STUDIA INFORMATICA 2013
Volume 34 Number 2B (112)

Łukasz WYCIŚLIK, Łukasz KULIG
Politechnika Śląska, Instytut Informatyki

ZASTOSOWANIE TECHNOLOGII OPENCL W SYSTEMIE ZADAŃ ROZPROSZONYCH W ARCHITEKTURZE NADZORCA-WYKONAWCA

Streszczenie. Artykuł opisuje zastosowanie technologii OpenCL do wykonywania obliczeń z wykorzystaniem procesora graficznego w systemie rozproszonym. Zastosowanie technologii OpenCL pozwoliło na wykorzystanie dowolnej jednostki obliczeniowej danej stacji roboczej (zarówno GPU, jak i CPU). Stworzony system jest elastyczny i pozwala na zmianę implementacji OpenCL.

Słowa kluczowe: przetwarzanie rozproszone, OSGi, OpenCL, nadzorca, wykonawca

THE TASK DISTRIBUTION SYSTEM WITH MASTER-SLAVE ARCHITECTURE USING OPENCL TECHNOLOGY

Summary. This article describes the usage of OpenCL technology to perform computations with usage of GPU in the task distribution system. Usage of OpenCL technology allowed to involve any kind of workstation's processing unit (GPU, CPU). This system is flexible and allows to change the OpenCL implementation.

Keywords: distributed computing, OSGi, OpenCL, master, slave

1. Wstęp

W ciągu ostatnich lat zauważalnie zmniejszyło się tempo wzrostu wydajności nowych układów procesorowych. Przyczyną tego są ograniczenia natury elektronicznej – miniaturyzacja tranzystorów. Spowodowało to wzrost zainteresowania i nakładów prac na alternatywne metody zwiększania wydajności obliczeniowej systemów komputerowych. Jedną z tych metod jest wprowadzenie procesorów wielordzeniowych, które dla systemu operacyjnego wi-

doczne są jako osobne jednostki obliczeniowe. Jednostki te mogą wykonywać wiele obliczeń równolegle. Jednakże z powodu wysokich kosztów tych jednostek obliczeniowych oraz ograniczeń natury technologicznej, zaczęto konstruować tzw. systemy rozproszone, w których skład wchodzą często tanie i ogólnodostępne komputery.

Kolejną z metod zwiększenia wydajności systemów komputerowych stało się wykorzystanie mocy obliczeniowej kart graficznych. Karty graficzne, a dokładniej procesory graficzne (ang. GPU), są rozwijane niezależnie od zwykłych procesorów. Dzięki całkowicie odmiennej architekturze, procesory graficzne w niektórych zastosowaniach mają o wiele większy potencjał obliczeniowy niżeli zwykłe procesory. Przykładem na to mogą być obecne superkomputery, które wykorzystują właśnie wiele jednostek GPU do przeprowadzania bardzo skomplikowanych obliczeń.

W przypadku przetwarzania zadań w systemie rozproszonym, w zależności od funkcjonalności i obowiązków danej jednostki w systemie, można wyróżnić kilka modeli przetwarzania rozproszonego.

Jednym z nich jest nadzorca-wykonawca (ang. „*master/slave*”), który określa model komunikacji, gdzie jedno urządzenie bądź proces ma kontrolę nad jednym bądź kilkoma innymi urządzeniami. W niektórych systemach nadzorca jest wybierany spośród grupy urządzeń, gdzie pozostałe urządzenia pełnią rolę wykonawców. Jest to zależność jednokierunkowa, ponieważ wykonawcy wykonują polecenia zlecane przez nadzorcę, jednak w niektórych przypadkach wymiana komunikatów może być dwukierunkowa. Przykładem takiego rozwiązania może być potrzeba zwrócenia wyników wykonywanych obliczeń od wykonawcy do nadzorcy bądź przesłanie danych o stanie danego wykonawcy do nadzorcy.

Kolejnym modelem jest peer-to-peer, który zakłada, że każda jednostka w sieci ma taką samą funkcjonalność. Jednostka może zarówno pełnić rolę serwera, jak i klienta, nie ma więc tak jak w przypadku modelu nadzorca-wykonawca określonego podziału na role. Komunikacja przy użyciu takiego modelu jest trudna. Wpierw odpowiednie węzły muszą siebie nawzajem zlokalizować, aby następnie móc się ze sobą komunikować.

Inna koncepcja przyświeca modelowi agentów mobilnych, którymi nazywamy programy przenoszące się z jednego komputera na kolejny, wykonując tym samym zlecone przez kogoś zadanie. Agentem mobilnym jest całkowity program – kod i dane – który może pracować niezależnie.

Rozwiązanie opisane w niniejszym artykule bazuje na modelu nadzorcy-wykonawcy, którego szkielet został zrealizowany już w ramach wcześniejszych prac [1], obecnie natomiast dobudowana została możliwość wykorzystania mocy obliczeniowej GPU.

2. Analiza

Stworzenie systemu obliczeń rozproszonych, opartego na architekturze nadzorcy-wykonawcy, mającego cechy systemu GRID [2] (możliwość współpracy z węzłami o heterogenicznej architekturze, pracującymi niejednokrotnie pod kontrolą różnych systemów operacyjnych) oraz wykorzystującego moc obliczeniową GPU, wymaga takiego doboru metod obliczeniowych, które byłby możliwe do zastosowania z rozwiązaniami GPU różnych architektur.

Początkowo karty graficzne rozwijane były jako niezależne architektury, których wspólnym mianownikiem były sterowniki dostarczane przez poszczególnych producentów, implementujące podstawowe funkcje systemów operacyjnych oraz bibliotek graficznych wyższego poziomu – np. OpenGL (ang. *Open Graphics Library*). Cechą wspólną tych bibliotek było jednak ograniczenie ich interfejsu programistycznego (ang. *API*) do wywołań deklaratywnych. Oznacza to, że funkcjonalność tych bibliotek jest z góry ograniczona przez ich projektantów, a jakiekolwiek rozszerzenia tej funkcjonalności wymagają implementowania kolejnych wersji. W pewnym momencie dostrzeżono, że ciągle rosnący potencjał obliczeniowy kart graficznych nie musi być ograniczony tylko i wyłącznie do zastosowań związanych z grafiką. Każdy z liczących się producentów „otworzył" swoje rozwiązania, udostępniając dodatkowo interfejs imperatywny, umożliwiający programistom wykorzystanie mocy obliczeniowej nie tylko do zastosowań przetwarzania grafiki. Niestety, z powodu różnic w architekturze poszczególnych rozwiązań, udostępnione modele przetwarzania imperatywnego były ze sobą niekompatybilne. Dwa główne i konkurujące ze sobą rozwiązania zaproponowały firmy NVIDIA (CUDA – Compute Unified Device Architecture) oraz ATI (STREAM). Dopiero wprowadzenie standardu wyższego poziomu, jakim jest np. OpenCL (ang. Open Computing Language) [3], pozwoliło na jednolite korzystanie z architektur GPU różnych dostawców. Spowodowało to zwiększenie popularności obliczeń na GPU, które wykorzystuje się obecnie w wielu dziedzinach, w tym również w bazach danych.

Standard ten jest obecnie implementowany na różnych platformach wykonawczych (np. Java, .NET) poprzez wiele, najczęściej ogólnodostępnych, bibliotek. Poniżej przedstawione zostaną pokrótce rozwiązania dla Javy.

2.1. JavaCL

JavaCL [4] jest jednym z projektów, które umożliwiają wykorzystanie technologii OpenCL w aplikacjach Javowych. Biblioteka JavaCL została stworzona przez Oliviera Chafika i obejmuje specjalny zestaw klas i interfejsów, które służą do wykonywania konkretnych typów zadań. Przykładem takiego zestawu jest zestaw klas do obsługi operacji na ma-

cierzach. Twórca tej biblioteki zapewnił łatwość wykonywania operacji, które dość często użytkownik musiał zdefiniować sam. Dzięki zestawowi klas obługi działań na macierzach, pomnożenie macierzy czy też jej transponowanie ogranicza się do wywołania metody na stworzonym obiekcie macierzy.

JavaCL wymaga stworzenia jądra OpenCL w specjalnym, stworzonym przez firmę Khronos, języku. Jądro musi się znajdować w osobnym pliku, który musi mieć odpowiedni standard – rozszerzenie i nazwę.

2.2. JOCL

Podobnie jak w przypadku JavaCL, projekt JOCL [5] jest otwarty i darmowy. JOCL umożliwia wykorzystanie technologii OpenCL w aplikacjach Javowych. Projekt ten jest dalej rozwijany, ale nie jest już tak popularny jak JavaCL. Istnieje możliwość prostej integracji tej biblioteki z projektem stworzonym z wykorzystaniem popularnego narzędzia Maven. Na stronie projektu dostępne są przykłady zastosowania, ale nie są one niestety tak bardzo rozbudowane jak w przypadku JavaCL. Dostępna jest także dokumentacja całego projektu. Dodatkowym atutem strony projektu JOCL jest udostępnienie użytkownikom informacji o innych dostępnych implementacjach OpenCL dla Javy.

Tak samo jak w przypadku JavaCL, JOCL potrzebuje zdefiniowanego jądra OpenCL w osobnym pliku, z zachowaniem odpowiednich standardów.

2.3. Aparapi

Aparapi [6] pozwala programistom Javy na wykorzystanie mocy obliczeniowej procesorów graficznych lub innych procesorów, poprzez równoległe wykonywanie fragmentów kodu. Biblioteka ta konwertuje bytecode Javy do OpenCL w czasie wykonywania programu i wykonuje go z wykorzystaniem procesora graficznego. W przypadku gdy nie jest możliwe wykonanie określonego fragmentu kodu na procesorze graficznym, Aparapi wykona ten kod, korzystając z puli wątków.

Twórcy Aparapi uważają, że hasło przewodnie Javy – „Napisz Raz Uruchamiaj Wszędzie" (z ang. *Write Once Run Anywhere*) – obejmuje także wykorzystanie procesorów graficznych, dlatego właśnie postanowili rozszerzyć możliwości Javy poprzez stworzenie tej biblioteki.

W przeciwieństwie do JavaCL oraz JOCL, Aparapi nie potrzebuje do działania zdefiniowanego w osobnym pliku jądra OpenCL. Biblioteka ta umożliwia tworzenie jądra w kodzie programu. Kod tego jądra jest w takim przypadku w całości kodem Javowym. Programista

Javy, korzystając z tego projektu, nie musi uczyć się odmiany języka C, tak jak w przypadku dwóch poprzednich bibliotek.

2.4. Wybór implementacji

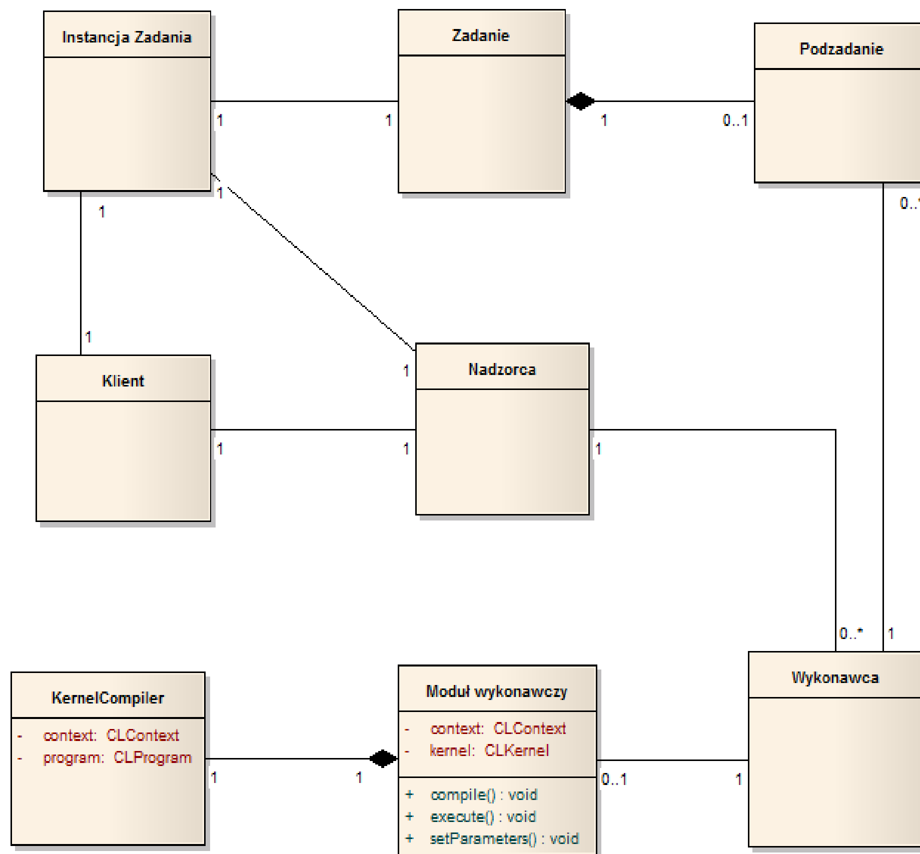
Na wybór implementacji na potrzeby opisywanego projektu składały się następujące czynniki:

- kompletna dokumentacja,
- wyczerpujące przykłady,
- łatwość integracji z istniejącym projektem,
- możliwość wykorzystania OSGi [7],
- możliwość zmiany jednej implementacji na inną.

Ostatecznie zdecydowano się na JavaCL, jako że rozwiązanie to okazało się najbardziej dojrzałe i najlepiej udokumentowane. Dodatkowo implementacja ta jest praktycznie idealnym odwzorowaniem API OpenCL, zaś w przypadku zmiany implementacji z JavaCL na JOCL wystarczy wprowadzić niewiele zmian. Biblioteki te są do siebie bardzo podobne. Obie wykorzystują zewnętrzne pliki z definicjami jądra OpenCL, mają bardzo podobne nazwy metod i klas, obie także wspierają OSGi.

3. Architektura i projekt

Na rysunku 1 znajduje się diagram klas przedstawiający elementy systemu. Aplikacja kliencka łączy się z Nadzorcą, zlecając tym samym wykonanie Instancji zadania o podanym identyfikatorze. Nadzorca pobiera z definicji Instancji zadanie dane wejściowe oraz identyfikator i wersję Zadania, do którego Instancja się odnosi i rozpoczyna wykonywanie zadania. Dla każdego Podzadania, Nadzorca łączy się z dostępnymi Wykonawcami i zleca im jego wykonanie.



Rys. 1. Podstawowe składowe systemu
Fig. 1. Basic components of the system

3.1. Projekt modułu

3.1.1. Moduł JavaCL

Moduł JavaCL odpowiada za przygotowanie środowiska zarówno dla Wykonawcy, jak i Nadzorcy, w celu wykonania metody OpenCL.

Moduł ten ma jedną klasę JavaCLModule, która dziedziczy po klasie bazowej wszystkich modułów w systemie ModuleBase. Moduł ten ma 3 odpowiedzialności:

- kompilację jądra OpenCL,
- ustawienie parametrów jądra,
- wykonanie metody zdefiniowanej w jądrze OpenCL.

Klasa modułu ma również prywatne pola, które służą do przechowywania, np. skompilowanego jądra OpenCL, oraz jako zmienne pomocnicze, gdyż trzy metody tego modułu są ze sobą ściśle powiązane. Nie da się wywołać drugiej nie wywołując uprzednio pierwszej.

3.1.2. Moduł OpenCL

Moduł OpenCL ma charakter czysto pomocniczy. Moduł ten skupia klasy będące tzw. klasami narzędziowymi. W przypadku tego modułu narzędzia te mają ściśle określony charakter – pomagają przy tworzeniu innych modułów wykorzystujących technologię OpenCL.

Moduł ten ma następujące klasy:

- *KernelUtils* – klasa zawierająca metodę do odczytywania jądra OpenCL z pliku do zmiennej łańcuchowej,
- *JavaCLKernelCompiler* – klasa, która tworzy jądro OpenCL, wykorzystując do tego implementację *JavaCL*. Klasa ta tworzy jądro, korzystając z najbardziej wydajnej jednostki pracującej w danym systemie.

3.2. Interfejs usługi

Interfejs modułu *JavaCL* wygląda następująco:

```
public interface IOpenCL {
    void compile(VString kernelData) throws IOException;
    void setParameters(VArray firstMatrix, VArray secondMatrix);
    void execute(VArray result);
}
```

Interfejs IOpenCL implementowany jest przez klasę *JavaCLModule*. Każdy moduł, który ma w założeniu wykorzystywać technologię OpenCL, powinien implementować interfejs *IOpenCL*.

Przeznaczenie metod usługi jest następujące:

- *compile* – metoda odpowiedzialna za kompilację jądra *OpenCL*, przekazanego w formie Stringa,
- *setParameters* – metoda odpowiedzialna za ustawienie parametrów dla jądra OpenCL,
- *execute* – metoda odpowiedzialna za wykonanie programu zapisanego w OpenCL i zwrócenie wyników.

3.3. Przykłady implementacji

Po uruchomieniu programu z wykorzystaniem technologii OpenCL, Nadzorca i Wykonawcy zaczynają wykonywać swoje zadanie. Pierwszym krokiem jest kompilacja jądra OpenCL, które zostało stworzone w pliku .cl. Nazwa pliku musi być taka sama jak nazwa metody OpenCL. Jeżeli moduł nie znajdzie takiej metody w pliku, zgłoszony zostanie wyjątek.

Przed samą kompilacją następuje utworzenie tzw. kontekstu OpenCL. Jest to proces, w którym sam OpenCL decyduje, czy użyć procesora graficznego czy zwykłego procesora. OpenCL wybiera najwydajniejszą jednostkę według własnych kryteriów.

Kompilacja polega na sparsowaniu pliku z metodą wykorzystującą OpenCL – tworzone są odpowiednie obiekty, które następnie umożliwiają wywołanie interesującej użytkownika metody. Kompilacja odbywa się poprzez wywołanie metody *compile*.

Kolejnym etapem jest ustawienie parametrów metody OpenCL. Odbywa się to poprzez metodę *setParameters*, która jest odpowiedzialna za stworzenie odpowiednich typów obiektów, tzw. buforów, które będą przechowywać dane podane przez użytkownika. Taka konwersja jest niezbędna, ponieważ OpenCL nie operuje na typach znanych z takich języków, jak Java czy C#, ma natomiast własne typy obiektów. Programista sam musi tę metodę zdefiniować, ponieważ nie wiadomo, jakie typy parametrów będą użyte w czasie wykonywania metody. Ustawienie parametrów bez uprzedniej kompilacji spowoduje zgłoszenie wyjątku.

Ostatnim krokiem jest wywołanie metody i zwrócenie wyników. Odbywa się to poprzez metodę *execute*, która wykonuje metodę OpenCL, odbiera z niej wyniki oraz konwertuje je na odpowiedni typ z języka Java.

Poniżej znajdują się przykładowe metody wykorzystujące OpenCL:

- metoda *compile*

```
@Command(id = "compile")
public void compile(VString kernelDataFile) {
   kernel = kernelCompiler.compile(kernelDataFile.toString());
}
```

- metoda setParameters

```
@Command(id = "setParameters")
public void setParameters(VArray firstArray, VArray secondArray)   {
   if (kernel == null) {
      log("Kernel is not set");
      return;
   }
   parametersCount = firstArray.size();
   Pointer<Float> firstArrayPointer = pointerToFloats(
                                          firstArray.toFloatArray());
   Pointer<Float>secondArrayPointer=pointerToFloats(
                                          secondArray.toFloatArray());
   CLBuffer<Float>arrayA=context.createFloatBuffer(
                                          Usage.Input,firstArrayPointer);
   CLBuffer<Float>arrayB=context.createFloatBuffer(
                                          Usage.Input,secondArrayPointer);
   Out=ontext.createFloatBuffer(Usage.Output, parametersCount);
   kernel.setArgs(arrayA, arrayB, out, parametersCount);
}
```

- metoda execute

```
@Command(id = "execute")
public void execute(VArray result) {
   CLQueue queue = context.createDefaultQueue();
   CLEvent addEvt = kernel.enqueueNDRange(queue, new int[] {parametersCount });
```

```
        Pointer<Float> outPtr = out.read(queue, addEvt);
        for (int i = 0; i < parametersCount; i++) {
            result.add(new VFloat(outPtr.get(i)));
        }
    }
```

4. Wnioski

Zastosowanie technologii OpenCL pozwoliło na stworzenie sieci obliczeniowej, w której każdy węzeł może mieć inną architekturę i pracować pod kontrolą innego systemu operacyjnego. Jedynym warunkiem jest istnienie implementacji maszyny wirtualnej Java dla danej architektury. Dodatkowo, jeśli któreś węzły dysponują jednostkami GPU, obliczenia mogą zostać przeprowadzone z wykorzystaniem tych jednostek (o ile ich moc obliczeniowa pozwoli zrealizować je szybciej niż za pomocą jednostki CPU). Oparcie systemu na szkielecie OSGi pozwoliło na elastyczną i modułową budowę. System zapewnia możliwość dystrybuowania zadań na jednostki wykonawcze, nadzorowanie wykonywania zadań w jednostkach wykonawczych oraz scalanie wyników.

Należy zauważyć, że technologia OpenCL nie nadaje się jednak do każdego rodzaju zadań. W szczególności w przypadku obliczeń z małą ilością danych wejściowych może się okazać, że większość czasu zajmuje kompilacja jądra OpenCL niż samo wykonanie zadania.

Dla popularnego i znanego już z przykładów z dokumentacji OpenCL zadania mnożenia macierzy, moduł wykorzystujący OpenCL, radzi sobie lepiej od modułu wykorzystującego zwykły procesor już przy najmniejszych rozmiarach macierzy. Można stwierdzić, że dla macierzy o większych rozmiarach i większej liczbie wykonawców, moduł wykorzystujący OpenCL będzie radził sobie jeszcze lepiej niż moduł, który korzysta tylko i wyłącznie z mocy obliczeniowej procesora.

Moduł wykorzystujący OpenCL został zaprojektowany pod kątem rozszerzalności. Można podmieniać implementacje OpenCL dla języka Java bez konieczności wprowadzenia poważnych zmian w systemie.

BIBLIOGRAFIA

1. Wyciślik Ł., Milczarek M.: Zastosowanie technologii OSGi w implementacji systemu zadań rozproszonych w architekturze nadzorca-wykonawca dla środowiska GRID. Studia Informatica, Vol. 33, No. 2B (106), Gliwice 2012, s. 419÷427.
2. Magoules F.: Fundamentals of GRID computing. 2010.
3. http://www.khronos.org/opencl/.

4. http://code.google.com/p/javacl/.
5. http://www.jocl.org/.
6. http://code.google.com/p/aparapi/.
7. McAffe J., VanderLei P., Archer S.: OSGi and Equinox. Creating highly Modular Java System.

Wpłynęło do Redakcji 16 stycznia 2013 r.

Abstract

This article describes the usage of OpenCL technology to perform calculations involving GPU in a distributed system. Usage of OpenCL technology allowed to use any kind of workstation's processing unit (GPU, CPU). Article describes OpenCL implementations for Java language, overall project architecture, created modules, service interface.

This system is flexible and allows to change the OpenCL implementation from one to another. In order to create module for specific computing domain, one needs only to implement a simple *IOpenCL* interface.

Adresy

Łukasz WYCIŚLIK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, lwycislik@polsl.pl.

Łukasz KULIG: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska.