WSTĘP DO PROJEKTU

Celem projektu jest **porównanie** czasów dla operacji mnożenia macierzy na **CPU** i **GPU**. Czas obliczeń na GPU mierzony jest wraz z transferem danych z CPU na GPU i z powrotem.
Język programowania użyty w projekcie - **Python**.

Rodzaje obliczeń:

- Naiwne mnożenie macierzy na słabym i mocnym CPU
- Mnożenie macierzy na słabym i mocnym CPU przy użyciu biblioteki NumPy
- Naiwne mnożenie macierzy na GPU przy użyciu podstawowego jądra
- Mnożenie macierzy metodą kafelkowania na GPU przy użyciu jądra z wykorzystaniem pamięci dzielonej dla macierzy rozmiaru 2^n
- Mnożenie macierzy metodą kafelkowania na GPU przy użyciu jądra z wykorzystaniem pamięci dzielonej dla macierzy dowolnego rozmiaru
- Mnożenie macierzy na GPU przy użyciu biblioteki Reikna
- Mnożenie macierzy na GPU przy użyciu biblioteki scikit-cuda - cublas
- Mnożenie macierzy na GPU przy użyciu biblioteki scikit-cuda - linalg

UŻYTE TECHNOLOGIE

NumPy - biblioteka języka programowania Python, dodająca obsługę dużych, wielowymiarowych tablic i macierzy, a także duży zbiór funkcji matematycznych wysokiego poziomu do obsługi tych tablic. Dlaczego warto jej używać?

1) Tablice w NumPy są kolekcjami o podobnych typach, gęsto upakowanych w pamięci.
2) Listy w Pythonie posiadają różne typy, co powoduje wiele dodatkowych ograniczeń i warunków przy wykonywaniu obliczeń na nich.
3) NumPy jest w stanie rozdzielić zadanie na podzadania i wykonywać je równolegle.
4) Funkcje w NumPy są zaimplementowane w C, co również zwiększa ich przewagę nad Pythonem.

PyCuda - biblioteka umożliwiająca łatwy, “Pythonowy” dostęp do Nvidia’s CUDA computation API. Dlaczego ją wybrałem?

1) Automatyczne zwolnienie pamięci, kiedy obiekt znika. Zmniejszenie możliwości wycieków pamięci.
2) Łatwe w użytkowaniu struktury, np. pycuda.gpuarray.GPUArray, które pozwalają używać ich w prostszy sposób niż w środowisku Nvidia’s CUDA opartym na języku C.
3) Automatyczne sprawdzanie błędów przetłumaczone na Pythonowe wyjątki.
4) Wyeliminowanie problemów związanych z uruchamianiem jądra GPU.

SCIKIT-CUDA - biblioteka, która dostarcza wiele interfejsów funkcji CUDA do Pythona takich jak CUBLAS, CUFFT, CUSOLVER. W projekcie zostały wykorzystane 2 moduły:

1) LINALG - Linear Algebra Routines and Classes
2) CUBLAS - Basic Linear Algebra Subprograms Routines

Reikna - biblioteka do GPGPU w Pythonie. Zawiera wiele różnych algorytmów GPU (np. mnożenie i losowanie macierzy) zbudowanych na podstawie PyCuda i PyOpenCL. Zawiera oddzielenie etapu przygotowania i wykonania danego zadania, co wpływa na maksymalizację wydajności etapu wykonania kosztem etapu przygotowania.

SPRZĘT

Obliczenia zostały przeprowadzone na dwóch procesorach i jednej karcie graficznej.

Procesor 1
Model name: Intel(R) Core(TM) i7-5600U @ 2.60GHz
CPU(s): 4
Thread(s) per core: 2
Core(s) per socket: 2
Socket(s): 1
CPU max MHz: 3200
L1 cache: 128 KB
L2 cache: 512 KB
L3 cache: 4096 KB

Procesor 2
Model name: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
CPU(s): 40
Thread(s) per core: 2
Core(s) per socket: 10
Socket(s): 2
CPU max MHz: 3100
L1 cache: 32 KB
L2 cache: 256 KB
L3 cache: 25600 KB

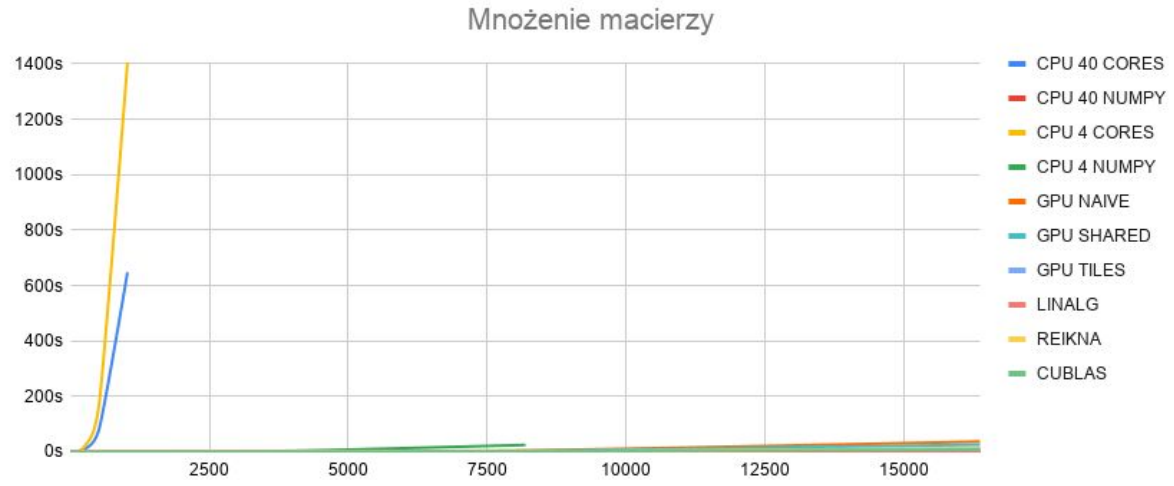
Karta graficzna
NVIDIA GeForce GTX 1070 Ti 8GB

WYKRESY

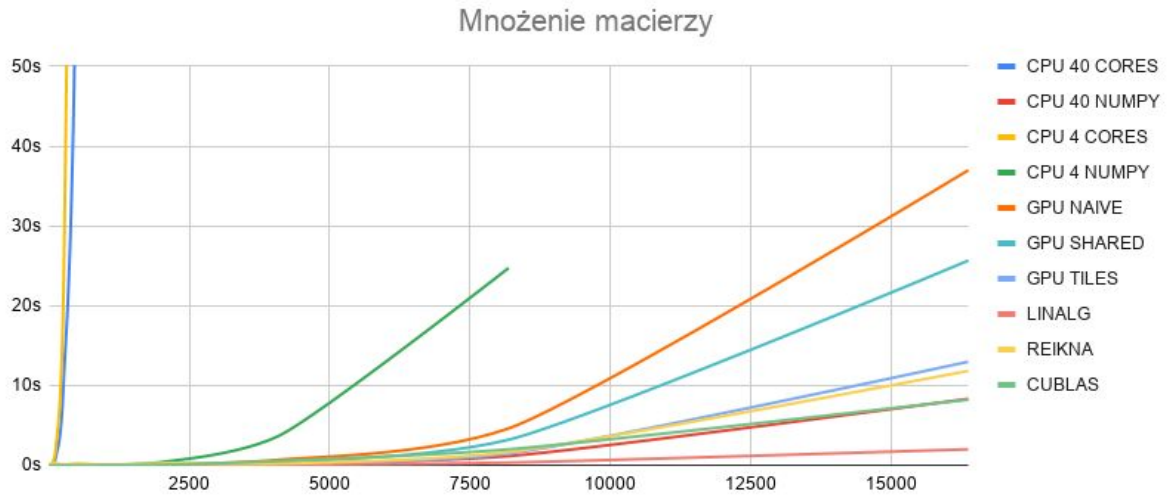
Wykresy przedstawiają czas wykonania operacji mnożenia macierzy w oparciu o rozmiar macierzy i rodzaj algorytmu. Wykresy niedokończone oznaczają zbyt długi czas wykonania (> 10 min).
Oś pozioma odpowiada za rozmiary macierzy, gdzie liczba N odpowiada rozmiarowi NxN macierzy wejściowych (np. 200 oznacza macierz 200x200).
Oś pionowa odpowiada za czas wykonywania operacji w sekundach.
Legenda przedstawia wykorzystane algorytmy, gdzie:

- CPU 40 CORES - mnożenie macierzy naiwnie na lepszym procesorze
- CPU 40 NUMPY - mnożenie macierzy z użyciem NumPy na lepszym procesorze
- CPU 4 CORES - mnożenie macierzy naiwnie na słabszym procesorze
- CPU 4 NUMPY - mnożenie macierzy z użyciem NumPy na słabszym procesorze
- GPU NAIVE - naiwne mnożenie macierzy na GPU przy użyciu podstawowego jądra
- GPU SHARED - mnożenie macierzy metodą kafelkowania na GPU przy użyciu jądra z wykorzystaniem pamięci dzielonej dla macierzy rozmiaru 2^n
- GPU TILES - mnożenie macierzy metodą kafelkowania na GPU przy użyciu jądra z wykorzystaniem pamięci dzielonej dla macierzy dowolnego rozmiaru
- LINALG - mnożenie macierzy na GPU przy użyciu biblioteki scikit-cuda - linalg
- REIKNA - mnożenie macierzy na GPU przy użyciu biblioteki Reikna
- CUBLAS - mnożenie macierzy na GPU przy użyciu biblioteki scikit-cuda - cublas

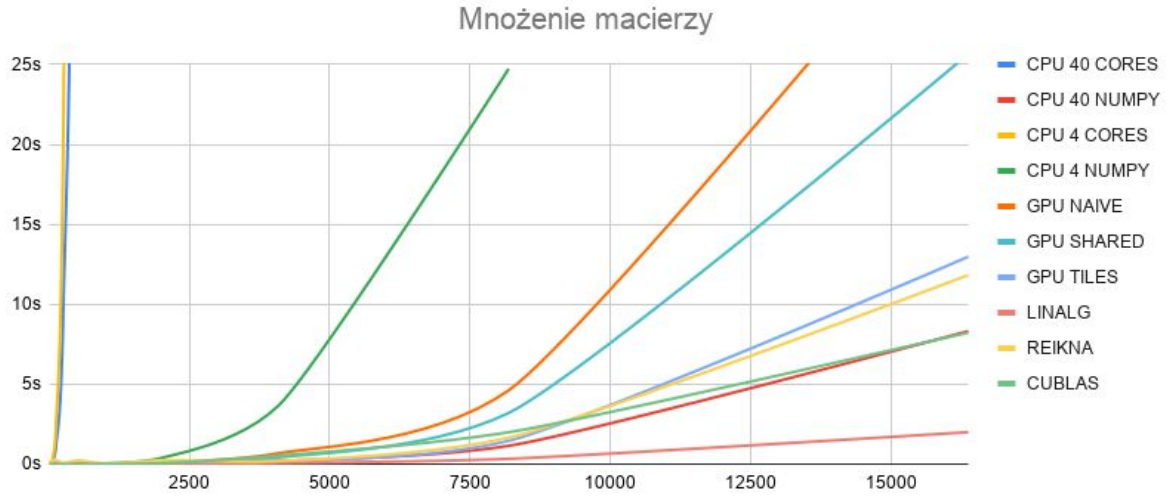
Widoczna bardzo słaba efektywność algorytmów naiwnych na CPU



Porównanie algorytmów w ograniczeniu do 50s.



Porównanie algorytmów w ograniczeniu do 25s.



Porównanie tabelaryczne

Rozmiar \ Algorytm	CPU 40 CORES	CPU 40 NUMPY	CPU 4 CORES	CPU 4 NUMPY	GPU NAIVE	GPU SHARED	GPU TILES	LINALG	REIKNA	CUBLAS
4	0,00031	0,000054	0,000531	0,000384	0,001713	0,000599	0,000571	0,000907	0,212057	0,001888
16	0,006676	0,00005	0,006003	0,000821	0,001235	0,000337	0,000323	0,000519	0,017617	0,001149
64	0,205642	0,000068	0,352022	0,000999	0,000909	0,00029	0,000275	0,000689	0,231303	0,002898
128	1,29822	0,001691	2,700997	0,001005	0,002421	0,000644	0,000649	0,001044	0,240901	0,001754
256	10,0316368	0,002279	21,335023	0,00098	0,008841	0,000768	0,000756	0,001805	0,071204	0,003039
512	81,25037	0,003475	170,009022	0,006977	0,013266	0,002182	0,00154	0,002356	0,230053	0,004591
1024	647,821219	0,016128	1405,384736	0,048971	0,023248	0,009417	0,004905	0,00402	0,04091	0,011292
2048	-	0,037607	-	0,422994	0,101122	0,063357	0,024603	0,010925	0,245395	0,055942
4096	-	0,200463	-	3,681998	0,681695	0,421496	0,168641	0,067117	0,200677	0,531674
8192	-	1,145355	-	24,733997	4,616684	3,201822	1,460323	0,332861	1,652045	1,994817
16384	-	8,321525	-	-	36,985566	25,656714	12,977047	1,999397	11,828416	8,227306

WNIOSKI

1) Algorytmy naiwne na CPU są diametralnie mniej efektywne od jakichkolwiek innych.
2) Najwydajniejszym algorytmem na GPU okazał się algorytm z biblioteki scikit-cuda, moduł linalg.
3) NumPy na odpowiednio dobrym procesorze potrafi na tyle dobrze zrównoleglić mnożenie macierzy, że czas wykonania operacji jest porównywalny z wykonaniem mnożenia na GPU przy modułu CUBLAS z biblioteki scikit-cuda.
4) Algorytmy pochodzące z bibliotek okazały się efektywniejsze od jąder GPU.
5) Ilość pamięci RAM na urządzeniach GPU jest zazwyczaj mniejsza niż na CPU, co nie pozwala na operowanie danymi odpowiednio dużych rozmiarów.