Massive Parallelization of HOG-based Algorithms for Object Detection

Darius Malysiak

Massive Parallelization of HOG-based Algorithms for Object Detection

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

December 13, 2016

Darius Malysiak

Geboren am 21.01.1983 in Chorzow

Berichter: Dr. Rolf P. Würtz
Prof. Dr.-Ing. Diana Göhringer
Prof. Dr.-Ing. Uwe Handmann

Mündliche Prüfung: 9.November 2016

Contents

Chapter

Abstract

This thesis addresses the question of how the HOG algorithm can be accelerated on massively parallel architectures. It introduces a formal framework of so called hypersystems which provide an abstract interface for actual computation systems. Its applicability is demonstrated by utilizing it for GPU accelerated mean shift computation, distributed training of small scale neural networks and distributed matrix multiplication. On the basis of this framework I present a concept for boosting existing HOG algorithms; in terms of reliability and computational efficiency. This concept provides the means for an efficient distribution of HOG algorithms in heterogenous computation systems. Furthermore I provide a software framework for distributed computation in heterogenous cluster systems, putting a special focus on Beowulf clusters with multi-GPU nodes. All developed algorithms are thoroughly analyzed; formally and by practical evaluation.

I. Foreword

"In science if you know what you are doing you should not be doing it.
In engineering if you do not know what you are doing you should not be doing it.
Of course, you seldom, if ever, see either pure state."

Richard Hamming

Long have I been looking for a fitting quote which could adequately state one of the important aspects during the creation of this thesis. When I began studying mathematics, electrical engineering and computer science I did not so because of career decisions but for the sake of understanding and knowledge itself. For a long time, I did not understand the difference between science and engineering, only equipped with the impression of sciences superiority I struggled with my work being degraded to engineering. It was the natural human wish to construct, to built, to provide ease in life, to conquer challenges and mankind's endless curiosity which lead to inventions like the airplane, the automobile and the computer. Each invention sparked entire areas of research, which in turn lead to improvements of existing machines or entirely new creations. An endless cycle between theory, machine, theory, machine... . Only with the advent of computers could the science of operating systems spark into existence, only with the desire for more beautiful computer graphics

would high density SIMT architectures become commonly available, this in turn created entire research groups which would develop algorithms for such systems. The study of systems, learning their structure and understanding their behaviour is what we commonly refer to as science, later we use this knowledge to deduce more knowledge. Often we try to reconstruct the studied system, by studying the problems during this endeavour we obtain knowledge and use it to improve the construction methods, this we call engineering. Sometimes, these construction methods yield solutions to scientific or engineering problems in different areas. One would be a fool putting science over engineering or vice versa, as the one could not thrive without the other. So what is engineering but another form of science and what is science but another form of engineering?

In hindsight, I have to admit that my developed concept was a daring speculation until it proved to be effective through several evaluations in practical problems. The concept has its limits and drawbacks, from a theoretical as well as practical perspective. I will explain those to the best of my knowledge. I hope researchers and engineers will at least find inspiration in my ideas, as they travel the endless cycle. I wish for my work not to have been in vain, practically or theoretically.

At this point I would like to thank several people, as without them, this work would have never been possible. With no respect to any order, beginning with Uwe Handmann who I thank for providing the infrastructure which allowed me to carry out the required experiments. He always gave me freedom to follow my interests, a gesture which I was very fortunate to receive. Many thanks go to my parents, who despite living in a much smaller world, loved me to an extent which allowed them to unquestionably support my way of life. They provided me with shelter for

many years and taught me as living examples that the lack of education will, in the end, only lead to needless conflicts. Without Rolf Würtz this thesis would have not been possible at all, his lectures at Ruhr-Universität Bochum always filled me with joy and intellectual satisfaction. It was him who, back in the days of my diploma thesis, sparked my interest in neural biology and its application in computer science. My mathematical skills would not have been as strongly developed without the inspiring lectures of Gerhard Knieper, Alfons Skirde and Herold Dehling whose love for detail and didactic excellence motivated me during my studies. The exceptional beauty and purity of mathematics could not have been taught better. Three very special friends, Marcel Naujoks, Bodo-Marcus Wolff and Philip Kulik, whom I have to thank for guiding me in life, for more than 2 decades they stood by my side especially in difficult times. *Should you read this, thank you!*

A good childhood is the foundation of all things to come; my grandfather who raised me with love and strong ideals, his sense of reality, wisdom of age and faith in me, gave me the feeling of being at home. This thesis is dedicated to him.

Last but not least I would like to express my deepest thanks to Marc Jansen who supported me with constructive questions during the process of writing this thesis.

Mathematics and natural sciences can be philosophically beautiful, they can lead to deep insight of oneself; but they can not provide sympathy, they can not take away loneliness, they can not provide love or warmth. If it was not for these last things, this thesis would have never been concluded. Believing that words can never adequately express my emotions to her, I will simply thank Anna-Katharina Römhild, as without her love my writing would have come to a halt.

לא תסתכל בקנקן , אלא המב שבתוכו

(Do no look at the jar, but at what is inside)

II. Introduction and Research Questions

Recent surveys in the field of High Performance Computing (HPC), e.g. [VN14a], [Owe04], [GR14] or [OLG+07], show a clear trend towards the application of Graphics Processing Units (GPUs) and high-density multi-core Central Processing Units (CPUs). Modern super-computers like the Tianhe-2 ([Tsa13]), which utilizes Xeon Phi cards or the Titan-Cray XK7, which utilizes K20X GPUs ([Rog12]), are just the most prominent examples. Yet many research institutions and companies tend to use Beowulf cluster as an economic alternative to large state-funded cluster systems. The need for computation power has grown continuously, especially with the advent of cost effective high-resolution image sensors and the need for large video-based surveillance systems. In the context of practical applications, rapid deployment of highly performant image processing systems poses an important aspect. Yet high performance can only be achieved if the application utilizes most, if not all, intrinsic architecture elements. Designing programs for cluster systems can be a challenging task, even in case of homogeneous systems one needs detailed knowledge about the system elements, e.g. communication latencies or the structure of processing elements. Until the advent of deep neural networks ([STE13]) many state-of-the-art approaches for object detection in images utilized histograms of

oriented gradients (HOGs). The computationally expensive calculation of those descriptors quickly found its way onto modern multi-core and GPU architectures, although in scenarios with many image sources even these massively parallelized versions would swiftly reach their limits ([HMGH14]).
Several of today's state-of-the-art systems, e.g. [ZWSL14] or [LTWT14]. [FYY+15]) still continue to use HOGs as supplementary information in order to boost their performance. In other cases, the benefits of HOGs, e.g. less required training data, outweigh those of other systems such as a slightly better recognition rate with much more training data. One can see these effects in the comparison of MultiFtr+CSS and HOG in [BOHS14]). Additionally, several improvements of the classic HOG approach have been suggested ([WHY09], [NHH11]). Yet all applications of HOGs impose the same performance bottleneck onto the system (i.e. the complex descriptor computation). In this thesis I provide theoretical as well as practical methods in order to circumvent the intrinsic drawbacks of HOG-based algorithms.
Before describing the outline of this thesis I would like to elaborate on the research questions that motivated my research. They can be stated as three successive questions which build upon each other. My initial motivation was to analyze how the classic HOG algorithm can be improved in terms of efficiency through parallelization, this naturally leads to the question about how the locally optimized approach can be generalized for large scale structures, finally one may ask how this generalization could be used in order to improve more complex system setups for object detection, especially existing hog-based architectures.
This thesis is structured as follows. Chapter III provides the background on organic computing ([vdM07]) and elaborates its

relevance, to this thesis as well as to parallel computing. The algorithmic aspects of parallel computing, which are most relevant for my developed approach, are introduced within chapter IV. Chapter V delves into the corresponding aspects of implementation. In order to evaluate the concept I developed an adequate software framework for cluster computation, chapter VI introduces the basic structure of the framework. Chapter VII explains the fundamental approaches (already existing as well as newly developed) for parallelized HOG computations. It also provides a comparison between new and existing strategies. The generalized concept of so called hypersystems is introduced in chapter VIII. It features a description of the general idea, its application to neural network training and distributed matrix multiplication. The thesis concludes with a novel approach for distributed HOG computation (chapter IX) and a detection pipeline which is capable of boosting existing HOG-based systems in detection quality while still maintaining real-time capabilities. An overall conclusion is provided in chapter X, which also revisits the research questions and critically elaborates on them.

III. Organic Computing and the Need for Adaptive Massive Parallel Systems

Organic computing describes a research field which aims at tackling problems of complex systems with methods observed in nature. The primary focus lies on the way how natural systems develop by means of self organization, organic computing pursuits the goal of developing a unified concept which gives rise to evolving and self-adaptive systems for given problems, similar to ontogenesis of organisms in biology.
A good summary of an emerging engineering problem is given by [AMO04] who expresses it through "*The unbelievable growth in the complexity of computer systems poses a difficult challenge on system design. To cope with these problems, new methodologies are needed that allow the reuse of existing designs in a hierarchical manner, and at the same time let the designer work on the highest possible abstraction level*". In the field of engineering one quickly encounters problems which result from rather fixed structures, be it the usage of a restricted computation platform, an overwhelming amount of model parameters or the debugging overhead of a software which becomes increasingly complicated. Using organic computing one would present the problem to a system which then would adapt itself by means of exploration

and evolution. As [vdM07] states, this approach represents an inversion of the classic paradigm, in which one develops software, analyzes the output, proceeds with debugging and continues with further iterations until the program outputs the desired result. Organic computing could lead to more optimal solutions as it would explore adequate alternatives and optimize the resulting parameters. One would only need to define a global system behaviour, i.e. define a conceptual context.
As we will see in chapter V a computing system can become so vastly complicated that developing efficient algorithms becomes an increasingly difficult and error prone task. The resulting algorithms become large and are often deeply coupled with the existing hardware architecture, which additionally makes it difficult to adapt them for other platforms. Observing the validity of Moore's law one will recognize its stagnation over time, the increasing difficulty of shrinking integrated circuits will eventually invalidate it ([HH10]). Following the eve of increasingly powerful single processing elements comes the dawn of massive amounts of simple (less powerful) homogeneous processing units. This is also marked by [vdM07] who identifies economical reasons for the collapse of Moore's prediction, he additionally agrees on the possibility of nanoscale computation with large numbers of computation units. Yet in order to harness the potential of such systems one has to embrace the inability of full system prediction, let alone, control. Even today one has to carefully design algorithms for systems such as GPUs or multicore CPUs, aside of the developers best intentions he must delegate a lot of faith into compiler optimizations. This is necessary since a detailed study of each applicable architecture, and the resulting algorithm optimization, would induce an impossible to mange overhead for developers and companies. In order to effectively manage such vast

and different systems one has to accept the loss of insight, which otherwise would become the bottleneck for pushing algorithmic efficiency and development progress ([vdM07], [CPJK04]). Parallel systems must be self-adaptive, at least regarding their optimal parameters, in the context of organic computing, they are also required to be self-adaptive with respect to their algorithmic structure.

My approach was motivated by exactly the same ideas as in organic computing, yet I did not aim for a fully adaptive system, i.e. a system which algorithmically adapts itself. Attempts in this direction have already been made in [MSSU11] who describes an organic processing cell. The same holds for [ABT04] who applies the intrinsic motivation of organic computing to self-configuring Field Programmable Gate Arrays (FPGAs). Many more examples exist, e.g. [WSR04], [BKS04], [Buc02], in which adaptivity and modularity emphasizes the goals of organic computing. Most of my explicitly stated algorithms are designed for a specific kind of computation architecture, e.g. GPUs with a SIMT concept. My solution is not fully recursive or in other words; it does not optimize itself on the algorithmic level i.e. it does not change its algorithmic behaviour or communication protocol. Yet it features self-adaptation on a parametric level for given algorithms, introducing additional abstraction layers one could most likely extend its expressive power and reach a more generic self-adaptivity. Following [vdM07] one can call my developed system a small universe which scratches the greater goal of organic computing.

"*An illustrative case in point concerns heterogeneous parallel programming. It is notoriously difficult to know the actual execution times of programming steps*" [vdM07], this statement perfectly states what kind of problem is address with may approach. In-

stead of creating models or following existing paradigms for designing algorithms I formally describe a self-adaptive method for multi-tier heterogeneous computation systems and analyze its benefits and drawbacks. I address the "symptom" of adaptive and efficient workload distribution and do not claim to provide much advance in order to reach the goal of organic computing. Yet, I show the potential of organic computing by applying similar thoughts in a practical application, i.e. object detection with hog-based algorithms.

IV. Paradigms in Parallel Programming

This chapter introduces several important concepts of parallel programming, discusses existing system structures and induced challenges.

IV.1. Algorithmic Parallelization

Parallelization in algorithmic tasks describes the attempt to split the workload into multiple segments which can be executed independently. This is not always the case as shown in the inherently sequential Alg. 1

Algorithm 1 LCG

Input: Seed s
Output: s

1: $s = 0$;
2: **for** $i = 0$ to 31 **do**
3: $\quad s = 4 * s + 3 \mod 25$;
4: **end for**

Each iteration depends on the result of the previous, i.e. the iterations depend onto each other. An example for a paralleliz-

able task is the simple task of summing up values v_i, which are contained in an array V.

Algorithm 2 Sum up values v1

Input: Array V
Output: s

```
1: s = 0;
2: for all v_i ∈ V do
3:     s = s + v_i;
4: end for
```

The iterations are independent, i.e. no iteration requires the result of a previous or later iteration. This can be exploited; let us assume n processing units are available, that n divides $|V|$ (i.e. $n||V|$) and let $V = \biguplus_{j=1}^{n} V_j$

Algorithm 3 Sum up values v2

Input: Array V
Output: s

```
1: (G_s) s = 0;
2: (P_j)j =getPUIdx(); s_j = 0
3: par. for _j all v_i ∈ V_j do
4:     s_j = s_j + v_i;
5: end par. for
6: (G){
7: for j = 1 to n do
8:     s = s + s_j;
9: end for
10: }
```

The operation *getPUIdx* returns the index of the calling execution unit, since this method is called locally by each unit the local variable j will be used as an index for the partial sums. Alg. 3 also introduces the pseudocode notion $(G_S\{...\})$, which indicates in line 1 the declaration of a global variable, i.e. a variable which is available to all execution units, this corresponding initialization will be executed by a single execution unit. In line 6 it indicates that only a single execution unit will process the for-loop. Furthermore the listing shows a parallel for-loop, with an execution unit local variable j, such a loop will be equally splitted among the n units. As a complement to "(G)" the listing introduces the notion for an instruction which will be executed by in parallel, such instructions or segments will be marked by $(P_S\{...\})$, similarly the index j indicates that j is a local variable for each involved execution unit.
Although one can split the processing equally among the available units, each of those units computes a partial result, which in the end must be merged in order to obtain the final result. Depending on the type of merge operation this may induce an overhead which outweighs the gain of the previous computation. Amdahl's law states

$$S(s) = \frac{1}{1 - p - p/s} \tag{IV.1}$$

with S being the speedup of the entire algorithm, s the speedup of the task and p the percentage of processing time which the parallelizable algorithm part took before being distributed onto multiple execution units. Since p and s are considered to be fixed units here one has to refer to Gustafson's law when expanding a system with increasing problem sizes

$$S(s) = 1 - p + sp \tag{IV.2}$$

The effect which motivated John Gustafson will become visible in later chapters where I discuss the developed algorithms in the context of increasing problem sizes.
Between lines 5 and 6 one has to wait for all execution units to finish processing, this step is usually referred to as unit synchronization. Depending on the architecture, e.g. a distributed system with large latencies, this task may become an efficiency bottleneck. Such situations, in which the communication overhead outweighs the computation benefit is commonly referred to as parallel slowdown. In chapter VIII I show that, depending on the problem such an effect can be induced when moving from a single computation node to multitier architectures.

IV.2. The Problem of Asserting Complexities

An algorithm has two important aspects; its correctness, i.e. its capability of producing the correct / expected result, and the time it needs in order to produce that output. The so called Landau notation represents a set of several asymptotic complexity measures $\mathcal{O}, o, \Omega, \omega, \theta$, which are defined as

Definition 1. *Let* $g, f : \mathbb{R} \to \mathbb{R}$

$$f = \mathcal{O}(g) \quad :\Leftrightarrow \quad \exists M > 0, x_0 \in \mathbb{R} \forall x \geq x_0 : |f(x)| \leq M|g(x)| \tag{IV.3}$$

$$f = o(g) \quad :\Leftrightarrow \quad \forall M > 0 \exists x_0 \in \mathbb{R} \forall x \geq x_0 : |f(x)| \leq M|g(x)| \tag{IV.4}$$

$$f = \Omega(g) \quad :\Leftrightarrow \quad \exists M > 0, x_0 \in \mathbb{R} \forall x \geq x_0 : |f(x)| \geq M|g(x)| \tag{IV.5}$$

$$f = \omega(g) \quad :\Leftrightarrow \quad \forall M > 0 \exists x_0 \in \mathbb{R} \forall x \geq x_0 : |f(x)| \geq M|g(x)| \tag{IV.6}$$

$$f = \theta(g) \quad :\Leftrightarrow \quad (f = \mathcal{O}(g) \wedge f = \Omega(g)) \tag{IV.7}$$

Analogously we define

$$\mathcal{O}(g) := \{f | f = \mathcal{O}(g)\} \tag{IV.8}$$
$$o(g) := \{f | f = o(g)\} \tag{IV.9}$$
$$\Omega(g) := \{f | f = \Omega(g)\} \tag{IV.10}$$
$$\omega(g) := \{f | f = \omega(g)\} \tag{IV.11}$$
$$\theta(g) := \{f | f = \theta(g)\} \tag{IV.12}$$

The functions input parameter x is considered to be some size parameter to the algorithm, the output $f(x)$ represents arbitrary time units, machine steps or memory units. In case of the previous example (Alg. 2) one can fix the parameter to be the size of V (i.e. $|V| =: m$), thus we obtain $f(m) = m$ and the complexity function $g(m) = m$, one can also say its complexity is $\mathcal{O}(m)$. Since $\mathcal{O}(m) \subseteq \mathcal{O}(m^2) \subseteq ...$ one usually states the set with the slowest growing function.
This method has only limited expressive power when it comes to the algorithm's application, in order to facilitate this let us compare the following algorithm with Alg. 2

Algorithm 4 Sum up values v3

Input: Array V
Output: s

```
1: s = 0;
2: for all v_i ∈ V do
3:     s = s + v_i;
4: end for
5: s = 0;
6: for all v_i ∈ V do
7:     s = s + v_i;
8: end for
```

Alg. 4 obviously produces the same result as the previous one, its complexity is still $\mathcal{O}(m)$ although it executes twice the amount of iterations. Thus one would prefer the first algorithm in the practical application, yet one can not make that decision by simply comparing the complexities. To put it in other words, the Landau notation omits constant factors within the complexity functions. In order to accommodate this drawback we will state all significant factors during the analysis of my developed algorithms. Regarding the previous comparison we would state the complexities of Alg. 2 and Alg. 4 as m and $2m$, respectively. This notation is practicable for theoretical applications yet unapplicable in the context of engineering. Since, depending on the time unit, it can make a huge difference if the algorithm requires 1 or 2 hours for execution.
Yet there is another important element in the analysis of algorithms in general; the computation model. Modern computation architectures feature vector processing units which are capable of executing multiple instructions at once. This in turn will re-

sult in different constants, e.g. a vector processing unit (single instruction multiple data, SIMD [Fly72]) could execute the summation of k elements from V (which would result in complexity expression $\frac{1}{k}m$ and $\frac{2}{k}m$). Authors who address this aspect usually choose an abstract computation model for their analysis, e.g. the Parallel Random Access Machine (PRAM) [RR10] with SIMD as a special case, the Bulk Synchronous Parallel Computer model (BSP) [Val90] or the LogP model [CKP+93]. One can always argue if the chosen model truly suits the requirements or the practical application, thus instead of choosing an abstract computation model I designed and analyzed the algorithms in the context of a specific architecture with the generic approach from the PRAM model. Using this strategy one can identify intrinsic parameters, which can be optimized through measurements of the real system.

IV.3. Parallel Reduction

An important technique for summing up (not necessarily the canonical addition) elements from an array is the so called parallel reduction. Getting back to the previous example let us assume the size of V as well as n is a power of 2, i.e. $|V| = m = 2^k, n = 2^l$, furthermore let (for now) $n = 1/2 * |V|$ and $\circledast$ be the summing up operation. The following algorithm shows the basic strategy

Algorithm 5 Sum up values par.-reduction

Input: Array V
Output: $V[0]$

```
1: (P_j)j = 2*getPUIdx();
2: (P_i){
3: for i = 1; i < m; i = 2 * i do
4:     if j mod (2 * i) == 0 then
5:         V[j/2] = V[j/2] ⊛ V[j/2 + i];
6:     end if
7:     SYNC UNITS
8: end for
9: }
```

Again we introduce a new concept, line 8 shows the synchronization call for all units, a unit will hold its execution until all other units reached that point. The basic idea is to maximize the number of utilized execution units, within the first loop iteration all units will participate, i.e. they will enter the if-clause, thus each unit will sum up two elements from V, e.g. unit 0 will sum $V[0], V[1]$ and store the result in $V[0]$. During the second iteration unit 0 will add $V[0], V[2]$ and again store the result in $V[0]$, yet only half of the available units will enter the if-clause. The final result will be stored in $V[0]$ and since Alg. 5 operates in-memory the contents of V will be modified. This leads to

Theorem 1. *The parallel reduction as stated in Alg. 5 has a runtime complexity of* $\mathcal{O}(\log(n) + f)$ *with* $f \in \mathcal{O}_{\circledast}$ *and* $\mathcal{O}_{\circledast}$ *being the complexity class of the* $\circledast$ *operation.*

Proof. The for-loop will be executed by all units in parallel, there are a total of $\log(m)$ iterations, thus the step complex-

ity is $\mathcal{O}(\log(m))$. In iteration i a total of $\frac{m}{2^i}$ operations will be executed in parallel, thus we obtain $\sum_{i=1}^{k} 2^{k-i} = m - 1$, which results in a work complexity of $\mathcal{O}(m)$, which in turn is work efficient. Following the proof of Brent's theorem ([Bre74]) the total complexity results in $\mathcal{O}(\frac{m}{n} f + \log(n))$. With the assumption that $n = m/2$ one obtains $\mathcal{O}(\log(n) + f)$. □

Depending on the formulation of the parallel reduction the runtime may change in its expression, yet the corresponding proof follows the same strategy as above.

V. GPU Architectures for General Purpose Computation

V.1. OpenCL and the Diversity of Hardware

Over the past 10 years, motivated by economical and practical reasons, a large variety of different hardware architectures for parallel computation sparked into existence. Each of them, e.g. ARM Mali, NVidia Kepler, NVidia Maxwell, AMD Tahiti, AMD Hawaii, Intel Core i7, just to name a few architecture *families*, exhibited different system parameters. In heterogeneous environments it would quickly become an obstacle to maintain an infrastructure of different programming interfaces and device specific optimizations. The OpenCL credo towards this challenge consists of two elements:

- Giving up insight to the system specific attributes.
- Providing an abstraction layer for applications.

I will only briefly describe the most relevant aspects of the abstraction layer, for more details one should refer to [SGS10] since

the languages' complete syntax or semantics can not be introduced in a few sentences. Yet, most of the OpenCL listings should be self-explanatory. OpenCL groups single or atomic processing units into sets, which are called *compute units*, a host system may contain multiple so called *compute devices*, each equipped with multiple compute units. The nomenclature of OpenCL refers to each single processing unit (i.e. each element in a compute unit) as a *processing element* (see Fig. V.1). Memory is distinguished by 3 categories, namely global, shared and local memory, on which the following section will elaborate. One should note that the actual computation system may be vastly different from the OpenCL model. Data exchange or communication in general is handled via global memory or zero-copy access (i.e. the compute device accesses the hosts' memory directly).
Each vendor who supports the OpenCL standard provides a corresponding compiler, e.g. a standalone executable binary or program library. A program (OpenCL *kernel*) written in OpenCL will be compiled into the format of the corresponding machine. All device specific optimizations *can* be delegated towards the compiler, which is common since many devices exhibit a complex structure which must be studied in order to harness its effectiveness. Thus one gives up the insight into the machine (for reasons identical to those stated in [vdM07, Ch. 2]).
My approach (Chapter VIII) essentially follows in OpenCL's footsteps, yet I approach the challenge from a different flank, i.e. from the context of multi-tier clusters systems.

V.2. Practical Challenges

During the design and implementation of certain algorithms, e.g. those to be executed on Single Instruction Multiple Data (SIMD) devices, one has to consider various intrinsic OpenCL aspects, which will be described in the following sections.

V.2.1. Choosing the Thread Grid Structure

The OpenCL model follows the SIMD paradigm i.e. it provides a device abstraction in which many threads $t_{x,y,z}$ execute the same instruction but on different data. A program can be executed by a number of threads which exceeds the amount of available processing elements, the required scheduling is device specific and delegated to the vendors OpenCL implementation. In the runtime context threads are grouped in so called workgroups $W_{i,j}$, whose maximal size is also device specific (the same holds for the maximal amount of workgroups). All workgroups are of identical size and exhibit a multidimensional grid structure. As mentioned before the workgroups themself are scheduling units of threads which are organized in a multidimensional so called thread grid G(see Fig. V.1). This thread grid groups the to-be-scheduled threads in thread blocks $T_{i,j}$.

The dimensionality itself is usually motivated by the data's structure, e.g. image processing tasks usually involve 2 dimensional grids for convenience reasons. Yet every multidimensional algorithm can be transformed into a 1 dimensional version. The most important aspect are the number of elements in a workgroup and the total amount of threads, which provide the means to mask memory latencies. In order to understand this, the physical realization of workgroups must be understood. All threads in a workgroup may be executed in lockstep, e.g. on GPUs,

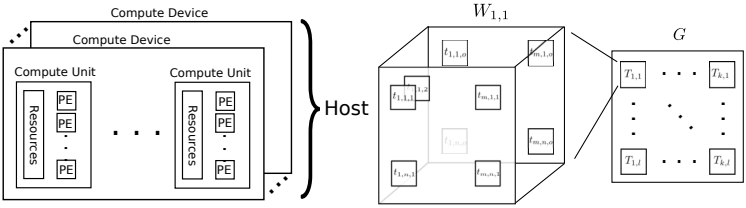


Figure V.1.: The left image depicts the OpenCL computation model, the right one illustrates a 2 dimensional OpenCL grid with 3 dimensional workgroups

or independent in parallel, e.g. CPUs. All workgroup threads can exchange data via workgroup local memory, which is called shared memory, additionally each thread has a certain amount of private memory, which is called local memory. Regarding best access latencies, highest bandwidth and smallest size the memory is grouped hierarchically as local→shared→global, in descending order. Local memory can exhibit latencies of ≈ 150 clock cycles while global memory can reach ≈ 700 cycles [MC15],[MZLC14]. Yet since the actual device may provide arbitrary memory types to the OpenCL model, it may occur that shared and global memory exhibit the same latency, e.g. in the case of Intel CPUs. During runtime each workgroup is executed by a single compute unit, the active workgroup is chosen from a set of waiting workgroups, which are referred to as inbound groups. The strategy behind masking of memory latencies consists of scheduling another inbound workgroup for execution once the previous group issued a memory request. In other words the time for memory accesses is used for processing instructions from other workgroups. Thus one is usually motivated to start as many threads as possible,

sadly there exists a counter effect to this strategy; scheduling overhead. The exchange of an active workgroup on a compute unit induces a memory swap, i.e. all registers of a compute unit will be saved in order to restore them later on, i.e. once the unit is ready to continue. Additionally the limited amount of swap memory limits the number of inbound units, this becomes apparent when workgroups allocate large amounts of shared memory, which can significantly decrease the number of units.

V.2.2. Thread Divergence

As stated in the previous section OpenCL programs can be executed in lockstep by all threads in a workgroup, i.e. all threads will execute the same instruction simultaneously. Yet this raises the question about how Alg. 6 is actually executed in the case of a thread grid with a workgroup of 256 threads (assuming one 1 compute unit with 256 processing elements).

Algorithm 6 Thread Divergence

```
1: i =get_global_id(0);   → get global index of PE
2: if i < 10 then
3:     ...; (a)
4: else
5:     ...; (b)
6: end if
```

Obviously all threads will be able to execute the algorithm in lockstep up to the first line, yet depending on the threads' id it will enter either the if or else segment. A semi lockstep execution can be assured if the device first executes all threads in

the if segment, letting all other threads execute nop-instructions in parallel and afterwards execute all threads in the else-branch with nop-instructions for the remaining threads. This will result in underutilization of available resources since the if-branch will utilize only 10 of the 256 processing elements, while the else-branch will use 246 of 256. Additionally the described semi-lockstep execution principle is equivalent with two successive program executions, one executes (a) and the other (b). This phenomenon is referred to as *thread divergence*, in this context the algorithm should be designed with as few branch elements as possible. Common strategies consist of

- Splitting the computation through skillful indexing in a way that only one branch will be taken by all threads in a workgroup.

- Eliminating branches by segmenting the computation steps into multiple algorithms.

V.2.3. Barrier Divergence

Another challenge is posed by *barrier divergence*, which is easily demonstrated with Alg. 7

Algorithm 7 Barrier Divergence

```
1: i =get_global_id(0);   → get global index of PE
2: if i < 10 then
3:     ...; (a)
4:     barrier(CLK_GLOBAL_MEM_FENCE);
5:     ...; (b)
6: else
7:     ...; (c)
8:     barrier(CLK_GLOBAL_MEM_FENCE);
9: end if
```

The barrier command is a synchronization call, a thread will wait at this call until all other threads reached the same position. According to the OpenCL specification all threads in workgroup must reach all barrier commands, the behaviour is undefined should this not be the case. Alg. 7 shows the problem, the algorithm will only end in defined behaviour if all threads enter both barrier calls, yet the branching will prohibit that. In order to solve that one has to introduce another branch segment (see Alg. 8) since the if-branch contains another instruction after the synchronization. Yet this will in turn increase the thread divergence and thus decrease the algorithm's efficiency.

Algorithm 8 Barrier Divergence - Solution

```
1: i =get_global_id(0);   → get global index of PE
2: if i < 10 then
3:     ...; (a)
4: else
5:     ...; (c)
6: end if
7: barrier(CLK_GLOBAL_MEM_FENCE);
8: if i < 10 then
9:     ...; (b)
10: else
11:    return;
12: end if
```

V.2.4. Memory Coalescing and Bank Conflicts

Depending on the device's memory architecture the data access to global memory can be optimized in certain ways. *Memory coalescing* refers to a technique if specific access patterns occur during runtime. Let us assume we have two 1 dimensional workgroups $W_{1,1}$ and $W_{1,2}$, each with 16 threads and a memory segment s_1 of 64 elements. Considering the access pattern depicted on the left image in Fig. V.2 certain devices can coalesce memory requests into one large request, the data will be delivered as one large data block to the requesting threads. With respect to the illustrated situation t_1 to t_{16} within $W_{1,1}$ will access m_1 to m_{16} in a sequential way. Since adjacent threads are requesting adjacent memory locations in a sequential way the system would coalesce them rather then utilize single memory requests. The right image in Fig. V.2 shows a situation of rather erratic

requests, which do not give the system any opportunity to coalesce these requests. Many modern devices also provide similar optimization, usually referred to as *request broadcasts*, which are applicable if adjacent threads request the same memory location. In such a case the device will deliver the data to all threads via memory broadcasts.
In the context of certain devices, shared memory requests can lead to an effect called bank conflicts, which are requests from different threads to different memory locations in the same memory bank. Thus one should attempt to linearize the access patterns to global memory and avoid multiple bank conflicts in each execution step.

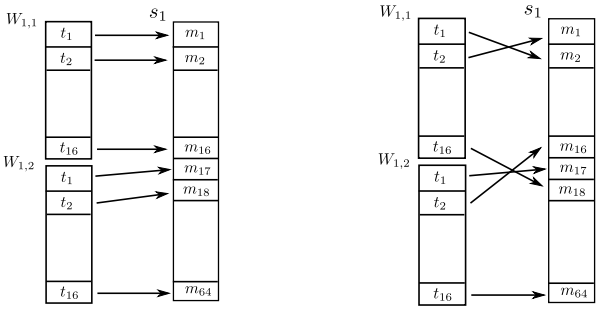


Figure V.2.: The situation for a possible memory coalescing is shown on the left image while the right one shows an access pattern which prohibits the coalescing.

V.2.5. Shared Memory

Another important optimizations technique for memory accesses is to simply utilize faster memory types. Global memory is usually considered when it comes to hold large amounts of data (i.e. above the megabyte range), shared memory is usually applied in the smaller areas i.e. when handling of a few kilobytes is required. Modern GPUs for example provide up to 48kB of shared memory for each compute unit[NVi12a], Alg. 9 shows a kernel which utilizes shared memory, the first step is to actually preload the data from global memory into the allocated shared memory area.

Algorithm 9 Parallel Reduction in Shared Memory

Input: Global array s_global

```
1: i =get_global_id(0);
2: __shared__ char s_mem[256];
3: s_mem[i] = s_global[i];
4: j = 2 * i;
5: char s=0;
6: for int k=1;k < 256;k=2*i do
7:     if j mod (2 * k) == 0 then
8:         s_local[j/2] = s_local[j/2] + s_local[j/2 + 1];
9:     end if
10:    barrier(CLK_GLOBAL_MEM_FENCE);
11: end for
12: if j == 0 then
13:    s_global[0] = s_local[0];
14: end if
```

The illustrated example performs the preloading in a coalesced

way, afterwards the preloaded data is used in the parallel reduction loop (note that, within Alg. 9, all requests to local memory avoid bank conflicts). In order to retrieve the reduction result one has to accept the thread divergence in the last lines, which let a single thread write the result into global memory.

VI. A Software Framework for Cluster Management and Distributed Computation

"In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering" (Donald Knuth, [Knu74])

In the context of existing approaches to cluster computing I present a newly developed modular framework 'SimpleHydra' for rapid deployment and management of Beowulf clusters. Instead of focusing only the pure computation tasks on homogeneous clusters (i.e. clusters with identically set up nodes), this framework aims to ease the configuration of heterogeneous clusters and to provide a low-level / high-level object-oriented Application Programming Interface (API) for low-latency distributed computing. The framework does not make any restrictions regarding the hardware and minimizes the use of external libraries to the case of special modules. Additionally SimpleHydra enables the user to develop highly dynamic cluster topologies. I describe the framework's general structure as well as time critical elements, give application examples in the 'Big-Data' context

during a research project and briefly discuss additional features. Furthermore I give a thorough theoretical time/space complexity analysis of the implemented methods and general approaches.

VI.1. Existing Frameworks

There exists a wide variety of different Big-Data problems, be it in scientific research or industrial applications. Developed solutions (algorithms or systems) often benefit from computation clusters, i.e. they are constructed to be parallelizable such that they may be distributed among many computation nodes.
Although cluster computing is a very interesting and active field of research, it is difficult to access for many (small) research institutes. Professional high performance systems are often unaffordable, thus universities or research institutes usually decide to use inexpensive Beowulf clusters [SBS+95] or related approaches. Examples are [DHL+03], [GDMO02] or [AV02], yet most clusters are designed to solve domain specific problems. If they provide a generic API, they often do not include support for cluster management, e.g. adding new nodes or updating/reconfiguring existing nodes. Additionally most Beowulf clusters assume heterogeneous nodes or a static topology, which is a serious restriction for partial system upgrades. Solutions are usually implemented by using plain communication abstractions like the Parallel Virtual Machine (PVM) [Sun90] or the Message Passing Interface (MPI) [12693], which provide a simple and efficient way for distributed computing, yet these APIs do not address generic concepts of cluster computing, e.g. load balancing strategies, node management. Many (domain specific) extensions for these interfaces exist, e.g. [DBPDP+06](enhanced PVM load balancing) or [DNGFV00](PVM over ATM lines), which address certain as-

pects of cluster computation. Often do frameworks include large dependencies to other external libraries, which are not guaranteed to work with future revisions.
The SCMS [UPAM00] framework addresses the management problems of Beowulf clusters and provides a practical set of functions. Yet its purpose is solely the management, it does not include inherent support for computation tasks. Building upon SCMS and other frameworks, SCE [UPAS01] provides a solution including support for computation tasks by using MPI. Yet, a rigorous analysis of its structure and implementation, e.g. of the low-level network communication with respect to current technologies, is omitted.
[SYT12] provides additional references to existing software approaches for cluster computation with a special focus on beowulf clusters. Additionally, [SYT12] also illustrates the challenge of deploying an application within a cluster environment and implicitly states the need for management facilities.
My aim is to address the problems of Beowulf based computation by providing an integrated but modular framework, which not only enables one to rapidly deploy and manage Beowulf clusters but also scales well for huge systems (>1000 nodes). Additionally the framework includes support for OpenCL based computation, which alongside the support for dynamic heterogeneous topologies provides the basis for a flexible system structure.
Section VI.2.1 outlines the general structure of SimpleHydra, while the critical aspect of network communication are addressed in section VI.2.2. The previously mentioned dynamic topologies are explained in section VI.2.3. I conclude this chapter with the description of the IGOR cluster which was utilized in the APFel research project [HMH13] for distributed and GPU-accelerated people detection. Furthermore the cluster provided the basis for

the evaluations in the following chapters.

VI.2. SimpleHydra

VI.2.1. A Coarse Look on the Structure

The frameworks modular structure incorporates the largest modules:

- Core
- Network
- Cluster

The 'Core'-module provides all basic data structures and management functions for the remaining elements, e.g. file system support, Inter-Process Communication (IPC) [Ste97] support, a thread management system, time measurement components, serialization facilities and others. All fundamental communication methods are provided by the 'Network'-module, due to its size and complexity it will be described in more detail within section VI.2.2. The 'Cluster'-module contains high level management routines which enable developers to quickly deploy canonical (i.e. framework provided) management clients and servers for a cluster infrastructure.
In addition to these modules, SimpleHydra (SH) provides a wide functional variety in the areas:

- data exchange, e.g. Matlab interface, XML support for basic XML access and configuration files or MySQL database connectivity.

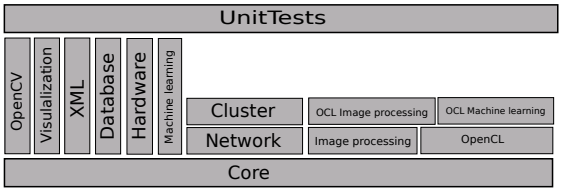


Figure VI.1.: The structure of SimpleHydra, each block represents a module, all modules beneath another module are required for its functionality

- image processing, e.g. elementary image manipulation, OpenCL based high performance object detection.
- machine learning e.g. LIBSVM wrapper, generic and adaptive neural networks with OpenCL support.
- hardware support for video input devices, e.g. V4L devices, AVT cameras.
- data visualization, e.g. video streams, images or functions.

Fig. VI.1 illustrates the described components and shows their dependencies, i.e. a module requires the components it stands on. The environment requirements in terms of soft- and hardware are very puristic throughout the different modules. Due to efficiency reasons, e.g. threading or time measurement, and financial aspects, e.g. regarding license costs, the decision was made to implement the framework only for Linux/Unix systems. SH provides interfaces to proprietary libraries such as Matlab, this of course requires the third party library to be present. Yet, the modular structure allows to easily remove the mentioned

module if one can not fulfill this dependency. Due to the frameworks size, I decided to utilize CMake and bash scripts for the build chain. This allows a fast creation of customized build configurations, e.g. for a small embedded system like a Raspberry Pi one could only build the modules 'Core' and 'Network'. The minimal software requirements are a Linux/Unix system, a C++ compiler, the C++ standard library and CMake ([HM03]). There are no hardware restrictions. In order to keep things brief the external dependencies for each module are only listed ('<o>' indicates it as being optional):

- Core(<o>libz, libpthread,<o>ncurses,<o>GSL)
- Network
- Cluster
- Database (libmysql || libmariadbclient, libboost_regex)
- Hardware (libVimbaCPP, libVimbaC)
- OpenCL (libOpenCL)
- Matlab (libmat)
- ImageProcessing (libpng, libjpeg)
- OpenCLImageProcessing
- OpenCLMachineLearning
- MachineLearning (libSVM)
- XML (libxml2)
- Visualization (Qt5, libcustomplot, qwt)

- UnitTests

The reason for such a sparse amount of small external libraries lies in the fact, that SH implements many data structures and elementary control mechanisms from scratch. This is needed to provide system-local thread safety while keeping the data access fast regarding primitive data containers, e.g. linked lists. The framework incorporates a build chain which generates release and debug make scripts for static and shared libraries (module wise). In addition to these libraries one can build an executable containing the unit tests. A detailed API description is available in [MLU16].

VI.2.2. Network Protocol and Communication Facilities

One of the most critical aspects in building a cluster is the communication bandwidth and latency between nodes. It is not only a question of choosing an appropriate physical interface but also the communication protocol. Professional high performance systems often use the Infiniband interface, which provides a 2.5GBs link [Jin01] and drive their data with the Transmission Control Protocol (TCP). Infiniband also has a much smaller latency of $\approx 1.7\mu s$ compared to GigE Ethernet $\approx 48\mu s$ [Cou09]. Although one might be tempted to use this interface for IPC, it does come with high hardware costs (NICs, switches etc.). Thus for small research institutes GigE (available on almost any modern computer) represents a cost efficient alternative to aforementioned HPC systems.
The concept of Beowulf clusters exists since 1995 [SBS+95] and initially described a set of Ethernet-connected workstations, whose communication based on a token exchange via UDP. Yet the User

Datagram Protocol (UDP) is a stateless Internet Protocol (IP) based method to transmit data, i.e. one does not have congestion control, receive control, ordering of packet fragments or reachability information about the communication partner. For simple domain specific Beowulf clusters the choice of using UDP might be well founded, e.g. small and sparsely exchanged tokens under the restriction of largely available network / node capacities. But for a generic approach, e.g. taking the management and control of the cluster into account, with respect to unknown fields of applications as well as heterogeneous hardware configurations, the control requirements for network communication will converge to the feature set of TCP.
Thus it was decided to implement the communication via connection-based TCP, the reasons for this choice will become clearer when I discuss the management feature set of SH. It should be mentioned that the SH framework does also utilize UDP, e.g. for dynamic cluster topologies, it is not restricted to TCP based communication. Yet, it does not provide high communication facilities with UDP.

VI.2.2.1. SH Communication

Before discussing the internal mechanism it should be pointed out that even though the described communication facilities will be carried throughout the remaining chapter; SH provides a generic API which allows developers to change/implement existing or new communication protocols down to the choice of sockets, e.g. as far as to choose packet sockets.
Communication, in management or computation relevant tasks, uses TCP payloads p of the form

$$p = [h|d] \tag{VI.1}$$

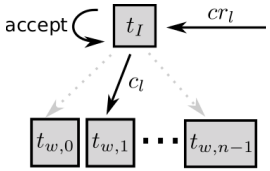


Figure VI.2.: The concept of binned worker threads; one thread t_I handles the incoming connection requests cr_l, creates the connection c_j and assigns it to an appropriate worker thread $t_{w,i}$ (bin)

where h is 4 bytes long and contains the size in bytes of the actual data d. Thus the shortest communication beacon will be 4 bytes large. In order to avoid synchronization problems or race conditions during heavy data exchanges, each data transmission exhibits a request-response form (see Sec. VI.2.2).

VI.2.2.2. Worker Threads

When it comes to socket communication under Linux/Unix systems the usual naive way of handling incoming connections is to start a single thread for each one of them. This approach is infeasible for large scale servers as it will clog the system with management overhead. Thus in large server applications the concept of binned worker threads is applied, this is depicted in Fig. VI.2 The system allocates a thread pool of n worker threads $t_{w,i}$ and starts a single connection handling thread t_I. Every connection request will be processed by t_I and delegated to a fitting thread $t_{w,i}$. The term 'fitting' already indicates that the choice of i is not arbitrary, the simplest strategy would be an even distribution among the workers. Only this approach was

implemented since the evaluation system exhibited a symmetry in terms of communication latencies and GPU setup, i.e. each single worker can handle up to m connections, in case of nm existing connections every incoming connection will be dropped. A more advanced way for example would consider the current load of the worker threads and choose the one with the lowest value.

VI.2.2.3. Efficient Socket Handling

The Linux kernel provides different mechanisms for accessing data in a socket (or checking for available data), namely *select*, *poll* and *epoll*. A call to *select* is the most basic way of checking for available data, it informs the kernel about all file descriptors (i.e. socket descriptors) it would like to check for new events. This approach does not scale well with a growing number of open file descriptors. The same holds for *poll*, which differs to *select* only in the number of maximal file descriptors (i.e. it has no fundamental limit compared to the bit mask approach of *select* [Ste97]). The *epoll* function removes this drawback as it only considers the active file descriptors, i.e. it does not require to provide the kernel with a list of desired elements. Both approaches were thoroughly analyzed in [GBSP04], who showed that *epoll* exhibits a measurable performance gain of up to 79% for sparse connection activity. Thus I decided to utilize *edge-triggered epoll* in SimpleHydra.
SimpleHydra's worker threads use so-called frame assemblers, in order to explain those we must begin with the problem they solve. For the sake of simplicity assume that only a single worker thread $t_{w,0}$ exists and handles m connections. For each of these m connections $t_{w,0}$ will have to assemble the corresponding data streams, as they may arrive in (ordered) fragments. Further-

more $t_{w,0}$ must apply the desired action, e.g. a callback, to the data streams payload. Thus each connection c_j is assigned a single frame assembler a_j, which handles the logic behind the assembling (buffer management and construction) as well as the interpretation of the data. The interpretation is done via frame handlers fh_j, which are an integral part of each frame assembler (one per assembler).
The process structure of socket management within a worker thread is illustrated through Alg. 10. Yet another problem arises in the context of binned worker threads. Let us assume $t_{w,0}$ processes the low-level socket descriptor sd_j of c_j, how does he find a_j efficiently in order to deliver the received data to it? In order to solve this an unordered hash list ($\mathcal{O}(\log(n))$) was utilized, it contained the tuples (sd_j, a_j) with sd_j being the key. One should note that this problem could be solved differently, e.g. by including a connection id into the header h, thus the worker thread could look up the frame assembler in a linear array. Yet this would require a management of available slots in the array. Using the socket descriptor as a key for the linear array itself is infeasible due to its numerical range (4/8 bytes), i.e. this would restrict the array to be continuously growing with each new connection (especially critical for the case of very frequent closed and reopened connections over a long time period), i.e. we would gain $\mathcal{O}(1)$ worst-case lookup time for the cost of a limited system runtime due to a finite memory amount. This dilemma can not be avoided for situations with a variable amount of non-persistent connections.

Algorithm 10 Worker thread $t_{w,i}$ socket management

```
1: while worker is active do
2:     (num, event) =getActiveConnections;    → epoll
3:     for i=0; num - 1 do    → determine request type
4:         if event[i].req=="disconnect" then
5:             find and delete connection from container;
6:         end if
7:         if event[i].req=="connect" then
8:             create and add connection to container;
9:         end if
10:        if event[i].req=="data" then
11:            find and call assembler a_j;
12:        end if
13:    end for
14: end while
```

Each worker thread contains a private epoll system, which is used to observe the socket descriptors of all assigned connections. Thus we can summarize the average time complexity T_{sock} of this approach as follows (for the sake of simplicity intuitive index names were chosen).

Lemma 1. *Let act_i be the number of active connections in $t_{w,i}$ with $i \in [0, n-1]$ and $act_i \in [1, m]$. Furthermore let D be a data structure, capable of holding integer values, with functions D_{get}, D_{add}, D_{del} and corresponding average complexity sets $\mathcal{O}_{get}(g(k))$, $\mathcal{O}_{add}(a(k))$, $\mathcal{O}_{del}(d(k))$ for k contained elements. Then the average time complexity for a single iteration of $t_{w,i}$ is*

$$T_{sock,i} = \mathcal{O}(\mathbb{E}(act_i) f(k)) \tag{VI.2}$$

with f being a function from the largest of the mentioned complexity sets and $\mathbb{E}(.)$ the expected value.

Furthermore the complete average complexity (for a single parallel iteration of all worker threads) is given by

$$T_{sock} \quad = \mathcal{O}(\max_i(\mathbb{E}(act_i))f(k)) \tag{VI.3}$$

Proof. We have to distinguish two cases, firstly the case of $\mathbb{E}(act_i) = 0$, where the above statements obviously hold. Secondly the more interesting case of $\mathbb{E}(act_i) > 0$. First one has to observe that $\mathbb{E}(act_i)$ can be splitted into $\mathbb{E}(dis_i) + \mathbb{E}(new_i) + \mathbb{E}(get_i)$, where the expectancy values refer to the case of disconnect requests, new connections and existing connections, respectively. Furthermore there exist factors α, β, γ with $\alpha\mathbb{E}(act_i) = \mathbb{E}(dis_i)$ etc. (for example $\beta = (\mathbb{E}(del_i) - \mathbb{E}(get_i))/\mathbb{E}(act_i)$). Every worker has to retrieve the active sockets, this can be done in constant time due to preallocated kernel structures (or in $\mathbb{E}(act_i)$ steps from a rigorous point of view). After the descriptors have been retrieved one must process each one of them (i.e. $\mathbb{E}(act_i)$ descriptors), they may inform the program over disconnections, new connections or data for existing connections. For each request type one must execute data structure routines, i.e. D_{del}, D_{get}, D_{add}, respectively. Let $g' \in \mathcal{O}_{get}(g(k)), a' \in \mathcal{O}_{add}(a(k)), d' \in \mathcal{O}_{del}(d(k))$ be arbitrary functions. The complexity for the processing of all requests is summarized by

$$\mathcal{O}(\mathbb{E}(act_i)(\alpha d' + \beta a' + \gamma g')) \tag{VI.4}$$

which is dominated by the function with the largest asymptotic behaviour, i.e. f. Thus we obtain the complexity for a single iteration and for multiple parallel iterations (as n is constant). □

On the basis of lemma 1 it is simple to conduct further runtime analysis depending on the assumed distribution of act_i and the utilized data structure D. The extension for inclusion of high level functions for each request type can be done by adding their complexity to the complexity of the corresponding data structure routines (i.e. d, a, g). One can also deduct that for a constant time complexity within the described threading concept, all of the data structures operations must be able to finish in constant average time.

VI.2.2.4. Memory Management and Space Efficiency

Apart from the operating systems send and receive buffers, two additional buffers are required in which an assembler a_j can iteratively construct the outgoing and incoming data streams. In the system each a_j constructs an appropriate receive buffer for every incoming datastream, thus memory is only allocated if it is required (depicted on the left image in Fig. VI.3). Regarding the outgoing data a different approach was chosen. The worker thread contains a single transmit buffer, which is shared among the managed sockets. Each socket will either send all of his queued data or fail, under this restriction one can reduce the amount of required memory significantly (see the right image in Fig.VI.3). Yet this strategy can not be applied for incoming (fragmented) data streams, as one has to store incomplete data streams over time until all fragments have been received.

The required buffer size for an incoming data stream is determined once the first 4 bytes (i.e. the header) have been received. Additionally the send process only considers the existing data in the shared output buffer, .i.e. it does not send all allocated buffer bytes. The output buffer size is determined during runtime by analyzing certain system attributes.

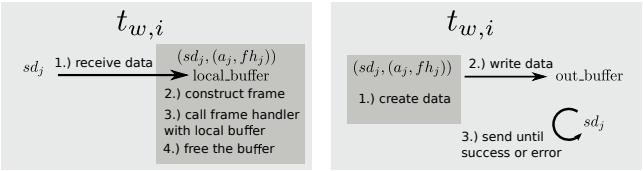


Figure VI.3.: Left side: $t_{w,i}$ receives a data fragment within an iteration and directs them to the appropriate assembler a_j, which stores it in a local buffer and continues with frame reconstruction. Once a frame has been completely received, the framehandler fh_j will commence the interpretation of the payload. Right side: the frame handler fh_j attempts to send data over sd_j, first the data is copied into the worker threads shared output buffer, afterwards the worker thread attempts send all data contained in the buffer (i.e. only the existing payload). The colored rectangles represent different contexts.

As mentioned before the protocol uses a simple request-response scheme, this simplifies the logic behind frame assembling. Through the use of TCP, data fragments are received in correct order, thus the assemble process is a simple concatenation of bytes. The process of frame construction is depicted in Alg. 11, each computational step can be done in $\mathcal{O}(1)$. The complexity is mainly determined by the call to fh_j, which can commence arbitrary actions with respect to the received payload.

Thus we can summarize the communication protocols complexity with

Theorem 2. *Let Ω be the average complexity set for actions taken by framehandlers fh_j in a given context and $\omega \in \Omega$. The basic SimpleHydra network communication system, with respect to a single (parallel) iteration of all worker threads, exhibits a complexity of*

$$T_{com}(.) = \mathcal{O}(\max_i(\mathbb{E}(act_i))(f(k) + w(.))) \tag{VI.5}$$

Proof. Follows directly from lemma 1 and the corresponding remarks. □

VI.2.3. Dynamic Topologies

The communication topology of a Beowulf cluster is star shaped, with a management node in the center, it distributes work among the available nodes (including itself). This topology is usually assumed to be static in terms of parameters such as node count or communication interface. Additionally it is assumed that the nodes are similar (if not identical) configured. Yet, some applications benefit from a dynamical topology, e.g. one which

Algorithm 11 Frame assembling in a_j

Input: data fragment d

```
1: [static init] buffer = ∅; bytes = 0; payload_size = 0;
2: if bytes < 4 then
3:     append d to buffer;
4:     bytes += sizeof(d);
5:     return
6: end if
7: if bytes ≥ 4 ∧ payload_size == 0 then
8:     payload_size = h    → h=bytes[0,..,3]
9:     append d to buffer;
10:    bytes += sizeof(d);
11:    return
12: end if
13: if payload_size > 0 then
14:    append d to buffer;
15:    bytes += sizeof(d);
16:    if bytes == payload_size then
17:        call fh_j with buffer
18:        buffer = ∅; bytes = 0; payload_size = 0;
19:    end if
20:    return
21: end if
```

allows the insertion or removal of nodes during runtime, where the nodes may be differently structured, e.g. powerful multi-GPU nodes.
Thus the framework was designed in a way which allows the configuration of such clusters, furthermore it provides a low-level API which allows not only the construction of highly dynamic topologies but also their runtime management.
The general structure of this system is depicted in Fig. VI.4, where the management node executes two distinct (independent but connected) subprograms; the *management service* and the *computation task*. The management service is capable of for example keeping track of each node's available computation resources, copying data onto nodes or executing arbitrary system commands. The computation task has the responsibility of managing work distribution among the nodes.

Each node runs the corresponding counter parts; the *management client* and the *SH Unit*. The concept of SH Units will be described in the next section, for now it should suffice to consider an SH Unit as distributed workload. Similar to the management node, the subprograms on a computation node are independent but connected. The motivation for this design was to keep the cluster stability as high as possible. Even if an SH Unit fails, e.g. enters an endless loop, the management node can still use the connection to the management client to stop the Unit through the node's operating system.
The management service and computation task are being executed in the same system process (but in separate threads). For the sake of simplicity let us assume that each node already runs the management client. First the management service will be started, it will find all available nodes on the network (predefined or dynamic, details in next subsection) and setup a connection

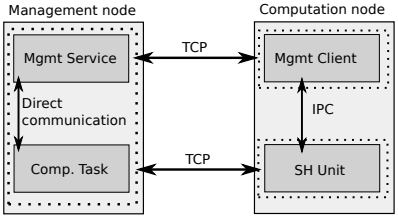


Figure VI.4.: The canonical network service structure of SimpleHydra, the dashed rectangles represent different address spaces. Management of the computation nodes is done via a TCP connection between two corresponding services. These services are independent of the actual computation task but can communicate with it either via direct addressing or IPC. The node interaction during computation tasks is also done via TCP connections.

to them (i.e. the running clients). Afterwards it will distribute SH Units among them and start the computation task. Each SH Unit will then connect to the computation task and the actual work may commence.
The computation nodes establish the connection, thus they have to handle only two connections (one for management and one for computation tasks), whereas the management node will handle its connections efficiently via the approach described in section VI.2.2.

VI.2.3.1. Self-Configuring Clusters

This section describes how the connections between nodes are being established. Within the previously described approach one might assume that the management node carries an initial list of

all potentially available nodes. This is not required as I designed SimpleHydra for self-configuring clusters.
The management client contains the so called UDP Remote Control Services (UDPRCS) feature, one of its functionalities being the ability to reply to home beacons (33 Byte large UDP broadcasts containing information about the management node). First the management node M will send a home beacon, all available nodes n_i may answer to it, M will wait for a defined time and create a list $N = \{n_i\}$ of available nodes. Independent of that, the nodes n_i will connect to the management service at M (the required data is extracted from the home beacon). The management node may use N to verify if all nodes have connected to it. Afterwards M will use these connections to distribute data and instructions to the nodes. Once the nodes have received all initial data they will execute the instructions, e.g. set up a connection to the computation task on M. This scheme is illustrated in Fig. VI.5, for the sake of understanding I omit the details of synchronization, e.g. the management client will wait until the SH Unit has been successfully deployed. Additionally I left out the details of network communication like response messages, the interested reader can find detailed descriptions in [MLU16].

Using the UDPRCS one can build architectures which allow the online expansion of computational resources. Yet, this approach works only on local subnets and thus SimpleHydra also supports the use of static node lists. These node lists allow the configuration of clusters in wide area networks, i.e. provide the possibility for grid computing.

VI.2.4. Cluster Management

One often underestimated point is the management of a cluster, this includes tasks as setting up single nodes, keeping track

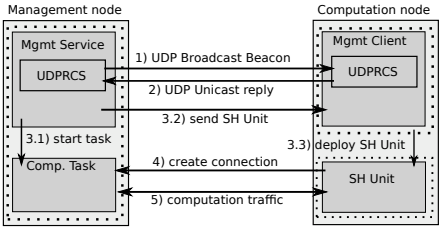


Figure VI.5.: The process of self-configuring clusters with SimpleHydra. The numbers inside the annotations denote the order of execution. First the management node attempts to find all available nodes on the local subnet via a UDP broadcast. The available nodes reply to the beacon and extract the servers connection information from it, e.g. address, port. Using this information they establish a connection to the management server which, once all nodes have connected, starts the computation task and sends an SH Unit to the nodes. Once a node received the Unit, it will deploy and execute it. The actual computation may then begin. For the sake of transparency the synchronization details of communication have been omitted.

of available nodes, updating software, e.g. libraries, on nodes and many more. For larger cluster systems (>20 nodes) these tasks create serious overhead for the administrator and introduce downtimes to the cluster. In order to accommodate this a generic function set is provided along with the management service, including e.g.:

- Remote shell (synchronous, asynchronous, parallel on multiple nodes)
- File copy

- Resource querying (CPU load, free Hard Disk Drive (HDD) space etc.)

A developer can use these functions to create solutions for more complex tasks, e.g. update multiple nodes by copying a script to them and using a parallel shell to execute it.

VI.2.5. Workload Distribution Paradigms

Let us consider the following scenario, an existing cluster with identically configured nodes (i.e. identical soft- and hardware) should be upgraded during runtime with one additional node. Yet this node contains almost completely different hard- and software (still a Linux/Unix system though). If the workload (i.e. the data and program code) would have been distributed as binary data, it would be impossible to use it on the new node. In order to address situations like this I developed the concept of SH Units.

VI.2.5.1. SH units

A SH Unit is a data structure $u = (\mathcal{P}, d)$, with $\mathcal{P}$ being an arbitrary payload and d a deployment script. Technically u has the form of a compressed archive, which contains C/C++ source files and binary data (i.e. the payload $\mathcal{P}$), the deployment script (i.e. d) usually consists of a single '*.sh' file. Alg. 12 illustrates the process of deployment on the computation node.

Algorithm 12 Deployment of an SH Unit u

Input: SH unit u

1: unpack archive into temporary folder
2: execute d → e.g. compile source files and execute the binary

The algorithm is very short as all deployment logic will be included in the deployment script. It can execute arbitrary commands, compiling the source files within the payload is only one of them. It may also gather system information and provide them to the executed binary or copy needed files to specific locations. The only statically defined step in the context of an SH Unit is the execution of the deployment script, which can also be used for administrative tasks. As mentioned in section VI.2.3 the started SH Unit (i.e. the compiled binary) can communicate with the management service through IPC. Thus, although the Unit is started as a new process the management service still has low-level control over it. In order to enable rapid prototyping I provide SH Unit / computation task templates for common use-cases such as map and pipe skeletons. The developer might also create completely new tasks with ease (since the system provides a transparent and generic class structure).

VI.2.5.2. Load Balancing

During runtime it is crucial to distribute the workload adequately among the nodes. Factors may be power efficiency [WL10], memory attributes [LLSW04] or domain specific optimizations [CK01], just to name a few. I do not aim to provide implemented solutions for any of such problems, instead my framework provides the needed infrastructure for implementing the

corresponding solutions.
The management service does not have to be completely idle during computation, for example it may collect information about the nodes. This can be done by techniques such as polling all nodes with a fixed frequency or letting the nodes themself transfer a status report (VI.6). The computation task may utilize this information for an arbitrary scheduling algorithm, e.g. use Hilbert curves ([AL11], [AB07]).

VI.2.6. Inner Workings of SH

Every connected node is being assigned a unique node id, which is stored in a temporary database on the management node. A computation task can access the contained entries (which also include other node metadata) and decide whether a node is capable of executing a certain operation. The details of collecting metadata is depicted in Fig. VI.6, the client c connects to M, during this process c gets assigned a node id and additionally sends a small system report R. R contains node attributes such as the number of OpenCL devices, CUDA devices, CPU information, the amount of Random Access Memory (RAM), available HDD space and many more. At the end of the registration process the report data is saved in a local database. A workload scheduling algorithm can utilize this meta information for a balanced distribution of tasks. It is also possible to periodically update the report information for a set of nodes (single updates are possible as well). For the sake of understanding any timers have been left out (which are used in steps 2 and 3) within this illustration.

The communication between management client and SH Unit on the node side is done via IPC mechanisms. For this and other purposes, SimpleHydra supports both, *System V* and *POSIX* based shared memory IPC. Another application for IPC is the

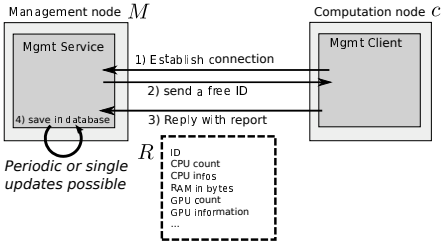


Figure VI.6.: The simplified registration process of a new node c and the management node M. First c establishes the connection to M, which in turn sends a free ID (this number is removed from the pool of available IDs until c disconnects or the registration fails). The computation node then acknowledges this number by sending it back as an attributes of a system report R. The management node then saves the report data in a local database and assigns the ID to the corresponding connection. It is possible to update the report data periodically (i.e. c sends periodic updates) or only per request (i.e. M requests an update from c).

libraries visualization module (see Fig. VI.7), which uses shared mutexes for synchronization with the Qt based window system. It must be noted that although SimpleHydra provides an object oriented interface to this window system, the system itself provides a low level C language interface, which allows its use in native C programs.

Let us briefly discuss how the communication protocol avoids race conditions and synchronization problems. Let us assume (without loss of generality) three communication parties A, B and C are communicating with each other, A and B are separate threads on a management node while C is a thread on a

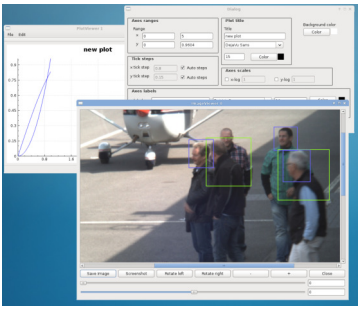


Figure VI.7.: The window system is part of the visualization module, it contains features such as sophisticated plot features, image viewers with annotation functions, video viewers and cluster management tools like parallel remote shells.

computation node. Furthermore let A be receiving a large file from C and B preparing to request a status report from C. The protocol uses a sequential request-response format, i.e. the communication tasks are queued and will be sequentially processed. In our case this means that while A receives data, no other communication will occur on this connection (i.e. B will wait for A to finish). This allows a simple and fast message parsing for each connection as the receiver can be sure that he receives a coherent payload stream, i.e. only the data from one communication context (no interleaved fragments). Once A has received all the data, the next enqueued communication will commence, i.e. B will send a request to C, which in turn will respond with a status report. It must be noted that such a connection scheme has its drawbacks, e.g. for a connection which transfers mostly large files, short messages will experience a great latency. Yet in the context of Beowulf cluster computing one usually avoids great

amounts of network communication due to the small bandwidth and high latency of Ethernet, additionally the exchanged data blocks are often small and in similar size. Due to these reasons I implemented the described sequential approach, yet SimpleHydra provides the user with freedom to implement a (situation-)optimized communication scheme.

VI.2.7. SimpleHydra Deployment in a Cluster

In order to efficiently use my software I also optimized the deployment for an existing infrastructure. A Beowulf cluster makes little to no assumptions about the used hardware ([SBS+95] refers to the computation nodes as simple workstations), thus a corresponding framework should be able to deal with a large variety of hardware configurations. The most relevant hardware elements are system memory, CPUs, GPUs, HDDs and network interfaces, which the framework handles in a generic manner. Many frameworks incorporate similar functionality but archive it through the use of external libraries, which are often restricted to certain hard- or software configurations and can change their APIs anytime (which forces one to adapt the framework).
In order to circumvent these problems and restrictions I designed the framework from scratch with only a minimal amount of external dependencies (the biggest being the Qt framework for the visualization module). This makes deployment in a network environment very easy, for the basic distribution of workloads one only needs to copy a small (prebuilt) client demon to the workstation (if no prebuilt demon is available, one has to compile it either on the workstation or with a fitting crosscompiler). No further configuration is required, especially no external libraries besides the C/C++ standard library. This deployment can be archived for example via a small shell script, the client demon it-

self is very puristic and effectively consumes no system resources. Once this demon has started, the workstation becomes an available computation node. It should be noted that this demon can reside on the workstation after the computation tasks finished, thus the workstation will be made available for tasks of another management node.
The demon also determines the available system features, e.g. OpenCL, CUDA, memory amount etc., yet it is also possible to configure it with a static configuration file, which defines the available services. These static configuration files are useful in the case of strongly varying or special environments, which can make it difficult to reliably determine certain system features.

VI.3. The Beowulf Cluster IGOR

IGOR consists of 15 nodes in total, which are equipped as follows:

- 1 nodes: Intel i7 4770, 3x Radeon 7990, 64GB RAM, 12TB HDD (3 disks)
- 6 nodes: Intel i7 4770, 1x Radeon 7970, 16GB RAM, 1TB HDD
- 2 nodes: Intel i7 4770, 1x Radeon R9 290x, 16GB RAM, 1TB HDD
- 2 nodes: Intel i7 4770, 1x Geforce GTX780-Ti, 16GB RAM, 1TB HDD
- 2 nodes: Intel i7 4770, 2x Radeon R9 290x, 64GB RAM, 8TB HDD (2 disks)
- 1 nodes: Intel i7 4770, 3x Geforce GTX780-Ti, 64GB RAM, 8TB HDD (2 disks)

Figure VI.8.: The two node types used in IGOR. The left picture shows very compact mini-ITX case with one discrete GPU, 16GB RAM and one HDD of 1TB. The right side depicts a large ATX tower with 64GB RAM, multiple GPUs and HDDs (8TB).

- 1 node: Intel i7 4770, 16GB RAM, 1TB HDD

The last node was used as a dedicated management node, i.e. it did not participate in the execution of SH Units (Fig. VI.8 shows the two different types of nodes). Thus in total IGOR features 56 x64 CPU-cores with 3.6GHz, 368GB of system memory and 55872 GPU shaders, with a total of $\approx$ 107 TFlops (synthetic, FP32 GPU) and 2.38 TFlops (synthetic, FP32 CPU). Except for the identical CPU the nodes exhibit different amounts of RAM, different GPU counts and types (this implies different local tool chains and interfaces) and different amounts of HDD storage. The nodes are interconnected via a dedicated GigE switch and used different versions of ArchLinux. All evaluations of the following chapters have been conducted on IGOR, be it the complete cluster or just a subset of nodes.

VII. Parallelizing the HOG

Since the pioneering work of [DT05] many architectures for object detection started to utilize histograms of oriented gradients (HOGs) as robust feature components. Sadly the computation of HOGs introduces a high computational complexity, yet the involved operations are highly parallelizable. The corresponding application of GPUs has been widely studied, e.g. in [SL11], [HKE+13] or [AWW12]. Yet, to my knowledge none of them addresses the complete algorithm since a certain, highly parallelizable, element is usually left out.
In this chapter I describe a theoretical framework for the mean shift algorithm itself and analyze its applicability for small scale as well as large scale parallelization. The analysis of runtime complexities respects generic system aspects such as memory access and parallelization factors. The algorithmic skeletons can be used for further theoretical studies or as a guideline for designing platform specific implementations, be it GPUs or large cluster systems.

VII.1. The HOG Algorithm

This section describes the computation steps of the classic HOG algorithm in the context of GPU (or more generally SIMD) architectures. The HOG algorithm consists of 6 atomic steps of which the first 5 are sequentially iterated in a cycle C until a

certain stopping criteria has been met. Since the novelty of my approach mainly resides in accelerating the last step, I will only briefly elaborate on the first steps.
The original HOG algorithm by Dalal involves a loop which sequentially executes 1+3 major phases: shrink image, shift detection window over the image, extract a HOG descriptor for each position, classify the descriptors. Where the last 3 phases are repeated until all possible window positions have been analyzed (depicted in Alg. 13 where $I, s, win_w, win_h, win_s$ represent the input image, scale factor, window width, window height and window stride, respectively). Once all descriptors have been classified, i.e. have been assigned a value which describes the system's certainty that the window contains the desired object, the detections are grouped by the mean shift algorithm MS.
The mean shift algorithm as explained in [Com03], [CM02] or in the original paper [FH75], is a non-parametric method which allows one to approximate and select modes in a sample from some arbitrary distribution of multidimensional data points. It is widely used in the area of image processing, e.g. as a clustering method in the HOG algorithm [DT05] or as tracking method in [Avi05]. Although it exhibits a simple design, it may become unfeasible in certain situations if the sample size grows to large. One example is its application in the HOG algorithm since, even if the detection system itself exhibits a high efficiency [HMH13], it may produce a number of results for which the mean shift clustering would become an efficiency bottleneck. Modern multicore CPUs and GPUs provide excellent architectures for parallelized mean shift execution. Yet up to this point I know of no existing theoretical analysis regarding the parallelization of this algorithm. There already exist GPU accelerated implementations, e.g. a generic but naive version within the OpenCV library or a

very specialized version for tracking applications in [LX09] using CUDA. My approach addresses exactly these points by providing an efficient and generic parallelized mean shift algorithm.

Algorithm 13 Classic HOG

Input: HOG parameters: $I, s, win_w, win_h, win_s, ...$
Output: l

```
1: I_c = I, s̃ = s, l = ∅;
2: while Detection window fits into current image do
3:     Shrink(I_c,s̃);
4:     Position the windows left upper corner at (0, 0);
5:     while I_c not covered do
6:         Shift Window by one quantum win_s;
7:         Compute gradient image I_G = (I_A, I_φ) for window content;
8:         Calculate histogram H = H(I_G);
9:         Classify H, add result to l;
10:    end while
11:    I_c = I, s̃ = s̃ ∗ s;
12: end while
13: Group elements from l via mean shift l = MS(l);
```

The implementation of the first steps follows [PR09] with only minor modifications in order to utilize more recent GPUs.

VII.1.1. Mean Shift based Non-Maximum Supression

In order to understand my motivation I will briefly discuss the original mean shift algorithm (with notation lend from [DT05]) and derive a massively parallelizable form. Let $S \subseteq_M \mathbb{R}^k$ be a sample of n k-dimensional data points drawn from a distribution

$\mathcal{D}$. For any given point $p \in S$ the mean shift algorithm will iteratively compute an approximation of the closest mode y_p to it. This is done via the following iteration equation

$$y_p = H_h(y_p) \sum_{i=1}^{n} \overline{\omega}_i(y_p) {H_i}^{-1} y_i \tag{VII.1}$$

with

$$\overline{\omega}_i(y_p) = \frac{|H_i|^{-1/2} \exp(-D^2[y_p, y_i, H_i]/2)}{\sum_{j=1}^{n} |H_j|^{-1/2} \exp(-D^2[y_p, y_j, H_j]/2)} \tag{VII.2}$$

$\biguplus_{i=1}^{n} \{y_i\} = S$ and

$$D^2[y_p, y_i, H_i] \quad := \quad (y_p - y_i)^T {H_i}^{-1} (y_p - y_i) \tag{VII.3}$$

These expressions require a more detailed explanation, H_i is the so called uncertainty matrix and takes the role of an estimated covariance matrix. Thus $D^2[y_p, y_j, H_j]$ becomes the Mahalanobis distance between y_p and y_j. Note that in its original form the algorithm uses an undefined kernel $K(y_p, y_i)$, yet as the Radial Basis Function (RBF) kernel (i.e. $K(y_p, y_i) = \exp(-|(y_p - y_i)|/(\sigma^2))$) is most common, K will be fixed to be of this form throughout the chapter. In addition to the generic kernel, the original algorithm does not utilize statistical methods to compute the distance between data points, thus it does not use any covariance matrices. The matrix H_h is defined indirectly through

$$H_h^{-1}(y) = \sum_{i=1}^{n} \overline{\omega}_i(y) H_i^{-1} \tag{VII.4}$$

In practical applications one usually has a closed expression to calculate H_i, i.e. $H_i = H_i(y_i)$.

Remark VII.1. *Note that if H_i is a diagonal matrix, H_h is one as well and can be calculated with high numerical stability.*

By rewriting eq. VII.1 and eq. VII.2 one obtains

$$y_p = H_h(y_p)\frac{\sum_{i=1}^{n} |H_i|^{-1/2} \exp(-D^2[y_p, y_i, H_i]/2){H_i}^{-1} y_i}{\sum_{j=1}^{n} |H_j|^{-1/2} \exp(-D^2[y_p, y_j, H_j]/2)} \tag{VII.5}$$

Which indicates that $|H_i|^{-1/2} \exp(-D^2[y_p, y_i, H_i]/2)$ occurs exactly the same amount of times in nominator and denominator. With this observation one can significantly reduce the required calculations by simply evaluating the expressions in lockstep. The same holds for eq. VII.4, which has the form

$$H_h^{-1}(y_p) = \frac{\sum_{i=1}^{n} |H_i|^{-1/2} \exp(-D^2[y_p, y_i, H_i]/2){H_i}^{-1}}{\sum_{j=1}^{n} |H_j|^{-1/2} \exp(-D^2[y_p, y_j, H_j]/2)} \tag{VII.6}$$

Furthermore the denominators in eq. VII.5 and eq. VII.6 cancel each other out. This induces the natural algorithm 14.

Algorithm 14 *approxMode* (calculation of eq. VII.1)

Input: start point $y_p \in \mathbb{R}^k$, n

1: $y_{p,t} = 0; H_{h,t} = 0$; $e = 0$;
2: **for** i=0; $n-1$ **do**
3: calculate $H_i^{-1}(y_i)$
4: $e = |H_i|^{-1/2} \exp(-D^2[y_p, y_i, H_i]/2)$;
5: $H_{h,t}$ += $eH_i^{-1}(y_i)$
6: $y_{p,t}$ += $eH_i^{-1}(y_i)y_i$;
7: **end for**
8: $y_p = H_{h,t}^{-1} y_{p,t}$

The runtime complexity is mainly determined by the calculation

of $H_i^{-1}(y_i)$, which involves two separate operations, the calculation of $H_{h,t}^{-1}$ and the determinant $|H_i|$. The computation of $H_i^{-1}(y_i)$ also involves the determination of H_i itself. Let the corresponding complexity classes be denoted by $\mathcal{O}_{H_i^{-1}}$ and $\mathcal{O}_{H_i}$, respectively. The expression $\mathcal{O}_{H_h^{-1}}$ shall denote the complexity class of determining $H_{h,t}^{-1}$ and with $\mathcal{O}_{|H_i|}$ that of calculating the determinant. Differentiating between both matrix inversions allows to incorporate inversion algorithms which exploit certain features of the matrices and thus differ in their complexity. Yet one also has to account for the elementary matrix operation, i.e. addition, matrix-vector multiplication and scalar product computation. We express this through the classes $\mathcal{O}_+, \mathcal{O}_{m*v}, \mathcal{O}_{v*v}$ respectively.

Lemma 2. *The complexity for a single execution of Alg. 14 in a general mean shift calculation is* $\mathcal{O}(n\cdot(\lambda+\kappa+\xi+\gamma+\rho+\phi)+\gamma+\eta)$, *with* $\kappa \in \mathcal{O}_{H_i^{-1}}, \lambda \in \mathcal{O}_{H_i}, \eta \in \mathcal{O}_{H_h^{-1}}$, $\xi \in \mathcal{O}_{|H_i|}$, $\gamma \in \mathcal{O}_{m*v}$, $\phi \in \mathcal{O}_+$ *and* $\rho \in \mathcal{O}_{v*v}$.

Proof. A total of n iterations will be executed. Each iteration requires (in following order): calculating H_i, inverting H_i, calculating the determinant $|H_i|$ and determining the Mahalanabois distance D^2. So far this yields $\lambda + \kappa + \xi + (\rho + \gamma) + \phi$. Scaling each component of H_i^{-1} by e is equally complex as adding two matrices, i.e. ϕ. Afterwards one has to account for updating $H_{h,t}$, multiplying y_i by eH_i^{-1} and updating $y_{p,t}$. This results in $\phi + \phi + \gamma + \rho$, as updating y_p has the same complexity as calculating the inner product. Summarized one obtains $n \cdot (\lambda + \kappa + \xi + \gamma + \rho + \phi)$, which is followed by η for inverting H_h and γ for calculating y_p. □

Remark VII.2. *For a general approach, using state-of-the-art algorithms, this complexity transforms to* $\mathcal{O}(n \cdot (\lambda + k^3))$.

The amount of iterations of Alg. 14 can be limited by an upper limit m_{it} and a distance threshold t_d for y_p between two consecutive iterations (see Alg. 17 for details). Let $\mathbb{E}_{it}(S)$ be the expected amount of Alg. 14 repetitions, which only depends on the current sample S. This implies an overall complexity of $\mathcal{O}(\mathbb{E} \cdot n^2 \cdot (\lambda + \kappa + \xi + \gamma + \rho + \phi) + n\gamma + n\eta)$, note that this expression refers to the approximation of all n modes.
As all mode approximations do not rely on each other it is possible to execute them in parallel. This would scale the complexity down by $1/z$ with z being the amount of Parallel Processing Units (PPUs).

VII.1.1.1. Diagonal Regular Covariance Matrices

With the above analysis it becomes obvious that for diagonal regular covariance matrices the stated complexity can be reduced dramatically. From now it shall be assumed that $H_i(y_i)) = diag(\sigma_1(y_i), ..., \sigma_k(y_i))$, which implies $\mathcal{O}_{H_i^{-1}} = \mathcal{O}_{H_h^{-1}} = \mathcal{O}(k)$ (i.e. only the non-zero elements along the diagonal must be inverted) and $\mathcal{O}_{|H_i|} = \mathcal{O}(k)$ (summing up only diagonal elements). In order to explain my approach for GPU architectures, the previously described method must be analyzed with respect to parallelism. Let us assume that each of the z processing units incorporates $\tau \leq n$ Inter-Process Communications (PUs) (i.e. units without complex mechanisms such as instruction prefetching or branch prediction). We will now distribute the iterations of the for-loop in Alg. 14 over the PUs. Note that if each PU executes a subset $I \subseteq \{1, ..., n\}$ of iterations, we would obtain the previously mentioned parallelization factor $1/z$ in the case of $|I| = n, \tau = 1$,

in case of $|I| = 1, \tau = n$ we would obtain a spectacular boost of $1/n$ per PPU (as all for-loop evaluation would be executed in parallel). The following sections will elaborate with great detail on such cases.
Using this approach Alg. 14 becomes Alg. 15, where the index *pu* indicates a variable local to each PU and the function 'getPUIdx()' returns the index of the PU. This formulation does not require $n = l \cdot \tau$, but implies that each PPU must calculate a complete evaluation of eq. VII.1 for only a single point y_p. The load factor f_{load} indicates how many iterations, i.e. $\lceil n/f_{load} \rceil$, should be handled by a single PU. This algorithm will be executed in parallel by each PU on a PPU, it does not require a specific execution paradigm such as a lockstep execution or any specific system architecture, e.g. a GPU SIMD environment. Although one would benefit from this strategy for a large n, it yields a significant drawback for small n as well as for a small number of PPUs with many PUs. For small n, i.e. $(n < \tau)$, one would like to be able to process multiple evaluations, i.e. for different y_p, on a single PPU in order to reduce underutilization. The same holds for a small number of PPUs, i.e. $z < n$. This issue can be solved through the concept of Virtual Parallel Processing Units (vPPUs) , which will be explained in the next section.

Remark VII.3. *If $\tau \nmid n$, the algorithm will underutilize the available PUs independent of the chosen value for f_{load}. Only for $f_{load} = \tau$ will all PUs be utilized and the amount of iterations per PU minimized.*

Let us now analyze the complexity of Alg. 15. Constants will only be carried into the analysis if they are induced through this algorithm, i.e. if they haven't occurred in Alg. 14. This way

Algorithm 15 *approxMode* (Parallel calculation of eq. VII.1)

Input: start point $y_p \in \mathbb{R}^k$, load factor $f_{load} \leq \tau$, τ, n

1: $(G)step = \lceil n/f_{load} \rceil$
2: $(P_{y_{p,t},i_{pu},i_{start,pu},i_{end,pu},y_{p,t,pu},i,H_{h,t,pu},e})\{$
3: $y_{p,t} = 0; H_{h,t} = 0;$
4: i_{pu} =getPUIdx(); $\quad \rightarrow$ PU local work begins here
5: $i_{start,pu} = i_{pu} \cdot step$
6: $i_{end,pu} = (i_{pu} + 1) \cdot step$
7: $y_{p,t,pu} = 0; H_{h,t,pu} = 0;\ e_{pu} = 0;$
8: **if** $i_{end,pu} - i_{start,pu} < step$ **then**
9: $\quad i_{end,pu} = i_{end,pu} - i_{start,pu}$
10: **end if**
11: **for** i=$i_{start,pu}$; $i_{end,pu}$ **do**
12: $\quad$ calculate $H_i^{-1}(y_i)$
13: $\quad e = |H_i|^{-1/2} \exp(-D^2[y_p, y_i, H_i]/2);$
14: $\quad H_{h,t,pu}$ += $eH_i^{-1}(y_i)$
15: $\quad y_{p,t,pu}$ += $eH_i^{-1}(y_i)y_i;$
16: **end for**
17: parallel reduction of $H_{h,t,pu}$ to $H_{h,t}$
18: parallel reduction of $y_{p,t,pu}$ to $y_{p,t}$
19: **if** $i_{pu} == 0$ **then**
20: $\quad y_p = H_{h,t}^{-1} y_{p,t}$
21: **end if**
22: $\}$

a direct comparison of both approaches remains possible. Furthermore due to the needed parallel reduction a uniform memory access (UMA, [Fly72]) architecture is assumed, with a fixed cost c for every data access on scalar elements.

Theorem 3. *Alg. 15 exhibits a complexity of* $\mathcal{O}(\lceil n/f_{load} \rceil \cdot (5k + \lambda) + 2\log(\tau) \cdot k + 2(\sum_{i=1}^{\log_2 \tau} \frac{\tau}{2^i}) \cdot k \cdot c + \aleph(k, \tau) + k)$. $f_{load} = \tau$ *is the optimal choice. If* $f_{load} \nmid n$ *then at least one PU will not be fully utilized. In case of* $log_2(\tau) \in \mathbb{N}$ *we obtain* $\aleph(k, \tau) = 0$.

Proof. All initializations can be done in constant time (line 1-9), the for loop exhibits $\lceil n/f_{load} \rceil$ iterations with the same complexity as in Alg. 14. Due to the diagonal matrices the complexity for the algorithm until line 16 is $\lceil n/f_{load} \rceil \cdot (5k + \lambda)$, as the PUs will execute their local for-loops in parallel. The interesting part are the parallel reductions in line 17 and 18. We will state the complexity only for line 17 as the same arguments hold for line 18. There are $\log_2(\tau)$ iterations involved in the reduction, if τ is not a power of two the Parallel Reduction (PR) will only process $l < k$ data elements with l being the nearest power of two. The remaining elements must be processed otherwise, e.g. sequentially, which induces an additional complexity term $\aleph(k, \tau)$ (which equals 0 in case that $\log_2(\tau) \in \mathbb{N}$). Each of the active PUs in one of the PR iterations has to sum up k elements, i.e. add $H_{h,t,pu'}$ of an adjacent PU pu' to its own copy. Thus in total there are $\log(\tau) \cdot k$ calculations. Yet, since each PU can access the memory of another one in a uniform way, we have to account for that with $(\sum_{i=1}^{\log_2 \tau} \frac{\tau}{2^i}) \cdot k \cdot c$. Since this holds for the second PR as well, we obtain the factor 2. Finally a single matrix multiplication remains, which has a complexity of k.
In case if $f_{load} < \tau$ we obtain $n/\lceil n/f_{load} \rceil < \tau$, which in turn implies the existence of at least one underutilized PU. Trivially

this holds as well for the case $n < \tau$. Note that $f_{load} = \tau$ is not sufficient for full utilization. Observe that $(l \cdot \tau + p = n) \Rightarrow n/\lceil n/\tau \rceil < \tau$. Thus only if additionally τ divides n we obtain full utilization. □

Remark VII.4. *In order to force* $\aleph(k, \tau) = 0$ *one could pad* n *to a power of* 2 *with numerically feasible dummy data. Furthermore one has to restrict the available number of PUs to a power of 2 closest to* τ*, i.e. set* $f_{load} = 2^x < \tau$*. Yet this in turn would imply underutilization, thus such a decision must be evaluated carefully.*

Although it might seem that the new algorithmic skeleton is much more complicated than the first one, I will show that with a fitting adaptation for the system architecture it can provide a significant improvement to the naive approach. Yet before that, we will increase the flexibility of Alg. 15.

VII.1.1.2. Virtual Parallel Processing Units

In order to circumvent the drawbacks of Alg.15 in the context of small data sets as well as in the case of only a few PPUs with many PUs, e.g. GPU architectures, the algorithm will be modified only marginally by subdividing the PUs on a PPU into virtual PPUs (vPPUs). For this purpose we introduce a new variable f_{vppu} which states how many of the PUs should form a vPPU. Thus the load factor becomes local to each vPPU.

Remark VII.5. *In order to keep the analysis simple I assume that* $f_{vppu} \mid \tau$*, the reader interested in the generic case can generalize the analysis with additional complexity terms in the same way I did with* $\aleph(k, \tau)$*.*

Algorithm 16 *approxMode* (Parallel calculation of eq. VII.1)

Input: start points $y_{p,j} \in \mathbb{R}^k$, load factor $f_{load} \leq \tau / f_{vppu}$, τ, n, f_{vppu}

1: $(G) step = \lceil n / f_{load} \rceil$
2: $(P_{y_{p,t}, i_{pu}, j_{pu}, i_{start,pu}, i_{end,pu}, y_{p,t,pu}, i, H_{h,t,pu}, e})\{$
3: $y_{p,t} = 0; H_{h,t} = 0;$
4: i_{pu} =getPUIdx(); $\quad \rightarrow$ PU local work begins here
5: $j_{pu} = \lfloor i_{pu} / f_{vppu} \rfloor;$
6: $i_{start,pu} = (i_{pu} \mod f_{vppu}) \cdot step$
7: $i_{end,pu} = ((i_{pu} \mod f_{vppu}) + 1) \cdot step$
8: $y_{p,t,pu} = 0; H_{h,t,pu} = 0;\ e_{pu} = 0;$
9: **if** $i_{end,pu} - i_{start,pu} < step$ **then**
10: $\quad i_{end,pu} = i_{end,pu} - i_{start,pu}$
11: **end if**
12: **for** i=$i_{start,pu}$; $i_{end,pu}$ **do**
13: $\quad$ calculate $H_i^{-1}(y_i)$
14: $\quad e = |H_i|^{-1/2} \exp(-D^2[y_{p,j_{pu}}, y_i, H_i]/2);$
15: $\quad H_{h,t,pu}$ += $eH_i^{-1}(y_i)$
16: $\quad y_{p,t,pu}$ += $eH_i^{-1}(y_i) y_i;$
17: **end for**
18: vPPU-local parallel reduction of $H_{h,t,pu}$ to $H_{h,t}$
19: vPPU-local parallel reduction of $y_{p,t,pu}$ to $y_{p,t}$
20: **if** $i_{pu} \mod f_{vppu} == 0$ **then**
21: $\quad y_p = H_{h,t}^{-1} y_{p,t}$
22: **end if**
23: $\}$

Thus the runtime complexity of Alg. 16 is given by

Theorem 4. *Alg. 16 exhibits a complexity of* $\mathcal{O}(\lceil n/f_{load} \rceil \cdot (5k + \lambda) + 2\log(\tau') \cdot k + 2(\sum_{i=1}^{\log_2 \tau'} \frac{\tau'}{2^i}) \cdot k \cdot c + \aleph(k, \tau') + k)$. $f_{load} = \tau / f_{vppu} =: \tau'$ *is the optimal choice. If* $(f_{load}/f_{vppu}) \nmid n$ *then at least one PU will not be fully utilized. In case of* $log_2(\tau') \in \mathbb{N}$ *we obtain* $\aleph(k, \tau') = 0$.

Proof. Follows directly from theorem 3 and remark VII.5. □

Remark VII.6. *The complexity of Alg.16 may seem identical to that of Alg. 15, yet Alg. 16 handles a set* $\{y_j\}_j$ *of data points!*

A general measure for PPU (and PU) underutilization can not be stated without defining the system architecture to use. An example for that would be a modern GPU which exhibits a very specific scheduling policy with respect to the amount of PU local memory requirements. Whereas a cluster system may distribute the computation in an arbitrary node topology. Only by defining such parameters one can truly assess the term utilization. A detailed example will be given in section VII.2 where I discuss the implementation for GPU architectures.
One should note that Alg. 16 does not impose any restriction onto the matrix structures, amount of data points or system architecture. It rather depicts a flexible skeleton for mean shift computation, which can be easily parallelized for arbitrary systems. The complete mean shift algorithm can be stated as follows. Alg. 17 is self-explanatory to the larger part. For every given data point y_i a corresponding mode $y_{p,i}$ is approximated. The for-loop iterations are independent to each other, thus they (this includes *approxMode* and *dist*) can be arbitrarily distributed among available PPUs. The algorithm *dist*

Algorithm 17 Mean shift clustering

Input: set of n data points $y_i \in \mathbb{R}^k$, t_d, ϵ, m_{it}

1: **for** i=0; $n-1$ **do** $\rightarrow$ arbitrary iteration-distribution over PPUs possible
2: $y_{p,i,prev} = 0$; $y_{p,i} = y_i$; $t_{d,i} = 0$, $c = 0$;
3: **while** $t_{d,i} > t_d \ || \ c < m_{it}$ **do**
4: $y_{p,i,prev} = y_p$;
5: $y_{p,i} = approxMode(y_{p,i}, ...)$;
6: $t_{d,i} = dist(y_{p,i,prev}, y_{p,i}, ...)$;
7: $c+ = 1$;
8: **end while**
9: **end for**
10: $\{\tilde{y}_p\} = groupModes(\{y_{p,i}\}, \epsilon)$

Algorithm 18 *dist*

Input: data points $y_i, y_j \in \mathbb{R}^k$, covariance matrix H_i

1: $d = D^2[y_j, y_i, H_i]$

is described in Alg. 18, its purpose is to calculate the Mahalanabois distance between two data points. The while-loop terminates if either the distance threshold t_d or the maximal number of iterations m_{it} has been reached. Afterwards the modes are grouped into ϵ-regions $\{\tilde{y}_p\}$ through algorithm $groupModes(\{y_{p,i}\}, \epsilon)$, which is stated in listing 19. The parameter ϵ defines the radius of a sphere s_y centered around each mode y, all adjacent modes within this sphere will be removed since they are considered to be equivalent with y. As depicted in the listing, the final result depends on the mode indexing, i.e. on how one iterates through $\{y_{p,i}\}_i$ This algorithm is inherently

Algorithm 19 *groupModes*

Input: data points $y_{p,i} \in \mathbb{R}^k$, ϵ

```
1:  c = 0;exists = 0;
2:  for i=0; n − 1 do
3:      exists = 0;
4:      for j=0; c do
5:          if dist(ỹ_{p,j}, y_{p,i}, ...) < ε then
6:              exists = 1;
7:              BREAK;
8:          end if
9:      end for
10:     if exists == 1 then
11:         add y_{p,i} to list of ỹ_p;
12:     end if
13: end for
```

sequential and thus can't be effectively parallelized. Yet in the context of Alg. 17 it can be exchanged for more efficient variants. The evaluations in section VIII.8 are based on Alg. 19, as

its runtime complexity has proven to be feasible for the practical application. The signature wildcards "..." indicate that arbitrary algorithms can be inserted at that position as long as their signatures are identical up to the wildcard.
We conclude the section with following

Theorem 5. *Alg. 17 exhibits a complexity of* $\mathcal{O}(\mathbb{E}(S)/(z{\cdot}\tau/f_{vppu})(\lceil n/f_{load}\rceil\cdot(5k+\lambda)+2\log(\tau')\cdot k+2(\sum_{i=1}^{\log_2\tau'}\frac{\tau'}{2^i})\cdot k\cdot c+\aleph(k,\tau')+2k)+\zeta)$. *With* $\zeta\in\mathcal{O}_{group}$ *being a function in the complexity class of Alg. 19 and* z *the amount of PPUs.*

Proof. Almost every part follows from theorem 4. Alg. 16 allows to compute up to $z\cdot\tau/f_{vppu}$ approximation steps in parallel. Whereas one final execution of Alg. 19 is required, thus the term $+\zeta$. □

VII.2. Application to GPU Computation

We begin by stating a last assumption regarding matrix structures, let

$$H_i(y_i)=diag(\sigma_1\exp(y_i^k),...,\sigma_{k-1}\exp(y_i^k),\sigma_k) \qquad \text{(VII.7)}$$

with σ_i being fixed parameters. Thus H_i is assumed to be diagonal and depends only on the last element of y_i (a^l denotes the l-th vector component).

Remark VII.7. *With this assumption one obtains* $\mathcal{O}_{H_i}=\mathcal{O}(k)$. *Thus the complexity of Alg.16 becomes* $\mathcal{O}(\lceil n/f_{load}\rceil\cdot(6k)+2\log(\tau')\cdot k+2(\sum_{i=1}^{\log_2\tau'}\frac{\tau'}{2^i})\cdot k\cdot c+k)$ *for padded data sets of size* n.

The concept of PPU/PU can be directly translated to GPU architectures as SMX/SP (Cuda) or compute unit/processing element (OpenCL). But as mentioned before, many aspects about

thread scheduling and memory architectures must be considered to utilize the efficiency of my approach. These techniques are well beyond the scope of this thesis, the interested reader might refer to [SGS10],[AMD13],[KH13],[NVi12b] or [SK10] for further details.
Implementing Alg. 19 was done in two succeeding stages using the OpenCL language, allowing its execution on multicore CPUs, NVidia GPUs as well as AMD GPUs. First I ported the algorithm naively, i.e. not considering coalesced global memory access and local bank conflicts. The second implementation utilizes preloading of data points into local memory and advanced synchronization mechanisms. In both cases the grouping of approximated modes was done on the host side using the system CPU. For the sake of readability I describe the optimization techniques for the second variant only briefly since they are self explanatory for the most part. Alg. 20 will preload the data points in sets of fixed size into local memory. As the local memory is available to all workitems in a workgroup, this might yield a speedup if multiple modes $\{y_{p,j}\}_j$ are approximated by one workgroup. Otherwise the workitems would access high-latency (i.e. ≈ 700 clock cycles) global memory. One should note the two distinct strategy elements, first the use of low latency memory and second the encapsulation of multiple node approximations on one workgroup. If one would choose to approximate a single mode per workgroup, there would be no gain from local memory. More or less surprisingly this approach yields little to no advance for computing devices capable of global memory broadcasts and lockstep execution, as the instructions are executing in lockstep. In other words, when all processing elements access the same memory position (local or global memory), this request will be delivered as an efficient broadcast to them. In case of a large

Algorithm 20 *approxModes* OpenCL implementation structure

Input: start points $y_{p,j} \in \mathbb{R}^k$, load factor $f_{load} \leq \tau / f_{vppu}$, τ, n, f_{vppu}, preload_size

1: $(G)\{$
2: init variables;
3: **for** x=0; preload_iters **do**
4: preload batch of data points into local memory
5: Synchronization s_1
6: **for** i=$i_{start,pu}$; $i_{end,pu}$ **do** $\rightarrow$ lockstep execution
7: calculate $H_i^{-1}(y_i)$
8: $e = |H_i|^{-1/2} \exp(-D^2[y_{p,j_{pu}}, y_i, H_i]/2)$;
9: $H_{h,t,pu}$ += $eH_i^{-1}(y_i)$
10: $y_{p,t,pu}$ += $eH_i^{-1}(y_i)y_i$;
11: **end for**
12: Synchronization s_2
13: vPPU-local parallel reduction of $H_{h,t,pu}$ to $H_{h,t}$
14: vPPU-local parallel reduction of $y_{p,t,pu}$ to $y_{p,t}$
15: **end for**
16: **if** $i_{pu} \mod f_{vppu} == 0$ **then**
17: $y_p = H_{h,t}^{-1} y_{p,t}$
18: **end if**
19: $\}$

number of scheduled workitems and no preloading, the global broadcast latencies could be effectively hidden and the preload approach would become contraproductive due to the imposed synchronization mechanisms. These synchronization elements are depicted as s_1 and s_2, they involve atomic integer operations and a semaphore shared among the workgroups workitems. Furthermore, changes in the algorithm's design can easily lead to thread divergence and thus, in the context of s_1/s_2, to barrier divergence [BCD+12], [BD13] as well.
Another critical point of Alg. 20 is the required amount of memory per workgroup. Multiple workgroups can be scheduled for execution on a compute unit, yet it is the amount of inbound workgroups which helps to mask the latencies of memory accesses. The number of inbound workgroups is determined by the available memory of a compute unit, the lower the memory consumption of a workgroup, the more of them can be kept inbound. Thus if one decides to consume too much local memory for preloading, this number will be significantly reduced.
These aspects will become visible within the next section. Yet, from a general perspective Alg. 20 allows one to achieve high throughput in case of memory types with large latencies, e.g. zero-copy host RAM [SGS10], additionally it provides the possibility to implement access patterns to local memory preloading if no local memory broadcasts are available. Furthermore my algorithm can be split across multiple GPUs in a single system.

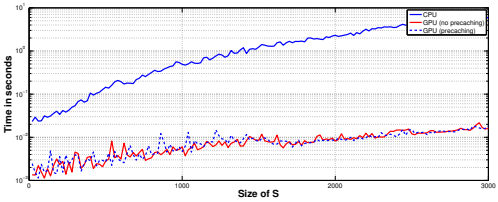


Figure VII.1.: Execution speeds of Alg. 17 and 20 on CPU and GPU. The GPU exhibits a speedup of up to ≈ 176 compared to the execution on CPU side.

VII.3. Results

VII.3.1. Mean Shift

I evaluated the algorithms on a Radeon7970, which provides 32 compute units (PPUs) with a total of 2048 stream cores (PUs), 32KB of local memory per workgroup and 3GB of global memory. Thus each PPU holds 64 PUs. The host system provided a Core-i7 3820 3.6GHz CPU, 64GB RAM and was running Arch-Linux 3.15.5-1. The parameters for the algorithm were identical for all experiments; $\epsilon = 0,\ m_{it} = 100,\ t_d = 10^{-5}$. The choice for $\epsilon = 0$ will become clearer when I explain the applied error measure. Due to the dynamic nature of GPUs, e.g. the temperature dependent behaviour, the results were averaged over 20 repetitions without changing the data. Synthetic samples S of varying sizes were generated, each sample contained 3-dimensional integer tuples.

The graphs in Fig. VII.1 depict the execution speeds for CPU and GPU. It becomes clear that the GPU provides a significant boost of up to ≈ 176 compared to the CPU. Furthermore one can

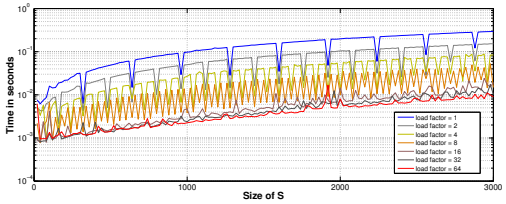


Figure VII.2.: Execution speeds of Alg. 17 and 20 on a GPU with different load factors.

observe that the preloading strategy of Alg. 20 does not yield any improvement. The overhead for caching and synchronization outweighs the potential speed gain. The experiment was not conducted for the CPU as it does not feature any form of shared memory (i.e. OpenCL shared memory requests are translated to global memory requests).

Fig. VII.2 shows the previously discussed effect of underutilization for various load factors. The optimal load factor should be 64 as the GPU provides 64 PUs per PPU, this becomes apparent through the shrinking average distance between consecutive load factors. The execution times seem to converge towards that of $f_{load} = 64$. Another interesting aspect is visualized in the equidistant "drops" along a graph for a single load factor. A close look at the x-axis (Fig. VII.3) reveals that for, e.g. $f_{load} = 1$, a significant boost of execution speed occurs every increment of 64 samples. This is explained by the fact that for $f_{load} = 1$ every compute unit will handle 64 mode approximations, thus at, e.g. $64 \cdot k$, all PUs in exactly k PPUs (if available) will execute the same instruction in lockstep. This allows for coherent memory access in form of broadcasts, i.e. every PU in a PPU will request the same address in lockstep with all other active PUs on the

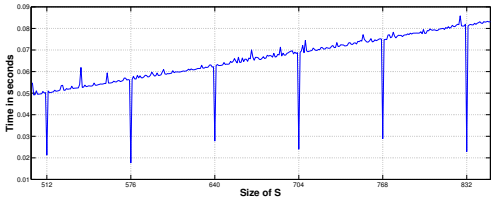


Figure VII.3.: Execution speeds of Alg. 17 on a GPU with a step size of 1 and load factor $f_{load} = 1$. At equidistant positions of 64 elements a significant boost of execution speed occurs. They are the result of favorable memory access patterns in situations of fully utilized PPUs.

GPU. For element counts which are not a multiple of 64, this situation does not occur. The same holds for load factors below 16, above 16 the effect of latency hiding takes over and masks memory access with a huge number of threads. One should note that the effect of underutilization and favorable memory access pattern are two distinct phenomena, which can be seen in Fig. VII.2 where "drop-free" line segments as well as "drops" are positioned significantly apart in a consistent way.

One critical point is the numerical stability of the GPU implementation as is does not utilize the same numerical precision as the CPU version. In order to measure the error between both implementations a simple measure was applied

$$E_{CPU-GPU} := \left(\sum_{i=1}^{|S|} |y_{p,i,CPU} - y_{p,i,GPU}|_{pos} \right) / |S| \quad \text{(VII.8)}$$

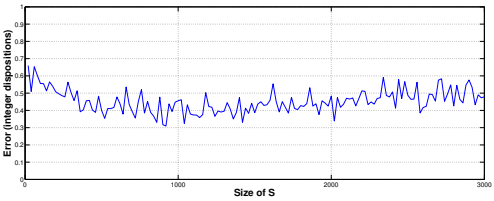


Figure VII.4.: $E_{CPU-GPU}$ for Alg. 17, the maximal error 0.7 clearly shows a difference in numerical behaviour. Yet the error remains negligible for the results even for large samples.

with $|a, b|_{pos}$ defined as

$$|a, b|_{pos} := \sum_{i=1}^{k} |a^i - b^i| \tag{VII.9}$$

being the amount of integer dispositions along all k dimensions. The results are depicted in Fig. VII.4, where it becomes visible that the error, although existing, becomes negligible as it never crosses 0.7 (even for large samples).

VIII. A Generic and Adaptive Approach for Workload Distribution in Multi-Tier Cluster Systems

"The most beautiful experience we can have is the mysterious. It is the fundamental emotion that stands at the cradle of true art and true science." (Albert Einstein, [Ein01])

This chapter introduces a generalization of the vPPU approach, the stated model supports multiple layers and provides an abstraction interface in order to apply different algorithms for specific systems. Additionally the model is arbitrarily recursive and thus applicable to any distributed heterogeneous system. The following sections incrementally define and motivate the model through application examples, first I will discuss how the vPPU model can be used in order to accelerate the training of small neural networks. This example points out a major aspect of organic computing and parallelization, even with deep understanding of the involved parallel architectures, e.g. a single Single Instruction Multiple Thread (SIMT) GPU like the NVidia GTX980, the developed algorithms will always be susceptible to effects such

as underlying compiler optimizations, driver schedule algorithms or host system latencies. Compiler optimizations can be turned off, yet this in turn makes in-depth knowledge of the specific GPU architecture mandatory. The scheduling strategy of a device driver poses an unpredictable element, this problem can be addressed by fixing the utilized driver. A similar thought holds for the host system with the additional challenge of predicting the system load and thus predicting the process list including each processes operations over time. Besides that, my results illustrate the interplay between the development of an optimized algorithm for a given problem and its parametrization. I will explain how structural attributes of the problem can influence the algorithm design, e.g. when to parallelize over the network weights and when over the data. In relation to organic computing this dynamic represents an example for how an algorithm (in this case represented by an external developer) could create a subalgorithm for specific problems. Furthermore it illustrates the limited insight into a given system.
The second example addresses the well known problem for distributed matrix multiplication, while the previous discussion focused on the development of a local algorithm, this example addresses the challenge of distributed computation over multiple levels. I discuss ways of predicting the behaviour of a distributed system without a specific model, in other words, I discuss methods of how to find the optimal parameters for a given family of algorithms.

VIII.1. Introduction to Neural Networks and Previous Work

The recent trend in neural networks towards deep architectures shifted the interest in training algorithms to situations which provide huge networks, e.g. convolutional neural networks with many large layers, and equally huge amounts of data, e.g. terabytes of image data. It is debatable if such large networks inherently represent a new tier of generalization performance (see [NYC14],[GSS14] and [HI15]), [BC14] provides an interesting comparison between deep and shallow neural networks. In this chapter I explain how the strategy described in [MH14b] can be applied for the training of small to mid range neural networks. Additionally I show that it outperforms previous (canonical) approaches, e.g. [OFJ09],[CWL+14], which rely on optimized Basic Linear Algebra Subprograms (BLAS)BLAS libraries or [RIdONF],[VN14b], which naively deploy thread grids, in terms of efficiency and scalability on parallel architectures. Besides the efficiency of my approach it also features the capability of being applicable to a large set of training algorithms, namely those which utilize the concept of backpropagation. My algorithms do not only increase the efficiency of neural network training but also of network evaluation. With the rise of highly performant embedded GPUs this can accelerate neural networks, e.g. on cell phones [BN02].
It is difficult to define general size terms for neural networks as they usually depend on the context of application, e.g. regarding the recognition of hand gestures one can obtain sufficient results with a network consisting of one hidden layer with ≈ 50 neurons [KMGH14]. Whereas in the field of object classification one thinks in terms of deep networks, which consist of multiple hid-

den layers, each with several 100 neurons [DCM+12], [RMN09]. In this thesis the term *small scale neural network* is defined as a network which contains $1-2$ layers each with $10-50$ and *mid scale neural network* as a network with $3-10$ layers each with $10-500$ (except for the input layer, which may contain many more elements). The amount of training samples is usually far greater than the input dimension and especially as the neuron count in each layer. Yet even for situations in which this assumption does not hold my model provides methods for a dynamic (run-time) adaptation in order to distribute the workload efficiently among multiple parallel computation units.
In order to keep the description simple I omit the details of including a bias term, which can be done through a simple parameter inclusion into the described algorithms (i.e. matrix expansion). Furthermore the listings can be easily adapted for different training algorithms.
Section VIII.2 defines all basic terms for the remaining chapter and provides a short introduction into the training of neural networks. In Section VIII.3 I introduce the concept of shadow networks, which play an integral role within my model. Section VIII.4 briefly describes the concept of virtual parallel processing units and extends it with a fusion approach in section VIII.5. The remaining algorithmic elements are only illustrated in section VIII.6.2 since they built upon previously described concepts. Section VIII.7 describes implementation details and discusses memory requirements. The results and a corresponding discussion are presented in sections VIII.8 and VIII.9, respectively.

VIII.2. Preliminary Definitions

In order to keep the techniques as general applicable as possible I purely consider feed-forward neural networks, i.e. the connections from a neuron in layer i lead only to neurons in layer $i+1$. Yet the methods can easily be applied to other fields of application, e.g. to recurrent networks, as long as the backpropagation framework is being used (see backpropagation through time [Wer90]). A neural network is defined as follows

Definition 2. *A neural network N with n layers is an acyclic directed graph $N = (V, E)$ with edge labels $W : E \to \mathbb{R}$ which fulfills: There is a partition $V = \biguplus_{i=1}^{n} V_i$ where $E = \{(v_i, v_j) | v_i \in V_l \wedge v_j \in V_{l+1}\}$. V_1 and V_n are called the input and output layer, respectively. Each vertex v_i is associated with a differentiable activation function $g_i : \mathbb{R} \to \mathbb{R}$ and a stimulus $a_i \in \mathbb{R}$.*
In the context of machine learning each vertex is called an artificial neuron and V_l is regarded as a layer of neurons, a v_i's corresponding layer V_l is indicated by a superscript $v_{\tilde{i}}^l$, by noting the layer one can enumerate the neurons relative to their layer, i.e. $0 \leq \tilde{i} \leq |V_l|$. Furthermore one can write $w_{\tilde{i},\tilde{j}}^l$ instead of $W((v_i, v_j))$ while simultaneously indicating v_i's layer with l (note the relative indices). The symbols $a_{\tilde{i}}^l$ and $g_{\tilde{i}}^l$ are defined analogously.

Remark VIII.1. *Note that the tilded index $\tilde{i}$ of a neuron v_i indicates an enumeration relative to its layer l, thus the relative index and layer are provided by sub- and superscript $v_{\tilde{i}}^l$, respectively.*

One can view the network as a standard transport network

with following semantics: network flow is accumulated at v_i^l via

$$a_{\tilde{i}}^l := \sum_{h:(h,i)\in E} W((v_h, v_i)) \cdot g_h(a_h) \tag{VIII.1}$$

Given a finite set of training data $D_{train} = \{(x,y)|x \in \mathbb{R}^{|V_1|} \wedge x \in \mathbb{R}^{|V_n|}\}$ and viewing the network as a differentiable function $f : \mathbb{R}^{|V_0|} \to \mathbb{R}^{|V_n|}$ the goal is to find a function W which succeeds in minimizing the loss $L_{N,D_{train}}$ defined by some arbitrary differentiable function $L : \mathbb{R}^{|V_0|} \times \mathbb{R}^{|V_n|} \to \mathbb{R}$, which in turn measures the difference between $f(x)$ and y for each $(x,y) \in D_{train}$. Thus the definition $L_{N,D_{train}} := \frac{1}{e} \sum_{(x,y)\in D_{train}} L(x,y)$.
Backpropagation is a technique of calculating the gradient $\partial L / \partial w_{\tilde{i},\tilde{j}}^l(x)$ for $(x,y) \in D_{train}$ and is often confused with actual algorithms such as vanilla backpropagation [Wer90] or ARC-Prop [Bai15]. From now on (w.l.o.g.) I will restrict myself to the canonical loss function

$$L_c(x,y) := \frac{1}{2}|f(x) - y|^2 \tag{VIII.2}$$

Given a sample (x,y) backpropagation calculates the corresponding gradient by executing the following steps:

1. Propagate x through the network and obtain y'
2. Calculate $\delta_{\tilde{i}}^{n-1} := g'^{n-1}_{\tilde{i}}(a_{\tilde{i}}^{n-1})(y_{\tilde{i}} - y'_{\tilde{i}})$ (with y_i being the vector elements of y)
3. Calculate $\delta_{\tilde{i}}^l$ for $0 < l < n-1$ by

$$\delta_{\tilde{i}}^l = g'^l_{\tilde{i}}(a_{\tilde{i}}^l) \cdot \sum_{j:(i,j)\in E} \delta_{\tilde{j}}^{l+1} \cdot w_{\tilde{i},\tilde{j}}^l \tag{VIII.3}$$

The steps so far are the reason for the methods name, since one propagates the δ values from the highest layer back to the second layer. Once the deltas have been calculated one continues to extract the gradient via

$$\frac{\partial L}{\partial w^l_{(\tilde{i},\tilde{j})}}(x) := \delta^{l+1}_{\tilde{j}} \cdot g^l_{\tilde{i}}(a^l_{\tilde{i}}) \tag{VIII.4}$$

A training algorithm usually updates the label function in a simple gradient descent with step size (learning rate) η by

$$w_{(\tilde{i},\tilde{j})} = w_{(\tilde{i},\tilde{j})} - \eta \cdot \frac{\partial L}{\partial w^l_{\tilde{i},\tilde{j}}}(x) \tag{VIII.5}$$

The update rules vary greatly among the different algorithms, yet all of them require the same basic computation of the gradient values. It is common to average the gradient over batches or mini-batches (a batch is a subset $D'_{train} \subseteq D_{train}$) before updating the weights ($|D'_{train}|$ is called an epoch).

VIII.3. Linear Algebra and Shadow Networks

In order to describe my motivation I will briefly elaborate on how to express the backpropagation technique in terms of matrix operations. Let $D_{x,train}$ be the set which contains only the x-elements ($x \in \mathbb{R}^m$) of each D_{train} member and let S be the sample matrix

$$S := \begin{pmatrix} x^1 & x^2 & x^3 & \dots & x^s \end{pmatrix} \tag{VIII.6}$$

where each column $x^i \in D_{x,train}$. The weights between layers $1, 2$ are enumerated with respect to some order O of the neurons

in layer 2. In other words, one creates a list of weights, e.g. by first listing all weights $(w^1_{\tilde{1},\tilde{1}}, w^1_{\tilde{2},\tilde{1}}, ..., w^1_{\tilde{m},\tilde{1}})$, i.e. all weights which lead to neuron $v^2_{\tilde{1}}$. These weights are followed by all weights which lead to neuron $v^2_{\tilde{2}}$ etc. O defines the order of the neurons in the top layer, in this example $O = (1, 2, ...)$. Viewing this list as a row-major linearized matrix one obtains

$$W := \begin{pmatrix} w^1_{\tilde{1},\tilde{1}} & w^1_{\tilde{2},\tilde{1}} & ... & w^1_{\tilde{m},\tilde{1}} \\ w^1_{\tilde{1},\tilde{2}} & w^1_{\tilde{2},\tilde{2}} & ... & w^1_{\tilde{m},\tilde{2}} \\ ... & ... & ... & ... \\ w^1_{\tilde{1},O(\tilde{k})} & w^1_{\tilde{2},O(\tilde{k})} & ... & w^1_{\tilde{m},O(\tilde{k})} \end{pmatrix} \tag{VIII.7}$$

with k being the number of neurons in layer 2. The product $W \cdot S$ will provide all layer 2 stimuli (i.e. sum values) for all samples

$$A := W \cdot S = \begin{pmatrix} a^{2,1}_{\tilde{1}} & a^{2,2}_{\tilde{1}} & ... & a^{2,s}_{\tilde{1}} \\ a^{2,1}_{\tilde{2}} & a^{2,2}_{\tilde{2}} & ... & a^{2,s}_{\tilde{2}} \\ ... & ... & ... & ... \\ a^{2,1}_{O(\tilde{k})} & a^{2,2}_{O(\tilde{k})} & ... & a^{2,s}_{O(\tilde{k})} \end{pmatrix} \tag{VIII.8}$$

the superscript $(2, 1)$ in $a^{2,1}_{O(\tilde{k})}$ indicates layer 2 and sample 1. This illustration for layers $1, 2$ can be transferred analogously to arbitrary layers $1 < l < n$ with the difference that the output of layer l does not consist of samples but of activation values $g^l_{\tilde{i}}(a^{l,1}_{\tilde{i}})$.

Definition 3. *Let k_l be the number of neurons in layer l, s the amount of samples to be propagated through the network, O an order of layer l and $W^l \in \mathbb{R}^{k_{l+1} \times k_l}$, $S^l \in \mathbb{R}^{k_l \times s}$, $A^{l+1} \in \mathbb{R}^{k_{l+1} \times s}$*

matrices of the form

$$W^l := \begin{pmatrix} w^l_{\tilde{1},O(\tilde{1})} & w^l_{\tilde{2},O(\tilde{1})} & ... & w^l_{\tilde{k_l},O(\tilde{1})} \\ w^l_{\tilde{1},O(\tilde{2})} & w^l_{\tilde{2},O(\tilde{2})} & ... & w^l_{\tilde{k_l},O(\tilde{2})} \\ ... & ... & ... & ... \\ w^l_{\tilde{1},O(\tilde{k_{l+1}})} & w^l_{\tilde{2},O(\tilde{k_{l+1}})} & ... & w^l_{\tilde{k_l},O(\tilde{k_{l+1}})} \end{pmatrix} \quad \text{(VIII.9)}$$

$$S^l := \begin{pmatrix} g^l_{\tilde{1}}(a^{l,1}_{\tilde{1}}) & g^l_{\tilde{1}}(a^{l,2}_{\tilde{1}}) & ... & g^l_{\tilde{1}}(a^{l,s}_{\tilde{1}}) \\ g^l_{\tilde{2}}(a^{l,1}_{\tilde{2}}) & g^l_{\tilde{2}}(a^{l,2}_{\tilde{2}}) & ... & g^l_{\tilde{2}}(a^{l,s}_{\tilde{2}}) \\ ... & ... & ... & ... \\ g^l_{\tilde{k_l}}(a^{l,1}_{\tilde{k_l}}) & g^l_{\tilde{k_l}}(a^{l,2}_{\tilde{k_l}}) & ... & g^l_{\tilde{k_l}}(a^{l,s}_{\tilde{k_l}}) \end{pmatrix} \quad \text{(VIII.10)}$$

$$A^{l+1} := W \cdot S = \begin{pmatrix} a^{l+1,1}_{O(\tilde{1})} & a^{l+1,2}_{O(\tilde{1})} & ... & a^{l+1,s}_{O(\tilde{1})} \\ a^{l+1,1}_{O(\tilde{2})} & a^{l+1,2}_{O(\tilde{2})} & ... & a^{l+1,s}_{O(\tilde{2})} \\ ... & ... & ... & ... \\ a^{l+1,1}_{O(\tilde{k_l})} & a^{l+1,2}_{O(\tilde{k_l})} & ... & a^{l+1,s}_{O(\tilde{k_l})} \end{pmatrix} \quad \text{(VIII.11)}$$

VIII.3.1. Shadow Networks

Shadow Networks are an integral part in my method as without them it would not be possible to distribute computation efficiently on large scale structures. Before the formal definition let us view two consecutive and fully connected layers V_l, V_{l+1} in a given neural network (see Fig. VIII.1), two shadows of this network are obtained by arbitrarily splitting V_l into 2 parts (see Fig. VIII.2). This results in two networks with a different structure between layer l and $l+1$, which effectively reduces the dimensionality of all samples propagated between both layers in the two shadows.

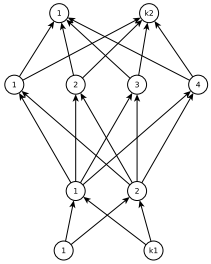


Figure VIII.1.: A network with four fully connected layers, two hidden layers V_2, V_3, each with 2 or 4 neurons respectively, the layer below V_2 has $k1 = 2$ neurons while the layer above V_3 has $k2 = 2$ neurons

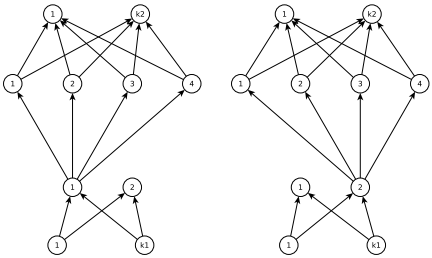


Figure VIII.2.: Two shadows of the network from Fig. VIII.1, the left shadow network only keeps the connections between neuron 1 in layer V_2 and all neurons in the layer above, while the second shadow network contains only the connections between neuron 2 and all neurons in the layer V_3.

Definition 4. *A split* $f_l := \{V \subseteq V_l\}$ *for a layer* V_l *with* $1 \leq l < n$ *is a partition* $V_l = \biguplus_{V \in f_l} V$. *Any neural network carries*

a canonical set of splits (one for each layer), where each split is defined as the corresponding layer itself. These splits are called canonical splits.

Remark VIII.2. *Although a split can be any partition I restrict the discussion to symmetric splits, i.e.* $|v_i| = |v_j|$ *for all* $v_i, v_j \in f_l$*. If not defined explicitly a layers split is considered to be the canonical split. Thus a set of splits is considered to contain exactly one split for each layer except the last one, which per definition is not associated with any split.*

Definition 5. *Let* N *be a neural network with* n *layers and* F *a set of splits. A shadow network* $N_{l,V}$ *of* N *is a network with* n *layers which are identical to those of* N *except for* V_l*, which is replaced by an arbitrary* $V \in f_l$*, the connections between* V *and* V_{l+1} *are given by* $E' := \{(v_i, v_j) | v_i \in V \wedge v_j \in V_{l+1} \wedge (v_i, v_j) \in E\}$*. For* f_l *the projection* P_l *is defined as* $\{N_{l,V} | V \in f_l\}$*.*

VIII.4. Complexity, Parallel Architectures and Virtual Parallel Processing Units

Propagating a batch of samples through a network can be expressed as a simple matrix multiplication, this already indicates the overall time complexity to be of $\mathcal{O}(k_{l+1} \cdot k_l \cdot s + 2 \cdot k_{l+1} \cdot s)$ for each set of two consecutive layers, the last term is a result of computing the activation values (and corresponding derivatives) separately. In case of an integrated computation one obtains $\mathcal{O}(k_{l+1} \cdot k_l \cdot s + k_{l+1} \cdot s)$. Since there are $n - 1$ such layers one obtains $\mathcal{O}(k_{l+1} \cdot k_l \cdot s \cdot (n - 1))$. One must note that such an expression is only a rough estimate as it omits the complexity

of multiplication, addition, memory accesses, computation model etc. Yet on the other hand one must note that firstly the amount of elements to compute does not change regarding the computation platform and secondly might require different optimizations which can't be foreseen at this point. Furthermore one might encounter instances of backpropagation which are simply to small to outweigh the management overhead or fit the workload distribution model. Thus I decided to fully linearize the problem and to embed it in an algorithmic framework, which allows an abstract optimization with respect to the underlying (system) computation model.
Another motivation for an abstract embedding was that not every researcher or engineer would go as far as customizing an already existing library for efficient matrix multiplication, as it might likely loose its efficiency, be difficult in terms of code complexity or simply be impossible due to lack of source code. It will most likely be easier to variate model parameters in order to increase the efficiency for an unknown platform. Furthermore system drivers might change and render highly optimized implementations useless until the next update arrives. The following sections will show that my approach outperforms optimized matrix methods by NVidia, AMD or Intel.
Yet, one still has to account for the backpropagation part. Once the batch has been propagated through the network, the following tasks remain: 1) calculate the errors 2) propagate these values back and 3) update the gradient, which exhibit complexities of $\mathcal{O}(k_n \cdot s)$, $\mathcal{O}(k_{l+1} \cdot k_l \cdot s \cdot (n-1))$ and $\mathcal{O}(k_{l+1} \cdot k_l \cdot s \cdot)$, respectively (note that the third part can be fused with the second).
With respect to large parallel computation systems one is tempted to simply distribute fragments onto single devices, yet one has

to consider elements such as communication latencies or memory limits. Such system details naturally induce groups of devices, onto which larger workload fragments will be distributed. The framework builds upon such predefined system groups and is motivated by the results of [MH14b].
The computational complexity of backpropagation is mainly characterized by three elements, the batch size, the neuron count in each layer and the amount of layers. I will apply the concept of vPPUs (see Chap. VII) in order to process a fully linearized neural network with respect to its training data.

VIII.5. Distributing the Computation on GPUs

A modern GPU features many (i.e. several thousand) so called shader units, in the context of general purpose computing these units are referred to as stream cores (AMD GPUs) or streaming processors (NVIDIA). Despite the high number of units it still presents a severe challenge to fully utilize them as they follow the SIMT concept ([RR12]). The hardware model, common to both vendors, groups the computation units into *compute units* or *streaming multiprocessors* for AMD or NVidia respectively. The OpenCL model maps this into a similar form by grouping threads into work groups, these workgroups will later be scheduled for execution on the previously described groups of computation units. In this section I will describe my approach by starting with a naive method and extend it through the PPU model, for now let us define a PU to be a single computation unit (or thread of execution) and an OpenCL workgroup to be a PPU.

VIII.5.1. A General Approach

Computing a single element of A^{l+1}, e.g. $a^{l+1,1}_{\tilde{O(1)}}$, requires a single dot product of complexity $\mathcal{O}(k_l)$. If one schedules each PU to compute one row of A^{l+1} (i.e. s dot products), the corresponding OpenCL program looks structurally as depicted in listing 21 (array indexing is omitted in order to keep the listing readable)

Algorithm 21 *computeDot* for rows of A^{l+1}

Input: Linearized matrices $W^l, S^l, l, k_l, k_{l+1}, s$

```
1:  (P_{i_PU,c,t,j}){
2:  i_PU = getPUIdx();
3:  //In case one can only start PUs in multiples of τ
4:  if i_PU ≥ k_{l+1} then
5:      return;
6:  end if
7:  c = s;
8:  t = 0;
9:  for j = 0; c do
10:     for i = 0; k_l do
11:         t = t + S_{i,j} · W_{i_PU,i};
12:     end for
13:     a^{l+1,j}_{ĩ_PU} = g^{l+1}_{ĩ_PU}(t);
14:     t = 0;
15: end for
16: }
```

With slight modifications this algorithm can also be used for calculating A^{l+1} not row by row but column by column, one simply has to set $c = k_{l+1}$ and modify the schedule.

Theorem 6. *Let there be z PPUs each with τ PUs. Under the*

assumption that z PPUs can be executed in parallel, the complexity of Alg. 21 is $\mathcal{O}(\frac{k_{l+1}}{z\tau} \cdot k_l \cdot s)$.
Furthermore following statements hold:

- *If each PU is scheduled to compute one column of* A^{l+1} *the complexity remains unchanged.*
- *The workload for each PU is* $\mathcal{O}(s \cdot k_l)$ *or* $\mathcal{O}(k_{l+1} \cdot k_l)$ *if the PUs are scheduled over rows or columns, respectively.*
- *In case of sequential linear addressing one obtains* A^{l+1} *in row-major or column-major form if the PUs are scheduled over rows or columns, respectively.*

Proof. Computing a row of A^{l+1} corresponds to calculate the dot products between a single row of W^l and all columns of S^l ($\mathcal{O}(k_l \cdot s)$), thus the complexity for computing all rows of A^l is $\mathcal{O}(k_{l+1} \cdot k_l \cdot s)$. Since $z\tau$ PUs work in parallel, each on one row, it results in a factor of $\frac{1}{z\tau}$. Computing a column (activation values for a single sample for all neurons in layer V^{l+1}) has a complexity of $\mathcal{O}(k_{l+1} \cdot k_l)$, since there are s samples the same complexity is obtained. The workload for each PU is obviously defined by the inner loops, which also determine the memory alignment in case of sequential linear addressing. □

This theorem implicates that:

- Should the neuron count in layer V^{l+1} be larger than the sample count s it will reduce the workload for each PU if it computes a row of A^{l+1}
- Should the neuron count in layer V^{l+1} be smaller than the sample count s it will reduce the workload for each PU if it computes a column of A^{l+1}

Note that the overall complexity remains unchanged, yet with less workload for each PU the overhead of for-loop iterations can be significantly reduced. Furthermore it spreads the work onto more threads, which in turn can be beneficial by masking memory latencies when accessing slow external memory.

Theorem 7. *If $s < k_{l+1}$ then the for-loop overhead is reduced by a factor of $\frac{k_{l+1}}{s} > 1$ compared a column-wise schedule. If $s > k_{l+1}$ then the for-loop overhead is reduced by a factor of $\frac{s}{k_{l+1}} > 1$ compared a row-wise schedule.*

With this theorem one can reduce the computation overhead yet one has to start more PUs and thus PPUs (which poses a dilemma as described in the next section). More PPUs can induce severe overhead in terms of low-level (i.e. hardware / driver) overhead.

VIII.5.2. Fused vPPUs

Despite the fact of many thousand PUs being available the data amount will usually outgrow this number and prevent a truly parallel execution. For these situations modern GPUs allow to schedule more PPUs (and thus PUs) than can actually be executed simultaneously. This introduces new aspects into the algorithm's design: On one hand we can rigorously think in the previously described scenario of having an unlimited amount of PUs available, on the other hand; the more the desired PU count exceeds the actual physical PU count one has to expect a larger efficiency drop due to physical scheduling mechanisms and memory latencies. A crucial element are the so called inbound workgroups which, in my model, can be described as PPUs of which the GPU is aware, i.e. of which the GPU knows what memory

segments have been requested. In other words one has to be careful regarding the number of PUs to utilize with respect to the complexity of a PUs program.
In order to find the optimal scheduling structure I applied the concept of virtual PPUs (vPPUs) in combination with a fusion approach. In order to spread Alg. 21 onto more threads (which is useful for small problem sizes in order to increase the GPU utilization) I define multiple vPPUs within a PPU. Thus in my model it is assumed that all vPPUs are of equal size μ which fulfills $\mu|\tau$ and $\mu|s$

Alg. 22 will spread the outer for-loop onto μ threads of which each will compute $\frac{s}{\mu}$ iterations or elements in one row of A^{l+1}. One would be tempted to assume that starting as many threads as possible (with respect to the data size) would increase the efficiency, yet I will later show that the previously described overhead penalties dictate different scheduling structures. These scheduling structures can not be induced by common matrix multiplication approaches.
All vPPUs of a certain PPU have access to a small set of PPU local resources, e.g. a small but fast cache of 40kB. Yet in order to utilize these limited resources one has to reduce the local problem size, e.g. one usually can not fit the required parts of W^l into this small memory (leaving out the overhead for preloading this data). Another limitation of the model is the fusion of vPPUs over different but adjacent PPUs, as so far we can calculate one row or column only on the same PPU (one row or column per vPPU, which is smaller than the PPU).
Fusing vPPUs can be done by a few simple modifications of Alg. 22, the result is depicted int Alg. 23 Note that Alg. 23 allows a transparent fusion of vPPUs on different PPUs, yet the preloading of data can induce a high penalty Ω since many threads will

Algorithm 22 *computeDot′* for rows of A^{l+1}

Input: Linearized matrices $W^l, S^l, l, k_l, k_{l+1}, s, \mu$

1: $(P_{i_{PU},c,t,j,i_{vPU},i_{vPPU},a,b,i})\{$
2: i_{PU} = getPUIdx();
3: $i_{vPU} = i_{PU} \mod \mu$;
4: $i_{vPPU} = i_{PU} DIV \mu$;
5: //In case one can only start PUs in multiples of τ
6: **if** $i_{vPPU} \geq k_{l+1}$ **then**
7: return;
8: **end if**
9: $c = s/\mu$;
10: $t = 0$;
11: $a = i_{vPU} \cdot c$
12: $b = (i_{vPU} + 1) \cdot c$
13: **for** $j = a$; b **do**
14: **for** $i = 0$; k_l **do**
15: $t = t + S_{i,j} \cdot W_{i_{vPU},i}$;
16: **end for**
17: $a^{l+1,j}_{\tilde{i}_{vPPU}} = g^{l+1}_{\tilde{i}_{vPPU}}(t)$;
18: $t = 0$;
19: **end for**
20: $\}$

Algorithm 23 *computeDot''* for rows of A^{l+1}

Input: Linearized matrices $W^l, S^l, l, k_l, k_{l+1}, s, \mu, f$

1: $(P_{i_{PU},c,t,j,i_{vPU},i_{vPPU},a,b,i})\{$
2: i_{PU} = getPUIdx();
3: $i_{vPPU} = (i_{PU} DIV \mu) DIV f$;
4: $i_{vPU} = i_{PU} \mod (f \cdot \mu)$;
5: //In case one can only start PUs in multiples of τ
6: **if** $i_{vPPU} \geq k_{l+1}$ **then**
7: return;
8: **end if**
9: Preload W^l fragment into local memory ω
10: $c = s/(\mu \cdot f)$;
11: $t = 0$;
12: $a = i_{vPU} \cdot c$
13: $b = (i_{vPU} + 1) \cdot c$
14: **for** $j = a$; b **do**
15: **for** $i = 0$; k_l **do**
16: $t = t + S_{i,j} \cdot \omega_{0,i}$;
17: **end for**
18: $a^{l+1,j}_{\tilde{i}_{vPPU}} = g^{l+1}_{\tilde{i}_{vPPU}}(t)$;
19: $t = 0$;
20: **end for**
21: }

be involved and the required code structure might grow fast in complexity (esp. when fusing vPPUs over multiple PPUs).

Theorem 8. *In order to carry the required fragment of W^l each fused group of vPPUs needs memory space to hold k_l numeric elements. Let $\alpha \in \mathbb{N}$ be the amount of local memory in terms of numeric elements. The maximal amount of fused vPPUs per PPU is $\lfloor \frac{\alpha}{k_l} \rfloor$.*

The overall complexity of Alg. 23 is given in following

Theorem 9. *Alg. 23 exhibits a complexity of $\mathcal{O}(\frac{k_{l+1}}{z\tau} \cdot k_l \cdot \frac{s}{f\mu} \cdot \beta(\tau', z') + \frac{z\tau}{f\mu} \cdot \Omega(k_l, f, \mu) + \aleph(k_l, f, \mu))$. With $\beta \geq 1$ being a scheduling penalty (parameterized by the physical amount PU count τ' per physical PPU and the amount of physical PPUs z') and $\aleph$ a penalty term in case that $f\mu \nmid \zeta\tau$ for $\zeta \in \mathbb{N}$ (i.e. penalty if vPPU fusion reaches into another PPU but not to its end)*

Proof. There are still only $z\tau$ (non-physical) PUs which calculate the elements, of which there are $k_{l+1} \cdot k_l \cdot s$. Due to the limited amount of physical PUs the efficiency will be scaled according to β. Furthermore the outer for-loops iteration count was reduced by a factor of $f\mu$. The second term accounts for memory latencies during the preloading of W^l fragments, $\Omega(k_l, f, \mu)$ represents the overhead for loading one such fragment by a group of fused vPPUs, of which there are $\frac{z\tau}{f\mu}$ in total. The last term accounts for the lost efficiency if a group of fused vPPUs reaches into another PPU but does not cover it completely, i.e. the efficiency will be lost as only a few vPPUs will benefit from the preloaded data. □

Remark VIII.3. *In practical scenarios it generally does not hold that $(\tau = \tau' \wedge z = z') \Rightarrow \beta \equiv 1$ as one has to account for*

structural hardware restrictions, e.g. limited amount of registers for PUs in a PPU.

With Alg. 23 one also obtains a new expression for the efficiency of the internal for-loops.

Theorem 10. *If* $\frac{s}{\mu f} < k_{l+1}$ *then the for-loop overhead is reduced by a factor of* $\frac{k_{l+1}\mu f}{s} > 1$ *compared a column-wise schedule. If* $\frac{s}{\mu f} > k_{l+1}$ *then the for-loop overhead is reduced by a factor of* $\frac{s}{k_{l+1}\mu f} > 1$ *compared a row-wise schedule.*

Note that this theorem is independent of any scheduling penalty β, yet the for-loops influence the scheduling (and thus β) as the amount of iterations is proportional to the amount of global memory accesses.

Remark VIII.4. *A higher efficiency of for-loops implies a smaller amount of global memory accesses in a vPPU, yet also reduces the gain by using local memory.*

VIII.5.3. Dimension Splits, Shadow Networks and Implementation

Before explaining the benefit of shadow networks I will summarize what was achieved so far. With vPPUs we obtained the ability to spread the row- or column-wise computation of A^{l+1} onto multiple threads, which helps to increase the efficiency of internal for-loops and hides memory latencies through a larger amount threads. By fusing vPPUs into groups I introduced the possibility of using local memory in order to cache fragments of W^l, furthermore the computation was spread onto even more threads. Yet this last step requires a delicate balance between scheduling penalty, preloading overhead and for loop-efficiency.

One should note that in Alg. 21, 22 and 23 all attempts of increasing the loop efficiency targeted the outer loop, which iterates over all neurons (A^l row-wise) or samples (A^{l+1} column-wise). Using any of these algorithms one can also compute $N_{l,V}$ by restricting the inner loop on the indices of V. Let $A^{l+1,V}$ be the stimulus matrix of layer V_l in $N_{l,V}$, i.e. the matrix which contains the stimuli but not the activation values, this matrix can be computed by Alg. 24

This corresponds to splitting the dot product $(w^l_{\tilde{1},O(\tilde{1})}\ w^l_{\tilde{2},O(\tilde{1})}\ \ldots\ w^l_{\tilde{k}_l,O(\tilde{1})}) \cdot (g^l_{\tilde{1}}(a^{l,1}_{\tilde{1}})\ g^l_{\tilde{2}}(a^{l,1}_{\tilde{2}})\ \ldots\ g^l_{\tilde{k}_l}(a^{l,1}_{\tilde{k}_l}))$ into two smaller dot products, one can view this a a temporary dimensional reduction. With shadow networks one obtains

Theorem 11. *Alg. 24 exhibits a complexity of* $\mathcal{O}(\frac{k_{l+1}}{z\tau} \cdot \frac{k_l}{|f_l|} \cdot \frac{s}{f\mu} \cdot \beta(\tau', z') + \frac{z\tau}{f\mu} \cdot \Omega(k_l, f, \mu) + \aleph(k_l, f, \mu))$. *Furthermore the for-loop efficiency is increased by a factor of* $|f_l|$ *compared to the computation of* A^{l+1}.

Proof. Follows directly from Alg. 23 since the only difference is the reduced iteration amount of the inner for-loop. □

In order to obtain the activation matrix A^{l+1} one has to compute $\tilde{A}^{l+1} := \sum_{V \in f_l} A^{l+1,V}$ and $g^l_{\tilde{1}}(a^{l,j}_{\tilde{i}})$ for all elements $t^{l,j}_{\tilde{i}}$ of $\tilde{A}^{l+1}$. This is accomplished by Alg. 25, which schedules one PU per element in A^{l+1} and assumes a row-wise linearization.

Obviously Alg. 25 induces an additional complexity of $\mathcal{O}(s \cdot k_{l+1} \cdot |f_l|)$, which implies

Theorem 12. *Using Alg. 24 to compute* A^{l+1} *induces a complexity of* $\mathcal{O}(\frac{k_{l+1}}{z\tau} \cdot \frac{k_l}{|f_l|} \cdot s \cdot \beta(\tau', z') + \frac{z\tau}{f\mu} \cdot \Omega(k_l, f, \mu) + \aleph(k_l, f, \mu) + s \cdot k_{l+1} \cdot |f_l|)$.

Algorithm 24 *computeDot''* for rows of $A^{l+1,V}$

Input: Linearized matrices $W^l, S^l, l, k_l, k_{l+1}, s, \mu, f$, index set V

1: $(P_{i_{PU},c,t,j,i_{vPU},i_{vPPU},a,b,i})\{$
2: i_{PU} = getPUIdx();
3: $i_{vPPU} = (i_{PU} DIV \mu) DIV f$;
4: $i_{vPU} = i_{PU} \mod (f \cdot \mu)$;
5: //In case one can only start PUs in multiples of τ
6: **if** $i_{vPPU} \geq k_{l+1}$ **then**
7: return;
8: **end if**
9: Preload W^l fragment into local memory ω
10: $c = s/(\mu \cdot f)$;
11: $t = 0$;
12: $a = i_{vPU} \cdot c$
13: $b = (i_{vPU} + 1) \cdot c$
14: **for** $j = a$; b **do**
15: **for** $i \in V$ **do**
16: $t = t + S_{i,j} \cdot \omega_{0,i}$;
17: **end for**
18: $a^{l+1,j}_{i_{vPPU}} = t$;
19: $t = 0$;
20: **end for**
21: $\}$

Algorithm 25 *mergeDot* for (row-wise) shadow stimuli $\tilde{A}^{l+1}$

Input: Set of linearized matrices $\{A^{l,i}\}$,l, k_l,k_{l+1}, s,f_l

1: $(P_{i_{PU},t,i,j,k})\{$
2: i_{PU} = getPUIdx();
3: //In case one can only start PUs in multiples of τ
4: **if** $i_{PU} \geq k_{l+1}s$ **then**
5: return;
6: **end if**
7: $t = 0$;
8: $i = i_{PU} DIV s$;
9: $j = i_{PU} \mod s$;
10: **for** $k = 0$; $|f_l|$ **do**
11: $t = t + \tilde{A}^{l+1,k}_{i,j}$;
12: **end for**
13: $a^{l+1,j}_{\tilde{i}} = g^{l+1}_{\tilde{i}}(t)$;
14: $\}$

Although only a few for-loop iterations are expected in Alg. 25, the vPPU model could still be applied, as it is possible to approach the matrix merge with a parallel reduction. For this work I did not evaluate this possibility since the algorithm's performance was sufficient.

VIII.6. Backpropagation, Delta Calculation and Weight Updating

VIII.6.1. Transposed Results

The developed algorithm essentially performs a matrix multiplication with included activation computation, yet during forward propagation it might happen that the memory alignment of the serialized result A^{l+1} changes, e.g. from row to column major. This is addressed in following

Lemma 3. *It holds that*

1. *Should each fused vPPU in Alg. 24 be fixed on a single column of A^l then A^{l+1} will be stored in column major alignment, this also corresponds to applying the efficiency rule for the case $k_l < s$.*
2. *In the case of 1) the memory layout of A^{l+1} corresponds to a row-major aligned ${A^{l+1}}^T$ (i.e. a change occurs).*
3. *In the case of 1) Alg. 24 must consider the problem $(A^T W)^T$ in order to uphold the correctness and $A^T W$ to uphold the efficiency statements for the case $W \cdot A$*
4. *Should in case of 3) $A^T W$ induce a new alignment change then it will rectify the previous change, i.e. $(A^T W)^T$ will*

be stored in row-major order. Otherwise it will be saved in column-major order, which corresponds to $A^T W$ in row major order

Which gives rise to following

Theorem 13. *Alg. 24 can be applied for forward propagation even in the case of memory alignment changes. It suffices to keep track of changes in alignment by checking if $k_l < s$ for the left factor in $W^l \cdot A^l$ on a layer basis. Should a change occur then the algorithm must consider the problem $A^T W$ for the next layer. The algorithm will then uphold its correctness and efficiency.*

VIII.6.2. Backpropagation

The described method can be applied analogously for backpropagation, as it can also be expressed in terms of the matrix product $(W^l)^T \cdot \Delta^l$. The matrix $\Delta^l \in \mathbb{R}^{k_l \times s}$ holds in row i the delta values $\delta_i^{l,j}$ for each neuron v_i^l and sample j. The matrix product can be computed with Alg. 24. The averaged gradient $< \partial^l >$ defined as

$$< \partial^l >:= \frac{1}{s} \sum_{k=1}^{s} \frac{\partial L^k}{\partial w_{\tilde{i},\tilde{j}}} \tag{VIII.12}$$

with L^k being the loss function for sample k, can be expressed as a matrix product

$$< \partial^l > = \frac{1}{s} \cdot G^l \cdot {\Delta^{l+1}}^T \tag{VIII.13}$$

$$= \frac{1}{s} \begin{pmatrix} g^{l,1}_{O(\tilde{1})} & g^{l,2}_{O(\tilde{1})} & ... & g^{l,s}_{O(\tilde{1})} \\ g^{l,1}_{O(\tilde{2})} & g^{l,2}_{O(\tilde{2})} & ... & g^{l,s}_{O(\tilde{2})} \\ ... & ... & ... & ... \\ g^{l,1}_{O(\tilde{k}_l)} & g^{l,2}_{O(\tilde{k}_l)} & ... & g^{l,s}_{O(\tilde{k}_l)} \end{pmatrix} \cdot \tag{VIII.14}$$

$$\begin{pmatrix} \delta^{l+1,1}_{O(\tilde{1})} & \delta^{l+1,2}_{O(\tilde{1})} & ... & \delta^{l+1,s}_{O(\tilde{1})} \\ \delta^{l+1,1}_{O(\tilde{2})} & \delta^{l+1,2}_{O(\tilde{2})} & ... & \delta^{l+1,s}_{O(\tilde{2})} \\ ... & ... & ... & ... \\ \delta^{l+1,1}_{O(\tilde{k}_{l+1})} & \delta^{l+1,2}_{O(\tilde{k}_{l+1})} & ... & \delta^{l+1,s}_{O(\tilde{k}_{l+1})} \end{pmatrix}^T \tag{VIII.15}$$

since for each element it holds that

$$< \partial^l >_{\tilde{i},\tilde{j}} = \frac{1}{s} \sum_{k=1}^{s} \delta^{l+1,k}_{\tilde{j}} g^{l,k}_{\tilde{i}} \tag{VIII.16}$$

In order to avoid non-coalesced memory access (through layout changes) the decision was made to calculate the averaged gradient with a separate layer-wise evaluation of a modified version of Alg. 24 (in which the final result is scaled by $\frac{1}{s}$).

VIII.7. Implementation

Most modern GPUs contain physical PPUs which allow the execution of up to 256 instructions simultaneously, thus we set the PPU size to this value. In order to keep the source code

simple it is assumed that the vPPU sizes are a power of 2 and smaller than 256. The programs were implemented in OpenCL, which allows a platform independent execution. Forward propagation up to layer V_{n-1} was done with a kernel $forwardProp1$ while the propagating input to the last layer was done with a kernel called $forwardProp2$ (which additionally calculated the δ values for the last layer), backward propagation was executed by the kernel *backwardProp*. All these kernels allow the virtual handling of shadow networks on each layer, the required merging was done through a kernel called $mergeMat$. The gradient was calculated with *calcGrad* on a layer basis. At this point a technical problem must be addressed: the high latency of host↔GPU data transfers. After each epoch an error check must be executed in order to check if the networks error is sufficiently low, this would require accessing the output layers deltas after each forward propagation. This approach is unfeasible since the latencies are much higher than the actual processing time. Thus $forwardProp2$ will compute the overall error and check whether it is under the set threshold, if so it will set the computed delta values to 0. This computation was done with an additional kernel $mergeMat2$. Note that this will effectively halt training, the host must periodically check whether convergence occurred. The frequency must be chosen according to the expected amount of epochs.
In order to find the best parameters for each layer I deducted a grid-search over the range of feasible values, this was done for the forward- and backward-propagation independently since the matrix dimensions differ.

VIII.7.1. Memory Requirements

Before discussing the results memory requirements must be elaborated. For each layer memory was allocated for A^l, W^l (for the first layer we consider $A^l = S$), Δ^l (for holding the delta values), G^l (which holds the derivatives) and ∂^l (holding the gradient values for each sample). In order to utilize memory coalescing one should keep a separate copy of each ${W^l}^T$ (this redundant copy increases the expected efficiency since alignment changes of A^{l+1} rarely occur). The required memory is given by

Theorem 14. *Storing all required data requires a total of* $(\sum_{l=1}^{n-1}(4 \cdot k_{l+1} \cdot s + 2 \cdot k_l \cdot k_{l+1} \cdot))sz$ *bytes, where* sz *equals the byte count which is required to store a floating point number. During computation one requires another* $\max_l((|f_l|A^l) + \max_l(|f_l|\partial^l)) \cdot sz$ *bytes.*

Proof. The element count in the matrices A^l, Δ^l, G^l and ∂^l is identical, thus $4 \cdot k_l \cdot s \cdot sz$ bytes are needed, whereas W^l and ${W^l}^T$ require $k_l \cdot k_{l+1} \cdot sz$ bytes each. If a layer uses a non-canonical split f_l we have to account for this as well by allocating a multiple $|f_l|$ of the corresponding matrix, it is sufficient to allocate a single scratch pad for each context (i.e. for computing ∂^l and A^l). □

In order to model non-fully connected networks one can extend my description with a binary matrix $B^l \in \{0,1\}^{|V_l| \times |V_{l+1}|}$ for each layer, which can be used to mask the weight values during computation.
Since certain system restrictions dictate a specific data layout, e.g. $(f\mu)|k_{l+1}$ for a column-wise computation, it will most likely force one to pad the data. Regarding dummy network extensions it is possible to use the mentioned binary matrix in order to mask them (for layer V_2 this will require to pad the input data

as well, e.g. with constant values for all samples). Whereas for the sample count s one could either simply omit samples in order to fulfill the requirements or alter the algorithm's description to enable an heterogeneous distribution among vPPUs. If required the dimension count can be reduced in the first place, e.g. with a principal component analysis.

VIII.7.2. Model Parameters

Hitherto I have not discussed the numerical range of feasible model parameters. In order to maximize the GPU efficiency one must launch enough threads to mask memory latencies while on the other hand keep the scheduling overhead feasibly low. The amount of threads is controlled by the number and size of vPPUs, whereas the number of fused vPPUs increases the benefit from shared memory. Yet using shared memory becomes feasible only if the number of samples/neurons per PPU as well as the size of network shadow does not exceed the amount of available shared memory. The required amount of shared memory thus represents the guideline for choosing the parameters, this is expressed formally through

Theorem 15. *Using shadow networks it holds that for Alg. 24 and layer V_l: In order to carry the required fragment of W^l each fused group of vPPUs needs memory space to hold $\frac{k_l}{|f_l|}$ number elements. Let $\alpha \in \mathbb{N}$ be the amount of local memory in terms of number elements. The maximal amount of fused vPPUs per PPU is $\lfloor \frac{\alpha |f_l|}{k_l} \rfloor$. The required fragment of S can be preloaded as well, yet this will require $\frac{s \cdot k_l}{|f_l|}$ additional memory units. This reduces the amount of fused vPPUs per PPU to $\lfloor \frac{(\alpha - \frac{s \cdot k_l}{|f_l|}) |f_l|}{k_l} \rfloor$ (if no PPU computes more than one shadow).*

Proof. All statements with respect to preloading W^l follow similarly as before only for a dimension reduced by a factor of $|f_l|$. Since all fused vPPUs (on the same PPU) will require the corresponding S fragment it can be preloaded by all PUs in the beginning, yet this will reduce the total amount of available shared memory to $\alpha - \frac{s \cdot k_l}{|f_l|}$. □

VIII.8. Results

The algorithms were evaluated on a Radeon7970, which provides 32 compute units (PPUs) with a total of 2048 stream cores (PUs), 32KB of local memory per workgroup and 3GB of global memory. Each PPU holds 64 PUs, which execute SIMD instructions. The host system provided a Core-i7 3820 3.6GHz CPU, 64GB RAM and was running ArchLinux 3.15.5-1 with the SimpleHydra SDK [MH14a].
I restricted my experiments onto the modules of backpropagation i.e. on the propagation methods themself. In this context I analyzed the following questions:

- What are the optimal model parameters for different problem setups (sample and neuron count) and what is their correlation?
- How does the model (with optimized parameters) compare to optimized matrix multiplication routines?

The graphs in Fig. VIII.3 depict the execution speeds of the SGEMM routine (i.e. routine for matrix multiplications with single precision accuracy) from the GNU Scientific Library (GSL), these numbers should be considered as the baseline for propagating batches through the network. The maximal time is roughly

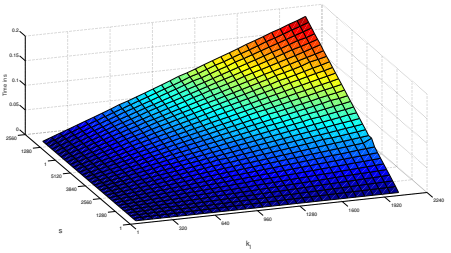


Figure VIII.3.: Execution speeds of the GSL SGEMM routine with increasing sizes of k_l and s with $k_{l+1} = 64$.

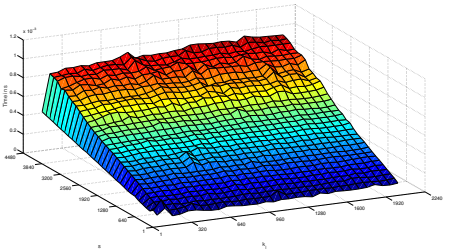


Figure VIII.4.: Execution speeds of the optimized AMD OpenCL SGEMM routine with increasing sizes of k_l and s with $k_{l+1} = 64$.

200ms for $k_l = 2048$ and $s = 4096$, the linear scaling is induced by the fact that only one core was used and the input dimensions were just too small in order to see the non-linear complexity of $\mathcal{O}(k_l k_{l+1} s)$.

Fig. VIII.4 shows the execution times for the same parameters, yet this time a highly optimized OpenCL BLAS library (clBLAS

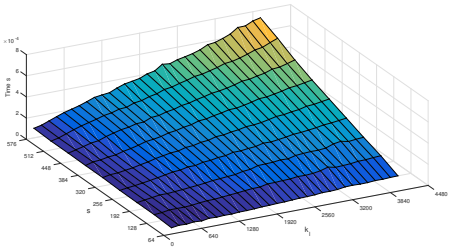


Figure VIII.5.: Execution speeds of the model with increasing sizes of k_l and s with $k_{l+1} = 64$. The graph shows only the range subset on which the model performed better than AMDs OpenCL SGEMM routine.

2.4) was used and the computation was carried out on the GPU. The plot shows that processing times have been reduced severely into the μs range, the maximal time is around $1ms$.

Finally Fig. VIII.5 shows the times for the optimized model (which were obtained by a grid search), I only show a subset since the efficiency of my model started to drop beyond this range. The performance gain is depicted in Fig. VIII.6, for a small sample count the efficiency (compared to the BLAS library) is increased by around $10\% - 350\%$, it is also evident that it increases with higher dimensions i.e. with larger k_l. Furthermore the efficiency drops significantly with a higher amount of training samples and falls below 0 beyond ≈ 768 samples.
The Fig. VIII.7,VIII.8 and VIII.9 show the optimal parameters which were used to obtain the times in Fig. VIII.5. By comparing Fig. VIII.7 and VIII.8 one can observe the balance between thread count and scheduling overhead. In case of few samples the GPU prefers larger vPPUs and a high fusion number. This

strategy becomes less beneficial with a higher amount of data, i.e. the thread count is reduced by lowering the vPPU size as well as the fusion factor. In other words, as the amount of required vPPUs increases with more data their size is scaled down in order to reduce the total amount of scheduled threads (i.e. keeping β at bay). Fig. VIII.9 shows how much the computation prefers to reduce a thread's complexity according to the sample dimensionality, i.e. it shows when the system prefers the threads in each vPPU to execute less for-loop iterations. With increasing k_l the efficiency is increased by reducing the number of iterations. Note the plateau-like character along the k_l axis. Tab. VIII.1 shows the performance gain for the actual training of a neural network. We used the glass and horse problem from the ProbenI database [P$^+$94] since they represent small scale problems, the network was trained with vanilla backpropagation. The figure shows four different approaches in comparison to each other; a multicore optimized implementation which distributes equally sized batch fragments onto the available CPU cores, a clBLAS version executed on the CPU as well as on a GPU and finally my model. The training set consisted of 214 samples each of dimension 9, which was increased to 16 by adding 0s. The network consisted of 3 layers, all 64 neurons in the second layer were equipped with a tanh activation function while the single neuron in the output layer contained the identity function. The initial weights were randomly chosen according to the fan-in rule of [GB10], yet during the evaluation they were kept identical among the different implementations (i.e. the initial state was identical among the networks). The learning rate η was set to 0.01, the momentum term α to 0.1 and all bias values were initialized to 0, i.e. the bias was not utilized at all. Using the OCL BLAS library on a multicore CPU yielded a perfor-

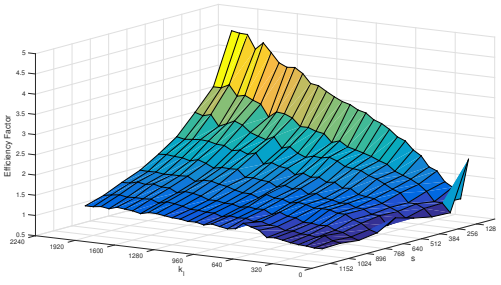


Figure VIII.6.: The efficiency of the developed model in comparison to AMDs OpenCL SGEMM routine, it scales linearly with the amount of training samples yet higher dimensionality increases the efficiency in general. The new computation model performs up to 4.5 times faster than the SGEMM routine.

mance factor of 3.91, which decreased to 3.85 on a GPU. This was expected as the library was optimized for larger matrices, the CPU outperforms the GPU due to the small amount of operations and low latency memory. Yet my approach performed 4.5 times faster than a multicore CPU implementation and thus showed to outperform the canonical matrix approach on GPUs. Additionally the approach was evaluated on the Proben1 horse problem, which consists of 300 samples with 51 parameters. Tab. VIII.2 shows the corresponding results, which also reinforce my model.

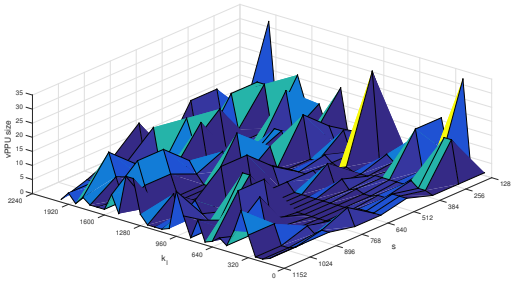


Figure VIII.7.: Optimal values for the vPPU size of the developed model. These values correspond directly with the values in Fig. VIII.5, i.e. they yielded the lowest time for the corresponding values of s and k_l.

VIII.9. Conclusion

So far I approached two classic problems within the field of parallel computation; neural network training and implicitly matrix multiplication. My work focused on small networks and the challenge of an efficient distribution on parallel architectures. I provided a novel approach to this problem and performed a thorough evaluation on a modern CPU and GPU, in the experiments I drew a direct comparison with methods for matrix multiplication. Overall I showed that my model outperforms highly optimized BLAS libraries for small problems and can provide a significant gain when training small neural networks on highly parallel architectures. The model provides several parameters which allow a dynamic adaptation for each problem setup. During the analysis I indicated how these values influence the trade-off between different forms of overhead, the practical analysis showed

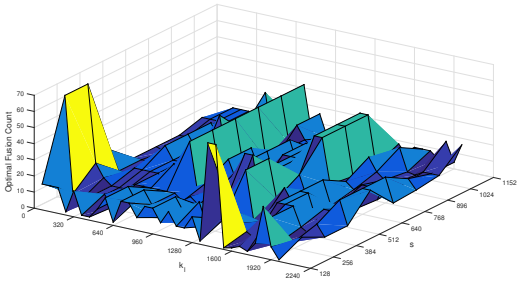


Figure VIII.8.: Optimal values for the vPPU fusion factor f. These values correspond directly with the values in Fig. VIII.5, i.e. they yielded the lowest time for the corresponding values of s and k_l.

that the optimal parameters are anything but trivially found. Yet the described approach introduces several restrictions; on one side it is necessary to determine the parameters for each layer and especially for each computation device (for a specific system) while on the other side one has to fulfill certain memory restrictions with respect to the underlying PPU groups. The parameter optimization is a layer-wise process, the overall network structure does not influence the efficiency. This might also be beneficial when training deep neural networks whose layers contain only a few neurons. More generally, my approach might also be applied to existing training algorithms and give rise to hybrid-methods which perform more efficiently on smaller data sets.
It remains to be seen how well this approach can be applied to larger networks, e.g. via splitting the problem onto multiple de-

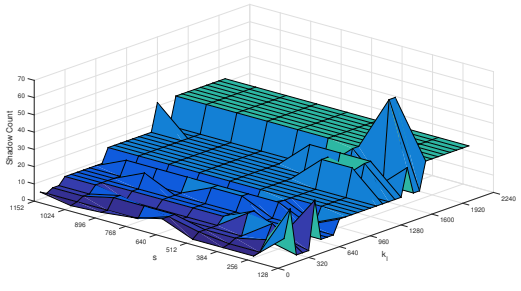


Figure VIII.9.: Optimal values for the amount of shadow networks. These values correspond directly with the values in Fig. VIII.5, i.e. they yielded the lowest time for the corresponding values of s and k_l.

vices or even distribute it within a cluster. Furthermore it should investigated how this model behaves on practical applications with smaller devices (esp. devices with different architectures). Through the use of the vPPU approach and the described problem linearization one can design a cluster distribution by simply applying the model recursively. Another interesting question is the applicability for different training algorithms in terms of efficiency (since it can be applied in general for any backpropagation based approach).

Table VIII.1.: Comparison of different training algorithms on the Proben1 glass problem.

Algorithm	MultiCore CPU	OCL BLAS CPU
#epochs	4118	4118
error	0.0899	0.0899
training time (epoch time)	6.51s (1.583ms)	1.655s (0.392ms)
rel. speed-up	-	3.91

Algorithm	OCL BLAS GPU	new model
#epochs	4120	4120
error	0.0801	0.0801
training time (epoch time)	1.69s (0.441ms)	1.4481s (0.351ms)
rel. speed-up	3.85	4.5

VIII.10. The Distributed Matrix Multiplication

There is a good reason why, when it comes to parallel algorithms, the first introductory example is usually the matrix multiplication. The classical matrix multiplication provides a fundamental way of handling / processing datasets for various applications, e.g. neural network training [EMA97], principal component analysis [F.R01], SVM-based classification [SS02] or linear discriminant analysis [FIS36], just to name a few in the field of machine learning. For many years algorithms have been continuously refined in order to gain distance to the canonical complexity of $\mathcal{O}(n^3)$. Assuming a sequential machine model the complexity was reduced to $\mathcal{O}(n^{2.807})$ [Str69] or even $\mathcal{O}(n^{2.3728639})$ [Gal14]. Yet both algorithms have little practical applicability due to hidden complexity constants [Rob05]. In the following sections I will describe how to efficiently distribute the canonical algorithm

Table VIII.2.: Comparison of different training algorithms on the Proben1 horse problem.

Algorithm	MultiCore CPU	OCL BLAS CPU
#epochs	9003	9003
error	0.08978	0.08978
training time (epoch time)	35.29s (3.92ms)	9.11s (1.002ms)
rel. speed-up	-	3.87
Algorithm	OCL BLAS GPU	new model
#epochs	9003	9003
error	0.08978	0.08978
training time (epoch time)	17.43s (1.9ms)	7.31s (0.812ms)
rel. speed-up	2.03	4.82

within a Beowulf cluster whose nodes are equipped with multiple modern GPUs. Thus I approach the challenge of finding optimal distributions among the node-local GPUs and among the cluster nodes.
Section VIII.11 begins the discussion with the question how to efficiently compute a matrix multiplication on a single GPU. Furthermore all basic terms for the remaining chapter will be defined and it will be briefly explained how to split the problem among different dimensions. Section VIII.12 will briefly describe the concept of virtual parallel processing units and extend it with a fusion approach. In addition to that the concept of hypersystems will be introduced, which represents the base for my theoretical model. The remaining algorithmic elements will be illustrated in section VIII.13 since they built upon previously described concepts. In section VIII.14 I discuss how to split the computation onto multiple GPUs, in this context I also explain the induced scheduling problem. Section VIII.15 generalizes the distribution

approach for arbitrarily structured hypersystems and provides an algorithm in order to optimize distribution parameters. The results and a corresponding discussion are presented in sections VIII.16 and VIII.17, respectively.

VIII.11. Preliminary Definitions

The Matrix multiplication Problem (MP) is defined as

Definition 6. *Let $A \in \mathbb{R}^{k \times l}, B \in \mathbb{R}^{l \times m}, C \in \mathbb{R}^{k \times m}$ be matrices. The matrix multiplication problem is to find a three-stage workload schedule S which defines the workload distribution among GPU-equipped nodes in a cluster system in order to maximize the efficiency of computing the product.*

$$C = A \cdot B \tag{VIII.17}$$

The first stage consists of finding a local distribution which defines how to schedule the work onto a single GPU, the second stage defines the deployment among multiple local GPUs while the third stage defines the distribution among multiple nodes.

The matrix product is defined as follows, each element $c_{i,j}$ of C is computed as $a_i^r \cdot b_j^c$, with a_i^r being row i of A and b_j^c being column j of B. The dot product exhibits a complexity of $\mathcal{O}(l)$, since there are $k \cdot m$ elements within C one obtains an overall complexity of $\mathcal{O}(klm)$. For the remaining chapter l should be referred to as the dimensionality of the problem and w.l.o.g. consider $k > m$.
The canonical algorithm can be expressed as in listing 26 Note the size hierarchy regarding the for-loops if $k > l$, the most outer loop (F1) will exhibit more iterations than the second loop (F2),

Algorithm 26 *naiveMul* computes $A \cdot B$

Input: A, B, C, k, l, m

```
1: for i = 0; k do
2:     for j = 0; m do
3:         c_{i,j} = 0;
4:         for n = 0; k do
5:             c_{i,j} = c_{i,j} + a_{i,k} · b_{k,j};
6:         end for
7:     end for
8: end for
```

yet the most inner loop (F3) entirely depends on the problem dimensionality. In order to reduce the iterations of F3 I utilize the following fact.

Definition 7. *Let $A \in \mathbb{R}^{k \times l}$, $s_A := \{I | I \subset \{1, ..., l\}\}$ is called a vertical partition of A if $\forall I, J \in s_A : I \cap J = \emptyset$. Every partition of A induces a set $\tilde{A}$ of matrices defined by*

$$\tilde{A}_I := \begin{pmatrix} a_{1,O(1)} & a_{1,O(1)} & ... & a_{1,O(|I|)} \\ a_{2,O(1)} & a_{2,O(1)} & ... & a_{2,O(|I|)} \\ ... & ... & ... & ... \\ a_{k,O(1)} & a_{k,O(1)} & ... & a_{k,O(|I|)} \end{pmatrix} \tag{VIII.18}$$

for all $\tilde{A}_I \in \tilde{A}$ with $I \in s_A$ and some order O on I. A horizontal partition is defined analogously $s'_A := \{I | I \subset \{1, ..., k\}\}$ and the induced matrix set $\tilde{A}'$

$$\tilde{A}'_I := \begin{pmatrix} a_{O(1),1} & a_{O(1),2} & ... & a_{O(1),l} \\ a_{O(2),1} & a_{O(2),2} & ... & a_{O(2),l} \\ ... & ... & ... & ... \\ a_{O(|I|),1} & a_{O(|I|),2} & ... & a_{O(|I|),l} \end{pmatrix} \tag{VIII.19}$$

for all $\tilde{A}'_I \in \tilde{A}'$ with $I \in s'_A$ and some order O on I.

Theorem 16. *Let $s_{A,B}$ be a vertical partition of A and a horizontal partition for B (i.e. the subsets are used as indices for both matrices). Then it holds that*

$$C = \sum_{I \in s_{A,B}} \tilde{A}_I \cdot \tilde{B}'_I \tag{VIII.20}$$

Proof. Let $I \in s_{A,B}$ be an arbitrary index set, each $x_{i,j}$ element of $\tilde{A}_I \cdot \tilde{B}'_I$ is

$$x_{i,j} = \sum_{d \in I} a_{i,O(d)} b_{O(d),j} \tag{VIII.21}$$

Since all $I \in s_{A,B}$ are disjoint one obtains all fragments of the original dot product

$$x_{i,j} = \sum_{d=1}^{l} a_{i,d} b_{d,j} \tag{VIII.22}$$

by summing over all I. □

Thus with a partition one has to compute multiple matrix products but with reduced dimensionality. Since modern GPUs are targeted as computation platform one must discuss memory alignments. The canonical way of saving matrices in 1-dimensional memory is to linearize them in a row- or column-major scheme, in which either all rows or columns are sequentially concatenated and saved, respectively. All following algorithms assume that A and B are saved in row-major format. Furthermore one should note that Alg. 26 iterates in a row-major scheme over the elements of C, in an actual implementation this will imply a row-major alignment of C. The case of $k < m$ introduces another challenge since in order to maintain the loop-hierarchy one has to consider the problem $B^T \cdot A$, which yields

$C = (B^T \cdot A)^T$. As shown in Alg. 26 it is possible to switch both loops in order to obtain the old order of sizes, yet this will result a column-major alignment of C or a row-major alignment of C^T.

VIII.12. Distribution of Loop Iterations

The integral challenge in the MP lies in the distribution of the for-loop iterations onto a large set of parallel processing units. Even though one can reduce the inner loop by splitting the initial MP into a set of MPs with smaller dimensionality, in the end one has to account for this with the overhead of merging the subproblems solutions. Before explaining the distribution I introduce a generalization of the vPPU concept of [MH14b].

Definition 8. *Let $\mathcal{S}$ be an arbitrarily structured computation system with a set Υ of atomic computation units. A $\mathcal{H}$ hypersystem is defined recursively through*

- *A hypersystem $\mathcal{H} = (\aleph, \Omega, \Xi)$ is a 3-tuple with $\aleph$ being a set of virtual parallel processing units (vPPUs), $\Omega = \{\omega(\mu, \nu) \in \mathbb{R} | \mu, \nu \in \aleph\}$ the set of communication costs between vPPUs and Ξ a set of system attributes for each vPPU.*

- *$\pi \in \Upsilon$ is a vPPU with communication cost $\Omega \equiv 0$ and $|\pi| = 0$.*

- *A vPPU is a set of vPPus with communication costs Ω between these units and Ξ a system description. A vPPU is at least capable of delegating work to its members.*

Remark VIII.5. *The vPPU concept of [MH14b] is a special case of my definition, in which 2 recursions are used i.e. gpu shader, vPPU ([MH14b]) and compute unit. The system attributes might refer to any combination of different (wrapped) hardware structures, e.g. a vPPU consisting of a group of CPUs and a group of GPUs. When distributing algorithm to vPPUs with different Ξ it might be required to modify the algorithm according to Ξ, e.g. rewrite/redesign it according to the SIMD paradigm. The atomic units can be vastly different according to this definition, an example would be a set of GPU shaders and CPUs of barrel CPUs [Kai96]. In the following sections a hardware defined group of computation units is considered to be a PPU, e.g. AMD compute units.*

With this definition one can model a Beowulf cluster [SBS+95] with non-homogeneous GPU-equipped nodes as follows

Definition 9. *A GPU-powered Beowulf cluster is a hypersystem $\mathcal{H}_B$ grouped into 3 vPPU **types** $vPPU_0$, $vPPU_1$ and $vPPU_2$, which are defined as follows.*

- *$vPPU_2$ describes the set of cluster nodes, the communication costs are $\omega_{ether} \cdot \alpha$ with ω_{ether} being the Ethernet latencies and α an algorithm dependent factor.*
- *$vPPU_1$ describes a single node with a set of GPUs (separate vPPUs), CPUs (separate vPPUs) and communication costs between these devices.*
- *$vPPU_0$ is the description of device-local hypersystems, e.g. GPUs (the entire definition in [MH14b] is located in this group).*

Beginning with $vPPU_0$ we now discuss the device-local distribution of iterations among SIMD GPUs. The algorithm in listing

27 shows the kernel structure in order to distribute the computation of c^r to a single PU (GPU hardware thread), this approach distributes the outermost for-loop onto the shaders of one GPU with a total of τ shaders per PPU. The total amount of shaders will most likely be smaller than k thus one has to account with scheduling costs. Furthermore even with k being large it might be too small in order to mask global memory latencies through high thread counts.

Algorithm 27 *computeRow* of C

Input: Linearized matrices A, B,k,l,m

1: $(P_{i_{PU},c,t,i,j})\{$
2: i_{PU} = getPUIdx();
3: //In case one can only start PUs in multiples of τ
4: **if** $i_{PU} \geq k$ **then**
5: return;
6: **end if**
7: $c = m$;
8: $t = 0$;
9: **for** $j = 0$; c **do**
10: **for** $i = 0$; l **do**
11: $t = t + A_{i_{PU},l} \cdot B_{l,j}$;
12: **end for**
13: $c_{i_{PU},j} = t$;
14: $t = 0$;
15: **end for**
16: }

Since the memory latencies largely outweigh scheduling costs we define PPU-local vPPUs, i.e. groups of shaders which will collectively compute the outer for-loop. This approach is depicted

in Alg. 28

Algorithm 28 *computeRow′* of C

Input: Linearized matrices A, B,k,l,m,μ

1: $(P_{i_{PU},c,t,j,i_{vPU},i_{vPPU},a,b,i})\{$
2: $i_{PU} =$ getPUIdx();
3: $i_{vPU} = i_{PU} \mod \mu$;
4: $i_{vPPU} = i_{PU} DIV \mu$;
5: //In case one can only start PUs in multiples of τ
6: **if** $i_{vPPU} \geq k$ **then**
7: return;
8: **end if**
9: $c = m/\mu$;
10: $t = 0$;
11: $a = i_{vPU} \cdot c$
12: $b = (i_{vPU} + 1) \cdot c$
13: **for** $j = a$; b **do**
14: **for** $i = 0$; l **do**
15: $t = t + A_{i_{vPPU},l} \cdot B_{l,j}$;
16: **end for**
17: $c_{i_{vPPU},j} = t$;
18: $t = 0$;
19: **end for**
20: $\}$

Here each vPPU is defined to be of size μ with $\mu|\tau$, which leads to the following

Theorem 17. *Alg. 28 exhibits a complexity of* $\mathcal{O}(\frac{k}{z\tau} \cdot l \cdot \frac{m}{\mu} \cdot \beta(z))$, *with* z *being the amount of* ***scheduled*** *PPUs and* β *the scheduling coefficient with respect to the amount of launched threads i.e. parameterized by* z.

This approach will yield a better latency masking, especially for smaller matrices, yet it still does not utilize any PPU-local memory. This is difficult in terms of available space since each vPPU would need a local copy of $a^r_{i_{vPPU}}$. Furthermore it will most likely be impossible to preload the entire matrix B into local memory as modern GPUs feature PPUs with at most 40kB of local memory. This issue will be addressed with matrix partitioning within the next section.
I extended the concept of [MH14b] with a fusion approach of multiple vPPUs on (different) PPUs, the corresponding algorithm is depicted in listing 29. This algorithm implicitly spreads the outer for-loop onto more threads with increasing scale f. Furthermore it virtually groups multiple vPPUs into a single one, which motivates the use of local memory as long as more than one fused vPPU resides on a PPU.

Note that it might be impossible to use local memory according to following

Theorem 18. *Let α be the amount of local memory in units of floating point numbers. If local memory is to be utilized then at most $\lfloor \frac{\alpha}{l} \rfloor$ fused vPPUs can be scheduled for a single PPU. The required memory is given by $l \cdot \lceil \frac{\tau}{f\mu} \rceil$.*

Proof. Each fragment consists of one row from A, which contains l elements and obviously implies the stated bound. The total required memory corresponds to the number of fused vPPUs per PPU, there are at most τ units in a fused group. Two situations might occur; if $f\mu < \tau$ then the total amount of fused $vPPUs$ is given by $\lceil \frac{\tau}{f\mu} \rceil$, if $f\mu \geq \tau$ then only one fused group (or a fragment of it) resides on the PPU. □

This makes it impossible to use local memory if $l > \alpha$. The complexity is

Algorithm 29 *computeRow″* for C

Input: Linearized matrices A, B,k,l,m,μ,f

1: $(P_{i_{PU},c,t,j,i_{vPU},i_{vPPU},a,b,i})\{$
2: i_{PU} = getPUIdx();
3: $i_{vPPU} = (i_{PU} DIV \mu) DIV f$;
4: $i_{vPU} = i_{PU} \mod (f \cdot \mu)$;
5: //In case one can only start PUs in multiples of τ
6: **if** $i_{vPPU} \geq k$ **then**
7: return;
8: **end if**
9: Preload A fragments into local memory x
10: $c = m/(\mu \cdot f)$;
11: $t = 0$;
12: $a = i_{vPU} \cdot c$
13: $b = (i_{vPU} + 1) \cdot c$
14: **for** $j = a$; b **do**
15: **for** $i = 0$; l **do**
16: $t = t + x_{i_{vPPU},l} \cdot B_{l,j}$;
17: **end for**
18: $c_{i_{vPPU},j} = t$;
19: $t = 0$;
20: **end for**
21: }

Theorem 19. *Alg. 29 exhibits a complexity of* $\mathcal{O}(\frac{k}{z\tau} \cdot l \cdot \frac{m}{f\mu} \cdot \beta(\tau, z) + \frac{z\tau}{f\mu} \cdot \Omega(l, f, \mu) + \aleph(l, f, \mu))$. *With* $\beta \geq 1$ *being a scheduling penalty and* $\aleph$ *a penalty term in case that* $f\mu \nmid \zeta\tau$ *for* $\zeta \in \mathbb{N}$ *(i.e. penalty if vPPU fusion reaches into another PPU but not to its end)*

Proof. There are still only $z\tau$ (non-physical) PUs which calculate the elements, of which there are $k \cdot l \cdot m$. Due to the limited amount of physical PUs the efficiency will be scaled according to β. Furthermore the outer for-loops iteration count was reduced by a factor of $f\mu$. The second term accounts for memory latencies during the preloading of A fragments, $\Omega(l, f, \mu)$ represents the overhead for loading one such fragment by a group of fused vPPUs, of which there are $\frac{z\tau}{f\mu}$ in total. The last term accounts for the lost efficiency if a group of fused vPPUs reaches into another PPU but does not cover it completely, i.e. the efficiency will be lost as only a few vPPUs will benefit from the preloaded data. □

VIII.13. Splitting the Dimensionality

This section extends the previous algorithm for inclusion of partitions. Recall that a partition splits the MP into MPs where $l' < l$. Although reducing the size of each row and column in A and B respectively, each split increases the required memory to hold all MP-local C' matrices. Additionally it introduces the cost of merging these matrices. Since I aim towards a distribution onto multiple heterogeneous devices we will consider partitions of different size. The main algorithm is depicted in listing 30

It is similar to Alg. 29 yet it solves only a reduced subproblem

Algorithm 30 *computeRow''* for C'

Input: Linearized matrices A, B, k, l, m, μ, f

1: $(P_{i_{PU},c,t,j,i_{vPU},i_{vPPU},a,b,i})\{$
2: i_{PU} = getPUIdx();
3: $i_{vPPU} = (i_{PU} DIV \mu) DIV f$;
4: $i_{vPU} = i_{PU} \mod (f \cdot \mu)$;
5: //In case one can only start PUs in multiples of τ
6: **if** $i_{vPPU} \geq k$ **then**
7: return;
8: **end if**
9: Preload A' fragments into local memory x
10: Preload B' into local memory $\tilde{B}$
11: $c = m/(\mu \cdot f)$;
12: $t = 0$;
13: $a = i_{vPU} \cdot c$
14: $b = (i_{vPU} + 1) \cdot c$
15: **for** $j = a$; b **do**
16: **for** $i = 0$; l **do**
17: $t = t + x_{i_{vPPU},l} \cdot \tilde{B}_{l,j}$;
18: **end for**
19: $c'_{i_{vPPU},j} = t$;
20: $t = 0$;
21: **end for**
22: }

and requires that l' is small enough in order to preload the entire matrix B'. The major aspects are summarized as follows

Theorem 20. *Let $\gamma = \lceil \frac{\tau}{f\mu} \rceil$, under the assumption that vPPUs won't be fused beyond a PPU the following statements are true:*

- *Each subproblem is limited in its dimensionality l' by $l' \leq \frac{\alpha}{\gamma+k}$.*
- *Dividing the initial MP symmetrically into subproblems, i.e. $l' \equiv \frac{l}{s}$, bounds the factor s by $s \geq \frac{l}{\alpha}(\gamma + k)$*
- *It holds that $\aleph \equiv 0$.*

Proof. The first statements follow directly from $\gamma \frac{l}{s} \leq \alpha - \frac{kl}{s}$. The last statement holds due to the assumption. □

The optimal parameters for a single device are determined by the hardware and software environment, i.e. they must be found empirically. Merging the subproblems is done by Alg. 31

Which exhibits a complexity of $\mathcal{O}(k \cdot m \cdot s)$. We can now finally state total time complexity of the model

Theorem 21. *Using Alg. 30 and 31 the total time complexity for calculating $A \cdot B$ is*

$$\mathcal{O}(s \cdot (\frac{k}{z\tau} \cdot \frac{l}{s} \cdot \frac{m}{f\mu} \cdot \beta(\tau, z) + \frac{z\tau}{f\mu} \cdot \Omega(l, f, \mu) + \aleph(l, f, \mu)) + \frac{k}{z\tau} \beta'(z, \tau) \cdot m \cdot s) \tag{VIII.23}$$

for a partition factor s.

VIII.14. Multi-GPU Computation

In order to deploy the computation efficiently among multiple GPUs in a single node (i.e. approaching the problem on layer

Algorithm 31 *mergeMP*

Input: Set of linearized matrices $\{C^{'},^{i}\}$,k, l,m, subproblem count s

1: $(P_{i_{PU},t,i,j})\{$
2: i_{PU} = getPUIdx();
3: //In case one can only start PUs in multiples of τ
4: **if** $i_{PU} \geq km$ **then**
5: return;
6: **end if**
7: $t = 0$;
8: $i = i_{PU} DIV l$;
9: $j = i_{PU} \mod l$;
10: **for** $k = 0$; s **do**
11: $t = t + C^{'},^{k}_{i,j}$;
12: **end for**
13: $c_{i,j} = t$;

$vPPU_1$) one has to distinguish two cases: all devices are identical or they are not. The first case can be solved by partitioning the matrix A horizontally in a symmetric way. For n devices this will affect the overall time complexity by reducing k to $k' := k/n$ and also reduce the memory requirements for each device (see section VIII.14.1). Yet one has to be careful and include the communication costs into this pragmatic solution.
The case of different devices requires performance metrics for each device, for this purpose I introduce the concept of profile surfaces.

Definition 10. *Let v be a vPPU with $n := |v|$ and $\mathcal{A}$ an algorithm whose complexity $\mathcal{C}(x_1, ..., x_m)$ depends on a set of m parameters x_i. Furthermore let $\mathcal{A}$ be segmented into b sequential steps q_j. The vPPU-local profile surface S_v is defined as*

$$S_v(x_1, ..., x_m, j) := t_v(q_j(x_1, ..., x_m)) \tag{VIII.24}$$

with $t_v(x)$ being the time that device v needed for computing q_j. The marginal time T_v is defined as

$$T_v(x_1, ..., x_m) := \sum_{j=1}^{b} S_v(x_1, ..., x_m, j) \tag{VIII.25}$$

In our case the complexity depends only on the reduced parameter k' (since the matrix dimensions are fixed and the k will be reduced to a MP-optimal k' for each device) and the algorithmic model consist of two steps, subproblem computation and result merging. The workload is split according to following steps, let there be n nodes in the system:

- Determine profile surfaces S_v for each GPU v.

- Find a sequence $K = (k'_v)_v$ with $k = \sum_{k'_v \in K} k'_v$ such that $\sum_{k'_v \in K} T_v(k'_v)$ is minimal.
- Distribute the work by assigning device v the computation of k'_v C-rows.

Computing the profile surfaces must be done only once for each MP setup, this holds for the second and third step as well, since the distribution schedule can be saved locally and be applied to any new data of same proportions. Keeping a database of already solved distribution problems as a look-up cache will give rise to a learning and self-optimizing system.
Sadly, the second step is NP-hard

Theorem 22. *Finding a sequence $K = (k'_v)_v$ with $k = \sum_{k'_v \in K} k'_v$ such that $\sum_{k'_v \in K} T_v(k'_v)$ is minimal ($\mathcal{T}$), is NP-hard.*

Proof. Let Multiple Choice Knapsack (MCKP) be as described in [LKC12] or [HLS10]. It will be shown that MCKP $\leq_P \mathcal{T}$ .The proof will be done in two steps. First it will be shown how an input of $\mathcal{T}$ relates to an input of MCKP. The resulting scheme will be used to transform an input of MCKP to one of $\mathcal{T}$.
An input I for $\mathcal{T}$ can be encoded as

$$
\begin{aligned}
I \quad = \quad & ((T_1(\tilde{k}_1^1), 1), ..., (T_1(\tilde{k}_{l_1}^1), 1), && \text{(VIII.26)}\\
& (T_2(\tilde{k}_1^2), 2), ..., (T_2(\tilde{k}_{l_2}^2), 2), && \text{(VIII.27)}\\
& ..., && \text{(VIII.28)}\\
& (T_d(\tilde{k}_1^d), d), ..., (T_d(\tilde{k}_{l_d}^d), d)) && \text{(VIII.29)}
\end{aligned}
$$

with d being the amount of devices and $\tilde{k}_l^m$ the l-th sample value along the k-axis for device m. This can be transformed to the

following input $\mathcal{M}$ of MCKP

$$G^j := \{(T'_j(\tilde{k}^j_1), j, \tilde{k}^j_1), ..., (T'_j(\tilde{k}^j_{l_j}), j, \tilde{k}^j_{l_j})\} \quad \text{(VIII.30)}$$

$$k = \sum_{j=1}^{d} \sum_{i=1}^{|G^j|} G^j_{i,3} x^j_i \quad \text{(VIII.31)}$$

$$\text{maximize } -\sum_{j=1}^{d} \sum_{i=1}^{|G^j|} G^j_{i,1} x^j_i \quad \text{(VIII.32)}$$

where $x^j_i \in 0, 1$ indicates a selection, G^j are the multiple choice groups from which exactly one item must be selected, $T'_i(z)$ is an integer representation of $T_i(z)$ which preserves the natural order among the times $T_i(.)$, finally $G^j_{i,z}$ represents the zth element within the ith item from group j. The transformation can be computed in linear time $\mathcal{O}(k \cdot d)$ (i.e. linear in the count of elements), note that an order preserving integer representation of a finite set of finite real numbers can be obtained by embedding them binary in a large enough integer container.
This transformation can now be used for tranforming an input of MCKP to one of $\mathcal{T}$ by creating multiple instances $\mathcal{M}$. Let m be an input for MCKP with d' groups G^j of elements, and one restriction of the form $\tilde{k} \geq$. This input can be directly transformed to an input for $\mathcal{I}$ with d' devices and matrix size $\tilde{k}$. In order to find an existing solution for m one has to create a total of $\tilde{k}$ instances for $\mathcal{I}$, each instance is obtained by decrementing $\tilde{k}$ by one. Thus one obtains a linear amount of algorithm evaluations. □

Remark VIII.6. *Depending on the sample steps one might not obtain a feasible solution to MP. Using a sample step of* 1 *along the profile surface's k-axis will ensure the existence of a solution.*

The previous theorem can be generalized for different algorithms i.e. including more dimensions with more parameters, it must be pointed out that despite its NP-hardness this model will deliver the empirical optimal schedule with respect to all complexity parameters. Since my definition bases on the vPPU model the scheduling approach is not bound to any device. Note that the effect of vPPU communication costs will be included in T_v.

VIII.14.1. Space Complexity

In this section I briefly discuss the space complexity of Alg. 30 with respect to a single GPU. In order to store the initial matrices one needs space for holding $k \cdot l + l \cdot m + k \cdot m$ real numbers. During computation s determines the amount of memory for holding the matrix fragments $C^{'},^{i}$ (note that s does not change the memory requirements for holding the initial data, furthermore no fragmentation of the initial data is required). Thus one obtains

Theorem 23. *Alg. 30 requires*

$$((k \cdot l + l \cdot m + k \cdot m) + \frac{l}{s} \cdot (k \cdot m \cdot s)) \cdot sz \tag{VIII.33}$$

bytes, with sz being the amount of bytes for holding a real number.

Proof. There are $\frac{l}{s}$ fragment matrices $C^{'},^{i} \in \mathbb{R}^{k \times (m \cdot s)}$. □

VIII.15. The Optimal Hypersystem Schedule

In section VIII.13 I have introduced the concept of profile surfaces and established their relevance to a practical problem (i.e. multi-GPU distribution Alg. 30). In this section I will show that the scheduling problem is a generally non-trivial element of each vPPU. Before we formalize the algorithm for obtaining an optimal schedule for all layers in a hypersystem let us temporarily focus only on $vPPU_1$ for a single node. In order to solve the scheduling problem within layer $vPPU_1$ we need the profile surfaces of each GPU in the node, these surfaces can be obtained through evaluation of Alg. 30 for different values of k. Since we are motivated to use these values in order to reduce the overall computation time we need to find an optimal distribution among the vPPUs on the GPU, i.e. for each k we have to find optimal values f, μ and s. In order to formally include SIMD devices, e.g. GPUs, let us define

Definition 11. *Let D be a SIMD device with n hardware-defined and identical groups of execution units. Each hardware-defined group , e.g. compute unit, is considered to be an atomic unit. Let $\mathcal{A}$ be an algorithm for D whose complexity is parameterized by $x_1, , ..., x_m$ and which can be divided into steps $q_1, ..., q_b$, the profile surface S_v for each atomic unit v is defined as*

$$S_v(x_1, ..., x_m, j) := \frac{1}{n} t_v(q_j(x_1, ..., x_m)) \tag{VIII.34}$$

with $t_v(q_j(x_1, ..., x_m)$ being the time that D needs to finish computation of $\mathcal{A}$ with $x_1, ..., x_m$. The marginal time T_v is defined

as

$$T_v(x_1, ..., x_m) := \sum_{j=1}^{b} S_v(x_1, ..., x_m, j) \tag{VIII.35}$$

Remark VIII.7. *With this definition we do not exclude the scheduling of an infinite number of work units, e.g. OpenCL workgroups, on the finite amount of atomic units. Additionally in case of GPUs this definition incorporates the element of scheduling overhead. Which is proportional to the amount of scheduled work units, which in turn are proportional to the parameters for each atomic unit. Due to the assumption that all atomic units are identical the described profile surface (for the parameter set of a single unit) is representative for the entire algorithm's performance (since each unit will calculate the same fraction).*

The SIMD scheduling problem is then defined as

Definition 12. *Find a set of parameters $x_1, ..., x_m$ which minimize $T_v(x_1, ..., x_m)$ for any v.*

As mentioned before the SIMD devices are GPUs of which each shader group is parameterized by f, μ, s. In order to find an optimal set of parameters one can simply execute an exhaustive search while considering the inter-parameter boundaries, e.g. $s \geq \frac{l}{\alpha}(\gamma + k)$. On layer $vPPU_1$ the algorithm consists of two steps, subproblem computation and solution merging, where a subproblem is obtained by splitting k into fractions for each device. It is very similar for $vPPU_0$, where each subproblem is obtained by splitting the dimensionality, thus on each layer the corresponding algorithm is divided into two steps. On closer inspection one will recognize that the first step in $vPPU_1$ induces both steps in $vPPU_0$ (in sequential order). Thus it follows

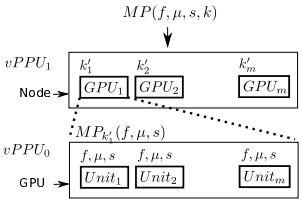


Figure VIII.10.: Parameters and structure of $vPPU_0, vPPU_1$. The initial problem has the parameters f, μ, s, k, each node on layer $vPPU_1$ will distribute among his contained vPPUs (i.e. GPUs) according to fragments k' of k. Each GPU on layer $vPPU_1$ will be confronted with the challenge of distributing the workload onto its units in terms of f, μ, s.

that optimizing step 1 in $vPPU_1$ requires an optimization of the marginal time T_v for each v in $vPPU_0$ (for k fixed by $vPPU_1$). The same holds for the merging step in $vPPU_1$, yet this step uses an algorithm in $vPPU_0$ which is unparameterized (the only parameters have already been fixed on $vPPU_1$). This is depicted in Fig. VIII.10 and VIII.11, note that it is not mandatory for each algorithm step to delegate work to another vPPU.

VIII.15.1. Generalization

This section finalizes the theoretical model.

Definition 13. *Let $\mathcal{H}$ be a hypersystem, $\mathcal{P}$ a computable problem and $\mathcal{G}$ a set of algorithm families g_x. Additionally let each vPPU v be associated with exactly one family $g_v \in \mathcal{G}$.*
The hypertree $\mathcal{T}_\mathcal{H}$ for $\mathcal{H}$ is a labeled tree with root of depth $z+1$ which fulfills:

- *Each leaf of $\mathcal{T}$ is labeled with an atomic unit of $\mathcal{H}$. All*

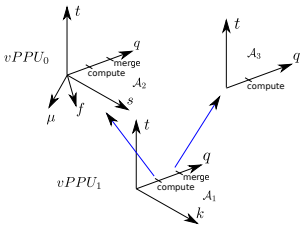


Figure VIII.11.: Optimization hierarchy for node-local matrix computation. In order to find an optimal distribution according to the structure of Fig. VIII.10, each node requires profile surfaces ($S_v = S_v(k')$) for each of its vPPUs v. Each profile surface must in turn provide optimized time values for each value k', i.e. each GPU must find the optimal parameters f, μ, s for a given k'. Furthermore this happens on a step-basis, here the compute-step of $vPPU_1$ requires an algorithm in $vPPU_1$, which in turn is splitted into two steps. Both steps must be executed sequentially (i.e. compute− >merge) in order to compute the result of the compute-step in $vPPU_1$.

atomic units of $\mathcal{H}$ are used as labels.

- *Every node (except for the root node) is labeled with a vPPU from $\mathcal{H}$.*
- *The parent v of a node v_0 or leaf l_0 is labeled with the vPPU to which the label of v_0 or l_0 belongs, respectively.*
- *The root node r is labeled with an imaginary vPPU $\lrcorner$ which is considered to be capable of distributing work in $\mathcal{H}$. Additionally r is labeled with a parameter Ω_r which contains the initial distribution parameters for $\mathcal{H}$, Ω_r is considered to be optimal.*

An optimal schedule $\beth$ *for* $\mathcal{H}$ *is a label-scheme for* $\mathcal{T}_{\mathcal{H}}$ *which associates each node and leaf* v *of* $\mathcal{T}_{\mathcal{H}}$ *with an index* x_v *for the corresponding label's algorithm family such that:*

- $\mathcal{P}$ *is solved if each vPPU* v *executes* $\mathcal{A}_{x_v} \in g_v$.
- *each vPPU communicates only with its parent or its children in* $\mathcal{T}_{\mathcal{H}}$
- *Let* L *be the level of* $\mathcal{T}_{\mathcal{H}}$ *after the root node,* $\max_{v \in L} T_v(x_v)$ *is minimized*

Remark VIII.8. *There exists exactly one tree for each hypersystem. The algorithms in each* $g \in \mathcal{G}$ *are indexed according to their parameter set, e.g.* $a_{4,6} \in g$ *would be an algorithm parameterized by two numbers, in this case with* 4 *and* 6 *as explicit values. The definition requires an already existing algorithm structure for the hypersystem. This structure must be designed with respect to each vPPUs system attributes* Ξ. *Each algorithm* $\mathcal{A}_{x_v} \in g_v$ *may still be parameterized, i.e.* $\mathcal{A}_{x_v} = \mathcal{A}_{x_v}(x, y, z, \&c)$. *These parameters are inputs from the corresponding parent node* u *in* $\mathcal{T}_{\mathcal{H}}$, *one can refer to such a set by* $P_v(u)$.

The following algorithm is a general approach in order to compute the optimal schedule the case that $\mathcal{G}$ contains families of algorithms which are parameterized by a finite set of integers (i.e. $\forall u : |P_v(u)| < \infty$).

Remark VIII.9. *A call to compute-*$\beth_2(r, \Omega_r)$ *will compute the optimal schedule, Alg. 34 will in general compute the optimal parameters for all nodes below it. Alg. 32 will return the profile surface of node* w *for a fixed parameter* x *from the upper layer. This is used by Alg. 33, which utilizes it in order to construct*

Algorithm 32 getOpt

Input: $\mathcal{T}_{\mathcal{H}}$, $w \in \mathcal{T}_{\mathcal{H}}$, parameter x

1: **for** each $\tilde{w} \in child(w)$ **do**
2: **for** each $\tilde{x} \in P_w(x)$ **do**
3: $S_{\tilde{w}}(\tilde{x}) :=$ getOpt$(\tilde{w}, \tilde{x})$
4: **end for**
5: **end for**
6: Solve $P_w(x)$-Scheduling problem for x with help of $\{S_{\tilde{w}}\}$
7: **return** time $T_w(x)$ with optimal schedule among the children $child(w)$.

Algorithm 33 compute-$\beth_1$ for $w \in \mathcal{T}_{\mathcal{H}}$

Input: $\mathcal{T}_{\mathcal{H}}$, $w \in \mathcal{T}_{\mathcal{H}}$, optimal parameter Ω for parent.

1: **for** each $\tilde{w} \in child(w)$ **do**
2: **for** each $\tilde{x} \in P_w(\Omega)$ **do**
3: $S_{\tilde{w}}(\tilde{x}) :=$ getOpt$(\tilde{w}, \tilde{x})$
4: **end for**
5: **end for**
6: Solve $P_w(\Omega)$-Scheduling problem for x with help of $\{S_{\tilde{w}}\}$
7: Fix optimal parameter $\mho$ as label x_v for w (i.e. select $A_{\mho}$ from g_w).
8: **return** $\mho$.

Algorithm 34 compute-$\beth_2$ for $w \in \mathcal{T}_{\mathcal{H}}$

Input: $\mathcal{T}_{\mathcal{H}}$, $w \in \mathcal{T}_{\mathcal{H}}$, optimal parameter Ω for parent.

1: **for** each $\tilde{w} \in child(w)$ **do**
2: $\mho :=$compute-$\beth_1(\tilde{w}, \Omega)$
3: compute-$\beth_2(\tilde{w}, \mho)$
4: **end for**

profile surfaces for w, which in turn will be used to fix the optimal parameters for w. The major difference to Alg. 32 is the non-recursive nature. One might be tempted to fix the optimal parameters during the recursion of Alg. 32 yet this would not yield an optimal schedule since $P_w = P_w(x)$. In other words the parent nodes parameter would need to be an optimal.
This algorithm will only work if the communication channels between a node and each of its children are independent with respect to the calculation of the profile surface, i.e. if the computation of a profile surface for child v is not influenced by a parallel computation of a profile surface for another child w.

Remark VIII.10. *If an algorithm of node u delegates work to a child v it must also accumulate the child's results before returning its own result, this holds recursively. Thus a profile surface S_u does not only represent the communication costs between u and v but also between u and all its computationally involved children. Furthermore it includes all overhead for data accumulation.*

Theorem 24. *If all parameter sets are finite, i.e. have a finite value set, then finding an optimal hypersystem schedule is NP-complete.*

Proof. Let us distinguish between two types of parameters;

- Parameters which define disjoint range subsets, e.g. K from the matrix multiplication example. Such parameters are referred to as *range parameters*.
- All other parameters which do not define range subsets, those are referred to as *tweak parameters*.

Let L be an arbitrary layer in a hypersystem, $v \in L$ an arbitrary node whose d children are symmetric with respect to their pa-

rameter sets S_i (i.e. all S_i have the same parameter types). The following cases must be distinguished

- **Only range parameters are in S_i**:
  This is a multidimensional form of MP.
- **One range parameter and one tweak parameter are in S_i**:
  The feasible parameters can be computed as follows. Following the strategy for MCKP encoding from theorem 22 one can create a feasible representation in polynomial time

$$
\begin{aligned}
I \quad = \quad & ((T_1(\tilde{k}_1^1, 1), 1), ..., (T_1(\tilde{k}_{l_1}^1, 1), 1), && \text{(VIII.36)}\\
& ((T_1(\tilde{k}_1^1, 2), 1), ..., (T_1(\tilde{k}_{l_1}^1, 2), 1), && \text{(VIII.37)}\\
& ((T_1(\tilde{k}_1^1, 3), 1), ..., (T_1(\tilde{k}_{l_1}^1, 3), 1), && \text{(VIII.38)}\\
& ..., && \text{(VIII.39)}\\
& ((T_1(\tilde{k}_1^1, m), 1), ..., (T_1(\tilde{k}_{l_1}^1, m), 1), && \text{(VIII.40)}\\
& (T_2(\tilde{k}_1^2, 1), 2), ..., (T_2(\tilde{k}_{l_2}^1, 1), 2), && \text{(VIII.41)}\\
& ..., && \text{(VIII.42)}\\
& (T_2(\tilde{k}_1^2, m), 2), ..., (T_2(\tilde{k}_{l_2}^1, m), 2), && \text{(VIII.43)}\\
& (T_d(\tilde{k}_1^d, 1), d), ..., (T_d(\tilde{k}_{l_d}^1, 1), d)) && \text{(VIII.44)}\\
& ..., && \text{(VIII.45)}\\
& (T_d(\tilde{k}_1^d, m), d), ..., (T_d(\tilde{k}_{l_d}^1, m), d) && \text{(VIII.46)}
\end{aligned}
$$

  Note that range and tweak parameters can always be represented by an enumeration starting at 1, in this example m represents the last index of the tweak parameter. For each T_i there exist m^d, with d being a system constant, thus the input can be computed in polynomial time.

All other possible cases follow directly from those above. In case of asymmetric children one can split the problem into subproblems for which the previous statements hold as well, the amount of subproblems is a system constant and scales the problem complexity linearly. □

Note that a hypersystem provides no explicit measure which states if the problem could be solved more efficiently with for example fewer vPPUs. The optimal schedule will always try to utilize all vPPUs, yet, the exclusion of vPPUs can be incorporated with adequate algorithm families and corresponding parameter sets.

VIII.16. Experiments and Cluster Distribution

The described approach was evaluated on a subset of IGOR which consisted of 8 nodes in total, these nodes were equipped as follows:

- 1 node: Intel i7 4770, 2x Radeon 7990 (4 GPUs), 64GB RAM
- 7 nodes: Intel i7 4770, 1x Radeon 7970, 16GB RAM
- 1 node: Intel i7 4770, 16GB RAM

The last node was used as a dedicated management node, in order to minimize communication costs all nodes were interconnected via a dedicated 1GBit Ethernet switch. The evaluation used the SimpleHydra framework [MH14a] and was performed under ArchLinux. Since each node was equipped with identical

GPUs the node-local scheduling problem became trivial, furthermore all nodes contained GPUs of the same kind which at the first look hinted that the cluster-local scheduling might be easily solvable as well. The first distribution strategy consisted of

- Let there be a total of d GPUs in the system and $d(v_i)$ the amount of GPUs in node v_i. Node v_i was assigned with $\frac{d(v_i)}{d}k$ rows of A.

The parameters f, μ, s for Alg. 30 were obtained via profile surfaces for a set of parameters (i.e. k values) which were to be expected during evaluation.

Motivated by the results from comparing optimized BLAS libraries with my approach a hybrid algorithm was designed. For matrix sizes below a certain threshold the developed approach should be applied, otherwise the optimized BLAS implementation. More formally; let there be n nodes v_i each with $d(v_i)$ identical GPUs and let $T \in \mathbb{N}$ be a threshold. Should the row count $\frac{k}{n \cdot d(v_i)}$ of A fall below T then Alg. 30 and 31 will be used, otherwise the node will execute an optimized BLAS routine. In order to utilize the maximal GPU performance all experiments were conducted with single precision accuracy.

VIII.16.1. Results

Fig. VIII.12 and VIII.13 show the execution times for the optimized BLAS library and my hybrid algorithm, respectively. The optimal threshold was defined to be at $k = 1280$ since beyond this point the efficiency of Alg. 30 dropped below that of the BLAS library. The new algorithm shows a significant improvement for small matrix problems. Note that both figures depict profile surfaces for a single node with only one algorithm step (multiplication) and two parameters (k and m). Each time value was

obtained by a profile surface on the GPU level, for those values which correspond to Alg.30 I obtained the corresponding values by finding the optimal values f, μ via the devices profile surface. As described before a symmetry argument was used to motivate the distribution scheme (i.e. the computation of an optimal schedule was avoided). Yet, the problem of assessing the communication costs between the computation nodes and the management node was also mentioned. The symmetry heuristic can only be upheld if the communication times fulfill:

- The values are always below the actual computation time on a node.
- The communication times do not vary too much with respect to the data amount which is send to all involved nodes.

We evaluated this by computing a profile surface on the management node (i.e. $T_{\mathcal{H}}$) with respect to the network latency. Fig. VIII.15 and VIII.14 depict the results, sadly both graphs clearly show that none of both demands can be fulfilled. An important observation is that only for the marginal subproblem sizes, i.e. $(1,1)-(m,1)$ and $(1,1)-(k,1)$. Since the distribution strategy splits along the k variable this forces us to distribute a single-row computation task (i.e. $k'=1$ for all nodes), which in turn requires a multi-GPU distribution with respect to m. Yet the expected marginal slack s_m (i.e. the gap between 0 and ΔL along the m dimension, see Fig. VIII.15) might be too low ($\mathbb{E}(s_m) = 0.4ms$) in order to guarantee positive ΔL values considering fluctuations in the communication cost. Thus in order to save the heuristic (and avoiding solving the scheduling problem) one is forced to reduce the communication costs between nodes, e.g. by exchanging the Ethernet interface for Infiniband.

The communication costs simply outweigh the low subproblem computation times.
Another solution would be to avoid distribution at all and use a single multi-GPU node, yet in this case the efficiency would be limited by the amount of GPUs. Viewing the facts in another light, one could relax the requirements and accept a distribution latency, in case of fixed k, l, m and a continuous data flow this latency would not increase and be carried throughout the computation stream. This in turn is equivalent to solving the scheduling problem only once.
We chose the latter solution and restricted (k, l, m) to $(50, 64, 1-48)$, i.e. k and l whereas the full m-range was kept. In order to find the best distribution the optimal values μ, f were stored in a cache (only one distributed cache since all nodes were equipped with the same GPUs), furthermore the profile surface $\Delta L' = S_1 + S_2$ was used (see Fig. VIII.15) as a virtual measure under the assumption that the communication costs would always outweigh the computation time. Note that $\Delta L'$ represents the profile surface for a single one-gpu node. Copying the results between host and GPU memory took at most $\approx 50\mu s$, which is below any computation time (the min. time was $93\mu s$). Since communication with GPUs occurs in parallel one can assume these time values for any multi-GPU systems. Thus one can use $\Delta L'$ for a node with n GPUs by transforming the k-coordinate with $\tilde{k}(k) := \frac{k}{n}$ (this assumes k is a multiple of n, which is an initial requirement nevertheless). Under the final assumption that communication with all nodes occurs in parallel the distribution problem can be solved by forming and solving a $d-Sum$ [PW10] instance on $\Delta L'$. Using a custom $d-Sum$ ([GS15]) solver I first

determined all feasible sets K^i of k', which are for example

$$\begin{array}{lcl} K^1 & = & \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 22\} \\ K^2 & = & \{1,\ 2,\ 3,\ 4,\ 5,\ 7,\ 10,\ 18\} \\ K^3 & = & \{1,\ 2,\ 4,\ 5,\ 8,\ 9,\ 10,\ 11\} \\ K^4 & = & \{1,\ 3,\ 5,\ 6,\ 7,\ 8,\ 9,\ 11\} \\ K^5 & = & \{2,\ 3,\ 4,\ 5,\ 6,\ 8,\ 9,\ 13\} \\ K^6 & = & \{2,\ 3,\ 4,\ 5,\ 7,\ 8,\ 9,\ 12\} \\ K^7 & = & \{2,\ 3,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10\} \\ ... & = & ... \end{array} \tag{VIII.47}$$

Afterwards all sets without at least one element $e : 4|e$ were filtered out, in this case only one set was removed. An optimized schedule can be constructed by selecting an element K^i which minimizes the corresponding time $\sum_{j=1}^{8} \Delta L'(K_j^i, m)$. We have 8 nodes, yet one is equipped with 4 GPUs, i.e. one node will distribute its k' rows onto 4 devices, which in turn will reduce the computation time. In order to account for that I created additional sets K_j^i for each K^i, one set for each element which is divisible by 4 (there are at most 8 such sets for each K^i). A few examples are depicted in Tab. VIII.3

Note that the introduced latency only depends on m and is given by $\max_{k \in K} S_2(k, m)$. Since the determined values are based on an idealized model I evaluated the actual computation time in the described cluster system. Fig. VIII.16 shows two lines, the dashed line represents the time of my theoretical model while the solid lines indicate the actual computation time when using the determined schedule. The difference to the calculated times can be explained by the fact that multiple communications occur in parallel, i.e. messages arrive asynchronously and the operating system may perform optimizations on the queued messages. Furthermore the profile surface exhibits a large vari-

Table VIII.3.: Several indices i for an optimal set K^i with respect to m. The time t represents the minimized total computation time when distributing a matrix multiplication for $k = 50, l = 64$. The value of j refers to the set K^i_j which was constructed from K^i. Note the tendency to split the largest k value onto multiple GPUs.

m	3	4	5	6	7	8
i	68	89	47	90	116	10
j	2	2	2	3	1	1
t (ms)	2.521	2.529	2.516	2.532	2.588	2.592

ance for small values (Fig. VIII.17), the profile surface for a single node was used as source for the described heuristic. For small values of m the schedule induces higher times, whereas for larger values the schedule outperforms the prediction. This indicates that my heuristic may indeed yield an approximation to the optimal schedule.

The evaluated times in Fig. VIII.16 indicate that a profile surface for the communication between a single node and the management node yields only a very coarse correlation between the predicted optimal times and the measured times. Yet with respect to my model the profile surfaces should be strong representations of a system's behaviour. As mentioned earlier the communication channels are assumed to be independent (and symmetric) to each other, i.e. measuring channel 1 and 2 in parallel should not produce results which are different to the single case. Since the curves in Fig.VIII.16 indicate interdependent channels the measurement strategy was changed. Instead of considering communication between one cluster node and the master node. The time for each possible schedule K^i was mea-

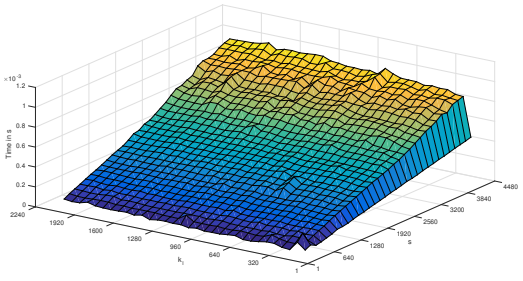


Figure VIII.12.: Execution speeds of the optimized AMD OpenCL SGEMM routine with increasing sizes of k and m for $l = 64$.

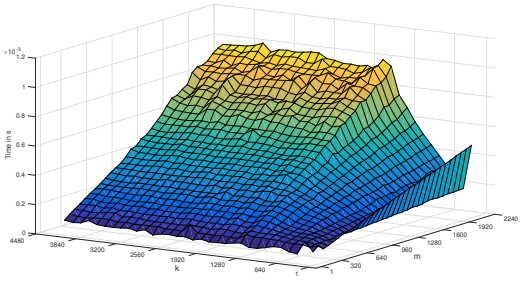


Figure VIII.13.: Execution speeds of the hybrid algorithm with increasing sizes of k and m for $l = 64$. The processing time for smaller matrices has been significantly reduced.

sured with all 8 nodes working in parallel. Fig.VIII.18 shows a resulting profile surface, note the variance along both dimen-

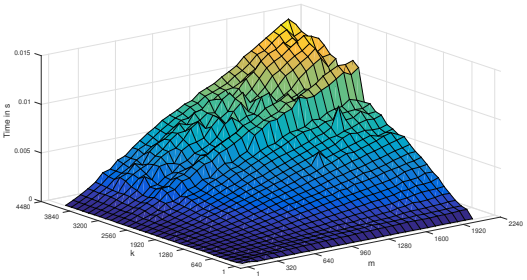


Figure VIII.14.: This graph shows the complete communication costs between the management node and a computation node when distributing matrix subproblems, i.e. each z-value represents the amount of time it requires to transfer the matrix data (including all program parameters) and receiving the results. Note the linear relation between data volume and time for larger matrices, this linearity occurs since the transmission time outweighs the low management overhead.

sions (i.e. m and k). Using such a surface one can determine an optimal schedule, i.e. find the optimal K^i for each m. As depicted in Fig. VIII.19 the measured results correspond much more with the predicted optimal times. Yet due the variance in the measurement one has to expect a discrepancy, in order to reduce this effect I applied rotating schedules along the m-dimension. In other words, I measured n profile surfaces S_i and calculated an optimal schedule $\beth_i$ for each one, during the evaluation $\beth_{m \mod n}$ was used for each m. This resulted in a significant reduction of the Root Mean Square Error (RMSE) between the expected prediction and the measurement. For increasing n

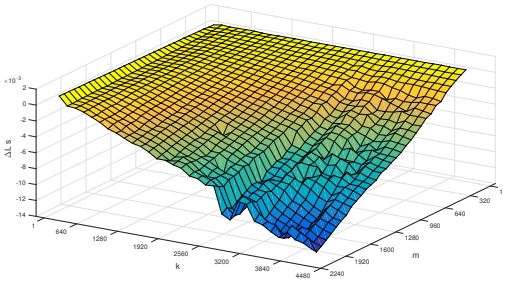


Figure VIII.15.: The difference $\Delta L := S_1 - S_2$ between the profile surface S_1 in Fig. VIII.13 and the surface S_2 in Fig. VIII.14. Note that almost every value lies below 0, i.e. the network latency surpasses almost every computation time (except for very small problem sizes).

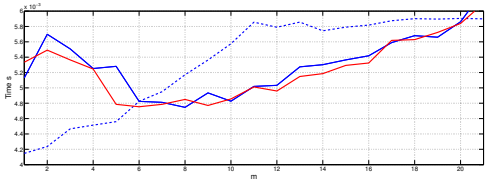


Figure VIII.16.: Execution speeds of the calculated model (dashed line) and the actual application of it in a multi-node cluster. The red line represents a multi-threaded communication while the blue line shows the time when only a single thread handles all the requests on the management node.

the RMSE decreased as shown in Tab. VIII.4, it also indicates that for $n > 4$ the RMSE might increase.

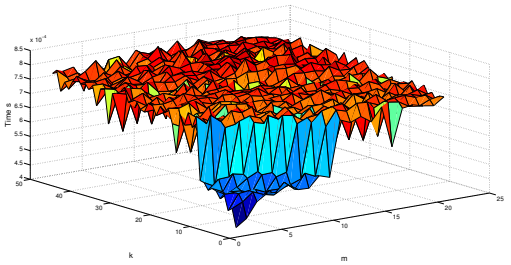


Figure VIII.17.: Complete communication costs between the management node and a computation node for small values of k and m.

Despite the large amount of idealizations the $d-Sum$ instance

Table VIII.4.: The RMSE between the expected optimal times $\mathbb{E}(P)$ and a measurement M_n with n rotating schedules. Each measurement was repeated 2000 times.

n	1	2	3	4	5
RMSE $\mathbb{E}(P) - M_n$ (ms)	1.594	1.523	1.451	1.410	1.49
overall time (ms)	110.53	110.25	110.14	109.84	110.34

still had to be solved, yet despite this apparent drawback one should note that it needs to be solved only once. Furthermore one can utilize the $d-sum$ solution for k' when confronted with larger values $k' < k$ by simply splitting the initial problem into subproblems of size k/k' and deploy them sequentially to the nodes. By using a cache for the lower vPPU layers one can significantly reduce the computation costs for approximating an ideal schedule. Yet with rising system heterogeneity the required memory amount will also rise.

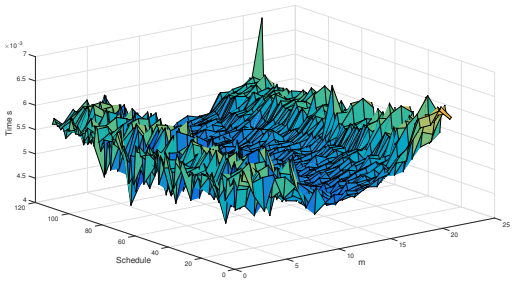


Figure VIII.18.: Complete communication costs between the management node and a single computation node for all possible schedules.

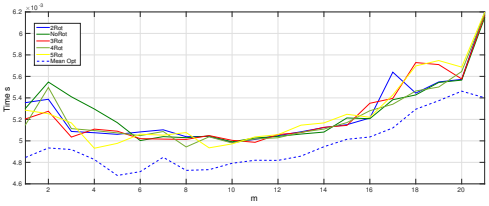


Figure VIII.19.: The expected optimal communication times (dashed line), and the evaluated times for up to 5 rotating schedules.

In order to counter the problem of measuring every possible parameter setup (especially without any heuristic) from a large set of possible values I evaluated the possibility of using interpolation on a coarse measurent. In order to get comparable results I first measured on a range of $m = [1, ..., 210]$, afterwards the

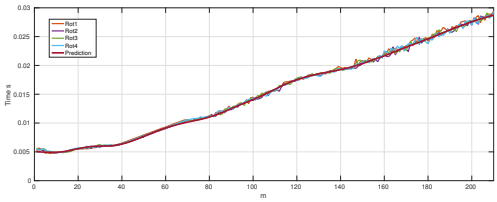


Figure VIII.20.: The expected optimal communication times (red line), and the evaluated times for up to 4 rotating schedules. The predicted times were obtained from sparse measurements via a neural network.

measured times were subsampled equidistant with a step size of 5. The subsampled data was used to train a neural network (1 hidden layer with 50 neurons) as predictor for the measurement gaps. The results are shown in Fig VIII.20 and were obtained by training the network via Levenberg-Marquardt with a maximal gradient error of 10^{-8}. The actual execution time was predicted in a range which should be acceptable for most application, especially if one considers the fact of sparse measurement data. Table VIII.5 shows the corresponding RMSE, as before no improvement was obtained beyond 2 rotating schedules, yet depending on the actual schedules the amount of helpful schedules may change.

VIII.17. Conclusion

Motivated by the results of [MH14b] I developed a hybrid algorithm for distributed matrix multiplication. In order to find the optimal parameters for each subalgorithm I created the abstract

Table VIII.5.: The RMSE between the expected optimal times $\mathbb{E}(P)$ and a measurent M_n with n rotating schedules. Each measurement was repeated 2000 times.

n	1	2	3	4
RMSE $\mathbb{E}(P) - M_n$ (ms)	4.609	3.649	3.722	4.121
overall time (s)	3.234	3.197	3.218	3.212

concept of hypersystems. Building upon this model I conducted a theoretical analysis and stated a general algorithm which obtains the optimal parameters through practical evaluations in the form of profile surfaces. These profile surfaces represent overall (locally optimized) execution times of subsystems, i.e. including overhead such as communication costs, for a certain choice of parameters. Sadly the algorithm proved to be NP-complete, yet due to symmetries in the system setup I attempted to obtain an estimation of optimal parameters. The evaluation of communication costs showed that even for small subproblem sizes the reduced computation is nearly outweighed. Thus I relaxed the demands and targeted a minimization of communication costs, i.e. communication latency were accepted. Such a decision is tantamount to solving the scheduling problem. With relatively keen assumptions about multi-GPU node-local communication costs and by using only the profile surface for communication costs between the management node and a single computation node, I showed that the system behaviour could be predicted in a coarse maner. The discrepancy between the prediction and the actual result indicated dependent communication channels, i.e. a measurement for a single node was not representative for the communication channels behaviour in case of multiple parallel communications. We solved this problem by considering a

profile surface of all possible schedules, the resulted prediction was much closer to the final result yet due to the measurement variance prediction outliers were still obtained. With the help of rotating schedules I could reduce the variances effect and obtained results with a reduced overall time.
The stated results also show that the hypersystem model can only optimize the parameters for an already existing algorithmic structure, it can not judge if the structure itself can be optimized. This became apparent by comparing execution times for multiplication of small matrices and the involved communication costs. If possible, small matrices should not be distributed to nodes at all but rather be computed locally on the management node (on a single GPU). Yet it is possible to include parameters into each subalgorithm which give rise to inherent structure modification, e.g. parameters which decide if a subalgorithm should be used at all.
It was already shown in [MH14b] that special cases of hypersystems provide a beneficial design approach for SIMD architectures. Yet it remains to be seen if the general hypersystem model provides the same gain for large distributed systems with deep hierarchies. Additionally it should be studied how the approach performs on distributed systems with high performance communication interfaces, in this context the question remains if my heuristic of rotating schedules can be upheld for a large node count and more irregular profile surfaces.

IX. Boosting HOG-based Algorithms

"Any complex adaptive system can, of course, make mistakes in spotting regularities. We human beings, who are prone to superstition and often engage in denial of the obvious, are all too familiar with such errors." (Murray Gell-Mann, [GM02])

Object detection systems which operate on large data streams require an efficient scaling with available computation power. I analyze how the use of tile-images can increase the efficiency (i.e. execution speed) of distributed HOG-based object detectors. Furthermore I discuss the challenges of using the developed algorithms in practical large scale scenarios. I show with a structured evaluation that my approach can provide a speed-up of 30-180% for existing architectures. Due to the its generic formulation it can be applied to a wide range of HOG-based (or similar) algorithms. In this context I also study the effects of applying my method to an existing detector and discuss a scalable strategy for distributing the computation among nodes in a cluster system.

IX.1. Introduction and Previous Work

In this section I present a method which is capable of boosting the efficiency of existing HOG detectors without structurally modifying them. Furthermore I discuss a scalable strategy for distributing the computation among nodes in a cluster system. Section IX.2 will briefly discuss the common approach to calculate HOG features on massively parallel architectures with a special focus on GPUs. The following section IX.3 introduces my framework and explains the challenges of practical application. This chapter concludes with sections IX.5 and IX.9, which present the results on standard image databases and an outlook for additional research, respectively.

IX.2. Histograms of Oriented Gradients

In order to understand my motivation for the so called tile-image approach one has to understand the general structure of a GPU implementation for HOG-based algorithms. Thus I will briefly explain the classic algorithm and use this description to introduce the common "tricks" of GPU-ports.

IX.2.1. The Algorithmic Structure

The scaling itself with $\mathcal{O}(I_w \cdot I_h)$ weighs heavily on the overall complexity let alone the histogram calculation for each window position. The expression 'I_c not covered' refers to the check if there exists an image position which can be covered by the detection window if it is shifted in multiples of win_s (note that $win_s = (x_s, y_s)$ is a 2-tuple which defines a horizontal and a vertical shift quantum, i.e. shifting is done with two for-loops).

In order to derive an expression for the complexity of Alg. 13 the following facts must be stated

Theorem 25. *Let* $S_e = \min\{I_w/win_w, I_h/win_h\}$ *and* I_c *an image on scale level* c*. Furthermore let* $b_w, b_h, b_x, b_y, c_w, c_h, c_x, c_y, n$ *be the block width, block height, block stride x, block stride y, cell width, cell height, cell stride x, cell stride y and bin count respectively. The following statements are true*

- *The total amount of scaling steps is* $A := \lfloor \log(S_e)/\log(s) + 1.0 \rfloor$
- *Shrinking the image has a complexity of* $B := \mathcal{O}(I_w \cdot I_h)$*.*
- *There are* $C := \frac{I_{c,w} - win_w + x_s}{x_s} \cdot \frac{I_{c,h} - win_h + y_s}{y_s}$ *window positions in* I_c*.*
- *Computing* I_G *with two 1-dim convolution masks requires* $\mathcal{O}(win_w \cdot win_h)$ *operations.*
- *There are* $D := \frac{win_w - b_w + b_x}{b_x} \cdot \frac{win_h - b_h + b_y}{b_y}$ *blocks in each window.*
- *There are* $E := \frac{b_w - c_w + c_x}{c_x} \cdot \frac{b_h - c_h + c_y}{c_y}$ *cells in a block.*
- *Computing the histogram for a single window requires* $\mathcal{O}(D \cdot E \cdot c_w \cdot c_h)$ *operations.*
- *Each histogram has* $F := D \cdot E \cdot n$ *elements.*

The total complexity of a single HOG iteration for a single image is

$$\mathcal{O}(A \cdot (B + C \cdot (win_w \cdot win_h + D \cdot E \cdot c_w \cdot c_h + F))) \quad \text{(IX.1)}$$

From the complexity expression is becomes clear that each HOG parameter plays an important role for the algorithm's runtime, which is anything but small.

IX.2.2. GPU Implementation

The usual structure of a GPU implementation is depicted in Alg. 35. Since a modern GPU features several thousand execution units one attempts to delegate at least one operation onto each execution unit. For some algorithm steps there are more operations than execution units, for others there may be more execution units than operations. Yet GPUs follow the SIMD approach, which enforces several restrictions onto the algorithm's structure. Explaining these challenges would be beyond the scope of this thesis, the interested reader might refer to [PR09].

Algorithm 35 GPU HOG

Input: HOG parameters: $s, win_w,$
1: $win_h, win_s, ...$
Output: l
2: $I_c = I$, $\tilde{s} = s^0$, $l = \emptyset$;
3: **while** Detection window fits into current image **do**
4: Shrink$_{GPU}(I_c,\tilde{s})$; $\rightarrow c_{shrink}$
5: Compute gradient image
6: $I_G = (I_A, I_\phi)$ for I_C
7: on GPU; $\rightarrow c_{grad}$
8: **par. for** all block positions i in I_G **do** $\rightarrow c_{hist}$
9: Calculate histograms
10: $H_i = H_i(I_G)$;
11: **end par. for**
12: **par. for** all block positions i in I_G **do** $\rightarrow c_{norm}$
13: Normalize histograms
14: $H_i = H_i(I_G)$;
15: **end par. for**
16: **par. for** all window positions i in I_G **do** $\rightarrow c_{classify}$
17: Classify H_i, add results to l;
18: **end par. for**
19: $I_c = I$, $\tilde{s} = \tilde{s} * s$;
20: **end while**
21: Group elements from l via
22: Mean-Shift $l = MS(l)$; $\rightarrow c_{group}$

The parallel for-loop in Alg. 35 only works under a "trick" common to all implementations which attempt to reach state of the art detection speed; the window stride must equal the block stride. This allows to precompute the block histograms for all window positions on a single scale. A complete analysis of this structure, e.g. in the PRAM [Gib89] model, is beyond the scope of this thesis. One should note that parallelization usually mod-

ifies the efficiency only by a factor $c = c(\tau, \mathcal{W})$ with τ being the shader count and $\mathcal{W}$ the set of hog parameters (the $\mathcal{O}$ notation omits such factors). The comments in Alg. 35 introduce such constants, the histogram normalization is usually achieved by an additional phase (due to the previously mentioned architecture restrictions), yet for the sake of simplicity it will be regarded as one stage. Note that each constant c_i in Alg. 35 has a different optimal shader count s_i for which c_i would equal 1. To illustrate this; $c_{shrink} = 1$ for $s_{shrink} = I_w \cdot I_h$ while $c_{classify} = 1$ for $s_{classify} =$"histogram size". If a GPU would provide $\max s_i$ shaders and $\forall i, j : s_i = s_j$, the complexity of Alg. 35 would be Θ(scaleCount), which is unfeasible as memory latencies and the SIMT programming model must be considered as well, let alone the fact that current GPUs provide only a relatively small amount of shaders. One implication is that not all phases can be equally efficient on the GPU, which is an inherent aspect of most GPU-based multi-phase algorithms, e.g. [YSMR10]. It is difficult to counter this problem, usually it is omitted for reasons of convenience or shadowed by arguments of a high speed up compared to a CPU implementation. Yet when it comes to practical (industrial) applications of such algorithms, it is often desired to save as much time as possible without overstepping a system's tolerance boundaries.

IX.2.3. Efficiency Factors

Notice that by theorem 25 the complexity is mainly influenced by the image size, since A, B and C directly depend on it. Several steps in Alg. 35 work on global memory, which exhibits very high latencies, for small amounts of schedules threads this results in large computation delays as shader units will have to wait for requested data. By scheduling a large number τ' of

threads it is possible to mask these delays especially in combination with techniques such as memory coalescing. Yet with increasing τ' the efficiency increase will stagnate as the overhead for scheduling will outweigh the gain of masking latencies. Note that due to the parallel approach in Alg. 35 all elements within the while-loop are influenced by the image size. Thus I argue that an existing HOG implementations efficiency can be increased by varying the image size.

IX.3. Cluster-based Computation

The structure of Alg. 13 gives rise to many different distribution strategies within computing clusters. One such method would be a strategy similar to pipelines in microprocessors, where each phase would be executed by a dedicated node. The state of the art total execution time for a classical HOG detection is roughly 70ms (GPU, image size 1600x1200, [HMH13]), whereas a multicore CPU requires 180ms (CPU, image size 320x240, [KSLO12]). Thus, not considering inter-node communication costs, it would require at least 10 CPUs to deliver the same performance. Motivated by these numbers I focused entirely on nodes equipped with multiple GPUs. This decision limits the distribution scheme to methods which compute entire HOG runs on each node, since communication latencies (GPU↔host and node↔node) outweigh the execution times for single phases.

IX.3.1. Efficiency through Tile Images

The concept of tile images is defined by

Definition 14. *Let $\mathcal{I}$ be a set of arbitrary images I_j. A tile image $I_\mathcal{T}$ is defined as an image which contains every image of $\mathcal{I}$ exactly once without overlap. The set $\mathcal{I}$ is called the base of $I_\mathcal{T}$ with each element being referred to as base image. The density $\delta_\mathcal{T}$ of $I_\mathcal{T}$ is defined as the area of $I_\mathcal{T}$ which is covered by any image from $\mathcal{I}$*

$$\delta_\mathcal{T} := \frac{1}{|I_\mathcal{T}|} \sum_{I \in \mathcal{I}} |I| \tag{IX.2}$$

Note that this definition imposes no restriction onto the maximal size of $I_\mathcal{T}$, yet since one is interested in minimizing the density $\delta_\mathcal{T}$ the maximal width and height are given by $\sum_{I \in \mathcal{I}} I_w$ and $\sum_{I \in \mathcal{I}} I_h$, respectively. This is a classic packing problem and corresponding strategies as well as algorithms have been studied by [LMM02] or [EG75]. Yet in the current context this problem becomes much more difficult due to time constraints.
Let us assume a HOG detector H requires n time units to process an image I and m units for an image J. The $I - J$ efficiency of H is defined as

$$E_{I,J}(H) := \frac{J_w}{I_w} \cdot \frac{J_h}{I_h} \cdot n - m \tag{IX.3}$$

The first product represents the amount of times that J can be fitted into I, thus $E_{I,J}(H)$ states the amount of time units which are saved if one processes one image J instead of $\frac{J_w}{I_w} \cdot \frac{J_h}{I_h}$ images I. For tile images this can be generalized to

Definition 15. *Let $I_\mathcal{T}$ be a tile image with base $\mathcal{I}$, H a HOG detector and $t_H(I)$ the processing time which H needs for I. The*

$I_\mathcal{T}$ efficiency of H is defined as

$$E_{I_\mathcal{T}}(H) := (\sum_{I \in \mathcal{I}} t_H(I)) - t_H(I_\mathcal{T}) \tag{IX.4}$$

Remark IX.1. *If $E_{I_\mathcal{T}}(H) = 0$ then H performs equally fast as if being called with single images, if $E_{I_\mathcal{T}}(H) > 0$ then H uses less time then for all single images. Note that the density $\delta_\mathcal{T}$ has an implicit effect on $E_{I_\mathcal{T}}(H)$, if for example $\mathcal{I}$ contains only one image I and $|I_\mathcal{T}| = 50|I|$ it is very likely that $E_{I_\mathcal{T}}(H) < 0$ whereas for $\delta_\mathcal{T} = 1$ one obtains $E_{I_\mathcal{T}}(H) = 0$. Thus it is desirable to aim for $\delta_\mathcal{T} = 1$.*

In case of $E_{I_\mathcal{T}}(H) > 0$, i.e. the tile image yielded an efficiency gain, the computation of $I_\mathcal{T}$ should not take longer than $E_{I_\mathcal{T}}(H)$, otherwise the gain would be canceled out.

IX.3.2. Boundary Detections

A tile image will contain hard boundaries which can effect the detection results. In order to filter these false-positives I propose the use of tile grids.

Definition 16. *Let $I_\mathcal{T}$ be a tile image with base $\mathcal{I}$. The tile grid $G_\mathcal{T}$ is a set of $|\mathcal{I}|$ 4-tuples (x, y, w, h) where x, y represents the position of a base images top-left corner within $I_\mathcal{T}$ and w, h the base images size. Each tuple is called a base-rectangle.*

I propose the following approach to purge false-positive detections from a tile image. Let $\{d_i\}$ be the detections on $I_\mathcal{T}$ and I a base image with corresponding base-rectangle r_I, A detection d_j is associated with I iff $|d_{j,A} \cap r_{i,A}|/|d_{j,A}| = 1.0 \wedge |d_{j,A}| \leq |r_{i,A}|$, all unassociated detections are omitted ($d_{j,A}, r_{i,A}$ denote the corresponding areas).

IX.3.3. Computing Tile Images

A restriction which was not mentioned so far is that the detector H remains unchanged in its implementation. This reduces the strategies for computing a tile image since the memory structure of each base image must be retained in the tile image. A simple memory copy into the linearized tile image is impossible as the tile images row stride differs from the base image. Thus one has to copy the images on a pixel- or row-base. Another observation comes from Alg. 35, it is obvious that the image size effects the runtime, yet the algorithm does not favor any specific image proportion. Expressed differently; only the actual image size, i.e. $I_w \cdot I_h$ effects the performance. This leaves only the question of how many lines of base images should be used in the tile image in order to maximize its density.
In the case of identically sized base images one can sequentially enqueue (i.e. concatenate them in one line) as many as required in order to maximize the efficiency (this implies $\delta_{\mathcal{T}} = 1$). Using this strategy for differently sized images will a) force the image height to be equal to the largest image height within the image base and b) result in a density of

$$\delta_{\mathcal{T}} = \frac{|I_{\mathcal{T}}| - \sum_{I \in \mathcal{I}} I_w \cdot (I_{\mathcal{T},h} - I_h)}{|I_{\mathcal{T}}|} < 1 \qquad \text{(IX.5)}$$

Each base images difference in height to $I_{\mathcal{T}}$ will reduce the density. I will not theoretically elaborate on this problem but rather discuss a solution for a concrete scenario.

IX.3.4. Cluster Distribution

It was shown by [HMGH14] that large surveillance systems do not only generate huge amounts of data but also that it requires

problem specific engineering in order to manage these amounts. In order to discuss a possible solution for the computation of tile images in realistic scenarios, I assume a system similar to the one described in [HMGH14]. Let us assume we have k video streams where each one provides images of constant size, additionally we assume to have k detectors each with a dedicated computation device. Having an equal amount of detectors is a realistic assumption if one desires to perform real time object detection, since each detection takes around $70ms$ (with a single Radeon 7970 GPU) one can process at most 14 frames per second. I propose a cluster structure as depicted in Fig. IX.1

The management node represents a central hub which receives detection requests and redistributes them onto the designated nodes. A detection requests is a 2-tuple containing an image and meta data such as sender address and image number. Since the image dimensions of each camera are known in advance it possible to optimize each node for a single camera. One such optimization is the size of each nodes image buffer, which holds a pre-allocated tile image into which each received image is written. Once the buffer has been filled ($\delta_{\mathcal{T}} = 1$) the tile image will be send to the detector for processing.
In order to understand how the optimal buffer size is determined one has to analyze several system attributes. Let us assume a continuous stream of images to node n_i, which contains a buffer of size z_1, the delay T until a client receives the results for the first batch of z_1 images is

$$T := (t_{prop,\uparrow} + t_{copy}) + (z_1 \cdot t_{tick} + t_{process,z_1} + t_{prop,\downarrow}) \quad \text{(IX.6)}$$

$t_{prop,\uparrow}$ is the time a request needs to arrive at the designated node, t_{copy} the time to copy an image into the tile image, $t_{process,z_1}$ the time to perform the detection process and $t_{prop,\downarrow}$ the time

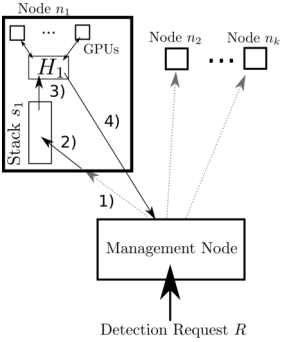


Figure IX.1.: A Beowulf cluster system for handling up to k parallel video streams. Each detection request is delegated by the the management node to the corresponding processing node (step 1). Each processing node contains a buffer stack whose size z_1 is optimized for the node-local detector H, each received image will be pushed onto the stack (step 2), which is essentially a tile image. Once the tile image has been filled it will be processed by H, which utilizes multiple node-local GPUs (step 3). Afterwards the results for all z_1 images will be returned to the request source (step 4).

until the nodes response reaches the recipient. The first term is an initial latency T_1, which is carried throughout the remaining stream while the second term represents the time T_2 until z_1 image requests have been issued by the sender and the detection results have been received (t_{tick} represents the tick rate). The delay until the client receives the second batch is $z_1 \cdot t_{tick} + t_{process,z_1} + t_{prop,\downarrow}$ relative to the first batch. In order to harness as much computational power as possible a larger z_1 is desired, yet it will leave the sender waiting for T_2 time units until the next batch of results arrives, this in turn makes a smaller z_1 favorable.

It must be pointed out that this equation only holds as long as $T_1 \leq t_{tick}$, otherwise each received image would be delayed by an additional amount of $T_1 - t_{tick}$ time units. This implies that one can process higher image frequencies if the systems performance is large enough, i.e. one should be motivated to decrease T_1. Furthermore it holds that for larger images a smaller amount of images needs to be buffered.
Although T_2 increases with higher values for z_1 the efficiency $E_{\mathcal{T}}(H)$ increases as well, thus a client may wait longer for receiving the next batch of detection results yet this time will be smaller compared to the situation where the client would obtain z_1 detections without using the tile image. As depicted in Fig. IX.1 a node might contain multiple GPUs here, independent of the workload distribution scheme, e.g. distributing single HOG phases or simply using multiple stacks, the GPUs performance will be determined by the amount of data which is available to the GPUs. In case of identical devices this might result in a linear scaling of T_2 with respect to the device count.

IX.3.5. Algorithmic Details

This section elaborates on the algorithms which determine t_{copy} and $t_{process,z_1}$. The remaining times $t_{prop,\uparrow}$ and $t_{prop,\downarrow}$ are determined mainly by the data structures which are used in order to look up the correct node or sender, respectively. When a node receives a request it executes the steps in Alg. 36. Since a continuous image stream is assumed it must be assured that any image which might arrive during an active detection are still enqueued on the node, i.e. at least two threads are active on Alg. 36. This approach works of course only if the detection and copying of $z_1 - 1$ images from $\mathcal{B}$ into $I_{\mathcal{T}}$ requires less than $(z_1 - 1) \cdot t_{tick}$ time units, i.e. $(z_1 - 1) \cdot t_{\mathcal{B}} + t_{process,z_1} \leq (z_1 - 1) \cdot t_{tick}$. Since

usually $t_{\mathcal{B}} = t_{copy}$ one gets the following restrictions

$$(t_{prop,\uparrow} + t_{copy}) \leq t_{tick} \quad \text{(IX.7)}$$

$$(z_1 - 1) \cdot t_{copy} + t_{process,z_1} \leq (z_1 - 1) \cdot t_{tick} \quad \text{(IX.8)}$$

The second restriction should be fulfilled implicitly if $t_{prop,\uparrow} + t_{prop,\downarrow} \leq t_{copy} + t_{process,z_1}$. Should one be able to ensure that $t_{copy} + t_{process} \leq t_{tick}$ than the multithreaded approach can be omitted completely.

Algorithm 36 Push on stack

Input: Request
$R = (I, (senderID, imgNo))$
if no detection runs **then**
  copy I into $I_{\mathcal{T}}$;
  register I in $G_{\mathcal{T}}$;
  if stack full **then**
    indicate running detection;
    execute detection on $I_{\mathcal{T}}$;
    if $\mathcal{B}$ contains images **then**
      copy all images from $\mathcal{B}$ into
      $I_{\mathcal{T}}$ and register them in $G_{\mathcal{T}}$;
    end if
    indicate no running detection;
  end if
else
  buffer image in local buffer $\mathcal{B}$;
end if

IX.4. Reducing Redundant Computations

As mentioned in section IX.2.2 state of the art GPU implementations assume that; the window stride must equal the block

stride. With this assumption one can precompute the gradient histograms for all window positions on a single scale. One can utilize this in order to use multiple detectors with identical HOG parameters on a single GPU. Alg. 37 shows the strategy in more detail. The main difference lies in the classification step where the precomputed histograms will be used in different Support Vector Machines (SVMs) d. With this approach one can significantly reduce the overhead in comparison to a sequential execution of Alg. 35.

Algorithm 37 GPU HOG 2

Input: HOG parameters: $s, win_w,$
1: $win_h, win_s, ...$
Output: l
2: $I_c = I$, $\tilde{s} = s^0$, $l = \emptyset$;
3: **while** Detection window fits into current image **do**
4: Shrink$_{GPU}(I_c,\tilde{s})$; $\rightarrow c_{shrink}$
5: Compute gradient image
6: $I_G = (I_A, I_\phi)$ for I_C
7: on GPU; $\rightarrow c_{grad}$
8: **par. for** all block positions i in I_G **do** $\rightarrow c_{hist}$
9: Calculate histograms
10: $H_i = H_i(I_G)$;
11: **end par. for**
12: **par. for** all block positions i in I_G **do** $\rightarrow c_{norm}$
13: Normalize histograms
14: $H_i = H_i(I_G)$;
15: **end par. for**
16: **for** all detectors d **do**
17: **par. for** all window positions i in I_G **do** $\rightarrow c_{classify}$
18: Classify H_i with SVM d, add results to l;
19: **end par. for**
20: **end for**
21: $I_c = I$, $\tilde{s} = \tilde{s} * s$;
22: **end while**
23: Group elements from l via
24: Mean-Shift $l = MS(l)$; $\rightarrow c_{group}$

The following theorem gives a more precise description

Theorem 26. *Let $\mathcal{D} = \{d_1, ..., d_n\}$ be a set of n HOG detectors with identical parameters, e.g. window size or window stride. The usage of Alg. 37 reduces the complexity by $(n-1)\cdot(c_{shrink}+$*

$c_{hist} + c_{norm}$) *time units compared to a sequential execution of Alg. 35 for each* $d_i \in \mathcal{D}$.

Proof. The complexity of a single scale step within Alg. 35 can be expressed through $c_{shrink} + c_{hist} + c_{norm} + c_{classify}$. This implies $n \cdot (c_{shrink} + c_{hist} + c_{norm} + c_{classify})$ for a sequentially executed scale step over all detectors. By using Alg. 37 the image shrinking, histogram calculation and normalization must be done only once, thus one saves $(n-1) \cdot (c_{shrink} + c_{hist} + c_{norm})$ time units for all detectors on a single scale. This in turn implies a complexity of $(c_{shrink} + c_{hist} + c_{norm}) + n \cdot c_{classify}$. Since the amount and form of scale steps remains unchanged in both algorithms, the previous statement about saved time units holds for both algorithms. □

IX.5. Evaluation

The described cluster architecture was implemented by using the SimpleHydra framework [MH14a], it consisted of 4 identically equipped processing nodes and a single management node (again a subset of the IGOR cluster). The processing nodes were equipped with one Radeon7970 GPU, a Core-i7 3.5 GHz CPU, 16GB RAM and ran ArchLinux with a 3.16 Kernel. The management node simulated the requests by using a local image database of images obtained from 4 different cameras on a small airport. Each camera sequence consisted of 1096 single images, which were streamed to the detector in chronological order of their recording (streaming was done with a frame rate of 10fps, which is the native speed of an AVT surveillance camera). I used a highly optimized HOG implementation which followed the original description by [DT05]. The following aspects were studied:

How different does an already optimized detector H behave for a sequence of tile images? How does $E_{\mathcal{T}}(H)$ develop? What values should be expected for $t_{copy}, t_{process,z_1}, t_{prop,\uparrow}, t_{prop,\downarrow}$? How low can t_{tick} get with respect to the latencies of my system? What is the behaviour of a generally optimized detector H for a sequence of tile images containing interleaved images of multiple cameras?

IX.5.1. Results

Table 1a shows the speed-up for images of size $1600x1200$, the efficiency increases continuously yet the increase begins to stagnate with more than 8 images (see the values of $\Delta E_{\mathcal{T}}(H)$). Using my strategy one can except to save $\approx 30\%$ of computation time. The gain becomes even more significant for smaller images, Tab. 1b shows the results for using images of size $640x480$. Using smaller images one can expect a speed-up of up to $\approx 180\%$. This difference in efficiency gain can be simply explained by the fact that a single small image will underutilize the large number of GPU shaders more than a large image, thus $E_{\mathcal{T}}(H)$ increases more significantly in relation to this time. With even more shader units the same effect is to be expected for larger images.

Fig. IX.2 shows that an example tile image and the corresponding detection, as indicated by that image there have been 0 detections on all image boundaries across all evaluated camera streams without even using the image grid $G_{\mathcal{T}}$. This shows that conventional grouping methods such as mean shift are likely to be sufficient in order to prevent boundary detections. The actual amount of boundary detections is depicted in Fig. IX.3, one can see that such detections indeed occur. If not filtered out before applying grouping procedures these detections may influence the final results.

In order to study the effect of boundary detections I evaluated

Table IX.1.: Processing times for differently sized tile images. The numbers in column T represent the stack size in units of images (each of size 1600x1200), t_{single} the time when single images would be used and $\Delta E_{\mathcal{T}}(H)$ the efficiency increase in relation to the last table row.

T	$t_{process,z_1}$	t_{single}	$E_{\mathcal{T}}(H)$	$\Delta E_{\mathcal{T}}(H)$	rel. speed-up
1	0.0429 s	0.0429	0.0000 s	-	-
2	0.0700 s	0.0858	0.0158 s	-	18.41 %
3	0.0978 s	0.1287	0.0309 s	95.57 %	24.01 %
4	0.1257 s	0.1716	0.0459 s	48.54 %	26.75 %
5	0.1542 s	0.2145	0.0603 s	31.37 %	28.11 %
6	0.1822 s	0.2574	0.0752 s	24.71 %	29.22 %
7	0.2109 s	0.3003	0.0894 s	18.88 %	29.77 %
8	0.2384 s	0.3432	0.1048 s	17.23 %	30.54 %
9	0.2677 s	0.3861	0.1184 s	12.98 %	30.67 %
10	0.2947 s	0.4290	0.1343 s	13.43 %	31.31 %

the amount of additional detections with the following metric

$$\Delta D(I_{\mathcal{T}}, I) := |H_I(I)| - |H_I(I_{\mathcal{T}})|, I \in \mathcal{I} \qquad \text{(IX.9)}$$

with $H_A(I)$ being the set of detections using detector H on image I and restricting the obtained results onto the area of image A. A positive value means that the tile image yielded less detections than the single image, a negative value indicates more results on the tile image while a 0 corresponds to an identical amount of detections on the tile image and corresponding base image. The results for one sequence are illustrated in Fig. IX.4 and IX.5, the use of an image grid had little to no effect onto the results. In fact there are two reasons why the amount of detections can differ; The bilinear interpolation during the downscaling step will yield different values beyond the first base image in $I_{\mathcal{T}}$. The

Figure IX.2.: An example tile image which contains 10 sequential images from a camera stream. Using mean-shift grouping alone was sufficient to avoid false-positive results on the image boundaries. Note the changing boundary values in the last five images

other reason is the image grid itself, as without it the amount of pre-grouping detections is increased, which in turn will influence the grouping results. The same behaviour was observed on the remaining 3 sequences. Finally I evaluated the same aspect on an interleaved sequence of all 4 camera streams (Fig. IX.6). Just as before there were no boundary detections without a pre-grouping filtering. The amount of detections differs by at most 2 detections, the amount of identical detection counts was nearly identical. Thus applying the image grid yielded no significant difference in that aspect. Although the results indicate a small difference in detection count one should see H as a different detector when using tile images. The mere values of ΔD are just an indicator for judging if the actual recognition rate will change. The results also show that one has to re-optimize the tile image detector for the specific image stream. The system was optimized in order to minimize the values of $t_{copy}, t_{prop,\uparrow}$ and $t_{prop,\downarrow}$, which were measured to be $t_{prop,\uparrow} \approx 4.767ms$, $t_{prop,\uparrow} \approx$

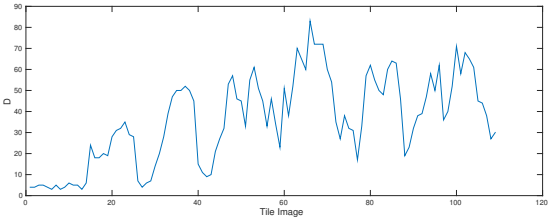


Figure IX.3.: Example of boundary detections on the tile images for a single camera stream before grouping.

$4.201ms$ and $t_{copy} = 14.3ms$ (the major part of t_{copy} was owed to the high latency of host-device data transfers). The measured values might change for larger systems (more streams or higher frame rate), yet the experiment indicate that $t_{prop,\uparrow}$ and $t_{prop,\downarrow}$ are not likely to violate the previously stated restrictions. Note that the second restriction is not violated by any parameter set from Tab. 1a or 1b.

IX.6. Boosting Results Through ROI Fusion and Non-Linear Metrics

Assuming an already trained detector, the computation steps for detection can be segmented into two sequentially executed groups G_1, G_2. Group G_1 contains all required image processing steps, i.e. the ordered sequence of bilinear image scaling; gradient computation; weighted gradient binning; histogram creation; histogram classification, which is executed multiple times. My developed method does not require any change in G_1 but in G_2, which consists of detection clustering and false positive reduc-

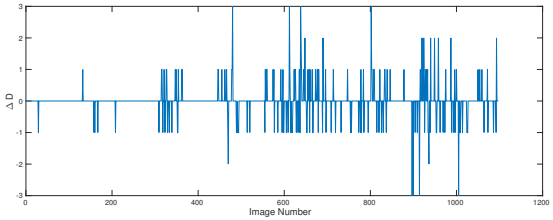


Figure IX.4.: ΔD for a single sequence without filtering boundary detections before grouping, detections on I and the corresponding tile image region differ by maximal 3 detections. 84% of all images yielded the same detection count.

tions. Furthermore I give a heuristic for optimizing the parameters before executing the sequence (G_1, G_2).
G_1 extracts pixelwise edge-gradients for an image patch (also referred to as window) $y_{s,i,j}$ (which is varied in size by scaling) and computes a corresponding histogram $h_{s,i,j}$, where s represent the current patch scale and i, j indicate the coordinates of y. Once all histograms have been extracted a linear SVM utilizes them to determine if the corresponding image patch contains an object or not, i.e. it performs a binary classification. Even in case of an SVM trained with a huge training set, this method yields a large amount of false-positive detections, e.g. misplaced detections or completely false results. This is often addressed by discarding all detections below a certain SVM score, i.e. one keeps $y_{s,i,j}$ only if $\omega_{s,i,j} < t_d$. This can reduce the ≈10000 positively classified patches down to ≈50, which usually removes many false positives yet keeps many adjacent scales and positions for correct classifications (see left image in Fig.IX.7). In order to reduce these detection groups down to an (ideally) single representant

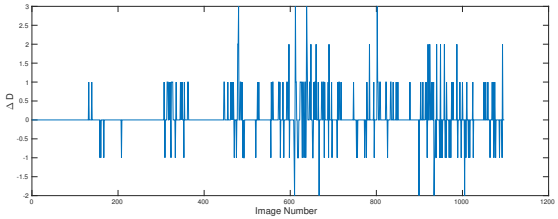


Figure IX.5.: ΔD for a single sequence with filtering boundary detections before grouping, detections on I and the corresponding tile image region differ by maximal 3 detections. 82% of all images yielded the same detection count.

one applies clustering methods such as the mean shift approach. Fig. IX.7 illustrates the detections after thresholding and a consecutive mean-shift clustering. As one can see in the left image, a few false-positives remain after the thresholded selection, these lonely Regions Of Interest (ROIs) are usually filtered out in clustering methods. Yet as the mean-shift algorithm incorporates the SVM scores, it is also possible to loose true-positives as depicted in the right image. Finetuning the threshold t_d yields only marginal improvements and results in an increased amount of detections, which in turn slows down clustering algorithms significantly. For weak classifiers one usually sets a high threshold to reduce the large amount of false-positives.

The HOG algorithm can be applied to a wide variety of objects (i.e. it is not limited to persons or body parts), this fact will become more evident in the following subsections, since they describe the construction of an upper-body detector based on abstract features.

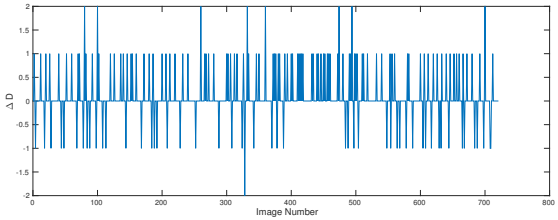


Figure IX.6.: ΔD for the first 700 images within an interleaved sequence without filtering boundary detections before grouping, detections on I and the corresponding tile image region differ by maximal 2 detections. 78% of all images yielded the same detection count.

IX.6.1. ROI Fusion

The following approach is motivated by the results of [DT05], who utilized a weighted variant of mean-shift clustering for the grouping of multiple detections in (x, y, s) space. Let D_1, D_2 be existing HOG detectors which are trained to find distinct parts of an object, e.g. an upper-body and a head detector respectively. Just as with classical mean shift algorithms, e.g. [CM02], in which one iteratively estimates the modes y_m of a distribution by

$$y_m = H_h(y_m) \sum_{i=1}^{n} \overline{\omega}_i(y_m) {H_i}^{-1} y_i \tag{IX.10}$$

with

$$\overline{\omega}_i(y_m) = \frac{|H_i|^{-1/2} \exp(-D^2[y_m, y_i, H_i]/2)}{\sum_{j=1}^{n} |H_j|^{-1/2} \exp(-D^2[y_m, y_j, H_j]/2)} \tag{IX.11}$$

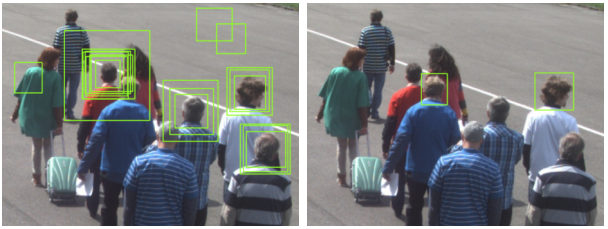


Figure IX.7.: Grouping of detections: The left image illustrates the detections after thresholding the SVM results, the right image shows the detections after application of mean-shift clustering.

Let $y_i = (x, y, s) \in \mathbb{R}^3$ be the elements of the sampled data (i.e. the windows obtained by a complete multiscale HOG run, x, y, y denoting the position of the window center and scale respectively), $H_i = diag(\sigma_x, \sigma_y, \sigma_s)$ the diagonal uncertainty matrix and

$$
\begin{aligned}
D^2[y_m, y_j, H_j] &:= (y_m - y_i)^T {H_i}^{-1} (y_m - y_i) && \text{(IX.12)} \\
&= \sigma_x((y_m)_1 - (y_i)_1)^2 + \sigma_y((y_m)_2 - (y_i)_2)^2 && \text{(IX.13)} \\
&\quad \sigma_s((y_m)_3 - (y_i)_3)^2 && \text{(IX.14)}
\end{aligned}
$$

the Mahalanabois distance between y_m and y_i ($(y)_i$ indicates the i-th vector element). I propose the following weighted extension, let y^1, y^2 denote the resulting windows from D_1, D_2 respectively and ω^1, ω^2 the corresponding SVM scores / weights. First one has to create feasible 5-dimensional features

$$
\tilde{y}_k := ((y_i^1)_1, (y_i^1)_2, (y_j^2)_1, (y_j^2)_2, (y_i^1)_3), \quad \tilde{\omega}_k := \omega_i^1 \omega_j^2 \qquad \text{(IX.15)}
$$

by grouping all feasible D_2 windows y_j^2 for a single D_1 window y_i^1. The selection criteria for this combination, which must be

fulfilled, are as follows

1. $\alpha_1(y_i^1)_3 \leq (y_j^2)_3 \leq \alpha_2(y_i^1)_3, \quad \alpha_1, \alpha_2 \in (0,1], \alpha_1 < \alpha_2$
2. Let w, h denote the width and height of y^1:
   $\beta_1 w \leq (y_j^2)_1 - (y_i^1)_1 \leq \beta_2 w, \quad \beta_1 < \beta_2, \ \beta_1, \beta_2 \in (0,1]$
3. $\beta_3 h \leq (y_j^2)_2 - (y_i^1)_2 \leq \beta_4 h, \quad \beta_3 < \beta_4, \ \beta_3, \beta_4 \in (0,1]$

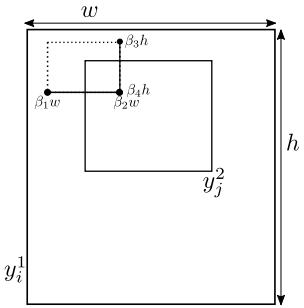


Figure IX.8.: The fusion of detections, each detection y_i^1 of detector D^1 is combined with all detections y_j^2 from detector D^2 if its left upper corner lies in the dashed rectangle. The same applies for the scale.

These rules represent position restrictions which combine windows y_j^2 only if they lie in a certain boundary relative to y_i^1 (see Fig. IX.8). A practical example would be to consider only head windows which lie completely within the upper-body window. This grouping lifts the upper-body windows into a 5-dimensional space and adds the position variance of each fitting head window y_i^2 to it. The scale remains unchanged since a scale equivalent is

defined with criteria 1. The mean-shift clustering was changed to a weighted variant

$$\overline{\omega}_i(y_m) = \frac{|H_i|^{-1/2}\tilde{\omega}_i \exp(-D^2[y_m, y_i, H_i]/2)}{\sum_{j=1}^{n} |H_j|^{-1/2}\tilde{\omega}_j \exp(-D^2[y_m, y_j, H_j]/2)} \quad \text{(IX.16)}$$

with $H_i = diag(\sigma_x^1, \sigma_y^1, \sigma_x^2, \sigma_y^2, \sigma_s)$. The uncertainty values σ_x^2, σ_y^2 should be set to a smaller value than σ_x^1, σ_y^1, since might be reasonable to put more certainty into D_2 so that it might stabilize the detection windows for the upper body. Since the amount of D_1 detections is increased I refer to this combination approach as sample spreading. the evaluation in section IX.9 shows that this strategy can yield a significant improvement in detection quality compared to D_1 alone. It should be pointed out that k can (in theory) reach values up to $|\{y_i^1\}| \cdot |\{y_j^2\}|$, which can even slow down an efficient implementation.

IX.6.2. A Nonlinear Metric for SVM weights

In order to reduce the computation time and increase the detection quality in the context of HOG applications, one usually first filters out all result windows with a SVM weight below a given threshold t_1. This strategy can be applied to accommodate the problem of too many items for the mean shift clustering. Yet this simple method can also remove a large amount of correct detections. The reason for this lies in the HOG algorithm, for large objects it will scale the corresponding image area down to the detection window size, this removes a large quantity of high-res image information. Such windows will exhibit a smaller SVM weight compared to smaller regions. Filtering according to t_1, which obviously will be chosen according to the higher SVM values (and thus the smaller windows), will remove many if not all

candidates for large objects. My approach addresses this problem under the assumption that not all large windows have been filtered out. I developed a simple strategy by rescaling the SVM weights (after filtering with t_1) according to

$$\omega'(\omega_i) := f(\omega_i)\omega_i \tag{IX.17}$$

with for example

$$f(\omega_i) := \begin{cases} \tau(-\beta((I_h - (y_i)_2)/I_h))(y_i)_3 & (y_i)_3 \geq \rho \\ 1 & else \end{cases} \tag{IX.18}$$

with I_h being the image height. The general effect of this transformation should be that SVM weights of large windows will be increased to rival with those of smaller windows during the mode estimation. Not only can this approach retain large windows but also remove infeasible small windows in an area of large windows. The scaling function f must be chosen to accommodate this goal. In the example above the scaling function f exhibits a linear behaviour, yet one might also use an nonlinear tesselated transformation for more complex scenes. Large objects usually appear in the lower part of the image while small objects inhibit the upper portion. Thus SVM weights of window candidates in the lower region should be scaled up, while the scaling vanishes linear in the upper image area. Furthermore only the weights of windows above a certain scale will be transformed, this prevents small windows in the lower area to be transformed as well. An example for this can be seen in Fig.IX.9.

Figure IX.9.: Weighting of near field windows: The left image shows the use of a single upper-body detector; the person in the lower part is not detected due to small amounts of candidate windows. Using transformed SVM weights one can see on the right image that the same detector now finds the person in the lower part and suppresses the small false detection.

IX.7. A Detection-Pipeline for Boosting the Detection Quality

In order to further motivate the techniques from section IX.6 I conducted a thorough evaluation by embedding them in a detection pipeline. This section shows that an efficient implementation of the previously described algorithms and ideas can boost HOG-based systems in terms of their detection quality while still maintaining previous time constraints.
The detection pipeline is depicted in Fig. IX.10. The first step may consist of any form of image preprocessing, the output is directed into the HOG detector, which performs the initial detection (of at least one feature) without any form of detection

grouping. All detections are forwarded into a metric-based selector, which consists of two steps; a thresholded reduction of detections and a metric scaling of SVM weights. After this point the results are forwarded into two parallel grouping branches, each consisting of two steps; detection grouping and a sanity check, which may use any available scene information in order to remove detections with impossible positions. The calculated detections from both branches are finally fused with a mean-shift grouping, these detections are forwarded into a so called *streaking* block. The streaking block is utilized in video streams and applies a simple heuristic (which is described in Alg. 38) in order to predict detections and eliminate short term detection gaps. The algorithm leaves out the details of how to find corresponding detections, i.e. it does not specify the form of feature vectors. In the following evaluation normalized intensity histograms have been used, additionally the distance between detections has been checked. More precisely

- The feature vector f_i consists of $n \cdot 256$ real numbers, 256 for each color channel.
- Feature vectors are compared by calculating the euclidean distance, which can not exceed a threshold t_f
- The distance between the upper left corner for two detections can not exceed a pixels along the x-axis and b pixels along the vertical.

Although simple in its design, this approach yields significant improvements compared to the canonical HOG algorithm. Yet, depending on the situation, e.g. image quality, a color histogram might become unfeasible since it is susceptible to image noise effects. This histogram based metric can be exchanged for more

complex descriptors and similarity measures, e.g. a gradient based descriptor with a weight based metric, thus one can adapt the described pipeline for such a scenario. The streaking results will be grouped by a last mean-shift step after having been filtered by a final sanity check.
Note that the classic HOG algorithm can be obtained by setting less restrictive parameters for the metric selection, defining the sanity check of the fusion branch to filter all results, setting the streaking history size $s = 1$, adapting the parameters of the last two grouping steps and the last sanity check.

Algorithm 38 Streaking

Input: Image I, detections $\mathcal{D}$, history size s, detection history $\mathcal{H} = \{(d_1, f_1, x_1, c_1), ..., (d_k, f_k, x_k, c_k)\}$ with shift buffer x_i of size s, feature vector f_i, match counter c_i and mismatch threshold t

```
b = 0;
for each d ∈ D do
    for each e_i ∈ H do
        compare the feature vector e_i and that of d (include additional sanity checks)
        if If the vectors match then
            Add position x of d to x_i;
            Update histogram of f_i with data at the position of d;
            Set c_i = 0;
            b = 1;
            break;
        else
            Set c_i = c_i + 1;
        end if
        if If c_i ≥ t then
            remove e_i from H;
        end if
    end for
    if b == 0 then
        Add new entry for d to H;
    end if
    b = 0;
end for
```

IX.8. Evaluation Setup

The motivation behind the design of the pipeline was to demonstrate the applicability of the developed algorithmic concepts.

Since the work in this thesis mainly targets the improvement of efficiency while preserving scalability I will show my works potential by improving the detection rate of an existing HOG implementation while preserving the previous time constraints. Two HOG detector were trained on the INRIA training set, one for the detection of entire human bodies (H_B) and one for detecting upper bodies (H_{UB}), the training was done according to the original protocol by [DT05]. Each corresponding SVM was obtained by a decadic grid search over $C \in [10^{-5}, 10^3]$ with a 10-fold cross-validation at every step. Since the INRIA database only provides labels for complete bodies, all upper body labels were extracted by using the upper third of each rectangular label ROI. Both HOG detectors have been utilized in the pipeline's detection block. Table IX.3 states the HOG parameters in more detail.
The pipeline was evaluated on the CAVIAR [CAV] database since it provides targets for all blocks in the pipeline;

1. The metric scaling of SVM weights becomes applicable due to strong size differences between objects in near and far field.

2. The ROI fusion is trivially applicable.

3. The streaking is applicable since the images are extracted from a continuous video stream.

Futhermore, this database represents a typical field for many parallel video streams; surveillance. In order to get comparable results between the classic standalone HOG detectors and the pipeline results, the mean-shift grouping parameters were kept identical for all grouping steps, $\sigma_x = 16, \sigma_y = 8, \sigma_s = 1.05, \epsilon = 1.0$.

The fusion grouping was done with $\alpha_1 = \alpha_2 = 0.05, \beta_1 = \beta_2 = 0.1, \beta_3 = \beta_4 = 0.1$ and $\sigma_x^1 = \sigma_x^2 = \sigma_x, \sigma_y^1 = \sigma_y^2 = \sigma_y$. The iteration count was limited to a maximum of 100.
A detection d is considered to be a true positive if it can be associated with a ground truth date d_{gt} such that

$$\frac{|d \cap d_{gt}|}{|d \cup d_{gt}|} \geq 0.7 \tag{IX.19}$$

The parameters for the metric scaling were set to $\tau = \beta = 1.0$ and $\rho_{Body} = -0.2, \rho_{UpperBody} = -0.1$, this represents an entirely linear scaling over the complete image. One should note that ρ represents an individual thresholding for each detector. All sanity checks consist of checking if a detection lies in an unfeasible region, which are defined through a polygon set $P = \{p_1, ..., p_n\}$. Since all images within the CAVIAR dataset have been recorded with a rather low resolution of $640x480$ the preprocessing consists of a GPU accelerated upscaling to $1600x1200$. This step is reasonable in the sense of applicability, the already trained detector should not need to be retrained for each different image format. Furthermore one would need to shrink the training images even more for smaller window sizes, this would introduce further information loss and a reduction in detection quality.
Two image sets of a specific scene were chosen, *WalkByShop1Cor* and *ThreePastShop1Cor*, the scene and the corresponding set P is visualized in Fig. IX.11. Besides the parameters for the metric weight scaling and the defined polygons no additional scene specific optimizations have been utilized.

IX.9. Results

The results on both image sets are depicted in Fig. IX.12 and IX.13, respectively. The plots in Fig. IX.12 illustrate the differences between the classic HOG algorithm and elements of the pipeline. Let $TP_i = (x_0, x_1, ..., x_k, ...)$ be the sequence of image-wise true positive counts obtained with algorithm i for each image, analogously let FP_i be the sequence of false positives. Both plots in the first row of Fig. IX.12 depict the difference $\delta^{TP}_{HOG,Metric} := TP_{Metric} - TP_{HOG}$ and $\delta^{FP}_{HOG,Metric} := FP_{Metric} - FP_{HOG}$, respectively. If $\delta^{TP}_{HOG,Metric}(k) > 0$ for some image index k then more true positives were obtained by using the HOG algorithm than with metric scaling of the weights. The same holds for the false positive count. It becomes obvious that less true positives were obtained with metric scaling, yet one has to accept significantly more false positives. This indicates a stabilizing effect onto the canonical HOG approach, which is also visible in the second row of plots, i.e. the results of comparing the HOG against the pipeline's fusion branch. The last row in Fig. IX.12 depicts the comparison between the HOG and the entire pipeline. The amount of false positives is significantly reduced while keeping the amount of true positives close to that of the classic HOG algorithm. One obtains a mean value of 4.3643 less false positives per image and 1.02 less true negatives per image.

The same effect was observed on the second image set, i.e. *WalkByShop1Cor*. Each row in Fig. IX.13 shows results similar to the previous ones, yet, the evaluation yielded a mean value of 4.3408 less false positives per image while decreasing the average amount of true positives by 0.1796 per image.

These results illustrate the potential of the constructed pipeline,

by using the detections of an existing HOG detector one obtain significantly less false positives while preserving the amount of true positives. As Fig. IX.14 shows, it turned out that a significant amount of fused detections was constructed within the fusion branch. This in turn posed a bottleneck for the pipeline's applicability, since grouping times would reach values up $\approx$ 200s (right plots in Fig. IX.14). Yet, by using the concept for massively parallelized mean shift computation (see chapter VII) this time could be reduced to a maximum of $\approx$ 10ms, which in turn led to the pipelines complete processing times as depicted in Fig. IX.15. The pipelines maximal detection time was $\approx$ 88ms, which still enables one to process $\approx$ 11 frames per second. An additional speed-up could be achieved by using the tile image approach from section IX.3 since the actual SVM-based detection process makes up about 1/3 of the total processing time (see the red line in Fig IX.15).
One has to note that the detection quality is determined to a large extend by the initial HOG detector. An improvement of the underlying detections would further boost the pipelines results, such an improvement may include scene specific SVM training or boosting approaches from machine learning.

Table IX.2.: Processing times for differently sized tile images. The numbers in column T represent the stack size in units of images (each of size 640x480), t_{single} the time when single images would be used and $\Delta E_{\mathcal{T}}(H)$ the efficiency increase in relation to the last table row.

T	$t_{process,z_1}$	t_{single}	$E_{\mathcal{T}}(H)$	$\Delta E_{\mathcal{T}}(H)$	rel. speed-up
1	0,0116 s s	0,0116 s	0,0000 s	-	-
2	0,0147 s	0,0233 s	0,0086 s	-	58,86 %
3	0,0190 s	0,0349 s	0,0160 s	85.25 %	84,35 %
4	0,0227 s	0,0466 s	0,0239 s	49.44 %	105,27 %
5	0,0266 s	0,0582 s	0,0316 s	32.34 %	118,80 %
6	0,0309 s	0,0699 s	0,0390 s	23.29 %	126,16 %
7	0,0337 s	0,0815 s	0,0478 s	22.73 %	142,04 %
8	0,0375 s	0,0932 s	0,0557 s	16.36 %	148,45 %
9	0,0407 s	0,1048 s	0,0641 s	15.14 %	157,41 %
10	0,0447 s	0,1165 s	0,0718 s	12.02 %	160,77 %
11	0,0485 s	0,1281 s	0,0796 s	10.88 %	164,19 %
12	0,0522 s	0,1398 s	0,0876 s	9.96 %	167,71 %
13	0,0554 s	0,1514 s	0,0960 s	9.69 %	173,44 %
14	0,0587 s	0,1631 s	0,1044 s	8.67 %	177,82 %
15	0,0621 s	0,1747 s	0,1126 s	7.93 %	181,50 %
16	0,0658 s	0,1864 s	0,1206 s	7.02 %	183,18 %
17	0,0692 s	0,1980 s	0,1288 s	6.82 %	186,00 %
18	0,0732 s	0,2097 s	0,1365 s	5.96 %	186,39 %
19	0,0767 s	0,2213 s	0,1446 s	5.95 %	188,42 %
20	0,0809 s	0,2330 s	0,1520 s	5.17 %	187,93 %

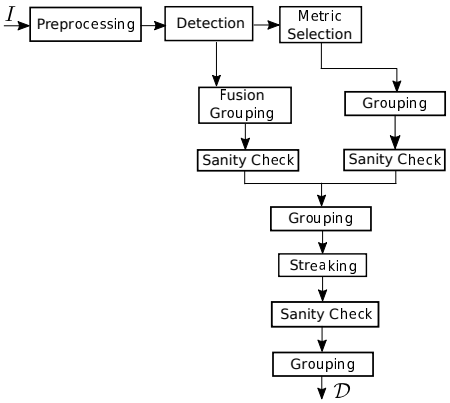
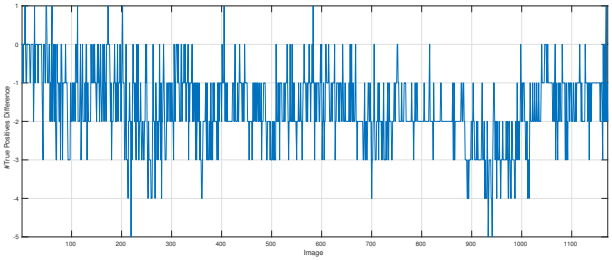


Figure IX.10.: A detection pipeline with ROI fusion and metric scaling of SVM weights. The image I will be preprocessed before being forwarded into the HOG-based detector without grouping. The third block removes all detections below a certain threshold and rescales the weights according to a scene specific metric. The initial and remaining detections are sent into two parallel grouping branches; a fusion grouping and a canonical grouping, respectively. The grouped results of both branches will be merged afterwards. In case of video streams from a static scene the streaking block can be utilized for reduction of detection gaps. All sanity checks are scene specific heuristics which remove detections at unfeasible places. The pipelines output consists of a grouped detection set $\mathcal{D}$.

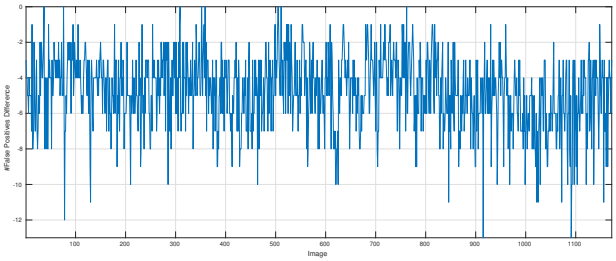
Figure IX.11.: A scene from the CAVIAR image database, the surveillance camera's position induces a significant size difference between objects in near and far field. Three polygons (P_1, P_2, P_3) have been defined and enclose image regions which should not contain pedestrians.

Table IX.3.: Parameters for both detectors, i.e. body and upper body HOG detectors

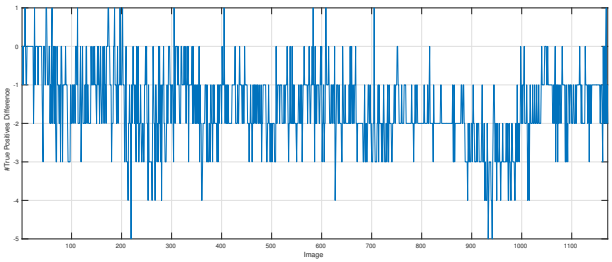
Parameter	H_B	H_{UB}
Cell size (pixel)	8×8	8×8
Block size (pixel)	16×16	16×16
Window size (pixel)	64×128	96×88
Block stride (pixel)	$(8, 8)$	$(8, 8)$
Window stride (pixel)	$(8, 8)$	$(8, 8)$
Bin count	9	9
Scale factor	1.05	1.05
σ	1.0	1.0



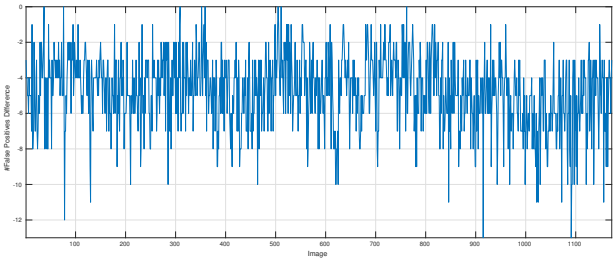
(a) HOG vs metric true positive count



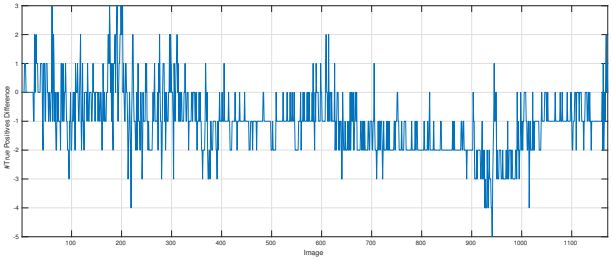
(b) HOG vs metric false positive count[6pt]



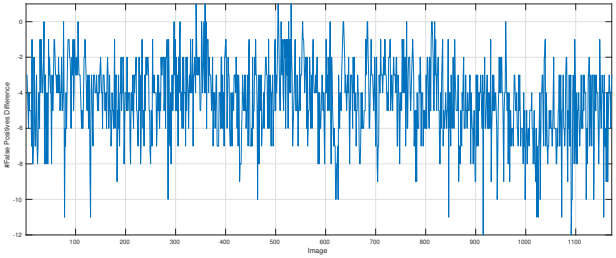
(c) HOG vs fusion true positive count



(d) HOG vs fusion false positive count[6pt]

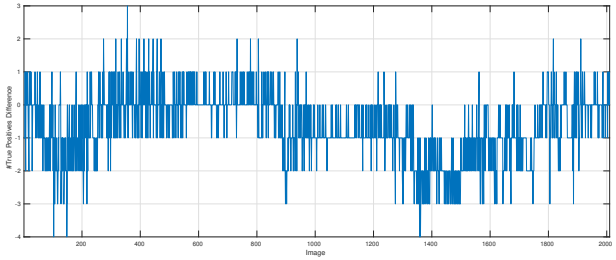


(e) HOG vs streak true positive count

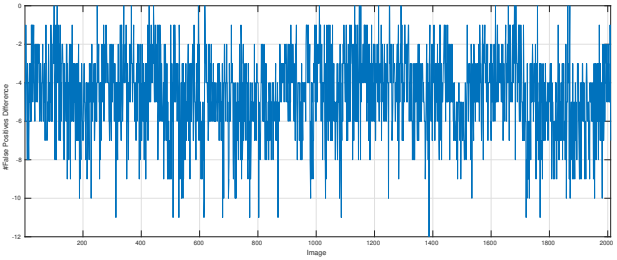


(f) HOG vs streak false positive count[6pt]

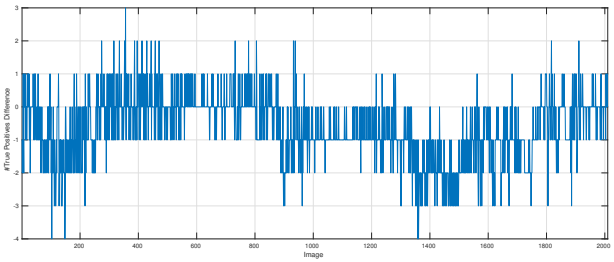
Figure IX.12.: Comparison of recall statistics for the canonical HOG body detector and pipeline segments on the *ThreePastShop1Cor* image set. Image a) shows the difference: #true positives fusion branch - #true positives classic HOG, image b) the corresponding false positive difference, images c) and d) express the same statistics for the complete pipeline and the classic HOG.



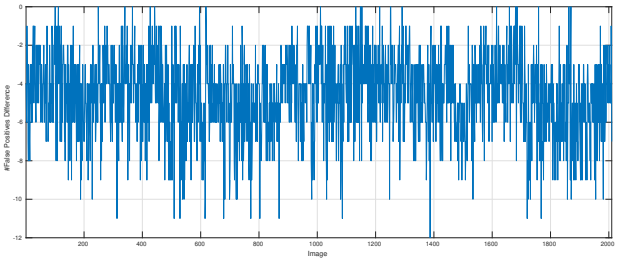
(a) HOG vs metric true positive count



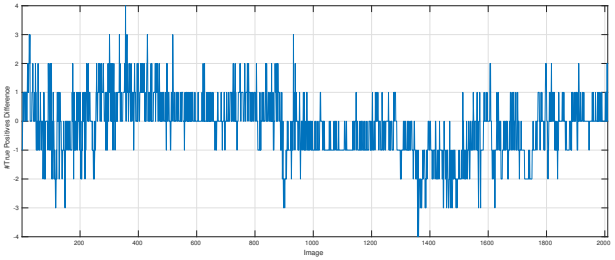
(b) HOG vs metric false positive count



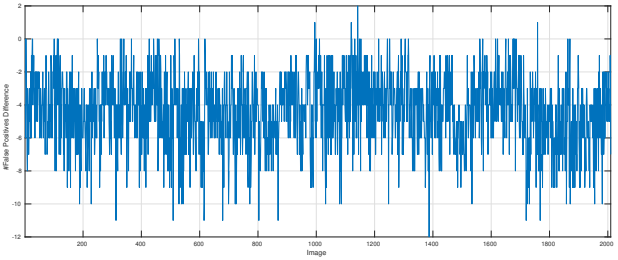
(c) HOG vs fusion true positive count



(d) HOG vs fusion false positive count

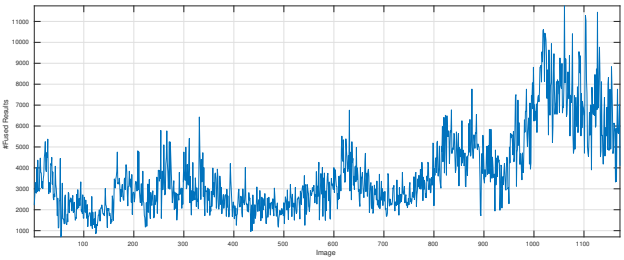


(e) HOG vs streak true positive count

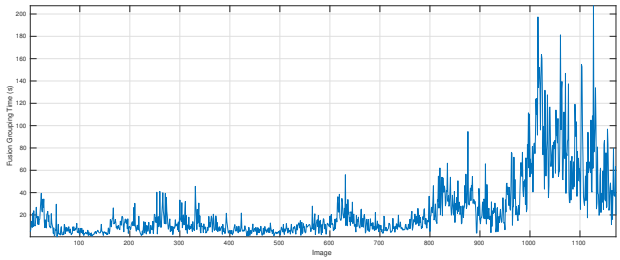


(f) HOG vs streak false positive count[6pt]

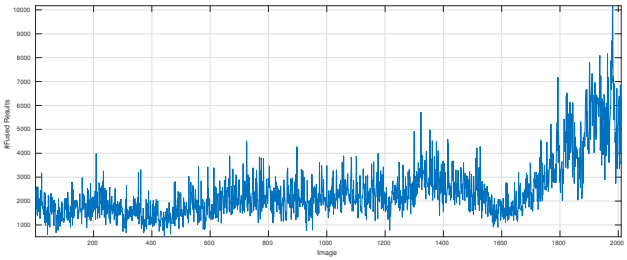
Figure IX.13.: Comparison of recall statistics for the canonical HOG body detector and pipeline segments on the *WalkByShop1Cor* image set. Image a) shows the difference: #true positives fusion branch - #true positives classic HOG, image b) the corresponding false positive difference, images c) and d) express the same statistics for the complete pipeline and the classic HOG.



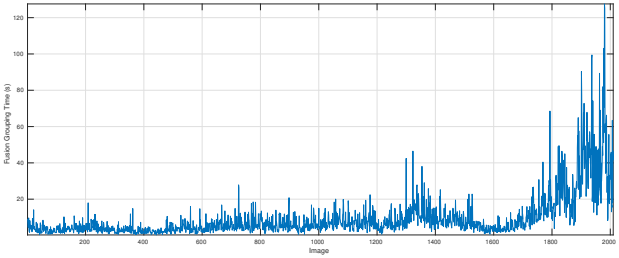
(a) Amount of computed multidim detections in the fusion branch (*ThreePastShop1Cor*)



(b) Grouping time in the fusion branch (*ThreePastShop1Cor*)



(c) Amount of computed multidim detections in the fusion branch (*WalkByShop1Cor*)



(d) Grouping time in the fusion branch (*WalkByShop1Cor*)[6pt]

Figure IX.14.: Statistics for the multidimensional mean-shift grouping on the *ThreePastShop1Cor* and *WalkByShop1Cor* image set. Image a) depicts the amount computed detection combinations for the multidimensional mean-shift grouping on the *ThreePastShop1Cor* image set. The corresponding grouping times are shown in image b). Images c) and d) visualize the same attributes on the *WalkByShop1Cor* image set.

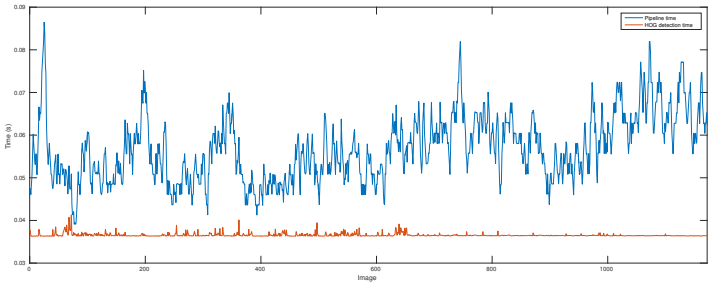


Figure IX.15.: Complete processing time (blue line) of the pipeline for the *ThreePastShop1Cor* image set. The red line depicts the detection time, note that these time values are invariant to the amount of detections since the grouping has been left out.

X. Conclusion and Revisiting the Research Questions

"What we usually consider as impossible are simply engineering problems... there's no law of physics preventing them." (Michio Kaku)

The following chapter summarizes conclusions from each chapter and discusses them from different points of view. As stated in chapter II, despite the advent of modern methods as e.g. deep learning, the classic HOG detection algorithm is still being applied in various state-of-the-art systems. Yet, several algorithmic aspects of GPU accelerated HOG variants indicated potential for computational acceleration. Thus, in this thesis I addressed three research questions

1. How can the HOG be improved through parallelization?
2. Can the developed methods be generalized so that they can be applied in different application fields?
3. Can the generalization be used in order to boost existing HOG-based systems?

Most GPU accelerated HOG variants mainly address the descriptor computation while using heuristics for non-maximum suppression. In chapter VII I addressed this last step directly by

formally describing a system independent massively parallelized version. From this generic algorithm I developed a GPU specific variant, which led to addressing the problem of GPU under- and overutilization with respect to the data size. In order to prevent such phenomena the concept of vPPUs was created, a meta layer for existing SIMD architectures. Through evaluations I showed its practical gain in the context of parallelized mean shift computation. While the common naive parallelization approach yielded speed-ups of up to ≈ 17 compared to CPU computation times, my approach increased this to ≈ 176.
With the concept of vPPUs in mind I generalized it to arbitrary computation system, giving rise to the concept of hypersystems. Chapter VIII motivates the use of vPPUs with another application example; training of small scale neural networks on massively parallel architectures. I showed that my approach can yield a speed-up of up to 2.37 compared to naive approaches. Since the computational challenge of training neural networks mainly consists of efficient matrix multiplication, I continued with an analysis of the classic matrix multiplication problem in section VIII.10. In this section I discussed the workload deployment in Beowulf cluster systems with a special focus on multi-GPU support. My analysis showed several things;

- The efficiency of matrix multiplication can be significantly increased for small scale problems by considering intrinsic elements of device and software, e.g. operating system, device driver.
- Deploying small workload fragments in an Ethernet based Beowulf cluster will most likely be unfeasible compared to the use of a single GPU-based computation node.
- Calculating an optimal deployment schedule in the context

of hypersystems is NP-complete.

- Exploiting configuration symmetries in a Beowulf cluster provides a heuristic for an efficient schedule computation. Yet, it still requires system measurements, which in turn may require much time.
- Subsampling a large configuration space for measurements is feasible if the gaps are interpolated adequately. This was demonstrated in section VIII.16.1 by using neural networks for interpolation.

Overall it becomes evident that hypersystems provide an optimization interface for system specific parameters without actually identifying them. Using hypersystems one determines the system behaviour in the context of the applied algorithms, in other words, the goal when using hypersystems is to increase the systems performance by changing the algorithm's parameters, not those of the underlying computation system. The extend to how far the performance can be increased is mainly determined by how much the vPPU concept can be embedded into the algorithm's structure. This was shown in chapters VII and VIII, during the construction of Alg. 20 in section VII.2 the load factor became a crucial tuning parameter. This was mainly due to the interplay between the hardware's operating mode (SIMT) and the vPPU concept. It was possible to implicitly optimize the deployment strategy by changing the load factor, i.e. the load factor became capable of controlling the scheduling overhead with respect to the total amount of scheduled threads. Yet, as shown in chapter VII, despite using the vPPU concept one has to study the underlying system in order to create algorithm parameters which may induce a performance gain during execution. Examples for this are f_{vppu}, τ, n in Alg. 20. The application examples,

training of small scale neural networks and matrix multiplication, also demonstrate that hypersystems can be used in order to improve the utilization of large parallel systems for "small" computation problems while still outperforming sequential or naive solutions. With the rise of many-core and GPU equipped mobile devices, e.g. cell phones, the need for efficient (and thus power saving) algorithms becomes a crucial element in application development. It is an interesting research question about how much gain the use of hypersystems can provide for redesigning existing algorithms for mobile devices.

At this point one should be reminded of organic computing's goal, which is to autonomously create such algorithms and determine optimal parameter values. I hope that, as long as this goal has not been reached, researchers and engineers will benefit at least to a small extent from my developed concepts. It also raises two interesting research question; which algorithms can be embedded most efficiently into the hypersystem concept and can this construction be automatized. Maybe one can even draw inspiration from hypersystems in order to approach challenges in organic computing.

In order to perform all these evaluations I developed a framework *SimpleHydra* which addresses several shortcomings of existing solutions. The most important features are its generic structure and fast deployment in heterogeneous system architectures, allowing it to embed already existing solution, e.g. MPI based programs. In chapter VI I introduced and discussed the algorithmic concepts in much detail, putting a special focus on the underlying software for network communication.

With these results and knowledge I revisited the HOG algorithm. Chapter IX introduced the idea of using tile-images, which in turn are motivated by the same idea as vPPUs and thus, hy-

persystems. I aimed to increase the GPU utilization without redesigning the existing algorithm, as shown in section IX.5 an increase by a factor up to 2.88 can be achieved. Furthermore I introduced a concept of deploying the detection in a heterogeneous cluster system, I identified the parameters for real time boundaries and showed that an Ethernet based system can easily fulfill them. These results have a large significance regarding existing HOG-based systems, since most hardware setups utilize an Ethernet based network topology. Furthermore, my approach requires no redesign of the detection algorithm, i.e. the actual implementation remains unchanged. Regarding future systems one can also draw applicability conclusions from my results, e.g. when more efficient network interfaces or even different detection algorithms are applied. Utilizing the thought of hypersystems and the motivation from section IX.2.3 one can examine different parallelizable algorithms for object detection and derive a more efficient form.
A very common application for object detection is that of video surveillance, in which a system must process many parallel video streams. Section IX.7 finally showed how an existing HOG based system can be augmented for this application; using the pipeline from IX it becomes possible to increase the systems reliability whereas the results from chapter IX.5 provide the means to increase its real time capabilities. The pipeline incorporates two developed concepts; metric scaling of SVM weights and ROI fusion, both being introduced in section IX.6.2 and IX.6.1, respectively. The results indicate a stabilizing effect onto the initial HOG detector, i.e. the amount of false positive detections can be reduced while retaining the amount of true positives. As shown in section IX.9 one can expect, considering an adequate choice of parameters, a reduction of false positives by a factor of

≈ 4 while reducing the amount of true positives only marginally with a factor of ≈ 0.2. Increasing the detectors initial reliability will most likely eliminate the reduction of false positives, yet this remains a question for future research.
Taking all results into account, one can not only efficiently boost an existing HOG detector's reliability through a pipeline, one also increase the detector's performance by utilizing the tile image approach in a distributed manner.

Bibliography

[12693] Mpi: A message passing interface. In *Supercomputing '93. Proceedings*, pages 878–883, Nov 1993.

[AB07] Masood Ahmed and Shahid Bokhari. Mapping with space filling surfaces. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1258–1269, 2007.

[ABT04] Ali Ahmadinia, Christophe Bobda, and Jürgen Teich. A dynamic scheduling and placement algorithm for reconfigurable hardware. In *Organic and Pervasive Computing–ARCS 2004*, pages 125–139. Springer, 2004.

[AL11] Mir Ashfaque Ali and SA Ladhake. Overview of space-filling curves and their applications in scheduling. *International Journal of Advances in Engineering and Technology*, 1:148–154, 2011.

[AMD13] AMD. Amd accelerated parallel processing opencl programming guide. 2013.

[AMO04] Péter Arató, Zoltán Ádám Mann, and András Orbán. Component-based hardware-software codesign. In *Organic and Pervasive Computing–ARCS 2004*, pages 169–183. Springer, 2004.

[AV02] Joel Adams and David Vos. Small-college supercomputing: Building a beowulf cluster at a comprehensive college. *SIGCSE Bull.*, 34(1):411–415, February 2002.

[Avi05] Shai Avidan. Ensemble tracking. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2:494–501, 2005.

[AWW12] Shoaib Azmat, Linda Wills, and Scott Wills. Accelerating adaptive background modeling on low-power integrated gpus. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 568–573. IEEE, 2012.

[Bai15] Todd M Bailey. Convergence of rprop and variants. *Neurocomputing*, 159:90–95, 2015.

[BC14] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, pages 2654–2662, 2014.

[BCD+12] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: A verifier for gpu kernels. *SIGPLAN Not.*, 47(10):113–132, October 2012.

[BD13] Ethel Bardsley and Alastair F Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of gpu kernels. 2013.

[BKS04] Jens Braunes, Steffen Köhler, and Rainer G Spallek. Recast: An evaluation framework for coarse-grain reconfigurable architectures. In

Organic and Pervasive Computing–ARCS 2004, pages 156–166. Springer, 2004.

[BN02] Azzedine Boukerche and Mirela Sechi M. Annoni Notare. Behavior-based intrusion detection in mobile phone systems. *Journal of Parallel and Distributed Computing*, 62(9):1476 – 1490, 2002.

[BOHS14] Rodrigo Benenson, Mohamed Omran, Jan Hosang, and Bernt Schiele. Ten years of pedestrian detection, what have we learned? In *Computer Vision-ECCV 2014 Workshops*, pages 613–627. Springer, 2014.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[Buc02] R. Buchty. *Cryptonite – A Programmable Crypto Processor Architecture for High-Bandwidth Applications*. PhD thesis, Sep 2002.

[CAV] Caviar: Context aware vision using image-based active recognition. `http://homepages.inf.ed.ac.uk/rbf/CAVIAR/`. Accessed: 2016-03-01.

[CK01] L. Cherkasova and M. Karlsson. Scalable web server cluster design with workload-aware request distribution strategy ward. In *Advanced Issues of E-Commerce and Web-Based Information Systems, WECWIS 2001, Third International Workshop on.*, pages 212–221, 2001.

[CKP+93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993*, pages 1–12, 1993.

[CM02] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, May 2002.

[Com03] D. Comaniciu. An algorithm for data-driven bandwidth selection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(2):281–288, Feb 2003.

[Cou09] HPC Advisory Council. Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing. Technical report, HPC Advisory Council, 2009.

[CPJK04] Tae-Sun Chung, Stein Park, Myung-Jin Jung, and Bumsoo Kim. STAFF: state transition applied fast flash translation layer. In *Organic and Pervasive Computing - ARCS 2004, International Conference on Architecture of Computing Systems, Augsburg, Germany, March 23-26, 2004, Proceedings*, pages 199–212, 2004.

[CWL+14] Xie Chen, Yongqiang Wang, Xunying Liu, Mark JF Gales, and Philip C Woodland. Efficient gpu-based training of recurrent neural network language models using spliced sentence bunch. *submitted to Proc. ISCA Interspeech*, 2014.

[DBPDP+06] C. Di Biagio, G. Pennella, E. De Paoli, R. Grandi, and F. Giammarino. Pvm advanced load balancing in industrial environment. In *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, pages 5 pp.–, Feb 2006.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[DHL+03] John Dubinski, RJ Humble, Chris Loken, Ue-Li Pen, and PG Martin. Mckenzie: A teraflops linux beowulf cluster for computational astrophysics. In *Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications*, 2003.

[DNGFV00] C. Di Napoli, M. Giordano, M.M. Furnari, and F. Vitobello. Pvm application-level tuning over atm. In *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, pages 391–397, 2000.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[EG75] P Erds and R.L Graham. On packing squares with equal squares. *Journal of Combinatorial Theory, Series A*, 19(1):119 – 123, 1975.

[Ein01] Albert Einstein. *The World As I See It*. New World, 2001.

[EMA97] S.W. Ellacott, J.C. Mason, and I.J. Anderson. *Mathematics of Neural Networks: Models, Algorithms and Applications*. Operations Research/-Computer Science Interfaces Series. Springer, 1997.

[FH75] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *Information Theory, IEEE Transactions on*, 21(1):32–40, Jan 1975.

[FIS36] R. A. FISHER. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[F.R01] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 6*, 2(11):559–572, 1901.

[FYY+15] H. Fukui, T. Yamashita, Y. Yamauchi, H. Fujiyoshi, and H. Murase. Pedestrian detection based on deep convolutional neural network with ensemble inference network. In *Intelligent Vehicles Symposium (IV), 2015 IEEE*, pages 223–228, June 2015.

[Gal14] François Le Gall. Powers of tensors and fast matrix multiplication. *CoRR*, abs/1401.7714, 2014.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GBSP04] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *In Proceedings of 6th Annual Linux Symposium*, 2004.

[GDMO02] Jeffrey D. Grant, RL Dunbrack, Frank J Manion, and Michael F Ochs. Beoblast: distributed blast and psi-blast on a beowulf cluster. *Bioinformatics*, 18(5):765–766, 2002.

[Gib89] Phillip B Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.

[GM02] Murray Gell-Mann. What is complexity? In *Complexity and industrial clusters*, pages 13–24. Springer, 2002.

[GR14] MB Giles and I Reguly. Trends in high-performance computing for engineering calculations. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2022):20130319, 2014.

[GS15] Omer Gold and Micha Sharir. Improved bounds for 3sum, k-sum, and linear degeneracy. *CoRR*, abs/1512.05279, 2015.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[HH10] M. Haselman and S. Hauck. The future of integrated circuits: A survey of nanoelectronics. *Proceedings of the IEEE*, 98(1):11–38, Jan 2010.

[HI15] Craig Henderson and Ebroul Izquierdo. Reflection invariance: an important consideration of image orientation. *CoRR*, abs/1506.02432, 2015.

[HKE+13] Miki Hirabayashi, Shigeo Kato, Masato Edahiro, Kenji Takeda, Taiki Kawano, and Seiichi Mita. Gpu implementations of object detection using hog features and deformable models. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on*, pages 106–111. IEEE, 2013.

[HLS10] Bing Han, Jimmy Leblet, and Gwendal Simon. Hard multidimensional multiple choice knapsack problems, an empirical study. *Computers & operations research*, 37(1):172–181, 2010.

[HM03] W. Hoffman and K. Martin. The cmake build manager. 28(1):40–47, 01 2003.

[HMGH14] S. Hommel, D. Malysiak, M. Grimm, and U. Handmann. Apfel - fast multi camera people tracking at airports, based on decentralized video indexing. *Science2 - Safety and Security*, 2014.

[HMH13] S. Hommel, D. Malysiak, and U. Handmann. Model of human clothes based on saliency maps. In *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, pages 551–556, Nov 2013.

[Jin01] Hai Jin. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001.

[Kai96] R.Y. Kain. *Advanced Computer Architecture: A Systems Design Approach*. Prentice Hall, 1996.

[KH13] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[KMGH14] Thomas Kopinski, Darius Malysiak, Alexander Gepperth, and Uwe Handmann. Time-of-flight based multi-sensor fusion strategies for hand gesture recognition. In *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*, pages 243–248. IEEE, 2014.

[Knu74] Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[KSLO12] M Kachouane, S Sahki, M Lakrouf, and N Ouadah. Hog based fast human detection. In *Microelectronics (ICM), 2012 24th International Conference on*, pages 1–4. IEEE, 2012.

[LKC12] Amine Lamine, Mahdi Khemakhem, and Habib Chabchoub. Knapsack problems involving dimensions, demands and multiple choice constraints: generalization and transformations between formulations. *International Journal of Advanced Science and Technology*, 46:71–94, 2012.

[LLSW04] Tyng-Yeu Liang, Yen-Tso Liu, Ce-Kuen Shieh, and Chun-Yi Wu. A new approach to distribute program workload on software dsm clusters. In *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, pages 201–206, May 2004.

[LMM02] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241 – 252, 2002.

[LTWT14] Ping Luo, Yonglong Tian, Xiaogang Wang, and Xiaoou Tang. Switchable deep network for pedestrian detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 899–906, 2014.

[LX09] P. Li and L. Xiao. Mean shift parallel tracking on gpu. *Pattern Recognition and Image Analysis*, page 120?127, 2009.

[MC15] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. 2015.

[MH14a] D. Malysiak and U. Handmann. An efficient framework for distributed computing in heterogeneous beowulf clusters and cluster-management. In *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*, Nov 2014.

[MH14b] Darius Malysiak and Uwe Handmann. An algorithmic skeleton for massively parallelized mean shift computation with applications to gpu architectures. In *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*, pages 109–116. IEEE, 2014.

[MLU16] Darius Malysiak, Angela Lausberg, and Handmann Uwe. Simplehydra. Technical report, Hochschule Ruhr West, 2016.

[MSSU11] Christian Mller-Schloer, Hartmut Schmeck, and Theo Ungerer, editors. *Organic Computing - A Paradigm Shift for Complex Systems*. Birkhaeuser, 2011.

[MZLC14] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. Benchmarking the memory hierarchy of modern gpus. In *IFIP International*

Conference on Network and Parallel Computing, pages 144–156. Springer, 2014.

[NHH11] Woonhyun Nam, Bohyung Han, and Joon Hee Han. Improving object localization using macrofeature layout selection. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 1801–1808, Nov 2011.

[NVi12a] NVidia. Kepler gk110 whitepaper. 2012.

[NVi12b] NVidia. Opencl programming guide for the cuda architecture. 2012.

[NYC14] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. *CoRR*, abs/1412.1897, 2014.

[OFJ09] Chris Oei, Gerald Friedland, and Adam Janin. Parallel training of a multi-layer perceptron on a gpu. Technical report, ICSI Technical Report, 2009.

[OLG+07] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[Owe04] John Owens. Gpus: Engines for future high-performance computing. Technical report, DTIC Document, 2004.

[P+94] Lutz Prechelt et al. Proben1: A set of neural network benchmark problems and benchmarking rules. 1994.

[PR09] Victor Prisacariu and Ian Reid. fasthog - a real-time gpu implementation of hog. Technical Report 2310/09, Department of Engineering Science, Oxford University, 2009.

[PW10] Mihai Patrascu and Ryan Williams. On the possibility of faster sat algorithms. In *SODA*, volume 10, pages 1065–1075. SIAM, 2010.

[RIdONF] Klaus Raizer, Hugo Sakai Idagawa, Eurípedes Guilherme de Oliveira Nóbrega, and Luiz Otávio Saraiva Ferreira. Performance comparison of training methods for mlp neural networks in gpu.

[RMN09] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.

[Rob05] Sara Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM news*, 38(9):1–3, 2005.

[Rog12] Jim Rogers. Power efficiency and performance with ornl's cray xk7 titan. In *SC Companion*, pages 1040–1050, 2012.

[RR10] Thomas Rauber and Gudula Rünger. *Parallel Programming - for Multicore and Cluster Systems*. Springer, 2010.

[RR12] T. Rauber and G. Rünger. *Parallele Programmierung*. Springer Berlin Heidelberg, 2012.

[SBS+95] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[SL11] Patrick Sudowe and Bastian Leibe. Efficient use of geometric constraints for sliding-window object detection in video. In *Computer Vision Systems*, pages 11–20. Springer, 2011.

[SS02] B. Schölkopf and A.J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive computation and machine learning. MIT Press, 2002.

[Ste97] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1997.

[STE13] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. In *Advances in Neural Information Processing Systems*, pages 2553–2561, 2013.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[Sun90] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[SYT12] Mohamed Faidz Mohamed Said, Saadiah Yahya, and Mohd Nasir Taib. Analysis of different programming primitives used in a beowulf cluster. *Analysis*, 1(02), 2012.

[Tsa13] Brian Tsay. The tianhe-2 supercomputer: Less than meets the eye? *SITC Bulletin Analysis*, 2013.

[UPAM00] P. Uthayopas, S. Paisitbenchapol, T. Angskun, and J. Maneesilp. System management framework and tools for beowulf cluster. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 2, pages 935–940 vol.2, May 2000.

[UPAS01] Putchong Uthayopas, Sugree Phatanapherom, Thara Angskun, and Somsak Sriprayoonsakul. Sce: A fully integrated software tool for beowulf cluster system. In *Proceedings of Linux Clusters: the HPC Revolution*, pages 25–27. Citeseer, 2001.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[vdM07] Christoph von der Malsburg. The organic future of information technology. In Rolf P. Würtz, editor, *Organic Computing*, pages 2–24. Springer, 2007.

[VN14a] Mario Vestias and Horacio Neto. Trends of cpu, gpu and fpga for high-performance computing. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[VN14b] Jagtap et al Viragkumar N. Fast efficient artificial neural network for handwritten digit recognition. *(IJCSIT) International Journal of Computer Science and Information Technologies, Vol.5(2)*, 2014.

[Wer90] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[WHY09] Xiaoyu Wang, Tony X Han, and Shuicheng Yan. An hog-lbp human detector with partial occlusion handling. In *Computer Vision, 2009 IEEE 12th*

International Conference on, pages 32–39. IEEE, 2009.

[WL10] Leping Wang and Y. Lu. Power-efficient workload distribution for virtualized server clusters. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10, Dec 2010.

[WSR04] Christian Wiegand, Christian Siemers, and Harald Richter. Definition of a configurable architecture for implementation of global cellular automaton. In *Organic and Pervasive Computing–ARCS 2004*, pages 140–155. Springer, 2004.

[YSMR10] Dmitri Yudanov, Muhammad Shaaban, Roy Melton, and Leon Reznik. Gpu-based simulation of spiking neural networks with real-time performance & high accuracy. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[ZWSL14] Will Y Zou, Xiaoyu Wang, Miao Sun, and Yuanqing Lin. Generic object detection with dense neural patterns and regionlets. *arXiv preprint arXiv:1404.4316*, 2014.

Index

I

K

L

M

N

O

List of Algorithms

List of Figures

Glossary

$< \partial^l >$

The average value of a finite numerical set or sequence ∂^l, i.e. $< \{1, 2, 3\} = 1/3(1 + 2 + 3) >$. 76

Υ

A set of atomic computation units in a hypersystem. 88

$\beth$

The optimal schedule for a given hypersystem. 100

$\mathcal{P}$

A computable problem. 100

$\lceil n/f_{load} \rceil$

The round up value of n/f_{load}, e.g. $\lceil 2.4 \rceil = 3$. 51

$\mathbb{E}(.)$

Expected value for a random variable or random function. 32

$\max_l$

The maximal value, $\max_S(f(S))$, of the function f parameterized by a variable set S, e.g. $\max_x(\exp(x)) = \infty$ or $\max_j(s_j) = 3$ for a sequence $s = (1, 1, 3, 1)$. 78

$\{y_j\}_j$

An enumerable set $\{y_j\}_j$ of elements y_j. 53

$\mathcal{H}$

A hypersystem. 88

$\mathcal{O}, o, \Omega, \omega, \theta$

Asymptotic measures for space and time complexities of algorithms or systems. 13

$\mathcal{T}$

The problem of finding a sequence which minimizes the marginal time within the layers of a hypersystem. 96

$\mathcal{T}_{\mathcal{H}}$

The unique hypertree which represents the hypersystem $\mathcal{H}$. 100

$|a, b|_{pos}$

Measure for vector displacement, let $a, b \in \mathbb{R}^n$ be two vectors, then $|a, b|_{pos} := \sum_{i=1}^{n} |a^i - b^i|$ represents the sum of displacements along each dimension. 59

$(G_S\{...\})$

Notation in pseudocode which indicates a global region $\{...\}$, i.e. a space whose variables contained in S are accessible by all threads. 12

$(P_S\{...\})$

Notation in pseudocode which indicates a thread local region $\{...\}$, i.e. a space whose variables contained in S are only accessible by a single thread. 12

$G_{\mathcal{T}}$

The tile grid for a tile image $I_{\mathcal{T}}$. 120

$I_{\mathcal{T}}$

A tile image. 119

$\biguplus_{j=1}^{n} V_j$

The union of disjoint sets V_j. 11

$\mathcal{I}$

An image set whose elements are used to construct a tile image $I_{\mathcal{T}}$, also called an image base of $I_{\mathcal{T}}$. 119

$\delta_{\mathcal{T}}$

Packing density of a tile image $I_{\mathcal{T}}$. 119

$\subseteq_M$

A multiset-subset of a multiset, e.g. $\{a, a, b\} \subseteq_M \{a, b\}$. 46

$n||V|$

Depending on the context this expression can either mean that n divides $|V|$ or that n is concatenated with $|V|$, i.e. resulting in n followed by $|V|$; $n|V|$. 11

$\mathcal{A}_{x_v}(x, y, z, \&c)$

Notation for a variadic function signature, i.e. the function $\mathcal{A}_{x_v}$ may contain more parameters than x, y, z. 101

$\leq_P$

Polynomial reduction of a problem A to another problem B, i.e. $A \leq_P B$. 96

$|A|$

The norm of A, if not specified differently it holds that: $|A| = card(A)$ in case of sets, $|A| = det(A)$ in case of matrices and $|A| = abs(A)$ in case of numerical values. 47

Acronyms

API
Application Programming Interface. 25

BLAS
Basic Linear Algebra Subprograms. 62

BSP
Bulk Synchronous Parallel Computer model. 14

CPU
Central Processing Unit. 7

FPGA
Field Programmable Gate Array. 10

GPU
Graphics Processing Unit. 7

HDD
Hard Disk Drive. 38

HPC
High Performance Computing. 7

IP

Internet Protocol. 29

IPC

Inter-Process Communication. 27

MCKP

Multiple Choice Knapsack. 96

MP

Matrix multiplication Problem. 86

MPI

Message Passing Interface. 26

PPU

Parallel Processing Unit. 49

PR

Parallel Reduction. 51

PRAM

Parallel Random Access Machine. 14

PU

Inter-Process Communication. 49

PVM

Parallel Virtual Machine. 26

RAM

Random Access Memory. 40

RBF

Radial Basis Function. 47

RMSE

Root Mean Square Error. 108

ROI

Region Of Interest. 130

SH

SimpleHydra. 27

SIMD

Single Instruction Multiple Data. 18

SIMT

Single Instruction Multiple Thread. 61

SVM

Support Vector Machine. 124

TCP

Transmission Control Protocol. 29

UDP

User Datagram Protocol. 29

UDPRCS

UDP Remote Control Services. 37

vPPU

Virtual Parallel Processing Unit. 49

Lebenslauf

Persönliche Daten

Name:	Darius Malysiak
Titel:	Dr.-Ing.
Akademischer Grad:	Dipl.-Ing (FH), B.Sc., M.Sc.
Adresse:	Bauvereinstr. 1 45136 Essen
Telefon:	0179-9019868
Email:	darius.malysiak@ruhr-uni-bochum.de

Familienstand:	ledig
Staatsangehörigkeit:	Deutsch
Geburtsdatum:	21. Januar 1983, in Chorzow (Polen)
Sprachkenntnisse:	Deutsch (fließend), Englisch (fließend), Französisch (Grundkenntnisse), Polnisch (Grundkenntnisse)

Beruflicher Werdegang:

2012-2014	Mitarbeiter innerhalb des vom Bundesministeriums für Bildung und Forschung (BMBF) geförderten Forschungsprojekts *APFel*
Seit 2010	Wissenschaftlicher Mitarbeiter am Institut Informatik der Hochschule Ruhr West

2007	Erfolgreiche Teilnahme am IBM-Mainframe Seminar der Hochschule Bochum
2004	Werkstudent bei Siemens-Witten
Seit 2003	Staatl. geprüfter informationstechnischer Assistent
2003-2004	Zivildienst in der EDV-Abteilung der Katholischen Kliniken Essen-Nord
2002	Erfolgreiche Teilnahme am EIB-Seminar des Heinz-Nixdorf Berufskollegs

Schulischer Werdegang:

Seit 2013	Promotionsstudent der Fakultät für Elektrotechnik. Arbeitstitel: „Effiziente Algorithmen für robustes (kernel-basiertes) maschinelles Lernen
2008-2012	Studium der Mathematik (Nebenfach Physik) an der Ruhr-Universität Bochum, Ø-Note: 2.2, B.Sc.
2008-2011	Studium der angewandten Informatik an der Ruhr-Universität Bochum, Ø-Note: 1.7, M.Sc.
2004-2008	Studium der Elektrotechnik an der Hochschule Bochum, Ø-Note: 1.5, Dipl.-Ing. (FH)
2000-2003	Heinz-Nixdorf Berufskolleg (Essen), Erwerb der Fachhochschulreife
1993-2000	Gertrud-Bäumer Realschule (Essen), Erwerb der Fachoberschulreife mit Qualifikationsvermerk

1989-1993	Karlschule (Grundschule in Essen)

Akademische Abschlussarbeiten:

2008	Diplomarbeit „Untersuchung der Datenredundanz innerhalb von Bündelgraphen zur Gesichtserkennung“ am Institut für Neuroinformatik der Ruhr-Universität Bochum. Note: 1.0
2011	Masterarbeit „GPU Assisted Implementation of Kernel-Based Template Attacks“ am Lehrstuhl für eingebette Sicherheit der Ruhr-Universität Bochum, Note: 1.3
2012	Bachelorarbeit „Generic Decoding Algorithms for Multiple Syndromes“ am Lehrstuhl für Kryptographie der Ruhr-Universität Bochum, Note: 1.3
2016	Dissertation „Massive Parallelization of HOG-Based Algorithms for Object Detection“ am Institut für Neuroinformatik der Ruhr-Universität Bochum, Note 1.0, Magna Cum Laude