Asynchroniczne i zdarzeniowe programowanie w PyOpenCL

Tomasz Rybak
tomasz.rybak@post.pl

PyConPL, 2012

1 Streszczenie

OpenCL to biblioteka umożliwiająca przeprowadzanie równoległych obliczeń; PyOpenCL umożliwia użycie jej w Pythonie. Napisanie programu który wydajnie działałby na jak największym zbiorze dostępnych urządzeń jest jednak bardzo trudne. Możemy jednak spróbować rozwiązać ten problemu inaczej, wykorzystując podejście znane z baz danych. Przekazując bibliotece co chcemy uzyskać (a nie w jaki sposób) oraz podając zależności obecne w danych dajemy możliwość wyboru najbardziej wydajnego rozwiązania dla tego konkretnego systemu. W przypadku OpenCL możemy tak zrobić wykorzystując programowanie zdarzeniowe i asynchroniczne.

2 Wprowadzenie do OpenCL i PyOpenCL

Nieustanny rozwój elektroniki sprawia że układy scalone mogą składać się z coraz większej liczby tranzystorów, jednak ograniczenia fizyczne, takie jak przekazanie mocy, konieczność chłodzenia, prędkość światła, sprawiają że zwiększanie częstotliwości układów zostało zatrzymane. Zamiast zwiększania częstotliwości producenci procesorów zwiększają możliwość równoległego wykonywania kodu oraz wprowadzają zróżnicowanie układów[8]. Zbytnie zróżnicowanie układów obliczeniowych utrudniłoby jednak naukę programowania i sprawiłoby że programy byłyby nieprzenośne. Z tego względu firmy takie jak Intel, AMD, NVIDIA, czy Apple zromadzone pod egidą organizacji Khronos stworzyły OpenCL[3], bibliotekę, API (Application Programming Interface) oraz język programowania przeznaczone do programowania równoległych układów obliczeniowych. OpenCL jest podobne w założeniach do OpenGL, również zarządzanego przez organizację Khronos.

OpenCL nie zakłada że układ obliczeniowy jest ściśle powiązany z komputerem. Wyróżnione są dwie strony: gospodarz (host), na którym uruchomiony jest program zarządzający obliczeniami, oraz urządzenie obliczeniowe (device), wykonujące niezbędne obliczenia i posiadające własną pamięć, niezależną od pamięci gospodarza. Program uruchomiony na gospodarzu ma za zadanie przygotować urządzenie (tworząc kontekst zarządzania), wczytać dane, przesłać je do urządzenia, zażądać wykonania obliczeń i pobrać ich wyniki.

OpenCL prezentuje hierarchiczny model udostępnianego sprzętu. Gospodarz może mieć dostęp do wielu platform; każda z nich jest zarządzana przez bibliotekę kliencką (ICD), często dostarczaną przez niezależne firmy. Każda platforma może posiadać wiele urządzeń (Computing Device). Każde urządzenie składa się z bloków (Computing Unit), z których każdy może wykonywać równolegle obliczenia na różnych danych, korzystając z pojedynczych układów wykonawczych (Processing Element). Bloki mogą działać niezależnie od siebie, co oznacza że każdy może wykonywać inny program; układy wykonawcze należące do jednego bloku muszą jednak wykonywać ten sam program. Blok (Computing Unit) jest najmniejszym obiektem którym możemy zarządzać z poziomu gospodarza; układami obliczeniowymi zarządza sterownik urządzenia. Model ten dość ściśle odpowiada fizycznej budowie procesorów graficznych GPU [2], jest jednak na tyle ogólny że umożliwia programowanie równoległe sprzętu innego typu.

Aby móc przeprowadzać obliczenia przy pomocy OpenCL należy stworzyć kontekst obliczeniowy. Kontekst może używać jedynie jednej platformy (nie może rozciągać się na kilka platform), lecz może współdzielić wszystkie urządzenia należące do tej jednej konkretnej platformy. Z kontekstem związane są funkcje wykonywane na urządzeniach (kernel), obiekty w pamięci oraz kolejki wykonania. Kolejka zawsze należy do jednego konkretnego urządzenia; nie można jej współdzielić tak jak kontekstu.

Całość OpenCL jest opisana z poziomu języka C. Nie wymaga to użycia zaawansowanych paradygmatów jak programowanie obiektowe czy funkcyjne i ułatwia użycie OpenCL w różnych językach w których istnieje możliwość wywołania bibliotek w języku C. Jednak jak zauważyli Jonathan Parri et al. w [5], nawet tak prosty pomysł jak SIMD (Single Instruction Multiple Data), umożliwiający równoległe wykonanie prostych obliczeń na danych w tablicy, oferowany przez MMX i SSE, nie jest wykorzystywany w językach wysokiego poziomu. Z kolei użycie biblioteki pisanej z myślą o języku C w innym (obiektowym lub funkcyjnym) języku wprowadza poczucie "obcości", jako że biblioteka taka nie umożliwia korzystania z idiomów danego języka. Z tego względu Andreas Kloeckner zaczął pracę nad PyOpenCL[4], biblioteką umożliwiającą użycie OpenCL z poziomu Pythona, a jednocześnie oferującą wszystkie zalety do których Python nas przyzwyczaił: automatyczne zarządzanie pamięcią (Garbage Collection), dynamizm, jak największa kompatybilność z numpy. Wykorzystanie PyOpenCL znacznie ułatwia i przyspiesza pisanie programów wykorzystujących OpenCL.

3 Zdarzenia

Najprostsze użycie OpenCL przebiega według schematu:

1. gospodarz inicjalizuje urządzenie;
2. gospodarz wczytuje dane i przesyła je do urządzenia;
3. gospodarz instruuje urządzenie jakie operacje wykonać na danych;
4. gospodarz oczekuje na zakończenie obliczeń;
5. gospodarz pobiera z urządzenia wynik obliczeń.

Jest to prosty przypadek (Program 1), obrazujący równocześnie bierność urządzenia które oczekuje na instrukcje ze strony gospodarza.

```
context = pyopencl.create_some_context()
queue = pyopencl.CommandQueue(context)
a = numpy.zeros(1000).astype(numpy.float32)
b = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, hostbuf=a)
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
int gid = get_global_id(0);
a[gid] = a[gid]+1.0;
    }
""").build()
program.increase(queue, a.shape, None, a_gpu)
pyopencl.enqueue_copy(queue, a_gpu, b)
```

Rysunek 1: Przykładowy program PyOpenCL

Została tu użyta jedna kolejka. W domyślnej konfiguracji OpenCL wykonuje zadania w takiej kolejności w jakiej umieściliśmy je w kolejce. Jeśli zadania zostały umieszczone w poprawnej kolejności urządzenie nie będzie próbowało wykonać obliczeń na danych których jeszcze nie ma w pamięci. Musimy jednak zwrócić uwagę na to aby żadna funkcja nie zaczęła się wykonywać zanim poprzednia się nie skończyła — szczególne ryzyko leży w transferze danych. W większości przypadków dba o to PyOpenCL.

Najprostszym sposobem jawnej synchronizacji jest żądanie zakończenia wszystkich zadań oczekujących w kolejce przed umieszczeniem kolejnego (Program 2). Czekanie na opróżnienie kolejki po wywołaniu każdej funkcji OpenCL nie jest jednak efektywne. W ten sposób niepotrzebnie zajmujemy gospodarza, który zamiast wykonywać coś pożytecznego (np. ładować dane z dysku) musi biernie i bezproduktywnie czekać na zakończenie obliczeń na urządzeniu. W ten sposób nie będziemy również w stanie efektywnie wykorzystać wielu urządzeń.

W obu powyższych przypadkach jedyna równoległość występuje w punkcie 3 (linia “program.increase”), gdzie obliczenia wykonywane są równolegle przez wiele układów obliczeniowych. Jeśli jednak mamy mniej danych niż układów wykonawczych, lub też musimy wykonywać obliczenia w pętli, to zarówno urządzenie jak i gospodarz przez dużą część czasu nie wykonują żadnych obliczeń, czekając jedno na drugie.

Lepszym rozwiązaniem jest wykorzystanie zdarzeń (Program 3). Umieszczamy żądanie w kolejce i pobieramy obiekt zdarzenia (Event). Następnie co jakiś czas sprawdzamy czy zdarzenie nie zmieniło stanu na “wykonane” (COMPLETE); w takim wypadku umieszczamy następne żądanie w kolejce. Pozwala to na równoległe wykonywanie obliczeń na urządzeniu i gospodarzu.

Zamiast regularnego sprawdzania stanu zdarzenia lepiej zażądać wykonania kodu przy zmianie stanu zdarzenia. OpenCL umożliwia podanie wskaźnika na funkcję która zaostanie wywołana jeśli zmieni się stan zdarzenia. Możemy w takiej funkcji zażądać wykonania innych instrukcji, np. umieszczenia kolejnego zdarzenia w kolejce. Funkcjonalność ta jest jednak niedostępna w PyOpenCL. OpenCL wymaga podania wskaźnika do funkcji w C a nie w Pythonie. Poza

```
context = pyopencl.create_some_context()
queue = pyopencl.CommandQueue(context)
a = numpy.zeros(1000).astype(numpy.float32)
b = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, shape=a.shape)
pyopencl.enqueue_copy(queue, a, a_gpu)
queue.flush()
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
int gid = get_global_id(0);
a[gid] = a[gid]+1.0;
    }
""").build()
queue.flush()
program.increase(queue, a.shape, None, a_gpu)
queue.flush()
pyopencl.enqueue_copy(queue, a_gpu, b)
queue.flush()
```

Rysunek 2: Przykładowy program używający synchronizacji

```
context = pyopencl.create_some_context()
queue = pyopencl.CommandQueue(context)
a = numpy.zeros(1000).astype(numpy.float32)
b = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, shape=a.shape)
event = pyopencl.enqueue_copy(queue, a, a_gpu)
event.wait()
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
int gid = get_global_id(0);
a[gid] = a[gid]+1.0;
    }
""").build()
event = program.increase(queue, a.shape, None, a_gpu)
while event.get_info(pyopencl.event_info.COMMAND_EXECUTION_STATUS)
        != pyopencl.command_execution_status.COMPLETE:
    pass
event = pyopencl.enqueue_copy(queue, a_gpu, b)
event.wait()
```

Rysunek 3: Program używający wątków do synchronizacji

```
context = pyopencl.create_some_context()
queue = pyopencl.CommandQueue(context)
a = numpy.zeros(1000).astype(numpy.float32)
b = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, shape=a.shape)
event0 = pyopencl.enqueue_copy(queue, a, a_gpu)
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
int gid = get_global_id(0);
a[gid] = a[gid]+1.0;
    }
""").build()
event1 = program.increase(queue, a.shape, None, a_gpu, wait_for=[event0])
event2 = pyopencl.enqueue_copy(queue, a_gpu, b, wait_for=[event1])
event2.wait()
```

Rysunek 4: Program instruujący OpenCL o zależności pomiędzy zadaniami

tym wywoływana funkcja musi być jak najprostsza, bez gwarancji kolejności wywołania, ani wątku w jakim zostanie wykonana.

Lepszą możliwością jest pozwolenie OpenCL na synchronizowanie zadań. Każda funkcja umieszczająca zadanie w kolejce zwraca zdarzenie związane z tym zadaniem (o czym wspomniałem powyżej) ale również przyjmuje listę zdarzeń od których zależy to zadanie. OpenCL wykorzystuje tę listę do zapewnienia że zadanie zostanie wykonane dopiero wówczas gdy wszystkie zadania zależne zostały zakończone. Dzięki temu nie musimy się martwić o czekanie na zakończenie transferu danych: OpenCL rozpocznie obliczenia jedynie wówczas gdy wszystkie niezbędne dane znajdą się w pamięci urządzenia. Oczywiście zależy to od tego czy powiadomiliśmy o tych danych przekazując odpowiednie zdarzenia jako argument wait_for.

Najkorzystniejsze jest wykorzystanie zdarzeń oraz kolejki out-of-order. Taka kolejka wykorzystuje mechanizm znany z procesorów: jeśli instrukcja może być wykonana (gdyż nie zależy od instrukcji jeszcze nie wykonanych) i możemy ją wykonać (gdyż mamy wolne zasoby) możemy tak zrobić, zmniejszając całkowity czas wykonania. OpenCL wykorzystuje tę technikę na wyższym poziomie abstrakcji, przeszukując kolejkę i wykonując całe zadania takie jak transmisja danych lub wykonanie obliczeń. W takiej sytuacji jednak należy bezwzględnie korzystać ze zdarzeń; w przeciwnym razie OpenCL będzie zakładało że żadne zadanie nie zależy od poprzedniego i będzie je wykonywało w dowolnej kolejności, zwracając niepoprawne wyniki.

Nie wszystkie implementacje OpenCL udostępniają kolejkę out-of-order. W takim wypadku możemy sobie jednak radzić tworząc kilka kolejek (Program **??**). Zadania umieszczone w każdej z nich będą wykonywane w kolejności umieszczenia, ale w przypadku wolnych mocy przerobowych OpenCL przejrzy wszystkie kolejki i wybierze zadanie z dowolnej z nich.

Użycie wielu kolejek może zwiększyć wydajność programu w przypadku wykonywania zadań które nie wykorzystują wszystkich bloków (wówczas można

```
context = pyopencl.create_some_context()
queue0 = pyopencl.CommandQueue(context)
queue1 = pyopencl.CommandQueue(context)
a = numpy.zeros(1000).astype(numpy.float32)
b = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, shape=a.shape)
event0 = pyopencl.enqueue_copy(queue0, a, a_gpu)
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
int gid = get_global_id(0);
a[gid] = a[gid]+1.0;
    }
""").build()
event1 = program.increase(queue1, a.shape, None, a_gpu, wait_for=[event0])
event2 = pyopencl.enqueue_copy(queue0, a_gpu, b, wait_for=[event1])
event2.wait()
```

Rysunek 5: Program wykorzystujący wiele kolejek

uruchomić dwa lub więcej zadań obliczeniowych), lub transferu danych z wykorzystaniem specjalizowanych układów; wówczas transfer danych i obliczenia, oczywiście nie na tychże danych, mogą przebiegać równolegle. Jak zauważył David Patterson [6], zwiększeniu szybkości układów nie towarzyszyło zwiększenie prędkości przesyłania danych (tzw. memory wall), zatem każde zmniejszenie czasu oczekiwania na dane niezbędne do obliczeń jest cenne.

Bardziej skomplikowana sytuacja ma miejsce gdy mamy kilka urządzeń (Program 6). Wówczas możemy równolegle wykonywać obliczenia na różnych danych. Tu jednak ważniejsze jest podzielenie danych na takie niezależne fragmenty, aby każde urządzenie mogło sobie poradzić z obliczeniami, oraz takie umieszczenie danych, aby jak najbardziej ograniczyć transfery. Doświadczanie podpowiada że często to transfer danych, a nie czas obliczeń staje się czynnikiem ograniczającym wydajność.

Zdarzenia są również przydatne przy współpracy OpenCL z innymi bibliotekami. Jednym z przykładów jest wykorzystanie zdarzeń do koordynacji współpracy OpenCL i OpenGL, gdy np. chcemy wizualizować dane na których dokonujemy obliczeń. Ważne jest wówczas aby nie próbować wyświetlać wciąż transferowanych danych, jak również by nie wykonywać obliczeń na danych które są obecnie wyświetlane i vice versa.

4 Rozszerzenia

Pisząc o OpenGL w kontekście OpenCL nie sposób nie wspomnieć o rozszerzeniach (extensions). OpenCL jest również w tym względzie bardzo podobne do OpenGL. OpenCL zostało zaimplementowane na różnych urządzeniach, które oferują różne możliwości. Uwzględnienie tych wszystkich możliwości w standardzie jest niemożliwe, lecz producenci sprzętu powinni móc udostępnić jego wszystkie możliwości programistom. Z tego względu OpenCL posiada model

```
platfoms = pyopencl.get_platforms()
devices = platforms[0].get_devices()
context = pyopencl.Context(devices)
queue0 = pyopencl.CommandQueue(context, device=devices[0])
queue1 = pyopencl.CommandQueue(context, device=devices[1])
a = numpy.zeros(1000).astype(numpy.float32)
b = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, shape=a.shape)
b_gpu = pyopencl.array.Array(context, shape=a.shape)
event0 = pyopencl.enqueue_copy(queue, a, a_gpu)
event1 = pyopencl.enqueue_copy(queue, b, b_gpu)
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
int gid = get_global_id(0);
a[gid] = a[gid]+1.0;
    }
""").build()
event2 = program.increase(queue, a.shape, None, a_gpu, wait_for=[event0])
event3 = program.increase(queue, b.shape, None, a_gpu, wait_for=[event1])
event4 = pyopencl.enqueue_copy(queue, a_gpu, a, wait_for=[event2])
event5 = pyopencl.enqueue_copy(queue, b_gpu, b, wait_for=[event3])
event4.wait()
event5.wait()
```

Rysunek 6: Program wykorzystujący dwa urządzenia

rozszerzeń wzorowany na rozszerzeniach OpenGL. Programista może pobrać listę udostępnianych rozszerzeń, a następne pobrać wskaźnik na funkcję implementującą konkretne rozszerzenie. Ponieważ jednak mechanizm ten zakłada korzystanie z języka C, jego użycie w PyOpenCL jest ograniczone.

5 Podsumowanie

OpenCL jest wciąż młodą, rozwijaną technologią. Nie wszyscy producenci udostępniają pełne możliwości (np. kolejki out-of-order) lub najnowsze wersje bibliotek (w chwili pisania tego artykułu NVIDIA wciąż nie udostępniła OpenCL 1.2, pomimo że minęło już 9 miesięcy od opublikowania standardu). Moim zdaniem jednak OpenCL ma poważne szanse stać się standardem w programowaniu, zwłaszcza że pojawiają się możliwości jego użycia na urządzeniach przenośnych (komórki), gdzie jego użycie przyczynia się od oszczędności energii.

OpenCL umożliwia programowanie różnych układów: CPU, GPU, FPGA oraz bardziej specjalizowanych układów. Zgodnie z przewidywaniami Shekhara Borkara et al.[1], pracowników firmy Intel, rozwój układów będzie zmierzał w stronę heterogenizacji. Na jednej kości będzie wiele układów różnych typów: kilka rdzeni ogólnego zastosowania, wiele szybkich lecz ograniczonych układów zdolnych do przeprowadzania obliczeń, specjalizowane układy kryptograficzne, etc. Już teraz możemy to zauważyć w przypadku układów SoC (System on Chip) używanych w telefonach komórkowych. Podobny trend jest prezentowany przez AMD w architekturze Fusion, gdzie na jednej kości jest CPU oraz GPU. Trend ten obrazuje konieczność korzystania z bibliotek (takich jak OpenCL) ułatwiających pisanie programów wykorzystujących tak różne orpgramowanie. Same biblioteki jednak nie wystarczą: konieczne będzie użycie nowych, równoległych struktur danych ([7]) oraz innych modeli programowania.

Literatura

[1] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of ACM*, 54:67–77, May 2011.

[2] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Communications of ACM*, 51(10):50–57, 2008.

[3] Khronos OpenCL Working Group. *The OpenCL Specification Version 1.2*, 11 2011. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf Accessed at 2012-07-30.

[4] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[5] Jonathan Parri, Daniel Shapiro, Miodrag Bolic, and Voicu Groza. Returning control to the programmer: Simd intrinsics for virtual machines. *Communications of ACM*, 54:38–43, April 2011.

[6] David A. Patterson. Latency lags bandwidth. *Communications of ACM*, 47(10):71–75, 2004.

[7] Nir Shavit. Data structures in the multicore age. *Communications of ACM*, 54:76–84, March 2011.

[8] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.