Intel-oneAPI para Computación Heterogénea
Intel-oneAPI for Heterogeneous Computing

Germán Castaño Roldán

Trabajo de fin de grado del Grado en Ingeniería Informática, Facultad de Informática, Universidad Complutense de Madrid

Curso 2020/2021

Director:
Carlos García Sánchez

Índice

Acknowledgments **3**

Resumen **4**
Palabras Clave 4

Abstract **5**
Key Words 5

Chapter 1: Introduction **6**
Work plan 7
Learning phase 7
Migration phase 7
Benchmarking phase 7

Chapter 2: Motivation and Objectives **8**
Motivation 8
Objectives 10

Chapter 3: State of the Art **10**
CUDA 11
CUDA Alternatives 12
High-Level Programming 12
OpenMP 12
OpenACC 12
C++ Libraries and Extensions 13
Python and Java 13
SYCL and oneAPI 13
FPGAs 15
Future Accelerators 15

Chapter 4: DPC++ **16**
Asynchronous exceptions 16
Device Selector 17
Buffers and Accessors 18
Unified Shared Memory 19
Queue and parallel_for 19

Chapter 5: DPCT & Rodinia **21**
DPC++ Compatibility Tool 21
Rodinia Benchmarks 22

Chapter 6: Methodology **24**
Migration process 24
Benchmarking process 29
Instrumentalization 30
CUDA vs oneAPI 30
Profiling 31
Other architectures 32
Chapter 7: Migration Results **33**
Warnings 33
Manual modifications 37
Problems encountered 39
Chapter 8: Performance Results **41**
Performance of the memory operations 41
Performance of the kernel execution 42
Overall performance 48
Chapter 9: Conclusions **49**
Bibliography **51**
Glossary **54**

Acknowledgments

Foremost, I would like to thank my director in this project, Carlos García Sánchez, not only for his support and guidance but for arousing my interest in heterogeneous computing during the subject "Programación de GPUs y Aceleradores", without which I wouldn't have chosen a project like this for my final degree project.

I also would like to express my gratitude to Alina Shadrina from Intel for her interest and the insight provided when problems arose during the migration from CUDA to oneAPI.

Resumen

"oneAPI es un modelo de programación unificado, abierto y basado en estándares, que ofrece una experiencia de desarrollador común en todas las arquitecturas de aceleradores, para un rendimiento de aplicaciones más rápido, más productividad y una mayor innovación."

-www.oneapi.com

La herramienta de compatibilidad DPC++ de Intel es un componente del oneAPI Base Toolkit. esta herramienta transforma automáticamente código CUDA en Data Parallel C++ (DPC++) ayudando en el proceso de migración.

Este proyecto consiste en un análisis de la herramienta de compatibilidad DPC++, considerando la intervención manual requerida y los problemas encontrados al migrar los benchmarks de Rodinia. Y un estudio comparativo del rendimiento obtenido por el código migrado.

Palabras Clave

Intel oneAPI, SYCL, Data Parallel C++, Herramienta de compatibilidad Data Parallel, CUDA, Computación heterogénea, Rodinia Benchmark Suite.

Abstract

"oneAPI is a cross-industry, open, standards-based unified programming model that delivers a common developer experience across accelerator architectures—for faster application performance, more productivity, and greater innovation."
-www.oneapi.com

The Intel DPC++ Compatibility Tool is a component of the Intel oneAPI base toolkit. This tool automatically transforms CUDA code into Data Parallel C++ (DPC++) assisting in the migration process.

This project consists of an analysis of the DPC++ Compatibility Tool, considering the manual intervention required and the problems encountered while migrating the Rodinia benchmarks. And a comparative study of the performance obtained by the migrated code.

Key Words

Intel oneAPI, SYCL, Data Parallel C++, Data Parallel Compatibility Tool, CUDA, Heterogeneous computing, Rodinia Benchmark Suite.

Chapter 1: Introduction

Nowadays heterogeneity is widely present in both high-performance computing and consumer electronics. These systems add a multitude of co-processors or accelerators, such as GPUs, TPUs, and FPGAs, to the traditional CPU. However, there isn't a simple, portable and efficient method to develop for these systems. Intel oneAPI aims to fill this role.

This project consisted on an evaluation of the Data Parallel C++ Compatibility Tool assessing the manual modifications needed by the resulting code and performing them in the cases where it was assumable, an instrumentalization of the original and the migrated code and the benchmarking in different heterogeneous systems. All the code generated, along with the outputs of the migration tool and the necessary makefiles and datasheets, is publically available in the github repository:
https://github.com/CR-G/Intel-oneAPI-for-Heterogeneous-Computing

All the problems and concerns encountered during the migration were discussed by email with staff from Intel, contributing to the improvement of the Compatibility Tool and oneAPI.

This document consists on the following 9 chapters:

- Chapter 1: Introduction. Includes a brief introduction to heterogeneous systems, the work performed, a description of the structure of this document and the work plan.
- Chapter 2: Motivation and Objectives. Discusses the motivation behind this project and its objectives.
- Chapter 3: State of the Art. Evolution and current status of the accelerator-based heterogeneous computing.
- Chapter 4: DPC++. Includes an introduction to Data Parallel C++ with an example program.
- Chapter 5: DPCT & Rodinia. Describes the Data Parallel C++ Compatibility Tool and the Rodinia Benchmark Suite.
- Chapter 6: Methodology. Explains the methodology followed by this project, including the migration process, the benchmarking and the instrumentalization of the code.
- Chapter 7: Migration Results. Presents the results of the migration process, discussing the necessary manual intervention and the problems encountered.
- Chapter 8: Performance Results. This chapter exposes the results of the benchmarking.
- Chapter 9: Conclusions.

Work plan

The project was divided into three phases, a learning phase, a migration phase and a benchmarking phase. During the length of the project regular meetings were held with the tutor to monitor the progress and discuss results. Due to the COVID-19 pandemic, all the meetings were held online via video calls.

Learning phase

During this phase I familiarized myself with DPC++ language and oneAPI with the help of the book *Data Parallel C++ Mastering DPC++ for Programing of Heterogeneous Systems using C++ and SYCL* [11], the examples provided by Intel and the video recordings of the Danysoft workshops about oneAPI [17] and with the Rodinia Benchmark Suite, which is widely used and implemented in CUDA.

This phase ended with the manual migration of the nn benchmark to ensure all the necessary aptitudes were acquired

Migration phase

The DPCT was used to migrate the Rodinia Benchmark suite during this phase. Also emails were exchanged with Intel representatives discussing the various problems that appeared with the migration tool.

Benchmarking phase

This phase included the modification of the benchmarks to enable the timing of the execution and the benchmarking itself in the Codeplay's docker and the Intel Devcloud.

The analysis with the Nvidia Visual Profiler was also performed during this phase.

Chapter 2: Motivation and Objectives

Motivation

In recent years a clear tendency towards heterogeneity in computing can be observed, not only in high-performance computers, where in the top-500 list Nvidia GPUs are the most used accelerators; but also in desktops and handheld devices. Nowadays most smartphones include a GPU and the Apple M1 [1] is a system on chip, used in the latest Apple computers and tablets, that includes an ARM CPU, a GPU and other accelerators like a 16-core Neural Engine for artificial intelligence applications.

These heterogeneous SOCs, more prominent every day, are clear indicators of this shift toward heterogeneity. A shift that makes it important to have a simple and unified way of developing for different heterogeneous architectures without depending on a specific vendor.

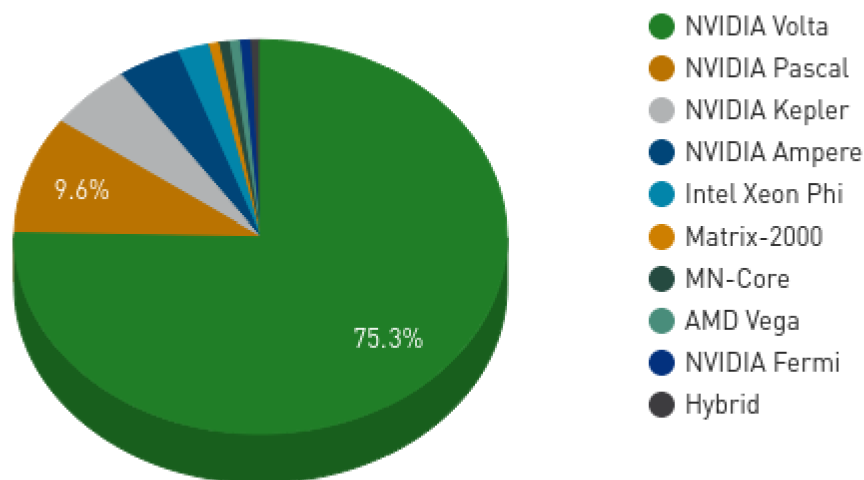


Figure 1: Accelerator system share for the November 2020 Top-500 list [2]

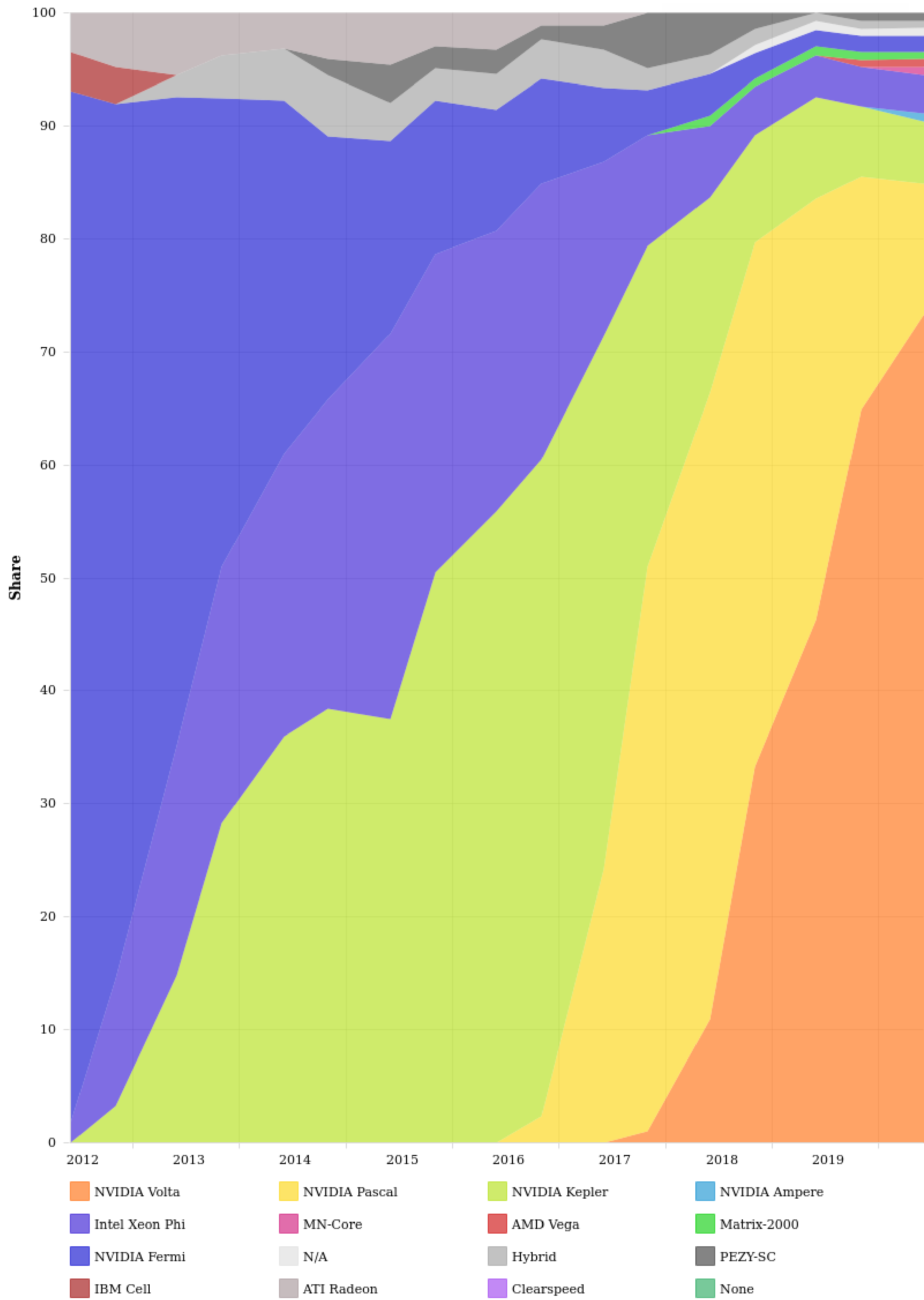


Figure 2: Evolution of the accelerator system share for the Top-500 list from 2012 [3]

Alongside SYCL, oneAPI is the most promising alternatives for heterogeneous programming, providing an open and user friendly approach and, due to the great investment and effort put by Intel on oneAPI, it is in the path to become a future standard in the ever growing world of heterogeneous programming.

As seen in Figures 1 and 2, the most widely used accelerators are Nvidia GPUs. These devices are mostly programmed using CUDA, Nvidia's proprietary programming model; this comes with a series of drawbacks:

- Being CUDA a low-level approach, an experienced programmer is necessary to achieve good performance.
- The language is proprietary and only available for Nvidia GPUs.
- There aren't automatic or guided tools for CUDA programming, that is, the compilers aren't sufficiently developed compared to other compilers aimed at CPUs, where SIMD parallelism is already included.

Objectives

The Intel DPC++ Compatibility Tool (DPCT) helps with the migration process automatically transforming most of the CUDA code into DPC++, greatly reducing the developer's workload.

The goal of this project is to perform an analysis of the migration process from CUDA to DPC++ using the Intel DPC++ Compatibility Tool and the performance obtained by the migrated DPC++ code in different GPU and CPU devices. For this the Rodinia benchmark suite has been migrated to DPC++ with the help of DPCT.

Why Rodinia? Rodinia was developed to address the lack of benchmark suites for accelerators. It was designed for heterogeneous computing infrastructures and includes OpenMP, OpenCL and CUDA implementations.
The Rodinia benchmark suite includes 23 applications in CUDA and most of them are also implemented in OpenMP and OpenCL. Furthermore it is widely used, the paper "Rodinia: A benchmark suite for heterogeneous computing" has more than 2800 citations, and has been previously migrated (manually) to SYCL.

Chapter 3: State of the Art

Before they became mainstream, general purpose GPUs were being evaluated. Matrix multiplication was evaluated in 2001 and GPU LU decomposition outperformed a CPU implementation in 2005. The Cell Processor, a collaboration between Toshiba, Sony and IBM and one of the first architectures to use accelerator-based heterogeneity to multi-media and general purpose applications. It was released in 2006 and used in the Sony PlayStation3 and the Roadrunner supercomputer [5]. Since then GPUs have been widely used as accelerators.

CUDA

Nvidia launched CUDA in 2007 alongside the Tesla GPU, aimed to support general purpose programming. CUDA is a parallel computing platform and programming model developed for general computing on GPUs [4], [5]. "With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs" [4].

CUDA API was more convenient for programmers than previous programming models. In CUDA applications, the sequential part of the workload runs on the CPU while the compute intensive portion of the application runs in parallel on the many GPU cores (see Figure 3). These parallel threads are grouped and executed in a warp, warps are mapped onto thread-blocks, and thread-blocks are mapped onto a grid and grid blocks (CUDA kernel grid in Figure 3) [4], [5].

With the growth of CUDA and general purpose GPUs (GPGPUs), supercomputers started to include GPU accelerators. In 2010 the Chinese Tianhe-1A launched with 7,168 Nvidia TeslaM2050 GPGPUs and became "the world's fastest computer". Later holders of this title also included GPGPUs [5].

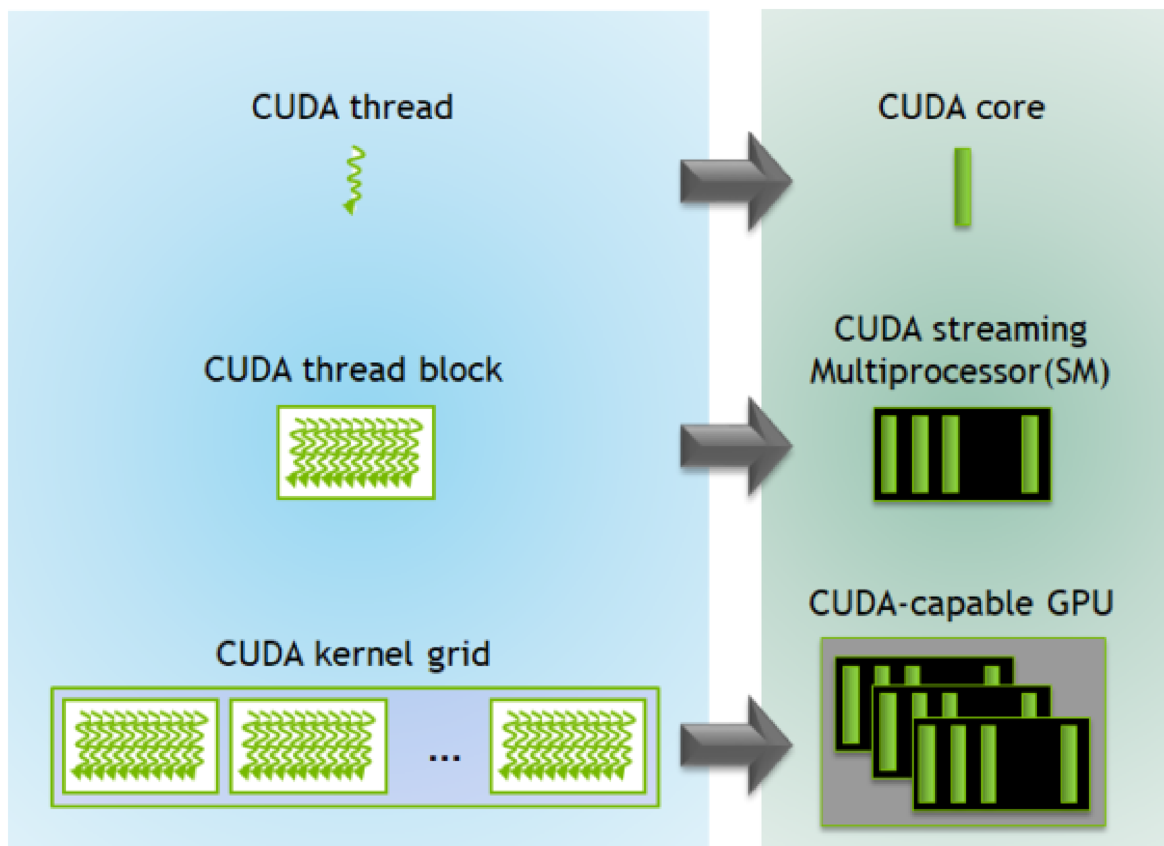


Figure 3: CUDA thread, block and grid on a GPU [6]

Nvidia GPUs are not only widely used in modern supercomputers, but also in personal computers. In 2020 nvidia held 64% of the total PC GPU market share and 80% of the PC discrete GPU market share [20].

CUDA Alternatives

Although CUDA remains the most widely used low-level GPGPU programming model, other approaches have been developed and some achieved a significant amount of use.

In 2018 Microsoft released its DirectCompute API, providing a GPGPU programming framework for Windows platforms. Although it had a similar abstraction level to CUDA, it was overshadowed by CUDA and other approaches, like OpenCL.

OpenCL, or Open Computing Language, was developed by Apple as an open source alternative to CUDA. In 2008 a proposal by Apple was submitted to the Kronos Group for the creation and management of an OpenCL language. By the end of the year the OpenCL 1.0 technical specification was released and supported by AMD, Nvidia, and IBM.

The abstraction level for the OpenCL device code is very similar to CUDA, but the host code is more verbose, making it require a significant amount of low-level and repetitive code. [5].

High-Level Programming

Answering the interest in GPU-based heterogeneous computation, some research-style approaches developed between 2006 and 2011 evolved into production level models. Some of them are:

OpenMP

Short for Open Multi-Processing, OpenMP was the main standard for multi-core CPU computing through the early 2000s. At the beginning of the 2010s there was a demand for OpenMP to support accelerators, resulting in the inclusion of new directives for GPU offloading to OpenMP 4.0, released in 2013. OpenMP 5.0 (2018) expanded support for accelerators and included additional directives for tasking and auto-parallelism [5].

OpenACC

OpenACC (Open Accelerators), one of the first high-level GPGPU programming approaches that remains with a significant user base, was released in 2012 to support the users of ORNL's Titan, one of the first large heterogeneous supercomputers. It allows users to expose parallelism, leaving the mapping of that parallelism to the compiler [5].

OpenACC was a collaboration between Cray, Nvidia (Titan was a Cray machine with Nvidia GPUs) and the Portland Group. The aim was to create an open and directive-based standard like OpenMP. [5]

C++ Libraries and Extensions

A multitude of C++ libraries and extensions have been developed to support heterogeneous computation such as Kokkos and Raja [5]. Other examples are: C++ AMP (Accelerated Massive Parallelism) (2012), C++ Boost (2015), Thrust (2012) and C++ Bolt (2014). Also newer versions of the C++ standard include different types of parallelism directly. C++ 17 has increased SIMD support [5].

Python and Java

Although most high-level hetero-geneous programming approaches target C and C++, some have been developed for Java and Python. PyCUDA and JCUDA provide CUDA wrappers for Python and Java respectively [5].

SYCL and oneAPI

SYCL is a cross-platform abstraction layer writing code for heterogeneous processors using standard ISO C++ with host and kernel code in a single source file. It uses generic programming with templates and lambda functions to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code across the extensive range of various acceleration APIs, such as OpenCL. Developers always have access to lower-level code through seamless integration with the native acceleration API through the interoperability mode, C/C++ libraries, and frameworks such as OpenCV or OpenMP [7].

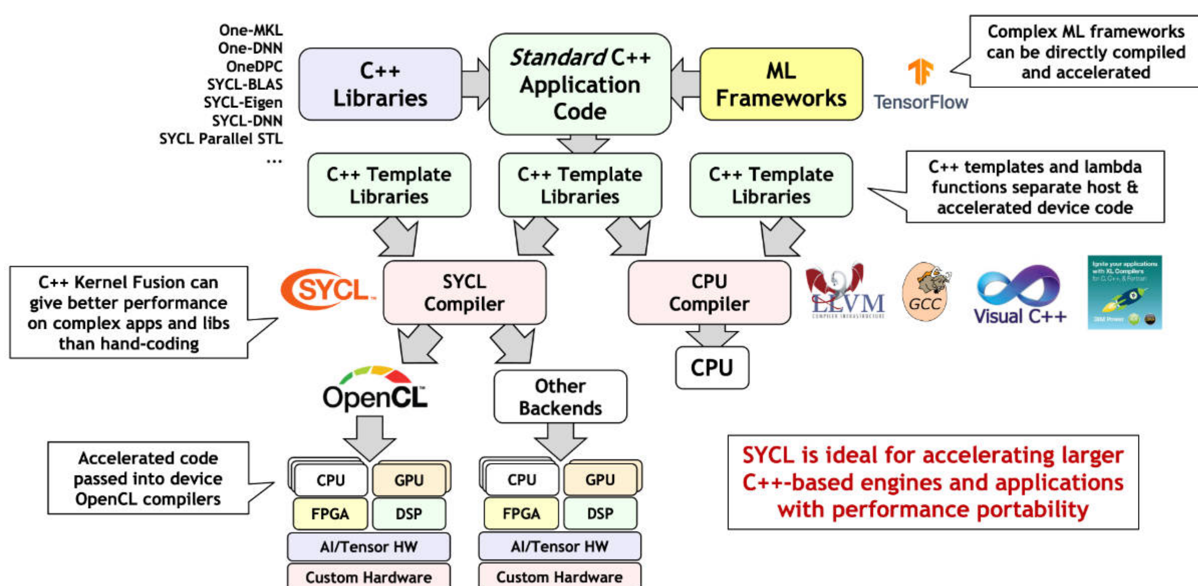


Figure 4: SYCL Single Source C++ Parallel Programming [7]

There are multiple SYCL implementations available, being Intel oneAPI one of them.

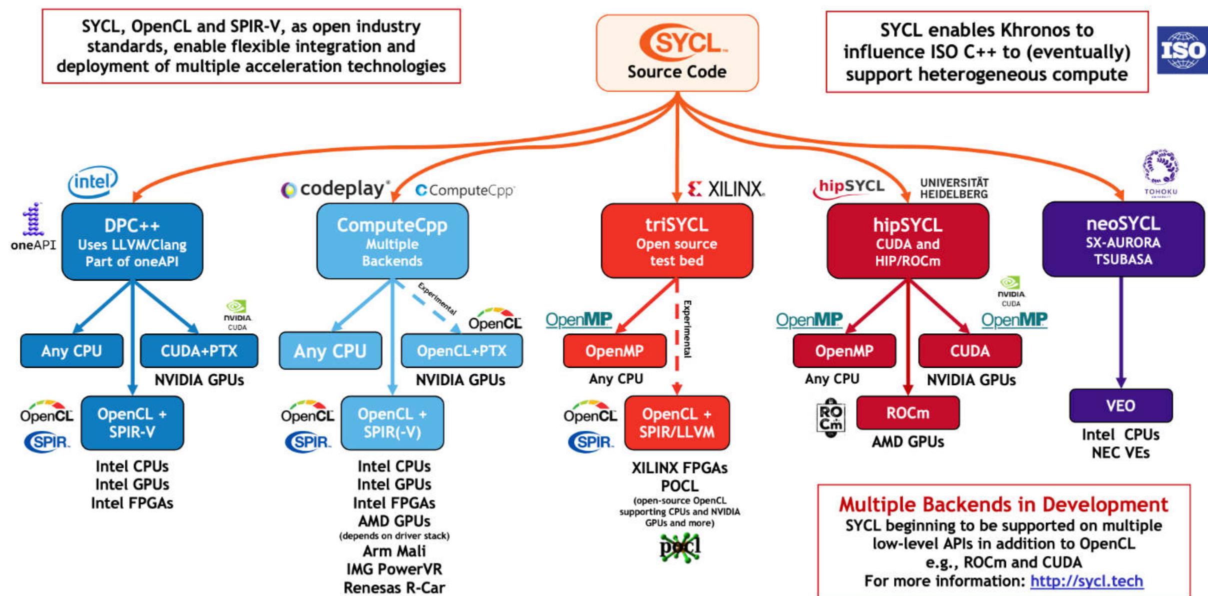


Figure 5: SYCL implementations [7]

OneAPI is an open, free and standards-based programming model that provides portability across accelerators and generations of hardware. It encapsulates several of the discussed technologies and approaches and is comprised of the following language and libraries for creating parallel applications:

- Data Parallel C++ (DPC++) Language: Launched in 2019, DPC++ is oneAPI's core language for programming accelerators and multiprocessors and integrates the SYCL and OpenCL standards with additional extensions.
- oneCCL : Communication primitives for scaling deep learning frameworks across multiple devices.
- oneDAL : Algorithms for accelerated data science.
- oneDNN : High performance implementations of primitives for deep learning frameworks.
- oneDPL : A companion to the oneAPI DPC++/C++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.
- oneMKL : High performance math routines for science, engineering, and financial applications.
- oneTBB : Library for adding thread-based parallelism to complex applications on multiprocessors.
- oneVPL : Algorithms for accelerated video processing.

OneAPI simplifies software development by providing the same languages and programming models across accelerator architectures [8], [5].

FPGAs

FPGAs haven't seen much adoption until the last few years. The real revolution for FPGAs, and their adoption as a heterogeneous accelerator, came from the replacement of the low-level Hardware Definition Languages used to program FPGAs with new programming approaches.

Several high-level options, or High Level Synthesis (HLS) tools, have been developed since the 1990s, however, the first viable HLS for scientific purposes was Altera's OpenCL SDK released in 2013, followed by Xilinx's OpenCL SDKs and HLS tools. Due to the widespread adoption of GPUs at this time, the usage of FPGAs as accelerators became a more promising idea. Intel acquired Altera in 2015 and rebranded it as the SDK as the Intel FPGA SDK for OpenCL. This SDK is still active nowadays.

Other projects aimed to make HLS approaches more accessible, usually by adding software layers on top of the vendor's HLS backends. Some examples are the OpenACC extension OpenARC (2016), the OmpSs framework extended in 2017 to support FPGAs and the ETH Zurich DaCe framework, that introduced a control-flow graph and GUI based interface [5].

Future Accelerators

On top of GPUs and FPGAs, more hardware is being explored for becoming accelerators for heterogeneous systems; like the TPU, a machine learning accelerator first released by Google in 2018 and quickly adopted by Nvidia [5]. From the same year, Oak Ridge National Lab has hosted an international conference on neuromorphic computing, which aims to emulate the operation and structure of the human brain. Two examples of neuromorphic accelerators are the IBM TrueNorth and the Intel Loihi [5], [19].

Quantum computing is also being explored for use as heterogeneous accelerators by companies such as D-Wave and IBM with the Q system [5].

These are only some of the accelerators that will populate the world of heterogeneous computing and more are explored every year, promising a future of extreme heterogeneity and full of challenges for programmers [5].

Chapter 4: DPC++

DPC++ is the heart of oneAPI. DPC++ programs are written in ISO C++ and use the SYCL parallel programming model to distribute computation across processing elements in a device. DPC++ extends SYCL with features for performance and productivity.

DPC++ is single source, device and host code can be included in the same source file. A DPC++ compiler generates code for both the host and device. Any C++ compiler can compile programs that only use the host subset of DPC++ [9].

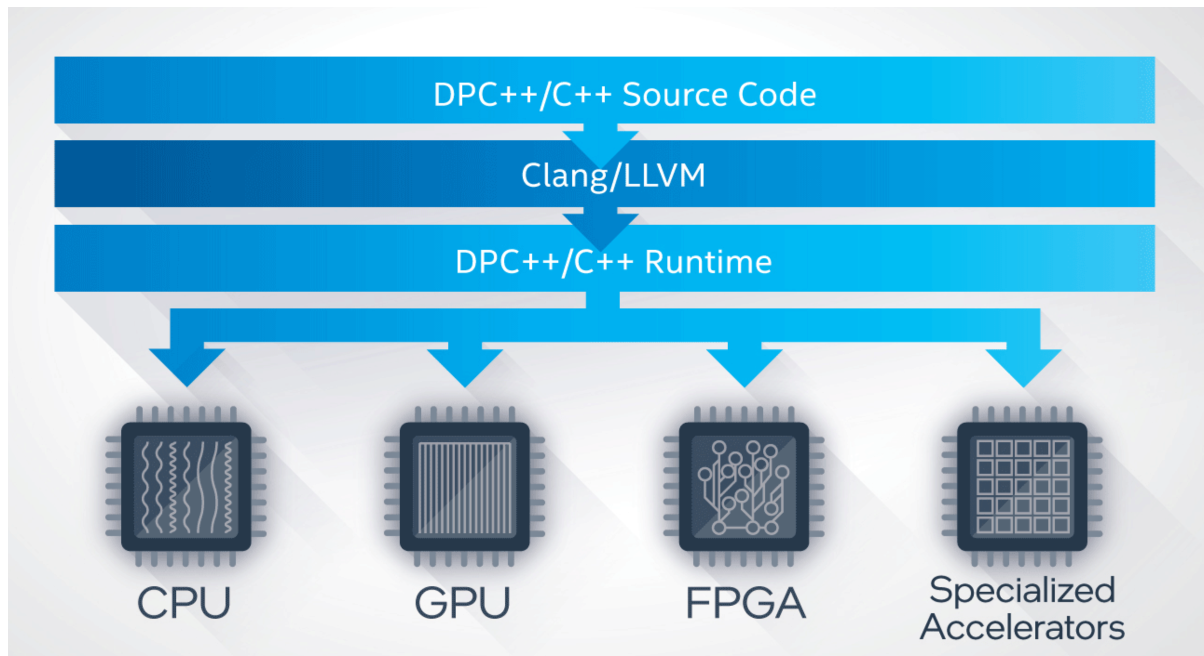


Figure 6: DPC++ Cross-Architecture Compiling [24]

A typical Data Parallel C++ consists of the following sections:

- Asynchronous exceptions from kernels
- Device selectors for different accelerators
- Buffers and accessors (or unified shared memory)
- Queues
- parallel_for kernel

This points can be easily explained following the vector_add [10] example provided by intel:

Asynchronous exceptions

Kernels can run asynchronously and in different stack frames so errors can't be propagated up to the stack. For this and in order to catch asynchronous exceptions, SYCL queues provide error handler functions.

```
static auto exception_handler = [ ](cl::sycl::exception_list eList) {
	for (std::exception_ptr const &e : eList) {
		try {
			std::rethrow_exception(e);
		}
		catch (std::exception const &e) {
			std::terminate();
		}
	}
};
… …
try {
   sycl::queue q(d_selector, exception_handler);
   … …
} catch (sycl::exception const &e) {
   … …
}
```

Device Selector

SYCL and oneAPI provide selectors that can discover and provide access to the hardware that is available on the environment (see Figure 7). The default_selector selects the most performant accelerator available.

```
// The default device selector will select the most performant device.
sycl::default_selector d_selector;
```

default_selector	Any device of the implementation's choosing.
host_selector	Select the host device (always available).
cpu_selector	Select a device that identifies itself as a CPU in device queries.
gpu_selector	Select a device that identifies itself as a GPU in device queries.
accelerator_selector	Select a device that identifies itself as an "accelerator," which includes FPGAs.

Figure 7: Built-in device selectors [11]

Buffers and Accessors

When using buffers, data declared on the host is wrapped in a buffer and is transferred to the accelerators implicitly by the DPC++ runtime. The accelerators read or write to the buffer through an accessor. The runtime also draws the kernel dependencies from the accessors used, then dispatches and runs the kernels in most efficient order.

```
 sycl::buffer a_buf(a_array);
 sycl::buffer b_buf(b_array);
 sycl::buffer sum_buf(sum_parallel.data(), num_items);
… …
 q.submit([&](sycl::handler &h) {

// Create an accessor for each buffer with access permission: read, write, or
// read/write. The accessor is used to access the memory in the buffer.

   sycl::accessor a(a_buf, h, read_only);
   sycl::accessor b(b_buf, h, read_only);

// The sum_accessor is used to store (with write permission) the sum data.

   sycl::accessor sum(sum_buf, h, write_only);
… …
 });
```

The three access modes for buffers are shown in Figure 8.

Access Mode	Description
read	Read-only access.
write	Write-only access. Previous contents not discarded.
read_write	Read and write access.

Figure 8: Buffer access modes [11]

Unified Shared Memory

Unified Shared Memory (USM) is an alternative to buffers for managing and accessing memory from the host and device. Explicit data movement with USM is accomplished, like in CUDA, with device allocations and a special memcpy() found in the queue and handler classes. Data can be allocated for device, host or both (shared), see Figure 9, and it is copied between host and device memory before and after the kernel executes [11] [12].

```
 int* a_device = sycl::malloc_device<int> num_items);
 int* b_device = sycl::malloc_device<int> num_items);
 int* sum_device = sycl::malloc_device<int> num_items);

 q.memcpy(a_device, a_array, sizeof(int) * num_items).wait();
 q.memcpy(b_device, b_array, sizeof(int) * num_items).wait();
… …
 // Kernel execution
… …
 q.memcpy(sum_array, sum_device, sizeof(int) * num_items).wait();
```

Allocation Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	✗	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	can migrate back and forth

Figure 9: USM allocation types [11]

Queue and parallel_for

All the context and states needed for kernel execution are encapsulated in a DPC++ queue. By default, a queue is created and associated with an accelerator, as indicated by Figure 10, through a default selector when no parameter is passed. It can also take a specific device selector and an asynchronous exception handler.

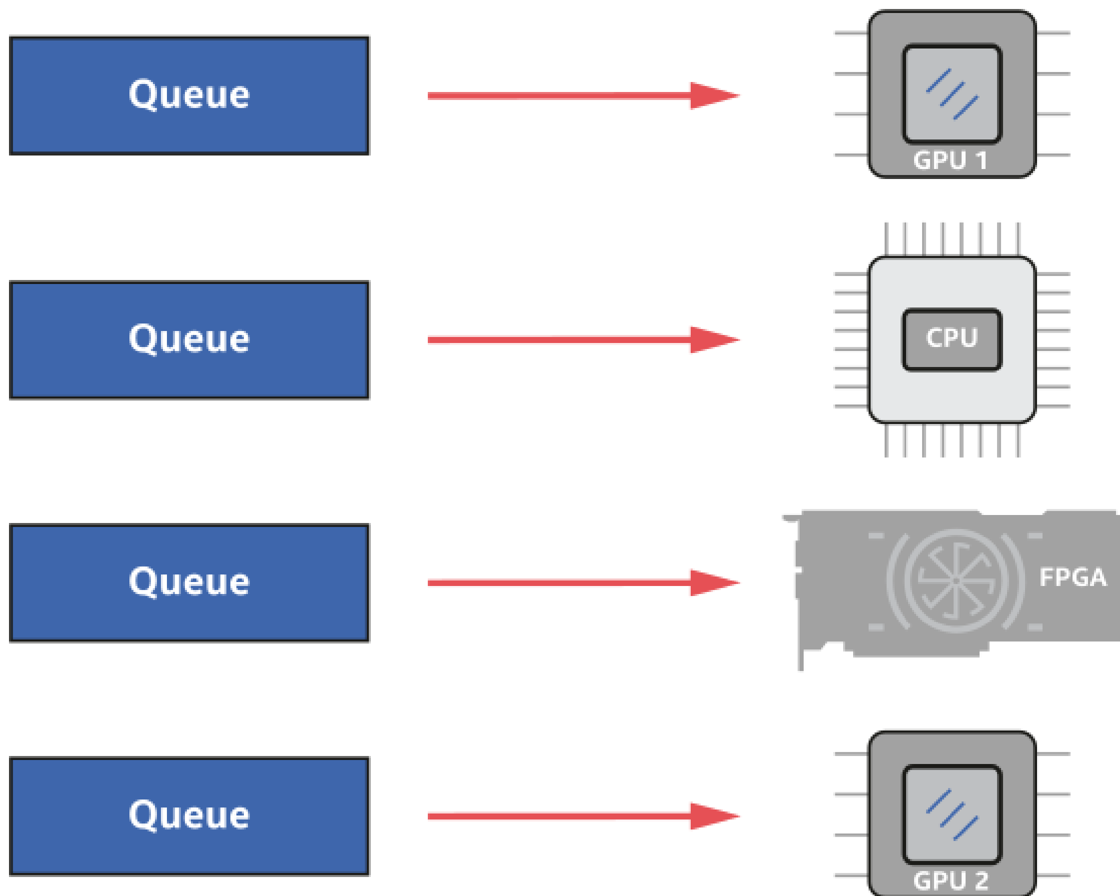


Figure 10: A queue is bound to a single device [11]

Kernels are enqueued to the queue and executed. There are different types of kernels, vector_add uses the basic data-parallel parallel_for kernel.

```
try {
   sycl::queue q(d_selector, exception_handler);
   … …
   q.submit([&](sycl::handler &h) {
   … …
      h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
  });
} catch (sycl::exception const &e) {
   … …
}
```

The kernel body is the addition of two arrays in the Lambda function. num_items, the first parameter of h.parallel_for specifies the range of data the kernel can process.
When using USM the kernel call would be:

```
h.parallel_for(num_items, [=](auto i) { sum_device[i] = a_device[i] + b_device[i]; });
```

Chapter 5: DPCT & Rodinia

DPC++ Compatibility Tool

The Intel DPC++ Compatibility Tool (DPCT) is a component of the Intel oneAPI Base Toolkit (see Figure 11) that assists the developer in the migration of a program that is written in CUDA to a program written in DPC++ [13].

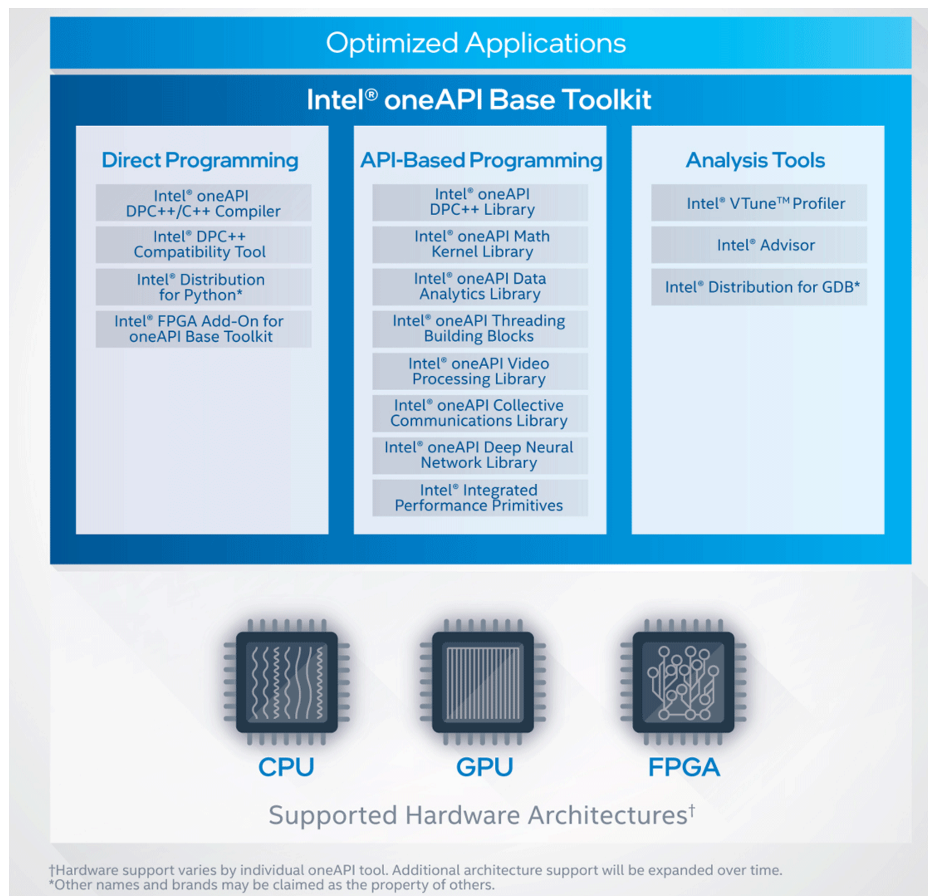


Figure 11: Intel oneAPI Base Toolkit [25]

The tool works by intercepting the build process and replacing CUDA code with the oneAPI counterpart.

Although DPCT automatically migrates most of the code, some manual work is required for a full migration. The tool outputs warnings to indicate how and where manual intervention is needed. These warnings have an assigned ID, of the form "DPCT10XX", that can be consulted in the Developer Guide and Reference. This guide contains a list of all the warnings, their description and a suggestion to fix it.

Rodinia Benchmarks

Quoting Rodinia's website:

> "A vision of heterogeneous computer systems that incorporate diverse accelerators and automatically select the best computational unit for a particular task is widely shared among researchers and many industry analysts; however, there are no agreed-upon benchmarks to support the research needed in the development of such a platform. There are many suites for parallel computing on general-purpose CPU architectures, but accelerators fall into a gap that is not covered by previous benchmark development. Rodinia is released to address this concern." [14].

The Rodinia Benchmark Suite is implemented in CUDA, OpenMP and OpenCL and includes the following applications:

Application	**Dwarves**	**Domain**	**Implementations**
Leukocyte	Structured Grid	Medical Imaging	CUDA, OMP, OCL
Heart Wall	Structured Grid	Medical Imaging	CUDA, OMP, OCL
MUMmerGPU	Graph Traversal	Bioinformatics	CUDA, OMP
CFD Solver	Unstructured Grid	Fluid Dynamics	CUDA, OMP, OCL
LU Decomposition	Dense Linear Algebra	Linear Algebra	CUDA, OMP, OCL
HotSpot	Structured Grid	Physics Simulation	CUDA, OMP, OCL
Back Propagation	Unstructured Grid	Pattern Recognition	CUDA, OMP, OCL
Needleman-Wunsch	Dynamic Programming	Bioinformatics	CUDA, OMP, OCL
Kmeans	Dense Linear Algebra	Data Mining	CUDA, OMP, OCL
Breadth-First Search	Graph Traversal	Graph Algorithms	CUDA, OMP, OCL
SRAD	Structured Grid	Image Processing	CUDA, OMP, OCL
Streamcluster	Dense Linear Algebra	Data Mining	CUDA, OMP, OCL

Particle Filter	Structured Grid	Medical Imaging	CUDA, OMP, OCL
PathFinder	Dynamic Programming	Grid Traversal	CUDA, OMP, OCL
Gaussian Elimination	Dense Linear Algebra	Linear Algebra	CUDA, OCL
k-Nearest Neighbors	Dense Linear Algebra	Data Mining	CUDA, OMP, OCL
LavaMD2	N-Body	Molecular Dynamics	CUDA, OMP, OCL
Myocyte	Structured Grid	Biological Simulation	CUDA, OMP, OCL
B+ Tree	Graph Traversal	Search	CUDA, OMP, OCL
GPUDWT	Spectral Method	Image/Video Compression	CUDA, OCL
Hybrid Sort	Sorting	Sorting Algorithms	CUDA, OCL
Hotspot3D	Structured Grid	Physics Simulation	CUDA, OMP, OCL
Huffman	Finite State Machine	Lossless data compression	CUDA, OCL

Table 1: Rodinia 3.1 applications [14]

Chapter 6: Methodology

The workflow, visualized in Figure 12, was divided into two main processes: the migration process and the benchmarking process.

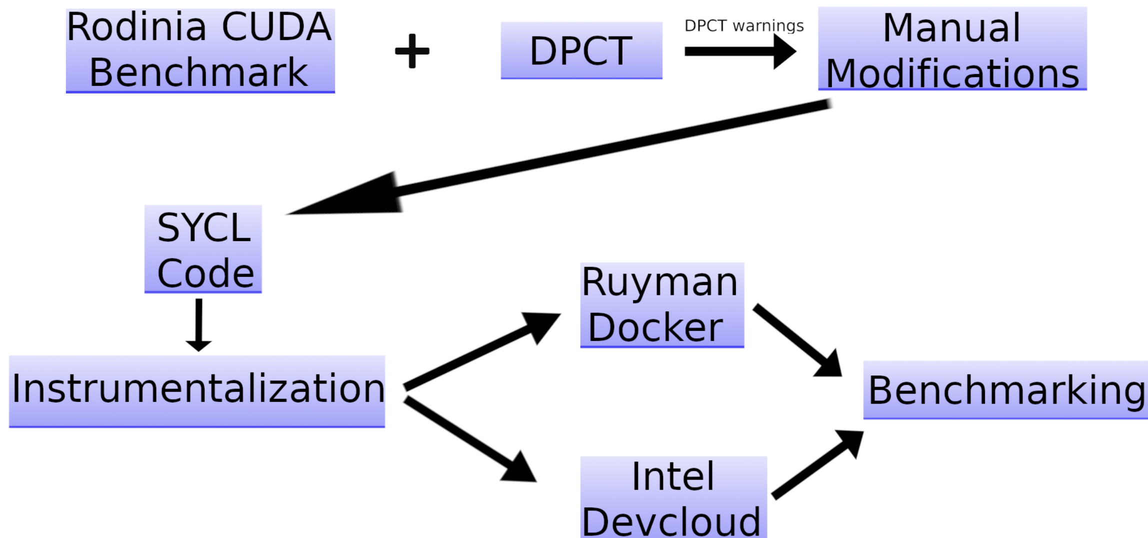


Figure 12: Workflow

Migration process

In order to migrate the Rodinia benchmarks for CUDA to DPC++ the following steps were followed:

1. Generate a compilation database with the tool intercept-build.
   This creates a json file with all the compiler invocations and stores the names of the input files and the compiler options.
   This is done with the command *intercept-build make*.

2. Use the Intel DPC++ Compatibility Tool to migrate the code.
   The command
   dpct -p compile_commands.json --in-root=. --out-root=migration
   migrates the files in the current directory and stores the result in the migration folder.
   During this step comments are inserted where the tool couldn't migrate the code or where the user should review the migration.

3. Verify the migration and address any DPCT warnings generated. Consult the Diagnostics Reference [18] for detailed information about the DPCT warnings.

4. Adapt the Makefiles to use the DPCPP compiler when appropriate and remove the CUDA specific compilation flags.

Figure 13 shows the DPCT usage flow.

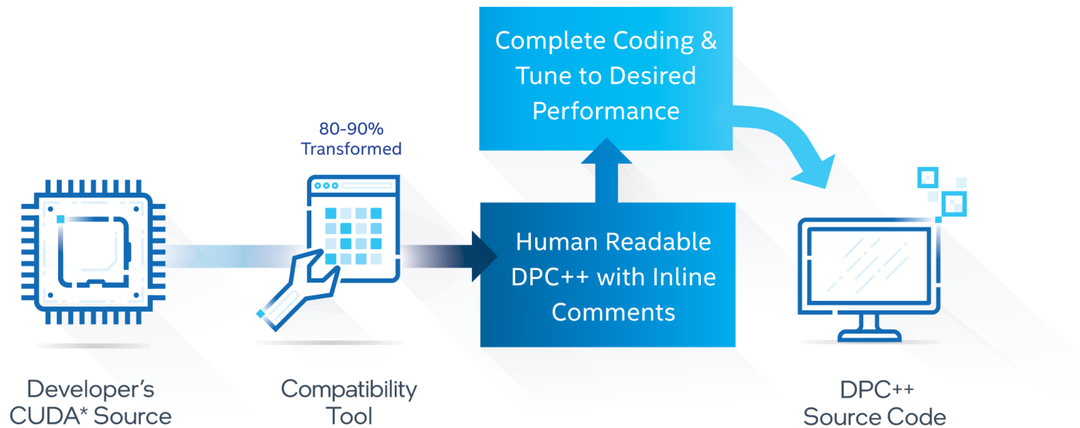


Figure 13: Intel DPC++ Compatibility Tool usage Flow [15]

The Intel DPC++ Compatibility Tool webpage includes the following example [15] to illustrate the migration of a simple source file:

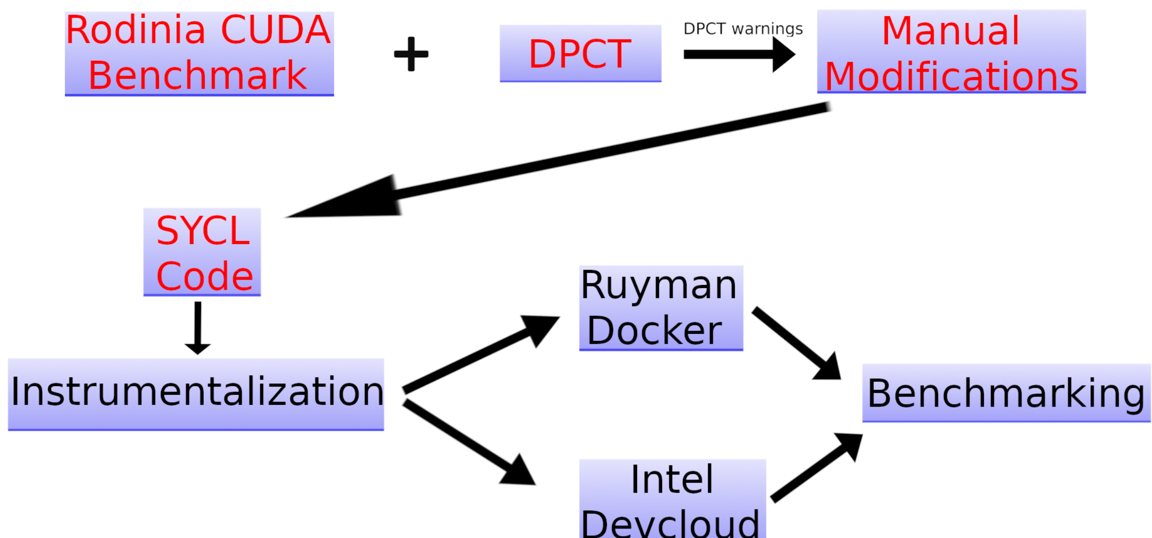


Figure 14: Workflow (migration)

```
#include <cuda.h>
#include <stdio.h>

const int vector_size = 256;

__global__ void SimpleAddKernel(float *A, int offset)
{
  A[threadIdx.x] = threadIdx.x + offset;

}int main()
{
  float *d_A;
  int offset = 10000;

  cudaMalloc( &d_A, vector_size * sizeof( float ) );
  SimpleAddKernel<<<1, vector_size>>>(d_A, offset);

  float result[vector_size] = { };
  cudaMemcpy(result, d_A, vector_size*sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree( d_A );

  for (int i = 0; i < vector_size; ++i) {
    if (i % 8 == 0) printf( "\n" );
    printf( "%.1f ", result[i] );
  }

  return 0;
}
```

Listing 1: Source CUDA code of the DPCT example [15]

This CUDA source is migrated to DPC++ mainly by replacing the CUDA expressions for the SYCL equivalent (threadId is changed to item_ct1.get_local_id(), cudaMalloc() to sycl::malloc_device(), etc) and transforming the kernel invocation to a submission to a SYCL queue of a parallel_for with a lambda expression.

```
1  #include <CL/sycl.hpp>
2  #include <dpct/dpct.hpp>
3  #include <stdio.h>
4
5  const int vector_size = 256;
6
7  void SimpleAddKernel(float *A, int offset, sycl::nd_item<3> item_ct1)
8  {
9    A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + offset;
10
11 }int main()
12 {
13   dpct::device_ext &dev_ct1 = dpct::get_current_device();
14   sycl::queue &q_ct1 = dev_ct1.default_queue();
15   float *d_A;
16   int offset = 10000;
17
18   d_A = sycl::malloc_device<float>(vector_size, q_ct1);
19   q_ct1.submit([&](sycl::handler &cgh) {
20     cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, vector_size),
21                                     sycl::range(1, 1, vector_size)),
22                      [=](sycl::nd_item<3> item_ct1) {
23                        SimpleAddKernel(d_A, offset, item_ct1);
24                      });
25   });
26
27   float result[vector_size] = { };
28   q_ct1.memcpy(result, d_A, vector_size * sizeof(float)).wait();
29
30   sycl::free(d_A, q_ct1);
31
32   for (int i = 0; i < vector_size; ++i) {
33     if (i % 8 == 0) printf( "\n" );
34     printf( "%.1f ", result[i] );
35   }
36
37   return 0;
38 }
```

Listing 2: Migrated DPC++ code of the DPCT example [15]

The resulting migrated code is fairly similar to the original CUDA source and, once the SYCL concepts are understood, easy to comprehend.

Listing 3 shows the differences between the original CUDA code of Listing 1 and the migrated DPC++ code of Listing 2.

```
#include <cuda.h>
#include <stdio.h>

const int vector_size = 256;

__global__ void SimpleAddKernel(float *A,
                                int offset)
{
  A[threadIdx.x] = threadIdx.x + offset;

}

int main()
{

  float *d_A;
  int offset = 10000;

  cudaMalloc( &d_A, vector_size * sizeof( float ) );

  SimpleAddKernel<<<1, vector_size>>>(d_A,
                                      offset);

  float result[vector_size] = { };
  cudaMemcpy(result, d_A, vector_size*sizeof(float),
                      cudaMemcpyDeviceToHost);

  cudaFree( d_A );

  for (int i = 0; i < vector_size; ++i) {
        if (i % 8 == 0) printf( "\n" );
        printf( "%.1f ", result[i] );
  }

  return 0;
}
```

```
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#include <stdio.h>

const int vector_size = 256;

void SimpleAddKernel(float *A, int offset,
                     sycl::nd_item<3> item_ct1)
{
  A[item_ct1.get_local_id(2)] =
        item_ct1.get_local_id(2) + offset;

}

int main()
{
  dpct::device_ext &dev_ct1 =
        dpct::get_current_device();
  sycl::queue &q_ct1 = dev_ct1.default_queue();

  float *d_A;
  int offset = 10000;

  d_A = sycl::malloc_device<float>(vector_size,
                                   q_ct1);
  q_ct1.submit([&](sycl::handler &cgh) {

cgh.parallel_for(sycl::nd_range(sycl::range(1, 1,
                                    vector_size),
                                sycl::range(1, 1,
                                    vector_size)),
                 [=](sycl::nd_item<3> item_ct1) {
                 SimpleAddKernel(d_A, offset,
                                 item_ct1);
                 });
  });

  float result[vector_size] = { };
  q_ct1.memcpy(result, d_A, vector_size *
               sizeof(float)).wait();

  sycl::free(d_A, q_ct1);

  for (int i = 0; i < vector_size; ++i) {
        if (i % 8 == 0) printf( "\n" );
        printf( "%.1f ", result[i] );
  }

  return 0;
}
```

Listing 3: Differences between CUDA and oneAPI

Benchmarking process

The evaluation focused on two different aspects, the performance of oneAPI against CUDA and the performance across different architectures.

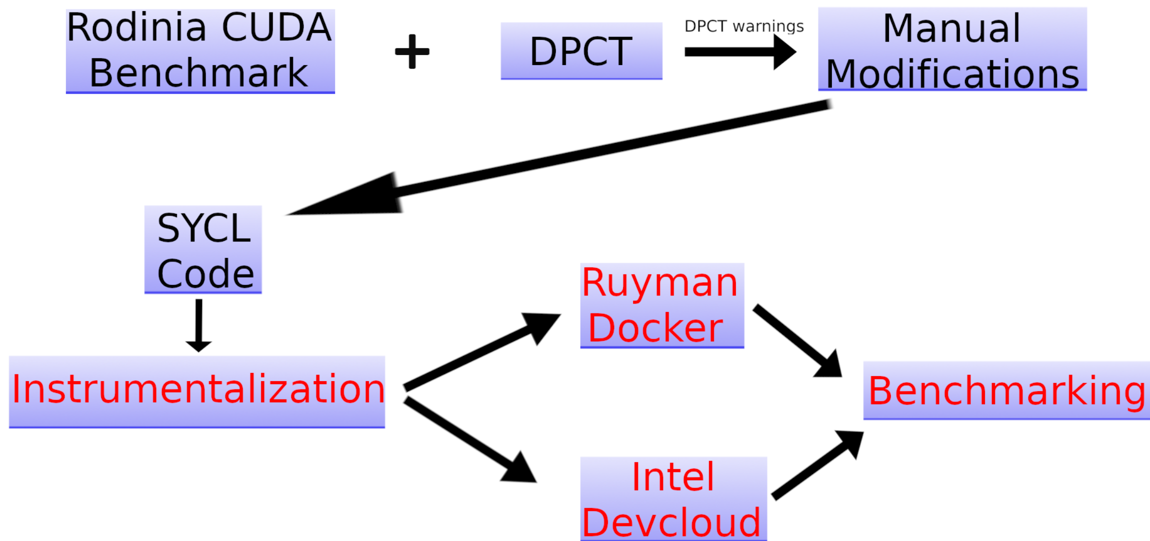


Figure 15: Workflow (benchmarking)

For this, the benchmarks were instrumentalized and executed with following problem sizes:

Benchmark	Problem size
b+tree	1M elements
backdrop	65536 input nodes
bfs	1M vertices
cfd	0,2M elements
dwt2d	1024×1024 images, forward 5/3 transform
gaussian	1024×1024 matrix
heartwall	104 frames
hotspot	512x512 matrix
hotspot3D	512x512 matrix
huffman	1MB file
lavaMD	10 boxes
lud	2048×2048 data points
myocyte	100 time steps
nn	5 nearest neighbors
nw	16000x16000 data points
particlefilter	1M points

pathfinder	100000x1000 2D grid
streamcluster	65536 points 256 dimensions

Table 2: Benchmarks and problem sizes

Instrumentalization

Most of the Rodinia benchmarks didn't output the time spent in the different execution stages, and the ones that did had their own output format. This is why all the benchmarks, the originals in cuda and the migrated ones in DPC++, had to be modified so all of them outputed the same measurements in the same format.

The modifications consisted mainly in surrounding the desired lines of code with:

```
#ifdef TIME_IT
time1 = get_time();
#endif
```

and

```
#ifdef TIME_IT
time2 = get_time();
#endif
```

and printing the difference of the measured times of each stage at the end of the execution.

CUDA vs oneAPI

For this part the environment used consisted of an **Nvidia GTX 1050 Ti GPU** (as device) and, as oneAPI does not currently support the CUDA backend, the Codeplay's ruyman/dpcpp_cuda_examples [16] docker. This docker image includes a DPC++ CUDA compiler that enables the execution of SYCL for CUDA.

The mentioned docker image was run with the command *docker run --privileged=true --mount source=dockerTFG,target=/mnt --gpus all -it ruyman/dpcpp_cuda_examples*

All the migrated benchmarks were executed inside this docker and the time spent across the different stages of the execution (set device, device memory allocation, copy memory host to device, copy memory device to host, device memory free and kernel execution) was measured.

After analyzing the results of these measurements, the benchmarks that had the worst performance compared to the original CUDA application received a more in-depth analysis using the Nvidia Visual Profiler [21].

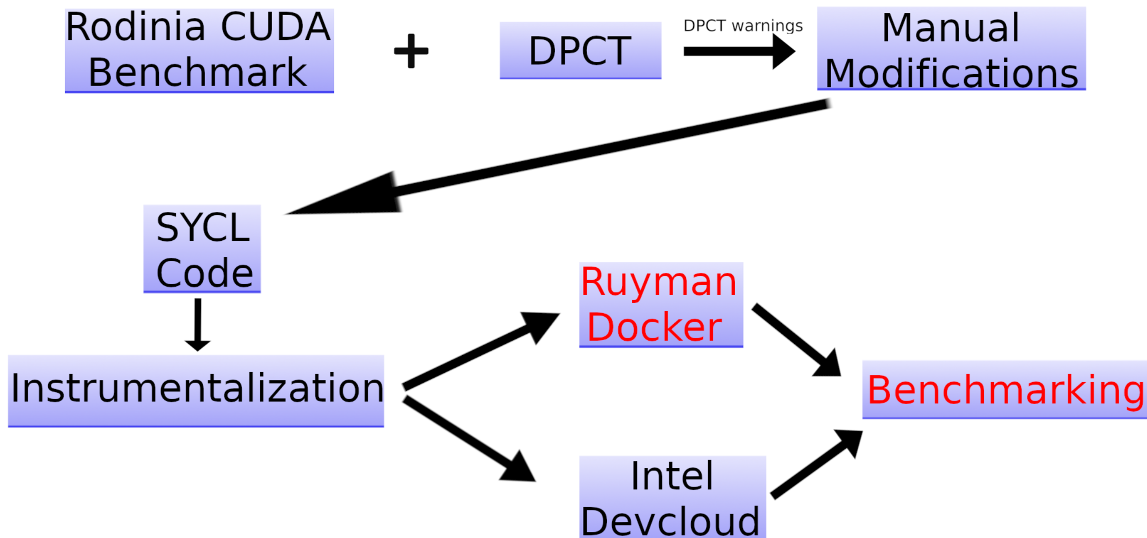


Figure 16: Workflow (CUDA vs oneAPI)

Profiling

On top of the execution of the instrumentalized code, the most relevant applications of the Rodinia benchmarks were analysed using the Nvidia Visual Profiler, a performance profiling tool provided by Nvidia.

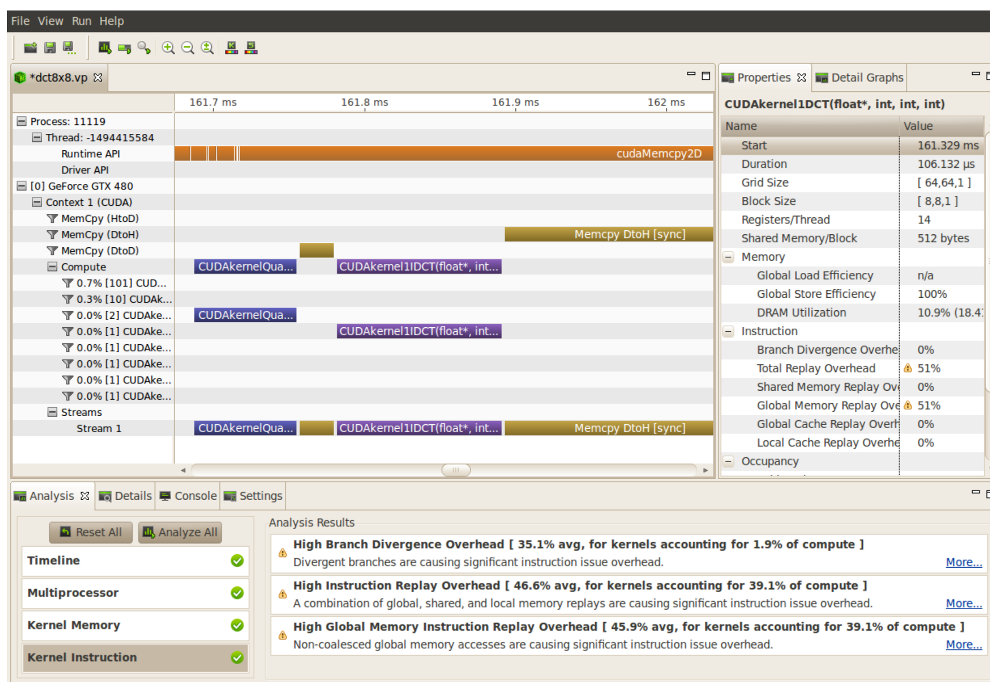


Figure 17: Example of the Nvidia Visual Profiler [21]

This tool allows performing a low-level analysis of the code, showing details about the kernels and the memory transfer not visible by other means.

Other architectures

Taking advantage of the portability provided by oneAPI, the benchmarks were also executed in the Intel Devcloud, with two **Intel XeonGold6128 CPUs** and an **Intel UHDGraphicsP630 GPU** as devices.

The Intel Devcloud allows its users to execute applications with different hardware environments that include various Intel CPUs, GPUs and FPGAs.

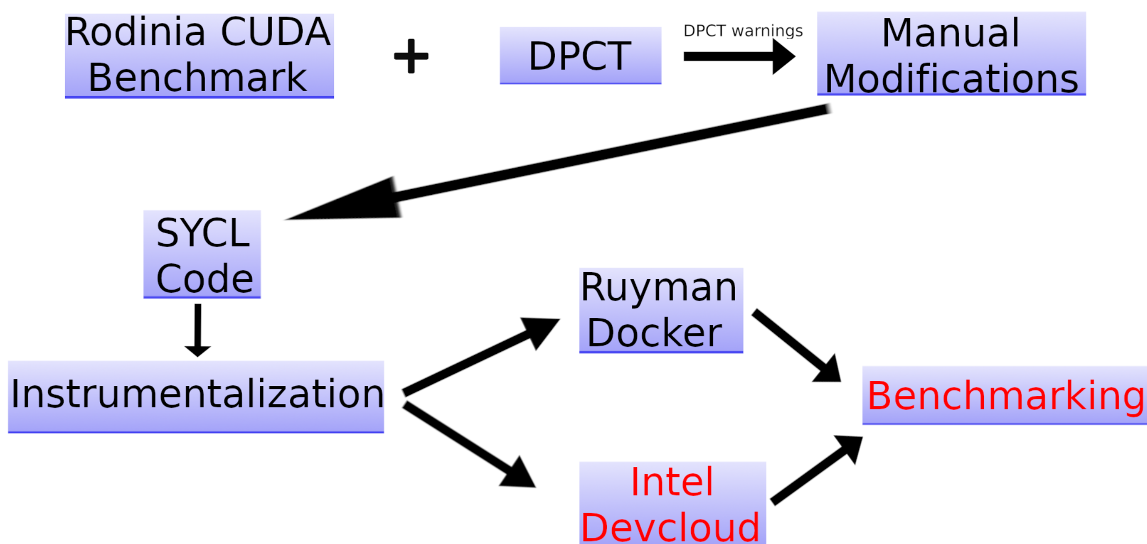


Figure 18: Workflow (benchmarking)

Chapter 7: Migration Results

During the migration process the DPC++ Compatibility Tool generated a series of warnings indicating possible problems and the need for manual intervention by the user. These are discussed in the following sections.

Warnings

Across all the benchmarks 99 files were processed by the DPC++ Compatibility Tool, with a total of 43485 lines of code.
It gave a total of 461 warnings, with an average of 4.65 warnings per file or a warning every 94.3 lines.

Warning code	**Number of appearances**
DPCT1000	4
DPCT1001	4
DPCT1003	171
DPCT1004	1
DPCT1005	15
DPCT1009	45
DPCT1010	38
DPCT1012	20
DPCT1019	1
DPCT1022	2
DPCT1024	2
DPCT1026	8
DPCT1027	2
DPCT1035	9
DPCT1039	7
DPCT1049	75

DPCT1051	2
DPCT1059	14
DPCT1064	2
DPCT1065	29
DPCT1072	1
DPCT1077	9
Total	461

Table 3: Warnings given by DPCT

See the Intel DPC++ Compatibility Tool Developer Guide and Reference [18] for more information about these warnings.

These warnings can be grouped in the following categories:

- Error handling related warnings:
  DPCT1000, DPCT1001, DPCT1003, DPCT1009, DPCT1010, DPCT1024.
  Total: 264 - 57.3%

- Device information related warnings:
  DPCT1005, DPCT1019, DPCT1022, DPCT1051, DPCT1072.
  Total: 21 - 4.6%

- Kernel invocation warnings:
  DPCT1049.
  Total: 75 - 16.3%

- Time measurement warnings:
  DPCT1012.
  Total: 20 - 4.3%

- Warnings caused by the removal of unnecessary function calls:
  DPCT1026, DPCT1027.
  Total 10 - 2.1%

- Warning generated because SYCL not supports something:
  DPCT1059.
  Total 14 - 3%

- Performance improving suggestions:
  DPCT1065.
  Total 29 - 6.3%

- Macro related warnings:
  DPCT1064, DPCT1077.
  Total 11 - 2.4%
- Other:
  DPCT1004, DPCT1035, DPCT1039
  Total 17 - 3.7%

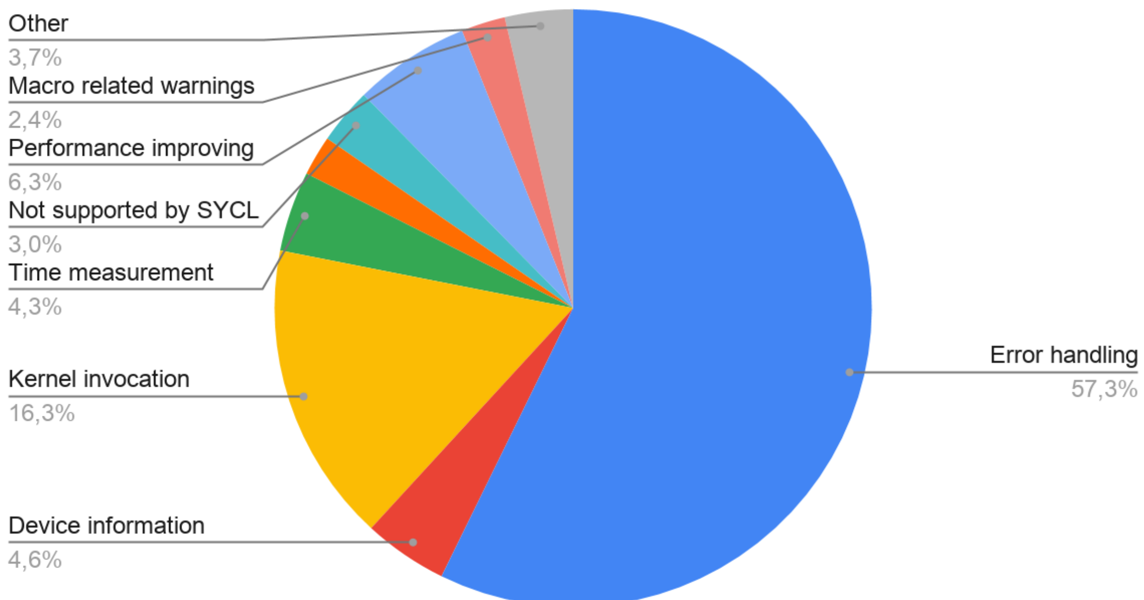


Figure 19: Distribution of the warnings generated by DPCT

Most of the warnings generated by the compatibility tool (57.3%) are caused by the fact that SYCL uses exceptions instead of error codes. Although it might be desirable in order to handle any error that might occur in runtime, no manual modifications were mandatory for these warnings as the tool modifies all error checks so they always return a success.

The second group of warnings appear in every kernel invocation and simply reminds the user the fact that the targeted device might have a smaller limit to the workgroup size. This is usually the case when comparing an Nvidia GPU with an integrated Intel GPU.

Of the rest it should be noted that the warnings related with macros and device information are the ones that require the most manual intervention from the user.

It is worth mentioning the fact that, as the migrated code is a benchmark, the amount of time measurement related warnings is much bigger than it would be in other codes.

In the end twenty out of the twenty three benchmarks were successfully migrated without major manual intervention, resulting in a success rate of almost 87%. In the cases where the migration wasn't successful it was due to issues known by Intel. Work is being done to solve these issues and a fix is planned in the next update (oneAPI 2021.3) for some of them.

Benchmark	**Successful migration**
b+tree	✓
backdrop	✓
bfs	✓
cfd	✓
dwt2d	✓
gaussian	✓
heartwall	✓
hotspot	✓
hotspot3D	✓
huffman	✓
hybridsort	X
kmeans	X
lavaMD	✓
leukocyte	✓
lud	✓
mummergpu	X
myocyte	✓
nn	✓
nw	✓
particlefilter	✓
pathfinder	✓
srad	✓
streamcluster	✓

Table 4: Migration successes

Manual modifications

After the automatic migration some manual modifications of the code were necessary, always addressing the warning messages generated by the tool. As a summary, we can illustrate in the following lines the main modifications performed:

- The workgroup size might need to be adjusted depending on the device used:

  DPCT suggests querying info::device::max_work_group_sice to get the device limit and adjust the workgroup size accordingly.

- When the block size is specified with a macro and used to create a sycl::range the expanded value should be changed back to the macro. When this is the case the tool leaves the original macro commented.

  For example:

```
q_ct1.submit([&](sycl::handler &cgh) {
        sycl::range<2> weight_matrix_range_ct1(16 /*HEIGHT*/, 16 /*WIDTH*/);
        ...
}
```

  should be replaced with

```
q_ct1.submit([&](sycl::handler &cgh) {
        sycl::range<2> weight_matrix_range_ct1(HEIGHT
        , WIDTH);
        ...
}
```

- As SYCL uses exceptions instead of error codes every check is modified by the tool to always succeed. Proper error checking might be added manually with try-catch constructions as it is detailed below:

  A source code line like

```
check_error(cudaMalloc((void **) &arrayX_GPU, sizeof(double)*Nparticles));
```

  will be migrated to

```
check_error((arrayX_GPU = sycl::malloc_device<double>(Nparticles, q_ct1), 0));
```

  In order to handle any possible runtime error this should be changed to something like

```
try{
  arrayX_GPU = sycl::malloc_device<double>(Nparticles,
                                                       q_ct1);
catch (sycl::exception const &e) {
  Handle exception
}
```

- The device selection logic must be manually reviewed as all DPC++ devices (not only the GPU) can be used to submit tasks. It is important to take this into account when the driver version is used to detect a GPU in the original CUDA source.

```
if (devProp.get_major_version() < 1) { … }
```

  used in the original CUDA code to detect the presence of a GPU can be replaced by

```
if (!dpct::get_current_device().is_gpu()){ … }
```

  In this example the execution finished when no GPU was available, but it has no sense on DPC++ as it allows to run the device code into other devices, such as CPU. In this case the check was modified so it continues the execution in another device if no GPU is available.

- Many CUDA device properties don't have a SYCL equivalent, are slightly different or aren't currently supported. This will cause in many occasions the retrieval of incorrect values. For this reason the user must manually review and correct the information queries to the device.

- If the tool makes some assumption to generate the DPC++ code, it will place a warning instructing the user to review the modified code.
  For example DPCT1039:

  > "The Intel® DPC++ Compatibility Tool deduces whether the first parameter of an atomic function points to a global memory address space or a local memory space, using the last assignment's value of the first parameter of the atomic function..."

- SYCL only supports 4-channel image format so the code needs to be manually adjusted. An example of the needed modifications is given in the Intel DPC++ Compatibility Tool Developer Guide and Reference [18].

- When a function call is used in a macro definition it might need to be migrated differently depending on how the macro is called. All uses of this macro must be reviewed.

- Inside a kernel, DPCT suggests replacing barrier() with *barrier(sycl::access::fence_space::local_space)* for better performance if there is no access to global memory. In this case the user should check the memory accesses and the modification.

- If a macro redefines a standard SYCL type it may cause conflicts. The developer guide and reference suggests the user to rename the macro.

Problems encountered

- CL_INVALID_IMAGE_SIZE: In every benchmark where an image data type is used the execution ends with an exception (CL_INVALID_IMAGE_SIZE). This is a known issue that occurs when the info::device::image2d_max_width value of the device is less than the width of the image passed into the kernel. After discussing this with some Intel's staff a workaround was suggested for CPU. There is no solution for GPU yet.

- When a function is called inside a complex macro sometimes the tool erroneously replaces the parameters of the function called with names from an invocation further down the code.

```
#define BIND_TEX_ARRAY(tex, arr, desc) do {
    CUDA_SAFE_CALL(cudaBindTextureToArray(tex, arr, desc));
    ++num_bind_tex_calls;
} while(0)
```

  was migrated to

```
#define BIND_TEX_ARRAY(tex, arr, desc) do {
  CUDA_SAFE_CALL((BIND_TEX_ARRAY(nodetex,
  (cudaArray*)ref->d_node_tex_array,
  nodeTextureDesc).attach(BIND_TEX_ARRAY(nodetex,
    (cudaArray*)ref->d_node_tex_array,nodeTextureDesc),
    BIND_TEX_ARRAY(nodetex,
     (cudaArray*)ref->d_node_tex_array, nodeTextureDesc)),
   0));
++num_bind_tex_calls;
  } while(0)
```

  This is a known bug and a fix is planned for update 2021.3

- The latest CUDA supported version is 11.1. This causes small problems with the intercept-build tool as it won't find some libraries. Re-executing the command with --append resumes the execution from the error, effectively fixing the problem.

- Leaving some positions of an array uninitialized might result in a segmentation fault. This occurred in the particlefilter benchmark where the function

```
void strelDisk(int * disk, int radius){
      int diameter = radius*2 - 1;
      int x, y;
      for(x = 0; x < diameter; x++){
            for(y = 0; y < diameter; y++){
                  double distance =
                  sqrt(pow((double)(x - radius + 1), 2) +
                        pow((double)(y - radius + 1), 2));
                  if(distance < radius)
                        disk[x*diameter + y] = 1;
            }
      }
}
```

left a position uninitialized when the distance was greater or equal to the radius. The original CUDA version ran without any problem, but this caused the mentioned segmentation fault in DPC++. The solution was setting to 0 the uninitialised memory positions.

Chapter 8: Performance Results

The results obtained during the benchmarking phase are discussed in this chapter. The first two sections address the results of the CUDA vs oneAPI benchmarking and the third one discusses the comparison of the oneAPI performance across different hardware architectures.

Performance of the memory operations

In order to compare the performance of the operations device memory allocation, copy memory host to device, copy memory device to host and device memory free, both, the original CUDA version and the migrated oneAPI versions of the mentioned benchmarks, were instrumentalized and executed three times using the same GPU (Nvidia GTX 1050 TI) as a device in the ruyman/dpcpp-cuda-examples-docker. The docker was needed because the current DPC++ compiler does not support the CUDA backend and this docker is provided with a clang++ capable of compiling the oneAPI sources for execution with the CUDA backend.

Each operation consist on:

- device memory allocation: Allocate memory in the device to store the data that will be used in the kernel.
- copy memory host to device: Copy the data from host memory to device memory.
- copy memory device to host: Copy the data from device memory to host memory.
- device memory free: Deallocate memory in the device.

Figure 20 shows the time spent on each memory operation across all the timed benchmarks.

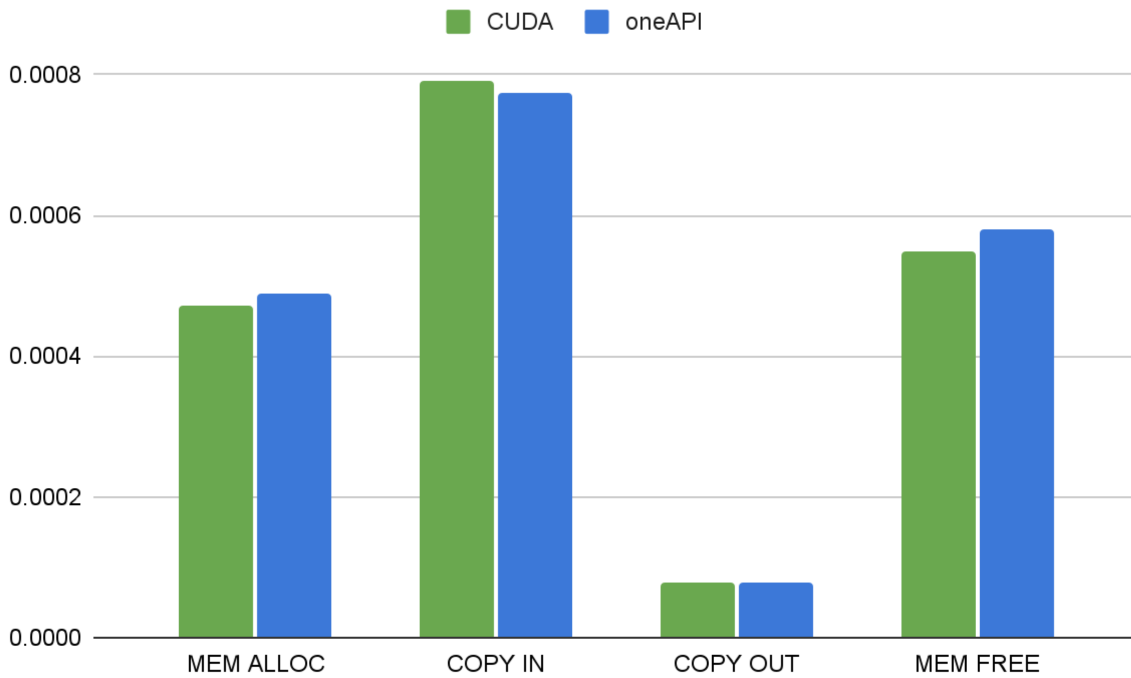


Figure 20: Performance of the memory operations

Although oneAPI achieves similar performance to CUDA on these operations, it introduces a series of CUDA API calls for context and event management (among others) that were not needed by the original CUDA program and this introduces a slight overhead of up to a few milliseconds, depending on the specifics of the application.

Performance of the kernel execution

For the kernel execution the same testing setup of the memory operations was used (Nvidia GTX 1050 TI as device in the ruyman/dpcpp-cuda-examples-docker)and the results can be observed in Figure 21.

Figure 21: Kernel execution times (logarithmic scale)

While some of the migrated applications achieved very similar performance to the CUDA original, others show a considerable overhead, varying from 25% to 160%.

A more in-depth analysis of the nn kernel performed with the Nvidia Visual Profiler revealed that, while in the CUDA version the bottleneck of the execution was the memory bandwidth of the device, in the oneAPI version the limiting factor were the arithmetic operations. Figure 22 and Figure 23 show that oneAPI introduces a great amount of miscellaneous instructions that weren't present in the CUDA version.

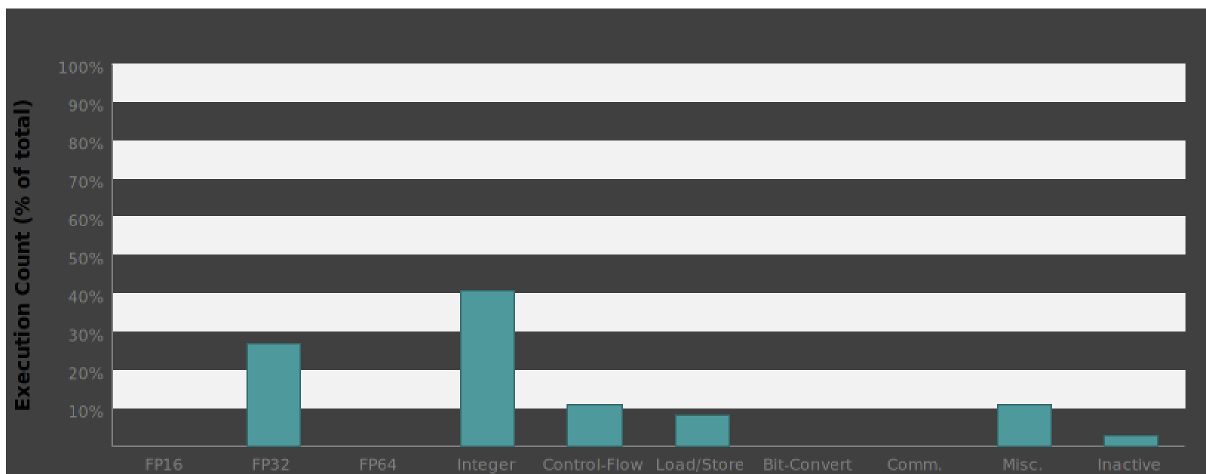


Figure 22: Instruction count for NN (CUDA)

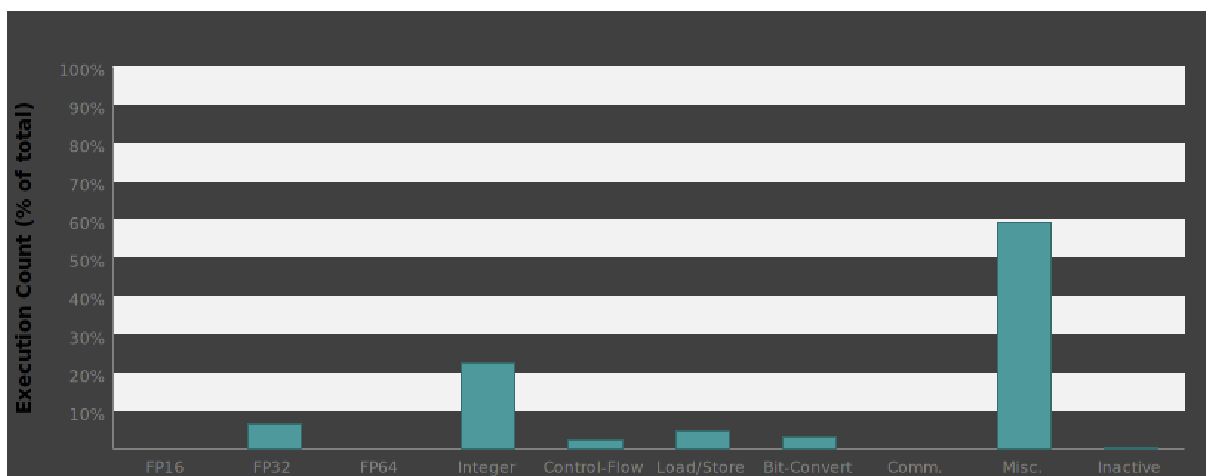


Figure 23: Instruction count for NN (oneAPI)

A look into the disassembly of the kernels clearly shows the added complexity of the oneAPI version.

```
ZTSZZ4mainENKUlRN2cl4sycl7handlerEE194_
18clES2_EUlNS0_7nd_itemILi3EEEE198_13:
        MOV R1, c[0x0][0x20] ;
        MOV R2, c[0x0][0x14] ;
        MOV R2, R2 ;
        MOV32I R0, 0x150 ;
        LDC R0, c[0x0][R0] ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        MOV R3, c[0x0][0x8] ;
        MOV R3, R3 ;
.L_7:
        I2I.U32.U32 R3, R3 ;
        MOV R4, R3 ;
        MOV R5, RZ ;
        MOV R3, R4 ;
        MOV R4, R5 ;
        MOV R3, R3 ;
        MOV R4, R4 ;
.L_8:
        S2R R5, SR_CTAID.Y ;
        MOV R5, R5 ;
        MOV R5, R5 ;
.L_10:
        S2R R6, SR_CTAID.X ;
        MOV R6, R6 ;
        I2I.U32.U32 R6, R6 ;
        MOV R6, R6 ;
        MOV R7, RZ ;
        MOV R9, R6 ;
        MOV R10, R7 ;
        MOV R9, R9 ;
        MOV R10, R10 ;
.L_9:
        S2R R6, SR_TID.X ;
        MOV R6, R6 ;
        MOV R6, R6 ;
.L_11:
        I2I.U32.U32 R6, R6 ;
        MOV R6, R6 ;
        MOV R7, RZ ;
        MOV R8, R6 ;
        MOV R6, R7 ;
        MOV R7, R8 ;
        MOV R8, R6 ;
.L_6:
        IMUL R2, R5, R2 ;
        I2I.U32.U32 R2, R2 ;
        MOV R2, R2 ;
        MOV R6, RZ ;
        IADD R5.CC, R2, R9 ;
        IADD.X R6, R6, R10 ;
        IMUL.U32.U32 R2, R5, R3 ;
        IMUL.U32.U32.HI R9, R5, R3 ;
        IMUL.U32.U32 R10, R6, R3 ;
        IMUL.U32.U32.HI R3, R6, R3 ;
        IMUL.U32.U32 R11, R5, R4 ;
        IMUL.U32.U32.HI R5, R5, R4 ;
        IMUL.U32.U32 R12, R6, R4 ;
        IMUL.U32.U32.HI R4, R6, R4 ;
        IADD R9.CC, R9, R10 ;
        IADD.X R3, R3, RZ ;
        IADD R5.CC, R5, R12 ;
        IADD.X R6, R4, RZ ;
        IADD R4.CC, R9, R11 ;
        IADD.X R3.CC, R3, R5 ;
        IADD.X R3, R6, RZ ;
        IADD R2.CC, R2, R7 ;
        IADD.X R4, R4, R8 ;
        MOV R3, R2 ;
        MOV R2, R2 ;
        MOV R4, R4 ;
        MOV R3, R3 ;
        ISETP.GE.AND P0, PT, R3, R0,
PT ;
        PSETP.AND.AND P0, PT, P0, PT,
PT ;
        @P0 BRA `(.L_1) ;
        MOV32I R0, 0x158 ;
        LDC R0, c[0x0][R0] ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        MOV32I R3, 0x154 ;
        LDC R3, c[0x0][R3] ;
        MOV R3, R3 ;
        MOV R3, R3 ;
        MOV32I R4, 0x148 ;
        LDC.64 R4, c[0x0][R4] ;
        MOV R10, R4 ;
        MOV R11, R5 ;
        MOV R10, R10 ;
        MOV R11, R11 ;
        MOV R10, R10 ;
        MOV R11, R11 ;
        MOV32I R4, 0x140 ;
        LDC.64 R4, c[0x0][R4] ;
        MOV R8, R4 ;
        MOV R9, R5 ;
        MOV R8, R8 ;
        MOV R9, R9 ;
        MOV R8, R8 ;
        MOV R9, R9 ;
.L_12:
        MOV R2, R2 ;
        I2I.S32.S32 R2, R2 ;
        SHR R4, R2, 0x1f ;
        MOV R5, R4 ;
        MOV R4, R2 ;
        MOV R4, R4 ;
        MOV R5, R5 ;
        MOV R6, R4 ;
        MOV R7, R5 ;
        MOV R6, R6 ;
        MOV R7, R7 ;
        SHF.L.U64 R2, R6, 0x2, R7 ;
        SHL R16, R6, 0x2 ;
        IADD R16.CC, R10, R16 ;
        IADD.X R2, R11, R2 ;
        MOV R16, R16 ;
        MOV R2, R2 ;
.L_13:
        SHF.L.U64 R7, R6, 0x3, R7 ;
        SHL R6, R6, 0x3 ;
        IADD R6.CC, R8, R6 ;
        IADD.X R7, R9, R7 ;
        MOV R4, R6 ;
        MOV R5, R7 ;
        MOV R4, R4 ;
        MOV R5, R5 ;
        LDG.E R4, [R4] ;
        FADD R3, R3, -R4 ;
        FMUL R3, R3, R3 ;
        IADD32I R4.CC, R6, 0x4 ;
        IADD.X R5, R7, RZ ;
        MOV R4, R4 ;
        MOV R5, R5 ;
        MOV R4, R4 ;
        MOV R5, R5 ;
        LDG.E R4, [R4] ;
        FADD R0, R0, -R4 ;
        FMUL R0, R0, R0 ;
        FADD R0, R3, R0 ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        MOV R0, R0 ;
        IADD32I R3, R0, -0xd000000 ;
        MOV32I R4, 0x727fffff ;
        ISETP.LE.U32.AND P0, PT, R3,
R4, PT ;
        MOV R0, R0 ;
        SSY `(.L_2) ;
        @P0 BRA `(.L_3) ;
        MOV R4, R0 ;
        MOV R4, R4 ;
        JCAL
`(__cuda_sm20_sqrt_rn_f32_slowpath) ;
        MOV R4, R4 ;
        MOV R4, R4 ;
        MOV R4, R4 ;
        MOV R4, R4 ;
.L_61:
        SYNC ;
.L_3:
        MUFU.RSQ R3, R0 ;
        FMUL.FTZ R4, R3, R0 ;
        FADD.FTZ R5, -RZ, -R4 ;
        FFMA R5, R5, R4, R0 ;
        MOV32I R0, 0x3f000000 ;
        FMUL.FTZ R0, R3, R0 ;
        FFMA R0, R5, R0, R4 ;
        MOV R4, R0 ;
.L_63:
        SYNC ;
.L_2:
        MOV R4, R4 ;
        MOV R4, R4 ;
        MOV R4, R4 ;
        MOV R4, R4 ;
        MOV R4, R4 ;
.L_14:
        MOV R3, R16 ;
        MOV R2, R2 ;
        LOP.XOR R3, R3, R2 ;
        LOP.XOR R2, R3, R2 ;
        LOP.XOR R3, R3, R2 ;
        STG.E [R2], R4 ;
.L_1:
        EXIT ;
        NOP ;
        NOP ;
        NOP ;
        NOP ;
        EXIT ;
.L_4:
        BRA `(.L_4) ;
        NOP;
        NOP;
        NOP;
.L_5:
```

Listing 4: Disassembly for nn kernel (oneAPI)

```
_Z6euclidP7latLongPfiff:
        MOV R1, c[0x0][0x20] ;
        S2R R0, SR_CTAID.Y ;
        S2R R2, SR_CTAID.X ;
        S2R R3, SR_TID.X ;
        XMAD.MRG R5, R0.reuse, c[0x0]
[0x14].H1, RZ ;
        XMAD R2, R0.reuse, c[0x0]
[0x14], R2 ;
        XMAD.PSL.CBCC R0, R0.H1,
R5.H1, R2 ;
        XMAD R3, R0, c[0x0] [0x8], R3 ;
        XMAD.MRG R2, R0.reuse, c[0x0]
[0x8].H1, RZ ;
        XMAD.PSL.CBCC R0, R0.H1,
R2.H1, R3 ;
        ISETP.GE.AND P0, PT, R0,
c[0x0][0x150], PT ;
        @P0 EXIT ;
        {        SHR R2, R0.reuse,
0x1d ;
        SSY `(.L_1)        }
        ISCADD R4.CC, R0,
c[0x0][0x140], 0x3 ;
        IADD.X R5, R2, c[0x0][0x144] ;
        LDG.E R3, [R4+0x4] ;
        LDG.E R2, [R4] ;
        FADD R3, -R3, c[0x0][0x158] ;
        FADD R2, -R2, c[0x0][0x154] ;
        FMUL R3, R3, R3 ;
        FFMA R6, R2, R2, R3 ;
:
        IADD32I R2, R6, -0xd000000 ;
        ISETP.GT.U32.AND P0, PT, R2,
c[0x2][0x0], PT ;
        SHR R3, R0.reuse, 0x1e ;
        ISCADD R2.CC, R0,
c[0x0][0x148], 0x2 ;
        {        IADD.X R3, R3,
c[0x0][0x14c] ;
        @!P0 BRA `(.L_2)        }
        CAL
`($_Z6euclidP7latLongPfiff$__cuda_sm20_sqrt
_rn_f32_slowpath) ;
.L_38:
        SYNC ;
.L_2:
        MUFU.RSQ R0, R6 ;
        FMUL.FTZ R4, R6, R0 ;
        FMUL.FTZ R7, R0, 0.5 ;
        FADD.FTZ R5, -R4.reuse, -RZ ;
        FFMA R5, R4, R5, R6 ;
.L_40:
        {        FFMA R0, R5, R7,
R4 ;
        SYNC        }
.L_1:
        STG.E [R2], R0 ;
        EXIT ;
$_Z6euclidP7latLongPfiff$__cuda_sm20_sqrt_r
n_f32_slowpath:
        LOP.AND.NZ P0, RZ, R6,
c[0x2][0x4] ;
        {        @!P0 MOV R0, R6 ;
        @!P0 RET        }
        FSETP.GEU.FTZ.AND P0, PT,
R6, RZ, PT ;
        {        @!P0 MOV32I R0,
0x7fffffff ;
        @!P0 RET        }
        FSETP.GTU.FTZ.AND P0, PT,
|R6|, +INF , PT ;
        {        @P0 FADD.FTZ R0,
R6, 1 ;
        @P0 RET   }
        FSETP.NEU.FTZ.AND P0, PT,
|R6|, +INF , PT ;
        @P0 FFMA R4, R6,
1.84467440737095516160e+19, RZ ;
        {        @!P0 MOV R0, R6 ;
        @P0 MUFU.RSQ R5, R4
        }
        @P0 FMUL.FTZ R7, R4, R5 ;
        @P0 FMUL.FTZ R8, R5, 0.5 ;
        @P0 FADD.FTZ R9, -R7.reuse,
-RZ ;
        @P0 FFMA R9, R7, R9, R4 ;
        @P0 FFMA R8, R9, R8, R7 ;
        {        @P0 FMUL.FTZ R0,
R8, 2.3283064365386962891e-10 ;
        RET        }
.L_3:
        BRA `(.L_3) ;
.L_44
```

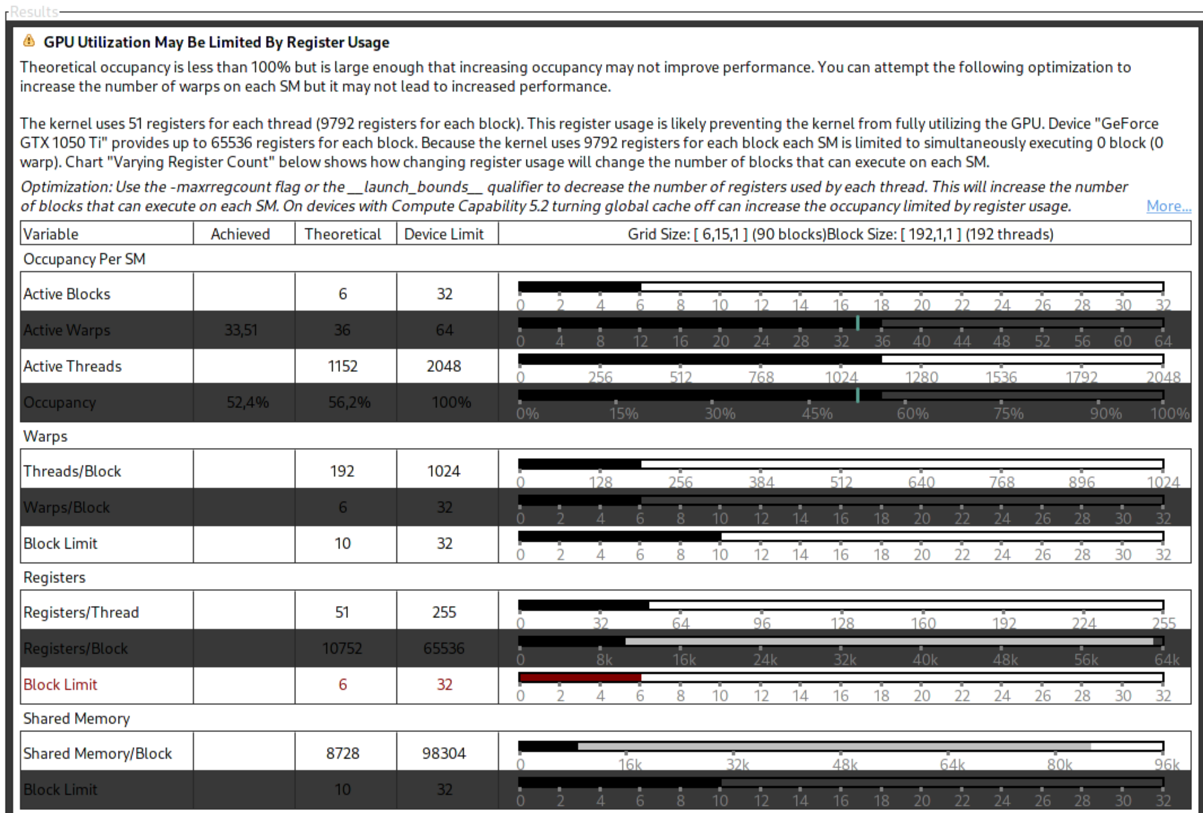
Listing 5: Disassembly for nn kernel (CUDA)

Listings 4 and 5 show a 200% increase in the number of instructions from CUDA to oneAPI. Specifically oneAPI uses almost 20 times more *MOV* instructions and doesn't utilice some of the more complex instructions that could improve the performance. See the CUDA Instruction Set Reference [26] for more information.

The other found cause of overhead, as shown by the profiling of dwt2d (Figures 24 & 25), is the register and shared memory utilization.

As can be seen, the CUDA version achieves better performance with the same resources by using a different register and memory mapping. The oneAPI version tries to use 8.523 KB of shared memory, but, since the used GPU only has 96KB of shared memory, this causes a massive overhead as the data must be continuously moved across the different device's memories.

These differences on the instructions and the memory mapping can be attributed to the compiler, being the oneAPI one much younger and less specific than the CUDA one.



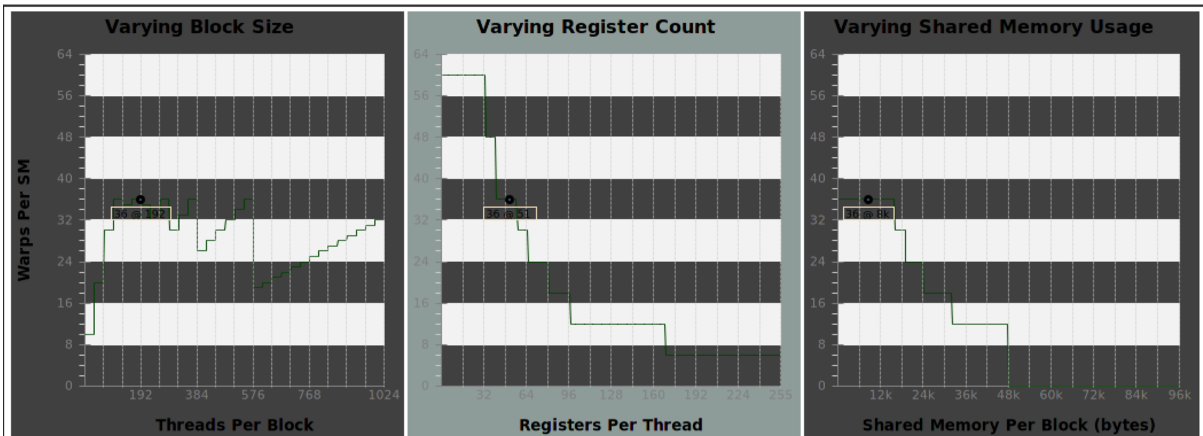


Figure 24: dwt2d (CUDA) latency analysis

Results

GPU Utilization Is Limited By Shared Memory Usage

The kernel uses 8.523 KiB of shared memory for each block. This shared memory usage is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 1050 Ti" is configured to have 96 KiB of shared memory for each SM. Because the kernel uses 8.523 KiB of shared memory for each block each SM is limited to simultaneously executing 0 block (0 warp). Chart "Varying Shared Memory Usage" below shows how changing shared memory usage will change the number of blocks that can execute on each SM.

Optimization: Reduce shared memory usage to increase the number of blocks that can execute on each SM. You can also increase the number of blocks that can execute on each SM by increasing the amount of shared memory available to your kernel. You do this by setting the preferred cache configuration to "prefer shared". More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [6,15,1] (90 blocks)Block Size: [192,1,1] (192 threads)
Occupancy Per SM				
Active Blocks		0	32	
Active Warps	31,6	0	64	
Active Threads		0	2048	
Occupancy	49,4%	0%	100%	
Warps				
Threads/Block		192	1024	
Warps/Block		6	32	
Block Limit		10	32	
Registers				
Registers/Thread		45	255	
Registers/Block		9216	65536	
Block Limit		6	32	
Shared Memory				
Shared Memory/Block		8728	98304	
Block Limit		0	32	

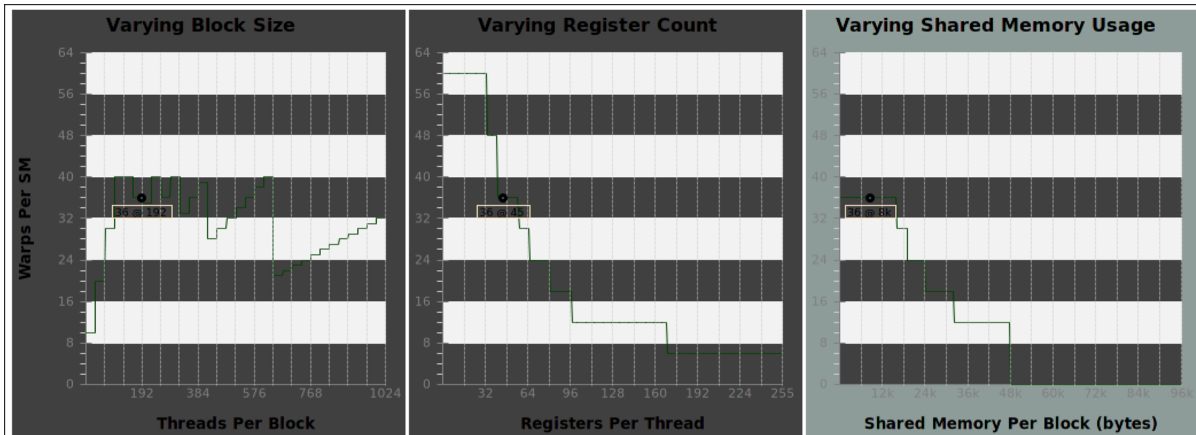


Figure 25: dwt2d (oneAPI) latency analysis

Overall performance

Intel oneAPI has the added advantage of not being restricted to one vendor like CUDA. For this reason the benchmarks were also tested in the Intel Devcloud using a pair of Intel XeonGold6128 CPU and an integrated Intel UHDGraphicsP630 GPU as devices.

As seen in Figure 26, although, as expected, the dedicated Nvidia GPU outperformed the CPU and the integrated GPU in almost every case, there are instances where the CPU and/or the integrated GPU matched or even outperformed the Nvidia GPU. This was the case when the application didn't take advantage of the massive parallelization capabilities of a GPU or the overhead caused by the movement of memory between the host and the device wasn't justified by the speedup provided by the accelerator.

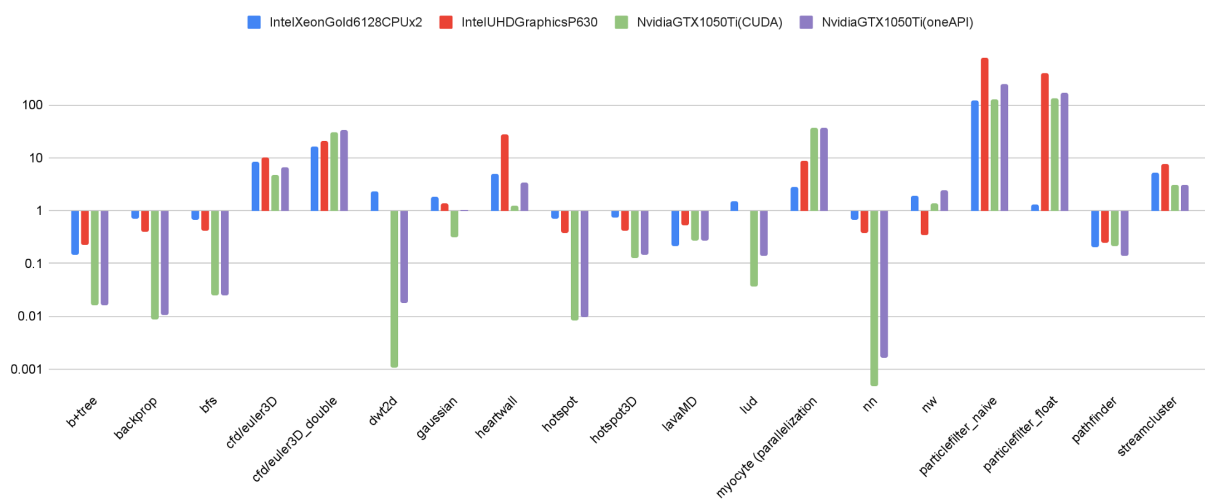


Figure 26: Total execution times across all tested devices (logarithmic scale)

Continuing with the example of nn that calculates the euclidean distance between a set of points, the execution was much faster in the dedicated GPU due to use of sqrt() and the high parallelization of the kernel. The integrated GPU takes advantage of this parallelization, thus outperforming the CPUs, but lacks the special compute units that the dedicated GPU has for calculating square roots.

By the other hand, the myocyte test is a clear example of a case where the speedup provided by the GPU does not justify the overhead introduced by the memory transfer. In this case the integrated GPU takes advantage of the reduced latency that provides being in the same chip as the host and outperforms the dedicated GPU, but the faster and more complex cores of the CPUs gives them the advantage in this case where the massive parallelization capabilities of the GPUs are not fully utilized.

Chapter 9: Conclusions

As stated in chapters 2 and 3, there is a clear tendency towards heterogeneity and, in recent years, many methods for programming these heterogeneous systems have appeared with greater or lesser success.

OneAPI aims to provide a simple and unified programming method for all these different heterogeneous systems, independently of the accelerator used or it's vendor, allowing high and low level approaches facilitating the programmer's work in this highly heterogeneous world we are moving towards.

The work done migrating, benchmarking and profiling the Rodinia Benchmark suite shows promising results:

- Although some work remains to be done, the Data Parallel C++ Compatibility Tool greatly streamlines the migration process from CUDA to oneAPI. Twenty out of the twenty three benchmarks were successfully migrated without major manual intervention, resulting in a success rate of almost 87% and allowing previously Nvidia specific code to run in other accelerators.

- Most of the problems encountered during the migration were caused by known bugs that will be fixed over time. These bugs were the main reason why the migration failed in hybridsort, kmeans and mummergpu.

- The performance of the resulting migrated code is, in many cases, comparable to the original CUDA source even without applying any optimization. Backprop, gaussian and streamcluster achieve a performance difference of 1% or less and bfs, euler3D_double, lavaMD, lud and pathfinder keep the overhead under 17%.

- Regarding the basic memory operations, device memory allocation, copy memory host to device, copy memory device to host and device memory free, the performance is equivalent and any overhead detected around these operations is caused by other instructions introduced by oneAPI.

- In the cases where the performance loss is more significant it is due to either poor memory and register usage or insufficient optimization of the compiled code. Both these problems should be solved, or at least mitigated, by a more mature and advanced compiler.

Even with the performance loss, the capability of executing the kernels in a multitude of different devices, not only Nvidia GPUs, is a great advantage over CUDA that justifies the effort put on the migration or, at least, makes it worth considering it. It is also worth noticing that oneAPI with the Intel XeonGold6128 CPUs achieved similar performance to the original CUDA version of particlefilter_naive and surpassed it in euler3D_double, myocyte and particlefilter_float.

Both, oneAPI and DPCT, are in active development. The results of this project were discussed with Intel's staff and most of the found problems and bugs were known by Intel; some fixes are even planned for the next update and further optimizations and compiler upgrades might reduce the observed performance losses.

All of this reaffirms the belief that oneAPI and SYCL will become a future standard for heterogeneous programming.

Bibliography

[1] "Apple M1 - Wikipedia" [Online]. Available: https://en.wikipedia.org/wiki/Apple_M1

[2] "List Statistics | TOP500" [Online]. Available: https://www.top500.org/statistics/list

[3] "Development over Time | TOP500" [Online]. Available: https://www.top500.org/statistics/overtime

[4] "CUDA Zone | NVIDIA Developer" [Online]. Available: https://developer.nvidia.com/cuda-zone

[5] Jacob Lambert, "Evolution of Programming Approaches for High-Performance Heterogeneous Systems", Computer Science University of Oregon, United States, December 14, 2020. Available: https://www.cs.uoregon.edu/Reports/AREA-202012-Lambert.pdf

[6] Pradeep Gupta, "CUDA Refresher: The CUDA Programming Model | Nvidia Developer Blog" [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model

[7] "SYCL Overview - The Khronos Group Inc" [Online]. Available: https://www.khronos.org/sycl/

[8] "Welcome to oneAPI - oneAPI Documentation" [Online]. Available: https://docs.oneapi.com/versions/latest/index.html

[9] "DPC++ Reference - DPC++ Reference Documentation" [Online]. Available: https://docs.oneapi.com/versions/latest/dpcpp/index.html#data-parallel-c-dpc

[10] "DPC++ Foundations Code Sample" [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/dpcpp-foundations-code-sample.html

[11] James Reinders, *et al.,* "Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL" Available: https://link.springer.com/book/10.1007%2F978-1-4842-5574-2

[12] "Data Parallel C++ USM Code Samples" [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/dpcpp-usm-code-sample.html

[13] "Get Started with the Intel DPC++ Compatibility Tool" [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-intel-dpcpp-compatibility-tool/top.html

[14] "start [Rodinia]" [Online]. Available: http://rodinia.cs.virginia.edu/doku.php

[15] "Intel DPC++ Compatibility Tool" [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compatibility-tool.html

[16] "Ruyk/dpcpp-cuda-examples-docker" [Online]. Available: https://github.com/Ruyk/dpcpp-cuda-examples-docker

[17] "Encuentro Online Desarrolladores Intel Software" [Online]. Available: https://www.danysoft.com/encuentro-online-desarrolladores-intel-software

[18] "Diagnostics Reference" [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top/diagnostics-reference.html

[19] "Neuromorphic Computing - Next Generation of AI" [Online]. Available: https://www.intel.com/content/www/us/en/research/neuromorphic-computing.html

[20] "Pandemic distorts global GPU market results | Jon Peddie Research" [Online]. Available: https://www.jonpeddie.com/press-releases/pandemic-distorts-global-gpu-market-results

[21] "NVIDIA Visual Profiler | NVIDIA Developer" [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler

[22] "Intel oneAPI Programming Guide - Glossary" [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/glossary.html

[23] "Thread block (CUDA programming) - Wikipedia" [Online]. Available: https://en.wikipedia.org/wiki/Thread_block_%28CUDA_programming%29

[24] "Intel oneAPI DPC++/C++ Compiler - Data Parallel C++ for Cross-Architecture Applications" [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compiler.html

[25] "Intel oneAPI Base Toolkit for Cross-Architecture Development" [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html

[26] "CUDA Binary Utilities :: CUDA Toolkit Documentation" [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref

Glossary

Accelerator: Specialized component containing compute resources that can quickly execute a subset of operations. Examples include CPU, FPGA, GPU [22].

Accessor: Communicates the desired location (host, device) and mode (read, write) of access [22].

Buffer: Memory object that communicates the type and number of items of that type to be communicated to the device for computation [22].

Compute unit: A grouping of processing elements into a 'core' that contains shared elements for use between the processing elements and with faster access than memory residing on other compute units on the device [22].

Device: An accelerator or specialized component containing compute resources that can quickly execute a subset of operations. A CPU can be employed as a device, but when it is, it is being employed as an accelerator. Examples include CPU, FPGA, GPU [22].

FPGA: Field-programmable gate array. It is an integrated circuit designed to be configured after manufacturing.

GPU: Graphics processing unit.

Kernel: Code that executes on the device [22].

Processing element: Individual engine for computation that makes up a compute unit [22].

Shared memory: Special memory region that can be accessed by all threads in the same group.

SIMD: Single instruction multiple data.

SOC: System on Chip.

TPU: Tensor Processing Unit. It is an AI accelerator for neural network machine learning.

Warp: A warp is a set of threads within a thread block such that all the threads in a warp execute the same instruction [23].

Workgroup: Collection of work-items that execute on a compute unit [22].

Work-item: Basic unit of computation in the oneAPI programming model. It is associated with a kernel which executes on the processing element [22].