Article

Darknet on OpenCL: a multi-platform tool for object detection and classification

Piotr Sowa [1,3,†,‡], **Jacek Izydorczyk** [2,‡]

1 Aptiv Technical Center, Kraków, Poland; piotr.sowa@aptiv.com
2 Silesian University of Technology, Gliwice, Poland; jacek.izydorczyk@ieee.org
3 Silesian University of Technology, Gliwice, Poland; piotr.sowa@polsl.pl
‡ These authors contributed equally to this work.

Abstract: The article's goal is to overview challenges and problems on the way from the state of the art CUDA accelerated neural networks code to multi-GPU code. For this purpose, the authors describe the journey of porting the existing in the GitHub, fully-featured CUDA accelerated Darknet engine to OpenCL. The article presents lessons learned and the techniques that were put in place to make this port happen. There are few other implementations on the GitHub that leverage the OpenCL standard, and a few have tried to port Darknet as well. Darknet is a well known convolutional neural network (CNN) framework. The authors of this article investigated all aspects of the porting and achieved the fully-featured Darknet engine on OpenCL. The effort was focused not only on the classification with the use of YOLO1, YOLO2, and YOLO3 CNN models. They also covered other aspects, such as training neural networks, and benchmarks to look for the weak points in the implementation. The GPU computing code substantially improves Darknet computing time compared to the standard CPU version by using underused hardware in existing systems. If the system is OpenCL-based, then it is practically hardware independent. In this article, the authors report comparisons of the computation and training performance compared to the existing CUDA-based Darknet engine in the various computers, including single board computers, and, different CNN use-cases. The authors found that the OpenCL version could perform as fast as the CUDA version in the compute aspect, but it is slower in memory transfer between RAM (CPU memory) and VRAM (GPU memory). It depends on the quality of OpenCL implementation only. Moreover, loosening hardware requirements by the OpenCL Darknet can boost applications of DNN, especially in the energy-sensitive applications of Artificial Intelligence (AI) and Machine Learning (ML).

Keywords: neural network; object detection; object classification; Darknet; programming.

1. Introduction

With the continued development of the semiconductor industry and improved computer architectures and the spread of cloud computing, deep learning neural networks (DNNs) have emerged as the most effective tool for machine learning (ML) and artificial intelligence (AI) [1]. Even the most ordinary devices are getting the computing power available in the past only to supercomputers [2]. All wirelessly connected devices collect petabytes of data that allow detecting objects and processes on an unprecedented scale [3]. The most notable examples are automatic face recognition [4], classification of hyperspectral data [5], automatic object detection and classification [6–8], remote gesture sensing [9–11], wireless detection, and location [12–14] – all powered by internet data. Many of these applications are the kind of regression problem for which DNN is the right solution [15]. Limits on the cloud's capacity forces to disperse computations to fog, so even the smallest device could be equipped with intelligence or merely with DNN [16]. The latter is currently mainly deployed on NVIDIA GPU-accelerated computers [33]. The proliferation of DNN aided sensing systems depends on solving two problems [17]. The first is to find new neural network architectures and learning methods to

 doi:10.20944/preprints202007.0506.v1

reduce the computational effort needed to deploy and maintain DNN. The second is to develop tools for multi-platform DNN implementation to enable the use of the computational power available now and soon. The article covers the latter problem. The authors overview the porting techniques and challenges that are needed to turn the CUDA DNN framework into an OpenCL-based framework on the example of the open-source Darknet neural network engine [18]. After about two years of the project and interactions in the AI and GPU Computing communities, the authors' goal is to summarise the experience and share it with the readers [38].

The organisation of the material is as follows. In Section 2, the authors discuss the overall GPU' architecture to help the reader understand the difference between the CPU and the GPU. The OpenCL library for GPU computing is then briefly discussed. The next element is the basic description of the neural networks for deep learning and the Darknet engine, followed by the advantages of the YOLO model. In section 3, the main challenges of the code porting are listed and illustrated with concise examples. The entire Darknet porting process is discussed and explained. Section 4 presents the experimental testbed used. Section 5 is dedicated to code testing and benchmarking. Section 6 contains conclusions. The project is open-source and available on GitHub [20].

2. Basics

2.1. The GPU Generic Architecture

When thinking about the architecture of any GPU device, the best way to understand it's uniqueness is to compare with CPU' architecture, designed for long-running tasks consisting of threads and processes in the Operating System. Any modern CPU has several compute cores fed by memory organized in 3 levels of cache memory: L1, L2, L3; L1 is the smallest and the fastest level. Each core is a relatively sophisticated computing machine organized to uncover independent instructions fetched from memory in a stream and execute these instructions as quickly as possible in parallel.

The CPU' core architecture is comparable to a big Walmart supermarket [29]. Clients (instructions) enter the market (the core) sequentially and spread over the shopping hall to find a proper stand or store shelf or locker (execution unit) to realize the purpose of arrival. One of them (instructions) takes something from the market, and another brings something to transform or leave for the next client (instruction). Customers (instructions) queue up for service at the stands or wait for the arrival of the goods (data). When the client accomplishes the mission, he (the instruction) queue up to pay and exit the market by vast exit doors in the same order as he entered. The goal of the core is to create an illusion that the execution of instructions continues strictly sequentially as they are ordered in the code. In reality, instructions are executed by the core in parallel, even though they are mutually dependent in a quite unpatterned manner. The cores run threads and tasks that are connected quite loosely but collectively make the impression to the human that the threads are one centrally managed computational system.

Now when we want to compare this to GPUs, we have thousands of cores. Each core executes instructions in the same order as they are fetched from local memory. Instructions run through a pipeline and transform the data downloaded from local memory. If the code consists of a myriad of loosely related tasks, which run in parallel, and we can repeat the execution and data exchange episodes in a well-established pattern, then we can schedule usable work for each of the thousands of cores of the GPU. This way, the task can be completed quickly. There is one more important thing to consider regarding the GPU: video RAM (VRAM) and system RAM (RAM). To allow fast operations on GPUs, we have to transfer data from RAM to VRAM, because a GPU has no access to RAM itself.

In essence, CPUs and GPUs execute simultaneously to achieve a synergy effect. The above overview seems to give, in a nutshell, a sufficient understanding of the CPU/GPU general architecture for practical computation usage.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

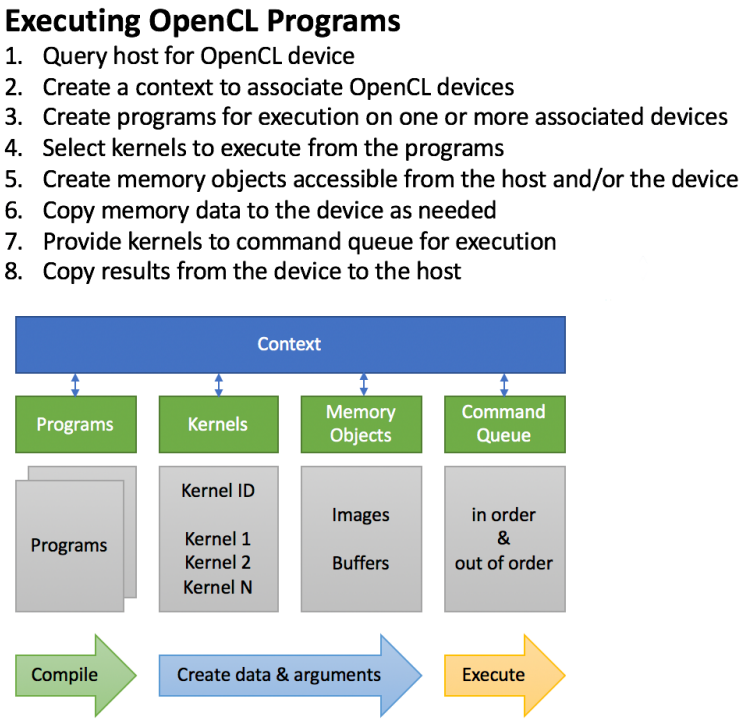

Figure 1. OpenCL Program Execution [30,31].

2.2. *OpenCL Library for GPU Computing*

The OpenCL abstraction fits all modern GPUs and uses context, queue, kernel, and memory buffer abstractions. This method seems to be very bright because context covers all GPUs in the system, a queue orders computing tasks on each GPU, and a kernel is a compiled code that waits for execution in the queue. Furthermore, the memory abstraction on VRAM eases to transfer data from and to the RAM of the system. This practical definition gives readers some understanding of the GPU versus CPU hardware. A GPU has many more cores than a CPU, which allows small computation tasks to run on the GPU. Processing of these tasks is much more efficient than on CPU, which is by design optimized for long tasks, threads, and processes in the Operating System. Fig. 1 shows the OpenCL program execution parts [31].

2.3. *Deep Learning Neural Networks*

Practical implementations of artificial neural networks (ANN) exploded ten years ago [2]. ANN, as a machine learning specific tool, has been studied since the middle of the last century. Conceptually, the animal nervous system was an inspiration for creating ANN. The base element of the ANN is a neuron, an element which non-linearly and non-reversible maps multiple inputs to one output. The neuron implements an idea of separation of input data into two independent classes by one sword cut. Even one neuron has useful functionality, and under the name "perceptron", was implemented over fifty years ago [2].

Meanwhile, the idea of a multilayered neuron structure ANNs emerged, in which the output of the previous layer feeds each layer, the number of layers counts in hundreds and thousands, and the number of neurons counts in millions. The improved and rethought ANN is known as a deep learning neural net (DNN) nowadays. Such a neural net is especially suitable for classification problems which might be applied to many different areas, such as a natural language translation system, an automated face recognition system, or an automated driver assistant system, to name only a few.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

For an ANN to be useful, the process of learning trims each neuron's parameters, which is very similar to a multi-parameter optimisation process. The number of trimming parameters counts hundreds of millions for a useful ANN. At the time, the ANN learning required a computer not possible outside of a few supercomputers. However, a new idea emerged. For each sophisticated ANN, there is a three-layer ANN that is functionally identical to the original ANN was mathematically proved. Therefore, scientists limited themselves to training only three-layer ANNs in the hope of finding in each case the Saint Graal - the most straightforward net which can realize functionality.

The initial results were disappointing. Only in a few cases does the training of neural networks lead to solving real problems. The hypothesis at that time was that the trouble laid in the trimming parameters. Mathematically, real numbers make up an infinite set that is dense and continuous. Computers do not easily imitate these abstract mathematical properties. For computers, the trimming parameters are numbers taken from a big yet finite set. Or perhaps the problem was in the learning algorithms used. Nevertheless, just ten years ago, personal computers and workstations were not ready for the fast computing necessary to model the deep learning neural networks.

2.4. Darknet

Darknet is a Convolutional Neural Networks (CNN) engine that uses configured CNN models [18]. The main application of this engine is to classify objects in the video stream, for example, from video files or cameras in real-time or near real-time. It is mainly written in programming language C originally running on GNU/Linux. It can be used on a slow CPU but can be accelerated by NVidia-based GPUs with the CUDA Toolkit and the addition of cuDNN to the Tensors Compute Toolkit if necessary. Compiler flags switch computing modes. The Darknet engine provides unique CNN training techniques and allows us to show it as a real-time object detection demonstration. The main goal of Darknet was to introduce You Only Look Once (YOLO) model [19]. There are already four versions of improved YOLO models. The main goal of the project described in this article was to move CUDA-GPU based code to OpenCL-GPU based code.

Training of some of the models that we are using today was simply reserved for High-Performance Computing (HPC) servers, also often called Computation Grid Clusters (CGC). Nowadays we have workstations with GPUs that are comparable with small HPC or CGC server farms. For example, the authors of this article used a workstation with two NVidia Titan RTX GPUs that together offered more than 10 thousand compute cores and 48 GB of VRAM. That computing power accelerates the training of deep learning neural networks. Moreover, mathematical models are ready to learn the features and differences between Cat or Dog, Car and Bus, and so on.

3. Porting methodology

The Darknet CNN engine can model various kinds of deep learning neural networks and allows, by a simple change of the configuration, the use of DNN or CNN models [18]. These models have excellent performance due to their unique architecture and the introduction of GPU acceleration. Unlike the region proposal classification networks (fast RCNN) or deformable parts models (DPM), input images calculations do not work in a deep pipeline. Object detection and classification are reduced by YOLO neural network to a regression problem. Objects are searched and classified simultaneously throughout the image [19]. Besides, modern CUDA-compliant GPUs accelerate DarkNet models. It is possible to examine the video stream in real-time. Author [19] claims that YOLO's base network runs at 45 fps on the TITAN X graphics card, while the fastest and simplified YOLO "tiny" versions of the network can process at 145 fps. The performance of the Darknet engine has prompted the authors to move the framework to OpenCL to allow its use on all modern GPU hardware and software.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

3.1. GPU-computing challenges

Before explaining the porting methodology, the authors want to list general problems that readers can find in any technical implementation of GPU and CPU computing. Issues that are solved early can reduce implementation time.

3.1.1. Abstraction of VRAM

Video RAM (VRAM) cannot be addressed and accessed directly by the CPU. The GPU cannot manage and access conventional RAM. Data to be processed by the GPU must first be transferred from RAM to VRAM. After completion of the calculation, the transfer of the most critical results from VRAM to RAM occurs. The OpenCL implementation hides details of the transfer mechanism. A helpful "pairs" rule is that every buffer in RAM should be permanently bound to the buffer in VRAM. In this situation, the Darknet project uses "pull" and "push" conventions for VRAM. The "pull" transfers data from the VRAM buffer to the RAM buffer and the "push" transfers from RAM to VRAM.

The goal of the introduction of "cl_mem_ext" abstraction was to formalise the "pair" rule. It turned out that this is the key to success after many different techniques tried by the authors. The "cl_mem_ext" type replaces the "cl_mem" structure. The last one is Video RAM (VRAM) in OpenCL. This structure contains, of course, "cl_mem", but also a pointer to the RAM associated with VRAM space.

Thanks to the "cl_mem_ext" abstraction the code not only gains readability but during data exchange between RAM and VRAM hardware acceleration turns on. It happens on the Intel platform because creating "cl_mem" (VRAM) is associated with direct access I/O map pointers. Given that OpenCL generally runs quite slowly on Intel GPUs, even this slight acceleration of memory exchange is noticeable. The abstraction described here is ubiquitous in code intended for GPU. However, the authors noted that when using CUDA, memory exchange between RAM and VRAM is about ten times faster than in OpenCL in any situation and it seems to be the main weakness of the OpenCL implementation. At least, for the NVidia Titan RTX, tests conducted on the Ubuntu 18.04 GNU/Linux showed it.

After careful analysis of the code, it turned out there are many cases of using the "pair" rule in the CNN DarkNet engine. Although in the CUDA version the rule is repeatedly broken. There are three significant exceptions to this rule on the OpenCL implementation in the training process of YOLO1, YOLO2, YOLO3 models. At the end of the training step, the layer.output_gpu (cl_mem_ext) is rewritten to the net.input (float *) to calculate LOSS[1], AVG IOU[2], AVG CLASS[3] and more factors of the training process because that factors can be computed efficiently only on CPU.

Before the idea of abstraction "cl_mem_ext", there were a few failures in the porting. Since the OpenCL Darknet testing method was a comprehensive CIFAR-10 training, the testing process was time-consuming. It was also an opportunity to introduce many bugs. The abstraction of "cl_mem_ext" helped to solve many of the problems encountered. Having this abstraction can be considered as a best-practice for proper OpenCL abstraction usage.

Listing 1: The cl_mem_ext abstraction

```
typedef struct _cl_mem_ext cl_mem_ext;
typedef struct _cl_mem_ext {
    cl_mem mem;
    cl_mem org;
    size_t len;
```

---

1 YOLO uses sum-squared error SSE for the LOSS function.
2 Intersect Over Union – a measure of rectangles similarity defined as ratio of common area to total field ratio.
3 Class of objects detected.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1


```
    size_t off;
    size_t obs;
    size_t cnt;
    cl_mem_ext (*cln)
      (cl_mem_ext buf);
    cl_mem_ext (*inc)
      (cl_mem_ext buf, int inc, size_t len);
    cl_mem_ext (*dec)
      (cl_mem_ext buf, int dec, size_t len);
    cl_mem_ext (*add)
      (cl_mem_ext buf, int add, size_t len);
    cl_mem_ext (*rem)
      (cl_mem_ext buf, int rem, size_t len);
    void* ptr;
    void* map;
    cl_command_queue que;
} cl_mem_ext;
```

All "cl_mem_ext" abstraction variables are detailed below. We have described each part of the abstraction that has a few more uses not included in this section, but the detailed description helps to read and understand all-purpose of it. The abstraction allows all "_gpu" suffix fields in structure types to possess all valuable information about the VRAM.

- mem - general usage cl_mem means VRAM;
- org - the original copy of cl_mem VRAM to read-only;
- len - length of the VRAM memory buffer;
- off - offset of possible sub-buffer of VRAM;
- obs - object size, most often it is sizeof(cl_float);
- cnt - counter of sub-buffer "jumps" on mem;
- cln - function to the clean function of the sub-buffer state;
- inc - function to use as a "+=" operator;
- dec - function to use as a "-=" operator;
- add - function to use as a "+" operator;
- rem - function to use as a "-" operator;
- ptr - pointer to RAM buffer read-only in all cases;
- map - mapped VRAM buffer to/from RAM transfers cases;
- que - reference to the OpenCL queue for this abstraction.

3.1.2. Prevent GPU-Computing run time errors

Effective debugging of GPU code is very difficult. It concerns the kernels, i.e. functions running on the GPU. There is no memory protection, like in the CPU-Computing run time. GPU code is checked statically during compilation, and in case of an error, the programmer fixes it. But during execution, if a parameter is specified that, for example, cause the code to override the GPU video memory outside the designated buffer, your computer's operating system may crash, you must restart it and fix the issue. Then the over runtime tests begin. The authors of this article used asserts to make sure the provided parameters on critical kernels are correct just before the call. In the case of erroneous value, the assert interrupts CPU-computing on the runtime, and the fixing process starts. In other words, the assert failure on the execution just before the call of kernel function on the GPU save implementation time because the assert checks provided parameters. The time execution cost of the assets is minimal and acceptable. Checking the "len" values is possible thanks to the "cl_mem_ext" abstraction.

Listing 2: CPU run time assert protection example

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1


```
void copy_gpu(
int N,
cl_mem_ext X, int INCX,
cl_mem_ext Y, int INCY) {
    assert(N <= X.len &&
    N <= Y.len &&
    X.len <= Y.len);

    copy_offset_gpu(
    N,
    X, 0, INCX,
    Y, 0, INCY);
}
```

In code from Listing 2, the assertion introduced in the CPU code checks the parameters given in run-time to the GPU code against range overrun in tables passed to the GPU code for calculation. The example is to copy data from one "cl_mem" VRAM buffer to another "cl_mem" VRAM buffer on GPU with acceleration. Parameter N is the number of threads and the size of both buffers at the same time. The 2-dimensional structure type provides it. The best practice is to check as many run-time errors as possible before calling the GPU code.

Listing 3: GPU code that needs protection

```
__kernel void copy_kernel(
int N,
__global float *X, int OFFX, int INCX,
__global float *Y, int OFFY, int INCY)
    int i = (get_group_id(0) +
    get_group_id(1)*get_num_groups(0)) *
    get_local_size(0) + get_local_id(0);
    if(i < N) Y[i*INCY + OFFY]
    = X[i*INCX + OFFX];
}
```

3.1.3. Multi-Threading for GPU-Computing modeling

The data model for multi-threading is the key to success in improving the computing speed on any workstation or multi-GPU server with a modern operating system. Multi-threading issues should be solved by the right data model, which helps to avoid synchronization techniques by correctly separating data. Of course, thread synchronization is necessary to combine the obtained values, but due to the inevitable threads stalling it should be used scarcely. In the solution described in this article, each thread runs in a separate OpenCL queue, which isolates programs and kernels running in the queue, such as the Listing 3 kernel, from other programs and kernels. The most important data model declaration used for multithreading shows Listing 4.

Listing 4: GPU declare separated thread values

```
extern int *gpusg;
extern int ngpusg;
extern __thread int opencl_device_id_t;
extern __thread int opencl_device_ct_t;
//...
cl_context opencl_context;
cl_command_queue* opencl_queues;
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1


```
cl_device_id* opencl_devices;
```

The declarations of variables used in Listing 4 are listed and detailed below. Listing 4 contains a proven formula for multithreading modelling in the programming language C. Whenever calculations take up a lot of time, and the results are combined and collected at the end, the formula is applicable to divide the work into a multithreaded collection of separate tasks. The "__thread" modifier should be set for a specific thread as soon as possible, as the appropriate OpenCL queue identifier is needed. The modifier "__thread" in C is called a "static thread", and each thread has a static copy of this value. In the "opencl_context" declaration, there can only be one value of this type. On the other hand, "opencl_queues" and "opencl_devices" are declared as dynamic run time arrays. The number of tables is equal to the number of physical GPUs used. In our DarkNet port, "-gpus" line parameter specifies this number. For example, the value of this argument "1,2" means using 2 GPUs with indexes 1 and 2 on the OpenCL platform. Each kernel has at its disposal separate array occurrences, i.e., for each chunk of the code executed on the GPU separate data exists. Here are the details of the thread model specification.

- *gpusg – pointer to the global array of the GPU indexes;
- ngpusg – global number or counter of all GPUs to use;
- opencl_device_id_t – GPU device id in particular thread;
- opencl_device_ct_t – GPU devices counter in particular thread;
- opencl_context – global one context of OpenCL platform;
- opencl_queues – global array of OpenCL queues (for multi-GPU);
- opencl_devices – global array of OpenCL devices (GPUs).

This solution has proven to be able to ensure the separation of multi-threading by the data model. It is used not only in the "opencl_set_device" function. The whole multi-threaded process also requires separating array pointers into compiled GPU kernels. But with "opencl_device_device_id_t" it can be easily accessible by indexing the table, for example in the form of "example_kernel_array[opencl_device_id_t]".

Listing 5: GPU set separated thread values

```
void opencl_set_device(int n) {
    opencl_device_ct_t = ngpusg;
    opencl_device_id_t = n;
}
```

In Listing 5, the authors present the method of using GPUs numbered, for example, 1, 2 in the operating system. The same devices are indexed 0, 1 in multi-threaded abstraction. This method should be called up in separate threads as soon as possible to apply it correctly in a multi-GPU computing model.

Figure 2 shows the Activity Monitor on the macOS controlling the computer in a multi-GPU configuration. The system contains 2 Radeon VII GPUs and an Intel UHD Graphics 630. The "darknet" process uses both Radeon GPUs to train the YOLO2 neural network. An example of YOLO2 training logs is shown later in Fig 6.

While working on the DarkNet OpenCL port, a powerful GPU load tuning mechanism was introduced. The idea is to use more than one thread, local thread space and data models for the function being called. For example, the idea is used for fast_mean function. The essence of the solution is the "tuning" parameter, which determines how many GPU threads are assigned to the kernel function.

Listing 6: Fast BLAS "fast_mean" invocation functions example

```
void fast_mean_gpu(
  cl_mem_ext x
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

Process Name	% GPU	GPU Time	PID	User
AMD Radeon VII (Slot 1)	88.0	1:05:38.35	-	-
darknet	88.0	1:05:38.35	3648	piotr
AMD Radeon VII (Slot 2)	85.0	1:05:05.82	-	-
darknet	85.0	1:05:05.82	3648	piotr
Intel® UHD Graphics 630 (Slot 0)	0.0	5:36.63	-	-
WindowServer	0.0	5:36.61	256	_windowserver

Figure 2. Multi-GPU Computing Monitor State Example [20].

```
  int batch, int filters, int spatial,
  cl_mem_ext mean) {

    int tuning = 16;
    dim2 dimGridG1;
    dimGridG1=dim2_create(tuning,filters);

    dim2 dimGridL1;
    dimGridL1 = dim2_create(tuning, 1);

    opencl_kernel_local(
    opencl_fast_mean_kernel
    [opencl_device_id_t],
    dimGridG1, dimGridL1, 14,
    &tuning, sizeof(cl_int),
    NULL, tuning*sizeof(cl_float),
    &filters, sizeof(cl_int),
    &batch, sizeof(cl_int),
    &spatial, sizeof(cl_int),
    &x.mem, sizeof(cl_mem),
    &mean.mem, sizeof(cl_mem));
}
```

It works very well and improves execution in the most nested loop by a number of tuning value that is computed dynamically be dividing the "filters" variable by "4". The last important information is that parameter "t" cannot be correctly checked in conditions or printed out. Listing 7 shows one of the core function used to calculate the average values from data collected in a three-dimensional "x" array (tensor) on GPU. The calculated values are returned in a one-dimensional "mean" array. The function takes the value of variable "i", which is the index of the output array "mean". Many calls of the function with different values of the "i" parameter that changed from 0 to "filters"-1, fill in the table "mean" with the calculated values. Each call is a separate, concurrent thread for the GPU, and the "mean" array is not necessarily filled in order. Each call is executed for a relatively long time because the variable "spatial" is at least 10k. Issued threads do not involve the entire GPU, because the "filters" variable usually does not exceed 128 and is less than the number of GPU cores. And that has to be optimized in following way. In the most internal loop, the "k" index is increased by the value of the total variable "tuning". A value of the "tuning" variable greater than 1 causes the number of function calls must be the product of the "tuning" and "filters" variables for the calculations to work correctly (the variable "t" changes from 0 to "tuning" -1 for all values of "i"). The values are collected in the array "sums", which is then aggregated with the condition "t == 0" for all variables 0 to "filters"-1 in the "mean" output array. Even with relatively small values for the "tuning" variable, all GPU cores participate in the calculation. The GPU function code is completely free of any synchronization or

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

atomic operations. This optimization is based on a data model, not a synchronization model. In other words, it is an optimization method where the same code is called in separate threads as many times as the value of the "tuning" variable is and can be considered as the OpenCL optimization good-practice.

Listing 7: Fast BLAS "fast_mean_kernel" invocation functions example

```
__kernel void fast_mean_kernel(
int tuning, __local float *sums,
int filters, int batch, int spatial,
__global float *x,
__global float *mean) {
    int t = get_global_id(0);
    if (t >= tuning) return;

    int i = get_global_id(1);
    if (i >= filters) return;

    sums[t] = 0;

    int j,k,s;

    for (j = 0; j < batch; ++j) {
        for (
        k = t;
        k < spatial;
        k += tuning) {
            int index =
            j * filters * spatial +
            i * spatial +
            k;
            sums[t] += x[index];
        }
    }

    barrier(CLK_GLOBAL_MEM_FENCE);

    if (t == tuning-1) {
        mean[i] = 0;
        for(s = 0; s < tuning; ++s) {
            mean[i] += sums[s];
        }
        mean[i] /= (spatial * batch);
    }
}
```

3.1.4. Other Challenges

The BENCHMARK compilation flag enables the collection of the full log of one calculation step. The flag dramatically simplifies the performance testing of the solution. Built-in Jet-Brains CLion debugger helps to fix calculation issues, which allows you to dig into the source of errors quickly. Tracking changes and comparing them with the original code was done using Scooter Software Beyond

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

Compare. Especially helpful was the rule-based comparison used in this tool. The select switch enables/disables the GPU in the runtime. An abandoned parameter, scheduled in the original DarkNet code for the same purpose, has been fixed. It is now a switch called "-nogpu" enabled back. Some of the computation was checked by the sandbox project in a separated environment [52]. DarkNet OpenCL port was tested on AMD Radeon VII, NVidia Titan RTX, Intel Iris 655, Mali GPU on the macOS, or GNU/Linx depending on the computer device capability.

3.2. Porting Darknet to the OpenCL

This sub-section provides the readers with information on the practical method that was done to port the Darknet engine from a CUDA-based to an OpenCL-based solution. In the first part, the authors removed the entire CUDA code from the GitHub fork repository. The shortened code could not even be compiled. All methods with "cuda" prefix have been renamed so that they start with the prefix "opencl", to satisfy the compiler. There are also empty functions in the opencl.c file with opencl.h header files to make compilation possible. At that point, the project had a syntactically correct C-code, but it didn't work. So, we needed to create a set of methods with similar signatures to create memory buffers, load the data from RAM to VRAM, etc., and replace the CUDA code with OpenCL code. Also on the replace stage, all "_gpu" suffix fields that were "float*" or "int*," for example in layer type, were replaced with the "cl_mem_ext" – the unique abstraction created for the "cl_mem".

The question is, why to introduce the "cl_mem_ext" type? That abstraction is needed to store a few things, the most important of which was a pointer to the RAM registry and the "cl_mem" VRAM registry as a pair – as described earlier. In the OpenCL code, each "cl_mem" must have its own "float*" or "int*" pointer. The rule is a subtle aspect of an OpenCL memory transfer technique. By following it, the authors make sure that most data transfers between RAM and VRAM only take place between statically paired buffers. There are only three exceptions. Data transfer in violation of this rule must go on LOCAL (in YOLO1 model), REGION (in YOLO2 model), and YOLO (in YOLO3 model) layers. These are the two main reasons for keeping memory registers as pairs. The first of these is the efficiency of the calculations. Faster data transfer is possible within an Intel OpenCL implementation. The second aspect is to make sure that the C code is consistent. The latter reason has been omitted many times in the original Darknet code because, in the case of buffers in CUDA code, buffer pairing has no impact on performance. With OpenCL code, this is not the case, and it was corrected.

That abstraction made milestones guaranteed that all elements of the code were ready to be improved and filled with the correct implementations. Some of the OpenCL code was taken from a few other computationally-ready applications on GitHub, for example from the [32] fork. The rest were created from scratch by the authors. We know that there are automatic translators from CUDA to OpenCL, such as CU2CL [33], and some other projects on GitHub use that type of tools, but checking the code created with the automatic translator is tedious. Sometimes the code does not work anyway. The authors of this article decided to use the Basic Linear Algebra Systems (BLAS) to do all work manually and create all kernels code from scratch or from CPU implementations to make sure all code will work.

Manually doing all the work was a significant milestone that ensured both computing and training worked for the CIFAR-10 model. This model was chosen as the fastest option to test the engine, and we used it to check CNN training processes on the CIFAR-10 images set. Then all code was carefully and slowly reviewed to provide aspects that were missing or no longer working from the original code. To facilitate this task, we have added a "-nogpu" switch that allows you to test all calculations on your CPU without recompilation. The first reason is to ensure that each time the C code compilation creates 100% the same binary code. The second reason is that it is better to have a switch to test and compare performance, rather than recompile the entire engine code every time. The authors also added new controls for the compilation like "BENCHMARK" and "LOSS_ONLY" for testing the performance of the solution and looking for some weak points and bottlenecks. Thanks to these tests, OpenCL methods for buffer allocation and the copying memory data between VRAM and RAM were improved. Thanks

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

to redesigned multi-threading models, CNN models can be trained in multi-GPU systems. For now, it is possible to train on multiple GPUs only on macOS. The reason is the author's patched clBLAS library, which initially had a non-thread secure implementation. All the above steps make the OpenCL version consistent and accurate. The code was then ready for GPU testing, such as AMD-based GPUs, and showed code-specific performance potential.

We have achieved support for Multi-GPU systems in training through the implementation of Trivial General Matrix Multiply (GEMM). It is not performance-optimized, but mathematically perfectly fine and multi-threading ready. This implementation is only a slightly tuned multiplication of the matrix; however, it clearly shows that clBLAS [23] and CLBlast [24] solutions that are fast and very well optimized are not ready for multi-threading computation yet. We believe that libraries will support multi-threading computation soon. The authors created an enhancement for clBLAS [23] and made it available on the Internet [39], making original GEMM for clBLAS [23] thread-safe. For the time being, the solution works only on macOS and Unix BSD, because macOS retains a unique BSD thread model, which makes the task easier. Threads in macOS are different from POSIX in GNU/Linux, where threads are shared memory processes, so it is a kind of imitation, not a full implementation of the thread model. But Multi-GPU techniques, even with our version of GEMM, are beneficial for CIFAR-10 class solutions to train the network on multi-GPU, which are separate PCIe cards without any hardware connection bridge.

The last to mention improvement is a permutation of the input image set for training propose. Each time the image set is loaded from storage to the RAM, and push to VRAM, we want to make sure that this set contains unique images.

4. Testing Environment: Hardware and Software

The tests we have done on the relatively modern workstation that consists of Asus Rampage V Edition 10 motherboard, Intel i7-5960X 8-core, 16-threads CPU, 64 GB of DDR4 2400 MHz RAM. The authors decided to use the latest 2019 GPUs. The first tests we run on 2 NVidia Titan RTX 24 GB DDR6 VRM GPUs in NVlink bridge configuration and 2 AMD Radeon VII 16GB HBM2 VRAM. That configuration allowed the authors to test not only single but also Multi-GPU computing scenarios. The CUDA solution supports Multi-GPU; on the OpenCL solution, the authors improved the original clBLAS library to support Multi-GPU on AMD cards as well correctly, but only on macOS [39]. For the presented research, the authors decided to use the latest GPUs available on the market, to step ahead, and predict that shortly the same power will be available for automotive, manufacturing, virtual reality, machine learning, and financial industries on the industrial devices. This increase in power was the decision behind the choice of environment.

All development was done on the Apple MacBook Pro 13-inch and tested with Intel Iris Plus Graphics 655 and Sonnet eGPU AMD Radeon 570 chip-based. The last GPU and CPU ready computer used by the authors for tests is the Asus Tinker Board S with Mali GPU-T760 and ARM Rockchip Quad-Core RK3288 CPU. This set of hardware allows authors to cover as many types of GPU equipment as can be obtained on the market. Most of the tests, especially on Intel and ARM architecture, was not presented in this article. Still, we have done to make sure that all platforms support OpenCL-based implementation of the Darknet engine. From the Operating System perspective, the authors used GNU/Linux and macOS. We want to point out that Darknet on OpenCL is ready to use on powerful workstations, including the Mac Pro 2019, and is the only option for this specific device.

Furthermore, the CUDA version of Darknet cannot work on the Mac Pro 2019 once it uses only AMD GPUs. There is another aspect of the multi-platform solution proposed by the authors. Darknet on OpenCL is applicable to recognise objects in photos using any Intel Iris GPU, which is available in almost every notebook with an Intel processor. The Intel GPU is slower than AMD or NVidia processors but requires no additional investment, and all classification is entirely hardware accelerated.

Last but not least, are make definition files (Make and CMake tools) for compilation on different platforms. The compilation switches implemented are NVidia, AMD, and ARM. They allow

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

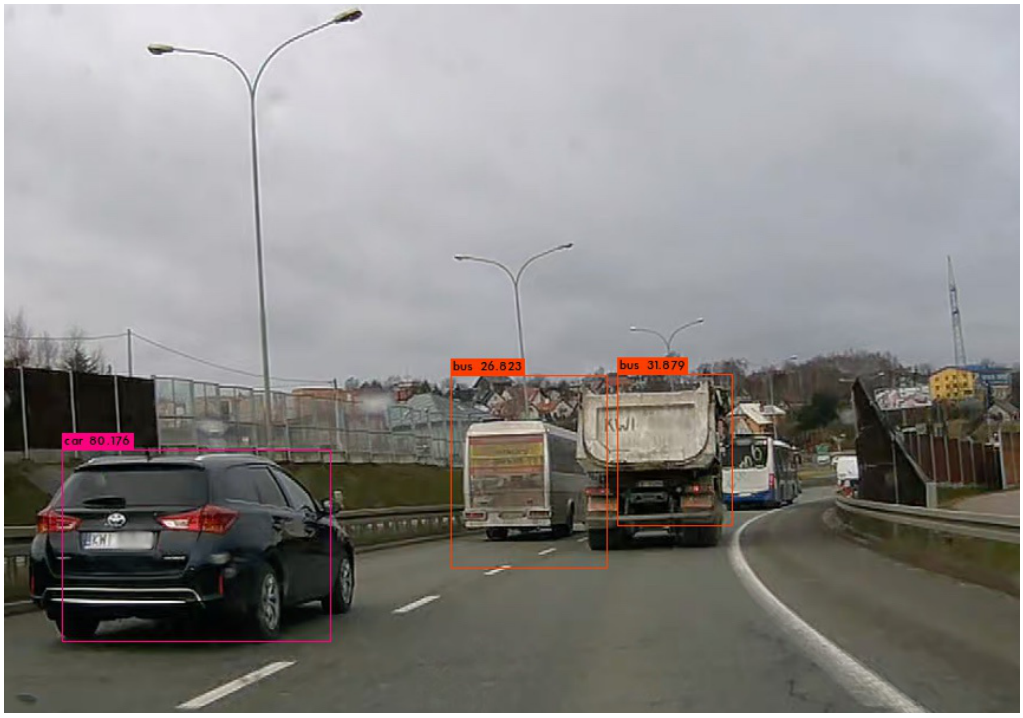

Figure 3. YOLO1 on Darknet on OpenCL Detection [20].

GNU/Linux to choose a supported platform and quickly perform not only tests but also benchmarks or simple training-based exercises on the lightweight CIFAR-10 model. The NVidia switch enables OpenCL on the CUDA Toolkit. The AMD switch turns on OpenCL from the AMD GPU. The ARM switch not only allows proper use of OpenCL for ARM on a single board computer but also allows you to use a Trivial GEMM implementation, only for testing. ARM support we have tested on a single board computer that natively supports OpenCL on the Mali T760 GPU. The authors also tried to use OpenCL on a DSP with Beagleboard AI and X15 computers to make sure OpenCL works with detection (YOLO2) and training (CIFAR-10), but the tests passed only on the Mali-T760 GPU.

Darknet on OpenCL, we tested on four models: CIFAR-10, YOLO1, YOLO2, and YOLO3. All of the models are deep. CIFAR-10, VOC [25], and COCO [26] are benchmarks for training and can be used for validation of any classification algorithms.

5. Results of Training and Computation

As one could foresee, CUDA on NVidia hardware runs faster than the OpenCL version of the Darknet CNN engine [27]. However, the latter version of the technology allows running on any OpenCL compatible device to accelerate computing. Even FPGA may be considered as the fastest computation devices right now, especially when the authors compare the power consumption of the FPGA [28] cards.

5.1. Comparison of results on CNN Models

The Darknet CNN engine offers several models to compare. In this section, the authors focus mostly on the YOLO2 (Fig. 4) and YOLO3 (Fig. 5) classifiers, with some usage of deprecated one named YOLO1 (Fig. 3). There are measurements taken for the detection of images, the detection timings, and the computation benchmarks for the CUDA and OpenCL versions. The best way to compare models is by using one of the pictures from the source code repository. All comparisons following we computed on the OpenCL engine version. The associated figures show the confidence score of each object detected in the picture. Each CNN model used (YOLO1, YOLO2, and YOLO3) detects practically the same items in the picture, but with a rising confidence score.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

Figure 4. YOLO2 on Darknet on OpenCL Detection [20].

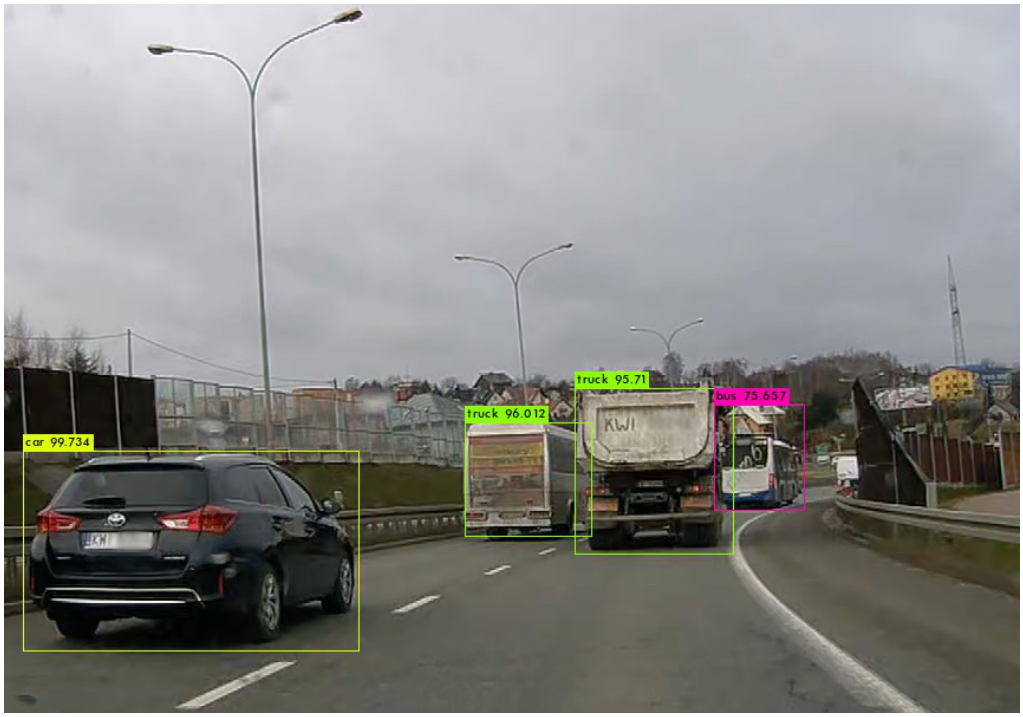

Figure 5. YOLO3 on Darknet on OpenCL Detection [20].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1


```
Loaded: 0.000030 seconds
Region Avg IOU: 0.686208, Class: 0.997445, Obj: 0.633591, No Obj: 0.012894, Avg Recall: 0.787879,  count: 33
Region Avg IOU: 0.833054, Class: 0.999552, Obj: 0.774208, No Obj: 0.009854, Avg Recall: 1.000000,  count: 16
Region Avg IOU: 0.838685, Class: 0.999665, Obj: 0.862343, No Obj: 0.010513, Avg Recall: 1.000000,  count: 15
Region Avg IOU: 0.722484, Class: 0.998001, Obj: 0.643654, No Obj: 0.009041, Avg Recall: 0.850000,  count: 20
Region Avg IOU: 0.606577, Class: 0.951389, Obj: 0.529169, No Obj: 0.010212, Avg Recall: 0.625000,  count: 24
Region Avg IOU: 0.750709, Class: 0.962238, Obj: 0.690864, No Obj: 0.011772, Avg Recall: 0.920000,  count: 25
Region Avg IOU: 0.767018, Class: 0.988973, Obj: 0.724122, No Obj: 0.010307, Avg Recall: 0.952381,  count: 21
Region Avg IOU: 0.826483, Class: 0.994594, Obj: 0.814103, No Obj: 0.011030, Avg Recall: 0.944444,  count: 18
Region Avg IOU: 0.841051, Class: 0.999232, Obj: 0.777486, No Obj: 0.012963, Avg Recall: 1.000000,  count: 23
Region Avg IOU: 0.778936, Class: 0.962295, Obj: 0.733225, No Obj: 0.011014, Avg Recall: 0.958333,  count: 24
Region Avg IOU: 0.837959, Class: 0.998432, Obj: 0.785130, No Obj: 0.008752, Avg Recall: 1.000000,  count: 16
Region Avg IOU: 0.827402, Class: 0.995851, Obj: 0.845254, No Obj: 0.007986, Avg Recall: 1.000000,  count: 11
Region Avg IOU: 0.845528, Class: 0.998950, Obj: 0.753365, No Obj: 0.008044, Avg Recall: 1.000000,  count: 12
Region Avg IOU: 0.764970, Class: 0.983667, Obj: 0.700350, No Obj: 0.011194, Avg Recall: 0.913043,  count: 23
Region Avg IOU: 0.827505, Class: 0.998928, Obj: 0.817443, No Obj: 0.010181, Avg Recall: 1.000000,  count: 14
Region Avg IOU: 0.800513, Class: 0.996055, Obj: 0.810604, No Obj: 0.010844, Avg Recall: 1.000000,  count: 18
Syncing... Done!
40368: 2.511185, 2.535058 avg, 0.000200 rate, 7.922064 seconds, 5167104 images
```

Figure 6. YOLO2 Training multi-GPU Process Step Example [20].

STEP CIFAR-10	macOS i7-8559U			GNU/Linux	
LAYER	CPU	OpenCL CPU	OpenCL Iris 655	OpenCL ARM CPU	OpenCL ARM Mali-T760
FW CONVOLUTIONAL	112971	17360	24201	1017011	56774
FW MAXPOOL	19181	123	6	84308	72
FW CONVOLUTIONAL	16420	8779	17081	181904	9372
FW CONVOLUTIONAL	13681	9019	519	128280	9278
FW CONVOLUTIONAL	160533	14448	3162	2370882	17605
FW MAXPOOL	8894	108	5	43811	139
FW CONVOLUTIONAL	6483	9898	1183	77127	9645
FW CONVOLUTIONAL	50109	22096	1422	640782	18777
FW CONVOLUTIONAL	13405	6166	586	156009	10955
FW CONVOLUTIONAL	940	6872	711	10511	10313
FW AVGPOOL	35	47	30	205	81
FW SOFTMAX	36	5147455	44760	223	315204
BW SOFTMAX	1	13	67	6	153
BW AVGPOOL	13	12	13	667	341
BW CONVOLUTIONAL	3208	17844	22998	22030	27390

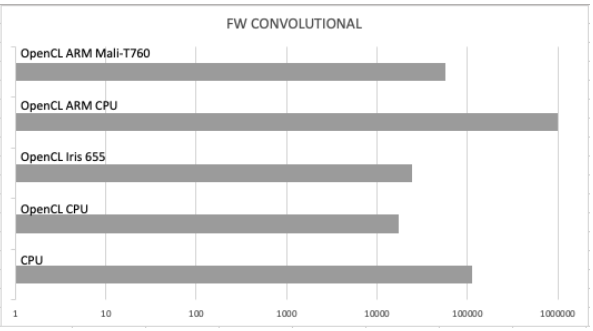

Figure 7. CIFAR-10 on Darknet Timings Layer Test (Slower GPUs) [20].

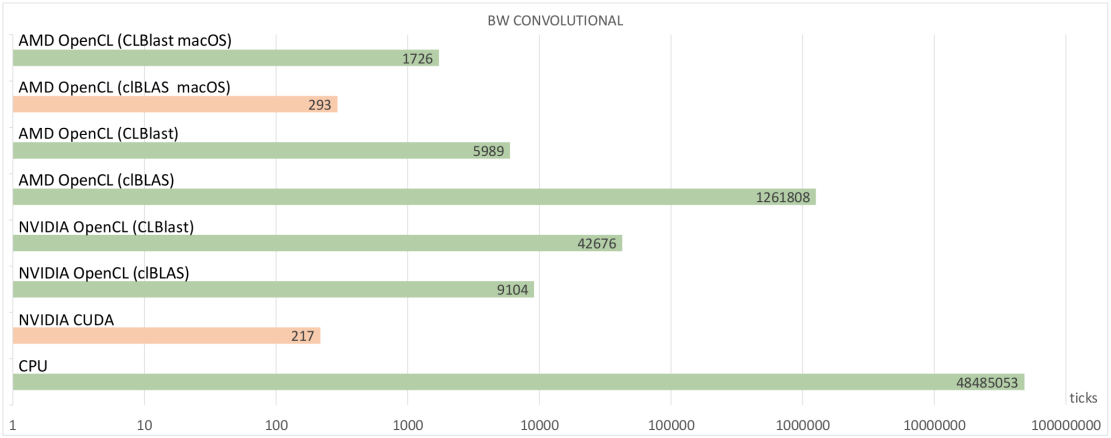

Figure 8. YOLO2 on Darknet Timings Layer Test (Faster GPUs) [18,20].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

STEPS			NVIDIA on GNU/Linux			AMD on GNU/Linux		AMD on macOS	
CUDA	OpenCL (clBLAS)	OpenCL (CLBlast)	CUDA	OpenCL (clBLAS)	OpenCL (CLBlast)	OpenCL (clBLAS)	OpenCL (CLBlast)	OpenCL (clBLAS)	OpenCL (CLBlast)
A	B	C	D	E	F	G	H	I	J
gradient_array_kernel	gradient_array_kernel	gradient_array_kernel	3	5	6	6	3	1	1
backward_bias_kernel	backward_bias_kernel	backward_bias_kernel	3	6	5	6	3	1	1
backward_scale_kernel	backward_scale_kernel	backward_scale_kernel	3	104018	113510	2	3	1	1
scale_bias_kernel	scale_bias_kernel	scale_bias_kernel	3	6	6	2	3	1	1
fast_mean_delta_kernel	fast_mean_delta_kernel	fast_mean_delta_kernel	3	2810	2933	2	4	1	2
fast_variance_delta_kernel	fast_variance_delta_kernel	fast_variance_delta_kernel	3	98	61232	8	3	1	1
normalize_delta_kernel	normalize_delta_kernel	normalize_delta_kernel	3	8	8	2	3	1	0
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	2	6	6	8	3	1	1
cublasSgemm	clblasSgemm	CLBlastSgemm	11	117556	34	200000	64	5	27
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	3	6	6	3	3	2	1
cublasSgemm	clblasSgemm	CLBlastSgemm	8	10	28	7	46	3	24
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	3	5	6	1	3	0	1
cublasSgemm	clblasSgemm	CLBlastSgemm	8	8	26	1	43	2	25
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	3	5	5	1	3	0	2
cublasSgemm	clblasSgemm	CLBlastSgemm	8	7	27	2	41	3	22
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	3	4	6	1	3	0	1
cublasSgemm	clblasSgemm	CLBlastSgemm	7	6	26	2	41	2	23
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	4	4	5	1	3	0	0
cublasSgemm	clblasSgemm	CLBlastSgemm	8	7	26	2	41	2	22
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	3	4	5	1	2	1	1
cublasSgemm	clblasSgemm	CLBlastSgemm	8	6	26	2	41	2	22
im2col_gpu_kernel	im2col_gpu_kernel	im2col_gpu_kernel	3	5	5	1	2	0	1
cublasSgemm	clblasSgemm	CLBlastSgemm	8	7	25	2	41	2	24
BW CONVOLUTIONAL	BW CONVOLUTIONAL	BW CONVOLUTIONAL	132	322780	178005	200105	488	76	261

Figure 9. YOLO2 on Darknet Timings Kernels Test [18,20].

5.2. Comparison of results on CUDA and OpenCL

In Fig. 7 authors compared on CIFAR-10 example, one step of YOLO neural net training on slower GPUs, once in Fig. 8 on faster GPUs. In Fig. 8 we first compare the same backpropagation calculation of the convolutional layer. The X-axis is logarithmic to show better the differences in timing as the number of "ticks" returned by time() C function. All measurements we did on the same workstation; the only difference was in the used GPUs. For the CUDA version, we used 2x NVidia Titan RTX. For the OpenCL version, we used 2x XFX AMD Radeon VII GPUs with the latest drivers and computation libraries on Ubuntu 18.04 GNU/Linux and macOS. As Fig. 8 shows, CPU-only computation is about $2 \cdot 10^5$ times slower than the original CUDA version. OpenCL implementations are several dozen to a thousand times faster than the CPU version. It depends on the hardware used and implementation details. One part of the OpenCL version is the multiplication of matrices, and clBLAS and CLBlast were measured. OpenCL with CLBlast for training is faster but does not work in all cases; for example, the classification may fail when one use CLBlast. This failure is why clBLAS is the default library used in the provided solution. The authors believe that in the future, both clBLAS and CLBLast libraries will evolve and Darknet in OpenCL will choose the best one. For now, in the source code of this version, the reader will find in the "patches" folder the "clblast.patch" file, which is ready to apply to replace clBLAS with CLBLast. However, the most surprising result we achieved with OpenCL on macOS. The training time for the first convolutional layer on an AMD GPU is only slightly longer than the training time for the same layer on NVIDIA CUDA – see Fig. 8. 60 compute units/3840 stream processors of AMD GPU hardly compares with NVIDIA GPU's 576 Tensor Cores. It seems that both chips are not far from each other in terms of performance and that careful implementation of OpenCL does not have to perform worse than proprietary CUDA technology.

Power of the OpenCL is even more evident in Fig. 9 where authors provided a detailed performance test of all CUDA and OpenCL "kernels" that we used in this GPU comparison. The test is about the first convolutional layer in YOLO2, whose training is the most complicated issue. The test technique uses "tics" returned by the function time() in the programming language C. Instrumentally,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

	ganyc717	Kylin-PHYTIUM	sowson	Comments
CIFAR-10 Training	YES	ERROR	YES	ganyc717 (16m) sowson (10m)
CIFAR-10 Compute	YES	ERROR	YES	
YOLO1 Test (dog)	NO	ERROR	YES	
YOLO2 Test (dog)	YES	ERROR	YES	
YOLO3 Test (dog)	YES	ERROR	YES	
YOLO1 Training	NO	ERROR	NO	
YOLO2 Training	YES	ERROR	YES	ganyc717 ~50% slower than sowson
YOLO3 Training	YES	ERROR	YES	ganyc717 ~50% slower than sowson
YOLO2 Demo (1080p@60FPS MP4)	YES	ERROR	YES	ganyc717 (20 FPS) sowson (24 FPS)
Same Structure as CUDA version	NO	NO	YES	

Figure 10. Darknet on OpenCL Comparision: "ganyc717" [34], "Kylin-PHYTIUM" [35] and our "sowson" [20].

calls of a particular method are surrounded by measurements of the number of "ticks" taken and the results are summed up for each method separately. That benchmark solution helped to not only compare solutions but also quickly identified the bottlenecks in the OpenCL-based Darknet during the implementation. Each time measurement setup was assuming the worst-case scenario in the first layer matrix sized 608 x 608 (width x height). The "BENCHMARK" compilation flag provides this option and creates a very detailed output log for only one worst-case step. So the calculations repeat without a start point hazard. The part of the mentioned log is in Fig. 9 for a detailed comparison. Additional time overhead is a difference between the value in the last row and a sum of all rows above. Each column is about a particular log scenario. Whenever the macOS/OpenCL kernel performed better than the CUDA counterpart, we use the cell green background. As you can see the use of the clBLAS library in OpenCL implementation guaranteed shorter backpropagation computation time than CUDA.

Another comparison is in Fig. 10, where the authors compared their solution [20] with another two implementations from GitHub [34,35]. The solution described here is about 50% faster on the training and 20% faster on the detection of a movie file at 1080p@60. Also, the presented solution is the only one that supports a deprecated YOLO1 model.

The plot in Fig. 11 is the Loss in Time for 10k steps of the YOLO3 training process on the original CUDA version of the Darknet engine. Axis Y is logarithmic. The plot in Fig. 12 is the Loss in Time for 10k steps of the YOLO3 training process on the OpenCL version of the Darknet engine. Axis Y is logarithmic. When we are trying to compare the plots, we see that shape is almost the same, values on Y-axis also. The only difference is the time of computation that more than seven times longer for OpenCL with a clBLAS version on the GNU/Linux. Note that profiling CUDA and OpenCL engine on the same hardware has shown that such a vast difference in training time is only a consequence of the difference in data transfer speed. The CUDA implementation transfers data between RAM and VRAM even ten times faster than OpenCL. This deficiency can be alleviated in the future and probably is a software-only issue.

5.3. Systems' hardware independency

The authors believe that creating the OpenCL port allows the use of new hardware, such as AMD-based hardware, that has recently become strong and fast. We think that researchers in the neural networks field may consider the use of this hardware, but only if the software for both training and computation is ready. So, what gaps can the software introduced via GPUs is resolved thanks to the OpenCL version.

The authors believe that the domination of one technology, CUDA, provided by NVidia, is not sustainable in the long run, and OpenCL-based software can contribute to the more sustainable development of the technology. Taking into account the applications of the OpenCL-based engine developed by the authors for neural network-based calculations, there are potentially many more applications for OpenCL-based software than for the CUDA engine.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

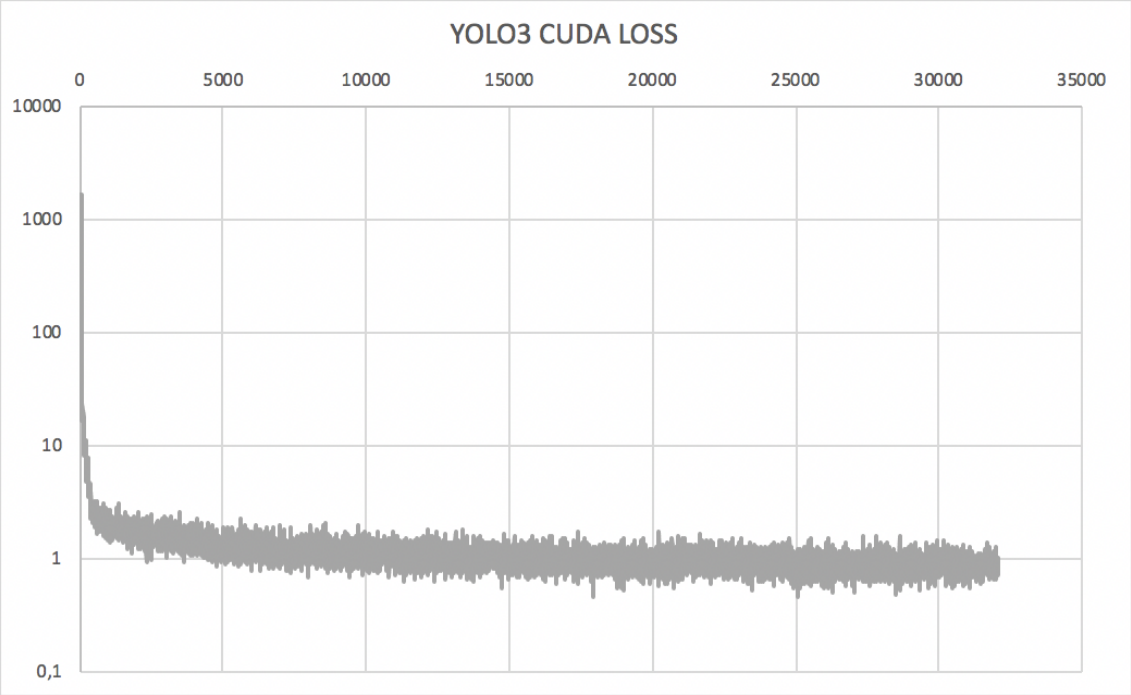

Figure 11. YOLO3 on Darknet on CUDA Training Loss Plot [18].

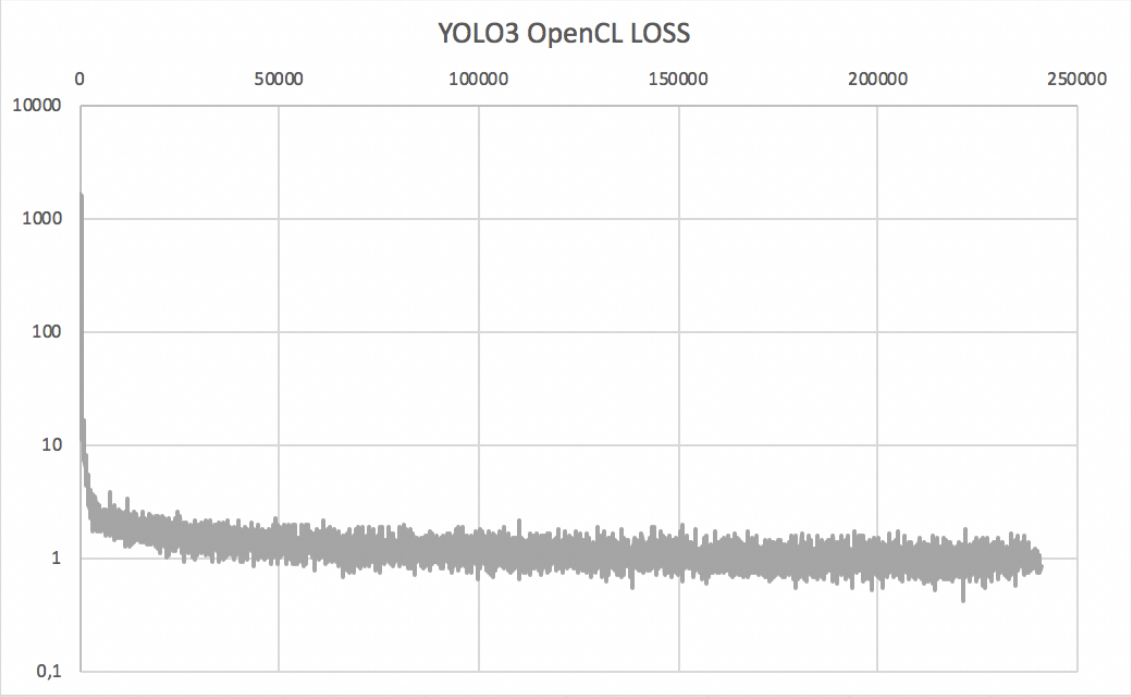

Figure 12. YOLO3 on Darknet on OpenCL Training Loss Plot [20].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 July 2020 doi:10.20944/preprints202007.0506.v1

5.4. Example of the Application

The Darknet CNN engine allows almost any kind of application. Basically, in most cases and models, it is used as a classifier. It is used mostly on images, thanks to the fact that deep learning neural networks can learn features and classify objects with very high accuracy. The picture is a matrix of correlated pixels. The CNN model can use data from many sensors collected as an input image for pattern recognition and alarm detection. Today virtually all smartphones have been equipped with the Mali GPU graphics chip (or similar) and can be used to classify data from onboard sensors of many types thanks to the fact that Mali GPU supports OpenCL 1.2 [40]. This type of recognition and detection can be potentially fault-tolerant because even if in such a system some data are flawed or not available, the entire pattern will still be detected and classified correctly [47]. In the same way in the automotive industry mass-produced cheap GPU type acceleration chip can make decisions essential for cybersecurity of onboard electronic systems [41].

Other possible applications may use the Intel® HD Graphics engine integrated within Intel® Core™ microarchitecture [49]. A vast base of industrial-grade PC is implemented in production lines, quality control, charging fees [48], etc. that fully supports OpenCL acceleration, and can accommodate a new DNN/OpenCL based applications. For instance, when a recognition system detects a car in front of the door, an AI system may be started. Not only the car plate is scanned, but the whole silhouette is recognized, classified, and an AI system can then open an access door and allow the car to drive into advisable place in the garage managed by forecasting subsystem [50].

The use of this type of technology in industrial plant lines could allow users to make decisions about the condition of their equipment with the help of vibration sensors. This equipment and these sensors can enable the early detection of a machine that fails. The user can order and replace the device just before it breaks down [43]. The value of such an early failure warning is very high, as any downtime in a factory is a huge loss, and it is prohibitively costly to have all replacement in the storage. Additionally, there can be many similar consumer-grade applications, such as adjusting salinity in aquariums, lighting control, intelligent home heating, water heating control, and much more [44–46].

5.5. Implications for Modern Engineers

Modern engineers can find significant value with the Darknet on OpenCL. Training and recognition are possible without recompilation for the implementation. It is also compatible with the CUDA version so that trained models can utilise CUDA or OpenCL optionally. In total, the OpenCL port contains 146 changed files, 20901 added lines, and 6656 removed lines of C code. The structure of both projects is the same, which means that once the CUDA version evolves, all new features will be easily portable to the OpenCL version.

5.6. Unique Capability of Darknet on OpenCL

As this article shows, the OpenCL version has a few additional built-in capabilities. One of them is the "BENCHMARK" and "LOSS_ONLY" compilation flag that allows the measurement of each GPU method computation and each layer computation time. These computation times allow users to look for the bottlenecks and also enables fine-tuning of the code. The OpenCL version also has an option to disable GPU acceleration and run on CPU only without re-compilation of the engine. And last but not least, this version is fully compatible with the CUDA version and allows users to compute and train the CNN model on all modern GPUs manufactured by the AMD, NVidia, Intel, Mali, and more vendors.

6. Conclusion

The port of the Darknet engine on OpenCL was not trivial. Many aspects and code changes we performed for this study. The accomplishment of this project took a lot of attention. Thanks to this port, Darknet may be used on macOS and GNU/Linux on OpenCL 1.2+ ready hardware,

which could bring to the entire AI Open Source community a great value. The OpenCL version is not slower than the CUDA-based version. The authors believe that very soon, thanks to improved OpenCL implementations, and improved matrix multiplication (GEMM) capability of clBLAS or CLBLast projects, the OpenCL version will be achieving similar or better performance than CUDA one, especially on the macOS.

Author Contributions: Conceptualization, P. Sowa, J. Izydorczyk; methodology, P. Sowa; software, P. Sowa; writing–original draft preparation, P. Sowa and J. Izydorczyk; writing–review and editing, P. Sowa and J. Izydorczyk; supervision, J.Izydorczyk.

Funding: This research was funded by the Polish Ministry of Science and Higher Education funding for Ph.D. studies under grant no DWD/3/33/2019.

Acknowledgments: The authors would like to thank Joseph Redmon for a great public domain open source Darknet CUDA-based project on GitHub [18].
Piotr Sowa would like to thank Professor Marian B. Gorzałczany for inspirations and shared his passion for the AI / ML subjects as Piotr Sowa's master thesis mentor.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	artifical inteligence
ANN	artificial neural network
CGC	computation grid cluster
CIFAR-10	Canadian Institute for Advanced Research dataset
CNN	convolutional neural network
COCO	common objects in context,
CPU	central processing unit
CUDA	compute unified device architecture
Darknet	open-source neural network framework
DNN	deep lerning neural network
GPU	graphics processing unit
HPC	high-performance computing
IOU	intersect over union
ML	machine learning
OpenCL	open computing language
RAM	random access memory
YOLO	you only look once
VRAM	video memory
VOC	visual object classes

References

1. LeCun, Y.; Bengio, Y. and Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444.
2. Sze, V.; Chen, Y.; Yang, T. and Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE* **2017**, *105*, 2295–2329, doi: 10.1109/JPROC.2017.2761740.
3. Liu, F.; Tang, G.; Li, Y.; Cai, Z.; Zhang, X. and Zhou, T. A Survey on Edge Computing Systems and Tools. *Proceedings of the IEEE* **2019**, *107*, 1537–1562, doi: 10.1109/JPROC.2019.2920341.
4. Abdallah, M.S.; Kim, H.; Ragab, M.E.; Hemayed, E.E. Zero-Shot Deep Learning for Media Mining: Person Spotting and Face Clustering in Video Big Data. *Electronics* **2019**, *8*, 1394.
5. Audebert, N.; Le Saux B. and Lefevre, S. Deep Learning for Classification of Hyperspectral Data: A Comparative Review. *IEEE Geoscience and Remote Sensing Magazine* **2019**, **7**, 159–173, doi: 10.1109/MGRS.2019.2912563.
6. Alexeev, A.; Kukharev, G.; Matveev, Y.; Matveev, A. A Highly Efficient Neural Network Solution for Automated Detection of Pointer Meters with Different Analog Scales Operating in Different Conditions. *Mathematics* **2020**, *8*, 1104, doi: 10.3390/math8071104.

7. Guo, P.; Xue, Z.; Mtema, Z.; Yeates, K.; Ginsburg, O.; Demarco, M.; Long, L.R.; Schiffman, M.; Antani, S. Ensemble Deep Learning for Cervix Image Selection toward Improving Reliability in Automated Cervical Precancer Screening. *Diagnostics* **2020**, *10*, 451, doi: 10.3390/diagnostics10070451.
8. Veeranampalayam Sivakumar, A.N.V.; Li, J.; Scott, S.; Psota, E.; J. Jhala, A.; Luck, J.D.; Shi, Y. Comparison of Object Detection and Patch-Based Classification Deep Learning Models on Mid- to Late-Season Weed Detection in UAV Imagery. *Remote Sens.* **2020**, *12*, 2136, doi: 10.3390/rs12132136.
9. Pham, H.H.; Salmane, H.; Khoudour, L.; Crouzil, A.; Zegers, P.; Velastin, S.A. Spatio–Temporal Image Representation of 3D Skeletal Movements for View-Invariant Action Recognition with Deep Convolutional Neural Networks. *Sensors* **2019**, *19*, 1932, doi: 10.3390/s19081932.
10. Wang, J.; Zhang, L.; Wang, C.; Ma, X.; Gao Q. and Lin, B. Device-Free Human Gesture Recognition With Generative Adversarial Networks. *IEEE Internet of Things Journal* **2020**, doi: 10.1109/JIOT.2020.2988291.
11. Jayasundara, V.; Jayasekara, H.; Samarasinghe T. and Hemachandra, K. T. Device-Free User Authentication, Activity Classification and Tracking using Passive Wi-Fi Sensing: A Deep Learning Based Approach. *IEEE Sensors Journal* **2020**, doi: 10.1109/JSEN.2020.2987386.
12. Di Domenico, S.; De Sanctis, M.; Cianca, E.; Giuliano F. and Bianchi, G. Exploring Training Options for RF Sensing Using CSI. *IEEE Communications Magazine* **2018**, *56*, 116–123, doi: 10.1109/MCOM.2018.1700145.
13. Wang, J.; Gao, Q.; Pan M. and Fang, Y. Device-Free Wireless Sensing: Challenges, Opportunities, and Applications. *IEEE Network* **2018**, *32*, 132–137, doi: 10.1109/MNET.2017.1700133.
14. Xu S. and Tian, Y. Device-Free Motion Detection via On-the-Air LTE Signals. *IEEE Communications Letters* **2018**, *22*, 1934–1937, doi: 10.1109/LCOMM.2018.2854587.
15. Lee, S.; Lee, G.; Jeon, G. Statistical Approaches Based on Deep Learning Regression for Verification of Normality of Blood Pressure Estimates. *Sensors* **2019**, *19*, 2137, doi: 10.3390/s19092137
16. Khattak, H. A.; Islam, S. U.; Din I. U. and Guizani, M. Integrating Fog Computing with VANETs: A Consumer Perspective. *IEEE Communications Standards Magazine* **2019**, *3*, 19–25, doi: 10.1109/MCOMSTD.2019.1800050.
17. Wang, J.; Gao, Q.; Ma, X.; Zhao Y. and Fang, Y. Learning to Sense: Deep Learning for Wireless Sensing with Less Training Efforts. *IEEE Wireless Communications* **2020**, *27*, 156–162, doi: 10.1109/MWC.001.1900409.
18. Redmon, J.; Darknet, [Online]. Available: https://github.com/pjreddie/darknet
19. Redmon, J.; Divvala, S.; Girshick R. and Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* **2016**, 779–788.
20. Sowa, P.; Darknet on OpenCL. [Online]. Available: https://github.com/sowson/darknet.
21. NVIDIA, CUDA Technology. [Online]. Available: http://www.nvidia.com/CUDA.
22. The Khronos Group Inc, OpenCL. [Online]. Available: https://www.khronos.org/opencl.
23. clMathLibraries, clBLAS [Online]. Available: https://github.com/clMathLibraries/clBLAS.
24. Nugteren, C.; CLBlast. [Online]. Available: https://github.com/CNugteren/CLBlast.
25. Everingham, M., Van Gool, L., Williams, C.K.I. et al. The PASCAL Visual Object Classes (VOC) Challenge. *Int J Comput Vis* **2010**, *88*, 303–338, [Online]. Available: https://doi.org/10.1007/s11263-009-0275-4.
26. Lin, T.-Y. *et al.* Microsoft COCO: Common Objects in Context, *arXiv preprint arXiv:1405.0312* **2014**.
27. Karimi, K.; Dickson, N. G. and Hamze, F. A Performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581* **2010**.
28. Intel® Stratix® 10 MX FPGA Development Kit. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/boards and kits/dev-kits/altera/kit-s10-mx.html.
29. Colwell, R. P. The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips, IEEE, 2006.
30. Khronos® OpenCL Working Group Version V2.2-11. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html.
31. Khronos® OpenCL Introduction. [Online]. Available: https://www.slideshare.net/Khronos_Group/opencl-overview-nov-2011.
32. MYESTRO Interactive GmbH Darknet. [Online]. Available: https://github.com/myestro/darknet.
33. CU2CL Translator of CUDA to OpenCL. [Online]. Available: https://github.com/vtsynergy/CU2CL.
34. Darknet-OpenCL GitHub project in C++. [Online]. Available: https://github.com/ganyc717/Darknet--On-OpenCL.
35. Darknet-OnenCL GitHub project in C++. [Online]. Available: https://github.com/Kylin-PHYTIUM/Darknet-On-OpenCL.

36. Koo, Y,; You, C. and Kim, S. OpenCL-Darknet: An OpenCL Implementation for Object Detection. *IEEE International Conference on Big Data and Smart Computing (BigComp)* **2018**, 631–634.
37. Koo, Y.; Kim, s.; Ha, K. OpenCL-Darknet: implementation and optimization of OpenCL-based deep learning object detection framework. *World Wide Web* **2020**, doi: 10.1007/s11280-020-00778-y.
38. Sowa, P. GPU Computing on OpenCL. [Online]. Available: https://iblog.isowa.io/2019/11/05/gpu-computing-on-opencl.
39. Sowa, P.; clBLAS for Multi-GPU GEMM on macOS. [Online]. Available: https://github.com/sowson/clBLAS
40. Capece, N. *et al.* Turning a Smartphone Selfie Into a Studio Portrait. *IEEE Computer Graphics and Applications* **2020**, *40*, 140–147, doi: 10.1109/MCG.2019.2958274.
41. Taylor, A. *et al.* Probing the Limits of Anomaly Detectors for Automobiles with a Cyberattack Framework *IEEE Intelligent Systems* **2018**, *33*, 54–62, doi: 10.1109/MIS.2018.111145054.
42. Falcini, F. *et al.* Deep Learning in Automotive Software. *IEEE Software* **2017**, *34*, 56–63, doi: 10.1109/MS.2017.79.
43. Liao, G. *et al.* Hydroelectric Generating Unit Fault Diagnosis Using 1-D Convolutional Neural Network and Gated Recurrent Unit in Small Hydro. *IEEE Sensors Journal* **2019**, *19*, 9352–9363, doi: 10.1109/JSEN.2019.2926095.
44. Bazrafkan S. and Corcoran, P. M. Pushing the AI Envelope: Merging Deep Networks to Accelerate Edge Artificial Intelligence in Consumer Electronics Devices and Systems. *IEEE Consumer Electronics Magazine* **2018**, *7*, 55–61, doi: 10.1109/MCE.2017.2775245.
45. Lemley, J. *et al.* Deep Learning for Consumer Devices and Services: Pushing the limits for machine learning, artificial intelligence, and computer vision. *IEEE Consumer Electronics Magazine* **2017**, *6*, 48–56, doi: 10.1109/MCE.2016.2640698.
46. Hsu, F. R. *et al.* A Study of User Interface with Wearable Devices Based on Computer Vision. *IEEE Consumer Electronics Magazine* **2020**, *9*, 43–48, doi: 10.1109/MCE.2019.2941463.
47. Ghosh, S. *et al.* Reliable pedestrian detection using a deep neural network trained on pedestrian counts. *IEEE International Conference on Image Processing (ICIP)* **2017**, 685–689, doi: 10.1109/ICIP.2017.8296368.
48. Randriamasy, M. *et al.* Formally Validated of Novel Tolling Service With the ITS-G5. *IEEE Access* **2019**, *7*, 41133–41144, doi: 10.1109/ACCESS.2019.2906046.
49. Developer Reference: OpenCL™ Runtime and Compiler for Intel® Graphics. [Online]. Available: https://software.intel.com/en-us/node/540387
50. Ji, Y. *et al.* Short-term forecasting of available parking space using wavelet neural network model. *IET Intelligent Transport Systems* **2015**, *9*, 202–209, doi: 10.1049/iet-its.2013.0184.
51. Khronos Group Releases OpenCL 3.0 [Online]. Available: https://www.khronos.org/news/press/khronos-group-releases-opencl-3.0.
52. Trivial OpenCL Sadbox with Examples [Online]. Available: https://github.com/sowson/gpucomp

Sample Availability: Samples of the compounds ..... are available from the authors.