Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра Обчислювальної Техніки

Лабораторна робота №2

з дисципліни "Проектування складних систем"

Тема: "Дослідження технології апаратного пропуску замків

(Intel’s Hardware Lock Elision)"

Виконав:

студент групи ІП-93

Домінський В.О.

Перевірив:

Долголенко О. М.

Київ 2023

Зміст

Мета:

Для виконання цієї роботи потрібен доступ до мікропроцесора Intel, що підтримує технологію TSX-NI. Створіть потоки з використанням інтерфейсу Callable з пакету java.util.concurrent (синхронізація потоків за допомогою інтерфейсу Lock із пакета java.util.concurrent.lock). Дослідіть швидкість виконання Вашої паралельної програми на цьому мікропроцесорі. Після цього відключіть Intel® TSX-NI за допомогою наступного налагоджування реєстру (для Windows 10): reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Kernel" /v DisableTsx /t REG_DWORD /d 1 /f. Перегрузіть комп'ютер, щоб ці зміни мали ефект та знову дослідіть швидкість виконання варіантів Вашої паралельної програми. Побудуйте порівняльні графіки й зробіть висновки по роботі. Для наступного включення технології Intel® TSX-NI, анологічно виконайте: reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Kernel" /v DisableTsx /t REG_DWORD /d 0 /f.

Вихідні дані:

Варіант – 12

Функції:

- A=B*MC+D*MZ+E*MM
- MG=min(D+E)*MM*MT-MZ*ME

Процесор з технологією Intel® TSX-NI - Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz

Хід роботи

За основу для даної лабораторної роботи візьмемо минулу, але зробимо такі зміни:

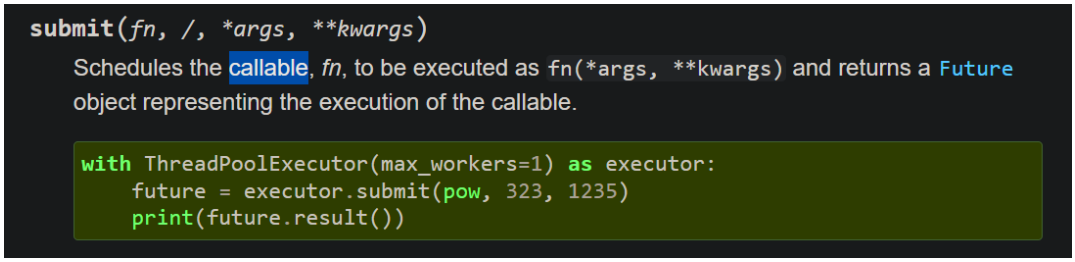
1. Створимо потоки з інтерфейсом callable. У java це java.util.concurrent, а в python - concurrent.futures. Додатково використаємо ThreadPoolExecutor, котрий може об’єднувати потоки в групи:

```python
def run_first_threads_second_run(self, i):
    self.ys_2_1.append(len(vector_b[i]))

    mainThread = LabFirstThread()

    self.start_time(i + 1, 1)

    with ThreadPoolExecutor(max_workers=4) as executor:
        future1 = executor.submit(lambda: mainThread.first_thread(vector_b[i], matrix_mc[i]))
        future2 = executor.submit(lambda: mainThread.second_thread(vector_d[i], matrix_mz[i]))
        future3 = executor.submit(lambda: mainThread.third_thread(vector_e[i], matrix_mm[i]))

        future1.result()
        future2.result()
        future3.result()

        future4 = executor.submit(lambda: mainThread.fourth_thread())

        future4.result()

        executor.shutdown()

    self.end_time()

    self.xs_2_1.append(self.actual_time)
    self.ys_2_2.append(len(vector_b[i]))

  def run_second_threads_second_run(self, i):
        mainThread = LabSecondThread()

        self.start_time(i + 1, 2)

        with ThreadPoolExecutor(max_workers=4) as executor:
            future1 = executor.submit(lambda: mainThread.first_thread(vector_d[i], vector_e[i]))
            future2 = executor.submit(lambda: mainThread.second_thread(matrix_mm[i], matrix_mt[i]))
            future3 = executor.submit(lambda: mainThread.third_thread(matrix_mz[i], matrix_me[i]))

            future1.result()
            future2.result()
            future3.result()

            future4 = executor.submit(lambda: mainThread.fourth_thread())

            future4.result()

            executor.shutdown()

        self.end_time()
        self.xs_2_2.append(self.actual_time)
```

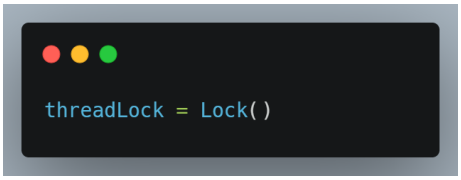
У даному випадку Ми використовуємо ThreadPoolExecutor, де робимо 4 future, кожен з яких представляє кінцевий результат асинхронної роботи та відповідає за власну частину виразу.

Як видно з коду, то спочатку Ми запускаємо перші 3, а вже потім – останній – 4-ий future, котрий фіналізує Наш результат, так як він залежить від усіх інших.

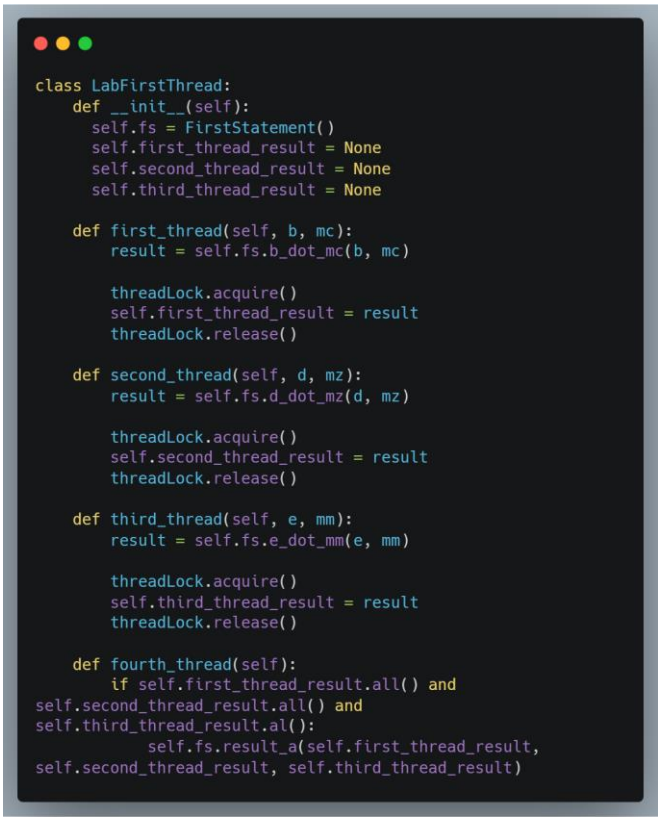
За допомогою методу submit Ми плануємо виклик future передаючи лямбда вирази методів (які є callable) для обчислення функції відповідно до варіанту роботи.



2. Зробити записи до глобальних змінних синхронними. Для цього використаємо Lock:



Та будемо працювати з ним при кожному записі:

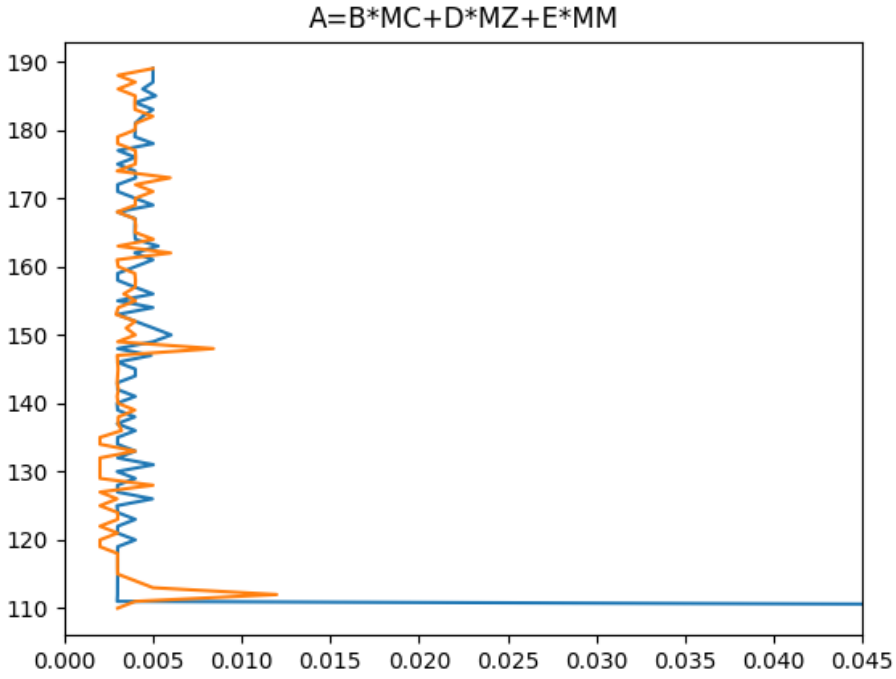


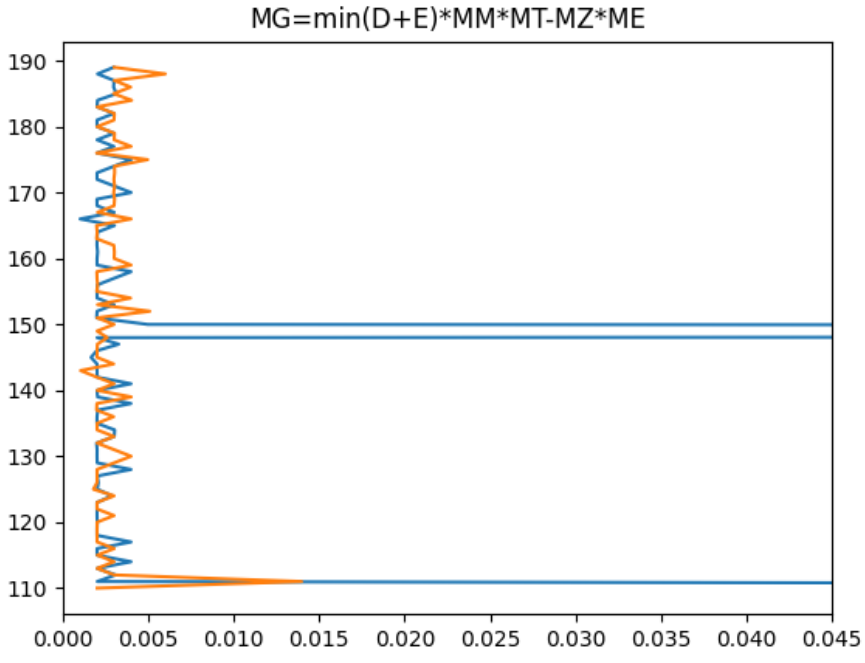
```
class LabSecondThread:
    def __init__(self):
      self.ss = SecondStatement()
      self.first_thread_result = None
      self.second_thread_result = None
      self.third_thread_result = None

    def first_thread(self, d, e):
        result = self.ss.min_d_add_e(d, e)

        threadLock.acquire()
        self.first_thread_result = result
        threadLock.release()

    def second_thread(self, mm, mt):
        result = self.ss.mm_dot_mt(mm, mt)

        threadLock.acquire()
        self.second_thread_result = result
        threadLock.release()

    def third_thread(self, mz, me):
        result = self.ss.mz_dot_me(mz, me)

        threadLock.acquire()
        self.third_thread_result  = result
        threadLock.release()

    def fourth_thread(self):
        if self.first_thread_result.all() and
self.second_thread_result.all() and
self.third_thread_result.all():
            self.ss.result_mg(self.first_thread_result,
self.second_thread_result, self.third_thread_result)
```

Метод acquire потрібен для закриття статусу Нашого локу, а release – навпаки – для відкриття. Таким чином робота зі змінними не буде відбуватись одночасно.

Також тепер результат обчислень не записується відразу до публічного поля, а попадає спочатку до локальної змінної. Зроблено це задля того, аби numpy міг нормально працювати, адже при лоці функцій, що використовують дану бібліотеку, обрахування виразу займає вкрай багато часу.

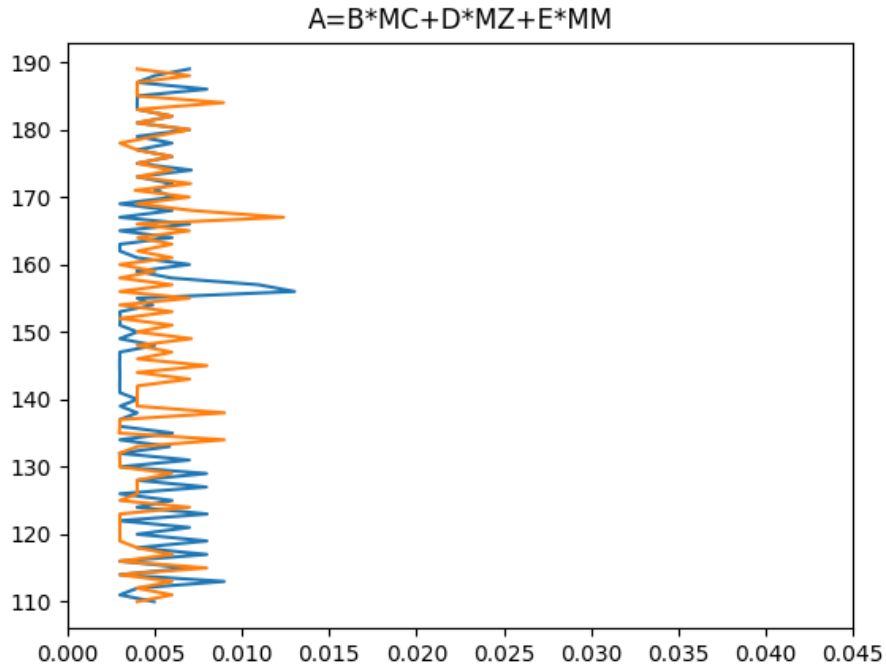
Тепер давайте запустимо цей код на персональному комп’ютері, де є процесор з підтримкою технології Intel® TSX-NI, але поки що активувати її не будемо:

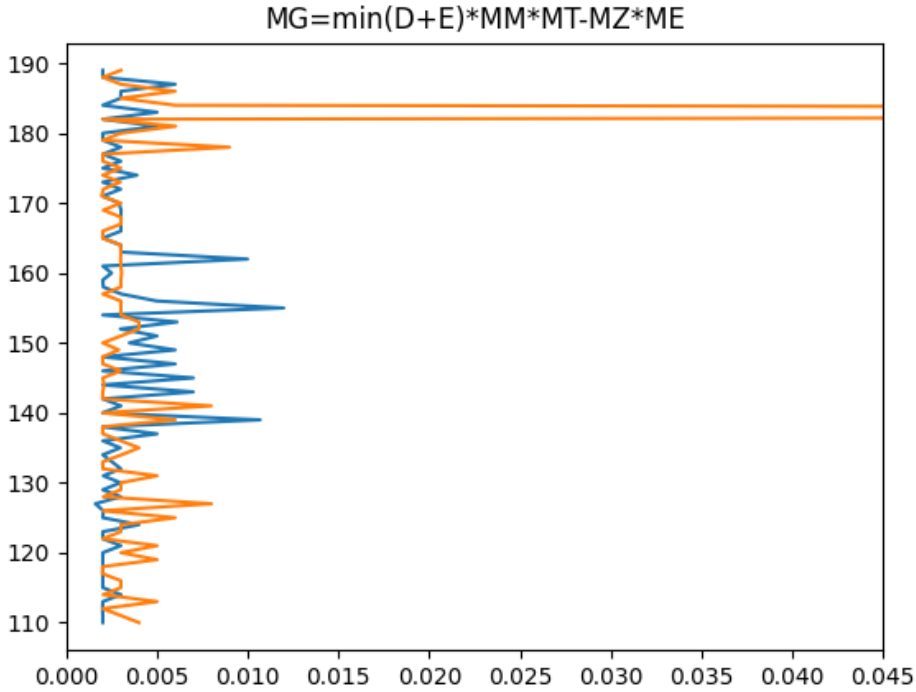




Тобто значення тримається в районі 0.0025 - 0.0037 і, в основному, має різницю між середніми мінімальним та максимальним значенням приблизно в 0.001

А зараз запустимо Лабораторну роботу з увімкненою технологією Intel® TSX-NI:





З графіків можна дійти до висновку, що коливання значень стало значно більшим, ніж було: з 0.0034 до 0.007

Висновок:

Створивши новий варіант лабораторної роботи Я думав, що при увімкненій технології Intel® TSX-NI можна значно збільшити продуктивність Мого коду, проте Я отримав зворотній результат. Спочатку Мені вважалося, що це щось пішло не так саме у Моїй роботі, але запитавши в знайомих почув той самий висновок.

Сама технологія має на увазі надбудову над системою роботи з кешем процесора, що оптимізує середовище виконання багатопотокових додатків, але тільки в тому випадку, якщо ці додатки використовують програмні інтерфейси TSX-NI.

Тому на Мою думку є кілька потенційних причин чому так сталося:

1. Це прискорення буде використовуватись лише у випадку використання програмного інтерфейсу TSX-NI

2. Існує певна недоробка з боку розробників, так як дана фіча, починаючи з 2014, поступово вимикалася на програмному рівні на багатьох процесорах і є шанс, що зараз, у 2023 році, ця технологія має недоліки, які і зменшують швидкість обчислень

У ході виконання роботи Я глибше познайомився з багатопоточністю у python, а саме з Lock (блокує певний фрагмент коду, доки робота з ним не закінчиться), ThreadPoolExecutor (об’єднує кілька потоків у групи) та future’ами (кінцевий результат асинхронної роботи), котрі використовуються передостаннім.

Посилання:

1. Робота на Google Colab – посилання
2. Робота на GitHub (разом з результатами роботи коду при увімкненій та вимкненій технології TSX-NI) – посилання