The Journal of Supercomputing (2022) 78:13814–13830
https://doi.org/10.1007/s11227-022-04417-3

HDNN: a cross-platform MLIR dialect for deep neural networks

Pablo Antonio Martínez[1] · Gregorio Bernabé[1] · José Manuel García[1]

Accepted: 28 February 2022 / Published online: 25 March 2022
© The Author(s) 2022, corrected publication 2022

Abstract
This paper presents HDNN, a proof-of-concept MLIR dialect for cross-platform computing specialized in deep neural networks. As target devices, HDNN supports CPUs, GPUs and TPUs. In this paper, we provide a comprehensive description of the HDNN dialect, outlining how this novel approach aims to solve the P^3 problem of parallel programming (portability, productivity, and performance). An HDNN program is device-agnostic, i.e., only the device specifier has to be changed to run a given workload in one device or another. Moreover, HDNN has been designed to be a domain-specific language, which ultimately helps programming productivity. Finally, HDNN relies on optimized libraries for heavy, performance-critical workloads. HDNN has been evaluated against other state-of-the-art machine learning frameworks on all the hardware platforms achieving excellent performance. We conclude that the ideas and concepts used in HDNN can be crucial for designing future generation compilers and programming languages to overcome the challenges of the forthcoming heterogeneous computing era.

Keywords High-performance computing · LLVM · MLIR · Heterogeneous computing · Domain-specific languages · Deep neural networks

1 Introduction and motivation

One of the most important factors in the fast evolution of computer science in recent years is the performance improvement in CPUs, which has been developed and vastly improved over time. While Moree's law was true, the center of

✉ Pablo Antonio Martínez
pabloantonio.martinezs@um.es

Gregorio Bernabé
gbernabe@um.es

José Manuel García
jmgarcia@um.es

[1] Computer Engineering Department, University of Murcia, Murcia, Spain

attention in computer architecture was always the CPU. Since Moore's law has ended [6], we need to find new ways to keep enhancing the performance of computers, either with or without the help of CPUs. This trend change opens a vast number of opportunities [8].

Thus, the new direction in computer architecture is looking for alternative architectures, like graphics processing units (GPUs), field-programmable gate arrays (FPGAs), or domain-specific accelerators (DSAs). Accelerators can offer orders of magnitude improvements in performance and energy efficiency compared to CPUs [3]. A specialized core can perform the operations for which it has been designed much more efficiently than a general-purpose core. Nowadays, computing systems based on heterogeneous chips include general-purpose cores, graphics processing units, and other specialized accelerator cores (FPGAs and DSAs) in the same chip.

In addition to heterogeneous computing, another recent dramatic irruption is machine learning, specifically deep learning. Since the first applications of deep neural networks (DNNs) to speech recognition and image processing, the importance of DNNs has grown exponentially. Many data centers and companies have opted for using DSAs for deep learning [11], which offer the performance and power efficiency needed to support the machine learning workloads nowadays. The appearance of multiple devices of execution has created a problem in parallel programming languages often known as P^3 [21]. We need programming languages to provide portability, productivity and performance among target devices in heterogeneous environments.

This paper presents heterogeneous deep neural network (HDNN), a proof-of-concept MLIR dialect for deep neural networks. HDNN currently supports convolution and softmax layers along with basic I/O functionality. HDNN is built with a compiler based on MLIR (which we refer to as `hdnn-opt`). HDNN generates code for CPUs, GPUs and TPUs, a domain-specific accelerator for machine learning.

HDNN programs are portable thanks to our MLIR-based ecosystem, following an idea of progressive lowering of high-level, device-agnostic to low-level operations, device-specific operations. Regarding productivity, HDNN allows programming using a single device-agnostic source code language using MLIR. Furthermore, HDNN uses optimized libraries for performance-critical operations achieving competitive performance against state-of-the-art approaches. Moreover, this approach has negligible overhead and thus allows for taking advantage of the full potential of the underlying libraries.

HDNN is a step forward in the P^3 solution by providing:

- Portability: because each hardware device needs different programming languages and frameworks, software complexity grows exponentially. Specifically, to run a given program in n heterogeneous devices, software developers need n different source codes, one for each device. However, HDNN source code is written once and can be targeted to three devices (CPUs, GPUs and TPUs). In other words, HDNN source code is device-agnostic and can be launched to any of the supported devices seamlessly.

- Productivity: heterogeneous languages often suffer from being too complex for the developer, reducing the productivity of software development. HDNN is a dialect for deep neural networks, and one of its design principles is simplicity. Softmax and convolution layers have been developed with domain-specific functionality, so HDNN is inherently easier to use than other languages and frameworks in the deep learning area.
- Performance: most state-of-the-art heterogeneous languages often provide worse performance than native programming. This issue is commonly referred to as performance portability. HDNN relies on optimized libraries for the heavy, performance-critical workloads to achieve it. Internally, HDNN uses the best-performant libraries for the deep learning field (oneDNN and cuDNN). Therefore, HDNN provides the best-known performance for each architecture.

The rest of this paper is organized as follows: Sect. 2 presents the background of accelerators and heterogeneous languages, LLVM and MLIR frameworks. In Sect. 3, we present the HDNN frontend, describing our proposed dialect at the programmer level. In Sect. 4, we present the HDNN backend, outlining how HDNN works “under the hood”. Section 5 evaluates HDNN from three points of view: portability, productivity and performance. Our proposal is compared to two state-of-the-art reference machine learning frameworks for the latter. Finally, Sect. 6 concludes the paper and gives some hints for future work.

2 Background and related work

2.1 Heterogeneous hardware and software

GPUs are the most common kind of accelerator. They are used for many domains (like graphics processing or deep learning). In the next iteration, in terms of specialization, we find FPGAs. For certain applications, FPGAs offer even better performance and efficiency than GPUs. Domain-specific accelerators (DSAs), also referred to as application-specific integrated circuits (ASICs), are the best in terms of specialization. To understand the importance of accelerators nowadays, we only need to look at the variety of accelerators available for each domain, like the Cerebras accelerator [17] or the tensor processing unit (TPU) [11] in the deep learning area.

From a software standpoint, the management of this heterogeneity is a challenge. Each hardware device works differently and is coded with specific languages, so performance is achieved following different approaches. Therefore, we seek software solutions that offer performance, portability and productivity (P^3) [21]. To manage the programming of the heterogeneous device, one approach is to have languages that allow running a program in different hardware devices using a unified source code. This way, software only needs a single-source code base for each device (CPU, GPU, etc.). In such a case, we say that the language is portable. The problem with these languages is that they are often too complex for the developer, reducing software development productivity. The learning curve to master one of these languages

is relatively high, which may be a barrier for new developers. Moreover, most of these languages often provide worse performance than native programming, which is commonly referred to as performance portability. An application is performant-portable when it achieves good performance on *all* the hardware platform that the application supports.

The problem known as P^3 [21, 25] is to develop a language that is portable and productive and provides performance portability. This problem is usually a trade-off [25] because it is impossible to get the best of the three worlds at the same time. Some frameworks and libraries have appeared trying to solve the P^3 problem. Some examples are: oneAPI [2], PHAST [20], OpenCL [23], HPVM [13] and Kokkos [5].

2.2 LLVM and MLIR

Regarding compilers, one of the de facto standards for building compilers is LLVM [14] or the more recent MLIR [15]. LLVM is a collection of modular compiler and toolchain technologies [14]. One of its goals is to provide a compiler infrastructure that compiles from and to an intermediate representation (IR). Intermediate representation is a program representation that sits between the source code and the compiled program. In the case of LLVM IR, it is based on static single assignment (SSA), which provides many important features in compilation workflows. The IR provides a lot of flexibility in the compilation process, which further translates into different optimizations. LLVM has been used for different applications in academia (HPVM [13] or Glow [22]) and industry (oneAPI [2], OpenCL [23] or XLA [16]). Furthermore, the idea of using optimized libraries to enhance the performance of applications using LLVM has also been studied [4].

One of the latest sub-projects of LLVM is MLIR [15]. A key difference between both is that MLIR introduces the idea of multilevel IR. Instead of translating source code to LLVM IR directly, MLIR offers a framework to do a progressive lowering of the IR. This way, IR starts from a high-level IR representation that gets transformed into lower-level IR at each compiler pass. This is known as *progressive lowering*. Those transformations are technically referred to as *transformation passes*. Each of the *transformation passes* modifies the IR with different goals. Thanks to this step-by-step transformations, the high-level semantics of the high-level code is preserved during IR transformations.

Even though MLIR was released recently, it already has had a significant impact since many projects use it in fields like image processing [7], quantum computing [19], or polyhedral compilation [12].

3 HDNN frontend

The `hdnn` dialect provides a set of operations to work with neural networks, but it does not add new data types as it was designed to cooperate with the already existent data types in MLIR, like the `tensor`. Currently, the dialect is not particularly productive from the programming standpoint because it has to be used directly at the

MLIR IR level. For that reason, the hdnn dialect is not designed as a user-friendly DSL. We leave for future work to design and implement a top-level DSL to ease the programming task.

3.1 HDNN operations

The hdnn dialect provides three kinds of operations: operations for creating regions, operations for the deep learning domain, and auxiliary operations. The only operation available for creating regions is hdnn.launch. The launch encompasses an MLIR region that may contain any operation. Those operations will be launched to the device specified in the operation argument. For deep learning, the hdnn dialect currently provides two layers: softmax and convolution. Both layers can only work in inference mode since they implement the feedforward phase. Finally, hdnn provides auxiliary functions to print an arbitrarily sized tensor, create an arbitrarily sized tensor with random data (useful to fill the layers with data), and an operation to mark the end of an MLIR function. hdnn operations are detailed in Table 1.

3.2 HDNN programming

In addition to the hdnn dialect, the HDNN compiler supports the tensor, affine, memref, and standard dialects. In essence, this means that the programmer can use any of these dialects to build an HDNN compliant program. However, the normal procedure in an HDNN program is to use only the hdnn and the tensor dialects, while the rest of them are only used in further lowering passes.

Running a layer in HDNN is straightforward, as can be seen in Fig. 1. The first operation corresponds to the launch operation. This operation dictates the device in which the computations will be executed (in the example, the GPU is selected). Then, random 3D tensors are generated. After that, the convolution layer is executed and the output is used by the softmax. Note that the MLIR code inside the launch operation is simple but, more importantly, device-agnostic. Only the hdnn.launch operation parameter must be modified to change the target device. At the moment, one limitation of HDNN launch operations is that they cannot be mixed together as they are treated as completely different tasks. Thus, the current implementation dictates that only the data created inside a region can be used. Still, it is not possible to use data from another region or data created outside of the region.

HDNN programming is straightforward because the function calls hide the complexity of an HDNN program. This makes sense because programmers often do not need to know or modify the code of a neural network layer. Furthermore, other common constructions like conditions or control flow can still be used in HDNN. One of the strengths of HDNN is the fact that data are stored in tensors, which are extremely flexible (following the MLIR idea, they are expressed in a very-high-level way). Thus, connecting different layers in HDNN is straightforward, as shown in Fig. 1.

Table 1 Summary of HDNN available operations to the user with their description. `f32` refers to simple precision data types

Operations for Creating Regions	
hdnn.launch {`dev` = *device*} { ... }	Create a region and launch everything inside it to the device specified in the dev parameter, which can take the values of: "`cpu`", "`gpu`" or "`tpu`"
Operations for Deep Learning	
hdnn.softmax(%i) {`iters` = *N*} : (tensor <NxCxWxf32>) -> tensor <NxCxWxf32>	Runs the softmax layer with input %i. iters parameter is optional and allows to run the layer for the specified number of iterations. The softmax operation receives a 3D tensor and outputs a 3D tensor of the same dimensions.
hdnn.conv(%i, %w, %b) {`iters` = *N*} : (tensor <NxCxHxWxf32>, tensor <N'xCxFxFxf32>, tensor <N'xf32>) -> tensor <NxN'x?x?>	Runs the convolution layer with the input passed as: • %i: the input image (a 4D tensor with dimensions NxCxHxW) • %w: the input weights (a 4D tensor with dimensions N'xCxFxF) • %b: the input bias (a 1D tensor with dimension N') and outputs the result as a 4D tensor. iters parameter is optional and allows to run the layer for the specified number of iterations.
Auxiliary Operations	
hdnn.print (%i): tensor <?x?x?x?xf32>	Prints a tensor of arbitrary size %i to the standard output
hdnn.random(): tensor <?x?x?x?xf32>	Creates a tensor of arbitrary size with random data
hdnn.return	Used to end a MLIR function (cannot be omitted)

```
func @main() -> i32 {
  hdnn.launch {dev = "gpu"} {
    %imgs = hdnn.random() : tensor <10x1x28x28xf32>
    %weig = hdnn.random() : tensor <20x1x5x5xf32>
    %bias = hdnn.random() : tensor <20xf32>

    %cout = "hdnn.conv"(%imgs, %weig, %bias) :
            (tensor <10x1x28x28xf32>, tensor <20x1x5x5xf32>,
            tensor <20xf32>) -> tensor <10x20x24x24xf32>
    %sout = "hdnn.softmax"(%cout) : (tensor <10x20x24x24xf32>) ->
            tensor <10x20x24x24xf32>

    hdnn.print %sout : tensor<10x20x24x24xf32>
  }
  hdnn.return
}
```

Fig. 1 HDNN program that runs the convolution and softmax layer in GPU

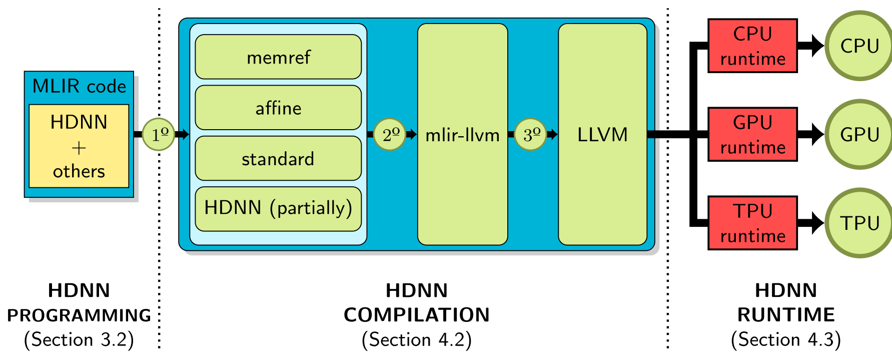

Fig. 2 HDNN compilation flow

4 HDNN backend

The HDNN backend architecture is built up of two components. The first one is the HDNN compiler (which we refer to as `hdnn-opt`), and the second one is the HDNN runtime (composed by the CPU, GPU and TPU runtimes).

4.1 HDNN architecture

The HDNN compilation flow is depicted in Fig. 2. First, the HDNN compiler transforms the MLIR code to LLVM. The original MLIR code suffers different modifications (partial lowering) before being converted to the LLVM IR code. This code is then compiled to an object file using clang. The HDNN runtime is written in C++, so it can also be compiled to object files using any compiler. Finally, object files are linked into a single binary file. Depending on the original MLIR code, the final binary file is executed on CPU, GPU or TPU. However, as it is tied to a specific

device, different devices cannot execute the MLIR code concurrently. Nonetheless, thanks to the HDNN design, this limitation could be eliminated easily to allow co-execution, which is a promising line for future work.

4.2 HDNN compilation process (*lowering*)

An HDNN program has to be *lowered* to transform the HDNN high-level IR code (MLIR) to a lower-level IR (LLVM). The lowering process is divided into three different passes, as depicted in Fig. 2.

4.2.1 First transformation pass

The first pass (marked as 1° in Fig. 2) takes as input the HDNN source file. In this file, operations are used inside the launch operation, so they are device-agnostic. The main goal is to replace device-agnostic with device-specific operations. For this task, the HDNN dialect has not only device-agnostic operations but also device-specific ones. For example, for the convolution, HDNN has the device-agnostic `hdnn.conv` and the device-specific `hdnn.conv.cpu`, `hdnn.conv.gpu`, and `hdnn.conv.tpu`. The multi-level nature of MLIR is convenient in this case; it is very interesting for heterogeneous compiling, as multilevel IR can transform a generic IR to one that is device-specific. In addition to the mentioned transformations, this pass also adds the operation `hdnn.init_gpu` when a launch operation on GPU is detected (explained in the following pass). Another objective is to lower high-level data types to lower-level ones, e.g., the `tensor` data type is lowered to the `memref` dialect to work with lower-level operations, like loads and stores.

When the pass has finished, only the `affine`, `memref`, and `standard` dialects are legal. Lastly, the `hdnn` dialect is said to be *dynamically legal* because not all the operations in the dialect are legal, just the device-specific operations generated by the compiler in the current pass. These device-specific operations are not available to the user as only the compiler can generate them. Note that the MLIR code generated in this pass is no longer device-agnostic since we have already specialized the operations to target a specific device. We leave for future work to experiment with more sophisticated approaches, like dynamically selecting the device depending on the system's load, thus allowing co-execution.

4.2.2 Second transformation pass

The input of the second pass (marked as 2° in Fig. 2) is the output of the previous one, which contains deep learning device-specific operations (neural network layers). The device-specific operations found in this pass have to be lowered. To do so, the `hdnn-opt` inserts calls to the HDNN runtime, which uses optimized libraries for running the layers. Depending on the layer and the device selected, the compiler also inserts calls to initialize the library used to do the computations. Data types used by the library also have to be initialized, and the compiler generates the IR

code at this moment. It is worth noting that when this pass has finished, only the `mlir-llvm` dialect is legal.

The rest of the transformations performed by the second pass are dependent on the device being targeted:

- *CPU lowering:* this lowering is the easiest because there are no memory movements. Thus, the compiler inserts the calls to the CPU runtime to lower the code when the CPU is selected.
- *GPU lowering:* when the IR contains GPU-specific operations (e.g., a convolution launched to the GPU), the compiler needs to provide mechanisms for data management between CPU and GPU. For this duty, MLIR provides a GPU dialect. During the second pass, the compiler generates GPU operations (that are part of the `gpu` dialect). However, the `gpu` dialect operations cannot be converted to `mlir-llvm`.[1] Our approach to circumvent this problem is to do a transitive lowering of the `gpu` dialect. Therefore, these operations (part of the `gpu`) are immediately lowered to other lower-level operations instead of leaving them to be translated in a further pass (transitive lowering). Another important thing to do is to lower the `hdnn.init_gpu` operation. This HDNN operation aims to initialize both a CUDA stream and a cuDNN handler. This operation is lowered in this pass, and it is done by calling the HDNN GPU runtime, which will take care of this.
- **TPU lowering:** even though we use a specific dialect to handle the GPU, we do not follow the same approach for TPU. The `gpu` dialect is used to manage memory allocations and memory transfers between CPU and GPU. Following this idea, there should be a dialect for the TPU, but there is no TPU dialect in standard MLIR, and we did not design it either. This decision comes from a limitation in the TPU runtime of HDNN. Currently, the memory allocation and movement in the TPU are managed implicitly, so it does not make sense to have a dialect that can express these operations as they cannot be executed using our current TPU runtime. The TPU lowering inserts calls to the TPU runtime, which handles both the memory and the computations in the TPU. A lower-level TPU runtime, along with a `tpu` dialect, would allow explicit memory management, as well as other optimizations.

4.2.3 Third transformation pass

The last pass (marked as 3° in Fig. 2) transforms the `mlir-llvm` into LLVM. This transformation is an automated process managed entirely by MLIR. Thus, the HDNN compiler invokes the appropriate MLIR function to do it. The output of this pass is LLVM code, which will be further compiled with the HDNN runtime into the executable file.

[1] While it has been discussed if this is how it should work or not, at the moment of writing, converting the `gpu` dialect to `mlir-llvm` is not possible.

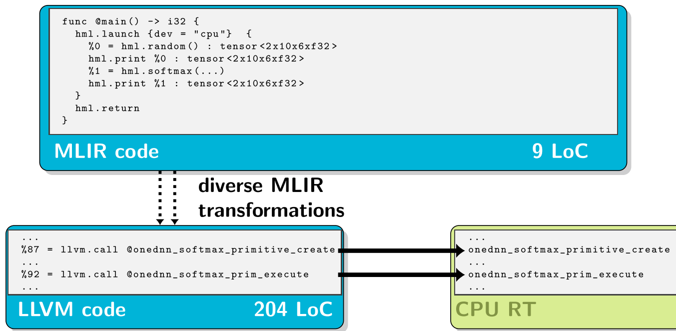

Fig. 3 HDNN example lowering with runtime communication

4.3 HDNN runtime

The HDNN runtime consists of one generic runtime and three device-specific runtimes: the CPU, GPU and TPU runtimes. The generic runtime is executed on the CPU. It provides functions for time measurements (useful for the evaluation) and functions for generating random floating-point values, which are called when the operation `hdnn.random` is used.

The HDNN runtime essentially acts as a middleware between the LLVM code generated by MLIR and the corresponding optimized library. Each runtime defines custom functions to operate with the corresponding library. These functions are used by the LLVM IR code, as depicted in Fig. 3. The HDNN compiler needs to generate function definitions for all of the functions defined in the runtime to connect LLVM and the libraries. These definitions are not linked when the LLVM code is generated since they reference functions that do not exist in the LLVM code. They are linked when the LLVM code is compiled altogether with the HDNN runtime, because the device-specific runtime is the one that provides the needed functions.

The device-specific runtimes provide two functions: to manage library-specific data structures and run the softmax and convolution layers. They delegate the layers' computation to the optimized libraries to do so. We chose the best-performant deep-learning library for each device. oneDNN is used for the CPU runtime, cuDNN for the GPU runtime, and PyTorch for the TPU runtime. The case of the TPU is different because there is no optimized, ready-to-use library, as is the case of oneDNN and cuDNN. We explored different alternatives like PyTorch, TensorFlow or XLA. PyTorch and TensorFlow provide an easy interface to run any layer on the TPU, but they only work with Python (both libraries offer limited support with languages different than Python). Therefore, to use TPUs from HDNN, we designed a TPU runtime that takes the inputs from HDNN, sends them to a Python code, and returns the data to HDNN coming from Python. This Python code uses PyTorch to run the layers on the TPU and XLA to communicate with the TPU itself. However, running a layer using PyTorch directly only makes use of 1 TPU core. Thus, we implemented a basic algorithm inside the TPU runtime to parallelize the code among the TPU cores. The algorithm follows an `allreduce` scheme, which is a common approach

Table 2 Source lines of code (SLOC) measured in different programming languages

Layer	HDNN	DeepDSL	oneAPI
Softmax	7	13	60
Convolution	9	13	78
Both	10	14	80

in the distributed DNN execution [1]. Essentially, the number of batches is divided between the number of TPU cores (data parallelism), and the master core gathers the partial results into a single one at the end of the execution.

5 Evaluation

In this section, we evaluate how portable, productive and performance portable HDNN is (P^3).

5.1 Portability and productivity

Figure 1 shows how the same code can be executed in different devices only by changing the device specifier. This feature demonstrates the portability of HDNN across different hardware devices.

Regarding productivity, we compared the complexity of an HDNN program with other alternatives. However, comparing a program complexity is yet not something purely objective. Different metrics have been proposed over time, but here we will use the lines of code of a program for simplicity. Before calculating the lines of code, non-essential lines (blank lines, comments) were removed, allowing for a fair comparison. We compared DeepDSL [26], a DSL for deep learning, and oneAPI [2] against HDNN. We use three basic programs executing softmax, convolution, and both as a testbed. The evaluated programs for oneAPI were based on previous work [18], and the DeepDSL programs were manually built from scratch. Table 2 summarizes the results. HDNN and DeepDSL needed similar lines, while oneAPI required more. This result emphasizes that HDNN source code is straightforward so that good productivity can be achieved with it, in the line of existent DSL.

5.2 Performance portability

This section compares the performance using the internal HDNN library against using a machine learning framework. This analysis is not meant to be a comparison of machine learning frameworks but instead to shed a bit of light on the competitiveness of HDNN concerning already existent approaches. In CPU and GPU, HDNN was compared against Caffe, and in the TPU, against PyTorch. For the first comparison, we developed custom softmax and convolution implementations using oneDNN and cuDNN. Essentially, these programs create tensors with random data and call the appropriate libraries. This way, the overhead caused by the communication

Table 3 Hardware specifications for the testbed environment (per chip)

	CPU (Intel)	GPU (NVIDIA)	TPU (Google) [9, 10]
Model	Xeon Gold 6238	RTX 2080 Ti	TPUv2
Release date	Q2 2019	Q3 2018	Q2 2017
Manufacturing process	14 nm	12 nm	16 nm
TDP	140 W	250 W	280 W
Configuration	2 Chips per host	1 chip per host	4 chips per host[1]
Cores/chip (total)	22 (44)	4352	2 (8)
Maximum frequency	3700 MHz[2]	1545 MHz	700 MHz
Peak performance (SP)	2.95 TFLOP/s	13.44 TFLOP/s	3.00 TFLOP/s
Memory size (on-chip)	30.25 MB L3	5.5 MB L3	32 MB
Memory type (off-chip)	DDR4 2933 MHz	GDDR6 1750 MHz	HBM
Memory bandwidth (off-chip)	140.7 GB/s	616.0 GB/s	~ 600.0 GB/s

[1] A single TPUv2 board contains 4 TPU chips with 2 TPU cores each.

[2] While maximum frequency is 3.7 GHz, real frequency when the CPU is using all the cores and running AVX-512 code is 2.1 GHz. Therefore, we used 2.1 GHz to calculate the peak performance

between HDNN and the corresponding runtime could be measured. For the second comparison, we developed tiny testing programs using Caffe and PyTorch. The performance evaluation was accomplished using only the feedforward phase, as the current HDNN implementation does not support backpropagation.

5.2.1 Test bed

The evaluation platform was divided into two parts. On the one hand, we have the CPU and GPU hardware platform, which was equipped with a double-socket Cascade Lake Intel Xeon Gold 6238 and a Turing NVIDIA RTX 2080 Ti. On the other hand, we have the TPU hardware platform, which was equipped with a TPUv2. Hardware details for the CPU, GPU and TPU used are shown in Table 3.

The CPU and GPU system ran on a CentOS 8.2 (4.18.0 kernel). LLVM was downloaded from the official Git repository, obtaining the code corresponding to the commit `cf72768`. We used this LLVM version to build HDNN. Our custom implementations of softmax and convolution examples of oneDNN and cuDNN were built using gcc 8.3.1. Furthermore, we used oneDNN version 1.9.6 and cuDNN version 8.2.4. Caffe testing programs were developed using the Caffe release in the official Git repository with commit `99bd997`. For the TPU, we used a Cloud TPU service in Google Cloud Platform. This system ran on a Debian 10 (4.19.0-14 kernel). In this case, we used the same LLVM version to build the HDNN compiler, and for the TPU backend we used Python 3.7 and PyTorch 1.9.

The evaluation was performed using simple precision data types. It is important to note that the TPU was designed to work in half precision, so its potential is reduced in this scenario, compared to the CPU and the GPU. We leave for future work the support of half precision workloads, which would make much better use of TPUs and other machine learning accelerators.

Table 4 Execution time of the softmax layer in CPU, GPU and TPU (in seconds)

Input	CPU			GPU			TPU		
	HDNN	Caffe (openBLAS)	Speedup	HDNN	Caffe (cuDNN)	Speedup	HDNN	PyTorch	Speedup
1	0.237	0.001	0.01×	0.001	0.015	15.0×	0.990	0.950	0.96×
2	33.32	151.88	4.55×	0.562	0.821	1.46×	1.220	1.248	1.02×
3	263.4	1529.1	5.80×	5.123	7.482	1.46×	31.69	31.88	1.06×
4	333.1	1554.5	4.66×	4.482	7.727	1.59×	30.31	30.25	0.99×

For the performance evaluation, we measured the execution time of each implementation (given in seconds), taking into account only computation times. Each experiment was repeated 5 times, and the values shown are the average over these 5 independent runs. For the softmax layer, we run each input during 1000 iterations and for the convolution, we run each input during 100 iterations.

5.2.2 Softmax

For the softmax layer, we designed 4 input sizes expressed in the triple (N,C,W), where N is meant to be the number of classifications that softmax needs to calculate, C is the number of channels and W is the width of the vector containing the softmax data. Input 1 is (2, 10, 6), Input 2 is (200, 100, 600), Input 3 is (2000, 100, 600) and Input 4 is (200, 100, 6000).

Performance results are shown in Table 4. It is worth noting that Caffe was compiled using openBLAS since it achieved better results than MKL.[2] In cuDNN, the `CUDNN_SOFTMAX_ACCURATE` algorithm was used (both for HDNN and the cuDNN test program) to avoid possible overflows when computing the softmax. When we compared the performance achieved by HDNN against the optimized libraries (oneDNN and cuDNN), we found that HDNN achieves the same performance as using the libraries directly. Therefore, we omitted the results in the tables for brevity. As we theorized before, we can conclude that HDNN suffers no overhead when communicating to the specialized backends.

Except for the first input, HDNN achieves speedup near or bigger than 5x compared with Caffe in CPU. The performance degradation in the first input comes from the fact that this input is tiny. For bigger workloads, HDNN consistently outperforms Caffe. In GPU, HDNN is around 1.5x faster than Caffe for all input except the first, reaching 15x. Finally, the results in TPU are very similar to the ones obtained by PyTorch.

[2] MKL outperformed openBLAS for all the operations needed in the softmax layer except for the division, which ran very slow compared to openBLAS.

Table 5 Execution time of the convolution layer in CPU, GPU and TPU (in seconds)

Input	CPU			GPU			TPU		
	HDNN	Caffe (MKL)	Speedup	HDNN	Caffe (cuDNN)	Speedup	HDNN	PyTorch	Speedup
1	0.101	0.505	5.01×	0.004	0.132	30.0×	1.016	1.063	1.05×
2	0.145	1.029	7.08×	0.012	0.162	13.2×	0.902	0.944	1.05×
3	0.145	1.029	7.08×	1.288	1.720	1.33×	2.226	2.227	1.02×
4	7.091	9.561	1.34×	1.550	2.094	1.35×	2.001	2.018	1.01×
5	7.898	13.722	1.73×	4.158	5.801	1.39×	5.130	5.229	1.02×
6	14.428	39.663	2.74×	1.157	2.120	1.83×	3.102	3.100	1.00×

5.2.3 Convolution layer

For the convolution layer, we designed a set of inputs based on real neural networks, for which we used the Sze et al. survey [24]. In this evaluation, we set the number of batches to 100 for all inputs. Inputs 1 and 2 are representative of the MNIST dataset. Although the MNIST dataset images are gray scale, we also tried using color images (3 channels instead of 1). Thus, the Input 1 size is $28 \times 28 \times 1$, and Input 2 is $28 \times 28 \times 3$. Both have 5×5 filters, but Input 1 has 20 filters, and Input 2 has 50. Inputs 3, 4, and 5 represent AlexNet networks (input sizes $227 \times 227 \times 3$ with 96 filters, and filter sizes of 3×3, 5×5, 11×11, respectively). Input 6 is based on ResNet (input size of $224 \times 224 \times 3$, with 64 7×7 filters).

We used the direct convolution algorithm in oneDNN (CPU), the only available algorithm that worked. In cuDNN, we used `cudnnGetConvolutionForward-Algorithm` to automatically get the fastest algorithm for the convolution in each case. We used this approach for HDNN and the cuDNN test program. In HDNN, the best algorithm is queried once and is used for all the iterations. According to this function, the best algorithm was `IMPLICIT_GEMM` for input 1, `IMPLICIT_PRECOMP_GEMM` for inputs 2,3,4 and 6, and `FFT_TILING` for input 5.[3]

Convolution performance results are shown in Table 5. We evaluated the performance using oneDNN and cuDNN directly, and we did not appreciate any negative impact on the performance in this case either, so we also omitted the results of oneDNN and cuDNN. In both the CPU and the CPU, HDNN achieves significant enhancements against Caffe, which highlights the fact that HDNN is competitive, thanks to the use of optimized libraries. As happened with the softmax layer, HDNN performs similarly to PyTorch. Therefore, we believe that the TPU-distributed algorithm implemented in the TPU runtime and the use of PyTorch-optimized primitives effectively take advantage of the full power of the TPU.

[3] The full name of the algorithms (which always starts with `CUDNN_CONVOLUTION_FWD_ALGO_`) was omitted for brevity.

In the light of the results, we can conclude that HDNN achieves competitive results against other machine learning frameworks. Even though a given HDNN program is only written once, it can be targeted to different hardware devices without any changes, and it also provides solid performance results.

6 Conclusions and future work

Heterogeneous computing is the future but also the present in computer architecture. However, heterogeneous language proposals often suffer worse performance than native programming. While they provide a unified language for heterogeneous computing, they fail to provide reliable performance.

In this work, we have proposed HDNN, a deep learning MLIR dialect for heterogeneous computing. HDNN provides a unified interface to run softmax and convolution layers, executed on x86_64 CPUs, NVIDIA GPUs and Google TPUs. Overall, HDNN provides:

- An unified language for deep learning in heterogeneous environments. Using different hardware devices is easy since the source code is the same, and only the device specifier has to be changed to select the desired device.
- Excellent performance. Our experiments show that HDNN is execution time competitive, thanks to the underlying optimized libraries.

With HDNN, we have shown a novel approach to the P^3 problem. HDNN is programming portable as its code is device-agnostic. To provide programming productivity, DSLs could be used, as they hide the complexity of specific domains. Lastly, we have demonstrated that HDNN is also performance portable using the best existing implementation for each device and workload, reusing specialized engineers' work. This approach is the complete opposite of a common idea in heterogeneous computing, trying to compile a unique code efficiently for multiple platforms. This goal is hard to fulfill because it is very complex to generalize a code to run it on completely different devices.

We plan to design a high-level DSL for HDNN to improve its productivity and usability for future work. Another relevant improvement to HDNN is implementing other essential layers in DNNs, like fully-connected and activation layers. By doing so, we could run complete neural network architectures. While the range of hardware covered by HDNN is decent, we also aim to support other hardware, especially FPGAs. To do so, we would not seek to generate device-specific code from HDNN but instead to run already implemented, optimized routines on the FPGA. Finally, a good research line is to add a scheduler to HDNN, allowing the co-execution of a program in several devices.

Acknowledgements Grant RTI2018-098156-B-C53 funded by MCIN/AEI/10.13039/501100011033 and by “ERDF A way of making Europe.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Ben-Nun T, Hoefler T (2019) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput Surv. https://doi.org/10.1145/3320060
2. Intel Corporation (2022) Intel oneAPI Programming Guide. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html (online). Accessed 03 Feb 2022
3. Dally William J, Yatish T, Song H (2020) Domain-specific hardware accelerators. Commun ACM 63(7):48–57. https://doi.org/10.1145/3361682
4. De Carvalho JP, Kuzma B, Korostelev I, Amaral JN, Barton C, Moreira J, Araujo G (2021) KernelFaRer: replacing native-code idioms with high-performance library calls. ACM Trans Arch Code Optim. https://doi.org/10.1145/3459010
5. Edwards HC, Trott CR (2013) Kokkos: enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), pp 18–24
6. Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: ISCA '11. Association for Computing Machinery, New York, pp 365–376. https://doi.org/10.1145/2000064.2000108
7. Gysi T, Müller C, Zinenko O, Herhut S, Davis E, Wicky T, Fuhrer O, Hoefler T, Grosser T (2020) Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation. ACM Trans Architect Code Optimization 18(4):1-23 arXiv:2005.13014
8. Hennessy John L, Patterson David A (2019) A new golden age for computer architecture. Commun ACM 62(2):48–60. https://doi.org/10.1145/3282307
9. Jouppi NP, Hyun Yoon D, Ashcraft M, et al (2021) Ten lessons from three generations shaped Google's TPUv4i : industrial product. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE Computer Society, Los Alamitos, pp 1–14. https://doi.org/10.1109/ISCA52012.2021.00010
10. Jouppi Norman P, Hyun YD, George K et al (2020) A domain-specific supercomputer for training deep neural networks. Commun ACM 63(7):67–78. https://doi.org/10.1145/3360307
11. Jouppi NP, Young C, Patil N, et al (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, New York. Association for Computing Machinery, pp 1-12. https://doi.org/10.1145/3079856.3080246
12. Komisarczyk K, Chelini L, Vadivel K, et al (2020) PET-to-MLIR: a polyhedral front-end for MLIR. In: 2020 23rd Euromicro Conference on Digital System Design (DSD), pp 551–556. https://doi.org/10.1109/DSD51259.2020.00091
13. Kotsifakou M, Srivastava P, Sinclair Matthew D et al (2018) HPVM: heterogeneous parallel virtual machine. SIGPLAN Not. 53(1):68–80. https://doi.org/10.1145/3200691.3178493
14. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto
15. Lattner C, Amini M, Bondhugula U, Cohen A, et al (2020) MLIR: a compiler infrastructure for the end of Moore's law. arXiv:2002.11054
16. Leary C, Wang T (2017) XLA: TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html (online). Accessed 03 Feb 2022

17. Lie S (2019) Wafer-scale deep learning. In: 2019 IEEE Hot Chips 31 Symposium (HCS), pp 1–31. https://doi.org/10.1109/HOTCHIPS.2019.8875628
18. Martínez PA, Peccerillo B, Bartolini S, García JM, Bernabé G (2022) Applying Intel's oneAPI to a machine learning case study. Concurr Comput Pract Exp. https://doi.org/10.1002/cpe.6917
19. McCaskey A, Nguyen T (2021) A MLIR dialect for quantum assembly languages. arXiv:2101.11365
20. Biagio P, Sandro B (2019) PHAST—a portable high-level modern C++ programming library for GPUs and multi-cores. IEEE Trans Parallel Distrib Syst 30(1):174–189. https://doi.org/10.1109/TPDS.2018.2855182
21. John Pennycook S, Sewall Jason D, Jacobsen Douglas W (2021) Navigating performance, portability, and productivity. Comput Sci Eng 23(5):28–38. https://doi.org/10.1109/MCSE.2021.3097276
22. Rotem N, Fix J, Abdulrasool S, et al (2019) Glow: graph lowering compiler techniques for neural networks. arXiv:1805.00907
23. Stone John E, David G, Guochun S (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(3):66–73. https://doi.org/10.1109/MCSE.2010.69
24. Vivienne S, Yu-Hsin C, Tien-Ju Y, Emer Joel S (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE 105(12):2295–2329. https://doi.org/10.1109/JPROC.2017.2761740
25. Michael W (2021) Performant, portable, and productive parallel programming with standard languages. Comput Sci Eng 23(5):39–45. https://doi.org/10.1109/MCSE.2021.3097167
26. Zhao T, Huang X, Cao Y (2017) DeepDSL: a compilation-based domain-specific language for deep learning. arXiv:1701.02284

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.