修士論文

メモリ空間の再利用に注目した再帰 Strassen アルゴリズム

LI RUIZHI

主指導教員 井口　寧

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和 5 年 6 月

概 要

行列の乗算は基本的な線形代数学の一部であり、機械学習、画像処理など様々な科学技術分野の数値計算に重要な役割を果たしている。ディープラーニング、エッジコンピューティングなど最近流行している技術は特に膨大なデータがあり、高精度が必要、大規模の行列乗算を必要となる機会が増えた。したがって大規模行列乗算を高速化する Strassen アルゴリズムが注目されている。

Strassen アルゴリズムは大きいサイズの行列の場合、標準的な行列乗算法と比べてより高速に計算できるアルゴリズムである。M が偶数である $M \times M$ の正方行列 $C = A \times B$ の場合、まず C、A、B はそれぞれ行列数が $\frac{M}{2}$、4 つのサブ行列に分け、Winograd で改良した Strassen アルゴリズムで 7 回の乗算と 15 回の加減算を行うことにより結果を求める。M が大きくなると Strassen アルゴリズムの演算回数が標準乗算より少なくなり、高速になる。A の行数と B の列数が対応する長方形行列は同じように考えることができ、padding 法で M が奇数の場合も対応できる。

Strassen アルゴリズムを GPU に応用するにはメモリ使用量の問題がある。これまでの研究には Malloc で確保した計算途中の結果を一時的に保存するための Temporary 行列を使用した、グローバルメモリ空間が有限な GPU にとって Strassen アルゴリズムの利点である大規模の行列乗算を応用しつらい問題がある。本研究では、NVIDIA GPU Tesla A100 を中心に、V100 と P100 を対比対象でメモリ空間の節約研究を行った。

メモリ空間を節約するため、再帰的な Strassen アルゴリズムにおいて、Malloc によるメモリ割り当てを避け、上位レベルの未使用行列を下位レベルに提供し、サポート行列として中間結果を保存する空間を提供することで、Malloc で確保した Temporary 行列全体を削除する方法を提案した。再利用できるメモリ空間がない問題に対して、メモリ空間使用分析アルゴリズムで下位 Level に提供できるメモリ空間を検索する上で計算順番の調整を行った。実験した結果、メモリ空間を重視した先行研究と比べて、Level-4 の計算に最大 18.11%のグローバルメモリが節約できた。結果により、最適化 Level の問題と更に高速化の問題が発見した。

実行速度が一番速い再帰の深度の問題、最適化 Level の問題に対して、GPU の乗算、加減算の性能関数の測定により判断する式を提案した。実験した結果、誤差が 2-7%あるが、各 Level の境界の関係が理論と合っている。

この上、更に高速化の探索について、先行研究を基にして、乗算だけではなく加減算の並列を含むことを中心に、部分同期を用いた並列処理手法を提案した。実験した結果、A100 で 40,960 サイズに対して、提案手法が最大 CUBLAS11.7 の標準行列乗算より 1.451 倍の SpeedUp が達成できた。

Abstract

Matrix multiplication is part of basic linear algebra and plays an important role in numerical computations in various scientific and technical fields such as machine learning and image processing. Recently popular technologies such as deep learning and edge computing have especially large amounts of data,requiring high precision,and there are more and more opportunities that require large-scale matrix multiplication. Therefore, the Strassen algorithm, which speeds up large-scale matrix multiplication, is attracting attention.

The Strassen algorithm is an algorithm that can calculate large-sized matrices faster than the standard matrix multiplication algorithm. For a square matrix $C = A \times B$, size $M \times M$, where M is even, C, A, and B are first divided into four submatrices each of size $\frac{M}{2}$, and the result is obtained by performing 7 multiplications and 15 additions/subtractions using the Strassen algorithm improved with Winograd. As M increases, the number of operations required by the Strassen algorithm becomes smaller than that of standard multiplication, making it faster. Rectangular matrices with A's number of rows and B's number of columns can be considered in the same way, and M can also be handled when odd using padding.

However, there is a memory usage problem when applying Strassen's algorithm to GPUs. Previous studies have used temporary matrices to save intermediate results during calculations, which require memory allocation using Malloc. This poses a challenge for GPUs with limited global memory space to apply the advantage of Strassen's algorithm for large-scale matrix multiplication. This study focuses on the NVIDIA GPU Tesla A100 and compares it with V100 and P100 to investigate ways to save memory space.

To save memory space, this study proposes a method that avoids memory allocation using Malloc in recursive Strassen's algorithm. Instead, the unused matrices at higher levels are provided to lower levels, and a space is provided to save intermediate results as support matrices. This allows the entire Temporary matrix allocated by Malloc to be removed. To address the problem of insufficient reusable memory space, a Memory space usage analysis algorithm was used to search available memory space at lower levels, while adjusting the calculation order. Experimental results showed that the proposed method could save up to 18.11% of global memory in Level-4 calculations compared to memory-focused previous studies. This revealed optimization level issues and further acceleration issues.

To address the issue of depth of recursive execution, which affects execution speed the most, this study proposes an equation to judge the performance of

GPU multiplication and addition/subtraction functions using measurements. The results of the experiments showed 2-7% error, but the relationship between the boundary of each level was consistent with theory.

Furthermore, to explore further acceleration possibilities, this study proposes a parallel processing technique using partial synchronization, focusing not only on multiplication but also on parallel addition and subtraction based on previous studies. Experimental results showed that the proposed method achieved up to 1.451 times speedup compared to the standard matrix multiplication in CUBLAS11.7 for size 40,960 on A100.

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

ディープラーニング、エッジコンピューティングなど深度学習に関する技術は近年流行している。これらの技術は特に膨大なデータ、高精度が必要、複雑な行列計算が特徴であり、数値計算に重要な役割を果たしている。CPU よりは速く処理できる、FPGA よりは汎用的な高い並列処理能力を持つ GPU がハードウェアとして、深度学習を含めて多くの分野に広く活用されている。しかし、実行効率が高い GPU 用のプログラムを作成には、GPU のアーキテクチャーを配慮してふさわしいプログラムが必要である。

GPU に得意な並列処理、特にパイプライン処理にとって一つ重要な研究方向が並列しやすい分割統治法である。分割統治法の問題とは、解決できない又はそのまま解決し難しい大規模の問題を小さな問題に分割して、解決することにより、最終的に最初の問題を解決する手法である。大規模行列処理を高速化するため、素朴な行列乗算より計算量が少ない、分割統治法の一つである Strassen アルゴリズムの高速化が今注目されている。

これまでの研究は Strassen アルゴリズムの CPU への応用が多く行なわれていたが、GPU への応用が最近の十年間ぐらいしかいない。Strassen アルゴリズムの GPU への最適化は大きくメモリ使用量と高速化二つ方向があるが、Strassen アルゴリズムのメモリ使用量が Strassen アルゴリズムの汎用化のボトルネックになった。

1.2　研究目的

本研究ではメモリ使用量の具体的な構成を解明する。そして、Winograd により改良した Strassen アルゴリズムの入力行列のメモリ空間を再利用することでメモリ使用量を減らすプログラムの実現を最初の目的にする。具体的の解決すべき課題は第 2.4 節に紹介する。

1.3　本論文の構成

上記の目的を達成するため、第 2 章に関連研究を報告する。本研究の提案三つの部分になる、第 3 章、第 4 章及び第 5 章で分けて報告する。具体的には：

第 2 章は本研究中に使用した機材、ツール、アルゴリズムと先行研究及び解決すべき問題を紹介する。

第 3 章はメモリ使用量を減らす目的としての Strassen-RUM 法と Strassen-RUM 法を達成するための補助アルゴリズム、再利用できるメモリ空間を分析するための「メモリ空間使用分析アルゴリズム」を紹介する。これが最初の目的、メモリ節約を目指す論点である。実験の結果により、第 4 章、第 5 章の問題を拡張問題として研究する。

第 4 章は Strassen-RUM 法により発見した Strassen アルゴリズムが異なる Level に対して、実行速度が違う、最適化 Level の選択問題と解決策、Strassen-RUM 法の最適化 Level を選定するための提案 2 を紹介する。

第 5 章は Strassen-RUM 法と提案 2 の結果を基にして更に高速化の余地があるかの探索と結果としての並列提案 (提案 3) を紹介する。

第 3 章, 第 4 章及び第 5 章の最後に各章の提案についての実験をする、実験環境と条件は第 3.4.1 節にまとめる。

論文中、重要な定義などが**太字**で表示する。

第2章　関連研究

2.1　はじめに

この章では先ず本研究にアルゴリズムの実行するハードウェア：GPU のアーキテクチャーの構造と演算に使うライブラリを説明する。その次に本研究のテーマ：Strassen アルゴリズムの概要と今までの研究手法、特に本研究の先行研究として採用した手法を紹介する。最後に先行研究の問題点、本研究で解決すべきの課題を説明する。

2.2　GPU 概要

本研究は Strassen アルゴリズムの GPU への実装をテーマにした研究である。この節は研究中使う GPU の基本出来なアーキテクチャーとよく使われる NVIDIA 開発した高性能演算ライブラリ CUBLAS を紹介する。

2.2.1　GPU のアーキテクチャー

GPU(Graphics Processing Unit) とは、グラフィックス処理に特化したコンピューターの拡張カード又は単体デバイス、定型化かつ大規模の演算に適したプロセッサである。GPU は CPU と比べて数倍の高速計算能力がある。2023 年の時点で NVIDIA 最新バージョン高速計算向き GPU H100 は 48TFlops(Tera Floating-point operations per second) 単精度浮動小数点演算ができる。これと比べて Intel の最新バージョン CPU i9-12900K は理論的に 614.4GFlops(GigaFLOPS) 単精度浮動小数点演算速度しかない、GPU と 78 倍の差がある。発展から見れば、GPU と CPU の計算速度の差が増えてつつある。

この計算能力の違いの本質は、算数演算装置 (ALU：arithmetic and logic unit) の数が GPU が圧倒的に多いからだ。それに、図 2.1 [1] は CPU と GPU のアーキテクチャーの違いを示した。GPU のアーキテクチャーは複数の ALU に制御機構 (Control) と L1 キャッシュ(Cache)、共有キャッシュを分配して SM(Straming Multi-processer) に構成する、CPU と比べて ALU の数が圧倒的に多い。ALU は大きく二つ種類に分けられる、CUDACore と TensorCore。CUDACore は普通の浮動小数点ユニット、GPU クロックごとに値の乗算を実行できる。TensorCore はテンソル

または行列演算を実行するために設計された専用の実行ユニット、クロックごとに64回の浮動小数点混合積和 (FMA) 演算を実行できる、図 2.2 [2] はFP16精度とFP32精度のTensorCoreの乗算と累積のイメージである。GPUのAmpere世代からFP64を直接演算できるTensorCoreがあり、今回の実験は高性能演算ライブラリCUBLASにより、行列乗算中この特性を利用した。

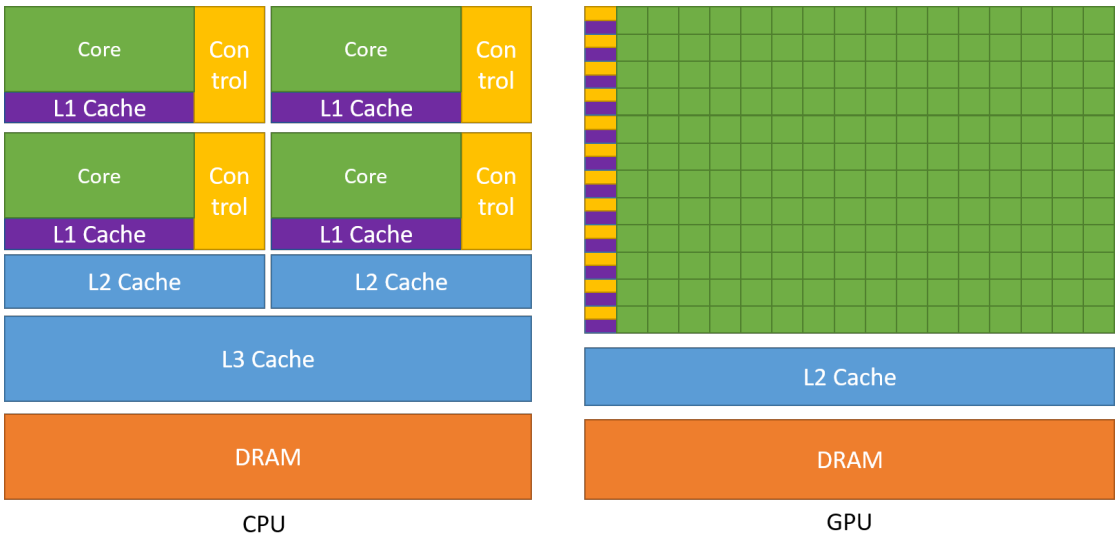


図 2.1: CPUとGPUの物理的なアーキテクチャーの比較 ([1] の図 The GPU Devotes More Transistors to Data Processing から引用)

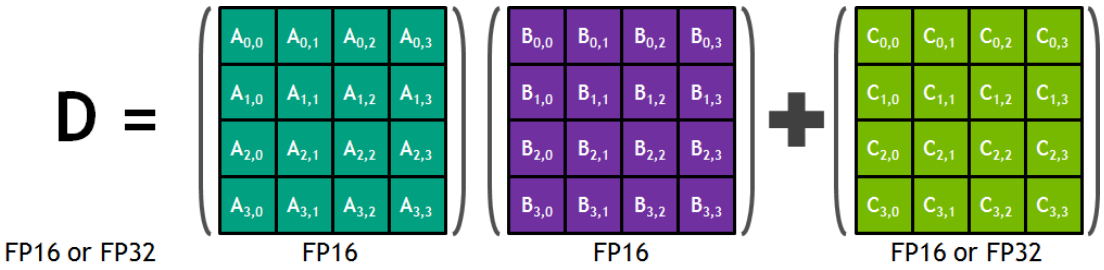


図 2.2: Tensor Core 4x4x4 行列の乗算と累積 ([2] の Figure 1 から引用)

GPUのメモリはCPUと比べて種類が多い。速度順番から見れば、一番速いのがレジスタ、この後がL1キャッシュと共有メモリ、この三種類はSM内に訪問できる、オンチップメモリと呼ばれる。すべてのSMが訪問できるメモリは速度順に、L2キャッシュ、定数メモリ、テクスチャメモリとグローバルメモリである。定数メモリとテクスチャメモリはSRAMで作り、定数とテクスチャ専用の読み取り

のみのメモリ。グローバルメモリが GPU にデータを格納する主な方法であり、画像、オーディオ、ビデオ、計算データなど、さまざまな種類のデータを格納できる。図 2.3 [3] は NVIDIA A100 の SM のアーキテクチャを示す。図 2.4 [3] は NVIDIA A100 の全体的なアーキテクチャを示す。

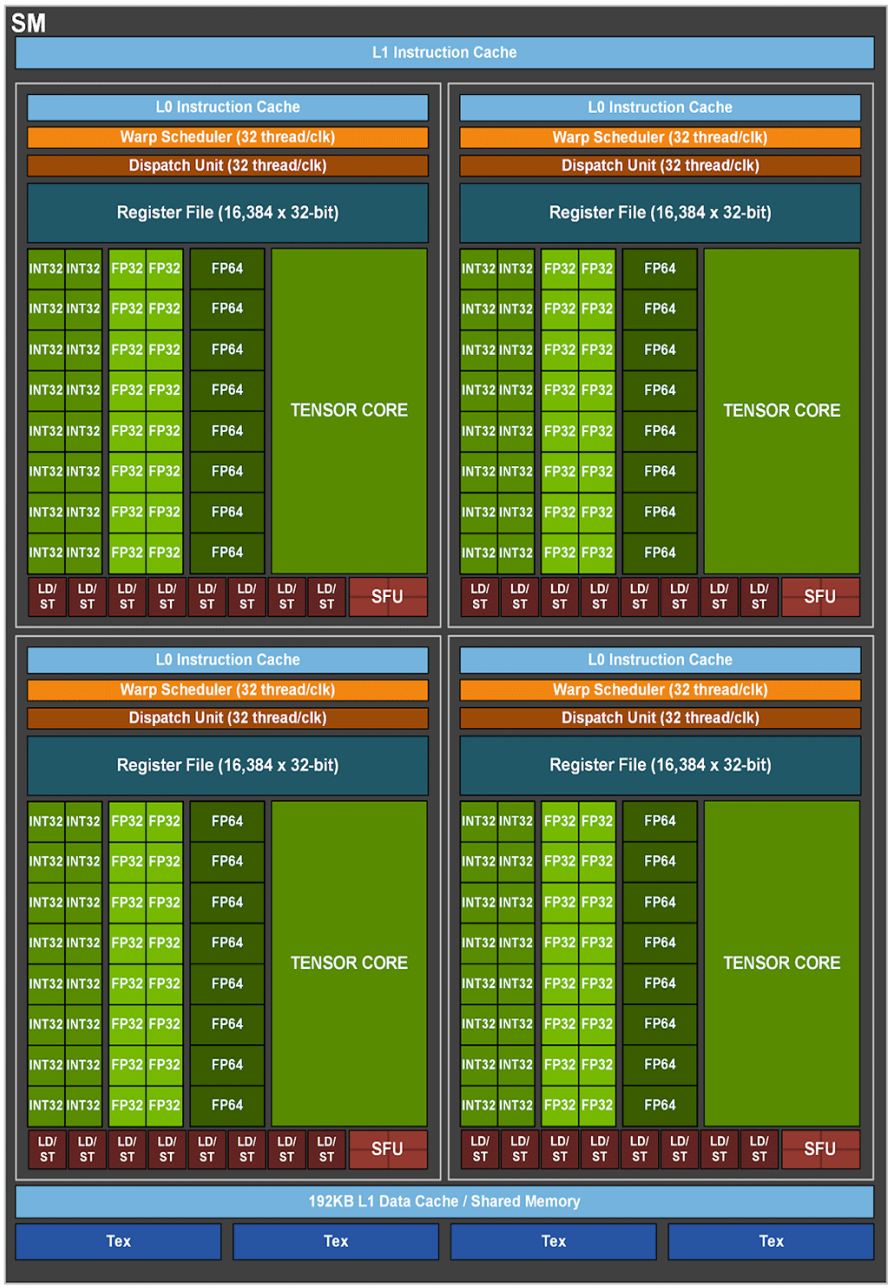


図 2.3: NVIDIA A100 の SM のアーキテクチャー ([3] の図 4 から引用)

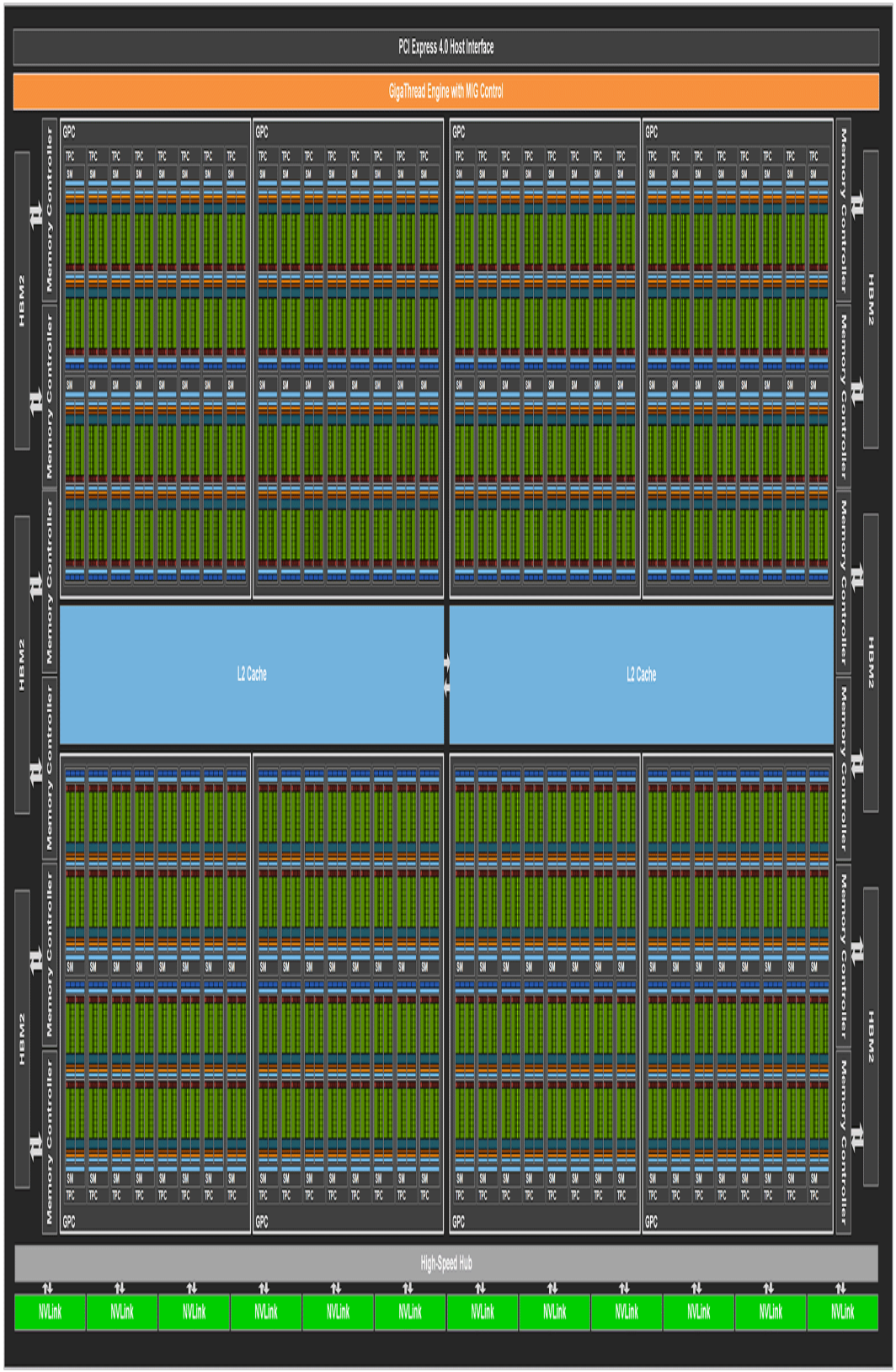


図 2.4: 128 個の SM を搭載した NVIDIA A100 アーキテクチャー ([3] の図 5 から引用)

2.2.2 GPU プログラム実行のアプローチ

CUDA プログラムを並列実行するために、計算任務の分割が必要である。基本的な分割単位は CPU と同じ thread である、thread32 個が Warp と呼ばれるグループに構成する。Warp は GPU 演算のスケジューリングユニット。Block は複数の Warp のグループの集合、SM に割り当てられると別の SM に割り当てられない。Grid は Block のグループの集合。つまり、一つのカーネルが生成した全部の thread が Grid を構成し、さらに Block、Warp に分けられる。図 2.5 [1] は thread、Block と Grid の関係を示す。

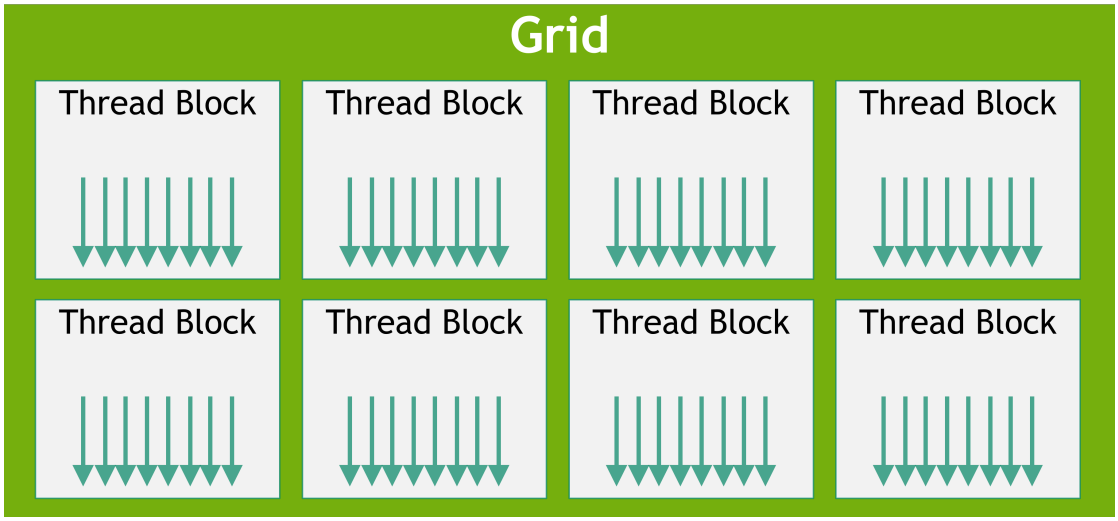


図 2.5: thread、Block と Grid のイメージ図 ([1] の Grid of Thread Blocks から引用)

メモリから見れば、すべての thread は L2 とグローバルメモリをアクセスできる。L1 キャッシュと共有メモリをアクセスできるのが SM に割り当てられる Block のみ、共有メモリは Block ごとに分けられて、Block 内の thread が共有できる。

GPU の実行スタイルは SIMT(Single Instruction Multiple Thread) と呼ばれる。CPU で行われる SIMD(Single Instruction Multiple Data) と SIMT は同じ一つの指令で複数の thread をコントロール方法だが、違いがある。SIMD はデータを中心に、各 thread がデータを利用し、同じ流れで並列演算を処理する。SIMT はアルゴリズムを選択実行できる複数の thread の集合、Warp の動き方である。例えば、あるアルゴリズムに if の文がある、このアルゴリズムを実行する Warp 内に 32 個 thread の中、16 が if の条件を満たす、半分が else を実行する。SIMT の場合、Warp 内全ての thread が実行の流れに if と else の実行時間をかかる、if を実行する場合、if の条件を満たさない thread が何も実行しないで待つ、else も同じである。この実行方法は SIMD の並列とは異なる、今回の実験を用いた CUBLAS ライブラリの行列乗算と加減算を理解するために、非常に重要である。

2.2.3 GPU の Stream

GPU の Stream は関連性なくの操作を同時に実行できるような仕組みである。Stream に入れる操作は CPU と GPU(Host と Device) 間のデータ転送、カーネルの呼び出しなどが含まれている。Stream の仕組みのイメージが二層車道である。二層は”メモリ接続”と”演算”で、それぞれに CPU との、GPU 内部の PCIE の帯域幅、GPU の SM の数、ALU の数など演算性能に関係がある。どの”層”の依頼関係がない”車道”も同時に異なる Stream に実行できるが、帯域幅或は演算性能のボトルネックになったら、Stream 間の並列実行が遅くなるか、並列実行ができなくなって逐次実行になるか、並列実行の優位性が発揮できなくなることがある。

2.2.4 高性能演算ライブラリ CUBLAS

NVIDIA が GPU や並列プロセッサ向きプログラミングのモデル及びプラットフォームを提供するために、2006 年に CUDA(Compute Unified Device Architecture) を導入した。C/C++、Fortran を拡張した CUDA C/C++と CUDA Fortran 二つの言語でプログラミングができる。

CUBLAS ライブラリは線形代数ライブラリ BLAS(Basic Linear Algebra Subprograms) を NVIDIA CUDA runtime に実装した、GPU の機能を利用して大幅に高速化のライブラリである。CUBLAS は、数多くの行列演算をサポートしており、その中には、行列乗算、行列加減算、ベクトル内積、行列転置、行列分解、逆行列の計算、行列式の計算などが含まれている。機械学習、ディープラーニング、数値解析、科学技術計算、金融工学、ビジネスアナリティクスなどの領域によく使われている。

2.3 Strassen アルゴリズム

2.3.1 Strassen アルゴリズム概要

素朴な行列乗算方法の複雑度が $O(\mathrm{n}^3)$ である。大きいサイズの行列乗算を速めに計算するため、様々な行列高速演算方法が提出されていった。分割統治法を用いて、複雑度が $\mathrm{n}^{log_2 7} \sim \mathrm{n}^{2.807}$ の Strassen アルゴリズム [4] がこれらの一つである。

M が偶数である $M \times M$ の正方行列 $C = A \times B$ の場合、Strassen アルゴリズムはまず C、A、B をそれぞれ、行列サイズが $\frac{M}{2}$、4 つのサブ行列に分け、サブ行列の 18 回の加減算と 7 回の乗算で行列 C を求める。素朴な $M \times M$ の行列乗算を $\frac{M}{2}$ サイズのサブ行列に分割して計算すれば 8 回の乗算と 8 回の加減算が必要である。計算量から見れば、$\frac{M}{2} \times \frac{M}{2}$ の行列乗算の演算量が $\frac{(2M-1)M^2}{8}$、加減算が

$\frac{M^2}{4}$、乗算と加減算が次元の差別がある。サブ行列サイズ $\frac{M}{2}$ が一定サイズ以上、1 回の行列乗算が 7 回の行列加減算より遅くなれば、Strassen アルゴリズムは素朴な行列乗算速くなる。

長方形行列の場合もこの方法で分割することができる。奇数の場合は 0 のベクトルを増加し、偶数行列に変更して計算することができる、計算した結果から、0 のベクトルを削除すれば正しい結果になる、この方法は Padding と呼ばれる [5]。

Wingrad は加減算の数を 18 個から 15 個に減らして、実用性が最も高いと呼ばれる Strassen-Wingrad アルゴリズム [6] を作成した。今までの研究はほぼこの方法を基づいて研究していた。今 Strassen アルゴリズムの基準は普通 Strassen-Wingrad アルゴリズムを指す。

本論文中の Strassen-Wingrad アルゴリズムはそのままで Strassen-Wingrad アルゴリズムを表す、18 個加減算ある Strassen アルゴリズムは Strassen-Original アルゴリズムで、Strassen アルゴリズムは Strassen-Wingrad アルゴリズムを基にして改良した一族を表示する。

式 2.1 とアルゴリズム 2.1 は Strassen-Wingrad アルゴリズムを示している。

Temporary 行列が計算の中間結果を一時的に保存するための行列、入力行列と出力行列以外メモリ空間が必要である。アルゴリズム 2.1 の S_1 から S_8、M_1 から M_7、V_1 から V_3 が Temporary 行列である。

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{2.1}$$

アルゴリズム 2.1 Strassen-Winograd アルゴリズム

1: $S_1 = A_{21} + A_{22}$
2: $S_2 = S_1 - A_{11}$
3: $S_3 = A_{11} - A_{21}$
4: $S_4 = A_{12} - S_2$
5: $S_5 = B_{12} - B_{11}$
6: $S_6 = B_{22} - S_5$
7: $S_7 = B_{22} - B_{12}$
8: $S_8 = S_6 - B_{21}$
9: $M_1 = S_2 \times S_6$
10: $M_2 = A_{11} \times B_{11}$
11: $M_3 = A_{12} \times B_{21}$
12: $M_4 = S_3 \times S_7$
13: $M_5 = S_1 \times S_5$
14: $M_6 = S_4 \times B_{22}$
15: $M_7 = A_{22} \times S_8$
16: $V_1 = M_1 + M_2$
17: $V_2 = V_1 + M_4$
18: $V_3 = M_5 + M_6$
19: $C_{11} = M_2 + M_3$
20: $C_{12} = V_1 + V_3$
21: $C_{21} = V_2 - M_7$
22: $C_{22} = V_2 + M_5$

より高速するため、1993 年、Kumar ら [7] は M_1 から M_7 のサブ行列乗算が再び Strassen アルゴリズムを応用した、この Strassen アルゴリズムを再帰的に呼び出す方法は Strassen's Matrix Multiplication Algorithm と呼ばれる、本文はその略称、**Multi-Level Strassen** と呼ぶ。

Multi-Level Strassen のイメージは図 2.6 に示す。Multi-Level Strassen の特徴は：

- 上位 Level の乗算が計算ではなく Strassen アルゴリズムでサブ行列に分割し、加減算だけが普通に計算する。このような Level は **HighLevel** と呼ぶ。
- 一番低い Level の乗算が普通の行列積演算を行う。このような Level は **LowLevel** と呼ぶ。

GPU の場合、LowLevel の行列積演算、HighLevel と LowLevel の加減算が通常 CUBLAS の関数で計算する。

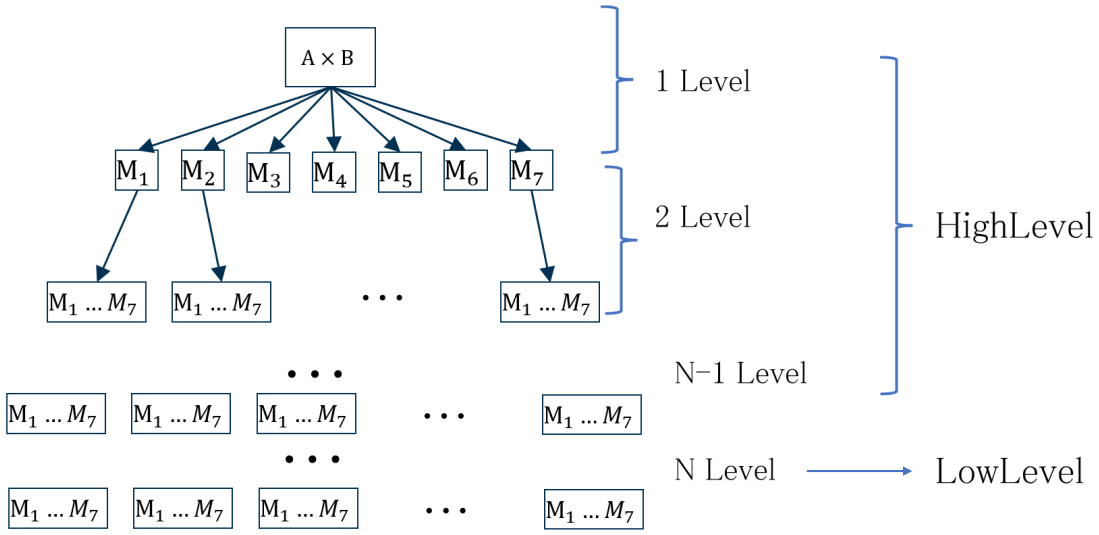


図 2.6: Multi-Level Strassen のイメージ図

2.3.2 Strassen アルゴリズムの発展

Strassen アルゴリズム最初は CPU 向けの研究である、1987 年から多くの研究が報告されていた。初期の研究中代表的な研究は David らの方法 [8] である、彼らの研究が当時のスーパーコンピューター Cray-2 と Cray Y-MP を用いて、LU 分解などを加速した。論文中は特に Malloc で作成した Temporary 行列について議論した、結果によると、CPU の場合、Temporary 行列を減らすと、データのメモリからの出し入れ回数も増え、通信による計算速度が落ちる場合がある、一方、Temporary 行列が多くなると、計算可能な行列サイズが小さくなるから、Strassen アルゴリズムの優位性が発揮できなくなる。90 年代以後、CPU の並列演算が注目され、プロセッサー間の通信とメモリ使用量の最適化が研究が増えた。その中、一番引用されているのが 1998 年、Thottethodi ら [9] はメモリ使用量を減らすことを目指して行列の四分木分解に基づくモートン順序と呼ばれる非標準の配列レイアウトを使用して、当時の方法より 25%以上の性能がある。2008 年、Kandegedara ら [10] は共有メモリを使って Strassen アルゴリズムを実行する報告があるが、これが高速化を目指す分散コンピュータに関する研究である。この後の CPU 上の研究は分散コンピュータ上の最適化が中心になる。

Strassen アルゴリズムの GPU への応用が 2011 年から報告されたが、今までは多くない。GPU についての研究は先行研究として第 2.3.3 節と第 2.3.4 節に紹介する。

特に、本研究の比較対象として高精度向け、倍精度を採用した研究、メモリ節約を目指す Lai らの方法と高速化手法がメモリ節約に応用できる大塚手法を詳しく説明する。

2.3.3 メモリ節約向けの先行研究

GPU のメモリ上には、2013 年、Yugopuspito ら [11] はメモリ節約を目指して単精度データに対して、NVIDIA GTX660 GPU で、CUBLAS 5.0 に 7 つ Temporary 行列あるアルゴリズムを実装した、これは GPU 向けの最初のメモリ最適化に関する論文である。最初、三月後、Lai ら [12] は CUBALS 5.0 の乗算関数を LowLevel に使って NVIDIA K10 と C2050 GPU で倍精度に対して、有限な GPU のメモリ空間を巡って Temporary 行列の数を 2 個減らした、この論文は多くの論文に引用されている、演算速度とメモリ使用量両方がいい成果がある。

Lai らの研究は逐次の提案手法と Strassen-Winograd アルゴリズムの並列二つの方向がある。

Lai らの逐次演算の順番 **(Steps)** はアルゴリズム 2.2 に示す[1]。逐次演算には Temporary 行列の再利用により、計算のスケジュールを改良した。Strassen-Winograd アルゴリズムの 18 個 Temporary 行列の中に途中で必要なくなる行列を分析し、Temporary 行列と出力行列の再利用により Temporary 行列の数を 2 個まで減らした。メモリ改良した 16 個 Temporary 行列を削除した一方、Temporary 行列の再利用により Steps 間の依頼関係が増えた。

逐次演算の速度について、Lai らの逐次演算が Multi-Level Strassen アルゴリズムに対応し、LowLevel の乗算 Step は CUBLAS 5.0 を利用した。K10 GPU に対して、単精度の最高 Level は Level-3、CUBLAS 5.0 と比べて加速比 **(SpeedUp)** が一番高いのが正方行列サイズ 15,360 の時、1.27 倍の SpeedUp が達成した。倍精度の最高 Level は Level-4 で、正方行列サイズ 8,192 の時 1.42 倍の SpeedUp が達成した。

Lai らの並列演算は Lai らの逐次手法の並列ではなく、依頼関係が少ない、並列性が高い Strassen-Wingrad アルゴリズムを並列化した。図 2.7 [12] は Strassen-Winograd アルゴリズムの依頼関係。

図 2.8 [12][2]は Lai らにより Strassen-Winograd アルゴリズムの並列化を示す。Lai らにより Strassen アルゴリズムの並列化は Stream 7 個使用した、7 並列になったが、Lai らの逐次演算スケジュールのメモリ使用量の利点が失ってしまった。

[1] n は Multi-Level Strassen アルゴリズムの Level を示す。

[2] CUBLAS Function の MA は CUBLAS の行列加減算、MM は CUBLAS の行列乗算を示す。

アルゴリズム 2.2 Strassen-Lai ら アルゴリズム

1: $T_1^n = A_{11}^n - A_{21}^n$
2: $T_2^n = B_{22}^n - B_{12}^n$
3: $C_{21}^n = T_1^n \times T_2^n$
4: $T_1^n = A_{21}^n + A_{22}^n$
5: $T_2^n = B_{12}^n - B_{11}^n$
6: $C_{22}^n = T_1^n \times T_2^n$
7: $T_1^n = T_1^n - A_{11}^n$
8: $T_2^n = B_{22}^n - T_2^n$
9: $C_{11}^n = T_1^n \times T_2^n$
10: $T_1^n = A_{12}^n - T_1^n$
11: $C_{12}^n = T_1^n \times B_{22}^n$
12: $C_{12}^n = C_{12}^n + C_{22}^n$
13: $T_1^n = A_{11}^n \times B_{11}^n$
14: $C_{11}^n = C_{11}^n + T_1^n$
15: $C_{12}^n = C_{12}^n + C_{11}^n$
16: $C_{11}^n = C_{11}^n + C_{21}^n$
17: $T_2^n = T_2^n - B_{21}^n$
18: $C_{21}^n = A_{22}^n \times T_2^n$
19: $C_{21}^n = C_{11}^n - C_{21}^n$
20: $C_{22}^n = C_{22}^n + C_{11}^n$
21: $C_{11}^n = A_{12}^n \times B_{21}^n$
22: $C_{11}^n = T_1^n + C_{11}^n$

2.3.4 高速化向けの先行研究

2011 年、Li ら [13] は高速化を目指して NVIDIA C1060 GPU で単精度データに対して行列サイズ $16,384 \times 16,384$ で Level-4 の Strassen アルゴリズムを作成して標準行列乗算関数 CUBLAS 3.0 の乗算の 1.36 倍の速度を達成した。2015 年、Khan ら [14] は高速化を目指して、NVIDIA k20C GPU で $10,240 \times 10,240$ で Level-1 の Strassen アルゴリズムで CUBLAS 5.5 の乗算の 2.2 倍を達成した。その後、2020 年大塚 [15] は高速化を目指して CUBLAS 10.0、NVIDIA K20、Tesla V100、Tesla P100 を使って LowLevel の Temporary 行列の削除を中心に高速化を研究した。

大塚の研究は 2020 年に報告され、使用した GPU は NVIDIA Tesla K20、Tesla P100 と Tesla V100 三種類世代が異なる GPU で、LowLevel の行列乗算部分が CUBLAS 10.1 を用いた。大塚の研究は三つのテーマがある。これらは Stream 並列数の GPU リソースによる制限、特に register 容量不足による制約 register

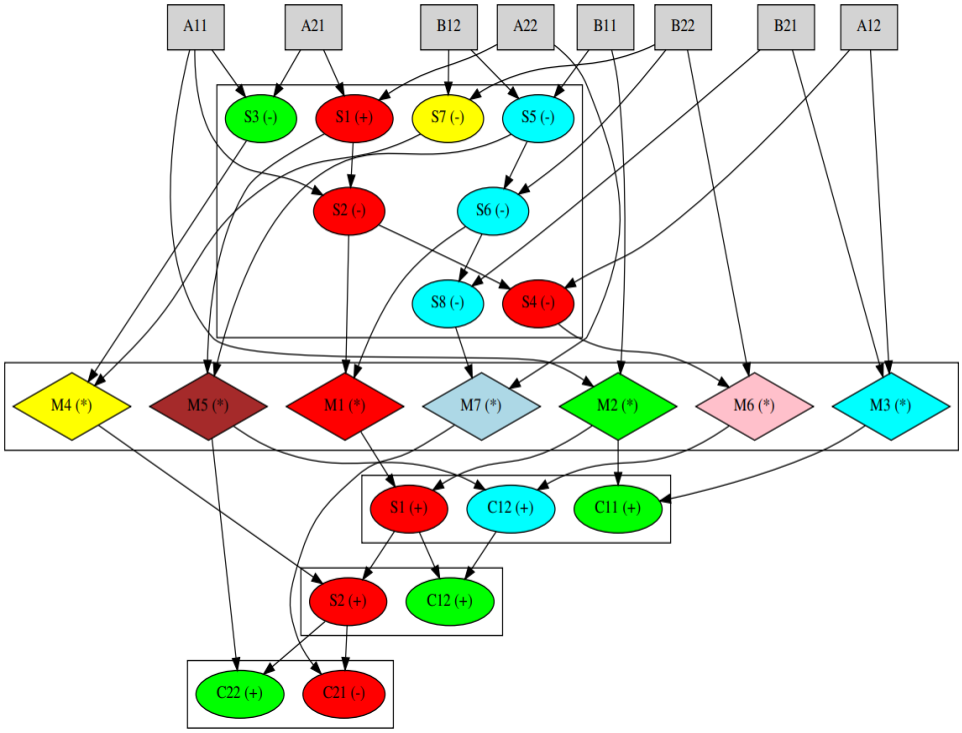


図 2.7: Strassen アルゴリズムの依頼関係 [13]

Step	Operation	CUBLAS Function	Stream
1	$S_1 = A_{21} + A_{22}$	MA	0
	$S_3 = A_{11} - A_{21}$	MA	1
	$S_5 = B_{12} - B_{11}$	MA	2
	$S_7 = B_{22} - B_{12}$	MA	3
	$S_2 = S_1 - A_{11}$	MA	0
	$S_6 = B_{22} - S_5$	MA	2
	$S_4 = A_{12} - S_2$	MA	0
	$S_8 = S_6 - B_{21}$	MA	2
2	$M_1 = S_2 \times S_6$	MM	0
	$M_2 = A_{11} \times B_{11}$	MM	1
	$M_3 = A_{12} \times B_{21}$	MM	2
	$M_4 = S_3 \times S_7$	MM	3
	$M_5 = S_1 \times S_5$	MM	4
	$M_6 = S_4 \times B_{22}$	MM	5
	$M_7 = A_{22} \times S_8$	MM	6
3	$S_1 = M_1 + M_2$	MA	0
	$C_{11} = M_2 + M_3$	MA	1
	$C_{12} = M_5 + M_6$	MA	2
4	$S_2 = S_1 + M_4$	MA	0
	$C_{12} = S_1 + C_{12}$	MA	1
5	$C_{21} = S_2 - M_7$	MA	0
	$C_{22} = S_2 + M_5$	MA	1

図 2.8: Strassen アルゴリズムの並列化 [13]

pressure [16] から Stream 並列数を予測、LowLevel の Temporary 行列を削除した 2 Level Strassen-OTSUKA 手法と GPU の発展により、将来の Strassen アルゴリズムの SpeedUp 予測式。Stream 並列数の予測と将来の SpeedUp の予測式は今回の研究外なので、詳しく説明しない。2 Level Strassen-OTSUKA 手法の LowLevel が Temporary 行列削除したので、メモリ節約に応用できるから、ここには大塚手法について紹介する。

大塚手法は高速化を目指した。注目点は Temporary 行列の処理時間、cudaMalloc() と cudaFree() 二つの関数がかかる時間である。

GPU Tesla V100 を使って測定したメモリ確保と解放の時間が Strassen アルゴリズム全体の計算時間の割合が図 2.9 [15] に示す。Strassen Lai らアルゴリズムと Strassen アルゴリズムの行列演算時間中 Temporary 行列の処理時間が行列サイズ小さい場合に大きな割合が占めることが明らかにした。

一方、Strassen アルゴリズムの分割ではなく通常の行列積演算が LowLevel だけある、そして、数多く LowLevel 行列積演算が上位の HighLevel の加減算より多い演算量があるから、演算量が LowLevel に集中している。そのため、LowLevel の Temporary 行列の処理時間が全体の演算速度に無視できない影響がある。

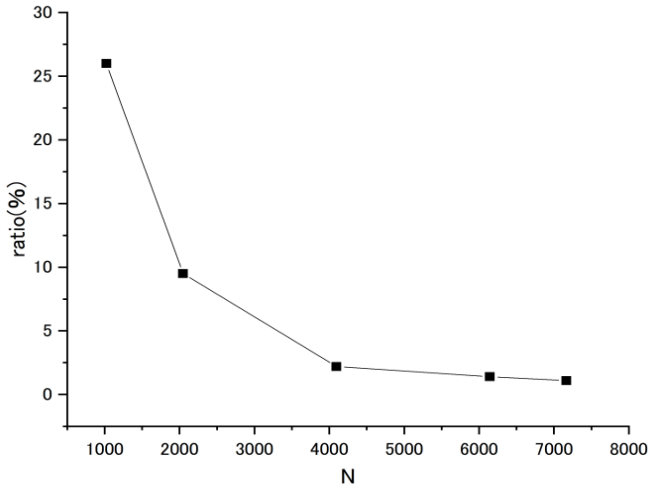


図 2.9: V100 の Strassen アルゴリズムの計算時間における Temporary 行列に関わる処理時間の割合 [15]

2 Level Strassen-OTSUKA 手法はすべての Temporary 行列を削除した LowLevel と 7 つ Temporary 行列ある HighLevel で構成された。逐次と並列二つのバージョンがある。大塚手法が LowLevel の Temporary 行列の削除より、Temporary 行列の処理時間を節約した。

大塚手法の逐次演算の HighLevel と LowLevel の演算 Steps はアルゴリズム 2.3、アルゴリズム 2.4 に示す。

大塚手法が LowLevel の Temporary 行列を削除した方法の原理が、Lai らの方法

に採用した出力行列の保有した値が演算の最初に演算には影響がないと同じ、入力行列の保有した値が演算中最後まで持っている必要がない。

原理を基にして、アルゴリズム 2.4 に大塚手法の LowLevel が一定の Step 後、保有した値が持っている必要がない入力行列 A と B の一部 A^2_{21}、B^2_{12}、B^2_{22}、B_{11} を再利用した。アルゴリズム 2.3 の HighLevel から見れば、B_{22}、B_{11}、A_{22}、A_{12} 及び B_{21} の一部が上書きされた、HighLevel の 7 つ Temporary 行列が上書きの影響を削除し、正しい結果を得るための中間結果を保存する行列である。

大塚手法が高速化のためとは言え、結果的に LowLevel の Temporary 行列を削除したので、メモリ使用量上のメリットもある。

アルゴリズム 2.3 逐次演算 Strassen-OTSUKA アルゴリズムの HighLevel

1: $S^1_1 = A^1_{11}$
2: $S^1_2 = A^1_{21}$
3: $S^1_3 = B^1_{12}$
4: $W^1_1 = S^1_1 - S^1_2$
5: $W^1_2 = B^1_{22} - S^1_3$
6: $M^1_1 = W^1_1 \times W^1_2$
7: $W^1_1 = S^1_2 + A^1_{22}$
8: $W^1_2 = S^1_3 - B^1_{11}$
9: $S^1_2 = W^1_1 \times W^1_2$
10: $W^1_1 = W^1_1 - S^1_1$
11: $W^1_2 = B^1_{22} - W^1_2$
12: $S^1_3 = W^1_1 \times W^1_2$
13: $W^1_2 = A^1_{12} - W^1_1$
14: $M^1_2 = W^1_1 \times B^1_{22}$
15: $M^1_2 = M^1_2 + S^1_2$
16: $W^1_1 = S^1_1 \times B^1_{11}$
17: $S^1_3 = W^1_1 + S^1_3$
18: $C^1_{12} = M^1_2 + S^1_3$
19: $S^1_3 = S^1_3 + M^1_1$
20: $W^1_2 = W^1_2 - B^1_{21}$
21: $M^1_1 = A^1_{22} \times W^1_2$
22: $C^1_{21} = S^1_3 - M^1_1$
23: $C^1_{22} = S^1_2 + S^1_3$
24: $S^1_3 = A^1_{12} \times B^1_{21}$
25: $C^1_{11} = W^1_1 + S^1_3$

アルゴリズム 2.4 逐次演算 Strassen-OTSUKA アルゴリズムの LowLevel

1: $C_{22}^2 = A_{11}^2 - A_{21}^2$
2: $C_{11}^2 = B_{22}^2 - B_{12}^2$
3: $C_{21}^2 = C_{22}^2 \times C_{11}^2$
4: $A_{21}^2 = A_{21}^2 + A_{22}^2$
5: $C_{11}^2 = B_{12}^2 - B_{11}^2$
6: $C_{22}^2 = A_{21}^2 \times C_{11}^2$
7: $A_{21}^2 = A_{21}^2 - A_{11}^2$
8: $B_{12}^2 = B_{22}^2 - C_{11}^2$
9: $C_{12}^2 = A_{21}^2 \times B_{12}^2$
10: $A_{21}^2 = A_{12}^2 - A_{21}^2$
11: $C_{12}^2 = A_{21}^2 \times B_{22}^2$
12: $C_{12}^2 = C_{12}^2 + C_{22}^2$
13: $B_{22}^2 = A_{11}^2 \times B_{11}^2$
14: $C_{11}^2 = C_{11}^2 + B_{22}^2$
15: $C_{12}^2 = C_{12}^2 + C_{11}^2$
16: $C_{11}^2 = C_{11}^2 + C_{21}^2$
17: $B_{11}^2 = B_{12}^2 - B_{21}^2$
18: $C_{21}^2 = A_{22}^2 \times B_{11}^2$
19: $C_{21}^2 = C_{11}^2 - C_{21}^2$
20: $C_{22}^2 = C_{22}^2 + C_{11}^2$
21: $C_{11}^2 = A_{12}^2 \times B_{21}^2$
22: $C_{11}^2 = B_{22}^2 + C_{11}^2$

大塚手法の目的である高速化の結果について GPU tesla V100 の実装結果、Lai ら法の逐次演算と比較すると、大塚逐次手法は 4000 から 8000 までの範囲で 0 個 Temporary 行列の LowLevel が 2 個 Temporary 行列の LowLevel より 1%から 2%高速になっていた。

大塚手法の並列スケジュールは表 2.1 に示す。大塚手法の並列化は Stream を用いて逐次手法の LowLevel の行列乗算を二つずつ並列した演算スケジュールを作成した。Lai らの Strassen-Winograd アルゴリズムの 7 並列と違い、二つ並列化の理由は Stream 並列数の GPU リソースによる制限から Stream 並列数を予測により、行列サイズの増加に伴い、並列化の能力が減らしている。V100 の場合 6144 サイズの行列サイズになると、実測並列数が 1 になってしまう。小さい行列に分割すると Strassen アルゴリズムが CUBLAS より遅い、大きいサイズに分割すると並列できないことを配慮して二つずつ並列化に決定した。演算の正しさを確保するため、並列化の前後は GPU のデバイス同期を二回実行する。

Step	アルゴリズム	Stream
1	$C^2_{22} = A^2_{11} - A^2_{21}$	
2	$C^2_{11} = B^2_{22} - B^2_{12}$	
3	$A^2_{21} = A^2_{21} \times A^2_{22}$	
4	$C^2_{12} = B^2_{12} + B^2_{11}$	
5	$C^2_{21} = C^2_{22} \times C^2_{11}$	0
	$C^2_{22} = A^2_{21} \times C^2_{11}$	1
6	$A^2_{21} = A^2_{21} - A^2_{11}$	
7	$B^2_{12} = B^2_{22} - C^2_{12}$	
8	$C^2_{12} = A^2_{12} - A^2_{21}$	
9	$C^2_{11} = A^2_{21} \times B^2_{12}$	0
	$A^2_{21} = C^2_{12} \times B^2_{22}$	1
10	$C^2_{12} = C^2_{12} + A^2_{11}$	
11	$B^2_{22} = A^2_{11} - B^2_{11}$	
12	$C^2_{11} = C^2_{11} + B^2_{22}$	
13	$C^2_{12} = C^2_{12} + C^2_{11}$	
14	$C^2_{11} = C^2_{11} + C^2_{21}$	
15	$B^2_{11} = B^2_{12} - B^2_{21}$	
16	$C^2_{21} = A^2_{22} \times B^2_{11}$	0
	$A^2_{11} = A^2_{12} \times B^2_{21}$	1
17	$C^2_{22} = C^2_{22} + C^2_{11}$	
18	$C^2_{21} = C^2_{11} - C^2_{21}$	
19	$C^2_{11} = B^2_{22} + A^2_{11}$	

表 2.1: 並列演算 Strassen-OTSUKA アルゴリズムの LowLevel

並列化の結果、GPU tesla V100 で実行すれば、行列サイズが 4096 の場合、Lai らの逐次方法と比べて 11%高速になって、8192 まで並列化の効果が次第に減少している、14,336 サイズになると、Lai ら法の 1.117 倍 SpeedUp と比べて大塚手法の並列方法が 1.124 倍ぐらい僅かな差だけある。Strassen アルゴリズムと CUBLAS の実行時間が同じ (SpeedUp=1) になる行列サイズも Lai らの逐次方法の 6656 サイズから 5200 に減らした。

大塚の論文によると、Lai らの最大 1.42 倍の SpeedUp が達成できない原因は行列演算のライブラリ、CUBLAS バージョンにより、行列演算の改良と配慮された。

2.4 解決すべき課題

大塚手法の LowLevel が Lai らの方法より Temporary 行列を 2 つから 0 つまで減らしたが HighLevel には再帰を正しく行うため 7 つの Temporary 行列に増加し

た。HighLevel の Temporary 行列が更に減らす余地がある。HighLevel にはしかし、LowLevel が再帰できるなら、HighLevel の Temporary 行列も削除できる、今メモリ使用量が一番少ない Lai らの方法よりメモリ使用量を減らせる。そのため、解決すべき課題として、まずは大塚手法の LowLevel が再帰できない原因を明らかにする、そして、速度を落とさず、再帰できるように、LowLevel と HighLevel のすべての Temporary 行列を削除する手法を提案する。

2.5 終わりに

本章は GPU のアーキテクチャ、高性能演算ライブラリ CUBLAS などを含め、本研究が利用したハードウェアについて紹介した。膨大な行列乗算が必要な技術の流行により、素朴な行列乗算より演算量が少ない Strassen アルゴリズムの発展を説明した。メモリ使用量を減らすために、先行研究の中特に各 Level の 18 個 Temporary 行列を 2 個まで減らした、GPU のメモリ使用量の減少を注目した Lai らの逐次演算法と LowLevel の Temporary 行列を削除した大塚手法が参考の価値がある。解決すべき課題として、大塚手法の LowLevel が再帰できない原因と解決法が必要である。

本研究の目標として、一番重要な方向が Strassen アルゴリズムのメモリ使用量の最適化である。第 3 章はこの目標を達成するため、問題の分析と提案を紹介する。

第3章　メモリ再利用により再帰 Strassenアルゴリズム

3.1　はじめに

第 2.4 節に説明した、大塚手法の LowLevel が再帰できれば、Lai らの方法よりメモリ空間を節約できる。本章は大塚方法の LowLevel の Temporary 行列削除方法を分析した上で再帰できない原因を解明する。そして、原因を基にして、再帰できる提案手法の考え方と実現条件を説明する。条件を判断するため、提案の一部である補助アルゴリズムを提案し、条件を満たしたメモリ使用量が減らせるプログラム Strassen-RUM 法を作成する。

3.2　先行研究に対するメモリ使用の分析

3.2.1　大塚手法 LowLevel が再帰できないの原因

第 2.3.4 節で紹介した、大塚は LowLevel が Temporary 行列なし、HighLevel が 7 つ Temporary 行列あるの Level-2 Strassen-OTSUKA アルゴリズムを作成した。大塚手法は Lai らの方法より高速化になった。メモリ使用量から見れば、HighLevel に大きいサイズの演算中、より多い Temporary 行列を使ったので、メモリ使用量は増えた。

大塚手法の LowLevel の Temporary 行列削除した手法が入力行列のメモリ空間を再利用である。再帰できない原因が LowLevel を再帰すると、生成した下位 Level に再利用した入力行列の一部が上書きされ、上位 Level の演算に影響を及ぼす。

図 3.1 を用いて LowLevel の再帰問題を説明する。例として、LowLevel を再帰して、2 Level Strassen アルゴリズムを作成すれば、Level-2 の Step 6 の行列乗算が再び Strassen アルゴリズムを利用して Level-3 を生成できる。Level-2 の Step 6 の A_{21}^2 が Level-3 の入力行列 A で、C_{11}^2 が B である。Level-3 の演算中の Step 4 で Level-3 の入力行列 A の一部 A_{21}^3 を上書きした、メモリ空間から見れば、上書きしたメモリ空間が A_{21}^2 の一部である。Level-3 が実行終わった後、Level-2 の Step 7 を実行する時、A_{21}^2 を使って演算が間違っている、最後の結果に影響を与えた。

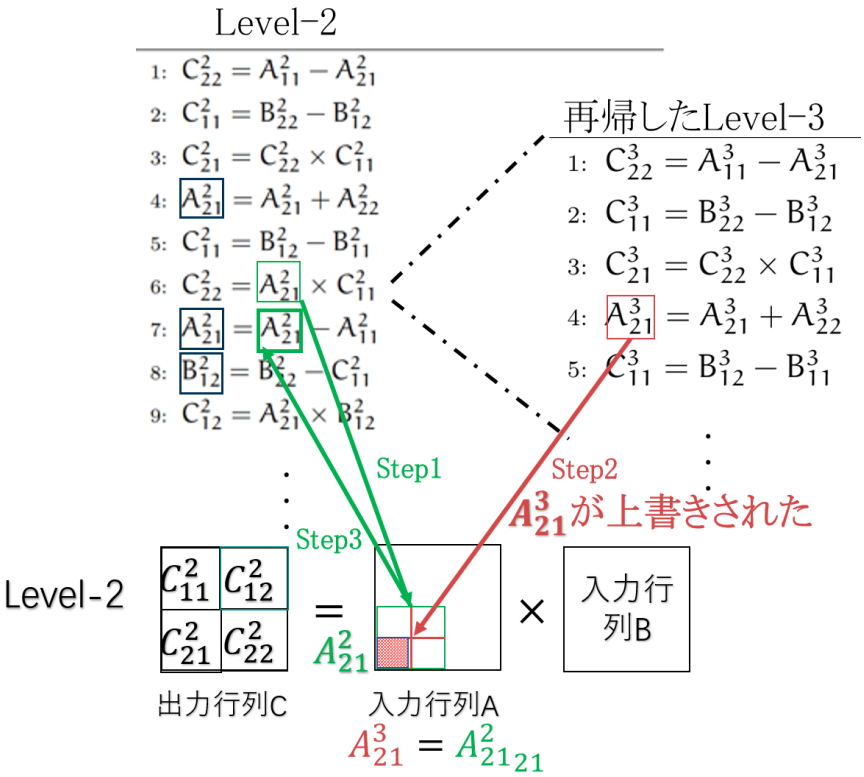


図 3.1: 大塚手法の LowLevel の再帰問題

要するに、大塚手法は同 Level のメモリの再利用により Temporary 行列なし LowLevel を作成したが、異なる Level のメモリ空間上問題がある。

3.2.2 再帰できない問題の解決策

異なる Level 問題を解決する発想として、図 3.1 の Level-2 の C^2_{12} は Step 9 まで利用されていない。この利用されていないメモリ空間を Step 6 の乗算に提供すれば、Level-3 の演算に入力行列のサイズと同じのメモリ空間を提供できる、これによって Temporary 行列と同じ追加メモリ空間を提供する効果があるが、cudaMalloc した空間ではないから GPU のメモリ使用量とメモリ確保と解放する時間に影響がない。

この下位 Level に追加メモリ空間を提供する、現有メモリ空間を使う仮想行列は**サポート行列**と定義する。サポート行列は下位 Level の入力行列と同じサイズ、4 つのメモリ空間を提供する。

Temporary 行列とサポート行列の区別は表 3.1 に示す。

	メモリ空間	どの Level に提供	機能
Temporary 行列	新たに作る	本 Level	追加メモリ空間
サポート行列	現有空間の再利用	下位 Level	を提供する

表 3.1: Temporary 行列とサポート行列の区別

3.2.3 メモリ空間節約率の理論値

大塚手法の LowLevel が再帰できると、Lai らの方法と大塚手法よりメモリ空間を節約できることの理論値が計算できる。

入力行列のサイズが正方行列 $M \times M$ の場合、$\frac{M}{2} \times \frac{M}{2}$ のメモリ空間を基準として、行列乗算 $A \times B = C$ が Temporary 行列なしの場合、二つ入力行列と一つ出力行列が 12 倍のメモリ空間が必要である。

Lai らの逐次演算方法が Level-N の場合、各 Level が二つ Temporary 行列あるから、必要なメモリ空間が約 14.6667 倍 (式 3.1)。

$$\lim_{N\to\infty} 12 + (2 + \frac{2}{4} \times \frac{1 - (\frac{1}{4})^{N-1}}{1 - \frac{1}{4}}) \approx 14.6667 \tag{3.1}$$

大塚手法は GPU Tesla V100 のグローバルメモリの制約で Level-2 だけ提案したが、7 つ Temporary 行列ある HighLevel を再帰的に拡張し、Level-N を作成すれば、必要なメモリ空間が約 21.3333 倍 (式 3.2)。

$$\lim_{N\to\infty} 12 + (7 + \frac{7}{4} \times \frac{1 - (\frac{1}{4})^{N-2}}{1 - \frac{1}{4}}) \approx 21.3333 \tag{3.2}$$

大塚手法の LowLevel が再帰できれば、cudaMalloc が使わないから、グローバルメモリから見れば、二つ入力行列と一つ出力行列だけ使用したメモリ空間が 12 倍しかない。理論的に大塚手法と比べて $(21.3333 - 12)/21.3333 \approx \mathbf{43.75\%}$、Lai らの方法と比べて $(14.6667 - 12)/14.6667 \approx \mathbf{18.18\%}$ グローバルメモリ空間を節約できる。

3.3 提案手法1 Strassen-RUM法:Temporary行列を削除した再帰Strassenアルゴリズム

Strassen-RUM(Reuse of unused matrices) 法はサポート行列を用いてすべての Temporary 行列を削除した Multi-Level に応用できるの再帰 Strassen アルゴリズムの作成を目的にする。Strassen-RUM 法には成立条件がある、これを満たす

ため、提案手法の一部としての補助アルゴリズムとして、「メモリ空間使用分析アルゴリズム」を提案する。

本節はまず提案手法 1 であり Strassen-RUM 法の概要を説明して、成立条件を明示する。この後メモリ空間使用分析アルゴリズムの手順で大塚手法を分析する、大塚手法が Strassen-RUM 法の成立条件を達成できない原因と解決法として提案手法である Strassen-RUM 法の Level-1 を説明する。そして再帰のため、Multi-Level への拡張とすべての Level の同一化を説明する。

3.3.1 Strassen-RUM 法の概要

Strassen-RUM 法の考え方がメモリ使用量を減らすために、再帰 Strassen アルゴリズムが必要な追加メモリ空間を新たに作らず、上位 Level のサポート行列から提供することにより、LowLevel だけではなく、全体的に Temporary 行列を削除する。

図 3.2 は Strassen-RUM 法の全体的なメモリ空間のイメージ。Level-1 は大塚手法の LowLevel と同じ、同 Level のメモリ再利用だけある、上書きされる部分が一部の入力行列。Level-2 からは同 Level のメモリ再利用以外、入力行列の上書きを避けるために、上位 Level から提供されたサポート行列のメモリ空間、異なる Level のメモリを再利用する。Level-2 から上書きされる部分が入力行列ではなく、サポート行列のメモリ空間を上書きされる。Level-2 からの各 Level のメモリ空間がサポート行列のメモリ空間を加えて、第 3.2 節のように $\frac{M}{2} \times \frac{M}{2}$ をのメモリ空間を基準として、16 倍空間がある。

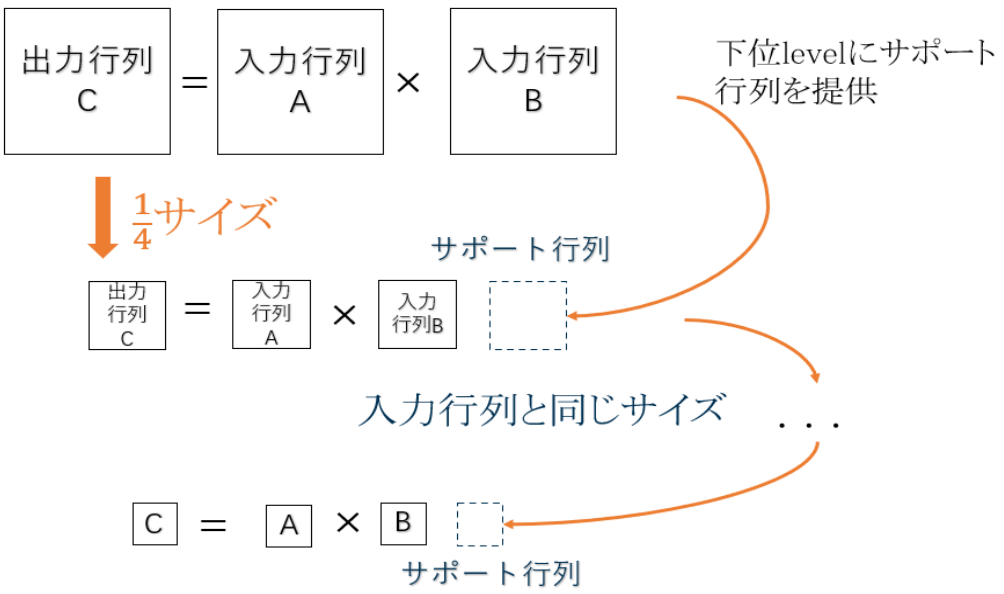


図 3.2: Strassen-RUM 法の me メモリ空間全体像のイメージ

上位 Level からもらったサポート行列で Temporary 行列をすべて削除する方法の実現には二つの条件がある：

- 存在条件：上位 Level が下位 Level に提供するサポート行列必要な再利用できるメモリ空間がある
- 唯一条件：上位 Level の現有の入力行列、出力行列のサブ行列の中に一つをサポート行列として選定し下位 Level に提供する

3.3.2 メモリ空間使用分析アルゴリズム

この節は第 3.3.1 節の存在条件を判断するために、メモリ空間使用分析アルゴリズムを提案する。

メモリ空間使用分析アルゴリズムは Strassen アルゴリズムを演算する時、各 Step のメモリ空間の使用状況を表す状態表を中心に、存在条件を解決するためのアルゴリズムである。その後、利用できるメモリ空間の中にサポート行列を選定し、唯一条件を解決できる。

存在条件の探索はまず Temporary 行列なしの大塚手法の LowLevel を入力して状態表を生成してサポート行列必要なメモリ空間があるかどうかを分析する。この場合、大塚手法の LowLevel は最初の Level として、すべての乗算に再利用できるメモリ空間があるなら、次の Level にサポート行列を提供できるので Strassen-RUM 法の考え方が応用できる、ない場合は計算順番の調整などが必要である。

メモリ空間使用分析アルゴリズムの具体的な説明が大塚手法の LowLevel を分析手順により説明する。大塚手法の LowLevel に対して、状態表の生成の手順と状態表による存在条件の判断が以下になる：

・状態表 Stb(State table)(24×12) **を生成する手順**

状態表 Stb は入力行列と出力行列を用いる 12 個のメモリ空間のライブラリを分析し、どの Step にどのメモリ空間を再利用することが全体的な演算に影響がないことを示す表である。表の大きさは 24×12。

Step1 を生成する前、初期化した状態表は図 3.3 に示す。

状態表の 12 列は入力行列と出力行列の 12 個のサブ行列に対応するメモリ空間のシリアルナンバー、これは全体的に Temporary 行列を削除した場合すべて使えるメモリ空間である。各メモリ空間の状態について、1 がメモリが使っている (再利用できない)、0 が使っていない (再利用できる) を表す。

状態表の 24 行は三つの部分に分けている。第 1 行に最初の使えるメモリ空間の状態を表す、第 2 行から 23 行が Strassen アルゴリズムの 22 個 Step に対応する各 Step の使えるメモリ空間状態である、最後の第 24 行が各メモリ空間最後使用した Step を記録する行。

最初の状態が CPU からデータを GPU に転送したの状態だから、すべての入力行列が使っている状態、出力行列が使っていないので $\{1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0\}$ に設定する。

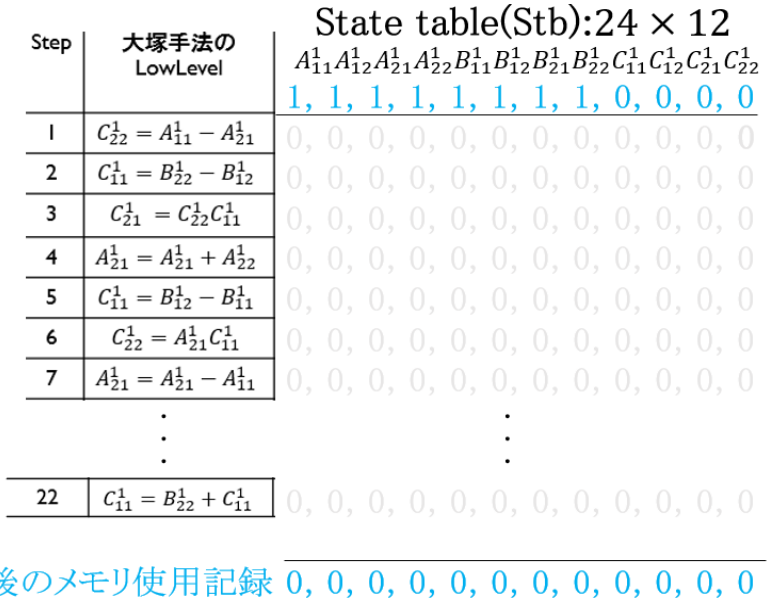


図 3.3: 状態表 Stb Step1 生成する前 (初期化) の状態

状態表残る部分の生成は Step1 から 4 までの生成を例として説明する。一行を生成する手順は以下になる：

- 1. 前の行の状態をコーピ。
- 2. 式の右辺に現れる変数に対応する状態表の変数を 1 にする、メモリ使用記録に Step 数を記入。図 3.4 からの Stb の中に生成している Step と Step 後のメモリ使用記録に対応する行の緑色の部分がこのステップ。
- 3. 式の左辺に現れる変数に対応する状態表の列に対して、メモリ使用記録の Step 数と現在の Step 数まで中間の値を 0 にする。図 3.4 からの Stb の中と Step 前のメモリ使用記録にあるの紫色の部分がこのステップ。
- 4. 式の右辺に現れる変数に対応する状態表のに変数を 1 にする、メモリ使用記録に Step 数+1 を記入。図 3.4 からの Stb の中に生成している Step と Step 後のメモリ使用記録に対応する行の赤色の部分がこのステップ。

Step1 から Step4 までの生成は以上の手順で、図 3.4 から図 3.7 まで示す。

Step1 を生成する時 (図 3.4)、まず手順 1 が初期状態をコーピし、Step1 の状態が $\{1,1,1,1,1,1,1,1,0,0,0,0\}$(青色部分) になる。手順 2 実行した後、Step1 に対応する状態表の行に式の右辺に現れる変数 A^1_{11} と A^1_{21} が 1 になった (緑色部分)、右辺の変数に対応するメモリ使用記録も Step 数、1 を記入した (緑色部分)、Step1 の状態が $\{1,1,1,1,1,1,1,1,0,0,0,0\}$、メモリ使用記録が $\{1,0,1,0,0,0,0,0,0,0,0,0\}$ に更新する。手順 3 のメモリ使用記録の Step 数が 0、現在の Step 数が 1、中間の値がないから、0 に変更する部分がない。手順 3 は実行しない。手順 4 が式の左辺に現れる変数 C^1_{22} に手順 2 の操作をする (赤色部分)、Step1 の状態が $\{1,1,1,1,1,1,1,1,0,0,0,1\}$、メモリ使用記録が $\{1,0,1,0,0,0,0,0,0,0,0,1\}$ に更新する。

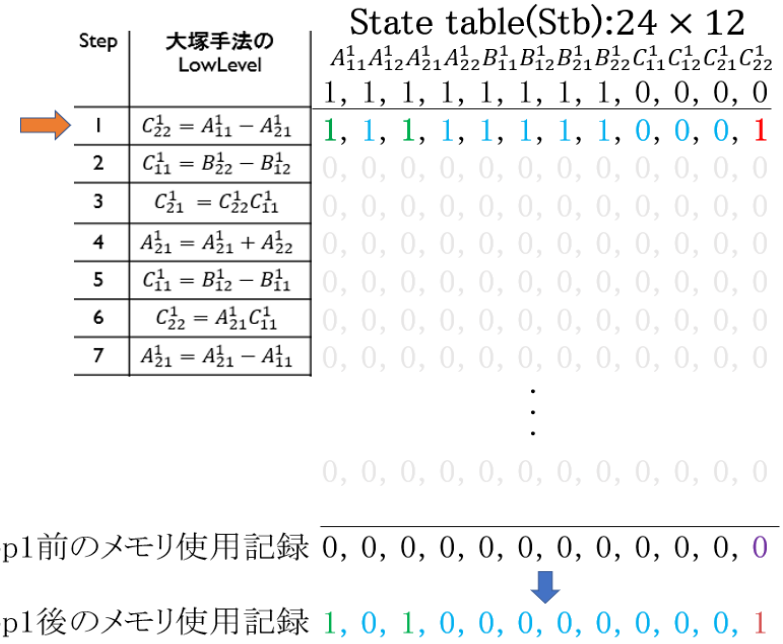


図 3.4: 状態表 Stb Step1 生成した後の状態

Step2 の場合 (図 3.5)、手順 3 のメモリ使用記録の Step 数が 0、現在の Step 数が 2、中間の値が 1 なので、Step1 に対応する C^1_{22} を 0 で上書きされた (紫色部分)。これは左辺に現れる変数が前回の使用 (Step0) から、今回の上書き (Step2) 前まで再利用できるを標記するための手順である。

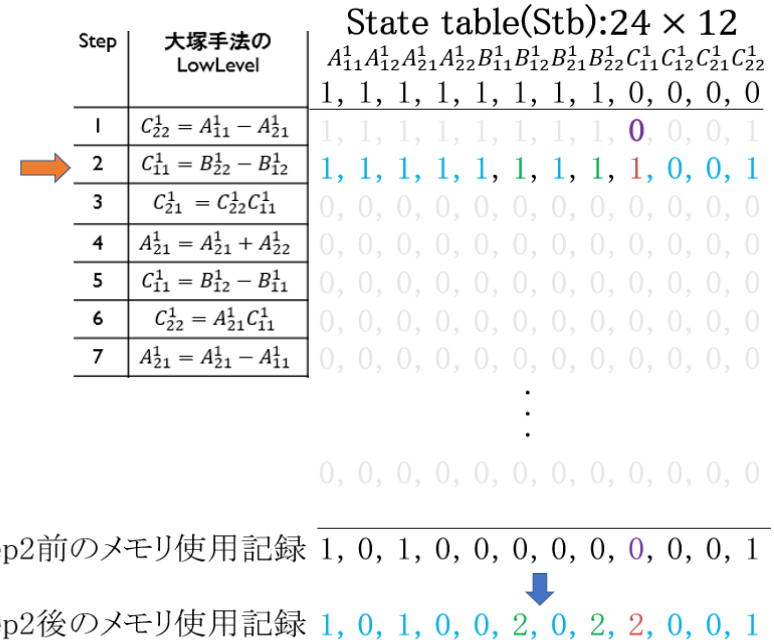


図 3.5: 状態表 Stb Step2 生成した後の状態

Step3 がサブ行列の乗算 Step である、、存在条件を判断する必要がある。図 3.6 から見れば、手順 1-4 実行した後、生成したの Step3 に対応する状態表の行に C_{12}^1 が 0 だから、この時点で、C_{12}^1 が使用されていないから、サポート行列として選定して下位 Level に提供し、上書きされても上位 Level 今後の計算に影響がない。

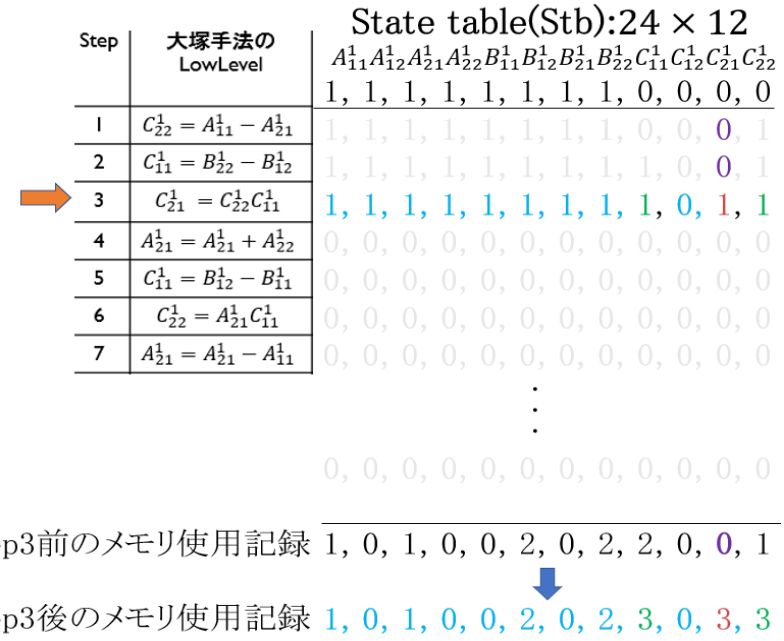


図 3.6: 状態表 Stb Step3 生成した後の状態

Step4、図 3.7 によって変数 A_{21}^1 が式の右辺と左辺両方あるから、手順 2 で右辺値のメモリ使用記録を更新した後手順 4 で左辺値を更新する必要がある。これが手順 2、4 が同じ処理だが一つの手順にならない理由である。

Step22 まで生成した後、入力行列の値が再利用できる状態、出力行列が計算結果を保持している。これを状態表に反映するため、入力行列に対して、後処理が必要である。処理が入力行列 A、B のメモリ使用記録の Step 数+1 から最後まで 0 にする。その結果が図 3.8 に示す、入力行列に 0 にした部分が黄色の部分。

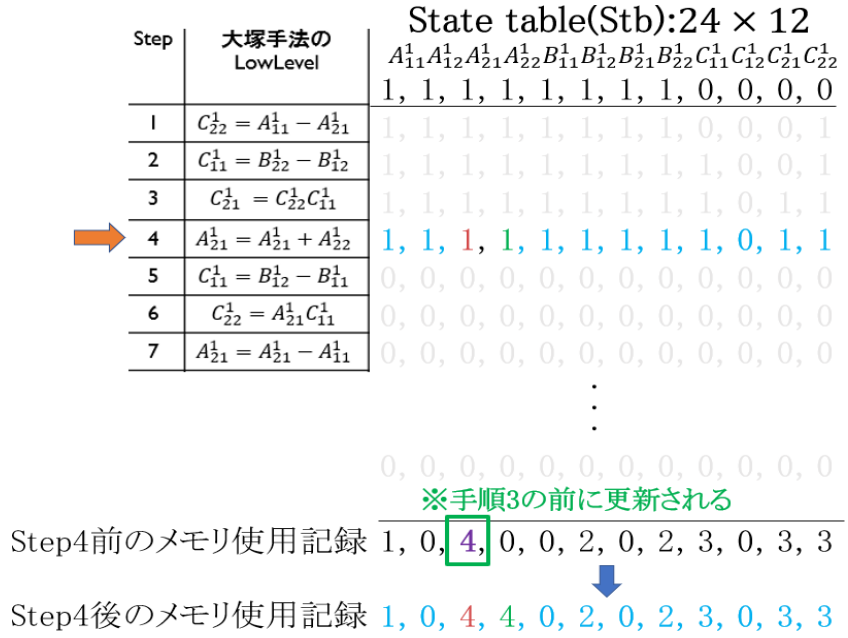


図 3.7: 状態表 Stb Step4 生成した後の状態

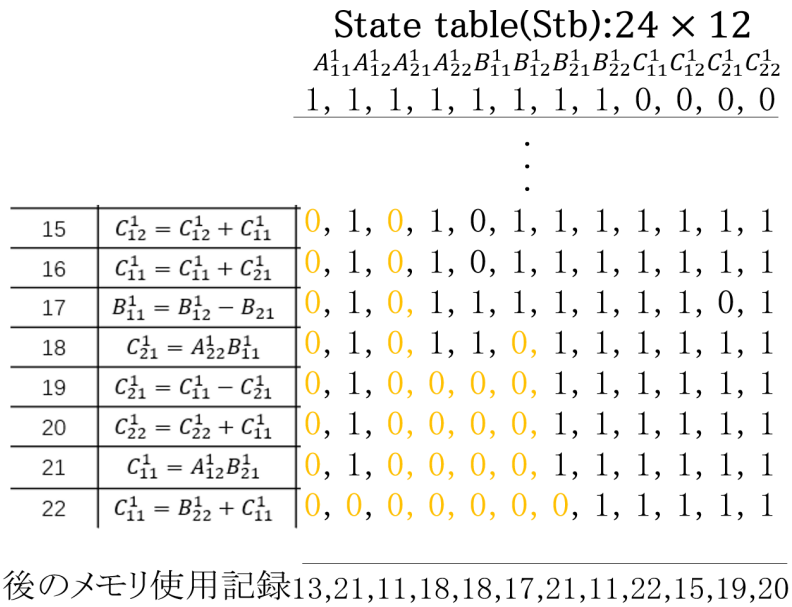


図 3.8: 状態表 Stb の後処理

・状態表による存在条件の判断

Temporary 行列なしの大塚手法 LowLevel アルゴリズムを用いて状態表の生成を実行した後、表 3.2[1]に示すように分析結果がある。

結果から見れば、Step9 と 11 に対応する状態表の行がすべて 1 なので、Stb の中

[1]Stb の中に 0 になる空間の列に加減算の部分が省略する

に 0 になる空間、異なる Level に提供する必要な再利用できるメモリ空間がない。大塚手法の LowLevel が理論上には再帰できないことが分かった。

Step	大塚手法の LowLevel	Stb の中に 0 になる空間
1	$C^1_{22} = A^1_{11} - A^1_{21}$	
2	$C^1_{11} = B^1_{22} - B^1_{12}$	
3	$C^1_{21} = C^1_{22} \times C^1_{11}$	C^1_{12}
4	$A^1_{21} = A^1_{21} + A^1_{22}$	
5	$C^1_{11} = B^1_{12} - B^1_{11}$	
6	$C^1_{22} = A^1_{21} \times C^1_{11}$	B^1_{12};C^1_{12}
7	$A^1_{21} = A^1_{21} - A^1_{11}$	
8	$B^1_{12} = B^1_{22} - C^1_{11}$	
9	$C^1_{12} = A^1_{21} \times B^1_{12}$	空間がない
10	$A^1_{21} = A^1_{12} - A^1_{21}$	
11	$C^1_{12} = A^1_{21} \times B^1_{22}$	空間がない
12	$C^1_{12} = C^1_{12} + C^1_{22}$	
13	$B^1_{22} = A^1_{11} \times B^1_{11}$	A^1_{21}
14	$C^1_{11} = C^1_{11} + B^1_{22}$	
15	$C^1_{12} = C^1_{12} + C^1_{11}$	
16	$C^1_{11} = C^1_{11} + C^1_{21}$	
17	$B^1_{11} = B^1_{12} - B^1_{21}$	
18	$C^1_{21} = A^1_{22} \times B^1_{11}$	A^1_{11};A^1_{21};B^1_{12}
19	$C^1_{21} = C^1_{11} - C^1_{21}$	
20	$C^1_{22} = C^1_{22} + C^1_{11}$	
21	$C^1_{11} = A^1_{12} \times B^1_{21}$	A^1_{11};A^1_{21};A^1_{22};B^1_{11};B^1_{12}
22	$C^1_{11} = B^1_{22} + C^2_{11}$	

表 3.2: Temporary 行列なしの大塚手法 LowLevel の再利用できるメモリ空間の分析結果

3.3.3 Strassen-RUM 法の Level-1

大塚手法の LowLevel の Step9 と 11 が再利用できるメモリ空間がないため、計算順番の調整などが必要である。

大塚手法の LowLevel の同 level のメモリ再利用を分析すると、図 3.9-a のように、大塚手法の LowLevel は A_{21}、B_{12}、B_{22}、B_{11}4 つのサブ行列を再利用した。しかし、この 4 つのサブ行列のライフタイム最大 3 つが重ねた (Step17)、つまり同時に再利用として使用したメモリ空間が最大三つだけあるから、一つ減らすことができる。

一方、計算順番の調整、特に出力行列、入力行列のメモリ空間を再利用する場合、Step 間の依頼関係を配慮する必要がある。そこで、大塚手法の LowLevel の Step13 が A_{11}^2 と B_{11}^2 二つ入力行列のサブ行列を使った、B_{22}^2 が一つ同 Level のメモリ再利用の空間である。Step13 には、式の右辺と左辺が Step1 から 12 までとは依頼関係がないことが分かった。そして、Step14 から、式の右辺にある二つのサブ行列が使っていないから、再利用できるメモリ空間になることが分かった。左辺が一つメモリ空間を再利用、右辺が二つメモリ空間を解放、Step13 により再利用できるメモリ空間が一つ増えることができる。

同 Level のメモリ使用量を減らす方法として、大塚手法の LowLevel の Step13 が Step8 の前に調整して、同 Level の再利用空間を新たに分配すると、同 Level メモリの再利用が三つあるアルゴリズムが図 3.9-b に示す。この計算順番を調整した Strassen アルゴリズムが **Strassen-RUM 法の Level-1** と定義する。

(a) 大塚手法の同 Level のメモリ再利用の分析　(b) Strassen-RUM 法の Level-1 同 Level のメモリ再利用の分析

図 3.9: 同 Level のメモリ空間再利用数の比較

Strassen-RUM 法の Level-1 で状態表を生成する結果を表 3.3 の左三列に示す、すべての乗算に少なくとも 1 つの再利用できるメモリ空間がある。存在条件を達成した。

唯一条件を達成するため、最短距離アルゴリズム、LRU アルゴリズムなど選択

法を利用して再利用できるメモリ空間 (Stb の中に 0 になる空間) からサポート行列を選択する実験をしたが、複数再利用できるメモリ空間にどれを選ぶことは演算速度に 1%以下の影響しかない。そのため、演算をできるだけ簡単するため、一つ再利用できるメモリ空間がある乗算がそのメモリ空間をサポート行列として選ぶ、複数の再利用できるメモリ空間がある乗算に、状態表の列の順番で最初 0 になる空間をサポート行列として選ぶ。

サポート行列を含める Strassen-RUM 法の Level-1 が表 3.3 に示す。

Step	Strassen-RUM 法の Level-1	Stb の中に 0 になる空間	サポート行列
1	$C^1_{11} = A^1_{11} - A^1_{21}$		
2	$C^1_{12} = B^1_{22} - B^1_{12}$		
3	$C^1_{21} = C^1_{11} \times C^1_{12}$	C^1_{22}	C^1_{22}
4	$C^1_{12} = A^1_{21} + A^1_{22}$		
5	$B^1_{12} = B^1_{12} - B^1_{11}$		
6	$C^1_{22} = C^1_{12} \times B^1_{12}$	A^1_{21};C^1_{11}	A^1_{21}
7	$C^1_{12} = C^1_{12} - A^1_{11}$		
8	$A^1_{21} = A^1_{11} \times B^1_{11}$	C^1_{11}	C^1_{11}
9	$B^1_{12} = B^1_{22} - B^1_{12}$		
10	$C^1_{11} = C^1_{12} \times B^1_{12}$	A^1_{11};B^1_{11}	A^1_{11}
11	$A^1_{11} = A^1_{12} - C^1_{12}$		
12	$C^1_{12} = A^1_{11} \times B^1_{22}$	B^1_{11}	B^1_{11}
13	$C^1_{12} = C^1_{12} + C^1_{22}$		
14	$C^1_{11} = C^1_{11} + A^1_{21}$		
15	$C^1_{12} = C^1_{12} + C^1_{11}$		
16	$C^1_{11} = C^1_{11} + C^1_{21}$		
17	$B^1_{12} = B^1_{12} - B^1_{21}$		
18	$C^1_{21} = A^1_{22} \times B^1_{12}$	A^1_{11};B^1_{11};B^1_{22}	A^1_{11}
19	$C^1_{21} = C^1_{11} - C^1_{21}$		
20	$C^1_{22} = C^1_{22} + C^1_{11}$		
21	$C^1_{11} = A^1_{12} \times B^1_{21}$	A^1_{11};A^1_{22};B^1_{11};B^1_{12};B^1_{22}	A^1_{11}
22	$C^1_{11} = A^1_{21} + C^1_{11}$		

表 3.3: Strassen-RUM 法の Level-1 とそのメモリ再利用の分析

青い部分が入力行列のメモリ再利用の空間

3.3.4 Strassen-RUM 法の Level-2~N へ拡張

Strassen-RUM 法の Level-1 の乗算が Level-2 を生成する時、Level-1 から提供されたサポート行列が Level-2 の S^2_{11}、S^2_{12}、S^2_{21}、S^2_{22}4 つのメモリ空間になる。

Strassen-RUM 法の Level-1 のアルゴリズムを再帰できるため、再帰で生成した Level-2 の入力行列を上書きすることを回避する必要がある。Level-2 に、サポート行列のサブ行列の空間を level-1 の入力行列の中に上書きされた部分に対応する A^2_{11}、A^2_{21}、B^2_{11}、B^2_{12} に引き換えれて、Level-2 の上書きされる部分がサポート行列のサブ行列に変更できる。存在条件により、再利用できるメモリ空間の中に選ばれたサポート行列の保有値が後の Step に使わない値なので、変化しても Level-1 に影響がない。

そして、Level-1 のサポート行列のサブ行列が更に Level-2 のサポート行列として選び、Level-3 に提供できる。Strassen-RUM 法が再帰できるアルゴリズムになる。

Strassen-RUM 法の Level-2~N は表 3.4 に示す。

Step	Strassen-RUM 法の Level-1	Stb の中に 0 になる空間	サポート行列
1	$C^n_{11} = A^n_{11} - A^n_{21}$		
2	$C^n_{12} = B^n_{22} - B^n_{12}$		
3	$C^n_{21} = C^n_{11} \times C^n_{12}$	C^n_{22}	C^n_{22}
4	$C^n_{12} = A^n_{21} + A^n_{22}$		
5	$S^n_{22} = B^n_{12} - B^n_{11}$		
6	$C^n_{22} = C^n_{12} \times S^n_{22}$	S^n_{12} ;C^n_{11}	S^n_{12}
7	$C^n_{12} = C^n_{12} - A^n_{11}$		
8	$S^n_{12} = A^n_{11} \times B^n_{11}$	C^n_{11}	C^n_{11}
9	$S^n_{22} = B^n_{22} - S^n_{22}$		
10	$C^n_{11} = C^n_{12} \times S^n_{22}$	S^n_{11};S^n_{21}	S^n_{11}
11	$S^n_{11} = A^n_{12} - C^n_{12}$		
12	$C^n_{12} = S^n_{11} \times B^n_{22}$	S^n_{21}	S^n_{21}
13	$C^n_{12} = C^n_{12} + C^n_{22}$		
14	$C^n_{11} = C^n_{11} + S^n_{12}$		
15	$C^n_{12} = C^n_{12} + C^n_{11}$		
16	$C^n_{11} = C^n_{11} + C^n_{21}$		
17	$S^n_{22} = S^n_{22} - B^n_{21}$		
18	$C^n_{21} = A^n_{22} \times S^n_{22}$	S^n_{11};S^n_{21} ;B^n_{22}	S^n_{11}
19	$C^n_{21} = C^n_{11} - C^n_{21}$		
20	$C^n_{22} = C^n_{22} + C^n_{11}$		
21	$C^n_{11} = A^n_{12} \times B^n_{21}$	S^n_{11};A^n_{22};S^n_{21};S^n_{22};B^n_{22}	S^n_{11}
22	$C^n_{11} = S^n_{12} + C^n_{11}$		

表 3.4: サポート行列付き Strassen-RUM 法の Level-2~N

緑色の部分が上書きを引き換えた部分

Strassen-RUM 法は大塚手法と同じ入力行列の上書きがあるが、大塚手法 LowLevel の再帰問題を解決し、7 つ Temporary 行列ある HighLevel だけ再帰できることを

回避できる。Strassen-RUM 法を作成することにより、HighLevel と LowLevel 両方の Temporary 行列を削除することができ、メモリ使用量を減らした。

3.3.5 Strassen-RUM 法の同一化

Strassen-RUM 法の Level-1 と Level-2~N のアルゴリズムから見れば、上書きされる部分だけ区別があるから、Level の同一化ができる。

同一化するため、Level-1 が Level-2 のように出力行列、入力行列以外 4 つサブ行列のメモリ空間が必要である。Level-1 に 4 つ偽サポート行列のサブ行列を作る。4 つの偽サポート行列のサブ行列は S_{11}^1 と A_{11}^1、S_{12}^1 と A_{21}^1、S_{21}^1 と B_{11}^1、S_{22}^1 と B_{12}^1 同じメモリ空間を使う。つまり、Level-1 の A_{11}^1、A_{21}^1、B_{11}^1、B_{12}^1 に別名を付ける。このことにより、Strassen-RUM 法の Level-1 のサポート行列付き演算スケジュールも表 3.4 で表示することができる、すべての Level を同一化した。

同一化したアルゴリズムはアルゴリズム 3.1 に示す。この後の議論には Level-1 と Level-2~N を明示しない。

アルゴリズム 3.1 Strassen-RUM アルゴリズム

1: $C_{11}^n = A_{11}^n - A_{21}^n$
2: $C_{12}^n = B_{22}^n - B_{12}^n$
3: $C_{21}^n = C_{11}^n \times C_{12}^n$
4: $C_{12}^n = A_{21}^n + A_{22}^n$
5: $S_{22}^n = B_{12}^n - B_{11}^n$
6: $C_{22}^n = C_{12}^n \times S_{22}^n$
7: $C_{12}^n = C_{12}^n - A_{11}^n$
8: $S_{12}^n = A_{11}^n \times B_{11}^n$
9: $S_{22}^n = B_{22}^n - S_{22}^n$
10: $C_{11}^n = C_{12}^n \times S_{22}^n$
11: $S_{11}^n = A_{12}^n - C_{12}^n$
12: $C_{12}^n = S_{11}^n \times B_{22}^n$
13: $C_{12}^n = C_{12}^n + C_{22}^n$
14: $C_{11}^n = C_{11}^n + S_{12}^n$
15: $C_{12}^n = C_{12}^n + C_{11}^n$
16: $C_{11}^n = C_{11}^n + C_{21}^n$
17: $S_{22}^n = S_{22}^n - B_{21}^n$
18: $C_{21}^n = A_{22}^n \times S_{22}^n$
19: $C_{21}^n = C_{11}^n - C_{21}^n$
20: $C_{22}^n = C_{22}^n + C_{11}^n$
21: $C_{11}^n = A_{12}^n \times B_{21}^n$
22: $C_{11}^n = S_{12}^n + C_{11}^n$

3.4 Strassen-RUM 法の評価実験

3.4.1 GPU での実験環境と条件

表 3.5、表 3.6、表 3.7 は今回の実験環境を示す。大規模の行列乗算に対応するため、GPU は TensorCore が倍精度も計算可能浮動小数点演算が高い、メモリ容量とメモリバンド幅も大きいの Tesla A100 [3]、倍精度計算ができない TensorCore が持っている Tesla V100 [17] と TensorCore がなし、メモリ容量と計算性能が相対的に弱い Tesla P100 [18] を用いた。実験が A100 を中心に、V100 と P100 の結果を対比に結論をつける。

表 3.5: GPU での実験環境-A100

Host	SuperA100
CPU	AMD EPYC 7302 ×2
Memory	DDR4 3200MHz ×16: 256GB
GPU	NVIDIA A100 80GB PCIe
OS	Ubuntu 18.04
CUDA	version 11.7
Compiler Optimization Flags	-O3 -gencode=arch=compute_80,code=sm_80
Library	CUBLAS

表 3.6: GPU での実験環境-V100

Host	V100
CPU	Inter(R)Core(TM)i7-6700k
Memory	DDR4 3200MHz 64GB
GPU	NVIDIA V100 32GB PCIe
OS	Ubuntu 18.04
CUDA	version 11.7
Compiler Optimization Flags	-O3 -gencode=arch=compute_70,code=sm_70
Library	CUBLAS

表 3.7: GPU での実験環境-P100

Host	P100
CPU	Inter(R)Xeon(R)E5-2637 v4
Memory	DDR4 3200MHz 64GB
GPU	NVIDIA P100 16GB PCIe
OS	Ubuntu 18.04
CUDA	version 11.7
Compiler Optimization Flags	-O3 -gencode=arch=compute_60,code=sm_60
Library	CUBLAS

実行時間の測定について、GPU の処理時間の測定は CUDA C/C++の cudaEventRecord() 関数を用いる。測定範囲がプログラムの GPU 上の実行時間のみを測定し、データ転送の時間は含まない。

メモリ使用量の測定は NVIDIA のプログラム分析ソフトウェア Nsight Systems を用いて Memory useage という測定値で実測を行う、GPU のグローバルメモリを使う最大値を測定。

実験の行列が A100 の演算能力、特に第三世代 TensorCore が得意な倍精度の密行列の乗算を狙って乱数で生成する。行列のサイズが最低 Level-4 まで分割可能な偶数の正方行列。第 2.3.1 節に紹介したように奇数と長方形行列の計算も可能だが、今回の実験には対象外とする。行列のサイズについて、A100 が 512 から 40,960 サイズまで、V100 は 512 から 24,576 サイズ、P100 は 512 から 16,384 サイズを選定する[2]。

実験の流れと数値の測定について、CPU の入力行列生成、GPU へデータ転送、行列演算、CPU へ出力行列を転送することを実験の 1 回で、すべてのデータが 10 回の平均値を採用する。10 回のデータに平均値の 95%-105%以外のデータがあれば、結果を再測定する。メモリ使用量が変わらないため、一回の測定で最大メモリ使用量を測定する。

この節も第 4 章、第 5 章の実験の環境と条件である。

[2]GPU のグローバルメモリが限界まで使うと、演算時間に影響があるから、今回の実験が最大 80%ぐらいの利用率、40,960 サイズ (A100)、24,576 サイズ (V100)、16,384 サイズ (P100) まで。

3.4.2 Strassen-RUM 法の評価実験

Strassen-RUM 法の評価	
測定目的	メモリ使用量を比較し、節約率計算する； 演算速度が落ちるかどうかを評価する
対象	提案手法、大塚手法 (逐次)、Lai らの方法
行列サイズ-A100	8,192、32,768
行列サイズ-V100	8,192、2,048
行列サイズ-P100	8,192、12,288

表 3.8: Strassen-RUM 法の実験の内容、サイズと目的

行列サイズについて、8,192 サイズ[3]が GPU 間の対比するため、同じサイズに設定する。他のサイズが小さいサイズから、大きいサイズまでの範囲を示すために設定したサイズである。

3.4.3 メモリ節約率の比較

測定したデータを表にした A100 の結果が表 3.9 になる。節約したメモリの比率が以下の式で計算した：

$$\text{節約したメモリの比率} = 1 - \frac{Strassen - RUM \text{ 法のメモリ使用量}}{\text{先行研究のメモリ使用量}} \tag{3.3}$$

行列サイズ	Level	Strassen-RUM 法	大塚手法		Lai らの方法	
		メモリ	メモリ	節約したメモリの比率/%	メモリ	節約したメモリの比率/%
8,192	Level-1	1.51	1.51	0.00	1.76	14.20
	Level-2	1.51	2.38	36.55	1.82	17.03
/GiB	Level-3	1.51	2.60	41.92	1.84	17.93
	Level-4	1.51	2.65	43.02	1.84	17.93
32,768	Level-1	24.01	24.01	0.00	28.01	14.28
	Level-2	24.01	38.01	36.83	29.01	17.24
/GiB	Level-3	24.01	41.51	42.16	29.26	17.94
	Level-4	24.01	42.38	43.35	29.32	18.11

表 3.9: 先行研究と比べて Strassen-RUM 法の Level-1~4 のメモリ節約率-A100

A100 の実験結果から見れば、すべての Temporary 行列を削除したので、Strassen-RUM 法のメモリ使用量は多 Level でも増加しない。大塚手法が HighLevel に 7 つ

[3]入力、出力行列のサイズが $8,192 \times 8,192$ を示す、他の実験内容のサイズの説明も同じ。

Temporary 行列、LowLevel に Temporary 行列なしの方法だから、Level-1 の時は Strassen-RUM 法と同じで、多 Level にすると、メモリ使用量が急速に増加した。Lai らの方法は各 Level 2 つの Temporary 行列だから、Level-1 の時は Strassen-RUM 法と大塚手法より多いメモリを使用した、多 Level の時も増加があるが、大塚手法よりは少ない。

メモリ使用量を減らすことを目指す Lai らの方法と比べて、Level-1 の時は 14%ぐらいのメモリを節約でき、最大 18.11%のメモリを節約できる。大塚手法の提案 Level と同じ Level-2 の場合、高速化を目指す大塚手法の 36%ぐらいのメモリを節約することができる、Level-4 の場合は最大 43.35%のメモリを節約できる。この二つの最大メモリ節約率は第 3.3.1 節の無限 Level で計算した理論値 18.18%(Lai らの方法)、43.75%(大塚手法) と合っていることが確認できる。

V100 の結果が表 3.10 に示す。

行列サイズ	Level	Strassen-RUM 法	大塚手法		Lai らの方法	
		メモリ	メモリ	節約したメモリの比率/%	メモリ	節約したメモリの比率/%
8,192	Level-1	1.51	1.51	0.00	1.76	14.20
	Level-2	1.51	2.38	36.55	1.82	17.03
/GiB	Level-3	1.51	2.60	41.92	1.84	17.93
	Level-4	1.51	2.66	43.23	1.84	17.93
2,048	Level-1	104.13	104.13	0.00	120.13	13.32
	Level-2	104.13	160.13	34.97	124.13	16.11
/MiB	Level-3	104.13	174.13	40.19	125.13	16.78
	Level-4	104.13	177.63	42.09	125.38	17.03

表 3.10: 先行研究と比べて Strassen-RUM 法の Level-1~4 のメモリ節約率-V100

P100 の結果が表 3.11 に示す。

行列サイズ	Level	Strassen-RUM 法	大塚手法		Lai らの方法	
		メモリ /GiB	メモリ /GiB	節約したメモリの比率/%	メモリ /GiB	節約したメモリの比率/%
8,192	Level-1	1.51	1.51	0.00	1.76	14.20
	Level-2	1.51	2.38	36.55	1.82	17.03
/GiB	Level-3	1.51	2.60	41.92	1.84	17.93
	Level-4	1.51	2.66	43.23	1.84	17.93
12,288	Level-1	3.38	3.38	0.00	3.95	14.43
	Level-2	3.38	5.35	36.82	4.09	17.36
/GiB	Level-3	3.38	5.84	42.12	4.12	17.96
	Level-4	3.38	5.97	43.38	4.13	18.16

表 3.11: 先行研究と比べて Strassen-RUM 法の Level-1~4 のメモリ節約率-P100

すべて結果から見れば、GPU と関係ない、同 Level の Strassen-RUM 法が同じ

程度のメモリを節約できる。そして、小さいサイズに対して、メモリ節約率が相対的に低いことが分かった。

データの中に小数部が同じ数字がある原因はGPUのカーネル実行する時の固定的な構造と思う。

3.4.4 演算速度の比較

メモリ使用量上の結果から見れば、Strassen-RUM法のメモリ使用量がLevel数と関係なく変化しない。この節はLevelにより変化した実行速度を図で分析する、Strassen-RUM法の速度が落ちるかどうかを評価する。メモリ使用量の変化を明らかにするため、表3.9から表3.11の結果を図に加えた。

A100でのStrassen-RUM法、大塚手法、Laiらの方法三つの演算の異なるLevelの演算速度の変化とメモリ使用量の変化が図3.10(8,192サイズ)と図3.11(32,768サイズ)に示す。

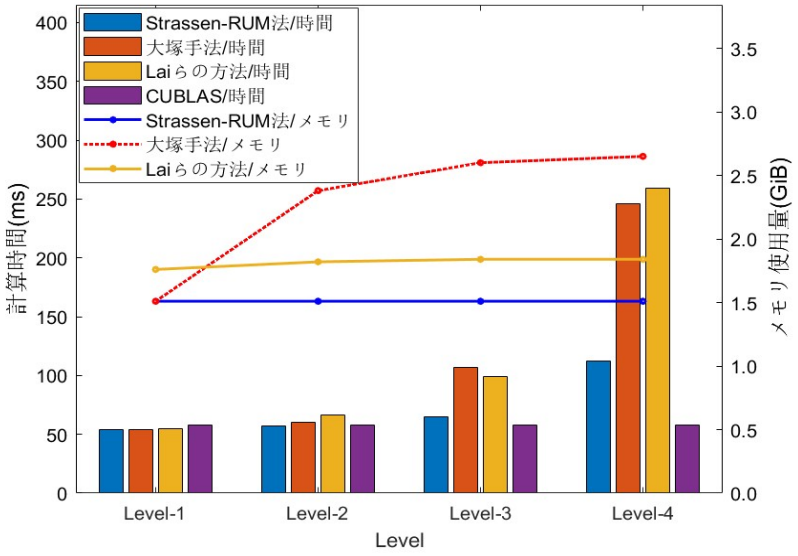


図 3.10: 8,192サイズ各Strassen法Level-1~4の時間、メモリ使用量-A100

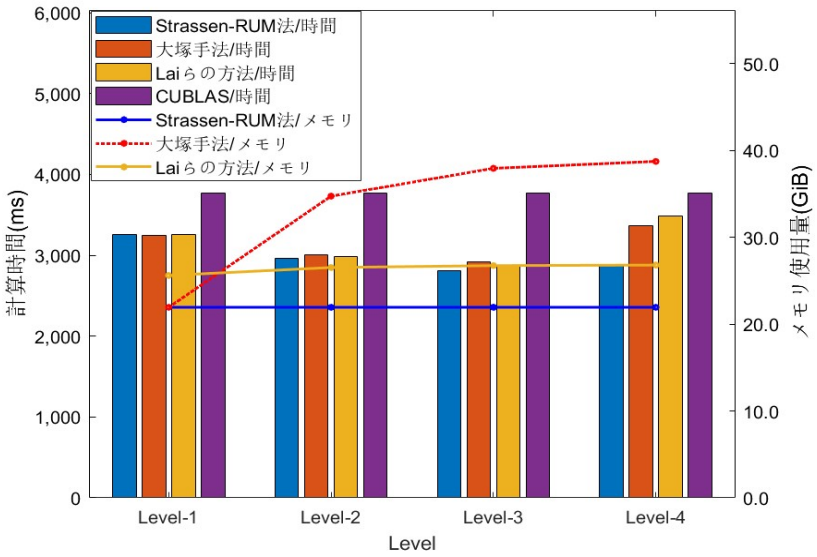


図 3.11: 32,768 サイズ各 Strassen 法 Level-1˜4 の時間、メモリ使用量-A100

A100 の結果の速度面から見れば、目的ではないだが、Strassen-RUM 法の演算速度のメリットもある。Strassen-RUM 法の演算時間が Level-1 の時、特に利点がないが、Level の増加とともに、優位性が見られる。8,192 サイズの Level-1 が演算時間一番短い Level である、Level-3、Level-4 に先行研究の 1/2 以下の時間で演算を完成できる。実用性から見れば、32,768 サイズの Level-3 が演算時間一番短い Level である、この時点で大塚手法より 103.904%、Lai らの方法より 102.267%高速化になる。

高速化になった原因として、大塚手法の高速化の原理は同じである。先行研究の図 2.9 で示したように、CUDA を実行する時、Temporary 行列の確保と解放する関数の時間がかなりかかる。大塚手法が Temporary 行列処理時間が全体の計算時間の割合が高い LowLevel の Temporary 行列を削除したが HighLevel には削除していない。大塚手法と比べて、Strassen-RUM 法がすべての Temporary 行列を削除したので、処理時間の割合が低い HighLevel の時間も削減したから、特に HighLevel が多い Level-3 以上の場合、速度のメリットがある。

補充として、大塚手法の逐次手法だけ見れば、7 つの Temporary 行列使った HighLevel の影響で小さいサイズ、Level-2 までは高速化になるが、大きいサイズ或は Level-3 以上になる、高速にはならない。これが当時の GPU アーキテクチャと CUBLAS バージョンによる認識の制限だと思う。

更に高速化の改良が一つの拡張問題になる、高速化の方法である Strassen-RUM 法の並列化の考えは第 5 章の並列化部分で紹介している。

V100 の実験結果が図 3.12(8,192 サイズ)、図 3.13(2,048 サイズ) に示す。

V100 の 2,048 サイズの結果から見れば、一定のサイズ以下の場合、Level-1 か

らLevel-4まではすべてCUBLAS演算より遅い場合もある、この時Strassenアルゴリズムの実用性がない、これはStrassenアルゴリズムアルゴリズムの特性である。8,192サイズと2,048サイズが同じ、Level-1が演算時間一番短いLevelである。8,192サイズにStrassen-RUM法の速度が大塚手法より100.425%、Laiらの方法より100.763%になる、ほぼ差がない。

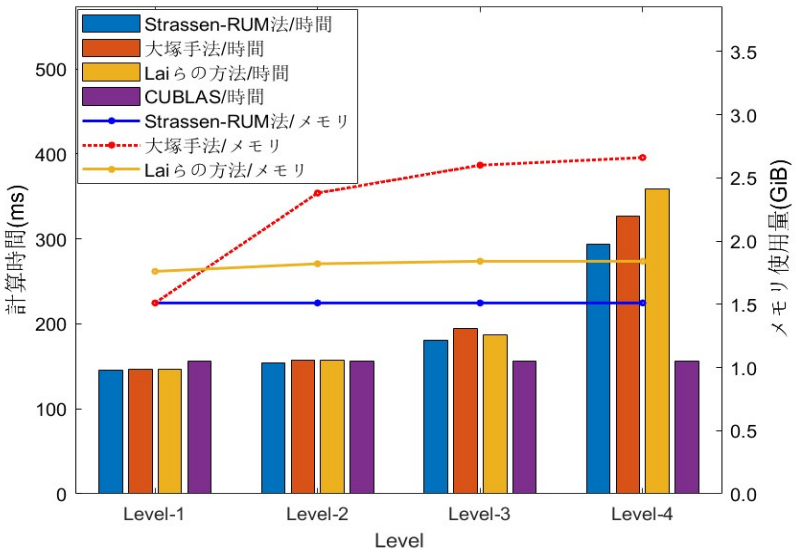


図 3.12: 8,192サイズ各Strassen法Level-1~4の時間、メモリ使用量-V100

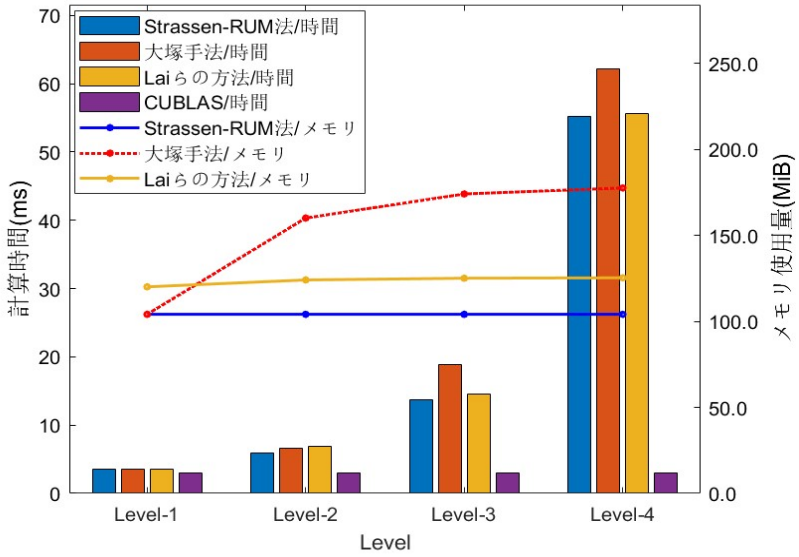


図 3.13: 2,048サイズ各Strassen法Level-1~4の時間、メモリ使用量-V100

P100 の実験結果が図 3.14(8,192 サイズ) 図 3.15(12,288 サイズ) に示す、8,192 サイズが Level-1、12,288 サイズが Level-2 が演算時間一番短い Level である。12,288 サイズに Strassen-RUM 法の速度が大塚手法より 104.817%、Lai らの方法より 104.257% に高速化になる。

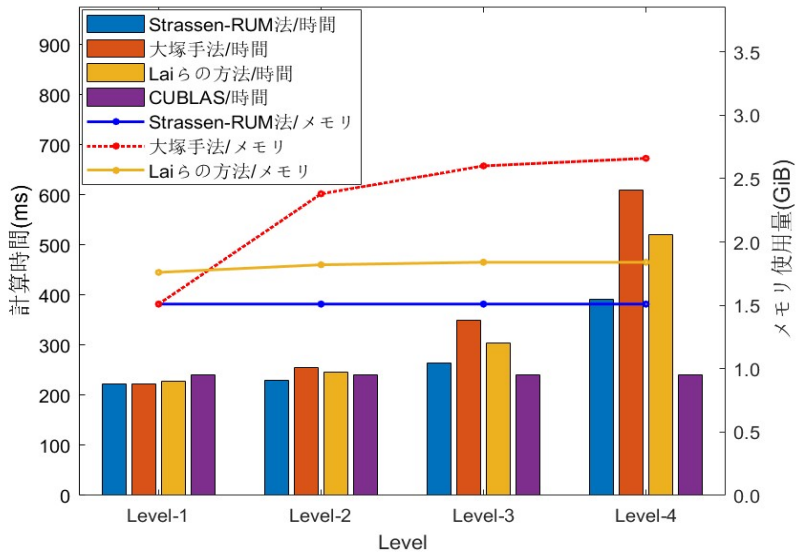


図 3.14: 8,192 サイズ各 Strassen 法 Level-1~4 の時間、メモリ使用量-P100

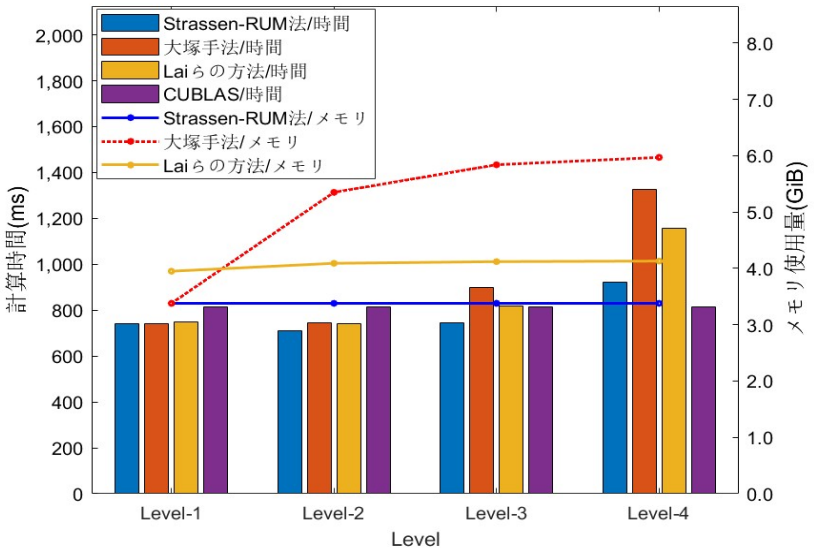


図 3.15: 12,288 サイズ各 Strassen 法 Level-1~4 の時間、メモリ使用量-P100

図 3.10 から図 3.15 までの 6 つの図から見れば、一番実行時間が短い Level の選

択は行列乗算のサイズと GPU に関係がある、この問題、最適化 Level の問題は第 4 章で討論している。

Strassen アルゴリズムの高速化の方法として、Temporary 行列を演算する前にグローバルプールに分配するやり方も Temporary 行列を削除した Strassen-RUM 法と同じ、すべての Temporary 行列の処理時間を削減できる。しかし、本研究の主要な目的が Strassen アルゴリズムを GPU へ実装する時、グローバルメモリ使用量の削減だから、Temporary 行列使用したメモリ空間を削除できないグローバルプールに分配するやり方が本研究中には採用していない。

3.5 終わりに

この節は大塚手法と Lai らの方法と比べて、大塚手法の LowLevel が再帰できない問題による膨大なメモリ使用量の問題を発見した。問題を解決するため、大塚手法の LowLevel の同 Level メモリ空間の再利用を基にして上位 Level から下位 Level に再利用できるメモリ空間をサポート行列として提供する方法を提案した。メモリ空間使用分析アルゴリズムは Strassen-RUM 法の実行を確保するために、アルゴリズムの再利用できるメモリ空間を検察する方法である。再利用できるメモリ空間を確保するため、大塚手法の LowLevel の 4 つ同 Level に再利用したメモリ空間を三つまで減らす、Strassen-RUM 法の Level-1 を作成した。その後、Level-2~N へ拡張と Strassen-RUM 法の同一化により、Strassen-RUM 法の演算スケジュールを作成した。

実験の結果、Strassen-RUM 法と先行研究の逐次演算の比較の結果、先行研究の Lai らの方法と大塚手法より、A100 でメモリ使用量が Level-4 の時それぞれ最大 18.11%と 43.35%を節約することが分かった。Temporary 行列すべて削除により、Strassen-RUM 法の演算速度も上がっていた、(A100)32,768 サイズに対して最適な Level-3 が大塚手法より 3.904%、Lai らの方法より 2.267%上がることが確認した。本章の実験、第 3.4.4 節の結論により、最適化 Level の問題と更に高速化への探索問題が発見した。本章の結果から、第 4 章は Strassen-RUM 法の最適化 Level の問題について分析する、特定な GPU 環境に対して、入力行列のサイズにより、実行時間が一番短い再帰の深度の判断を目的にする。

メモリ空間使用分析アルゴリズムが演算中メモリ空間のライフタイムを分析するアルゴリズムである、今回はサポート行列を探すことに利用したが、他のメモリ空間が演算中変化があるアルゴリズム、或いは資源占有分析に活用できることを期待している。

第4章　再帰による最適化Level問題

4.1　はじめに

第 3.4.4 節の実験中発見した、第 3 章の提案手法 Starssen-RUM 法の異なる GPU の異なる行列サイズに対して、計算が一番速いの Level が異なる問題 (最適化 Level の問題) がある。本章はこの問題の判断方法について紹介する。最初は最適化 Level の問題は何かの問題について説明する。そして、先行研究を基にして最適化 Level の判断法を提案する。

4.2　最適化問題の分析

4.2.1　最適化 Level 問題の概要

先行研究大塚の研究 [15] によると、GPU Tesla V100 16GB のグローバルメモリの制限により、最大実行可能な 15,000 ぐらいのサイズに対する Level-2 以上は適応されないから、大塚手法が 2 Level Strassen アルゴリズムを決定した。

しかし、GPU の発展により、新世代の A100 がより速い演算速度を持って、グローバルメモリも V100 の 16GB、32GB から、40GB、80GB にアップデートした。現時点最新版の NVIDIA GPU H100 には A100 と比べて演算速度が三倍になるが、グローバルメモリが A100 と同じ 80GB である。V100 と A100 のグローバルメモリが最大 5 倍の差があり、演算可能な最大サイズの一番速い Level が Level-2 ではない。A100 と H100 のグローバルメモリが同じ、演算速度の差もある、一番速い Level が違う可能性もある。

GPU の差別を把握して、入力行列のサイズに応じて提案手法 Strassen-RUM 法が一番速い Level を選べるのが最適化 Level 問題である。

4.2.2　最適化問題の本質

第 2.3.1 節によると、Strassen-RUM 法も含む現有の GPU 向け Strassen アルゴリズムの分割ではなく、通常の行列積演算を行う部分は LowLevel の乗算だけなので、演算量が LowLevel に集中している。この通常の行列積演算が普通、高性能演算ライブラリ CUBLAS のインターフェースを使っている。

演算量がLowLevelに集中していることは、LowLevelがすべてのLevel中Strassenアルゴリズムの実行速度に一番影響があるLevelである。入力行列が一定になると、最適化 Level の問題の本質が HighLevel がどこまで分割すれば分割した LowLevel の数と演算速度のバランスの影響によって全体の演算速度が一番速いの問題になる。つまり、CUBLAS の乗算関数を使う LowLevel がどのぐらいになれば、一番速くなる問題になる。

4.3 提案 2: 最適化 Level の推定法

提案 2 が Strassen-RUM 法の異なるサイズの実行速度一番速い Level を判定するの提案である。

4.3.1 GPU 特徴を配慮した最適化 Level の推定方法

今まで GPU 上の先行研究により、異なる手法の Strassen アルゴリズムが同じ、サイズが小さい時、CUBLAS 乗算より遅くなり、サイズの増加とともに、CUBLAS 乗算より速くなる。どこまで分割すれば LowLevel の演算が一番速いの問題はこの性質に関係がある。

図 4.1 に、CUBLAS 乗算と Strassen アルゴリズムの演算時間比により最適化 Level の推定方法を紹介する。

同時点 P は CUBLAS 乗算時間と Strassen アルゴリズム乗算時間ほぼ同じになる行列のサイズである。このサイズ以上の場合、Strassen アルゴリズムの方が CUBLAS 乗算より速いので、Strassen アルゴリズムを採用して、行列を 4 分割で計算する。

例えば、入力行列が 16,384 の場合、Strassen アルゴリズムを利用すれば、LowLevel で CUBLAS 乗算のサイズが 8,192 サイズだが、8,192 サイズも P 点の右側になるので、Strassen アルゴリズムをもう一度使うほうがより速くなる。そして、3 回の分割後、サイズが P 点の左側に落ちた、CUBALS 乗算が Strassen アルゴリズムより速くなるので、再分割しない。最適化 Level が Level-3 と判断できる。

Level-1、2、3 が最適な Level として判定する範囲が 0 から 2P、2P から 4P、4P から 8P。Level 間の二倍の**倍数関係**がある。各 Level の範囲間の同時点 P の倍数 (2P、4P、8P) に対応するサイズが Level 間の**境界線**と定義する。

Level数	CUBLAS演算サイズ
入力	16384
1	8192
2	4096
3	2048

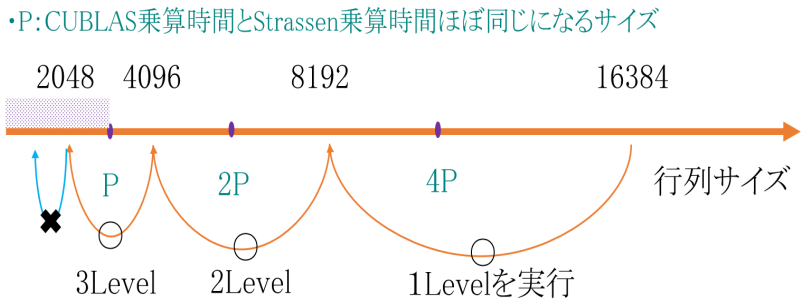


図 4.1: CUBLAS 乗算と Strassen アルゴリズムの演算時間比により最適化 Level の推定

4.3.2 最適化 Level の推定手順

Strassen-RUM 法の最適化 Level を求めるため、同時点 P の定義に示した、GPU の CUBLAS の乗算時間と LowLevel の Strassen-RUM 法の実行時間が必要である。

CUBLAS の乗算時間が CUDA の関数で直接測定できる。Strassen-RUM 法が Temporary 行列なしなので、メモリ確保などの影響の考えは必要ないから、LowLevel の Strassen-RUM 法の実行時間が 15 個のサブ行列の CUBALS 加減算と 7 個 CUBALS 乗算だけに表示できる。

これにより、最適化 Level を推定する必要な関数を定義する。x が入力行列のサイズを表す。

- $Gemm(x)$ が CUBLAS 乗算をサイズ $x \times x$ 大きさの正方行列の乗算を計算するためかかる時間、$Geam(x)$ が CUBLAS 加減算を正方行列の加減算を計算するためかかる時間を表す、この二つは実験する時具体的な関数を測定する。
- 式 4.1、$strassen_{RUM}(x)$ が Temporary 行列なしの LowLevel の Strassen アルゴリズムの実行時間を CUBLAS 乗算 $Gemm(x)$、CUBLAS 加減算 $Geam(x)$ で推定する式。
  LowLevel の Strassen アルゴリズムの乗算時間、15 個のサブ行列の加減算と 7 個乗算に分解できる、Strassen-RUM 法が Temporary 行列なしなので、メモリ確保などの影響を考える必要がない。

$$strassen_{RUM}(x) = 15 \times Geam(\frac{x}{2}) + 7 \times Gemm(\frac{x}{2}) \tag{4.1}$$

- 式 4.2、$SpeedUp(x)$ が先行研究中 CUBLAS での乗算にかかる時間と Strassen アルゴリズム一族かかる時間の比である。Strassen-RUM 法も Strassen アルゴリズム一族の一つなのでこの定義を利用できる。

$SpeedUp(x)$ により、Strassen アルゴリズムが CUBLAS 乗算より速くなれば、1 以上になる。1 以上のサイズが **Starssen アルゴリズムの有効範囲**と定義できる。

$$\begin{aligned} SpeedUp(x) &= Gemm(x) \div strassen_{RUM}(x) \\ &= Gemm(x) \div (15 \times Geam(\frac{x}{2}) + 7 \times Gemm(\frac{x}{2}) \end{aligned} \tag{4.2}$$

- 式 4.3 が式 4.2 を利用して同時点 P のサイズを推定する式。

$$SpeedUp(P) \approx 1 \tag{4.3}$$

- 最後に式 4.4 は同時点 P により、Strassen-RUM 法の最適化 Level を判定する式である。

$$OptimLevel(x) = \begin{cases} 1, & 0 \leq \mathrm{x} < 2\mathrm{P} \\ 2, & 2\mathrm{P} \leq \mathrm{x} < 4\mathrm{P} \\ 3, & 4\mathrm{P} \leq \mathrm{x} < 8\mathrm{P} \\ ... \end{cases} \tag{4.4}$$

式 4.2 から 4.4 を用いて、Strassen-RUM 法に対して、入力行列のサイズにより、最適な Level を推定する手順が：

- 1. 使っている実験環境の GPU の異なるサイズに対して、加減算時間 (Geam(x)) と乗算時間 (Gemm(x)) を測定し、MATLAB フィッティングして関数を求める、Strassen-RUM アルゴリズムの乗算時間 ($strassen_{RUM}(x)$) を計算する。
- 2.SpeedUp(x) を式 4.2 で計算し、式 4.3 で同時点 P に対応する行列サイズを求める。
- 3. 同時点 P と入力行列のサイズと対比して、式 4.4 で実験環境に関する最適な Level を推定する。

$Gemm(x)$ と $Geam(x)$ を測定した後、実験環境が変わらない場合、手順 3 だけで入力行列のサイズを用いて、最適な Level を推定することができる。

4.4 実験:Strassen-RUM 法の最適化 Level の評価

実験環境と条件は第 3.4.1 節に参照する。

4.4.1 境界線の検証実験

最適化 Level の測定	
測定目的	最適化 level のサイズの境界線を検証する
対象	Strassen-RUM 法
行列サイズ-A100	512 から 40,960
行列サイズ-V100	512 から 24,576
行列サイズ-P100	512 から 16,384

表 4.1: 最適な Level の実験の内容、サイズと目的

4.4.2 相関係数の測定

$Gemm(x)$ と $Geam(x)$ の測定は実験内容中にある各 GPU の実験サイズの範囲内にサイズの増加に従って密集から疎まで 10 個以上の測定点を選択し、第 3.4.1 節の基準で CUBLAS の加減算と乗算の時間を測定して、MATLAB で R_square(決定係数[1]) ≈ 1 のフィッティング関数を作る。

MATLAB でフィッティングした GPU Tesla A100、V100 と P100、CUBLAS バージョン 11.7 の性能関数 $Gemm(x)$ が三次関数、$Geam(x)$ が二次関数である。式 4.5 は三次関数の $Gemm(x)$ を、式 4.6 は二次関数の $Geam(x)$ を示す。

$$Gemm(x) = P1 \times x^3 + P2 \times x^2 + P3 \times x + P4 \tag{4.5}$$

$$Geam(x) = P1 \times x^2 + P2 \times x + P3 \tag{4.6}$$

フィッティングした値は表 4.2、SSE は残差平方和、R_square は決定係数。

A100 の性能関数	P1	P2	P3	P4	SSE	R_square
$Gemm(x)$	1.015e-10	6.727e-08	4.789e-04	0.9884	16.8795	1.0000
$Geam(x)$	1.386e-08	5.026e-06	0.03578	-	0.002700	1.0000
V100 の性能関数	P1	P2	P3	P4	SSE	R_square
$Gemm(x)$	2.79e-10	1.15e-08	-0.0008513	1.6940	38.9300	1.0000
$Geam(x)$	1.385e-08	5.042e-06	0.03585	-	0.002601	1.0000
V100 の性能関数	P1	P2	P3	P4	SSE	R_square
$Gemm(x)$	4.347e-10	3.821e-08	-3.907e-05	0.4332	2.0980	1.0000
$Geam(x)$	4.269e-08	5.093e-07	0.01638	-	0.0001960	1.0000

表 4.2: 測定した GPU Tesla A100、V100 と P100 の性能関数

[1]R_square が 1 に近い程相対的な残差が少ないことを表す、よく見かける値は 0〜1 のあたり。– https://ja.wikipedia.org/wiki/決定係数

これにより計算した $SpeedUp(x)$ は図 4.2(A100)、図 4.3(V100) と図 4.4(P100) に示す。

青い線は $SpeedUp(P) \approx 1$ の位置を示している。同時点 P は定義 (4.3) のような概数だが、誤差を測定するため、推定値そのまま使う。A100 の同時点 P が 6,342、V100 が 5,087、P100 が 5,128。

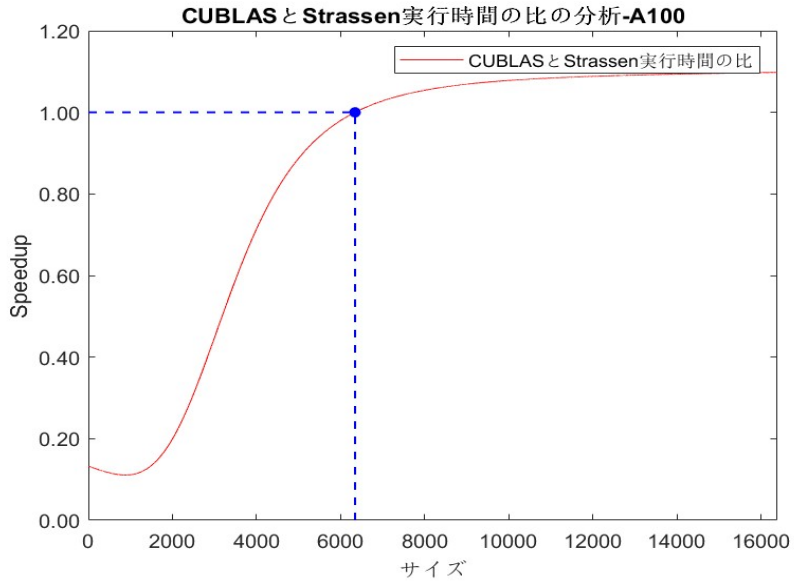


図 4.2: Strassen-RUM 法の $SpeedUp(x)$ 関数と同時点 P の推定値-A100

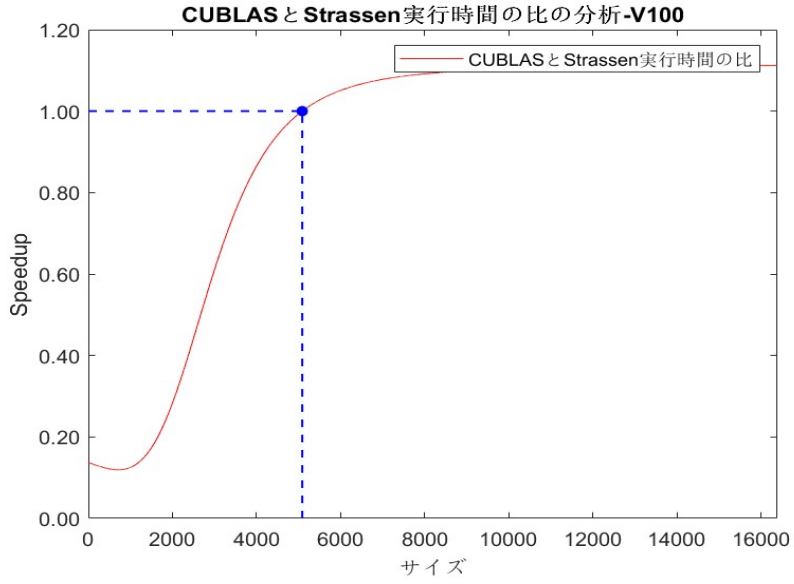


図 4.3: Strassen-RUM 法の $SpeedUp(x)$ 関数と同時点 P の推定値-V100

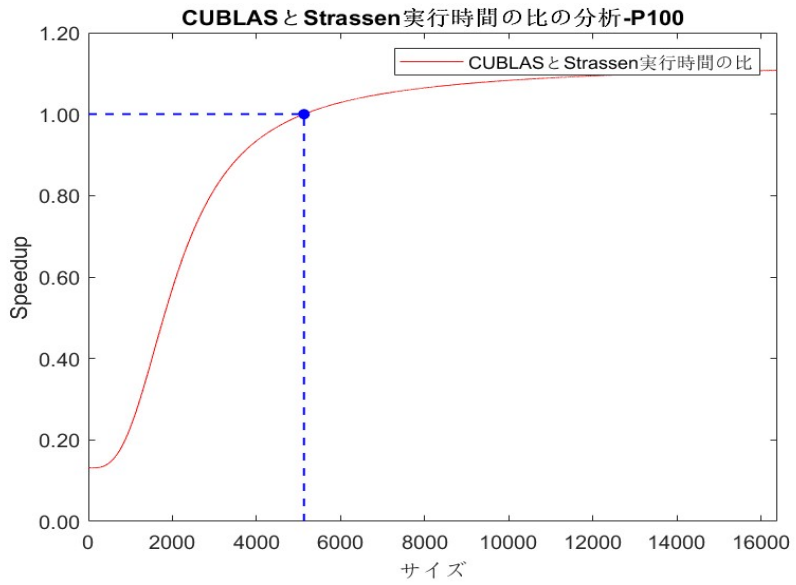


図 4.4: Strassen-RUM 法の $SpeedUp(x)$ 関数と同時点 P の推定値-P100

4.4.3 最適化 Level の評価

最適化 Level について、実際に測定した Level の境界線と同時点 P による予測した Level の境界線が図 4.5、図 4.6 と図 4.7 に示す。

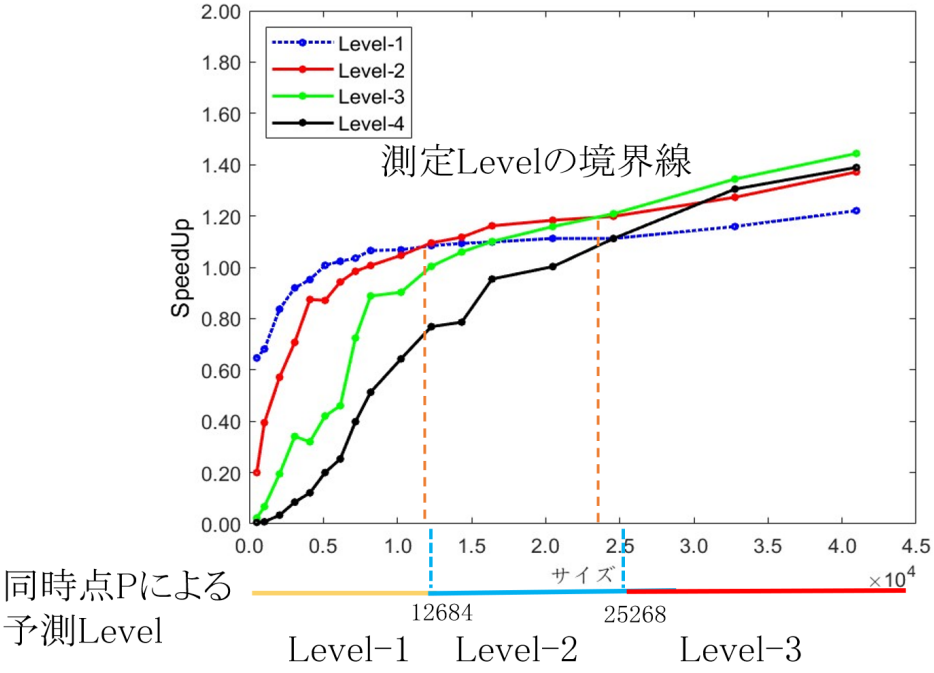


図 4.5: 最適化 Level の予測値と測定値-A100

図 4.5 に A100 の結果を示す、同時点 P が 6,342 である。

推定した最適化 Level の範囲が式 4.4 による、Level-1 が 0 ～ 12,684 サイズ、Level-2 が 12,684 ～ 25,268 サイズ、Level-3 が 25,268 ～ 50,536 サイズである、この部分が図 4.5 の下半に示した。

実際の測定 Level の境界線は図 4.5 の上半にある、Level-1 から 4 までの速度変化を表示した、一番高いのが対応サイズの最適化 Level である。Level-1 と Level-2 の境界線は約 12,000、Level-2 と Level-3 は約 24,000。

結果から見れば、まず最適化 Level 間の倍数関係が推定と合っている。同時点 P の倍増により、予測 Level と測定 Level の偏差も大きくなる、誤差全体が $1-\frac{24000}{25268}$ ≈5-6%ぐらいの程度である。

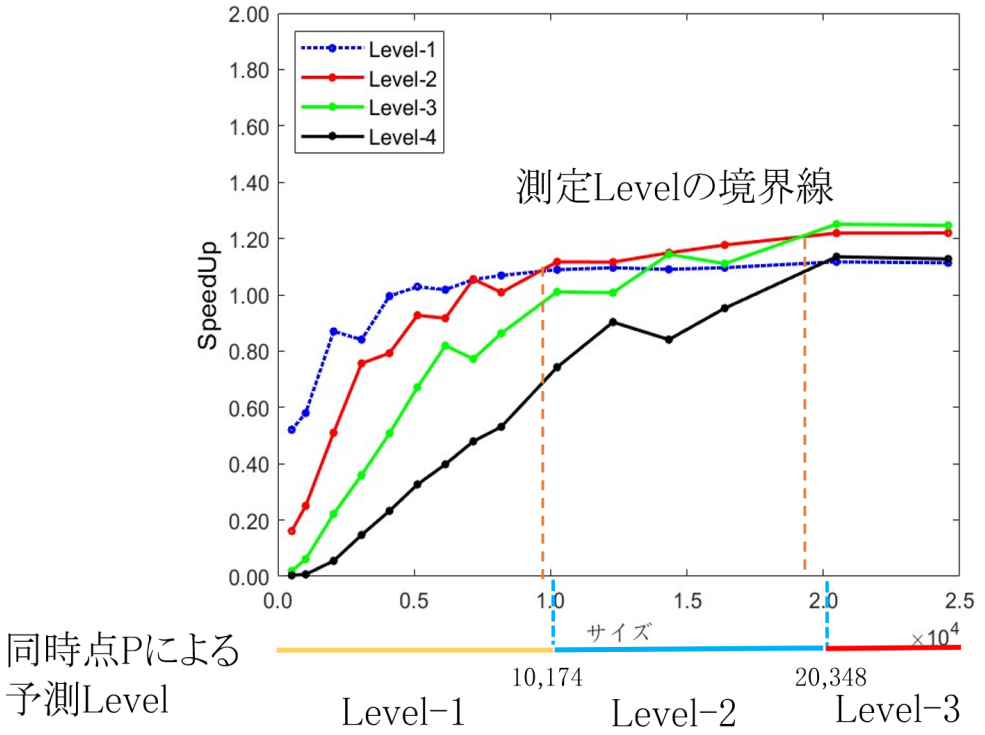


図 4.6: 最適化 Level の予測値と測定値-V100

図 4.6 に V100 の結果を示す、同時点が 5,087 である。

推定した最適化 Level の範囲が式 4.4 による、Level-1 が 0 ～ 10,174 サイズ、Level-2 が 10,174 ～ 20,358 サイズ、Level-3 が 20,258 ～ 40,516 サイズである、この部分が図 4.6 の下半に示した。実際の測定 Level の境界線は図 4.6 の上半にある、Level-1 と Level-2 の境界線は約 9,500、Level-2 と Level-3 は 19,000。

結果から見れば、まず誤差が $1-\frac{19000}{20348}$ ≈6-7%である。そして、A100 と比べて、V100 の Level-1 と Level-2 の境界線が 10,000 サイズから、12,000 までが曖昧になる (ほぼ同じ SpeedUp)、Level-2 と Level-3 の境界線も曖昧である。

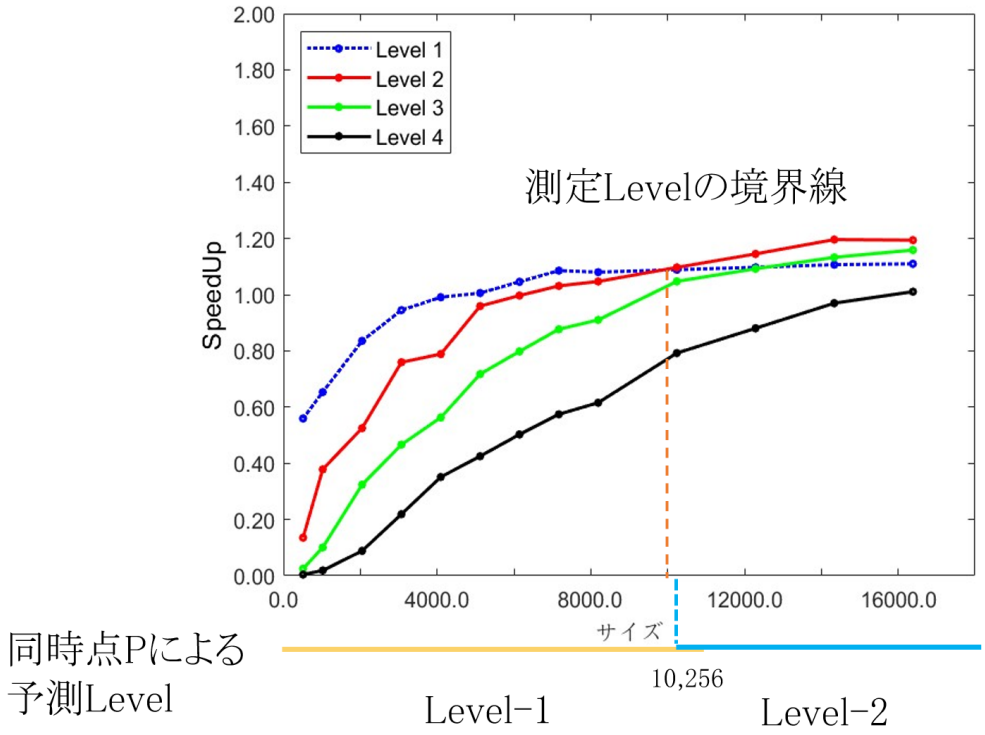


図 4.7: 最適化 Level の予測値と測定値-P100

図 4.7 に P100 の結果を示す、同時点が 5,128 である。

推定した最適化 Level の範囲が式 4.4 による、Level-1 が $0 \sim 10,256$ サイズ、Level-2 が $10,256 \sim 20,512$ サイズ、この部分が図 4.7 の下半に示した、Level-1 と Level-2 の境界線は約 10,000。実際の測定 Level の境界線は図 4.7 の上半にある。Level-1 と Level-2 の境界線は約 10,000。

結果から見れば、まず Level-3 がないから、倍数関係の判定ができない。誤差が $1 - \frac{10000}{10256} \approx$2-3%ぐらい程度である。

予測 Level がすべて測定 Level の境界線より 2-7%大きくなる原因として、$Gemm(x)$ と $Geam(x)$ を測定する時はデータのロード、計算、結果を出す順番で実行している、Strassen アルゴリズムの演算時間 $Strassen_{RUM}(x)$ を推定する時は単純に 15 個の加減算と 7 個乗算の時間の和を利用している。しかし、実際の GPU 演算する時、演算と次のデータの準備がある程度並列しているので、実際の $Strassen_{RUM}(x)$ が推定値より小さい、$SpeedUp(x)$ の定義から見れば、推定の $SpeedUp(x)$ が 1 になる時、実際の $SpeedUp(x)$ が既に 1 以上になった。つまり、同時点 P が推定値より小さい。同時点の測定については更に改良の余地がある。

4.5 終わりに

この節はStrassenアルゴリズムの再帰深度について分析し、問題を分割回数に転換し、解決の方法を提案した。現有のGPU向けStrassenアルゴリズムの分割ではなく、具体的な乗算を行う部分はLowLevelの乗算だけの性質とStrassenアルゴリズムの演算速度の変化を利用して、計算する行列のサイズが同時点Pの左辺か右辺に落ちることにより、再分割かどうかを判断する。この方法で、使うGPUの同時点Pを測定すれば、Strassen-RUM法に対して、入力行列のサイズにより、実行速度が一番速いLevelを判断できる。

最適化levelの実験の結果、提案した最適化Levelの判断基準はほぼ測定値と合っていることが分かったが2%-7%の誤差がある。最適化Levelの選択により、入力行列のサイズでStrassen-RUM法の一番速い実行Levelの選択が可能になる。第5章はStrassen-RUM法とStrassen-RUM法の最適化Levelの結果を踏まえて更に高速化への探索のために、Strassen-RUM法の並列化について研究する。具体的には、大塚手法の並列方針を分析した上でStrassen-RUM法の並列方針を決めるを目的にする。

第5章　提案手法の並列化

5.1　はじめに

本章は第 3.4.4 節、第 3 章の逐次演算スケジュールに対応する逐次試験中発見した問題、更に高速化できるかの問題について、Strassen-RUM 法の並列化手法を研究し、アルゴリズムの高速化を目的にする。そのため、まず大塚手法の並列手法の現状を調査する、発見した問題に応じて改良法を提案する。最後は Strassen-RUM 法の並列スケジュールを作成する。

5.2　先行研究の並列手法の現状調査

5.2.1　先行研究の並列方針

大塚手法の並列スケジュールは第 2 章の先行研究の紹介に表 2.1 で示した。

大塚手法の並列化は V100 の各 SM のレジスタ容量を配慮した二並列手法である。並列には小さいサイズの行列計算に影響が大きいから、LowLevel だけ並列を採用する、HighLevel は逐次方法で実行する。並列処理の対象は一番時間かかる乗算である、乗算を二つずつ並んで、二つの Stream に入れて並列実行を行う。並列部分の直前と直後は CPU と GPU 間のデバイス同期が二回必要である。

5.2.2　先行研究並列化の問題点

Strassen-RUM 法並列化の予備実験、大塚手法の並列化の実装結果が図 5.1 に示す。大塚手法の並列スケジュールは GPU Tesla V100、CUBALS バージョン 10.1 で実装した。大塚の論文によると、並列手法の 5000 サイズ以上が CUBLAS10.1 の乗算関数より速くなり、8000 サイズまで逐次手法と比べて高速化したと報告された。予備実験として大塚手法を同じコードで CUBLAS バージョン 11.7 の GPU Tesla V100 で実装した結果、CUBLAS、ドライバーなどの演算性能が上がったから、SpeedUp が下がった一方、CUBALS 並列性能の改善で並列化の効果も曖昧になった。

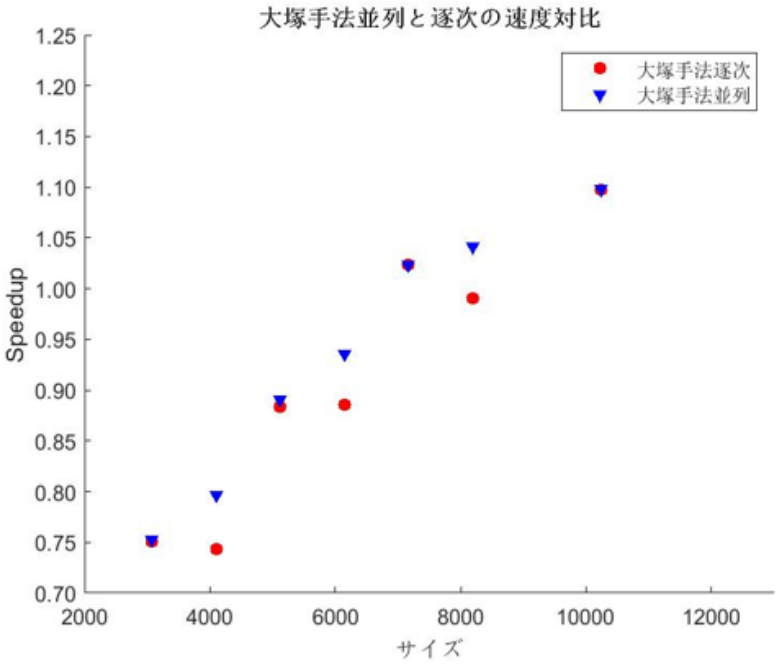


図 5.1: CUBLAS、ドライバーなどの性能改善により大塚手法の並列効果の変化

図 5.2 と図 5.3 は大塚手法の逐次と並列スケジュールを NVIDIA の分析ツール Nsight Systems で分析した結果。実験環境は倍精度、4,096 サイズの正方行列を CUBLAS バージョン 11.7、GPU A100(80GB) に実装[1]。一番上のオレンジ色、ピークと緑色はそれぞれ画像、計算パイプの使用率、SM の使用率と TensorCore の使用率である。

分析した結果、逐次バージョンが 10.795ms で並列が 11.390ms、並列化手法は逆に遅くなった。その原因は二つある。

- 一部 (全部ではない) の二つの並列した乗算が逐次より逆に遅くなった。
  図 5.2 の中に一番下の行には GPU がカーネルを実行する順番である。その中に、二つ LowLevel の乗算を逐次で実行する時間が 260.652us である、図 5.3 の中に同じ二つ乗算が並列実行すると 280.744us がかかった。
  原因として、逐次の乗算が CUBLAS11.7 に画像、計算パイプ、SM と TensorCore(図 5.2 の上の三行) 三つ計算速度に関する資源を十分使いました、乗算だけを並列すると、並列により資源の利用率があまり上がってできない。そして、資源競争で逆に遅くなった。一方、加減算の部分が逆に演算量が少ないから資源の使用率が低い、改善する余地がある。
- デバイス同期に時間がかかる。
  図 5.2 と図 5.3 の中に下の赤色矢印のところ、関数の引き換えの時間 (1.110us) と比べてデバイス同期が 10 倍ぐらい時間 (11.522us) がかかる。

[1]本研究の実験環境は具体的に第 3.4.1 節に紹介する。

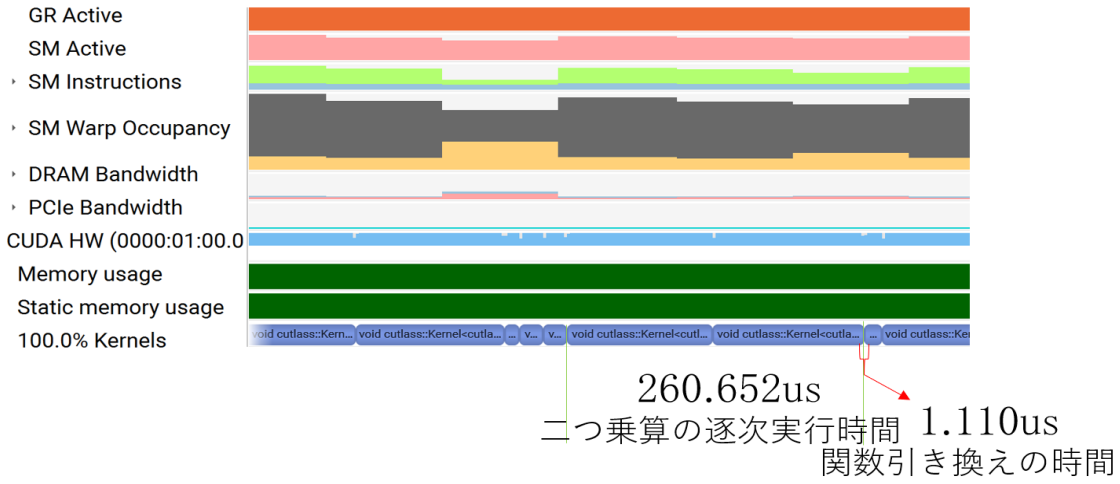


図 5.2: Nsight Systems で大塚手法逐次実行の分析 (部分)

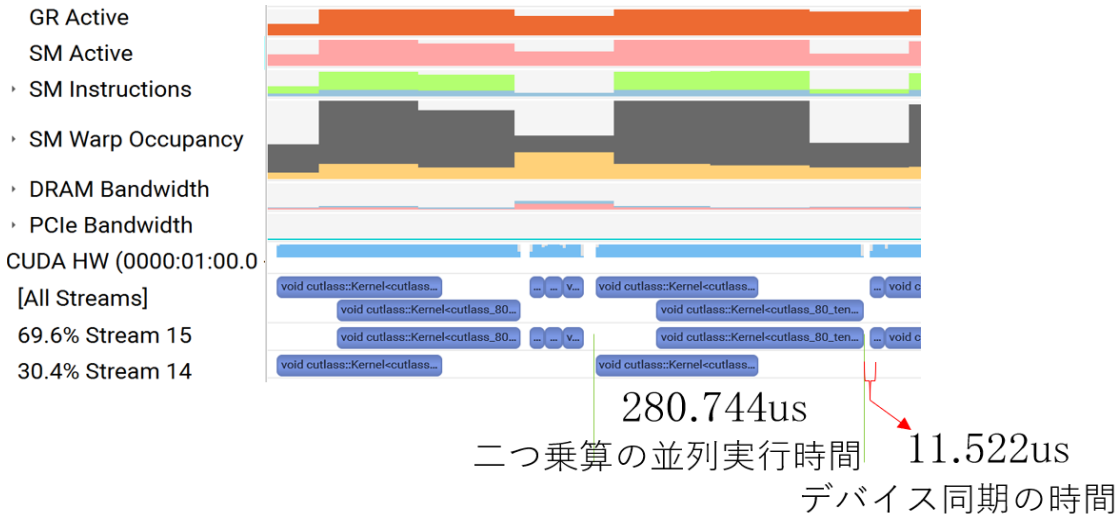


図 5.3: Nsight Systems で大塚手法並列実行の分析 (部分)

5.3 提案 3:Strassen-RUM 法の並列化

大塚手法の並列化問題を改良するため、提案 3 は提案手法 Strassen-RUM 法を並列化した再帰 Strassen アルゴリズムを作成するを目的にする。順番が並列方針を決める上で並列スケジュールを作成する。

5.3.1 Strassen-RUM 法の並列方針

第 5.2.2 節の分析により、Strassen-RUM 法の並列方針は以下になる：

- 1. 大塚手法の並列スケジュールの乗算部分は GPU の演算性能を十分使っているが、加減算はない。GPU の使用率を上げるの同時に資源競争を回避するため、Strassen-RUM 法の並列は乗算だけ並列化の代わりに、乗算と加減算、加減算と加減算の並列も含む。
- 2. デバイス同期の時間を節約するように、デバイス同期の代わりに、各stream中は逐次実行を利用して部分同期にする。部分同期というのは CPU と GPU の同期ではなく、CPU と Stream を同時する。別の Stream が同期の影響を受けなく、計算することができる。
  デバイス同期と部分同期は図 5.4 に示す。

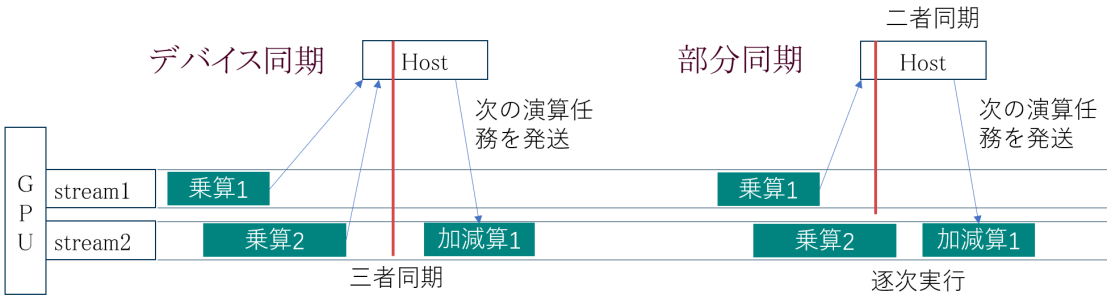


図 5.4: デバイス同期と部分同期

5.3.2 Strassen-RUM 法の並列スケジュール

並列方針により、Strassen-RUM 法のについて、各 Step の依頼関係とサイズの増加により並列性の削減 [15] により Stream の数は大塚手法と同じ二並列にする。

並列方針により、各 Step と Stream の対応関係を決める方法具体的には、前後の Step が同じ変数があるなら同じ Steam に入れる (Steam 中が逐次の性質を利用)、ないなら、異なる Stream に入れる、そして、他の Stream に変わった変数を利用する Step を Stream に発送する前に、変わった変数を持つ Stream と CPU を部分同期をする。

並列スケジュールは図 3.6 に示す。

Step	Strassen-RUM法の並列スケジュール	stream
1	$C_{11}^{n} = A_{11}^{n} - A_{21}^{n}$	1
2	$C_{12}^{n} = B_{22}^{n} - B_{12}^{n}$	2
3	$C_{21}^{n} = C_{11}^{n} C_{12}^{n}$	2
4	$C_{12}^{n} = A_{21}^{n} + A_{22}^{n}$	2
5	$S_{22}^{n} = B_{12}^{n} - B_{11}^{n}$	1
6	$C_{22}^{n} = C_{12}^{n} S_{22}^{n}$	1
7	$C_{12}^{n} = C_{12}^{n} - A_{11}^{n}$	1
8	$S_{12}^{n} = A_{11}^{n} B_{11}^{n}$	2
9	$S_{22}^{n} = B_{22}^{n} - S_{22}^{n}$	2
10	$C_{11}^{n} = C_{12}^{n} S_{22}^{n}$	2
11	$S_{11}^{n} = A_{12}^{n} - C_{12}^{n}$	1

Step	Strassen-RUM法の並列スケジュール	stream
12	$C_{12}^{n} = S_{11}^{n} B_{22}^{n}$	1
13	$C_{12}^{n} = C_{12}^{n} + C_{22}^{n}$	1
14	$C_{11}^{n} = C_{11}^{n} + S_{12}^{n}$	2
15	$C_{12}^{n} = C_{12}^{n} + C_{11}^{n}$	2
16	$C_{11}^{n} = C_{11}^{n} + C_{21}^{n}$	2
17	$S_{22}^{n} = S_{22}^{n} - B_{21}$	1
18	$C_{21}^{n} = A_{22}^{n} S_{22}^{n}$	2
19	$C_{21}^{n} = C_{11}^{n} - C_{21}^{n}$	2
20	$C_{22}^{n} = C_{22}^{n} + C_{11}^{n}$	1
21	$C_{11}^{n} = A_{12}^{n} B_{21}^{n}$	1
22	$C_{11}^{n} = S_{12}^{n} + C_{11}^{n}$	1

図 5.5: Strassen-RUM 法の並列スケジュール

Step5、15、18、21 を実行する前に部分同期が必要である。加減算も並列に含まれるので、HighLevel でも加減算部分の並列化の効果があるから、全 Level 並列化にする。

5.4 実験:Strassen-RUM 法の並列化の評価

実験環境と条件は第 3.4.1 節に参照する。

5.4.1 Strassen-RUM 法の並列手法の評価実験

Strassen-RUM 法の並列手法の評価	
測定目的	Strassen-RUM 法の並列方針の高速化効果を検証
対象	Strassen-RUM 法、大塚手法
行列サイズ-A100	最適化 Level の 512 から 40,960
行列サイズ-V100	最適化 Level の 512 から 24,576
行列サイズ-P100	最適化 Level の 512 から 24,576

表 5.1: 並列化の実験対象、サイズと目的

並列手法の評価はまず Strassen-RUM 法と大塚手法の Nsight Systems による GPU の使用率を比較する。この後は最適化 Level を採用した Level-1 から Level-3 の比較である。

Lai らの方法は並列していないため、比較対象ではない。

5.4.2 GPU 使用率上の比較

V100 と P100 は Nsight systems の GPU metrics 分析ができないので、A100 だけ比較した。

A100 で 4096 サイズの Strassen アルゴリズムを計算する時、GPU 使用率の比較が図 5.6 に示す。比較内容が一番上の GR Active(画像、計算パイプの使用率)、SM Active(SM の使用率) と SM Instructions(中身が TensorCore の使用率) である。

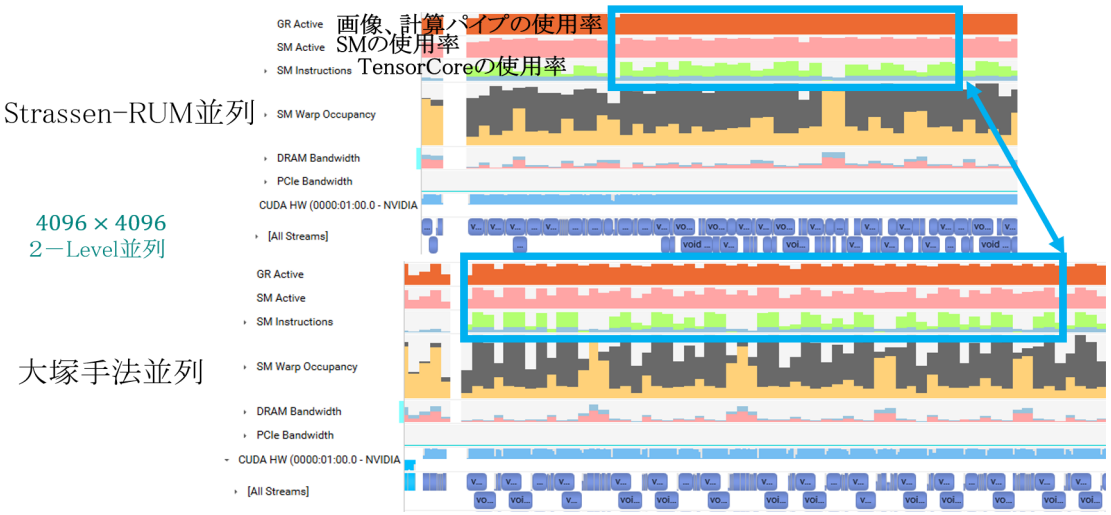


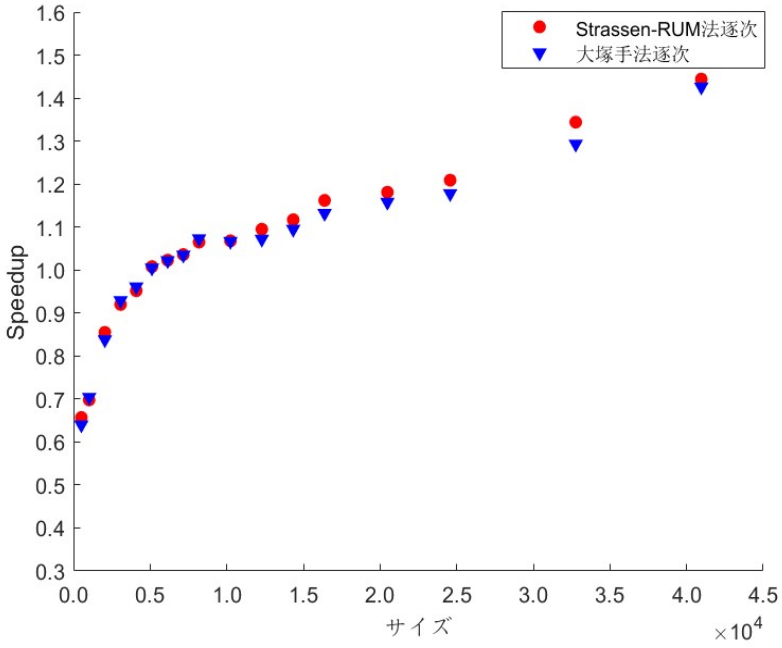
図 5.6: Nsight systems による Strassen-RUM 法と大塚手法の並列化効果の比較-A100

結果から見れば、Strassen-RUM 法の並列方針により大塚手法と比べて全体的に使用率が上がっていた、その中画像、計算パイプの使用率がほぼ 100%になった。

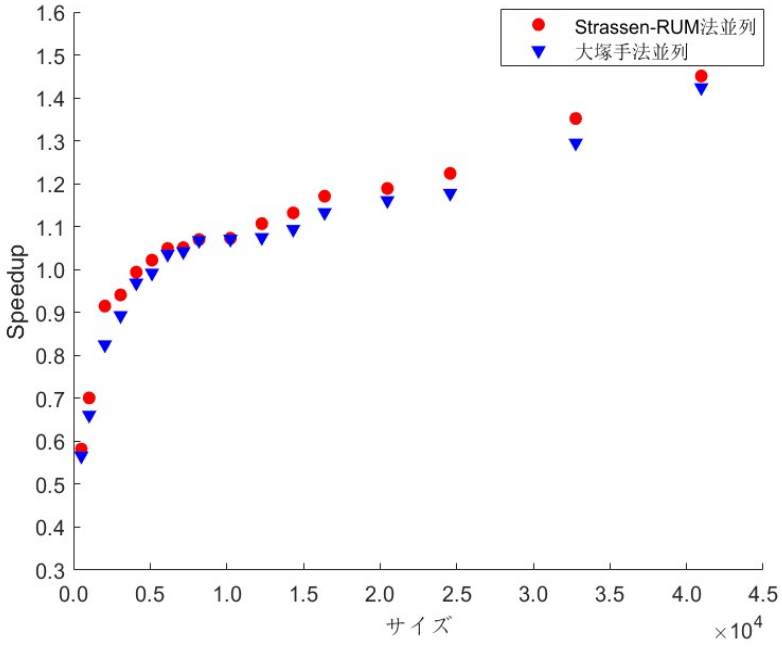
5.4.3 逐次と並列手法の比較

並列効果を示すために、Strassen-RUM 法と大塚手法に対して逐次演算と並列演算を比較する。

A100 で Level-1、2、3(512 サイズから 40,960 サイズまで) の比較は図 5.7 に示す。



(a) 逐次演算の比較



(b) 並列演算の比較

図 5.7: Strassen-RUM 法と大塚手法に対して逐次演算と並列演算の Level-1，2，3 の比較-A100

有効範囲 ($SpeedUp(x) > 1$) から見れば、逐次演算の場合、大塚手法と Strassen-

RUM 法が同じ 5,000 サイズ、並列した後、Strassen-RUM 法が 4,000 サイズ近くになった。

Level-1 の場合、二つ方法の逐次演算と並列演算の差がないから、この部分がほぼ並列方針による効果が分かれる。8,000 サイズまで Strassen-RUM 法が大塚手法より速くなった、最大値が 2,048 サイズ時、10.94%の高速化が達成した。1024 サイズ以下の場合、並列化が逆に遅くなった。

Level-2、3 の場合、二つ方法の逐次演算の場合、Level の増加とともに Temporary 行列削除による効果が良くなる。大きいサイズの場合、GPU の使用率が十分高いから、並列しても効果があるが、Level-1 と比べて、差が小さくなる。Strassen-RUM 法の最大の SpeedUp が 40,960 サイズで 1.451 倍達成した。

V100 で Level-1、2、3(512 サイズから 24,576 サイズまで) の比較は図 5.8 に示す。

有効範囲について、逐次演算の場合、大塚手法と Strassen-RUM 法が同じ 4,000 サイズ、並列化した後、あまり変化がない。

Level-1 に対して、2,000 サイズまで並列化の効果があるが、これ以上ほぼ差がない。A100 と同じで 1,024 サイズ以下は逆効果になる。

Level-2、3 の場合、Temporary 行列削除の効果がほぼないだが、並列化の効果が少しある。全体的に大塚手法の並列化よりは速くなった。Strassen-RUM 法の最大の SpeedUp が A100 より小さい、24,576 サイズで 1.286 倍達成した。

P100 で Level-1、2(512 サイズから 16,384 サイズまで) の比較は図 5.9 に示す。

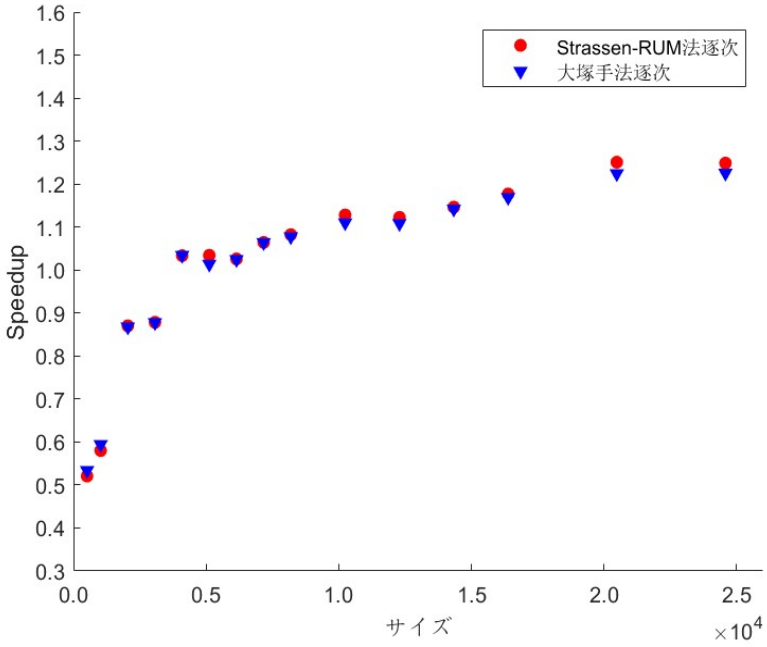
有効範囲について、逐次演算の場合、大塚手法と Strassen-RUM 法が同じ 4,000 サイズ、並列化した後が同じ 3,000 サイズになった。5,12 サイズ以下は逆効果になる。

Level-1 に対して、4,000 サイズまで並列化の効果がある。

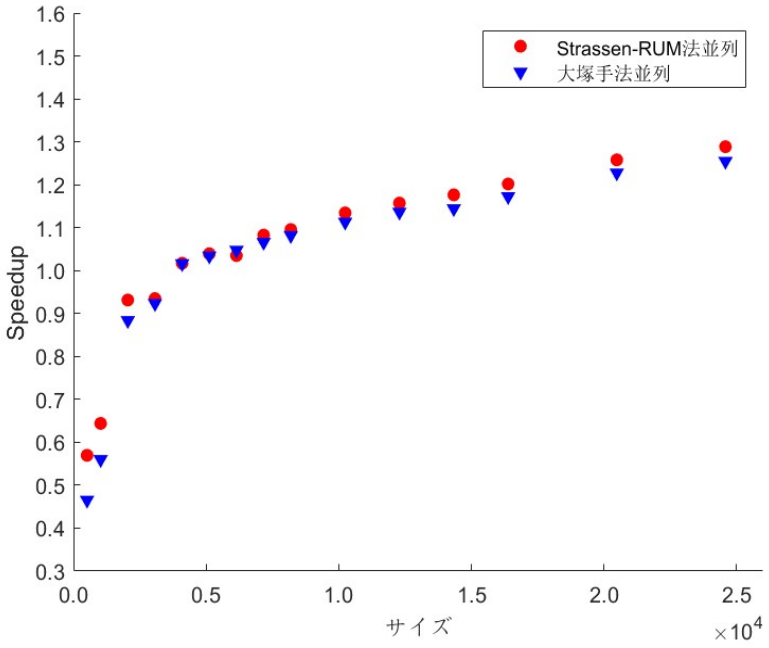
Level-2 の場合が A100 の結論と同じ、Temporary 行列削除する効果が Level-1 より良いが並列化の効果が悪くなった。Strassen-RUM 法の最大の SpeedUp が 16,384 サイズで 1.211 倍達成した。

A100、V100 と P100 の結果から見れば、並列化が有効範囲の増加に利点があることが分かった。並列化の逆効果について、Nsight　Systems でカーネルの実行を分析すると、原因としては Strassen-RUM 法が部分同期を採用したが普通の関数の引き換えよりはまだ時間がかかることである。

最大の SpeedUp から見れば、V100 と P100 が大体同じである、A100 の Tensor-Core が直接倍精度の演算ができるから、数多くの小さい行列の乗算に分解するほうが TensorCore をより利用できるから、SpeedUp がより良いことが原因であると推測する。

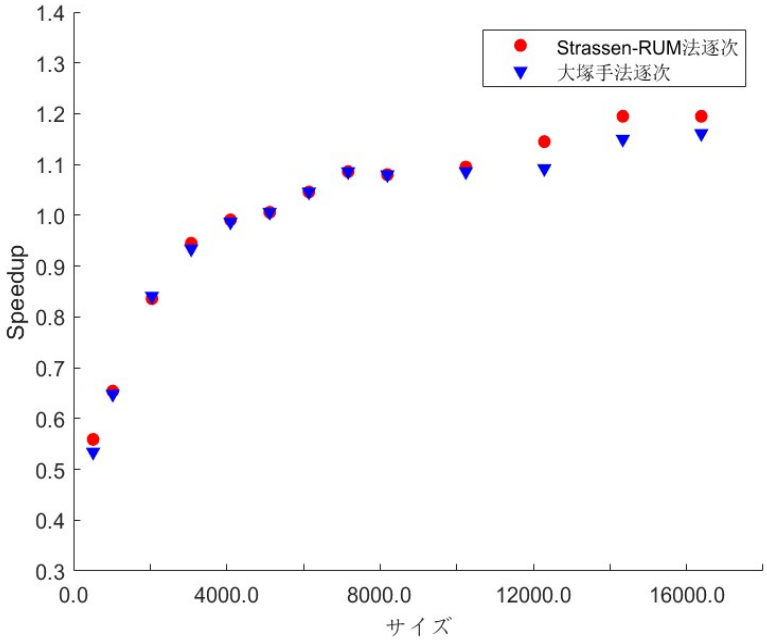


(a) 逐次演算の比較

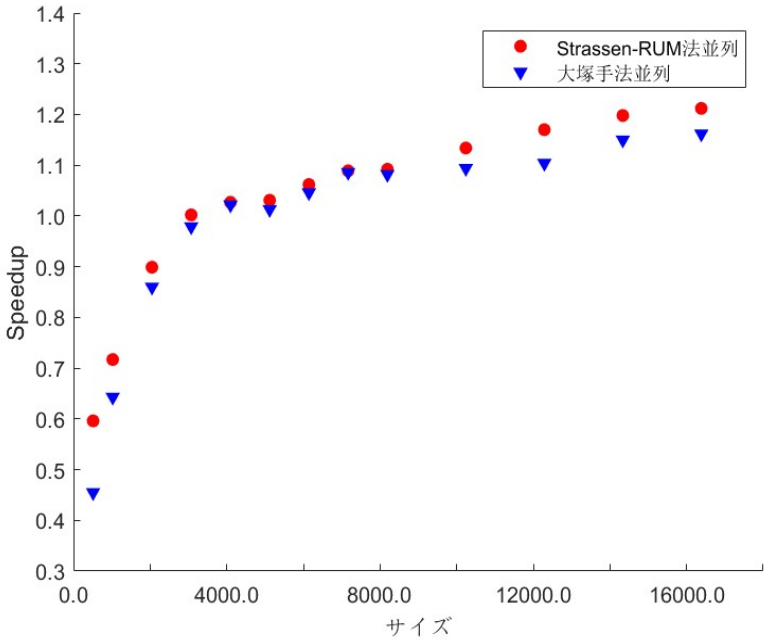


(b) 並列演算の比較

図 5.8: Strassen-RUM 法と大塚手法に対して逐次演算と並列演算の Level-1，2，3 の比較-V100



(a) 逐次演算の比較



(b) 並列演算の比較

図 5.9: Strassen-RUM 法と大塚手法に対して逐次演算と並列演算の Level-1，2，3 の比較-P100

5.5 補充試験：CUDA バージョン変更の影響

先行研究大塚の研究 [15] と本章最初、先行研究並列化の問題点を探す時の予備実験の結果によると、CUBLAS バージョンを変更すると、Strassen アルゴリズムの演算速度に変化がある。

CUBLAS バージョンの変更が提案手法である Strassen-RUM 法にどの影響を与えるかを明らかにするため、第 3.4.1 節にある実験環境と条件が CUBLAS バージョンだけ **11.7 から 10.1** に変換するだけ、他の環境と条件が変化しないで補充実験をする。

5.5.1 CUDA バージョン変更による影響の実験

CUDA バージョン変更による影響の実験	
測定目的	CUDA バージョンの変更する場合、 1.Strassen-RUM 法速度の変化があるか 2. 大塚手法と比べて結果が変化するか
対象	Strassen-RUM 法、大塚手法
CUBLAS バージョン	11.7 と 10.1
行列サイズ-V100	最適化 Level の 512 から 24,576
行列サイズ-P100	最適化 Level の 512 から 24,576

表 5.2: 並列化の実験対象、サイズと目的

A100 が CUBLAS バージョン 11.1 からサポートされているから、CUBLAS バージョン 10.1 には使えないので V100 と P100 だけを実験の対象にする。

異なる CUBLAS バージョンの実現は Docker で実現した。CUBLAS バージョンを変更した後、新たに Makefile でプログラムを生成する。

5.5.2 Strassen-RUM 法実行速度の変化

CUBLAS バージョンの変化による Strassen-RUM 法の演算速度の変化を示すために、V100、P100 に対して、CUBLAS バージョン 11.7 と CUBLAS バージョン 10.1 の演算速度を測定した。

V100 に対して、Strassen-RUM 法の最適な Level の CUBLAS バージョン 11.7 の演算速度と最適な Level の CUBLAS バージョン 10.1 の演算速度の比と差が図 5.10 に示す。左軸、点図が異なるバージョンの速度の比 (式 5.1)、右軸、線図が異なるバージョンの速度の差 (式 5.2) である。

$$\text{実行時間の比} = CUBLAS11.7\text{の実行時間}/CUBLAS10.1\text{の実行時間} \tag{5.1}$$

$$\text{実行時間の差} = CUBLAS11.7\text{の実行時間} - CUBLAS10.1\text{の実行時間} \tag{5.2}$$

図 5.10 の逐次演算の結果から見れば、入力行列のサイズが 512 の時だけ二つバージョンの実験時間の比が 1.2 倍以上になったが、6,144 サイズ以下が 1 倍に少し偏差がある。6,144 サイズ以上の場合、実行時間の比がほぼ 1 倍になる、バージョンに影響が少ない。実行時間の差から見れば、行列サイズが大きくなると、最適化 Level の実行時間が最大 35.934ms の差があるから、バージョンに影響が少ないとは言え、ないとは言えない。

図 5.10 の並列演算の結果がほぼ逐次演算と同じだが、比と差の波動が逐次演算より小さい。Strassen-RUM 法の並列演算が逐次演算より GPU の資源を利用しているから、異なる CUBLAS バージョンの差が小さくなるのが原因と推定する。

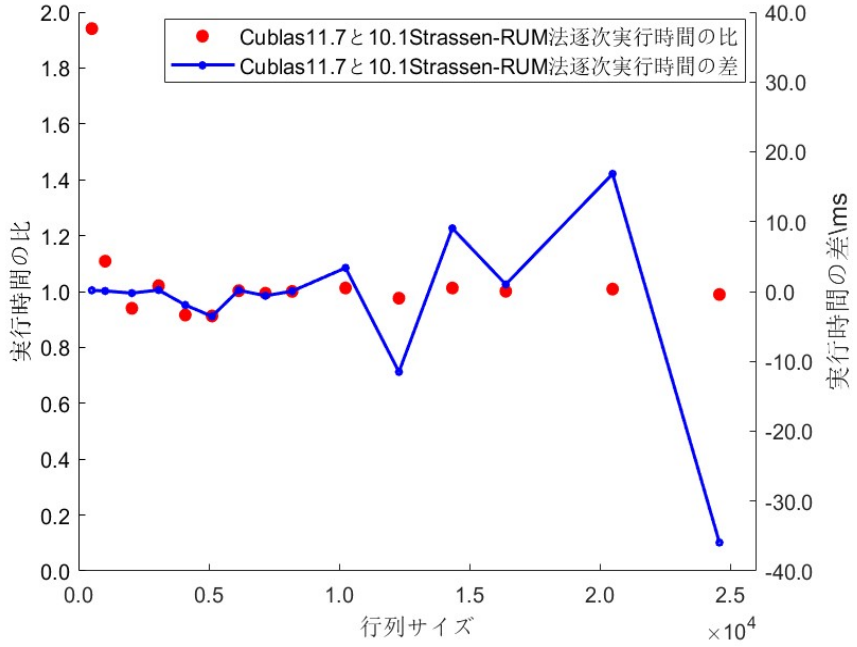
V100 の二つバージョンの実験時間の比と差から見れば、Strassen-RUM 法も Strassen アルゴリズムの先行研究と同じ、CUBLAS バージョンの変化による実行時間が影響されていることが分かった。

P100 の Strassen-RUM 法の最適な Level の CUBLAS バージョン 11.7 と 10.1 の演算速度の比と差が図 5.11 に示す。

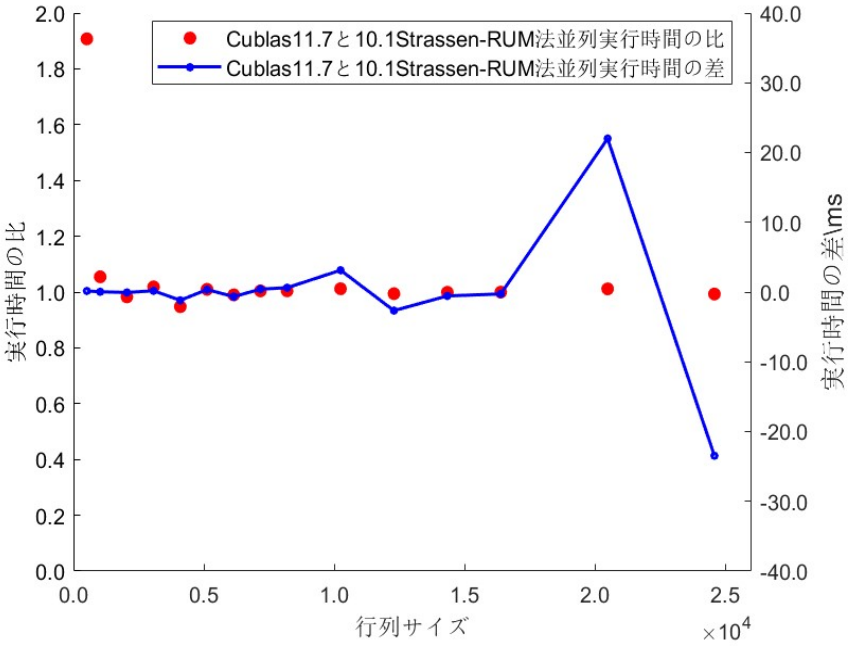
V100 の結果と違って、P100 に対して、Strassen-RUM 法の逐次演算と並列演算が CUBLAS バージョンの変化による影響がほぼない。4,096 サイズだけ、CUBLAS バージョン 11.7 の実行速度が少し速くなった。

V100 と P100 の結果が違う理由として、P100 が CUBLAS バージョン 8.0 から、V100 が 9.0 からサポートされていた。10.1 バージョン以降、P100 は既に最適化されたから、CUDA の調整が特にしていなかったが V100 には調整されていたから、演算速度が少し変化があると思う。

結論として、Strassen-RUM 法の実行速度が CUBLAS バージョンに関係ある。この結論により、CUBLAS バージョンの変更により、大塚手法と比べて、Temporary 行列を削除した利点と並列方針の改良が効くかどうかを実験する必要がある。

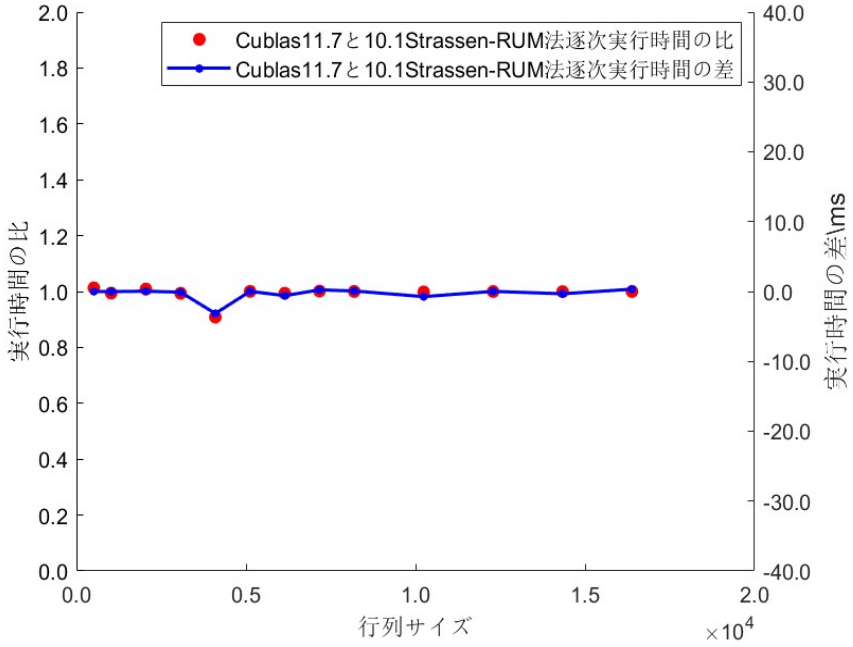


(a) 逐次演算の比較

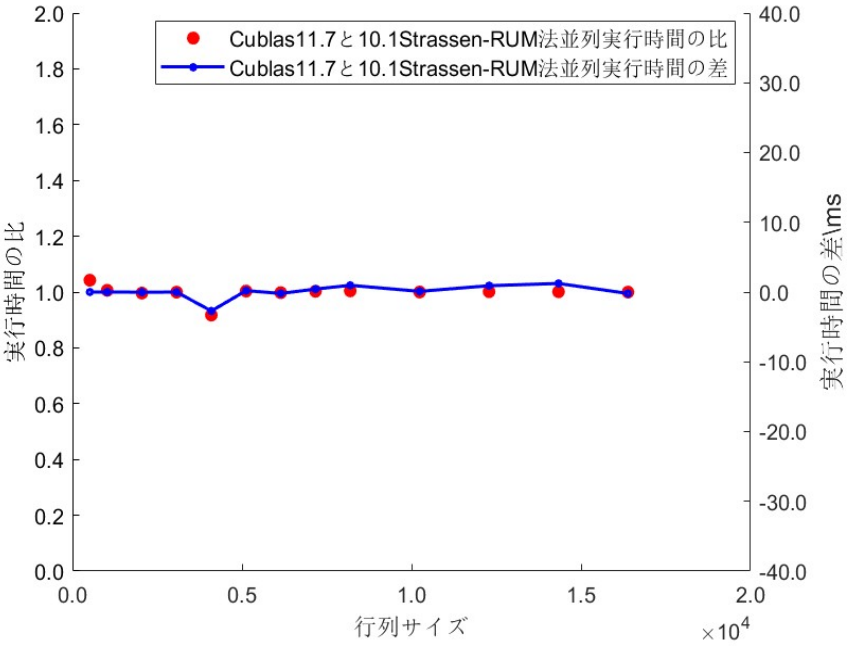


(b) 並列演算の比較

図 5.10: Strassen-RUM 法の最適な Level の CUBLAS バージョン 11.7 と 10.1 演算速度の比と差-V100



(a) 逐次演算の比較



(b) 並列演算の比較

図 5.11: Strassen-RUM 法の最適な Level の CUBLAS バージョン 11.7 と 10.1 演算速度の比と差-P100

5.5.3 先行研究と比較の変化

CUBALS バージョン 10.1 で、大塚手法と Strassen-RUM 法の最適化 Level の逐次と並列演算の速度を測定した。

CUBALS バージョン 10.1、V100 で Level-1、2、3(512 サイズから 24,576 サイズまで) の比較は図 5.12 に示す。

V100、CUBLAS バージョン 10.1 の逐次演算の結果から見れば、バージョン 11.7 との比と差が大きい Level-1 の一部の SpeedUp が不安定な状態になったが、入力行列のサイズが大きくなると、バージョン 11.7 と同じ、Strassen-RUM 法が大塚手法より速くなる。

CUBALS バージョン 10.1、P100 で Level-1、2、3(512 サイズから 16,384 サイズまで) の比較は図 5.13 に示す。

並列の結果が逐次の結果と比べて、よりバージョン 11.7 の結果に似ている、Level-1 に並列の効果が高い、Level-2、3 に Temporary 行列を削除した効果が高くなる。逐次演算との対比が第 5.5.2 節の結論と同じ、並列の方が CUBLAS バージョンの変化による影響が小さい。

P100、CUBLAS バージョン 10.1 の大塚手法との対比が CUBLAS バージョン 11.7 と比べて、大塚手法も Strassen-RUM 法と同じ速度の変化があまりないので、結果と結論がバージョン 11.7 と同じになる。

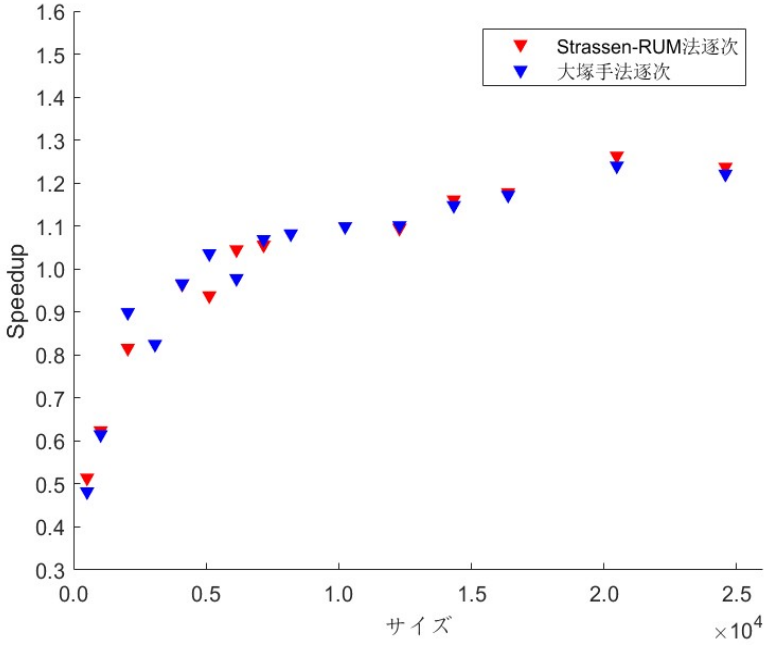
図 5.10 から図 5.13 から見れば、まず CUBLAS バージョンの変化による Strassen アルゴリズムの実行速度が変化することが分かった。そして、速度を変化しても、提案手法 Strassen-RUM 法と Strassen-RUM 法の並列方針が演算速度の改善に良い影響があることが分かった。

5.6 終わりに

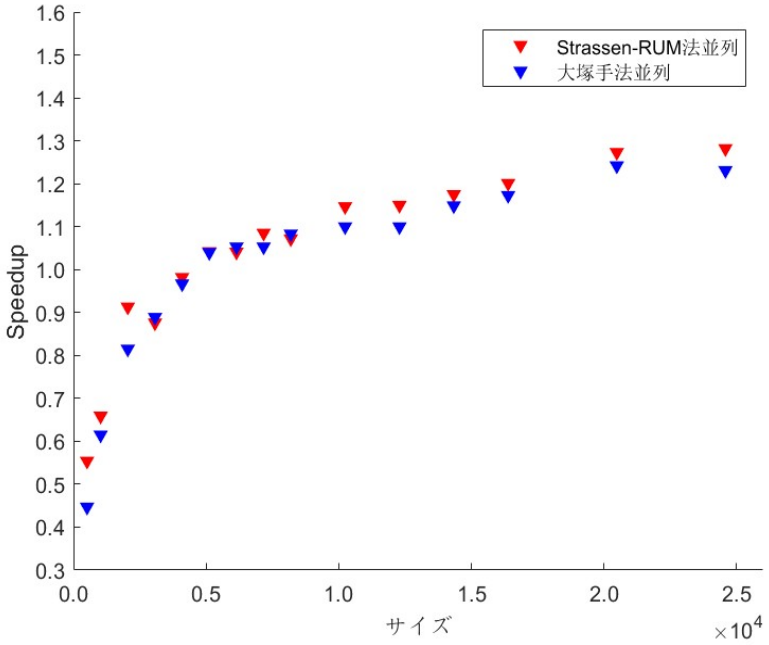
本章はまず大塚手法の並列方針を改めて紹介した。その並列方針の問題点を実験で発見した。分析結果から、乗算だけ並列の代わりに、加減算も並列に含むこととデバイス同期の代わりに部分同期を使うことを Strassen-RUM 法の並列方針で提案した。

並列方針により、全 Level に適用する Strassen-RUM 法の二並列スケジュールを作成した。並列効果から見れば、Level-1 に 8,000 サイズまで効果があり、最大 10.94%(A100) の高速化が達成できた。Level-2、3 の場合は並列効果が Level-1 と比べて良くないが Temporary 行列の削除による Level の増加とともに、演算速度の改善がある、最大 CUBLAS11.7 の標準行列乗算の 1.451 倍 (A100) 達成した。

補充実験により、CUBLAS バージョンの変更が Strassen-RUM 法を含む Strassen アルゴリズム一族に影響があるが、Strassen-RUM 法とその並列方針が演算速度の改善に効くことが分かった。

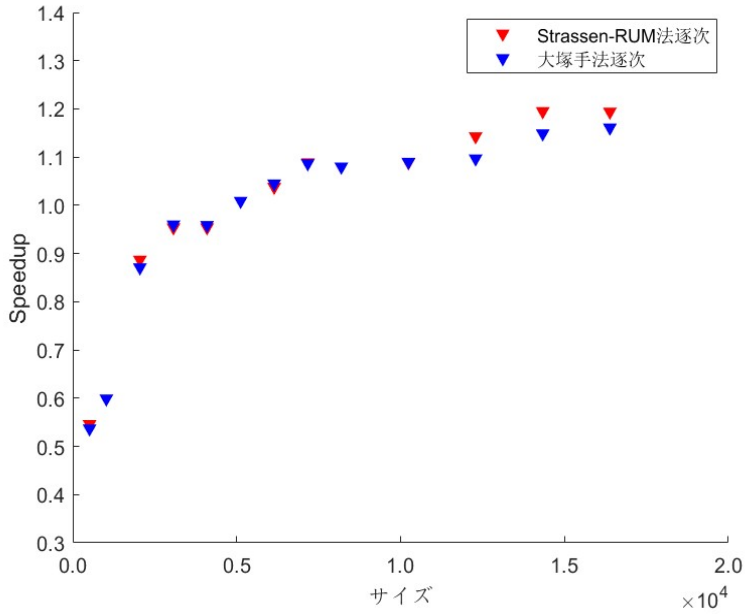


(a) 逐次演算の比較

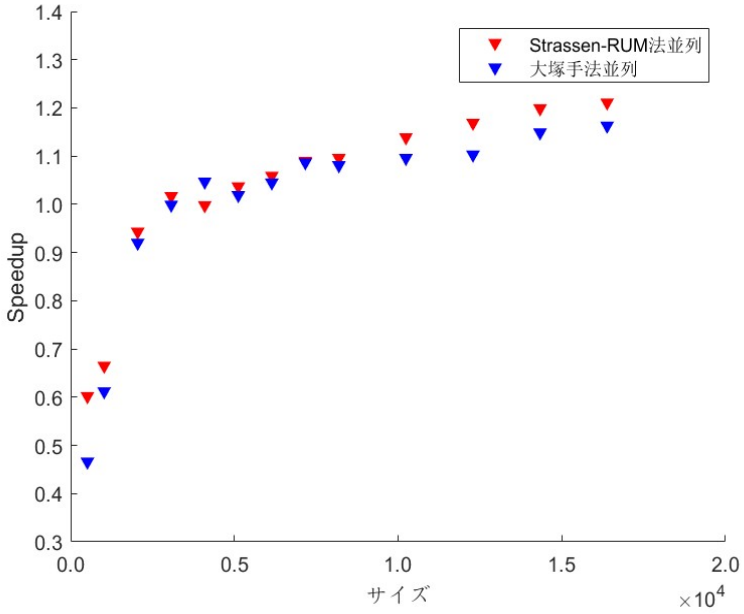


(b) 並列演算の比較

図 5.12: Strassen-RUM 法と大塚手法に対して逐次演算と並列演算の Level-1，2，3 の比較-V100,CUBLAS10.1



(a) 逐次演算の比較



(b) 並列演算の比較

図 5.13: Strassen-RUM 法と大塚手法に対して逐次演算と並列演算の Level-1，2，3 の比較-P100,CUBLAS10.1

第6章　結言

本研究では、主に GPU を用いた行列乗算方法の一つ、Strassen アルゴリズムのメモリ使用量を減らすことを中心に、この上で高速化の提案をした。また最適な Level の判断方法について研究を行った。NVIDIA GPU Tesla A100 を中心に、V100 と P100 を対比で提案の内容を評価した。本研究の主要な寄与は以下になる：

- 第 3 章では、Strassen アルゴリズムのメモリ使用量を減らすことを解決するため、大塚手法を基にして、再帰 Strassen アルゴリズムに対して、必要なメモリ空間を Malloc で確保せず、上位 Level の未使用行列を下位 Level のサポート行列として提供することにより、全体的に Temporary 行列を削除する方法を提案した。提案の考え方に応じて、サポート行列必要な再利用できるメモリ空間を提供できる、再帰可能な逐次演算スケジュールを作成した、このスケジュールが提案手法 Strassen-RUM 法と命名する。実験によると、Strassen-RUM 法がメモリ使用量を減らすことを目指す Lai らの方法より最大 18.11%、大塚手法より最大 43.35%のメモリを節約できる。すべての Temporary 行列の削除より、逐次プログラムの実行速度も 2 ～ 4% ぐらい上がっていた。速度面の結果により、最適化 Level と更に高速化の問題が発見した。
- 第 4 章では、最適化 Level の問題を解決するため、Strassen アルゴリズムの最適な深度の問題を分割回数に転換し、再帰予測問題の式複雑化を回避した。CUBLAS 乗算時間と Strassen アルゴリズム乗算時間がほぼ同じの入力行列サイズ、同時点 P により、各 Level の最適な範囲を選択する方法を提案した。実験によると、A100、V100 と P100 で同時点 P より予測した最適な Level の範囲が実測した最適な Level の範囲は 2-7%の誤差があるが、最適化 Level の倍数関係が理論と同じ。同時点の測定には更に改良の余地がある。
- 第 5 章では、第 3 章の Strassen-RUM 法の逐次演算スケジュールの上に、更に高速化を目指す研究である。CUBLAS バージョンによる並列性能の向上に応じて、GPU の使用率を上げるの同時に資源競争を回避するため、乗算だけ並列化ではなく、加減算も並列に含むこととデバイス同期時間を節約できる部分同期の並列手法を提案した。実験によると、A100 の場合、8,000 サイズ以下の場合、先行研究より最大 10.94%高速になった。40,960 サイズに対して、Strassen-RUM 法の並列化が最大 CUBLAS11.7 の標準行列乗算より 1.451 倍の SpeedUp が達成できた。V100 と P100 にも Strassen-RUM 法の並

列方針が適用している。CUBLAS バージョンの変更が演算速度に影響があるが、Strassen-RUM 法とその並列方針が演算速度の改善に効く。

今後の研究として、LowLevel の乗算と加減算が CUBLAS の関数ではなく、特化の演算カーネルの作成より、Temporary 行列の削除や、TensorCore を十分利用して乗算と加減算の合併演算の作成。及び複数 GPU の連合演算に提案手法の応用などまだ改良の余地がある。今後の展望として、本研究の再利用手法を他の分割統治アルゴリズムへの応用、及び Strassen アルゴリズムなど高速乗算方法が GPU の大規模の行列乗算への応用などが期待される。

謝 辞

本研究の遂行にあたり、多くの方々にご指導ご鞭撻を賜りました。

研究方針について丁寧な助言や指導をしていただいた井口 寧教授に厚く御礼申し上げます。

同学科教授金子 峰雄先生、並びに同学科教授田中 清史先生には、本論文の作成にあたり、中間審査に適切なご助言を賜りました。感謝申し上げます。

最初に GPU に関する貴重なアドバイスや意見を頂きまして、井口研究室の先輩伊藤 健一氏に感謝申し上げます。

井口研究室の同期、後輩の皆様には、修士研究の二年三月間のあたり多くのご助言、意見を頂きました。本当にありがとうございました。

最後に、研究ツールとして、演算能力が高い GPU を作って研究順調にを進められる NVIDIA 株式会社に感謝いたします。

研究業績

[1] LI RUIZHI, 井口 寧. “入力行列を保持するメモリ空間の再利用に注目した再帰 Strassen アルゴリズムのメモリ最適化”, 2022 年度 電気・情報関係学会北陸支部連合大会, F2-3, 1 page, 金沢大学, Sep. 03, 2022

参考文献

[1] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#the-benefits-of-using-gpus

[2] S. Y. Jeremy Appleyard, "Programming Tensor Cores in CUDA 9," NVIDIA Corporation, NVIDIA Technical Report, 10 2017. [Online]. Available: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

[3] NVIDIA Corporation, "Nvidia a100 tensor コア gpu アーキテクチャ," Tech. Rep., 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/ja/Solutions/Data-Center/documents/nvidia-ampere-architecture-whitepaper-jp.pdf

[4] V. STRASSEN, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 13, pp. 354–356, 1969.

[5] 坂本真貴人, 藤井昭宏, and 田中輝雄, "Strassen のアルゴリズムを付加した行列積自動チューニングライブラリ," *Research Report of High Performance Computing(HPC)*, vol. 2013, no. 6, pp. 1–7, 02 2013.

[6] S. Winograd, "On multiplication of 2 × 2 matrices," *Linear Algebra and its Applications*, 1971.

[7] B. Kumar, C.-H. Huang, R. Johnson, and P. Sadayappan, "A tensor product formulation of strassen's matrix multiplication algorithm with memory reduction," in *[1993] Proceedings Seventh International Parallel Processing Symposium*, 1993, pp. 582–588.

[8] D. H. Bailey, K. Lee, and H. D. Simon, "Using strassen ' s algorithm to accelerate the solution of linear systems," *The Journal of Supercomputing*, vol. 4, no. 4, p. 357–371, Nov 2004.

[9] M. Thottethodi, S. Chatterjee, and A. Lebeck, "Tuning strassen's matrix multiplication for memory efficiency," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998, pp. 36–36.

[10] M. Kandegedara and D. N. Ranasinghe, "Functional parallelism with shared memory and distributed memory approaches," in *2008 IEEE Region 10 and the Third international Conference on Industrial and Information Systems*, 2008, pp. 1–6.

[11] P. Yugopuspito, Sutrisno, and R. Hudi, "Breaking through memory limitation in gpu parallel processing using strassen algorithm," in *2013 International Conference on Computer, Control, Informatics and Its Applications (IC3INA)*, 2013, pp. 201–205.

[12] P.-W. Lai, H. Arafat, V. Elango, and P. Sadayappan, "Accelerating strassen-winograd's matrix multiplication algorithm on gpus," *IEEE International Conference on High Performance Computing, Data, and Analytics*, 2013.

[13] J. Li, S. Ranka, and S. Sahni, "Strassen ' s matrix multiplication on gpus," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, Jan 2012.

[14] A. ul Hasan Khan, M. Al-Mouhamed, and A. Fatayer, "Optimizing strassen matrix multiply on gpus," in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2015, pp. 1–6.

[15] 大塚達史, "Gpu 向き strassen アルゴリズムの最適化," Master's thesis, 北陸先端科学技術大学院大学, Mar 2020.

[16] NVIDIA Corporation, "CUDA C Best Practices Guide," NVIDIA Technical Report, 4 2023. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[17] N. Corporation, "Nvidia tesla v100 gpu アーキテクチャ," Tech. Rep., 2017. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf

[18] NVIDIA Corporation, "Nvidia tesla p100," Tech. Rep., 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf