Програмиране с CUDA за Ubuntu

Георги Петров

Програмиране с CUDA за Ubuntu

УВОДЕН УЧЕБНИК ЗА C/C++ ПРОГРАМИРАНЕ НА GP-GPU ТЕХНОЛОГИЯТА НА NVIDIA В LINUX UBUNTU

София
България

Първо издание 2012

Посвещавам тази книга на моите родители

Съдържание

1 Развитие на GP-GPU технологията 7

1.1 Системи за масивна паралелна обработка на данни 7

2 CUDA - Compute Unified Device Architecture 31

2.1 Хардуерна архитектура 31

2.2 Типове памет 40

2.3 Развитие на технологията CUDA 43

3 Инсталиране и настройка на работната среда 47

3.1 Инсталиране и конфигуриране на Linux Ubuntu 47

3.2 Инсталиране и конфигуриране на CUDA: драйвер за разработка, TOOLKIT и SDK 62

3.3 Компилиране на програми с nvcc 78

3.4 Писане на програми с gedit или geany 80

4 Програмиране с CUDA 85

4.1 "Hello World from CUDA!" 85

4.2 Измерване на бързодействието на програмата 100

4.3 Визуализация на 2D изображения с OpenGL . 109

4.4 Помощен файл за изчертаване на изображения - OpenGL.h 118
4.5 Да направим приложението по-бързо с CUDA 123
4.6 Генериране на шум 135
4.7 Изчисление и визуализация на хистограми с CPU 136
4.8 Изчисление на хистограми с CUDA 147
4.9 Синхронизация при операции със споделената памет 153
4.10 Прихващане на грешки в CUDA 157
4.11 Параметри на хардуера 159

5 Обработка на изображения с OpenCV 163
5.1 Инсталиране на OpenCV 163
5.2 Визуализация и съхраняване на изображения 166
5.3 Достъп до пикселите на изображението 169
5.4 Смесване на CUDA код в C приложение . . . 175
5.5 Обработка на изображението с CUDA 179
5.6 Работа с видео файлове 186
5.7 Обработка на видео файлове с CUDA 187
5.8 Съхраняване на резултатите 194

Предговор

Напредъкът в съвременатa микроелектроника допринесе за създаването на малки и евтини многоядрени процесори, бързи многоканални компютърни памети и гигабитови системни интерфейси за връзка между тях. Развитието на модерните изчислителни системи показва явното преимущество на хибридно хетерогенните архитектури. [1] Днес почти всеки собственик на персонален компютър произведен след 2006г. може да се гордее с притежанието на изчислителна мощност стотици хиляди пъти надвишаваща тази на мейнфрейм - суперкомпютрите от близкото минало. Напредъка в хардуерните технологии изисква от програмистите непрекъснато да усвояват нови езици и средства за писането на приложения, които да отговарят на силно повишаващите се очаквания на крайния потребител. В този процес от особено значение е програмистът да получи възможно по-пълен достъп до възможностите на новото оборудване, без за това да налага да усвоява изцяло нов език за програмиране. От друга страна развитието на General-purpose computing on graphics processing units (GP-GPU) технологията, съчетано с масовото производство на графични видео карти

[1] Хетерогенната изчислителна система е изградена от различни по предназначение и възможности електронни схеми за обработка на информация. Днешните компютри, PC и мобилни устройства притежават отделен едно или многоядрен централен процесор и отделен графичен процесор.

допринесе за нарастващото им ползване сред инженерите и учените в провеждането на симулации и математически изчисления. Благодарение на GP-GPU тези симулации типично отнемащи много време се извършват практически в реално време. Този учебник е посветен на писането на C++ програми за Linux Ubuntu, работещи с високопроизводителната технология за масивни паралелни изчисления – Compute Unified Device Architecture (CUDA) на корпорация Nvidia. Технологията CUDA позволява използването на възможностите на многоядрените 3D графични ускорители [2] от типа GeForce, Quadro, Tesla и други за провеждането на математически изчисления. Съвременните видео карти притежават високочестотни многоядрени графични процесори и големи обеми линейно адресируема памет. Тези им преимущества съчетани с високо производителните интерфейси за връзка с дънната платка на компютъра - PCI Express (6.4 Gbps) допринесоха за масовото им навлизане първоначално между любителите на видеоигри, а в последствие и сред програмистите и инженерите. Тези техни свойства ги правят незаменими при реализацията на паралелни програми за обработка на цифрово видео, звук, снимки, провеждането на компютърни симулации, разнообразни математически изчисления и други. Това е сравнително нова и масово навлизаща технология, чиито възможности и потенциал ще се развиват през следващите 5-6 години. Днес графичните ускорители масово навлизат и в мобилните и преносими устройства, което е гаранция за бума на този тип софтуерни приложения, който тепърва предтои. Усвояването на технологията CUDA е важно за всеки един инженер, учен или програмист на C++. Тези нови умения ще му дадат значително преимущество при писането на бързи програмни и за създаване на графични

[2] Това са видеокарти с възможност за оптимизирани изчисление при визуализация на 3D обекти, сцени и анимация

и симулационни изчислителни модели. Не случайно появата на CUDA се описва от повечето експерти, като „либерализация на супер компютрите“. CUDA дава възможност на всеки един инженер, учен или математик да притежава персонална суперкомпютърна система способна да извършва милиарди математически изчисления в секунда и при това на цена до 2000$за нова конфигурация. Считам, че учебникът е подходящ за напреднали компютърни потребители (предполага се че потребителите знаят какво е операционна система, могат да ползват текстови редактори, да менажират файлове, архиви и папки). Учебникът ще е достъпен и за програмисти на Windows и Linux C/C++ приложения. Нямащите опит с Linux могат да преминат към глава: „Инсталиране и настройка на работнатта среда“, а запознатите с тази опирационна система могат да четат директно примерите от: „Вашата първа CUDA програма“. Идеята за написването на учебника възникна при работата ми по разработката на първият в България университетски курс за програмиране с CUDA за студентите от Нов български университет. Същественият проблем на всяка нова технология, е че в първия момент на развитие тя е най-трудна за усвояване поради липса на учебници и поради стремглавите начални изменения в архитектурата и начините за нейното ползване. Естествено в интернет могат да се открият множество сайтове посветени на тази нова технология, за справка вижте библиографията. Учебникът съдържа стандартни проверени примери и задачи, някои от които заимствани от други автори. Надявам се да бъде ценен за всички, които желаят да се научат да пишат бързи програми работещи върху CUDA. Всичко което ви е нужно е наличието на персонален компютър тип PC (Personal Computer) в следната минимална конфигурация: CPU Intel 2GHz, RAM 1GB, HDD 80GB, дънна платка със слот PCI Express 8/16, видео карта от типа GeForce 100-500 или по-висок клас (списък на съв-

местимите модели може да намерите на сайта на производителя http://www.nvidia.com). Естествено работата на по-бърза машина ще ви донесе повече удоволствие и полза. Учебният материал изисква предварителни познания и опит в инсталирането и работата със разнообразен софтуер, и в частност Ubuntu. Въпреки че повечето команди за инсталирането и конфигурирането на подходящите програмни пакети да са описани, предварителният опит при работа с Linux ще ви е от полза. В процесът на работа може да се наложи изтеглянето и инсталирането на допълнителни безплатни софтуерни пакети, затова работете на компютър имащ бърза неограничена интернет връзка. Все пак, предварителният опит в писането на програми на C/C++ ще ви е добре дошъл. В края на учебника е даден кратък уводен курс съдържащ най-важните езикови конструкции за програмиране на C++ (програми, променливи, функции, работа с файлове). На съвсем начинаещите ще препоръчам следният английски учебник: „C++ Language Tutorial“, на Juan Soulié , който може да намерите на този линк: http://www.cplusplus.com/doc/tutorial. На приложените дискове ще намерите: 32 битова версия на операционната система Ubuntu 10.04 (използвана е тази версия поради по-добрата съвместимост с CUDA) и пълния пакет CUDA TOOLKIT v.4.X съдържащ инсталационните файлове за начално конфигуриране на системата. Онези от вас, на които тази технология се стори интересна могат да получат по-голяма теоретична представа за нея и да изучат по-сложни примери. Ако сте решили да се занимавате професионално с тази технология задължително репоръчвам да прочетат книгите [1, 2]. Всичкият софтуер, който се ползва в упражненията към този учебник е безплатен. Вие допълнително може да изтегляте нови (и 64 битови) версии на Ubuntu и CUDA TOOLKIT от сайтовете: http://www.ubuntu.com/ и http://developer.nvidia.com/category/z zone. Тук е мястото да благодаря на следните хора без ко-

ито този учебник нямаше да се получи: г-жа Chandra Cheij от NVIDIA, и моя колега г-н Николай Димитров, и всички други колеги и студенти, които указаха съдействие и любопитство при подготовка на учебния материал.

Глава 1

Развитие на GP-GPU технологията

Първа глава на този учебник ще ви даде обща представа за развитието на супер компютрите. Ще разберете малко повече за историята на 3D компютърните игри, и това как тези приложения стават основа за либерализация на супер компютрите, правейки ги общодостъпни за масовия програмист и потребител.

1.1 Системи за масивна паралелна обработка на данни

Развитието на цифровия хардуер и софтуер, както и алгоритмите за цифрова обработка и анализ на сигнали и изображения, позволиха такива системи да се интегрират в класически приложения ползвани във всяка една човешка дейност. От системите за графична обработка и предпечат на произведения на изкуството, кино филми, аудио записи, 3D анимация и игри, до високо прецизните медицински

компютърни томографи и магнитни резонансни скенери [1], както и системите за сигнална обработка, компресия на цифрово видео и много други сфери и приложения. Макар наличните на пазара компютърни конфигурации днес да надвишават хилядократно по производителност и бързодействие своите предшественици от преди 10 години, тези системи са оптимизирани за работа с така наречените настолни приложения: текстообработка, електкронни таблици, графични редактори, среди за програмиране и др. Този тип приложения са оптимизирани за офис работа, или иначе казано за обработка на символни цифрови данни. Допреди 1988г. все още никой освен IBM не се досеща и дори, че компютрите могат да се използват за музициране. Класическите процесорни архитектури (Intel, AMD x86, x64 модели) предлагани днес на пазара в най-разнообразни конфигурации 2, 4 и 8 ядрени блокове позволяват многократно повишаване на бързодействието на потребителския и системен софтуер. Тези хардуерни системи са създадени предимно за символна обработка и последователни изчисления, което ги прави нефункционални в приложения за сигнална и статистическа обработка. Естествено за компенсация на техните недостатъци по отношение на сигналната обработка са създадени така наречените цифрови сигнални процесори (DSP – Digital Signal Processor) и конфигурируеми логически матрици (FPGA - Field-Programmable Gate Array). Обикновенно тези хардуерни решения интегрират в себе си и стандартен централен процесор върху който работи обслужваща операционна система работеща в реално време. Този тип процесори се инсталират предимно във вградени системи, комуникационни интерфейси, радари, сонараи, медицински уреди и редица военни приложения. Те имат един съществен недостатък – прека-

[1] Технологии позволяващи направата на 2D и 3D изображения на вътрешността на човешкото тяло

лено голямо време за разработка и интегриране на нови приложения, което ги прави практически неприложими за масово ползване и провеждане на редовни научни изследвания, симулации, а и навсякъде където се налага бърза смяна на програмното обезпечаване. От друга страна в зората на компютрите, когато микрочиповете били кът изобщо никой не се е досещал за DSP и FPGA. Високата цена и сложност направили компютрите достояние само на развитите държави: САЩ, Великобритания, Франция, Германия, Япония, СССР и др. Тези машини били поместване в военни държавни учреждения, научни институти и университети. Достъпът до тях бил ограничен, и като цяло това обособило и тенденциите в развитието им, а именно те да останат големи и скъпи в продължение на 20-30 години. За научни изчисления години наред се разработват суперкомпютърни системи, като: Connection Machine (1980), MasPar (1987), Cray (1972-1995), която по-късно се слива със Silicon Graphics (1996).

Ако се върнем по-назад във времето, не може да не споменем, че през далечната 1960г. г-н Seymour Cray, електро инженер създава първият суперкомпютър в света. По това време той работи в Control Data Corporation, САЩ, чиято основна дейност е защита на информация. Новото в неговата машина, е че ползва множество процесори, за да раздели един типичен изчислителен проблем на отделни изчислителни задачи изпълнявани едновременно. Дефакто обаче първият реален прототип е пуснат през 1964г. наричан CDC 6600. Тази машина остава най-мощният супер компютър в света до 1969г. Първоначално машината е инсталирана в CERN, Женева, а по-късно в Бъркли, Калифорния, което го прави и първия супер компютър в САЩ. През 1976г. Cray 1 е вече на пазара, а през 1985г. е създаден Cray 2 с нечуваните до тогава 1.9 гигафлопса. Тази невероятна машина остава най-

бърза в света до 1990г. През 1993г. се появява японският Numerical Wind Tunnel създаден за симулация на флуиди. Машината притежава производителност от 100 гигафлопса и остава след най-бързите компютри до 1996г.

Основен проблем на ранните системи било оптимизирането на процесорното време. Програмите първоначално се „пишели“ чрез превключване на куплонзи, а по-късно и върху надупчени хартиени ленти, наричани перфо карти. Тези системи се развиват динамично до преди 1980г. След въвеждането на персоналните и офис компютрите ролята на супер компютрите започва да намалява. След 1990г. персоналните компютри започват да се използват за направата на WEB сървъри и това е повратна точка в тяхното развитие. Масовото им производство прави тези системи евтини, което позволява направата на клъстери от персонални компютри свързани по етернет. Естествено през този период супер компютрите продължават да заемат водещо място сред научните среди. Обаче този тип системи са прекалено скъпи и достъп до тях продължават да имат само учените и изследователите от големите университети. Цената на една такава система е огромна, което ги прави неприложими за конвенционални нужди. Интересно е да се знае, че от системата CM-1 има продадени едва стотина броя, а от MasPar около 200. Такъв тип компютърни системи са все още финансово неефективни за решения в съвременните университети, фирми занимаващи се с графична обработка, болнични заведения в отделенията за образна диагностика, академии и други и те са принципно немислими за индивидуални изследвания.

Суперкомпютрите по света

Ако сравним супер компютрите по света можем да отчетем че през 2009г. Московският държавен университет изгражда най-мощният супер компютър в цяла Източна

Европа с производителност от 350 терафлопса, имащ пиковата мощност от 414 терафлопса. (*T-G-K FLOP (Terra, Giga, Kilo FLoating Point Operations Per Second, операции с плаваща запетайка в секунда) Като вторият по мощност в цял свят е изграден в Китай и притежава 1206 терафлопса, сравнен с "Ягуар", който дефакто е най-мощният през 2009г. създаден от Cray Inc. имащ колосалните 224162 микропроцесора и производителност от 1759 терафлопа, с пиковата производителност 2331 терафлопа. През 2011г. първоначално за най-мощен е определен Tianhe-1A с нечуваните 2.6 петафлопса намиращ се в Tianjin, Китай последван по-късно същата година от Fujitsu K computer с пикова производителност от 10.51 петафлопса, намиращ се в Кобе, Япония. През 2012г. е пуснат супер компютъра Sequoia, интегриращ IBM BlueGene/Q с производителност от 16.32 петафлопсаon. Машината е изградена от 1,572,864 ядра. Sequoia позволява на САЩ за пръв път от 2009г. отново да се нареди на първо място в света на супер компютрите. Подробна обновяваща се статистика може да видите тук http://top500.org/

От своя страна българския суперкомпютър IBM Blue Gene/P [15], внесен през септември 2008г. е с Linux - базирана платформа и има 8192 микропроцесора, предоставящ невижданите до момента 23 терафлопса. По отношение на използваните операционни системи може да спомена, че до 2002г. доминиращ на пазара остават UNIX базираните системи, като след това до 2009г. делът на UNIX остава едва 8-12%, споделяни с Windows и BSD, като процентите продължава да падат. Почти всички останали нови системи (88% от пазара) са заети от Linux базирани системи. Ето защо и тази книга е посветена на програмирането на CUDA за Linux Ubuntu. Това е напълно обяснимо поради отворените стандарти и ниска себестойност, възможност за бърза модификация, добра документация и липса на огра-

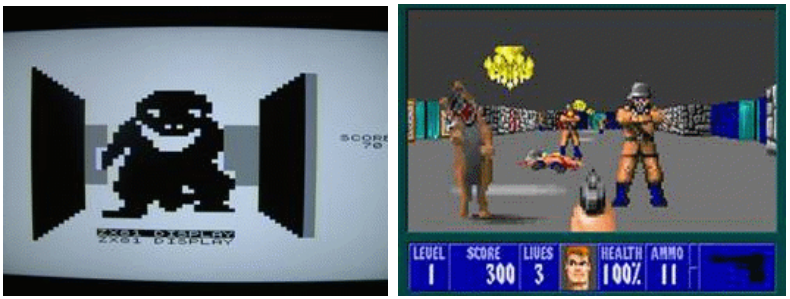
ничения за използване и промяна, както и скалируемост по отношение на добавяне на поддръжка за нови процесори и дискови масиви. Все пак, когато имате 100 процесора и ви се наложи да закупите лицензии за всички тях, цената на операционната система формира основен дял във вашия бюджет. Дори е възможно да се доближи до цената на вашия клъстер. Поради тези причини днес доминиращ дял в света на супер компютрите заемат Linux базираните операционни системи.

Да започнем от игрите

Тъй като този учебник е посветен на CUDA, а това е технология произлязла от „несериозните“ приложения, като компютърните игри и тримерна графика, ще разкажем малко повече за тях. През 90те години се наблюдава развитието на разпределените многопроцесорни архитектури, което практически представлява множество компютри свързани и работещи в една високоскоростна етернет (оптична) мрежа. Ефективността на този тип системи е висока, но въпреки стократно по-ниските цени от класическите супер компютри, тези системи продължават да имат прекалено висок експлоатационен разход на електроенергия, климатизация и пространство, а също така те не са особено добре приложими за провеждане на обемни еднотипни изчисления тъй като латенцията за подаване и зареждане на отделните под програми и данни в тях е доста голяма (1-10 секунди до минути в зависимост от обема данни). За осигуряване на максимална достъпност на научните организации до голям изчислителен ресурс широко разпространение получи доброволната програма BOINC (Berkeley Open Infrastructure for Network Computing), с помощта на която даден научен проблем се разпределя на отделни малки задачи и посредством интернет те се стартират на хиляди потребителски компютри. Някои от прог-

рамите които имат най-висока популярност са свързани със симулации на климатични процеси, изследване на земетръсната активност, генетика, нано технологии, симулация на флуиди, търсене на извънземен разум и финансови приложения. С развитието на персоналните компютри след 1990г. те все по-често стават повече средство за забавление, отколкото работен инструмент. Интересно е да се види как това на пръв поглед странично приложение на компютрите позволява агрегирането на нов пазарен сегмент(създаването на пазарно търсене на продукти и услуги, които не са атрактивни или са прекалено скъпи за потребителите към момента на въвеждането им) за специализиран хардуер, основно в областта на тримерната графична обработка. Първата 3D игра излязла през 1981г. наричана „3D Monster Maze“ се счита за основоположник на нова ера в компютърните забавления Фигура 1.1. Естествено, днес този екранен изглед буди само усмивки и умиление.

Фигура 1.1: Скриншот от **3D Monster Maze** - първата 3D игра за персонален компютър, и **Wolfenstein 3D**



През 80-те години персоналните компютри не предоставят високо бързодействие нито пък имат удобен потребителски интерфейс и добра графика, което налага изчакване от около 10 години до масовото навлизане на 3D игрите. Играта, която променя света е Wolfenstein 3D. Тя излиза на пазара през май 1992г. и заема колосалните за то-

гава 1.44MB флопи дисково пространство. Това я прави уникално лесно разпространима, тъй като по това време компютърните програми се пренасяха чрез дискети с този обем. И макар тази история да няма пряко отношение към научните изследвания, именно игрите създават предпоставка за разработката на нов тип видеокарти, наричани 3D графични ускорители. В началото на 90-те години масово ползвани са компютрите базирани на историческия микропроцесор - Intel 386 и в последствие Intel 486, както и Motorola 68000. Техните графични карти не позволяват интегрирането на 3D графични възможности. Този тип приложения са прекалено бавни тъй като изчисленията за изобразяване на графиката отнемат по-голяма част от изчислителното време на централния процесор. Първата графична карта имаща хардуер за 3D графичен ускорител е Cirrus Logic Laguna 3D, обаче за начало на новата ера може да кажем появата на 3Dfx Voodoo през 1996г. Тя се счита за първият 3D графичен ускорител имащ невероятен пазарен успех сред геймърите [5]. От този момент светът на персоналните компютри се променя. Най мощните персонални компютри започват да се създават именно и заради геймърите. Много скоро производителите на друг тип софтуер предимно за проектиране и архитектура започват да ползват новите възможности на графичните 3D видеоускорители. Пазарният сегмент расте стремглаво, скоро 3D ускорителите, първоначално предлагани като допълнителна PCI платка, започват да се вграждат в стандартните видео карти, а класическите софтуерни приложения за графична обработка и 3D симулации, като 3D Studio Max започват да използват новия хардуер. 3D Studio Max - програмен продукт за триизмерно анимиране и моделиране, е пуснат с това търговско име за Windows NT, но първоначално е произвеждан за MS-DOS с името Autodesk 3D Studio през 1990г. Новият хардуер е многократно по-евтин от ползваните до момента специализирани графични работни станции

на Silicon Graphics. Тази компания произвежда специализирани компютърни системи предназначени за анимиране и 3D компютърна графика. Едновременно с това развитие на графичните карти следва да отбележим модификациите на системните шини за свързване на видео картите към дънните платки на персоналните компютри Таблица 1.1. Също така и възможностите за ползването на видео паметта, която е вградена във видеоплатките. Тези нововъведения способстват за възможността централния процесор да обменя бързо големи масиви данни с видео платката, като по този начин освобождава системен ресурс за потребителските програми. Друго важно преимущество на този тип архитектура е възможността видеокартата да обменя автономно данни с определена част от паметта на компютъра. В този случай се извършва така нареченото заключване на кешираната странична оперативна памет на системата, което става чрез специални функции предвидени в повечето управляващи програми на болшинството видео карти, независимо от техния производител. Това дефакто отделя централния процесор от процеса на копиране на данни касаещи визуализацията на графични обекти на екрана. Това позволява на централния процесор да отделя повече време за други из изчисления и работа на операционната система.

Основното нововъведение в 3D графичните карти се явява появата на специализирани процесори за обработка на 3D графика, т.н. GPU (Graphics Processing Unit). Като водещи в комерсиалния пазарен сегмент може да се определят компаниите ATI (закупена през 1996г. от AMD) и компанията NVIDIA. Но хардуерът сам по себе си не допринася особено за масовото му ползване в среда MS-DOS. През 1994г. Microsoft представя Windows 95. До този момент производителите на игри считат MS-DOS за по-

Таблица 1.1: Еволюция на системните шини за допълнителна компютърна периферия

Шина	**Едновременно предавани битове**	**Скорост (MB/s)**	**Протокол**
PCI	32/64	132/800	паралелен
AGP	1x32	264	паралелен
AGP	2x32	528	паралелен
AGP	4x32	1000	паралелен
AGP	8x32	2000	паралелен
PCIe x1	1	250/500	сериен
PCIe x4	1x4	1000/2000	сериен
PCIe x8	1x8	2000/4000	сериен
PCIe x16	1x16	4000/8000	сериен
PCIe x16	2.0 1x16	8000/16000	сериен
IDE	(ATA100)	100 MB/s	
IDE	(ATA133)	133 MB/s	
SATA		150MB/s	
Gigabit Ethernet		125 MB/s	
IEEE1394B Firewire		100 MB/s	

удобна система при игрите. Това което до този момент отличава DOS, като операционна система е възможността за директен достъп до хардуера, който е скрит в Windows. В Windows 95 този проблем е решен, като програмистите на игри вече разполагат със стандартен интерфейс за изчертаване на видео изображения и достъп до клавиатурата, мишката и джойстика. Друг проблем остава ползването на системните таймери, които не разрешава на игро производителите добро времеделене на операциите и довеждат до съществено влошаване на бързината на игрите при Windows 3.1 и 95. Дори при Windows 95 производителите на 3D и 2D игри продължават да използват т.н. MS DOS режим на работа. За решаването на този проблем през 1995г. Microsoft създават DirectX, известна още като Windows Games SDK. Към настоящия момент наличната на пазара версия е DirectX 11. От своя страна функциите на средата дават на програмистите възможност за ползване в реално време на системните ресурси: графична карта,

джойстик, клавиатура и мишка, а също така и въвежда т.н. мултимедийни файлови формати и прецизни таймери (прецизните таймери все още са особеност, която само операционната система Windows позволява да се ползва от приложните програмисти на Direc X приложения, дори в Linux обикновенните дистрибуции тези функционалности са трудно използваеми в игрите). Всички тези нововъведения правят възможно лесното приложно ползване на новия хардуер, без това да налага програмистите на игри да пренаписват драйверите за всеки нов хардуер. До този момент всяка игра писана за MS-DOS ползва собствен видеодрайвер и драйвер за звуковата карта на компютъра. Direct X разширява броят поддържани видео карти и 3D ускорители имащи драйвери за Windows. До този момент операционната система Linux не е много популярна, а и интернет е едва във своята зора. Един от основните проблеми на Linux по това време беше свързан с разпознаването на графичните видео карти. Следва да се отбележи, че в началото на 90-те години Silicon Graphics промотира IRIS GL (Integrated Raster Imaging System Graphics Library), която е графична система за работни станции. Потребителските програмни функции поддържат създаване на графичен интерфейс, като прозорци, вход от клавиатура и мишка. Тази среда се появява преди X Window System и е ориентирана предимно към специализиран хардуер. Поради ограниченията на лиценза на IRIS през 1992г. Silicon Graphics създават OpenGL 1.0 (Open Graphics Library), който е крос платформен интерфейс с независима поддръжка на драйверите на видео картите и входно изходните устройства. Силното развитието на графичния хардуер, както и потребителските функции за 3D графика Direct X и OpenGL довеждат до идеята за това графичните карти да се използват за общо приложими математически изчисления, основно за обработка на изображения. За това свидетелства появата на вградени възможности във видео картите за обработка на MPEG 2

компресирано видео (във видеоплатката ATI - Rage Series, и видеоплатката на NVIDIA - GeForce). В момента на поява на тези продукти наличните 32 битови процесорни системи (Intel, AMD, Cirix) работят на честоти до 260-300MHz, което ги прави неприложими за компресията и обработката на цифрово видео в реално време. Появата на тази нова възможност в графичните карти и интегрирането и с Direct X и OpenGL значително променя погледа на програмистите занимаващи се с изчислителни проблеми към този тип хардуер. Следва да отбележим, че идеята за ползване на графичните процесори, като основа на математически системи за паралелна обработка датира още от 1978г. Тогава компанията Ikonas Graphics Systems проектира растерен дисплей за оборудване на пилотски кабини по поръчка на NASA Langley Research Center. Това е последвано от Pixel Machine – 1989г. и Pixel-Planes 5 – 1992г.. Бумът в развитието на графичния хардуер позволява съществено изменение на начина, по който тези системи могат да се ползват за общо приложими математически изчислителни задачи. Тези разработки довеждат до общото развитие на концепцията GP-GPU (General-Purpose Computation Using Graphics Hardware http://www.gpgpu.org/). Тази концепция позволява бързото решаване на множество сложни последователни алгоритми подлежащи на паралелизация, каквито са обработката и филтрацията на сигналите, компресирането на видео, криптографски анализ, статистически изчисления и други алгоритми обработващи днотипно големи масиви от данни. Разбира се за нуждите на сигналната обработка водещи производители на чипове, години наред, модифицират и създават специални сигнални процесори (DSP), чието внедряване е предимно в специализирани разработки. В комерсиалния сегмент DSP заемат място след средата на 90-те години предимно при направа на цифрови видеокамери, видеоконферентни системи и различни научни области. Тези платформи са трудни за усвояване от

обикновенните компютърни програмисти, при тях отсъства универсалност, и дори днес все още остават обект на внедряване предимно във вградени приложения с висока крайна цена. Програмирането и проектирането на DSP базирани системи не притежава гъвкавостта на програмирането за персонални компютри. Възможностите предлагани от съвременните GPU позволява на всеки средно статистически потребител и програмист на потребителски софтуер да има достъп до високопроизводителни мултипроцесорни разширения с ниска себестойност. При това цените на подобни платки варират между 60-3000 в зависимост от броят мултипроцесори (16-480 и повече) и количеството вградена видео памет (256MB – 4GB). Тези системи се развиват до такава степен, че днес позволяват постигането на пикова мощност от до 1 терафлопс в една графична видео карта. Съвременните многоядрени Intel и AMD процесори също поддържат подобни пикови изчислителни мощности, обаче методът на работа на централния процесор съществено се различава от работата на графичния процесор, чиято основна задача е обработката на пиксели. Също така върху централния процесор работят редица служебни под програми от операционната система, като извикването на отделни подпрограми и превключването мужду отделните процеси изпълнявани в операционната система отнема значително време. За разлика от тези системи GPU е специално проектиран за паралелна обработка на еднотипни изчислителни процеси върху строго определени области от данни фиксирани постоянно във видео паметта на графичните адаптери. Работата на GPU не се прекъсва от работата на операционната система, една функция стартирана на GPU не бива прекъсвана от други функции до пълното приключване на всички стартирани процеси, GPU притежава възможност за директно извеждане на получените резултати под формата на 2D, 3D графика върху екрана без при това да се налага прекъсване и допълнителна ра-

Таблица 1.2: Сравнение на любителски и геймърски видеокарти на NVIDIA поддържащи CUDA

GeForce	**Core Clock**	**MHz Memory Clock**	**Memory Interface**	**Memory Transfer Rate**
210	589/1,402	1 GHz	64-bit	8 GB/s
GT 220	625/1,360	1.58 GHz	128-bit	25.28 GB/s
GTS 250	738/1,836	2.2 GHz	256-bit	70.4 GB/s
GTX 280	602/1,296	2.21 GHz	512-bit	141.7 GB/s
GTS 450	783/1,566	3.6 GHz	128-bit	57.7 GB/s
GTX 480	700/1,401	3,696 MHz	384-bit	177.4 GB/s
GT 520	810/1,620	1.8 GHz	64-bit	14.4 GB/s
GTX 590	607/1,215	3,414 MHz x2	384-bit x2	163.9 GB/s x2

бота на централния процесор. Освен това CPU са предвидени за символни изчисления в многопотребителски и многопроцесен режим, докато GPU са предвидени основно за обработка на пиксели (отделни данни) чрез идентични изчислителни процеси. Естествено има редица задачи чиято реализация върху GPU процесор не носи съществено повишаване на бързодействието на програмата, а често дори я влошава. При всички случаи обаче може да очаквате, че всеки изчислителен проблем, който може да сведете до аналогична графична изчислителна задача, където данните са пиксели може и ще бъде по-добре изпълним върху GPU отколкото върху CPU. За да добиете по-ясна представа за развитието на технологията при картите на NVIDIA вижте по-долната сравнителна Табл. 1.3. Може да се различат две основни продуктови линии Quadro - специализирана за професионални графични приложения и GeForce - предназначена основно за крайния потребител и геймърите. Следва да се отбележи, че персоналният университетски супер компютър FASTRA II е базиран изцяло на комерсиална технология от 6x двуядрени GTX295 и 1x едноядрена GTX275.

Освен поддържащи CUDA графични видео карти, съществуват и специализирани PCI Express карти, съдържащи само GPU процесори и по-големи обеми RAM. Тези карти са подходящи за вграждане, както в стандартни PC така и в шкафове. Характерно за CUDA е необходимостта в истемата да има поне един реален графичен адаптер с видео изход. Това значи, че ако предвиждате да разработвате система с карта Tesla ще ви се наложи, като минимум притежанието на допълнителна GeForce карта с минимални графични възможности. По-долу е дадено сравнение на TESLA продуктовата гама на NVIDIA предназначена за създаване на мощнш персонални изчислителни системи (Табл. 4). Следва да се обърне внимание, че системите монтирани в шкафове може да бъдат разширявани до стотици терафлопса. Обърнете внимание, че това са пикови мощности и са достижими само за някои алгоритми при условие, че данните са заредени в глобалната вградена памет на графичните процесори. Характерно за GPU супер компютрите, това са клъстери от GPU процесори със собствена памет, е основно силно пониженатa им пикова мощност. Например сравнението на 512 ядрен клъстер със супер компютрите Fastra I и II [вж.**University of Antwerp builds desktop supercomputer with 13 NVIDIA GPUs, от T. De Maesschalck**] показва между 7 до 20 кратно повишаване на енергийната ефективност при ползването на CUDA. Това означава, че изграждането на CUDA клъстери е не само изгодно по отношение на първоначалната инвестиция, но и е препоръчително по отношение на текущия разход за електричество и климатизация на помещенията (5-20 пъти по-нисък в сравнение с ползването на стандартни CPU базирани многоядрени супер компютри).

Продуктовата линия на TESLA е подходяща за изграждане на дестоп супер компютри и вграждане в шкафове.

Таблица 1.3: Сравнение на професионални видеокарти на NVIDIA поддържащи CUDA

модел	**код на модела**	**шина**	**памет MB**	**трансфер на данните GB/s**	**брой изобразени пиксели (M/s)**
Quadro	NV10GL	AGP 4x	64	2.66	480
Quadro4 380XGL	NV18GL	AGP 8x	128	8.2	1100
Quadro FX1400	NV41	PCIe x16	128	19.2	4200
Quadro FX4500X2	G70	PCIe x16	1024	33.6	11280
Quadro FX56002	G80	PCIe 2.0 x16	1536	76.8	38400
Quadro FX5800	G100GL-U	PCIe 2.0 x16	4096	102	52000

Таблица 1.4: Сравнение на модели графични карти на NVIDIA и броят мултипроцесорни и памет в тях

Конфигурация	**Код на модела**	**Архитектура**	**Брой процесори**	**Памет MB**	**Трансфер GB/s**	**GFLOPS**
GPU Processor	C870	G80	128	1536	77	519
Deskside Supercomputer	D870	G80	256	2x1536	154	1037
GPU Server	S870	G80	512	4x1536	307	2074
GPU Processor	C1060	G200	240	4096	102	936
GPU Server	S1070	G200	960	4x4096	410	4320
GPU Server	S1075	G200	960	4x4096	410	4320

Типично приложение е ползването на картата C1060 за изграждане на до 3.8 терафлопса в единично PC. Някои от моделите предлагани Фигура 2.3 на пазара, са поддържани от Dell и ASUS. Характерно за изграждането на подобна система е необходимостта от високопроизводителни шини PCI Express, а що се отнася до избора на централен процесор е добре това да бъде многоядрен Intel с възможно по-голям кеш (4-8MB). По-големият кеш предполага по-малко латенция при обръщението към паметта на системата, оставяйки повече време за трансфер на данни от твърдите дискове към оперативната памет и респективно паметта на видео адаптерите.

Фигура 1.2: Типове продукти TESLA, за вграждане в стандартно PC – C1060 и за монтаж в шкафове Tesla S1070 GPU сървър

Следва да споменем, че най-разпорстранените гейм конзолите от типа: Xbox, Xbox 360, Deramcast, Wii, PS2 и PS3 позволяват създаването на персонални супер компютри. По-долу е дадена сравнителна таблица за броят продадени геймърски конзоли към 2009г., които не са PC базирани, както и изчислителната им мощност. Тези конзоли се програмират по-трудно и не са така удобни като продуктите на GeForce и TESLA, при тях липсват много от възможностите предоставяни от класическите PC, тъй като те са предназначени за крайни потребители на игри, а не програмисти. Следва да се отбележи че след 2010г. Масово се

Таблица 1.5: Сравнение на изчислителната мощност на комерсиални гейм конзоли – видео игри към началото на 2009г.

Конзола	**Скорост на процесора в гигафлопса**	**Общо продадени терафлопса**	**Брой продадени конзоли (милиони)**
Xbox	5.8	139200	24
Xbox 360	115.2	2188800	19
Dreamcast	1.4	8400	6
Wii	2.9	75400	26
PS2	6.3	768800	124
PS3	218	283400	13
Общо	349.6	3464000	212

предлагат и редица други мобилни игрови устройства имащи вградени възможности с CUDA съвместим хардуер.

Какво променя CUDA?

След като добихте обща идея за световните тенденции и развитието на супер компютрите ще разгледаме малко повече подробности от появата на CUDA. На 11 октомври 1999г. Е пусната на пазара видео картата GeForce 256. Тя се счита за родоначалник на GP-GPU отворената архитектура, като в този модел за първи път се предвижда възможност програмистът да контролира директно графичнвия процесор, като така може да го оплзва за изчисления с общо предназначение. Видео картите от серия GeForce 3 анонсирани през 2001г. Са първите поддържащи Microsoft DirectX 8.0. Този стандарт изисква изчисленията на пикселните шейдъри и вертексите да става в хардуера на видео адаптера. Шейдърите (pixel shader) се ползват в графичните карти за изчисляването на цвета и други атрибути на всеки един изобразяван на екрана пиксел. Вертексите (vertex) се ползват за описание на точките, които са обик-

новенно върховете на трийгълни равнини ползвани за изрисуване на триизмерни обекти намиращи се в 3D пространството, върху 2D равнина на монитора, която потребителят гледа. В процесът на работа гледната точка и взаимното положение на моделите на обектите в 3D пространството се мени, което налага ново и ново преизчисляване на изходния 2D екранен образ при всяка една промяна. Този процес изисква интензивни математически изчисления и именно поради нарастващите потребности на потребителите да играят игри с по-висока резолюция и динамика налага това изчисление да става в специализираните графични процесори на видео картите. През 2006г. NVIDIA анонсира първата си видео карта - GeForce8800 GTX, която е във всъщност първата видео карта с обособени GPU процесори. Тази нова карта използва и няколко нови хардуерни компонента специфични за CUDA, които именно ни позволяват да програмираме графичните мултипроцесори за стандартни математически изчисления без да се налага познаването на системите DirectX и OpenGL. След това нововъведение CUDA вече е дъстъпна за всички инженери и учени, които могат да пишат C програми, но нямат опит в работата с тримерна графика. Наклоко месеца след анонсирането на GTX 8800 NVIDIA пуска и първият CUDA C компилатор. Заедно с него NVIDIA предлага и специализиран драйвер за видео картата, който позволява ползването на CUDA архитектурата от програмисти без опит в 3D графиката. Днес с GPU технологията на NVIDIA в професионалните видеокарти имате възможност да създадете персонален супер комюптър с пикова производителност от 12 терафлопса и повече. Разглеждайки различните модели разпределени изчислителни системи следва да се спрем на един уникален проект наричан FASTRA I и II създадени във Vision Labs към университета в Антверпен, Белгия (http://fastra2.ua.ac.be/). Втората версия на системата е резултат от взаимодействието между студенти,

Tones.be и ASUS, за създаването на най-мощншя десктоп супер компютър в света притежаващ 13 GPU и имащ пикова производителност от 11 терафлопса на цена от 6000 евро. Системата е използвана при реализацията на софтуер за реконструкция на медицински MRI (изображения от ядрено магнитен резонанс). Типично този тип приложения изискват много време за реконструкция на изображенията, което е в порядъка на 3 до 12 часа за конвенционалните процесори. Тъй като болшинството болници не разполагат с изчислителен ресурс от повече от 1-2 терафлопса общо, процесът на диагностика отнема значително време. Обаче използването на CUDA във FASTRA II показва как тази технология може да се ползва за масово въвеждане дори при ниски бюджети практически навсякъде.

Видео карти поддържащи CUDA

Видео картите от серията GeForce са най-достъпни, като цени за масовия потребител. Ако не пишете специализиран научен софтуер, който ще се изпълнява на карти Tesla ползването на GeForce е достатъчно. Освен това комерсиалният софтуер написан за тази серия видеокарти ще работи на почти всички машини имащи подобна видео карта. Индексите x.x изписвани зад модела на всеки графичен поцесор показват какви са специфичните изчислителни възможности на определаната архитектура (Compute Capability) на видеокартата. В това число по-новите версии поддържат т.н. атомарни операции, които ви гарантират, че когато даден процес адресира определен адрес от глобалната или локална памет тя няма да бъде достъпна за другите процеси преди неговото приключване. Също така различните версии на хардуера позволяват ползването на изчислителни операции с т.н. двойна прецизност над различни операнди. Това в известна степен забавя работата на вашите програми, но гарантира, че данните които желаете

да обработите с по-висока прецизност ще бъдат пресметнати правилно. Различните версии поддържат и различни методи за поточно изпълнение на програмите, графична съвместимост с OpenGL и Direct X. При покупка на графична карта е важно да изберете възможно по-висока версия на хардуера, това ще ви гарантира достъп до различни улесняващи програмирането опции и команди, недостъпни за по-старите версии на графичните процесори. Това е особено важно за обемисти инженерни и научни изчислителни задачи, при които получаването на високо прецизни резултати от симулациите е важно условие за ползването на GPU вместо класически клъстерни CPU базирани паралелни системи за обработка на информация.

Подробен актуален списък на продуктите наNVIDIA поддържащи CUDA може да видите от сайта им:
http://developer.nvidia.com/cuda-gpus.

Компютърни системи поддръжащи CUDA

CUDA се поддръжа вече от всички видеокарти предлагани на пазара с NVIDIA чипсет. Същественият проблем при реализация на изчислителна платформа е наличието на компютърна конфигурация имаща следните ключови параметри: наличие на мощно захранване (над 450W до 1KW), наличие на достатъчно RAM (поне 1 GB без горен лимит, до 4GB за 32 битовите операционни системи).

По-големият RAM ще ви позволи да алокирате и заключите големи масиви данни в основната памет без това да налага на операционната система да кешира тези данни на твърдия диск, това ще увеличи съществено скоростта на трансфер на данните от хоста във видео картата и обратно. Наличието на добро охлаждане и допълнителни вентилатори в кутията е задължително. Купете най-нискошумящите вентилатори, когато броят им е голям шумът е непоносим. Наличие на филтри за засмуквания въздух от вентилато-

Таблица 1.6: Видео карти GeForce поддържащи CUDA и версията на изчислителната архитектура

Модели			
GTX 560 Ti 2.1	GTX 550 Ti 2.1	GTX 460 2.1	GTS 450 2.1
GTS 450* 2.1	GTX 590 2.0	GTX 580 2.0	GTX 570 2.0
GTX 480 2.0	GTX 470 2.0	GTX 465 2.0	GTX 295 1.3
GTX 285 1.3	GTX 285Mac1.3	GTX 280 1.3	GTX 275 1.3
GTX 260 1.3	GT 520 2.1	GT 440 2.1	GT 440* 2.1
GT 430 2.1	GT 430* 2.1	GT 420* 1.0	GT 240 1.2
GT 220* 1.2	210* 1.2	GTS 250 1.1	GTS 150 1.1
GT 130* 1.1	GT 120* 1.1	G100* 1.1	9800 GX2 1.1
9800 GTX+ 1.1	9800 GTX 1.1	9600 GSO 1.1	9500 GT 1.1
8800 GTS 1.1	8800 GT 1.1	8800 GS 1.1	8600 GTS 1.1
8600 GT 1.1	8500 GT 1.1	8400 GS 1.1	9400 mGPU 1.1
9300 mGPU 1.1	8300 mGPU 1.1	8200 mGPU 1.1	8100 mGPU 1.1
8800 Ultra 1.0	8800 GTX 1.0	GT 340* 1.0	GT 330* 1.0
GT 320* 1.0	315* 1.0	310* 1.0	9800 GT 1.0
9600 GT 1.0	9400GT 1.0	GT 635M 2.1	GT 630M 2.1
610M 2.1	GTX 580M 2.1	GTX 570M 2.1	GTX 560M 2.1
GT 555M 2.1	GT 550M 2.1	GT 540M 2.1	GT 525M 2.1
GT 520MX 2.1	GT 520M 2.1	GTX 485M 2.1	GTX 470M 2.1
GTX 460M 2.1	GT 445M 2.1	GT 435M 2.1	GT 420M 2.1
GT 415M 2.1	GTX 480M 2.0	GTS 360M 1.2	GTS 350M 1.2
GT 335M 1.2	GT 330M 1.2	GT 325M 1.2	GT 240M 1.2
G210M 1.2	310M 1.2	305M 1.2	GTX 285M 1.1
GTX 280M 1.1	GTX 260M 1.1	9800M GTX 1.1	8800M GTX 1.1
GTS 260M 1.1	GTS 250M 1.1	9800M GT 1.1	9600M GT 1.1
8800M GTS 1.1	9800M GTS 1.1	GT 230M 1.1	9700M GT 1.1
9650M GS 1.1	9700M GT 1.1	9650M GS 1.1	9600M GT 1.1
9600M GS 1.1	9500M GS 1.1	8700M GT 1.1	8600M GT 1.1
8600M GS 1.1	9500M G 1.1	9300M G 1.1	8400M GS 1.1
G210M 1.1	G110M 1.1	9300M GS 1.1	9200M GS 1.1
9100M G 1.1	8400M GT 1.1	G105M 1.1	

рите (ако ще монтирате кутията на пода или предвиждате да работи в лоши условия без перманетно обслужване това е задължително), наличие на добър Intel процесор и бърз твърд диск. При избора на процесор (при наличие на поне 4GB RAM) следва да обърнете особено внимание той да има възможно по-голям кеш. Големият кеш ви гарантира, че вашите основни програми ще продължат да работят достатъчно бързо, когато обменяте големи масиви данни между компютърната памет и паметта на видеокартата (4-8MB е препоръчително). По отношение на дънната платка следва да обърнете внимание тя да има поне 1 пълноскоростен PCI-Ейзиеяя интерфейс 16x, ако желаете да направите платформа с повече видеокарти обърнете внимание отново на захранването и на избора на дънна платка, не винаги дъната с два експрес слота поддържат 16x режим към двата слота при поставени 2 видеокарти. Препоръката ми е ако се двоумите дали да направите система с 2 или повече карти при условие, че можете да купите система с 1 мощна карта да закупите системата с по-мощна видеокарта. Ако се ориентирате към професионалната серия Tesla, имайте предвид, че ползването и изисква във вашата система да има инсталирана минимум още една видеократа с видео изход (това налага ползването на дъно с 2 PCI Express слота)!

Таблица 1.7: Мобилни и професионални карти поддържащи CUDA и версията на изчислителната архитектура

Мобилни			
NVS 4200M 2.1	NVS 5100M 1.2	NVS 3100M 1.2	NVS 2100M 1.2
Tesla			
C2075 2.0	C2050/C2070 2.0	C1060 1.3	C870 1.0
D870 1.0	M2050/M2070/ M2075/M2090 2.0	S1070 1.3	M1060 1.3
S870 1.0			

Таблица 1.8: Възможности на различните версии изчислителни архитектури на CUDA

Функционалност	**1.0**	**1.1**	**1.2**	**1.3**	**2.X**
32 bit целочислени атомарни операции в глобалната памет	-	ДА	ДА	ДА	ДА
32 bit целочислени атомарни операции в споделената памет	-	-	ДА	ДА	ДА
64 bit целочислени атомарни операции в глобалната памет	-	-	ДА	ДА	ДА
Нецелочислена аритметика с двойна прецизност	-	-	-	ДА	ДА
64 bit целочислени атомарни операции в споделената памет	-	-	-	-	ДА
32 bit нецелочислени атомарни операции в споделената и глобалната памет	-	-	-	-	X
3D изчислителна решетка с процеси	-	-	-	-	X
Максимален брой процеси в един блок	512	512	512	512	1024
Обем локална памет на процес	16KB	16KB	16KB	16KB	512KB

Глава 2

CUDA - Compute Unified Device Architecture

В тази глава ще научите малко повече за технологията CUDA, и по-точно как CUDA разпределя ресурсите на паметта на видеокартата, как CUDA разпределя една изчислителна задача на изчислителни клъстери и отделни изчислителни процеси. Ще добиете обща представа за възможностите на технологията и възможното и развитие в близко бъдеще.

2.1 Хардуерна архитектура

Технологията CUDA се счита за основоположник на либерализацията на "супер компютинга", често е наричана демократична технология, именно защото дава равен достъп на всички инженери и учени до равноправен евтин и качествен изчислителен ресурс. Чрез CUDA съвместимите продукти всеки програмист или учен има достъп до огромен изчислителен ресурс в рамките на стандартните бюджети за закупуване на съвременно PC, като цените на полу про-

фесионалните продукти започват от 250$до 1300$за обикновенни офис платформи. Не на последно място, CUDA показва колко е полезно да играете 3D компютърни игри и да правите всички онези безсмислени на пръв поглед неща, които рано или късно водят до нови технологии и нови приложения! Приложения и алгоритми чиято, проверка изискваше десетки дни, днес могат да бъдат изпълнени за часове или дори минути в зависимост от ползвания хардуер. Нещо повече, машини като Cray и IBM Blue Gene консумират много повече електроенергия и имат много повисоки текущи разходи по поддръжката отколкото един CUDA - Tesla суперкомпютър. Подобна персонална супермашина може да поставите на вашето офис бюро! Освен това не бива да пренебрегвате и чувството, което досегът с тази супер технология ще ви донесе, ежедневно, помагайки ви да решите бързо, качествено и прецизно вашите инженерни изчислителни проблеми. В краен случай ако не можете да програмирате върху тази супер технология винаги може да я използвате за игра на любимите си 3D игри! Нещо което не може да направите с един университетски клъстер. Бързодействието на програмите, писани за CUDA съвместими устройства зависи основно от ползвания хардуер, като с малко трикове програмите ви ще могат да работят на всички поддържащи технологията устройства [7]. Принципно архитектурата на GPU се различава от тази на CPU (Фиг. 4). Едновременно с това, когато през 2009г. започнах да използвам тази технология повечето програмисти дори не бяха чували за нея и възможностите и. Масовото схващане беше и остава, че трябва да пишат код за многоядрени паралени процесори или FPGA и DSP платформи. Никой от тези проекти обаче все още не постигна заслужено развитие, което е обосновано поради високите начални инвестиции необходими за създаване на реконфигурируем супер компютър с FPGA и DSP. CUDA е друга бира! Нещо съвсем различно, дефакто тя се преконфигу-

рира за секунди, достатъчно е да напишете нова програма и да я стартирате. При нея не е нежно да създавате нов хордуер, да създавате платки, да пишете на интересни екзотични езици, като VHDL, да компилирате тествате и конфигурирате електронните компоненти.

По информация на производителя съвременните GPU не са част от самия графичен процесор на видео картата. И двата чипа могат да използват общата видео памет, която е многократно по-бърза от стандартната RAM инсталирана на дънните платки на персоналните компютри. При GPU технологията се заделя сравнително малка част от кеша, наричан споделена памет (_shared_), в която може да се извършват обработки от множество АЛУ (аритметично логически устройства) едновременно. Друга особеност е че се ползва мнококанална видео памет наричана глобална (_global_). Един алгоритъм се разбива на под процеси, обединени в блокове, а те от своя страна в клъстери от блокове. В процеса на работа всеки един процес се изпълнява псевдо паралелно, като при това в зависимост от броят мултипроцесори се постига по-бързо изпълнение на всички тези процеси във времето. Характерно за CUDA е ползването на Single-Instruction Multiple-Thread (SIMT) архитектура, като при това данните записани в различни области от паметта могат да бъдат обработвани едновременно от множество АЛУ в няколко мултипроцесора едновременно. При стандартните процесори AMD и Intel голяма част от транзисторите в чипа се ползват за изграждане на локални кеш памети и по-сложна контролна логика на контролера на процесора. При CUDA има повече АЛУ, което означава, че повече еднотипни изчисления ще бъдат извършени върху даден масив от данни за определено време. Разбира се различни техники за оптимизацията на този процес могат да се използват в процеса на програмирането на системата, за да се постигне максимално високо бързодействие

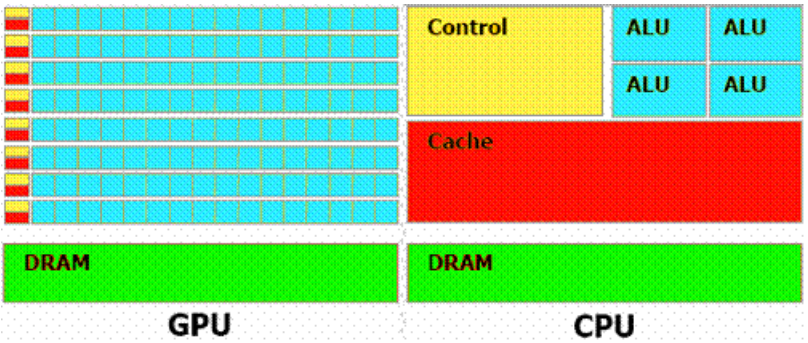
на програмите. Тук ще се спра над типовете компютърни архитектури, най-общата представа за тях ще ви даде повече знания за това кога и за какво можете да ги ползвате. Масово използваните днес процесорни архитектури са Харвардска и Фон Нойман. Първият модел, разпределя отделни блокове памет за съхранение на инструкциите и данните, а вторият тип позволяват на паметта да съдържа едновременно програмни инструкции и данни (Фиг. 4). Освен това процесорите могат да се различават по начина по който обработват инструкциите и данните, като съществуват основно четири модела:

- **SIMD** (Single Instruction, Multiple Data),
  SIMT (Single Instruction Multiple Threads) - CUDA
- **MISD** (Multiple Instruction, Single Data),
- **SISD** (Single Instruction, Single Data) и
- **MIMD** (Multiple Instruction stream, Multiple Data stream).

SIMD предполага използването на множество АЛУ работещи едновременно над съседни области памет с еднакви инструкции, каквито са съвременните DSP процесори. CUDA е модификация на SIMD наричана SIMT. В тази технология всеки процес обработва данните върху свои собствени регистри, докато при SIMD програмистът предварително трябва да пакетира данните.

MISD позволява обработката на една област памет едновременно от множество АЛУ. SISD се ползва при микроконтролери и микропроцесори от нисък клас, като при тях е налично едно АЛУ можещо да обработва една данна в даден момент от време, това са обикновенно миниатурни

Фигура 2.1: Архитеркурни особености на GPU и CPU, (изт. NVIDIA)

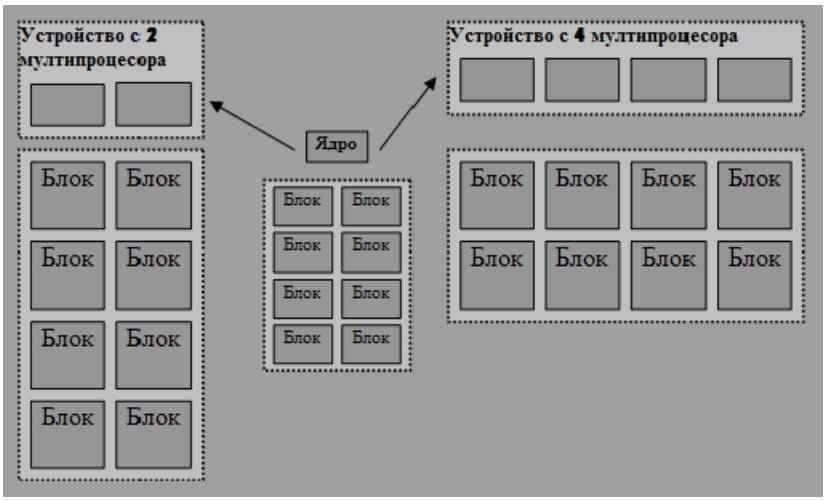


вградени приложения с ниска консумация на електроенергия. MIMD позволява мащабна асинхронна обработка на данни, при която всеки отделен процесор обработва данни намиращи се в различни области на паметта, каквито са съвременните много ядрените и многопроцесорни системи. Подобни системи са подходящи за разпределени офис и интернет приложения, уеб услуги, където всеки потребител работи над различно приложение, което обработва различни данни постъпващи в асинхронен режим. Спецификата на CUDA я доближава много до съвременните DSP процесори, като при това запазва гъвкавостта и възможностите за лесната промяна на програмната конфигурация характерна за съвременните персонални компютри. Характерна разлика обаче е ползването на паметта, при DSP за постигане на бърза сигнална обработка има специализирани преместващи регистри, които са скъпи и това ограничава практическото внедряване на подобни системи. Въпреки това в практиката при реализацията на комплексен изчислителен център често се ползват разнообразни сървърна технологии: CPU, DSP, FPGA и CUDA базирани. Тяхната цел е да позволят поточното въвеждане на данните, които ще се обработват и обработката им върху възможно най-

подходящата архитектура. Това ги оскъпява, а програмирането им е трудно и изисква ползването на различни софтуерни инструменти. Но основната разлика с DSP, е че при тях не може да правите съществени изменения в програмния код. Съществуват процесори за линейна обработка на данни и такива за нелинейна, тоест един DSP чип има строго специфична област на приложение. При тях постигането на по-високо бързодействие изисква добавяне на още чипове. От друга страна драйверите на CUDA разпределят автоматично отделните изчисления над пълния набор налични мултипроцесори в системата, което дава възможност една част групирани в блокове задачи да бъде осъществима на карта с по-малък брой мултипроцесори или такава с по-голям брой без да се налага програмистът да мисли за разпределянето на русурсите. Това е непосилна задача за класическите C програми работещи върху DSP. Новите версии на драйверите позволяват едно масивно изчисление да се разпредели едновременно върху няколко графични карти, и при това без програмистът да трябва да учи да ползва драйверът на видеократите. Това позволява скалируемост и мултиплатформеност на приложенията, например ако вашата програма работи върху мобилен телефон с CUDA съвместим хардуер тя ще консумира по-малко енергия за сметка на по-бавно изпълнение, а ако работи върху супер компютър ще се изпълнява по-бързо. И най-важното потребителят няма да е ангажиран с допълнителни действия освен да пусне своята програма. Друга особеност, която би ви накарала да изберете CUDA при реализация на комплексни програми за обработка на сигнали и изображения вместо DSP, е че вие можете да ползвате десектки готови математичсеки функции от CUDA SDK, с които да извършвате 1D, 2D и дори 3D фурие трансформации и матрични операции без да се налага да пишете специфичен софтуер оптимизиран за специфични DSP чипове. С две думи, ако вашето приложение не е вградено, няма съществени изиск-

вания към безотказността му, няма да се ползва в самолети, автомобили и военни съоръжения, непременно следва да изберете CUDA.

Фигура 2.2: Изпълнение на една и съща изчислителна задача заемаща 8 блока върху различни архитектури имащи 2 и 4 мултипроцесора отнема различно време, но резултатът ще бъде еднакъв.



В най-общия случай за да извършим някакви изчисления паралелно ще се стартират n на брой процеса. Данните и съответно инструкциите от тези процеси се обособяват в т.н. блокове, които от своя страна се обособяват в т.н. рамки (клъстери). По този начин може да се оптимизират ресурсите на видео картата, като глобална памет и брой мултипроцесори. Това позволява на драйвера да ползва различни хардуерни архитектури, зареждайки в тях повече или по-малко на брой блокове, респективно сумарен броя едновременно изпълнявани процеси. Това става прозрачно за програмиста и позволява ползването на хардуер с различни възможности без да се налага повторно компилиране на софтуера. При повечето случаи с цел оптимизиране

на броят обръщения към глобалната памет, а това са над 95% от реалните примери, се налага ползването на допълнителни механизми за синхронизация на процесите преди да се прехвърли управлението върху друга група процеси от следващ блок. Версиите на CUDA позволяват използването и на специфични операции, наричани атомарни. При тях например версия 1.1 на хардуера има вградени команди за извършване на изчисления над дадена клетка от глобалната памет. Версия 1.2 на хардуера позволява такива действие върху споделената памет във всеки един мултипроцесор. Тези действия разрешават постигането на високо бързодействие. Следва да се каже, че ползването на мултипроцесорите за обработка на елементи намиращи се в глобалната памет е многократно по-бавно, това налага ползването на споделена и константна памет в зависимост от ситуацията. Малки порции данни, респективно процеси се зареждат в споделената памет на всеки мултипроцесор. След като всички процеси в обработвания блок приключат системата приема за обработка друг блок. Данните от междинните резултати се съхраняват в споделената памет, като след приключване на всеки един блок те се записват в глобалната памет. Това позволява постигането на високо бързодействие, т.к. обръщенията към глобалната памет се намаляват. При това всеки мултипроцесор работи с локални копия на данните и микропрограмите. Архитектурата предоставя многоядрени процесори, обединени в т.н. мултипроцесорен блок, всеки от които има отделни АЛУ за изчисления на целочислени аргументи и отделно АЛУ за изчисления на числа с плаваща запетая. Не всички изчислителни версии на хардуера поддържат еднакъв тип глобални и локални атомарни операции над различни типове данни. Начинаещите програмисти следва да обърнат внимание на следните основни термини и тяхното значение:

Half-Warp – това е група от 16 процеса чакащи в опашката

за последователно изпълнение. Half-warp процесите се изпълняват едновременно. Например процесите от 0->15 ще се намират в един и същи под блок, 16->31 в друг и т.н., но е възможно по-нови версии на хардуера да поддържат друго деление.

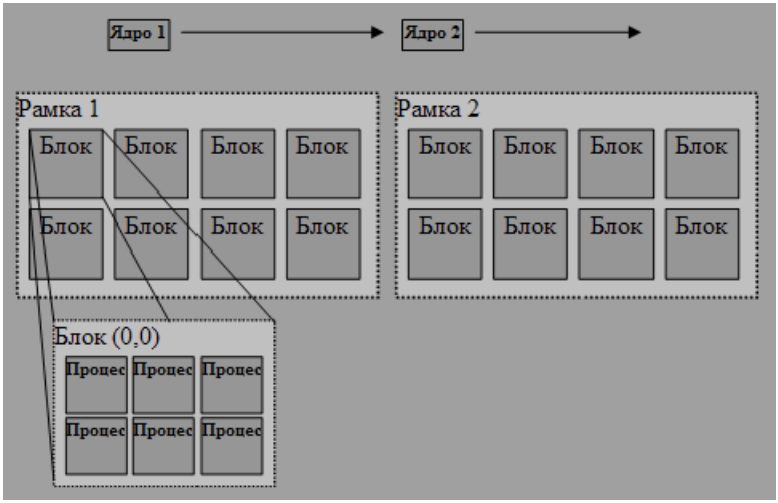
Warp – това е група от 32 съседни процеса, те принципно се изпълняват в паралел. Това налага специфични методи за синхронизация, целящи изчакване на завършването на всички процеси на дадена итерация от изчисленията преди да бъде приет следващия блок за обработка.

Block – блокът е набор от процеси, които правят едно и също нещо над различни елементи от масив с данни или едни и същи споделени данни. Поради технически причини минимум 192 процеса са нужни в един блок за оптимизация на закъсненията между тях. Един типичен блок има 256 максимум 512 процеса. Блокът може да има 1D, 2D и 3D структура, като сумарния брой процеси не бива да надвишава 512. Следва да се има предвид, че процесите в един блок се синхронизират по-бързо и така по-лесно обменят данни помежду си чрез споделената памет. Следва да се има предвид, че 8 конкурентни блока могат да се изпълняват на един мултипроцесор.

Grid – "решетката или рамка" позволява да се създават макро блокове (клъстери) от обособени процесни блокове. Отделните блокове се синхронизират трудно, процесите в един блок също не могат да се синхронизират с процесите принадлежащи на друг блок. Отделни "grid"масиви се генерират за всяка специфична изчислителна задача (наричана kernel - ядро), обхващащи определени данни, като при това всеки различен грид може да се създава за нови и вече записани в паметта на картата данни (фиг. 7). Димeнсиите на изчислителния клъстер са 1D, 2D и 3D, като максималния брой блокове в него не може да надвиши 65536 във всяка една посока.

Kernel – "ядро" това е глобална функция (__global__), която най-общо казано се изпълнява върху CUDA съвместимия хардуер, като тези функции могат да бъдат викани от основната програма или да се викат от други функции работещи върху CUDA. Ядрото обикновенно се свързва с даден набор функции, които имат за цел да извършат еднотипна обработка на всички данни записани в една изчислителна рамка. Ядрото може да вика функции за обработка на данни дефинирани за видими само в рамките на видеокартата (__device__).

Фигура 2.3: Разпределение на отделните процеси в блокове и ползване на различни рамки от процеси за различни изчислителни задачи.

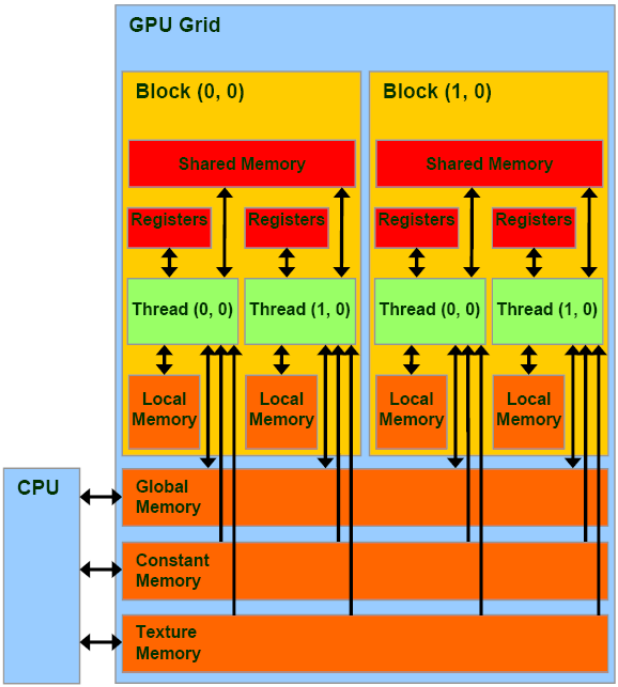


2.2 Типове памет

За CUDA програмистите от съществено значение да правят разлика мужду типовете памет интегрирани във видео

картата и съответно предимствата и недостатъците от нейното използване:

Фигура 2.4: Видове памети използвани в CUDA.



Глобална памет – това е инсталираната физическа памет на графичната карта, чийто обем може да варира между 128 и 4000 MB. Организацията ѝ е линейна, в тази памет, както отделните мултипроцесори с изпълняваните върху тях kernel, а също така и драйверите на видеокартата работещи на хост компютъра могат да пишат и четат данни. Всички процеси могат да пишат и четат данни в глобалната памет, също така процеси стартирани на хост компютъра могат да пишат и четат от и в тази памет чрез функциите предвидени от драйвера на видеокартата. Четенето и записа в нея са най-бързи при четене и запис на

данни в поредни адреси. За постигане на точни резултати се налага ползването на функции за синхронизация между процесите изпълнявани в отделни изчислителни блокове, така че да се гарантира, че даденият изчислителен блок ще приключи изпълнението на всички предвидени за изпълнение процеси преди да бъде зареден следващ изчислителен блок. Когато много процеси четат и записват данни в тази обща глобална памет програмата става бавна. Освен това когато два или повече процеса четат и записват данни от междинни изчисление в едни и същи адреси на глобалната памет се налага ползването на допълнителна синхронизация или т.н. атомарни операции, които ни гарантират, че никой следващ процес няма да получи монополен достъп до дадения адрес от глобалната памет преди предходния да приключи успешно. Най-високо бързодействие се постига при последователно четене и запис на 16 байтови блокове данни, имайте това предвид, когато реализирате kernel функции обработващи големи масиви данни. Понякога е удобно данните да се пакетират на 16 байтови групи, като самият kernel адресира отделните байтове по отделно, за да бъде изпълнена дадена изчислителна операция над тях.

Локална памет – това е памет в която може да чете и записва данни само един процес. Обикновенно се ползва за съхраняване на междинни резултати от изчисления.

Споделена памет – всеки един мултипроцесор има малък обем от памет наричан споделен, с размер 16KB. Тази памет се използва от отделните процеси в рамките на този блок за бързо четене и запис на данни. Важно е да се знае, че споделената памет в един блок е видима само и единствено за процесите вътре в него. Като пример ще кажем, че е възможно да имаме няколко блока работещи едновременно върху един и същи мултипроцесор. Тази памет е регистрова и е в пъти по-бърза от глобалната памет на картата.

Памет за константи – GPU притежава и малък обем памет 8kb, която е по-бърза от глобалната памет, а достъпът до нея позволява по-висока степен на паралелизъм.

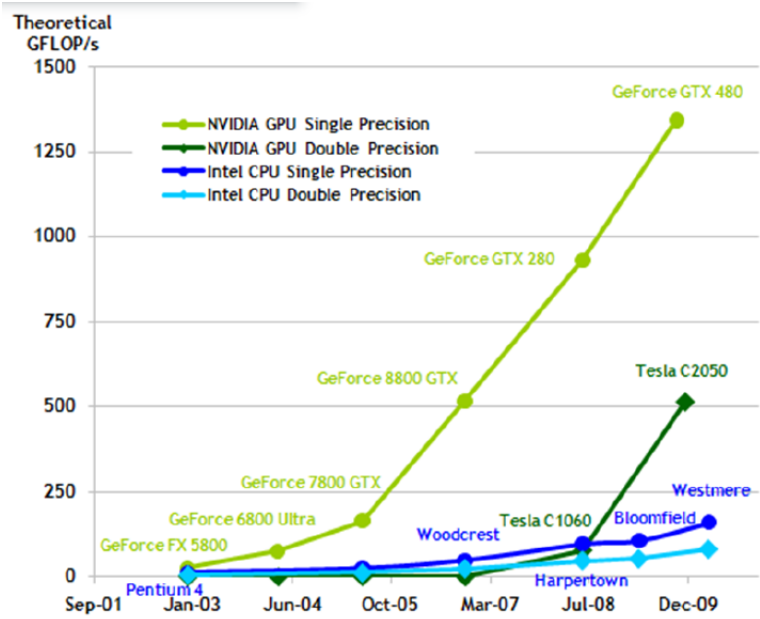
Памет за текстури – GPU притежава и текстурна памет, която може да се ползва при определени обстоятелства, като например за линейна филтрация на данни. Структурата на тази памет позволява запис на 2D матрици. Тази памет се кешира и е принципно само за четене.

2.3 Развитие на технологията CUDA

Проектът CUDA излиза на бял свят заедно с видео карта - G80 (ноември 2006г.), като първата SDK е пусната февруари 2007г. Първата версия с номер 1.0 се поддържа до излизането на професионалните клъстерни карти Tesla (юни 2007г.) и е съвместима с процесорите G80. Тя е предназначена за пазара на изчислителни системи. Версия 1.1 въвежда функции за ползването на NVIDIA драйверите, като са поддържани карти след GeForce 8 и по-новите версия. Това е ключов момент за програмистите, тъй като прави възможно ползването на CUDA върху всички видове видеокарти, а също така и като симулатор върху стандартен компютърен процесор. Следва да се отбележи, че много от преимуществата са достъпни само за 64-битовата версия на Windows. Версия CUDA 2.0 е готова заедно с картите GeForce GTX 200, като бета версията се въвежда през началото на 2008. Следващата версия поддържа: изчисления върху нецелочислени променливи с двойна прецизност (float, double), като хардуерно това е постижимо само от модел GT200. Системата има поддръжка за Windows Vista (32- и 64-битова версия), Mac OS X и Linux. Разработените с всяка следваща версия приложения могат да работят само със същата и по-нови версии на CUDA SDK, като не

могат да бъдат поддържани от драйверите на по-старата версия. На фиг. 9 е дадено сравнение на производителността със съвременни Intel процесори излезли от производство в същия интервал от време. Този учебник е посветен на наличната към момента му на написване стабилна версия 4.0 CUDA TOOLKIT.

Фигура 2.5: Сравнение на производителността между поредни GeForce продукти и процесори Intel (изт. NVIDIA)



Тъй като това са видео карти, нормално е да се очаква, че ползваните в тях технологии за обмен на данните ще надвишават многократно скоростите на трансфер на данни между CPU и RAM на компютрите. Едновременно с това следва да се отчете, че NVIDIA забавят излизането на пазара на продукти с вградени атомични операции за работа с глобалната памет и споделените ресурси. Вместо това се представят нови платки имащи увеличена тактова честота

и потребление на енергоконсумацията. В действителност за пазара на игри това не е нужно, но именно този клас платки са масово достъпни като цени за повечето програмисти. За разлика например, една карта TESLA има до 4-5 пъти по-висока цена за същата производителност както една GeForce карта. Особената разлика, че TESLA поддържа всички атомарни операции и има вградени АЛУ за аритметични операции с двойна прецизност. Освен тези обки особености всеки модел видео карти притежава специфични отличителни черти. Например всички карти имат съвместими функции позволяващи четене и запис на данни в глобалната и споделената памет, четене и запис на данни в текстурите и константната памет. Но подобно на различните технологии и поколения процесори и различните поколения мултипроцесори имат допълнителни функции. Едни от тези важни за бързодействието функции са тъй наречените атомарни операции. Важно е да се знае, че действия с атомарни операции над глобалната памет са достъпни в архитектура 1.1, а действия с атомарни операции над споделената памет са достъпни от архитектура 1.2. Същевременно програми написани за 1.0 са изпълними върху следващото поколение карти, без поддръжка на обратна съвместимост. Естествено ако смятате да ползвате технологията за инженерни изчисления и симулации се ориентирайте към версии 1.3 или 2.X.

Глава 3

Инсталиране и настройка на работната среда

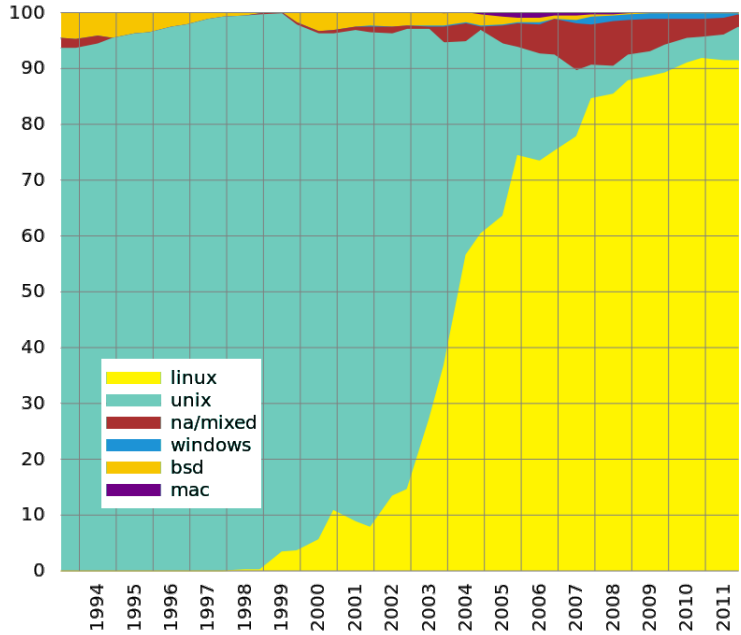
В тази глава ще научите как да инсталирате операционна система Linux Ubuntu, като отделна операционна система или директно върху Windows. Ще разберете как да конфигурирате драйвера на видеокартата, CUDA Software Development Kit (SDK) и CUDA TOOLKIT (съдържащ нужните помощни програми и компилатора nvcc). Ще разгледаме някои команди, които ще ползвамте често при работа с Ubuntu. Ще покажем как да пишем CUDA програми с текстовия редактор gedit и geany, и съответно как да компилираме програмите през команден ред или през geany.

3.1 Инсталиране и конфигуриране на Linux Ubuntu

Тази глава е посветена на инсталирането и конфигурирането на работната среда за писане, компилиране и изпъл-

нение на CUDA съвместими програми. Примерите разгледани в този учебник са предвидени за компилиране и изпълнение върху операционна система Linux Ubuntu. Изборът на Linux се основава на факта, че през последните 10г. безплатните версии на Linux успяха да заемат над 85% от всички инсталирани суперкомпютри по света Фиг. 3.1, като към това се прибавят още 12%, които се заемат от системи ползващи платени Linux дистрибуции.

Фигура 3.1: Използвани операционни системи за изграждане на суперкомпютри по света. Изт. http://i.top500.org/stats



За онези от вас, които за първи път ще използват Linux е предвиден кратък увод за това как да инсталирате Ubuntu. Ако вече имате Ubuntu може да преминете директно към инсталирането на CUDA TOOLKIT и SDK. Описани са и някои основни команди за работа с файлове и папки в

Linux Ubuntu. Разгледани са двете основни програми, с които вие може да пишете свой програмен код в Linux. Това са текстовия редактор gedit и интегрираната среда за разработка на приложения geany. Обсъдени са и начините за компилиране на вашите програми директно от командния ред. За да започнете разработка на програми вие трябва да свалите и инсталирате версия CUDA TOOLKIT 4.0 (или по-нова варсия). CUDA SDK съдържа набор от софтуерни инструменти, компилатора **nvcc**, документи и приложения позволяващи компилирането на CUDA съвместими програми. Последна версия на нужните ви пакети със софтуер и драйвери може да изтеглите от сайта на производителя (http://developer.nvidia.com/cuda-toolkit-40). За инсталиране на CUDA TOOLKIT вие трябва да имате подходяща хардуерна конфигурация, работеща с една от следните операционни системи:

- Windows - 7, XP, VISTA
- Linux - Fedora, RedHat, Ubuntu, OpenSUSE, Linux Enterprise Server
- Mac OS X

Ако притежавате хартиен екземпляр на учебника на приложените дискове ще намерите инстарационна версия на Ubuntu 10.04 и инсталационни файлове на CUDA TOOLKIT 4.0, CUDA SDK и драйвери за разработка. Ако това е първата ви среща с Linux препоръчвам първо да прочетете тази глава, а при нужда да потърсите повече информация за инсталирането и работата с Ubuntu. Съществуват няколко основни начина за инсталиране на операционна система Ubuntu: на свободен твърд диск или свободен логически дял предназначен за него, върху Windows, като приложение или да го стартирате от така нареченото Live CD или

USB. Препоръчваме абсолютно начинаещите да инсталират специална дистрибуция на Ubuntu наричана Wubi. Тази дистрибуция ви позволява да качите Ubuntu директно върху Windows дял на твърдия диск на компютъра, като процесът по инсталиране прилича много на инсталирането на обикновенно Windows приложение. След рестартиране и зареждане на BIOS ще бъдете попитани коя операционна система да изберете да бъде стартирана. Този начин на инсталация е най-елементарен, като винаги когато поискате може да деинсталирате вашето Ubuntu по начина по който се дейнсталира стандартно приложение. За да получите максималното бързодействие от системата Linux ще се наложи да я инталирате върху специално предназначен дял от вашия твърд диск. За да започнете инсталацията на Ubuntu на отделен дял от вашия твърд диск следва да рестартирате вашия компютър и да проверите в настройките на BIOS (Basic Input / Output System) дали е конфигуриран, така че при първоначално пускане да проверява за наличен стартиращ (boot) диск поставен във CD-ROM/DVD устройството.

BIOS – Това е вградената в дънната платка на компютъра микро операционна система за управление и начално стартиране на компютъра. BIOS е достъпна всеки път, когато включите компютъра си. Съдържа всички основни процедури, необходими за начално стартиране на системата, в това число функции за тестване и комуникация между отделните хардуерни компоненти и периферните устройства. Тези процедури включват инструкция за обработване на изхода на екрана, отпечатване на принтер, сверяване на датата и часа, работа с твърди дискове, флопи дискови устройства и други дискови носители, като CD, DVD.

Възможно в момента на четене на учебника да е излязла нова версия на CUDA TOOLKIT. Ако описаните тук

процедури по нейното инсталиране не съвпадат с описаното тук ще трябва да се обърнете към документацията на съответния софтуер. Въпреки всичко на сайта на NVIDIA може да откриете последна версия на документа: **"CUDA Getting Started Guide (Linux)"**, в който се описват основните стъпки при инсталиране на CUDA TOOLKIT в различните версии на операционната система. Ако това е първият ви досег с CUDA ви препоръчвам при работа с примерите да използвате версия 4.0 или 4.1 с които гарантираме, че разгледаните примери ще работят задоволително. Нужния софтуер може да намерите на приложения диск.

Ако имате съмнения как точно да стартирате BIOS и да го конфигурирате се обърнете към описанието на дънната платка, а ако нямате книжка може да потърсите описанието и на сайта на производителя. След като конфигурирате BIOS да проверява първоначално за наличие на BOOT диск във вашия CD/DVD ROM, следва да добавите за второ BOOT устройство да бъде конфигуриран вашият първи твърд диск. (Това може да изиска малко време, особено ако го правите за първи път или на системата ви са инсталирани повече от един дискови носители.) Обикновенно влизането в BIOS става веднага след рестартиране на компютъра и натискането на бутон Delete, по-рядко някой от бутоните F1 -F12. След извършване на тази начална конфигурация следва да поставите инсталационния диск с Ubuntu във вашето дисково устройство и да рестартирате системата. Ако всичко е протеко добре след начално стартиране на екрана ще се изведе въпрос дали да започне операция по зареждане на ситемата от вашия инсталационен диск. Самата операционна система може да се инсталира и по други начини: чрез USB памет или директно в Windows, повече за това може да намерите тук: http://www.ubuntu.com/download. Началните стъпки по зареждане на Ubuntu от USB флаш

памет са подобни на зареждането от CD/DVD. Не всеки BIOS може да пъддържа първоначално зареждане на системата от USB памет. Ако правите това с USB памет, е добре преди да рестартирате компютъра и да влезнете в конфигурационната програма на BIOS да поставите USB паметта в свободен слот, в противен случай BIOS няма да я разпознае и съответно няма да можете да конфигурирате зареждащата програма да стартира първо от USB сменяемия външен диск. Останалите стъпки по инсталацията на Ubuntu са сходни за CD/DVD и USB. След начално зареждане на системата, за да потвърдите начало на инсталирането натиснете клавиша Enter. Бързината на инсталиране зависи от типа компютър и може да отнеме от 20 до 60 минути. Операционната система ще ви предостави опция да започнете стартирането и/или да я заредите директно от от компактдиска. Зареждането от компакт диска е препоръчително за начинаещи потребители които стартират Ubuntu за първи път. Така ще добиете увереност и ще имате готовност за това как изглежда тя и какво ви очаква. Независимо от опцията ще се наложи да се изчака няколко минути за да се зареди ядрото на операционната система. В процеса на работа при зареждане на системата от CD-ROM или USB промените, които правите или действията, които предприемате със заредената по този начин операционна система ще се загубят при изгасяне на компютъра. Това важи до стартиране на програмата за инсталиране.

При поставяне на Live CD (или Desktop CD) в CD устройството и стартиране на компютъра ви посреща начален инсталационен екран Фиг(). Ако не предприемете никакви действия след определено време стартирането на системата ще започне автоматично. (В зависимост от дистрибуцията началният екран ще се различава, възможно е Ubuntu да започне директно с процедура за зареждане на операционната система, или пък да изчака няколко секунди за

Фигура 3.2: Начално меню за избор на процедура по инсталиране на Ubuntu

натискане на произволен клавиш от клавиатурата, за да започне процесът на начолно инсталиране на системата. Ако решите да работите с CUDA върху Ubuntu сървър имайте предвид, че зареждането на гафичната среда не е автоматично, като тя може да бъде инсталирана в последствие. На начинаещите препоръчвам да изберат възможността за стартиране в "безопасен" графичен режим. След избора на първата точка от горното меню Убунту се зарежда за няколко минути в зависимост от бързодействието на компютъра. След приключване на този процес ще се изведе подобен екранен изглед 3.2.

След този момент вие разполагате с напълнофункционална операционна система, която е заредене в оперативната памет на машината. В този режим вие може да я разгледате, и дори да работите в интернет, ако компютърът ви

Фигура 3.3: Екранен изглед на Ubuntu стартирано от CD.

е свързан в мрежа чрез жичен или безжичен стандартен интерфейс. (В много малко от случаите Ubuntu може да не разпознае мрежовият интерфейс. Цените на стандартни мрежови карти днес са под 10лв. Ако случаят е такъв спестете си главоболия и купете и инсталирайте допълнителна мрежова карта на свободен PCI слот.). Без да сте инсталирали все още нищо върху вашият компютър вие имате достъп до всички програми. След като се "разходите" из Убунту и решите, че ви харесва, следва да започне същинската инсталация на системата. Процесът ще премине през следните основни стъпки:

Избор на език на инсталацията: Убунту е преведен на много езици и един от тях е и Български. При тази стъпка действията ви се свеждат до прост избор на езика, на който искате да протече инсталацията.

Избор на часови пояс: тук трябва да изберете държава и град на обитаване, за да може Убунту да определи правилно часови пояс и текущ час, които ще бъдат динамично изтегляни от специализиран сървър. Тази настройка е важна и в случаите когато ще желаете да ползвате допълнителни услуги базирани на вашето местоположение, като прогноза за времето и др.

Избор на клавиатурна подредба: Ubuntu има поддръжка за различни клавиатурни подредби, може да добавите и допълнителни езици, ако желаете. Не се притеснявайте може да направите това във всеки един момент след приключване на инсталацията.

Подготовка на твърдият диск за инсталиране: задължително преди да започнете инсталиране, ако на вашият компютър е била инсталирана друга работеща операционна система, като Windows архивирайте и запишете на компакт диск или друг външен носител всички важни документи, папки, програми и данни, които може да ви потрябват в последствие. (Тук е мястото да кажем, че това е важно действие, което всеки един уважаващ се програмист трябва да прави редовно. Архивирането и датирането на вашите проекти във всяка една стъпка на разработка е основна задача, която може да ви спести седмици труд.) Linux разпознава хардуерът на вашия компютър, като система от входни и изходни файлове. Досега вероятно може и да не наете, но и Windows също разпознава и работи с входно изходните устройства, независимо от типа им, като с файлове, но това в повечето случаи остава скрито за вас.

Твърдите дискове на компютъра в Windows се наричат: A, B, C, D и т.н., като A и B са запазени имена за първите две флопи дискови устройства, които е напълно възможно да отсъстват на вашият по-модерен компютър. В Linux твърдите дискове имат следните имена [7][стр.76-77]:

- /dev/hda – първи IDE диск
- /dev/hdb - втори IDE диск
- /dev/sda – първи SATA/SCSI диск
- /dev/sdb – втори SATA/SCSI диск
- /dev/fd0 – първо флопи дисково устройство (A)
- /dev/fd1 – второ фропи дисково устройство (B)
- /dev/eda – ESD дискове срещат се на старите IBM

Ако вашето дъно е произведено след 2010г. най-вероятно сте оборудвани с поне един SATA твърд диск на него. BIOS на дънните платки позволява на съвременните твърди дискове със сериен интерфейс да бъдат разпознавани от операционната система, като IDE твърди дискове с параралелен интерфейс. При стартиране на BIOS проверете настройките на вашият компютър.

В класическият случай вие ще трябва да инсталирате Ubuntu върху вашата съществуваща система с Windows. Това налага да отделите един логически или физически диск за инсталиране на новата операционна система. Ако вашият компютър е инсталиран върху един единствен физически и логически твърд диск, и не е оставено свободно дисково пространство ще трябва да опитате да работите с Wubi (дистрибуция на операционната система Ubuntu, която позволява безопасното и инсталиране и деинсталиране върху Windows по начин подобен на инсталирането на обикновенна програма). Ако това не ви върши работа ще се наложи да използвате други програми, като Partition Magic с които да преоразмерите вашият твърд диск, при наличие на свободно пространство. В краен случай може да закупите нов или ползван допълнителен твърд диск за вашия компютър. Ако инсталирате Ubuntu за първи път

отделете няколко часа време, потърсете и прочетете в интернет следните теми: и „WUBI или как да инсталираме Убунту от работната среда на Windows“. Тук е момента да спомена малко повече за бързодействието на системата. Много съвременни дънни платки и операционни системи, сред които и Ubuntu позволяват създаването на Redundant array of independent disks (RAID) дискови масиви. Тази технология ви дава възможност да се създаде дисков масив от два еднакви твърди диска, като при това е възможно да се постигне по-висока скорост на четене и запис на данните от диска на компютъра. Това е важно за напредналите програмисти на CUDA тъй като вашите програми дефакто ще бъдат предназначени за масивна паралелна обработка на големи обеми данни, като при това ще се налага активното ползване на твърдия диск на компютъра ви и следователно колкото той е по-.бърз толкова по-бързо ще работят вашете програми. Инсталирането на тази технология (RAID 0 – позволява постигането на скорости на четене и запис на данни от съвременните твърди дискове със скорост до 3GBps, а при ползване на по-ново поколение дискове до 6GBps.) обаче изисква повече познания и не го препоръчвам на начинаещите потребители, тъй като повреда в единия твърд диск ще причини загуба на данните за цялата система. Споменавам тази технология единствено, за да имате идея за възможното и приложение при създаването на комерсиални, инженерни и научни системи за паралелна обработка на информация с CUDA. Ако инсталиране Ubuntu върху компютър с инсталиран Windows е възможно тази опция да е била предварително активирана, това в най-общия случай означава, че преди инсталацията на Ubuntu ще трябва задължително да архивирате вашите важни файлове и документи на отделен носител. При всички случаи, ако сте начинаещ препоръчвам да извършвате експеричентите върху компютър, на който нямате отговорна информация и важни програми с които работи-

те ежедневно. Ако използвате приложеният компакт диск, ще може директно да инсталирате Ubuntu 10.04 върху вашата машина, който ви гарантира съвместимост с CUDA Toolkit 4.0. Също така ше добавя информативно, че ако след инсталирането на Ubuntu и има проблем с разпознаването на вашите Windows дялове следва да потърсите повече информация за това как да инсталирате „grub 2“ върху вашето Ubuntu. Това е програмата, която управлява началният избор на операционна система, която да се стартира сред пускане на компютъра. По принцип инсталирането на Ubuntu няма да доведе до подобни проблеми. За да се предпаците от грешки прочетете внимателно извежданата от съветника по инсталацията информация. Ubuntu ще ви предложи следните няклоко режима на инсталиране:

- Инсталиране върху празен твърд диск (чиста инсталация, препоръчва се ако желаете да получите максимално висока производителност на системата).
- Автоматично заемане на незаетото място на диска, това е абсолютно безопасен начин за инсталиране, но това значи, че или работите на чисто нов компютър или Windows не е конфигуриран да вижда целият размер на вашите дискове.
- Автоматично изтриване на целият твърд диск и запис, този вариант се избира когато сте решили да инсталирате Ubuntu върху чиста система. Тази опция ще изтрие цялото съдържание и съществуващи логически дялове на вашия твърд диск.
- Ръчно разделяне на диска, този тип инсталация изисква повече познания и внимание, но ви дава редица преимущества за това къде и как да инсталирате Ubuntu. Съвременните помощни програми за инсталирането и разделянето на дисковете са достатъчно

интуитивни, но не препоръчвам да правите това ако инсталирате Ubuntu за първи път.

Оставащи стъпки от инсталацията

Мигриране на документи и настройки: при тази стъпка инсталиращата програма на Ubuntu ще провери дали върху компютъра ви има инсталирани други операционни системи, като Windows. Ако бъдат открити ще ви бъде предложено да вземете потребителското име, парола, лични документи, настройки на браузъра и други, за да ви улесни при началното адаптиране за работа със системата.

Създаване на потребители: в тази стъпка ще трябва да създаде и опише основният потребител, който ще работи със системата. Задължително запомнете вашето потребителско име с администраторски правомощия и неговата парола. Исканите параметри са: име, потребителско име, парола и име на компютъра. Тъй като ние ще пишем, а това е свързано с инсталиране и конфигуриране на системата ще трябва да регистрирате само един потребител с администраторски права. По-късно може да добавяте нови потребители в процесът на работа.

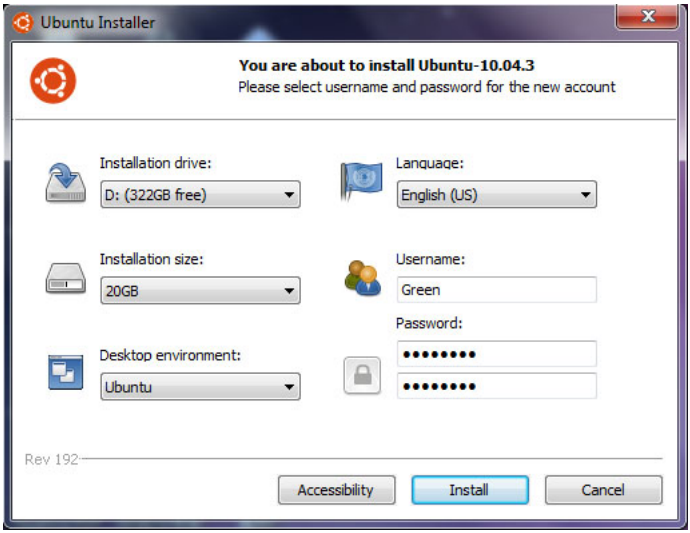
Преглед на настройките и начало на процеса по инсталиране на Ubuntu: тази стъпка е с информативен характер. При нея в един прозорец са сместени всички настройки, които са правени в предходните стъпки. След като ги прегледате, инсталацията на Убунту се задейства с натискане на бутона Install. С това вашата работа по инсталацията приключва. Ubuntu започва инсталиране, като извежда информация докъде и стигнало то и още колко време остава. Инсталацията ще отнеме между 20 и 60 минути. Като приключи ще се изведе съобщение, че може да рестартирате компютъра. Компакт дисковото устройство ще бъде извадено автоматично, махнете го от дисковия плейър и рестартирайте системата. Ако всичко приключи

успешно след рестартиране ще ви посрещне следния екран. Въведете вашето потребителско име и парола, за да започнете вашата първа потребителска сесия в Ubuntu. Както ще видите процедурата по инсталиране на Ubuntu е сравнително елементарна, ако не успеете от първия път опитайте пак. Ако преминавате от Windows непременно експериментирайте първо с Wubi. След като стартирате системата за първи път отделете няколко минути и разгледайте менютата и. При работа в мрежа Ubuntu постоянно ще проверява за нови актуализации на системата, включително ще ви предлага изтеглянето и инсталирането на нови ядра на операционната система. Тъй като основната ни цел е да пишем програми за CUDA не инсталирайте автоматично предлаганите ви програмни модификации, това може да доведе до проблеми в работата ви със CUDA toolkit и видео драйверите.

Фигура 3.4: Екран за вход в Ubuntu след стартиране на компютъра. (Може да се различава в зависимост от версията на операционната система.)

При инсталирането на Wubi всичко което е необходимо е да направите е да изтеглите инсталационния файл http://www.ubuntu.com/download/ubuntu/windows-installer и да го стартирате. След като стартирате инсталатора ще трябва да изберете обема на твърдия диск, който да бъде динамично алокиран от инсталатора във файловата система на Windows. Освен това задължително трябва да определите име на администратора на системата и парола за влизане в нея. Тези данни ще се ползват за последващо влизане в системата след приключване на инсталацията. Ако Wubi не ви допадне, макар да е напълно функционална система, винаги може да го изтриете, по същия начин по който се деинсталират приложения в Wondows.

Фигура 3.5: Процедура за инсталиране на Wubi - в Windows.



3.2 Инсталиране и конфигуриране на CUDA: драйвер за разработка, TOOLKIT и SDK

Инсталирането на CUDA върху Linux Ubuntu е по силите и на сравнително начинаещи програмисти. Изборът на тази платформа е естествен поради следните факти: Ubuntu е безплатна и бързо развиващаа се Linux дистрибуция, която е подходяща за начинаещи потребители, много производители на софтуер за разработка вече създават работоспособни версии на системите си за Linux, практически над 90% от супер компютрите в света използват Linux базирани операционни системи (TOP500.Org). Самата система е широко и лесно мащабируема, като позволява направата на изчислителни клъстери (Cloud), и не на последно място NVIDIA предлага изцяло пълна Linux поддръжка на драйверни програми за хардуера и развойната среда за CUDA. Тъй като примерите са на езикът C++ е възможно директното им компилиране и в среда Windows. След CUDA Toolkit 4.0 използването на CUDA стана много по-елементарно и надеждно с Ubuntu. Самата операционна система е изключително стабилна и има приятен графичен интерфейс. Ако сте начинаещ потребител и разработчик бъдете готов за някои проблеми свързани с инсталирането и конфигурирането на драйверните програми за видео картата. Възможни проблеми бихте имали и при промяна на конфигурацията на системата, затова след инсталирането и избягвайте ъпдейти с нови ядра на операционната система. Тази система сама проверява за наличие на ъпдейти, в така наречените репозиториуми. Това са адреси на софтуерни доставчици, които системата периодично може да проверява за актуализации и да ви извежда съобщения за препоръчани актуализации. Ако решите да ъпдейтвате сис-

темата с ново ядро, имайте предвид за възможни начални проблеми с работата на видео картата. Затова една добре настроена система за разработка на приложения следва да не се актуализира освен след детайлно четене на форуми и постове за ползването и с CUDA.

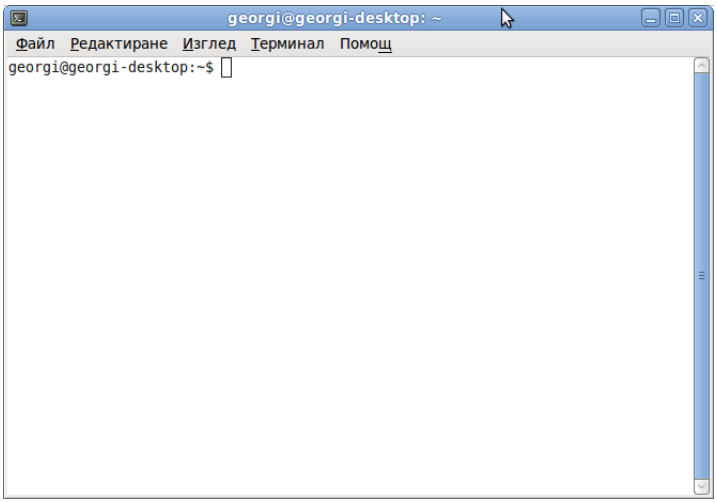
Това което ще получите от Ubuntu е практическа безотказност и среда на работа, която за разлика от Windows не променя бързодействието, отказоустойчивостта си и външният си вид в процеса на работа. Самият синтаксис на CUDA е модифицирано C++ с някои разширени команди позволяващи ви да управлявате по-пълноценно драйверите на CUDA на високо ниво. Преди да започнете проверете дали вече разполагате с персонален компютър имащ подходящи параметри. Може да използвате компютър намиращ се във вашата университетска лаборатория. За да програмирате реални приложения се изисква да работите с CUDA съвместима видео карта и достатъчно мощна платформа. Машина със следните минимални изисквания ще бъде подходяща: CPU Intel желателно Dual Core или повече (2GHz), RAM поне 1GB, HDD 50-100GB и видео карта от серията GeForce 200 или по-висока ще е напълно достатъчно. Имайте предвид, че за да получите добре бързодействаща система е редно да ползвате видео карта, като минимум с 128 ядра и поне 512MB оперативна памет. Естествено може да инталирате Ubuntu и CUDA върху компютър без 3D видео карта на NVIDIA но тогава ще работите без хардуерна поддръжка на системата. Това ще доведе до далеч по-ниски параметри на бързодействието на вашите програми. В този случай следва да се ограничите в използването на CUDA симулатора, който идва със стандартния пакет на драйверите за разработка. Ползвайки симулатор програмирането на CUDA няма да ви донесе нужнотоо удоволствие, тъй като програмите ви ще вървят далеч по-бавно дори от програмите написани за работа централния проце-

сор на вашия компютър. Въпреки всичко началното заползнаване със системата е възможно дори и в този случай. CUDA Developer SDK предлагания от NVIDIA комплект за програмиране се включват примери с програмен код, които включват:

- Паралелни сортировки;
- Умножение на матрици;
- Транспониране на матрици;
- Измерване на производителността с таймери;
- Паралелно събиране (scan) на големи и малки масиви;
- Конволюция на изображения;
- Уейвлет трансформация;
- Изобразяване на графика с OpenGL и Direct3D;
- Използване на CUDA BLAS FFT библиотека;
- CPU-GPU C и C++ миксиране на код;
- Монте-Карло алгоритми;
- Паралелен генератор на случайни числа;
- Паралелно изчисление на хистограми;
- Отстраняване на шум в изображения;
- Реализация на филтър на Собел за оконтуряване на изображения и др.

За да инсталираме пакети и програми в Ubuntu ще използваме терминала. Това е програмен прозорец, който много прилича на MS-DOS prompt в Windows. Не се плашете, терминалът се стартира от менюто „Applications->Accesoaries->Terminal“ или „Програми->Помощни програми->Терминал“ Фиг (). Изгледът на терминала може да се различава, както по цвят, така и по това какво е изписано в него. Терминалът е основният начин за работа с Ubuntu и други Linux/UNUX операционни системи. Постепенно ще свикнете с него. За разлика от Windows и другите затворени операционни системи, които „правят това което си искат“, Linux прави само това, което му наредите.

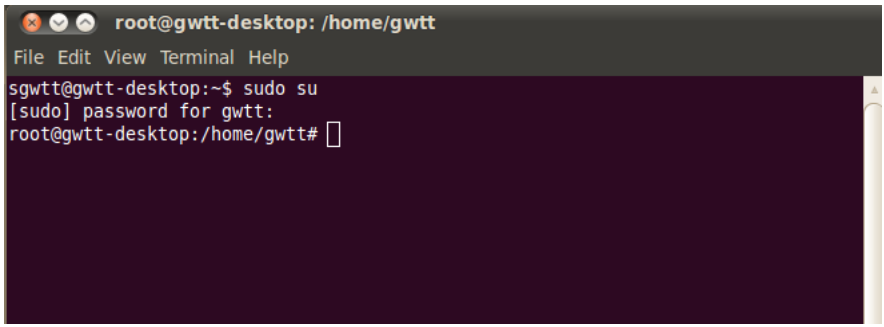
Фигура 3.6: Терминалът - основният комуникационен интерфейс между вас и Linux.



За да добиете администраторски права за работа в терминала ще трябва да въведете следната команда: „sudo su“, като след това Ubuntu ще изведе подсещане да въведете вашата парола, тя е същата като тази използвана за начално влизане в системата. Запомнете, за да инсталирате

и конфигурирате системата следва да влизате в нея, като администратор, а не като потребител. Sudo ви позволява да имате административни привилегии над изпълнението на системата. Помнете, Linux различава размерът на буквите които изписвате, затова файли или папки с името: „file“ и „FILE“ са две различни неща.

Фигура 3.7: Изпълнете командата ”sudo su”, за да влезнете в режим на администратор на системата.



Работа с терминал

При работа в терминала следва да спазвате това правило, същото важи и при писането на вашите програми. Linux, за разлика от Windows, различава символите написани с главни и малки букви. Ще ви трябват и следните няколко потребителски команди, за да може успешно да се навигирате в Ubuntu:

- **sudo** – вход в администраторски режим на работа, след тази команда може да се извика команда или операция, която изисква администраторски привилегии. Командата изисква въвеждане на административна парола. Командата може да се изпълни и еднократно, като sudo su, след което не е нужно нейното повторно въвеждане при всяка следваща команда.

Команди за работа с файлове:

- **cd** – преход в папка (директория)
- **cd ..** - връщане в предната директория
- **pwd** – показва пълния път до папката в която се намирате
- **ls** – (като dir) печати списък с файловете в текущата папка, като папките и файлите са оцветени в различен цвят
- **cp** – корипа файл
- **mv** – изрязва и премества файл (mv 1.txt 1 – копира файл 1.txt в папка 1)
- **rm** – изтрива файл, при изтриване на папка rm -r име на папката
- **mkdir** – създава папка
- **dir** – (не се поддържа от всички Linux дистрибуции, подобно на "ls") показва съдържанието на папката включително и директориите без да ги оцветява

С командата "dir" вие може да видите съдържанието в текущо избраната папка, например:

```
root@gwtt-desktop:~# dir
altera  CUDASDK  NVIDIA_GPU_Computing_SDK
```

Ако желаете да влезнете в една от изброените папки ще изпълзвате командата "cd":

```
root@gwtt-desktop:~# cd CUDASDK
root@gwtt-desktop:~/CUDASDK#
```

Ако сега отново изпълните командата „dir“ ще се изведе съдържанието на папката „CUDASDK“:

```
root@gwtt-desktop:~/CUDASDK#dir
C          cudpp_license.txt   Documentation.html
   Makefile shared
CUDALibraries  doc         License.txt          OpenCL
```

Използвайки командата „cd“ без параметри, ще се върнете в основната папка. Ако искате да се върнете само на едно ниво, при многократно влизане във вложени папки се ползва командата „cd ..“:

```
root@gwtt-desktop:~/CUDASDK# cd C
root@gwtt-desktop:~/CUDASDK/C# cd ..
root@gwtt-desktop:~/CUDASDK#
```

командата „pwd“ ви позволява да видите пълното име на пътя в папката в която се намирате:

```
root@gwtt-desktop:~/CUDASDK# pwd
/root/CUDASDK
```

За повече информация и начално запознаване потърсете в интернет за „terminal comands Ubuntu“ или отидете директно на този линк:
https://help.ubuntu.com/community/UsingTheTerminal

Тези четири команди са ви достатъчни, за да започнете работа с терминала. Ако имате време изучете и останалите, това ще ви спести доста усилия в следващ момент. За изпълнението на примерите от този учебник може да ползвате само тези команди, като копирането и преместването на файловете извършвате от графичния интерфейс. Вие може да времнете в програма подобна на Windows Explorer по всяко време, от менюто Places->Computer. Някои от папките, които ще инсталирате като администратор ще бъдат заключени. Тогава ще използваме още една команда (chown), за да получим и достъп до тях през графичния интерфейс, но за това по-късно.

Инсталиране на видео драйвер

Ако сте инсталирали успешно Ubuntu най-вероятно вашата видео карта е била правилно разпозната и инсталирана. Драйверът е тази системна програма, която прави

управлението на видео картата възможно от самата операционна система. Без разпознат драйвер за видео картата системата ви ще работи в редуциран графичен режим, с ниска резолюция или лошо качество на цветовете. Вие може да изтеглите последна версия на вашия драйвер от тук: http://www.geforce.com/Drivers.

Видео драйвер за разработка

Предполагам вече сте успели успешно да инсталирате вашето Ubuntu и сте се разходили из неговите основни менюта, най-вероятно сте влязли и в интернет. Първото нещо което трябва да направим е да реконфигурираме системата така че да може да използва драйверът на вашата видео карта правилно. В процесът на инсталация Ubuntu е стартирал, а можеби и разпознал, точния или безопасен графичен драйве. Точния драйвер на системата може да изберете от сайта на производителя http://developer.nvidia.com/cuda-toolkit-40. На приложения диск ще намерите следните три архива с които започва нашето пътешествие в страната на CUDA:

- **devdriver_4.0_linux_32_270.41.19.run**
- **cudatoolkit_4.0.17_linux_32_ubuntu10.10.run**
- **gpucomputingsdk_4.0.17_linux.run**

Добре е да копирате предварително тези архиви във вашата работна папка Documents, това ще ускори процеса на инсталирането им. Възможно е в момента на четене на този учебник да има нова версия на CUDA Toolkit, която може да свалите от сайта на NVIDIA (http://developer.nvidia.com), като инсталирането и конфигурирането може да се различава от това на версия 4. Ако имате проблеми с нещо винаги търсете помощ в интернет.

За да инсталираме видео драйвера подходящ за разработка на приложения трябва да деактивирате вашият текущ видео драйвер, за целта отворете терминала и въведете следния ред:

```
$ sudo apt-get purge remove nvidia*
```

След това трябва да създадете нов конфигурационен файл в папка **/etc/modprobe.d**, като въведете даденото по-долу съдържание. Целта на тази операция е да се избегне конфликти между работно инсталирания видео драйвер на системата и драйвера за разработка, който ще инсталираме по-късно. Следната команда ще отвори конфигурационния файл, в който вие следва да въведете съдържанието дадено по-долу. Изписвайки командата **gedit** вие ще стартирате текстов редактор в графичен режим, като в него ще се отвори или съществуващ вече файл или чисто нов файл. След промени на тези конфигурационни файлове трябва задължително да ги съхранявате, понякога ще се налага рестартиране на системата или изпълнението на команди, които да актуализират направените промени валидни за работещата операционна система.

```
gedit /etc/modprobe.d/blacklist-nouveau.conf
```

Добавете това съдържание в отворилият се текстов редактор, съхранете файла и го затворете, след това отново ще се върнете в терминала. Всички редове в конфигурационните файлове започващи с # ще бъдат интерпретирани от Linux, като коментари ползвани за записване на помощни бележки.

```
# /etc/modprobe.d/blacklist-nouveau.conf
blacklist nvidiafb
blacklist nouveau
blacklist rivafb
blacklist rivatv
blacklist vga16fb
```

```
options nouveau modeset=0
```

След това ръчно трябва да актуализирате образа на ядрото със следната команда в терминала, след нейното изпълнение следва да рестартирате операционната система:

```
update-initramfs -u
```

Инсталиране на инструментите за разработка

След успешното конфигуриране на Ubuntu с основния видео драйвер вече може да инсталирате средствата за разработка на CUDA. Изпълнете следните команди в терминала, това изисква интернет връзка и може да отнеме известно време в зависимост от състоянието на системата.

```
apt-get update
apt-get install build-essential
```

Инсталиране на OpenGL драйвер за разработка

В CUDA Toolkit 4.0 има примери за ползване на OpenGL, това е библиотека с отворен код, която ви позволява да работите с компютърна графика, виде и изображения, звук и работа с различни входно изходни устройства, като: мишки, джойстици и др. Поради това вие следва да инсталирате OpenGL средата за разработка върху вашата система, тази операция може да отнеме до няколко минути в зависимост от скоростта на вашия интернет.

```
apt-get install freeglut3-dev libxi-dev
  libxmu-dev
```

CUDA Developer Driver

Драйверът за разработка се намира на приложения диск с име "devdriver_4.0_linux_32_270.41.19.run". Възможно е

да не разполагате с такъв диск, или поради друга причина да искате да свалите ново копие на файла. Може да направите това от сайта на NVIDIA:
http://developer.nvidia.com/object/cuda_download.html
Тук има една особеност, за да инсталираме видео драйвера ние трябва да излезнем от графичен режим на работа в системата. Трябва да напуснем графичния режим с комбинацията от клавиши Alt+Ctrl+F2. След това трябва отново да се логнем в конзолата, използвайте вашето администраторско потребителско име и парола, след това изпълнете отново командата „sudo su“, която ще изиска повторна автентификация на паролата. Следната команда ще спре графичния мениджър на системата.

```
sudo service gdm stop
```

Сега вече може да инсталираме подходящия драйвер. Използвайте командата "cd", за да се придвижите до папката в която сте копирали драйверът за разработка. В моя случай файлът с архива съдържащ видео драйвера се намира в папката "Downloads", изпълете командата "cd" с параметри в зависимост от това къде се намира файлът.

```
root@gwtt-desktop:~# cd /home/gwtt/Downloads
root@gwtt-desktop:/home/gwtt/Downloads# dir
devdriver_4.0_linux_32_270.41.19.run
```

Тази команда ще стартира инсталирането на видео драйвера. В процесът на работа ще ви се извеждат няколко съобщения с възможност за избор. Четете внимателно, но в повечето случаи всичко което трябва да парвите е да избирате "OK".

```
sh devdriver_4.0_linux_32_270.41.19.run
```

След това изпълнете следната команда, за да рестартирате системата:

```
sudo reboot
```

Ако всичко е приключило успешно вие ще стартирате Ubuntu с новия графичен драйвер директно в графичен режим. При неуспех е възможно да се наложи да повторите горната процедура наново. При това след окончателното инсталиране на системата или при ъпдейт (което не препоръчваме да правите след, като веднъж настроите работната среда) е възможно да се наложи повторно преинсталиране на драйверите за разработка. Това се случва понякога при качване на нова версия на OpenGL - средата за 3D графика, която се ползва в някои от дадени по-долу примери.

CUDA Toolkit

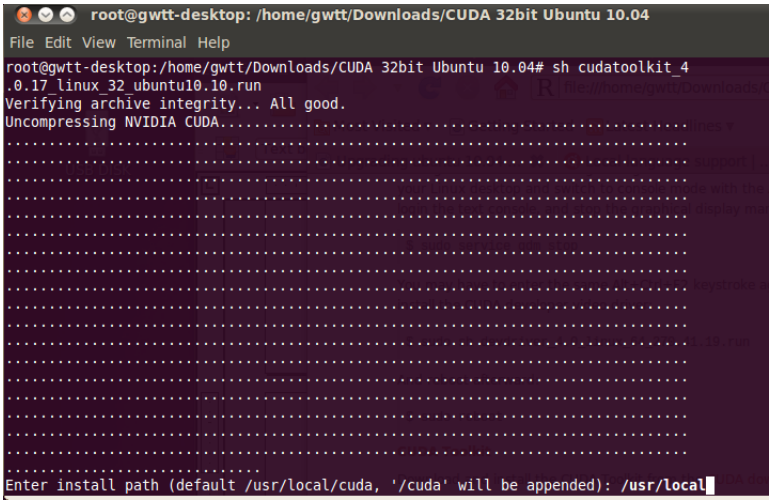
Това е наборът от инструменти, включително и nvcc компилатора, които ще са ви необходими, за да стартирате готовите демонстрационни приложения, да пишете и създавате собствени програми за CUDA. Може да намерите пакета на приложения диск, файл с име: cudatoolkit_4.0.17_linux_32_ubuntu10.10.run, или да го свалите от сайта на производителя. В моя случай файлът се намира отново в папката "Downloads". За да започне инсталирането му първосе преместете до папката в която той се намира и след това въведете следната команда:

Инсталирането на наборът софтуерни инструменти става със следната команда:

```
sh cudatoolkit_4.0.17_linux.run
```

Процесът може да отнеме няколко минути. По подразбиране CUDA toolkit ще се инсталира в папката: "/usr/local/cuda". След приключване на инсталацията следва да направите още няколко допълнителни стъпки, за да направите възможно използването на току що инсталираните софтуер-

Фигура 3.8: Инсталиране на CUDA TOOLKIT.



ни инструменти за разработка през терминала на Ubuntu. След като се инсталира е възможно да се наложи да ползвате телрминала, за да се придвижите в нея и да изпълните следната команда, а да получите отново административен достъп до папката в графичен режим: "**chown -hR username:root /usr/local/cuda/**".

Където "username" е вашето потребителско име. Това се налага тъй като ние инсталираме всички софтуерни пакети в административен режим. В моя случай това е:

```
chown -hR gwtt:root /usr/local/cuda
```

Остават още няколко стъпки, които ще направят ползването на току що инсталирания кит достъпен за работа. Следва да ползваме програмата gedit и да променим съдържанието на следните конфигурационни файли, както следва: Да настроим стандартните пътища до работните папки - PATH:

```
gedit /etc/environment
```

Промените в тези конфигурационни файли може да не представляват проблем, ако това е ново инсталирана система, но яко работите върху чужд компютър много внимавайте какво триете и добавяте. Непременно създавайте резервни копия на тези файли. За целта може да ползвате командата на текстовия редактор gedit File->Save As. Сега следва да откриете следния ред във файла "environment":

PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/usr/games"

променете съдържанието му на:

PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games:**/usr/local/cuda/bin**"

Съхранете файла със същото име на същото място. Сега следва да презаредим новото съдържание на този файл, за да го направим активно в системата без да се налага рестартирането и. Това може да стане с командата:

```
source /etc/environment
```

Остава ни да обявим и пътят до бгблиотеките необходими на компилатора за работа - LD_LIBRARY_PATH:

```
gedit /etc/ld.so.conf.d/cuda.conf
```

Добавете следните два реда в началото на празния файл и го съхранете, след това затворете текстовия редактор.

```
/usr/local/cuda/lib64
/usr/local/cuda/lib
Save and quit the editor.
```

Сега следва да презаредим новото съдържание на този файл, за да го направим активно в системата без да се налага рестартиране.

```
sudo ldconfig
```

Остава ин още малко работа, преди да имаме работоспособна CUDA работна среда. Следва да модифицирате „ /.bashrc“ файла, който се намира във вашата домашна папка, като изпълните следната команда:

```
root@gwtt-desktop:~# gedit ~/.bashrc
```

Това ще отвори съдържанието на файла в текстовия редактор gedit, преместете се в края на файла и добавете следните 3 реда, след това рестартирайте терминала (В тези файлове коментарите, текстът който указва нещо на потребителя и помага четенето на файла се записват с диез "#"):

```
export CUDA_HOME="/usr/local/cuda"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:
                        ${CUDA_HOME}/lib"
export PATH=${CUDA_HOME}/bin:${PATH}
```

Изпълнете следната команда, за да напарвите промените валидни за цялата система:

```
$ sudo source ~/.bashrc
```

Сега вече имаме напълно работоспособна среда за програмиране, но остана да настроим и стандартния компилатор gcc, за работа със новата среда. Това можеи да стане при наличие на интернет връзка, изпълнете следните команди.

```
$ sudo apt-add-repository
                ppa:ubuntu-x-swat/x-updates
$ sudo apt-get update
$ sudo apt-get install gcc-4.4
                g++-4.4 nvidia-curren
$ sudo apt-get install gcc-4.3 g++-4.3
```

CUDA SDK (Software Development Kit)

Този пакет може да намерите на приложения диск под името "gpucomputingsdk_4.0.17_linux.run" или да свалите наново от сайта на производителя. За да го инсталирате изпълнете следните команди:

```
root@gwtt-desktop:~# cd /home/gwtt/Downloads
root@gwtt-desktop:/home/gwtt/Downloads#
        sh gpucomputingsdk_4.0.17_linux.run
```

В процеса на инсталация ще трябва да уточните мястото на което ще се инсталира SDK кита, и мястото на което се намира CUDA toolkit. За място на инсталация изберете: „ /CUDASDK“ и потвърдете мястото на CUDA toolkit "/usr/local/cuda". Това е всичко, вече може да пристъпим към компилиране на приложенията. Използвайки терминала се преместете в папката на току що инсталирания SDK и изпълнете следната команда. Това ще компилира всички инсталирани в системата примери, като ако приключи успешно вие ще имате изпълними файлове на примерите. Ако този процес не приключи успешно е възможно да се наложи да повторите действията описани по-горе, без да преинсталирате драйвера, а само CUDA toolkit, като обърнете внимание на конфигурационните файлове.

```
root@gwtt-desktop:~/CUDASDK/C# make
```

Ако сте съблюдавали указаните инструкции по инсталиране на CUDA TOOLKIT до тук, пътищата за файловете ще бъдат същите. Стартирането на примерите може да направите по следния начин:

```
~/CUDASDK/C/bin/linux/release# ./deviceQuery
```

Ако на екрана се изпише подобна информация, значи всичко е наред и можем да пристъпим към компилацията на примерите.

```
[deviceQuery] starting...
./deviceQuery Starting...
CUDA Device Query (Runtime API) version (CUDART static linking)
Found 1 CUDA Capable device(s)
Device 0: "GeForce GTS 250"
CUDA Driver Version / Runtime Version 4.0 / 4.0
CUDA Capability Major/Minor version number: 1.1
Total amount of global memory: 1023 MBytes (1073020928 bytes)
(16) Multiprocessors x ( 8) CUDA Cores/MP: 128 CUDA Cores
GPU Clock Speed: 1.51 GHz
Memory Clock rate: 1000.00 Mhz
Memory Bus Width: 256-bit
Max Texture Dimension Size (x,y,z) 1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(8192) x 512, 2D=(8192,8192) x 512
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 16384 bytes
Total number of registers available per block: 8192
Warp size: 32
Maximum number of threads per block: 512
Maximum sizes of each dimension of a block: 512 x 512 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
Maximum memory pitch: 2147483647 bytes
Texture alignment: 256 bytes
Concurrent copy and execution: Yes with 1 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Concurrent kernel execution: No
Alignment requirement for Surfaces: Yes
Device has ECC support enabled: No
Device is using TCC driver mode: No
Device supports Unified Addressing (UVA): No
Device PCI Bus ID / PCI location ID: 1 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
>
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.0, CUDA Runtime
Version = 4.0, NumDevs = 1, Device = GeForce GTS 250 [deviceQuery] test results...
PASSED

    Press ENTER to exit...
```

3.3 Компилиране на програми с nvcc

Компилирането на CUDA програми може да става през команден ред от терминала или чрез специализирано при-

ложение улесняващо и интегриращо вашата работа с проектите. В Windows CUDA toolkit се интегрира във Visual Studio 2008-2010. В Linux процесът на разработка на приложения изисква малко повече услилия. По-горе ние компилирахме всички примери от CUDA SDK чрез използване на така наречен make файл. В подобни make файли се описват последователности от команди, местоположение на изходни файли, билиотеки с функции и други подробности свързани с компилацията на приложенията. Направата на подобен make файл ще дам по-долу. За момента ще се спра над няколко елементарни команди чрез които може да компилирате вашите примери. Обикновенно файлите съдържащи CUDA изпълним код се именуват с разширение „*име на файла.cu“. Компилаторът на NVIDIA се нарича nvcc, повече за неговото използване може да намерите в „The CUDA Compiler Driver NVCC .pdf“ [4], който се намира в папка „/usr/local/cuda/doc“. Най общо казано текстът на вашите програми съдържа части от код, който ще се изпълнява върху централния процесор и части от код, който ще се изпълнява върху графичния процесор. В процесът на анализ на кода nvcc определя коя част от кода да се компилира от gcc и коя от него (процесът е доста сложен и няма да се спираме на подробности). Това ни дава възможност да пишем смесен код в едни и същи файли. Ако предположим че имаме вече написана програма съдържаща се в един файл ние бихме могли да я компилираме със следната команда изписана в терминала. За целта трябва да се навигирате до папката в която стои вашият проект и да изпълните следната команда:

```
nvcc -o my_prog my_prog.cu
```

В този ред има няколко важни параметъра, първо е викането на компилатора nvcc, параметърът „-o“ задава името на изходния изпълним файл, последвано от името на

сорс файла, който съдържа изхподния код на програмата. Съществуват случаи в които компилирането на програмите ще изисква малко по-различен синтаксис, това е при използване в изходния код на атомарни операции, за които ще стане дума по-късно, и когато се налага ползване на допълнителни средства, като OpenGL библиотека при изчертаване на графика.

```
nvcc -lglut my_file.cu -o my_file
```

Компилирането на код съдържащ атомарни операции е допустимо само ако той ще се изполни върху устройства с изчислителни възможности 1.1 (за атомарни операции в глобалната памет) 1.2 (за атомарни операции в споделената памет).

```
nvcc -arch=sm_11 -v my_file -o my_file
```

```
nvcc -arch=sm_12 -v my_file -o my_file
```

След като вашата програма се компилира успешно вие можете да стартирате съответния файл от същата папка с командата (./име на файла):

```
./my_file
```

3.4 Писане на програми с gedit или geany

Писане на програми с gedit

Това е елементарен текстов редактор, той се инсталира по подразбиране по време на инсталацията на операционната система. Можете да го стартирате от менюто Applications->Accessories->gedit Text Editor. Самият редактор прите-

жава няколко основни функции, които го правят незаменим в писането и работа с конфигурационни файлове. Редакторът има възможност за инсталация на допълнителни приставки наричани plugins. Редакторът разпознава и „оцветява“ синтаксиса на много езици за програмиране: C/C++, Java, PHP и др. Това значи че файловете с разширение: *.c, *.cpp, *.cu ще бъдат по-лесно четими от вас. Повече за текстовия редактор gedit може да намерите на http://projects.gnome.org/ Тази програма може да се ползва ефективно при писане на кода за вашите проекти. След създаване на файлите следва да ги съхранявате с подходящи имена и разширение *.cu. Компилирането на тези файли става през терминала изпълнявайки следната команда:

nvcc -o име-на-изпълнимия файл име-на-файла-със-сорс-кода.cu

стартирането на програмата може да стане с командата:

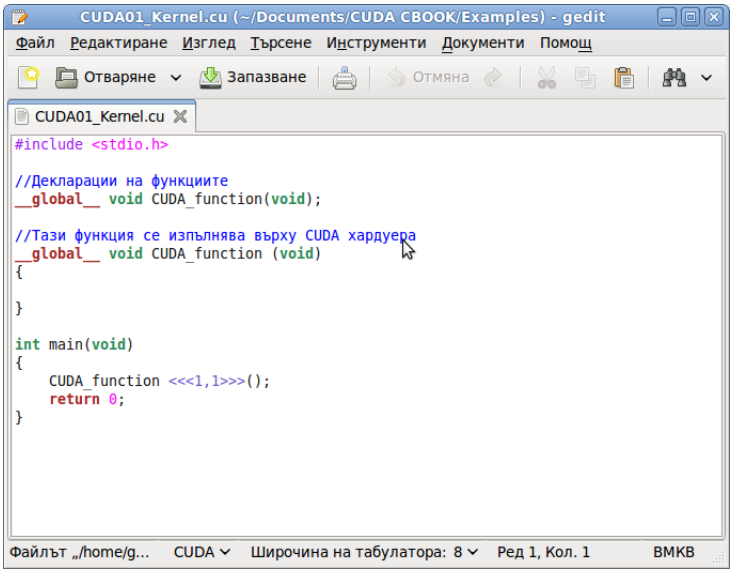
./име-на-изпълнимия файл

Писане на програми с geany

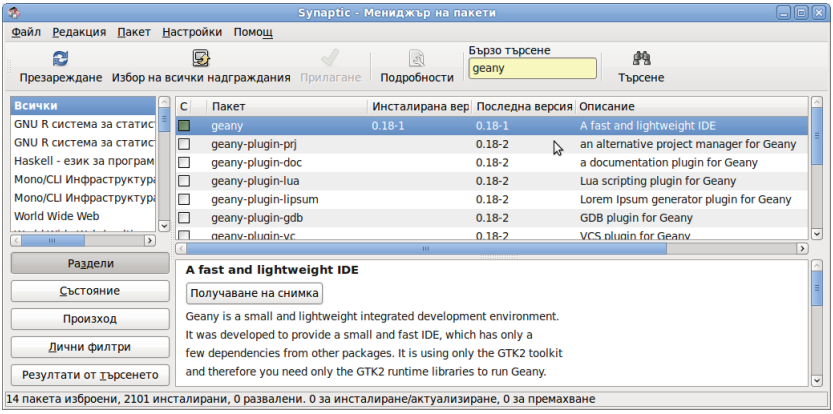
Както видяхте писането и компилирането на код с gedit не е особено вълнуващо, но все пак е нещо. Съществува друга удобна програмка, с която по-лесно може да създавате и управлявате вашите проекти, а също така да компилирате готовите приложения. Тази програмка се нарича geany и може да намерите на сайта: http://www.geany.org/. Тя не се инсталира по подразбиране заедно с Ubuntu, затова трябва да я качим допълнително. За целта ще използваме Synaptic package Manager, който може да стартирате от Systel->Administration->Synaptic package Manager.

От менюто Settings->Repositories следва да добавим нов адрес за проверка на пакети за инсталиране. Изберете опцията Other Software Tab и натиснете Add бутона, като в полето APT Line напишете:
„ppa:ferramroberto/linuxfreedomlucid“. След това пътвърде-

Фигура 3.9: Програмата gedit може да се използва за набиране на сорс кода на вашите програми.



Фигура 3.10: Избор и инсталиране на нови пакети чрез synaptic.



те изборът и се върнете в основния прозорец на мениджъра на пакети. След тази стъпка може да иберете бутона

Reload, като след като тази операция приключи в полето Quick search напишете: „geany“. От отворилият се списък с приложения маркирайте всички geany пакети. За да извършите инсталация иберете бутона Apply. След като това действие приключи програмата geany ще може да отворите през основното меню: Applications->Programming->Geany. Ако всичко мине успешно вие вече имате работна среда за писане на C/C++ програми и CUDA. Ако всичко е приключило успешно може да преминете към писането, компилирането и стартирането на вашите първи CUDA приложения. Характерно за geany е че ви дава възможнсот директно да генерирате вашите приложения от изподния код на програмите и да ги стартирате, без при това да се налага да ползвате едновременно с това и терминала. В зависимост от работния език на операционната ви система менютата на програмата ще са или на английски или на български. Създаването на нов файл става чрез командата File->New или Файл->Нов. Следва да имате навика да давате имена на файлите веднага след тяхното създаване. Кръстете вашият нов файл HelloWorld.cu. За да може вашите програми да бъдат извеждани от програмата с цветен текст, т.н. sintacsis highligting, трябва да отворите мунюто Instruments->Configuration Files->Filetipe_Extensions.conf. В този файл намерете следните реда:

C++=*.cpp;*.cxx;*.c++;*.cc;*.h;*.hpp;*.hxx;*.h++;*.hh;*.C;

И ги модифицирайте по следния начин:

C++=*.cpp;*.cxx;*.c++;*.cc;*.h;*.hpp;*.hxx;*.h++;*.hh;*.C;*.cu;

След тази операция вашият C++ код за CUDA приложенията съхраняван в *.cu файли ще бъде автоматично разпознаван от програмата и оцветяван. Оцветяването на кода ви позволява по-лесно да се ориентирате в изходния код, естествено може да пишете програми и с друг текстов редактор, който няма подобни функционалности. Програмата geany има една допълнителна полезна функционал-

ност, в вертикалния подп прозорец Symbols вие ще може да виждате имената на ползваните в кода функции. Кликвайки върху дадена функция програмата автоматично ще ви навигира до съответното място в кода на програмата, което прави работата с програмния ви код по-бърза. Преди да имате настроена работна среда ще трябва да направим още една промяна. Тъй като geany по подразбиране използва вградения в Ubuntu gcc компилатор ние ще трябва да му укажем да използва nvcc. Това става от менюто Build->Set included files and components или Построй->Задай включени файлове и аргументи. Ще се отвори даденят по-долу прозорец. За компилиране на стандартни CUDA програми без OpenGL и атомарни операции подменете кода в полето Build или Построй на:

```
nvcc -o "\% e" "\% f"
```

След тази промяна направата на вашите CUDA програми може да стане с клавиша F9. Ако в процесът на компилиране възникнат грешки в долния прозорец ще се изведат грешки. Ако всичко приключи успешно може да стартирате вашите програми с клавиша F5. Ако всички тези действия преминат успешно вие вече имате инсталирана и настроена работна среда за всички примери от този учебник. Дали ще пишете програмите с geany или с gedit, като ги компилирате през команден ред оставям на вас. Успешна работа!

Глава 4

Програмиране с CUDA

В тази глава ще научите повече за писането на реални програми изпълнявани върху CUDA съвместим хардуер. В следващите няколко примера ще разберете как се резервират ресурси от паметта на видеокартата, как се описват функциите изпълнявани върху CUDA хардуера, как става прехвърлянето на данни от RAM на компютъра в RAM на видео картата. Ще видите по какъв начин се извикват и изпълняват т.н. kernel функции изпълнявани върху видео картата и как се задават параметрите на изчислителните клъстери и блокове с процеси. В няколко поредни примера ще разберете повече за това как да направите вашите програми по-бързи с CUDA. В главата са разгледани и примери използващи OpenGL за изчертаване на 2D и 1D графики във вашите програми.

4.1 "Hello World from CUDA!"

Почти всички съвременни книги по програмиране започват с програмата „Hello World!“, или нейният еквивалент. За да спазя тази традиция и този учебник ще започне с

нея. Разгледаните тук програми ще бъдат представяни с целия си изходен код, а описанието на функциите ще бъде давано в текста. Пълните изходни кодове на програмите може да намарите на приложения компакт диск или на сайта на вашия курс. В зависимост от времето, с което разполагате, ви препоръчвам да пишете програмите сами без да ги копирате от приложения диск с примери. Правлинят подход обикновенно изисква предварително разглеждане на изходния код на програмата, компилирането на готовия пример, неговото изучаване чрез промени в параметрите на програмата, и последващо написване на кода изцяло наново. Само така ще добиете нежният опит да пишете специфичния за CUDA синтаксис. Тъй като е възможно уважаемият читател да не е запознат с програмирането на C/C++ в края на учебника е дадена отделна глава съдържаща кратко приложение за синтаксиса и структурата на C програмите. За да започнете да ги четете и да се ориентирате в тях е необходимо да се запознаете като минимум с приложената информация. Примерите които ще разглдаме имат фокус към реализацията на уводни паралелни алгоритми с CUDA, фокусирайки се над основните проблеми, които бихте срещнали, когато започнете да пишете сами свои собствени приложения. На всички начинаещи програмисти препоръчвам следния учебник: „C++ Language Tutorial“, на Juan Soulié , който може да намерите на този линк: http://www.cplusplus.com/doc/tutorial.

Структурата на C/C++ програмите които ползвам тук е елементарна. Езикът C е подходящият език за писане на инженерни приложения, в това число системи за управление, обработка на информация, управляващи програми. Голяма част от микроконтролерите и сигналните процесори най-лесно се програмират на C/C++. Не на последно място операционната система Linux в по-голямата сди част е написана на C/C++. CUDA поддържа и други езици,

като: Fortran, Python, Jcuda. Същевременно с това CUDA предлага възможност за интеграция със следните широко използвани инженерни приложения, като: MATLAB, Mathematica, LabVIEW. Подобен тип интеграция все още не е достатъчно бързодействаща. Помнете, че максимално бързодействие от CUDA програмите бихте имали само ако пишете основен код на C/C++ и естествено ползвате възможно по-бърз компютър и видеокарта от по-високо поколение.

Тъй като сега е моментът да започнем, за съвсем начинаещите ще спомена, че всяка C/C++ програма има следните основни компоненти: директиви за пред процесор, главна функция **main()**, други функции, променливи и различни програмни структури за управление изпълнението на програмата. Следващия пример е елементарен, програмата ще извежда съобщението „Hello World!" на екрана на вашия компютърен терминал. За целта създайте нов текстов файл, препоръчвам това да стане с текстовия редактор **geany**, но може да ползвате и **gedit**. Именувайте файла „**HelloWorld_CPU.cu**". След това въведете дадения подолу изходен програмен код. Тази програма съдържа две функции, **main()** и **Hello()**. Обърнете внимание, че само функцията **main()** се описва веднъж. Всички останали функции във вашите C/C++ програми ще следва да обявявате предварително преди да ги използвате, тоест да създавате така наречените прототипи. Същото правило важи и за променливите и масивите от данни, които ще ползвате във вашите програми. Има компилатори, които могат да заобиколят това изискване. Помнете в C програмите всяка функция и/или променлива, която ползвате трябва предварително да бъде обявена, като стриктно посочите нейният тип (не всички езици за програмиране изискват това от вас). Единствено главната функция **main()** не следва да се декларира предварително. В следващите няколко примера ще покажем как да пишем и CUDA kernel функциите

Listing 4.1: Hello World from CPU

```
#include <stdio.h>

void Hello();

int main(void){
    Hello();
    return 0;
}

void Hello(){
  printf("Hello World from CPU!\n");
}
```

Тази програма би могла да се направи и без допълнителната функция **Hello()**, но това е направено целенасочено. В примерите по-долу ще реализираме различни приложни задачи с различни допълнителни функции, които ще извикваме от функцията **main()**. Следващата ни задача, ще бъде реализация на алгоритъм за шифриране на текст изпълняван върху централния процесор (CPU) на компютъра. След това ще добавим функция, която да изпълнява процеса по дешифриране на шифрограмата върху графичната карта (GPU). След това ползвайки специални функции за работа с таймери ще измерим времето нужно за изпълнение на програмите върху CPU и GPU процесорите.

Възниква резонният въпрос, как подобна сложна функционалност да се напише така, че да се изпълнява от програма върху CUDA съвместим хардуер? Във всъщност CUDA не може самостоятелно да се използва за извеждане на съобщения върху екрана на компютъра. Но бихме могли да създадем програмен код, който да дешифрира текстовия низ **„Hello World from CUDA!“**, а след това използвайки стандартната функция **printf()** в основната програма да изведем върху терминала дешифрираното съ-

общение. Този на пръв поглед елементарен пример има за цел да ви запознае с основните програмни конструкции в една типична CUDA програма.

CUDA програмата представлява смесица от C код, който ще се изпълнява върху CPU и GPU на вашия компютър. Това означава, че по някакъв стандартен начин трябва да укажем на компилатора **nvcc** коя част от кода, и/или коя функция, да се изпълни върху CUDA и коя върху CPU. Всяка CUDA функция може да бъде обявена, като: **__global__** и респективно да бъде извиквана от всяка точка на основната C програма. За жалост създаването на CUDA изпълнима функция все още не означава, че тя ще свърши полезна работа. В CUDA се използва графичната видео карта, за да извършваме обемисти еднотипни изчисления върху много данни едновременно. Тези данни обаче предварително трябва да бъдат прехвърлени от RAM на компютъра в RAM на видео картата, наричан глобална видео памет. Поради това, преди да извършим каквото и да е полезно изчисление трябва да заделим и запълним буфер от оперативната памет на компютъра с данни, които да обработваме. Това могат да бъдат данни прочетени от графичен файл намиращ се на твърдия диск, кадри постъпващи от видео камерата или поток от данни постъпващ от интернет. След това да заделим съответния ресурс памет във видео картата (помнете всяка видео карта има строго определен ресурс глобална и локална памет, тази памет е линейно адресируема, но за правилния достъп до нея в режим на работа се налагат специфични синхронизации). След като заделим необходимия обем памет във видео картата трябва да копираме данните от хост компютъра във видео картата. Това е възможно ползвайки готови функции дефинирани в CUDA. След като обработим данните с специално написани **kernel** функции в CUDA най-вероятно ще трябва да ги върнем обратно в паметта

на компютъра. В някои случаи ако става въпрос за графична обработка, CUDA ви дава възможност директно да визуализирате обработените данни върху дисплея, без да се налага тяхното повторно връщане от RAM на компютъра в графичната видео карта. Това се ползва в режим на изчертаване на графика с OpenGL или DirectX. След приключване на изчисленията и запазване на данните ние трябва да освободим заетата видео памет, така че тя да е достъпна за други приложения и функции. Освен това, когато извикваме дадена kernel функция трябва да предадем следните важни параметри: **брой процеси в блок, брой блокове в решетката, брой блокове в клъстер и т.н.**. При това може да предадем, като параметри: указатели към глобална видео памет и стойности на променливи нужни за изчисленията. CUDA функциите могат да бъдат викани само от основната програма, но не могат да викат функции във вашата основна програма. Ето как би изглеждал кода на програмата изпълняващ една функция върху CUDA:

Listing 4.2: Hello World form CUDA

```
#include <stdio.h>

__global__ void CUDA_function(void);

__global__ void CUDA_function (void){
}

int main(void){
    CUDA_function <<<1,1>>>();
    return 0;
}
```

Тази програма по същество не прави нищо съществено. Основното е че в нея имаме декларирана **kernel** функция, която принципно би се изпълнила върху CUDA хардуера, малко по-късно ще и придадем и функционалност:

```
__global__ void CUDA_function(void)
```

Тази функция може да повикаме от основната програма по следния начин във функцията **main()**:

```
CUDA_function <<<1,1>>>()
```

За да имаме повече полза от този пример ще го модифицираме, така че функцията **CUDA_function()** да извършва някаква полезна работа. На тази функция ще предаваме масив от криптирани ASCII символи (American Standard Code for Information Interchange – това е международно признат стандарт позволяващ обмен на текстови символи между компютрите. В него всеки символ се задава с уникална 8 битова последователност, тоест един символ може да се представи с един байт). Този низ от текстови символи ще обособим в масив от типа **char[]**. След като инициализираме масива с текстови символи трябва да запазим съответната памет във видео картата. Преди това трябва да определим обема памет в байтове, която ще ни е необходима. Това ще направим с функцията **sizeof(...)**. След това и ще заделим нужния ни ресурс RAM и ще върнем указател към динамично заделена памет във видео картата с функцията **cudaMalloc()**. Остава ни само да копираме данните от масива в паметта на компютъра в диначино заделения масив в паметта на видео картата. Копирането на данни от хост машината към видео картата става със специална функция **cudaMemcpy()**. Тя приема 4 параметъра: указател към паметта къде да се копират данните, указател към началото на паметта откъдето да се копират данните, брой байтове които да се копират и три "магически" назначения, указващи посоката на копиране на данните:

- **cudaMemcpyHostToDevice** – указва данните да се

копират от RAM на компютъра в RAM на видео картата;

- **cudaMemcpyDeviceToHost** – указва данните да се копират от RAM на видео картата в RAM на компютъра;

- **cudaMemcpyDeviceToDevice** – указва данните да се копират от едно място в RAM на видео картата на друго пак в RAM на видео картата.

След това действие остава още една необичайна процедура, която не е типична за извикането на функциите в C програмите. В първа глава видяхме как CUDA разпределя една изчислителна задача на отделни изчислителни единици наричани процеси. Процесите за паралелна обработка се групират в блокове, които в зависимост от приложението и версията на хардуера могат да бъдат 1, 2 и 3 мерни. Тези блокове се подрежда в изчислителни клъстери. Във всъщност програмистът управлява разпределението на отделните изчислителни процеси в блокове и клъстери. При извикване на **kernel** функциите от основната програма програмистът задава размерите на изчислителния клъстер и блокове в него. Веднъж дефинирана тази настройка не може да бъде променена до приключване на изпълнението на kernel функцията. За да бъде примерът по-изчерпателен ще вземем под внимание дължината на масива: **"Hello World from CUDA"**, която е 22 ASCII символа (или 22 байта). Бихме могли да разбием процеса по декодиране на съобщението използвайки клъстер от 2 отделни блока с по 11 отделни процеса във всеки блок. Тоест всеки един символ ще бъде обработен от един отделен процес за дешифриране. Обикновенно с цел по-лесно модифициране на програмите CUDA предвижда специален тип променлива **dim3** за указване параметрите на изчислителния блок.

Това е така защото блоковете, процесите и клъстерите могат да бъдат едно, дву и тримерни, като в зависимост от конкретната видео карта максимално допустимите димен-сии на тези 3D масиви варират. В конкретния пример зада-ваме едномерна размерност на тези променливи ползвайки типовете променливи **dimGrid(2)** и **dimBlock(11)**. Функ-цията **dimBlock (sizeof(str_h)/2)** приема за параметър резултат върнат от функцията **sizeof(str_h/2)**, която из-числява дължината в байтове на текстовия масив **str_h**. След това CUDA ще извика функцията **__global__ void kernelHello** върху мултипроцесорите на видео картата. Та-зи функция ще извикаме указвайки димeнсиите на изчис-лителния клъстер и блоковете с процеси, които ще са ни нужни за приключване на изчислението:

```
bf kernelHello <<<dimGrid, dimBlock>>> (str_d,
    sizeof(str_h))
```

Обърнете внимание, че функциите изпълнявани върху видео картата могат да приемат следните параметри: указа-тели към масиви в глобалната видеопамет и променливи. Kernel функциите могат да манипулират единствено дан-ни записани в паметта на видео картата. Резултатите от работата на kernel функциите се записват отново и един-ствено в паметта на видео картата. След приключване на изчисленията данните следва да се прехвърлени в паметта на хост компютъра чрез функцията:

```
cudaMemcpy(str\_h, str_d, size,
    cudaMemcpyDeviceToHost)}
```

След приключване на това действие вече може да освобо-дим динамично заетата памет във видео картата чрез фун-кцията: **cudaFree(str_d)**. В практиката CUDA се ползва за обработка на много еднотипни масиви данни, например видео кадри, изображения и др. В тези случаи е удобно и

практично да се ползват веднъж заделени масиви памет във видео картата и RAM на компютъра. Това ще спести време нужно за непрекъснато резервиране и освобождаване на русерси от паметта на системата. При необходимост от особено бърза обработка, копиране на огромни масиви данни в практиката се ползват някои трикове, чрез които програмистът може да фиксира виртуалните русерси RAM в компютъра правейки така обмена на данни между видеото картата и паметта на системата многократно по-бързи. За това ще стане дума по-късно. За да не губим ценно време по-долу е даден листинга на цялата програма.

Listing 4.3: Шифриране на текст в CUDA

```
#include <stdio.h>
void Hello();
__global__ void kernelHello(char* str, int N,
    char code);

__global__ void kernelHello (char* str, int N,
    char code) {
       int idx = blockIdx.x * blockDim.x +
           threadIdx.x;
       if(idx<N)
        str[idx] = str[idx] xor code;
}

int main(void) {
     char code = 0x21;
     char str_h[] = "Hello World from CUDA!";
     for(int i = 0; i<22; i++)
       str_h[i] = str_h[i] xor code;

     char *str_d;
     size_t size = sizeof(str_h);
     cudaMalloc((void**)&str_d,size);
     cudaMemcpy(str_d, str_h, size,
         cudaMemcpyHostToDevice);
     dim3 dimGrid(2);
     dim3 dimBlock(sizeof(str_h)/2);
```

```
    kernelHello <<<dimGrid,dimBlock>>>(str_d,
        sizeof(str_h), code);
    cudaMemcpy(str_h, str_d, size,
        cudaMemcpyDeviceToHost);

    printf("%s\n",str_h);
    cudaFree(str_d);
    return 0;
}
```

Какво се случва в "__global__ void kernelHello"?

Ако това е вашата първа CUDA програма, най-вероятно разгадаването на кодът **kernelHello(...)** функцията ви изглежда, като китайски йероглиф. Както казахме при активиране на функции, които ще се изпълняват върху видео картата предварително указваме димесиите на изчислителния клъстер. При повикването на функцията ние предадохме следните параметри: **dimGrid=2** и **dimBlock=11**.

```
kernelHello <<<dimGrid, dimBlock>>>(str_d, sizeof
    (str_h), code)

__global__ void kernelHello (char* str, int N,
    char code)
{
        int idx=blockIdx.x*blockDim.x+threadIdx.x;
        if(idx<N)
            str[idx]=str[idx] xor code;
}
```

При извикване на kernel освен размерите на изчислителната решетка трябва да предадем аргументите на функцията: указател към началния адрес на паметта в се съдържат данните и броя байтове, които ще обработваме. Освен това, като параметър подаваме и кодът за декодиране на съобщението: **code=0x21**. CUDA ви позволява да обработвате 3D клъстери съдържащи 3D блокове с процеси, като това се поддържа от изчислителни архитектури

2.X. Дефинирайки параметри на изчислителния клъстер ние указваме на драйвера да разпредели изчислението в 2 блока съдържащи по 11 процеса. За да определим номера на процеса, и съответно да можем да определим коя данна от масива да обработим и къде да запишем резултата, CUDA предвижда ползването на следните индекси автоматично изчислявани за всеки един процес, те може да са 1, 2 и 3 мерни в зависимост от изчислителната задача и възможностите на хардуера:

- **blockIdx** – това е номера на обработвания блок с процеси. Когато процесите се разполагат в 2D решетка те могат да се индексират по: blockIdx.x и blockIdx.y, а в 3D решетка по: blockIdx.x, blockIdx.y и blockIdx.z. Адресирането на блоковете става, както адресирането на елементите на 1D, 2D или 3D матрица от стойности;

- **blockDim** – връща размерността на блока в произволно избрано направление: blockDim.x, blockDim.y и blockDim.z. В зависимост от архитектурата се допускат максимален брой процеси в един блок, както и максималните стойности на измеренията на този блок;

- **threadIdx** – връща индекса на всеки един процес в един изчислителен блок. Максималната стойност на този индекс не може да превишава съответно размерността на блока (blockDim) по координатите: x, y и z.

Тези променливи указват поредния номер на изпълнявания блок, размерностите на блока и номера на изпълнявания процес в блока. По принцип тези променливи съдържат 3 димeнсии: x, y, z, но за нуждите на първия пример ние ще обработваме едномерни масиви с данни. Умножавайки номера на поредно изпълнявания блок по размера на

елементарния изчислителен блок и сумирайки това произведение с поредния номер на процеса изпълняван в съответния блок ние можем точно да определим позицията **idx**, тоест номера на процеса който изпълняваме. Този индекс ползваме, за да адресираме поредния елемент от масива с данни от който ще четем данни нужни за изчислителния процес и на който можем да върнем получения резултат от изчислението. Тъй като CUDA няма да спре обработката до изчерпване на последния подаден изчислителен блок и процес в него, в кода на функцията извършваме непрекъсната проверка за това дали поредно изпълнявания процес не излиза извън рамките на масива с данни. Това става с проверката: **if(idx<N)**, ако условието не се изпълнява изчислението се прекратява. Следващия ред представлява самият процес по дешифриране: **str[idx] = str[idx] xor 0x21**. Всичко което трябва да направим е да компилираме и стартираме програмата. Както виждате мигрирането на една C програма върху CUDA е достатъчно лесно занимание, но в някои случаи може да е предизвикателство. Във всъщност това е далеч от ефективната и бърза програма, но в общия случай би ви донесло поне двойно по-високо бързодействие в сравнение с класическа програма изпълнявана върху централния процесор на компютъра. Вашата програма ще бъде толкова по-ефективна, колкото по-голям обем данни обработвате и колкото повече мултипроцесори има във вашата видео карта.

В какво се крие разликата на kernel с класическата C програма?

Ако трябваше да решим проблемът с дешифриране на 22 байтовия масив от символи по класически метод в една C програма, най-вероятно бихме ползвали цикъл при който последователно да обходим всеки един елемент от масива и да го дешифрираме изпълнявайки операцията изключва-

що или (xor) върху него и записвайки резултата в изходния масив със стойности. Това би изглеждало така:

```
for(int i=0; i<22; i++)
    str[i] = str[i] xor code;
```

Този цикъл е еквивалентен на последователното изпълнение на следния обемист код:

```
str[0]  = str[0]  xor code;
str[1]  = str[1]  xor code;
.......................
str[21] = str[21] xor code;
```

Проблемът с бързодействието тук не е свързан особено с начина на написване на този код, естествено ако трябва да бъдем коректни изпълнението на цикъла ще отнеме малко повече време в сравнение с последователното изпълнение на 22 реда код, но пък програмата ще стане по-голяма и трудна за четене особено ако цикълът има за цел да обходи по-голям масив. За разлика от класическата C програма, CUDA ще извърши именно това, което дадохме по-горе. Всеки един елемент на масива с данни str[] ще бъде обработен от отделен изчислителен процес. Съществената разлика тук, е че всеки един процес ще се изпълнява в рамките на предварително обособения изчислителен блок. В нашия случай броят блокове е 2, а броят процеси във всеки един от тях е 11. Нашият код ще прилича на 22 копия паралелно изпълняващи се процеса, които изглеждат така:

```
int idx = blockId.x * blockDim.x + threadIdx.x;
str[idx] = str[idx] xor code;
```

Ползвайки индекса на текущо изпълнявания процес в активния блок ние можем да адресираме елементите на масива, който обработваме. Това е еквивалентно на едновременното изпълняване на две копия на 11 отделни процеса,

Таблица 4.1: 2 блока с по 11 процеса се изчисляват паралелно

block 0	**block 1**
int idx = 0*21+0; str[idx] = str[idx] xor code;	int idx = 1*21+0; str[idx] = str[idx] xor code;
int idx = 0*21+1; str[idx] = str[idx] xor code;	int idx = 1*21+1; str[idx] = str[idx] xor code;
int idx = 0*21+2; str[idx] = str[idx] xor code;	int idx = 1*21+2; str[idx] = str[idx] xor code;
int idx = 0*21+3; str[idx] = str[idx] xor code;	int idx = 1*21+3; str[idx] = str[idx] xor code;
int idx = 0*21+4; str[idx] = str[idx] xor code;	int idx = 1*21+4; str[idx] = str[idx] xor code;
int idx = 0*21+5; str[idx] = str[idx] xor code;	int idx = 1*21+5; str[idx] = str[idx] xor code;
int idx = 0*21+6; str[idx] = str[idx] xor code;	int idx = 1*21+6; str[idx] = str[idx] xor code;
int idx = 0*21+7; str[idx] = str[idx] xor code;	int idx = 1*21+7; str[idx] = str[idx] xor code;
int idx = 0*21+8; str[idx] = str[idx] xor code;	int idx = 1*21+8; str[idx] = str[idx] xor code;
int idx = 0*21+9; str[idx] = str[idx] xor code;	int idx = 1*21+9; str[idx] = str[idx] xor code;
int idx = 0*21+10; str[idx] = str[idx] xor code;	int idx = 1*21+10; str[idx] = str[idx] xor code;

като по този начин всичките 22 байта ще бъдат дешифрирани едновременно.

Когато разработвате CUDA програми параметрите на броят процеси, размерност на блока и броят процеси изпълнявани едновременно от даден блок могат да бъдат важни за правилното подбиране размерността на изчислителния клъстер. Това би могло да стане чрез ползване на следния инструмент: **CUDA Occupancy Calculator - NVIDIA**, потърсете го в интернет. Това е ексел файл, чрез който може да подберете оптимални димeнсии за изчислителния клъстер според архитектурата на видеокартата с която работите. Крайната цел на всяка CUDA програма е тя да бъде така оптимизирана, че наличния брой мултипроцесори да

бъдат равномено натоварени, за да може изчислителната задача да приключи по-бързо.

4.2 Измерване на бързодействието на програмата

Този пример ще покаже как да измерваме времето за изпълнението на дадена функция от вашата основна програма и времето необходимо за изпълнение на даден kernel в CUDA. Определянето на времето за изпълнение на функциите в основната програма може да стане елементарно чрез извикване на функцията **clock()** включен в <**time.h**>, която ползва вградените функциите на операционната система. Тази функция работи коректно в Linux но не и в Windows. Функцията **clock()** връща абсолютното време в микросекунди, което е изтекло от стартирането на програмата. За да определим времето за изпълнение на определен сегмент от кода трябва да извикаме **clock()** в началото на изпълнение на функцията, като запазим полученото време в променлива, и след приключване на изследваните функции отново да повикаме **clock()**. Изваждайки полученото време в началото на повикване на функциите от полученото време в края на изпълнението им ще получим абсолютното време в микросекунди което е изтегкло от началния момент на изпълнение на нашия програмен код до крайния момент на приключването му.

Важно е да знаете, че ако искате да измерите времето за изпълнение на под програма или CUDA kernel с функцията **clock()** тя няма да ви върне коректен резултат. Това е така защото таймерът на програмата е активен и отброява тактовите импулси само ако програмата, която го вика е активна. За да измерим времето необходимо за изпълнение на даден CUDA kernel ще ползваме специалните

функции **CUDA - cudaEvent...(...)**. За да направим примерното приложение по-смислено ще натоварим системата с повече изчисления. В този пример ще извършим сумиране на два масива от целочислени стойности (вектори) и ще запишем изходните данни в трети масив. След това ще измерим времето нужно за изпълнение на тази функция върху стандартния процесор и после в CUDA. Сравнявайки двете времена ние можем да определим с прецизност до микросекунда дали нашият CUDA код е по-бърз и до колко е по-ефективен.

Освен функцията **clock()** в операционните системи има и допълнителни вградени функции, които може да ползвате за определяне времето за изпълнение на дадена функция, а също така и за синхронизиране между отделните независимо изпълнявани функции (или групи функции) наричани процеси и потоци. Ползвайки ги вие можете да дефинирате паралелно изпълнявани блокове от код във вашите програми. Изпълнението на "паралелен" код върху CPU е в т.н. псевдо паралелен режим, освен в случаите когато не стартирате програмата върху многоядрен процесор и когато операционната система поддържа такова изпълнение. За програмите, които работят върху CPU нов процес може да бъде стартиран от основната програма, като при това тя може да изчака до неговото приключване или да завърши преди дъщерния процес да е приключил. Всеки един процес може да бъде поставен в режим на изчакване за възникване на дадено събитие. Например прекъсване, импулс от таймер или сигнал предаван чрез т.н. семафори. Процесът обикновенно може да се изпълни еднократно или многократно до изпълнение на предварително зададено условие за неговото преустановяване. Многозадачното програмиране е ефикасно и може да бъде полезно при разработка на програми за масивна паралелна обработка на данни, в които бихте искали вашата основна програма да

продължава да изпълнява различни задачи докато CUDA kernel се изпълнява върху вашата видеокарта. По-подробно за това как да пишете многозадачни програми изпълнявани върху CPU ще намерите в Глава 6. За да измерите времето за изпълнение на последователно изпълняван ков във вашата основна C програма, може да използвте функцията **clock()** по следния начин:

Listing 4.4: Измерване скоростта на основната програма с таймери.

```
#include <time.h>
...
clock_t h_start, h_stop;
h_start = clock();
...
for(int i = 0; i<size; i++) {
  c1[i] = a[i] + b[i]; }
...
h_stop = clock();
h_stop = h_stop-h_start;
printf("CPU time: %f us\n",(float)h_sto));
```

Пример с множество процеси ще разгледаме по-късно, а тук ще се спрем основно на това да измерим бързодействието на даден CUDA kernel. Този пример е полезен и то друга гледна точка. В него по-добре ще добиете представа за изчислителните възможности на CUDA по отношение на максималния брой процеси, които могат да бъдат активни в един изчислителен клъстер. Всеки клъстер може да има размерност 65536 x 65536 x 1 (като следващите версии на видео картите може да поддържат и тримерни клъстери). Всеки един блок в изчислителния клъстер може да има сумарен брой процеси не по-голям от 512, или повече в зависимост от хардуерната версия. Тъй като блоковете могат да имат триизмерни структури, общия брой процеси намиращ се в тях не може да надвише 512 за версия до 1.3, а при версия 2.X този брой е 1024 процеса в блок. Например

дименсиите на един блок за архитектура 1.1, 1.2 и 1.3 могат да бъдат 8x8x8 = 512 или 16x16x2 = 512, но не и 16x16x4 = 1024, което е валидно за архитектура 2.X. Имайте предвид, че за да стане този код по-ефикасен особено при обработка на големи (гигабайтови) масиви от данни се правят допълнителни трикове, като ползването на атомарни операции и споделената памет в мултипроцесорите.

За да изчислим времето което вашият CUDA код отнема нещата стоят малко по-сложно. За щастие CUDA предвижда готови функции, които ни дават нужната функционалност. За разлика от **clock()**, която връща времето с точност до 1 микросекунда, в CUDA точността е 0.5 микросекунди. Естествено подобна прецизност не е от особено значение, но е полезна. При масивните изчисления обикновенно времето за изпълнение на едно изчислително ядро, предвид това че изчисленията са обемни обикновенно е десетки микросекунди и дори секунди или минути. Има нещо особено важно за CUDA, бързината на кратките алгоритми, независимо от това колко малко време отнема изчислението им върху видео картата, са лимитирани и от скоростта на системната шина за обмен на данни между компютърната RAM и видео паметта. Това значи, че е резонно за едно правилно сравнение на резултатите да вземете предвид не само времето за изпълнение на изчислителното ядро с процеси, но и времето нужно за копиране на данните от RAM на компютъра в RAM на видео картата и обратно. Това обославя и някои особенсти от ползването на CUDA. Добре би било да прехвърлите възможно повече данни на един път в паметта на видео картата, да извършите възможно повече изчисления върху тях и чак тогава да върнете резултатите в паметта на компютъра. Примерът тук е далеч от това оптимално използване, но основната ни цел е да натоварим графичния процесор, за да видим какво бързодействие ще ни предостави той. Нещо повече, този

пример дава възможност да експериментирате с възможностите за разлини варианти на брой изчислителни блокове в клъстер и брой процеси в тях (примерът показва работа с едномерни масиви от данни). Респективно удобно би било размерът на блоковете да е също едномерен, размерът на клъстера също. Това ще доведе до следните ограничения, нашият код ще може да изпълни максимално 65536 блока с по 512 процеса в тях. Това е еквивалентно на възможността да обработим линеен масив от 33,542,144 единични данни. По-долу ще извършим модификация на алгоритъма, така че да може да обработваме повече данни в клъстера, правейки го двумерен. Първо ще създадем две допълнителни функции с които ще може да измерим времето в нашите CUDA програми. Ще създадем нов файл **"CUDA_Timer.cu"**, функциите от който ще използваме в другите примери за определяне времето за изпълнение на даден CUDA kernel:

Listing 4.5: Измерване скоростта на CUDA програма с CUDA таймери.

```
#include <stdio.h>

void CUDA_Timer_Start(cudaEvent_t* t_start,
    cudaEvent_t* t_stop);
float CUDA_Timer_Stop(cudaEvent_t* t_start,
    cudaEvent_t* t_stop);

void CUDA_Timer_Start(cudaEvent_t* t_start,
    cudaEvent_t* t_stop) {
  cudaEventCreate(t_start);
  cudaEventCreate(t_stop);
  cudaEventRecord(*t_start, 0);
  printf("Timer GPU start!\n");
}

float CUDA_Timer_Stop(cudaEvent_t* t_start,
    cudaEvent_t* t_stop) {
  float elapsedTime = 0;
```

```
    cudaEventRecord(*t_stop, 0);
    cudaEventSynchronize(*t_stop);
    cudaEventElapsedTime(&elapsedTime, *t_start, *
        t_stop);
    cudaEventDestroy(*t_start);
    cudaEventDestroy(*t_stop);
    printf("Time GPU: %f ms\n",elapsedTime);
    return elapsedTime;
}
```

За да използвате този файл многократно във вашите програми следва да го съхраните в папката на вашия текущ проект. За да ползвате функциите декларирани в него, трябва да го включите в основната програма с директивата:
#include "CUDA_Timer.cu"
Напишете я в началото на програмата под основните декларирани хедър файлове (под "stdio.h" например). Това указва на компилатора, че ще ползвате функциите дефинирани в зададения файл. Преди да ползвате функциите: **CUDA_Timer_Start(...)** и **CUDA_Timer_Stop(...)** трябва да дефинирате две нови член променливи, които ще ползваме за да съхраняваме текущите стойности на таймера. Направете това в началото на програмата по следния начин: **cudaEvent_t start, stop**. Функцията **CUDA_Timer_Stop(...)** има вградена опция да отпечатва в терминала изтеклото време между нейното стартиране и спиране. В случай че не желаете тази функсионалност трябва да я коментирате с две наклонени черти кода:

```
// printf("Time GPU: %f ms \n",elapsedTime);
```

За да измерите времето на един kernel трябва първо да извикате функцията: Timer_Start() предавайки и за параметър адресите на двете променливи за съхраняване на времето от таймера. За да получите колко време е изминало

между стартирането и приключването на kernel функцията трябва да ползвате функцията Timer_Stop().

```
float elapsedTime;
cudaEvent_t start, stop;

CUDA_Timer_Start(&start, &stop);
kernel <<<dimGrid,dimBlock>>>(...);
elapsedTime = CUDA_Timer_Stop (&start, &stop);
```

След тази операция в променливата **elapsedTime** ще се намира стойността на времето за изпълнение на вашето ядро в милисекунди. Тузи функции могат да бъдат ползвани само и единствено за прецизно измерване на времето за изпълнение на kernel функции, но не и за функции върху хост компютъра.

Пълен код на програмата

Listing 4.6: Използване на CUDA таймери във вашия код.

```
#include <stdio.h>
#include "CUDA_Timer.h"

#define S (60 * 1000 * 1000)
#define BLOCKX (32)
#define BLOCKY (1)
#define GRIDX (32)
#define GRIDY (65535)

__global__ void dev_Add(int* a, int* b, int* c,
    int size) {
int idx = (blockDim.x*blockIdx.x+threadIdx.x);
int idy = (blockIdx.y*blockDim.x*gridDim.x);
int id = idx+idy;
if(id < S){
   c[id] = a[id] + b[id];
}}

int main(void) {
```

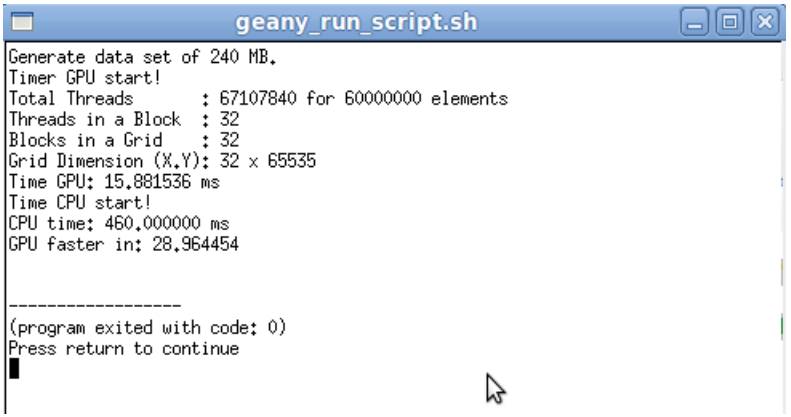
```
int *a, *b, *c, *c1;
int *dev_a, *dev_b, *dev_c;
int size = S;
a = new int [S];
b = new int [S];
c = new int [S];
c1 = new int [S];

printf("Generate data set of %d MB.\n",sizeof(int
    )*size/1000000);
for(int i = 0; i<S; i++){
   a[i] = i;
   b[i] = i;
   c[i] = 0;
}
int s = sizeof(int) * S;
cudaMalloc((void**)&dev_a,s);
cudaMalloc((void**)&dev_b,s);
cudaMalloc((void**)&dev_c,s);

cudaMemcpy(dev_a,a,s,cudaMemcpyHostToDevice);
cudaMemcpy(dev_b,b,s,cudaMemcpyHostToDevice);
cudaEvent_t start,stop;
CUDA_Timer_Start(&start,&stop);
dim3 dimGrid(GRIDX,GRIDY);
dim3 dimBlock(BLOCKX,BLOCKY);

printf("Total Threads: %d for %d elements\n",
    dimGrid.x*dimBlock.x*dimGrid.y*dimBlock.y,S);
printf("Threads in a Block: %d\n",dimBlock.x);
printf("Blocks in a Grid: %d\n",dimGrid.x);
printf("Grid Dimension (X.Y): %d x %d\n",dimGrid.
    x,dimGrid.y);

dev_Add <<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c,
    S);
float elapsedTime=CUDA_Timer_Stop(&start, &stop);
cudaMemcpy(c,dev_c,s,cudaMemcpyDeviceToHost);

clock_t h_start, h_stop;
printf("Time CPU start!\n");
h_start = clock();
```

```
for(int i=0; i<S; i++){
   c1[i]=a[i]+b[i];
}
h_stop = clock();
printf("CPU time: %f ms\n",(float)(h_stop-h_start
   )/1000);
printf("GPU faster in: %f\n",(float)(h_stop-
   h_start)/(1000*elapsedTime));

int count = 0;
for(int i = 0; i<S; i++){
   if(c1[i]!=c[i]){
      printf("ER[%d],c1=%d,c=%d\n",i,c1[i],c[i]);
      count ++;
   if(count >10)
      i = S;
}}
delete[] a;
delete[] b;
delete[] c;
delete[] c1;
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```

След като успешно компилирате и изпълните тази програма би следвало да получите следния подобен текст върху терминала Фиг. 4.1. Отделете време и експериментирайте с различни размери на масивите с данни и дименсиите на изчислителните блокове. Направете експерименти променяйки стойностите на броя процеси изпълнявани в един блок, и респективно броя колони в клъстер. Имайте предвид, че за правилно изпълнение на изчислението броят процеси следва да бъде по-голям или поне равен на броят елементи от масива с данни, които искаме да обработим. Може да експериментирате с различни видове изчисления: събиране, изваждане, умножение, деление, тригонометрични функции и да наблюдавате как това ще се отрази на бързодейс-

твието на вашия CUDA kernel сравнимо с времето нуждо за същото изчисление върху CPU на компютъра.

Фигура 4.1: Измерване на бързодействието на вашия код с "CUDA_Timer.h".



4.3 Визуализация на 2D изображения с OpenGL

В този пример, ще научите как се създават и визуализират 2D изображения в Linux използвайки OpenGL (Open Graphics Library). Разгледания пример показва имплементирането на по-сложни алгоритми върху GPU платформата. За щастие обаче Linux ни предоставя серия библиотеки, като OpenGL и Open CV с които работата с 2D и 3D изображения е елементарен. Този пример има за цел да създаде отделен помощен файл позволяващ ви да изобразявате 2D матрици с байтови стойности (сиви - полутонови изображения). В началото на учебника стана дума за това как да се инсталира OpenGL средата за разработка върху Ubuntu. Възможно е различните версии на Ubuntu да изис-

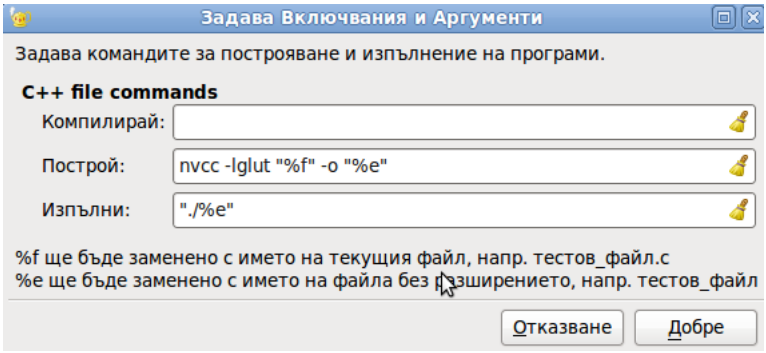
кват различна инсталация, ако имате проблеми проверете това в google. Ако все още не сте инсталирали OpenGL сега е момента да го направите чрез следната команда:

```
apt-get install freeglut3-dev libxi-dev libxmu-dev
```

Ако вече сте направили това нека пристъпим към разработката на нашия помощен файл за изобразяване на 2D сиви полутонови изображения с OpenGL. Преди компилацията на OpenGL приложения с geany имайте предвид да извършите настройките в менюто build Фигура 4.2 , като зададете следния команден ред за направа на приложението (когато ще го използвате в CUDA програми ще ползваме компилатора nvcc, а в обикновенни случаи g++). Сега geany ще създава вашите приложения с натискането на клавиш F9, а стартирането им с F5.

```
nvcc -lglut \%f -o \%e

g++ -lglut \%f -o \%e
```

Фигура 4.2: Настройка на geany.



За да визуализираме 2D изображения с OpenGL ще използваме следния пример, който ще рисува цветни петна

на екрана, като при натискане на левия или десния бутон на мишката размерът и цвета на петната ще се променя. При реализацията на научни и инженерни приложения, изходните данни от дадено изчисление често трябва да бъдат визуализирани в 2D или 3D графики. Един бърз и сравнително елементарен начин за това може да бъде ползването на OpenGL. За да ползвате OpenGL във вашите приложения вие трябва да включите следните хедър файлове във вашия проект: **GL/freeglut.h** и **GL/gl.h**, като освен тях ще се налага да ползвате и **stdlib.h** и **stdio.h**, както и специфични хедър файлове за CUDA приложенията.

В по-долния пример ще извършим няколко неща: ще използваме функцията main(), за да инициализираме OpenGL и да зададем параметрите на прозореца, в който ще изобразяваме дадено изображение. За това ще ползваме функции на OpenGL специално предназначени за управление на прозорците. Тези функции са:

- **glutInit (int *argc, char **argv)** - инициализира Graphics Library Utility Toolkit (GLUT) и обработва възможните параметри подадени от командния ред (що се касае до Linux X графичния сървър, това може да са опции за това какъв е екрана и каква геометрия ще се ползвайки Функцията glutInit() трябва да се извиква всеки път преди която и да е друга GLUT функция.

- **glutInitDisplayMode (unsigned int mode)** - ни позволява да зададем формата на ползвания цветен модел: RGBA или цветен индексен. Също така може да се укаже дали ще се ползва единично или двойно буферизиран прозорец. (В случаите когато се работи с цветен индексен модел, трябва да се заредят и съответните цветове в т.н. цветна карта. За това се ползва функцията glutSetColor()). Тази функция

може да се ползва и за указване на допълнителни параметри на прозореца: depth, stencil, multisampling, and/or accumulation buffer. Например, ако искаме да създадем прозорец, който е двойно буферизиран, ще се ползва RGBA цветния модел ще се извика следния параметър: glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH).

- **glutInitWindowPosition (int x, int y)** - указва позицията на горния ляв ъгъл на прозореца спрямо горния ляв ъгъл на екрана.

- **glutInitWindowSize (int width, int height)** - задава размера на прозореца в пиксели,

- **glutInitContextVersion (int majorVersion, int minorVersion)** - определя коя версия на OpenGL желаем да ползваме. (Това е нова опция валидна само за Freeglut, и е въведена с OpenGL версия 3.0).

- **glutInitContextFlags (int flags)** - ни позволява да укажем OpenGL контекста, който желаем да ползваме. За нормална работа с OpenGL не е задължително предаването на този параметър. Ползването на функцията може да е обосновано ако желаете приложението ви да е съвместимо с нови верси на OpenGL контекст.

- **int glutCreateWindow (char *string)** - създава прозорец с OpenGL контекст. Тази функция връща уникален идентификатор на новия прозорец. Имайте предвид, че до извикване на функцията glutMainLoop() този прозорец няма да бъде визеялизиран.

Освен тези функции OpenGL ви позволява да използвате и други заготовки на функции, които се изпълняват за

изпълнение на определени задачи: изчертаване на прозореца, функции за обработка на задачи, когято OpenGL не е зает с графична обработка, а също така и функции улесняващи работата и взаимодействието на потребителите с вашите програми - мишка и клавиатура. Тези функции са:

- **glutDisplayFunc(void (*func)(void))** е най-важната функция, която ще използвате. Когато GLUT определи, че контекста на прозореца трябва да бъде изобразен наново, тази функция ще бъде извикана. Функцията се регистрира при изпълнение на: glutDisplayFunc(). Следователно в тази функция следва да се изпълнят всички под функции, които са необходими за преизчертаване на екранния изглед или сцена. Ако програмата измени контекста на прозореца, понякога ще извиквате функцията glutPostRedisplay(), която се предава на glutMainLoop() задание да се извика преизчертаване при първа възможност.

- **glutIdleFunc(void (*func)(void))** - може да бъде използвана за регистриране на функция, която ще се изпълнява когато няма други събития, които OpenGL и вашата програма няма да отработват. Това може да бъде полезно при изпълнение на непрекъсната анимация или обработка на фона на изображението при комплексни сцени с повече движещи се обекти.

Следните функции могат да бъдат ползвани за регистрация на функции, които магат да бъдат викани от OpenGL във вашата програма при възникване на специфични събития:

- **glutReshapeFunc(void (*func)(int w, int h))** - индикира какво действие трябва да бъде изпълнено когато прозорецът променя своите размери.

- **glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))** - ви позволява да свързвате входа от клавиатурата и в зависимост от натиснатия клавиш да изпълните адна или друга операция във вашата програма.

- **glutMouseFunc(void (*func)(int button, int state, int x, int y))** - свързва определена функция за отработване на събитията свързани с натискане и освобождаване бугоните на мишката.

- **glutMotionFunc(void (*func)(int x, int y))** - регистрира функция, която ще бъде активирана когато мишката се премества докато е натиснат неин бутон.

За нашият елементарен пример, тези начални познания за основните функции, които правят вашата OpenGL програма интерактивна са достатъчни. В следващия пример ще ползваме някои от тези функции, за да покажем как с OpenGL ще можем да изобразяваме 2D повърхности, под формата на цветни или сиви полутонови изображения. В практиката това е често срещана задача, когато даден резултат от симулация или математическо изчисление генерира графикя, която показва нагледно резултата от изпълнението на дадена функция. OpenGL може да работи независимо дали на вашата система има инсталиран графичен адаптер поддържащ хардуерно ускорение или тези задачи ще се изпълнят от централния процесор на компютъра. В нашия случай, когято работим с CUDA съвместим хардуер винаги ще ползваме хардуерното ускорение. Съществува възможност да ползваме OpenGL съвместно с CUDA, така че да не се налага данните, които ще изобразяваме да бъдат връщани обратно в паметта на компютъра и предавани за изчертаване от OpenGL, който отново ползва графичната карта на системата. Подобен пример ще раз-

гледаме по-късно. Дадения по-долу листинг на програмен код ще рисува цветни петна.

Listing 4.7: Изобразяване на 2D изображения с OpenGL.

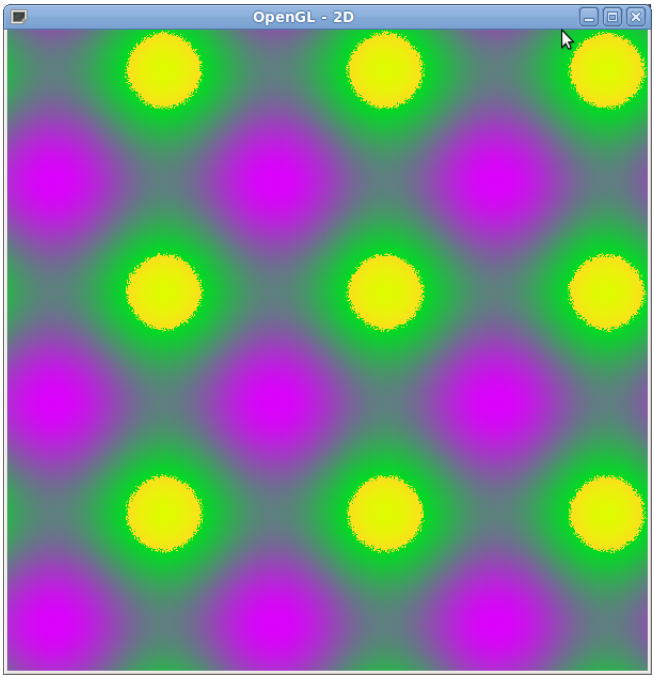
```
#include <stdio.h>
#include "GL/freeglut.h"
#include "GL/gl.h"
#include <math.h>

#define WIN_X 600
#define WIN_Y 600
//g++ -lglut "\%f" -o "\%e"
unsigned int* pixels;
int win_x;
int win_y;
int step;

void GL_Display();
void GL_Mouse(int button,int state,int x,int y);
void GL_Key(unsigned char key,int x,int y);
void GL_Reshape(int w, int h);
void GL_Idle();
void GL_Init();
void GL_Init(void(*draw_surface)(),int X,int Y,
    unsigned int* pixels);
void Draw();

int main(int argc, char** argv){
    step = 30;
    pixels = new unsigned int[WIN_X * WIN_Y];
    win_x = WIN_X;
    win_y = WIN_Y;
    Draw();
    int c=1;
    glutInit( &c, argv );
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA);
    glutInitWindowSize(WIN_X,WIN_Y);
    glutInitWindowPosition(100,100);
    glutCreateWindow("OpenGL - 2D");
    GL_Init();
    glutDisplayFunc(GL_Display);
```

```
    glutIdleFunc(GL_Idle);
    glutMouseFunc(GL_Mouse);
    glutKeyboardFunc(GL_Key);
    glutMainLoop();
    delete[] pixels;
    return 0;
}

void Draw() {
    unsigned char clr = 0;
    double X, Y = 0;
for(int x=0;x<win_x;x++)
    for(int y=0;y<win_y;y++){
        X = (double)x;
        X = 64+64*sin((rand()\%5+win_x/2-X)/step);
        Y = (double)y;
        Y = 64+64*sin((rand()\%5+win_y/2-Y)/step);
        clr = (unsigned char)(X+Y);
        if(clr>250)
            clr = clr -rand()\%5;
        if(clr<10)
            clr = clr + rand()\%5;
        pixels[x+y*WIN_X]=((0x00000000+clr)<<16)
            +((0x000000FF-clr)<<8)-step+clr;
}}

void GL_Display() {
    glClear( GL_COLOR_BUFFER_BIT);
    glDrawPixels( win_x,win_y,GL_RGBA,
        GL_UNSIGNED_BYTE,pixels);
    glutSwapBuffers();
    glFlush();
}

void GL_Init() {
    glClearColor( 1.0, 1.0, 1.0, 0.0 );
    glClear( GL_COLOR_BUFFER_BIT );
}

void GL_Idle() {
    Draw();
    glutPostRedisplay();
```

```
}

void GL_Mouse(int button,int state,int x,int y){
switch(button){
    case GLUT_LEFT_BUTTON:
        if(state == GLUT_DOWN)
            step ++;
    break;
    case GLUT_RIGHT_BUTTON:
        if(state == GLUT_DOWN)
            step --;
    break;
    default:
    break;
}}

void GL_Key( unsigned char key, int x, int y ){
    switch (key) {
        case 27:
            exit(0);
            break;
}}
```

При натискане на левия или десния бутон на мишката, чрез функцията GL_Mouse(), управляваме параметрите на изображението. При натискане на бутона Esc от клавиатурата се активира функцията GL_Key(), която проверява ASCII кода на натиснатия бутон и ако той съвпада с кода на Esc бутона, се изпълнява затваряне на прозореца. Този пример показва как може да използвате OpenGL, за да управлявате действието на вашата програма според действията на потребителя. По подобен начин можете да активирате една или друга функционалност на вашия код следейки за натискането на определени клавиши, бутони и движение на мишката, действия с джойстик и други входни устройства.

Фигура 4.3: Екранен изход на генерираното изображение.



4.4 Помощен файл за изчертаване на изображения - OpenGL.h

За да улесним ползването на OpenGL в останалите примери от учебника ще създадем следния помощен файл: **CUDA_OpenGL.h**. Кодът ще позволява да създаваме нов прозорец за визуализация на 2D фигури със зададени параметри: височина и ширина, както и ще има функция позволяваща генерирането на сиво полутоново изображение от правоъгълна матрица със стойности на пиксели, които да визуализираме. Избираме сивото полутоново изображение вместо цветното т.к. по-натам в учебника ще демонстрираме действието на различни алгоритми за обработка на изображения, които най-елементарно се де-

монстрират върху сиви полутонови изображения. Целта на основното упражнението е да измерим времето нужно за генериране на псевдо случайно 2D изображене с различни размери върху централния процесор, с времето нужно за неговата генерация с CUDA. Няма да се спирам върху подробности за работата с OpenGL, тъй като на това има посветени редица книги и интернет туториали [6]. Следва да имате предвид, че по подобен начин бихте могли да визуализирате и графики на тримерни функции, които често се използват в инженерните изчисления. Основната функция в този код е **GL_Init(...)**, тя инициализира OpenGL и задава параметрите на прозореца. В нея се викат последователно две важни функции: **glutDisplayFunc(GL_Render)** и **glutIdleFunc(GL_Idle)**. Първата указва, коя функция да се използва за пречертаване на прозореца, а втората указва коя функция да се вика от прозореца, когато няма потребителска активност за допълнителни изчисления. Всичко което ще правим във функцията **GL_Render()** е да зареждаме нови стойности в изходния буфер с данни (пиксели) за визуализация. Функцията **GL_Idle(void)** ще бъде изпълнявана винаги, когато основното приложение няма неизпълнени задачи. Именно в кода на тази функция ние ще извикваме функция от основната програма, с която отново и отново ще генерираме стойности на пикселите, които ще визуализираме. Този пример притежава един недостатък, свързан с бързодействието на програмата. В него ние копираме масивите с данни за визуализация от паметта на видеокртата в RAM на компютъра, като след това OpenGL отново ги връща в паметта на видео картата. В случаите, когато визуализацията не е основният изчислителен проблем в нашата програма, данните които генерираме за изобразяване може директно да бъдат предадени за визуализация от OpenGL. Подобен пример ще бъде даден по-късно.

Listing 4.8: Изчертаване на 2D изображения с OpenGL.h.

```
#include <stdio.h>
#include "GL/freeglut.h"
#include "GL/gl.h"

unsigned int* pix;
int win_x;
int win_y;
static void GL_Render();
static void GL_Idle( void );
void GL_Init(void (*draw_surface)(),int X,int Y,
    unsigned int* pixels);
void (*draw)();

static void GL_Render() {
    glClearColor( 0.0, 0.0, 0.0, 1.0 );
    glClear( GL_COLOR_BUFFER_BIT );
    glDrawPixels( win_x, win_y, GL_RGBA,
        GL_UNSIGNED_BYTE, pix );
    glutSwapBuffers();
}

void GL_Init (void (*draw_surface)(),int X,int Y,
    unsigned int* pixels){
    pix = pixels;
    win_x = X;
    win_y = Y;
    draw = draw_surface;
    int c=1;
    char* dummy = "";
    glutInit( &c, &dummy );
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA )
        ;
    glutInitWindowSize(win_x,win_y);
    glutInitWindowPosition(100,100);
    glutCreateWindow("OpenGL - First window");
    glutDisplayFunc(GL_Render);
    glutIdleFunc( GL_Idle );
    glutMainLoop();
}
```

```
static void GL_Idle( void ) {
    clock_t h_start, h_stop;
    h_start = clock();
    draw();
    glutPostRedisplay();
    h_stop = clock();
    h_stop = h_stop-h-start;
    printf("CPU time: %f ms\n",(float)h_stop/1000)
        ;
}
```

По-долу е даден текста на основната програма, която инициализира OpenGL и в която е описана функцията за генериране на 2D изображение.

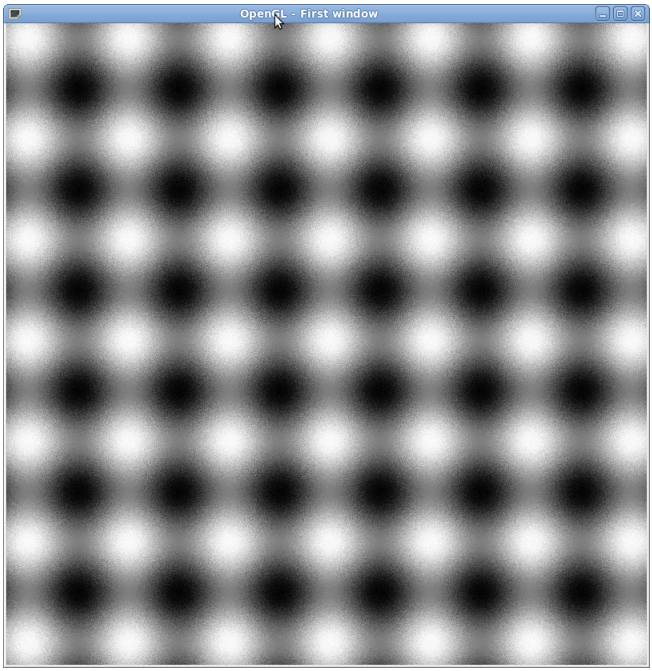
Listing 4.9: Използване на CUDA_OpenGL.h.

```
#include <stdio.h>
#include "CUDA_OpenGL.h"

#define WIN_X 800
#define WIN_Y 800

unsigned int* pixels;
void Draw_Surface_CPU();

int main(int argc, char** argv) {
  pixels = new unsigned int[WIN_X * WIN_Y];
  Draw_Surface_CPU();
  GL_Init((void (*)())Draw_Surface_CPU,WIN_X,
      WIN_Y,pixels);
  delete [] pixels;
    return 0;
}

void Draw_Surface_CPU() {
  unsigned char clr = 0;
  double X, Y = 0;
  for(int x=0;x<WIN_X;x++)
    for(int y=0;y<WIN_X;y++)
      {
```

```
            X = (double)x;
            X = 64+64*sin((rand()%10+X)/20);
            Y = (double)y;
            Y = 64+64*sin((rand()%10+Y)/20);
            clr = (unsigned char)(X+Y);
            if(clr>250)
               clr = clr -rand()%10;
            if(clr<10)
               clr = clr + rand()%10;
      pixels[x+y*WIN_X] = ((0x00000000+clr)<<16)+((0
          x00000000+clr)<<8)+clr;
      }
   }
```

Този код би следвало да създаде непрекъснато обновяващ се прозорец подобен на дадения по-долу. В отделно стартирана конзола ще се изписва времето в милисекунди необходимо на вашата система, за да преизчисли и преизчертае новите данни върху екрана. Ако вашето приложение работи вие вече имате уменията, с които да рисувате и изобразявате разнообразни 2D графични обекти от двумерни байтови матрици. За изчислението на стойностите ползваме фурнция изпълнявана върху централния процесор. В сладващия пример ще покажем как да правим това чрез GPU процесора, което ще деминстрира преимуществото му при решаване на подобен тип изчислителни "линейни" задачи. Съществуват алгоритми, чието изчисление върху GPU е значително по-сложно и трудоемко. Това е така защото графичните процесори са разработени предимно за обработка на изображения. Те биха обработили бързо всякакъв тип данни ако можем да оприличим изчислителния проблем, на специфична обработка на изображение, където отделните данни са пиксели съставящи това изображение (1D, 2D или 3D).

Фигура 4.4: Екранен изглед на първата ви OpenGL програма.



4.5 Да направим приложението по-бързо с CUDA

Сега ще модифицираме изходния код от по-горния пример, така че да има функция ползваща хардуерната платформа на CUDA. За целта ще дефинираме функция заделяща необходимите ресурси в CUDA и викаща съответното ядро, с което да обособим изчислителната функционалност. След изчислението ще върнем обратно данните в паметта на компютъра, от там ще ги подадем за изчертамане в OpenGL. Тъй като графичните мелтипроцесори не поддържат генерирането на случайни числа ние ще направим следващия пример в две стъпки. Първо ще реализираме

примера без възможност за генериране на случайни числа, като го реализираме върху CPU. За да определим времето нужно за това изчисление ще ползваме C функцията **clock()**. След това ще реализираме примера върху GPU и ще определим времето необходимо за изчислението с помощния файл, който създадохме **"CUDA_Timer.h"**. За да отчетем ефекта от мащаба ще изследваме това ускоряване при по-големи и по-малки по размер двумерни матрици. За наше улеснение в началото на кода ще добавим дефиниции за размера на изчислителния блок и респективно клъстер. По-късно ще изследвате влиянието на промените на тези стойности върху изчислителната скорост на приложението.

```
#define BLOCKX (16)
#define BLOCKY (16)
#define GRIDX (WIN_X/BLOCKX)
#define GRIDY (WIN_Y/BLOCKY)
```

Тъй като ще работим с CUDA ще ни е необходим и нов указател към заделените във видео картата ресурси памет, ползвани за съхраняване матрицата на изображението **unsigned int* dev_pixels**. По-долу е даден пълния текст на програмата. За да опростим процеса на изчертаване ще обособим функцията викаща CUDA ядрото **void Draw_Surface_GPU()** и самото ядро **__global__ void dev_draw_surface(unsigned int* pix,int size)**. Както ще забележите не правим съществена промяна в изходния код, но е премахната функцията за генериране на случайни числа rand() ползвана в предния пример. CUDA не предоставя готова функциуналност за генериране на псевдо случайни числа. За да определите времето нужно за изпълнение на функциите следва преди компилацията да закоментирате едната от тях и да изпълните приложението върху CPU, а после върху GPU:

```
GL_Init((void (*)())Draw_Surface_CPU, WIN_X,
    WIN_Y, pixels)
GL_Init((void (*)())Draw_Surface_GPU, WIN_X,
    WIN_Y, pixels)
```

Listing 4.10: Използване на CUDA при генериране на 2D изображение и визуализация с OpenGL.h.

```
#include <stdio.h>
#include "CUDA_OpenGL.h"
#include "CUDA_Timer1.cu"

#define WIN_X 1000
#define WIN_Y 1000
#define BLOCKX (16)
#define BLOCKY (16)
#define GRIDX (WIN_X/BLOCKX)
#define GRIDY (WIN_Y/BLOCKY)

unsigned int* pixels;
unsigned int* dev_pixels;
int s;
void Draw_Surface_CPU();
void Draw_Surface_GPU();


__global__ void dev_draw_surface(unsigned int*
    pix,int size){
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int pos = x + y*blockDim.x*gridDim.x;
  unsigned char clr = 0;
  float X, Y = 0;
  if(pos < size) {
    X = (float)x;
    X = 64+64*sin(X/20);
    Y = (float)y;
    Y = 64+64*sin(Y/20);
    clr = (unsigned char)(X+Y);
  pix[x+y*WIN_X] = ((0x00000000+clr)<<16)+((0
      x00000000+clr)<<8)+clr;
```

```
  }
}

int main(int argc, char** argv) {
  s = sizeof(unsigned int) * (WIN_X * WIN_Y);
  pixels = new unsigned int[WIN_X * WIN_Y];
  cudaMalloc((void**)&dev_pixels,s);
  Draw_Surface_CPU();
  Draw_Surface_GPU();
//  GL_Init((void (*)())Draw_Surface_CPU, WIN_X,
   WIN_Y, pixels);
  GL_Init((void (*)())Draw_Surface_GPU, WIN_X,
     WIN_Y, pixels);
  delete[] pixels;
  cudaFree(dev_pixels);
    return 0;
}

void Draw_Surface_CPU() {
  clock_t h_start, h_stop;
  h_start = clock();
  unsigned char clr = 0;
  double X, Y = 0;
  for(int x=0;x<WIN_X;x++)
    for(int y=0;y<WIN_X;y++)
      {
        X = (double)x;
        X = 64+64*sin(X/20);
        Y = (double)y;
        Y = 64+64*sin(Y/20);
         clr = (unsigned char)(X+Y);
  pixels[x+y*WIN_X] = ((0x00000000+clr)<<16)+((0
     x00000000+clr)<<8)+clr;
      }
  h_stop = clock();
  printf("CPU time: %f ms\n",(float)(h_stop-
     h_start)/100);
}

void Draw_Surface_GPU() {
  cudaEvent_t start, stop;
  CUDA_Timer_Start(&start, &stop);
```

```
    dim3 dimGrid(GRIDX,GRIDY);
    dim3 dimBlock(BLOCKX,BLOCKY);
    dev_draw_surface <<<dimGrid,dimBlock>>>(
        dev_pixels, s);
    cudaMemcpy(pixels, dev_pixels, s,
        cudaMemcpyDeviceToHost);
    float elapsedTime = CUDA_Timer_Stop(&start, &
        stop);
    printf("GPU Time: %f ms\n", elapsedTime);
}
```

Този пример имаше за цел да покаже преимуществото от ползването на CUDA при реализацията на един стандартен еднотипен изчислителен проблем. Не обръщайте особено внимание на функциите ползвани за изчертаване на екрана, те не са от съществено значение, основната им цел е да изобразят някакви фигури. Основното което следва да запомните от този пример е възможността да ползваме CUDA при решението на стандартни обемисти изчислителни проблеми. В моят случай ускорението което постигнах на CPU Intel Core 2 2.4GHz и GeForce 250 GTX е дадено в долната таблица. Отделете време и направете подобно сравнение за вашата система, като изчислите ускорението, което получавате при ползване на CUDA. Имайте предвид че при определянето на времето нужно за изчисление и времето нужно за копиране на данните от паметта на видео картата в оперативната памет на компютъра също може да бъде включено. В конкретния случай времето за копиране на данните е 10 пъти повече от необходимото време за самото изчисление върху GPU. При малки ядра времето за изчисление варира силно тъй като същественият компонент в него е времето нужно за прехвърляне на данните от паметта на видео картата в паметта на компютъра и обратно, което зависи от натоварването на системата и скоростта на системната шина. Важно е че общото ускорение варира в границите 200-3000 пъти, което е достатъчно

Таблица 4.2: Сравнение на производителността на приложението за генериране на 2D изображение върху CPU и CUDA (CPU-Intel Dual Core 2.4GHz, GPU-GTX250).

Размер на матрицата	**CPU изчислителна процедура**	**GPU само изчислителната процедура**	**GPU с копиране на данните в RAM**
100x100	0.01ms	0.018 ms	0.3 – 0.5 ms
300x300	110 ms	0.060 ms	0.3 – 0.5 ms
500x500	350 ms	0.170 ms	1.1 - 1.4 ms
800x800	1000 ms	0.308 ms	2.4 ms
1000x1000	1500 ms	0.470 ms	3,7 ms

условие за ползването на CUDA при решаването на подобен тип задачи! Ядрото е доста елементарно реализирано, тук идеята е да покажем до колко това ускорение е възможно. Имайте предвид, че не са извършени проверки за точно формиране на размера на изчислителните блокове в клъстера, което ще води до грешки в изходното изображение. Това може да се коригира, ако промените размера на клъстера, така че да размерността на картината да се дели на точен брой пъти на размера на блока.

Ускорена работа с RAM

Съществува техническа възможност сравнително бързо да ускорим работата на нашите CUDA програми. Както забелязахте времето нужно за това една програма написана за CUDA да приключи би могло условно да се раздели на 3 отделни интервала: време необходимо за заделяне на нужните масиви в RAM на компютъра и копиране на данните върху видеокартата, време за изчисление на CUDA kernel, и време за обратно копиране на резултатите от изчислението в RAM на компютъра. След като видяхме как CUDA може да ускори работата на програмата ни бихме могли да

експериментариме с ползването на т.н. заключена странична памет в RAM на компютъра. Този тип памет съществено ще ускори работата на CUDA програмите. Компютърната памет най-общо казано се дели на две области. Първата седи по-долу във физическите адреси, това е исторически обособено. Даден тип хардуер може да адресира директно тези адреси от паметта, или да прави това чрез контролера за директен достъп до паметта т.н. DMAC - Direct Memory Access Controller (за по-кратно DMA). Драйверите на звуковите карти, мрежовите интерфейси и видео картите ползват DMA, за да освободят централния процесор от необходимостта на управлява процеса на четене и запис на данни в тях. Това съществено ускорява работата на компютъра, особено ако върху него работи многозадачна операционна система. Освен това определени адреси от RAM на компютъра е възможно да бъдат директно адресирани от специфичен хардуер. Така хардуерните карти могат да "пишат" и "четат" данни независимо от централния процесор. Този процес обикновенно използва DMA контролерите, тъй като така се осигурява по-добра междуплатформена съвместимост на хардуера с различни операционни системи. Работата на драйвера на дадена хардуерна карта е да заделi един или няколко буфера първична памет, като преди това установи самият хардуер в режим на самоинициализация. Така когато има готов пакет данни, които трябва да бъдат прехвърлени от RAM на системата в хардуерната карта и обратно не се прекъсва работата на централния процесор при копирането на всеки байт данни. При запълване на първичния буфер се предизвиква програмно прекъсване на драйверната програма, което може да се използва от основната програма, за да и укаже, че в първичния буфер данни ими записани валидни стойности. Въз основа на това прекъсване основната програма може вземе решение да прочете тези данни и да ги копира во предварително заделена RAM. Аналогичен е процесът на запис

на данни от основната програма в първичния буфер. След като първичния буфер съдържа необходимият масив данни, се предизвиква активиране на DMA контролера, който от своя страна копира независимо от централния процесор тези данни в хардуерната карта. След като данните бъдат прехвърлени от RAM в хардуерната карта отново се предизвиква прекъсване в основната програма. Чрез него се съобщава, че първичният буфер е празен. За да се осигури по-равномерен режим на работа драйверът и основната програма може да ползват няколко първични и вторични буфера. По този начин при достигане края на даден първичен буфер DMA контролера се пренасочва да обработи следващия запълнен буфер с данни. Това е така защото е възможно в този момент централния процесор да бъде зает с други изчисления и това да предизвика закъснение при записа или четенето на данни в първичния буфер.

Във всъщност CUDA драйвера винаги ползва контролера за директен достъп до паметта, но когато заделяме RAM нужна за съхраняване на данните чрез функциите **new()**, **delete()** или **malloc()**, **free()** ние във всъщност заделяме блокове от паметта на компютъра намиращи се в горните адреси на RAM, която по подразбиране може да бъде временно записана на твърдия диск. Когато нашата програма копира данните от така декларираните масиви тя първо ги извилича на блокове и ги записва в първичните буфери ползвани от драйвера и DMA. След това в режим на самоинициализация контролера прехвърля данните във видео картата през PCI Express шината. Обратният процес протича по същия начин отново чрез прехвърляне на данните чрез контролера за пряк достъп до паметта на отделни блокове с поетапното им копиране в горните адреси на RAM. Този процес бави работата на CUDA програмата, като във всъщност вашата програма върши двойна работа.

Ползване на оператори new() и delete() за динамично управление на паметта, заделяне на 1000 елементен масив с целочислени 4 битови елементи и последващото му освобождаване:

```
int* i;
i = new int[1000];
...
delete i;
```

Ползване на оператори malloc() и free() за динамично управление на паметта, заделяне на 1000 елементен масив с целочислени 4 битови елементи и последващото му освобождаване. Ползването на malloc() изисква да укажем размерът на заделяния блок памет в байтове, поради тази причина използваме оператора sizeof(int), с който определяме размера в байтове на типа елемент от който ще бъде съставен масива. В случая това е целочислена 4 битова стойност. Размерността на типа променлива умножаваме по броят елементи в масива, за да определим реалният обем необходими ни байтове за формиране на искания масив.

```
int* i;
i = (int*) malloc (1000 * sizeof(int));
...
free (i);
```

Съществува метод чрез който вашата CUDA програма да заделя т.н. заключена странична памет директно достъпна от драйвера на видеокартата. При ползването на стандартните функции за динамично заделена памет при отсъствие на достатъчно RAM, операционната система автоматично ще кешира копия на неизползваните в даден момент страници памет върху твърдия диск на компютъра. Това е така наречения SWAPPING (буквално значи разместване на дадена страница от паметта и записването и върху твърдия диск, като освободената област се предоста-

вя на други процеси и програми). Тогава ако вашата CUDA програма се опита да чете или пише в тази област от RAM, операционната система първо ще трябва да зареди липсващата страница в RAM от твърдия диск и след това да извърши копирането на данните. Чрез функциите **mlock()** и **munlock()** ние бихме могли да "заключим" нужната ни динамично заделена памет във всякя една C++ програма. Така тя няма да се кешира върху твърдия диск. Това крие някои опасности, т.к. върху операционната система е възможно да работят и други приложения, които в този момент ще изпитват недостиг от оперативна памет и те самите ще трябва по-активно да ползват “суапването - swap“ на данни. Ето защо ползването на тези методи следва да бъде внимателно и обосновано.

При използването на функциите mlock() и munlock(), като параметър се предава указател на началото на областта данни, която ще бъде заключена, тоест ще остава резидентна в RAM на компютъра и дължината на сегмента в байтове.

```
int* i;
i = new int[1000];

mlock   (i[0], 1000 * sizeof(int));
munlock(i[0], 1000 * sizeof(int));

delete i;
```

Въпреки че този механизъм ще ускори работата на програмите ви, той не решава проблема с ползването на контролера за пряк достъп до паметта при копиране на данни от/и върху видео картата. За решението му можем да ползваме специализираните функции **cudaHostAlloc()** и **cudaFreeHost()** и да заделяме необходимата ни памет в RAM на компютъра, като имаме предвид че така CUDA драйвера ще може да ползва директно контролера за ди-

ректен достъп до паметта при трансфера на данните. Това следва да се прави особено внимателно при системи с малък обем памет (<1GB). Един типично инсталиран и работещ Линукс заема около 400-800MB RAM. Това значи че ползването на тези методи за заключване на паметта бързо би довело до снижена работоспособност на системата при липса на нужната и RAM за работа на останалите приложения. Помнете че тази памет следва да бъде освободена веднага след като приключите работа с нея. Тоест вашето основно приложение може да ползва други не заключени страници от паметта за пост обработка на рузелтатите, като ползвате заключените масиви в паметта само и единствено по време на изпълнението на CUDA kernel. Примерът може да откриете в папка **Ch4.5.1** на приложения диск. Съществеото различие между предходната реализация се изразява само в методите ползвани за заделяне на паметта във функцията **main()**, дадени в листигна по-долу.

```
int main(int argc, char** argv){
  s = sizeof(int) * (WIN_X * WIN_Y);
//  pixels = new int[WIN_X * WIN_Y];
  cudaHostAlloc((void**)&pixels, WIN_X*WIN_Y*
      sizeof(*pixels), cudaHostAllocDefault);
  cudaMalloc((void**)&dev_pixels,s);
  GL_Init((void (*)())Draw_Surface_GPU, WIN_X,
      WIN_Y, pixels);
//  delete[] pixels;
  cudaFree(pixels);
  cudaFree(dev_pixels);
    return 0;
}
```

Кооперативна съвместимост на OpenGL с CUDA

В предните примери видяхме как може да използваме CUDA, за да увеличим скоростта на изчисление при линейна обработка на масиви с данни. Крайната цел на

подобен вид приложения е да се генерира и визуализира 2D или 3D изображение. В предния пример ние направихме няколко оптимизации, които правят нашата програма по-бърза. Видяхме как е възможно да заключим страничната памет на системата, при което да ускорим работата на CUDA драйвера свързана с копиране на данните от видеокартата в паметта на компютъра и обратно. Подобен трансфер на данните обратно в паметта на компютъра е обоснован само ако тези данни подлежат на допълнителна обработка, транспортиране по мрежа или съхраняване във файл. Тъй като данните, които получаваме подлежат само на визуализация с OpenGL те отново трябва да бъдат върнати във видеокартата за изобразяване. CUDA и OpenGL ни предоставят инструмент за обезпечаване на взаимната работа. Чрез обезпечаване на графичната съвместимост CUDA може да резервира ресурси на видео картата, които са директно достъпни от OpenGL за визуализация и същевременно са достъпни за обработка от CUDA kernel функциите. При това всичко което програмистът следва да направи е да обособи такъв тип OpenGL ресурси, които могат да бъдат достъпни и от CUDA драйвера. При тази операция ние спестяваме времето нежно за обратно връщане на данните преди визуализацията и последващото им копиране отново във видеокартата за визуализация. Примерът използва двойна буферизация на екранния изглед, като следва да се има предвид, че когато OpenGL и CUDA имат монополен достъп до данните. Тоест когато генерираме данните с kernel функция ние не можем да ги визуализираме, а когато ги изобразяваме ние не можем да ги обработваме. За щастие CUDA предвижда специални инструменти, които да правят тази задача по-лесна. Пълният код на примера може да намерите в папката **Ch4.5.2**. Основните разлики между досегашните програми са във файла CUDA_OpenGL.h.

4.6 Генериране на шум

За да генерираме шум в нашето изображение ползвайки CUDA ще заменим досегашната функция с дадената по-долу. Този метод за генериране на шум не е оптимален, но той показва как можем да решим подобен проблем свързан с нуждата от генериране на големи обеми псевдо случаен шум - случайни числа. За да генерираме псевдо случаен шум можем да ползваме преместващ регистър, на чийто вход се подават избрани битове обединени от операцията изключващо или (XOR). Ние ще подходим още по-елементарно и ще създадем два отделни масива в паметта на видео картата, единият е досегашния **unsigned int* dev_pixels**, който съхранява данните за изходното изображение, и **unsigned int* dev_rnd** за шума. Преди да изобразим изходното изображение ние ще сумираме чистия образ със шумовия образ, като управлявайки коефициента на зашумяване ще можем да управляваме и степента на замързяване на изображението. Обърнете внимание на начина, по който генерираме шума. Прочитаме поредни стойности от паметта на видео картата и извършваме операция изключващо или с други поредно записани стойности от паметта и. Резултатът е пискел имащ случайна за нас стойност. Макар методът да не е оптимален може да бъде полезен. Шум в цифровите изображения се ползва, когато се визуализират 3D обекти с текстури (изображения с които обектите се "обличат", като с кожа). Добавянето на шум подобрява реалистичното възприемане от човешкото око на не дотам добре моделираните обекти.

```
__global__ void dev_draw_surface(unsigned int*
    pix,unsigned int* rn, int size){
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int pos = x + y;
  unsigned char rnd;
```

```
    unsigned char clr = 0;
    float X, Y = 0;

    if(pos < size){
      rnd = (unsigned char)(rn[x/2+y*WIN_X] xor rn[
          x+(y/2)*WIN_X]);
      X = (float)x;
      X = 64+64*sin(X/20);
      Y = (float)y;
      Y = 64+64*sin(Y/20);
      rnd = (unsigned char)(X+Y)+rnd;
    rn[x+y*WIN_X] = ((0x00000000+rnd)<<16)+((0
        x00000000+rnd)<<8)+rnd;

      int clr1 = (unsigned char)(X+Y) + rnd/10;
      if(clr1>255)
        clr = clr1 - rnd/10;
      else
        clr = clr1;
    pix[x+y*WIN_X] = ((0x00000000+clr/2)<<16)+((0
        x00000000+clr)<<8)+clr;
    }
  }
```

4.7 Изчисление и визуализация на хистограми с CPU

Този пример обединява няколко нови неща: как да изобразявате линии в 2D изображение с OpenGL, как да изчислявате хистограми върху CPU и GPU и някои особености относно работата със споделената памет и атомарните операции в CUDA. Тези базови възможности ще ви бъдат полезни при създаването на по-сложни приложения за обработка на изображения и видео информация.

Сама по себе си хистограмата представлява съвкупност

от статистически данни касаещи честотата (броят) на пиксели имащи дадени стойности в цялото изображения. В английските текстове може да го срещнете още и като PDF (Probability Distribution Function). Тъй като ние обработваме сиви полутонови изображения, без цвят, и предварително знаем, че те са 8 битови нашата хистограма ще има 256 сегмента. Всеки един сегмент ще показва колко пиксела със съответната сива стойност са били намерени в изображението, което обработваме. Броят стълбове съответства на пълния набор възможни стойности, които всеки един пиксел – точка от нашето изходно изображение би могъл да заема. Нашата цел се състои в това да проверим стойността на всички точки на изображението и да преброим броят на повторение на всеки един пиксел с определена яркост в него. Съществуват и по-сложни модели на хистограми: 2D, 3D, 4D и т.н. Като цяло ще се убедите, че изчислението на хистограмите не е сложно и е лесно постижимо за алгоритми работещи върху централния процесор на компютъра, но когато това на пръв поглед елементарно действие следва да се извърши от GPU нещата стават доста сложни. Първо нека модифицираме кода, който използвахме за изобразяване на 2D фигури в OpenGL. Данните за хистограмата ще съхраняваме в отделен масив unsigned **int [256]**. Ще създадем още няколко допълнителни функции: **GL_Init(...)**, с допълнителен аргумент – указател сочещ към изходящия масив съхраняващ хистограмата, и функцията **GL_Render(...)**, в която се намира същинската част от кода изобразяваща хистограмата в долния ляв ъгъл на изходното изображение. Ще добавим още една функция, имаща за цел да определи максималната стойност в масива на хистограмата - функцията **find_max(...)**. Имайте предвид, че в OpenGL се използват относителни координати на екрана. Центъра на екрана има координати X=0, Y=0. Съответно най-добната лява точка на екрана има стойности -1 -1, най-горната дясна +1 +1, най-долната

дясна +1 -1, а най-горната дясна +1 +1.

Приложението ползва следните три файла:

- **GUDA_GPU_histo_2D.cu**
- **CUDA_OpenGL_2D_histo.h**
- **CUDA_Timer1.cu**

```
void GL_Init(void (*draw_surface)(),int X,int Y,
    unsigned int* pixels,unsigned int* hist){
  ...
  histo = hist;
  ...
}

static int find_max(unsigned int* arr,int lenght)
    {
  unsigned int max = 0;
   for(int i=0; i<lenght; i++){
    if(arr[i] > max)
      max = arr[i];
  }
  return max;
}

static void GL_Render() {
      glClearColor( 255.0, 255.0, 255.0, 1.0);
      glClear( GL_COLOR_BUFFER_BIT );
      glDrawPixels( win_x, win_y, GL_RGBA,
          GL_UNSIGNED_BYTE, pix);
  glColor3f(0, 200, 0);//RGB
  glBegin(GL_LINES);
  float x = -1;
  float y = -1;
  unsigned int max = 0
  max = find_max(histo,256);

  for(int i=0; i< 256; i++){
    x = x + 0.001;
    y = -1 + (float)(histo[i])/max;
```

```
        glVertex2d(x, -1);
        glVertex2d(x, y);
    }
        glEnd();
        glutSwapBuffers();
        glFlush();
}
```

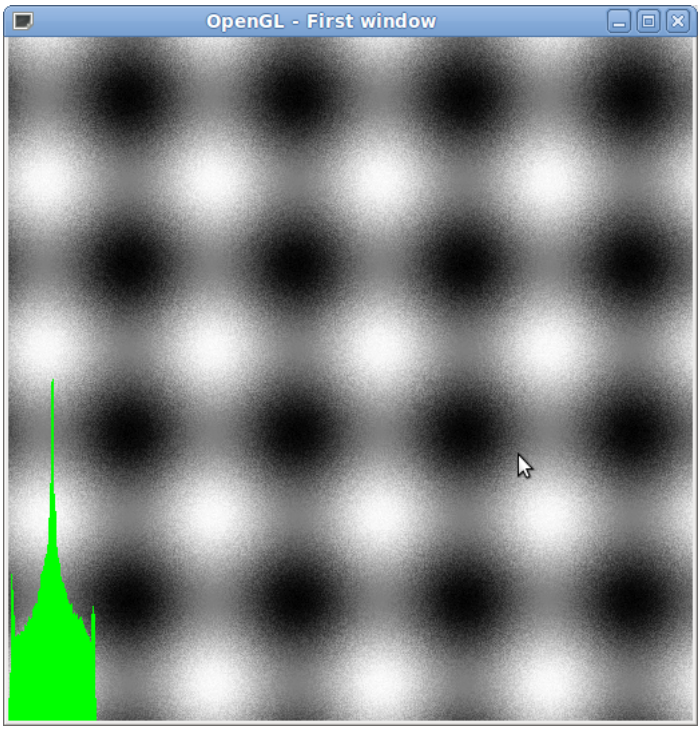
След като извършихме тези модификации в изходния код позволяващ ни да изобразяваме 2D фигури и 1D хистограми на екрана с OpenGL ще трябва да нанесем корекции и в изходния код на файла **GUDA_GPU_histo_2D.cu** в който преди това описахме функции за генерирарне на фигури. Тъй като ще обработваме 8 битови сиви полутонови изображения ще ни е необходим един 256 байтов масив за съхранение на хистограмите, който ще дефинираме, като глобална променлива: **unsigned int hi[256]**. След това ще добавим и функция, която да изчислява самата хистограма: **CPU_Histogram()**, която ще викаме в края на функцията **Draw_Surface_GPU()**.

```
void CPU_Histogram() {
  for(int i=0; i<256; i++)
    hi[i] = 0;

  unsigned char pix = 0;
  for(unsigned int i=0; i< WIN_X*WIN_Y; i++){
  pix = pixels[i] & 0x000000FF;
  hi[pix] = hi[pix] + 1;
  }
}
```

Тъй като функцията C**PU_Histogram()** ще се извиква при всяко генериране на стойности в 2D масива от пиксели, които формират изображението преди да изчисляваме неговата хистограма трябва да нулираме предходните стойности. След това ще обходим точка по точка масива от пиксели и ще проверяваме всяка една от тях каква стой-

ност има.

Фигура 4.5: Хистограмата на изображението се изобразява в долния ляв ъгъл на прозореца.

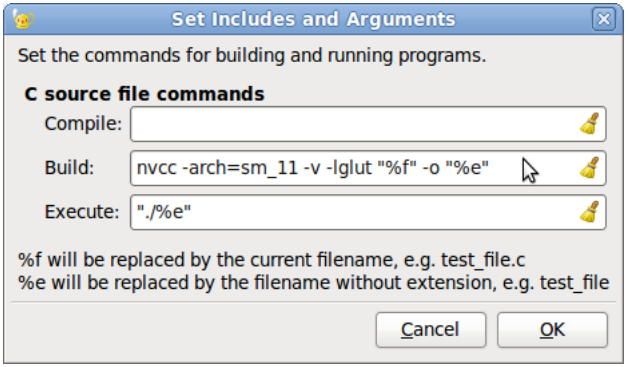


Трикът на алгоритъма се крие в това, че за да преброим елементите с точно определена стойност ние ще обходим пикселине на цялото изображение, като ще инкрементираме онези елементи на хистограмата чиито адреси съвпадат с конкретната стойност на всеки един пиксел. Ако пиксема има стойност 10 ние ще инкрементираме елемент 10, ако има стойност 25 ще инкрементираме елемент 25 и т.н. Това прави този алгоритъм бърз и именно поради това хистограмните методи се ползват често за реализация на алгоритми за обработка на изображения. Тези методи

стапат към т.н. еднопикселни вероятностно статитстически методи за обработка, освен тях има и многопикселни хистограмни методи. Както ще видим по-долу нещата при CUDA не стоят по същия начин и това налага прибягване до разнообразни хитрости свързани с ползването на споделената памет и атомарните операции. Ще наричаме това „висш пилотаж“. Ако компилираме кодът създаден до сега би следвало да получите подобен на дадения тук екранен изглед Фиг. [?]. Хистограмата на изображението ще е динамично меняща се зелена графика в долния ляв ъгъл на основния прозорец. Тук е моментът да експериментирате за вида на хистограмата в зависимост от типа фигура, която изчертавате, а също така и от нивата на шума, който генерираме.

Компилирането на готовият проект може да стане с командата **nvcc -arch=sm_11 -v -lglut "%f" -o "%e"**, изписана в командния ред, или командата еднократно въведена в geany менюто - Build->Set Inludes and Arguments.

Фигура 4.6: Настройки на geany за компилация на приложения с атомарни операции и OpenGL поддръжка.



Listing 4.11: Изходен код на файла "CUDA_GPU_histo_2D_CPU.cpp".

```
#include <stdio.h>
#include <cuda.h>
#include "CUDA_OpenGL_2D_histo.h"
#include "CUDA_Timer1.cu"

#define WIN_X 800
#define WIN_Y 800
#define BLOCKX (16)
#define BLOCKY (16)
#define GRIDX (WIN_X/BLOCKX)
#define GRIDY (WIN_Y/BLOCKY)

unsigned int* pixels;
unsigned int* dev_pixels;
unsigned int* dev_rnd;
unsigned int hi[256];

int s;
void Draw_Surface_CPU();
void Draw_Surface_GPU();
void CPU_Histogram();

__global__ void dev_draw_surface(unsigned int*
    pix,unsigned int* rn, int size){
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int pos = x + y;
  unsigned char rnd;

  unsigned char clr = 0;
  float X, Y = 0;

  if(pos < size){
    rnd = (unsigned char)(rn[x+y*WIN_X] xor rn[x
        +1+y*WIN_X]);
    X = (float)x;
    X = 64+64*sin(X/18);
    Y = (float)y;
    Y = 64+64*sin(Y/18);
    rnd = (unsigned char)(X+Y) + rnd;
    rn[x+y*WIN_X] = ((0x00000000 + rnd)<<16)+((0
        x00000000 + rnd)<<8)+rnd;
```

```
      int clr1 = (unsigned char)(X+Y) + rnd/10;
      if(clr1>255)
        clr = clr1 - rnd/10;
      else
        clr = clr1;
    //  clr = (unsigned char)(X+Y);
      pix[x+y*WIN_X] = ((0x00000000 + clr)<<16)+((0
          x00000000 + clr)<<8)+clr;
    }
  }

  int main(int argc, char** argv)
  {
    s = sizeof(unsigned int) * (WIN_X * WIN_Y);
    pixels = new unsigned int[WIN_X * WIN_Y];
    dev_rnd =  new unsigned int[WIN_X * WIN_Y];
    cudaMalloc((void**)&dev_pixels,s);
    cudaMalloc((void**)&dev_rnd,s);

    Draw_Surface_CPU();
    Draw_Surface_GPU();

  //  GL_Init((void (*)())Draw_Surface_CPU, WIN_X,
      WIN_Y, pixels, &hi[0]);
    GL_Init((void (*)())Draw_Surface_GPU, WIN_X,
        WIN_Y, pixels, &hi[0]);

    delete[] pixels;
    delete[] dev_rnd;
    cudaFree(dev_pixels);
    cudaFree(dev_rnd);
      return 0;
  }

  void Draw_Surface_CPU() {
    clock_t h_start, h_stop;
    h_start = clock();
    unsigned char clr = 0;
    double X, Y = 0;
    for(int x=0;x<WIN_X;x++)
      for(int y=0;y<WIN_X;y++)
```

```
        {
          X = (double)x;
          X = 64+64*sin((rand()%10+X)/20);
          //X = 64+64*sin(X/20);
          Y = (double)y;
          Y = 64+64*sin((rand()%10+Y)/20);
          //Y = 64+64*sin(Y/20);
          clr = (unsigned char)(X+Y);
          if(clr>250)
            clr = clr - rand()%10;
          if(clr<10)
            clr = clr + rand()%10;
          pixels[x+y*WIN_X] = ((0x00000000 + clr)
              <<16)+((0x00000000 + clr)<<8)+clr;
        }
    h_stop = clock();
    printf("CPU time: %f ms\n",(float)(h_stop-
        h_start)/1000);
    CPU_Histogram();
}

void Draw_Surface_GPU() {
    cudaEvent_t start, stop;
    CUDA_Timer_Start(&start, &stop);
    dim3 dimGrid(GRIDX,GRIDY);
    dim3 dimBlock(BLOCKX,BLOCKY);
    dev_draw_surface <<<dimGrid,dimBlock>>>(
        dev_pixels, dev_rnd, s);
    cudaMemcpy(pixels, dev_pixels, s,
        cudaMemcpyDeviceToHost);
    float elapsedTime = CUDA_Timer_Stop(&start, &
        stop);
    printf("GPU Time: %f\n", elapsedTime);
    CPU_Histogram();
}

void CPU_Histogram()
{
    for(int i=0; i<256; i++)
      hi[i] = 0;

    unsigned char pix = 0;
```

```
    for(unsigned int i=0; i< WIN_X*WIN_Y; i++){
      pix = pixels[i] & 0x000000FF;
      hi[pix] = hi[pix] + 1;
    }
  }
```

Listing 4.12: Изходен код на файла "CUDA_OpenGL_2D_histo.h".

```
#include <stdio.h>
#include "GL/freeglut.h"
#include "GL/gl.h"

unsigned int* pix;
unsigned int* histo;
int win_x;
int win_y;

static void GL_Render();
static void GL_Idle( void );
void GL_Init(void (*draw_surface)(), int X, int Y
    , unsigned int* pixels, unsigned int* hist);
static void (*draw)();//
                  ,

static int find_max(unsigned int* arr, int lenght
    );

static void GL_Render()
{
     glClearColor( 255.0, 255.0, 255.0, 1.0 );
     glClear( GL_COLOR_BUFFER_BIT );
     glDrawPixels( win_x, win_y, GL_RGBA,
         GL_UNSIGNED_BYTE, pix );

   glColor3f(0, 200, 0);//RGB
   glBegin(GL_LINES);
   float x = -1;
   float y = -1;
   unsigned int max = 0;
   max = find_max(histo,256);
```

```
    for(int i=0; i< 256; i++){
      x = x + 0.001;
      y = -1 + (float)(histo[i])/max;
      glVertex2d(x, -1);
      glVertex2d(x, y);
    }
    glEnd();
      glutSwapBuffers();
      glFlush();
}

void GL_Init(void (*draw_surface)(), int X, int Y
    , unsigned int* pixels, unsigned int* hist)
{
  pix = pixels;
  histo = hist;
  win_x = X;
  win_y = Y;
  draw = draw_surface;
  int c=1;
    char* dummy = "";
    glutInit( &c, &dummy );
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA
        );
    glutInitWindowSize(win_x,win_y);
    glutInitWindowPosition(100,100);
    glutCreateWindow("OpenGL - First window");
    glutDisplayFunc(GL_Render);
    glutIdleFunc( GL_Idle );
  glutMainLoop();
}

static void GL_Idle( void ) {
  draw();
  glutPostRedisplay();
 }

 static int find_max(unsigned int* arr, int
     lenght){
   unsigned int max = 0;
   for(int i=0; i<lenght; i++){
```

```
    if(arr[i] > max)
      max = arr[i];
  }
  return max;
 }
```

Listing 4.13: Изходен код на файла "CUDA_Timer1.cu".

```
#include <stdio.h>

void CUDA_Timer_Start(cudaEvent_t* t_start,
    cudaEvent_t* t_stop);
float CUDA_Timer_Stop(cudaEvent_t* t_start,
    cudaEvent_t* t_stop);

void CUDA_Timer_Start(cudaEvent_t* t_start,
    cudaEvent_t* t_stop) {
    cudaEventCreate(t_start);
    cudaEventCreate(t_stop);
    cudaEventRecord(*t_start, 0);
}

float CUDA_Timer_Stop(cudaEvent_t* t_start,
    cudaEvent_t* t_stop) {
    float elapsedTime = 0;
    cudaEventRecord(*t_stop, 0);
    cudaEventSynchronize(*t_stop);
    cudaEventElapsedTime(&elapsedTime, *t_start,
        *t_stop);
    cudaEventDestroy(*t_start);
    cudaEventDestroy(*t_stop);
    return elapsedTime;
 }
```

4.8 Изчисление на хистограми с CUDA

Изчисление с глобални атомарни операции

Първоналачно ще опитаме да имплементираме алгоритъма за изчисление на хистограми върху CUDA почти директно, както той е реализиран в основната програма. За да бъда коректен ще кажа, че в този пример представлява една различна имплементаия на алгоритъма за изчисление на 1D хистограма с 256 елемента върху централния процесор. По-натам в текста ще говорим и за така наречените 2D хистограми. В предните няколко примера видяхме как с CUDA бихме могли да изчисляваме по-бързо някои алгоритми. Дали това обаче ще доведе до успех ако директно опитаме да имплементираме алгоритъма за изчисление на глобалната хистограма на изображението върху CUDA? Преди да отговорите на този въпрос се замислете как във всъщност се изчислява хистограмата? Това става чрез инкрементиране на поредни адреси в даден масив със стойности. В нашият случай имайки карта GTX250 средната скорост за изчисление на 1 мегабайтов образ е близо 4 пъти по-голяма в сравнение с времето нуждо за изчисление на същата хистограма върху централния процесор. Това е така поради факта, че цялото изображение се разбива на решетка от блокове, всеки от който представлява решетка от процеси. На първи поглед това позволява паралелното изчисление на хистограмата за всеки един блок. Това на пръв поглед би трябвало да повиши неимоверно скоростта на работа на нашата програма. Но това в действителност не е вярно, тъй като всички блокове ще се съревновават да получат достъп едновременно до едни и същи адреси на глобалната памет на видео картата в които ще съхраняваме финалния образ на хистограмата. Това сериозно ще забави нашата паралелна програма. На помощ ни идват атомарните операции. С тях ние можем директно и бързо да инкрементираме адреси от глобалната памет и адреси от споделената памет на мултипроцесорите. Атомарните операции са непрекъсваеми инструкции. Те ни гарантират, че само един единствен процес ще получи достъп до дадена

част от паметта докато започналата операция не приключи. Тези операции обаче не са достъпни във всички архитектури на видео карти поддържащи CUDA. Атомарни операци над глобалната памет са дъстъпни при версия 1.1, а атомарни операции над споделената памет са допустими върху архитектура 1.2 и по-висока. Възможно е вашата видео карта да няма подобна функционалност, но въпреки всичко разгледайте следващите две реализации на алгоритъма. Това може да е добър повод да закупите по-нов модел видео карта от серията GeForce. Друга особеност на примера е че генерирането на изображението и изчислението на неговата хистограма ще бъдат обособени в две отделни kernels и викани последователно от основния код на програмата. След това данните от хистограмата ще бъдат копирани обратно в паметта на компютъра и подавани на OpenGL за визуализация. Естествено тук бихте запитали не е ли нецелесъобразно да копираме данните от видео паметта в оперативната памет на компютъра и след това отново чрез OpennGL да ги изобразяваме с помощща на видео картата? Естествено да, но в следващите примери ще видим как да избегнем този недостатък. За момента ще се фокусираме над имплементацията на два метода за изчисление на хистограми с CUDA и атомарни операции. Този пример е заимстван от готови примери в интернет, но показва нагледно ползата от атомарните операции. За улеснение на проследяването на кода ще обособим викането на ядрото в отделна функция **GPU_Histogram()**, като за проследяване на времето за изпълнение на функцията сме дефинирали таймери. А самото ядро е: **__global__ void dev_histo_1(unsigned int* pix, unsigned int* h, int size)**. Имайте предвид, че данни с обем дори 1000x1000 пиксела са далеч под възможностите на една съвременна GeForce карта. За да добием по-добра представа за времето за изпълнение по-късно ще модифицираме кода така, че той да се вика неколкократно, като ще усредним време-

то нужно за изпълнение на самата функция за изчисление на хистограмите. Този подход може да използвате когато определяте производителността на CUDA kernel във различни програми.

Приложението ползва следните три файла:

- **GUDA_GPU_histo_2D_GPU.cu**
- **CUDA_OpenGL_2D_histo.h**
- **CUDA_Timer1.cu**

Новите функции, които следва да добавите в кода от предния пример са: **GPU_Histogram()** и **__global__ void dev_histo_1(...)** използваща глобални атомарни операции за изчисление на хистограмата.

```
void GPU_Histogram() {
  for(int i=0; i<256; i++)
    hi[i] = 0;
  cudaEvent_t start, stop;
  CUDA_Timer_Start(&start, &stop);
  dim3 dimGrid(GRIDX, GRIDY);
  dim3 dimBlock(BLOCKX, BLOCKY);
  cudaMemcpy(dev_hist, hi, 256*sizeof(unsigned
      int), cudaMemcpyHostToDevice);
  dev_histo_1 <<<dimGrid,dimBlock>>>(dev_pixels,
      dev_hist, s);
  cudaMemcpy(hi, dev_hist, 256*sizeof(unsigned
      int), cudaMemcpyDeviceToHost);
  float elapsedTime = CUDA_Timer_Stop(&start, &
      stop);
  printf(                          , elapsedTime)
      ;
}
```

По-долната функция ще реализира същинското изчисление на хистограмата ползвайки глобални атомарни операции. Тези операции ви позволяват да пишете програми

ползващи аритметични операции, при които записването на резултата от изчислението на данни получени от глобалната или локалната памет става за един процесорен цикъл. Атомарните операции се поддържат от архитектура 1.1 нагоре, като архитектура 1.2 поддържа и атомарни операции с данни записани в споделената памет.

```
__global__ void dev_histo_1(unsigned int* pix,
    unsigned int* h, unsigned int* multiple_hist,
    int size){
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int pos = x + y * blockDim.x * gridDim.x;
unsigned char dat = 0;
if(pos < size){
    dat = pix[pos] & 0x000000FF;
    atomicAdd( &h[dat], 1);
  }
}
```

Кодът от примерите по-горе няма да бъде достатъчно бърз в сравнение с кода написан за изчисление върху CPU на компютъра. Това е така защото множество стартирали процеси изпълнявани върху няколко мултипроцесора ще опитват да четат и пишат данни в едни области на глобалната памет едновременно, което ще доведе до колизии и забавяне. Ето защо, за да ускорим този код бихме могли да ползваме две възможни решения. Първото решение е дадено в CUDA SDK пример, който може да намерите в папката която онсталирахте вашето SDK: NVIDIA_GPU_Computing_SDK/C/bin/linux/release/histogram. Другият вариант е да закупите видео карта с хардуерна поддръжка на архитектура 1.2 или по-висока, този вариант е препоръчителен, тъй като много от вашите бъдещи алгоритми ще имат потребност от подобен вид глобални и локални атомарни операции.

Изчисление с глобални и споделени атомарни операции

Бихме могли да получим още по-високо бързодействие ако използваме по-активно споделената памет на всеки мултипроцесор. Ще припомня че процесите работещи в един изчислителен блок могат да ползват споделената памет заедно. Това означава, че докато един мултипроцесор е натоварен и работи активно със споделената си кеш памет, другите биха могли да получат достъп до глобалната памет. Това ще ускори работата на нашето приложение до няколко стотици пъти [2]. За да ползваме версията с атомарни операции върху кеш паметта на мултипроцесорите трябва да разполагаме с видео карта с изчислителна архитектура версия 1.2 и нагоре. Накратко нашето приложение ще копира сегмент от данните касаещ всеки един изчислителен блок в кеш паметта на мултипроцесора. След това ползвайки локални атомарни операции ще се изчислява копие на локална хистограма за копираните изходни данни в споделената памет. След това междинната хистограма ще се добави в глобалната памет към другите стойности записани от другите изчислителни блокове. Този пример ползва още една важна команда **__syncthreads()**. Тази команда е важна защото без нея процесите изчислявани в един блок ще се изпълняват в произволен ред. Това ще доведе до поява на грешки, тъй като някои процеси няма да могат да довършат своята работа в момента на зареждане на друг изчислителен блок. Командата **__syncthreads()** ще накара CUDA хардуера да изчака до приключване изпълнението на всички процеси в един изчислителен блок и чак след това ще пристъпи към изчислението на процеси от друг изчислителен блок.

```
__global__ void histo_kernel(unsigned char *
    buffer,long size,unsigned int *histo){
     __shared__ unsigned int temp[256];
     temp[threadIdx.x] = 0;
```

```
        __syncthreads();
        int i = threadIdx.x+blockIdx.x*blockDim.x;
        int stride = blockDim.x * gridDim.x;
        while (i<size) {
            atomicAdd(&temp[buffer[i]],1);
            i += stride;
        }
        __syncthreads();
        atomicAdd(&(histo[threadIdx.x]),temp[
            threadIdx.x]);
    }
```

4.9 Синхронизация при операции със споделената памет

Впредните два примера видяхме как може да изчисляваме хистограми с помощтта на глобални и локални атомарни операции и изполване на споделената памет **__shared__**. Едновременно с това в последния пример използвахме функцията **__syncthreads()**. Тази функция работи, като бариера за всички процеси в блока. При нейната употреба нито един процес, който е достигнал бариерата не може да продължи докато всички процеси в блока не достигнат до бариерата. На пръви поглед това би забавило програмите ви, но в този пример ще видите как употребата на **__syncthreads()** повишава прецизността на вашите програми, ползващи споделена памет. В този пример ползваме CUDA, за да изчертаваме фигури подобни на тези от предните примери. Разликата се състои в това, че ползваме споделената памет, за да изчисляваме отделните части на изображението, а след това прехвърляме тези данни в глобалната памет на видео картата. При това основното изображение на разбито на отделни блокове с размер 16x16 пиксела. Всеки блок се изчислява отделно, след из-

числението на блока данните в обратен ден се копират в глобалната памет. Основният фокус на примера е не какво изчертаваме, а функцията **__syncthreads()**.

Listing 4.14: Използване на споделената памет и __syncthreads().

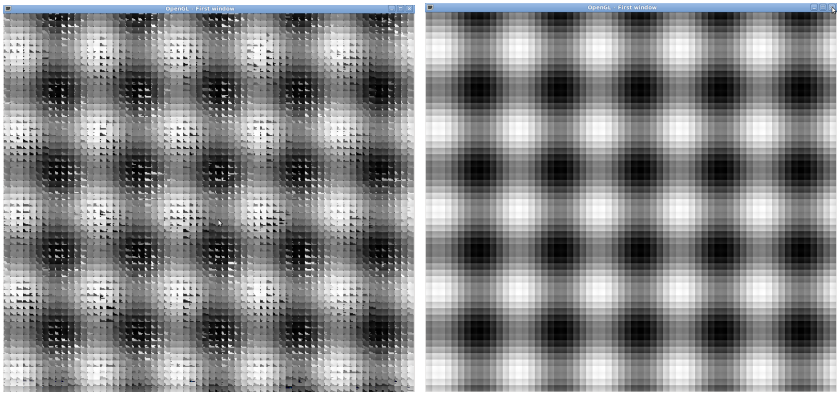
```
#include <stdio.h>
#include "CUDA_OpenGL.h"
#include "CUDA_Timer1.cu"

#define WIN_X 1024
#define WIN_Y 1024

#define BLOCKX (16)
#define BLOCKY (16)
#define GRIDX (WIN_X/BLOCKX)
#define GRIDY (WIN_Y/BLOCKY)

unsigned int* pixels;
unsigned int* dev_pixels;
int s;
void Draw_Surface_GPU();

//nvcc -lglut "%f" -o "%e"
__global__ void dev_draw_surface(unsigned int*
    pix,int size){
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int pos = x + y*blockDim.x*gridDim.x;
  unsigned char clr = 0;
  unsigned int  color = 0;
  const float period = 32;
  __shared__ int     shared[16][16];

  clr = (unsigned char)(127+127*(sin((y/period)))
    + (127 + 127*(sin(32+x/ period))));

  color = ((0x00000000 +clr )<<16)+((0x00000000 +
      clr)<<8)+clr;
  shared[15-threadIdx.x][15-threadIdx.y] = color;
//   __syncthreads();
```

```
    pix[pos] = shared[threadIdx.x][threadIdx.y];
}

int main(int argc, char** argv)
{
    s = sizeof(unsigned int) * (WIN_X * WIN_Y);
    pixels = new unsigned int[WIN_X * WIN_Y];
    cudaMalloc((void**)&dev_pixels, s);

    GL_Init((void (*)())Draw_Surface_GPU, WIN_X,
        WIN_Y, pixels);

    delete[] pixels;
    cudaFree(dev_pixels);
      return 0;
}

void Draw_Surface_GPU() {
    cudaEvent_t start, stop;
    CUDA_Timer_Start(&start, &stop);
    dim3 dimGrid(GRIDX,GRIDY);
    dim3 dimBlock(BLOCKX,BLOCKY);
    dev_draw_surface <<<dimGrid,dimBlock>>>(
        dev_pixels, s);
    cudaMemcpy(pixels, dev_pixels, s,
        cudaMemcpyDeviceToHost);
    float elapsedTime = CUDA_Timer_Stop(&start, &
        stop);
    printf("GPU Time: %f ms\n", elapsedTime);
}
```

Резултатът от използването на функцията **__syncthreads())** е виден на долните фугури. В първия случай (ляво) не сме използвали синхронизация на процесите при което не всички изпълнявани процеси са били завършени преди пристъпване към изчисление на нов блок с данни. Във втроия случай (дясно) синхронизацията е използвани, което е видно по резултата.

За да видите разликата в изпълнението на програмата

Фигура 4.7: Резултати от използване на **__syncthreads()** при работа със споделената памет.



при неизползване и използване на **__syncthreads()** първо компилирайте кода без тази функция, а след това с нея. Тъй като програмата изчертава фигурата, като сегменти на отделните блокове от пиксели с размери 16x16, стойността на всеки пиксел се пресмята от отделен процес. Резултатът се записва в адреси **threadIdx.x** и **threadIdx.y** на квадратна матрица намираща се в споделената памет:

```
shared[15-threadIdx.x][15-threadIdx.y]=color;
```

След това всеки един пиксел от споделената памет се прехвърля в глобалната памет:

```
pix[pos]=shared[threadIdx.x][threadIdx.y];
```

Тъй като индексирането на масивите в споделената памет и прехвърлянето на данните в глобалната памет не става последователно, за някои пиксели изчислителния процес няма да е приключил, докато в това време друг достигнал до това място в кода процес ще записва данните в глобалнатапамет. Ако не използваме **__syncthreads()** резултатът ще е непълноценно изчисление съдържащо множес-

тво пропуски. Чрез въвеждането на синхронизация всички процеси в изчислителния блок ще спрат изпълнението си докато и последният активен процес не достигне до бариерата. Тази операция е често ползвана при работа със споделената памет, без нейното ползване много изчислителни алгоритми с CUDA няма да да работят коректно.

4.10 Прихващане на грешки в CUDA

Тази полезна функция може да ползвате при определяне на грешки възникнали в процеса на работа на вашите CUDA програми. Функцията приема за параметри поредната викана от вас функция, името на файла **__FILE__** в който тя се намира и номера на реда **__LINE__** във вашия код в който тази функция се намира. Функцията може да използвате при дебъгване на вашите програми за извеждане на възникнали грешки в процеса на работа. За да я използвате следва да я дефинирате или в отделен файл към вашата основна програма или в кода на съответната програма. В основата на нашата функция седи изброяването **cudaError_t**, която е дефинирана във файла **driver_types.h**. В него са изброени следните възможни грешки:

```
/*DEVICE_BUILTIN*/
enum cudaError
{
 cudaSuccess = 0,
 cudaErrorMissingConfiguration,
 cudaErrorMemoryAllocation,
 cudaErrorInitializationerror,
 cudaErrorLaunchFailure,
 cudaErrorPriorLaunchFailure,
 cudaErrorLaunchTimeout,
 cudaErrorLaunchOutOfResources,
 cudaErrorInvalidDeviceFunction,
```

```
    cudaErrorInvalidConfiguration,
    cudaErrorInvalidDevice,
    cudaErrorInvalidValue,
    cudaErrorInvalidPitchValue,
    cudaErrorInvalidSymbol,
    cudaErrorMapBufferObjectFailed,
    cudaErrorUnmapBufferObjectFailed,
    cudaErrorInvalidHostPointer,
    cudaErrorInvalidDevicePointer,
    cudaErrorInvalidTexture,
    cudaErrorInvalidTextureBinding,
    cudaErrorInvalidChannelDescriptor,
    cudaErrorInvalidMemcpyDirection,
    cudaErrorAddressOfConstant,
    cudaErrorTextureFetchFailed,
    cudaErrorTextureNotBound,
    cudaErrorSynchronizationerror,
    cudaErrorInvalidFilterSetting,
    cudaErrorInvalidNormSetting,
    cudaErrorMixedDeviceExecution,
    cudaErrorCudartUnloading,
    cudaErrorUnknown,
    cudaErrorNotYetImplemented,
    cudaErrorMemoryValueTooLarge,
    cudaErrorInvalidResourceHandle,
    cudaErrorNotReady,
    cudaErrorStartupFailure = 0x7f,
    cudaErrorApiFailureBase = 10000
};
/*DEVICE_BUILTIN*/
typedef enum cudaError cudaError_t;
```

За да използваме тази функция ще дефинираме следната кратка функция:

```
static void static void HandleError(cudaError_t
    err,const char *file,int file_line);
```

Функцията съдържа следния код:

```
static void HandleError(cudaError_t err,const
    char *file,int file_line) {
```

```
    if (err != cudaSuccess) {
      const char* er;
      er = cudaGetErrorString(err);
      printf(                          , er, file,
          file_line);
      exit( EXIT_FAILURE );
     }
  }
```

За да ползваме тази функция, когато извикваме CUDA функция без извикване на ядро-изчислителен блок ние следва да я предадем по следния начин:

```
HandleError(cudaFunction(...),__FILE__,__LINE__);
```

При възникване на грешка компилаторът ще изведе съответното съобщение, като ще ни бъде изведен файла и номера на програмния ред в който е възникнала грешката. Където **__FILE__** и **__LINE__** са стандартни макроси, които повечето компилатори би следвало да разпознават. За удобство може да обособите тази функция в отделен помощен файл, където да опишете и други често ползвани функции. При писането на по-малки програми това не е нужно, но може да окаже съществено преимущество при откриване на грешки в обемист код. В конкретните примери в този учебник тази функционалност не е ползвана.

4.11 Параметри на хардуера

При работа върху CUDA платформи е възможно много от тях да са оборудвани с няколко съвместими графични адаптори имащи и различни изчислителни възможности. В тези случаи е възможно вашата програма да трябва да избере най-оптималния хардуер за дадена задача. Също

така дадени секции от кода на вашата програма може да са написани по различен начин, например даден код може да е създаден за работа с атомарни операции и също така да сте предвидили функция със същата функционалност без поддръжка на атомарни операции. Ето защо е добре да можете да прочетете колко и какви устройства са инсталирани и достъпни във вашата система. Този пример е посветен именно на това.

Приложението ползва следния файл:

- **GUDA_GPU_Devices.cu**

Listing 4.15: Изходен код на файла "CUDA_GPU_Devices.cu".

```
#include "stdio.h"
void print_dev_prop(int dev_num);

int main( void ) {
    int dev_count;
    cudaGetDeviceCount(&dev_count);
    printf( "Fount %d CUDA devices installed on
        your system.\n", dev_count );
    for(int i=0; i<dev_count; i++) {
    print_dev_prop(i);
  }
}

void print_dev_prop(int dev_num){
  cudaDeviceProp  cudaProp;
  cudaGetDeviceProperties(&cudaProp, dev_num);
  printf("\n\tDevice %d properties:\n",dev_num);
  printf("\tDevice name: %s\n",cudaProp.name);
  printf("\tCompute capability: %d.%d\n",
      cudaProp.major, cudaProp.minor);
  printf("\tSpeed: %f GHz\n",(float)cudaProp.
      clockRate/1000000);
  printf("\tGlobal memory: %f MB\n",
    (float)cudaProp.totalGlobalMem/1000000);
```

```
    printf("\tConstant memory: %ld B\n",(long int)
        cudaProp.totalConstMem);
    printf("\tMaximum memory pitch: %ld\n",(long
        int)cudaProp.memPitch);
    printf("\tTexture alignment: %ld\n",(long int)
        cudaProp.textureAlignment);
    printf("\n\tNumben of multiprocessors: %d\n",
         cudaProp.multiProcessorCount);
    printf("\tShared memory per multiprocessor: %ld
         B\n",(long int)cudaProp.sharedMemPerBlock);
    printf("\tRegisters per multiprocessor: %d\n",
        cudaProp.regsPerBlock);
    printf("\tThreads in warp: %d\n",cudaProp.
        warpSize);
    printf("\tMaximum treads per block: %d\n",
        cudaProp.maxThreadsPerBlock);
    printf("\tMaximum thread dimension: x%d y%d z
            %d\n",cudaProp.maxThreadsDim[0],
        cudaProp.maxThreadsDim[1],cudaProp.
        maxThreadsDim[2]);
    printf("\tMaximum grid simension: x%d y%d z%d\n
        ", cudaProp.maxGridSize[0],
    cudaProp.maxGridSize[1],cudaProp.maxGridSize
        [2]);
    printf("\tChose device: %d\n", dev_num);
    if(cudaProp.major>=1 & cudaProp.minor>=1)
      cudaChooseDevice(&dev_num, &cudaProp);
    printf("\n");
}
```

Ако успешно сте стигнали до тук, вие вече можете да пишете елементарни изчислителни програми ползващи възможностите на CUDA. Запознахме се с ресурсите на видео картите (глобална RAM и локална кеш памет), работа с OpenGL за изчертаване на 2D изображения, научихте малко повече за използването на споделената памет и атомарните операции. Видяхте как може да реализирате собствен клас за прихващане на грешките възникнали при извикване на CUDA функциите, което може да бъде полезно при

работа с изчислително активни приложения. Възможността да определите параметрите на хардуера ще ви бъде полезна, когато пишете крос платформени приложения, които трябва да вървят върху различни поколения CUDA видео карти, като в зависимост от параметрите на картата изпълняват едни или други ядра от вашата CUDA програма или заделят и освобождават различни ресурси глобална памет и зареждат изчислителни блокове с различни димензии. В следващия раздел ще разгледаме малко повече за взаимната работа между CUDA и OpenGL, а също така и за това как да инсталирате и използвате OpenCV библиотеката, за да отваряте и обработвате с CUDA цифрови изображения и кадри от видео записи.

Глава 5

Обработка на изображения с OpenCV

В тази глава ще научите как да инсталирате и използвате OpenCV (Open Source Computer Vision) в Ubuntu, за да създавате собствени приложения за обработка на цифрови снимки и видео записи. Ще научите как да отваряте, визуализирате, и обработвате цифрови изображения и кадри от видео записи. В отделни примери ще бъдат разгледани възможностите за обработка на изображения с CUDA.

5.1 Инсталиране на OpenCV

OpenCV е безплатна софтуерна библиотека предназначена да улесни работата на приложните програмисти при създаване на бързодействащи и кросплатформени приложения за цифрова обработка на изображения. Софтуерът е достъпен свободно за научна и комерсиална работа. Библиотеката предлага интерфейс към следните езици за програмиране: C++, C, Python и Java. Софтуерът може да се използва в Windows, Linux, Android и Mac, което прави ваши-

те приложения кросплатформени. Библиотеката включва над 2500 оптимизирани примитиви на функции за цифрова обработка на изображения. С нея лесно може да пишете програми използващи цифрови видеокамери, обработващи цифрови видеозаписи и други изображения. В този учебник ще ползваме OpenCV основно, за да можем да получим достъп до функции за отваряне и съхраняване на файлове с цифрови изображения, а също така и работа с цифрови видео записи в Ubuntu. Без подобна библиотека разработката на професионални пролжения за обработка на изображения ще бъде силно затруднено, ето защо ползването на OpenCV е гаранция за успех на вашите проекти. Независимо от това, че библиотеката притежава достатъчен набор от готови функции нашата основна цел ще бъде да покажем как може да я използваме за осигуряване на изходните дзанни за тестването на високопроизводителни паралелни алгоритми за обработка на изображения с CUDA. Тъй като OpenCV ползва C++ компилатор, ако още не сте го инсталирали във вашата система ще трябва да инсталирате първо него.

```
$ sudo apt-get install g++
```

OpenCV изисква GTK+ 2.0 или по-висока версия, за да може да визуализира изображенията. Библиотеката GTK е ви предоставя универсален интерфейс за работа с графични файли. За да го видите коя версия на GTK+ е инсталирате във вашата система използвайте по-долната команда:

```
dpkg -l | grep libgtk
```

Ако тази команда не даде никакъв изход в терминала, вие ще трябва да инсталирате GTK+. За да пристъпите към инсталиране въведете следната команда.

```
$ sudo apt-get install libgtk2.0-dev
```

За да инсталирате самата OpenCV следва да използвате Synaptic Package Manager - помощната програма за инсталиране на пакети в Ubuntu. (System> Administration> Synaptic Package Manager). След като отворите инсталатора на пакети потърсете "opencv" и инсталирайте следните основни пакети:

- libcv
- libcv-dev
- libcvaux
- libcvaux-dev
- libhighgui
- libhighgui-dev
- opencv-doc

След приключване на инсталирането въведете следните редове в терминала, те ще направят видими за системата основните пътища към файловете на библиотеката:

```
export LD_LIBRARY_PATH=/home/opencv/lib
export PKG_CONFIG_PATH=/home/opencv/lib/pkgconfig
```

Може да проверите местоположението на библиотеката с командите:

```
$ pkg-config --cflags opencv
-I/usr/include/opencv

$ pkg-config --libs opencv
-lcv -lhighgui -lcvaux -lml -lcxcore
```

Информацията получена с тези команди ще използваме в нашите приложения в процесът на компилиране, за

да укажем къде се намират изходните файлове на библиотеката. За да компилираме успешно OpenCV приложение трябва да използваме следната команда:

```
g++ -I/usr/include/opencv -lcxcore -lhighgui -lm
        -o
```

5.2 Визуализация и съхраняване на изображения

За да проверим дали успешно сме инсталирали OpenCV ще създадем една базова програма, която може да намерите в повечето интернет сайтове, като първи пример за използването на библиотеката. Програмата ще може да отваря изображение във форматите: jpg, bmp, png, като визуализира резултата в прозорец и съхранява копие на отворения файл във форматите: bmp, png, jpg.

Listing 5.1: Отваряне и съхраняване на изображения с OpenCV.

```
#include <cv.h>
#include <highgui.h>
#include <stdio.h>

#define CV_IMWRITE_JPEG_QUALITY 1

int main(int argc, char *argv[]) {
  int p[3];
  p[0] = CV_IMWRITE_JPEG_QUALITY;
  p[1] = 10;
  p[2] = 0;

  IplImage* img=0;
  IplImage* img_bw=0;
  printf("Hello\n");
```

```
if(argv[1] != 0){
  img = cvLoadImage(argv[1], 1); // 0-BW
  cvSaveImage( "image.bmp", img,0);
  cvSaveImage( "image.png", img,0);
  cvSaveImage( "image1.jpg", img,p);
  p[1] = 10;
  cvSaveImage( "image2.jpg", img,p);
  img_bw = cvCreateImage(cvGetSize(img),
      IPL_DEPTH_8U,1);
  cvCvtColor(img,img_bw,CV_RGB2GRAY);
  cvSaveImage( "image2bw.jpg", img_bw,p);
  }
else
        printf("Enter filename\n");

if(img != 0) {
  cvNamedWindow("Display", CV_WINDOW_AUTOSIZE);
  cvShowImage("Display", img);
  cvWaitKey(0);
  cvDestroyWindow("Display");
  }
else
  printf("File not found\n");
    return 0;
}
```

За да стартирате програмата трябва да използвате терминала. Като параметър при стартирането на програмата следва да подадете името и пътя на произволен файл (JPG, BMP, PNG или друг) по следния начин:

```
$ ./load_save_image New.jpg
```

Където New.jpg е името на съответното изходно изображение, което по подразбиране би следвало предварително да сте поместили в същата папка, където се намира вашата програма. Имайте предвид, че програмата автоматично ще преоразмери прозореца с оригиналния размер на изображението, което ще отваряте. След изпълнението и в ра-

ботната папка ще бъдат генерирани още 4 изображения с различни разширения, като едно от тях ще бъде черно бяло с ниско качество. Този пример илюстрира колко бързо и лесно бихте могли да ползвате функциите на OpenCV, за да отваряте и съхранявате изходни изображения записани в различни файлови формати във вашите програми за обработка на данни.

В този код се използват два указателя към отворените изображения от типа IplImage. Тези указатели ще ползваме, за да предаваме, катож параметър местоположението на заредените и съхранявани изображения между функциите на OpenCV. Функцията **cvLoadImage("filename",1)** приема 2 параметъра: името на файла който да заредим и дали да заредим изображението, като сиво полутоново-0 или цветно-1. След тази команда указателят **img** вече ще сочи към областта от паметта в която се намира матрицата с байтове съответстващи на зареденото изображение. В следващите три реда използвайки функцията **cvSaveImage** съхраняваме последователно зареденото изображение в три различни файлови формата: bmp, png и jpg. По подразбиране в нашия код използваме масивът **int p[3]**, за да указваме параметрите на изображението, вторият елемент от този масив указва за качеството на jpg изображенията, което варира от 0 of 100 (по подразбиране то е 95). След промяна на параметъра указващ качеството на изображението ние извършваме съхраняване на повторно негово копие в jpg формат. След това използвайки функцията **cvCreateImage** създаваме копие на изходното изображение и инициализираме указателя **img_bw** с неговото местоположение. С следващата команда извършваме преобразуване на изображението в сиво полутоново чрез функция **cvCreateImage(cvGetSize(img),IPL_DEPTH_8U,1);**. След като вече имаме черно бяло копие на изходното изображение ние съхраняваме това копие, като jpg файл. С

функциите: **cvNamedWindow(...), cvShowImage(...), cvWaitKey(...)** и **cvDestroyWindow(...)** последователно създаваме и изобразяваме прозореца в който да покажем заредения файл, инициализираме функция изчакваща до натискане на бутон "0" от клавиатурата и унищожаваме прозореца.

5.3 Достъп до пикселите на изображението

След като имаме възможност да отваряме и съхраняваме файлове на изображения съхранявани в различни формати, може да пристъпим към обработката на тези изображения с CUDA. За нуждите на този пример ще създадем приложение, което да отваря изображение, да го конвертира в сиво полутоново и да изчислява хистограмата му. Изчислението на хистограмите на изображенията е често срещан метод за цифрова обработка. Хистограмата представя информация за честотата на поява на пиксели с определена яркост в рамките на цялото изображение. Съществуват разновидности на хистограмите на цифрови изображения ползвани за различни нужди. Болшинството хистограмни методи за обработка спадат към тъй наречените еднопикселни вероятностно статистически алгоритми за обработка на цифрови изображения. За да изчислим яркостната хистограма на изображението ние първо трябва да го трансформираме в сиво полутоново. Един от най-разпространените формати за съхраняване на цветни цифрови изображения и фотографии е RGB (red, green, blue). Този формат е масово използван във видео мониторите и компютърните дисплей, тъй като отчита спецификата на човешкото зрение да може да интерпретира различни комбинации от тези три цвята (излъчвани от монитора), ка-

то останалите цветове. *(Повече за другите видове цветни формати може да намерите в интернет, като търсите следните ключови думи: "Color Models". Освен RGB съществуват и други цветни модели, като: CMYK, LAB, HSV, HSL, NCS и др.).* Един от най-елементарните начини да конвертирате едно цветно изображение в сиво полутоново е да осредните стойностите на отделните канали ползвани за представяне на червения, зеления и синия канал:

$$Grey = \frac{(R + G + B)}{3} \tag{5.1}$$

Един по-коректен модел отразяващ чувствителността на човешкото зрение към интензитета на отделните цветови компоненти е:

$$Grey = 0.3 * R + 0.59 * G + 0.11 * B \tag{5.2}$$

Ще реализираме този код за изпълнение върху CPU и GPU, а също така и чрез готовите функции на OpenCV. (OpenGL може също да бъде ползван за елементарни обработки на изображения, включително изчисляване на хистограми и линейна филтрация.) Ще сравним резултатите и времето за изпълнение на всяка една от трите реализации. Базовият код, който ни позволява да отваряме и кондертираме цветни изображения в сиви полутонови е даден подолу. За да компилирате кода настройте от менюто build на geany следните опции **g++ -I/usr/include/opencv -lcxcore -lhighgui -lm "%f" -o "%e"**.

Listing 5.2: Конвертиране на RGB изображение в сиво полутоново чрез OpenCV.

```
#include <stdio.h>
#include <time.h>
#include "cv.h"
#include "highgui.h"
```

```
int main(int argc, char *argv[]){
  int i;
  clock_t start, end;
    IplImage *img_RGB = cvLoadImage( argv[1],
        CV_LOAD_IMAGE_COLOR );
    IplImage *img_BW;
    int width  = img_RGB->width;
    int height = img_RGB->height;

    img_BW = cvCreateImage( cvSize( width, height
        ), IPL_DEPTH_8U, 1 );
    start = clock();
    for(i=0; i<100;i++)
    cvCvtColor( img_RGB, img_BW, CV_RGB2GRAY );
    end = clock();
  printf("Time to convert [%d]x[%d] pixels RGB to
      BW: %f us\n", width, height,(float)(end-
      start)/100);

    cvShowImage("RGB Image", img_RGB);
    cvWaitKey(0);
    cvShowImage("Greyscale Image", img_BW);
    cvWaitKey(0);
  cvDestroyWindow("Greyscale Image");
  cvDestroyWindow("RGB Image");
}
```

Програмата може да стартирате от терминала със следната команда, където името и разширението на файла на изображението може да бъде в зависимост от наличното ви изображение, което трябва да се намира в папката на приложението:

```
$ ./RGB-Grey GreenCuda.jpg
```

Кодът на тази програма ползва следните ключови функции: **cvLoadImage(...)** за зареждане на изображението, **cvCreateImage(...)** за създаване на ново изображение, което има същите параметри, като зареденото, **cvCvtColor(...)** за конвертиране на RGB изображението в сиво полутоно-

во, **cvShowImage(...)** за да изобрази последователно оригиналното цветно изображение и функцията **cvWaitKey(...)** която ще изчака до натискането на клавиш, за да продължи програмата. Програмата извършва 100 поледни преобразувания на чветното изображение в сиво полутоново, след което изчислява средното време за еднократно конвертиране на изображението в микросекунди. Това се прави с цел по-прецизно определяне на времето нужно за конвертиране на дадено изображение. Скоростта ще зависи основно от неговите размери, а също така и от производителността на вашия CPU. Тази програма ползва готовите функции на OpenCV.

В разгледания по-горе код функциите по извличане и конвертиране и цветното изображение в сиво полутоново става изцяло чрез функциите на OpenCV. Ще модифицираме така кода, че да можем да извлечем RGB стойностите на всеки пиксел от матрицата на изходното изображение и след това да извършваме изчисление съгласно формула (5.2).

Listing 5.3: Преобразуване на RGB изображение в сиво полутоново.

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <sys/mman.h>
#include "cv.h"
#include "highgui.h"
//g++ -I/usr/include/opencv -lcxcore -lhighgui -
    lm "%f" -o "%e"
//g++ "%f" -o "%e" -I/usr/include/opencv -lcxcore
     -lhighgui -lm  -lopencv_stitching -
    lopencv_gpu
void RGB_BW_CPU(unsigned char *p, int *w, int *h,
     int *ch, int *s);

int main(int argc, char *argv[]){
```

```
int i;
unsigned char *img_d;

clock_t start, end;
  IplImage *img_RGB = cvLoadImage( argv[1],
     CV_LOAD_IMAGE_COLOR );
  IplImage *img_BW;
  IplImage *img_BW_CPU;
  int width  = img_RGB->width;
  int height = img_RGB->height;

  int nchannels = img_RGB->nChannels;
int step       = img_RGB->widthStep;
printf("Image channels %d\n",nchannels);
printf("Image step %d\n",step);

  img_BW = cvCreateImage( cvSize( width, height
      ), IPL_DEPTH_8U, 1 );
  img_BW_CPU = cvCreateImage( cvSize( width,
      height ), 8, 3 );
  cvCopy(img_RGB, img_BW_CPU);

  unsigned char *ptr = (unsigned char*)
     img_BW_CPU->imageData;
//  mlock(ptr,height*step*width);
  start = clock();
  for(i=0; i<100; i++)
    RGB_BW_CPU(ptr, &width, &height, &nchannels
       , &step);
  end = clock();
//  munlock(ptr,height*step*width);

int size = height*step * width*nchannels;
  printf("Time to convert CPU [%d]x[%d] pixels
     RGB to BW: %f us\n", width, height,(float)
     (end-start)/100);
  cvShowImage("CPU Grey Image", img_BW_CPU);
  cvWaitKey(0);

  start = clock();
  for(i=0; i<100;i++)
  cvCvtColor( img_RGB, img_BW, CV_RGB2GRAY);
```

```
    end = clock();
  printf("Time to convert OpenCV [%d]x[%d] pixels
      RGB to BW: %f us\n", width, height,(float)(
     end-start)/100);

    cvShowImage("RGB Image", img_RGB);
    cvWaitKey(0);
    cvShowImage("Greyscale Image", img_BW);
    cvWaitKey(0);
    cvDestroyWindow("CPU Grey Image");
  cvDestroyWindow("Greyscale Image");
  cvDestroyWindow("RGB Image");
  cvReleaseImage(&img_RGB);
  cvReleaseImage(&img_BW);
}

void RGB_BW_CPU(unsigned char *p, int *w, int *h,
    int *ch, int *s){
  int r, g, b, y, x, BW, posx, posy, pos;
  posy = 0;
  r = g = b = 0;
  for(y=0 ; y<*h; y++) {
    posx = 0;
    for(x=0 ; x<*w; x++) {
      posx = x+x+x;
      pos = posy + posx;
      r = p[pos];
      g = p[pos + 1];
      b = p[pos + 2];
      BW = ( r + g + b )/ 3;
    //  BW = (30*r+59*g+11*b)/100;
      p[pos] = BW;
      p[pos + 1] = BW;
      p[pos + 2] = BW;
    }
    posy = posy + *s;//*w*3;
    }
}
```

Като цяло този код има един съществен недостатък, достъпът до елементите на изображението с указател не е

достатъчно бърз. Това означава, че така написаното приложение, освен ако не е изрично необходимо да обработваме пикселите му чрез указател ще бъде преобразувано в сиво полутоново по-бързо с вградените функции на OpenCV. Проблемът основно се състои в това, че пикселите съдържат по 3 байта информация, които ние четем и записваме отделно. Това формира 6 поредни четения и запис в масива на изходното изображение. Следващата стъпка в подобряването на програмата ще бъде свързан с имплементацията на алгоритъма върху CUDA. Преди това обаче ще се запознаем с разработката на приложения на C, които използват свързани функционалности на CUDA код. Този подход ще ни позволи да компилираме в едно приложение код изпълняван върху CUDA и върху CPU.

5.4 Смесване на CUDA код в C приложение

Често може да ви се наложи да вмъкнете функционалности на CUDA код в C приложение. Типичен пример е комбинирането на различни функционални библиотеки, като например OpenCV с CUDA kernel. За целта първо ще разгледаме едно елементарно приложение демонстриращо ползването на CUDA код в C програми. Това приложение ще дублира функционалността на нашето първо приложение **Hello World from CUDA!**. Този проект ще има три основни файла: **kernel.cu, h.h, main.c**. Това приложение има и друга особеност, то ще се компилира с т.н. **makefile**. Това е специален тип файл, който се използва за компилиране на сложни C програми в Linux. Най-общо казано този файл съдържа последователностите от инструкции, които би следвало да извършим ръчно една след друга, за да компилираме и свържем няколко изходни обектни файла

в едно приложение. Съдържанието на файловете е както следва:

Listing 5.4: Съдържание на соновния main.c файл съдържащ кода на C програмата.

```
#include "h.h"

extern void kernel_wrapper(char *ch, int n, char
    code);

int main(int argc, char *argv[]){
  int i;
    char code = 0x21;
  char str_h[] = "Hello World from CUDA!";

  for(i = 0; i<22; i++)
    str_h[i] = str_h[i] ^ code;

    kernel_wrapper(str_h, 22, code);

  printf("%s\n",str_h);
    return 0;
}
```

Listing 5.5: Описание на kernel.cu файла съдържащ нашият CUDA kernel.

```
#include "h.h"

extern "C" void kernel_wrapper(char *ch, int n,
    char code);

__global__ void kernel(char *ch, int n, char code
    ){

  int idx = blockIdx.x * blockDim.x + threadIdx.x
      ;
  if(idx<n)
  ch[idx] = ch[idx] ^ code;
}
```

```
void kernel_wrapper(char *ch, int n, char code){
  printf( "Starting kernel execution\n" );
  char *str_d;
    cudaMalloc((void**)&str_d,n);
  cudaMemcpy(str_d, ch, n, cudaMemcpyHostToDevice
      );
  dim3 dimGrid(2);
    dim3 dimBlock(n/2);

    kernel <<<dimGrid,dimBlock>>>(str_d, n, code)
        ;

    cudaMemcpy(ch, str_d, n,
        cudaMemcpyDeviceToHost);
    printf( "Finish kernel execution\n" );
  cudaFree(str_d);
}
```

Listing 5.6: Описание на общия хедър файл h.h.

```
#ifndef __H_H_
#define __H_H_
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda.h>
#include <cuda_runtime.h>
#endif
```

Съдържанието на **makefile** е дадено по-долу, за да компилирате приложението всички горепосочени файлове, заедно с **makefile** трябва да бъдат в една папка. Самото компилиране се извършва с написване на uf;dkodjd: **$ make** в терминала когато се намирате в папката на проекта. Този файл ще изпълни последователно следните действия: първо ще бъдат създадени обектните файли: **main.o kernel.o**, след което ще бъде извикана командата gcc -lm -L /usr/local/cuda/lib -lcudart -o run main.o kernel.o. Резултатният изпълним файл ще се намира в същото папка и ще носи името: **run**. След

като компилацията и свързването мине успешно в папката на проекта ще се появят два нови файла **main.o kernel.o** и изпълнимия файл **run**.

Listing 5.7: Съдържание на makefile за компилиране и свързване на проекта.

```
run: main.o kernel.o
  gcc -lm -L /usr/local/cuda/lib -lcudart -o run
      main.o kernel.o

main.o: main.c h.h
  gcc -lm -I /usr/local/cuda/include -c -o main.o
       main.c

kernel.o: kernel.cu h.h
  nvcc -c -o kernel.o kernel.cu
```

За да стартирате приложението изпълнете командата: **$ ./run** резултатът следва да прилича на дадения по-долу:

```
...$ ./run
Starting kernel execution
Finish kernel execution
Hello World from CUDA!
```

Този сложен метод на компилиране се ползва често при писането на големи програми в Linux. Нещо повече проектите могат да бъдат обособени в отделни обектни файлове (с разширение *.o). По такъв начин ако някакъв отделен сегмент от вашата програма бъде обособен там няма да е нужно той да се компилира след всяка промяна на кода в другите файли, а само тогава, когато сте извършили промяна в изходния му код. Това пести време особено при работа с големи проекти. В следващия пример ще използваме подобен тип миксиране на CUDA код със стандартна C++ програма ползваща библиотеката OpenCV. Проектът може да компилирате и през geany като в Set includes and arguments в полето build напишете make.

5.5 Обработка на изображението с CUDA

Ако сте стигнали до тук и все още този учебник не ви се струва скучен следва да знаете, че се намирате на прага на радикални промени във вашите умения да пишете бързи и ефективни програми. След като изпълните следващия пример вие ще притежавате пълният технологичен арсенал от умения позволяващ ви да пишете сложни програми за графична и видео обработка работещи под Linux и изпълнявани върху CUDA. Следващите примери в учебника са оптимизирани за видео карти с изчислителна архитектура от 1.2 нагоре. Писането на сложен CUDA код работещ бързо върху архитектура 1.1 и по-стари версии изисква много детайлни познания за функционирането на CUDA kernel и ползването на паметта. Примерите (след този) ползват атомарни изчислителни операции за работа с глобалната видео памет и локалната кеш памет на графичните мултипроцесори. Тези операции спестяват големи главоблъсканици при трансформирането на класически C/C++ изчислителен къд към CUDA. Възможно е примерите да не са реализирани съвсем оптимално и вие самите да създадете по-бързи техни прототипи. Основно е търсено по-високо бързодействие в сравнение с изпълнението на алгоритмите върху CPU. Всеки пример е придружен от стандартна C функция реализираща конкретната обработка и неоптимизиран CUDA kernel.

Първият пример ще ползва предходните два примера, за да реализира функционалност за преобразуване на RGB изображение в сиво полутоново с CUDA kernel. За целта основната функционалност на програмата ще остане същата, като в kernel.h файла ще бъде копирано съдържанието на масива с данни от пиксели за цветното изображение, ще бъде извикан kernel, който да извърши трансформаци-

ята в сиво полутоново и след това получените данни ще бъдат обратно върнати в основната програма за изобразяване. Този код ще ползва още две интересни функции: **mlock(...) munlock(...)**. С тези функции масивът с данни от изображението ще бъде заключен в основната памет на компютъра, така че данните да не бъдат кеширани върху хард диска, като виртуална памет. Този подход е особено ефективен при обработка на големи клъстери с данни върху CUDA. За изпълнение на тези операции CUDA предлага и специализирана функция, за която ще стане дума по-късно. Целта е данните веднъж заредени в паметта на компютъра да останат там до приключване изпълнението на CUDA kernel и връщането на резултата за изобразяване. Това няма да ви се наложи ако вашият компютър има над 2GB RAM и всичко което правите е да компилирате този пример, но в реалните системи, които обработват огромни масиви от данни - видео, графика и т.н. всяка програма ползва динамично заделена RAM памет. С целтаоптимизация на използването на паметта отделните и сегменти се кешират върху твърдия диск на компютъра. Това икономисва RAM нежна за други приложения, но същевременно забавя значително времето нужно ни за работа с тези масиви данни.

Listing 5.8: Съдържание на соновната програма.

```
#include "h.h"
#include <time.h>
#include <sys/mman.h>
#include "cv.h"
#include "highgui.h"

extern void kernel_image(unsigned char *pix, int
    size, int x, int y);
extern void kernel_image_32(int *pix, int size,
    int x, int y);

void RGB_BW_CPU(unsigned char *p, int *w, int *h,
```

```
    int *ch, int *s);
void Load_Convert_Display(int argc, char *argv[])
    ;

int main(int argc, char *argv[]){
  Load_Convert_Display(argc, argv);
    return 0;
}
void Load_Convert_Display(int argc, char *argv[])
    {
  int i;
  unsigned char *img_d;

  clock_t start, end;
    IplImage *img_RGB = cvLoadImage( argv[1],
        CV_LOAD_IMAGE_COLOR );
    IplImage *img_BW;
    IplImage *img_BW_CPU;
    int width  = img_RGB->width;
    int height = img_RGB->height;

    int nchannels = img_RGB->nChannels;
  int step      = img_RGB->widthStep;
  printf("Image channels %d\n",nchannels);
  printf("Image step %d\n",step);

    img_BW = cvCreateImage( cvSize( width, height
        ), IPL_DEPTH_8U, 1 );
    img_BW_CPU = cvCreateImage( cvSize( width,
        height ), IPL_DEPTH_8U, 3 );
    cvCopy(img_RGB, img_BW_CPU,0);
    unsigned char *ptr = (unsigned char*)
        img_BW_CPU->imageData;
    int *ptr_32 = (int*) img_BW_CPU->imageData;

  mlock(ptr,height*step*width);
    kernel_image(ptr, width*height*nchannels,
        img_BW_CPU->width, img_BW_CPU->height);

    start = clock();
    for(i=0; i<100; i++)
      RGB_BW_CPU(ptr, &width, &height, &nchannels
```

```
            , &step);

    end = clock();
    printf("NUmber of pixels %d \n",width*height*
        nchannels);
    munlock(ptr,height*step*width);

    int size = height*step * width*nchannels;
      printf("Time to convert CPU [%d]x[%d] pixels
          RGB to BW: %f ms\n", width, height,(float)
          (end-start)/100000);
      cvShowImage("CPU Grey Image", img_BW_CPU);
      cvWaitKey(0);

      start = clock();
      for(i=0; i<100;i++)
      cvCvtColor( img_RGB, img_BW, CV_RGB2GRAY);
      end = clock();
    printf("Time to convert OpenCV [%d]x[%d] pixels
        RGB to BW: %f ms\n", width, height,(float)(
        end-start)/100000);

      cvShowImage("RGB Image", img_RGB);
      cvWaitKey(0);
      cvShowImage("Greyscale Image", img_BW);
      cvWaitKey(0);
      cvDestroyWindow("CPU Grey Image");
    cvDestroyWindow("Greyscale Image");
    cvDestroyWindow("RGB Image");
    cvReleaseImage(&img_RGB);
    cvReleaseImage(&img_BW);
}

void RGB_BW_CPU(unsigned char *p, int *w, int *h,
      int *ch, int *s){
    int r, g, b, y, x, BW, posx, posy, pos;
    posy = 0;
    r = g = b = 0;
    for(y=0 ; y<*h; y++) {
      posx = 0;
      for(x=0 ; x<*w; x++) {
        posx = 3*x;
```

```
            pos = posy + posx;
            r = p[pos];
            g = p[pos + 1];
            b = p[pos + 2];
          //  BW = ( r + g + b )/ 3;
            BW = (30*r+59*g+11*b)/100;
            p[pos] = BW;
            p[pos + 1] = BW;
            p[pos + 2] = BW;
          }
          posy = posy + *s;//*w*3;
          }
    }
```

Следващия файл съдържа описание на експортираните kernel функции. Основната програма извиква функцията: **void kernel_image (unsigned char *pix, int size, int x, int y);** Тази функция се експортира от nvcc, като стандартна C функция, която може да бъде викана от основното приложение, все едно, че CUDA подпрограмата е описана в отделна динамично свързваща се библиотека. След процесът на последователно компилиране на стандартната C програма и CUDA kernel двата обектни файла ще бъдат сляти в един общ изпълним файл.

Listing 5.9: Съдържание CUDA kernel.

```
#include "h.h"
#include "CUDA_Timer.cu"

extern "C" void kernel_image(unsigned char *pix,
    int size, int x, int y);

__global__ void kernel(unsigned char *pix_rgb,
    unsigned char * pix_bw, int size, int x, int y
    ){

  int idx = threadIdx.x + blockIdx.x * blockDim.x
      ;
  int idy = threadIdx.y + blockIdx.y * blockDim.y
```

```
        ;
    int dx = idx * 3;
    int pos = dx + idy *( x *3) + idy*2;
    unsigned int R, G, B, BW;
    if(idx<x && idy<y) {
       B = pix_rgb[pos];
       G = pix_rgb[1+pos];
       R = pix_rgb[2+pos];
       BW = (30*R+59*G+11*B)/100;
       pix_bw[pos+0] = BW;
       pix_bw[pos+1] = BW;
       pix_bw[pos+2] = BW;
    }
}

void kernel_image(unsigned char *pix, int size,
    int x, int y){
    printf( "Starting kernel execution\n" );
    unsigned char *pix_rgb;
    unsigned char  *pix_bw;
    cudaEvent_t start, stop;
    CUDA_Timer_Start(&start, &stop);
      cudaMalloc((void**)&pix_rgb, size);
      cudaMalloc((void**)&pix_bw, size);

    cudaMemcpy(pix_rgb, pix, size,
        cudaMemcpyHostToDevice);
    dim3 dimGrid(1+x/16, 1+y/16);
      dim3 dimBlock(16,16);

    for(int i=0; i<100; i++)
      kernel <<<dimGrid,dimBlock>>>(pix_rgb, pix_bw
          , size, x, y);

      cudaMemcpy(pix, pix_bw, size,
          cudaMemcpyDeviceToHost);
      cudaFree(pix_rgb);
      cudaFree(pix_bw);
    float elapsedTime = CUDA_Timer_Stop(&start, &
        stop);
    printf( "Kernel time: %f ms\n", elapsedTime
        /100);
```

```
    printf( "Grid: %dx%d, block %dx%d \n", dimGrid.
        x, dimGrid.y, dimBlock.x, dimBlock.y);
}
```

Следващия листинг съдържа кода необходим и на двата сорс файла по-горе, в който са описани необходимите общи библиотеки.

Listing 5.10: Съдържание на соновния h.h файл съдържащ кода на C програмата.

```
#ifndef __H_H_
#define __H_H_
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda.h>
#include <cuda_runtime.h>
#endif
```

За да компилирате така подготвения проект, всички файли следва да се намират в една и съща папка, като с този make файла се задават последователните операции за компилация и свързване на обектните файлове, от които ще се свърже финалното приложение. След като веднъж е създаден този файл за прекомпилация на проекта в конзолата следва да извиквате единствено командата **make**.

Listing 5.11: Съдържание на makefile нужен за компилация на проекта.

```
run: main.o kernel.o
  gcc -lcxcore -lhighgui -lm -L /usr/local/cuda/
      lib -lcudart -o run main.o kernel.o

main.o: main.c h.h
  gcc -I /usr/include/opencv -lcxcore -lhighgui -
      lm -I /usr/local/cuda/include -c -o main.o
      main.c
```

```
kernel.o: kernel.cu h.h
  nvcc -c -o kernel.o kernel.cu
```

Изходното приложение ще има името run, като стартирането му следва да става по следния начин от конзола:

```
$ ./run cuda.jpg
```

За да сравним времето нужно за изпълнение на операцията чрез стандартната OpenCV функционалност, която се изпълнява върху CPU и току що написания елементарен CUDA kernel следва да използваме таймери и осредняване на резултатите за поне 100 поредни изчисления. Финалният резултат не е особено обнадеждаващ, по-долу е даден резултата от изпълнението на трите версии на операцията за конвертиране на цветно RGB изображение в сиво полутоново.

```
Kernel time: 2.374897 ms
Time to convert CPU [950]x[633] pixels RGB to BW:
     11.50 ms
Time to convert OpenCV [950]x[633] pixels RGB to
   BW: 2.50 ms
```

Съществениият проблем, който можем да забележим тук, е че стандратната OpenCV функционалност е написана достатъчно оптимално, така че нейната обикновенна CPU C версия се справя далеч, по-бавно от нея. Нашият CUDA kernel работи съвсем забележимо по-бързо.

5.6 Работа с видео файлове

Преди да пристъпите към ползването на OpenGL за работа с видео файлове се увурете, че в системата ви са инсталирани съответните видео декодиращи и кодиращи програми

и библиотеки за работа с различни формати изображения и видео файли. В терминала изпълнете следната команда:

```
sudo apt-get install build-essential libgtk2.0-
    dev libjpeg62-dev
libtiff4-dev libjasper-dev libopenexr-dev cmake
    python-dev python-numpy libtbb-dev libeigen2-
    dev yasm libfaac-dev
libopencore-amrnb-dev libopencore-amrwb-dev
    libtheora-dev libvorbis-dev libxvidcore-dev
```

И след това изтеглете, компилирайте и инсталирайте нужните ви кодеци, тази процедура може да отнеме повече време:

```
cd ~
wget http://ffmpeg.org/releases/ffmpeg-0.7-rc1.
    tar.gz
tar -xvzf ffmpeg-0.7-rc1.tar.gz
cd ffmpeg-0.7-rc1
./configure --enable-gpl --enable-version3 --
    enable-nonfree --enable-postproc --enable-
    libfaac --enable-libopencore-amrnb
 --enable-libopencore-amrwb --enable-libtheora --
     enable-libvorbis --enable-libxvid --enable-
     x11grab --enable-swscale --enable-shared
make
sudo make install
```

5.7 Обработка на видео файлове с CUDA

След като се научихме да отваряме и обработваме изображения с CUDA и OpenCV ще модифицираме малко горния пример, за да се научим как да ползваме OpenCV за работа с видео файлове. В този пример ще разгледаме приложение, което се стартира от командния ред, като параме-

тър му се предава името на видео файл който да обработва. Приложението отваря файла и обработва изображението по четири различни метода: с вградените функции на OpenCV, с код написан от нас работещ върху CPU и GPU.

Този пример отново включва смесването на C++ код с CUDA код. Поради тази причина компилирането се осъществява отново с makefile. Структурата и имената на файловете остава същата, изменяме само main.c файла, така че да можем вместо изображения да зареждаме видео файл от който да прочитаме последователни кадри, да ги конвертираме в черно бели и да ги визуализираме в четири под прозореца.

Listing 5.12: Съдържание на соновната програма.

```
#include "h.h"
#include <time.h>
#include <sys/mman.h>
#include "cv.h"
#include "highgui.h"

extern void kernel_image(unsigned char *pix, int
    size, int x, int y);
extern void kernel_image_32(int *pix, int size,
    int x, int y);

void RGB_BW_CPU(unsigned char *p, int *w, int *h,
     int *s);
int Load_Convert_Display(int argc, char *argv[]);

int main(int argc, char *argv[]){
  Load_Convert_Display(argc, argv);
    return 0;
}
int Load_Convert_Display(int argc, char *argv[]){
  int i;
  int key;
  unsigned char *img_d;
  int width;
  int height;
```

```
int step;
int nchannels;
unsigned char *ptr;

clock_t start, end;
  IplImage *frame_RGB;
  IplImage *frame_BW;
  IplImage *frame_BW_CPU;
  IplImage *frame_BW_GPU;

assert( argc == 2 );
CvCapture *capture = cvCaptureFromAVI( argv[1]
    );


if( !capture ) return 1;

  /* get fps, needed to set the delay */
  int fps = ( int )cvGetCaptureProperty(
      capture, CV_CAP_PROP_FPS );
  width  = ( int )cvGetCaptureProperty( capture
      , CV_CAP_PROP_FRAME_WIDTH );
height = ( int )cvGetCaptureProperty( capture,
   CV_CAP_PROP_FRAME_HEIGHT );
CvSize size = cvSize(width,height);

CvVideoWriter* writer = cvCreateVideoWriter("
    MyVideo.avi",CV_FOURCC('D','I','V','X'),fps,
    size,IPL_DEPTH_8U);

  cvNamedWindow( "video original", 0 );
      cvMoveWindow("video original", 10, 10);
  cvNamedWindow( "video OpenCV", 0 );
      cvMoveWindow("video OpenCV", 20 + width,
      10);
  cvNamedWindow( "video CPU", 0 );cvMoveWindow
      ("video CPU", 10, 65 + height );
  cvNamedWindow( "video GPU", 0 );cvMoveWindow
      ("video GPU", 20 + width, 65 + height);

  while( key != 'q' ) {
      /* get a frame */
```

```
    frame_RGB = cvQueryFrame(capture);
//  cvSetCaptureProperty(capture,
   CV_CAP_PROP_POS_FRAMES,i);
  //i = i +100;
    //    for(i=0;i<100;i++)
     //cvGrabFrame(capture);
        frame_RGB = cvQueryFrame(capture);//
            cvRetrieveFrame(capture);

      step    = frame_RGB->widthStep;
      width   = frame_RGB->width;
    height = frame_RGB->height;

      frame_BW = cvCreateImage( cvSize( width,
          height ), IPL_DEPTH_8U, 1 );
      frame_BW_CPU = cvCreateImage( cvSize( width
          , height ), IPL_DEPTH_8U, 3 );
      frame_BW_GPU = cvCreateImage( cvSize( width
          , height ), IPL_DEPTH_8U, 3 );

    start = clock();
    cvCvtColor( frame_RGB,frame_BW, CV_RGB2GRAY )
        ;
    end = clock();
    printf( "OpenCV time: %f ms\n", (float)(end-
        start)/100000);

    cvCopy(frame_RGB, frame_BW_CPU,0);
      ptr = (unsigned char*)frame_BW_CPU->
          imageData;
      start = clock();
        RGB_BW_CPU(ptr, &width, &height, &step);
      end = clock();
      printf( "CPU time:    %f ms\n", (float)(end
          -start)/100000);

    cvCopy(frame_RGB, frame_BW_GPU,0);
    nchannels = frame_BW_GPU->nChannels;
      //mlock(ptr,height*step*width);
      ptr = (unsigned char*)frame_BW_GPU->
          imageData;
        kernel_image(ptr, width*height*nchannels,
```

```
                frame_BW_GPU->width, frame_BW_GPU->
                height);
        //munlock(ptr,height*step*width);

          if( !frame_RGB ) break;
          /* display frame */
          cvShowImage("video original", frame_RGB);
          cvShowImage("video OpenCV", frame_BW);
          cvShowImage("video CPU", frame_BW_CPU);
          cvShowImage("video GPU", frame_BW_GPU);

          cvWriteFrame( writer,frame_BW_GPU);

           /* quit if user press 'q' */
          key = cvWaitKey(1000 / fps );// wait
              1000/fps ms for input
      }

      cvReleaseCapture( &capture );
    cvDestroyWindow( "video original");
      cvDestroyWindow( "video OpenCV");
      cvDestroyWindow( "video CPU");
      cvDestroyWindow( "video GPU");
      cvReleaseImage(&frame_BW_CPU);
      cvReleaseImage(&frame_BW_GPU);
    cvReleaseImage(&frame_RGB);
    cvReleaseImage(&frame_BW);

    cvReleaseVideoWriter( &writer );

}

void RGB_BW_CPU(unsigned char *p, int *w, int *h,
     int *s){
    int r, g, b, y, x, BW, posx, posy, pos;
    posy = 0;
    r = g = b = 0;
    for(y=0 ; y<*h; y++) {
      posx = 0;
      for(x=0 ; x<*w; x++) {
        posx = 3*x;
        pos = posy + posx;
```

```
        r = p[pos];
        g = p[pos + 1];
        b = p[pos + 2];
      //  BW = ( r + g + b )/ 3;
        BW = (30*r+59*g+11*b)/100;
        p[pos] = BW;
        p[pos + 1] = BW;
        p[pos + 2] = BW;
      }
      posy = posy + *s;//*w*3;
      }
}
```

Listing 5.13: Съдържание на CUDA kernel.

```
#include "h.h"
#include "CUDA_Timer.cu"

extern "C" void kernel_image(unsigned char *pix,
    int size, int x, int y);

__global__ void kernel(unsigned char *pix_rgb,
    unsigned char * pix_bw, int size, int x, int y
    ){

  int idx = threadIdx.x + blockIdx.x * blockDim.x
      ;
  int idy = threadIdx.y + blockIdx.y * blockDim.y
      ;
  int dx = idx * 3;
  int pos = dx + idy *( x *3) + idy*3;
  unsigned int R, G, B, BW;
  if(idx<=x && idy<=y) {
     B = pix_rgb[pos];
     G = pix_rgb[1+pos];
     R = pix_rgb[2+pos];
     BW = (30*R+59*G+11*B)/100;
     //__syncthreads();
     pix_bw[pos+0] = BW;
     pix_bw[pos+1] = BW;
     pix_bw[pos+2] = BW;
```

```
    }
}

void kernel_image(unsigned char *pix, int size,
    int x, int y){
//  printf( "Starting kernel execution\n" );
  unsigned char *pix_rgb;
  unsigned char  *pix_bw;
  cudaEvent_t start, stop;
  CUDA_Timer_Start(&start, &stop);
    cudaMalloc((void**)&pix_rgb, size);
    cudaMalloc((void**)&pix_bw, size);
 //   printf( "GPU memory alocated\n" );

  cudaMemcpy(pix_rgb, pix, size,
      cudaMemcpyHostToDevice);
//  printf( "CPU to GPU memory copy\n" );
  dim3 dimGrid(1+x/16, 1+y/16);
    dim3 dimBlock(16,16);

  kernel <<<dimGrid,dimBlock>>>(pix_rgb, pix_bw,
      size, x, y);

    cudaMemcpy(pix, pix_bw, size,
        cudaMemcpyDeviceToHost);
 //   printf( "GPU to CPU memory copy\n" );
    cudaFree(pix_rgb);
 //   printf( "GPU rgb free\n" );
    cudaFree(pix_bw);
  float elapsedTime = CUDA_Timer_Stop(&start, &
      stop);
  printf( "GPU time:    %f ms\n", elapsedTime);
//  printf( "Grid: %dx%d, block %dx%d \n",
    dimGrid.x, dimGrid.y, dimBlock.x, dimBlock.y);
}
```

5.8 Съхраняване на резултатите

След като обработим даден видео файл, най-вероятно ще ни бъде необходимо не само да визуализираме резултатите от обработката, но и да съхраним обработените видео кадри в нов филм. Ползвайки OpenCV този процес е елементарен. Всички изменения в кода, които следва да направим са добавянето на следните 3 основни реда код:

CvVideoWriter* writer = cvCreateVideoWriter("Video.avi CV_FOURCC('D','I','V','X'),fps,size,IPL_DEPTH_8U);

cvWriteFrame(writer,frame_BW_GPU);

cvReleaseVideoWriter(&writer);

Всички промени в кода са отразени в main.c файла и не засягат CUDA kernel функциите, като така можем да ползваме по-горния пример. Резултатът от работата на тази програма ще бъде, че след нейното приключване в основната папка на проекта ви ще се намира един нов *.avi файл съдържащ изходния резултат от обработката на CUDA kernel функцията. Съществения смисъл на този пример е да покаже колко елементарно можете да ползвате OpenCV функциите, не само за да отваряте и обработвате видео файли, но и да съхранявате резултатите от вашата обработка.

Честито, вие вече разполагате с необходимия арсенал от инструменти и познания за създаване на сложни софтуерни приложения, които да могат да ползват възможностите на CUDA съвместимия хардуер и да обработват разнообразни цифрови изображения и дори видео записи. Комбинирайки научените в предните глави методи за обработка сравнително лесно бихте могли да модифицирате изходните кодове, така че вашите програми да ползват възможностите на CUDA например, за да изчислявате хистограми на цифрови изображения. Следва да имате предвид следните няколко съображения, когато решите дали да ползвате

CUDA или CPU: порвото нещо което следва да съобразите е до колко по-бързо работи вашият CUDA kernel за решение на конкретната задача и дали бихте могли да го модифицирате с цел оптимизация, но това не е основният проблем. Когато работите с CUDA винаги помнете, че данните биват преместени поне един или два пъти от RAM в паметта на видеокартата и обратно в RAM на хост компютъра. Към това време следва да добавите и времето за обработка на данните от вашия kernel. Например във видео обработката е резонно да се обмисли варианта няколко десетки кадри първо да бъдат прехвърлени от RAM на компютъра в паметта на видеокартата и след това ползвайки едни и същи функции те да бъдат обработени, като пакет. При това ефективността на вашата програма ще бъде толкова по-висока, колкото повече обработки ще може да извършите при едно копиране на изходните данни и връщането на рузултата в основната памет на компютъра. Например при работа с OpenCV особено при малки изображения времето за трансфер на данните е далеч над времето нужно за обработка от стандартната OpenCV функция, така че подобни обработки следва да се правят на CUDA хардуер само когато имате достатъчно основателна изчислителна задача, тоест много математически обработки изискващ изчисление на тригонометрически и други функции чието изпълнение върху CPU ще бъде неефективно.

Библиография

[1] David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*

[2] Jason Sanders, Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*

[3] Rob Farber, *CUDA, Supercomputing for the Masses*, DrDobb's 2008

[4] NVIDIA, *The CUDA Compiler Driver NVCC*

[5] Nicholas Paufler, *The 3D Chipset Wars: A Chronicle of the Past, Present, and Future*

[6] Addison Wesley, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, 2009

[7] П.Фолькердинг, К.Рейчард, Е. Фостер-Джонсон, *Установка и конфигурирование Linux*, 2000 изд. Питер

[8] Петров Г., *Приложение на CUDA при създаването на високопроизводителни паралелни системи за обработка на сигнали и изображения*, Телеком 2010,14-15 октомври NSTC, София, България

инж. Георги Петров в момента работи в „Телекомуникации“ в Нов български университет. През 2005 г. завършва магистърска програма “Мениджмънт на телекомуникациите” в НБУ, а през 2008 г. защитава дисертация на тема: “Многомерна цифрова обработка и анализ на последователности от изображения”. Основната му изследователска и преподавателска дейност е свързана с НБУ, където работи от 2003 г. Има опит в аутсорсинг проекти в областта на дизайн на преносими устройства и софтуер за запис и обработка на сигнали. Работил е като независим консултант към Центъра за усъвършенстване на кадри на Международния съюз по телекомуникации (ITU Женева) по програмите: Technology trends и Spectrum management. Автор и съавтор е на множество статии в интердисциплинарни направления. От 2002 г. се занимава професионално с програмиране на приложения за обработка на сигнали и изображения, а от 2008 г. с VHDL дизайн. Професионалните му интереси са свързани с развитието и внедряването на съвременни широколентови телекомуникационни системи, технологии и услуги. Разработва и въвежда първия университетски курс по програмиране за CUDA в България: TCMM159 Семинар: „Паралелна обработка на сигнали и изображения с CUDA“, който в момента се води в магистърска програма „Телекомуникации“ на НБУ. Курсът е отворен за свободно платено посещение по време на семинарите провеждани в департамента.

„Програмиране с CUDA за Ubuntu" е първият учебник на български език, който ще ви отвори портала към света на паралелни супер компютри базирани върху видо картите от серията GeForce на NVIDIA. Писането на паралелен C++ код никога досега не е било полесно и вълнуващо. Днес с операционна система Linux работят над 85% от супер компютрите по света, ето защо ползването на CUDA и Linux ще са от решаващо значение за вашето професионално развитие, като програмист на свръх бързи изчислителни и графични симулационни приложения.

Учебникът е подходящ за всички C/C++ програмисти, учени и инженери. Всичкият необходим ви софтуер е напълно безплатен и може да бъде свален от интернет. В учебника последователно са разгледани примери за приложението на CUDA при създаване на елементарни и по-сложни типови програми. Обърнато е внимание как да ползвате възможностите на OpenGL и OpenCV за създаване на графични приложения и обработката на цифрови изображения и видео филми.

Учебникът е предназначен за всички програмисти, инженери и учени, които желаят да пишат свои собствени програми, работещи върху масово достъпна хардуерна платформа - видео картите от серията GeForce. С тази технология можете да създадете свой собствен супер компютър, с който вашите изчислителни задачи да бъдат свършени прецизно и за много кратко време.

CUDA е достъпна и съвременна технология, усвояването и дори в не особена дълбочина ще ви позволи да ускорите от два до десет пъти бързината на вашите досегашни програми. В учебника се дават примери за на някои "трикове", с които да направите приложенията си дори още по-бързи използвайки специфични особености на новите изчислителни архитектури.

Ако вашата основна работа не е свързана с програмиране, познавайки възможностите на CUDA ще можете по-добре да работите и поставяте задачите на вашите колеги програмисти. Много езици за програмиране, като: C/C++, Java, Fortran, Python имат възможности да използват ресурсите на CUDA. Голям брой мобилни устройсва, лаптопи и таблети и дори мобилни телефони притежават видео ускорители поддържащи CUDA. Разширете своят арсенал на програмист с уменията да правите наистина бързи приложения работещи върху масово достъпна и популярна съвременна хардуерна архитектура за паралелни изчисления.