Adam Borowski, 9.06.2015

Przetwarzanie równolegle CUDA

Sprawozdanie z projektu

Testowany sprzęt

Cecha	0: GeForce GTX 480	1: GeForce GTS 450
machine / device ID	des09 / 0	des09 / 1
CUDA Driver Version / Runtime Version	6.0	6.0
CUDA Capability Major/Minor version number	2.0	**2.1**
Total amount of global memory	**1536 MB**	1024MB
Number of multiprocessors	**15**	4
Number of CUDA Cores/MP	32	**48**
Number of CUDA Cores	**480**	192
GPU Clock rate	1401	**1622**
Memory Clock rate	**1848**	1804
L2 Cache size	**786432**	262144
Total amount of constant memory	65536	65536
Total amount of shared memory per block	49152	49152
Total number of registers available per block	32768	32768
warp size	32	32
Max number of threads per block	1024	1024
Max dimension size of a thread block	(1024, 1024, 64)	(1024, 1024, 64)
Max dimension size of a grid size	(65535, 65535, 65535)	(65535, 65535, 65535)

Aspekty testowania

- rozmiar danych wejściowych (2000,20000, 200000,2000000 elementów do przetworzenia)
- dostępne zasoby
  - liczba wątków typu B (2,4,8,16)
  - liczba bloków (1,10,100,300,400)
- dostępne urządzenia
  - karta 0: GeForce GTX 480
  - karta 1: GeForce GTS 450
- wykorzystane optymalizacje
  - skompilowany kod niezoptymalizowany
  - skompilowany kod zoptymalizowany

Liczba kombinacji aspektów testowania: **4x4x5x2x2=320**
Liczba uruchomień pojedynczego testu (każda kombinacja uruchamiana 5 razy, z czego do raportu obliczona zostanie średnia arytmetyczna dla każdej kombinacji): **320x5 = 1600**

Testowanie automatyczne (Python)

Zaimplementowano mechanizm licznika z przepełnieniami aby wygenerować wszystkie kombinacje parametrów uruchomienia. W każdej iteracji wywoływany jest program i strumień wyjściowy przetworzony jest przez program **grep** w celu znalezienia wyniku pomiaru. Tester uruchamia się dwa razy, po jednym dla programu zoptymalizowanego i bez optymalizacji.

```
###### registers.py ######
class CyclicRegister:
    def __init__(self, array):
        self.array = array
        self.count = len(array)
        self.current = 0
        self.overflowOccured = False
    def get(self):
        return self.array[self.current]
    def next(self):
        self.current += 1
        if self.current == self.count:
            self.current = 0
            self.overflowOccured = True
            return True
        return False
    def isOverflowOccured(self):
        return self.overflowOccured


class OverflowManager:
    def __init__(self, registers):
        self.registers = registers
    def next(self):
        for register in self.registers:
            overflowOccured = register.next()
            if not overflowOccured:
                break
    def isOverflowOccured(self):
        return self.registers[-1].isOverflowOccured()
    def getArray(self):
        return [register.get() for register in self.registers]
```

```
###### tester.py ######
import os
from registers import *

criteria = {
    "inputSize": [2000, 20000, 200000, 2000000],
    "numBThreads": [2, 4, 8, 16],
    "numBlocks": [1, 10, 100, 300, 400],
    "device": [0, 1]
}

mgr = OverflowManager([
    CyclicRegister(criteria["inputSize"]),
    CyclicRegister(criteria["numBThreads"]),
    CyclicRegister(criteria["numBlocks"]),
    CyclicRegister(criteria["device"])
])
while True:
    current = mgr.getArray()
    # <test execution>
    conf = {
        "inputSize": current[0],
        "numBThreads": current[1],
        "numBlocks": current[2],
        "device": current[3]
    }

    cmd = "[./CUDA_EXECUTABLE] 128 " \
          "{0[numBThreads]} 128 {0[numBlocks]} {0[device]}
{0[inputSize]}" \
          " | grep -oPe \"(?<=process time: )(.*)(?=
ms)\"".format(conf)
    results = []
    numTests = 5
    avg = 0
    # gather time results and calculate average
    for i in range(0, numTests):
        time = float(os.popen(cmd).read())
        avg += time
        results.append(str(time))
    avg /= numTests

    outputLine = [str(c) for c in current] + results +
[str(avg)]

    print "\t".join(outputLine)

    # </test execution>
    mgr.next()
    if mgr.isOverflowOccured():
        break
```

Przetwarzanie wyników testów i przygotowanie do prezentacji

Testy automatyczne wygenerowały pliki TSV, które zostały zaimportowane do arkusza Google. Arkusz dla każdego przypadku testowania posiada odpowiedni wiersz, a w kolumnach definiowane są poszczególne parametry uruchomienia (np. rozmiar danych, nr urządzenia) oraz poszczególne wyniki pomiarów czasu tego przypadku testowego jak i uśredniony wynik testu:

input size	num B threads	num blocks	dev id	optimization	test1	test2	test3	test4	test5	test avg
...	...	...	...	...	...	...	...	...	...	...
2000	2	1	0	0	4.982144	4.978368	5.031936	4.975712	4.976032	4.9888384
20000	2	1	0	0	45.466686	45.51939	45.464127	45.465218	45.465378	45.4761598
200000	2	1	0	0	451.795563	451.847382	451.849792	451.877777	451.849396	451.843982
2000000	2	1	0	0	4515.958984	4515.986816	4515.991211	4515.953125	4515.991211	4515.976269
...	...	...	...	...	...	...	...	...	...	...

Z powyższych danych generowane są różne tabele przestawne (Pivot Table), które pokazują zależności pomiaru czasu od dwóch wybranych parametrów uruchomienia przy arbitralnym ustawieniu pozostałych parametrów. Kolumna/wiersz “Grand Total” pokazuje uśrednione wartości dla danego wiersza/kolumny. Ponadto, w celu prezentacji wykresów 3D wykorzystano program Octave. Przykładowy kod renderujący wykres 3D:

```
x=[1 10 100     300 400];
y=[2000 20000 200000 2000000];
v=[1.113        0.614   0.613   1.310   1.662
0.986   0.546   1.532   4.023   5.270
6.133   1.112   0.845   1.657   2.010
5.375   1.638   2.649   5.143   6.389
58.379  6.127   4.814   5.273   5.678
51.219  15.284  15.179  17.146  18.394
581.630 58.409  41.388  40.897  41.507
510.511 153.188 129.860 133.345 135.090];

not_optim = v(1:2:end,:);
optim = v(2:2:end, :);
xx=x';
yy=y';

hold on;
hSurface = mesh(xx, yy,not_optim);
set(hSurface,'EdgeColor',[0.4, 1, 0.2], 'FaceAlpha', 0);
hSurface=mesh(xx, yy,optim);
set(hSurface,'EdgeColor',[1, 0.4, 0.2], 'FaceAlpha', 0);
legend("dev 0","dev 1");
title("czas w zależności od liczby bloków i rozmiaru danych");
xlabel('liczba bloków')
ylabel('rozmiar danych')
zlabel('czas [ms]')
grid on;
box off;
hold off;
view(40,15);
```

Prezentacja wyników pomiarów czasów

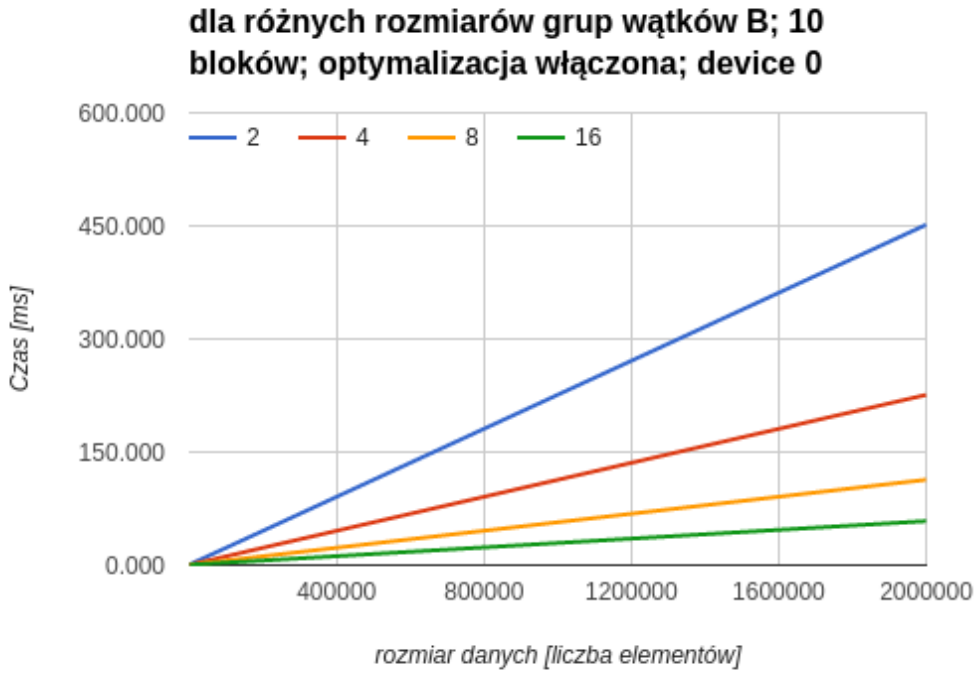
W tym rozdziale zostaną pokazane specjalnie przygotowane tabele przestawne (Pivot tables) jako źródło analizy, wykresy na nich bazujące i komentarze dotyczące analizy, spostrzeżeń i obserwacji.

Zależność od rozmiarów danych wejściowych

Dane (10 bloków, optymalizacja włączona, device 0)

dane▼liczba wątków B ►	2	4	8	16	Grand Total
2000	0.588	0.592	0.590	0.614	0.596
20000	4.971	2.538	1.565	1.112	2.546
200000	45.424	22.994	11.788	6.127	21.583
2000000	451.538	225.853	113.276	58.409	212.269
Grand Total	125.630	62.994	31.805	16.566	59.249

Wykres



Interpretacja

Gdy danych jest na tyle dużo, że bloki są zmuszone przetwarzać sekwencyjnie mniejsze porcje danych, widać, że czas obliczeń jest wprost proporcjonalny do ilości danych wejściowych. Widać również, że czas obliczeń jest odrotnie proporcjonalny do liczby wątków grupy B, czyli wątków aktywnie liczących w ramach jednego bloku.

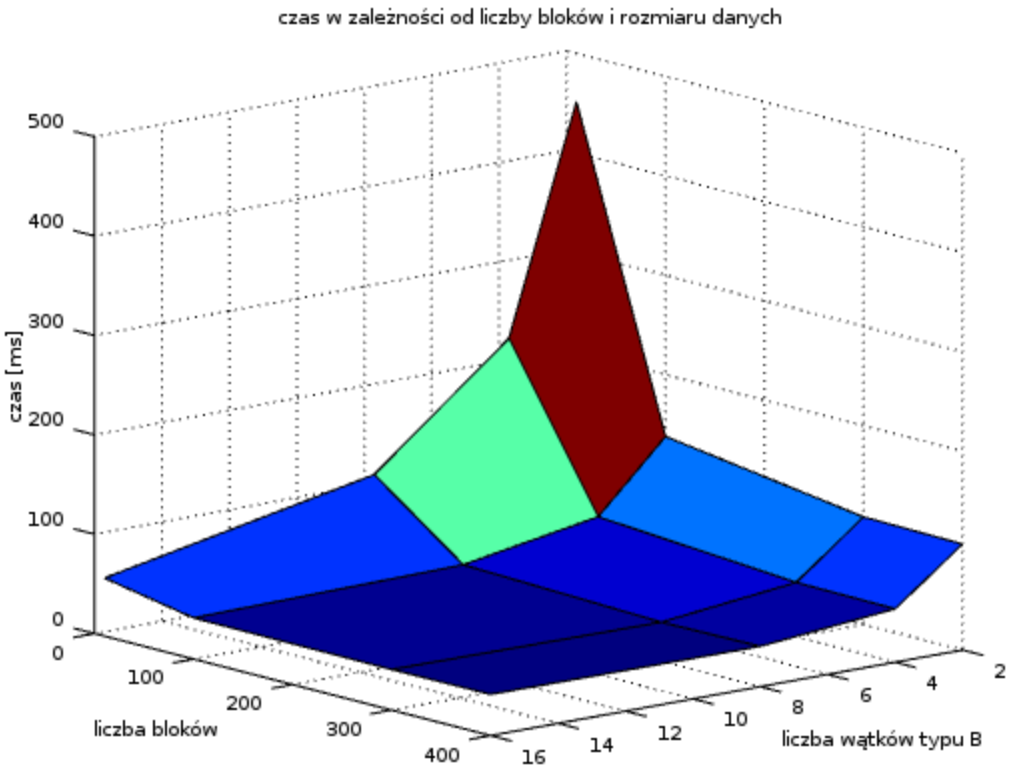
Zależność od dostępnych zasobów

Poprzez dostępne zasoby rozumie się liczbę wykorzystanych bloków i liczbę wątków w grupie B (aktywnie liczących) w bloku. Aby najlepiej pokazać tę zależność, zaprezentowano wykres 3D.

Dane (2mln danych, włączona optymalizacja, device 0)

l. bloków ▼ l. wątków typu B ►	2	4	8	16	Grand Total
1	4511.264	2255.661	1128.521	581.630	2119.269
10	451.538	225.853	113.276	58.409	212.269
100	137.834	69.333	45.664	41.388	73.555
300	108.184	54.565	39.200	40.897	60.712
400	106.540	53.719	40.592	41.507	60.590
Grand Total	1063.072	531.826	273.450	152.766	505.279

Wykres (dla liczby bloków 10 - 400)



Interpretacja

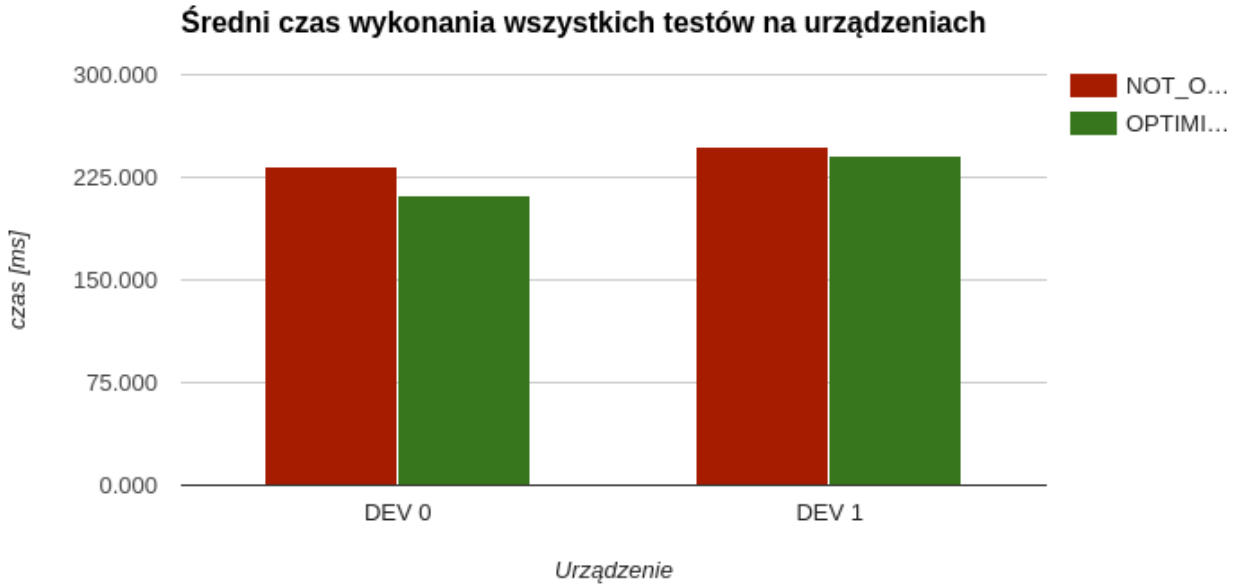
Widać, że dwuargumentowa funkcja czasu wykonania od zasobów ma swoje maksimum w przypadku 10 bloków i 2 wątków liczących, a maksumum w przyadku 400 bloków i 16 wątków liczących. Zawsze należy pamiętać o ograniczeniu wielkości pamięci cache w bloku i maksymalnego rozmiaru grida, dlatego oba czynniki wpływające na wykorzystanie zasobów mają granicę, której przekroczenie spowoduje błąd CUDA.

Zależność od urządzenia

Dane (2mln danych, 10 bloków)

	NOT_OPTIM	OPTIMIZED	Grand Total
DEV 0	232.450	212.269	222.360
DEV 1	247.054	241.051	244.053

Wykres



Interpretacja

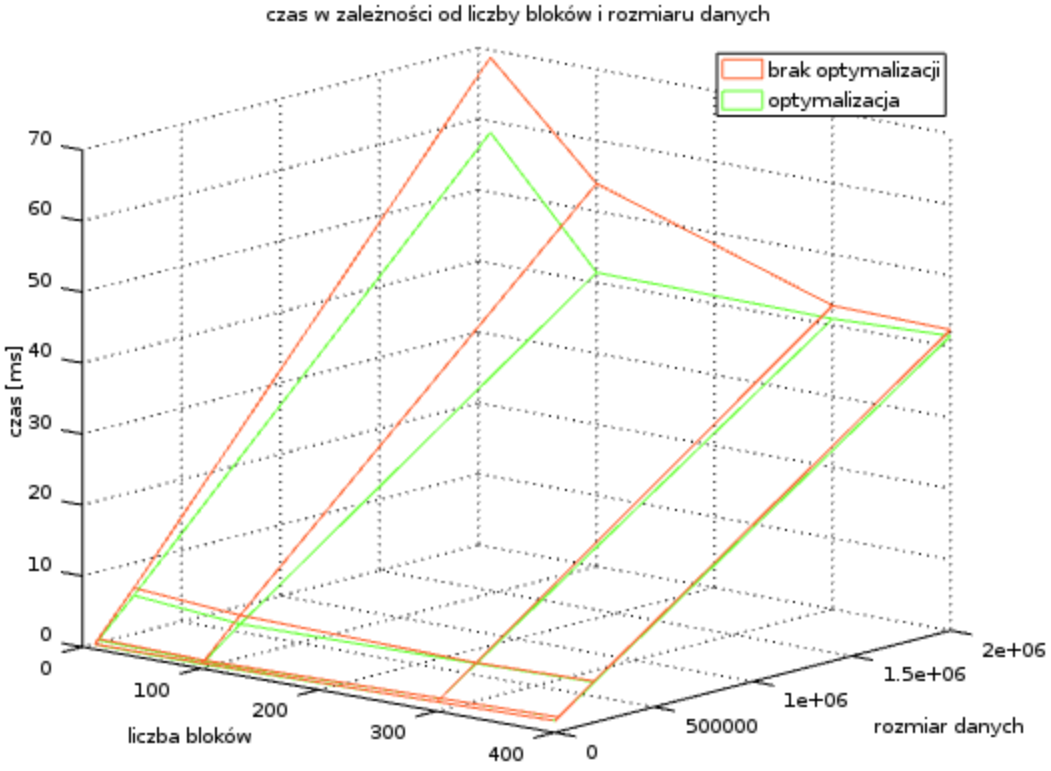
Na powyższym wykresie jak i przy okazji dalszych analiz widać, że wprowadzone optymalizacje nie polepszyły znacznie wydajności aplikacji.

Zależność od rozmiarów danych, dostępnych zasobów i wprowadzonych optymalizacji

Dane

rozmiar ▼	liczba bloków ►	1	10	100	300	400	Grand Total
2000	**NOT_OPTIM**	1.226	0.697	0.761	1.311	1.662	1.131
	OPTIMIZED	1.113	0.614	0.613	1.310	1.662	1.062
20000	**NOT_OPTIM**	6.793	1.302	0.987	1.662	2.027	2.554
	OPTIMIZED	6.133	1.112	0.845	1.657	2.010	2.351
200000	**NOT_OPTIM**	64.716	7.179	6.041	5.396	5.798	17.826
	OPTIMIZED	58.379	6.127	4.814	5.273	5.678	16.054
2000000	**NOT_OPTIM**	645.684	68.966	53.915	42.754	42.376	170.739
	OPTIMIZED	581.630	58.409	41.388	40.897	41.507	152.766

Wykres (dla liczby bloków 10 - 400)



Interpretacja

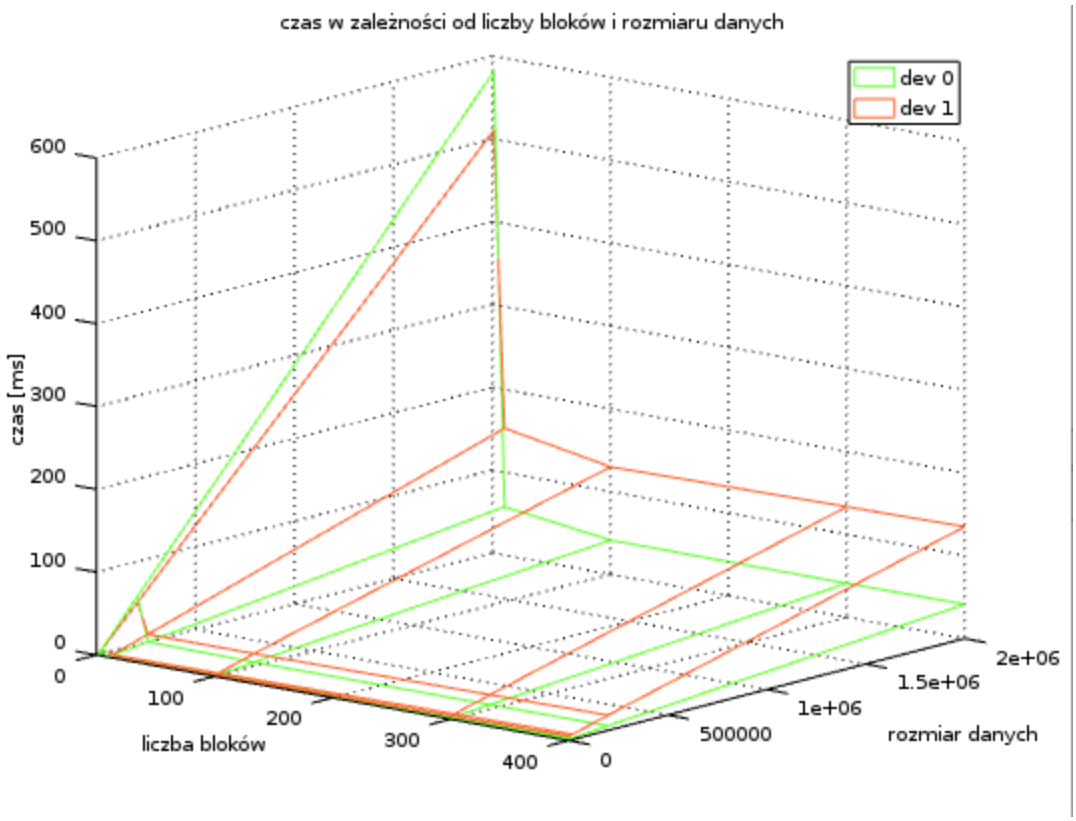
Widać wyraźny skok wydajności z liczby bloków równej 1 na 10 i potem na 100. W przypadku używania tylko jednego bloku scheduler nie mógł wykorzystać potencjału GPU gdyż każda porcja danych przetwarzana jest w praktyce sekwencyjnie przez blok. Z kolei wprowadzenie większej liczby bloków niż 300 przestaje być opłacalne, ponieważ GPU ma ograniczone możliwości przetwarzania bloków równoczesnie, scheduler serializuje wątki oczekujących bloków. Z kolei opłacalność optymalizacji maleje wraz ze wzrostem liczby bloków.

Zależność od rozmiarów danych, dostępnych zasobów i urządzenia

Dane

rozmiar ▼	liczba bloków ►	1	10	100	300	400	Grand Total
2000	**DEV 0**	1.113	0.614	0.613	1.310	1.662	1.062
	DEV 1	0.986	0.546	1.532	4.023	5.270	2.471
20000	**DEV 0**	6.133	1.112	0.845	1.657	2.010	2.351
	DEV 1	5.375	1.638	2.649	5.143	6.389	4.239
200000	**DEV 0**	58.379	6.127	4.814	5.273	5.678	16.054
	DEV 1	51.219	15.284	15.179	17.146	18.394	23.444
2000000	**DEV 0**	581.630	58.409	41.388	40.897	41.507	152.766
	DEV 1	510.511	153.188	129.860	133.345	135.090	212.399

Wykres



Interpretacja

W ekstramalnym przypadku (1 blok) szybszy GTX480 działa wolniej niż wolniejszy GTS450. Warto jednak zauważyć, że GTS450 niektóre parametry ma lepsze:

Cecha	0: GeForce GTX 480	1: GeForce GTS 450
Number of CUDA Cores/MP	32	48
GPU Clock rate	1401	1622

W normalnych przypadkach, gdy jest używanych więcej bloków, GTX480 znacznie wygrywa konkurencję z GTS450.

Drugim ciekawym spostrzeżeniem jest to, że czas obliczeń jest liniowo zależny od rozmiaru danych, gdyż testy obejmują na tyle duże zbiory danych, że w każdym przypadku bloki są równomiernie obłożone pracą do wykonania.