Jednostka GPU oraz OpenCL

– wprowadzenie

Artykuł prezentuje ewolucję układów GPU, ich współczesną architekturę oraz możliwości zastosowania w aplikacji – pod tym kątem omówiono framework OpenCL. W kolejnej części artykułu, zaprezentujemy przykładowe aplikacje oraz wskazówki odnośnie do środowiska projektowego, sposobu kompilowania i uruchomienia oprogramowania na platformie z mikroprocesorem i.MX6 firmy Freescale.

Nie można omówić frameworka OpenCL ani procesora graficznego GPU, również w kontekście systemu bazującego na pojedynczym układzie scalonym SoC, bez zrozumienia podstaw architektury GPU oraz kształtujących ją czynników, dlatego na początek podamy nieco informacji mających na celu wprowadzenie do tej tematyki.

Podstawy współczesnej architektury GPU

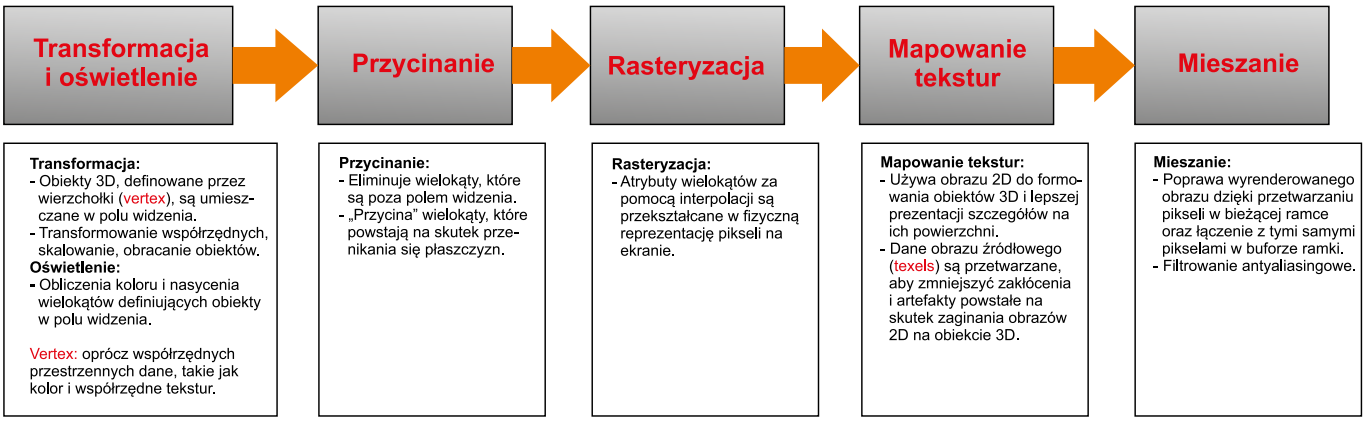
Współcześnie powszechnie wykorzystuje się procesory GPU. Korzystają z nich interfejsy graficzne, które są ukoronowaniem ponad 30 lat ewolucji – początkowo opracowywane i wytwarzane przez nowatorskie, dopiero powstające przedsiębiorstwa, zajmujące się grafiką komputerową. Współcześnie te układy są wytwarzane przez wiele firm i znalazły zastosowanie w komputerach PC, konsolach do gier, smartfonach i innych urządzeniach powszechnego użytku.

Dzięki zastosowaniu w urządzeniach przenośnych, na parametry procesorów GPU silnie wpływają wymagania stawiane układom SoC, między innymi jak najmniejszy pobór mocy. Licząc od pierwszego, monochromatycznego kontrolera obrazu CDP1861 z połowy lat 70-tych, obsługującego bardzo małą rozdzielczość wynoszącą zaledwie 62×128 pikseli, poprzez kartę EGA pracującą z rozdzielczością 640×350 pikseli w 16 kolorach, wprowadzoną do powszechnego użytku w połowie lat osiemdziesiątych (niewiele później wprowadzono karty VGA/SVGA o rozdzielczości do 800×600), musieliśmy czekać aż do wczesnych lat dziewięćdziesiątych na pierwsze produkty, które zaczęły kształtować architekturę współczesnych układów GPU.

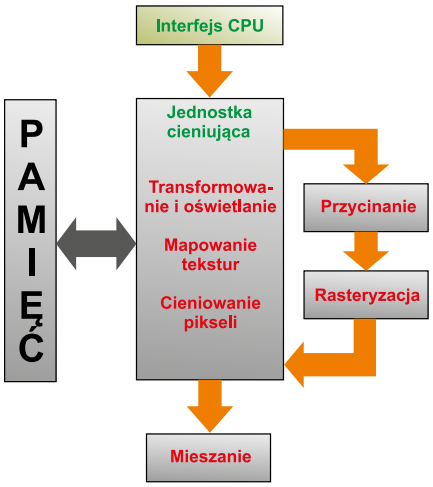
W 1996 roku na rynku już byli obecni główni gracze, którzy zdominowali na prawie dwie dekady świat grafiki 3D (większość z tych firm już nie istnieje). Robili oni ogromne postępy, najpierw napędzane przez potrzeby firm tworzących gry komputerowe, a z upływem czasu przez producentów niemal wszystkich użytkowanych przez nas urządzeń. W toku rozwoju GPU wykonano również API (OpenGL oraz Direct 3D), które wprowadziły znormalizowane sposoby korzystania z funkcji graficznych oferowanych przez GPU i ułatwiły tworzenie oprogramowania niejako niezależnie od sprzętu. Zastąpiły one produkty natywne, opracowane przez pionierskie firmy (z jedynym wyjątkiem – Glide firmy 3dfxpenGL przetrwało, ale zostało zbyt późno udostępnione na zasadach *open source*, co doprowadziło do „śmierci" tego bardzo lubianego pioniera grafiki 3D). Zapoczątkowany w ten sposób szybki rozwój doprowadził do opracowania przetwarzania strumieniowego[1], które na wystarczająco wysokim poziomie abstrakcji w przeważającej części jest takie samo, bez względu na rodzaj GPU. Schemat przetwarzania strumieniowego pokazano na **rysunku 1**.

Pierwsze procesory GPU były używane do przyśpieszenie przetwarzania danych, które było wykonywane po procesie „transformacji i oświetlenia" (T&L) wykonywanym przez CPU. Postępująca skala integracji układów scalonych pozwoliła na umieszczenie coraz większej i większej liczby tranzystorów w strukturze układu scalonego, co oczywiście wpłynęło na jego funkcjonalność. Umożliwiło to podjęcie kilku znaczących kroków:

- Przejecie przez GPU procesu „transformacji i oświetlenia" odciążyło jednostkę centralną i zapewniła znaczący wzrost wydajności komputerów PC. Zostało to osiągnięte w końcu lat dziewięćdziesiątych i szybko zaadoptowane przez wszystkich dostawców jednostek GPU-3D.
- Możliwość programowania przetwarzania wierzchołków (cieniowania wierzchołków) oraz przetwarzania pikseli (cieniowania pikseli) otwarły nowe możliwości dla programistów gier i interfejsów graficznych w znaczeniu tworzenia sceny oraz efektów. Przemieszczenie z implementacji stałych funkcji do przetwarzania strumieniowego zostało wykonane za pomocą programowalnych jednostek cieniujących (shaders):
  - Model programowania jest zdefiniowany przez „Shaders model", który zawiera specyfikację instrukcji, rejestrów oraz operacji.



Rysunek 1. Schemat przetwarzania potokowego w GPU



Rysunek 2. GPU z bazującą na przetwarzaniu potokowym jednostką cieniującą

– „Shaders model" oraz jego definicje były początkowo różne dla przetwarzania wierzchołków oraz pikseli, ale zostały zunifikowane, poczynając od Shader Model 4.0 (w Direct 3D)/ Unified Shader Model (OpenGL). Pierwsza generacja GPU zawierająca implementację zunifikowanego modelu cieniowania była dostępna na rynku w połowie roku 2000.
– Model danych ewoluował z liczb *integer* do zmiennoprzecinkowych o pojedynczej lub nawet podwójnej precyzji.

Początkowo jednostki cieniujące miały bardzo ograniczoną liczbę instrukcji. Współcześnie zakres ich zastosowań znacznie poszerzył się, a postęp technologiczny doprowadził do uzyskania dobrego balansu pomiędzy elastycznością wykorzystania ich możliwości i wydajności, a poborem mocy. Wprowadzono również zgodność ze standardem reprezentacji liczb zmiennopozycyjnych – IEEE 754. W wielu implementacjach jednostka cieniująca może być zintegrowana z DSP, silnie zoptymalizowana pod kątem operacji matematycznych, które muszą być wykonywane przez GPU. Podobne techniki i architektury są używane we współczesnych procesorach DSP (lub były używane w przeszłości) i w pierwszych rdzeniach procesorów GPU: VLIW, SIMD, z przetwarzaniem wektorowym.

Zunifikowany model jednostki cieniującej użyty w przetwarzaniu potokowym wykonywanym przez GPU pokazano na **rysunku 2**. Należy zauważyć, że tym razem pokazujemy pamięć, ponieważ typowo jednostki cieniujące mają dostęp do obszaru pamięci przeznaczonego dla obrazu lub pamięci systemowej przeznaczonej dla GPU i zintegrowanej w SoC.

Opracowanie zunifikowanych jednostek cieniujących zaowocowało nową jednostką miary względnej wydajności GPU: liczba rdzeni oraz częstotliwość ich taktowania. Te dwa parametry łącznie są typowo używane do określenia miary maksymalnej zdolności obliczeniowej GPU wyrażonej w liczbie operacji zmiennoprzecinkowych na sekundę (FLOPS – FLoating point Operations Per Second).

Jako przykład wsparcia dla GFX w mainstramowym SoC na **rysunku 3** pokazano trzy GPU zastosowane w Freescale i.MX6Q/D. W jego najbardziej rozbudowanej wersji w rodzinie i.MX6, procesor GPU wspierający wyświetlanie 3D ma 4 rdzenie cieniujące o wydajności 24 GFLOPS. Należy przy tym zauważyć obecność dwóch dodatkowych rdzeni przeprowadzających wyspecjalizowane operacje dla zwiększenia wydajności typowej aplikacji embedded: przyśpieszają one kompozycję warstw i grafiki wektorowej.

Zadania realizowane przez GPU i ich typy

Omawiane zagadnień związanych z GPU rozpocznijmy od wymienienia parametrów charakterystycznych dla typowych zadań przetwarzania realizowanych przez jednostkę cieniującą:

- Ten sam zestaw operacji (wyrażonych w sekwencjach instrukcji) może być użyty dla ogromnej liczby danych wejściowych. Obiekty składające się z tysięcy wielokątów są poddawane tej samej operacji T&L (transformacja i oświetlenie). Niektóre z nich zawierają piksele/teksle.
- Próbki danych wejściowych są przetwarzane jednocześnie.
- Przetwarzanie potokowe jest liniowe (występują jedynie niewielkie rozgałęzienia).
- Dane wejściowe poddawane przetwarzaniu potokowemu mają dobrą lokalizację przestrzenną. Umożliwia to dostęp oprogramowania do pamięci w sposób liniowy.
- Zadanie przetwarzania jest niewrażliwe na opóźnienia. Wystarczająca dla uzyskania obrazu o dobrej jakości częstotliwość ramek wynosi 25…30 FPS (lub 40…33 ms/ramkę). Tak długo, jak w czasie trwania ramki można zakończyć jej przetwarzanie, kolejność przetwarzania oraz opóźnienie nie są istotne.

Konstruktorzy procesorów GPU, gdy wybierają sposób, w jaki wykorzystają strukturę półprzewodnikową o danej wielkości/liczbie tranzystorów:

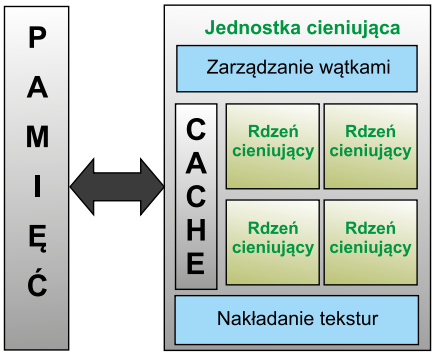
- Maksymalizują zdolność obliczeniową kosztem zdolności do gromadzenia danych.
- Maksymalizują możliwości równoczesnej obsługi wątków i równoległego przetwarzania danych.

Z tego powodu, pokazana na **rysunku 4**, zunifikowana jednostka cieniująca wbudowana w GPU składa się z pewnej liczby identycznych rdzeni, pamięci cache, jednostki zarządzania wątkami (więcej na ten temat będzie dalej) oraz jednostki dostępu do tekstur (wykonującej konwersję i próbkowanie). W wielu implementacjach każda jednostka cieniująca ma możliwości układu SIMD. Liczba jednostek różni się zależnie od rynku docelowego i wymaganej wydajności. Układy SoC, którym stawia się wymaganie niewielkiego poboru mocy, mają od jednego do kilku rdzeni cieniujących. Inaczej jest w kartach graficznych z „najwyższej półki", których procesory mogą mieć nawet kilka tysięcy rdzeni.

Łatwo domyślić się, że w razie potrzeby w procesorach GPU identyczna jednostka cieniująca jest replikowana wiele razy. Omawiane wielu identycznych kopii nie ma sensu i dlatego omówimy tylko jedną wyposażoną w cztery rdzenie cieniujące. Procesor GPU o najwyższej jakości miałby kilka zunifikowanych kilka podobnych bloków cieniujących, mających dziesiątki lub setki rdzeni.

Uwaga odnośnie do pamięci cache: lokalna pamięć cache jest nadal wymagana i obecna, ale współczynnik wielkości pamięci cache do zdolności przetwarzania jest znacząco poniżej tego, co pokazałoby typowe

Rysunek 3. Jednostka GPU procesora i.MX6



Rysunek 4. Blok zunifikowanych jednostek cieniujących

CPU. Okaże się to oczywiste, jeśli spojrzymy na fotografie struktur CPU i GPU próbując je porównać wizualnie.

Przy zaprezentowanej wyżej architekturze maksymalną wydajność osiąga się, gdy każda jednostka cieniująca wykonuje tę samą instrukcję na różnych punktach w zestawie danych (piksele na określonym obszarze ekranu wymagające tego samego przetwarzania, wertykale pewnych obiektów wymagających tego samego przetwarzania itp.). To minimalizuje proces pobierania i dekodowania instrukcji oraz umożliwia przetwarzanie danych z największą wydajnością.

Zadaniem bloku zarządzania wątkami jest efektywne kolejkowanie grup wątków, które są gotowe do uruchomienia. Grupowanie wątków oraz ich kolejkowanie jest bardzo ważne z perspektywy paradygmatu programowania użytego dla GPU. Jest on gruntownie odwzorowany w środowisku OpenCL. Typowo, dla każdego GPU jest pewna optymalna wielkość grupy wątków (np. wynikająca z liczby rdzeni jednostek cieniujących, które równolegle wykonują tę samą instrukcję) i powinno się ją wziąć pod uwagę podczas wykonywania aplikacji mającej osiągnąć jak największą wydajność.

Z punktu widzenia wydajności bardzo duże znaczenie mają instrukcje skoków, ponieważ zostaną one wykonane przez wszystkie rdzenie jednostki cieniującej. Z tego powodu, skoki potencjalnie obniżają efektywność całego przetwarzania wątków i powinno się ich używać ze szczególną ostrożnością.

Język diagnostyki sprzętu oraz framework – wprowadzenie do OpenCL

Pionierzy próbujący wykorzystać możliwości przetwarzania GPU rozpoczynali od OpenGL. Pomimo zastrzeżeń odnośnie do sposobu reprezentowania danych wejściowych i prymitywów, za pomocą OpenGL mogły być wykonane różne operacje na danych cyfrowych.

Przetwarzanie obrazu jest jedną z domen, gdzie takie techniki są często używane. Zamiana obrazów na tekstury pozwala na łatwe przeprowadzanie szeregu operacji filtrowania, jednak dla wypromowania jednostki GPU do statusu technologii mainstreamowej było potrzebne coś lepszego – standaryzowany, wpierany przez wielu (jeśli nie przez wszystkich) dostawców GPU, przyjazny oraz relatywnie łatwy w użyciu (przez doświadczonych programistów) framework. Taki jest OpenCL i gdy wprowadzono go na rynek, został dobrze przyjęty jako standard, jednak ocenę łatwości jego użycia pozostawiamy tym, którzy za jego pomocą opracowują programy. Warto przy tym wziąć pod uwagę dostępne wsparcie.

OpenCL jest specyfikacją API wykonaną przez Khronos Group. Pozwala ona na asynchroniczne programowanie wielordzeniowe dla różnych platform – heterogenicznych środowisk komputerowych. Współcześnie ten framework umożliwia tworzenie oprogramowania dla wielu modeli GPU, CPU (typowo wykorzystujące możliwości SIMD), procesorów DSP oraz układów FPGA. Pierwsza rewizja specyfikacji (1.0) została udostępniona w 2009 r. Najnowsza, dostępna w trakcie pisania tego artykułu wersja 2.0, została opublikowana w listopadzie 2013 r. Zawiera ona profile: pełny FP (*Full Profile*) oraz uproszczony EP (*Embedded Profile*). Niektóre opcje, które są obowiązkowe w profilu FP, są opcjonalne w EP. Z drugiej strony, pewne ograniczenia i obowiązkowe opcje profilu EP są „rozluźnione" w FP.

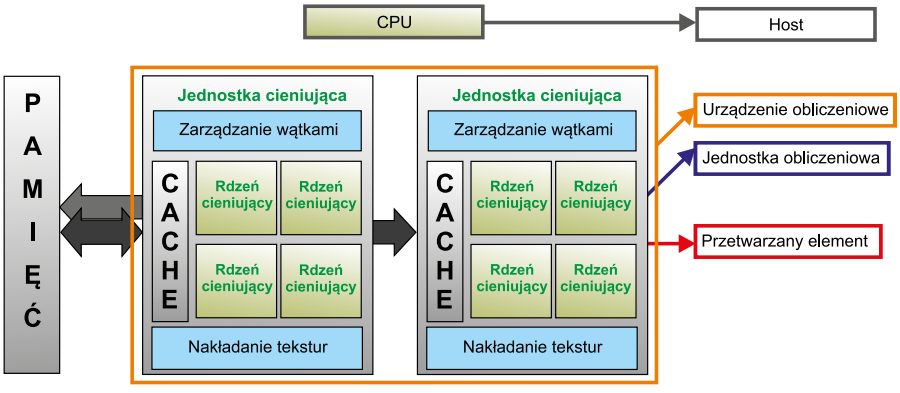
To, co próbuje się osiągnąć za pomocą OpenCL, jest dużym wyzwaniem: zaoferować przenośny framework, który pozwoli na efektywne i bezproblemowe wykorzystanie sprzętu o różnej architekturze, o różnych możliwościach przetwarzania danych, instrukcji i wątków. Dla przykładu, w GPU typowo przetwarza równolegle instrukcje i dane, natomiast wielordzeniowe CPU/SIMD (po prostu współczesne CPU, instalowane w komputerze PC lub w urządzeniu przenośnym) – dane i wątki.

OpenCL wprowadza podział urządzeń na „Host" oraz „Compute device". „Compute device" jest pewną abstrakcyjną liczbą maszyn przetwarzających (może ich być więcej niż jedna w systemie), podczas gdy „Host" jest częścią systemu najlepiej dopasowaną do uruchomienia oprogramowania i kontroli złożonego kodu. Host zarządza wykonywaniem aplikacji, zasilaniem oraz synchronizacją wątków realizowanych przez „Compute device". Oba komponenty typowego systemu – „Host" oraz „Compute device" – pokazano na **rysunku 5**. Należy przy tym zwrócić uwagę, że chociaż pokazaliśmy tylko jedno „Compute device", w systemie może być ich wiele i mogą one być różnych typów (np. wielordzeniowe GPU i/lub wiele DSP, i/lub wiele FPGA zarządzanych przez to samo CPU).

Typowo „Host" jest programowany w języku C/C++, ale jego API pozwala na wirtualne powiązanie z jakimkolwiek innym językiem programowania, np. Java, Python, Tuby, OCaml. Dodatkowo, „Host" prezentuje warstwy API (zapewniając warstwę abstrakcji ponad urządzeniami „Compute device" w systemie) i środowisko uruchomieniowe, które pozwala na zarządzanie urządzeniami przetwarzającymi (wykrywanie, enumerację i konfigurowanie, zapisywanie procedur do realizacji, synchronizację, alokowanie zasobów). „Compute Device" jest programowane z użyciem OpenCL C (podzbiór ISO C99 z rozszerzeniami języka). Ograniczenia (stała wielkość obszarów, brak rekurencji, brak wskaźników do funkcji, brak pól bitowych) oraz rozszerzenia języka (nowe słowa kluczowe i pragmy, obszary notacji i inne) są wprowadzone dla ułatwienia zadań oraz zrównoleglenia wektorów w środowisku uruchomieniowym i podczas procesu kompilacji. Funkcje przeznaczone do uruchomienia na urządzeniu „Compute device" są nazywane „Kernel".

Aby zrozumieć dalszą część artykułu, musimy wytłumaczyć podstawowe pojęcia związane z urządzeniem przetwarzającym – Compute Device. Są to przede wszystkim:

- **Element przetwarzany**: najmniejsza jednostka przetwarzana. Typowo będzie to operacja na wektorze, a w naszym przykładzie, opisanym wyżej, jest pojedynczy rdzeń jednostki cieniującej. Później zobaczymy, że istnieją również inne architektury, w których element przetwarzany jest mapowany różnicowo.
- **Jednostka przetwarzająca**: najmniejszy blok używany na poziomie wątku. Typowo jest to grupa elementów procedowanych, która jest umieszczona



Rysunek 5. System z punktu widzenia OpenCL

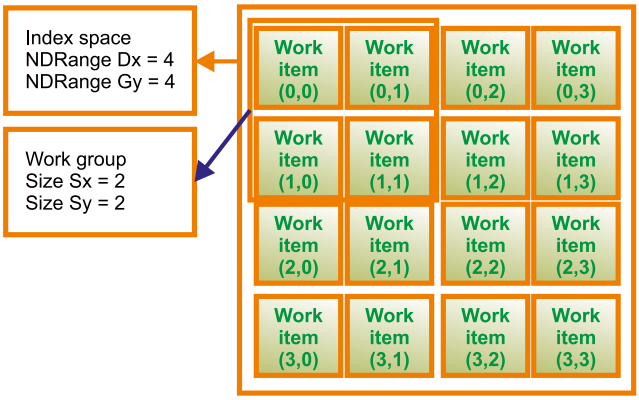
pomiędzy jednostką zarządzającą pojedynczym wątkiem, a jednostką zarządzającą. Oznacza to, że ta grupa będzie przetwarzana krok po kroku w tym samym strumieniu instrukcji. W naszym przykładzie, jednostka obliczeniowa jest reprezentowana przez blok zunifikowanych jednostek cieniujących.

Ważne jest zrozumienie przebiegu procesu przetwarzania w OpenCL. Spróbujmy użyć jako przykładu matrycy (OpenCL może przetwarzać macierze jedno-, dwu- i trzywymiarowe), która jest przetwarzana przez kernel.

Ogólnie, macierz wejściowa do przetwarzania jest nazywana „Index space" (**rysunek 6**) i ma wielkość „NDRange". W naszym wypadku mamy Gx=4 i Gy=4 (uwaga – wymiary macierzy *x*, *y*, *z* nie muszą być równe). Macierz „Index space" będzie przetwarzana w segmentach „Work group" będących grupą elementów roboczych, które będą obrabiane przez co najmniej jedną jednostkę obliczeniową (wiele jednostek obliczeniowych może być używanych równolegle). Upraszczając, możemy spodziewać się, że liczba elementów roboczych w grupie roboczej odpowiada liczbie elementów przetwarzanych w rzeczonej jednostce obliczeniowej, a zatem w naszym przykładzie pokazujemy grupę roboczą zawierającą 4 elementy (rzeczywistość jest nieco bardziej złożona). Każdy element macierzy wejściowej (część grupy roboczej) nazywany „Work item" będzie przetwarzany przez instancję kernela operującą na elemencie przetwarzanym i jest identyfikowana za pomocą unikalnego identyfikatora „Global ID". Jest on nadawany przez pozycję elementu roboczego w przestrzeni „Index" grupy roboczej. Każda grupa robocza jest unikalna i ma ściśle określoną pozycję w przestrzeni „Index", więc identyfikator „Global ID" każdego elementu roboczego może być wyznaczony przez jego pozycję w grupie roboczej. Należy również wprowadzić koncepcję „Local ID", który jest indeksem elementu w obrębie danej grupy roboczej. Ta koncepcja bazuje na podziale przetwarzania w obrębie grupy dla lepszego użycia pamięci lokalnej.

Istotną składową procesu przetwarzania jest optymalne wykorzystanie pamięci. W OpenCL zdefiniowano następujące typy pamięci:

- **Global Memory**: dostępna dla „Host" i wszystkich „Compute device" w systemie. Fizycznie może to być pamięć systemowa lub pamięć wbudowana w SoC, jeśli tylko może być udostępniana poza SoC.
- **Constant Memory**: ma te same właściwości jak Global Memory, ale jest przeznaczona tylko do odczytu.
- **Local Memory**: pamięć dostępna tylko dla elementów roboczych należących do grupy roboczej.
- **Private Memory**: dostępna dla pojedynczej instancji kernela/elementu roboczego. Nie jest widziana przez inne elementy robocze.



Rysunek 6. Dane wejściowe w postaci macierzy „Index space"

Przekazywanie danych pomiędzy „Host" i „Compute device" odbywa się za pomocą:

- API, które pozwala Hostowi na tworzenie obiektów w pamięci Global Memory, dostępnych dla „Hosta" oraz „Compute device".
- Mapowanie regionów pamięci, takich jak te obszary pamięci są dostępne do zapisu/odczytu dla Hosta oraz jednostek Compute Device, pozwalając na wymianę danych, jeśli to potrzebne. Należy zauważyć, że wydajność mapowania w porównaniu do kopiowania może różnić się w zależności od wypadku, zależnie od sposobu obsługi pamięci cache przez Hosta oraz prędkości interfejsu.
- API pozwala na kontrolę operacji na pamięci oraz synchronizację za pomocą zdarzeń.

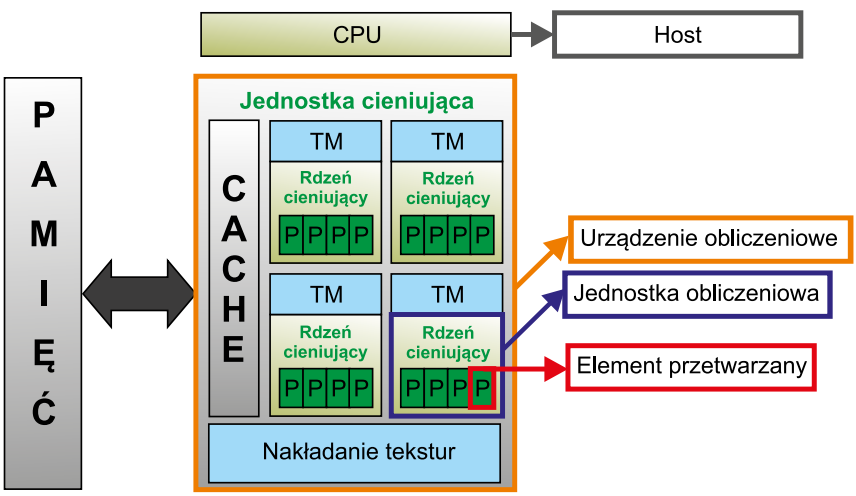
Teraz mamy wszystkie elementy procesu przetwarzania: instancję kernela, dane wejściowe (przestrzeń Index) zorganizowane w postaci elementów roboczych, które są zatrudniane przez element przetwarzany, możliwość wymiany danych pomiędzy „Host" i „Compute device" oraz możliwość ich synchronizacji. Podsumowując, przyjrzyjmy się szczegółom budowy GPU 3D typu GC2000 (**rysunek 7**) wbudowanemu w strukturę SoC i.MX6 z perspektywy OpenCL, jako platformy, która będzie użyta w kolejnych rozdziałach naszej aplikacji OpenCL typu „Hello world" (opiszemy ją w drugiej części tego artykułu).

Porównując z ogólną architekturą, której użyliśmy w przykładzie, każda jednostka cieniująca GC2000 jest procesorem SIMD mieszczącym 4 liczby zmiennopozycyjne o pojedynczej precyzji (SPFP). Dlatego też każda jednostka cieniująca może przetwarzać 4 elementy oraz ma niezależny układ zarządzający wątkami, pozwalający uruchomienie dla nich niezależnej instancji kernela oraz reprezentujący jednostkę obliczeniową. Compute device (którym jest cały rdzeń GPU) pozwala na równoległe przetwarzanie 16 elementów danych. W związku z tym wielkość grupy roboczej zapewniającej uzyskanie maksymalnej wydajności to 16 elementów. Jakakolwiek grupa mająca mniej niż 4 elementy nie będzie przetwarzana efektywnie, ponieważ co najmniej jeden element nie będzie procedowany.

Penisoara Nicusor
Freescale Semiconductor

(Endnotes)

[1] Przetwarzanie strumieniowe lub uniwersalny GPU polega na wykorzystaniu GPU w roli zmodyfikowanej wersji procesora strumieniowego, który udostępnia ogromna moc operacji zmiennoprzecinkowych w potoku nowoczesnego, sprzętowego przetwarzania graficznego oraz dla uniwersalnych obliczeń naukowych. Osiąga się to przez rezygnację z zaszycia operacji obliczeniowych w sprzęcie wyłącznie dla potrzeb grafiki.



Rysunek 7. GC2000 3D GPU z punktu widzenia OpenCL