Міністерство освіти і науки України
Національний університет водного господарства та природокористування
Навчально-науковий інститут кібернетики, інформаційних технологій та інженерії
Кафедра обчислювальної техніки

04-04-278М

METHODICAL GUIDELINES

for laboratory classes in the course
«Parallel and Distributed Calculations»
for higher education students of the first (Bachelor’s) level
in the educational degree programme «Computer Engineering»
of the field of study 123 «Computer Engineering»
of full-time and part-time forms of study

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

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

Рекомендовано
науково-методичною радою
з якості ННІКІТІ
Протокол № 7 від 17.06.2024 р.

Rivne – 2024

Methodical guidelines for laboratory classes in the course «Parallel and Distributed Calculations» for higher education students of the first (Bachelor's) level in the educational degree programme «Computer Engineering» of the field of study 123 «Computer Engineering» of full-time and part-time forms of study. [Electronic edition] / Boichura M. V., Shatnyi S. V., Shatna A. V. – Rivne : NUWEE, 2024. – 50 p.

Методичні вказівки до лабораторних робіт з навчальної дисципліни «Паралельні та розподілені обчислення» для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Комп'ютерна інженерія» спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання. [Електронне видання] / Бойчура М. В., Шатний С. В., Шатна А. В. – Рівне: НУВГП, 2024. – 50 с.

Compilers:

Boichura M. V., PhD, Associate Professor of the Department of Computer Engineering;

Shatnyi S. V., PhD, Associate Professor of the Department of Computer Engineering;

Shatna A. V., Senior Lecturer of the Department of Computer Engineering.

Укладачі:

Бойчура М. В., к.т.н., доцент кафедри обчислювальної техніки;

Шатний С. В., к.т.н., доцент кафедри обчислювальної техніки;

Шатна А. В., старший викладач кафедри обчислювальної техніки.

Responsible for the publication: Sydor A. I., PhD, Associate Professor, Acting Head of the Department of Computer Engineering.

Відповідальний за випуск: Сидор А. І., к.т.н., доцент, в.о. завідувача кафедри обчислювальної техніки.

Head of the support group
in the field of study
123 «Computer Engineering» Sydor A. I.

Керівник групи забезпечення
спеціальності
123 «Комп'ютерна інженерія» Сидор А. І.

CONTENT

Introduction

In the field of computer engineering, it is crucial to be able to build applications that can operate with maximum capacity of computer hardware. This can be achieved through the ability to operate with parallel threads, which allows for the use of multiple cores of the central processor during the program's operation. Unfortunately, many students learn how to compose program code without taking into consideration the possibility of executing it on parallel threads.

However, the advantages of parallel computing go beyond just using the central processor's multiple cores. Modern computers possess powerful computing devices like graphics cards, which can often execute code hundreds of thousands of times faster than on dozens of powerful CPU cores. By acquiring knowledge and skills in parallel computing, one can build effective programs that extract the maximum potential of the hardware.

The discipline «Parallel and Distributed Computing» is an optional component of the educational program 123 "Computer Engineering" at the bachelor's level of education. The subject provides basic training for a specialist in parallel programming and distributed computing, setting the trajectory for further application of the acquired skills in other aspects of the specialty defined in the Bachelor of Computer Engineering course.

Students studying this subject will acquire theoretical knowledge and basic practical skills in creating parallel programs using common specialized libraries. In addition to the three laboratory works, it is possible to complete other tasks or use other high-performance computing technologies on CPUs/graphics cards or arbitrary programming languages with the consent of the lecturer.

Presenting reports is not required, however, to obtain at least a minimal mark, students should be able to explain each line of code. Only such an approach can help you understand when a given application requires parallelization and what technology should be used.

Finally, completing the course will enable graduates to design and develop parallel and distributed programs. This will prove valuable when planning and executing scientific research, creating games, cryptocurrency mining, or implementing real-time information systems.

As a result of completing all laboratory works of the first module, students should:

know:

– advantages and disadvantages of using parallel computing;

– cases when it is advisable to replace non-parallel code with parallel code;

– basic syntax of CUDA, OpenCL, MPI, OpenMP technologies.

to be able to:

– install libraries for working with CUDA, OpenCL, MPI, OpenMP on various platforms, update drivers, and customize development environments;

– modify non-parallel code into parallel code;

– operate with video card memory and RAM;

– develop parallel programs in teams;

– analyze the characteristics of computing devices according to the criteria of the possibility and feasibility of parallelization;

– successfully combine several high-performance computing technologies;

– programing in accordance with the principles of Coding Conventions.

Вступ

У галузі комп'ютерної інженерії дуже важливо вміти створювати програми, які здатні використовувати максимальні потужності комп'ютерного обладнання. Цього можна досягти завдяки використанню паралельних потоків, що дозволяє задіювати декілька ядер центрального процесора під час роботи програми. На жаль, зазвичай студентів навчають писати програмний код без врахування можливості його виконання на паралельних потоках.

Однак переваги паралельних обчислень виходять за рамки простого використання декількох ядер центрального процесора. Сучасні комп'ютери мають потужні обчислювальні пристрої, такі як відеокарти, які часто можуть виконувати код у сотні тисяч разів швидше, ніж на десятках потужних ядер центрального процесора. Здобувши знання та навички паралельних обчислень, студенти зможуть створювати ефективний програмний код, який враховує максимальний потенціал апаратного забезпечення.

Освітня компонента «Паралельні та розподілені обчислення» є вибірковою складовою освітньо-професійної програми 123 «Комп'ютерна інженерія» бакалаврського рівня вищої освіти. Навчальна дисципліна забезпечує базову підготовку фахівця з паралельного програмування та розподілених обчислень, задаючи траєкторію для подальшого застосування набутих навичок в інших аспектах спеціальності, визначених у курсі підготовки бакалавра з комп'ютерної інженерії.

Студенти, які вивчають цю дисципліну, отримають теоретичні знання та базові практичні навички створення паралельних програм з використанням поширених спеціалізованих бібліотек. В межах лабораторних робіт, за згодою викладача, можна виконувати й інші, окрім запропонованих викладачем, завдання або використовувати інші мови програмування чи технології високопродуктивних обчислень на процесорах/відеокартах.

Представлення звітів не вимагається, однак для отримання хоча б мінімальної оцінки з навчальної дисципліни, студенти повинні вміти пояснити кожен рядок коду. Лише таких підхід сприятиме розумінню того, в яких випадках конкретна програма вимагає розпаралелювання обчислень і яку технологію варто використати.

Виконання усіх наведених завдань дозволить випускникам проектувати та розробляти паралельні і розподілені програми. Це стане в нагоді при плануванні та виконанні наукових досліджень, створенні ігор, майнінгу криптовалют чи впровадженні інформаційних систем реального часу.

У результаті виконання всіх лабораторних робіт першого модуля студенти повинні знати переваги та недоліки використання паралельних обчислень, випадки, коли доцільно замінити звичайний код на паралельний, базовий синтаксис технологій CUDA, OpenCL, MPI, OpenMP; вміти: встановлювати бібліотеки для роботи з CUDA, OpenCL, MPI, OpenMP на різних платформах, оновлювати драйвери та налаштовувати середовища розробки, модифікувати звичайний код у паралельний, працювати з пам'яттю відеокарти та оперативною пам'яттю, командно розробляти програми з реалізацією розпаралелення, аналізувати характеристики обчислювальних пристроїв за критеріями можливості та доцільності розпаралелювання, вдало поєднують декілька високопродуктивних обчислювальних технологій, дотримуватись принципів Coding Conventions.

Laboratory work 1
High-performance computing using video cards

Goals

1. Learn how to investigate hardware to check its compatibility with various HPC technologies on graphics cards.
2. Learn to code relatively simple parallel programs using high-performance computing technologies on video cards.

1.1. Introduction to OpenCL and CUDA technologies

CUDA (Compute Unified Device Architecture) is a software and hardware architecture for parallel computing that enhances computing performance through the use of NVIDIA graphics processors. The CUDA SDK provides the possibility to include calls to subprograms executed on NVIDIA GPUs to the code of C programs. This is achieved through commands written in a special dialect of C. The CUDA architecture allows developers to organize access to the instruction set of the graphics accelerator and manage its memory at the programmer's discretion.

OpenCL (Open Computing Language) is a framework that allows for parallelization of computing on GPUs and CPUs, as well as on FPGAs. The framework includes a programming language based on the C99 standard and an application programming interface (API). OpenCL provides instruction-level and data-level parallelism, complementing the capabilities of the GPU.

The goal of OpenCL is to complement OpenGL and OpenAL, which are industry standards for three-dimensional computer graphics and sound, by leveraging the capabilities of the GPU. It is a fully open standard available under free licenses and is developed and supported by the Khronos Group, a consortium of large companies.

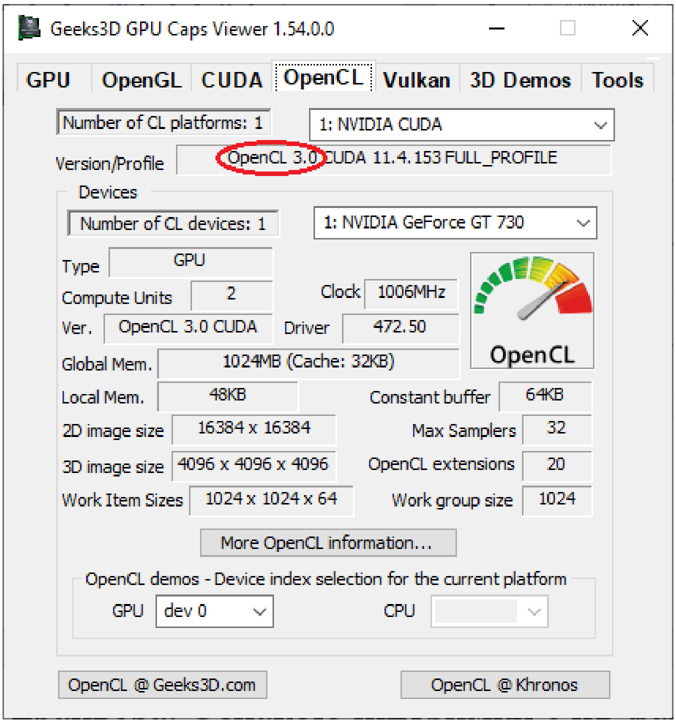
For viewing the video characteristics of your computer, you can use the free GPU Caps Viewer software, available for download at the following link: https://www.geeks3d.com/dl/show/710.

1.2.Provide OpenCL programming capabilities in Microsoft Visual Studio with an existing NVIDIA graphics card

Installing OpenCL

To ensure your driver supports OpenCL technology, you can utilize the GPU Caps Viewer program (Fig. 1.1) or inspect the driver settings (Fig. 1.2). If OpenCL support is not present, consider updating to a more modern driver version. If the latest driver does not support OpenCL, you might need to upgrade to a newer graphics card or execute tasks on a different computer or in the cloud.

If your existing driver version does support OpenCL, the next step is to install the CUDA-Toolkit from the official NVIDIA website. However, it is essential to first determine which versions of the CUDA-Toolkit are compatible with your driver. To do this, follow these steps:



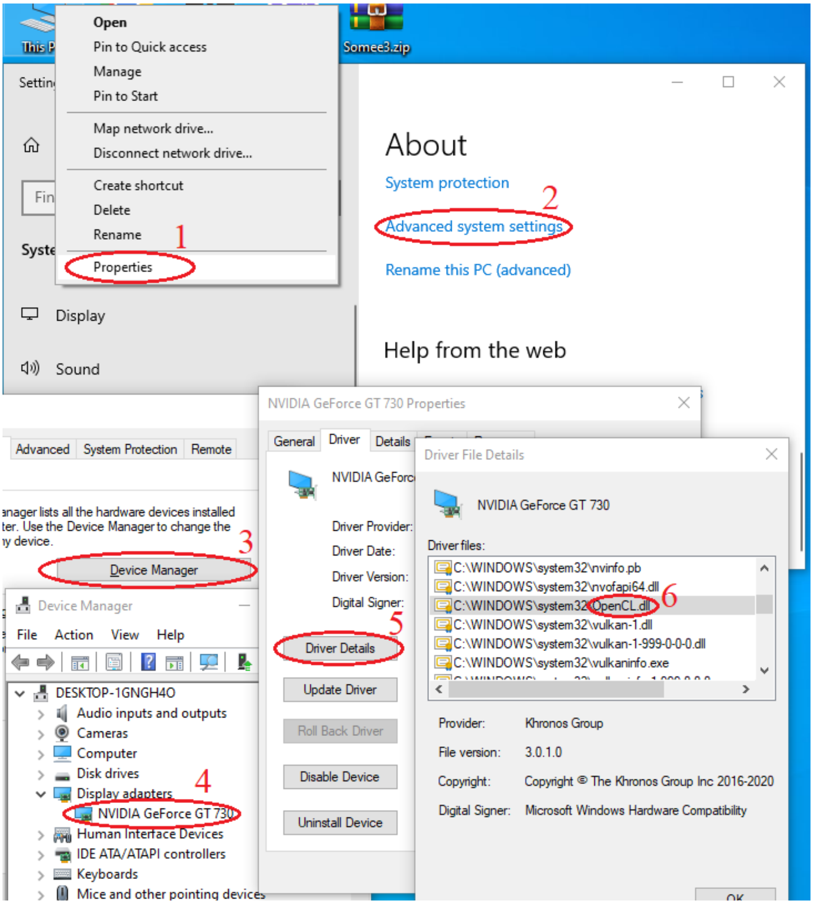
Number of CL platforms: 1 — 1: NVIDIA CUDA
Version/Profile: OpenCL 3.0 CUDA 11.4.153 FULL_PROFILE

Devices
Number of CL devices: 1 — 1: NVIDIA GeForce GT 730
Type: GPU
Compute Units: 2
Clock: 1006MHz
Ver.: OpenCL 3.0 CUDA
Driver: 472.50
Global Mem.: 1024MB (Cache: 32KB)
Local Mem.: 48KB
Constant buffer: 64KB
2D image size: 16384 x 16384
Max Samplers: 32
3D image size: 4096 x 4096 x 4096
OpenCL extensions: 20
Work Item Sizes: 1024 x 1024 x 64
Work group size: 1024

OpenCL demos - Device index selection for the current platform
GPU: dev 0
CPU:

Fig. 1.1. An example of the GPU Caps Viewer program’s tray in case of OpenCL technology support (the presence of the OpenCL version indicates support)

1. Check the version of your graphics card driver using a resource like: https://www.nvidia.com/download/index.aspx.
2. Visit https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html and identify the suitable version of the CUDA-Toolkit (see Table 2 or Table 3 on the web page).
3. Proceed to install the CUDA-Toolkit from https://developer.nvidia.com/cuda-toolkit-archive.
4. After installation, ensure to restart your computer to save the new system paths settings.



Driver files:

```
C:\WINDOWS\system32\nvinfo.pb
C:\WINDOWS\system32\nvofapi64.dll
C:\WINDOWS\system32\OpenCL.dll
C:\WINDOWS\system32\vulkan-1.dll
C:\WINDOWS\system32\vulkan-1-999-0-0-0.dll
C:\WINDOWS\system32\vulkaninfo.exe
```

Provider:	Khronos Group
File version:	3.0.1.0
Copyright:	Copyright © The Khronos Group Inc 2016-2020
Digital Signer:	Microsoft Windows Hardware Compatibility

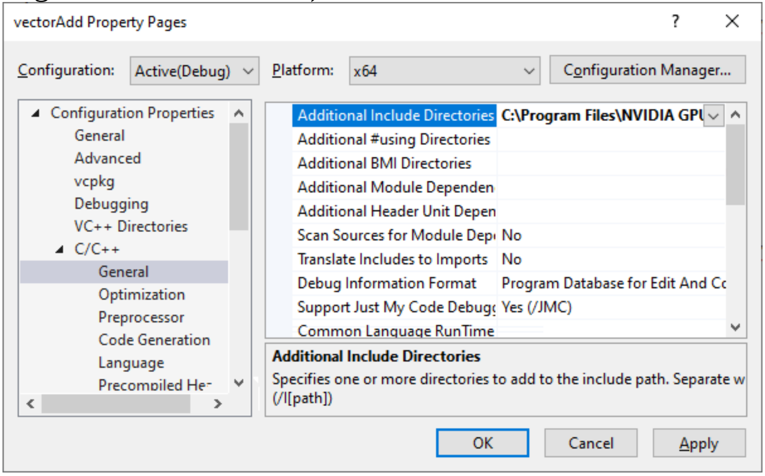
Fig. 1.2. A typical sequence of steps to check OpenCL technology support (support is indicated by the presence of the OpenCL.dll file)

In scenarios where older driver versions necessitate additional component installations apart from the CUDA-Toolkit, it is advisable to seek guidance from your lecturer.

Creating your first OpenCL project

It is convenient to develop parallel programs using Microsoft Visual Studio. In order to create and run an OpenCL project for NVIDIA graphics cards under a 64-bit Windows operating system, you only need to create an empty C++ console project. To create your first OpenCL project, follow these steps to ensure smooth development:

1. Create an empty C++ console project to get started.
2. Include the path of header files (e.g., *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\include*) in the properties of the project at *C/C++ → General → Additional Include Directories* (Fig. 1.3). This directory is created upon installing the CUDA-Toolkit.
3. Add a required path like *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\lib\x64* to the location *Linker → General → Additional Library Directories* (Fig. 1.4).
4. Specify the file name *OpenCL.lib* in the location *Linker → Input → Additional Dependencies* to complete the setup (refer to Fig. 1.5 for illustrations).

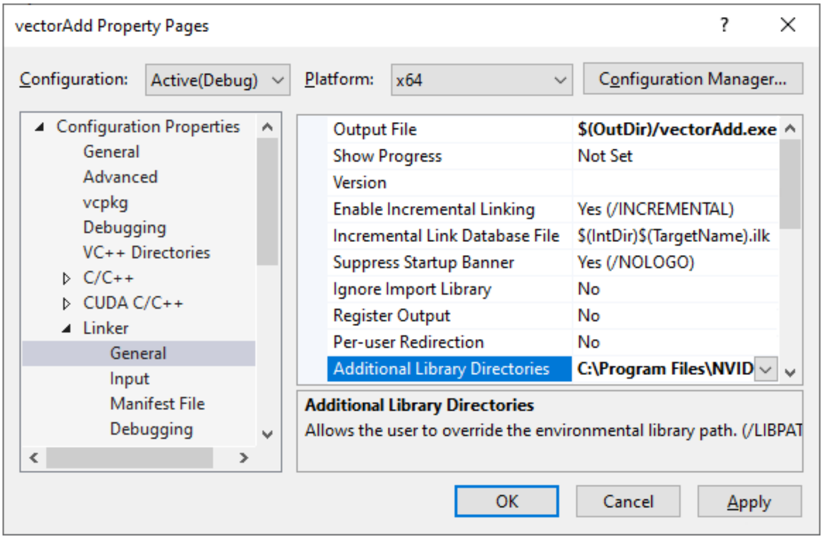


Configuration: Active(Debug) Platform: x64

Additional Include Directories	C:\Program Files\NVIDIA GPU
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependen	
Additional Header Unit Depen	
Scan Sources for Module Dep	No
Translate Includes to Imports	No
Debug Information Format	Program Database for Edit And Co
Support Just My Code Debug	Yes (/JMC)
Common Language RunTime	

Additional Include Directories
Specifies one or more directories to add to the include path. Separate w
(/I[path])

Fig. 1.3. Example of adding the path of the OpenCL header files location

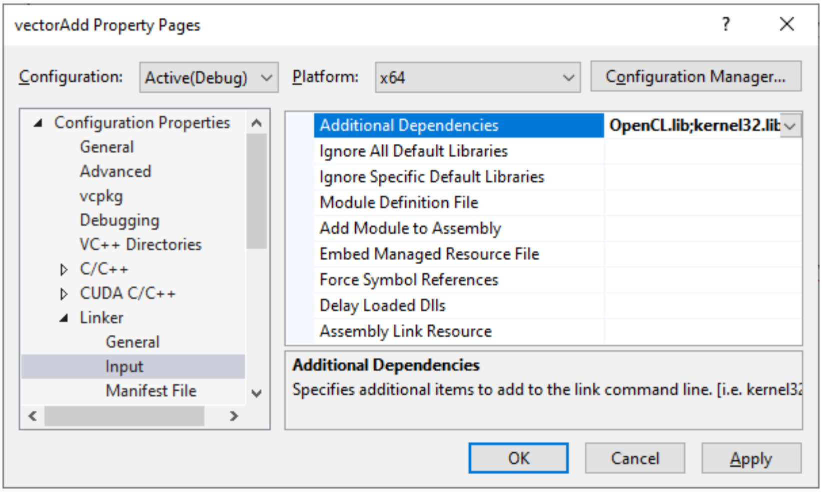


Configuration: Active(Debug) Platform: x64 Configuration Manager...

Output File	**$(OutDir)/vectorAdd.exe**
Show Progress	Not Set
Version	
Enable Incremental Linking	Yes (/INCREMENTAL)
Incremental Link Database File	$(IntDir)$(TargetName).ilk
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**C:\Program Files\NVID**

Additional Library Directories
Allows the user to override the environmental library path. (/LIBPAT

Fig. 1.4. An example of adding the path to the location of OpenCL library files



Configuration: Active(Debug) Platform: x64 Configuration Manager...

Additional Dependencies	**OpenCL.lib;kernel32.lib**
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32

Fig. 1.5. An example of adding the OpenCL.lib library as an additional dependency

There are only 2 files we just need to create:

1. Kernel file: vector_add_kernel.cl with code:

```
__kernel void vector_add(__global const int* A, __global const int* B, __global int* C)
{
  //Get the index of the current item for further processing
  int i = get_global_id(0);
  // Adding vectors
  C[i] = A[i] + B[i];
}
```

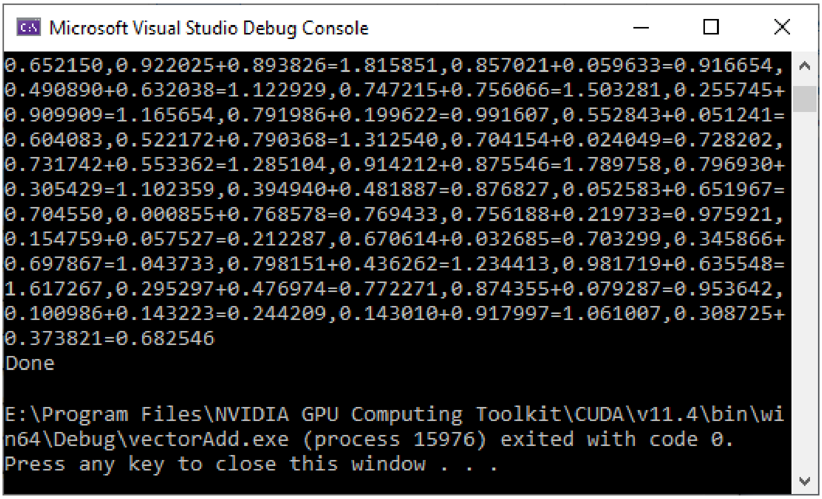
2. A file of the form *.cpp:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

#pragma comment(lib, "OpenCL.lib");

#ifdef __APPLE__
  #include <OpenCL/opencl.h>
#else
  #include <CL/cl.h>
#endif
#define MAX_SOURCE_SIZE (0x100000)

int main(void)
{
  // creating two vectors
  int i;
  const int LIST_SIZE = 1024;
  int* A = (int*)malloc(sizeof(int) * LIST_SIZE);
  int* B = (int*)malloc(sizeof(int) * LIST_SIZE);
  for (i = 0; i < LIST_SIZE; i++)
  {
    A[i] = i;
    B[i] = LIST_SIZE - i;
  }
  // reading kernel source code from vector_add_kernel.cl
  FILE* fp;
```

```
char* source_str;
size_t source_size;
fp = fopen("vector_add_kernel.cl", "r");
if (!fp)
{
  fprintf(stderr, "Failed to load kernel.\n");
  exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);
// Get information about platforms and devices
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &ret_num_devices);
// Creating an OpenCL context
cl_context context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
// Creating a command queue
cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
// Creating memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context,
  CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context,
  CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context,
  CL_MEM_WRITE_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
// Copying vectors to memory buffers
ret = clEnqueueWriteBuffer(command_queue,
  a_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue,
  b_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int), B, 0, NULL, NULL);
// Creating a program from the kernel source code
cl_program program = clCreateProgramWithSource(context, 1,
  (const char**)&source_str, (const size_t*)&source_size, &ret);
// Creating an executable file
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
// Creating an OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
// Setting kernel arguments
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&a_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&b_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&c_mem_obj);
// Executing the kernel
size_t global_item_size = LIST_SIZE;
size_t local_item_size = 64;
ret = clEnqueueNDRangeKernel(command_queue,
  kernel, 1, NULL, &global_item_size,
  &local_item_size, 0, NULL, NULL);
// Reading the result from the device to the local list C
int* C = (int*)malloc(sizeof(int) * LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue,
  c_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int), C, 0, NULL, NULL);
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(a_mem_obj);
ret = clReleaseMemObject(b_mem_obj);
ret = clReleaseMemObject(c_mem_obj);
```

```
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);
    free(A);
    free(B);
    free(C);
    return 0;
}
```

If we include the print function, the last lines of the program result may look like the one shown in Fig. 1.6.



Microsoft Visual Studio Debug Console

```
0.652150,0.922025+0.893826=1.815851,0.857021+0.059633=0.916654,
0.490890+0.632038=1.122929,0.747215+0.756066=1.503281,0.255745+
0.909909=1.165654,0.791986+0.199622=0.991607,0.552843+0.051241=
0.604083,0.522172+0.790368=1.312540,0.704154+0.024049=0.728202,
0.731742+0.553362=1.285104,0.914212+0.875546=1.789758,0.796930+
0.305429=1.102359,0.394940+0.481887=0.876827,0.052583+0.651967=
0.704550,0.000855+0.768578=0.769433,0.756188+0.219733=0.975921,
0.154759+0.057527=0.212287,0.670614+0.032685=0.703299,0.345866+
0.697867=1.043733,0.798151+0.436262=1.234413,0.981719+0.635548=
1.617267,0.295297+0.476974=0.772271,0.874355+0.079287=0.953642,
0.100986+0.143223=0.244209,0.143010+0.917997=1.061007,0.308725+
0.373821=0.682546
Done

E:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.4\bin\wi
n64\Debug\vectorAdd.exe (process 15976) exited with code 0.
Press any key to close this window . . .
```

Fig. 1.6. An example of displaying the last lines in the console window

Additionally

There may be cases when the code works correctly, but there is no syntax highlighting for OpenCL. This problem can be solved as follows:

1. Choose the *Tools* → *Options* menu item.
2. Find the *Text Editor* → *File Extension* in the list of options.

3. Type *cl* in the *Extension* field; *Microsoft Visual C++* in the *Editor* field.

4. Click *Add*.

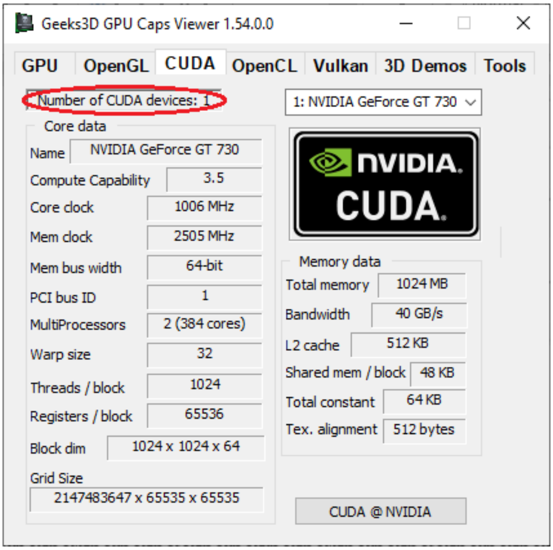
5. Click *OK*.

6. Restart the Microsoft Visual Studio environment.

1.3. Provide CUDA programming capabilities in Microsoft Visual Studio with an existing NVIDIA graphics card

Installing CUDA

To verify CUDA technology support, utilize the GPU Caps Viewer program (Fig. 1.7) or inspect the driver settings (Fig. 1.8). If the driver does not support CUDA, consider installing a more recent version or upgrading to a more modern graphics card. Alternatively, you can perform tasks on another computer or in the cloud.



Number of CUDA devices: 1

1: NVIDIA GeForce GT 730

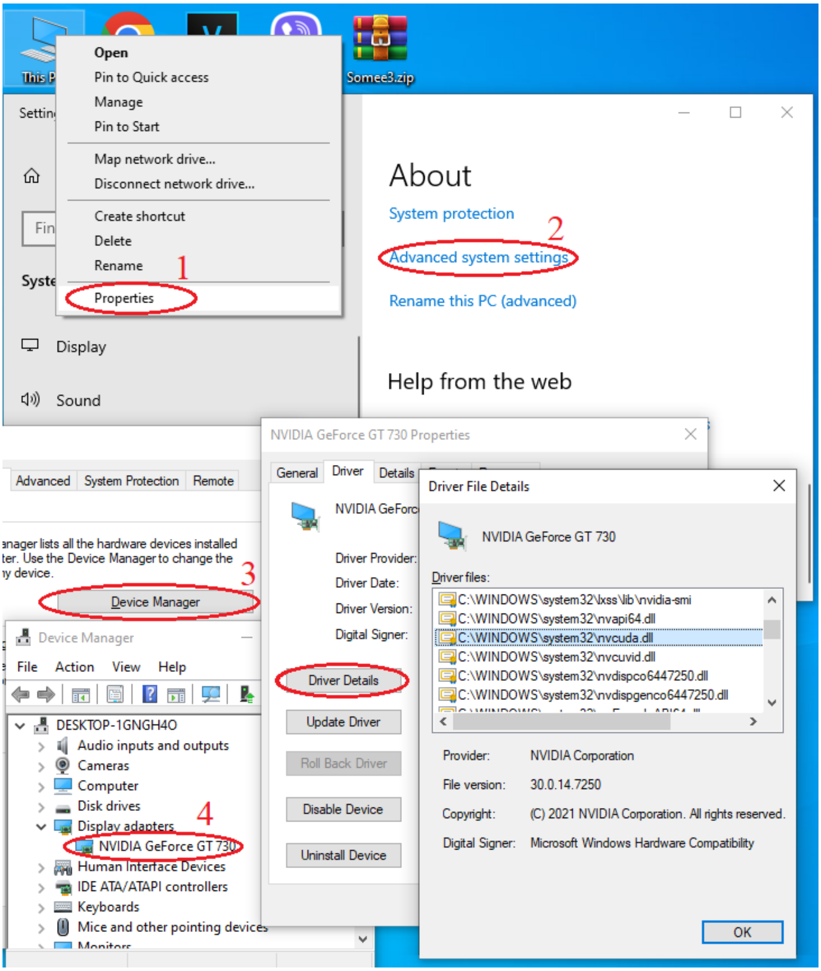
Core data

Name	NVIDIA GeForce GT 730
Compute Capability	3.5
Core clock	1006 MHz
Mem clock	2505 MHz
Mem bus width	64-bit
PCI bus ID	1
MultiProcessors	2 (384 cores)
Warp size	32
Threads / block	1024
Registers / block	65536
Block dim	1024 x 1024 x 64
Grid Size	2147483647 x 65535 x 65535

Memory data

Total memory	1024 MB
Bandwidth	40 GB/s
L2 cache	512 KB
Shared mem / block	48 KB
Total constant	64 KB
Tex. alignment	512 bytes

Fig. 1.7. An example of the GPU Caps Viewer program’s wreath in the case of CUDA technology support (the number of devices is greater than 0, which indicates CUDA support)



anager lists all the hardware devices installed
ter. Use the Device Manager to change the
ty device.

Driver files:

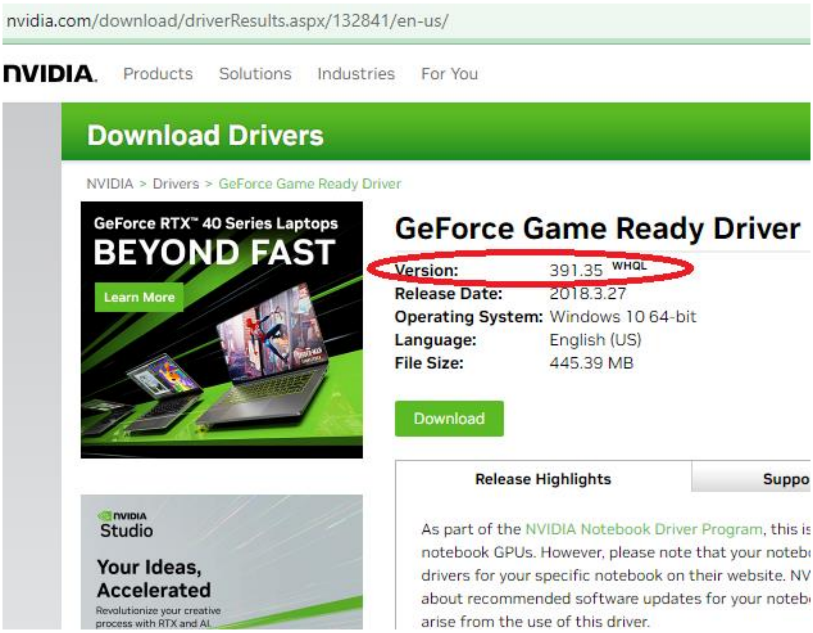
C:\WINDOWS\system32\lxss\lib\nvidia-smi
C:\WINDOWS\system32\nvapi64.dll
C:\WINDOWS\system32\nvcuda.dll
C:\WINDOWS\system32\nvcuvid.dll
C:\WINDOWS\system32\nvdispco6447250.dll
C:\WINDOWS\system32\nvdispgenco6447250.dll

Provider:	NVIDIA Corporation
File version:	30.0.14.7250
Copyright:	(C) 2021 NVIDIA Corporation. All rights reserved.
Digital Signer:	Microsoft Windows Hardware Compatibility

Fig. 1.8. A typical sequence of steps to check for CUDA support (the presence of a text like in the Driver File Details window is a sign of support)

Follow these steps to ensure the possibility of programming in CUDA:

1. Use the following link: https://www.nvidia.com/download/index.aspx, select the current driver version for your graphics card, and install it (see Fig. 1.9).
2. Compare the driver version with Table 2 or 3 at https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html to determine the compatible CUDA Toolkit version.
3. Install the identified driver and CUDA-Toolkit from https://developer.nvidia.com/cuda-toolkit-archive.
4. After installation, restart your computer.


nvidia.com/download/driverResults.aspx/132841/en-us/

GeForce Game Ready Driver

Version:	391.35 WHQL
Release Date:	2018.3.27
Operating System:	Windows 10 64-bit
Language:	English (US)
File Size:	445.39 MB

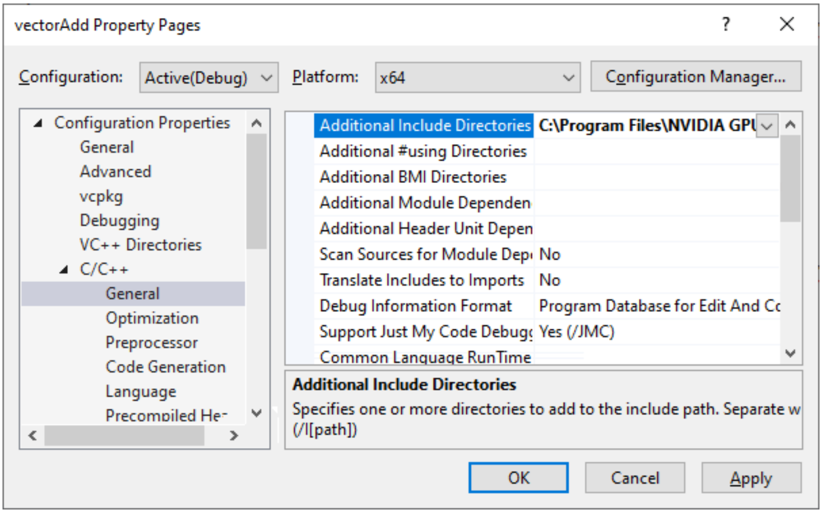
As part of the NVIDIA Notebook Driver Program, this is
notebook GPUs. However, please note that your noteb
drivers for your specific notebook on their website. NV
about recommended software updates for your noteb
arise from the use of this driver.

Fig. 1.9. A window with the latest version of some NVIDIA driver

Creating your first CUDA project

To create your first CUDA project on a 64-bit Windows operating system in Microsoft Visual Studio, follow these steps:

1. Start Microsoft Visual Studio and search among the project templates for one containing the word “CUDA”. Create a project using this template.
2. If the CUDA template is not available, create an empty C++ console project instead.
3. Access the properties of the created project and add the header files' address located at C:\Programme Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\include\ to the location *C/C++ → General → Additional Include Directories* as shown in Fig. 1.10. This directory is created during CUDA-Toolkit installation.
4. Next, add a path such as C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.5\lib\x64 to the location *Linker → General → Additional Library Directories* as shown in Fig. 1.11.
5. Finally, locate the *Linker → Input → Additional Dependencies* location and add a file similar to *CUDA.lib*, as displayed in Fig. 1.12 to complete the setup.

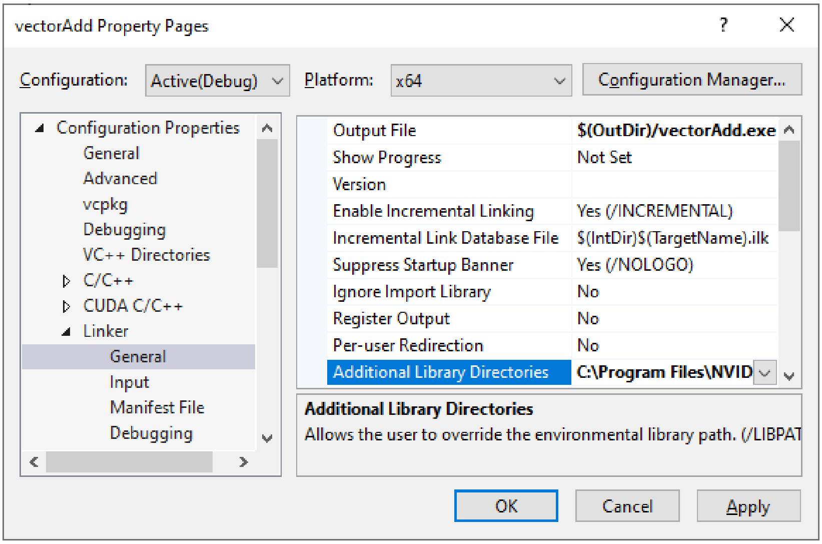

vectorAdd Property Pages

Configuration: Active(Debug) Platform: x64 Configuration Manager...

Additional Include Directories	C:\Program Files\NVIDIA GPU
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependen	
Additional Header Unit Depen	
Scan Sources for Module Depe	No
Translate Includes to Imports	No
Debug Information Format	Program Database for Edit And Co
Support Just My Code Debug	Yes (/JMC)
Common Language RunTime	

Additional Include Directories
Specifies one or more directories to add to the include path. Separate w
(/I[path])

Fig. 1.10. Example of adding the address of the location of CUDA header files

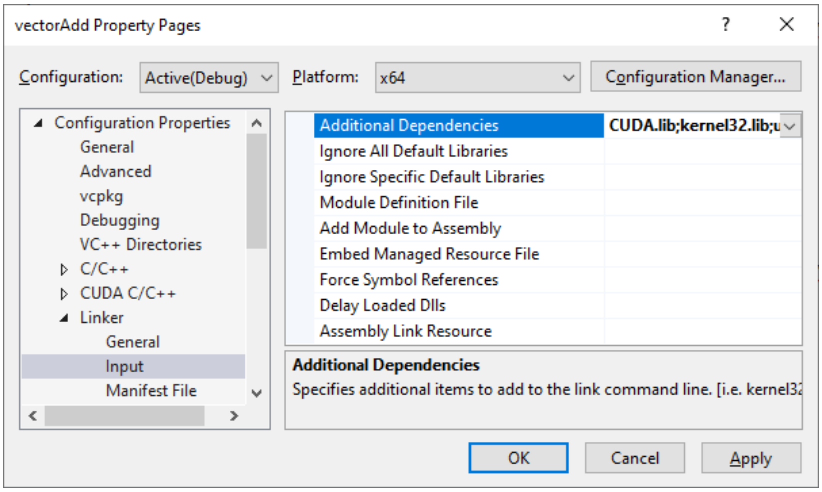

vectorAdd Property Pages

Configuration: Active(Debug) Platform: x64 Configuration Manager...

Output File	$(OutDir)/vectorAdd.exe
Show Progress	Not Set
Version	
Enable Incremental Linking	Yes (/INCREMENTAL)
Incremental Link Database File	$(IntDir)$(TargetName).ilk
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	C:\Program Files\NVID

Additional Library Directories
Allows the user to override the environmental library path. (/LIBPAT

Fig. 1.11. Example of adding a path to the location of CUDA library files



Configuration: Active(Debug)
Platform: x64
Configuration Manager...

Additional Dependencies: CUDA.lib;kernel32.lib;u

Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32

Fig. 1.12. Example of adding the CUDA.lib library as an additional dependency

It remains to create 1 file:

```
__global__ void vectorAdd(const float *A, const
    float *B, float *C, int numElements)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < numElements)
  {
    C[i] = A[i] + B[i];
  }
}
int main()
{
  cudaError_t err = cudaSuccess;
  int numElements = 50000000;
  size_t size = numElements * sizeof(float);
  printf("[Vector addition of %d elements]\n",
    numElements);

  float *h_A = (float *)malloc(size);
  float *h_B = (float *)malloc(size);
float *h_C = (float *)malloc(size);

if (h_A == NULL || h_B == NULL || h_C == NULL)
  exit(EXIT_FAILURE);
for (int i = 0; i < numElements; ++i)
{
  h_A[i] = rand()/(float)RAND_MAX;
  h_B[i] = rand()/(float)RAND_MAX;
}

float *d_A = NULL;
float *d_B = NULL;
float *d_C = NULL;

err = cudaMalloc((void **)&d_A, size);
if (err != cudaSuccess)
  exit(EXIT_FAILURE);
err = cudaMalloc((void **)&d_B, size);
if (err != cudaSuccess)
  exit(EXIT_FAILURE);
err = cudaMalloc((void **)&d_C, size);
if (err != cudaSuccess)
  exit(EXIT_FAILURE);

printf("Copy input data from the host memory to
   the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size,
   cudaMemcpyHostToDevice);
if (err != cudaSuccess)
  exit(EXIT_FAILURE);
err = cudaMemcpy(d_B, h_B, size,
   cudaMemcpyHostToDevice);
if (err != cudaSuccess)
  exit(EXIT_FAILURE);

int threadsPerBlock = 256;
int blocksPerGrid = (numElements +
   threadsPerBlock - 1) / threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d
   threads\n", blocksPerGrid, threadsPerBlock);
vectorAdd<<<blocksPerGrid,
   threadsPerBlock>>>(d_A, d_B, d_C, numElements);
  err = cudaGetLastError();
  if (err != cudaSuccess)
    exit(EXIT_FAILURE);

  printf("Copy output data from the CUDA device to the host memory\n");
  err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  if (err != cudaSuccess)
    exit(EXIT_FAILURE);

  err = cudaFree(d_A);
  if (err != cudaSuccess)
    exit(EXIT_FAILURE);
  err = cudaFree(d_B);
  if (err != cudaSuccess)
    exit(EXIT_FAILURE);
  err = cudaFree(d_C);
  if (err != cudaSuccess)
    exit(EXIT_FAILURE);

  free(h_A);
  free(h_B);
  free(h_C);

  printf("Done\n");
  return 0;
}
```

If you also add output functions, the last lines of the program result may look like the one shown in Fig. 1.6.

Additionally

There may be cases when the code works correctly, but there is no syntax highlighting for CUDA. This problem can be solved as follows:

1. Select the *Tools* → *Options* menu item.
2. Search for *Text Editor* → *File Extension* in the list of options

3. Type *cu* in the *Extension* field; select *Microsoft Visual C++* in the *Editor* field.
4. Click Add.
5. Click OK.
6. Restart the *Microsoft Visual Studio* environment.

1.4. Provide OpenCL programming capabilities in Microsoft Visual Studio with an existing AMD graphics card

Installing OpenCL

To determine if your driver supports OpenCL technology, check with the GPU Caps Viewer program (Fig. 1.1) or refer to the driver settings. An indication of support is the existence of a file like OpenCL.dll (Fig. 1.2). If this specific file is not present, it is recommended to update to a more recent version of the driver. In cases where the latest driver version does not offer OpenCL support, consider upgrading to a more modern graphics card or completing tasks on a different computer or in the cloud.

If the current driver version does support OpenCL, the next step involves downloading the SDK through one of the two methods provided:

1. Utilizing vcpkg (this's preferred method).

   1.1. Install Git for Windows from https://gitforwindows.org/ and then restart your computer.

   1.2. Use a command prompt in administrator mode to execute the following steps sequentially:

   - 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. Restart your computer to apply the changes.

2. Access the open source project on GitHub by visiting: https://github.com/GPUOpen-LibrariesAndSDKs/OCL-SDK/releases.

Creating your first OpenCL project

If you want to create and execute your first OpenCL project for AMD graphics card on a 64-bit Windows operating system, it is beneficial to use Microsoft Visual Studio. Utilize an empty C++ console project as the basis for development. Should any obstacles arise when attempting to access libraries, follow the listed steps:

1. Create an empty C++ console project.
2. In the project's properties, go to *C/C++ → General → Additional Include Directories* and add the path of header files, such as C:\Users\Roma\Downloads\lightOCLSDK\include, from the unzipped SDK downloaded on GitHub.
3. Add a new path, such as *C:\Users\Roma\Downloads\lightOCLSDK\lib\x86_64*, to the project's properties location *Linker → General → Additional Library Directories*.
4. Add a filename like *OpenCL.lib* to the *Linker → Input → Additional Dependencies* location in the project's properties.

It remains to create 2 files:

1. The kernel file: vector_add_kernel.cl with the code

```
__kernel void vector_add(__global const int* A,
        __global const int* B, __global int* C)
{
   //Get the index of the current item to process
   int i = get_global_id(0);
   // Adding vectors
   C[i] = A[i] + B[i];
}
```

2. A file of the form *.cpp:

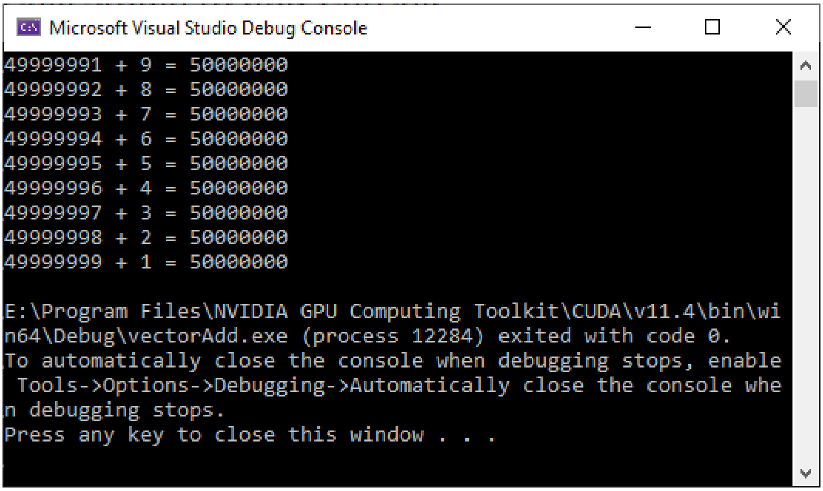
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
```

```c

#pragma comment(lib, "OpenCL.lib");

#ifdef __APPLE__
  #include <OpenCL/opencl.h>
#else
  #include <CL/cl.h>
#endif
#define MAX_SOURCE_SIZE (0x100000)

int main(void)
{
  // creating two vectors
  int i;
  const int LIST_SIZE = 100000000 / 2;
  int* A = (int*)malloc(sizeof(int) *LIST_SIZE  );
  int* B = (int*)malloc(sizeof(int) *LIST_SIZE  );
  for (i = 0; i < LIST_SIZE; i++)
  {
    A[i] = i;
    B[i] = LIST_SIZE - i;
  }
  // reading kernel source code from vector_add_kernel.cl
  FILE* fp;
  char* source_str;
  size_t source_size;
  fp = fopen("vector_add_kernel.cl", "r");
  if (!fp)
  {
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
  }
  source_str = (char*)malloc(MAX_SOURCE_SIZE);
  source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
  fclose(fp);
  // Get information about platforms and devices
  cl_platform_id platform_id = NULL;
  cl_device_id device_id = NULL;
  cl_uint ret_num_devices;
  cl_uint ret_num_platforms;
```

```
cl_int ret = clGetPlatformIDs(1, &platform_id,
  &ret_num_platforms);
ret = clGetDeviceIDs(platform_id,
  CL_DEVICE_TYPE_GPU, 1, &device_id,
  &ret_num_devices);
// Creating an OpenCL context
cl_context context = clCreateContext(NULL, 1,
  &device_id, NULL, NULL, &ret);
// Creating a command queue
cl_command_queue command_queue =
  clCreateCommandQueue(context, device_id, 0,
  &ret);
// Creating memory buffers on the device for each
  vector
cl_mem a_mem_obj = clCreateBuffer(context,
  CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int),
  NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context,
  CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int),
  NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context,
  CL_MEM_WRITE_ONLY, LIST_SIZE * sizeof(int),
  NULL, &ret);
// Copying vectors to memory buffers
ret = clEnqueueWriteBuffer(command_queue,
  a_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
  A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue,
  b_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
  B, 0, NULL, NULL);
// Creating a program from the kernel source code
cl_program program =
  clCreateProgramWithSource(context, 1,
(const char**)&source_str, (const
  size_t*)&source_size, &ret);
// Creating an executable file
ret = clBuildProgram(program, 1, &device_id,
  NULL, NULL, NULL);
// Creating an OpenCL kernel
cl_kernel kernel = clCreateKernel(program,
  "vector_add", &ret);
// Setting kernel arguments
```

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
  (void*)&a_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),
  (void*)&b_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),
  (void*)&c_mem_obj);
// Executing the kernel
size_t global_item_size = LIST_SIZE;
size_t local_item_size = 64;
ret = clEnqueueNDRangeKernel(command_queue,
  kernel, 1, NULL, &global_item_size,
  &local_item_size, 0, NULL, NULL);
// Reading the result from the device to the
  local list C
int* C = (int*)malloc(sizeof(int) * LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue,
  c_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int),
  C, 0, NULL, NULL);
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(a_mem_obj);
ret = clReleaseMemObject(b_mem_obj);
ret = clReleaseMemObject(c_mem_obj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
free(A);
free(B);
free(C);
return 0;
}
```

If you also add output functions, the last lines of the program result can look like the one shown in Fig. 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 . . .
```

Fig. 1.13. Example of the last lines of the output window that displays the result of adding two vectors

Additionally

There may be cases when the code works correctly, but there is no syntax highlighting for OpenCL. This problem can be solved as follows:

1. Select the Tools → Options menu item.
2. In the list of options, find Text Editor → File Extension.
3. Enter *cl* in the Extension field; select *Microsoft Visual C++* in the Editor field.
4. Click Add.
5. Click OK.
6. Restart the Microsoft Visual Studio environment.

1.5. Statements

1. Configure the working capability of OpenCL or CUDA on your computer or laptop (*20% of points*).

2. Create two separate kernel functions, each dedicated to its unique operation according to the chosen option. If OpenCL technology is being used, it is recommended to create two corresponding files with the extension *.cl (*10% of points*).
3. Determine the number of array elements to check the maximum capabilities of your computing device (*10% of points*).
4. Implement the execution of both cores separately and in parallel according to the chosen option (*50% of points*).
5. Programmatically compare the amount of time required to perform calculations in parallel versus sequentially (*10% of points*).
6. Analyze and explain the calculated results.

Options for accomplishing the task

The following options are available for selecting the operation to be performed:

Option 1.	Addition and subtraction.
Option 2.	Addition and multiplication.
Option 3.	Addition and division.
Option 4.	Addition and floor division.
Option 5.	Addition and division with remainder.
Option 6.	Subtraction and multiplication.
Option 7.	Subtraction and division.
Option 8.	Subtraction and floor division.
Option 9.	Subtraction and division with remainder.
Option 10.	Multiplication and division.
Option 11.	Multiplication and floor division.
Option 12.	Multiplication and division with remainder.
Option 13.	Division and floor division.
Option 14.	Division and division with remainder.
Option 15.	Floor division and division with remainder.

Additional materials

1. An introductory video on CUDA basics and some examples in Python: https://www.youtube.com/watch?v=r9IqwpMR9TE&list=PLqXS1b2lRpYTUHPp2MYkgXS7v6qA-JsF.
2. Lessons about CUDA: https://www.youtube.com/watch?v=m0nhePeHwFs&list=PLKK11Ligqititws0ZOoGk3SW-TZCar4dK&index=1
3. An introductory video about OpenCL: https://www.youtube.com/watch?v=V4RfPfHQPC8&list=PLiwt1iVUib9s6vyEqdpcgAq7NBRlp9mAY&index=1
4. Lectures about OpenCL: https://www.youtube.com/watch?v=8D6yhpiQVVI&list=PLDivN33Lbf6cLqZ5_i_k0KeMaQKEctMgS

Questions for independent training

1. What is the `__global__` qualifier used for?
2. What is the `blockdim.x * blockidx.x + threadidx.x` expression for?
3. Why is it important to specify an if statement `if (i < numElements)` in the kernel?
4. What serious limitation has the `new` operator compared to the `malloc` function?
5. Why is it advisable to use the `if` statement directly after using the `malloc` function?
6. What is the difference between the `malloc` and `cudamalloc` functions?
7. How does the `cudamemcpy` function perform?
8. What are the essence of the `threadsperblock` and `blockspergrid` variables?
9. What do angle brackets mean in the expression `func<<<blockspergrid, threadsperblock>>>(d_a, d_b, d_c, numElements)`?

10. What is the difference between the `cudafree` and `free` functions?

11. What is the difference between a device and a host in CUDA terminology?

12. What is the `__kernel` keyword used for?

13. What is the purpose of the `get_global_id(0)` function?

14. What is the difference between the `clGetPlatformIDs` and `clGetDeviceIDs` functions?

15. What is the purpose of the `clCreateContext` function?

16. What is the purpose of the `clCreateCommandQueue` function?

17. How do the `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` functions performs?

18. What is the purpose of the `clBuildProgram` function?

19. What is the purpose of the `clCreateKernel` function?

20. What is the purpose of the `clEnqueueNDRangeKernel` function?

Laboratory work 2
High-performance computing on processors

Goals

1. Learn how to create relatively simple programs using high-performance computing technologies on CPU cores.
2. Learn how to analyze and compare different HPC technologies.

2.1. Introduction to MPI technology

Message Passing Interface (MPI) serves as a critical software interface (API) for facilitating information transfer by enabling the communication of messages among processes striving for a shared objective. Essentially, the concept behind MPI involves adapting the technology of message transmission between pairs or groups of subscribers for the realm of parallel computing.

The utilization of MPI stands as the predominant standard for data exchange interfaces in the realm of parallel programming, with implementations available across a plethora of computer platforms. This technology is particularly instrumental in crafting programs designed for clusters and supercomputers, with the primary mode of communication among MPI processes involving the exchange of messages with one another.

2.2. Provide MPI programming capabilities in Microsoft Visual Studio with an existing AMD graphics card

Installing MPI

A guide for Installing MPI on a Windows operating system and creating the first MPI project is as below.

Implementing MPI Technology:

1. Begin by installing Microsoft MPI from https://www.microsoft.com/en-us/download/details.aspx?id=100593. Following the installation, restart the computer.
2. Proceed by installing Git for Windows, which can be obtained from https://gitforwindows.org, and then restart the computer.
3. Using a command prompt that runs in administrator mode, execute the following steps in sequence:

   3.1. Navigate to the directory by typing "cd c:\Program Files".

   3.2. Clone the vcpkg repository by running the command "git clone https://github.com/Microsoft/vcpkg.git".

   3.3. Execute the bootstrap-vcpkg.bat file by typing ".\vcpkg\bootstrap-vcpkg.bat".

   3.4. Install the msmpi package for x64 Windows using ".\vcpkg install msmpi:x64-windows".

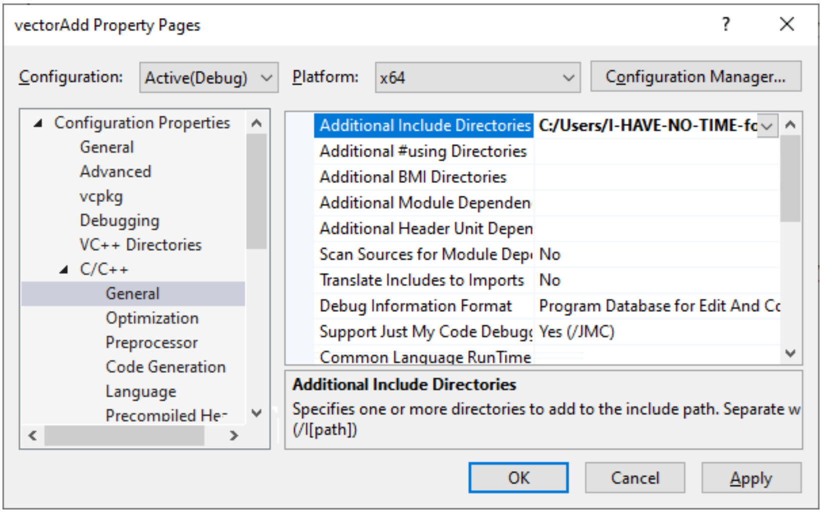
   3.5. Integrate the installed by running ".\vcpkg integrate install". After completing these steps, restart the computer.

Creating the first MPI project

Microsoft Visual Studio provides a convenient environment for developing parallel programs. To set up an MPI project for a 64-bit Windows operating system, simply create an empty C++ console project and input the desired MPI code. Should issues relating to library access arise, you will need to follow these steps:

1. Start by creating an empty C++ console project.
2. In the project's properties, navigate to *C/C++* → *General* → *Additional Include Directories* and include the path of the header files, such as *C:/Users/I-HAVE-NO-TIME-for/vcpkg/packages/msmpi_x64-windows/include/*. This

directory becomes accessible after installing the msmpi package (refer to Fig. 2.1 for guidance).


vectorAdd Property Pages

Configuration: Active(Debug) Platform: x64 Configuration Manager...

- Configuration Properties
  - General
  - Advanced
  - vcpkg
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled He-

Additional Include Directories	C:/Users/I-HAVE-NO-TIME-fc
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependen	
Additional Header Unit Depen	
Scan Sources for Module Depe	No
Translate Includes to Imports	No
Debug Information Format	Program Database for Edit And Cc
Support Just My Code Debugg	Yes (/JMC)
Common Language RunTime	

Additional Include Directories
Specifies one or more directories to add to the include path. Separate w
(/I[path])

Fig. 2.1. Example of insert the address of MPI header files location

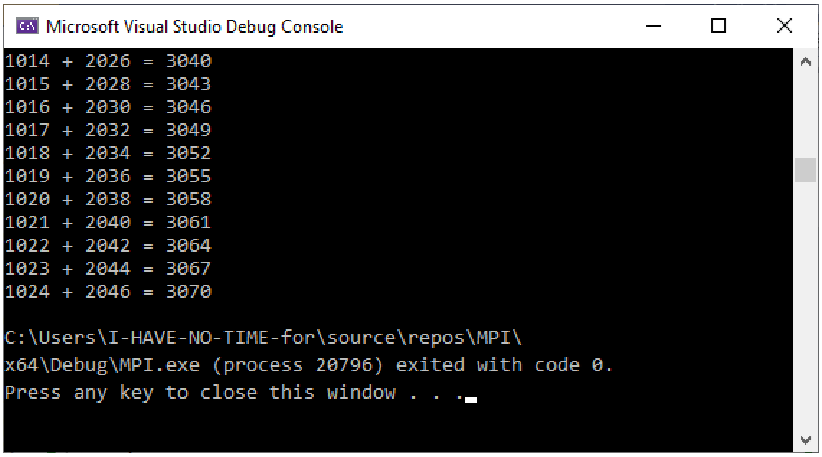
We need to place the following code in a *.c or *.cpp file:

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define MASTER 0 //Create a synonym for the rank of the main process
#define ARRAY_SIZE 1024 //Set the sizes for arrays.

int main(int argc, char* argv[]) //Entry point to the parallel program
{

  // Create pointers to the addresses where the arrays will be located
  int* a = NULL;
  int* b = NULL;
  int* c = NULL;

  int total_proc;   // total number of processes
  int rank;         // the rank of each process
  int n_per_proc;   // elements allocated to the process
  int n = ARRAY_SIZE;
  int i;

  MPI_Status status; // used to check for MPI errors.

  // 1. Initialize the MPI environment
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &total_proc);
  // 2. Assign the total number of processes to the total_proc variable
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // 3. Assign the number of the current process to the rank variable

  // Pointers to a number of short arrays that will be used for parallel additions
  int* ap;
  int* bp;
  int* cp;

  // 4. The main process initializes the arrays that will be used for addition.
  if (rank == MASTER)
  {
    a = (int*)malloc(sizeof(int) * n);
    b = (int*)malloc(sizeof(int) * n);
    c = (int*)malloc(sizeof(int) * n);

    for (i = 0; i < n; i++)
    {
      a[i] = i + 1;
    }
    for (i = 0; i < n; i++)
    {
```

```
    b[i] = i * 2;
  }
}

// set the number of pairs of elements that will be added in each process
n_per_proc = n / total_proc;

// 5. Memory allocation for shortened arrays
ap = (int*)malloc(sizeof(int) * n_per_proc);
bp = (int*)malloc(sizeof(int) * n_per_proc);
cp = (int*)malloc(sizeof(int) * n_per_proc);

// 6. Uniform distribution of array a into shortened arrays ap
MPI_Scatter(a, n_per_proc, MPI_INT, ap, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);
// Uniform distribution of array b into shortened arrays bp
MPI_Scatter(b, n_per_proc, MPI_INT, bp, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);

// 7. Pairwise addition of elements of shortened arrays
for (i = 0; i < n_per_proc; i++)
{
  cp[i] = ap[i] + bp[i];
}

// 8. Collecting shortened arrays cp into a single array c
MPI_Gather(cp, n_per_proc, MPI_INT, c, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);

// Outputting results in the main process
if (rank == MASTER)
{
  for (i = 0; i < n; i++)
  {
    printf("%lld + %lld = %lld\n", a[i], b[i], c[i]);
  }
  }

  // Freeing up memory
  if (rank == MASTER)
  {
    free(a);
    free(b);
    free(c);
  }
  free(ap);
  free(bp);
  free(cp);

  // 9. Terminating MPI processes and environment
  MPI_Finalize();

  return 0;
}
```

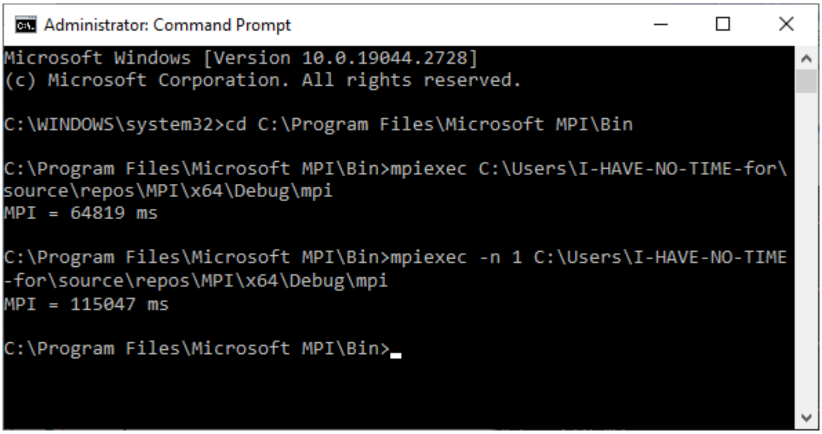
An example of outputting the last lines of the program result is shown in Fig. 2.2.

If you are executing the provided code from the Microsoft Visual Studio environment, it is important to note that the computations will be carried out on a single processor core. For actual parallelization, it is recommended to utilize the command line of your operating system, preferably in administrator mode. By making a few adjustments to the source code and assuming the mpiexec file is situated in the folder C:\Program Files\Microsoft MPI\Bin, while the compiled C++ project executable file is located in C:\Users\I-HAVE-NO-TIME-for\source\repos\MPI\x64\Debug\mpi, the command line prompt may resemble the example depicted in Fig. 2.3.



```
1014 + 2026 = 3040
1015 + 2028 = 3043
1016 + 2030 = 3046
1017 + 2032 = 3049
1018 + 2034 = 3052
1019 + 2036 = 3055
1020 + 2038 = 3058
1021 + 2040 = 3061
1022 + 2042 = 3064
1023 + 2044 = 3067
1024 + 2046 = 3070

C:\Users\I-HAVE-NO-TIME-for\source\repos\MPI\
x64\Debug\MPI.exe (process 20796) exited with code 0.
Press any key to close this window . . .
```

Fig. 2.2. Console window of outputting the result of the program for adding two vectors



```
Microsoft Windows [Version 10.0.19044.2728]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd C:\Program Files\Microsoft MPI\Bin

C:\Program Files\Microsoft MPI\Bin>mpiexec C:\Users\I-HAVE-NO-TIME-for\
source\repos\MPI\x64\Debug\mpi
MPI = 64819 ms

C:\Program Files\Microsoft MPI\Bin>mpiexec -n 1 C:\Users\I-HAVE-NO-TIME
-for\source\repos\MPI\x64\Debug\mpi
MPI = 115047 ms

C:\Program Files\Microsoft MPI\Bin>
```

Fig. 2.3. Examples of executing a parallel program on MPI technology using a different number of threads

Additionally

It is recommended to install the Microsoft MPI program in the default folder. Otherwise, you may have problems with the installation of the msmpi package.

2.3. Statements

1. Configure MPI technology on your computer or laptop to enable its functionality (*20% of points*).
2. Adjust the existing code based on the chosen option from the first lab (*50% of points*).
3. Efficiently distribute the workload among processor cores to ensure optimal performance of the operations in accordance the chosen option (*10% of points*).
4. Determine the appropriate number of processes to check the maximum capabilities of your computing device (*10% of points*).
5. Perform a programmatically time-based comparison of the calculation performance across MPI, CUDA(OpenCL), and sequential technologies (*10% of points*).
6. Perform a thorough analysis of the results obtained and provide insightful explanations.

Additional materials

1. Fundamentals of MPI:
https://www.youtube.com/watch?v=c0C9mQaxsD4
2. More complex cases of MPI usage:
https://www.youtube.com/watch?v=q9OfXis50Rg

Questions for independent training

1. What is the purpose of the `MPI_Init` function?
2. What is the difference between the `MPI_Comm_size` and `MPI_Comm_rank` functions?
3. What is a process rank?
4. What does the `MPI_Scatter` function do?
5. What does the `MPI_Gather` function do?
6. What is the purpose of the `MPI_Finalize` function?
7. Which rank is usually used as the main process?

8. What does `MPI_COMM_WORLD` mean?
9. What types of data are available in MPI?
10. What function is used to free previously allocated memory in MPI technology?
11. What of the well-known things does MPI technology imitate?
12. What physical device is designed for MPI computing?
13. What is the usual limitation when running an MPI program in Microsoft Visual Studio?
14. What command should be entered to calculate on exactly two threads?
15. How many threads/cores are used by default to perform calculations when running an MPI program from the command line?
16. Which of the following functions does not exist in MPI: `MPI_Init`, `MPI_Send`, `MPI_Free`?
17. What is `MPI_Status`?
18. What arguments does the `MPI_Init` function receive?
19. What does the `MPI_Recv` function do?
20. What does the `MPI_Reduce` function do?

Laboratory work 3
Compatible high-performance computing on graphics cards and central processors

Goals

1. Learn how to use a fast method of parallelizing computations on processors.
2. Learn how to combine the computing resources of HPC of video cards and processors.
3. Learn ways to analyse different computing technologies.

3.1. Introduction to OpenMP technology

OpenMP, which stands for Open Multi-Processing, is an open standard for parallelizing programs between processor cores. It employs preprocessor directives, or "pragmas", library procedures, and environment variables to facilitate the creation of multi-threaded applications intended for use on multiprocessor systems with shared memory.

Key elements of the standard include structures for creating threads (*parallel* directive), work distribution between threads (*DO/for* and *section* directives), data management (use of the *shared* and *private* memory classes to define variable memory), thread synchronization (*critical*, *atomic*, and *barrier* directives), and runtime support library procedures (such as *omp_get_thread_num*) and environment variables (e.g. *OMP_NUM_THREADS*).

3.2. Creating the first OpenMP project in Microsoft Visual Studio

Developing parallel programs with Microsoft Visual Studio is a convenient option. To create a relatively simplistic program, such as one that adds two vectors with OpenMP parallelization, follow the below-mentioned steps:

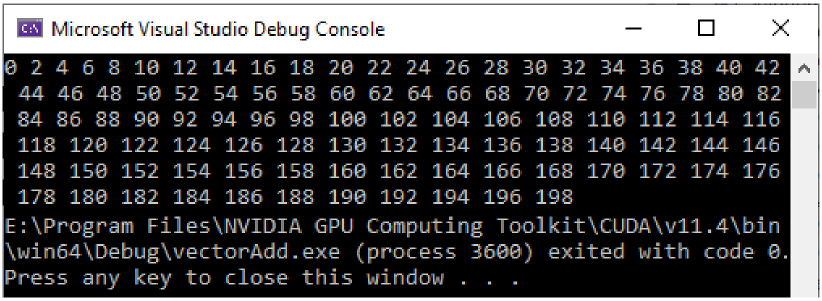
1. Create an empty C++ console project.
2. Place the following code in a *.c or *.cpp file:

```
#include <omp.h>
#include <iostream>

int main()
{
  const long n = 100;
  int* a = (int*)malloc(sizeof(int) * n);
  int* b = (int*)malloc(sizeof(int) * n);
  int* c = (int*)malloc(sizeof(int) * n);
  for (int i = 0; i < n; i++)
  {
    a[i] = i;
    b[i] = i;
  }

  #pragma omp parallel for
  for (int i = 0; i < n; i++)
  {
    c[i] = a[i] + b[i];
  }

  for (int i = 0; i < n; i++)
  {
    printf("%i ", c[i]);
  }
}
```

An example of the output is shown in Fig. 3.1.

3.3.Statements

The tasks involve the integration of CUDA/OpenCL and OpenMP technologies to enhance performance, compare calculation times across multiple technologies, and analyze the obtained results.



```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42
 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82
 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116
 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146
 148 150 152 154 156 158 160 162 164 166 168 170 172 174 176
 178 180 182 184 186 188 190 192 194 196 198
E:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.4\bin
\win64\Debug\vectorAdd.exe (process 3600) exited with code 0.
Press any key to close this window . . .
```

Fig. 3.1. An example of executing a program code using OpenMP technology

1. Adjust the existing code based on your chosen option from the first lab (10% *of points*).
2. Integrate the capabilities of CUDA/OpenCL and OpenMP technologies effectively to combine their power and efficiency (*50% of points*).
3. Compare the time performance of computations using MPI, OpenMP, CUDA (OpenCL) and sequential technologies programmatically (*20% of points*).
4. Provide a comprehensive analysis of the results and explain them in detail to demonstrate your understanding (*20% of points*).

Additional materials

1. Video tutorials about OpenMP:
https://www.youtube.com/watch?v=SLnh7yS52-I&list=PL3xCBlatwrsWhsdHq3JFJiuQ60gHzYtFU&index=1.
2. Fundamentals of OpenMP:
https://www.youtube.com/watch?v=YdHv_2AT4GI&list=PLQsgurGJM5piDxUkHX15V6xrAFHzzWfMh&index=2.

Questions for independent training

1. What is the purpose of the `omp.h` library?
2. How can the `#pragma omp parallel` fragment affect on program execution?
3. How does the `omp_get_thread_num()` function work?
4. What is the difference between `shared` and `private` variables?
5. What does `#pragma omp barrier` mean?
6. What does `#pragma omp critical { ... }` mean?
7. How does the code controlled by the `#pragma omp for { ... }` directive work?
8. What does `the #pragma omp parallel private(partial_Sum) shared(total_Sum) mean`?
9. What is the complexity of combining several HPC technologies?
10. What might be the compiler's reaction if you use `#pragma omp` without connecting the `omp.h` library?
11. What construction is used to create threads?
12. What structures exist for distributing work between threads?
13. What constructions are used for determining the memory class of variables?
14. What are the ways to synchronize threads?
15. What is *OMP_NUM_THREADS*?
16. What technology is more efficient: MPI or OpenMP?
17. What does the `#pragma omp master` string mean?
18. What steps should be performed to ensure the ability to use OpenMP?
19. What is the difference between the `#pragma omp parallel` and `#pragma omp` expressions in the case of adding elements of two arrays?
20. How does the compiler react on the `#pragma parallel omp parallel` entry?

Bibliography

Primary

1. Бойчура М. В., Шатний С. В., Шатна А. В. 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]. Рівне : НУВГП, 2023. 49 с.
2. Пасічник В. В., Лупенко С. А., Луців А. М. Паралельні та розподілені обчислення. Львів : Магнолія 2006, 2024. 664 с.
3. Hwu W.-M.W., Kirk D.B., Hajj I.E. Programming Massively Parallel Processors: A Hands-on Approach. 4th ed. Cambridge : Morgan Kaufmann, 2022. 580 p.
4. Deakin T., Mattson T.G. Programming Your GPU with OpenMP: Performance Portability for GPUs. Cambridge : The MIT Press, 2023. 336 p.
5. Barlas G. Multicore and GPU Programming: An Integrated Approach. 2nd ed. Burlington : Morgan Kaufmann, 2022. 1024 p.
6. Chopp D.L. Introduction to High Performance Scientific Computing. Philadelphia : SIAM - Society for Industrial and Applied Mathematics, 2019. 453 p.
7. Dash N. Ultimate Parallel and Distributed Computing with Julia For Data Science. Delhi : Orange Education Pvt Ltd, 2024. 484 p.
8. Peters T. Parallel Python with Dask: Perform distributed computing, concurrent programming and manage large dataset. Delhi : GitforGits, 2023. 172 p.

9. Nelli F. Parallel and High Performance Programming with Python. Delhi : AVA, 2023. 392 p.

10. Scott L.R., Clark T., Bagheri B. Scientific Parallel Computing. Princeton : Princeton University Press, 2021. 392 p.

Secondary

11. 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.

12. 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.

13. Overview - CUDA Python 12.5.0 documentation. URL: https://nvidia.github.io/cuda-python/overview.html (Last accessed: 01.06.2024).

14. CUDA C++ Best Practices Guide. Release 12.5. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (Last accessed: 01.06.2024).

15. The programming guide to the CUDA model and interface. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Last accessed: 01.06.2024).

16. The OpenCL™ C Specification. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf (Last accessed: 01.06.2024).

17. MPI: A Message-Passing Interface Standard. Version 4.1. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (Last accessed: 01.06.2024).

18. Using MPI with C - RC University of Colorado Boulder documentation. URL: https://curc.readthedocs.io/en/latest/programming/MPI-C.html (Last accessed: 01.06.2024).
19. Home - OpenMP. URL: http://www.openmp.org (Last accessed: 01.06.2024).
20. OpenMP | LLNL HPC Tutorials. URL: https://hpc-tutorials.llnl.gov/openmp/ (Last accessed: 01.06.2024).
21. OpenMP Application Programming Interface. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (Last accessed: 01.06.2024).
22. Using OpenMP with C - RCU of Colorado Boulder documentation. URL: https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html (Last accessed: 01.06.2024).

Useful

23. MPI Forum. URL: https://www.mpi-forum.org/ (Last accessed: 01.06.2024).
24. Ukraine - Distributed Computing Team. URL: http://distributed.org.ua (Last accessed: 01.06.2024).