УДК 004.4, 004.9, 004.42, 004.67, 004.915, 004.427

# ОДИН ПІДХІД ДО ВИРІШЕННЯ ПРОБЛЕМИ УНІВЕРСАЛЬНОГО ВИКОРИСТАННЯ МОВИ ПРОГРАМУВАННЯ OPENCL НА РІЗНИХ GPU

*А.М. Лавренюк[1], С.І. Лавренюк[2]*

[1]Фізико-технічний інститут НТУ України "КПІ"
проспект Перемоги 37, корпус № 1, м. Київ, 03056, Україна,
тел.: 236 7098, 454 9876
[2]Інститут кібернетики ім. В.М. Глушкова НАН України,
проспект Академіка Глушкова, 40, Київ, 03680 МСП, Україна,
тел.: 526 2008, факс 526 7418

Розглянуто питання застосування шаблонів функцій мови програмування С++ для проектування універсальних ядер-програм на OpenCL для оптимізації обчислень на GPU різної архітектури.

This paper about the question of the use of templates features programming language C + + to design universal nuclear programs on OpenCL for GPU optimization calculations on different architectures.

## Вступ

Незважаючи на інтенсивний розвиток грід-мереж та хмарних обчислень проблема доступності потужних ресурсів для наукових задач доволі актуальна. Особливо залишається потреба у високопродуктивних обчисленнях за зниженими цінами, так як побудова сучасного кластера або великої грід-мережі є доволі затратними.

Різноманітні організації певний час розглядають потенційні переваги використання графічних процесорів (GPU) для комплексного моделювання та аналізу даних. Адже якщо глянути поверхнево, то перехід GPU технології із використання для обробки спеціальних ефектів в 3D графіці для ігор та кіно до використання в наукових дослідженнях не такий вже дивний. Як науковці, так і дизайнери комп'ютерної графіки мають обробляти великі обсяги даних, причому дуже швидко. Це актуально і для розв’язування рівнянь гідродинаміки, що використовуються для моделювання потоків повітря в атмосфері при прогнозуванні змін клімату і для моделювання в сейсморозвідці, медицині, біології та інших галузях науки [1, 2].

Учені хочуть мати можливості обробляти великі обсяги даних у більш короткі періоди часу.

На сучасному етапі вченим приходиться працювати з набагато більшою кількістю даних, ніж спроможні обробити їх одно- або двоядерні настільні ПК, а також виконувати математичні моделювання, які мають працювати більш точно та стабільно, ніж будь-коли раніше.

Все це ставить під загрозу те, що сучасні наукові проблеми не можуть бути вирішені сьогодні через обмеження в обчислювальній потужності. Деякі експерти до цього часу обговорюють питання про те, чи призначені поточні графічні процесори для широкого застосування. Але графічні процесори вже зарекомендували себе, як обчислювальні ресурси, у таких областях, як медична обробки зображень і аналіз сейсмічних даних.

Експерти в минулому сподівалися, що прискорення обчислень може бути досягнуто за рахунок розподілу навантаження по групах багатьох багатоядерних процесорів, але GPU захисники вважають, що велика частина цих обчислювань буде здійснюватися на графічних процесорах у майбутньому, а CPU з декількома ядрами будуть виступати як менеджери для управління більшістю важкої роботи, яка піде на відкуп на GPU.

GPU навряд чи зробить процесор зайвим, останній має більший обсяг пам'яті та можливість більш ефективно виконувати організацію та управління різними роботами, що виконуються на комп'ютері, а також прийняття рішень про те, як краще поводитися з задачею. Замість цього, GPU буде виступати в як ферми по високоефективній обробці – де дані вивантажені з процесора для виконання повторюваних розрахунків при дуже швидкому темпі обробляються на GPU. Оброблені дані будуть потім повертатися до CPU, які будуть використовувати результати для формування рішення задачі.

Графічні процесори ефективні при виконанні швидких розрахунків у зв'язку з їх будовою, що дозволяє виконувати більше 320 обчислень за одиницю часу на одному GPU. Процесор може виконувати паралельні обчислення, але вони обмежені кількістю доступних ядер. Частота (ГГц) процесорів знаходиться на етапі, коли при її збільшенні отримується малий виграш у потужності, водночас як потужність графічного процесора в даний час стрімко зростає, і графічні процесори будуть здатні обробляти дані ще більш швидкими темпами і в майбутньому.

© А.М. Лавринюк, С.І. Лавренюк, 2012

## Векторні та скалярні операції на GPU

GPU складається з уніфікованих процесорів, які NVIDIA називає уніфікованими потоковими процесорами (Unified Streaming Processors, SP) та являють собою скалярні процесори загального призначення для обробки даних с плаваючою комою. Традиційно в процесорах існує два типи математики: векторна та скалярна. У випадку векторної математики дані ( операнди ) представляються в вигляді *n* - вимірних векторів, при цьому над великим масивом даних виконується всього одна операція. Самий простий приклад – колір пікселя задається у вигляді чотирьохвимірного вектора з координатами R, G, B, A, де перші три координати (R, G, B) задають колір пікселя, а остання – його прозорість. Як простий приклад векторної операції можна розглянути складні кольори двох пікселів. При цьому одна операція виконується одночасно над всіма вісьмома операндами (двома 4-мірними векторами). У скалярній математиці операції виконуються над парою чисел. Зрозуміло, що векторна обробка збільшує швидкість та ефективність обробки за рахунок того, що обробка цілого набору (вектора) даних виконується однією командою. Приклад програмний цикл:

```
for(short i = 0; i<n; i++)
{
        A[i] = B[i]+C[i];
}
```

У скалярному режимі потрібно згенерувати цілу послідовність команд: прочитати елемент B[i], прочитати елемент C[i], виконати операцію додавання, записати результат в A[i], збільшити параметр циклу, перевірити умову циклу... В векторному режимі масиви елементів B[i] та C[i] можна розглядати як *n* - вимірні вектори. В такому разі цей фрагмент коду перетворюється в наступну послідовність: загрузити масив B, загрузити масив C, виконати операцію векторного додавання і записати результат у масив A.

Так, наприклад, у популярному донедавна графічному процесорі NVIDIA GeForce 8800 реалізовані уніфіковані скалярні процесори, а не векторні. Векторна архітектура є у певній мірі традиційними для графічних процесорів по причині того, що вони працюють з переважною кількістю операцій з векторними даними, такими як компонентна R-G-B-A-обробока в піксельних шейдерах та геометричне перетворення 4x4-матриць у вершинних шейдерах.

У графічних процесорах NVIDIA попереднього покоління, а також у графічних процесорах ATI використовується векторна архітектура обчислювальних блоків. У результаті векторні обчислювальних блоки в останніх версіях графічних процесорів ATI можуть виконувати одну векторну операцію за такт для чотирьохелементних (4- вимірних) векторів або одну векторну операцію для трьохелементних векторів плюс одну скалярну операцію (схема «3+1»). Векторні обчислювальні блоки в графічних процесорах NVIDIA GeForce 6x і GeForce 7x працюють за схемою «2+2», тобто можуть виконувати одночасно дві векторні операції для двохелементних векторів або одну векторну операцію для чотирьохелементних векторів.

В останній час спостерігається перехід від векторних до скалярних обчислень. Так, проаналізував сотні шейдерних програм, розробники NVIDIA пришли до висновку, що традиційна векторна архітектура менш ефективно використовує обчислювальні ресурси, ніж скалярний дизайн процесорних модулів, особливо в випадку обробки складних змішаних шейдерів, що поєднують векторні та скалярні інструкції. Крім того, достатньо складно добитися ефективної обробки скалярних обчислень за допомогою векторних обчислювальних модулів. Тому в уніфікованих процесорах NVIDIA використовуються скалярні обчислювальні блоки. При цьому векторний шейдерний програмний код перетворюється в скалярні операції безпосередньо графічним процесором, наприклад, GeForce 8800 [3].

## Про стандарт OpenCL

Враховуючи те, що сучасні комп'ютери часто включають високо паралельні CPU, GPU та інші типи процесорів, важливо дати можливість розробникам програмного забезпечення (ПЗ) використовувати ці гетерогенні процесорні системи в повній мірі.

Створення ПЗ для гетерогенних паралельних обчислювальних платформ є складним процесом, оскільки класичні підходи до програмування для багатоядерних CPU та GPU значно різняться. Моделі програмування CPU хоча в більшості випадків і основані на стандартах, але зазвичай вони пропонують наявність спільного адресного простору та не враховують можливості векторних операцій.

Моделі програмування GPU загального призначення, що включають складні ієрархії пам'яті та векторні операції, традиційно залишаються залежними від платформи. Ці обмеження утруднюють доступ розробників до загальної бази вихідних кодів для CPU, GPU та інших типів процесорів. Як ніколи, нині потрібно надавати можливість розробникам програмного забезпечення ефективно використовувати всі переваги гетерогенних обчислювальних платформ – від високопродуктивних обчислювальних серверів, через персональні комп'ютери до мобільних пристроїв, які містять різнорідні паралельні CPU, GPU та інші процесори.

OpenCL (Open Computing Language) [4] – відкритий, не потребує ліцензійних відрахувань стандарт для універсального паралельного програмування різних типів процесорів. Стандарт надає програмістам переносний та ефективний доступ до всієї мощі гетерогенних обчислювальних платформ.

У результаті, OpenCL спроможний сформувати базовий рівень паралельного обчислювального коду незалежних від апаратної платформи програмних інструментів, проміжного ПЗ та інших видів програм.

Стандарт OpenCL:

- підтримує різні моделі паралелізму;
- використовує підмножину стандарту C99 з розширеннями для підтримки паралелізму;
- підтримує стандарт арифметики чисел с плаваючою комою IEEE 754;
- визначає профілі конфігурацій для переносних та вбудовуваних пристроїв.

## Проблема використання мови програмування OpenCl на «векторних» та «скалярних» GPU

Вже давно іде інтенсивний розвиток великих і малих кластерів з GPU. Багато з таких кластерів доступні в грід-мережі, як грід-вузли [5, 6]. Щоб отримати максимальний приріст продуктивності рекомендують задіяти багато GPU доступних на вузлі [7]. Відповідно часто можливі випадки, коли програма буде виконуватися на GPU різної архітектури та різних поколінь [8].

Останнім часом визначилося два лідери у виробництві графічних карт з підтримкою паралельних обчислень Nvidia, AMD/ATI, котрі по різному відносяться до векторних операцій. У GPU Nvidia не підтримуються векторні операції, в AMD/ATI – підтримуються, при чому на різних GPU довжини векторів певних типів різна.

Програма на OpenCL може вести себе по різному на різних GPU [7]. Це підтверджується проведеними експериментами.

Так, з рис. 1 видно, що при запуску програми на OpenCL на CPU в режимі GPU ми маємо зменшення часу виконання програми при збільшенні вектора даних. Причому це справедливо для трьох основних типів даних (слід зазначити, що тип double поки підтримується не на всіх GPU, тому з цим типом даних експериментів не проводили).

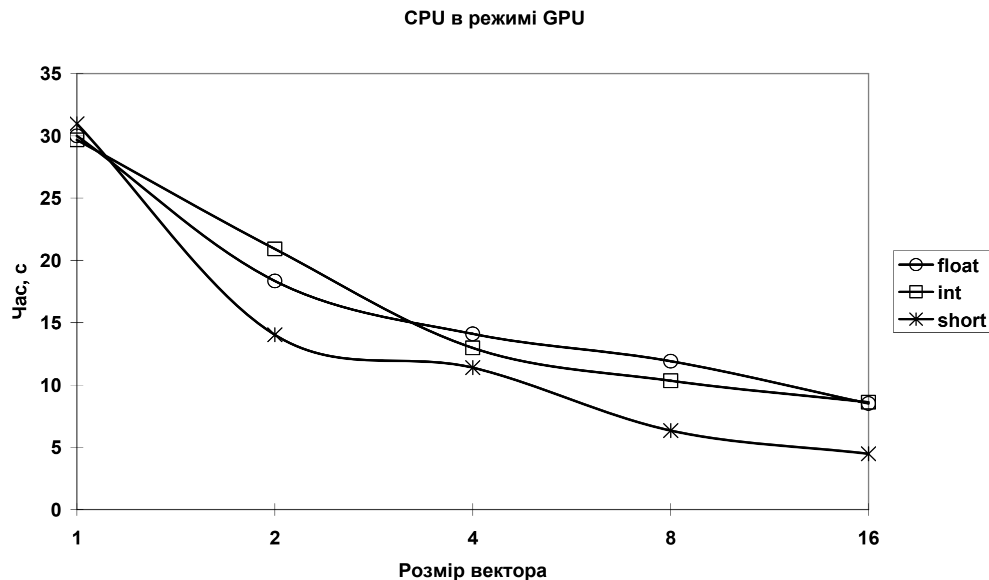


Рис. 1. Залежність часу обчислень від типу даних та довжини вектора даних на CPU в режимі GPU

CPU, що використовувалося в даному експерименті підтримує такі типи даних і довжини векторів: CHAR 16, SHORT 8, INT 4, LONG 2, FLOAT 4, DOUBLE 0.

З рис. 2 видно, що найкращий результат роботи програми не на будь-якій довжині вектора, а тільки на тих, що підтримує GPU. GPU, що використовувалося в даному експерименті підтримує такі типи даних і довжини векторів: CHAR 16, SHORT 8, INT 4, LONG 2, FLOAT 4, DOUBLE 0.

GPU, що використано в експериментах рис. 3, 4 підтримують такі типи даних і довжини векторів: CHAR 1, SHORT 1, INT 1, LONG 1, FLOAT 1, DOUBLE 1. Тобто, всі вектори розбиваються на скаляри і далі над ними виконуються операції.

Нажаль, тут не всі GPU ведуть себе однаково. Відповідно, якщо напишемо програму на OpenCL без підтримки векторних операцій ми втратимо в потужності на тих пристроях, що підтримують векторні операції. Якщо напишемо програму на OpenCL, яка підтримує векторні операції, то на одних GPU (рис. 3, 4) при певних довжинах векторів ми можемо не втратити потужності, а на інших (рис. 5.) можемо отримати негативний ефект – зменшення потужності.

Писати декілька різних програм-ядер на OpenCL можна лише, якщо програма доволі проста. При великих програмах це дуже трудомістка задача і підтримувати різні варіанти програми не проста задача.

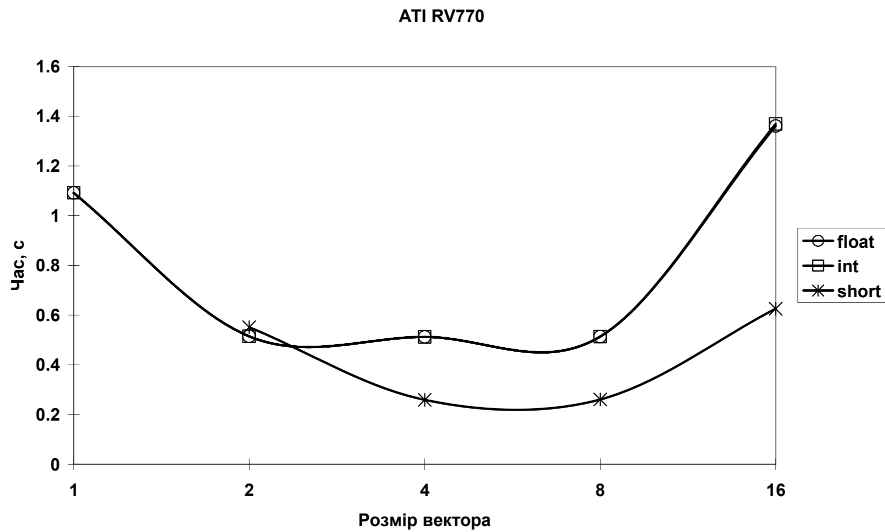


Рис. 2. Залежність часу обчислень від типу даних та довжини вектора даних на графічній карті ATI RV770

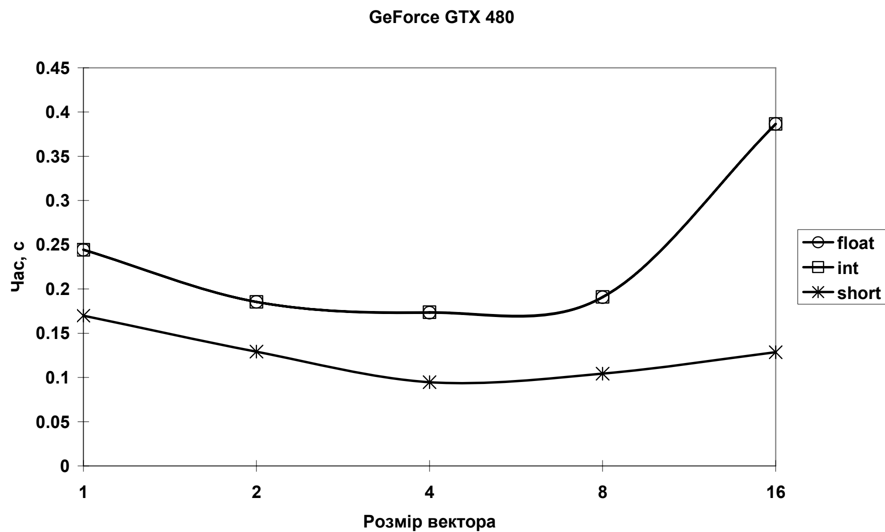


Рис. 3. Залежність часу обчислень від типу даних та довжини вектора даних на графічній карті NVidia GeForce 480

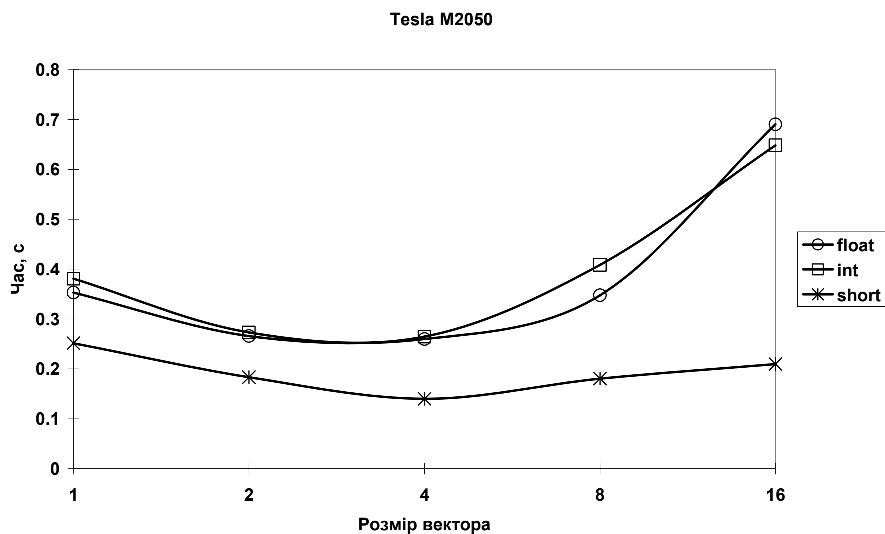


Рис. 4. Залежність часу обчислень від типу даних та довжини вектора даних на графічній карті NVidia Tesla M2050

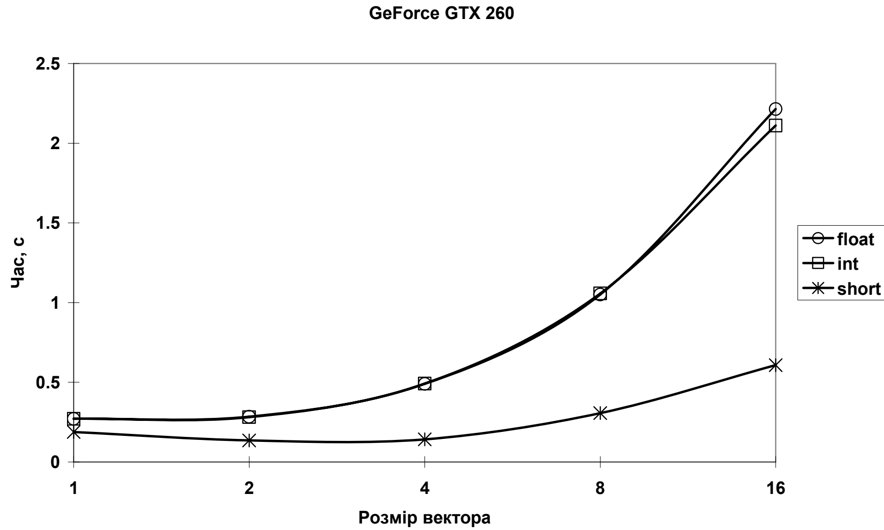


Рис. 5. Залежність часу обчислень від типу даних та довжини вектора даних на графічній карті NVidia GeForce 260

## Оптимізація програм для використання векторних і скалярних операцій GPU

На рис. 6. запропоновано алгоритм оптимізації програми OpenCL для роботи з векторними та скалярними GPU.

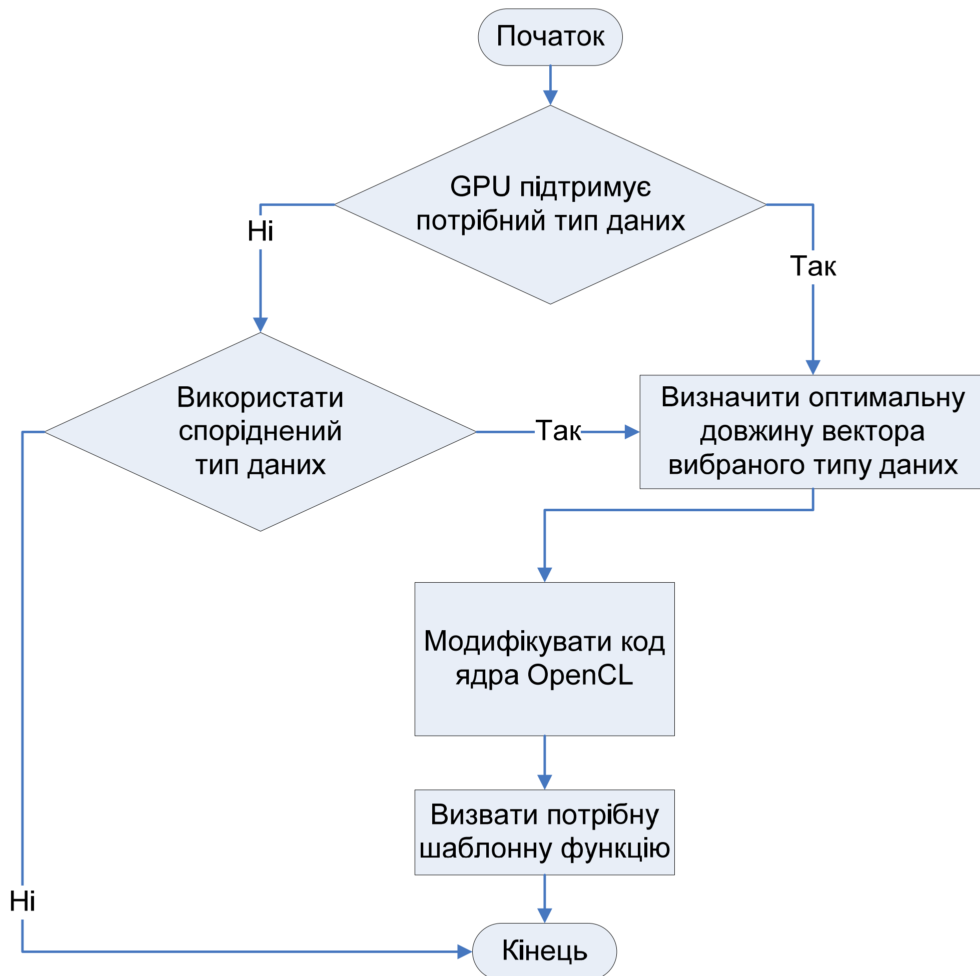


Рис. 6. Алгоритм оптимізації програми OpenCL для роботи з векторними та скалярними GPU

Програмно легко визначити типи даних та довжини векторів, що підтримує GPU. Але всі скалярні GPU дадуть, як результат, довжину вектора рівною 1, хоча, як видно з проведених експериментів оптимальна швидкість обробки даних може бути на більшій довжині вектора ніж 1. Тому пропонується провести запуск тестових програм на різних довжинах вектору даних і визначити оптимальну довжину вектора даних, що дає максимальну швидкодію. Як варіант тестових задач можливо використати програму-ядро із набором операцій, які будуть використовуватися в реальній програмі-ядрі.

Приклад:

```
typedef __global int4 mtype;
__kernel void kernel_test(mtype * A, mtype * B, mtype * C, mtype * D, const unsigned int nx_max)
{
  int j=get_global_id(1);
  int i=get_global_id(0);
  int nx_BLOCK=nx_max;
  A[j*nx_BLOCK+i]=(B[j*nx_BLOCK+i]-C[j*nx_BLOCK+i])*D[j*nx_BLOCK+i];
}
```

Після того, як визначено тип даних та довжину вектора даних, необхідно провести програмно зміну визначення типу mtype в програмі-ядрі.

Так як програма-ядро на OpenCL зберігається в вигляді текстової змінної типу char *, то не важно в ядрі модифікувати рядок програми (нажаль поки OpenCL не підтримує шаблони, так як ця мова є похідною від С, а шаблони функцій введені в мові С++ [9]):

```
typedef __global int4 mtype;
```

А ядро програми на OpenCL залишати не змінним. І в ньому використовувати перевизначений вище тип даних.

```
__kernel void kernel_test(  mtype * A, mtype * B, mtype * C, mtype * D, const unsigned int nx_max)
{
int j=get_global_id(1);
  int i=get_global_id(0);
  int nx_BLOCK=nx_max;

A[j*nx_BLOCK+i]=(B[j*nx_BLOCK+i]-C[j*nx_BLOCK+i])*D[j*nx_BLOCK+i];
}
```

З програмою, що буде виконуватися на хості та готовити дані для ядра і запускати на виконання програму-ядро на OpenCL ситуація трохи складніша. Динамічно перевизначити тип даних складно.

Як вихід запропоновано і перевірено наступний підхід.

Зробити мінімум дві функції, перша визначає типи даних, довжини векторів, та викликає на виконання функцію, що визначена на основі шаблона.

Приклад такої функції, що викликає шаблонну функцію:

```
main_run()
{
…
    type1=5; //Нам потрібний тип double

    ret=testtype(type1, len1); //перевіряємо, чи підтримує GPU потрібний нам тип даних і проводимо тест на
    визначення оптимальної довжими вектора даних на вибраного GPU

    if(ret!=0) {
    printf("GPU не пітримує необхідного типу надних і для нашої задачі неможливо перейти на споріднений тип
    даних!");
    exit(-1);
    }

    if(len1==4)
    {
        const int N=4;
```

```
        if(type1==4)
        {
            float f[N];
            ret=run(f);
        }
        if(type1==2)
        {
            int t[N];
            ret=run(t);
        }
        if(type1==1)
        {
            short s[N];
            ret=run(s);
        }
    }
    …
}
```

Приклад шаблонної функції для підготовки даних для програми-ядра на OpenCL та запуску програми-ядра на GPU:

```
template <class Type, int size> int run( const Type (&r_array)[size] )
{
        const int loc_size = size;
        typedef union {Type s[loc_size];} mtype;  //Створюємо векторний тип на основані типу Type, довжина вектора loc_size

    mtype *B=NULL;
    int sizeXZ=nx_max*ny_max;

    int nx_max4=nx_max/loc_size;
    int sizeXZ4=nx_max4*ny_max;

    B=new mtype[sizeXZ4];

    for(int j=1;j<ny_max-1; j++)
        for(int i=1;i<nx_max-1; i++)
                for(int k=0;k<loc_size;k++)
                        B[j*nx_max+i].s[k]=(Type) 123.456f*i;
    // Вирішуємо проблему різної довжини векторів
    // Вирішуємо проблему приведення до векторного типу mtype через приведення до базового типу (Type)
…
```

На перший погляд здається, що все це можна було б вирішити за допомогою векторних типів даних, таких як cl_float4, cl_int8. Але тут виникає проблема при довжині вектора 1. Так, наприклад, при оголошенні cl_float *B  і виконанні коду B[j*nx_max+i].s[k]=… буде виникати помилка. А вставка операцій умовного оператора, для того, щоб обійти помилку буде сповільнювати роботу програми (при оптимізації програми на C++ рекомендовано уникати використання else в операторі if) та зменшувати зручність роботи з кодом:

```
    for(int j=1;j<ny_max-1; j++)
        for(int i=1;i<nx_max-1; i++)
        if(loc_size==1)
            B[j*nx_max+i]=(Type) 123.456f*i;
        else
                for(int k=0;k<loc_size;k++)
                        B[j*nx_max+i].s[k]=(Type) 123.456f*i;
```

При запуску програми-ядра на OpenCL варто врахувати те, що розмір рядків матриці буде зменшено, пропорційно до довжини вектора:

```
nx_max4=nx_max/loc_size;
```

І функцію, що запускає на виконання програму-ядро на OpenCL слід викликати наступним чином:

```
size_t globalgroup[3]={nx_max4, ny_max, 1};
err=clEnqueueNDRangeKernel (queue, kernel, work_dim, NULL, globalgroup, NULL , 0, NULL, NULL);
```

## Висновки

Наведено результати використання одного з підходів до підвищення продуктивності програм на мові OpenCL при запуску їх на GPU пристроях різної архітектури. Запропонований підхід дає можливість:

1) визначити за допомогою тесту оптимальний режим роботи з GPU;
2) уніфікувати програму для роботи з різними GPU пристроями з малими затратами, та з прозорістю програмного коду та легкістю його подальшої підтримки та модернізації;
3) використовувати переваги векторних операцій на GPU, які підтримують векторні операції;
4) уникати проблем, які можуть виникати, при запуску «векторної» програми на GPU пристроях, що не підтримують векторні операції.


1. *Li Bo, Liu Guo-feng, Liu Hong.* A method of accelerating seismic Pre-stack time migration by GPU . //[Електронний ресурс] - http://cm.seg.org/documents/10161/997244/1202.pdf
2. *Virasin Archirapatkave, Hendra Sumilo, Simon Chong Wee See, Tiranee Achalakul.* GPGPU Acceleration Algorithm for Medical Image Reconstruction // ISPA '11 Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications table of contents P. 41–46 Publisher IEEE Computer Society Washington, USA.
3. *Графические* процессоры GeForce [Електронний ресурс] http://www.nvidia.ru/object/geforce_family_ru.html
4. *OpenCL* – The open standard for parallel programming of heterogeneous systems [Електронний ресурс] http://www.khronos.org/opencl/
5. *Lizandro Solano-Quinde, Zhi Jian Wang, Brett Bode, and Arun K. Somani*. Unstructured grid applications on GPU: performance analysis and improvement // In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4). ACM, New York, NY, USA, Article 13, 8 pages.
6. *Corrigan A., Camelli F., Rainald L.* Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware // 19th AIAA Computational Fluid Dynamics, Jun, 2009.
7. *Chris Jang.* OpenCL™ Optimization Case Study: GATLAS – Designing Kernels with Auto-Tuning [Електронний ресурс] - http://golem5.org/gatlas/CaseStudyGATLAS.htm
8. *Marcus Hinders.* GPU Computations in Heterogeneous Grid Environments, Joint Research Report [Електронний ресурс] - http://www.techila.fi/technology/technology-docs/
9. *ISO/IEC* 14882:1998 Programming languages - C++ //[Електронний ресурс] -http://www.iso.org/iso/catalogue_detail.htm?csnumber=25845