נ.נ. בתכנון משולב חומרה/תוכנה

מסמך תיעוד פרויקט

תאריך הגשה: 06.09.2025

מגישים:
אמיר זועבי - 212606222
ניר שיף - 212980395

כל הקוד, כולל אוטומציה לריצה ותוצאות, נמצאים ב:
https://github.com/AmZu1212/SOFTWARE-HARDWARE-CODESIGN
לחצו לפתיחת ה Repository

קצת רקע על Python לפני שנתחיל:

נתמקד במימוש CPython שהוא המימוש הרשמי והנפוץ ביותר.

Python Source Code: הקוד הבסיסי של פייתון, קובץ ה - .py, רצף של תווים.

Lexical Analysis: ה Interpreter הופך את ה - source code של פייתון ליחידות משמעותיות שנקראות tokens. במקרה וישנן טעויות של הזחה או תחביר, הן יתגלו בשלב הזה.

Parsing: בוחן את ה tokens ובונה מהם Abstract Syntax Tree , המתאר את המבנה ההיררכי של הקוד.

Abstract Syntax Tree: מוודא שהסינטקס של הקוד נכון ושהוא עומד בחוקי הדקדוק של פייתון.

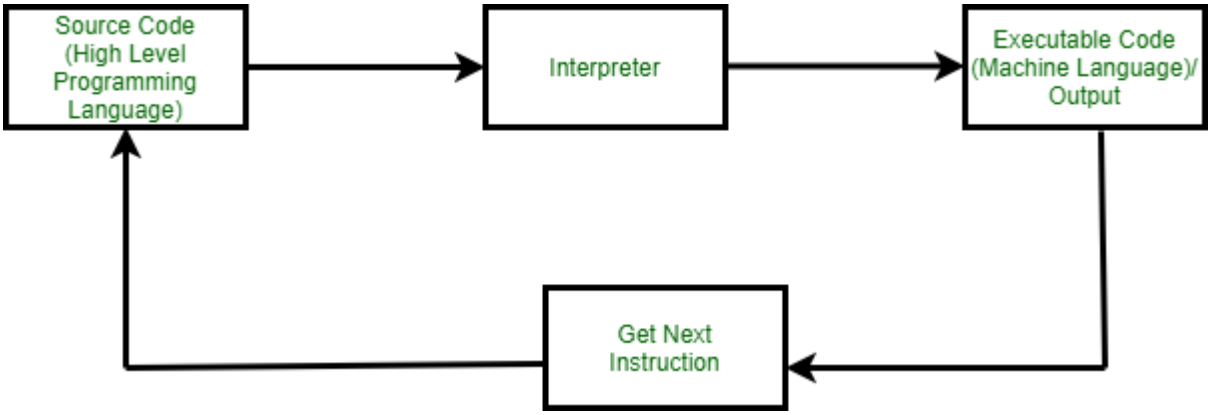
Bytecode Compilation: הפיכת ה - AST לקוד הביניים של פייתון, אשר אינו תלוי בפלטפורמה.

Python Virtual Machine: לולאת Fetch - Decode - Execute הכתובה בשפת c. ה - PVM לוקח כל פקודה מ - Bytecode מפרש אותה ומבצע אותה על ידי קריאה לפונקציות ב C.

Execution: ביצוע הפונקציות שה PVM קרא להן. הפונקציות מומשות בשפת C והן מקומפלות מראש, כלומר הן כבר כתובות בשפת מכונה.

<u>הסבר עבור שימוש בספריות C עם פייתון:</u>

במקרה הזה הקוד עדיין עובר כרגיל עד לשלב ה - Bytecode, אבל כאשר יש שימוש בפונקציית ספריה הוא אינו מתבצע בצעדים ב interpreter אלא מעביר שליטה ישירות לפונקציית C שבדרך כלל כבר קומפלה. בצורה כזו אין צורך לעבור דרך ה interpreter בכל צעד עד הביצוע, יש הרבה פחות overhead והביצוע הרבה יותר מהיר.



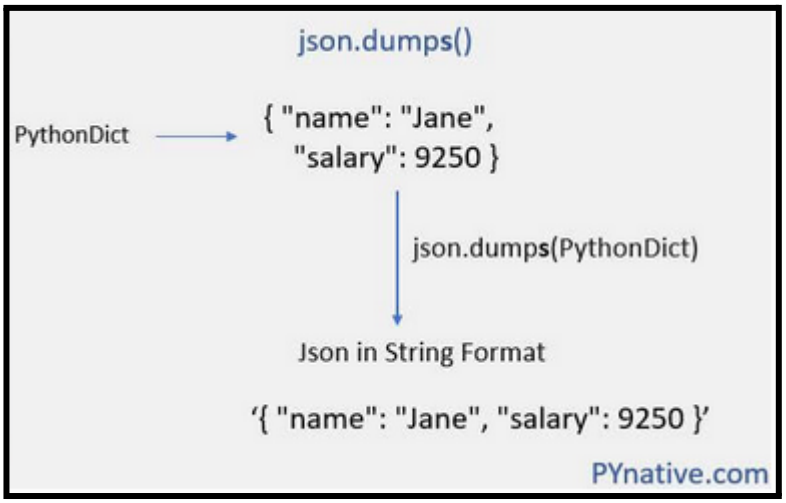
בנצ'מרק 1 - Json Dumps

בחרנו בבנצ'מרק מאחר והוא היה הדוגמה שבחרתם, וגם מאחר שרצינו להתחיל עם משהו קל יותר ל"חימום".
להלן הקוד של ה benchmark של ה json_dumps:

```python
import json
import sys

import pyperf # pyright: ignore[reportMissingImports]


EMPTY = ({}, 2000)
SIMPLE_DATA = {'key1': 0, 'key2': True, 'key3': 'value', 'key4': 'foo',
               'key5': 'string'}
SIMPLE = (SIMPLE_DATA, 1000)
NESTED_DATA = {'key1': 0, 'key2': SIMPLE[0], 'key3': 'value', 'key4': SIMPLE[0],
               'key5': SIMPLE[0], 'key': '\u0105\u0107\u017c'}
NESTED = (NESTED_DATA, 1000)
HUGE = ([NESTED[0]] * 1000, 1)

CASES = ['EMPTY', 'SIMPLE', 'NESTED', 'HUGE']


def bench_json_dumps(data):
    for obj, count_it in data:
        for _ in count_it:
            json.dumps(obj)


def add_cmdline_args(cmd, args):
    if args.cases:
        cmd.extend(("--cases", args.cases))


def main():
    runner = pyperf.Runner(add_cmdline_args=add_cmdline_args)
    runner.argparser.add_argument("--cases",
                                  help="Comma separated list of cases. Available cases: %s. By default, run all cases."
                                       % ', '.join(CASES))
    runner.metadata['description'] = "Benchmark json.dumps()"

    args = runner.parse_args()
    if args.cases:
        cases = []
        for case in args.cases.split(','):
            case = case.strip()
            if case:
                cases.append(case)
        if not cases:
            print("ERROR: empty list of cases")
            sys.exit(1)
    else:
        cases = CASES

    data = []
    for case in cases:
        obj, count = globals()[case]
        data.append((obj, range(count)))

    runner.bench_func('json_dumps', bench_json_dumps, data)


if __name__ == '__main__':
    main()
```

מטרת ה benchmark היא להמיר סוגים של אובייקטים שונים בפייתון למחרוזות בפורמט JSON, בעצרת הרצת הפונקציה `json.dumps` על מספר סוגים שונים של אובייקטים בפייתון, בגדלים\מורכבויות שונות.

הנה תמונה להמחשת פעולת הפונקציה:



courtesy of pynative.com

האובייקטים שהגדירו בנצ'מרק, הינם:

`EMPTY`: מילון ריק (`{}`) שמתבצע עליו 2000 פעמים.

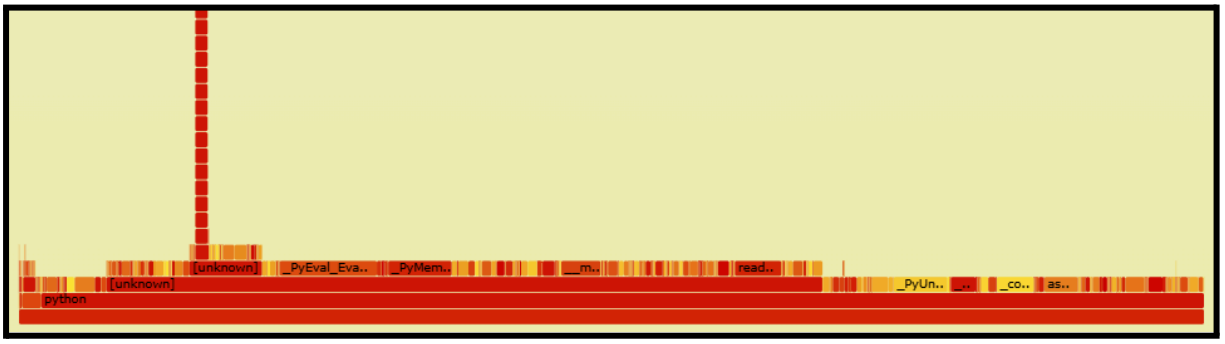
`SIMPLE`: מילון פשוט עם מפתחות וערכים מסוגים בסיסיים (int, bool, string), מתבצע 1000 פעמים.

`NESTED`: מילון מורכב שבתוכו נמצאים גם `SIMPLE`, מתבצע 1000 פעמים.

`HUGE`: רשימה המכילה את `NESTED` אלף פעמים, מתבצע רק פעם אחת (כי היא מאוד כבדה).

הערה: ניתן לבחור על איזה מן האובייקטים לרוץ, למשל בעזרת דגלי SIMPLE,EMPTY,NESTED בשורת הפקודה וכדומה. במדידות שלנו, הרצנו ללא דגלים, כלומר הסקריפט מריץ את כל הסוגים (כפי שביצעתם בהדרכת הפרויקט).

במבט ראשוני על ה flamegraph של הבנצ'מרק, לא כל כך ברור מה ה bottleneck. קשה גם לזהות את הפונקציות בגרף, מאחר ורובם המסומנות כ unknown.



הערה: גם בכלים אחרים, לא היה ברור מהdata לאיזה כיוון ללכת מבחינת אופטימיזציה מאחר ואין bottleneck ברור.

לכן עברנו לקרוא את ה source code של הבנצ'מרק, בשביל אולי למצוא שם משהו בעצמנו.

ניתן לראות כי ה workload ב benchmark הזה מגיע משני גורמים עיקריים:

1. הכמות והסיבוכיות של האובייקטים שנשלחים לפונקציית ההמרה.

2. אופן מימוש הפונקציה `json.dumps` שמבצעת את פעולת הקידוד.

מסיבות ברורות, אין ביכולתנו לשנות את האובייקטים עצמם, שכן שינוי כזה יפגע בטוהר (ובפואנטה) של ההשוואה. עם זאת, ניתן לשפר ביצועים על ידי החלפה או אופטימיזציה של פונקציית הקידוד עצמה.

בתחילת הסקריפט, ניתן לראות כי `json.dumps` ממומשת באמצעות `import` של המודול `json`, מודול סטנדרטי בפייתון. בפועל, זה wrapper שכתוב בפייתון (.py), אשר מפעיל את פונקציית ההמרה האמיתית, המוגדרת בתוך "`CPython`": המימוש הנפוץ של פייתון, הכתוב ב C ומקומפל כחלק מה interpreter, ומגיע כסטנדרט בהתקנות של Linux Ubuntu.

הערה: מאחר ומדובר על קבצי קוד באורך 450 ו 1000 כל אחד, אנו נעבור ונצרף קטעים נבחרים החשובים להסברים בלבד, ואת אופן הפעולה נסביר על קצה המזלג, מספיק בשביל להבין את ההבדל באימפלמנטציה, ואת הסיבה לשינוי בזמן הריצה.

המעטפת מבצעת את הפעולות הבאות:

- אם קיים המימוש ב C (שמו הוא `c_make_encoder`, מתוך המודול `_json` נראה היכן זה מתבצע בהמשך), הקוד ייבחר אוטומטית להשתמש בו, זהו הנתיב המהיר והיעיל יותר.

- אם מסיבה כלשהי המימוש ב C לא קיים (למשל, אם ה interpreter קומפל ללא הקובץ `_json.c`), קיימת גרסת גיבוי של פונקציית ההמרה, הכתובה כולה בפייתון. והיא כמובן איטית יותר, אבל מבטיחה תקינות.

ההחלטה מתבצעת כאן (לקוח מתוך `cpython/Lib/json/encoder.py`):

```
251         if _one_shot and c_make_encoder is not None:
252             _iterencode = c_make_encoder(
253                 markers, self.default, _encoder, indent,
254                 self.key_separator, self.item_separator, self.sort_keys,
255                 self.skipkeys, self.allow_nan)
256         else:
257             _iterencode = _make_iterencode(
258                 markers, self.default, _encoder, indent, floatstr,
259                 self.key_separator, self.item_separator, self.sort_keys,
260                 self.skipkeys, _one_shot)
261         return _iterencode(o, 0)
```

C-Call

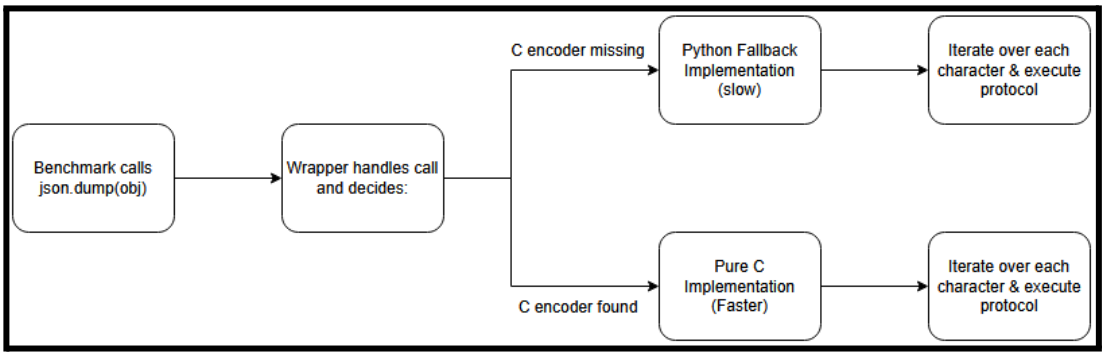
Python Back-up

ברור לנו, שהמימוש C הוא העדיף כאן מבחינת זמן הריצה. מכאן, ניתן להסיק שהשיפור בביצועי ההמרה יתבצע או דרך שיפור המימוש הקיים ב C, או דרך החלפתו בספריה חיצונית מהירה יותר כגון "`ORJSON`" (שנציג בקרוב), מבלי לשנות את מבנה או כמות הנתונים (עם שינוי קטן בסוג הערך המוחזר מהפונקציה).

הערה: לא נעבור על המימוש בפייתון, מאחר והוא משמעותית איטי יותר מהמימוש ב C, ולכן לאנליזה שלנו נתעלם ממנו.

המימוש של `json` ב C, "מאיץ" את תהליך ההמרה של אובייקטים למחרוזות JSON. הוא מקבל את כל ההגדרות (כמו indents, sort_keys וכו') מהאובייקט (`PyEncoderObject`) שנוצר ב wrapper שקורא לו מהפייתון, ומבצע את ההמרה בפונקציות C. במהלך העבודה, הוא עובר תו-תו על הקלט (למשל, מחרוזות), ובודק האם התו ניתן להדפסה ישירה או שדורש escape (ובמקרה הצורך ממיר אותו לרצף מתאים כמו `uXXXX\\`).

למען הסבר פשוט, מצורף תרשים זרימה:



מאחר ואנחנו מחפשים דרך לבצע את הפעולה יותר מהיר מקוד C, נצטרך למצוא מימוש, שהוא יותר יעיל אלגוריתמית, או מממש את הפעולה בעזרת פיצ'רים חומרתיים.

הערה: שינוי המימוש ב C, ידרוש קימפול מחדש של ה interpreter, ולכן עם שיקול דעת זה, יהיה עדיף להשתמש בספריה מהירה יותר (עם api דומה), מאשר לכתוב קוד חדש בעצמנו ולקמפל את פייתון לגמרי מחדש.

לאחר חיפוש באינטרנט, מצאנו כי ישנה ספריה פופולרית הנקראת `ORJSON`, המממשת את אותה הפעולה, והיא כתובה בשפה מודרנית שהיא יחסית low level כמו C, שקוראים לה (Rust).

אחד היתרונות הבולטים של Rust על פני C הוא ביכולת של הקומפיילר שלה לבצע אופטימיזציות "under the hood", וביניהם וקטוריזציה אוטומטית של קוד באופן חכם ובטוח. נוסף לכך, הרבה מהטיפוסים של פייתון נתמכים בצורה ישירה בRust, וזה ממש מקל על המימוש ועל הסיבוכיות של הקוד.

בנוסף, קומפיילרים של C לרוב אינם מבצעים וקטוריזציה אוטומטית באגרסיביות, אלא הם חייבים "החזקת יד", בעזרת pragma's בשביל לסמן לקומפיילר שכאן מותר לווקטר. אמנם גם ב C ניתן לווקטור פעולות ידנית (למשל בעזרת הספרייה `immintrin.h`), אך מדובר בתהליך מייגע ומסורבל (עשינו זאת בפרוייקט הביניים) ובמיוחד בקובץ גדול (כמו מימוש ה C שהוא בן 2000 שורות).

לאור הנאמר לעיל והסיבוכים שיגרמו משינוי קוד ה C, נעשה שימוש בספריית ה `ORJSON` עבור ה encoder שלה.

הערה: נשים לב, כי הפונקציה שלה, לא מחזירה string, אלא bytes, ולכן נצטרך להפעיל עוד פונקציית decode, שתמיר את הפלט ל string. נראה עוד מעט שזה לא משפיע כל כך על הביצועים, וכי שינוי הספריה מוביל לשיפורים גולים בזמן הריצה ולכן ההשפעה של עוד שלב decode זניחה.

נוסיף את מקטע הקוד הנ"ל על מנת לבחור בספריה המתאימה כרצוננו:

```
if USE_ORJSON:
    import orjson
    def dumps(obj):
        return orjson.dumps(obj).decode('utf-8')
else:
    import json
    def dumps(obj):
        return json.dumps(obj)
```

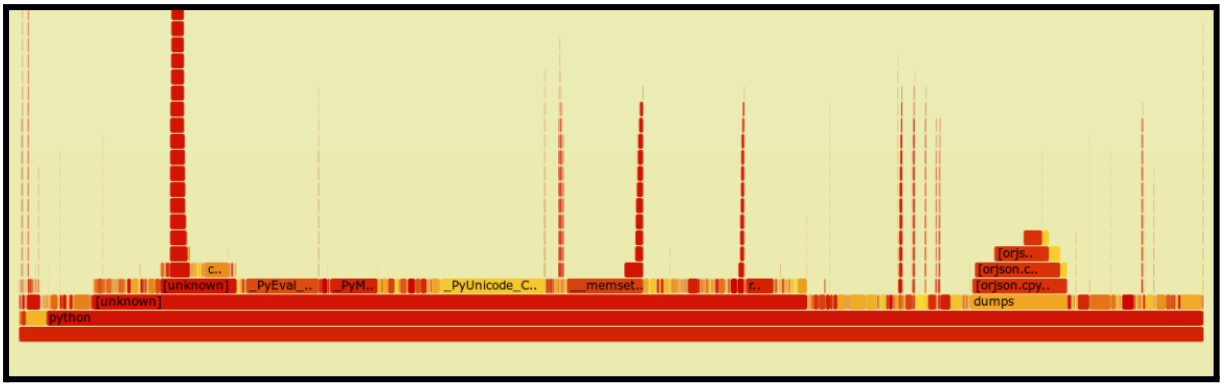
ונשנה את קריאת הפונקציה ל:

```
def bench_json_dumps(data):
    for obj, count_it in data:
        for _ in count_it:
            dumps(obj)
```

ובעזרת הדגל USE_ORJSON, נעשה import לספריה המתאימה.

הערה: שאר הקוד נשאר אותו דבר.

הנה ה flamegraph של האופטימיזציה:



ניתן לראות כי הגרף נראה זהה, למעט הגבעה לקראת הסוף, המציגה את הביצועים של ORJSON.

ועכשיו החלק המעניין, התוצאות:

עבור `json library`:

```
### json_dumps ###
Mean +- std dev: 62.7 ms +- 1.8 ms


 Performance counter stats for 'python3-dbg -m pyperformance run --bench json_dumps':

      89305144500      instructions              #    1.94  insn per cycle
      46070462407      cycles                    #    2.575 GHz
      23302712616      branches                  #    1.303 G/sec
        173333897      branch-misses             #    0.74% of all branches
          5436050      cache-misses
             3384      context-switches          #  189.148 /sec
           213625      page-faults               #   11.941 K/sec
         17890.71 msec task-clock                #    0.975 CPUs utilized

     18.341956008 seconds time elapsed

     17.240967000 seconds user
      0.700652000 seconds sys
```

עבור `ORJSON library`:

```
### json_dumps ###
Mean +- std dev: 10.8 ms +- 0.4 ms


 Performance counter stats for 'python3-dbg -m pyperformance run --bench json_dumps':

     120358911847      instructions              #    2.08  insn per cycle
      57799975143      cycles                    #    2.578 GHz
      32136331615      branches                  #    1.433 G/sec
        200095918      branch-misses             #    0.62% of all branches
          6197237      cache-misses
             3744      context-switches          #  166.995 /sec
           307639      page-faults               #   13.722 K/sec
         22419.78 msec task-clock                #    0.979 CPUs utilized

     22.903973937 seconds time elapsed

     21.603676000 seconds user
      0.872413000 seconds sys
```

והכל בטבלה אחת:

Metric	JSON Library	ORJSON Library
Execution Time [msec]	62.7 ± 1.8	10.8 ± 0.4
CPU Utilization	0.975	0.979
Context Switches	3,384	3,744
Page Faults	213,625	307,639
Cycles	46,070,462,407	57,799,975,143
Instructions	89,305,144,500	120,358,911,847
IPC (Instructions per Cycle)	1.94	2.08
Branches	23,302,712,616	32,136,331,615
Branch Misses	173,333,897 (0.74%)	200,095,918 (0.62%)

הערה: הזמנים הם ממוצע מעל 21 ריצות, ושאר הנתונים הם של הריצה המלאה של כל בנצ'מרק

ניתן לראות שיש speedup משמעותי של כמעט 6x, שה IPC משתפר, ושיש קצת פחות branch misses. בכל אופן, ניתן לראות שיפור גדול בזמן הבנצ'מרק (המדד העיקרי שלנו) ושיש ניצולת\יעילות יותר טובה.

הערה: זמן הריצה הכולל (כולל perf & python3-dbg) עלה ב 2-3 שניות, ואנחנו לא בטוחים למה, עם זאת תוצאות הבנצ'מרק עצמו יותר טובות, שזה מה שחיפשנו לשפר.

הצעת חומרה:

כפי שצוין קודם, השיפור המתקבל בעת שימוש בספריה ORJSON נובע במידה רבה מהסתמכות על אופטימיזציות ה SIMD, המובנת ברוב המעבדים המודרניים, ובעוד ייעולים שהקומפיילר שלה עושה.

לכן, קשה להציע שינוי חומרתי נוסף שיהיה מעשי או כדאי. פעולת קידוד/פענוח JSON היא אמנם נפוצה, אך אינה מצדיקה פיתוח של מאיץ ייעודי משל עצמה. עם זאת, בכל זאת נציע תוספת חומרתית על מנת להאיץ את הפעולה.

לאור כך שהמקביליות לא תרמה speed up גדול (לא יותר מסדר גודל 1), נרצה להאיץ חלק אחר ב compute. אנו נציע תוספות ל ISA, כלומר סט של פקודות אשר יאיץ לנו את צוואר הבקבוק (escape ים של סטרינגים, המרות int/float). בכך האלגוריתמים עם המלא case-ים וה branch-ים יוחלפו בפקודות מכונה שממומשות בחומרה.

סט הפקודות:

פולט מחרוזת עם ולידציה + escapes	$JSTR$
פולט מספר שלם בעשרוני ללא חלוקות איטיות	$JINT$
פולט float בקידוד מתאים	$JFLT$
פולט ליטרלים קבועים (null/true/false)	$JLIT$
פותח/סוגר {} או [], עם ניהול פסיקים אוטומטי	$JPUSH/JPOP$
בדיקת מקום בבאפר מראש, מונעת overflow	$JLen$

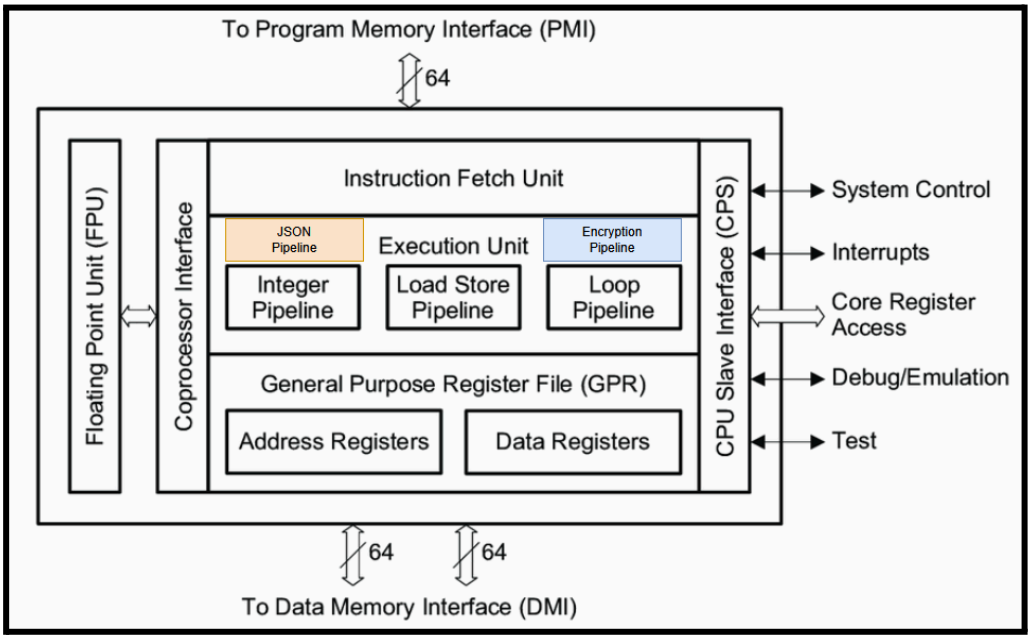
לדוגמה עבור ההמרה של האובייקט $\{"a": 1, "b": true\}$, קוד האסמבלי יהיה:

```
JPUSH.OBJ                ; {
JSTR   &"a", 1           ; "a"
JINT   1                 ; 1
JSTR   &"b", 1           ; "b"
JLIT   TRUE              ; true
JPOP                     ; }
```

שינויי תוכנה:

- תמיכה ברמת הקומפיילר ל feature.
- הוספת אינטרינזיקות (intrinsics) או ספריה לפקודות.
- דגלים וכו' יוכלו להיות מומרים ברמת הספריה כפרמטרים לפקודות.

דיאגרמת בלוקים:



הערכת ביצועים:

- ראינו שלמרות ש ORJSON האיץ את החישובים, קיבלנו ספידאפ רק של 6x.
- לדעתינו בעזרת פקודות החומרה, נוכל להאיץ כל המרה ב 1-3 סדרי גודל ($\mu sec \Leftarrow msec$).

כמובן, הופסת סט פקודות שכזה מסבך את הארכיטקטורה של המעבד, ולכן זה כנראה לא הצעה אקטואלית, במיוחד לא למטלה הספציפית הזאת, שלמרות שהיא נפוצה, לא מצדיקה הוספת רכיב חומרה שיסבך את את הארכיטקטורה עד כדי כך כאשר ישנה פשרה קיימת (מעבר לספריית C).

בנצ'מרק Crypto PyAES - 2

נספח: ה benchmark המקורי עם קצת הערות שלנו

```python
#!/usr/bin/env python


# A version to make edits and performance optimizations.

# This is AES-128 in CTR mode, implemented in pure Python.
# It uses the pyaes library, which is a standalone AES implementation
# in pure Python. It is not a wrapper around any C code.
import pyperf

import pyaes


# 23,000 bytes in size
CLEARTEXT = b"This is a test. What could possibly go wrong? " * 500

# 128-bit key (16 bytes)
KEY = b'\xa1\xf6%\x8c\x87}_\xcd\x89dHE8\xbf\xc9,'


def bench_pyaes(loops):
    range_it = range(loops)
    t0 = pyperf.perf_counter()

    # benchmark workload
    for loops in range_it:
        aes = pyaes.AESModeOfOperationCTR(KEY)
        ciphertext = aes.encrypt(CLEARTEXT)

        # need to reset IV for decryption
        aes = pyaes.AESModeOfOperationCTR(KEY)
        plaintext = aes.decrypt(ciphertext)

        # explicitly destroy the pyaes object
        aes = None

    dt = pyperf.perf_counter() - t0
    if plaintext != CLEARTEXT:
        raise Exception("decrypt error!")

    return dt


# some pyperf boilerplate, ignore.
if __name__ == "__main__":
    runner = pyperf.Runner()
    runner.metadata['description'] = ("Pure-Python Implementation "
                                      "of the AES block-cipher")
    runner.bench_time_func('crypto_pyaes', bench_pyaes)
```

בעיקרון, הבנצ'מרק מצפין טקסט, ואז מפענח אותו loops פעמים. ובסוף עושה בדיקת נכונות לאיטרציה האחרונה כבדיקת שפיות.

בעמוד הבא נסביר כיצד AES עובד, ולאחר מכן נסביר את הגישה שלקחנו עבור האופטימיזציה.

הערה, לאור טבע האלגוריתם (שנסביר בעמוד הבא), קל להבין כי ה bottleneck בבנצ'מרק הזה, יהיה בגלל ה compute, מאחר ועיקר העבודה כלולה בטרנספורמציות של ביטים.

אז כיצד AES עובד?

ראשית, לוקחים plaintext בגודל של 128 ביטים, או לחילופים 16 בתים, ומסדרים אותו במטריצה של 4X4, אשר נקראת המצב של האלגוריתם. עושים סידור נוסף גם למפתח.

נתחיל בתאור כל הפעולות שהאלגוריתם מבצע:

`AddRoundKey:` ביצוע פעולת XOR בין הביטים של המצב לבין הביטים של המפתח.

`KeyExpansion:` הרחבה של המפתח המקורי ל - 11 מפתחות שונים אשר כל אחד מהם תלוי במפתח המקורי. ישנן פעולות קבועות שהאלגוריתם משתמש בהם על מנת לבצע את ההרחבה הזו. חלק מהפעולות כוללות הוספת ערך קבוע, הזזת בתים, החלפת בתים וכדומה.

`SubBytes:` כל בית במצב מוחלף בבית אחר. ההחלפה מתבצעת על ידי האלגוריתם באופן מדויק. ההחלפה אינה לינארית פשוטה, אלא ביטוי מתמטי מורכב שאינו לינארי. ההחלפה ידועה לציבור וניתן למצוא אותה באינטרנט.

`MixColumns:` ערבוב הערכים בכל עמודה. עבור כל עמודה בנפרד, מתבצע ביטוי מתמטי שמשלב את ארבעת הבתים על מנת לקבל בתים חדשים. החישוב מתבצע מעל שדה סופי בגודל 256, על מנת שעבור כל ביטוי נקבל ערך שגודלו בית אחד.

`ShiftRows:` ביצוע הזזה של הערכים בכל שורה במספר צעדים שונה. השורה הראשונה לא זזה. השורה השניה זזה צעד אחד, השלישית שני צעדים, והרביעית שלושה צעדים.

כעת נתאר את האלגוריתם השלם:

תחילה מבצעים `KeyExpansion`, ואז `AddRoundKey` עבור המפתח הראשון. לאחר מכן מתחילים לבצע סבבים כאשר בכל אחד מהם,

1. `SubBytes`
2. `ShiftRows`
3. `MixColumns`
4. `AddRoundKey` (לפי מספר הסבב)

הצעדים האלה מתבצעים אחד אחרי השני במשך 11 סבבים (עבור AES-128) עד שמקבלים את התוצאה הסופית של ההצפנה. יש לציין שבסבב האחרון לא מתבצע `AddRoundKey`.

ביצוע ה CTR:

ברוב המקרים, ה plaintext שאותו צריך להצפין יהיה ארוך יותר מ 128 ביטים. במצבים כאלה משתמשים במצב פעולה כמו CTR.

תחילה לוקחים nonce באורך 96 ביטים. ה nonce לא חייב להיות סודי, אבל הוא חייב להיות ייחודי לכל הצפנה. בנוסף מגדירים counter באורך 32 ביטים, עבור הבלוק הראשון של ה plaintext הערך יהיה 0, עבור הבלוק השני 1, וכן הלאה.

עבור כל בלוק של ה plaintext מבצעים שרשור בין ה nonce ל counter. ה nonce ממוקם בחלק העליון, וה counter בחלק התחתון. כך נוצר $CounterBlock_i$, כאשר i מציין את מספר הבלוק.

על כל CounterBlock מפעילים את אלגוריתם AES עם המפתח הנתון. לאחר מכן מבצעים XOR בין התוצאה שהתקבלה לבין בלוק ה i של ה plaintext. התוצאה היא בלוק ההצפנה המתאים (ciphertext block). כך ממשיכים עבור כל הבלוקים עד לסיום ההצפנה.

במקרה שבו הבלוק האחרון קצר מ־128 ביטים, ממשיכים כרגיל, יוצרים CounterBlock נוסף, מפעילים AES, ומבצעים XOR. ההבדל היחיד הוא שה XOR מתבצע מול הבלוק החלקי בלבד, ולכן אין צורך ב padding.

ניתן לראות, שהבנצ'מרק לא מסובך, כל הסיבוכיות באלגוריתם מגיעה מהפעולה של encrypt/decrypt.
הספריה pyaes, כתובה כולה בפייתון טהור, ולא בצורה מקבילית. ולכן אפילו שימוש בספריית C שעושה בדיוק אותו דבר, תהיה מהירה יותר. עם זאת, נוכל לקבל האצה משמעוית יותר בכמה סדרי גודל, בעזרת כלי חומרתי שממומש במעבד של ה VM שלנו.

```
root@ubuntu ~/SOFTWARE-HARDWARE-CODESIGN # grep -E 'aes|avx2|vaes' /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm con
stant_tsc arch_perfmon rep_good nopl xtopology cpuid tsc_known_freq pni pclmulqdq vmx ssse3 fma cx16 pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_de
adline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpri
ority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx xsaveopt arat umip md_clear arch_capabilities
root@ubuntu ~/SOFTWARE-HARDWARE-CODESIGN #
```

פקודה המציגה את ה features של המעבד. ניתן לראות שהוא תומך ב AVX2 ו ב AES

AES-NI זה סט פקודות x86, התומך בפעולות של ה סטנדרט של ה AES וממש רצות כפקודות מכונה. לכן, בעזרתן, נוכל לקבל האצה של כמה סדרי גודל ב compute שלנו (שזה ה bottleneck). לאחר חיפוש באינטרנט, מצאנו ספרייה שמספקת API נוח לפייתון, שבסוף מריצה כלי שקרא OpenSSL, שאוטומטית מזהה איזו פקודות מכונה יש כדי להאיץ את ההצפנה, ובעצם עושה שימוש ב AES-NI כדי לבצע את מה שאנחנו רוצים (encrypt/decrypt). לספריה זו קוראים "`cryptography`".

נוסיף באופן דומה לבנצ'מרק הקודם, את מקטע הקוד החדש, שהוא זהה כולו לבנצ'מרק המקורי, למעט קריאות הפונקציה של ההצפנה והפענוח.

```python
if USE_CRYPTOGRAPHY_LIB:
    from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

    def bench_aes(loops):
        range_it = range(loops)
        t0 = pyperf.perf_counter()

        for _ in range_it:
            # Encrypt
            cipher = Cipher(algorithms.AES(KEY), modes.CTR(IV))
            enc = cipher.encryptor()
            ciphertext = enc.update(CLEARTEXT) + enc.finalize()

            # Decrypt
            cipher = Cipher(algorithms.AES(KEY), modes.CTR(IV))
            dec = cipher.decryptor()
            plaintext = dec.update(ciphertext) + dec.finalize()

        dt = pyperf.perf_counter() - t0
        if plaintext != CLEARTEXT:
            raise Exception("decrypt error!")
        return dt
```

הקוד שמשתמש בספריה החדשה

הקוד הקודם לצורך השוואה:

```python
else:
    import pyaes

    def bench_aes(loops):
        range_it = range(loops)
        t0 = pyperf.perf_counter()

        for _ in range_it:
            aes = pyaes.AESModeOfOperationCTR(KEY)
            ciphertext = aes.encrypt(CLEARTEXT)

            aes = pyaes.AESModeOfOperationCTR(KEY)
            plaintext = aes.decrypt(ciphertext)

        dt = pyperf.perf_counter() - t0
        if plaintext != CLEARTEXT:
            raise Exception("decrypt error!")
        return dt
```

ניתן לראות שהקוד זהה בלוגיקה שלו בין השניים, למעט השימוש ב API החדש

וכעת לתוצאות:

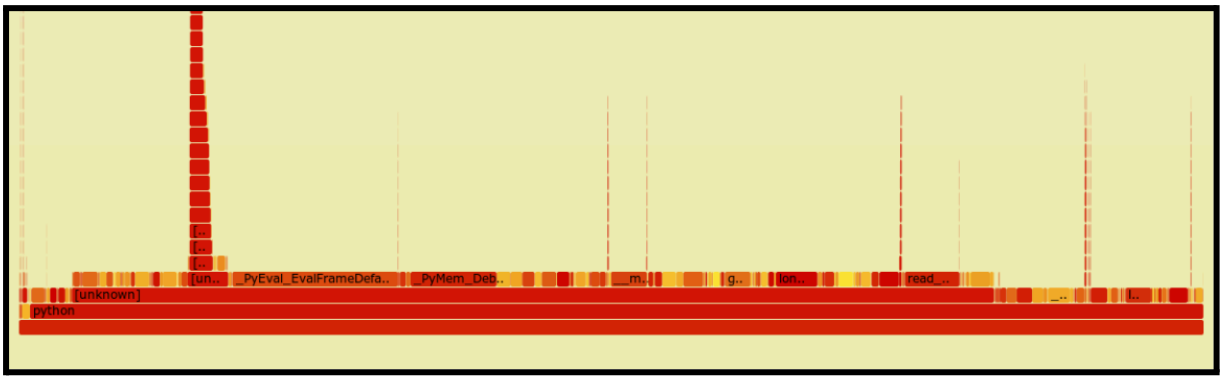
```
### crypto_pyaes ###
Mean +- std dev: 609 ms +- 13 ms


 Performance counter stats for 'python3-dbg -m pyperformance run --bench crypto_pyaes':

      150210005089      cycles                    #    2.583 GHz
      338286117899      instructions              #    2.25  insn per cycle
       85967708131      branches                  #    1.478 G/sec
         354441206      branch-misses             #    0.41% of all branches
          11623567      cache-misses
              5195      context-switches          #   89.319 /sec
                 0      cpu-migrations            #    0.000 /sec
          58162.54 msec task-clock                #    0.982 CPUs utilized

      59.249884442 seconds time elapsed

      57.628063000 seconds user
       0.630373000 seconds sys
```

perf stats עבור הגרסה המקורית

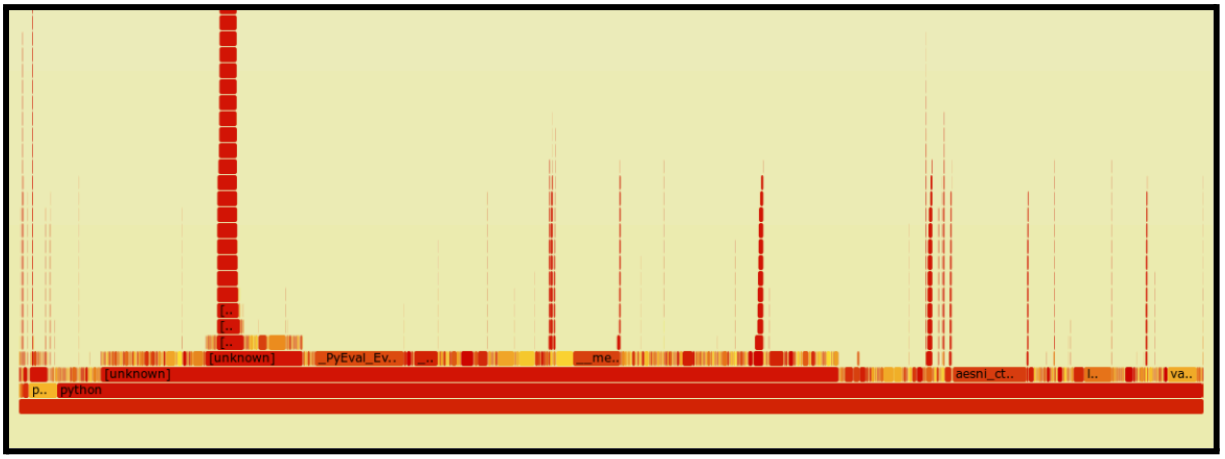
```
### crypto_pyaes ###
Mean +- std dev: 97.4 us +- 2.6 us


 Performance counter stats for 'python3-dbg -m pyperformance run --bench crypto_pyaes':

       64481268793      cycles                    #    2.572 GHz
       93117206067      instructions              #    1.44  insn per cycle
       20562099962      branches                  #  820.297 M/sec
         246238835      branch-misses             #    1.20% of all branches
          12059767      cache-misses
              3816      context-switches          #  152.234 /sec
                 0      cpu-migrations            #    0.000 /sec
          25066.66 msec task-clock                #    0.978 CPUs utilized

      25.619293635 seconds time elapsed

      24.344810000 seconds user
       0.764770000 seconds sys
```

perf stats עבור הגרסה החדשה

תוצאות ה flamegraph של שני הגרסאות:



flamegraph עבור הגרסה המקורית



flamegraph עבור הגרסה החדשה

ניתן לראות שהגרפים שונים בכמות המשבצות ברמת האחרונה, כעת יש לנו המון משבצות קצרות, וניתן לראות סקשן של aesni שרץ עבור הפקודות החדשות שלנו.

תוצאות ה perf stat בטבלה להשוואה:

Metric	PyAES Library	Cryptography API
Execution Time	609±13 [ms]	97.4±2.6 [µs]
CPU Utilization	0.982	0.978
Context Switches	5,195	3816
Cycles	150,210,005,089	64,481,268,793
Instructions	338,286,117,899	93,117,206,067
IPC (Instructions per Cycle)	2.25	1.44
Branches	85,967,708,131	20,562,099,962
Branch Misses	354,441,206 (0.41%)	246,238,835 (1.20%)

הסבר תוצאות:

ניתן לראות כי קיבלנו האצה בכמה סדרי גודל של הבנצ'מרק שלנו. ניתן לראות כי כמות הסייקלים והפקודות, קטנה משמעותית, למרות שזה לא מייצג טוב את השיפור במהירות, אלא יותר מייצג את ה overhead של python-dbg ו perf מאשר את של התוכנית שלנו. ניתן לראות שלמרות שיש יותר branch misses, ופחות IPC, אנחנו בכל אופן מסיימים הרבה יותר מהר.

הצעת חומרה:

בזכות ה AES-NI Instructions, השגנו latency נמוך מאוד. אבל אם נתחשב במידע שלנו שהוא היה רק 23KB, זה לא הצפנה מאוד כבדה. עבור workload יותר גדול, למשל אולי כמו בשרת שצריך לשלוח המון חבילות מוצפנות, נרצה רכיב מהיר שיש לו יותר throughput.

אפשר אולי להציע רכיב חומרתי\FPGA, שיקבל כניסות כמו Key, IV ו data stream של ה plaintext, ושכל שלב בהצפנה (למשל `MixColumns`, `SubBytes`, `AddRoundKey` וכו') יתבצע ב pipeline, ובכך לאחר כל clock cycle נקבל בלוק מוכן.

נפרט כעת על המימוש החומרתי:

המאיץ יתמקד בהצפנת AES במצב CTR. המימוש מבוסס על pipeline מלא של שלבי האלגוריתם (SubBytes, ShiftRows, MixColumns, AddRoundKey), כך שלאחר מילוי הצינור מתקבל בלוק מוצפן חדש בכל מחזור שעון. כך נקבל throughput גבוה, שמתאים במיוחד לעיבוד רציף של כמויות גדולות של data.

הממשק של ה pipeline יהיה כזה:

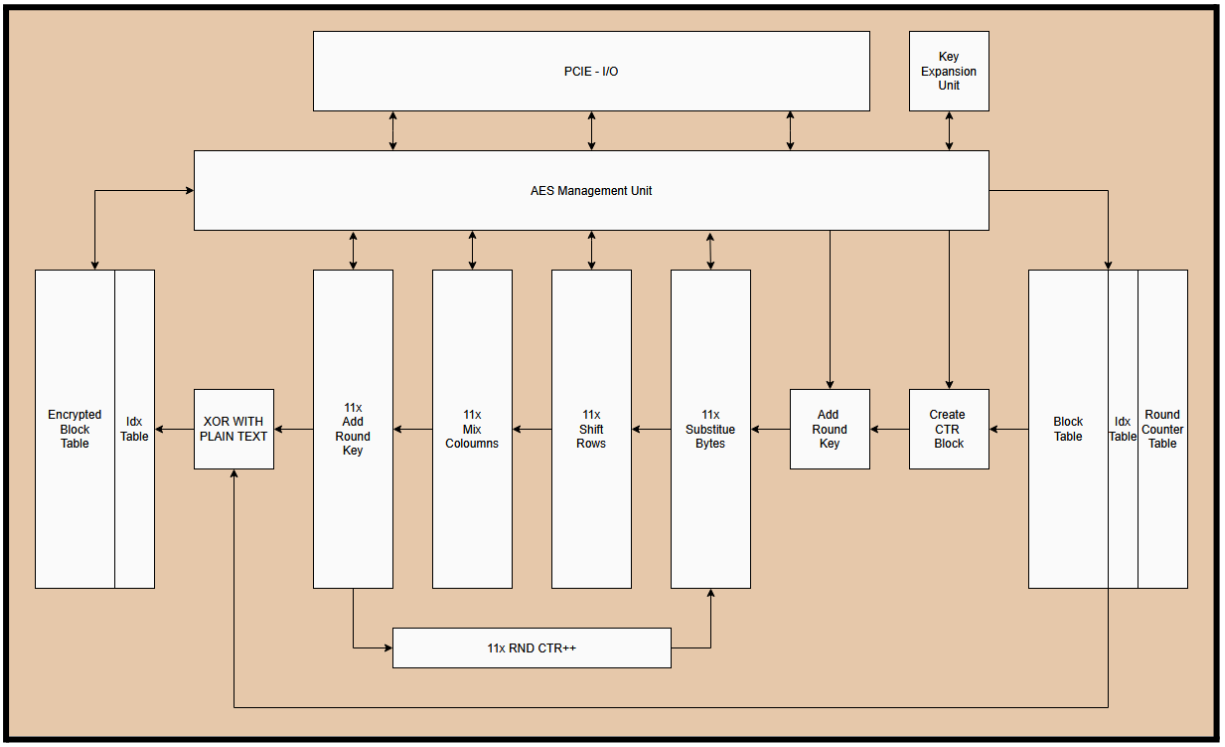
כניסות:

- מפתח הצפנה (128 bit)
- וקטור אתחול ומונה (IV + Counter)
- זרם נתוני plaintext

יציאות:

- זרם נתוני ciphertext
- סטטוס בסיסי (סיום/שגיאה)

דיאגרמת בלוקים פשוטה של הלוגיקה (מימוש עבור 128bit):



ממשק דרך PCIe:
המאיץ יעבוד ככרטיס סטנדרטי מעל PCIe. מערכת ההפעלה והאפליקציה יתקשרו איתו בעזרת הדרייברים, כאשר נתונים נשלחים אליו והוא מחזיר את התוצאה המוצפנת. מבחינת התוכנה, הדבר יתבטא בקריאה פשוטה לפונקציית דרייבר פשוטה כמו $aes\_ctr\_encrypt(in,\ out,\ key,\ iv)$, בעוד שהביצוע יקרה על הכרטיס.

שיערוך ביצועים:

Throughput תיאורטי:

$clk\ freq \times 128\ bit$ (נזניח מילוי pipe).

אז לדוגמה:

$$300MHz \times 128\ Bits \times 1\ lane \approx 4\ Gbps$$

(כמובן שאנחנו גם תלויים במגבלות ה BUS/Memory).

Latency תיאורטי:

קשה לתת הערכה מדויקת ל Latency, שכן הוא מושפע לא רק מהצפנת הבלוקים עצמם אלא בעיקר מקצב התקשורת על גבי ה Bus. עבור workloads גדולים, עיכוב התקשורת זניח ביחס לכמות הנתונים, ולכן המאיץ מספק יתרון ברור ב throughput. לעומת זאת, עבור workloads קטנים, ה overhead של העברת המידע למאיץ דרך ה Bus משמעותי יותר מהזמן של ההצפנה עצמה.

במקרים כאלה עדיף להישאר בצד ה CPU, שכן בזכות AES-NI ההצפנה מתבצעת במהירות גבוהה במיוחד וב Latency נמוך מאוד.

(למשל בבנצ'מרק שלנו, עם data בגודל 23KB בלבד התקבל זמן של בערך 100µs, תוצאה שאולי לא מצדיקה העברה למאיץ חומרה, בגלל עלות התקשורת שמעל ה Bus)

הסבר בנוגע לשימוש ב AI בפרויקט:

נעזרנו בכלי AI, ספציפית ChatGPT ו Github CoPilot עבור אוטומציה של סקריפטים לריצה של ה benchmarks לנוחיותכם.

נוסף לכך נעזרנו בכלי AI על מנת לבצע חיפוש יותר יעיל באינטרנט והגעה למקורות. בנימה זאת, המקורות שהשתמשנו בהם מצורפים בעמוד הבא.

הסבר בנוגע לשימוש בסקריפטים ובאוטומציה:

הוראות ההפעלה והשימוש בסקריפטים להרצת הבנצ'מרקים נמצאים ב "README.md" של ה Repository בGithub. כפי שצוין בדף השער, כל קבצי הפרוייקט נמצאים בקישור:

https://github.com/AmZu1212/SOFTWARE-HARDWARE-CODESIGN

ניתן לבצע pull ולעשות שימוש בקבצים.

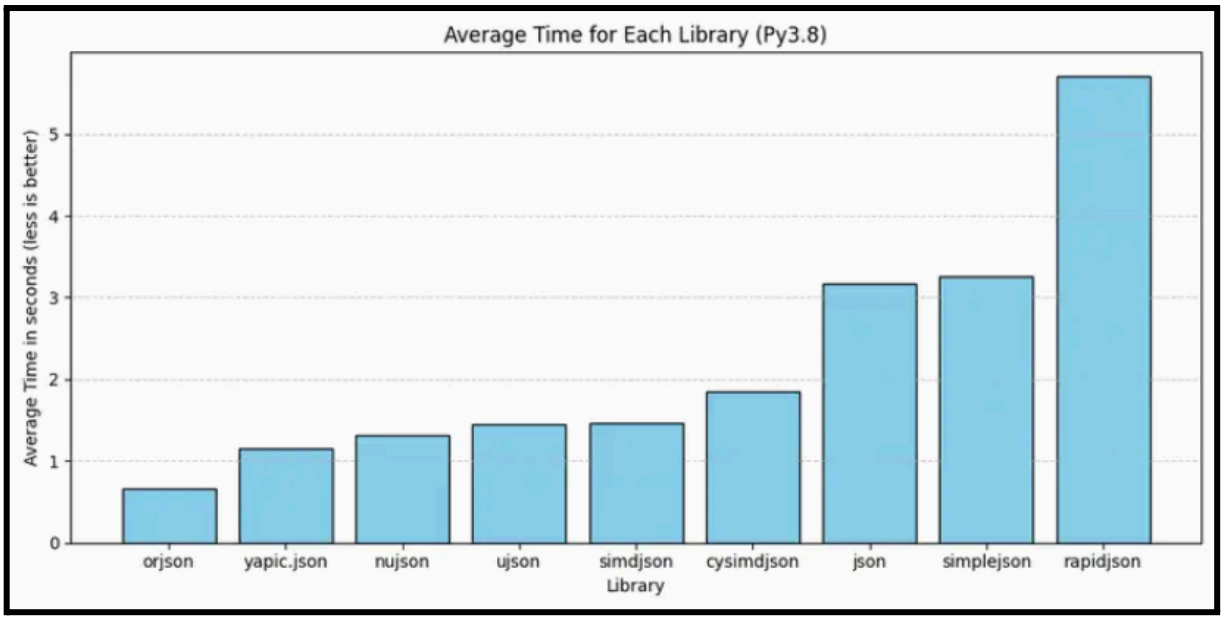
מקורות:

CPython: source code

https://github.com/python/cpython/blob/main/Lib/json/encoder.py
https://github.com/python/cpython/blob/main/Modules/_json.c

Finding the fastest Python JSON library on all Python versions

https://catnotfoundnear.github.io/finding-the-fastest-python-json-library-on-all-python-versions-8-compared.html



ORJSON: source code

https://github.com/ijl/orjson

AES - How to Design Secure Encryption

https://www.youtube.com/watch?v=C4ATDMIz5wc

AES Instruction Set

https://en.wikipedia.org/wiki/AES_instruction_set