УДК 519.688:004

ОБЗОР JIT-КОМПИЛЯТОРА NUMBA КАК ИНСТРУМЕНТА ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА GPU

Лавринов М.И.

ФГБОУ ВО «Иркутский государственный университет», Иркутск, e-mail: Lm-ne@yandex.ru

Представлен обзор свободного программного обеспечения JIT-компилятора Numba для языка программирования Python от компании Anaconda, Inc на примере вычисления автокорреляционной функции сигнала заданной длины в дискретном времени для поиска оптимального для каждой цепочки. Сигнал в данной задаче принимает значения -1 или 1 и имеет длину от 1 до 35 тактов. Оптимальным в данной задаче считается сигнал с наибольшим отношением амплитуды основного лепестка автокорреляционной функции к боковому лепестку. Для нахождения оптимального сигнала необходимо вычислить автокорреляционную функцию для всех вариантов сигналов заданной длины, в связи с этим сложность задачи возрастает с увеличением длины цепочки и составляет $O(2^n)$. Таким образом, поиск оптимального сигнала в рамках данной задачи длиной в 35 тактов при простейшем последовательном вычислении на языке Python без использования дополнительных библиотек занимает до 39 суток. При использовании языка Python с JIT-компилятором Numba и дополнительного модуля numba.cuda для вычислений на GPU время поиска оптимального сигнала для цепочек той же длины можно сократить до 5 ч.

Ключевые слова: параллельное программирование, Python, Numba, поиск оптимального сигнала, автокорреляционная функция

AN OVERVIEW OF THE NUMBA JIT COMPILER AS A GPU PARALLEL COMPUTING TOOL

Lavrinov M.I.

Irkutsk State University, Irkutsk, e-mail: Lm-ne@yandex.ru

Presenting an overview of the free software Numba JIT compiler for the Python programming language from Anaconda, Inc. based on the value of the autocorrelation function of a signal given in discrete time to find a specific per-string. The signal in this task takes values from -1 or 1 and has a value from 1 to 35 cycles. The optimal signal in this problem is considered to be the signal with the maximum deviation of the estimate of the severity of the lobe of the autocorrelation function to the side lobe. To find a common signal, the required autocorrelation characteristic is calculated for all variants of signals of a given load; in connection with this complexity, it increases with the detection of a chain and is $O(2^n)$. Thus, the search for a wide range of signals within this long task of 35 cycles with a simple sequence of calculations in the Python language without using the extended library takes up to 39 days. When building the Python language with the Numba JIT compiler and the additional numba.cuda module for GPU computing, the search time for a measured signal for chains of the same frequency can reach 5 hours.

Keywords: parallel programming, Python, Numba, optimal signal search, autocorrelation function

В работе рассматриваются сигналы, которые принимают значения -1 или 1 и имеют длину от 1 до 35 тактов. Оптимальным будет считаться сигнал с наибольшим отношением амплитуды основного лепестка автокорреляционной функции к боковому лепестку. Для нахождения оптимального сигнала необходимо вычислить автокорреляционную функцию для всех вариантов сигналов заданной длины.

Полученные в результате выполнения программы оптимальные сигналы могут применяться для решения задач теории сигналов и использоваться, к примеру, в радиолокации. Для разработки программы предлагается использование языка Python с JIT-компилятором Numba и дополнительным модулем numba.cuda для параллельных вычислений на GPU.

Пакет Numba создан компанией Anaconda, Inc. (ранее – Continuum Analytics). Данный пакет даёт возможность ускорить программы при помощи высокопроизводительных функций, написанных непосредственно на языке Python. Использование специальных аннотаций (декораторов с различными параметрами) при функциях, перегруженных вычислениями и/или обрабатывающих массивы, позволяет компилировать код прямо во время исполнения (just-in-time) в машинные инструкции, приближая производительность такого кода к производительности программ, написанных на языках C/C++ и Fortran [1].

Пакет Numba генерирует оптимизированный машинный код с помощью компилятора LLVM. Поддерживается компиляция кода Python для последующего исполнения как на CPU, так и на GPU: в виде ядер и функций для устройств CUDA от nVidia или ядер и функций для устройств HSA (Heterogenous System Architecture) от AMD [2]. В данной статье будет рассматриваться применение пакета Numba для исполнения вычислений на GPU от компании nVidia. Простейший способ работы с Numba – это

применять декорирование с помощью декоратора @jit, предписывающее Numba компилировать помеченную функцию, используя заданные при её вызове типы параметров. Можно также указать заранее, для каких типов параметров компилировать функцию, – с помощью строки, именуемой сигнатурой функции и передаваемой декоратору, в которой ключевыми словами типов (или их сокращениями) указаны типы возвращаемого и передаваемых параметров [2].

Большая часть программного интерфейса CUDA доступна через модуль numba.cuda, подключаемый с помощью импорта вида from numba import cuda [3].

Цель данного исследования – исследование возможности использования пакета Numba от Anaconda, Inc для проведения ресурсоемких вычислений на примере определения автокорреляционной функции дискретных сигналов заданной длины.

Материалы и методы исследования

Для расчетов времени вычисления автокорреляционной функции сигнала заданной длины были разработаны четыре программы, каждая из которых является оптимизацией предыдущей программы:

– первая выполняет последовательное вычисление функции для каждой цепочки;

– вторая программа выполняет последовательное вычисление функции для заранее сгенерированного массива цепочек одной длины;

– третья программа выполняет параллельное вычисление функции для заранее сгенерированного массива цепочек одной длины с использованием CPU;

– четвертая программа выполняет параллельное вычисление функции для заранее сгенерированного массива цепочек одной длины с использованием GPU.

Рассмотрим данные программы подробнее (рис. 1, 2).

Последовательная реализация без предварительной генерации массива

```
def ak_func_demo(signal):
    akf = [ ]
    for i in range(len(signal)): akf.append(0)
    for i in range(len(signal)):
        for j in range(len(signal)):
            if i+j < len(signal) :
                akf[i] = akf[i] + signal[i+j] * signal[j]
    return akf
def int_to_signal(x):
    x = list(bin(x))[2::]
    for i in range(len(x)):
        x[i] = int(x[i])
        if x[i] == 0 : x[i]=-1
    return(x)
def max_amplitude(akf):
    for i in range(len(akf)):
        if i == 0 : akf[i] = 0
        akf[i] = abs(akf[i])
    max_amp = max(akf)
    return max_amp
for k in range(35):
    x = k
    s = [0, x+1]
    for i in range(2**x):
        signal = 2**x
        signal = signal + i
        signal_list = int_to_signal(signal)
        akf = ak_func_demo(signal_list)
        if max_amplitude(akf) <= s[1] :
            s[0] = bin(signal)[2::]
            s[1] = max_amplitude(akf)
```

Последовательная реализация с предварительной генерацией массива

```
def ak_func_demo(k):
    X = np.zeros((2**k,k+1))
    for i in range(2**k):
        signal = 2**k
        signal = signal + i
        signal_list=''
        while signal > 0:
            signal_list = str(signal % 2) + signal_list
            signal = signal // 2
        for j in range(len(signal_list)):
            if signal_list[j] == '0' : X[i][j]=-1
            if signal_list[j] == '1' : X[i][j]=1
    akf = np.zeros_like(X)
    max_amplitude = k + 2
    for i in range(2**k):
        for d in range(len(X[i])):
            for g in range(len(X[i])):
                if d+g < len(X[i]) :
                    akf[i][d] = akf[i][d]+X[i][d+g]*X[i][g]
        akf[i][0] = 0
        akf[i][0] = max(akf[i])
        if akf[i][0] <= max_amplitude :
            max_amplitude = akf[i][0]
            optimal_line = X[i]
    return optimal_line, max_amplitude
k=35
for i in range(k):
  s = ak_func_demo(i)
```

Рис. 1. Листинги программ с последовательным вычислением АКФ

Параллельная реализация вычислений на CPU

```
@numba.jit(nopython=True, parallel = True)
def ak_func_demo(k):
    X = np.zeros((2**k,k+1))
    for i in range(2**k):
        signal = 2**k
        signal = signal + i
        signal_list=''
        while signal > 0:
            signal_list = str(signal % 2) + signal_list
            signal = signal // 2
        for j in range(len(signal_list)):
            if signal_list[j] == '0' : X[i][j]=-1
            if signal_list[j] == '1' : X[i][j]=1
    akf = np.zeros_like(X)
    max_amplitude = k + 2

    for i in range(2**k):
        for d in range(len(X[i])):
            for g in range(len(X[i])):
                if d+g < len(X[i]) :
                    akf[i][d] = akf[i][d] + X[i][d+g] * X[i][g]
        akf[i][0] = 0
        akf[i][0] = max(akf[i])

        if akf[i][0] <= max_amplitude :
            max_amplitude = akf[i][0]
            optimal_line = X[i]
    return optimal_line, max_amplitude

k=35

for i in range(k):
    s = ak_func_demo(i)
```

Параллельная реализация вычислений на GPU

```
@cuda.jit(device=True)
def ak_func_demo(X, akf, k):
    akf = np.zeros_like(X)
    max_amplitude = k + 2
    for i in range(2**k):
        for d in range(len(X[i])):
            for g in range(len(X[i])):
                if d+g < len(X[i]) :
                    akf[i][d] = akf[i][d]+X[i][d+g]*X[i][g]
        akf[i][0] = 0
        akf[i][0] = max(akf[i])
        if akf[i][0] <= max_amplitude :
            max_amplitude = akf[i][0]
            optimal_line = X[i]
    return optimal_line, max_amplitude
k=35
X = np.zeros((2**k,k+1))
akf = np.zeros_like(X)
for i in range(2**k):
    signal = 2**k
    signal = signal + i
    signal_list=''
    while signal > 0:
        signal_list = str(signal % 2) + signal_list
        signal = signal // 2
    for j in range(len(signal_list)):
        if signal_list[j] == '0' : X[i][j]=-1
        if signal_list[j] == '1' : X[i][j]=1
device = cuda.get_current_device()
d_X = cuda.to_device(X)
d_akf = cuda.to_device(akf)
d_k = cuda.to_device(k)
tpb = device.WARP_SIZE
bpg = int(np.ceil((k)/tpb))
ak_func_demo[bpg, tpb](d_X, d_akf, d_k)
s = d_akf.copy_to_host()
```

Рис. 2. Листинги программ с параллельным вычислением АКФ

Программа с последовательным вычислением функции без предварительной генерации массивов выполнена с использованием встроенных функций языка Python и модуля Numpy. В данной программе перевод строки битов из нулей и единиц (простейший вариант представления сигнала заданной длины с двумя возможными значениями в такт времени), а также вычисление амплитуды бокового лепестка вынесены в отдельные функции.

Как было сказано ранее, во второй программе перед вычислением автокорреляционной функции генерируется весь массив цепочек заданной длины. Также во второй программе учитываются особенности работы пакета Numba. Пакет Numba не обрабатывает вызовы сторонних функций внутри задекорированной [4], поэтому перевод строки битов из нулей и единиц, а также вычисление амплитуды бокового лепестка были внесены в основную функцию.

Третья программа отличается от второй только наличием декоратора @jit. Так как вторая программа уже оптимизирована для параллельных вычислений, этого оказалось достаточно, чтобы вычисления проводились параллельно на CPU [5; 6].

В четвертой программе параллельные вычисления были перенесены на GPU. Тут нужно отметить одну важную концепцию: когда какие-либо вычисления должны быть выполнены на GPU, соответствующие данные должны быть перенесены в глобальную память GPU, а результаты вычислений после этого могут быть перенесены обратно

на хост. Эти операции выполняются при помощи функций cuda.to_device() и copy_to_host(), предоставляемых в библиотеке Numba на Python [7].

Листинги описанных программ приведены в таблице ниже. Для компактности в приведенных листингах опущены импорты библиотек и вызовы функций, не относящихся конкретно к вычислению автокорреляционной функции.

Результаты исследования и их обсуждение

Измерения времени выполнения программ с последовательным вычислением автокорреляционной функции и программы с параллельным вычислением на CPU производились на персональном компьютере со следующими характеристиками:

− Операционная система: Windows 10 Pro.
− CPU: Intel Xeon x3440 2.5 x 8 ГГц 64-bit.
− RAM: DDR3 2 x 8 Гб.
− ПО: Anaconda ver. 2020.11 с Python 3.9.6.

Из-за невозможности вычисления на GPU с использованием персонального компьютера ввиду отсутствия CUDA-устройства (видеокарты от компании nVidia) измерение времени выполнения параллельных вычислений на GPU было решено проводить с использованием сервиса Colaboratory от компании Google.

Colaboratory, или сокращенно Colab, – продукт компании Google Research. Colab позволяет писать и выполнять произвольный код Python через браузер и особенно хорошо подходит для машинного обучения, анализа данных и образования. С технической точки зрения Colab – это размещенная на хосте служба Jupyter Notebook, которая не требует настройки для использования, но при этом предоставляет бесплатный доступ к вычислительным ресурсам, включая графические процессоры на базе чипов nVidia [8].

Важно заметить, что из-за возрастающей сложности вычислений при каждом увеличении длины цепочки на один бит в два раза, ожидание результата становится неоправданно долгим. Поэтому результаты, ожидание которых достигает 4 ч и более, экстраполируются на более длинные цепочки.

Полученные результаты измерений представлены на графике (рис. 3).

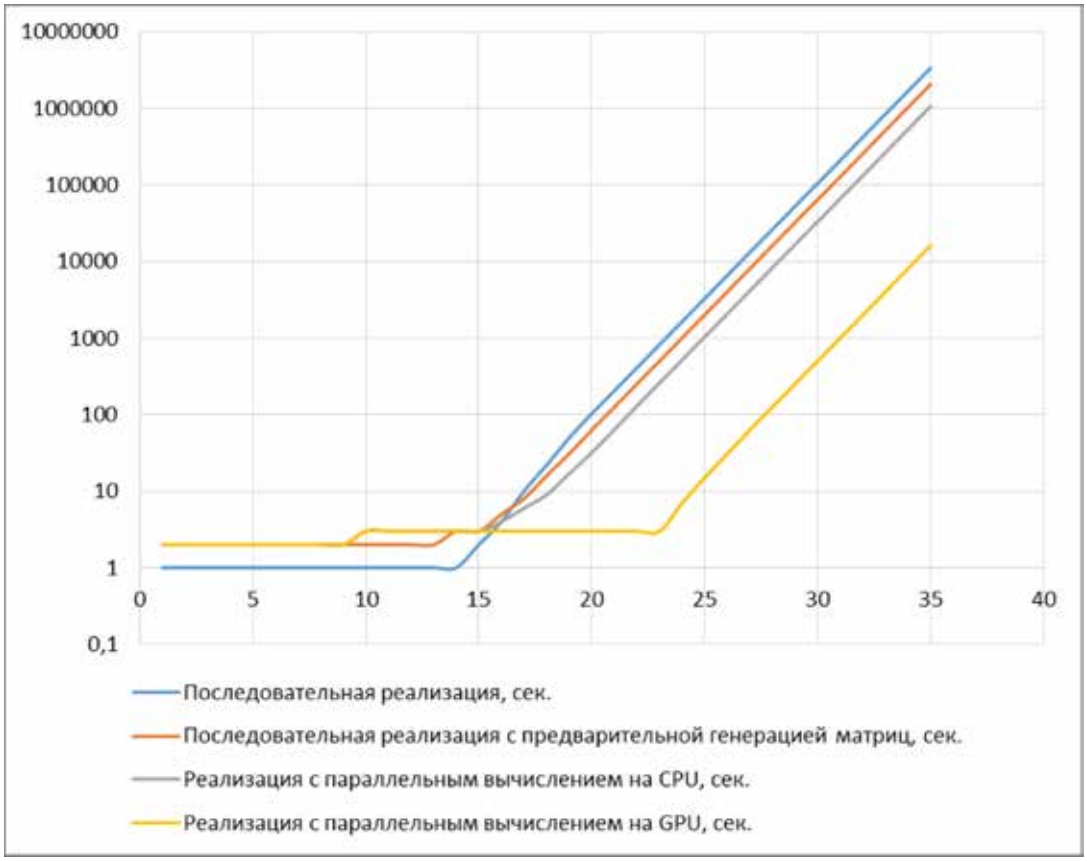

Рис. 3. Результаты измерений

Как видно из рисунка, вычисление автокорреляционной функции (АКФ) для цепочки битов длиной 35 символов занимает 39 суток. При этом даже просто оптимизация кода увеличивает скорость вычислений примерно в полтора раза. Вычисление АКФ для цепочки из 35 битов происходит уже в течение 24 суток.

Применение JIT-компилятора Numba на оптимизированном коде снижает время вычисления более чем в два раза. При использовании только CPU вычисление АКФ для цепочки из 35 бит занимает 12 суток. Это чуть более чем в три раза быстрее, чем вычисление при простейшей последовательной реализации.

При использовании JIT-компилятора Numba с модулем numba.cuda, который позволяет перенести вычисления на GPU от компании nVidia скорость расчета возросла в 66 раз по сравнению с вычислениями на CPU. В данном случае вычисление АКФ на цепочках до 35 символов занимает менее 5 ч. Возможно, при оптимизации программы результат можно улучшить и сократить время расчетов в несколько раз, однако эта задача выходит за рамки данной статьи.

Заключение

Полученные в рамках данной работы данные о скорости параллельных вычислений АКФ с использованием JIT-компилятора Numba в целом и модулем numba.cuda в частности показали, что указанные инструменты действительно помогают ускорить вычисление в десятки раз. Таким образом, с использованием Numba можно добиться вычисления АКФ функции не только до 35-битных цепочек, но и более длинных за адекватное время.

Список литературы

1. Антонюк В.А. GPU+Python Параллельные вычисления в рамках языка Python. М.: Физический факультет МГУ им. М.В. Ломоносова, 2018. 48 с.

2. Mungoli A. How GPU Computing literally saved me at work? // Walmart Global Tech Blog [Электронный ресурс]. URL: https://medium.com/walmartglobaltech/how-gpu-computing-literally-saved-me-at-work-fc1dc70f48b6 (дата обращения: 04.07.2021).

3. Anaconda Inc, Numba for CUDA GPUs. Kernel invocation. 2021 // Writing CUDA Kernels. [Электронный ресурс]. URL: https://numba.pydata.org/numba-doc/latest/cuda/kernels.html#kernel-invocation (дата обращения: 03.07.2021).

4. Ernest Kim Better Python Parallelization with Numba on CPU and GPU // Towards Data Science. [Электронный ресурс]. URL: https://towardsdatascience.com/better-parallelization-with-numba-3a41ca69452e/ (дата обращения: 07.07.2021).

5. Siu Kwan Lam, Graham Markall, Davin Potts ContinuumIO numbapro-examples // Examples of Numba and Accelerate in use. [Электронный ресурс]. URL: https://github.com/ContinuumIO/numbapro-examples (дата обращения: 06.07.2021).

6. Google Colaboratory Frequently Asked Questions 2021 // Colaboratory Frequently Asked Questions. [Электронный ресурс]. URL: https://research.google.com/colaboratory/faq.html (дата обращения: 14.07.2021).

7. Anaconda Inc, Numba documentation 2021 // Numba documentation. [Электронный ресурс] URL: http://numba.pydata.org/numba-doc/latest/ (дата обращения: 06.07.2021).