JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 10 (2019) 32-35
Wysłane: 2018-11-25
Przyjęte: 2018-11-29

Wykorzystanie CPU i GPU do obliczeń w Matlabie

Jarosław Woźniak*
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule zostały przedstawione wybrane rozwiązania wykorzystujące procesory CPU oraz procesory graficzne GPU do obliczeń w środowisku Matlab. Porównywano różne metody wykonywania obliczeń na CPU, jak i na GPU. Zostały wskazane różnice, wady, zalety oraz skutki stosowania wybranych sposobów obliczeń.

Słowa kluczowe: CPU; GPU; Matlab

*Autor do korespondencji.

Adres e-mail: jaroslaw.wozniak@pollub.edu.pl

The use of CPU and GPU for calculations in Matlab

Jarosław Woźniak*
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The article presents selected solutions using CPU processors and GPUs for calculations in the Matlab environment. Various methods of performing calculations on the CPU as well as on the GPU were compared. Differences, disadvantages, advantages and effects of using selected calculation methods have been indicated.

Keywords: CPU; GPU; Matlab

*Corresponding author.

E-mail address: jaroslaw.wozniak@pollub.edu.pl

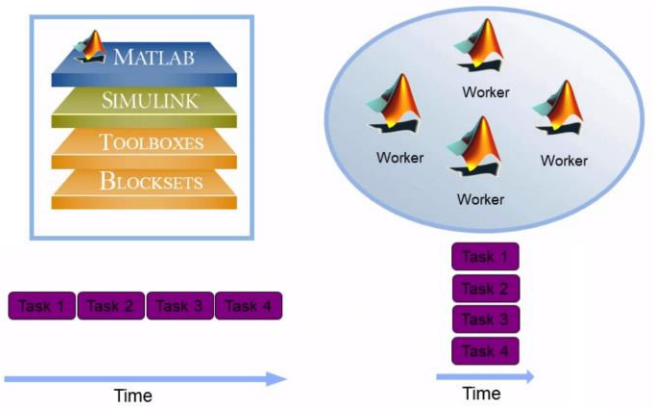
1. Wstęp

Matlab jest to język programowania, przeznaczony do obliczeń technicznych. Jest to również interaktywne środowisko, do którego wbudowano operacje na wektorach, macierzach i tablicach, które tworzą matematyczną podstawę do obliczeń naukowych i technicznych. Dzięki temu możliwe jest szybsze tworzenie i wykorzystywanie algorytmów obliczeniowych w porównaniu do języków tradycyjnych (C, Fortran), gdyż nie jest koniecznym deklarowanie zmiennych ich typów i adresów [1]. W tak popularnym i atrakcyjnym środowisku jakim jest Matlab istotne jest wykorzystanie nowoczesnych technologii takich jak obliczenia równoległe [2]. Kilka wybranych metod implementacji obliczeń równoległych w Matlabie prezentuje niniejszy artykuł.

2. Obliczenia równoległe w Matlabie

Do obliczeń równoległych w Matlabie służy dodatek Parallel Computing Toolbox. Pozwala on na rozwiązanie obliczeń dużej ilości danych przy użyciu wielordzeniowych procesorów, kart graficznych oraz klastrów komputerowych. Wspiera również konstrukcje wysokiego poziomu, równoległe pętle for, specjalne typy tablic i spersonalizowane algorytmy numeryczne. Używając tych funkcjonalności, nie jest konieczne programowania CUDA i MPI. Można skorzystać z tego dodatku wraz z dodatkiem Simulink do równoległego uruchomienia wielu symulacji modeli [3].

Narzędzie to pozwala wykorzystać pełną moc obliczeniową komputerów wielordzeniowych uruchamiając aplikację na tzw. *workerach*, czyli silnikach obliczeniowych Matlaba, które działają lokalnie. Bez zmian kodu można uruchamiać te same aplikacje w klastrze komputerowym lub usługach gridowych (za pomocą *MATLAB Distributed Computing Server*). Aplikacje można uruchamiać interaktywnie lub wsadowo [4].



Rys. 1. Podział na zadania i ich realizacja w czasie dla jednego rdzenia (z lewej) i wielu rdzeni – workerów (z prawej) [5]

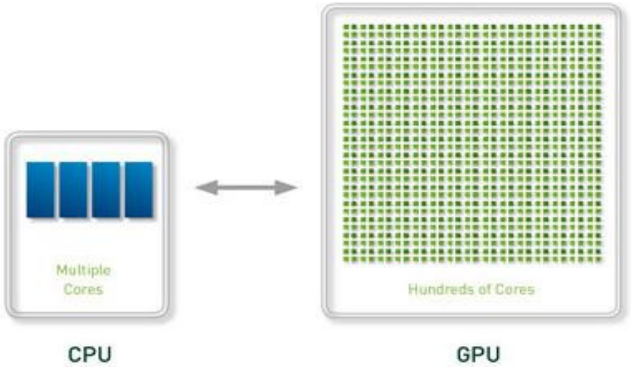
Równoległe przetwarzanie danych pozwala na jednoczesne wykonywanie wielu obliczeń. Duże problemy można często podzielić na mniejsze, które są następnie rozwiązywane w tym samym czasie, co prezentuje powyższy rysunek.

Przedstawia on również rozłożenie zadań w czasie – w przypadku szeregowego wykonywania zadań na jednym rdzeniu oraz ich równoległego wykonania przy wielu workerach – związanych z liczbą rdzeni w procesorze. Główne powody, dla których warto rozważyć przetwarzanie równoległe, to:

1) oszczędność czasu poprzez rozprowadzenie zadań i wykonanie ich równolegle,
2) rozwiązanie problemów z dużą ilością danych poprzez ich dystrybucję,
3) pełniejsze wykorzystanie zasobów komputera i skalowanie do klastrów i chmury obliczeniowej [6, 7].

3. Procesory GPU

Obliczenia na GPU rozpoczęto ponad dekadę temu, kiedy programiści zaczęli wykorzystywać jednostki przetwarzania graficznego (GPU) do zadań obliczeniowych, wymagających duże ilości danych. Na przestrzeni lat naukowcy zajmujący się komputerami wraz z badaczami w dziedzinach takich jak np. obrazowanie medyczne dostrzegli ogromny potencjał wykorzystania procesorów graficznych w obliczeniach wysokiej wydajności (HPC) do ogólnych celów. Termin GPGPU (Komputery ogólnego przeznaczenia GPU) został szybko ustanowiony [8].

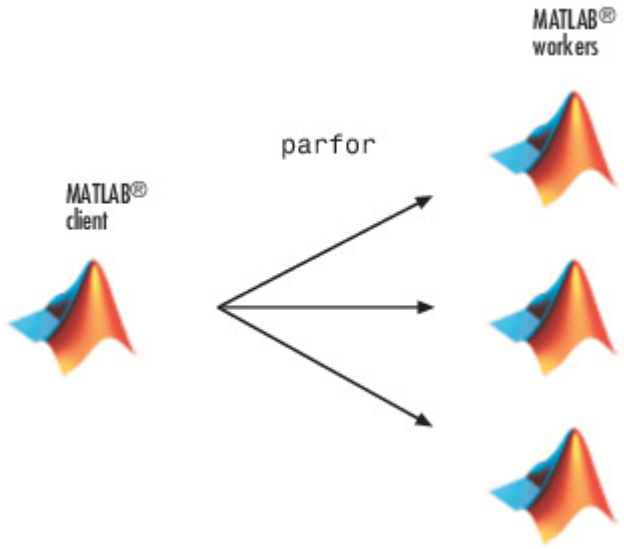


Rys. 2. Porównanie ilości rdzeni w CPU oraz w GPU [8].

Procesory CPU i procesory graficzne GPU przetwarzają zadania na różne sposoby. CPU składa się z zaledwie kilku rdzeni, które są zoptymalizowane do sekwencyjnego przetwarzania szeregowego. Natomiast GPU jest zbudowany z setek lub tysięcy rdzeni, które mogą obsługiwać tysiące wątków jednocześnie [9].

4. Przykład wykorzystania pętli *parfor*

W tym przykładzie pokazane jest działanie wolnej pętli for, a następnie obliczenia są przyspieszone za pomocą pętli parfor, która rozdziela wykonanie iteracji pętli pomiędzy wątki w puli równoległej.



Rys. 3. Zobrazowanie działania pętli parfor [10].

Ten przykład oblicza promień spektralny macierzy i przekształca pętlę for w pętlę parfor. Dodatkowo dodane są polecenia mierzące czas wykonania poleceń, by zmierzyć wynikowe przyspieszenie [11].

1. W edytorze Matlab wprowadzona jest pętla for oraz dodane polecenia tic i toc, aby zmierzyć czas, który upłynął, co prezentuje przykład 1. Wszystkie instrukcje są umieszczone w skrypcie o nazwie *parforTest.m*.

Przykład 1. Pomiar czasu funkcjami tic i toc.

```
tic
n = 200;
A = 500;
a = zeros(n);
for i = 1:n
 a(i) = max(abs(eig(rand(A))));
end
toc
```

2. Uruchomienie skryptu i zanotowanie upływu czasu.

```
>> parforTest

Elapsed time is 27.883551 seconds.
```

3. W skrypcie pętla for zastąpiona jest przez pętlę parfor – przykład 2.

Przykład 2. Wykorzystanie pętli parfor.

```
tic
n = 200;
A = 500;
a = zeros(n);
parfor i = 1:n
 a(i) = max(abs(eig(rand(A))));
end
toc
```

4. W tym momencie zalecane jest włączenie parallel pool z narzędzia Parallel Computing Toolbox, które uruchamia workery i udostępnia im kod (ta procedura nie jest niezbędna, ponieważ kompilator przy sprawdzeniu frazy parfor, automatycznie uruchomi parallel pool i zezwoli wszystkim

workerom na pracę z kodem, lecz czas tego uruchomienia zakłóciłby rzeczywiste różnice pomiarowe dla sprawdzanego przykładu).

5. Uruchomienie skryptu i zanotowanie czasu wykonania.

```
>> parforTest

Elapsed time is 10.102277 seconds.
```

Pętla parfor działa na czterech workerach. Jej wykonanie jest około trzy razy szybsze niż odpowiadającej jej pętli for. Przyspieszenie jest mniejsze niż idealne przyspieszenie dla czterech wątków. Jest to spowodowane równoległym obciążeniem, w tym czasem wymaganym do przesłania danych z klienta do workera i z powrotem. Ten przykład pokazuje dobre przyspieszenie ze stosunkowo małym dodatkowym narzutem i korzyści z konwersji do pętli parfor. Nie wszystkie iteracje pętli mogą zostać przekształcone w szybsze pętle parfor. Jednym z kluczowych wymagań używania pętli parfor jest to, że poszczególne iteracje muszą być od siebie niezależne [11].

5. Porównanie obliczeń na CPU i GPU w Matlabie

W tym punkcie zostanie przedstawiony przykład porównujący obliczenia wykonywane na CPU oraz na GPU w środowisku Matlab. Przedstawione jest wykorzystanie transformaty Fouriera do analizy spektralnej.

W tym przykładzie wykorzystany jest Parallel Computing Toolbox, aby wykonać szybką transformatę Fouriera (FFT) na GPU. Zastosowaniem FFT jest znalezienie składowych częstotliwości sygnału, ukrytego w sygnale czasowym.

Przedstawiony jest przykład wykonujący obliczenia na jednym wątku CPU oraz wykorzystujący wszystkie dostępne wątki procesora. Następnie pokazane jest wykonanie tego zadania na procesorze graficznym GPU. Na końcu przedstawione są wyniki czasu realizacji przykładu i porównanie ich ze sobą [12].

Początkowo zostaje zasymulowany sygnał. Zostają wzięte pod uwagę dane próbki o częstotliwości 1000 Hz i utworzona oś czasu przyjętego przykładu dla dużej liczby próbek. Ponadto utworzone są dwie sinusoidy o częstotliwościach *freq1* i *freq2*. Dla wszystkich trzech sposobów, implementacja pozostaje taka sama, przedstawia ją przykład 3.

Przykład 3. Implementacja utworzenia danych próbki.

```
sampleFreq = 1000;
sampleTime = 1/sampleFreq;
numSamples = 2^23;
freql = 2 * pi * 50;
freq2 = 2 * pi * 120;
```

Sygnał składa się z dwóch komponentów harmonicznych. Pierwszym krokiem jest utworzenie wektora czasu *timeVec* i obliczenie sygnału *signal* jako połączenia dwóch wyżej utworzonych sinusoidalnych częstotliwości.

```
timeVec = (0:numSamples-1) * sampleTime;
signal = sin( freql .* timeVec ) + sin( freq2 .* timeVec );
```

Dodane są losowe zakłócenia dla sygnału.

```
signal = signal + 2 * randn ( size ( timeVec ) );
```

Określenie częstotliwości składowych sygnału umożliwia dyskretna transformata Fouriera w postaci funkcji *Fast Fourier Transform*.

```
transformedSignal = fft( signal );
```

Dodatkowo obliczona jest gęstość widmowa mocy mierząca energię na różnych częstotliwościach.

```
powerSpectrum = transformedSignal .* conj
(transformedSignal) ./ numSamples;
```

W drugim przypadku – przy wykorzystaniu wszystkich wątków, uruchamiana jest równoległa pula w dodatku Parallel Computing Toolbox, pozwalającą na wykorzystanie wszystkich wątków procesora.

Implementacja tego sposobu jest zrealizowana w bloku *spmd* (single program, multiple data). Jego ogólną postać instrukcji prezentuje przykład 4.

Przykład 4. Szkielet bloku *spmd.*

```
spmd
   instrukcje
end
```

W środku bloku są umieszczone instrukcje do wykonania. Zadania są automatycznie rozmieszczone na wszystkie dostępne wątki. Poza tą zmianą, kod nie różni się niczym od tego, który był wykonywany na jednym wątku.

Realizacja zadania dla GPU to przede wszystkim zmiana deklaracji wektora czasu poprzez użycie funkcji *gpuArray* do przesłania danych do procesora GPU w celu dalszego przetwarzania.

```
timeVec = gpuArray( (0:numSamples-1) * sampleTime );
```

W przykładzie 5. jest przedstawiona zawartość skryptu, który jest utworzony do realizacji i prostego porównania wszystkich sprawdzanych rozwiązań wykonywania obliczeń.

Przykład 5. Zawartość wykonanego skryptu.

```
%% DATA INITIALIZATION
sampleFreq = 1000;
sampleTime = 1/sampleFreq;
numSamples = 2^23;
freql   = 2 * pi * 50;
freq2   = 2 * pi * 120;

%% CPU
tic;
timeVec = (0:numSamples-1) * sampleTime;
signal = sin( freql .* timeVec ) + sin( freq2 .* timeVec );
signal = signal + 2 * randn ( size ( timeVec ) );
transformedSignal = fft( signal );
powerSpectrum = transformedSignal .* conj
(transformedSignal) ./ numSamples;
tCPU = toc;
disp(['tCPU = ' num2str(tCPU)]);

%% CPU 4 WORKERS
tic;
spmd
timeVec = (0:numSamples-1) * sampleTime;
```

```
signal = sin( freql .* timeVec ) + sin( freq2 .* timeVec );
signal = signal + 2 * randn ( size ( timeVec ) );
transformedSignal = fft( signal );
powerSpectrum = transformedSignal .* conj
(transformedSignal) ./ numSamples;
end
tCPU_all = toc;
disp(['tCPU_all = ' num2str(tCPU_all)]);

%% GPU
tic;
timeVec = gpuArray( (0:numSamples-1) * sampleTime );
signal  = sin ( freql .* timeVec ) + sin( freq2 .* timeVec );
signal = signal + 2 * randn ( size ( timeVec ) );
transformedSignal = fft ( signal );
powerSpectrum = transformedSignal .* conj
(transformedSignal) ./ numSamples;
tGPU = toc;
disp(['tGPU = ' num2str(tGPU)]);

%% SUMMARY
tDIFF1 = tCPU_all / tCPU;
disp(['tCPU_all/tCPU = ' num2str(tDIFF1)]);
tDIFF2 = tCPU / tGPU;
disp(['tCPU/tGPU = ' num2str(tDIFF2)]);
tDIFF3 = tCPU_all / tGPU;
disp(['tCPU_all/tGPU = ' num2str(tDIFF3)]);
```

Skrypt wykorzystywał polecenia tic oraz toc, dzięki którym zostały zapisane czasy wykonania operacji dla procesora CPU i GPU. Wartości zwrócone po wykonaniu skryptu zostały przedstawione w przykładzie 6.

Przykład 6. Otrzymane wyniki.

```
tCPU = 1.3864
tCPU_all = 4.4697
tGPU = 0.36826
tCPU_all/tCPU = 3.2239
tCPU/tGPU = 3.7648
tCPU_all/tGPU = 12.1372
```

Przeprowadzenie stu symulacji pozwoliło otrzymać średnio około czterokrotnie krótsze czasy wykonania operacji dla procesora graficznego względem sposobu z jednym wątkiem oraz ponad 12-krotne przyspieszenie względem uruchomienia rozwiązania z blokiem *spmd*. które jest ponad trzykrotnie wolniejsze od pracującego na jednym wątku.

6. Wnioski

Prowadząc badania na czterordzeniowym procesorze, w próbie jego pełnego wykorzystania – zastosowaniu pętli *parfor* odnotowano około trzykrotnie lepsze wyniki czasowe przeprowadzonych działań. Wyniki nie były idealnie czterokrotnie lepsze, czego można się było teoretycznie spodziewać. Przygotowanie bloku równoległego, przesyłanie danych z pamięci RAM do pamięci karty graficznej i synchronizacja wątków wprowadza dodatkowy czas potrzebny do wykonania części równoległej kodu.

Rozwiązanie z blokiem *spmd* jest najwolniejsze dla badanego przykładu, ponieważ więcej czasu zajmuje komunikacja między wątkami niż same obliczenia.

Procesory graficzne nie zastępują architektury CPU. Są raczej potężnymi akceleratorami dla istniejącej infrastruktury. Procesory przyspieszane przez GPU odciążają część aplikacji intensywnie wykorzystującą obliczenia na GPU, podczas gdy pozostała część kodu nadal działa na procesorze. Z perspektywy użytkownika aplikacje działają znacznie szybciej. Podczas gdy przetwarzanie ogólnego przeznaczenia nadal jest domeną CPU, procesory graficzne są szkieletem sprzętowym prawie wszystkich intensywnych aplikacji obliczeniowych [13].

Jednak procesory są bardziej elastyczne niż GPU, mają większy zestaw instrukcji, dzięki czemu mogą wykonywać szerszy zakres zadań. CPU pracuje również z wyższymi maksymalnymi prędkościami zegara i jest w stanie zarządzać wejściami i wyjściami wszystkich komponentów komputera.

Literatura

[1] MATLAB Product Description - MathWorks Documentation, *https://www.mathworks.com/help/matlab/learn_matlab/product-description.html* [01.08.2018]

[2] K. Banasiak, Algorytmizacja i programowanie w MATLABIE, Wydawnictwo BTC, 2017.

[3] Parallel Computing Toolbox - Documentation, *https://www.mathworks.com/help/distcomp/* [01.08.2018].

[4] J. W. Sut, Y. Kim, MATLAB and Parallel Computing Toolbox, 2014, 99-125.

[5] Obliczenia równoległe w środowisku Matlab - MathWorks Video and Webinars, *https://www.mathworks.com/videos/parallel-computing-in-matlab-116769.html* [01.08.2018].

[6] B. Mrozek, „Obliczenia równoległe w Matlab-ie," Pomiary Automatyka Robotyka, tom R. 15, nr 2, pp. 285-294, 2011.

[7] I. Azzini, R. Muresano, M. Ratto, Dragonfly: A multi-platform parallel toolbox for MATLAB/Octave, 2018, 21-42.

[8] What is GPU computing?, *https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx* [01.08.2018].

[9] M. Sourouri, J. Langguth, F. Spiga, S. B. Baden. X. Cai, CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters, 2015, 17-26.

[10] Using parfor-loop - MathWorks Documentation,, *https://www.mathworks.com/help/distcomp/interactively-run-a-loop-in-parallel.html#responsive_offcanvas* [01.08.2018].

[11] What Is Parallel Computing?, *https://www.mathworks.com/help/distcomp/what-is-parallel-computing.html* [01.08.2018].

[12] Using FFT on the GPU for Spectral Analysis MathWorks – Documentation, *https://www.mathworks.com/help/distcomp/examples/using-fft-on-the-gpu-for-spectral-analysis.html* [01.08.2018].

[13] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Welling, M. Kohler, Preconditioned Krylov solvers on GPUs, 2017, 32-44.