JAIST Repository

https://dspace.jaist.ac.jp/

Title	GPU向きStrassenアルゴリズムの最適化		
Author(s)	大塚,達史		
Citation			
Issue Date	2020-03		
Туре	Thesis or Dissertation		
Text version	author		
URL	http://hdl.handle.net/10119/16385		
Rights			
Description	Supervisor: 井口 寧, 先端科学技術研究科, 修士 (情報科学)		



Japan Advanced Institute of Science and Technology

修士論文

GPU 向き Strassen アルゴリズムの最適化

1810032 大	、塚 達史
-----------	-------

主指導教員	井口 寧 教授
審査委員主査	井口 寧 教授
審査委員	金子 峰雄 教授
	田中 清史 准教授
	本郷 研太 准教授

北陸先端科学技術大学院大学 (情報科学)

2020年2月

先端科学技術研究科

Abstract

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. 高速 化を行う上で二つの問題があること見出した. 第一の問題は部分行列の乗算の並 列実行数が行列サイズが増加するにつれて減少してしまうことである. これまで のGPUによるStrassenアルゴリズムの研究は計算途中の結果を保存するための Temporary 行列を使用していた. 第二の問題はGPUではTemporary 行列を使用 するためのメモリ確保と解放の時間を要することである. 本研究では第一の問題に ついてはGPUリソースの影響を明らかにすることにより行列サイズごとの最適な 並列実行数を決定した. 第二の問題はTemporary 行列を削除した演算スケジュー リングを作成することでメモリ確保と解放の時間を削減することができた. 本研 究は NVIDIA GPU Tesla V100, Tesla P100 および Tesla K20 を使用し, 部分行列 の乗算に CUBLAS-10. 1を用いて性能比較を行った. その結果本研究のプログラ ムは Strassen アルゴリズムの GPU による計算の研究で最速の計算プログラムよ りもすべての行列のサイズで高速となった. GPU V100を用いた行数 4096 × 4096 の計算で先行研究のプログラムより 11 %高速になった.

Abstract

We made acceleration of matrix multiplication by the Strassen algorithm on GPU. In achieving the acceleration we found two problems. The first problem is a decrease in the number of parallel executions of multiplications of submatrices. Earlier studies on the Strassen algorithm with GPUs employed temporary matrices for storing intermediate results. The second problem is the time spent by allocation and deallocation of memories for these temporary matrices. For overcoming the first problem we determined the optimal number of parallel executions of submatrices multiplications by clarifying the influence of GPU resources. For overcoming the second problem we reduced the time for the memory allocation and deallocation by making an operation schedule without a temporary matrix. NVIDIA GPU Tesla V100, Tesla P100and Tesla K20 were used, and the performance evaluation was made by using CUBLAS library 10. 1. For all matrix sizes the program made in the present study is faster than programs made in earlier studies on the Strassen algorithm with GPUs. For multiplication of 4096×4096 matrices our program is faster than the fastest program in earlier studies by 11 %.

目 次

第1章	序論 1
1.1	研究背景 1
1.2	研究目的
1.3	本研究の貢献 2
第2章	関連研究 3
2.1	はじめに
2.2	GPUのアーキテクチャーと演算性能 3
	2.2.1 GPUのアーキテクチャ 3
	2.2.2 GPUによる高速化のアプローチ
	2.2.3 GPUstream
	2.2.4 GPU 数値計算ライブラリ
2.3	Strassen アルゴリズム
	2.3.1 Strassen アルゴリズムの概要 7
	2.3.2 Strassen アルゴリズムの CPU 計算
	2.3.3 Strassen アルゴリズムの GPU 計算 10
2.4	解決すべき課題
2.5	まとめ
第3章	評価環境の構築 19
3.1	はじめに
3.2	実験および解析方法 19
3.3	実験環境
3.4	まとめ
第4章	部分行列乗算の並列数の最適化 21
4.1	はじめに
4.2	stream 並列数の GPU リソースによる制限 21
4.3	提案
4.4	評価
	4.4.1 stream 並列数の予測 23
	4.4.2 並列化よる高速化 23
4.5	考察

4.6	まとめ	26
第5章	Temporary 行列の削除	28
5.1	はじめに	28
5.2	Temporary 行列に関する処理時間の割合	28
5.3	提案	29
	5.3.1 Temporary 行列を削除した 1-level Stassen アルゴリズム	29
	5.3.2 Temporary 行列を削除した 2-level Stassen アルゴリズム	30
5.4	評価	33
5.5	考察	35
5.6	まとめ	36
第6章	Strassen アルゴリズムの speedup 予測式	37
6.1	はじめに	37
6.2	visual profiler によるデータ解析	37
6.3	予測式の作成	38
6.4	実測値との比較	39
6.5	考察	40
6.6	まとめ	42
第7章	結論	44

図目次

2.1	GPU アーキテクチャ簡略図	4
2.2	GPUstream	6
2.3	OpenBLAS と CUBLAS の処理時間の比較	7
2.4	Strassen-Winograd アルゴリズムの並列計算の模式図	16
4.1	行数の増加に伴う部分行列の乗算の stream 並列数の変化	22
4.2	stream 並列化による speedup の行数依存性(V100)	25
4.3	stream 並列化による speedup の行数依存性(P100)	27
5.1	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 (V100)	29
5.2	2-level Strassen アルゴリズムの模式図	32
5.3	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)	35
6.1	行列の計算時間の行数依存性の log-log plot (V100)	38
6.2	メモリの確保と解放の処理時間の行数依存性.........	39
6.3	行列乗算の計算時間の理論値と実験値の比較(V100)	40
6.4	行列加減算の計算時間の理論値と実験値の比較(V100)	41
6.5	Strassen アルゴリズムによる speedup の理論値と実験値の比較(V100)	
66	41 Straggon アルゴリブムによる gnood up の理論値と実験値の比較(D100)	
0.0	42	
6.7	Strassen アルゴリズムによる speedupの理論値と実験値の比較(K20)	
	43	

表目次

2.1	Algorithm2,3,4,5 の変数と関数	14
2.2	先行研究の Temporary 行列 2 個の serial 計算プログラムの演算スケ	
	ジュール	15
2.3	Temporary 行列 15 個の並列計算プログラムの演算スケジュール	17
3.1	実験環境	20
4.1	Algorithm6 の変数	23
4.2	V100:stream 並列数の予測値と実測値	24
4.3	P100:stream 並列数の予測値と実測値	24
4.4	K20:stream 並列数の予測値と実測値	25
4.5	Temporary 行列を2個含む stream 並列実行プログラムの演算スケ	
	ジュール	26
5.1	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	30
5.2	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	31
5.3	2-level Strassen プログラムの upper level の演算スケジュール	33
5.4	2-level Strassen プログラムの lower level の演算スケジュール	34
5.5	先行研究と本研究の 2-level Strassen プログラムの speedup の比較 .	36

第1章 序論

1.1 研究背景

GPUはその高い並列処理能力により近年多くの分野で広く活用されている.し かしGPU用のプログラム作成において,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.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. Lobeiras ら [1] は, 分割統治アルゴリズム型の計算問題にGPU上のレジスタなどのリソース容量があ たえる影響を解析し,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 るのか十分に明らかにされていない.そのためアーキテクチャーが進歩し続けてい る GPU で今後 Strassen アルゴリズムの計算がどのような仕方で高速化するか不 明な状況である.

1.2 研究目的

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する.特に先行研究 [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のかを明らかにする.その結果を基にして先行研究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する.

上述の目的を達成するために行った本研究の結果を4章,5章および6章で報告 する.部分行列の乗算の並列化に対してのGPUリソースの影響の仕方の解明と並 列数の最適化について4章で報告する.5章でTemporary行列の個数の計算時間 への影響を調べた結果と,その結果に基づいて作成したTemporary行列を削除し たプログラム説明する.5章ではさらに計算の高速化について本研究のプログラム と先行研究[2]のプログラムを比較した結果を報告する.6章では異なった世代の GPUに共通に適用できるStrassenアルゴリズムの行列の標準的な乗算に対する Speedupの予測式について説明し,その予測式の値と実測値を比較した結果を報告 する.

1.3 本研究の貢献

本研究の貢献は以下のように要約される.本研究で開発したプログラムはこれ まで報告されてきた Strassen アルゴリズムの GPU による計算の研究で最速の逐 次計算プログラム [2] よりもすべての行列のサイズで高速となった. Strassen アル ゴリズムの計算の高速化に CPU による Strassen アルゴリズムの計算の場合とは 異なる GPU 特有のリソースの影響の仕方を初めて明らかにした.部分行列の並列 化を制限するリソース要因は, GPU の場合は SM あたりの register 容量と GPU の SM の個数である.また Temporary 行列の個数を減らすと GPU の場合は計算速 度は高くなる. Strassen アルゴリズムによる行列の乗算の標準的な方法に対する speedup の予測式を導いた.予測式に基づき 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明らかにした.

第2章 関連研究

2.1 はじめに

この章では初めに本研究に関係する GPU のアーキテクチャーの特徴と演算 性能を説明する.その次に本研究の主要なテーマである Strassen アルゴリズムに ついて説明する.その後でこれまでの Strassen アルゴリズムの CPU による研究と GPU による研究を説明する.最後にこれらの研究を踏まえて本研究で解決すべき 課題を説明する.

2.2 GPUのアーキテクチャーと演算性能

本研究では Strassen アルゴリズムによる行列の乗算の高速化に GPU のリソー スが影響する仕方を解析し, その上でリソースの影響を系統的にプログラムに組み 込みこむことで計算を高速化することを目指した. 2. 2節では GPU のリソースと は何かを明らかにするために GPU の基本的なアーキテクチャーと演算性能を説明 する.

2.2.1 GPUのアーキテクチャ

この節では、本研究の対象である行列の乗算に密接に関係する GPU のアーキテ クチャーを説明する.

GPUはCPUに比べてはるかに高速の計算能力を有している. 2016 年の時点 でNVIDIA GPUのFLOPS(floating point operations per second)はIntel CPUの FLOPSの約10倍となっている[3]. この計算能力の違いは,GPUでは圧倒的に多 くのトランジスターを算術演算装置(ALU)に割あてていることによる.CPUでは ALUに加えて制御機構(Control)とキャッシュ(Cache)に多くのトランジスター を割り当てている.その結果GPUでは,メモリとのデータ転送による遅延が表に 現れないように効率的に多くの算術演算を実行でき高いFLOPSが可能になる.大 きなサイズの行列の乗算ではメモリからの初期データの転送に比べて非常に多く の算術演算を行う.

図 2.1 に示したように GPU は複数の Streaming multi-processor (SM) から構成されている. SM の数は各世代の GPU で異なり, 例えば GPU V100 は 80 個の SM



図 2.1: GPU アーキテクチャ簡略図

によって構成されている. それぞれの SM は複数個の core からなっている. CUDA プログラムがカーネルを呼び出すと一つの SM に数十から数百の thread が演算を 実行するために生成される. 一つのカーネルによって生成される thread 全体は grid を構成し, grid は thread のグループからなる複数の block に分けられる. さらに一 つの block 中の thread は 32 個が単位の warp と呼ばれるグループに分けられる.

GPUの主要なメモリは3種類ある. それらはGPU中の全ての thread からア クセスできる global memory, 各 SM の中に存在して一つの block 中の thread が共 有できる shared memory, そして各 thread に個別に割り当てられる register である. 容量は global memory, shared memory, register の順に小さくなり, 一方アクセス 速度はこの順で速くなる. これらの三種類のメモリの他に補助的なメモリとして L1 と L2 のキャッシュとコンスタントメモリがある.

2.2.2 GPUによる高速化のアプローチ

GPUで多数のthreadを用いて演算を行うが、その方法はSingle Instruction Multiple Thread(SIMT) 実行スタイルと呼ばれている [1]. これはベクトル processor な どの CPU で行われる Single Instruction Multiple Data(SIMD) 実行スタイルと似 ているが、重要な違いがある. SIMD では各 thread が多数のデータを並列に演算処 理する. SIMT の場合はそれに加えて block や warp のグループ化を通して多くの thread が組織的に演算を実行する. この組織的な実行は concurrent 実行と呼ばれ る. warp 内の thread は concurrent に動作し、そして block 内の warp は concurrent に動作する. concurrent は parallel (並列)とは異なる. Parallel は同時に複数の thread, あるいは warp が動作することである. 一方 concurrent はデバイスのリソー スが許す条件の下で,ある時間枠の中でできる限り時間をかけないように複数の thread や warp が動作する. Parallel 動作は concurrent 動作の一種類であるが全て ではない. 例えばメモリからのデータの転送が thread の parallel 動作をするのに間 に合わない場合や register が十分に存在しない場合は, thread は部分的に parallel 動作部分的に逐次動作,あるいは全部逐次動作する. その場合でも concurrent 動作 ではできる限り短時間になるように, すなわち待ち時間のないように thread の動 作が効率よく制御される. この SIMT と concurrent 動作は行列の乗算と加減算を CUBLAS ライブラリが実行するやり方を理解するために非常に重要である.

2.2.3 GPUstream



図 2.2: GPUstream

GPUのストリームは一連の非同期操作のことである. GPUと CPU 間のデータ 転送, GPUのメモリ確保, カーネル呼び出しが含まれている. ストリームは SM から thread 実行数が用意できるのであれば 2.2 のカーネル 1 とカーネル 2 のように 並列実行することが可能である. しかし実行できる thread を用意できない場合ス トリーム 3 のようにストリーム並列実行を宣言しても逐次実行になってしまう.

2.2.4 GPU 数値計算ライブラリ

本研究ではGPUにおける行列の乗算と加減算にCUBLAS ライブラリを使用した. CUBLAS ライブラリは既存の線形代数ルーチンのライブラリである BLAS(Basic Linear Algebra Subprograms)を NVIDIA CUDA runtime 上に実装したライブラ リである. 図 2.3CPU の数値計算ライブラリである OpenBLAS と CUBLAS を用 いた行列の乗算の処理時間の比較を示す. 使用機器は GPU は P100 で CPU は IntelCore i7 6700K である. 図 2.3 の通り CPU の高速ライブラリである BLAS に 対して CUBLAS はどのデータサイズ N でも 7 倍以上速く計算することができる. この CUBLAS による行列の乗算の高速化は 2.1.1 節で説明した GPU のアーキ テクチャーの特性に起因している.

2.3 Strassen アルゴリズム

行列の乗算を高速に行うことができる Strassen アルゴリズムは大規模な数値計 算の分野で最近注目を集めているが, その内容は広く知られていない. したがって 2.3.1 で Strassen アルゴリズムについて説明する. Strassen アルゴリズムの CPU による計算はこれまで数多く研究されてきた. それらの研究を通して CPU 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明らかに されてきた. したがって 2.3.2 で GPU のリソースが影響する仕方を解明する本



図 2.3: OpenBLAS と CUBLAS の処理時間の比較

研究にとって参考になる CPU による研究を紹介する.本研究はこれまで報告され て来た GPU のための Strassen アルゴリズムのプログラムより高速なプログラム の作成を目指した.したがって本研究が比較すべき先行研究のプログラムを 2.3. 3 で説明する.

2.3.1 Strassen アルゴリズムの概要

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 [4]. N が偶数である $N \times N$ の正方行列 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 $N/2 \times N/2$ の部分行列の 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(行列要素の乗算と加算の回数) 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が標準的 な計算よりも演算回数が少なくなり, したがって高速になる. 長方形行列の場合も 同じように考えることができ, N が奇数の場合は padding の方法を使うことにより 同様に高速になる.

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 [5]. このアルゴリズムは Strassen-Winograd アルゴリ ズムと呼ばれ最も実用性の高い Strassen 型のアルゴリズムとして, これまでほとん どすべて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 アルゴリズムと記す.式 (2.1) と Alorithm 1 に Strassen アルゴリズムを 示す.

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

Algorithm 1 Strassen-alorithm

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

式から分かるように、Strassen アルゴリズムの計算は、途中で S_1 から S_8 、 M_1 から M_7 と V_1 から V_3 の Temporary 行列が導かれ、その要素をメモリに一時的に保存 する必要がある。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。 Multi-level Strassen の各レベルで行列の標準的な乗算よりも高速になる、すなわち speedup が1以上であると multi-level Strassen 全体での speedup は 1-level Strassen よりも高くなる。

2.3.2 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. 2. 3. 3節で説明するように, これらの研究に比べて GPU による研究は 2011 年に初めて報告され, その数は少ない. GPU を用いた場合にどのような GPU 特有 の問題が生じるか本研究で明らかにするために, この節では CPU による研究で明 らかにされたアーキテクチャーに関する問題を説明する. CPU を用いた Strassen アルゴリズムの主要な研究は文献 [6] で説明されている.

初期の CPU による研究は, Cray-2や Cray Y-MP などのベクトルプロセッサー から構成されるコンピューターを用いて 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 [7]. 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. このことは行列の標準的な乗算に対する高い speedup が得られなくなることを意味する. Temporary 行列の数を減らすと使用す るメモリは減る. しかし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, そのため通信時間が 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.これらの研究では shared-memory machine ある いは distributed-memory machine が用いられている.ここでの shared-memory は GPU の shared memory とは違い, むしろ global memory に近い役割を果たしてい る.Strassen アルゴリズムの部分行列の 7 個の乗算を複数の 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 [8]. これまでの CPU を用いた研究結果を基にして IBM の Engineering and Scientific Subroutine Library (ESSL) に CPU による計算のための Strassen アルゴリズムが 含まれている [6].

2.3.3 Strassen アルゴリズムの GPU 計算

IEEE explore の検索によると GPU を用いた Strassen アルゴリズムの計算はこ れまでに4 論文報告されている.それらの中で計算方法が明確に説明されており, 他の最近の論文で引用されている論文は Li ら [9] と Lai らの論文 [2] である.ここ ではこの2 論文について説明する.

2011 年に Li らは最初の 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,行数 16384 × 16384 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 らの計算および 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 を用いた単精度 3-level Strassen,正方行列のサイズ 15360 で1.27 倍 の speedup, 倍精度 4-level Strassen,正方行列のサイズ 8192 で1.42 倍の speedup を達成している.彼らはまた非正方行列および奇数サイズ行列の Strassen アルゴ リズムの計算方法を報告している.

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 [10].表 2.2 にこ の演算スケジュールを示す.またこのプログラムの Strassen アルゴリズムによる 計算の部分を Algorithm2,3,4,5 に示す.ここで部分行列の乗算の並列化とは、部分 行列の乗算を実行するカーネルの並列のことであり、それは各カーネル内のスレッ ドの並列とは基本的に異なる.

Algorithm 2 host device

Lai らは, 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図 2.4 とその演算スケジュールを表 2.3 に示す.

Algorithm 3 StrassenGPU Input: A[n][n], B[n][n], C[n][n], depthOutput: C[n][n] $A_{11}[0...n/2][0...n/2] = A[0...n/2 - 1][0...n/2 - 1]$ $A_{12}[0...n/2][0...n/2] = A[n/2...n][0...n/2-1]$ $A_{21}[0...n/2][0...n/2] = A[0...n/2 - 1][n/2...n]$ $A_{22}[0...n/2][0...n/2] = A[n/2...n][n/2...n]$ $B_{11}[0...n/2][0...n/2] = B[0...n/2 - 1][0...n/2 - 1]$ $B_{12}[0...n/2][0...n/2] = B[n/2...n][0...n/2-1]$ $B_{21}[0...n/2][0...n/2] = B[0...n/2 - 1][n/2...n]$ $B_{22}[0...n/2][0...n/2] = B[n/2...n][n/2...n]$ $C_{11}[0...n/2][0...n/2] = C[0...n/2 - 1][0...n/2 - 1]$ $C_{12}[0...n/2][0...n/2] = C[n/2...n][0...n/2-1]$ $C_{21}[0...n/2][0...n/2] = C[0...n/2 - 1][n/2...n]$ $C_{22}[0...n/2][0...n/2] = C[n/2...n][n/2...n]$ $CudaMalloc(T_1, n^2/4), CudaMalloc(T_2, n^2/4)$ if $(depth \le 1)$ then $low level(A[n][n], B[n][n], C[n][n], T_1[n/2][n/2], T_2[n/2][n/2])$ else $uplevel(A[n][n], B[n/2][n], C[n][n], T_1[n/2][n/2], T_2[n/2][n/2], depth - 1)$ end if CudaFree(T1), CudaFree(T2)

Algorithm 4 lowlevel

Input: $A[n][n], B[n][n], C[n][n], T_1[n][n], T_2[n][n]$ Output: C[n][n]表 2.2の演算スケジュール通り以下は行っている $T_1[0...n/2][0...n/2] = GpuAdd(A_{11}[0...n/2][0...n/2], A_{21}[0...n/2][0...n/2], -)$ $T_2[0...n/2][0...n/2] = GpuAdd(B_{22}[0...n/2][0...n/2], B_{12}[0...n/2][0...n/2][0...n/2], -)$ $C_{21}[0...n/2][0...n/2] = GpuMul(T_1[0...n/2][0...n/2], T_2[0...n/2][0...n/2]]$ $T_1[0...n/2][0...n/2] = GpuAdd(A_{21}[0...n/2][0...n/2], A_{22}[0...n/2][0...n/2], +)$ $T_2[0...n/2][0...n/2] = GpuAdd(B_{12}[0...n/2][0...n/2], B_{11}[0...n/2][0...n/2], -)$ $C_{22}[0...n/2][0...n/2] = GpuMul(T_1[0...n/2][0...n/2], T_2[0...n/2][0...n/2]]$ $T_1[0...n/2][0...n/2] = GpuAdd(T_1[0...n/2][0...n/2], A_{11}[0...n/2][0...n/2], -)$ $T_2[0...n/2][0...n/2] = GpuAdd(B_{22}[0...n/2][0...n/2], T_2[0...n/2][0...n/2], -)$ $C_{11}[0...n/2][0...n/2] = GpuMul(T_1[0...n/2][0...n/2], T_2[0...n/2][0...n/2]]$ $T_1[0...n/2][0...n/2] = GpuAdd(A_{12}[0...n/2][0...n/2], T_1[0...n/2][0...n/2], -)$ $C_{12}[0...n/2][0...n/2] = GpuMul(T_1[0...n/2][0...n/2], B_{22}[0...n/2][0...n/2]]$ $C_{12}[0...n/2][0...n/2] = GpuAdd(C_{12}[0...n/2][0...n/2], C_{22}[0...n/2][0...n/2], +)$ $T_1[0...n/2][0...n/2] = GpuAdd(A_{11}[0...n/2][0...n/2], B_{11}[0...n/2][0...n/2], -)$ $C_{11}[0...n/2][0...n/2] = GpuAdd(C_{11}[0...n/2][0...n/2], T_1[0...n/2][0...n/2], +)$ $C_{12}[0...n/2][0...n/2] = GpuAdd(C_{12}[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], +)$ $C_{11}[0...n/2][0...n/2] = GpuAdd(C_{11}[0...n/2][0...n/2], C_{21}[0...n/2][0...n/2], +)$ $T_2[0...n/2][0...n/2] = GpuAdd(T_2[0...n/2][0...n/2], B_{21}[0...n/2][0...n/2], -)$ $C_{21}[0...n/2][0...n/2] = GpuMul(A_{22}[0...n/2][0...n/2], T_2[0...n/2][0...n/2]]$ $C_{21}[0...n/2][0...n/2] = GpuAdd(C_{11}[0...n/2][0...n/2], C_{21}[0...n/2][0...n/2][0...n/2], -)$ $C_{22}[0...n/2][0...n/2] = GpuAdd(C_{22}[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], +)$ $C_{11}[0...n/2][0...n/2] = GpuMul(A_{12}[0...n/2][0...n/2], B_{21}[0...n/2][0...n/2][0...n/2]]$ $C_{11}[0...n/2][0...n/2] = GpuAdd(T_1[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], +)$

Algorithm 5 uplevel **Input:** $A[n][n], B[n][n], C[n][n], T_1[n][n], T_2[n][n], depth$ Output: C[n][n]表 2.2の演算スケジュールで行うが乗算は StrassenGPU で行う $T_1[0...n/2][0...n/2] = GpuAdd(A_{11}[0...n/2][0...n/2], A_{21}[0...n/2][0...n/2], -)$ $T_2[0...n/2][0...n/2] = GpuAdd(B_{22}[0...n/2][0...n/2], B_{12}[0...n/2][0...n/2][0...n/2], -)$ $StrassenGPU(T_1[0...n/2][0...n/2], T_2[0...n/2][0...n/2], C_{21}[0...n/2][0...n/2], depth)$ $T_1[0...n/2][0...n/2] = GpuAdd(A_{21}[0...n/2][0...n/2], A_{22}[0...n/2][0...n/2], +)$ $T_2[0...n/2][0...n/2] = GpuAdd(B_{12}[0...n/2][0...n/2], B_{11}[0...n/2][0...n/2], -)$ $C_{22}[0...n/2][0...n/2] = StrassenGPU(T_1[0...n/2][0...n/2], T_2[0...n/2][0...n/2][0...n/2], C_{21}[0...n/2][0...n/2][0...n/2]]$ $T_1[0...n/2][0...n/2] = GpuAdd(T_1[0...n/2][0...n/2], A_{11}[0...n/2][0...n/2], -)$ $T_{2}[0...n/2][0...n/2] = GpuAdd(B_{22}[0...n/2][0...n/2], T_{2}[0...n/2][0...n/2][0...n/2], -)$ $StrassenGPU(T_1[0...n/2][0...n/2], T_2[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], depth)$ $T_1[0...n/2][0...n/2] = GpuAdd(A_{12}[0...n/2][0...n/2], T_1[0...n/2][0...n/2], -)$ $StrassenGPU(T_1[0...n/2][0...n/2], B_{22}[0...n/2][0...n/2], C_{12}[0...n/2][0...n/2][0...n/2], depth)$ $C_{12}[0...n/2][0...n/2] = GpuAdd(C_{12}[0...n/2][0...n/2], C_{22}[0...n/2][0...n/2], +)$ $T_1[0...n/2][0...n/2] = GpuAdd(A_{11}[0...n/2][0...n/2], B_{11}[0...n/2][0...n/2], -)$ $C_{11}[0...n/2][0...n/2] = GpuAdd(C_{11}[0...n/2][0...n/2], T_1[0...n/2][0...n/2], +)$ $C_{12}[0...n/2][0...n/2] = GpuAdd(C_{12}[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], +)$ $C_{11}[0...n/2][0...n/2] = GpuAdd(C_{11}[0...n/2][0...n/2], C_{21}[0...n/2][0...n/2], +)$ $T_2[0...n/2][0...n/2] = GpuAdd(T_2[0...n/2][0...n/2], B_{21}[0...n/2][0...n/2][0...n/2], -)$ $StrassenGPU(A_{22}[0...n/2][0...n/2], T_2[0...n/2][0...n/2], C_{21}[0...n/2][0...n/2], depth)$ $C_{21}[0...n/2][0...n/2] = GpuAdd(C_{11}[0...n/2][0...n/2], C_{21}[0...n/2][0...n/2][0...n/2], -)$ $C_{22}[0...n/2][0...n/2] = GpuAdd(C_{22}[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], +)$ $StrassenGPU(A_{12}[0...n/2][0...n/2], B_{21}[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], depth)$ $C_{11}[0...n/2][0...n/2] = GpuAdd(T_1[0...n/2][0...n/2], C_{11}[0...n/2][0...n/2], +)$

	変数
A, B, C	初期行列
h_A, h_B, h_C	CPUの A, B, C の行列
d_A, d_B, d_C	GPU の A, B, C の行列
NX	行数 = $N/2$
NY	列数 = $N/2$
depth	分割回数
$A_{11}, A_{12}, A_{21}, A_{22}$	Aの部分行列
$B_{11}, B_{12}, B_{21}, B_{22}$	Bの部分行列
$C_{11}, C_{12}, C_{21}, C_{22}$	<i>C</i> の部分行列
T_1, T_2	Temporary 行列
	関数
CudaMalloc(A, N)	GPUメモリ確保.N のサイズ分確保
Cudamemcpy(A, B)	BからAヘコピー
GpuAdd(A, B, +or-) = cublasDgeam 関数	cublasによる加減算 $A + B$ 又は $A - B$
GpuMul(A, B) = cublasDgemm 関数	cublas によるによる乗算.AB
CudaFree	GPU メモリ解放

表 2.1: Algorithm2,3,4,5の変数と関数

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 は行数 2048 以下の範囲で見られ,行数 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,その具体的な説明をしていない.そしてこの並列プロ グラムを部分行列のサイズが小さい multi-level Strassen の最低の level の bottom level で使用することを提案している.

2.4 解決すべき課題

本研究ではLaiらのGPUのためのStrassenをさらに高速化にする. そのために 以下の課題を解決しなければならない.

(1) 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Strassen アルゴリズムを用いてない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.本研究で解決すべき課題の一つ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少する理由を具体的に明らかにすることである.GPUアーキテクチャーのどのような特性によってこのような制限が生じるのか明らかにし、その制限を緩和する方法を提案をする.この課題の解決については4章で報告する.

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

表 2.2: 先行研究の Temporary 行列 2 個の serial 計算プログラムの演算スケジュール

- (2) 先行研究では 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. CPU による Strassen アルゴリズムの研究 [7] によれば,Temporary 行列の数を減らすと使用するメモリが減り,そのためより大きなサイズの行列の計算が可能になる.一方 Temporary 行列の数を減らすとメモリとのデータの出し入れの回数が増え,その結果より計算時間がかかることになる.2.1.1で説明したように,GPU のアーキテクチャーは CPU とは大きく異なる考え方,つまり大量の算術演算を同時に実施できるように設計されている.したがって 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.このことを明らかにするのが本研究の次の課題であり,その解決については5章で報告する.
- (3) 先行研究で使用した GPU からさらに進化した GPU が現在まで次々に登場



図 2.4: Strassen-Winograd アルゴリズムの並列計算の模式図

し, 今後さらに進化した機器が登場することが期待される. このような状況の 下で, ある特定の世代の GPU でのみ可能な Strassen アルゴリズムの計算の 高速化を目指すのはあまり意味がない. それよりも 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. この課題の解決については 6 章で報告する.

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

表 2.3: Temporary 行列 15 個の並列計算プログラムの演算スケジュール

2.5 まとめ

本章では本研究に関わるこれまで報告されてきた主要な研究結果について論じ た.2.2節でGPUのアーキテクチャーと演算能力を説明した.GPUではCPU に比べてより大量の算術演算を concurrent に実施できるようにアーキテクチャー が構成されており、その結果高速の算術演算処理性能が可能となる.2.3.1節 で Strassen アルゴリズムを説明し,Strassen アルゴリズムの計算の途中で生じる Temporary 行列が計算の高速化に関わる要因であることを説明した.2.3.2節で これまで行われて来た CPUを用いた Strassen アルゴリズムの研究を紹介し,どの ように CPU のリソースが Temporary 行列の個数と部分行列の乗算の並列化に関 係するか説明した.2.3.3でこれまでの GPU を用いた Strassen アルゴリズムの 研究の中で重要なもの紹介し,本研究で比較すべき先行研究のプログラムを説明し た.以上の説明をもとに本研究で解決すべき課題を 2.4 で説明した.これらの議 論を通して GPU による Strassen アルゴリズムの計算の高速化を目指す本研究が 置かれた状況が示された.

第3章 評価環境の構築

3.1 はじめに

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 か解明するために次のような実験を構成した.与えられた行列のサイズについて標 準的な行列の乗算と Strassen アルゴリズムによる行列の乗算を実行し,それぞれの 計算の処理時間と Strassen アルゴリズムの標準的な乗算に対する Speedup の行列 サイズ依存性を 4,5,6章で調べた.Strassen アルゴリズムについては Temporary 変数の個数および部分行列の並列数を変えたプログラムを作成し,それぞれの処 理時間と Speedup の行列サイズ依存性を調べた.このように調べられたプログラ ムの中に先行研究 [2] のプログラムも含めた.また各計算の実行において NVIDIA Visual Profiler を使用して部分行列の乗算,加減算,メモリの確保などの個別の処 理にかかった時間を計測し,各演算処理に使われた thread 数や register 容量などの リソースを調べた.これらの作業を三種類の GPU を使用して実行した.

3.2 実験および解析方法

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.

- (1) 偶数 N の N × N 正方行列のみを対象とした. 偶数 N の N × N 正方行列 に対しては余分な行や列を加える padding などの付加的な処理を必要とせ ずに Strassen アルゴリズムを適用することができるので, GPU リソースの Strassen アルゴリズム計算への基本的な影響の仕方を明らかにすることが容 易となる. また行列のサイズは行数 N と一つのパラメーターだけ表すことが できる.
- (2) 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.
- (3)本研究で比較した先行研究の著者から得たプログラムが倍精度計算用だったので、本研究全体を通して倍精で計算した.また計算の高速化について倍精度計算で見出した傾向はそのまま単精度にも適用できる.

GPU	Tesla $K20[11]$	Tesla P100[12]	Tesla V100[13]
SMs	13	56	80
Peak FP64 TFLOPS	1. 7	5.3	7.8
Memory Size	$5~\mathrm{GB}$	16 GB	16GB
global memory band 幅	$208 \ \mathrm{GB/s}$	$720 \ \mathrm{GB/s}$	$900~{ m GB/s}$

表 3.1: 実験環境

- (4) 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.
- (5) プログラムを GPU で走らせた際の計算過程は NVIDIA Visual Profiler を用 いて調べた。

本研究は計算の高速化について先行研究 [2] のプログラムと本研究で作成したプ ログラムを比較した.そのために先行研究の著者から彼らのプログラムを提供し てもらい,それを本研究の実験環境で使用した.本研究のプログラムの Strassen ア ルゴリズムによる計算部分は 2.3.2 節で示した Algorithm2 を使用し,その演算スケ ジュールの部分を4章と5章で説明するプログラムごとに変えた.

3.3 実験環境

使用した GPU は NVIDIA Tesla K20, P100, および V100 である. このうち V100 は最新世代の NVIDIA GPU である. 表 3. 1 にこれら 3 種類の GPU の本研究に 関わる特性を記した. Cuda version は三台とも 10.1 である. V100 は NVIDIA の最 新の GPU であり、現時点で最高の演算性能を有している. K20 は V100 に比べて かなり 古い世代の GPU なので, 二つの GPU の大きな性能の違いが Strassen アル ゴリズムの計算速度にどのように現れるか調べることができる. 一方 P100 は V100 の一世代前の GPU であるので, 二つの GPU の性能のわずかな違いが Strassen ア ルゴリズムの計算速度に現れるかどうか調べることができる.

3.4 まとめ

本章では本研究の目的を達成するため4,5,6章の実験で先行研究[2]と比較する ための実験方法、解析方法及び実験環境について報告した.そして幅広い機器で 本研究が適用できることを示すために Tesla K20,Tesla P100, Tesla V100の3世 代の GPU を用いる理由を説明した.

第4章 部分行列乗算の並列数の最 適化

4.1 はじめに

本章では2.4節で説明した課題(1)を解決するために並列化の効果が行列のサ イズの増加とともに急速に減少する原因を具体的に明らかにする.その原因から 行数ごとの最適な並列実行数を決定できる方法を提示する.

4.2 stream 並列数の GPU リソースによる制限

先行研究[2]で行列のサイズの増加とともに部分行列の計算の並列化による speedup が急速に減少することが報告されていたので,表 2.3 で示した演算スケジュールに よる先行研究の並列プログラムを V100 で走らせ,その計算過程を visual profiler で調べた. 図 4.1 は行数 1024, 2048 および 10240 の場合の部分行列の計算過程の visual profiler time-line による表示である.緑色で表示された部分行列の乗算は, 行数 1,024 で7個の並列,行数 2048 で 3 個の並列,そして行数 10240 で逐次実行と なっている.Strassen アルゴリズムの CPU による計算の場合は,使用する計算機 のメモリ容量の不足によって部分行列の乗算の並列化が制限される.N×N行列 の計算に必要とされるメモリ容量は N² に比例する.したがって,この GPU で行 数 10240 の逐次計算が可能であるので,行数が 1024 から 2048 への増加で並列数が 7 から 3 に減少するのがメモリ容量の不足によるとして説明できない.

4.3 提案

4.2節で報告したように Strassen アルゴリズムの7個の部分行列の乗算を並列 実行するように作成されたプログラムを GPU で実行しても, 行列サイズの増加と ともに並列数が減少する.4.2節で考察したように, この並列数の制限は CPU の 場合のような GPU 全体のメモリの不足によっては説明できない. CPU と異なる GPU の重要な特性は大量の算術演算を実行するために多くの thread が生起され, それらが算術演算を実行するために大量の register を使用することである.した がって本研究では register 容量の不足が並列数の制限に影響している可能性を提案



図 4.1: 行数の増加に伴う部分行列の乗算の stream 並列数の変化

した.その可能性を実証するために Visual Profiler を使用して各行列サイズに対して、また三種類の GPU に対して部分行列の乗算に使われた thread 数や register 容量などのリソースを調べることにした.

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 thread 数 n_{th} と各 thread の register 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 multi-processor(SM) あたりの最大 register 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 [11, 12, 13]. 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 thread 数 n_{th}^{SM} が 求められる. この n_{th}^{SM} で使用する thread 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^{max}_{SM} 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 [11, 12 13]. n^{max}_{SM} を n_{SM} で割った値は並列に実行できる CUBLAS の乗算ライブラリの数にすなわち予測並列数になる. 以上の導出手順を式で表す.

$$n_{th}^{SM} = r_{max}/r_{th} \tag{4.1}$$

$$n_{SM} = n_{th} / n_{th}^{SM} \tag{4.2}$$

予測並列数 =
$$n_{SM}^{max}/n_{SM}$$
 (4.3)

	変数
r_{max}	<i>SM</i> あたりの最大 <i>register</i> 容量
r_{th}	各 thread の register 容量
n_{th}	カーネルが使用する総 thread 数
n_{th}^{SM}	一つの SM が実行する thread 数
n_{SM}	カーネルが使用する <i>SM</i> 数
n_{SM}^{max}	<i>GPU</i> が有する総 <i>SM</i> 数

表 4.1: Algorithm6 の変数

Algorithm 6 予測並列数の算出
Input: $r_{max}, r_{th}, n_{th}, n_{SM}^{max}$
Output: 予測並列数
$1: n_{th}^{SM} = r_{max}/r_{th}$
$2: n_{SM} = n_{th} / n_{th}^{SM}$
$3: 予測並列数 = n_{SM}^{max}/n_{SM}$

4.4 評価

4.4.1 stream 並列数の予測

本研究で使用した V100, P100 および K20 について 4.3 節で提案した式により 導いた値と visual profiler に表示された実際の並列数(実測並列数)を表 4.2, 4.3, 4.4 に示した.これらの表で予測並列数が 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,予測並列数が7 個を超えても実測並列数は7 個になる.

表4.2に見られるように実測並列数は予測並列数と一致あるいは非常に近い値と なっている.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 register 容量の不足によることを示している.

4.4.2 並列化よる高速化

4. 3. 1節で部分行列の乗算の並列化が GPU の各 SM が保有できる最大 register 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 ることを説明した. 最新の CUBLAS ライブラリ 10. 1を使用すると, Strassen アル ゴリズムによる計算が標準的な乗算の処理時間に近くなり, さらに短くなるのは行 数が 4000 以上である. この 4000 以上の行数の範囲で部分行列の乗算の並列化の効 果を V100, P100 の二世代の GPU について計算実験により調べた. 前節の表 4.2, 4.3, 4.4 によれば V100 のみ 2 個の並列化が可能で, P100 と K20 では逐次実行にな ることが予想される.

N	n_{th}	r_{th}	n_{th}^{SM}	n_{SM}	予測並列数	実測並列数
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 4.2: V100:stream 並列数の予測値と実測値

N	n_{th}	r/th	n_{th}^{SM}	n_{SM}	予測並列数	実測並列数
1,024	65, 536	64	4,000	16. 7	3. 4	3
2,048	32,768	240	1,067	30. 7	1.8	2
4,096	65, 536	240	1,067	61.4	0.91	1
6, 144	147, 456	240	1,067	138	0. 41	1
8, 192	262, 144	232	1, 130	238	0. 24	1
12, 288	589, 824	232	1, 130	535	0.10	1

表 4.3: P100:stream 並列数の予測値と実測値

計算に用いたプログラムの演算スケジュールを 4.5 に示す. このプログラムの Temporary 行列の個数は先行研究の逐次実行プログラムと同じ 2 個である. 演算 スケジュールの表に見られるように部分行列 7 個の内 6 個を 2 個づつ並列にして いる.表4.2 に示したように V100 について行数 4000 以上で部分行列 2 個の並列 が予想されるからである.計算結果を図 4.2 と図 4.3 に示す. V100 の場合は先行研 究のプログラムに比較して並列プログラムは Speedup は行数 4000 から 7000 の範 囲で高くなっている.並列化の効果は行数の増加につれて徐々に低下している.一 方 P100 の場合は並列プログラムの Speedup は先行研究の Speedup とほとんど変 わりはない.この結果は表 4.3 から予想された通りである.

4.5 考察

この章では部分行列の乗算の並列化の制限がGPUでは register 容量の不足によっ て起きることを明らかにした. CPU による Strassen アルゴリズムの計算の場合は 計算機が保有するメモリの不足によって起きる [8]. この場合のメモリは GPU の global memory に相当する. この GPU と CPU の違いは 2 章で説明した GPU アー キテクチャーの特徴に起因する.

行列の乗算を GPU で実行する場合, 一つの thread は積行列 C の一つの要素を 計算する. 積行列の一つの要素 *c_{ij}* の値は式

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{iN}b_{Nj} \tag{4.4}$$

N	n_{th}	r/th	n_{th}^{SM}	n_{SM}	予測並列数	実測並列数
1,024	4,096	255	1,003	4. 1	3. 2	2
2,048	16, 384	255	1,003	16. 3	0.80	1
4,096	65, 536	255	1,003	65.3	0. 20	1
6, 144	147, 456	255	1,003	147	0.09	1
8, 192	262, 144	255	1,003	261	0.05	1
12, 288	589, 824	255	1,003	588	0. 02	1

表 4.4: K20:stream 並列数の予測値と実測値



図 4.2: stream 並列化による speedup の行数依存性(V100)

で計算されるので合計 2N-1回の演算を thread は実行する. 行列のサイズが大き くなると一つの thread が行う演算回数は多くなるので,その分だけ多くの register 容量が必要となる. 表 4.2, 4.3, 4.4 で見られるように thread あたりの register 容量 r/th は各 GPU のリソースが許容する r/th の最大値 255 byte 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 thread 数も多くなり,並列実行 を行うと register 容量の不足が起きる. この現象は GPU アーキテクチャーの特徴, 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. NVIDIA の文 献 [14] にはこのような register 容量の不足によるプログラム実行の制約が register pressure と呼ばれ, それが様々な場合に起きることが記載されている.

Step	Operation	Stream
1	$T_1 = A_{11} - A_{21}$	
2	$C_{11} = B_{22} - B_{12}$	
3	$T_2 = A_{21}A_{22}$	
4	$C_{12} = B_{12} + B_{11}$	
5	$C_{12} = T_1 C_{11}$	0
	$C_{22} = T_2 C_{11}$	1
6	$T_1 = T_2 - A_{11}$	
7	$T_2 = B_{22} - C_{12}$	
8	$C_{12} = A_{12} - T_1$	
9	$C_{11} = T_1 T_2$	0
	$T_1 = C_{12}B_{22}$	1
10	$C_{12} = C_{12} + T_1$	
11	$B_{22} = A_{11} - B_{11}$	
12	$C_{11} = C_{11} + B_{22}$	
13	$C_{12} = C_{12} + C_{11}$	
14	$C_{11} = C_{11} + C_{21}$	
15	$T_1 = T_2 - B_{21}$	
16	$C_{21} = A_{22}T_1$	0
	$T_1 = A_{12}B_{21}$	1
17	$C_{22} = C_{22} + C_{11}$	
18	$C_{21} = C_{11} - C_{21}$	
19	$C_{11} = B_{22} + T_1$	

表 4.5: Temporary 行列を2個含む stream 並列実行プログラムの演算スケジュール

4.6 まとめ

visual profiler のデータ解析により行列サイズが大きくなると部分行列の乗算の 並列実行数が低下する原因が resister 容量と SM の数であることを明らかにした. この結果を基にして V100, P100 および K40 の各 GPU について対象とする行列の サイズについて実行可能な並列数を予測することができた. SM 数が最も多い V100 の場合は Strassen アルゴリズムによる乗算の標準的な乗算にたいする Speedup が 1.0付近なる行数 4000 から 6000 で並列化による speedup の増分が 9 % から 5 % となった.この並列化による寄与を含め且つ Temporary 行列の影響を削減したこ とによって高速化したプログラムを5章で報告する.



図 4.3: stream 並列化による speedup の行数依存性(P100)

第5章 Temporary 行列の削除

5.1 はじめに

本章では2.4節で説明した課題(2)を解決するために Temporary 行列の個数の 計算時間への影響を調べた結果を最初に報告する.次にその結果に基づいて作成し た Temporary 行列を削除した 1-level Strassen アルゴリズムと 2-level Strassen ア ルゴリズムのプログラムを提案する.これらのプログラムには4章で導いた部分 行列の乗算の最適な並列化を含めた.5.3節で提案した Temporary 行列を排除 し たプログラムを先行研究のプログラムと比較した実験の結果を報告する.本章で 報告する実験には部分行列の乗算の並列化による計算の高速化が期待できる V100 を用いた.

5.2 Temporary 行列に関する処理時間の割合

Strassen アルゴリズムの GPU による計算で Temporary 行列の使用による影響 を調べるために, 先行研究の 2 個の Temporary 行列を使用した逐次計算プログラ ムと Temporary 行列を 15 個使用した並列計算プログラムを GPU 走らせ, その計 算過程を visual profiler で調べた. その結果各 Temporary 行列に cudaMalloc によ るメモリの確保と cudaFree によるメモリの解放の処理時間がかかり, その処理時 間が行数の小さい場合に全体の計算時間に大きな割合を占めることが明らかになっ た. 図 5-1 に先行研究の 2 個の Temporary 行列を使用した逐次計算プログラムを GPU V100 で走らせた場合に Temporary 行列のためのメモリ確保とメモリ解放の 処理時間の全体の計算時間に占める割合を示す.



図 5.1: 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 (V100)

図 5.1 に見られるように行数が 2000 以下では大きな割合を占め, 行数の増加と ともに割合は減少する. 6 章で説明するように cudamalloc と cudaFree によるメモ リの確保と解放の処理時間は行列のサイズが大きくなるにつれて増加する. しかし 部分行列の乗算と加減算の処理時間はそれよりも急速に増大するので, これらの割 合は行列のサイズの増加とともに減少する. 5. 4 節で説明するように行数が 5000 から 6000 で Strassen アルゴリズムによる計算は標準的な乗算よりも1から2%高 速になる. この範囲でメモリ確保と解放の処理時間は全体の計算時間の同程度の割 合を占めている. したがってこれらの処理時間は Strassen アルゴリズムによる計 算をより高速化するためには無視できないオーバーヘッドである. Temporary 行 列のためのメモリの確保と解放を cudaMalloc と cudaFree を使用しないで行うこ とは現状の CUDA C による実装では不可能である.

5.3 提案

5.3.1 Temporary 行列を削除した 1-level Stassen アルゴリズム

先行研究で使用した 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 C₁₁, C₁₂, C₂₁, C₂₂ を Temporary 行列の代わりに用いている [10].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導いた.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をより高速化する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表

Step	Operation
1	$C_{22} = A_{11} - A_{21}$
2	$C_{11} = B_{22} - B_{12}$
3	$C_{21} = C_{22}C_{11}$
4	$A_{21} = A_{21} + A_{22}$
5	$C_{11} = B_{12} - B_{11}$
6	$C_{22} = A_{21}C_{11}$
7	$A_{21} = A_{21} - A_{11}$
8	$B_1 2 = B_{22} - C_{11}$
9	$C_{12} = A_{21}B_{12}$
10	$A_{21} = A_{12} - A_{21}$
11	$C_{12} = A_{21}B_{22}$
12	$C_{12} = C_{12} + C_{22}$
13	$B_{22} = A_{11} - B_{11}$
14	$C_{11} = C_{11} + B_{22}$
15	$C_{12} = C_{12} + C_{11}$
16	$C_{11} = C_{11} + C_{21}$
17	$B_{11} = B_{12} - B_{21}$
18	$C_{21} = A_{22}B_{11}$
19	$C_{21} = C_{11} - C_{21}$
20	$C_{22} = C_{22} + C_{11}$
21	$C_{11} = A_{12}B_{21}$
22	$C_{11} = B_{22} + C_{11}$

表 5.1: 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

5.1 に示す. 表 5.1 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.

4章で報告したように V100 を用いた場合は Strassen アルゴリズムによる行列の 乗算が行列の標準的な乗算よりも高速になる行列のサイズで部分行列の乗算を並列 実行することが可能になる.それ故 Temporary 行列の削除に加えて部分行列の乗 算を2個づつ並列にしたプログラムを作成した.そのプログラムの演算スケジュー ルを 5.2 に示す.

5.3.2 Temporary 行列を削除した 2-level Stassen アルゴリズム

Multi-level Strassen は 1-level Strassen で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. 各 level での 1-level

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

表 5.2: 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

Strassen の標準的な乗算に対する speedup が 1. 0 を越えていると, multi-level Strassen 全体の speedup はさらに高くなる. もし含められた 1-level Strassen の speedup が 1. 0 未満であると multi-level Strassen 全体の speedup は低下する. 5-5 節で説明するように 1-level Strassen は V100 を使用した場合行数が 5, 200 以上で speedup は 1. 0 以上になる. 一方行数が 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. 複数の GPU を使用するとメモリ が増えるので、より大きなサイズの行列の計算が可能になるが、GPU 間の遅い通 信のために Strassen アルゴリズムによる高速化の利点が失われる. 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 5.2 に 2-level Strassen を模式的に示す.

図 5.2 で二つの level を元の行数の upper level と, それを半分のサイズの部分 行列に分割した lower level を示した.本研究では lower level の計算に最速である Temporary 行列を削除し部分行列の乗算を 2 個づつ並列にした演算スケジュールを



図 5.2: 2-level Strassen アルゴリズムの模式図

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な upper level の演算スケジュールを作成しなければならない.

Upper level の演算スケジュールを導くために,最初に初期行列 A と B の部分 行列のデータを一時的に保存するための 8 個の Temporary 行列と部分行列の乗算 の結果を一時的に保存する 7 個の Temporary 行列を含んだ演算スケジュールを作 成した.次に計算途中でのデータの引き継ぎに誤りが出ないように確認しながら Temporary 行列の個数を 11 まで減らした.その次の段階で Lower level の演算スケ ジュールで再利用される行列 A と B の部分行列の個数を最小の 3 まで減らし,そ の条件のもとで Upper level の演算スケジュールの Temporary 行列の個数を 7 まで 減らした.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なければならない.こ のような条件の下で導いた upper level と lower level の演算スケジュールを表 5.3 と 5.4 に示す.

Step	Operation
1	$S_1 = A_{11}$
2	$S_2 = A_{21}$
3	$S_3 = B_{12}$
4	$T_1 = S_1 - S_2$
5	$T_2 = B_{22} - S_3$
6	$M_1 = T_1 T_2$
7	$T_1 = S_2 + A_{22}$
8	$T_2 = S_3 - B_{11}$
9	$S_2 = T_1 T_2$
10	$T_1 = T_1 - S_1$
11	$M_2 = T_1 B_{22}$
12	$M_2 = M_2 + S_2$
13	$T_1 = T_1 B_{11}$
14	$S_3 = S_2 + T_1$
15	$C_{12} = M_2 + S_3$
16	$S_3 = S_3 + M_1$
17	$T_2 = T_2 - B_{21}$
18	$M_1 = A_{22}T_2$
19	$C_{21} = S_3 - M_1$
20	$C_{22} = S_2 + S_3$
21	$S_3 = A_{12}B_{21}$
22	$C_{11} = T_1 + \overline{S_3}$

表 5.3: 2-level Strassen プログラムの upper level の演算スケジュール

5.4 評価

Temporary 行列を削除した 1-level Strassen アルゴリズム と 2-level Strassen アル ゴリズムのプログラムの計算速度を 2 個の Temporary 行列を含む先行研究のプログ ラムの計算速度を V100 GPU を用いて比較した.また 4.3 で説明した Temporary 行列を削除した上で部分行列の乗算を並列にした 1-level Strassen アルゴリズムの プログラムも比較した.1-level Strassen アルゴリズムの実験結果を図 5.3 に示す. 計算速度は行列の標準的な乗算に対する speedup で表した.

図 5.3 に見られるように行数が 4000 から 8000 の範囲で Temporary 行列が 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.この高速化の値は図 5.1 に示した結果とよく対応していて, 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ことを示している. 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

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

表 5.4: 2-level Strassen プログラムの lower level の演算スケジュール

することは, CPU 計算の場合と逆になっている. このことの原因については 5.6 節で考察する.

図 5.2 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. 並列化の結果行数が 4096 で先行研究のプログラムよりも 11 %高速なり, 行数 8192 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数 5200 で標準的な乗算と同じ計算速度になり, それ以 上の行数で標準的な乗算よりも高速になっている. 一方先行研究のプログラムは行 数 6656 で標準的な乗算と同じ計算速度になり, それ以上で標準的な乗算よりも高 速になる.

表 5.5 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グラムでそれぞれ 2 個である.表 5-5



図 5.3: 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)

の結果は行数 12288 で speedup の差は大きく, 行数の増加とともに差が減少することを示している.

5.5 考察

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 くなると使用するメモリの容量が減るので,より大きなサイズの行列の乗算が可 能になる.しかし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 [7].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.GPU は高い計算処理能 力を実現するために多量の core を使った算術計算を global memory とのデータの 転送と並列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算術演算と は別に逐次処理される [15].しかも GPU では Temporary 行列のためのメモリの確 保と解放が低速の CPU-GPU 間の通信を介して行われる.その結果 GPU ではメモ リの確保と解放は無視できないほどの時間を要する expensive operation とみなさ

行数 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 5.5: 先行研究と本研究の 2-level Strassen プログラムの speedup の比較

れている [14].

本研究で作成した Temporary 行列を削除し部分行列の乗算を並列実行するプロ グラムは V100 の計算で行数 5200 以上で行列の標準的な乗算よりも高速なり, 2level Strassen のプログラムは行数 14336 で標準的な乗算に対する speedup は 12 % となった.そして 1-level Strassen と 2-level Strassen の両方で先行研究の逐次計算 プログラムよりも高速になった. 2-5 で説明したように先行研究の逐次計算プログ ラムは K10 GPU を用いた倍精度 4-level Strassen, 正方行列のサイズ 8192 で標準的 な乗算に対して 42 %の speedup を達成している.また 1-level Strassen の倍精度計 算で行数 2048 以上で標準的な乗算よりも高速になっている [2].先行研究の逐次計 算プログラムの文献 [2] の speedup と本研究での speedup の違いは用いた CUBLAS ライブラリに起因する.文献 [2] で報告された計算では CUBLAS-5.0を使用して いる.一方本研究では先行研究のプログラムにも CUBLAS-10.1を使用している. CUBLAS-5.0から何度も改良されてきた CUBLAS-10.1では行列の乗算の速度 はかなり高速になっている.したがって行列の標準的な乗算に.対する Strassen ア ルゴリズムの計算の speedup は低くなる.

5.6 まとめ

Temporary 行列の数を減らすことで GPU のメモリ確保と解放の時間を削減 することで GPU Starassen を高速化できることが本章で明らかになった. そして 1 level, 2 level そして 4 章の部分行列並列実行の Temporary 行列削減のスケジューリ ングを導出した. V100 でそのスケジューリングによるプログラムを計測した結果 先行研究 [2] のスケジューリングプログラムよりも高速化できていることが図 5.3 と表 5.5 で示している. そして先行研究よりも最大で 11 %の高速化を実現するこ とができた.

第6章 Strassen アルゴリズムの speedup予測式

6.1 はじめに

4章と5章でGPUによるStrassenアルゴリズムの計算の高速化を目指した研究 の結果を報告した.本章ではこれらの章とは異なり,Strassenアルゴリズムの計算 の行列の標準的な乗算に対するSpeedupを理論的に予測できる式の導出を報告す る.そのような予測式が与えられれば今後導入されるものを含めた様々なGPUに ついて,例えばどの行列サイズからSpeedupが1.0を越えるかなど予測できるよ うになる.

6.2 visual profiler によるデータ解析

CUBLAS-10.1による行列の乗算と加算の行数依存性を visual profiler を使って 調べた.図 6.1に V100 で行った乗算と加算の処理時間を行数に対しての log-log plot を示す.乗算も加算も傾きが行数 N が 500 付近で変化し,その両側でほぼ直線 となっている.傾きを算出したところ,乗算の場合は 500 以下で 1500 以上で 3 と なった.このことは 500 以下で処理時間は N に比例し, 500 以上で N³ に比例して いることを意味する.加算の場合は傾きは 500 以下で 0 となり, 500 以上で 2 となっ た.このことは 500 以下で N⁰ に比例し, 500 以上で N² に比例していることを意味 する.

行列の乗算の場合は積行列の一つの要素 c_{ij} は式(4.1)で計算されるので、2(N-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

$$c_{ij} = a_{ij} + b_{ij} \tag{6.1}$$

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 N² であるので,図 6.1 は,積行列と和行列の各要素は行数 500 以下では並列計算され, 500 以上では逐次計算されていることを示唆している.この並列計算から逐次計算 への移行は thread の concurrent 実行に起因していることを 6.5考察で議論する.



図 6.1: 行列の計算時間の行数依存性の log-log plot (V100)

6.3 予測式の作成

Strassen アルゴリズムの計算が乗算の標準的な乗算よりも高速になる行列のサ イズは1,000よりもかなり大きい範囲である.したがってこ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導く.図5.4の結果に基づき,CUBLAS による行列の乗算の計算時間は次式で与えられると考えられる.

$$F_{mul}(N) = (1/F)(2N-1)N^2$$
(6.2)

ここで N は行列のサイズであり F は peakFP64(FLOPS)[11, 12, 13] 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に 現れないと仮定する. CUBLAS による行列の加減算の計算時間は次式で与えられ ると考えられる.

$$F_{add}(N) = (3 \times 8/G)N^2 \tag{6.3}$$

ここでGは global memory band 幅 [11, 12 13] 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.

上の2式を用いると1-level Strassen アルゴリズムの逐次計算の時間は

$$t_{Strassen}(N) = 7F_{mul}(N/2) + 15F_{add}(N/2) + n_{Tempo}(g_{malloc}(N/2) + g_{free}(N/2))$$
(6.4)

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く,図 6.2 に示した V100 で Strassen アルゴリズムの計 算をし, visual profiler によって求めた実測値を使う.



図 6.2: メモリの確保と解放の処理時間の行数依存性

CUBLAS による行列の標準的な乗算の計算時間は

$$t_{standard}(N) = F_{mul}(N) \tag{6.5}$$

である. したがって Strassen アルゴリズムの計算の speedup S は

$$S = t_{standard}(N)/t_{Strassen}(N)$$
(6.6)

で計算される.

6.4 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 6.3 と 6.4 に それぞれの理論値/実験値の比の行数依存性を示した.実験値は V100 によって求 めた.理論値の計算で F と G の値は 3 章の表 3.1 にある値を用いた.乗算の場合 は行数 7000 以上で比は一定で 0.90 である.加減算の場合は行数 4000 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 N³ 比例関係から,加減算の場合は N 比例関係からのズレによると考 えられる.



図 6.3: 行列乗算の計算時間の理論値と実験値の比較(V100)

図 6.5, 6.6, 6.7 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. 行列の合図の小 さいところではズレは大きいが, 行数への依存性の傾向は一致いている. V100 の 場合は他の GPU に比べてずれが大きいが, その理由については 6.5 節で考察する.

6.5 考察

本章で図 6.1 に示した実験結果に基づいて CUBLAS による行列の乗算と加減算 は大きな行列のサイズに対しては算術演算を逐次計算していると仮定した.その 仮定に基づいて導いた計算時間の理論式は,大きな行数の範囲で実測値とよく一致 した.このことは,CUBLAS では積行列の要素と和(差)行列の要素を小さい行 列の範囲で並列計算されるが,行数が増加するにつれて逐次計算に移行することを 意味している.このような計算の仕方の変化は2.1 で説明した thread や warp の concurrent 実行に起因する.Concurrent 実行とはデバイスのリソースが許す条件 の下で,ある時間枠の中でできる限り時間をかけないように複数の thread や warp が動作することである.行列の乗算の場合は,global memory から shared memory にデータが集団として移され,その移されたデータは積行列の要素の計算に繰り返 し使用される.そのため乗算の並列実行から逐次実行への移行はメモリからのデー タ移動が間に合わないことによるとは考えられない.この場合のリソースの条件



図 6.4: 行列加減算の計算時間の理論値と実験値の比較(V100)



図 6.5: Strassen アルゴリズムによる speedup の理論値と実験値の比較(V100)

は4.2節で部分行列の乗算の並列化の制限で説明したように register 容量の不足 によると考えられる.一方行列の加減算の場合は和(差)行列の一つの要素を計 算するのに毎回 global memory からデータを移さなければならない.そのため行 列のサイズが増加するにつれて並列計算をするのにメモリからのデータ移動が間 に合わなくなり,その結果逐次実行に移行すると考えられる.

図 6. 5, 6. 6, 6. 7の結果は大きな行数で speedup の予測式が実測値と非常によ く一致していることを示している.しかしこのことは speedup の予測式が非常に 正確であることを意味しているわけではない.図 6.3 と 6.4 の結果は行列の乗算と 加減算のより時間の理論値は実測値より 8 から 10 % ずれていることを示している. ただし,そのずれかたは乗算と加減算の両方で同じように起きている.その結果互 いのずれが打ち消し合い speedup 予測値は見かけ上実測値とよく一致するように なる.三世代の GPU について行数が 6000 以上で計測値と実測値の差が 2 % 以内



図 6.6: Strassen アルゴリズムによる speedup の理論値と実験値の比較(P100)

である。比較このことは予測式を speedup の値を正確に予測できるものと考える べきではなく, むしろおおよその値を示すものと考えるべきである.

Speedup の予測式の別の重要な役割は, 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 (6.3) 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 (6.3) 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 (6.3) 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. このことは4章の結果とともに今後 SM の数 が V100 より大きく増えた GPU が可能になれば行列の乗算は大きく高速化うるこ とが期待される.

6.6 まとめ

Strassen アルゴリズムによる行列の乗算の標準的な方法に対する speedup の予 測式を導いた.予測式は三世代の GPU について行数が 6000 以上で計測値と実測 値の差が2%以内で再現した.この結果に基づき 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考察をした.



図 6.7: Strassen アルゴリズムによる speedup の理論値と実験値の比較(K20)

第7章 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. NVIDIA GPU Tesla V100, P100 および K20 を使用し, 部分行列の乗算に CUBLAS-10. 1 を 用いた. また計算過程を NVIDIA visual profiler を用いて解析することにより計算 の高速化に影響する GPU のリソース要因を調べた. 本研究の主要な寄与は以下の 三点である.

- (1)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 による計算の研究で最速の逐次計算プログラム [2] よりもすべての行列のサ イズで高速となった. GPU V100 を用いた行数 4096 × 4096 の計算で先行研 究のプログラムより 11 %高速になった.また本研究の 1-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 も行数 5200 × 5200 以上で高速になり, 2-level Strassen アルゴリズムのプロ グラムは行数 14336 × 14336 で 12 %高速になった.これらの結果を4 章と5 章で報告した.
- (2) Strassen アルゴリズムの計算の高速化に CPU による Strassen アルゴリズムの計算の場合とは異なる GPU 特有のリソースの影響の仕方を初めて明らかにした.部分行列の並列化を制限するリソース要因は CPU の場合は計算機全体のメモリ容量であるが, GPU の場合は SM あたりの register 容量と GPU の SM の個数である.また Temporary 行列の個数を減らすと CPU の場合はメモリとのデータの出し入れの回数が増えるために計算速度は低下するが, GPU の場合は逆に計算速度は高くなる.この理由は GPU では Temporary 行列のためのメモリの確保と解放が低速の CPU-GPU 間の通信を介して行われるからである.これらの結果を4章と5章で報告した.
- (3) Strassen アルゴリズムによる行列の乗算の標準的な方法に対する speedupの 予測式を導いた.予測式は三世代の GPU の実測値をよく再現した.この結 果に基づき 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明らかにした.さらに様々な世代の GPU に適 用できる Strassen アルゴリズム計算のプログラムの標準的な行列の乗算に対 してどの行数から本研究の Strassen アルゴリズムが高速になるのか予測する ことができるようになった.これらの結果を6章で報告した.

本研究の結果は Strassen アルゴリズムによる行列の乗算の高速化に GPU が

今後重要な役割を果たすことができることを示している.今後 SM の個数が V100 よりも多い GPU が実現すれば Strassen アルゴリズムの部分行列の乗 算の並列化をさらに高めることができ,その結果 Strassen アルゴリズムの高 速化を通して GPU が科学技術の多くの数値計算の問題に威力を発揮し得る ことを本研究は示唆している.今後の展望として本研究のアプローチを他の 分割統治アルゴリズムにも適用する可能性を検討する。

参考文献

- 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] Nvidia Corp, NVIDIA CUDA C PROGRAMING GUIDE, October 2018.
- [4] V. Strassen. Gaussian elimination is not optimal. Numer. Math., 13:354-356, 1969.
- [5] S. Winograd. On multiplication of 2×2 matices. Linear Algebra and Application, 4:381-388, 1971.
- [6] A. R. Benson and G. Ballard, A framework for practical parallel fast matrix multiplication, in Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 42-53.
- [7] D. H. Bailey, K. Lee and H. D. Simon, Using Starssen's Algorithm to Accelerate the Solution of Linear Systems, J. Supercomputing, 4, 357-371, 1990.
- [8] 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.
- [9] 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.

- [10] 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.
- [11] Nvidia Corp, NVIDIA Tesla K20 (Whitepaper), 2012.
- [12] Nvidia Corp, NVIDIA Tesla P100 (Whitepaper), 2016.
- [13] Nvidia corp, NVIDIA Tesla V100 (Whitepaper), 2017.
- [14] Nvidia Corp, NVIDIA CUDA C BEST PRACTICES GUIDE, p. 41 October 2018.
- [15] X. Huang, et al., XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines, In Computer and Information Technology(CIT), 2010 IEEE, 1134-1139, July 2010.