UNIVERSITÀ DEGLI STUDI DI MESSINA
Dipartimento di Ingegneria

Corso di Laurea Magistrale in

Engineering and Computer Science (LM-18

LM-32)

Subject: Parallel Programming

OpenCL on the Raspberry Pi 3B+ Model

Student:
Abdirashova Aruzhan

Docente:
Prof. Salvatore Distefano

A.A. 2024

Contents

PARALLEL PROGRAMMING REPORT
OpenCL implementation on the Raspberry pi 3B+

1. Introduction

Raspberry Pi is a single-board computer often used in hobbyist projects as well as the controller of robots. It contains GPIO pins that are useful in bidirectional communication with multiple different peripherals over interfaces such as SPI, I2C and UART, as well as USB which allow to connect more common peripherals. It also has a CSI camera interface port with higher bandwidth which is useful for image processing and robotic vision. Currently, these tasks are often processed using OpenCV with the use of Raspberry Pi's ARM CPU and its SIMD extension – NEON.
The aim of this thesis is to measure performance and feasibility of the usage of Raspberry Pi 3 and its GPU in executing OpenCL applications. VideoCore IV – the GPU used in Raspberry Pi 3 and previous releases of this single-board computer – does not officially support OpenCL.
However, an open-source and community-developed driver exists. I would like to compare the performance of this driver varying the number of threads and adjusting parameters to analyze their impact on speedup and efficiency. What is more, I aim to identify optimal configurations for different workloads, highlighting the strengths and limitations of each processing unit.
Some of the experiments I would like to conduct are based on the research which was already done regarding the comparison of parallel computation performance between desktop and embedded solutions. The report will document the experimental setup, results, and key insights, providing a comprehensive understanding of parallel computing strategies in rendering applications.

2. OpenCL

OpenCL provides a programming language and runtime API to support the development of close-to-the-metal software which is both efficient and portable. Additionally, OpenCL provides low-level hardware abstractions that allow OpenCL implementations to expose many details of underlying hardware. These low-level abstractions are the platform, memory, and executions models described in the OpenCL specification. Understanding how these concepts translate into physical implementations on an FPGA is necessary for application optimization[1].
Programs written with this framework differ significantly from traditional C or Java programs. First, the program does not have an entry point as such. It is launched from the main program (host program). Secondly, the whole work is based on parallelization of calculations. That's why OpenCL programs are a description of a certain worker that will process some piece of data or task in parallel. In terms of OpenCL such a worker is called kernel. Thirdly, it is quite common to compile a kernel while the main host program is running.

Starting and working with OpenCL looks as follows:

- device initialization
- loading kernel code from a text file or from a binary blob
- kernel compilation
- initialization of clipboards
- in the main program loop:
    - copying data into GPU buffers
    - calculation of the required number of kernel calculations. Here you should take into account the supported parallelism of the device, the program algorithm and the amount of available data
    - kernel call
    - waiting for results
    - (optionally) calling another kernel to which the result of calculations of the first kernel is passed as input but with a different level of parallelism.

Of all this, the most important is parallelization of calculations.

2.1 OpenCL platform model

The OpenCL platform model defines the logical representation of all hardware capable of executing an OpenCL program. OpenCL platforms are defined by the grouping of a host processor and one or more OpenCL compute devices. The host processor, which runs the OS for the system, is also responsible for the general bookkeeping and task launch duties associated with the execution of OpenCL applications. The device is the hardware element in the system on which the compute kernels of an OpenCL application are executed. Each device is further divided into a set of compute units. The number of compute units depends on the target hardware. A compute unit is further subdivided into processing elements. A processing element is the fundamental computation engine in the compute unit, which is responsible for executing the operations of one work item[2].

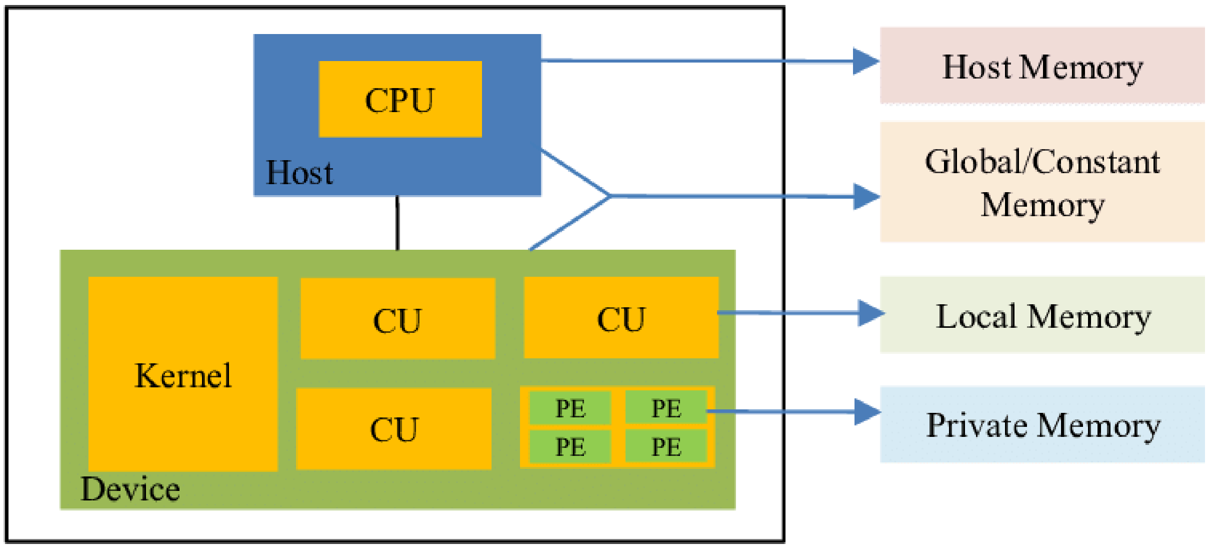


Figure-1. OpenCL platform model.[12]

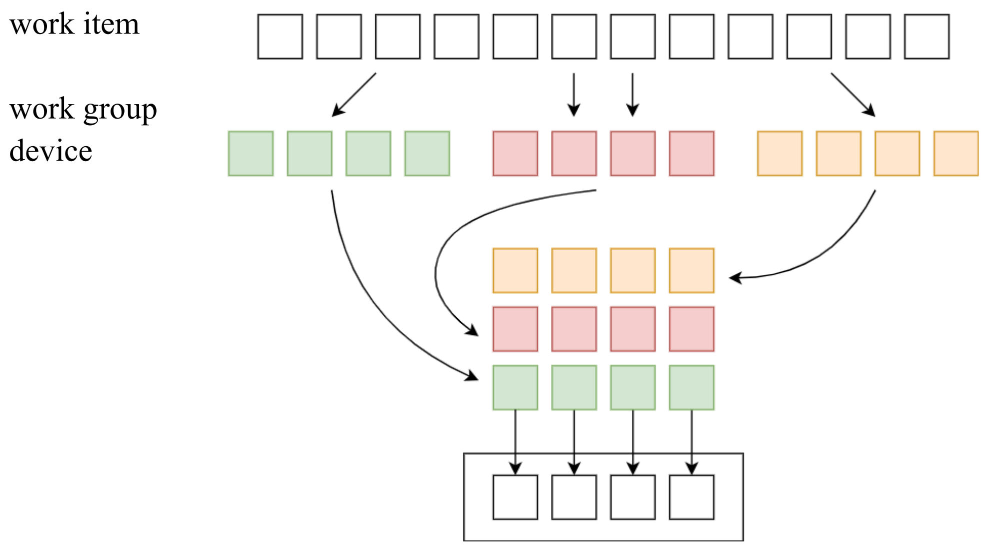
The preceding figure shows a conceptual view of the OpenCL platform model. An OpenCL platform always starts with a host processor. The host processor has the following responsibilities:

- Manage the operating system and enable drivers for all devices.
- Execute the application host program.
- Set up all global memory buffers and manage data transfer between the host and the device.
- Monitor the status of all compute units in the system.

Overall, here are several concepts in OpenCL:

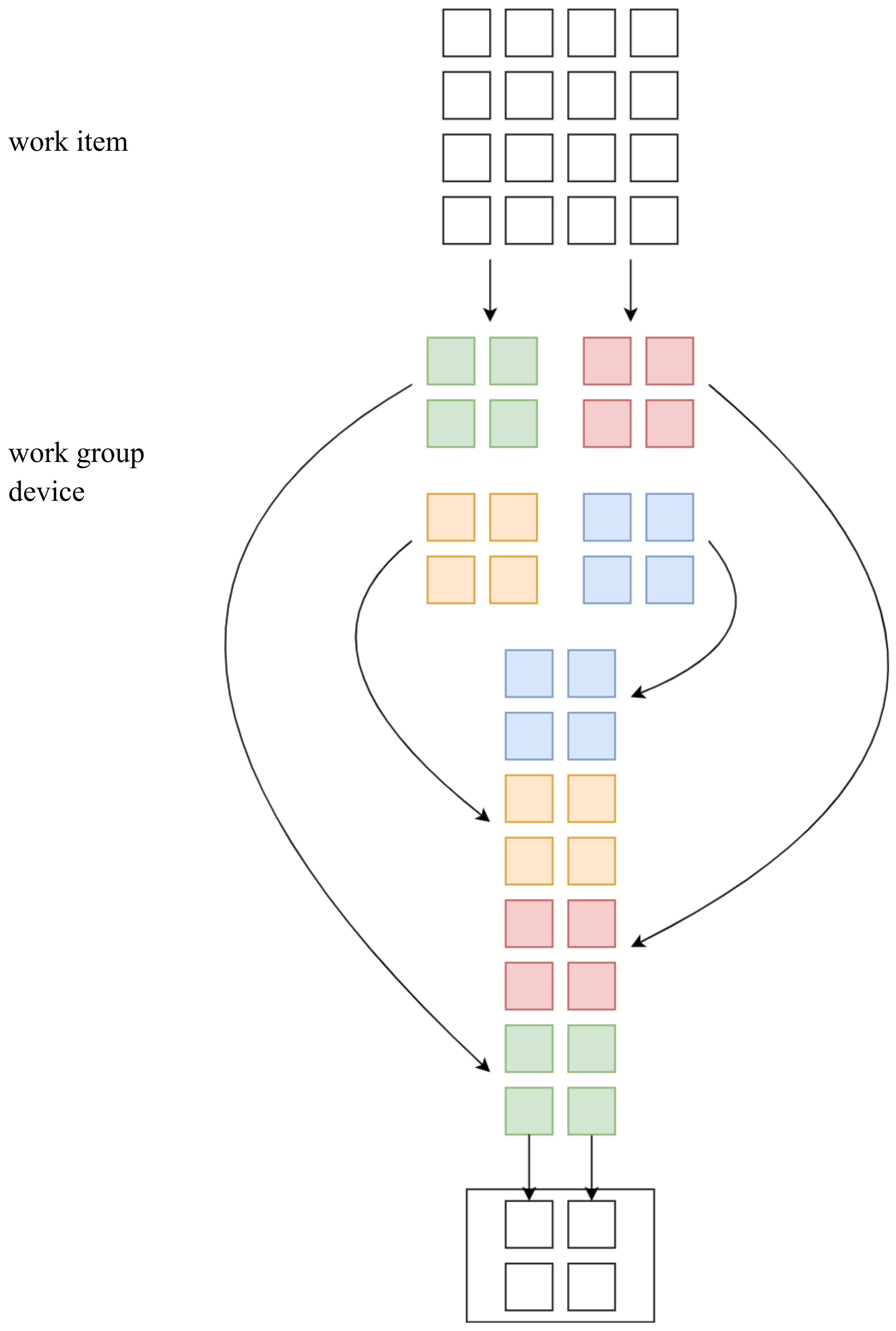
- work-item. This is a minimal and indivisible piece of work. In fact, kernel code processes exactly this one work-item. OpenCL implies that the programmer will divide data processing into such small pieces that will be executed in parallel.
- work-dim or work dimension. The dimensionality of the data to be processed. For value 1 the data is a vector, 2 - 2D, 3 - 3D.
- work group. Several work-items can be grouped into one work-group. The whole work is divided into several work groups and each work-item within such a group is run independently.

All this is quite difficult to imagine, so we try to visualize it.[14]

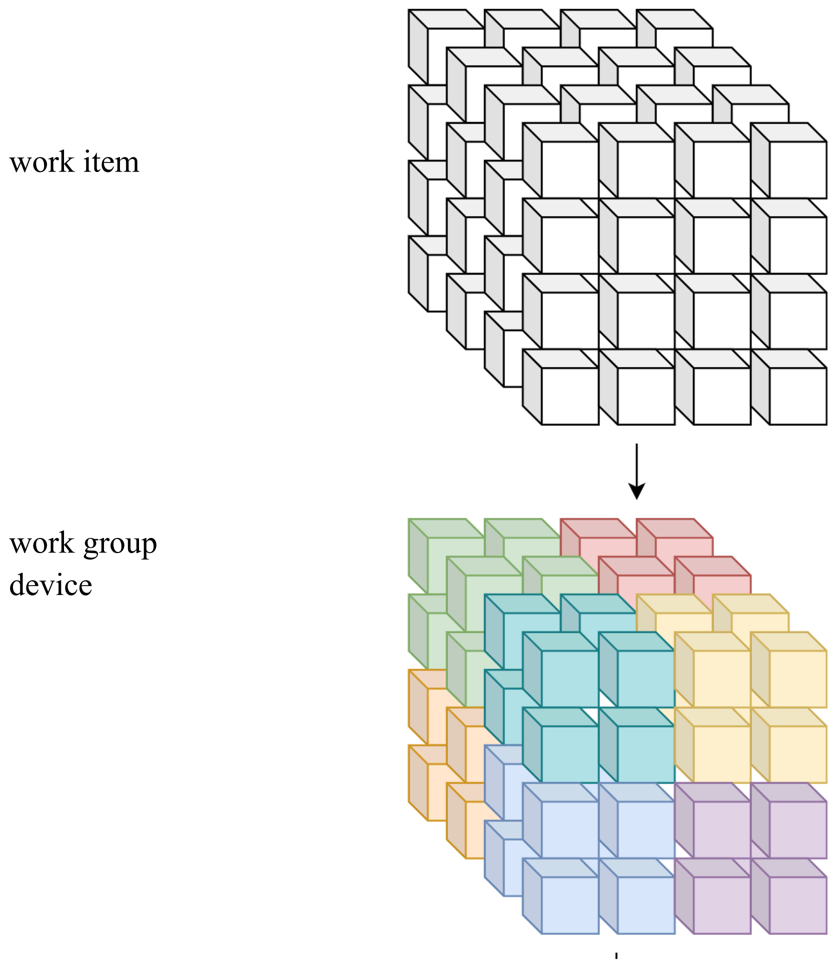


The task is broken down into work-items. These are such transparent squares at the top. After that it is grouped into a work group. And then work groups are run on parallel cores on the device.

So far everything is quite simple, but what programming on GPU without 2D? If work-dim=2, work-items can be part of a 2D array.



The problem is broken down into two-dimensional elements and further by analogy with one-dimensional vectors. An example of such a task is multiplication of matrices. Next, there is 3D.


work item
work group
device

3. Raspberry Pi

The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.
What's more, the Raspberry Pi has the ability to interact with the outside world, and has been used in a wide array of digital maker projects, from music machines and parent detectors to weather stations and tweeting birdhouses with infra-red cameras. We want to see the Raspberry Pi being used by kids all over the world to learn to program and understand how computers work[6].
For just $35 this credit-card sized Pi features an ARM processor, RAM, graphical capabilities, and all the standard hardware ports you would find on a computer.
Onboard storage does not come with the Raspberry Pi. The original Pi Models A and B used a full-size SD card but all newer models require a micro SD card. The quick start guide for the Pi recommends a microSD card with a sustained write speed of at least class 4. My personal preference is to always opt for the class 10 microSD card with at least 8 GB capacity. It depends on how much you plan to read/write but there has been a noticeable improvement and the cost difference is minimal. Though the microSD card capacity recommendation is 4 GB, this is typically related to image installations. When installing NOOBS be sure to use at least an 8 GB card. Here's an example of a class 10, 8 GB microSD card from SanDisk.

While the Raspberry Pi 3 and the Pi 2 Model B essentially look the same and cost the same, the Pi 3 is roughly 50% faster than the previous version. Both models have 1 GB RAM and both use a fourth-generation VideoCore GPU, but the quad-core 900 MHz CPU of the Pi 2 Model B was upgraded to a 1.2 GHz CPU on the Pi 3. Another exciting upgrade was the slight shifting of a few components on the board to allow for the long-awaited WiFi / Bluetooth SoC & Chip antenna. Gone are the days of using all of the precious USB ports on WiFi and Bluetooth dongles! See below for a chart comparing the main specifications of the Raspberry Pi 2 Model B and the Pi 3. Note the Raspberry Pi 3 is officially called "Pi 3 Model B" but there was never a "Pi 3 Model A".
GPIO plug-in expansion boards are available to provide additional or upgraded functionality such as sound/video cards, sensors, power relays, and more. You can also find other great peripherals like cameras, touch screens, GPS antennas, and even a retro USB NES controller. Typically when you get setup you will plug your keyboard and mouse into the USB ports on the Pi and connect a monitor via HDMI. Normally this results in booting up with your selected operating system and corresponding graphical user interface (GUI). However, there are projects that don't require the Raspberry Pi to use a normal GUI and allow you to operate in a "headless mode". In that case, it is possible to connect to the Pi over the network via SSH (Secure Shell). In fact, it's very likely that if you're running a Linux variation on your

Raspberry Pi, the SSH option is already available by default. If you're on a Mac, you can use Terminal or Putty on Windows. Using this method, you simply log into the Pi from your main computer and all the magic can be done through remote command line access. No need to have a monitor, keyboard or mouse directly connected to the Pi.
So, how to speed up signal processing on Raspberrypi? The first thing that comes to mind is to use SIMD instructions of the processor. They allow you to speed up the application by about 4 times. Instead of four multiplication operations, SIMD instructions allow you to perform one at a time on four floats.
However, Raspberrypi 1 does not support SIMD instructions. That's why the speed of work there is extremely low. perf_xlating test on Raspberrypi 3 is executed 0.024855 seconds, and on Raspberrypi 1 - 0.332934. That is an order of magnitude slower.
All Raspberrypi are built on SoCs from Broadcom. This means that one chip has both CPU and GPU. And the GPU is often hardly used at all! I tried to figure out if the speed of data processing would increase with the use of GPU. Programs for GPU are usually written in OpenCL. Or with the help of proprietary SDK, for example, Nvidia cuda toolkit. Under Raspberrypi there is VC4CL project, which implements ABI OpenCL for VideoCore IV GPU. Here and further I will give examples from this project.

3.1 VideoCore IV - the GPU of Raspberry Pi

Broadcom VideoCore IV, contained in the SoC of Raspberry Pi is quite a limited graphics processing unit. It contains 3 slices with 4 Quad Processing Units (a SIMD core) each. A QPU contains two different and asymmetric Arithmetic Logic Units - adder and multiplier, which provide just basic operations of adding, subtracting, multiplication, max, min, bit logic operations on32-bit floating point values and integers. However, multiplication on integers is limited to 24-bit. Additionally, each slice contains a Special Function Unit which is shared by 4 QPUs and provides various advanced mathematical functions like square root or logarithms. A QPU can process a quad vector in parallel and virtually 16-way vector in successive 4 clock cycles. In the whole
VideoCore IV there are 16 semaphores shared by all QPUs[8].
Because the entire GPU has only 12 QPUs, local work size for OpenCL is limited to overall 12 work-items. Anything above the limit is not scheduled. However it is proposed in design of OpenCL Framework for Embedded Multi-core processors, that in case of a larger local work size than the amount of hardware cores, the driver should enqueue multiple copies of the kernel.
Moreover the whole SoC has no embedded memory nor L3 cache, and shares DDR2 DRAM which limits the data transfers. In Raspberry Pi the GPU located is clocked at 400 MHz by default. The CPU's frequency can be controlled at each system start which can reduce power consumption but influence the overall performance[9].

3.2 Memory access

In contrast to powerful desktop graphics cards such as the Nvidia GTX 960, the VideoCore IV GPU does not have its own graphics memory on which data can be temporarily stored for calculations. Instead, the QPUs can access the main memory directly via the components of the VDW (VPM DMA Writer) and VCD (Vertex Memory DMA) via DMA (Direct Memory Access). In order to provide enough memory for graphical applications, the amount of memory reserved for the VideoCore IV is defined in the /boot/config.txt file in the Raspbian OS when booting. For example, OpenGL shaders or vertices for 3D applications are temporarily stored in this memory area for calculation in the QPUs. However, the QPUs can access not only the statically reserved area, but the entire working memory, which leads to a security problem. The OpenGL implementation in Mesa for the VideoCore IV GPU combats this problem by checking the shader code during compilation. However, since OpenCL kernels, in contrast to OpenGL shaders, receive the memory areas on which they perform their calculations at runtime via the kernel parameters, access to safetycritical areas cannot be controlled during compilation. To solve this problem, all programmes that use the OpenCL implementation in this work must therefore first be executed with root rights. This ensures that the user already has all rights and that there is no need to manipulate the system via OpenCL kernels.

3.3 Setup

A cluster of n Raspberry Pi 3 running each :

- Raspberry Pi OS (Legacy, 32-bit, A port of Debian Bullseye)
- VC4CL(OpenCL 3.0, Videocore 4)
- optional POCL to have a second OpenCL platform using the ARM CPU(in my case, I installed it, but did not use)

Testing VC4CL To verify that the OpenCL environment was correctly set up, clinfo was installed and executed, obtaining the following output:

```
Number of platforms                                    2
  Platform Name                        OpenCL for the Raspberry Pi VideoCore IV GPU
  Platform Vendor                                      doe300
  Platform Version                     OpenCL 1.2 VC4CL 0.4.9999 (b5a6097)
  Platform Profile                     EMBEDDED_PROFILE
  Platform Extensions                  cl_khr_il_program cl_khr_spir cl_khr_create_command_queue
cl_altera_device_temperature cl_altera_live_object_tracking cl_khr_icd cl_khr_extended_versioning
cl_khr_spirv_no_integer_wrap_decoration cl_khr_suggested_local_work_size cl_vc4cl_performance_counters
  Platform Extensions function suffix    VC4CL

  Platform Name                   Portable Computing Language
  Platform Vendor                 The pocl project
  Platform Version                          OpenCL 1.2 pocl 1.6, None+Asserts, LLVM 9.0.1, RELOC, SLEEF,
FP16, POCL_DEBUG
  Platform Profile                          FULL_PROFILE
  Platform Extensions                       cl_khr_icd
  Platform Extensions function suffix       POCL

  Platform Name                             OpenCL for the Raspberry Pi VideoCore IV GPU
Number of devices                           1
  Device Name                               VideoCore IV GPU
  Device Vendor                             Broadcom
  Device Vendor ID                          0x14e4
  Device Version                            OpenCL 1.2 VC4CL 0.4.9999 (b5a6097)
  Device Numeric Version                    0x402000 (1.2.0)
  Driver Version                            0.4.9999
  Device OpenCL C Version                   OpenCL C 1.2
  Device Type                               GPU
  Device Profile                            EMBEDDED_PROFILE
  Device Available                          Yes
  Compiler Available                        Yes
  Linker Available                          Yes
  Max compute units                         1
  Available core IDs                        0
gpuserv: vc_gpuserv_init: starting initialisation
  Max clock frequency                       300MHz
  Core Temperature (Altera)                 46 C
  Device Partition                          (core)
    Max number of sub-devices               0
    Supported partition types               None
    Supported affinity domains              (n/a)
  Max work item dimensions                  3
  Max work item sizes                       12x12x12
  Max work group size                       12
  Preferred work group size multiple (kernel)    1
  Preferred / native vector sizes
    char                                    16 / 16
    short                                   16 / 16
    int                                     16 / 16
    long                                    0 / 0
    half                                    0 / 0        (n/a)
    float                                   16 / 16
    double                                  0 / 0        (n/a)
  Half-precision Floating-point support     (n/a)
  Single-precision Floating-point support   (core)
    Denormals                               No
    Infinity and NANs                       No
    Round to nearest                        No
    Round to zero                           Yes
    Round to infinity                       No
    IEEE754-2008 fused multiply-add         No
    Support is emulated in software         No
    Correctly-rounded divide and sqrt operations  No
  Double-precision Floating-point support   (n/a)
  Address bits                              32, Little-Endian
  Global memory size                           67108864 (64MiB)
  Error Correction support                     No
  Max memory allocation                         33554432 (32MiB)
  Unified memory for Host and Device              Yes
  Minimum alignment for any data type             64 bytes
  Alignment of base address                     512 bits (64 bytes)
  Global Memory cache type                       Read/Write
  Global Memory cache size                       32768 (32KiB)
  Global Memory cache line size                   64 bytes
  Image support                             No
  Local memory type                            Global
  Local memory size                           67108864 (64MiB)
  Max number of constant args                    32
  Max constant buffer size                     67108864 (64MiB)
  Max size of kernel argument                    256
  Queue properties
    Out-of-order execution                     No
    Profiling                               Yes
  Prefer user sync for interop                  Yes
  Profiling timer resolution                   1ns
  Execution capabilities
    Run OpenCL kernels                          Yes
    Run native kernels                        No
    IL version                              SPIR-V_1.5 SPIR_1.2
    ILs with version                          SPIR                                        0x402000 (1.2.0)
                                          SPIR-V                                      0x405000 (1.5.0)
    SPIR versions                             1.2
  printf() buffer size                        0
  Built-in kernels                            (n/a)
  Built-in kernels with version                   (n/a)
  Device Extensions                            cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
cl_khr_byte_addressable_store cl_nv_pragma_unroll cl_arm_core_id cl_ext_atomic_counters_32
cl_khr_initialize_memory cl_arm_integer_dot_product_int8 cl_arm_integer_dot_product_accumulate_int8
cl_arm_integer_dot_product_accumulate_int16 cl_arm_integer_dot_product_accumulate_saturate_int8
cl_khr_expect_assume cl_khr_il_program cl_khr_spir cl_khr_create_command_queue
cl_altera_device_temperature cl_altera_live_object_tracking cl_khr_icd cl_khr_extended_versioning
cl_khr_spirv_no_integer_wrap_decoration cl_khr_suggested_local_work_size cl_vc4cl_performance_counters
  Device Extensions with Version    cl_khr_global_int32_base_atomics    0x400000 (1.0.0)
                                    cl_khr_global_int32_extended_atomics    0x400000 (1.0.0)
                                    cl_khr_local_int32_base_atomics          0x400000 (1.0.0)
                                    cl_khr_local_int32_extended_atomics      0x400000 (1.0.0)
                                    cl_khr_byte_addressable_store           0x400000 (1.0.0)
                                    cl_nv_pragma_unroll                            0 (0.0.0)
                                    cl_arm_core_id                          0x800000 (2.0.0)
                                    cl_ext_atomic_counters_32               0x1400000 (5.0.0)
                                    cl_khr_initialize_memory                0x400000 (1.0.0)
                                    cl_arm_integer_dot_product_int8         0xc00000 (3.0.0)
                   cl_arm_integer_dot_product_accumulate_int8               0xc00000 (3.0.0)
                   cl_arm_integer_dot_product_accumulate_int16              0xc00000 (3.0.0)
                   cl_arm_integer_dot_product_accumulate_saturate_int8      0xc00000 (3.0.0)
```

```
                                        cl_khr_expect_assume                   0x400000 (1.0.0)
                                        cl_khr_il_program                      0x400000 (1.0.0)
                                        cl_khr_spir                            0x400000 (1.0.0)
                                        cl_khr_create_command_queue            0x400000 (1.0.0)
                                        cl_altera_device_temperature                  0 (0.0.0)
                                        cl_altera_live_object_tracking                0 (0.0.0)
                                        cl_khr_icd                                0x400000 (1.0.0)
                                        cl_khr_extended_versioning                0x400000 (1.0.0)
                                        cl_khr_spirv_no_integer_wrap_decoration         0 (0.0.0)
                                        cl_khr_suggested_local_work_size          0x400000(1.0.0)
                                        cl_vc4cl_performance_counters                   0 (0.0.0)

  Platform Name                              Portable Computing Language
Number of devices                              1
  Device Name                               pthread-cortex-a53
  Device Vendor                             ARM
  Device Vendor ID                            0x13b5
  Device Version                            OpenCL 1.2 pocl HSTR:
pthread-armv6k-unknown-linux-gnueabihf-arm1156t2f-s
  Driver Version                            1.6
  Device OpenCL C Version                        OpenCL C 1.2 pocl
  Device Type                               CPU
  Device Profile                            FULL_PROFILE
  Device Available                            Yes
  Compiler Available                            Yes
  Linker Available                            Yes
  Max compute units                             4
  Max clock frequency                           1200MHz
  Device Partition                           (core)
    Max number of sub-devices                       4
    Supported partition types                   equally, by counts
    Supported affinity domains                    (n/a)
  Max work item dimensions                          3
  Max work item sizes                         4096x4096x4096
  Max work group size                         4096
  Preferred work group size multiple (kernel)     8
  Preferred / native vector sizes
    char                                    16 / 16
    short                                    8 / 8
    int                                     4 / 4
    long                                     2 / 2
    half                                    8 / 8       (cl_khr_fp16)
    float                                   4 / 4
    double                                    2 / 2       (cl_khr_fp64)
  Half-precision Floating-point support          (cl_khr_fp16)
    Denormals                               No
    Infinity and NANs                          No
    Round to nearest                          No
    Round to zero                            No
    Round to infinity                         No
    IEEE754-2008 fused multiply-add                No
    Support is emulated in software             No
  Single-precision Floating-point support         (core)
    Denormals                                      No
    Infinity and NANs                              Yes
    Round to nearest                               Yes
    Round to zero                                  No
    Round to infinity                              No
    IEEE754-2008 fused multiply-add                No
    Support is emulated in software                No
    Correctly-rounded divide and sqrt operations  No
  Double-precision Floating-point support         (cl_khr_fp64)
    Denormals                                      Yes
    Infinity and NANs                              Yes
    Round to nearest                               Yes
    Round to zero                                  Yes
    Round to infinity                              Yes
    IEEE754-2008 fused multiply-add                Yes
    Support is emulated in software                No
  Address bits                                     32, Little-Endian
  Global memory size                               585047040 (557.9MiB)
  Error Correction support                         No
  Max memory allocation                            268435456 (256MiB)
  Unified memory for Host and Device               Yes
  Minimum alignment for any data type              128 bytes
  Alignment of base address                        1024 bits (128 bytes)
  Global Memory cache type                         None
  Image support                                    Yes
    Max number of samplers per kernel              16
    Max size for 1D images from buffer             16777216 pixels
    Max 1D or 2D image array size                  2048 images
    Max 2D image size                              8192x8192 pixels
    Max 3D image size                              2048x2048x2048 pixels
    Max number of read image args                  128
    Max number of write image args                 128
  Local memory type                                Global
  Local memory size                                524288 (512KiB)
  Max number of constant args                      8
  Max constant buffer size                         524288 (512KiB)
  Max size of kernel argument                      1024
  Queue properties
    Out-of-order execution                         Yes
    Profiling                                      Yes
  Prefer user sync for interop                     Yes
  Profiling timer resolution                       1ns
  Execution capabilities
    Run OpenCL kernels                             Yes
    Run native kernels                             Yes
  printf() buffer size                             16777216 (16MiB)
  Built-in kernels                                 (n/a)
  Device Extensions                                cl_khr_byte_addressable_store cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
cl_khr_3d_image_writes cl_khr_fp16 cl_khr_fp64
```

```
NULL platform behavior
  clGetPlatformInfo(NULL, CL_PLATFORM_NAME, ...)  OpenCL for the Raspberry Pi VideoCore IV GPU
  clGetDeviceIDs(NULL, CL_DEVICE_TYPE_ALL, ...)   Success [VC4CL]
  clCreateContext(NULL, ...) [default]            Success [VC4CL]
  clCreateContext(NULL, ...) [other]              Success [POCL]
  clCreateContextFromType(NULL, CL_DEVICE_TYPE_DEFAULT)  Success (1)
    Platform Name                                 OpenCL for the Raspberry Pi VideoCore IV GPU
    Device Name                                   VideoCore IV GPU
  clCreateContextFromType(NULL, CL_DEVICE_TYPE_CPU)  No devices found in platform
  clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU)  Success (1)
    Platform Name                                 OpenCL for the Raspberry Pi VideoCore IV GPU
    Device Name                                   VideoCore IV GPU
  clCreateContextFromType(NULL, CL_DEVICE_TYPE_ACCELERATOR)  No devices found in platform
  clCreateContextFromType(NULL, CL_DEVICE_TYPE_CUSTOM)  No devices found in platform
  clCreateContextFromType(NULL, CL_DEVICE_TYPE_ALL)  Success (1)
    Platform Name                                 OpenCL for the Raspberry Pi VideoCore IV GPU
    Device Name                                   VideoCore IV GPU

ICD loader properties
  ICD loader Name                                 OpenCL ICD Loader
  ICD loader Vendor                               OCL Icd free software
  ICD loader Version                              2.2.14
  ICD loader Profile                              OpenCL 3.0
```

4. Algorithm

Kernel Codes:

sum.cl:

```
__kernel void sum(__global const float16 *a_g, __global const float16 *b_g, __global
float16 *res_g) {
   int gid = get_global_id(0);
   res_g[gid] = a_g[gid] + b_g[gid];
}
```

sum_mul.cl:

```
__kernel void sum_mul(__global const float16 *a_g, __global const float16 *b_g, __global
float16 *res_add, __global float16 *res_mul) {
   int gid = get_global_id(0);
   res_add[gid] = a_g[gid] + b_g[gid];
   res_mul[gid] = a_g[gid] * b_g[gid];
}
```

The OpenCL kernels you are written in the OpenCL language. OpenCL host code is written in Python programming language(PyOpenCL).

4.1 Decomposition

A problem can be parallelized only if it can be divided into independent subproblems. If the problem can be divided, it's possible to use a decomposition method.

Two main decomposition methods:

1. Divide-and-conquer

• iteratively break a problem into smaller subproblems until the subproblems fit well on the computational resources provided

2. Scatter-gather

• send a subset of the input data to each parallel resource, and then collect the results of the computation and combine them into a result data set

Perform element-wise operations on two large vectors:

- **Vector Addition (sum Kernel)**: Each element of the result vector res_add is computed as the sum of corresponding elements from input vectors a and b.
- **Vector Addition and Multiplication (sum_mul Kernel)**: Each element of the result vectors res_add and res_mul is computed in parallel.

Task Parallelism: Each addition and multiplication operation is independent. Therefore, both operations can be parallelized effectively, with each task corresponding to a single element computation. It's possible to create indipendent subproblems. This code possesses significant data-level parallelism because it's possible to perform the same operation in parallel to all the elements of A and B to produce C.

In the vector addition example, each chunk of data could be executed as an independent thread. On modern CPUs, the overhead of creating threads is so high that the chunks need to be large. In practice, usually a few threads (about as many as the number of CPU cores) and each is given a large amount of work to do. For GPU programming, there is low overhead for thread creation, so we can create one thread per loop iteration.

.

Communication

The execution model specifies that devices perform tasks based on commands which are sent from the host to the device. This includes data transfer, synchronization, and handling of results.

A command-queue is the communication mechanism that the host uses to request action by a device. Once the host has decided which devices to work with and a context has been created, one command-queue needs to be created per device. The API function clCreateCommandQueue() (deprecated in OpenCL 2.0 and substituted by clCreateCommandQueueWithProprierties) is used to create a command-queue.

CPU Implementation

- Data Transfer: No inter-task communication is required as each addition and multiplication is independent.
- Data Handling: Input vectors are processed in-memory, and results are stored back in-memory.

GPU Implementation

1. Host to Device Transfer: Input vectors A and B are transferred from host (CPU) memory to device (GPU) memory. Create buffers on the GPU for input vectors (a_g and b_g) and result vectors (res_add_g and res_mul_g for sum_mul, and res_g for sum).

2. Parallel Computation:
    - The sum kernel is executed to compute the addition results.
    - The sum_mul kernel is executed to compute both addition and multiplication results.
3. Device to Host Transfer: Results from the GPU (i.e., res_add and res_mul from sum_mul, or res_add from sum) are copied back to the host memory for further processing or validation. Ensure that data transfers and kernel executions are complete before accessing the results.

Aggregation

For both kernels, the results are collected in the GPU buffers (res_add_g and res_mul_g for sum_mul, res_g for sum).
After kernel execution, these results are copied back to host memory where they are aggregated for validation or further processing. The results from the GPU are compared with CPU computations to ensure correctness and to validate the accuracy of the GPU results.

4.2 Mapping

Thread mapping determines which threads will access which data
– Proper mappings can align with hardware and provide large performance benefits.
– Improper mappings can be disastrous to performance.

Kernel Mapping:

- Work Items and Work Groups:
    - Each work-item (thread) is assigned to compute one element of the vector. This is defined by the get_global_id() function in OpenCL.
    - Local Work Size: Specifies how many work-items are grouped into a local work group. Adjusting this can influence performance and efficiency.
- Global Work Size: Defines the total number of work-items to execute, which corresponds to the size of the vector (vector_size in the provided code).

Execution Strategy:

- Local Work Size Variations: The local work size can be varied to determine the optimal configuration for performance. For example:
    - Local Work Size: 12
    - Local Work Size: 10
    - Local Work Size: 8
- Global Work Size Adjustment: For each local work size, the global work size is adjusted to be a multiple of the local work size to ensure complete coverage of the vector elements.

So for each work-item we can define two types of identifier:
- global-id: A unique global ID given to each work item in the global NDRange
- local-id: A unique local ID given to each work item within a work group

5. Benchmark-metrics

In this section, we are going to compare Speedup and Efficiency of OpenCL implementations. As the VC4CL implementation does not support double or half data types, these benchmarks are not executed. The integer benchmark is also skipped, as the code generated for this is still faulty and causes the QPUs used to freeze. Calculations are performed simultaneously on all twelve QPUs. The results are compared with the results of the benchmarks on the CPU-side OpenCL implementation pocl.

On the one hand, the pure computing power on the GPU is significantly higher than for equivalent calculations on the CPU. However, this comes at the cost of a slower memory connection. Taking into account that the times are measured around the function for starting a kernel and therefore include the overhead for creating the memory for the kernel code , copying the kernel code and determining the work item parameters, the results obtained are very satisfactory.

In addition to the overhead of starting and terminating kernel code, most performance losses can be attributed to a handful of points: Some frequently used arithmetic operations, as well as division (of integers and floating point numbers) and the general form of multiplication of integers are not supported by the instruction set of the QPUs and must therefore be emulated in software. As a result, however, these operations require many more clock cycles than a calculation in hardware would need. However, if the valid value range of the operands is restricted, the compiler can be helped to generate more efficient implementations of the operations.

Another bottleneck is the memory access: Since all QPUs share a VPM and thus the main memory access and the VPM can only handle one read and one write operation at a time, each access to the main memory is synchronised via the hardware mutex. This means that if a kernel is executed in parallel on all 12 available QPUs, eleven of the QPUs must wait until the first processor has completed its memory access, at least for the first memory access. The QPU that is last in line is even blocked for eleven memory accesses, which in turn results in more than 100 clock cycles of blocking time. Memory accesses cannot be avoided as OpenCL kernels cannot have return values and the parameters usually only contain the addresses of the memory areas to be used and constant values. However, an attempt can be made to minimise memory accesses by reading the required values inital and writing them back to the main memory at the end, while all calculations are performed on local variables

Therefore, another optimization is also used to reduce the influence of this overhead: The executed code of an OpenCL kernel is the same for all work items of all work groups. Only the parameters change between the executed instances of a kernel. It is therefore a good idea to start several instances in succession with one system call, each of which only has modified parameters. To do this, a loop is added around the kernel's machine code, which repeats the complete execution. A further "hidden" parameter is used to determine how often the kernel code should be executed.

In the picture below is shown programming steps required to prepare an OpenCL kernel to work, starting with creating a context and a command queue to instantiate kernels. Then, the input data is prepared by allocating buffers on the host side, copying data from the host memory to the device memory, and dispatching the kernel.

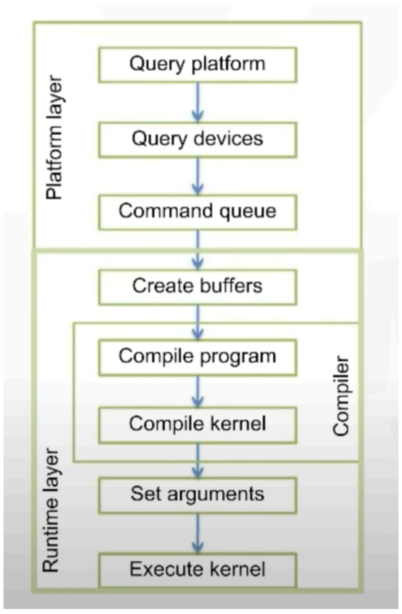


Figure-2. Programming steps.[13]

This example compares the timings of adding vectors on the CPU versus adding vectors on the GPU varying the number of threads, the latter of which has different implementations.

The code runs the following implementations of adding large vectors. The vectors are added together 10000 times.

6. Experimental setup and results

All experiments were conducted on the Raspberry Pi 3B+ model, which features a quad-core ARM Cortex-A53 CPU and the Broadcom VideoCore IV GPU. The software environment included the VC4CL library for OpenCL support on the GPU and POCL for the CPU, both adhering to OpenCL 1.2 specifications.

The primary algorithms used for this evaluation were:

- **Vector Addition** (sum kernel)
- **Vector Addition and Multiplication** (sum_mul kernel)

Both algorithms were parallelized using OpenCL, and benchmarks were run for varying local work sizes: 12, 10, and 8. The results were compared in terms of execution time, speedup, and efficiency. The experiments aimed to evaluate how different work sizes affected performance on the GPU, while also comparing these results to the CPU implementation.

Vector Addition (sum Kernel):

The sum kernel was executed for element-wise addition on both the CPU and GPU. The performance for different local work sizes is summarized below:

Local work size	T(Videocore 4)	T(cpu)	Speedup	Efficiency
(12,)	6.854ms	13.77ms	2.01	0.145
(10,)	5.233ms	13.78ms	2.633	0.19
(8,)	4.852ms	13.66ms	2.815	0.206

As we can see, it shows that the GPU outperforms the CPU significantly, especially for smaller work sizes, due to better resource utilization and reduced overhead. The local work size (8,) achieved the best speedup and efficiency, as it aligns more closely with the GPU's internal architecture, leading to less idle time for its QPUs.

Vector Addition and Multiplication (sum_mul Kernel):

The sum_mul kernel performs both addition and multiplication. The CPU and GPU performance comparisons are provided below:

Local work size	T(Videocore 4)	T(cpu)	Speedup	Efficiency
(12,)	6.892ms	27.93ms	4.05	0.33

(10,)	5.579ms	27.73ms	4.97	0.179
(8,)	5.405ms	27.11ms	5.01	0.184

As observed in the table, the GPU demonstrates a more significant performance advantage in this benchmark, with a 5 times speedup for the local work size (8,) configuration. This suggests that the GPU's parallel architecture is particularly well-suited for operations that involve multiple calculations on the same data.

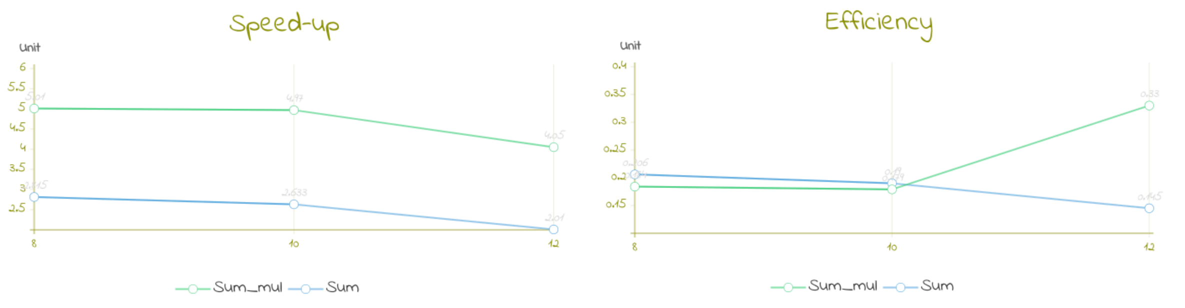


Figure-3. Speed-up and Efficiency performance metrics comparisons.

In our investigation of the optimal local work size for kernel execution on the VideoCore IV GPU, we observed that smaller work sizes, such as (8,), consistently outperformed larger sizes like (12,). This suggests that the GPU's architecture favors smaller work groups due to better resource utilization, more efficient memory access patterns, and reduced synchronization overhead. Consequently, for applications targeting the VideoCore IV GPU, it is recommended to experiment with and prefer smaller local work sizes to achieve optimal performance.

While the GPU outperforms the CPU in both benchmarks, it is important to note that memory access remains a significant bottleneck for the GPU. The VideoCore IV architecture, with only 12 QPUs, shares a single memory access channel, meaning that simultaneous memory access by multiple QPUs can lead to substantial waiting times. The results suggest that reducing memory access operations or improving memory access patterns could further enhance the performance of the GPU.

If additional experiments were conducted with more complex algorithms, such as matrix multiplication, it is expected that the GPU would exhibit even more significant speedups due to its parallel nature. For example, in matrix multiplication, where data-level parallelism is high, GPUs typically outperform CPUs by a large margin because the problem can be decomposed into a large number of independent subproblems.

7. Conclusion

In my project I implemented OpenCL on Raspberry Pi’s VideoCore IV GPU, using VC4CL library. I showed why OpenCL and distributed computing are needed at all. My aim was to understand whether the GPU can replace or at least approach the CPU in terms of performance when performing simple benchmarks. The experiments demonstrate the effectiveness of GPU acceleration for vector operations. Although the speedup improves with larger thread counts, efficiency tends to decrease due to overheads associated with managing more threads. Thus, the choice of local work size should balance performance gains with management overhead to achieve optimal results.

Despite the GPU's clear advantages, memory access remains a critical bottleneck, suggesting that further optimizations in memory handling could lead to even greater performance improvements. Future work could involve exploring more complex algorithms and heterogeneous computing setups, where both CPU and GPU are utilized in parallel for even greater computational throughput.

8. Library

1.https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/pet1504034296131.html
2.https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/fpn1504034296297.html
3.https://github.com/dernasherbrezon/clDsp-test.git
4.https://downloads.ti.com/mctools/esd/docs/opencl/optimization/host_code.html
5.https://yunusmuhammad007.medium.com/?p=5147e8a5f053
6.https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/#:~:text=It's%20capable%20of%20doing%20everything,%2Dprocessing%2C%20and%20playing%20games.
7.https://www.arrow.com/en/research-and-events/articles/top-ten-things-to-know-about-the-raspberry-pi-3
8. Jung-Hyun Hong, Young-Ho Ahn, Byung-Jin Kim and Ki-Seok Chung, "Design of OpenCL
Framework for Embedded Multi-core Processors", IEEE, 2014.
9. VideoCore IV 3D Architecture Guide, Broadcom Corporation, 2013
https://docs.broadcom.com/doc/12358545 (access: 07.09.2020).
10. https://abhitronix.github.io/2019/01/15/VC4CL-1/
11. https://qengineering.eu/install-opencl-on-raspberry-pi-3.html
12.https://www.researchgate.net/publication/313872353_Efficient_FPGA_Implementation_of_OpenCL_High-Performance_Computing_Applications_via_High-Level_Synthesis
13. https://www.youtube.com/watch?v=V4buZh_ttjc&ab_channel=YanLuo
14.https://www.lrde.epita.fr/~carlinet/cours/GPGPU/IR%20GPU%20Computing%20-%20lecture%202%20-%202.%20KITE-OpenCL-course