

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Пояснительная записка к курсовому проекту по дисциплине
«Архитектура вычислительных систем»

Тема: Перемножение матриц с использованием технологии CUDA

Выполнил:
Студент гр. 753505
Альховский Е. Н.

Проверил:
Леченко А. В.

Минск 2019

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ.....	2
ВВЕДЕНИЕ.....	3
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ.....	4
1.1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ.....	4
1.2 ТЕХНОЛОГИИ И СРЕДСТВА РАЗРАБОТКИ.....	7
2. АЛГОРИТМЫ УМНОЖЕНИЯ МАТРИЦ.....	8
2.1 АЛГОРИТМ УМНОЖЕНИЯ МАТРИЦ ПО ЧАСТЯМ.....	8
2.2 АЛГОРИТМ ШТРАССЕНА.....	9
3. СРАВНЕНИЕ ВРЕМЕНИ РАБОТЫ НАИВНОГО АЛГОРИТМА С АЛГОРИТМОМ ШТРАССЕНА УМНОЖЕНИЯ МАТРИЦАХ.....	16
ЗАКЛЮЧЕНИЕ.....	18
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ.....	19

ВВЕДЕНИЕ

Умножение матриц — это один из базовых алгоритмов, который широко применяется в различных численных методах, и в частности в алгоритмах машинного обучения. Многие реализации прямого и обратного распространения сигнала в сверточных слоях нейронной сети базируются на этой операции. Так порой до 90-95% всего времени, затрачиваемого на машинное обучение, приходится именно на эту операцию. Почему так происходит? Ответ кроется в очень эффективной реализации этого алгоритма для процессоров, графических ускорителей (а в последнее время и специальных ускорителей матричного умножения). Матричное умножение — один из немногих алгоритмов, которые позволяют эффективно задействовать все вычислительные ресурсы современных процессоров и графических ускорителей. Поэтому не удивительно, что многие алгоритмы стараются свести к матричному умножению — дополнительная расходы, связанные с подготовкой данных, как правило с лихвой окупаются общим ускорением алгоритмов.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы быстрого перемножения матриц

Сложность вычисления произведения матриц по определению составляет $O(n^3)$, однако существуют более эффективные алгоритмы, применяющиеся для больших матриц. Вопрос о предельной скорости умножения больших матриц, также как и вопрос о построении наиболее быстрых и устойчивых практических алгоритмов умножения больших матриц остаётся одной из нерешённых проблем линейной алгебры.

• Алгоритм Штрассена (1969)

Первый алгоритм быстрого умножения больших матриц был разработан Фолькером Штрассеном в 1969. В основе алгоритма лежит рекурсивное разбиение матриц на блоки 2×2 . Штрассен доказал, что матрицы 2×2 можно некоммутативно перемножить с помощью семи умножений, поэтому на каждом этапе рекурсии выполняется семь умножений вместо восьми. В результате асимптотическая сложность этого алгоритма составляет $O(n^{\log_2 7}) = O(n^{2.81})$. Недостатком данного метода является бóльшая сложность программирования по сравнению со стандартным алгоритмом, слабая численная устойчивость и большой объём используемой памяти. Разработан ряд алгоритмов на основе метода Штрассена, которые улучшают численную устойчивость, скорость по константе и другие его характеристики. Тем не менее, в силу простоты алгоритм Штрассена остаётся одним из практических алгоритмов умножения больших матриц. Штрассен также выдвинул следующую гипотезу Штрассена: для сколь угодно малого ϵ существует алгоритм, при достаточно больших натуральных n гарантирующий перемножение двух матриц размера $n \times n$ за $O(n^{2+\epsilon})$ операций.

- Дальнейшие улучшения показателя степени ω для скорости матричного умножения

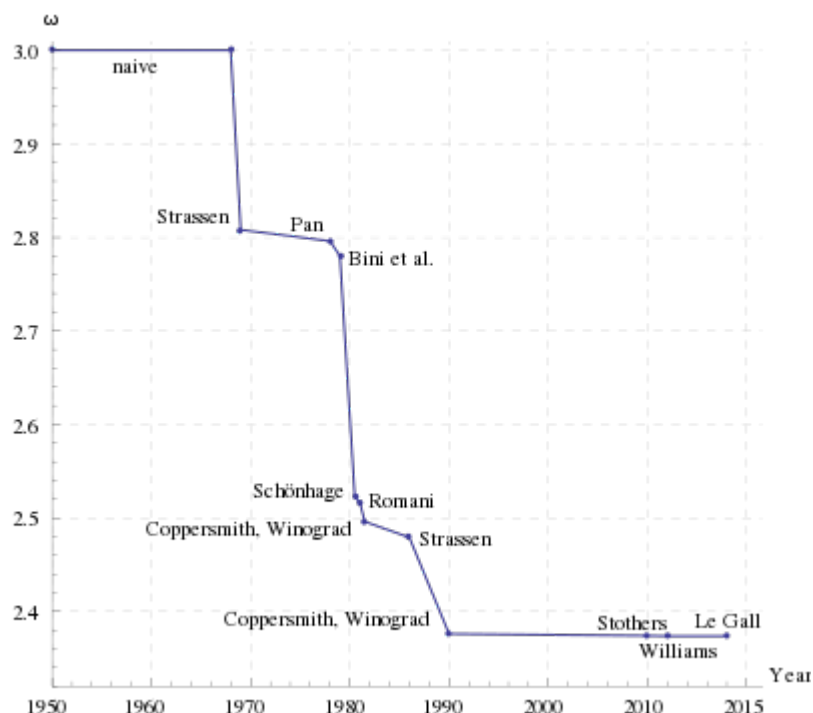


Рисунок 1.1. Хронология улучшения оценок показателя степени ω для скорости матричного умножения.

В дальнейшем оценки скорости умножения больших матриц многократно улучшались. Однако эти алгоритмы носили теоретический, в основном приближённый характер. В силу неустойчивости алгоритмов приближённого умножения в настоящее время они не используются на практике.

- **Алгоритм Пана (1978)**

В 1978 Пан предложил свой метод умножения матриц, сложность которого составила $O(n^{2.78041})$.

- **Алгоритм Бини (1979)**

В 1979 группа итальянских учёных во главе с Бини разработала алгоритм умножения матриц с использованием тензоров. Его сложность составляет $O(n^{2.7799})$.

- **Алгоритмы Шёнхаге (1981)**

В 1981 Шёнхаге представил метод, работающий со скоростью $O(n^{2.695})$. Оценка получена с помощью подхода, названного частичным матричным умножением. Позже ему удалось получить оценку $O(n^{2.6087})$.

Затем Шёнхаге на базе метода прямых сумм получил оценку сложности $O(n^{2.548})$ Романи сумел понизить оценку до $O(n^{2.5166})$, а Пан — до $O(n^{2.5161})$.

- **Алгоритм Копперсмита — Винограда (1990)**

В 1990 Копперсмит и Виноград опубликовали алгоритм, который в модификации Вильямс Василевской 2011 года умножает матрицы со скоростью $O(n^{2.3727})$. Этот алгоритм использует идеи, схожие с алгоритмом Штрассена. На сегодняшний день модификации алгоритма Копперсмита-Винограда являются наиболее асимптотически быстрыми. Но тот факт, что полученные улучшения ничтожны, позволяет говорить о существовании «барьера Копперсмита-Винограда» в асимптотических оценках скорости алгоритмов. Алгоритм Копперсмита-Винограда эффективен только на матрицах астрономического размера и на практике применяться не может.

- **Связь с теорией групп (2003)**

В 2003 Кох и др. рассмотрели в своих работах алгоритмы Штрассена и Копперсмита-Винограда в контексте теории групп. Они показали, что гипотеза Штрассена справедлива, если выполняется одна из гипотез теории групп.

1.2 ТЕХНОЛОГИИ И СРЕДСТВА РАЗРАБОТКИ

Алгоритмы разрабатывались на C++ с использованием **CUDA**.

CUDA (изначально аббр. от англ. Compute Unified Device Architecture) — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

Данная архитектура позволяет использовать **SIMD** инструкции, что позволяет ещё больше ускорить алгоритмы умножения матриц.

В программе использовалась библиотека **cuBLAS** (CUDA Basic Linear Algebra Subroutine library).

Чтобы использовать **API cuBLAS**, приложение должно выделить необходимые матрицы и векторы в пространстве памяти графического процессора, заполнить их данными, вызвать последовательность требуемых функций cuBLAS и затем загрузить результаты из пространства памяти графического процессора обратно на хост. API cuBLAS также предоставляет вспомогательные функции для записи и извлечения данных из графического процессора.

2. АЛГОРИТМЫ УМНОЖЕНИЯ МАТРИЦ

2.1 АЛГОРИТМ УМНОЖЕНИЯ МАТРИЦ ПО ЧАСТЯМ

Идея

Для матриц помещающихся в глобальную память GPU можно сразу вычислить их произведение с использованием *cuBLAS* (стандартная функция умножения матриц в **CUDA**)

В противном случае можно разбить матрицу на блоки, умножить их, а затем собрать всё воедино.

$$\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \end{bmatrix} = \begin{bmatrix} A_1 B_1 & A_1 B_2 \\ A_2 B_1 & A_2 B_2 \end{bmatrix}$$

Рисунок 2.1.1. Алгоритм умножения по частям

Алгоритм

Пусть есть две матрицы A и B размерами $n \times k$ и $k \times m$.

Разобьём матрицу A на $\lceil \frac{n}{a} \rceil$ частей, где a - количество строк в каждой подматрице. Если для $\lceil \frac{n}{a} \rceil$ ой матрицы не хватило строк, то дозаполняем их нулями.

Таким образом получаем $\lceil \frac{n}{a} \rceil$ матриц размером $a \times k$ для матрицы A обозначим A_i).

Таким же образом разбиваем матрицу B , но не по строкам, а по столбцам, т.е. матрица B разбивается на $\lceil \frac{m}{b} \rceil$ подматриц (обозначим B_j), где b - количество столбцов в каждой подматрице.

Получаем ещё $\lceil \frac{m}{b} \rceil$ матриц размером $k \times b$ для матрицы B .

Теперь если перемножить матрицы $A_i \cdot B_j$, то таким образом итоговая матрица будет являться подматрицей ответа с началом в элементе $[a \cdot i, b \cdot j]$, то есть заполнит подматрицу $[a \cdot i, b \cdot j], [(a+1) \cdot i, b \cdot j], [a \cdot i, (b+1) \cdot j], [(a+1) \cdot i, (b+1) \cdot j]$

Потребление памяти

$GPU\ memory\ usage = (k * (a + b) + a * b) * \text{sizeof}(\text{type})$, где a - количество строк в каждой подматрице A , b - количество столбцов в каждой подматрице B .

Теперь, изменяя a и b , можно изменять необходимый объём памяти.

Пример использования

Частный случай такого алгоритма, когда матрица B загружается полностью, а матрица A берётся частями, т.е., например, при чтении из файла.

В таком случае можно поблочно считывать из файла матрицу A , храня только одну подматрицу, умножать её на матрицу B , получать результат в качестве результата размер блока строк в итоговой матрице и сразу же записывать её в файла, чтобы также не занимать место.

2.2 АЛГОРИТМ ШТРАССЕНА

Идея, схожая с перемножением матриц по частям, лежит в основе работы алгоритма Штрассена. Здесь нужно будет перемножать матрицы 2×2 .

Данный алгоритм может перемножать матрицы за время $O(n^{\log_2^7})$ или $O(n^{2.81})$ (кстати, самый быстрый известный алгоритм может перемножать матрицы за время $O(n^{2.38})$).

Начнём с простого.

Разобьём матрицы A и B на 4 равные части (сами матрицы должны быть квадратные и одинаковой размерности), таким же образом представим матрицу C . Получим что-то такое - 3 матрицы размером 2×2 :

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

Рисунок 2.2.1. Пример матриц

Тогда, чтобы вычислить матрицу C можно воспользоваться формулами для умножения матриц:

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Рисунок 2.2.2. Алгоритм умножения матриц

В таком случае необходимо провести *8 операций умножения и 4 операции сложения*. Причём данные операции умножения мы можем выполнять рекурсивно, используя наш текущий алгоритм, пока размер матрицы > 1 (при матрице размером 1 просто $c = a \cdot b$).

Сложность такого алгоритма следующая: на каждый новый вызов нашего алгоритма проблема уменьшается в 2 раза, но таких уменьшающих операций у нас 8 (сложение можно не учитывать, т.к. это всего $O(n^2)$). Это можно записать следующим образом:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

Теперь воспользуемся мастер теоремой:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

Рисунок 2.2.3. Мастер теорема

И получим $a=8$, $b=2$, $f(n)=n^2$ - это первый случай, т.е. $\log_a b = \log_2 8 = 3 > 2$, т.е. сложность такого алгоритма $O(n^3)$. Никакого выигрыша не получаем.

Штрассен и Виноград

Штрассен предложил алгоритм, при котором вместо 8 умножений необходимо сделать всего 7, но количество сложений / вычитаний увеличится до 18.

Свой вариант для решение данной задачи предложил также и Winograd. В его варианте данного алгоритма необходимо провести 7 операций умножения и 15 операций сложения / вычитания, а это немного быстрее, чем у Strassen'a.

В таком случае изменяется формула:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

В мастер теореме получаем $a=7$, $b=2$, $f(n)=n^2$, это первый случай - $\log_a b = \log_2 7 = 2.81 > 2$, т.е. сложность такого алгоритма $O(n^{2.81})$. Это уже поинтереснее, чем $O(n^3)$.

Сложность

Сложность данного алгоритма: $O(n^{\log_2 7})$.

Формулы

Штрассен предложил вычислять результат произведения матриц следующим образом:

$$\begin{array}{ll} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) & C_{11} = M_1 + M_4 - M_5 + M_7 \\ M_2 = (A_{21} + A_{22})B_{11} & C_{12} = M_3 + M_5 \\ M_3 = A_{11}(B_{12} - B_{22}) & C_{21} = M_2 + M_4 \\ M_4 = A_{22}(B_{21} - B_{11}) & C_{22} = M_1 - M_2 + M_3 + M_6 \\ M_5 = (A_{11} + A_{12})B_{22} & \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) & \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) & \end{array}$$

Рисунок 2.2.4. Алгоритм Штрассена

В свою очередь Виноград предложил так:

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

Рисунок 2.2.5. Алгоритм Винограда

Реализация

Одноуровневая

Больше не уменьшается размер матрицы, считая, что текущий размер нас удовлетворяет, и мы готовы потратить на умножение $O(n^3)$ операций.

Step	Computation	GPU Kernel
1	$C_{12} = A_{21} - A_{11}$	<i>sub</i> (A_{21}, A_{11}, C_{12})
2	$C_{21} = B_{11} + B_{12}$	<i>add</i> (B_{11}, B_{12}, C_{21})
3	$C_{22} = C_{12} * C_{21}$	<i>mul</i> (C_{12}, C_{21}, C_{22})
4	$C_{12} = A_{12} - A_{22}$	<i>sub</i> (A_{12}, A_{22}, C_{12})
5	$C_{21} = B_{21} + B_{22}$	<i>add</i> (B_{21}, B_{22}, C_{21})
6	$C_{11} = C_{12} * C_{21}$	<i>mul</i> (C_{12}, C_{21}, C_{11})
7	$C_{12} = A_{11} + A_{22}$	<i>add</i> (A_{11}, A_{22}, C_{12})
8	$C_{21} = B_{11} + B_{22}$	<i>add</i> (B_{11}, B_{22}, C_{21})
9	$T_1 = C_{12} * C_{21}$	
10	$C_{11} = T_1 + C_{11}$	
11	$C_{22} = T_1 + C_{22}$	<i>mulIncInc</i> ($C_{12}, C_{21}, C_{11}, C_{22}$)
12	$T_2 = A_{21} + A_{22}$	<i>add</i> (A_{21}, A_{22}, T_2)
13	$C_{21} = T_2 * B_{11}$	
14	$C_{22} = C_{22} - C_{21}$	<i>mulStoreDec</i> ($T_2, B_{11}, C_{21}, C_{22}$)
15	$T_1 = B_{21} - B_{11}$	<i>sub</i> (B_{21}, B_{11}, T_1)
16	$T_2 = A_{22} * T_1$	
17	$C_{21} = C_{21} + T_2$	
18	$C_{11} = C_{11} + T_2$	<i>mulIncInc</i> ($A_{22}, T_1, C_{21}, C_{11}$)
19	$T_1 = B_{12} - B_{22}$	<i>sub</i> (B_{12}, B_{22}, T_1)
20	$C_{12} = A_{11} * T_1$	
21	$C_{22} = C_{22} + C_{12}$	<i>mulStoreInc</i> ($A_{11}, T_1, C_{12}, C_{22}$)
22	$T_2 = A_{11} + A_{12}$	<i>add</i> (A_{11}, A_{12}, T_2)
23	$T_1 = T_2 * B_{22}$	
24	$C_{12} = C_{12} + T_1$	
25	$C_{11} = C_{11} - T_1$	<i>mulIncDec</i> ($T_2, B_{22}, C_{12}, C_{11}$)

Figure 4. GPU kernels in Strassen implementation

Рисунок 2.2.6. Одноуровневая реализация алгоритма Штрассена

Step	Computation	GPU Kernel
1	$T_1 = A_{11} - A_{21}$	$sub(A_{11}, A_{21}, T_1)$
2	$T_2 = B_{22} - B_{12}$	$sub(B_{22}, B_{12}, T_2)$
3	$C_{21} = T_1 * T_2$	$mul(T_1, T_2, C_{21})$
4	$T_1 = A_{21} + A_{22}$	$add(A_{21}, A_{22}, T_1)$
5	$T_2 = B_{12} - B_{11}$	$sub(B_{12}, B_{11}, T_2)$
6	$C_{22} = T_1 * T_2$	$mul(T_1, T_2, C_{22})$
7	$T_1 = T_1 - A_{11}$	$sub(T_1, A_{11}, T_1)$
8	$T_2 = B_{22} - T_2$	$sub(B_{22}, T_2, T_2)$
9	$C_{11} = T_1 * T_2$	$mul(T_1, T_2, C_{11})$
10	$T_1 = A_{12} - T_1$	$sub(A_{12}, T_1, T_1)$
11	$C_{12} = T_1 * B_{22}$	
12	$C_{12} = C_{22} + C_{12}$	$mulAdd(T_1, B_{22}, C_{22}, C_{12})$
13	$T_1 = A_{11} * B_{11}$	
14	$C_{11} = C_{11} + T_1$	
15	$C_{12} = C_{11} + C_{12}$	
16	$C_{11} = C_{11} + C_{21}$	$mulIncIncInc(A_{11}, B_{11}, T_1, C_{21}, C_{11}, C_{12})$
17	$T_2 = T_2 - B_{21}$	$sub(T_2, B_{21}, T_2)$
18	$C_{21} = A_{22} * T_2$	
19	$C_{21} = C_{11} - C_{21}$	
20	$C_{22} = C_{11} + C_{22}$	$mulSubInc(A_{22}, T_2, C_{11}, C_{21}, C_{22})$
21	$C_{11} = A_{12} * B_{21}$	
22	$C_{11} = T_1 + C_{11}$	$mulAdd(A_{12}, B_{21}, T_1, C_{11})$

Figure 5. GPU kernels in Douglas et al.'s [7] implementation of Winograd variant

Рисунок 2.2.7. Одноуровневая реализация алгоритма Винограда

Рекурсивная

Размер матрицы стоит ещё уменьшить, потому что тратить на умножение $O(n^3)$ дорого.

```

Strassen(A, B, C, n) {
  if (n <= τ1) compute C = A * B using GPU8;
  else if (n <= τ2) compute C = A * B using Figure 4;
  else {
    C12 = A21 - A11; C21 = B11 + B12;
    Strassen(C12, C21, C22, n/2); // M6
    C12 = A12 - A22; C21 = B21 + B22;
    Strassen(C12, C21, C11, n/2); // M7
    C12 = A11 + A22; C21 = B11 + B22;
    Strassen(C12, C21, T1, n/2); // M1
    (C11 +, C22 +) = T1; T2 = A21 + A22;
    Strassen(T2, B11, C21, n/2); // M2
    C22 -= C21; T1 = B21 - B11;
    Strassen(A22, T1, T2, n/2); // M4
    (C11 +, C21 +) = T2; T1 = B12 - B22;
    Strassen(A11, T1, C12, n/2); // M3
    C22 += C12; T2 = A11 + A12;
    Strassen(T2, B22, T1, n/2); // M5
    (C11 -, C12 +) = T1;
  }
}

```

Figure 8. Strassen's GPU matrix multiply

Рисунок 2.2.8. Рекурсивная реализация алгоритма Штрассена

```

Winograd(A, B, C, n)
{
  if (n <=  $\tau_1$ ) compute  $C = A * B$  using GPU8;
  else if (n <=  $\tau_2$ ) compute  $C = A * B$  using Figure 5;
  else {
     $T_1 = A_{11} - A_{21}; T_2 = B_{22} - B_{12};$ 
    Winograd( $T_1, T_2, C_{21}, n/2$ ); //M4
     $T_1 = A_{21} + A_{22}; T_2 = B_{12} - B_{11};$ 
    Winograd( $T_1, T_2, C_{22}, n/2$ ); //M5
     $T_1 -= A_{11}; T_2 = B_{22} - T_2;$ 
    Winograd( $T_1, T_2, C_{11}, n/2$ ); //M1
     $T_1 = A_{12} - T_1;$ 
    Winograd( $T_1, B_{22}, C_{12}, n/2$ ); //M6
     $C_{12} += C_{22};$ 
    Winograd( $A_{11}, B_{11}, T_1, n/2$ ); //M2
     $(C_{12}, C_{11}) = (C_{11} + C_{12} + T_1, C_{11} + C_{21} + T_1);$ 
     $T_2 -= B_{21};$ 
    Winograd( $A_{22}, T_2, C_{21}, n/2$ );
     $(C_{21}, C_{22}) = (C_{11} - C_{21}, C_{11} + C_{22});$ 
    Winograd( $A_{12}, B_{21}, C_{11}, n/2$ ); //M3
     $C_{11} += T_1;$ 
  }
}

```

Figure 15. Winograd's GPU matrix multiply

Рисунок 2.2.9. Рекурсивная реализация алгоритма Винограда

Параметры

В реализации рассматривалось две версии:

- одноуровневая
- рекурсивная

Переход от рекурсивной к одноуровневой зависит от параметров.

Время работы функций в зависимости от параметров (среднее значение на 10 итерациях, матриц размером 128×128):

- для алгоритма Штрассена - это количество рекурсивных вызовов
- для умножения по частям - это размер подматриц

```

Strassen time: 0.0556287s for 1 recursion depth
Strassen time: 0.40641s for 2 recursion depth
Strassen time: 2.86887s for 3 recursion depth
Strassen time: 19.8099s for 4 recursion depth

```

Рисунок 2.2.10. Время работы алгоритма Штрассена

Как видно из времени работы - алгоритм Штрассена работает медленнее остальных. Это нормально, потому что матрица маленького размера.

Такие алгоритмы работают медленнее на маленьких матрицах, потому что у них большая константа.


```

MM in parts time: 0.00539216s for A block size: 128, for B block size: 128
MM in parts time: 0.00777094s for A block size: 128, for B block size: 64
MM in parts time: 0.0135004s for A block size: 128, for B block size: 32
MM in parts time: 0.0242131s for A block size: 128, for B block size: 16
MM in parts time: 0.0483261s for A block size: 128, for B block size: 8
MM in parts time: 0.0935054s for A block size: 128, for B block size: 4
MM in parts time: 0.182073s for A block size: 128, for B block size: 2
MM in parts time: 0.37262s for A block size: 128, for B block size: 1
MM in parts time: 0.00736975s for A block size: 64, for B block size: 128
MM in parts time: 0.0125402s for A block size: 64, for B block size: 64
MM in parts time: 0.0225559s for A block size: 64, for B block size: 32
MM in parts time: 0.0440445s for A block size: 64, for B block size: 16
MM in parts time: 0.0862236s for A block size: 64, for B block size: 8
MM in parts time: 0.178217s for A block size: 64, for B block size: 4
MM in parts time: 0.347736s for A block size: 64, for B block size: 2
MM in parts time: 0.691622s for A block size: 64, for B block size: 1
MM in parts time: 0.013473s for A block size: 32, for B block size: 128
MM in parts time: 0.0229816s for A block size: 32, for B block size: 64
MM in parts time: 0.0435518s for A block size: 32, for B block size: 32
MM in parts time: 0.0872021s for A block size: 32, for B block size: 16
MM in parts time: 0.17431s for A block size: 32, for B block size: 8
MM in parts time: 0.34719s for A block size: 32, for B block size: 4
MM in parts time: 0.741399s for A block size: 32, for B block size: 2
MM in parts time: 1.36722s for A block size: 32, for B block size: 1
MM in parts time: 0.0292041s for A block size: 16, for B block size: 128
MM in parts time: 0.044589s for A block size: 16, for B block size: 64
MM in parts time: 0.0896381s for A block size: 16, for B block size: 32
MM in parts time: 0.171082s for A block size: 16, for B block size: 16
MM in parts time: 0.344392s for A block size: 16, for B block size: 8
MM in parts time: 0.682767s for A block size: 16, for B block size: 4
MM in parts time: 1.36418s for A block size: 16, for B block size: 2
MM in parts time: 2.70471s for A block size: 16, for B block size: 1
MM in parts time: 0.048494s for A block size: 8, for B block size: 128
MM in parts time: 0.0900307s for A block size: 8, for B block size: 64
MM in parts time: 0.173385s for A block size: 8, for B block size: 32
MM in parts time: 0.346356s for A block size: 8, for B block size: 16
MM in parts time: 0.686416s for A block size: 8, for B block size: 8
MM in parts time: 1.3845s for A block size: 8, for B block size: 4
MM in parts time: 2.76677s for A block size: 8, for B block size: 2
MM in parts time: 5.47027s for A block size: 8, for B block size: 1
MM in parts time: 0.0974718s for A block size: 4, for B block size: 128
MM in parts time: 0.1797s for A block size: 4, for B block size: 64
MM in parts time: 0.348962s for A block size: 4, for B block size: 32
MM in parts time: 0.677291s for A block size: 4, for B block size: 16
MM in parts time: 1.37596s for A block size: 4, for B block size: 8
MM in parts time: 2.7312s for A block size: 4, for B block size: 4
MM in parts time: 5.45821s for A block size: 4, for B block size: 2
MM in parts time: 10.8172s for A block size: 4, for B block size: 1
MM in parts time: 0.191023s for A block size: 2, for B block size: 128

```

Рисунок 2.2.11. Время работы алгоритма умножения матриц по частям

Заметно, что с уменьшением размеров матриц время работы увеличивается.

3. СРАВНЕНИЕ ВРЕМЕНИ РАБОТЫ НАИВНОГО АЛГОРИТМА С АЛГОРИТМОМ ШТРАССЕНА УМНОЖЕНИЯ МАТРИЦАХ

Техника

Замерения проводились на CPU *i5-8365*.

Рассматривались матрицы размерами $2^i, i=[1, 10]$.

Для каждой матрицы и метода проводилось 100 итераций, затем бралось среднее арифметическое время работы.

Результаты

Matrix size	Naive algo time (ms)	Strassen algo time (ms)
2	0.0001	0.0003
4	0.0008	0.0011
8	0.0053	0.0067
16	0.0318	0.0458
32	0.1512	0.2052
64	1.131	1.2524
128	9.3869	7.6067
256	98.562	53.5292
512	870.147	394.859
1024	28809.7	2795.72

Рисунок 3.1. Время работы методов в виде таблицы

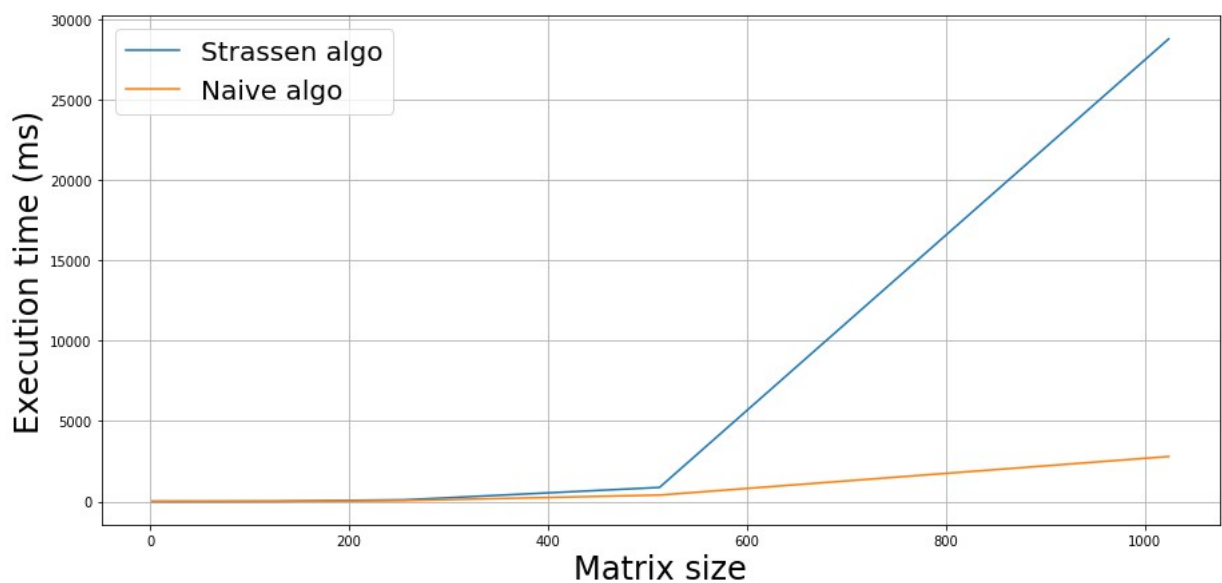


Рисунок 3.2. Время работы методов в виде графика

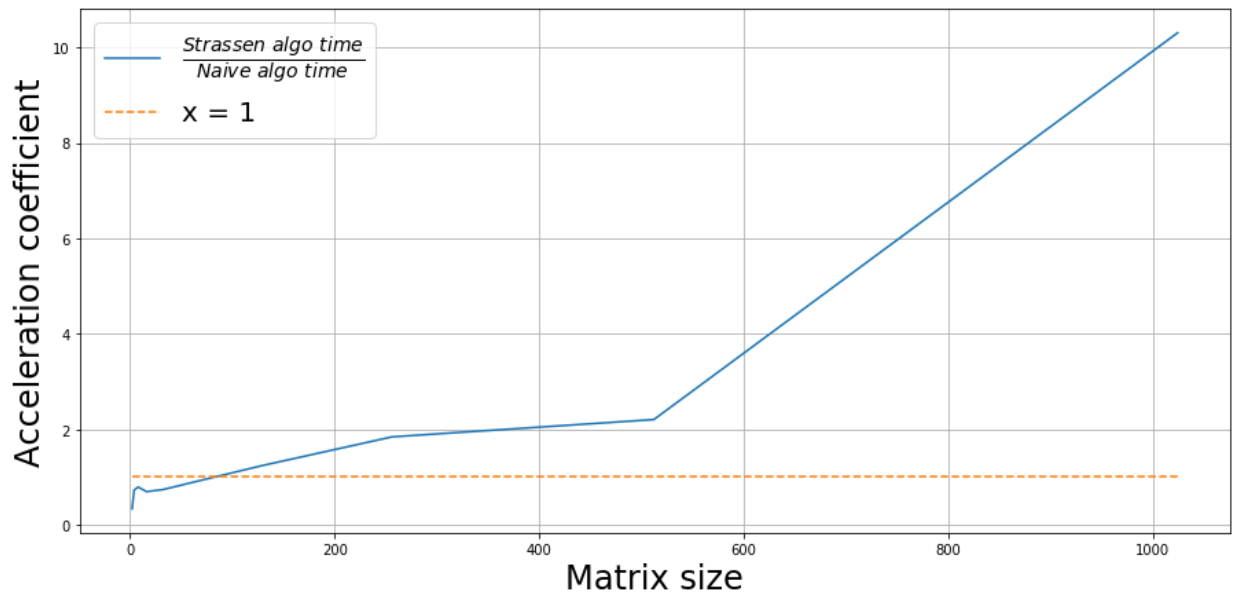


Рисунок 3.3. Коэффициент ускорения алгоритма Штрассена по сравнению с наивным алгоритмом

Выводы

Из графика (для ускорения работы) можно сделать вывод, что он является степенной функцией.

Также можно заметить, что для маленьких размерностей наивный алгоритм быстрее. Всё потому что константа перед алгоритмом Штрассена гораздо больше, чем перед наивным (в первом 7 умножений и 15 сложений матриц (для алгоритма Винограда), а во втором 8 умножений, но 4 сложения матриц).

ЗАКЛЮЧЕНИЕ

В результате проделанной работы было реализовано несколько алгоритмов умножения матриц, демонстрирующих, что существуют различные нетривиальные алгоритмы с более хорошей асимптотикой. Было проанализировано время работы этих алгоритмов и сделаны соответствующие выводы.

Также были использованы SIMD инструкции (архитектура CUDA), которые позволяют ещё больше ускорять матричные вычисления.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Strassen's Matrix Multiplication on GPUs. [электронный ресурс] - https://staff.kfupm.edu.sa/ics/ahkhan/Resources/Papers/Numerical%20Algorithms/strassen_gpu.pdf
2. CUDA Toolkit Documentation. [электронный ресурс] - <https://docs.nvidia.com/cuda/index.html>
3. cuBLAS [электронный ресурс] - <https://docs.nvidia.com/cuda/cublas/index.html>