

Porównanie wydajności oraz ocena łatwości w użyciu wybranych narzędzi programowania równoległego.

Rafał Lenart

18 grudnia 2023

Cel projektu

Celem przeprowadzanej oceny jest określenie przypadków użycia oraz ewaluacja użyteczności i wydajności wybranych narzędzi służących do programowania równoległego. Szczególnej uwadze zostanie poddana biblioteka distributed-ranges z pakietu oneAPI firmy Intel.

Opinia ta zostanie wydana na podstawie implementacji wybranych algorytmów i pomiarze wydajności czasowej i pamięciowej programów. Dodatkowo, zważając na fakt, iż jedną z głównych intencji twórców biblioteki distributed-ranges, która będzie głównym kandydatem do porównania, jest usprawnienie procesu pisania kodu dla programistów, krytyce zostanie poddana łatwość pracy z danym narzędziem oraz ilość napisanego kodu.

Plan działania

1. Wybór podobnych do siebie technologii/narzędzi użytych do porównania.
2. Dobór algorytmów do implementacji.
3. Implementacja, pisanie kodu.
4. Pomiar prędkości oraz zużycia pamięci. Wyznaczenie innych kryteriów oceny takich jak ilość linijek kod lub porównanie czasu jaki zajęło pisanie i zrozumienie danego narzędzia.
5. Dokonanie porównania technologii z wzięciem pod uwagę wszystkich kryteriów.

Użyte technologie

Użyte technologie

Wyborem technologii do porównania kierowało podobieństwo przypadków użycia. Jako, że biblioteka distributed-ranges służy do programowania systemów z pamięcią rozproszoną, pozostałe narzędzia również będą pracować na pamięci rozproszonej. Narzędziem które jest używane wewnątrz biblioteki jest model SYCL który również będzie brany pod uwagę.

SYCL

SYCL jest modelem programowania pozwalającym aplikacji na przełączanie i współpracę pomiędzy różnymi akceleratorami sprzętowymi takimi jak CPU, GPU i FPGA (bezpośrednio programowalne macierze bramek).

Model ten dostarcza warstwę abstrakcji ułatwiającą użytkowanie ujednoliconej pamięci współdzielonej (Unified Shared Memory).

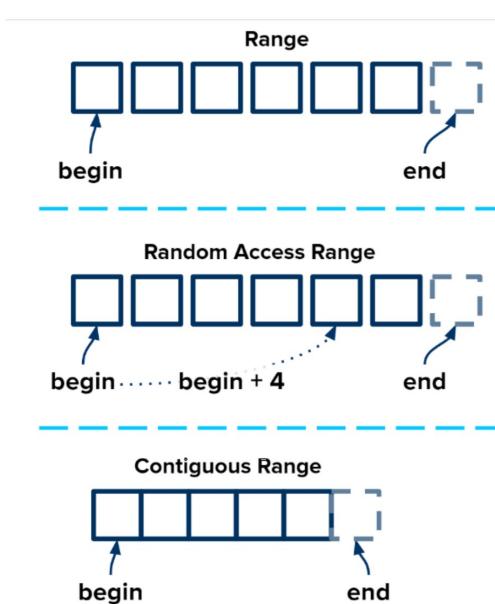
distributed-ranges

Biblioteka distributed-ranges to twór bazujący na SYCL oraz wprowadzonej w standardzie C++ bibliotece ranges.

Celem twórców narzędzia jest usprawnienie pracy z pamięcią rozproszoną podczas programowania systemów z wieloma GPU/CPU z równoczesnym utrzymaniem wydajności podobnej do modelu SYCL.

ranges

Range (z ang., zakres) - to struktura posiadająca początek i koniec (`begin()` i `end()`) udostępniająca ustandardyzowany sposób iterowania po jej wartościach. Standard C++20 udostępnia kilka konceptów posiadających różne cechy. na przykład `random_access_range` lub `contiguous_range`.



Message Passing Interface (MPI, z ang., interfejs transmisji wiadomości) to standard przesyłania komunikatów między procesami programów równoległych. Jest on obecnie dominującym modelem wykorzystywanym w superkomputerach. MPI umożliwia pracę na rozproszonych systemach pamięci.

Wybrane algorytmy

Radix-2 FFT

Szybka transformata Fouriera Radix-2 to algorytm który dzieli wektor wejściowy na dane indeksowane parzyście i te indeksowane nieparzyście. Po podziale dla obu wektorów obliczana jest wartość dyskretnej transformaty Fouriera po czym wyniki są łączone w całość. Ważnym ograniczeniem algorytmu jest to, że dla n będącego wielkością wektora wejściowego prawdziwe jest

$$x \in \mathbb{N}$$

$$n = 2^x$$

Zastosowana metoda "dziel i zwyciężaj" sprawia, że algorytm jest dobrym kandydatem do zrównoleglania.

Radix-2 FFT

Wzór dyskretnej transformaty Fouriera

$$\sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}$$

Radix-2 FFT

Przykład pseudokodu wykonania algorytmu:

```
import cmath
def radix2(vec):
    n = length(vec)
    if n <= 1:
        return x
    # Podział na dwie równe części
    even = radix2(vec[0::2])
    odd = radix2(vec[1::2])
    # Łączenie rezultatów
    for k in range(n // 2):
        t = cmath.exp(-2j * cmath.pi * k / n) * odd[k]
        vec[k] = even[k] + t
        vec[k+n // 2] = even[k] - t
    return vec
```

Algorytm QR Obliczania wartości własnych

Dla $A \in \mathbb{C}^{n \times n}$ liczba $\lambda \in \mathbb{C}$ to wartość własna macierzy a niezerowy wektor $x \in \mathbb{C}^n$ jest wektorem własnym, jeśli

$$Ax = \lambda x$$

liczby λ to pierwiastki wielomianu charakterystycznego macierzy A. Widmem macierzy nazywamy zbiór jej wartości własnych

Algorytm QR Obliczania wartości własnych

Macierze $Q, R \in \mathbb{R}^{n \times n}$ tworzą rozkład QR macierzy $A \in \mathbb{R}^{n \times n}$ kiedy

$$A = QR$$

R jest macierzą trójkątną górną natomiast Q jest macierzą ortogonalną.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Macierz trójkątna górna

Algorytm QR Obliczania wartości własne

Macierz ortogonalna to taka macierz kwadratowa $A \in M_n(\mathbb{R})$ która spełnia równość

$$A^{-1} = A^T$$

$$\bullet \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Algorytm QR Obliczania wartości własnych

Macierz Hessenberga - to taka macierz trójkątna górna która bezpośrednio pod diagonalą posiada dodatkową przekątną.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ 0 & a_{3,2} & \cdots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix}.$$

Dowolną macierz można przekształcić do tej postaci metodą Householdera

metoda Householdera

```
import numpy as np

def householder(A):
    m, n = A.shape
    Q = np.eye(m)

    for k in range(min(m-1, n)):
        x = A[k:, k]
        # Wektor Householdera
        v = x - np.linalg.norm(x) * np.eye(len(x), 1, 0)
        H = np.eye(m)
        H[k:, k:] -= 2 * np.outer(v, v) / np.linalg.norm(v)**2
        A = H @ A
        Q = Q @ H

    return Q, A
```

Algorytm QR Obliczania wartości własnych

```
def qr_iteration(A, max_iter=1000, tol=1e-6):
    Q, R = np.linalg.qr(A)

    for i in range(max_iter):
        A = np.dot(R, Q)
        Q, R = np.linalg.qr(A)

    # Sprawdzamy warunek stopu
    if np.abs(np.diag(A) - np.diag(R * Q)).max() < tol:
        break

    eigenvalues = np.diag(A)
    return eigenvalues
```

Conjugate Gradient