Łukasz Nozdrzykowski, Magdalena Nozdrzykowska

Modele szacowania czasów wykonywania się pętli programowych z wykorzystaniem programowania równoległego na CPU oraz z wykorzystaniem obliczeń na GPU przy użyciu OpenCl

JEL: L97 10.24136/atest.2018.501
Data zgłoszenia: 19.11.2018 Data akceptacji: 15.12.2018

W artykule autorzy przedstawiają modele szacowania czasów wykonywania się pętli programowych zgodnych z modelem FAN nieposiadającym zależności lub posiadającym zależności, ale tylko w ciele pętli, które wykonywane mogą być przez centralne jednostki obliczeniowe CPU jak i multiprocesory strumieniowe zwane rdzeniami kart graficznych GPU. Zaprezentowane w niniejszym artykule modele szacowania czasów wykonywania tych pętli pozwolą na określanie tego, czy obliczenia w zastanym środowisku obliczeniowym warto wykonywać z użyciem posiadanego procesora CPU czy korzystniejsze będzie wykorzystanie do obliczeń posiadanej, często nowoczesnej, karty graficznej z wydajną jednostką GPU i bardzo szybką pamięcią stosowaną we współczesnych kartach graficznych. Wraz z zaprezentowanymi modelami przedstawiono także testy potwierdzające poprawność opracowanych modeli szacowania czasu. Celem powstania tych modeli jest dostarczenie metod przyspieszania działania aplikacji realizujących różne zadania, w tym zadania transportowe, takie jak przyspieszone przeszukiwanie rozwiązań, przeszukiwanie ścieżek w grafach, czy przyspieszanie algorytmów przetwarzania obrazów w systemach wizyjnych pojazdów autonomicznych i semiautonomicznych, przy czym modele te pozwalają na zbudowanie systemu automatycznego rozdzielania zadań pomiędzy CPU i GPU przy zmienności zasobów obliczeniowych.

Słowa kluczowe: pętle programowe, szacowanie czasu wykonania pętli, programowanie CPU i GPGPU

Wstęp

Poszukiwanie nowych źródeł mocy obliczeniowych staje się szczególnie ważne tam, gdzie istnieje potrzeba przetwarzania dużych ilości danych w krótkim czasie. Przykładem takiego systemu jest system rozpoznawania obrazów jednostek autonomicznych i semiautonomicznych, jak statki morskie, których rozwój jest aktualnym tematem branży morskiej [1]. Systemy rozpoznawania obrazów muszą w trybie praktycznie rzeczywistym dokonywać przetwarzania, segmentowania i identyfikacji obrazów cyfrowych na potrzeby bezpiecznej żeglugi. Rozwijany w Polsce system AVAL (Autonomous Vessel with an Air Look) [2] pod kierownictwem Politechniki Białostockiej zakłada wykorzystanie kamery drona do wykrywanie obiektów „niewidzianych" przez tradycyjne systemy statków (AIS, ARPA) w celu dostarczenia systemom autonomicznego sterowania statkiem informacji o zagrożeniach na trajektorii statku. Celem jest odpowiednio szybkie wykrywanie obiektów z dużym wyprzedzeniem w sposób zapewniający bezpieczną żeglugę. Taki system ostrzega przed tym czego nie widzi radar czy system automatycznej identyfikacji statków. Pozwala wykrywać duże zwierzęta morskie, pływające zgubione kontenery, góry lodowe oraz małe jednostki. Wysłany dron pomocniczy pozwala na realizację dokładnego rozpoznania obiektu niebezpiecznego, w tym jego rodzaju, wymiarów, kursu i prędkości, co w założeniu pozwolić ma na skuteczne ominięcie zagrożenia. Takich systemów jest wiele i to niekoniecznie w transporcie morskim, ale w tym przypadku możliwe jest wykorzystywanie dużych mocy obliczeniowych, w tym pochodzących nie tylko z klasycznego CPU, ale i ze współczesnych kart graficznych. Jest to szczególnie istotne przy przyspieszaniu obliczeń w algorytmach, które wykorzystują iteracje pętlami programowymi, a zwłaszcza przy wykorzystaniu pętli programowych bez zależności lub z zależnościami w ciele pętli. Takim algorytmem jest chociażby podstawowy w segmentacji obrazu algorytm binaryzacji progowej obrazu czy algorytmy filtracji splotowej. Pojawia się jednak pytanie w jaki sposób rozdzielić zadania obliczeniowe pomiędzy CPU i GPU przy zmiennym ich obciążeniu i rodzaju posiadanych zasobów obliczeniowych. W niniejszym artykule zaproponowano metody szacowania czasów wykonywania się pętli programowych, które mogą być realizowanych zarówno na CPU jaki GPU, w celu automatyzacji procesu rozdzielania zadań.

Ostatnie lata rozwoju przemysłu komputerowego zaowocował skierowaniem się w kierunku technologii przetwarzania równoległego. Powodem tego stanu rzeczy był głównie brak dalszego zwiększania wydajności procesorów poprzez proste zwiększanie ich częstotliwości taktowania. Producenci układów obliczeniowych, w wyniku zbliżenia się do granic możliwości ulepszania technik produkcji układów scalonych, zmuszeni zostali do skierowania swoich produktów w kierunku wielordzeniowych jednostek przetwarzania danych.

Kolejnym etapem był rozwój jednostek przetwarzania grafiki, czyli procesorów GPU. Poprzez odejście w ich architekturze od jednostek wykonujących obliczenia na shaderach wierzchołków oraz pikseli i przejście na połączony potok przetwarzania stało się możliwe wykonywanie obliczeń ogólnego przeznaczenia. Zastosowano w nich szereg jednostek ALU zbudowanych zgodnie z normą IEEE co przetwarzania arytmetyki liczb zmiennoprzecinkowych pojedynczej precyzji i wbudowano zestawy instrukcji umożliwiające na wykonywanie obliczeń ogólnych. Jednocześnie umożliwiono realizację swobodnego dostępu do pamięci, a także wprowadzono mechanizm pamięci współdzielonej. Następnie pojawiły się narzędzia programistyczne do wykonywania obliczeń z wykorzystaniem GPGPU na architekturze NVIDIA Cuda czy na AMD ATI Stream. Obok architektury NVIDIA Cuda pojawił się także framework OpenCL pozwalający na pisanie programów działających na heterogenicznych platformach zbudowanych z różnych jednostek obliczeniowych [3].

W niniejszym artykule autorzy podejmują próbę wyznaczenia modeli szacujących czas wykonywania się pętli programowych w momencie wykorzystania do ich uruchamiania procesorów CPU zbudowanej w architekturze wielordzeniowej oraz czasu wykonywania tych samych pętli przez wielordzeniowe układy graficzne GPU, które programowane są zgodnie z podejściem GPGPU. Celem powstania tych modeli jest umożliwienia powstawania narzędzi automatycznego decydowania o wykorzystaniu CPU lub GPU do wykonywania obliczeń ogólnych przy posiadanych aktualnie zasobach sprzętowych. W artykule omówione zostaną modele szacowania czasu wykonywania

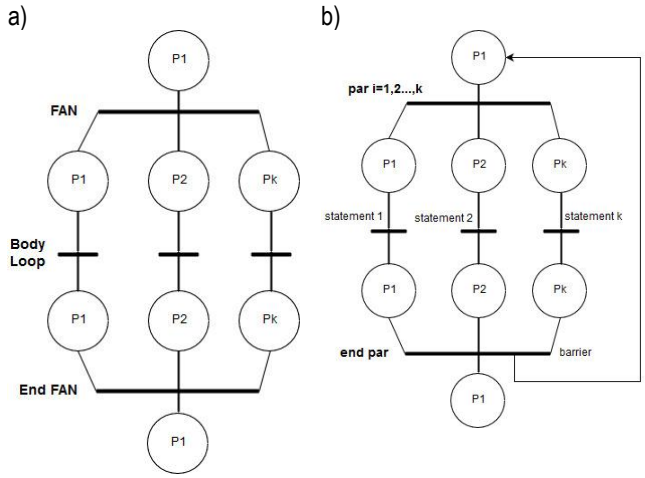
się pętli, które mogą być wykonywane na CPU lub na GPU, a zatem zgodne z podejściem SPMD (Single Program Multiple Data). Pozwoli to określać, kiedy daną pętlę wykonać w sposób równoległy na procesorze CPU, a kiedy lepszym rozwiązaniem będzie przesłanie obliczeń do GPU.

1.Schematy wykonywania się pętli programowych na wielordzeniowych procesorach CPU

Zrównoleglanie kodu źródłowego ma na celu przyspieszenie wykonywania obliczeń na wielu wątkach (rdzeniach) w porównaniu do wykonania tego kodu na jednym wątku (rdzeniu) [4]. Wykonywanie kodu na wielu wątkach czy rdzeniach nie gwarantuje osiągnięcia przyspieszenia obliczeń. Dzieje się tak z kilku powodów:

- wielkości zadania – małe obliczenia mogą trwać krócej niż utworzenie, zarządzenie i synchronizacja obszarem równoległym;
- zależności danych – powodują one konieczność wprowadzenia przekształceń, które mogą wymagać dodatkowych synchronizacji.
- nieoptymalna liczba wątków – im więcej wątków jest tworzonych tym dłuższy czas tworzenie, zarządzania i synchronizacji tych wątków.
- ziarnistość – liczba obliczeń wykonywanych niezależnie przez procesory, pomiędzy punktami synchronizacji lub przesłaniem komunikatów [5].

W kodzie źródłowym mogą wystąpić zależności danych: odczyt-zapis gdzie ze zmiennej najpierw są odczytywane dane a w kolejnych linach kodu do tej samej zmiennej są zapisywane dane; zapis-odczyt w tej zależności do danej zmiennej najpierw zapisujemy dane a później je odczytujemy; zapis-zapis w tym typie do danej zmiennej zapisujemy co najmniej dwa razy różne informacje. Zależności danych mogą powodować, że wątki działające asynchronicznie wykonają operację odczytu lub zapisu przed innym wątkiem, który wg algorytmu powinien wykonać tę operacje wcześniej. W takiej sytuacji w pamięci komputera znajdować się będzie zła wartość, która spowoduje błędne wyniki obliczeń. W celu zapobiegnięcia takim sytuacjom w przypadku wykrycia zależności danych wprowadza się transformacje kodu źródłowego, które honorują zależności danych [6].



Rys. 1 Graf transformacji pętli programowej typu FAN (a) oraz typu PAR (b).

Podstawowymi transformacjami kodu źródłowego są transformacje FAN, PAR i PIPE, które stosuje się następujących sytuacjach: transformacja FAN – w przypadku braku zależności danych lub gdy zależności znajdują się wewnątrz pętli programowej; transformacja PAR – w przypadku występowania zależności pomiędzy iteracjami pętli oraz braku zależności wewnątrz ciała pętli; transformacja PIPE – w przypadku występowania zależności danych wewnątrz ciała pętli oraz pomiędzy iteracjami tej pętli.

W transformacji FAN składa się z trzech faz rozgłaszania obliczeń i agregacji wątek główny dzieli iteracje pętli oraz dane pomiędzy wątki przesyłając ich dane do obliczeń, wątki wykonują obliczenia a następnie przesyłają je do watka głównego, który je agreguje i zapisuje w pamięci komputera. Każdy wątek wykonuje te same operacje na różnych danych. Rysunek 1a przedstawia schematyczne działania transformacji FAN. Kolejną transformacją jest transformacja PAR w tym przypadku wątek główny dzieli instrukcje wewnątrz ciała pętli pomiędzy poszczególne wątki. Aby można było zrównoleglić pętle zgodnie z transformacją PAR musi ona posiadać co najmniej dwie instrukcje wewnątrz ciała pętli. Każdy wątek wykonuje inną instrukcję na tych samych danych a po każdej iteracji dane są synchronizowane za pomocą bariery. Całość jest powtarzana przez wszystkie iteracje. Rysunek 1b przedstawia schematyczne działania transformacji PAR.

Każda z transformacji rozwiązuje inny problem zależności danych transformacja FAN zależności wewnątrz ciała pętli natomiast PAR pomiędzy iteracjami. Zależności danych mogą być wszystkich trzech typów lub mogą one nie wystąpić w przypadku transformacji FAN. Obie transformacje wykorzystują inny typ równoległości transformacja FAN implementuje równoległość typu SIMD (Single Instruction Multiple Data) natomiast transformacja PAR implementuje równoległość MIMD (Multiple Instruction Multiple Data). Transformacja PAR będzie wolniejszą transformacją od transformacji FAN przez barierę po każdej iteracji, która zmusza szybsze wątki, aby zaczekały na wolniejsze wątki. W transformacji FAN wątki działają asynchronicznie, czyli niezależnie od siebie wykonują przydzielone zadania. Szybsze wątki mogą po zakończeniu zadania otrzymać kolejne obliczenia przez co zwiększane jest przyspieszenie obliczeń.

Trzeci typ transformacji PIPE wykorzystywany jest w sytuacji, gdy istnieją zależności wewnątrz ciała pętli oraz pomiędzy iteracjami. W takiej sytuacji wewnątrz ciała pętli dodawane są opóźnienia, które dają czas wątkom na zapisanie nowej wartości w pamięci komputera oraz po każdej iteracji stosuje się barierę synchronizującą wątki. Jest to najwolniejsza transformacja ze wszystkich, jednakże eliminuje wszystkie możliwe zależności pomiędzy danymi. Ze względu na sposób przetwarzania danych przez karty graficzne i nie możności ustawienia opóźnień w czasie przetwarzania przez kartę graficzną transformacja PIPE nie będzie dalej rozważana [7].

2.Modele czasu wykonywania się pętli programowych na CPU

Autorzy w swoich poprzednich pracach przedstawili model szacowania czasu dla procesorów wielordzeniowych oraz komputerów wieloprocesorowych z pamięcią dzieloną. W szacowaniu czasu wykorzystuje się informacje o środowisku testowym oraz analizuje pętlę programową. Model szacujący czas wykonania się pętli programowej dla transformacji FAN przedstawia wzór 1.

$$T(n) = \sum_{k=1}^{K} \frac{(r_k \cdot li \cdot z_k)}{lp \cdot n} + (w \cdot md) + cw + ti \qquad (1)$$

gdzie,

r – czas wykonania pojedynczej operacji

w – czas komunikacji

li – liczba iteracji (w pętlach zagnieżdżonych sumuje się wszystkie iteracje)

md – liczba danych potrzebnych wątkowi do obliczeń

z – liczba operacji wewnątrz ciała pętli

lp – stopień potoku w procesorze

cw – czas synchronizacji wątków

ti – czas inicjalizacji pomiarów

k – typ lokalności danych

n – liczba wątków

Model podzielono na trzy człony pierwszy z nich stanowi sumę iloczynu czasu wykonana się pojedynczej linii kodu na procesorze pomnożoną przez liczbę iteracji oraz liczbę operacji wewnątrz ciała pętli, całość dzielona jest przesz liczbę stopni potoku w procesorze pomnożoną przez liczbę wątków. Suma tego członu iterowana jest po poszczególnych lokalnościach danych określanych na podstawie analizy kodu. Jeżeli analizowany kod nie posiada zmiennych tablicowych mamy jeden typ lokalności, jeżeli oprócz zwykłych zmiennych mamy zmienne tablicowe, lokalność danych będzie wynosić dwa, jeżeli w zmiennych tablicowych są nie jednostkowe przeskoki typów lokalności danych będzie więcej. Drugi człon modelu odpowiada za szacowanie czasu przesłania danych pomiędzy wątki. Czas przesłania pojedynczej danej pomiędzy wątkami mnożony jest przez liczbę danych potrzebną do obliczeń. Ostatni człon zawiera czas synchronizacji wątków oraz błąd metody pomiarowej wykorzystywanego w czasie pomiarów rzeczywistych.

Prezentowany model testowano na pętlach z benchmarku NAS i benchmartku UTDSP osiągając średni błąd szacowania wynoszący 22% [8]. Dla modelu (1) przeprowadzono również analizę istotności parametrów modelu z wykorzystaniem miękkiej redukcji atrybutów. Wynik analizy pokazał, że wszystkie parametry modelu posiadają wysoką istotność, co oznacza, że nie można ich zredukować [9].

3.Schemat wykonywania obliczeń z wykorzystaniem GPGPU

W przeciwieństwie do rozdzielania obliczeń na poszczególne rdzenie centralnej jednostki jaką jest CPU, w przypadku przekazywania obliczeń do procesora GPU wykonywanych jest kilka dodatkowych operacji, które nie występują w przypadku CPU. Dotyczy to zwłaszcza wykorzystania frameworku OpenCL. Schemat operacji można podzielić na następujące kroki [10,11]:

1. Wykrycie i wybór platformy obliczeniowej
2. Wykrycie i wybór urządzenia
3. Utworzenie kontekstu obliczeniowego
4. Utworzenie kolejki zadań
5. Budowa jądra obliczeniowego
6. Alokacja pamięci po stronie urządzenia obliczeniowego
7. Ustalenie parametrów dla jądra obliczeniowego z kopiowanie danych do pamięci urządzenia
8. Uruchomienie jądra obliczeniowego
9. Odczyt danych z urządzenia obliczeniowego
10. Zwolnienie zasobów

Kroki tego schematu pomiędzy punktem 6 do 9 wykonywane są po stronie GPU lub też przy jego udziale. Z tego też powodu w modelu szacowania czasu wykonywania się pętli programowych z wykorzystaniem GPGPU należy uwzględniać także czasy wszystkich tych operacji. Zadaniem systemu gospodarza (hosta) jest sterowanie procesem przetwarzania danych przez jednostki obliczeniowe karty graficznej. System gospodarza zleca wykonanie zadania poprzez przesłanie danych, przekazanie procedur obliczeniowych oraz odczytanie danych wynikowych. W całym procesie, system gospodarza jest nadzorcą [12]. Po stronie karty graficznej wykonywana jest procedura obliczeniowa na przesłanych danych. Taka procedura nosi miano jądra obliczeniowego (kernela). Same kernele są uruchamiane w sposób asynchroniczny.

W przypadku NVIDIA Cuda schemat ten nieznacznie ulega uproszczeniu, gdyż ten sposób operuje jedynie na kartach graficznych tego producenta, więc możliwe było zrezygnowanie z procesu wyboru konkretnej platformy. Niemniej nadal występuje tu alokacja pamięci po stronie karty graficznej, kopiowanie do niej danych, kompilacja i uruchomienie jądra programu po stronie karty graficznej ze wcześniejszym ustaleniem parametrów uruchomieniowych, a finalnie kopiowanie wyników do pamięci hosta i zwolnienie zasobów [13]. Programowanie GPGPU zakłada, że funkcje kerneli nie zwracają żadnych wartości, zapisują wyniki w pamięci karty, stąd host musi je finalnie odebrać.

4.Szacowanie czasu wykonywania operacji z wykorzystaniem GPGPU

Mechanizmy zaszyte w języki programowania pozwalają w łatwy sposób mierzyć czasy wykonywania operacji zarówno po stronie CPU jak i GPU. Brakuje natomiast możliwości przewidywania jaki będzie ten czas jeszcze przed zleceniem obliczeń. Jest to ważne z takiego powodu, że czas wykonania obliczeń po stronie GPU może być jednak dłuższy od tego, gdy do obliczeń zostanie użyty jedynie CPU. Wykorzystanie obliczeń na karcie graficznej będzie opłacalne wtedy, gdy czas obliczeń wraz z przesłaniem danych będzie krótszy na karcie graficznej niż z wykorzystaniem procesora głównego. A zatem zostanie spełniona nierówność (2).

$$Czas\_obliczeń\_na\_GPU < Czas\_obliczeń\_na\_CPU \qquad (2)$$

Dzieje się tak, gdyż jednym z głównych czynników determinujących czas obliczeń jest przesył danych między systemem gospodarza, a jednostką obliczeniową, jaką jest GPU [14]. Na przepustowość danych wpływa wiele czynników, w tym wybór pamięci w której trzymane są dane, jak dane są przechowywane oraz wiele innych czynników.

Pomiar przepustowości może być przeprowadzony w teoretycznie prosty sposób. Do tego celu należy znać prędkość taktowania pamięci oraz szerokość magistrali. Teoretyczna prędkość przesyłu danych do pamięci wyrażona jest wzorem (3).

$$Prędkość = Taktowanie\_w\_Hz \cdot \frac{Szerokość\_magistrali}{8 \cdot 10^9} \qquad (3)$$

Przykładowo dla wykorzystywanej do testów karcie graficznej firmy NVIDIA GeForce GTX 1080 prędkość powinna wynieść około 320 GBps. Zgodnie z parametrami tej karty, gdzie taktowanie wynosi 10010 MHz, a szerokość szyny wynosi 256 bit według podanej zależności uzyskuje się przesył:

$$10010000000 \cdot \frac{256}{8 \cdot 10^9} = 320.32\ GBps$$

A zatem wynik jest prawidłowy i można ten wzór wykorzystać do szacowania czasu przesyłu pojedynczej komórki pamięci wyrażonej w jednostce bajtowej do pamięci karty graficznej jak i jej odbioru do pamięci gospodarza.

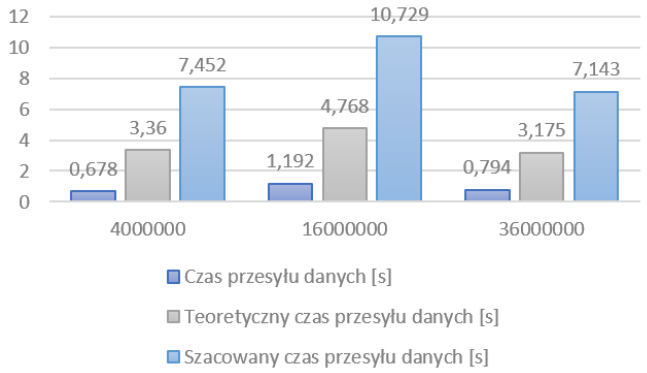
Niestety, jest to czas przesyłu danych teoretyczny, gdzie nie występują wąskie gardła w postaci wolniejszej pamięci RAM z której kopiowane są dane do pamięci karty graficznej, a także występująca optymalizacja przesyłu danych przez algorytmy w systemie OpenCL. Te i inne czynniki powodują, że czas prawdziwy różni się od czasu rzeczywistego. Dla dokładnego szacowania czasu przesyłu danych w dynamicznie zmieniającym się środowisku wykonawczym, autorzy niniejszego artykułu proponują przeprowadzenie prostego testu pomiarowego przesyłu niewielkiej tablicy do jądra wykonawczego bez wykonywania dodatkowych operacji wewnątrz jądra. Przedstawiona propozycja dotyczy wykonywania jądra programu na danych w postaci tablic pierwotnie dwuwymiarowych (w OpenCL przekształca się je na jednowymiarowe). Dla innych danych należy ten sposób przystosować do zastanej sytuacji, w tym uwzględniając rozmiar reprezentacji bajtowej dla poszczególnych typów przesyłanych danych. Do pomiaru czasu przesyłu danych do i z pamięci karty graficznej wykorzystywana jest instrukcja programistyczna w postaci funkcji clGetEventProfilingInfo z parametrami CL_PROFILING_COMMAND_START oraz CL_PROFILING_COMMAND_END, które wykonywane są na zdarzeniach (eventach) przekazywanych do funkcji sterujących środowiska OpenCL. Dla wymuszenia prawidłowości pomiaru po wykonaniu przesyłania danych należy wymusić zakończenie poprzez wykorzystanie funkcji clWairForEvents. Poniższy listing

1 przedstawia przykład prawidłowo wykonanego pomiaru czasu przesyłu porcji danych w postaci dwuwymiarowej tablicy.

```
cl_ulong time_start, time_end;

clEnqueueWriteBuffer(clCommandQue, d_A, CL_TRUE,
    0, mem_size_A, h_A, 0, NULL, &event);
clWaitForEvents(1, &event);

clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
    sizeof(time_start), &time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
    sizeof(time_end), &time_end, NULL)

double total_time = (time_end - time_start)/ 1000000.0;
```

Listing 1 Kod mierzący czas transmisji danych

Na wykresie zaprezentowanym na rysunku 2 przedstawiono oszacowane czasy przesyłania różnej wielkości tablic w porównaniu do teoretycznego czasu przesyłu tych samych tablic zgodnie z zależnością 2.



Rys. 2 Czas transmisji tablicy o różnych rozmiarach

Dla lepszego szacowania czasu autorzy wykonywali pomiar dziesięciokrotnie z odrzuceniem wartości odstających. Za wartość odstającą uznawano pomiar, który odstawał od średniej wartości o kilka rzędów wielkości. Jak widać na rysunku 2 czas szacowania przesyłu danych w zaproponowany sposób pozwala uzyskiwać znacznie lepsze dopasowania w stosunku do szacowania zależnością 3. Średni procentowy błąd oszacowania metodą teoretyczną wynosi 20 procent w odniesieniu do średniego zmierzonego czasu rzeczywistego, natomiast zaproponowaną w artykule zaledwie 5%.

W podobny sposób oszacować należy czas wykonywania jądra programu napisanego z wykorzystaniem OpenCL. Listing Y przedstawia kernel programu pozwalającego w łatwy sposób zmierzenie czasu wykonania pojedynczej instrukcji w OpenCL, przy czym w podanych przykładzie operacje będą wykonywane na tablicy dwuwymiarowej. Dla innych danych listing ten należy przystosować do zastanego kodu źródłowego.

```
__kernel void matrixMov(__global float* C,
          __global float* A, __global float* B,
          int wA, int wB){
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    for (int k = 0; k < wA; ++k)
    {
        C[ty * wA + tx] = A[ty * wA + k];
    }
}
```

Listing 2 Kod źródłowy przedstawiający podstawę do wyznaczania pojedynczej operacji wykonywanej przez kernel GPGPU

Dla lepszego doszacowania warto pomiar wykonać kilkukrotnie, niemniej nawet pojedynczy pomiar daje dobre rezultaty. Dla lepszego zrozumienia warto tutaj określać czas wykonywania instrukcji z wykorzystaniem tylko jednej jednostki obliczeniowej karty graficznej.

Całościowy wzór na szacowanie czasu wykonywania kodu źródłowego w środowisku OpenCL przedstawia zależność (4).

$$Czas = \frac{Z \cdot li \cdot r}{N} + w_S + w_R + c \qquad (4)$$

, gdzie:
Z – Liczba operacji w kernelu
Li – liczba wywołań kernela
r – czas jednej operacji
N – liczba jednostek wykonawczych
ws – czas przesyłu danych
wr – czas odbioru danych
c – czas kompilacji jądra

We wzorze 4 uwzględniono liczbę operacji realizowanych przez kernel, w tym występowanie obliczeń na zmiennych tymczasowych i operacje przeliczania adresów w tablicy jednowymiarowej zgodnie z reprezentacją dwuwymiarową, liczbę wywołań kernela odpowiadającym tradycyjnym iteracjom w pętlach programowych, czas wykonania pojedynczej instrukcji, a także sumę szacowanych czasów przesyłu danych do i z pamięci karty graficznej oraz czas przygotowania jądra polegającego na jego kompilacji. Wszystkie te dane są następnie dzielone przez liczbę jednostek wykonawczych biorących udział w obliczeniach.

5. Testy poprawności szacowania czasu wykonania kodu źródłowego w OpenCL

Do testów poprawności szacowania z wykorzystaniem OpenCL wykorzystano stacje roboczą działającą pod kontrolą systemu operacyjnego Windows 10 ze środowiskiem programistycznym MS Visual Studio 2015 i OpenCLopencl. W stacji tej wykorzystywane są dwa procesory Intel Xeon E5-2620 z taktowaniem 2.1 GHz oraz łączną liczbą 32 wątków sprzętowych, 64GB pamięci DDR4-2133 ECC oraz karta graficzna NVIDIA GeForce GTX 1080 8GB GDDR5X.

W listingu 3 (a-d) przedstawiono szacowane czasy wykonywania różnych kerneli w odniesieniu do zmierzonych średnich czasów rzeczywistych.

a)
```
__kernel void matriAdd(__global float* C,
          __global float* A, __global float* B,
          int wA, int wB){
    int tx = get_global_id(0);
    int ty = get_global_id(1);
    C[ty * wA + tx] = A[ty * wA + tx] + B[ty * wA + tx];
}
```

b)
```
__kernel void matrixAdd3(__global float* C,
          __global float* A, __global float* B,
          int wA, int wB){
    int tx = get_global_id(0);
    int ty = get_global_id(1);
    C[ty * wA + tx] = A[ty * wA + tx] + B[ty * wA + tx]+
          B[ty * wA + tx];
}
```

c)
```
__kernel void matrixLoopAdd(__global float* C,
          __global float* A, __global float* B,
          int wA, int wB){
    int tx = get_global_id(0);
    int ty = get_global_id(1);

    float value = 0;
    for (int k = 0; k < wA; ++k){
        value += A[ty * wA + k]+ B[k * wB + tx];
    }
    C[ty * wA + tx] = value;
}
```

Listing 3 Kody źródłowe kerneli: a). sumowanie dwóch tablic, b). sumowanie trzech tablic, c). sumowanie kolumnami dwóch tablic

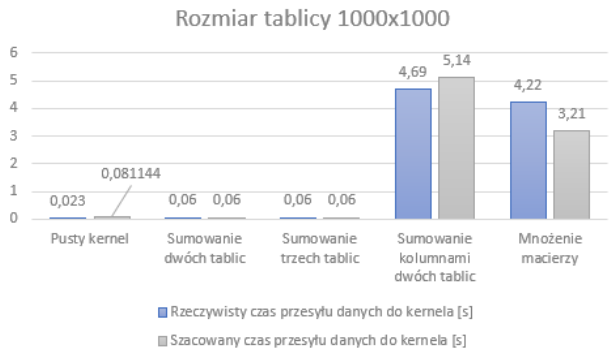
```
__kernel void matrixMul(__global float* C,
          __global float* A, __global float* B,
          int wA, int wB)
{
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    float value = 0;
    for (int k = 0; k < wA; ++k){
      float elementA = A[ty * wA + k];
      float elementB = B[k * wB + tx];
      value += elementA * elementB;
    }
    C[ty * wA + tx] = value;
}
```

d)

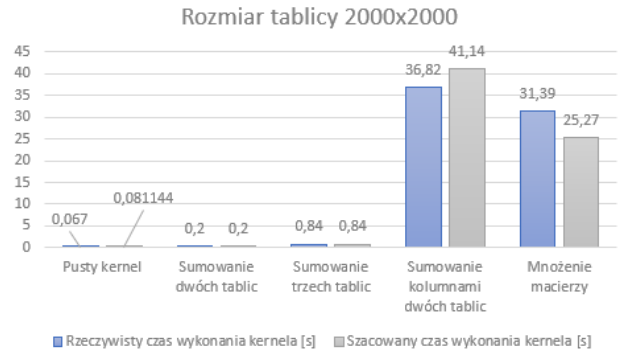
Listing 3 Kody źródłowe kerneli: d). mnożenie macierzy

W celu zweryfikowania poprawności proponowanej metody wykorzystano kernele realizujące: operacje puste, sumowanie dwóch tablic, sumowanie trzech tablic, sumowanie w pętli kolumn dwóch tablic oraz mnożenie macierzowe. Obliczenia były wykonywane na tablicach kwadratowych o wymiarze 1000x1000, 2000x2000 oraz 3000x3000. Kompilator ustawiony był do kompilacji w trybie x64. W obliczeniach testowych wykorzystywano 16·8 jednostek obliczeniowych karty graficznej.

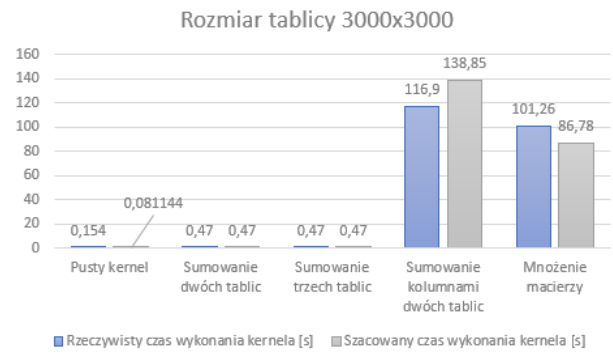
a)



b)

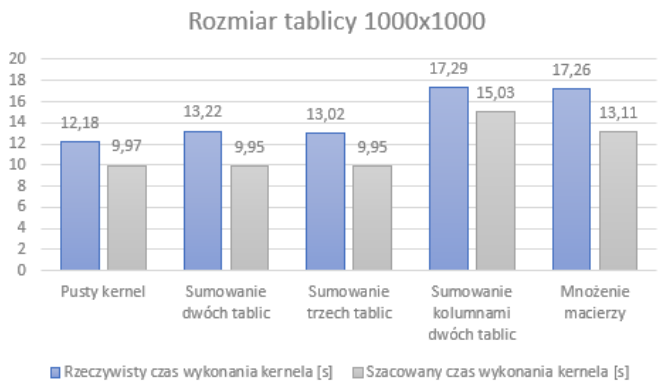


c)

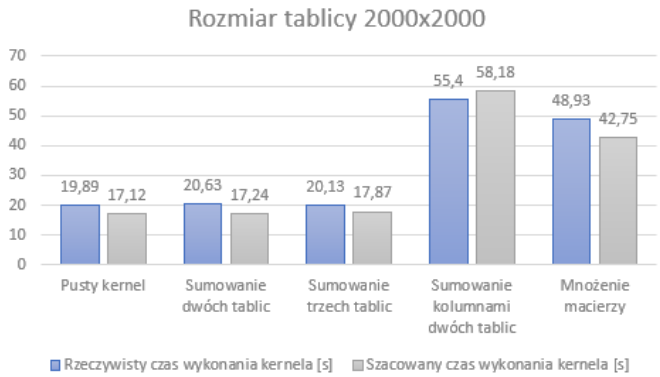


Rys. 3 Wykresy sprawdzenia poprawności zależności 3 co do szacowania czasu przesyłu danych do GPU

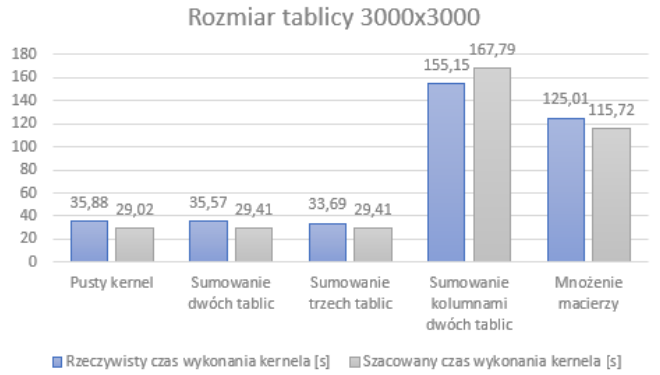
a)



b)



c)



Rys. 4 Wykresy prezentujące poprawność zależności 3 co do szacowania czasu wykonywania kerneli

Na rysunkach 3 i 4, przedstawiono proponowany schemat szacowania czasu wykonywania kodu źródłowego w OpenCL w zaproponowany sposób wykazuje się wysokim stopniem doszacowania, gdzie średni błąd procentowy w stosunku do średniego czasu rzeczywistego wynosił 5%. Wysoki poziom doszacowania czasu wykonywania pętli programowych na GPU stosunku do czasów rzeczywistych dotyczy zarówno czasów przesyłu danych pomiędzy pamięcią RAM komputera hosta, a pamięcią karty graficznej, jak i czasu wykonywania się samych kerneli w procesorze graficznym. Przedstawione dane potwierdzają poprawność prezentowanego modelu i wraz ze wcześniej przygotowanym modelem dla CPU pozwala na stosowanie ich do szacowania czasów wykonywania się pętli typu PAR w celu efektywnego rozdzielania zadań pomiędzy CPU i GPU w obliczeniach programowania hybrydowego na CPU i GPU.

Podsumowanie

W artykule przedstawiono modele szacowania czasów wykonywania się pętli programowych z wykorzystaniem centralnego procesora obliczeniowego oraz z wykorzystaniem nowego podejścia opierającego się na multiprocesorach dostępnych w nowoczesnych kartach graficznych. Przedstawione metody szacowania pozwalają określać czasy wykonywania się operacji na podstawie kodów źródłowych oraz prostych testów dotyczących czasów wykonywania się pojedynczej instrukcji czy czasów przesyłania pojedynczych porcji danych. Poprzez takie szacowanie istnieje możliwość pisania kodów programów, które będą pozwalały na określanie w sposób dynamiczny czy dany kod należałoby wykonać z wykorzystaniem CPU w danej chwili czy korzystniejsze będzie wykorzystanie do obliczeń określonej puli jednostek wykonawczej GPU. Przedstawione modele pozwalają także pisać uniwersalne kody dla środowisk hybrydowych jednocześnie wykorzystujących CPU jak i GPU w swoich obliczeniach. Jednocześnie zaproponowany sposób szacowania czasu z wykorzystaniem OpenCL jest na tyle uniwersalny, że z łatwością można go użyć do szacowania czasu przy zastosowaniu programowania z wykorzystaniem NVIDIA Cuda SDK. Przedstawione modele zostaną wykorzystane w pracach autorów w ramach zespołu badawczego Akademii Morskiej w Szczecinie nad rozwojem autonomicznych i semiautonomicznych jednostek transportu morskiego w czasochłonnych algorytmach wyznaczania trajektorii ruchu statków, rozwiązywania sytuacji kolizyjnych, rozwiązywania manewru ostatniej szansy i innych związanych z manewrowaniem tego rodzaju jednostek.

Bibliografia:

1. Burmeister H.-C., Bruhn W., Rødseth Ø., Porathe T., Autonomous Unmanned Merchant Vessel and its Contribution towards the e-Navigation Implementation: The MUNIN Perspective, International Journal of e-Navigation and Maritime Economy, Volume 1, December 2014, pp. 1-13
2. J. Koszelew, P. Wolejsza and D. Oldziej, "Autonomous Vessel with an Air Look," 2018 Baltic Geodetic Congress (BGC Geomatics), Olsztyn, 2018, pp. 102-106
3. Sanders J., Kandrot E., CUDAby Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2011
4. Rauber T., Runger G., Parallel Programming for multicore and cluster systems, Springer-Verlag 2012
5. Wolf F., Freitag F., Mohr B., Moore S., Wylie B., Large Event Traces in Parallel Performance Analysis, ARCS Workshops, 2006
6. Gebali F., Algorithms and Parallel Computing, Wiley, 2011
7. Lewis T., Foundations of Parallel Programming: A Machine-Independent Ap-proach. IEEE Computer Society Press, 1992
8. Wróbel M. (2015) Models for Estimating the Execution Time of Software Loops in Parallel and Distributed Systems. In: Zamojski W., Mazurkiewicz J., Sugier J., Walkowiak T., Kacprzyk J. (eds) Theory and Engineering of Complex Systems and Dependability. DepCoS-RELCOMEX 2015. Advances in Intelligent Systems and Computing, vol 365. Springer, Cham
9. Nozdrzykowski Ł., Nozdrzykowska M. (2018) Testing the Significance of Parameters of Models Estimating Execution Time of Parallel Program Loops According to the Open MPI Standard. In: Zamojski W., Mazurkiewicz J., Sugier J., Walkowiak T., Kacprzyk J. (eds) Advances in Dependability Engineering of Complex Systems. DepCoS-RELCOMEX 2017. Advances in Intelligent Systems and Computing, vol 582. Springer, Cham
10. Cegielski M. (2016) Parallel computation of transient processes on OpenCL framework, Przegląd Elektrotechniczny, ISSN 0033-2097, R. 92 NR 7/201
11. Thouti K., Sathe S. R,(2013) A Methodology for Translating C Programs to OpenCL, International Journal of Computer Applications (0975–8887) Volume 82–No3, November 2013
12. Sawerwain M., OpenCL Akceleracja GPU w praktyce, PWN, 2014
13. Farber R, Cuda Application Design and Development, Morgan Kaufmann, 2012
14. Nvidia's opencl best practices guide. Dostęp online: https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf, 2011

Models for estimating the time of program loop execution in parallel on a CPU and with the use of OpenCL computation on a GPU

The authors present models for estimating the time of execution of program loops compliant with the FAN model with no data dependencies or with data dependencies only within the body programming loop, which can be executed either by CPUs or by stream multiprocessors referred to as GPU cores. The models presented will make it possible to determine whether it would be more efficient to execute computation in the existing environment using the CPU (Central Processing Unit) or a state-of-the-art graphics card with a high-performance GPU (Graphics Processing Unit) and super-fast memory, often implemented in modern graphics cards. Validity checks confirming the developed time estimation model for GPU are presented. The purpose of these models is to provide methods for accelerating the performance of applications performing various tasks, including transport tasks, such as accelerated solution searching, searching paths in graphs, or accelerating image processing algorithms in vision systems of autonomous and semiautonomous vehicles, where these models allow to build an automatic task distribution system between the CPU and the GPU with the variability of computing resources.

Keywords: programming loop, estimating the time of the loop, programming CPU and GPGPU

Autorzy:

dr inż. **Łukasz Nozdrzykowski** – Akademia Morska w Szczecinie, Wydział Nawigacyjny, Instytut Technologii Morskich

mgr inż. **Magdalena Nozdrzykowska** – Akademia Morska w Szczecinie, Wydział Nawigacyjny, Instytut Technologii Morskich