

GPU 向き Strassen アルゴリズムの最適化

大塚 達史 井口寧

概要: GPU を用いた Strassen アルゴリズムによる行列の乗算の高速化を行った。NVIDIA GPU V100, P100 および K20 を使用し、部分行列の乗算に CUBLAS ライブラリ 10.1 を用いた。Strassen アルゴリズムは行列の乗算を高速化する方法として知られているが、GPU に適用するためには計算の途中に出てくる temporary 行列のためのメモリの確保と解放に余分な時間を要すること、および部分行列の乗算の並列実行数が GPU リソースによって制限される問題がある。これらの問題を解決するために本研究では temporary 行列を削除し、そして部分行列の乗算を 2 個並列実行するプログラムを作成した。本研究のプログラムはこれまで報告されてきた Strassen アルゴリズムの GPU による計算の先行研究で最速のプログラムよりもすべての行列のサイズで高速となった。V100 を用いた行列サイズ $5,200 \times 5200$ の計算で先行研究のプログラムより 7 % 高速になった。また本研究の 2-level Strassen アルゴリズムのプログラムは CUBLAS-10.1 を用いた行列の標準的な乗算よりも行列サイズ $14,336 \times 14,336$ で 12 % 高速になった。逐次計算 1-level Strassen アルゴリズムの speedup の予測式を導いた。予測式は三代目の GPU による実測値を正確に再現した。

Abstract: We made acceleration of matrix-matrix multiplication by employing Strassen's algorithm with GPU. NVIDIA GPU V100, P100 and K20 were used, and multiplications of sub-matrices were carried out by using CUBLAS-10.1. Strassen's algorithm is known to accelerate matrix-matrix multiplication, but two problems are present in GPU implementation of Strassen's algorithm. The first problem is an extra time resulting from memory allocation and deallocation for temporary matrices which occur during the course of calculations, and the second problem is the restriction of parallel execution of sub-matrix multiplications due to insufficient GPU resources. In order to overcome these problems we developed a program without a temporary matrix and with parallel execution of two sub-matrix multiplications. Our program is faster than the fastest program reported in earlier studies on Strassen's algorithm with GPU. On V100 it is faster than the earlier program by 7 % for $5,200 \times 5200$ matrix size. Our program of 2-level Strassen's algorithm is faster than the standard matrix-matrix multiplication with CUBLAS-10.1 by 12 % for $14,336 \times 14,336$ matrix size. We derived a theoretical equation of the speedup for 1-level Strassen's algorithm with GPU. The equation accurately reproduces experimental values for three generations of GPUs.

1. はじめに

GPU はその高い並列計算処理能力のために近年多くの分野で広く活用されている。しかし GPU を使用するためのプログラムの作成では、アーキテクチャの複雑な特性を考慮しなければならない。そのためこれまでの GPU のプログラム作成では auto-tuning あるいは試行錯誤を繰り返す hand-tuning の方法が多く用いられてきた。最近 Lobeiras らは [1] は、分割統治アルゴリズム型の計算問題に GPU のアーキテクチャーに起因するリソースの影響を解析し、その影響を系統的にプログラムに組み込み計算を高速化する方法を開発した。彼らが対象とした問題は Fast Fourier Transform と tri-diagonal system solver アルゴリズムである。本研究では分割統治型の問題の一つである行列の乗算

の Strassen アルゴリズムの計算の高速化に同様なアプローチで取り組むことを目指した。

行列の乗算は基本的な線形代数計算の一つであり、多くの科学技術分野の数値計算で重要な位置を占めている。最近は大きなサイズの行列の乗算を必要とする機会が増え、大きなサイズの行列の乗算を高速化する Strassen アルゴリズムが注目を集めている。これまで Strassen アルゴリズムの CPU による計算は多く行われてきたが、それに比べ GPU による計算の報告は少ない。それらの計算の中での最速となるプログラムが Lai ら [2] によって報告されている。しかし彼らの論文では Strassen アルゴリズムの計算の高速化に GPU のリソースがどのように影響しているのかが十分に明らかにされていない。

本研究では GPU のリソースが Strassen アルゴリズムの計算の高速化にどのように影響するか解明した。特に

^{†1} 現在、北陸先端科学技術大学院大学

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

$$\begin{array}{l} S_1 = A_{21} + A_{22} \\ S_2 = S_1 - A_{11} \\ S_3 = A_{11} - A_{21} \\ S_4 = A_{12} - S_2 \\ S_5 = B_{12} - B_{11} \\ S_6 = B_{22} - S_5 \\ S_7 = B_{22} - B_{12} \\ S_8 = S_6 - B_{21} \end{array} \quad \begin{array}{l} M_1 = S_2 \times S_6 \\ M_2 = A_{11} \times B_{11} \\ M_3 = A_{12} \times B_{21} \\ M_4 = S_3 \times S_7 \\ M_5 = S_1 \times S_5 \\ M_6 = S_4 \times B_{22} \\ M_7 = A_{22} \times S_8 \end{array} \quad \begin{array}{l} V_1 = M_1 + M_2 \\ V_2 = V_1 + M_4 \\ V_3 = M_5 + M_6 \\ C_{11} = M_2 + M_3 \\ C_{12} = V_1 + V_3 \\ C_{21} = V_2 - M_7 \\ C_{22} = V_2 + M_5 \end{array}$$

Temporary行列

図 1 Strassen アルゴリズム

先行研究 [2] で取り上げられた部分行列の乗算の並列化と Strassen アルゴリズムの計算の途中で出てくる temporary 行列の個数にリソースが具体的にどのように関係するかを明らかにした。その結果を基にして先行研究の最速のプログラムを超える高速のプログラムを実現することを目的とした。

2. 関連研究

2.1 Strassen アルゴリズム

Strassen アルゴリズムは大きなサイズの行列の乗算を標準的な方法よりも高速に実行するアルゴリズムである [3]。\$N\$ が偶数である \$N \times N\$ 正方行列 \$A\$ と \$B\$ の乗算の場合は、行列 \$A\$ と \$B\$ をそれぞれ \$N/2 \times N/2\$ の 4 個の部分行列に分け、7 回の部分行列の乗算と 18 回の加減算を行うことにより積行列 \$C\$ を求める。元の \$N \times N\$ 行列の標準的な乗算は \$N/2 \times N/2\$ 部分行列の 8 回の乗算とそれらの積の 7 回の加算と等価なので、Strassen アルゴリズムは、乗算が 1 回少なく、加減算が 11 回多くなる。\$N/2 \times N/2\$ 行列の乗算の演算回数は \$(2N - 1)N^2/8\$ であり、加減算の演算回数は \$N^2/4\$ である。したがって行列のサイズ \$N\$ が大きくなると乗算の回数が少ない Strassen アルゴリズムによる計算は標準的な計算よりも演算回数が少なくなり、その結果高速になる。

Winograd は Strassen アルゴリズムを改良して加減算の回数を 18 から 15 に減らしたアルゴリズムを導いた [4]。このアルゴリズムは Strassen-Winograd アルゴリズムと呼ばれ実用性の高い Strassen 型のアルゴリズムとして、これまで多くの計算機を用いた研究で使われてきた。本論文では、このアルゴリズムを Strassen アルゴリズムと記し、その式を図 1 に示す。

図 1 で \$S_1 \sim S_8\$, \$M_1 \sim M_7\$, \$V_1 \sim V_3\$ は計算の途中で生じる temporary 行列である。これらの temporary 行列の要素はメモリに一時的に保存される必要がある。また \$M_1\$ から \$M_7\$ は 7 回の部分行列の乗算によって得られる。これらの部分行列の乗算の並列実行は Strassen アルゴリズムによる計算の高速化に大きく寄与する。部分行列への分割を一回だけ行って Strassen アルゴリズムを応用する方法を

1-level Strassen と呼ぶ。分割した部分行列の 7 回の乗算に再び Strassen アルゴリズムを応用し、これを何度も繰り返す方法を multi-level Strassen と呼ぶ。

2.2 Strassen アルゴリズムの CPU 計算

Strassen アルゴリズムの CPU による計算は 1987 年以後多くの研究で行われてきた。初期の CPU による研究は、ベクトルプロセッサから構成されるコンピュータを用いて Strassen アルゴリズムの逐次計算を行った。これらの研究では Strassen アルゴリズムで必要となる temporary 行列の数を減らす効果が調べられた [5]。Temporary 行列の数が多いとそれだけ多くの余分のメモリを使用することになる。その結果より多くのメモリを必要とする大きなサイズの行列の計算ができなくなる。一方 temporary 行列の数を減らすと使用するメモリは減るが、そのために数少ない temporary 行列を何度も使うことになる。その結果データのメモリからの出し入れの回数が増えて通信時間がより多くかかり計算速度は低下する。

1990 年代半ば以後現在までの CPU による大半の研究では Strassen アルゴリズムの並列計算のやり方が調べられた。Strassen アルゴリズムの部分行列の 7 個の乗算を使用する processor に分配して並列に実行する方法を breadth-first-step(BFS) と呼ぶ。この方法とは異なり、7 個の乗算をすべての processor を用いて逐次的に計算する方法を depth-first-step(DFS) と呼ぶ。BFS は多くのメモリを必要とする。使用する計算機のメモリが十分に備わっていない場合は並列計算は制限される。一方 DFS は少ないメモリの使用で済むが、データ転送の時間がより多くなる。multi-level Strassen の計算を行うに際して、部分行列のサイズが異なる各 level に BFS と DFS をどのように割り当てるのが使用する計算機にとって最適化か見出すことがこれらの研究の目的であった [6]。

2.3 Strassen アルゴリズムの GPU 計算

2011 年に Li ら [7] は最初の GPU を用いた Strassen アルゴリズムの計算を報告した。彼らは部分行列の乗算のために、自分たちで作成した行列の乗算のカーネルを用いている。使用した GPU は NVIDIA C1060 である。彼らの計算では、Strassen アルゴリズムを用いた単精度、4-level Strassen、行列サイズ \$16,384 \times 16,384\$ で CUBLAS-3.0 の標準的な行列の乗算に対して 1.36 倍の speedup を達成している。

2013 年に報告された Lai らの研究 [2] では、Strassen アルゴリズムの部分行列の計算に CUBLAS-5.0 を用いている。使用した GPU は NVIDIA C2050 および K10 である。彼らの Strassen アルゴリズムの計算は Li 等の計算および CUBLAS-5.0 を用いた標準的な乗算よりも高速になっている。CUBLAS-5.0 の標準的な乗算よりも K10 GPU を用い

Temporary行列 T1,T2

Step	Operation	Step	Operation
1	$T1 = A11 - A21$	12	$C12 = C12 + C22$
2	$T2 = B22 - B12$	13	$T1 = A11 \times B11$
3	$C21 = T1 \times T2$	14	$C11 = C11 + T1$
4	$T1 = A21 + A22$	15	$C12 = C12 + C11$
5	$T2 = B12 - B11$	16	$C11 = C11 + C21$
6	$C22 = T1 \times T2$	17	$T2 = T2 - B21$
7	$T1 = T1 - A11$	18	$C21 = A22 \times T2$
8	$T2 = B22 - T2$	19	$C21 = C11 - C21$
9	$C11 = T1 \times T2$	20	$C22 = C22 + C11$
10	$T1 = A12 - T1$	21	$C11 = A12 \times B21$
11	$C12 = T1 \times B22$	22	$C11 = T1 + C11$

図 2 先行研究の temporary 行列 2 個の逐次計算プログラムの演算スケジュール

temporary行列15個 (S1~S8, M1~M7)

Step 1	step2	step3
$S1 = A21 + A22$	$M1 = S2 \times S6$	$S1 = M1 + M2$
$S2 = S1 - A11$	$M2 = A11 \times B11$	$C11 = M2 + M3$
$S3 = A11 - A21$	$M3 = A12 \times B21$	$C12 = M5 + M6$
$S4 = A12 - S2$	$M4 = S3 \times S7$	$S2 = S1 + M4$
$S5 = B12 - B11$	$M5 = S1 \times S5$	$C12 = S1 + C12$
$S6 = B22 - S5$	$M6 = S4 \times B22$	$C21 = S2 - M7$
$S7 = B22 - B12$	$M7 = A22 \times S8$	$C22 = S2 + M5$
$S8 = S6 - B21$		

図 3 temporary 行列 15 個の並列計算プログラムの演算スケジュール

た単精度 3-level Strassen, 正方向列のサイズ 15,360 で 1.27 倍の speedup, 倍精度 4-level Strassen, 正方向列のサイズ 8,192 で 1.42 倍の speedup を達成している。

上記の speedup を達成したプログラムは temporary 行列を 2 個だけ使用した逐次実行プログラムである。この逐次プログラムの演算スケジュールは CPU による Strassen アルゴリズム計算で用いられた演算スケジュールである [8]。それを図 2 で示す。

Lai 等は、上記の逐次プログラムの他に、部分行列の乗算と加減算を並列にしたプログラムを提案している。この並列プログラムでは合計 15 個の temporary 行列を含んでいる。この並列プログラムを表した演算スケジュールを図 3 で示す。彼らの計算実験によれば、並列計算プログラムの逐次計算プログラムに対する speedup は行列サイズ 2,048 以下の範囲で見られ、行列サイズ 32 から 256 の範囲で 1.68 倍から 1.15 倍へと急激に減少している。彼らはこの speedup の減少を GPU の workload が飽和したためとしているが、その理由の具体的な説明をしていない。

2.4 解決すべき課題

本研究では以下の課題の解決を目指した。

(i) Strassen アルゴリズムの計算を高速化する最も有力な方法は部分行列の乗算の並列化である。しかし先行研究では並列化の効果は Strassen アルゴリズムが標準的な行列の乗算よりも遅い小さな行列のサイズでのみ現れた。本研究では並列化の効果が行列のサイズの増加とともに急速に減

GPU	Tesla K20	Tesla P100	Tesla V100
SMs	13	56	80
Peak FP64 TFLOPS	1.7	5.3	7.8
Memory Size	5 GB	16 GB	16GB
GMBW	208 GB/s	720 GB/s	900 GB/s

表 1 実験環境

少する理由を具体的に明らかにする。

(ii) 先行研究 [2] では temporary 行列を 2 個使用した逐次計算プログラムで最も高速な結果を得ている。しかし temporary 行列 2 個がなぜ最適であるのか論文で説明されていない。temporary 行列の個数の影響については GPU の場合は CPU の場合と異なる可能性がある。この違いを明らかにするのが本研究の次の課題である。

(iii) GPU の基本的なリソース要因がどのように Strassen アルゴリズムの計算速度に影響するのか理論的に明らかにするのが望ましい。そのことが明らかになれば、様々な世代の GPU に適用できる Strassen アルゴリズムの計算のプログラムの最適化法を見出すことが可能になる。この課題の解決のために GPU による Strassen アルゴリズムの計算の speedup の予測式を導出する。

3. 対象とする行列と評価環境

本研究で GPU を用いた行列の Strassen アルゴリズムによる計算と標準的な行列の乗算に、次のような方法を統一して用いた。

(i) 偶数 N の $N \times N$ 正方向列のみを対象とした。(ii) 行列の標準的な乗算と Strassen アルゴリズムの部分行列の乗算に CUBLAS ライブラリの最新 version である CUBLAS-10.1 を用いた。(iii) 倍精度計算のみを対象とした。(iv) 行列の標準的な乗算と Strassen アルゴリズムによる乗算の処理時間の比較に両方に共通する処理は除いた。それらは初期行列 A と B および積行列 C のためのメモリ確保と解放、そしてメモリ A と B へのホスト CPU からの初期データの移動とメモリ C からホスト CPU への移動である。

使用した GPU は NVIDIA K20GPU, P100GPU, および V100GPU である。表 1 にこれら 3 種類の GPU の本研究に関わる特性を記した [9,10,11]。表中 GMBW は global memory band 幅である。

4. 部分行列乗算の並列数の最適化

GPU による Strassen アルゴリズムの部分行列の乗算の並列化が制限される原因を解明するために NVIDIA visual profiler をを使って解析した。その結果並列化した乗算を実行するのにレジスタ容量が不足することが原因であることが明らかになった。つまり CUBLAS ライブラリでは乗算をする部分行列のサイズごとに、使用するスレッド数 n_{th} と各スレッドのレジスタ容量 r/th が決まり、それらが visual profiler に表示される。一方 GPU には streaming

N	n_{th}	r/th	n_{th}^{SM}	n_{SM}	P_{theo}	P_{exp}
1,024	16,384	74	3,459	4.7	16	7
2,048	32,768	242	1,058	31	2.6	3
4,096	65,536	242	1,058	62	1.3	2
6,144	147,456	242	1,058	139	0.58	1
8,192	262,144	242	1,094	240	0.33	1
12,288	589,824	242	1,094	240	0.15	1

表 2 V100:stream 並列数の予測値と実測値

multi-processor(SM)あたりの最大レジスタ容量 r_{max} が定められている。その値は V100, P100, K20 に共通で 256 kB である [9,10,11]。したがって r_{max} を r/th で割ることにより、一つの SM に存在するスレッド数 n_{th}^{SM} が求められる。この n_{th}^{SM} で使用するスレッド数 n_{th} を割ると部分行列の一回の乗算で使用する SM 数 n_{SM} が求められる。

GPU が有する SM 数 n_{SM}^{max} は各世代の GPU で決まっている。その値は V100, P100, K20 でそれぞれ 80, 56, 13 である。 n_{SM}^{max} を n_{SM} で割った値は並列に実行できる部分行列の乗算の数すなわち予測並列数 P_{theo} になる。本研究で使った V100 についてこれらの値と visual profiler に表示された実際の並列数 P_{exp} を表 2 に示した。この表で P_{theo} が 1.0 を下回る場合は逐次計算、すなわち並列数が 1 であることを意味する。また Strassen アルゴリズムの部分行列の乗算の数は 7 個であるので、 P_{theo} が 7 個を超えても P_{exp} は 7 個になる。

表 2 に見られるように実測並列数 P_{exp} は予測並列数 P_{theo} と一致している。P100 と K20 についても同様の比較を行ったところ、予測並列数と実測並列数は一致あるいは非常に近い値となった。このことは GPU による Strassen アルゴリズムの部分行列の乗算の並列化が制限される原因がレジスタ容量の不足によることを示している。

5. temporary 行列の削除

5.1 temporary 行列を削除した逐次実行 1-level Strassen アルゴリズム

先行研究で使った temporary 行列を 2 個含んだ演算スケジュールは積行列の部分行列 $C_{11}, C_{12}, C_{21}, C_{22}$ を temporary 行列の代わりに用いている。それらに加えて本研究では temporary 行列の個数を 0 にするために初期行列 A と B の部分行列を再利用する逐次計算演算スケジュールを用いた。これらの部分行列のためのメモリの確保と解放は Strassen アルゴリズムによる計算と標準的な乗算の両方で共通して行われるので、Strassen アルゴリズムによる計算の高速化のためのオーバーヘッドにならない。このようにして temporary 行列を削除した演算スケジュールを図 4 に示す。図 4 の演算スケジュールを導くに際してデータの引き継ぎが正しく行われることを確認するために、Strassen アルゴリズムによる計算の結果と標準的な乗算の結果に大きな違いがないか調べた。

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

図 4 temporary 行列を削除した逐次計算プログラムの演算スケジュール

Step	Operation	Stream	Step	Operation	Stream
1	$C_{22} = A_{11} - A_{21}$		10	$C_{12} = C_{22} + A_{21}$	
2	$C_{11} = B_{22} - B_{12}$		11	$B_{22} = A_{11} \times B_{11}$	
3	$A_{21} = A_{21} + A_{22}$		12	$C_{11} = B_{22} + C_{11}$	
4	$C_{12} = B_{12} - B_{11}$		13	$C_{12} = C_{11} + C_{12}$	
5	$C_{21} = C_{22} \times C_{11}$	0	14	$C_{11} = C_{11} + C_{21}$	
	$C_{22} = A_{21} \times C_{12}$	1	15	$B_{11} = B_{12} - B_{21}$	
6	$A_{21} = A_{21} - A_{11}$		16	$C_{21} = A_{22} \times B_{11}$	0
7	$B_{12} = B_{22} - C_{11}$			$A_{11} = A_{12} \times B_{21}$	1
8	$C_{12} = A_{21} \times B_{12}$		17	$C_{21} = C_{11} - C_{21}$	
9	$C_{11} = A_{21} \times B_{12}$	0	18	$C_{22} = C_{11} + C_{22}$	
	$A_{21} = C_{12} \times B_{22}$	1	19	$C_{11} = B_{22} + A_{11}$	

図 5 temporary 行列を削除した stream 並列実行プログラムの演算スケジュール

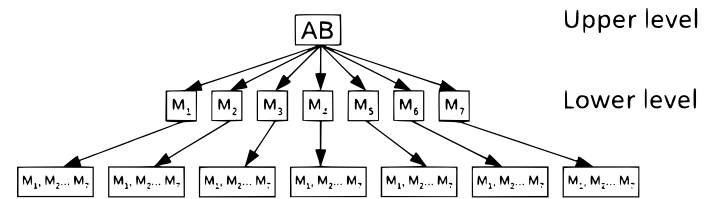


図 6 2-level Strassen アルゴリズムの模式図

5.2 temporary 行列を削除した並列実行 1-level Strassen アルゴリズム

図 5 に三世代の GPU V100, P100, K20 について並列化による計算の高速化を調べるために作成した並列プログラムの演算スケジュールを示す。部分行列の乗算の並列化が GPU の各 SM が保有できるレジスタ容量によって制限を受け、その結果並列数は GPU が有する SM 数に大きく依存する。このプログラムでは temporary 行列の数は 0 であり、部分行列の乗算 7 個の内、6 個を 2 個ずつ並列にしている。

5.3 temporary 行列を削除した 2-level Strassen アルゴリズム

図 6 に temporary 行列を削除した 2-level Strassen を模式的に示す。1-level Strassen は V100 を使用した場合行列サイズが 5,200 以上で speedup は 1.0 以上になる。一方行列サイズが 15,000 以上になると global memory の容量不足のためにプログラムの実行不可能となる。したがって本研究の実験環境では、multi-level Strassen は二つの level だけを含むことになる。

図 6 で二つの level を元の行列サイズの upper level と、それを半分のサイズの部分行列に分割した lower level を示し

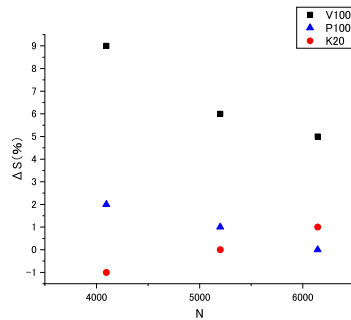


図 7 stream 並列数による speedup の行列サイズ依存性 (V100, P100, K20)

た. 本研究では lower level の計算に temporary 行列を削除し部分行列の乗算を 2 個ずつ並列にした演算スケジュールを用いた. 理由は行列の小さいサイズで temporary 行列の影響と並列化の効果が大きくなるからである. temporary 行列の代わりに初期行列の A と B の部分行列を使うと lower level の計算の後初期データが上書きされる. また積行列 C の部分行列も使われるので lower level での部分行列の一つ乗算の結果が次の乗算で上書きされる可能性がある. これらの上書きによって誤った結果が生じないように upper level の演算スケジュールを作成しなければならない. 様々な演算スケジュールを検討した結果 temporary 行列を最低 7 個含めなければならないと結論された. また lower level の演算スケジュールで temporary 行列の代わりに再利用される初期行列の部分行列の個数をできる限り少なくした.

6. 評価

4,000 以上の行列サイズの範囲で部分行列の乗算の並列化の効果を V100, P100, K20 の三世代の GPU について計算実験により調べた. 図 7 は三世代の GPU の並列プログラムの逐次実行プログラムに対する speedup の変化率を行列サイズ 4,096, 5,120 および 6,144 について示している. 計算では CUBLAS-10.1 を使用した. SM の最も多い V100 でのみ並列化による speedup がこの行列範囲で観測され, そして行列サイズの増加とともに speedup の効果は減少している.

図 8 に temporary 行列を二個含む先行研究 [2] の逐次実行プログラムを V100 で走らせた時のメモリの確保と解放の処理時間の全計算時間に対する割合を示す. 図はメモリの確保と解放が小さなサイズの行列の計算で無視できないオーバーヘッドになることを示唆している.

図 9 に temporary 行列を削除した逐次計算と並列計算プログラムの計算速度を 2 個の temporary 行列を含む先行研究 [2] のプログラムの計算速度と V100 GPU を用いて比較した結果を示す. 計算速度は行列の標準的な乗算に対する speedup で表した. これらの計算すべてで CUBLAS-10.1

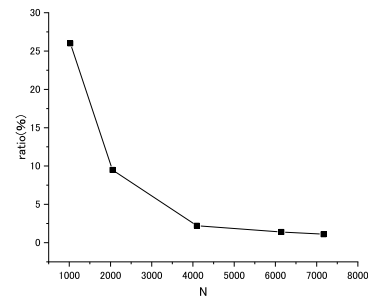


図 8 メモリの確保と解放の処理時間の全計算時間に対する割合

を使用した.

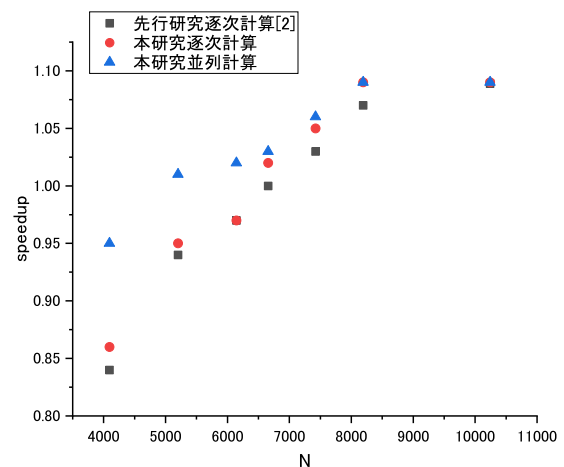


図 9 temporary 行列を削除した逐次計算と並列計算プログラムの先行研究プログラムに対する speedup(V100)

図 9 に見られるように行列サイズが 4,000 から 8,000 の範囲で temporary 変数が 0 個の逐次計算プログラムは 2 個の逐次計算プログラムより 1 % から 2 % 高速になっている. この高速化は temporary 行列のためのメモリの確保と解放の処理時間を除いたことによる. すなわち Strassen アルゴリズムの計算が temporary 行列の個数を減らすことにより高速化する.

図 9 は部分行列の乗算の並列化によってさらに高速化することを示している. 並列化の結果行列サイズが 5,200 で先行研究のプログラムよりも 7 % 高速なり, 行列サイズ 8,192 まで並列化の効果は次第に減少している. temporary 行列の削除と並列化により高速化したプログラムは行列サイズ 5,200 で標準的な乗算と同じ計算速度になり, それ以上の行列サイズで標準的な乗算よりも高速になっている.

表 3 に本研究で作成した 2-level Strassen アルゴリズムのプログラムの speedup と先行研究の逐次算プログラムを 2-level 繰り返した場合の speedup を示した. 本研究のプログラムの temporary 行列の個数は upper level で 7, lower level で 0 であるのに対して, 先行研究の場合は両方のプロ

行列サイズ N	先行研究 [2] speedup	本研究 speedup
12,288	1.06	1.10
13,312	1.10	1.11
14,336	1.117	1.124

表 3 先行研究と本研究の 2-level Strassen プログラムの speedup の比較

グラムでそれぞれ 2 個である。表 3 の結果は行列サイズ 12,288 で speedup の差は大きく、行列サイズの増加とともに差が減少することを示している。

7. 考察

4 節で部分行列の乗算の並列化の制限が GPU ではレジスタ容量の不足によって起きることを明らかにした。一方 CPU による Strassen アルゴリズムの計算の場合は計算機の保有するメモリの不足によって起きる [6]。この GPU と CPU の違いの原因は以下のように説明される。行列の乗算を GPU で実行する場合、一つのスレッドは積行列 C の一つの要素を $(2N - 1)$ 回の演算で求める。行列のサイズが大きくなると一つのスレッドが行う演算回数は多くなるので、その分だけ多くのレジスタ容量が必要となる。表 2 で見られるようにスレッドあたりのレジスタ容量 r/th は各 GPU のリソースが許容する r/th の最大値 255 B に近い値となっている。そして積行列の要素も多くなるので必要となるスレッド数も多くなり、並列実行を行うとレジスタ容量の不足が起きる。この現象は GPU アーキテクチャの特徴、すなわちできる限り大量の算術演算をメモリとのデータ転送とオーバーラップさせることで高い FLOPS を実現するように設計されていることによる。

Strassen アルゴリズムの CPU による計算では、temporary 行列の個数が少なくなるとメモリとの間でのデータの出し入れの回数が増えるので計算速度は低下する [5]。本研究で調べた GPU による計算の場合は CPU の場合と逆に temporary 行列の個数を減らすと計算速度は高くなった。この違いはアーキテクチャに起因する GPU の特性によって説明できる。GPU は高い計算処理能力を実現するために多量の算術計算を global memory とのデータの転送と並列に実行することによりデータ転送の処理時間を隠すようにしている。しかし cudaMalloc と cudaFree 関数によるメモリの確保と解放の場合は global memory との通信は算術演算に対して逐次処理される [12]。その結果 Strassen アルゴリズムの GPU による計算ではメモリの確保と解放は無視できないオーバーヘッドになる。

8. Strassen アルゴリズムの speedup の予測

GPU の基本的なリソース要因がどのように Strassen アルゴリズムの計算速度に影響するのか理論的に明らかにするのが望ましい。そのことが明らかになれば、様々な世代の GPU に共通に適用できる Strassen アルゴリズム計算の

プログラムの最適化法を見出すことが可能になる。

8.1 予測の方針

CUBLAS-10.1 による行列の乗算と加算の行列サイズ依存性を visual profiler を使って調べた。その結果行列サイズが 2,000 以上で、乗算の場合は処理時間が N^3 に比例し、加算の場合は N^2 に比例していた。行列の乗算の場合は積行列の一つの要素 c_{ij} は

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{iN}b_{Nj} \quad (1)$$

で計算されるので、 $2N - 1$ 回の算術演算によって得られる。一方行列の加算の場合の和行列の一つの要素は

$$c_{ij} = a_{ij} + b_{ij} \quad (2)$$

で与えられるので算術演算は 1 回である。積行列と和行列の要素数はそれぞれ N^2 であるので、積行列と和行列の各要素は行列サイズ 2,000 以上では逐次計算されていると考えられる。

行列のサイズが 2,000 以上の範囲に適用できる Strassen アルゴリズムの計算時間の理論式を以下のように仮定した。

$$F_{mul}(N) = (1/F)(2N - 1)N^2 \quad (3)$$

ここで F は peakFP64(FLOPS) である。大きなサイズの行列の乗算の場合はデータは global memory から shared memory に移されそこで何度も再利用されるので、global memory とのデータ移動の時間は式 (3) に現れない。行列の加減算の計算時間は次式で与えられると仮定する。

$$F_{add}(N) = (3 \times 8/G)N^2 \quad (4)$$

ここで G は global memory band 幅である。8 は倍精度の byte 数、3 は一つの要素の計算のための global memory とのデータ移動回数である。大きなサイズの行列の加減算の場合は一つの要素の計算時間は global memory とのデータ移動に要する時間で実質的に決まると仮定する。

式 (3) と (4) を用いると 1-level Strassen アルゴリズムの逐次計算の時間は

$$t_{Strassen}(N) = 7F_{mul}(N/2) + 15F_{add}(N/2) + n_{tempo}(g_{malloc}(N/2) + g_{free}(N/2)) \quad (5)$$

となる。ここで n_{tempo} は temporary 行列の数であり、 $g_{malloc}(N/2)$ と $g_{free}(N/2)$ はサイズ $N/2$ の temporary 行列 1 個のためのメモリ確保と解放に要する時間である。これらの時間の理論式はないので、visual profiler によって求めた実測値が使われる。

CUBLAS による行列の標準的な乗算の計算時間は

$$t_{standard}(N) = F_{mul}(N) \quad (6)$$

である。したがって Strassen アルゴリズムの計算の speedup

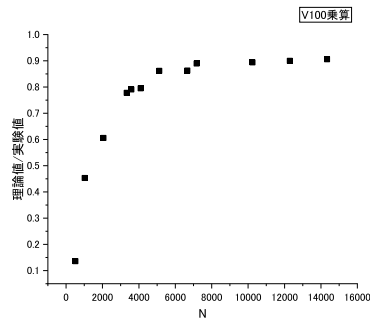


図 10 行列乗算の計算時間の理論値と実験値の比較 (V100)

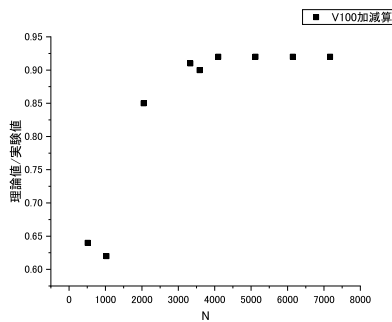


図 11 行列加減算の計算時間の理論値と実験値の比較 (V100)

S は

$$S = t_{standard}(N)/t_{Strassen}(N) \quad (7)$$

で計算される。

8.2 実測値との比較

前節で導いた Strassen アルゴリズムの speedup の予測式の有効性を検証するために V100 について行列の乗算と加減算の計算時間の理論値と実験値を比較した。図 10 と図 11 にそれぞれの理論値/実験値の比の行列サイズ依存性を示した。理論値の計算で F と G の値は表 1 にある値を用いた。乗算の場合は行列サイズ 7,000 以上で比は一定で 0.90 である。加減算の場合は行列サイズ 4,000 以上で比は一定で 0.92 である。このことは両方の場合とも大きな行列のサイズで理論式は実験値をほぼ正確に表していることを意味している。比が 0.90 が 0.92 と 1.0 より少し低いのは F と G の実際の値が公表されている値よりも低いことによると考えられる。行列のサイズが小さくなるにつれて比は徐々に減少している。この減少は乗算の場合は N^3 比例関係から、加減算の場合は N^2 比例関係からのズレによると考えられる。

図 12,13,14 に temporary 行列の数 n_{tempo} が 0 の場合の speedup S の予測値と実測値を三世代の GPU である V100, P100 と K20 について比較した。いずれの場合も大きなサイズの行列で予測値は実測値とよく一致している。行列のサイズの小さいところではズレは大きいですが、行列サイズへ

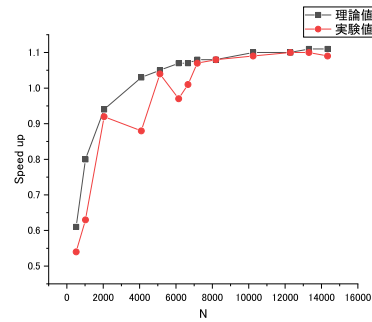


図 12 Strassen アルゴリズムによる speedup の理論値と実験値の比較 (V100)

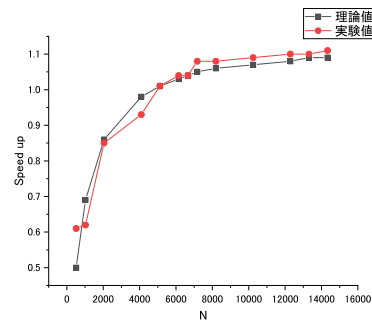


図 13 Strassen アルゴリズムによる speedup の理論値と実験値の比較 (P100)

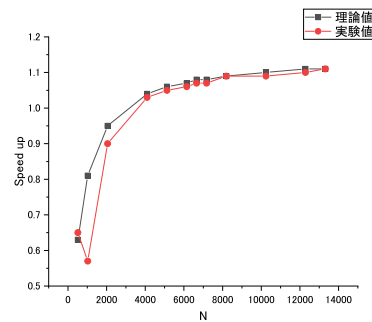


図 14 Strassen アルゴリズムによる speedup の理論値と実験値の比較 (K20)

の依存性の傾向は一致している。V100 の場合は他の GPU に比べてズレが大きいですが、その原因は V100 に新しく導入された個別のスレッドスケジューリング機能と関係していると考えられる。

speedup の予測式の重要な役割は、Strassen アルゴリズムの計算の高速化に GPU のリソースがどのように影響するか明瞭に示すことである。Strassen アルゴリズムの計算時間の理論式 (5) は、speedup の値にまず peakFP64(FLOPS) F と global memory band 幅 G が競合する形で影響することを示している。そしてメモリの確保と解放の処理時間が付加的に影響することを示している。式 (5) はさらに部分行列の乗算の並列化が Strassen アルゴリズムの高速化に大きく寄与することを示唆している。なぜならば式 (5) の

右辺の第一項の係数7が並列化によってより小さな数に変わるからである。このことから6節に示した結果とともに今後SMの数がV100より大きく増えたGPUが可能になればStrassenアルゴリズムによる行列の乗算は大きく高速化することが期待される。

9. 結論

GPUを用いたStrassenアルゴリズムによる行列の乗算の高速化を行った。Strassenアルゴリズムは行列の乗算を高速化する方法として知られているが、GPUに適用するためには計算の途中に出てくるtemporary行列のためのメモリの確保と解放に余分な時間を要すること、および部分行列の乗算の並列実行数がGPUリソースによって制限される問題がある。これらの問題を解決するために本研究ではtemporary行列を削除し、そして部分行列の乗算を2個並列実行するプログラムを作成した。本研究のプログラムはこれまで報告されてきたStrassenアルゴリズムのGPUによる計算の先行研究で最速のプログラムよりもすべての行列のサイズで高速となった。V100を用いた行列サイズ $5,200 \times 5,200$ の計算で先行研究のプログラムより7%高速になった。また本研究の2-level StrassenアルゴリズムのプログラムはCUBLAS-10.1を用いた行列の標準的な乗算よりも行列サイズ $14,336 \times 14,336$ で12%高速になった。逐次計算1-level Strassenアルゴリズムのspeedupの予測式を導いた。予測式は三代目のGPUによる実測値を正確に再現した。

参考文献

- [1] J. Lobeiras, et al. Designing efficient index-digi algorithm for CUDA GPU architecture, IEEE Trans. Parallel Distrib. Syst., vol. 27, no. 10, pp. 1331-1343, 2016.
- [2] Pai-Wei Lai, Humayun Arafat, Venmugil Elango and P. Sadayappan, Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs, 2013 International Conference on Computer, Control, Informatics and Its Applications (IC3INA), pp. 139-148, 2013.
- [3] V. Strassen. Gaussian elimination is not optimal. Numer. Math., 13:354-356, 1969.
- [4] S. Winograd. On multiplication of 2×2 matrices. Linear Algebra and Application, 4:381-388, 1971.
- [5] D. H. Bailey, K. Lee and H. D. Simon, Using Strassen's Algorithm to Accelerate the Solution of Linear Systems, J. Supercomputing, 4, 357-371, 1990.
- [6] G. Ballard, et al., Communication-optimal parallel algorithm for Strassen's matrix multiplication, In Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, 193-204, ACM, 2012.
- [7] J. Li, S. Ranka and S. Sahni, Strassen's matrix multiplication on GPUs, In Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS '11, 157-164, Washington DC, USA, 2011. IEEE Computer Society.
- [8] C. C. Douglas, et al., GEMMW: A portable level 3 BLAS winograd variant of strassen's matrix multiply algorithm, J. Comput. Phys., 110(1): 1-10, Jan. 1994.
- [9] Nvidia Corp, NVIDIA Tesla K20 (Whitepaper), 2012.
- [10] Nvidia Corp, NVIDIA Tesla P100 (Whitepaper), 2016.
- [11] Nvidia corp, NVIDIA Tesla V100 (Whitepaper), 2017.
- [12] X. Huang, et al., XMallo: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines, In Computer and Information Technology(CIT), 2010 IEEE, 1134-1139, July 2010.