

Ryzen AI NPU における疎行列ベクトル積の高速化

小紫 仁嗣[†] 薄井 真之^{†,††} 小野 貴継^{†††} 高前田伸也^{†,††}

[†] 東京大学
^{††} 理化学研究所
^{†††} 福岡大学

あらまし 本研究では空間データフローアーキテクチャを有する Ryzen AI NPU (XDNA2) において、メモリ律速な行列ベクトル積を高速化する疎行列演算法を提案する。XDNA2 は明示的なデータ移動制御により高いデータ転送効率を誇るが、外部メモリ帯域が限られているためメモリ律速である行列ベクトル積の実行速度が制約される課題がある。この課題に対し、演算およびデータ転送の両面を削減可能な疎行列ベクトル積 (SpMV) に着目し、XDNA2 における固定長データ転送制約と親和性の高い ELL 形式を用いた演算カーネルを実装した。評価の結果、疎行列の各行の非ゼロ要素数が均一な条件下でスパース率 87.5% において、密行列ベクトル積に対し 3.93 倍の高速化を達成した。

キーワード 疎行列ベクトル積, ELL 形式, Ryzen AI NPU, XDNA2

1. ま え が き

疎行列ベクトル積は科学技術計算や枝刈りを施した機械学習や大規模言語モデルにおいて非常に重要な計算処理である。近年、これらの計算を含む AI 処理をエッジデバイスやコンシューマ PC 上で高効率に実行するため、AMD 社は消費者向けチップである Ryzen AI プロセッサにおいて、空間データフローアーキテクチャを有する XDNA2 NPU を搭載している。

XDNA2 では明示的かつ決定的にデータ移動を制御することで高い内部転送効率を実現する。一方で、主記憶との帯域幅は最大 60GB/s 程度に制限されており [1], 疎行列をそのまま行列ベクトル積 (GEMV) で計算すると演算量に対してデータ転送量が過大となりメモリ律速に陥る。

このメモリ帯域の制約を打破し、演算器の稼働率を向上させるためには、疎行列内のゼロ要素の転送と計算を省略する疎行列ベクトル積 (SpMV) の導入が不可欠である。しかし、XDNA2 はハードウェア効率を高めるために固定長でのデータ転送を前提としており、行ごとに要素数が可変となる一般的な疎行列形式 (CSR 形式など) をそのまま適用すると、複数コアでの並列処理において深刻な集約オーバーヘッドが生じてしまう。そのため、XDNA2 のアーキテクチャ特性と制約に適合した SpMV の実装手法はこれまで確立されていなかった。本研究では、XDNA2 NPU に適した疎行列ベクトル積の演算法を提案する。具体的には、XDNA2 で必須となる明示的なデータ移動の制御に適した疎行列格納方式や XDNA2 内の各演算機へのデータ分配方法を提案し、その性能を評価する。

2. 背 景

2.1 Ryzen AI NPU(XDNA2)

2.1.1 アーキテクチャ

XDNA2 [2] では各種タイルが 2 次元配置された空間アーキテ

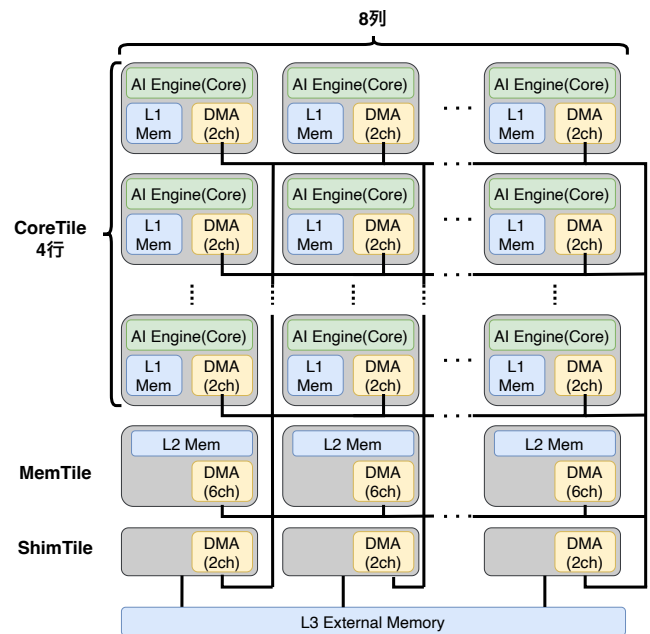


図 1: XDNA2 のタイル配置

クチャ (図 1) を採用している。具体的には、8 列 × 4 行 (計 32 基) のコアタイルの各列下部に、メモリタイルと shim タイルが 1 基ずつ配置される構成である。コアタイルには、AI Engine と呼ばれる演算機がある。AI Engine は Single Instruction, Multiple data (SIMD) 且つ Very Long Instruction Word (VLIW) のプロセッサであり、スカラー演算 Unit とベクトル演算 Unit を有する。コアタイルには 64KB の L1 メモリが備わっている。メモリタイルは 512KB の L2 メモリが備わっている。shim タイルは CPU・内蔵 GPU・NPU 間で共有される主記憶 (Unified Main Memory) と XDNA2 間の通信を担う。XDNA2 において、この主記憶はメモリ階層上の L3 として位置付けられている。

2.1.2 データ転送機構

XDNA2には、タイル間のデータ転送経路としてインターコネクト、カスケード、隣接タイルへのメモリアクセスの3種類が存在する。カスケードおよび隣接タイルへのメモリアクセスは、コアタイル間の局所的なデータ転送に特化した機構であるが、本実装ではこれらを使用せず、すべてのデータ転送をインターコネクト経由で行う。インターコネクトは、ソフトウェアによって経路を設定できるストリーム・スイッチを採用しており、どのタイル間でもデータ転送ができる。インターコネクトはDMAと呼ばれるインターフェースを介してタイル間のデータを転送する。DMA1ポートにつき64ビット幅でデータを転送する。コアタイルとshimタイルのDMAには入力ポートと出力ポートがそれぞれ2つずつあり、メモータイルのDMAには入力ポートと出力ポートがそれぞれ6つずつある。

主記憶(L3)とのデータ通信はshimタイルが担い、インターコネクトを介してNPU内の各タイルへデータの送受信を行う。XDNA2ではshimタイルが列方向に8個配置されており、各タイル128ビット幅での転送が可能のため、全体で合計1,024ビット幅の通信インターフェースを持つ。実機は1,800MHzで動作しているため、ハードウェアの理想的な最大メモリ帯域幅は $230.4\text{GB/s}(1,024\text{bit} \div 8\text{byte/bit} \times 1,800\text{MHz})$ となる。また、今回評価に使用した実機における主記憶の帯域幅は120GB/sまで対応している。しかし、評価で使用したAMD Ryzen AI 9 HX 370プロセッサにおけるXDNA2では最大帯域幅が60GB/sに制限される[1]。したがって、本研究の実験環境における主記憶とXDNA2間の理想メモリ帯域幅は60GB/sとなる。

2.1.3 コンパイルフロー

XDNAのプログラミングは、空間アーキテクチャの定義を行うPython APIであるIRON[3]と、各タイル上の具体的な演算を記述するC++カーネルプログラムによって構成される。IRONは、Pythonによる記述をAI Engine専用の中間表現であるMLIR-AIE[4]へ変換し、ハードウェアの詳細な制御を可能にする。本環境における実行用バイナリの生成は、以下の手順で行われる。まず、IRONを用いてデータフローやタイル配置などの構造定義(Python)をMLIR形式の構造記述に変換する。これと並行して、AIE API[5]に基づいて記述されたC++カーネルコードを専用コンパイラPeano[6]でコンパイルし、カーネルオブジェクトを生成する。次に、これら構造記述とカーネルオブジェクトを結合し、デバイス初期化時にロードされるハードウェア構成情報を含むバイナリ(XCLBIN)を生成する。さらに、構造記述からデータ移動やカーネル起動に必要な実行時命令を抽出し、バイナリファイルとして出力する。実行時は、XCLBINによるデバイス初期化後にこの命令シーケンスをロードして処理を行う。

2.2 疎行列ベクトル積

科学計算や深層学習の推論処理などで多用される一般行列ベクトル積(GEMV)は、 $y = Ax$ (A は行列、 x はベクトル)で表される演算である(本研究では計算の本質的特性を議論するため、単純な行列とベクトルの積とする)。しかし、この計算はデータの再利用性が低く、メモリ律速になりやすいという課題

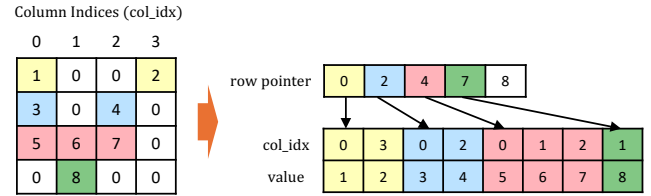


図 2: CSR 形式

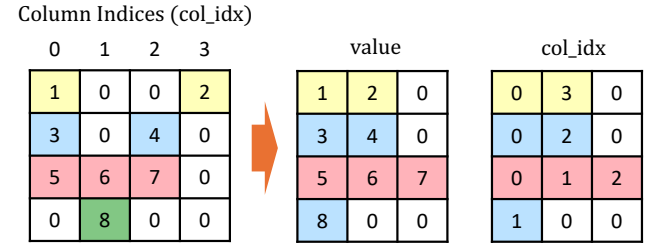


図 3: ELL 形式

がある。XDNA2ではメモリ帯域が制限されており、行列の要素の大部分がゼロである疎行列に対してGEMVで演算すると著しい性能低下を引き起こす。本研究ではこのボトルネックを解消するため、XDNA2に疎行列ベクトル積(SpMV)を実装する。SpMVは、不要なゼロ要素の計算とデータ転送を省略することで演算を高速化する手法である。その実行にあたっては、非ゼロ要素を効率的に格納するデータ形式が用いられる。代表的な形式としてCSR形式(図2)があるが、各行の非ゼロ要素数のばらつきによりループ長が不規則となり、ベクトル演算機におけるハードウェア並列化の恩恵を受けにくい場合がある。

一方、並列演算器での効率的な実行に適した形式としてELL形式(図3)がある。ELL形式は、元の疎行列の各行において非ゼロ要素を左詰めで配置し、全行の中で最大の非ゼロ要素数を基準として、満たない行ではゼロパディングし長さを揃える手法である。各値は対応する元の列番号(col_idx)を記録した配列とともに管理される。ELL形式は、全ての行のループ長が一定になるという特徴を持つ。これにより、並列演算時における計算ユニット間での負荷分散が容易になるだけでなく、ループ境界の判定が不要となるため、効率的なベクトル演算が可能になる。一方で、非ゼロ要素数が様でない場合においては、パディングによるデータ量の増加を招くという欠点も併せ持つ。

3. 提案手法

3.1 XDNA2に適した疎行列データ形式の選定

本研究では、XDNA2アーキテクチャのデータ移動制約を考慮し、ELL形式を採用した。

XDNA2のデータ転送機構は、固定長転送を前提として設計されている。しかし、CSR形式などの行ごとの要素数が可変である疎行列形式を採用した場合、データを固定長で機械的に分割すると同一行の要素が複数のコアに跨って分配され、コアが演算する行数もコア毎に可変となる。

これらの現象は、並列実行時においてベクトル演算器が得意とする複雑な集約処理を引き起こす。集約処理が必要になる

原因は二つある。一つ目は、同一行の要素が複数のコアにまたがることで、各コアが算出した部分和から最終的な解を求めるためのコア間の集約処理が必須となる。二つ目はデータの出力においても、各コアが担当する出力ベクトル y の要素数が不定となるため、固定長転送制約を満たすための複雑な処理（データの集約）が求められる。各コアは出力ベクトル y に一旦パディングを施して固定長データとして集約用タイルへ転送し、そこでパディング部分を削除して連続したデータ領域へと再構築した上で主記憶へ書き戻さなければならない。これらの集約・再構築処理は通信・同期・計算オーバーヘッドを招き、ベクトルプロセッサの高い並列演算性能を著しく低下させる要因となる。

これに対し、ELL 形式は行ごとの要素数が一定であるため、各コアへ割り当てる行数を制御でき、上記の問題を防ぐことができる。また、一般に ELL 形式は行ごとの非ゼロ要素数に偏りがある場合にパディングによるデータ量の増加を招くが、深層学習における疎な重み行列は、非ゼロ要素が比較的均一に分布する特性を持つことが知られている [7]。このため、ELL 形式を採用してもパディングの影響を最小限に抑えつつ、XDNA2 において効率的なマルチコアで疎行列ベクトル積を実現できる。

ELL 形式を保存する際には、value 行列と col_idx 行列は分けて保存するのが一般的だが、本研究では、value 行列と col_idx 行列を結合し一つの行列として XDNA2 に転送した。この理由は、XDNA2 において、外部メモリからデータをコアに転送する際には 1 コアあたり基本的に DMA のチャンネル数の分だけの種類のデータしか入出力できないという制約があるからである。第 2.1.2 節より、XDNA2 では基本的に 1 コアあたり 2 つのデータの入力と 2 つのデータの出力しかできない。SpMV では、入力として疎行列 A とベクトル x が必要になる。この時点で入力の 2ch 分を使用してしてしまうので、ELL 形式を value 行列と col_idx 行列に分けて転送することはできないため、二つの行列を結合して一つの行列にして転送している。また、value 行列は bf16 で保存し、col_idx 行列は uint16 で保存し NPU に転送している。

3.2 SpMV 演算カーネル

本研究では、XDNA2 アーキテクチャの AI Engine が備えるベクトル演算器を最大限に活用するため、異なるデータアクセスパターンを持つ二種類の SpMV カーネルを実装し、その性能を比較した。図 4(a) は入力となる疎行列を示しており、これを ELLPACK 形式へ変換した際のアクセスの違いを (b) および (c) に示す。図では視認性を考慮し、16 要素を一括計算できるように設定しているが、実際のカーネルでは 32 要素を一括計算できるようなカーネルになっている。

一つ目のカーネルは、行優先 (Row-Major) SpMV カーネルである。本カーネルは、図 4(b) に示すように、行方向に連続する 16 要素に対してベクトル命令を適用し、一括計算を行う。各行はメモリ上で連続して配置されており、疎行列ベクトル積を実行するには全行をコア数に応じて均等に分割し、各コアへ割り当てている。なお、図では視認性を考慮し行タイルサイズを 16 としているが、実際の実装では疎行列のサイズや使用するコア数に基づき行タイルサイズを設定している。

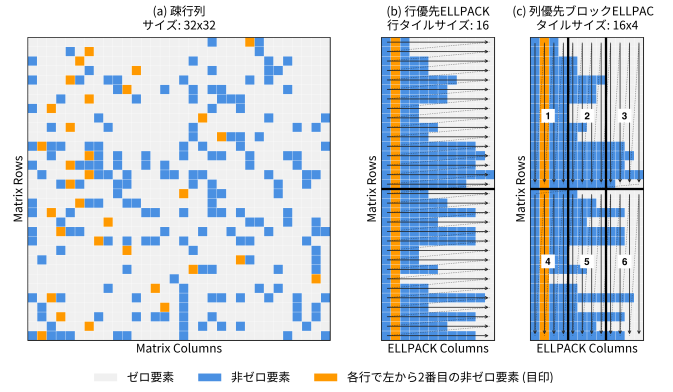


図 4: ELL におけるアクセスパターン

二つ目のカーネルは、ブロック列優先 (Block-Column-Major) SpMV カーネルである。本カーネルは、図 4(c) に示すように、ブロック内部において列方向に連続する 16 要素を一括計算する。ここでのタイル分割は二つの役割を持つ。まず、縦方向の分割（行ブロック）によって各コアへのデータ分配単位が決定される。次に、横方向の分割（列ブロック）によって、各コアが処理すべきデータが時系列順に分割される。例えば、図中のコア 1 にはタイル 1, 2, 3 が、コア 2 にはタイル 4, 5, 6 がそれぞれ割り当てられ、順次処理が行われる。なお、図では視認性を考慮しタイルサイズを 16×4 としているが、実際の実装では XDNA2 の L1 メモリ容量およびデータ転送制約に基づき、タイルサイズを 32×16 に設定している。また、ベクトル演算時のメモリアクセス効率を最大化するため、各タイル内のデータ順序が列優先となるよう、事前にデータの再配置を行っている。

行優先 SpMV カーネルの疑似コードを図 5 に示す。このカーネルでは、 $y = A * x$ の演算を行い、疎行列 A は ELLPACK 形式で保持している。XDNA2 のコアは、第 2.1.1 節で述べたように SIMD かつ VLIW のプロセッサであり、1 サイクルで複数のデータに対する並列演算が可能である。疑似コード内の A_col (行列 A の列インデックス) および A_val (行列 A の値) は、ELL 形式を採用することでメモリ上に連続して配置される。これにより、一度の命令で複数個のデータをベクトルレジスタへ一括転送するベクトルロードが可能となり、L1 メモリから演算器への高速なデータ供給が実現される。しかし、ベクトル B の参照においては、 $B[A_col[k]]$ のように非ゼロ要素の列番号に基づいた間接参照が発生する。SpMV の特性上列番号配列は不連続であり、 $B[A_col[k]]$ は不連続なメモリアクセスとなるため、ベクトルロードを適用することができない。その結果、 $B[A_col[k]]$ のメモリアクセスは 1 要素ずつ個別に読み出すスカラロードとして実行される。高い理論演算性能を持つ NPU コアにおいて、このスカラロードはプログラム全体の実行時間を増大させる要因の一つである。また、本現象は列優先 SpMV カーネルにおいても生じている。

3.3 メモリタイルを介したデータ転送による 32 コア並列 SpMV の実現

SpMV は不連続なメモリアクセスに伴うスカラロードが発生


```

1 // 疎行列ベクトル積 (SpMV) の単一コアにおけるカーネル疑似コード
2 // y = A * x を計算する
3 // A: 疎行列 (ELLPACK形式) A_col: 列インデックス配列, A_val: 非ゼロ要素配列
4 // x: 入力ベクトル, y: 出力ベクトル
5 // ell_width: 1行あたりの最大非ゼロ要素数 (ELLPACKの列数)
6 // Rows: 単一コアが処理する行数
7 for (uint32_t i = 0; i < Rows; i++) {
8     // 部分和を保存するアキュムレータをゼロで初期化
9     vector_acc = vector_zeros<float32, 32>();
10
11     // 疎行列 A を ELL形式で表した際の列インデックス(col)および非ゼロ要素(val)へのポインタ
12     int16_t *ptr_A_col = &A_col[i * ell_width];
13     bfloat16 *ptr_A_val = &A_val[i * ell_width];
14
15     // 非ゼロ要素数(ell_width)にわたるループ
16     for (uint32_t j = 0; j < ell_width; j += 32) {
17         // 【ベクトルロード】 疎行列 A のデータを SIMDレジスタへ一括転送
18         vector_A_col = vector_load<32>(ptr_A_col);
19         vector_A_val = vector_load<32>(ptr_A_val);
20
21         // 【スカラーロード】 不連続アクセスによるボトルネック
22         // A の列インデックスに基づき、入力ベクトル x から要素を個別に読み出す
23         aie::vector<bfloat16, 32> vector_x;
24         for (int k = 0; k < 32; k++) {
25             vector_x[k] = x[vector_A_col[k]];
26         }
27
28         // 積和演算 (ベクトルUnitで一括計算)
29         vector_acc = vector_mac(vector_acc, vector_A_val, vector_x);
30
31         ptr_A_col += 32;
32         ptr_A_val += 32;
33     }
34
35     // 32要素の加算結果を統合し、出力ベクトル y の第 i 要素に格納
36     float row_sum = reduce_add(vector_acc);
37     y[i] = (bfloat16)row_sum;
38 }

```

図 5: 行優先 SpMV カーネルの疑似コード

するため、全てのロードと演算がベクトル化可能な GEMV と比較して、AI Engine の演算器の利用効率が低下する。一方で、SpMV は疎行列を扱うため、同一行列サイズの GEMV と比較すると総データ転送量は少ない。これらの特性から、SpMV はメモリ帯域よりも演算器側がボトルネックとなりやすく、GEMV に比べて多くのコアを並列動作させる余地がある。したがって、いかに通信の制約を回避して効率的な並列実装を行うかが重要となる。前節で述べた通り、XDNA2 のアーキテクチャでは、Shim タイルおよび各コアタイルが保持するデータ入出力チャンネル数に制約がある。従って、コアを単純に並列配置すると、Shim タイル側の DMA チャンネルが不足し、データ供給が不可能になるという問題が生じる。この問題を解決するため、提案手法ではメモリタイルをデータハブとして利用した。以下では、 4×8 の二次元上に配置された XDNA2 のタイルのある 1 列において、4 つのコアタイルに SpMV の入力データ (疎行列 A・入力ベクトル x) と出力データ (出力ベクトル y) を転送する際のデータの動きを解説する。プログラムの開始時に、各コアへ入力ベクトル x を転送し、プログラム終了まで各コアの L1 メモリで保持する。次に行列 A の転送において、図 6 に示す通り、Shim タイルから 4 コア分に相当する行列データをメモリタイルへ一括転送する。その後、メモリタイルがデータをコア毎に分割し、各コアタイルへ個別に配信する。この「一括転送とメモリタイル内での分配」という転送手法により、Shim タイルの DMA チャンネル消費を抑制した。出力フェーズにおいても同様の構造を採用し、各コアから算出された部分的な出力ベクトル y を一旦メモリタイル内で集約し、Shim タイルへ向けて一括転送する。このようなメモリタイルによるデータの分配・集約機構を導入することで、最大 32 コアを用いた SpMV の並列実行を実現した。

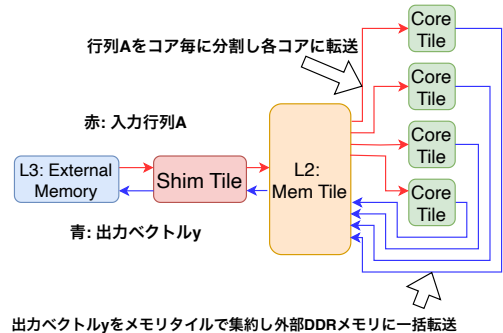


図 6: メモリタイルを経由したデータ転送

表 1: ハードウェア構成

項目	仕様
Processor	AMD Ryzen AI 9 HX 370
NPU Architecture	AMD XDNA 2
Memory Capacity	32 GB (8 GB \times 4)
Memory Type	LPDDR5-7500
Memory Bus Width	128 bit (32 bit \times 4 channels)

表 2: ソフトウェア構成

項目	バージョン / 詳細
OS	Ubuntu 24.04.3 LTS
XDNA 開発環境	
XDNA Driver & XRT	2.20.0
NPU Firmware	1.0.20.31
mlir-aie	1.1.3

4. 評価

本章では、提案手法である XDNA2 上の SpMV カーネルの性能を評価する。実験環境について、第 4.1 節で述べた後、第 4.3 節において、行列サイズを固定しスパース率を変化させた際の性能および実効メモリ帯域幅に与える影響を評価する。続いて、第 4.4 節において、データサイズと実効帯域幅の相関を調査し、提案手法における処理性能の飽和点と固定オーバーヘッドの影響について考察する。

4.1 実験環境

4.1.1 ハードウェア構成

実験には、AMD Ryzen AI 9 HX 370 プロセッサを搭載したデスクトップ PC を使用した。また、本機のプロセッサには NPU として XDNA2 が搭載されている。主なハードウェア仕様を表 1 に示す。実験時の NPU Power Mode は「Turbo」に設定されている。この設定により実験中の動作周波数は、MP-NPU Clock が 1,267MHz、H Clock が 1,800MHz に固定されていることを確認した。

4.1.2 ソフトウェア構成

本研究の実装および評価は、Ubuntu Linux 上で行った。詳細なバージョン情報を表 2 に示す。

4.2 評価指標と測定方法

性能指標として「実行時間」および「実効メモリ帯域幅」を用

いる。実効メモリ帯域幅は、プログラム実行中に NPU が入出力した総データ量 (GB) を、プログラムの実行時間 (s) で除算することで算出した。NPU が入出力した総データ量とは $y = Ax$ において、行列 A および入出力ベクトル x, y の合計データ量である。測定にあたっては、NPU の構成時やランタイムの初期化時のオーバーヘッドを除くため、最初の二回の実行をウォームアップとして除外し、3 回目の実行結果を評価対象とする測定を 5 回実施し、その平均値を測定値とした。

4.3 スパース率の変化に対する性能評価

本節では、行列サイズを固定した条件下でスパース率を変化させた際、提案手法の実行時間および実効メモリ帯域幅がどのように推移するかを測定・評価する。測定時の入力データには、Llama3-70B [8] の重み行列形状 (28,672 × 8,192) を模した ELL 形式のランダムな疎行列 A と、同モデルの入力サイズ (8,192 × 1) に対応するランダムなベクトル x を用いた。この疎行列 A は、全ての行の非ゼロ要素数を均一に設定している。一般的な疎行列を ELL 形式に変換する場合、行ごとの非ゼロ要素数のばらつきに応じてパディングが必要となり、これに起因する無駄な演算やデータ転送が発生する。対して本実験では全行の非ゼロ要素数が均一な行列を用いることで、このパディングによるオーバーヘッドを排除している。したがって、本実験は ELL 形式特有の構造的オーバーヘッドを含まない、提案手法の理想的な最大性能を評価するものである。行列 A の非ゼロ要素数を変化させることで、行列 A のスパース率を変化させその時の実行時間、GEMV に対する速度向上比、および実効メモリ帯域幅を測定・算出した。

4.3.1 ELL 形式によるデータ圧縮効果による SpMV の高速化

XDNA2 の 32 コア全てを使用した際にスパース率を変化させた際の実行時間の推移を図 7 に示す。図よりスパース率の上昇につれ一貫して実行時間は減少していることがわかる。また、第 3.2 節で用意した二種類のカーネルの実行時間に差はあまり見られないことがわかる。

次に、XDNA2 の 32 コア全てを使用した際の既存の行列ベクトル積 (GEMV) の実行に対する提案手法の速度向上比とデータ圧縮比の関係を図 8 に示す。スパース率 75% ~ 96.88% の範囲においては、GEMV に対する速度向上比とデータ圧縮比がほぼ一致している。一般に計算機性能が演算能力に依存する状況では、データ量を削減しても高速化の効果は限定的となる。これに対し、速度向上比がデータ圧縮比とほぼ一致するという本結果は、XDNA2 上での SpMV がメモリ律速であることを裏付けている。一方で、スパース率が 98.44% 以上の領域では、速度向上比の伸びがデータ圧縮比の伸びを下回っている。スパース率が 98.44% 以上の領域では図 7 より実行時間が数百 μ s まで短縮されている。そのためこの領域では、カーネル起動にかかるレイテンシや制御等の固定オーバーヘッドの影響が実行時間に対して大きくなり、データ圧縮効果通りの速度向上が得られなくなったと考えられる。

4.3.2 実効メモリ帯域幅の利用効率

実効メモリ帯域幅の測定結果を図 9 に示す。スパース率

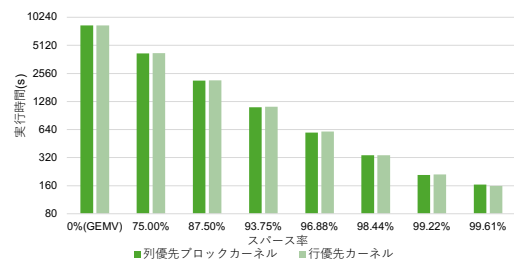
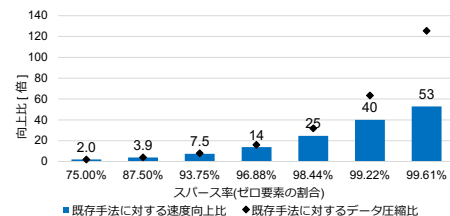
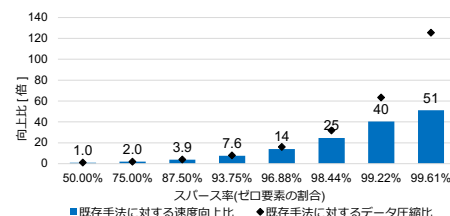


図 7: 32 コア使用時のスパース率ごとの実行時間



(a) 行優先カーネル

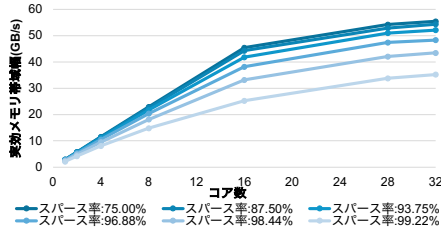


(b) 列優先ブロックカーネル

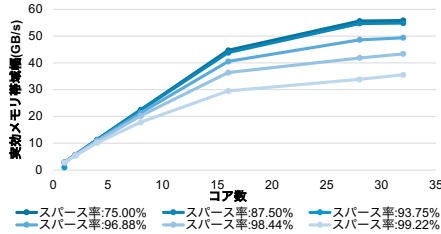
図 8: 32 コア使用時の GEMV に対する速度向上比とデータ圧縮比

75% ~ 96.88% の範囲では 32 コア使用した状態において、XDNA2 の理論メモリ帯域幅上限である 60GB/s の近辺まで帯域幅が出ており、SpMV においても GEMV と同様にメモリ律速になっていることがわかる。スパース率が 98.44% 以上では、コア数が多い場合に実効メモリ帯域幅の低下が見られる。これは前節で述べた通り、実行時間が短くなり、固定オーバーヘッドの影響が相対的に大きくなったためであると推察される。

高スパース率 (98.44% 以上) 領域における固定オーバーヘッドの影響を排除するため、行数を従来の 20 倍 (573,440 × 8,192) に拡大して実効メモリ帯域幅を再測定した (図 10)。従来サイズ (28,672 × 8,192) では最大 43.4GB/s に留まっていた帯域幅は、行数を拡大することで大幅に改善し、32 コア動作時には全スパース率において 52.8GB/s 以上を達成した。特筆すべき点として、高スパース率領域において行優先カーネルと列優先ブロックカーネルの性能特性に差異が確認された。スパース率 99.61% (ELL 形式における列数が 32) かつコア数が少ないときは、列優先ブロックカーネルが行優先カーネルを有意に上回る性能を示した。この要因はベクトル演算の効率性にあると考えられる。列優先ブロックカーネルは列方向に加算を行うため安定した性能を発揮する一方、行優先カーネルは行方向に加算を行うため、ELL の列数が小さい場合にベクトル演算のパイプライン化が阻害され、効率低下を招いたと推察される。

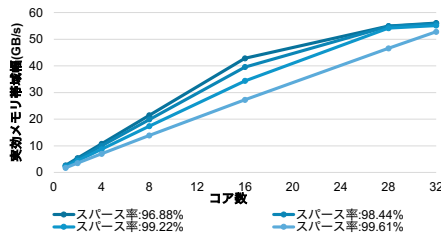


(a) 行優先カーネル

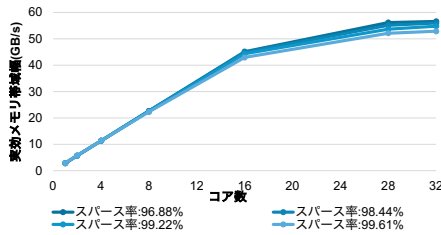


(b) 列優先ブロックカーネル

図 9: コア数毎のメモリ帯域幅



(a) 行優先カーネル



(b) 列優先ブロックカーネル

図 10: 行数を 20 倍に拡大した際のコア数毎のメモリ帯域幅

4.4 総データ転送量が実効メモリ帯域幅に与える影響

前節までの評価により、高スパース率領域における実効メモリ帯域幅の低下は、データ転送量の減少に伴う固定オーバーヘッドの顕在化が主因であると推察された。本節では、この挙動をより詳細に検証するため、スパース率を固定した条件下で行列サイズのみを変化させ、総データ転送量が実効メモリ帯域幅に与える影響を定量的に評価する。スパース率を 87.5% に固定し、行列サイズのみを変化させた際の総転送量と実効帯域幅の関係を図 11 に示す。この図より入出力データ量が多ければ多いほど実効メモリ帯域は増えるが、データ量が 32MB を越えたあたりから増加が緩やかになり最終的に XDNA2 の理想メモリバンド幅である 60GB/s に近い値で飽和することがわかる。

以上の測定結果により、実効メモリ帯域幅は総データ転送量に強く依存し、データ量が不十分な領域では固定オーバーヘッドの影響が顕在化し、実効メモリ帯域幅が低下することが明ら

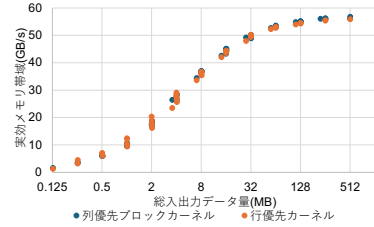


図 11: 総データ転送量と実効メモリ帯域幅の相関

かになった。したがって、XDNA2 の SpMV において 60GB/s という理想メモリ帯域幅に近い実効帯域幅を達成するためには、一定以上のデータ転送量が必要であることが示された。

5. おわりに

本研究では、固定長データ転送と親和性の高い ELL 形式を用い、XDNA2 に適した SpMV 演算法を提案し、その性能を評価した。その結果、 $28,672 \times 8,192$ の行列ベクトル積において、スパース率 87.5% かつ各行の非ゼロ要素数が均等に分布している理想条件下において、既存の密行列ベクトル積に対し 3.93 倍の高速化を達成した。

今後の課題として、非ゼロ要素数が行ごとに不均一な一般的な疎行列への対応が挙げられる。本研究で使用した ELL 形式では、要素数のばらつきに応じて最大要素数に合わせたパディングが発生し、スパース化によるメモリ帯域節約の効果が損なわれる場合がある。そのため、XDNA2 における固定長転送の制約を満たしつつ、可変長な非ゼロ要素を効率的に処理できる疎行列格納手法ならびに演算法の確立が今後の課題として挙げられる。

謝 辞

本研究の一部は JST CREST JPMJCR21D2 の支援による。

文 献

- [1] A. Deshmukh, V. Y. Raparti and S. Hsu: “Zen-Attention: A Compiler Framework for Dynamic Attention Folding on AMD NPUs” (2025).
- [2] Advanced Micro Devices, Inc.: “Versal Adaptive SoC AIE-ML v2 Architecture Manual (AM027)”, <https://docs.amd.com/t/en-US/am027-versal-aie-ml-v2> (2025). Accessed: 2026-01-26.
- [3] E. Hunhoff, J. Melber, K. Denolf, A. Bisca, S. Bayliss, S. Neuendorfer, J. Fifield, J. Lo, P. Vasireddy, P. James-Roxby and E. Keller: “Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface”, 2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 85–94 (2025).
- [4] Advanced Micro Devices, Inc.: “mlir-aie: IRON API and MLIR-based AI Engine Toolchain”, <https://github.com/amd/IRON>. Accessed: 2026-01-26.
- [5] Advanced Micro Devices, Inc.: “AI Engine API User Guide”, https://download.amd.com/docnav/aiengine/xilinx2025_2/aiengine_api/aie_api/doc/index.html (2025). Accessed: 2026-01-26.
- [6] Advanced Micro Devices, Inc.: “peano: AIEngine Fork of LLVM”, <https://github.com/Xilinx/llvm-aie>. Accessed: 2026-01-26.
- [7] L. Wilkinson, K. Cheshmi and M. M. Dehnavi: “Register Tiling for Unstructured Sparsity in Neural Network Inference”, Proc. ACM Program. Lang., 7, PLDI (2023).
- [8] AI@Meta: “Llama 3 Model Card” (2024). Accessed: 2026-01-26.