МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМ. В. ДАЛЯ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ЕЛЕКТРОНІКИ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ІНЖЕНЕРІЇ

До захисту допускається
Т.в.о. завідувача кафедри
______________ Сафонова С.О.
«____»____________ 20__ р.

МАГІСТЕРСЬКА РОБОТА

НА ТЕМУ:

Дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів

Освітньо-кваліфікаційний рівень "Магістр"
Спеціальність 123 –"Комп'ютерна інженерія"

Науковий керівник роботи: ____________ (підпис) Д.О. Недзельський (ініціали, прізвище)

Консультант з охорони праці: ____________ (підпис) Я.О. Критська (ініціали, прізвище)

Студент: ____________ (підпис) Ю.М. Алимов (ініціали, прізвище)

Група: КІ-18зм

Сєвєродонецьк 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет Інформаційних технологій та електроніки
Кафедра Комп'ютерних наук та інженерії
Освітньо-кваліфікаційний рівень магістр
Напрям підготовки ______
(шифр і назва)
Спеціальність 123 –"Комп'ютерна інженерія"
(шифр і назва)

ЗАТВЕРДЖУЮ:
Т. в.о. завідувача кафедри ______
______ С.О. Сафонова
«___» ______ 20__ р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Алимову Юрію Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів

керівник проекту (роботи) Недзельський Д.О., доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від " 11 " 10 2019 р. № 135/15.15
2. Строк подання студентом роботи 16.01.2020
3. Вихідні дані до роботи матеріали науково-дослідної практики, наукові статті спеціальна література, матеріали мережі інтернет

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз засобів векторної обробки в ядрах сучасних процесорів. Постановка задачі. 2. Дослідження типових алгоритмів і розробка програм. 3. Аналіз результатів обчислювальних експериментів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Електронні плакати

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці та безпека в надзвичайних ситуаціях	Критська Я.О.		

7. Дата видачі завдання ____________________

Керівник ____________________
(підпис)

Завдання прийняв до виконання ____________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	а) Збір та вивчення джерел інформації для написання дипломної роботи; б) складання бібліографії наукових джерел	02.09.2019-10.09.2019	
2	Виконання та оформлення розділу з охорони праці	10.09.2019-15.09.2019	
3	Написання першого розділу, аналіз засобів векторної обробки в ядрах сучасних процесорів. Постановка задачі досліджень	15.09.2019-25.10.2019	
4	Написання другого розділу, дослідження типових алгоритмів і розробка програм	25.10.2019-20.11.2019	
5	Написання третього розділу, аналіз результатів обчислювальних експериментів	20.11.2019-05.01.2020	
6	Оформлення пояснювальної записки	05.01.2020-16.01.2020	
7	Захист дипломного проекту	21.01.2020	

Студент ____________ Алимов Ю.М.
(підпис) (прізвище та ініціали)

Науковий керівник ____________ Недзельський Д.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Алимов Ю.М. Дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів.

Предметом дослідження є ефективність використання засобів векторної обробки ядер сучасних процесорів при вирішенні ряду завдань комп'ютерами з універсальними високопродуктивними процесорами.

При дослідженні проводився огляд технологій, котрі використовують інструкції SIMD з моменту їх появи; дослідженні загальні алгоритми лінійної алгебри; розроблені алгоритми програмної частини, які детально дослідженні та описані.

Ключові слова: матриця, AVX, SSE, SIMD, інтринсіки, алгоритм, програма.

АННОТАЦИЯ

Алымов Ю.Н. Исследование эффективности использования средств векторной обработки ядер современных процессоров.

Предметом исследования является эффективность использования средств векторной обработки ядер современных процессоров при решении ряда задач компьютерами с универсальными высокопроизводительными процессорами.

При исследовании проводился обзор технологий, использующих инструкции SIMD с момента их появления; исследованы общие алгоритмы линейной алгебры; разработаны алгоритмы программной части, которые подробно исследованы и описаны.

Ключевые слова: матрица, AVX, SSE, SIMD, интринсики, алгоритм, программа.

ABSTRACT

Alymov Yu. Research into the effectiveness of using tools vector processing for modern processors.

The subject of the study is the efficiency of using the core vector processing tools of modern processors in solving a number of problems by computers with universal high-performance processors.

The study reviewed technologies using SIMD instructions from the moment they appeared; studied common algorithms of linear algebra; developed algorithms of the software part, which are detailed and described.

Keywords: Matrix, AVX, SSE, SIMD, intrinsic, algorithm, program.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК 7
ВСТУП 8
РОЗДІЛ 1 11
АНАЛІЗ ЗАСОБІВ ВЕКТОРНОЇ ОБРОБКИ В ЯДРАХ СУЧАСНИХ ПРОЦЕСОРІВ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕНЬ 11
1.1 Технологія MMX 11
1.1.1 Команди MMX 11
1.1.2 Типи даних застосовуваних команд технології MMX 11
1.1.3 Особливість застосування команд MMX і команд співпроцесора 11
1.1.4 Циклічна арифметика і арифметика з насиченням 12
1.2 Технологія SSE 13
1.2.1 Застосування SSE 13
1.2.2 Команди SSE-розширення 17
1.3 Технологія SSE2 18
1.4 Технології SSE3 – SSE4 20
1.4.1 Опис SSE3 20
1.4.2 Список процесорних команд та їх призначення 20
1.5 Технології AVX – AVX-512 22
1.5.1 Опис технології AVX 22
1.5.2 Можливості AVX-команд 23
1.5.3 Підтримка AVX, FMA і AES-технологій 23
1.5.4 Особливості використання AVX–інструкцій 24
1.5.5 Огляд набору інструкцій 24
1.5.6 Класи інструкцій технології AVX 27
1.5.7 Розширення технології AVX-512 28
1.5.8 Векторні розширення в архітектурах IBM, ARM 8-A 30
1.6 Аналіз методів і підходів для вирішення задачі програмування алгоритмів 31
1.6.1 Метод спекулятивної багатопоточності 33
1.6.2 Функціональний підхід до паралельного програмування 35
1.7 Постановка наукової задачі та обґрунтування методики досліджень 36
1.8 Висновки до першого розділу 37
РОЗДІЛ 2 39
ДОСЛІДЖЕННЯ ТИПОВИХ АЛГОРИТМІВ І РОЗРОБКА ПРОГРАМ 39
2.1 Опис способу організації пам'яті 39
2.2 Опис середовища розробки Microsoft Visual Studio 39
2.3 Визначення матриці 40
2.4 Вступ множення матриць 40
2.4.1 Опис алгоритму множення матриць 40
2.4.2 Програмний алгоритм множення матриць 41
2.5 Опис алгоритму транспортування матриць 51
2.5.1 Загальні формули транспонування матриць 51
2.5.2 Програмний алгоритм транспонування матриць 52
2.6 Вступ звернення матриць 55
2.6.1 Алгоритм звернення матриць 55

2.6.2 Алгоритм знаходження зворотної матриці ... 57
2.6.3 Програмний алгоритм звернення матриць ... 58
2.7 Висновки до другого розділу ... 66
РОЗДІЛ 3 ... 67
АНАЛІЗ РЕЗУЛЬТАТІВ ОБЧИСЛЮВАЛЬНИХ ЕКСПЕРИМЕНТІВ ... 67
3.1 Результати експериментів ... 67
3.2 Результати множення матриць ... 69
3.3 Результати транспонування матриць ... 71
3.4 Результат звернення матриць ... 74
3.5 Висновки до третього розділу ... 77
РОЗДІЛ 4 ... 78
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ ... 78
4.1 Загальні питання з охорони праці ... 78
4.1.1 Правові та організаційні основи охорони праці ... 78
4.1.2 Організаційно-технічні заходи з безпеки праці ... 79
4.2 Аналіз стану умов праці ... 79
4.2.1 Вимоги до приміщень ... 80
4.2.2 Вимоги до організації місця праці ... 81
4.2.3 Навантаження та напруженість процесу праці ... 82
4.3 Виробнича санітарія ... 82
4.3.1 Аналіз небезпечних та шкідливих факторів при виробництві (експлуатації) виробу ... 82
4.3.2 Пожежна безпека ... 83
4.4 Освітлення ... 84
4.5 Вентилювання ... 86
4.6 Заходи з організації виробничого середовища та попередження виникнення надзвичайних ситуацій ... 86
4.7 Екологія ... 88
4.8 Висновки до розділу 4 ... 90
4.9 Перелік корисних посилань до розділу 4 ... 91
ВИСНОВКИ ... 92
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 94
Додаток А. Лістинг коду множення матриць ... 97
Додаток Б. Лістинг коду транспонування матриць ... 100
Додаток С. Лістинг коду звернення матриць ... 103
Додаток Д. Комп’ютерна презентація ... 106

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

AVX	—	Advanced Vector Extensions **-** набір SIMD інструкцій;
SSE	—	Streaming SIMD Extensions, потокове SIMD **-** розширення процесора;
SIMD	—	Single Instruction, Multiple Data, одна команда **-** безліч даних;
FMA	—	Fused Multiply Add **-** операція множення з плаваючою точкою;
SVE	—	Scalable Vector Extension.

ВСТУП

Обґрунтування вибору теми дослідження. Серед різних сегментів галузі продовжує зростати необхідність у більш високих обчислювальних потужностях. Для задоволення зростаючого попиту і розвитку моделей застосування Intel продовжує розробляти інноваційні рішення.

Intel AVX — це набір команд, призначений для додатків, що інтенсивно використовують операції з плаваючою точкою. Він був випущений на початку 2011 року як частина сімейства процесорів під кодовою назвою Sandy Bridge на основі мікро архітектури Intel та використовується для різних платформ, від ноутбуків до серверів. Intel AVX підвищує продуктивність завдяки більш широким векторам, новому розширеному синтаксису і багатій функціональності. Це забезпечує краще управління даними і додатками загального призначення, наприклад, для обробки зображень, аудіо/відео, наукових імітацій, фінансового аналізу, тривимірного моделювання та аналізу.

В даний час на практиці часто виникають задачі, в яких для знаходження рішення використовуються різні алгоритми лінійної алгебри. Значна частина чисельних методів вирішення лінійних і нелінійних завдань включає в себе рішення систем лінійних рівнянь як елементарний крок відповідного алгоритму. Розвиток обчислювальної техніки дозволив перейти від простих моделей до складніших, у вигляді диференціальних рівнянь та їх дискретним аналогам на сітках. Цей перехід привів до необхідності вирішення великих розріджених систем лінійних алгебраїчних рівнянь з матрицями нерегулярної структури. Виникаючі на практиці системи часто є розрідженими, тобто містять велику кількість нульових елементів, а також мають велику кількість невідомих. Тому для їх вирішення потрібні значні технічні ресурси, адже на обчислення йде досить багато часу.

З іншого боку, процесори розвиваються в бік великої кількості паралельних [17] обчислень. Тобто конвеєризація суперскалярність, векторні операції - все це рівні паралелізму, наприклад коли намагаємося одночасно виконувати кілька арифметичних і логічних операцій. Векторизацію також можна вважати технологією одночасного виконання інструкцій. Зазвичай її називають "паралелізмом даних" при цьому мають на увазі заміну послідовного виконання скалярних операцій операціями над декількома даними [14]. Все це призводить до необхідності модифікації відомих алгоритмів з метою підвищення їх продуктивності. Впровадження векторних інструкцій в алгоритми дозволить скоротити витрати на обчислення. Завдяки цим засобам можна отримати більш ефективний алгоритм (за критерієм часу виконання).

Зараз практично у всіх процесорах загального призначення присутні набори коротких векторних інструкцій. Це важливе архітектурне нововведення останніх десятиліть, яке дозволяє

значно збільшити продуктивність процесора на мультимедійних і обчислювальних завданнях, в яких присутній паралелізм [17] на рівні даних. Найбільш поширеними методами введення векторних інструкцій в програму є автоматична векторизація компілятором, використання асемблерних вставок або спеціальні бібліотечні функцій (intrinsics). Вони найбільш ефективно з точки зору продуктивності кінцевого коду, однак, призводить до збільшення складності розробки.

В подальшому деякі нові продукти матимуть підтримку 512-бітної версії SIMD. Програми зможуть упаковувати числа з плаваючою точкою з подвоєною восьмикратною або одиничною шістнадцяти кратною ступенем точності в рамках 512-бітних векторів, а також восьми 64-бітних і шістнадцяти 32-розрядних цілих чисел. Це дозволить подвоїти обробку елементів даних, які можуть бути оброблені Intel AVX/AVX2 за допомогою однієї команди, і в чотири рази збільшити можливості Intel SSE. Команди Intel AVX-512 дуже важливі, так як вони відкривають нові можливості підвищення продуктивності для найвимогливіших обчислювальних завдань.

Мета і завдання дослідження. Метою магістерської роботи є дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів при вирішенні ряду завдань комп'ютерами з універсальними високопродуктивними процесорами.

Для досягнення поставленої мети, необхідно було вирішити такі завдання:

1. розглянути ряд типових завдань, для рішення яких необхідна висока продуктивність;
2. вибрати кілька алгоритмів з лінійною алгеброю;
3. розробити і налагодити програми без використання засобів векторної обробки і з використанням засобів векторної обробки;
4. зробити порівняння про час виконання і кількість тактів процесору розроблених алгоритмів;
5. зробити висновки про ефективність використання векторної обробки в комп'ютерах з універсальними процесорами.

Об’єкт, методи та джерела дослідження. Основним об’єктом дослідження є векторизація в ядрах сучасних процесорів. Методи виконання роботи: програмний, графічний, порівняльний.

Наукова новизна отриманих результатів. Розвиток обчислювальної техніки дозволив перейти від простих моделей до складніших, у вигляді диференціальних рівнянь та їх дискретним аналогам на сітках. Цей перехід привів до необхідності вирішення великих розріджених систем лінійних алгебраїчних рівнянь з матрицями нерегулярної структури. Виникаючі на практиці системи часто є розрідженими, тобто містять велику кількість нульових елементів, а також мають велику кількість невідомих. Тому для їх вирішення потрібні значні технічні ресурси, адже на обчислення йде досить багато часу, більшість алгоритмів розроблялось без використання

векторної обробки що призводило до великих і громіздкий програмних алгоритмів.

Дослідивши проблематику була обрана тема, дослідження ефективності використання засобів векторних обробки ядер сучасних процесорів, розроблене практичне застосування AVX-інструкцій в алгоритмах транспонування, звернення матриць а також удосконалення алгоритму множення матриць. AVX - інструкції спростили написання алгоритмів, зменшили кількість математичних операцій, і поліпшили час виконання програм.

Практичне значення одержаних результатів. Результати та рекомендації магістерської роботи можуть бути використані для інтенсивних обчислень з плаваючою точкою в мультимедіа програмах і наукових завданнях. Там, де можлива більш висока ступінь паралелізму, збільшуючи продуктивність з дійсними числами.

Особистий внесок здобувача. Дисертаційне дослідження є самостійно виконаною роботою, в якій відображено особистий авторський підхід та особисто отримані теоретичні, прикладні та програмні результати, які відносяться до вирішення задачі дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів. Формулювання мети та завдань дослідження проводилось спільно з науковим керівником.

Структура і обсяг роботи. Магістерська робота складається зі вступу, 4 розділів, 5 висновків, список використаних джерел з 54 найменувань. Загальний обсяг роботи складає 126 сторінок. В магістерській роботі міститься 8 таблиць, 35 рисунків, 25 формул, 4 додатки.

РОЗДІЛ 1
АНАЛІЗ ЗАСОБІВ ВЕКТОРНОЇ ОБРОБКИ В ЯДРАХ СУЧАСНИХ ПРОЦЕСОРІВ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕНЬ

1.1 Технологія ММХ

1.1.1 Команди ММХ

Команди ММХ – це команди паралельної обробки цілочисельних даних для використання в мультимедійних додатках (Multimedia extensions – мультимедійні розширення). Вони призначені для одночасної обробки декількох елементів даних за одну інструкцію [17].

Команди ММХ включені в групу команд технології під назвою SIMD (Single Instruction, Multiple Data – одна команда, багато даних). За допомогою коду на асемблері в SIMD-розширеннях можна писати компактний і швидкий код для критичних фрагментів коду з використанням спеціальних інструкцій процесора для SIMD розширення [10].

1.1.2 Типи даних застосовуваних команд технології ММХ

Команди технології ММХ працюють з типами даних: 64 - розрядними цілочисельними даними, а також з даними, упакованими в групи загальною довжиною 64 біта. Такі дані можуть перебувати в пам'яті або у восьми ММХ - регістрах, які позначаються як ММ0-ММ7.

Команди ММХ-розширення виконуються в тому ж режимі процесора, що і команди з плаваючою точкою. При роботі з ММХ командами використовуються регістри стека математичного співпроцесора R0-R7. При цьому використовуються лише 64 розряди, а стекова організація, яка потрібна для операцій співпроцесора, не використовується. При спільному використанні математичного співпроцесора і ММХ-розширення останньою використовуваною командою ММХ-розширення повинна бути команда emms. Ця команда забезпечує коректний перехід процесора від виконання фрагмента програмного коду з ММХ - командами до обробки звичайних команд з плаваючою точкою співпроцесора.

1.1.3 Особливість застосування команд ММХ і команд співпроцесора

Команди ММХ-розширення забезпечують паралельну обробку до восьми байтів або чотирьох слів, або двох подвійних слів, а також підтримують роботу з такими типами даних [13]:

– упаковані байти - один 64-розрядний регістр містить 8 байтів;
– упаковані слова - один 64-розрядний регістр містить чотири 16-розрядних слова;
– упаковані подвійні слова - один 64-розрядний регістр 12 містить два 32-розрядних слова;
– 64-розрядні слова (quadword).

1.1.4 Циклічна арифметика і арифметика з насиченням

Обробка даних MMX-розширення може виконуватися одним із двох способів: з використанням або циклічної арифметики, або арифметики з насиченням.

Якщо команда задіє циклічну арифметику і результат операції виходить за двійкову розрядну сітку за видом даних, то результат виходить шляхом вирахування від максимально допустимого числа цього результату [18].

Якщо команда використовує арифметику з насиченням і результат операції перевищує максимальне уявне значення, то в вихідний операнд записується це максимальне значення (відбувається "насичення"). Першою буквою MMX-команд є буква "p".

Більшість команд мають суфікс, який визначає тип даних і використовувану арифметику:

– us (unsigned saturation) – беззнакове насичення, при якому, якщо результат операції перевищує максимальне значення, в вихідний операнд записується це максимальне значення (відбувається "насичення"). Якщо результат операції виявився менше нижньої межі допустимого діапазону, то в вихідний операнд записується мінімально можливе значення;

– s або ss (signed saturation) - знакове насичення, при якому використовується арифметика з насиченням, дані зі знаком;

– якщо в суфіксі немає ні символу s, ні символів ss, то застосовується циклічна арифметика (wraparound). Якщо в цьому випадку результат операції виходить за двійкову розрядну сітку за видом даних, то "зайві" старші біти результату відкидаються;

– b, w, d, q- це літери, які позначають певний тип даних. Якщо в суфіксі є дві з цих букв, то перша з них відповідає вхідному операнду, а друга - вихідного.

Дані зі знаком і без знака мають допустимий діапазон. Отже, якщо використовується арифметика з насиченням, то при виході результату операції за межі допустимого діапазону в операнд будуть записані значення в залежності від типу даних.

Допустимий діапазон даних наведено в таблиці 1.1. MMX-команди [16] умовно можна розділити на такі групи:

– команди передачі даних між регістрами MMX і цілочисельними регістрами і пам'яттю;

– арифметичні команди (додавання, віднімання, множення і комбінація множення і додавання);

– команди упаковки і розпаковування;

– команди порівняння на рівність або за величиною;

– логічні команди;

– команди зсуву (логічні і арифметичні);

– команди управління станом (MMX - встановлення ознак порожніх регістрів в слові тегів).

Таблиця 1.1 – Допустимий діапазон даних

Тип даних	Мінімальне значення	Максимальне значення
Байт зі знаком	80h (-128)	7Fh (127)
Байт без знаку	00h	FFh (256)
Слово зі знаком	8000h (-32768)	7FFFh (32767)
Слово без знаку	0000h	FFFFh (65535)
Подвійне слово зі знаком	80000000h (-2147483648)	7FFFFFFFh (2147483647)
Подвійне слово без знаку	00000000h	FFFFFFFFh (4294967295)
Збільшений учетверо слово зі знаком	1000 0000 0000 0000h	7FFFF FFFF FFFF FFFFh
Збільшений учетверо слово без знаком	0000 0000 0000 0000h	0FFFF FFFF FFFF FFFFh

У слові тегів вільному регістру відповідає комбінація 11, інші комбінації вказують лише на зайнятість регістра. Після кожної операції MMX біти тегів в регістрі призначення очищаються. Вільні у MMX біти [79:64] регістрів FPU заповнюються одиницями, так що помилкова обробка даних MMX інструкцією FPU призведе до операції винятку.

1.2 Технологія SSE

1.2.1 Застосування SSE

У 3D-графіку часто зустрічаються групи операцій, які можна виконати за один такт за допомогою SIMD-команд (single instruction, multiple data, тобто одна команда - багато даних [19]). Такими операціями є інтерполяція векторів, скалярне множення векторів, нормування векторів, інтерполяція компонент кольору (наприклад, RGB) і так далі. До SIMD-команд належать і команди SSE розширення.

Технологія SSE дозволила подолати 2-і основних проблеми MMX: при використанні

MMX неможливо було одночасно використовувати інструкції співпроцесора, оскільки його регістри були спільними з регістрами MMX, і можливість MMX працювати тільки з цілими числами.

Недоліком команд SSE є те, що в них відсутні команди отримання тригонометричних функцій, а при перенесенні значень функцій, отриманих на співпроцесор в регістри MMX - катастрофічно втрачається точність, наприклад, fcos, fst xr, movss XMM0, xr

SSE-розширення з'явилося в процесорах Intel Pentium III і доповнює MMX-розширення засобами обробки даних з плаваючою точкою. SSE-команди можна паралельно обробляти з командами математичного співпроцесора.

SSE-розширення реалізовано як апаратно-програмний модуль, який включає додаткові вісім XMM0-XMM7 регістрів розрядністю в 128 біт і 32-розрядний регістр управління/стану MXCSR для 32-бітного Protected Mode режиму і додатковими XMM8-XMM15 для режиму 64-бітного Long Mode.

Формат представлення даних SSE-розширення з плаваючою точкою в короткому форматі (Single Precision Floating Point, SPFP) наведено на рисунку 1.1

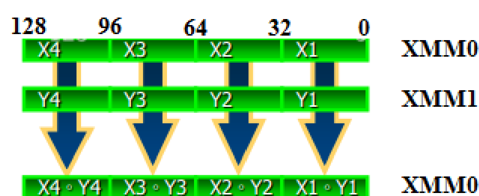

Рисунок 1.1 – Формат даних SSE-розширень [23]

Кожне 32-розрядне число з плаваючою точкою має 1 знаковий біт, 8 біт порядку і 23 біта мантиси, що відповідає стандарту IEEE-754 для формату чисел одинарної точності з плаваючою точкою.

Діапазон зміни цифр в SSE-форматі дорівнює 2^{-126} - ^{2127}. У зв'язку з тим, що формати команд співпроцесора і SSE-команд не однакові, в деяких випадках при різних межах вирівнювання результати обчислень з використанням форматів FPU і SSE можуть відрізнятися.

Структура полів регістра управління/стану (MXCSR) багато в чому являє структуру, яка реалізована в регістрах стану (swr) і управління (cwr) математичного співпроцесора. Процесом обчислень можна управляти шляхом зміни певних значень в полях цього регістра.

Для завантаження і збереження MXCSR використовуються команди:

– LDMXCSR m32 - завантажити MXCSR з пам'яті;
– STMXCSR m32 - зберегти MXCSR в пам'ять;

– FXRSTOR m512bytes - завантажити з пам'яті середу виконання (всіх регістрів) FPU, MMX, SSE;
– FXSAVE m512bytes - зберегти в пам'яті середу виконання (всіх регістрів) FPU, MMX, SSE.

При виникненні виняткових ситуацій встановлюються біти 0 - 5 в регістрі управління/стану (MXCSR). Формат полів регістра MXCSR і їх призначення наведені на рисунку 1.2.

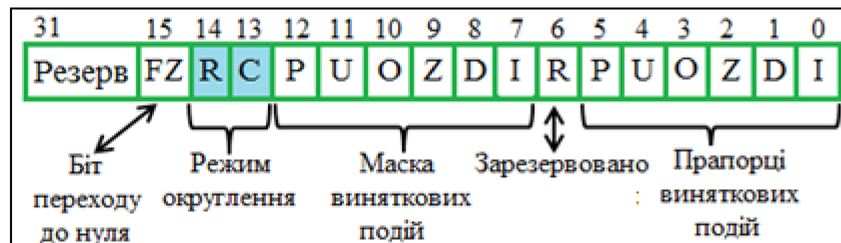

Рисунок 1.2 – Регістр управління/стану команд SSE-розширень.

Кожна виняткова ситуація може бути замаскована шляхом встановлення в 1 біт 7-12 регістра MXCSR. Якщо будь-які винятки замасковано, то воно обробляється процесором стандартних алгоритмом, після чого продовжується виконання коду.

Найбільше значення в регістрі управління/стану команд SSE-розширень мають біти 13-14 поля rc. За замовчуванням встановлюється режим округлення до найближчого значення числа з плаваючою точкою в короткому форматі.

Ці біти можна встановити програмно, причому можливі такі комбінації:

– 00 - округлення до найближчого числа;
– 01 - округлення до меншого числа;
– 10 - округлення до більшого числа;
– 11 - округлення з відкиданням дробової частини.

Біт 15 використовується, якщо результат операції близький до нуля. При цьому процесор виконує наступні дії:

а) повертає значення 0 і знак результату;
б) встановлює прапори (ознаки) P (біт 5) і U (біт 4);
в) маскує біти винятків.

Значна частина команд SSE може виконуватися в двох варіантах: як в скалярному так і в паралельному. Це стосується арифметичних команд і команд порівняння.

Команди паралельних арифметичних операцій [17] обробляють одночасно 4 подвійних слова і мають в своїй мнемоніці суфікс ps.

Команди скалярних операцій обробляють тільки молодші 32 - розрядні подвійні слова упакованих операндів (рис 1.3), не зачіпаючи при цьому три старших подвійних слова. Виняток становлять деякі команди скалярних передач даних. Мнемонічне позначення цих команд включає суфікс ss.

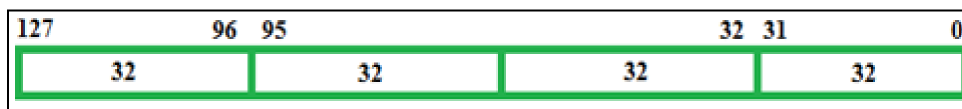

Рисунок 1.3 – Упаковані числа операндів 4x32 біта SSE [23]

В процесі обробки даних команди SSE-розширення можуть порушувати виняткові ситуації, які виникають, якщо відбувається одна з наступних подій:

а) некоректна операція (invalid operation) - I;

б) денормалізованний операнд (denormalized operand) - D;

в) розподіл на 0 (divide-by-zero) - Z;

г) арифметичне порівняння (numeric overflow) - O;

д) втрата значущих рядів (numeric underflow) - U;

е) втрата точності (inexact result) - P.

Всі команди SSE, крім спеціально обумовлених, при роботі з пам'яттю вимагають вирівнювання операнда на кордон 16 байтів. Інакше виникає помилка виконання. Для вирівнювання даних компілятори надають спеціальне ключове слово ALIGN 16 воно округлює лічильник конструкції.

Тип команд, які використовує процесор визначається відповідними розрядами:

– SSE: CPUID.01H:EDX[bit25]=1; Streaming SIMD Extension
– SSE2: CPUID.01H:EDX[bit26]=1
– SSE3: CPUID.01H:ECX[bit0]=1
– SSSE3: CPUID.01H:ECX[bit9]=1
– SSE4.1: CPUID.01H:ECX[bit19]=1
– SSE4.2: CPUID.01H:ECX[bit20]=1
– AVX: CPUID.01H:ECX[bit28]=1; Advanced Vector Extension
– FMA: CPUID.01H:ECX[bit12]=1; Fused Multiply Add

Для визначення підтримки SSE-команд використовується 25-й розряд регістра rdx і можливий фрагмент коду [7].

1.2.2 Команди SSE-розширення

Команди SSE-розширення можна розділити на кілька груп:

а) Команди для чисел з плаваючою точкою.

1) команди пересилки:

– скалярний тип: MOVSS;

– упаковані типи: MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS;

2) арифметичні команди:

– скалярні типи: ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS;

– упаковані типи: ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS;

3) команди порівняння:

– скалярні типи: CMPSS, COMISS, UCOMISS;

– упаковані типи: CMPPS;

4) перемішування і розпакування:

– упаковані типи: SHUFPS, UNPCKHPS, UNPCKLPS;

5) команди для перетворення типів:

– скалярні типи: CVTSI2SS, CVTSS2SI, CVTTSS2SI;

– упаковані типи: CVTPI2PS, CVTPS2PI, CVTTPS2PI;

6) побітові логічні операції:

– упаковані типи: ANDPS, ORPS, XORPS, ANDNPS.

б) Команди для цілих чисел:

1) арифметичні команди:

– PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW;

2) команди пересилки:

– PEXTRW, PINSRW;

3) інші:

– PMOVMSKB, PSHUFW.

в) Інші команди:

1) робота з регістром MXCSR:

– LDMXCSR, STMXCSR;

2) керування кешем і пам'яттю:

– MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE.

1.3 Технологія SSE2

Технологія SSE2 (Streaming SIMD Extensions 2) розроблена для застосування в процесорах Intel Pentium 4. Її призначення - підвищити ефективність операцій з 128-розрядними даними в форматі плаваючою точкою з подвійною точністю (double-precision floating point) і з цілочисельними даними. Ця технологія дозволяє розробляти високопродуктивні додатки для 3D-графіки і 3D-геометрії, моделювання і симуляції процесів, обробки сигналів, 3D анімації, кодування декодування, розпізнавання мови і так далі [7].

Технологія SSE2 розширює можливості MMX за рахунок використання 128-розрядних регістрів замість 64-розрядних, забезпечуючи високу ефективність паралельних обчислень. Висока продуктивність можлива також за рахунок включення в SSE2 нових типів даних: 128-розрядних операндів з плаваючою точкою подвійної точності і 128-розрядних упакованих цілих чисел.

Технологія SSE2 дозволяє:

– покращити управління даними в кеші;
– підвищити продуктивність операцій, які вимагають високої точності;
– розширити до 128 бітів діапазон оброблюваних 64 - розрядними командами операндів.

До групи SSE2 входять інструкції, які виконують 144 нових операцій. Більшість нових інструкцій двох адресні. Перший операнд є приймачем (dest), а другий - джерелом (src). Приймач, як правило, знаходиться в 128-бітному регістрі XMM, джерело може перебувати як в регістрі XMM, так і в оперативній пам'яті. Винятком є тільки інструкції пересилання, в яких приймач може розташовуватися в пам'яті. Третій операнд (якщо він є) - це ціле число, розмір якого не перевищує одного байта.

При роботі з числами можливе виконання тієї ж операції над однією парою або над двома парами чисел. У першому випадку ім'я операції закінчується буквами sd (скалярні дані), а в другому - буквами pd (упаковані дані). Для SSE2 вираз "скалярні дані" означає, що одне 64-бітове число розташовується в молодшій половині 128-бітному регістрі. А вираз "упаковані дані" означає, що два 64-бітних числа розташовані в одному 128-бітному регістрі або в пам'яті поспіль один за одним.

SSE-команди використовують вісім 128-розрядних регістрів (XMM0 - XMM7) і можуть працювати в скалярному або паралельному режимі.

SSE2-команди оперують з такими типами даних:

– упаковані і скалярні числа з плаваючою точкою в короткому форматі;

– упаковані і скалярні числа з плаваючою точкою подвійної точності;

– упаковані і скалярні цілі числа розміром 128 біт.

Формат даних 64-розрядного упакованого числа SSE2-розширення наведено на рисунку 1.4.

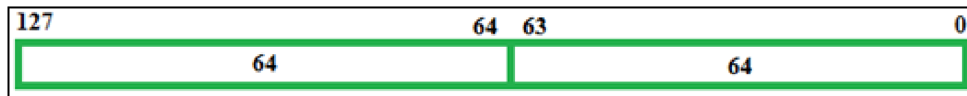

Рисунок 1.4 – Упаковане число 2х64 біта SSE2 [23]

За допомогою паралельних команд можна одночасно обробляти всі упаковані операнди, тоді як за допомогою скалярних - тільки молодший операнд. Команди SSE2-розширення в більшості випадків вимагають вирівнювання адрес операндів в пам'яті по 16-байтовому кордону, хоча з цього правила є деякі виключення, наприклад, команда завантаження або збереження операнда в НЕ вирівняною області пам'яті.

Для того, щоб переконатися в підтримці команд SSE2 необхідно виконати команду cpuid, проаналізувати 26-й біт регістра EDX, якщо цей біт відмінний від нуля, то технологія SSE2 підтримується процесором.

Для визначення підтримки SSE2-команд використовується 26-й розряд регістра rdx і можливий фрагмент коду [19]

Команди обробки 128-розрядних даних з плаваючою точкою поділяються на такі групи команд:

– переміщення (пересилання, передача) даних;

– арифметичні (додавання, віднімання, множення, ділення, витяг квадратного кореня і пошук максимуму/мінімуму);

– порівняння;

– логічні операції;

– розпакування і розподіл даних;

– перетворення форматів даних;

– управління станом обчислень;

– управління хешуванням даних.

1.4 Технології SSE3 – SSE4

1.4.1 Опис SSE3

SSE3 - третя версія SIMD-розширення Intel (нащадок SSE, SSE2 і MMX). Набір SSE3 містить 13 інструкцій: FISTTP (x87), MOVSLDUP (SSE), MOVSHDUP (SSE), MOVDDUP (SSE2), LDDQU (SSE/SSE2), ADDSUBPD (SSE), ADDSUBPD (SSE2), HADDPS (SSE), HSUBPS (SSE), HADDPD (SSE2), HSUBPD (SSE2), MONITOR (для AMD аналога немає в SSE3), MWAIT (для AMD аналога немає в SSE3).

Найбільш помітна зміна - можливість горизонтальної роботи з регістрами (команди додавання і віднімання кількох значень, що зберігаються в одному регістрі). Ці команди спростили ряд DSP- і 3D-операцій. Існує також нова команда fisttp для перетворення числа з плаваючою точкою з вершини стека співпроцесора в ціле число із записом в осередок пам'яті без необхідності вносити зміни в глобальному режимі округлення, наприклад, fisttp res1.

SSE4 складається з 54 інструкцій. Ці команди підтримуються мікропроцесорами, випущеними з листопада 2008 року. Команди SSE4 працюють тільки з 128-бітними регістрами xmm0 - xmm1, наведено на рисунку 1.5.

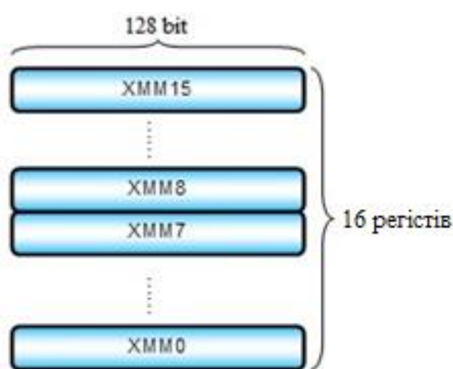

Рисунок 1.5 – SSE3-SSE4 – регістри xmm0 - xmm15 [24]

1.4.2 Список процесорних команд та їх призначення

Різні упаковані операції - це процесорні команди PMULLD, PMULDQ. З їх допомогою здійснюється підтримка чотирьох знакових (і без знаку) 32x32-бітних множень за одну інструкцію.

Скалярний добуток з плаваючою точкою - це процесорні команди DPPS, DPPD. З їх допомогою здійснюється підвищена продуктивність обробки даних масиву структур за допомогою підтримки скалярних добутків з одинарною і подвійною точністю.

Упаковане сполучення - це процесорні команди BLENDPS, BLENDPD, BLENDVPS,

BLENDVPD, PBLENDVB, PBLENDDW. У загальному випадку, ці команди копіюють одне поле від 24 джерела і переносять його в місце призначення. Ці процесорні команди підвищують продуктивність операцій сполучення для більшості розмірів полів за допомогою упаковки операцій множення в єдину інструкцію.

Упаковані цілочисельні максимальні і мінімальні значення - це процесорні команди PMINSB, PMAXSB, PMINUW, PMAXUW, PMINUD, PMAXUD, PMINDS, PMAXSD. Вони порівнюють упаковані знакові/без знаку на рівні байт слів/подвійних слів цілочисельні значення в операнді призначення і в вихідному операнді та повертають мінімальне або максимальне значення за одну інструкцію для кожного упакованого операнда в операнді призначення.

Округлення значень з плаваючою точкою - це процесорні команди ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD. Вони ефективно округлюють скаляр і упакований операнд з одинарною або подвійною точністю до цілочисельного значення.

Вставка витяг регістрів - це процесорні команди INSERTPS, PINSRB, PINSRD, PINSRQ, EXTRACTPS, PEXTRB, PEXTRD, PEXTRW, PEXTRQ. Ці процесорні команди спрощують процес встановлення та виймання між регістрами або пам'яттю.

Упаковане перетворення форматів - це процесорні команди PMOVSXBW, PMOVZXBW, PMOVSXBD, PMOVZXBD, PMOVSXBQ, PMOVZXBQ, PMOVSXWD, PMOVZXWD, PMOVSXWQ, PMOVZXWQ, PMOVSXDQ, PMOVZXDQ. Вони перетворять упаковане цілочисельне значення (з регістра XMM або пам'яті) в цілочисельне значення більш широкого типу зі знаковим або нульовим розширенням.

Упакована перевірка і установка - це процесорна команда PTEST. Вона здійснює більш швидке розгалуження для архітектури SIMD, здійснюване для підтримки векторизованого коду.

Упаковане визначення ідентичності - це процесорні команди PCMPEQQ, PCMPGTQ. Архітектура SIMD визначає ідентичність упакованих значень QWORD в операнді призначення і в вихідному операнді. Упаковка DWORD в без знаковий формат WORD. Це процесорна команда PACKUSDW. Вона перетворює упакований знаковий DWORD в упакований формат WORD без знаку за допомогою без знакового розширення для обробки умов переповнення.

Поліпшені рядкові операції - це процесорні команди PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. Ці команди містять в собі велику кількість можливостей обробки рядків і 25 тексту, які зазвичай вимагають участі більшої кількості кодів операцій. В результаті підвищується продуктивність сканування вірусів, пошуку тексту, строковою обробки бібліотек, таких, як ZLIB, баз даних, компіляторів (рис. 1.6).

1999	2000	2004	2006	2007	2008
Intel® SSE	Intel® SSE2	Intel® SSE3	Intel SSSE3	Intel® SSE4.1	Intel® SSE4.2
70 new instructions 4 single-precision vector FP scalar FP instructions cacheability instructions control & conversion instructions media extensions	144 new instructions 2 double-precision vector FP 8/16/32/64 vector integer 128-bit integer memory & power management	13 new instructions FP vector calculation x87 integer conversion 128-bit integer unaligned load thread sync.	32 new instructions enhanced packed integer calculation	47 new instructions packed integer calculation & conversion better vectorization by compiler load with streaming hint	7 new instructions string (XML) processing POP-Count CRC32

Рисунок 1.6 – Розвиток Intel SSE технологій [25]

1.5 Технології AVX – AVX-512

1.5.1 Опис технології AVX

Технологія AVX - це 256-бітне розширення набору інструкцій для технології SSE, призначене для додатків з плаваючою точкою (рис 1.7). Технологія AVX покращує продуктивність завдяки більш широким векторам, новому розширювальному синтаксису і багатим функціональним можливостям. Технологія AVX2 була анонсована в 2013 році, розширюючи можливості обробки вектора через домени даних з плаваючою точкою і цілими числами. Це призвело до більш високої продуктивності та більш ефективного управління даними в широкому діапазоні додатків. Прикладами можуть служити обробка зображень і аудіо/відео, наукове моделювання, фінансова аналітика, а також 3D-моделювання і аналіз.

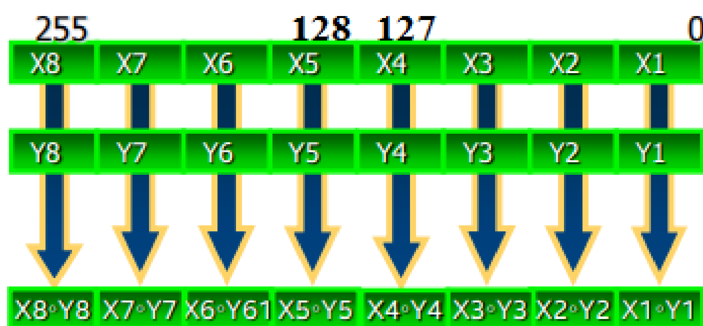

Рисунок 1.7 – Формат даних AVX-розширень [23]

1.5.2 Можливості AVX-команд

Технологія Advanced Vector Extensions (AVX) - це набір інструкцій для виконання операції з однією інструкцією і кількома даних (SIMD) на процесорах архітектури фірми Intel. Ці інструкції розширюють попередні пропозиції SIMD (інструкції MMX™ Streaming SIMD Extensions (SSE)) шляхом додавання нових функцій. 128-розрядні регістри SIMD розширили до 256 розрядних. В майбутньому технологія AVX буде підтримувати 512 - і 1024- бітові команди:

- команди з трьома операндами, наприклад, такі як A = B + C;
- спрощено вимоги до вирівнювання адрес оперативної пам'яті;
- для майбутніх доповнень команд розроблена нова схема кодування з використанням технології VEX.

Всі команди AVX, а також деякі інші команди, описані в довіднику [1].

1.5.3 Підтримка AVX, FMA і AES-технологій

Для використання технології AVX потрібна його підтримка не тільки процесором, але і операційною системою (вона повинна зберігати верхні 128-біт AVX регістрів при перемиканні контексту).

Розробники технології AVX передбачили спосіб дізнатися про підтримку цього набору інструкцій операційною системою. ОС зберігає відновлює контекст технології AVX з допомогою інструкцій XSAVE/XRSTOR, а конфігуруються ці команди за допомогою розширених контрольних регістрів (extended control register). Це регістр-XCR0, він же XFEATURE_ENABLED_MASK. Отримати його можна, записавши в ecx номер регістру (для XCR0 – це 0) і викликавши команду XGETBV. 64-бітне значення регістру буде збережено в парі регістрів edx, eax.

Виставлений нульовий біт регістра XFEATURE_ENABLED_MASK означає, що команда XSAVE зберігає стан FPU-регістрів (цей біт завжди виставлений), перший біт – збереження SSE-регістрів (молодші 128 біт AVX регістру), а виставлений другий біт – збереження старших 128 біт AVX регістра.

Таким чином, щоб бути впевненим, що система зберігає стан AVX-регістрів при перемиканні контекстів, потрібно переконатися, що в регістрі XFEATURE_ENABLED_MASK виставлені біти 1 і 2. Перед викликом команди XGETBV потрібно переконатися, що ОС дійсно використовує інструкції XSAVE/XRSTOR для управління контекстами. Робиться це за допомогою виклику інструкції CPUID з параметром eax=1: якщо ОС включила управління збереженням/відновленням контексту за допомогою інструкцій XSAVE/XRSTOR, то після

виконання CPUID в 27-му біту регістра ecx буде 1.

1.5.4 Особливості використання AVX–інструкцій

1. Небажано змішувати SSE- і AVX-інструкції. Щоб перейти від виконання AVX-інструкцій до SSE-інструкцій процесор зберігає в спеціальному кеші верхні 128 бітів AVX-регістрів, на що може піти півсотні тактів. Після виконання SSE-інструкцій процесор знову повернеться до виконання AVX-інструкцій. Він відновить верхні 128 бітів AVX регістрів, на що піде ще півсотні тактів.
2. Уникати збереження верхньої частини AVX-регістрів при переході до SSE-коду. Цього можна досягти, якщо обнулити верхні 128 бітів AVX-регістрів за допомогою команд vzeroupper або vzeroall. Ці команди працюють дуже швидко.
3. Вирівнювати на 16 байт перед виконанням команд завантаження/збереження вирівняних даних vmovaps/vmovapd/vmovdq.
4. Підпрограма на Windows x64 не повинна змінювати регістри xmm6-xmm15 (або відповідні їм регістри ymm6-ymm15). Перед виконанні підпрограм необхідно їх зберегти в стеці і перед виходом з підпрограм відновити їх з стека.

1.5.5 Огляд набору інструкцій

Нові інструкції кодуються з використанням того, що фірма Intel називає префіксом VEX, який представляє собою двох - або 3-байтовий префікс, призначений для усунення складності поточного і майбутнього кодування інструкцій x86/x64. Два нових префікса VEX формуються з двох застарілих 32-розрядних інструкцій - Load Pointer з використанням DS (LDS-0xC4, 3-байтовая форма) і Load Pointer з використанням ES (LES-0xC5, 2-байтовая форма) - які завантажують регістри сегментів DS і ES в 32-розрядному режимі. У 64-розрядному режимі опкоди LDS і LES генерують виняток неприпустимого опкода, але в технології AVX ці колі операцій (опкоди) повторно використовуються для кодування нових префіксів команд. В результаті інструкції VEX можна використовувати тільки при роботі в 64-бітному режимі. Префікси дозволяють кодувати більше регістрів, ніж попередні інструкції x86, і необхідні для доступу до нових 256-бітовим регістрів SIMD або використання трьох - і 4-операндного синтаксису. Користувача не потрібно турбуватися про це (якщо тільки ви не пишете асемблери або дизасемблери).

Інструкції SIMD дозволяють обробляти кілька частин даних в один крок, прискорюючи пропускну здатність для багатьох завдань, від кодування і декодування відео до обробки

зображень, аналізу даних і моделювання фізики.

Більш старі, пов'язані інструкції технології SSE також підтримують різні підписані і без знакові цілочисельні розміри, включаючи підписані і без знакові байти (B, 8-розрядні), word (W, 16-розрядні), doubleword (DW, 32-розрядні), quadword (QW, 64-розрядні) і doublequadword (DQ, 128-розрядні) довжини. Не всі інструкції доступні у всіх комбінаціях розмірів для отримання додаткової інформації дивіться посилання [10].

Апаратна підтримка технології AVX (і FMA) складається з 16 256-розрядних регістрів YMM0-YMM15 і 32-розрядного регістра управління стану під назвою MXCSR. Регістри YMM розташовані поверх старіших 128-розрядних регістрів XMM, використовуваних для Intel SSE, розглядаючи регістри XMM як нижню половину відповідного регістра YMM, як показано на рисунку 1.8.

Біти 0-5 MXCSR вказують на виключення SIMD з плаваючою точкою з "липкими" бітами-після установки вони залишаються встановленими до тих пір, поки не будуть очищені за допомогою LDMXCSR або FXRSTOR. Біти 7-12 маскують окремі винятки при установці, спочатку встановлені включенням живлення або скиданням. Біти 0-5 являють собою неприпустиму операцію, поділ на нуль, переповнення, підтік і точність відповідно.

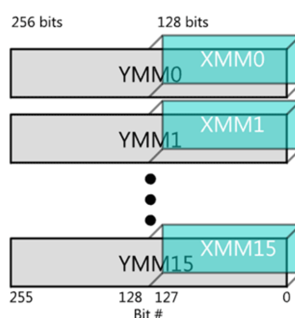

Рисунок 1.8 – Регістри XMM і YMM [22]

Типи даних використовувані в інструкціях технології SSE і технології AVX наведені на рисунках 1.9, 1.10. Грубо кажучи, для технології AVX допускається будь-кратний 32-розрядний або 64-розрядний тип з плаваючою точкою, який додає до 128 або 256 біт, а також кратні будь-якого цілочисельного типу, який додає до 128 бітам.

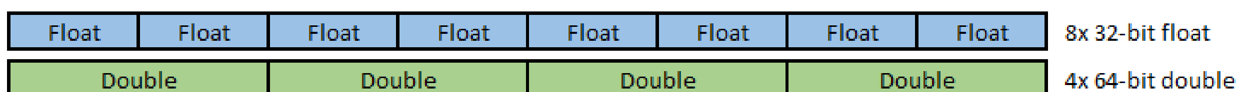

Рисунок 1.9 – AVX Data Types [24]

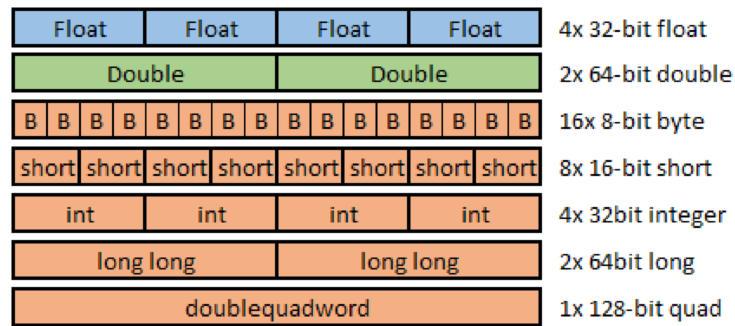

Рисунок 1.10 – SSE Data Types [24]

Інструкції часто приходять в скалярному і векторному варіантах (рис. 1.11). Векторні версії працюють шляхом обробки даних в регістрах в паралельному режимі "SIMD", скалярна версія працює тільки на одного запису в кожному регістрі. Ця відмінність дозволяє менше переміщати дані для деяких алгоритмів, забезпечуючи кращу загальну пропускну здатність.

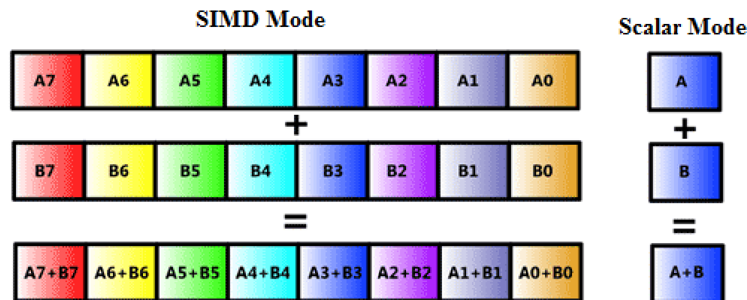

Рисунок 1.11 – SIMD проти Scalar- операцій [22]

Дані вирівняні по пам'яті, коли дані, які будуть використовуватися в якості N-байт шматка, зберігаються на кордоні пам'яті n-байта. Наприклад, при завантаженні 256-розрядних даних в регістри YMM, якщо джерело даних вирівняно по 256-бітному каналу, дані називаються вирівняними.

Для операцій технології SSE вирівнювання пам'яті було необхідно, якщо явно не вказані. Наприклад, у технології SSE існували спеціальні інструкції для операцій з вирівнюванням пам'яті і без вирівнювання пам'яті, такі як MOVAPD (Move-aligned packed double) і MOVUPD (переміщення-невирівняні запаковані подвійні) інструкції. Інструкції, не розділені на два, як це потрібно вирівняні доступи.

Технологія AVX пом'якшила деякі вимоги до вирівнювання пам'яті, тому тепер AVX за замовчуванням дозволяє не вирівняний доступ, однак цей доступ може призвести до зниження продуктивності, тому старе правило проектування даних для вирівнювання пам'яті, як і раніше, є гарною практикою (16-байтове вирівнювання для 128-бітного доступу і 32-байтове вирівнювання для 256-бітного доступу). Основними винятками є розширені версії інструкцій

технології SSE, які явно вимагають вирівняних по пам'яті даних: ці інструкції як і раніше вимагають вирівняних даних. Довідник з програмування технології Advanced Vector Extension, [20].

Інша проблема продуктивності, крім проблем з неузгодженими даними, полягає в тому, що змішування застарілих інструкцій XMM-only і більш нових інструкцій технології AVX викликає затримки, тому зводити до мінімуму переходи між інструкціями VEX-кодуванні і застарілим кодом технології SSE. Іншими словами, забороняється змішувати VEX-префіксні інструкції та інструкції без vex-префікса для оптимальної пропускної здатності. Якщо це необхідно зробити, мінімізуйте переходи між цими двома інструкціями шляхом групування інструкцій одного і того ж класу VEX non-VEX. Крім того, немає ніякого штрафу за перехід, якщо верхні біти YMM встановлені в нуль через VZEROUPPER або VZEROALL, які компілятори повинні автоматично вставляти. Ця вставка вимагає додаткової інструкції, тому профілювання рекомендується.

1.5.6 Класи інструкцій технології AVX

Як вже згадувалося, технологія AVX додає підтримку для багатьох нових інструкцій і розширює поточні інструкції Технології SSE до нових 256-розрядних регістрів, причому більшість старих інструкцій Технології SSE мають версію технології AVX з V-префіксом для доступу до нових розмірів регістрів і форм трьох операндів. Залежно від того, як підраховуються інструкції, існує до декількох сотень нових інструкцій технології AVX.

Наприклад, стара 2-операндна інструкція технології SSE ADDPS xmm1, xmm2/m128 тепер можна виразити в 3-операндном синтаксисі як VADDPS xmm1, xmm2, xmm3/m128 або 256-бітний регістр, який використовує форму VADDPS ymm1, ymm2, ymm3/m256. Кілька інструкцій дозволяють використовувати чотири операнда, наприклад VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4, який умовно копіює значення з плаваючою точкою одинарної точності з ymm2 або ymm3/m256 Для ymm1 на основі масок всередині ymm4. Це поліпшення в порівнянні з попередньою формою, де xmm0 був неявно необхідний, вимагаючи від компіляторів звільнити xmm0. Тепер, коли всі регістри явні, існує більше свободи для розподілу регістрів. Тут, m128 є 128-розрядної коміркою пам'яті, xmm1 це 128-бітний регістр, і так далі.

Деякі нові інструкції є тільки у VEX (не розширення технології SSE), включаючи багато способів переміщення даних в регістри YMM і з них. Приклади є корисними VBROADCASTS[S/D], який завантажує одне значення в усі елементи регістру XMM або YMM, а також способи перемішування даних в регістрі за допомогою VPERMILP[S/D].

Технологія AVX додає арифметичні інструкції для варіантів додавання, віднімання,

множення, ділення, квадратного кореня, порівняння, мінімуму, максимуму та округлення для одно - і двох точкових упакованих і скалярних даних з плаваючою точкою. Багато нових умовних предикатів також корисні для 128-бітної технології SSE, даючи 32 типи порівняння. Технологія AVX також включає інструкції, яка охоплює логічні, змішані, перетворені, тестові, запаковані, розпаковані, перемішані, завантажені і збережені. Набір інструментів також додає нові інструкції, в тому числі недиференційовану вибірку (широкомовну передачу одиночних або множинних даних 256-бітне призначення, масковане переміщення примітивів для умовного завантаження і зберігання), вставку і витяг декількох SIMD-даних 256-розрядні регістри SIMD і з них, перестановку примітивів для управління даними в регістрі, обробку гілок і запаковані інструкції тестування.

1.5.7 Розширення технології AVX-512

Команди технології AVX-512 працюють з 32-ма векторними регістрами з шириною 512 біт і вісьмома спеціальними регістрами маскування (рис. 1.12). Технологія AVX-512 яце гнучкий набір команд, що включає в себе підтримку широкомовного, вбудованого маскування для прогнозування, вбудованого управління команд округлення даних з плаваючою точкою, має вбудовані функції придушення помилок операцій з плаваючою точкою, команди розкиду, високошвидкісні математичні команди і компактне представлення великих перемішуючих значень.

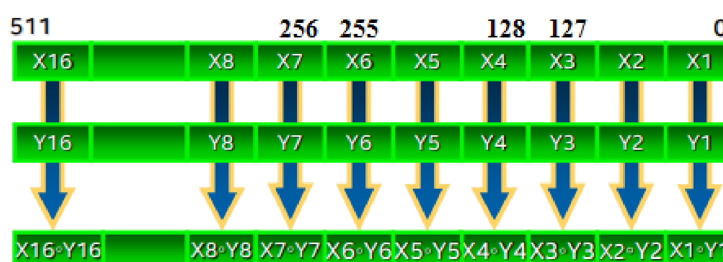

Рисунок 1.12 – Формат даних AVX-512- розширень [23]

Технологія AVX-512 сумісносна з командами технології AVX, які надійніше попередніх характеристик переходу до нових параметрів для операцій SIMD. На відміну від технології SSE і технології AVX, які не можуть використовуватися спільно без втрат продуктивності, спільне використання команд технології AVX і технології AVX-512 підтримується без будь-яких негативних наслідків. Регістри технології AVX YMM0-YMM15 призначаються для регістрів технології AVX-512 ZMM0-ZMM15 (в режимі x86-64) подібно до того, як регістри технології SSE призначаються для регістрів технології AVX. Тому в процесорах з підтримкою команд

технології AVX-512, команди технології AVX і технології AVX2 працюють з меншим числом бітів (128 або 256) перших 16 регістрів ZMM.

Майбутні настільні процесори технології під кодовою назвою Skylake не підтримуватимуть набір команд AVX-512. Це стало долею тільки серверних моделей процесорів Xeon, які, втім, також застосовуються і в робочих станціях. Раніше очікувалося, що підтримка 512-бітних команд SIMD, відомих під загальною назвою AVX3 стане однією з ключових особливостей структури Skylake і допоможе йому демонструвати неперевершений рівень продуктивності в додатках, які вміють використовувати нові набори команд.

Але фірма Intel вирішила не включати підтримку будь-яких розширень технології AVX-512 в звичайній, "побутовій" версії структури Skylake, в той час, як майбутні покоління процесорів Xeon з цією ж структурою будуть підтримувати такі розширення. Але навіть Xeon не будуть мати підтримку деяких 512-бітних команд, з якими вміють працювати співпроцесори фірми Intel Xeon Knights Landing. Раніше очікувалося, що процесори з структурою Skylake будуть підтримувати набір команд AVX 3.2.

Для порівняння, процесор Knights Landing підтримує AVX 3.1. Лише процесори, починаючи з процесорів під кодовим ім'ям Cannonlake, підтримують більшість 512-бітних розширень AVX. В таблиці 1.2 наведено процесори, які підтримують чи не підтримують команди.

Таблиця 1.2 - Команди в сучасних процесорів фірми Intel

CPU	Skylake	Skylake Xeon	CannonLake	Knights Landing	Haswell
SSE	+	+	+	+	+
SSE2	+	+	+	+	+
SSE3	+	+	+	+	+
SSE4.1	+	+	+	+	+
SSE4.2	+	+	+	+	+
FMA3	+	+	+	+	+
FMA4	-	-	-	-	-
AVX	+	+	+	+	+
AVX2	+	+	+	+	+
AVX512F	-	+	+	+	-
AVX512CDI	-	+	+	+	-
AVX512PFI	-	-	-	+	-
AVX512ERI	-	-	-	+	-
AVX512VLI	-	+	+	-	-
AVX512BW	-	+	+	-	-
AVX512DQ	-	+	+	-	-
AVX512IFMA52	-	-	+	-	-
AVX512VBMI	-	-	+	-	-
SHA	-	-	+	-	-
AES	+	+	+	+	+

Що стосується користі від 512-бітних команд, то вони, звичайно, будуть корисні у сфері високопродуктивних обчислень, але від їх використання може виграти і звичайний користувач, особливо якщо мова йде про вимогливих мультимедійних додатках. Відмова від підтримки 512-бітних команд «побутовими» універсальними процесорами в підсумку призведе до значного зниження темпів їх впровадження в програмне забезпечення. В цілому ж, без набору команд AVX 3.2 структура Skylake втрачає більшу частину своєї потенційної привабливості і практично перестає відрізнятися від структур Haswell і Broadwell. Еволюція SIMD команд наведена на рисунку 1.13.

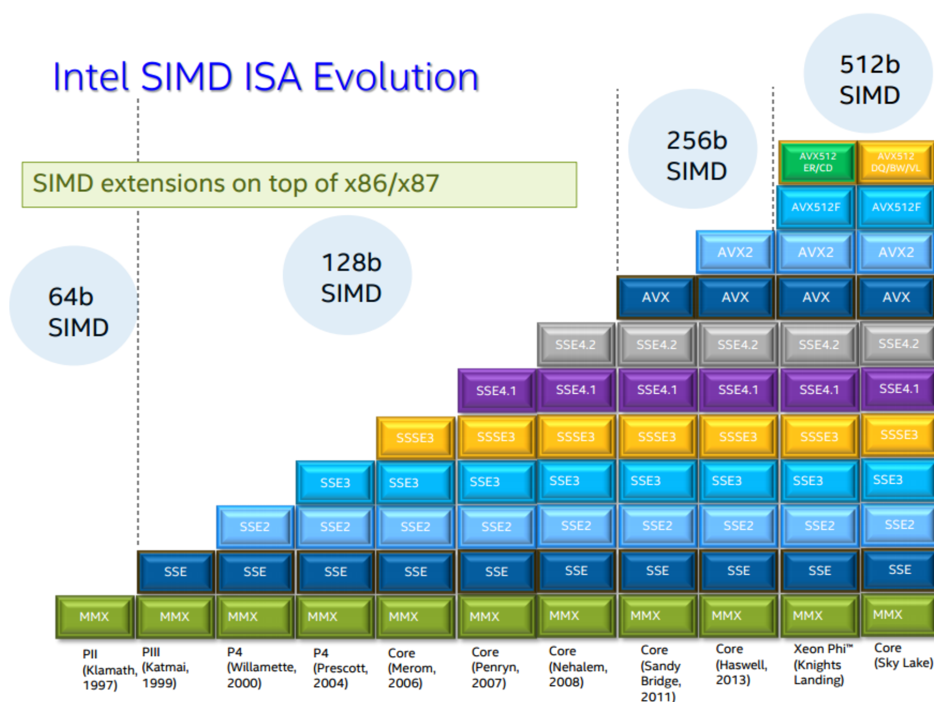

Рисунок 1.13 – Еволюція SIMD набору команд [21]

1.5.8 Векторні розширення в архітектурах IBM, ARM 8-A

Використання SIMD-команд є не тільки в процесорах фірм Intel і AMD. Технології SIMD-команд використовуються в процесорах фірм IBM, фірми ARM - в процесорах з архітектурою ARMv8-A [20]. Однак комп'ютери з процесорами фірми IBM поширені не так широко, як комп'ютери з процесорами фірм Intel і AMD. Тому реалізації SIMD-команд в процесорах фірми IBM в даній магістерській роботі не розглядається.

Аналогічна ситуація з розширеннями у вигляді SIMD-команд в процесорах фірми ARM. Тільки 22 серпня 2016 року фірма ARM Holdings анонсувала процесорну технологію Scalable Vector Extension (SVE), за допомогою якої британська компанія планує освоїти ринок

суперкомп'ютерів, кинувши виклик Intel і IBM [14]. SVE являє собою розширення для архітектури ARMv8-A, завдяки якому розробники процесорів зможуть вибирати потрібну довжину векторного регістра від 128 до 2048 розрядів з кроком в 128 розрядів. Оновлення підтримує модель програмування vector-length agnostic (VLA), що дозволяє адаптуватися до наявної довжини векторів. У фірмі ARM відзначають, що написаний з використанням SVE програмний код надалі не зажадає оптимізації. Залежно від апаратної реалізації процесу при проведенні розрахунків буде обрана оптимальна розрядність векторів. SVE призначений для створення серверних рішень та обладнання для високопродуктивних обчислень (HPC).

1.6 Аналіз методів і підходів для вирішення задачі програмування алгоритмів

Векторизація або векторна обробка масивів даних пов'язана з можливістю виконання команди над кількома операндами одночасно. При використанні векторизації можна в кілька разів підвищити швидкість обробки. Необхідність застосування векторизації видно з процесу розвитку процесорів фірми Intel. Команди класу SIMD над регістрами технології MMX (64 розряду) з'явилися в перших процесорах Pentium. Потім SIMD-команди знайшли свій розвиток в технологіях SSE (128 розрядів) і AVX (256 розрядів). В даний час з'являються процесори з SIMD-регістрами розрядністю 1024. Кількість SIMD-команд перевершує кількість команд обчислювального ядра процесора. SIMD-технологія підтримана як на низькому, так і на високому рівні програмування.

На низькому рівні застосовуються асемблерні вставки і векторні intrinsic функції бібліотеки типу xmmintrin.h. На високому рівні для векторизації використовують або спеціальні директиви, наприклад, бібліотеки OpenMP 4.0, або векторизацію бібліотеки immintrin.h при виборі необхідного рівня оптимізації компілятора. Так як стоїть задача розробити програми для вирішення поставлених завдань без використання засобів векторної обробки і з використанням засобів векторної обробки, то це означає, що потрібно використовувати як паралельне програмування так і послідовне.

Питання паралельного програмування, а також автоматизації розпаралелювання послідовних програм для певного класу паралельних архітектур описані в роботах [15].

Далі детально розглядається, які методи найкраще використовувати для поставленої задачі. Перетворення розпаралелювання зазвичай здійснюються у дві стадії. На першій — стадії аналізу, в процесі дослідження вхідної програми виявляється паралелізм алгоритму або завдання. Результат фіксується в машинно-незалежному вигляді. На другій — стадії синтезу, генерується паралельна програма, еквівалентна вхідній, відповідно до особливостей архітектури цільової паралельної системи.

На стадії аналізу проводиться виявлення прихованого паралелізму в вхідній послідовній програмі. Для цього використовуються методи виявлення залежностей між операційними об'єктами програми (залежностей з управління) і залежностей між інформаційними об'єктами програми (залежностей за даними).

Виявлення потенційного паралелізму послідовної програми ґрунтується на аналізі залежностей складових її частин одна від одної [6]. У якості подібних частин, в залежності від обраного масштабу розгляду — дрібнозернистого або грубозернистого, можуть розглядатися окремі оператори (інструкції), групи операторів присвоювання, блоки, ітерації циклу, умовні оператори, виконання процедур після виклику. Узята за основу модель крупнозернистого паралелізму, дозволяє вибрати в якості одиниці планування групи операторів, об'єднані в регіони [17].

Розглянемо різні типи відносин, можливі між регіонами g_i, $g_j \in G$ з позиції їх потенційного паралельного виконання:

– одночасність g_i b_{par} g_j, регіони g_i, g_j можуть виконуватися одночасно і звертатися до використовуваних елементів пам'яті в довільному порядку;

– упорядкованість g_i b_{ord} g_j, регіон g_i повинен вибрати все, що йому потрібно, перш ніж регіон g_j запише свої результати;

– консервативність g_i b_{con} g_j, регіон g_i повинен записати свої результати раніше, ніж g_j;

– послідовність g_i b_{seq} g_j, регіон g_i повинен бути завершений до початку g_j.

Виявлення відносин між регіонами дозволить висловити потенційний паралелізм послідовної програми таким чином, щоб це не вплинуло на коректність отриманого при її паралельному виконанні результату.

Більшість методів аналізу залежностей засновані на графічному поданні програми [3-8]. Вони виконують побудову графів залежностей за даними і по управлінню та діляться в свою чергу на дві великі групи: статичні і динамічні.

Статичні методи виконуються на етапі трансляції вхідного тексту програми [6-8]. Можливості статичних методів є обмеженими, тому що не завжди можливо виявити повністю всі інформаційні залежності між операторами, в зв'язку з тим, що при аналізі тексту програми ніколи не відомі значення змінних, які використовуються в ній. Крім того, в сучасних мовах програмування існує широкий набір засобів, що дозволяють здійснювати неявний (непрямий) доступ до інформаційних об'єктів. Прикладом може служити використання покажчиків та їх розіменування, організація доступу до елементів масивів за індексом, використання формальних параметрів, процедурних змінних, віртуальних методів класів. Все це суттєво ускладнює завдання аналізу потоку даних і знижує ефективність виявлення паралелізму статичними методами.

Динамічні методи досліджують програму на етапі її виконання [6-8]. Динамічний аналіз програм заснований на впровадженні в вихідну програму додаткових операторів, які проводять аналіз. Отримана програма виконується на деякому тестовому наборі вхідних даних (або декількох наборах), і під час виконання збирається інформація про фактичні залежності, присутні в програмі на даному конкретному наборі даних. Такий підхід дозволяє виробляти виявлення залежностей в багатьох ситуаціях, коли статичний аналіз неможливий. Оскільки аналіз відбувається під час виконання програми, аналізатору доступні значення всіх змінних, присутніх в програмі. Тому з'являється можливість проаналізувати будь-які складні й заплутані види залежностей. При цьому під динамічним розпаралелювання розуміється спосіб виявлення паралельних регіонів і планування їх виконання безпосередньо під час виконання (run-time) програми.

1.6.1 Метод спекулятивної багатопоточності

Найбільш підходящим методом динамічного розпаралелювання послідовних програм для багатоядерних обчислювальних систем зі спільною пам'яттю, з урахуванням зазначених вище недоліків, є метод спекулятивної багато поточності [6-8].

Суть методу полягає в наступному. Серед безлічі всіх регіонів n послідовної програми виявляються регіони які мають залежності, характер яких не може бути встановлений на етапі трансляції програми через неоднозначність. Метод спекулятивної багато поточності вказує на паралельне багато поточне виконання подібних регіонів, в розрахунку на те, що інформаційні залежності на стадії виконання не виявляться, вдаючись до позиції крайнього оптимізму (спекулюючи на удачу). У разі якщо подібні надії виправдають себе, буде отриманий виграш в продуктивності, в іншому випадку, результати обчислень регіону повинні бути анульовані, і він буде виконаний повторно, що призведе до накладних витрат. Регіони програми, до яких застосовується такий метод, будемо називати спекулятивними регіонами.

Для того, щоб організувати багато поточне виконання спекулятивних регіонів, формується динамічна послідовність стадій їх виконання, які називаються епохами $\forall e_i \in EP \mid i = \overline{1,N}, N = \mid EP \mid$. Наприклад, для циклічного регіону епохами є окремі ітерації циклу. Кожна епоха забезпечується локальним буфером пам'яті для збереження критичних до можливих порушень залежно даних.

Подібний підхід дозволяє фактично проводити обробку окремих ітерацій циклу в паралельному конвеєрному режимі, заповнюючи ступені конвеєра (ядра процесора) новими епохами в міру їх звільнення. Такий підхід мінімізує можливі втрати, навіть в разі повторного

виконання невдалих епох.

1.6.2 Функціональний підхід до паралельного програмування

Мови паралельного програмування, що використовують явне управління обчислювальним процесом, дають можливість окреслити максимальний паралелізм завдання, але забезпечують це не найзручнішим способом [17]. Це пояснюється наступними причинами:

– програміст сам повинен формувати всі паралельні фрагменти і стежити за коректною синхронізацією даних;

– використання в мовах такого типу «ручного» управління пам'яттю може привести до конфліктів між процесами в боротьбі за загальний ресурс (програмісту доводиться ретельно стежити за розподілом пам'яті або явно дотримуватися принципу єдиного присвоювання);

– при явному розпаралелювання в ході програмування не завжди адекватно можна відобразити в створюваній програмі паралелізм даних, властивий задачі, що, в свою чергу може привести до подальших перекручувань при перенесенні написаної програми на іншу архітектуру [11].

При розпаралелюванні послідовних програм дуже рідко забезпечується досягнення прийнятного рівня паралелізму через обмеження обчислювального методу, обраного програмістом, що часто обумовлюється стереотипами послідовного мислення. В реальній ситуації врахувати особливості різних паралельних систем виявилося набагато важче, ніж це спочатку передбачалося.

Створення прикладних паралельних програм, орієнтованих на обробку інформаційних потоків, зручніше здійснювати із застосуванням функціональних мов паралельного програмування, в яких виконання кожного оператора здійснюється по готовності його даних. Вони не вимагають явного опису паралелізму [17] завдання, який в цьому випадку визначається відповідно до інформаційних зв'язків. Використання такої мови дозволяє:

– створювати мобільні програми з паралелізмом на рівні операторів, обмеженим лише засобом для вирішення завдання;

– забезпечити перенесення програми на конкретну архітектуру без її зміни незалежно від числа доступних ядер;

– проводити оптимізацію програми за багатьма параметрами з урахуванням специфіки архітектури ВС, для якої здійснюється трансляція без урахування керуючих зв'язків програми.

Як інструмент для вирішення виникаючих проблем при розробці багато поточних додатків пропонується використовувати функціональну мову паралельного програмування С ++.

1.7 Постановка наукової задачі та обґрунтування методики досліджень

Дослідити ефективність використання засобів векторної обробки ядер сучасних процесорів:

- розглянути ряд типових завдань, для рішення яких необхідна висока продуктивність;
- вибрати кілька алгоритмів з лінійною алгеброю, на прикладі множення матриць, транспонування матриць, обернення матриць.
- розробити програми без використання засобів векторної обробки і з використанням засобів векторної обробки;
- порівняти дані про час виконання програм і кількості тактів процесору.

Після написання алгоритмів зробити висновки про ефективність виконання цих алгоритмів без використання засобів векторної обробки і з використанням засобів векторної обробки та порівняти результати.

1.8 Висновки до першого розділу

1. Аналізуючи дану тематику важливо відзначити процесори архітектури Intel підтримують векторні інструкції. Тобто існує можливість створення векторів різних типів і застосування цих інструкцій котрі працюють з векторами і отримують на виході вектор результатів. Це інструкції технології SSE, набір інструкцій розвивається, доповнюється новими командами і можливостями. Технології SSE2, SSE3, SSE4, AVX – розширення первісної ідеї. Проблема полягає в тому, що мови програмування спочатку скалярні і потрібно робити певні зусилля, щоб використовувати цю можливість обчислювальних ядер.

2. Векторизація — це модифікація коду, яка замінює скалярний код на векторний. Тобто скалярні дані упаковуються в вектора і скалярні операції замінюються на операції з векторами (пакетами).

3. Можна використовувати бібліотеки для програмування векторизації вручну, використовувати якийсь оптимізуючий компілятор з автовекторизацією або оптимізовані бібліотеки з утилітами, що використовують векторні інструкції. Існує маса обмежень на можливість застосування цієї модифікації. До того-ж є багато факторів, що визначають її вигідність. Тому програмування алгоритмів дозволить дослідити які технології покращать результати, оскільки оптимізація-це те, що покращує продуктивність. Щоб векторизація її поліпшувала, необхідно виконання деяких умов. Тобто векторизація – це всього лише «можливість». Потрібно попрацювати, щоб втілити цю можливість в реальні досягнення.

4. Наявність декількох обчислювальних ядер на сучасному процесорі дає можливість досягнення високої продуктивності програми розподілом обчислень між цими ядрами. Але і це всього лише «можливість». Наявність мільйона високопрофесійних програмістів не дає надії на те, що вони зможуть реалізувати оптимізуючий компілятор за годину. Перетворення однопотокового коду в багатопотоковий вимагає серйозних зусиль. Деякі оптимізуючи компілятори мають автопаралелізатор, який зводить процес створення багатопотокового додатка до процесу додавання однієї опції при компіляції. Але на шляху автопаралелізатора стільки підводних каменів, що в багатьох випадках оптимізації компілятора потрібна допомога, а в багатьох випадках він просто безсилий. Великих успіхів можна досягти використовуючи OpenMP директиви, але в більшості випадків потрібні зусилля для того, щоб розпаралелити спочатку одно потоковий алгоритм.

5. Аналізуючи проблематику даної роботи можна виділити три основних характеристики визначаючих продуктивність і дві великі можливості, які можуть бути реалізовані:

1) якість роботи підсистеми пам'яті;

2) кількість умовних переходів і якість роботи провісника переходів;
3) рівень інструкційного паралелізму;
4) використання векторизації;
5) використання багатопотокових обчислень.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ТИПОВИХ АЛГОРИТМІВ І РОЗРОБКА ПРОГРАМ

2.1 Опис способу організації пам'яті

При програмуванні алгоритму важливу роль відіграє організація пам'яті. Практично кожна операція розрахунку включає завантаження даних з пам'яті і вивантаження в неї результатів. Тому робота з пам'яттю повинна бути максимально ефективною. За замовчуванням використовується невирівняне виділення пам'яті, при якому блоки даних не вміщаються в межах машинних слів або зрушені щодо цих кордонів. Робота з невирівняними даними на багатьох операціях здійснюється істотно довше, ніж з вирівняними, а для деяких систем і команд вона неприпустима [26].

Зокрема, при використанні векторних інструкцій існують команди для вирівняних завантажених даних в регістри, так і для невирівняних, але при цьому невирівняне завантаження приблизно в 3 рази довше вирівняного. У зв'язку з цим, першим кроком для оптимізації коду є вирівняне виділення пам'яті під масиви даних. З використанням векторних регістрів пов'язана також проблема розмірності матриць, пов'язана з необхідністю завантаження в регістр на 2, 4 або 8 речових елементів подвійної точності даних, згрупованих по три елемента. Для ефективного завантаження в регістр розмірність матриць і векторів повинна бути 3x4 і 1x4 відповідно, щоб завантажувати їх по рядках. Цього можна досягти, якщо доповнити кожен рядок фіктивним четвертим елементом, рівним 0, який фактично не бере участі в операціях і не впливає на результат. Після цього рядок матриці або цілком поміщається у векторний регістр (для AVX), або займає два повних регістра меншого розміру (для SSE), можна також завантажувати в регістр два рядки одночасно.

2.2 Опис середовища розробки Microsoft Visual Studio

Microsoft Visual Studio це лінійка продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення і ряд інших інструментальних засобів. Цей продукти дозволяє розробляти як консольні додатки, так і додатки з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms.

Visual Studio включає в себе редактор вихідного коду з підтримкою технології IntelliSense і можливістю найпростішого рефакторингу коду. В програмі є вбудований відладчик він може працювати як відладчик рівня вихідного коду, так і відладчик машинного рівня. Інші вбудовані

інструменти включають в себе редактор форм для спрощення створення графічного інтерфейсу програми, веб-редактор, дизайнер класів і дизайнер схеми бази даних.

2.3 Визначення матриці

Матриця – це двовимірний масив, кожен елемент якого має два індекси, номер рядка – і і номер стовпця – j. Тому для роботи з елементами матриці необхідно використовувати два цикли. Якщо значеннями параметра першого циклу будуть номери рядків матриці, то значеннями параметра другого - стовпці (або навпаки).

2.4 Вступ множення матриць

Класичний алгоритм матричного множення є ітеративним і орієнтований на послідовне обчислення рядків результуючої матриці **C**. Відомо, якщо матриці А та В квадратні розміру ***n*** х ***n***, кожен елемент матриці С є скалярним множенням рядка і стовпця матриць А і В, тобто при множенні квадратних матриць розміру ***n*** х ***n*** кількість виконаних операцій має порядок $0(n^3)$.

Наразі продовжують інтенсивна розроблятися різні паралельні алгоритми множення матриць. У паралельних алгоритмах, виходячи з визначення операції матричного множення, обчислення всіх елементів матриці **C** може бути виконано незалежно один від одного, тобто використовується властивість паралелізму за даними [27]. Тому основний принцип організації паралельних обчислень полягає у використанні в якості базового потоку обчислення одного елемента результуючої матриці **C**, тобто обчислення над елементами одного рядка матриці А та одного стовпця матриці В. Очевидно, загальна кількість потоків одержуваних при такому підході, дорівнює n^2 що визначає кількість необхідних обчислювальних пристроїв.

2.4.1 Опис алгоритму множення матриць

Множення матриці А на матрицю В визначено, лише коли число стовпців першої матриці в добутку дорівнює числу рядків другої [27]. Тоді добутком матриць А розміру ***m*** х ***k*** і матриць В розміру ***k*** х ***n*** називається матриця С розміру ***m*** х ***n***, кожен елемент якої c_{ij} дорівнює сумі попарних добутків (рис. 2.1) елементів і-го рядка матриці А на відповідні елементи j-го стовпця матриці В, тобто:

$$c_{ij=}a_{i1}b_{1j}+a_{i2}b_{2j}+\ldots+a_{in}b_{nj}=\sum_{s=1}^{n}a_{is}b_{sj} \tag{2.1}$$

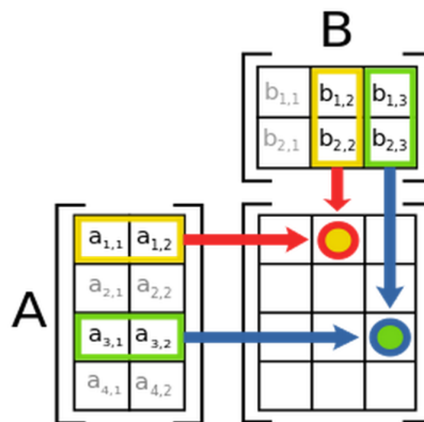

Рисунок 2.1 – Добуток матриць [28]

значення на перетинах зазначених кружечками будуть:

$$a_{1,2} = (a_{1,1}, a_{1,2}) \cdot (b_{1,1}, b_{1,2}) = a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \quad (2.2)$$

$$b_{3,3} = (a_{3,1}, a_{3,2}) \cdot (b_{1,3}, b_{2,3}) = a_{3,1}b_{1,3} + a_{3,2}b_{2,3} \quad (2.3)$$

Нехай кожна сума виду $a_1 + a_2 + \ldots + a_m$ буде скорочено позначатися як: $\sum_{i=1}^{m} a$

Якщо розглядається сума доданків a_{ij}, що залежать від двох індексів $i = 1, 2,\ldots,m$ і $j = 1, 2,\ldots,n$, то для її обчислення можна спочатку знайти суми елементів з фіксованим першим індексом всіх $i = 1, 2,\ldots, m$ і $j = 1, 2,\ldots,n$. Звернемо увагу на розміри матриці С, число рядків матриці-добутку збігається з числом рядків першої, а число стовпців - з числом стовпців другої з перемножуваних матриць (рис. 2.2).

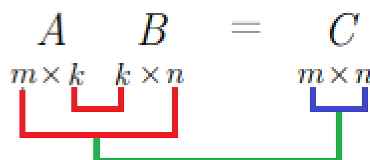

Рисунок 2.2 – Число рядків, число стовпців матриці С

2.4.2 Програмний алгоритм множення матриць

Розглянемо множення матриць різних розмірів що складаються з ***n*** рядків і ***m*** стовпців, заміримо час виконання, подивимось на кількість тактів процесору програмного алгоритму, опишемо результат. Тип даних матриць представляє дійсне число float у пам'яті воно займає 4 байта (32 біта). Очевидний програмний алгоритм множення матриць розміру ***n*** х ***m***, являє собою звичайне множення в циклі, його можна представити в такому виді:

```
for (int i = 0; i < n; ++i)
{
for (int j = 0; j < n; ++j)
{
m[i][j] = 0;
for (int k = 0; k < n; ++k)
{
m[i][j] += A[i][k] * B[k][j];
}}}
```

де *m[i][j]* – результуюча матриця, *A[i][k]*, *B[k][j]* – матриці які перемножуються і додаються між собою. Розглянемо реалізацію алгоритму без використання векторної обробки, алгоритм має вид:

```
// бібліотеки
#include <intrin.h>
// вихідна матриця
union matrix{
float m[4][4];
};
// стандартна реалізація множення матриць
void matrix_multiply_stand(matrix& out, const
matrix& A, const matrix& B){
matrix t; // отримуємо вихідну матрицю
// перемножуємо матриці між собою
for (int i = 0; i < 4; i++){
for (int j = 0; j < 4; j++){
t.m[i][j] = 0;
for (int k = 0; k < 4; k++){
t.m[i][j] += A.m[i][k] * B.m[k][j];
}}}
out = t;
}
// тестування функцій коду
static void run_standart(matrix* out, const
matrix* A, const matrix* B, int count){
int mask = 0; // виставляємо маску
for (int i = 0; i < count; i++){
int j = i & mask;
matrix_multiply_stand(out[j], A[j], B[j]);
}
}
// рандомна матриця виду 4x4, 8x8, 16x16,
32x32
static void random_matrix(matrix& M){
for (int i = 0; i < 4; i++){
for (int j = 0; j < 4; j++){
M.m[i][j] = (rand() - 16384.0f) / 1024.0f;
}}}
// головна програма
int main(void){
// запустимо час
clock_t start = clock();
static const struct {
const char* name;
void (*run)(matrix* out, const matrix* A, const
matrix* B, int count);
}
variants[] = { { "стандарт", run_standart }, };
matrix A, B, out, stand_out; // отримуємо
структуру
random_matrix(A); // рандомна матриця A
random_matrix(B); // рандомна матриця B
matrix_multiply_stand(stand_out, A, B); //
перемножуємо матриці
static const int nperfvars = (int)(sizeof(variants)
/ sizeof(*variants));
// у циклі перебираємо запущену матрицю
для обчислення
for (int i = 0; i < nperfvars; i++){
static const int nruns = 4096;
static const int muls_per_run = 4096;
unsigned long long best_time = ~0ull;
for (int run = 0; run < nruns; run++){
// заміримо час
unsigned long long time = __rdtsc();
variants[i].run(&out, &A, &B, muls_per_run);
time = __rdtsc() - time;
if (time < best_time){
best_time = time;
}}
// виведемо час
double cycles_run = (double)best_time /
(double)muls_per_run;
printf("%12s: %.2f тактів\n", variants[i].name,
cycles_run);
clock_t end = clock();
```

```
double seconds = (double)(end - start) /
CLOCKS_PER_SEC;
printf("\nThe time: %f seconds\n", seconds);}
return 0;}
```

Опишемо алгоритм наступним чином. Для початку визначимося з розмірністю масиву який поміщаємо в структуру *union matrix* в нашому випадку одинарні 32-бітні числа матриці з плаваючою точкою типу *float m[4][4]*, під них виділяємо пам'ять.

В функції matrix_multiply_stand() отримуємо посилання змінних, позначаються посилання символом амперсанда &, де:

– *const matrix& A* – посилання змінної матриці A;
– *const matrix& B* – посилання змінної матриці B;
– *matrix& out* – посилання змінної результуючої матриці.

Оголосимо змінну *matrix t*, в яку за пишемо результат нової матриці.

1) Для множення матриць використовуємо цикли *for()*:
   – множимо елемент першого рядка першої матриці *A.m[i][k]* на елемент першого стовпця другої матриці *B.m[k][j]*;
   – множимо другий елемент першого рядка першої матриці *A.m[i][k]* на другий елемент першого стовпця другої матриці *B.m[k][j]*;
   – робимо те ж саме з кожним елементом, поки не дійдемо до кінця як першого рядка першої матриці *A.m[i][k]*, так і першого стовпця другої матриці *B.m[k][j]*;
   – складемо отримані множення *t.m[i][j] += A.m[i][k] * B.m[k][j]*;
   – отриманий результат буде першим елементом першого рядка множення матриць.

2) Другий рядок матриці знаходиться аналогічно множенням елементів другого рядка першої матриці на елементи кожного стовпця другої матриці результати записуються в нову матрицю після кожного додавання, приклад наведено на рисунку 2.3.

3) Робимо це з кожним рядком першої матриці, поки всі рядки нової матриці не будуть заповнені [29].

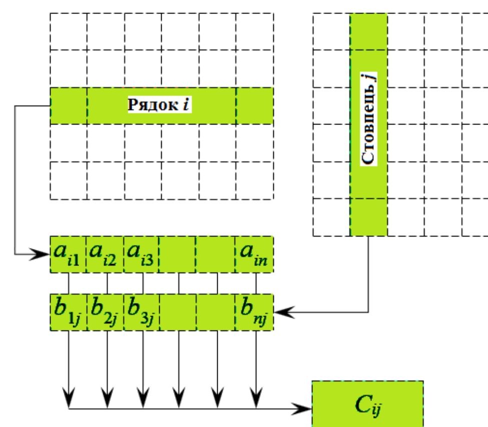

Рисунок 2.3 – Множення матриці A на матрицю B отриманням нової матриці C

Для запуску тестування алгоритму знадобиться ще одна функція назвем її *run_standart()* в ній отримуємо вказівник - змінних значення яких є адреса комірки виділеної пам'яті, вказівник позначається * зірочкою, де:

– *const matrix* A* – покажчик, матриці A;
– *const matrix* B* – покажчик, матриці B;
– *matrix* out* – покажчик, результуючої матриці;
– *int count* [30] - зміна в якій зберігається кількість циклів ітерації.

У зміну *int mask* встановлюємо маску для посилань яка дорівнює 0, запускаємо цикл *for()* і перебираємо змінну *int count* поки не дійдемо кінця ітерацій, змінній *int j* [30] присвоїмо маску яку ми встановили раніше, визвавши функцію *matrix_multiply_stand()* встановлюємо нову маску кожному посиланню тим самим зануляємо адресу виділеної пам'яті, це потрібно для того щоб в посиланні початкове значення було нульове.

В функції *random_matrix()* отримаємо посилання змінної матриці *matrix & M*. Посилання позначаються символом амперсанда &. Запускаємо цикли *for()* заповнюючи i-ті стовпці і j-ті рядки матриці *m[i][j]* рандомними числами, це потрібно для прискорення написання чисел матриць в автономному режимі, не даючи користувачеві заповнювати їх в ручну.

Переходимо в головну програму *main()*. В якій використовуємо всі описані вище функції. В структурі *static const struct* отримаємо вказівник - змінних значення яких є адреса комірки виділеної пам'яті:

– *const char* name* – покажчик, ім'я функції запуску тестування;
– *const matrix* A* – покажчик, матриці A;
– *const matrix* B* – покажчик, матриці B;
– *matrix* out* – покажчик, результуючої матриці;
– *perf_variants []* – массив із функцією *run_standart ()*.

У змінну *const int nperfvars* [30] записуємо розмір масиву *sizeof(perf_variants)* який беремо із структури *perf_variants[]* котрий зберігає в собі функцію запуск тестування *run_standart()*. Получивши структуру для матриць *matrix Aperf, Bperf, out*, заповнимо матрицю *Aperf* і матрицю *Bperf* рандомними числами функцією *random_matrix()*.

Запускаємо перший цикл *for()* в ньому перебираємо змінну *int nperfvars* [30] яка містить в собі функцію *run_standart()*, перебираючи цю змінну до тих пір поки не дійдемо кінця, в цьому циклі оголосимо змінні для тестування коду:

– *static const int nruns = 4096* – зміна в якій зберігається кількість прогону тестування всього коду;

– *static const int muls_per_run = 4096* – зміна в якій зберігається кількість ітерацій множення матриць;

– *unsigned long long best_time = ~0ull* – зміна в якій зберігається найкращий час виконання.

Запускаємо другий цикл *for()* в ньому перебираємо змінну *int nruns* [30] до тих пір поки не достигнемо значення котре вказано у цій змінній, оголосимо в циклі змінну *unsigned __int64 time = __rdtsc()* в якій зберігається час і кількість тактів циклу. Визвавши *perf_variants[i]* структуру передаючи в неї посилання змінних:

– *&out* – посилання змінної результуючої матриці;

– *&Aperf* – посилання змінної матриці A;

– *&Bperf* – посилання змінної матриці B;

– *muls_per_run* – кількість ітерацій множення матриць.

Запускаємо тестування і вимірюємо час виконання програми, кількість тактів процесору, та кількість циклів цього алгоритму, виведемо матрицю на екран за необхідністю.

Результат обчислення даного алгоритму, без використання векторної обробки, надано в розділі 3 (табл. 3.1).

Цей алгоритм можна оптимізувати використовуючи набір інструкції-AVX. В сучасних компіляторах є підтримка AVX - інструкцій без їх включення написана програма не буде працювати, при компіляції програм будуть виводитись різні помилки зв’язані з цим. При створенні нової програми за стандартом ці інструкції відключені, для роботи з інструкціями потрібно зайти в властивості компілятора і включити підтримку інструкцій для конкретного проекту (рис 2.4).

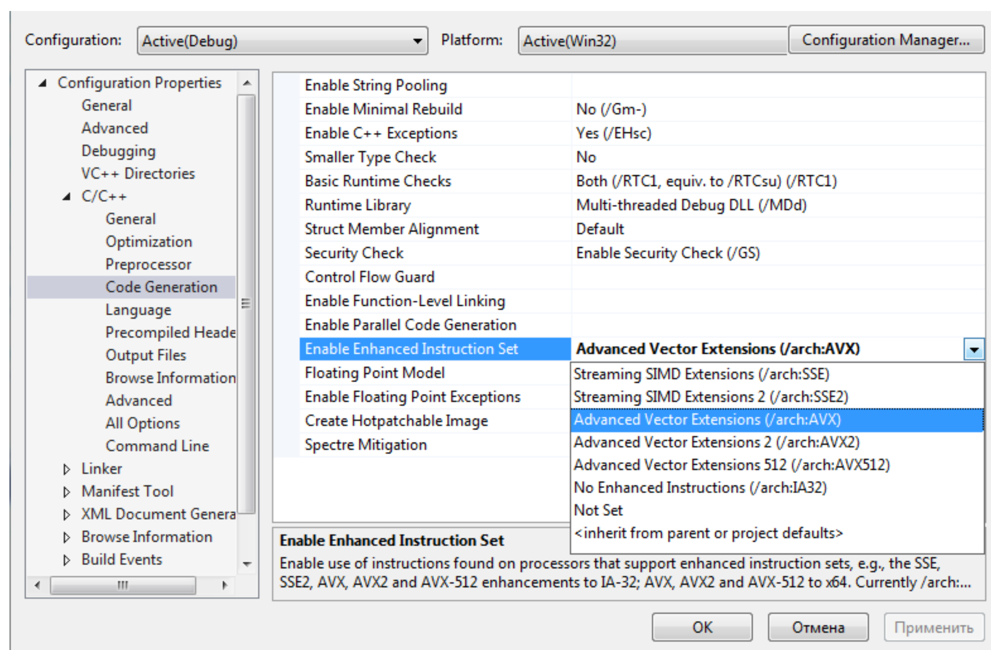

Рисунок 2.4 – Включення AVX інструкцій в сучасних компіляторах [31]

Перейдемо к програмній реалізації, використовуючи векторну обробку [32] інструкцій SIMD. Повна реалізація оптимізованого програмного алгоритму з використанням векторної обробки має вид:

```
// бібліотеки
#include <immintrin.h>
#include <intrin.h>
// вихідна матриця
union matrix{
float m[8][8];
__m128 row[8];
};
// лінійна комбінація, що використовує інструкції AVX за правилами XMM
static inline __m128 lincomb_AVX_4(const float* a, const matrix& B){
__m128 result;
result = _mm_mul_ps(_mm_broadcast_ss(&a[0]), B.row[0]);
result = _mm_add_ps(result, _mm_mul_ps(_mm_broadcast_ss(&a[1]), B.row[1]));
result = _mm_add_ps(result, _mm_mul_ps(_mm_broadcast_ss(&a[2]), B.row[2]));
result = _mm_add_ps(result, _mm_mul_ps(_mm_broadcast_ss(&a[3]), B.row[3]));
return result;
}
// використовуючи інструкції AVX, шириною 4, це буде швидше, якщо A знаходиться в пам'яті.
void matrix_multiply_AVX_4(matrix& out, const matrix& A, const matrix& B){
unsigned int i;
_mm256_zeroupper();
for (i = 0; i < sizeof(A.m) / sizeof(A.m[0]); i++){
__m128 outx = lincomb_AVX_4(A.m[0+i], B);
out.row[0 + i] = outx;
}}
static void run_AVX_4(matrix* out, const matrix* A, const matrix* B, int count){
// залишаємо алгоритм без зміни
...
}
// рандомна матриця виду 4x4, 8x8, 16x16, 32x32
static void random_matrix(matrix& M){
// залишаємо алгоритм без зміни
...
}
// головна програма
int main(int argc, char** argv){
static const struct {
const char* name;
void (*run)(matrix* out, const matrix* A, const matrix* B, int count);
}
perf_variants[] = { { "AVX_4", run_AVX_4 }, };
// все що нижче залишаємо без змін
...
}
return 0;
}
```

Опис алгоритму множення матриць з 128-бітними регістрами XMM, інструкцій SIMD. AVX інструкції досить легко задіяти безпосередньо в язик програмування C++ за допомогою вбудованих функцій (intrinsics). Для їх використання на початку програми підключаємо бібліотеку в заголовному файлі *#include <immintrin.h>*. Як і у звичайному алгоритмі без векторної обробки, визначимося з розмірністю масиву який поміщаємо в структуру *union matrix* в нашому випадку одинарні 32-бітні числа матриці з плаваючою точкою типу *float m[4][4]*, та одинарні 32-бітні числа інструкцій AVX *__m128 row[4]*, для всіх масивів виділяємо пам'ять.

Для початку треба змінити вищеописані функції під векторну реалізацію. Використовуючи лінійну комбінацію AVX інструкцій регістрів XMM, в функції *lincomb_AVX_4()*

отримуємо вказівник - змінної значення якої є адреса комірки виділеної пам'яті, та посилання на змінну:

- *const float* a* - покажчик, матриці А;
- *const matrix& B* - посилання змінної матриці В.

У параметр *__m128 result* зберігаємо результат множення матриць:

1) функцією *_mm_mul_ps* множимо упаковані одинарні 32-розрядні елементи *&a[0]*, *B.row[0]* з плаваючою точкою і зберігаємо результат в змінну *result* [1];

2) функцією *_mm_broadcast_ss* передаємо широкомовні одинарні 32-розрядні елементи *&a[0])*, *B.row[0]* з плаваючою точкою з пам'яті у всі елементи *result*;

3) функцією *_mm_add_ps* додаємо упаковані одинарні 32-розрядні елементи *&a[0])*, *B.row[0]* з плаваючою точкою і зберігаємо результати в змінну *result* [1];

4) коли перемножили матрицю повертаємо змінну *result*, для використання цих обчислень в інших функціях.

Легкими маніпуляціями над інструкціями ми зробили множення матриць, без всякого зусилля.

Об’явимо функцію *matrix_multiply_AVX_4()* в якій отримаємо посилання змінних:

- *const matrix& A* – посилання змінної матриці А;
- *const matrix& B* – посилання змінної матриці В;
- *matrix& out* – посилання змінної результуючої матриці.

Функцією *_mm256_zeroupper()* обнуляємо верхні 128-біт всіх регістрів YMM, нижні 128-біт регістрів не зміняться. Запускаємо цикл *for()* [33] в ньому перебираємо матрицю *A.m* получивши розмір функцією *sizeof()*, всередині циклу об’явимо змінну *__m128 outx* присвоїмо їй функцію множення матриць *lincomb_AVX_4()* в неї передаємо матрицю *A.m[0+i]* збільшуючи її на 1 у кожній ітерації, збільшення потрібно для того, щоб всі елементи матриці потрапили в цикл *for()* [33], і матрицю *B* її залишаємо без змін, якщо матриця *A.m[]* зберігається в пам'яті це може бути краще, ніж зберігання матриці в звичайному вигляді, так як вона вже знаходиться в регістрах. Присвоїмо вихідний матриці *out.row* значення змінної *__m128 outx* збільшуючи матрицю *out.row[0+i]* на 1 у кожній ітерації.

Як і для першого алгоритму без векторної обробки використовуємо такі самі функції, для всіх цих функцій алгоритм залишається аналогічним, їх описувати не будемо, міняючи тільки назву на нову якщо це потрібно:

- *run_standart ()* міняємо назву на *run_AVX_4()*;
- *random_matrix ()* залишаємо з такою ж назвою;
- *main ()* залишаємо з такою ж назвою.

Результат обчислення даного алгоритму, з використанням векторної обробки, надано в розділі 3 (табл. 3.1). Така реалізація помітно підвищує продуктивність і швидкість виконання

зменшуючи кількість циклів множення матриць. Алгоритм залишився колишнім лінійним, на відміну тим що ми використали набір інструкцій AVX який оперує з векторами розмірністю 128 біт. Це дозволило виконати 4 операцій для дійсних чисел з одинарною точністю за один такт.

Давайте спробуємо і цей алгоритм оптимізувати. Використовуючи подвійну лінійну комбінацію інструкцій AVX за правилами YMM [34]. Повна реалізація оптимізованого програмного алгоритму з використанням векторної обробки для подвійної комбінації має вид:

```
// бібліотеки
#include <immintrin.h>
#include <intrin.h>
// вихідна матриця
union matrix
{
float m[32][32];
__m128 row[32];
};
// подвійна лінійна комбінація з
використанням інструкцій AVX за
правилами YMM
static inline __m256
twolincomb_AVX_8(__m256 A01, const
matrix& B)
{
__m256 result;
result =
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0x00),
_mm256_broadcast_ps(&B.row[0]));
result = _mm256_add_ps(result,
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0x55),
_mm256_broadcast_ps(&B.row[1])));
result = _mm256_add_ps(result,
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0xaa),
_mm256_broadcast_ps(&B.row[2])));
result = _mm256_add_ps(result,
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0xff),
_mm256_broadcast_ps(&B.row[3])));
return result;
}
// це має бути помітно швидше з
фактичними 256-бітними широкими
векторними блоками (Intel);
void matrix_multiply_AVX_8(matrix& out,
const matrix& A, const matrix& B){
_mm256_zeroupper();  // перемножуємо
матриці між собою
__m256 A1 =
_mm256_loadu_ps(&A.m[0][1]);
__m256 A2 =
_mm256_loadu_ps(&A.m[2][3]);
__m256 A3 =
_mm256_loadu_ps(&A.m[4][5]);
__m256 A4 =
_mm256_loadu_ps(&A.m[6][7]);
__m256 A5 =
_mm256_loadu_ps(&A.m[8][9]);
__m256 A6 =
_mm256_loadu_ps(&A.m[10][11]);
__m256 A7 =
_mm256_loadu_ps(&A.m[12][13]);
__m256 A8 =
_mm256_loadu_ps(&A.m[14][15]);
__m256 A9 =
_mm256_loadu_ps(&A.m[16][17]);
__m256 A10 =
_mm256_loadu_ps(&A.m[18][19]);
__m256 A11 =
_mm256_loadu_ps(&A.m[20][21]);
__m256 A12 =
_mm256_loadu_ps(&A.m[22][23]);
__m256 A13 =
_mm256_loadu_ps(&A.m[24][25]);
__m256 A14 =
_mm256_loadu_ps(&A.m[26][27]);
__m256 A15 =
_mm256_loadu_ps(&A.m[28][29]);
__m256 A16 =
_mm256_loadu_ps(&A.m[30][31]);
__m256 out1 = twolincomb_AVX_8(A1, B);
__m256 out2 = twolincomb_AVX_8(A2, B);
__m256 out3 = twolincomb_AVX_8(A3, B);
__m256 out4 = twolincomb_AVX_8(A4, B);
__m256 out5 = twolincomb_AVX_8(A5, B);
__m256 out6 = twolincomb_AVX_8(A6, B);
__m256 out7 = twolincomb_AVX_8(A7, B);
__m256 out8 = twolincomb_AVX_8(A8, B);
__m256 out9 = twolincomb_AVX_8(A9, B);
__m256 out10 = twolincomb_AVX_8(A10, B);
```

```
__m256 out11 = twolincomb_AVX_8(A11, B);
__m256 out12 = twolincomb_AVX_8(A12, B);
__m256 out13 = twolincomb_AVX_8(A13, B);
__m256 out14 = twolincomb_AVX_8(A14, B);
__m256 out15 = twolincomb_AVX_8(A15, B);
__m256 out16 = twolincomb_AVX_8(A16, B);
_mm256_storeu_ps(&out.m[0][1], out1);
_mm256_storeu_ps(&out.m[2][3], out2);
_mm256_storeu_ps(&out.m[4][5], out3);
_mm256_storeu_ps(&out.m[6][7], out4);
_mm256_storeu_ps(&out.m[8][9], out5);
_mm256_storeu_ps(&out.m[10][11], out6);
_mm256_storeu_ps(&out.m[12][13], out7);
_mm256_storeu_ps(&out.m[14][15], out8);
_mm256_storeu_ps(&out.m[16][17], out9);
_mm256_storeu_ps(&out.m[18][19], out10);
_mm256_storeu_ps(&out.m[20][21], out11);
_mm256_storeu_ps(&out.m[22][23], out12);
_mm256_storeu_ps(&out.m[24][25], out13);
_mm256_storeu_ps(&out.m[26][27], out14);
_mm256_storeu_ps(&out.m[28][29], out15);
_mm256_storeu_ps(&out.m[30][31], out16);
}
static void run_AVX_8(matrix* out, const
matrix* A, const matrix* B, int count)
{
// залишаємо алгоритм без зміни
}
// рандомна матриця виду 4x4, 8x8, 16x16,
32x32
static void random_matrix(matrix& M)
{
// залишаємо алгоритм без зміни
}
// головна програма
int main(int argc, char** argv)
{
static const struct
{
const char* name;
void (*run)(matrix* out, const matrix* A, const
matrix* B, int count);}
perf_variants[] = { { "AVX_8", run_AVX_8 },
};
// все що нижче залишаємо без змін
...
}
return 0;
}
```

Опис алгоритму множення матриць з 256-бітними регістрами YMM, інструкцій SIMD. На початку програми підключаємо бібліотеку *#include <immintrin.h>*. Як і в другому алгоритмі з векторними інструкціями визначаємось з розміром масиву, який залишаємо без змін. Змінюємо вищеописані функції які були написані для 128-бітних регістрів на нові 256-бітних регістрів використовуючи векторну реалізацію.

В функції *twolincomb_AVX_8()* отримуємо змінну регістрів та посилання на змінну:

– *__m256 A01* – зміна, для регістрів;

– *const matrix& B*. – зміна, посилання на матрицю B.

У параметр *__m256 result* зберігаємо результат множення матриць, міняючи функції які використовували для 128-бітних регістрів на 256-бітні:

1) Функцією *_mm256_mul_ps()* множимо упаковані елементи з плаваючою точкою одинарної точності (32-розрядні) в *A01* і *B* і зберігаємо результати у змінну *result*.

2) Функцією *_mm256_shuffle_ps()* [1] перемішуємо одинарні прецизійні (32-розрядні) елементи з плаваючою точкою в межах 128-розрядних смуг руху за допомогою елемента управління в imm8 і зберігаємо результати у змінну *result*, параметрами функції є:

– A01 – матриця A;

– A01 – матриця B;

– 0x00 - елемент управління imm8.

3) Функцією *_mm256_broadcast_ps()* передаємо 128 біт з пам'яті що складається з 4 упакованих одно поточних (32-бітних) елементів з плаваючою точкою в усі елементи *result*.

4) Функцією *_mm256_add_ps()* [1] додаємо упаковані одно поточні (32-розрядні) елементи з плаваючою точкою в *A01* і *B* і зберігаємо результати в *result*.

5) Коли перемножили матрицю повертаємо змінну *result*.

Не важкими маніпуляціями над інструкціями з 256-біт зробили множення матриць, без всякого зусилля.

В функції *matrix_multiply_AVX_8()* отримуємо посилання змінних:

– *const matrix& A* – посилання змінної матриці A;

– *const matrix& B* – посилання змінної матриці B;

– *matrix& out* – посилання змінної результуючої матриці.

Для початку функцією *_mm256_zeroupper()* обнулимо верхні 128-біт всіх регістрів YMM, нижні 128-біт регістрів не зміняться. Кожній змінній *__m256 A1…An* завантажимо 256-бітні регістри що складаються з 8 упакованих одно розрядних (32-розрядних) елементів з плаваючою точкою з пам'яті в посилання *&A.m[0][1]*. *mem_addr* не потребує вирівнювання.

Вихідним змінним *__m256 out1…outn* присвоїмо подвійну лінійну комбінацію множення матриць *twolincomb_AVX_8()* в цю функцію передаємо змінні *A1…An*, а матрицю *B* залишимо без змін.

Зберігаємо 256-розрядні регістри що складаються з 8 упакованих одно розрядних (32-розрядних) елементів з плаваючою точкою змінні *out1…outn* в пам'ять. *mem_addr* не потребує вирівнювання [1].

Як і для другого алгоритму з векторною обробкою використовуємо такі самі функції, міняючи назву на нову якщо це потрібно:

– *run_AVX_4()* міняємо назву на *run_AVX_8()*;

– *random_matrix ()* залишаємо з такою ж назвою;

– *main ()* залишаємо з такою ж назвою.

для всіх цих функцій алгоритм залишається аналогічним, їх описувати не будемо.

Результат обчислення даного алгоритму, для 256-бітних регістрів з використанням векторної обробки, надано в магістерській роботі розділ 3 табл. 3. Така реалізація в рази швидше вище описаного методу, вона підвищує продуктивність і швидкість виконання зменшуючи кількість циклів множення матриць.

Алгоритм змінився на подвійну лінійну комбінацію, використовуючи інший набір інструкцій AVX який оперує з векторами розмірністю 256 біт. Це дозволило виконати 8 операцій для дійсних чисел з одинарною точністю за один такт.

2.5 Опис алгоритму транспортування матриць

Визначення. Перехід від матриці A до матриці A^T, в якій рядки і стовпці помінялися місцями зі збереженням порядку, називається транспонуванням матриці [35].

Матриця A^T є транспонованою до матриці A (рис. 2.5).

$$A^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix} A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Рисунок 2.5 – матриця A транспонована в матрицю A^T

З визначення випливає, що якщо матриця $A = (a_{ij})$ має розмір ***m*** x ***n***, то транспонована матриця $A^T = a_{ij}^t$ має розмір ***n*** x ***m***, причому матричні елементи $a_{ij}^t = a_{ji}$.

2.5.1 Загальні формули транспонування матриць

1) Подвійне транспонування повертає вихідну матрицю.

$$(A^T) = A^T \qquad (2.4)$$

2) Транспонування суми матриць еквівалентно сумі транспонованих доданків [36].

$$(A+B)^T = A^T + B^T \qquad (2.5)$$

3) Транспонування добутку двох матриць еквівалентно добутку транспонованих матриць, взятих у зворотному порядку.

$$(AB)^T = B^T A^T \qquad (2.6)$$

4) Добуток матриці на свою транспоновану

$$A^T A \text{ або } AA^T \tag{2.7}$$

завжди має результатом симетричну квадратну матрицю.

5) Якщо матриця A квадратна, то значення її визначника не залежить від транспонування.

$$D(A) = D(A^T). \tag{2.8}$$

2.5.2 Програмний алгоритм транспонування матриць

Розглянемо транспонування матриць де кількість рядків $\boldsymbol{m}$ менше кількості стовпців $\boldsymbol{n}$, заміримо час виконання програми, елементи матриці мають тип даних double, у пам'яті займають 8 байт (64 біта). Очевидна реалізація запрограмованої частини виглядає таким чином:

```
for (i = 0; i < m; i++)
{
for (j = 0; j < n; j++)
{
transposed[j][i] = source[i][j];
}}
```

де перший цикл *for()* [33] прохід по рядкам матриці, другий цикл *for()* [33] прохід по стовпцям матриці, в другому циклі виконується транспортування із матриці *source[i][j]* в матрицю *transposed[j][i]*. Повна реалізації програми без використання інструкцій-AVX має такий вид:

```
#include <stdio.h>
#include <time.h>
using namespace std;
#pragma intrinsic(__rdtsc)
#define M 32   // рядки
#define N 64   // стовпці
double* transpose_standart(double* a, int m,
int n){
double* b = new double[M * N];
for (int i = 0; i < m; ++i){
for (int j = 0; j < n; ++j){
b[j * m + i] = a[i * n + j];
}}
delete a;
return b;
}
// ініціалізація матриці
void init (double* a, int m, int n){
for (int i = 0; i < m; i++){
for (int j = 0; j < n; j++){
a[i * n + j] = i * n + j;
}}}
// виведення матриці
void print (double* a, int m, int n){
for (int i = 0; i < m; i++){
for (int j = 0; j < n; j++){
printf ("%4.llf", a[i*n + j]);
printf("\n");
}}}
int main (){
// виділяємо пам'ять під матрицю
double* a = new double[M*N];
// ініціалізація матриці
init (a, M, N);
printf ("\nsource matrix\n");
print (a, M, N);
```

```
// замір часу
unsigned __int64 time = __rdtsc();
clock_t start = clock(); // пуск часу
// транспонування без інструкцій AVX
a = transpose_standart(a, M, N);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
printf("\ntranspose matrix\n");
print(a, N, M);
clock_t end = clock(); зупинити час
printf("\nprocesor tacts : %I64d\n", time);
double seconds = (double)(end - start) /
CLOCKS_PER_SEC;
printf("\nThe time: %.2f seconds\n", seconds);
return 0;
}
```

На початку програми підключаємо бібліотеки в заголовному файлі *#include*. Об'являємо директиви для роботи з рядками і стовпцями матриці:

– *#define M 32* – рядки матриці;
– *#define N 64* – стовпці матриці.

Створюємо дійсну *double* функцію з покажчиком* *transpose_standart()*, у функцію передаємо вказівник – змінної, значення якої є адреса комірки виділеної пам'яті, і змінні стовпця та рядка:

– *double* a* – покажчик, дійсного числа матриці A;
– *int m* – змінна стовпця;
– *int n* – змінна рядка.

Виділяємо пам'ять дійсного числа для змінної *double* b*, це потрібно щоб зберегти рядки і стовпці у пам'ять. Запускаємо цикли *for()*, проходимо по рядкам *i < m* і стовпцям *j < n* поки не дійдемо кінця першого і другого циклу. У другому циклі *for()* [33]виконуємо транспонування матриць, де:

– *a[i * n + j]* – транспортована матриця *a[j][i]*;
– *b[j * m + i]* – отримана матриця при транспортуванні *b[i][j]*;
– *b[j * m + i] = a[i * n + j]* – транспортування матриці *a* в матрицю *b*.

Видаляємо покажчик матриці *a* функцією *delete a*, робимо це для звільнення пам'яті, повертаємо транспортовану матрицю *return b*.

Функція *init_matrix()* потрібна для ініціалізації матриці – це дозволить створити матрицю із *m* рядків і *n* стовпців заповнюючи її числами, якщо цього не зробити на екрані ми побачимо помилки.

Заходимо у функцію *main()*, виділяємо пам'ять для дійсного числа змінної *double* a*, це потрібно щоб зберегти рядки і стовпці у пам'ять. Робимо ініціалізацію матриці *init_matrix(a, M, N)*, виводимо матрицю на екран, тільки потрібно враховувати те, якщо будемо виводити ініціалізовану матрицю, час виконання програми збільшиться. Заміряємо час і кількість тактів який збережемо у зміну *unsigned __int64 time* [37], запускаємо функцію транспортування матриці *transpose(a, M, N)*, яку вже описали вище, рахуємо кількість тактів процесору і час виконання, виводимо матрицю на екран, враховуємо те, якщо будемо виводити транспортовану матрицю час виконання програми збільшиться, завершаємо роботу.

Результат обчислення транспонування матриць без використанням векторної обробки, наведено в розділі 3 (табл. 3.2). Використовуючи (intrinsic) векторну обробку, алгоритм можна оптимізувати наступним чином, реалізація алгоритму має вид:

```
#include <stdio.h>
#include <intrin.h>
#include <immintrin.h>
#include <time.h>
```

```
using namespace std;
#pragma intrinsic(__rdtsc)
#define K 4   // кількість рядків з матриці
#define M 8   // рядки
#define N 16  // стовпці
double* transpose_AVX (double* a, int m, int
n){
double* b = new double[M*N];
__m256d t[K], r[K];
// прохід по рядках
for (int i = 0; i < m; i += K){
// прохід по стовпцях
for (int j = 0; j < n; j += K){
// завантажуємо k рядків з матриці
for (int k = 0; k < K; k++){
r[k] = _mm256_load_pd (& (a [(i + k) * n +
j]));
}
t[0] = _mm256_unpacklo_pd(r[0], r[1]);
t[1] = _mm256_unpackhi_pd(r[0], r[1]);
t[2] = _mm256_unpacklo_pd(r[2], r[3]);
t[3] = _mm256_unpackhi_pd(r[2], r[3]);
// замінити старші половини строки 0 на
молодші половини 2 строки // 00 04 08 12
r[0] = _mm256_permute2f128_pd(t[0], t[2],
0x20);
// замінити старші половини строки 1 на
молодші половинки 3 строки // 01 05 09 13
r[1] = _mm256_permute2f128_pd(t[1], t[3],
0x20);
// замінити молодші половини строки 0 на
старшій половині 2 строки // 02 06 10 14
r[2] = _mm256_permute2f128_pd(t[0], t[2],
0x31);
// замінити молодші половини строки 1 на
старшій половині 3 строки // 03 07 11 15
r[3] = _mm256_permute2f128_pd(t[1], t[3],
0x31);
// запис результату
for (int k = 0; k < K; k++)
_mm256_storeu_pd (& (b [(j + k) *m + i]),
r[k]);
}
}
delete a;
return b;
}
// ініціалізація матриці
void init (double* a, int m, int n){
for (int i = 0; i < m; i++){
for (int j = 0; j < n; j++){
a[i * n + j] = i * n + j;
}}}
// виведення матриці
void print (double* a, int m, int n){
for (int i = 0; i < m; i++){
for (int j = 0; j < n; j++){
printf ("%4.llf", a[i*n + j]);
printf("\n");
}}}
int main (){
// виділяємо пам'ять під матрицю
double* a = new double[M*N];
// ініціалізація матриці
init (a, M, N);
printf ("\nsource matrix\n");
print (a, M, N);
// замір часу
unsigned __int64 time = __rdtsc();
clock_t start = clock(); // пуск часу
// транспонування з AVX інструкціями
a = transpose_AVX(a, M, N);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
printf("\ntranspose matrix\n");
print(a, N, M);
clock_t end = clock(); // зупинити час
printf("\nprocessor tacts : %I64d\n", time);
double seconds = (double)(end - start) /
CLOCKS_PER_SEC;
printf("\nThe time: %.2f seconds\n", seconds);
return 0;
}
```

Для роботи з інструкціями AVX, на початку програми підключаємо бібліотеку в заголовному файлі *#include <immintrin.h>*. Об’являємо директиви для роботи з матрицями:

- *#define K 8* – кількість рядків отримані із матриці;
- *#define M 32* – рядки матриці;
- *#define N 64* – стовпці матриці.

Створюємо дійсну *double* функцію з покажчиком* *transpose_AVX()*, у функцію передаємо

вказівник – змінної, значення якої є адреса комірки виділеної пам'яті, і змінні стовпця та рядка:

– *double* a* - покажчик, дійсного числа матриці А;
– *int m* – змінна стовпця;
– *int n* – змінна рядка.

Виділяємо пам'ять дійсного числа для змінної *double* b*, це потрібно щоб зберегти рядки і стовпці у пам'ять, завантажуємо у тимчасові масиви *__m256d t[K]*, *r[K]* кількість рядків із матриці, це дозволить отримати рядки матриці і оперувати над ними получаючи одночасно ту кількість рядків яка зберігається в директиві *#define K*.

Розбиваємо нашу велику матрицю на невеликі ділянки, які зручно поміщаються в векторні регістри. Завантажуємо дані в регістри, вивантажуємо з маскою за новими адресами. Маленький залишок транспонуємо в циклі. Запускаємо цикли *for()* [33], проходимо по рядкам $i < m$ і стовпцям $j < n$ поки не дійдемо кінця першого і другого циклу. У другому циклі *for()* [33] виконуємо транспонування матриць:

1) Функцією *_mm256_unpacklo_pd* розпакуємо та перемішуємо елементи з плаваючою точкою з нижньої половини кожної 128-бітної смуги у масиви *r[0]* та *r[1]* і зберігаємо результати у масив *t[0]*.

2) Функцією *_mm256_unpackhi_pd* [1] розпакуємо та з'єднуємо елементи з плаваючою точкою з верхньої половини кожної 128-бітної смуги у масиви *r[0]* та *r[1]* і зберігаємо результати у масив *t[0]*.

3) Функцією *_mm256_permute2f128_pd* [1] перемішуємо 128-бітні складені з 2 упакованих елементів з плаваючою точкою, вибраних imm8 з *t[0]* та *t[2],* і зберігаємо результати у масив *r[0]*, транспонуємо матрицю:

– заміняємо старші половини строки 0 на молодші половини 2 строки;
– заміняємо старші половини строки 1 на молодші половинки 3 строки;
– заміняємо молодші половини строки 0 на старшій половині 2 строки;
– заміняємо молодші половини строки 1 на старшій половині 3 строки.

Зберігаємо результат у 256-бітні складені з 4 упакованих подвійних елементів плаваючою точкою з пам'яті. mem_addr не потрібно вирівнювати.

Запускаєм цикл *for()* [33] в ньому проходимо по рядкам із матриці $k < K$, у змінну *r[k]* завантажимо 256-бітні складені з 4 упакованих елементів з пам'яті в *a[]*. mem_addr має бути вирівняний на 32-байтовій межі або може бути створено загальний виняток захисту [1].

Видаляємо покажчик матриці *a* функцією *delete a*, робимо це для звільнення пам'яті, повертаємо транспортовану матрицю *return b*.

Функція *init_matrix()* потрібна для ініціалізації матриці – це дозволить створити матрицю із *m* рядків і *n* стовпців заповнюючи її числами, якщо цього не зробити на екрані ми побачимо помилки.

Заходимо у функцію *main()*, виділяємо пам’ять для дійсного числа змінної *double* a*, це потрібно щоб зберегти рядки і стовпці у пам’ять. Робимо ініціалізацію матриці *init_matrix(a, M, N)*, виводимо матрицю на екран, тільки потрібно враховувати те, якщо будемо виводити ініціалізовану матрицю, час виконання програми збільшиться. Заміряємо час і кількість тактів який збережемо у зміну *unsigned __int64 time*, запускаємо функцію транспортування матриці *transpose_AVX(a, M, N)*, яку вже описали вище, рахуємо кількість тактів процесору і час виконання, виводимо матрицю на екран, враховуємо те, якщо будемо виводити транспортовану матрицю час виконання програми збільшиться, завершаємо роботу.

Результат обчислення транспонування матриць з використанням векторної обробки, наведено в розділі 3 (табл. 3.2).

2.6 Вступ звернення матриць

При вирішенні практичних завдань виникає необхідність вибору того чи іншого методу звернення матриць або його паралельної реалізації. Поява програмного забезпечення для обчислень загального призначення з використанням прискорювачів дозволило на ряді завдань, в тому числі обчислювальної лінійної алгебри, отримувати прискорення обчислень в десятки і сотні разів, в порівнянні з центральним процесором. Багато потоків можуть ефективно виконувати одночасно велике число простих арифметичних операцій, що характерно для мультиплікативних і адитивних операцій з векторами і матрицями. Разом з тим, послідовні операції і розгалуження, характерні для розкладання матриць на трикутні множники, виконуються повільніше прискорювачами, ніж ядрами універсальних процесорів.

2.6.1 Алгоритм звернення матриць

Для будь-якого числа $a \neq 0$ існує зворотне число a^{-1}, таке що $aa^{-1} = a^{-1}a = 1$. Аналогічне поняття вдається ввести і для деяких квадратних матриць. Нехай *A* - квадратна матриця n–го порядку, а *E* - одинична матриця того ж порядку [39].

Визначення. Матриця *B* називається правою зворотною до матриці *A*, якщо в результаті множення матриці *A* на матрицю *B* справа виходить одинична матриця того ж порядку, що і матриця *A*, тобто.

$$AB = E \tag{2.9}$$

Визначення. Матриця C називається лівою оберненою до матриці A, якщо в результаті множення матриці A на матрицю C зліва виходить одинична матриця того ж порядку, що і матриця A, тобто.

$$CA = E \tag{2.10}$$

Очевидно, що зворотні матриці B і C самі є квадратними і того ж порядку що і матриця A. Введення двох зворотних матриць (правої і лівої), а не однієї, як це має місце в разі звичайних чисел, пов'язано з відсутністю властивості комутативності при перемноженні матриць. Неважко переконатися в тому, що якщо визначені матриці B і C існують, то вони збігаються між собою, тобто [39]

$$C = B. \tag{2.11}$$

Дійсно, тому що порядки матриць B, C і E збігаються, то згідно $AE = EA = A$.

$$CE = C \text{ і } EB = B \tag{2.12}$$

Тоді, враховуючи властивість асоціативності при перемноженні матриць, отримуємо:

$$C = CE = C\,(AB) = (CA)\,B = EB = B\,. \tag{2.13}$$

Визначення. Квадратна матриця A^{-1} називається зворотною до матриці A, якщо в результаті множення матриць A на A^{-1} як справа, так і зліва, виходить одинична матриця того ж порядку, що і матриця A, тобто [39].

$$A{*}A^{-1} = A^{-1}{*}A = E. \tag{2.14}$$

Визначення. Квадратна матриця A, детермінант якої відмінний від нуля ($\det A \neq 0$), називається не виродженою, в іншому випадку (тобто коли $\det A = 0$), матриця називається виродженою.

Для існування зворотної матриці A^{-1} необхідно і достатньо, щоб матриця A була не виродженою, тобто, щоб $\det A \neq 0$. Нехай задана квадратна матриця (рис 2.6)

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

Рисунок 2.6 – Квадратна матриця

тоді зворотну до неї матрицю A^{-1} можна обчислити за формулою (рис 2.7):

$$A^{-1} = \frac{1}{\det A} \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix}^T = \frac{1}{\det A} \begin{pmatrix} A_{11} & A_{21} & \dots & A_{n1} \\ A_{12} & A_{22} & \dots & A_{n2} \\ \dots & \dots & \dots & \dots \\ A_{1n} & A_{n2} & \dots & A_{nn} \end{pmatrix}$$

Рисунок 2.7 – Формула зворотної матриці

де $A_{ij} = (-1)^{i+j} M_{ij}$ – алгебраїчне доповнення до елементу a_{ij}.

2.6.2 Алгоритм знаходження зворотної матриці

Для знаходження зворотної матриці необхідно виконати наступну послідовність дій [38]:

1) Знайти визначник A вихідної матриці A. Якщо det A = 0, то матриця A вироджена, отже, зворотна матриця A^{-1} не існує. Якщо det A ≠ 0, то матриця A невироджена і зворотна матриця A^{-1} існує.

2) Вибираєм перший зліва стовпець матриці, в якому є хоч одне відмінне від нуля значення.

3) Якщо саме верхнє число в цьому стовпці нуль, то змінюємо весь перший рядок матриці з іншим рядком матриці, де в цій колонці немає нуля.

4) Всі елементи першого рядка ділим на верхній елемент вибраного стовпця.

5) З решти рядків віднімаєм перший рядок, помножену на перший елемент відповідного рядка, з метою отримати першим елементом кожного рядка нуль.

6) Далі проводим таку ж процедуру з матрицею, що виходить з вихідної матриці після викреслювання першого рядка і першого стовпця.

7) Після повторення цієї процедури (n – 1) раз отримуємо верхню трикутну матрицю

8) Віднімаємо з передостаннього рядка останній рядок, помножений на відповідний коефіцієнт, з тим, щоб у передостанньому рядку залишилась лише 1-ця на головній діагоналі.

9) Повторюємо попередній крок для наступних рядків [38].

У підсумку отримуємо одиничну матрицю і рішення на місці вільного вектора (з ним необхідно проводити всі ті ж перетворення).

2.6.3 Програмний алгоритм звернення матриць

Розглянемо алгоритм, методу Гаусса-Жордана (зворотна матриця) розміру ***n*** x ***n***, що зберігається по стовпцях N, дані якого мають тип double, у пам'яті цей тип займає 8 байт (64 біта). Реалізуємо стандартний алгоритм:

```
#include <stdio.h>
#include <intrin.h>
using namespace std;
#pragma intrinsic(__rdtsc)
#define N 8 // матриця розміру 8x8
// виведення матриці
void print (double* a){
// прохід по рядках
for (int i = 0; i < N; i++){
// прохід по стовпцях
for (int j = 0; j < N; j++)
// виведення елемента матриці
printf ("%10.5llf", a[i * N + j]);
printf("\n");
}
printf("\n");
}
// ініціалізація матриці
void init (double* a){
// прохід по рядках
for (int i = 0; i < N; i++)
// прохід по стовпцях
for (int j = 0; j < N; j++){
// елемент дорівнює значенню свого індексу
в матриці
a[i * N + j] = (double)(i * N + j);
}}
// обмін рядками
// передаємо відразу вихідну і додаткову
матрицю
// і індекси поточного рядка і стовпця
int SwapLine (double* a, double* b, int i, int
j){
// запам'ятаємо вихідний рядок
// і перейдемо на наступну
int k = i++;
double tmp;     // тимчасовий
// пошук рядка з першим елементом не
дорівнює 0
// йдемо по рядках
for (; i < N; i++){
// порівняння з 0
if (a[i * N + j] != 0){
// цикл обміну рядків по всій довжині
for (j = 0; j < N; j++){
// спочатку для вихідного масиву
// беремо поточний елемент
tmp = a[k * N + j];
// пишемо на це місце їх іншого рядка
a[k * N + j] = a[i * N + j];
// в інший рядок з іншої
a[i * N + j] = tmp;
// те ж саме для додаткової матриці
tmp = b[k * N + j];
b[k * N + j] = b[i * N + j];
b[i * N + j] = tmp;
}
return i; // повертаємо індекс
}
}
// або помилку
return -1;
}
// алгоритм простого методу по Гаусса-
Жордана
double* SimpleInverse(double* a){
// додаткова матриця де буде результат
зворотної
double* b = new double[N * N];
// приводимо до одиничної
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++){
// головна діагональ в 1
if (i == j) b[i * N + j] = 1;
else b[i * N + j] = 0; // решта в 0
}
double dta = 0;
// прохід по стовпцях
for (int j = 0; j < N; j++){
// обмін рядками якщо є необхідність
if (a[j * N + j] == 0){
if (SwapLine (a, b, j, j) == -1)
return NULL; // у разі помилки вихід
}
// головну діагональ вихідної матриці
приводимо до 1
```

```
dta = a[j * N + j]; // беремо перший елемент
рядка
for (int i = 0; i < N; i++){
// ділимо весь рядок на взяті елемент
a[j * N + i] /= dta;
b[j * N + i] /= dta;
}
// прохід по рядках вниз привести до 0
// прямий хід
for (int i = j + 1; i < N; i++){
// беремо коефіцієнт
dta = a[i * N + j];
for (int k = 0; k < N; k++)
{
// віднімаємо рядок помножену на
коефіцієнт
a[i * N + k] -= dta * a[j * N + k];
b[i * N + k] -= dta * b[j * N + k];}}}
// зворотний хід приводимо до 0 вище
головної діагоналі
// прохід по стовпцях
for (int j = N - 1; j >= 0; j--){
// прохід по рядках вгору
for (int i = j - 1; i >= 0; i--){
// беремо коефіцієнт
dta = a[i * N + j];
for (int k = 0; k < N; k++){
// віднімаємо рядок помножену на
коефіцієнт
a[i * N + k] -= dta * a[j * N + k];
b[i * N + k] -= dta * b[j * N + k];}}}
return b; // повертаємо зворотну матрицю
}
// головна програма
int main ()
{
// виділяємо під матрицю
double* a = new double [N * N];
init(a);
printf ("\nsource matrix\n");
print(a);
// замір часу
unsigned __int64 time = __rdtsc ();
a = SimpleInverse(a);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
printf ("\nsimple inverse matrix\n");
print(a);
printf ("\nrdtsc: %I64d", time);
getchar ();
return 0;
}
```

Перейдемо до опису програмного алгоритму опишемо його наступним чином. На початку програми підключаємо бібліотеки в заголовному файлі *#include*. Об’являємо директиву для роботи з зворотною матрицею розміру 8x8 - *#define N 8* матриця.

Створюємо функцію *print()* - яка потрібна для друку матриці на екран, в дужках передаємо вказівник змінної *double* a* – значення якої є адреса комірки виділеної пам'яті. В середині цієї функції проходимо по рядкам *N* і стовпцям *N*, і друкуємо матрицю на екран якщо це потрібно.

Створюємо функцію *init()* – яка потрібна для ініціалізація матриці, в дужках передаємо вказівник змінної *double* a* – значення якої є адреса комірки виділеної пам'яті. В середині цієї функції проходимо циклом *for()* [33] по рядкам *N* і стовпцям *N*, елементи яких дорівнюють значенню свого індексу, це дозволить створити матрицю із *N* рядків і *N* стовпців заповнюючи її рандомними числами, без цієї функції прийдеться в ручну створювати матрицю що є не дуже хорошим тоном.

Створюємо функцію *SwapLine()* – котра потрібна для обміну рядками, в дужках передаємо відразу вихідну *double* a* і додаткову *double* b* матрицю, і індекси поточного рядка *int i* [30] і стовпця *int j* [30]. В середині цієї функції робимо такі дії. Запам’ятовуємо вихідний рядок *int k* і переходимо на наступний, створюємо зміну для тимчасового значення *double tmp*. Переходимо у перший цикл *for()* [33] робимо пошук рядка з першим елементом який не дорівнює нулю,

проходимо по рядках *N* перевіряючи чи не дорівнює значення рядка 0. Переходимо у другий цикл *for()* в ньому робимо обмін рядків по всій довжині:

1) спочатку для вихідного масиву беремо поточний елемент;
2) пишемо на це місце їх іншого рядка;
3) в інший рядок з іншої і так далі;
4) те ж саме робимо для додаткової матриці;
5) наприкінці цієї функцію повертаємо індекс, або помилку.

Створюємо функцію *SimpleInverse()* – алгоритму простого методу по Гаусса-Жордана, в дужках передаємо вказівник змінної *double* a* – значення якої є адреса комірки виділеної пам'яті. В середині цієї функції робимо наступні дії.

Для початку виділимо пам’ять для змінної *double* b*, це буде додаткова матриця куди збережемо результат зворотної матриці.

Наступним шагом проводимо додаткову матрицю до одиничного значення, якщо рядок дорівнює стовпцю тоді головна діагональ знаходиться в одиниці а решта в нуль.

Створюємо пусту зміну *double dta*, в циклі проходимо по стовпцях, робимо обмін рядками якщо є необхідність, у разі помилки завершаємо роботу. Головну діагональ вихідної матриці приводимо до одиниці, беремо перший елемент рядка *dta*, ділимо весь рядок на узяті елементи:

1. a[j * N + i] /= dta;
2. b[j * N + i] /= dta.

Проходимо по рядках, нижній рядок приводимо до нуля, беремо коефіцієнт віднімаємо рядок помножену на коефіцієнт:

3. a[i * N + k] -= dta * a[j * N + k];
4. b[i * N + k] -= dta * b[j * N + k].

Зворотний хід вище головної діагоналі приводимо до 0, в циклі *for()* [33] проходимо по стовпцях, проходимо по рядкам вгору, беремо коефіцієнт *dta* далі в циклі віднімаємо рядок помноженої матриці на коефіцієнт:

5. a[i * N + k] -= dta * a[j * N + k];
6. b[i * N + k] -= dta * b[j * N + k];
7. наприкінці функції повертаємо зворотну матрицю *return b*.

Створюємо головну функцію *main()*, виділяємо пам’ять для дійсного числа змінної *double* a*, це потрібно щоб зберегти рядки і стовпці у пам’ять. Робимо ініціалізацію матриці *init (a)*, виводимо матрицю на екран, тільки потрібно враховувати те, якщо будемо виводити ініціалізовану матрицю, час виконання програми збільшиться. Заміряємо час і кількість тактів який збережемо у зміну *unsigned __int64 time* [37], запускаємо функцію *SimpleInverse()* методу по Гаусса-Жордана зворотної матриці, рахуємо кількість тактів процесору і час виконання,

виводимо матрицю на екран якщо вона не велика, враховуємо те, якщо буде виводитись зворотна матриця час виконання програми збільшиться, завершаємо роботу.

Результат обчислення зворотної матриці без використання векторної обробки і автовекторизації компілятором, наведено у розділі 3 табл. 3.3. Дивлячись на результати цього алгоритму, можна зробити висновок що цей алгоритм можна оптимізувати використовуючи векторну обробку і автовекторизацію. Повна реалізація цього алгоритму має вид:

```
#include <stdio.h>
#include <intrin.h>
#include <immintrin.h>
using namespace std;
#pragma intrinsic(__rdtsc)
#define K 4 // зміщення у рядка матриці на 4
double числа
#define N 8 // матриця розміру 8x8
// виведення матриці
void print (double* a){
// прохід по рядках
for (int i = 0; i < N; i++){
// прохід по стовпцях
for (int j = 0; j < N; j++)
// виведення елемента матриці
printf ("%10.5llf", a[i * N + j]);
printf("\n");
}
printf("\n");
}
// ініціалізація матриці
void init (double* a){
// прохід по рядках
for (int i = 0; i < N; i++)
// прохід по стовпцях
for (int j = 0; j < N; j++){
// елемент дорівнює значенню свого індексу
в матриці
a[i * N + j] = (double)(i * N + j);
}
}
// обмін рядками
// передаємо відразу вихідну і додаткову
матрицю
// і індекси поточного рядка і стовпця
int AVXSwapLine (double* a, double* b, int i,
int j){
int k = i++; // запам'ятаємо рядок
__m256d row1, row2;
for (; i < N; i++){
if (a[i * N + j] != 0){
for (j = 0; j < N; j += K){
// беремо по 4 double з матриці для
вихідного рядка
row1 = _mm256_load_pd(&(a[k * N + j]));
// і для другої
row2 = _mm256_load_pd(&(a[i * N + j]));
// міняємо місцями в пам'яті матриці 4
double
_mm256_storeu_pd(&(a[k * N + j]), row2);
_mm256_storeu_pd(&(a[i * N + j]), row1);
// теж саме для додаткової матриці
row1 = _mm256_load_pd(&(b[k * N + j]));
row2 = _mm256_load_pd(&(b[i * N + j]));
// міняємо місцями в пам'яті матриці
_mm256_storeu_pd(&(b[k * N + j]), row2);
_mm256_storeu_pd(&(b[i * N + j]), row1);
}
return i;
}}
return -1;
}
// алгоритм для avx
// аналогічно як для простого методу, за
винятком
// операцій прискорюють виконання ~ в 4
рази
double* AVXInverse (double* a){
double* b = new double [N * N];
__m256d row1, row2, row3, row4, tmp;
// приводимо до одиничної матриці
додаткову
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++){
if (i == j) b[i * N + j] = 1;
else b[i * N + j] = 0;
}
double dta = 0;
// прохід по стовпцях
for (int j = 0; j < N; j++){
// обмін рядками
if (a[j * N + j] == 0)
{
if (AVXSwapLine (a, b, j, j) == -1)
```

```
return NULL;
}
// приведення на головній діагоналі вихідної матриці до 1
dta = a[j * N + j]; // делитель
// виставити всі 4 double в дільник
tmp = _mm256_set1_pd(dta);
// прохід по всьому рядка зі зміщенням в 4
for (int i = 0; i < N; i += K){
// беремо з вихідної матриці
row1 = _mm256_load_pd(&(a[j * N + i]));
// беремо значення з додатковою
row2 = _mm256_load_pd(&(b[j * N + i]));
// ділимо рядки на взятий коефіцієнт
row1 = _mm256_div_pd(row1, tmp);
row2 = _mm256_div_pd(row2, tmp);
// пишемо назад в пам'ять
_mm256_storeu_pd(&(a[j * N + i]), row1);
_mm256_storeu_pd(&(b[j * N + i]), row2);
}
// прямий хід, нижче головної діагоналі вихідної в 0
// додаткова тільки реагує на дії над вихідною матрицею
for (int i = j + 1; i < N; i++){
// беремо коефіцієнт
dta = a[i * N + j];
// заповнюємо коефіцієнтом всі 4
tmp = _mm256_set1_pd(dta);
// проходимся по рядку
for (int k = 0; k < N; k += K){
// беремо з матриць значення по 4
row1 = _mm256_load_pd(&(a[j * N + k]));
row2 = _mm256_load_pd(&(b[j * N + k]));
// множимо на коефіцієнт
row1 = _mm256_mul_pd(row1, tmp);
row2 = _mm256_mul_pd(row2, tmp);
// беремо значення куди будемо писати в інший рядок результати
row3 = _mm256_load_pd(&(a[i * N + k]));
row4 = _mm256_load_pd(&(b[i * N + k]));
// віднімаємо множне
row3 = _mm256_sub_pd(row3, row1);
row4 = _mm256_sub_pd(row4, row2);
// і пишемо назад результати
_mm256_storeu_pd(&(a[i * N + k]), row3);
_mm256_storeu_pd(&(b[i * N + k]), row4);
}}}
// зворотній хід вище головної діагоналі
// прохід по стовпцях
for (int j = N - 1; j >= 0; j--){
// прохід по рядках вгору привести до 0
for (int i = j - 1; i >= 0; i--){
// беремо коефіцієнт
dta = a[i * N + j];
// наводимо всі 4 коефіцієнтом
tmp = _mm256_set1_pd(dta);
for (int k = 0; k < N; k += K){
// беремо рядки які будемо віднімати
row1 = _mm256_load_pd(&(a[j * N + k]));
row2 = _mm256_load_pd(&(b[j * N + k]));
// множимо на коефіцієнт
row1 = _mm256_mul_pd(row1, tmp);
row2 = _mm256_mul_pd(row2, tmp);
// значення для віднімання
row3 = _mm256_load_pd(&(a[i * N + k]));
row4 = _mm256_load_pd(&(b[i * N + k]));
// віднімати
row3 = _mm256_sub_pd(row3, row1);
row4 = _mm256_sub_pd(row4, row2);
// пишемо назад
_mm256_storeu_pd(&(a[i * N + k]), row3);
_mm256_storeu_pd(&(b[i * N + k]), row4);
}}}
return b; // повертаємо зворотну матрицю
}
// головна програма
int main(){
// виділяємо під матрицю
double* a = new double [N * N];
init(a);
printf ("\nsource matrix\n");
print(a);
// замір часу
time = __rdtsc ();
a = AVXInverse(a);
// розрахунок часу виконання в тактах
time = __rdtsc () - time;
printf ("\avx inverse matrix\n");
print(a);
printf ("\nrdtsc: %I64d", time);
getchar ();
return 0;
}
```

Алгоритм з використанням векторних інструкцій AVX виглядає наступним чином. На початку програми підключаємо бібліотеку для роботи з інструкціями, в заголовному файлі

#include *<immintrin.h>*. Так як і в звичайному алгоритмі без використання AVX інструкцій об’явимо директиви добавивши ще одну:

1. #define K 4 - зміщення рядка матриці на 4 double числа;
2. #define N 8 - матриця розміру 8x8.

Директива *K* знадобиться для зміщення рядка у матриці на 4 double числа, цим виграємо за часом зменшивши такти процесора в 4 рази. При збільшенні цього параметра директиви, результат прискорення алгоритму не збільшиться, так як AVX інструкції працюють з double числами і можуть оперувати тільки з 4 елемента в пам'яті, якщо ми будемо працювати з типом числа int, інструкції AVX зможуть одночасно оперувати з 32 бітними числами в пам'яті.

Створюємо функції *print()* і функцію *init()*, вони аналогічні стандартному алгоритму залишаються без змін.

Створюємо функцію *AVXSwapLine ()* – котра потрібна для обміну рядками, в дужках передаємо відразу вихідну *double* a* і додаткову *double* b* матрицю, і індекси поточного рядка *int i* і стовпця *int j* [30]. В середині цієї функції запам’ятовуємо вихідний рядок *int k* [30] і переходимо на наступний, створюємо зміну для тимчасового значення *__m256d row1, row2*.

Переходимо у перший цикл *for()* [33] робимо пошук рядка з першим елементом який не дорівнює нулю, проходимо по рядках *N* перевіряючи чи не дорівнює значення рядка 0. Переходимо у другий цикл *for ()* [33] в ньому робимо обмін рядків по всій довжині:

1. беремо по 4 double з матриці для вихідного рядка *row1* і для другого рядка *row2*;
2. міняємо місцями в пам'яті матриці 4 double числа;
3. те ж саме робимо для додаткової матриці;
4. міняємо місцями в пам'яті матрицю;
5. наприкінці цієї функцію повертаємо індекс, або помилку.

Створюємо функцію *AVXInverse()* – алгоритму простого методу по Гаусса-Жордана, в дужках передаємо вказівник змінної *double* a* – значення якої є адреса комірки виділеної пам'яті. В середині цієї функції робимо наступні дії.

Для початку виділимо пам’ять для змінної *double* b*, це буде додаткова матриця куди збережемо результат зворотної матриці. Створимо тимчасові зміні *__m256d row1, row2, row3, row4, tmp*. Наступним шагом проводимо додаткову матрицю до одиничного значення, якщо рядок дорівнює стовпцю тоді головна діагональ знаходиться в одиниці а решта в нуль.

Створюємо дільник *double dta*, виставляєм всі 4 double числа в дільник *tmp = _mm256_set1_pd(dta)* [1], в циклі *for()* проходимо по всьому рядку зі зміщенням в 4 елемента. беремо значення з вихідної матриці *row1* беремо значення з додаткової матриці *row2* ділимо рядки на взятий коефіцієнт і записуємо назад у пам’ять:

1. row1 = _mm256_load_pd(&(a[j * N + i])) – вихідна матриця;

2. row2 = _mm256_load_pd(&(b[j * N + i])) – додаткова матриця;
3. row1 = _mm256_div_pd(row1, tmp) – рядки які діляться на коефіцієнт;
4. row2 = _mm256_div_pd(row2, tmp) – рядки які діляться на коефіцієнт;
5. _mm256_storeu_pd(&(a[j * N + i]), row1) – пишемо назад у пам'ять;
6. _mm256_storeu_pd(&(b[j * N + i]), row2) – пишемо назад у пам'ять.

В циклі *for()* робимо прямий хід, нижче головної діагоналі вихідної матриці в нуль, додаткова матриця реагує тільки на дії над вихідною матрицею. беремо коефіцієнт і заповнюємо коефіцієнтом всі 4 елемента матриці функцією *_mm256_set1_pd()*, заходимо в цикл *for()* проходимо по рядкам кожного елемента, і беремо з матриць значення по 4 елемента:

1. row1 = _mm256_load_pd(&(a[j * N + k]));
2. row2 = _mm256_load_pd(&(b[j * N + k])).

множимо на коефіцієнт елементи,

3. row1 = _mm256_mul_pd(row1, tmp);
4. row2 = _mm256_mul_pd(row2, tmp);

беремо значення куди будемо писати інший рядок результату

5. row3 = _mm256_load_pd(&(a[i * N + k]));
6. row4 = _mm256_load_pd(&(b[i * N + k]));

віднімаємо множене

7. row3 = _mm256_sub_pd(row3, row1);
8. row4 = _mm256_sub_pd(row4, row2);

і пишемо назад результати.

Далі робимо зворотний хід вище головної діагоналі приводимо до 0, в циклі *for()* [33] проходимо по стовпцях, проходимо по рядкам вгору приводимо значення до нуль, беремо коефіцієнт *dta* наводимо всі 4 коефіцієнти елементів, в циклі *for()* [33] беремо рядки які будемо віднімати

1. row1 = _mm256_load_pd(&(a[j * N + k]));
2. row2 = _mm256_load_pd(&(b[j * N + k]));

множимо на коефіцієнт взятих елементів

3. row1 = _mm256_mul_pd(row1, tmp);
4. row2 = _mm256_mul_pd(row2, tmp);

значення нижче для віднімання

5. row3 = _mm256_load_pd(&(a[i * N + k]));
6. row4 = _mm256_load_pd(&(b[i * N + k]));

віднімаємо значення

7. row3 = _mm256_sub_pd(row3, row1);

8. row4 = _mm256_sub_pd(row4, row2);

пишемо результат назад у зміні

9. _mm256_storeu_pd(&(a[i * N + k]), row3);

10. _mm256_storeu_pd(&(b[i * N + k]), row4);

наприкінці функції повертаємо зворотну матрицю *return b*.

Створюємо головну функцію *main()*, вона буде аналогічною до стандартного алгоритму.

Результат обчислення зворотної матриці з використанням векторної обробки і автовекторизації наведено у розділі 3 (табл. 3.3).

2.7 Висновки до другого розділу

1. При програмуванні алгоритму важливу роль відіграє організація пам'яті. Практично кожна операція розрахунку включає завантаження даних з пам'яті і вивантаження в неї результатів. Тому робота з пам'яттю повинна бути максимально ефективною. За замовчуванням використовується невирівняне виділення пам'яті, при якому блоки даних не вміщаються в межах машинних слів або зрушені щодо цих кордонів. Робота з невирівняними даними на багатьох операціях здійснюється істотно довше, ніж з вирівняними, а для деяких систем і команд вона неприпустима.
2. В якості програми для розробки було обрано середовище Microsoft Visual Studio. Microsoft Visual Studio це лінійка продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення і ряд інших інструментальних засобів. Цей продукти дозволяє розробляти як консольні додатки, так і додатки з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms.
3. Матриця – це двовимірний масив, кожен елемент якого має два індекси, номер рядка – i і номер стовпця – j. Тому для роботи з елементами матриці необхідно використовувати два цикли. Якщо значеннями параметра першого циклу будуть номери рядків матриці, то значеннями параметра другого - стовпці (або навпаки).
4. Згідно з наведеними визначеннями алгоритмів лінійної алгебри, таких як множення, транспонування, звернення матриць були розроблені програми зі стандартним методами та із застосуванням AVX - інструкцій.

Підводячи підсумки другого розділу, було описано спосіб організації пам’яті, обране середовище розробки, описані програмі алгоритми, завірено кількість тактів, час виконання програм для різних реалізацій алгоритмів, результати яких представлені у розділі 3.

РОЗДІЛ 3
АНАЛІЗ РЕЗУЛЬТАТІВ ОБЧИСЛЮВАЛЬНИХ ЕКСПЕРИМЕНТІВ

3.1 Результати експериментів

Перш ніж приступити до написання програм, були обрані методи і алгоритми які були запрограмовані у другому розділі. Тести проводились на різних версіях компілятору запуску x86/x64 програми Microsoft Visual Studio, з різними розмірами матриць $\boldsymbol{n}$ x $\boldsymbol{m}$.

Враховуючи результати, отримані від використання розширених наборів інструкцій, було вирішено оцінити ефективність та швидкість виконання подібних оптимізацій в рамках роботи множення, транспонування, звернення матриць.

Використовуючи векторну обробку були розроблені алгоритми. В якості критерію оцінки використовувався час, витрачений на виконання певних розрахунків над матрицями різних розмірностей, завантажених в регістри а також підрахунок кількості тактів процесору. Дані розрахунки включали в себе:

1. розробку програм без використання засобів векторної обробки;
2. розробку програм з використанням засобів векторної обробки;
3. порівняння даних про час виконання програм і кількості тактів процесору;
4. зробити висновки про ефективність використання векторної обробки в комп'ютерах з універсальними процесорами.

Вибір розрахунків був обумовлений тим, що вони в основному оперують даними, представляють із себе набори векторів, матриць і лінійних масивів, при роботі з якими приріст швидкодії від наборів інструкцій і включенням автовекторизації компілятором проявляє себе найбільш сильно.

Обчислювальний експеримент був здійснений на персональному комп'ютері (рис.3.1) з наступними характеристиками:

- ОС: Windows 7 x64 Ultimate
- процесор: i5-3210M 2.50 GHz (рис.3.2);
- memory RAM: 6 Гб;
- тип системи 64-bit.

В якості розробки використовувалась програма Microsoft Visual Studio 2019 C++ (рис.3.3) із компілятором Compiler версії (142), який підтримує всі нові інструкції SIMD.

View basic information about your computer

Windows edition

Windows 7 Ultimate

Copyright © 2009 Microsoft Corporation. All rights reserved.

Service Pack 1

System

Manufacturer:

Rating: 5,8 Windows Experience Index

Processor: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz 2.50 GHz

Installed memory (RAM): 6,00 GB (5,89 GB usable)

System type: 64-bit Operating System

Рисунок 3.1 – Основні відомості персонального комп'ютера

CPU | Caches | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name	Intel Core i5 3210M				
Code Name	Ivy Bridge	Max TDP	35.0 W		
Package	Socket 988B rPGA				
Technology	22 nm	Core VID	0.826 V		
Specification	Intel® Core™ i5-3210M CPU @ 2.50GHz				
Family	6	Model	A	Stepping	9
Ext. Family	6	Ext. Model	3A	Revision	E1/L1
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX				

intel inside CORE i5

Clocks (Core #0)

Core Speed	2194.76 MHz
Multiplier	x 22.0 (12 - 31)
Bus Speed	99.76 MHz
Rated FSB	

Cache

L1 Data	2 x 32 KBytes	8-way
L1 Inst.	2 x 32 KBytes	8-way
Level 2	2 x 256 KBytes	8-way
Level 3	3 MBytes	12-way

Selection Socket #1 Cores 2 Threads 4

CPU-Z Ver. 1.90.1.x32 Tools Validate Close

Рисунок 3.2 – Характеристики процесора Intel Core i5-3210M

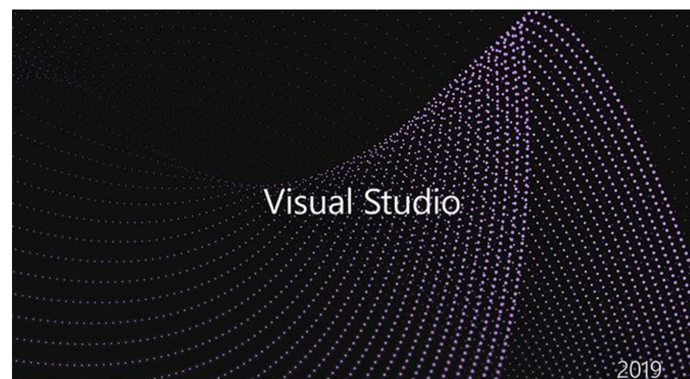

Рисунок 3.3 – Вікно завантаження програми Microsoft Visual Studio 2019

3.2 Результати множення матриць

Результати порівнянь алгоритмів множення матриць виконаних, без використання векторної обробки і алгоритму з використанням векторної обробки, приведені в таблиці 3.1, в ній можна спостерігати на різність кількості тактів процесору і часу виконання програм.

При великих обсягах даних компілятор може видавати помилки, про переповнення пам'яті. При великих значеннях кількості ітерацій параметру *nruns*, час виконання програм може затягнутися на довгий час, особливо це може позначитися для звичайного алгоритму без використання інструкцій AVX. Отже для оптимізованих алгоритмів множення матриць час виконання збільшиться, при цьому швидкодія буде краще, тому що алгоритми працюють з векторними інструкціями.

Таблиця 3.1 – Результати вимірів множень матриць з векторною і без векторної обробки.

Множення матриць різних розмірностей **n** x **m**					
Матриця	Алгоритми	Компілятор x86 x64		Кількість ітерацій nruns	
		Кількість тактів		Час виконання в секундах	
4x4	STANDART	727,29	738,91	1,82 s	1,79 s
	AVX128	619,88	710,80	1.14 s	1,27 s
	AVX256	598,20	542,77	1,12 s	0,96 s
8x8	STANDART	4594,14	4528,52	8,68 s	8,74 s
	AVX128	1154,78	1348,22	2,10 s	2,46 s
	AVX256	1126,60	1010,50	2,00 s	1,85 s
16x16	STANDART	35066,22	35001,08	62,02 s	61,86 s
	AVX128	2241,60	2639,65	4,02 s	4,71 s
	AVX256	1901,10	1767,65	3,43 s	3,16 s
32x32	STANDART	28996,49	300699,52	508,49 s	528,29 s
	AVX128	4391,62	5232,57	7,88 s	9,19 s
	AVX256	4240,78	3951,89	7,59 s	6,99 s

Ґрунтуючись на отриманих даних таблиці 3.1. були побудовані графіки залежності від наборів використаних команд (рис. 3.4, 3.5). На графіках можна спостерігати, що при великій кількості тактів процесору пропускна здатність зменшується, чим більш час виконання тим більш довше програма буде виконуватись.

Зменшення тактів і часу виконання досягається при використані інструкції AVX з включенням автовекторизації компілятору в програмі Microsoft Visual Studio у налаштуванні програми.

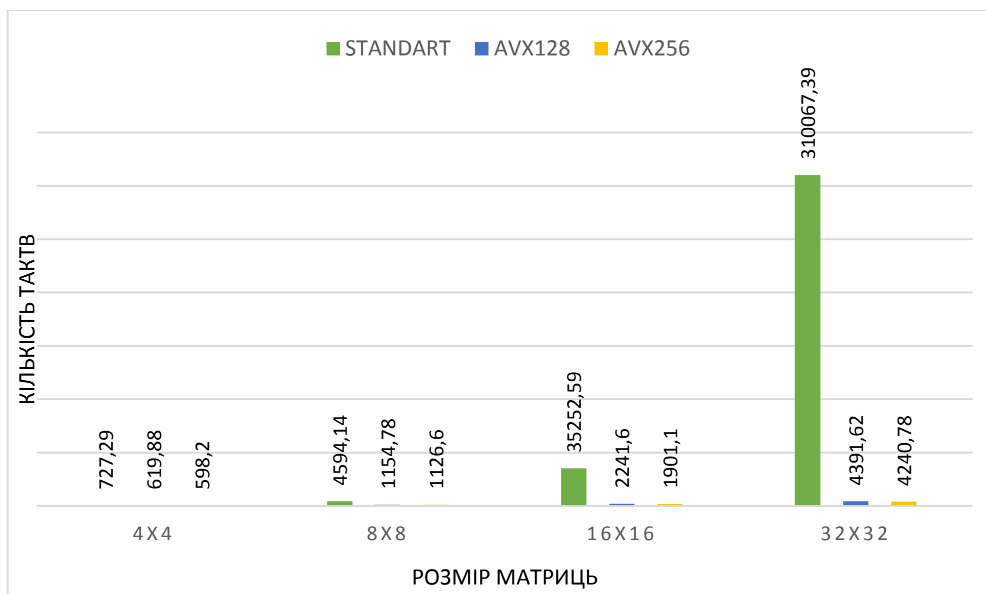

Рисунок 3.4 – Залежність тактів процесору виконаних від наборів використовуваних команд для різних розмірів матриць

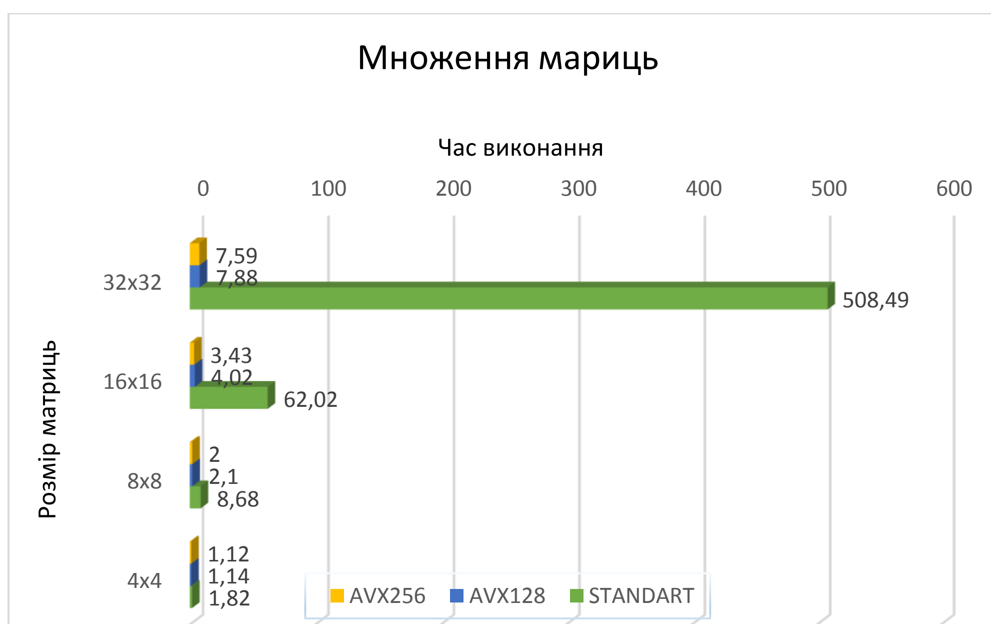

Рисунок 3.5 – Залежність часу виконання від наборів використаних команд для різних розмірів матриць

Запускаємо програму множення матриць на виконання як для 32 бітної так і для 64 бітної компіляції, очікуємо кінця виконання, після припинення програми на екрані бачимо результати трьох алгоритмів, кількості тактів і часу виконання (рис. 3.6, 3.7), ліворуч бачимо матрицю малої розмірності, праворуч навпаки матриця великої розмірності.

```
The time: 62.02 seconds
    standart: 35066.22 tacts
The time: 4.02 seconds
       AVX_4: 2241.60 tacts
The time: 3.43 seconds
       AVX_8: 1901.10 tacts
```

```
The time: 508.49 seconds
    standart: 289960.19 tacts
The time: 7.88 seconds
       AVX_4: 4391.62 tacts
The time: 7.59 seconds
       AVX_8: 4240.78 tacts
```

Рисунок 3.6 – Результат множення матриць на версії платформі x32-бітної

```
The time: 61.86 seconds
    standart: 35001.08 tacts
The time: 4.71 seconds
       AVX_4: 2639.65 tacts
The time: 3.16 seconds
       AVX_8: 1767.65 tacts
```

```
The time: 528.29 seconds
    standart: 300699.50 tacts
The time: 9.19 seconds
       AVX_4: 5232.57 tacts
The time: 6.99 seconds
       AVX_8: 3951.89 tacts
```

Рисунок 3.7 – Результат множення матриць на версії платформи x64-бітної

Після запусків програм було зроблено порівняння вимірів продуктивності. Як видно з результатів експерименту, використання розширених наборів інструкцій позитивним чином позначається на швидкодії множення матриць, найкращий ефект досягається при використанні наборів інструкцій AVX, при цьому значення щодо приросту виявляється досить близькими до значень приросту досягнутим раніше між 128-бітними и 256-бітними регістрами AVX. Відносний приріст від задіяння автовекторизації виявився в рази нижче отриманого в ідеальних умовах і варіювався від 20% при залученні інструкцій AVX 128-бітних до 25% в AVX 256-бітних відповідно. Загальний приріст склав в середньому близько 22%, що в будь-якому випадку є дуже хорошим результатом, враховуючи ту легкість, з якою він був досягнутий.

3.3 Результати транспонування матриць

При написані програмного алгоритму транспонуванні матриць обрано різну розмірність матриць де кількість рядків $\boldsymbol{m}$ менше кількості стовпців $\boldsymbol{n}$. Результати порівнянь алгоритмів транспонування матриць., виконаних, без використання векторної обробки і алгоритму з використанням векторної обробки, наведені в таблиці 3.2.

В таблиці можна спостерігати на параметри які задаються для матриць, K – кількість рядків із отриманої матриці, використовується тільки для алгоритму з інструкціями, M – розмір рядків матриці, N – розмір стовпців матриці, tacts – різність кількості тактів процесору і seconds – час виконання програм.

Щоб досягти максимальної ефективності і оптимізації транспонування матриць, використовуємо AVX інструкції (табл. 3.2). Можна спостерігати, що при збільшенні параметрів встановлених рядків і стовпців алгоритму без векторної обробки, кількість тактів збільшується час виконання майже не змінюється. Для алгоритму з інструкціями AVX додаємо ще один параметр K, який дозволяє завантажувати 4 рядки одночасно, що покращує продуктивність алгоритму в 4 рази. Якщо параметр K збільшити, програма виконається, але прискорення не буде. Цей параметр задає тільки розмір масивів, в коді він бере тільки 4 значення з цього масиву, а саме перші 4 осередки в пам'яті. При великих значеннях параметрів рядків і стовпців матриць компілятор може видавати помилки про переповнення пам'яті.

Таблиця 3.2 – Результати вимірів транспортувань матриць з векторною і без векторної обробки.

Транспортування матриць з різними параметрами					
Алгоритми	Параметри			Виконання програм	
	K – параметр	M – рядки	N – стовпці	tacts	seconds
STANDART	не використовується	32	64	93797	0,00
AVX	4	32	64	41230	0,00
STANDART	не використовується	128	256	923079	0,00
AVX	4	128	256	649963	0,00
STANDART	не використовується	512	1024	113426085	0,04
AVX	4	512	1024	48679154	0,02
STANDART	не використовується	4096	8192	2245409159	0,90
AVX	4	4096	8192	1244661280	0,50
STANDART	не використовується	8192	11000	6608633546	2,65
AVX	4	8192	11000	3864488297	1,55

Ґрунтуючись на отриманих даних таблиці 3.2. був побудований графік (рис. 3.8). На ньому можна спостерігати, що при збільшені параметрів рядків і стовпців, кількість тактів теж збільшується, як для стандартного так і для оптимізованого алгоритму, але при цьому алгоритм з інструкціями AVX виграє. Тому для виграшу додали ще один параметр K, в якому отримуємо 4 рядки з пам'яті. Це призвело до зменшення тактів процесора приблизно на 18% це в 4 рази менше, ніж алгоритм без інструкцій.

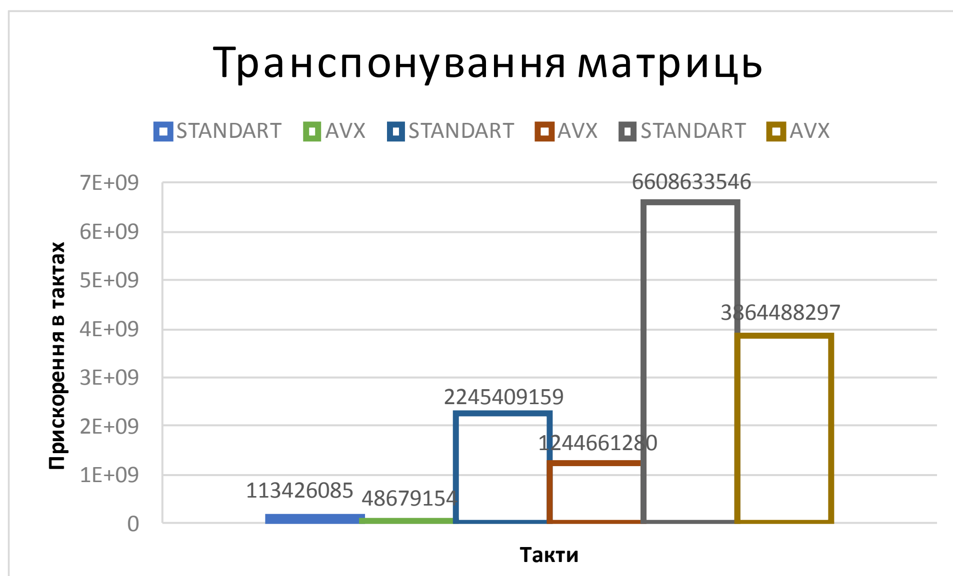

Рисунок 3.8 – Графік порівняння тактів обох алгоритмів

Запускаємо програму транспонування матриць на виконання і очікуємо кінця виконання. Після припинення програми на екрані бачимо вихідну матрицю, яка транспонована в іншу, кількість тактів і час виконання обох алгоритмів (рис. 3.9).

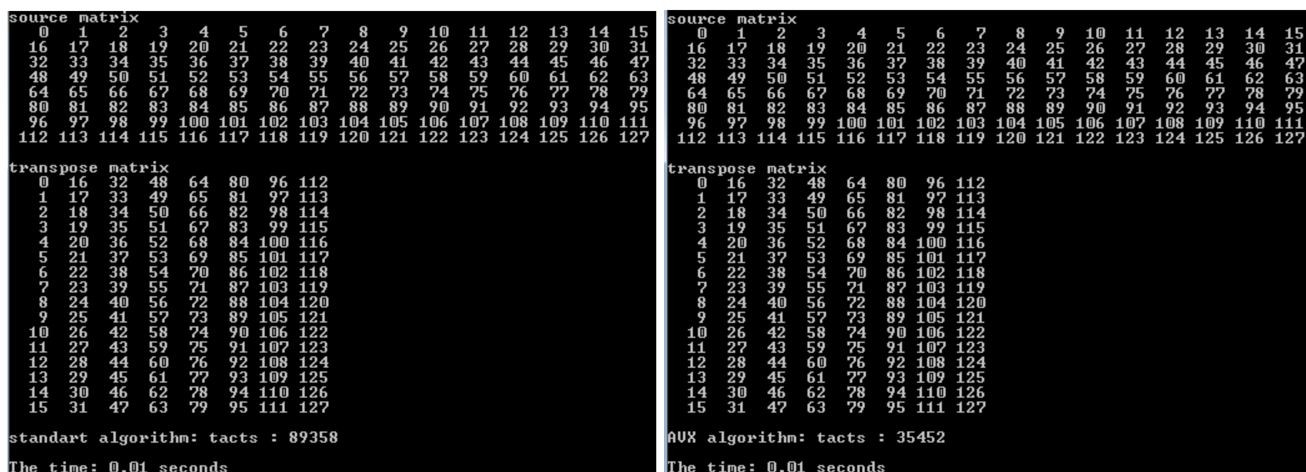

Рисунок 3.9 – Результат транспонування матриці з 8 рядків і 16 стовпців

Після запусків програм було зроблено порівняння. Як видно з результатів експерименту, якщо збільшити параметр рядків M і стовпців N при транспонуванні матриць стандартним алгоритмом, кількість тактів збільшиться в декілька мільйонів і навіть мільярдів раз, але при цьому час виконання майже не змінився. Це говорить про те, що транспонування матриць дуже легка операція для процесору. При транспонуванні матриць набір інструкцій AVX в 4 рази зменшив кількість тактів процесору, ніж в стандартному алгоритмі. Загальний приріст склав в середньому близько 40%, що є ефективним результатом при програмуванні алгоритму.

3.4 Результат звернення матриць

При написані програми звернень матриць була обрана матриця розміру ***n*** x ***n***. Матриця завжди повинна бути квадратною, якщо матриця не квадратна рішення не виконується. Результати порівнянь алгоритмів звернення матриць, виконаних без використання векторної обробки і алгоритму з використанням векторної обробки, наведені в таблиці 3.3.

В таблиці можна спостерігати на параметри, які задаються для матриць:

1. K – зміщення у рядка матриці на 4 double числа, використовується тільки для алгоритму з інструкціями AVX;
2. N – розмір матриці в нашому випадку ***n*** x ***n***;
3. такти процесору – кількість тактів процесору виконаної програми.

Для досягнення максимальної оптимізації звернення матриць використовуємо AVX інструкції і автовекторизацію компілятором програми Microsoft Visual Studio C++. В таблиці 3.3 можна спостерігати, що при збільшенні параметру N для обох алгоритмів кількість тактів збільшується. Но при цьому алгоритм з інструкціями AVX виграє по зменшенню тактів, тому для виграшу додаємо ще один параметр M, який дозволяє зробити зміщення у рядках матриці на 4 double числа в пам’яті, це покращує швидкодію алгоритму в 4 рази. Якщо параметр M збільшити наприклад до 8-ми, програма виконається, але прискорення не буде. Це обумовлено тим, що параметр M візьме 4 елементи масиву у пам’ять, а інші 4 елементи видалить, так як AVX роботає з числами double, які займають 4 комірки в пам’яті. При великих значень параметру N компілятор може видавати помилку про переповнення пам'яті і вийти із неї.

Таблиця 3.3 – Результати вимірів звернення матриць з векторною і без векторної обробки.

Обернення матриць			
Алгоритми	Параметри		Такти процесору
	M – зміщення double чисел в пам’яті	N – розмір матриць	
STANDART	не використовується	32x32	1538938
AVX	4	32x32	808152
STANDART	не використовується	128x128	63590262
AVX	4	128x128	47140752
STANDART	не використовується	512x512	2915557871
AVX	4	512x512	2781829980
STANDART	не використовується	2000x2000	124048763646
AVX	4	2000x2000	132905481881
STANDART	не використовується	4000x4000	1138619719773
AVX	4	4000x4000	1057021845289
STANDART	не використовується	8000x8000	10715209455232
AVX	4	8000x8000	8903995132158

Ґрунтуючись на отриманих даних таблиці 3.3. був побудований графік (рис. 3.10). На ньому можна спостерігати, що при збільшені параметру N кількість тактів теж збільшується як для стандартного, так і для оптимізованого алгоритму. Але при цьому алгоритм з інструкціями AVX виграє, тому для виграшу додали ще один параметр M, в якому отримуємо зміщення double чисел в пам’яті і включили автовекторизацію компілятором. Це призвело до зменшення тактів процесора приблизно в 40%, це в 2 рази менше, ніж алгоритм без інструкцій.

Рисунок 3.10 – Графік порівняння тактів обох алгоритмів обернення матриць

Запускаємо програму обернення матриць на виконання. Після припинення програми на екрані бачимо вихідну матрицю і матрицю обернену до неї, кількість тактів обох алгоритмів, виведення матриць на екран показано на рисунку 3.11. Щоб довести працездатність програми і правильність її рішення, можна скористатися онлайн калькулятором. Переходимо на сайт [41] вибираємо матрицю розміру 4x4, водимо матрицю в даному випадку матриця розміру 4x4 (рис. 3.11), звіряємо з відповіддю inverse matrix, дивимось правильність рішення цього алгоритму в онлайн ресурсі, якщо збігається, то програмний алгоритм реалізовано вірно.

```
source matrix
   0.00000    1.00000    2.00000    3.00000
   4.00000    0.00000    6.00000    7.00000
   8.00000    9.00000    0.00000   11.00000
  12.00000   13.00000   14.00000    0.00000

simple inverse matrix
  -0.48333    0.14167    0.04167    0.00833
   0.26667   -0.16667    0.03333    0.03333
   0.16667    0.03333   -0.06667    0.03333
   0.13333    0.03333    0.03333   -0.03333

tacts : 18638
```

```
source matrix
   0.00000    1.00000    2.00000    3.00000
   4.00000    0.00000    6.00000    7.00000
   8.00000    9.00000    0.00000   11.00000
  12.00000   13.00000   14.00000    0.00000

avx inverse matrix
  -0.48333    0.14167    0.04167    0.00833
   0.26667   -0.16667    0.03333    0.03333
   0.16667    0.03333   -0.06667    0.03333
   0.13333    0.03333    0.03333   -0.03333

tacts : 12740
```

Рисунок 3.11 – Результат обернення матриць розміру 4x4

Так само щоб перевірити рішення алгоритму матриці розміру 8x8, переходимо на інший сайт [42] вибираємо матрицю розміру 8x8, водимо матрицю як на рисунках 3.12-3.13 значення із source matrix, звіряємо результати з inverse matrix і дивимось чи співпали вони. Якщо співпали то рішення вважається вірним.

```
source matrix
   0.00000   1.00000   2.00000   3.00000   4.00000   5.00000   6.00000   7.00000
   8.00000   0.00000  10.00000  11.00000  12.00000  13.00000  14.00000  15.00000
  16.00000  17.00000   0.00000  19.00000  20.00000  21.00000  22.00000  23.00000
  24.00000  25.00000  26.00000   0.00000  28.00000  29.00000  30.00000  31.00000
  32.00000  33.00000  34.00000  35.00000   0.00000  37.00000  38.00000  39.00000
  40.00000  41.00000  42.00000  43.00000  44.00000   0.00000  46.00000  47.00000
  48.00000  49.00000  50.00000  51.00000  52.00000  53.00000   0.00000  55.00000
  56.00000  57.00000  58.00000  59.00000  60.00000  61.00000  62.00000   0.00000


simple inverse matrix
  -0.33472   0.08780   0.03224   0.01372   0.00446  -0.00109  -0.00479  -0.00744
   0.11905  -0.10714   0.00397   0.00397   0.00397   0.00397   0.00397   0.00397
   0.06349   0.00397  -0.05159   0.00397   0.00397   0.00397   0.00397   0.00397
   0.04497   0.00397   0.00397  -0.03307   0.00397   0.00397   0.00397   0.00397
   0.03571   0.00397   0.00397   0.00397  -0.02381   0.00397   0.00397   0.00397
   0.03016   0.00397   0.00397   0.00397   0.00397  -0.01825   0.00397   0.00397
   0.02646   0.00397   0.00397   0.00397   0.00397   0.00397  -0.01455   0.00397
   0.02381   0.00397   0.00397   0.00397   0.00397   0.00397   0.00397  -0.01190


tacts : 21143
```

Рисунок 3.12 – Результат обернення матриць розміру 8x8 без векторної обробки

```
source matrix
   0.00000   1.00000   2.00000   3.00000   4.00000   5.00000   6.00000   7.00000
   8.00000   0.00000  10.00000  11.00000  12.00000  13.00000  14.00000  15.00000
  16.00000  17.00000   0.00000  19.00000  20.00000  21.00000  22.00000  23.00000
  24.00000  25.00000  26.00000   0.00000  28.00000  29.00000  30.00000  31.00000
  32.00000  33.00000  34.00000  35.00000   0.00000  37.00000  38.00000  39.00000
  40.00000  41.00000  42.00000  43.00000  44.00000   0.00000  46.00000  47.00000
  48.00000  49.00000  50.00000  51.00000  52.00000  53.00000   0.00000  55.00000
  56.00000  57.00000  58.00000  59.00000  60.00000  61.00000  62.00000   0.00000


avx inverse matrix
  -0.33472   0.08780   0.03224   0.01372   0.00446  -0.00109  -0.00479  -0.00744
   0.11905  -0.10714   0.00397   0.00397   0.00397   0.00397   0.00397   0.00397
   0.06349   0.00397  -0.05159   0.00397   0.00397   0.00397   0.00397   0.00397
   0.04497   0.00397   0.00397  -0.03307   0.00397   0.00397   0.00397   0.00397
   0.03571   0.00397   0.00397   0.00397  -0.02381   0.00397   0.00397   0.00397
   0.03016   0.00397   0.00397   0.00397   0.00397  -0.01825   0.00397   0.00397
   0.02646   0.00397   0.00397   0.00397   0.00397   0.00397  -0.01455   0.00397
   0.02381   0.00397   0.00397   0.00397   0.00397   0.00397   0.00397  -0.01190


tacts : 18936
```

Рисунок 3.13 – Результат обернення матриць розміру 8x8 з векторною обробки

Є ще один спосіб перевірити роботу обернення матриць за допомогою програми Excel. Детально як працює програма описувати не буду, лише наведу приклад роботи з матрицею розміру 8x8. Для початку водимо елементи матриці в комірки програми (рис 3.14).

	A	B	C	D	E	F	G	H
1	0	1	2	3	4	5	6	7
2	8	0	10	11	12	13	14	15
3	16	17	0	19	20	21	22	23
4	24	25	26	0	28	29	30	31
5	32	33	34	35	0	37	38	39
6	40	41	42	43	44	0	46	47
7	48	49	50	51	52	53	0	55
8	56	57	58	59	60	61	62	0

Рисунок 3.14 – Введення матриці розміром 8х8 обернення матриць

Далі в порожньому місці комірок виділяємо діапазон A10 - H17, тиснемо клавішу F2, а потім одночасно клавіші Ctrl + Shift + Enter і бачимо результат оберненої матриці (рис 3.15), котрий можна порівняти з отриманим програмним шляхом.

10	-0,33472	0,087798	0,032242	0,013724	0,004464	-0,00109	-0,00479	-0,00744
11	0,119048	-0,10714	0,003968	0,003968	0,003968	0,003968	0,003968	0,003968
12	0,063492	0,003968	-0,05159	0,003968	0,003968	0,003968	0,003968	0,003968
13	0,044974	0,003968	0,003968	-0,03307	0,003968	0,003968	0,003968	0,003968
14	0,035714	0,003968	0,003968	0,003968	-0,02381	0,003968	0,003968	0,003968
15	0,030159	0,003968	0,003968	0,003968	0,003968	-0,01825	0,003968	0,003968
16	0,026455	0,003968	0,003968	0,003968	0,003968	0,003968	-0,01455	0,003968
17	0,02381	0,003968	0,003968	0,003968	0,003968	0,003968	0,003968	-0,0119

Рисунок 3.15 – Результат перевірки обернення матриць розміру 8х8

3.5 Висновки до третього розділу

При проведені дослідження ефективності використання векторної обробки ядер сучасних процесорів були зроблені наступні висновки.

Автоматичне використання розширених наборів інструкцій AVX і автовекторизації компілятором позитивно позначається на швидкодії програм. Найбільш відчутний приріст швидкості досягається при активації обох цих способів.

У реальних задачах множень, транспонувань, звернень матриць, ефективність автовекторизації в рази менше їх теоретичного максимуму, який спостерігався при програмній реалізації.

Використання сучасних наборів інструкцій AVX - 256-бітними регістрами для множення, транспонування і звернення матриць є більш кращими, ніж використовувати стандартні алгоритми, завдяки їх більш високій швидкості і меншому розміру генерованого машинного коду та декількох виконаних операцій за один такт.

Розглянуті алгоритми у розділі 2 не вимагають вносити зміни у вихідний програмний код, тому обов'язково повинні використовуватися в якості першого кроку на шляху оптимізації будь-яких програм.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Загальні питання з охорони праці

Умови праці на робочому місці, безпека технологічних процесів, машин, механізмів, устаткування та інших засобів виробництва, стан засобів колективного та індивідуального захисту, що використовуються працівником, а також санітарно-побутові умови повинні відповідати вимогам нормативних актів про охорону праці. В законі України «Про охорону праці» [11] визначається, що охорона праці - це система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних і лікувально-профілактичних заходів та засобів, спрямованих на збереження життя, здоров'я і працездатності людини у процесі трудової діяльності.

При роботі з обчислювальною технікою змінюються фізичні і хімічні фактори навколишнього середовища: виникає статична електрика, електромагнітне випромінювання, змінюється температура і вологість, рівень вміст кисню і озону в повітрі. Повітря забруднюється шкідливими хімічними речовинами антропогенного походження за рахунок деструкції полімерних матеріалів, які використовуються для обробки приміщень та обладнання. Неправильна організація робочого місця сприяє загальному і локальній напрузі м'язів шиї, тулуба, верхніх кінцівок, викривлення хребта і розвитку остеохондрозу. На всіх підприємствах, в установах, організаціях повинні створюватися безпечні і нешкідливі умови праці.

4.1.1 Правові та організаційні основи охорони праці

Державна політика в галузі охорони праці визначається відповідно до Конституції України Верховною Радою України і спрямована на створення належних, безпечних і здорових умов праці, запобігання нещасним випадкам та професійним захворюванням. Відповідно до статті 3 Закону України «Про охорону праці» [11] (далі – Закону) законодавство про охорону праці складається з Закону, Кодексу законів про працю України, Закону України "Про загальнообов'язкове державне соціальне страхування від нещасного випадку на виробництві та професійного захворювання, які спричинили втрату працездатності" [1] а прийнятих відповідно до них нормативно-правових актів, норм міжнародного договору (ратифіковані Конвенції і Рекомендації МОТ, директиви Європейської Ради).

Користувачі персональних комп’ютерів, для яких ця робота є головною, підлягають

медичним оглядам: попереднім — під час влаштування на роботу і періодичним — протягом професійної діяльності раз на два роки. Жінок з часу встановлення вагітності та в період годування дитини грудьми до роботи з ПК не допускають.

Наявні трудові відносини між працівниками і роботодавцями в Україні за темою дипломного проекту регулюються Кодексом законів про працю (КЗпП) України, відповідно до якого права працюючої людини на охорону праці охороняються всебічно та норми охорони праці неухильно інтегровані до правил внутрішнього розпорядку організації/підприємства.

4.1.2 Організаційно-технічні заходи з безпеки праці

В організації/підприємстві проводиться навчання і перевірка знань з питань охорони праці відповідно до вимог Типового положення про порядок проведення навчання і перевірки знань з питань охорони праці, затвердженого наказом Держнаглядохоронпраці України від 26.01.2005 N 15, зареєстрованого в Міністерстві юстиції України 15.02.2005 за N 231/10511 [2].

Обов’язковими вимогами враховане наступне:

− не слід допускати до роботи осіб, що в установленому порядку не пройшли навчання, інструктаж та перевірку знань з охорони праці, пожежної безпеки та цих Правил.

− на підприємстві/організації, де експлуатуються ЕОМ з відео дисплейними терміналами (ВДТ) і периферійними пристроями (ПП), розробляється інструкція з охорони праці.

− ознайомлення з правилами безпеки праці, одержання відповідних інструктажів засвідчується у журналі інструктажів.

− перед допуском до самостійної роботи кожен працівник має право на навчання з питань охорони праці і роботодавець зобов’язаний, і проводить таке навчання у вигляді двох інструктажів з питань охорони праці.

4.2 Аналіз стану умов праці

Робота над створенням дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів проходитиме в приміщенні відповідної установи (компанії, підприємстві тощо). Для даної роботи достатньо однієї людини, для якої надано робоче місце зі стаціонарним комп’ютером.

Обчислювальна техніка при функціонуванні має наступні експлуатаційні характеристики:

− робоче живлення 220 В;

− частота живильної мережі 50 Гц;

− споживана потужність в межах 300 Вт.

При роботі на персональних ПЕОМ користувач наражається на небезпеку ураження електричним струмом. Приміщення для обчислювальної техніки за ступенем небезпеки ураження людини електричним струмом відноситься до приміщень без підвищеної небезпеки. Тяжкість роботи персоналу, що обслуговує і працює на ПЕОМ, відноситься до категорії 1а - легкі фізичні навантаження. При обслуговуванні обчислювальної техніки мають місце фізичні і психофізіологічні небезпечні та шкідливі виробничі фактори:

- підвищене значення напруги в електричному ланцюзі, замикання якого може відбутися через тіло людини;
- підвищена або знижена температура повітря робочої зони;
- підвищена або знижена рухливість повітря;
- підвищена або знижена вологість;
- підвищений рівень електромагнітних полів у робочій зоні;
- відсутність або нестача природного світла;
- підвищена пульсація світлового потоку;
- недостатнє освітлення робочого місця;
- підвищена статична електроенергія.

4.2.1 Вимоги до приміщень

Для захисту людей від ураження електричним струмом при дотику до металевих неструмоведучих частин, які можуть опинитися під напругою в результаті пошкодження ізоляції, передбачаються наступні заходи:

- захисне заземлення або занулення металевих частин електроустановок, які доступні для дотику людини й не мають інших видів захисту, що забезпечують електробезпеку;
- захисне відключення;
- електричний поділ мереж;
- використання малої напруги;
- ізоляція струмоведучих частин;
- огорожу електроустановок;
- шина заземлення виконується провідником з опором не більше 4-х Ом

Геометричні розміри приміщення зазначені в (табл. 4.1).

Таблиця 4.1 – Розміри приміщення

Найменування	Значення
Довжина, м	6

Ширина, м	5
Висота, м	2,5
Площа, м^2	30
Об’єм, м^3	75

Розмір площі для одного робочого місця оператора персонального комп’ютера має бути не менше 6 кв. м, а об’єм — не менше 20 куб. м. Отже, дане приміщення цілком відповідає зазначеним нормам.

Для забезпечення потрібного рівного освітленості кімната має вікно та систему загального рівномірного освітлення, що встановлена на стелі. Для дотримання вимог пожежної безпеки встановлено порошковий вогнегасник та систему автоматичної пожежної сигналізації.

4.2.2 Вимоги до організації місця праці

При порівнянні відповідності характеристик робочого місця нормативним основні вимоги до організації робочого місця і відповідними фактичними значеннями для робочого місця констатуємо повну відповідність (табл. 4.2).

Таблиця 4.2 - Характеристики робочого місця

Найменування параметра	Фактичне значення	Нормативне значення
Висота робочої поверхні, мм	750	680-800
Висота простору для ніг, мм	730	не менше 600
Ширина простору для ніг, мм	660	не менше 500
Глибина простору для ніг, мм	700	не менше 650
Висота поверхні сидіння, мм	470	400-500
Ширина сидіння, мм	400	не менше 400
Глибина сидіння, мм	400	не менше 400
Висота поверхні спинки, мм	600	не менше 300
Ширина опорної поверхні спинки, мм	500	не менше 380
Радіус кривини спинки в горизонтальній площині, мм	400	400
Відстань від очей до екрану дисплея, мм	800	700-800

Приміщення кабінету має об’єм 70 м^3, площу – 30 м^2.

Температура в приміщенні протягом року коливається у межах 18–24°С, відносна вологість — близько 50%. Система вентилювання приміщення — природна неорганізована, а опалення — автономне.

Розміщення вікон забезпечує природне освітлення з коефіцієнтом природного освітлення не менше 1,5%, а загальне штучне освітлення, яке здійснюється за допомогою трьох люмінесцентних ламп, забезпечує рівень освітленості не менше 200 Лк.

За ступенем пожежної безпеки приміщення належить до категорії В.

4.2.3 Навантаження та напруженість процесу праці

За фізичним навантаженням робота відноситься до категорії легкі роботи (Іа), її виконують сидячи з періодичним ходінням. Щодо характеру організування виконання дипломної роботи, то він підпадає під нав’язаний режим, оскільки певні розділи роботи необхідно виконати у встановлені конкретні терміни. За ступенем нервово-психічної напруги виконання роботи можна віднести до ІІ – ІІІ ступеня і кваліфікувати як помірно напружений – напружений за умови успішного виконання поставлених завдань.

Роботу за дипломним проектом визнано, таку, що займає 50% часу робочого дня та за восьмигодинної робочої зміни рекомендовано встановити додаткові регламентовані перерви тривалістю 15 хв. через кожну годину роботи.

4.3 Виробнича санітарія

На підставі аналізу небезпечних та шкідливих факторів при виробництві (експлуатації), пожежної безпеки можуть бути надалі вирішені питання необхідності забезпечення працюючих достатньою кількістю освітлення, вентиляції повітря, організації заземлення, тощо [3].

4.3.1 Аналіз небезпечних та шкідливих факторів при виробництві (експлуатації) виробу

Роботу, пов'язану з ЕОП з ВДТ, у тому числі на тих, які мають робочі місця, обладнані ЕОМ з ВДТ і ПП, виконують із забезпеченням виконання НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями»[7], які встановлюють вимоги безпеки до обладнання робочих місць, до роботи із застосуванням ЕОМ з ВДТ і ПП. Переважно роботи за проектами виконують у кабінетах чи інших приміщеннях, де використовують різноманітне електрообладнання, зокрема персональні комп’ютери (ПК) та периферійні пристрої.

Основними робочими характеристиками персонального комп'ютера є наступні:

– робоча напруга U = +220В ± 5%;
– робочий струм I = 2А;
– споживана потужність P = 350 Вт.

Робоче місце має відповідати вимогам Державних санітарних правил і норм роботи з

візуальними дисплейними терміналами електронно-обчислювальних машин, затверджених постановою Головного державного санітарного лікаря України від 10.12.98 N 7 [4].

Аналіз небезпечних та шкідливих виробничих факторів виконується у табличній формі (табл. 4.3).

Таблиця 4.3 – Аналіз небезпечних і шкідливих виробничих факторів

Небезпечні і шкідливі виробничі фактори	Джерела факторів (види робіт)	Кількісна оцінка	Нормативні документи
1	2	3	А
Фізичні			
- підвищений рівень напруги електричної мережі, замикання якої може відбутися через тіло людини	-//-	4	ГОСТ 13109-97 [5]
- недостатність природного світла	порушення умов праці (вимог до приміщень)	2	ДБН В.2.5-28:2018 [6]
- недостатнє освітлення робочої зони	порушення гігієнічних параметрів виробничого середовища	3	ДБН В.2.5-28:2018 [6]
психофізіологічні:			
- нервово-психічна перевантаження (розумове, перенапруження аналізаторів-зорових)	- пошук інформації для постановки теми; - пошук та аналіз аналогів і літератури; - пошук наявних технологій, моделювання та аналіз алгоритмів; - виконання роботи за темою диплома, тестування; - оформлення роботи	4	НПАОП 0.00-7.15-18 [7] ДСанПіН 3.3.2.007-98 [4]
- фізичні (статичне – сидіння)	порушення умов праці (організації місця праці- сидіння користувача,) та організації робочого часу - безперервна робота)	2	НПАОП 0.00-7.15-18 [7] ДСанПіН 3.3.2.007-98 [4]

4.3.2 Пожежна безпека

Приміщення оснащено системою автоматичної пожежної сигналізації, має 1 вогнегасник ВП-5 із зарядом вогнегасної речовини 8-12 кг, відповідно до вимог чинного законодавства України. Проходи до засобів пожежогасіння вільні, не захаращуються та у разі потреби забезпечувати евакуацію всіх людей, які перебувають у приміщенні через один евакуаційний вихід з дверима на шляху евакуації, що відчинятися в напрямку виходу з будівлі від робочого місця. В приміщенні наявна затверджена «План-схема евакуації з кабінету (приміщення)».

Пожежна безпека при застосуванні ЕОМ забезпечується:

1) системою запобігання пожежі,
2) системою протипожежного захисту,
3) організаційно-технічними заходами.

Згідно ДСТУ Б В.1.1-36:2016 [8] таке приміщення, площею 30 м^2, відноситься до категорії "В" (пожежонебезпечної) та для протипожежного захисту в ньому проектом передбачено устаткування автоматичною пожежною сигналізацією із застосуванням датчиків-сповіщувачів РІД-1 (сповіщувач димовий ізоляційний) в кількості 1 шт., і застосуванням первинних засобів пожежогасіння. Відповідно до норм первинних засобів пожежогасінні пропонується використовувати:

- ручний вуглекислий вогнегасник ОУ-5 в кількості 1 шт.;
- повсть 11 м^2, кошму 2×1,5 м^2 або азбестове полотно 2×2 м^2 в кількості 1 шт.

Горючими матеріалами в приміщенні, де розташовані ЕОМ, є:

1) поліамід – матеріал корпусу мікросхем, горюча речовина, температура самозаймання 420º С,
2) полівінілхлорид – ізоляційний матеріал, горюча речовина, температура запалювання 335º С, температура самозаймання 530º С,
3) склотекстоліт ДЦ – матеріал друкарських плат, важкогорючий матеріал, показник горючості 1.7А, не схильний до температурного самозаймання,
4) А) пластикат кабельний №.489 – матеріал ізоляції кабелів, горючий матеріал, показник горючості більше 2.1,
5) деревина – будівельний і обробний матеріал, з якого виготовлені меблі, горючий матеріал, показник горючості більше 2.1, температура запалювання 255º С, температура самозаймання 399º С.

Простори усередині приміщень в межах, яких можуть утворюватися або знаходиться пожежонебезпечні речовини і матеріали відповідно до [8] відносяться до пожежонебезпечної зони класу П-IIа. Це обумовлено тим, що в приміщенні знаходяться тверді горючі та важкозаймисті речовини та матеріали. Приміщенню, у якому розташоване робоче місце, присвоюється II ступень вогнестійкості.

4.4 Освітлення

Освітленість приміщення має велике значення при роботі на ПЕОМ. Вона багато в чому визначається колірною і мережевий обстановкою. Для зменшеного поглинання світла стеля і стіни вище панелей (1,5-1,7м.). Якщо вони не облицьовані звукопоглинальним матеріалом,

фарбуються білою водоемульсійною фарбою (коефіцієнт відбиття повинен бути не менше 0,7). Для забарвлення стіни панелей рекомендується віддавати перевагу світлим фарбам.

Природне освітлення, коли робочі місця з ПЕОМ розташовуються в один ряд по довжині приміщення на відстані 0,8 - 1,0 м від стіни з віконними прорізами, і екрани знаходяться перпендикулярно цієї стіни. Основний потік природного світла при цій повинен бути зліва. Не допускається спрямування основного світлового потоку природного світла праворуч, ззаду і спереду працює на ПЕОМ. Оптимальна відстань очей до екрана відео монітора повинна становити 60-70 см, допустиме не менше 50 см. Розглядати інформацію ближче 50 см не рекомендується.

У приміщенні, де розташовані ЕОМ передбачається природне бічне освітлення, рівень якого відповідає ДБН В.2.5-28:2018 [6]. Джерелом природного освітлення є сонячне світло. Регулярно повинен проводитися контроль освітленості, який підтверджує, що рівень освітленості задовольняє ДБН і для даного приміщення в світлий час доби достатньо природного освітлення.

Розрахунок освітлення.

$$S_b = \left(\frac{1}{5} \div \frac{1}{10}\right) \cdot S_n, \tag{4.1}$$

де S_b – площа віконних прорізів, м^2;

S_n – площа підлоги, м^2.

$$S_n = a \cdot b = 5 \cdot 6 = 30 \text{ м}^2, \tag{4.2}$$

$$S = 1/6 \cdot 30 = 5 \text{ м}^2.$$

Приймаємо 1 вікно площею S=5 м^2.

Розрахунок штучного освітлення проводиться за коефіцієнтами використання світлового потоку, яким визначається потік, необхідний для створення заданої освітленості при загальному рівномірному освітленні.

Розрахунок кількості світильників здійснюється за формулою:

$$N = E \cdot S \cdot Z \cdot K / (F \cdot U \cdot M) \tag{4.3}$$

де N - число світильників;

E - нормоване освітлення;

S - площа підлоги, м^2, S=30 м^2;

Z - поправний коефіцієнт світильника (Z = 1,15 для ламп розжарювання та ДРЛ; Z = 1,1

для люмінесцентних ламп) приймаємо рівним 1,1;

K - коефіцієнт запасу, що враховує зниження освітленості в процесі експлуатації – 1,5;

U - коефіцієнт використання, що залежить від типу світильника, показника індексу приміщення і т. п. - 0,575;

M - число люмінесцентних ламп у світильнику - 3;

F - світловий потік – 1750 лм (для ЛБ - 30).

Згідно вимог ДБН В.2.5-28-2018, [6] освітлення робочого місця оператора обчислювальної техніки повинно бути не менше 200 лк.

$$N = 200 \cdot 30 \cdot 1.1 \cdot 1.5 / (1750 \cdot 0.575 \cdot 3) = 3{,}27 \approx 3 \quad (4.4)$$

Обираємо кількість світильників, що дорівнює 3.

4.5 Вентилювання

У приміщенні, де знаходяться ПК, повітрообмін реалізується за допомогою природної організованої вентиляції (вентиляційні шахти), тобто при V приміщення > 40 м3 на одного працюючого допускається природна вентиляція. Цей метод забезпечує приток потрібної кількості свіжого повітря, що визначається в СНіП.

Також має здійснюватися провітрювання приміщення, в залежності від погодних умов, тривалість повинна бути не менше 10 хв. Найкращий обмін повітря здійснюється при наскрізному провітрюванні.

4.6 Заходи з організації виробничого середовища та попередження виникнення надзвичайних ситуацій

Відповідно до санітарно-гігієнічних нормативів та правил експлуатації обладнання наводимо приклади деяких заходів безпеки.

1) Заходи безпеки під час експлуатації персонального комп'ютера та периферійних пристроїв передбачають:

− правильне організування місця праці та дотримання оптимальних режимів праці та відпочинку під час роботи з ПК;

− експлуатацію сертифікованого обладнання;

− дотримання заходів електробезпеки;

в) якщо об'єм приміщення становить понад 40 м^3, допускається природна вентиляція, у

випадку, коли немає виділення шкідливих речовин.

– зниження рівня шуму та вібрації:

а) у джерелі виникнення, шляхом застосування раціональних конструкцій, нових матеріалів і технологічних процесів;

б) звукоізолювання устаткування за допомогою глушників, резонаторів, кожухів, захисних конструкцій, оздоблення стін, стелі, підлоги тощо.

Розрахунок захисного заземлення (забезпечення електробезпеки будівлі).

Загальний опір захисного заземлення визначається за формулою:

$$R_{ззп} = \frac{R_з \cdot R_п}{R_п \cdot n \cdot \eta_з + R_з \cdot \eta_п} \tag{4.5}$$

де Rз - опір заземлення, якими когут бать труби, опори, кути і т.п., Ом;

Rш - опір опори, яке з'єднує заземлювачі, Ом;

п - кількість заземлювачів;

ηз - коефіцієнт екранування заземлювача; приймається в межах 0,2 ÷ 0,9; ηз = 0,7

ηш - коефіцієнт екранування сполучної стійки; приймається в межах 0,1 ÷ 0,7; ηш = 0,5;

Опір заземлення визначається за формулою:

$$R_з = \frac{\rho}{2\pi \cdot l} \cdot \left(\ln \frac{2 \cdot l}{d} + \frac{1}{2} \ln \frac{4 \cdot t + l}{4 \cdot t - l} \right) \tag{4.6}$$

де ρ - питомий опір ґрунту, залежить від типу ґрунту, Ом·м;

для піску - 400 ÷ 700 Ом·м; приймаємо ρ = 400 Ом·м;

l - довжина заземлювача, м; для труб - 2-3 м; l = 3 м;

d - діаметр заземлювача, м; для труб - 0,03-0,05 м; d = 0,05 м;

t - відстань від середини забитого в ґрунт заземлювача до рівня землі, м; t = 2 м.

$$R_з = \frac{400}{2 \cdot 3{,}14 \cdot 3} \left(\ln \frac{2 \cdot 3}{0{,}05} + \frac{1}{2} \ln \frac{4 \cdot 2 + 3}{4 \cdot 2 - 3} \right) = 110 \text{, Ом} \tag{4.7}$$

Опір смуги, що з'єднує заземлювачі, визначається за формулою:

$$R_{ш} = \frac{\rho}{2\pi \cdot L} \cdot \ln \frac{2 \cdot L^2}{b \cdot t^1}, \quad (4.8)$$

де L - довжина смуги, що з'єднує заземлювачі (м) і приблизно дорівнює периметру будівлі: $П_{буд.} = 42 \cdot 2 + 38 \cdot 2 = 160$ м; L = 160 м;

b - ширина смуги, м; b = 0,03 м;

t_1 - глибина заземлення від рівня землі, м; $t_1 = 0,5$ м.

$$R_{п} = \frac{400}{2 \cdot 3,14 \cdot 160} \cdot \ln \frac{2 \cdot 160^2}{0,03 \cdot 0,5} = 5,99, \text{ Ом} \quad (4.9)$$

Кількість заземлювачів захисного заземлення визначається за формулою:

$$n = \frac{2 \cdot R_{з}}{4 \cdot \eta_{з}}, \quad (4.10)$$

де 4 - допустимий загальний опір, Ом;

2 - коефіцієнт сезонності.

Визначаємо загальний опір захисного заземлення:

$$R_{ззп} = \frac{110 \cdot 5,99}{5,99 \cdot 79 \cdot 0,7 + 110 \cdot 0,5} = 1,7 \quad (4.11)$$

Висновок: дане захисне заземлення буде забезпечувати електробезпеку будівлі, так як виконується умова: $R_{ззп} < 4$ Ом.

4.7 Екологія

Діяльність за темою магістерської роботи, а саме: дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів в процесі її виконання впливає на навколишнє природнє середовище і регламентується нормами діючого законодавства: Законом України «Про охорону навколишнього природного середовища»[9], Законом України

«Про забезпечення санітарного та епідемічного благополуччя населення»[12], Законом України «Про відходи»[10].

В процесі діяльності виконанням дипломного проектування виникають процеси поводження з відходами ІТ галузі. Нижче надано перелік відходів, що утворюються в процесі роботи:

- Відпрацьовані люмінесцентні лампи - І клас небезпеки.
- Змінні носії інформації - IV клас небезпеки.
- Відпрацьовані вогнегасники - IV клас небезпеки.
- Макулатура - IV клас небезпеки.

4.8 Висновки до розділу 4

В результаті проведеної роботи було зроблено аналіз умов праці, шкідливих та небезпечних чинників, з якими стикається робітник. Було визначено параметри і певні характеристики приміщення для роботи над запропонованим проектом; описано, які заходи потрібно зробити для того, щоб дане приміщення відповідало необхідним нормам і було комфортним і безпечним для робітника. Приведені рекомендації щодо організації робочого місця, а також важливу інформацію щодо пожежної та електробезпеки. Була наведена схема, розміри приміщення та наведено значення температури, вологості й рухливості повітря, необхідна кількість і потужність ламп та інші параметри, значення яких впливає на умови праці робітника, а також – наведені інструкції з охорони праці, техніки безпеки при роботі на комп’ютері.

А також визначені основні екологічні аспекти впливу на навколишнє природнє середовище та зазначені заходи щодо поводження з ними.

4.9 Перелік корисних посилань до розділу 4

1. Закон України "Про загальнообов'язкове державне соціальне страхування від нещасного випадку на виробництві та професійного захворювання, які спричинили втрату працездатності". Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/2180-14
2. Про затвердження Типового положення про порядок проведення навчання і перевірки знань з питань охорони праці (НПАОП 0.00-4.12-05). Наказ від 26.01.2005 №15. Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/z0231-05
3. Санітарні норми мікроклімату виробничих приміщень ДСН 3.3.6.042-99. Постанова N 42 від 01.12.99. Режим доступу: www. URL: https://zakon.rada.gov.ua/rada/show/va042282-99
4. ДСанПіН 3.3.2.007-98 Гігієнічні вимоги до організації роботи з візуальними дисплейними терміналами електронно-обчислювальних машин. Режим доступу: www. URL: https://dnaop.com/html/31667/doc-ДСанПІН_3.3.2.007-98
5. ГОСТ 13109-97 Норми якості електричної енергії в системах електропостачання загального призначення. Наказ від 21.11.1997 № 12-97. Режим доступу: www. URL: https://dnaop.com/html/42313/doc-ГОСТ_13109-97
6. ДБН В.2.5-28:2018. Природне і штучне освітлення. Режим доступу: www. Наказ від 03.10.2018 № 264. Режим доступу: www. URL: https://dbn.co.ua/load/normativy/dbn/dbn_v_2_5_28/1-1-0-1188
7. НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями». Зареєстровано в Міністерстві юстиції України 25 квітня 2018 р. за № 508/31960. Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/z0508-18
8. ДСТУ Б В.1.1-36:2016 «Визначення категорій приміщень, будинків та зовнішніх установок за вибухопожежною та пожежною небезпекою». Наказ від 15.06.2016 №158. Режим доступу: www. URL: https://zakon.rada.gov.ua/rada/show/v0158858-16
9. Закон України «Про охорону навколишнього природного середовища» . Вводиться в дію Постановою ВР № 1268-XII від 26.06.91, ВВР, 1991, № 41, ст.547. Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/1264-12
10. Закон України «Про відходи». Відомості Верховної Ради України (ВВР), 1998, № 36-37, ст.242. Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/187/98-вр
11. Закон України «Про охорону праці» Відомості Верховної Ради України (ВВР), 1992, № 49, ст.668. Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/2694-12
12. Закон України «Про забезпечення санітарного та епідемічного благополуччя населення». Відомості Верховної Ради України (ВВР), 1994, № 27, ст.218. Режим доступу: www. URL: https://zakon.rada.gov.ua/laws/show/4004-12

ВИСНОВКИ

Intel AVX — це набір команд, призначений для додатків, що інтенсивно використовують операції з плаваючою точкою. Intel AVX підвищує продуктивність завдяки більш широким векторам, новому розширеному синтаксису і багатій функціональності. Це забезпечує краще управління даними і додатками загального призначення для обробки зображень, аудіо/відео, наукових імітацій, фінансового аналізу, тривимірного моделювання та аналізу.

Векторизація — це модифікація коду, яка замінює скалярний код на векторний. Тобто скалярні дані упаковуються в вектора і скалярні операції замінюються на операції з векторами.

Важливо відзначити, що процесори архітектури Intel, підтримують векторні інструкції. Тобто існує можливість створення векторів різних типів і застосування даних інструкцій, працюючи з векторами і отримуючи на виході вектор результатів. Проблема полягає в тому, що мови програмування спочатку скалярні і потрібно робити певні зусилля, щоб використовувати цю можливість обчислювальних ядер.

Наявність декількох обчислювальних ядер на сучасному процесорі дає можливість досягнення високої продуктивності програми розподілом обчислень між цими ядрами. Але і це всього лише «можливість». Перетворення однопотокового коду в багатопотоковий вимагає серйозних зусиль. Деякі оптимізуючи компілятори мають автопаралелізатор, який зводить процес створення багатопотокового додатка до процесу додавання однієї опції при компіляції. Але в більшості випадків потрібні зусилля для того, щоб розпаралелити спочатку одно потоковий алгоритм.

Можна виділити три основні характеристики, які визначають продуктивність і дві великі можливості, які можуть бути реалізовані:

1. якість роботи підсистеми пам'яті;
2. кількість умовних переходів;
3. рівень інструкційного паралелізму;
4. використання векторизації;
5. використання багатопотокових обчислень.

Досліджуючи ефективність векторної обробки ядер сучасних процесорів, було зроблено наступний висновок.

Автоматичне використання розширених наборів інструкцій AVX і автовекторизації компілятором позитивно позначається на швидкодії програм. Найбільш відчутний приріст швидкості досягається при активації обох способів. У реальних задачах ефективність автовекторизації в рази менше їх теоретичного максимуму, який спостерігався при програмній реалізації.

Використання сучасних наборів інструкцій AVX є більш кращими завдяки їх більш високій швидкості і меншому розміру генерованого машинного коду та декількох виконаних операцій за один такт.

Розглянуті алгоритми не вимагають вносити зміни у вихідний програмний код, тому обов'язково повинні використовуватися в якості першого кроку на шляху оптимізації будь-яких програм.

Для досягнення поставленої мети були вирішені такі завдання:

1. розглянуто ряд типових завдань, для рішення яких необхідна висока продуктивність;
2. вибрані кілька алгоритмів лінійної алгебри;
3. розроблені та налагоджені програми без використання засобів векторної обробки і з використанням засобів векторної обробки;
4. виконано порівняння про час виконання і кількість тактів процесора для розроблених алгоритмів;
5. сформульовані висновки про ефективність використання векторної обробки в комп'ютерах з універсальними процесорами.

В якості подальших досліджень можна вивчити особливості більш складних способів оптимізації – наприклад, ручного використання сучасних наборів SIMD (AVX, AVX2) у критичних ділянках коду.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Руководство Intel Intrinsics Guide. [Електронний ресурс]. Режим доступу: www. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide
2. Означення матриці. Основні операції над матрицями. [Електронний ресурс]. Режим доступу: www. URL: http://www.mathros.net.ua/oznachennja-matryci-osnovni-operacii-nad-matrycjamy.html
3. Множення матриць. [Електронний ресурс]. Режим доступу: www. URL: https://www.berdov.com/works/matrix/umnozhenie-matric/
4. Диференціальне рівняння з частинними похідними. [Електронний ресурс]. Режим доступу: www. URL: http://yukhym.com/uk/prikladi-diferentsialnikh-rivnyan/diferentsialni-rivnyannya-osnovni-ponyattya.html
5. Клепко В. Ю., Голець В. Л. Вища математика в прикладах і задачах 2е видання. Київ “Центр учбової літератури” 2009. С.24.
6. Молодяков С.О. Методологія програмної інженерії та методи паралельної обробки інформації. Міжнародний науковий журнал “Інтернаука” № 1 (23), 1 т., 2017. С. 107-108.
7. Технологии разработки параллельных программ для современных многоядерных процессоров [Електронний ресурс]. Режим доступу: www. URL: https://cyberleninka.ru/article/v/tehnologii-razrabotki-parallelnyh-programm-dlya-sovremennyh-mnogoyadernyh-protsessorov
8. Медведев Ю.С., Дык Данг, Терехов В.В. Анализ современных методов статического и динамического анализа программ и повышения качества программного обеспечения
9. Сборник научных статей VI Международной научно-практической конференции «Научные чтения имени профессора Н.Е. Жуковского» 2015 года. С. 15. [Електронний ресурс]. Режим доступу: www. URL: http://id-yug.com/images/id-yug/Book_id-yug/432-f.pdf#page=18
10. Intel® AVX-512 Instructions and Their Use in the Implementation of Math Functions [Електронний ресурс]. Режим доступу: www. URL: https://composter.com.ua/documents/AVX512.pdf
11. Корячко В.П., Скворцов С.В., Таганов А.И., Шибанов А.П. Эволюция автоматизированного проектирования электронно-вычислительных средств. Радиотехника. 2012. № 3. С. 97–103. 3.
12. MMX (Multimedia Extensions). [Електронний ресурс]. Режим доступу: www. URL: https://www.chaynikam.info/ukr/cpu_mmx.html
13. Скворцов С.В. Оптимизация кода для суперскалярных процессоров с использованием дизъюнктивных графов. Программирование. 1996. № 2. С. 41–52.

14. Бакулев А.В. Модели и алгоритмы организации мобильных параллельных вычислений в среде многоядерных процессоров. Диссертация на соискание ученой степени кандидата технических наук. Рязань: РГРТУ, 2010. 177 с.

15. Команды MMX. [Електронний ресурс]. Режим доступу: www. URL: http://www.codenet.ru/progr/optimize/mmx.php

16. Першин А.С., Скворцов С.В. Распределение регистровой памяти в системах параллельной обработки данных. Системы управления и информационные технологии. 2007. № 1 (27). С. 65–70.

17. Bakulev A.B., Bakuleva M.A., Avilkina S.B. Mathematical methods and algorithms of mobile parallel computing on the base of multi-core processors // European researcher. 2012. V. 33. № 11–1. P. 1826–1834.

18. Технології та інструкції, що використовуються в процесорах [Електронний ресурс]. Режим доступу: www. URL: https://www.chaynikam.info/ukr/cpu_tech.html

19. Эффективность автоматической векторизации циклов. [Електронний ресурс]. Режим доступу: www. URL: https://cyberleninka.ru/article/v/effektivnost-avtomaticheskoy-vektorizatsii-tsiklov-na-arhitekturah-intel-64-i-intel-xeon-phi

20. Векторизація. [Електронний ресурс]. Режим доступу: www. URL: https://algorithmica.org/ru/sse

21. SIMD. [Електронний ресурс]. Режим доступу: www. URL: https://zhuanlan.zhihu.com/p/55327037

22. SSE (Streaming SIMD Extentions). [Електронний ресурс]. Режим доступу: www. URL: https://songho.ca/misc/sse/sse.html

23. Регистри SSE & AVX. [Електронний ресурс]. Режим доступу: www. URL: https://tech.io/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx

24. SIMD Continuous Evolution. [Електронний ресурс]. Режим доступу: www. URL: https://chrisadkin.io/2015/06/04/under-the-hood-of-the-batch-engine-simd-with-sql-server-2016-ctp.

25. Способы организации памяти. [Електронний ресурс]. Режим доступу: www. URL: https://studopedia.ru/2_101436_sposobi-organizatsii-pamyati.html

26. Операции над матрицами, свойства операций. [Електронний ресурс]. Режим доступу: www. URL: http://www.cleverstudents.ru/matrix/operations_on_matrices.html

27. Умножение матриц. [Електронний ресурс]. Режим доступу: www. URL: https://www.wikiwand.com/ru/%D0%A3%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86

28. Алгоритм умножения матриц. [Електронний ресурс]. Режим доступу: www. URL: https://www.math10.com/ru/vysshaya-matematika/matrix/umnozhenie-matric.html

29. Типы данных. [Електронний ресурс]. Режим доступу: www. URL: https://metanit.com/cpp/tutorial/2.3.php.

30. Visual Studio 2019. [Електронний ресурс]. Режим доступу: www. URL: https://visualstudio.microsoft.com/ru/vs/

31. Ускорьте выполнение ресурсоемких рабочих нагрузок. [Електронний ресурс]. Режим доступу: www. URL: https://www.intel.ru/content/www/ru/ru/architecture-and-technology/avx-512-overview.html

32. Цикл for в C++. [Електронний ресурс]. Режим доступу: www. URL: http://cppstudio.com/post/348/

33. Лінійна комбінація . [Електронний ресурс]. Режим доступу: www. URL: https://uk.wikipedia.org/wiki/%D0%9B%D1%96%D0%BD%D1%96%D0%B9%D0%BD%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B1%D1%96%D0%BD%D0%B0%D1%86%D1%96%D1%8F

34. Как транспонировать матрицу. [Електронний ресурс]. Режим доступу: www. URL: https://ru.wikihow.com/%D1%82%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%BE%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D1%82%D1%8C-%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D1%83

35. Формулы и свойства. [Електронний ресурс]. Режим доступу: www. URL: https://xn--24-6kcaa2awqnc8dd.xn--p1ai/kak-transponirovat-matricu.html

36. Data Type Ranges. [Електронний ресурс]. Режим доступу: www. URL: https://docs.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=vs-2017

37. Как найти обратную матрицу. [Електронний ресурс]. Режим доступу: www. URL: http://www.mathprofi.ru/kak_naiti_obratnuyu_matricu.html

38. Нахождение обратной матрицы. [Електронний ресурс]. Режим доступу: www. URL: http://mathprofi.ru/metod_zhordano_gaussa_nahozhdenie_obratnoi_matricy.html

39. Метод Гаусса — Жордана. [Електронний ресурс]. Режим доступу: www. URL: - https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%93%D0%B0%D1%83%D1%81%D1%81%D0%B0_%E2%80%94_%D0%96%D0%BE%D1%80%D0%B4%D0%B0%D0%BD%D0%B0.

40. Онлайн калькулятор. [Електронний ресурс]. Режим доступу: www. URL: https://ru.onlinemschool.com/math/assistance/matrix/inverse/.

41. Обернення матриць. [Електронний ресурс]. Режим доступу: www. URL: https://math.semestr.ru/matrix/index.php.

42. Технология расширений набора команд. [Електронний ресурс]. Режим доступу: www. URL: Intelhttps://www.intel.ru/content/www/ru/ru/support/articles/000005779/processors.html

Додаток А. Лістинг коду множення матриць

```
#include <immintrin.h>
#include <intrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
// вихідна матриця
union matrix{
float m[4][4];
__m128 row[4];
};
// стандартна реалізація множення матриць
void matrix_multiply_stand(matrix& out, const
matrix& A, const matrix& B){
matrix t; // отримуємо вихідну матрицю
// перемножуємо і додаємо матриці
for (int i = 0; i < 4; ++i){
for (int j = 0; j < 4; ++j){
t.m[i][j] = 0;
for (int k = 0; k < 4; ++k){
t.m[i][j] += A.m[i][k] * B.m[k][j];
}
}
}
out = t;
}
// тестування функцій коду
static void run_standart(matrix* out, const
matrix* A, const matrix* B, int count){
int mask = 0; // встановлюємо маску
for (int i = 0; i < count; i++){
int j = i & mask;
matrix_multiply_stand(out[j], A[j], B[j]);
}
}
// інша лінійна комбінація, що використовує
інструкції AVX на XMM register
static inline __m128 lincomb_AVX_4(const
float* a, const matrix& B){
__m128 result;
result =
_mm_mul_ps(_mm_broadcast_ss(&a[0]),
B.row[0]);
result = _mm_add_ps(result,
_mm_mul_ps(_mm_broadcast_ss(&a[1]),
B.row[1]));
result = _mm_add_ps(result,
_mm_mul_ps(_mm_broadcast_ss(&a[2]),
B.row[2]));
result = _mm_add_ps(result,
_mm_mul_ps(_mm_broadcast_ss(&a[3]),
B.row[3]));
return result;
}
// використовуючи інструкції AVX,
шириною 4
// це може бути краще, якщо A знаходиться
в пам'яті.
void matrix_multiply_AVX_4(matrix& out,
const matrix& A, const matrix& B){
_mm256_zeroupper();
for (int i = 0; i < sizeof(A.m) / sizeof(A.m[0]);
i++){
__m128 outx = lincomb_AVX_4(A.m[0+i], B);
out.row[0 + i] = outx;
}
}
static void run_AVX_4(matrix* out, const
matrix* A, const matrix* B, int count){
int mask = 0; // виставляємо маску
for (int i = 0; i < count; i++){
int j = i & mask;
matrix_multiply_AVX_4(out[j], A[j], B[j]);
}
}
// подвійна лінійна комбінація з
використанням інструкцій AVX за
правилами YMM
static inline __m256
twolincomb_AVX_8(__m256 A01, const
matrix& B){
__m256 result;
result =
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0x00),
_mm256_broadcast_ps(&B.row[0]));
result = _mm256_add_ps(result,
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0x55),
_mm256_broadcast_ps(&B.row[1])));
result = _mm256_add_ps(result,
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0xaa),
_mm256_broadcast_ps(&B.row[2])));
result = _mm256_add_ps(result,
_mm256_mul_ps(_mm256_shuffle_ps(A01,
A01, 0xff),
_mm256_broadcast_ps(&B.row[3])));
return result;
```

```
}
// це має бути помітно швидше з фактичними 256-бітними широкими векторними блоками (Intel)
void matrix_multiply_AVX_8(matrix& out, const matrix& A, const matrix& B){
_mm256_zeroupper();
__m256 A1 = _mm256_loadu_ps(&A.m[0][1]);
__m256 A2 = _mm256_loadu_ps(&A.m[2][3]);
__m256 A3 = _mm256_loadu_ps(&A.m[4][5]);
__m256 A4 = _mm256_loadu_ps(&A.m[6][7]);
__m256 A5 = _mm256_loadu_ps(&A.m[8][9]);
__m256 A6 = _mm256_loadu_ps(&A.m[10][11]);
__m256 A7 = _mm256_loadu_ps(&A.m[12][13]);
__m256 A8 = _mm256_loadu_ps(&A.m[14][15]);
__m256 A9 = _mm256_loadu_ps(&A.m[16][17]);
__m256 A10 = _mm256_loadu_ps(&A.m[18][19]);
__m256 A11 = _mm256_loadu_ps(&A.m[20][21]);
__m256 A12 = _mm256_loadu_ps(&A.m[22][23]);
__m256 A13 = _mm256_loadu_ps(&A.m[24][25]);
__m256 A14 = _mm256_loadu_ps(&A.m[26][27]);
__m256 A15 = _mm256_loadu_ps(&A.m[28][29]);
__m256 A16 = _mm256_loadu_ps(&A.m[30][31]);
__m256 out1 = twolincomb_AVX_8(A1, B);
__m256 out2 = twolincomb_AVX_8(A2, B);
__m256 out3 = twolincomb_AVX_8(A3, B);
__m256 out4 = twolincomb_AVX_8(A4, B);
__m256 out5 = twolincomb_AVX_8(A5, B);
__m256 out6 = twolincomb_AVX_8(A6, B);
__m256 out7 = twolincomb_AVX_8(A7, B);
__m256 out8 = twolincomb_AVX_8(A8, B);
__m256 out9 = twolincomb_AVX_8(A9, B);
__m256 out10 = twolincomb_AVX_8(A10, B);
__m256 out11 = twolincomb_AVX_8(A11, B);
__m256 out12 = twolincomb_AVX_8(A12, B);
__m256 out13 = twolincomb_AVX_8(A13, B);
__m256 out14 = twolincomb_AVX_8(A14, B);
__m256 out15 = twolincomb_AVX_8(A15, B);
__m256 out16 = twolincomb_AVX_8(A16, B);
_mm256_storeu_ps(&out.m[0][1], out1);
_mm256_storeu_ps(&out.m[2][3], out2);
_mm256_storeu_ps(&out.m[4][5], out3);
_mm256_storeu_ps(&out.m[6][7], out4);
_mm256_storeu_ps(&out.m[8][9], out5);
_mm256_storeu_ps(&out.m[10][11], out6);
_mm256_storeu_ps(&out.m[12][13], out7);
_mm256_storeu_ps(&out.m[14][15], out8);
_mm256_storeu_ps(&out.m[16][17], out9);
_mm256_storeu_ps(&out.m[18][19], out10);
_mm256_storeu_ps(&out.m[20][21], out11);
_mm256_storeu_ps(&out.m[22][23], out12);
_mm256_storeu_ps(&out.m[24][25], out13);
_mm256_storeu_ps(&out.m[26][27], out14);
_mm256_storeu_ps(&out.m[28][29], out15);
_mm256_storeu_ps(&out.m[30][31], out16);
}
static void run_AVX_8(matrix* out, const matrix* A, const matrix* B, int count){
int mask = 0; // виставляємо маску
for (int i = 0; i < count; i++){
int j = i & mask;
matrix_multiply_AVX_8(out[j], A[j], B[j]);
}
}
// рандомних матриця 4x4, 8x8, 16x16, 32x32
static void random_matrix(matrix& M){
for (int i = 0; i < 4; i++){
for (int j = 0; j < 4; j++)
{
M.m[i][j] = (rand() - 16384.0f) / 1024.0f;
}
}
}
// головна програма
int main(int argc, char** argv){
printf("ok.\n");

// тести
static const struct {
const char* name;
void (*run)(matrix* out, const matrix* A, const matrix* B, int count);
}
perf_variants[] = {
{ "standart", run_standart },
{ "AVX_4",  run_AVX_4 },
{ "AVX_8",  run_AVX_8 },
};
```

```
static const int nperfvars =
(int)(sizeof(perf_variants) /
sizeof(*perf_variants));
matrix Aperf, Bperf, out;
random_matrix(Aperf);
random_matrix(Bperf);
for (int i = 0; i < nperfvars; i++){
clock_t start = clock();
static const int nruns = 1024;//1024 кількість
циклів
static const int muls_per_run = 4096; //4096
unsigned long long best_time = ~0ull;
for (int run = 0; run < nruns; run++){
unsigned __int64 time = __rdtsc();
perf_variants[i].run(&out, &Aperf, &Bperf,
muls_per_run);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
if (time < best_time)
best_time = time;
}
clock_t end = clock();
double seconds = (double)(end - start) /
CLOCKS_PER_SEC;
printf("\nThe time: %.2f seconds\n", seconds);
double cycles_per_run = (double)best_time /
(double)muls_per_run;
printf("\n%12s: %.2f tacts\n",
perf_variants[i].name, cycles_per_run);
}
return 0;
}
```

Додаток Б. Лістинг коду транспонування матриць

Програмний алгоритм, транспонування матриць, ***m*** рядків і ***n*** стовпців, без використання векторної обробки.

```
#include <stdio.h>
#include <intrin.h>
#include <time.h>
using namespace std;
#pragma intrinsic(__rdtsc)
#define M 8   // рядки
#define N 16  // стовпці
// транспонування матриць
double* transpose_standart(double* a, int m,
int n){
// виділяємо під матрицю пам'ять
double* b = new double[M * N];
for (int i = 0; i < m; ++i){
for (int j = 0; j < n; ++j){
b[j * m + i] = a[i * n + j];
}
}
delete a;
return b;
}
// ініціалізація матриці
void init_matrix(double* a, int m, int n){
for (int i = 0; i < m; i++) {
for (int j = 0; j < n; j++) {
a[i * n + j] = i * n + j;
}
}
}
// виведення матриці
void print(double* a, int m, int n){
for (int i = 0; i < m; i++){
for (int j = 0; j < n; j++)
printf("%4.f", a[i * n + j]);
printf("\n");
}
}
int main(){
// виділяємо під матрицю пам'ять
double* a = new double[M * N];
init_matrix(a, M, N);
printf("\nsource matrix\n");
print(a, M, N);
// замір часу
unsigned __int64 time = __rdtsc();
clock_t start = clock();
// транспонування
a = transpose_standart(a, M, N);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
printf("\ntranspose matrix\n");
print(a, N, M);
clock_t end = clock();
printf("\nStandart algorithm: "); //standart
printf("tacts : %I64d\n", time);
double seconds = (double)(end - start) /
CLOCKS_PER_SEC;
printf("\nThe time: %.2f seconds\n", seconds);
return 0;
}
```

Програмний алгоритм, транспонування матриць, ***m*** рядків і ***n*** стовпців, з використанням векторної обробки.

```
#include <stdio.h>
#include <intrin.h>
#include <immintrin.h>
#include <time.h>
using namespace std;
#pragma intrinsic(__rdtsc)
#define K 4    // кількість рядків із отриманої
матриці
#define M 8   // рядки
#define N 16  // стовпці
double* transpose_AVX(double* a, int m, int
n){
double* b = new double[M*N];
__m256d t[K], r[K];
// прохід по рядках
for (int i = 0; i < m; i += K){
// прохід по стовпцях
for (int j = 0; j < n; j += K){
```

```
// завантажуємо k рядків із отриманої матриці
for (int k = 0; k < K; k++){
r[k] = _mm256_load_pd(&(a[(i + k) * n + j]));
}
// вихідний квадрат матриці
// 00 01 02 03
// 04 05 06 07
// 08 09 10 11
// 12 13 14 15
t[0] = _mm256_unpacklo_pd(r[0], r[1]);
t[1] = _mm256_unpackhi_pd(r[0], r[1]);
t[2] = _mm256_unpacklo_pd(r[2], r[3]);
t[3] = _mm256_unpackhi_pd(r[2], r[3]);
// заміна старшої половини рядка 0 на молодшу половину 2 рядка
        // 00 04 08 12
r[0] = _mm256_permute2f128_pd(t[0], t[2], 0x20);
// Заміна старшої половини рядка 1 на молодшу половину 3 рядка
        // 01 05 09 13
r[1] = _mm256_permute2f128_pd(t[1], t[3], 0x20);
// Заміна молодшої половини рядка 0 на старшу половину 2 рядка
        // 02 06 10 14
r[2] = _mm256_permute2f128_pd(t[0], t[2], 0x31);
// Заміна молодшої половини рядка 1 на старшу половину 3 рядка
        // 03 07 11 15
r[3] = _mm256_permute2f128_pd(t[1], t[3], 0x31);
// запис результату
for (int k = 0; k < K; k++)
_mm256_storeu_pd(&(b[(j + k)*m + i]), r[k]);
}
}
delete a;
return b;
}
// ініціалізація матриці
void init_matrix(double* a, int m, int n){
for (int i = 0; i < m; i++) {
for (int j = 0; j < n; j++) {
a[i * n + j] = i * n + j;
}
}
}
// виведення матриці
void print(double* a, int m, int n){
for (int i = 0; i < m; i++){
for (int j = 0; j < n; j++)
printf("%4.f", a[i * n + j]);
printf("\n");
}
}
int main(){
// виділяємо під матрицю пам'ять
double* a = new double[M * N];
init_matrix(a, M, N);
printf("\nsource matrix\n");
print(a, M, N);
// замір часу
unsigned __int64 time = __rdtsc();
clock_t start = clock();
// транспонування
a = transpose_AVX(a, M, N);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
printf("\ntranspose matrix\n");
print(a, N, M);
clock_t end = clock();
printf("\nAVX algorithm: "); //AVX
printf("tacts : %I64d\n", time);
double seconds = (double)(end - start) / CLOCKS_PER_SEC;
printf("\nThe time: %.2f seconds\n", seconds);
return 0;
}
```

Додаток С. Лістинг коду звернення матриць

```
#include <stdio.h>
#include <intrin.h>
#include <immintrin.h>
using namespace std;
#pragma intrinsic(__rdtsc)
#define K 4 // зміщення у рядка матриці на 4
double числа
#define N 8 // матриця розміру 8x8
// виведення матриці
void print (double* a){
// прохід по рядках
for (int i = 0; i < N; i++){
// прохід по стовпцях
for (int j = 0; j < N; j++)
// виведення елемента матриці
printf ("%10.5llf", a[i * N + j]);
printf("\n");
}
printf("\n");
}
// ініціалізація матриці
void init (double* a){
// прохід по рядках
for (int i = 0; i < N; i++)
// прохід по стовпцях
for (int j = 0; j < N; j++){
// елемент дорівнює значенню свого індексу
в матриці
a[i * N + j] = (double)(i * N + j);
}
}
// обмін рядками
// передаємо відразу вихідну і додаткову
матрицю
// і індекси поточного рядка і стовпця
int SwapLine (double* a, double* b, int i, int
j){
// запам'ятаємо вихідний рядок
// і перейдемо на наступну
int k = i++;
double tmp;     // тимчасовий
// пошук рядка з першим елементом не
дорівнює 0
// йдемо по рядках
for (; i < N; i++){
// порівняння з 0
if (a[i * N + j] != 0){
// цикл обміну рядків по всій довжині
for (j = 0; j < N; j++){
// спочатку для вихідного масиву
// беремо поточний елемент
tmp = a[k * N + j];
// пишемо на це місце їх іншого рядка
a[k * N + j] = a[i * N + j];
// в інший рядок з іншої
a[i * N + j] = tmp;
// те ж саме для додаткової матриці
tmp = b[k * N + j];
b[k * N + j] = b[i * N + j];
b[i * N + j] = tmp;
}
return i; // повертаємо індекс
}
}
// або помилку
return -1;
}
// алгоритм простого методу по Гаусса-
Жордана
double* SimpleInverse(double* a){
// додаткова матриця де буде результат
зворотної
double* b = new double[N * N];

// приводимо до одиничної
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++){
// головна діагональ в 1
if (i == j) b[i * N + j] = 1;
else b[i * N + j] = 0; // решта в 0
}
double dta = 0;
// прохід по стовпцях
for (int j = 0; j < N; j++){
// обмін рядками якщо є необхідність
if (a[j * N + j] == 0){
if (SwapLine (a, b, j, j) == -1)
return NULL; // у разі помилки вихід
}
// головну діагональ вихідної матриці
приводимо до 1
dta = a[j * N + j]; // беремо перший елемент
рядка
for (int i = 0; i < N; i++){
// ділимо весь рядок на взяті елемент
a[j * N + i] /= dta;
b[j * N + i] /= dta;
}
```

```
// прохід по рядках вниз привести до 0
// прямий хід
for (int i = j + 1; i < N; i++){
// беремо коефіцієнт
dta = a[i * N + j];
for (int k = 0; k < N; k++){
// віднімаємо рядок помножену на
коефіцієнт
a[i * N + k] -= dta * a[j * N + k];
b[i * N + k] -= dta * b[j * N + k];
}
}
}
// зворотний хід приводимо до 0 вище
головної діагоналі
// прохід по стовпцях
for (int j = N - 1; j >= 0; j--){
// прохід по рядках вгору
for (int i = j - 1; i >= 0; i--){
// беремо коефіцієнт
dta = a[i * N + j];
for (int k = 0; k < N; k++){
// віднімаємо рядок помножену на
коефіцієнт
a[i * N + k] -= dta * a[j * N + k];
b[i * N + k] -= dta * b[j * N + k];
}
}
}
return b; // повертаємо зворотну матрицю
}
// обмін рядками
// передаємо відразу вихідну і додаткову
матрицю
// і індекси поточного рядка і стовпця
int AVXSwapLine (double* a, double* b, int i,
int j){
int k = i++; // запам'ятаємо рядок
__m256d row1, row2;
for (; i < N; i++){
if (a[i * N + j] != 0){
for (j = 0; j < N; j += K){
// беремо по 4 double з матриці для
вихідного рядка
row1 = _mm256_load_pd(&(a[k * N + j]));
// і для другої
row2 = _mm256_load_pd(&(a[i * N + j]));
// міняємо місцями в пам'яті матриці 4
double
_mm256_storeu_pd(&(a[k * N + j]), row2);
_mm256_storeu_pd(&(a[i * N + j]), row1);
// теж саме для додаткової матриці
row1 = _mm256_load_pd(&(b[k * N + j]));
row2 = _mm256_load_pd(&(b[i * N + j]));
// міняємо місцями в пам'яті матриці
_mm256_storeu_pd(&(b[k * N + j]), row2);
_mm256_storeu_pd(&(b[i * N + j]), row1);
}
return i;
}
}
return -1;
}
// алгоритм для avx
// аналогічно як для простого методу, за
винятком
// операцій прискорюють виконання ~ в 4
рази
double* AVXInverse (double* a){
double* b = new double [N * N];
__m256d row1, row2, row3, row4, tmp;
// приводимо до одиничної матриці
додаткову
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++){
if (i == j) b[i * N + j] = 1;
else b[i * N + j] = 0;
}
double dta = 0;
// прохід по стовпцях
for (int j = 0; j < N; j++){
// обмін рядками
if (a[j * N + j] == 0)
{
if (AVXSwapLine (a, b, j, j) == -1)
return NULL;
}
// приведення на головній діагоналі вихідної
матриці до 1
dta = a[j * N + j]; // делитель
// виставити всі 4 double в дільник
tmp = _mm256_set1_pd(dta);
// прохід по всьому рядка зі зміщенням в 4
for (int i = 0; i < N; i += K){
// беремо з вихідної матриці
row1 = _mm256_load_pd(&(a[j * N + i]));
// беремо значення з додатковою
row2 = _mm256_load_pd(&(b[j * N + i]));
// ділимо рядки на взятий коефіцієнт
row1 = _mm256_div_pd(row1, tmp);
row2 = _mm256_div_pd(row2, tmp);
// пишемо назад в пам'ять
_mm256_storeu_pd(&(a[j * N + i]), row1);
_mm256_storeu_pd(&(b[j * N + i]), row2);
```

```
}
// прямий хід, нижче головної діагоналі
вихідної в 0
// додаткова тільки реагує на дії над
вихідною матрицею
for (int i = j + 1; i < N; i++){
// беремо коефіцієнт
dta = a[i * N + j];
// заповнюємо коефіцієнтом всі 4
tmp = _mm256_set1_pd(dta);
// проходимся по рядку
for (int k = 0; k < N; k += K){
// беремо з матриць значення по 4
row1 = _mm256_load_pd(&(a[j * N + k]));
row2 = _mm256_load_pd(&(b[j * N + k]));
// множимо на коефіцієнт
row1 = _mm256_mul_pd(row1, tmp);
row2 = _mm256_mul_pd(row2, tmp);
// беремо значення куди будемо писати в
інший рядок результати
row3 = _mm256_load_pd(&(a[i * N + k]));
row4 = _mm256_load_pd(&(b[i * N + k]));
// віднімаємо множне
row3 = _mm256_sub_pd(row3, row1);
row4 = _mm256_sub_pd(row4, row2);
// і пишемо назад результати
_mm256_storeu_pd(&(a[i * N + k]), row3);
_mm256_storeu_pd(&(b[i * N + k]), row4);
}
}
}
// зворотній хід вище головної діагоналі
// прохід по стовпцях
for (int j = N - 1; j >= 0; j--){
// прохід по рядках вгору привести до 0
for (int i = j - 1; i >= 0; i--){
// беремо коефіцієнт
dta = a[i * N + j];
// наводимо всі 4 коефіцієнтом
tmp = _mm256_set1_pd(dta);
for (int k = 0; k < N; k += K)
{
// беремо рядки які будемо віднімати
row1 = _mm256_load_pd(&(a[j * N + k]));
row2 = _mm256_load_pd(&(b[j * N + k]));
// множимо на коефіцієнт
row1 = _mm256_mul_pd(row1, tmp);
row2 = _mm256_mul_pd(row2, tmp);
// значення для віднімання
row3 = _mm256_load_pd(&(a[i * N + k]));
row4 = _mm256_load_pd(&(b[i * N + k]));
// віднімати
row3 = _mm256_sub_pd(row3, row1);
row4 = _mm256_sub_pd(row4, row2);
// пишемо назад
_mm256_storeu_pd(&(a[i * N + k]), row3);
_mm256_storeu_pd(&(b[i * N + k]), row4);
}}}
return b; // повертаємо зворотну матрицю
}
// головна програма
int main(){
// виділяємо під матрицю
double* a = new double [N * N];
init(a);
printf ("\nsource matrix\n");
print(a);
// замір часу
unsigned __int64 time = __rdtsc ();
a = SimpleInverse(a);
// розрахунок часу виконання в тактах
time = __rdtsc() - time;
printf ("\nsimple inverse matrix\n");
print(a);
printf ("\nrdtsc: %I64d", time);
init(a);
printf ("\nsource matrix\n");
print(a);
// замір часу
time = __rdtsc ();
a = AVXInverse(a);
// розрахунок часу виконання в тактах
time = __rdtsc () - time;
printf ("\avx inverse matrix\n");
print(a);
printf ("\nrdtsc: %I64d", time);
getchar ();
return 0;
}
```

Додаток Д. Комп’ютерна презентація

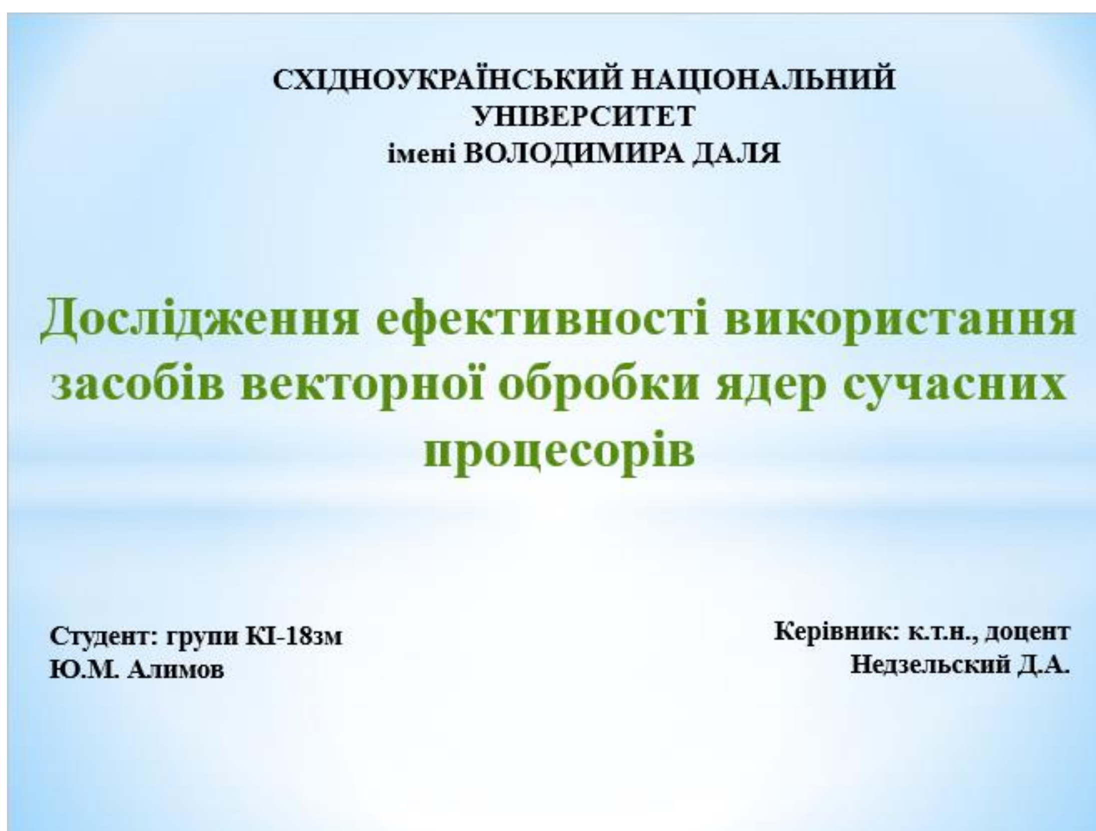

Рисунок Д.1 – Слайд №1

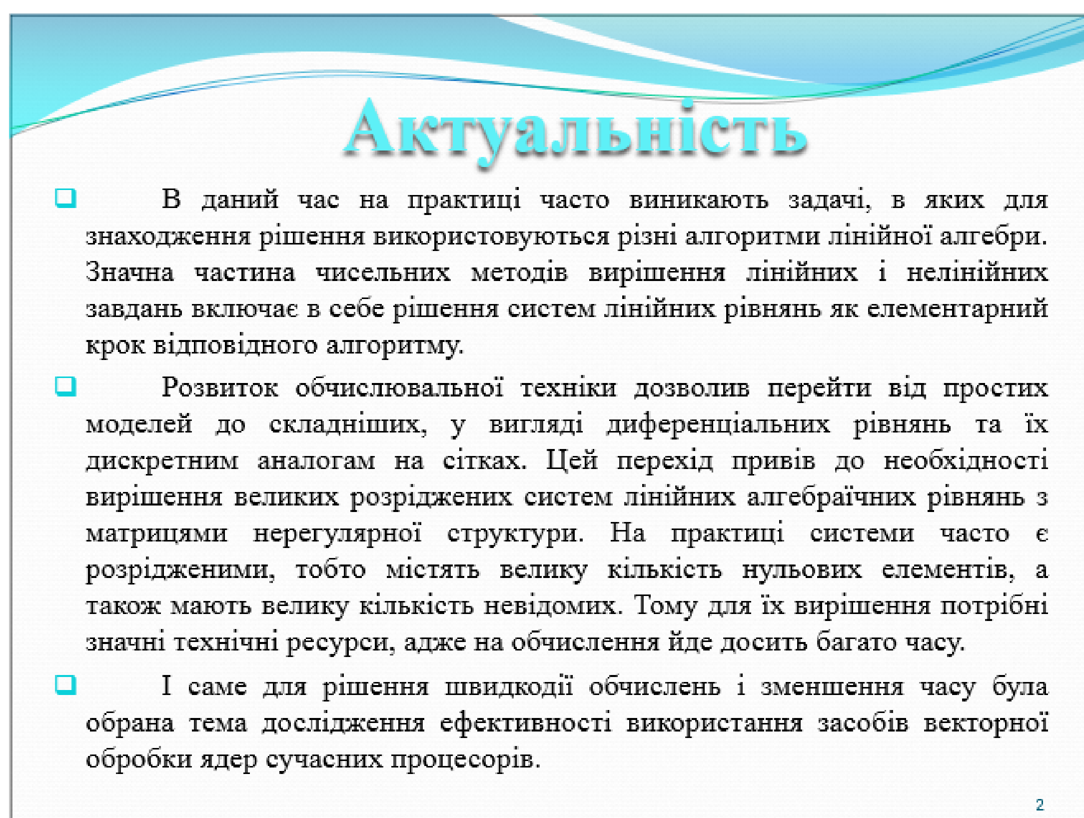

Рисунок Д.2 – Слайд №2

Мета

Метою магістерської роботи є дослідження ефективності використання засобів векторної обробки ядер сучасних процесорів при вирішенні ряду завдань комп'ютерами з універсальними високопродуктивними процесорами.

Основні задачі магістерської роботи:

- розглянути ряд типових завдань, для рішення яких необхідна висока продуктивність;
- вибрати кілька алгоритмів з лінійною алгеброю;
- розробити і налагодити програми, без векторної обробки і з векторною обробки;
- зробити порівняння про час виконання і кількості тактів процесору для розроблених алгоритмів;
- зробити висновки про ефективність використання векторної обробки в комп'ютерах з універсальними процесорами.

3

Рисунок Д.3 – Слайд №3

Переваги AVX інструкцій

- Впровадження векторних інструкцій в алгоритми дозволить скоротити витрати на обчислення.
- Завдяки цим засобам можна отримати більш ефективний алгоритм (за критерієм часу виконання).

4

Рисунок Д.4 – Слайд №4

Розділ 1

Аналіз засобів векторної обробки

- Векторизація - це модифікація коду, яка замінює скалярний код на векторний. Тобто скалярні дані упаковуються в вектора і скалярні операції замінюються на операції з векторами.
- Векторні обчислення - це вид паралельних обчислень з паралелізмом на рівні даних (SIMD - Single Instruction Multiple Data)

Scalar loop

```
for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

SIMD loop (4 elements)

```
for (int i = 0; i < N; i += 4)
    c[i:4] = a[i:4] + b[i:4];
```

	x	y	z	w
a[i:4]	1.0	2.0	3.0	4.0
	+	+	+	+
b[i:4]	5.0	6.0	7.0	8.0
c[i:4]	6.0	8.0	10.0	12.0

5

Рисунок Д.5 – Слайд №5

Розділ 1

Векторні розширення включають

- Векторні регістри зберігають безлічі скалярних значень:
- mm0-mm7, xmm0-xmm15, ymm0-ymm15.
- Векторні команди використовуються для роботи з векторними регістрами:
- paddb mm1, mm3;
- mulpd xmm0, xmm1.

Особливості векторних команд роботи з пам'яттю

- Існують команди вирівняного і невирівняного звернення до пам'яті.
- Команди вирівняного звернення вимагають збалансованої адреси та працюють максимально швидко.
- Команди невирівняного звернення можуть звертатися з будь-якої адреси, але можуть працювати повільніше.

6

Рисунок Д.6 – Слайд №6

Рисунок Д.7 – Слайд №7

Розділ 2

Дослідження типових алгоритмів і розробка програм

- ✓ було досліджено звичайні алгоритми з лінійної алгебри.
- ✓ були розроблені програми зі стандартним методами та із застосуванням AVX інструкцій для досліджених алгоритмів.
- ✓ вибрано середовище розробки Microsoft Visual Studio 2019 на платформі win32 мови програмування високого рівня C++.
- ✓ виконано тестування програми, а так само налагодження цих програми.

8

Рисунок Д.8 – Слайд №8

Розділ 2

Опис алгоритму множення матриць

- Множення матриці A на матрицю B визначено, лише коли число стовпців першої матриці в добутку дорівнює числу рядків другої. Тоді добутком матриць A розміру $m \times k$ і матриць B розміру $k \times n$ називається матриця C розміру $m \times n$ рис 2.1, кожен елемент якої c_{ij} дорівнює сумі попарних добутків рис. 2.2 елементів i-го рядка матриці A на відповідні елементи j-го стовпця матриці B, тобто:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} = \sum_{s=1}^{n} a_{is}b_{sj}$$

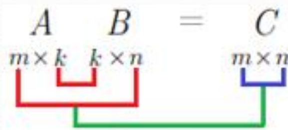

Рисунок 2.1 Матриця **C**

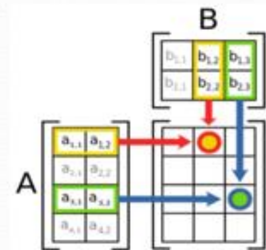

Рисунок 2.2 Попарні добутки

9

Рисунок Д.9 – Слайд №9

Розділ 2

Програмна реалізація алгоритму множення матриць

```
85
86
87    // подвійна лінійна комбінація з використанням інструкцій AVX за правилами YMM
88   static inline __m256 twolincomb_AVX_8(__m256 A01, const matrix& B)
89    {
90        __m256 result;
91        result = _mm256_mul_ps(_mm256_shuffle_ps(A01, A01, 0x00), _mm256_broadcast_ps(&B.row[0]));
92        result = _mm256_add_ps(result, _mm256_mul_ps(_mm256_shuffle_ps(A01, A01, 0x55), _mm256_broadcast_ps(&B.row[1])));
93        result = _mm256_add_ps(result, _mm256_mul_ps(_mm256_shuffle_ps(A01, A01, 0xaa), _mm256_broadcast_ps(&B.row[2])));
94        result = _mm256_add_ps(result, _mm256_mul_ps(_mm256_shuffle_ps(A01, A01, 0xff), _mm256_broadcast_ps(&B.row[3])));
95        return result;
96    }
97
98    // це має бути помітно швидше з фактичними 256-бітними широкими векторними блоками (Intel);
99   void matrix_multiply_AVX_8(matrix& out, const matrix& A, const matrix& B)
100   {
101
102       _mm256_zeroupper();
103
104       __m256 A1 = _mm256_loadu_ps(&A.m[0][1]);
105       __m256 A2 = _mm256_loadu_ps(&A.m[2][3]);
106       __m256 A3 = _mm256_loadu_ps(&A.m[4][5]);
107       __m256 A4 = _mm256_loadu_ps(&A.m[6][7]);
108       __m256 A5 = _mm256_loadu_ps(&A.m[8][9]);
109       __m256 A6 = _mm256_loadu_ps(&A.m[10][11]);
110       __m256 A7 = _mm256_loadu_ps(&A.m[12][13]);
111       __m256 A8 = _mm256_loadu_ps(&A.m[14][15]);
112       __m256 A9 = _mm256_loadu_ps(&A.m[16][17]);
113       __m256 A10 = _mm256_loadu_ps(&A.m[18][19]);
114       __m256 A11 = _mm256_loadu_ps(&A.m[20][21]);
115       __m256 A12 = _mm256_loadu_ps(&A.m[22][23]);
116       __m256 A13 = _mm256_loadu_ps(&A.m[24][25]);
117       __m256 A14 = _mm256_loadu_ps(&A.m[26][27]);
```

10

Рисунок Д.10 – Слайд №10

Розділ 2

Опис алгоритму транспонування матриць

- **Визначення.** Перехід від матриці A до матриці A^T, в якій рядки і стовпці помінялися місцями зі збереженням порядку, називається транспонуванням матриці.
- Матриця A^T є транспонованою до матриці A .

$$A^T = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix} \quad A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

11

Рисунок Д.11 – Слайд №11

Розділ 2

Програмний алгоритм транспонування матриць

Transpose.cpp*

Transpose (Global Scope)

```
22      // прохід по стовпцях
23      for (int j = 0; j < n; j += K)
24      {
25          // завантажуємо k рядків з матриці
26          for (int k = 0; k < K; k++)
27          {
28              r[k] = _mm256_load_pd(&(a[(i + k) * n + j]));
29          }
30
31          // вихідний квадрант
32          // 00 01 02 03
33          // 04 05 06 07
34          // 08 09 10 11
35          // 12 13 14 15
36
37          t[0] = _mm256_unpacklo_pd(r[0], r[1]);
38          t[1] = _mm256_unpackhi_pd(r[0], r[1]);
39          t[2] = _mm256_unpacklo_pd(r[2], r[3]);
40          t[3] = _mm256_unpackhi_pd(r[2], r[3]);
41
42          // замінити старші половини строки 0 на молодші половини 2 строки // 00 04 08 12
43          r[0] = _mm256_permute2f128_pd(t[0], t[2], 0x20);
44          // замінити старші половини строки 1 на молодші половинки 3 строки // 01 05 09 13
45          r[1] = _mm256_permute2f128_pd(t[1], t[3], 0x20);
46          // замінити молодші половини строки 0 на старшій половині 2 строки // 02 06 10 14
47          r[2] = _mm256_permute2f128_pd(t[0], t[2], 0x31);
48          // замінити молодші половини строки 1 на старшій половині 3 строки // 03 07 11 15
49          r[3] = _mm256_permute2f128_pd(t[1], t[3], 0x31);
50
51          // запис результату
52          for (int k = 0; k < K; k++)
53              _mm256_storeu_pd(&(b[(j + k) * m + i]), r[k]);
54      }
55  }
```

12

Рисунок Д.12 – Слайд №12

Розділ 2

Опис алгоритму звернення матриць

- **визначення**. матриця називається зворотного по відношенню до квадратної матриці, якщо при множенні цієї матриці на зворотну, як справа, так і зліва виходить одинична матриця:
- з визначення випливає, що тільки квадратна матриця має зворотну, при цьому зворотна матриця також є квадратної того ж порядку, однак не кожна квадратна матриця має зворотну.

1	-1	-3	5
81	0	58	43
0	32	0	-12
-31	0	41	13

Вихідна матриця

$A^{-1}=$

-0,0338	0,00767	-0,00106	-0,0133
0,0644	0,000654	0,0333	0,00378
-0,08	0,00524	-0,0025	0,0111
0,172	0,00174	0,00536	0,0101

Обернена матриця

13

Рисунок Д.13 – Слайд №13

Розділ 2

Програмний алгоритм звернення матриць

obratnaya.cpp

obratnaya | (Global Scope) | AVXSwapLine(double * a, double * b, int i, int j)

```
129
130 int AVXSwapLine(double* a, double* b, int i, int j)
131 {
132     int k = i++; // запам'ятаємо рядок
133     __m256d row1, row2;
134 
135     for (; i < N; i++)
136     {
137         if (a[i * N + j] != 0)
138         {
139             for (j = 0; j < N; j += K)
140             {
141                 // беремо по 4 double з матриці для вихідного рядка
142                 row1 = _mm256_load_pd(&(a[k * N + j]));
143                 // і для другої
144                 row2 = _mm256_load_pd(&(a[i * N + j]));
145                 // міняємо місцями в пам'яті матриці 4 double
146                 _mm256_storeu_pd(&(a[k * N + j]), row2);
147                 _mm256_storeu_pd(&(a[i * N + j]), row1);
148                 // теж саме для додаткової матриці
149                 row1 = _mm256_load_pd(&(b[k * N + j]));
150                 row2 = _mm256_load_pd(&(b[i * N + j]));
151                 // міняємо місцями в пам'яті матриці
152                 _mm256_storeu_pd(&(b[k * N + j]), row2);
153                 _mm256_storeu_pd(&(b[i * N + j]), row1);
154             }
155             return i;
156         }
157     }
158     return -1;
159 }
160
```

14

Рисунок Д.14 – Слайд №14

Розділ 3

Аналіз результатів обчислювальних експериментів

- Тести проводились на різних версіях компілятору запуску x86/x64 програми Microsoft Visual Studio, з різними розмірами матриць ***n*** x ***m***.
- Враховуючи результати, отримані від використання розширених наборів інструкцій, було вирішено оцінити ефективність та швидкість виконання подібних оптимізацій в рамках роботи множення, транспонування, звернення матриць.
- В якості критерію оцінки використовувався час, витрачений на виконання певних розрахунків над матрицями різних розмірностей а також підрахунок кількості тактів процесору. Дані розрахунки включали в себе:
  - розробку програм без використання засобів векторної обробки;
  - розробку програм з використанням засобів векторної обробки;
  - порівняння даних про час виконання програм і кількості тактів процесору;
  - висновки про ефективність використання векторної обробки в комп'ютерах з універсальними процесорами.

15

Рисунок Д.15 – Слайд №15

Розділ 3

Результати множення матриць

Множення матриць різних розмірностей **n x m**					
Матриця	Алгоритми	Компілятор x86 x64		Кількість ітерацій nruns	
		Кількість тактів		Час виконання в секундах	
4x4	STANDART	727,29	738,91	1,82 s	1,79 s
	AVX128	619,88	710,80	1.14 s	1,27 s
	AVX256	598,20	542,77	1,12 s	0,96 s
8x8	STANDART	4594,14	4528,52	8,68 s	8,74 s
	AVX128	1154,78	1348,22	2,10 s	2,46 s
	AVX256	1126,60	1010,50	2,00 s	1,85 s
16x16	STANDART	35066,22	35001,08	62,02 s	61,86 s
	AVX128	2241,60	2639,65	4,02 s	4,71 s
	AVX256	1901,10	1767,65	3,43 s	3,16 s
32x32	STANDART	28996,49	300699,52	508,49 s	528,29 s
	AVX128	4391,62	5232,57	7,88 s	9,19 s
	AVX256	4240,78	3951,89	7,59 s	6,99 s

16

Рисунок Д.16 – Слайд №16

Рисунок Д.17 – Слайд №17

Рисунок Д.18 – Слайд №18

Розділ 3

Результати транспонування матриць

Транспортування матриць з різними параметрами					
Алгоритми	Параметри			Виконання програм	
	K – параметр	M – рядки	N – стовпці	tacts	seconds
STANDART	не використовується	32	64	93797	0,00
AVX	4	32	64	41230	0,00
STANDART	не використовується	128	256	923079	0,00
AVX	4	128	256	649963	0,00
STANDART	не використовується	512	1024	113426085	0,04
AVX	4	512	1024	48679154	0,02
STANDART	не використовується	4096	8192	2245409159	0,90
AVX	4	4096	8192	1244661280	0,50
STANDART	не використовується	8192	11000	6608633546	2,65
AVX	4	8192	11000	3864488297	1,55

19

Рисунок Д.19 – Слайд №19

Рисунок Д.20 – Слайд №20

Розділ 3

Результати транспонування матриць

- На зображенні показано роботу транспонування матриці, розміром 8x16, обох алгоритмів без векторної і з векторною обробкою.

```
source matrix
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127

transpose matrix
   0  16  32  48  64  80  96 112
   1  17  33  49  65  81  97 113
   2  18  34  50  66  82  98 114
   3  19  35  51  67  83  99 115
   4  20  36  52  68  84 100 116
   5  21  37  53  69  85 101 117
   6  22  38  54  70  86 102 118
   7  23  39  55  71  87 103 119
   8  24  40  56  72  88 104 120
   9  25  41  57  73  89 105 121
  10  26  42  58  74  90 106 122
  11  27  43  59  75  91 107 123
  12  28  44  60  76  92 108 124
  13  29  45  61  77  93 109 125
  14  30  46  62  78  94 110 126
  15  31  47  63  79  95 111 127

standart algorithm: tacts : 89358

The time: 0.01 seconds
```

```
source matrix
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127

transpose matrix
   0  16  32  48  64  80  96 112
   1  17  33  49  65  81  97 113
   2  18  34  50  66  82  98 114
   3  19  35  51  67  83  99 115
   4  20  36  52  68  84 100 116
   5  21  37  53  69  85 101 117
   6  22  38  54  70  86 102 118
   7  23  39  55  71  87 103 119
   8  24  40  56  72  88 104 120
   9  25  41  57  73  89 105 121
  10  26  42  58  74  90 106 122
  11  27  43  59  75  91 107 123
  12  28  44  60  76  92 108 124
  13  29  45  61  77  93 109 125
  14  30  46  62  78  94 110 126
  15  31  47  63  79  95 111 127

AVX algorithm: tacts : 35452

The time: 0.01 seconds
```

Результат транспонування матриці з 8 рядків і 16 стовпців

21

Рисунок Д.21 – Слайд №21

Розділ 3

Результат звернення матриць

Обернення матриць			
Алгоритми	Параметри		Такти процесору
	M – зміщення double чисел в пам'яті	N – розмір матриць	
STANDART	не використовується	32x32	1538938
AVX	4	32x32	808152
STANDART	не використовується	128x128	63590262
AVX	4	128x128	47140752
STANDART	не використовується	512x512	2915557871
AVX	4	512x512	2781829980
STANDART	не використовується	2000x2000	124048763646
AVX	4	2000x2000	132905481881
STANDART	не використовується	4000x4000	1138619719773
AVX	4	4000x4000	1057021845289
STANDART	не використовується	8000x8000	10715209455232
AVX	4	8000x8000	8903995132158

22

Рисунок Д.22 – Слайд №22

Рисунок Д.23 – Слайд №23

Рисунок Д.24 – Слайд №24

Висновки

- Наявність декількох обчислювальних ядер на сучасному процесорі дає можливість досягнення високої продуктивності програми розподілом обчислень між цими ядрами. Але і це всього лише «можливість». Перетворення однопотокового коду в багатопотоковий вимагає серйозних зусиль. Деякі оптимізуючи компілятори мають автопаралелізатор, який зводить процес створення багатопотокового додатка до процесу додавання однієї опції при компіляції. Але в більшості випадків потрібні зусилля для того, щоб розпаралелити спочатку одно потоковий алгоритм.
- Для досягнення поставленої мети, були вирішені такі завдання:
- ✓ розглянули ряд типових завдань, для рішення яких необхідна висока продуктивність;
- ✓ вибрали кілька алгоритмів з лінійної алгебри;
- ✓ розробили і налагодили програми, без використання засобів векторної обробки і з використанням засобів векторної обробки;
- ✓ зробили порівняння про час виконання і кількості тактів процесору для розроблених алгоритмів.

25

Рисунок Д.25 – Слайд №25

Дякую за увагу!

26

Рисунок Д.26 – Слайд №26