

Tensor コアの API の構造解析を用いた 拡張ライブラリの開発

大友 広幸^{1,a)} 横田 理央^{2,b)}

概要: 近年盛んに研究が行われている深層学習では行列積計算が多く行われ、行列積専用のプロセッサや行列積計算ユニットを持つプロセッサの開発が多く行われている。NVIDIA は Volta アーキテクチャより混合精度行列積計算回路である Tensor コアを GPU に搭載し、その理論性能は 100TFlop/s 以上に達する。Tensor コアを用いるための WMMA (Warp Matrix Multiply Accumulate) API ではメモリ上にある行列データを fragment と呼ばれるレジスタにコピーし Tensor コアに入力する。しかし WMMA API で提供されている関数は機能が限られており、利用目的によっては冗長なメモリアクセスを行うこととなる。そこで本研究では fragment の構造解析を通して WMMA API を用いた場合と比較し高速かつ省メモリに Tensor コアを用いるための拡張ライブラリを開発を行った。

1. はじめに

深層学習では行われる計算の 90% 以上が行列積計算となる場合もあり [5]、行列積計算を高速化することで深層学習の高速化が期待される。行列積計算の高速化を行うために多くの行列積専用プロセッサや行列積計算ユニットを搭載したプロセッサの開発が行われている [6]。Google では TPU (Tensor Processing Unit) と呼ばれる行列積計算専用プロセッサの開発が行われており [1]。また、Preferred Networks では MN-Core^{*1} と呼ばれる行列演算専用回路を搭載したプロセッサの開発が行われている。NVIDIA の GPU では Volta アーキテクチャより混合精度行列積和計算回路である Tensor コアを搭載し、1 GPU あたり混合精度積和演算において 100 TFlop/s 以上の性能を持つ。Tensor コアは行列積和計算 $D \leftarrow AB + C$ を行う演算器であるが、 A, B の精度は半精度、 C, D の精度は単精度または半精度である必要がある。Tensor コアでは内部の積算を完全精度 (Full precision)、足しこみを単精度 (FP32) で行うことにより、Tensor コアを用いずに半精度行列積を計算した場合と比較して計算精度の劣化が少ないという特長がある。一方で Tensor コアを用いて単精度行列積を計算する場合、積を計算する行列 A, B を半精度へ変換する必要があり、単

精度行列積としての精度は劣化する。この精度劣化は精度補正計算を行うことで緩和することが可能である [2], [4]。

Tensor コアは NVIDIA が開発を行っている CUDA の線形代数ライブラリである cuBLAS の GEMM (GEneral Matrix Multiply) 関数や CUTLASS^{*2} で利用可能な他、WMMA (Warp Matrix Multiply and Accumulate) API を用いてカーネル関数内で用いることが可能である。WMMA API では 1 Warp (32 スレッド) が協調して決められた大きさの行列積和計算を計算する関数が提供されている。この行列積和計算はメモリ上にある行列データを各スレッドが fragment と呼ばれるレジスタ配列に分割して保持し、これを Tensor コアへ入力することで計算を行う。Tensor コアはその演算性能の高さより B/F 比が相対的に小さく、アプリケーション開発においてはメモリアクセスの効率化が重要な課題となる。

本稿でははじめに背景として WMMA API の使い方と Tensor コアを用いた単精度精度補正行列積について述べる。次に WMMA API の fragment の内部構造の解析手法について述べ、これを用いた WMMA API の拡張ライブラリとして、4 種類の関数 `load_vector` 関数、`make_identity` 関数、`load_matrix_with_op` 関数、`foreach` 関数についてその設計と計算性能の向上について述べる。

表記

Tensor コアは Volta 世代より搭載された浮動小数点数行列積和計算回路である HMMA に加え、Turing 世代より搭

¹ 東京工業大学情報理工学院
東京都目黒区大岡山 2-12-1, 152-8550

² 東京工業大学学術国際情報センター

a) ootomo.h@rio.gsic.titech.ac.jp

b) riokota@gsic.titech.ac.jp

*1 <https://projects.preferred.jp/mn-core/>

*2 <https://github.com/NVIDIA/cutlass>

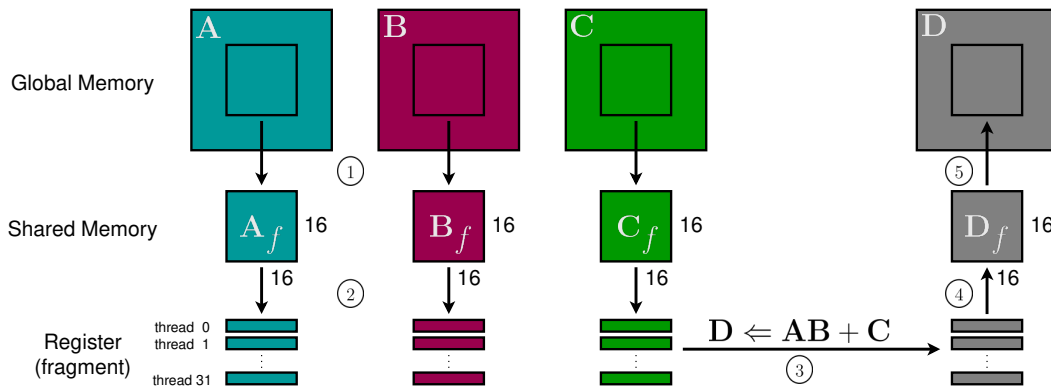


図 1: WMMA API を用いた行列積和計算 $D \leftarrow AB + C$ の処理の流れ

載された整数行列積和計算回路である IMMA がある。本稿では HMMA を指して Tensor コアと呼ぶ。

2. 背景

2.1 WMMA API を用いた行列積和計算

WMMA API では $m_f \times k_f$ 行列 A_f , $k_f \times n_f$ 行列 B_f , $m_f \times n_f$ 行列 C_f, D_f に対し行列積和 $D_f \leftarrow A_f B_f + C_f$ を行うための関数群と、各スレッドが入力行列の一部を保持するための fragment 構造体を提供している。このとき (m_f, n_f, k_f) は 3 種類 $(16, 16, 16)$, $(32, 8, 16)$, $(8, 32, 16)$ と決められている^{*3}。このため WMMA API を用いた任意の大きさの行列に対する行列積では、入力行列を分割し、これらの大きさの部分行列積を用いることで全体の行列積を完成させることとなる。

例として Global メモリにある行列 A, B, C の大きさが 16×16 である部分行列 A_f, B_f, C_f に対する行列積和 $D_f \leftarrow A_f B_f + C_f$ を計算する手順についての説明を行う (図 1)。

- ① Global メモリから Shared メモリへ A_f, B_f, C_f をコピー。
- ② Shared メモリにある A_f, B_f, C_f の fragment をレジスタ上に作成。作成には WMMA API の `nvcuda::wmma::load_matrix_sync` 関数を用いる。これにより 1 Warp 内の各スレッドは A_f, B_f, C_f の一部をレジスタに保持することとなる。
- ③ fragment を入力とし、Tensor コアを用いて

$$D_f \leftarrow A_f B_f + C_f$$

を計算。計算には WMMA API の `nvcuda::wmma::mma_sync` 関数を用いる。この際結果の行列 D_f は fragment として得る。

^{*3} 執筆時点で最新の CUDA 10.2.89 の場合。CUDA のアセンブリ言語である PTX では `mma.m8n8k4` と呼ばれる $(8, 8, 4)$ に対する命令があり、インラインアセンブラを用いて記述することで利用可能である

- ④ fragment 状態の D_f を Shared メモリへ書き出す。書き出しには WMMA API の `nvcuda::wmma::store_matrix_sync` 関数を用いる。

- ⑤ Shared Memory から Global メモリへ D_f をコピー。

`nvcuda::wmma::load_matrix_sync` 関数及び `nvcuda::wmma::store_matrix_sync` 関数には入出力行列に対し、そのメモリアドレスに対しては 128bit 境界の Alignment 制約が、Leading Dimension に対しては 16byte 長の倍数^{*4}という制約がある。しかし Global メモリにある行列 A, B, C がこれらの制約を常に満たしていることはない。そこで①及び⑤において Shared メモリを用いてこれらの制約を常に満たすよう調整を行っている。

2.2 fragment について

WMMA API の fragment は配列とその配列の大きさをメンバとして持つ構造体であり、その定義はソースコード 1 のようになっている。

ソースコード 1: `nvcuda::wmma::fragment` 構造体

```
1 template<typename Use, int m, int n, int k,
2     typename T, typename Layout>
3 class fragment {
4     enum {num_elements = XX};
5     T x[num_elements];
6 };
```

template 引数はそれぞれ次の意味を持つ。

- Use : A_f, B_f, C_f, D_f のどの行列の fragment かであり、 A_f であれば `matrix_a`, B_f であれば `matrix_b` であり、 C_f, D_f であれば `accumulate` を指定。
- m, n, k : 行列の大きさ m_f, n_f, k_f 。
- T : 行列の型。 A_f, B_f であれば `half`, C_f, D_f であれば

^{*4} FP32 であれば 2, FP16 であれば 4 となる。

ば `half` または `float` を指定.

- **Layout** : メモリ上の行列が列優先ならば `col_major`, 行優先ならば `row_major` を指定.

この様に `fragment` では `template` 引数にてその元の行列の大きさや用途を指定する必要がある, これにより内部配列 `x[]` の大きさ `num_elements` や, `x[]` が元の行列のどの要素を保持するかが異なる.

2.3 Tensor コアを用いた精度補正単精度行列積

Tensor コアへの積を計算する 2 つの行列 $\mathbf{A}_f, \mathbf{B}_f$ の入力は半精度とする必要があり, Tensor コアを用いて単精度行列積を計算する場合は半精度への変換を行う必要がある. しかし入力行列を半精度へ変換することから, その行列積の精度は Tensor コアを用いずに単精度演算器を用いて計算した場合と比較して劣化する. この問題を緩和する手法として, Tensor コアの内部の乗算及び足しこみが高精度で行われることを利用した精度補正を行うことで計算精度の劣化を抑えることができる.

単精度行列積 $\mathbf{C}_{FP32} = \mathbf{A}_{FP32}\mathbf{B}_{FP32}$ を計算することを考える. 精度補正を行わない場合は, 精度変換

$$\mathbf{A}_{FP16} \leftarrow \text{cnvtToFP16}(\mathbf{A}_{FP32})$$

$$\mathbf{B}_{FP16} \leftarrow \text{cnvtToFP16}(\mathbf{B}_{FP32})$$

を行った後, Tensor コアによる出力を単精度とする行列積 TC を用いて

$$\hat{\mathbf{C}}_{FP32} \leftarrow \text{TC}(\mathbf{A}_{FP16}, \mathbf{B}_{FP16})$$

と計算する.

一方, 精度補正を行う場合, 型変換により行列の各要素で発生する誤差からなる行列 $\Delta\mathbf{M}_{FP16}$

$$\Delta\mathbf{M}_{FP16} = \text{cnvtToFP16}(\mathbf{M}_{FP32} - \text{cnvtToFP32}(\mathbf{M}_{FP16})) \quad (1)$$

を用いて

$$\begin{aligned} \tilde{\mathbf{C}}_{FP32} \leftarrow & \text{TC}(\mathbf{A}_{FP16}, \mathbf{B}_{FP16}) \\ & + \underbrace{\text{TC}(\Delta\mathbf{A}_{FP16}, \mathbf{B}_{FP16}) + \text{TC}(\mathbf{A}_{FP16}, \Delta\mathbf{B}_{FP16})}_{\text{精度補正項}} \end{aligned} \quad (2)$$

と計算する. 以降 ΔM を誤差行列と呼ぶ.

3. fragment の解析

各スレッドの `fragment` の内部配列 `x[]` は元の行列を分割してその一部を保持する. WMMA API では 1 Warp (32 Threads) が協調して行列積と計算を行うため, Warp 内の各スレッドは別々の行列要素を `fragment` として保持することとなる. 本章では各スレッドが `fragment` として元の

行列のどの要素を保持するかの解析手法とその解析結果について述べる.

3.1 解析手法

`fragment` の内部配列の解析は次の手順で行う.

1. `fragment` として読み込む $p \times q$ 行列 $\mathbf{M}$ に対し, $\mathbf{M}_{i,j} = i + p \times j$ とする (i, j は 0 Origin).
2. `load_matrix_sync` 関数を用いて行列 $\mathbf{M}$ の `fragment` f を作成.
3. 1 Warp 中の各スレッドが持つ f の内部配列を `printf` 関数にて出力.

この手順により各スレッドの `fragment` が元の行列のどの要素を保持するかを調査することが可能である. 例としてこの解析を行うプログラムのソースコードをソースコード 4 に示す. このソースコードは表 1 の場合の `fragment` の解析を行う. この場合, `fragment` の内部配列の大きさ `num_elements` は 16 である.

template 引数	値
Use	<code>matrix_a</code>
<code>m, n, k</code>	(16, 16, 16)
T	<code>half</code>
Layout	<code>col_major</code>

表 1: ソースコード 4 により解析される `fragment` の詳細

3.2 解析結果

ソースコード 4 のプログラムを Virtual architecture `sm_70` でコンパイルし実行することにより得られる出力を表 2 に示す. この表は Warp 内のスレッド ID が `tid` であるスレッドの `fragment` の内部配列 `x[]` が保持する行列 $\mathbf{M}$ の要素を表す. ここで関数 `get_m_table(tid, i)` を, スレッド番号が `tid` であるスレッドの `x[i]` を返す関数とする. このとき表 2 を表す `get_m_table` 関数はソースコード 2 と定義可能である.

ソースコード 2: 表 2 を表す `get_m_table` 関数

```
1 unsigned get_m_table(
2     const unsigned tid,
3     const unsigned i) {
4     return (((tid >> 2) & 0b1) << 3)
5         + (((tid >> 4) << 2)
6         + ((tid & 0b11) << 4)
7         + ((i >> 2) << 6)
8         + (i & 0b11));
9 }
```

本研究では fragment の解析結果として, Virtual architecture sm_70, sm_75 それぞれにおける $(m, n, k) = (16, 16, 16)$ でのすべての template 引数の組み合わせに対する fragment の `get_m_table` 関数を得た.

4. WMMA API 拡張

Tensor コアはその理論性能が 100TFlop/s 以上であり, NVIDIA の GPU に搭載されている浮動小数演算回路と比較して高い計算性能を持つ. このためメモリアクセスの効率化が課題となるが, 本研究では fragment の解析を通してメモリアクセスを削減する WMMA API の拡張ライブラリ [3] の開発を行った.

4.1 load_vector_sync 関数

例えば QR 分解アルゴリズムの一つである HouseholderQR では, 行列ベクトル積やベクトルの直積を計算する必要がある. この様なベクトルを用いる計算を Tensor コアを用いて行う場合, ベクトルの fragment を作成する必要がある. 通常この fragment は次の手順で行列の 0 埋めを用いて作成される.

1. メモリ上で行列 M の全要素を 0 で初期化.
2. メモリ上で M の 1 列目に読み込むベクトルの要素をコピー.
3. `load_matrix_sync` 関数を用いて M の fragment を作成.

この手順では 0 埋めされるメモリ領域と 0 埋めを行うメモリアクセスが必要とされる. そこで WMMA API 拡張として, 0 埋めされるメモリ領域とメモリアクセスを必要とせずにベクトルの fragment の作成を行う `load_vector_sync` 関数の開発を行った (図 2).

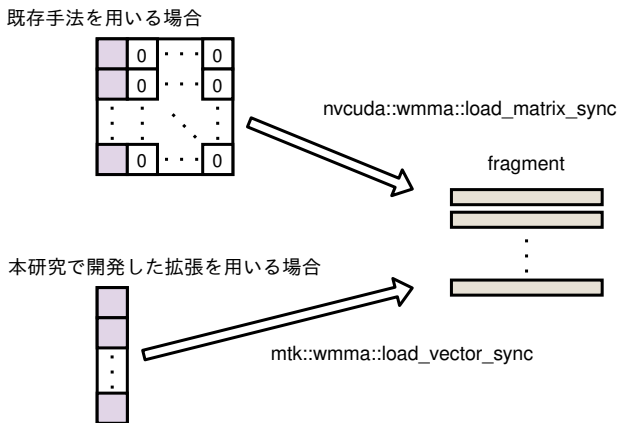


図 2: load_vector_sync 関数の概要図

4.1.1 設計方針

読み込むベクトルを $v[N]$ としたとき, `load_vector_sync` 関数は次の方針で実装を行う.

1. fragment の内部配列 $x[]$ の全要素に 0 を代入.
2. スレッド tid において `get_m_index(tid, i) < N` の場合 $x[i] = v[i]$. ただし実際の実装においては `get_m_index(tid, i) < N` を満たす i はコンパイル時に決定できるため, この予め決められた i についてのみ $x[i] = v[i]$ を行う.

これにより, ベクトルを列数 n_f の行列の fragment として読み込む場合, 本関数は 0 埋めを行う場合と比較して $1/n_f$ のメモリ消費量で fragment を作成することができる.

4.1.2 性能評価

`load_vector_sync` 関数の性能評価では, Batched QR 分解などで必要となるベクトルの Batched 直積計算を行い, ベクトルの fragment の作成に拡張関数を用いなかった場合に対する高速化率の調査を行った. Batched 直積計算では `batch_size` 個の Global メモリ上にあるベクトル $v_i \in \text{half}^{32}$ に対し直積 $v_i v_i^T$ を並列に計算する. この際 1 Warp あたり 1 つの直積計算を行うものとし, 用いる fragment の (m, n, k) は $(16, 16, 16)$ とする. 高速化率 R_{Speedup} をは本拡張関数を用いて計算した場合の計算時間 T_{mtk} と, 用いずメモリ上で 0 埋めを行うことでベクトルの fragment を作成した場合の計算時間 T_{nvcuda} を用いて

$$R_{\text{Speedup}} = \frac{T_{\text{nvcuda}}}{T_{\text{mtk}}}$$

と計算する.

この評価方法で `batch_size` を $2^{14} \sim 2^{20}$ と 2 倍ずつ変化させた場合の高速化率 R_{Speedup} が図 3 である. 評価は sm_70 と sm_75 それぞれに対して行った. この結果より, どちらの Virtual architecture においても本研究で開発した `load_vector_sync` 関数を用いることで高速にベクトルの fragment の読み込みが行えていることが分かる.

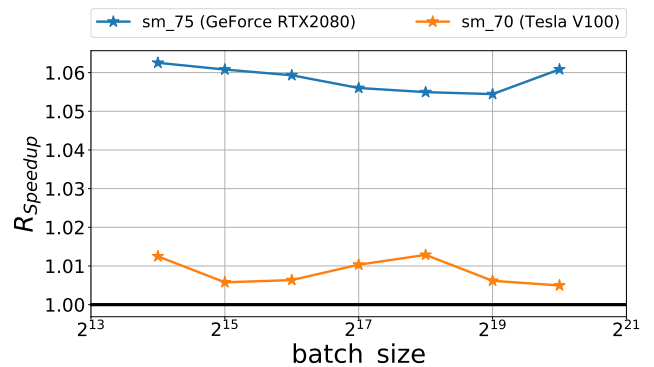


図 3: load_vector_sync 関数を用いることによる Batched 直積計算の高速化率

4.2 make_identity 関数

逆行列計算の安定化のための正則化を行う場合や HouseholderQR で陽に Householder 行列を計算する場合など、単位行列や単位行列の定数倍の行列 $\alpha \mathbf{I}$ を加算を行う場合がある。このような計算を Tensor コアで行う場合、 $\alpha \mathbf{I}$ の fragment が必要となる。通常この fragment はメモリ上に $\alpha \mathbf{I}$ を作成し、`nvcuda::wmma::load_matrix_sync` 関数で読み込むことで作成される。この行列は対角成分が α 、非対角成分が 0 と 2 値のみであるため、メモリ上に行列を陽に作ることなく fragment を直接計算することが容易である。そこでこの様な行列の fragment をメモリを必要とせずに作成する `make_identity` 関数の開発を行った。

既存手法を用いる場合

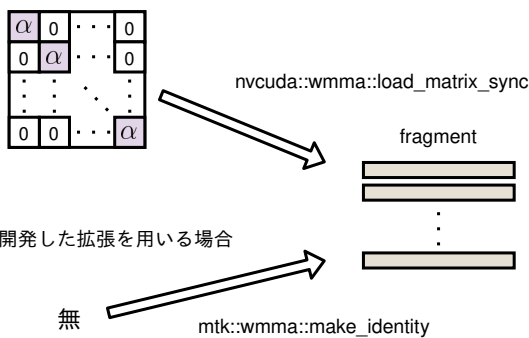


図 4: `make_identity` 関数の概要図

4.2.1 設計方針

`make_identity` 関数は正方行列の accumulator に対してのみ必要となるため、元行列の大きさ (m, n, k) が (16, 16, 16) で Use が accumulator の場合についてのみ実装を行う。実装方針は次のとおりである。

1. fragment の内部配列 `x[]` の全要素に 0 を代入。
2. スレッド `tid` において `get_m_index(tid, i) % 17 == 0` の場合 `x[i] =  $\alpha$` 。

この実装によりメモリを必要とせずに $\alpha \mathbf{I}$ の fragment を作成することができる。

4.2.2 性能評価

`make_identity` 関数の性能評価では、4.1.2 節の直積計算後に単位行列を加算する処理を加えた処理を行い、単位行列 fragment の作成に拡張関数を用いなかった場合に対する高速化率の評価を行った。ただしベクトルの読み込みにはどちらの場合も `load_vector_sync` 関数を用いた。高速化率 R_{Speedup} は 4.1.2 節と同様に算出する。

この評価方法で `batch_size` を $2^{14} \sim 2^{20}$ と 2 倍ずつ変化した場合の高速化率 R_{Speedup} が図 5 である。この結果より、どちらの Virtual architecture においても本研究で開発した `make_identity` 関数を用いることで高速に行列 $\alpha \mathbf{I}$ の fragment の作成が行えていることが分かる。

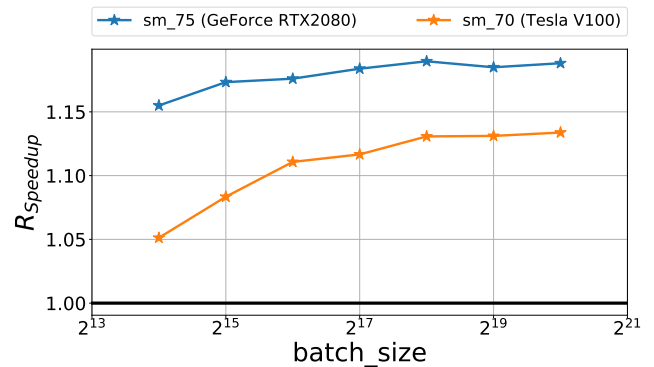


図 5: `make_identity` 関数を用いることによる高速化率

4.3 load_matrix_with_op_sync 関数

Tensor コアを用いた精度補正単精度行列積 (2.3 節) では誤差行列の fragment を必要とする。通常この誤差行列計算の fragment は、次の手順でメモリ上で誤差行列計算を行うことで作成される (図 6-A)。

- ① $\mathbf{M}_{\text{FP16}} \leftarrow \text{cnvtToFP16}(\mathbf{M}_{\text{FP32}})$ 。
- ② $\mathbf{M}_{\text{FP16}}$ を fragment として読み込む。
- ③ 誤差行列

$$\Delta \mathbf{M}_{\text{FP16}} \leftarrow \text{cnvtToFP16}(\mathbf{M}_{\text{FP32}} - \text{cnvtToFP32}(\mathbf{M}_{\text{F16}}))$$

をメモリ上で計算。

- ④ $\Delta \mathbf{M}_{\text{FP16}}$ を fragment として読み込む。

③ではメモリ上の 2 つの行列に対する減算により誤差行列計算の計算を行うが、既にレジスタ上に読み込まれている $\mathbf{M}_{\text{FP16}}$ の fragment を用いることでメモリアクセスの削減を行え、高速化が期待される (図 6-B)。

4.3.1 設計方針

`load_matrix_with_op_sync` 関数では、メモリ上の行列から fragment を作成する際に、要素ごとにラムダ関数を用いて任意の処理を行えるようにする。ラムダ関数 `func` は 2 引数とし、`load_matrix_with_op_sync` 関数内でソースコード 3 ように呼び出される。

ソースコード 3: `load_matrix_with_op_sync` 関数のソースコードの一部

```

1 // tid : Warp 内スレッド番号
2 // smem : 読み込む行列 M
3 // frag : 出力 fragment
4 for (i = 0; i < frag.num_elements; i++) {
5     const auto m_index = get_m_index(tid, i);
6     const auto v = smem[m_index];
7     frag.x[i] = func(i, v);
8 }

```

すなわち、`func` を `v` に関する恒等関数

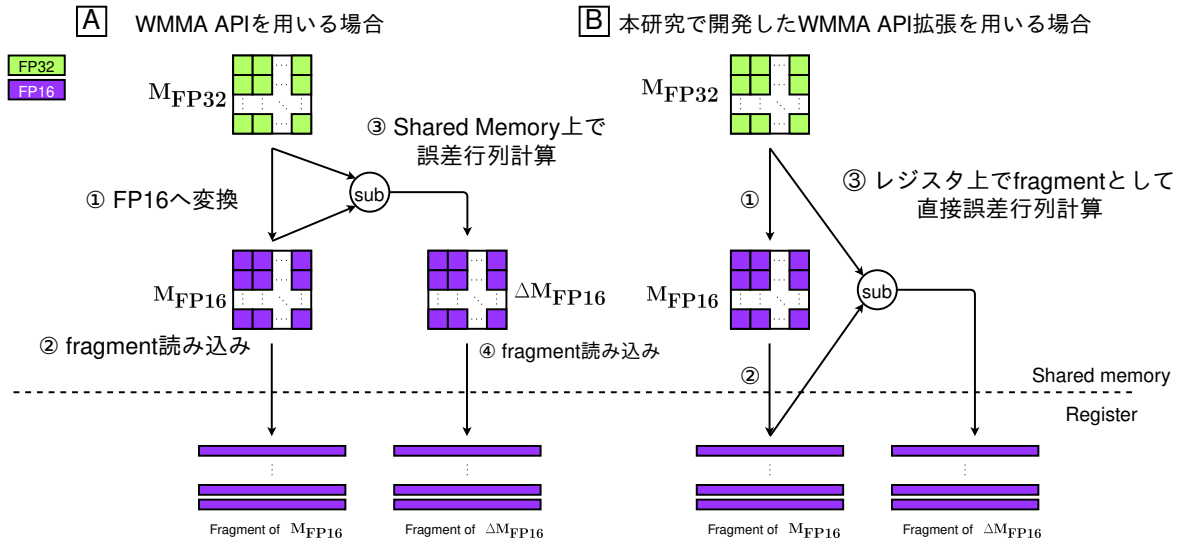


図 6: 誤差行列計算の比較. **A**: メモリ上で誤差行列計算を行い, `load_matrix_sync` 関数を用いて fragment として読み込む. **B**: `load_matrix_with_op` 関数を用いてレジスタ上で誤差行列 fragment を直接計算する.

```
1 func = [(i, v) {return v;}]
```

とすれば `load_matrix_with_op_sync` 関数は WMMA API の `load_matrix_sync` 関数と同様の処理を行うこととなる. ここで `func` 関数を

```
1 // frag_fp16 :  $M_{16}$  の fragment
2 func = [(i, v) {
3   return __float2half(
4     v - __half2float(frag_fp16[i])
5   );
6 }]
```

とすると, `load_matrix_with_op_sync` 関数を用いて誤差行列の fragment をメモリアクセスを削減した上で計算することができる.

4.3.2 性能評価

`load_matrix_with_op_sync` 関数の性能評価では, 2つの $m \times m$ 単精度行列に対する精度補正行列積計算 $C \leftarrow AB$ を行い, 誤差行列計算に拡張関数を用いなかった場合に対する高速化率の評価を行った. 行列積では結果の行列 C を 32×32 の部分行列に分割し, 1つの部分行列を 1 Block (256 Threads) が計算するよう実装した. 1 Warp は 2つの 32×32 行列の積 $A_{32 \times 32} B_{32 \times 32}$ を計算し, 足しこみを行う (図 7). 行列積 $A_{32 \times 32} B_{32 \times 32}$ は, それぞれの行列を 16×16 の部分行列 4 つに分割し, $(m, n, k) = (16, 16, 16)$ の fragment を用いて Tensor コアによる計算を行う. このため $A_{32 \times 32}$, $B_{32 \times 32}$ はそれぞれに対し 4 つの fragment が作られ, 誤差行列の fragment も 4 つ作られることとなる.

このベンチマークで行列の大きさ m を $2^7 \sim 2^{15}$ と 2 倍ずつ変化させた場合の高速化率が図 8 である. この結

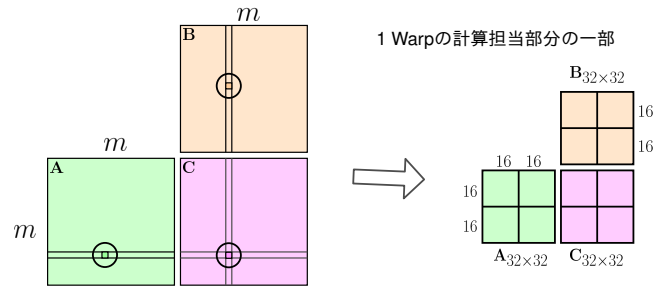


図 7: `load_matrix_with_op_sync` 関数による高速化率計算ベンチマークに用いる行列積の実装

果, Virtual architecture が `sm_75` では高速化し, `sm_70` では低速化した. $A_{32 \times 32}$, $B_{32 \times 32}$ の誤差行列の fragment はそれぞれ 4 つあるため, これらの作成のために `load_matrix_with_op_sync` 関数計 8 回呼ぶこととなる. `load_matrix_with_op_sync` 関数内部では `get_m_index` 関数の計算を行うが, `sm_70` ではこれを計 8 回行う計算コストがメモリ上で誤差行列計算を行うコストを上回ったため低速化したものと考えられる. そこで, 複数の誤差行列 fragment の作成に対し, `get_m_index` 関数の計算を 1 回のみとして計算コストを抑えた関数が 4.4 節の `foreach` 関数である.

4.4 foreach 関数

`load_matrix_with_op_sync` 関数を用いた N 個の誤差行列 fragment の計算では, `load_matrix_with_op_sync` 関数を N 回呼ぶ必要があり, `get_m_index` 関数の計算が N 回行われる. この `get_m_index` 関数の計算を 1 回とするよう改良した関数が `foreach` 関数である. すなわち `load_matrix_with_op_sync` 関数では

```
1 for (unsigned i = 0; i < N; i++) {
```

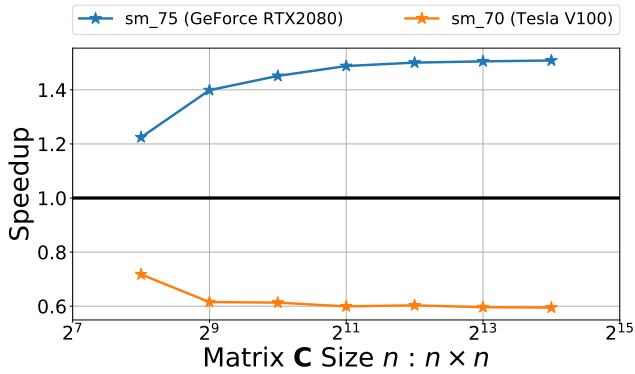


図 8: `load_matrix_with_op_sync` 関数を用いることによる精度補正行列積計算の高速化率

```
2 load_matrix_with_op_sync(
3     frag[i], mat_ptr[i],
4     ldm, func);
5 }
```

と `load_matrix_with_op_sync` 関数の外側で N 回の繰り返しを行っていた処理を

```
1 // m_index = get_m_index(j, tid)
2 foreach(
3     [] (j, m_index) {
4         for (unsigned i = 0; i < N; i++) {
5             frag[i].x[j] = func(j, ptr[i][m_index]);
6         }
7     });
```

のように `get_m_index` 関数の計算 1 回に対し N 個の fragment に対する要素の代入を行うことができ、これにより `get_m_index` 関数の計算回数を削減できる。

4.4.1 性能評価

`foreach` 関数の性能評価では、4.3.2 節と同様の単精度行列積を用いて行った (図 9)。この結果、本研究の拡張ライブラリを用いることで両 Virtual architecture において単精度精度補正行列積を高速に計算可能であることが確認された。

4.4.2 Bank conflict 回避計算による性能劣化

例えば表 2 の `tid = 0`, 2 の行を見ると、各 i に対し表の値が 32 ずれていることが分かる。このため、`load_matrix_with_op_sync` 関数のソースコード 3 において 1 Warp 中のすべてのスレッドが $i = 0$ から fragment の内部配列への代入を行うと、Shared メモリの Bank conflict が起こりうる。これは `foreach` 関数においても同様の可能性がある。そこで Bank conflict を回避するため、ループ変数 i の初期値をスレッド番号 `tid` ごとに `skew` だけずらす実装を行った。

4.3.2 節と同様のベンチマークを用いて Bank conflict の

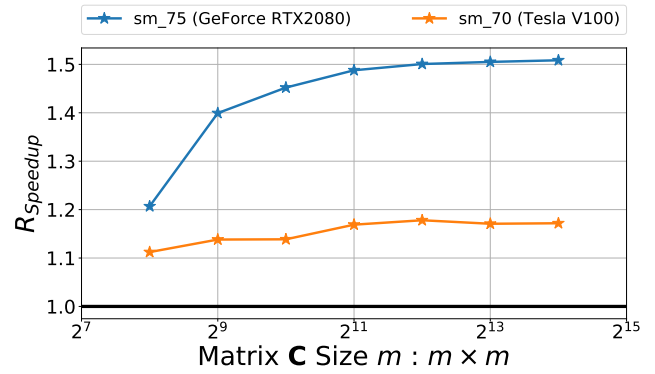


図 9: `foreach` 関数を用いることによる精度補正単精度行列積の高速化率

回避計算を行わなかった場合に対する高速化率を調査したところ、計算性能の低下が確認された (図 10)。これは Shared メモリの Bank conflict による遅延と比較して、その回避計算のコストが高いことによると考えられる。この実験から、本研究で開発を行った拡張ライブラリでは Bank conflict 回避を行わない方針とした。

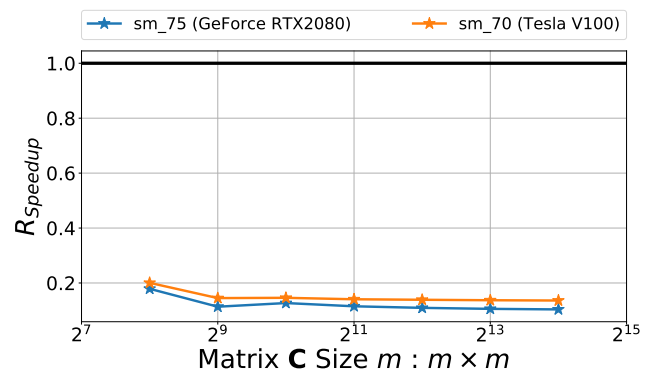


図 10: Shared メモリの Bank conflict 回避計算を行うことによる計算性能の低下

5. 結論

本稿では WMMA API の fragment の解析手法を提示し、解析結果を用いた WMMA API の拡張関数の開発手法を示した。この手法により開発した拡張関数を用いることで省メモリかつ高速な Tensor コアの利用が可能であることが確認された。

今後の課題

Tensor コアを用いたアプリケーションへの適用

既に Tensor コアを用いた TSQR (Tall-Skinny QR) への適用を行い、計算性能の向上が確認されている [4]。今後他のアプリケーションへの適用を行い、その有用性の検証を行う。

謝辞

本研究は JST CREST JPMJCR19F5 の支援を受けたものである。本研究は JSPS 科研費 JP18H03248 の支援を受けたものである。

参考文献

- [1] Norman P. Jouppi and 76 others. In-datacenter performance analysis of a tensor processing unit. *Proceedings - International Symposium on Computer Architecture*, Part F1286:1–12, 2017.
- [2] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pages 522–531, 2018.
- [3] Hiroyuki Ootomo. WMMA API extension, 2019. https://github.com/enpls0/wmma_extension.
- [4] Hiroyuki Ootomo and Yokota Rio. TSQR on Tensor Cores. *Research poster session presented at the Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 19*, 2019.
- [5] Dubey Pradeep and Khosrowshahi Amir. Scaling to Meet the Growing Needs of AI, 2016. <https://software.intel.com/en-us/articles/scaling-to-meet-the-growing-needs-of-ai>.
- [6] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and Benchmarking of Machine Learning Accelerators. *ArXiv 1908.11348*, 2019.

fragment 解析コード

ソースコード 4: fragment 解析コードの例

```
1 #include <mma.h>
2 constexpr unsigned warp_size = 32u;
3
4 __global__ void analyze_fragment() {
5     // 行列 M
6     __shared__ half matrix[16 * 16];
7     init_matrix(matrix, 16, 16);
8
9     // fragment の宣言
10    nvcc::mma::fragment<nvcc::mma::matrix_a, 16, 16, 16
11        , half, nvcc::mma::col_major> frag;
12    // fragment の作成
13    nvcc::mma::load_matrix_sync(frag, matrix, 16);
14
15    // fragment の内部配列を表示
16    for (unsigned tid = 0; tid < warp_size; tid++) {
17        if (tid == threadIdx.x) {
18            for (unsigned i = 0; i < fragment.num_elements; i++) {
19                printf("%03u,", static_cast<unsigned>(__half2float(frag.x[i])));
20            }
21            printf("\n");
22        }
23    }
24 }
25
26 int main() {
27     analyze_fragment<<<1, warp_size>>>>();
28     cudaDeviceSynchronize();
29 }
```

- `init_matrix(matrix, M, N)` 関数は $M \times N$ 行列を表す半精度配列 `matrix[]` に対し, `matrix[i] = __float2half(i)` を満たすようメモリを初期化する関数である.

fragment の解析結果例

tid	x[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
0	000	001	002	003	064	065	066	067	128	129	130	131	192	193	194	195
1	016	017	018	019	080	081	082	083	144	145	146	147	208	209	210	211
2	032	033	034	035	096	097	098	099	160	161	162	163	224	225	226	227
3	048	049	050	051	112	113	114	115	176	177	178	179	240	241	242	243
4	008	009	010	011	072	073	074	075	136	137	138	139	200	201	202	203
5	024	025	026	027	088	089	090	091	152	153	154	155	216	217	218	219
6	040	041	042	043	104	105	106	107	168	169	170	171	232	233	234	235
7	056	057	058	059	120	121	122	123	184	185	186	187	248	249	250	251
8	000	001	002	003	064	065	066	067	128	129	130	131	192	193	194	195
9	016	017	018	019	080	081	082	083	144	145	146	147	208	209	210	211
10	032	033	034	035	096	097	098	099	160	161	162	163	224	225	226	227
11	048	049	050	051	112	113	114	115	176	177	178	179	240	241	242	243
12	008	009	010	011	072	073	074	075	136	137	138	139	200	201	202	203
13	024	025	026	027	088	089	090	091	152	153	154	155	216	217	218	219
14	040	041	042	043	104	105	106	107	168	169	170	171	232	233	234	235
15	056	057	058	059	120	121	122	123	184	185	186	187	248	249	250	251
16	004	005	006	007	068	069	070	071	132	133	134	135	196	197	198	199
17	020	021	022	023	084	085	086	087	148	149	150	151	212	213	214	215
18	036	037	038	039	100	101	102	103	164	165	166	167	228	229	230	231
19	052	053	054	055	116	117	118	119	180	181	182	183	244	245	246	247
20	012	013	014	015	076	077	078	079	140	141	142	143	204	205	206	207
21	028	029	030	031	092	093	094	095	156	157	158	159	220	221	222	223
22	044	045	046	047	108	109	110	111	172	173	174	175	236	237	238	239
23	060	061	062	063	124	125	126	127	188	189	190	191	252	253	254	255
24	004	005	006	007	068	069	070	071	132	133	134	135	196	197	198	199
25	020	021	022	023	084	085	086	087	148	149	150	151	212	213	214	215
26	036	037	038	039	100	101	102	103	164	165	166	167	228	229	230	231
27	052	053	054	055	116	117	118	119	180	181	182	183	244	245	246	247
28	012	013	014	015	076	077	078	079	140	141	142	143	204	205	206	207
29	028	029	030	031	092	093	094	095	156	157	158	159	220	221	222	223
30	044	045	046	047	108	109	110	111	172	173	174	175	236	237	238	239
31	060	061	062	063	124	125	126	127	188	189	190	191	252	253	254	255

表 2: fragment の内部構造例. Warp 内でのスレッド ID が tid のスレッドの fragment の内部配列 x[i].