SEBASTIAN IWASZENKO

Użycie akceleracji GPU do rozwiązywania wybranych modeli kinetycznych gazyfikacji węgla

Gazyfikacja węgla uznawana jest za jedną z tzw. czystych technologii węglowych. Chociaż jest znana już względnie długo, jej złożoność wciąż pozostaje wyzwaniem dla naukowców na całym świecie. Jednym z narzędzi używanych w badaniach jest symulacja. W pracy zbadano możliwości użycia GPGPU w modelowaniu gazyfikacji węgla. Użyto wybranego zbioru modeli (objętościowego, rdzenia bezreaktacyjnego i Johnsona). Modele oraz metody rozwiązań numerycznych zaimplementowano, jako kod szeregowy i równoległy. Zbadano czas realizacji obydwu metod oraz określono przyspieszenie kodu równoległego. Sprawdzono również wpływ wywołania funkcji matematycznej w kodzie GPU. Wyniki wskazują, że dla wszystkich modeli kod równoległy powoduje znaczne przyspieszenie obliczeń w stosunku do odpowiednika szeregowego, pod warunkiem, że użyje się wystarczająco dużego zbioru równań. Dlatego zaleca się użycie dedykowanego kodu GPU do symulacji gazyfikacji węgla w każdym przypadku, gdy wymagane jest rozwiązanie dużych systemów ODE.

Słowa kluczowe: *GPGPU, modelowanie gazyfikacji węgla, obliczenia równoległe*

1. WPROWADZENIE

Gazyfikacja węgla i podziemna gazyfikacja węgla to przykłady technologii, które pozwalają użyć węgla jako źródła energii lub substratu dla przemysłu chemicznego o względnie niskim wpływie na środowisko. Chociaż idea tej technologii jest znana od XVIII w., sam proces jest skomplikowany i trzeba dołożyć wielu starań, by przeprowadzić go efektywnie i bezpiecznie. Symulacja komputerowa jest jednym z narzędzi używanych do badania tego procesu. Jednakże obliczenia są zwykle wymagające i zajmują dużo czasu [16]. Uważa się, że możliwości, jakie dają obliczenia równoległe pomogą usunąć powyższy problem.

W pracy zbadano możliwości użycia GPGPU do modelowania gazyfikacji węgla. Do obliczeń wybrano trzy powszechnie używane modele, znane jako model objętościowy, model rdzenia bezreaktacyjnego i model Johnsona [12]. Każdy model opisuje zmianę karbonizatu w czasie w formie ODE.. Wszystkie trzy modele należą do grupy modeli kinetycznych. Najbardziej interesujące zastosowanie podejścia równoległego dotyczy rozwiązania nie pojedynczego równania, ale względnie dużego zestawu równań. Jest to powszechne w przypadku gdy trzeba użyć zestawu różnych parametrów do obliczania lub gdy konieczny jest jednoczesny postęp procesu dla kilkuset cząsteczek karbonizatu. Dotyczy to również przypadku, gdy w kawałku karbonizatu stosuje się podział na zestaw elementów. We wszystkich wymienionych przypadkach należy rozwiązać pewną liczbę niezależnych od siebie równań.

Pozostała część artykułu ma następującą strukturę: rozdział 2 zawiera ogólne informacje o GPGPU i jego programowaniu, rozdział 3 przedstawia wykorzystane modele i metody liczbowe, rozdział 4 opisuje zastosowany sprzęt i przypadki testowe, rozdział 5 zawiera wyniki i ich omówienie i, na koniec, rozdział 6 stanowi podsumowanie badań i przedstawia sformułowane wnioski z tych badań.

2. UŻYCIE GPU W OBLICZENIACH OGÓLNEGO PRZEZNACZENIA

Szybki rozwój obwodów półprzewodnikowych skutkował niesamowitym wręcz wzrostem stopnia ich złożoności. Prawo Moore'a, sformułowane

w latach 70-tych ubiegłego wieku, przewidywało zdublowanie liczby tranzystorów na obwodzie zintegrowanym co dwa lata [13]. Moc obliczeniowa sprzętu rozwijała się według tego samego wzorca. Jednakże wzrost integracji i wydajności wymagał coraz większego wysiłku. Na początku XXI wieku stało się oczywiste, że dalszy rozwój w tym tempie będzie prawie niemożliwy. Drastyczne spowolnienie zaobserwowano po roku 2004 z powodu czynników, które można ująć pod jednym terminem „mur z cegieł" (*brick wall*) [2]. Pojęcie to obejmuje wszystkie „mury", jakie powstają na drodze dalszego wzrostu mocy obliczeniowej i poziomu integracji. Większa moc obliczeniowa jest obecnie bardziej związana z obliczaniem równoległym niż ze zwiększeniem prędkości konkretnego komponentu. Nowoczesne zestawy GPU są przykładem tej strategii. Składają się one z setek elementów obliczeniowych i są przydatne zwłaszcza do rozwiązywania intensywnych obliczeniowo problemów, które można przedstawić w formie równoległej.

Jedno z najbardziej popularnych środowisk do zastosowania GPU jako głównego elementu przetwarzania oferuje firma NVIDIA. Narzędzie CUDA pozwala użytkownikom stworzyć kod równoległy za pomocą rozszerzenia C/C+++. Kompilator nvcc jest w stanie przekształcić kod do formy uruchamialnej przez graficzne multiprocesory strumieniowe [3]. Architektura CUDA może być uznana za rodzaj maszyny SIMD. Program przetwarzany jest przez elementy zwane multiprocesorami. Każdy multiprocesor jest w stanie obsługiwać setki wątków ułożonych w bloki. Dla każdego wątku dostępne są trzy poziomy pamięci. Po pierwsze, można użyć pamięci lokalnej, osobnej dla każdego wątku. Chociaż pamięć ta jest bardzo szybka, nie można jej wymieniać pomiędzy poszczególnymi wątkami. Pamięć globalna, z drugiej strony, jest dostępna dla wszystkich wątków, ale jest względnie wolna. Gdzieś pomiędzy nimi znajduje się pamięć współdzielona, która jest wspólna dla zestawu wątków działających w ramach tego samego bloku. Pamięć ta jest wolniejsza niż pamięć lokalna, ale szybsza niż globalna. Wszystkie rodzaje pamięci zlokalizowane są na karcie GPU.

Procedura ustawienia obliczeń na układach GPU przy pomocy CUDA składa się z kilku kroków. Po pierwsze, CUDA dokonuje rozróżnienia pomiędzy kodami i danymi dedykowanymi dla CPU i dla GPU. Pierwszy nazywany jest „hostem" (*host*) i nie wymaga innego postępowania niż zwykłe postępowanie w programach C lub C+++. Drugi nazywany jest „urządzeniem" (*device*) i w tym przypadku przychodzi z pomocą wsparcie z rozszerzeń i bibliotek CUDA. Aby aktywować kod na urządzeniu, należy zaimplementować specjalną funkcję nazywaną „jądrem" (*kernel*). Funkcja zaznaczana jest za pomocą przedrostka (specyfikatora) __global__ i może być dostarczana z parametrami. Ważne jest, aby wszystkie dane dostarczone jako parametry, zwłaszcza wskaźniki do obiektów jądra, były zlokalizowane w jednej z pamięci zarządzanych przez GPU. Dlatego, zazwyczaj wywołanie funkcji *kernel* ma następującą strukturę:

1. Przydzielanie pamięci na urządzeniu;
2. Kopiowanie danych z hosta na urządzenie;
3. Uruchomienie jądra i oczekiwanie na wyniki;
4. Kopiowanie danych (wyników z urządzenia na hosta;
5. Zwolnienie pamięci na urządzeniu.

W funkcji jądra, jako takiej dostępne są predefiniowane zmienne. Można ich użyć na przykład do identyfikacji wątku aktualnie przetwarzającego jądro. CUDA nie daje gwarancji kolejności, w jakiej uruchamiane są wątki przetwarzające jądra, ani też nie dostarcza mechanizmów domyślnej synchronizacji wątków.

Obliczanie za pomocą GPU i CUDA wzbudza w ostatnich latach coraz więcej uwagi. Przyspieszona realizacja działań może przynieść korzyści w wielu dziedzinach zastosowań. Jedną z nich jest na pewno przetwarzanie obrazów. Większość używanych algorytmów można skutecznie przedstawić w formie dostosowanej do wykonania równoległego. Realizowane są badania skoncentrowane na nowym podejściu do filtrowania i poprawiania wydajności za pomocą funkcji jądra wykorzystujących operacje typu *scatter* w miejsce bardziej naturalnego rozwiązania bazującego na operacji typu *gather* [15]. Niektórzy naukowcy zbadali udoskonalenie przetwarzania obrazów po aplikacji GPU. Zaobserwowano, że możliwe jest przetwarzanie z prędkością od 10 do 20 razy większą [5]. Możliwości GPU użyto również do rozwiązania zagadnienia 3 ciał (zastosowanego w symulacjach molekularnych) [18]. Podjęto również wiele prób zastosowania przyspieszenia GPU w obliczeniach i wysokowydajnych technologiach obliczeniowych: od badań mających na celu stworzenie dedykowanych, wysoce skutecznych metod do multiplikacji macierzy [11], przez narzędzia do rozwiązywania problemów algebraicznych [10], aż po rozwiązywanie problemów w różnych rodzajach przepływów [7] [8] [14] a także zastosowanie metod skończonych. [6]. Inne zastosowania to optymalizacja [1] lub aplikacje bazodanowe [17].

3. MODELE GAZYFIKACJI KARBONIZATU

Do testów obliczania skuteczności użyto trzech modeli gazyfikacji karbonizatu. Modele te twor zą serię, od najprostszych do najbardziej złożonych. W niniejszym rozdziale opisano każdy z tych modeli.

Model objętościowy jest modelem jednorodnym. Zakłada się, że gazyfikacja odbywa się w całej objętości cząsteczki karbonizatu. Proces postępuje aż do zużycia całości karbonizatu. Matematyczne przedstawienie modelu opisane jest za pomocą następującego równania [12]:

$$\frac{dx(t)}{dt} = k(1 - x(t)) \tag{1}$$

Współczynnik k jest stałą szybkości reakcji. Zazwyczaj zakłada się, że k jest funkcją temperatury według prawa Arrheniusa.

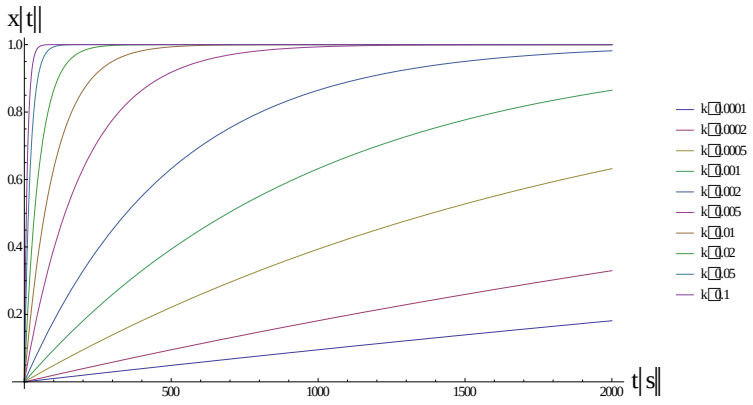
Model rdzenia bezreaktacyjnego opiera się na założeniu, że reakcja zachodzi wyłącznie na granicy węgla/karbonizatu i fazy gazowej. Rdzeń bezreaktacyjny utworzony jest z węgla, który nie bierze udziału w reakcji. Rdzeń zmniejsza się w miarę postępu reakcji. Równanie modelu wygląda następująco [8]:

$$\frac{dx(t)}{dt} = k \cdot (1 - x(t))^{\frac{2}{3}} \tag{2}$$

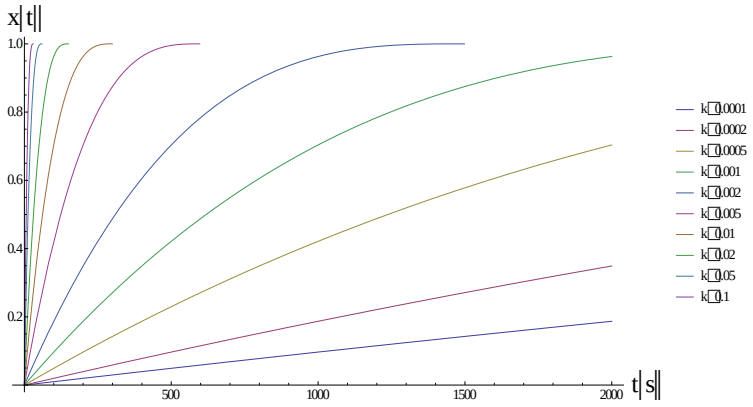
Najbardziej złożone równanie dotyczy modelu Johnsona. Model ten rozszerza model rdzenia bezreaktacyjnego uwzględniając opór środowiska porowatego. Opór ten ma wpływ na transport substratu i produktów do i z powierzchni cząsteczki. Jest to wynikiem porowatości węgla i popiołów. Podczas tej reakcji następuje znaczna zmiana powierzchni dostępnej dla procesów chemicznych. Równanie modelu Johnsona przedstawia się następująco:

$$\frac{dx(t)}{dt} = k \cdot (1 - x(t))^{\frac{2}{3}} \exp\left(-ax(t)^2\right) \tag{3}$$

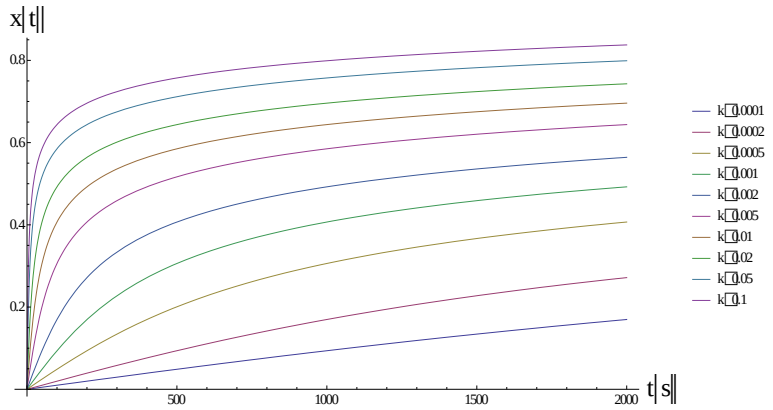
Symbol k to stała reakcji. Zakłada się, że w modelu Johnsona reprezentuje ona rodzaj karbonizatu i jego charakterystyczne cechy. Czynnik αx^2 opisuje wpływ zmieniającej się powierzchni i oporu transportu na szybkość reakcji. Poniższe rysunki pokazują typowe przebiegi zależności konwersji karbonizatu w czasie przewidywane przez każdy z wyżej opisanych modeli [9].



Rys. 1. Model objętościowy – prognoza konwersji karbonizatu w czasie



Rys. 2. Model rdzenia bezreaktacyjnego – prognoza konwersji karbonizatu w czasie



Rys. 3. Model Johnsona – prognoza konwersji karbonizatu w czasie

Wszystkie równania rozwiązano za pomocą metody Eulera:

$$x(t_{n+1}) = x(t_n) + hf\left(x(t_n), t_n\right) \tag{4}$$

Symbol *f* oznacza prawą stronę ODE, natomiast *h* – wielkość kroku czasowego. Metoda Eulera to najprostszy schemat różnicowy do rozwiązania ODE. Metoda Eulera znana jest ze swojej ograniczonej dokładności, jak również z zależności od wybranej wartości *h* wielkości kroku czasowego. Jednakże, jest wystarczająca do przetestowania skuteczności przyspieszenia GPU. Warto wspomnieć, że prawa strona ODE wyliczana jest tylko raz podczas całego procesu obliczeniowego.

4. PRZYPADKI TESTOWE

Kod testowy zaimplementowano przy pomocy C++ oraz zintegrowanego środowiska tworzenia oprogramowania Visual Studio. Kod został zaprojektowany, jako aplikacja na konsolę. Procedura odpowiedzialna za rozwiązywanie równań modelu otrzymuje wskaźnik na funkcję reprezentującą prawą stronę ODE. Podczas implementacji kodów szeregowych i równoległych, położono nacisk na utrzymanie jak największego podobieństwa obu wersji. CUDA Toolkit 7.5 posłużył jako platforma dla kodu CUDA. Wszystkie obliczenia przeprowadzono za pomocą reprezentacji liczb rzeczywistych pojedynczej precyzji.

Obliczenia zostały przeprowadzone w trzech seriach ze zmieniającą się liczbą kroków i równań w jednym zestawie (rozmiar zestawu równań). Założono, że dla każdego modelu obliczenia odbywały się przy zmianie rozmiaru systemu od 1 do 1000 równań. Obliczenia dla każdej wersji rozmiaru systemu i liczby kroków przeprowadzono 30 razy. Następnie wynik średni został zapisany jako reprezentatywny. Wyniki zebrano w plikach tekstowych csv. Wszystkie pomiary obliczono z dokładnością do milisekund (std::chrono::high_resolution_clock). Testy przeprowadzono na stacji roboczej pracującej pod systemem operacyjnym Windows 8.1 Pro. Intel Core i5-3579K CPU, działający przy 3.4 GHz, został użyty do realizacji kodu szeregowego. Kod równoległy CUDA zrealizowano przy pomocy karty graficznej NVIDIA GeForce 660Ti wyposażonej w 1344 rdzenie CUDA. Wszystkie testy wykonano przy nieobciążonym systemie obsługującym aplikację testującą, a wyniki zostały zapisane w pliki tekstowym. Zebrane dane były następnie przesłane do arkusza Excel do dalszej analizy.

5. UZYSKANE REZULTATY

Wybrane dane z danych zebranych przedstawiono w tabelach 1, 2 i 3. Można zaobserwować, że użycie funkcji matematycznych w obliczeniach miało duży wpływ na czas obliczania. Na przykład, przy 100 równaniach i 1500 krokach, czas obliczania dla modelu rdzenia bezreaktacyjnego i modelu Johnsona jest odpowiednio około dwa razy i trzy razy dłuższy niż dla modelu objętościowego. Warto zauważyć, że kod CUDA zabiera nawet więcej czasu obliczeniowego do użycia funkcji matematycznych. W takim samym przypadku, czyli 100 równań i 1500 kroków, model rdzenia bezreaktacyjnego potrzebuje pięć razy więcej czasu niż model objętościowy. Przy czym różnica między modelem rdzenia bezreaktacyjnego a modelem Johnsona jest o wiele mniejsza i wynosi około 10%. Prawdopodobne jest, że załadowanie bibliotek funkcji matematycznych do karty CUDA zabiera dużo czasu. Jednak, gdy biblioteka jest już załadowana, można jej użyć do obliczeń z większą skutecznością.

Można również zauważyć inny interesujący układ, tzn. zależność czasu obliczania od rozmiaru zestawu równań i liczby kroków. Jak można było przewidzieć, w przypadku obliczeń szeregowych zwiększenie rozmiaru zestawu równań albo wzrost liczby kroków prowadzi do wydłużenia czasu obliczania. Jeśli układ równań i liczba kroków są wystarczająco duże to można zaobserwować zależność o charakterze zbliżonym do liniowego (Rys. 4 i 5). Dla mniejszych wartości, wpływ zadań systemu (np. przełączenie kontekstów) jest prawdopodobnie wystarczająco duży, aby utrzymać liniowość. Sytuacja w przypadku równoległego kodu CUDA jest inna. Kod ten nie pokazuje prawie żadnej zależności od rozmiaru zestawu równań. Pomimo 100 lub 500 przeprowadzonych równań, różnica w czasach obliczania nie przekracza 5%. Poza tym, wspomniana różnica między czasami obliczania dla modelu rdzenia bezreaktacyjnego i modelu Johnsona mieści się w zakresie 10%. Zaobserwowana zależność pozwala na sformułowanie praktycznej reguły do zastosowania w obliczeniach dokonywanych za pomocą CUDA – w każdym przypadku, gdy w grę wchodzą duże układy równań przywołujące funkcje matematyczne, kod równoległy jest wart rozważenia. Jest on mniej wrażliwy na liczbę wywołań funkcji matematycznych i zmiany rozmiaru zestawu równań. W świetle tych stwierdzeń, należy wspomnieć, że narzut, jaki nakłada na obliczenia CUDA może być niemożliwy do akceptacji dla małego układu równań z małą liczbą kroków czasowych. Chociaż słuszność tej uwagi można sprawdzić analizując tabele od 1 do 3, jest ona również jasno przedstawiona w tabelach od 4 do 6, zawierających przyspieszenia kodu równoległego CUDA (przyspieszenie obliczone jest jako czas realizacji kodu CUDA w stosunku do czasu realizacji kodu CPU). Ponadto, wyniki przedstawiono na rysunkach 6 i 7.

Tabela 1

Czasy obliczeń w [ms] dla modelu objętościowego (przypadki gdy kod równoległy jest szybszy od szeregowego zaznaczono tłustym drukiem)

Równania	Kroki (iteracje)							
	Szeregowy				CUDA			
	1000	2500	5000	10000	1000	2500	5000	10000
100	5,95	6,02	12,03	24,65	**1,75**	**4,01**	**7,51**	**15,01**
200	4,84	12,01	24,01	48,10	**1,51**	**3,77**	**7,53**	**15,05**
300	7,21	18,03	36,00	71,97	**1,51**	**3,77**	**7,54**	**15,08**
400	9,60	23,98	48,04	96,01	**1,51**	**3,78**	**7,55**	**15,09**
500	11,99	29,99	59,98	119,92	**1,52**	**3,78**	**7,56**	**15,12**
1000	24,15	60,35	120,47	241,45	**1,53**	**3,81**	**7,62**	**15,24**

Tabela 2

Czasy obliczeń w [ms] dla modelu rdzenia bezreaktacyjnego (przypadki gdy kod równoległy jest szybszy od szeregowego zaznaczono tłustym drukiem)

Równania	Kroki (iteracje)							
	Szeregowy				Równoległy CUDA			
	1000	2500	5000	10000	1000	2500	5000	10000
100	8,57	14,62	29,14	58,14	11,03	24,41	48,82	97,64
200	11,72	29,35	58,13	116,11	**9,77**	**24,42**	**48,84**	**97,68**
300	17,53	43,62	87,15	174,07	**10,25**	**25,63**	**51,27**	**102,55**
400	23,35	58,14	116,09	232,28	**10,50**	**26,26**	**52,51**	**105,02**
500	29,18	72,91	145,09	289,92	**10,60**	**26,50**	**52,98**	**105,96**
1000	58,18	145,06	289,94	579,61	**10,75**	**26,86**	**53,71**	**107,39**

Tabela 3

Czasy obliczeń w [ms] dla modelu Johnsona (przypadki gdy kod równoległy jest szybszy od szeregowego zaznaczono tłustym drukiem)

Równania	Kroki (iteracje)							
	Szeregowy				Równoległy CUDA			
	1000	2500	5000	10000	1000	2500	5000	10000
100	12,24	22,23	44,28	88,47	12,29	27,69	55,38	110,77
200	17,85	44,33	88,48	176,84	**11,08**	**27,70**	**55,41**	**110,82**
300	26,80	66,45	132,65	265,10	**11,58**	**28,95**	**57,90**	**115,80**
400	35,55	88,52	176,85	353,35	**11,83**	**29,56**	**59,12**	**118,25**
500	44,42	110,62	220,90	441,67	**11,91**	**29,78**	**59,55**	**119,10**
1000	88,64	221,07	441,78	883,75	**12,06**	**30,15**	**60,28**	**120,54**

Tabela 4

Przyspieszenie kodu równoległego dla modelu objętościowego (przypadki gdy kod równoległy jest szybszy od szeregowego zaznaczono tłustym drukiem)

Równania	Kroki (iteracje)								
	1000	2000	3000	4000	5000	6000	7000	8000	9000
100	**3,40**	**1,38**	**1,57**	**1,61**	**1,60**	**1,61**	**1,61**	**1,61**	**1,61**
200	**3,21**	**3,19**	**3,20**	**3,20**	**3,19**	**3,19**	**3,19**	**3,23**	**3,19**
300	**4,77**	**4,78**	**4,78**	**4,78**	**4,77**	**4,78**	**4,77**	**4,77**	**4,78**
400	**6,35**	**6,35**	**6,36**	**6,36**	**6,36**	**6,38**	**6,37**	**6,36**	**6,36**
500	**7,91**	**7,93**	**7,93**	**7,93**	**7,93**	**7,93**	**7,93**	**7,94**	**7,93**
1000	**15,81**	**15,83**	**15,82**	**15,80**	**15,80**	**15,83**	**15,85**	**15,86**	**15,97**

Tabela 5

Przyspieszenie kodu równoległego dla modelu rdzenia bezreaktacyjnego (przypadki gdy kod równoległy jest szybszy od szeregowego zaznaczono tłustym drukiem)

Równania	Kroki (iteracje)								
	1000	2000	3000	4000	5000	6000	7000	8000	9000
100	0,78	0,60	0,60	0,60	0,60	0,60	0,60	0,60	0,60
200	**1,20**	**1,19**	**1,19**	**1,19**	**1,19**	**1,19**	**1,19**	**1,19**	**1,19**
300	**1,71**	**1,70**	**1,70**	**1,70**	**1,70**	**1,70**	**1,70**	**1,70**	**1,70**
400	**2,22**	**2,21**	**2,21**	**2,21**	**2,21**	**2,21**	**2,21**	**2,21**	**2,21**
500	**2,75**	**2,75**	**2,75**	**2,74**	**2,74**	**2,74**	**2,74**	**2,74**	**2,74**
1000	**5,41**	**5,40**	**5,40**	**5,40**	**5,40**	**5,40**	**5,40**	**5,40**	**5,40**

Tabela 6

Przyspieszenie kodu równoległego dla modelu Johnsona (przypadki gdy kod równoległy jest szybszy od szeregowego zaznaczono tłustym drukiem)

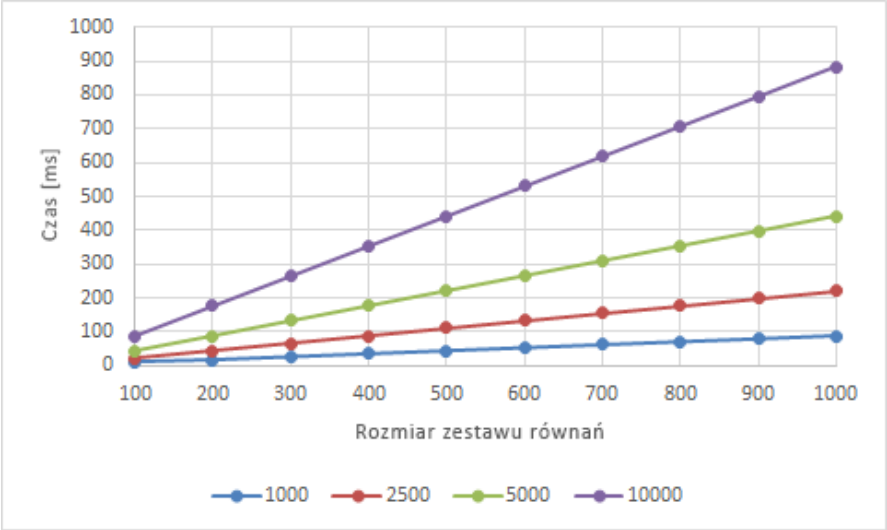
Równania	Kroki (iteracje)								
	1000	2000	3000	4000	5000	6000	7000	8000	9000
100	1,00	0,82	0,80	0,80	0,80	0,80	0,80	0,80	0,80
200	**1,61**	**1,60**	**1,60**	**1,60**	**1,60**	**1,60**	**1,60**	**1,60**	**1,60**
300	**2,31**	**2,30**	**2,29**	**2,29**	**2,29**	**2,29**	**2,29**	**2,29**	**2,29**
400	**3,01**	**3,00**	**2,99**	**2,99**	**2,99**	**2,99**	**2,99**	**2,99**	**2,99**
500	**3,73**	**3,72**	**3,71**	**3,71**	**3,71**	**3,71**	**3,71**	**3,71**	**3,71**
1000	**7,35**	**7,34**	**7,33**	**7,33**	**7,33**	**7,33**	**7,33**	**7,33**	**7,33**

Można zaobserwować, że w przypadku modelu objętościowego kod CUDA ma przewagę nad kodem szeregowym kiedy liczba równań w zestawie przekracza 30 i to samo dotyczy liczby iteracji. W przypadku mniejszych zadań obliczeniowych szybciej jest przeprowadzić je na CPU jako kod szeregowy. Maksymalne przyspieszenie zarejestrowane dla modelu objętościowego wyniosło 15,97 dla tysiąca równań z dziewięcioma tysiącami iteracji.

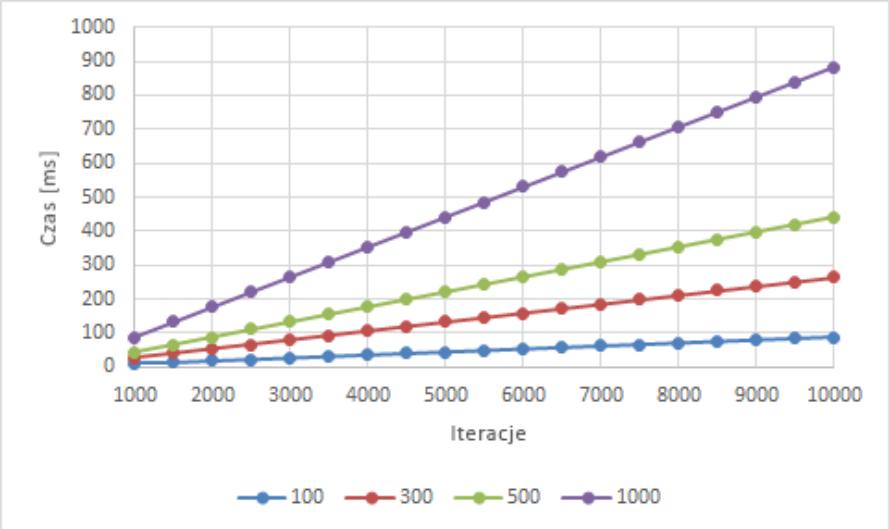
Przyspieszenie dla modelu rdzenia bezreaktacyjnego jest niższe niż dla modelu objętościowego. Zatem słuszne wydaje się być stwierdzenie, że 200 równań to limit, do którego kod CUDA posiada uzasadnioną przewagę nad kodem szeregowym. Maksymalne przyspieszenie zarejestrowane dla modelu rdzenia bezreaktacyjnego wyniosło 5,41 dla tysiąca równań z tysiącem iteracji.

Równania modelu Johnsona zachowują się podobnie do tych z modelu rdzenia bezreaktacyjnego. Jednakże model Johnsona oferuje większe przyspieszenie, jeżeli obliczenia prowadzone są z użyciem CUDA. Maksymalne przyspieszenie zarejestrowane dla modelu Johnsona wyniosło 7,35 dla tysiąca równań z tysiącem iteracji

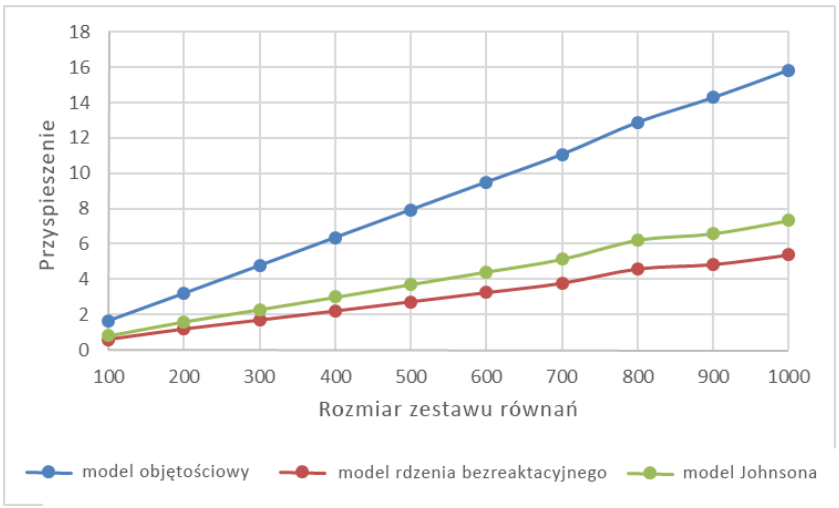
Warto wspomnieć, że wartości przyspieszenia osiągają pewną wartość stałą jeśli liczba kroków jest względnie duża (Rys. 7). Zależność ta dotyczy wszystkich testowanych modeli.



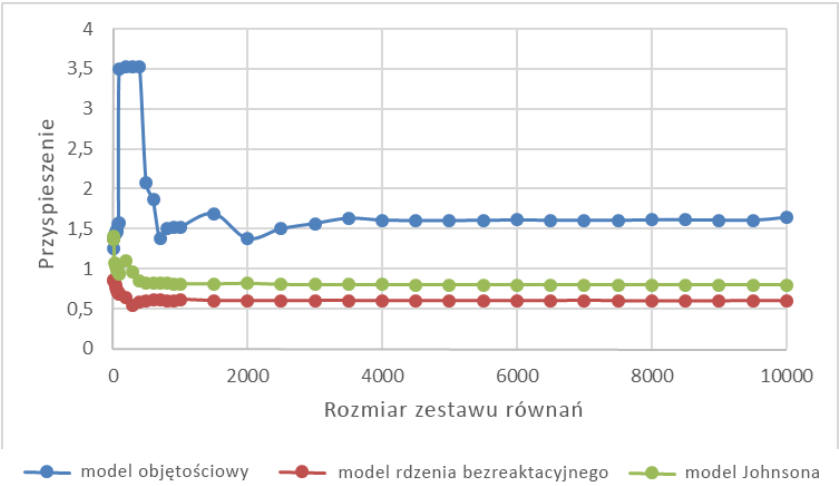
Rys. 4. Przyspieszenie kodu CUDA, jako funkcja rozmiaru zestawu równań



Rys. 5. Przyspieszenie kodu CUDA jako funkcja liczby iteracji



Rys. 6. Przyspieszenie kodu CUDA jako funkcja rozmiaru zestawu równań



Rys. 7. Przyspieszenie kodu CUDA jako funkcja liczby kroków (iteracji)

6. PODSUMOWANIE

W artykule przedstawiono zastosowanie implementacji szeregowej i równoległej dla wybranych modeli gazyfikacji karbonizatu. Do testów wybrano modele: objętościowy, rdzenia bezreaktacyjnego i Johnsona. Modele te oraz narzędzia do rozwiązywania problemów algebraicznych zostały wdrożone, jako kody szeregowe i równoległe. Kod szeregowy uruchomiony został na CPU za pomocą pojedynczego wątku, podczas gdy kod równoległy uruchomiono na GPU. Wyniki pokazują, że kod równoległy działa znacznie szybciej niż szeregowy, dopóki nie ma wywołań funkcji matematycznych w ramach kodu lub rozwiązywany zestaw równań jest wystarczająco duży. Zaobserwowano, że wartość przyspieszenia osiągała constans dla danej liczby równań, pod warunkiem że liczba iteracji była znaczna. Ponadto zauważono, że kod równoległy był mniej wrażliwy na wywołania bibliotecznych funkcji matematycznych niż funkcji reprezentujących prawą stronę modeli ODE.

W świetle powyższych obserwacji sformułowano następujące wnioski:

- Przy mniejszej liczbie równań zazwyczaj lepiej jest użyć szeregowego kodu CPU niż wersji równoległych GPU. Istnieje pewna liczba równań i iteracji, poniżej której kod szeregowy przynosi więcej korzyści niż równoległy;
- Kod równoległy CUDA pracuje znacznie lepiej w przypadku prostych równań. Wydajność kodu CUDA spadała bardziej niż szeregowego przy wywołaniu bibliotecznych funkcji matematycznych. Przy zwiększeniu liczby wywołań, spadek wydajności kodu równoległego był mniejszy niż w przypadku kodu szeregowego.
- Istnieje pewna wartość graniczna przyspieszenia dla każdego modelu i każdego rozmiaru zestawu równań.

Podziękowania

W artykule przedstawiono rezultaty pracy realizowanej w Głównym Instytucie Górnictwa w ramach projektu statutowego „Badanie wpływu zastosowanej architektury sprzętowej na efektywność obliczeń symulacyjnych wybranych modeli zgazowania węgla”, nr GIG: 11420255-350.

Literatura

1. Arca B, Ghisu T, Trunfio GA.: GPU-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard. Journal of Computational Science 11, 2015 pp. 258-68.
2. Asanovic K., Bodik R., Catanzaro B.C., Gebis J.J., Husbands P., Keutzer K., Patterson D. A., Plishker W. L., Shalf, J., Williams S. W., Yelick K. A.: The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB EECS-2006-183. Electrical Engineering and Computer Sciences. University of California Berkeley 2006.
3. Brodtkorb AR, Hagen TR, Saetra ML.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing 73(1), 2013 pp.4-13.
4. Castaño-Díez D, Moser D, Schoenegger A, Pruggnaller S, Frangakis A.S.: Performance evaluation of image processing algorithms on the GPU. Journal of structural biology 164(1), 2008 pp.153-60.
5. Díez DC, Mueller H, Frangakis AS.: Implementation and performance evaluation of reconstruction algorithms on graphics processors. Journal of Structural Biology 157(1), 2007 pp.288-95.
6. Fialko S.: Parallel direct solver for solving systems of linear equations resulting from finite element method on multi-core desktops and workstations. Computers & Mathematics with Applications 70(12), 2015 pp.2968-87.
7. Fu L, Gao Z, Xu K, Xu F.: A multi-block viscous flow solver based on GPU parallel methodology. Computers & Fluids 95, 2014 pp.19-39.
8. He X, Wang Z, Liu T.: Solving Two-Dimensional Euler Equations on GPU. Procedia Engineering 61, 2013 pp.57-62.
9. Iwaszenko S.: Using Mathematica software for coal gasification simulations–Selected kinetic model application. Journal of Sustainable Mining 14(1), 2015 pp.9-21.
10. Liu H, Yang B, Chen Z.: Accelerating algebraic multigrid solvers on NVIDIA GPUs. Computers & Mathematics with Applications 70(5), 2015 pp.1162-1181.
11. Matsumoto K, Nakasato N, Sakai T, Yahagi H, Sedukhin SG.: Multi-level optimization of matrix multiplication for GPU-equipped systems. Procedia Computer Science 4, 2011 pp.342-351.
12. Molina, A., Mondragón, F.: Reactivity of coal gasification with steam and CO_2. Fuel 77(15), 1998 pp.1831–1839. doi:10.1016/S0016-2361(98)00123-9.
13. Moore G.E.: Progress in Integrated Electronics. Technical Digest 1975. International Electron Devices Meeting. IEEE, 1975 pp. 11-13.
14. Oyarzun G, Borrell R, Gorobets A, Lehmkuhl O, Oliva A.: Direct numerical simulation of incompressible flows on unstructured meshes using hybrid CPU/GPU supercomputers. Procedia Engineering 61, 2013 pp.87-93.
15. da Silva J, Ansorge R, Jena R.: Efficient scatter-based kernel superposition on GPU. Journal of Parallel and Distributed Computing 84, 2015 pp.15-23.
16. Wachowicz, J., Janoszek, T., Iwaszenko, S.: Model tests of the coal gasification process. Archives of Mining Sciences 55, 2010 pp.249–262.
17. Walkowiak S, Wawruch K, Nowotka M, Ligowski L, Rudnicki W.: Exploring utilisation of GPU for database applications. Procedia Computer Science 1(1), 2010 pp.505-513.
18. Yaseen A, Ji H, Li Y.: A load-balancing workload distribution scheme for three-body interaction computation on Graphics Processing Units (GPU). Journal of Parallel and Distributed Computing 87, 2016 pp.91-101.

dr inż. SEBASTIAN IWASZENKO
Głowny Instytut Górnictwa GIG
Zakład Terenów Poprzemysłowych
i Gospodarki Odpadami
e-mail: siwaszenko@gig.eu