פרקטיקום

Kamatech // next silicon

מגישה: מלכה אבלס

Signature:

*

הכרות עם החברה

* *

דרישות ואפיון

* * *

פיתוח

אודות החברה:

NextSilicon היא חברה טכנולוגית ישראלית המתמחה בפיתוח פתרונות מחשוב מתקדמים, המיועדים להאיץ תהליכי עיבוד נתונים. החברה מתמקדת בשיפור ביצועים בתשתיות מחשוב עתירות משאבים, תוך שימוש באלגוריתמים ייחודיים ובפיתוחי חומרה חדשניים, שמאפשרים גישה יעילה ומהירה יותר לעיבוד נתונים בהיקף רחב (High Performance Computing - HPC).

NEXTSILICON

חזון החברה

להוביל את תחום המחשוב עתיר הביצועים
ולשפר את זמני העיבוד בתחומים מגוונים
.כמו מחקר מדעי, בינה מלאכותית, ופיננסים

ערך מוסף

פתרונות החברה מספקים קיצור משמעותי
של זמני חישוב וצריכת חשמל נמוכה יותר
בהשוואה לפתרונות הקיימים בשוק.

מוקד פיתוח

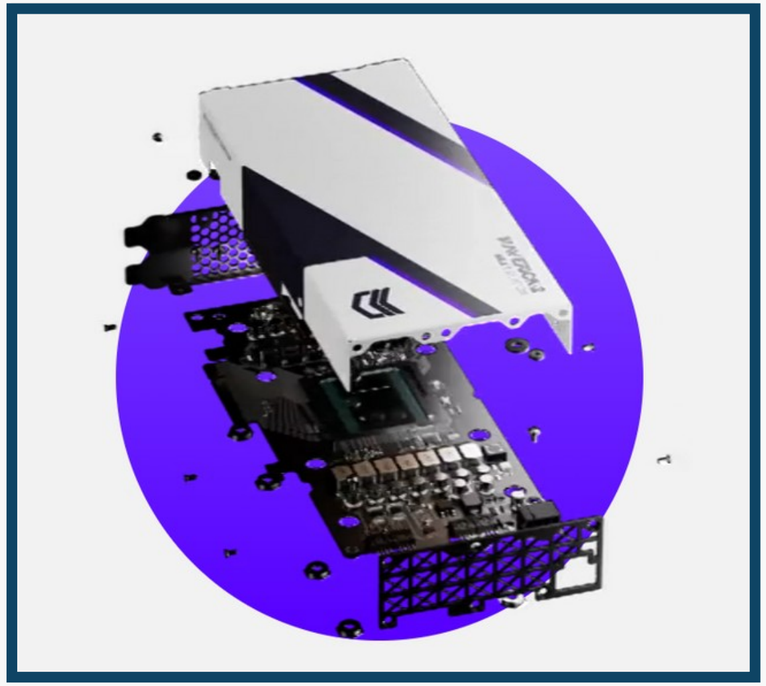
החברה מתמקדת בשילוב בין חומרה
לתוכנה, תוך התאמתם לצרכיי חודיים של
לקוחות מתעשיות מגוונות.

NEXTSILICON

NEXTSILICON

מוצר החברה

Maverick-2 Intelligent Compute Accelerator (ICA) מייצג שינוי מהפכני בתחום המחשוב עם חשיפת האצת החומרה החכמה המוגדרת בתוכנה המספקת יכולת הסתגלות בזמן אמתוביצועים ללא תחרות. טכנולוגיה פורצת דרך זו נועדה לספק ביצועיוויעילות מעולים עבוריישומיHPC, AIויישומי מסדי נתונים וקטוריים.



NEXTSILICON

לקוחות החברה

- **בינה מלאכותית ולמידה עמוקה** – חברות טכנולוגיה וסטארטאפים.
- **מחקר אקדמי ומדעי** – מרכזי מחקר, אוניברסיטאות ומכוני מחקר.
- **פיננסים ובנקאות** – עיבוד נתונים וניהול סיכונים.
- **בריאות וביוטכנולוגיה** – ניתוח גנטי ופיתוח תרופות.
- **אנרגיה ותעשיות הנדסה** –סימולציות מורכבות.

מבנה הצוות בפרויקט:

בנקסט סיליקון פועלים צוותים בתחומי פיתוח שונים. בפרויקט זה פיתחנו כלי עזר למפתחי החברה בהנחיית:

- ראש הצוות שמשתמש בכלי. היא הגדירה את דרישות המוצר.
- מנהל הפרויקטים - עזר בפתרון בעיות מורכבות, ועשה CR לכל אורך התהליך.
- מנחה נוספת שהיתה איתנו לכל אורך התהליך, חילקה משימות, ודאגה להתקדמות יומיומית.

הצוות שלנו כלל שמונה חברות: חמש כתבו ב -Python ושלוש ב -C++ (כולל אני). חלק מהמשימות בוצעו על ידי שתי הקבוצות במקביל, ובסופן נבחרה הגרסה היעילה יותר. משימות אחרות הותאמו לאופי השפה, כמו פיתוח ממשק משתמש (Python) ומימוש Multi-Threading (C++).

מתודולוגיית העבודה:

Weekly עם המנחה מהחברה:

סיכום שבועי עם המנטורית מהחברה לבדיקת התקדמות הפרוויקט, וחידוד אפיון הפרוויקט.

תהליך Code Review:

מפגשים עם מנהל הפרוויקט, לבחינת הקוד והתאמתו לדרישות החברה.

Daily בתחילת היום:

פגישה יומית קצרה עם ראש הצוות לבדיקת ההתקדמות, פתרון בעיות, וחלוקת משימות חדשות.

ניהול משימות וגרסאות:

שימוש ב-GitHub לניהול גרסאות, ניהול משימות וביצועשינויים בצורה מסודרת. כולל יצירתIssues לניהול משימות, וביצוע CR בשיתוף פעולה עם הצוות.

תקשורת ושיתוף פעולה:

תיאום ציפיות רציף ושקיפות מלאה עם כל הגורמים המעורבים בפרוויקט.

תאור המוצר שפיתחנו:

Generic Parser for Telemetry

המוצר שפותח הוא כלי דיבג לחברת נקסט סיליקון. זהו פרסר גנרי, המקבל נתוני טלמטריה בצורה של מידע בינארי, מתוך רכיב החומרה של חברת נקסט סיליקון. הוא הופך את הנתונים למידע יעיל המוצג בטבלאות ב BD.

אפיון ודרישות המוצר:

- קליטת הנתונים בצורה אמינה ויעילה.
- פירוק המידע הגולמי לפורמט מובן, תוך פרסור כל נתון על פי הtype המתאים לו.
- אחסון הנתונים בDB לשימוש עתידי, בדגש על אחסון נכון שיקל על ביצוע אנליזות בעתיד.
- יצירת UI להצגת המידע, הכולל פילטרים ודיאגרמות.

ביצוע ופיתוח:

הכרת הטכנולוגיות, פיתוח וביצוע של תתי המשימות

תוך מתן דגש על כתיבת קוד יעיל וברור.

סביבת העבודה:

בפרויקט השתמשנו במספר טכנולוגיות וכלי פיתוח:

- Visual Studio, VScode - לכתיבת הקוד.
- wsl - windows subsystem for linux- להרצה בסביבת linux על המחשב המקומי.
- שימוש ב - Git, Github - לניהול גרסאות.
- לכתיבת התוכנה השתמשנו בשפת C++, היא מיישמת עקרונות של תכנות פרוצדורלי, תכנות מונחה עצמים ותכנות גנרי. שפה זו מהירה ביותר, והיא מהשפות הפופולריות בקרב מתכנתים בעולם.
- לאינטגרציה בין C++ לפיתון, השתמשנו בספרית PyBind11 שאפשרה לנו לייצא חלקים מהקוד שלנו ב C++ כמודולים, ולפיתון לשלב את המודולים בתוך הקוד שלהם.
- לבניית קוד ה CPP של התוכנה השתמשנו ב Cmake - מערכת בניה בקוד פתוח, שהיא חוצת פלטפורמות ונותנת את האפשרות לעבוד על win I linux במקביל.
- לבדיקת נכונות הקוד כתבנו טסטים על הקוד תוך שימוש בספרית DocTest.

אפיון שלבי העבודה בצוות C++:

קבלת הנתונים.

קריאת קבצים בפורמטtlm, המייצגים packets buffers בגדלים שונים.
העברת המידע למבנה נתונים היעיל ביותר בשביל המשך התהליך.

0	0	0	1	0	1	0	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0
1	0	0	1	1	0	0	1	0	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1							

name	CNTBits	type
num	8	UnsignedLittleEndian
flag	1	bool
str	8	char
sum	32	Floating Precision
cnt	16	UnsignedBigEndian

עיבוד הנתונים.

לקיחת כל מספר ביטים עפ"י קובץ קונפיגורציה, ופרסורם על פי הtype המתאים- גם כן לפי קובץ הקונפיגורציה.
כאן עיקר האתגר היה למצואdesign המתאים לגנריות של הפרסור.

אחסון הנתונים.

התממשקות לסוג DB המתאים, ויצירת טבלאות בצורה היררכית, שתקל על האנליזות בהמשך.

name	value
num	21
flag	1
str	H
sum	11.38
cnt	1003

משימות

מאחר והצוות שלי היה קטן (3),

כל אחד מחברי הצוות לקח חלק פעיל בכל אחד

משלבי הפיתוח.

להלן חלק מהמשימות שהיו בתחום אחריותי.

משימה1.

חשיבה על desing מתאים ללוגיקה של

הפרסור הגנרי, שימוש ב Design Petterns.

משימה1.

הבעיה:

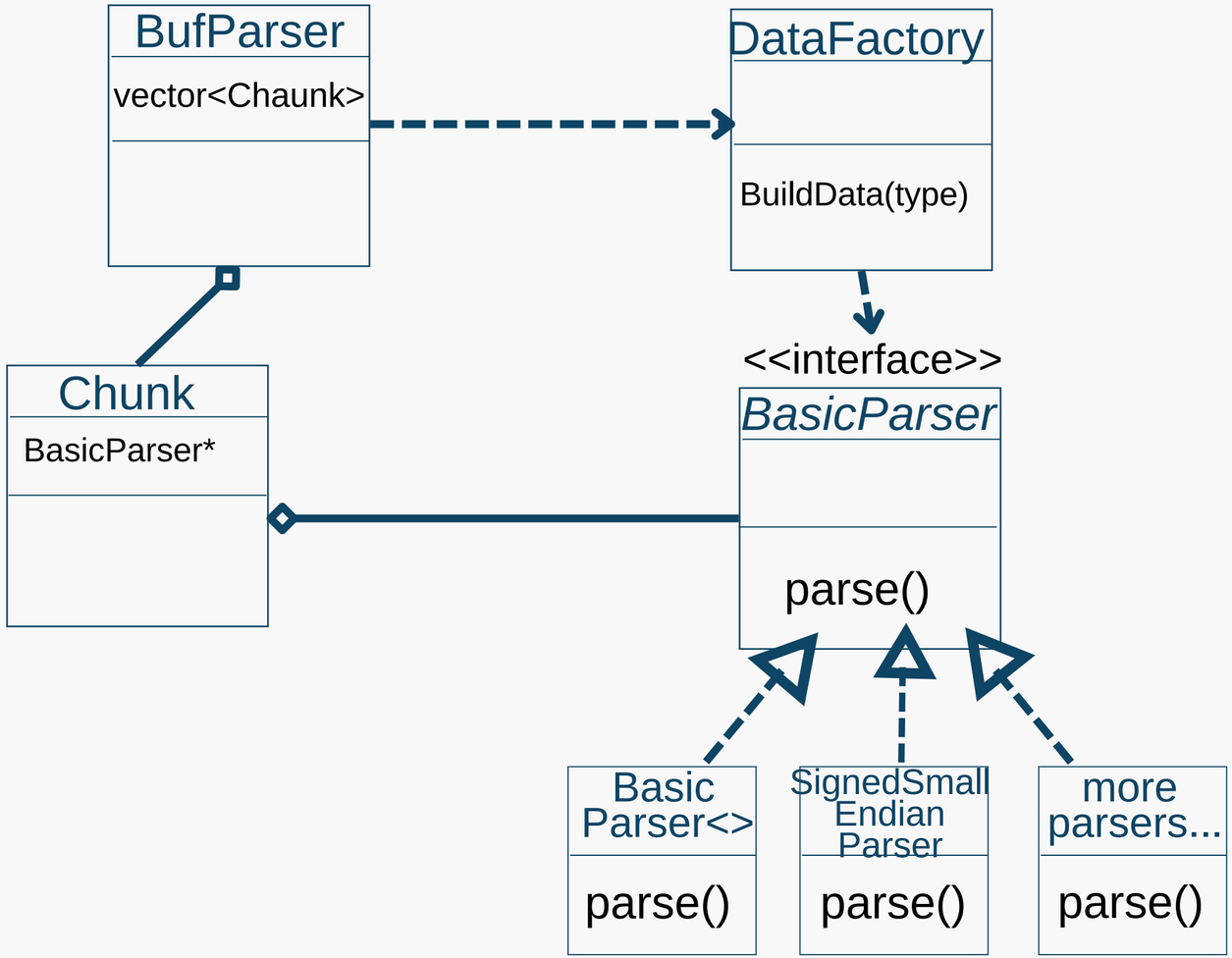
בכדי לפרסר packet שלם, נדרשנו לעבור עליו ברצף, ולשלוח כל רצף סיביות, לפונקציית Parse אחרת, על פי ה Type אותו קראנו מהקובץ קונפיגורציה כ -String.

הפתרון:

השתמשנו ב Design Pattern - Factory.
כך יכולנו לשלוח כל רצף ביטים, ביחד עם ה String שהתקבל, למחלקת הFactory , והיא היתה אחראית לשלוח לפונקצית Parse המתאימה ולהחזיר את התוצאה המפורסרת

משימה1.

תרשים הUML של המחלקות כפי שמומש:



משימה 2.

פיתוח פונקציות Parse גנריות המסוגלות להתמודד עם סוגי נתונים שונים בצורה גמישה, יעילה ומדויקת.

משימה 2.

אתגרים עיקריים:

- המרה מייצוג של ביטים, לנתונים מטיפוס מסוים.
- שמירה על ביצועים, תוך הבטחת תקינות נתונים.
- התאמה לתקן Endianness בהתאם לטיפוס הנדרש, וייצוגו במערכת ההפעלה.

מחקר מקדים ושיקולים:

- הבנת מבנה המייצג של מספרים בפורמטים Floating-Point ו-Signed.
- הבנת יתרונות וחסרונות של שימוש בפוינטרים דינמיים.
- דרכים לתמיכה ב -Endianness גדול וקטן.

דוגמה ליישום Signed Big Endian Parser

משימה 2.

תהליך העבודה של הפונקציה:

- בדיקת תקינות גודל הווקטור.
- טיפול ב-MSB לקביעת סימן (שלילי/חיובי).
- שימוש ב -parse_to_uint64_t להמרת הביטים לערך מספרי.
- טיפול במבנה Two's Complement במקרה של ערכים שליליים.
- התאמת Endianness.

```
void* SignedBigEndian::parse_to_void(std::vector<bool>& bit_array)
{
    if (bit_array.size() > sizeof(signed) * 8) {
        throw std::invalid_argument("Vector size exceeds the size of signed type");
    }
    bool msb = bit_array[0];
    bit_array.erase(bit_array.begin());
    uint64_t value = parse_to_uint64_t(bit_array);
    // Handle two's complement if the number is negative
    if (msb) {
        value = (~value & ((1ULL << (bit_array.size())) - 1)) + 1;
        value = -static_cast<int64_t>(value);
    }
    signed* data = new signed();
    memcpy(data, &value, sizeof(*data));
    *data = BasicParser::Swap_endians<signed>(*data);
    return data;
}
```

דוגמה ליישום Floating Precision Parser

משימה 2.

תהליך העבודה של הפונקציה:

- אתחול ערכים: מקצים זיכרון למספר מסוג float ומאתחלים אותו ל -0.
- חישוב המספר השלם: יוצרים ערך שלם על בסיס הביטים הראשונים בווקטור.
- חישוב ה -Mantissa: מחשבים את החלק העשרוני של המספר משאר הביטים.
- שילוב הערכים: מוסיפים את החלק העשרוני לערך השלם.
- החזרת מצביע: מחזירים את הכתובת של המשתנה שהתקבל.

```
void* FloatingPrecisionParser::parse_to_void(std::vector<bool>& bit_array)
{
    int before = 2;

    float* f = new float();
    *f = 0.0f;
    int mask = 0;
    for (int i = 0; i < before; i++) {    // create the integer number;
        mask <<= 1;
        if (bit_array[i])
            mask |= 1;
    }
    (*f) += mask;
    mask = 0;       //create the mantissa number
    for (int i = before; i < bit_array.size(); i++) {
        mask <<= 1;
        if (bit_array[i])
            mask |= 1;
    }
    int mul = 1, temp = mask;    //calculate the mul of the mask
    while (temp > 9) {
        mul *= 10;
        temp /= 10;
    }
    (*f) += float(mask) / (mul * 10);      //add the mantissa to the value
    return f;
}
```

משימה 3.

חקירה אודות DB שונים, הבנת היתרונות
והחסרונות למול הצרכים שלנו.

משימה 3.

תהליך החקר:

זיהוי הדרישות העיקריות:

- כתיבה מקבילית
- שליפות יעילות לפי חתימות זמן
- גמישות עם סוגי מידע שונים

בחינה של 5 סוגי DB מרכזיים:

- SQLite
- PostgreSQL
- MongiDB
- MySQL

מסקנה:

כדאי לאפשר תמיכה

ב - SQLite(פשטות)

ו - PostgreSQL (יכולות גבוהות יותר)

משימה 3.

מסמך החקר...

	PostgreSQL	MySQL	SQLite	MongoDB	TimescaleDB
Scalability	Billions of rows (10^9+), good performance for thousands of concurrent users	Up to 10^7 on a single server, tens of millions of rows at maximum	Suitable for small projects with up to millions of rows (10^6)	Billions of documents	Billions of rows and terabytes of time-series data
Replication (Is it important?)	Asynchronous replication, suitable for 10-20 servers	Asynchronous replication between 2-3 servers	No support at all	Advanced automatic replication across dozens of servers	Asynchronous replication based on PostgreSQL
Performance	Very fast even under heavy loads (seconds to tens of milliseconds)	Good for small-to-medium systems (hundreds of milliseconds)	Good performance for small projects (tens of thousands of rows)	High performance in millions of requests per second under distributed loads	Optimized for time-series data, excellent for concurrent reads and writes
Simultaneous Read/Write, Concurrency	Excellent support due to MVCC	Supports read/write concurrently, delays may occur under heavy loads	Limited, reads may be blocked during writes	Supports concurrent read/write well under heavy loads	Optimized for continuous reading and writing of time-series data
Ease of Use	Requires extensive knowledge but very flexible for advanced users	Considered user-friendly for beginners	Simple and minimalistic, suited for small projects and learning	Relatively easy with a convenient API, especially for complex documents	Optimized for advanced use, especially with time-series data
Supported Platforms	All the databases listed work with Windows, and are suitable for all platforms we use				
Data Model	Relational	Relational	Relational	Document-based (BSON)	Relational with extensions for time-series
Query Language	SQL	SQL	SQL	JSON-like query language (SQL-like interface)	SQL with time-series extensions
Transactions	Full ACID support (supports complex transactions)	Supports ACID, though with some limitations in certain cases	Supports ACID (limited to simple transactions)	Partial ACID support, including distributed transactions across multiple documents	Full ACID support, like PostgreSQL
Indexing	Wide range of indexes (B-Tree, GIST, GIN, etc.)	B-Tree and other basic indexes	Basic indexes (B-Tree)	B-Tree, Geospatial, Text	Advanced indexes tailored for time-series data
Unique Features	JSONB support, wide data range, many extensions	Good performance for medium-scale projects, supports easy replication	Very lightweight, ideal for mobile or embedded applications	NoSQL, flexible schemas, advanced automatic sharding	Time-series extensions, automatic sharding, handling large time-series datasets

Summary:
The parameters that I believe are most important for us are highlighted. It seems that SQLite, which we've used so far, is too small and simple.
MongoDB – its primary advantage is its non-relational structure, which we don't need.
MySQL – if its scale is sufficient for us, it could be a good choice, with the main advantage being ease of use. If the scale isn't sufficient, we are left with TimescaleDB (based on PostgreSQL) or PostgreSQL itself. Both offer excellent performance. The decision between them depends on whether most of our queries are time-based. Their drawback is their complexity compared to the other options.

משימה 4.

נתינת אפשרות למשתמש לבחור בכל הרצה מחדש באיזה סוג DB הוא מעוניין להשתמש בהרצה זו.

משימה 4.

פתרון שנבחר:

יצירת קובץ קונפיגורציה חיצוני (בפורמט JSON).

הקובץ מאפשר למשתמש לבחור בין SQLite ל -PostgreSQL בעת ריצת התוכנית.

דוגמא לקובץ:

```
{
  "db_type": "PostgreSQL",
  "connection_string": "host=localhost dbname=mydb user=user password=pass"
}
```

משימה 4.

הטמעת התמיכה בשני סוגי ה DB.

```
class WriterInterface {
protected:
    virtual void insertAgent(int id) = 0;
    virtual void createPacketsTable(int id) = 0;
    virtual void insertEventID(int, int) = 0;
    virtual void createEventIdTable(int idTable, int) = 0;
public:
    virtual void openDB(std::string) = 0;
    virtual void insertPacket(const std::vector<Chunk>& chunks,
                              int location, int eventId) = 0;
    virtual void runTransaction() = 0;
    virtual void createGeneralAgentsTable() = 0;
    virtual void dropTable(std::string tableName) = 0;
    virtual std::vector<std::string> readRowFromDB(std::string tableName) = 0;
    virtual void dropDB(std::string) = 0;
    virtual ~WriterInterface() {}
    virtual void resizeInsertStmtsSize(int) = 0;
};
```

הוספנו אינטרפייס שכלל
את הפונקציות הנדרשות
להממשקות ל DB.
לאחר מכן, יצרנו מחלקות מתאימות עבור כל
אחד מסוגי ה DB.
במהלך הריצה, התוכנית ניגשת לקובץ
הקונפיגורציה שהמשתמש הגדיר, וכך יודעת
לאיזה מימוש של האינטרפייס עליה לפנות.

משימה 5.

כתיבת benchmark יעודי לשני חלקי הצוות (C++, פיתון) המודד את זמני הריצה של כל חלק בתוכנית.
התאמת התוכנית של C++, כך שתוציא את הפלט החישובי המתאים ל benchmark.

משימה 5.

מטרות:

- מדידה והשוואה של זמני הריצה בין שני חלקי התוכנית (C++ ו -Python).
- יצירת כלי אובייקטיבי לאיסוף נתונים על ביצועים.

פונקציות עיקריות של ה benchmark:

- ניתוח זמני ריצה על קבצי בדיקה.
- שמירת התוצאות במסד נתונים לצורך חישוב סטטיסטיקות.
- ניהול התאמה אוטומטית בין פורמט פלט של C++ לפלט הקריא ב benchmark.

משימה 5.

אז איך זה עובד?

בכל הפעלה, מתווספת עמודה למסד הנתונים עבור זמן הביצוע של המנתח ותהליך הכתיבה של כל קובץ בספריית הקבצים הבינאריים. ליד כל עמודה, עמודה נוספת מחשבת את אחוז השינוי בהתבסס על ריצות קודמות.

תוצאות לדוגמא של ה benchmark:

id	file_name	parse_run1date2024...	write_run1date2...	pars...	write...	parse_run2date202...	write_run2date20...	parse_rate2date20241...	write_rate2date202410...
1	file_1.tlm	0.189281702041626	11.5172324180603	0.0%	0.0%	0.266046524047852	12.4135112762451	-40.55585995805542%	-7.7820679973372275%
2	file_10.tlm	0.187435626983643	10.37005853652...	0.0%	0.0%	0.25636625289917	12.9617867469788	-36.775626397612406%	-24.992416400733465%
3	file_2.tlm	0.770872592926025	66.80474734306...	0.0%	0.0%	1.55506539344788	114.717771053314	-101.72793892506493%	-71.72098633081023%
4	file_3.tlm	0.876379251480103	82.38655710220...	0.0%	0.0%	1.9355845451355	151.5356798172	-120.86152106710881%	-83.93253116429469%
5	file_4.tlm	0.492505311965942	32.1158685684204	0.0%	0.0%	0.921828508377075	51.2385103702545	-87.17128241671065%	-59.54265805109642%
6	file_5.tlm	0.324883460998535	14.8524181842804	0.0%	0.0%	0.399684429168701	19.4816055297852	-23.02393847328026%	-31.167903354648494%
7	file_6.tlm	0.134221076965332	6.05247569084167	0.0%	0.0%	0.191941976547241	8.41913652420044	-43.004348413042436%	-39.10235999691651%
8	file_7.tlm	0.0641334056854248	3.258056402206...	0.0%	0.0%	0.0830249786376953	4.70609712600708	-29.456681350954472%	-44.4449249810414%
9	file_8.tlm	0.567838668823242	46.6978170871735	0.0%	0.0%	1.17455887794495	78.5449464321136	-106.84728646237467%	-68.19832559943708%
10	file_9.tlm	0.34026575088501	21.077006816864	0.0%	0.0%	0.535982608795166	29.1877043247223	-57.518823860793724%	-38.48125864517353%

משימה 5.

התאמת הקוד ב-c++:

- הוספת מנגנוני מדידת זמן לקוד.
- הוצאת פלט מובנה ומותאם עבור קובץ הבנצ'מרק.
- שמירה על מבנה מודולרי כדי להקל על הרצת הבנצ'מרק.

```
int main() {
    auto start = std::chrono::high_resolution_clock::now();

    // תהליך הפרסור של הקובץ
    parseAndStore_TLMfile("input_file.tlm");

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Runtime: " << duration.count() << " seconds" << std::endl;

    return 0;
}
```

שימוש ב std::chrono-
למדידת זמן ריצה.
פלט התוצאה כפורמט
קריא ל benchmark.

משימה 5.

התוצאה:

- זיהוי יתרונות וחסרונות של כל שפה במונחי ביצועים.

מסקנות:

- ב -C++ נצפה שיפור בביצועים בפרסור קבצים גדולים.
- התאמות בקוד משפרות את שיתוף הפעולה בין שני חלקי התוכנית.

Thank you