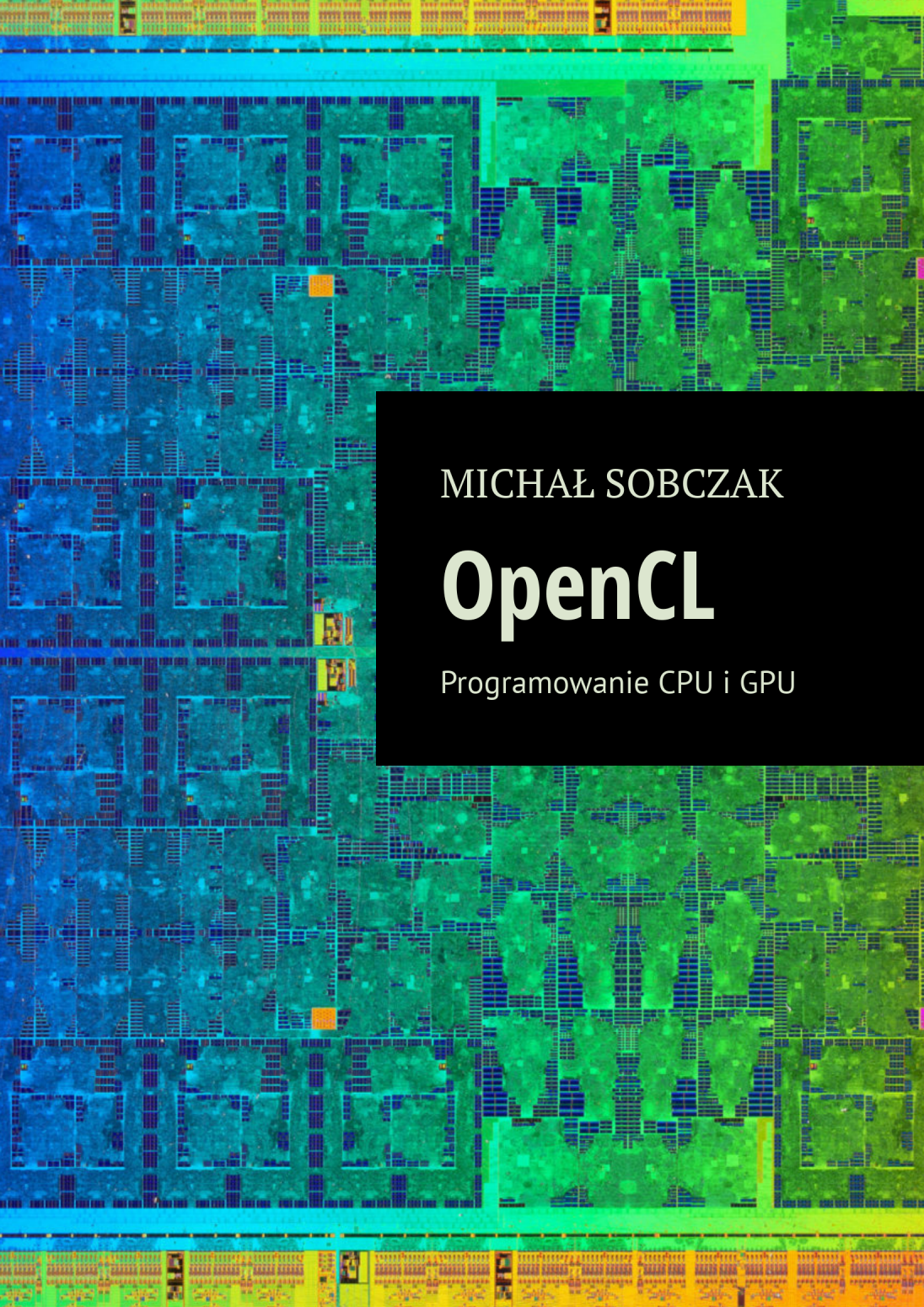


MICHAŁ SOBCZAK

OpenCL

Programowanie CPU i GPU

Michał Sobczak

OpenCL

Ridero

2022

ISBN 978-83-8273-803-2
Książka powstała w inteligentnym systemie wydawniczym Ridero

Spis treści

1. WPROWADZENIE

Mamy rok 2022. Minęła ponad dekada od pojawienia się OpenCL. Omówię standard, przykładowy sprzęt, a także zaprezentuję rzeczywiste przykłady, które można wykorzystywać obecnie zarówno na historycznym jak i najnowszym sprzęcie, co ma pokazać swego rodzaju uniwersalność tego rozwiązania. Przy okazji zobaczymy jak ewoluowała technologia na przestrzeni ostatnich lat i czy teoria odpowiada praktyce.

Udostępnione przykłady pokażą, że dzięki stosowaniu standardu OpenCL możemy uzyskać nawet **50-krotne przyspieszenie działania algorytmu (na korzyść GPU)** porównując czas pracy CPU i GPU. Zacznijmy jednak od podstaw, tak aby zrozumieć, dlaczego tak się dzieje…

[W] Jednym z aspektów **wysokowydajnego przetwarzania** jest wykorzystanie karty graficznej. Możemy zastosować do tego standard OpenCL (*Open Computing Language*), który został opracowany i opublikowany w **2009** roku. Standard utrzymywany jest przez **Khronos Group**. Inicjatorem przedsięwzięcia jest firma Apple. Wersja 3 standardu została wydana w 2020 roku. Jest to API, które **umożliwia wywoływanie kodu z maszyny typu host na zuniwersalizowanych urządzeniach przetwarzania danych**. Takim urządzeniem może być zarówno **GPU,** ale również **CPU**. Z punktu widzenia kodu programu, który ma wykonać obliczenia, napisanym w zbiorze języka **C99/C++11**, nie ma to zatem szczególnego znaczenia.

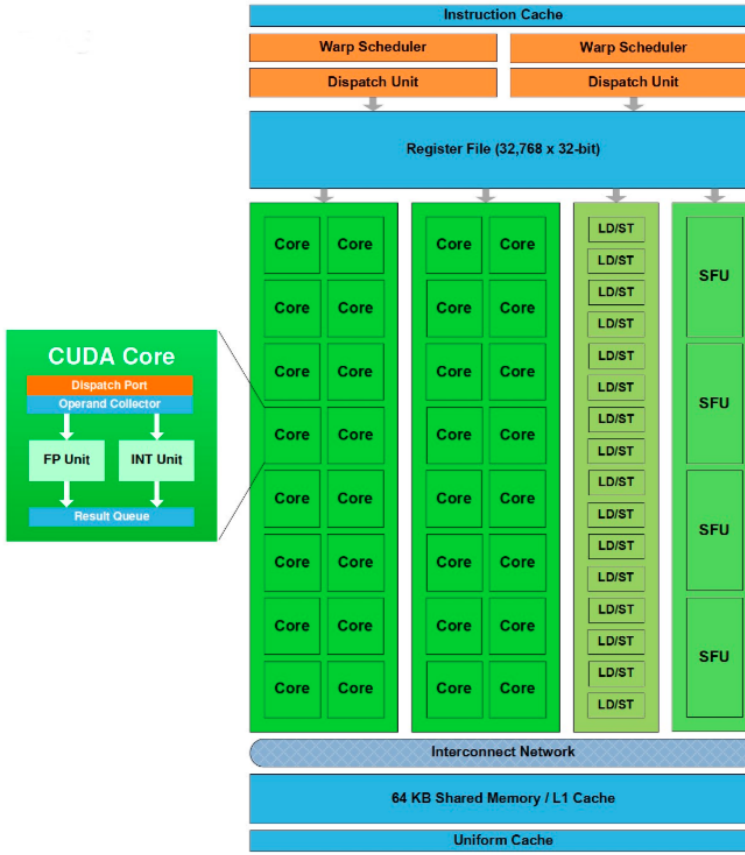
Uwaga: poza firmą-inicjatorem, nad standardem pracowało i pracuje wiele organizacji takich jak AMD, ARM, Intel, Nvidia czy Samsung oraz kilka mniej znanych szerokiej publiczności jak Altera, Creative, Imagination, Qualcomm, Vivante, Xilinx i ZiliLABS.

2. STANDARD

OpenCL jako definicja

[W] [219] OpenCL jest **tylko i wyłącznie standardem**, który jest uzależniony od **konkretnych implementacji**. Poza implementacjami na **CPU** i **GPU** występują również takowe na **DSP** oraz **FPGA**. Standard ten umożliwia **uruchamianie zadań operujących na danych, korzystać z maksymalnych możliwości współbieżności**, równoległego przetwarzania. Główną zauważalną korzyścią ze stosowania innych urządzeń docelowych niż CPU, np. GPU, jest fakt posiadania przez nie **wysoce wyspecjalizowanych jednostek obliczeniowych**. Nie możemy tutaj mówić wprost o rdzeniach, gdyż architektura takiego urządzenia jest nieco inna od tradycyjnie pojmowanej dotyczącej CPU. Jako rdzenie przeważnie traktuje się **jednostki obliczeniowe będące procesorami strumieniowymi** (SP), które są grupowane w **multiprocesory strumieniowe** (SMP), a te z kolei zgrupowane w **klastry przetwarzania wątkowego** (TPC).

Uwaga: stosowane słownictwo dotyczy głównie kart graficznych, w szczególności kart NVIDIA

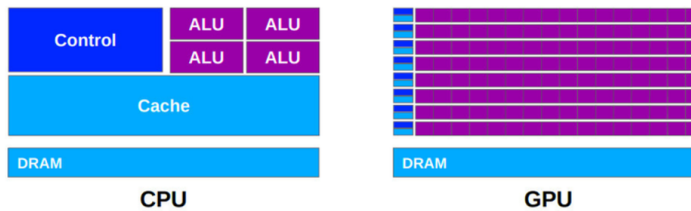


Rys. 1 — Budowa urządzenia klasy NVIDIA [233]

2.1. GPU a CPU

Różnice pomiędzy typami urządzeń

Różnice w architekturze CPU względem GPU są **znaczące**. Dla CPU większe znaczenie ma **uniwersalność rdzeni**. Mają one **małą tolerancję na opóźnienia i niską przepustowość**. GPU z kolei mają **wysoką tolerancję na opóźnienia, ale dużą przepustowość**. Dla CPU występuje potrzeba posiadania pamięci typu *cache* o znacznych rozmiarach, większych niż w przypadku GPU z racji większego nastawienia na przetwarzanie zadań w sposób uniwersalny niż samych wyłącznie danych.



Rys. 2 — Porównanie CPU i GPU [302]

Pojawia się tutaj pojęcie **SIMD (*Single Instruction Multiple Data*)**. Dzięki takiej zmianie podejścia (względem dotychczasowego, przed pojawieniem się OpenCL), możliwe jest nastawienie na posiadanie większej ilości jednostek obliczeniowych w GPU kosztem posiadania mniejszej ilości pamięci podręcznej.

2.2. Wątki i synchronizacja

Zrównoleglanie przetwarzania danych

[302] Obecne, tj. 2022 GPU posiadają **tysiące jednostek**, CPU najwyżej **kilkadziesiąt**. Dla CPU wątki w programach są synchronizowane właśnie w programach i przez to jest to kosztowne czasowo. Wymagane jest na to wiele cykli zegara. Dla GPU ten problem nie występuje, ponieważ nie ma potrzeby, ale i możliwości synchronizowania zadań, ponieważ w dużej mierze **operujemy wyłącznie na zrównoleglaniu danych, a nie samych zadań**. Wątki w przypadku GPU są **niezależne**. Wątki w zakresie GPU są **zarządzane z poziomu sprzętowego**. Poza samą różnicą w podejściu mamy dostępnych w GPU wiele różnych technik optymalizacyjnych takich jak lokalne pamięci podręczne, które mogą, ale nie muszą być współdzielone pomiędzy wątkami i ich grupami.

2.3. Zunifikowane jednostki cieniujące

Zmiana architektury zwiększająca wydajność

[302] Poniżej znajduje się krótki rys historyczny **prezentujący drogę dojścia do wprowadzenia standardu OpenCL** poprzez uniwersalizację jednostek przetwarzania danych:

*i. W **1995** roku przeciętny układ umożliwiał obliczenie 50 tys. trójkątów na sekundę (TPS), 1 mln operacji na pikselach (POPS) na sekundę. Układ taki posiadał około 1 mln tranzystorów.*

*ii. W **1999** roku układy umożliwiały przetwarzanie nawet 15 mln TPS, 480 mln POPS. Układy posiadały około 20 mln tranzystorów.*

*iii. W **2001** roku układy typu NVIDIA GeForce 3 umożliwiały kolejno 100 mln TPS, 1 miliard POPS i posiadały około 50 mln tranzystorów. Wprowadzono również wsparcie dla cieniowania (Shader Model 1.0).*

*iv. W roku **2003** seria FX to już DirectX w wersji 9.0, języki HLSL, Cg, GLSL. Przeciętna karta graficzna umożliwia 200 mln TPS, 2 mld POPS, posiada 120 mln tranzystorów.*

*v. Rok **2004** to Shader Model 3.0 i seria NVIDIA GeForce 6 posiadająca 200 mln tranzystorów, umożliwiająca 600 mln TPS, 12 mld POPS. Wprowadza się również wsparcia dla HDR.*

vi. Wraz z Shader Model 4.0 pojawia się przetwarzanie w oparciu o CUDA i procesory strumieniowe. Układ NVIDIA G80 to 681 mln tranzystorów.

[302] Wprowadzenie zunifikowanej architektury obliczeniowej powoduje wiele korzyści, które dotąd nie były możliwe

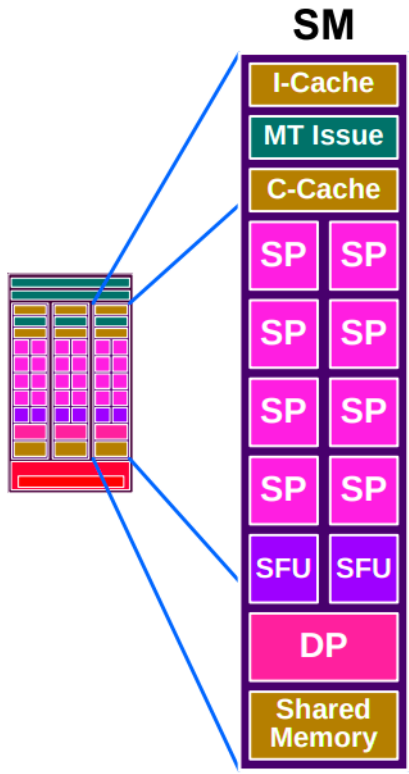
do osiągnięcia. Chodzi mianowicie o **jednolitość i uniwersalność jednostek przetwarzających dane**. Poprzednie architektury, opierające się na rozdzielaniu sprzętowych funkcji analogicznie do rozwiązań na warstwie programowej, powodowały **nierównomierne rozłożenie obciążenia** w zależności od jego charakterystyki. Tym samym, dzięki zastosowaniu nowego podejścia, **możliwe jest zarówno obsłużenie obliczeń na geometrii jak i na cieniowaniu**.

Możliwości obliczeniowe dzięki takiej unifikacji pozwoliły na osiągnięcie **znaczącego przyrostu mocy obliczeniowej** z racji zmiany procesu technologicznego. Zarówno **ilość operacji zmiennoprzecinkowych jak i przepustowość pamięci znacznie wzrosły**, w szczególności porównując ten aspekt do rozwiązań opartych wyłącznie na CPU.

2.4. Multiprocesor strumieniowy (SMP)

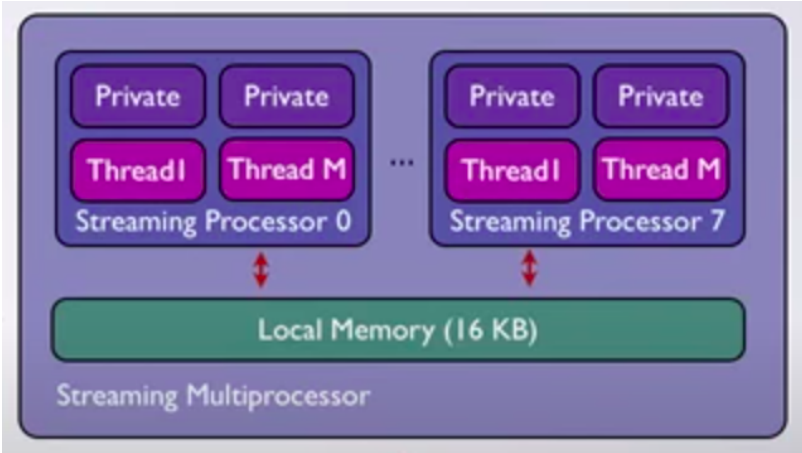
Zgrupowanie jednostek obliczeniowych

[302] Każdy typowy SMP poza samymi **jednostkami obliczeniowymi** (SP), składa się również z **szeregu innych komponentów**. Są to jednostki specjalne służące do **funkcji trygonometrycznych** (SFU). Występuje również **jednostka obliczeń zmiennoprzecinkowych** (DP), 2 pamięci podręczne (*I-Cache, C-Cache*) oraz **pamięć współdzielona** (*Shared Memory*).



Rys. 3 — Multiprocesor strumieniowy [302]

[009] W ramach pojedynczego SMP dostęp również jest możliwy do pamięci **współdzielonej pomiędzy wszystkimi wątkami w ramach tego konkretnego multiprocesora**. Tutaj jednak mamy do czynienia z niewielkimi rozmiarami takiej pamięci podręcznej, np. 16 lub 32 KB.



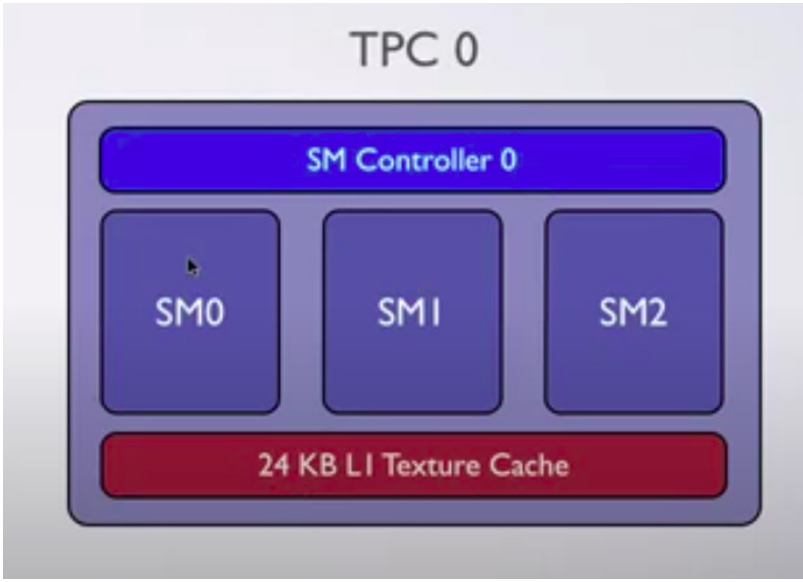
Rys. 4 — SMP z punktu widzenia wątków

Uwaga: Na okładce książki zamieszczono fotografię procesora graficznego NVIDIA GP104, stosowanego m.in. w kartach GeForce GTX 1060.

2.5. Strategia SIMD/T a Warp

Zasady przetwarzania danych i zadań

[010] Aby zarządzać i efektywnie wykorzystywać zarówno dużą ilość wątków jak i wysoką przepustowość konieczne jest **ustalenie zasad przetwarzania**. Istotnym pojęcie w tym przypadku jest tak zwany *Warp*, który można przetłumaczyć jako swego rodzaju **osnowa**, których istotną właściwością jest **szerokość** oraz to, że **wątki w ramach tej osnowy wykonują taką samą instrukcję w danym momencie**.



Rys. 5 — Thread Processing Cluster

Sprzętowy układ pobiera w jednym momencie instrukcje do wykonania i dlatego są one dostępne i możliwe

do uruchomienia **w tym samym czasie**. Osnowy (*Warp*) są podstawową **jednostką planowania w czasie wykonania wątków**. Strategia SIMT (*Single Instruction Multiple Threads*) jest tutaj bardziej odpowiednim sformułowaniem **kładącym nacisk na wątki konkretnie, a nie na sam aspekt przetwarzania danych** przez funkcje w *kernelach* przekazywanych do tych wątków.

2.6. Model pamięci

Uniwersalne poziomy w porównaniu do fizycznej implementacji

[W] Zapoznanie się z aspektem sprzętowym zdecydowanie pomoże zrozumieć **zasadę działania obliczeń w OpenCL**. W szczególności jeśli chodzi o budowę *kerneli*, charakterystykę danych i ich rozkładu przy przekazywaniu do przetwarzania. Mamy zatem następujące **poziomy pamięci**:

i. ***CPU*** *i jego pamięci podręczne (Lx)*

ii. ***pamięć hosta****, jest to DRAM komputera z CPU*

iii. pamięć ***DRAM*** *karty graficznej*

iv. ***pamięci podręczne*** *karty graficznej*

Z punktu widzenia OpenCL mamy **następujące poziomy**:

i. pamięć ***globalna****, wysokich opóźnień, współdzielona pomiędzy wszystkimi jednostkami*

ii. pamięć na wartości ***stałe****, tylko do odczytu, zapisywalna wyłącznie z hosta*

iii. pamięć ***lokalna****, współdzielona w grupie jednostek*

iv. pamięć ***prywatna****, swego rodzaju rejestr, przypisana per jednostka*

Nie ma konieczności aby dana implementacja OpenCL zawierała wszystkie ww. poziomy. Należy mieć to na względzie podczas przygotowywania i optymalizowania kodu. Wymagane jest aby uwzględnić, na jakim sprzęcie dany kod powinien być uruchamiany aby takie przetwarzanie było **maksymalnie efektywne**.

2.7. Pojęcia i zasada działania

Podstawowa terminologia podziału wsadu zadaniowego

[009] **Poprzez wątek w terminologii NVIDIA rozumiemy *Work-Item*. Poprzez blok wątków rozumiemy *Work-Group***. Wcześniej wspomniany *Warp*, który można określić jako osnowa, to sposób faktycznej organizacji bloków wątków. **Każda taka osnowa to 32 wątki**. Każdy z nich (tj. wątków), wywoła *kernel* przekazany do układu obliczeniowego. **Tylko wątki w tej samej grupie mogą wymieniać się danymi z użyciem pamięci o szybkim dostępie**. Tylko tak zgrupowane zadania mogą być wzajemnie synchronizowane.

Każda instrukcja przekazana jako *kernel* wykonywana jest w tym samym momencie przez wątki pracujące w ramach osnowy (*Warp*). **Pamięć lokalna to np. 16 lub 32 KB per SMP (procesor multistrumieniowy), a zatem współdzielona poszczególne procesory strumieniowe w jego ramach**. Wątki mogą współpracować przy ładowaniu danych do pamięci lokalnej. Wiele wątków uzyskuje dostęp do tej pamięci w tym samym momencie, zatem rozwiązanie sprzętowym jest podzielenie tej przestrzeni pamięciowe na tak zwany banki. **Równoczesny dostęp do tego samego banku przez dwa różne wątki może doprowadzić do konfliktu**.

> *Uwaga: Chociaż wprowadzone pojęcia mogą wydawać się skomplikowane, istotnym jest zrozumieć rozkład pamięci i zasady uzyskiwania do niej dostępu, aby możliwie najbardziej efektywnie korzystać z wszystkich korzyści wynikających z prawidłowego obciążenia takiego układu.*

3. PAKIET BAZOWY

Podstawa dla dalszych przykładów

W poprzednim rozdziale przedstawiony został standard OpenCL pod kątem jego założeń. Zaprezentowałem główne różnice w architekturze GPU względem CPU. Co prawda, programy przeznaczone dla OpenCL można uruchamiać zarówno na uniwersalnych procesorach jak i kartach graficznych, to znajomość ich budowy **uwypukla różnice nie tylko w budowie, ale i wydajności, jaką możemy uzyskać na poszczególnych typach urządzeń**. Stąd większy nacisk na pojęcia typu zunifikowane jednostki cieniujące czy multiprocesory strumieniowe.

W tym rozdziale przedstawię **bazowy pakiet przygotowany w języku Java**, który będzie podstawą do pisania *kerneli*, czyli programów w języku C przeznaczonych do uruchamiania bezpośrednio na urządzeniach obliczeniowych implementujących standard OpenCL.

3.1. Konfiguracja projektu i modułów

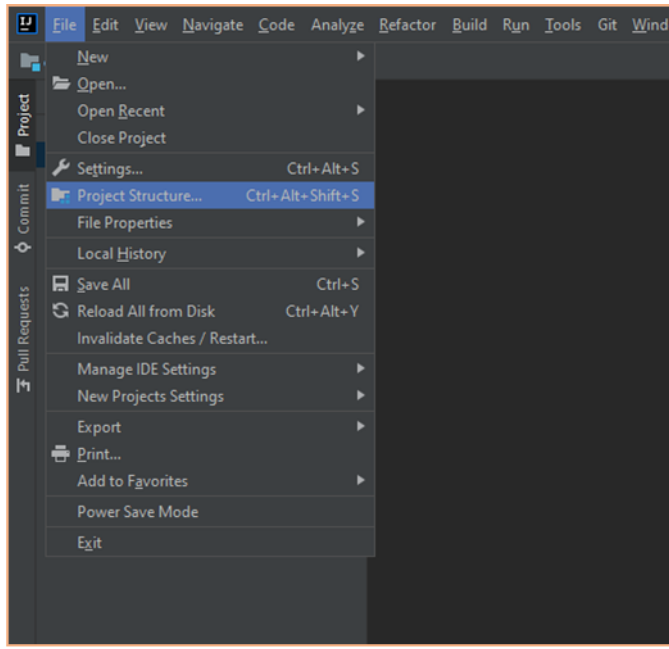
Polecanym przeze mnie IDE (środowisko programistyczne) do tych ćwiczeń jest IntelliJ IDEA. Jest to **zaawansowany edytor**, w którym nawet wersja podstawowa (nieodpłatna) jest bardzo wygodna w użyciu. Zaczynamy od zainstalowania edytora, a następnie skopiowania projektu, który znajduje się pod adresem:

github.com/michalasobczak/simple_hpc

w folderze:

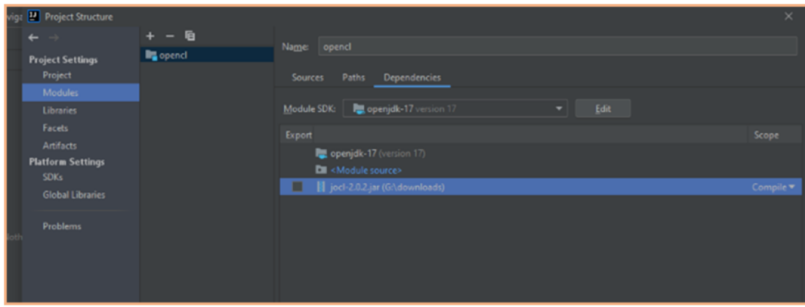
opencl

Znajduje się tam w pierwszej kolejności pakiet bazowy, tj. w folderze ***src*** oraz pakiety z przykładami, z pozostałych folderach.



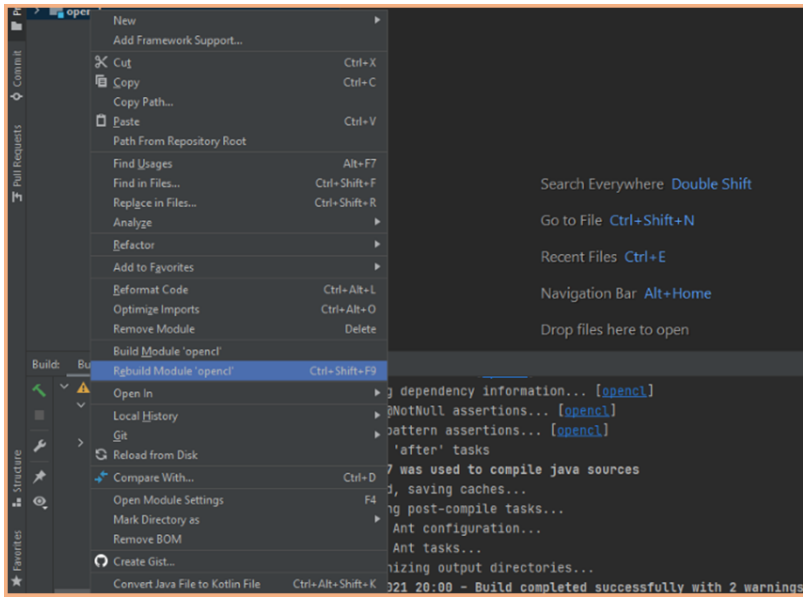
Rys. 6 — IntelliJ IDEA, widok główny

Projekt należy otworzyć. Następnie z poziomu *File — Project Structure* w sekcji *Modules* należy skonfigurować moduł określony jako ***opencl***. Wymagane jest podanie SDK języka Java, może to być wersja ***openjdk-17***. Należy też wskazać bibliotekę *jocl*, może być w wersji 2.0.2 w postaci pliku JAR.



Rys. 7 — Konfiguracja struktury projektu

Na początek warto usunąć wszystkie moduły poprzez ich zaznaczenie i wciśnięcie przycisku *Delete*. Nie są one na tym etapie potrzebne. W pierwszej kolejności należy zbudować moduł ***opencl***. Jeśli posiadamy zainstalowane SDK, jak również mamy wskazaną bibliotekę ***jocl***, wówczas moduł powinien się skompilować bez żadnych przeszkód.



Rys. 8 — Zbudowanie modułu podstawowego

W IntelliJ IDEA mamy możliwość **automatycznego pobrania z internetu SDK**, jeśli takowego nie posiadamy zainstalowanego. Nie musimy zatem pobierać tej zależności samodzielnie, możemy zrobić to bezpośrednio z poziomu edytora.

3.2. Struktura

Należy zacząć od tego, że pakiet bazowy nazwany jest:

com. michalasobczak. opencl

Zawiera on szereg klas, które są opisane poniżej. Jest to **szkielet dla wszystkich aplikacji**, w których będą prezentowane konkretne już zastosowania.

Utils to najprostsza klasa, która zawiera jedną metodę służącą do logowania na wyjście. Wraz z przechodzeniem do kolejnych przykładów będzie można tutaj dodawać kolejne metody, które nie kwalifikują się ani do konfiguracji platformy *PlatformConfigurationSet* ani klasy bieżącego uruchomienia *RuntimeConfigurationSet*.

RuntimeConfigurationSet to klasa odpowiadająca za konfigurację uruchomieniową. Zawiera miejsce na zbiory przechowujące dostępne platformy i urządzenia. Daje możliwość **wskazywania, na których z nich chcemy uruchamiać kod *kernela***. Odpowiadają za to metody *selectPlatform* oraz *selectDevice*.

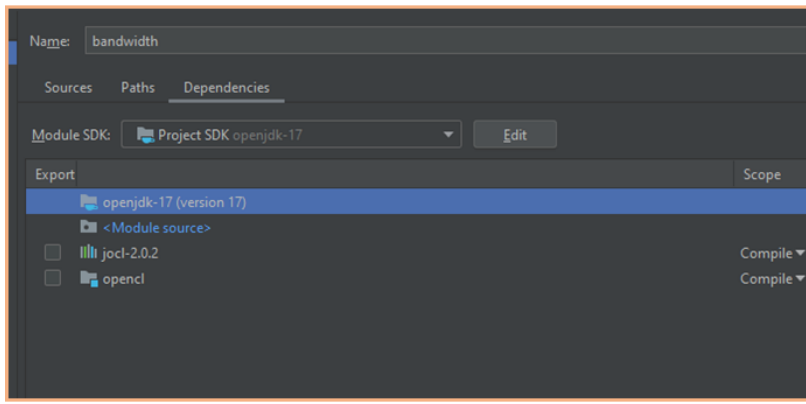
PlatformParametersSet, ostatnia klasa z pakietu bazowego odpowiada za ustawienia związane z wyborem i działaniem platformy uruchomieniowej. Za **otrzymanie listy platform i urządzeń** odpowiada metoda *getPlatformsAndDevices*. Aby uzyskać **informacje dotyczące urządzenia**, takie jak nazwa, typ urządzenia, ilość jednostek obliczeniowych itd., wywołuje się metodę *getDevicesInfo*. Metodami pomocniczymi w pobieraniu informacji ze sterownika OpenCL są *getInt*, *getInts*, *getLong*, *getLongs*, *getString*, *getSize* oraz *getSizes*.

3.3. Przykład podstawowy

Za przykład podstawowy posłuży moduł *bandwidth*

com.michalasobczak.bandwidth

Dodajemy go do projektu w konfiguracji struktury projektu. Jako zależność wskazujemy bibliotekę *jocl* oraz moduł bazowy, czyli *opencl*.



Rys. 9 — Import modułu bandwidth

Poza modułem bazowym, **przykład posiada kilka własnych klas, które odpowiedzialne są za uruchomienie programu w języku C zwanego *kernelem***. Aby taki *kernel* przygotować należy go załadować i odpowiednio skonfigurować.

KernelConfigurationSet to klasa, której zadaniem jest utrzymywanie miejsca na dane wejściowe oraz dane wyjściowe z programu, a także parametryzację *kernela*. Każdy przykład będzie posiadał swoją własną implementację tej klasy, gdyż zawiera ona szczegóły specyficzne dla konkretnego przykładu,

takie jak charakterystyka danych wejściowych i wyjściowych, czy też sposób przetwarzania *kerneli*.

Za **obsługę danych** odpowiadają metody *getSrcArrayA*, *getDstArrayA* oraz *initializeSrcArrayA* i *initializeDstArray*. Za inicjalizację wejściowych danych losowych odpowiada metoda *generateSampleRandomData*. W obszarze danych mamy jeszcze metodę pomocniczą *printSrcArray*, służącą do wydruku danych wejściowych, jeśli jest ich relatywnie niedużo.

Do **inicjalizacji kontekstu pracy** (Listing 1) wymagane jest **przygotowanie** pola *contextProperties* typu *cl_context_properties* pochodzącego z biblioteki *jocl*.

```
public void createContext() {
    this.contextProperties = new cl_context_properties();
    this.contextProperties.addProperty(CL_CONTEXT_PLATFORM, RuntimeConfigurationSet.platform);
    this.context = clCreateContext(this.contextProperties, 1,
          new cl_device_id[]{RuntimeConfigurationSet.device}, null, null, null);
}
```

Listing 1 — Przygotowanie pól kontekstu pracy

Następnie ten parametr w postaci platformy jest wykorzystany do **utworzenia kontekstu** metodą *clCreateContext,* gdzie dodatkowo podajemy również wybrane urządzenie:

```
new cl_device_id[]{RuntimeConfigurationSet.device}
```

Aby móc przesyłać programy w postaci *kerneli*, należy **utworzyć kolejkę zadań**. Odpowiada za to metoda *createCommandQueue*, wywołująca metodę *clCreateCommandQueue* z pakietu *jocl*. Przekazuje się tam kontekst pracy (*context*) oraz wskazuje konkretne urządzenie (*RuntimeConfigurationSet device*), dla którego chcemy utworzyć kolejkę.

Aby *kernel* mógł **operować na danych**, które uprzednio są wygenerowane jako wejściowe, jak również aby mógł przekazać do programu wyniki, należy utworzyć odpowiednie

zmienne buforowe, będące wskaźnikami typu *cl_mem* rozszerzające typ *NativePointerObject*. Ich inicjalizacja odbywa się w metodzie *createBuffers*.

Dane wejściowe (z programu sterującego do *kernela*) uzyskują następujące flagi:

```
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR
```

Dane wyjściowe (z *kernela* do programu sterującego) uzyskują natomiast flagę:

```
CL_MEM_READ_WRITE
```

W obu przypadkach, tj. zmiennych buforowych, w przykładzie (*com.michalasobczak.bandwidth*) zarówno dane wejściowe jak i wyjściowe mają wskazany **rozmiar**:

```
(long) Sizeof.cl_float * this. n
```

Oznacza to, że rozmiar bufora w jednym wymiarze (bo o takim przykładzie tutaj mówimy) jest **wielokrotnością rozmiaru typu *cl_float***. Wielokrotność ta wynika z **ilości danych wskazanych** jako *n*, tj. *global_work_size*. Warto dodać, że dla buforu danych wejściowych podajemy wskaźnik typu *Pointer* do faktycznych danych:

```
KernelConfigurationSet.srcA
```

Dla **bufora** danych wyjściowych nie podajemy tego wskaźnika. Będzie miało to dopiero znaczenie przy wykonaniu *kernela* i czytaniu danych wynikowych. Poza samą **treścią** *kernela*, czyli **właściwego programu obliczeniowego** pracującego na zasadzie SIMD, istotny jest fakt **załadowania do pamięci** programu sterującego między innymi tym *kernelem* (tj. przykładu w *com.michalasobczak.bandwidth*). Wczytanie to odbywa się w metodzie *readKernelFile*. Jest to zwykłe wczytanie pliku z dysku. Treść programu *kernela* ładowana jest do pola *content*.

Po **wczytaniu** *kernela* przez program sterujący należy jego **treść przekazać do utworzonego wcześniej kontekstu pra-**

cy. Za inicjalizację (Listing 2) odpowiada metoda *initializeKernel*. Wywołuje ona metodę *clCreateProgramWithSource* z pakietu *jocl*. Program **budujemy** za pomocą metody *clBuildProgram* przekazując utworzony przed momentem program. Inicjalizacja pola *cl_kernel*, czyli de facto **inicjalizacja samego *kernela***, to wywołanie metody *clCreateKernel*. Parametryzacja *kernela* będzie kompletna dopiero po **przekazaniu mu wskaźników do buforów danych wejściowych i wyjściowych** za pomocą metody *clSetKernelArg*.

```
public void initializeKernel() {
    this.program = clCreateProgramWithSource(this.context, 1,
          new String[]{ this.content }, null, null);
    clBuildProgram(this.program, 0, null, "-cl-std=CL1.1", null, null);
    this.kernel = clCreateKernel(this.program, "sampleKernel", null);
    clSetKernelArg(this.kernel, 0, Sizeof.cl_mem, Pointer.to(this.memObjects[0]));
    clSetKernelArg(this.kernel, 1, Sizeof.cl_mem, Pointer.to(this.memObjects[1]));
}
```

Listing 2 — Initializacja kernela

Jedną z bardziej istotnych kwestii dotyczących wydajności wykonania *kernela* na urządzeniach obliczeniowych jest **wskazanie rozmiaru globalnego i lokalnego**. Rozmiar globalny jest **przeważnie ilością danych** do przetworzenia (w większości przypadków i w uproszczeniu). Rozmiar globalny **może być różny od ilości danych** i wtedy wskazuje ilość wątków do uruchomienia, na których będziemy pracować na zadanym zestawie danych. Powinien być **wielokrotnością** rozmiaru lokalnego. Rozmiar lokalny nie może być większy niż rozmiar globalny, to raczej oczywiste, ale nie może również przekraczać określonej wartości zdefiniowanej jako limit danego urządzenia, np. 256 czy 1024. W poprzednim rozdziale wspomniałem o tak zwanych osnowach/falach (tj. *Warp*), które w sprzętowy sposób determinują równoległe przetwarzanie.

> „Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently” [010]

[214] Oznacza to, że powinno się **ustalać rozmiar lokalny jako przynajmniej wartość 32 dla urządzeń NVIDIA oraz wartość 64 dla kart graficznych AMD**. Jeśli zamierzamy ustalić rozmiar lokalny większy, wówczas powinien być on wielokrotnością podanych wartości. Za konfigurację rozmiarów (Listing 3) odpowiada metoda *configureWork*.

```
public void configureWork() {
    this.global_work_size = new long[] { this.n } ;
    this.local_work_size  = new long[] { 32 };
}
```

Listing 3 — Konfiguracja rozmiarów

Aby **uruchomić tak przygotowany *kernel*** należy wywołać metodę *runKernel*. Aby **zakolejkować** program, należy użyć metody *clEnqueueNDRangeKernel*. Odczyt bufora wyników to wywołanie metody *clEnqueueReadBuffer*. **Wskazujemy tutaj miejsce docelowe na wynik**, tj. pole *memObjects* jak również rozmiar tych danych, w tym wypadku wielokrotność rozmiaru zmiennej *cl_float*.

Zarówno zakolejkowanie uruchomienia, jak i czytanie z bufora jest **asynchroniczne w tym sensie, że określenie dokładnego momentu wykonania konkretnego wątku *kernela* może być trudne**.

Uwaga: Należy mieć też na względzie opóźnienia wynikające z uruchamiania poszczególnych fragmentów programu sterującego oraz transferu danych pomiędzy gospodarzem, a urządzeniem obliczeniowym (jeśli są to różne urządzenia).

Poprawne **zakończenie** wykonywania *kernela* należy uzupełnić o **zwolnienie wszystkich uprzednio zaalokowanych zasobów** (Listing 4). Zwalniamy bufory, *kernel*, program, kolejkę oraz kontekst. Zwolnienie to odbywa się w metodzie *releaseResources*.

```
public void releaseResources() {
    clReleaseMemObject(this.memObjects[0]);
    clReleaseMemObject(this.memObjects[1]);
    clReleaseKernel(this.kernel);
    clReleaseProgram(this.program);
    clReleaseCommandQueue(this.commandQueue);
    clReleaseContext(this.context);
}
```

Listing 4 — Zwolnienie zasobów

Drugim z elementów podstawowego przykładu jest klasa *Main*. Zawiera ona wszelkie niezbędne wywołania konfiguracyjne oraz parametryzacyjne wymagane aby uruchomić plik z programem *kernela* na urządzeniu obliczeniowym. W pierwszej kolejności tworzymy **konfiguracje** platformy, uruchomienia i *kernela* (Listing 5).

```
Utils.log("1. Initialize configuration classes");
int n = 1024;
PlatformParametersSet p = new PlatformParametersSet();
RuntimeConfigurationSet r = new RuntimeConfigurationSet();
KernelConfigurationSet c = new KernelConfigurationSet(n);
```

Listing 5 — Inicjalizacja

W drugiej kolejności **wypisujemy dostępne platformy i urządzenia** oraz ich parametry sprzętowe wraz z ograniczeniami (Listing 6).

```
Utils.log("2. Get and print platform/devices parameters");
p.getPlatformsAndDevices();
p.getDevicesInfo();
```

Listing 6 — Dostępne platformy i urządzenia

Następnie przychodzi czas na **wskazanie platformy i urządzenia** na tej platformie, na którym chcemy wykonywać nasze obliczenia (Listing 7).

```
Utils.log("3. Platform and device selection");
r.selectPlatform(0);
r.selectDevice(0);
```

Listing 7 — Wskazanie zestawu obliczeniowego

Po wybraniu platformy i urządzenia należy **utworzyć przestrzeń na dane wejściowe i wyjściowe**, tak aby program miał na czym pracować (Listing 8).

```
Utils.log("4. Create input and output data");
c.initializeSrcArrayA();
c.initializeDstArray();
c.generateSampleRandomData();
c.printSrcArray();
```

Listing 8 — Przestrzenie na dane

Przed uruchomieniem programu, należy zapewnić mu odpowiednio **kontekst pracy**, a także utworzyć **kolejkę zadań**. Należy też **zainicjować bufory** (Listing 9).

```
Utils.log("5. Read kernel file, create program, pass arguments");
c.readKernelFile();
c.initializeKernel();
c.configureWork();
```

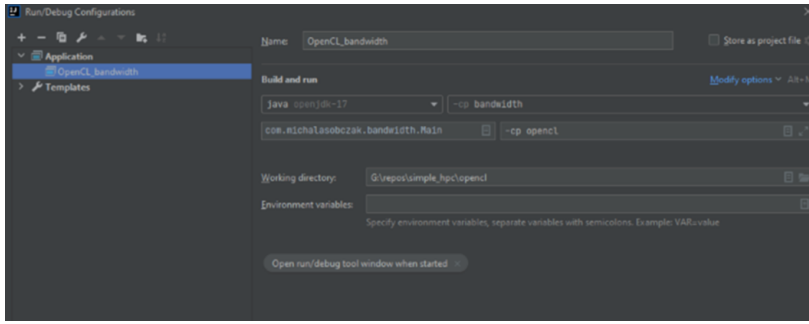
Listing 9 — Wczytanie kernela

Mając przygotowane wszystkie niezbędne komponenty i konfiguracje, uruchamiamy *kernel*. W tym samym momencie nastąpi **wczytanie bufora wynikowego**. Po zakończonej pracy zwalniamy zasoby (Listing 10) i jeśli to wymagane wyświetlamy wyniki oraz inne dane diagnostyczne z tym związane.

```
Utils.log("6. Run kernel, read buffer");
c.runKernel(1);
Utils.log("7. Release kernel, program, and memory objects");
c.releaseResources();
Utils.log("8. Debug results");
```

Listing 10 — Uruchomienie i pobranie wyników

Po przeglądzie wszystkich elementów zarówno pakietu bazowego jak i przykładu (tj. *bandwidth*), należy skonfigurować możliwość uruchamiania takiego zestawu z poziomu edytora. Aby to zrobić należy wybrać „*Add Configuration*” u góry po prawej stronie edytora.



Rys. 10 — Konfiguracja uruchamiania programu

Szczegóły dotyczące samego **uruchomienia** projektu znajdują się w **następnym rozdziale**, gdzie standard OpenCL będzie opisany bardziej szczegółowo, a przykłady będą rozwiązywały rzeczywiste problemy. W przykładowym projekcie postać *kernela* jest następująca:

```
__kernel void sampleKernel (__global const float *a,
                            __global float *d)
{
    __private int gid = get_global_id(0);
    d[gid] = a[gid];
}
```

Listing 11 — Treść kernela

Zadaniem ww. *kernela* (Listing 11) jest **przepisanie danych wejściowych jako dane wyjściowe**. Aby móc to zrobić, każde wywołanie/wątek pobiera swój unikalny identyfikator globalny dzięki czemu możemy uzyskiwać dostęp do danych przekazanych przez gospodarza do takiego programu wykonywanego na jednostce obliczeniowej.

4. POSZUKIWANIE OGRANICZEŃ

Aby dobrze poznać specyfikę działania urządzeń obliczeniowych i samych kerneli należy zapoznać się z ograniczeniami, jakie stawiają przed użytkownikiem.

4.1. Sterowniki

Kompatybilność pomiędzy klasami i architekturami

Przez dłuższy czas wykonywałem różne próby z kartami graficznymi na systemie Windows 10. Głównym problemem był fakt, że karty te były z różnych generacji.

Karty Quadro FX 5800 i Tesla K20Xm **nie mają wspólnego sterownika**. Karty GTX 650 Ti i 9400 GT mają co prawda wspólny sterownik, ale wraz z kolejnymi aktualizacjami systemu **przestaje on działać zgodnie z oczekiwaniami**, przez sporą różnicę w generacjach. Karty 8800 GTX i 9400 GT mają **wspólny sterownik** i to zestawienie wydaje się być najbardziej sensowne aby pokazać cokolwiek związanego z współpracą różnych urządzeń działających jednocześnie. Sterownik ten to 21.21.13.4201/342.01.

Kolejna kwestia to **brak aktywnego OpenCL w przypadku połączeń RDP**, czyli pulpitem zdalnym. Jedynie w przypadku kart do obliczeń będziemy mieli taką możliwość. Oznacza to, że istnieje teoretyczna możliwość pracy z Tesla K20Xm, ale i tak wymagana będzie jeszcze jedna karta z wyjściem wideo. Komputer (przeważnie) nie wystartuje jeśli nie ma obecnej karty z wyświetlaczem. Podłączenie jednak takiej karty obliczeniowej skutkuje tym, że trzeba zainstalować również kartę dla monitora. Jeśli płyta główna lub procesor nie posiadają takiego układu, wtedy należy albo szukać karty innego producenta lub liczyć na to, że sterowniki dla innej klasy urządzeń danego producenta nie będą konfliktowały.

Uwaga: Dla przykładu, na parze kart 8800 GTX i 9400 GT, OpenCL po RDP nie działa. Na parze kart GTX 650 Ti i Tesla K20Xm, OpenCL po RDP działa.

Jeśli karty są z tej samej serii architektury, np. Kepler, wówczas jest znaczna szansa, że sterownik przeznaczony dla GeForce, np. GTX 650 Ti **będzie pasował** do Tesla K20Km. Wersja sterownika, która w ten sposób została sprawdzone to 27.21.14.6140/461.40. Należy najpierw zainstalować GeForce w komputerze, kolejno zainstalować sterownik. Następnie zainstalować drugą kartę, a Windows 10 powinien sobie po kilku minutach poradzić z wybraniem tego samego sterownika dla tej drugiej karty.

Jeśli zezwolimy systemowi Windows 10 na aktualizację sterownika, wówczas zainstalowana wersja może wspierać jedną z kart, podczas gdy druga karta otrzyma inny sterownik co powoduje **konflikt**. Finalnie możemy nawet uszkodzić system, co będzie wymagało **odzyskania z punktu przywracania** lub wykonanie nowej instalacji. Wygodniej byłoby gdyby można było korzystać z różnych generacji i klas kart graficznych NVIDIA, ale są w tym zakresie zdecydowane ograniczenia.

Należy też wspomnieć, że **mocniejsze karty wymagają wielu linii zasilania**, 6 lub 8 pinowych. Jeśli zatem chcemy mieć więcej niż jedną kartę do obliczeń, wówczas należy zagwarantować odpowiednio duży zasilacz i komplet końcówek. Na moim testowym zestawie, tj. HP z800, zasilacz ma ponad 1kW mocy.

Uwaga: Karta ATI Radeon HD 4550 dotychczas testowana była z zestawie z jedną kartą (na sterowniku CAL 1.4.1734).

4.2. Ilość jednostek obliczeniowych

Jak dalece możemy zrównoleglać dane

[225] [227] Pierwszy parametr, na jaki chciałbym zwrócić uwagę to

```
CL_DEVICE_MAX_COMPUTE_UNITS
```

Jest to maksymalna dostępna liczba jednostek obliczeniowych. W przypadku uniwersalnych procesorów będzie to liczba rdzeni lub wątków. Jest to zatem dość przewidywalna i znana wartość. Większość osób wie jak sprawdzić ile ich CPU posiada logicznych jednostek obliczeniowych. Inaczej jednak sprawa się ma w przypadku układów graficznych, czyli GPU. Tutaj kiedy program sterujący zwróci wartość np. 3 będzie to rzeczywiście oznaczało, że w urządzeniu znajdują się 3 **jednostki**, ale nie będą to jednak tylko te jednostki, które będą wykonywały obliczenia.

W architekturze zunifikowanych jednostek cieniujących, na przykładzie sprzętu NVIDIA będzie to **ilość multiprocesorów strumieniowych** (SMP). Każdy z tych SMP posiada **wiele procesorów strumieniowych** (SP). Przykładowo układ **NVIDIA GK106**, na którym oparta jest karta **GeForce 650 Ti** posiada 4 multiprocesory, ale wszystkich jednostek obliczeniowych jest aż 768 sztuk, każdy taktowany zegarem 928 MHz uzyskując wydajność 1420 GFLOPS FMA.

4.3. Maksymalny rozmiar grupy

Limit współpracy synchronizowanych elementów

[216] [222] Kolejnym istotnym parametrem jest:

CL_DEVICE_MAX_WORK_GROUP_SIZE

który oznacza maksymalny rozmiar grupy. Poprzez pojęcie grupy rozumiemy tutaj **ilość elementów zebranych jako jeden zbiór mogący się wzajemnie synchronizować**, tj. uzyskać dostęp do wspólnej pamięci podręcznej. Dla przykładowej karty graficznej, [223] na jakiej są uruchamiane przykłady z niniejszej książki, tj. **NVIDIA Tesla K20Xm**, maksymalny rozmiar grupy wskazany jest jako 1024 elementów. Jeśli w metodzie *KernelConfigurationSet*. *configureWork* wskażemy wartość większą wówczas otrzymamy następujący komunikat o błędzie (Rys. 11).

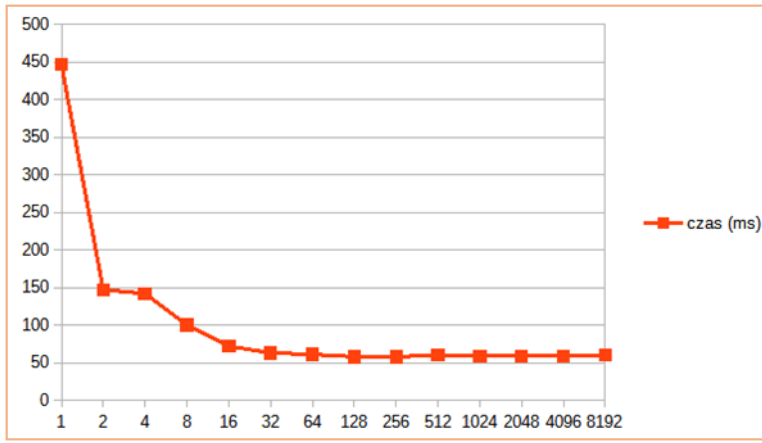
```
4. Create context and command queue and buffers
5. Read kernel file, create program, pass arguments
6. Run kernel, read buffer
Exception in thread "main" org.jocl.CLException Create breakpoint : CL_INVALID_WORK_GROUP_SIZE
    at org.jocl.CL.checkResult(CL.java:825)
    at org.jocl.CL.clEnqueueNDRangeKernel(CL.java:20954)
    at com.michalasobczak.bandwidth.KernelConfigurationSet.runKernel(KernelConfigurationSet.java:140)
    at com.michalasobczak.bandwidth.Main.main(Main.java:58)

Process finished with exit code 1
```

Rys. 11 — Kod błędu w przypadku przekroczenia maksymalnego rozmiaru grupy

4.4. Optymalny rozmiar lokalny

Doświadczalne poszukiwania

[221] Aby uzyskać najlepszą wydajność należy zadany problem rozpatrzyć nie tylko z punktu widzenia algorytmu, ale również **konfiguracji pracy do wykonania**. Parametryzując program sterujący **rozmiarem** lokalnym od wartości 1 do maksymalnej wartości 8192 (dla tego konkretnego procesora Intel i3—2328M, Rys. 12) uzyskujemy całkiem klarowny obraz dlaczego należy uwzględniać specyfikę konstrukcji sprzętu.



Rys. 12 — Rozmiar lokalny a czas (Intel i3—2328M)

W **oficjalnym podręczniku architektury CUDA** (tj. [010]) możemy zapoznać się z uzasadnieniem dlaczego należy pamiętać o wykonywaniu zadań z rozmiarem lokalnym będących określonymi wielokrotnościami:

> „When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. Section 2.1 describes how thread IDs relate to thread indices in the block.” [010]

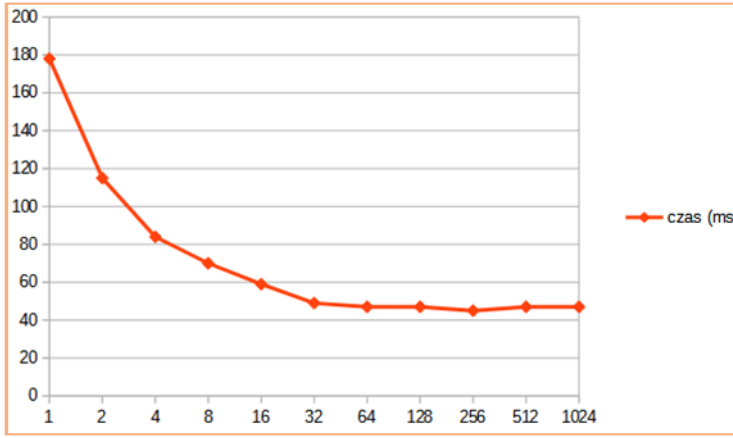
Dla porównania, uruchamiając tego sam kod *kernela*, tj.

```
d[gid] = (a[gid] * 2.0) — 1.0;
```

na karcie graficznej NVIDIA Tesla K20Xm uzyskamy zasadniczo ten sam wniosek. Początkowe wartości rozmiaru lokalnego takie jak 1 i 2 mają nieco bardziej gładki przebieg, bez zdecydowanego spadku jak obserwujemy to w przypadku CPU. Warto zwrócić uwagę, że karta ta posiada aż 14 jednostek obliczeniowych dających razem **2688 jednostek cieniujących, czyli faktycznych rdzeni taktowanych z prędkością 732 MHz**. Częstotliwość pracy jest możliwa do uzyskania poprzez odczytanie parametru

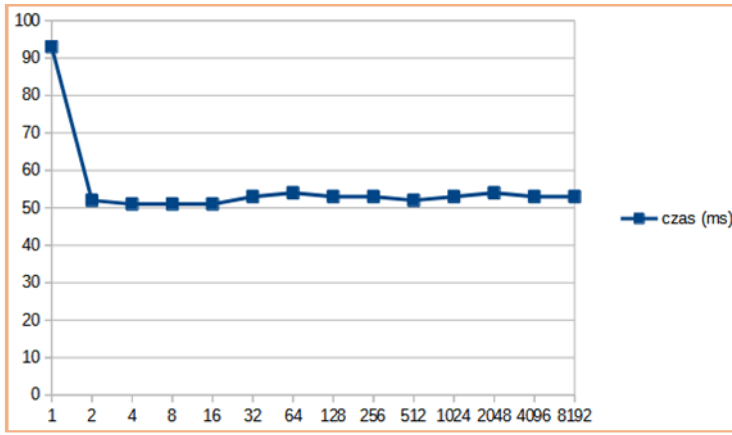
```
CL_DEVICE_MAX_CLOCK_FREQUENCY
```

Widać, że przy optymalnym ustawieniu rozmiaru lokalnego **czas niezbędny na wykonanie całości pracy** jest porównywalny z pozostałymi badanymi urządzeniami (Rys. 13).



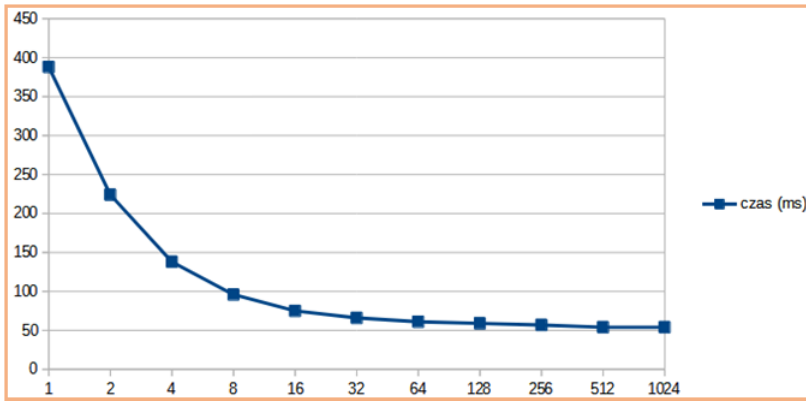
Rys. 13 — Rozmiar lokalny a czas (NVIDIA Tesla K20Xm)

Kolejnym badanym urządzeniem jest **para procesorów Intel Xeon X5660** @ 2.80 GHz dająca łącznie 24 procesory logiczne. Tak samo jak wcześniej badany procesor Intel, również ten działa pod kontrolą platformy Intel OpenCL. Wyniki są zdecydowanie inne (Rys. 14). **Nie widać tutaj sposobu, w jaki oprogramowanie jak i sprzęt rozdzielają prace do wykonania**.



Rys. 14 — Rozmiar lokalny a czas (Intel Xeon X5560)

Ostatnim testem w tej sekcji jest sprawdzenie *kernela* na karcie GTX 650 Ti, która jest około 3 razy mniej wydajna w porównaniu do karty Tesla K20XM, lecz dla przykładu podstawowego wcale tej różnicy znacząco nie zauważymy (Rys. 15).



Rys. 15 — Rozmiar lokalny a czas (GeForce GTX 650 Ti)

Aby poznać rzeczywiste korzyści ze stosowania obliczeń dedykowanych układom GPU należy wziąć pod uwagę jednak nieco więcej aspektów niż tylko prosty algorytm, gdyż na takim korzyści nie są tak klarownie widoczne (lub nawet wcale).

4.5. Procedura uruchomienia

Przygotowanie i wykonanie programu kernela

Wróćmy teraz do samego uruchamiania *kerneli*. Zgodnie z tym, co zostało wcześniej opisane, każdy z kolejnych kroków jest identyfikowany przez tekst wypisywany z właściwym mu numerem. **Pierwszy z tych kroków**, to włączenie trybu przechwytywania wyjątków. Kolejny krok oznaczony jako nr 1 to inicjalizacja (Listing 12) obiektów klas *PlatformParametrsSet*, *RuntimeConfigutationSet* oraz *KernelConfigurationSet*.

```
0. OpenCL specific configuration
- Enable exceptions and subsequently omit error checks in this sample
1. Initialize configuration classes
- PlatformParametersSet
- RuntimeConfigurationSet
- KernelConfigurationSet
```

Listing 12 — Sekcje nr 0 i 1

Krok nr 2 to po pierwsze pobranie wszystkich dostępnych platform i urządzeń obecnych w środowisku (Listing 13). Mogą być to przede wszystkim urządzenia klasy CPU, GPU, ale nie tylko. Odpowiada za to metoda *getPlatformsAndDevices*. Po drugie, jest to wypisanie informacji o tychże urządzeniach. Dla każdego z tych urządzeń wypisywany jest szereg parametrów.

```
2. Get and print platform/devices parameters
- PlatformParametersSet::getPlatforms
- Number of platforms: 3
- #0: Number of devices in platform NVIDIA CUDA: 1, cl_platform_id[0x199fab77af0]
- #1: Number of devices in platform Intel(R) OpenCL HD Graphics: 1, cl_platform_id[0x199fab7b1d0]
- #2: Number of devices in platform Intel(R) OpenCL: 1, cl_platform_id[0x199fb73de68]
- PlatformParametersSet::getDevices
--- #0: Info for device NVIDIA GeForce RTX 3050 Ti Laptop GPU, cl_device_id[0x199fab77b40]: ---
```

Listing 13 — Sekcja nr 2

Lista parametrów zawiera identyfikatory, takie jak nazwa, producent czy wersja sterownika, oraz jakiego typu jest dane urządzenie (Listing 14). Mamy też informacje o ilości jednostek obliczeniowych (wraz z maksymalną częstotliwością taktowania), maksymalnej ilości wymiarów oraz maksymalnych ilości elementów w każdym z tych wymiarów.

```
CL_DEVICE_NAME:                          NVIDIA GeForce RTX 3050 Ti Laptop GPU
CL_DEVICE_VENDOR:                        NVIDIA Corporation
CL_DRIVER_VERSION:                       496.76
CL_DEVICE_TYPE:                          CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS:             20
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:      3
CL_DEVICE_MAX_WORK_ITEM_SIZES:           1024 / 1024 / 64
CL_DEVICE_MAX_WORK_GROUP_SIZE:           1024
CL_DEVICE_MAX_CLOCK_FREQUENCY:           1695 MHz
```

Listing 14 — Parametry urządzenia (sekcja nr 2)

Maksymalny rozmiar grupy, czyli ilość elementów lokalnych uwzględniać powinien wszystkie wymiary.

W dalszej części mamy informację o kwestiach związanych z pamięcią (Listing 15), takie jak szerokość adresowania, maksymalny rozmiar pamięci do alokowania, maksymalna ilość pamięci w całym urządzeniu, typ pamięci lokalnej wraz z jej rozmiarem. Dowiadujemy się też o rozmiarze bufora na wartości stałe oraz sposobu obsługi kolejkowanych *kerneli* do wykonania.

```
CL_DEVICE_ADDRESS_BITS:              64
CL_DEVICE_MAX_MEM_ALLOC_SIZE:        1023 MByte
CL_DEVICE_GLOBAL_MEM_SIZE:           4095 MByte
CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
CL_DEVICE_LOCAL_MEM_TYPE:            local
CL_DEVICE_LOCAL_MEM_SIZE:            48 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
CL_DEVICE_QUEUE_PROPERTIES:          CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
CL_DEVICE_QUEUE_PROPERTIES:          CL_QUEUE_PROFILING_ENABLE
```

Listing 15 — Parametry pamięci (sekcja nr 2)

Co prawda wydaje się, że wszystkie urządzenia zgodne ze standardem OpenCL obsługują API do obsługi obrazów, jednak o tym, czy urządzenie wspiera ten standard, możemy się również dowiedzieć. W szczególności będą istotne **maksymalne rozmiary w przypadku 2 lub 3 wymiarów** (Listing 16).

```
CL_DEVICE_IMAGE_SUPPORT:         1
CL_DEVICE_MAX_READ_IMAGE_ARGS:   256
CL_DEVICE_MAX_WRITE_IMAGE_ARGS: 32
CL_DEVICE_SINGLE_FP_CONFIG:
          CL_FP_DENORM
          CL_FP_INF_NAN
          CL_FP_ROUND_TO_NEAREST
          CL_FP_ROUND_TO_ZERO
          CL_FP_ROUND_TO_INF
          CL_FP_FMA
          CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT

CL_DEVICE_2D_MAX_WIDTH           32768
CL_DEVICE_2D_MAX_HEIGHT          32768
CL_DEVICE_3D_MAX_WIDTH           16384
CL_DEVICE_3D_MAX_HEIGHT          16384
CL_DEVICE_3D_MAX_DEPTH           16384
```

Listing 16 — Parametry wymiarów (sekcja nr 2)

Parametr:

CL_DEVICE_SINGLE_FP_CONFIG

mówi nam o specyfice **obsługi liczb zmiennoprzecinkowych pojedynczej precyzji**. Ostatni parametr wskazuje nam, jakich typów danych powinniśmy używać aby uzyskać najlepszą wydajność. W przypadku przykładowego CPU będą to typy podstawowe. Może się jednak zdarzyć, że zamiast *int* powinniśmy stosować typ *int4* lub *int16* jeśli tak zostanie nam tutaj wskazane.

```
CL_DEVICE_PREFERRED_VECTOR_WIDTH_ CHAR 1, SHORT 1, INT
1, LONG 1, FLOAT 1, DOUBLE 1
```

Kolejna sekcja o numerze 3 (Listing 17) to wybór platformy i urządzenia. Na tym konkretnie urządzeniu będziemy wykonywać program *kernela*. **Sekcja nr 4** to alokowanie danych wejściowych wraz z buforem, oraz bufor na dane wyjściowe. Jako, że przykład dotychczasowy dotyczył danych pseudolosowych wymagane jest również aby takie dane zostały wygenerowane. Tworzymy tutaj również kontekst pracy i kolejkę. **Sekcja nr 5** to odczytanie treści *kernela*, utworzenie programu i przekazanie do niego argumentów. **Sekcja nr 6** to uruchomienie *kernela* i odczytanie bufora wynikowego. **Sekcja nr 7** to zwolnienie wcześniej zaalokowanych zasobów. **Sekcja nr 8** to opcjonalne wypisanie wyniku w zależności od przykładu.

```
3. Platform and device selection
selectPlatform: cl_platform_id[0x199fab77af0]
selectDevice: cl_device_id[0x199fab77b40]
4. Create input and output data
- Allocating sample data
- Allocating return buffer
- Started randomizing
- Finished randomizing
4. Create context and command queue and buffers
5. Read kernel file, create program, pass arguments
6. Run kernel, read buffer
Took OpenCL read result: 162ms
7. Release kernel, program, and memory objects
8. Debug results
```

Listing 17 — Sekcje od 3 do 8

O profilowaniu wykonywanego *kernela* będę pisał za moment. Jeśli mamy bowiem uzyskać korzyść na czasie wykonywania programu, musimy mieć **narzędzia aby zmierzyć czas kolejkowania, obliczeń oraz czytania wyniku**.

4.6. Przykładowy błąd

Sposób obsługi nieoczekiwanych dostępów

[226] Zanim przejdziemy dalej, chciałbym się na chwilę zatrzymać na przykładzie dotyczącym błędu w *kernelu*. Wyjątek poniżej spowodowany jest **błędem w dostępie do pamięci**.

```
Process finished with exit code -1073741819 (0xC0000005)
```

Błąd taki **prokuruje** się w następujący sposób (Listing 18):

```
__private int n = 10;
__private int myArray[10];
for (int i=0; i<n; i++) {
    myArray[i] = i;
}
d[gid] = (a[gid] * 2.0) - myArray[gid];
```

Listing 18 — bandwidth/kernel. c

Jako wsad do zadania (rozmiar globalny) przekazujemy więcej niż rozmiar inicjowanej w *kernelu* tablicy (*n*). **Próba dostępu dotyczy zatem obszaru nieokreślonego**. W języku C stosowanym w OpenCL nie mamy możliwości deklarowania długości tablicy ze zmiennej.

Co ciekawe, po zakończeniu pisania książki, przystępując do jej składu, przeglądałem przykład po przykładzie celem ich weryfikacji. Zauważyłem, że przykład zaprezentowany w niniejszym podrozdziale zachowuje się inaczej niż w momencie kiedy ten przykład przygotowałem. Na nowym komputerze (Dell G15 z grafiką GeForce RTX 3050 Ti) wyjątku tego nie mo-

gę uzyskać. Jeśli użyję grafiki zintegrowanej (tj. Intel UHD) wówczas mam błąd kompilacji ze względu na brak rozszerzenia

```
cl_khr_gl_msaa_sharing
```

Wykonując kod na CPU co prawda otrzymuje błąd, ale nieco inny niż na oryginalnym sprzęcie. Na oryginalnym CPU (tj. Intel i3 oraz i7 drugiej generacji) punktem granicznym jest wartość 4096 elementów rozmiary globalnego. Uzyskujemy wtedy wyjątek maszyny wirtualnej Java. Na oryginalnym GPU (tj. NVS 4200) uzyskujemy jeszcze inny kod błędu:

```
INVALID error code: -9999
```

Zrozumiałe jest dla mnie, że mamy tutaj do czynienia przede wszystkim z błędem o nieokreślonym działaniu, aczkolwiek liczyłem, że nie będzie to powodowało aż tak dużych różnic.

> *Uwaga: Istnieje wiele innych ograniczeń, które w toku dalszych rozdziałów będą prezentowane. Dotyczy to dostępności funkcji oraz składni, która w przypadku programowania innego niż pod OpenCL, byłaby dostępna.*

4.7. Profilowanie

[215] Dotychczasowy przykład (tj. *bandwidth*) składał się z pakietu bazowego (tj. *opencl*), konfiguracji i samego *kernela*. Program wysyłany do urządzenia nie był w żadnym stopniu skomplikowany. Jeśli jednak zacznie być, będziemy potrzebowali poznać jego wydajność, poprawnie ją mierzyć. W tej sekcji przedstawię kilka zmian, jakie trzeba wykonać aby **uzyskać dostęp do informacji diagnostycznych związanych z profilowaniem programu *kernela* i współpracy urządzenia typu *host* z docelowym urządzeniem obliczeniowym**. Zaczynamy od rozszerzenia definicji kolejki (Listing 19).

```
public void createCommandQueue() {
    // Create a command-queue for the selected device
    long properties = 0;
    properties |= CL.CL_QUEUE_PROFILING_ENABLE;
    this.commandQueue = clCreateCommandQueue(this.context,
         RuntimeConfigurationSet.device, properties, null);
}
```

Listing 19 — Parametr CL_QUEUE_PROFILING_ENABLE

> *Uwaga: Na tym etapie modyfikowany będzie przykład podstawowy. Jeśli kogoś będzie interesowała jego wersja bez profilowania, można ją uzyskać w historii repozytorium.*

Kolejna zmiana dotyczy **utworzenia zdarzenia i przekazania go** do metody *clEnqueueNDRangeKernel*. Analogiczna zmiana dotyczy bufora odczytu, tu również tworzymy zdarzenie i dodajemy je do wywołania metody *clEnqueueRe-*

adBuffer. Po zmianie metoda *runKernel* ma postać (Listing 20):

```
cl_event kernelEvent0 = new cl_event();
clEnqueueNDRangeKernel(this.commandQueue, this.kernel, 1, null,
        this.global_work_size, this.local_work_size, 0, null, kernelEvent0);
//
System.out.println("Waiting for kernel events...");
CL.clWaitForEvents(1, new cl_event[]{kernelEvent0});
//
cl_event readEvent0 = new cl_event();
clEnqueueReadBuffer(this.commandQueue, this.memObjects[1], CL_TRUE, 0,
        (long) n * Sizeof.cl_float, KernelConfigurationSet.dst, 0, null, readEvent0);
//
System.out.println("Waiting for read events...");
CL.clWaitForEvents(1, new cl_event[]{readEvent0});
```

Listing 20 — Zmiany do runKernel

W następnej kolejności chcemy **wypisać na ekran informacje**, które dzięki stosowaniu zdarzeń jesteśmy w stanie uzyskać (Listing 21):

```
ExecutionStatistics executionStatistics = new ExecutionStatistics();
executionStatistics.addEntry("kernel0", kernelEvent0);
executionStatistics.addEntry("  read0", readEvent0);
executionStatistics.print();
```

Listing 21 — Obiekt informacji diagnostycznych

Za warstwę prezentacji odpowiadają klasy *ExecutionStatistics* oraz *Entry*. W szczególności należy zwrócić uwagę na metodę konstruktora tej drugiej (Listing 22). **Pobierane są kolejno czasy kolejkowania, przekazania do wykonania, uruchomienia oraz zakończenia wykonywania po stronie urządzenia docelowego**.

```
Entry(String name, cl_event event)
{
    this.name = name;
    CL.clGetEventProfilingInfo(
            event, CL.CL_PROFILING_COMMAND_QUEUED,
            Sizeof.cl_ulong, Pointer.to(queuedTime), null);
    CL.clGetEventProfilingInfo(
            event, CL.CL_PROFILING_COMMAND_SUBMIT,
            Sizeof.cl_ulong, Pointer.to(submitTime), null);
    CL.clGetEventProfilingInfo(
            event, CL.CL_PROFILING_COMMAND_START,
            Sizeof.cl_ulong, Pointer.to(startTime), null);
    CL.clGetEventProfilingInfo(
            event, CL.CL_PROFILING_COMMAND_END,
            Sizeof.cl_ulong, Pointer.to(endTime), null);
}
```

Listing 22 — Konstruktor klasy Entry

4.8. Computing Capabilities

Maksymalizacja utylizacji urzadzeń NVIDIA

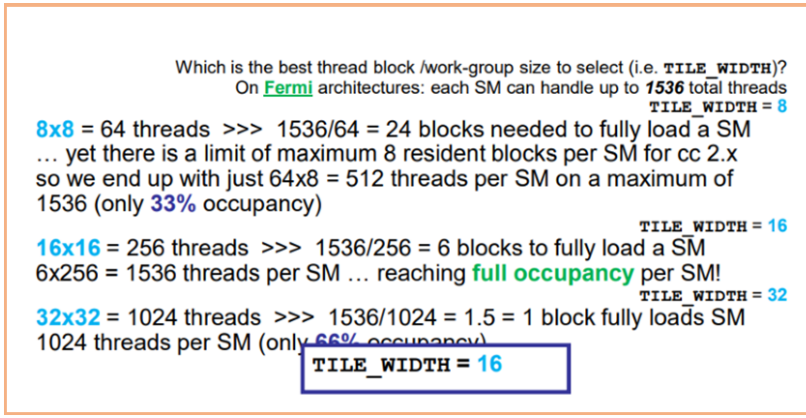
Na dzisiaj (tj. 2022) pojęcie CC (*Computing Capabilities*) oznaczane jest wersjami począwszy od 1.0 do 8.6. Urządzenia oznaczone jako obsługujące wersję 2.0 mają wyszczególnione, że obsługują następujące **parametry per multiprocesor strumieniowy** (SMP):

i. 8 bloków (grup)

ii. 48 sztuk warp

iii. 1536 wątków (elementów)

Warto wspomnieć, że maksymalny rozmiar grupy, tj. rozmiar lokalny wynosi 1024 elementów. Na *warp* składają się zaś 32 elementy.



Rys. 16 — Optymalny rozmiar lokalny architektury Fermi [303]

Biorąc za przykład kartę NVIDIA GeForce GTX 650 Ti, posiadającą 4 jednostki wykonawcze, daje to teoretyczną możliwość uruchomienia (ale nie jednoczesnego działania) 4 x 48 x 32, czyli 6144 wątków. Posiadając 768 sztuk jednostek cieniujących daje to 192 sztuki per multiprocesor strumieniowy. Pojedynczy SMP posiadając teoretyczną możliwość uruchomienia 1536 sztuk wątków może i tak w tym wypadku wykonać jedynie 192 operacji w cyklu zegara, gdyż tyle posiada arytmometrów. Idąc dalej, jeśli zatem maksymalna ilość grup (bloków) per SMP to w tym wypadku 8, to optymalny rozmiar grupy (Rys. 16) powinien wynosić 1536/8, czyli 192. Uzyskamy wtedy maksymalną utylizację. Oczywiście należy pamiętać, że **nie tylko od utylizacji zależy maksymalizacja wydajności**, ale również od:

i. ***ilości cykli zegara*** *niezbędnych do wykonania obliczeń*

ii. ***współbieżności*** *wykonywanych obliczeń*

iii. ilości i sposobu ***dostępu do pamięci*** *obecnych na urządzeniu*

iv. ***rozgałęzień*** *i* ***barier***

Należy pamiętać, że ilość elementów czy grup obecnych do wykonania nie jest jednoznaczna z ich faktycznym wykonywaniem. Są one strumieniowane do urządzenia, ale **proces obliczeniowy ograniczony jest oczywiście możliwościami sprzętowymi**. Różnice występują również pomiędzy dostawcami kart graficznych, ale bardziej znaczące będą pomiędzy kategoriami urządzeń (CPU i GPU).

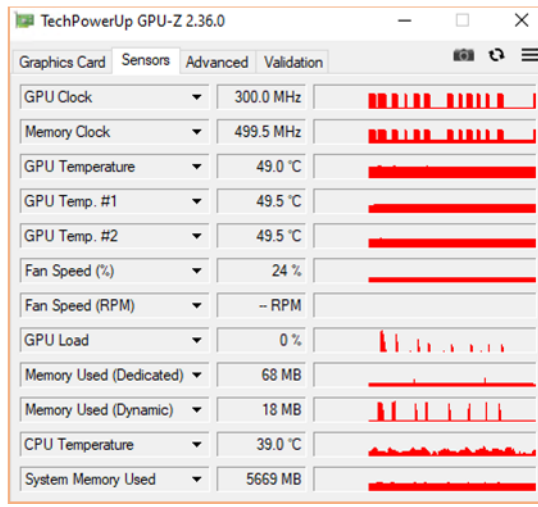
Uwaga: urządzenia w większości znanych mi przypadków potrafią pobierać kolejne rozkazy do wykonania na zapas lub nawet próbować zgadywać ścieżkę wykonywania. Często również wszystkie ścieżki są wykonywane, a obliczenia, które były w niewłaściwie oszacowanej ścieżce, zwyczajnie odrzucane.

4.9. GPU Load

Rozmiar lokalny a wysycenie karty graficznej

W sekcji dotyczącej profilowania został wprowadzony **podział na część związaną z obliczeniami oraz część odczytu danych** z pamięci urządzenia. Aby ten przykład nieco bardziej uwypuklić do przykładu została wybrana karta graficzna ATI Radeon HD 4550 oparta na układzie RV710. Jest to **jedna z pierwszych kart obsługujących standard OpenCL**.

Karta ta wykonuje przykład około 10-krotnie wolniej niż pozostałe użyte dotychczas urządzenia (Rys. 17). Istotne jest jednak to, że zmienna jest w tym wypadku jedynie część dotycząca obliczeń, a nie samego odczytu pamięci po wykonaniu tychże obliczeń. Być może nie jest to najbardziej spektakularne odkrycie, w szczególności, że dotychczas odczyt z pamięci nie był w żaden sposób parametryzowany. Potwierdza to jednak oczekiwania.



Rys. 18 — Parametry pracy ATI Radeon HD 4550

Zadając różne rozmiary lokalne, ustawiamy nie tylko rozmiar grupy, ale pośrednio wpływamy na ilość grup, która w danym momencie może zostać przekazana do realizacji do urządzenia. Jak już wspomniałem wcześniej, nie oznacza to zapewne, że grupy te i wątki będą od razu realizowane. Proszę zwrócić uwagę na sekcję *GPU Load* na zrzucie z programu GPU-Z (Rys. 18). Począwszy od rozmiaru 1 i dalej 2, 4, 8, 16, 32, 64 oraz 128 (maksymalny dla tego urządzenia) poziom utylizacji oraz czas trwania tego obciążenia całkiem dobrze pokrywa się z wykresem czasu realizacji z mechanizmu profilowania z samej już aplikacji.

Uwaga: nieznaczne różnice pomiędzy tymi wykresami wynikają z tego, że urządzenie ma zmienną częstotliwość pracy zarówno jednostek obliczeniowych jak i pamięci i nie mamy określonego w prosty sposób czasu reakcji oraz ewentualnych opóźnieniach związanych z inicjalizacją urządzeń i wymianą danych pomiędzy nimi.

4.10. Indeksacja

Przegląd podstawowych funkcji siatki obliczeniowej

Do identyfikowania elementów, które przekazywane są do przetwarzania możemy wykorzystywać kilka **funkcji wbudowanych** w OpenCL. Jest to przede wszystkim **funkcja pobierania identyfikatora globalnego *get_global_id***. Możemy taką funkcją pobierać **indeks globalny** w kilku wymiarach, jeśli takowe stosujemy. Jeśli założymy, że *kernel* zawiera następujący kod:

```
__private int gid = get_global_id(0);
d[gid] = gid;
```

wówczas wywołanie go, zwróci następującą tablicę wynikową:

```
Result:
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0,
14.0, 15.0]
```

Przy założeniu, że rozmiar lokalny wynosi 4, to *kernel* w (skróconej) postaci:

```
__private int gid = get_global_id(0);
__private int lid = get_local_id(0);
d[gid] = lid;
```

zwróci następujący wynik:

```
Result: [0.0, 1.0, 2.0, 3.0, 0.0, 1.0, 2.0, 3.0, 0.0, 1.0, 2.0, 3.0, 0.0, 1.0,
2.0, 3.0]
```

Indeksacja od 0 do 3 oznacza, że mamy w każdej grupie o rozmiarze 4 pobierany kolejny element w każdym równolegle działającym wątku. **Do pobierania identyfikatora lokalnego wykorzystujemy funkcję *get_local_id***. Idąc tym tropem

mamy również do dyspozycji inne funkcje odpowiedzialne za pobranie identyfikatorów kolejnych wyróżników, np. numer kolejny grupy (*get_group_id*), ilość grup (*get_num_groups*), rozmiar lokalny (*get_local_size*).

Obliczając indeks bieżącego elementu, zamiast używać funkcji wbudowanej *get_global_id*, możemy zastosować pośrednie obliczenie:

```
__private int gr = get_group_id(0);
__private int ls = get_local_size(0);
__private int lid = get_local_id(0);
__private int eid = (gr*ls)+lid;
```

Wykorzystując numer grupy, rozmiar grupy i numer kolejny w grupie, uzyskujemy **równoważnik identyfikatora globalnego** (*eid*). Jeśli z kolei chcemy się dowiedzieć jaki jest rozmiar globalny (*gs*):

```
__private int gc = get_num_groups(0);
__private int ls = get_local_size(0);
__private int gs = gc*ls;
```

4.11. Szeregowanie

Kolejność wykonywania zadań

[224] Jak już wcześniej wspomniałem, nie ma możliwości skutecznego i pewnego określenia **kolejności uruchamiania poszczególnych zadań**. Jest co prawda możliwość ustawiania blokad synchronizacyjnych, ale wtedy znamy tylko punkt spotkania się wątków, a nie ich ścieżkę dojścia. Mamy jednak możliwość potwierdzenia przypuszczenia, że wykonywanie jest **dowolne**. Mówi nam o tym pierwsza z flag konfiguracji kolejki:

```
CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_PROFILING_ENABLE
```

Począwszy od rozmiaru globalnego wynoszącego 64 elementy, sprawdzenie polegające na wypisaniu na ekran komunikatu z wewnątrz *kernela*, pokazuje wykonywanie poza kolejnością. Oczywiście jest to jedynie **domniemanie**, ponieważ użycie funkcji *printf*, która jest **poza standardem** OpenCL, samo w sobie jest ryzykowne aby wyciągać z tego jakiekolwiek poważne wnioski. Traktowałbym to jako ciekawostkę. Nie ma bowiem gwarancji w jaki sposób obsługiwany jest tutaj **bufor** i czy wpisywanego do niego komunikaty są **szeregowane czasem** wstawienia, czy też kolejność jest przypadkowa. Nie wiadomo.

```
__kernel void sampleKernel(__global const float *a,
                           __global float *d)
{
    __private int gid = get_global_id(0);
    __private int gr  = get_group_id(0);
    __private int gc  = get_num_groups(0);
    __private int lid = get_local_id(0);
    __private int ls  = get_local_size(0);
    //
    __private int eid = (gr*ls)+lid;
    __private int gs  = gc*ls;
    //
    printf(" %i \n", gid);
    d[gid] = gs;
}
```

Listing 23 — bandwidth/kernel4.c

Jeśli taki sam test wykonamy na karcie ATI Radeon HD 4550 to dowiemy się, że urządzenie to wspiera jedynie wersję OpenCL 1.0 AMD-APP 937.2 pracując na sterowniku z 2015 roku. Samo urządzenie zostało wyprodukowane w końcówce 2008 roku. **Urządzenie o takim profilu nie wspiera wielu rzeczy obecnych w urządzeniach nowszej generacji** jak np. funkcja *printf* czy wsparcia dla przetwarzania obrazów (Rys. 19).

```
Exception in thread "main" org.jocl.CLException: CL_BUILD_PROGRAM_FAILURE
Build log for device 0:
"C:\Users\ja\AppData\Local\Temp\OCL220.tmp.cl", line 6: error: function
        "printf" declared implicitly
      printf("I'm at %u\n", gid);
      ^

1 error detected in the compilation of "C:\Users\ja\AppData\Local\Temp\OCL220.tmp.cl".

Internal error: clc compiler invocation failed.


    at org.jocl.CL.clBuildProgram(CL.java:11299)
    at com.michalsobczak.bandwidth.KernelConfigurationSet.initializeKernel(KernelConfigurationSet.java:125)
    at com.michalsobczak.bandwidth.Main.main(Main.java:53)

Process finished with exit code 1
```

Rys. 19 — Brak bezpośredniego wsparcia dla funkcji printf

Uwaga: Dla wartości mniejszych, tj. 8, 16 czy 32 wyświetlane liczby są w szeregu kolejnych następujących po sobie.

4.12. Kolejność danych

[009] **OpenCL jest standardem uniwersalnym**. Może się zatem zdarzyć, że porządek bitów typów skalarnych systemu gospodarza będzie inny niż analogiczny porządek na urządzeniach obliczeniowych OpenCL. Sprawdzając informacje o urządzeniu, parametry dotyczące porządku, tj. kolejności bitów znajdziemy w polu:

```
CL_DEVICE_ENDIAN_LITTLE
```

Jeśli jest ustawione na wartość *CL_TRUE*, wówczas mamy *little-endian*, czyli najmniej znaczący bit znajduje się wtedy po prawej stronie. W przeciwnym razie będzie to *big-endian*. Jeśli występuje różnica pomiędzy urządzeniami, wówczas może się zdarzyć, że będzie trzeba szeregować dane we własnym zakresie tak aby uzyskać ich oczekiwaną postać.

4.13. Bariery

Synchronizacja wątków w dostępie do pamięci

[218] Aby umożliwić równoczesny dostęp do pamięci lokalnej lub globalnej, program musi albo zastosować operację atomiczną albo postawić barierę. **Wszystkie wątki danej grupy w przypadku postawienia bariery będą na nią czekały żeby przystąpić do dalszego wykonania programu**. Tyle mówi teoria. Mowa jest również o tym, że warunkowanie barier, to znaczy doprowadzenie do sytuacji, w której nie wszystkie wątki dotrą w to miejsce, **może powodować zawieszenie urządzenia obliczeniowego**. Jest to bardzo kusząca opcja żeby ją spróbować.

Powinniśmy oczekiwać jedynie nieoczekiwanego, bowiem do bariery dotrą te wątki, których globalny identyfikator jest mniejszy niż 10. Pozostałe wątki nie wiedzą nic o tej barierze.

```
Result: [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 0.0, 0.0, 10.0, 11.0, 12.0,
13.0, 14.0, 15.0]
```

Problem zatem jest w momencie, gdy wątki objęte barierą próbują się z niej wydostać. W badanym przypadku rozmiar globalny to 16, a lokalny to 4. Pierwsza (od 0 do 3) i druga (od 4 do 7) grupa wykonują i synchronizują się pomyślnie. Trzecia (od 8 do 11) grupa natomiast **ma w połowie stan nieokreślony, gdyż identyfikatory 8 i 9 będąc mniejszymi od 10 natrafiają na barierę, którą chcą synchronizować z pozostałymi elementami grupy.** Nie nastąpi to jednak nigdy, ponieważ elementy o identyfikatorach 10 i 11 nie wiedzą o tej barierze wobec czego kończą swoje działanie. Stan elementów 8 i 9 jest nieokreślony, nie dotarły one najpewniej do końca i zostały abortowane. Niniejszy test (Listing 24) wykonany został na urządzeniu typu CPU. Próbowałem wielu postaci *kernela* tak aby zawiesić urządzenie, ale niestety się

to nie udało. Udało mi się jednak znaleźć **ciekawą różnicę pomiędzy zachowaniem kodu** wykonywanym na SDK od AMD lub Intel.

```
__private int gid = get_global_id(0);
if (gid < 10) {
    printf("I'm at %u\n", gid);
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
}
d[gid] = gid;
```

Listing 24 — bandwidth/kernel5.c

W badanym przykładzie (Listing 25) występuje nieskończona pętla *while*. Jednak od implementacji SDK zależy w jaki sposób zachowa się urządzenie obliczeniowe. Na SDK od AMD na karcie graficznej HD 4550 ***kernel* wykona się, aczkolwiek nie dotrze z oczywistych względów do końca**. Na tym samym SDK, ale na procesorze Intel E3 1220 ***kernel* się nie wykonuje, a SDK zwraca kod błędu o enigmatycznej postaci**. Jeśli jednak ten sam przykład uruchomimy na SDK od Intel na procesorze również od Intel, *kernel* się wykonuje, co powoduje że pętla rzeczywiście będzie blokowała czas procesora aż do momentu przerwania tego wykonania przez użytkownika.

```
__private int gid = get_global_id(0);
int j = 0;
while (1) {
    j = j + 1;
}
d[gid] = j;
```

Listing 25 — bandwidth/kernel6.c

Wniosek zatem jest taki, że w pewnych szczególnych wypadkach działanie *kernela* może być nieokreślone, a nawet różne na różnych SDK.

[228] Zarówno bariera lokalna

> CLK_LOCAL_MEM_FENCE

jak i ta globalna

> CLK_LOCAL_MEM_FENCE

odnosi się tylko i wyłącznie do synchronizacji elementów znajdujących się w tej samej grupie. Elementy z danej grupy wykonywane są na jednej jednostce obliczeniowej w ramach urządzenia i korzystają z pamięci lokalnej tej konkretnej jednostki. Z tej przyczyny zakłada się zasadniczo brak bezpośredniej możliwości synchronizacji tych elementów z poziomu *kernela*. Szczegółowo opisuje to poniższy fragment:

> „Since a barrier can only synchronize execution of the work items in the work group, what are the CLK _LOCAL_MEM_FENCE or CLK_GLOBAL _MEM_FENCE flags to the barrier function good for? In short, they control memory consistency for the work items in the work group. If you wrote a value to local memory with one work item in the work group, and you want to read it with a different work item in the work group, you'll want to fence local memory (this is the

> common case). Likewise, if you wrote a value to global memory with one work item in the work group, and you want to read it with a different work item in the work group, you'll want to fence global memory. Note specifically that there are no global memory consistency guarantees for work items executing in different work groups" [228]

Jeśli zatem korzystamy z pamięci lokalnej powinniśmy synchronizować (stosując barierę) lokalnie, jeśli natomiast korzystamy z pamięci lokalnej (dalej w ramach grupy) powinniśmy synchronizować globalnie (również stosując barierę). **Jeśli chcemy synchronizować globalnie pomiędzy wszystkimi grupami wówczas uzyskujemy to poprzez uruchomienie wielu *kerneli* i własnoręczne scalenie i synchronizację danych**.

4.14. Operacje atomiczne

Gwarancja spójności operacji arytmetycznych

Jak już zostało to wcześniej wskazane, uruchamianie *kerneli* jest realizowane przez mechanizm, który bierze pod uwagę specyfikę sprzętową „lejka” na wątki, grupowanie wątków w zestawy, zlecenie wykonywania zestawów na multiprocesorach, obecność pamięci lokalnej oraz możliwości synchronizacyjne. Jest to całkiem spory zbiór czynników, jakie należy za każdym razem mieć na uwadze pisząc kod do wykonania na urządzeniu obliczeniowym. Rozpatrywany przykład (uprzedzam, że celowo wadliwy, Listing 26) **próbuje modyfikować wartość *d[N]*, jednak nie będzie możliwości aby mogło to działanie zostać wykonane poprawnie**. Wykonywanie poszczególnych wątków jest zasadniczo od siebie niezależne.

```
__private int gid = get_global_id(0);
__private int group_id = get_group_id(0);
d[group_id] = d[group_id] + 1;
printf("%u %u %u \n", group_id, gid, d[group_id]);
```

Listing 26 — bandwidth/kernel7.c

Przy rozmiarze lokalnym dalej wynoszącym 4 moglibyśmy oczekiwać, że dla każdego identyfikatora grupy, w zbiorze wynikowym ujrzymy wartość… no właśnie ciężko stwierdzić jaką. Wynikiem w każdej grupie jest liczba 1. **Wszystkie wątki próbują wykonać operację „w tym samym” momencie**. Pokazuje to zatem konieczność zastosowania innych mechanizmów, a konkretnie operacji atomicznych.

Należy mieć na względzie ograniczenie w postaci **inkrementacji jedynie wartości całkowitych**. Nie możemy tymi funkcjami operować w żadnym bezpośrednim stopniu na wartościach zmiennoprzecinkowych. Używamy tutaj zmiennej lokalnej, ponieważ interesuje nas liczenie elementów w danej grupie. **Zmienne lokalne nie mogą być inicjowane podczas deklaracji, stąd konieczność rozdzielenia tych dwóch operacji**.

[011] W drugiej kolejności stawiamy lokalną barierę (Listing 27), tak aby zapewnić spójność tych zmiennych lokalnych w ramach grupy. Jeśli usuniemy barierę, wówczas wynik będzie niestety taki sam (przykład pochodzi z literatury, gdzie bariera ta w tym miejscu wygląda jak swego rodzaju nadgorliwość). Następny element to już właściwa **funkcja atomiczna**, czyli *atomic_inc*. Funkcja ta przyjmuje adres, pod którym będzie ona zmieniała wartość o jeden. Następnie mamy przypisanie wynikowej wartości inkrementacji (wykonanych tutaj przez wszystkie wątki z grupy) do bufora wynikowego. Na samym końcu mamy wypisanie wartości kontrolnych na wyjściu. Należy w tym przypadku stosowania funkcji *printf* mieć na uwadze jej specyficzną implementację dla OpenCL oraz znacząca ograniczenia (tj. modyfikatorów). Po wywołaniu inkrementacji atomicznej, możemy również wykonać synchronizację lokalną, gdyż modyfikujemy i odczytujemy zmienną lokalną.

> *Uwaga: niektóre urządzenia ze starszymi wersjami standardu mogą nie obsługiwać operacji atomicznych.*

```
__private int gid = get_global_id(0);
__private int group_id = get_group_id(0);
__local int shared;
shared = 0.0;
barrier(CLK_LOCAL_MEM_FENCE);
atomic_inc(&shared);
d[group_id] = shared;
printf("%u %u %f %i\n", group_id, gid, d[group_id], shared);
```

Listing 27 — bandwidth/kernel8.c

4.15. Pamięć lokalna

Dane współdzielone w ramach grupy

W OpenCL mamy do dyspozycji kilka poziomów pamięci, globalną, lokalną i prywatną (oraz stałe), a także pamięć gospodarza. Dostęp do nich jest **zróżnicowany ze względu na sprzęt, wersję obsługiwanego standardu oraz specyfikę programu. Rozmiar pamięci lokalnej** określony jest przez właściwość

CL_DEVICE_LOCAL_MEM_SIZE

która wskazuje, z jakiej wielkości możemy korzystać na **przechowanie danych w ramach pojedynczej grupy wątków**. Pamięć ta może być rzeczywiście lokalna względem multiprocesora strumieniowego lub globalna jak w przypadku CPU.

```
#define N (8*1024)
__kernel void sampleKernel(__global const float *a,
                           __global float *d) {
    __private int gid = get_global_id(0);
    __private int lid = get_local_id(0);
    __local int myArray[N];
    if (lid == 0) {
        for (int i=0; i<N; i++) {
            myArray[i] = i+1;
        }
        printf("%i, %i\n", gid, myArray[0]);
    }
}
```

Listing 28 — bandwidth/kernel12.c

Jeśli zatem wiemy, że mamy do dyspozycji 32 KB pamięci lokalnej, a rozmiar zmiennej typu *integer* to 4 bajty, wówczas możemy zaalokować przestrzeń na 8192 elementy (Listing 28)

bowiem tyle właśnie **zmieści się we wspomnianym rozmiarze**. Jeśli do *N* dodamy 1, kompilator zgłosi **wyjątek**.

```
Exception in thread „main” org.jocl.CLException: CL_OUT_OF_RE-
SOURCES
```

5. RÓŻNICE POMIĘDZY WERSJAMI STANDARDU

5.1. Nieokreślone zachowanie

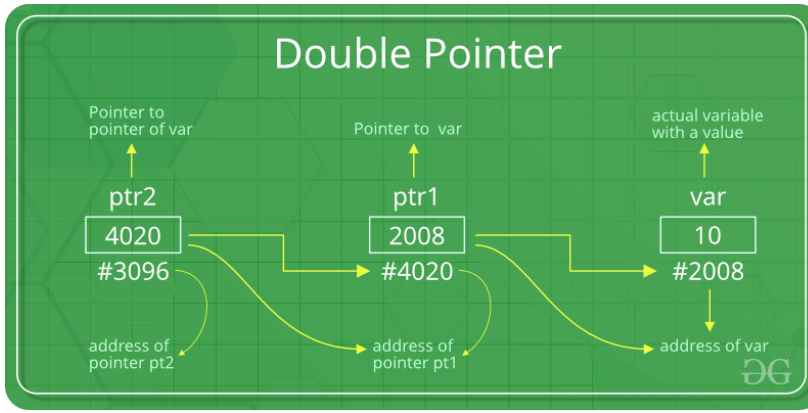
Kiedy wykraczamy poza definicję standardu (i próbujemy coś zepsuć...)

[012] [013] Jakkolwiek celem niniejszej książki nie jest przedstawienie specyfiki programowania w języku C, to wydaje się, że **różnice pomiędzy programami pisanymi wprost na CPU z użyciem standardowego kompilatora, a programami pisanymi w C99 na OpenCL** są warte wspomnienia.

```
void first(int *ptr) {
    *ptr = 2;
}

int main(int argc, char *argv[]) {
    int *ptr1;
    ptr1 = 0;
    printf("%p\n", ptr1);
    first(&ptr1);
    printf("%p\n", ptr1);
    return 0;
}
```

Listing 29 — reference_c/main2.c

W przykładzie (Listing 29) **powinno** się zastosować **podwójny wskaźnik** w nagłówku funkcji *first*. Jeśli tego nie zrobimy, wówczas w kompilatorze GCC otrzymamy ostrzeżenie o niekompatybilnym typie wskaźnika. Przekazujemy bowiem w wywołaniu adres wskaźnika *ptr1*. W takiej jednak postaci kod się kompiluje i wykonuje.



Rys. 20 — Podwójny wskaźnik [230]

[231] Jeśli weźmiemy ten sam przykład, tylko w C99 w OpenCL wówczas zostanie on poprawnie skompilowany i wykonany, ale tylko na CPU. **Jeśli spróbujemy zastosować podwójny wskaźnik, wówczas otrzymamy błąd kompilacji**.

```
error: passing '__local int *__private *" to parameter of type '__gene-
ric int *__generic *" changes address space of nested pointer
first(&ptr1);
```

Jeśli zastosujemy pojedynczy wskaźnik, jak poniżej, wówczas kod się skompiluje bez ostrzeżeń i jak wcześniej wspomniałem, jego **wykonanie na CPU będzie bezproblemowe.**

```
void first(int *ptr) {
    *ptr = 2;
}

__kernel void sampleKernel(__global const float *a, __global float *d) {
    __private int gid = get_global_id(0);
    __private int group_id = get_group_id(0);
    __local int *ptr1;
    ptr1 = 0;
    printf("ptr1: %i, %p\n", gid, ptr1);
    first(&ptr1);
    printf("ptr1: %i, %p\n", gid, ptr1);
}
```

Listing 30 — bandwidth/kernel11.c

Próba uruchomienia na NVIDIA GeForce 940MX (Listing 30) zakończy się **przekroczeniem czasu oczekiwania** po około 2 minutach wykonania. Czy powstaje z tego kodu coś, co powoduje zapętlenie czy zachowanie jest inne, tego nie wiadomo. Celem było jedynie pokazać, że programowanie w zasadniczo tym samym języku, ale na dwóch różnych platformach uruchomieniowych to czasami różne zachowania.

Uwaga: przy okazji wspomnę, że uruchamianie programu z poziomu IntelliJ IDEA to skrót Shift + F10. Dla Embarcadero Dev-C++ jest to F11.

5.2. Nieznaczne różnice

Rzutowanie w funkcji printf

Weźmy za przykład **podstawowy** kod w języku C (Listing 31). Definiujemy tutaj zmienną *var* oraz wskaźnik *ptr*. Wskaźnik *ptr* uzyskuje adres zmiennej *var*. Zmienna wskazywana przez wskaźnik jest kolejno ma zwiększaną wartość o 2, a następnie poprzez wywołanie funkcji *first* zwiększaną o 1, również za pomocą wskaźnika. Kontrolnie wypisujemy wskazywany adres, wartość wskazywaną przez wskaźnik oraz samą zmienną.

```c
#include <stdio.h>
#include <stdlib.h>

void first(int *ptr) {
    (*ptr)++;
}

int main(int argc, char *argv[]) {
    int var = 0;
    int *ptr = &var;
    printf("ptr: %p, %i, %i\n", ptr, *ptr, var);
    *ptr = *ptr + 2;
    printf("ptr: %p, %i, %i\n", ptr, *ptr, var);
    first(ptr);
    printf("ptr: %p, %i, %i\n", ptr, *ptr, var);
    return 0;
}
```

Listing 31 — reference_c/main. c

Wynikiem działania jest:

```
ptr: 000000000065FE14, 0, 0
ptr: 000000000065FE14, 2, 2
ptr: 000000000065FE14, 3, 3
```

Adres wskazywanego obszaru może nie być powtarzalny pomiędzy kolejnymi uruchomieniami. Należy zwrócić uwagę,

że aby uzyskać wypisanie na ekran wartości wskazywanej wskaźnikiem **nie trzeba wykonywać rzutowania wprost** do typu wskazywanego przez wskaźnik.

Dla porównania całkiem zbliżony kod w OpenCL. Mamy tutaj (Listing 32) zmienną *var* oraz wskaźnik *ptr*. Wskaźnik uzyskuje adres zmiennej. Następnie analogicznie do poprzedniego przykładu, zwiększamy wskazywaną zmienną o 2, a następnie z wykorzystaniem funkcji *first*, o 1.

```
void first(int *ptr) {
    (*ptr)++;
}

__kernel void sampleKernel(__global const float *a, __global float *d) {
    __private int gid = get_global_id(0);
    __private int group_id = get_group_id(0);
    int var;
    int *ptr = 0;
    printf("%i, %p\n", gid, ptr);
    var = 4;
    ptr = &var;
    printf("%i, %p, %i\n", gid, ptr, (int)*ptr);
    *ptr = *ptr + 2;
    printf("%i, %p, %i\n", gid, ptr, (int)*ptr);
    first(ptr);
    printf("%i, %p, %i\n", gid, ptr, (int)*ptr);
    d[gid] = (float)*ptr;
}
```

Listing 32 — bandwidth/kernel9.c

Wynikiem działania jest:

```
0, 0000000000000000
0, 000000D9301FDFF8, 4
0, 000000D9301FDFF8, 6
0, 000000D9301FDFF8, 7
```

Jak zatem widać zachowanie jest **podobne**. Dla osobnych elementów wykonywane działania są niezależne i powinny operować na osobnych przestrzeniach adresowych. **Ze względu na pewne różnice pomiędzy wersjami języka C oraz architekturą samych urządzeń, w przypadku funkcji *printf* należy** wykonać rzutowanie do typu *int*.

5.3. OpenCL 1.2 a 2.0

Zmiana przestrzeni adresowej wskaźników

[229] Poprzednie dwa przykłady prezentują zaledwie podstawowe aspekty programowania w języku C w porównaniu kompilacji GCC i OpenCL. Miały za zadanie potwierdzić, że w dalszych testach określone zachowanie ma uzasadnienie. Pierwszą konkretną różnicę, którą chciałbym zaakcentować to możliwość **wskazania określonej wersji standardu** przy kompilacji programu *kernela*. Robimy to przy okazji wywołania funkcji *initializeKernel* (*KernelConfigurationSet. java*):

```
clBuildProgram(this.program, 0, null, "-cl-std=CL1.2”, null, null);
```

Na 4 parametrze wywołania funkcji *clBuildProgram* możemy podać wartość

— cl-std=CL1.2

lub

— cl-std=CL2.0

lub każdą inną wersję standardu, które SDK i urządzenie jest w stanie przyjąć. Aby zweryfikować rzeczywisty wpływ zmiany wersji użyjemy następującego przykładu (Listing 33).

```
void foo(int *ptr, int _gid) {
    *ptr = *ptr + _gid;
}

__kernel void sampleKernel(__global const float *a, __global float *d) {
    __private int gid = get_global_id(0);
    __private int group_id = get_group_id(0);
    local int *ptr = 0;
    local int var;
    var = 2;
    ptr = &var;
    foo(ptr, gid);
    printf("%i, %p, %i, %i\n", gid, ptr, (int)*ptr, var);
    d[gid] = 0.1;
}
```

Listing 33 — bandwidth/kernel10.c

Przykład ten dla wersji OpenCL 2.0 (CL2.0) wykonuje się poprawnie. Jeśli jednak wskażemy wersję 1.2 (CL1.2) wówczas otrzymamy komunikat błędu:

> error: passing '__local int *__private” to parameter of type '__private int *” changes address space of pointer

Przyjmuje się, że funkcje bez określonych kwalifikatorów przestrzeni domyślnie wykorzystują przestrzeni prywatnej. Począwszy od wersji 2.0 możemy pominąć wskazywanie wprost tych kwalifikatorów, dzięki czemu kodu mamy do napisania mniej. **Nie ma bowiem potrzeby pisać tych samych funkcji z różnymi kwalifikatorami przestrzeni adresowej.**

Uwaga: uruchomienie tego przykładu jest praktycznie niemożliwe w przypadku urządzenia ATI Radeon HD 4550, gdyż po pierwsze nie posiada obsługi funkcji printf, a po drugie nie mamy możliwości ustawić go w tryb pracy większy niż OpenCL 1.1. Dostaniemy też ostrzeżenie o braku włączonej obsługi liczb typu double.

6. PRZYKŁADY JEDNOWYMIAROWE

6.1. Projekcja ortogonalna

Połączenie przykładu jedno i quasi wielowymiarowego

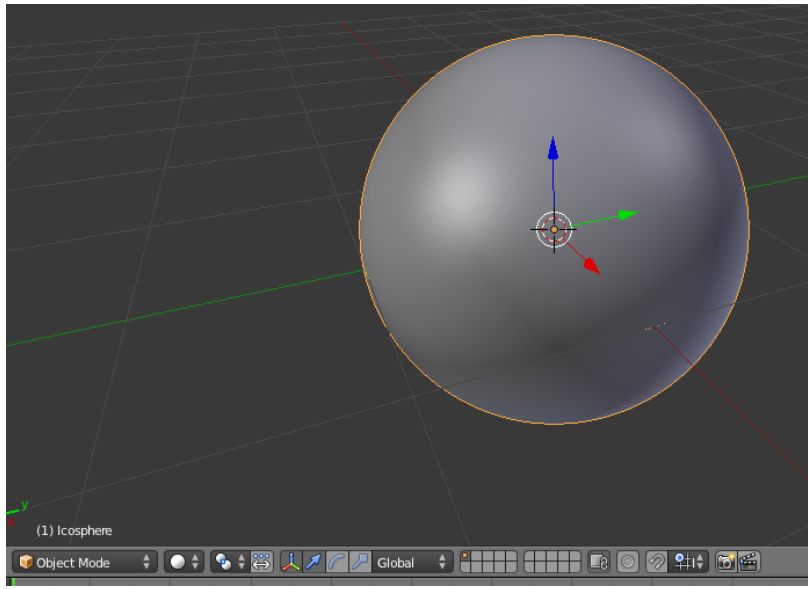
Pierwszym przykładem jaki chce zaprezentować w tym rozdziale jest **połączenie OpenCL z reprezentacją graficzną**. Wykorzystanym algorytmem jest rzut prostopadły, tj. **projekcja ortogonalna**. Algorytm ten pochodzi z mojego innego repozytorium, konkretnie *c64* z *GitHub*. Wykorzystany jest **model trójwymiarowy** wykonany w programie Blender 2.65. Sam **algorytm** wygląda w następujący sposób (Listing 34).

```
__kernel void sampleKernel(__global const int *a,
                           __global int *d)
{
    // gets
    __private int gid = get_global_id(0);
    __private int gr  = get_group_id(0);
    __private int lid = get_local_id(0);
    // calc
    if (lid == 0) {
        int start_x  = 100;
        int start_y  = 700;
        int start_z  = 500;
        int x2, y2, z2 = 0;
        int d1, d2, bx, by = 0;
        x2 = start_x + a[gid];
        y2 = start_y + a[gid+1];
        z2 = start_z + a[gid+2];
        d1 = z2/30;
        d2 = z2/20;
        bx = (x2*1) + d1;
        by = (y2*1) + d2 - 150;
        d[(gr*2)+0]  =  bx;
        d[(gr*2)+1]  =  by;
    } // lid == 0
}
```

Listing 34 — c64_3d/kernel. c

Uwaga: zasadniczo powinniśmy korzystać ze wszystkich wątków w grupie. W podanym przykładzie z 3 wątków w grupie tylko 1 wątek wykona rzeczywistą pracę. Jest to znane ograniczenie takiego podejścia. Jest to w końcu przykład jedynie quasi wielowymiarowy. Przykłady próbujące wykorzystać maksymalną ilość wątków znajdują się dalej (np. redukcja).

Algorytm zakłada **rozmiar lokalny wynoszący 3**, ponieważ mamy właśnie 3 wymiary wejściowe, acz samo zadanie jest jednowymiarowe. Przykład ten pokazuje, że nie musimy na siłę korzystać z wielowymiarowości. Obliczenia wykonywane są jednak zbiorczo w **zerowym** elemencie każdej grupy, a **wynik** dwuelementowy umieszczany w płaskiej tablicy. Dane **wejściowe** to również płaska tablica. Definiujemy **referencyjny punkt startowy** w zmiennych *start_x*, *start_y* oraz *start_z*. Projekcja odbywa się poprzez **dzielenie wsadu do wyniku współrzędnych** *x* i *y*, odpowiednio przez właściwą wartość dla wymiaru *x* i dla wymiaru *y*. Zmienne *bx* i *by* operują właśnie na tym wsadzie, przy czym wymiar *y* posiada jeszcze **przesunięcie** o stałą wartość. Finalnie, obliczenia przekazywane są do tablicy *d*, gdzie indeks elementu to **numer grupy** pobrany wcześniej, pomnożony przez liczbę wymiarów wynikowych i powiększony o **numer kolejny wymiaru**.



Rys. 21 — Icosphere w programie Blender (163 842 wierzchołki)

W programie Blender (Rys. 21) obiekt **przygotowujemy** i zapisujemy go jako plik w formacie *blend*. Następnie **eksportujemy** ten sam obiekt do formatu *obj*. Interesują nas tylko i wyłącznie wierzchołki, zatem eksport krawędzi, materiałów itp. możemy po prostu pominąć. Aby **przygotować** dane wejściowe do obliczeń, model przepuszczamy przez następujący program (Listing 35).

```
# OPEN FILE
file = File.new(ARGV[0], "r")
# MAIN LOOP
lines = []
while (line = file.gets)
    lines << line
end # while
# CLOSE FILE
file.close
# VERTICES
puts "===> VERTICES"
vertices_count = 0
lines.each_with_index do |line,index|
    split_line = line.split(" ")
    first_element = split_line[0]
    if first_element == 'v' then
        x = (split_line[1].to_f*100).to_i + 50
        y = (split_line[2].to_f*100).to_i - 150
        z = (split_line[3].to_f*100).to_i
        print "#{x},#{y},#{z},\n"
        vertices_count=vertices_count+1
    end # if
end # lines.each
puts "\n===> VERTICES COUNT = #{vertices_count}"
```

Listing 35 — c64_3d/0_parser. rb

Program uruchamiamy w następujący sposób:

ruby 0_parser. rb object. obj > sphere.model

Wynikiem jest **zbiór wierzchołków** o współrzędnych *x*, *y* i *z*. Są one przygotowane poprzez **względne znormalizowanie** ich postaci oraz **przesunięcie** tak aby prezentowane wierzchołki były „w kadrze”. Przykładowo:

50, -250, 0,
122, -194, 52,
23, -194, 85,

W programie przygotowującym (w module *c64_3d*) w języku Java, w klasie *Main* mamy następujący fragment, który **ładuje** uprzednio wygenerowane dane wejściowe. Proszę zwrócić uwagę (Listing 36), że klasa rozszerza *JFrame*.

```
public class Main extends JFrame {
    public static int[] vertices_3d;
    public static int vertices_3d_count;
    public static int vertices_2d_count;
    public static int[] vertices_2d;
    public static int n;
```

Listing 36 — Zmienne na dane o modelu

Następnie mamy kod (Listing 37) odpowiedzialny za **czytanie zawartości pliku z modelem** i wstawianie tych wartości do swego rodzaju **wektora**. Te same dane wstawiamy do płaskiej tablicy oraz uzupełniamy liczniki ilości elementów.

```
ArrayList<Integer> vector = new ArrayList<Integer>();
List<String> model = Files.readAllLines(Path.of("sphere.model"));
for (String line : model) {
    String[] parsedLine = line.split(",");
    vector.add(Integer.parseInt(parsedLine[0]));
    vector.add(Integer.parseInt(parsedLine[1]));
    vector.add(Integer.parseInt(parsedLine[2]));
}
System.out.println("Vector size: " + vector.size());
int size = vector.size();
vertices_3d = new int[size];
for (int i = 0; i <= size - 1; i++) {
    vertices_3d[i] = vector.get(i);
}
vertices_3d_count = vertices_3d.length;
vertices_2d_count = vertices_3d.length;
vertices_2d = new int[vertices_3d.length];
Main.n = vertices_3d.length;
```

Listing 37 — Odczytanie pliku z modelem

Aby móc **wyświetlić wynik** przekazujemy instancję *Runnable* do *SwingUtilities*. W metodzie *run* ustawiamy, żeby okno było **widoczne** (Listing 38).

```
Utils.log("8. Debug results");
c.printResults();

Utils.log("9. Initialize UI");
SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        new Main().setVisible(true);
    }
});
```

Listing 38 — Wyświetlenie wyniku w postaci graficznej

Ustawiamy rozmiar okna (Listing 39) i określamy, że w metodzie *draw* będziemy **rysować jednopunktowe odcinki** na podstawie wyliczonych współrzędnych przez program obliczeniowy przekazany do *kernela* OpenCL.

```
public Main() {
    super("Passing OpenCL calculations to UI");
    setSize(1000, 600);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
}
void draw(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    for (int i=0; i<=Main.vertices_2d.length-2; i+=2) {
        g2d.drawLine(Main.vertices_2d[i], Main.vertices_2d[i+1],
                     Main.vertices_2d[i], Main.vertices_2d[i+1]);
    }
}
public void paint(Graphics g) {
    super.paint(g);
    draw(g);
}
```

Listing 39 — Rysowanie metodą drawLine

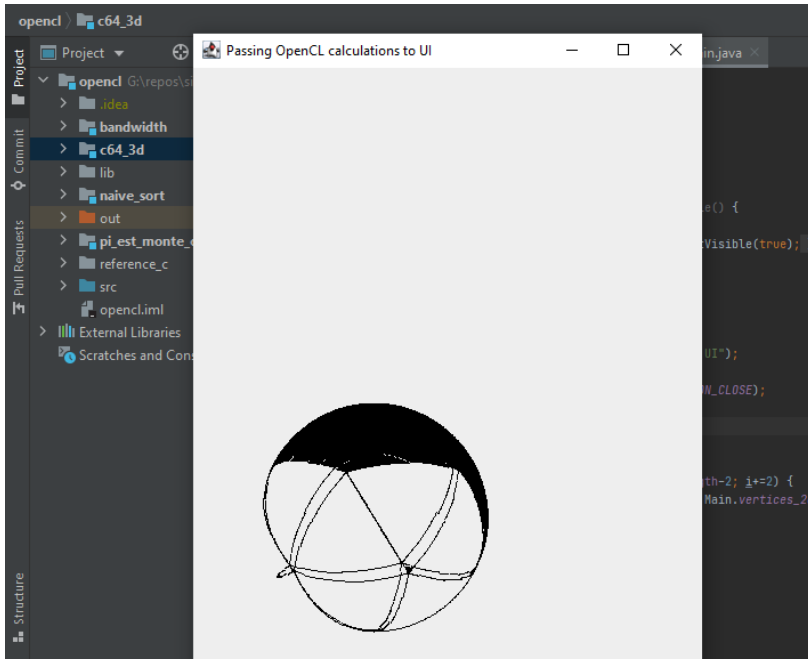
W klasie *KernelConfigurationSet* wskazujemy, że źródłowym zbiorem są współrzędne z **trzema wymiarami**, nato-

miast wynikiem jest zbiór współrzędnych **dwuwymiarowych**, takich jakie możliwe są do **odwzorowania** na płaszczyźnie (Listing 40).

```
public void initializeSrcArrayA() {
    srcA = Pointer.to(Main.vertices_3d);
}
public void initializeDstArray() {
    dst = Pointer.to(Main.vertices_2d);
}
```

Listing 40 — Dane wejściowe i wyjściowe

Uzyskałem ciekawe **porównanie** podczas uruchamiania tego programu na (**zestaw nr 1**) procesorze Intel i3—2328M ze zintegrowaną grafiką Intel HD Graphics 3000. Samo obliczenie OpenCL trwało **10 ms** uwzględniając zarówno procesowanie jak i przesył danych. Rysowanie w okno trwa jednak około 4 — 5 sekund (!), co wydaje się być absurdalnym wynikiem (Rys. 22).



Rys. 22 — Widoczny proces rysowania na procesorze Intel ze zintegrowaną grafiką

Ten sam program uruchomiony na (**zestaw nr 2**) AMD Radeon HD6350 wykonuje się w **30 ms**, a rysowanie trwa przysłowiowe **mgnienie oka**. W tym zestawie występuje również Intel Xeon E3 1220, ale nie bierze udziału w obliczeniu, pytanie czy bierze udział w rysowaniu (nie sądzę...). Jeśli na tym zestawie przełączymy obliczanie na procesor, wówczas czas trwania tego obliczenia i transferu danych łącznie wynosi niemal 100 ms, czyli **3 krotnie więcej**.

6.2. Metoda Monte-Carlo

Estymacja liczby Pi

[104] Mając obliczone pole kwadratu i koła wpisanego w ten kwadrat możemy wstecznie wywnioskować wartość liczby Pi. Oczywistym będzie, że to samo Pi byłoby nam potrzebne do obliczenia pola koła. Pojawia się tutaj jednak metoda Monte-Carlo. Polega ona na **losowaniu wartości współrzędnych od 0 do r**, które jest promieniem tegoż koła. Losując odpowiednio dużo punktów, powinniśmy otrzymać z pewnym prawdopodobieństwem **rozsądne przybliżenie liczby Pi**. Przykład z tego rozdziału znajduje się w module *pi_est_monte_carlo* (Listing 41).

```
__kernel void sampleKernel(__global const float *a,
        __global const float *b,
        __global int *d)
{
    const float r = 1.0f;
    const int idx = get_global_id(0);
    if (a[idx]*a[idx] + b[idx]*b[idx] < r) {
        d[idx] = 1;
    }
    else {
        d[idx] = 0;
    }
}
```

Listing 41 — pi_est_monte_carlo/kernel. c

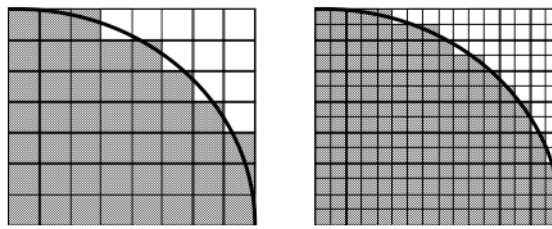
W *kernelu* przyjmujemy dane wejściowe z tablic *a* i *b*. Znajdują się tam liczby pseudolosowe wygenerowane w programie sterującym (Listing 42). **Sprawdzamy czy wylosowany punkt znajduje się wewnątrz koła (jego okręgu)**. Jeśli tak jest wówczas w danych wyjściowych zapisujemy liczbę 1, w prze-

ciwnym razie będzie to 0. Do losowania danych wejściowych używamy klasy *Random* i metody *nextFloat* jak w przykładzie (*pi_est_monte_carlo/KernelConfigurationSet. java*).

```
public void generateSampleRandomData() {
    System.out.println(" - Started randomizing");
    Random rd = new Random();
    for (int i = 0; i <= n - 1; i++) {
        srcArrayA[i] = rd.nextFloat();
        srcArrayB[i] = rd.nextFloat();
    }
    System.out.println(" - Finished randomizing");
}
```

Listing 42 — Przygotowanie danych testowych

Zasadniczo, **ilość** danych wejściowych ma znaczenie, ale bardziej to czy dane te powtarzają się, oraz jaki jest **zakres losowanych danych** (czyli *de facto* promień koła). W tym przykładzie jednak, przykład pozostaje przykładem. Postaram się nie wnikać w kwestie interpretacji wyniku (Rys. 23).



Rys. 23 — Precyzja wyniku przybliżenia uzależniona od danych wejściowych [104]

Dla próbki 33.5 mln (1024 * 1024 * 32) elementów tworzone są dwa bufory wejściowe i jeden bufor wyjściowy. Przy takich ilościach danych musimy spojrzeć na **maksimum dostępne do alokacji**. Odpowiada za to pole:

```
CL_DEVICE_MAX_MEM_ALLOC_SIZE
```

Dla Intel Core i3—2328M (**zestaw nr 1**) jest to 4066 MB. Jednak dla karty AMD Radeon HD6350 (**zestaw nr 2**) jest to zaledwie 192 MB. Na pierwszym zestawie oszacowanie liczby Pi z użyciem algorytmu Monte-Carlo to średnio 66 ms. Na drugim zestawie jest to 425 ms. W obu przypadkach stosowany jest rozmiar grupy o wymiarze 32 elementów. Dla zestawu nr 2 **degradacja wydajności** jest zauważalna pomiędzy rozmiarami 1 do 8. Od 16 do 128 (maksimum dla tej karty) wynik jest bardzo zbliżony.

> *Uwaga: Warto wspomnieć, że w przypadku urządzeń ATI/AMD nie mówimy o Warp, ale o Wave Front jako określenie na szerokość „osnowy". Przyznam, że takie określenie jakie jest stosowane przez AMD bardziej do mnie przemawia*

Kolejna ciekawa rzecz to fakt, że możemy uruchamiać programy OpenCL na procesorach Intel z wykorzystaniem SDK od AMD. Wydajność w tym przypadku **pozostawia wiele do życzenia**. Procesor Intel Xeon E3 1220 daje średnio 543 ms czasu wykonywania. Jest to niemal **10 krotnie dłużej** niż procesor Intel i3—2328M pracujący na SDK od Intela na zestawie nr 1.

Zestaw nr 3 to stacja HP Z800. Pierwsze urządzenie to karta Tesla K20Xm, gdzie uzyskujemy 46 ms. Drugie urządzenie to GTX 650 Ti, gdzie uzyskujemy 76 ms. Trzecie urządzenie do procesor Intel Xeon X5660 (w sumie 24 jednostki), gdzie

uzyskujemy 74 ms. Szczególnie interesującym wynikiem jest ten ostatni. **Procesor ten ma wyższe taktowanie rdzenia, większą ilość rdzeni, a przegrał z dużo słabszym Intel i3—2328M**. Według rankingu wydajności na UserBenchmark ten konkretny Xeon jest o 43% efektywnie szybszy. Ciekawe dlaczego nie ma to potwierdzenia w wynikach. Cieszy natomiast, że Tesla K20Xm posiadając **najwięcej jednostek obliczeniowych wykonała program najszybciej** z wszystkich testowanych dotychczas urządzeń.

Wyniki z trzech zestawów potwierdzają powszechne stwierdzenie opisujące OpenCL jako standard na kod, który można uruchamiać na **różnych platformach**, ale bez przenoszenia wszystkich **reguł dotyczących uzyskiwanych wydajności**. Na te wydajności składa się zatem więcej czynników, choćby takich jak **wersja i typ kontrolera pamięci, płyty głównej, przepustowość urządzeń, a nawet rozmiar pamięci operacyjnej**.

Uwaga: niniejszy przykład wymaga podania jako Working directory katalogu opencl. Tak samo jak w przypadku modułu bandwidth. Wyjątkiem tutaj jest moduł c64_3d, który wymaga ścieżki opencl\c64_3d

Po wykonaniu obliczeń i zaczytaniu bufora wynikowego wykonujemy **sumę całej tablicy** (*Arrays.stream.sum,* Listing 43), a następnie sumę mnożymy przez 4 i dzielimy wynik przez liczbę elementów.

```
float nf     = n*1.0F;
int sum      = Arrays.stream(dstArray).sum();
float result = (4.0F * sum) / nf;
System.out.println("PI EST MC: " + result);
```

Listing 43 — Sumowanie wyników szacowania

W wyniku uzyskujemy przykładowo wartość 3.1417913. Jest to zatem **jedynie przybliżenie**, ale tylko tyle ten algorytm oferuje. Podsumowując wydajność (Rys. 24).

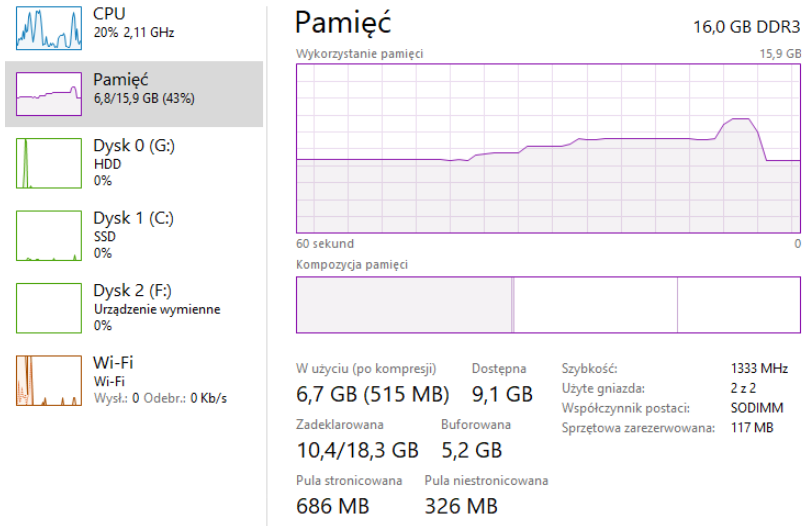
URZĄDZENIE	WYNIK
NVIDIA Tesla K20Xm	46ms
Intel Core i3-2328M	66ms
Intel Xeon X5660	74ms
NVIDIA Geforce GTX 650 Ti	76ms
AMD Radeon HD6350 (AMD SDK)	425ms
Intel Xeon E3 1220 (AMD SDK)	543ms

Rys. 24 — Zestawienie wydajności

Mam wrażenie, że AMD SDK ma jakiś problem i że wyniki nie są reprezentatywne.

Jeśli spojrzymy na **aspekt pamięciowy** programów w standardzie OpenCL to wydaje się, że nie ma zbyt dużo narzutu na postać obiektów w pamięci dla danych, które przekazujemy w buforach. Wykorzystując poprzedni przykład, czyli estymację liczby Pi z użyciem metody Monte-Carlo, ustalamy

ilość elementów bufora na 1024 * 1024 * 160 elementów. Urządzenie testowe to procesor Intel Core i3—2328M. **Buforów wejściowych mamy 2 oraz 1 bufor wynikowy**.



Rys. 25 — Alokacja buforów i uruchomienie kernela

W momencie **zaalokowania buforów** użycie pamięci operacyjnej wzrosło o 2 GB (Rys. 25). W momencie **uruchomienia *kernela*** użycie wzrasta o kolejne 2 GB. **Sumaryczne** użycie pamięci to 4 GB. Bufor źródłowy występuje po stronie gospodarza, dane są kopiowane do urządzenia obliczeniowego. Każdy bufor to około 650 MB wzrostu użycia. Przy założeniu, że są to tablice *float*, wówczas wiemy, że każdy element powinien mieć 4 bajty.

Pierwszą z poczynionych optymalizacji była zmiana tablicy wynikowej z liczb całkowitych na zmiennoprzecinkowe, tak aby można było łatwiej testować pisanie i czytanie buforów. Uzasadnienie jest zatem wyłącznie dydaktyczne (Listing 44).

```
float nf  = n*1.0F;
int sum = 0;
for (int i=0; i<dstArray.length; i++) {
    sum = sum + (int)dstArray[i];
}
float result = (4.0F * sum) / nf;
```

Listing 44 — Szacowanie liczby Pi

Wydzieliłem również klasy odpowiedzialne za statystykę wywołań, tj. *ExecutionStatistics* oraz *Entry*. Mają one teraz następującą postać (Listing 45).

```
package com.michalasobczak.opencl;
import org.jocl.cl_event;
import java.util.ArrayList;
import java.util.List;
public class ExecutionStatistics {
    private List<Entry> entries = new ArrayList<Entry>();
    public void addEntry(String name, cl_event event) {
        entries.add(new Entry(name, event));
    }
    public void clear()
    {
        entries.clear();
    }
    private void normalize() {
        long minQueuedTime = Long.MAX_VALUE;
        for (Entry entry : entries) {
            minQueuedTime = Math.min(minQueuedTime,
                                     entry.getQueuedTime());
        }
        for (Entry entry : entries) {
            entry.normalize(minQueuedTime);
        }
    }
    public void print() {
        normalize();
        for (Entry entry : entries) {
            entry.print();
        }
    }
}  // ExecutionStatistics
```

Listing 45 — opencl/ExecutionStatistics. java

Jest to raczej zabieg kosmetyczny. W kolejnych przykładach będzie natomiast mniej kodu do analizowania, gdyż te dwa pliki (Listing 46) teraz **przynależą do modułu bazowego**, czyli *opencl*.

```
package com.michalasobczak.opencl;
import org.jocl.CL;
import org.jocl.Pointer;
import org.jocl.Sizeof;
import org.jocl.cl_event;
public class Entry {
    private String name;
    private long submitTime[] = new long[1];
    private long queuedTime[] = new long[1];
    private long startTime[] = new long[1];
    private long endTime[] = new long[1];
    Entry(String name, cl_event event) {
        this.name = name;
        CL.clGetEventProfilingInfo(event, CL.CL_PROFILING_COMMAND_QUEUED,
                Sizeof.cl_ulong, Pointer.to(queuedTime), null);
        CL.clGetEventProfilingInfo(event, CL.CL_PROFILING_COMMAND_SUBMIT,
                Sizeof.cl_ulong, Pointer.to(submitTime), null);
        CL.clGetEventProfilingInfo(event, CL.CL_PROFILING_COMMAND_START,
                Sizeof.cl_ulong, Pointer.to(startTime), null);
        CL.clGetEventProfilingInfo(event, CL.CL_PROFILING_COMMAND_END,
                Sizeof.cl_ulong, Pointer.to(endTime), null);
    }
    void normalize(long baseTime) {
        submitTime[0] -= baseTime;
        queuedTime[0] -= baseTime;
        startTime[0] -= baseTime;
        endTime[0] -= baseTime;
    }
    long getQueuedTime() {
        return queuedTime[0];
    }
    void print() {
        System.out.println("Event "+name+": ");
        System.out.println("Queued : "+ String.format("%8.3f", queuedTime[0]/1e6)+" ms");
        System.out.println("Submit : "+ String.format("%8.3f", submitTime[0]/1e6)+" ms");
        System.out.println("Start  : "+ String.format("%8.3f", startTime[0]/1e6)+" ms");
        System.out.println("End    : "+ String.format("%8.3f", endTime[0]/1e6)+" ms");
        long duration = endTime[0]-startTime[0];
        System.out.println("Time   : "+ String.format("%8.3f", duration / 1e6)+" ms");
    }
}
```

Listing 46 — opencl/Entry. java

Przechodząc do samego algorytmu estymacji, zastanawiałem się **od ilu elementów zaczniemy uzyskiwać wartość wynikową zbliżoną do rzeczywistej liczby Pi**. I tak dla 64 elementów jest to 3.3, a dalej (Rys. 26).

ILOŚĆ ELEMENTÓW	WYNIK
128	3.18
256	3.09
512	3.11
1024	3.10
2048	3.16
4096	3.16
8128	3.15
1024 * 16	3.12
1024 * 32	3.14
1024 * 64	3.13
1024 * 128	3.14
1024 * 1024 * 128	3.1416

Rys. 26 — Zestawienie dokładności szacowań

Poziom precyzji do **2 miejsc po przecinku** uzyskujemy dopiero przy 1024 * 128 elementów. Jeśli interesuje nas większa „precyzja", wówczas będzie to dopiero 1024 * 1024 * 128 elementów.

[103] Jeśli bufor wejściowy przekazywany jest do urządzenia obliczeniowego razem z *kernelem*, wówczas **zapis bufora do urządzenia realizowany jest wraz z uruchomieniem *kernela***. Tak przynajmniej można wnioskować z faktu, że wprost tego nie wywołujemy w takim wypadku. Jeśli jednak chcemy mieć **większą kontrolę nad kolejnością przygotowania i ładowania buforów**, wówczas możemy taką operację wydzielić. Żeby jednak mieć pewność, że rzeczywiście w tym konkretnym momencie taki zapis ma miejsce, przeprowadziłem test (Listing 47).

```
cl_event writeEvent0 = new cl_event();
cl_event writeEvent1 = new cl_event();
if (withWriteEvent) {
    System.out.println("write 0");
    clEnqueueWriteBuffer(this.commandQueue,this.memObjects[0], true, 0, 0,
            KernelConfigurationSet.srcA, 0, null, writeEvent0);
    System.out.println("write 1");
    clEnqueueWriteBuffer(this.commandQueue,this.memObjects[1], true, 0, 0,
            KernelConfigurationSet.srcB, 0, null, writeEvent1);
    System.out.println("Waiting for write events...");
    CL.clWaitForEvents(1, new cl_event[]{writeEvent0});
    CL.clWaitForEvents(1, new cl_event[]{writeEvent1});
}
```

Listing 47 — clWaitForEvents

Pierwsze pytanie to **czy kopiowanie odbywa się w oczekiwanym momencie**, to znaczy po zakolejkowaniu takiego zapisu. Teoretycznie, stosując blokujące operacje nie będzie konieczności wywoływania ***clFinish*** lub ***clFlush***. Sprawdzam to, ponieważ na SDK od Intela kod ten przenosi oba bufory podczas kopiowania tego pierwszego. Takie przynajmniej odniosłem wrażenie. Jest to prawdopodobne, gdyż urządzenie gospodarza i urządzenie obliczeniowe to ten sam sprzęt, a podział jest tylko umowny i ustanowiony przez standard, a nie przez fizyczne ograniczenia.

> *Uwaga: Na SDK od AMD ten sam kod zwraca błąd **CL_INVALID_VALUE**, zarówno na GPU od AMD jak i CPU pod Intela. Nie wiem dlaczego. Tak samo jak nie wiem dlaczego czasami NVIDIA Tesla K20Xm zwraca kod błędu -9999.*

Weźmy próbkę 1024 * 1024 * 32 elementów. Na karcie NVIDIA GTX 650 Ti zaczynamy od poziomu 248 MB (tyle jest użyte, jeśli użyjemy tej karty jako wyjście ekranu). Po **kopiowaniu pierwszego bufora** uzyskujemy 397 MB. Po **skopiowaniu drugiego bufora** uzyskujemy 525 MB. Po **wykonaniu**

i zaczytaniu bufora wynikowego uzyskujemy 653 MB. Pierwsze przejście to zatem 149 MB różnicy. Drugie przejście to 128 MB. Trzecie przejście to również 128 MB. Ta i poprzednia wartość (tj. 128 MB) to właśnie ok. **33.5 mln elementów typu *float*, każdy po 4 bajty**.

[307] **Teoretyczna przepustowość** urządzenia obliczeniowego to aspekt obejmując częstotliwość pracy pamięci oraz szerokości szyny danych. Przykładowo jeśli urządzenie posiada pamięć taktowaną 800 MHz, a szyna to 64 bity, wówczas przepustowość wynosi 12.8 GB/s.

```
800 x 10^6 x ( 64/8 ) x 2 ) / 109 = 12.8 GB/sec
```

W równaniu pojawia się liczba 2, gdyż w tym konkretnym przypadku mówimy o pamięci typu DDR, czyli **pamięci o podwójnej jednostce danych**.

Na GTX 650 Ti wykonując 700 zapisów bufora 33.5 mln (1024 * 1024 * 32) elementów *float* do urządzenia obliczeniowego, trwa to około 1 sekundy. Oznacza to 22.4 GB/s transferu. Urządzenie to ma podane 86.4 GB/s jako maksimum (pomiędzy procesorem graficznym, a jego pamięcią). **Uzyskany wynik przepustowości efektywnej, jest niemal 4-krotnie gorszy od przepustowości teoretycznej**. Po pierwsze przepustowości efektywna i teoretyczna to dwa różne światy. Po drugie przepustowość pomiędzy gospodarzem, a obliczeniami to również dwa odrębne aspekty. Należy brać pod uwagę maksimum dla magistrali PCI-E, które dla wersji 1.0/1.1 wynosi 4GB/s, dla wersji 2.0/2.1 wynosi 8GB/s, dla wersji 3.0/3.1 wynosi 15 GB/s, a dla wersji 4.0 wynosi 31 GB/s. Finalnie wersja 5.0 to maksymalnie 63 GB/s. Na moim testowym zestawie posiadam PCI-E 2.0 x16, zatem moje maksimum może wynosić najwyżej 8GB/s. Skąd więc 22.4 GB/s transferu? Zapewne (i chyba tylko taka opcja pozostaje) napotykamy tutaj na ponowne wykorzystywanie przesłanych danych.

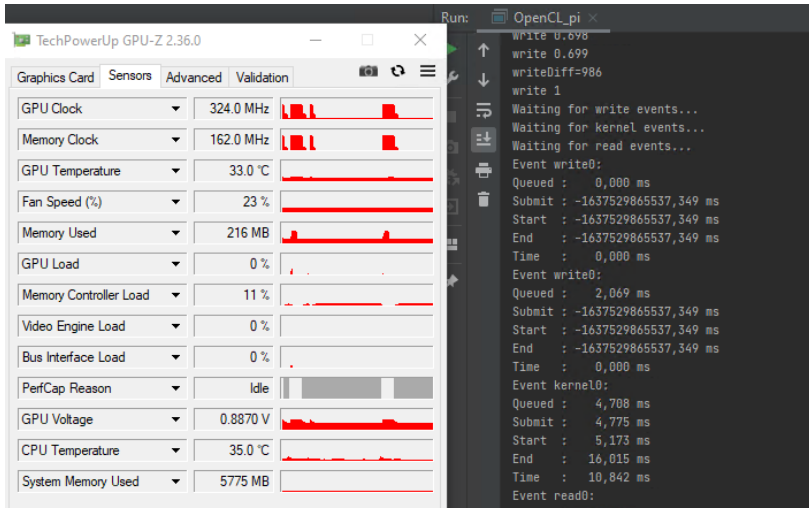
[307] Aby dany problem i algorytm go rozwiązujący miały sens w stosowaniu w OpenCL należy uwzględnić kilka aspek-

tów. **Problem musi dać się podzielić na równoległe elementy**. Należy pomóc urządzeniu we właściwym wykorzystaniu pamięci gospodarza i urządzenia obliczeniowego. **Ruch na magistrali PCI-E powinien być ograniczony do minimum**.

> *„The complexity of operations should justify the cost of moving data to the device. Code that transfers data for brief use by a small number of threads will see little or no performance lift. The ideal scenario is one in which many threads perform a substantial amount of work." [307]*

Jeśli już transmitujemy dane do urządzenia, to problem, który wymaga tych danych powinien być **na tyle złożony, aby uzasadnić koszt czasu transmisji danych** pomiędzy urządzeniami. Dane powinny **pozostawać na urządzeniu** obliczeniowym tak długo jak to tylko możliwe, **unikając zbędnych transferów**. Jeśli problem jest wielokrokowy, wówczas należy również starać się te **dane pozostawić bez ruchu**, nawet gdy ich przeliczenie byłoby szybsze i wygodniejsze u gospodarza.

Na koniec ciekawostka. Jeśli wykonamy kod na urządzeniu NVIDIA GeForce GTX 650 Ti i włączymy opcję odrębnej transmisji buforów, wówczas podczas pomiaru czasu otrzymamy następujący wynik (Rys. 27).



Rys. 27 — Pomiar czasu zapisu buforów na GTX 650 Ti.

6.3. Sortowanie

Szeregowanie danych

> *Uwaga: algorytm nawiązuje koncepcyjnie do trzech typów sortowań: sortowania przez zliczanie lub sortowania kubełkowego oraz sortowania bibliotecznego.*

Idealnym przypadkiem dla problemów rozwiązywanych za pomocą OpenCL są takie, w których przetwarzanie elementów jest od siebie **niezależne**. Jeśli tylko jednak musimy synchronizować się pomiędzy elementami, wówczas generuje to dodatkowe **ograniczenia algorytmu**. Należy pamiętać, że urządzenia wspierające standard OpenCL w wersji 1.1, nie wspierają operacji atomicznych zatem jedyną opcją w przypadku takich urządzeń jest stosować bariery.

Przykład sortowania (moduł *naive_sort*) będzie **najbardziej trywialnym z możliwych i dodatkowo może być czasami niezbyt dokładny. Należy go traktować jedynie jako ciekawostkę**. Wymaga danych ze zbioru o znanej rozpiętości.

W pierwszej kolejności (Listing 48) alokujemy tablicę i bufor wynikowy jako **100-krotność rozmiaru liczby elementów**. Taki sam rozmiar używamy w kolejkowaniu bufora wynikowego do realizacji przez urządzenie.

```
public float[] getDstArrayA() {
    System.out.println(" - Allocating return buffer");
    return new float[this.n * 100];
}
public void createBuffers() {
    // Allocate the memory objects for the input- and output data
    this.memObjects[0] = clCreateBuffer(this.context,
            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
            (long) Sizeof.cl_float * this.n, KernelConfigurationSet.srcA, null);
    this.memObjects[1] = clCreateBuffer(this.context, CL_MEM_READ_WRITE,
            (long) Sizeof.cl_float * this.n * 100, null, null);
}
public void runKernel(int iterations) {
    for (int i = 0; i<iterations; i++) {
        // Execute the kernel & Read the output data
        long aTime = ZonedDateTime.now().toInstant().toEpochMilli();
        clEnqueueNDRangeKernel(this.commandQueue, this.kernel, 1, null,
                this.global_work_size, this.local_work_size, 0, null, null);
        clEnqueueReadBuffer(this.commandQueue, this.memObjects[1],
                CL_TRUE, 0, (long) n * Sizeof.cl_float*100,
                KernelConfigurationSet.dst, 0, null, null);
        long bTime = ZonedDateTime.now().toInstant().toEpochMilli();
        System.out.println("Took OpenCL read result: "
                           + String.valueOf(bTime - aTime) + "ms");
    }
}
```

Listing 48 — Wielokrotne uruchamianie kernela

W *kernelu* natomiast (Listing 49) obliczamy jaki mamy „krok" poszczególnych elementów. Przy założeniu, że liczby losowe są z zakresu od 0.0 do 1.0, wówczas przy ilości elementów N=1024, każdy taki krok wynosi 0.0009765625.

```
__kernel void sampleKernel(__global const float *a,
        __global float *d)
{
    __private int gc  = get_num_groups(0);
    __private int ls  = get_local_size(0);
    __private int gs  = gc*ls;
    float per_item = 1.0/gs;
    __private int gid = get_global_id(0);
    __private int lid = get_local_id(0);
    __private float current = a[gid];
    __private int location = (int)((current/per_item)*100);
    printf("%u, %u, %f -> %u\n", lid, gid, current, location);
    d[location] = current;
}
```

Listing 49 — naive_sort/kernel. c|

Finalne umiejscowienie elementu w tablicy wynikowej polega na obliczeniu pozycji na podstawie kroku. Jest to zatem metoda szacunkowa, która może powodować braki w wyniku. Jeśli mówimy o obliczeniach, gdzie precyzja jest wymaga wówczas takie szacunkowe rozwiązania odpadają. Jeśli jednak mówimy o problemach, gdzie rozwiązanie szacunkowe jest wystarczając, wtedy możemy coś takiego rozważyć. **Przykład ten pokazuje ile możemy uzyskać jeśli będziemy chcieli za wszelką cenę uniknąć synchronizacji**.

W przykładzie, gdzie mamy 8 elementów, a rozmiar lokalny to 4, uzyskujemy następujący wynik kwalifikowania elementów po poszczególnych pozycji:

```
0, 0, 0.506890 -> 405
1, 1, 0.388219 -> 310
2, 2, 0.791039 -> 632
3, 3, 0.925037 -> 740
0, 4, 0.231281 -> 185
1, 5, 0.264114 -> 211
2, 6, 0.861319 -> 689
3, 7, 0.750138 -> 600
```

Wynik to:

```
nonZeroElements: [0.23128092, 0.26411432, 0.38821888,
0.50688976, 0.75013787, 0.7910388, 0.8613192, 0.9250373]
```

Na takiej małej próbce mamy wynik poprawny zarówno pod kątem kompletności danych jak i ich uszeregowania. Jeśli jednak zwiększymy próbkę znacząco, do np. 1 mln elementów, wówczas wynik może być nieprecyzyjny. Dodatkowo dochodzi kwestia **ograniczeń pamięciowych**. Jeśli chcemy sortować na starszym urządzeniu bez obsługi operacji atomicznych i unikać stosowania barier, to i tak natrafimy na problem, że urządzenia te mają niewielkie rozmiary dostępnych pamięci.

Szukając informacji o pasujących algorytmach sortowania dla środowiska równoległego natrafiłem na książkę „*OpenCL in Action*” [014], w której czytam:

> *„One fascinating aspect of OpenCL is that you don't have to configure these loops in your kernel. Instead, your kernel only executes code that would lie inside the innermost loop." [014]*

Prawdą jest, że mając kod, który operuje na wielu pętlach lub nawet rekurencji przechodząc na OpenCL do *kernela* wyciągamy jedynie **sam środek przetwarzania**. Nie jest jednak precyzyjnym stwierdzeniem, że nie musimy konfigurować naszych pętli i rekursji. Tej drugiej w ogóle w OpenCL nie uzyskamy ze względu na ograniczenia standardu. Jeśli chodzi natomiast o konfigurację pętli, to aby móc realizować różnego rodzaju algorytmy, będziemy musieli brać pod uwagę i rozmiar globalny i rozmiar lokalny, a także bariery, co w efekcie oznacza, że pętle będziemy konfigurować tylko na innym poziomie. **Wszelkie algorytmy sortowania oparte o rekurencji odpadają już z definicji**. Należy albo szukać **innych algorytmów** albo **modyfikować** te operujące na rekurencji tak aby obchodziły się bez niej.

6.4. Redukcja

Wykorzystanie pamięci lokalnej do sumowania

W dotychczas zaprezentowanych przykładach starałem się **unikać nadmiernego stosowania pamięci lokalnej**, ponieważ jakiekolwiek odwołania do pamięci znacząco ograniczają sensowność używania OpenCL do obliczeń. Istotą jest aby **maksymalizować zrównoleglenie prac, minimalizować synchronizację pomiędzy nimi oraz limitować odczyty i zapisy pamięci**. Problemy rzeczywiste jednak wymagają przeważnie aby albo intensywnie korzystać z pamięci albo mocno synchronizować i to nie tylko w ramach grupy, ale najlepiej pomiędzy wszystkimi grupami, co jak wiemy jest niemożliwe do wykonania w jednym przebiegu *kerneli* OpenCL.

[232] Po pierwsze nie ma możliwości wypełnienia lokalnej pamięci z poziomu gospodarza. Pamięć lokalna może zostać przekazana jako niezainicjalizowany bufor (Listing 50) w postaci argumentu *kernela*. Pamięć taka może być również użyta i inicjalizowana już bezpośrednio w *kernelu*. Stosowanie pamięci lokalnej jest szczególnie istotne w urządzeniach wspierających jedynie wersję standardu 1.0 oraz 1.1 ponieważ **nie umożliwiają one stosowania tablic zmiennych długości**.

[014] Redukcję zaczynamy od **zdefiniowania pamięci lokalnej** w *KernelConfigurationSet.initializeKernel*:

```
clSetKernelArg(this.kernel, 0, Sizeof.cl_mem, Pointer.to(this.memObjects[0]));
clSetKernelArg(this.kernel, 1, Sizeof.cl_mem, Pointer.to(this.memObjects[1]));
clSetKernelArg(this.kernel, 2, Sizeof.cl_float * this.loc, null);
```

Listing 50 — Argumenty kernela

Deklarujemy poza rozmiarem globalnym, również rozmiar lokalny (Listing 51), tak aby można było wygodniej nim operować podczas testów różnych rozmiarów:

```
    private final int n;
    private final int loc;
    public KernelConfigurationSet(int n, int loc) {
        this.n = n;
        this.loc = loc;
        System.out.println(" - KernelConfigurationSet");
    }
    public void configureWork() {
        this.global_work_size = new long[] { this.n } ;
        this.local_work_size  = new long[] { this.loc };
    }
```

Listing 51 — Rozmiar lokalny

W *kernelu* stosujemy **pamięć lokalną, pętlę redukcyjną oraz dwie bariery** (Listing 52).

```
__kernel void sampleKernel(__global const float *a,
                           __global float *d,
                           __local float *partial_sums) {
    __private int gid = get_global_id(0);
    __private int lid = get_local_id(0);
    __private int group_id = get_group_id(0);
    partial_sums[lid] = a[gid];
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int i=ls/2; i>0; i >>= 1) {
        if (lid < i) {
            partial_sums[lid] += partial_sums[lid + i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (lid == 0) {
        d[group_id] = partial_sums[0];
    }
}
```

Listing 52 — reduction/kernel. java

Algorytm ten składa się z 4 kroków. **Pierwszy** jest to zadeklarowanie pamięci lokalnej z poziomu gospodarza jako

argumentu do *kernela*, a następnie załadowanie danych wejściowych do tejże pamięci posługując się indeksem lokalnym. Przy założeniu, że mamy 16 elementów, a grupa ma rozmiar 4, również rozmiar globalny to 16, a rozmiar lokalny to 4. Daje to razem 4 grupy po 4 elementy każda. Każda grupa otrzymują swoją przestrzeń lokalną na multiprocesorze strumieniowym, do której inne grupy nie mają dostępu. Pod każde z 4 elementów kopiowany jest kolejny element wskazywany przez indeks lokalny. Pamięć lokalna jest ograniczona nawet jeśli realizowana sprzętowo jest w pamięci globalnej. Przeważnie jest to 32 lub 48 KB. Daje to od 8 do 12 tys. zmiennych typu *float*. Rozmiar lokalny jednakże jest również ograniczony, tj. rozmiar grupy maksymalny możliwy do stosowania. Przeważnie będzie to od 128 (starsze GPU), 1024 (nowsze GPU) do 8192 elementów (CPU). Mając skopiowane elementy z pamięci globalnej do właściwych danym grupom pamięciom lokalnym, przechodzimy do **drugiego** kroku. Jest nim postawienie bariery, która oznacza, że wszystkie wątki z danej grupy muszą dotrzeć do tego punktu aby mogły iść dalej. Oznacza to *de facto* ich synchronizację. **Trzeci** krok to pętla redukcyjna z kolejną barierą. **Zaczynamy od połowy rozmiaru lokalnego i z każdym krokiem dzieląc go na pół sumujemy kolejne elementu z obu połówek grupy. Każdorazowe przejście pętli to bariera w ramach grupy**. **Czwarty**, ostatni krok to przepisanie sumy częściowej do tablicy wynikowej, ale tylko w zerowym wątku każdej z grupy.

Po pierwsze przykład ten wykorzystuje wszystkie wątki jedynie w momencie, gdy kopiowane są dane z pamięci globalnej do lokalnej, co jest sporym ograniczeniem (warunek *lid* < *i*). Po drugie używamy ograniczonych zasobów pamięci lokalnej i rozmiaru maksymalnego grupy. Po trzecie w końcu, wynikiem tego algorytmu jest jedynie suma każdej z grup, a nie finalny wynik. Aby ten finalny wynik uzyskać musimy wykonać dodatkowe czynności. Możemy albo manualnie (w postaci rozwinięcia pętli) sumować elementy, bądź z poziomu gospodarza

wielokrotnie uruchamiać *kernel* celem **ponownego przeliczania**, tym razem sum.

Na próbce 67 mln elementów na karcie Quadro 4200M obliczenie 65 tys. sum częściowych zajmuje 546 ms. Na procesorze Intel i7—2640M jest to 313ms. Na karcie RTX 3050 Ti trwa to 171 ms.

6.5. Sortowanie kombinowane

Testy różnych podejść

Realizując jakikolwiek algorytm sortowania równoległego będziemy musieli korzystać z pamięci lokalnej i synchronizowć elementy. Dlatego warto pamiętać o tym, że **ilość zasobów jest ograniczona**:

> *„Shared memory is allocated per block. The shared memory per block is limited by the shared memory per SM. If one block uses 24kb shared memory in the above K420 GPU, two blocks may stay in the same SM. The shared memory is split into banks. It is important to understand this in order to write program avoiding banks confict." [233]*

[308] **Dostęp do pamięci globalnej to 400 — 600 cykli**, natomiast **dostęp do pamięci lokalnej to zaledwie 24 cykle**. Dostęp do pamięci prywatnej jest zapewne jeszcze krótszy, ale brak ku temu danych.

[234] Jeśli chodzi natomiast o **pamięć prywatną** to jej **rozmiar i lokalizacja** nie są określone przez standard. W zależności od architektury, implementacja może być znacząco różna. Tak jak nie ma teoretycznego limitu, tak za każdym razem natrafimy na limit rzeczywiście zastosowany czy to w sprzęcie czy w sterowniku OpenCL. Możemy przyjąć jednak, że póki algorytm działa akceptowalnie szybko, to zmienne prywatne są przechowywane w rejestrach lokalnych danego procesora strumieniowego. Jeśli jednak zauważymy regres w wydajności, który został zainicjowany przez zwiększenie ilości danych prywatnych, wówczas najpewniej

zostały one **ulokowane w pamięci globalnej**, której czas dostępu jest zasadniczo dłuższy niż rejestru prywatnego. [308] **Maksymalna liczba rejestrów** per pojedynczy wątek jest określona przez generację sprzętu. Dla CC 3.5 mamy 64 tys. 32-bitowych rejestrów per każdym multiprocesor strumieniowy. Ograniczenie per wątek to 255 sztuk.

Wydajność sprzętu możemy mierzyć **dla każdego z jego komponentu z osobna, lub jako całości**. Przykładowo, jeśli mówimy o procesorach CPU to mierzymy prędkość wykonywania **instrukcji**, czyli ile **cykli** potrzeba na poszczególne operacje aby wykonać program w całości. Jeśli operacje trwają tyle samo cykli, a **częstotliwość** pracy jest taka sama, wówczas teoretyczna prędkość powinna być bardzo zbliżona. Jeśli procesor ma więcej niż jeden rdzeń, wówczas będzie **synchronizował** dostęp do pamięci *cache* oraz ewentualnie wykorzystywał algorytmy **predykcyjnego rozgałęziania ścieżki wykonania**. Powinna interesować nas też sumaryczna ilość tranzystorów oraz **wielkość procesu technologicznego**, gdyż warunkuje to użycie określonej mocy dla takiego urządzenia, a co za tym idzie wymaganego chłodzenia i finalnie, uzyskiwanej wydajności pracy przy założonych stratach. Tak wygląda **teoria**.

Praktyka natomiast to w tym wypadku kolejny test sortowania (Listing 53), ale tym razem czysto teoretyczny, gdyż operujący na pojedynczym wątku na karcie graficznej, tak aby móc porównać wydajność per rdzeń, czyli procesor strumieniowy.

Do **testu** użyłem dwóch kart. Pierwsza z nich to NVIDIA NVS (Quadro) 4200M gdzie jednostka cieniująca posiada taktowanie maksymalne 1480 MHz. Druga karta to NVIDIA RTX 3050 Ti (80W) z maksymalną częstotliwością pracy procesora strumieniowego wynoszącą 1695 MHZ. Chciałbym podkreślić jasno, że te karty dzieli 10 lat rozwoju technologii procesorowej (40 nm vs 8 nm). Częstotliwość pracy to zaledwie jeden z czynników jaki należy brać tutaj pod uwagę.

```
#define MAX 1024*2
__kernel void sampleKernel(__global const float *a,
        __global       float *d,
        __local        float *local_data) {
    __private int gid = get_global_id(0);
    __private int group_id = get_group_id(0);
    __private float tmp = 0.0f;
    //
    for (int i=0; i<MAX; i++) {
        local_data[i] = a[i];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if (gid == 0) {
        for (int t=0;t<MAX-1;t++) {
            for (int u=0;u<MAX-t-1;u++) {
                if (local_data[u] > local_data[u+1]) {
                    tmp = local_data[u];
                    local_data[u] = local_data[u+1];
                    local_data[u+1] = tmp;
                } // bubble if and swap
            } // bubble B
        } // bubble A
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int g=0; g<MAX; g++) {
        d[(group_id*MAX)+g] = local_data[g];
    }
}
```

Listing 53 — sort2/kernel2.c

Przy założeniu, że rozmiar lokalny i globalny wynosi 1, a rozmiar tablicy wejściowej i jednakowej tablicy lokalnej to MAX = 1024 dla NVS 4200M sortowanie bąbelkowe takiego zbioru trwa 35ms. Na RTX 3050 Ti takie samo sortowanie trwa zaledwie 10 ms. Jest to zatem ponad 3 krotnie szybciej.

Jednocześnie, posortowanie tego zbioru na procesorze, na platformie Java trwa poniżej 1 ms zarówno dla i7—2640M jak i i5—10200H. Wniosek jest taki, że **siłą programowania w standardzie OpenCL jest złożoność obliczeniowa**, a nie algorytmy, gdzie mamy dużo dostępów do danych. **Dostęp do danych i synchronizacja wątków w grupie są bardzo czasochłonne**.

```
__kernel void sampleKernel(__global const float *a, __global float *d, __local float *local_data) {
    __private int ls  = get_local_size(0);
    __private int gid = get_global_id(0);
    __private int lid = get_local_id(0);
    __private int group_id = get_group_id(0);
    float tmp = 0.0f;
    float per_element = 0.0f;
    int counter = 0;
    local_data[lid] = a[gid];
    barrier(CLK_LOCAL_MEM_FENCE);
    // step 2, 4, 8, 16, 32 itd.
```

Listing 54a — sort2/kernel. c

```
for (int step=2; step<=ls; step=step*2) {
    // UWAGA:
    // Tylko dla elementów zerowych w danych podgrupach grup roboczych
    if (lid % step == 0) {
        counter = 1;
        //printf("group_id=%u, lid=%u, step=%u\n", group_id, lid, step);
        for (int e=lid; e<lid+step; e++) {
            // Uwaga:
            // Dla podgrup 2 elementowych wykonujemy sort-swap
            if (step == 2 && e % 2 == 0) {
                if (local_data[e] > local_data[e+1]) {
                    tmp = local_data[e];
                    local_data[e] = local_data[e+1];
                    local_data[e+1] = tmp;
                }
            }
            // Uwaga:
            // Dla podgrup o większym rozmiarze od 4 do rozmiaru lokalnego
            // wykonujemy inny typ szeregowania danych
            else if (step >= 4 && step < ls && (e-lid)<step/2) {
                if (local_data[e] > local_data[e+(step/2)]) {
                    tmp = local_data[e];
                    local_data[e] = local_data[e+(step/2)];
                    local_data[e+(step/2)] = tmp;
                }
                per_element = 1.0/step;
            }
            counter = counter + 1;
        } // e/lid for
        // BUBBLE
        if (step == ls) {
            for (int t=0;t<step-1;t++) {
                for (int u=0;u<step-t-1;u++) {
                    if (local_data[u] > local_data[u+1]) {
                        tmp = local_data[u];
                        local_data[u] = local_data[u+1];
                        local_data[u+1] = tmp;
                    } // bubble if and swap
                } // bubble B
            } // bubble A
        } // final step
    } // zero element
    // UWAGA:
    // Nie ma potrzeby aby elementy inne niż 0 czekały,
    // i tak nic więcej nie będą robiły
    // chyba, że chcemy używać printf
    barrier(CLK_LOCAL_MEM_FENCE);
} // step for
```

Listing 54b — sort2/kernel. c

```
    // Uwaga:
    // Na koniec dla 0 elementu
    if (lid == 0) {
        for (int g=0; g<ls; g++) {
            d[(group_id*ls)+g] = local_data[g];
        }
    }
}
```

Listing 54c — sort2/kernel. c

Dla porównania przygotowałem nieco bardziej złożony przykład (Listing 54). Używa on trzech rodzajów sortowań. Zbiór lokalny (skopiowany z globalnej pamięci), np. o rozmiarze maksymalnym 1024 jest **dzielony na podgrupy** 2, 4, 8, 16, 32, 64, 128, 256, 512 oraz 1024 elementy, a każdym z kolejnych kroków pętli. Dla par elementów, czyli przejścia z krokiem o wielkości 2 wykonujemy zwykłe porównanie elementów z ich zamianą w wypadku, gdy ich szeregowanie jest niezgodne z oczekiwanym. W wypadku **kolejnych przebiegów**, tj. od 4 do 512 stosujemy sortowanie połówkowe sortując każdą z lewych połówek podgrupy względem ich prawych stron. Powinno to znacząco wpłynąć na **porządek zbioru** co w wypadku **ostatniego sortowania bąbelkowego** może mieć znaczenie. Sortowanie takie, stosujemy tylko w ostatnim kroku, kiedy rozmiar podgrupy, tj. kroku jest równy rozmiarowi lokalnemu grupy.

Miałem nadzieje, że **ograniczając ilość odwołań do pamięci** uda się uzyskać lepszy rezultat niż 35 ms, ale byłem w błędzie. Takie podejście niewiele zmienia. Należy znacząco zmienić podejście do danych, inaczej będziemy oscylować wokoło tych samych wyników. Oczywiście przy założeniu, że algorytmy są iteracyjne z racji tego, że nie mamy dostępu do rekurencji. Jeśli jednak weźmiemy pod uwagę inne algorytmy iteracyjne, nierekursywne, to w zasadzie zostaje nam **sorto-**

wanie bitoniczne oraz pozycyjne (*radix sort*). Możemy też rozważyć prezentowany wcześniej przykład sortowania oparty koncepcyjnie na sortowaniu bibliotecznym.

6.6. Sortowanie z przesunięciem

Uruchamianie kerneli z przesunięciem

Nie musimy się ograniczać do uruchamiania jednego tylko i wyłącznie *kernela* na zadanym zbiorze danych. W module *sort2* znajduje się kolejny program testowy, tj. *kernel3.c*. Jest to kolejna próba (Listing 55) wykorzystania możliwości, jakie daje standard OpenCL. **Uruchamiany będzie tutaj ten sam *kernel* wielokrotnie**.

```
#define MAX 1024*16
__kernel void sampleKernel(__global float *a,
                           __global float *d,
                           __local float *local_data) {
    __private int ls  = get_local_size(0);
    __private int gid = get_global_id(0);
    __private int lid = get_local_id(0);
    __private int group_id = get_group_id(0);
    float tmp = 0.0;
    if (d[gid] == 0L) {
        d[gid] = a[gid];
        barrier(CLK_GLOBAL_MEM_FENCE);
    }
    if (gid < MAX) {
        if (lid % 2 == 0) {
            if (d[gid] > d[gid+1]) {
                tmp = d[gid];
                d[gid] = d[gid+1];
                d[gid+1] = tmp;
                barrier(CLK_GLOBAL_MEM_FENCE);
            }
        }
    }
}
```

Listing 55 — sort2/kernel3.c

Aby móc uruchomić *kernel* wielokrotnie (Listing 56), dla algorytmu określamy, że jest to N-razy, czyli tyle ile jest elementów w zbiorze. Grupy są **dwuelementowe**

tak aby móc sprawdzać ich podzielność przez dwa na potrzeby sortowania.

```
cl_event kernelEvent0 = new cl_event();
for (int j=0; j<=(this.n/2); j++) {
    clEnqueueNDRangeKernel(this.commandQueue, this.kernel, 1, off0,
            this.global_work_size, this.local_work_size, 0, null, kernelEvent0);
    clFinish(this.commandQueue);
    clEnqueueNDRangeKernel(this.commandQueue, this.kernel, 1, off1,
            this.global_work_size, this.local_work_size, 0, null, kernelEvent0);
    clFinish(this.commandQueue);
}
```

Listing 56 — clEnqueueNDRangeKernel

Aby mieć pewność, że *kernele* są nie tylko przekazywane do kolejki, ale również od razu wykonywane, wywoływana jest metoda *clFinish*. **Czynnikiem zmiennym w tym algorytmie jest *offset*, czyli przesunięcie stosowane na identyfikatorach w ramach pamięci globalnej**. Przekazując naprzemiennie 0 oraz 1 jako przesunięcie powodujemy, że za każdym razem para danych lokalnych może być inna. Efekt działania tak skonstruowanego mechanizmu jest jednak **wątpliwy**. Co prawda czas działania pojedynczego przejścia jest marginalny, tj. poniżej 1 ms, jednak jeśli uruchomimy *kernel* wiele tysięcy razy, wówczas napotykamy na narzut czasowy związany z samym uruchomieniem *kernela*. Koncepcja ta może zdawać egzamin tylko wtedy, gdy ograniczymy ilość następujących po sobie wywołań *kernela* do niezbędnego minimum, nie traktując tego zabiegu jako elementu algorytmu.

7. PRZYKŁADY WIELOWYMIAROWE

Dotychczas prezentowane były przykłady jednowymiarowe, głównie dlatego, że standard udostępnia taką opcję. **Żeby jednak poznać główny powód, dla którego moglibyśmy chcieć korzystać z OpenCL należy zacząć stosować programy wielowymiarowe**. Z pewnością można uzyskać satysfakcjonujące rezultaty również z problemami jednowymiarowymi, aczkolwiek należy mieć na względzie, że będziemy przede wszystkim ograniczeni brakiem możliwości synchronizacji pomiędzy grupami. Przynajmniej bez dodatkowego kodu, który i tak może spowodować, że całość ze względu na transfer danych pomiędzy pamięciami o różnych prędkościach, może okazać się nieopłacalna. Obliczenia matematyczne z określonymi algorytmami to główne zastosowanie. Można też wskazać, że urządzenia obsługujące standard OpenCL mogą na bieżąco wspomagać działanie głównych procesorów, wtedy nie muszą być one wcale szybsze, **wystarczy że pełnią rolę pomocnika**.

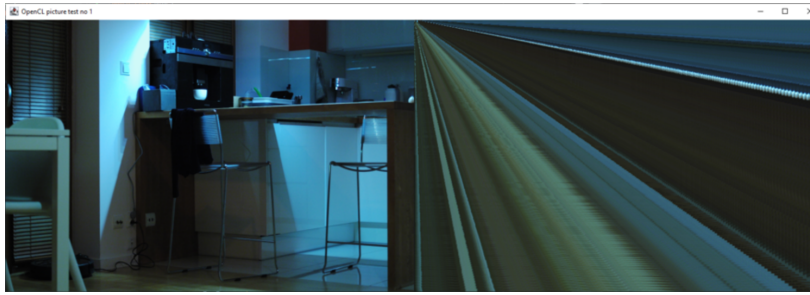
7.1. Przeskalowanie

Przykład ten jest przykładem hybrydowym. Działać może bowiem albo w trybie konsolowym albo w trybie graficznym. Warstwa prezentacyjna jest tutaj dodatkiem. Kod zarządzający warstwą graficzną jak i uruchomieniem *kernela* jest nieco inny niż w przypadku poprzednich przykładów. Wynika to z potrzeby uproszczenia całości (Listing 57). Dodanie warstwy graficznej zwiększa złożoność i tak już całkiem skomplikowanego zestawu klas i metod. Zanim przejdę do tychże detali implementacyjnych zaznaczę, że **przyspieszenie, o jakim możemy mówić wynosi od kilku do kilkunastu razy pomiędzy poszczególnymi urządzeniami obecnymi w maszynie**. Takie różnice nie były dotychczas widoczne przy okazji przykładów jednowymiarowych. Zacząłem się nawet przez chwilę zastanawiać, jakie konkretnie algorytmy warto zaprezentować tak aby korzyści te były jasno widoczne, a nie tylko pozostawały z sferze domniemywań.

```
public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Utils.log("***** picture                    *****");
                Utils.log("***** com.michalasobczak.picture  *****");
                CL.setExceptionsEnabled(true);
                PlatformParametersSet p = new PlatformParametersSet();
                RuntimeConfigurationSet r = new RuntimeConfigurationSet();
                KernelConfigurationSet c = new KernelConfigurationSet();
                c.initImages("picture/src/com/michalasobczak/picture/input2.png");
                c.initPanel();
                p.getPlatformsAndDevices();
                p.getDevicesInfo();
                r.selectPlatform(1);
                r.selectDevice(1);
                c.createContext();
                c.createCommandQueue();
                c.checkforImageSupport();
                c.readKernelFile();
                c.initImageMem();
                c.initializeKernel();
                Thread thread = new Thread(new Runnable() {
                    public void run() {
                        float x = 1.0f;
                        while (x < 100.0f) {
                            c.setKernelArgs(x);
                            c.runKernel(1);
                            c.outputLabel.repaint();
                            x = x + 0.01f;
                        }
                    }
                });
                thread.setDaemon(true);
                thread.start();
            }
        });
    } // main
} // class Main
```

Listing 57 — picture/Main. java

Nowymi metodami względem poprzednich przykładów są *initImages*, *initPanel*, *checkForImageSupport*, *initImageMem* oraz cała sekcja wątku z metodą *setKernelArgs*. Usunięto generowanie pseudolosowych danych wejściowych w postaci tablic. Zamiast tego mamy tworzenie obiektów obrazów. Jeśli nie chcemy oglądać wyników graficznych (Rys. 28), wówczas komentujemy metody *iniPanel* oraz *repaint*.



Rys. 28 — Prezentacja graficzna obrazu wejściowego i wyniku

Do wczytania pliku PNG z obrazem używamy klas *ImageIO*, *BufferedImage* oraz *Graphics*. Pobieramy rozmiary zdjęcia, długość i szerokość. Przygotowywane są obiekty (Listing 58) dla obrazu wejściowego i wyjściowego. Odczyt pliku wejściowego wymaga przechwytywania wyjątków.

```
public static BufferedImage createBufferedImage(String fileName) {
    BufferedImage image = null;
    try {
        image = ImageIO.read(new File(fileName));
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    int sizeX = image.getWidth();
    int sizeY = image.getHeight();
    BufferedImage result = new BufferedImage(sizeX, sizeY,
                           BufferedImage.TYPE_INT_RGB);
    Graphics g = result.createGraphics();
    g.drawImage(image, 0, 0, null);
    g.dispose();
    return result;
}
public void initImages(String path) {
    String fileName = path;
    inputImage = createBufferedImage(fileName);
    imageSizeX = inputImage.getWidth();
    imageSizeY = inputImage.getHeight();
    System.out.println("x=" + imageSizeX + ", y=" + imageSizeY);
    outputImage = new BufferedImage(imageSizeX, imageSizeY,
                                    BufferedImage.TYPE_INT_RGB);
}
```

Listing 58 — BufferedImage

Metoda *initPanel* przeznaczona jest (Listing 59) do utworzenia przestrzeni na obraz wejściowy (ten po lewej stronie) i obraz wyjściowy (ten po stronie prawej). Następnie oba te panele dodawane są do ramki tworzonej za pomocą klasy *JFrame*. Nie chciałbym się zbytnio skupiać na kwestiach związanych z interfejsem użytkownika, bo nie taki jest przedmiot niniejszej publikacji. Budowa tego interfejsu jest zatem ograniczona do niezbędnego minimum (Listing 59).

```
public void initPanel() {
    JPanel mainPanel = new JPanel(new GridLayout(1,0));
    JLabel inputLabel = new JLabel(new ImageIcon(inputImage));
    mainPanel.add(inputLabel, BorderLayout.CENTER);
    outputLabel = new JLabel(new ImageIcon(outputImage));
    mainPanel.add(outputLabel, BorderLayout.CENTER);
    JFrame frame = new JFrame("OpenCL picture test no 1");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(new BorderLayout());
    frame.add(mainPanel, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}
```

Listing 59 — initPanel

Jako że nie wszystkie urządzenia wspierają obsługę obrazów, należy wprost to sprawdzić. Służy do tego parametr

CL_DEVICE_IMAGE_SUPPORT

w wywołaniu metody *clGetDeviceInfo*. Pobieramy go również nieco później przy okazji szczegółów dotyczących obecnych w systemie urządzeń. Jeśli urządzenie nie wspiera obsługi obrazów, wówczas program zakończy działanie (Listing 60).

```
public void checkForImageSupport() {
    int imageSupport[] = new int[1];
    clGetDeviceInfo(RuntimeConfigurationSet.device, CL.CL_DEVICE_IMAGE_SUPPORT,
            Sizeof.cl_int, Pointer.to(imageSupport), null);
    System.out.println("Images supported: " + (imageSupport[0]==1));
    if (imageSupport[0] == 0) {
        System.out.println("Images are not supported");
        System.exit(1);
        return;
    }
}
```

Listing 60 — clGetDeviceInfo

Uprzednio załadowany obraz, teraz wczytywany jest do zmiennej typu *DataBufferInt*, do której wskaźnik klasy *Pointer* przekazywany jest do metody *clCreateImage2D* (Listing 61). Dane w formacie to liczby całkowite bez znaku, a kolejność składowych to RGBA. Obraz wyjściowy, a raczej przestrzeń w pamięci (klasy *cl_mem*), są tworzone bez wskazania na żaden wskaźnik.

```
public void initImageMem() {
    DataBufferInt dataBufferSrc = (DataBufferInt)inputImage.getRaster().getDataBuffer();
    int dataSrc[] = dataBufferSrc.getData();
    cl_image_format imageFormat = new cl_image_format();
    imageFormat.image_channel_order = CL_RGBA;
    imageFormat.image_channel_data_type = CL_UNSIGNED_INT8;
    inputImageMem = clCreateImage2D(
            context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
            new cl_image_format[]{imageFormat}, imageSizeX, imageSizeY,
            imageSizeX * Sizeof.cl_uint, Pointer.to(dataSrc), null);
    outputImageMem = clCreateImage2D(
            context, CL_MEM_WRITE_ONLY,
            new cl_image_format[]{imageFormat}, imageSizeX, imageSizeY,
            0, null, null);
}
```

Listing 61 — clCreateImage2D

Inicjalizacja *kernela* z podaniem rozmiaru globalnego jest realizowana nieco nich dotychczas. Przykład będąc dwuwymiarowym przyjmuje za wymiar pierwszy szerokość, a za wymiar drugi wysokość obrazu (Listing 62).

```
public void initializeKernel() {
    this.program = clCreateProgramWithSource(this.context, 1,
                    new String[]{ this.content }, null, null);
    clBuildProgram(this.program, 0, null, null, null, null);
    this.kernel = clCreateKernel(this.program, "sampleKernel", null);
    globalWorkSize = new long[2];
    globalWorkSize[0] = imageSizeX;
    globalWorkSize[1] = imageSizeY;
}
```

Listing 62 — Rozmiar globalny

Ustawianie argumentów *kernela* przeniesione zostało do osobnej metody (Listing 63). Pierwszy i drugi parametr do odpowiednio obraz wejściowy i obraz wyjściowy. Trzeci parametr służy do podawania liczby parametryzującej *kernel*.

```
public void setKernelArgs(float factor) {
    clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inputImageMem));
    clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outputImageMem));
    clSetKernelArg(kernel, 2, Sizeof.cl_float, Pointer.to(new float[] { factor } ));
}
```

Listing 63 — Argumenty kernela

Uruchomienie *kernela* jest standardowe, aczkolwiek wskazujemy, że przekazujemy do niego dane dwuwymiarowe. Drugą różnicą względem poprzednich przykładów jest brak wskazania rozmiaru lokalnego. Podajemy tutaj *null* licząc na optymalny wybór mechanizmu uruchomieniowego.

```
clEnqueueNDRangeKernel(this.commandQueue, this.kernel, 2, null,
this.globalWorkSize, null, 0, null, kernelEvent0);
```

Odczyt wyniku działania *kernela* to wywołanie (Listing 64) metody *clEnqueReadImage*. Używa ona przestrzeni pamięci

identyfikowanej przez zmienną *outputImageMem,* a dane zapisywane są do obiektu *dataDst* za pośrednictwem wskaźnika.

```
DataBufferInt dataBufferDst = (DataBufferInt)outputImage.getRaster().getDataBuffer();
int dataDst[] = dataBufferDst.getData();
clEnqueueReadImage(this.commandQueue, outputImageMem, true, new long[3],
        new long[]{imageSizeX, imageSizeY, 1), imageSizeX * Sizeof.cl_uint,
        0, Pointer.to(dataDst), 0, null, readEvent0);
```

Listing 64 — DataBufferInt

Sam plik *kernela* jest całkiem krótki (Listing 65). Na początku ustawiamy parametry *samplera*, który tutaj znaleźć się musi, aczkolwiek w tym konkretnym przykładzie nie będzie miał szczególnego zastosowania. We właściwej metodzie pobieramy identyfikatory poszczególnych wymiarów jak również rozmiary obrazu wejściowego. Wektory dwuelementowe budują piksel wejściowy oraz piksel wyjściowy. Odczyt i zapis to odpowiednio funkcje *read_imageui* oraz *write_imageui*.

```
const sampler_t samplerIn = CLK_NORMALIZED_COORDS_FALSE
                          | CLK_ADDRESS_CLAMP
                          | CLK_FILTER_NEAREST;
__kernel void sampleKernel(__read_only  image2d_t sourceImage,
  __write_only image2d_t targetImage,
  float factor) {
    int gidX = get_global_id(0);
    int gidY = get_global_id(1);
    int w = get_image_width(sourceImage);
    int h = get_image_height(sourceImage);
    int2 posIn  = {gidX, gidY};
    int2 posOut = {gidX/factor, gidY/factor};
    uint4 pixel = read_imageui(sourceImage, samplerIn, posIn);
    write_imageui(targetImage, posOut, pixel);
}
```

Listing 65 — picture/kernel. c

Proszę spojrzeć na sposób wywołania skalowania w pliku *Main. java*. Jest to pętla, w której wywoływany jest *kernel* za każdym razem z innym parametrem w postaci zmiennej *factor*. Odpowiada ona za współczynnik skalowania. **W efekcie uzyskujemy interesujący wzór składający się z coraz to mniejszych przylegających do siebie obrazów.**

Uwaga: porównawcze wyniki wydajności tego przykładu znajdują się w kolejnym podrozdziale.

7.2. Wnioski dotyczące wydajności

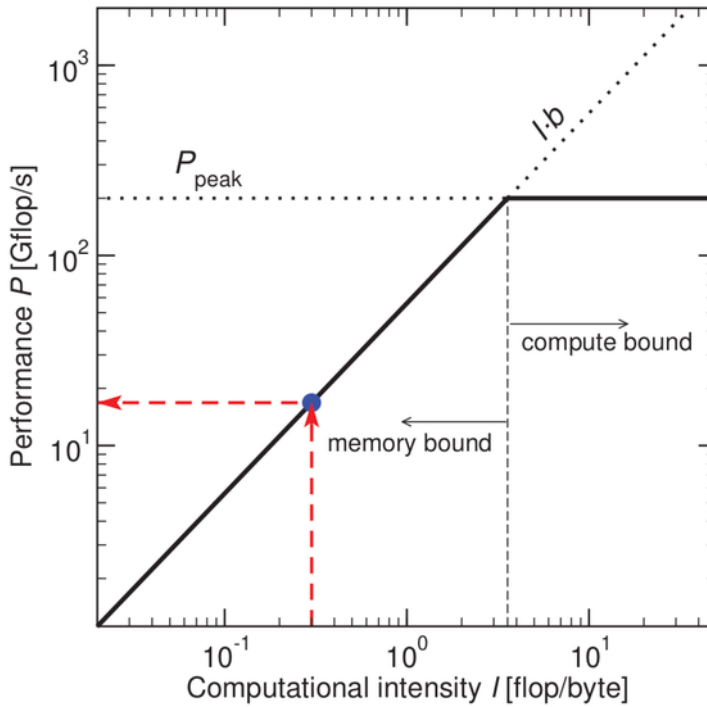
Dla przykładu wielowymiarowego dotyczącego skalowania uzyskane zostały **satysfakcjonujące rezultaty**. **Porównujemy wydajność działania CPU i GPU.**

Na zestawie Dell G15 z procesorem Intel Core i5 10200H czas obliczeń to 0.5ms, czas odczytów to 0.17ms. Na karcie Intel UHD Graphics czas obliczeń to 0.19ms, czas odczytów to 0.17ms. Na karcie NVIDIA GeForce RTX 3050 Ti czas obliczeń to 0.01ms, czas odczytów to 0.23ms. **Pomiędzy CPU, a dedykowanym GPU mamy zatem 50 razy szybsze przetwarzanie algorytmu**. Co ciekawe, czas odczytu jest o około 30% wolniejszy. Potwierdza to zatem wcześniejsze wnioski o tym, że konfiguracja dostępów do pamięci ma tutaj krytyczne znaczenie.

Na drugim badanym zestawie, tj. Lenovo Thinkpad T420s mamy porównanie pomiędzy procesorem Intel Core i7—2640M, a dedykowaną kartą NVIDIA NVS 4200M. Czas obliczeń na procesorze wynosi 1.2ms, a czas odczytu 0.23ms. Czasy te mają większe odchylenia niż w pierwszym badanym zestawie. Czas obliczeń na dedykowanej karcie to 0.5ms, a czas odczytu 0.23ms. **Przyspieszenie GPU względem CPU to zatem około 2 razy**. Czas odczytu podobnie jak w pierwszym zestawie jest zbliżony.

Różnica w przyspieszeniach wynika z różnicy pomiędzy CPU i GPU, gdzie w pierwszym zestawie ilość jednostek cieniujących (tj. 2560) znacząco przewyższa ilość wątków procesora (tj. 8). Drugi zestaw nie ma takiej dużej różnicy w złożoności sprzętu. Procesor posiada 4 wątki, a karta 48 jednostek cieniujących. Warte jednak wspomnienia jest fakt, że na całkowitą wydajność ma wpływ znacznie więcej czyn-

ników niż tylko ilość komponentów. **Karta NVIDIA NVS 4200M posiada zaledwie jeden multiprocesor strumieniowy podczas gdy karta NVIDIA GeForce RTX 3050 Ti posiada ich aż 20**. Stąd też tak duża różnica w przypadku tamtego zestawu, gdzie uzyskane przyspieszenie to aż 50 razy.



Rys. 29 — Przykładowy model Roofline [W]

Aby poznać teoretyczne możliwości zwiększania wydajności algorytmów (Rys. 29) dzięki stosowaniu standardu OpenCL warto skierować uwagę na przykład na **model Roofline, którego parametrami wejściowymi są ograniczenia sprzęto-**

we, takie jak maksymalna moc obliczeniowa oraz maksymalna przepustowość pamięci.

8. SPRZĘT

W przykładach w książce wykorzystanych zostało wiele różnych urządzeń obliczeniowych, zarówno CPU jak i GPU. W tej części zaprezentuję czym jest ten sprzęt. Specyfikacje dla kart graficznych pochodzą z serwisu techpowerup.com [230], natomiast dla procesorów, z serwisu technical.city [231]. Dla każdego urządzenia dodałem komentarz wskazujący na co warto zwrócić uwagę. Większość z tych urządzeń będzie stosowana przy okazji jednej z kolejnych części serii, tj. programowania grafiki 3D z wykorzystaniem między innymi standardu OpenGL.

8.1. (2010) CPU Intel Xeon X5660

Procesor zainstalowany w stacji HP z800 w ilości 2 sztuk. Razem daje to 24 wątki przetwarzania. Wprowadzony na rynek w 2010 roku wcale nie ustępuje znacząco nowym konstrukcjom jak na przykład Intel Core i5—10200H. W zastosowaniach przy wykorzystaniu wszystkich wątków może być nawet wydajniejszy.

Rdzeni	6
Strumieni	12
Częstotliwość podstawowa	2.8 GHz
Maksymalna częstotliwość	3.2 GHz
Pamięć podręczna 1-go poziomu	64 KB (na rdzeń)
Pamięć podręczna 2-go poziomu	256 KB (na rdzeń)
Pamięć podręczna 3-go poziomu	12 MB (łącznie)
Proces technologiczny	32 nm
Rozmiar kryształu	239 mm^2
Maksymalna temperatura rdzenia	81 °C
Ilość tranzystorów	1,170 million
Obsługa 64 bitów	+
Zgodność z Windows 11	-
Odblokowany mnożnik	-
Dopuszczalne napięcie rdzenia	0.75V-1.35V

Specyfikacja procesora Intel Xeon x5660

8.2. (2009) CPU Intel Xeon X5570

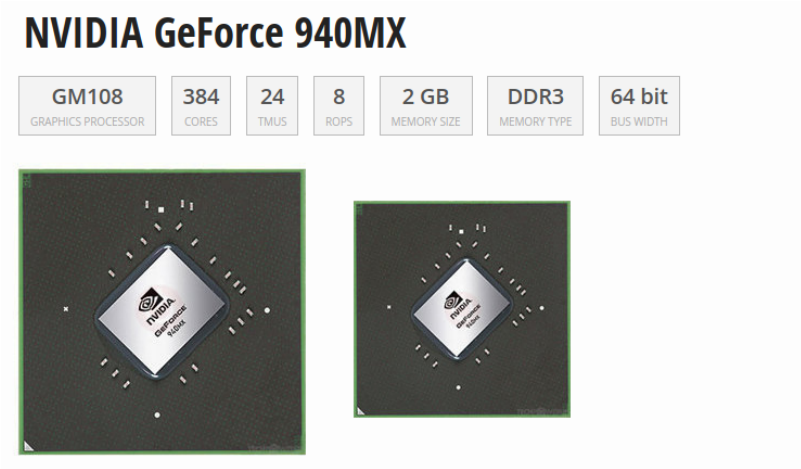
Ten procesor wprowadzony na rynek w 2009 roku przez mniejszą liczbę tranzystorów i wątków raczej nadaję się do mniej intensywnych obliczeń. Idealnie sprawdzi się jako komponent serwera o niskim poborze mocy zachowując przy tym satysfakcjonujący poziom wydajności. Procesor w 2021 roku kosztował mnie 9 zł.

Rdzeni	4
Strumieni	8
Częstotliwość podstawowa	2.93 GHz
Maksymalna częstotliwość	3.33 GHz
Pamięć podręczna 1-go poziomu	64 KB (na rdzeń)
Pamięć podręczna 2-go poziomu	256 KB (na rdzeń)
Pamięć podręczna 3-go poziomu	8 MB (łącznie)
Proces technologiczny	45 nm
Rozmiar kryształu	263 mm^2
Maksymalna temperatura rdzenia	75 °C
Ilość tranzystorów	731 million
Obsługa 64 bitów	+
Zgodność z Windows 11	-
Odblokowany mnożnik	-
Dopuszczalne napięcie rdzenia	0.75V -1.35V

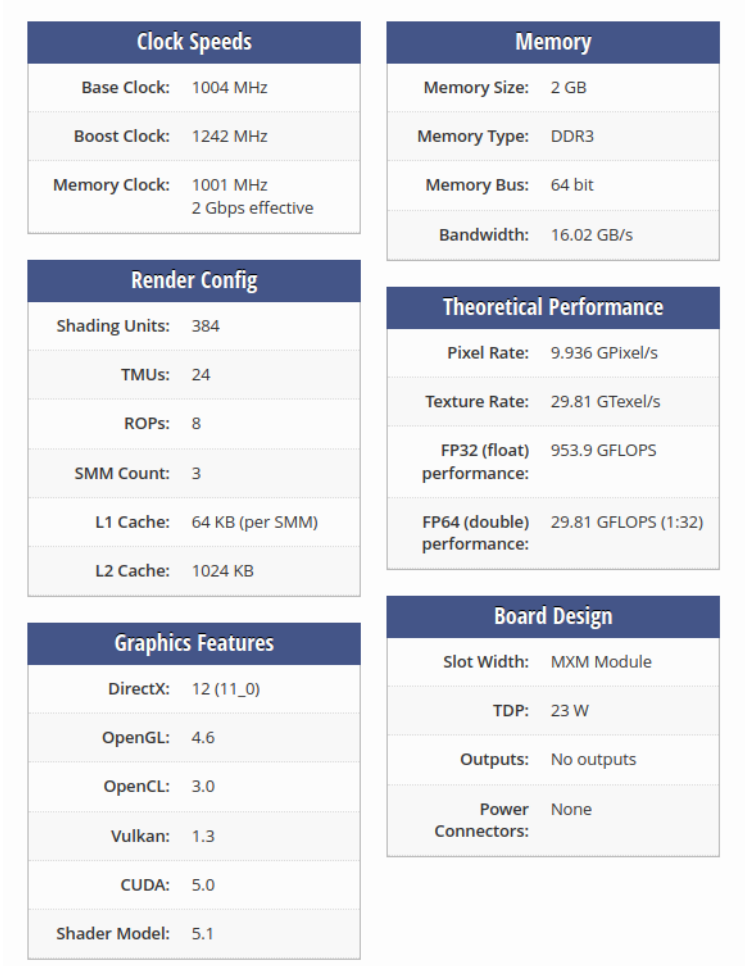
Specyfikacja procesora Intel Xeon x5570

8.3. (2015) GPU NVIDIA GeForce 940MX

Karta ta wbudowana w komputer Lenovo Thinkpad E470 ma bardzo dobre parametry pod kątem wydajności, ale problematyczne jest odprowadzanie ciepła. W systemie Ubuntu Linux, gdzie zarządzanie energią (w systemie i sterowniku) nie jest optymalnie oprogramowane możemy spodziewać się wysokich temperatur, które zdecydowanie obniżą komfort użytkowania.



Chip karty NVIDIA GeForce 940MX



Clock Speeds	
Base Clock:	1004 MHz
Boost Clock:	1242 MHz
Memory Clock:	1001 MHz 2 Gbps effective

Memory	
Memory Size:	2 GB
Memory Type:	DDR3
Memory Bus:	64 bit
Bandwidth:	16.02 GB/s

Render Config	
Shading Units:	384
TMUs:	24
ROPs:	8
SMM Count:	3
L1 Cache:	64 KB (per SMM)
L2 Cache:	1024 KB

Theoretical Performance	
Pixel Rate:	9.936 GPixel/s
Texture Rate:	29.81 GTexel/s
FP32 (float) performance:	953.9 GFLOPS
FP64 (double) performance:	29.81 GFLOPS (1:32)

Graphics Features	
DirectX:	12 (11_0)
OpenGL:	4.6
OpenCL:	3.0
Vulkan:	1.3
CUDA:	5.0
Shader Model:	5.1

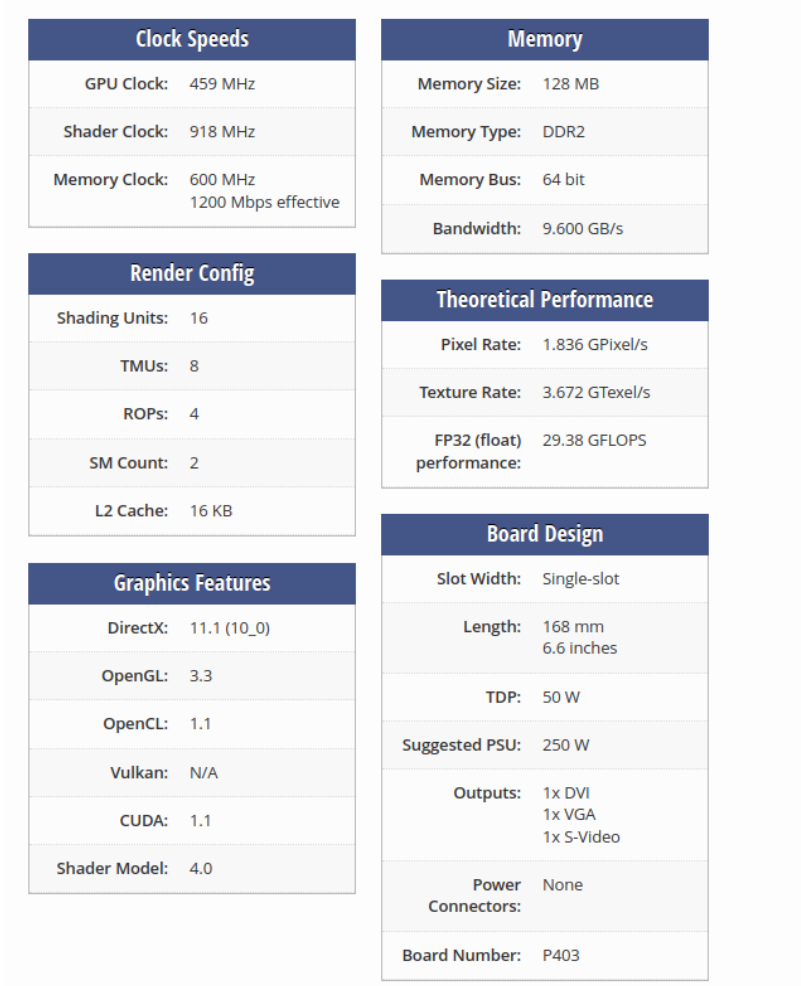
Board Design	
Slot Width:	MXM Module
TDP:	23 W
Outputs:	No outputs
Power Connectors:	None

Specyfikacja karty NVIDIA GeForce 940MX

8.4. (2008) GPU NVIDIA GeForce 9400 GT 512MB

Karta wprowadzona na rynek w 2008 roku. Jest to zatem jedna z pierwszych z obsługą standardu OpenCL. Oferuje podstawową wydajność, która wystarczy jedynie do mniej wymagających zadań. Karta referencyjna posiadała zaledwie 128MB pamięci. Uzyskuje wydajność obliczeniową FP32 na poziomie niecałych 30 GFLOPS.

Wygląd karty NVIDIA GeForce 9400 GT



Clock Speeds	
GPU Clock:	459 MHz
Shader Clock:	918 MHz
Memory Clock:	600 MHz 1200 Mbps effective

Render Config	
Shading Units:	16
TMUs:	8
ROPs:	4
SM Count:	2
L2 Cache:	16 KB

Graphics Features	
DirectX:	11.1 (10_0)
OpenGL:	3.3
OpenCL:	1.1
Vulkan:	N/A
CUDA:	1.1
Shader Model:	4.0

Memory	
Memory Size:	128 MB
Memory Type:	DDR2
Memory Bus:	64 bit
Bandwidth:	9.600 GB/s

Theoretical Performance	
Pixel Rate:	1.836 GPixel/s
Texture Rate:	3.672 GTexel/s
FP32 (float) performance:	29.38 GFLOPS

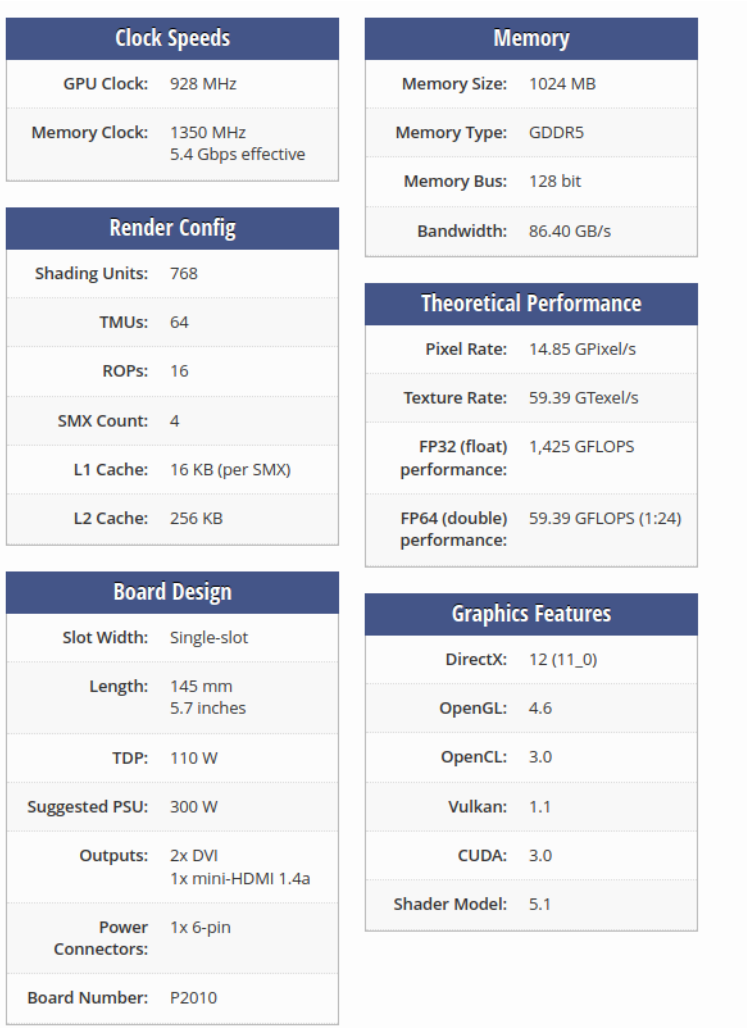
Board Design	
Slot Width:	Single-slot
Length:	168 mm 6.6 inches
TDP:	50 W
Suggested PSU:	250 W
Outputs:	1x DVI 1x VGA 1x S-Video
Power Connectors:	None
Board Number:	P403

Specyfikacja karty NVIDIA GeForce 9400 GT

8.5. (2012) GPU NVIDIA Zotac GTX 650Ti

Karta wprowadzona na rynek w 2012 roku, zatem jest to starsza już konstrukcja, jednak o zadziwiająco dobrych parametrach. Posiada aż 768 jednostek cieniujących i wsparcie dla OpenCL 3.0. Poza zastosowaniami obliczeniowym, nawet w 2022 roku można na niej zagrać w niektóre nowe tytuły. Minusem karty marki Zotac jest sposób umiejscowienia mocowania złącza zasilania, które w stacji HP z800 powoduje brak możliwości domknięcia wewnętrznej pokrywy akustycznej.

Wygląd karty NVIDIA GeForce GTX 650 Ti



Clock Speeds	
GPU Clock:	928 MHz
Memory Clock:	1350 MHz 5.4 Gbps effective

Render Config	
Shading Units:	768
TMUs:	64
ROPs:	16
SMX Count:	4
L1 Cache:	16 KB (per SMX)
L2 Cache:	256 KB

Board Design	
Slot Width:	Single-slot
Length:	145 mm 5.7 inches
TDP:	110 W
Suggested PSU:	300 W
Outputs:	2x DVI 1x mini-HDMI 1.4a
Power Connectors:	1x 6-pin
Board Number:	P2010

Memory	
Memory Size:	1024 MB
Memory Type:	GDDR5
Memory Bus:	128 bit
Bandwidth:	86.40 GB/s

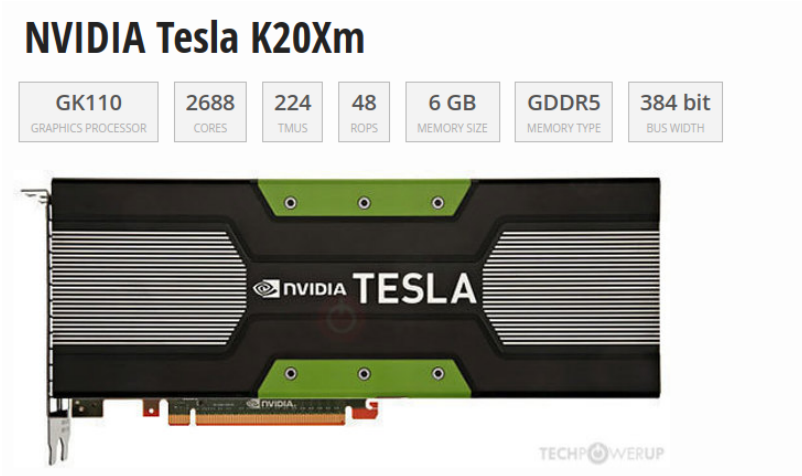
Theoretical Performance	
Pixel Rate:	14.85 GPixel/s
Texture Rate:	59.39 GTexel/s
FP32 (float) performance:	1,425 GFLOPS
FP64 (double) performance:	59.39 GFLOPS (1:24)

Graphics Features	
DirectX:	12 (11_0)
OpenGL:	4.6
OpenCL:	3.0
Vulkan:	1.1
CUDA:	3.0
Shader Model:	5.1

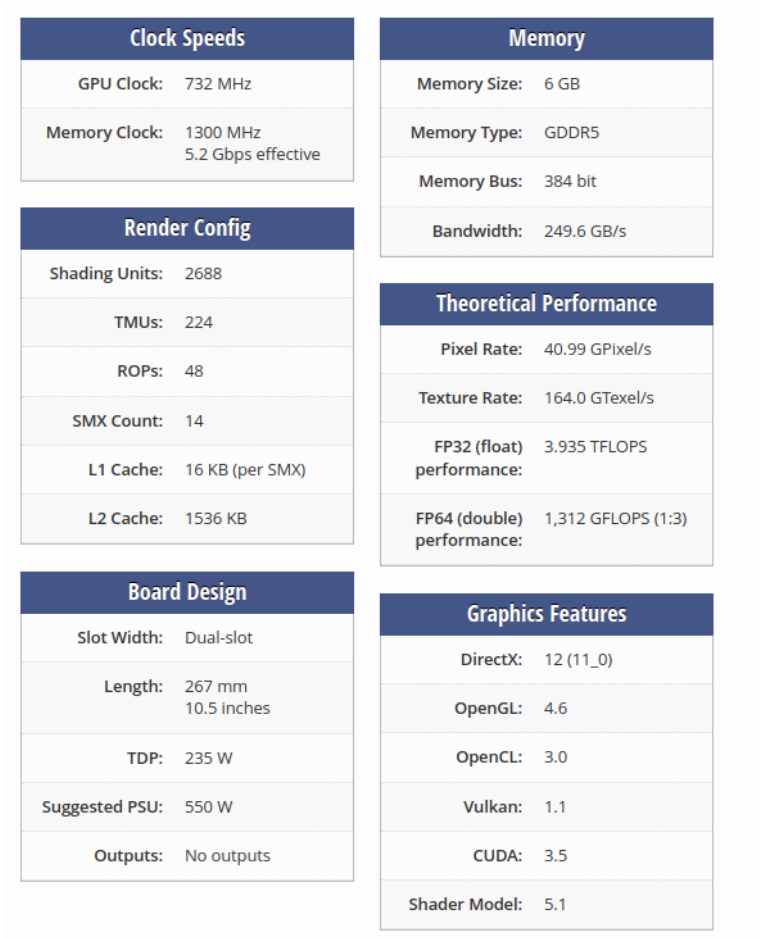
Specyfikacja karty NVIDIA GeForce GTX 650 Ti

8.6. (2012) GPU NVIDIA Tesla K20Xm

Karta klasy profesjonalnej, bez wyjść wideo. Przeznaczona jest wyłącznie do zadań obliczeniowych lub jako wirtualny adapter w serwerach wirtualizacyjnych. Posiada aż 2688 jednostek cieniujących oraz 6GB pamięci. Porównawczo karta ta ma wydajność karty NVIDIA GeForce GTX 1650 z 2019 roku, jednakże będziemy mogli to odczuć jedynie w zakresie zadań obliczeniowych. Bazowa cena tej karty to 7699 USD. W 2021 roku kupiłem ją za nieco ponad 200 zł. Uzyskuje znakomitą wydajność na poziomie 3935 GFLOPS w teście FP32.



Wygląd karty NVIDIA Tesla K20Xm



Clock Speeds	
GPU Clock:	732 MHz
Memory Clock:	1300 MHz 5.2 Gbps effective

Render Config	
Shading Units:	2688
TMUs:	224
ROPs:	48
SMX Count:	14
L1 Cache:	16 KB (per SMX)
L2 Cache:	1536 KB

Board Design	
Slot Width:	Dual-slot
Length:	267 mm 10.5 inches
TDP:	235 W
Suggested PSU:	550 W
Outputs:	No outputs

Memory	
Memory Size:	6 GB
Memory Type:	GDDR5
Memory Bus:	384 bit
Bandwidth:	249.6 GB/s

Theoretical Performance	
Pixel Rate:	40.99 GPixel/s
Texture Rate:	164.0 GTexel/s
FP32 (float) performance:	3.935 TFLOPS
FP64 (double) performance:	1,312 GFLOPS (1:3)

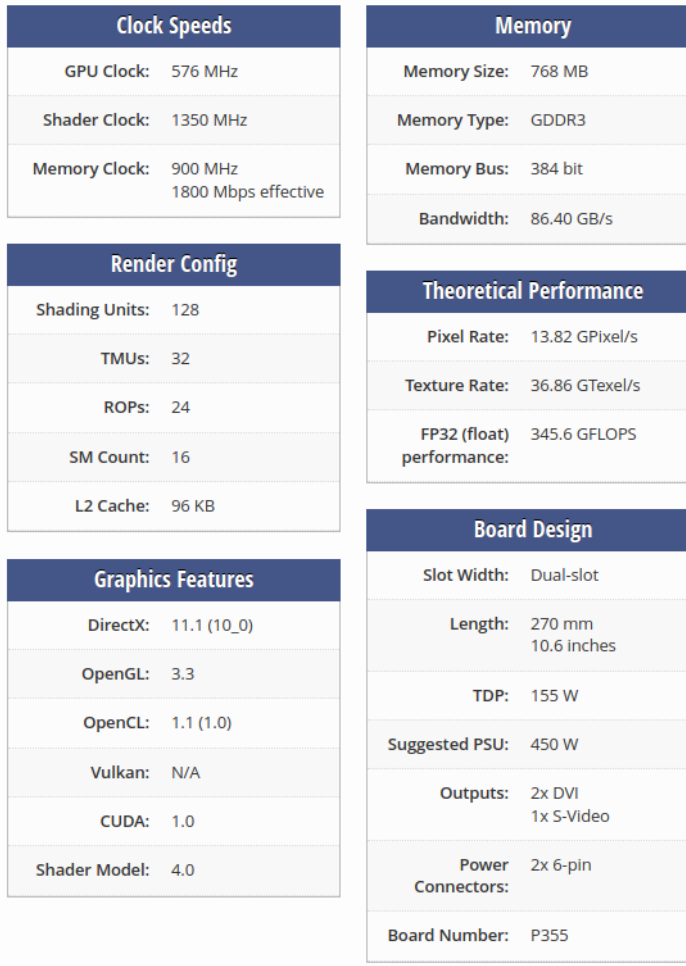
Graphics Features	
DirectX:	12 (11_0)
OpenGL:	4.6
OpenCL:	3.0
Vulkan:	1.1
CUDA:	3.5
Shader Model:	5.1

Specyfikacja karty NVIDIA Telsa K20Xm

8.7. (2006) GPU NVIDIA Asus 8800 GTX 768MB

Karta ta to jedna z pierwszych jeśli faktycznie nie pierwsza obsługująca standard OpenCL 1.0. Posiada 128 jednostek cieniujących. W momencie wprowadzenia na rynek w 2006 roku kosztowała 599 USD. W 2021 roku kupiłem ją za 170 zł. Jest teoretycznie 10 razy mniej wydajna obliczeniowo niż NVIDIA Tesla K20Xm. Z drugiej jednak strony jest około 10 razy bardziej wydajna niż NVIDIA GeForce 9400 GT.

Wygląd karty NVIDIA GeForce 8800 GTX



Clock Speeds	
GPU Clock:	576 MHz
Shader Clock:	1350 MHz
Memory Clock:	900 MHz 1800 Mbps effective

Memory	
Memory Size:	768 MB
Memory Type:	GDDR3
Memory Bus:	384 bit
Bandwidth:	86.40 GB/s

Render Config	
Shading Units:	128
TMUs:	32
ROPs:	24
SM Count:	16
L2 Cache:	96 KB

Theoretical Performance	
Pixel Rate:	13.82 GPixel/s
Texture Rate:	36.86 GTexel/s
FP32 (float) performance:	345.6 GFLOPS

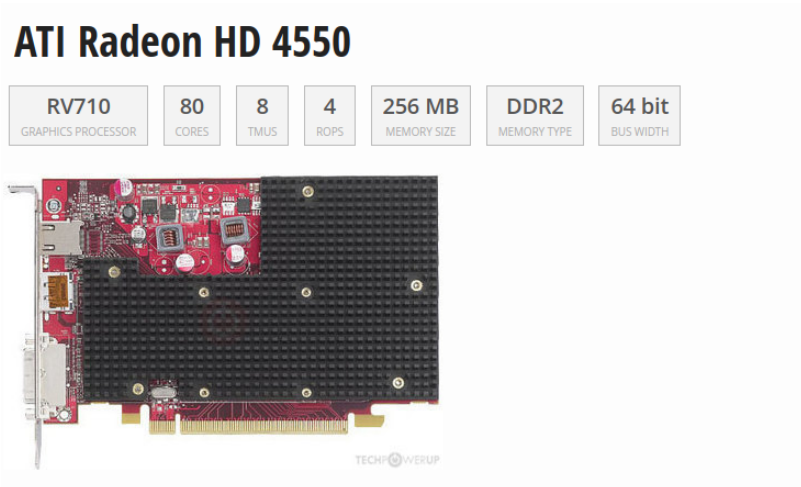
Graphics Features	
DirectX:	11.1 (10_0)
OpenGL:	3.3
OpenCL:	1.1 (1.0)
Vulkan:	N/A
CUDA:	1.0
Shader Model:	4.0

Board Design	
Slot Width:	Dual-slot
Length:	270 mm 10.6 inches
TDP:	155 W
Suggested PSU:	450 W
Outputs:	2x DVI 1x S-Video
Power Connectors:	2x 6-pin
Board Number:	P355

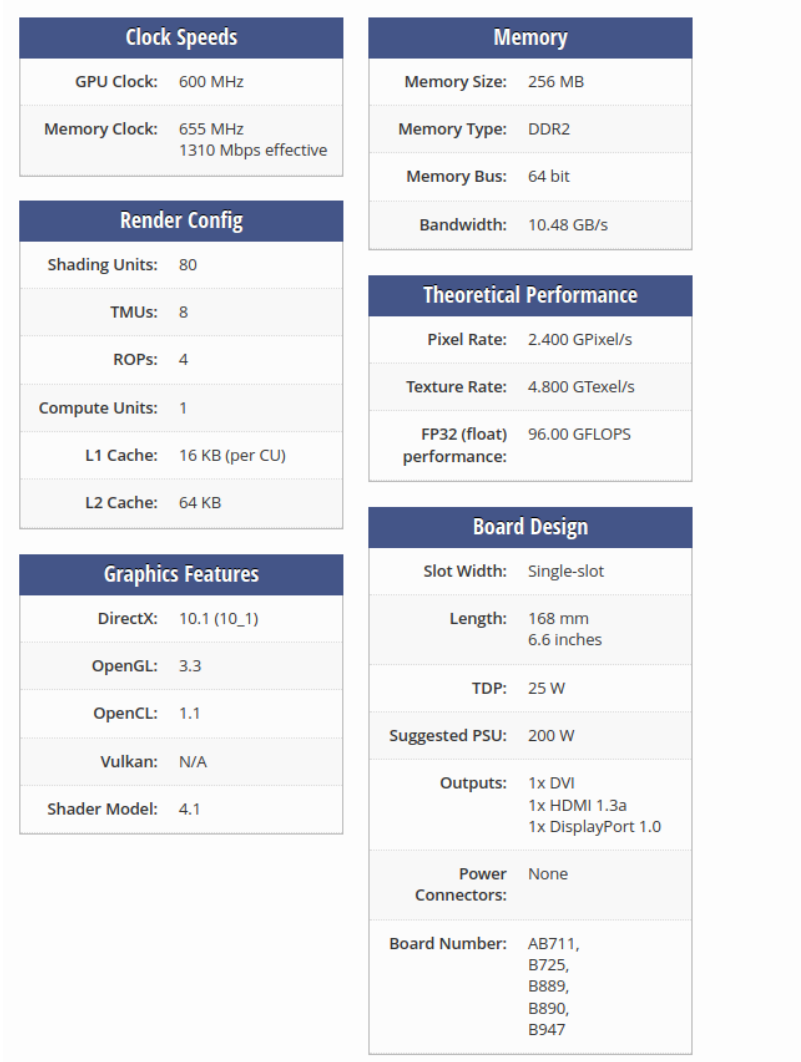
Specyfikacja karty NVIDIA GeForce 8800 GTX

8.8. (2008) GPU AMD Radeon HD4550 512MB

Karta wprowadzona na rynek w 2008 roku. Jest to urządzenie o bardzo podstawowych parametrach. Posiada 80 jednostek cieniujących, niewielką ilość pamięci i wydajność FP32 na poziomie 96 GFLOPS. Idealnie sprawdzi się w zestawach o niższym poborze mocy.



Wygląd karty ATI Radeon HD 4550



Clock Speeds	
GPU Clock:	600 MHz
Memory Clock:	655 MHz 1310 Mbps effective

Render Config	
Shading Units:	80
TMUs:	8
ROPs:	4
Compute Units:	1
L1 Cache:	16 KB (per CU)
L2 Cache:	64 KB

Graphics Features	
DirectX:	10.1 (10_1)
OpenGL:	3.3
OpenCL:	1.1
Vulkan:	N/A
Shader Model:	4.1

Memory	
Memory Size:	256 MB
Memory Type:	DDR2
Memory Bus:	64 bit
Bandwidth:	10.48 GB/s

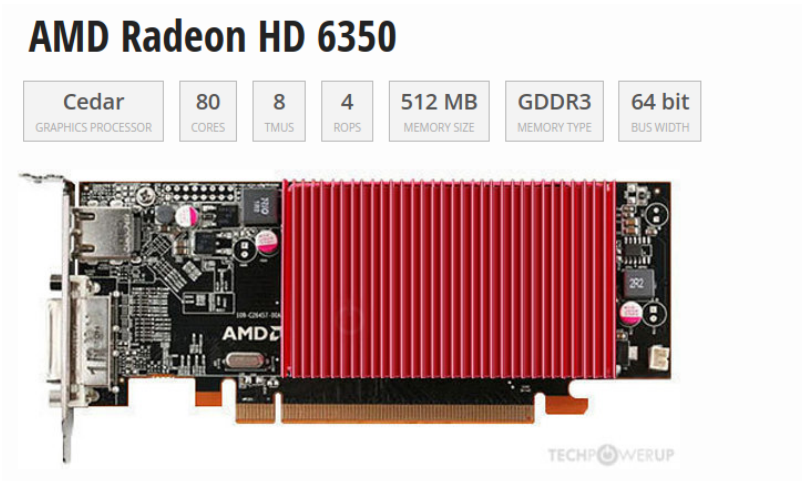
Theoretical Performance	
Pixel Rate:	2.400 GPixel/s
Texture Rate:	4.800 GTexel/s
FP32 (float) performance:	96.00 GFLOPS

Board Design	
Slot Width:	Single-slot
Length:	168 mm 6.6 inches
TDP:	25 W
Suggested PSU:	200 W
Outputs:	1x DVI 1x HDMI 1.3a 1x DisplayPort 1.0
Power Connectors:	None
Board Number:	AB711, B725, B889, B890, B947

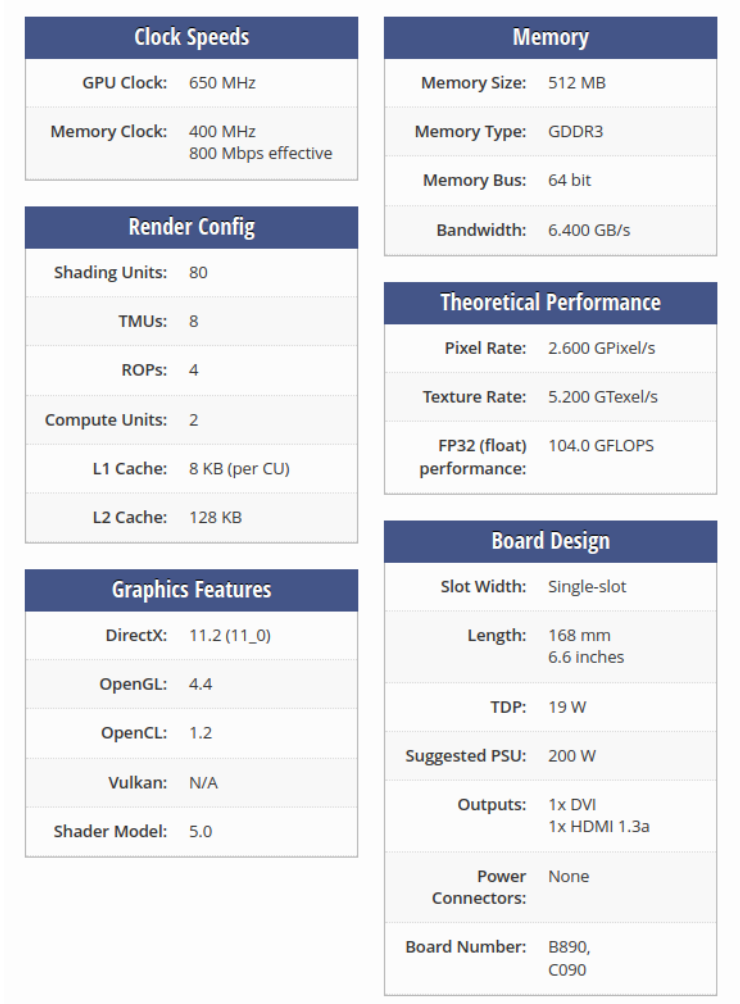
Specyfikacja karty ATI Radeon HD 4550

8.9. (2011) GPU AMD Radeon HD6350 512MB

Karta wprowadzona na rynek w 2011 roku. Testowana była pod kątem współdziałania 2 kart jednocześnie. Tylko taki test był przedmiotem zainteresowania. Karta nie zachwyca parametrami. Jej początkowa cena to zaledwie 23 USD. W 2021 zapłaciłem za nią 19 zł za sztukę. Pobór mocy to zaledwie 19W, sprawdzi się zatem w rozwiązaniach energooszczędnych.



Wygląd karty AMD Radeon HD 6350



Clock Speeds	
GPU Clock:	650 MHz
Memory Clock:	400 MHz 800 Mbps effective

Render Config	
Shading Units:	80
TMUs:	8
ROPs:	4
Compute Units:	2
L1 Cache:	8 KB (per CU)
L2 Cache:	128 KB

Graphics Features	
DirectX:	11.2 (11_0)
OpenGL:	4.4
OpenCL:	1.2
Vulkan:	N/A
Shader Model:	5.0

Memory	
Memory Size:	512 MB
Memory Type:	GDDR3
Memory Bus:	64 bit
Bandwidth:	6.400 GB/s

Theoretical Performance	
Pixel Rate:	2.600 GPixel/s
Texture Rate:	5.200 GTexel/s
FP32 (float) performance:	104.0 GFLOPS

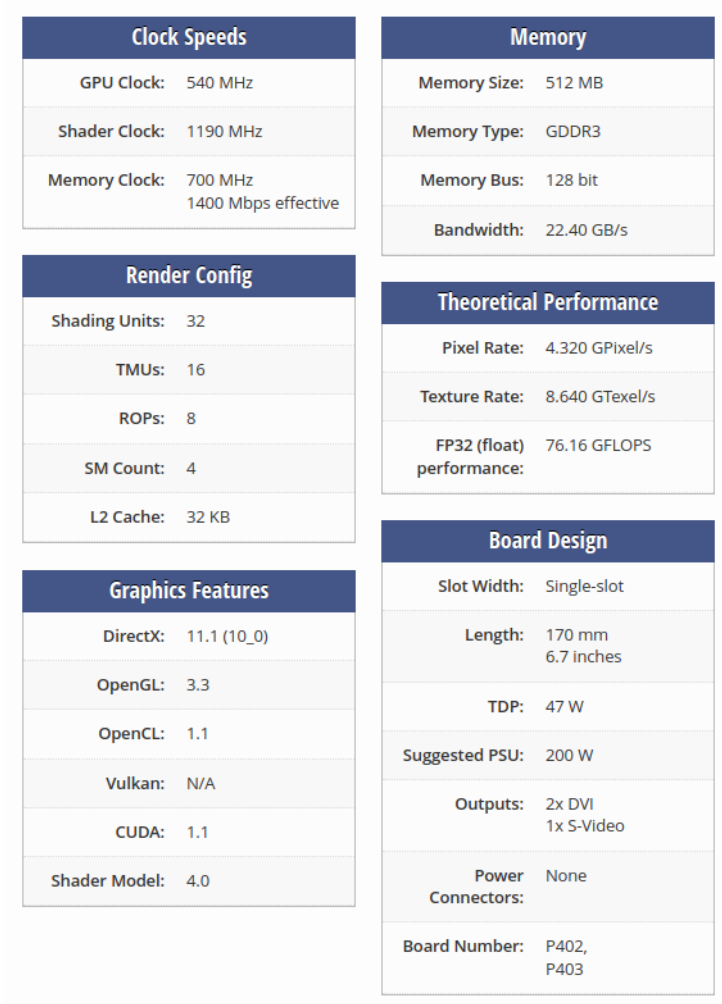
Board Design	
Slot Width:	Single-slot
Length:	168 mm 6.6 inches
TDP:	19 W
Suggested PSU:	200 W
Outputs:	1x DVI 1x HDMI 1.3a
Power Connectors:	None
Board Number:	B890, C090

Specyfikacja karty AMD Radeon HD 6350

8.10. (2007) GPU NVIDIA GeForce 8600 GT 1GB

Karta ta jako jedna z pierwszych wspierała standard OpenCL. Wprowadzona na rynek w 2007 roku. Jej początkowa cena wynosiła 159 USD. W 2021 roku zakupiłem ją za 69 zł za sztukę z oryginalnym opakowaniem. Posiada 32 jednostki cieniujące. Wydajność FP32 wynosi 76 GFLOPS.

Wygląd karty NVIDIA GeForce 8600 GT



Clock Speeds	
GPU Clock:	540 MHz
Shader Clock:	1190 MHz
Memory Clock:	700 MHz 1400 Mbps effective

Render Config	
Shading Units:	32
TMUs:	16
ROPs:	8
SM Count:	4
L2 Cache:	32 KB

Graphics Features	
DirectX:	11.1 (10_0)
OpenGL:	3.3
OpenCL:	1.1
Vulkan:	N/A
CUDA:	1.1
Shader Model:	4.0

Memory	
Memory Size:	512 MB
Memory Type:	GDDR3
Memory Bus:	128 bit
Bandwidth:	22.40 GB/s

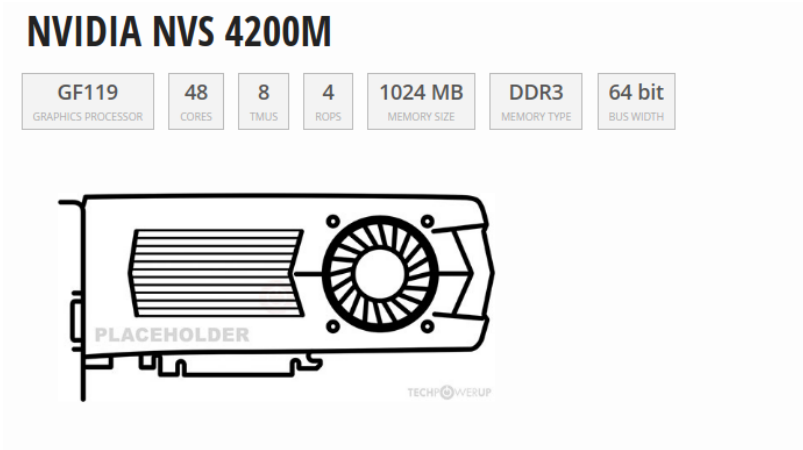
Theoretical Performance	
Pixel Rate:	4.320 GPixel/s
Texture Rate:	8.640 GTexel/s
FP32 (float) performance:	76.16 GFLOPS

Board Design	
Slot Width:	Single-slot
Length:	170 mm 6.7 inches
TDP:	47 W
Suggested PSU:	200 W
Outputs:	2x DVI 1x S-Video
Power Connectors:	None
Board Number:	P402, P403

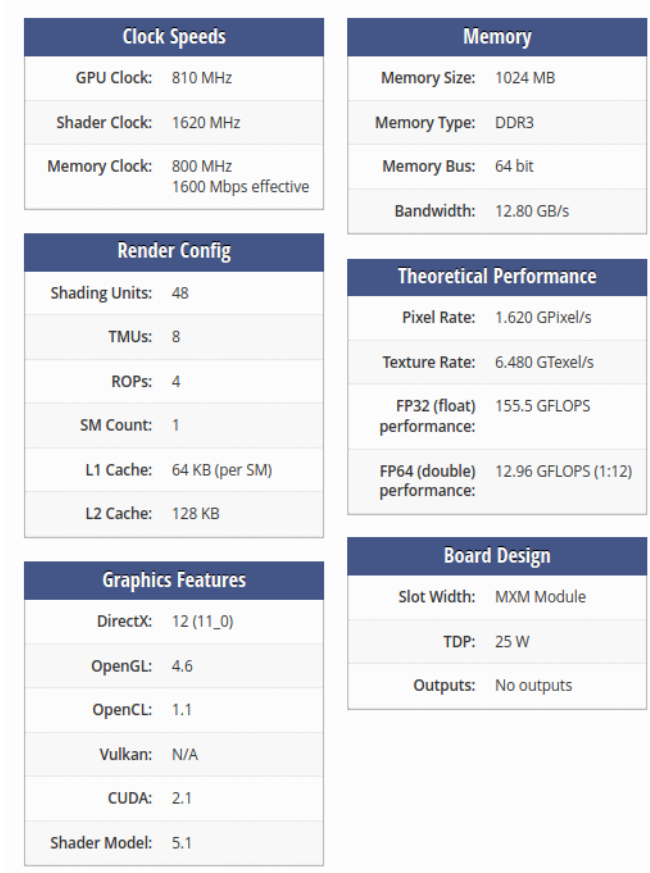
Specyfikacja karty NVIDIA GeForce 8600 GT

8.11. (2011) GPU NVIDIA Quadro 4200M 1GB

Karta ta jest wbudowana w komputer Lenovo Thinkpad T420s. Posiada 48 jednostek cieniujących. Jej ograniczona wydajność wynika z niskiego poboru mocy na poziomie 25W. Pomimo tego uzyskuje przyzwoite 155 GFLOPS w teście FP32. Jest to dwukrotnie więcej niż NVIDIA GeForce 8600 GT. Obecność tej karty w teście wynikała z konieczności posiada zestawu z zarówno procesorem jak i kartą graficzną wspierającą standard OpenCL.



Podsumowanie dla karty NVIDIA NVS 4200M



Clock Speeds	
GPU Clock:	810 MHz
Shader Clock:	1620 MHz
Memory Clock:	800 MHz 1600 Mbps effective

Render Config	
Shading Units:	48
TMUs:	8
ROPs:	4
SM Count:	1
L1 Cache:	64 KB (per SM)
L2 Cache:	128 KB

Graphics Features	
DirectX:	12 (11_0)
OpenGL:	4.6
OpenCL:	1.1
Vulkan:	N/A
CUDA:	2.1
Shader Model:	5.1

Memory	
Memory Size:	1024 MB
Memory Type:	DDR3
Memory Bus:	64 bit
Bandwidth:	12.80 GB/s

Theoretical Performance	
Pixel Rate:	1.620 GPixel/s
Texture Rate:	6.480 GTexel/s
FP32 (float) performance:	155.5 GFLOPS
FP64 (double) performance:	12.96 GFLOPS (1:12)

Board Design	
Slot Width:	MXM Module
TDP:	25 W
Outputs:	No outputs

Specyfikacja karty NVIDIA NVS 4200M

8.12. (2011) CPU Intel Core i7 —2640M

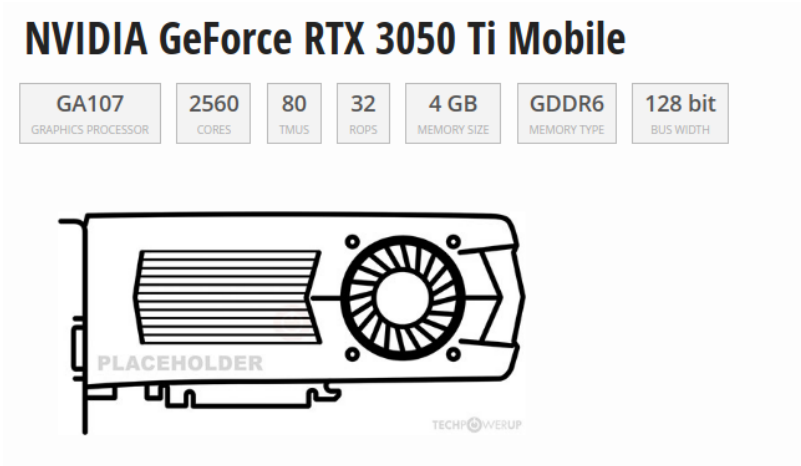
Procesor zainstalowany w komputerze Lenovo Thinkpad T420s. Oferuje przyzwoitą wydajność w 4 wątkach. Zdecydowanym minusem jest wysoka maksymalna temperatura pracy wynosząca aż 100 stopni. Powoduje to, że przy większym obciążeniu rzeczywiście otrzymamy temperatury na poziomie powyżej 90 stopni co jest mało komfortowe. Dzieje się tak również przy wyczyszczeniu wentylatorów i wymianie pasty termoprzewodzącej. Ratunkiem jest stosowanie kontrolera dla pracy wentylatorów, który może nieco poprawić sytuację.

Rdzeni	2
Strumieni	4
Częstotliwość podstawowa	2.8 GHz
Maksymalna częstotliwość	3.5 GHz
Pamięć podręczna 1-go poziomu	128 KB
Pamięć podręczna 2-go poziomu	512 KB
Pamięć podręczna 3-go poziomu	4 MB
Proces technologiczny	32 nm
Rozmiar kryształu	149 mm^2
Maksymalna temperatura rdzenia	100 °C
Ilość tranzystorów	624 Million
Obsługa 64 bitów	+
Zgodność z Windows 11	-
Odblokowany mnożnik	-

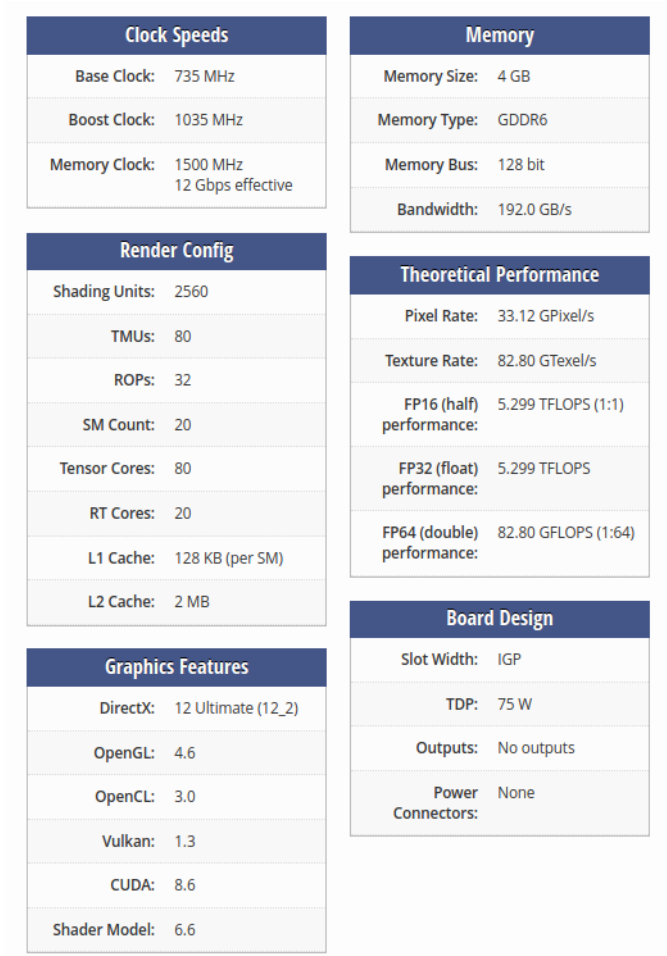
Specyfikacja procesora Intel Core i7—2640M

8.13. (2021) GPU NVIDIA GeForce RTX 3050 Ti 4GB

Karta jednej z najnowszych serii. Wprowadzona na rynek w 2021 roku. Posiada aż 2560 jednostek cieniujących, pamięć GDDR6 rozmiaru 4GB. Szyna danych to jednak tylko 128 bitów. Jesteśmy ograniczeni dodatkowo 80W poboru mocy, jest to bowiem wersja dla komputerów przenośnych. Zainstalowana jest w komputerze Dell G15. Uzyskuje aż 5299 GFLOPS w teście FP32 i aż 82 GFLOPS w teście FP64. Posiada 20 rdzeni typu *ray-traycing*. Wspiera wersję 3.0 standardu OpenCL.



Podsumowanie dla karty NVIDIA GeForce RTX 3050 Ti



Clock Speeds	
Base Clock:	735 MHz
Boost Clock:	1035 MHz
Memory Clock:	1500 MHz 12 Gbps effective

Render Config	
Shading Units:	2560
TMUs:	80
ROPs:	32
SM Count:	20
Tensor Cores:	80
RT Cores:	20
L1 Cache:	128 KB (per SM)
L2 Cache:	2 MB

Graphics Features	
DirectX:	12 Ultimate (12_2)
OpenGL:	4.6
OpenCL:	3.0
Vulkan:	1.3
CUDA:	8.6
Shader Model:	6.6

Memory	
Memory Size:	4 GB
Memory Type:	GDDR6
Memory Bus:	128 bit
Bandwidth:	192.0 GB/s

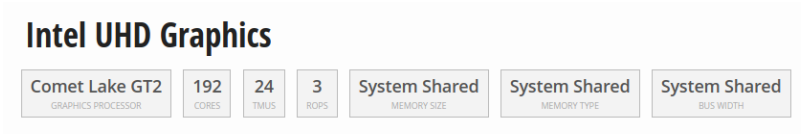
Theoretical Performance	
Pixel Rate:	33.12 GPixel/s
Texture Rate:	82.80 GTexel/s
FP16 (half) performance:	5.299 TFLOPS (1:1)
FP32 (float) performance:	5.299 TFLOPS
FP64 (double) performance:	82.80 GFLOPS (1:64)

Board Design	
Slot Width:	IGP
TDP:	75 W
Outputs:	No outputs
Power Connectors:	None

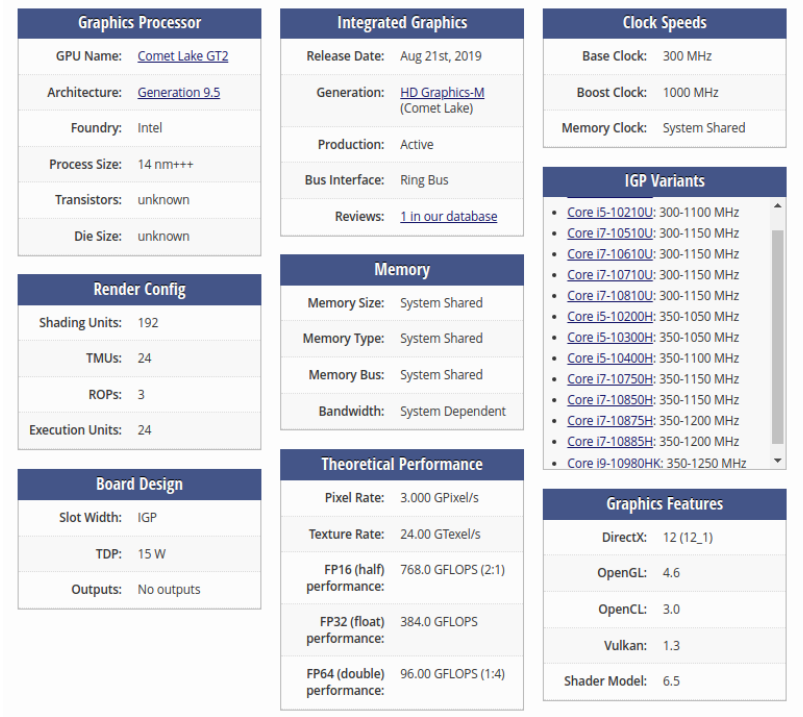
Specyfikacja karty NVIDIA GeForce RTX 3050 Ti

8.14. (2019) GPU Intel UHD Graphics

Karta wbudowana w układ procesora Intel Core i5—10200H. Posiada 192 jednostki cieniujące. Maksymalny pobór mocy to 15W. Wspiera jednak większość nowych standardów. Uzyskujemy przyzwoitą wydajność obliczeniową na poziomie 384 GFLOPS w teście FP32. W niektórych przypadkach stosowanie tego układu do obliczeń będzie zasadne, gdyż ma krótszą ścieżkę dostępu do danych z procesora i pamięci niż w przypadku dedykowanej karty graficznej. Stosowanie zatem tandemów CPU plus wbudowana grafika może być skuteczne w przyspieszaniu pracy wielu aplikacji.



Podsumowanie dla karty Intel UHD Graphics



Graphics Processor

GPU Name:	Comet Lake GT2
Architecture:	Generation 9.5
Foundry:	Intel
Process Size:	14 nm+++
Transistors:	unknown
Die Size:	unknown

Render Config

Shading Units:	192
TMUs:	24
ROPs:	3
Execution Units:	24

Board Design

Slot Width:	IGP
TDP:	15 W
Outputs:	No outputs

Integrated Graphics

Release Date:	Aug 21st, 2019
Generation:	HD Graphics-M (Comet Lake)
Production:	Active
Bus Interface:	Ring Bus
Reviews:	1 in our database

Memory

Memory Size:	System Shared
Memory Type:	System Shared
Memory Bus:	System Shared
Bandwidth:	System Dependent

Theoretical Performance

Pixel Rate:	3.000 GPixel/s
Texture Rate:	24.00 GTexel/s
FP16 (half) performance:	768.0 GFLOPS (2:1)
FP32 (float) performance:	384.0 GFLOPS
FP64 (double) performance:	96.00 GFLOPS (1:4)

Clock Speeds

Base Clock:	300 MHz
Boost Clock:	1000 MHz
Memory Clock:	System Shared

IGP Variants

- Core i5-10210U: 300-1100 MHz
- Core i7-10510U: 300-1150 MHz
- Core i7-10610U: 300-1150 MHz
- Core i7-10710U: 300-1150 MHz
- Core i7-10810U: 300-1150 MHz
- Core i5-10200H: 350-1050 MHz
- Core i5-10300H: 350-1050 MHz
- Core i5-10400H: 350-1100 MHz
- Core i7-10750H: 350-1150 MHz
- Core i7-10850H: 350-1150 MHz
- Core i7-10875H: 350-1200 MHz
- Core i7-10885H: 350-1200 MHz
- Core i9-10980HK: 350-1250 MHz

Graphics Features

DirectX:	12 (12_1)
OpenGL:	4.6
OpenCL:	3.0
Vulkan:	1.3
Shader Model:	6.5

Specyfikacja karty Intel UHD Graphics

8.15. (2012) CPU Intel Core i3 —2328M

Jest to najsłabszy procesor z wszystkich stosowanych w testach. W testach teoretycznych będzie on kilkukrotnie mniej wydajny niż pozostałe. W praktyce w większości zastosowań podstawowych różnic nie odczujemy. Różnice pomiędzy bardzo słabymi, a bardzo mocnymi urządzeniami zauważalne będą w obliczeniach intensywnych. Procesor ten testowano w komputerze Lenovo Thinkpad T420i.

Rdzeni	2
Strumieni	4
Częstotliwość podstawowa	2.2 GHz
Maksymalna częstotliwość	2.2 GHz
Pamięć podręczna 1-go poziomu	64K (na rdzeń)
Pamięć podręczna 2-go poziomu	256K (na rdzeń)
Pamięć podręczna 3-go poziomu	3 MB (łącznie)
Proces technologiczny	32 nm
Rozmiar kryształu	149 mm^2
Maksymalna temperatura rdzenia	85C (PGA); 100C (BGA)
Ilość tranzystorów	624 million
Obsługa 64 bitów	+
Zgodność z Windows 11	-
Odblokowany mnożnik	-

Specyfikacja procesora Intel Core i3 — 2328M

8.16. (2012) CPU Intel Xeon E3 1220

Procesor testowany w zestawie HP z210. Wprowadzony na rynek w 2011 roku. Pomimo dekady różnicy nie odstaje on teoretyczną wydajnością bardzo znacząco od nowszych rozwiązań. Może być zatem stosowany w rozwiązaniach energooszczędnych. Całkowity pobór mocy takiego zestawu pod niewielkim obciążeniem to zaledwie 45W.

Rdzeni	4
Strumieni	4
Częstotliwość podstawowa	3.1 GHz
Maksymalna częstotliwość	3.4 GHz
Pamięć podręczna 1-go poziomu	64 KB (na rdzeń)
Pamięć podręczna 2-go poziomu	256 KB (na rdzeń)
Pamięć podręczna 3-go poziomu	8 MB (łącznie)
Proces technologiczny	32 nm
Rozmiar kryształu	216 mm^2
Maksymalna temperatura rdzenia	69 °C
Ilość tranzystorów	1,160 million
Obsługa 64 bitów	+
Zgodność z Windows 11	-
Odblokowany mnożnik	-

Specyfikacja procesora Intel Xeon E3 1220

8.17. (2020) CPU Intel Core i5 —10200H

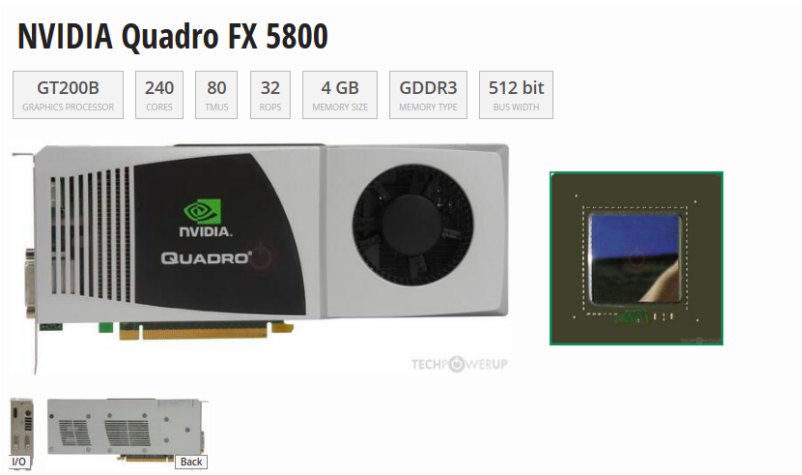
Kolejnym testowym urządzeniem jest procesor Intel Core dziesiątej generacji. Jest to bardzo wydajna jednostka. Posiada 8 wątków o wysokiej maksymalnej częstotliwości pracy. W połączeniu z wbudowaną grafiką UHD można uzyskiwać ciekawe efekty synergii, w tym krótkiego czasu dostępu i wymiany danych.

Rdzeni	4
Strumieni	8
Częstotliwość podstawowa	2.4 GHz
Maksymalna częstotliwość	4.1 GHz
Pamięć podręczna 1-go poziomu	64K (na rdzeń)
Pamięć podręczna 2-go poziomu	256K (na rdzeń)
Pamięć podręczna 3-go poziomu	6 MB (łącznie)
Proces technologiczny	14 nm
Maksymalna temperatura rdzenia	100 °C
Maksymalna temperatura obudowy (TCase)	72 °C
Obsługa 64 bitów	+
Zgodność z Windows 11	+
Odblokowany mnożnik	-

Specyfikacja procesora Intel Core i5 — 10200H

8.18. (2008) GPU NVIDIA Quadro FX 5800

Karta pochodzi z 2008 roku. W momencie premiery kosztowała 3499 USD. Wydajność na poziomie 622 GFLOPS w teście FP32 w tamtym czasie to bardzo dobry wynik. Obecnie też nie jest to najgorszy wynik a posiadanie 4GB pamięci oznacza, że karta i jej 240 jednostek cieniujących może być stosowana również dzisiaj do zastosowań profesjonalnych.



Wygląd karty NVIDIA Quadro FX 5800

Clock Speeds	
GPU Clock:	610 MHz
Shader Clock:	1296 MHz
Memory Clock:	800 MHz 1600 Mbps effective

Render Config	
Shading Units:	240
TMUs:	80
ROPs:	32
SM Count:	30
L2 Cache:	256 KB

Board Design	
Slot Width:	Dual-slot
Length:	267 mm 10.5 inches
Width:	111 mm 4.4 inches
Height:	40 mm 1.6 inches
TDP:	189 W
Suggested PSU:	450 W
Outputs:	2x DVI 1x DisplayPort 1x S-Video
Power Connectors:	1x 6-pin + 1x 8-pin
Board Number:	P607

Memory	
Memory Size:	4 GB
Memory Type:	GDDR3
Memory Bus:	512 bit
Bandwidth:	102.4 GB/s

Theoretical Performance	
Pixel Rate:	19.52 GPixel/s
Texture Rate:	48.80 GTexel/s
FP32 (float) performance:	622.1 GFLOPS
FP64 (double) performance:	77.76 GFLOPS (1:8)

Graphics Features	
DirectX:	11.1 (10_0)
OpenGL:	3.3
OpenCL:	1.1
Vulkan:	N/A
CUDA:	1.3
Shader Model:	4.0

Specyfikacja karty NVIDIA Quadro FX 5800

9. PODSUMOWANIE

Poświęciwszy sporo czasu na testowanie różnego rodzaju sprzętu CPU i GPU oraz poszczególnych wersji standardu OpenCL **mam kilka wniosków**. Przede wszystkim wskazałbym, że OpenCL nie jest wykorzystywany w zbyt wielu aplikacjach, a być powinien. Korzyści z jego stosowania jest dużo, natomiast zrozumiałym jest, że narzut programistyczny na dodanie jego obsługi do programu może być często zbyt duży. Jeśli chodzi o sam sprzęt, to wsparcie jest bardzo duże począwszy od urządzeń z 2006 roku. Zdarzają się jednak pewne zestawy komputerowe, w których brak jest np. karty obsługującej OpenCL jak choćby Intel HD Graphics 3000. Wsparcie samego procesora w takim zestawie będzie niewystarczające, gdyż **atutem stosowania OpenCL będzie współdziałanie wielu urządzeń jednocześnie.**

Gdzie zauważymy pozytywne rezultaty, a gdzie ich nie będzie? **Nie zauważymy praktycznie nic tam, gdzie algorytm wybrany do pracy będzie nieadekwatny, to jest taki, którego nie da się podzielić i uwspółbieżnić**. Nie zauważymy również nic pozytywnego jeśli ciągle będziemy musieli transmitować i odczytywać dane bowiem na tych **transferach** stracimy bardzo dużo. **Pozytywne rezultaty zauważymy tam, gdzie zastosujemy wielowymiarowość i intensywne obliczeniowo algorytmy, np. w przetwarzaniu grafiki**. Interesującym wnioskiem jest również sugestia aby **współużytkować moc obliczeniową CPU i wbudowanego w niego GPU. Unikamy wtedy niektórych ograniczeń na transferach danych**.

Standard OpenCL, podobnie jak OpenGL swoje lata świetności ma już za sobą. Firmy opracowujące takie standardy na przestrzeni lat opracowały swoje równoległe rozwiązania takie jak Metal czy CUDA [233]. Standardy niosą wartość jeśli są rzeczywiście uniwersalne i przejrzyste. Takim właśnie standardem jest OpenCL. Standardem tego typu nie będzie ani Metal ani CUDA. Naturalnym porównaniem są uniwersalne platformy uruchomieniowe. Jest nią Java, ale nie. NET Framework. **Mam nadzieje, że standard OpenCL jednak przetrwa**

próbę czasu, a programiści będą korzystać z mocy obliczeniowej wszystkich dostępnych urządzeń.

10. BIBLIOGRAFIA

10.1. Książki

[008] „OpenCL: A Parallel Programming Standard”, praca zbiorowa, Computing in Science and Eng., 2010

[009] „OpenCL. Akceleracja GPU w praktyce”, M. Sewerwain, Helion, 2014

[010] „OpenCL Programming Guide for CUDA Architecture”, NVIDIA, 2012

[011] „OpenCL Programming by Example”, Banger, Bhattacharyya, Packt, 2013

[012] „Język C”, Kernighan, Ritchie, WNT, 1987

[013] „Język C, interpretacja standardu”, J. Bielecki, WNT, 1987

[014] „OpenCL In Action”, M. Scarpino, Manning, 2012

10.2. Wykłady

[103] „OpenCL buffers and complete examples”, Thomas More

[104] „Obliczanie liczby π metodą Monte-Carlo”, Uniwersytet Warszawski, 2006

10.3. Internet

[W] „Wikipedia”, praca zbiorowa, wikipedia.org

[213] „Tak działają najszybsze karty graficzne”, praca zbiorowa, chip.pl, 2008

[214] „headache for clEnqueueNDRangeKernel local work size”, stackoverflow.com

[215] „measuring execution time of OpenCL kernels”, stackoverflow.com

[216] „Understanding kernels, work-groups and work-items”, ti.com

[217] „Six Ways to SAXPY”, M. Harris, nvidia.com, 2012

[218] „Barriers in OpenCL”, stackoverflow.com

[219] „Do your CPU and GPU support OpenCL”, V. Hindriksen, StreamHPC, 2011

[220] „GT200: Nvidia GeForce GTX 280 analysis”, geeks3d. com, 2008

[221] „Info about device query”, khronos.org, 2014

[222] „Local and global work size limits”, khronos.org, 2016

[223] „Akcelerator graficzny Nvidia Tesla K20X dla superkomputerów”, P. Maziarz, benchmark.pl, 2012

[224] „How do I know if the kernels are executing concurrently”, stackoverflow.com

[225] „opencl — questions about global and local scale of work”, try2explore.com

[226] „OpenCL local_work_size issues”, stackoverflow.com

[227] „Confused by CL_DEVICE_MAX_COMPUTE_UNITS”, stackoverflow.com

[228] „Struggling to understand barrier CLK_GLOBAL_MEM_FENCE”, reddit.com

[229] „The Generic Address Space in OpenCL 2.0”, intel.com

[230] „Double Pointer”, geeksforgeeks.org

[231] Specyfikacje GPU, techpowerup.com

[232] Specyfikacje CPU, technical.city

[233] „2020 GPGPU Roundup: Metal vs. CUDA vs. OpenCL, AMD vs. Nvidia”, macfinder.co.uk

Dotychczasowe książki autora to:

„Podstawy elektroniki i budowa komputera w symulatorze”, 2021
„Jakość oprogramowania, podręcznik dla profesjonalistów”, 2020
„Programowanie w języku Ruby, mikrousługi i konteneryzacja”, 2019
„Ruby on Rails, ćwiczenia”, 2007

Niniejsza książka opisuje standard OpenCL, który umożliwia połączenie mocy obliczeniowej procesorów i kart graficznych.

ISBN 978-83-8273-803-2

9 788382 738032 >

Rideró

Wydaj książkę
profesjonalnie!