International Journal of Engineering & Technology, 13 (2) (2024) 398-408

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Implementing the Multi-Layer Perceptron Algorithm on NVidia GPUs

Laszlo Marak*

[1] *J. Selye University, Bratislavská cesta 3322, 945 01, Komárno, Slovakia*
[*] *Corresponding author E-mail: marakl@ujs.sk*

Abstract

With the adoption of machine learning algorithms for image processing tasks and the ever growing need for embedded device applications, the developers use several methods to optimize the computational efficiency of their applications. Optimization of algorithms can be challenging and developers must apply non-trivial strategies to exploit the computational resources of computer architectures more efficiently. In this article we are describing an efficient GPU implementation for the Multi-Layer Perceptron (MLP) algorithm. The MLP is a basic algorithm for machine learning and artificial intelligence, and is an excellent example of the difficulties surrounding GPGPU programming and optimization. As an independent observation we discuss the memory management of GPU-s and methods to simplify the memory allocation process.

Keywords: *MLP, Multi-Layer Perceptron, GPU Implementation, OpenCL, Parallel Implementation*

1. Introduction

The traditional *big O* metric, used to describe the algorithms "efficiency" has limitations for practical usability. For example, *big O* hides any constant multiplier, which can result differences in orders of magnitudes between implementations. As computer architectures become more complex, the *big O* efficiency is no longer sufficient to describe the practical usability of an implementation of an algorithm. The matter becomes even worse in the new emerging heterogenous systems, as the processing throughput or the *theoretical computational capacity* can only be achieved if several strict conditions are fulfilled. The most important problems arising during heterogenous optimization are: 1. parallelization 2. warping 3. memory bandwidth bottleneck 4. local memory (cache) usage 5. static scheduling and 6. work-item synchronization. An efficient implementation has to address all of these issues. In this paper we are discussing a particular class of algorithms, the Multi-Layer Perceptron, and its efficient implementation. The perceptron is a core algorithm used for machine learning, and therefore properly understanding implementation challenges is crucial in exploiting its full potential.

2. Prior Work

Machine Learning and specifically neural networks have become one of the most popular algorithms for classification. Prior application specific algorithms have often been replaced by generic neural networks trained for specific application [22, 31, 20]. One of the most challenging issues arising from the application of neural networks remains the computational complexity that these algorithms require. It has been apparent, that classical architectures, such as x86 are not suited for high performance computation [24]. Even with modern extensions, such as SSE or AVX there processors lag behind in executing neural networks [10, 11]. In the past few years the tendency has been clear. Virtually each major CPU manufacturer has acquired or implemented its own GPU solutions. In some cases, such as in the case of NVidia, the company has diversified into the generic computing market by introducing ARM based processor cores in it's product lineup [9]. For the moment NVidia seems to be the leader of GPGPU computing as well as machine learning, having a 65% market share of the machine learning market and 97% market share in the accelerator market [19]. With its CUDA platform NVidia remains the "go to" solution for high performance computing tasks. Other technologies are also emerging, such as the FPGA platforms, popularized by Xilinx (acquired by AMD in June 2023) and Altera (acquired by Intel in 2015 and becoming independent again in 2024). These platforms promise a greater flexibility than comparable GPGPU solutions, but for the moment implementing algorithms on FPGA remains more challenging in some cases than comparable GPGPU solutions. Apple is experimenting with its own Apple Neural Engine (ANE) solution, which is directly optimized for executing neural networks [14, 3]. Unfortunately there are very few publicly available peer reviewed sources on ANE, and even if such

resources will be accessible in the future, it is questionable how will the ANE be licensed to third party developers using non Apple products. While alternative hardware implementations remain a promising solution for possible future use, there is no indication for the moment that they would replace GPGPUs in the foreseeable future.

The use of neural networks for machine learning applications has a very rich literature [29, 33]. Authors have been experimenting with GPGPU implementations even before general frameworks, such as CUDA, became available. Authors in [2] used the OpenGL API to implement the computing of 2D spin systems on GPUs. Many major companies are maintaining machine learning frameworks, such as NVidia Deep Neural Networks[1], Intel OneApi [2], as well as many independent solutions, such as DLib [21], OpenCV [18, 17], TensorFlow [1] or PyTorch [30]. Most of these publications give little insight into the details of the particular implementation, indeed these libraries are mostly used by developers without particular interest in the specifics of the implementation. In this article we will give a details insight into the challenges of parallel implementations as well as the strategies to overcome these challenges.

3. The Implemented Algorithm

The Multi-Layer Perceptron is a type of *non-parametric classifier*, which does not assume any particular distribution of the samples within a class. Instead, these methods approximate mapping between class labels and samples by learning from examples. Since all our training examples are labeled, these approximations can theoretically be learned by any *supervised* learning method. The final MLP response calculated by the classifier M can be summarized as:

$$r = M(\mathrm{PCA}(\mathrm{norm}(I))) \tag{1}$$

where I is the input image, norm is the normalization method, PCA is the Principal Component Analysis (discussed in Section 3.) and r is a calculated response. We use PCA as specific example of unsupervised machine learning algorithm, but it could be substituted with any linear unsupervised method, such as Canonical Correlation Analysis, linear Support Vector Machine, etc. In this section we are going to describe the essence of the Multi-Layer Perceptron universal approximator and classifier algorithm. From the algorithmic point-of-view we are going to emphasize the simulation part of the algorithm, as opposed to the learning part, which is less sensitive to optimization as it can be carried out in advance and offline.

Principal Component Analysis. PCA is a common unsupervised classification algorithm preprocessing technique, which is often applied during classification. During the PCA, we would like to "de-correlate" the data, that is to say transform the linearly correlated training data into possibly uncorrelated data. Formally, we would like to find an orthonormal transformation matrix, such that the first component of the transformed dataset has the highest possible variance, and generally for any component $q[k]$, $\mathbb{V}(q[k]) \geq \mathbb{V}(q[l])\forall l \geq k$. We call this space the *principal component space*. Here we emphasize that we could substitute any linear transformation without loss of efficiency.

The orthonormal transformation has the advantage of being linear and invertible, so there is no data loss. We can, however, extend the methodology to provide lossy compression in order to impose diversity of the training data. Let us assume the following: using the optimal transformation matrix $\mathbf{Y}$, for some component k, the variance $\mathbb{V}(q[k])$ is low. We can say, that the k-th component does not tell us anything about the training data; we cannot discriminate the class of the classifier based on $q[k]$ since all $q[k]$-s are "similar". We can define a threshold v_t and we can ignore the components which together account for less than v_t percent of the total variation of the training data. This also gives us an estimate as to the dimensionality of the data.

Such transformation has double benefits: it can accelerate the calculation as we only need to calculate the first n components, and it can decrease the size of the necessary learning set as we eliminate much of the redundancy found in the training data. $\mathbf{Y}$ can be determined from the training set, and an element $\bar{\mathbf{x}}$ can be transformed as $\mathbf{Y} \cdot \bar{\mathbf{x}}$ both for during the training and the simulation.

3.1. Multi-Layer Perceptron (MLP)

The MLP is a feed-forward artificial neural network which carries out the approximation of a classifier, denoted by M [13]. The neural network is a directed weighted graph $G(V,E,w)$, where the neurons $v \in V$ use an activator function Φ[3]. The response of a neuron v is calculated as the value of the activator function with the linear combination of incoming connections (edges). We note the connections from v_i to v_k by $v_i \to v_j$. The response of a neuron can then be expressed as:

$$y(v) = \Phi\left(\sum_{\substack{x \in V \\ x \to v \in E}} w(x \to v) y(x)\right) \tag{2}$$

The model of a neural network can be observed on Fig. 1. A typical neural network consists of an input layer, a hidden layer and an output layer. The neurons between each consecutive layers are fully connected:

$$V = L_i \cup L_h \cup L_o \tag{3}$$

$$E = \left(\bigcup_{\substack{\forall i \in L_i \\ \forall h \in L_h}} i \to h\right) \bigcup \left(\bigcup_{\substack{\forall h \in L_h \\ \forall o \in L_o}} h \to o\right) \tag{4}$$

[1] https://developer.nvidia.com/cudnn

[2] https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html#gs.9idrnd

[3] Biologically inspired, the activator function represents the excitement of the "neuron" in question. In it's simplest form Φ is a binary function taking two values $\{0,1\}$. In practice we use nonlinear sigmoid functions such as $\Phi(x) = \exp\left(-\frac{||x-c||}{2\sigma^2}\right)$ or $\Phi(x) = (||x-c||^2 + a^2)^{\pm\frac{1}{2}}$. In these functions a, c and σ are parameters. The sigmoid activator functions permit us to approximate functions from a broader family than a binary function would otherwise allow. The activator functions are choosen as C^∞ functions with a smooth threshold.

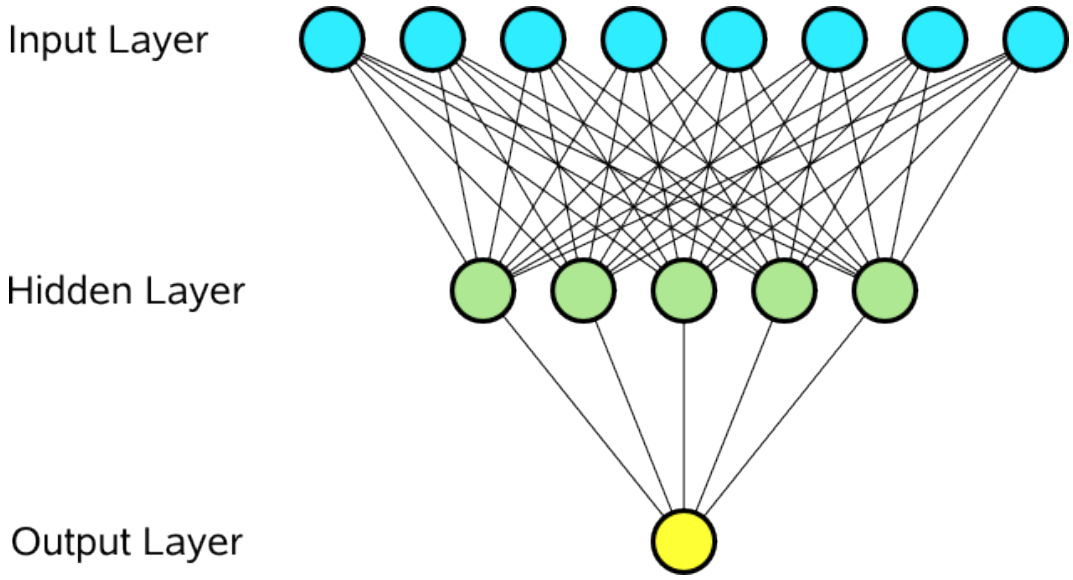

Figure 1: An example of a feed-forward neural network. The input data is presented on the *input layer*. This network contains a single *hidden layer*. The response of the M classifier can be acquired from the *output layer*.

ALGORITHM 1: Calculate the Response for a Single Neuron in the Network

Input: The Input Graph (`image`), the Requested Node (`v`)
Output: The Response of a Single Neuron (`result`)

```
sum = 0;
forall x ∈ G.nodes do
    if G.is_edge(x,v) then
        sum += G.get_edge(x,v).weight * neural_response(graph, x)
    end
end
result = v.phi(v.input + sum);
return result;
```

The response of this network can be calculated as:

$$M(\bar{\mathbf{x}}) = \Phi\left[\sum_{\forall h \in L_h} w(h \to o)\Phi\left(\sum_{\forall i \in L_i} w(i \to h) \cdot i\right)\right] \tag{5}$$

For a more detailed explanation on neural networks and the way to estimate the weights w, please refer to [4]. They have also proven that given that the training set can be modeled with some exponential distribution, $M(\bar{\mathbf{x}})$ can be interpreted as a probability $M(\bar{\mathbf{x}}_i) = \mathscr{P}(x_i \text{ is in the class of the classifier})$.
From the architectural point-of-view the most important part of the classifier M is the simulation mode. We suppose, that we have acquired the weights w of the network and we would like to simulate the network on unknown data. We would like to implement (5) efficiently on several hardware. For this, we can use the assumption of total connectivity, which will be discussed in Section 3.1.1.

3.1.1. Optimized CPU Classification with MLP

In this section we are going to discuss the MLP on the CPU. As described in Section 3.1, the MLP is a weighted directed class with an in-node activator function Φ. In its simplest form, the value of a neuron can be calculated as (2), the weighted sum of its incoming stimuli (algorithm 2.), with the neural response for a single node calculated by algorithm 1.
This model needs to be extended, before it can be implemented efficiently. Now we are going to consider the fact, that the graph has 3 layers and that these layers are fully-forward connected. We define an $|L_i| \times |L_h|$ size matrix `weights`, a $|L_i|$ long vector `inputs` and a $|L_h|$ long vector `output` as:

$$\texttt{weights}[q,w] := \texttt{G.get\_edge(q,w).weight} \forall q \in L_i, \forall w \in L_h \tag{6}$$

$$\texttt{inputs}[q] := \texttt{G.get\_node(q).input} \forall q \in L_i \tag{7}$$

$$\texttt{output}[q] := \texttt{G.get\_edge(q,o).weight} \forall q \in L_h, o = L_o \tag{8}$$

We note, that in (8), $|L_o| = 1$, so the edges all go into the same output node. With such consideration, the response of the output node $M(L_o[0])$ can be calculated as:

$$M(L_o[0]) = \Phi(\Phi(\texttt{input} \cdot \texttt{weights}) \cdot \texttt{output}) \tag{9}$$

With this simplification in mind, the modified neural response function for the output node is presented on Algorithm 4. The corresponding schema appears on Fig. 2. As we can see from the code, this version is already composed of simple linear operations. There is an important

ALGORITHM 2: Recursive Calculation of the Neural Response

Input: The Input Image (`image`), the Neural Network (`graph`), and the Output Node (`o`) of the Neural Network
Output: The Response Map of the Neural Network (`result`)

```
result = [];
for patch ∈ image do
    normalized_patch = normalize(patch);
    compressed_patch = PCA(normalized_patch);
    G = graph(compressed_patch);
    result[p]=neural_response( G, G.get_node(output) );
end
return result;
```

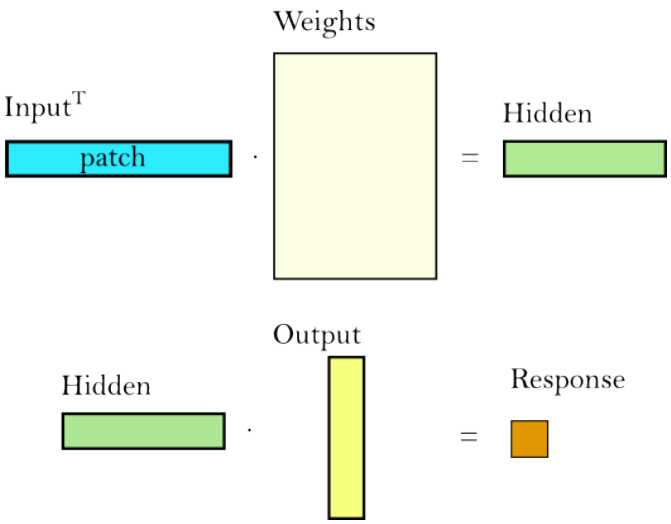

Figure 2: The schematic calculation corresponding with Algorithm 4.

part of the computer architecture design, which concerns linear algebra operations [15]. To improve the performance even further, we are going to fuse the neural response calculations, and merge all the calculation into constant number of linear algebra operations as on Fig. 3. Let us define the matrix named `inputs` as:

$$\texttt{inputs}[q,\cdot] = N(I[p[q]]), \forall p \in I \tag{10}$$

here $p[q]$ notes the q-th input. In this case we can call the `neural_response_optimized` function, from Algorithm 4 using `inputs` as parameters. As result we will get the responses for a complete set of inputs using a constant number of matrix multiplications.
This presumes that we can evaluate more inputs independently, which is often the case with classification algorithms [16]. In some cases we could also create sparse band matrices with the input matrices and weight matrices in the matrix diagonals. In both cases the complete MLP evaluations simplifies into a constant number of linear algebra operations.
In the next section we are going to discuss the necessary conditions which we have to meet in order to provide an efficient implementation of MLP on the GPU.

4. The NVidia GPU Architecture

Most of NVidia's GPU architecture description comes from its own public manuals and programming guides [27, 25, 28]. Recently [7] have published a detailed paper collecting all the architecture specific information that is publicly know about the architecture. In this section we are discussing the NVidia architecture up to the depth which is necessary to understand the code optimization carried out in MLP. The NVidia GPUs are divided into *symmetric multiprocessors* (SM), which are equivalent processing elements, sharing the same global memory. The SMs are further divided into processing elements called *CUDA cores*. CUDA cores are not fully blown independent processors and cannot execute fully independent calculations. This means, that there is a limit as to the diversity of calculations which we can carry out on the CUDA cores. The CUDA cores are grouped into *CUDA warps*, which are going to be discussed in Section 4.
Another constraint is, that in order to maximize the processing throughput, we need all the CUDA cores being occupied in every clock step. The pattern commonly referred to as "No Transistor Left Behind" [23]. In order to execute an operation at a given clock step, a CUDA core has to have all the necessary input data stored in its SM's registers. The memory accesses of NVidia GPUs can be all the way to the shared memory (cache), as opposed to the x86 processors, where the cache is managed automatically. As the memory speed is much slower than the theoretical peek calculation capacity, it is important to minimize the necessary memory throughput. In conclusion, the memory streaming and distribution also plays a crucial role in any GPU implementation. We will discuss the question further in Section 5.

WARP Execution. One of the main distinctions of the GPGPU architecture is, that it supports a high number of concurrent threads[4]. The GPU threads are much different from the POSIX threads, both by their capabilities and they relationship to each other. The OpenCL operators are called kernels. Modern GPUs support simultaneous kernels.

[4]Some publications refer to these threads as: *fibers*

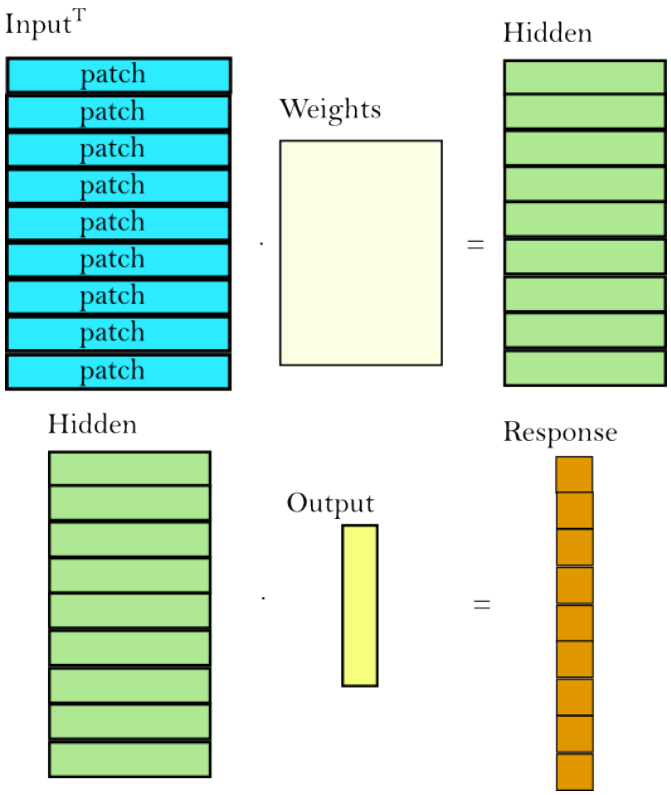

Figure 3: The schematic calculation by fusing all the inputs into a single matrix.

ALGORITHM 3: Calculating the Neural Response with Linear Algebra

Input: `input`: the input vector of the neural network, `weights`: the hidden layer weights of the neural network, `output`: the output weights, `phi`: the activator function (it is usually the same for each node)
Output: The Neural Response for the Complete Map (`result`)
`hidden` = `transpose(input)` · `weights`;
`hidden_activated` = `map( phi, hidden )`;
`response` = `hidden_activated` · `output`;
`result` = `map( phi, response )`;
return `result`;

For a comparison let's consider a typical high-end CPU system. The AMD Ryzen Threadripper PRO 7995WX contains 96 cores and supports up to 192 threads. In a four-cpu server setup the number of threads could be higher. If, however, we consider some gaming GPUs, like GeForce RTX 4090, 16384 cuda cores in 128 symmetric multiprocessors, each supporting 512 threads, giving a theoretical capacity of $128 \cdot 512 = 65530$ threads. A considerable difference, however it is obvious that the division of the calculation is much more complex and prohibitive than that of a CPU system.
The execution of the thread is tied to the SM on which it has been launched. This is an important property for the shared memory, which will be discussed in Section 4.
Further complicating the problem, the GPU threads are grouped into *CUDA warps*. Using the current technology the CUDA warps consist of 32 GPU threads. Threads of the same warp execute the same PTX instruction in *lock*, at every clock step. This makes branching even more inconvenient. If the code within a CUDA warp *diverges*, still every CUDA core executes the operation of the diverged branch. At the end of the diverged branch, the unnecessary results are discarded. As a result, all the branches have to be treated separately. In theory, if all the threads within a warp diverge, it would cause a performance to drop $32\times$.

Branching and Smallest Warps. In order to avoid the branch divergence, the programmer has to guarantee, that the threads execute the same instruction in each clock step at least within the same warp. This constraint permits, however, for the threads of separate warps to take

ALGORITHM 4: First Modified Neural Response

Input: `image`: Input Image, `normalize`: Normalization Vector, `neural_weights`: The Neural Network Weights for the Input Layer, `PCA`: The Compression Algorithm Matrix, `output`: The Neural Network Weights for the Output Layer, `phi`: The Activator Function
Output: The Response Maps Calculated from the Image (`result`)
result = [];
forall *patch* ∈ *image* **do**
    normalized_patch = normalize(patch);
    weight = neural_weights · PCA;
    result[p] = neural_response_optimized(normalized_patch, weights, output, phi);
end
return result;

separate execution paths. At this point it is still important to note that if separate warps have substantially longer execution times, then still the SMs can become idle while waiting for the execution of other groups to finish.

The Global Memory Model The SMs of the GPUs have a *global* memory accessible to them. The global memory is readable and writable by all the threads on all the SMs, but the memory writes and reads cannot be synchronized. The coherency, therefore, has to be guaranteed without global semaphores.
One of the key features of NVidia GPUs for global memory access is the memory *coalescence*. Simply speaking, global memory access prefers linear memory accesses. If the global memory is accessed sequentially by all the threads of a *half warp*, the separate memory reads can be coalesced into a single memory transaction. The linear memory access can be relaxed by using the shared memory, which does not penalize the random memory accesses. This implies a strong preference for accessing the memory in a linear way.

Shared Memory to Accelerate Calculation. Each SM contains a shared memory which is accessible by the threads executed on the SM. The shared memory has limited size, but it can be accessed in random order[5]. The shared memory is similar to the CPU cache in its distance from the processor, but it functions differently. While the content of the CPU cache is managed by the cache coherence automatically, the shared memory of the GPU has to be managed manually by the programmer. This gives the programmer full control over the content of the memory, but requires extra effort during the implementation.

5. Optimization Strategies for the MLP

One of the most important constraints of GPU programming is, that for some algorithms there is no "optimum implementation". Even on a single platform optimal implementations can differ based on the input size. There are several individual optimization strategies that we can apply based on the specific problem [6].
We can try to understand this problem based on the following example: a modified matrix multiplication problem. Let's suppose that we have n pairs of matrices $\mathbf{A}_i$, $\mathbf{B}_i$ and we are interested in the matrix product vector defined by:

$$\mathbf{C}_i := \mathbf{A}_i \cdot \mathbf{B}_i \forall i \tag{11}$$

The matrix multiplication is a basic example presented in the programming guide usually solved [26] by caching a moving patch of the matrices in the local memory of the device and cumulating the results of the calculations. This algorithm has a complexity of $O(n^3)$. There are several sequential (usually recursive) algorithms with better complexity [35, 8, 34]. The matrix multiplication is a good example of problems where the more efficient big O algorithm results in a practically slower and thus less efficient code. Further complicating the problem, the Strassen algorithms[6], is faster on the CPU then the naïve multiplication algorithm for some inputs. However, the GPU implementation of the Strassen algorithm is less efficient as it is more difficult to parallelize then the naïve implementation.
When discussing the efficiency of the moving patch matrix multiplication algorithm, the authors (sometimes silently) presuppose, that the size of the matrix is big enough to occupy the complete GPU device[7]. In the naïve implementation we create one thread for every value/pixel of the result matrix $\mathbf{C}$. If the result matrix size is usually bigger then $23 \times 23 \approx 500$ then the multiplication can saturate the GPU. However, for a really efficient implementation the size of the matrix must be around 10^6 elements. In conclusion, if in the above example we have n matrices, where the individual matrix sizes are $\approx 10^6$ we can apply the patch sliding algorithm with success. On the other hand, what happens if we have 10^6 matrices with the size of 20×20 or less? As we need at least 512 threads to saturate the GPU. We face an inefficiency when all the matrices are calculated in sequential kernel calls. We can, however, make an interesting observation: if the small inputs can fit into the GPU's shared memory, then the efficient implementation could be made by eliminating the sliding patch and calculating the separate matrices in threads instead of separate kernel calls or separate groups. This way the matrices could be calculated in a single kernel call. This kind of optimization strategy is a type of embarrassingly independent polyhedral optimization.
This is an important example, hence it demonstrates how an optimal implementation can be dependent on the size and distribution of the actual input. In this example we cannot treat the multiplication modularly, in effect we have to consider the complete problem during the optimization.

5.1. Operator Based Optimization and Real-Time Image Processing

In the case of conventional linear algebra libraries, the developers define an API[8]. The API consists of standardized *operator* calls. If we implement an optimized version of the library with the same interface, we can optimize every algorithm, which have been implemented using these API calls. In the case of the matrix multiplication, the `BLAS` API would implement the following operator:

```
def mult(matA, matB)
```
(12)

where `matA` and `matB` represent the two input matrices. We can extend this API call to include some low latency linear operators, which can be applied during the multiplication. An extended call could include these additional parameters:

```
def gemm(alpha, matA, matB, beta, matC)
```
(13)

where `alpha` and `beta` are floating point constants and the calculated expression is:

$$\mathbf{R} = \texttt{alpha} \cdot \texttt{matA} \cdot \texttt{matB} + \texttt{beta} \cdot \texttt{matC} \tag{14}$$

in such extension the intent is similar to that of a cross-component optimization, that is to reduce the memory reads and writes necessary for the calculation. This however results more complex APIs which have to address several corner cases.

[5] we do not discuss bank conflicts here
[6] with complexity $O(n^{2.8074})$
[7] Another assumption is that the matrix is "square-like" as the patch moving algorithm is less efficient on asymmetric or "thin matrices".
[8] Application Programming Interface

There are further possible optimizations, which may be related to the actual hardware. One of these optimization is the memory alignment. If the memory of the matrices is properly aligned, the memory transfer can be carried out by SIMD calls. This kind of SIMD optimization is used in [12]. In the case of memory alignment, however, the solutions are not drop-in-replacements of the original implementations, as the memory management has to be altered in order to use the optimized operators.
In rare cases a library can have several implementations of the same operators, including properly and improperly aligned cases, multi-threading and single-threading cases, SIMD enabled and SIMD disabled cases, etc. This solution was applied by [15]. This solution is however very cumbersome, as in some cases dozens of implementations are necessary for a single operator. These approaches can only be successful in fields where we have durable APIs.

Implementing MLP Using Optimized Linear Algebra Operators. Some GPU implementations, may not be suited of all input sizes. We have prepared a GPU implementation of the MLP using OpenCV's CUDA GPU module [5]. After the detailed profiling we have concluded, that the optimized general CUDA operators have not been optimal for the input sizes we used for testing. In this case it has not been possible to achieve optimal performance using pre-implemented operators.
In the case of MLP, both the input sizes and the sheer number of necessary operator calls can be prohibitive. Our test implementation consists of ≈ 10000 simple operator calls (kernel calls) for each input data record. In order to reduce the number of kernel calls we have been merging the kernels together. There are libraries which treat cases of multiple small matrices [16]. However, the implementations might still be limited to the cases which have been anticipated during the design.
In our implementation we have analyzed the detailed performance profile of our application. We could confirm from the high resolution sample that our kernel was executed sequentially, but further more, on the Stream cumulated time line we could observe the kernel scheduling penalty. The gaps between the kernel executions on the cumulated time line took up to 30μs per kernel launch compared to the 3μs of the approximate kernel execution time.
From the analysis we have concluded that final optimization of critical applications, might still require the modification or reimplementation of some existing operators.

5.2. Polyhedral Optimization

As mentioned in Section 4 the GPU threads can take separate execution paths if certain conditions are met. However, to achieve maximum performance, every thread within the same warp has to take same execution paths. If samples of a simulation can be evaluated simultaneously, then we can collect inputs of the same size and can calculate the responses for these inputs in the same OpenCL group. This polyhedral optimization idea constitutes a basic GPU optimization strategy. With this partitioning we have established the polyhedral independence, as the separate response maps are calculated using different classifiers, so they can be calculated independently.

Shared Memory Usage. As demonstrated on several CUDA and OpenCL examples the shared memory usage can accelerate the CUDA cores' access to the input for the calculation. The shared memory also does not suffer from the coalescing constraints as the global memory does. It is, therefore important to store as much of the necessary input in the shared memory of the device. There are two major challenges of the shared memory. As its size is limited compared to the global memory it is important to reduce the memory footprint of the calculation. During the the cross-component optimization in Section 5.3, we were able to reduce the memory consumption for our MLP classifier to fit into the shared memory of the device. In this case, when the complete calculation can be carried out in the shared memory, much of the *memory round-trips* can be eliminated.

Manual Memory Management. There are several limitation of the OpenCL framework compared to regular high performance programming frameworks, such as C++. One such limitation is its memory management capabilites. While we can allocate basic arrays, OpenCL does not support the allocation of multi-dimensional arrays, such as arrays of pointers. This is inconvenient and we are going to give an example of a case when such allocation is inevitable.
Let us imagine a normalization operator, where inputs of the image are processed and normalized according to some metric. If these normalized inputs are later processed by the threads of OpenCL we need to store them in the shared memory. If we need to store one sample per thread in the warp, we have to allocate `groupsize` inputs (images) possibly each of different size.
We give another example of the multi-dimensional array allocation. This example is rather convenience then necessity, however it can considerably improve the readability of the OpenCL code. OpenCL supports C-like structures, but pointers cannot be a part of OpenCL structures. If we otherwise use the object oriented programming paradigm, the data of the used classes have to be converted into OpenCL before it can be processed on the GPU. As pointers cannot remain in the structures, while converting them into OpenCL, we usually reserve a separate parameter for every memory allocation. This programmer behavior results in long function signatures, with many parameters. This code is error-prone and full of mangling pointers.
To address the issues above, we have implemented a memory allocator class, with OpenCL structure embedding and multi-dimensional array support. During the kernel call we pass a single pointer to the kernel, usually denoted as (`void * self`), and the pointers of the arrays are replaced by their first index relative to this pointer. Thus if we want to access an array of integers in the OpenCL kernel, We can do so using the following cast:

```
((__global int*)self)[ start + index ] = value                    (15)
```

where `index` is the index of the element that we are trying to access. This way we can allocate arbitrary arrays (similar to those in C) including pointers (represented by integers) embedded in the structures.
In our benchmark we have implemented two memory allocation algorithms, the buddy memory allocator and a naïve memory allocator. The naïve memory allocator is dense, whereas the buddy memory allocation supports memory release between kernel calls.
The memory allocation works as follows. In the beginning we allocate a single continuous chunk of memory. If we are targeting the shared memory of the device, the size of the allocation is 48KiB. The allocator class returns the *starting index* of the allocated array relative to the pointer of the address. The elements of the allocated array can be accessed with the (15) cast. The allocator takes care of the insufficient

memory and reports the gross allocated memory size[9]. In the case of the naïve allocator it is enough to transfer the gross allocated memory to the device. This way there is no overhead using our memory allocation compared for the `clCreateBuffer` API call.
Aside the above mentioned advantages of the manual memory allocator, there is and additional advantage which simplifies code optimization. Let us suppose that we would like to carry out a calculation and the input/output data fits into the shared memory of the device. In this case we would like to carry out the calculation in the shared memory. As the memory is passed on to the kernel as a single continuous array, the complete allocation can be copied into and from the shared memory with a single loop at the beginning of the kernel. Using this approach, the same OpenCL code can be used for shared memory (cache) calculations simply by replacing the `__global` keywords with `__local`.

5.3. Cross-Component Optimization

In order to fit into the local shared memory, we need to reduce the memory footprint of the calculations. The reduction of the memory footprint is the explicit intent of the cross-component optimization. If we refer to implementation optimization or *code optimization*, we pre-suppose that the total number of executed instructions remain the same after the optimization[10].
We are going to give an example for the cross-component optimization. Let us imagine a case, when we need to carry out a linear transformation on a normalized image:

$$\mathrm{PCA}(\mathrm{norm}(I)) \tag{16}$$

The modular approach would be implement as that on Algorithm 5. In the code there are two hidden memory round-trips. We have to store the `normalized_image`. Between lines 1 and 2 we have to write and read the image into and from the global memory. The reason is, that the shared memory is not preserved in OpenCL between kernel calls. If, however the operators are calculated in a single kernel call, and the image fits into the cache, nothing prevents us from merging the two operators into a single kernel call, eliminating thus the need for the half of the global reads and writes.

ALGORITHM 5: Modular Calculation of the Normalized Compressed Image

```
normed_image = norm(I);
compressed_image = PCA(normed_image);
```

We have applied examples as the one above to eliminate the memory round-trips and also to reduce the memory footprint of the operators. In the next section we are going to give a detailed technical description of the applied optimization steps.

MLP Cross-Component Optimization In a case of a fully connected perceptron, the transition from the input layer to the hidden layer can calculated using a matrix product. As the PCA is also a linear transformation, the compression and the first transition can be merge together into a single linear transformation. The pseudocode of this implementation can be seen on Algorithm 4.
The neural response merging is already a considerable improvement over the recursive algorithm, but we can improve it further, if we specify the normalization algorithm. Here we are going to give an example for a simple normalization, which applies a shifting and a striding factor on the image. In some cases, specifically if the inputs are overlapping, it is obvious, that storing separately every normalized patch can be extravagant. If the normalization method can be applied pixel-wise, then the normalization can be merged with the neural response calculation. More specifically, we can normalize the image during the multiplication. In the final version the normalization operator merely extracts the statistical properties of the sample and the normalization is carried out during the transition from the input layer to the hidden layer.
With these cross-component optimizations we were able to reduce the memory footprint of our application considerably. After the reduction we could carry out efficient polyhedral calculations.

5.4. Vector Processing

Some architectures support instruction level parallelism. The most popular example is the x86 architecture's SSE instruction set. The SSE instructions on x86 execute mathematical operations on 128b *SIMD vectors*. These vectors are regarded as 4 concatenated 32b values[11]. There is, of course, no technological reason for the vectors to have 128b length. As OpenCL is a general purpose framework it includes support for SIMD vectors. These vectors can range from $2\times$8b integers to $16\times$ double precision variables.
While the CUDA cores do not support native SSE instructions, there are cases where the SIMD vectorization can improve the performance considerably. One of such examples are the short 4×8b vectors. In image processing applications we often work on 8b unsigned grayscale images. During the calculation, these images are stored in packed form both in global- and the shared memory. As the shared memory banks are 32b size, the pixels of the image have to be extracted using shifting operations. If we vectorize the 8b pixels into 32b OpenCL vectors, the compiler can reduce the number of necessary shifts. We managed to vectorize the image normalization, and the result is presented in our benchmark.

5.5. Summary of the Optimization Strategies

During algorithm implementation, there are several optimization strategies which we can apply to improve the performance of the algorithm. Unfortunately, currently we cannot expect the OpenCL compilers to do this task for us. Such optimizations include: 1. polyhedrization with the intent to increase the parallelism of the calculation 2. cross-component optimization in order reduce the memory access operations and 3. vectorization in order to reduce the number of instructions executed during the computation by the device. We have applied all these optimization strategies manually to the MLP algorithm, which is a core machine learning algorithm used in artificial intelligence applications

[9] the memory size including the gaps between the memory chunks
[10] Therefore we explicitly exclude approximations with alternative algorithms, like polynomial approximation of functions or Fourier transforms used to accelerate convolutions.
[11] usually integers or floating points

today. In the next section we are going to collect the benchmarks of our implementation on several different hardware and in different stages of the development.

6. Benchmarks

During the development of the MLP, we have created five separate implementations. The implementation have been compared to each other, and each of them furnished the same results within 5 orders of magnitude. The separate implementations include:

a) The original implementation was created based on [32]. The first implementation is straight forward. Originally created in Matlabn, and later ported into C++. This implementation is presented in the benchmark, and the implementation used for the benchmark is single threaded.
b) The optimized C++ implementation was based on the observation, that some operations are proportionally faster, if they are carried out on bigger input sizes. In this implementation, the image inputs are extracted from the image and cumulated line-by-line. Finally the neural response is calculated on every sample of a response line in one operation.
c) There have been two prior GPU implementations. They have been based on the implementations *a)* and *b)* respectively. In these implementation we have replaced the CPP operators with their GPU equivalents. We have been careful to replace all the operators, to eliminate the possible PCI bus bottlenecks in both cases.
d) In the optimized GPU implementation we have applied the polyhedral and cross-component strategies discussed in Section 5. This implementation is part of the benchmark.

Performance Bottleneck. Every computer system has a finite computational performance. In practice this means that every binary calculation will take positive time. In the classical algorithmic approach of the big O performance metric we consider the computer as single "black box" which can execute an instruction in $\delta\tau$ time. It we have an estimate of $f(n)$ instructions for a given input size, then the time of the calculation will be given by $\frac{\delta\tau \cdot f(n)}{\mathscr{C}}$. In this formula $\mathscr{C}$ denotes the hardware concurrency, which in a simplistic model denotes the number of CPU cores in the system. In practice, however, this estimation will not give us accurate results, and the actual performance can be different by orders of magnitudes. In some cases the performance will not increase with the increasing hardware concurrency. To estimate the performance of a modern hardware we consider several technical aspects. Such aspects can be: 1. the number of executable instructions per second 2. local memory bandwidth 3. local memory bank 4. global memory bandwidth 5. global semaphore latency, etc. From the implementation point of view we can say that an algorithm saturates some technical aspects of the hardware. That aspect becomes a *bottleneck* for the algorithm. If we want to improve the performance of the algorithm on the given hardware, we have to address these bottlenecks in the order of their severity.
As we have mentioned in Section 4 the global memory had been copied into the local memory in a single dense chunk. In the experiment referred to as *Polyhedral Cache Memtransfer* we only measured the cache fill-up time without doing the actual calculation. From this we could measure that the kernels have spent $\frac{0.100143s}{0.33538s} = 29.9\%$ of their time in actual memory copy. If we would omit the memory copy time in the benchmark we would get at a theoretical performance of $0.33538s - 0.100143s \approx 425.1\text{sps}$[12].
It is interesting to consider the improvement by vectorization. By comparing the vectorized version to the non vectorized implementation, we could get a $\frac{0.44331s}{0.33538s} = 32.3\%$ improvement over the total calculation, which would translate to $\frac{0.44331s - 0.100143s}{0.33538s - 0.100143s} = 45.88\%$ of computation improvement, if we would disregard the overhead of the memory transfer[13]. The vectorization is thus a very important code optimization strategy.

7. Results

We present the results in a normalized manner. To benchmark the speed of execution, we use the samples per second measure, which represents the individual samples which have been evaluated by the algorithm. The main benchmarks can be seen in Table 4. Finally in table 5. we present some profiling information collected with the NVidia profiler. Here we can see, that the speed of the implementation can be up to orders of magnitude faster depending of the implementation. In our particular case we have achieved approximately three times improvement compared to the CPU implementation. It has been reported by other authors, that the GPU can generally provide up to ten times speed improvement compared to a reasonably chosen CPU, considering algorithms which can be implemented in parallel. This further justifies the trend that GPGPU has an important role in computer science and that in the future we will have to continue consider heterogenous architectures in our algorithmic efforts.

Conclusion

During this study we have been considering a general machine learning algorithm, called the Multi-Layer Perceptron. After analyzing and profiling the line-by-line GPU implementations we have analyzed the polyhedricity of MLP and proposed a fully polyhedric GPU implementation. During this implementation we proposed a method for allocating multi-dimensional vectors and embedding pointers in OpenCL structures. This feature is missing from the current OpenCL framework. We have compared the optimized GPU implementation with a highly optimized CPU implementation of the MLP. After benchmarking, we have concluded that the performance of the algorithm have increased considerably.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner,

[12] samples per second
[13] from the global memory of the device to the shared memory of the device

Implementation	Compiler	Samples Per Second	Reference Time
Simple Sequential MLP	Intel Compiler	**0.91sps**	109.714s
Optimized Sequential MLP	Intel Compiler	**15.33sps**	6.52295s
Optimized Parallel MLP (16 threads)	Intel Compiler	**96.47sps**	1.03658s
Polyhedral Cache Benchmark (groupsize 256)	NVidia OpenCL	**998.57sps**	0.100143s
Polyhedral Cache Benchmark (groupsize 448)	NVidia OpenCL	**1074.70sps**	0.093049s
Polyhedral Global Mem. MLP	NVidia OpenCL	**88.40sps**	1.13117s
Polyhedral Cache MLP (groupsize 256) No Vectorization	NVidia OpenCL	**225.58sps**	0.44331s
Polyhedral Cache MLP (groupsize 256)	NVidia OpenCL	**298.17sps**	0.33538s

Figure 4: The measured execution times.

Metric	Average	Maximum
Duration	3.237ms	5.803ms
Achieved Occupancy	0.3	0.3
Warp Execution Efficiency	87.2%	97.6%
Global Memory Load Efficiency	100%	100 %
Branch Efficiency	97.7%	98%
Multiprocessor Efficiency	89.8%	90.8%
Instructions Executed	42 320 964	74 032 826
Shared Bank Conflicts	1 076 110	1 154 128
Threads Launched	12 544	12 544

Figure 5: Metrics Collected by the NVidia Profiler for the Polyhedral MLP Cache Implementation

Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
[2] Viola Anselmi, Giovanni Conti, and Francesco Di Renzo. Gpu computing for 2-d spin systems: Cuda vs opengl, 2008.
[3] Apple. Deploying transformers on the apple neural engine. *Apple*.
[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2006.
[5] Gary Bradskim and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media, 2nd edition, September 2013.
[6] André R. Brodtkorb, Trond R. Hagen, and Martin Saetra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73 (2013)(1):4–13, January 2013.
[7] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (gpu) programming strategies and trends in {GPU} computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013. Metaheuristics on GPUs.
[8] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. Computational algebraic complexity editorial.
[9] NVIDIA Corporation. Nvidia gh200 grace hopper superchip architecture. Technical report, NVidia, 2024.
[10] Paresh Dave. Nvidia chip shortages leave ai startups scrambling for computing power. *WIRED*.
[11] Erin Griffith. The desperate hunt for the a.i. boom's most indispensable prize. *The New York Times*.
[12] Gaël Guennebaud and Benoit Jacob. Eigen, May 2013. Presentation.
[13] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.
[14] Matthijs Hollemans. The neural engine — what do we know about it? *Github*.
[15] Intel(tm) Corporation. Math kernel library.
[16] Intel(tm) Corporation. *Intel(tm) Integrated Performance Primitives Reference Manual, Volume 3: Small Matrices and Realistic Rendering*, 319375-022us edition, 2012.
[17] Itseez. *The OpenCV Reference Manual*, 2.4.9.0 edition, April 2014.
[18] Itseez. Open source computer vision library. https://github.com/itseez/opencv, 2015.
[19] Amand Joshi. Surprise, surprise! nvidia owns two-thirds of the data center ai chip market and 97 *Linkedin*.
[20] Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. 1(1):1867–1874, 2014.
[21] Davis E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.
[22] Davis E. King. Max-margin object detection. *CoRR*, abs/1502.00046, 2015.
[23] Raja Koduri. No transistor left behind, 8 2020. At Hot Chips 2020, Raja Koduri, senior vice president, chief architect and general manager of Architecture, Graphics and Software at Intel, delivered a keynote presentation.
[24] Raúl Nozal and José Luis Bosque. Exploiting co-execution with oneapi: heterogeneity from a modern perspective. *CoRR*, abs/2106.01726, 2021.
[25] NVidia. *OpenCL Programming Guide for the CUDA Architecture*, 2.3 edition, August 2009.
[26] NVidia. *OpenCL Best Practices Guide*, February 2011.
[27] NVidia. *CUDA C Best Practices Guide*, dg-05603-001-v5.0 edition, October 2012.
[28] NVidia. *CUDA C Programming Guide*, pg-02829-001-v5.0 edition, October 2012.
[29] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.
[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.
[31] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.
[32] Jason M. Saragih, Simon Lucey, Simon, and Jeffrey F. Cohn. Deformable model fitting by regularized landmark mean-shift. *International Journal of Computer Vision*, 91(2):200–215, 2011.
[33] Tristan Udby and Yun Tian. A generic neural network implementation on gpu and its performance benchmark. In Kohei Arai, editor, *Proceedings of the Future Technologies Conference (FTC) 2022, Volume 3*, pages 138–154, Cham, 2023. Springer International Publishing.
[34] Virginia Vassilevska Williams. Breaking the Coppersmith-Winograd barrier. 2011.
[35] Strassen Volker. Gaussian elimination is not optimal. *Numerical Mathematics*, 13(4):354–356, 1969.