Методичні рекомендації до лабораторних робіт  
з дисципліни «Теорія і організація розподіленої обробки даних»  
варіант «Розробка програм GPGPU на базі технології CUDA»

ЗМІСТ

[Лабораторна робота №1. Знайомство з GPCPU на базі технології Nvidia Cuda 4](#_Toc500937826)

[Теоретичні відомості 4](#_Toc500937827)

[1. Вступ до GPCPU і причини його виникнення 4](#_Toc500937828)

[2. Основна різниця між CPU і GPU в паралельних розрахунках 6](#_Toc500937829)

[3. Використання паралельних обчислень на GPU 10](#_Toc500937830)

[4. Платформа паралельних обчислень Nvidia Cuda і її можливості 11](#_Toc500937831)

[5. Встановлення і налаштування середовища Cuda 11](#_Toc500937832)

[6. Створення перших проектів на базі технології Cuda 15](#_Toc500937833)

[Завдання до лабораторної роботи 16](#_Toc500937834)

[Посилання на джерела 17](#_Toc500937835)

[Лабораторна робота №2. Знайомство з ключовими моментами роботи з GPGPU на базі технології Nvidia Cuda 18](#_Toc500937836)

[Теоретичні відомості 18](#_Toc500937837)

[1. Модель програмування з використанням технології Cuda 18](#_Toc500937838)

[2. Розширення мови C для Cuda 21](#_Toc500937839)

[3. Основи Cuda host API 22](#_Toc500937840)

[4. Приклад практичного використання 23](#_Toc500937841)

[5. Розрахунок часу виконання 29](#_Toc500937842)

[Завдання до лабораторної роботи 30](#_Toc500937843)

[Посилання на джерела 30](#_Toc500937844)

[Лабораторна робота №3. Знайомство з роботою з пам’яттю у GPGPU на базі технології Nvidia Cuda 32](#_Toc500937845)

[Теоретичні відомості 32](#_Toc500937846)

[1. Відеокарта і типи пам'яті 32](#_Toc500937847)

[2. Налагодження програмного коду Cuda 34](#_Toc500937848)

[3. Практичне використання розділеної пам’яті 35](#_Toc500937849)

[Завдання до лабораторної роботи 39](#_Toc500937850)

[Посилання на джерела 39](#_Toc500937851)

[Лабораторна робота №4. Знайомство з оптимізацією пам’яті у GPGPU на базі технології Nvidia Cuda 40](#_Toc500937852)

[Теоретичні відомості 40](#_Toc500937853)

[1. Проблематика роботи з глобальною пам’яттю 40](#_Toc500937854)

[2. Вирівнювання розмірів використовуваних типів 40](#_Toc500937855)

[3. Використання об'єднаних запитів 42](#_Toc500937856)

[4. Структури масивів або масиви структур 46](#_Toc500937857)

[5. Проблематика доступу до пам’яті 47](#_Toc500937858)

[Завдання до лабораторної роботи 47](#_Toc500937859)

[Посилання на джерела 47](#_Toc500937860)

[Лабораторна робота №5. Знайомство з обробкою зображень на базі технології Nvidia Cuda 48](#_Toc500937861)

[Теоретичні відомості 48](#_Toc500937862)

[1. Робота з зображеннями за допомогою бібліотеки OpenCV 48](#_Toc500937863)

[2. Практична обробка зображення 48](#_Toc500937864)

[Завдання до лабораторної роботи 52](#_Toc500937865)

[Посилання на джерела 52](#_Toc500937866)

# Лабораторна робота №1. Знайомство з GPCPU на базі технології Nvidia Cuda

Мета роботи: загальне знайомство з технологією сімейства GPCPU і Nvidia Cuda, ознайомлення з засобами програмування на базі технології Cuda і їх налаштування, створення першої програми на базі Cuda SDK, ознайомлення з прикладами наведеними у GPU Computing SDK

## Теоретичні відомості

### Вступ до GPCPU і причини його виникнення

Не так давно паралельні обчислення перейшли до масового ринку пов'язаного з 3D графікою. Універсальні пристрої з багатоядерними процесорами для паралельних векторних обчислень, використовуваних в 3D-графіці, досягають високої пікової продуктивності, яка універсальним процесорам не під силу. Звичайно, максимальна швидкість досягається лише в ряді зручних завдань і має деякі обмеження, але такі пристрої вже почали досить широко застосовувати в сферах, для яких вони спочатку не призначалися.

Тобто, для 3D відео прискорювачів з'явилися перші технології неграфічних розрахунків загального призначення GPGPU (General-Purpose computation on GPUs). Оскільки відеочипи містять сотні математичних виконавчих блоків, і ця міць може використовуватися для значного прискорення безлічі обчислювально інтенсивних додатків. Сучасні покоління GPU мають гнучку архітектуру, що разом з високо рівневими мовами програмування і програмно-апаратними архітектурами розкриває ці можливості і робить їх значно доступнішими.

Створення GPCPU спонукала поява досить швидких і гнучких шейдерних програм, які здатні виконувати сучасні відеочипи.

Це спонукало виникненню ідеї застосування GPU не тільки для обрахунків зображення в 3D додатках, а й для проведення обрахунків в інших паралельних розрахунках. На першому етапі розвитку цього підходу використовувалися графічні API: OpenGL і Direct3D, коли дані до відеочипів передавалися у вигляді текстур, а розрахункові програми завантажувалися в вигляді шейдеров. Недоліками цього процесу є складність програмування, низька швидкість обміну даними між CPU і GPU і деякі інші обмеження.

Обчислення на GPU оказались життєздатним і почали активно розвиватися. Що спонукало виробників відеочипів анонсували відповідні платформи і засоби розробки для них. На відміну від попередніх моделей програмування GPU, ці були виконані з урахуванням прямого доступу до апаратних можливостей відеокарт. Це ліквідувало деякі з важливих обмежень попередніх моделей GPGPU, що використовують традиційний графічний конвеєр і відповідні інтерфейси Direct3D або OpenGL.

Випуск спеціалізованих засобів програмування від виготовлювачів відеокарт спонукав ще більший розвиток паралельних обчислень на відео картах. До сучасних засобів програмування відносяться:

– Nvidia Cuda (технологія GPGPU, що дозволяє реалізовувати на мові програмування Сі (а також C++ і C#) алгоритми, здійснимі на графічних процесорах прискорювачів GeForce);

– AMD FireStream (технологія GPGPU, що дозволяє реалізовувати алгоритми, здійснимі на графічних процесорах прискорювачів ATI);

– Microsoft DirectCompute (інтерфейс програмування додатків (API), який входить до складу DirectX і призначений для виконання обчислень загального призначення на графічних процесорах);

– OpenCL (галузевий стандарт для написання програм, пов'язаних з паралельними обчисленнями на графічних і центральних процесорах, що підтримується консорціумом Khronos Group, до якого входять компанії AMD, Apple, ARM, Intel, Nvidia, Sony Computer Entertainment та інші компанії).

– OpenMP (програмний стандарт для паралельного програмування, який з 4 версії включив в себе специфікації стандарту OpenACC, який призначений для роботи з прискорювачами, в тому числі GPU).

Окрім описаних існують і виникають нові засоби паралельного програмування для відеокарт. Але усі засоби умовно можна поділити на дві категорії. Перша з них засоби від виробників відеокарт, які забезпечують максимальну продуктивність при роботі з відеокартами певного виробника. Друга – універсальні засоби, які забезпечують сумісність майже з усіма наявними на ринку відеокартами.

### Основна різниця між CPU і GPU в паралельних розрахунках

Підвищення частот універсальних процесорів зіштовхнулося з фізичними обмеженнями. Ріст частоти процесорів призводить до збільшення кількості тепла, що виділяється при роботі процесора. Дана проблема як правило вирішувалась переходом на більш мілкий технічний процес. Більш малий технічний процес вимагає використання технології літографії з меншою довжиною хвилі. Якщо розміри елементів в кристалі мікросхеми відповідають розміру довжини хвилі то можливі аберації в обчислюваннях. Зменшення довжини хвилі поступово приводить до рентгенівського діапазону, що само за собою є граничною зоною можливостей технології літографії.

З цієї причини для підвищення продуктивності особливу увагу стали приділяти виробництву багато ядерних процесорів. Продукція, що зараз процесори містять лише ядра призначені для звичайних додатків, мають у свої основі MIMD (множинний потік команд і даних) архітекту. Кожне ядро працює окремо від інших, виконуючи різні інструкції для різних процесів.

Звичайно не будь яка задача може бути ефективно перенесена на багато ядерний процесор, проте існують класи задач в який такий підхід дає суттєвий виграш в швидкості.

Ідея багато ядерності процесорів може застосовуватися не тільки для CPU, але й має своє відображення в архітектурі графічних процесорів GPU. Для того щоб зрозуміти відмінність обрахунку даних розглянемо архітектору CPU і GPU.

В відеочипах основний блок це мультипроцесор з декількома десятками або сотнями ядер і сотнями або тисячами ALU в цілому, кількома тисячами регістрів і певною кількістю поділюваної загальної пам'яті. Крім того, відеокарта містить швидку глобальну пам'ять з доступом до неї всіх мультипроцесорів, локальну пам'ять в кожному мультипроцесорі, а також спеціальну пам'ять для констант.

Ядра мультипроцесора в GPU маю SIMD (одиночний потік команд, безліч потоків даних) архітектуру. Окрім цього ці ядра можуть виконувати одні і ті ж самі інструкції одночасно, такий стиль програмування є звичайним для графічних алгоритмів і деяких наукових завдань, але вимагає специфічного програмування. Зате такий підхід дозволяє збільшити кількість виконавчих блоків за рахунок їх спрощення.

Тобто основними відмінностями між архітектурами CPU і GPU є, те що:

– ядра CPU створені для виконання одного потоку послідовних інструкцій з максимальною продуктивністю, а GPU проектуються для швидкого виконання великої кількості паралельно виконуваних потоків інструкцій;

– універсальні процесори оптимізовані для досягнення високої продуктивності єдиного потоку команд, який займається обробкою і цілі числа і числа з плаваючою крапкою. При цьому доступ до пам'яті випадковий.

Розробники CPU намагаються домогтися виконання якомога більшої кількості інструкцій паралельно, для збільшення продуктивності. Для цього у процесорів з'явилося супер скалярне виконання, що забезпечує виконання декількох інструкцій за такт, а також виникнення технології позачергового виконанням інструкцій. Але у паралельного виконання послідовного потоку інструкцій є певні базові обмеження і збільшенням кількості виконавчих блоків кратного збільшення швидкості не добитися.

У відеочипів робота проста і роз паралельна спочатку. Відеочип приймає на вході групу полігонів, проводить всі необхідні операції, і на виході видає пікселі. Обробка полігонів і пікселів незалежна, їх можна обробляти паралельно, окремо один від одного. Тому, через спочатку паралельної організації роботи в GPU використовується велика кількість виконавчих блоків, які легко завантажити, на відміну від послідовного потоку інструкцій для CPU. Крім того, сучасні GPU також можуть виконувати більше однієї інструкції за такт (dual issue). Так, архітектури деяких сучасних чипів в деяких умовах запускають одночасно виконання операції MAD + MUL або MAD + SFU.

GPU має відмінності від CPU ще й за принципами доступу до пам'яті. В GPU він пов'язаний і легко передбачуваний - якщо з пам'яті читається тексель текстури, то через деякий час прийде час і для сусідніх текселей. Та й під час запису той же - піксель записується до фрейм буферу, і через кілька тактів буде записуватися розташований поруч з ним. Тому організація пам'яті відрізняється від тієї, що використовується в CPU. І відеочипів, на відміну від універсальних процесорів, просто не потрібна кеш-пам'ять великого розміру.

Також робота з пам'яттю у GPU і CPU дещо відрізняється. Так, не всі центральні процесори мають вбудовані контролери пам'яті, а у всіх GPU зазвичай мають багато контролерів. Крім того, на GPU застосовується більш швидка пам'ять, і в результаті відеочипу доступна в рази більша пропускна здатність пам'яті, що також дуже важливо для паралельних розрахунків, що оперують з величезними потоками даних.

В універсальних процесорах великі кількості транзисторів і площа чипу йде на буфери команд, апаратне передбачення розгалуження і величезні обсяги кеш-пам'яті на чипі. Всі ці апаратні блоки потрібні для прискорення виконання нечисленних потоків команд. Відеочипи витрачають транзистори на масиви виконавчих блоків; блоків керування потоками; невеликий обсяг пам'ять, що розділяється і контролери пам'яті на кілька каналів. Зазначені вище особливості не прискорюють виконання окремих потоків, але дозволяють чипу обробляти декількох тисяч потоків, одночасно виконуються чіпом і вимагають високої пропускної здатності пам'яті.

Окрім цих особливостей GPU і CPU мають відмінності в кешуванні. CPU використовують кеш-пам'ять для збільшення продуктивності за рахунок зниження затримок доступу до пам'яті, а GPU використовують кеш або загальну пам'ять для збільшення смуги пропускання. CPU знижують затримки доступу до пам'яті за допомогою кеш-пам'яті великого розміру, а також передбачення розгалужень коду. Ці апаратні частини займають більшу частину площі чипа і споживають багато енергії. Відеочипи обходять проблему затримок доступу до пам'яті за допомогою одночасного виконання тисяч потоків - в той час, коли один з потоків очікує даних з пам'яті, відеочип може виконувати обчислення іншого потоку без очікування і затримок. Є безліч відмінностей і в підтримці багато поточності.

CPU виконує 1-2 потоку обчислень на одне процесорний ядро, а відео чипи можуть підтримувати значно більше команд на кожен мультипроцессор. Якщо перемикання з одного потоку на інший для CPU коштує сотні тактів, то GPU переключає кілька потоків за один такт.

Крім того, центральні процесори використовують SIMD (одна інструкція виконується над численними даними) блоки для векторних обчислень, а відеочипи застосовують SIMT (одна інструкція і декілька потоків) для скалярної обробки потоків. SIMT не вимагає, щоб розробник перетворював дані в вектори, і допускає довільні розгалуження в потоках.

Коротко можна сказати, що на відміну від сучасних універсальних CPU, відеочипи призначені для паралельних обчислень з великою кількістю арифметичних операцій. І значно більше число транзисторів GPU працює за прямим призначенням - обробці масивів даних, а не керує виконанням (flow control) нечисленних послідовних обчислювальних потоків. Схема того, скільки місця в CPU і GPU займає різноманітна логіка наведено на рис. 2.1.

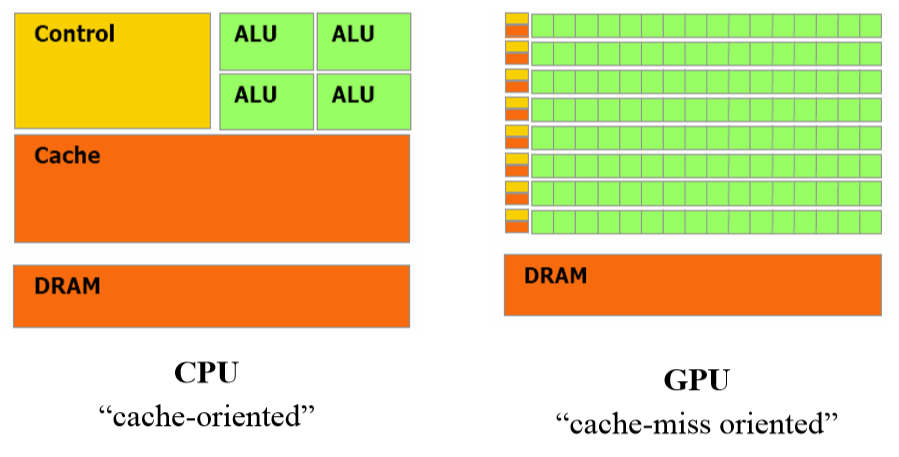


Рисунок 2.1 – Схема архітектури CPU і GPU

У підсумку, основою для ефективного використання потужності GPU в наукових та інших неграфічних розрахунках є розпаралелювання алгоритмів на сотні виконавчих блоків, наявних в відеочіпах. Наприклад, безліч додатків по молекулярному моделювання відмінно пристосоване для розрахунків на відеочіпах, вони вимагають великих обчислювальних потужностей і тому зручні для паралельних обчислень. А використання декількох GPU дає ще більше обчислювальних потужностей для вирішення подібних завдань.

Виконання розрахунків на GPU показує відмінні результати в алгоритмах, що використовують паралельну обробку даних. Тобто, коли одну і ту ж послідовність математичних операцій застосовують до великого обсягу даних. При цьому кращі результати досягаються, якщо відношення числа арифметичних інструкцій до числа звернень до пам'яті досить велике. Це пред'являє менші вимоги до управління виконанням (flow control), а висока щільність математики і великий обсяг даних скасовує необхідність у великих кешах, як на CPU.

В результаті всіх описаних вище відмінностей, теоретична продуктивність GPU значно перевершує продуктивність CPU. Але досягти таких показників на практиці дуже складна задача.

### Використання паралельних обчислень на GPU

В середньому, при перенесенні обчислень на GPU, у багатьох задачах досягається прискорення в 5-30 разів, у порівнянні з швидкими універсальними процесорами. Тому приклади використання обчислення на відеокартах можна знайти в програмах призначених для економічних і фінансових обчислень, моделюванні клімату, аналізу даних, оборони і розвідки, машинного навчання, обчислень промислових процесів, роботи з мультимедійним контентом, візуалізації медичних даних, пошуку природних копалин, дослідницьких завдань в науковій сфері, криптографічних алгоритмів, розпізнавання образів, тощо. Подробиці про ці та інші застосування обчислень на GPU можна знайти на сайті компанії Nvidia в розділі за технологією CUDA.

Як ми бачимо список застосувань обчислень на відеокартах досить великий, але і це ще не все оскільки він розширюється кожен день. Кожного дня знаходяться і інші області застосування паралельних розрахунків на відеочіпах.

### Платформа паралельних обчислень Nvidia Cuda і її можливості

Nvidia Cuda – це архітектура паралельних обчислень від Nvidia, що дозволяє істотно збільшити обчислювальну продуктивність завдяки використанню GPU (графічних процесорів).

Платформа паралельних обчислень Cuda забезпечує набір розширень для мов C і С++, що дозволяють висловлювати як паралелізм даних, так і паралелізм завдань на рівні дрібних і великих структурних одиниць. Програміст може вибрати засоби розробки: мови високого рівня, такі як C, C++, Fortran або ж відкриті стандарти, такі як директиви OpenACC. Платформа паралельних обчислень CUDA використовується на сьогоднішній день в тисячах GPU-прискорених додатків і тисячах опублікованих наукових статтях.

Повний список засобів розробки і екосистема рішень Cuda доступний на сайті Nvidia. А на порталі Cuda Zone можна дізнатися більше про існуючи розробки з Cuda.

### Встановлення і налаштування середовища Cuda

Встановлення середовища Cuda умовно можна поділити на декілька етапів. Перший включає в себе завантаження встановлення програмного забезпечення Nvidia. Другий – перевірку працездатності середовища Cuda. Третій – налаштування середовища програмування MS Visual Studio і перевірку працездатності встановлених налаштувань.

Ознайомитися з офіційною інструкцією встановлення можна на станиці офіційної документації ([http://docs.Nvidia.com/cuda/index.html](http://docs.nvidia.com/cuda/index.html)) чи виконати по етапне встановлення середовища Cuda для Windows наступним чином:

1. Завантаження і встановлення драйверу для відеокарти, що доступні за посиланням [http://www.Nvidia.com/drivers](http://www.nvidia.com/drivers). Потрібно звернути увагу, що версія драйвера повинна відповідати вимогам, зазначеним в примітках до релізу (Release Notes) Cuda Toolkit
2. Завантаження драйверів для розробників (Cuda Developer Drivers), які знаходяться в розділі Cuda Toolkit: [https://developer.Nvidia.com/cuda-toolkit](https://developer.nvidia.com/cuda-toolkit). Цей програмний пакет містить інструменти, бібліотеки, заголовні файли для компіляції програм за допомогою MS Visual Studio. GPU Computing SDK містить в собі демонстраційні приклади, які вже попередньо конфігуровано для зручної роботи в середовищі MS Visual Studio, і є виключно корисним при вивченні технології, але не потрібні для створення і запуску додатків.
3. Встановлення Cuda Toolkit. Після запуску інсталятора пакет поетапно дотримуйтеся процесу установки. При установці Cuda Toolkit необхідно обов'язково поставити галочку навпроти Cuda Toolkit Visual Studio Integration (повинна бути встановлена за замовчуванням).

Інструментарій Cuda за замовчуванням встановлюється в каталог “%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v#”, де # – версія Cuda. Ця папка включає в себе наступні каталоги й файли: “Bin\” – виконувані файли компілятора і робочі бібліотеки; “Include\” – заголовки, необхідні для компіляції додатків Cuda; “Lib\” – необхідні бібліотеки для зв'язку додатків Cuda. Cuda Documentation за замовчуванням встановлюється у каталог інструментарію Cuda в каталог “doc/”.

GPU Computing SDK за замовчуванням встановлюється у каталог “%ProgramData%\NVIDIA Corporation\CUDA Samples\v#”, де # – версія Cuda. Як вже було сказано, даний програмний комплект містить не тільки вихідні коди програм для безлічі прикладів і завдань, а й шаблони для Microsoft Visual Studio

Також під час установки створюються змінні оточення, які можуть знадобитися для налаштування, а саме змінні CUDA\_PATH, CUDA\_BIN\_PATH, CUDA\_LIB\_PATH, CUDA\_INC\_PATH.

1. Перевірка робото здатності середовища Cuda. Для цього необхідно провести запуск одного з демонстраційних прикладів які знаходяться у відповідному каталозі Cuda Toolkit (“%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v#\extras\demo\_suite“, де # – версія Cuda). У випаду, якщо запуск програми не відбувається чи призводить до помилки. То виникли помилки під час розгорнення Cuda чи обладнання робочого комп’ютера не підтримує відповідну версію технології Cuda.
2. Якщо ви використовуєте сучасні версії середовища Cuda і MS Visual Studio до меню створення проектів повинний додатися проект “NVIDIA CUDA # Runtime” (де # - ваша версія Cuda), якщо цього не відбилось о ви можете завантажити шаблон проекту з онлайн магазину шаблонів проектів, використати фали інтеграції з MS Visual Studio, чи створити проект на базі шаблону проекту C++ зробивши певні налаштування.

Для налаштування проекту проект на базі шаблону проекту C++ необхідно зробити наступні дії:

1. Налаштування роботи з файлами з розширенням “.cu”. Для того, щоб MS Visual Studio правильно опрацьовував файли з даним розоренням необхідно провести ряд маніпуляцій. По перше, розширення необхідно додати в розширення для включення в проекти С++ для цього відкриваємо меню опції (Tools → Options), після цього відкриваємо проекту VC++ (Projects and Solutions → VC++ Project Settings) в строчці “Extensions To Include” додаємо розширення “.cu” и “.cu.h” (без лапок). По друге, потрібно включити підсідку синтаксису і авто підстановку в редакторі коду для цього перейти меню опцій (Tools → Options) де перейти в розділ параметрів файлових розширень текстового редактору (Tools → Options → Text Editor → File Extension) і додати нове розширення “.cu” і обрати Microsoft VC++ в якості редактору коду.
2. Налаштування підсвічування зарезервованих слів синтаксису Cuda, для цього якщо присутній файл “usertype.dat” знаходить у каталозі чи в підкаталогі вашої версії VS Studio “%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v#\doc\syntaxhighlighting” (де # – версія Cuda) то скопіювати його в під каталог “Common7\IDE”, який знаходиться за розміщенням Visual Studio.
3. Для того щоб середовище знала, як потрібно компілювати “.cu” файли, необхідно використовувати спеціальні правила компіляції цих проектів. CUDA SDK містить такі файли, звані rules.

Необхідно скопіювати до під каталогу “\VC\VCProjectDefaults” MS Visual Studio зміст каталогу “%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v#\extras\visual\_studio\_integration\rules“ (де # – версія Cuda). Окрім цього, необхідно скопіювати до каталогу “%ProgramFiles%\MSBuild\ Microsoft.Cpp\v4.0\BuildCustomizations” зміст каталогу “%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v#\extras\visual\_studio\_integration\MSBuildExtensions“ (де # – версія Cuda).

1. В параметрах MS Visual Studio необхідно вказати каталоги файлів заголовків та фалів бібліотек використовуючи відповідні зміні оточення. Налаштування шляхів здійснюється в відповідному розділі налаштувань (Tools → Options → Projects and Solutions → VC++ Directories). Також необхідно додати бібліотеку cudart.lib до додаткових залежностей проекту (Project → Properties → Linker → Input → Additional Dependencies).

Після виконання цих дій для перевірки правильності налаштувань необхідно провести компіляцію та запуск одного з включених до GPU Computing SDK прикладів. У випадку якщо компіляція не вдалась перевірити зроблені налаштування.

В каталозі прикладів знаходяться файли рішень для різних версій MS Visual Studio. Необхідно обрати рішення відповідне до вашої версії Visual Studio і запустити його. Після цього буде здійснена міграція проектів даного рішення до вашої версії Visual Studio (Ця операція незворотна). Тому бажано скопіювати вміст папки і працювати з копією для запобігання пошкодження прикладів. В якості проекту для перевірки можна обрати проект за замовчанням “acyncAPI”, що ілюструє виконання подій в Cuda.

Після виконання встановлення середовища Cuda і його налаштування ви зможете створювати пусті проекти Cuda.

### Створення перших проектів на базі технології Cuda

Створимо перший проект Cuda для цього створюємо пустий проект у MS Visual Studio і компілюємо програму “Cuda: Hello, world!”. Вихідний код даної наведена на рис 6.1.

#include "cuda\_runtime.h"

#include "stdio.h"

\_\_global\_\_ void kernel() {

printf("Hello, world!\n");

}

int main(int agrc, char\* argv[]) {

printf("Now cuda say:\n");

kernel<<<2, 2>>>();

cudaDeviceSynchronize();

return 0;

}

Рисунок 6.1 – Вихідний код програми “Cuda: Hello, world!”

Після успішного створення першої програми напишемо програму, яка виводить параметри середовища. Відповідний вихідний код наведена на рис 6.2.

#include "cuda\_runtime.h"

#include "stdio.h"

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

int deviceCount;

cudaDeviceProp deviceProp;

cudaGetDeviceCount(&deviceCount);

printf("Device count: %d\n\n", deviceCount);

for (int i = 0; i < deviceCount; i++) {

cudaGetDeviceProperties(&deviceProp, i);

printf("Device %d name: %s\n", i + 1, deviceProp.name);

printf("Total global memory: %zu\n", deviceProp.totalGlobalMem);

printf("Shared memory per block: %zu\n", deviceProp.sharedMemPerBlock);

printf("Registers per block: %d\n", deviceProp.regsPerBlock);

printf("Warp size: %d\n", deviceProp.warpSize);

printf("Memory pitch: %zu\n", deviceProp.memPitch);

printf("Max threads per block: %d\n", deviceProp.maxThreadsPerBlock);

printf("Max threads dimensions: x = %d, y = %d, z = %d\n", deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);

printf("Max grid size: x = %d, y = %d, z = %d\n", deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);

printf("Clock rate: %d\n", deviceProp.clockRate);

printf("Total constant memory: %zu\n", deviceProp.totalConstMem);

printf("Compute capability: %d.%d\n", deviceProp.major, deviceProp.minor);

printf("Texture alignment: %zu\n", deviceProp.textureAlignment);

printf("Device overlap: %d\n", deviceProp.deviceOverlap);

printf("Multiprocessor count: %d\n", deviceProp.multiProcessorCount);

printf("Kernel execution timeout enabled: %s\n", deviceProp.kernelExecTimeoutEnabled ? "true" : "false");

}

return 0;

}

Рисунок 6.2 – Вихідний код програми для виведення   
параметрів середовища Cuda

## Завдання до лабораторної роботи

1. Встановити і налаштувати середовища Cuda на робочій машині.
2. Модифікувати код програми “Cuda: Hello, world!” для виведення ядром в файл фрази “I am from N block, M thread (global index: K)”, де параметри N, M, K мають відповідні значення.
3. Створити і скомпілювати проект, який виводить параметри середовища Cuda, пояснити значення відповідних параметрів.
4. Ознайомитися і описати приклади наведені у GPU Computing SDK.

## Посилання на джерела

1. Wikipedia. Обчислення загального призначення на графічних процесорах [Електронний ресурс] – Режим доступу: [https://uk.wikipedia.org/wiki/ Обчислення\_загального\_призначення\_на\_графічних\_процесорах](https://uk.wikipedia.org/wiki/Обчислення_загального_призначення_на_графічних_процесорах)
2. Nvidia CUDA. Неграфические вычисления на графических процессорах [Електронний ресурс] – Режим доступу: <https://www.ixbt.com/video3/cuda-1.shtml>
3. Базовый видеокурс по NVIDIA Cuda и OpenACC (Е. Перепёлкин, 2014 г.) [Електронний ресурс] – Режим доступу: [https://www.youtube.com/watch ?v=XXfxHIpSlTw&list=PLhlTilzRdxykC7dmxh0Xp2kJNy0LSTUfO](https://www.youtube.com/watch?v=XXfxHIpSlTw&list=PLhlTilzRdxykC7dmxh0Xp2kJNy0LSTUfO)
4. CUDA Basics. Supercomputing Tutorial [Електронний ресурс] – Режим доступу: [http://www.Nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf](http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf)

# Лабораторна робота №2. Знайомство з ключовими моментами роботи з GPGPU на базі технології Nvidia Cuda

Мета роботи: знайомство з додатковими розширеннями для мови C необхідними для написання коду для GPU, основами Cuda host API і створення програми на базу Cuda.

## Теоретичні відомості

### Модель програмування з використанням технології Cuda

Cuda використовує паралельну модель обчислень, коли кожен з SIMD процесорів виконує ту ж інструкцію над різними елементами даних паралельно. GPU є обчислювальним пристроєм, співпроцесором (device) для центрального процесора (host), що володіє власною пам'яттю і обробляють паралельно велику кількість потоків. Ядром (kernel) називається функція для GPU, що виконується потоками.

Модель програмування в Cuda передбачає групування потоків. Потоки об'єднуються в блоки потоків (thread block) - одномірні або двовимірні сітки потоків, що взаємодіють між собою за допомогою розділяється пам'яті і точок синхронізації. Програма (ядро, kernel) виповнюється над сіткою (grid) блоків потоків (thread blocks). Одночасно виконується одна сітка. Кожен блок може бути одно-, дво- або тривимірним за формою. Графічне представлення цього процесу наведено на рис 1.1.

Блоки потоків виконуються у вигляді невеликих груп, що знаваються варпами (warp).

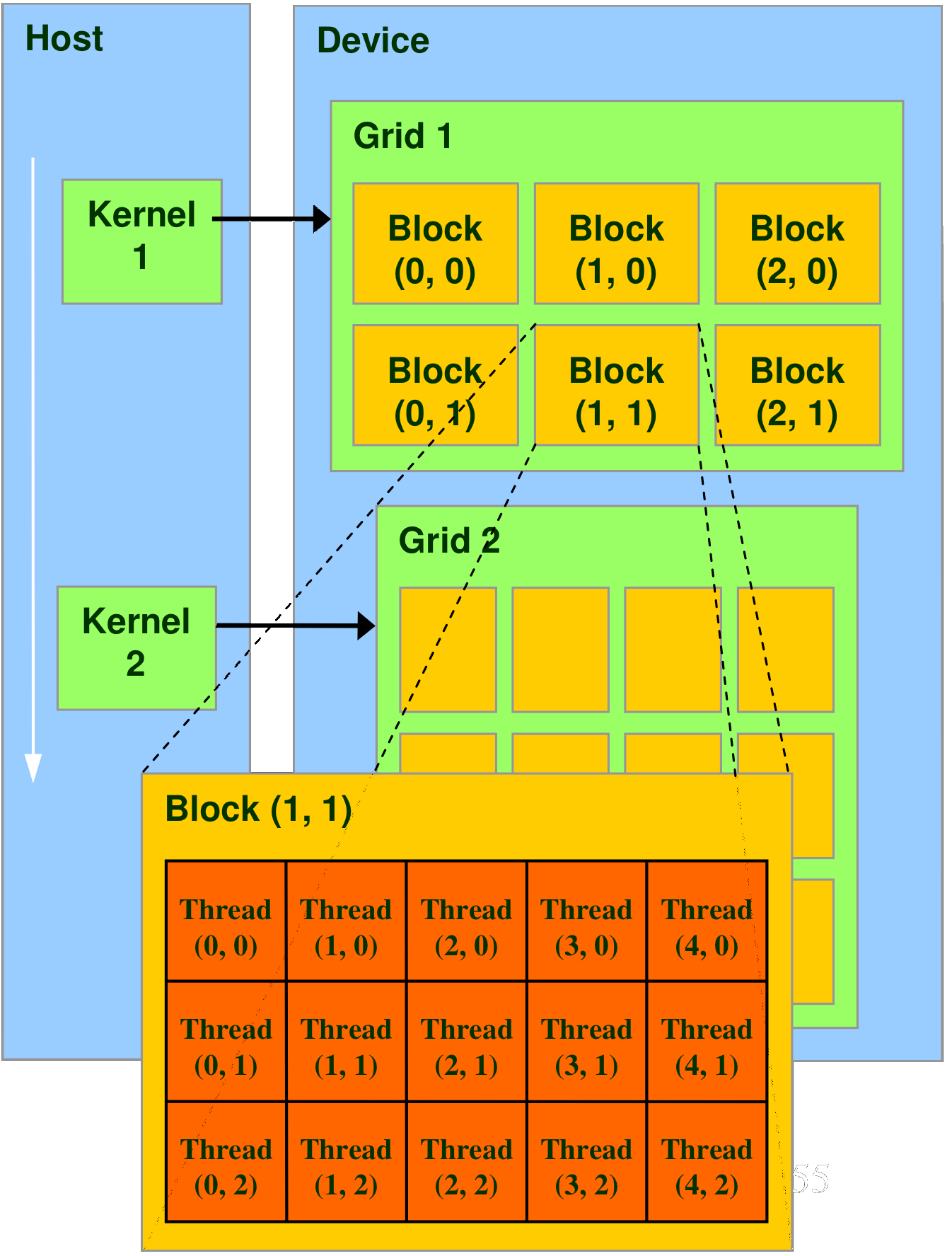


Рисунок 1.1 – Структура виконання обчислень на пристрої GPU

Угруповання блоків в сітки дозволяє піти від обмежень і застосувати ядро до більшої кількості потоків за один виклик. Це допомагає і при масштабуванні. Якщо у GPU недостатньо ресурсів, він буде виконувати блоки послідовно. У зворотному випадку, блоки можуть виконуватися паралельно, що важливо для оптимального розподілу роботи на відеочіпах різного рівня, починаючи від мобільних і інтегрованих.

Якщо казати більш предметно, то на логічному рівні все виглядає наступним чином. Верхній рівень ядра GPU складається з блоків, які групуються в сітку або грід (grid) розмірністю N1 \* N2 \* N3, причому якщо N3 дорівнює одиниці, то сітка двовимірна. Це зображено на рис 1.2.

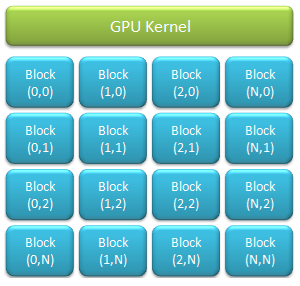


Рисунок 1.2 – Обчислювальний пристрій GPU

Розмірність сітки блоків можна дізнатися з допомогу функції cudaGetDeviceProperties, в отриманій структурі за це відповідає поле maxGridSize.

Будь-блок в свою чергу складається з ниток (threads), які є безпосередніми виконавцями обчислень. Нитки в блоці сформовані у вигляді тривимірного масиву (рис. 1.3), розмірність якого так само можна дізнатися за допомогою функції cudaGetDeviceProperties, за це відповідає поле maxThreadsDim.

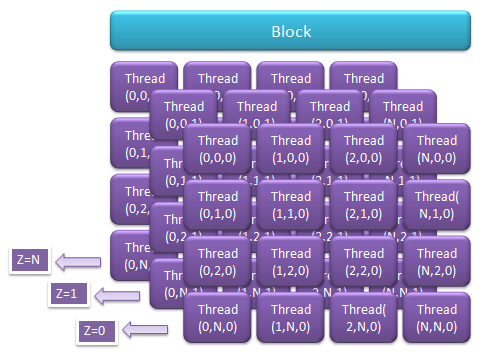


Рисунок 1.3 – Устрій блоку GPU

При використанні GPU можливо задіяти грід необхідного розміру і конфігурувати блоки під потреби певного завдання.

### Розширення мови C для Cuda

Сама технологія Cuda (компілятор nvcc.exe) вводить ряд додаткових розширень для мови C, які необхідні для написання коду для GPU:

1. Специфікатори функцій. Специфікатори функцій показують, як і звідки буду виконуватися функції. Всього в Cuda 3 специфікатори: \_\_host\_\_ – виконуватися на CPU, викликається з CPU (в принципі його можна і не вказувати); \_\_global\_\_ – Виконується на GPU, викликається з CPU; \_\_device\_\_ – Виконується на GPU, викликається з GPU.
2. Специфікатори запуску ядра GPU. Специфікатори запуску ядра служать для опису кількості блоків, ниток і пам'яті, які ви хочете виділити при розрахунку на GPU. Синтаксис запуску ядра має наступний вигляд:

kernel<<<gridSize, blockSize, sharedMemSize, cudaStream>>>(…); , де kernel – представляє функція ядра, що має специфікатор \_\_global\_\_; gridSize – розмірність сітки блоків (dim3), виділеної для розрахунків; blockSize – розмір блоку (dim3), виділеного для розрахунків; sharedMemSize – розмір додаткової пам'яті, що виділяється при запуску ядра; cudaStream – змінна cudaStream\_t, що задає потік, в якому буде проведений виклик.

1. Вбудовані змінні. Вбудовані змінні слугують для ідентифікації ниток, блоків і ін. параметрів при виконанні коду в ядрі GPU. До вбудованих змінних відносять: gridDim – розмірність гріда, виділеного при поточному виклику ядра (має тип dim3); blockDim – розмірність блоку, виділеного при поточному виклик ядра (має тип dim3); blockIdx – індекс поточного блоку в обчисленні на GPU (має тип uint3); threadIdx – індекс поточної нитки в обчисленні на GPU (має тип uint3); warpSize – розмір варпу (має тип int).
2. Специфікатори змінних. Специфікатори змінних служать для вказівки типу використовуваної пам'яті GPU. Детально будуть розглянути в наступній лабораторній роботі.
3. Додаткові типи змінних і їх специфікатори. Також детально будуть розглянуті в наступній лабораторній роботі.
4. Додані функції. Cuda підтримує всі математичні функції зі стандартної бібліотеки C. Однак при цьому слід мати на увазі, що більшість стандартних математичних функцій використовує число в подвійною точністю (double). Але оскільки для GPU операції з double-числами виконуються повільніше, ніж операції з float-числами, то краще там, де це можливо, використовувати float-аналоги стандартних функцій.

### Основи Cuda host API

Перед тим, як приступити до безпосереднього використання Cuda для обчислень, необхідно ознайомитися з так званим Cuda host API, який є сполучною ланкою між CPU і GPU. Cuda host API надає в розпорядження програміста ряд функцій, які можуть бути використані тільки CPU.

Cuda host API в свою чергу можна розділити на низко рівневе API під назвою Cuda driver API, яке надає доступ до драйвера призначеного для користувача режиму CUDA, і високо рівневе API – Cuda runtime API. Для написання програм на початковому етапі ці API можна вважати взаємовиключними, тобто при написанні програм можливо працювати тільки з одним з них. Для наведення прикладів використовується Cuda runtime API.

Ці функції відповідають за:

* Device Management - включає функції для загального управління GPU (отримання інформації про можливості GPU, перемикання між GPU при роботі SLI-режимі і т.д.);
* Thread Management - управління нитками;
* Stream Management - управління потоками;
* Event Management - функція створення і управління подіями;
* Execution Control - функції запуску і виконання ядра CUDA;
* Memory Management - функції управління пам'яттю GPU;
* Texture Reference Manager - робота з об'єктами текстур через CUDA;
* OpenGL Interoperability - функції по взаємодії з OpenGL API;
* Direct3D Interoperability - функції по взаємодії з Direct3D API;
* Error Handling - функції обробки помилок.

Cuda driver API представляє собою низько рівневий API, що дає більше можливостей програмісту, а й вимагає більшого обсягу коду і необхідність явних налаштувань, вимога явною ініціалізації і відсутність підтримки режиму емуляції. Відсутність приводить до не можливості компілювати, запускати і налагоджувати коди на Cuda з CPU.

Cuda runtime API – це високо рівневий API, до того ж Cuda runtime API не вимагає явної ініціалізації – вона відбувається автоматично при першому виклику якої-небудь функції. Даний API підтримує емуляцію. Плюсом є можливість використання додаткових бібліотек (CUFFT, CUBLAS, CUDPP).

Слід так само відзначити, що у кожної функції Cuda runtime API є прямий аналог в Cuda driver API, тобто перехід з Cuda runtime API на Cuda driver API не дуже складний, зворотне в загальному випадку не вірно.

### Приклад практичного використання

Як було сказано, нитка – безпосередній виконавець обчислень. Розглянемо роботу окремо взятого блоку, щоб на прикладі зрозуміти яким чином відбувається розпаралелювання обчислень.

Постановка завдання: Обчислити суму двох векторів розмірністю в N елементів.

Нам відома максимальні розміри нашого блоку: 512 \* 512 \* 64 ниток. Так як вектор у нас одновимірний, то поки обмежимося використанням x-виміру нашого блоку, тобто задіємо тільки одну смугу ниток з блоку. Зауважимо, що x-розмірність блоку 512, тобто, ми можемо скласти за один раз вектори, довжина яких не перевищує 512 елементи ().



Рисунок 4.1 – Смуга ниток з використовуваного блоку.

У самій програмі необхідно виконати наступні етапи:

1. Отримати дані для розрахунків.
2. Скопіювати ці дані в GPU пам'ять.
3. Провести обчислення в GPU через функцію ядра.
4. Скопіювати обчислені дані з GPU пам'яті в ОЗУ.
5. Подивитися результати. Вивільнити використовувані ресурси.

Переходимо безпосередньо до написання коду. Насамперед напишемо функцію ядра, яка і буде здійснювати складання векторів (рис 4.2).

// Функція складання двох векторів

\_\_global\_\_ void addVector(float\* left, float\* right, float\* result) {

// Отримати id поточної нитки.

int idx = threadIdx.x;

// Розраховуємо результат.

result[idx] = left[idx] + right[idx];

}

Рисунок 4.2 – Функція складання двох векторів

Таким чином, розпаралелювання буде виконано автоматично при запуску ядра. У цій функції так само використовується вбудована змінна threadIdx і її поле x, яка дозволяє задати відповідність між розрахунком елемента вектора і ниткою в блоці. Робимо розрахунок кожного елемента вектора в окремій нитки.

Код, що відповідає за отримання даних для розрахунків і копіювання даних до GPU наведено на рис 4.3.

#define SIZE 512

\_\_host\_\_ int main() {

// Виділяємо пам'ять під вектора

float \*vec1 = new float[SIZE], \*vec2 = new float[SIZE], \*vec3 = new float[SIZE];

// Ініціалізіруем значення векторів

for (int i = 0; i < SIZE; i++) vec1[i] = vec2[i] = i;

// Покажчики на пам'ять відеокарти

float \*devVec1, \*devVec2, \*devVec3;

// Виділяємо пам'ять для векторів на відеокарті

cudaMalloc((void\*\*)&devVec1, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec2, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec3, sizeof(float) \* SIZE);

// Копіюємо дані в пам'ять відеокарти

cudaMemcpy(devVec1, vec1, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

cudaMemcpy(devVec2, vec2, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

…

}

Рисунок 4.3 – Отримання даних розрахунків і копіювання їх до GPU

Виділення пам'яті на відеокарті використовується за допомогою функції, яка має наступний прототип:

cudaError\_t cudaMalloc (void \*\* devPtr, size\_t count) , де

* devPtr – покажчик, в який записується адреса виділеної пам'яті,
* count – розмір виділеної пам'яті в байтах.

Функція може повертати наступні значення:

* cudaSuccess – при вдалому виділення пам'яті,
* cudaErrorMemoryAllocation – при помилці виділення пам'яті.

Копіювання даних в пам'ять відеокарти за допомогою функції, яка має наступний прототип:

cudaError\_t cudaMemcpy(void\* dst, const void\* src ,size\_t count, enum cudaMemcpyKind kind) , де

* dst – покажчик, що містить адресу місця-призначення копіювання,
* src – покажчик, що містить адресу джерела копіювання,
* count – розмір копійованого ресурсу в байтах,
* cudaMemcpyKind – перерахування, яке вказує напрямок копіювання (може бути cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice).

Функція може повертати наступні значення:

* cudaSuccess – при вдалому копіюванні,
* cudaErrorInvalidValue – якщо неправильні аргументу (наприклад, розмір копіювання негативний),
* cudaErrorInvalidDevicePointer – невірний покажчик пам'яті в відеокарті,
* cudaErrorInvalidMemcpyDirection – неправильний напрямок (наприклад, переплутано джерело і місце-призначення копіювання)

Тепер переходимо до безпосереднього виклику ядра для обчислення на GPU, його код наведено на рис. 4.4. Оскільки у нашому випадку визначати розмір гріда і блоку необов'язково, так як використовуємо всього один блок і один вимір в блоці, тому код на рис. 4.4 можна записати іншим чином (рис 4.5).

…

// Розмір гріду, що використовується

dim3 gridSize = dim3(1, 1, 1);

// Розмір блоку, що використовується

dim3 blockSize = dim3(SIZE, 1, 1);

// Виконуємо виклик функції ядра

addVector<<<gridSize, blockSize>>>(devVec1, devVec2, devVec3);

…

Рисунок 4.4 – Виклик ядра обчислень GPU

…

addVector<<<1, SIZE>>>(devVec1, devVec2, devVec3);

…

Рисунок 4.5 – Виклик ядра обчислень GPU

Тепер нам залишається скопіювати результат розрахунку з відеопам'яті в пам'ять хоста. Але у функцій ядра при цьому є особливість - асинхронне виконання, тобто, якщо після виклику ядра почав працювати наступну ділянку коду, то це ще не означає, що GPU виконав розрахунки. Для завершення роботи заданої функції ядра необхідно використовувати засоби синхронізації, наприклад події. Тому, перед копіюванням результатів на хост виконуємо синхронізацію ниток GPU через події.

…

// Виконуємо виклик функції ядра

addVector<<<gridSize, blockSize>>>(devVec1, devVec2, devVec3);

// Обробник події

cudaEvent\_t syncEvent;

// Створюємо подію

cudaEventCreate(&syncEvent);

// Записуємо подію

cudaEventRecord(syncEvent, 0);

// Синхронізуємо подію

cudaEventSynchronize(syncEvent);

// Тільки тепер отримуємо результат розрахунку

cudaMemcpy(vec3, devVec3, sizeof(float) \* SIZE, cudaMemcpyDeviceToHost);

…

Рисунок 4.6 – Отримання результатів обчислень з GPU

Розглянемо більш докладно функції з Event Managment API.

Події створюється за допомогою функції, яка має наступний прототип:

cudaError\_t cudaEventCreate(cudaEvent\_t\* event) , де

* event – покажчик для запису обробника події.

Функція може повертати наступні значення:

* cudaSuccess – в разі успіху,
* cudaErrorInitializationError – помилка ініціалізації,
* cudaErrorPriorLaunchFailure – помилка при попередньому асинхронному запуску функції,
* cudaErrorInvalidValue – невірне значення,
* cudaErrorMemoryAllocation – помилка виділення пам'яті.

Запис обробника події здійснюється за допомогою функції, яка має наступний прототип:

cudaError\_t cudaEventRecord(cudaEvent\_t event, CUstream stream) , де

* event – покажчик обробнику події,
* stream – номер потоку, в якому записуємо (в нашому випадку це основний нульовий потік).

Функція може повертати наступні значення:

* cudaSuccess – в разі успіху,
* cudaErrorInvalidValue – невірне значення,
* cudaErrorPriorLaunchFailure – помилка при попередньому асинхронному запуску функції,
* cudaErrorInvalidResourceHandle - невірний обробник події.

Синхронізація події виконується функцією cudaEventSynchronize. Ця функція чекає закінчення роботи всіх ниток GPU і проходження заданої події і тільки потім віддає управління викликає програмі. Прототип функції має вигляд:

cudaError\_t cudaEventSynchronize(cudaEvent\_t event) , де

* event – покажчик обробнику події, проходження якого очкується.

Функція може повертати наступні значення:

* cudaSuccess – в разі успіху,
* cudaErrorInitializationError – помилка ініціалізації,
* cudaErrorPriorLaunchFailure – помилка при попередньому асинхронному запуску функції,
* cudaErrorInvalidValue – невірне значення,
* cudaErrorInvalidResourceHandle – невірний обробник події.

Зрозуміти, як працює cudaEventSynchronize, можна з схеми на рис 4.7. У наведеній схемі блок “Очікування проходження події (синхронізація)” і є нашою функцією.



Рисунок 4.7 – Синхронізація роботи основної і GPU програм

Код виведення результату на екран і чистики виділених ресурсів зображено на рис 4.8.

…

// Результати розрахунку

for (int i = 0; i < SIZE; i++) printf("Element #%i: %.1f\n", i, vec3[i]);

// Вивільняються ресурсів

// Видалення події

cudaEventDestroy(syncEvent);

// Вивільнення пам'яті на відеокарті

cudaFree(devVec1);

cudaFree(devVec2);

cudaFree(devVec3);

// Вивільнення памьяті основної програми

delete[] vec1, vec2, vec3;

…

Рисунок 4.8 – Виведення результату на екран та очистка ресурсів

Події вивільнюся за допомогою функції, яка має наступний прототип:

cudaError\_t cudaEventCreate(cudaEvent\_t\* event) , де

* event – покажчик для вивільнення обробнику події.

Функція може повертати наступні значення:

* cudaSuccess – в разі успіху,
* cudaErrorPriorLaunchFailure – помилка при попередньому асинхронному запуску функції,
* cudaErrorInvalidValue – невірне значення,
* cudaErrorInvalidResourceHandle – невірний обробник події.

При створені програми були описані найголовніші моменти, які необхідно знати для роботи з Cuda.

### Розрахунок часу виконання

У разі якщо розрахунки виконуються тільки на CPU, то для виміру часу розрахунків використовується функція GetTickCount(), яка підключається з windows.h. Для виміру часу розрахунків на GPU використовується функція, яка має наступний прототип:

cudaError\_t cudaEventElapsedTime(float \*time, cudaEvent\_t start, cudaEvent\_t end) , де

Функція може повертати наступні значення:

* cudaSuccess – в разі успіху,
* cudaErrorInvalidValue – невірне значення,
* cudaErrorInitializationError – помилка ініціалізації,
* cudaErrorPriorLaunchFailure – помилка при попередньому асинхронному запуску функції,
* cudaErrorInvalidResourceHandle – невірний обробник події.

## Завдання до лабораторної роботи

1. Реалізувати програму, що описана в пункті 4 теоретичної частини
2. Написати аналогічну програму, з наступними відмінностями:
   1. Передбачити можливості отримання векторів для обрахунку за допомогою випадкової генерації
   2. Кількість елементів в векторі задавати за допомогою командної строки
   3. Реалізувати підтримку векторів більшого розміру, ніж розмір блоку гріду
3. Модифікувати написану у другому пункті програму для порівняння часових характеристик в залежності від розміру векторів. Маються на увазі показники часу виконання сумування на CPU, чистого часу сумування на GPU та часу з передачею даних до GPU. Зробити висновки до отриманих результатів.
4. Реалізувати програму, що здійснює складення двох матриць до якою виставляються вимоги аналогічні зазначеним у другому пункті. А також виконати оцінку аналогічну описаній у третьому пункті.

## Посилання на джерела

1. Базовый видеокурс по NVIDIA Cuda и OpenACC (Е. Перепёлкин, 2014 г.) [Електронний ресурс] – Режим доступу: [https://www.youtube.com/watch ?v=XXfxHIpSlTw&list=PLhlTilzRdxykC7dmxh0Xp2kJNy0LSTUfO](https://www.youtube.com/watch?v=XXfxHIpSlTw&list=PLhlTilzRdxykC7dmxh0Xp2kJNy0LSTUfO)
2. CUDA Basics. Supercomputing Tutorial [Електронний ресурс] – Режим доступу: [http://www.Nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf](http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf)
3. А.В. Григорьев, И.С. Еремеев, М.И. Алексеева. Учебное пособие. Параллельное программирование с использованием технологии Cuda [Електронний ресурс] – Режим доступу: <http://edu.chpc.ru/cuda/main.html>

# Лабораторна робота №3. Знайомство з роботою з пам’яттю у GPGPU на базі технології Nvidia Cuda

Мета роботи: знайомство з програмним устроєм GPU, ключовими моментами компілятора CUDA, інтерфейсом Cuda runtime API, а також створення програм для нескладних математичних обчислень.

## Теоретичні відомості

### Відеокарта і типи пам'яті

При використанні GPU розробнику доступно декілька видів пам'яті: регістри, локальна, глобальна, колективна, константна і текстурна пам'яті. Кожна з цих типів пам'яті має певне призначення, яке обумовлюється її технічними параметрами (швидкість роботи, рівень доступу на читання і запис). Ієрархія типів пам'яті представлена на рис. 1.1.

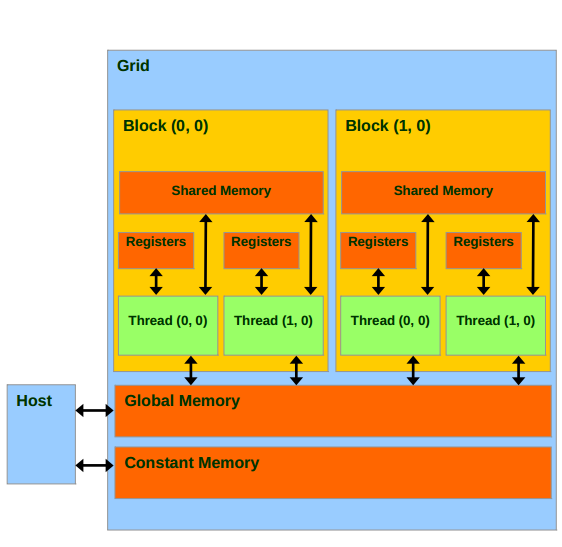


Рисунок 1.1 – Ієрархія типів пам’яті Cuda

Відповідно до знаходження пам’яті в ієрархії вона має свої характеристики та відмінності. Опишемо доступні типи пам’яті:

1. Реєстрова пам'ять (register memory) є найшвидшою з усіх видів. Визначити кількість регістрів доступних GPU можна за допомогою вже добре відомої функції cudaGetDeviceProperties. Розрахувати кількість регістрів, доступних однієї нитки GPU, так само не складає труднощів, для цього необхідно розділити загальне число регістрів на твір кількості ниток в блоці і кількості блоків в гріді. Всі регістри GPU 32 розрядні. У Cuda немає явних способів використання реєстрової пам'яті, всю роботу по розміщенню даних в регістрах бере на себе компілятор.
2. Локальна пам'ять (local memory) може бути використана компілятором при великій кількості локальних змінних в будь-якої функції. За швидкісними характеристиками локальна пам'ять значно повільніше, ніж реєстрова. У документації від Nvidia рекомендується використовувати локальну пам'ять тільки в найнеобхідніших випадках. Явних засобів, що дозволяють блокувати використання локальної пам'яті, не передбачено, тому при падінні продуктивності варто ретельно проаналізувати код і виключити зайві локальні змінні.
3. Колективна пам'ять (shared memory) ставитися до швидкого типу пам'яті. Пам'ять, що розділяється рекомендується використовувати для мінімізації звернення до глобальної пам'яті, а так само для зберігання локальних змінних функцій. Адресація пам'яті, що між нитками потоку однакова в межах одного блоку, що може бути використано для обміну даними між потоками в межах одного блоку. Для розміщення даних в пам'яті, що використовується специфікатор \_\_shared\_\_.
4. Константна пам'ять (constant memory) є досить швидкою з доступних GPU. Відмінною особливістю константної пам'яті є можливість запису даних з хоста, але при цьому в межах GPU можливо лише читання з цієї пам'яті, що й обумовлює її назву. Для розміщення даних в константної пам'яті передбачений специфікатор \_\_constant\_\_. Якщо необхідно використовувати масив в константної пам'яті, то його розмір необхідно вказати заздалегідь, так як динамічне виділення на відміну від глобальної пам'яті в константної не підтримується. Для запису з хоста в константну пам'ять використовується функція cudaMemcpyToSymbol, і для копіювання з пристрою (співпроцесора) на хост cudaMemcpyFromSymbol, як видно цей підхід дещо відрізняється від підходу при роботі з глобальною пам'яттю.
5. Текстурна пам'ять (texture memory), як і випливає з назви, призначена головним чином для роботи з текстурами. Текстурна пам'ять має специфічні особливості в адресації, читанні і запису даних. Більш докладно ознайомитися з текстурною пам'яттю можна при розгляді питань обробки зображень на GPU.

### Налагодження програмного коду Cuda

Існує невеликий спосіб налагодження програм. Як відомо, функції з Cuda runtime API можуть повертати різні коди помилок. Щоб спростити собі життя можна використовувати макрос для вилову помилок. Приклад такого макросу наведений на рис 2.1.

// Визначення змінної середовища

#define CUDA\_DEBUG

// Виведення діагностичної інформації

#ifdef CUDA\_DEBUG

#define CUDA\_CHECK\_ERROR(err) \

if (err != cudaSuccess) { \

printf("Cuda error: %s\n", cudaGetErrorString(err)); \

printf("Error in file: %s, line: %i\n", \_\_FILE\_\_, \_\_LINE\_\_); \

}

#else

#define CUDA\_CHECK\_ERROR(err)

#endif

Рисунок 2.1 – Макрос відлову помилок Cuda

Як видно, в разі, якщо визначена змінна середовища CUDA\_DEBUG, відбувається перевірка коду помилки і виводитися інформація про файл і рядок, де сталася помилка. Цю зміну можна включити при компіляції під час налагодження і відключити при компіляції під реліз. Також можна внести певні зміни до відповідних параметрів компіляції, а саме вказати її в заголовок препроцесора який використовується при компілюванні під час налагоджені .

### Практичне використання розділеної пам’яті

Будемо розглядати роботу пам’яті на прикладі алгоритму транспонування матриці. Для того щоб побачити вплив використання пам'яті на швидкість обчислень слід написати функцію, яка буде використовувати тільки глобальну пам'ять. Приклад такої функції наведено на рис 3.1.

// Функція транспонування матриці без використання глобальної пам'яті

// \* inputMatrix - покажчик на вихідну матрицю

// \* outputMatrix - покажчик на матрицю результат

// \* width - ширина вихідної матриці (вона ж висота матриці-результату)

// \* height - висота вихідної матриці (вона ж ширина матриці-результату)

\_\_global\_\_ void transposeMatrixGlobal(float\* inputMatrix, float\* outputMatrix, int width, int height) {

// Розрахунок індексів матриці

int xIndex = blockDim.x \* blockIdx.x + threadIdx.x;

int yIndex = blockDim.y \* blockIdx.y + threadIdx.y;

if ((xIndex < width) && (yIndex < height)) {

// Лінійний індекс елемента рядки вихідної матриці

int inputIdx = xIndex + width \* yIndex;

// Лінійний індекс елемента стовпця матриці-результату

int outputIdx = yIndex + height \* xIndex;

// Встановлення елементу

outputMatrix[outputIdx] = inputMatrix[inputIdx];

}

}

Рисунок 3.1 – Функція транспонування без використання колективної пам’яті

Ця функція просто копіює рядки вихідної матриці в стовпці матриці-результату. Єдиний складний момент - це визначення індексів елементів матриць, тут необхідно пам'ятати, що при виклику ядра може бути використані різні розмірності блоків і гріда, для цього і використовуються вбудовані змінні blockDim, blockIdx.

Тепер напишемо функцію транспонування, яка використовує колективну пам'ять. Приклад такої функції наведено на рис 3.2.

#define BLOCK\_DIM 16

// Функція транспонування матриці з використанням колективної пам'яті

// \* inputMatrix - покажчик на вихідну матрицю

// \* outputMatrix - покажчик на матрицю результат

// \* width - ширина вихідної матриці (вона ж висота матриці-результату)

// \* height - висота вихідної матриці (вона ж ширина матриці-результату)

\_\_global\_\_ void transposeMatrixShared(float\* inputMatrix, float\* outputMatrix, int width, int height) {

\_\_shared\_\_ float temp[BLOCK\_DIM][BLOCK\_DIM];

// Розрахунок індексів матриці

int xIndex = blockIdx.x \* blockDim.x + threadIdx.x;

int yIndex = blockIdx.y \* blockDim.y + threadIdx.y;

if ((xIndex < width) && (yIndex < height)) {

// Линейный индекс элемента строки исходной матрицы

int idx = yIndex \* width + xIndex;

//Копируем элементы исходной матрицы

temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];

}

//Синхронизируем все нити в блоке

\_\_syncthreads();

xIndex = blockIdx.y \* blockDim.y + threadIdx.x;

yIndex = blockIdx.x \* blockDim.x + threadIdx.y;

if ((xIndex < height) && (yIndex < width)) {

// Линейный индекс элемента строки исходной матрицы

int idx = yIndex \* height + xIndex;

//Копируем элементы исходной матрицы

outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];

}

}

Рисунок 3.2 – Функція транспонування з використанням колективної пам’яті

У цій функції я використовую пам'ять, що розділяється в вигляді двовимірного масиву.

Як вже було сказано, адресація пам'яті, що в межах одного блоку однакова для всіх потоків, тому, щоб уникнути колізій при доступі і записи, кожному елементу в масиві відповідає одна нитка в блоці.

Після копіювання елементів вихідної матриці в буфер temp, викликається функція \_\_syncthreads. Ця функція синхронізує потоки в межах блоку. Її відмінність від інших способів синхронізації заключається в тому, що вона виполняеться тільки на GPU.

В кінці відбувається копіювання збережених елементів вихідної матриці в матрицю-результат, відповідно до правила транспонування.

Може здатися, що ця функція повинна виконуватися повільніше, ніж її версія без пам'яті, що, де немає ніяких посередників. Але насправді копіювання з глобальної пам'яті в глобальну працює значно повільніше, ніж зв'язка глобальна пам'ять - колективна пам'ять - глобальна пам'ять.

Хочу зауважити, що перевіряти межі масивів матриць варто вручну, в GPU немає апаратних засобів для стеження за межами масивів.

Тепер напишемо функцію транспонування, яка виповнюється тільки на CPU. Приклад такої функції наведено на рис 3.3.

// Функція транспонування матриці, яка виконується на CPU

\_\_host\_\_ void transposeMatrixCPU(float \*inputMatrix, float \*outputMatrix, int width, int height) {

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

outputMatrix[x \* height + y] = inputMatrix[y \* width + x];

}

}

}

Рисунок 3.3 – Функція транспонування, що виконується на CPU

Для збереження отриманих результатів будемо записувати вихідну матрицю і результат в файли за допомогою функції printMatrixToFile. Якщо матриці дуже великі, то висновок даних в файли може сильно уповільнити виконання програми. Приклад вихідного коду даної функції наведено на рис 3.4.

// Функція виведення матриці на екран

\_\_host\_\_ void printMatrixToFile(char\* fileName, float\* matrix, int width, int height) {

FILE \*file = fopen(fileName, "wt");

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

fprintf(file, "%.0f\t", matrix[y \* width + x]);

}

fprintf(file, "\n");

}

fclose(file);

}

Рисунок 3.4 – Функція виведення матриці на екран

Тепер необхідно згенеруємо дані для розрахунків, скопіювати їх з хоста на девайс, в разі використання GPU, зробити виміри продуктивності і очистити ресурси. Так як ці етапи приблизно такі ж, що й описані в попередній роботі, то не будемо пояснювати код функції: Приклад даної функції зображено на рис 3.5.

#define GPU\_SLOW 1

#define GPU\_FAST 2

#define CPU 3

// Кількість навантажувальних циклів

#define ITERATIONS 20

\_\_host\_\_ int main() {

// Ширина і висота матриці

int width = 2048, height = 1536;

// Розмір масиву для збереження матриці

int matrixSize = width \* height;

// Кількість байтів що займає матриця

int byteSize = matrixSize \* sizeof(float);

//Выделяем память под матрицы на хосте

float\* inputMatrix = new float[matrixSize];

float\* outputMatrix = new float[matrixSize];

//Заполняем исходную матрицу данными

for (int i = 0; i < matrixSize; i++)

inputMatrix[i] = i;

// Вибираємо спосіб розрахунку транспонованою матриці

printf("Select compute mode: 1 - Slow GPU, 2 - Fast GPU, 3 - CPU\n");

int mode;

scanf("%i", &mode);

// Записуємо вихідну матрицю в файл

printMatrixToFile("before.txt", inputMatrix, width, height);

// Якщо используеться тільки CPU

if (mode == CPU) {

int start = GetTickCount();

for (int i = 0; i < ITERATIONS; i++) {

transposeMatrixCPU(inputMatrix, outputMatrix, width, height);

}

// Виводимо час виконання функції на CPU (в мілліекундах)

printf("CPU compute time: %i\n", GetTickCount() - start);

}

// У разі розрахунку на GPU

else {

float \*devInputMatrix, \*devOutputMatrix;

// Виділяємо глобальну пам'ять для зберігання даних на пристрої

CUDA\_CHECK\_ERROR(cudaMalloc((void\*\*)&devInputMatrix, byteSize));

CUDA\_CHECK\_ERROR(cudaMalloc((void\*\*)&devOutputMatrix, byteSize));

// Копіюємо вихідну матрицю з хоста на девайс

CUDA\_CHECK\_ERROR(cudaMemcpy(devInputMatrix, inputMatrix, byteSize, cudaMemcpyHostToDevice));

// Конфігурація запуску ядра

dim3 gridSize = dim3(width / BLOCK\_DIM, height / BLOCK\_DIM, 1);

dim3 blockSize = dim3(BLOCK\_DIM, BLOCK\_DIM, 1);

cudaEvent\_t start, stop;

// Створюємо події для синхронізації і виміру часу роботи GPU

CUDA\_CHECK\_ERROR(cudaEventCreate(&start));

CUDA\_CHECK\_ERROR(cudaEventCreate(&stop));

//Отмечаем старт расчетов на GPU

cudaEventRecord(start, 0);

// Використовується функція без суспільної пам'яті

if (mode == GPU\_SLOW) {

for (int i = 0; i < ITERATIONS; i++) {

transposeMatrixGlobal<<<gridSize, blockSize>>>(devInputMatrix, devOutputMatrix, width, height);

}

}

// Використовується функція з суспільною пам'яттю

else if (mode == GPU\_FAST) {

for (int i = 0; i < ITERATIONS; i++) {

transposeMatrixShared<<<gridSize, blockSize>>>(devInputMatrix, devOutputMatrix, width, height);

}

}

// Відзначаємо закінчення розрахунку

cudaEventRecord(stop, 0);

// Синхронізуються з моментом закінчення розрахунків

cudaEventSynchronize(stop);

// Розраховуємо час роботи GPU

float time = 0;

cudaEventElapsedTime(&time, start, stop);

// Виводимо час розрахунку в консоль

printf("GPU compute time: %.0f\n", time);

// Копіюємо результат з девайса на хост

CUDA\_CHECK\_ERROR(cudaMemcpy(outputMatrix, devOutputMatrix, byteSize, cudaMemcpyDeviceToHost));

// Чистимо ресурси на відеокарті

CUDA\_CHECK\_ERROR(cudaFree(devInputMatrix));

CUDA\_CHECK\_ERROR(cudaFree(devOutputMatrix));

CUDA\_CHECK\_ERROR(cudaEventDestroy(start));

CUDA\_CHECK\_ERROR(cudaEventDestroy(stop));

}

// Записуємо матрицю-результат в файл

printMatrixToFile("after.txt", outputMatrix, height, width);

// Чистимо пам'ять на хості

delete[] inputMatrix, outputMatrix;

return 0;

}

Рисунок 3.5 – Головна функція програми

## Завдання до лабораторної роботи

1. Реалізувати програму, що описана в пункті 3 теоретичної частини. І виконати оцінки швидкості виконання обчислень при різних даних.
2. Зробити висновки про типи пам’яті, навести приклади та сценарії використання типів пам’яті CUDA.

## Посилання на джерела

1. А.В. Григорьев, И.С. Еремеев, М.И. Алексеева. Учебное пособие. Параллельное программирование с использованием технологии Cuda [Електронний ресурс] – Режим доступу: <http://edu.chpc.ru/cuda/main.html>
2. Е.С.Борисов. Технология параллельного программирования Cuda [Електронний ресурс] – Режим доступу: <http://mechanoid.kiev.ua/parallel-cuda.html>
3. Программирование на CUDA. Работа с памятью [Електронний ресурс] – Режим доступу: <http://steps3d.narod.ru/tutorials/cuda-2-tutorial.html>

# Лабораторна робота №4. Знайомство з оптимізацією пам’яті у GPGPU на базі технології Nvidia Cuda

Мета роботи: ознайомлення з принципами оптимізація роботи з глобальною пам'яттю при програмуванні GPU.

## Теоретичні відомості

У GPU є ряд особливостей, ігнорування яких може коштувати багаторазової втрати продуктивності при використанні глобальної пам'яті. Але якщо врахувати всі тонкощі, то можна отримати дійсно ефективні CUDA-програми.

### Проблематика роботи з глобальною пам’яттю

Обсяг глобальної пам'яті найбільший з усіх типів пам'яті, але в той же час ця пам'ять – сама повільна за технічними характеристиками: швидкості зчитування та запису.

У попередній лабораторній роботі розглядалося транспонування матриці. Для підвищення продуктивності використовувався буфер пам'яті, що дозволило збільшити продуктивність майже в чотири рази. Але було досить дивно бачити це збільшення при надмірному посереднику. Секрет же криється в правильному зверненні до глобальної пам'яті.

Можна виділити два способи оптимізації в роботі з глобальною пам'яттю: вирівнювання розмірів використовуваних типів і використання об'єднаних запитів.

### Вирівнювання розмірів використовуваних типів

Вирівнювання типу даних дозволяє скомпілювати запит в глобальну пам'ять в одну команду GPU, в іншому випадку компілятор згенерує додатковий код, що може значно знизити продуктивність. Для оптимальної продуктивності тип даних повинен мати розмірність 4, 8 або 16 байт.

Якщо розмір типу не відповідає 4, 8 або 16 байтам, то краще використовувати тип більшої розмірності або провести вирівнювання за допомогою ключового слова \_\_align\_\_ (розмір вирівнювання).

Приклад оптимізації при використанні вбудованих CUDA-типів. Розмір типу int3 – 12 байт, доступ до пам'яті буде не оптимальним, кращій використовувати тип int4 (16 байтів), навіть якщо четвертий компонент не потрібен. Приклад використання обох типів наведено на рис 2.1 і 2.2, відповідно.

\_\_device\_\_ int3 data[512];

\_\_global\_\_ void initData()

{

int idx = threadIdx.x;

data[idx] = make\_int3(idx, idx, idx);

};

Рисунок 2.1 – Не оптимальне використання пам’яті за швидкістю,  
но оптимальне за використанням пам’яті

\_\_device\_\_ int4 data[512];

\_\_global\_\_ void initData()

{

int idx = threadIdx.x;

data[idx] = make\_int4(idx, idx, idx, 0);

};

Рисунок 2.2 – Оптимальне використання пам’яті за швидкістю,  
но не оптимальне за використанням пам’яті

У разі роботи зі структурами необхідно використовувати ключове слово \_\_align\_\_, яке дозволяє вирівнювати тип за заданим розміром. Приклад вирівняної і не вирівняної структур наведено на рис 2.3 і 2.4, відповідно. розміру структури. До вирівнювання розмір структури vector3 складе 12 байт, а після вирівнювання розмір vector3 складе 16 байт:

struct vector3 {

float x;

float y;

float z;

};

int main() {

printf("%i\n", sizeof(vector3));

return 0;

};

Рисунок 2.3 – Не оптимальне використання пам’яті за швидкістю,  
но оптимальне за використанням пам’яті

struct \_\_align\_\_(16) vector3 {

float x;

float y;

float z;

};

int main() {

printf("%i\n", sizeof(vector3));

return 0;

};

Рисунок 2.4 – Оптимальне використання пам’яті за швидкістю,  
но не оптимальне за використанням пам’яті

### Використання об'єднаних запитів

Куди більший приріст продуктивності можна отримати при об'єднанні великої кількості запит в глобальну пам'ять в один (іноді запити називають транзакціями). У документації Nvidia це має назву coalescing global memory accesses. Але, перед тим, як перейти до безпосереднього обговорення того, що необхідно для об'єднання запитів в пам'ять, необхідно знати пару додаткових речей про роботу GPU.

Для контролю виконання роботи ниток GPU використовує так званий варп. З програмної точки зору варп представляє пул ниток. Саме в межах цього варпу відбувається паралельна робота ниток, які були запитані при виклику ядра, саме в варпі нитки можуть взаємодіяти між собою. Розмір варпу для всіх GPU становить 32 чи більше в нових відеокартах, тобто паралельно в варпі виконуються тільки 32 нитки.

Одночасно на GPU можна запустити кілька варпів, це кількість визначається розмірами доступною реєстрової та розділяється пам'яті. Інша цікава особливість, що для доступу до пам'яті використовується half-warp, тобто на початку до пам'яті звертаються перші 16 ниток, а потім друга половина з 16 ниток. Доступ відбувається саме так оскільки це пов'язано з первинними завданнями GPU – обробкою графіки.

Тепер розглянемо вимоги, необхідні для об'єднання запитів в глобальну пам'ять. Не забуваємо, що звернення до пам'яті відбувається через half-warp.

Умови необхідні для об'єднання при зверненні в пам'ять залежать від версії Compute Capability 1.0 і 1.1, більше подробиць можна дізнатися в документації від Nvidia:

* Нитки повинні звертатися або до 32-бітових слів, даючи при цьому в результаті один 64-байтовий блок (транзакцію), або до 64-бітових слів, даючи при цьому один 128-байтовий блок (транзакцію).
* Якщо використовується звернення до 128-бітовим словами, то в результаті буде виконано дві транзакції, кожна з яких поверне по 128 байт інформації.
* Нитки повинні звертатися до елементів пам'яті послідовно, кожної наступної нитки має відповідати наступне слово в пам'яті (деякі нитки можуть взагалі не звертатися до відповідних словами).
* Всі 16 слів повинні бути в межах блоку пам'яті, до якого виконується доступ.

Пара приміток до умов:

* Під словами розуміється будь-який тип даних, головне - дотримання потрібних розмірностей.
* Розмірність слів дана в бітах, а розмірність одержуваних блоків даних в байтах.

Якщо нитки half-warp не задовольняють будь-якого з цих умов, то кожне звернення до пам'яті відбувається як окрема транзакція. На рис 3.1 і 3.2 наводяться типові патерни звернення, які дають об'єднання і не дають об'єднання, відповідно.

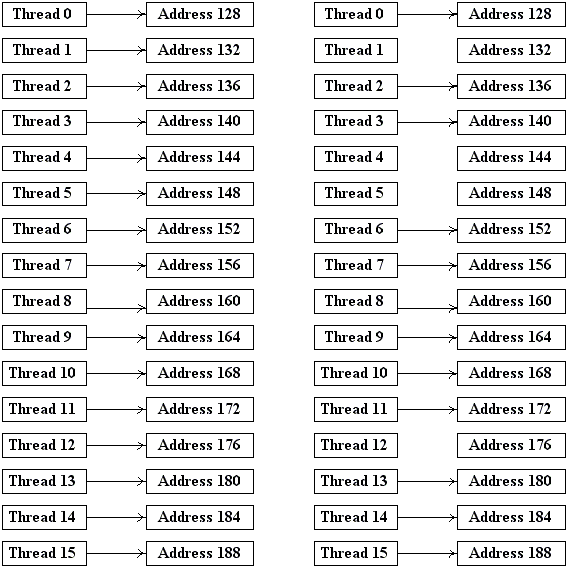


Рисунок 3.1 – Патерн запитів до пам’яті, що дає об'єднання   
згідно до Compute Capability 1.0 і 1.1

На рис. 3.1 наведено типові патерни звернення до пам'яті, що призводять до об'єднання запитів в одну транзакцію. Зліва у нас виконані всі умови, праворуч – просто для частини ниток пропущено звернення до відповідних слів (що так само дозволяє додати фіктивні звернення і звести до випадку зліва).

В той час на рис. 3.2 зліва для ниток 4 і 5 порушений порядок звернення до слів, а справа порушена умова вирівнювання - хоча слова, до яких йде звернення і утворюють безперервний блок з 64 байт, але початок цього блоку (за адресою 132) не кратне його розміру (16 байт ).

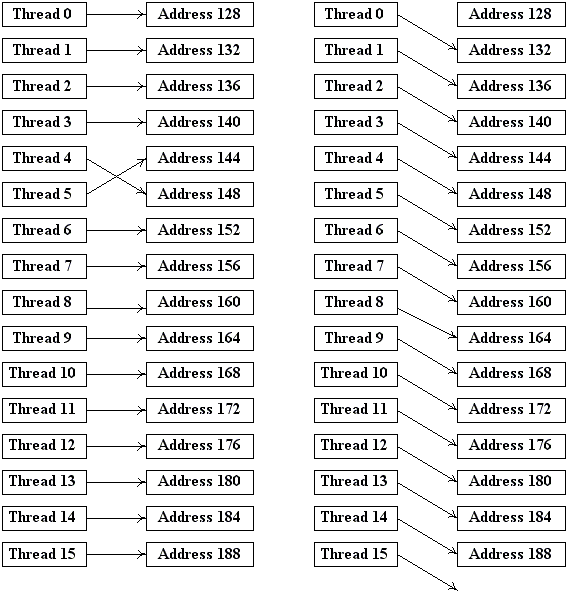


Рисунок 3.2 – Патерн запитів до пам’яті, що не дає об'єднання   
згідно до Compute Capability 1.0 і 1.1

Для GPU з сучасними Compute Capability 1.2 об'єднання запитів в один відбуватиметься, якщо слова, до яких йде звернення ниток, лежать в одному сегменті розміру 32 байта (якщо всі нитки звертаються до 8-бітовим словами), 64 байта (якщо всі нитки звертаються до 16-бітових слів) і 128 байт (якщо всі нитки звертаються до 32-бітових або 64-бітових слів). Добутий сегмент (блок) має бути вирівняний по 32/64/128 байтам.

Звернемо увагу, що в цьому випадку порядок, в якому нитки звертаються до слів, не має ніякого значення і ситуація на рис. 3.2 зліва призведе до об'єднання всіх запитів в одну транзакцію. Якщо йде звернення до n відповідних сегментах, то відбувається угруповання запитів в n транзакцій (тільки для GPU з Compute Capability 1.2 і вище).

Параметри нових Compute Capability можна завжди знайти у офіційній документації Nvidia у розділі “Maximize Instruction Throughput”

### Структури масивів або масиви структур

Ще одним моментом роботи з глобальною пам'яттю є те, що набагато ефективніше з точки зору об'єднання запитів до пам'яті є використання не масивів структур, а структури масивів (окремих компонент вихідної структури).

Тобто якщо є необхідність використання масиву структур, то краще створити окремі масиви компонентів структури, що дозволить зменшити кількість запитів в глобальну пам'ять за рахунок об'єднань.

struct \_\_align\_\_(16) vec3 {

float x;

float y;

float z;

};

\_\_device\_\_ vec3 data[SIZE];

\_\_global\_\_ void initData()

{

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

data[idx].x = idx;

data[idx].y = idx \* 2;

data[idx].z = idx \* 3;

};

Рисунок 4.1 – Неефективна робота з глобальною пам'яттю

\_\_device\_\_ float x[SIZE];

\_\_device\_\_ float y[SIZE];

\_\_device\_\_ float z[SIZE];

\_\_global\_\_ void initArr() {

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

x[idx] = idx;

y[idx] = idx \* 2;

z[idx] = idx \* 3;

};

Рисунок 4.2 – Ефективна робота з глобальною пам'яттю

У першому випадку використання масиву векторів для звернення до кожного полю структури необхідний окремий запит на згадку, у другому випадку за рахунок об'єднання досить 3 запитів для кожного half-warp. В середньому, цей підхід дозволяє збільшити продуктивність в 2 рази.

### Проблематика доступу до пам’яті

Ніколи не потрібно намагайтеся змінювати значення однієї осередку пам'яті декількома нитками одночасно. Це найчастіша помилка в багато потоковому програмуванні. Насправді Cuda не гарантує атомарного доступу для кожної нитки до певної області пам'яті, тому результати можуть вийти не зовсім такими, як очікується. Хоча атомарні операції в Cuda і існують, краще використовувати концепцію незмінних даних і зберігати результати розрахунків в нових об'єктах, які і передавати на наступні етапи розрахунків.

## Завдання до лабораторної роботи

1. Практично перевірити викладанні в теоретичній роботі відомості і наглядно продемонструвати отримані результати
2. Розгляни рішення створені у попередніх лабораторних роботах з точки зору розглянутої теоретичної інформації. Зробити висновки.

## Посилання на джерела

1. Программирование на CUDA. Работа с памятью [Електронний ресурс] – Режим доступу: <http://steps3d.narod.ru/tutorials/cuda-2-tutorial.html>
2. Марьин Д. Ф. Модель памяти GPU/CUDA Global memory [Електронний ресурс] – Режим доступу: [cmnd.bashedu.ru/docs/lectures/CUDA/Lection 03 - Memory model.pdf](http://cmnd.bashedu.ru/docs/lectures/CUDA/Lection%2003%20-%20Memory%20model.pdf)

# Лабораторна робота №5. Знайомство з обробкою зображень на базі технології Nvidia Cuda

Мета роботи: ознайомлення з основами роботи з зображення на базі бібліотеки OpenCV.

## Теоретичні відомості

### Робота з зображеннями за допомогою бібліотеки OpenCV

OpenCV (Open Source Computer Vision Library, бібліотека комп'ютерного зору з відкритим кодом) — бібліотека функцій та алгоритмів комп'ютерного зору, обробки зображень і чисельних алгоритмів загального призначення з відкритим кодом. Бібліотека надає засоби для обробки і аналізу вмісту зображень, у тому числі розпізнавання об'єктів на фотографіях (наприклад, осіб і фігур людей, тексту тощо), відстежування руху об'єктів, перетворення зображень, застосування методів машинного навчання і виявлення загальних елементів на різних зображеннях.

Оскільки бібліотека має достатньо скромну документацію, то більш детального з нею можна ознайомитися за допомогою публікацій і посібників, посилання на які наведено на відповідній сторінці офіційного сайту (<https://opencv.org/links.html>).

З тієї причини, що ми будемо використовувати тільки базові функції, а приклади їх використання будуть наведені то можна детально не ознайомлюватися з бібліотекою.

### Практична обробка зображення

Завдання поставимо досить просте: конвертація кольорового зображення у відтінки сірого. Для цього, яскравість пікселя pix в сірій шкалі обчислюється за формулою: Y = 0.299 \* pix.R + 0.587 \* pix.G + 0.114 \* pix.B.

Для порівняння продуктивності CPU і GPU результату будемо використовувати процесорну реалізацію алгоритму на OpenMP.

Розробку програми почнемо з створення функції, що відповідає за читання файлу із зображення і готує покажчики на кольорове і зараження у відтінках сірого зображення, тобто ця функція відповідає за доступ до показників вхідного та вихідного зображення. Приклад відповідної функції наведено на рис. 2.1.

// Підготовка показників вхідного та вихідного зображень

template <class T1, class T2>

void prepareImagePointers(

const char \* const inputImageFileName, cv::Mat& inputImage, T1\*\* inputImageArray,

cv::Mat& outputImage, T2\*\* outputImageArray, const int outputImageType)

{

using namespace std;

using namespace cv;

inputImage = imread(inputImageFileName, IMREAD\_COLOR);

if (inputImage.empty())

{

cerr << "Couldn't open input file." << endl;

exit(1);

}

// Розмітка пам'яті для збереження вихідного зображення

outputImage.create(inputImage.rows, inputImage.cols, outputImageType);

cvtColor(inputImage, inputImage, cv::COLOR\_BGR2BGRA);

\*inputImageArray = (T1\*)inputImage.ptr<char>(0);

\*outputImageArray = (T2\*)outputImage.ptr<char>(0);

}

Рисунок 2.1 – Функція підготовки показників на вхідне і вихідне зображення

Після цього створюємо функцію, що відповідає за OpenMP реалізацію проекту. Приклад коду цієї функції наведено на рис 2.2.

// Конвертування у відтінки сірого за допомогою OpenMP

void RGBtoGrayscaleOpenMP(uchar4 \*imageArray, unsigned char \*imageGrayArray, int numRows, int numCols) {

#pragma omp parallel for collapse(2)

for (int i = 0; i < numRows; ++i) {

for (int j = 0; j < numCols; ++j) {

const uchar4 pixel = imageArray[i\*numCols + j];

imageGrayArray[i\*numCols + j] = 0.299f\*pixel.x + 0.587f\*pixel.y + 0.114f\*pixel.z;

}

}

}

Рисунок 2.2 – Функція конвертування у відтінки сірого за допомогою OpenMP

Після цього створюємо макрос для налагодження функцій Cuda. Як вже згадувалось існує велика кількість варіантів реалізації відповідного макросу. В цій лабораторній реалізуємо його трішки іншим чином ніж в попередній. Код відповідного макросу наведений на рис 2.3.

#define checkCudaErrors(val) check( (val), #val, \_\_FILE\_\_, \_\_LINE\_\_)

template<typename T>

void check(T err, const char\* const func, const char\* const file, const int line) {

if (err != cudaSuccess) {

std::cerr << "CUDA error at: " << file << ":" << line << std::endl;

std::cerr << cudaGetErrorString(err) << " " << func << std::endl;

exit(1);

}

}

Рисунок 2.3 – Макрос для налагодження функції Cuda

Тепер реалізуємо функцію, що виконує наївну функцію конвертації зображення після цього модифікуємо її згідно з розглянутою у попередній лабораторній роботі теорією. Приклад відповідних функцій наведено на рис 2.4

// Оптимізована функція конвертації зображення

\_\_global\_\_ void rgba\_to\_grayscale\_optimized(const uchar4\* const d\_imageRGBA,

unsigned char\* const d\_imageGray,

int numRows, int numCols,

int elemsPerThread) {

int y = blockDim.y\*blockIdx.y + threadIdx.y;

int x = blockDim.x\*blockIdx.x + threadIdx.x;

const int loop\_start = (x / WARP\_SIZE \* WARP\_SIZE)\*(elemsPerThread - 1) + x;

for (int i = loop\_start, j = 0; j<elemsPerThread && i<numCols; i += WARP\_SIZE, ++j) {

const int offset = y\*numCols + i;

const uchar4 pixel = d\_imageRGBA[offset];

d\_imageGray[offset] = 0.299f\*pixel.x + 0.587f\*pixel.y + 0.114f\*pixel.z;

}

}

// Оптимізована функція конвертації зображення

\_\_global\_\_ void rgba\_to\_grayscale\_simple(const uchar4\* const d\_imageRGBA,

unsigned char\* const d\_imageGray,

int numRows, int numCols) {

int y = blockDim.y\*blockIdx.y + threadIdx.y;

int x = blockDim.x\*blockIdx.x + threadIdx.x;

if (x >= numCols || y >= numRows)

return;

const int offset = y\*numCols + x;

const uchar4 pixel = d\_imageRGBA[offset];

d\_imageGray[offset] = 0.299f\*pixel.x + 0.587f\*pixel.y + 0.114f\*pixel.z;

}

Рисунок 2.4 – Функції конвертування у відтінки сірого за допомогою Cuda

Тепер напишемо функцію, яка буде виділити пам'ять на під вхідні дані у пам’яті GPU, перемістити їх з CPU на GPU, виділити пам'ять під вихідні дані на GPU і повертати отриманий результат до CPU. Приклад відповідної функцій наведено на рис. 2.5.

// Конвертування зображень за допомогою Cuda двома способами

void RGBtoGrayscaleCUDA(const uchar4 \* const h\_imageRGBA, unsigned char\* const h\_imageGray, size\_t numRows, size\_t numCols) {

uchar4 \*d\_imageRGBA;

unsigned char \*d\_imageGray;

const size\_t numPixels = numRows \* numCols;

cudaSetDevice(0);

checkCudaErrors(cudaGetLastError());

checkCudaErrors(cudaMalloc(&d\_imageRGBA, sizeof(uchar4) \* numPixels));

checkCudaErrors(cudaMalloc(&d\_imageGray, sizeof(unsigned char) \* numPixels));

checkCudaErrors(cudaMemcpy(d\_imageRGBA, h\_imageRGBA, sizeof(uchar4) \* numPixels, cudaMemcpyHostToDevice));

dim3 blockSize;

dim3 gridSize;

int threadNum;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

threadNum = 1024;

blockSize = dim3(threadNum, 1, 1);

gridSize = dim3(numCols / threadNum + 1, numRows, 1);

cudaEventRecord(start);

rgba\_to\_grayscale\_simple << <gridSize, blockSize >> >(d\_imageRGBA, d\_imageGray, numRows, numCols);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());

float milliseconds = 0;

cudaEventElapsedTime(&milliseconds, start, stop);

std::cout << "CUDA time simple (ms): " << milliseconds << std::endl;

threadNum = 128;

const int elemsPerThread = 16;

blockSize = dim3(threadNum, 1, 1);

gridSize = dim3(numCols / (threadNum\*elemsPerThread) + 1, numRows, 1);

cudaEventRecord(start);

rgba\_to\_grayscale\_optimized << <gridSize, blockSize >> >(d\_imageRGBA, d\_imageGray, numRows, numCols, elemsPerThread);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());

milliseconds = 0;

cudaEventElapsedTime(&milliseconds, start, stop);

std::cout << "CUDA time optimized (ms): " << milliseconds << std::endl;

checkCudaErrors(cudaMemcpy(h\_imageGray, d\_imageGray, sizeof(unsigned char) \* numPixels, cudaMemcpyDeviceToHost));

cudaFree(d\_imageGray);

cudaFree(d\_imageRGBA);

}

Рисунок 2.5 – Функція хосту, що здійснює конвертування зображення   
двома способами за допомогою Cuda

І тепер для того, щоб запустити і протестувати написану програму напишемо головну функцію програми. Приклад коду відповідної функції наведений на рис. 2.6.

int main(int argc, char\*\* argv) {

using namespace cv;

using namespace std;

using namespace std::chrono;

if (argc != 2) {

cout << " Usage: convert\_to\_grayscale imagefile" << endl;

return -1;

}

Mat image, imageGray;

uchar4 \*imageArray;

unsigned char \*imageGrayArray;

prepareImagePointers(argv[1], image, &imageArray, imageGray, &imageGrayArray, CV\_8UC1);

int numRows = image.rows, numCols = image.cols;

auto start = system\_clock::now();

RGBtoGrayscaleOpenMP(imageArray, imageGrayArray, numRows, numCols);

auto duration = duration\_cast<milliseconds>(system\_clock::now() - start);

cout << "OpenMP time (ms):" << duration.count() << endl;

memset(imageGrayArray, 0, sizeof(unsigned char)\*numRows\*numCols);

RGBtoGrayscaleCUDA(imageArray, imageGrayArray, numRows, numCols);

return 0;

}

Рисунок 2.6 – Головна функція програми

## Завдання до лабораторної роботи

1. Реалізувати програму, що описана в пункті 2 теоретичної частини. І виконати оцінки швидкості виконання обчислень при різних розмірах зображень. Оцінити чисту швидкість обчислень Cuda і з передачею даних, а також оцінити швидкість роботи OpenMP. Зробити висновки.
2. За аналогічним підходом реалізувати на Cuda і оцінити один з алгоритмів: “Salt and Paper”, “Фільтр Гаусса” чи “Медіанний фільтр”. Зробити відповідні висновки.

## Посилання на джерела

1. UDACITY COURSE. Intro to Parallel Programming by Nvidia. Using Cuda to Harness the Power of GPUs [Електронний ресурс] – Режим доступу: https://www.udacity.com/course/intro-to-parallel-programming--cs344
2. Параллельное программирование с CUDA. Часть 1: Введение [Електронний ресурс] – Режим доступу: [https://habrahabr.ru/company/ epam\_systems/blog/245503/](https://habrahabr.ru/company/epam_systems/blog/245503/)
3. Параллельное программирование с CUDA. Часть 2: Аппаратное обеспечение GPU и шаблоны параллельной коммуникации [Електронний ресурс] – Режим доступу: [https://habrahabr.ru/company/epam\_systems/blog /245523/](https://habrahabr.ru/company/epam_systems/blog/245523/)
4. Параллельное программирование с CUDA. Часть 3: Фундаментальные алгоритмы GPU: свертка (reduce), сканирование (scan) и гистограмма (histogram) [Електронний ресурс] – Режим доступу: [https://habrahabr.ru /company/epam\_systems/blog/247805/](https://habrahabr.ru/company/epam_systems/blog/247805/)
5. Пример оптимизации вычислений на Cuda [Електронний ресурс] – Режим доступу: <https://habrahabr.ru/post/211194/>