Міністерство освіти і науки України

Національний університет водного господарства та природокористування

Кафедра обчислювальної техніки

04-04-258М

Методичні вказівки

до лабораторних робіт з навчальної дисципліни
«Паралельні та розподілені обчислення» (частина 1)
для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Комп’ютерна інженерія» спеціальності 123 «Комп’ютерна інженерія» денної і заочної форм навчання

Рекомендовано науково-методичною радою з якості ННІАКОТ
Протокол № 8 від 19.06.2023 р.

Рівне – 2023

Методичні вказівки до лабораторних робіт з навчальної дисципліни «Паралельні та розподілені обчислення» (частина 1) для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Комп'ютерна інженерія» спеціальності 123 «Комп'ютерна інженерія» денної і заочної форм навчання [Електронне видання] / Бойчура М. В., Шатний С. В., Шатна А. В. – Рівне : НУВГП, 2023. – 49 с.

Укладачі: Бойчура М. В., к.т.н., старший викладач кафедри обчислювальної техніки;
Шатний С. В., к.т.н., доцент кафедри обчислювальної техніки;
Шатна А. В., старший викладач кафедри обчислювальної техніки.

Відповідальний за випуск: Круліковський Б. Б., к.т.н., доцент, завідувач кафедри обчислювальної техніки.

Керівник (гарант) освітньої програми
«Комп'ютерна інженерія»
спеціальності
123 «Комп'ютерна інженерія» Сидор А. І.

© М. В. Бойчура,
С. В. Шатний,
А. В. Шатна, 2023
© НУВГП, 2023

ЗМІСТ

Вступ 4

Лабораторна робота №1 Високопродуктивні обчислення на відеокартах 7

1.1. Вступ до технологій OpenCL та CUDA 7

1.2. Забезпечення можливостей програмування на технології OpenCL у Microsoft Visual Studio при наявній відеокарті від компанії NVIDIA......................................8

1.3. Забезпечення можливостей програмування на технології CUDA у Microsoft Visual Studio при наявній відеокарті від компанії NVIDIA....................................18

1.4. Забезпечення можливостей програмування на технології OpenCL у Microsoft Visual Studio при наявній відеокарті від компанії AMD 26

1.5. Завдання 32

Лабораторна робота №2 Високопродуктивні обчислення на процесорах...35

2.1. Вступ до технології MPI.....................................35

2.2. Забезпечення можливостей програмування на технології MPI у Microsoft Visual Studio при наявній відеокарті від компанії AMD 36

2.3. Завдання 42

Лабораторна робота №3 Сумісні високопродуктивні обчислення на відеокартах та процесорах 44

3.1. Вступ до технології OpenMP..............................44

3.2. Створення першого проекту на OpenMP у Microsoft Visual Studio...45

3.3. Завдання 46

Рекомендована література 48

Вступ

Як правило, студенти навчаються складати програмний код, не приділяючи увагу питанням можливості його виконання на паралельних потоках. Таким чином, у процесі роботи програми може задіюватись лише одне ядро центрального процесора, а інші – простоювати. Вміння оперувати паралельними потоками дозволяє фахівцю з комп’ютерної інженерії будувати додатки різного призначення із можливістю використання максимальних потужностей комп’ютерної техніки. Це буває критично важливим, наприклад, у широкому класі наукових досліджень, при розробці власної криптовалюти, майнінгу криптовалюти, розробці ігор, веб-сайтів, інформаційних систем реального часу тощо.

З іншого боку, окрім центральних процесорів, сучасні комп’ютери мають ще один потужний обчислювальний пристрій, а саме: відеокарта. Насправді у наведеному вище переліку застосувань паралельних обчислень часто не беруть до уваги потужності центральних процесорів, а зосереджуються лише на можливостях відеокарт. Останні зачасту можуть мати тисячі порівняно непотужних ядер, але за рахунок, в першу чергу, їх кількості, інколи можуть забезпечити виконання коду в сотні тисяч разів швидше, ніж на десятках потужних ядер центрального процесора.

Отже, набуття знань та навичок з паралельних обчислень сприятиме побудові ефективних програм.

Предмет «Паралельні та розподілені обчислення» є вибірковою компонентою освітньої програми 123 «Комп’ютерна інженерія» бакалаврського рівня здобуття освіти. Дані методичні вказівки містять постановки завдань перших трьох лабораторних робіт, покрокові

інструкції встановлення популярних бібліотек розпаралелення обчислень, приклади працюючого коду, критерії оцінювання та контрольні запитання. Виконання даних лабораторних робіт дозволить набути навички оцінювання виду: чи потребує та чи інша програма розпаралелення обчислень та яку технологію доцільно, при цьому, застосовувати.

Результати виконаних завдань можна представляти викладачу у вигляді звіту або просто демонструвати код програми і вікно з розв’язками.

Для отримання навіть мінімальної оцінки студент повинен пояснити кожен рядок коду.

Як альтернатива, студенти можуть виконувати й інші завдання, навіть за допомогою інших технологій високопродуктивних обчислень на процесорах/відеокартах чи довільних мовах програмування. Але попередньо узгодьте деталі з Вашим викладачем. Також передбачений варіант перезарахування балів за результатами неформальної освіти.

Дисципліна «Паралельні та розподілені обчислення» вивчається у 6-му семестрі і є ключовою для оволодіння знаннями та навичками, які готують до професійної діяльності з планування, проектування та розробки паралельних і розподілених програм. Курс передбачає базову підготовку фахівця з паралельного програмування та розподілених обчислень, задаючи траєкторію подальшого застосування отриманих навичок в інших аспектах спеціальності, визначених у курсі бакалавра з комп’ютерної інженерії.

Метою першого модуля даної дисципліни є надання теоретичних знань і базових практичних навичок у створенні паралельних програм за допомогою поширених спеціалізованих бібліотек.

В результаті виконання усіх лабораторних робіт першого модуля студенти повинні:

Знати:

– переваги та недоліки застосування паралельних обчислень;

– випадки доцільності заміни непаралельного коду на паралельний;

– базовий синтаксис технологій CUDA, OpenCL, MPI, OpenMP.

Вміти:

– встановлювати бібліотеки для роботи з CUDA, OpenCL, MPI, OpenMP на різні платформи, оновлювати драйвери та налаштовувати середовища розробки;

– модифіковувати непаралельний код в паралельний;

– оперувати пам’яттю відеокарти та оперативною пам’яттю;

– командно розробляти паралельні програми;

– аналізувати характеристики обчислювальних пристроїв за критеріями можливості і доцільності розпаралелення;

– вдало комбінувати кілька технологій високопродуктивних обчислень;

– програмувати відповідно до принципів Coding Conventions.

Лабораторна робота №1
Високопродуктивні обчислення на відеокартах

Мета

1. Навчитись досліджувати апаратне забезпечення на його сумісність з різними технологіями високопродуктивних обчислень на відеокартах.
2. Навчитись писати порівняно прості паралельні програми із використанням технологій високопродуктивних обчислень на відеокартах.

1.1. Вступ до технологій OpenCL та CUDA

OpenCL (Open Computing Language – відкрита мова розрахунків) – це фреймворк, який дозволяє проводити розпаралелювання обчислень на графічних та центральних процесорах; на FPGA.

У фреймворк OpenCL входять мова програмування, яка базується на стандарті C99 та прикладний програмний інтерфейс (API). OpenCL забезпечує паралельність на рівні інструкцій та на рівні даних і є реалізацією концепції GPGPU. OpenCL – повністю відкритий стандарт, його використання доступне на базі вільних ліцензій.

Мета OpenCL полягає в тому, щоб доповнити OpenGL і OpenAL, які є відкритими галузевими стандартами для тривимірної комп’ютерної графіки і звуку, користуючись можливостями GPU. OpenCL розроблявся і підтримується некомерційним консорціумом Khronos Group, в який входять багато великих компаній, включаючи Apple, AMD, ARM, Intel, NVIDIA, Qualcomm, Sun Microsystems, Sony Computer Entertainment та інші.

CUDA (Compute Unified Device Architecture) – програмно-апаратна архітектура паралельних обчислень, яка дозволяє істотно збільшити обчислювальну

продуктивність завдяки використанню графічних процесорів фірми NVIDIA.

CUDA SDK надає можливість включати в текст програм на C виклик підпрограм, що виконуються на графічних процесорах NVIDIA. Це реалізовано шляхом команд, які записуються на особливому діалекті C. Архітектура CUDA дає розробнику можливість на свій розсуд організовувати доступ до набору інструкцій графічного прискорювача й керувати його пам'яттю.

Для перегляду відео характеристик комп'ютера рекомендується використання безкоштовного програмного забезпечення GPU Caps Viewer, яке можна завантажити за посиланням: https://www.geeks3d.com/dl/show/710.

1.2. Забезпечення можливостей програмування на технології OpenCL у Microsoft Visual Studio при наявній відеокарті від компанії NVIDIA

Встановлення OpenCL

Дізнатись чи підтримує Ваш драйвер технологію OpenCL можна за допомогою програми GPU Caps Viewer (рис. 1.1), а також в налаштуваннях драйвера. В останньому випадку, ознакою підтримки є наявність файла виду OpenCL.dll (рис. 1.2). Якщо відсутня підтримка, то варто встановити більш сучасну версію драйвера. Якщо й остання версія драйвера не підтримує OpenCL, то доцільно придбати більш сучасну відеокарту або виконувати завдання за іншим комп'ютером чи в хмарному середовищі.

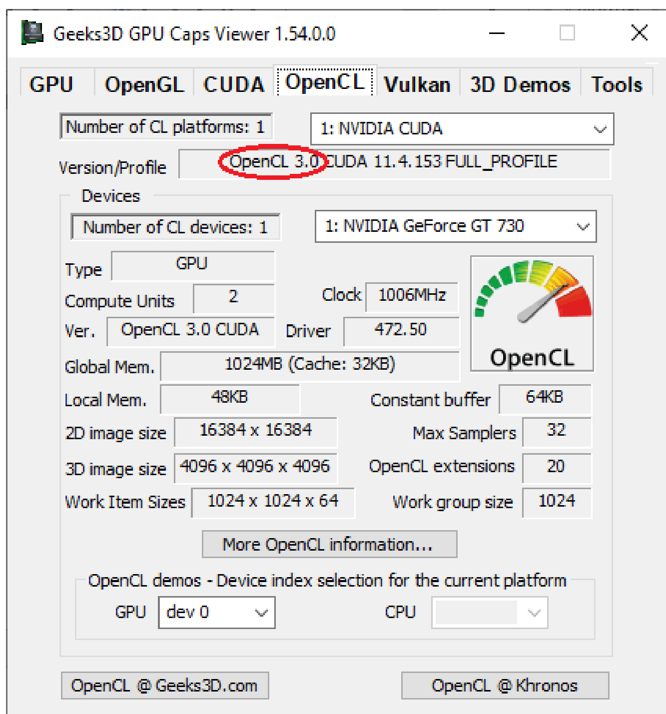

Рис. 1.1. Приклад вінка програми GPU Caps Viewer у випадку підтримки технології OpenCL (наявність версії OpenCL свідчить про підтримку)

Якщо наявна версія драйвера підтримує використання OpenCL, то наступним кроком є встановлення CUDA-Toolkit з офіційного сайту NVIDIA. Проте варто спершу визначити які саме версії CUDA-Toolkit підтримує Ваш драйвер. Тому варто діяти наступним чином:

1. Дізнатись версію драйвера відеокарти, наприклад, з використанням посилання: https://www.nvidia.com/download/index.aspx.

2. Перейти за посиланням: https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html та визначити

підходящу версію CUDA-Toolkit (див. табл. 2 або табл. 3 на веб-сторінці).

3. Встановити CUDA-Toolkit: https://developer.nvidia.com/cuda-toolkit-archive.

4. Перезавантажити комп'ютер для збереження нових системних шляхів.

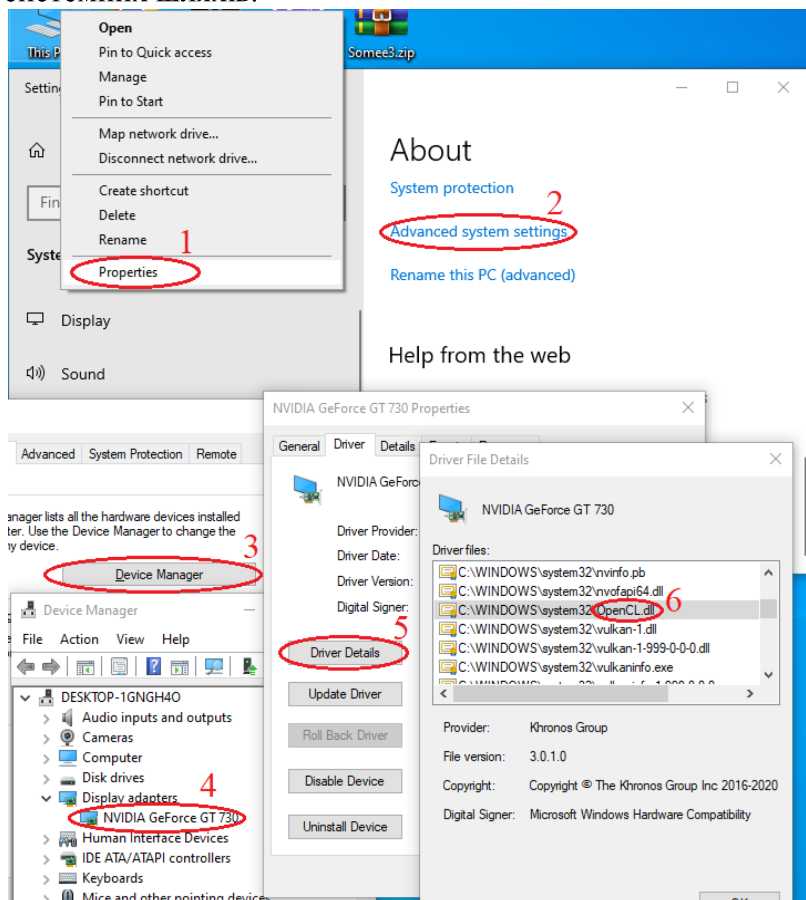

Рис. 1.2. Типова послідовність кроків для перевірки підтримуваності технології OpenCL (про підтримку свідчить наявність файлу OpenCL.dll)

На старіших версіях драйверів, окрім CUDA-Toolkit, можливо знадобиться встановлення додаткових компонентів «вручну». У таких випадках доцільною є додаткова консультація з викладачем.

Створення першого проекту на OpenCL

Зручною є розробка паралельних програм саме на Microsoft Visual Studio. Для створення і подальшого запуску проекту на OpenCL для NVIDIA-відеокарт під керуванням 64-розрядної операційної системи Windows, потрібно лише створити порожній консольний проект С++. Якщо ж мають місце проблеми із доступом до бібліотек, то доведеться здійснити наступну послідовність кроків:

1. Створити порожній консольний проект С++.
2. У властивостях створеного проекта за розташуванням C/C++ → *General* → *Additional Include Directories* додати адресу розміщення заголовних файлів виду: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\include\ (рис. 1.3). Дана папка з’являється в результаті встановлення CUDA-Toolkit.
3. У розташування *Linker* → *General* → *Additional Library Directories* варто добавити шлях виду C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\lib\x64 (рис. 1.4).
4. За розташуванням *Linker* → *Input* → *Additional Dependencies* варто добавити файл виду OpenCL.lib (рис. 1.5).

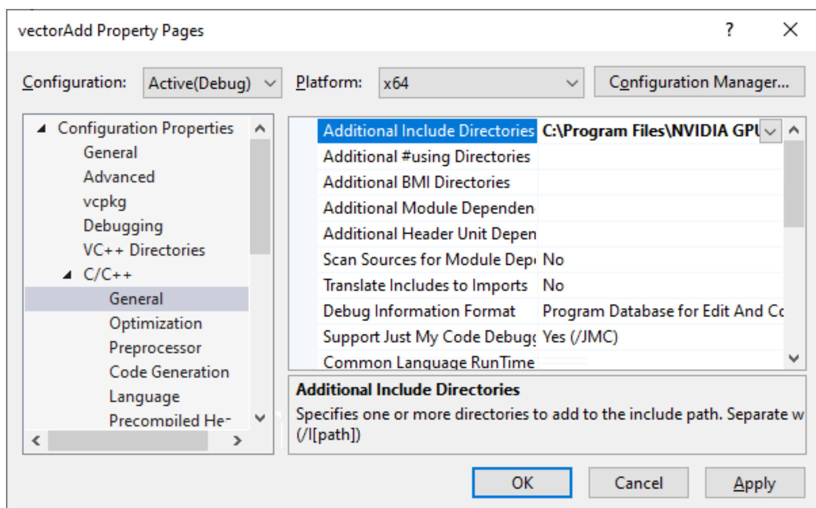

Рис. 1.3. Приклад додавання адреси розташування заголовних файлів OpenCL

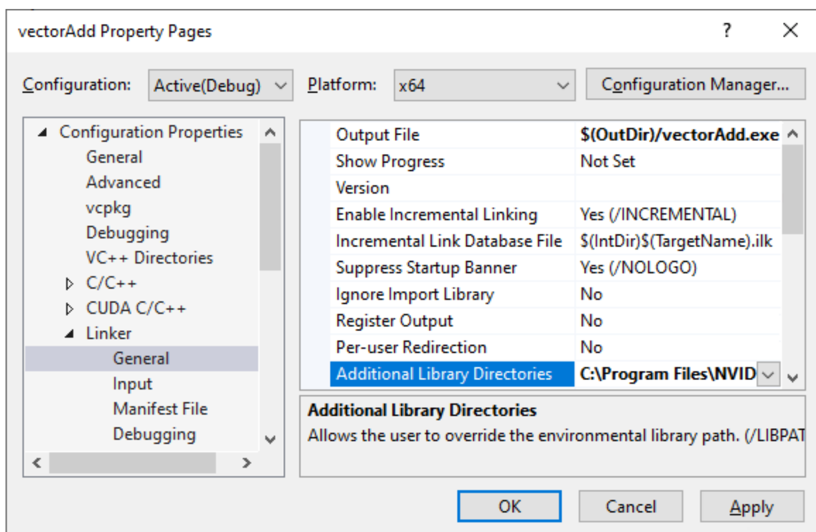

Рис. 1.4. Приклад додавання шляху розташування файлів-бібліотек OpenCL

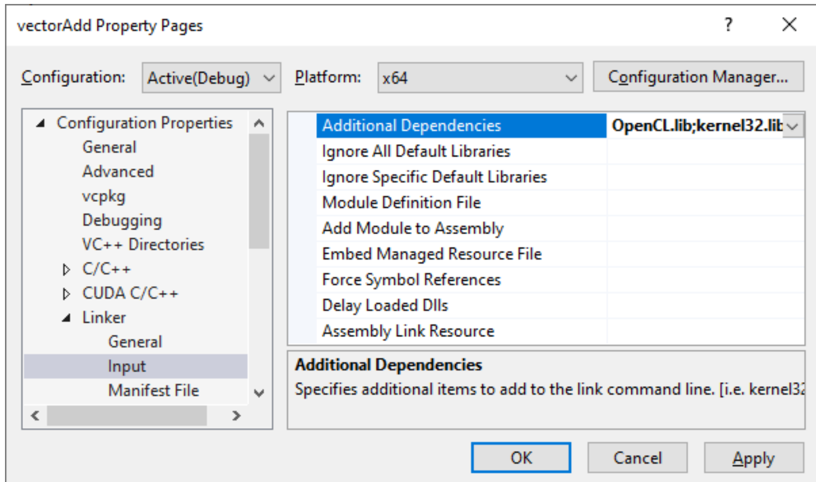

Рис. 1.5. Приклад додавання бібліотеки OpenCL.lib у якості додаткової залежності

Залишається створити 2 файли:

1. Файл ядра: vector_add_kernel.cl з кодом:

```
1.  __kernel void vector_add(__global const int* A,
        __global const int* B, __global int* C)
1.  {
2.    //Отримати індекс поточного елемента для обробки
3.    int i = get_global_id(0);
4.    // Додавання векторів
5.    C[i] = A[i] + B[i];
6.  }
```

2. Файл виду *.cpp:

```
1.  #define _CRT_SECURE_NO_WARNINGS
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.
5.  #pragma comment(lib, "OpenCL.lib");
6.
7.  #ifdef __APPLE__
8.    #include <OpenCL/opencl.h>
9.  #else
10.   #include <CL/cl.h>
```

```
11. #endif
12. #define MAX_SOURCE_SIZE (0x100000)
13.
14. int main(void)
15. {
16.   // створення двох векторів
17.   int i;
18.   const int LIST_SIZE = 1024;
19.   int* A = (int*)malloc(sizeof(int) * LIST_SIZE);
20.   int* B = (int*)malloc(sizeof(int) * LIST_SIZE);
21.   for (i = 0; i < LIST_SIZE; i++)
22.   {
23.     A[i] = i;
24.     B[i] = LIST_SIZE - i;
25.   }
26.   // читання вихідного коду ядра з
        vector_add_kernel.cl
27.   FILE* fp;
28.   char* source_str;
29.   size_t source_size;
30.   fp = fopen("vector_add_kernel.cl", "r");
31.   if (!fp)
32.   {
33.     fprintf(stderr, "Failed to load kernel.\n");
34.     exit(1);
35.   }
36.   source_str = (char*)malloc(MAX_SOURCE_SIZE);
37.   source_size = fread(source_str, 1,
        MAX_SOURCE_SIZE, fp);
38.   fclose(fp);
39.   // Отримання інформації про платформи і пристрої
40.   cl_platform_id platform_id = NULL;
41.   cl_device_id device_id = NULL;
42.   cl_uint ret_num_devices;
43.   cl_uint ret_num_platforms;
44.   cl_int ret = clGetPlatformIDs(1, &platform_id,
        &ret_num_platforms);
45.   ret = clGetDeviceIDs(platform_id,
        CL_DEVICE_TYPE_GPU, 1, &device_id,
        &ret_num_devices);
46.   // Створення OpenCL контексту
```

```
47.   cl_context context = clCreateContext(NULL, 1,
        &device_id, NULL, NULL, &ret);
48.   // Створення черги команд
49.   cl_command_queue command_queue =
        clCreateCommandQueue(context, device_id, 0,
        &ret);
50.   // Створення буферів пам’яті на пристрої для
        кожного вектора
51.   cl_mem a_mem_obj = clCreateBuffer(context,
52.     CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int),
        NULL, &ret);
53.   cl_mem b_mem_obj = clCreateBuffer(context,
54.     CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int),
        NULL, &ret);
55.   cl_mem c_mem_obj = clCreateBuffer(context,
56.     CL_MEM_WRITE_ONLY, LIST_SIZE * sizeof(int),
        NULL, &ret);
57.   // Копіювання векторів у буфери пам’яті
58.   ret = clEnqueueWriteBuffer(command_queue,
59.     a_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
        A, 0, NULL, NULL);
60.   ret = clEnqueueWriteBuffer(command_queue,
61.     b_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
        B, 0, NULL, NULL);
62.   // Створення  програми з вихідного коду ядра
63.   cl_program program =
        clCreateProgramWithSource(context, 1,
64.     (const char**)&source_str, (const
        size_t*)&source_size, &ret);
65.   // Створення виконуваного файлу
66.   ret = clBuildProgram(program, 1, &device_id,
        NULL, NULL, NULL);
67.   // Створення OpenCL ядра
68.   cl_kernel kernel = clCreateKernel(program,
        "vector_add", &ret);
69.   // Встановлення аргументів ядра
70.   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
        (void*)&a_mem_obj);
71.   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),
        (void*)&b_mem_obj);
72.   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),
        (void*)&c_mem_obj);
```

```
73.   // Виконання ядра
74.   size_t global_item_size = LIST_SIZE;
75.   size_t local_item_size = 64;
76.   ret = clEnqueueNDRangeKernel(command_queue,
77.     kernel, 1, NULL, &global_item_size,
78.     &local_item_size, 0, NULL, NULL);
79.   // Читання результату з пристрою в локальний
        список C
80.   int* C = (int*)malloc(sizeof(int) * LIST_SIZE);
81.   ret = clEnqueueReadBuffer(command_queue,
82.     c_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
        C, 0, NULL, NULL);
83.   ret = clFlush(command_queue);
84.   ret = clFinish(command_queue);
85.   ret = clReleaseKernel(kernel);
86.   ret = clReleaseProgram(program);
87.   ret = clReleaseMemObject(a_mem_obj);
88.   ret = clReleaseMemObject(b_mem_obj);
89.   ret = clReleaseMemObject(c_mem_obj);
90.   ret = clReleaseCommandQueue(command_queue);
91.   ret = clReleaseContext(context);
92.   free(A);
93.   free(B);
94.   free(C);
95.   return 0;
96. }
```

Якщо ще добавити функції виведення, то останні рядки результату виконання програми можуть мати вигляд, наведений на рис. 1.6.

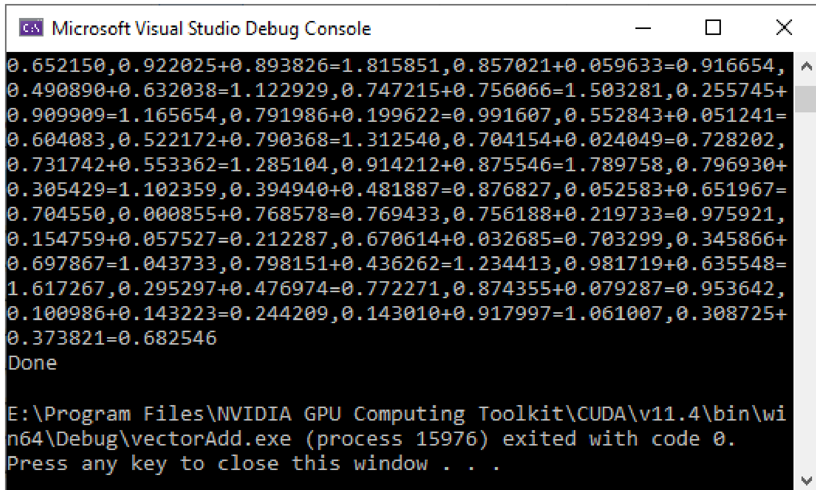

Рис. 1.6. Приклад виведення останніх рядків у консольному вікні

Додатково

Можливими є випадки, коли код працює коректно, але немає підсвічування синтаксису для OpenCL. Ця проблема вирішується наступним чином:

1. Обираєте пункт меню Tools → Options.
2. У переліку пунктів знаходите Text Editor → File Extension.
3. Друкуєте *cl* в полі Extension; *Microsoft Visual C++* – в полі Editor.
4. Натискаєте Add.
5. Натискаєте OK.
6. Перезавантажуєте середовище Microsoft Visual Studio.

1.3. Забезпечення можливостей програмування на технології CUDA у Microsoft Visual Studio при наявній відеокарті від компанії NVIDIA

Встановлення CUDA

Дізнатись чи підтримує Ваш драйвер технологію CUDA можна за допомогою програми GPU Caps Viewer (рис. 1.7), а також в налаштуваннях драйвера. В останньому

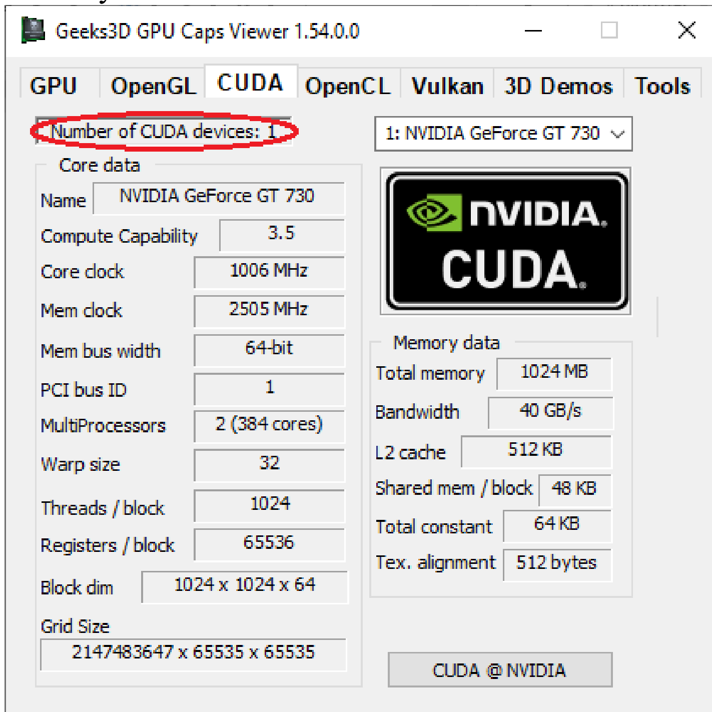

Рис. 1.7. Приклад вінка програми GPU Caps Viewer у випадку підтримки технології CUDA (кількість пристроїв більша за 0, що свідчить про підтримку CUDA)

випадку, ознакою підтримки є наявність файла виду nvcuda.dll (рис. 1.8). Якщо відсутня підтримка, то варто

встановити більш сучасну версію драйвера. Якщо й остання версія драйвера не підтримує CUDA, то доцільно придбати більш сучасну відеокарту або виконувати завдання за іншим комп'ютером чи в хмарному середовищі.

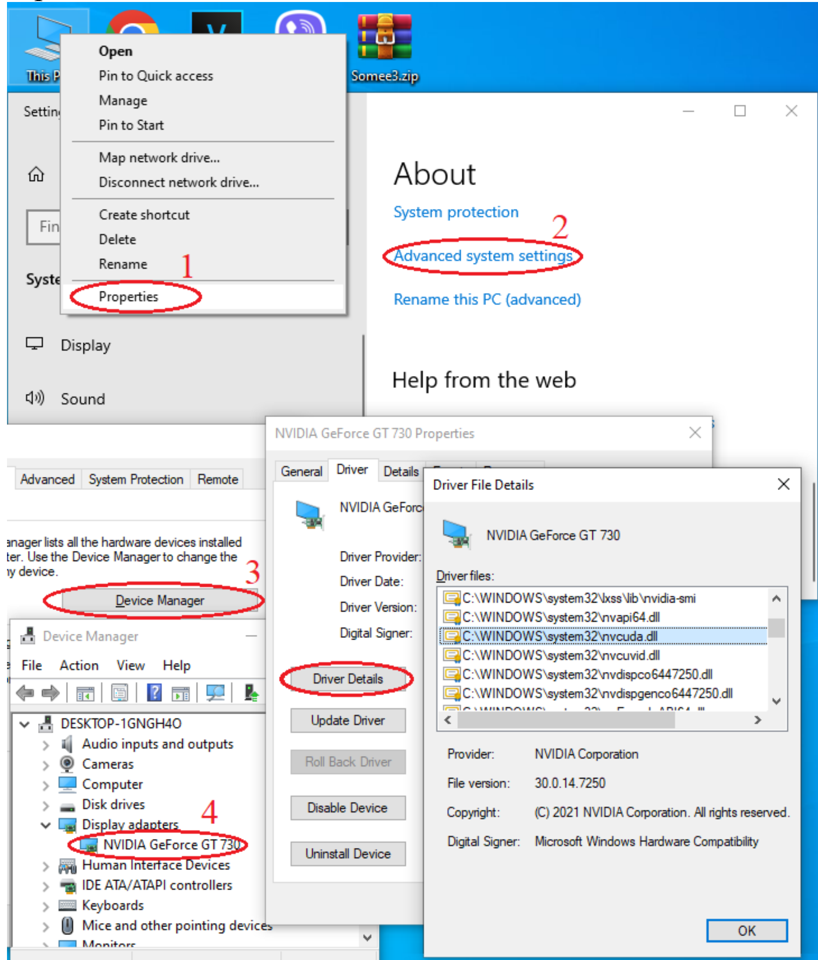

Рис. 1.8. Типова послідовність кроків для перевірки підтримки технології CUDA (у вікні Driver File Details відмічено рядок, наявність якого є ознакою підтримки)

Для забезпечення можливості програмувати на технології CUDA потрібно виконати наступну послідовність дій:

1. Обрати актуальну версію драйвера використовуваної відеокарти за посиланням: https://www.nvidia.com/download/index.aspx та натиснути Search. Після цього Вас перемістить на сторінку виду рис. 1.9. Тут важливим для подальших дій є встановлення актуальної версії драйвера.

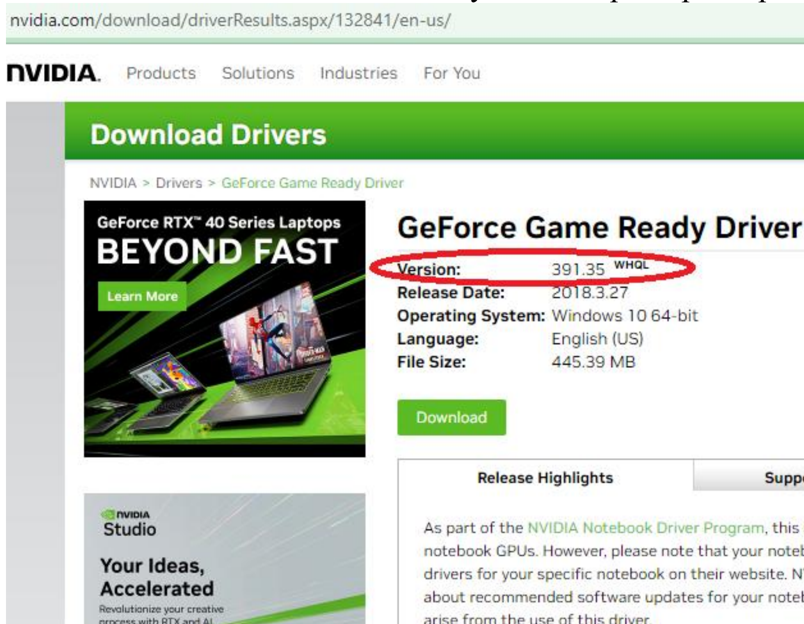

Рис. 1.9. Вікно із позначенням останньої версії деякого драйвера компанії NVIDIA

2. Співставити версію драйвера із таблицею 2 або 3 за посиланням: https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html та визначити сумісну версію CUDA-Toolkit.

3. Встановити знайдений драйвер і CUDA-Toolkit. CUDA-Toolkit можна встановити, перейшовши за посиланням https://developer.nvidia.com/cuda-toolkit-archive.

4. Перезавантажити комп’ютер.

Створення першого проекту на CUDA

Зручною є розробка паралельних програм на Microsoft Visual Studio. Для створення проекту на CUDA під керуванням 64-розрядної операційної системи Windows, варто виконати наступну послідовність кроків:

1. Запустити Microsoft Visual Studio.
2. Знайти серед шаблонів проектів шаблон, який містить слово: CUDA.
3. Створити проект.

Якщо ж шаблон CUDA відсутній, то доведеться здійснити наступну послідовність кроків:

1. Створити порожній консольний проект C++.
2. У властивостях створеного проекта за розташуванням C/C++ → *General* → *Additional Include Directories* додати адресу розташування заголовних файлів виду: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\include\ (рис. 1.10). Дана папка з’являється в результаті встановлення CUDA-Toolkit.
3. У розташування *Linker* → *General* → *Additional Library Directories* варто добавити шлях виду C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\lib\x64 (рис. 1.11).
4. У розташування *Linker* → *Input* → *Additional Dependencies* варто добавити файл виду CUDA.lib (рис. 1.12).

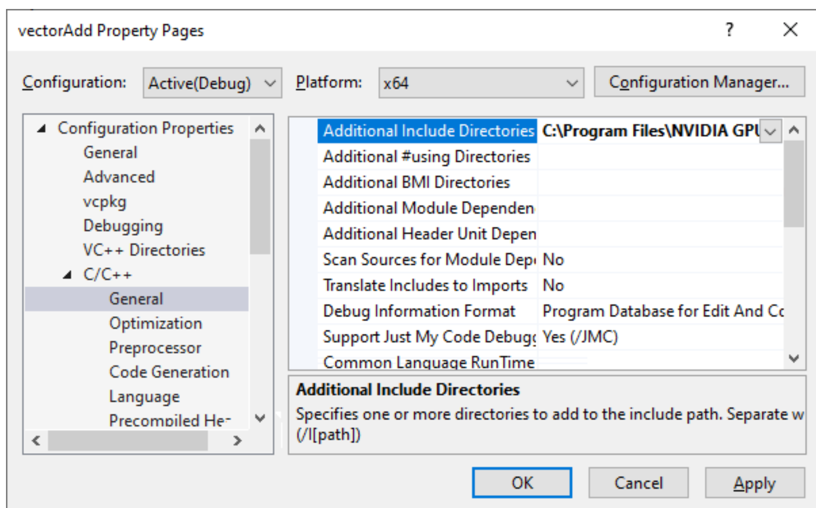

Рис. 1.10. Приклад додавання адреси розташування заголовних файлів CUDA

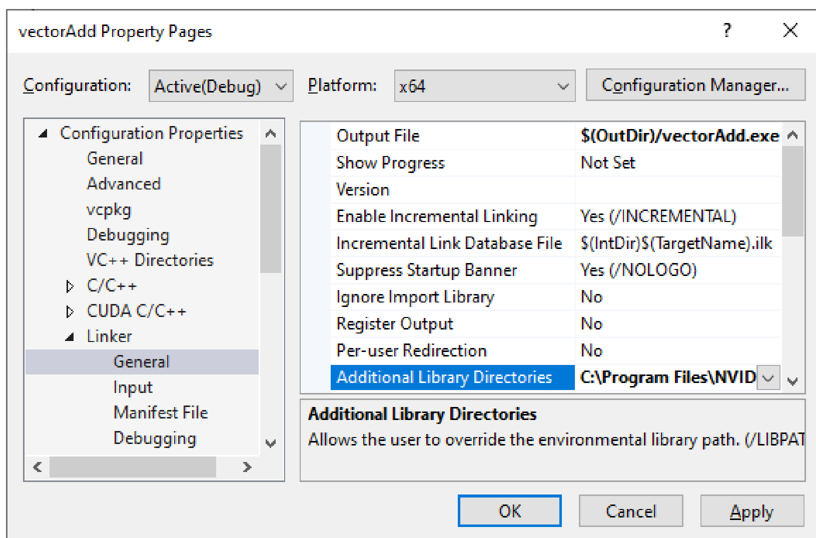

Рис. 1.11. Приклад додавання шляху розташування файлів-бібліотек CUDA

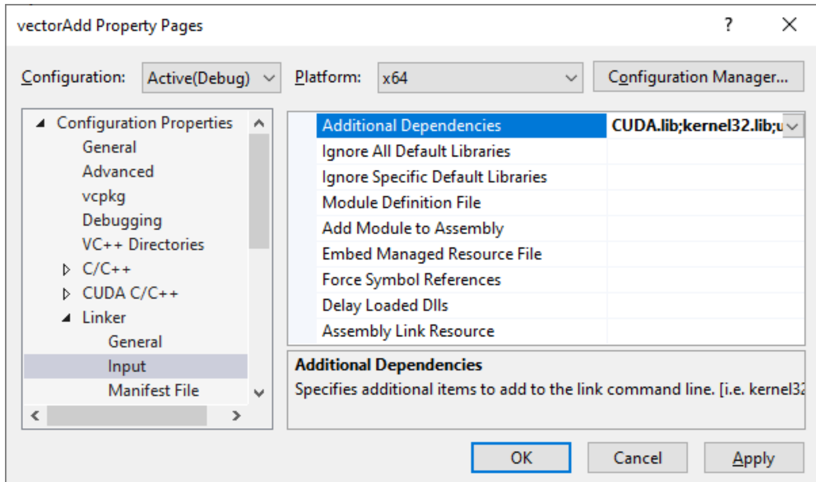

Рис. 1.12. Приклад додавання бібліотеки CUDA.lib у якості додаткової залежності

Залишається створити 1 файл:

```
1.  __global__ void vectorAdd(const float *A, const
       float *B, float *C, int numElements)
2.  {
3.    int i = blockDim.x * blockIdx.x + threadIdx.x;
4.    if (i < numElements)
5.    {
6.      C[i] = A[i] + B[i];
7.    }
8.  }
9.  int main()
10. {
11.   cudaError_t err = cudaSuccess;
12.   int numElements = 50000000;
13.   size_t size = numElements * sizeof(float);
14.   printf("[Vector addition of %d elements]\n",
       numElements);
15.
16.   float *h_A = (float *)malloc(size);
17.   float *h_B = (float *)malloc(size);
18.   float *h_C = (float *)malloc(size);
19.
```

```
20.   if (h_A == NULL || h_B == NULL || h_C == NULL)
21.     exit(EXIT_FAILURE);
22.   for (int i = 0; i < numElements; ++i)
23.   {
24.     h_A[i] = rand()/(float)RAND_MAX;
25.     h_B[i] = rand()/(float)RAND_MAX;
26.   }
27.
28.   float *d_A = NULL;
29.   float *d_B = NULL;
30.   float *d_C = NULL;
31.
32.   err = cudaMalloc((void **)&d_A, size);
33.   if (err != cudaSuccess)
34.     exit(EXIT_FAILURE);
35.   err = cudaMalloc((void **)&d_B, size);
36.   if (err != cudaSuccess)
37.     exit(EXIT_FAILURE);
38.   err = cudaMalloc((void **)&d_C, size);
39.   if (err != cudaSuccess)
40.     exit(EXIT_FAILURE);
41.
42.   printf("Copy input data from the host memory to
        the CUDA device\n");
43.   err = cudaMemcpy(d_A, h_A, size,
        cudaMemcpyHostToDevice);
44.   if (err != cudaSuccess)
45.     exit(EXIT_FAILURE);
46.   err = cudaMemcpy(d_B, h_B, size,
        cudaMemcpyHostToDevice);
47.   if (err != cudaSuccess)
48.     exit(EXIT_FAILURE);
49.
50.   int threadsPerBlock = 256;
51.   int blocksPerGrid = (numElements +
        threadsPerBlock - 1) / threadsPerBlock;
52.   printf("CUDA kernel launch with %d blocks of %d
        threads\n", blocksPerGrid, threadsPerBlock);
53.   vectorAdd<<<blocksPerGrid,
        threadsPerBlock>>>(d_A, d_B, d_C, numElements);
54.   err = cudaGetLastError();
55.   if (err != cudaSuccess)
```

```
56.     exit(EXIT_FAILURE);
57.
58.   printf("Copy output data from the CUDA device to
        the host memory\n");
59.   err = cudaMemcpy(h_C, d_C, size,
        cudaMemcpyDeviceToHost);
60.   if (err != cudaSuccess)
61.     exit(EXIT_FAILURE);
62.
63.   err = cudaFree(d_A);
64.   if (err != cudaSuccess)
65.     exit(EXIT_FAILURE);
66.   err = cudaFree(d_B);
67.   if (err != cudaSuccess)
68.     exit(EXIT_FAILURE);
69.   err = cudaFree(d_C);
70.   if (err != cudaSuccess)
71.     exit(EXIT_FAILURE);
72.
73.   free(h_A);
74.   free(h_B);
75.   free(h_C);
76.
77.   printf("Done\n");
78.   return 0;
79. }
```

Якщо ще добавити функції виведення, то останні рядки результату виконання програми можуть мати вигляд, подібний до рис. 1.6.

Додатково

Можливими є випадки, коли код працює коректно, але немає підсвітки синтаксису для CUDA. Цю проблему можна вирішити наступним чином:

1. Обирати пункт меню Tools → Options.
2. У переліку пунктів знайти Text Editor → File Extension.
3. Надрукувати *cu* в полі Extension; обрати *Microsoft Visual C++* – в полі Editor.

4. Натиснути Add.
5. Натиснути OK.
6. Перезавантажити середовище Microsoft Visual Studio.

1.4. Забезпечення можливостей програмування на технології OpenCL у Microsoft Visual Studio при наявній відеокарті від компанії AMD

Встановлення OpenCL

Дізнатись чи підтримує Ваш драйвер технологію OpenCL можна за допомогою програми GPU Caps Viewer (рис. 1.1), а також в налаштуваннях драйвера. Ознакою підтримки є наявність файла виду OpenCL.dll (рис. 1.2). Якщо даного файла не існує, то варто встановити більш сучасну версію драйвера. Якщо й остання версія драйвера не підтримує OpenCL, то доцільно придбати більш сучасну відеокарту або виконувати завдання за іншим комп’ютером чи в хмарному середовищі.

Якщо встановлена версія драйвера підтримує використання OpenCL, то наступним кроком є скачування SDK одним із наступних способів (рекомендується спосіб 1):

1. За допомогою vcpkg.
   1.1. Встановити Git for Windows (https://gitforwindows.org/). Перезавантажити комп’ютер.
   1.2. За допомогою командного рядка, запущеного в режимі адміністратора, послідовно виконати наступне:
   - cd c:\Program Files
   - git clone https://github.com/Microsoft/vcpkg.git
   - .\vcpkg\bootstrap-vcpkg.bat
   - .\vcpkg install opencl:x64-windows
   - .\vcpkg integrate install

   1.3. Перезавантажити комп’ютер.

2. Скачати OpenSource проект з Github: https://github.com/GPUOpen-LibrariesAndSDKs/OCL-SDK/releases.

Створення першого проекту на OpenCL

Зручною є розробка паралельних програм на Microsoft Visual Studio. Для запуску проекту на OpenCL для AMD-відеокарт під керуванням 64-розрядної операційної системи Windows, потрібно створити порожній консольний проект C++. Якщо ж виникатимуть помилки, пов'язані із неможливістю доступитись до OpenCL-бібліотек, то доведеться здійснити наступну послідовність кроків:

1. Створити порожній консольний проект C++.
2. У властивостях створеного проекта за розташуванням *C/C++ → General → Additional Include Directories* додати адресу розташування заголовних файлів виду: C:\Users\Roma\Downloads\lightOCLSDK\include\. Дана папка з'являється після розархівовування SDK, скачаного з Github.
3. У розташування *Linker → General → Additional Library Directories* варто добавити шлях виду C:\Users\Roma\Downloads\lightOCLSDK\lib\x86_64.
4. У розташування *Linker → Input → Additional Dependencies* варто добавити файл виду OpenCL.lib.

Залишається створити 2 файли:

1. Файл ядра: vector_add_kernel.cl з кодом

```
1.  __kernel void vector_add(__global const int* A,
        __global const int* B, __global int* C)
2.  {
3.    //Отримати індекс поточного елемента для обробки
4.    int i = get_global_id(0);
5.    // Додавання векторів
6.    C[i] = A[i] + B[i];
7.  }
```

2.Файл виду *.cpp:

```
1. #define _CRT_SECURE_NO_WARNINGS
2. #include <stdio.h>
3. #include <stdlib.h>
4.
5. #pragma comment(lib, "OpenCL.lib");
6.
7. #ifdef __APPLE__
8.   #include <OpenCL/opencl.h>
9. #else
10.   #include <CL/cl.h>
11. #endif
12. #define MAX_SOURCE_SIZE (0x100000)
13.
14. int main(void)
15. {
16.   // створення двох векторів
17.   int i;
18.   const int LIST_SIZE = 100000000 / 2;
19.   int* A = (int*)malloc(sizeof(int) *   LIST_SIZE);
20.   int* B = (int*)malloc(sizeof(int) *   LIST_SIZE);
21.   for (i = 0; i < LIST_SIZE; i++)
22.   {
23.     A[i] = i;
24.     B[i] = LIST_SIZE - i;
25.   }
26.   // читання вихідного коду ядра з
        vector_add_kernel.cl
27.   FILE* fp;
28.   char* source_str;
29.   size_t source_size;
30.   fp = fopen("vector_add_kernel.cl", "r");
31.   if (!fp)
32.   {
33.     fprintf(stderr, "Failed to load kernel.\n");
34.     exit(1);
35.   }
36.   source_str = (char*)malloc(MAX_SOURCE_SIZE);
37.   source_size = fread(source_str, 1,
        MAX_SOURCE_SIZE, fp);
38.   fclose(fp);
39.   // Отримання інформації про платформи і пристрої
```

```
40.  cl_platform_id platform_id = NULL;
41.  cl_device_id device_id = NULL;
42.  cl_uint ret_num_devices;
43.  cl_uint ret_num_platforms;
44.  cl_int ret = clGetPlatformIDs(1, &platform_id,
       &ret_num_platforms);
45.  ret = clGetDeviceIDs(platform_id,
       CL_DEVICE_TYPE_GPU, 1, &device_id,
       &ret_num_devices);
46.  // Створення OpenCL контексту
47.  cl_context context = clCreateContext(NULL, 1,
       &device_id, NULL, NULL, &ret);
48.  // Створення черги команд
49.  cl_command_queue command_queue =
       clCreateCommandQueue(context, device_id, 0,
       &ret);
50.  // Створення буферів пам’яті на пристрої для
       кожного вектора
51.  cl_mem a_mem_obj = clCreateBuffer(context,
52.    CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int),
       NULL, &ret);
53.  cl_mem b_mem_obj = clCreateBuffer(context,
54.    CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int),
       NULL, &ret);
55.  cl_mem c_mem_obj = clCreateBuffer(context,
56.    CL_MEM_WRITE_ONLY, LIST_SIZE * sizeof(int),
       NULL, &ret);
57.  // Копіювання векторів у буфери пам’яті
58.  ret = clEnqueueWriteBuffer(command_queue,
59.    a_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
       A, 0, NULL, NULL);
60.  ret = clEnqueueWriteBuffer(command_queue,
61.    b_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
       B, 0, NULL, NULL);
62.  // Створення  програми з вихідного коду ядра
63.  cl_program program =
       clCreateProgramWithSource(context, 1,
64.  (const char**)&source_str, (const
       size_t*)&source_size, &ret);
65.  // Створення виконуваного файлу
66.  ret = clBuildProgram(program, 1, &device_id,
       NULL, NULL, NULL);
```

```
67.   // Створення OpenCL ядра
68.   cl_kernel kernel = clCreateKernel(program,
        "vector_add", &ret);
69.   // Встановлення аргументів ядра
70.   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
        (void*)&a_mem_obj);
71.   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),
        (void*)&b_mem_obj);
72.   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),
        (void*)&c_mem_obj);
73.   // Виконання ядра
74.   size_t global_item_size = LIST_SIZE;
75.   size_t local_item_size = 64;
76.   ret = clEnqueueNDRangeKernel(command_queue,
77.     kernel, 1, NULL, &global_item_size,
78.     &local_item_size, 0, NULL, NULL);
79.   // Читання результату з пристрою в локальний
        список C
80.   int* C = (int*)malloc(sizeof(int) * LIST_SIZE);
81.   ret = clEnqueueReadBuffer(command_queue,
82.     c_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
        C, 0, NULL, NULL);
83.   ret = clFlush(command_queue);
84.   ret = clFinish(command_queue);
85.   ret = clReleaseKernel(kernel);
86.   ret = clReleaseProgram(program);
87.   ret = clReleaseMemObject(a_mem_obj);
88.   ret = clReleaseMemObject(b_mem_obj);
89.   ret = clReleaseMemObject(c_mem_obj);
90.   ret = clReleaseCommandQueue(command_queue);
91.   ret = clReleaseContext(context);
92.   free(A);
93.   free(B);
94.   free(C);
95.   return 0;
96. }
```

Якщо ще добавити функції виведення, то останні рядки результату виконання програми можуть мати вигляд, зображений на рис. 1.13.

```
Microsoft Visual Studio Debug Console
49999991 + 9 = 50000000
49999992 + 8 = 50000000
49999993 + 7 = 50000000
49999994 + 6 = 50000000
49999995 + 5 = 50000000
49999996 + 4 = 50000000
49999997 + 3 = 50000000
49999998 + 2 = 50000000
49999999 + 1 = 50000000

E:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.4\bin\wi
n64\Debug\vectorAdd.exe (process 12284) exited with code 0.
To automatically close the console when debugging stops, enable
 Tools->Options->Debugging->Automatically close the console whe
n debugging stops.
Press any key to close this window . . .
```

Рис. 1.13. Приклад останніх рядків вікна виведення результату додавання двох векторів

Додатково

Можливими є випадки, коли код працює коректно, але немає підсвітки синтаксису для OpenCL. Цю проблемо можна вирішити наступним чином:

1. Вибрати пункт меню Tools → Options.
2. У переліку пунктів знайти Text Editor → File Extension.
3. Ввести *cl* в полі Extension; обрати *Microsoft Visual C++* в полі Editor.
4. Натиснути Add.
5. Натиснути OK.
6. Перезавантажити середовище Microsoft Visual Studio.

1.5. Завдання

1. *(20% балів)* Налаштуйте можливість працювати з OpenCL або CUDA на Вашому комп'ютері/ноутбуці.
2. *(10% балів)* Створіть 2 окремі функції-ядра, кожне з яких працювало б на власну операцію згідно варіанту. У випадку роботи з технологією OpenCL рекомендується створити відповідні 2 файли з розширенням *.cl.
3. *(10% балів)* Підберіть таку кількість елементів масивів, щоб по-максимуму дослідити можливості свого обчислювального пристрою.
4. *(50% балів)* Реалізуйте, щоб усі 2 ядра поокремо паралельно виконували операції згідно варіанту.
5. *(10% балів)* Програмно проведіть порівняння кількості часу, необхідного для проведення розрахунків паралельно/послідовно.
6. Проаналізуйте та поясніть отримані результати.

Варіанти до виконання поставленого завдання

Варіант 1. Додавання та віднімання.
Варіант 2. Додавання та множення.
Варіант 3. Додавання та ділення.
Варіант 4. Додавання та ділення націло.
Варіант 5. Додавання та ділення з остачею.
Варіант 6. Віднімання та множення.
Варіант 7. Віднімання та ділення.
Варіант 8. Віднімання та ділення націло.
Варіант 9. Віднімання та ділення з остачею.
Варіант 10. Множення та ділення.
Варіант 11. Множення та ділення націло.
Варіант 12. Множення та ділення з остачею.
Варіант 13. Ділення та ділення націло.
Варіант 14. Ділення та ділення з остачею.
Варіант 15. Ділення націло та ділення з остачею.

Додаткові матеріали

1. Вступне відео по основах CUDA та деякі приклади на мові Python: https://www.youtube.com/watch?v=r9IqwpMR9TE&list=PLqXS1b2lRpYTUHPp2MYkgXS7v6_qA-JsF.
2. Уроки по CUDA: https://www.youtube.com/watch?v=m0nhePeHwFs&list=PLKK11Ligqititws0ZOoGk3SW-TZCar4dK&index=1
3. Вступне відео по OpenCL: https://www.youtube.com/watch?v=V4RfPfHQPC8&list=PLiwt1iVUib9s6vyEqdpcgAq7NBRlp9mAY&index=1
4. Лекції по OpenCL: https://www.youtube.com/watch?v=8D6yhpiQVVI&list=PLDivN33Lbf6cLqZ5_i_k0KeMaQKEctMgS

Контрольні запитання

1. Для чого використовується кваліфікатор `__global__`?
2. Для чого призначений вираз `blockdim.x * blockidx.x + threadidx.x`?
3. Навіщо у ядрі вказувати вираз виду `if (i < numElements)`?
4. Яке серйозне обмеження має оператор `new` в порівнянні з функцією `malloc`?
5. Для чого доцільно використовувати оператор вибору одразу після застосування функції `malloc`?
6. Чим відрізняються функції `malloc` та `cudamalloc`?
7. Як працює функція `cudamemcpy`?
8. Яка суть змінних `threadsperblock` та `blockspergrid`?
9. Що означають кутові дужки у вирзі `func<<<blockspergrid, threadsperblock>>>(d_a, d_b, d_c, numElements)`?
10. Яка різниця між функціями `cudafree` і `free`?

11. Яка різниця між пристроєм та хостом у термінології CUDA?

12. Для чого використовується ключове слово __kernel?

13. Для чого призначена функція get_global_id(0)?

14. Яка різниця між функціями clGetPlatformIDs та clGetDeviceIDs?

15. Для чого призначена функція clCreateContext?

16. Для чого призначена функція clCreateCommandQueue?

17. Як працюють функції clEnqueueWriteBuffer і clEnqueueReadBuffer?

18. Для чого призначена функція clBuildProgram?

19. Для чого призначена функція clCreateKernel?

20. Для чого призначена функція clEnqueueNDRangeKernel?

Лабораторна робота №2
Високопродуктивні обчислення на процесорах

Мета

1. Навчитись створювати порівняно прості програми із використанням технологій високопродуктивних обчислень на ядрах центрального процесора.
2. Навчитись аналізувати та порівнювати різні технології високопродуктивних обчислень.

2.1. Вступ до технології MPI

Message Passing Interface (MPI, інтерфейс передачі повідомлень) – програмний інтерфейс (API) для передачі інформації, який дозволяє обмінюватися повідомленнями між процесами, що працюють на досягнення спільної мети. Іншими словами, розробники MPI запозичили ідею технології передачі повідомлень між парами (або й групою) абонентів та адаптували інтерфейс під паралельні обчислення.

MPI є найбільш поширеним стандартом інтерфейсу обміну даними в паралельному програмуванні; існують його реалізації для великої кількості комп'ютерних платформ. Використовується для розробки програм для кластерів і суперкомп'ютерів. Основним засобом комунікації між процесами MPI є передача повідомлень один одному.

2.2. Забезпечення можливостей програмування на технології MPI у Microsoft Visual Studio при наявній відеокарті від компанії AMD

Встановлення MPI

Для реалізації можливості роботи з технологією MPI на комп’ютері під керування операційної систем Windows доцільно виконати наступну послідовність кроків:

1. Встановити Microsoft MPI (https://www.microsoft.com/en-us/download/details.aspx?id=100593). Перезавантажити комп’ютер.
2. Встановити Git for Windows (https://gitforwindows.org/). Перезавантажити комп’ютер.
3. За допомогою командного рядка, запущеного в режимі адміністратора, послідовно виконати наступне:
   - cd c:\Program Files
   - git clone https://github.com/Microsoft/vcpkg.git
   - .\vcpkg\bootstrap-vcpkg.bat
   - .\vcpkg install msmpi:x64-windows
   - .\vcpkg integrate install
4. Перезавантажити комп’ютер.

Створення першого проекту на MPI

Зручною є розробка паралельних програм на Microsoft Visual Studio. Для створення проекту на MPI для 64-розрядної операційної системи Windows, потрібно лише створити порожній консольний проект С++ та ввести MPI-код. Якщо ж виникатимуть помилки, пов’язані із неможливістю доступитись до бібліотек MPI, то доведеться здійснити наступну послідовність кроків:

1. Створити порожній консольний проект С++.
2. У властивостях створеного проекта за розташуванням C/C++ → *General* → *Additional Include Directories* додати адресу розташування заголовних файлів виду: C:/Users/I-

HAVE-NO-TIME-for/vcpkg/packages/msmpi_x64-windows/include/ (рис. 2.1). Дана папка з’являється в результаті встановлення пекеджа msmpi.

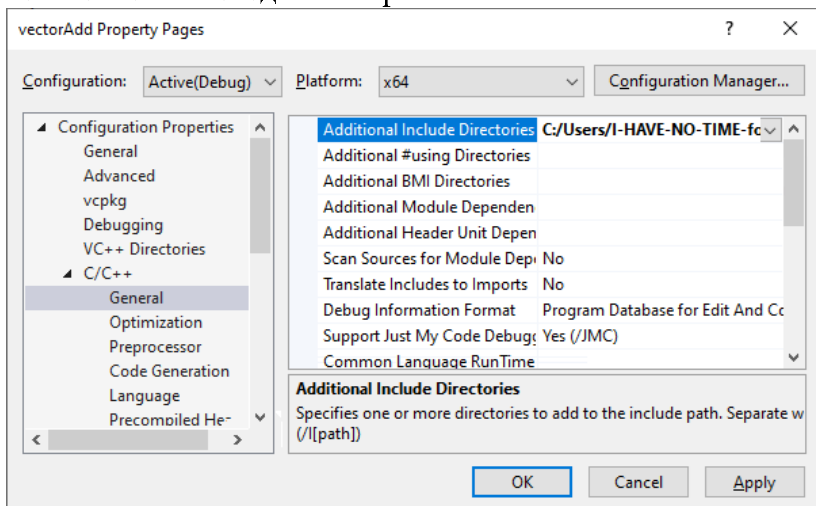

Рис. 2.1. Приклад додавання адреси розташування заголовних файлів MPI

Залишається помістити наступний код у файл формату *.c або *.cpp:

```
1.   #include "mpi.h"
2.   #include <stdio.h>
3.   #include <stdlib.h>
4.
5.   #define MASTER 0          //Створюємо синонім для
         рангу основного процесу
6.   #define ARRAY_SIZE 1024  //Задаємо розміри для
         масивів.
7.
8.   int main(int argc, char* argv[]) //Точка входу в
         паралельну програму
9.   {
10.
11.    // Створюємо вказівники на адреси, де
         міститимуться масиви
```

```
12.   int* a = NULL;
13.   int* b = NULL;
14.   int* c = NULL;
15.
16.   int total_proc;   // загальна кількість процесів
17.   int rank;         // ранг кожного процесу
18.   int n_per_proc;   // елементів, виділених на
        процес
19.   int n = ARRAY_SIZE;
20.   int i;
21.
22.   MPI_Status status;   // використовується для
        перевірки помилок MPI.
23.
24.   // 1. Ініціалізація середовища MPI
25.   MPI_Init(&argc, &argv);
26.   MPI_Comm_size(MPI_COMM_WORLD, &total_proc);
27.   // 2. Присвоюємо в змінну total_proc загальну
        кількість процесів
28.   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
29.   // 3. Присвоюємо у змінну rank номер поточного
        процесу
30.
31.   // Вказівники на низку коротких масивів, які
        використовуватимуться для паралельних додавань
32.   int* ap;
33.   int* bp;
34.   int* cp;
35.
36.   // 4. В основному процесі ініціалізуються
        масиви, які використовуватимуться для
        додавання.
37.   if (rank == MASTER)
38.   {
39.     a = (int*)malloc(sizeof(int) * n);
40.     b = (int*)malloc(sizeof(int) * n);
41.     c = (int*)malloc(sizeof(int) * n);
42.
43.     for (i = 0; i < n; i++)
44.     {
45.       a[i] = i + 1;
46.     }
```

```
47.       for (i = 0; i < n; i++)
48.       {
49.         b[i] = i * 2;
50.       }
51.     }
52.
53.     // задаємо кількість пар елементів, які
          додаватимуться у кожному окремому процесі
54.     n_per_proc = n / total_proc;
55.
56.     // 5. Виділення пам’яті під вкорочені масиви
57.     ap = (int*)malloc(sizeof(int) * n_per_proc);
58.     bp = (int*)malloc(sizeof(int) * n_per_proc);
59.     cp = (int*)malloc(sizeof(int) * n_per_proc);
60.
61.     // 6. Рівномірне розподілення масиву  a  в
          укорочені масиви  ap
62.     MPI_Scatter(a, n_per_proc, MPI_INT, ap,
          n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);
63.     // Рівномірне розподілення масиву  b  в
          укорочені масиви  bp
64.     MPI_Scatter(b, n_per_proc, MPI_INT, bp,
          n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);
65.
66.     // 7. Попарне додавання елементів укорочених
          масивів
67.     for (i = 0; i < n_per_proc; i++)
68.     {
69.       cp[i] = ap[i] + bp[i];
70.     }
71.
72.     // 8. Збирання укорочених масивів  cp  в єдиний
          масив  c
73.     MPI_Gather(cp, n_per_proc, MPI_INT, c,
          n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);
74.
75.     // Виведення результатів в основному процесі
76.     if (rank == MASTER)
77.     {
78.       for (i = 0; i < n; i++)
79.       {
```

```
80.        printf("%lld + %lld = %lld\n", a[i], b[i],
          c[i]);
81.       }
82.     }
83.
84.     // Звільнення пам’яті
85.     if (rank == MASTER)
86.     {
87.       free(a);
88.       free(b);
89.       free(c);
90.     }
91.     free(ap);
92.     free(bp);
93.     free(cp);
94.
95.     // 9. Завершення процесів і середовища MPI
96.     MPI_Finalize();
97.
98.     return 0;
99.  }
```

Приклад виведення останніх рядків результату виконання програми зображений на рис. 2.2.

Якщо здійснено запуск наведеного коду з середовища Microsoft Visual Studio, то варто розуміти, що розрахунки виконуються лише на одному ядрі процесора. Для справжнього розпаралелення доцільно скористатись командним рядком Вашої операційної системи (звичайно ж, у режимі адміністратора). Для зручності, дещо модифікувавши вихідний код, за умови, що файл mpiexec знаходиться у папці C:\Program Files\Microsoft MPI\Bin, а відкомпільований виконуваний файл C++ проекту – у C:\Users\I-HAVE-NO-TIME-for\source\repos\MPI\x64\Debug \mpi, вікно командного рядка може мати вигляд, зображений на рис. 2.3.

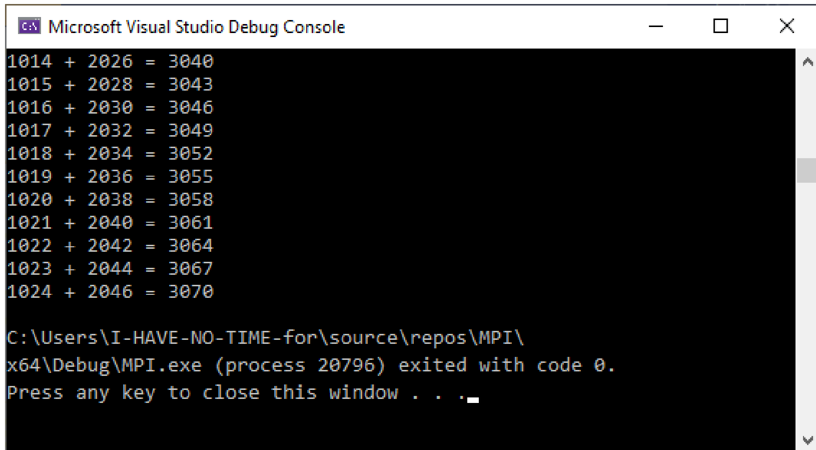

Рис. 2.2. Консольне вікно виведення результату роботи програми додавання двох векторів

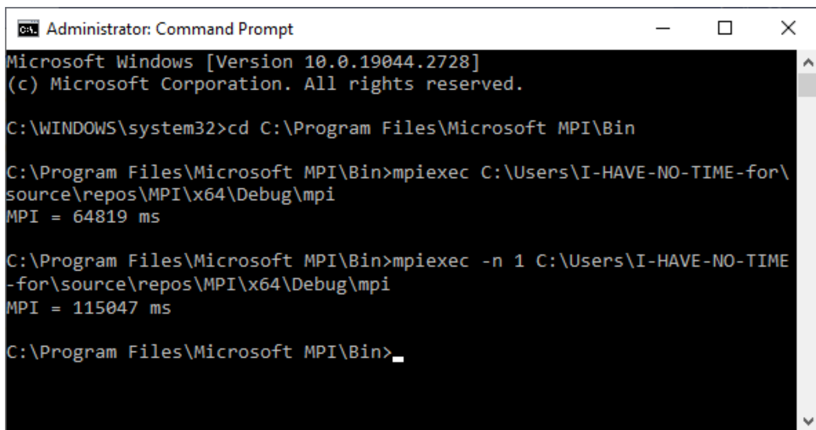

Рис. 2.3. Приклади виконанні паралельної програми на технології MPI при використанні різної кількості потоків

Додатково

Рекомендується встановлювати програму Microsoft MPI у папку за замовчуванням. Інакше можливі проблеми із встановленням пекеджа msmpi.

2.3. Завдання

1. *(20% балів)* Налаштуйте можливість працювати з технологією MPI на Вашому комп’ютері/ноутбуці.
2. *(50% балів)* Модифікуйте наведений вище код згідно Вашого варіанту першої лабораторної роботи.
3. *(10% балів)* Розподіліть роботу між ядрами процесора таким чином, щоб операції згідно варіанту виконувались якомога ефективніше.
4. *(10% балів)* Підберіть таку кількість процесів, щоб по-максимуму дослідити можливості свого обчислювального пристрою.
5. *(10% балів)* Програмно проведіть порівняння кількості часу, необхідного для проведення розрахунків на технологіях MPI/CUDA(OpenCL)/послідовно.
6. Проаналізуйте та поясніть отримані результати.

Додаткові матеріали

1. Основи MPI: https://www.youtube.com/watch?v=c0C9mQaxsD4
2. Складніші випадки використання MPI: https://www.youtube.com/watch?v=q9OfXis50Rg

Контрольні запитання

1. Для чого призначена функція `MPI_Init`?
2. Яка різниця між функціями `MPI_Comm_size` та `MPI_Comm_rank`?
3. Що таке ранг процесу?
4. Як працює функція `MPI_Scatter`?
5. Як працює функція `MPI_Gather`?
6. Для чого призначена функція `MPI_Finalize`?
7. Який ранг зазвичай використовується у якості основного процесу?
8. Що означає `MPI_COMM_WORLD`?

9. Які типи даних має MPI?

10. Яку функцію використовують для звільнення виділеної раніше пам’яті у випадку використання технології MPI?

11. Що із загальновідомого імітує технологія MPI?

12. Який фізичний пристрій призначений для обчислень на технології MPI?

13. Яке зазвичай наявне обмеження при запуску MPI-програми у середовищі Microsoft Visual Studio?

14. Яку команду варто вводити для обчислення рівно на двох потоках?

15. На скількох потоках/ядрах за замовчуванням виконуються обчислення при запуску на виконання MPI-програми з командного рядка?

16. Якої з перелічених функцій не існує в MPI: `MPI_Init`, `MPI_Send`, `MPI_Free`?

17. Що таке `MPI_Status`?

18. Які аргументи приймає функція `MPI_Init`?

19. Як працює функція `MPI_Recv`?

20. Як працює функція `MPI_Reduce`?

Лабораторна робота №3
Сумісні високопродуктивні обчислення на відеокартах та процесорах

Мета

1. Навчитись використанню швидкого методу розпаралелення обчислень на процесорах.
2. Навчитись об'єднувати обчислювальні ресурси високопродуктивних обчислень відеокарт та процесорів.
3. Навчитись аналізувати різні технології обчислень.

3.1. Вступ до технології OpenMP

OpenMP (Open Multi-Processing) – відкритий стандарт для розпаралелювання програм переважно між ядрами процесорів. Реалізується шляхом застосування директив препроцесора «прагм», бібліотечних процедур та змінних оточення, які призначені для програмування багатопотокових додатків на багатопроцесорних системах із загальною пам'яттю.

Ключові елементи стандарту:

- конструкції для створення потоків (директива parallel);
- конструкції розподілу роботи між потоками (директиви DO/for та section);
- конструкції для управління роботою з даними (вирази, які містять слова shared та private, – для визначення класу пам'яті змінних);
- конструкції для синхронізації потоків (директиви critical, atomic та barrier);
- процедури бібліотеки підтримки часу виконання (наприклад, omp_get_thread_num);
- змінні оточення (наприклад, OMP_NUM_THREADS).

3.2. Створення першого проекту на OpenMP у Microsoft Visual Studio

Зручною є розробка паралельних програм на Microsoft Visual Studio. Для створення порівняно простої програми, наприклад, додавання двох векторів з розпаралеленням засобами OpenMP, варто виконати наступну послідовність кроків:

1. Створити порожній консольний проект C++.
2. Помістити наступний код у файл формату *.c або *.cpp:

```
1.  #include <omp.h>
2.  #include <iostream>
3.
4.  int main()
5.  {
6.    const long n = 100;
7.    int* a = (int*)malloc(sizeof(int) * n);
8.    int* b = (int*)malloc(sizeof(int) * n);
9.    int* c = (int*)malloc(sizeof(int) * n);
10.   for (int i = 0; i < n; i++)
11.   {
12.     a[i] = i;
13.     b[i] = i;
14.   }
15.
16.   #pragma omp parallel for
17.   for (int i = 0; i < n; i++)
18.   {
19.     c[i] = a[i] + b[i];
20.   }
21.
22.   for (int i = 0; i < n; i++)
23.   {
24.     printf("%i ", c[i]);
25.   }
26. }
```

Приклад виведення результату зображено на рис. 3.1.

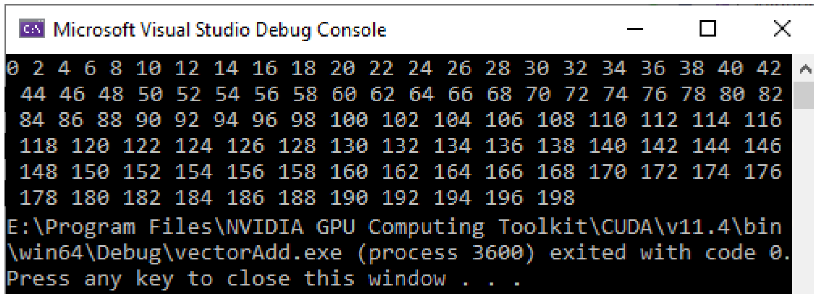

Рис. 3.1. Приклад виконання програми, написаної за допомогою технології OpenMP

3.3. Завдання

1. *(10% балів)* Модифікуйте наведений вище код згідно Вашого варіанту першої лабораторної роботи.
2. *(50% балів)* Якомога ефективніше об’єднайте можливості технологій CUDA/OpenCL та OpenMP.
3. *(20% балів)* Програмно проведіть порівняння кількості часу, необхідного для проведення розрахунків на технологіях MPI/OpenMP/CUDA(OpenCL)/послідовно.
4. Проаналізуйте та поясніть отримані результати.

Додаткові матеріали

1. Відео-уроки по OpenMP: https://www.youtube.com/watch?v=SLnh7yS52-I&list=PL3xCBlatwrsWhsdHq3JFJiuQ60gHzYtFU&index=1.
2. Основи OpenMP: https://www.youtube.com/watch?v=YdHv_2AT4GI&list=PLQsgurGJM5piDxUkHX15V6xrAFHzzWfMh&index=2.

Контрольні запитання

1. Для чого призначена бібліотека `omp.h`?
2. Яким чином фрагмент `#pragma omp parallel` може вплинути на виконання програми?

3. Як працює функція omp_get_thread_num()?
4. Яка різниця між shared та private змінними?
5. Що означає запис #pragma omp barrier?
6. Що означає запис #pragma omp critical { … }?
7. Яким чином функціонує код, який підпорядкований директиві #pragma omp for { … }?
8. Що означає запис #pragma omp parallel private(partial_Sum) shared(total_Sum)?
9. У чому полягає складність поєднання кількох технологій високопродуктивних обчислень?
10. Яка буде реакція компілятора у випадку використання #pragma omp без підключення бібліотеки omp.h?
11. Яку конструкцію використовують для створення потоків?
12. Які існують конструкції для розподілу роботи між потоками?
13. За допомогою яких конструкцій визначають клас пам'яті змінних?
14. Якими способами можна синхронізувати потоки?
15. Що таке OMP_NUM_THREADS?
16. Яка технологія є ефективнішою: MPI чи OpenMP?
17. Що значить рядок #pragma omp master?
18. Які дії потрібно виконати для забезпечення можливості роботи з OpenMP?
19. Яка різниця між виразами #pragma omp parallel та #pragma omp у випадку додавання елементів двох масивів?
20. Як відреагує компілятор на запис #pragma parallel omp parallel?

Рекомендована література

Основна

1. Корочкін О. В., Русанова О. В. Паралельні та розподілені обчислення. Вибрані розділи : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2020. 123 с.

2. Коцовський В. М. Теорія паралельних обчислень : навч. посіб. Ужгород : ПП «АУТДОР-Шарк», 2021. 188 с.

3. Минайленко Р. М. Паралельні та розподілені обчислення : навч. посіб. Кропивницький : Видавець Лисенко В. Ф., 2021. 153 с.

4. Лісовенко І. Д., Яковлєва І. Д. Паралельні та розподілені обчислення : навч. посіб. Чернівці : ЧНУ, 2022. 120 с.

5. Восс М., Асенхо Р., Рейндерс Дж. Паралельне програмування на С++ з допомогою бібліотеки ТВВ. Київ : ДМК Прес, 2020. 674 с.

6. Малашонок Г. І., Сідько А. А. Паралельні обчислення на розподіленій пам’яті: OpenMPI, Java, Math Partner : підручник. Київ : НаУКМА, 2020. 266 с.

Додаткова

7. Bomba A. Ya., Kuzlo M. T., Michuta O. R., Boichura M. V. On a method of image reconstruction of anisotropic media using applied quasipotential tomographic data. *Mathematical Modeling and Computing*. 2019. Vol. 6 (2). P. 211–219.

8. Vlasyuk A., Zhukovskyy V., Zhukovska N., Shatnyi S. Parallel Computing optimization of Two- Dimensional Mathematical Modeling of Contaminant Migration in Catalytic Porous Media. *ACIT’2020:* Proceedings of 2020 10th International Conference on Advanced Computer Information Technologies (Deggendorf, Sep. 16-18, 2020). Deggendorf, 2020. P. 23–28.

9. Overview - CUDA Python 12.1.0 documentation. URL: https://nvidia.github.io/cuda-python/overview.html (Last accessed: 02.06.2023).
10. CUDA C++ Best Practices Guide. Release 12.1. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (Last accessed: 02.06.2023).
11. The programming guide to the CUDA model and interface. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Last accessed: 02.06.2023).
12. The OpenCL™ C Specification. URL: https://registry.kh ronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf (Last accessed: 02.06.2023).
13. MPI: A Message-Passing Interface Standard. Version 4.0. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (Last accessed: 02.06.2023).
14. Using MPI with C — RC University of Colorado Boulder documentation. URL: https://curc.readthedocs.io/en/latest/programming/MPI-C.html (Last accessed: 02.06.2023).
15. Home - OpenMP. URL: http://www.openmp.org (Last accessed: 02.06.2023).
16. OpenMP | LLNL HPC Tutorials. URL: https://hpc-tutorials.llnl.gov/openmp/ (Last accessed: 02.06.2023).
17. OpenMP Application Programming Interface. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (Last accessed: 02.06.2023).
18. Using OpenMP with C — RCU of Colorado Boulder documentation. URL: https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html (Last accessed: 02.06.2023).

Корисні посилання

19.MPI Forum. URL: https://www.mpi-forum.org/ (Last accessed: 02.06.2023).
20.Ukraine - Distributed Computing Team. URL: http://distributed.org.ua/?newlang=ua (Last accessed: 02.06.2023).