Raport

Zrównoleglenie Heat Transfer 2D przy użyciu CUDA

Franciszek Grzędzicki grzedzicki@mat.umk.pl

Celem niniejszego raportu jest przedstawienie problemu rozchodzenia ciepła oraz zaimplementowanie jego rozwiązania wykorzystując bibliotekę CUDA w języku Python.

1. Abstrakt

Do rozwiązania problemu wykorzystano równanie przewodnictwa cieplnego metodą różnic skończonych dzięki czemu w prosty sposób otrzymamy wzór możliwy do zaimplementowania do siatki 2D. Wykorzystałem 3 sposoby wykonania obliczeń: czysty Python, Numba oraz CUDA.

2. Wzór

Podstawowy wzór na przewodnictwo cieplne: $\frac{\partial u}{\partial t} - \alpha \nabla u = 0$, aby móc wykorzystać wzór w

przestrzeni 2D wzór ten zapisujemy jako: $\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x} + \frac{\partial^2 u}{\partial y}\right) = 0$, gdzie t to jednostka czasu, x oraz y to współrzędne w przestrzeni, a α to współczynnik wyrównania temperatur.

Wykorzystując metodę różnic skończonych wyprowadzoną z szeregu Taylora otrzymujemy $f'(a) \approx \frac{f(a+h)-f(a)}{h}$. Następnie zapiszemy nasze zmienne jako: $x_i = i\Delta x, y_j = j\Delta y, t_k = k\Delta t$.

Tym czego szukamy jest $u(x, y, t) = u_{i,j}^k$. Dzięki temu otrzymujemy

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

Zakładamy, że siatka jest kwadratowa, więc $\Delta x = \Delta y$, wtedy mamy

$u_{i,j}^{k+1} = \gamma\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - 4u_{i,j}^k\right) + u_{i,j}^k$, gdzie $\gamma = \alpha \frac{\Delta t}{\Delta x^2}$.

3. Maszyna testowa

Testy przeprowadzona na maszynie:

- CPU: Intel(R) Xeon(R) CPU @ 2.20GHz
- GPU: Tesla T4 15.0 GB
- RAM: 12.7 GB

4. Implementacja sekwencyjna

```
def cpu_calculate(u):
    newArray = u.copy()
    for i in range(1, size-1, delta_x):
        for j in range(1, size-1, delta_x):
            val = (gamma * (u[i+1, j] + u[i-1, j] + u[i, j+1] + u[i, j-1] - 4*u[i, j]) + u[i, j])
            newArray[i, j] = math.floor(val * 1e3) / 1e3
    return newArray
```

5. Implementacja Numba

```
@njit
def cpu_calculate_numba(u):
    newArray = u.copy()
    for i in range(1, size-1, delta_x):
        for j in range(1, size-1, delta_x):
            val = (gamma * (u[i+1, j] + u[i-1, j] + u[i, j+1] + u[i, j-1] - 4*u[i, j]) + u[i, j])
            newArray[i, j] = math.floor(val * 1e3) / 1e3
    return newArray
```

6. Implementacja Numba – CUDA

```
@cuda.jit
def parallel_gpu(u,gpu_new_array_par):
    i, j = cuda.grid(2)
    if ((i > 0 and i < size-1) and (j > 0 and j < size-1)):
        val = (gamma * (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j]) + u[i][j])
        gpu_new_array_par[i][j] = math.floor(val * 1e3) / 1e3
```

```
def start_parallel_gpu(iter):
    start_parallel_gpu = cp.cuda.Event()
    end_parallel_gpu = cp.cuda.Event()
    start_parallel_gpu.record()
    d_a = numba.cuda.to_device(original_array)
    d_b = numba.cuda.to_device(original_array)
    for i in range(0,iter):
            parallel_gpu[(nblocksx, nblocksy), (nthreads, nthreads)](
              d_a, d_b
            )
            d_a.copy_to_device(d_b)
    gpu_result = d_a.copy_to_host()
    end_parallel_gpu.record()
    end_parallel_gpu.synchronize()
    t_gpu = cp.cuda.get_elapsed_time(start_parallel_gpu, end_parallel_gpu)
    return gpu_result, t_gpu
```

7. Sposób implementacji

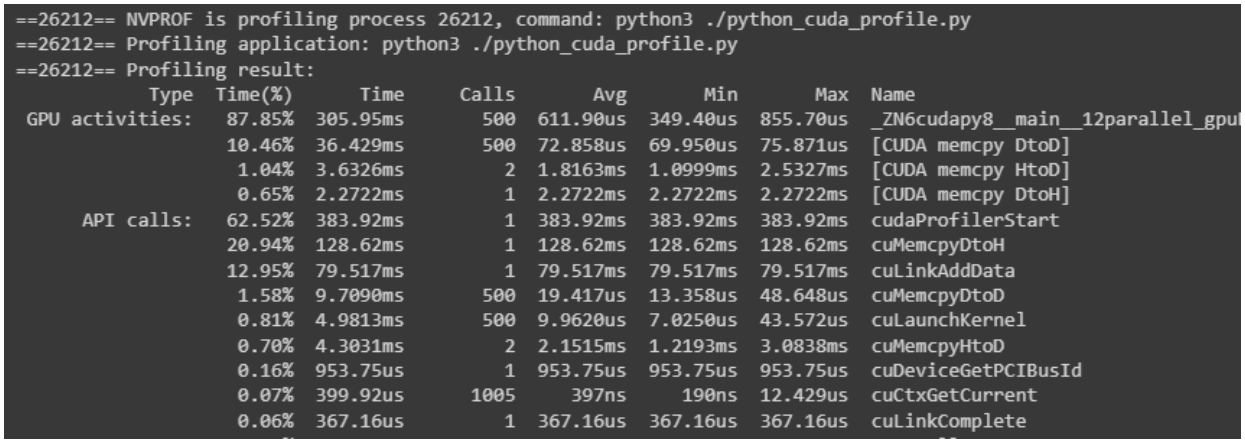
Zgodnie ze wzorem, posiadając siatkę 2D obliczeń dokonuje się nie zmieniając wartości brzegowych. Mając to na uwadze obliczeń dokonujemy od pierwszego wiersza oraz pierwszej kolumny, nie dotykamy 0, a kończmy na przedostatniej linii oraz przedostatniej kolumnie. Do obliczenia temperatury danej komórki musimy znać temperatury z poprzedniego kroku jej sąsiada z góry, dołu, lewej strony oraz prawej.

Język Python jest przyjemnym oraz łatwym językiem do pisania kodu, lecz niestety wykorzystuje interpreter do wykonywania kodu. Z tego powodu czas wykonania jest bardzo wysoki przy nawet niewielkich obliczeniach. Dlatego bardzo często wykorzystuje się Numbę – bibliotekę Pythona, która umożliwia wykonanie kodu przez kompilację just-in-time, która przekształca kod Pythona w zoptymalizowany kod maszynowy, co przyspiesza jego wykonanie. Mimo sporych zysków czasowych w porównaniu z czasem wykonanie przez zwykły kod Python, postanowiłem sprawdzić czy wykorzystanie CUDA może jeszcze bardziej polepszyć czasy.

Istnieją różne sposoby na implementację CUDA, jak np. biblioteka CuPy, która umożliwia wykorzystanie GPU do przyspieszenia obliczeń numerycznych. W moim przypadku postanowiłem wykorzystać zarazem tę bibliotekę aby móc wykorzystywać funkcjonalność taką jak *cp.cuda.Event* oraz *.record* w celu dokonania benchmarków czasów wykonania kodu. Do obsługi GPU, tablicy oraz kopiowania danych z CPU na GPU i odwrotnie wykorzystałem *numba.cuda*.

8. Wyniki czasowe

	Python	Numba	Numba - CUDA
100x100 100s	2.182s	**0.001s**	0.422s
100x100 1000s	21.903s	**0.005s**	0.367s
1.000x1.000 100s	4:20min	0.72s	**0.05s**
1.000x1.000 1000s	41min	0.70s	**0.35s**
10.000x10.000 100s	-------	54.31s	**1.26s**
10.000x10.000 1.000s	-------	9min	**7.81s**
10.000x10.000 10.000s	-------	1:30h	**1:13min**



```
==26212== NVPROF is profiling process 26212, command: python3 ./python_cuda_profile.py
==26212== Profiling application: python3 ./python_cuda_profile.py
==26212== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   87.85%  305.95ms       500  611.90us  349.40us  855.70us  _ZN6cudapy8__main__12parallel_gpu
                   10.46%  36.429ms       500  72.858us  69.950us  75.871us  [CUDA memcpy DtoD]
                    1.04%  3.6326ms         2  1.8163ms  1.0999ms  2.5327ms  [CUDA memcpy HtoD]
                    0.65%  2.2722ms         1  2.2722ms  2.2722ms  2.2722ms  [CUDA memcpy DtoH]
      API calls:   62.52%  383.92ms         1  383.92ms  383.92ms  383.92ms  cudaProfilerStart
                   20.94%  128.62ms         1  128.62ms  128.62ms  128.62ms  cuMemcpyDtoH
                   12.95%  79.517ms         1  79.517ms  79.517ms  79.517ms  cuLinkAddData
                    1.58%  9.7090ms       500  19.417us  13.358us  48.648us  cuMemcpyDtoD
                    0.81%  4.9813ms       500  9.9620us  7.0250us  43.572us  cuLaunchKernel
                    0.70%  4.3031ms         2  2.1515ms  1.2193ms  3.0838ms  cuMemcpyHtoD
                    0.16%  953.75us         1  953.75us  953.75us  953.75us  cuDeviceGetPCIBusId
                    0.07%  399.92us      1005     397ns     190ns  12.429us  cuCtxGetCurrent
                    0.06%  367.16us         1  367.16us  367.16us  367.16us  cuLinkComplete
```

9. Podsumowanie

Wykorzystując CUDA udało się uzyskać zaskakująco szybkie wyniki obliczeń, które były równe wynikom z obliczeń na CPU. Łatwość pisania w Pythonie w połączeniu z mocą obliczeniową GPU sprawia, że wykorzystywanie Pythona w celu cięższych obliczeń staje się wydajne.

10. Bibliografia

- https://www.youtube.com/watch?v=LJPSRp0ZeJo&list=PLZOZfX_TaWAHZOgn8CRjpqRElp5Dd-GaY&index=13&ab_channel=CPPMechEngTutorials
- https://en.wikipedia.org/wiki/Heat_equation
- https://studyres.com/doc/16038790/finite-difference-methods-in-2d-heat-transfer