Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki teoretycznej i Systemów Informacyjno-Pomiarowych

Praca dyplomowa inżynierska

na kierunku Informatyka Stosowana
w specjalności Inżynieria Oprogramowania

Symulacja Cieczy przy pomocy procesora graficznego

Mikołaj Klębowski
Numer albumu 299253

promotor
dr inż. Zuzanna Krawczyk

Warszawa 2022

Spis treści

Streszczenie

Celem tej pracy dyplomowej jest zaprezentowanie i porównanie różnych technik programowania na karcie graficznej na przykładzie symulacji cieczy.

Pierwsza część tej pracy omawia przyczyny trendu polegającemu na tworzeniu oprogramowania co raz bardziej opartego o obliczenia równoległe.

Następny rozdział prezentuje wybrane techniki programowania na kartach graficznych. Omawia ich model programowania oraz model pamięci. Dodatkowo bezpośrednio porównuje do siebie rozwiązania, które podchodzą do tego zagadnienia w podobny sposób.

Rozdział trzeci przedstawia podstawy fizyczne dynamiki płynów, na podstawie których powstają symulacje.

Część czwarta przedstawia stworzoną aplikację. Omawia różnice pomiędzy poszczególnymi rodzajami symulacji. Przedstawia wymagania potrzebne do jej uruchomianie, wymienia technologie wybrane do stworzenia aplikacji. Tłumaczy sposób działania symulacji. W ostatniej części prezentuje jej wewnętrzną budowę i interfejs użytkownika.

Ostatni rozdział opisuje sposoby wyliczania możliwego do uzyskania przyśpieszenia, uzyskane przyśpieszenie oraz wnioski z uzyskanych danych.

Słowa kluczowe :

Cuda, OpenACC, QT, Symulacja cieczy, GPU

Abstract

The aim of this thesis is to present various technics of programing on graphics processing unit.

The first part of this work discusses the reasons for the trend of creating software that is increasingly based on parallel computing.

The next chapter presents selected programming techniques using graphics cards. It discusses their programming and memory models. In addition, it directly compares solutions that similarly approach this issue.

The third chapter presents the physical basis of fluid dynamics on witch simulations are based.

The fourth part presents the created application. Discusses the differences between different types of simulations. It presents the requirements needed to run it and lists the technologies selected to create the application. Explains how the simulation works. In the last part, it presents its internal structure and user interface.

The last chapter describes the methods of calculating the possible acceleration. Presents the acceleration obtained and the conclusions from the obtained data.

Keywords:

Cuda, OpenACC, QT, Fluid simulation, GPU

Wstęp

Ruch cieczy jest bardzo skomplikowanym zagadnieniem, który nie zostało jeszcze w pełni poznane. Pomimo tego, jego symulacje są niezbędne w wielu zagadnieniach technicznych, budowlanych oraz wizualnych. Bardzo wiele obiektów wykazuje właściwości płynów. Woda w szklance, powietrze opływające skrzydła samolotu czy wnętrze ziemi w długim czasie, podlegają zasadom opisujących ruch płynów.

Podczas przeprowadzaniu symulacji tego typu, potrzeba wykonać wiele obliczeń, które w tradycyjny sekwencyjny sposób, zajęłyby zbyt dużo czasu. Rozwiązaniem tego problemu są obliczenia równoległe. Używając dostępnych rdzeni do jednoczesnej obróbki danych, można ten proces znacznie przyspieszyć. Najwięcej rdzeni na rynku mają karty graficzne. W porównaniu do procesorów które, poza jednym wyjątkiem, zawierają ich obecnie mniej niż 100, w GPU ich liczbę można liczyć w tysiącach. Dlatego też w celu stworzenia takiej symulacji postanowiłem użyć do obliczeń karty graficznej.

Głównymi celami niniejszej pracy dyplomowej są:

- Prezentacja genezy rozwoju układów wielordzeniowych
- Przedstawienie wybranych metod programowania karty graficznej
- Pokazanie podstawowych zasad fizycznych dotyczących ruchu cieczy
- Demonstracja wykonanej aplikacji
- Omówienie skuteczności wykorzystanych technik programowania na karcie graficznej

Obliczenia sekwencyjne i równoległe

Algorytm to schemat postępowania, składający się z określonego ciągu czynności, prowadzący do rozwiązania ustalonego problemu.[16] Czynności te można podzielić na dwie grupy. Do pierwszej należą te, które muszą zostać wykonane szeregowo, a do drugiej te które mogą zostać zrównoleglone. Proporcja pomiędzy nimi dla każdego algorytmu jest inna. Najłatwiejsze do wykonania równolegle są pętle, w których do wykonania pojedynczej iteracji nie są potrzebne dane z poprzedniej. Występują one często w symulacjach fizycznych zjawisk, gdzie trzeba policzyć wartości dla różnych punktów w przestrzeni.

W wczesnych latach informatyki, z powodu ograniczeń sprzętowych, jedynym sposobem na wykonanie obliczeń równoległych było wykorzystanie większej ilości połączonych ze sobą maszyn lub płyt z wieloma kośćmi CPU. Obecnie pomimo posiadania wielu rdzeni procesory ustępują pod tym względem kartom graficznym.

Rozwój wielordzeniowych CPU

Od powstania pierwszych mikroprocesorów, w roku 1971, głównymi sposobami zwiększenia możliwości układów obliczeniowych było zwiększanie częstotliwości ich pracy oraz optymalizacja istniejących i dodawanie nowych instrukcji, które wykonają bardziej skomplikowane obliczenia podczas mniejszej ilości cyklów. Taki stan rzeczy utrzymywał się do 2000r.

Wyższa częstotliwość pracy procesora wymagała większego poboru mocy zasilania. Większa ilość pobieranej energii elektrycznej spowodowała zwiększenie ilości wydzielanego ciepła przez układ. Podniosło to znacznie temperaturę pracy układów scalonych. Pasywne chłodzenie przestało wystarczać do utrzymania odpowiedniej temperatury podczas pracy.

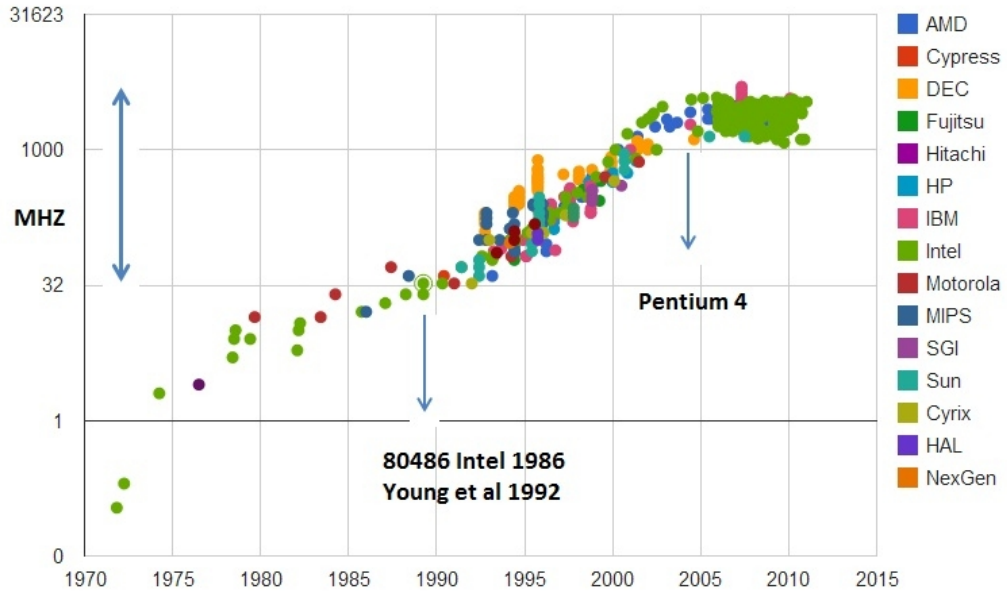

Illustration 1: Wykres częstotliwości pracy modeli CPU na przestrzeni lat

Rozwiązaniem okazało się zastosowanie dodatkowych urządzeń, których celem było chłodzenie. Obecnie stosuje się chłodzenie powietrzem oraz chłodzenie wodne. Niestety i te sposoby odprowadzania ciepła z układu, pomimo ciągłego rozwoju, nie były w stanie nadążyć za ciepłem generowanym przez CPU. Dlatego 15 lat temu projektanci procesorów zbliżyli się do granicy fizycznych możliwości układów krzemowych, dzięki czemu częstotliwość pracy przestała rosnąć i zatrzymała się na maksymalnym poziomie około 5Ghz. Rekordem uzyskanym dzięki chłodzeniu przy pomocy ciekłego azotu jest 8,7 GHz.

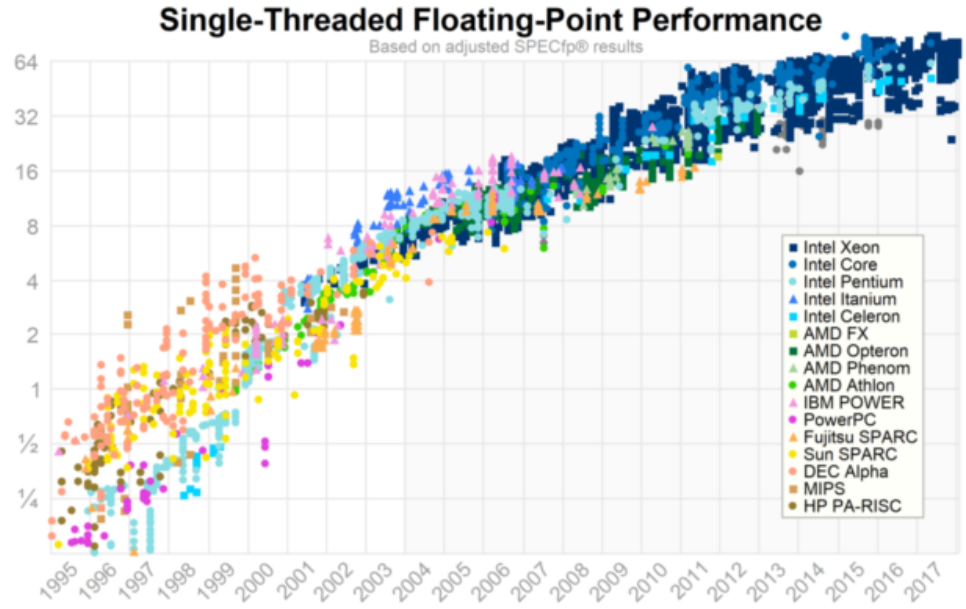

Illustration 2: Wyniki benchmarka Spec dla liczb zmienno przecinkowych

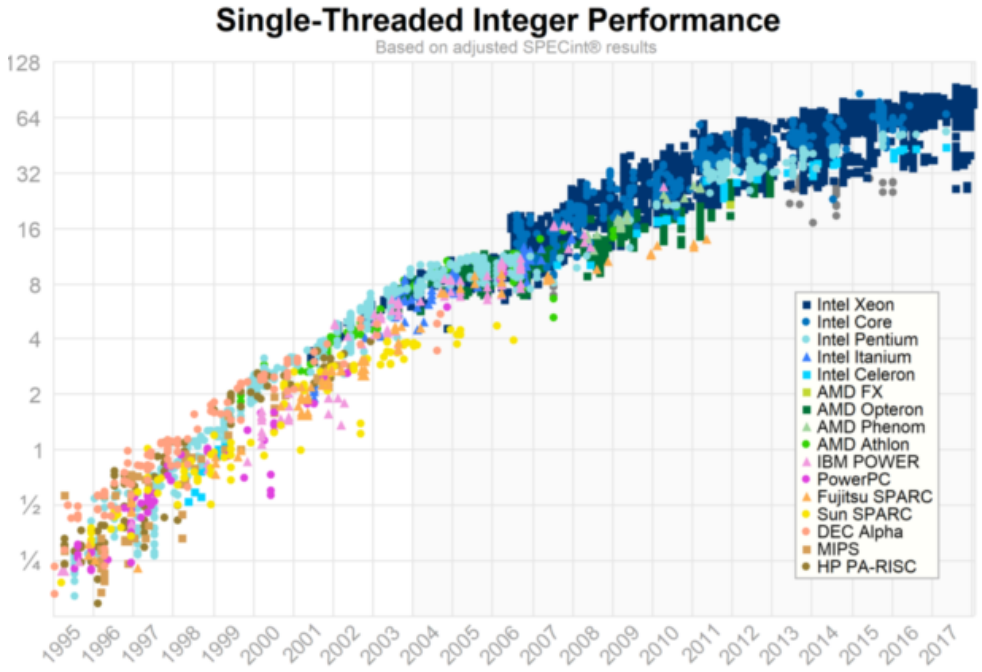

Illustration 3: Wyniki benchmarka Spec dla liczb całkowitych

Co raz większe trudności z osiągnięciem wyższej częstotliwości pracy procesora spowodowały spowolnienie rozwoju wydajności pojedynczych rdzeni. Można to zaobserwować na danych benchmarku SPECi dla wydajności jednowątkowej operacji na liczbach całkowitych i zmiennoprzecinkowych.

Jak widać na powyższych wykresach nadal, z każdą generacją nowych procesorów udaje się zwiększyć możliwości obliczeniowe pojedynczych rdzeni, jednak ten proces spowolnił i jest osiągany tylko dzięki zastosowaniu nowych zestawów instrukcji takich jak sse i avx. Wprowadzają one polecenia do obliczeń na wielu liczbach na raz co jest pewnego rodzaju zrównoleglaniem działania pojedynczego rdzenia.

Stojąc przed wyzwaniem stworzenia co raz lepszych maszyn producenci opracowali procesory wielordzeniowe. Pierwszym takim urządzeniem było Power4 od IBM wydane w 2001 roku. Został on stworzony na potrzeby serwerów. W maju 2005 kolejne firmy wydały swoje pierwsze dwurdzeniowe procesory były to Pentium D od Intela oraz Athlon 64 x2 od AMD.

Ich główną przewagą była możliwość wykonywania dwóch programów naraz, jednak niewiele aplikacji potrafiło wykorzystać stwarzane przez to możliwości. Dopiero sukces nowych konstrukcji zapoczątkowały nowy trend polegający na tworzenie procesorów z coraz to większą liczbą rdzeni.

Obecnie największą ilością rdzeni w jednym procesorze może pochwalić się firma Ampere z swoim 128 rdzeniowym Altra Max. Jednak AMD już zapowiedziało na 2023 rok swoje 128 rdzeniowe modele.

Pomimo tak dużych liczb pod względem liczby rdzeni nie mogą się one równać z możliwościami kart graficznych.

Rozwój GPU

Układy wspierające renderowanie i wyświetlanie grafiki istniały już od lat 70. ubiegłego wieku, jednak obecną formę przybrały dopiero w roku 1999 dzięki premierze GTX 256 mającą 4 równolegle pipeliny (połączone ze sobą układy wyspecjalizowane w odpowiednich zadaniach). W każdym pipelinie zastosowano jedną jednostkę cieniującą. Jednostka cieniująca to pojedynczy rdzeń mogący wykonać prostsze względem rdzeni procesora operacje. Ich nazwa pochodzi od pierwotnego celu jakim było wyliczanie cieni w grafice 3D. Każdy producent nazywa je inaczej, jednak nie zmienia to faktu, że to one są najbardziej liczne i wszechstronne w obecnych kartach graficznych. Współcześnie w najbardziej zaawansowanych wersjach ich liczba może wynieść nawet 6912 sztuk.

Technologie programowania kart graficznych

Do programowania kart graficznych istnieje wiele sposobów, które można podzielić ze względu na przeznaczenie. Pierwsza grupa służy do zastosowań związanych z grafiką komputerowa, a druga do obliczeń równoległych.

W tej pracy skupię się na rozwiązaniach przeznaczonych do obliczeń równoległych.

OpenACC

OpenACC jest wysoko poziomowym standardem dla języków C, C++ i Fortran. Został on stworzony w celu uproszczenia konwersji programu z sekwencyjnego na równoległy. Model programowania OpenACC polega na wskazaniu kompilatorowi przy pomocy dyrektyw pragma kodu, który nadaje się do zrównoleglenia. Dyrektywy wyglądają następująco

#pragma acc directive [clause [[,] clause]...] []

W zależności od rodzaju użytej dyrektywy można przekazać kompilatorowi różny poziom kontroli nad kodem.

Kernels – wskazuje obszar w którym może znajdować możliwy do wykonania równolegle kod, jednak rozpoznanie jego i sposób w jaki to zostanie wykonane zależy od kompilatora[ACC PG]. Jest to najprostsza do użycia dyrektywa, jednak jakość wykonanej implementacji nie zależy od programisty.

Parallel – wskazuje obszar kodu który ma zostać wykonany równolegle, jednak sama w sobie nie jest przydatna i z tego powodu najczęściej jest łączona z dyrektywą loop

Loop – wskazuje ona dodatkowe informacje na temat następnej pętli występującej w kodzie, w celu zapewnienia poprawności lub zoptymalizowania wygenerowanego kodu.

Routine – daje kompilatorowi niezbędne informacje o funkcjach, które mogą zostać wywołane z zrównoleglanych obszarów.

Independent – informuje kompilator, że poszczególne iteracje są niezależne

Seq – wskazuje kod który musi być wykonany szeregowo.

W przeciwieństwie do nisko poziomowych sposobów programowania na karcie graficznej nie trzeba ręcznie zarządzać pamięcią. Utrudnia to pełne wykorzystanie możliwości urządzeń, jednak jest znacznie łatwiejszy w użyciu. OpenACC pozwala na skorzystanie z pamięci jednorodnej jak i oddzielnej pamięci podzielanej na przestrzeń procesora i urządzenia.

Niestety jest to tylko standard i tylko część kompilatorów go obsługuje. Od czasu wcielenia przez przez firmę Nvidia kompilatora PGI do Nvidia HPC SDK, OpenACC działa tylko na systemach operacyjnych opartych na Linuxie.

Cuda

Cuda to platforma i API stworzona przez firmę Nvidia w 2007 umożliwiająca pisanie programów wykonywanych na kartach graficznych tej firmy. Niestety działa tylko z urządzeniami firmy Nvidia. Dzięki temu CUDA jest szybsza niż inne rozwiązania, ponieważ jest tworzona razem z urządzeniami, które mają z nią współpracować.

CUDA opiera się na języku C/C++, jednak do jej działania jest potrzebny specjalnie do tego przeznaczony kompilator. Pliki źródłowe mają rozszerzenie .cu, a nagłówkowe .cuh. Oprócz kompilatora, zostały przygotowane specjalne debugery Nsight każdy przeznaczony do innego zastosowania.

Programowanie cuda opiera się na tworzeniu funkcji zwanych kernel. Kiedy kernel zostaje wywołany jest on równolegle wykonywany na określonej ilości wątków.

Wątki

Wątki są zorganizowane w bloki. Każdy blok może zawierać do 1024 wątków. Wątki w bloku są ułożone w jednym, dwóch, lub trzech wymiarach. Taka struktura ułatwia rozdzielenie obliczeń pomiędzy poszczególne wątki w zależności czy obliczenia wykonywane są na wektorach, macierzach czy danych opisujące przestrzeń trójwymiarową. Każdy kernel ma wbudowaną zmienną threadIdx posiadającą indeksy x, y, z danego wątku. Wzór na obliczenie indeksu wygląda następująco.

Int threadId = (blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z) * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x;

Wątki są wykonywane w warpach. Każdy warp posiada ich dokładnie 32 dlatego, aby nie marnować możliwości obliczeniowych urządzenia, w bloku powinny znajdować się wątki w liczbie będącej wielokrotnością 32. Wszystkie wątki w warpie wykonują tę samą instrukcje, dlatego instrukcje sterujące powodujące różny sposób wykonania programu w różnych wątkach tego samego warpu spowalniają jego działanie.

Tak samo jak wątki są częścią bloków, tak wszystkie bloki należą do siatki. Bloki w siatce również mogą zostać ułożone w jedno, dwu lub trójwymiarowej przestrzeni. W przeciwieństwie do wątków, nie istnieje limit ilości bloków w siatce. W celu pozyskania indeksów bloków należy skorzystać z zmiennej blockIdx, a rozmiary siatki dostępne są w zmiennej gridDim. Podczas wywołania Kernela informacje o strukturze wątków i bloków jakich chcemy wykorzystać przekazujemy w składni < < <strukturaBloków, strukturaWątków > > > gdzie obydwie te informacja mogą być przekazane jako pojedyncza liczba lub zmiennej specjalnego typu dim3.

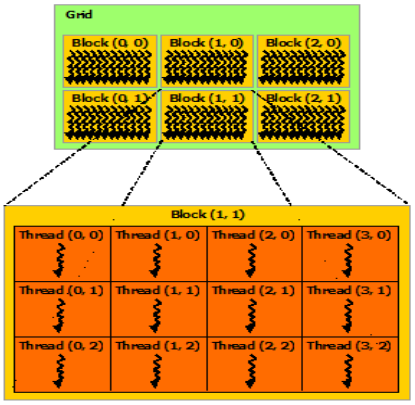

Illustration 4: Hierarchia wątków w CUDA

Pamięć

Na kartach graficznych istnieje kilka rodzai pamięci. Każdy wątek ma dostęp do swoich prywatnych rejestrów. Bloki posiadają pamięć zwaną shared, aby umieścić w niej dane należy przed typem zmiennej użyć słowa klucz __shared__. Używając tej pamięci należy pamiętać, że każdy blok będzie miał własną kopie danych, jak również nie ma żadnego sposobu, aby skorzystać z pamięci innego bloku.

Kolejnym typem jest pamięć globalna. Jest ona dostępna dla każdego wątku. Niestety dostęp do niej trwa dłużej. W celu upewnienia się, że zostaną w niej umieszczone dane można użyć słowa klucz __device__. Jednym z najważniejszych elementów optymalizacji jest zadbanie o odpowiednią lokalizacje danych. Nie tylko wewnątrz samej karty graficznej, ale również pomiędzy, pamięcią CPU, a GPU. Transfer danych pomiędzy urządzenie spowalnia w znaczny sposób działanie, i powinien być jak najbardziej zminimalizowany.

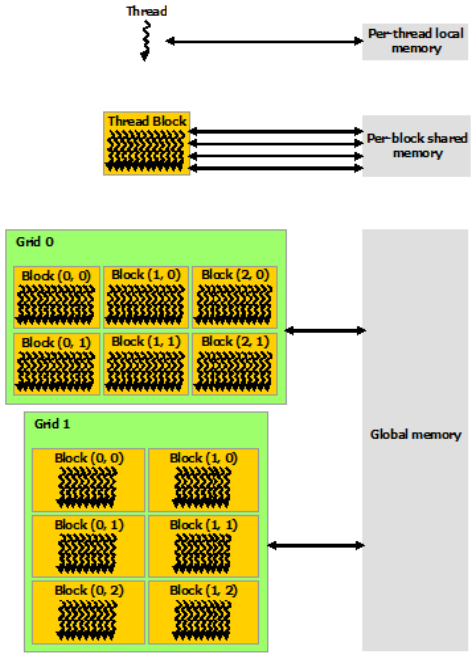

Illustration 5: Model pamięci CUDA

OpenCL

OpenCl to otwarty standard obecnie zarządzany przez Khronos Group. Został on stworzony, jako odpowiedź na Cuda. Pozwala programować na szerokiej gamie urządzeń. Jest wspierany przez wielu producentów, w tym Nvidie i Apple pomimo posiadania własnych konkurencyjnych rozwiązań. Jest to zdecydowanie najlepszy wybór jeżeli, zależy nam na wieloplatformowości.

Tak samo jak w CUDA urządzenia wykonują program zwany Kernel. Kernele są tworzone przy pomocy języka OpenCL C powstałego na podstawie C99. Podczas wykonywania programu Kernel jest przekazywany do Runtime API w formie tekstu, gdzie jest kompilowany. Dopiero wtedy jest wykonywany.

Model programowania i pamięci jest bardzo podobny do Cudy, dlatego ominę szczegółowy opis.

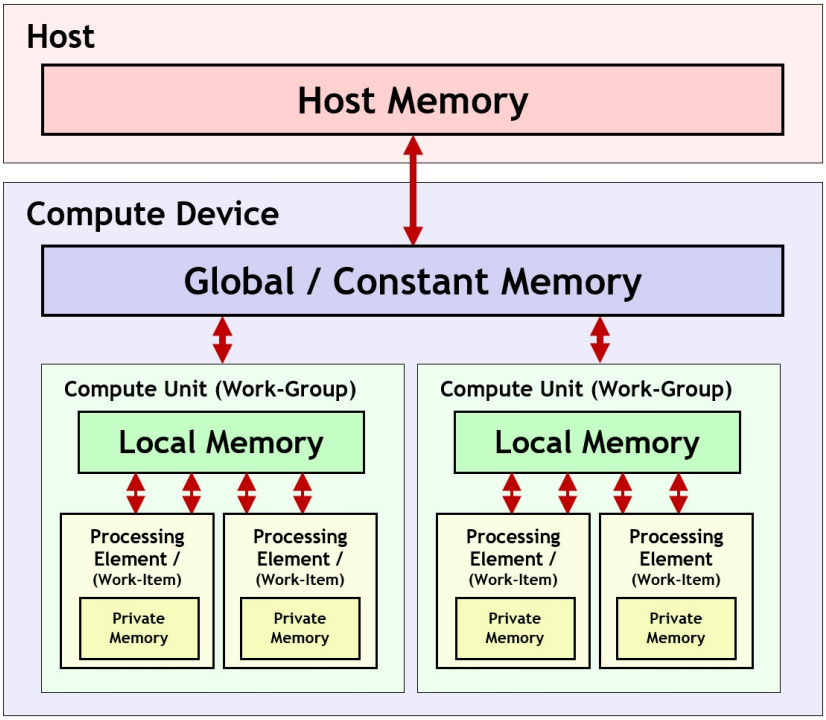

Illustration 6: Model pamięci OpenCL

OpenCL vs CUDA

Ze względu na swoje podobieństwo te dwa systemy są często ze sobą porównywane.

Podstawową różnicą jest model rozwoju obydwu platform. CUDA jest zamkniętym oprogramowaniem zależnym od Nvidia, natomiast OpenCL jest otwarty i każdy może z niego skorzystać. Dzięki temu, jeśli oprogramowanie ma być dostępne dla szerokiego grona odbiorców właściwym wyborem jest OpenCL.

Cuda jest szybsza w działaniu z dwóch powodów. Pierwszym jest różnica w sposobie wykonywania. Kernel Cuda musi być kompilowany, odpowiednim narzędziem, przed uruchomieniem, a OpenCL wykonuje kompilacje podczas trwania programu. Kolejny powód to zamknięta struktura CUDA. Została stworzona do pracy tylko z układami obliczeniowymi firmy, która ją stworzyła. Pozwoliło to na znaczną optymalizację, w przeciwieństwie do OpenCL, gdzie prędkość działania zależny również od tego jak dobrze do jego obsługi został przystosowany przez producenta układ.

Cuda jest znacznie dojrzalsza i szybciej rozwijana od OpenCL. Za ten stan odpowiada model rozwoju obydwu platform. Nvidia może dodawać kolejne funkcjonalności do Cudy według swojego uznania, natomiast do wprowadzenia zmian w OpenCL jest potrzebna zgoda firm współpracujących nad nim.

Po za warstwą techniczną warto porównać ich popularność, ponieważ przekłada się ona na czas wspierania danej platformy i zaangażowanie jej twórczość w rozwój. Zdecydowanie w tej kwestii wygrywa CUDA z wieloma poradnikami, jak i wieloma dobrymi materiałami edukacyjnymi przygotowanymi przez samego producenta.

Zarówno OpenCL jak i CUDA działają na najpopularniejszych systemach operacyjnych, Windows, MacOS, z rodziny Linux.

Podstawy fizyczne

Jednym z aspektów obliczeń komputerowych, które najbardziej zyskały na zrównolegleniu są sztuczna inteligencja oraz symulacje zjawisk fizycznych. Oprócz celów badawczych i konstrukcyjnych często są używane w grafice komputerowej.

Znaczenie symboli

ρ – gęstość cieczy

u – wektor prędkości

t – czas

p – ciśnienie

∇p – gradient zmian ciśnienia

F – zewnętrzne siły działające na ciecz

μ – lepkość

Równania Naviera-Stoke'a.

Podstawowymi równaniami opisując ruch cieczy są równania Naviera-Stoke'a.

$$\nabla \cdot u = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0$$

$$\rho \frac{\partial u}{\partial t} = -\nabla p + \mu \nabla^2 u + \rho F$$

Obydwa wzory opierają się na podstawowych prawach fizyki. Pierwsze z nich zapewnia nas o przestrzeganiu prawa zachowania masy. Suma pochodnych prędkości w każdym kierunku musi być równa zero, aby podczas symulacji nie pojawiła się lub nie zniknęła ciecz.

Drugie równanie wyprowadzane z prawa zachowania pędu powoduje, że energia w układzie pozostaje stała. Składa się ono z 4 części. $\rho \frac{\partial u}{\partial t}$ opisuje zmiany prędkości cieczy. Następnie w równaniu pojawia się gradient ciśnienia $-\nabla p$. To dzięki niemu wiemy, jakie siły wynikające z różnicy ciśnień działają na płyn.

Następnie $\mu \nabla^2 u$ opisuje siły oporów wewnętrznych wynikających z lepkości. Lepkość powodowana jest przekazywaniem przez cząsteczki cieczy części energii kinetycznej.

Ostatnim fragmentem równania jest ρF będące sumarycznym działaniem sił zewnętrznych. W zależności od celu symulacji, można ten fragment pominąć, umieścić

podstawowe siły jak grawitacja, albo nawet siły magnetyczne w celu symulacji powstawania gwiazd.

Równania Naviera-Stroke’a nie są jednak idealne i mają swoje ograniczenia.

- Równania opisują ciecz Newtonowską
- Ciecz jest izotropowa i jednorodna

Rozwiązanie równań

Równania te są w pełni rozumiane tylko w przestrzeni dwuwymiarowej. W przestrzeni trójwymiarowej nadal nie istnieje sposób, dzięki któremu moglibyśmy, przewidzieć lub unikać sytuacji w których na przykład wektor prędkości będzie nieskończenie długi. To zagadnienie zostało uznane za jeden z problemów milenijnych i nadal nie został rozwiązany.

Aplikacja

W tej części pracy chciałbym zaprezentować graficzną aplikacje zawierającą prostą interaktywną dwuwymiarową symulacją cieczy. Zawarłem w niej implementacje symulacji wykonywane sekwencyjne oraz dwie wykonywane przy pomocy GPU.

Rodzaje symulacji

Istnieje wiele technik pozwalających na przeprowadzenie symulacji cieczy. Znaczna większość z nich zalicza się do jednej z dwóch grup, symulacji cząsteczkowych lub symulacji wolumetrycznych

Symulacje cząsteczkowe

SPH(ang. Smoothed particle hydrodynamics) – to metoda symulacji płynów opiera się na zbiorze cząstek. Wylicza ona ruch i siły działające na każdą cząsteczkę indywidualnie, przy pomocy gradientów. Pozwala to na symulacje o zmieniającym się lub skomplikowanym kształcie. Pomimo wielu zalet ma pewne wady. W szczególności jest o wiele bardziej złożona obliczeniowo niż symulacje wolumetryczne, oraz nie da się jej zrównoleglić w takim samym stopniu.

Symulacja Wolumetryczny

Ten rodzaj symulacji polega na śledzeniu wartości dla fragmentów podzielonej przestrzeni. Nie symuluje się tutaj bezpośrednio cieczy tylko wylicza ile jej przepłynie z jednego miejsca w drugie. Pomimo ograniczeń, ta metoda cechuje się o wiele mniejszym poziomem złożoności obliczeniowej i można zrównoleglić większą część programu niż w SPH. Jest ona najczęściej używana w symulacjach 2D. Z tego też powodu wykorzystałem ten rodzaj symulacji. W przypadku symulacji wolumetrycznych każdy fragment może być obliczony na oddzielnym wątku.

Wymagania

Program został wykonany i przetestowany na systemach operacyjnych Windows 10 i Linux Ubuntu 20.

Do jej uruchomienia jest potrzebna karta graficzna firmy Nvidia charakteryzujące się compute capability (mozliwości obliczeniowe) w wersji 8.6 lub nowszej.

Użyte technologie

C/C++

Najwięcej narzędzi do pisania programów wykonywanych na karcie graficznej powstało w oparciu o język C/C++, jak można było zauważyć w wcześniejszym fragmencie pracy. Został on wybrany z powodu relatywnie niskiego poziomu manipulacji pamięci i szybkości działania. Jest to zrozumiałe biorąc pod uwagę główne zastosowanie programowania na GPU którym jest przyśpieszenie wykonywania obliczeń. Dodatkową zaletą tego języka jest długotrwały stabilny rozwój, dzięki czemu wielu ludzi ma z nim doświadczenie. Obecnie odpowiada za niego organizacja ISO wydająca co 3 lata nowe standardy.

CUDA

Wybrałem CUDA zamiast OpenCL, ponieważ pozwala osiągnąć lepsze przyspieszenie, posiada większą społeczność oraz wywołanie kernela jest mniej skomplikowane niż w OpenCL.

Ważnym ograniczeniem jest aby karta graficzna miała compute capability (pl. zdolności obliczeniowe) na poziomie 8.6. W celu uruchomienia kernela na karcie graficznej używam Cooperative group, dostępne od tego standardu. Pozwala to na synchronizacje wszystkich uruchomionych wątków, a nie tylko te znajdujące się w pojedynczym bloku. Wcześniej, aby móc to zrobić należało podzielić kernel na części tak, aby poprzedni kończył się, a następny zaczynał w miejscu w którym potrzebna była synchronizacja wątków.

OpenACC

Zdecydowałem się na implementacje symulacji przy użyciu OpenACC, ponieważ prezentuje on inne podejście niż pozostałe technologie. Oprócz wydajności, skupiono się również na łatwości zmiany kodu z sekwencyjnego na równoległy.

Git

W celu kontroli zmian użyłem usługi github. Repozytorium składa się z dwóch gałęzi. Odpowiadają one za wersje na różne systemy operacyjne, Windows oraz Linux Ubuntu.

Link do repozytorium: https://github.com/greenhoody/Fluid-Simulation.

QT

QT to rozbudowana platforma stworzona przy pomocy języka C/C++ służąca do tworzenia graficznych interfejsów użytkownika. Tak samo jak cuda posiada ona wrappery pozwalające na używanie jej z Pythonem.

Biblioteki QT, oprócz obsługi GUI, zawierają moduły pozwalające na skorzystaniu z wielu innych możliwości komputera. Są przygotowane do obsługi sieci, baz danych, openGL, procesów oraz wątków.

QT zawiera w sobie narzędzia potrzebne do budowania aplikacji, są to:

- Meta Object Compiler – kompilator dla plików .cpp i .h stworzonych przez użytkownika.
- User Interface Compiler – kompilator plików .ui, odpowiedzialnych za wygląd aplikacji
- qmake – tworzący Makefile na podstawie definicji projektu .pro
- QT Designer – aplikacja służąca do tworzenia GUI. Istnieje wtyczka dla Visual Studio pozwalająca na korzystanie z tego narzędzia przez IDE.
- Qt Linguist – służy do tworzenia różnych wersji językowych

Jej dystrybucja jest prowadzona na różnych płatnych licencjach, jednak jest dostępna również wersja open source.

Głównym element elementem tej platformy jest system sygnałów i slotów. Sygnał to funkcja obiektu wysyłający komunikat, a slot to funkcja tego samego lub innego obiektu która jest wywoływana, kiedy otrzyma on komunikat. W celu umieszczenia w klasie funkcji slotu lub sygnału, musi ona dziedziczyć bezpośrednio lub pośrednio po klasie QObject, zawierać macro Q_OBJECT oraz należy użyć słowa kluczowego slots lub signals dla funkcji w pliku nagłówkowym. Poniżej znajdują się fragment stworzonego przeze mnie deklaracji klasy wykorzystującej te mechanizmy

```
class MyGraphicsView : public QGraphicsView{

   Q_OBJECT

public slots:

   void refresh();

   void start();  (...) };

}
```

```
void MyGraphicsView::refresh(){
    simulation->NextFrame(pixels);
    for (int i = 1; i < this->width() ; i++) {
        for (int j = 1; j < this->height() ; j++) {
            image->setPixelColor(i - 1, j - 1, getColor(pixels[IX(i,
j)]));
        }
    }
    pixMapItem->setPixmap(QPixmap::fromImage(*image));
    this->show();
}
```

Jak widać implementacja funkcji typu slot nie różni się niczym od zwykłych fukcji.

Sygnały i sloty można połączyć ze sobą przy pomocy QTDesigner lub bezpośrednio w kodzie.

```
QObject::connect(timer, &QTimer::timeout, this, &MyGraphicsView::refresh);
```

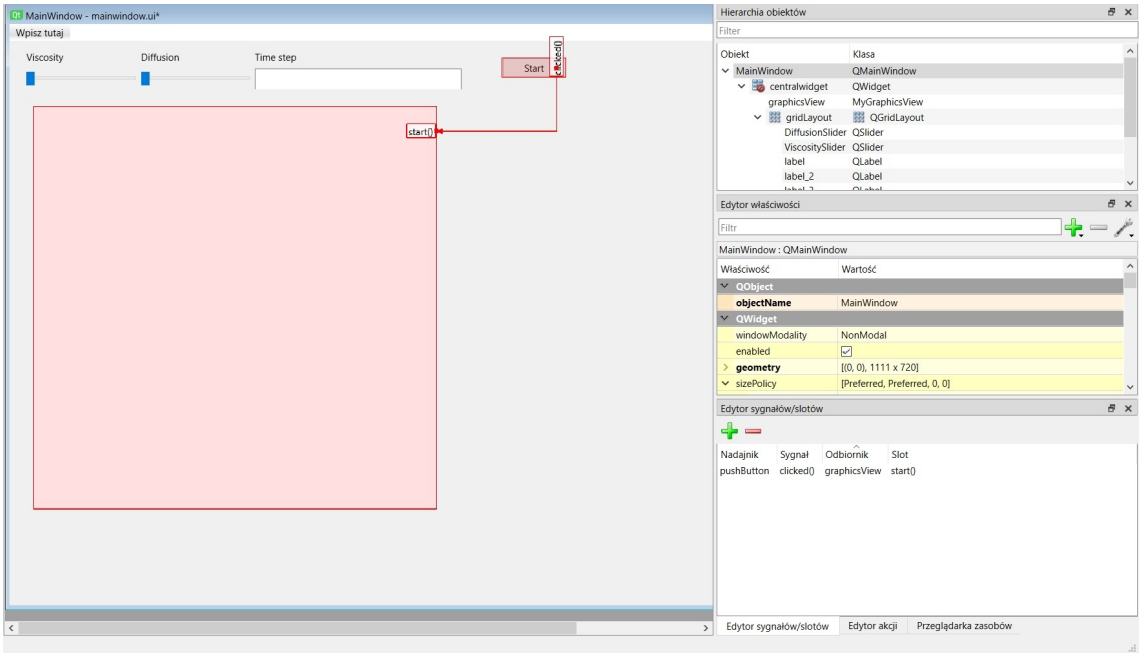

Illustration 7: Qt Creator okno łączenia signal slot

Algorytm Symulacji

Symulację oparłem na rozwiązaniu zaproponowanym przez Jos Stam'a „Real-Time Fluid Dynamics for Games". Sposób symulacji przedstawiony w tym artykule ma prosta budowę jednocześnie pozwalającą na szeroką gamę modyfikacji, dzięki czemu możliwe jest dodanie trzeciego wymiaru czy zrównoleglenie kodu.

Algorytm ten zakłada, że ciecz jest nieściśliwa. Przykładem takiej cieczy jest woda. Aby zrozumieć sposób działania symulacji należy wyobrazić pojemnik z zabarwioną przezroczystą cieczą. Podczas ruchu cieczy, porusza się za nią barwnik oraz podlega on dyfuzji. Symulacja ta ukazuje ruch tego barwnika w wodzie.

Tak jak każda symulacja wolumetryczna, ta również dzieli powierzchnie na oddzielne fragmenty. Przyjmuje się, że każdy fragment jest identycznym kwadratem. Informacje o przestrzeni są przechowywane w tablicach, gdzie każdy element odpowiada pojedynczemu fragmentowi. Potrzebne dane to gęstość oraz wektory prędkości płynu. Z racji, że dane niezbędne do wygenerowania kolejnego kroku symulacji, pochodzą z poprzedniego jej stanu, każda tablica ma swoją kopię, aby podczas obliczeń ich nie stracić. Oprócz wcześniej wspomnianych informacji potrzebne są wartości skalarne opisujące lepkość, tempo dyfuzji oraz czas jaki ma upłynąć wewnątrz symulacji pomiędzy klatkami.

Każda iteracja składa się z dwóch głównych kroków. Pierwszy to zmiany prędkości, a drugi to ruch cieczy. Obydwa etapy wykorzystują te same funkcję z wyjątkiem project(), która jest używana wyłącznie w pierwszej części.

Symulacja ta składa się z czterech/trzech kroków.

- Dodania gęstości, prędkości
- Dyfuzja
- Projekcja (tylko dla wektorów prędkości)
- Adwekcja (przemieszczenie wzdłuż odpowiednich wektorów)

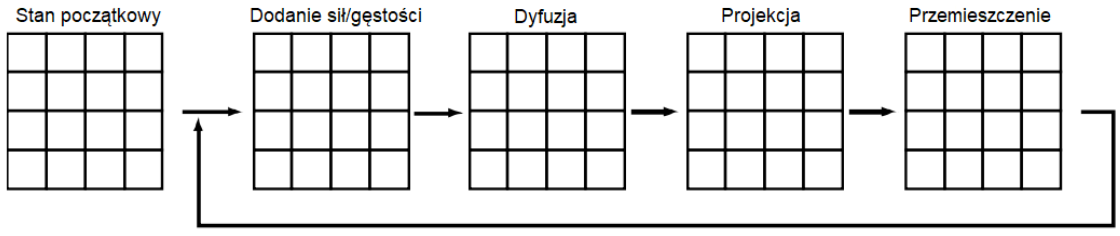

Illustration 8: Wykres działania algorytmu

Pierwszy etap symulacji jest najprostszy, ponieważ są dodawane odpowiednie wartości opisujące gęstość barwnika/prędkości dla danych pól.

Następny etap, odpowiada za ruch wynikający z zjawiska dyfuzji. Dowolna ciecz czy gaz przemieszcza się naturalnie z przestrzeni o większym stężeniu substancji do przestrzeni o mniejszym stężeniu tej substancji. Pod uwagę są brane tylko 4

sąsiadujące obszary. Najprostsza metoda polegająca na zsumowaniu różnic między fragmentem, a jego sąsiadami jak na podanym przykładzie, może być niestabilna dla dużych przepływów.

x0[IX(i,j)] = x[IX(i,j)] - a*(x[IX(i-1,j)]+x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)] -4*x[IX(i,j)]); [RFTDG]

Taka sytuacja może spowodować oscylacje, a nawet wartości ujemne dla gęstości. W celu uniknięcia tego znajduje się gęstości, które po cofnięciu w czasie dadzą stan początkowy.

x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]+ x[IX(i,j-1)]+x[IX(i,j+1)]))/(1+4*a); [RFTDG]

Kolejnym krokiem jest projekcja. Jest ona wykonywana tylko dla wektorów prędkości. Po wcześniejszych krokach rzadko zdarza się sytuacja, w której po adwekcji płynów nie zmieni się ilość podstawowej cieczy w fragmentach przestrzeni. Taka sytuacja łamała by założenie, że symulacja dotyczy cieczy nieściśliwej. Projekcja gwarantuje nam, że ruch cieczy nie spowoduje jej kompresji. Dodatkowo krok ten powoduje zawirowania czyniąc wygenerowany obraz bardziej realistyczny,

Ostatni krok to adwekcja, inaczej ruch cieczy wzdłuż wektorów prędkości lub zmiany wektorów prędkości, na podstawie ich poprzednich wartości. Główną ideą tego kroku jest potraktowanie barwnika jako cząstek. Jednak zamiast sprawdzać jak dana cząsteczka poruszyła by się, sprawdzamy skąd cząsteczka znajdująca się w centrum fragmentu przybyłaby. Niestety ten sposób działania powoduje zmianę sumarycznej ilości cieczy w całej symulacji.

Budowa wewnętrzna

Wewnętrzna budowa wersji na Linuxa jest bardziej rozbudowana od tej na Windowsa z powodu występowania OpenACC. Dlatego też, poniżej znajduje się wykres programu w wersji dla Linuxa.

Illustration 9: Wykres UML aplikacji

W celu utworzenia obiektów symulacji zastosowałem wzorzec projektowy metoda wytwórcza, który można zobaczyć w dolnej części diagramu. Dostarcza on interfejs do stworzenia obiektów, które zostaną stworzone w klasach potomnych.[MW] Był on bardzo pomocny podczas tworzenia prototypów różnych wersji symulacji.

Interface Factory składa się z jednej funkcji, której cele jest utworzenie obiektu spełniającego interfejs Simulation. Rodzaj klasy jakiego będzie obiekt symulacji, zależy od klasy implementującej interface Factory. Każda symulacja musi spełniać interfejs Simulation, przez który GUI będzie się komunikowało z nią.

Program posiada 3 różne implementacje symulacji, pierwsza wykonywana sekwencyjnie, druga stworzona przy pomocy OpenACC, a trzecia wersja CUDA.

Klasy CudaSimulation i OpenACCSimulation posiadają referencje do odpowiednich bibliotek, gdzie znajdują się implementacje symulacji. Powodem tego jest konieczność ich kompilacji przy pomocy innego kompilatora lub sposobu niż GUI. Biblioteka libcuda musiała być skompilowana przy pomocy kompilatora nvcc, a libacc z flagą -fopenacc. Funkcja AddVelocity odpowiada za dodanie barwnika do cieczy, a AddVelocity za wprawienie wszystkiego w ruch.

Standardowa klasa frameworku QT QgraphicView, nie posiadała implementacji wszystkich potrzebnych funkcji. Z tego powodu musiałem stworzyć klasę MyGraphicView. Dodałem w niej implementacje funkcji rejestrujących pozycje kliknięcia, puszczenia przycisków myszki oraz śledzącą ruch myszy. Dzięki temu mogłem zaimplementować dodawania przez użytkownika barwnika, oraz prędkości w trakcie trwania symulacji. Dodatkowo znalazł się w niej obiekt klasy Qtimer, którego zadaniem jest odliczanie czasu, aby pokazać kolejną klatkę symulacji.

Obiektem zawierającym cały interfejs użytkownika jest obiekt klasy MainWindow.

GUI

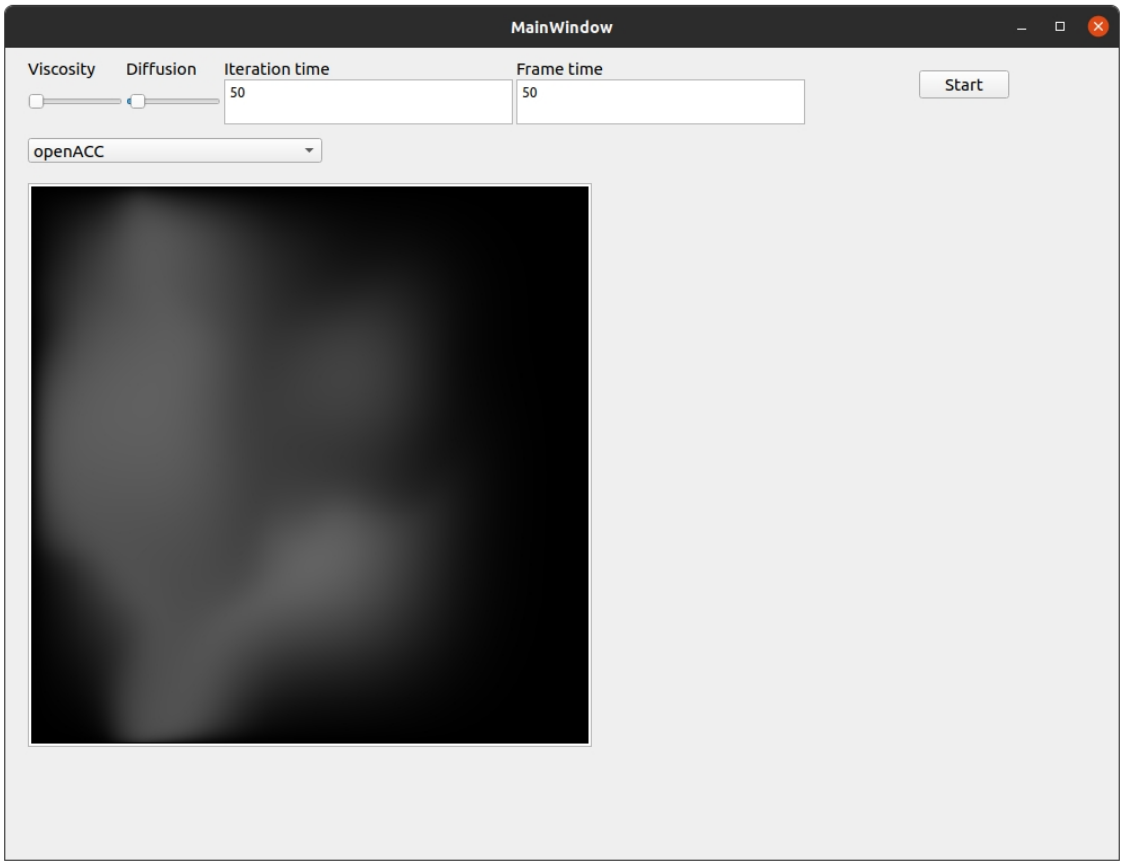

Illustration 10: Interface graficzny aplikacji

W powyższy sposób prezentuje się interfejs użytkownika. Suwaki Viscosity i Diffusion pozwalają na ustawienie podstawowych parametrów symulacji, wartości tempa dyfuzji i lepkości cieczy. W polu Iteration time znajduje się ilość czasu wewnątrz symulacji jaka ma minąć pomiędzy klatkami podana w milisekundach. Do Frame time, należy wpisać jak długi ma być czas pomiędzy kolejnymi klatkami, również w milisekundach. Domyślną wartością dla obydwu tych pól jest 50 ms. Po wciśnięciu przycisku start poprzednia symulacja, jeśli istniała, jest usuwana, a następnie tworzona nowa. Poniżej suwaków znajduje się lista z rodzajami implementacji symulacji do wyboru.

Na samym dole znajduje się główny element pokazujący stan symulacji. Im jaśniejszy kolor tym w danym pikselu występuje więcej barwnika. Po uruchomieniu symulacji można ją sterować za pomocą myszki. Lewym przyciskiem zaznacza się obszar, w którym ma zostać dodany barwnik do cieczy. Podczas trzymania prawego przycisku myszy program śledzi położenie kursora i na podstawie jego ruchów dodaje odpowiednią ilość siły wprowadzając ciecz w ruch.

Najlepszym sposobem na przekazanie wartości z elementów Viscosity, Diffusion oraz time step jest przekazanie referencji do nich obiekcie klasy MyGraphicView już po inicjalizacji wszystkich elementów przez co powstała funkcja giveRequiredElements()

```
void MyGraphicsView::giveRequiredElements(QSlider* v, QSlider* d, QPlainTextEdit *ts)

{

   this->v = v;

   this->d = d;

   this->ts = ts;

}
```

Funkcja start() klasy MyGraphicsView musiała przejąć, część pracy konstruktora polegającą na stworzeniu obiektów niezbędnych do wyświetlenia wyników symulacji. Wynika to z tego, że funkcje width() i height() nie zwracają poprawnych wymiarów wyświetlanego obszaru w konstruktorze. Po zakończeniu pracy konstruktora zwracane są poprawne dane. Powoduje to dodatkową pracę przy starcie symulacji, ponieważ należy utworzyć obiekty potrzebne do wyświetlenia wygenerowanej grafiki w innym miejscy niż konstruktor.

Przyspieszenie

W tym rozdziale omówię, uzyskane przyśpieszenia działania funkcji produkującej kolejne iteracje symulacji po zastosowaniu karty graficznej.

Łączny czas wygenerowania losowej klatki wyniósł 152487010 nanosekund. Czas wykonywania części sekwencyjnej losowej klatki wyniósł 58165 ns, co daje 0.038% całości trwania funkcji. Należy zwrócić uwagę, że większość tego czasu jest przeznaczana na kopiowaniu wyników do odpowiedniej tabeli przy użyciu memcpy(). Natomiast część nadająca się do zrównoleglenia 152428845 nanosekund co dało 99,962%.

Program był uruchamiany na karcie graficznej z 3584 rdzeniami CUDA.

Prawo Amdahl’a

Prawo Amdhal’a powstałe w 1967r. opisuje maksymalne możliwe do uzyskania przyśpieszenie dla algorytmu, który można częściowo zrównoleglić. Każde zadanie może zostać podzielone na dwie części, możliwą i nie możliwą do wykonania równoległego.

$$S = \frac{1}{s + \frac{p}{N}} \quad \text{[RAL]}$$

N – Liczba rdzeni

S - przyśpieszenie

p – procent czasu spędzonego na obliczeniach równoległych

s – procent czasu spędzonego na obliczeniach sekwencyjnych

Z tego prawa wynika bardzo istotny wniosek. Dodawanie kolejnych rdzeni od pewnego ilości przestanie przynosić zauważalne korzyści.

Na podstawie powyższego wzoru powstał jeszcze prostszy wzór do wyliczenie maksymalnego teoretycznego przyśpieszenia. Jego głównym założeniem jest ilość jednostek obliczeniowych dążących do nieskończoności, gdzie każda ma równy udział w obliczeniach.

$$\lim_{N \to \infty} T(N) = \frac{1}{1 - p}$$

Z drugiego wzoru można wyliczyć, że symulacja może zostać przyspieszona 2631 krotnie. Jest to wartość o wiele wyższa od pierwszego wzoru, który dał wynik 1517 krotnego przyspieszenia dla mojej karty graficznje

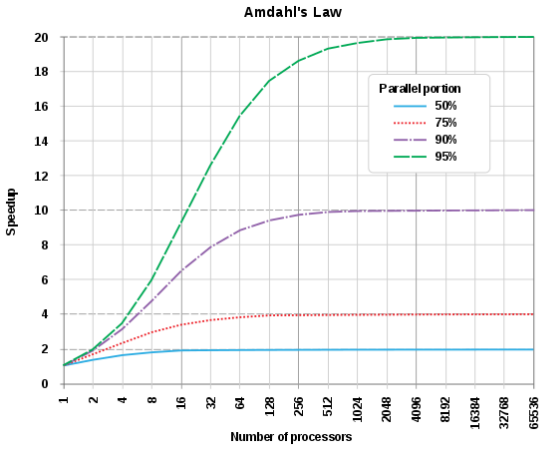

Illustration 11: Wykres przyspieszenia według prawa Amdahl'a

Prawo Gustafsona

Prawo Amdhal'a nie jest pozbawione wad. Często są możliwe do uzyskania przyśpieszania większe niż wynikałoby to z prawa Amdhala. Prawo Gustafsona zwraca wyższe wyniki od prawa Amdhala, ponieważ przyjmuje, że wraz z większą ilością zasobów, zmienia się sposób rozwiązania problemu, jak również jego wielkość. Prawo Amdhala wychodzi z założenia, że nakład potrzebnej pracy jest wartością stałą. Pomija ono fakt, że ludzie dopasowują rozmiar danych, aby otrzymać wynik w rozsądnym czasie. Z tego też powodu powstał nowy sposób obliczania przyśpieszenia.

Prawo Gustafsona mówi, że jeśli do wykonania zadania z częścią szeregową wynoszącą s procent, przeznaczymy liczbę rdzenii N oraz przeskalujemy ilość danych tak, aby zachować czas jego wykonania, to przyśpieszenie wyniesie

$$S = s + N\left(1 - s\right) = N - s\left(N - s\right)$$

Jak widać jest to funkcja liniowa.

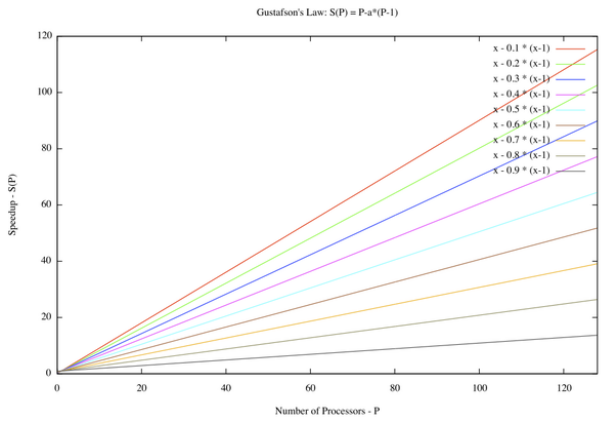

Illustration 12: Wykres przyśpieszenia według prawa Gustafsona

Z powyższego wzoru można wyliczyć, że funkcja symulacji może zostać przyśpieszona 3583 krotnie.

Osiągnięte przyspieszenie.

W poniższej tabeli przedstawiłem wyniki średniego czasu potrzebnego do wygenerowania klatki, dla symulacji o wymiarach 500 na 500 pikseli oraz 250 na 250 pikseli.

	Windows	Linux Ubuntu
Sekwencyjny 500x500	157,96 ms	153 ms
Cuda 500x500	20,18 ms (zakres od 3ms do 140 ms)	1,9 ms
OpenACC 500x500	_	19ms
Sekwencyjny 250x250	39ms	40ms
Cuda 250x250	14,58 ms (zakres od 1 do 163 ms)	1,5ms
OpenACC 250x50	_	7,6 ms

Z powyższych danych można wyciągnąć kilka wniosków.

Cuda na Windows 10 jest niestabilna. W porównania do Ubuntu gdzie czas generacji poszczególnych klatek wynosił od 1 do 3 ms, to w Windowsie zakres ten wynosił od 1 do 140 ms dla symulacji 500 na 500 pikseli oraz od 1 do 163 ms dla rozmiaru 250 na 250 pikseli. W celu sprawdzenie czy ta różnica nie wynika z obciążenia innymi programami w windowsie 10 zamknąłem niemal wszystkie możliwe procesy, jednak sytuacja się nie zmieniła. Po tym wyniku zmieniłem strategie i uruchomiłem aplikację na systemie Ubuntu podczas odtwarzania pliku video w przeglądarce i jednoczesnego działania gry 3D. Pomimo delikatnie wolniejszego tempa pracy poszczególne klatki nadal były obliczane w 2 lub 3 ms.

Kolejną ważną różnicą jest prędkość działania pomiędzy Cuda, a OpenACC. W implementacji przy pomocy OpenACC potrzebny jest transfer dużej ilości danych pomiędzy pomiędzy procesorem, a GPU podczas gdy przy implementacji Cuda taki transfer dokonuje się raz na klatkę.

Maksymalnym uzyskanym przyspieszeniem wyniosło 80,5. Wynik ten został osiągnięty dla symulacji obszaru 500x500 na systemie Ubuntu przy użyciu technologii CUDA. Wraz z czterokrotnym zmniejszeniem ilości danych przyśpieszenie to spadło do 26,7.

Widać znaczną różnicę w przyśpieszeniu pomiędzy różnymi ilościami co zgadza się z założeniami prawa Gustawsona.

Uzyskane przeze mnie przyśpieszenia są od kilkunastu do kilkudziesięciu razy mniejsze niż wynikałoby to z wyliczeń teoretycznych. Pierwszym powodem takiego stanu rzeczy jest zbyt małe doświadczenie jakie posiadam w tego typów projektach. Zapewne przy odpowiedniej wiedzy i praktyce można ten program jeszcze bardziej zoptymalizować. Kolejną przyczyną jest dodanie przesyłania danych pomiędzy CPU, a GPU. Powoduje to dodatkową pracę sekwencyjną. Szczególnie dotknięta tym została implementacja przy pomocy OpenACC.

Spis Ilustracji

Bibliografia

[1]: OpenACC API guide, 2021

[2] ACC PG: OpenACC Programmingand Best Practices Guide, 2021

[3] RFTDG: Jos Stam, Real-Time Fluid Dynamics for Gamse, 2003

[4] RAL: John L. Gustafon, Reevaluating Amdahl's Law, 1988

[5] MW: Piotr Szawdyński, metoda wytwórcza, data dostępu 20 kwiecień 2022

[6] https://refactoring.guru/pl/design-patterns/factory-method, data dostępu 25 kwietnia 2022

[7] https://doc.qt.io/qt.html data dostępu: 30 kwietnia 2022

[8] https://wiki.qt.io/ data dostępu: 30 kwietnia 2022

[9] https://developer.nvidia.com/blog/even-easier-introduction-cuda/ data dostepu: 30 marca 2022

[10] https://developer.nvidia.com/blog/cooperative-groups/ data dostępu : 30 kwietnia 2022

[11] https://docs.nvidia.com/cuda/ data dostępu 30 kwietnia 2022.

[12] https://www.youtube.com/watch?v=qsYE1wMEMPA data dostepu 10 grudzień 2021

[13] https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations dostęp 11 grudnia 2021

[14] https://www.cantorsparadise.com/the-navier-stokes-equations-461f7453d79e dostęp 11 grudnia 2021

[15] https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics dostęp 12 grudnia 2021

[16] https://pl.wikipedia.org/wiki/Algorytm dostęp 30 kwietnia 2022