Piotr Radecki
Dawid Krzos

Równanie Ciepła 2D

1. Wstęp Teoretyczny

Równanie ciepła w 2D może być zapisane jako:

$$\frac{du}{dt} - \alpha\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right) = 0$$

Gdzie:

- u(t, x, y) jest funkcją ciepła zależną od czasu i położenia. Nazywany potem obrazem.
- t to czas
- x,y to współrzędne

Do rozwiązania równania zostanie użyta metoda skończonych metod. Może być zapisana jako:

$$f'(a) = \frac{f(a+h) - f(a)}{h}$$

Oznacza to, że przyblizamy pochodną skończonymi różnicami.

Jeśli oznaczymy:

$$x_i = \Delta x$$

$$y_i = \Delta y$$

$$z_i = \Delta z$$

$$u_{i,j}^k = u(x, y, t)$$

To możemy zapisać:

$$\frac{u_{i,j}^{k+1}}{u_{i,j}^k} - \alpha\left(\frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{\Delta x^2} + \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{\Delta y^2}\right) = 0$$

Co można także zapisać jako:

$$u_{i,j}^{k+1} = \gamma\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - 4u_{i,j}^k\right) + u_{i,j}^k$$

Ten wzór będzie bezpośrednio implementowany w programach.

Przy czym elementy skrajne obrazu będą miały stałe wartości i będą stanowić tzw. warunki przegowe. Ich niezmienna temperatura będzie wymuszać zmiany temperatury wewnątrz obrazu.

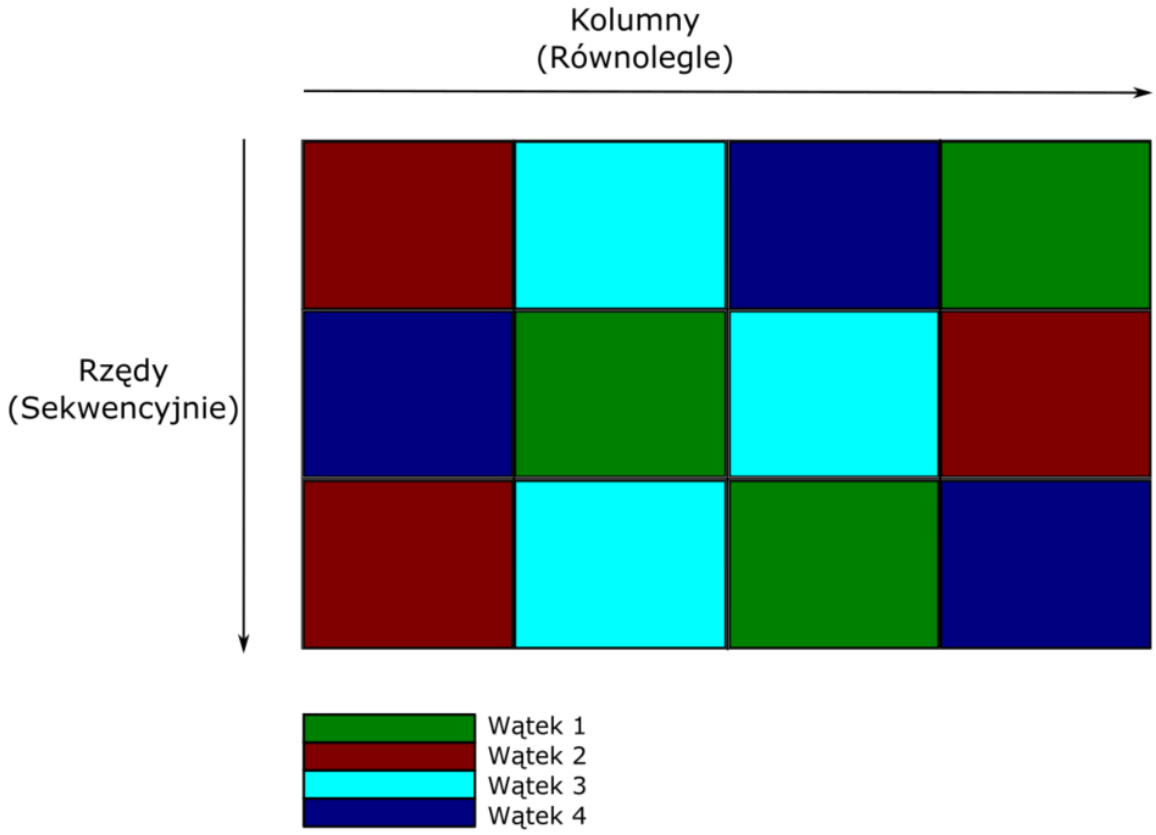
Wszystkie poniższe równania można także zastąpić opisem intujcyjnym. Zmiana temperatury w czasie w danym punkcie zależy od różnicy temperatur wokół tego punktu. Jeśli różnica jest duża temperatura szybko się zmienia. Jest to zwykłe zjawisko dyfuzji.

2. Implemencja OpenMP

W implementacji OpenMP użyto pętli for z dyrektywą #pragma omp parallel for.

Program oblicza kolejno pewną ilość iteracji i w każdej akutalizuje wartość funkcji 2D. Iteracja po kolejnych rzędach odbywa się sekwencyjnie. Natomiast elementy rzędu są dzielone pomiędzy ilość wątków w każdej iteracji.

Jest to zobrazowane poniżej:



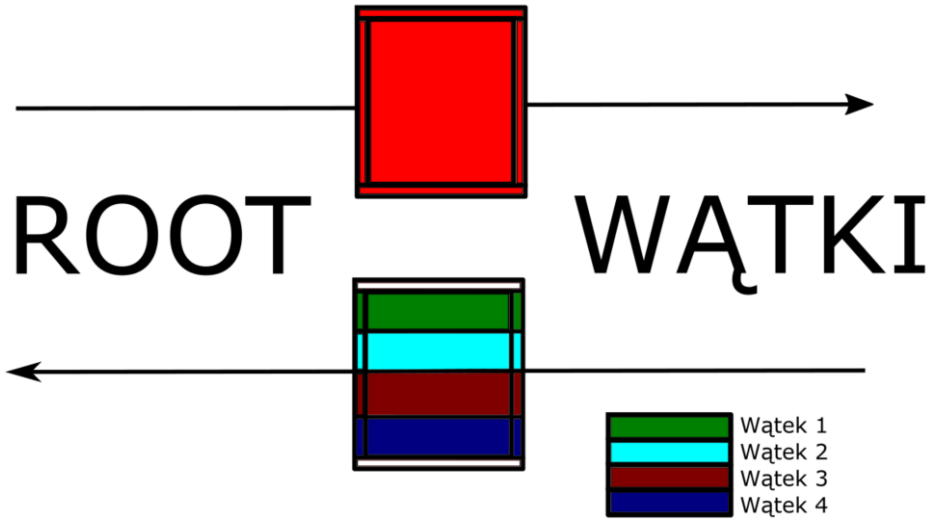
Kod jest dostępny w folderze heat_openmp w załączonym pliku.

3. Implementacja MPI

W implementacji MPI jest użyty następujący algorytm:

- Root rozgłasza pełny obraz za pomocą *Broadcast*.
- Każdy z wątków oblicza swoją część obrazu – tym razem obraz podzielony rzędami, nie kolumnami.
- Za pomocą *Gather* root zbiera części obrazu poza górną i dolną krawędzią, które nie są obliczane.
- Krawędź górna i dolna są kopiowane z obrazu początkowego.
- Całość jest powtarzana aż do wyczerpania się ustalonej liczby iteracji.

Ograniczeniem algorytmu jest to, że (liczba rzędów - 2) jest musi być podzielna przez ilość wątków. Poniżej ilustracja algorytmu.



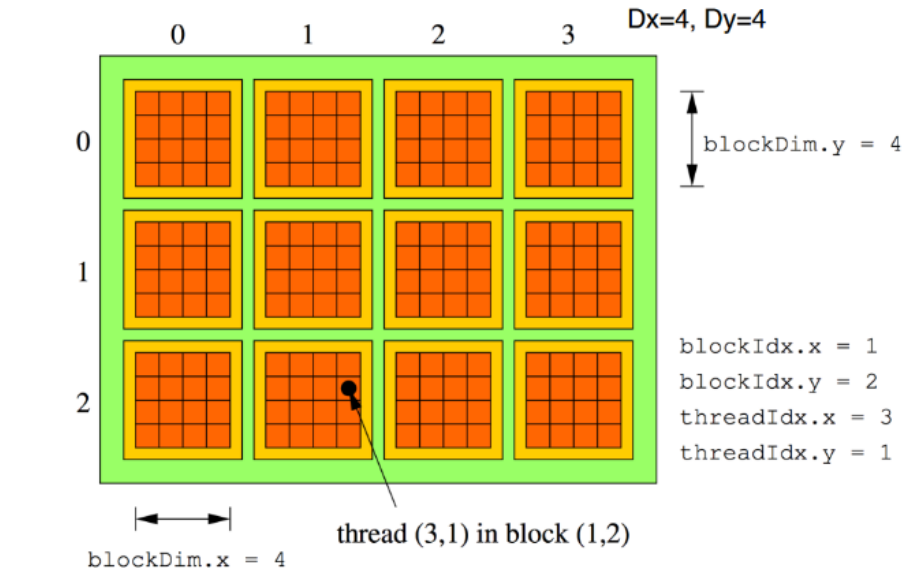
Kod jest dostępny w folderze heat_mpi.c.

4. Implementacja CUDA

Implementacja równania w CUDA przygotowana została w trzech wersjach, które różnią się metodami zarządzania pamięcią. We wszystkich wersjach przyjęto taką samą konfigurację kernela tj. dwuwymiarowa siatka mapowana jest na dwuwymiarowe bloki wątków tak, by sąsiednie wątki trafiały na sąsiednie elementy siatki. Przy pomocy dyrektywy #define

definiowany jest rozmiar pojedynczego bloku (liczba wątków w każdym z kierunków). Następnie, w zależności od rozmiaru siatki obliczana jest ilość bloków.

Na poniższym rysunku przedstawiony został sposób indeksowania wątków (w naszej implementacji indeks wątku odpowiada indeksowi elementu na dwuwymiarowej siatce danych).



threadIdx:

$$x = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x} = 1 \times 4 + 3 = 7$$

$$y = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y} = 2 \times 4 + 1 = 9$$

https://kdm.icm.edu.pl/Tutorials/GPU-intro/introduction.en/

Opis poszczególnych wersji:

- **heat_gpu.cu**

  W podstawowej wersji, z każdą iteracją pętli głównej, dane kopiowane są najpierw z hosta do GPU (*cudaMemcpyHostToDevice*) . Następnie, tworzony jest kernel, w którym każdy z wątków wykonuje obliczenia dla odpowiadającego sobie elementu. Indeksy elementów siatki wyznaczane są w oparciu o identyfikatory wątków oraz bloków. Po wykonaniu kodu kernela, dane kopiowane są z powrotem do pamięci hosta (*cudaMemcpyDeviceToHost*). Przed kolejną iteracją zamieniane są wskaźniki do zestawów danych.

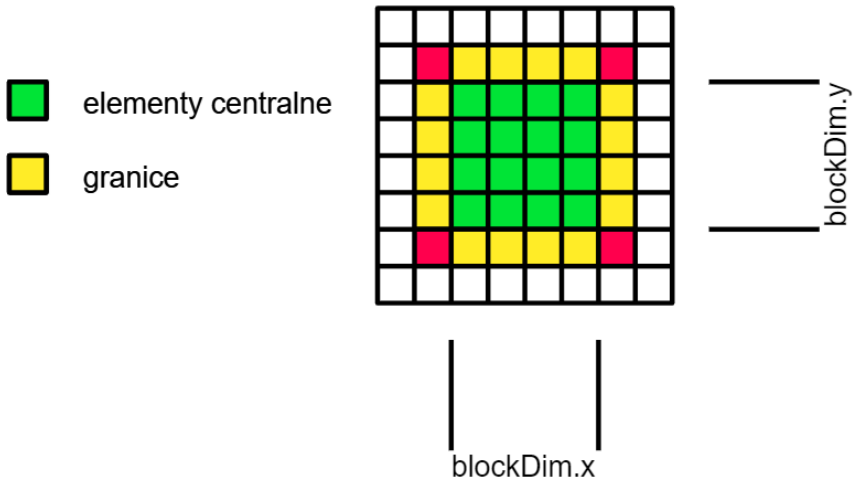
- **heat_gpu_data.cu**

  W tej wersji została zredukowana ilość transferów danych host->device oraz device->host. Taka optymalizacja mogła zostać zastosowana gdyż nie ma potrzeby posiadania dostępu do danych po każdej iteracji. Zamiast tego, na czas wykonywania obliczeń, dane

przechowywane są po stronie GPU. Dopiero po zakończeniu wszystkich iteracji kopiowane są z powrotem do hosta.

- **heat_gpu_shared.cu**

W ostatecznej wersji została dodatkowo zredukowana ilość zawołań do pamięci globalnej w kernelu GPU. Zamiast tego wykorzystana została pamięć typu shared, która jest współdzielona pomiędzy wszystkimi wątkami w obrębie bloku. Wymaga to wczytania danych do współdzielonego bufora na początku wykonywania kodu kernela. Każdy z wątków wczytuje swój element. Dodatkowo, wątki, których elementy leżą na granicy bloków, wczytują także kolejny sąsiadujący element. Przed rozpoczęciem wykonywania obliczeń upewniamy się, że wszystkie wątki w obrębie bloku dokonały zapisu do współdzielonego bufora (*__syncthreads()*). Na poniższym rysunku przedstawiony został sposób mapowania elementów z siatki do współdzielonej pamięci. Do bufora trafiają elementy centralne (kolor zielony) oraz elementy graniczne (kolor żółty).



Porównanie wydajności algorytmu dla 3000 iteracji na siatce o rozmiarze 256x208 i warunkach brzegowych TOP = BOTTOM = LEFT = RIGHT = 100. Przyjęty rozmiar bloku to 16x16.

Wersja	Czas [ms]
heat_gpu.cu	701.04
heat_gpu_data.cu	9.68
heat_gpu_shared.cu	7.48

Jak można było się spodziewać, najszybsza okazała się wersja korzystająca ze współdzielonej pamięci. Znaczącą optymalizacją okazała się redukcja transferów danych pomiędzy hostem a urządzeniem.

5. Format wejścia wyjścia

Warunki początkowe i rozmiar obrazu ustawiane są makrami w programie. Możliwe jest wybranie temperatury poszczególnych krawędzi i pozostałej części obrazu (np. górna krawędź 100, pozostałe krawędzie 0, reszta elementów 0).

Wyjście programu jest podawane w postaci tablicy kompatybilnej z np.array w Pytonie. Możliwe jest obejrzenie wyników za pomocą matplotlib używając skryptu show.py.

Przykładowy uzyskany obrazy.

Warunki początkowe: Brzegi 0, reszta obrazu nagrzana do 100.

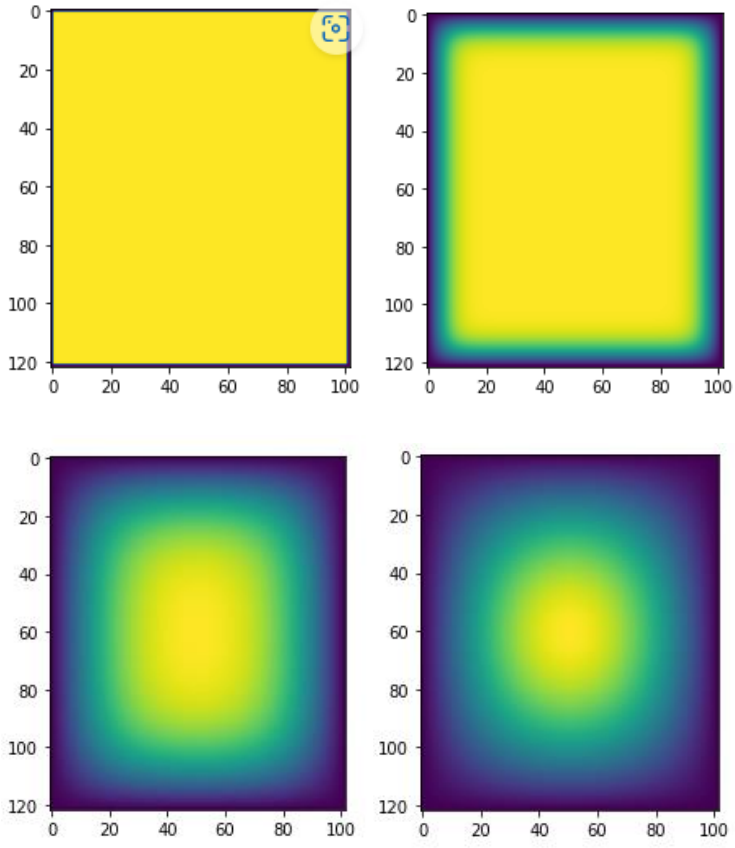


Fig. Obraz po kolejno: 0, 100, 1000 i 1000 iteracji.

Można zaobserwować wychładzanie środkowej części obrazu.

Przykładowy obraz ma wymiary 100 x 122. W przypadku większych obrazów może być problem z kopiowaniem tekstu zawierającego macierz.

6. Analiza wydajności algorytmów

W analizie sprawdzano jak szybko algorytmy OpenMP, OpenMPI oraz CUDA policzą 1000 iteracji algorytmu dla obrazu o wymiarach 8002 x 2002 (8000 x 2000 plus brzegi).

Uzyskane czasy:

Biblioteka	**1 wątek**	**2 wątki**	**4 wątki**
OpenMP	27742 ms	13439 ms	9981 ms
OpenMPI	190299 ms	138083 ms	147892 ms
	Blok 4x4	**Blok 16x16**	**Blok 32x32**
CUDA (shared)	1454 ms	278 ms	258 ms

Wyniki wskazują, że im większy rozmiar bloku CUDA tym krótszy czas wykonywania algorytmu. Dzieje się tak dlatego, że im więcej wątków znajduje się w bloku, tym większe jest użycie pamięci współdzielonej i mniej operacji odczytu i zapisu do pamięci globalnej.