高速な DD 演算を目的としているため Cray-Add 方式を採用した．

```
1  void DD_ADD(double c[2], double a[2], double b[2])
2  {
3    double v;
4    double sh;
5    double eh;
6
7    // (sh, eh) = Two-Sum(a[0], b[0])
8    sh = a[0] + b[0];
9    v  = sh - a[0];
10   eh = (a[0] - (eh - v)) + (b[0] - v) ;
11
12   eh = eh + a[1] + b[1];
13
14   // (c[0], c[1]) = FAST-Two-Sum(sh, eh)
15   c[0] = sh + eh;
16   c[1] = eh - (c[0] - eh);
17 }
```

図 4.3: DD 加算（$c = a + b$）のプログラム

4.2.2　DD 演算の実装

DD 加算の実装

本節では DD 加算のプログラムの実装について述べる．図 4.3 に DD 加算; $c = a + b$ のプログラムを示す．ここで a, b, c は DD 型を意味する長さ 2 の倍精度配列で，配列の 0 番目が hi，1 番目が lo を意味する．

DD 加算は倍精度加減算からなり，倍精度演算の回数（演算量）は 11 Flops（floating-point operations）である．Flop は倍精度加減算または乗算を 1 Flop として数え，FMA 演算は 2 Flops として数える．また，Flop や Flops（Flop の複数形）は演算数，FLOPS (Floating-Point Operation Per Second) は性能の単位とし，本論文では大文字と小文字で区別して表記する．

x と y を倍精度数とし，$x + y$ の結果を $fl(x + y)$，丸めを $err(x + y)$ と表す．このとき $x + y = fl(x + y) + err(x + y)$ を満たす．これらは IEEE Std. 754-2008 [7] に定められた倍精度演算で round-to-even 丸めと仮定する．

図 4.3 の DD 加算（$c = a + b$）では，

- まず $sh = fl(a[0] + b[0])$ (L. 8)，$eh = err(a[0] + b[0])$ (LL. 9〜10) を求め，
- 次に a と b の下位と eh の加算を行うことにより，DD 加算 $a + b$ の近似 $sh + eh$ を得る

表 4.3: 混合精度加算（$c = a + b$）の演算量 [Flops]

c	a	b	演算量
Double	Double	Double	1
Double	Double	DD	8
Double	DD	DD	9
DD	Double	DD	10
DD	Double	Double	6
DD	DD	DD	11

(L. 12).

- 下位の足し合わせにより $\frac{1}{2}\mathrm{ulp}(sh) \geq |eh|$ とならない場合があるので，再度 sh と eh に LL. 9〜10 と同じ操作を行う．ここでは sh と eh の大小関係がわかっているため，計算量の少ないアルゴリズム (FAST-Two-Sum) を用いる (LL. 15〜16).

計算手順からわかる通り DD 加算は誤差を求める過程の計算順序が重要で，変数間の依存性が高く，並行化による高速化は期待できない．

次に DD 加算を倍精度と DD の混合精度で行う事を考える．倍精度と DD の混合精度加算の演算量を表 4.3 に示す．混合精度演算では倍精度変数とした変数の下位パートに関わる計算を省略できる．なお倍精度と倍精度の結果を DD に返す $c_{DD} = a_D + b_D$ は Two-Sum アルゴリズムを用いて内部の演算を高精度に行うようにした．

たとえば，倍精度変数 a_D，DD 変数 b_{DD}, c_{DD} に対する $c_{DD} = a_D + b_{DD}$ では図 4.3 の $a[1]$ に関わる計算が不要となるため，L. 12 の加算を削減でき演算量を 10 Flops にできる．

DD 乗算の実装

本節では DD 乗算のプログラムの実装について述べる．DD 乗算; $c = a * b$ のプログラムを図 4.4 に示す．前節同様に a, b, c は DD 変数で配列の 0 番目が hi, 1 番目が lo を意味する．DD 乗算は FMA 命令が使える計算環境では計算量の少ないアルゴリズム（Two-Prod-FMA アルゴリズム）が使える．“USE_FMA” は FMA 命令が使用可能かを表すマクロ変数で，“fma(x, y, z)” は FMA 命令を用いて $x \times y + z$ を行う関数であるとする．

DD 乗算（$c = a \times b$）は，

- まず $p0 = fl(a[0] \times b[0])$， $p1 = err(a[0] \times b[0])$ (LL. 10〜20) を求め，

```
1  double split = pow(2.0, 27) + 1; // 2^27+1
2  void SPLIT(double x, double h, double l){
3    double tmp;
4    tmp = split * x;
5    h = tmp - (tmp - x);
6    l = x - h;
7  }
8  void DD_MULT(double c[2], double a[2], double b[2]){
9    double p0, p1, ah, al, bh, bl;
10   if (USE_FMA == true){
11     p0 = -a[0] * b[0];
12     p1 = fma(a[0], b[0], p0);
13     p0 = -p0;
14   }
15   else{
16     p0 = a[0] * b[0];
17     SPLIT(a[0], ah, al);
18     SPLIT(b[0], bh, bl);
19     p1 = ((ah*bh-p0)+ah*bl+al*bh)+al*bl;
20   }
21
22   p1 = p1 + (a[0] * b[1]) + (a[1] * b[0]);
23
24   c[0] = p0 + p1;
25   c[1] = p1 - (c[0] - p1);
26 }
```

図 4.4: DD 乗算（$c = a \times b$）のプログラム

表 4.4: FMA 命令を用いない場合の混合精度乗算（$c = a \times b$）の演算量 [Flop]

c	a	b	演算量
Double	Double	Double	1
Double	Double	DD	21
Double	DD	DD	22
DD	Double	DD	23
DD	Double	Double	17
DD	DD	DD	24

- $p1 = fl(p1 + fl(a[0] \times b[1]) + fl(a[1] \times b[0])$ により DD 乗算 $a \times b$ の近似をえる (L. 22).
- 上記の加算により $\frac{1}{2}\mathrm{ulp}(p0) \geq |p1|$ とならない場合があるので，図 4.3 の LL. 13〜14 と同様の操作を行う (LL. 24〜25).

DD 乗算も DD 加算と同様に誤差を求める過程の計算順序が重要なため変数間の依存性が高く，高い並列性は期待できない．また，このアルゴリズムを SIMD 命令や FMA 命令を用いて実装することを考えると L. 22 を FMA 演算 2 回に置き換えられる．FMA 演算を 2 Flops と数え，符号反転を演算数に含まないとき DD_MULT の計算量は次のようになる．

- FMA 命令が使用できない場合は加減算 15 回，乗算 9 回 (24 Flops)
- FMA 命令が使用できる場合は加算 3 回，乗算 1 回，FMA 演算 3 回 (10 Flops)

次に倍精度型と DD 型の混合精度乗算を行う場合の演算量について述べる．表 4.4 に FMA 命令を用いない場合の演算量，表 4.5 に FMA 命令を用いる場合の演算量を示す．なお倍精度と倍精度の結果を DD に返す $c_{DD} = a_D \times b_D$ は，Two-Prod または Two-Prod-FMA アルゴリズムを用いて内部の演算を高精度に行うようにした．

4.2.3 DD 演算の Byte / Flop

DD 演算は倍精度演算と比べて B/F が低いという特徴がある．例えば倍精度加算 $c = a + b$ は 3 つの倍精度数 ($8 \times 3 = 24$ byte) のロードに対して演算量は 1 Flop で，B/F は 24 である．一方，DD 加算は 3 つの DD 数 ($16 \times 3 = 48$) に対するロードまたはストアが必要であるのに対して演算量は 11 Flops で，B/F は約 4.36 である．表 4.6 に，変数 a, b を加算または乗算した結果を変数 c に格納する場合の B/F を示す．

表 4.5: FMA 命令を用いる場合の混合精度乗算（$c = a \times b$）の演算量 [Flop]

c	a	b	演算量
Double	Double	Double	1
Double	Double	DD	7
Double	DD	DD	8
DD	Double	DD	9
DD	Double	Double	3
DD	DD	DD	10

表 4.6: 変数 a, b を加算または乗算した結果を c に格納する演算の B/F

c	a	b	加算	乗算（FMA なし）	乗算（FMA あり）
Double	Double	Double	24.0	24.0	24.0
Double	Double	DD	4.00	1.52	4.57
Double	DD	DD	4.44	1.82	5.00
DD	Double	DD	4.00	1.74	4.44
DD	Double	Double	5.33	1.88	10.6
DD	DD	DD	4.36	2.00	4.80

4.2.4 スカラ演算の実装

DD のアルゴリズムは SIMD 化やマルチスレッド化ができないため，FMA 命令を用いる以外に高速化はほとんどない．そのため図 4.3, 4.4 のプログラムをそのまま使うことになる．倍精度を基準とした DD スカラ演算の計算時間の比は演算量の比とほぼ等しくなると考えられる．

DD のスカラ演算ライブラリに QD ライブラリ [26] がある．QD ライブラリは Hida らが開発した C++で動作するライブラリで，DD 数を “dd_real” クラスとして実装しており演算子オーバーロードによってビット演算子を除いたすべての演算子を用いて DD のスカラ型と倍精度型に対する処理を行うことができる．

QD ライブラリはコンパイル時のオプションによって FMA 命令の有無を指定することで DD 乗算のアルゴリズムを切り替えられる．また，四則演算や算術演算をすべて C++の inline 関数で定義し，関数をコンパイル時に展開することで関数呼び出しのオーバヘッドがないように実装されている．

5 章において QD ライブラリの性能を評価し，倍精度との性能比が演算量の比とほぼ等しいことを確認できたため，スカラ演算については QD ライブラリを使用することにした．

4.3 ベクトル演算

本節ではベクトル演算の高速化方法と実装について述べる．

4.3.1 DD 配列のメモリ格納方法

DD 数を配列としてもった場合，次の 2 通りのメモリレイアウトが考えられる．

データ構造 1 *hi* と *lo* を交互に格納.

データ構造 2 *hi* と *lo* を別の配列に格納.

データ構造 1 は AoS（Array of Structure）型，データ構造 2 は SoA（Structure of Array）型とよばれる形式である [83]．AoS 型と SoA 型で DD 配列を作成した場合のメモリレイアウトを図 4.5 に示す．

AoS 型と SoA 型は複素数型や座標系のメモリレイアウトにおいてもよく議論されるメモリレイアウトである．これらはメモリへのアクセスパターンが異なるため実行効率や並列化効率に影響がある．どのようなアクセスを頻繁に行うことを想定するかによって使い分けられる [84].

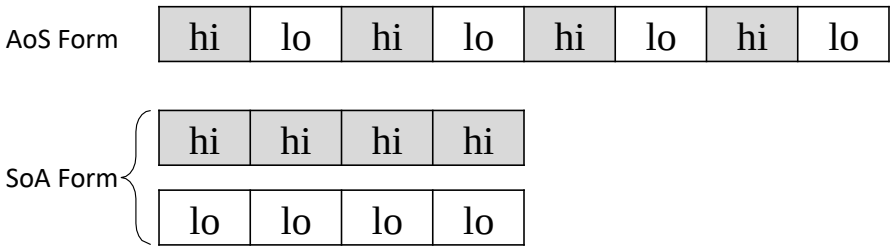

図 4.5: DD 配列のデータレイアウト

AoS 型は QD ライブラリの dd_real 型を配列として宣言すればプログラミング言語の機能で “[]” などを用いて簡単に要素にアクセスする機能を提供できる．SIMD 命令を用いて AoS の配列を処理することを考えたとき，AoS 型は load 命令を用いて連続した *hi* または *lo* へのアクセスができない．また，AoS 型は倍精度配列に変換を行うとき一時配列を確保して *hi* のみを取り出す処理が必要になる．

SoA 型は *hi* または *lo* に SIMD 命令を用いて連続してアクセスでき，SoA 型の倍精度への変換は *hi* のポインタのみを取り出して *lo* のメモリを解放するだけでよいためほとんど時間はかからない．一方で要素へのアクセスは 2 本の配列からそれぞれ対応する要素を取り出す必要があるため不向きである．本研究では Krylov 部分空間法に必要な基本演算に対して SIMD 命令を用いて高速化することが優先と考えて SoA 型を用いる．

4.3.2　ベクトル演算の実装

ベクトル演算に対して OpenMP を用いてマルチスレッド化し，各スレッドの処理を表 4.7 に示した関数を用いて SIMD 化する．ベクトル演算を SIMD 化する場合，倍精度では SIMD レジスタに対して連続な値を読み込んで演算を行う．DD 型では *hi* と *lo* に対応する SIMD レジスタを用意してそれぞれの連続な値を読み込んで演算を行う．

倍精度や DD 型を組み合わせてベクトル演算を行う関数に入力することを考えたとき，すべての引数の組み合わせについて実装するのは非現実的である．加算や乗算の実装は入力によって演算量が異なるためすべての組み合わせについて実装する必要があるが，内積や SpMV などの演算はそれらを組み合わせることで実装できると考え共通化を検討した．加算や乗算の関数をカーネル関数とよぶ．

バックエンドとカーネル演算を階層化して分離し，カーネル演算を SIMD 化することで，SIMD 命令を用いることによるプログラムのハードウェア依存を限定的なものにする．

$c = a + b$ を行う加算のカーネル関数の実装を考える．SIMD 化を行わない場合，a, b, c は

倍精度型または dd_real 型であるため C++のオーバーロード機能を用いてそれぞれの入力に対して実装を行えばよい．一方で SIMD 化を行う場合，a, b, c は 4 つのデータを格納した SIMD レジスタ型である必要がある．DD では hi, lo の値を格納する 2 本の SIMD レジスタ型が必要になるため引数の数が倍精度と異なる．倍精度と DD のカーネル関数の引数の数を同じにするため，SIMD レジスタ “__m256d” 型 1 つをメンバにもつ “D_SIMD_reg” および “__m256d” 型 2 つをメンバにもつ “DD_SIMD_reg” 構造体を定義した．C++のオーバーロード機能を用いて a, b, c にどちらの構造体を入力しても動作するようなカーネル関数を実装した．

DD では hi, lo それぞれに対してロードやストアを行わなければならないため，SIMD レジスタにデータをロード，ストアするカーネル関数も実装である．例えばロード用のカーネル関数は d_real_vector または dd_real_vector およびロード，ストアする配列の番号を入力することで “D_SIMD_reg” または “DD_SIMD_reg” に値をロードして出力するようにした．これらを SIMD レジスタ構造体とよぶ．これらを用いてカーネル関数を利用するベクトルや疎行列に対する基本演算の関数の実装を共通化する．

表 4.7 に実装したカーネル関数の一覧を示す．R1, R2, R3 は “D_SIMD_reg” または “DD_SIMD_reg”, V は d_real_vector または dd_real_vector, S1, S2, S3, S4 は DD スカラまたは倍精度スカラ，i は配列の要素番号に対応する int 型，N はベクトルの長さを表す int 型の変数である．これらの関数はすべて C++の inline 関数で定義した．

このようにソフトウェアを階層化したことで，異なる SIMD 命令に対応するために必要なプログラムの追加や修正をカーネル関数に集中させることができる．将来のアーキテクチャにおいて SIMD 長の異なる SIMD 命令に対応させることも容易であると考えられる．そのためソフトウェアの実装ではヘッダファイルに SIMD 長を設定するマクロ変数を定義し，コンパイル時に指定することで異なる SIMD 長に対するカーネル関数の実装を切り替えられるようにした．今回の実装では SIMD 命令を用いない場合は SIMD 長を 1 に設定し，AVX や AVX2 を用いる場合は 4 を設定する．将来的に SIMD 長が 8 の SIMD 命令に対応する場合は SIMD 長を設定するマクロ変数に 8 を指定した場合に対する処理を追加すればよい．ただし，SET 関数だけは引数の数が SIMD 命令の同時処理数になっているため SIMD 長が変わったときにはバックエンドの SET 関数の呼び出しを修正する必要がある．

図 4.6 にこれらのカーネル関数を用いて AVX2 の命令を用いた SIMD 化および OpenMP を用いた 4 スレッドのマルチスレッド化を行った倍精度 axpy の例を示す．

L.9 は OpenMP の指示句で，“{}” で囲まれたブロックをマルチスレッド化する．L.13 の関数 “get_start,” “get_end” は各スレッドの計算する領域の先頭と終了を返す補助関数，“size” 関数は d_real_vector のベクトル長を返すメンバ関数で，SIMD_LENGTH は SIMD の同時処理数を表すマクロ変数である．

表 4.7: カーネル関数の一覧

関数名	演算内容
void ADD(R1, R2, R3)	SIMD レジスタの加算 (R3 = R1 + R2)
void SUB(R1, R2, R3)	SIMD レジスタの減算 (R3 = R1 - R2)
void MUL(R1, R2, R3)	SIMD レジスタの乗算 (R3 = R1 × R2)
void ADDMUL(R1, R2, R3)	SIMD レジスタの積和演算 (R3 += R1 × R2)
R SET(S1, S2, S3, S4)	4 つの倍精度数または DD 数を格納したレジスタを返す (R = {S1, S2, S3, S4})
R LOAD(V, i)	V の i 番目から 4 つの連続した倍精度値, または i 番目, i+N 番目から DD 数を格納したレジスタを返す (R = {V[i], V[i+1], V[i+2], V[i+3]})
R BROADCAST(S)	1 つの倍精度数を R のすべての要素に格納
void STORE(R, V, i)	d_real_vector では SIMD レジスタの要素を 1 つの連続したメモリアドレスに格納 dd_real_vector では *hi*, *lo* の対応する連続したメモリアドレスに格納.

NOTE: R1, R2, R3 は SIMD レジスタ構造体，S1, S2, S3, S4 は倍精度型または DD スカラクラス，V は倍精度または DD ベクトルクラス，i は int 型.

```
1  #include "DD-AVX_AVX2_core.hpp"
2
3  void axpy(double alpha, d_real_vector x, d_real_vector y)
4  {
5      D_SIMD_reg alpha_reg;
6      D_SIMD_reg x_reg;
7      D_SIMD_reg y_reg;
8
9  #pragma omp parallel private(alpha_reg, x_reg, y_reg)
10   {
11     alpha_reg = BROADCAST(alpha);
12
13       for(int i = get_start(x.size()); i < get_end(x.size()) - (
             SIMD_LENGTH-1); i+=SIMD_LENGTH){
14         x_reg = LOAD(x, i);
15         y_reg = LOAD(y, i);
16
17         // y_reg = alpha_reg * x_reg + y_reg.
18         ADDMUL(alpha_reg, x_reg, y_reg);
19
20         STORE(y_reg, y, i);
21       }
22   }
23   //Processing remainder using QD Library.
24 }
```

図 4.6: AVX2, OpenMP を用いて SIMD 化，マルチスレッド化した倍精度 axpy

このプログラムでは次の手順で倍精度 axpy を実行する．

1. LL.5〜7 で α, x , y を格納するための AVX2 の SIMD レジスタ構造体の定義．
2. L.9 で LL.10〜24 のブロックをマルチスレッド化．“private()” 句によって AVX2 の SIMD レジスタ構造体の変数を各スレッドの独立した変数（ローカル変数）に指定．
3. L.11 で α を SIMD レジスタ型の $alpha\_reg$ のすべての要素に対して読込．
4. LL.14〜15 で $x[i]$, $y[i]$ から連続した 4 要素を対応する SIMD レジスタに格納．
5. L.18 で $alpha\_reg$, x_reg, y_reg に対する積和演算を実行．
6. L.20 で y_reg の計算結果を $y[i]$ から連続する 4 要素を格納．
7. これらの処理を AVX2 を用いて 4 つ同時に演算するため，L.13 で SIMD_LENGTH を用いて SIMD の同時処理数ずつインクリメントするループで繰り返す．

配列の長さが SIMD 長とスレッド数の倍数でない場合は SIMD 命令を用いて 4 つの要素を同時演算できないことによる余り（端数）に対する計算が L.23 で必要になる．ベクトル演算では端数が発生した場合は端数の数だけスカラ演算を行うことで端数の計算を行う．

このプログラムは L.3 の関数の引数および LL.5〜7 の “D_SIMD_reg” の宣言を “DD_SIMD_reg” に変更しても動作する．ソフトウェアにおける実装では，プログラムに入力された型への依存をなくすため，L.3 の関数の引数は C++のテンプレート機能を用いて共通化した．また，LL.5 7 に対しては入力された型を判定する “typeid()” という C++の標準関数を用いて関数に入力された型をそれぞれ調べ，対応する型の SIMD レジスタ構造体を宣言するようにした．この型を調べるための条件分岐はベクトルや行列に対する演算の関数の最初に 1 回行うだけであるため，演算と比べればごく短い時間であり性能に与える影響はほとんどない．これらの実装によって基本演算のプログラムから SIMD や型の依存性を減らし，実装を共通化しやすくなる．

4.4　疎行列とベクトルに対する演算

一般的に疎行列を表現するためのデータ表現ではメモリデータ量を節約するために零要素を記憶しない．疎行列の格納形式は非零要素の分布や利用するハードウェアの高速化技術に合わせて様々な形式が提案されてきた [2, 60, 61, 63]．本節では今回用いる格納形式について述べる．

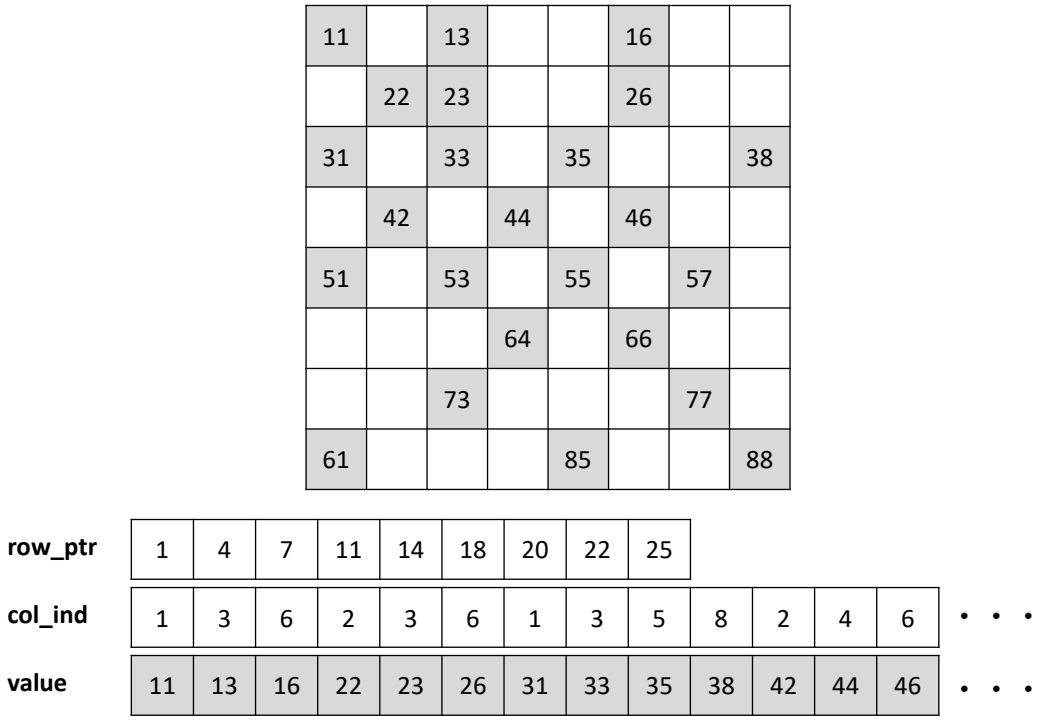

図 4.7: CRS（Compressed Row Storage）形式の構造

CRS（Compressed Row Storage）形式

最も一般的な圧縮格納形式に CRS（Compressed Row Storage）形式 [2] がある．図 4.7 に 8 × 8 の疎行列を CRS 形式で保持した例を示す.

CRS 形式は疎行列の非零要素のみを行方向に圧縮して格納する格納形式である．疎行列の行数（行列サイズ）を N，非零要素数を nnz (the Number of Non-Zeros) としたとき行列の非零要素を格納する配列 *value*，列インデックス配列 *column_index* (*col_ind*)，行ポインタ配列 *row_pointer* (*row_ptr*) の 3 つの配列で構成される．それぞれ次のような配列である．

1. 非零要素の値を格納する長さ nnz の倍精度型配列 *value*
2. 配列 *value* に格納された非零要素の列番号を格納する長さ nnz の整数型配列 *col_ind*
3. 各行の先頭非零要素が先頭から数えて何要素目にあるかを格納する長さ $N + 1$ の整数型配列 *row_ptr*

row_ptr の $N + 1$ 番目の要素に非零要素の数を入れることで非零要素の数を別の変数を用いて記憶しておく必要がなくなる．

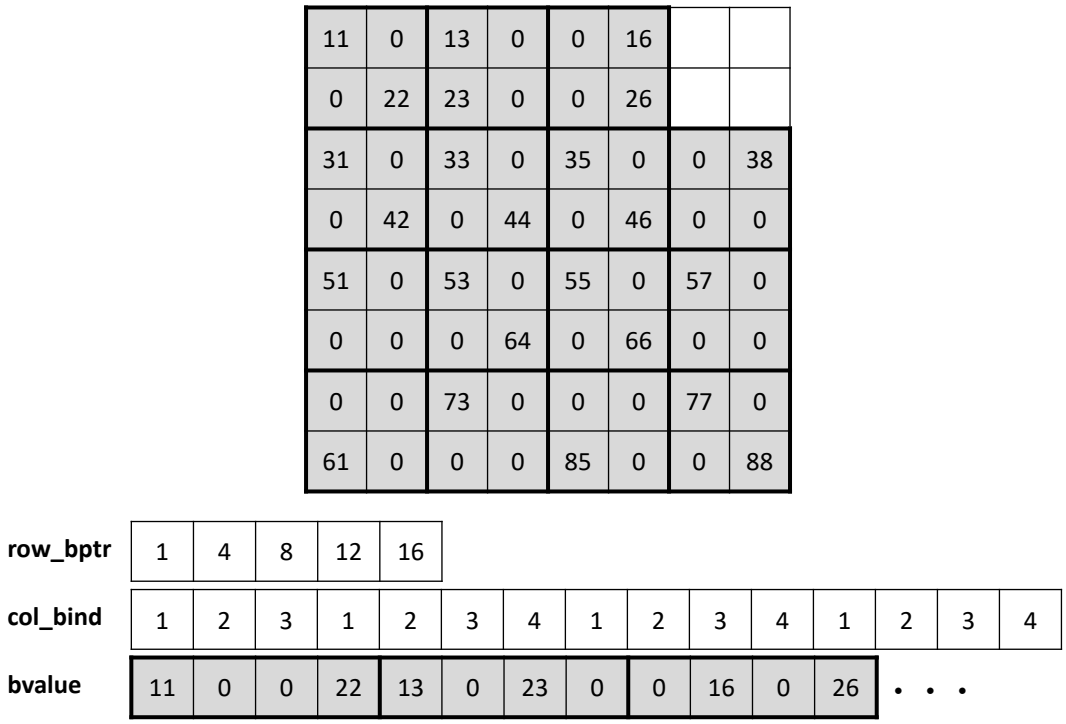

図 4.8: BCRS（Block CRS）形式の構造

BCRS（Block CRS）形式

次に Block Compressed Row Storage （BCRS）[2] 形式について述べる．BCRS 形式はサイズ $r \times c$ の小密行列 （ブロック）を CRS 形式のように格納する格納形式である．図 4.8 に 8×8 の疎行列を $r = 2, c = 2$ の BCRS（BCRS2x2）形式で保持した例を示す．

BCRS 形式はブロックの要素を格納する配列 *block_value* (*value*)，ブロック列のインデックス配列 *column_block_index* (*col_bind*)，ブロック行の先頭アドレスを示す配列 *row__block_pointer* (*row_bptr*) の 3 つで構成される．生成されるブロックの数を the number of blocks (*blk*)，行列の行数を N，r 行のまとまりをブロック行とよぶときそれぞれ次のような配列である．

1. ブロック内の要素の値を格納する長さ $blk \times r \times c$ の倍精度型配列 *bvalue*
2. 配列 *bvalue* に格納されたブロックの開始列番号を格納する長さ *blk* の整数型配列 *col_bind*
3. 各ブロック行の開始位置が配列 *col_bind* のどの要素から開始しているかを格納する長さ $(N/r) + 1$ の整数型配列 *row_bptr*

すべての要素が 0 となるブロックは作成しない．CRS 形式と比較して BCRS 形式を用いることの利点はブロック内で要素に連続してアクセスできるためキャッシュやメモリに対する

```
1  for(int bi=0;bi<r;bi++){
2    int i = bi*r;
3    int ii = 0;
4    int kk = Aout.bptr[bi];
5    while(i+ii<n && ii<=r-1){
6      for(k=Ain.row_ptr[i+ii]; k<Ain.row_ptr[i+ii+1]; k++]){
7        Aout.col_bind[kk] = Ain.col_ind[k] / c;
8        Aout.bvalue[Ain.row_ind[k] * r + ii] = Ain.value[k];
9        kk = kk + 1;
10     }
11     ii = ii+1;
12   }
13   Aout.row_bptr[bi]=kk;
14 }
```

図 4.9: CRS 形式からブロックサイズ $r \times c$ の BCRS 形式への変換

アクセスの改善効果が期待できること，インデックスを用いた間接参照を減らせること，インデックス配列に必要なメモリデータ量を CRS 形式と比べて減らせることである．一方でブロック内に零要素を含めるため疎行列ベクトル積の演算量や *value* 配列のメモリデータ量が最大で $r \times c$ 倍に増加する [85].

CRS 形式をブロックサイズが $r \times c$ の BCRS 形式に変換するアルゴリズムを図 4.9 に示す．このとき Ain は入力の CRS 形式の行列，Aout は出力の BCRS 形式の行列である．実装は SPARSEKIT [85] を参考にした．また，CRS 形式の行列の行数が r または c の倍数でない場合は零要素を詰めて r または c の倍数に揃えるようにした．

4.4.1 疎行列とベクトルの積（SpMV）; $\boldsymbol{y} = A\boldsymbol{x}$

SpMV の高速化方針

本節では疎行列とベクトルの積; $\boldsymbol{y} = A\boldsymbol{x}$ (Sparse matrix and vector multiplication, SpMV) の SIMD 化，マルチスレッド化について述べる．

CRS 形式では行単位で処理を分割することでスレッドに割り当て，各スレッドは行の要素に対して AVX2 を用いて 4 つずつ処理を行う．SIMD 命令は SIMD 長に応じた複数の要素を必ず同時演算しなければならないことから，AVX や AVX2 では各行で 4 つずつ処理した余り $\{1, 2, 3\}$が発生する可能性がある．このような余りを端数とよび，端数に対する計算を行うことを端数処理とよぶ．

また，各行の値を 4 つずつ計算すると行のすべての要素の計算後にベクトル $\boldsymbol{y}$ の対応する要素に SIMD レジスタ内の 4 つの要素を足し合わせて格納する必要がある．これは入力が DD であれば DD 加算を用いて足し合わせを行う．

これらの処理のために次の 2 つの補助関数を実装した．

reduction 関数　SIMD レジスタ内の要素を ADD 関数を用いて結果に足し込む

fraction_processing 関数　AVX2 において各行で発生する端数{1, 2, 3}を計算

reduction 関数は ADD 関数を 3 回用いてトーナメント方式で SIMD レジスタ内の 4 つの要素を足し込む．このときの DD における演算量は倍精度加減算 $11 \times 3 = 33$ Flops である．

fraction_processing 関数は端数処理を行う補助関数である．端数の数を r としたとき端数処理の方法として次の 3 つを考えた．

1. “Padding in execution” 方式: SET 関数の引数に対し計算する要素数が 4 になるように 0 を代入し，ADDMUL 関数を用いて端数処理を行う．必要な処理は端数の数を判定するための条件分岐と DD_ADD_MULT の呼び出し最大 1 回である．アルゴリズム 8 に端数が 3 の場合の処理の例を示す．

アルゴリズム 8 Padding in execution

```
if(r == 3)
    A_reg = SET(A.value[j], A.value[j + 1], A.value[j + 2], 0);
    //process four elements with AVX or AVX2 instruction;
    r = r − 3;
```

2. “Using SSE2 and Scalar” 方式: SSE2 と Scalar 命令（SIMD を用いない命令）を組み合わせてアルゴリズム 9 のように端数の計算を行う．必要な処理は SSE2 または Scalar 命令の加算関数の呼び出し最大 2 回である．

アルゴリズム 9 Using SSE2 and Scalar

```
if(r ≧ 2)
    //process two elements with SSE2 instruction;
    r = r − 2;
if(r == 1)
    //process one element with scalar instruction;
    r = r − 1;
```

```
1  void SpMV_CRS(d_real_SpMat A, d_real_vector x, d_real_vector y)
2  {
3    #pragma omp parallel for
4    for(int i=0;i<A.size();i++){
5      D_SIMD_reg y_reg = zeros;
6      for(int j=A.row_ptr[i];j<A.row_ptr[i+1]-3;j+=4){
7        D_SIMD_reg x_reg = SET(x[A.col_ind[j]],
8            x[A.col_ind[j+1]],
9            x[A.col_ind[j+2]],
10           x[A.col_ind[j+3]]);
11       D_SIMD_reg A_reg = LOAD(A.value, j);
12
13       // y_reg = A_reg * x_reg + y_reg.
14       ADDMUL(A_reg, x_reg, y_reg);
15     }
16     fraction_processing(A, x, y, y_reg);
17     y[i] = reduction(y_reg);
18   }
19 }
```

図 4.10: AVX2 と OpenMP を用いた CRS 形式の SpMV

3. “Using Scalar” 方式: アルゴリズム 10 のように SIMD を用いずに端数の数だけ計算を行う．必要な処理は最大 3 回の加算関数の呼び出しである．

アルゴリズム 10 Using Scalar

for$(;r<0;r=r-1)$

　　//process one element with scalar instruction;

これ以外に CRS 形式の生成時にあらかじめ各行に対して 0 を代入することで端数処理を回避する “Padding in creation CRS” 方式も考えられる．

CRS 形式の SpMV

本節では CRS 形式の SpMV のプログラムの実装と特徴について述べる．AVX2 と OpenMP を用いて CRS 形式の SpMV を倍精度で計算するプログラムを図 4.10 に示す．L.4 の “size” 関数は d_real_SpMat の行列の行数を取得する関数，L.5 の *zeros* は事前に定義したすべての要素が零のレジスタである．

このプログラムは次の手順で CRS 形式の SpMV を実行する．

1. L.3 で OpenMP の “parallel for” 句を用いて L.4 のループをマルチスレッド化
2. 行の要素を 4 つずつ計算するために L.6 のループを 4 ずつインクリメントするループで繰り返す
3. L.5 で各行の計算結果を格納するための AVX2 の SIMD レジスタ構造体を初期化
4. LL.7〜10 で疎行列の非零要素に対応する $\boldsymbol{x}$ の要素を *col_ind* を用いて SIMD レジスタ構造体に格納
5. L.11 で *value* から連続した 4 要素を SIMD レジスタ構造体に格納
6. L.12 で *A_reg*, *x_reg*, *y_reg* に対する積和演算を実行
7. L.16 で行で発生した端数{1, 2, 3}を fraction_processing 関数を用いて計算
8. L.17 で行の計算結果を保持する SIMD レジスタ構造体 *y_reg* の 4 つの要素を reduction 関数を用いて y に足し込む

これらのことから，CRS 形式の SpMV には高速化のうえで次の点に問題があることがわかった．

1. ベクトル $\boldsymbol{x}$ への間接参照（SET 関数）
2. 各行の端数処理（fraction_processing 関数）
3. SIMD レジスタ構造体の要素を $\boldsymbol{y}$ に足し込む計算（reduction 関数）

また，CRS 形式の SpMV では SET 関数が必要なため，プログラムに SIMD の同時処理数に対する依存性が発生する．

BCRS 形式の SpMV

CRS 形式における AVX2 を用いた SpMV の問題点は SIMD 命令が複数の演算を同時実行しなければならないことに起因している．そこで本節では BCRS 形式のブロックサイズを 4 の倍数にし，4 つの要素を同時処理する SIMD 命令に特化させることを検討する．

ブロックサイズが 4 になる BCRS 形式を用いれば必ず 4 つの値を SIMD 命令を用いて同時処理できる．ブロック内の要素に連続にアクセスすることによって高速化効果が得られるが，ブロックに零要素を含めるため演算量やメモリデータ量が増加する．

表 4.8: 各格納形式の SpMV の特徴

	CRS	BCRS1x4	BCRS4x1	BCRS2x2
$\boldsymbol{x}$ の読込	SET	LOAD	BROADCAST	SET
$\boldsymbol{y}$ の読込	zero	zero	zero	zero
$\boldsymbol{y}$ の書込	reduction	reduction	STORE	reduction（2 行おき）
端数処理	各行	none	none	none
演算量の増加（最大）	x1	x4	x4	x4

そこで $r \times c = 4$ となる形式を考える．これは $r = 1, c = 4$ の BCRS4x1 形式，$r = 2, c = 2$ の BCRS2x2 形式，$r = 4, c = 1$ の BCRS4x1 形式の 3 つである．

これらが SIMD を用いた際にどのような特徴になるかを考える．CRS 形式と BCRS1x4, 2x2, 4x1 形式の SpMV の特徴を表 4.8 に示す．ここで “zero” は事前に用意したすべての要素が零の SIMD レジスタを初期化のために代入する処理を表す．“zero” はレジスタに対するレジスタの代入であるため，メモリアクセスを必要とせず性能への影響は小さい．

CRS の問題点をすべて解決できるのは BCRS4x1 形式である．BCRS2x2 形式は端数処理の削減とメモリアクセスの改善効果しかないため性能改善効果はとぼしいと予想できる．BCRS1x4 形式は BCRS4x1 と比べ reduction を用いた $\boldsymbol{y}$ への足し込みを各行で必要とするため BCRS4x1 形式と比べ性能が劣ると予想されるが，疎行列の構造によってはブロックに零を含める数が異なることで BCRS4x1 形式と比べ高速になる可能性がある．5 章で BCRS1x4 と BCRS4x1 の性能評価を行う．

BCRS1x4 形式の SpMV の実装

本節では実装した BCRS1x4 形式の SpMV のプログラムを紹介する．ブロックサイズが $r = 1$, $c = 4$ の BCRS1x4 形式の SpMV のプログラムを図 4.11 に示す．なお，L.4 の d_real_SpMat のメンバ関数 “block_row” は行列のブロック行数を返す関数である．

このプログラムは次の手順で BCRS1x4 形式の SpMV を実行する．

1. L.3 で OpenMP の “parallel for” 句を用いて L.4 のループをマルチスレッド化．
2. L.5 で各行の計算結果を格納するための AVX2 の SIMD レジスタを初期化．
3. L.8 で *col_bind* を用いてブロックに対応する $\boldsymbol{x}$ の 4 つの連続した要素を SIMD レジスタに格納．

```
1  void SpMV_BCRS1x4(d_real_SpMat A, d_real_vector x, d_real_vector y)
2  {
3    #pragma omp parallel for
4    for(int i=0;i<A.block_row();i++){
5      D_SIMD_reg y_reg = zeros;
6
7      for(int jb=A.row_bptr[i];jb<A.row_bptr[i+1];jb++){
8        D_SIMD_reg x_reg = load(x[A.col_bind[jb*4]);
9        D_SIMD_reg A_reg = LOAD(A.bvalue, jb*4);
10
11       // y_reg = A_reg * x_reg + y_reg.
12       ADDMUL(A_reg, x_reg, y_reg);
13     }
14     y[i] += reduction(y_reg);
15   }
16 }
```

図 4.11: AVX2 と OpenMP を用いた BCRS1x4 形式の SpMV

4. L.9 で *bvalue* からブロックの連続した 4 要素を SIMD レジスタに格納.

5. L.12 で *A_reg*, *x_reg*, *y_reg* に対する積和演算を実行.

6. L.6 のループをブロック行に存在するブロックの数だけ繰り返す.

7. 行の計算結果を保持する SIMD レジスタ *y_reg* の 4 つの要素を reduction 関数を用いて y に足し込む.

AVX2 を用いた BCRS1x4 の SpMV は列方向へのループ (L.7) が *row_bptr* に制御され, *bvalue* や $\boldsymbol{x}$ から SIMD レジスタへの読み込みをブロックサイズである 4 おきに行う. 長さ 4 のブロック単位で計算するため端数処理は発生しない. 一方, 各行で各ブロックの計算結果を格納した SIMD レジスタ *y_reg* の 4 つの要素を足し込むための reduction 関数の呼び出しが必要になる.

BCRS4x1 形式の SpMV の実装

ブロックサイズが $r = 4$, $c = 1$ の BCRS4x1 形式の SpMV のプログラムを図 4.12 に示す. このプログラムは次の手順で BCRS1x4 形式の SpMV を実行する.

1. L.3 で OpenMP の “parallel for” 句を用いて L.4 のループをマルチスレッド化. このループは BCRS 形式のブロック行回繰り返される.

```
1  void SpMV_BCRS4x1(d_real_SpMat A, d_real_vector x, d_real_vector y)
2  {
3    #pragma omp parallel for
4    for(int ib=0;ib<A.block_row();ib++){ //block_row is about N/4.
5      D_SIMD_reg y_reg = zeros;
6
7      for(int jb=A.row_bptr[ib];jb<A.row_bptr[ib+1];jb++){
8        D_SIMD_reg x_reg = BROADCAST(x[A.col_bind[jb]]);
9        D_SIMD_reg A_reg = LOAD(A.bvalue, jb*4);
10
11       // y_reg = A_reg * x_reg + y_reg.
12       ADDMUL(A_reg, x_reg, y_reg);
13     }
14     y[ib*4] = STORE(y_reg);
15   }
16 }
```

図 4.12: AVX2 と OpenMP を用いた BCRS4x1 形式の SpMV

2. L.5 で各ブロック行の計算結果を格納するための AVX2 の SIMD レジスタを初期化.

3. L.8 で *col_bind* を用いてブロックの列番号に対応する $\boldsymbol{x}$ の値を SIMD レジスタのすべての要素に格納.

4. L.9 で *bvalue* からブロックの連続した 4 要素を SIMD レジスタに格納.

5. L.12 で *A_reg*, *x_reg*, *y_reg* に対する積和演算を実行.

6. L.6 のループを行に存在するブロックの数だけ繰り返す.

7. 行の計算結果を保持する SIMD レジスタ *y_reg* の値を y[ib] から連続した 4 要素に格納.

AVX2 を用いた BCRS4x1 の DD-SpMV は端数処理や $\boldsymbol{y}$ への足し込みが発生しない．また，$\boldsymbol{x}$ のロードは broadcast 命令で行われる．行に対するループが 4 行おきになることで $\boldsymbol{y}$ へのストアの回数が CRS 形式や BCRS1x4 形式と比べて 1/4 となる．

4.4.2　転置疎行列とベクトルの積（TSpMV）; $\boldsymbol{y} = A^T\boldsymbol{x}$

TSpMV の高速化方針

本節では転置疎行列とベクトルの積 $\boldsymbol{y} = A^T\boldsymbol{x}$; (Transposed sparse matrix and vector multiplication, TSpMV) の SIMD 化，マルチスレッド化について検討する．

CRS 形式は行方向に非零要素を圧縮する格納形式である．列方向に非零要素を圧縮する格納形式として Compressed Column Storage（CCS）形式や Block CCS（BCCS）形式がある [2]. 列方向に圧縮する格納形式を用いれば行方向に圧縮する格納形式における SpMV とほとんど同じプログラムでほとんど同じ性能が期待できる．一方で TSpMV が必要なアルゴリズムは，ほとんどの場合 SpMV も必要になるため CCS や BCCS 形式だと行方向と列方向の形式を両方もたなければならず 2 倍のメモリデータ量が必要となる．

そこでメモリ使用量を重視した実装として，CRS やブロックサイズが $r \times c$ の BCRS 形式を前提として，ベクトル $\boldsymbol{x}, \boldsymbol{y}$ へのアクセスパターンを変更することで行方向に圧縮する格納形式 1 つで TSpMV を行うことを検討する．

CRS 形式の TSpMV

CRS 形式の SpMV ではベクトル $\boldsymbol{x}$ の非連続なデータの読込（SET 関数）が必要なことから，CRS 形式の TSpMV ではベクトル $\boldsymbol{y}$ への非連続な書込が必要になる．そのため TSpMV の実装に必要な次の補助関数を実装した．

scatter 関数　AVX2 のレジスタの 4 つの要素を非連続な 4 つのメモリアドレスに格納．長さ 4 の一時的な配列にレジスタの値を保存し，SIMD を用いずに格納する実現する．

AVX2 と OpenMP を用いて CRS 形式の TSpMV を倍精度で計算するプログラムを図 4.13 に示す．“OMP_NUM_THREADS” はスレッド数を表すマクロ変数で，“omp_get_thread_num()” は自分のスレッド番号を取得する関数である．また，L.4 では d_real_vector の初期化時に第 2 引数に 0 を与えることですべての値を零で初期化している．

このプログラムは次の手順で CRS 形式の TSpMV を実行する．

1. L.3 でスレッド数を取得し，L.4 でスレッド数分 × 行数の長さの一時配列 $work$ を 0 で初期化して確保.
2. L.5 で OpenMP の “parallel” 句を用いてスレッドを立ち上げる．
3. L.7 でスレッドローカルな変数 k にスレッド番号を取得.

```
1  void TSpMV_CRS(d_real_SpMat A, d_real_vector x, d_real_vector y)
2  {
3    int num_threads = OMP_NUM_THREADS;
4    d_real_vector work(y.size() * num_threads, 0);// allocate vector
5  #pragma omp parallel
6    {
7      int k = omp_get_thread_num();
8  #pragma omp for
9      for(int i=0;i<A.size();i++){
10       int jj = i + k * y.size();
11       D_SIMD_reg x_reg = BROADCAST(x[i]);
12       for(int j=A.row_ptr[i];j<A.row_ptr[i+1]-3;j+=4){
13         D_SIMD_reg y_reg = SET(y[A.col_ind[j+0]],
14             y[A.col_ind[j+1]],
15             y[A.col_ind[j+2]],
16             y[A.col_ind[j+3]]);
17         D_SIMD_reg A_reg = LOAD(A.val, j);
18         ADDMUL(A_reg, x_reg, y_reg);
19         scatter(&work[A.col_ind[jj+0],
20             &work[A.col_ind[jj+1]],
21             &work[A.col_ind[jj+2]],
22             &work[A.col_ind[jj+3]],
23             y_reg);
24       }
25       fraction_processing(A, x, y, y_reg);
26     }
27   }
28 #pragma omp parallel for
29     for(int i=0; i<num_threads; i++)
30       for(int j=0; j<y.size(); j++)
31         y[j] += work[j + i * y.size()];
32 }
```

図 4.13: AVX2 と OpenMP を用いた CRS 形式の TSpMV

4. L.8 で L.9 のループをスレッド並列化.
5. L.10 で $k * A.size()$ を用いて各スレッドがアクセスする $work$ の領域を求める.
6. L.11 で行番号に対応した $\boldsymbol{x}$ の値を SIMD レジスタの 4 つの要素に格納.
7. LL.13〜16 で疎行列の非零要素に対応する $\boldsymbol{y}$ の要素を col_ind を用いて SIMD レジスタに格納.
8. L.17 で $value$ から疎行列の連続した 4 要素を SIMD レジスタに格納.
9. L.18 で A_reg, x_reg, y_reg に対する積和演算を実行.
10. LL.19〜23 で y_reg の値を col_ind を用いて非連続な一時配列 $work$ の 4 つの領域に scatter 関数を用いて格納.
11. L.12 のループを行に存在する非零要素の数だけ繰り返す.
12. L.25 で行で発生した端数{1, 2, 3}を fraction_processing を用いて計算.
13. L.28〜31 ですべてのスレッドの計算結果を保持する $work$ の値を $\boldsymbol{y}$ に足し込む.

CRS 形式の SpMV を行方向にスレッド並列化した場合，$\boldsymbol{y}$ への書込は行番号，$\boldsymbol{x}$ のへの読込は列番号に依存する．一方で CRS 形式の TSpMV では $\boldsymbol{x}$ と $\boldsymbol{y}$ のアクセスパターンの関係が逆になり，各スレッドが $\boldsymbol{y}$ に対して列番号のインデックス配列を用いて結果の足し込みを行うことが必要になる．各スレッドが同時に同じ領域に足しこみを行うことを回避するためには，各スレッドが計算結果を保存するために $\boldsymbol{y}$ と同じ大きさの一時配列 $work$ を確保し，各スレッドの計算結果を $work$ に格納し，計算の最後にすべてのスレッドの $work$ の値を総和する必要がある．また，$\boldsymbol{y}$ へのアクセスが不連続になるため SET, scatter 関数が必要になる．

特に $work$ の初期化や値の総和はスレッド数に応じて必要なため，コア数の多い環境ではメモリデータ量や計算時間が大きな問題になることが予想される．

これらのことから，CRS 形式の TSpMV には高速化のうえで次の点に問題があることがわかった.

1. ベクトル $\boldsymbol{y}$ への間接参照（SET, scatter 関数）.
2. 各行の端数処理（fraction_processing 関数）.
3. スレッド数分の一時配列 $work$ の初期化，総和計算.

TSpMV の改善方針

本節では，CRS 形式の TSpMV における問題点の解決方法について考える．

TSpMV は $\boldsymbol{x}$ と $\boldsymbol{y}$ のアクセスパターンおよびロード，ストアの関係が SpMV と逆になる．SpMV において $\boldsymbol{x}$ に対し LOAD 関数を用いてアクセスする BCRS1x4 形式は TSpMV では $\boldsymbol{y}$ に対して STORE を用いてアクセスできるが，一時配列を必要とする問題は解決できない．また，SpMV において $\boldsymbol{x}$ に対し BROADCAST 関数を用いてアクセスする BCRS4x1 形式は，TSpMV では $\boldsymbol{y}$ に対する reduction 関数を用いたアクセスになり，一時配列を必要とする問題も解決できないと考えられる．

これらの問題を解決するために各スレッドがアクセスする列の範囲を制御することで解決することを考える．列番号に対してスレッド番号を用いて何らかの制御を加える場合，BCRS1x4 形式は $c = 4$ のため，列番号に対する制御を行いにくい．BCRS4x1 形式は $c = 1$ のため列方向への制御を行ってもブロックをまたぐことを考える必要はない．また，SpMV において最も CRS 形式の問題を改善できる形式であるため，BCRS4x1 形式において列番号とスレッド番号を用いて各スレッドのアクセスパターンを制御する方式について検討した．

BCRS4x1 (C) 形式の TSpMV

BCR4x1 形式において列番号とスレッド番号を用いて各スレッドのアクセスパターンを制御する方式を “BCRS4x1 (C)” 方式とよぶ．(C) は “column-wise multi-threading” を意味する．AVX2 と OpenMP を用いて倍精度の TSpMV を BCRS4x1 (C) 方式で計算するプログラムを図 4.14 に示す．

このプログラムは次の手順で BCRS4x1 (C) 方式の TSpMV を実行する．

1. L.3 でスレッド数を取得する．
2. L.4 で OpenMP の “parallel” 句を用いてスレッドを立ち上げる．
3. L.6 でスレッドローカルな変数 k にスレッド番号を取得.
4. LL.7〜8 でスレッド数と自スレッド番号から各スレッドが計算する列数の開始位置 (*cstart*) と終了位置 (*cend*) をローカル変数に求める．
5. L.10 で行番号に対応した $\boldsymbol{x}$ の値を SIMD レジスタの 4 つの要素に格納.
6. L.11 で L.12 の列の計算を行うループをマルチスレッド化.
7. L.13 で自スレッドの cstart から cend の範囲の列のみを計算する．

```
1  void TSpMV_BCRS4x1(d_real_SpMat A, d_real_vector x, d_real_vector y)
2  {
3      int num_threads = OMP_NUM_THREADS;
4  #pragma omp parallel
5    {
6      int k = omp_get_thread_num();
7      int cstart = N / num_threads * k;
8      int cend = N / num_threads * (k+1);
9      for(int ib=1;ib<A.block_row();ib++){
10       D_SIMD_reg x_reg = LOAD(x, ib);
11 #pragma omp for
12       for(int jb=A.row_bptr[ib];jb<A.brow_ptr[ib+1];jb++){
13         if(cstart < A.col_ind[jb] <= cend)
14         {
15           D_SIMD_reg y_reg = BROADCAST(y, A.col_bind[jb*4]);
16           D_SIMD_reg A_reg = LOAD(A.val, jb*4);
17           ADDMUL(A_reg, x_reg, y_reg);
18
19           y[ib*4] = STORE(y_reg);
20         }
21       }
22     }
23   }
24 }
```

図 4.14: AVX2 と OpenMP を用いた BCRS4x1 (C) 方式の TSpMV

表 4.9: 各格納形式の TSpMV の特徴

	CRS	BCRS1x4	BCRS4x1	BCRS4x1 (C)
$\boldsymbol{x}$ の読込	BROADCAST	BROADCAST	LOAD	LOAD
$\boldsymbol{y}$ の読込	SET	LOAD	BROADCAST	BROADCAST
$\boldsymbol{y}$ の書込	scatter	STORE	reduction	STORE
端数処理	each row	none	none	none
演算量の増加（最大）	x1	x4	x4	x4
一時配列 *work* の長さ	T ×N	T × N	T × N	none

NOTE: N は行列の行数，T はスレッド数.

8. L.15 で疎行列の非零要素に対応する $\boldsymbol{y}$ の要素を *col_ind* を用いて SIMD レジスタに格納.
9. L.16 で *value* から疎行列の連続した 4 要素を SIMD レジスタに格納.
10. L.18 で *A_reg*, *x_reg*, *y_reg* に対する積和演算を実行.
11. 行の計算結果を保持する SIMD レジスタ *y_reg* の 4 つの要素を STORE 関数を用いて $\boldsymbol{y}$ に格納.

BCRS4x1 (C) 方式は各スレッドがすべての行に対してアクセスし，*cstart* と *cend* を用いて計算範囲を制御する．各行で計算範囲を判定するための条件分岐が発生するが，スレッド数分の一時配列（*work*）を必要とせず，不連続なアクセスや端数処理も発生しない．なお，厳密には *cstart* と *cend* がスレッド数で割り切れない場合の処理を記述する必要がある.

各格納形式における TSpMV の比較

CRS 形式，BCRS1x4 形式，BCRS4x1 形式，BCRS4x1 (C) 方式の TSpMV の特徴を表 4.9 にまとめる．ここで T はスレッド数，N は行列の行数である.

CRS 形式は $\boldsymbol{y}$ への不連続な読込，書込や端数処理，スレッド数分の一時配列の初期化と総和が必要になる．BCRS1x4, 4x1 形式は $\boldsymbol{y}$ への不連続な読込，書込や端数処理は必要ないが，スレッド数分の一時配列が必要で，演算量，メモリデータ量が最大で 4 倍に増加する．BCRS4x1 形式は $\boldsymbol{y}$ の読込のための非連続なロードをなくせるが，4 要素計算するごとに SIMD レジスタの要素を reduction 関数を用いて総和する処理が発生するため現実的ではない．BCRS4x1 (C) 方式を用いることでスレッド数分の一時配列をなくせるが，計算範囲を判定するための条件分岐などが必要になる.

第5章　性能評価

本章ではこれまで述べたベクトルや疎行列に対する基本演算の性能を評価する．

5.1　評価方法と実験環境

5.1.1　性能指標

DD 演算の性能を評価するとき，浮動小数点演算の回数を数える方法が 2 種類考えられる．

1. 倍精度演算を 1 回 と数える（倍精度換算の演算量）

2. DD 演算を 1 回 と数える（DD 換算の演算量）

1. は，1 秒間に倍精度演算を何回行えたかで，倍精度向けの性能（Performance for double）とよび，この単位を “DFLOPS” とよぶ，このとき FMA 演算の演算回数は 2 回と数える．これは演算器の理論性能に対する評価に用いる．

2. は，1 秒間に DD 加算または乗算を何回行えたかで，DD 向けの性能（Performance for DD）とよび，この単位を “DDFLOPS” とよぶ．これは演算数の異なる精度や演算同士を比較する際に用いる．

例えば倍精度の加算と乗算を 1 回として数えた場合，DD 加算は表 4.3 から倍精度演算換算で 11 回，FMA 命令を用いない場合の DD 乗算は表 4.5 から倍精度演算換算で 24 回で，合計 35 回である．一方で DD の積和演算は DD の加算と乗算をそれぞれ浮動小数点演算 1 回として数えれば演算数は 2 回である．そのため DD の積和演算を N 回行った時間を t としたときの性能は，

- 倍精度演算向けの性能: $35N$ / t (FMA 命令を用いないとき) : DDFLOPS
- 倍精度演算向けの性能: $21N$ / t (FMA 命令を用いたとき) : DDFLOPS
- DD 演算向けの性能：$2N$ / t : DFLOPS

と計算できる．なお，浮動小数点演算を 1 秒間に 10^9 回行った単位としてそれぞれ Giga DFLOPS（GDFLOPS）, Giga DDFLOPS（GDDFLOPS）と表記する．

また，BCRS 形式の SpMV と TSpMV は零要素を含めることにより CRS 形式と比べて演算量が増加するため，これらの定義によって CRS 形式と比較して評価することは難しい．そのため BCRS 形式の SpMV と TSpMV の評価には性能でなく時間を用いる．

5.1.2　ピーク性能と補正ピーク性能

これまでの DD 演算を高速化する研究 [13–15, 49, 57] では，DD の性能を評価するとき実行時間や性能を倍精度の結果やピーク性能と比較している．

FMA 演算器を有する CPU のピーク性能は各サイクルで FMA 演算を行った場合として定義されている．一方で FMA 演算は積と和がペアで存在する場合しか利用できないことや，DD 演算は積と比べて和の演算が多いが Haswell アーキテクチャは加算命令を並列して行えないことから，DD 演算の実行時にピーク性能が出ることはない．

そのため本研究では現在一般的なハードウェアにおいて DD 演算の性能を評価する場合に和，積，積和の数の比率から実際に期待できるピーク性能を再定義して性能を評価することを提案する．これにより実際にハードウェアの効率をどの程度引きだせたかをより厳密に評価できる．この性能を補正ピーク性能と定義する．

例として内積や SpMV に使われる DD の積和演算において期待できる倍精度向けの性能を求めた．FMA 命令を用いない場合，1 要素の計算は倍精度の加減算命令 26 回，倍精度乗算 9 回で構成される．FMA 命令を用いた場合の DD 積和演算の演算量は 21 Flops で，倍精度の加減算 14 回，乗算 1 回，FMA3 回によって構成される．

Haswell は加算，乗算，FMA 命令を発行できるポートと乗算，FMA 命令を発行できるポートを 1 器ずつ備える．演算器が浮動小数点演算を 1 回行うのに 1 cycle かかると仮定したとき，すべての和と積，和と FMA が同時に実行できた場合でも加算命令を発行するための 14 cycles が必要になる．

2 つの演算器で FMA 演算を 14 cycles 行った場合，$14 \times 2 \times 2$ 回の FMA 演算，つまり 56 Flops 計算できる．一方で DD 積和演算は 14 cycles かけて 21 Flops しか計算できないため，補正ピーク性能はピーク性能の 21 / 56 = 37.5 %となる．このように演算器の構成とアルゴリズムの演算の内訳を考慮することで性能からプロセッサの稼働率を正しく評価できる．

次にこの考え方に基づいて FMA 命令を用いた場合と用いない場合に期待できる性能向上について見積もる．FMA を用いない場合の DD 積和演算は倍精度加算 21 回，乗算 14 回からなるため，すべての加算と乗算が同時に行えた場合でも 21 cycles かかる．FMA を用いた場合の DD 積和演算は 14 cycles かかるため FMA 命令を用いることによる効果は 21 / 14 = 1.5 倍と計算できる．これらの演算器構成に基づいた補正ピーク性能や見積もりを基に DD 演算の

性能を評価する．

実験では FMA 命令と SIMD 命令を用いない実装を “Scalar,” AVX を用いた実装を “AVX,” FMA 命令，AVX2 を用いた場合の実装を “AVX2,” FMA 命令，AVX2 を用いた場合の倍精度の実装を “double” とよぶ．

また，マルチスレッド化の評価に用いる 1 スレッドの結果は OpenMP のスレッド数の環境変数に 1 に設定したものを用いた．

5.1.3 計測方法

今回対象とする演算の多くは 1 回だけ行う場合は多くの時間がかかるものではないため，時間計測関数の呼び出し時間のオーバヘッドなどによる影響を受けやすい．そのため時間の計測は繰り返し実行したときの平均時間を用いる．これは Krylov 部分空間法において繰り返し利用されることを考えれば問題はない．

時間計測には C++の “std::chrono::system_clock::now()” 関数を用い，反復回数は各実験に対して 100 反復以上かつ合計時間が 100 ms. 以上になるように定めた．

5.1.4 実験環境

実験環境を表 5.1 に示す．コアに対し最大で 1 スレッドを割り当てるため，ハイパースレッディングは無効にした．

コンパイラオプションは次の 7 種類である．DD 演算のアルゴリズムは命令の順序が重要であるため，最適化オプションである “-O3” オプションをつけると最適化によって答えが合わなくなった．命令の並び替えを抑制する “-fp-model precise” をつければ答えが合うようになったことからこれを用いた．また，より強い最適化抑制オプションに “-fp-model strict” がある．こちらを用いても答えは合うことが確認できたが，性能が “-fp-model precise” と比べて低下したので使用しない．

1. 最適化を行う “-O3”
2. OpenMP を有効化する “-openmp”
3. コンパイラによる自動ベクトル化を抑制する “-no-vec”
4. AVX を有効にする “-xAVX”
5. AVX2 を有効にする “-xCORE-AVX2”

表 5.1: 実験環境

CPU	Intel Core i7 4770@3.4GHz Intel Haswell Architecture
Number of core	4
L3 cache size	8 MB
Memory	16 GB DDR3-1600 dual channel
Memory band-width	25.6 GB/s (12.8 × 2)
OS	Fedora 20
Compiler	Intel C/C++ Compiler 13.0.1
Compiler Option (Scalar)	-O3 -openmp -no-vec -fp-model precise
Compiler Option (AVX)	-O3 -openmp -xAVX -fp-model precise
Compiler Option (AVX2)	-O3 -openmp -xCORE-AVX2 -mfma -fp-model precise
Compiler Option (double)	-O3 -openmp -xCORE-AVX2 -mfma
OpenMP Scheduling	guided

6. FMA 命令を有効にする “-mfma”

7. 命令の並び替えを抑制し精度を保つ “-fp-model precise”

Intel core i7 4770 のピーク性能は，

- 3.4[GHz] × 4 [core]× 4 (SIMD) × 4 (FMA × 2) = 217.6 [GDFLOPS]

である．

今回の実験環境におけるメモリバンド幅は 25.6 GB/s，ピーク性能は 217.6 GDFLOPS で，B/F は 25.6 / 217.6 = 0.12 である．

5.2　スカラ演算の性能

5.2.1　比較対象

本節では QD ライブラリを用いた DD のスカラに対する演算の評価を行う．DD 演算は 4 倍精度相当の精度を実現する手法として他の実現手法とくらべて高速であるといわれている [25]．そこで QD ライブラリの “dd_real” 型と次の 4 つを比較する．

1. C++の倍精度型

2. Intel C/C++ compiler (icc) [86] の 4 倍精度型

3. GMP [42] における 4 倍精度相当の多倍長型

4. exflib [87] における 4 倍精度相当の多倍長型

2. はコンパイラで 4 倍精度 (binary128) [7] をサポートしているもので，C / C++コンパイラである GNU Compiler Collection (gcc) [88] の Ver. 4.6（2011 年）以降などにも搭載されている．icc では 4 倍精度型を “_Quad” 型として実装している．

3. と 4. の GMP や exflib は DD 型とは異なる構造の多倍長型で DD が multiple-component 形式とよばれるのに対し，multiple-digit 形式とよばれる指数部を表現するための整数型の変数と仮数部を表現数するための浮動小数点数の配列がセットになったものである [25]．GMP は符号部を 1 bit 文字型，指数部を 32 bit または 64 bit 整数型，仮数部を倍精度浮動小数点数の配列で表現する．exflib は符号部 1 bit と指数部 63 bit をあわせて倍精度浮動小数点数，仮数部を倍精度浮動小数点数の配列で表現する [89].

multiple-digit 形式の利点は仮数部を配列でもつため桁数の増減に柔軟で精度を動的に変えられることである．一方で 128 bit のメモリデータ量では binary128 相当の精度を達成することができないことや，データ構造が異なるため倍精度型と組み合わせるには倍精度型を multiple-digit 形式の倍精度相当の精度に変換しなければならないなどの問題点がある．たとえば exflib のデータ形式で binary128 相当の精度を作るには，符号部，指数部のために倍精度浮動小数点数 1 つ（64 bit）, 仮数部のために 64 bit × 2 = 128 bit で，合計 192 bit が必要になる．

5.2.2 QD ライブラリの性能

FMA 命令以外の高速化手法を適用していない DD 演算と倍精度演算の計算時間の比は演算量の比と同程度になることが予想できる．そこで計算時間の比が演算量の比と等しくなることを確認するため，QD ライブラリを用いた DD の四則演算と倍精度の四則演算をそれぞれ 10^6 回行ったときの時間を表 5.2 に示す．なお，倍精度の乗算には FMA 命令は用いず倍精度乗算を行った結果を載せた．

- DD 加算は倍精度の 11 倍の演算量で，計算時間は約 11.3 倍．計算時間は演算量の比とほぼ等しい．
- FMA を用いない DD 乗算は倍精度の 24 倍の演算量で，計算時間は約 24.2 倍．計算時間は演算量の比とほぼ等しい．

表 5.2: QD ライブラリを用いて DD スカラ演算を 10^6 回行う時間 [ms.]（比）

	Double	DD
加算	0.64 (1.00)	7.33 (11.3)
乗算	0.65 (1.00)	15.78 (24.2)
		8.07 (12.4) (FMA あり)
除算	3.61 (1.00)	101.80 (28.2)

- FMA を用いた DD 乗算は倍精度の 10 倍の演算量で，計算時間は約 12.4 倍．他の演算と比べて倍精度との比が大きいのは演算量に含まれない 2 回の符号反転（図 4.4 の L.11, L.13）のためと考えられ，符号反転を 1 演算としてカウントすると演算量は 12 になるため，計算時間は演算量の比とほぼ等しい．

- DD 除算は倍精度の 27 倍の演算量で，計算時間は約 28.2 倍．計算時間は演算量の比とほぼ等しい．

QD ライブラリの DD スカラ演算は演算量とほぼ等しい性能であることから，関数呼び出しなどのオーバヘッドによる影響は少なく QD ライブラリには高速化する余地はほとんどないと考えられる．

次に他の高精度演算ライブラリとの比較を行った．GMP と exflib には仮数部 104 bit を指定した．GMP と exflib は配列の長さで仮数部を表現するため指定された精度を満たす最小の配列長が確保される．そのため厳密には binary128 型や DD 型よりも高い精度型が作られる．それぞれのソフトウェアの積和演算を 10^6 回行ったときの平均時間は次のようになった．

- icc を用いた C++の倍精度演算は 1.31 [ms].

- QD ライブラリの FMA 命令を用いた DD 演算は 15.5 [ms].

- GMP で DD 相当の仮数部 104 bit を指定したものは 35.6 [ms].

- exflib で DD 相当の仮数部 104 bit を指定したものは 21.7 [ms].

- icc の binary128（“_Quad”）は 20.1 [ms].

倍精度演算と DD 演算の実行時間は表 5.2 の加算と乗算の実行時間の和とほとんど変わらない．DD 演算は GMP, exflib, icc の実装に対してそれぞれ約 2.3, 1.4, 1.3 倍高速で，他の高精度演算ライブラリよりも高速であることが確認できた．これらの結果から開発するソフト

表 5.3: DD ベクトル演算の演算量，カッコ内は倍精度加減算：乗算：FMA 演算の数, Load / Store はメモリ読込，書込を行うベクトルの数

	Load	Store	FMA が使えない場合の演算量	FMA が使える場合の演算量
axpy	2	1	35 (26:9:0)	21 (14:1:3)
axpyz	2	1	35 (26:9:0)	21 (14:1:3)
xpay	2	1	35 (26:9:0)	21 (14:1:3)
dot	2	0	35 (26:9:0)	21 (14:1:3)
nrm2	1	0	31 (24:7:0)	21 (14:1:3)
scale	1	1	24 (15:9:0)	10 (3:1:3)

NOTE:α, val は DD スカラ, $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ は DD ベクトル.

ウェアで QD ライブラリをリンクし，スカラ演算には QD ライブラリを用いる．なお，GMP を用いたプログラムは本研究の過程で開発した Xev-GMP という C 言語の倍精度プログラムを，GMP を用いた任意多倍長プログラムへ自動変換する機構を用いて自動生成した．これについては付録に載せた．

5.3 ベクトル演算の性能

本節では DD のベクトル演算に対して FMA 命令，SIMD 命令，マルチスレッドを用いた場合の性能を評価する．

5.3.1 ベクトル演算の特徴

表 5.3 に DD のベクトル演算における倍精度換算の演算量，表 5.4 に B / F を示す．また，4 スレッドで並列化を行った場合のベクトル演算の補正ピーク性能を表 5.5 に示す．

演算によって必要な倍精度加算と倍精度乗算の数が異なるため補正ピーク性能も異なる．たとえば scale は他の演算と比べて相対的に乗算と FMA 演算の比率が多いため補正ピーク性能が高い．これらの値を基に DD のベクトル演算の性能を評価する．

5.3.2 ベクトル演算の性能

メモリ性能の影響を受けにくい，データがすべて L3 キャッシュに収まるベクトルサイズ N = 10^5 において入力がすべて DD のベクトル演算を 1, および 4 スレッドのときの性能を評価し

表 5.4: ベクトル演算における 1 演算あたりのメモリ要求量 [Byte / Flop]

	double	DD（FMA が使えない場合）	DD（FMA が使える場合）
axpy	12	1.37	2.09
axpyz	12	1.37	2.09
xpay	12	1.37	2.09
dot	8	0.91	1.39
nrm2	4	0.52	0.70
scale	16	1.33	1.60

表 5.5: DD ベクトル演算における倍精度向けの補正ピーク性能（4 threads） [GDFLOPS]

	Scalar	AVX	AVX2
axpy	15.5	62.0	81.6
axpyz	15.5	62.0	81.6
xpay	15.5	62.0	81.6
dot	15.5	62.0	81.6
nrm2	13.6	54.4	81.6
scale	13.6	54.4	108.8

表 5.6: 1 スレッドでの DD ベクトル演算の実行効率 [GDFLOPS]（補正ピーク性能比 [%]）

	Scalar	AVX	AVX2
axpy	3.32 (86)	15.2 (98)	19.7 (97)
axpyz	3.21 (83)	14.9 (96)	19.9 (98)
xpay	3.26 (84)	15.1 (97)	19.3 (95)
dot	3.29 (85)	15.2 (98)	18.5 (91)
nrm2	2.82 (83)	12.6 (93)	18.1 (89)
scale	2.84 (81)	12.2 (90)	24.1 (89)

表 5.7: 4 スレッドでの DD ベクトル演算の実行効率 [GDFLOPS] （補正ピーク性能比 [%]）

	Scalar	AVX	AVX2
axpy	12.6 (81)	59.5 (96)	75.0 (92)
axpyz	12.7 (82)	59.5 (96)	73.4 (90)
xpay	12.4 (80)	60.8 (98)	74.3 (91)
dot	12.9 (83)	58.9 (95)	71.8 (88)
nrm2	12.6 (80)	48.9 (90)	71.3 (87)
scale	11.3 (83)	48.4 (89)	93.6 (86)

た．このとき 1 本のベクトルサイズは 1.6 MB である．また，実験結果は 10^5 回実行した平均時間を用いた．

表 5.6 に 1 スレッドの性能，表 5.7 に 4 スレッドの性能を示す．このとき性能は倍精度向けの性能 [GDFLOPS] である．これはプロセッサの実行効率の指標で Scalar および AVX と比べて AVX2 は演算量が異なるため，時間の比とは異なる．

DD ベクトル演算の性能はいずれも高く補正ピーク性能の 80 %以上で，すべてのケースで高い性能が出せていることがわかる．また，すべてのケースで 1 スレッドから 4 スレッドにしたことによるマルチスレッド化の効果は 3.6〜3.9 倍で並列化が有効に行えていることが確認できた．

次に 1 スレッドの結果に着目し SIMD 命令と FMA 命令の利用による効果について述べる．Scalar に対する AVX の性能向上比は 4.5〜4.6 倍である．AVX によって期待される性能向上は 4 倍であるが，これを上回った．これは Scalar が 2 オペランドの命令であるのに対し AVX は 3 または 4 オペランドの命令であるため，レジスタ退避，復元のための命令（move 命令）が削減されたからと考えられる．コンパイラが生成した axpy のアセンブリコードを確認する

と Scalar は 1 ループあたりロードおよびストア命令，積および和命令，move 命令がそれぞれ 3, 35, 13 回であるのに対し，AVX では 3, 35, 3 回で，10 回の move 命令が削減されていることが確認できた．

AVX に対する AVX2 の FMA 命令を用いた性能向上比は scale 以外の演算では 1.2〜1.4 倍である．FMA 演算の利用によって最大で 1.5 倍の性能向上が見込めるが，演算量の少ない DD 乗算を用い，ループ内の演算量が少なくなったことでパイプラインや命令の並べ替えなどが行いにくくなり 1.2〜1.4 倍程度に収まったのだと考えられる．計算時間を調べると AVX2 を用いた axpy の計算時間は AVX の axpy に対し約半分の時間で計算できており，FMA 命令の利用は有効であることを確認した．

これらの結果から高速化手法の適用によって性能が悪くなるものはなく，データがキャッシュに収まるケースにおいて適用した高速化手法が有効であることが確認できた．

5.3.3　ベクトル演算のメモリアクセス性能

ベクトル演算はメモリに対して連続でキャッシュを用いてデータの再利用を行えないためメモリ性能に影響を受けやすい演算である．DD のベクトル演算のメモリアクセス性能を評価するため，axpy のベクトルサイズ N を 10^3 から 8.0×10^5 まで変化させ，AVX2 の 1 スレッドおよび 4 スレッドで行った結果を図 5.1 に示す．このときベクトルサイズ N が 2.6×10^5 まで L3 キャッシュ（8 MB）に収まる．

axpy は，1 回の計算に DD 変数のロードを 2 回，ストアを 1 回行う．そのためベクトル 1 要素を演算するために必要なメモリへのデータアクセス量は $16 \times (2 + 1) = 48$ [byte] である．メモリバンド幅（今回の環境では 25.6 [GB/s]）に対してメモリ効率をどれだけ引き出せたか（帯域利用効率とよぶ）はベクトルの長さが N のとき次のように求められる．

- 25.6 (memory bandwidth) / (N * 16 * 3 / elapsed time)

データがキャッシュに収まるときメモリの帯域利用効率は 250%を超えた．このときのマルチスレッド化の効果は約 3.6 倍である．

データがキャッシュに収まらないとき性能は約 10 GDFLOPS まで低下した．これは補正ピーク性能に対し約 12 %しか出ていない．このときのメモリの帯域利用効率は 86〜98%である．DD のベクトル演算はメモリ性能に制約を受け，データがキャッシュに収まらないときにはマルチスレッド化は効果がないことがわかった．

すべてのベクトル演算に対しデータがキャッシュに収まらないベクトルサイズ $N = 8.0 \times 10^5$ において，各高速化手法および倍精度の性能を比較した結果を図 5.2 に示す．ここで 1T は 1

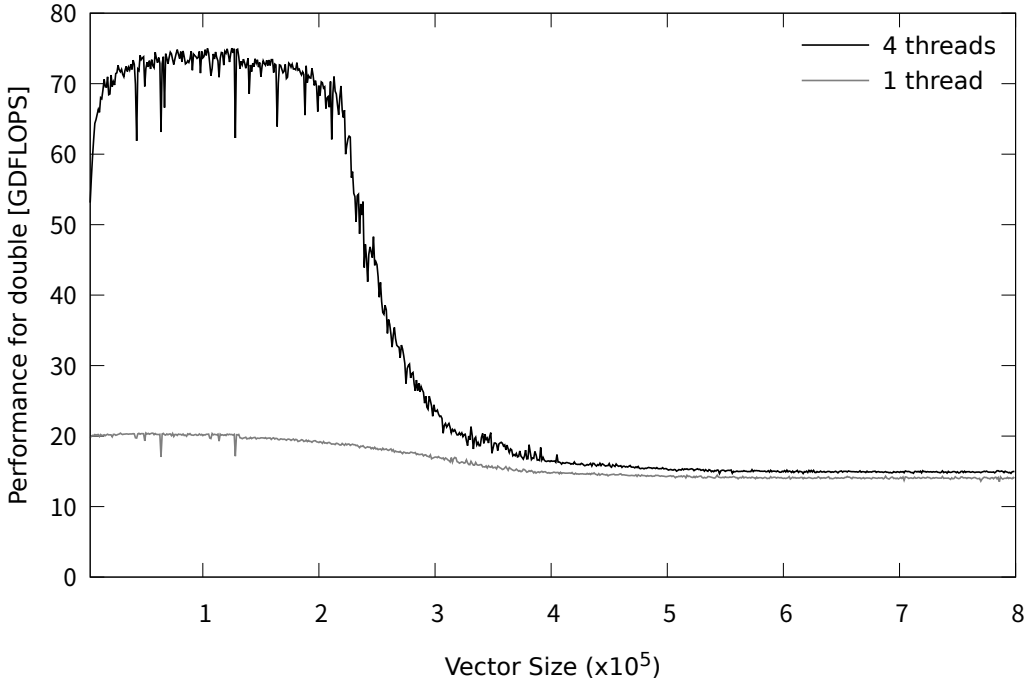

図 5.1: DD の axpy ($\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$) の 1, 4 スレッド並列における性能 [GDFLOPS]

スレッド，4T は 4 スレッドを意味する．これらの性能は DD 向けの性能 [GDDFLOPS] で同一演算内における性能比は実行時間の比と等しい．

Scalar, 1 スレッドの結果は表 5.6 と同程度の性能になった．Scalar (1T) は DD 演算の B / F が低いためメモリから必要なデータ供給にかかる時間よりも計算に時間がかかり，プロセッサの処理性能がボトルネックとなったと考えられる．これに対し並列化を行った Scalar (4T) や AVX2 (4T) はどの演算においても性能が Scalar の 2〜3 倍程度にとどまった．このことから DD のベクトル演算における SIMD 化やマルチスレッド化による高速化の効果はメモリ性能に制約を受け 2〜3 倍以上の高速化は期待できないと考えられる．

また，AVX2 (4T) と Scalar (4T) の性能はほぼ同等であることから，ベクトル演算に対して SIMD 命令や FMA 命令を用いることは性能に悪影響を与えないことがわかった．

一般的に倍精度のベクトル演算は B/F が高いためメモリ性能に制約を受ける．double (4T) の性能は AVX2 (4T) や Scalar (4T) の約 2 倍であることから，データがキャッシュに収まらないときの性能比はメモリデータ量の比によって生じていると考えられる．また，演算ごとに性能が異なるのは演算ごとに必要とするロード，ストアのデータ量が異なるためと考えられる．

FMA を用いない場合の axpy の B/F は 1.37 である．これは 1 演算を行う間に 1.37 Byte のデータが供給されればプロセッサがデータの供給を待つことなく計算を行い続けられることを意味する．

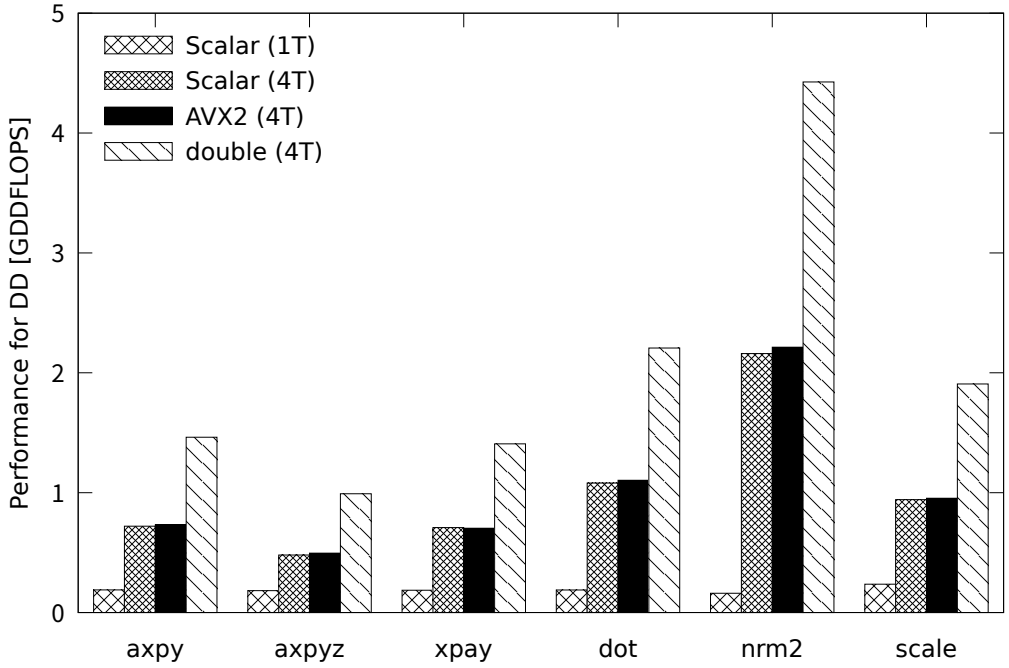

図 5.2: データがキャッシュに収まらない場合のベクトル演算の性能 [GDDFLOPS]（ベクトルサイズ 10^5）

Scalar(4T) の補正ピーク性能は 15.5 GDFLOPS である．今回のシステムのメモリバンド幅は 25.6 GB/s であるため，15.5 GDFLOPS の性能が出たときプロセッサがメモリに 1 演算あたりに要求できるデータ量は 25.6 / 15.5 = 1.65 byte である．これは axpy に必要な 1.37 に比べて高いことから，Scalar の axpy の性能がメモリ性能でなくプロセッサの性能に制約を受けたことは妥当であると考えられる．

一方で並列化によってプロセッサからメモリへの要求量が増加することでボトルネックの要因がプロセッサの性能からメモリの性能に変わり，性能はメモリ性能に制約を受けるようになった．同様にメモリ制約に制約を受けていると考えられる倍精度のベクトル演算に対して性能比がメモリデータ量の比である 2 倍であることから，これ以上の高速化は難しいと考えられる．

一方で SIMD 化やマルチスレッド化によって性能に悪影響を与えるケースはないこと，データがキャッシュに収まるケースでは SIMD 化とマルチスレッド化によって性能が補正ピーク性能の 85 %以上を出せていること，Scalar(1T) の性能はメモリ性能に制約を受けるまで達していないことから，DD のベクトル演算に対して SIMD 化や FMA 化を行うことは有用であると考えられる．

5.4 疎行列ベクトル積の性能

本節では前章で挙げた CRS 形式と BCRS 形式における DD の疎行列ベクトル積に対して FMA 命令，SIMD 命令，マルチスレッドを用いた性能を評価する．

5.4.1 メモリへのデータ要求量の削減

ベクトル演算の性能はメモリ性能に制約を受け，データがキャッシュに収まらない問題サイズでは並列化の効果が十分に受けられなかった．そのためベクトル演算と比べて多くの時間がかかる SpMV や TSpMV ではメモリへのデータ要求量の削減について検討する必要があると考えた．

行列の要素の精度を倍精度にすることでメモリへのデータ要求量を削減することに着目した．多くの Krylov 部分空間法では行列の値は書き換えられないため，行列を倍精度としても丸め誤差の影響はなく，小武守らの反復解法ライブラリ Lis [56] においても用いられている手法 [15] である．これにより行列を DD でもつ場合と比べてメモリデータ量を約半分にでき，疎行列ベクトル積のメモリへのデータ要求量が削減できる．

CRS 形式の疎行列ベクトル積; $y = Ax$ は，1 要素の計算を行うためにベクトル $\boldsymbol{x}$，$\boldsymbol{y}$，疎行列の列インデックス，要素の値をメモリから読み込む必要がある．

疎行列ベクトル積の計算に必要な積和演算の B/F ついて考える．倍精度の疎行列ベクトル積は演算量が 2 Flops，ベクトルは倍精度，列インデックスは 4 バイト整数型，行列の要素の値は倍精度であるため，1 命令あたりのメモリへの要求量は 28 (Byte) / 2 (Flops) = 14 Byte / Flop である．

ベクトルと行列の要素の値を DD としたとき FMA を用いた場合の DD の積和演算の演算量は 21 Flops である．このとき 1 命令あたりのメモリへの要求量は 52 (Byte) / 21 (Flops) = 2.48 Byte / Flop となる．一方でベクトルを DD，行列の要素を倍精度にしたとき積和演算に必要な演算量は 19 Flops でよく，1 命令あたりのメモリへの要求量は 44 (Byte)/ 19 (Flops) = 2.32 Byte / Flop である．これは行列の要素を DD とした場合と比べて約 6%少ない．また，FMA を用いない場合においても同様に計算量を 35 Flops から 33 Flops に減らすことができ，1.58 Byte / Flop から 1.33 Byte / Flop と約 16%少ない．

そのため本論文では対象とする疎行列ベクトル積や転置疎行列ベクトル積を行列の要素を倍精度，ベクトルを DD とした $y_{DD} = A_D x_{DD}$（以下，DD-SpMV）および $y_{DD} = A_D^T x_{DD}$（以下，DD-TSpMV）とする．

また，前節のベクトル演算の結果から，DD スカラに対する高速化である FMA 命令を利用した DD 乗算のアルゴリズムの使用は性能に悪影響を与えることはないことがわかった．そ

のため疎行列に対する評価ではすべてのケースに FMA 命令を用いて SIMD 化，マルチスレッド化を主なターゲットにした評価を行う．

DD-SpMV や DD-TSpMV の核となる倍精度と DD の積和演算; $y_{DD} = A_D\ x_{DD}$ は FMA を用いる場合は演算量は加減算 14 回，乗算 1 回，FMA2 回によって構成される．この演算の補正ピーク性能を 5.1.2 節の方法で求めるとピーク性能の約 34 %になる．これは今回のシステムでは約 74.0 GDFLOPS である．

5.4.2　対象問題

実験には次の 2 種類の疎行列セットを用い，実験結果には行列ごとに 100 回反復計測した平均を用いた．表 5.8 に問題セット B の 23 問の行列サイズや非零要素数を示す．

問題セット A　SuiteSparse Matrix Collection [90] からえた行列サイズ 10^3 以上の疎行列 100 問

問題セット B　問題セット A から選んだ代表的な 23 問

問題セット A, B に加えて任意にサイズや非零要素数を変えられる疎行列として非零要素の配置が,

if(0 ≤ j - i < m) a_{ij} = value

else $a_{ij} = 0$

を満たす 1 行あたり m 個の非零要素が連続して配置された test(m) を作成した．

5.4.3　CRS 形式の疎行列ベクトル積

DD-SpMV の端数処理

はじめに 4.4.1 節で述べた端数処理方法の比較を行う．これは疎行列ベクトル積において各行の非零要素を SIMD 命令を用いて 4 つずつ計算していった際に生じる余り{1, 2, 3}の計算である．次の 4 種類の実装を比較し，最も高速なものを選ぶ．

- “Padding in execution” 方式
- “Using SSE2 and Scalar” 方式
- “Using Scalar” 方式
- “Padding in creation CRS” 方式

表 5.8: 問題セット B の疎行列

Matrix Name	N	nnz	nnz/row
msc01440	1,440	44,998	31.2
bcsstk13	2,003	83,883	41.9
fv1	9,604	85,264	8.9
nasa4704	4,704	104,756	22.3
bcsstk15	3,948	117,816	29.8
aft01	8,205	125,567	15.3
Dubcova1	16,129	253,009	15.7
s2rmq4m1	5,489	263,351	48.0
bcsstk16	4,884	290,378	59.5
Na5	5,832	305,630	52.4
Kuu	7,102	340,200	47.9
c-56	35,910	380,240	10.6
apache1	80,800	542,184	6.7
olafu	16,146	1,015,156	62.9
Dubcova2	65,025	1,030,225	15.8
case39	40,216	1,042,160	25.9
bcsstk36	23,052	1,143,140	49.6
raefsky4	19,779	1,316,789	66.6
SiO	33,401	1,317,655	39.4
bcsstk35	30,237	1,450,163	48.0
bcsstk39	46,772	2,060,662	44.1
TSOPF_FS_b162_c4	40,798	2,398,220	58.8
nasasrb	54,870	2,677,324	48.8

表 5.9: 各端数処理手法の計算時間 [ms] (4 threads, N=10^4)

	m = 63	m = 1023
Padding in execution	25	38
Using SSE2 and Scalar	40	46
Using Scalar	25	39
Padding in creation CRS	24	38

各行で必ず端数が発生する N=10^4 の test(63), test(1023) を用いて評価を行った．AVX2 を用いた 4 スレッドでの実行時間を表 5.9 に示す．

“Padding in creation CRS” 方式は事前に端数が発生しないように行列に零要素を詰める手法で，端数処理が発生せず最も高速である．一方で “Padding in creation CRS” 方式は疎行列の作成時にデータサイズが増えたり，生成時間がかかるなどの問題点がある．

次に高速なものが “Padding in execution” 方式である．“Padding in Creation CRS” 方式を基準とした実行時間の比は，test(63) で約 1.04 倍, test(1023) でほぼ同様である．“Padding in execution” 方式は生成時間にも影響がない．このことから本研究では “Padding in execution” 方式が CRS 形式の DD-SpMV において最適な端数の計算方法であると結論づけ，端数の処理に “Padding in execution” 方式を用いる．

また，“Using SSE2 and Scalar” が最も時間がかかる．これは Intel CPU では AVX，AVX2 の ymm レジスタと SSE2 の xmm レジスタは論理的に別のレジスタであるが，物理的には同じレジスタを使っているため同一コード内で両方の命令を用いると論理的整合性を保つためにレジスタの内容をすべて退避，復元するためである．

CRS 形式の DD-SpMV のメモリアクセス性能

本節では CRS 形式の DD-SpMV におけるメモリアクセスの影響を評価する．test(32) の行列サイズを 10^3 から 4.0×10^5 まで 10^3 ずつ変化させたときの AVX2 における 1, 4 スレッドの倍精度向けの性能 [GDFLOPS] を図 5.3 に示す．test(32) は 1 行あたり 32 の非零要素が連続して配置され，端数処理やランダムアクセスが発生せず，行列サイズ 1.9×10^4 までデータがキャッシュに収まる．

データがキャッシュに収まる，収まらないに関わらずマルチスレッド化により 3.0〜3.4 倍に性能が向上した．行列を倍精度にしたことにより CRS 形式の DD-SpMV はメモリ性能に制約を受けないことが確認できた．

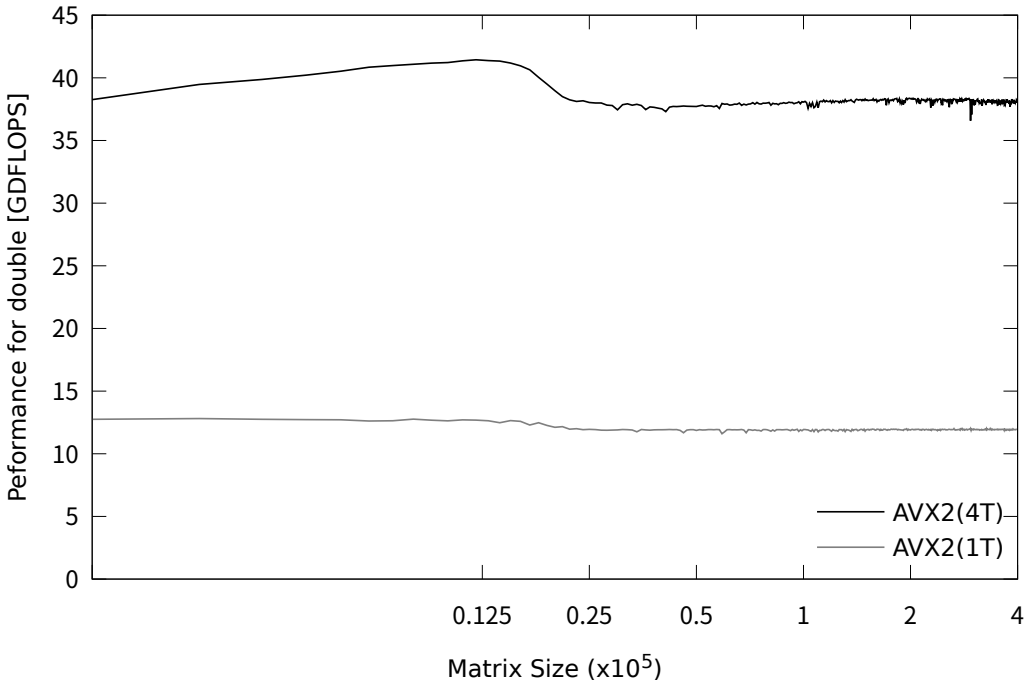

図 5.3: DD-SpMV におけるメモリアクセスの影響 [GDFLOPS] (1, 4 threads)

CRS 形式の DD-SpMV の性能劣化要因の影響

本節では 4.4.1 節で述べた AVX2 を用いた CRS 形式の SpMV における 3 つの問題点の影響について評価する．また，これら 3 つの問題点を CRS 形式の SpMV における性能劣化要因とよぶ．プログラムの実装からわかった AVX2 を用いた CRS 形式の SpMV の性能劣化要因は次の 3 つである．

1. ベクトル $\boldsymbol{x}$ への間接参照（SET 関数）によるランダムアクセス
2. 各行の端数処理（fraction_processing 関数）
3. SIMD レジスタ内の要素を $\boldsymbol{y}$ に足し込む計算が発生（reduction 関数）

はじめに行列サイズが 10^5 の test(m) の 1 行あたりの非零要素数 m を変化させて DD-SpMV を行った結果を図 5.4 に示す．縦軸は倍精度向けの性能，横軸は test(m) の 1 行あたりの非零要素数 m で，スレッド数は 4 である．test(m) では各業の非零要素が連続して配置されていることから性能劣化要因 1. のランダムアクセスの影響はほとんどないと考えられ，性能劣化要因 2. と 3. の影響を評価できる．

Scalar では $m = 1$ の場合を除き，性能はほとんど 7.6 GDFLOPS で一定である．一方で AVX2 は性能が m に従って増加した．端数処理が発生せず十分に m が大きい test(80) の性能は約 53.2

GDFLOPS で，Scalar と比べて約 7.0 倍の性能であるのに対し，端数処理を必要とし，m が小さい test(5) の性能は約 9.5 GDFLOPS で，Scalar と比べて約 1.3 倍の性能である．端数処理や $\boldsymbol{y}$ への足し込みは各行で 1 回発生するため，m が小さいときはほとんどの時間が要素の計算ではなくこれらの処理にかかるためと考えられる．

1 回の浮動小数点演算命令が 1 cycle で処理できると仮定し，Scalar と AVX2 において 1 行計算するために必要な cycle 数について考える．test(5) において Scalar で必要な処理は 5 回の積和演算で，1 回の積和演算は 33 Flops であるため $33 \times 5 = 165$ cycles である．一方で AVX2 では，最初の 4 つの要素を SIMD 命令を用いて同時計算するために 19 cycles, 1 つの端数を計算するために 19 cycles, $\boldsymbol{y}$ への足し込みのために 33 cycles かかり合計は 71 cycles である．AVX2 は 4 つの要素を同時演算しても Scalar と比べて 57 %程度しか cycle 数を減らせていない．端数処理では条件分岐が必要となることや，SIMD レジスタ内の要素を $\boldsymbol{y}$ に足し込むためには一時配列なども必要となることから，AVX2 において 1 行あたりの非零要素数が少ないときは性能向上効果がえにくかったと考えられる．

test(5) における AVX2 を用いた場合の性能は補正ピーク性能の 74.0 GDFLOPS に対し約 13 %で，test(80) のときは約 72 %である．このことから 1 行あたり非零要素数が多く性能劣化要因の影響が少ない問題であれば CRS 形式の DD-SpMV は補正ピーク性能に対して高い性能が得られるが，1 行あたりの非零要素数が少ない行列では性能劣化要因の影響が大きく，性能がほとんど出ないことがわかった．

次に様々な疎行列の非零パターンをもつ問題セット B を用いて AVX2 を用いた CRS 形式の DD-SpMV の性能劣化要因の影響を評価する．ランダムアクセスの影響は各スレッドが計算するタイミングなどにも依存するため正確に評価することは難しい．そこで次の 2 つの方法を用いて 2 つの方法が概ね同じであることを確認することで CRS 形式の SpMV における問題点の影響を評価した．

1 つめはそれぞれのプログラムから性能劣化要因をコメントアウトした次の 3 つのプログラムと比較する方法である．比較用のプログラムであるため答えは合わないが，CRS 形式の DD-SpMV との時間の差を取ることで性能劣化要因における影響時間を見積もれると考えた．

1. 間接参照をなくすために SET を LOAD 命令に変更
2. 端数処理を行わない
3. 行番号に対応する $\boldsymbol{y}$ に対し，reduction 関数を用いて SIMD レジスタ内の要素の足し込みをせずに AVX2 の SIMD レジスタの下位 64 bit を $\boldsymbol{y}$ に格納

2 つめは Intel 社が提供している VTune Amplifier [91] プロファイラを用いる方法である．こ

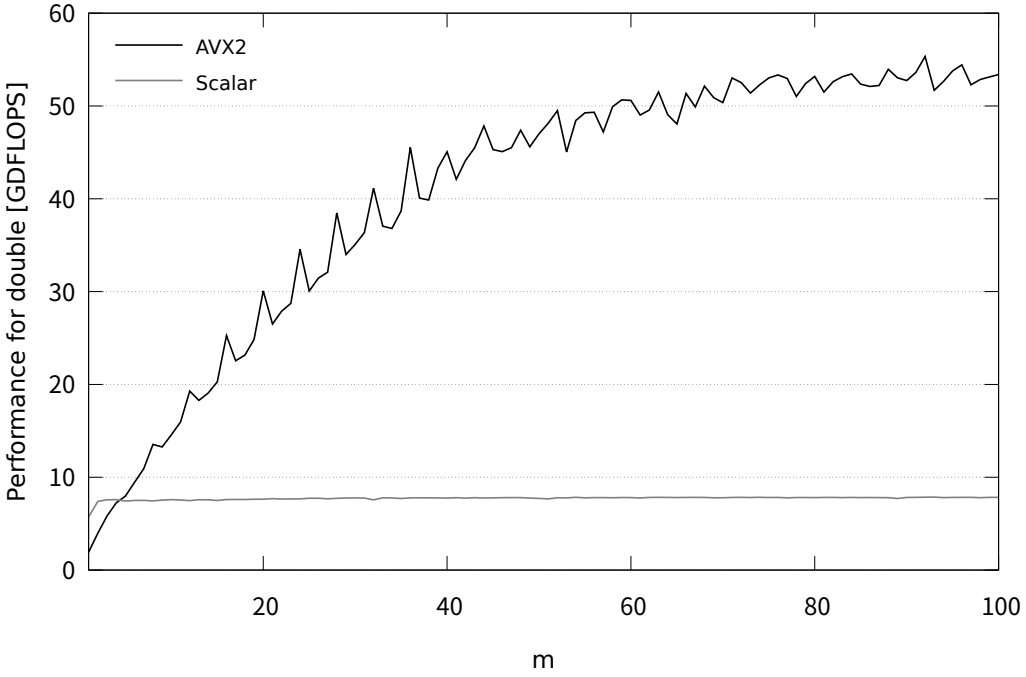

図 5.4: DD-SpMV における非零要素数と性能の関係（4 threads, test(m), N=10^5）

れは関数呼び出しの時間やキャッシュヒット率を取得できる．inline 関数は展開されるため関数呼び出しの時間を取得できないことから，SET や LOAD などの命令を inline 関数から通常の関数呼び出しに変更して利用した．なお，最適化を抑制するコンパイラへの指示句を用いてこれらの関数がコンパイラによって inline 展開が行われないようにした．ランダムアクセスの影響はプロファイラによって取得したそれぞれの行列に対して DD-SpMV を行ったときのキャッシュヒット率の大小関係から見積もった．

これらの 2 つの方法からえられた結果を比較するとそれぞれの内訳や大小関係はほとんど変わらなかったため，ある程度信用できる結果であると考えられる．

図 5.5 に問題セット B に対する AVX2 を用いた 4 スレッドの CRS 形式の DD-SpMV において性能劣化要因が占める全体時間の割合を示す．結果から AVX2 を用いた CRS 形式の DD-SpMV において性能劣化要因は DD-SpMV の実行時間の 60〜90%を占めており性能を大幅に低下させていることが推定できた．

性能劣化要因の影響が特に大きい疎行列は “Dubcova1” と “apache1” で，それぞれ 1 行あたりの平均非零要素数が 15.7 と 6.7 である．一方で性能劣化要因の影響が小さい疎行列は “bcsstk36” や “nasasrb” で，それぞれ 1 行あたりの平均非零要素数が 49.6 と 48.8 である．

性能は非零要素数の配置にも影響されるため 1 行あたりの平均非零要素数のみが性能を決定するわけではないが，1 行あたりの平均非零要素数が小さい疎行列は test(m) の m を変化さ

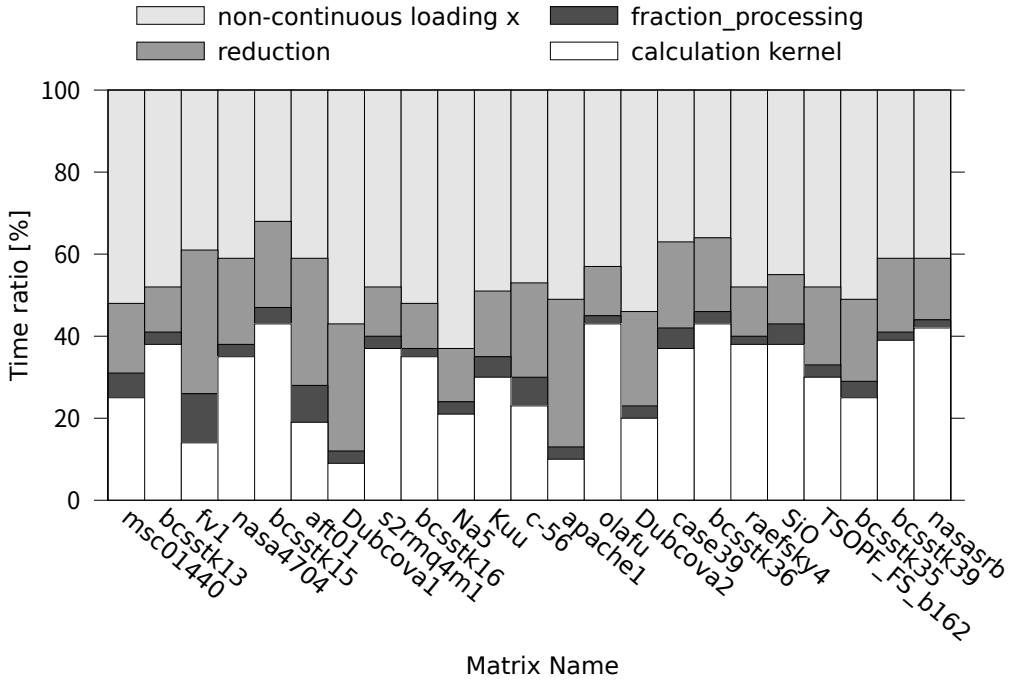

図 5.5: AVX2 を用いた CRS 形式の DD-SpMV の性能劣化要因の影響（4 threads）

せたときの結果と同様に性能がでにくい傾向にあることがわかった．

5.4.4　BCRS 形式の疎行列ベクトル積

BCRS 形式の DD-SpMV の計算時間の内訳

CRS 形式の問題点を解決するため，ブロックサイズを AVX2 の同時演算数である 4 に合わせた BCRS1x4, 4x1 形式を用いて DD-SpMV を行った場合の性能について評価する．

端数処理と SIMD レジスタの要素を $\boldsymbol{y}$ への足し込む処理にかかる時間を評価するため，行列サイズが 10^5 の test(m) において m が 4 の倍数である test(32) と m が 4 の倍数ではない test(33) における AVX2 を用いた DD-SpMV の時間を比較した．表 5.10 に AVX2 を用いた DD-SpMV を CRS，BCRS4x1 形式，BCRS1x4 形式で 4 スレッドで実行したときの時間を示す．

test(32) は BCRS 形式は最初の数行を除いたすべてのブロックが 0 を含まないため，演算量は CRS 形式とほぼ同一になる．test(32) に対する DD-SpMV は DD_ADD_MULT 関数が 8×10^5 回，test(33) では 9×10^5 回発生する．それ以外の計算として，CRS 形式は 10^5 回の端数処理と $\boldsymbol{y}$ の足し込み，BCRS1x4 は 10^5 回の $\boldsymbol{y}$ の足し込みが発生する．

テスト用行列 test(32)，test(33) に対する結果を比べることで次のことが推定できる．

表 5.10: テスト用行列における AVX2 を用いた DD-SpMV の実行時間 [ms] (4 threads, test(m), N=10^5)

	CRS	BCRS1x4	BCRS4x1
test(32)	2.75	2.14	1.88
test(33)	3.15	2.33	2.11

- BCRS1x4 と CRS 形式における test(32) の結果から，メモリアクセスの改善効果は 2.75-2.14=0.61[ms].
- BCRS1x4 形式と CRS 形式における test(32) と test(33) から，DD_ADD_MULT 関数を除いた端数処理 10^5 回のオーバヘッドは (3.15-2.75)-(2.33-2.14)=0.21[ms].
- test(32) における BCRS1x4 形式と BCRS4x1 形式の結果から，$\boldsymbol{y}$ への足し込み 10^5 回にかかる時間は 2.14-1.88=0.26[ms].

この推定は図 5.5 の結果と比べても妥当であると考えられる．BCRS4x1 形式を用いることで連続にアクセスできる構造をもつ疎行列においても端数処理や $\boldsymbol{y}$ への足し込みをなくせるため，test(33) において BCRS4x1 は CRS と比べ 2.75[ms] / 3.15[ms] = 約 87%まで実行時間を短縮した．

BCRS 形式におけるメモリアクセスの影響

BCRS 形式の DD-SpMV におけるメモリアクセスの影響を分析するため，1 行あたり 32 の非零要素をもつ test(32) の行列サイズ N を 10^4 から 4.0×10^5 まで 5000 ずつ変化させた．図 5.6 に，CRS，BCRS1x4，BCRS4x1 形式における DD-SpMV を 4 スレッド行った時間を示す．このとき test(32) は 1.9×10^4 までデータがキャッシュに収まる．

test(m) は行内で非零要素が連続しているため，ベクトル x へのアクセスは 1 度読み込めばキャッシュから再利用できると仮定すると，CPU がメモリからロード，ストアしたデータ量は DD ベクトル $\boldsymbol{x}$，$\boldsymbol{y}$，倍精度の *value*，4 バイト整数型の *col_ind*, *row_ptr* または *col_bind*, *row_bptr* である．

各行列サイズにおける計算時間と理論性能を比較する．本実験環境のメモリバンド幅は 25.6 GB/s であるため，ベクトル，行列の要素，インデックス配列のデータがすべてキャッシュに収まる行列サイズ 10^4 のとき CRS 形式，BCRS4x1 のデータサイズはどちらも約 3.6 [MB]，データがキャッシュに収まらない行列サイズ 4.0×10^5 のとき CRS 形式のデータサイズは 168 [MB]，BCRS4x1 は約 167 [MB] である．

CRS 形式は SIMD レジスタ内の要素を $\boldsymbol{y}$ に足し込む処理が必要になるため，行列サイズを N，非零要素数を nnz とすると DD-SpMV の演算量は $N \times 33 + nnz \times 19$ である．今回用いた test(32) は BCRS4x1 形式によってブロックを作成しても零をほとんど含まず演算量が増加しないため，演算量を $nnz \times 19$ と仮定して実行時間と比較すると次のようになった．

データがキャッシュに収まるとき（N= 10^4），

- CRS 形式は理論上 0.067[ms] で計算できる．実行時間は 0.28[ms] で，理論性能の約 24%．データ転送速度は 3.6[MB] / 0.28[ms] = 12.9 [GB/s] でメモリバンド幅の約 50%．
- BCRS 形式は理論上 0.064[ms] で計算できる．実行時間は 0.18[ms] で，理論性能の約 36%．データ転送速度は 3.6[MB] / 0.18[ms] = 20.0 [GB/s] でメモリバンド幅の約 78%．

データがキャッシュに収まらないとき（N=4.0×10^5），

- CRS 形式は理論上 2.7[ms] で計算できる．実行時間は 10.5[ms] で，理論性能の約 26%．データ転送速度は 168[MB] / 10.5[ms] = 16.0 [GB/s] でメモリバンド幅の約 63%．
- BCRS 形式は理論上 2.5[ms] で計算できる．実行時間は 7.2[ms] で，理論性能の約 35%．データ転送速度は 168[MB] / 7.2[ms] = 23.2 [GB/s] でメモリバンド幅の約 90%．

キャッシュに収まる場合と収まらない場合をそれぞれの格納形式について比較すると，データ転送速度や理論性能の変化は小さく性能は演算器がボトルネックになっていると考えられる．

次にスレッド数を 1, 2, 4 に変化させ，メモリへの要求を増減させて評価を行った．BCRS4x1 において 1, 2, 4 スレッドにおけるキャッシュに収まるときの実行時間はそれぞれ約 0.63[ms], 0.33[ms], 0.18[ms] となった．2, 4 スレッドのマルチスレッド化の効果は 1 スレッドと比べ約 1.9 倍, 3.5 倍である．キャッシュに収まらないときの実行時間はそれぞれ約 7.2[ms], 12.9[ms], 24.5[ms] となった．このとき 2, 4 スレッド並列したことによる高速化効果は 1 スレッドと比べ約 1.9 倍，3.4 倍である．キャッシュに収まる，収まらないにかかわらず並列化の効果はスレッド数の増加に従って陽に増加しており．性能はメモリ性能に制約を受けず，演算器がボトルネックになっていると考えられる．

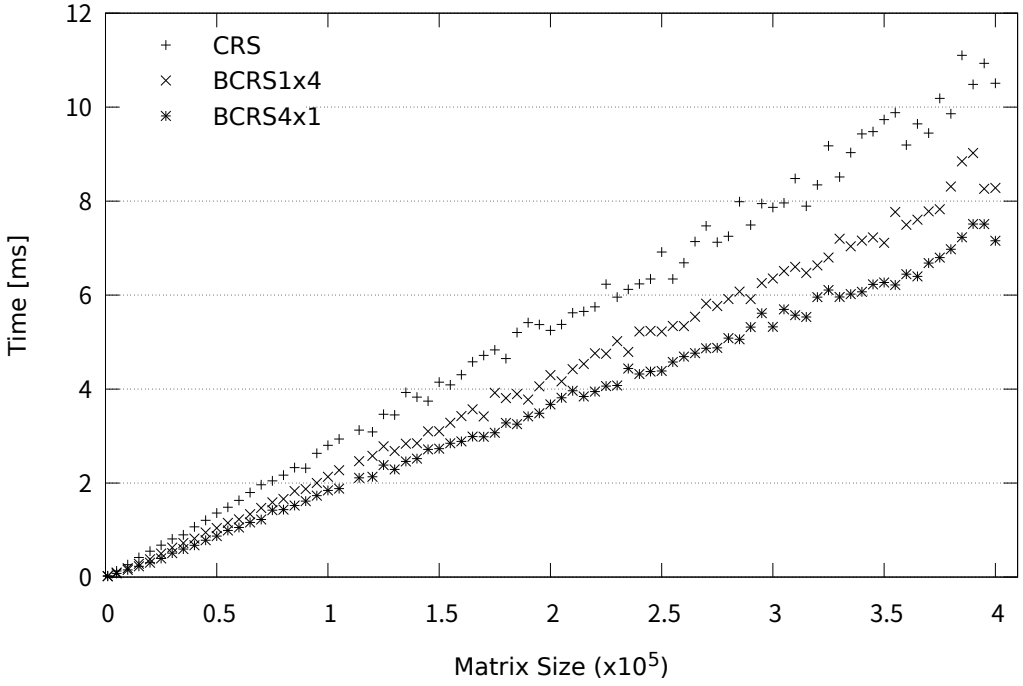

図 5.6: 各格納形式におけるメモリアクセスの影響（4 threads, test(32)）

BCRS4x1 形式の SIMD 化の効果

BCRS4x1 形式の SIMD 化の効率を評価するために問題セット A を対象に比較を行う．

図 5.7 に Scalar の CRS 形式，AVX の CRS 形式，AVX2 の BCRS4x1 において問題セット A に対して DD-SpMV を 4 スレッドで行った結果を示す．縦軸は AVX2 を用いた CRS 形式と BCRS4x1 形式の計算時間の比，横軸は CRS における AVX2 と Scalar の計算時間の比で，値が小さいほど SIMD 化や BCRS4x1 形式を用いたことによる高速化の効果が大きいことを示す．“comp. ratio” は BCRS4x1 と CRS の演算量の比（computation ratio）で，CRS 形式と比べて零要素を含むブロックを作ったことで生じる演算量の増加率を意味し，最大値は 4 である．

CRS 形式を用いた場合における AVX2 の DD-SpMV は Scalar と比べて平均で約 4.86 倍高速である．AVX2 を用いた場合における BCRS4x1 形式の DD-SpMV は CRS 形式と比べて平均で約 2.61 倍高速である．AVX2 にすることで CRS, BCRS4x1 のどちらを用いても Scalar の CRS と比べて遅いのは 100 問題中 2 問のみである．この 2 つの問題は nnz/row が 8 以下かつ行列サイズが 10^5 以下の小さい問題で，各行で SIMD 命令を使った同時処理がほとんど行えないため SIMD 化による並列化効率がでにくく，行列サイズが小さいためメモリアクセスの改善効果もえられなかったと考えられる．

AVX2 を用いた BCRS4x1 形式の DD-SpMV が AVX2 を用いた CRS 形式の DD-SpMV より

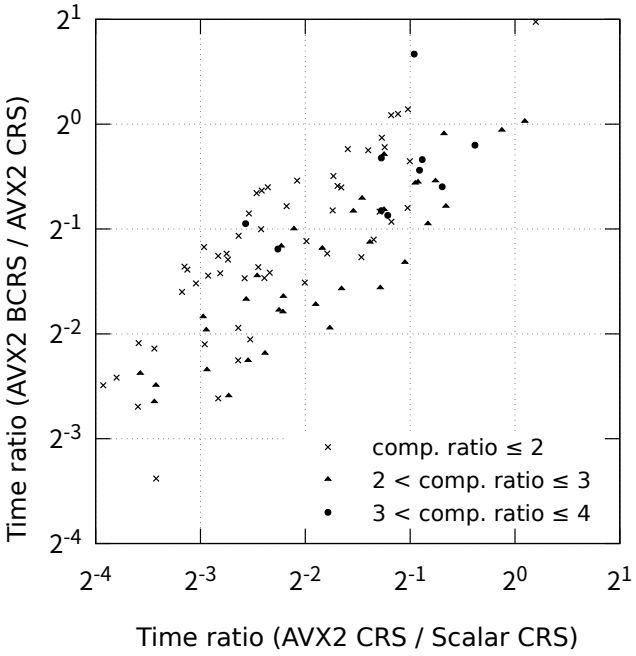

図 5.7: AVX2 を用いた BCRS4x1 形式と CRS 形式の DD-SpMV の 4 スレッドにおける時間の比

も遅いケースは 100 問題中 6 問題である．これらは行列サイズが 10^5 以下の小さい問題で，行列サイズが 10^5 以上の問題であれば BCRS4x1 形式が CRS 形式よりも性能が低いケースはない．

次に，BCRS4x1 形式と BCRS1x4 形式を比較する．BCRS1x4 と BCRS4x1 は作られるブロックの配置が異なるため非零パターンによっては BCRS1x4 のほうが性能がよくなる可能性もあるためである．問題セット A に対して BCRS1x4 と BCRS4x1 で AVX2 を用いた DD-SpMV を 4 スレッドで行ったときの時間の比を図 5.8 に示す．

比較の結果，BCRS4x1 形式の DD-SpMV は BCRS1x4 形式の DD-SpMV と比べて平均で約 1.55 倍高速である．BCRS4x1 形式は 100 問題中 96 問題で高速で，最も遅いケースでも BCRS1x4 形式と比べて約 1.22 倍の時間の増加である．また，BCRS1x4 形式の DD-SpMV が BCRS4x1 形式の DD-SpMV よりも性能が低いのは疎行列のサイズが 10^5 以下の小さい問題のみで，ほとんどの問題で BCRS4x1 形式のほうが BCRS1x4 形式よりも高速に DD-SpMV を行えることがわかった．

次に CRS 形式と BCRS 形式の使い分けの必要性について考える．問題セット A に対して DD-SpMV を 4 スレッドで行った場合における各格納形式の合計時間と，それぞれの行列に対して最適な格納形式を選んだ場合の合計時間を表 5.11 に示す．なお，最適な格納形式はそ

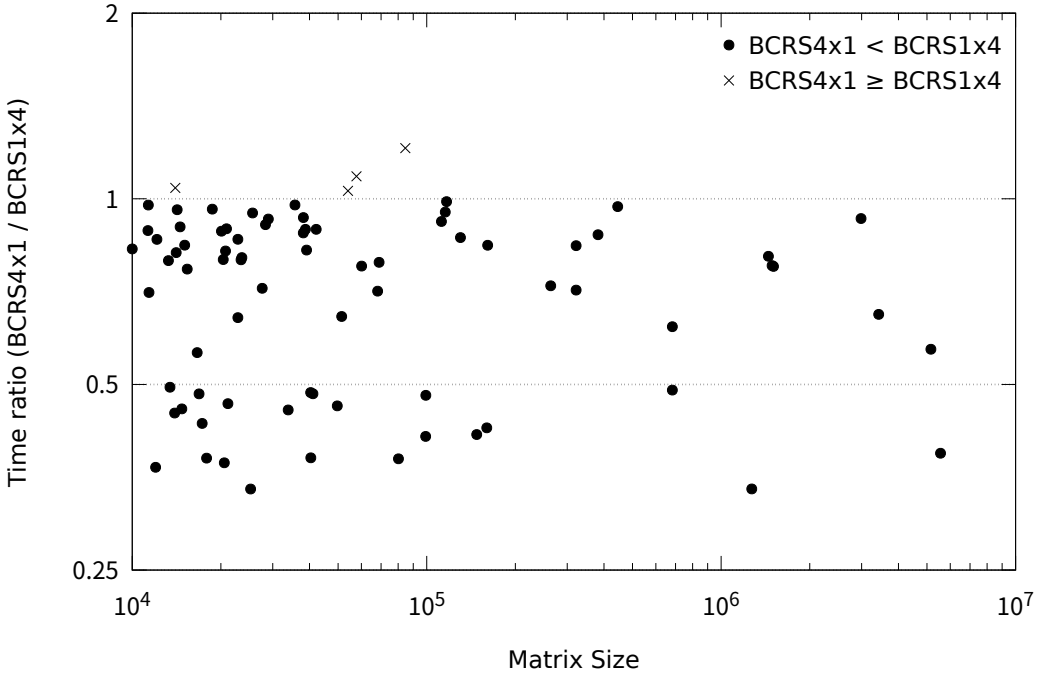

図 5.8: BCRS1x4 形式と BCRS4x1 形式における AVX2 を用いた DD-SpMV の時間の比

れぞれの結果から手動で選んだもので，判別にかかる時間などは含まれていない．

結果から BCRS4x1 のみで計測した場合と最適な格納形式を選んだ場合の合計時間の比は 1.01 と小さく，格納形式を使い分けることの効果は少ない．

5.5 転置疎行列ベクトル積の性能

CRS 形式の DD-TSpMV の性能劣化要因の影響

AVX2 を用いた CRS 形式の DD-TSpMV ではベクトル $\boldsymbol{y}$ への間接参照（SET 関数, scatter 関数）や，各行の端数処理（fraction_processing 関数），スレッド数分の一時配列 *work* の初期化と総和が問題になる．そこで前節と同様の方法で，CRS 形式の DD-TSpMV におけるこれら 3 つの問題点の影響について評価を行った．

図 5.9 に問題セット A に対して AVX2 を用いた 4 スレッドの CRS 形式の DD-TSpMV を行った場合の性能劣化要因が占める全体時間の割合を示す．結果から AVX2 を用いた CRS 形式の DD-TSpMV において性能劣化要因が実行時間の 65〜89%を占めており，性能を大幅に低下させていることが確認できた．

性能劣化要因の中で最も影響が大きいのはベクトル y に対する非連続なストアで 47 %〜69 %である．次に影響が大きいのはスレッド数分の一時配列の初期化と総和計算で 3 %〜13 %

表 5.11: 問題セット A に対する各格納形式の SpMV の合計時間 [ms]（格納形式を最適に組み合わせた場合に対する時間の比）

	Total elapsed time	The number of the best matrices
CRS	730 (1.37)	4
BCRS1x4	880 (1.33)	4
BCRS4x1	520 (1.01)	92
The best combination	510 (1)	100

である．一時配列はスレッド数分必要であるため，コア数の多い環境では初期化や総和計算による影響はさらに大きくなることが予想される．

BCRS (C) 方式の DD-TSpMV

行方向にマルチスレッド化した AVX を用いた BCRS1x4 や BCRS4x1 の DD-TSpMV はスレッド数本の一時配列の初期化や総和が必要になることや，各行における SIMD レジスタ内の総和が必要になることが予想される．

この問題を各スレッドの担当する列の範囲を制限する BCRS4x1 (C) 方式を用いて解決することを考えた．行列サイズ 10^5 の test(32) に対してそれぞれの格納形式で AVX2 を用いた DD-SpMV, DD-TSpMV を 4 スレッドで行った場合の時間を表 5.12 に示す．ここで BCRS4x1 の列方向のスレッド並列化が BCRS4x1 (C) である．

CRS 形式の DD-TSpMV は DD-SpMV の約 2 倍の時間がかかる．行方向にマルチスレッド化した場合，BCRS4x1 形式は CRS 形式と比べて約 3.4 倍遅い．これは SIMD 命令を用いた 4 要素の計算ごとに reduction 関数を用いた SIMD 内のレジスタ要素の足し込みが必要なためである．これに対し列方向にマルチスレッド化した BCRS4x1 (C) 方式の DD-TSpMV は CRS 形式の DD-TSpMV と比べて約 1.78 倍高速で，行方向にマルチスレッド化した BCRS1x4 形式の DD-TSpMV と比べて約 1.22 倍高速である．

また，BCRS4x1 (C) 方式の DD-TSpMV は CRS 形式の DD-SpMV と比べて性能差が小さく約 1.12 倍程度の時間の増加にしかならない．BCRS4x1 はスレッドの計算範囲を制御した BCRS4x1 (C) 方式を用いることで 1 つの格納形式で DD-SpMV，DD-TSpMV の両方を効率よく実行できる．

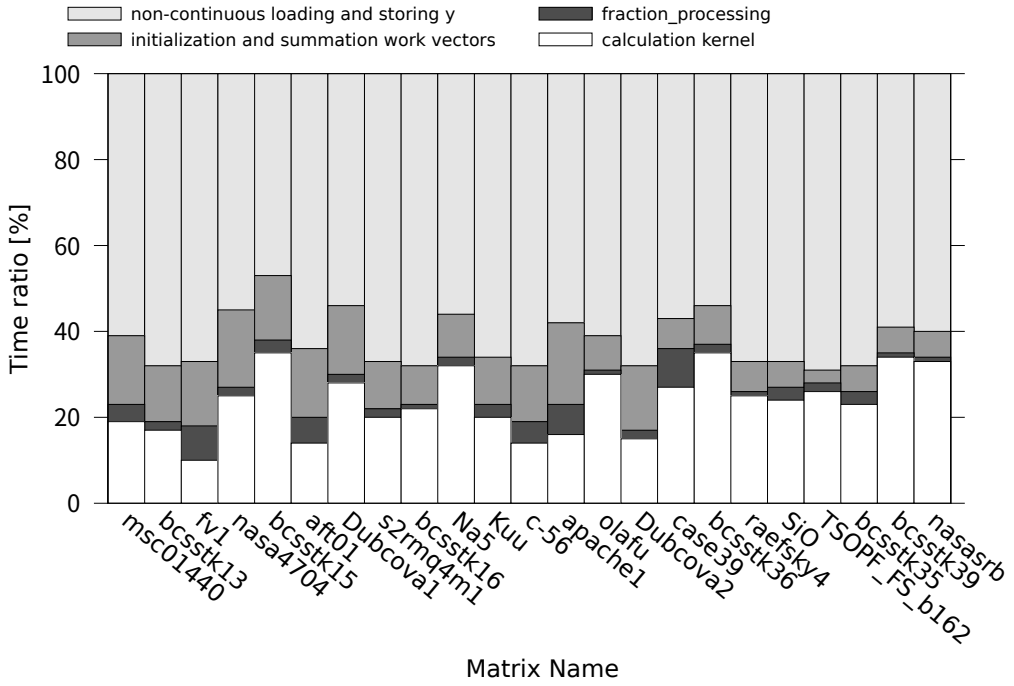

図 5.9: CRS 形式の DD-TSpMV の性能劣化要因の影響（4 threads）

表 5.12: 各格納形式における DD-SpMV, DD-TSpMV の実行時間 [ms]（4 threads, test(32), N=10^5）

	DD=SpMV	DD-TSpMV	
		行方向のスレッド並列化	列方向のスレッド並列化
CRS	2.14	3.97	4.31
BCRS1x4	2.02	2.94	4.91
BCRS4x1	1.74	13.31	2.41

図 5.10 に Scalar の CRS 形式，AVX の CRS 形式，AVX2 の BCRS4x1 において問題セット A を対象に DD-TSpMV を 4 スレッドで行った結果を示す．縦軸は AVX2 を用いた CRS 形式と BCRS4x1 形式の計算時間の比，横軸は CRS における AVX2 と Scalar の計算時間の比で，値が小さいほど SIMD 化や BCRS4x1 形式を用いたことによる高速化の効果が大きいことを示す．“comp. ratio” は BCRS4x1 と CRS の演算量の比である．

CRS 形式を用いた場合における AVX2 の DD-TSpMV は Scalar と比べて平均で約 5.28 倍高速である．AVX2 を用いた場合における BCRS4x1 形式の DD-TSpMV は CRS 形式と比べて平均で約 5.91 倍高速である．AVX2 にすることで Scalar より遅くなった問題はなく，BCRS4x1 を用

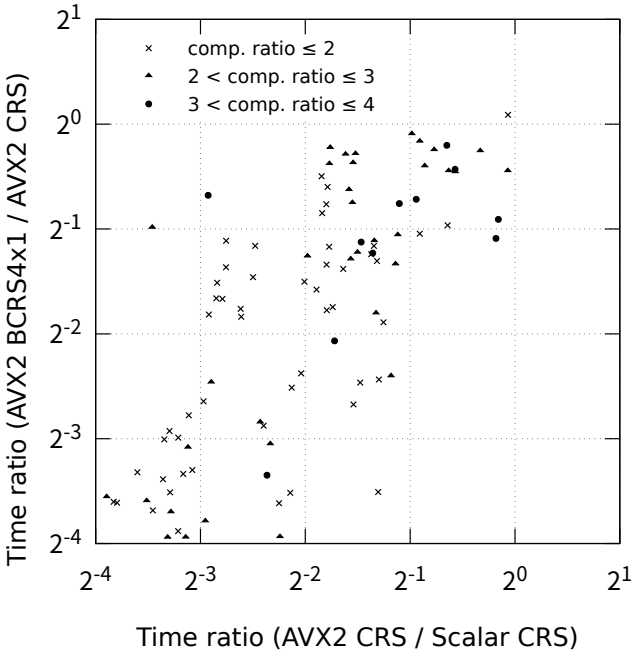

図 5.10: AVX2 を用いた BCRS4x1 形式と CRS 形式の DD-TSpMV の 4 スレッドにおける時間の比

いたことで CRS 形式よりも遅くなるケースは 1 問である．これは SpMV においても BCRS4x1 化の効果のなかったサイズ 10^4 程度の小さい問題で，サイズが小さいためメモリアクセスの改善効果が発揮されにくいためと考えられる．

以上の結果より BCRS4x1 はアクセスパターンを変更するだけで SpMV, TSpMV の両方に対して SIMD 化やマルチスレッド化の問題点を解決できることから，1 つの格納形式で SpMV と TSpMV を効率的に計算できることがわかった．

5.6　BiCG 法の性能

前節までで述べた基本演算の実装をバックエンドとしたソフトウェア ; DD-AVX v3 (`https://github.com/t-hishinuma/DD-AVX_v3`) を開発した．本節では DD-AVX v3 を用いて Krylov 部分空間法の 1 つである BiCG 法を実装した際の性能を評価する．

BiCG 法のアルゴリズムをアルゴリズム 11 に示す．BiCG 法は 1 反復あたり axpy を 3 回，xpay を 2 回，dot を 2 回，nrm2 を 1 回，SpMV を 1 回，TSpMV を 1 回実行する．また，付録 B に DD-AVX v3 を用いて実装した BiCG 法のプログラムを載せた．付録のプログラムに対し変数宣言の変更および CRS 形式から BCRS4x1 への変換を追加することで DD の BiCG 法を

実装した．

アルゴリズム **11** BiCG 法のアルゴリズム

1: Set an initial guess $\boldsymbol{x}_0$
2: Compute $\boldsymbol{r}_0 = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_0$
3: Set an arbitary vector $\boldsymbol{r}^*_0$ s.t. $(\boldsymbol{r}_0, \boldsymbol{r}^*_0) \neq 0$
4: Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$, $\boldsymbol{p}^*_0 - \boldsymbol{r}^*_0$
5: **for** i = 1, 2, 3,... **do**
6: $\quad \alpha_i = (\boldsymbol{r}^*_{i-1}, \boldsymbol{r}_{i-1})/(\mathbf{A}\boldsymbol{p}_{i-1}, \boldsymbol{p}^*_{i-1})$
7: $\quad \boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha_i \boldsymbol{p}_{i-1}$
8: $\quad \boldsymbol{r}_i = \boldsymbol{r}_{i-1} - \alpha_i \mathbf{A}\boldsymbol{p}_{i-1}$
9: $\quad \boldsymbol{r}^*_i = \boldsymbol{r}^*_{i-1} - \alpha_i \mathbf{A}^T \boldsymbol{p}^*_{i-1}$
10: $\quad$ **if** $(||\boldsymbol{r}_i|| \leqq \epsilon)$ **then break**
11: $\quad \beta_i = (\boldsymbol{r}_i, \boldsymbol{r}^*_i)/(\boldsymbol{r}_{i-1}, \boldsymbol{r}^*_{i-1})$
12: $\quad \boldsymbol{p}_i = \boldsymbol{r}_i + \beta_i \boldsymbol{p}_{i-1}$
13: $\quad \boldsymbol{p}^*_i = \boldsymbol{r}^*_i + \beta_i \boldsymbol{p}^*_{i-1}$
14: **end for**

BiCG 法を実装したときの DD の基本演算の性能を評価するため，AVX2 を用いた 4 スレッドにおける DD の CRS 形式および BCRS4x1 形式の BiCG 法の性能，AVX2 を用いた 4 スレッドにおける倍精度の CRS 形式の BiCG 法の性能を比較した．問題セット B に対し BiCG 法を 100 反復した時間を図 5.11 に示す．

実験は反復回数を 100 反復に固定し，繰り返し部分のみを計測した．初期値の設定などの時間は含んでいない．反復回数を固定するのは倍精度と DD で反復回数が異なるため求解までの時間では倍精度と比較した性能評価がしにくいためである．また，高精度演算による収束改善効果は既に文献 [8–10, 16, 21, 22] で評価されているため，本論文では収束改善効果については議論しない．

結果から DD の BCRS4x1 形式における計算時間は倍精度と比べて 1.1〜1.3 倍で，倍精度と比べて DD の演算を実用的な時間で計算できることが確認できた．

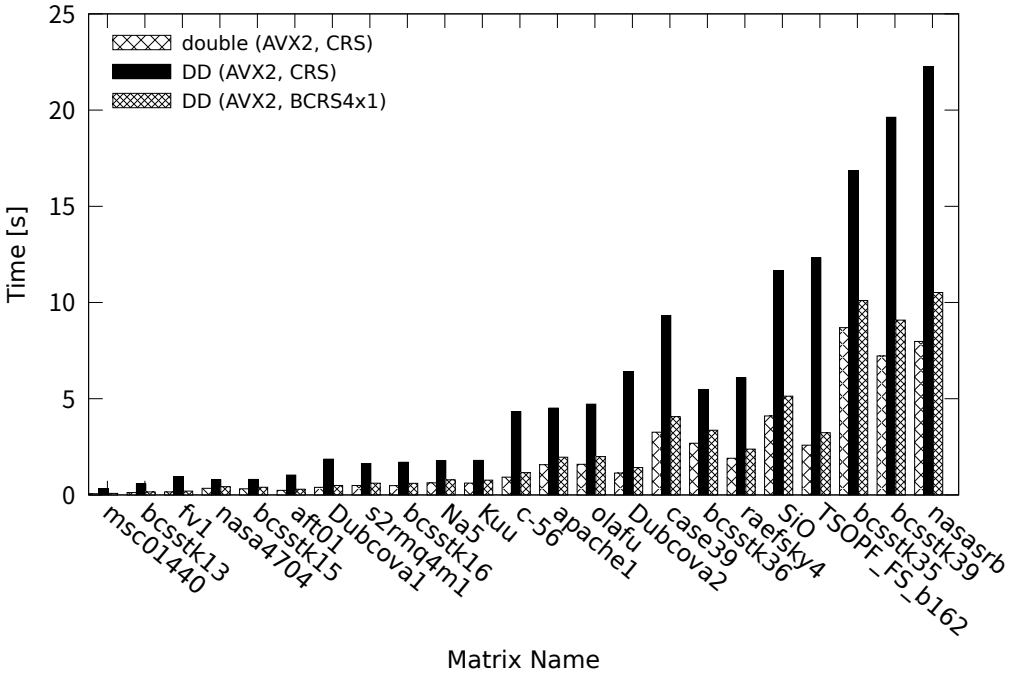

図 5.11: BiCG 法 100 反復の実行時間 [s]

第6章　結論

6.1　まとめ

連立一次方程式の近似解を求める手法である Krylov 部分空間法は丸め誤差により反復回数が増大したり，近似解が求まらなくなったりする．Krylov 部分空間法はスカラ，ベクトル，疎行列に対する演算の組み合わせによって構成され，最も時間がかかる演算は疎行列ベクトル積（SpMV）および転置疎行列ベクトル積（TSpMV） である．収束を改善するために高精度演算を用いる手法では，計算時間やメモリデータ量が多く必要となることや標準的なプログラミング環境では高精度演算を行うためのソフトウェアが少ないことが問題となる．

現実的な計算時間で高精度演算を行うために倍精度と比べて 2 倍の精度である 4 倍精度に着目した．4 倍精度演算の実現手法の 1 つである Double-Double 精度（DD）演算は 2 つの倍精度数を用いて 4 倍精度相当の演算を実行する手法で，倍精度演算 10〜20 回を組み合わせることで実行できる．

本研究はマルチコア CPU において Krylov 部分空間法の実装に必要な DD の基本演算を高速化することを目的とした．具体的な高速化のアプローチとして，代表的な Krylov 部分空間法の実装に必要な DD の基本演算を現在のほとんどの汎用プロセッサでサポートされている FMA（Fused Multiply-Add）命令，SIMD（Single Instruction Multiple Data Streaming）命令，マルチスレッドを用いた．高速化した演算を用いて倍精度と DD を組みわせてこれらの基本演算を扱えるソフトウェアを開発した．

はじめに 1.2 節において疎行列を係数行列とした Krylov 部分空間法を倍精度と DD を組み合わせて実装できるソフトウェアを開発するために必要な機能を検討した．対象とする演算は CG 法，BiCG 法，GMRES 法に必要なスカラ，ベクトル，疎行列に対する演算（基本演算）とし，これらに対する高速化を検討した．

マルチコア CPU において基本演算を高速化している先行研究として，文献 [15] では，疎行列に対する DD の計算を同時処理数が 2 の SIMD 命令を用いて高速化している．一方で現在主流である同時処理数が 4 または 8 つの SIMD 命令を用いて疎行列に対する DD の計算を高速化している研究は確認されていない．また，メモリデータ量や計算量の削減のために倍精度型と DD 型を組み合わせて使う研究 [11, 16] なども行われている．

そこで本研究では，文献 [15] で用いられている SIMD 命令の 2 倍の同時処理数をもつ同時処理数 4 つの SIMD 命令を用いて基本演算の高速化を行った．また，将来的に同時処理数が 8 つ以上の SIMD 命令に対してソフトウェアの拡張を行うことを考え，ソフトウェアの設計においては SIMD 命令の同時処理数の変更によるプログラムの修正が少なくできるソフトウェアの構成について検討した．

高速化した基本演算に対して倍精度型と DD 型を組み合わせて入力することでユーザが簡単に高速化 Krylov 部分空間法を実装できるようにするために，次の 3 つのソフトウェア要件を定めた．

要件 1 Krylov 部分空間法に必要なスカラ，ベクトル，疎行列に対する演算があること

要件 2 DD のスカラ，ベクトル，疎行列に対する演算がマルチコア CPU で高速に実行できること

要件 3 倍精度と DD の混合精度演算が実装できるインタフェースをもつこと

2 章では関連研究と比較した上で本研究の意義を明らかにした．マルチコア CPU において SIMD 命令を用いた DD の疎行列に対する演算の高速化は同時処理数が 2 つの SIMD 命令でしか行われておらず，倍精度演算とは演算にかかる時間の内訳（性能特性，ボトルネック）が明らかになっていないことや，倍精度と DD を組み合わせて DD の疎行列に対する演算を扱う研究 [58] は行われているが，この研究では並列化などの高速化は行われていないことから，DD の疎行列に対する演算の性能特性を明らかにし，性能特性に基づいてマルチコア CPU の高速化手法の効率的な利用方法を提案することや，高速化された基本演算の関数への入力として倍精度型と DD 型を組み合わせて扱うことができるソフトウェアのインタフェースを定めることは，将来のアーキテクチャでも使える高速かつ高精度な物理シミュレーションを行うための基盤技術として有用である．

3 章ではユーザにソフトウェアをどのように提供するかについて検討した．近年，低速だが高機能なインタプリタ型のプログラミング言語から，高速な C, C++, Fortran などの言語で実装された基本演算を呼び出して利用するケースが増加している．数値計算に特化した Rust, Julia，MATLAB のような言語では BLAS などを核としたベクトル型や疎行列型が定義されており，高速な数値計算のプログラムを簡単に実装できる．これらのプログラミング言語では C, C++, Fortran のライブラリを読み込んだり，インタフェースを自動生成する機能がある．

これらのことから Rust, Julia, MATLAB などのベクトル型，疎行列型の機能を参考に，これらの型がもつ機能に合わせた機能を C++を用いて倍精度と DD のベクトル，疎行列クラスを定義し，ユーザには C++から変換された Rust, Julia, MATLAB などのインタフェースを提供

することにした．クラスのメンバ関数としてスカラ，ベクトル，疎行列に対する演算を定義した．

倍精度と DD を組み合わせて扱うためのインタフェースを考えると，“型名+演算名” のような関数名の変更による精度の切替方法を採用することは非現実的であると考えられる．精度の指定はクラスの宣言時に行うことが妥当と考え，宣言部分を変更することで全体の処理が内部で切り替わる機能を実現するため，C++のオーバーロードやテンプレート機能を用いることで関数名は変更せず，変数の宣言を変えるだけで精度を切り替えられるインタフェースを構築することにした．

4 章では FMA 命令，SIMD 命令，マルチスレッドの概要および DD 演算のアルゴリズムについて述べたうえで，これらの高速化手法を用いた DD の基本演算の高速化方法について検討し，5 章で性能評価を行った．

はじめに SIMD 化を行う方針として，将来的にソフトウェアの実装を行うことを考え，ソフトウェアの階層を基本演算の実装を行う階層と SIMD 化された四則演算などをおこなう階層にわけて実装を行うことにした．これにより異なる同時処理数の SIMD 命令に対応する場合でもプログラムの変更や追加を限定的なものにできる．

これまでの DD 演算を高速化する研究 [13–15, 49, 57] では，DD の性能を評価するとき実行時間や性能を倍精度の結果やピーク性能と比較している．一方，DD 演算は倍精度加算と乗算の数に偏りのあるアルゴリズムである．FMA 演算器を有する CPU のピーク性能は FMA 演算を毎サイクル行った場合として定義されているため DD 演算ではピーク性能がでない．

本研究では倍精度加算，乗算，FMA の数から実際に期待できる DD 演算のピーク性能を再定義して性能を評価することを提案し，この性能を補正ピーク性能と定義した．補正ピーク性能により実際にハードウェアの効率をどの程度引きだせたかをより厳密に評価できた．

また，DD のベクトルや疎行列に対する演算では DD の配列の保持方法に AoS 型と SoA 型の 2 種類が考えられる．SIMD 命令を用いてこれらの配列に対してアクセスすることを考えたとき AoS 型は連続した *hi*, または *lo* 要素へのアクセスができないため，load 命令を用いられず性能が低下する可能性が高い．そこで *hi* または *lo* に SIMD 命令を用いて連続してアクセスできる SoA 型を用いて DD の配列を保持することにした．

DD のベクトル演算はキャッシュにベクトルデータがすべて収まる場合は SIMD 命令，FMA 命令の利用やマルチスレッドによる高い性能が発揮でき，すべてのケースで補正ピーク性能の 80 %以上となった．一方でキャッシュにベクトルデータが収まらないサイズでは性能がメモリ性能に制約を受け，性能は補正ピーク性能とくらべて約 12.2 %程度まで減少した．

DD の性能は，同様にメモリ性能に制約を受ける倍精度の性能の約半分でメモリ帯域の利用効率は 86〜98%だった．DD は倍精度とのメモリデータ量の比が 2 倍であることやメモリ帯

域の利用効率から，これらはメモリ性能を限界まで引きだせている．

DD ベクトル演算の結果から，多くの計算時間が必要と考えられる SpMV，TSpMV においてメモリへのデータ要求量の削減を検討することが必要になった．多くの Krylov 部分空間法では行列の値は書き換えられないことが多いため，行列は倍精度としても丸め誤差の影響はない．これにより行列を DD でもつ場合と比べてメモリデータ量を半分にでき，疎行列ベクトル積のメモリへのデータ要求量を約 6 %削減できる．

そのため本論文では対象とする疎行列ベクトル積や転置疎行列ベクトル積を行列の要素を倍精度，ベクトルを DD とした $y_{DD} = A_D x_{DD}$（次，DD-SpMV）および $y_{DD} = A_D^T x_{DD}$（次，DD-TSpMV）とした．

一般的に疎行列を表現するためのデータ表現には零要素を記憶しない形式を用いる．疎行列の一般的な格納形式である CRS 形式を検討した結果，SIMD 命令を用いて 4 つの要素を連続して扱いにくく，いくつかの処理が必要なことがわかった．ベクトル $\boldsymbol{x}$ へのランダムアクセス，各行におけるベクトル $\boldsymbol{y}$ への要素の足し込み，各行における 4 つ同時に演算できないことによる余りの計算（端数処理）である．

CRS 形式の DD-SpMV のメモリアクセス性能の評価を各行に 32 個の非零要素を連続して配置したテスト用の疎行列を用いて行った結果，テスト用の疎行列のサイズを大きくしても性能はメモリ性能に制約を受けず，SIMD 化によって約 3.4 倍に性能が向上した．

また，キャッシュに収まらないサイズのテスト用の疎行列において，行列サイズを固定して各行の非零要素の数を変化させた結果，各行の非零要素数が 5 のときの性能は補正ピーク性能に対し約 13 %で，各行の非零要素数が 80 のときは約 72 %であった．

疎行列のサイズを大きくしても性能が低下しないことや，各行の非零要素数が十分に大きいとき，補正ピーク性能と比べて高い性能が出せていることから，行列を倍精度に変えてメモリへのデータ要求量を削減するアプローチが有効であることがわかった．

一方で各行における非零要素が少ない行列では，計算時間に対してベクトル $\boldsymbol{y}$ への要素の足し込みや端数処理の影響が大きく，性能がほとんど出ないことがわかった．

SuiteSpase Matrix Collection から取得した様々な非零パターンの 23 の問題に対して CRS 形式の DD-SpMV を実行すると，性能はテスト用の疎行列と比べて約半分程度まで低下した．プロファイラによって分析した結果や計算結果は合わなくなるが非連続なロード，ストアを連続したロード，ストア命令に置き換えたプログラムと比較した結果と比較することで性能劣化要因による影響を見積もると，SpMV の実行時間の 60〜90%，TSpMV の実行時間の 65〜89%がこれらの処理によるものであることがわかった．

CRS 形式における性能劣化要因の影響が大きいことから，4 つのデータを連続して処理できる格納形式である BCRS（Block CRS）形式に着目した．BCRS 形式は零要素を含むサイズ

$r \times c$ のブロックを CRS 形式のように行方向に格納する格納形式で，ブロック内の要素に対して連続したアクセスを行えるが，ブロック内に零要素を含めるため演算量やデータ量が最大で $r \times c$ 倍に増加する．ブロックのサイズが SIMD 命令の同時演算数である 4 ($r \times c = 4$) に合わせることで，SIMD 命令を用いて必ず 4 つの要素を連続して扱うことを考えた．

ブロックのサイズが SIMD の同時演算数である 4 ($r \times c = 4$) になるのはブロックサイズ BCRS1x4，4x1，2x2 の 3 通りである．これらの特徴を比較検討した結果，CRS 形式の性能劣化要因を改善することのできるブロックサイズは 1x4，4x1 であることがわかった．

SuiteSparse Matrix Collection から取得したサイズや非零パターンの異なる 100 問題を用いて BCRS4x1 形式の DD-SpMV の性能を評価した．BCRS4x1 形式は CRS 形式と比べて平均で約 2.61 倍，BCRS1x4 形式と比べて約 1.55 倍高速という結果をえた．BCRS4x1 形式が CRS 形式や BCRS1x4 形式と比べて遅いケースは，行列サイズが小さくベクトルがキャッシュに収まるためランダムアクセスの影響を受けにくい問題だった．

100 問題のうち BCRS4x1 形式が最適なものが 92 問，CRS 形式が最適なものが 4 問，BCRS1x4 形式が最適なものが 4 問であった．それぞれの行列に対し最適な格納形式を使い分けた場合，100 問の SpMV を 510 ミリ秒で実行できる．これに対し BCRS4x1 形式のみで実行した場合は 520 ミリ秒で実行できる．最適に使い分けた場合との比は約 1.01 倍しかなく BCRS4x1 形式を他の格納形式と使い分ける必要はないと考えられる．

CRS 形式の TSpMV ではベクトル $\boldsymbol{y}$ への書き込みは行番号，ベクトル $\boldsymbol{x}$ への読み込みは列番号に依存する．一方，CRS 形式の TSpMV では $\boldsymbol{x}$ と $\boldsymbol{y}$ のアクセスパターンの関係が逆になるため，$\boldsymbol{y}$ への書き込みが列番号に依存し，各スレッドが列番号を示すインデックス配列に対応したベクトル $\boldsymbol{y}$ に同時に書き込みを行うためスレッドセーフにならないため，スレッド本数分の一時ベクトルを確保する必要があり，CRS 形式の DD-TSpMV は DD-SpMV の約 2 倍の時間がかかる．

この問題に対し各スレッドの担当する列の範囲を制限することで解決することを考え，BCRS4x1 形式の各スレッドの担当する列の範囲を制限し，列方向に対してマルチスレッド化する BCRS4x1(C) 方式を実装した．

DD-SpMV と同様の 100 問題を用いて DD-TSpMV の性能を評価した．これにより BCRS4x1(C) 方式は CRS 形式と比べて 5.91 倍高速という結果をえた．BCRS4x1(C) 方式の DD-TSpMV は CRS 形式の SpMV と比べて 1.1 倍，BCRS4x1 形式の SpMV と比べて 1.4 程度の時間の増加で計算できる．これらのことから BCRS4x1 形式はアクセスパターンを変更した実装を用いることで高速に SpMV と TSpMV を計算できることを明らかにした．

これらの基本演算の実装を核としたソフトウェアとして DD-AVX v3 を開発した．開発したソフトウェアを用いれば BiCG 法の計算時間を倍精度と比べて 1.1〜1.3 倍の時間で 4 倍精度

で実行できる．

また，今回開発したソフトウェアは SIMD 命令に依存したプログラムと基本演算のプログラムを分離できており，ベクトルや疎行列に対する基本演算のプログラムにハードウェア依存の SIMD 命令は登場しない．そのため他のハードウェアに対応させる際は DD の基本操作のみを変更すれば良いと考えられ，将来的に他のハードウェアでプログラムを動作させるうえでのハードルは低いと考えられる．

これらの結果から Krylov 部分空間法に必要な演算の中で最も処理の時間がかかる疎行列に対する演算を並列化するうえでの性能特性，および性能特性に基づいた SIMD 化，マルチスレッド化したことによる高速化の方法が明らかになった．CRS 形式の SpMV，TSpMV は間接参照による影響や，4 つの要素を同時に計算できないことの影響が大きく SIMD 化に不向きである．これに対して SIMD 命令の同時処理数にあわせてブロック化した BCRS4x1 形式は CRS 形式の問題点を解決し SpMV, TSpMV を高速に実行できることを示した．

これらの基本演算を倍精度と DD を組み合わせて簡単に扱うことができるソフトウェアを開発した．SIMD 化されたカーネル演算を用いて倍精度，DD のスカラ，行列，疎行列クラスを定義し，これらを組み合わせて入力できるインタフェースを C++を用いて構築したことで，変数の宣言を変更するだけで問題の性質に合わせて Krylov 部分空間法の変数や演算の簡単に精度を切り替えられる．

本研究では，現在一般的なプロセッサに搭載されている FMA 命令，SIMD 命令，マルチスレッドを用いて DD の疎行列演算の演算特性を明らかにし，これらに基づいた実装により高速が可能であることを示した．これらを扱うソフトウェアのインタフェースを定義したことで高速化された演算を少ないプログラムの変更で簡単に扱うことができる．本研究で提案した SIMD 命令に合わせてデータをフィッティングさせることで高速化を行うアプローチやそれらを利用するためのソフトウェアインタフェースは，汎用的かつ実用的な高精度 Krylov 部分空間法の実現を可能にできるという点で数値計算の基盤技術として有用である．

6.2　今後の課題

実験の結果から今回のシステムにおいて DD 演算は十分に高速化できており，基本演算に対する高速化の余地はあまりないと考えられる．そのため今後の課題はより大規模な問題を扱うための分散メモリ環境への拡張や，今回実装した基本演算の実装をより多くの環境に適用させていくことが挙げられる．

開発したソフトウェアを分散メモリ環境でも動作するように拡張することを考えると，マルチコア CPU において十分に高速化されていることから，高速化の観点ではあまり問題は生

じないと考えている．一方で実装面では MPI (Message Passing Interface) の拡張が挙げられる．DD のデータを分散メモリ環境向けのプログラムによって通信することが必要になるため，今回開発した DD のスカラやベクトルを入力できる MPI の拡張が必要となると考えられる．

また，将来のアーキテクチャに対してソフトウェアを対応していくことは必要である．近年，Intel から SIMD 命令の同時処理数が 8 つの AVX512 [52] が登場した．本研究ではソフトウェアの設計において SIMD 命令を用いたプログラムをカーネル関数として基本演算と分離して実装したため，プログラムの書き換えを限定的にでき，ソフトウェアを対応させることはあまり難しくないと考えられる．

性能面では，SIMD 長が長くなった場合は CRS 形式の性能劣化要因による影響はレジスタ内の要素の足し込みに必要な演算数や端数処理が発生する頻度が増加するため，より顕著な問題になることが予想できる．一方で今回提案した BCRS 形式を用いる場合，SIMD 長が伸びればブロックサイズを大きくしなければならないため演算量やメモリデータ量の増加も大きくなる．

そのため問題に応じた使い分けが必要になる可能性がある．演算の増加量は BCRS を生成する段階で判定が可能なため，演算の増加量に応じて生成時に判定して BCRS を作らないなどの工夫を行えば将来のアーキテクチャにおいても十分に適用可能であると考えられる．

謝辞

本研究を遂行するにあたり，多くの方々からご協力やご指導をいただきました．心から感謝の意を表します．

筑波大学図書館情報メディア系 長谷川秀彦教授には，学部 4 年次より 8 年間にわたり温かいご指導ご鞭撻をいただきました．学外との共同研究や国内外の会議など，多岐にわたる挑戦の機会を与えていただきました．重ねて心から感謝申し上げます．

指導教員である筑波大学図書館情報メディア系 森継修一教授には，ご多忙のなか指導教員をお引き受けいただき，熱心かつ親身なご指導だけでなく，研究生活における様々なご支援をいただきました．

副指導教員である筑波大学図書館情報メディア系 佐藤哲司教授，松本紳教授には，鋭いご指摘，ご助言とともに温かい励ましのお言葉をいただきました．

審査委員を引き受けてくださった筑波大学図書館情報メディア系 中井央准教授，阪口哲男准教授，および同大学 計算科学研究センター 高橋大介教授には，本学位論文の審査を通じて研究の立ち位置や評価方法，用語法など様々な角度から詳細かつ的確なご助言をいただきました．

工学院大学の田中輝雄教授，藤井昭宏准教授，元工学院大学の小柳義夫教授（現高度情報科学技術研究機構 サイエンスアドバイザー）には，学士，修士課程において温かいご指導ご鞭撻をいただきました．博士課程への進学後もご指導ご鞭撻をいただき，公私ともに様々なご助言をいただきました．

東北大学 滝沢寛之教授，同大学 平澤将一先生（現国立情報学研究所特任研究員）には，本研究で計算結果の検証に使用した Xev-GMP の開発において，様々なご支援とご助言をいただきました．

本研究を遂行するにあたり，国内外の会議を通じてお付き合いいただいた様々な先生方から沢山の知識やご助言をいただきました．特に静岡理工科大学 幸谷智紀教授，理化学研究所 今村 俊幸先生，同研究所 椋木大地先生，会津大学 中里直人教授，高エネルギー加速器研究機構 石川正准教授，湯浅富久子教授からは学部 4 年次の頃より多倍長計算の現状や応用など様々なことをご教授いただきました．

理化学研究所 中田真秀先生には個人的にもお付き合いいただき，年末年始にお願いしたに

も関わらず本論文への的確なご指摘をいただきました．また，中田先生には会社での 4 倍精度ライブラリ開発の研究開発においても共同研究という形でご指導いただき，いくつかの会議で発表する機会を与えてくださいました．

また，本研究の遂行にあたり，会社の皆様には多くのご支援や支えをいただきました．本研究で行ったソフトウェア開発における基礎知識は，会社で進めてきた研究開発によって培われたものです．

株式会社科学計算総合研究所 代表取締役社長の井原遊氏および株式会社 PEZY Computing 元代表取締役社長 齊藤元章氏，同社 代表取締役 高橋一夫氏，株式会社 ExaScaler 代表取締役社長 木村耕行氏，同社 元 CTO の鳥居淳氏には，研究開発における議論やご指導だけでなく論文執筆のための休暇等の調整など全面的なサポートをしていただきました．

株式会社科学計算総合研究所の五十嵐亮氏，寺村俊紀氏，堀江正信氏，森田直樹氏には，ソフトウェア開発に関するご助言だけでなく，私生活においても様々なご支援をいただきました．特に五十嵐亮氏と森田直樹氏には本論文への的確なご指摘をいただきました．

工学院大学の先輩であり，株式会社 PEZY Computing の先輩でもある坂本亮氏には，大学では並列計算の基礎，会社ではソフトウェア開発の基礎について丁寧にご指導いただきました．特に深夜 1 時過ぎにアセンブリコードを送り，どこがボトルネックか電話で質問したことに朝までお付き合いいただいたこと，この場を借りて感謝とお詫び申し上げます．

株式会社 PEZY Computing の田中英行氏，中村孝史氏，黒澤範行氏，山浦優気氏他，ソフトウェア開発部の皆様にはソフトウェア開発に関するご助言だけでなく私生活においても様々なご支援をいただきました．

筑波大学 長谷川研究室の太田凌氏，同大学 佐藤研究室の山本修平氏，伏見卓恭氏には，研究に関する議論だけでなく筑波大学における研究生活を支えていただきました．

工学院大学情報学部 先進ソフトウェア研究室および高性能計算研究室の皆様には本研究について様々なご議論，ご指摘をいただきました．特に共著者として論文を執筆してくださった浅川圭介氏，佐藤真之介氏，佐々木信一氏，丸地賢氏，花上直樹氏，榊原巧磨氏，斯波柾氏，愛沢菜穂氏，土肥樹氏，伊藤友太氏からは数多くの知見，刺激を受けました．

工学院大学の同期である丸山拓也氏，日永田和嗣氏は，友人として研究が行き詰まったときなど苦しいときに叱咤激励していただき，私の研究を推進する上で大きな原動力となる心強い存在でした．

最後に，これまで一貫して暖かく応援してくれた両親と家族に心から感謝し，研究生活および本論文の校閲に全面的に協力してくれた妻 菱沼香に本論文を捧げます．

参考文献

[1] 寒川光，藤野清次，長嶋利夫，高橋大介, “HPC プログラミング,” pp. 2–23, オーム社, 2009.

[2] R. Barrett, M. W. Berry, T. F. Chan, J. W. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, “Templates for the solution of linear systems: building blocks for iterative methods,” pp. 55–65, SIAM, 1994.

[3] 櫻井鉄也，松尾宇泰，片桐孝洋, “数値線形代数の数理と HPC,” pp. 34–35, 共立出版, 2018.

[4] Y. Saad, “Iterative Methods for Sparse Linear Systems,” pp. 151–243, SIAM, 2003.

[5] A. Greenbaum, “Iterative Methods for Solving Linear Systems,” pp. 61–75, SIAM, 1997.

[6] J. W. Demmel and H. D. Nguyen, “Numerical Reproducibility and Accuracy at Exascale,” *2013 IEEE 21st Symposium on Computer Arithmetic*, pp. 235–237, 2013.

[7] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, 2008.

[8] M. Furuichi, D. A. May, and P. J. Tackley, “Development of a Stokes flow solver robust to large viscosity jumps using a Schur complement approach with mixed precision arithmetic,” *Journal of Computational Physics*, vol. 230, no. 24, pp. 8835–8851, 2011.

[9] T. Kouya, “A Highly Efficient Implementation of Multiple Precision Sparse Matrix-Vector Multiplication and Its Application to Product-type Krylov Subspace Methods,” *International Journal of Numerical Methods and Applications*, vol. 7, pp. 107–119, 2012.

[10] 幸谷智紀, “多倍長精度数値計算: GNU MP, MPFR, QD によるプログラミング,” pp. 106–124, 森北出版, 2019.

[11] S. M. Rump, “Verification methods: Rigorous results using floating-point arithmetic,” *Acta Numerica*, vol. 19, pp. 287–449, 2010.

[12] D. H. Bailey, “High-Precision Floating-Point Arithmetic in Scientific Computation,” *Computing in science & engineering*, vol. 7, no. 3, pp. 54–61, 2005.

[13] D. Mukunoki and D. Takahashi, “Implementation and Evaluation of Quadruple Precision BLAS Functions on GPUs,” *International Workshop on Applied Parallel Computing*, pp. 249–259, 2010.

[14] 八木武尊，長谷川秀彦，石渡恵美子, “並列処理を用いた対話的多倍長演算環境 MuPAT の高速化,” 第 *17* 回情報科学技術フォーラム講演論文集第 1 分冊, vol. CB-005, pp. 43–48, 2018.

[15] 小武守恒，藤井昭宏，長谷川秀彦，西田晃, “反復法ライブラリ向け 4 倍精度演算の実装と SSE2 を用いた高速化,” 情報処理学会論文誌コンピューティングシステム *(ACS)*, vol. 1, no. 1, pp. 73–84, 2008.

[16] I. Yamazaki, S. Tomov, T. Dong, and J. J. Dongarra, “Mixed-Precision Orthogonalization Scheme and Adaptive Step Size for Improving the Stability and Performance of CA-GMRES on GPUs,” *International Conference on High Performance Computing for Computational Science*, pp. 17–30, 2014.

[17] L. N. Trefethen and D. Bau III, “NUMERICAL LINEAR ALGEBRA,” pp. 243–312, SIAM, 1997.

[18] Intel, “Intel Math Kernel Library.” `https://software.intel.com/en-us/mkl/`, (参照 2020-01-01).

[19] C. C. Paige and M. A. Saunders, “SOLUTION OF SPARSE INDEFINITE SYSTEMS OF LINEAR EQUATIONS,” *SIAM Journal on Numerical Analysis*, vol. 12, no. 4, pp. 617–629, 1975.

[20] S. L. Zhang, “GPBi-CG: Generalized product-type methods based on Bi-CG for solving nonsymmetric linear systems,” *SIAM Journal on Scientific Computing*, vol. 18, no. 2, pp. 537–551, 1997.

[21] I. Yamazaki, S. Tomov, and J. J. Dongarra, “Mixed-Precision Cholesky QR Factorization and Its Case Studies on Multicore CPU with Multiple GPUs,” *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C307–C330, 2015.

[22] T. Saito, E. Ishiwata, and H. Hasegawa, “Analysis of the GCR method with mixed precision arithmetic using QuPAT,” *Journal of Computational Science*, vol. 3, no. 3, pp. 87–91, 2012.

[23] T. J. Dekker, “A Floating-Point Technique for Extending the Available Precision,” *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[24] D. E. Knuth, “The Art of Computer Programming Volume 2: Seminumerical Algorithms,” pp. 236–237, Addison-Wesley, 1969.

[25] Y. Hida, X. S. Li, and D. H. Bailey, “Quad-double arithmetic: Algorithms, Implementation, and Application,” *Technical Report LBNL-46996*, pp. 1–28, 2000.

[26] D. H. Bailey, “QD: A double-double and quad-double package for Fortran and C++.” https://www.davidhbailey.com/dhbsoftware/, (参照 2020-01-01).

[27] MathWorks, “MATLAB.” https://mathworks.com/products/matlab.html, (参照 2020-01-01).

[28] JuliaLang.org, “The Julia Programming Language.” https://julialang.org/, (参照 2020-01-01).

[29] “DoubleDouble; A MATLAB library for extended ("double double") precision, giving close to quad precision.” https://github.com/tholden/DoubleDouble, (参照 2020-01-01).

[30] “DoubleDouble.jl; Julia package for performing extended-precision arithmetic using pairs of floating-point numbers.” https://github.com/JuliaMath/DoubleDouble.jl, (参照 2020-01-01).

[31] M. Lu, B. He, and Q. Luo, “Supporting extended precision on graphics processors,” *Proceedings of the sixth international workshop on data management on new hardware*, pp. 19–26, 2010.

[32] M. Joldes, J. M. Muller, V. Popescu, and W. Tucker, “CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications,” *International Congress on Mathematical Software*, pp. 232–240, 2016.

[33] M. Joldes, “CAMPARY –CudA Multiple Precision ARithmetic librarY Boost C++ Libraries.” http://homepages.laas.fr/mmjoldes/campary/, (参照 2020-01-01).

[34] IBM, “128-bit long double floating-point data type.” https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.genprogc/128bit_long_double_floating-point_datatype.htm, (参照 2020-01-01).

[35] IBM, “IBM XL Fortran for AIX, V16.1.0, Language Reference.” https://www.ibm.com/support/knowledgecenter/SSGH4D_16.1.0/com.ibm.compilers.aix.doc/langref.pdf?view=kc, (参照 2020-01-01).

[36] IBM, “IBM XL Fortran for Linux, V16.1.1, Language Reference.” https://www.ibm.com/support/knowledgecenter/SSAT4T_16.1.1/com.ibm.compilers.linux.doc/langref.pdf?view=kc, (参照 2020-01-01).

[37] C. Daramy Loirat, D. Defour, F. De Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J. M. Muller, “CR-LIBM A library of correctly rounded elementary functions in double-precision,” *Technical report, LIP, ensl-01529804*, 2006.

[38] 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*, vol. 14, no. 1, pp. 1–17, 1988.

[39] Z. Xianyi, “OpenBLAS.” https://www.openblas.net/, (参照 2020-01-01).

[40] X. S. Li, J. W. Demmel, D. H. Bailey, Y. Hida, J. Iskandar, W. Kahan, K. Anil, M. C. Martin, B. Thompson, T. Tung, and D. Yoo, “XBLAS–extra precise basic linear algebra subroutines.” https://www.netlib.org/xblas/, (参照 2020-01-01).

[41] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, and M. C. Martin, “Design, Implementation and Testing of Extended and Mixed precision BLAS,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 152–205, 2002.

[42] Free Software Foundation, “GMP, The GNU Multiple Precision Arithmetic Library.” https://gmplib.org/, (参照 2020-01-01).

[43] INRIA, “The GNU MPFR Library, A C library for multiple-precision floating-point computations with correct rounding.” https://www.mpfr.org/, (参照 2020-01-01).

[44] M. Nakata, “MPACK.” http://mplapack.sourceforge.net/, (参照 2020-01-01).

[45] OpenMP ARB, “OpenMP, The OpenMP API specification for parallel programming.” http://www.openmp.org/wp/, (参照 2020-01-01).

[46] 中田真秀, “MPACK: 高精度 BLAS, LAPACK の概要と性能評価,” 情報処理学会研究報告, vol. 2012-HPC-136, no. 18, pp. 1–7, 2012.

[47] M. Nakata, “A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP,-QD and-DD,” *2010 IEEE International Symposium on Computer-Aided Control System Design*, pp. 29–34, 2010.

[48] Khronos Group, “OpenCL - The open standard for parallel programming of heterogeneous systems, .” https://www.khronos.org/opencl/, (参照 2020-01-01).

[49] 中村光典，中里直人, “OpenCL による四倍精度行列積の高速化,” 情報処理学会研究報告, vol. 2012-HPC-133, no. 27, pp. 1–8, 2012.

[50] 国立研究開発法人日本原子力研究開発機構システム計算科学センター, “4 倍精度 Basic Linear Algebra Subprograms: QPBLAS.” https://ccse.jaea.go.jp/software/QPBLAS/, (参照 2020-01-01).

[51] S. Yamada, T. Ina, N. Sasa, Y. Idomura, M. Machida, and T. Imamura, “Quadruple-precision BLAS using Bailey’s arithmetic with FMA instruction: its performance and applications,” *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1418–1425, 2017.

[52] Intel, “Intel Intrinsics Guide.” https://software.intel.com/sites/landingpage/IntrinsicsGuide/, (参照 2020-01-01).

[53] Y. Hotaka, “MuPAT: Multiple Precision Arithmetic Toolbox.” https://www.ed.tus.ac.jp/1419521/, (参照 2020-01-01).

[54] “Sparse BLAS.” http://www.netlib.org/utk/people/JackDongarra/etemplates/node381.html, (参照 2020-01-01).

[55] NVIDIA, “cuSPARSE, the CUDA sparse matrix library.” https://docs.nvidia.com/cuda/cusparse/index.html, (参照 2020-01-01).

[56] The Scalable Software Infrastructure Project, “反復解法ライブラリ Lis.” https://www.ssisc.org/lis/, (参照 2020-01-01).

[57] D. Mukunoki and D. Takahashi, “Using Quadruple Precision Arithmetic to Accelerate Krylov Subspace Methods on GPUs,” *International Conference on Parallel Processing and Applied Mathematics (PPAM2013)*, pp. 632–642, 2013.

[58] S. Kikkawa, T. Saito, E. Ishiwata, and H. Hasegawa, “Development and acceleration of multiple precision arithmetic toolbox MuPAT for Scilab,” *JSIAM Letters*, vol. 5, pp. 9–12, 2013.

[59] Scilab Enterprises, “Scilab.” https://www.scilab.org/, (参照 2020-01-01).

[60] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida, “Performance Evaluation of Parallel Sparse Matrix–Vector Products on SGI Altix3700,” *International Workshop on OpenMP (IWoMP 2005)*, pp. 153–163, 2005.

[61] E. J. Im, K. Yelick, and R. Vuduc, “Sparsity: Optimization Framework for Sparse Matrix Kernels,” *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[62] A. Buluç, S. Williams, L. Oliker, and J. W. Demmel, “Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication,” *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 721–733, 2011.

[63] E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi,” *International Conference on Parallel Processing and Applied Mathematics (PPAM2013)*, pp. 559–570, 2013.

[64] N. Morita, K. Yonekura, I. Yasuzumi, T. Mitsuyoshi, G. Hashimoto, and H. Okuda, “Development of 3×3 DOF blocking structural elements to enhance the computational intensity of iterative linear solver,” *Mechanical Engineering Letters*, vol. 2, pp. 1–6, 2016.

[65] 一般社団法人 FrontISTR Commons, “FrontISTR, オープンソース大規模並列 FEM 非線形構造解析プログラム.” https://www.frontistr.com/, (参照 2020-01-01).

[66] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, “Parallel distributed computing using Python,” *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, 2011.

[67] O. Bröker, O. Chinellato, and R. Geus, “Using Python for large scale linear algebra applications,” *Future Generation Computer Systems*, vol. 21, no. 6, pp. 969–979, 2005.

[68] Python Software Foundation, “Programming language Python.” https://www.python.org/, (参照 2020-01-01).

[69] T. Oliphant, “NumPy, the fundamental package for scientific computing with Python.” https://numpy.org/, (参照 2020-01-01).

[70] Scipy developers, “SciPy, Python-based ecosystem of open-source software for mathematics, science, and engineering.” https://www.scipy.org/, (参照 2020-01-01).

[71] Mozilla and individual contributors, “bindgen, Generate Rust bindings for C and C++ libraries.” https://docs.rs/bindgen/0.51.0/bindgen/, (参照 2020-01-01).

[72] cppreference.com, “std::vector, C++ Reference.” https://cppreference.com/w/cpp/container/vector, (参照 2020-01-01).

[73] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: past, present and future,” *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[74] cppreference.com, “Common mathematical functions, C++ Reference.” https://en.cppreference.com/w/cpp/numeric/math/fma, (参照 2020-01-01).

[75] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, “Parallel programming in OpenMP,” *Morgan kaufmann*, 2000.

[76] J. L. Hennessy and D. A. Patterson, “Computer architecture: a quantitative approach,” Elsevier, 2011.

[77] U. Drepper and I. Molnar, “The native POSIX thread library for Linux,” *White Paper, Red Hat Inc*, 2003.

[78] T. Jain and T. Agrawal, “The Haswell Microarchitecture-4th Generation Processor,” *International Journal of Computer Science and Information Technologies*, vol. 4, no. 3, pp. 477–480, 2013.

[79] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, “The microarchitecture of the Pentium® 4 processor,” *Intel Technology Journal*, pp. 1–13, 2001.

[80] L. Gwennap, “Sandy Bridge spans generations,” *Microprocessor Report*, vol. 9, no. 27, pp. 10–01, 2010.

[81] Y. Nievergelt, “Scalar Fused Multiply-Add Instructions Produce Floating-Point Matrix Arithmetic Provably Accurate to the Penultimate Digit,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 1, pp. 27–48, 2003.

[82] T. Ogita, S. M. Rump, and S. Oishi, “ACCURATE SUM AND DOT PRODUCT,” *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005.

[83] Intel, “Memory Layout Transformations.” https://software.intel.com/en-us/articles/memory-layout-transformations, 2013 (参照 2020-01-01).

[84] R. Strzodka, “Abstraction for AoS and SoA layout in C++,” *GPU computing gems Jade edition*, pp. 429–441, 2012.

[85] Y. Saad, “SPARSEKIT: a basic tool kit for sparse matrix computation (version2).” https://www-users.cs.umn.edu/~saad/software/SPARSKIT/, 2019 (参照 2020-01-01).

[86] Intel, “Intel C/C++ Compiler, Developer Zone..” https://software.intel.com/en-us/c-compilers, (参照 2020-01-01).

[87] H. Fujiwara, “exflib - extend precision floating-point arithmetic library.” http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/, (参照 2020-01-01).

[88] Free Software Foundation, “GCC, the GNU Compiler Collection.” https://gcc.gnu.org/, (参照 2020-01-01).

[89] 藤原宏志, “科学技術計算に適した多倍長数値計算環境の構築と数値的に不安定なスキームの直接計算の実現,” 情報処理学会論文誌コンピューティングシステム *(ACS)*, vol. 48, pp. 22–30, 2007.

[90] “SuiteSparse Matrix Collection.” https://sparse.tamu.edu, (参照 2020-01-01).

[91] Intel, “Intel Vtune Amplifier.” https://www.xlsoft.com/jp/products/intel/vtune/index.html, (参照 2020-01-01).

[92] H. Takizawa, S. Hirasawa, Y. Hayashi, R. Egawa, and H. Kobayashi, “Xevolver: An XML-based code translation framework for supporting HPC application migration,” *2014 21st International Conference on High Performance Computing (HiPC)*, pp. 1–11, 2014.

研究業績一覧

博士論文に関する研究業績

査読制度のある学術雑誌に掲載された論文

1. 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “AVX2 を用いた倍精度 BCRS 形式疎行列と倍々精度ベクトル積の高速化, ” 情報処理学会論文誌コンピューティングシステム (ACS), vol. 7, no.4, pp. 25–33, 2014.

査読制度のある国際会議録に掲載された論文

1. Toshiaki Hishinuma, Teruo Tanaka, and Hidehiko Hasegawa, “SIMD Parallel Sparse Matrix-Vector and Transposed-Matrix-Vector Multiplication in DD Precision,” High Performance Computing for Computational Science – VECPAR 2016, LNCS 10150, Springer, pp. 21–34, Porto, Portugal, Jul. 2016.

2. Toshiaki Hishinuma, Akihiro Fujii, Teruo Tanaka, and Hidehiko Hasegawa, “AVX acceleration of DD arithmetic between a sparse matrix and vector,” Parallel Processing and Applied Mathematics, Part 1, LNCS 8384, Springer, pp. 622–631, Warsaw, Poland, Sep. 2013.

査読制度のある国内会議録に掲載された論文

1. 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “AVX を用いた倍々精度疎行列ベクトル積の高速化,” 2013 年ハイパフォーマンスコンピューティングと計算科学シンポジウム (HPCS2013), pp. 23–31, 東京, 2013.1.

その他

1. Toshiaki Hishinuma and Hidehiko Hasegawa, “AVX2 acceleration of SpMV and vector operations with Double-double precision vectors,” the 10th International Workshop on Parallel Matrix Algorithms and Applications (PMAA ’18), Zurich, Switzerland, June 2018 (Talk).

2. Hidehiko Hasegawa and Toshiaki Hishinuma, “Robust and Fast BiCG Method using SIMD-Accelerated DD Arithmetic,” The International Conference on Preconditioning Techniques for Scientific and Industrial Applications (Preconditioning 2017), Poster Presentation, Vancouver, Canada, July 2017 (Poster presentation).

3. 菱沼 利彰, “倍精度と倍々精度の混合精度反復解法の評価,” 第 5 回 大規模並列数値計算技術に関する研究集会 — 多倍長計算と精度保証 — (LSPANC 2017), 神戸, 2017.3 (口頭発表).

4. 菱沼 利彰, “AVX2 を用いた倍々精度演算の反復解法への適用と評価,” 日本応用数理学会 2016 年度年会, 「正会員主催 OS : 先進的環境おける数値計算と関連基盤技術」, pp. 1–2, 福岡, 2016.9.

5. 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “AVX2 を用いた倍精度疎行列と倍々精度ベクトルの積における精度と性能,” 日本応用数理学会 2015 年度年会, 「正会員主催 OS : 多倍長精度浮動小数点演算の高速化手法と応用」, pp. 1–2, 石川, 2015.9.

6. 菱沼 利彰，藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “AVX2 を用いた倍々精度反復解法の高速化,” 情報処理学会研究報告, vol. 2015-HPC-148, no. 9, pp. 1–7, 大分, 2015.2.

7. Toshiaki Hishinuma, Akihiro Fujii, Teruo Tanaka, and Hidehiko Hasegawa, “Fast computation of double precision sparse matrix in BCRS and DD vector product using AVX2,” 11th International Meeting High Performance Computing for Computional Science (VECPAR2014), Eugene, Oregon, USA, July 2014 (Poster presentation).

8. 佐藤 真之介, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, “AVX を用いた BCRS 形式疎行列ベクトル積の特性評価,” 第 76 回情報処理学会全国大会, pp. 215–216, 東京, 2014.3.

9. 菱沼 利彰, 田中 輝雄, 長谷川 秀彦, “倍精度 BCRS 形式疎行列と倍々精度ベクトル積の AVX2 による高速計算,” 第 4 回多倍長精度計算フォーラム, 東京, 2014.3 (口頭発表).

10. 菱沼 利彰, 田中 輝雄, 長谷川 秀彦, “疎行列ベクトル積に対する OpenMP スケジューリング方式の分析,” 2014 年ハイパフォーマンスコンピューティングと計算科学シンポジウム (HPCS2014), p.31, 東京, 2014.1 (ポスター発表).

11. 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “AVX を用いた倍々精度疎行列ベクトル積の高速化,” 第 3 回多倍長精度計算フォーラム, 東京, 2013.3 (口頭発表).

12. 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “AVX を用いた倍々精度疎行列ベクトル積 -転置行列-,” 2013 年ハイパフォーマンスコンピューティングと計算科学シンポジウム (HPCS2013)，p.76, 東京, 2013.1 (ポスター発表).

13. Toshiaki Hishinuma, Akihiro Fujii, Toshiaki Tanaka, and Hidehiko Hasegawa, “An Evaluation of Double-Double Precision Operation for Iterative Solver Library Using AVX,” The 11th International Symposium on Advanced Technology (ISAT-Special), P-E10-I, Tokyo, Oct., 2012 (Poster presentation).

14. 菱沼 利彰, 浅川 圭介, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, “反復解法ライブラリ向け倍々精度演算の AVX を用いた高速化,” 情報処理学会研究報告, vol. 2012-HPC-135, no. 16, pp. 1–6, 鳥取, 2012.8.

その他の研究業績

査読制度のある国際会議録に掲載された論文

1. Toshiaki Hishinuma and Maho Nakata, “pzqd: PEZY-SC2 acceleration of double-double precision arithmetic library for high-precision BLAS,” International Conference on Computational and Experimental Engineering and Sciences, Mechanisms and Machine Science, vol. 75, Springer, pp. 717–736, Tokyo, Japan, Mar. 2019.

2. Toshiaki Hishinuma, Takuma Sakakibara, Akihiro Fujii, Teruo Tanaka, and Shoichi Hirasawa, “Xev-GMP: Automatic Code Generation for GMP Multiple-Precision Code from C Code,” 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), pp. 295–298, Paris, France, Aug. 2016.

その他

1. 菱沼 利彰, 中田 真秀, “PEZY-SC2 上における倍々精度演算ライブラリ pzqd を用いた倍々精度 Rgemm の高速化,” 情報処理学会研究報告, vol. 2018-HPC-167, no. 10, pp. 1–9, 那覇, 2018.12.

2. 菱沼 利彰, 黒澤 範行, “PEZY-SC3 に向けた PEZY-SC シリーズ向け OpenFOAM の実装と性能評価,” オープン CAE シンポジウム 2018, pp. 1–6, 川崎, 2018.12.

3. 菱沼 利彰, 中田 真秀, “PEZY-SC2 上における倍々精度 Rgemm の実装と評価,” 日本応用数理学会 2018 年度年会, 「正会員主催 OS: 多倍長精度浮動小数点演算の高速化手法と応用」, pp.1–2, 名古屋, 2018.9.

4. Toshiaki Hishinuma, Ryo Sakamoto, Hitoshi Ishikawa, “Implementation and Evaluation of BLAS on PEZY-SC/SC2 Processor,” SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP’18), Tokyo, March 2018.

5. 土肥 樹, 伊藤 友太, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, “Intel/KNL における倍々精度疎行列ベクトル積の特性評価,” 第 80 回情報処理学会全国大会, pp. 51–52, 東京, 2018.3.

6. 伊藤 友太, 土肥 樹, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, “Knights Landing における倍々精度基本演算のハイブリッド並列の特性評価,” 第 80 回情報処理学会全国大会, pp. 55–56, 東京, 2018.3.

7. 愛沢 菜穂，斯波 柾，菱沼 利彰，藤井 昭宏，田中 輝雄，平澤 将一, “GMP 対応 MPI 通信ライブラリの設計と基本性能評価,” 第 5 回 大規模並列数値計算技術に関する研究集会 — 多倍長計算と精度保証 — (LSPANC 2017), 神戸, 2017.3 (口頭発表).

8. 斯波 柾，愛沢 菜穂，菱沼 利彰，藤井 昭宏，田中 輝雄，平澤 将一, “Xev-GMP への MPI 通信機能の拡張と反復解法への適用に向けた評価,” 第 5 回 大規模並列数値計算技術に関する研究集会 — 多倍長計算と精度保証 — (LSPANC 2017), 神戸, 2017.3 (口頭発表).

9. 斯波 柾，菱沼 利彰，田中 輝雄，藤井 昭宏，平澤 将一, “多倍長精度プログラムの自動生成機構 Xev-GMP における混合精度プログラムの生成と評価,” 情報処理学会研究報告, vol. 2016-HPC-157, no. 3, pp. 1–8, 札幌, 2016.12.

10. 菱沼 利彰，藤井 昭宏，田中 輝雄，平澤 将一, “GMP を用いた混合精度型プログラムの自動生成機構の提案,” 日本応用数理学会 2016 年度年会, 「正会員主催 OS：多倍長精度浮動小数点演算の高速化手法と応用」, pp. 1–2, 福岡, 2016.9.

11. 榊原 巧磨, 佐々木 信一, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 平澤 将一, “GMP ライブラリを用いた任意多倍長プログラムへの自動変換機構の提案,” 情報処理学会研究報告, vol. 2015-HPC-152, no. 6, pp. 1–8, 北海道, 2015.12.

12. 佐々木 信一, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 椋木 大地, 今村 俊幸, “京・FX10 における倍々精度演算の高速化, 情報処理学会研究報告,” vol. 2015-HPC-151, no. 15, pp. 1–7, 2015.9.

13. 丸地 賢, 佐々木 信一, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 平澤 将一, “Xevolver を用いた GMP コードへの自動変換機能の実装,” 第 77 回情報処理学会全国大会, pp. 37–38, 京都, 2015.3.

14. 花上 直樹, 佐々木 信一, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, “Hierarchical Diagonal Blocking を用いた疎行列ベクトル積の特性評価,” 第 77 回情報処理学会全国大会, pp.31–32, 京都, 2015.3.

15. 佐々木 信一, 菱沼 利彰, 藤井 昭宏, 田中 輝雄, “Many Integrated Corearchitecture における倍々精度疎行列ベクトル積,” 情報処理学会研究報告, vol. 2014-HPC-145, no. 16, pp. 1–7, 2014.7.

16. Hidekatsu Kayama, Toshiaki Hishinuma, and Eijiroh Ohki, “Shorting of Transfer Duration Using File Transfer Striping,” The 11th International Symposium on Advanced Technology (ISAT-Special), P-E12-I, Kogakuin University, Tokyo, Oct., 2012 (Poster presentation).

付録A　DD-AVX v3の機能

開発したDD-AVX v3 (https://github.com/t-hishinuma/DD-AVX_v3) のベクトル，疎行列クラスの機能について紹介する．

ベクトルクラスのメンバ関数

C++の標準ライブラリでは，配列の拡張として“std::vector”というクラスを用意している [72]．これはテンプレートクラスで，宣言時に“<>”を用いて型を指定することで，指定した型のstd::vectorが生成される．std::vectorは従来のCの配列の機能を拡張したもので，クラスのメンバ関数として配列長の変更，取得，要素の挿入，削除，代入演算子を用いたコピーなどを提供している．d_real_vector, dd_real_vectorはstd::vectorを継承して定義した．また，メンバ変数の追加は行っていない．

はじめにd_real_vectorとdd_real_vectorをstd::vector同様に扱えるようにした．倍精度ベクトルクラスであるd_real_vectorはstd::vectorの機能をすべてそのまま使うことができる．一方，DDベクトルクラスであるdd_real_vectorはSoA型のためほとんどの機能が正しく動作しない．よく使われるstd::vectorの関数を再定義することでstd::vector相当のDDのベクトルクラスを実装した．表A.1に実装したdd_real_vectorの関数を示す．Sはdoubleまたはdd_real，Vはd_real_vectorまたはdd_real_vectorである．また，すべての入力値は内部で変更されないため定数を示す“const”をつけているが表では省略した．

“operator=”は“=”演算子のオーバーロードを意味する．dd_real_vectorには倍精度型またはQDライブラリのdd_real型のstd::vectorをAoSからSoAに変換して代入できるようにした．変換には時間がかかるが，他のソフトウェアなどによって作られたstd::vectorを代入できることは有用であると考えられる．

また，dd_real_vectorはstd::vectorのもつコンストラクタによるメモリ確保の機能を再定義することでDDにあわせて指定された配列長Nに対し倍精度型のstd::vectorに対して2倍のメモリ空間を確保するようにした．このときstd::vectorクラスのベクトルのサイズを保持する変数にはベクトル自体の長さであるNをもたせた．長さNの倍精度型のstd::vector2つによって構成され，それぞれ*hi*と*lo*を表現する．また，*hio*と*lo*はメモリ上で連続して配置さ

表 A.1: DD ベクトルクラスに再定義した std::vector の関数

名前	説明
dd_real_vector(int n)	空のベクトルを作成
dd_real_vector(int n)	長さ n のベクトルを作成
dd_real_vector(int n, S a)	すべての要素が a の長さ n のベクトルを作成
int size()	要素数を返す
dd_real operator[int i]	i 番目の要素を返す
dd_real at(int i)	i 番目の要素を返す
void push_back(S a)	末尾へ要素 a を追加
void insert(int i, S a)	i 番目の要素に a を挿入
void erase(int i)	i 番目の要素の削除
dd_real_vector operator=(V x)	ベクトル型の x を代入
dd_real_vector operator=(std::vector<double>& x)	std::vector 型の x を代入
dd_real_vector operator=(std::vector<dd_real>& x)	std::vector 型の x を代入
double* data()	配列の先頭ポインタを返す

れるようにし，“data()” は *hi* の先頭ポインタを返す関数とした．

これらに加えて Rust, Julia, MATLAB のベクトル型に実装されている機能を調査し，必要な関数をメンバ関数として加えた．d_real_vector に追加した関数を表 A.2，d_real_vector に追加した関数を表 A.3 に示す．

“operator 型名 ()” は型の変換（キャスト）の定義を意味する．d_real_vector, dd_real_vector はそれぞれの変換に加えて倍精度の std::vector に変換できるようにした．これにより計算結果などを C++の標準ライブラリなどに渡すことができる．

“operator +” は加算演算子の定義（C++では演算子オーバーロードとよばれる）を意味する．たとえば “dd_real_vector operator+(V x)” は，d_real_vector または dd_real_vector のベクトル x を入力とし，自身と x の和を V 型で返す関数である．スカラ，ベクトルに対する算術演算の実装は四則演算の+, -, *, /，および代入と四則演算を同時に行う演算子の+=, -=, *=, /=を実装した．これらはすべて同じインタフェースであるため冗長と考え，算術演算は加算のみを載せた．また，dd_real_vector と d_real_vector の一致判定は dd_real_vector の *lo* がすべて零で *hi* の値がすべて一致するときのみ真を返すようにした．

表 A.2: 倍精度型のベクトルクラスに追加で実装した関数（四則演算は加算のみを抜粋）

名前	説明
operator dd_real_vector()	d_real_vector へのキャスト
operator std::vector<double>()	double 型の std::vector へのキャスト
operator std::vector<dd_real>()	dd_real 型の std::vector へのキャスト
dd_real_vector operator+(dd_real_vector& x)	自身と x の和を dd_real_vector として返す
d_real_vector operator+(d_real_vector& x)	自身と x の和を d_real_vector として返す
d_real_vector operator+(std::vector<double>& x)	自身と x の和を d_real_vector として返す
d_real_vector operator+(double a)	自身の各要素と x の和を d_real_vector として返す
dd_real_vector operator+(dd_real a)	自身と x の和を dd_real_vector として返す
bool operator==(V& x)	x との一致判定
bool operator!=(V& x)	x との不一致判定

表 A.3: DD ベクトルクラス型の追加で実装した関数（四則演算は加算のみを抜粋）

名前	説明
operator d_real_vector()	d_real_vector へのキャスト
operator std::vector<double>()	double 型の std::vector へのキャスト
operator std::vector<dd_real>()	dd_real 型の std::vector へのキャスト
dd_real_vector operator+(dd_real_vector& x)	自身と x の和を dd_real_vector として返す
dd_real_vector operator+(d_real_vector& x)	自身と x の和を dd_real_vector として返す
dd_real_vector operator+(std::vector<double>& x)	自身と x の和を dd_real_vector として返す
dd_real_vector operator+(double a)	自身の各要素と x の和を d_real_vector として返す
dd_real_vector operator+(dd_real a)	自身と x の和を dd_real_vector として返す
bool operator==(V& x)	x との一致判定
bool operator!=(V& x)	x との不一致判定

表 A.4: 倍精度と DD の疎行列クラスの機能の一覧

名前	説明
d_real_SpMat(string file, string format)	file から format 型の疎行列を生成.
d_real_SpMat(int n, int nnz)	N 行，nnz 個の非零要素のメモリを確保
d_real_SpMat operatordd_real_SpMat()	dd_real_SpMat 型へのキャスト
M operator=(M& A)	d_real_SpMat 型の A を代入
M operator=(M& A)	dd_real_SpMat 型の A を代入
void convert(string format)	format 型への変換
S at(int i, int j)	i 行 j 列の要素を返す
void at(int i, int j, S a)	i 行 j 列の要素に a を代入
int size()	行数の取得 (get_row() 関数を用いてもよい)
int get_col()	列数の取得
int get_brow()	ブロック行数の取得
int get_bcol()	ブロック列数の取得
V get_row(int i)	i 行目の取り出し
V get_col(int j)	j 列目の取り出し
V get_diag()	対角要素の取り出し

疎行列クラスのメンバ関数

表 A.4 に，d_real_SpMat の機能を示す．ここで S は double または dd_real，V は dd_real_vector または d_real_vector, M は d_real_SpMat または dd_real_SpMat，file は文字列型である．d_real_SpMat クラスの機能はコンストラクタ，キャストの対応関係が変わっているだけであるため省略する．

メンバ変数は行数や列数を保持する int 型の変数と，std::vector のインデックス配列，d_real_vector の非零要素の値の配列で，dd_real_SpMat の場合は非零要素の値の配列が dd_real_vector になる．format は疎行列の格納形式を示す文字列で “CRS”, “BCRS1x4”, “BCRS4x1” が入力できる．格納形式の指定は初期化時のコンストラクタで指定するか，“convert” 関数を用いることで他の格納形式に変換できる．

これらのクラスを入力できる基本演算の関数を C++のテンプレート機能を用いて定義した．基本演算の関数のインタフェースは表 3.1，3.2 のとおりである．スカラとして double と dd_real，ベクトルとして倍精度の std::vector，d_real_vector および dd_real_vector，疎行列として d_real_SpMat, dd_real_SpMat が入力できる．

付録B　DD-AVX v3を用いたBiCG法のプログラム

C++を用いたBiCG法のプログラムを図B.1に示す．このプログラムでは，ファイル“matrix.mtx”をBCRS4x1として読み込んで作られた要素の精度が倍精度の疎行列A，長さがAの行数と等しいDDのベクトルxを0で初期化, bを1で初期化に対し，Krylov部分空間法の1つであるBiCG法を相対残差が1.0e-12になるかAの行数回だけ反復させて$A\boldsymbol{x} = \boldsymbol{b}$を解くプログラムである．DD-AVX v3を用いることで，BiCG法のプログラムを50行程度で実装できる．

変数に対する指定は変数の宣言時だけでよく，各基本演算の関数呼び出しに精度の指定は必要ない．これは，どの変数の精度を変えても動作することを確認している．L.6に示すとおり，d_real_vector, dd_real_vectorの宣言や初期化はstd::vectorと同様にコンストラクタを用いて行うことができる．

次にDD-AVXのC++のインタフェースからbindgen [71]を用いてRustのインタフェースを生成し，Rustで作成したBiCG法のプログラムを図B.2に示す．

C++のプログラムとの違いとして，L. 59ですべての値が0で長さが倍精度行列型のAと同じベクトルをRustの標準ベクトル型を用いてtmpvecとして宣言し，DDのベクトルxの初期化はtmpvecを代入することで実装している点がある．C++のstd::vectorとの代入や演算を定義したことで，Rustの標準ベクトル型との代入や演算も生成され，Rustの標準ベクトル型と組みわせて利用できていることがわかる．

また，RustのBiCG法のプログラムを実行し，C++のBiCG法のプログラムと比べて実行時間の差がほとんどないことを確認した．今回開発したDD-AVXのC++のインタフェースを自動変換することで，疎行列やベクトルの生成，結果の分析などはC++と比べて高機能なプログラミング言語の標準機能を用い，時間がかかる演算はC++で実装された高速なバックエンドを呼び出して利用できる．

```
1  #include"dd_avx.hpp"
2  #include<iostream>
3  using dd_avx;
4
5  bool BiCG(d_real_SpMat A, dd_real_vector& x, dd_real_vector& b, double
       tol){
6    dd_real_vector r(A.size(), 0.0);
7
8    r = b - A * x; // r = b - Ax
9    dd_real_vector rtld = r; // rtld = r
10
11   dd_real_vector ptld = rtld; //p*0 = r*0
12   dd_real_vector p = r; //p0 = r0
13
14   dd_real_vector q(A.size(), 0.0);
15
16   dd_real alpha =0, beta = 0, rho=0, rho_old=1;
17   dd_real rnrm = 0, tmp1 = 0;
18
19   for(int iter = 0; iter < A.row(); iter++)
20   {
21     dot(r, rtld, rho);
22     beta = rho / rho_old;
23
24     xpay(r, beta, p);
25     SpMV(A, p, q);
26     xpay(rtld, beta, ptld);
27     TSpMV(A, ptld, qtld);
28
29     dot(ptld, q, tmp1);
30     alpha = rho / tmp1;
31
32     axpy(alpha, p, x);
33     axpy(-alpha, q, r);
34
35     // convergence check
36     nrm2(r, rnrm);
37     if( rnrm < tol ){
38       return true;
39     }
40
```

```
41      axpy(-alpha, qtld, rtld);
42      rho_old = rho;
43    }
44    return false;
45  }
46
47  int main(){
48    d_real_SpMat A("matrix.mtx", "BCRS4x1");
49    dd_real_vector x(A.size(), 0.0);
50    dd_real_vector b(A.size(), 1.0);
51
52    if( BiCG(A, x, b, 1.0e-12) )
53      for(int i=0; i < x.size(); i++)
54        std::cout << x[i] << std::endl;  // output vector
55
56    return 0;
57  }
```

図 B.1: BiCG 法のプログラム（C++）

```
1  mod dd_avx;
2  use dd_avx::*;
3
4  fn BiCG(A: d_real_SpMAT, x: dd_real_vector, b: dd_real_vector, tol: f64)
       -> bool {
5      let r = dd_real_vector(A.size(), 0.0);
6
7      // r = b - Ax
8      r = b - A * x;
9
10     // rtld = r
11     let dd_real_vector rtld = r;
12
13     //p*0 = r*0
14     let dd_real_vector ptld = rtld;
15     //p0 = r0
16     let dd_real_vector p = r;
17
18     let q = dd_real_vector(A.size(), 0.0);
19
20     let alpha = dd_real(0.0);
21     let beta = dd_real(0.0);
22     let rho = dd_real(0.0);
23     let rho_old = dd_real(0.0);
24     let rnrm = dd_real(0.0);
25     let tmp1 = dd_real(0.0);
26
27     for(int iter = 0; iter < A.row(); iter++)
28     {
29         dot(r, rtld, rho);
30         beta = rho / rho_old;
31
32         xpay(r, beta, p);
33         SpMV(A, p, q);
34         xpay(rtld, beta, ptld);
35         TSpMV(A, ptld, qtld);
36
37         dot(ptld, q, tmp1);
38         alpha = rho / tmp1;
39
40         axpy(alpha, p, x);
```

```
41          axpy(-alpha, q, r);
42
43          // convergence check
44          nrm2(r, rnrm);
45          if rnrm < tol {
46              return true;
47          }
48
49          axpy(-alpha, qtld, rtld);
50          rho_old = rho;
51      }
52      return false;
53  }
54
55  fn main() {
56
57      let A = d_real_SpMat("matrix.mtx", "BCRS4x1");
58
59      let tmpvec : vec![0.0; A.size()];
60      let x = dd_real_vector(tmpvec);
61      let b = dd_real_vector(A.size(), 1.0);
62
63      if BiCG(A, x, b, 1.0e-12) {
64
65          for i in 0..x.size() {
66              println!("{}", x[i]);
67          }
68      }
69  }
```

図 B.2: BiCG 法のプログラム（Rust）

付録C　GMPを用いたプログラムの自動生成

GMP では任意多倍長型の浮動小数点数として “mpf_t” という構造体を定義している．変数の使用前後に独自の構造体を初期化，解放する関数への受け渡しを行ったり，算術演算の式は演算子ではなく手続きの形式で行う必要がある．

そのため，プログラムの大部分を変更する必要があり，利用するための実装に必要なコストが高い．

そこで，Takizawa ら [92] が開発した，コードを構文木として扱い，変換ルーチンを作成できる Xevolver フレームワークを用いて，C 言語の倍精度プログラム（C コード）を，GMP を用いた任意多倍長プログラム（GMP コード）へ自動変換する機構，Xev-GMP を実装し，Web ブラウザからプログラムをアップロードすることで変換されたプログラムを出力する Xev-GMP-Web を開発した (http://xev.arch.is.tohoku.ac.jp/XevWeb/Xev-GMP-Web.html).

Xev-GMP は，C コードにディレクティブとして精度の情報を追記することで，構文木を解析してコード内の倍精度変数をすべて GMP の任意多倍長変数にした GMP コードを生成するプログラムである．

これは本論文において結果の検証や QD ライブラリのベンチマークを行うために用いている．

Xev-GMP は，たとえば “a = b + c * d” は，図 C.1 の左のような構文木として扱い，右のような構文木を生成し，次のようなコードが得られる．

```
mpf_mul(tmp3, c, d);

mpf_add(tmp1, b, tmp3);

mpf_set(a, tmp1);
```

それぞれ順に，GMP の乗算，加算，代入関数である．

手続きの形式への変換において，一時変数の追加が必要になる．算術演算式を構文木として見た際のノード番号から，自動的に一時変数を生成した．

Xev-GMP は，Xevolver への指示句をプログラムの先頭に入れ，指示句の引数として仮数部の精度を与えるだけで C コードを GMP コードに変換できる．

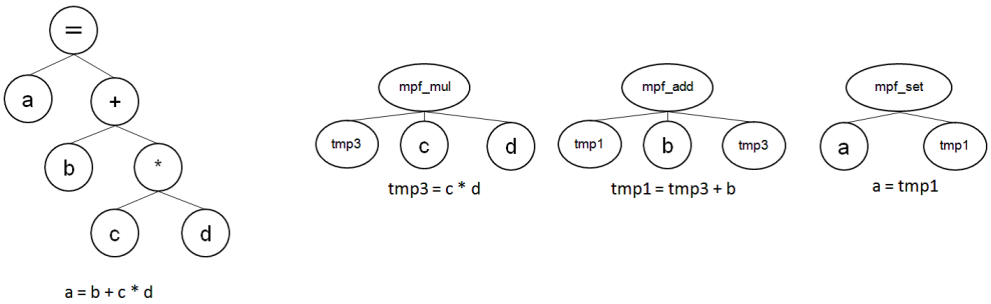

図 C.1: “a = b + c * d” の構文木 (左: C コード, 右: GMP コード)

機械的に倍精度のプログラムを高精度なプログラムに変換できるため，結果の検証や比較において有用である．