Heterogeneous Programming

C++ with CUDA, HIP, openCL, SYCL & Intel oneAPI

By Pouya Ebadollahyvahed

14 Jan. 2025

Developing Codes to run on CPU, GPU, NPU & FPGA

Sample Codes in

C++17 & C++20

Heterogeneous Programming

a Skill for Today & Tomorrow

By: Pouya Ebadollahyvahed

Heterogeneous Programming is Writing Programs to be Executed on Multiple Types of Processors:

1- CPUs
2- GPUs
3- NPUs
4- FPGAs

1. Table of Contents

2. What is Heterogeneous Programming & Why is it Important?

Fifteen or twenty years ago, developing codes to run on a machine (=computer) was much easier than today. Since most of the machines had single CPU (or several CPUs but the same type), developers just needed to pick-up a compiler, write their codes and then compile and run them.

Although GPUs has been primarily designed to create computer graphics (& videos) but in mid-90s programmers realized they can use GPUs as general-purpose processors and run ordinary calculations on them (for example, matrix multiplication) and that is why these processors are also being called GP-GPU (General Purpose GPU).

GPUs architecture is designed based on many (thousands of) computational cores while each core is much simpler and runs at lower frequency rather than a CPU core, so to use GPU's processing power in efficient way, computational algorithms should be designed to massively **run in parallel** and it is one the major differences between programming models for CPUs and GPUs.

These days most of the machines (laptops, desktop, servers and HPC nodes) have several types of processors on them (CPUs, GPUs & NPUs) so a **software system may run on several processors at the same time which means a software program can run some processes on CPU cores and some other processes on GPU cores. Developing such software, is called Heterogeneous Programming.** It is called Heterogeneous Programming because, programmer needs to use different programming models, techniques and libraries to run codes on different processor types (like CPUs and GPUs).

Heterogeneous Computing refers to a computational system (like a desktop computer or a server or an HPC node) which has multiple types of processors (likes CPUs and GPUs).

During last few years and by emerging AI (Artificial Intelligence) and deep learning (Neural Networks), demand for processing power increased a lot so utilizing all types of available processors on machines and developing codes to run both on CPUs & GPUs (**Heterogeneous Programming**) is getting more and more important. I personally believe that **in near future <u>Heterogeneous Programming</u> will turn to a mandatory requirement for "<u>Process Intense</u>" programs like AI software**.

Following table lists the most commonly found processors on computational machines (laptops, desktops, servers and HPC nodes) and provides simple comparison between them:

Processor	Advantage	Major Market Players	Application	Product Example	Cores
CPU	**Low Latency**	Intel AMD	Desktop	Core i7 14700 Ryzen-9 9900X	4 – 20 (typically)
			Server	Xeon Gold 6544 Epyc 9845	20 – 200 (typically)
GPU	**High Throughput**	Nvidia AMD Intel	Desktop	Nvidia RTX 4070 Radeon RX 7900 Iris Xe	Thousands
		Nvidia AMD Intel	Server (HPC)	B200 (Blackwell) MI300X Flex Series	Tens of Thousands
NPU AI Accelerator	**Low Power**	Intel AMD NVidia Qualcomm & …	Desktop Mobile	Inside Intel Ultra Core CPUs, Gaudi 3 Inside AMD Ryzen 8000G series CPUs Inside Nvidia GPUs (Tensor Cores) Hexagon NPU inside Snapdragon 855	
FPGA	**Combinational Circuit, any logic can be designed on it**	AMD (former XILINX) Intel (former ALTERA)	Aerospace Mining Realtime Apps & …	Spartan6 XC6SLX25T Cyclone 10 10CX105	-

There are Two Approaches for Heterogeneous Programming (Developing Codes for Several Processor Types)**:**

- Using several sets of APIs, programming models, libraries & frameworks to support all types of processors in their own native way. For each processor, its own library is used (following table).
- **Unified Way**: Using cross-platform abstraction layers and frameworks to write just one version of code which can be compiled & run on different processor types (e.g. SYCL: discussed in chapter 5)

How to Develop Codes to Run on CPUs:

Developing codes to run on CPUs is extremely straight forward since almost all of standard functions & libraries in programming languages, OS APIs and system calls are developed to run on CPUs only.

How to Develop Codes to Run on GPUs:

The most commonly used libraries, APIs & frameworks to develop codes to run on GPUs can be listed as:

Library, API or Framework	**Logo**	**Initial Release**	**Open or Closed Source?**	**Released By**	**Supported GPUs**	**Also Supports….**
CUDA	NVIDIA CUDA	2007	Closed	NVidia	Nvidia	-
HIP	AMD ROCm	2016	Open	AMD	AMD	Nvidia GPUs
oneAPI	1 oneAPI	2020		Intel	Intel	Theoretically all Processors through SYCL

In next chapters we discuss about above-mentioned libraries, frameworks and APIs.

3. Programming for Nvidia GPUs with **CUDA**

CUDA is a parallel computing platform and API developed by NVidia to support code development **to run on Nvidia GPUs**. It supports C/C++, Fortran and Python languages.

CUDA library can be freely downloaded from[1]: https://developer.nvidia.com/cuda-downloads

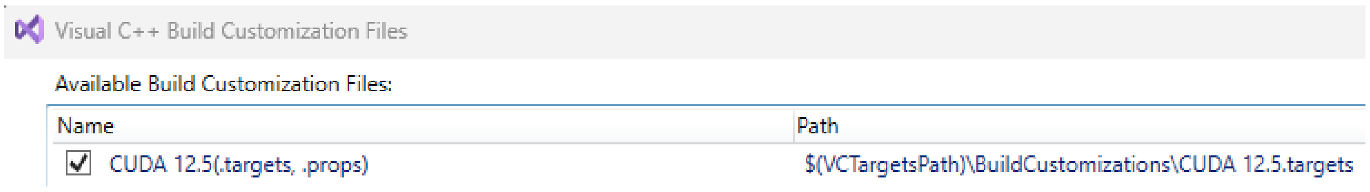
How can I check if my machine has CUDA compatible GPU (& drivers) or no?

After installing CUDA, you may run one of its command line tools named **deviceQuery**. This command, lists all CUDA compatible GPUs & their specifications which are installed on the machine. This tool is usually located in this path: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.5\extras\demo_suite

After installing CUDA, how can I use it?

After installing CUDA & writing codes (like the example code of next page) there are 2 ways to compile it:

1- Use CUDA native compiler (**nvcc**) in command line
2- CUDA automatically integrates into Visual Studio (if it is already installed). In VS, Create CUDA type project or add CUDA to existing project by right clicking on "Project" in the "Solution Explorer" then choose "Built Dependencies | Built Customizations..." and check "CUDA" (following screenshot)


Visual C++ Build Customization Files

Available Build Customization Files:

Name	Path
✓ CUDA 12.5(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\CUDA 12.5.targets

[1] **Note:** Prior to installing CUDA, you need to install proper driver for your GPU (= Video Adaptor / Graphics Card). Usually installing Nvidia drivers on Windows is straight forward also on some distributions of Linux like RHEL it is easy but installing it on some distributions like Debian is a bit challenging.

Let’s start with an actual easy example in CUDA:

```
// Pre-requirement     : Installing CUDA and integrating it to Visual Studio
//
// On Windows          : Visual Studio,
//                       Create CUDA type project or add CUDA to project:
//                       Right Click on Project in the Solution Explorer then
//                       Built Dependencies | Built Customizations... | check CUDA
// Note                : file name should have .cu extension

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>

__global__ void thread_in_GPU (int* nums_in_GPU, int *subTot_in_GPU)
{
    int threadNo = threadIdx.x;
    subTot_in_GPU[threadNo] = nums_in_GPU[threadNo*2]+nums_in_GPU[threadNo*2 + 1];
}

int main()
{
    int nums[] = { 33, 12, -5, 15 }, subTot[2], *nums_in_GPU, *subTot_in_GPU;
    int sNums{std::size(nums)*sizeof(int)}, sSubTot{std::size(subTot)*sizeof(int)};


    if (cudaSuccess != cudaSetDevice(0)) return -1;//Select a GPU to run code on it
    if (cudaSuccess != cudaMalloc((void**)&nums_in_GPU, sNums)) return -2;
    if (cudaSuccess != cudaMalloc((void**)&subTot_in_GPU, sSubTot)) return -3;

    if (cudaSuccess != cudaMemcpy(nums_in_GPU, nums, sNums,cudaMemcpyHostToDevice))
        return -4;

    thread_in_GPU <<<1, 2 >>> (nums_in_GPU, subTot_in_GPU);//Creates 1*2= 2 THREADS

    if (cudaSuccess != cudaDeviceSynchronize()) return -5; // Wait threads finish

  if (cudaSuccess!=cudaMemcpy(subTot,subTot_in_GPU,sSubTot,cudaMemcpyDeviceToHost))
        return -6;

    cudaFree(nums_in_GPU);
    cudaFree(subTot_in_GPU);

    cudaDeviceReset();

    int grand_total{ subTot[0] + subTot[1] };

    std::cout << "Sum of all numbers is: " << grand_total << std::endl;
}

// Printed Result: 55
```

Since above code is written using CUDA framework & API, it runs only on Nvidia GPUs.

Here is how the code in previous page works:

1- The function `thread_in_GPU()` is a thread function which runs on GPU (also called kernel)

   a. `__global__` identifier shows that the function is called from CPU code but runs on GPU

   b. `__device__` identifier shows that the function is called from GPU code and runs on GPU

   c. `__host__` identifier shows that the function is called from CPU code and runs on CPU

2- GPUs have their own memory (usually installed on Graphics Card), it is called "**Device Memory**"

3- The memory which is installed on motherboard (mainboard) and used by CPU is called "**Host Memory**"

4- To run a code on GPU, all of the GPU accessed variables, arrays & objects should be in the Device Memory

   a. To store a variable, array or object in the "Device Memory" we need to do memory allocation

   b. `cudaMalloc()` allocates memory in Device Memory

   c. We can copy variables, arrays, buffers & objects from host memory to device memory & vice versa

   d. `cudaMemcpy()` copies a memory buffer from device to host memory & vice versa

   e. `cudaFree()` frees previously allocated buffer in the Device Memory

5- Unlike CPUs, Nvidia GPUs comprises of several SMs (Stream Processors) and Each SM has several cores

   a. You may consider SM as a small processor

   b. For example, my machine has 36 SMs & each SM has 128 cores so in total I have 4608 cores

   c. In CUDA architecture multi-thread codes comprises of Thread Blocks and Threads per Block

6- Creating threads in CUDA is simple, and can be done like this:

   a. `Thread_function_name <<<N, M>>> (arg1, arg2, …)`

   b. Above N is number of thread blocks and M is number of threads per block

   c. Above total number of created threads will be (N * M)

   d. Maximum number of N is 1024

7- `cudaDeviceSynchronize()` is similar to `join()` function, it waits for all threads to finish execution

8- Function `cudaSetDevice()` selects which GPU the code should run on.

   a. It is usually the first function which is called in CUDA codes

   b. This function has one integer argument where "0" means 1st GPU, "1" means 2nd GPU and so on

After compiling CUDA codes, you will receive an executable file but if you want to run it on another Windows machine which CUDA has not been installed on it, even by choosing “static linking” in Visual Studio, you still need to keep “**cudart64_12.dll**” file (from CUDA package) beside your executable file.

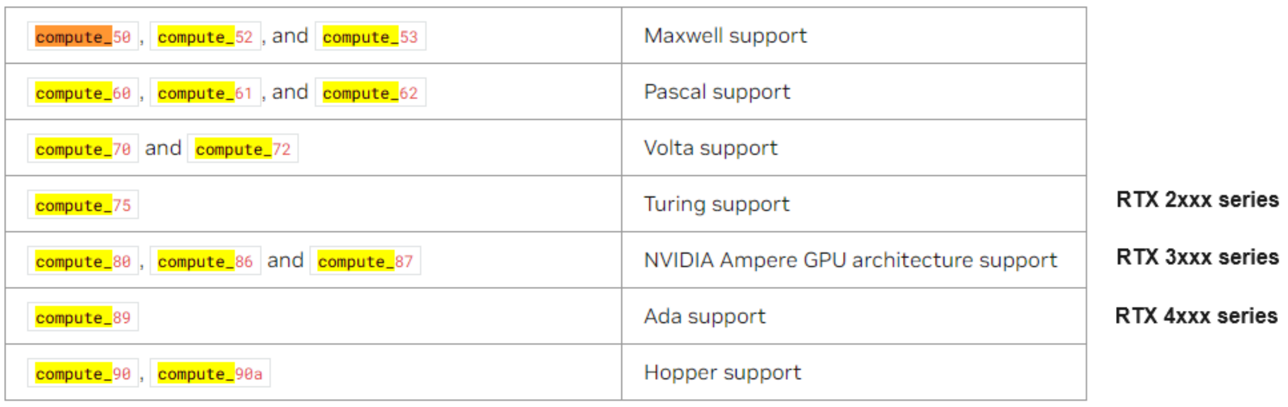
Note: when you are creating a CUDA project, you may choose minimum version of the architecture of target GPU. By default the target is version 52 which belongs to very old GPUs, but you can change it to version 80 (as an example). In this case your program (the exacutable file) runs only on GPUs which has architecture version 80 or higher and it will not run on older GPUs but the benefit is that you can use CUDA functions which are added in these newer versions.

For example, function `__nanosleep()` is added in compute version 70 so if you want to use this function in your code then you need to change the compute version in your project to version 70 or higher and by doing this, your program (executable file) won’t run on older GPUs like Maxwell & Pascal.

To change compute architecture version in Visual Studio Projects:

- Right Click on Project -> Properties | Configuration Properties | CUDA C/C++ | Device
- In “Code Generation” change values to “**compute_80,sm_80**” or any other desired values
- **compute_XX** is virtual architecture of target platform and **sm_XX** is real architecture

Following table shows architecture version of GPUs and their related GPU models (names):



`compute_50`, `compute_52`, and `compute_53`	Maxwell support	
`compute_60`, `compute_61`, and `compute_62`	Pascal support	
`compute_70` and `compute_72`	Volta support	
`compute_75`	Turing support	**RTX 2xxx series**
`compute_80`, `compute_86` and `compute_87`	NVIDIA Ampere GPU architecture support	**RTX 3xxx series**
`compute_89`	Ada support	**RTX 4xxx series**
`compute_90`, `compute_90a`	Hopper support	

Hardware architecture of GPUs are based on many (thousands of) computational cores so GPU programming model is “parallel programming” and programmers who develop code for GPUs are usually well familiar with parallel programming techniques although when developing codes for GPUs (at least with CUDA), one important thing need to be considered: GPUs use their own memory (device memory) so when we are dealing with variables, objects, buffers and arrays, all of them are located in “device memory” (not in the main memory of machine which is called host memory) and here comes a big problem: **many C++ standard mechanisms & STL algorithms like atomic variables and mutexes cannot be used**, so synchronization between GPU threads gets a bit challenging and only CUDA functions and mechanisms can be used for this purpose.

CUDA, Pros & Cons:

Pros:

- Developed to support GPUs of only one vendor: It has minimal over-head & **High Performance**
- Developed by the hardware designer & vendor: Almost all features of GPUs are supported
- New versions are continuously released to give early support to the newest introduced GPUs
- Rich in documentations and programming forums

Cones:

- Only supports Nvidia GPUs

Why CUDA is so much Popular Among GPU Programmers?

In the above list, you may find only 1 disadvantage for CUDA: it supports only Nvidia GPUs. According to a new GPU sales report for the first quarter of 2024 compiled by Jon Peddle Research (JPR)[2], Nvidia’s market share is more than 80%, so this disadvantage can be ignored in front of above mentioned 4 advantages.

[2] Source: https://www.extremetech.com/gaming/analyst-nvidia-gpu-market-share-now-at-88-amd-with-12#:~:text=Analyst%3A%20Nvidia%20GPU%20Market%20Share,%2C%20AMD%20With%2012%25%20%7C%20Extremetech

4. Programming for AMD GPUs with HIP

If you want to use GPUs as GP-GPUs (General Purpose GPUs) and run ordinary calculations on them like matrix multiplication or AI processes (e.g. training a neural network), you need to have:

- A proper hardware platform (GPU itself)
- Software libraries, frameworks and APIs to support programming on that hardware

Nvidia released CUDA in 2007 to enable programmers to develop codes to run on Nvidia GPUs. Since they already had great GPUs, programmers quickly picked-up CUDA as a trivial choice for GPU programming. AMD started manufacturing GPUs since 2006 (since they acquired famous Canadian company of ATI) but they didn't have proper software package to support programmers to develop codes for AMD GPUs (similar to CUDA and in that level). Since CUDA was (and is) only supporting Nvidia GPUs, it couldn't be used to develop codes on AMD GPUs.

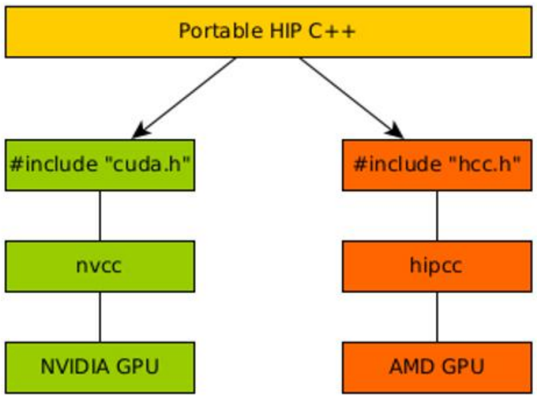
At last, in 2016 AMD also released their own rich software package to enable programmers to develop codes to run on AMD GPUs. This package is called **ROCm**, and it includes a programming model named **HIP** which is very similar to CUDA. **ROCm** also includes some other tools and libraries (like for HPC).

When **HIP** is released, it was almost 9 years that programmers got used to program in CUDA, so convincing programmers to adopt a new programming model (HIP) and to switch to it was difficult, for this purpose AMD chose three smart initiatives:

1- AMD made **ROCm** Open source (CUDA is closed source)

2- HIP API functions have very similar names & prototypes to CUDA API functions (easy to learn)

3- HIP not only supports coding on AMD GPUs but also supports coding on Nvidia GPUs (by CUDA)



Some examples of HIP API functions and their equivalent functions in CUDA:

```
__host__cudaError_t cudaSetDevice (int device)
hipError_t           hipSetDevice (int deviceId)

__host__cudaError_t cudaDeviceReset (void)
hipError_t           hipDeviceReset (void)

__host____device__cudaError_t cudaMalloc (void **devPtr, size_t size)
hipError_t                     hipMalloc (void **ptr   , size_t size)

__host__cudaError_t cudaMemset (void *devPtr, int value, size_t count)
hipError_t           hipMemset (void *dst   , int value, size_t sizeBytes)

__host__cudaError_t cudaMemcpy(void *dst, const void *src, size_t count    ,cudaMemcpyKind kind)
hipError_t           hipMemcpy(void *dst, const void *src, size_t sizeBytes, hipMemcpyKind kind)

__host____device__cudaError_t cudaFree (void *devPtr)
hipError_t                     hipFree (void *ptr)
```

5. Unified Heterogeneous Programming with openCL & SYCL

As it is mentioned on page 5:

There are Two Approaches for Heterogeneous Programming (Developing Codes for Several Processor Types)**:**

- Using several sets of APIs, programming models, libraries & frameworks to support all types of processors in their own native way. For each processor, its own libraries are used (like CUDA).
- **Unified Way**: Using cross-platform abstraction layers, frameworks & high-level APIs to write just one version of code which can be compiled & run on different processor types (like using SYCL).

Examples of the first approach are **CUDA** & **HIP** those we discussed in previous two chapters (3 & 4) but this approach has a great disadvantage: programmers need to learn a new programming model, a new set of API functions and get familiar with a new library for each brand of GPU and for each processor type. Obviously, it takes a lot of time & effort from programmers, also it has poor portability. It has poor portability because when a code needs to be ported from GPU of a vendor to run on GPU of another vendor, it takes a long time and a lot of effort to apply all required changes in source codes[3].

To cope with this challenge several cross-company groups are founded to **develop (and ratify) interoperability standards and unified methods** to enable programmers to write just one single version of code but to compile and run it (theoretically) on any type of processor from any vendor. The goal was to define abstraction layers and high-level libraries to make it happen.

[3] It worth to mention that there are some software tools to migrates codes from CUDA to HIP & from CUDA to oneAPI.

In year 2000, some big brands like ATI Technologies (manufacturer of Radeon GPUs, now part of AMD), Intel, SGI and Sun Microsystems founded an open, non-profit group which is called **Khronos Group** and the aim was to **develop open interoperability standards in the domain of Computer Graphics** and Parallel Computing.

Some of distinguished members of Khronos Group are: AMD, Apple, ARM, Google, Intel, Nvidia, Qualcomm, Samsung, Sony, Microsoft and Red Hat (last two are contributor members).

In 2009, **Khronos Group** introduced **openCL** (Open Computing Language) which is a framework for unified heterogeneous programming. It is developed to enable programmers to develop single version of codes to compile (& run) them on CPUs, GPUs, DSPs & FPGAs. openCL is a **C-like language** which is based on C99, **it also provides an API** for C/C++ languages (also third-party APIs are developed to be used in Python, Java, Perl and .NET).

At the time of writing this article, a long list of GPUs and processors from vendors like Nvidia, AMD, Intel, Qualcomm, Samsung and SPI are supporting openCL.

```
// Source of the following code is:
//                                   https://en.wikipedia.org/wiki/OpenCL
// Following is an example for openCL

// Multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
__kernel void matvec(__global const float *A, __global const float *x,
                     uint ncols, __global float *y){
    size_t i = get_global_id(0);               // Global id, used as the row index
    __global float const *a = &A[i*ncols];     // Pointer to the i'th row
    float sum = 0.f;                           // Accumulator for dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

openCL was a great step forward in unified heterogeneous programming but it has a small problem: it is a low-level framework so working with it is not easy; to solve this problem and to make the programming easier, in 2014 Khronos Group released a high-level programming model, named **SYCL** based on **openCL**.

SYCL (**Sy**stem wide **C**ompute **L**anguage) is flavor of the day and recently grabbed great attention of programmers and software industry professionals & companies.

In 2014, Khronos Group released a **single source** eDSL (embedded Domain Specific Language) based on pure C++17 which is called **SYCL**.

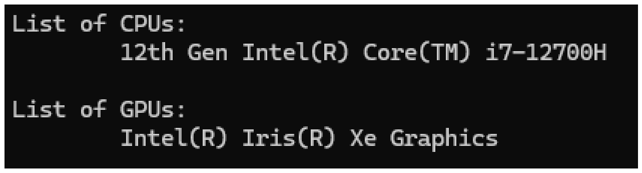
SYCL is a high-level programming model for unified heterogeneous programming, in other words developers can write codes in SYCL that can be compiled and run on several types of CPUs, GPUs & NPUs.

Following is a simple example of SYCL code which can be compiled in Visual Studio or by Intel DPC++:

```
// Pre-requirement      : Installing Intel oneAPI (one of SYCL implementations)
//
// On Windows option 1 : icpx -fsycl  file_name.cpp (see notes on the next page)
//
// On Windows option 2 : Visual Studio
//                        Create “DPC++ Console” type project then in the
//                        Solution Explorer, right click on the project and in the
//                        "Project Properties | Configuration | VC++ Directories”
//                        Add C:\Program Files(x86)\Intel\oneAPI\2025.0\include
//                                to "Include Directors" and
//                        Add C:\Program Files(x86)\Intel\oneAPI\2025.0\lib
//                                to "Library Directors"
//

#include <sycl/sycl.hpp>
#include <iostream>

int main()
{
        auto cpus = sycl::platform(sycl::cpu_selector_v).get_devices();
        auto gpus = sycl::platform(sycl::gpu_selector_v).get_devices();


        std::cout << "List of CPUs:" << std::endl;
        for (auto cpu : cpus) {
            sycl::queue qCPU(cpu);
            std::cout<<"\t"<<qCPU.get_device().get_info<sycl::info::device::name>();
            std::cout << std::endl;
        }

        std::cout << std::endl << "List of GPUs:" << std::endl;
        for (auto gpu : gpus) {
            sycl::queue qGPU(gpu);
            std::cout<<"\t"<<qGPU.get_device().get_info<sycl::info::device::name>();
            std::cout << std::endl;
        }
}
```

Example code of the previous page gets list of all supported CPUs and GPUs on the system and prints their names on the screen. In the next chapter, we will have a more advanced SYCL example to show how to run processes on these CPUs & GPUs.

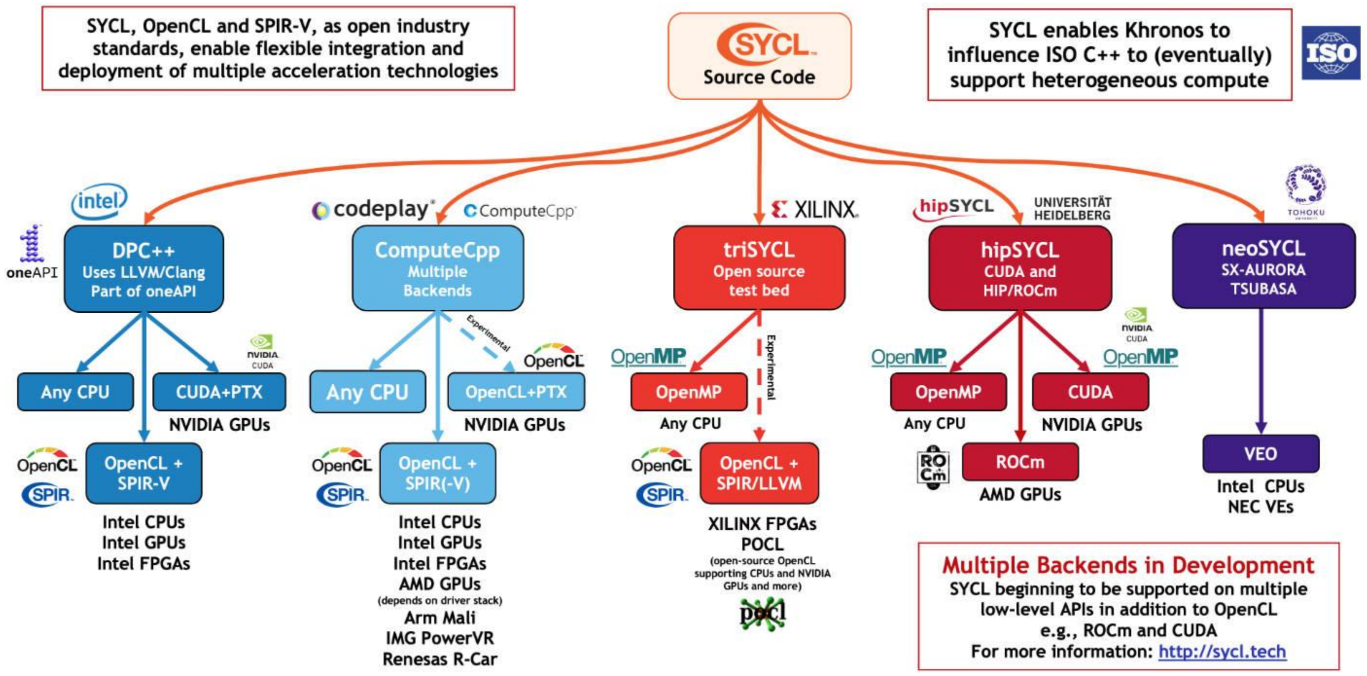
Opposite screenshot is an actual output of the example code in the previous page[4]:



```
List of CPUs:
        12th Gen Intel(R) Core(TM) i7-12700H

List of GPUs:
        Intel(R) Iris(R) Xe Graphics
```

To run the example code of the previous page, you need to compile it with a SYCL supported C++ compiler. There are several C++ compilers which implemented and support SYCL. In this writing we use DPC++ which is Intel's modern C++ compiler and is part of Intel oneAPI package.

After installing Intel oneAPI, if you already had Visual Studio (2019 or 2022) it will be integrated to Visual Studio and you will be able to write and compile your codes in Visual Studio or do it directly with Intel DPC++. Following picture shows some of SYCL implementations; the left one is DPC++ (Intel oneAPI).



SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies
SYCL Source Code
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
ISO
intel
oneAPI
DPC++ Uses LLVM/Clang Part of oneAPI
Any CPU
CUDA+PTX
NVIDIA GPUs
OpenCL + SPIR-V
Intel CPUs
Intel GPUs
Intel FPGAs
codeplay
ComputeCpp
ComputeCpp Multiple Backends
Experimental
Any CPU
OpenCL+PTX
NVIDIA GPUs
OpenCL + SPIR(-V)
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depends on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car
XILINX
triSYCL Open source test bed
OpenMP
Any CPU
Experimental
OpenCL + SPIR/LLVM
XILINX FPGAs
POCL
(open-source OpenCL supporting CPUs and NVIDIA GPUs and more)
hipSYCL
UNIVERSITÄT HEIDELBERG
hipSYCL CUDA and HIP/ROCm
OpenMP
Any CPU
CUDA
NVIDIA GPUs
ROCm
AMD GPUs
neoSYCL SX-AURORA TSUBASA
VEO
Intel CPUs
NEC VEs
Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL
e.g., ROCm and CUDA
For more information: http://sycl.tech

Figure 1 Source: https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

[4] You can also get this list by running "**sycl-ls"** command in "Intel oneAPI command prompt for Intel 64 for Visual Studio 2022"

To compile **SYCL** based codes, you can download and install **Intel oneAPI** toolkit from:

https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html

After installing Intel oneAPI, if you already have Visual Studio 2022 or 2019, it integrates to VS and after that you will have 2 options to write and compile your SYCL codes:

1- **In Visual Studio**:
   a. Create "DPC++ Console" type project
   b. In the Solution Explorer, right click on the project and in the "Project Properties | Configuration | VC++ Directories"
   c. Add "**C:\Program Files(x86)\Intel\oneAPI\2025.0\include**" to "**Include Directors**"
   d. Add "**C:\Program Files(x86)\Intel\oneAPI\2025.0\lib**" to "**Library Directors**"

2- **Directly by Intel DPC++ Compiler**:
   a. In Win startup, launch "**Intel oneAPI command prompt for Intel 64 for Visual Studio 2022**"
   b. In the opened command prompt type: `icpx –fsycl  file_name.cpp`

As it can be seen in the picture of previous page, Intel DPC++ compiler is not the only implementation of SYCL but in April 2024 & by releasing oneAPI version 2024.1, Intel announced, their DPC++ is the first C++ compiler which fully supports SYCL 2020 specification.

To write codes using oneAPI, programmers may choose two approaches:

1- Directly write their SYCL codes in C++
2- Use oneAPI provided APIs & use its rich set of libraries like oneDPL to code easier and faster

Note: Although in early days of **SYCL**, it was built based on openCL but in 2020, its developers mentioned that it has no reference to openCL and the term "SYCL" is not an acronym **any more**.

6. Intel oneAPI Base Toolkit[5]

Intel oneAPI Base Toolkit is a modern and rich toolkit developed for programmers and software developers, which includes:

1- Modern C/C++ compiler fully supporting **SYCL**[6]

   a. Intel's modern compilers are `icx` (for C) and `icpx` (for C++)

      i. Intel's compilers always show the best performance even for ordinary codes

      ii. They are replacing Intel's classic compilers: `icc` & `icl`

2- A rich set of libraries, mainly for parallel computing

   a. Examples: oneDPL and oneTBB

oneAPI can be installed both on Linux and on Windows operating systems while one of the main goals behind developing it was to compete with Nvidia's CUDA.

At the heart of oneAPI, is Intel's modern C/C++ compiler which is called **DPC++** (Data Parallel C++). It is built on LLVM's Clang front-end[7]. **LLVM** is a project started in 2000 in the University of Illinois to develop **interoperable and cross platform compiler toolchains and compiler reusable modules**.

One of the most popular outcomes of LLVM project is Clang compiler which beside GNU's `gcc/g++`, is commonly used in UNIX-like systems (like BSD & Linux) for compiling C/C++, Objective-C and Objective-C++ codes. Microsoft also has been supporting LLVM since Visual Studio 2019 (V16.2) for openMP and some other modules.

Sometimes Visual Studio's IntelliSense doesn't Detect SYCL Classes but Codes Compile Correctly

[5] Intel oneAPI has other toolkits like for **HPC** but in this writing we discuss only about their Base Toolkit
[6] In April 2024 & by releasing oneAPI V 2024.1, Intel announced, DPC++ is the1st C++ compiler which fully supports SYCL 2020
[7] Source: https://www.intel.com/content/www/us/en/developer/articles/technical/getting-to-know-llvm-based-oneapi-compilers.html

Almost a year ago a new technology consortium is formed by participation of some big brands like **Intel**, Google, ARM, Qualcomm, Samsung, Imagination, and VMware to add more energy to **oneAPI** project. This consortium is called **UXL** (Unified Acceleration Foundation). UXL Foundation and Khronos Group on June 10, 2024, announced their collaboration on the SYCL Open Standard for C++ Programming of AI, HPC, and Safety-Critical Systems.

It is already described on page 17, **how to compile and run SYCL based codes either in Visual Studio** or directly by Intel's modern DPC++ compiler in command line. This procedure also can be used to compile and run any other oneAPI based project in C++.

On the next page you can find an easy example which is written in C++17. This code uses **SYCL** and **oneDPL** library from **oneAPI** to sort an array.

The beauty of this code is that, **it can run both on CPU & GPU** only by changing value of a single variable.

Here is how it works:

1- In the first line after `main` there is a Boolean variable named `run_on_cpu`
   a. If programmer sets it to `true` the sorting algorithm runs on **CPU**
   b. If programmer sets it to `false` the sorting algorithm runs on **GPU**
2- This program has 2 sections
   a. In section 1, all of the installed and available CPUs and GPUs are detected, and based on the value of `run_on_cpu` the 1st CPU or the 1st GPU is chosen to run the code
   b. In section 2, the sorting algorithm (`oneapi::dpl::sort` function) runs on a device (processor) which is chosen in the previous section.

```
// Pre-requirement     : Installing Intel oneAPI
//
// On Windows option 1 : icpx -fsycl  file_name.cpp (see notes on the next page)
//
// On Windows option 2 : Visual Studio
//                       Create “DPC++ Console” type project then in the
//                       Solution Explorer, right click on the project and in the
//                       "Project Properties | Configuration | VC++ Directories”
//                       Add C:\Program Files(x86)\Intel\oneAPI\2025.0\include
//                               to "Include Directors" and
//                       Add C:\Program Files(x86)\Intel\oneAPI\2025.0\lib
//                               to "Library Directors"
//

#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <sycl/sycl.hpp>
#include <iostream>
#include <vector>

int main()
{
	bool run_on_cpu{ false }; // "true" to run on CPU, "false" to run on GPU

	sycl::queue queue_device;

	if (true == run_on_cpu) // Choosing 1st CPU to Run Algorithms on it
	{
		auto cpus = sycl::platform(sycl::cpu_selector_v).get_devices();
		if (cpus.size() < 0) {
			std::cout << "no CPUs Found to Run on" << std::endl;
			return -1;
		}
		sycl::queue queue_CPU(cpus[0]); // 1st CPU
		queue_device = queue_CPU;
	}
	else // Choosing 1st GPU to Run Algorithms on it
	{
		auto gpus = sycl::platform(sycl::gpu_selector_v).get_devices();
		if (gpus.size() < 0) {
			std::cout << "no GPUs Found to Run on" << std::endl;
			return -2;
		}
		sycl::queue queue_GPU(gpus[0]); // 1st GPU
		queue_device = queue_GPU;
	}

	std::vector<int> nums{ 9, 4, 0, 3, 1, 6, -4, 7 };

	oneapi::dpl::sort(oneapi::dpl::execution::make_device_policy(queue_device),
	                  nums.begin(), nums.end());

	for (auto n : nums)
		std::cout << " " << n;
	std::cout << std::endl;
}
// Printed Result: -4 0 1 3 4 6 7 9
```

Above example code, uses SYCL & oneDPL to sort an array on CPU or GPU (by choice of programmer)

7. Test platform (Hardware & Software Specification)

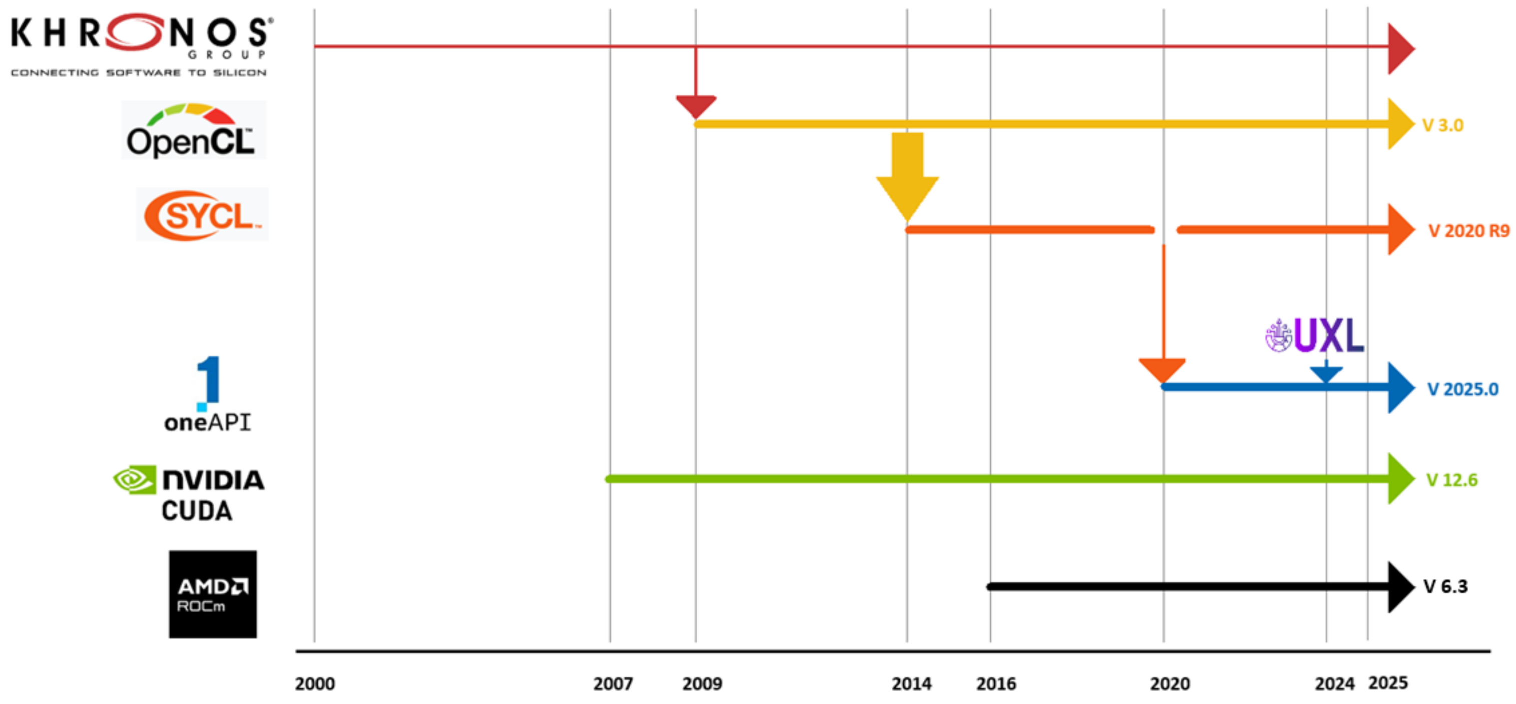
Machine	Laptop Asus TUF Gaming F15
CPU	Intel Core i7 12700 (6 P-Core + 8 E-Core, in total 20 cores) @ 4 GHz
RAM	32 GB @ 3.2 GHZ
GPU	Nvidia RTX 4070 (36 SMs, each has 128 cores, in total 4608 CUDA cores)
	Intel Iris Xe (96 cores)
GRAM (Nvidia)	8 GB @ 8 GHZ
OS	Windows 11 Enterprise 64-bit (24H2)
Language(s)	ISO C++17, C++20
Compiler	Visual Studio 2022 V17.12 (MSVC) nvcc 12.5 for CUDA Intel icpx (dpc++) 2025.0
Compile Mode	64-bit (Release Mode)

8. Table of Acronyms and Abbreviations

Acronym / Abbreviation	Stands For
AI	Artificial Intelligence
AMD	Advanced Micro Devices Inc.
API	Application Programming Interface
BSD	Berkeley Software Distribution
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPC++	Data Parallel C++
DSP	Digital Signal Processor
E-Core	Efficient Core (CPU)
eDSL	Embedded Domain Specific Language
e.g.	Example
FPGA	Field Programmable Gate Array
G	Giga (almost 10 to the power of 9)
GHZ	Giga Hertz
GPU	Graphics Processing Unit
GP-GPU	General Purpose Graphics Processing Unit
HIP	Heterogeneous compute Interface for Portability
HPC	**High Performance Computing**
ISO	Independent System Operator
M	Mega (almost one million)
MCU	Micro Controller Unit
MHz	Mega Hertz (Mega means million)
MPU	Micro Processor Unit
MSVC	Microsoft Visual C
NPU	Neural Processing Unit
nvcc	Nvidia CUDA Compiler
oneAPI	one Application Programming Interface

oneDPL	one Data Parallel Library
openCL	Open Computing Language
oneTBB	one Threading Building Blocks
OS	Operating System
P-Core	Performance Core (CPU)
RAM	Random Access Memory
RTX	Ray tracing Texel eXtreme - Ray Tracing eXperiance
SP	Stream Processor
SM	Streaming Multiprocessors
Std	Standard
STL	Standard Template Library (C++)
SYCL	System wide Compute Language
UXL	Unified Acceleration Foundation
V	Version
VERILOG	VERIfication & LOGic
VHDL	VHSIC (Very high-speed integrated circuit) Hardware Description Language
VS	Visual Studio

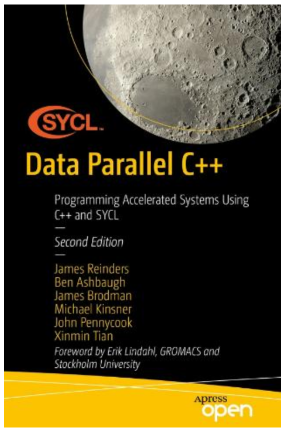
9. Chronogram



10. References

1- **Data Parallel C++ (SYCL) free e-book (2023)**:

https://link.springer.com/book/10.1007/978-1-4842-9691-2



2- **CUDA C++ Programming Guide**:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

3- **Intel oneAPI Base Toolkit Documentation**:

https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-documentation.html

4- **mini-Handbook of Parallel Programming**, my post on Linked-in on 4 December 2024

https://www.linkedin.com/posts/pouya-ebadollahyvahed_mini-handbook-of-parallel-programming-activity-7270045199630749696-DZ24?utm_source=share&utm_medium=member_desktop

5- **Multi-Layer Perceptron (MLP) Neural Networks (ANN) & Back Propagation Training Method**, my introductory post on Linked-in on 22 April 2024

https://www.linkedin.com/posts/pouya-ebadollahyvahed_learning-neural-networks-from-scratch-in-activity-7188151746433564675-ye8J?utm_source=share&utm_medium=member_desktop