Andrey GRISHKEVICH

Politechnika Częstochowska, Instytut Informatyki

doi:10.15199/48.2017.03.11

Modele symulacyjne do szacowania wskaźników niezawodności strukturalnej systemów elektroenergetycznych tworzone na bazie kart graficznych

Streszczenie. *Zrealizowano implementację algorytmiczną modelu symulującego funkcjonowanie dwóch elementów systemu elektroenergetycznego korzystając z kart graficznych (GPU, Graphics Processing Unit) działających w środowisku wykonawczym C++ AMP (Accelerated Massive Parallelism). Umożliwiło to zwiększenie wydajności obliczeniowej dzięki równoległemu wykorzystaniu kilkuset rdzeni GPU w porównaniu do wykorzystania kilku rdzeni centralnej jednostki obliczeniowej (CPU, Central Processing Unit). Podano wyniki testowania modelu symulacyjnego.*

Abstract*. An algorithmic simulation model of operation for two elements of a power system with the usage of the graphics processing unit (GPU) based on the C++ AMP is implemented. It allows to increase the performance of calculations by the parallel use of several hundred GPU cores compared with several cores of central processing unit (CPU). The test results for the simulation model are presented. (**Modelling of the electric power system elements operation in terms of reliability with the usage of graphics processing unit**).*

Słowa kluczowe: GPGPU, symulacja równoległa, C++ AMP, model Markowa.
Keywords: GPGPU, parallel simulation, C++ AMP, Markow model.

Wstęp

Metody symulacji mają szereg zalet. Są łatwe do zrozumienia i realizacji. Pozwalają rozważać różne rozkłady zmiennych losowych. Jednak wymagają one więcej testów (mocy obliczeniowej komputera), których liczba znacznie wzrasta się wraz ze wzrostem wymiaru problemu.

Przy symulacji niezawodności tradycyjnym programem sekwencyjnym na komputerze osobistym osiągnięcie dopuszczalnej dokładności adekwatnej do wymaganego wymiaru wydaje się nieosiągalne ze względu na długi czas symulacji. Jednakże taka symulacja pozwala oszacować wymaganą do takich obliczeń moc obliczeniową [1].

Utworzenie (prostego) klastera obliczeniowego [2] pozwoli osiągnąć wydajność obliczeniową zapewniającą praktyczne zastosowanie technik modelowania. Takie podejście w sposób nieunikniony prowadzi do przeniesienia obliczeń z komputera osobistego do klastera / superkomputera. To z kolei generuje znaczne trudności organizacyjne, logistyczne, techniczne i ekonomiczne.

Centralny procesor (CPU) komputera osobistego nie jest jego jedynym urządzeniem obliczeniowym. Znaczną moc obliczeniową mają karty graficzne (GPU). Modernizacja technologiczna GPU doprowadziła do tego, że moc obliczeniowa GPU znacznie przewyższa moc obliczeniową CPU. Korzystanie z kart graficznych stanowi alternatywę dla korzystania z klasterów / superkomputerów, umożliwiając bardziej wygodny sposób dostępu do zasobów obliczeniowych. Poniżej znajduje się opis symulacji komputerowej na GPU funkcjonowania dwóch elementów systemu elektroenergetycznego ze względu na niezawodność.

Wybór podejścia do realizacji modelu symulacyjnego na akceleratorze graficznym

Budowa jednostki przetwarzania graficznego (GPU) znacznie różni się od budowy centralnej jednostki obliczeniowej / wykonawczej (CPU). Zorientowanie na specyficzne operacje grafiki komputerowej doprowadziło do istotnie różnej architektury takich urządzeń. W nowoczesnych CPU znajdują się dziesiątki rdzeni, a w nowoczesnych GPU - tysiące.

Centralna jednostka obliczeniowa jest procesorem ogólnego przeznaczenia, zoptymalizowanym do wykonania pojedynczego strumienia kolejnych instrukcji z maksymalną wydajnością. Procesor graficzny jest zaprojektowany do wykonywania ogromnej liczby równoległych strumieni instrukcji. Strumienie poleceń GPU są od początku zrównoleglone i brak jest w procesorze graficznym nakładów na zrównoleglenie instrukcji. GPU i CPU różnią się również zasadami dostępu do pamięci. Dostęp do pamięci procesora graficznego jest łatwo przewidywalny i dlatego, w odróżnieniu od jednostki centralnej, nie jest tu potrzebna duża pamięć podręczna (cache).

W procesorach ogólnego przeznaczenia większość obszaru chipa zajmują różne bufory instrukcji i danych, bloki dekodowania, bloki sprzętowego przewidywania rozgałęzień, bloki zmiany kolejności poleceń i pamięć podręczna (cache) pierwszego, drugiego i trzeciego poziomu. Wszystkie te bloki sprzętowe potrzebne są do przyspieszenia strumieni instrukcji dzięki ich zrównolegleniu na poziomie rdzenia procesora. Główny obszar procesora graficznego zajęty jest przez liczne jednostki wykonawcze, co pozwala mu jednocześnie przetwarzać tysiące strumieni instrukcji. W odróżnieniu od nowoczesnych procesorów CPU, procesory graficzne przeznaczone są do obliczeń równoległych z dużą liczbą operacji arytmetycznych.

Znajomość OpenGL / DirectX i konieczność programowania shaderów powstrzymywały rozprzestrzenianie się obliczeń ogólnego przeznaczenia na układach GPU (general-purpose computing on graphics processing units, GPGPU). Jedną z pierwszych prób przyjaznego użytkownikowi opisania "ogólnych" (naukowych, analitycznych, inżynierskich, itd.) obliczeń na GPU była opracowana przez firmę Nvidia technologia CUDA (Compute Unified Device Architecture) [3]. Za rozwinięcie tego podejścia można uważać opracowaną przez firmę Microsoft Corporation technologię przyspieszania aplikacji, napisanych w języku C++, dzięki wykonaniu kodu na procesorach graficznych C++ Accelerated Massive Parallelism (C++ AMP) [4]. Model programowania w C++ AMP opiera się na bibliotece i dwóch rozszerzeniach języka C++, zintegrowanych w kompilator Visual Studio. Zastosowanie tej technologii wymaga od karty graficznej wsparcia DirectX 11. Program komputerowy wykorzystujący C++ AMP może działać na kartach graficznych różnych producentów, w odróżnieniu od programów komputerowych wykorzystujących CUDA.

Tekst programu komputerowego do obliczeń na GPU różni się od tekstu programu do obliczeń na CPU. Obliczenia na GPU opisuje podzbiór poleceń (`restrict(amp)`) języka programowania C++. Na

przykład, polecenie `go to` nie nadaje się do wykonania na GPU. W związku z tym model komputerowy do symulacji niezawodności na GPU znacznie różni się od modelu rozproszonego, przedstawionego w [2].

Model Markowa funkcjonowania dwóch elementów systemu elektroenergetycznego z punktu widzenia niezawodności

Każdy element `E` systemu elektroenergetycznego (w zastosowaniu do elektroenergetyki to transformator, wyłącznik itd.) może znajdować się w jednym z czterech stanów. Przyjmujemy założenie, że `En` – stan normalnej pracy elementu, `Es` – stan między uszkodzeniem elementu i zakończeniem przełączeń operacyjnych, `Er` – stan awaryjnego remontu elementu, `Em` – stan remontu zapobiegawczego (zamierzonego odłączenia) elementu; `tExy` – losowy czas przejścia elementu `E` ze stanu `Ex` do stanu `Ey`. Stan systemu zależy od stanu każdego elementu systemu.

Przestrzeń stanów i diagram przejść pomiędzy stanami przy modelowaniu niezawodności funkcjonowania dwóch elementów `I`, `K` są przedstawione na rys. 1 [2]. Dla wygody stany ponumerowano, czyli `IsKn=S4`. W sumie dla modelu dwóch elementów rozważa się 15 stanów. Testowe dane numeryczne dotyczące wskaźników niezawodności elementów `I`, `K` systemu elektroenergetycznego są wzięte z [2]: `TIns0`=1/0,01 [rok] – średni czas przejścia elementu `I` ze stanu `In` do stanu `Is`; `TInm0`=1/2,2 [rok]; `TIsr0`=2/8760 [rok]; `TIrn0`=11,39/8760 [rok]; `TImn0`=7,96/8760 [rok]; `TKns0`=1/0,04 [rok]; `TKnm0`=1 [rok]; `TKsr0`=2/8760 [rok]; `TKrn0`=2,19/8760 [rok]; `TKmn0`=7/8760 [rok]. Prezentowane wartości liczbowe były traktowane jako parametry rozkładu wykładniczego odpowiedniej zmiennej losowej.

Algorytmiczna implementacja modelu symulacyjnego w języku programowania C++ AMP

Proces modelowania polega na generowaniu sekwencji stanów systemu, począwszy od pewnego stanu początkowego, na przykład `InKn`. Dla każdego stanu elementu `Ex` generuje się zmienną losową `tExy`, opisującą czas pobytu elementu w odpowiednim stanie [2].

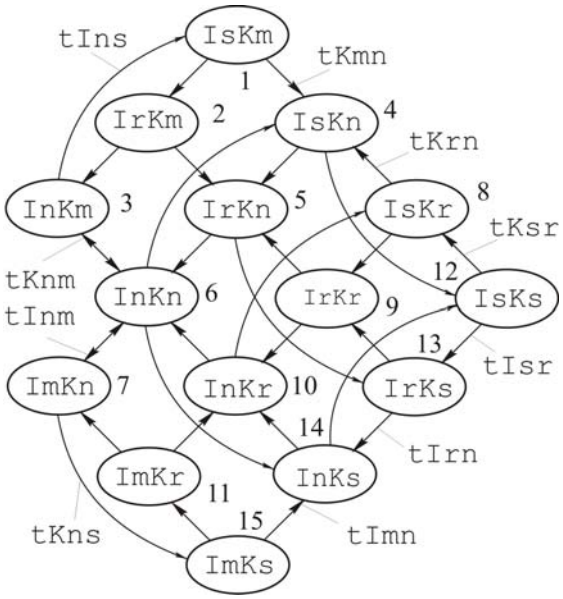
Całkowity czas symulacji `T` i czas `TSk` przebywania w stanie `Sk` określany jest przez liczbę testów `N`, która jest obliczana według wzoru

(1) `N = licznik1_serii*Niti*licznik2_TimeOUT`,

gdzie `licznik1_serii` – liczba serii, `Niti * licznik2_TimeOUT` – liczba testów w jednej serii.

Liczbę testów w serii dla karty graficznej wybiera się eksperymentalnie, tak aby obciążenie GPU trwało krócej niż 2 sekundy, co zapewnia stabilną pracę GPU. W systemie Microsoft Windows, aby zapobiec zbyt intensywnemu zużywaniu zasobów układu graficznego przez proces (co skutkuje tym, że wyświetlacz przestaje odpowiadać), wykorzystuje się mechanizm TDR (Time-Out Detection and Recovery). Domyślnie TDR resetuje reprezentację akceleratora graficznego jeśli bufor bezpośredniego dostępu do pamięci jest zajęty dłużej niż dwie sekundy. W tym przypadku program napisany w C++ AMP przestaje działać (zawiesza się). Liczba testów w serii składa się z liczby wątków `Niti` uruchomionych w procesorze graficznym i liczby prób przeprowadzonych dla pojedynczego wątku `licznik2_TimeOUT`. Tak więc proces symulacji zadany jest algorytmicznie kodem

```
#include <amp.h>
   ...
const int Niti = 1536;
extent<1> e_Niti (Niti);
   ...
 for (i = 0; i < licznik1_serii; i++) {
 parallel_for_each(e_Niti, [](index<1> idx)
restrict(amp) {
  for (j = 0; j < licznik2_TimeOUT; j++) {
   ...
}}}
```



Rys.1. Diagram przejść między stanami układu dwóch elementów

Polecenie `parallel_for_each` uruchamia obliczenia na GPU, a ekstent `e_Niti` określa konfigurację środowiska obliczeniowego GPU – liczbę równolegle uruchomionych wątków (nici, procesów) w GPU. W tym przypadku równolegle uruchamia się 1536 nici (procesy równoległe).

Podłączenie biblioteki generacji liczb losowych [5] zapewnia pracę na GPU równoległego generatora liczb losowych

```
#include "amp_tinymt_rng.h"
   ...
int seed = 5489;
tinymt_collection<rank> myrand(e_Niti, seed
+ i);
parallel_for_each(e_Niti, [](index<1> idx)
restrict(amp) {
auto t = myrand[idx];
t.next_single();
   ...
}
```

Wywołanie metody `next_single()` zapewnia generowanie następnej liczby losowej (rozkład równomierny) z przedziału [0,1].

Poniżej przedstawiono fragment kodu programu komputerowego symulacji procesu przejścia między stanami w modelu z dwóch elementów w języku programowania C++ z ograniczeniem `restrict(amp)` w stosunku do stanu `IsKn`

```
   ...
bool S06_InKn = true;
bool S04_IsKn = false;
bool S05_IrKn = false;
bool S12_IsKs = false;
   ...
if (S04_IsKn){
   if (tIsr <= tKns)
//IsKn -> IrKn
{tKns = tKns - tIsr;
 TS4 = TS4 + tIsr;
uzyskać nową wartość
zmiennej losowej tIsr;
 //tIsr: EXPO(TIsr0)
 S04_IsKn = false;
 S05_IrKn = true;}

else   //IsKn -> IsKs
{tIsr = tIsr - tKns;
 TS4 = TS4 + tKns;
uzyskać nową wartość
zmiennej losowej tKns;
//tKns : EXPO(TKns0)
 S04_IsKn = false;
 S12_IsKs = true;}}
   ...
```

W przedstawionym fragmencie kodu aktualny stan symulowanego systemu jest określany przez wartość "`TRUE`" zmiennej logicznej `SXX_XxXx`. Gwarantowane przejście do symulacji nowego stanu systemu dokonuje się w kolejnej iteracji pętli

```
  for (j = 0; j<licznik2_TimeOUT; j++)
{
   ...
}
```

Symulacja zmiennej losowej `tExy` o rozkładzie wykładniczym z parametrem `TExy0` w języku programowania C++ AMP zadawana jest za pomocą rozkładu równomiernego (funkcja `t.next_single()` biblioteki generacji liczb losowych) wyrażeniem

```
#include "amp_math.h"
   ...
do { tExy = -(TExy0*precise_math::
log(t.next_single())); }
while ( tExy==INFINITY );
//tExy: EXPO(TExy0)
```

Funkcja `log()` z przestrzeni nazw `Concurrency :: precise_math` [6] utworzona jest do obliczeń na układzie graficznym. Pętla

```
do { ... }
   while ( ... )
```

eliminuje błędy generatora liczb losowych polegające na wytwarzaniu dużych (nieskończonych, `INFINITY`) liczb.

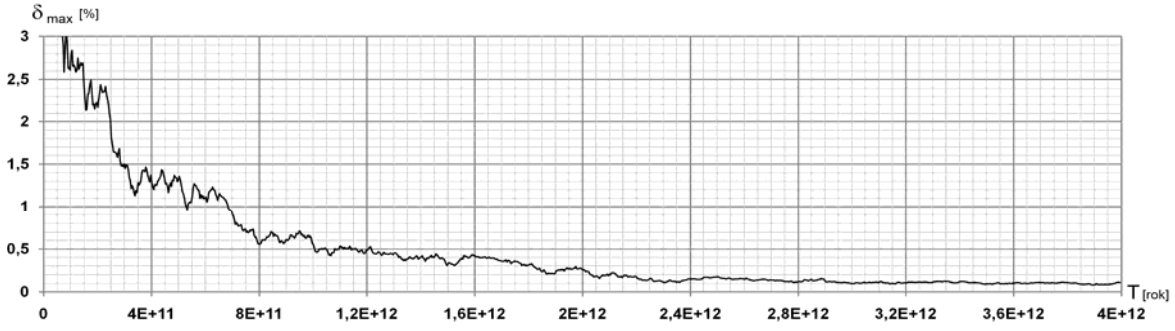
Testowanie modelu symulacyjnego

Obliczenia przeprowadzono z użyciem komputera Intel(R) Core(TM) CPU i5-3317U @ 1.70GHz i i3-3110M @ 2.40GHz (4 rdzenie) z kartą graficzną NVIDIA GeForce GT 740M (384 rdzenie NVIDIA CUDA albo unified shader processors).

Jedna seria modelowania (~$3{,}65*10^9$ lat) wykonywana jest przez 79,2 [s] na GPU. Symulacja czasu `TIK`=$5{,}43*10^{11}$ ($4{,}00*10^{12}$) lat wymagała 3:27 (24:06) godzin pracy wykorzystywanego komputera. Osiągnięto ~4,5 krotne przyspieszenie w porównaniu z 2 węzłowym klasterem obliczeniowym [2].

Rysunek 2 jest wykresem maksymalnego błędu względnego δ_{max} stanu na końcu przedziału symulacji. Z wykresu na rys. 2 widać, że przy interwale modelowania większym niż $2{,}5*10^{11}$ ($7*10^{11}$; $1{,}3*10^{12}$) lat maksymalny błąd względny stanu systemu jest mniejszy niż 2% (1%; 0,5%). Taka dokładność jest osiągana odpowiednio po 1:30 (4:10; 7:50) godzinach pracy komputera.

Otrzymane wyniki (czas osiągnięcia określonej dokładności) świadczą o akceptowalnym czasie modelowania prawdopodobieństwa stanów na GPU przy obliczaniu niezawodności na komputerze osobistym. Korzystanie z proponowanego modelu symulacyjnego (zamiast wzorów [7]) do obliczania wkładu stanów w wypadkowe wskaźniki niezawodności systemu elektroenergetycznego pozwala zwiększyć dokładność wyników obliczań niezawodności [7,8] dzięki stosowaniu dowolnych (różniących się od rozkładu wykładniczego) rozkładów zmiennych losowych.



Rys.2. Maksymalny błąd względny stanów na końcu przedziału symulacji

Wnioski

GPU mają wystarczającą moc obliczeniową, aby zapewnić akceptowalną dokładność symulacji niezawodności strukturalnej złożonych systemów elektroenergetycznych w rozsądnym czasie.

Wskazane jest aby wyposażyć komputery, wykonujące w szczególności obliczenia matematyczne metodą symulacji, w wydajną kartę graficzną z dużą liczbą rdzeni i ze wsparciem operacji matematycznych wykonywanych

z podwójną precyzją. Przeniesienie obliczeń z CPU do GPU skraca czas wykonywania obliczeń symulacyjnych.

Wydaje się, że perspektywiczną byłaby organizacja obliczeń równoległych z wymianą informacji między CPU i GPU do modelowania niezawodności złożonych systemów energetycznych.

***Autor**: prof. dr hab. inż. Andrey Grishkevich, Politechnika Częstochowska, Instytut Informatyki, al. Armii Krajowej 17, 42-200 Częstochowa, E-mail:* a.grischkevich@el.pcz.czest.pl*; grishkev_ramb@rambler.ru*

LITERATURA

[1] Grishkevich A., Burmutaev A., Modelowanie statystyczne oszacowań interwałowych wskaźników niezawodności strukturalnej układów elektrycznych, *Przegląd Elektrotechniczny* (*Electrical Review*), 88 (2012), nr. 8, 77-79

[2] Grishkevich A., Rozproszone modele symulacyjne pozwalające oszacować wskaźniki niezawodności strukturalnej systemów elektroenergetycznych, *Przegląd Elektrotechniczny* (*Electrical Review*), 91 (2015), nr. 12, 106-109

[3] Sanders J., Kandrot E., CUDA by example: an introduction to general-purpose GPU programming, Addison-Wesley (2011)

[4] Gregory K., Miller A., C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++, Punished with the authorization of Microsoft Corporation by O'Relly Media, Inc (2012)

[5] C++ AMP RNG Library (Random Number Generator). (http://amprng.codeplex.com/)

[6] Concurrency: precise_math Namespace. (https://msdn.microsoft.com/en-us/library/hh553049.aspx)

[7] Grishkevich A.A., Hudym V.I., Kruczinin A.M., Sawicki A., Zagadnienia energetyczne wybranych współczesnych urządzeń i systemów elektrostalowniczych, Wydawnictwo Politechniki Częstochowskiej, Częstochowa (2010)

[8] Grishkevich A., Burmutaev A., Piątek Ł., Frequency and outage duration in electric power systems, *Przegląd Elektrotechniczny* (*Electrical Review*), 85 (2009), nr. 3, 220-222