Doctoral Dissertation
Doctoral Program in Electronics Enginering (31st cycle)

Performance Optimization of Memory Intensive Applications on FPGA Accelerator

Arslan Arif
* * * * * *

Supervisor
Prof. Luciano Lavagno, Supervisor

Doctoral Examination Committee:
Prof. Jordi Cortadella Fortuny, Referee, Universitat Politecnica de Catalunya, Spain
Prof. Frederic Petrot, Referee, Universite Grenoble Alpes, France
Prof. Paolo F. M. Ienne Lopez, Ecole Polytechnique Federale de Lausanne, Switzerland
Prof. Mihai T. Lazarescu, Politecnico di Torino, Italy
Prof. Alex Yokovlev, Newcastle University, England

Politecnico di Torino
February 28, 2019

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Arslan Arif

Turin, February 28, 2019

Summary

Hardware accelerators are a fundamental part of modern high performance computing (HPC) systems due to their performance capabilities. The two most commonly used accelerators are GPUs and FPGAs. Despite the easier programmability and better memory performance of GPUs, generally FPGAs perform equally well for computationally challenging applications while dramatically reducing the energy consumption. Furthermore, with the availability of high level synthesis (HLS), the use of FPGAs has become easier. This makes them an excellent candidate for modern HPC systems. This dissertation describes my research work done in the field of electronic design automation with the major focus on optimizing memory intensive applications modeled using high level language for FPGAs. This work can be split into two parts, one dealing with manual memory optimization while other advocates the use of automated algorithms to select and optimize the best application-specific cache layout.

The first part covers the manual optimization of a realistic smart city application. The application implements two image processing algorithms in OpenCL language which computes velocity and density of vehicles on urban streets in real time. Several different implementations of these memory hungry algorithms are considered. The results show that using suitable optimizations and HLS optimization directives, FPGAs can produce results with performance similar to a GPU with an order of magnitude less energy consumption.

The second part of the dissertation starts by observing that custom data caches implemented on FPGAs are only useful if their layout is in accordance to their data access pattern. In this work, we present a tool, PEDAL (Pattern Evinced Determination of Appropriate Layout), that can automatically tune the custom data caches based on analyzing address traces. PEDAL uses artificial intelligence algorithms to detect the pattern of each array and then design the optimal cache for that pattern. The comparison of the results of PEDAL with the exhaustive search of cache configurations and cache designed through a state-of-the-art algorithm from the literature proves that it can produce better configurations in less time.

Acknowledgements

First and foremost, I would like to thank Allah Almighty for all His blessings bestowed upon me. Without His will, I would not have made it this far. It is His blessing that I am able to complete my research project. It was He who provided me enough courage and strength to take on this challenge, accomplish it and make every hurdle easy for me.

I attribute myself lucky to have such a wise and kind supervisor and mentor, Prof. Luciano Lavagno, who was available to me in every hour of consultation and need. His expertise, patience, demeanor, and tolerance are some of the qualities for which I am thankful to him. He made this task as easy for me as was possible. He was always there to address my work-related as well as other general queries even with his extremely busy schedule.

Besides, I would also like to thank all the faculty members and researchers from the department of electronics and telecommunications (DET) at Politecnico di Torino, especially Prof. Mihai Teodor Lazarescu, who supported me in many ways during the course of my PhD. I would also like to take this opportunity to acknowledge the support of my group mates (past and present) at the High-level synthesis group at Polito, especially Liang Ma and Fahad bin Muslim. The discussions with them have always been extremely rewarding and I learned a lot from them.

My immense thankfulness goes for my family members including my parents, siblings and others for their unconditional support and prayers. This goes without saying that my wife Anum has a huge contribution in my degree.

Finally, I am extremely thankful to the higher education commission (HEC) Pakistan for funding my doctorate at the Politecnico di Torino. This is a great initiative by the government of Pakistan to create a pool of high quality researchers who can ultimately contribute to the prosperity of the nation. I find myself more equipped after my PhD to contribute to this goal.

I would like to dedicate this thesis to my loving parents

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the modern day world, the level of automation in the industry is constantly reaching new horizons. Both industry and the society at large are increasingly exploiting machine learning (ML), artificial intelligence (AI), Internet of Things (IoT) and Big Data applications. This in turn will make high performance computing the core of almost every industry. In order to yield this level of productivity, modern electronic devices are not only enhanced to perform multitasking, but are also required to do it in real time. Moore's law is reaching its end, mostly due to economic reasons, because only a few companies can afford to pay the exponentially increasing mask costs, and even increasing per-transistor costs. Thus the only approach to keep improving performance to the level required by the above mentioned application, with a reasonable energy cost, is to move some of the software load to dedicated heterogeneous architectures, where the heaviest parts are accelerated in hardware.

The main purpose to employ heterogeneous systems is to obtain the required performance for computationally expensive applications while achieving better energy efficiency [32], [46]. These systems generally consists of a multicore CPU along with various kinds of (typically programmable) accelerators. General purpose graphical processing units (GPGPUs) are traditionally used as accelerators as they provide the highest performance, albeit with a staggering energy-per-operation cost. The main issue with GPU based heterogeneous systems is that they are inefficient in terms of power consumption. In contrast to that, field programmable gate arrays (FPGAs) provide considerable performance while only consuming a fraction of energy as compared to GPUs. Hence, FPGAs are a strong competitor of GPUs for modern high performance computing (HPC) systems.

Although the computational capabilities of FPGA-based heterogeneous systems are very high, these systems require optimal data handling techniques in order to be effective. The main challenge in optimizing these systems is handling the memory bottleneck. This occurs when the data processing speed of the system is greater than the speed at which memory is able to provide (or, less frequently,

store) data. This is due to the fact that traditional CPU caches, which gives the programmer the illusion of a huge, yet very fast flat memory space, are not available for the portion of the computation that is offloaded to the FPGA. Thus the designer himself is responsible to optimize the HW memory architecture. The time required to fetch data and make it available for processing should be (by Amdahl's law) comparable with the processing frequency to obtain optimal results. The amount of data modern applications are dealing with ranges from hundreds of megabytes to terabytes. Therefore, this data needs to be stored in larger memories (DRAM or even Flash) which are slower to access. These memory accesses are costly not only in terms of time but also in terms of power and energy consumption. Moreover, as we discussed before, any delay in availability of data, also increases the computational time.

Now we conclude from the above discussion that memory is the major bottleneck for HPC systems. There is no point in optimizing the computations without optimizing the data as we cannot achieve high computational throughput if we do not have the data to process. In this work we focus on a HW design methodology which starts from a C/C++ specification, where the memory access model reflects modern CPU architecture, and hence it is not typically suited for HW implementation. Hence the first thing that we need to optimize is memory accesses. The work presented in this dissertation considers memory hungry and computationally expensive applications as test cases and our results validate our hypothesis that *without optimizing the data transfer for on chip design, state-of-the-art high-level synthesis tools fail to work*. We have provided manual and automated solutions to improve the availability of data for applications.

1.1 FPGA based heterogeneous computing system

As discussed above, high performance computing requires sophisticated hardware and software resources. The performance of a processor can no longer be increased, since the early 2000's, by simply increasing its clock frequency, due to power reasons[9]. Moreover, traditional processor parallelism, in the form of superscalarity or hyper-threading, has also long reached its limits. Finally, multi-core processors are very energy-ineffficient due to the fetch-decode-execute cycle and the very general-purpose datapath and memory architecture that they offer. Therefore, it is a general consensus that heterogeneous systems are needed to provide the required performance in this regard.

Modern computing devices should have the capability to process large amounts of data. Extraction and categorization of vast amounts of data requires expensive and sophisticated software. For example, in the field of image processing, processing the live feed for even a single camera requires a dedicated central processing

unit (CPU) [28]. This need of more performance requires computer accelerators. The most commonly used computer accelerator in this domain is the Graphical Processing Unit (GPU). GPUs provide higher memory bandwidth, higher floating point throughput and a more favorable architecture for data parallelism than processors. Due to these properties, they are used in modern high performance computing (HPC) systems as accelerators [74]. However, the main drawback of HPC systems based on GPU accelerators is that they consume large amount of power [25].

To overcome the power inefficiency of GPU-based HPC systems, modern field programmable gate arrays (FPGAs) can be used. FPGA devices require less operating power and energy per operation while providing reasonable processing speed as compared to GPUs [54]. When comparing them with multi-core CPUs, especially with regards to data center applications, it was observed that the performance gap keeps widening between the two. In summary, FPGAs are known to be more energy efficient than both CPUs and GPUs [69]. Moreover, FPGAs are well known for their reconfigurability as well as their energy efficiency. Acknowledging these capabilities, Microsoft, Baidu and Amazon now also use FPGAs as accelerators rather than GPUs in their data centers [53].

FPGAs are, however, complex to program. Hardware description languages (HDL) such as Verilog or VHDL are commonly used for this task. Most of the modern applications are developed in high level languages and it requires a lot of effort on the designer's end to define corresponding modules. Moreover, designer cannot explore many micro-architectural options using these low level languages due to long design and verification cycles.

To counter the issue of programming in low level HDL languages, designers now focus on high-level synthesis (HLS). HLS promises to generate register transfer logic (RTL) directly from algorithms written in high level languages e.g C, C++, OpenCL or SystemC. Moreover, HLS tools provide a number of directives to facilitate the designer in exploring different micro architectural solutions. Thus, we can say that HLS provides the capability to program FPGAs through the use of high-level languages, consequently reducing the design time debugging and analysis [50, 23].

1.2 Problem Statement

The discussion above concluded that off chip memory accesses are expensive. They are not only expensive in terms of execution time, but also in terms of power and energy consumption. Moreover, if memory accesses are not optimized, then even other operations cannot be scheduled for efficient execution.

For example, in the simple case of vector addition, the only operation is a sum of two numbers. But if the data is stored in external DRAM, then it needs to

fetch the required elements from both the arrays, then it will perform addition and after that it will write the result back to the global memory. This means that in order to pipeline its computation with an Initiation Interval of 1 (i.e. starting a new elementwise addition every clock cycle) the global memory interface needs to support three operations (2 reads and 1 write) per clock cycle. This is very hard to sustain, even with modern DDR3 and DDR4 interfaces, for more than one such vector addition kernel on an FPGA (and a modern FPGA can support thousands of such computations in parallel). This quickly leads to saturation of the memory interface, and requires more efficient access techniques, that will be explored in this thesis. Thus the overall performance of the code degrades. If, hypothetically, we could make the whole data available on chip, then the operations in the work-items can be executed more efficiently. However, data can be copied on chip beforehand only for small applications, while for memory intensive applications involving image processing or machine learning, this option is not always feasible. Therefore we need to have a compromise between the two options to find the most optimal point.

In this thesis, the main focus is to improve the performance of a code which is memory bound, i.e. which requires a large amount of data transfer between external DRAM and silicon chip. The optimization can be application specific i.e. manually annotating the source code for fetching the memory in an appropriate way, or can be automatic, i.e. by using a tool to optimally implement this fetching for the designer.

In the first portion of the thesis, we want to optimize a real time application which is memory bound manually. The application implements two image processing algorithms on live video streaming of high definition (HD) quality (1280x720) at a rate of 25 frames per second. The main goal is to achieve real time video processing which requires a data transfer of more than 1.5 gigabytes (GB) per frame in unoptimized form.

The second part of the research considers the use of some *plug and play* application-specific caches to make the life easier for designer. The main aim of this work is to find an automated way to optimally choose the parameters for these caches, so that they can be used for any application without virtually any designer's effort.

1.3 Contributions

In this thesis, the focus of the research is to optimize memory bound applications for FPGAs. As discussed in section 1.2, the work is divided into two major sections. One section deals with the manual optimization of highly memory bounded application while the other section proposes a tool to find optimal cache layouts. All of the work is done for FPGA applications written in high level languages, such as C/C++ or OpenCL.

In the first portion of the thesis, a realistic smart city application is optimized

for real time processing. The application processes incoming live video streams from the cameras to get velocity and density information for traffic on roads. It uses two image processing algorithms, Optical Flow and Background Subtraction that are computationally as well as memory expensive. The challenge of their real time implementation is met using GPUs and FPGAs, which were not feasible without accelerators. A very large design space was explored for multi architectural solutions and then the best solution, with respect to the end-user constraints (a company providing smart city infrastructure) was selected. All of the optimization done in this regard are explained in Chapter 3. These optimizations, although are application specific, can also be utilized with other memory hungry applications. The comparison of both accelerators shows that FPGAs are more suitable in terms of power and energy consumption than both CPUs and GPUs. High-level synthesis (HLS) with manual code optimizations is used to get the desired FPGA hardware and performance from GPU-optimized OpenCL code. Finally, the whole application is verified using Amazon Web Service (AWS) machines with FPGAs for functional verification and performance analysis. The final proposed design was able to process live video feed from roads for detection of number of vehicles and their speed.

The second part of the dissertation focuses on finding the application specific custom data cache configuration automatically. It discusses the architecture of an inline cache from the literature that reduces the programmer effort to optimize global memory (DRAM) accesses for any out-of-the-box code. This cache architecture previously needed manual intervention to find the most appropriate layout based on the application. This work first applies one of the best known general-purpose heuristic cache sizing algorithms that was developed in the literature to find the optimal cache configuration. The results of this algorithm are compared with exhaustive search to adapt the cache to different memory access patterns. The results show that this heuristics can result in very sub-optimal configurations in all cases. Therefore, this work presents a tool to find the optimal cache configuration for each array mapped to DRAM. Using PEDAL, the configuration to obtain the best cache configuration for a specific application is automatically selected based on data access pattern. The results are verified using different applications and benchmarks. The cache architecture, layouts and data access patterns are discussed in detail in Chapter 4. It also discusses the tuning heuristic and algorithms and compares the performance of both of them.

1.4 Organization of the thesis

This thesis presents a collection of the work done in the field of electronic design automation (EDA) for FPGAs using high-level synthesis (HLS) with an emphasis on designs with efficient off-chip memory accesses. The work is divided into two

major portions and their organization is as under:

- Chapter 2 discusses Heterogeneous Systems and their architectures. It also explains their advantages and disadvantages. Moreover it also provides some introduction about high level synthesis (HLS) and the OpenCL programming platform.
- Chapter 3 discusses a video processing application which was co-designed in cooperation with Acciona, a Spanish company providing smart city solutions, in the context of an H2020 European project, and which was optimized by me for FPGA implementation. This chapter discusses in detail two commonly used video processing algorithms, their implementation and results generated by them to help the traffic flow. This chapter also includes the optimization carried out in order to achieve the required performance of 25 frames per second.
- Chapter 4 explains the architecture for custom inline data caches designed to be synthesized for FPGAs. It also discusses an automated algorithm that tunes the cache to obtain the best layout for the application under consideration. The effectiveness of the algorithm is tested against heuristics for different applications and benchmarks.
- Chapter 5 concludes the work. It also states the possible work which can be done in this field in future.

Chapter 2

Heterogeneous Systems

High-Performance Computing and data-intensive applications, such as Machine Learning, Artificial Intelligence, and big data processing, are becoming more and more common both in large data centers and on embedded platforms. Thus, while the processing speed of, e.g., Neural Network training or database sorting, remains a primary concern, energy consumption is quickly gaining importance. Homogeneous hardware architectures, e.g., multi-core general purpose Xeon processors, no longer meet the heaviest computation requirements especially from the point of view of energy efficiency [33].

A heterogeneous system refers to a system comprising of several different processors and cores. Such multi-core architectures offer high performance along with better power efficiency by not only using additional processor cores but by using specialized hardware called *accelerators* to handle certain computationally challenging portions of the applications. Thus, heterogeneous systems that cluster together different types of processors and hardware, such as CPU-GPU or CPU-FPGA, are able to achieve the best performance/cost/energy trade-offs for computationally-intensive parallel algorithms [71].

2.1 Heterogeneous System Architecture

As discussed before, a single core processor cannot provide the required performance for modern applications. Multicore processors have shown some promise in this field. As the name suggest, multicore processors are a number of processors packed in a single chip. They are able to show task and data level parallelism using parallel programming libraries [74], i.e OpenMP or MPI. These multicore processors still cannot beat the advantage we can obtain from system with hardware accelerator support, neither in terms of performance, nor in terms of energy efficiency.

Simple Xeon processor based systems are not as efficient as currently available

systems with accelerator support. A key point to keep in mind here is that these accelerators do not operate in stand alone fashion but they rely on traditional processors to manage them. CPUs are responsible to initiate the accelerators and upload and download the data as required by them [14], [30]. This make the heterogeneous systems more powerful as they take benefit from computationally advanced accelerators while the scheduling is done by multi-purpose central processing unit.

The accelerators used in such heterogeneous systems may be GPUs, FPGAs or a combination of both. In the terminology of heterogeneous system architecture, the multi-core processor is typically called a *host* while the hardware platforms used to accelerate certain portions of the applications are called *devices*. A typical heterogeneous system is shown in Fig. 2.1. The various important components of such a heterogeneous system are described here briefly.

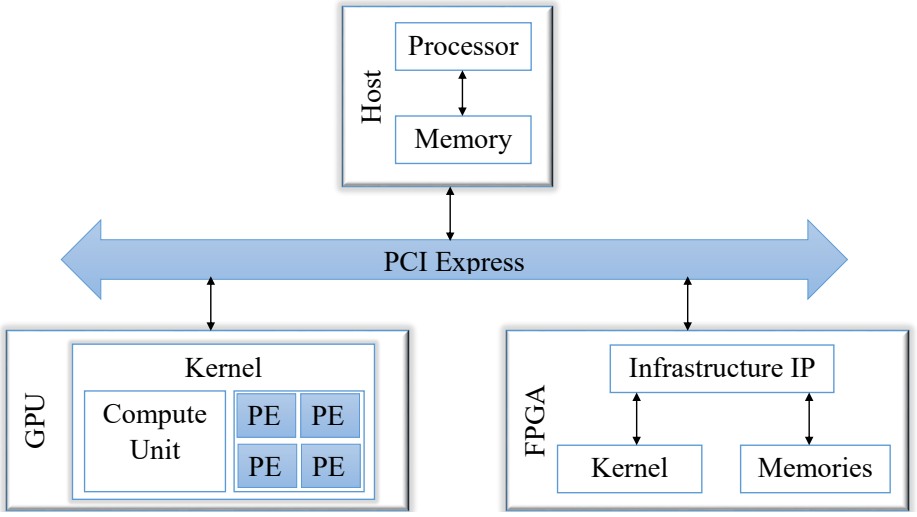


Figure 2.1: A Typical Heterogeneous System Architecture

2.1.1 Graphics Processing Units

As the name suggest, graphical processing units were originally designed to handle the advancement in field of graphics. The interest for GPUs or GPGPUs (general purpose graphical processing units) in the field of high performance computing developed after 2007, when NVIDIA introduced parallel programming framework, CUDA. CUDA (Compute Unified Device Architecture) is a parallel programming framework, designed for programming GPU based heterogeneous systems.

GPUs mainly consists of several processing elements which execute the kernel in parallel manner. These streaming processors are generally multicore and have

several components including ALUs (Arithmetic Logic Units), load/store units, caches etc. They execute the kernel code in SIMD (Single Instruction Multiple Data) fashion, which means that streaming microprocessors implement the same set of instruction to different data. In GPUs more emphasis is on data operations instead of data control and caching and hence their caches are smaller as compared to CPUs. GPUs also have their own device memory of few gigabytes [14].

GPUs do not operate in stand alone fashion. They always act as co-processor with CPUs, where CPU acts as a *host* and GPU acts as a *device*. Host is responsible for enviroment setting and data management for the device and they are connected through PCI-Express bus as shown in Fig. 2.1.

2.1.2 Field Programmable Gate Arrays

The primary use of FPGAs was to implement discrete logic. Currently, recognizing their abilities and potential, their application is expanded to a variety of fields ranging from embedded systems to high performance computing systems [14]. Modern FPGAs provide a great alternative to GPUs in the field of high performance computing. The main issue with the GPUs is their energy inefficiency [25] which makes the case for FPGAs even strong. FPGAs can provide same amount of computational abilities while consuming a fraction of power. The application specific architecture of FPGAs reduces the need of multiplexing which provides great energy saving. Similarly hardwired control logic eliminates a lot of control instructions and is also a major reason for energy efficiency of FPGAs.

Unlike GPUs the architecture of an FPGA is not fixed, but can be customized according to the requirements of the application. A typical FPGA consists of logic blocks, memory blocks and DSP slices each surrounded by programmable interconnects as shown in Fig. 2.2. The interconnects among different logic and memory blocks of FPGAs are programmable, which provide a lot of flexibility to its architecture. Similarly, the I/O blocks are also not fixed and can be programmed according to the application.

The flexible FPGA architecture where provide a lot of opportunities to the designer on one end, it also posses some challenges for designers to optimally configure the architecture. Therefore, designers need to have good knowledge of the hardware and corresponding configurations in order to obtain good results. This problem is however resolved by the use of HLS (section 2.2) and languages like OpenCL (section 2.3).

2.1.3 ECOSCALE

Many HPC centers are operating around the globe. Some of them are implemented using CPUs only while others have accelerator support in them as well. Previously the trend was to use GPUs in data centers but currently acknowledging

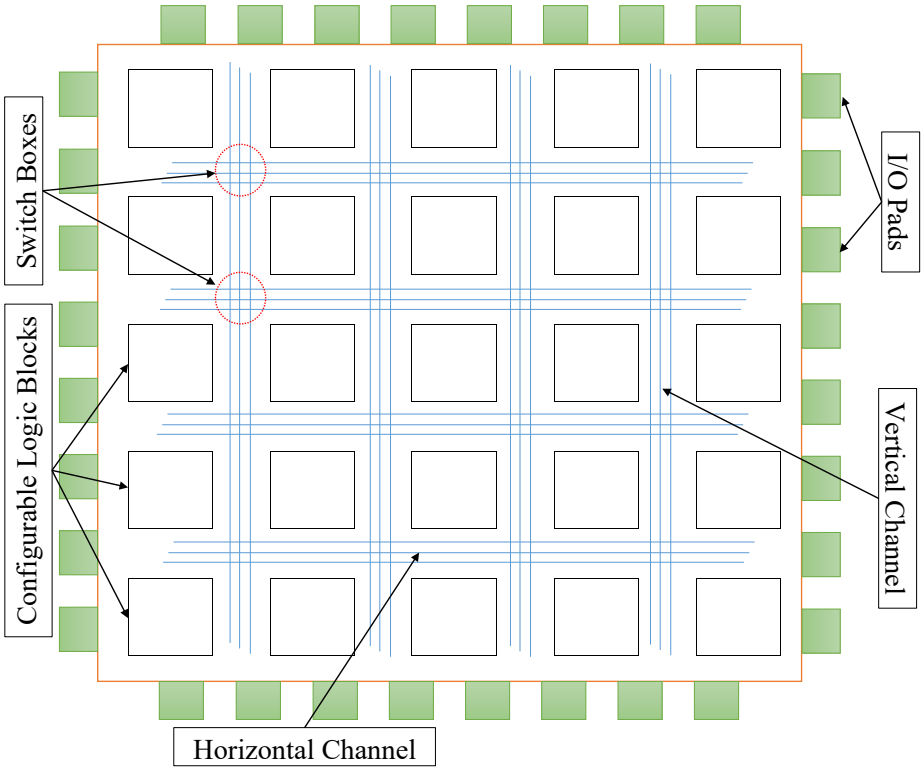


Figure 2.2: FPGA Architecture

the capabilities of FPGA, some of the major data centers in world like Microsoft, Amazon, Baidu are also using FPGAs rather than GPUs [53].

There are certain HPC servers that target to provide an energy-efficient architecture by sharing numerous reconfigurable accelerators. In order to provide a scalable approach, the architecture should be tailored to the needs of the HPC applications as well to the characteristics of the hardware platform. ECOSCALE (Energy-efficient heterogeneous COmputing at exaSCALE) is a project under the H2020 European research framework. The main goal of this project is to provide a hybrid MPI+OpenCL programming environment, a hierarchical architecture, a runtime system and middleware, and a shared distributed reconfigurable FPGA based acceleration [27].

ECOSCALE offers a hierarchical heterogeneous architecture with the purpose of achieving exascale performance in an energy-efficient manner. It proposes to adopt two key architectural features in order to achieve this goal: UNIMEM (Unified Memory) and UNILOGIC (Unified Logic). UNIMEM was first prpoosed by the EUROSERVER project [26] and provides efficient uniform access, including low-overhead ultra-scalable cache coherency, within each partition of a shared Partitioned Global Address Space (PGAS). UNILOGIC, which is first being proposed

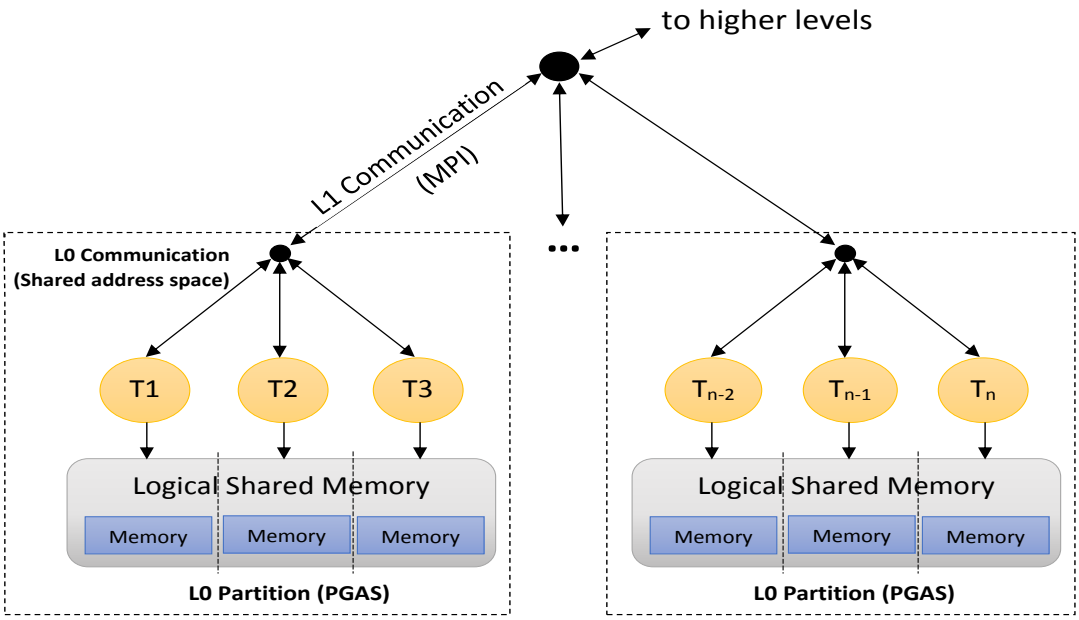


Figure 2.3: Hierarchical partitioning (tasks, memory, communication) of an HPC application in ECOSCALE platform [27]

by ECOSCALE, extends UNIMEM to offer shared partitioned re-configurable resources on FPGAs. The proposed HPC design flow, supported by implementation tools and a run-time software layer, partitions the HPC application design into several nodes. These nodes communicate through a hierarchical communication infrastructure as shown in Figure 2.3. Each Worker node (basically, an HPC board) includes processing units, programmable logic, and memory. Within a PGAS domain (several Worker nodes), this architecture offers shared partitioned re-configurable resources and a shared partitioned global address space which can be accessed through regular load and store instructions by both the processors and the programmable logic. A key goal of this architecture is to be transparently programmable with a high-level language like OpenCL.

ECOSCALE targets to provide an energy-efficient architecture by sharing numerous reconfigurable accelerators. In order to provide a scalable approach, the ECOSCALE architecture should be tailored to the needs of the HPC applications as well to the characteristics of the hardware platform.

2.2 High Level Synthesis

The main hindrance in exploiting the potential of FPGAs was the tedious job of coding them in hardware description languages. The use of FPGAs is now very convenient thanks to high level synthesis (HLS). Generation of quality register transfer level (RTL) from high level specifications is a great achievement in electronic design automation. Previously, HLS designs were considered inefficient, but after the recent development in designing tools and availability of different synthesis directives, it is gaining the interest of designers. It not only reduces the designer's efforts but also provides fast design cycles by minimizing the manual effort.

Designing a suitable hardware requires a number of steps to be followed in specific order. Most of the hardware design projects start from an executable model of high level language. This model is generally developed to verify the behaviour of the task to be performed. This model is tuned and tested at different stages to verify the correct functionality of the model. Once tested and verified, then this model goes under a number of steps before it takes form of an actual hardware implementation. The final architecture is then described in the form RTL, generally written in VHDL or Verilog. There are certain drawbacks of this process, including long design and verification cycles and manual nature of the process. High level synthesis tools can automate this whole process into an error free path from abstraction to RTL generation.

Some of the major advantages provided by high level synthesis are:

- Automates the whole process from abstract level design to RTL generation
- Accelerated design times
- Provides directives to explore the whole design space just by small modifications
- Debugging the algorithm is much easier and less time consuming
- Allows high level of portability between different platforms
- In most of the cases designers do not need to worry about detail architectures, i.e. clocks, design hierarchy, processes etc.
- Modules once synthesized can be reused more effectively

The reduction in design efforts allows the designer to freely focus on their main design functionality and care less about implementation details. These details are automatically tuned by the tool according to the design specifications and hardware selected. Another major advantage as stated above is the portability among different platforms. This also means that designer can switch between the hardware to choose the best according to his needs.

2.2.1 High-level synthesis based Design Space Exploration

Modern FPGAs, such as the Stratix from Altera and the Virtex, UltraScale families from Xilinx, offer to the designer millions of Configurable Logic Blocks (CLBs) and Flip-Flops, megabytes of on-chip the Block RAM (BRAMs), hundreds of multiply-and-accumulate units (DSPs), and many other dedicated hardware blocks, including ARM Cortex processors [80]. Moreover, very recent design flows from both Altera/Intel and Xilinx promise software-like development for applications that are entirely written in a high-level language, like C, C++ or OpenCL, and are then compiled and synthesized for heterogeneous CPU-FPGA platforms. In particular, parallel languages that were originally developed to program GPUs, can now be used to program heterogeneous platforms such as PCs with FPGA boards, or Zynq platforms which include a multi-core CPU and a large FPGA [66].

However, the expected performance is typically not achieved by simply recompiling, via High-Level Synthesis for an FPGA target, an algorithm that was originally written for execution on a CPU or GPU. This is because the CPU or GPU architectures are fixed, hence most compiler decisions are local and relatively simple, such as intra-basic block scheduling or peephole optimizations. However, in an FPGA *the architecture is adapted to the application, rather than the application to the architecture*. While this can achieve much better optimization levels, it also implies that many more high-level decisions must be made during synthesis. HLS tools are able to automatically implement these decisions, but even their latest generations need to be directed to do so by a human or by a (very time-consuming) Design Space Exploration tool.

While the optimizations performed by a CPU or GPU compiler are considered excellent when they speed up execution by a factor of 2, the following HLS techniques can dramatically optimize the execution time of algorithms on FPGAs even by orders of magnitude. Most of them apply to loops, which are a major source of concurrency in high-level code and some languages, such as OpenCL, explicitly state that some loops can be arbitrarily parallelized, because iterations do not depend on each other:

1. *Loop pipelining* starts new iterations of a source code loop before the previous ones are completed. It is one of the best options for loop optimization in HLS, since it usually boosts the performance at a very low cost [29, p. 61]. The number of clock cycles between successive loop iteration starts (inverse of the throughput) is also called the "Initiation Interval" of the pipeline (in the best case, it can be one clock cycle). It is fully decoupled from the time it takes to complete one iteration, the pipeline "latency". Usually, memory or data dependencies between successive iterations ("loop-carried dependencies") are the bottlenecks that increase the initiation interval. Several other synthesis techniques, e.g., array partitioning or loop interchange [43], can be applied to ameliorate this problem.

2. *Loop unrolling* creates multiple copies of the loop body to be executed fully in parallel. In some cases it can achieve even more performance than by means of pipelining, but typically at a huge resource (i.e., area) cost. A loop can be fully or partially unrolled and in both cases the maximum performance can be achieved only by means of array partitioning and may require arithmetic evaluation restructuring (e.g., adder tree balancing) [29, p. 51]. In OpenCL (similar to CUDA), the loop over work groups can be unrolled arbitrarily by definition. Thus, like on a GPU, the performance on an FPGA can be increased by instantiating multiple work groups until the computing or routing resources, or its memory bandwidth are saturated [66, 49]

3. *Exploiting on-chip memory.* Most modern FPGAs integrate thousands of independent BRAMs on chip for a total of many MBs of storage. Accesses to these memories are both much faster in terms of latency and much more parallelizable than those to off-chip memories [78]. Many algorithms, especially the memory-intensive ones that are addressed in this article, achieve the best acceleration only by moving frequently-accessed data that reside in off-chip memories into on-chip BRAMs (or another kind of FPGA memory called LUTRAMs). As mentioned above, on-chip memories that are not carefully optimized by using partitioning directives can often become bottlenecks, because of the limited number of access ports that they offer. While on a GPU the maximum number of concurrent accesses to independent addresses (and the meaning of "independent") is fixed by the GPU architect, on an FPGA it must be carefully chosen by the designer, because more parallelism often implies a higher cost. Memory partitioning or memory reshaping according to user directives or to automated analysis of access patterns of a given algorithm can dramatically increase the memory bandwidth and achieve a much higher level of concurrency.

4. *Optimizing global memory interfaces.* Other methods to improve performance include instantiating multiple DRAM access ports or increasing their bit width.

   On a GPU, the global memory interface subsystem receives memory read or write requests from the threads or work items that are executing on its compute units, and *coalesces* these requests whenever possible, in order to match both the available memory word size and bus burst transfer capabilities. For example, 16 accesses to adjacent properly aligned 32-bit integer array elements can be grouped *automatically at runtime* into a single 512-bit memory read, or to a burst of 4 128-bit memory reads, depending on the DRAM interface width.

   On an FPGA, these groupings must be performed manually and at compile time, which requires a lot of design and tool usage expertise.

2.3 Open Computing Language

Open Computing language or *OpenCL* is a parallel programming framework based on C99 and C++11 which support parallel programming model. It is widely used for programming heterogeneous and multicore platforms [34], [68]. As the name suggests, it is an open source standard which is maintained by Khronos Group.The main advantage offered by OpenCL is execution portability. It allows to run the same design accross different platforms with few modifications.

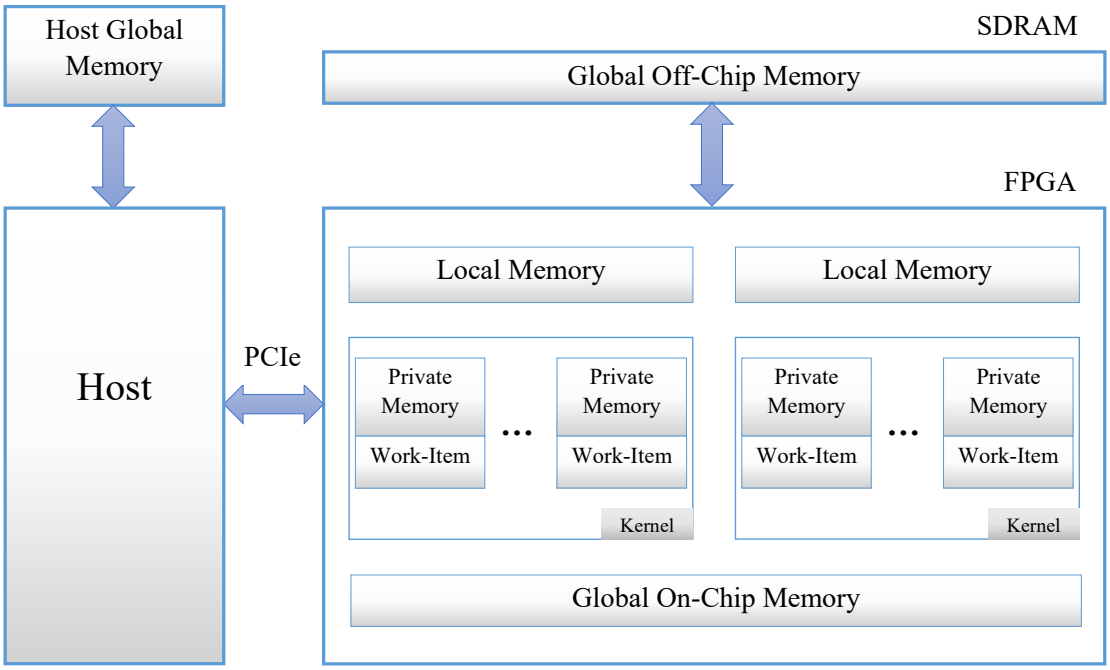


Figure 2.4: Platform and Memory Model of OpenCL

Some definitions/terminologies used by OpenCL model are shown in ifg. 2.4 and defined as under:

- **Host:** It is a (multi-core) processor, which is responsible for setting up the environment and managing tasks on the *device*.
- **Device:** Device is the word that represents the hardware accelerator in the system
- **Kernel:** It is the computationally expensive piece of code that is designed to run on the *device*.
- **Compute Unit (CU):** An OpenCL device can implement multiple copies of same design, called compute units.

- **Work-items & work-groups:** Concurrent implementations of kernel body are termed as work-items and a collection of these work items is called work groups. The designer can choose the size of work-items per work-group.
- **Global Memory:** It the shared memory between all the work-groups. It is the slowest device memory to access and have the size order of a few gigabytes (GB).
- **Local Memory:** This is a private memory of each work group, which means it is shared by all the work items within that work-group. It is faster than global memory.
- **Private Memory:** It the smallest, but fastest memory to access in the whole model. It is the private memory of each work-item and generally use to store some temporary variables.

It should be noted that memory management in OpenCL is done explicitly i.e. by moving data from host memory to global memory to local memory and then back. To ensure memory consistency and provide better synchronization within a workgroup, if needed, OpenCL also provides the concept of *barriers* [81].

Chapter 3

Smart City Application

Cities are seeing massive urbanization worldwide, thus increasing the pressure on infrastructure to sustain private and public transportation. Adding intelligence to traditional traffic management and city planning strategies is essential to preserve and even improve quality of life for citizens under this enormous increase of population. Traffic causes increased delays, thus reducing the opportunity for city dwellers to earn money by performing productive activities. It also poses health hazards due to pollution and accidents. Several public and private entities (ranging from public transportation providers, to city planners, to traffic light control, to taxi and car sharing providers, to individual drivers) can profit from the widespread availability of real-time information about traffic flows. Part of the work described in this chapter has been previously published in "Performance and energy-efficient implementation of a smart city application on FPGAs" [8].

This application can improve traffic-related problems in modern continuously growing cities based on the information provided by the citizens and/or extracted by monitoring their habits. Various methodologies and sensors can be used to achieve this goal.

This application will provide cost-effective and scalable real time analysis of traffic in cities that can then be harnessed by other smart city services and applications (e.g. intelligent traffic management tools) in order to reduce traffic-related impacts on the quality of life of citizens. Videos obtained from cameras can provide reliable information about the traffic flow on roads. The basic idea, as shown in Fig. 3.1. is that the cameras acquire the images, which are then processed using image-processing algorithms. After that, the data is stored in a database and accessed on demand.

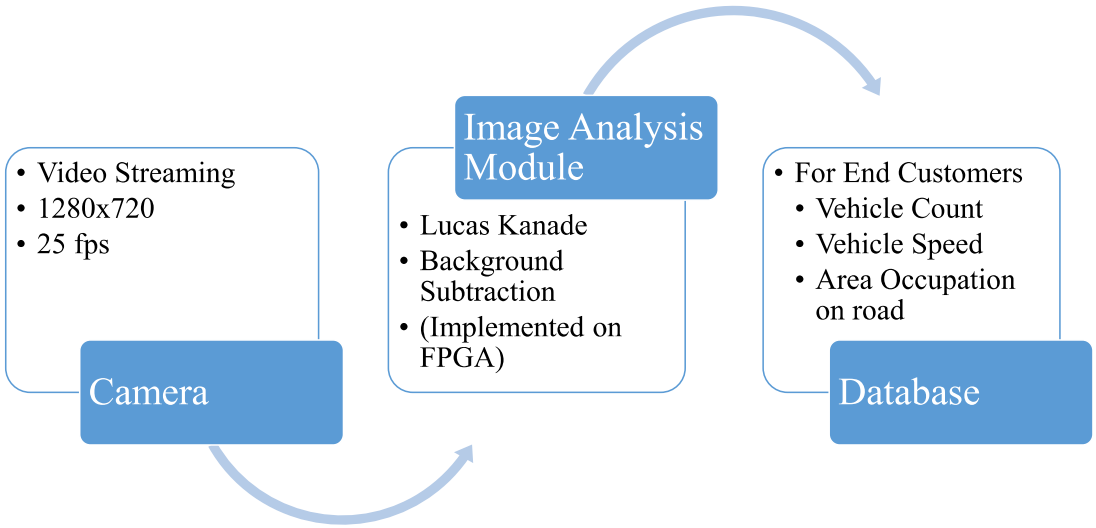
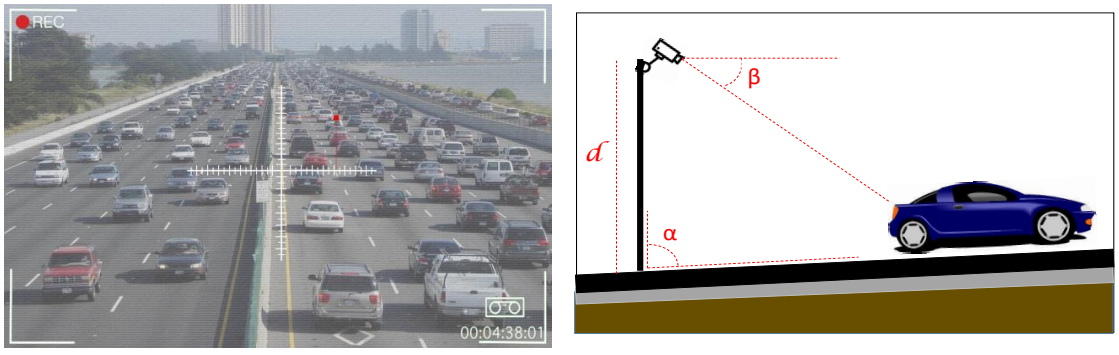


Figure 3.1: Application Overview

3.1 Application

The main goal of the application described in this work is to extract data from video surveillance cameras and make it available to different services. The objective is to provide real-time information which can be used to optimize, for example the street lighting and traffic light systems installed in cities. The application will analyze the images recorded by the cameras installed in cities and will apply a set of algorithms in order to detect the presence of people and vehicles and to compute the density of traffic at each specific location.



(a) Camera view of the road

(b) Road Parameters w.r.t camera

Figure 3.2: Camera view

For this purpose, cameras are installed on roads (Fig. 3.2a). Their parameters like height from ground, angle of elevation etc. and road parameters like width, etc.

are already assumed to be available for processing, as shown in Fig. 3.2b, together with other constants like the minimum value for detecting a change of speed.

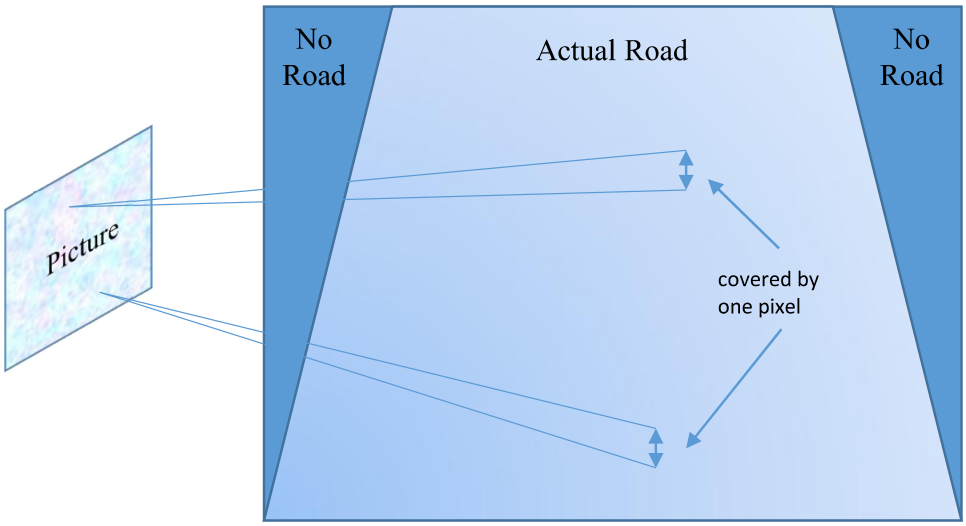


Figure 3.3: Video Frame vs Ground reality

In most places, cameras cannot be positioned directly above a road. Most of the times they will have a prospective view, as shown in Fig. 3.2a. So we need input values to map the road with respect to the camera pixels. We need three types of information.

1. Whether a pixel covers a road area
2. How much area each pixel covers
3. How much distance each pixel covers in the direction of the camera

The presence or absence of the road allows us to apply the algorithm only on the part of the camera frame that we are interested in and hence save computational resources. The area value is used to find the percentage of the road occupied by moving objects. Finally the distance is used to compute the velocity of the vehicles. All of them can be calculated from camera resolution, aperture, focal length and height over the road. Another important thing to note here is that, as we move away from the camera, the distance represented by one pixel increases. Therefore, the distance value for each pixel is different. It is calculated once for each stationary camera and then used repeatedly to save time and computational resources.

Fig. 3.4 shows the general work-flow of the image analysis module in detail. Two configuration files containing road and camera parameters are used as inputs,

Figure 3.4: General Workflow of Image Analysis module

in addition to the image to be analyzed. This module can be instantiated, as many times as needed, once for each descriptor that is desired, so that it is possible to detect many kinds of objects at the same time.

3.2 Related Work

A lot of work has been carried out on smart cities in the last 20 years [2]. For some reviewers smart cities are still confusing [4]. Definitions range from information and communication technology (ICT) networks in city environments [3]; to various ICT attributes in a city [7]. Some relate the term with indexes like the level of education of citizens or in terms of financial security etc. [5] while others thinks about it in terms of urban living labs [39]. All of these implications are alternative schools of thought and most researchers point towards the complexity and scale of the smart city domain [6].

The monitoring of roads for security and traffic management purposes is one of the main topics in this domain. Modern smart cities measure the traffic so that they can optimize the utilization of the roads and streets by taking actions which can improve traffic flow. Video-based approaches have been researched to monitor the flow of vehicles in order to obtain rich information about vehicles on roads (speed, type of vehicle, plate number, color etc.) [15].

Vision-based traffic monitoring applications have seen many advances thanks to several research projects that were aimed at improving them. In 1986, the European automotive industry launched the PROMETHEUS European Research Program [75]. It was a pioneer project which intended to improve traffic efficiency and reduce road fatalities [72]. Later, the Defense Advanced Research Projects Agency introduced the VSAM project to create an automated video understanding technology which can be used in urban and battlefield surveillance applications of

the future [20]. Within this structural framework, a number of advanced surveillance techniques were demonstrated in an end-to-end testbed system which included tracking from moving and stationary camera platforms and real-time moving object detection as well as multi-camera and active camera control tracking techniques. The cooperative effort of these two pioneering projects remained active for about two decades. As a result, new European frameworks evolved to cover a variety of visual monitoring systems for road safety and intelligent transportation. In the early 2000s, the ADVISOR project was implemented successfully to spot abnormal user behaviors and develop a monitoring system for public transportation [47, 48, 24].

There are several methods which can extract and classify raw images of vehicles. These methods are chiefly feature-based and require hand-coding for detection and classification of specific features of each kind of vehicle. Tian et al. [70] and Buch et al. [15] surveyed some of these methods. In the fields of intelligent transportation systems and computer vision, intelligent visual surveillance plays a key role [84]. An important early task is foreground detection, which is also known as background subtraction. Many applications such as object recognition, tracking, and anomaly detection can be implemented based on foreground detection. [82] [73].

An application was proposed in the Artemis Arrowhead Project [36] that can detect patterns of pedestrians and vehicles. According to the authors, based on this information, the application can also extract a set of parameters such as the density of vehicles and people, the average time during which the elements remain stationary, the trajectories followed by the objects, etc. Subsequently, these parameters are offered as a service to external parties, such as public administrations or private companies that are interested in using the data to optimize the efficiency of existing systems (e.g., traffic control systems or streetlight management) or develop other potential applications that can take advantage of them (e.g., tourism or security).

Many existing systems, which are concerned about privacy of the citizens, employ some sort of censorship so that human or AI users are not able to see and inadvertently recognize any person in the camera footage. This can be done either in the form of a superimposed black box, which blocks out the eyes or face of the person, masking each person in each frame or blocking images of certain places altogether [11, 51, 58, 59, 61, 65]. However, this approach cannot achieve full privacy. Most of the time we do not require any sort of information related to individuals while working with applications related to computer vision. Thus, the developer should be aware of the information being collected either advertently or inadvertently and of what are the real requirements for the application.[17]

3.3 Algorithms

To extract the required information from the video stream, two image processing algorithms are applied. One is the background subtraction algorithm, while other is Lucas Kanade Algorithm for optical flow. A sample frame from one of the cameras is shown in fig 3.5. The image is split into two portions to separate the information on two road recorded in one frame. These information is passed to the algorithm in the configuration file for separate calculations.

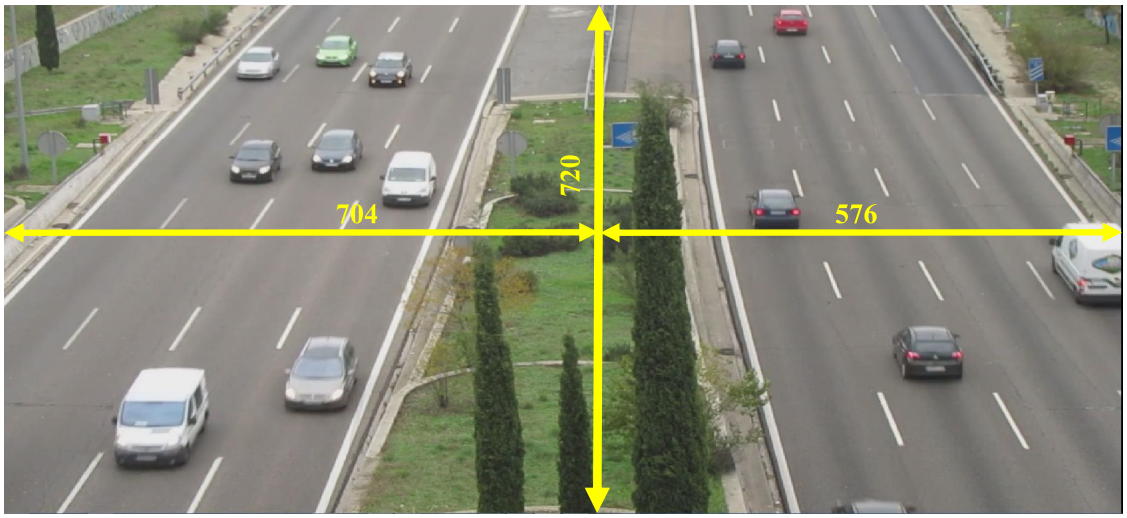


Figure 3.5: Sample Frame

3.3.1 Background Subtraction

Algorithm 1 is based on a background subtraction and object tracking method. One popular implementation was made available by Laurence Bender et al. as part of the SCENE package [63], available in the Sourceforge repository (Fig. 3.6). The algorithm performs motion detection principle by calculating the change in corresponding pixel values with respect to the reference stationary background. The portion of the road where movement is detected gives an idea about the amount of traffic. Moreover, the algorithm also constantly updates the reference background image (in case a moving object is now at rest).

Scene is an open source multiplatform computer vision framework that performs background subtraction and object tracking using algorithms based on neural networks and fuzzy classification rules. It was mainly designed as a toolkit for the rapid development of interactive art projects that explore dynamics of complex environments (for example public spaces).

Scene defines five different model implementations if OpenCL for background subtraction. They are

- Simple Gaussian
- Fuzzy Gaussian
- Mixture of Gaussian
- Adaptive Self-Organizing Map (SOM)
- Fuzzy Adaptive Self-Organizing Map (SOM)

Figure 3.6: Output of the Background Subtraction Algorithm [64]

Our chosen algorithm takes four frames (images) as input, including the reference stationary background, the frame under the consideration, the preceding frame and the succeeding frame. For each pixel, it performs a weighted difference on the corresponding pixels of three consecutive frames. If this difference is zero, it implies that there is no movement in the corresponding pixel, hence no update is needed for the total moving area or the reference background. On the other hand, non-zero values corresponds to some change in the consecutive video frames around the pixel. The value can be a positive or a negative number according to the direction of movement with respect to the camera. If the absolute of this value is larger than the threshold set for movement detection and some change is also detected in the current frame pixel w.r.t. the reference background, then the global accumulator of the moving area is updated by adding the area of the road occupied by the current pixel. If the weighted difference is less than the threshold

for N-1 frames, then the algorithm updates the reference background pixel with the current pixel. N is the minimum number of frames required to declare the pixel to be part of the stationary background. The value of N can be set according to the application.

Algorithm 1 Background Subtraction algorithm

Require: Four grayscale images $image_{-1}$, $image_0$, $image_1$ and $image_{bg}$ & Count array

Ensure: $image_{out}$, Updated $image_{bg}$ and Count array & Total Area with Movement

```
1: for j = 0 to HEIGHT − 1 do
2:   for i = 0 to WIDTH − 1 do
3:     PIX = (j ∗ WIDTH) + i
4:     lat = 0
5:     if PIX is on ROAD then
6:       center ← PIX
7:       left ← PIX − 10
8:       right ← PIX + 10
9:       lat ← Abs(sum of weighted difference of left, right and center pixels of
         all three images)
10:    end if
11:    if (lat< threshold) & (Count[PIX] ≥ N) then
12:      image_bg[center] ← image_0[center]
13:    else
14:      Count[PIX]++
15:    end if
16:    if ((image_0[center] - image_bg[center]> Background threshold) & (lat>
       threshold)) then
17:      image_out[center] ← image_0[center]
18:      Increment Area with Movement
19:    else
20:      image_out[center] ← 0
21:    end if
22:  end for
23: end for
```

As described above, the algorithm needs three consecutive frames and a reference stationary background image to distinguish between moving and stationary objects. After the computation of one set of frames, the next frame is fed to the kernel and the oldest one is removed from the set. The result is shown in Figure

3.7.

Here the static areas are detected as background and converted to black, while pixels where movements have been detected are shown as gray-scale pixels of the original frame. We also compute the portion of the road that is occupied by moving objects. In this set of frames, it is equal to $11.2m^2$ on the side where traffic is coming towards the camera, and it is $6.55m^2$ on the side where traffic is moving away from the camera.

Figure 3.7: Output of Background Subtraction

3.3.2 Lucas Kanade Algorithm

Since the background subtraction module can only find the area occupied by moving objects on the roads, another method is needed to measure the velocity of vehicles, based on the Lucas Kanade algorithm for optical flow [42]. An implementation of the Lucas Kanade's Optical Flow algorithm developed by Altera [52] in OpenCL with a 52x52 window size is shown in Fig. 3.8.

A window size of NxN means that the optical flow for one pixels is computed with respect to the neighboring N/2 pixels on each side of that pixel i.e. the pixel under consideration is in the center of a matrix of pixels having (N+1) rows and columns. For each pixel in the window, a partial derivative with respect to its horizontal (I_x) and vertical (I_y) neighbors is computed. The size of the window is a compromise between true negative and false positive change detection. Therefore it should be chosen by an expert with respect to area covered by each pixel and other parameters. In this paper we uses a 15x15 window.

A pyramidal implementation [12] is used to refine the optical flow calculation

and the iterative Lucas-Kanade Optical flow computation is used for the core calculations. For each pixel, computed partial derivatives within the window and the difference among the pixel values in the current and next frames are used to calculate the velocity of each moving object (it is zero if the area covered by the pixel is stationary). The magnitude is the speed of the object whereas the sign shows whether it moves towards the camera or away from it.

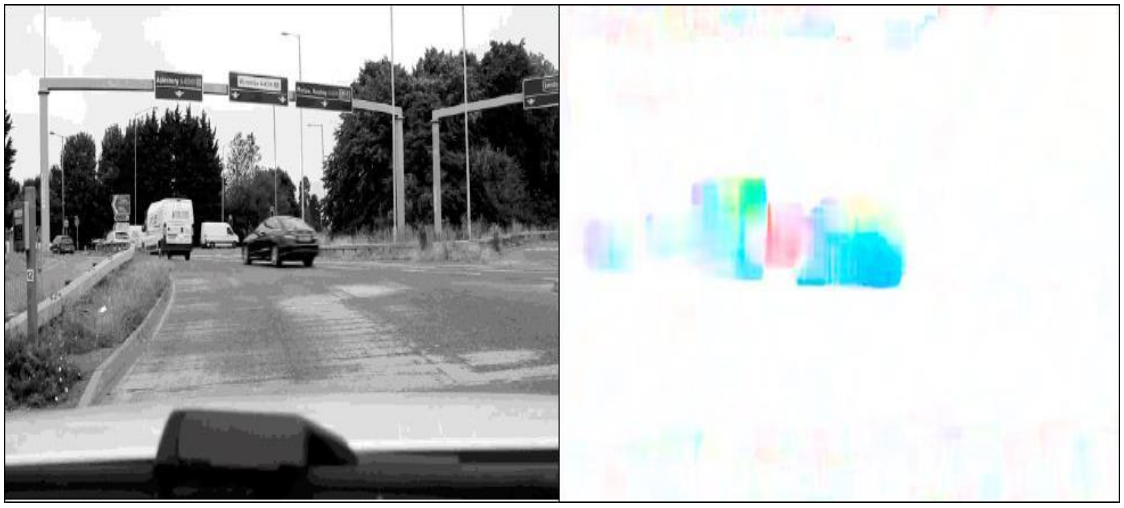


Figure 3.8: Altera's Implementation of Lucas Kanade Algorithm [52]

In our implementation of the Lucas-Kanade Algorithm (Algorithm 2), for each set of calculations, we need two consecutive image frames and a set of input parameters depending on the road conditions and camera angles. Similar to Background subtraction, each new frame replaces the older one. The optical flow is computed for all the pixels of the image (in this case for a 1280x720 resolution). Two images using 8 bits per pixel are compared with a window size of 15. Moreover, the obtained values are mapped to a single color representing both relative velocity and direction, as shown in figure 3.9. The graphical output from these images is shown in Fig. 3.10. The stationary regions are represented by white pixels, while moving objects are mapped to colors according to their speed and direction.

To calculate the average velocity of traffic with the Optical Flow algorithm one needs to know the distance between the camera and the recorded objects. In order to avoid expensive and complex solutions for a real time depth measurement, an approximation for calculating the distance corresponding to each pixel of the image is used based on static camera parameters, such as road plane inclination, camera orientation and field of view. For the current frame as reference. The average velocity coming towards the camera is about $118 km/h$ while the velocity moving away is -$67 km/h$.

Algorithm 2 Lucas-Kanade algorithm

Require: two frames of images $image_0$ and $image_1$ **and** other coefficients
Ensure: v_{opt}

for $j = 0$ to $HEIGHT - 1$ **do**
  for $i = 0$ to $WIDTH - 1$ **do**
    $G_{2\times 2} \leftarrow 0$
    $b_{2\times 1} \leftarrow 0$
    for $w_j = -w_y$ to w_y **do**
      for $w_i = -w_x$ to w_x **do**
        $center \leftarrow \text{Pos}(i + w_i, j + w_j)$
        $left \leftarrow \text{Pos}(i + w_i - 1, j + w_j)$
        $right \leftarrow \text{Pos}(i + w_i + 1, j + w_j)$
        $up \leftarrow \text{Pos}(i + w_i, j + w_j - 1)$
        $down \leftarrow \text{Pos}(i + w_i, j + w_j + 1)$
        $im^0_{\text{val}} \leftarrow image_0[center]$
        $im^1_{\text{val}} \leftarrow image_1[center]$
        $\delta I \leftarrow d(im^0_{\text{val}}, im^1_{\text{val}})$
        $im^0_{\text{left}} \leftarrow image_0[left]$
        $im^0_{\text{right}} \leftarrow image_0[right]$
        $I_x \leftarrow (im^0_{\text{right}} - im^0_{\text{left}})/2$
        $im^0_{\text{up}} \leftarrow image_0[up]$
        $im^0_{\text{down}} \leftarrow image_0[down]$
        $I_y \leftarrow (im^0_{\text{down}} - im^0_{\text{up}})/2$
        $G \leftarrow G + g_{2\times 2}(I_x, I_y)$
        $b \leftarrow b + f_{2\times 1}(\delta I, I_x, I_y)$
      end for
    end for
    $G \leftarrow \text{inverse}(G)$
    $v_{\text{opt}}[j][i] \leftarrow G \times b$
  end for
end for

We can also find the speed in any specific lane of the road, by dividing the pictures in separate lanes instead of two parts as we did in Fig. 3.5. This can be achieved, if required, by minor adjustments in the input configuration file.

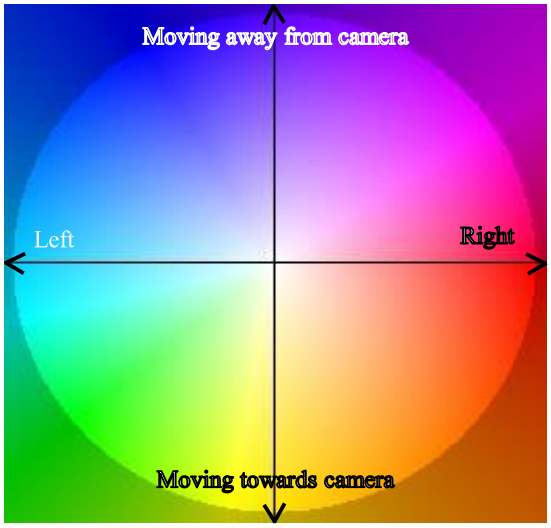


Figure 3.9: Lucas Kanade's Disparity Map

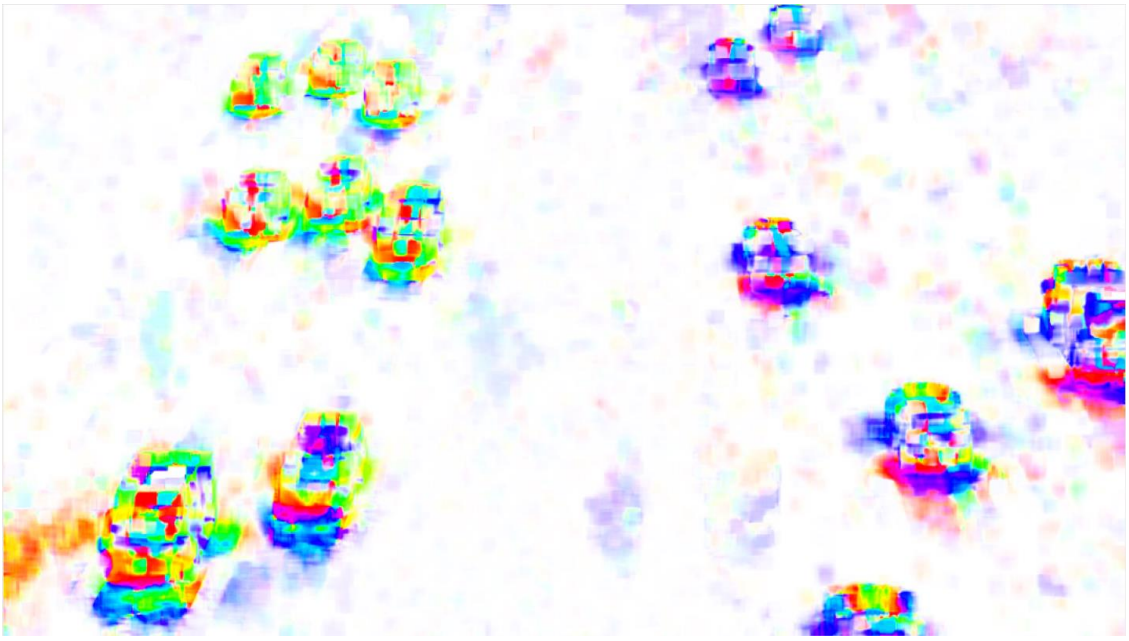


Figure 3.10: Output of Lucas Kanade Algorithm

3.3.3 Implementation Model

Two types of implementation are possible for this system on the basis of the location of computational and storage units. One is decentralized, where each camera has its own processing unit. The other is centralized, where all the processing by a set of closely situated cameras is done on one single server.

Decentralized Architecture

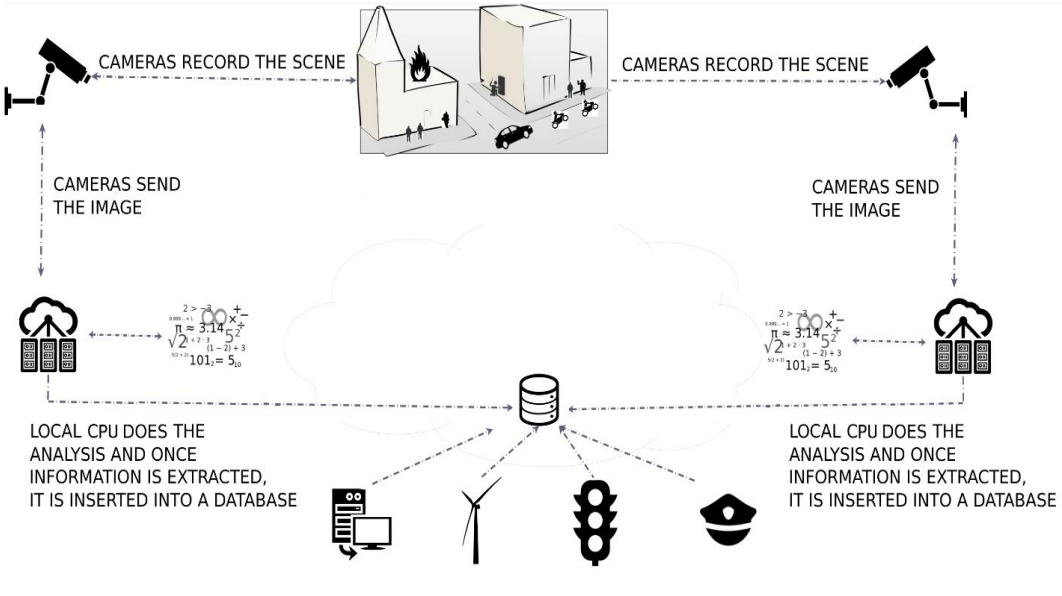


Figure 3.11: Decentralized Model

Fig. 3.11 represents the decentralized architecture version of the application. Due to the high computational requirements , a dedicated CPU would be needed for each camera installed in the monitored scenario. Once the image (which must be processed in real time) is captured, the pre-processing unit associated to that camera processes the signal for detecting the elements present in the image. Afterwards, it sends a picture with some meta-data to the central processing unit in which all of the information is processed and stored to be offered to the customers within a cloud architecture.

Centralized Architecture

On the other hand, Fig. 3.12 depicts an architecture in which one processing unit is used by a number of cameras. The idea is to combine the processing unit with the central database where all the data is offered to the customer. This means that no camera has a dedicated processing unit attached, which dramatically increases the amount of data to be processed centrally in real time.

After analyzing both options, the second alternative is considered more appropriate because of the costs of implementation, application software management, maintenance costs to resolve hardware failures, improved safety etc. In Figure 3.13, the scheme for the proposed solution is presented. A major factor for choosing a

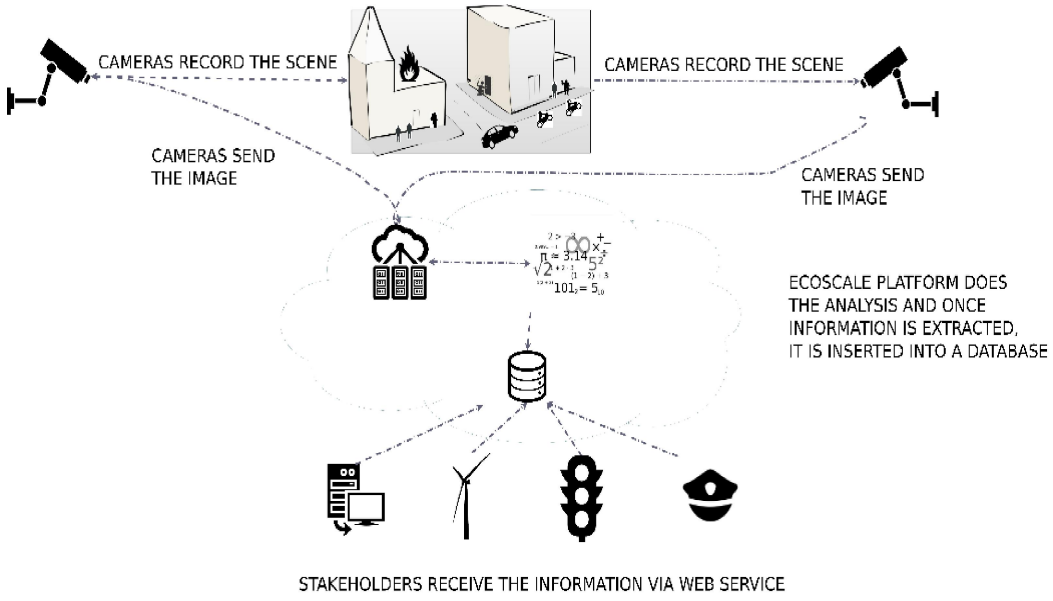


Figure 3.12: Centralized Model

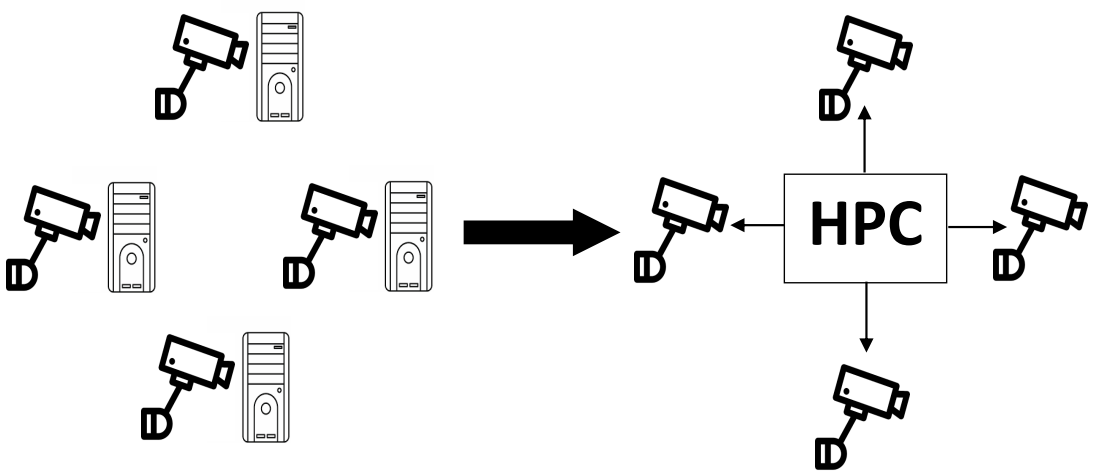


Figure 3.13: From decentralized to centralized architecture

centralized system would be the achievable energy efficiency by using latest generation FPGA devices, which are very power-efficient but too expensive to be deployed in a decentralized architecture.

3.4 Constraints

However, the use of cameras poses some disadvantages. The first major drawback is the breach of privacy. Citizens usually feel uncomfortable and insecure when their movements are being monitored and they tend to oppose any such system. To overcome this disadvantage, the end users of our application are not given the raw data. Rather they are provided with only the result of the processing of the images recorded by the cameras. This ensures both the protection of personal information and the value of data.

Note that, in our implementation of the application, the processed images or data extracted from them contain no personal information, thus we can safely say that we have achieved the objective of personal data integrity and we are not forwarding any sort of personal or privileged information to any third party.

Another difficulty in the use of such systems is the huge effort required to compute and process data by image analysis algorithms. For instance, cameras should be deployed every 50 meters or so in order to obtain a density that can provide complete information for a city. A big city with an urban area of $360km^2$ would require the use of about 100,000 active cameras. This can be supported only by extreme parallel computing techniques.

This issue is resolved by the use of centralized architecture. Moreover the computational power is provided by the ECOSCALE platform, whose architecture is discussed in Chapter 2. Therefore, we can say that the application can be implemented in real world.

3.5 Optimizations

Most of the operations carried out in image processing are pixel-based, with no or very few dependencies on other pixel output values. This provides a very good basis for a parallel implementation of image processing algorithms that work on each pixel either simultaneously or in a pipelined fashion (Fig. 3.14). In this way we can reduce the frame processing time and hence we can achieve a real time processing frequency, which is about 25fps for the target application.

Several optimization that we performed on the code to make it optimal for FPGA design are explained below.

3.5.1 Memory-related optimizations

In the context of memory-related key issues and observations, it has to be stated that the most important aspect that affects performance is the data I/O to and from the host as well as the hardware accelerators. The highest I/O bandwidth can be achieved by using streams and DMA controllers. This approach though

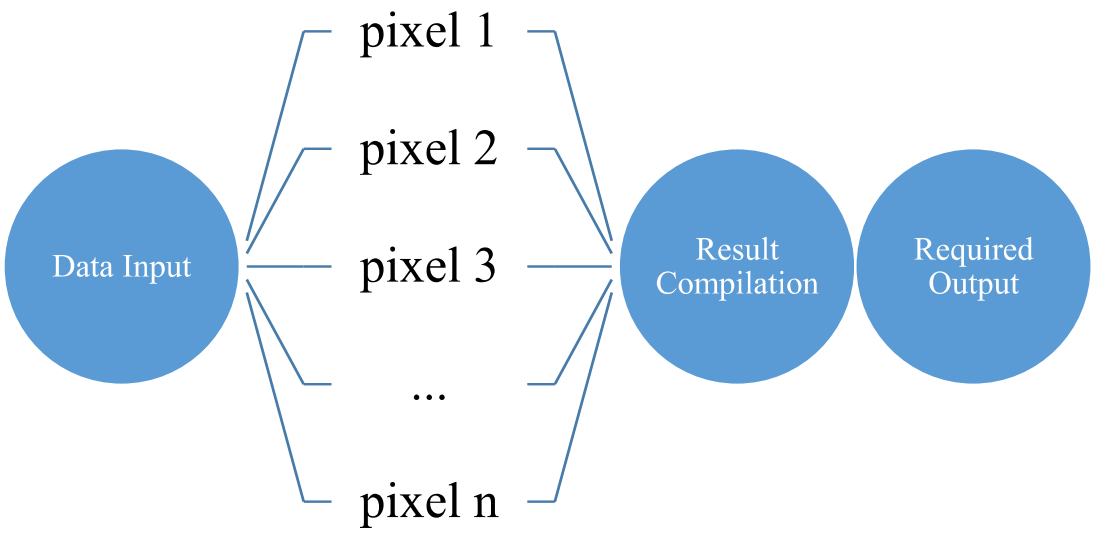


Figure 3.14: Overview of parallelism in image processing Algorithms

has one critical restriction, requiring that the data have to be stored in contiguous memory space, which is not always possible. In this application, the kernels are generated using OpenCL which only supports memory mapped interfaces. As a result streams cannot be utilized. Hence, we require other methods and coding examples that make the memory mapped interface as efficient as possible. In the ideal scenario it can achieve comparable bandwidth to the streaming case, i.e. only two times slower. These methods can be categorised as follows:

- Inferring memory bursts
- Utilizing the maximum bus width
- Eliminating unnecessary memory accesses

Inferring Memory Bursts

As mentioned above OpenCL kernels do not support streaming interfaces, they support the AXIlite and AXI master interfaces. The AXImaster interface supports memory bursts of up to 256 words on the Ultrascale+ FPGAs.

Subsequently, in order for the kernel to infer memory bursts, the code has to be written in certain ways, which the tool can identify and produce the correct interface that accesses the memory using bursts. The first way is the usage of a function defined for this purpose *async_work_group_copy()* and the second is to read the data into local or private memory, which is using BRAMs in the FPGA,

inside a *for loop*. The *async_work_group_copy()* function is defined for OpenCL language and transfer data between global and local (work group) memory only. It cannot transfer data for private memory. The maximum burst size supported is 256 words per burst. The following code segment (Fig. 3.15) describes how bursts can be inferred using a pipelined for loop.

```
//for loop

#define TMPSIZE 256

#define INPUTSIZE X

void foo(__global float input, __global  float output) {

        __private float temp[TMPSIZE]

        for (int j=0; j< INPUTSIZE/TMPSIZE; j++) {
                for (int i=0; i< TMPSIZE; i++) {
                #pragma HLS PIPELINE
                        temp[i] = input[i+ TMPSIZE*j];
                }
                ..................
                ..................
                ..................
                ..................
        }
}
```

Figure 3.15: Example for burst access of data from DRAM

Utilization of maximum Bus width

Another really important optimization is the utilization of the maximum bus width. In our case that is 128bits which is restricted by the host HP ports that are used for communication between the processing system and the programmable logic. In order for the kernel to have a 128-bit data width, the arguments must use vector data types, e.g. float4, int4 etc. This allows the most efficient utilization of the PS-PL bandwidth as it utilizes the full width of the HP ports.

Eliminating Unnecessary Memory Accesses

The previous two subsections provide I/O optimization that are platform and technology specific and always affect performance. Here, we present memory-related optimization that is application specific and concern mainly the minimization of the DDR memory accesses.

Based on the source code description of an algorithm, it is not possible for the platform to recognize whether the application accesses the same data multiple times from the DDR memory or not. Hence, depending on the application, it highly recommended to increase the usage of temporary buffers (BRAMs) or temporary registers, that store data inside the PL to be further processed, since this can significantly improve the performance as it reduces DDR accesses. For example in the case of image filter applications, line buffers can be used to store the lines or pixels required for the filter. A 3x3 filter requires 9 memory accesses per result. If the data are stored in line buffers the same operation requires only 1 memory access as the data are already stored in BRAMs.

3.5.2 Computational optimization

In this subsection we provide a summary of source code optimization that were implemented to generate an efficient hardware accelerator module that will eventually lead to a cost-effective application execution. These guidelines are principally split into the following sections.

- Loop pipeling and unrolling
- Array partitioning
- Minimization of operations using temporary registers
- Division using constants
- Reusing the calculations
- Piecewise Linear approximation

Loop pipelining and unrolling

In sequential languages like C/C++, commands in loops are executed one after the other i.e. each statement is executed in the next clock cycle than the previous one (assuming the ideal case of one instruction per clock cycle). To introduce parallelism, we have two options, either we can pipeline a loop or we can unroll it.

Unrolling means that we make copies of loop and execute the loop iterations in parallel. These copies and parallelism is controlled by unrolling factor. Pipelining

means that loop iterations overlap each other. This means that first instruction of second loop iteration starts with the second instruction of first iteration.

Both of them have their advantages and disadvantages. It reduces the execution time of the application with increase in resource utilization. Therefore, the designer should be careful for choosing the optimal option. In our application, we pieplined the inner most loop of the code in order to achieve parallelism without exploding the resource utilization. Some smaller are also unrolled where resource utilization was not too high.

In some of the cases these directives, although applied, were not able to achieve the required Initiation Interval (II). Some of these reasons are addressed below which were the cause of reduced performance. These issues were addressed in order to achieve better performance.

Array partitioning

One of the major reason for not achieving the required pipeline performance is the memory access bottleneck. If we need to fetch more than one memory location from the same BRAM, than this process can reduce the parallelism due to limited number of memory ports.

This issue can addressed by storing the same memory into different BRAMs or FFs. So, data can be fetched from different memory locations simultaneously. The partition can be complete, i.e. storing everything separately in registers. This can be expensive and sometimes impossible to synthesis or place and route for the tool. Other possible options are block and cyclic which are selected depending on the access pattern of the data. Generally, cyclic partition is preferred for sequential access while block partition for strided access.

Minimisation of operations by using temporary registers

The high level synthesis tool does not reuse results that have been already calculated and stored in registers, whereas it uses new LUTs to create new hardware, e.g. :
int k1 = a + b;
int k2 = a + b + c;

In this case, the tool will use resources to create k1, but for k2 will not use the result stored in k1, but will use new resources to recalculate a+b. In order to eliminate such unnecessary resource utilization the code should be changed to:
int k1 = a + b;
int k2 = k1 + c;

Division with constants transform to multiplication

One more optimization that can be used extensively is the avoidance of division operations when the divisor is constant. Division is the most expensive mathematical operation with respect to resources. For that reason, every single division with constant divisors should be replaced by a multiplication by inverting the divisor. For instance
double k = a/b;
with b being constant, is transformed to :
double bmult= 1.0/b;
double k = a*bmult;.

Reusing the calculations

There are some calculations which are done within one work item, can be reused in the succeeding work items (within the same work group). For example if we are working with a sliding window over an image and performing convolution with a fixed filter, then those computations can be reused in the next work item for the overlapping elements of sliding window.

The flowchart in Fig. 3.16 explains the modification. Assuming a window size of 15x15, the left side shows the conventional way of performing all the computations for a pixel within two nested loops with 15 iterations each. It requires 225 iterations to compute the final value of each pixel and there is no reuse of the computed values.

The flow chart on the right hand side depicts a reuse-oriented modification. In the first loop, each work item computes the values for the columns of the window under consideration and stores them in a local buffer. The second loop uses those values over all the columns in the window to compute the final value of the pixel. In this way, not only we reduced the total number of iterations from 225 to 15, but also we can reuse the values computed for 14 columns, which are needed in the next sliding window. This optimization will also result in better pipelining of the individual loops.

Piecewise Linear approximation

Trigonometric computations, especially involving floating point numbers, are expensive on FPGAs. Sometimes it is possible to use a piecewise linear mapping approximation instead of accurate trigonometric computations. In our case, the piecewise linear approximations get a performance improvement of around 15x.

Our specific application requires generating an image from the output data, mostly for debugging purposes. In this case, instead of using the tangent function to map it using a colour coded disparity map, a piecewise linear approximation results in acceptable results with remarkable increase in performance of the application. The first code snippet in Fig. 3.17a one uses the inverse tangent function to map the

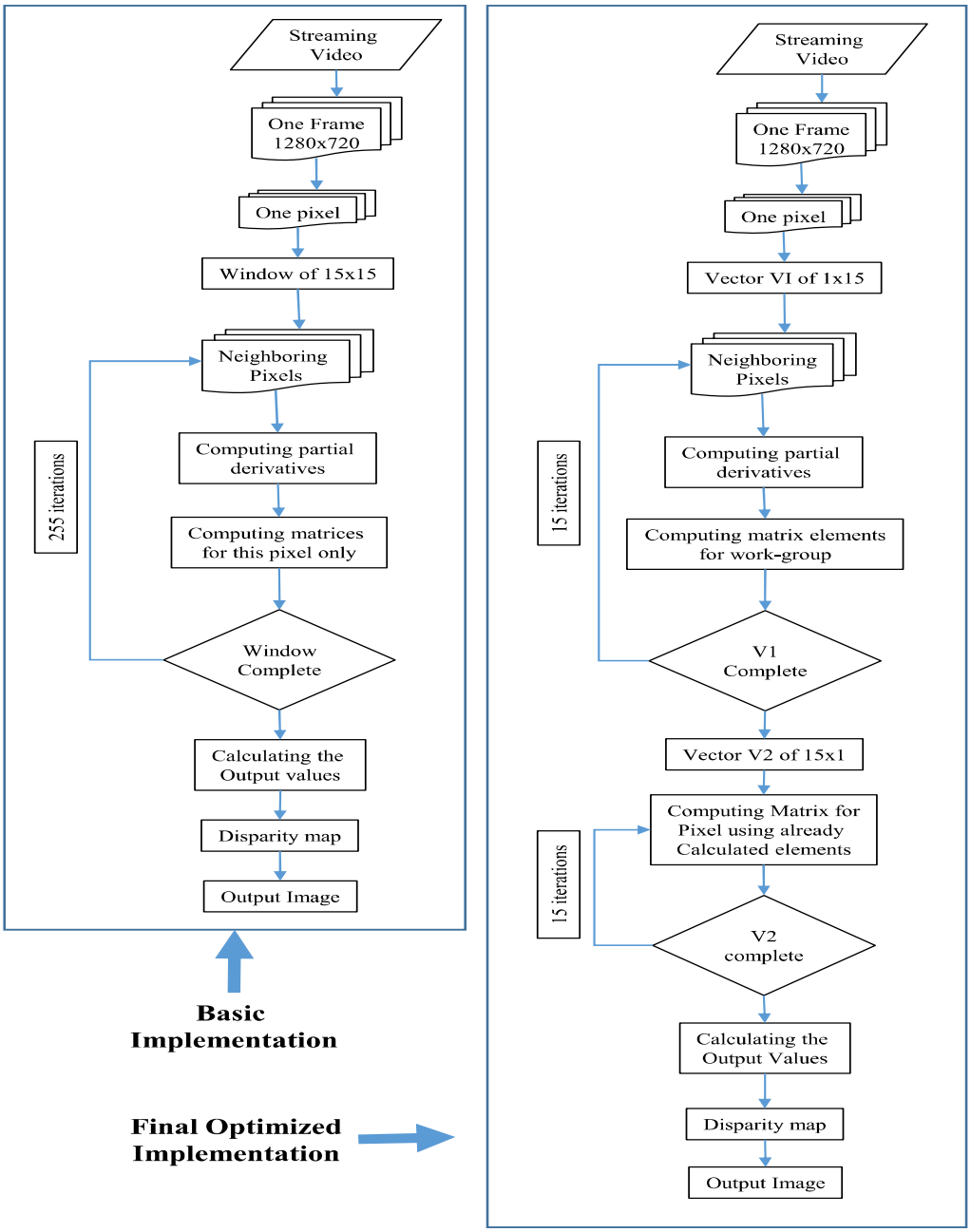


Figure 3.16: Flowchart for KNP

image to a set of 55 different colour values, while the second one uses a piecewise linear approximation to generate the image. There is not much of a difference between two outputs as shown in Fig. 3.17b & Fig. 3.17c, especially when the goal

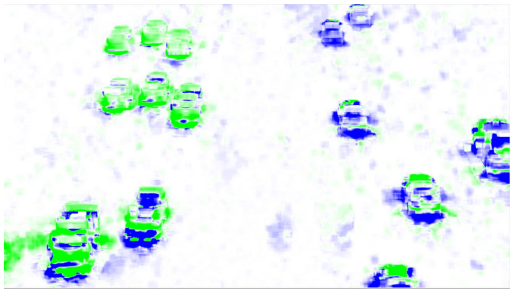
```
float rad = sqrt(fx * fx + fy * fy);
float a = atan2(-fy, -fx) / (float)M_PI;
float fk = (a + 1.0f) / 2.0f * (l_ncols - 1);
int k0 = (int)fk;

colS col;
col.s0 = *(l_colorwheel + k0 * 3 + 0);
col.s1 = *(l_colorwheel + k0 * 3 + 1);
col.s2 = *(l_colorwheel + k0 * 3 + 2);
if (rad > 1) rad = 1; // max at 1
pix[0] = (255 - rad * (255 - col.s0));
pix[1] = (col.s1 = 255 - rad * (255 - col.s1));
pix[2] =  (255 - rad * (255 - col.s2));
```

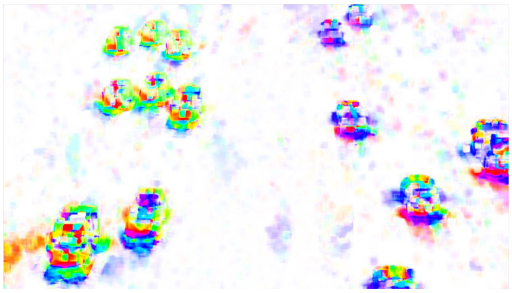
```
#define VMIN (-1.0f)
#define VMAX (1.0f)
#define coef_a (0.5f*(VMIN+VMAX))
#define coef_b (510.0f/(VMAX-VMIN))
uchar limit_uchar(float x)
{
        if (x<255.1f)
        {
                if (x>0) return (uchar) x;
                else return (uchar) 0;
        }
        else return (uchar) 255;
}
/*
Generates a color as a function of v: if v<vmed, assign red,
if v>vmed assign green
At the middle color=white
*/

void assigncolor_white(float v,uchar color[3])
{
        color[0]=limit_uchar(255-fabs(v-coef_a)*coef_b);
        color[1]=limit_uchar((v-VMIN)*coef_b);
        color[2]=limit_uchar((VMAX-v)*coef_b);
}
```

(a) Piecewise linear substitution for tangent function



(b) Using Piecewise linear approximation



(c) Using trigonometric computation

Figure 3.17: Difference between two outputs for Lucas Kanade Algorithm

is just debug the algorithm. On the other hand it have a huge impact and cost saving for running it on FPGAs.

3.6 Implementations

The application was first designed for CPU implemented to verify the functional correctness. After that it was optimized for GPUs. Once it was verified and was able to achieve the required performance, then it was optimized for FPGAs to achieve better power and energy consumption. The CPU implementation was carried out to get the best possible solution available.

3.6.1 CPU

As stated above, the CPU implementation was just to verify that the algorithms producing the desired outputs. To verify the functionality, consecutive image frames were applied as input and the corresponding outputs were verified with the ground reality. Moreover, the execution time and energy consumption was noted for future comparison with the results of accelerators.

The CPU that we are considering is an Intel Xeon E3-1241(v3) with a clock frequency of 3.5GHz and maximum power consumption of 80 Watts. The background subtraction algorithm takes **47.68 *msec*** to process one frame while Lucas Kanade algorithm takes **5925.78 *msec*** per frame. This also confirms that CPU cannot be used to perform real time video processing for this quality of images.

3.6.2 GPU

After the functional verification of the application, parallel implementation is done with the help of GPUs. For GPU implementation, application was coded in OpenCL language (explained in section 2.3). The purpose of this implementation is to achieve the required throughput with minimum possible cost. In this implementation, we are considering an NVIDIA GeForce GTX960 GPU. It has 2GB of global memory and bandwidth of 112 GB/s with a maximum power consumption of 120 Watt.

The major optimization done for the implementation on GPU are enlisted.

Lucas Kanade Algorithm

- All variables, functions and operations of double precision have been changed to float
- The pyramid array size has been reduced by half every level
- Alpha channel has been removed in all the functions
- ffmpeg and SDL libraries have been used instead of OpenCV to decode video and generate image output
- A mask has been added to remove the pixels outside the road
- Non-blocking calls to *clEnqueueWriteBuffer()* have been used

Background Subtraction Algorithm

- All variables, functions and operations of double precision have been changed to float

- Background calculation has been modified in order to not to be affected by car light reflections and shadows
- Background adaptation to large light differences between frames due to camera auto shutter
- A mask has been added to remove the pixels outside the road

The optimized GPU code was able to perform the required computations in time for both the algorithms (separately). The GPU device time for background subtraction algorithm is **28.16 *msec*** per frame while for Lucas Kanade algorithm it is **42.68 *msec***. The power comsumption and its comparision with other devices is presented in section 3.6.4.

3.6.3 FPGA

After testing the basic functionality of the algorithms, we optimized them in order to get the maximum efficiency with a minimum use of resources in the smallest amount of computational time. Performance analysis was carried out using RTL simulation on a virtual board including a Virtex 7 FPGA from Xilinx and then on real hardware, using the Amazon Web Services (AWS) Elastic Compute Cloud (Amazon EC2). The available resources on these boards are shown in Table 3.1. Note that in order to complete RTL simulations (for Virtex 7) in a reasonable amount of time, we used an image resolution of 1280x4 and we extrapolated the simulation results to the real image size. On AWS, on the other hand, the complete frame was used to verify the results. For high level synthesis, we used SDAccel v2016.4, 2017.1 and 2017.4 from Xilinx.

Moreover, simulations were carried out for a single compute unit and then a suitable number of compute units that could fit on the FPGA were used for each algorithm. In contrast to a CPU or GPU, an FPGA does not have a fixed architecture but the HLS tool generates a custom computation and memory architecture from each application. The term "*compute unit*" (CU) refers to a specialized hardware architecture (processing core) for a given application. The designer can use multiple parallel CUs (within the available resources) to boost the performance of each application.

Basic GPU optimized code

The first implementation is done on the code, which was optimized for GPU. There were two main goals from this implementation.

- To find the areas where performance improvement is required.
- To compare the design efficiency and design to cost ratio with all the optimization applied later.

Table 3.1: Target FPGAs and boards

Target Device Name	**ADM-PCIE-7V3:1ddr:3.0**	**AWS-F1:4ddr-xpr-2pr:4.0**
FPGA Part (Xilinx)	**Virtex-7** XC7VX690T-2	**Virtex UltraScale+** xcvu9p-2-i
Clock Frequency	200MHz	250MHz
Memory Bandwidth	9.6GB/s	11.25GB/s
BRAMs	2940	4320
URAMs	-	960
DSPs	3600	6840
FFs	866400	2364480
LUTs	433200	1182240

The results of the implementation are shown in the table 3.2. We can see that the GPU optimized version of code on FPGA is even worse than CPU. Therefore we will apply the optimization to that to get the required 25 Hz performance for both of the algorithms. The first major bottleneck identified in this version is the global memory access.

Table 3.2: Kernel Execution time and Resource Utilization for Basic Design

Algorithm	**Time** *(msec)* *per frame*	**Resource Utilization**			
		BRAM	**DSP**	**FF**	**LUT**
Background Subtraction	7313.112	5	3	14447	35019
Lucas Kanade	44209.98	31	56	21367	37080

Introduction of line buffer

Lucas Kanade have 6 accesses to global memory in the inner most loop and more than 1250 accesses by each work item to the global memory. Similarly in case of Background subtraction, we have more than 15 memory accesses per work item. This posses a huge latency in design as we discussed earlier these accesses are expensive. This issue can be resolved by using burst transfers (explained in section 3.5.1). In this case we had to use a line buffer big enough to copy data required by more than a single work item, mainly because of underlying two reasons:

- All of the accesses by each work item are not sequential, but are in window pattern and for burst we need sequential accesses.
- There is a lot of data reuse among neighbouring work items as the data needed by the algorithm is in the form of sliding window

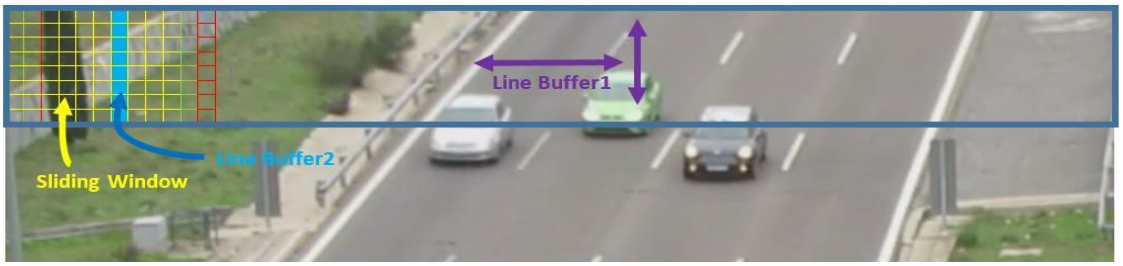


Figure 3.18: Line Buffers for Lucas Kanade

Therefore, we used a line buffer that will burst read all the data required by the work group into the buffer using *async_work_group_copy()* function. This results, as shown in Table 3.3, in reduction of execution time by 5 times for Background subtraction and more than 3 times for Lucas Kanade. The concept of line buffer is further explained in figure 3.18. Here the "Line Buffer 1" is used for saving a copy of global data. "Line Buffer 2" is used for calculation reuse and its explained later.

Using Temporary Buffers and Reusing the calculations

After minimizing the issue of global data transfer, next thing to optimize is loops and arrays. Loops need to be pipelined or unrolled to achieve parallelism while arrays need to be partitioned to facilitate that parallelism.

Moreover, there are certain computations which are done by each work item over the same data, especially in the case of sliding windows. As explained above in section 3.5, these calculations should be reused to increase the design throughput. Similarly if the value of a certain memory is amended more than once in a single work item, then it should be stored in temporary buffers until the final value is determined and then written back.

Table 3.3: Kernel Execution time and Resource Utilization for Design with Line Buffer

Algorithm	**Time** *(msec)* *per frame*	**Resource Utilization**			
		BRAM	**DSP**	**FF**	**LUT**
Background Subtraction	1467.108	22	5	10979	31700
Lucas Kanade	14883.87	122	51	18410	24613

Local buffers (within work groups) and temporary registers (within work items) have been used for Lucas Kanade while for Background subtraction only temporary registers are required. The performance increase of about 4 times for Lucas Kanade while more than 14 times for background subtraction is achieved in this case as shown in table 3.4.

Table 3.4: Kernel Execution time and Resource Utilization for Design with calculation reuse

Algorithm	**Time** *(msec)* *per frame*	**Resource Utilization**			
		BRAM	**DSP**	**FF**	**LUT**
Background Subtraction	103.81	24	5	9165	15978
Lucas Kanade	3751.2	182	52	51025	100777

Applying piece-wise linear approximation

In implementation of Lucas Kanade algorithm, there was a floating point trigonometric computation which was not only very expensive, but also causing a rise in

Initiation Interval (II) of the loop, as it requires more than one clock cycle to finish. As explain in section 3.5.2, we replaces it to get the better performance of the design as the image was not the principle output of the algorithm. It provides us with a performance increase of 18 times as shown in Table 3.5.

Table 3.5: Kernel Execution time and Resource Utilization for Design with piecewise linear approximation

Algorithm	**Time** *(msec)* *per frame*	**Resource Utilization**			
		BRAM	**DSP**	**FF**	**LUT**
Lucas Kanade	207.313	178	175	35683	36072

Final Implementation

The best time that we achieved using Hardware emulation was 103 *msec* per frame, hence not sufficient to achieve 25 fps. For this purpose we need to use at least 3 parallel compute units, which multiplies all the resources by a factor of 3 as shown in Table 3.6. This still uses only about 12% of the resources of a Virtex 7 FPGA, which can thus processes frames from 5 cameras. The results obtained from AWS EC2 board shows an increase in performance which was expected as Ultrascale+ is a newer generation FPGA than Virtex 7.

In order to satisfy real-time requirements, we have to use 6 Compute Units for the core calculations of the Lucas Kanade Algorithm. As we witnessed from background subtraction as well, the results obtained from AWS EC2 for the Lucas Kanade algorithm are very comparable to the hardware emulation results. In both cases performance improved and the amount of available resources increase significantly on a Virtex Ultrascale+ with respect to the Virtex 7. Hence we were able to feed the data from 4 cameras in real-time to the EC2 board.

One more thing, the accuracy of the application is a trade off with resources. In certain conditions, if we want to process the video at 10 Hz instead of 25, we can reduce the number of compute units and thus can save the resource. In this way, each FPGA can cater 2.5 times more cameras. This kind of implementation can be useful in areas with reduced traffic speed.

Summing up all the results discussed above, we achieved our goal of real time calculation of the portion of the road that is used by traffic and of average vehicular velocity. Moreover, Table 3.6 shows that we have not exceeded our resource

utilization limit, while performing the full processing of the data from one camera on a relatively old Virtex 7 FPGA. The results of actual Hardware implementation on the Amazon EC2 cloud platform are shown in Table 3.7.

Table 3.6: Total Resource Utilization for Virtex 7

Algorithm	**Compute Units (CU)**	**Total Resources Utilized**			
		BRAM	**DSP**	**FF**	**LUT**
Background Subtraction	3	72	15	27495	47934
Lucas Kanade	6	1068	1050	214098	216432
Total	**9**	**1140**	**1065**	**241593**	**264366**
Available	-	2940	3600	866400	433200
% Utilization	**-**	**38.77%**	**29.58%**	**27.88%**	**61.02%**

3.6.4 Performance and energy comparison

The final aspect to consider is what advantage we have achieved in terms of power and energy consumption (per computation) with respect to GPUs and CPUs. The power consumption for the FPGAs was estimated using the Xilinx Power Estimator (XPE) tool while for the GPU it was measured using NVIDIA System Management Interface (nvidia-smi).

As we can see from Table 3.8 and Table 3.9 the FPGA is much more energy efficient as compared to both CPU and GPU. Moreover, the computation of Lucas Kanade is not possible in real time using only a single CPU, as it takes around 6 seconds to process each frame. As we can see both, performance and energy consumption, are much better than on a CPU and energy consumption is much better than on a GPU.

Table 3.7: Total Resource Utilization for UltraScale+ (AWS-EC2)

Algorithm	**Compute Units (CU)**	**Total Resources Utilized**			
		BRAM	**DSP**	**FF**	**LUT**
Background Subtraction	3	65	15	18723	17859
Lucas Kanade	6	812	246	176970	168280
Total	**13**	**877**	**261**	**195693**	**186139**
Available	-	4320	6840	2364480	1182240
% Utilization	**-**	**20.30%**	**3.81%**	**8.27%**	**15.74%**

Table 3.8: Power Consumption per Frame for Background Subtraction

Parameters	**FPGA**		**GPU**	**CPU**
	Ultrascale+	**Virtex 7**		
Device Time ($msec$)	27.80	34.6	28.16	47.68
Device Power (W)	4.55	2.760	26	10
Energy (mJ)	126.49	95.496	732.16	476.8

Table 3.9: Power Consumption per Frame for Lucas Kanade Algorithm

Parameters	**FPGA**		**GPU**	**CPU**
	Ultrascale+	**Virtex 7**		
Device Time ($msec$)	37.31	36.34	42.68	5925.78
Device Power (W)	8.0	8.385	75	10
Energy (mJ)	298.48	304.7	3201	59257.8

Chapter 4

Cache Architecture and Tuning

The efficiency of any system is determined by the slowest element in that system. Currently, the main bottleneck in efficiency of modern systems is memory. The secondary or even primary memory in modern day systems are very slow as compared to the processing speed these systems provide. Therefore, the need of a fast memory accusation system is evident. That is where cache comes into play. Bell et al. [10] defines cache as: " A cache memory is a fast buffer memory between the processor and the primary memory." The cache is a temporary storage where the data is stored for a short time and a copy of data is stored in the main memory.

The hardware caches are expensive, so typically they are of smaller size as compared to primary and secondary memories. A comparison of different types of hardware memories is shown in figure 4.1. Nevertheless, in computing, caches have a lot of impact. The data needed for computing applications is mostly near to the reference. There are two types of data locality. *Temporal locality* is when the same data is re-requested in negligible time, and *spatial locality*, where the other (requested) data is stored physically close to data that has already been requested.

Two main terminologies w.r.t cache functioning are cache hit and cache miss. A *cache hit* means that the cache memory have the data which is required by the program, while a *cache miss* occurs when it does not have it. In case of cache hits, data is provided to the program from cache which is way faster than physical memory or RAM, or recomputing the same calculation. This implies that cache hits can improve the performance of a program significantly. Part of the work described in this chapter has been previously published in "Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis" [44].

Caches have been used for a long time in the domain of general-purpose CPUs. However, in that case a *single cache* is used for all the data that the processor accesses in the main memory (at most separate caches are used for code and data). This means that access conflicts between different variables (or sections of arrays) in the source code may limit the cache performance, unless sophisticated multi-way

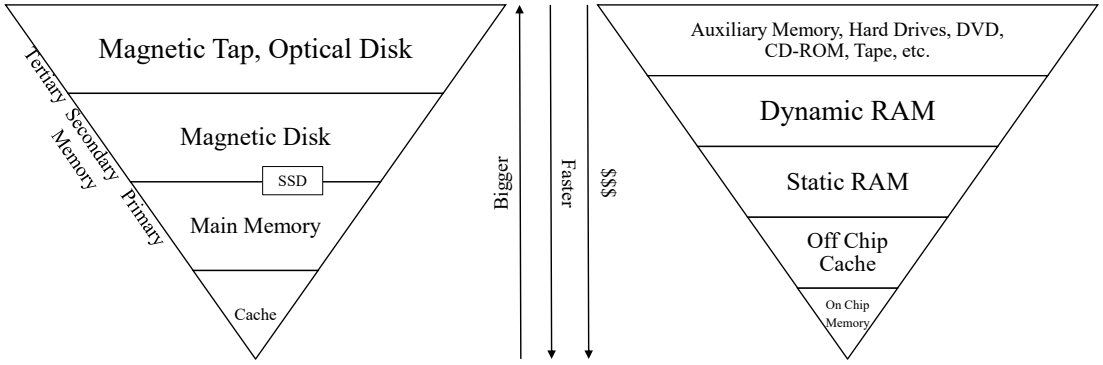


Figure 4.1: A comparison of hardware memories

or even fully-associative architectures are used. Even in that case, the "hot cache" phenomenon [21] hampers several common algorithms.

In [44] we specialize caches for HLS in several directions, which advocate the use of *a separate cache for each source array that is mapped to DRAM*, to minimize the conflicts and to enable the efficient use of direct-mapped caches

In this work, we present the design automation support for static or simulation-based address sequence analysis to identify the best cache architecture for a given application. In first part of this work, analysis is done by a heuristic algorithm while later we also designed a tool, PEDAL(Pattern Evinced Determination of Appropriate Layout). Using these techniques, we can accelerate the design of optimal caches. In this chapter, first we explained the architecture of cache designed in [44] and later we provide the analysis of different cache layouts and tools to find the optimal configuration.

4.1 Related Work

Modern CPUs generally include up to three levels of cache in order to reduce both data access time and energy. As the level increases, both latency and cache size (hence access power and energy) increase. These caches implement different access, replacement and coherency strategies to achieve the best *average performance* for all kinds of algorithms. Research on improving general-purpose caches is abundant. To cite just a few, Jouppi in [37] introduced an improvement to direct-mapped caches using a small fully-associative cache, the so-called victim cache or miss cache. In [57], Qureshi et al. presented a V-way (variable way) cache to reduce the miss conflicts existing in traditional C-way (constant way) set-associative caches. The set-balancing cache [62] and the adaptive hybrid cache [21] were introduced for similar reasons, targeting unbalanced accesses to main memory. For multi-processor

systems, Matthew et al. [45] designed configurable L1 caches for the MicroBlaze soft processor implemented on Xilinx FPGAs and achieved up to 41% speedup by using a 32KiB 4-way cache with LRU replacement. In the same setting, Kalokerinos et al. [38] presented an integrated network interface and cache controller, significantly improving hardware utilization.

Latency of memory-intensive applications is particularly significant in FPGAs due to off-chip memory bandwidth limitations. Many researchers addressed this area by exploiting the highly configurable on-chip memory architecture. For example, Cheng et al. [18] developed a trace analysis method to detect relations among all memory accesses. Performance was greatly improved by caching independent data in separate local memories. Adler et al. [1] used BRAMs as statically-managed scratchpads rather than dynamically-managed caches, and described a management system for different levels of local storage. Choi et al. [19] implemented a multi-ported cache based on the so-called live-value table, aimed at a system architecture where both the host processor and multiple accelerators are on the same chip. In their approach, both the processor and the accelerators access the same off-chip memory via a single custom multi-port cache, which of course may become a performance bottleneck. Putnam et al. [56] provided a cache-based solution to simultaneously increase performance and reduce power consumption, since external DRAM accesses require much higher power than on-chip SRAM. In this design methodology, the CHiMPS HLS tool first compiles the high-level code (written in C) to an intermediate representation and then the caches are optimized according to the memory access patterns. Similarly, Winterstein [77] also used the LLVM intermediate language to maximize the utilization of BRAMs to accelerate a specific algorithm (tree reflection).

Our approach is inspired by some of these works, in particular to reduce access conflicts by *using a separate cache, possibly with a different architecture, for each source code array* mapped to external DRAM.

The second part in this work is to determine the optimal cache layout for each array automatically. Much of the related work in this regard, emphasizes the design of specialized buffers or caches to move data on and off the chip. There is not much work on automatically tuning the layout of caches. Either the proposed caches are not tuneable, or tuning must be done manually, which requires significant effort and development time. Some work in this regard includes [76, 56, 77]. Putnam et al. [56] first generated the intermediate representation files using the CHiMPS HLS tool, whereas Winterstein et al. [77] used LLVM intermediate files to get the memory access traces. We do not focus on memory access trace collection, but rather on how to use these traces to optimize the cache architecture. Wingbermuehle et al. [76] first synthesized the hardware kernels to estimate design resource occupation, then used the remaining memory resources to copy data on chip.

4.2 Architecture

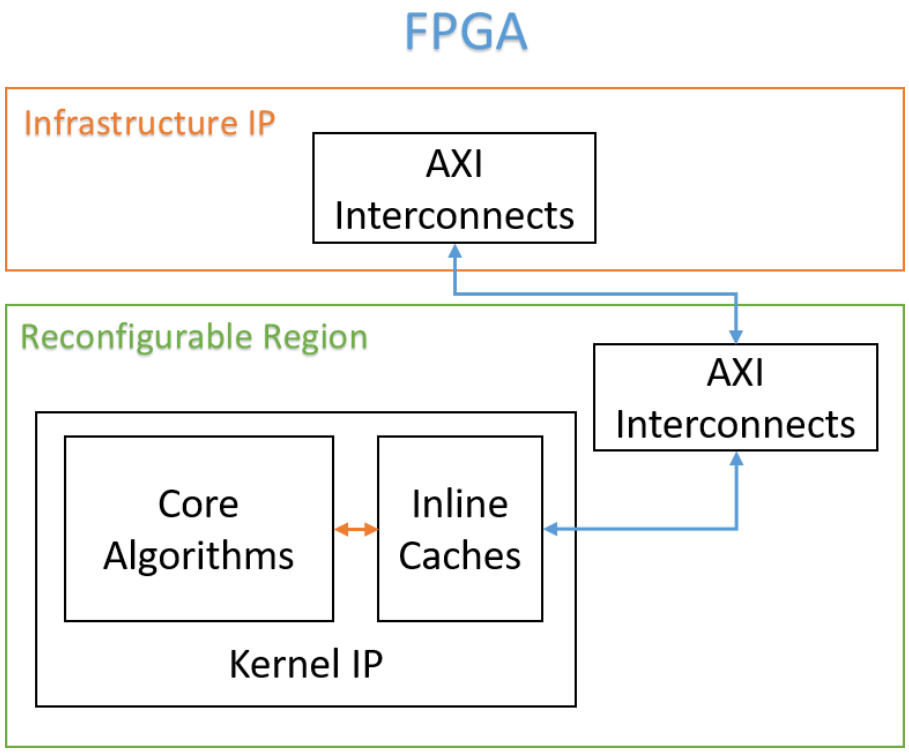


Figure 4.2: Inline cache

As shown in Figure 4.2, our caches are directly "inlined" in the algorithms to be accelerated. In this way, *the "golden" code that has been functionally verified by SW emulation does not need to be changed for high-performance implementation.* Only the top-level module interface (which is typically much smaller and simpler than its often intricate algorithmic code) requires some small changes, as illustrated below. In the resulting RTL, the caches are directly synthesized as part of the kernel IP.

Since the HLS tools that we currently use for synthesis do not support classes or templates in OpenCL kernel code, all our examples below are based on the C++ language. However, this is only to ease prototyping our flow. The same mechanism could be implemented also in OpenCL by slightly modifying the OpenCL HLS front-end.

As mentioned above, the design has to be modified only slightly in order to insert the inline caches in the interface of the original kernel. Further changes to the flow will be needed to analyze the array access patterns and to optimize the cache architectures. Automation of these new steps is left to future work. In this paper we perform this task by hand.

As shown in Figure 4.3, some analysis of the external memory access traces is

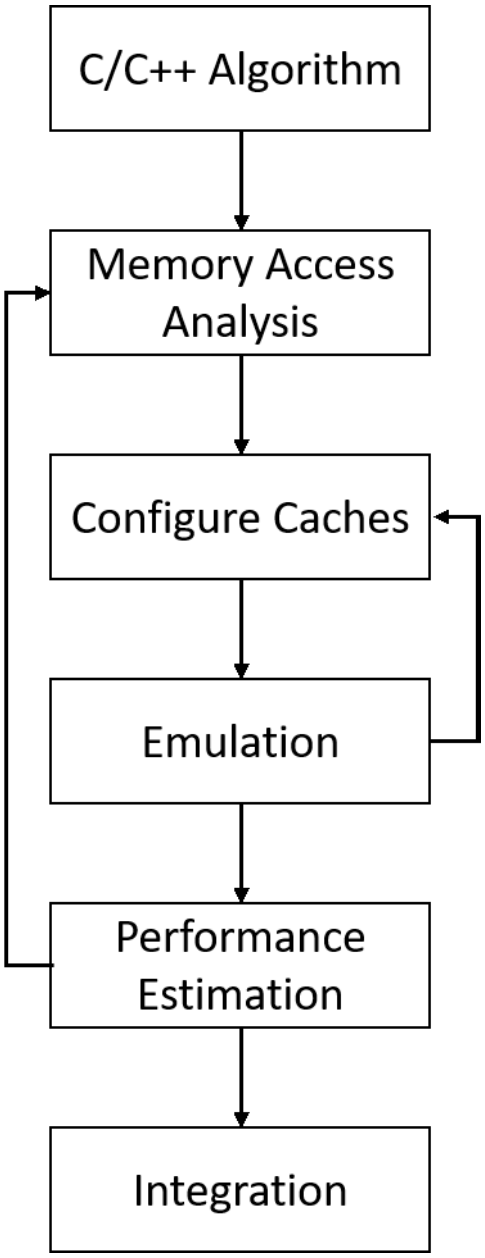


Figure 4.3: Design flow with caches

necessary to find the best cache parameters to maximize the reuse with an acceptable area cost (we will describe this in more detail in section 4.3). Note that this access analysis is needed only for arrays mapped to external ("global" in OpenCL terminology) memory, and not for the local arrays or scalars. This only requires the designers to make a few modification to the top-level function interface to replace the original data types of the global array variables with a template cache data

type.

We propose and describe several kinds of inline caches, e.g., direct-mapped and set associative, selected based on the memory-trace pattern of the applications to be optimized. Remember that in our work a separate cache is implemented for each array mapped to global memory. This means that *performance is largely independent of the global memory addresses at which each array is allocated*, and that there are no conflicts between different arrays.

4.2.1 Direct-Mapped Cache

As its name indicates, each element of each array in the external memory has a corresponding fixed position in the cache, according to a fixed bit field of the address. The line bits in the middle of the address determine to which line in the cache it is mapped, while the word bits define the position within the cache line. The tag bits are used to check whether a given address is contained in the corresponding line of the cache ("cache hit") or not ("cache miss"). In the latter case, the cache fetches the correct data from external memory and updates the corresponding cache line and tag.

Each cache line is read with a single AXI bus access, possibly using a burst (depending on the line and data bus bit widths), and stored into the cache. The write policy for the caches that we implemented is write-back, i.e., only the cache is updated initially, while the external memory is updated only when the cache needs to be flushed, either due to a write miss or due to the completion of the accelerator execution. As mentioned above, in this work we assume an execution model similar to that of OpenCL, in which global arrays cannot be read and written at the same time by the same HW-accelerated function (kernel). This avoids all kinds of coherency issues for our caches, and typically enables them to be read-only or write-only. As usual, we keep valid and dirty bits for each cache line, to indicate if it contains valid data from memory or data that needs to be written back to memory.

In this research, the direct-mapped cache was designed in C++ by using a template class. The template arguments, as mentioned above, define the type of one element of the cache and of the corresponding off-chip memory global array, the line size and the word size. The constructor initializes the base address of the corresponding off-chip memory array (typically the value of a pointer argument of the OpenCL kernel or C++ top-level function) and other variables, like the valid and dirty bits. In HLS, the constructor is typically executed as part of the reset sequence of the HW block. A C++ namespace is used to choose among a read-only, write-only or read-write cache.

In the algorithmic code to be implemented via HLS, the external memory is usually accessed by using the `operator[]` or the `operator*` on a pointer passed from the interface. Hence, we overloaded the `operator[]` for the cache type, for uses on

both the left hand side (write) and the right hand side (read) of an assignment[1]. This allows us to change only the interface of the function to be synthesized, not its code, thus dramatically reducing the design time and the likelihood of coding errors.

The interface to external memory can be defined simply by instantiating the cache type, with the appropriate template parameters, instead of every source array that is mapped to off-chip DRAM. The constructor and destructor that we created for the cache types take care of all the bookkeeping, from initializing the cache as empty (resetting all valid and dirty bits), to flushing an output cache and printing the statistics in a simulation context, when the accelerator completes its operation.

Note that since the cache access functions (for reading and writing) are inlined into the high level kernel code, the synthesized kernel takes care of both executing the computation using the cached data, and reading/writing data from/to the main memory in case of misses. As we mentioned above, this somewhat reduces the achievable performance, but it dramatically simplifies the design flow and is consistent with OpenCL philosophy, where the work items themselves take care of moving the data from global to local memory.

In order to achieve the best performance, the data width of the AXI interfaces that are used to transfer a line to and from external DRAM should have the same size as a cache line, so that a read or write can be completed in one clock cycle (plus global memory latency in case of reads, of course). If the line length is larger than the global memory read size, then burst accesses will automatically be used by our design. This is one of the key advantages that the designer gets for free by using our caches.

Algorithm 3 and Algorithm 4 demonstrate how a cache reads or writes an address of global memory. The pair of variables *request* and *hit* are used as performance counters to enable cache parameter tuning also when an FPGA is used as a rapid prototyping platform, and can be accessed via FPGA-provided debugging mechanisms (e.g., via JTAG). The *valid* and *dirty* arrays have Boolean elements. The *tags* array contain unsigned integers of the appropriate length. The *array* array is used to store all the lines of data in the cache.

The two algorithms share a similar structure. Lines 1-4 handle cache hits. The address is split into three pieces, namely *tag*, *line* and *word*, then the value (or values, for the set-associative case) stored in *tags* is compared with the *tag* part of the address. If it is a hit, the following operation is the read from (or write to) *array*, on line 16. In both cases, the actual location of the data within the line depends on the value of *word*. If it is not a hit, then a new read from the external memory is necessary (after writing back the dirty line in case of a write or

[1]We managed to overload differently the read and write accesses to call a different cache access function, by exploiting an inner class as an agent [83].

Algorithm 3 Read data from direct-mapped cache

Require: 32-bit *addr* **and** *Cache* with a pointer *ptr_mem* to external memory
Ensure: $data = Cache[addr]$

```
1:  tag, line, word ← addr
2:  request ← request + 1
3:  if tag = Cache.tags[line] and Cache.valid[line] then
4:      hit ← hit + 1
5:  else
6:      if Cache.dirty[line] then
7:          location ← Cache.tags[line], line
8:          ptr_mem[location] ← Cache.array[line]
9:          Cache.dirty[line] ← false
10:     end if
11:     loc ← addr >> LINE_BITS
12:     Cache.array[line] ← ptr_mem[loc]
13: end if
14: Cache.tags[line] ← tag
15: Cache.valid[line] ← true
16: return  data ← Cache.array[line].slice(word)
```

read/write cache).

In many algorithms, and in particular in the most massively parallel cases written in languages such as OpenCl, the uses of each array argument of a kernel are either read-only or write-only. Hence, we designed a special cache for these read-only and write-only memory accesses in order to speed up the synthesis, reduce the cost, and improve the performance. For instance, a read-only cache does not need to check if a line is dirty. Algorithm 3 and Algorithm 4 show the get() and set() functions for this case.

4.2.2 Set-Associative Cache

in some algorithms (e.g., sorting, FFT), *data read by successive external memory accesses are not located at contiguous addresses*. In the worst case, accesses with the same stride as the line size would cause the lowest performance, since all accesses could become misses. For these applications, using a set-associative cache is the easiest solution that does not require code changes.

Figure 4.4 shows an example of a 2-way set-associative cache. The data fetched from main memory can be stored in any cache set. The replace policy that we are using in our example code is Least Recent Used (LRU), but other algorithms can

Algorithm 4 Write data to direct-mapped cache

Require: 32-bit *addr* **and** *data* **and** *Cache* with a pointer *ptr_mem* to external memory
Ensure: $Cache[addr] = data$

```
1:  tag, line, word ← addr
2:  request ← request + 1
3:  if tag = Cache.tags[line] and Cache.valid[line] then
4:      hit ← hit + 1
5:  else
6:      if Cache.dirty[line] then
7:          location ← Cache.tags[line], line
8:          ptr_mem[location] ← Cache.array[line]
9:      end if
10:     loc ← addr >> LINE_BITS
11:     Cache.array[line] ← ptr_mem[loc]
12: end if
13: Cache.tags[line] ← tag
14: Cache.valid[line] ← true
15: Cache.dirty[line] ← true
16: Cache.array[line].slice(word) ← data
```

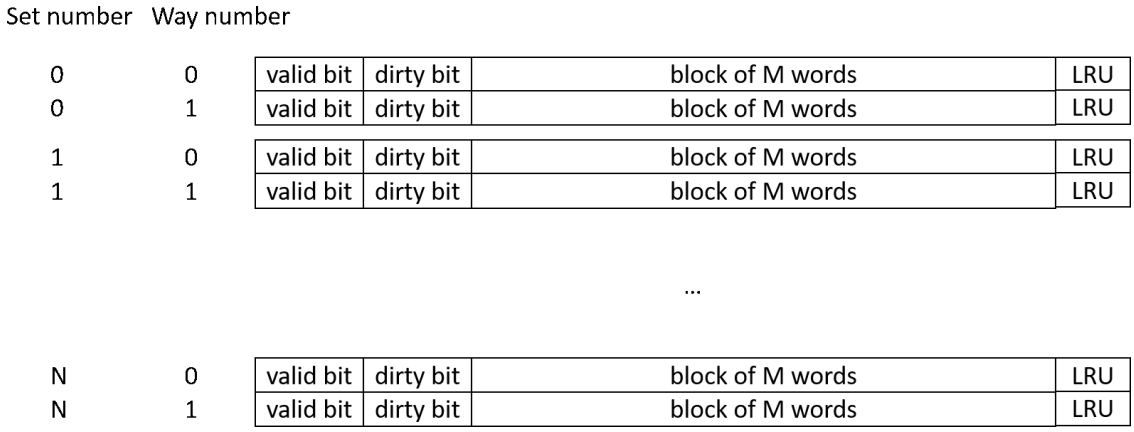


Figure 4.4: Diagram of a two-way set-associative cache

be implemented as well. In Figure 4.4, the *LRU* field records the last time when a cache line has been read or written. In this research, we use as time stamp (i.e., *LRU* value) the *request* counter, which was also used for statistical purposes in Algorithm 3 and Algorithm 4.

Designers should carefully choose the number of ways of a set-associative cache

when optimizing the performance, because a large number of ways causes higher resource utilization. The adaptation of traditional cache simulators to our methodology, basically by having a separate cache for each kernel argument, is left to future work.

Just like in the case of direct mapped caches, also for set-associative caches we have three variants: read-only, write-only and read-write. In this work we did not consider fully associative caches due to the high cost of the Content Addressable Memory.

4.3 Memory Access Patterns

As discussed above, we focus on multi-dimensional array access patterns because they are commonly used in our target application domain, namely image and video processing, neural networks, and so on. Moreover, as we mentioned above, we assume that:

1. These arrays are mapped to external DRAM,
2. The implementation has a dedicated cache for each concurrent process and for each individual array,
3. When different processes, e.g., OpenCL kernels, access the same array, then either the application guarantees that concurrent accesses occur in different parts of the array, or the implementation ensures this (e.g., using ping-pong buffers).

For these reasons, the discussion below will not consider interference among different data structures, nor among different processes.

In this section we consider how an application process accesses a given array and we analyze the various access patterns that we have observed in the literature. Some of the most commonly used patterns, which are discussed more at length below, are:

- sequential (or unit stride) within inner loop iterations,
- sequential within inner loop iterations with overlap among outer loop iterations,
- sequential with non-unit stride,
- window or neighborhood-based,
- random.

They are shown in Figure 4.5 and cover about 95% of the applications [35]. For example, in matrix multiplication one array is accessed in row-major order (i.e. sequential), while the other one is accessed in column-major order (i.e. with a stride equal to row length).

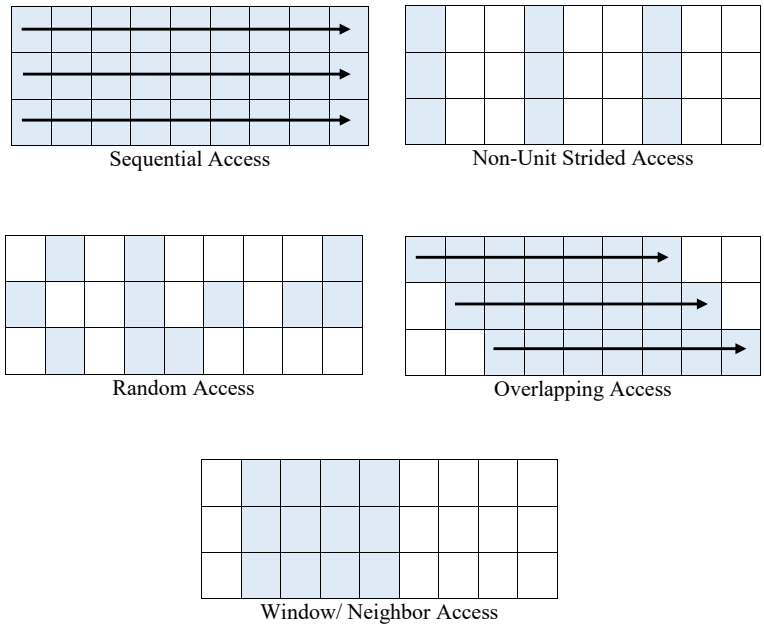


Figure 4.5: Different types of memory access patterns

In a first part of our research, we created a large number of test cases covering several variants of these patterns, and for each one we exhaustively searched for the best cache architecture under various cache parameter choices, analyzing hit rates and execution times. The results for exhaustive search are colour coded just for better visualization. The scale of dark green to dark red depicts the increase in execution time, among all the values of the table.

Note that, as discussed below, the best hit rate does not always result in the best performance when considering high-level synthesis of hardware. This is because a complex cache architecture (e.g., multi-way set-associative) may have a negative impact in terms of clock cycle or throughput, and hence perform worse than a simple one of comparable size. This means that some of the tenets of traditional cache architecture explorations for processors (i.e. a higher hit rate is always better) do not hold in our domain, which *requires a new cache exploration methodology.*

All results were generated using the Xilinx SDx Design Suite and implemented

on the FPGA included in the Amazon Web Services F1 instances, namely a Xilinx Virtex Ultrascale+ with about 2.5M logic elements.

4.3.1 Sequential access

Sequential memory access is probably the most commonly used pattern, in which each element is accessed once, right after the previous one in the array.

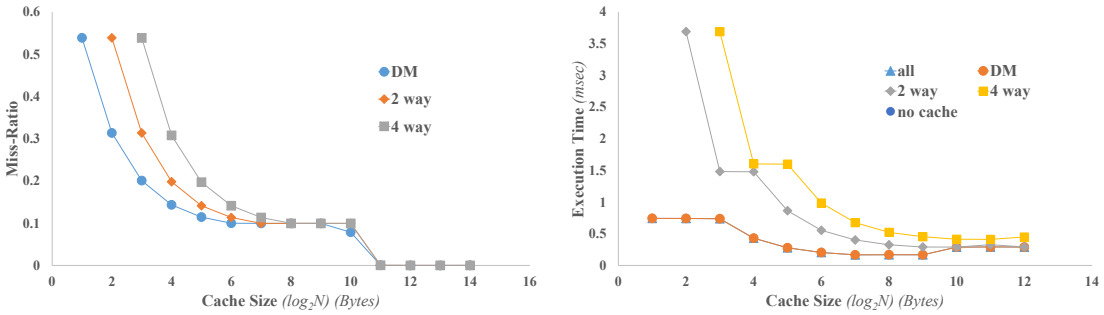
Although there is no data reuse, a cache can still improve performance over direct DRAM access, because it enables burst memory accesses. HLS tools like Vivado HLS attempt to generate burst accesses when they detect sequential memory reads or writes, but they are not always able to infer it. The simple line fetching loop inside the cache implementation, carefully crafted to fit tool requirements for burst access implementation, would ensure the use of the very efficient burst memory access mode.

Elements ($\log_2$n)	LINE SIZE (Bytes)																				
	2			4			8			16			32			64			128		
	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W
1	0.743303																				
2	0.743389	3.69254		0.742696																	
3	0.745078	1.48251	3.6911	0.741476	2.46052		0.738231														
4	1.47952	1.48102	1.60332	0.743252	1.47944	2.46032	0.739427	1.84779		0.431077											
5	1.48282	1.60526	1.60434	1.47686	1.47605	1.59986	0.741606	0.864281	1.84566	0.43364	1.53626		0.279505								
6	1.48285	1.47956	2.95343	1.47812	1.60031	1.59924	0.861225	0.86402	0.986129	0.433284	0.554709	1.53757	0.276929	1.3828		0.205245					
7	1.48168	1.60344	1.60328	1.47951	1.47756	1.725	0.864541	0.987961	0.983733	0.557074	0.554988	0.676309	0.278205	0.402538	1.3839	0.204098	1.30995		0.167499		
8		1.60393	2.95553	1.4798	1.59929	1.59928	0.861645	0.862825	1.10899	0.556974	0.676726	0.676269	0.402156	0.40044	0.522877	0.203555	0.32647	1.31163	0.169552	1.27281	
9			2.95627		1.59951	1.72304	0.864027	0.984213	0.984946	0.553618	0.556258	0.801523	0.400667	0.523638	0.522074	0.326986	0.331067	0.453522	0.168656	0.291584	1.27307
10						1.72575		0.984726	1.1081	0.55588	0.554102	0.677642	0.402075	0.400243	0.645931	0.326373	0.44973	0.451552	0.289431	0.289744	0.414557
11									1.10743		0.676023	0.677076	0.399621	0.4004	0.522008	0.327702	0.327626	0.575187	0.29214	0.413754	0.412388
12												0.800743		0.522564	0.522578	0.328639	0.330048	0.449097	0.290624	0.29222	0.536505

Figure 4.6: Execution Time for Sequential Memory accesses

An exhaustive cache layout search has been conducted on a sample code with an array having sequential memory accesses. The dimension of array used in this case is 256x144 with a datatype of 8 bit "unsigned char". The results of exhaustive search are shown in the table in fig 4.6. Our experimental results confirm that hit rate and performance of sequential memory accesses uniformly increase with line size. As can be seen in Figure 4.7a, increasing cache associativity *at equal cache size* does not increase hit rate. Hit rate is directly correlated only with line size.

Note also that the best performance *for our specific benchmark* is achieved for



(a) Layout vs miss rate

(b) Layout vs execution time

Figure 4.7: Pareto-optimal configurations for sequential memory accesses

an intermediate cache size of 512 words (Fig 4.7b). This is somewhat counter-intuitive and suggests that using the largest cache that fits on the chosen FPGA chip may be sub-optimal. This is due to the fact that for large associative caches Vivado HLS increases the Initiation Interval of the innermost loop, due to the cost of cache lookup.

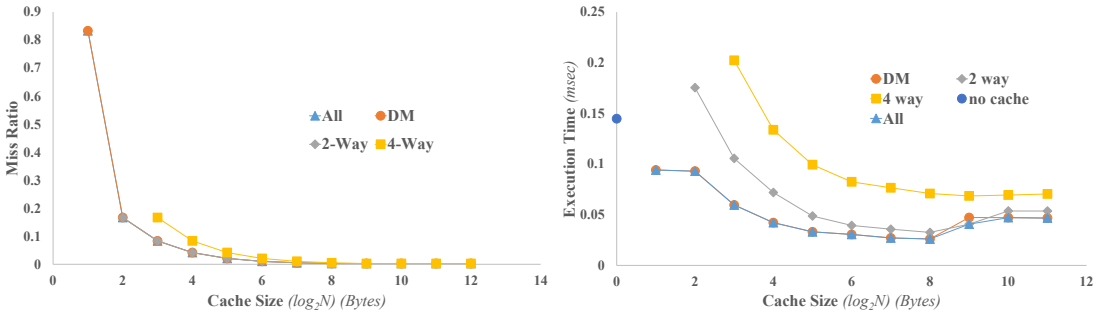
4.3.2 Overlapping access

As the name suggests, *overlapping memory access means that some data used in one iteration of an outer loop (e.g., one work item in OpenCL), where an inner loop performs sequential access, are also used in its next iteration, along with some new data.* It is one of the most cache friendly access patterns, because it also provides a significant amount of data reuse in addition to the already mentioned advantage of burst transfer. An important factor in this case is the number of elements accessed in the inner loop after which we have the reuse. In the best case, the cache is large enough to hold all data accessed by the inner loop until the outer loop restarts, and hence reuses them. Performance is degraded, however, when cached elements are prematurely replaced by inner loop iterations that need to access more data than cache capacity.

As we did in sequential access pattern, an exhaustive search was conducted for overlapping access pattern as well. The test case have similar array dimension of 256x144 and three consecutive elements are accessed in each iteration of inner-most loop. In this way we have an overlap of two words between two consecutive loop iterations. The results are shown in fig. 4.8. Our experimental results confirm that, like sequential, direct mapped cache having larger line size outperformed others. We can also see that the results of two way cache are also very close to the optimal ones in this case, especially with a cache size greater than or equal to 64 words.

Elements (log_2n)	LINE SIZE (Bytes)																				
	2			4			8			16			32			64			128		
	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W
1	2.42037																				
2	1.60486	6.14801		1.93996																	
3	1.60494	3.07584	6.27486	1.60074	4.91757		1.15428														
4	3.20161	3.20221	3.56833	1.59942	1.8471	5.04072	0.985059	4.30468		0.762285											
5	3.19975	3.44656	3.567	1.97038	1.9718	2.33729	0.985729	1.2314	4.42653	0.677179	3.99586		0.641513								
6	3.19872	3.19837	3.69017	1.9708	2.21627	2.33736	1.35508	1.35418	1.72192	0.677905	0.802843	3.99413	0.524067	3.84429		0.472171					
7	3.19786			1.96879	1.96811	2.46009	1.35489	1.5985	1.72384	1.04538	0.923417	1.41392	0.524817	0.645845	3.84109	0.451709	3.76789		0.423344		
8				1.96918		2.33599	1.35293	1.35249	1.96693	1.04684	1.16843	1.41462	0.892036	0.769858	1.26236	0.449397	0.573957	4.01345	0.41503	3.73169	
9							1.35361		1.72131	1.04599	1.16832	1.65971	0.892666	1.01512	1.26247	0.818712	0.696878	1.18834	0.412818	0.535396	3.97763
10										1.04776	1.16835	1.41392	0.894172	1.01496	1.50502	0.820291	0.943709	1.18776	0.781847	0.660812	1.15175
11													0.892563	1.01342	1.25963	0.821875	0.94293	1.43383	0.783893	0.905611	1.1497
12																0.820881	0.94111	1.18742	0.78325	0.904784	1.39742

Figure 4.8: Execution Time for Overlapping Memory accesses



(a) Layout vs Miss-Rate

(b) Layout vs Execution Time

Figure 4.9: Pareto-Optimal configurations for Overlapping access

4.3.3 Non-unit stride

Some applications access data in patterns which, while regularly spaced, are not adjacent to each other. If stride length is of one row, then such a pattern can also be called column linear. If the stride is small, then increasing cache line size results in higher hit rate, as for sequential access. In case the stride is equal to or larger than cache line size, then the cache does not help much. For smaller strides, it is recommended that the product of associativity and number of cache lines be equal to number of accesses done by the innermost loop.

For strided access, our exhaustive search included a stride of 8, 16 and 64 elements. Three different variants were tried just to verify the relationship between

strided access and cache layout. We also conducted experiments among number of accesses in the innermost loop. We have presented the exhaustive results of experimentation for stride of 8 elements (fig. 4.10) as other were similar to that, as can be seen from fig 4.11d.

As we can see in Figure 4.11, although the direct mapped cache performs poorly (Figure 4.11a & 4.11b) until we have more elements in the cache than stride size, the execution time (Figure 4.11c) of an N-way associative cache with the same total size is either higher than or the same as the direct mapped cache. This phenomenon occurs, as mentioned above, because a more complex architecture leads to a lower throughput for the innermost (pipelined or unrolled) loop.

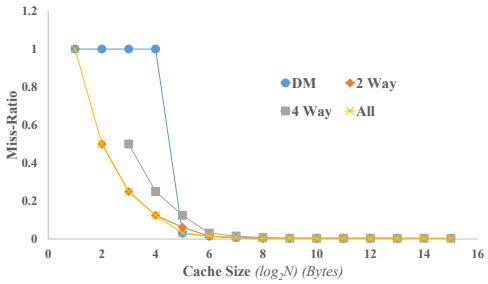
Elements (log_2n)	LINE SIZE (Bytes)																				
	2			4			8			16			32			64			128		
	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W
1	1.64729																				
2	1.64709	7.35104		1.64605																	
3		2.0029	7.47547	1.64753	4.9045		1.64691														
4	2.9559	3.12281	3.24586	1.47592	1.65588	5.02961	0.863251	3.68158		1.21153											
5	2.95168	3.07729	3.20018	1.72473	1.72327	1.97001	0.860102	0.985346	3.1968	0.555804	2.767		0.954431								
6	2.95558	2.95397	3.32198	1.72305	1.84738	1.96903	1.10678	1.10786	1.35345	0.553555	0.678209	2.76837	0.400493	2.61474		0.496451					
7	2.95547		3.19997	1.72392	1.72115	2.09097	1.10615	1.35367	1.35543	0.800326	0.798538	1.04562	0.399093	0.524073	2.61247	0.326636	2.5395		0.378398		
8				1.72572	1.84557	1.96973	1.10963	1.10685	1.47766	0.801476	0.92313	1.04514	0.645888	0.645945	0.893089	0.329395	0.449647	2.6641	0.292047	2.50332	
9							1.10672	1.23008	1.35312	0.801336	0.921931	1.16701	0.647547	0.768946	0.89112	0.574723	0.572371	0.819495	0.291047	0.412419	2.62751
10										0.799707	0.921668	1.04474	0.6488	0.770462	1.01622	0.574027	0.697568	0.819875	0.536406	0.537389	0.781904
11													0.648564	0.768239	0.891377	0.575617	0.698174	0.94303	0.538304	0.658227	0.781371
12																0.572375	0.695628	0.81866	0.537578	0.658293	0.904401

Figure 4.10: Execution Time for Stride Memory accesses (Stride of 8 elements)

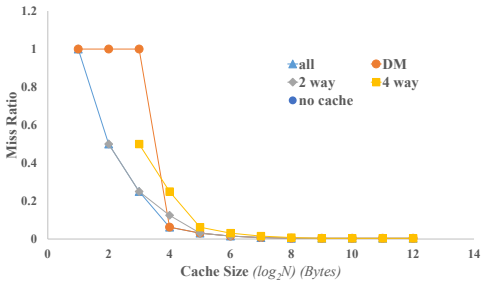
4.3.4 Window / neighbour

Most image processing algorithms work on a sliding window or neighboring elements (e.g., a diamond-shaped neighborhood). This means that for each pixel that is being considered, its surrounding pixels are also required for a computation. Within each row, access is linear with overlap (as long as the window stride is not larger than line size). On the other hand, it is also strided among rows. Hence, since we do not want to flush previous row data, the number of cache lines should be at least the number of rows.

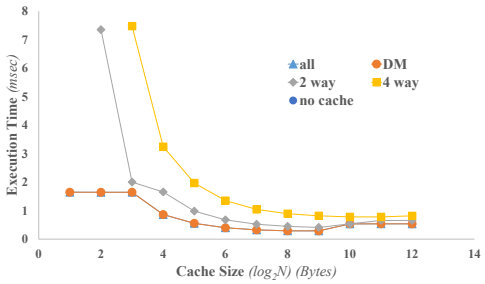
The results of exhaustive search over an array size of 32x16 is shown in the figure 4.12. The window of 4x4 was used for experimental purposes. Contradicting



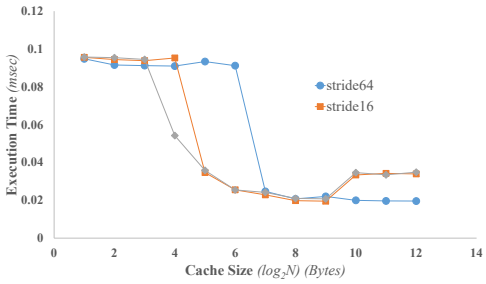
(a) Layout vs miss rate (stride of 16 elements)



(b) Layout vs miss rate (stride of 8 elements)



(c) Layout vs Execution Time (stride of 8 elements)



(d) Execution Time for DM Cache for different Strides

Figure 4.11: Pareto-Optimal configurations for Stride access

to other access patterns (Fig. 4.13), 4 way cache shows better performance not only for hit rates, but also in terms of execution times. Still the simple bookkeeping of direct mapped cache, despite of its bad hit ratio, also make it a good candidate. One thing to note here is that since the array is small, so direct mapped cache is performing well. In case of large arrays/real applications, direct mapped caches do not perform very well in case of window access pattern.

4.3.5 Random

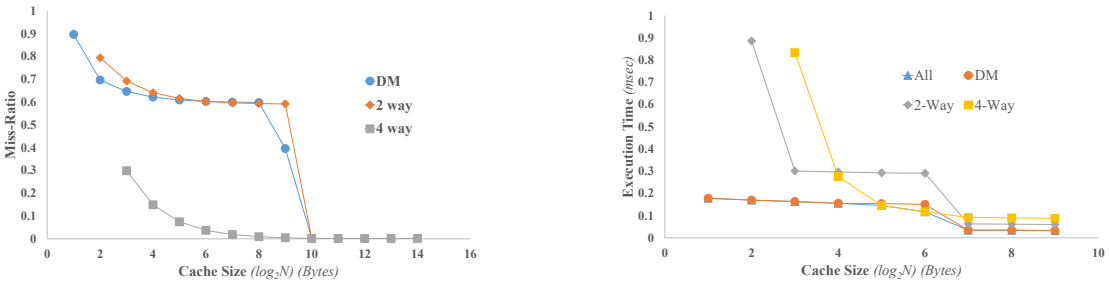
Although this pattern is not very common among our target applications, it is used by some, such as sorting. Even though in this case we cannot draw any general conclusions, caches still can help if there is at least some level of locality, for example because the algorithm was optimized to work well on a CPU to exploit its cache.

4.4 Cache Tuning using Heuristics

A basic and unguided approach for cache tuning would be to try all the possible configurations for the cache. In this way we can get the *miss rate* and cost of each

Elements (log_2n)	LINE SIZE (Bytes)																				
	2			4			8			16			32			64			128		
	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W	DM	2W	4W
1	0.178142																				
2	0.17411	0.886072		0.169572																	
3	0.168023	0.300666	0.833524	0.164161	0.673333		0.163028														
4	0.303661	0.299676	0.275929	0.163089	0.296365	0.613458	0.156098	0.577696		0.155308											
5	0.302595	0.350386	0.198558	0.301309	0.294296	0.145498	0.157814	0.292339	0.424993	0.154111	0.531924		0.395582								
6	0.287292	0.294202	0.229934	0.286266	0.3334	0.143529	0.284403	0.290737	0.11697	0.150573	0.290277	0.331671	0.294962	0.33446		0.41534					
7	0.091114	0.119055	0.117776	0.07569	0.07501	0.130321	0.06637	0.096921	0.094702	0.065154	0.062685	0.091297	0.034283	0.33356	0.281627	0.196051	0.345189		0.421091		
8		0.117596	0.147507	0.077583	0.102886	0.103748	0.068699	0.068799	0.121342	0.063501	0.089578	0.089591	0.33235	0.061278	0.448664	0.034273	0.347744	0.280735	0.196505	0.344519	
9			0.147008		0.103692	0.131074	0.067046	0.095772	0.095981	0.062048	0.063248	0.118496	0.333519	0.088618	0.087635	0.345285	0.059763	0.466586	0.032237	0.345875	0.277826

Figure 4.12: Execution Time for Window Memory accesses



(a) Layout vs Miss-Rate

(b) Layout vs Execution Time

Figure 4.13: Pareto-Optimal configurations for 4x4 Window access

configuration in terms of resources and energy. With all this data we can choose the best possible configuration for our application. The biggest problem with this issue is time. The exhaustive search will take a lot of time as there could be many possible configurations with different line sizes, number of lines and assosiativity. Therefore, there should be a heuristic that should reduce these configurations among only the most optimal ones.

4.4.1 Heuristics in Literature

Many heuristics are available in the literature for tuning different parameters of cache. Here we present a heuristic, proposed by Gordon-Ross et al. in her book chapter [31] as well as many publications. In their heuristic, the three parameter they tuned are cache line size, cache size , assosiativity and way prediction. First

they did experiments to find out the order of impact by each parameter on efficiency of cache. In this way they prioritize the one with high impact.

The heuristic they developed based on the importance of parameters is:

1. *with a 2 Kbyte, direct-mapped cache with a 16 byte line size. Increase the cache size to 4 Kbytes. If the increase in cache size causes a decrease in energy consumption, increase the cache size to 8 Kbytes. Choose the cache size with the best energy consumption.*

2. *For the best cache size determined in step 1, increase the line size from 16 bytes to 32 bytes. If the increase in line size causes a decrease in energy consumption, increase the line size to 64 bytes. Choose the line size with the best energy consumption.*

3. *For the best cache size determined in step 1 and the best line size determined in step 2, increase the associativity to 2 ways. If the increase in associativity causes a decrease in energy consumption, increase the associativity to 4 ways. Choose the associativity with the best energy consumption.*

4. *If step (3) determined the best associativity to be greater than 1, determine if enabling way prediction results in energy savings.*

Their heuristic search on average, **5.8** configurations to find the most optimal cache layout. They also claimed that this heuristic find the most optimal cache configuration in all the cases.

In our case, as we discussed above, synthesizing a large cache with Vivado HLS is sometimes not useful. Moreover this phenomena can also be seen from the exhaustive search of different patterns shown above. Therefore, working with our cache, we swap the first two steps in order to make it more useful.

In our implementation of the heuristic, algorithm starts with a 8 byte line size as shown in figure 4.14. Initial line size of 8 bytes is chosen as this is the minimum size suggested by the tool to infer burst transfer. After that algorithm increase it and compare the performance with last configuration. We can keep on doing that until there is a $\alpha\%$ reduction in kernel execution time. The value of α can be defined by the user. In our case we used $\alpha = 10$. Once we do not get a performance increase (PI) of $\alpha\%$, we looked for the number of lines and finally for the assosiativity. The same rule is applied for these two factors as well. The results of the heuristics are explained in section 4.4.2.

4.4.2 Experimental Results

In order to evaluate the effectiveness of the heuristic with our cache, we applied the heuristic [31] to all the patterns described in section 4.3. The results of the

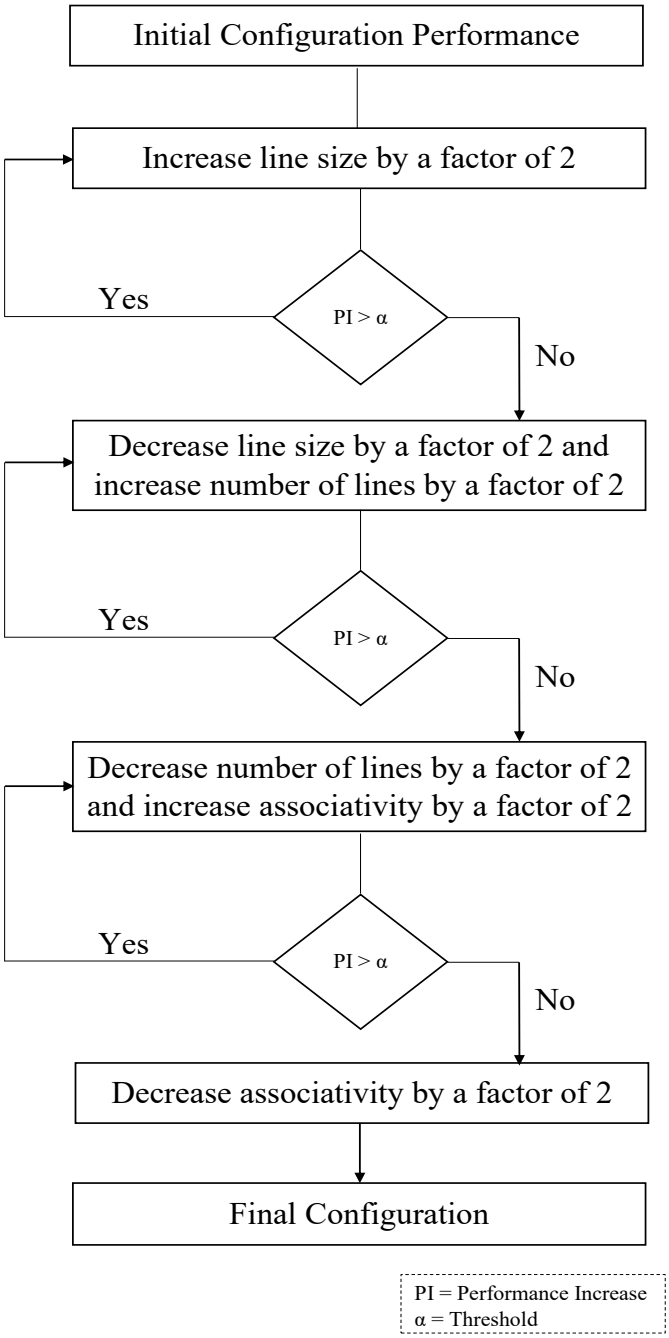
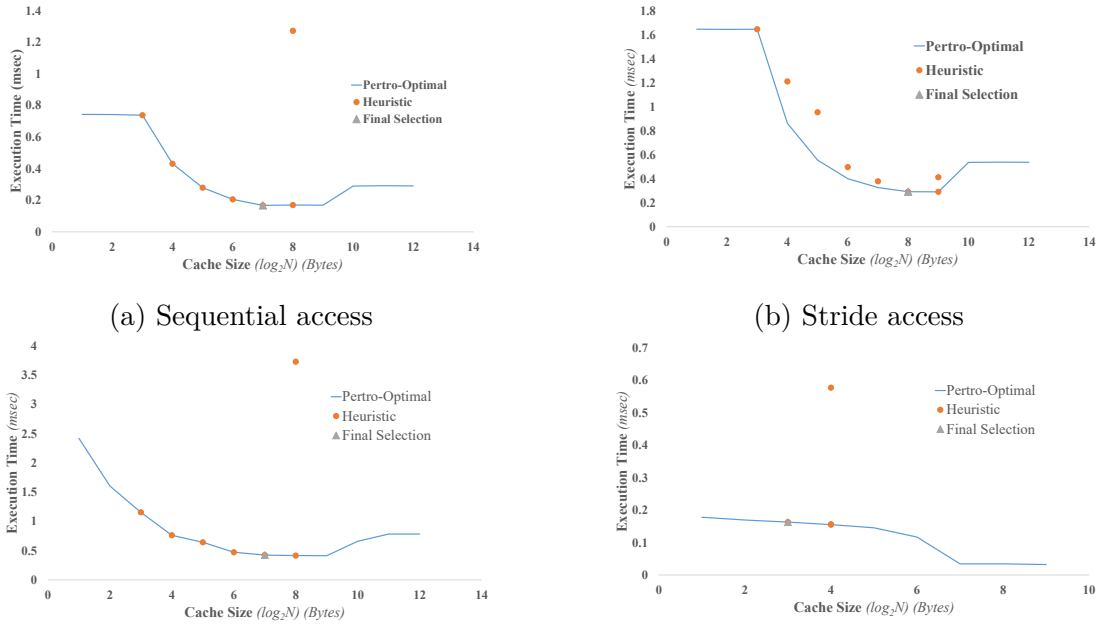


Figure 4.14: Heuristic for cache layout

heuristics were compared with the Pareto-optimal points of the exhaustive search for each of the patterns and are shown in fig. 4.15.



(a) Sequential access

(b) Stride access

(c) Overlap access

(d) Window access

Figure 4.15: Results for applying heuristics to Different access patterns

Sequential access

In cases where array access pattern is sequential, results shown by heuristic algorithm are very impressive as shown in fig. 4.15a. The heuristic algorithm was able to find the most suitable layout in 7 configurations instead of around 140 configurations of the cache layout we explored in exhaustive search. Other than that, the main positive aspect about this algorithm is that it was able to reach the most appropriate or recommended layout in this case. One thing to note here is that, this heuristic would have totally failed if we had started our exploration for the line size of less than 8 words. This was another reason we started our heuristic from this point.

Overlap access

As discussed above, overlapping access pattern is not only similar to sequential access pattern, but also more cache friendly. They also show a linear performance increase with increase in line size of the cache.

The results of heuristic (as shown in fig 4.15c) when applied to overlap access pattern are also impressive and are able to find an optimal cache layout in 7 configurations. In this case, the results of heuristic algorithm are even more stable than sequential access, as even if we start from a line size lower than 8 bytes, heuristic will still be able to find a decent implementation.

Stride access

In designs, where access pattern from the global memory is strided, heuristic is effective in some cases while fails in the other. For example, in case of stride of 8 elements, we can see that (fig. 4.15b) the heuristic is able to find a decent layout in 8 configurations. This is true for all the strides less than or equal to 8 bytes.

The problem arises if the stride size is greater than 8 bytes, than we will not have a decrease in execution time until the line size of the cache is greater than the stride (fig. 4.11d). Therefore, this heuristic will not be able to find a beneficial cache configuration for the design. This problem can be resolved by changing the initial point of reference to a greater line size according to the size of strided access, but it is not trivial for the heuristic to determine and require manual modification. Therefore, we can say that, in case of strided access, the success of heuristic is subject to manual intervention.

Window or Neighbour access

As we have witnessed from the results of stride access pattern, if there is no improvement for some cache configurations, the heuristic algorithm cease to function. This is the reason why the heuristic failed for window access pattern. The heuristic algorithm looked for 4 configurations, but since there was not a considerable decrease in execution time for the kernel, the cache is not configured optimally.

Therefore, we can say that heuristic algorithm is easy to develop but cannot guarantee decent results in every case.

4.5 PEDAL

As discussed above, if the cache architecture is not choosen properly, then a good performance to cost ratio cannot be achieved. Moreover, finding a decent layout using heuristic algorithms is not trivial mainly for the two reasons:

1. They do not guarantee a decent configuration every time
2. They require the results of a few hardware emulations or Implementations which are time consuming

In this regard, we present a technique called ‘PEDAL’ (Pattern Evinced Determination of Appropriate Layout) to customize data caches. It provides the designer with a Pareto-optimal cache architecture, using a simulation-based array access pattern analysis. We focus mostly on array address analysis, because they are the most common large data structures used in the kind of applications targeted by this work. Since these applications feature static addressing patterns and limited

runtime decision making, both simulation and polyhedral analysis techniques can be used. Here we focus on the former and leave the latter to future work.

Figure 4.16 shows the design flow. First of all, an annotated simulation collects the trace of all array accesses. Then we use machine learning algorithms to analyze and classify the access patterns and size and number of accesses per work item for *each array*, to finally suggest the optimal layout for its dedicated cache.

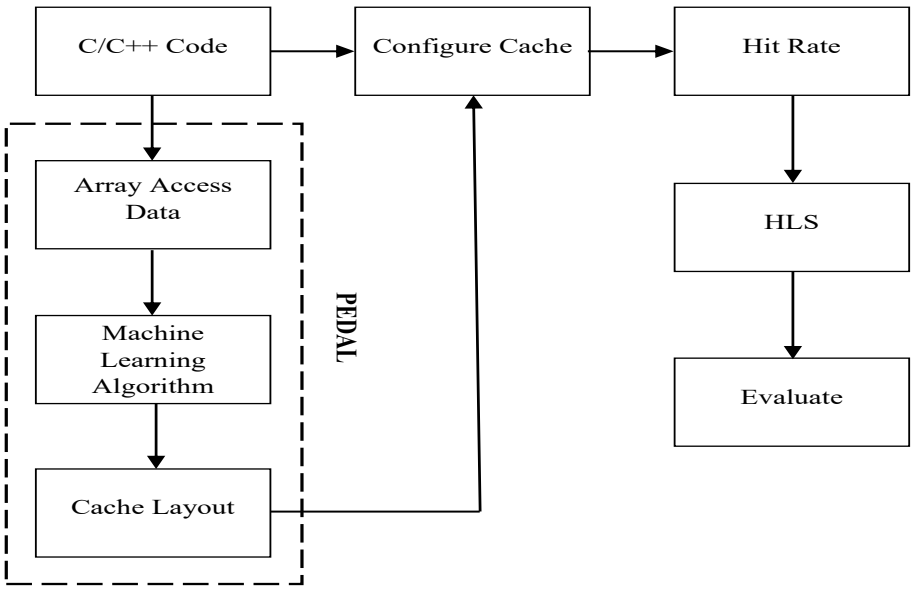


Figure 4.16: Design flow of proposed method (PEDAL)

PEDAL saves significant design effort and speeds up considerably memory optimization. It can be applied to any out-of-the-box or partially optimized C++ or OpenCL code, and to any embedded cache architecture. It can run in fully automated mode or provide the designer with Pareto-optimal options to choose from, and does not require any RTL design knowledge.

4.5.1 Algorithm

As described in fig 4.17, the first step by PEDAL is to extract the data access pattern for each global array automatically and then based on trained AI algorithm, it detects the pattern of data accessed by that array. For this purpose the source code of the kernel is annotated using partools [40] to get the access pattern, whenever a read or write operation is performed on the array. The indices of the

array accessed are dumped into a file which is then later used as a test set to detect the pattern.

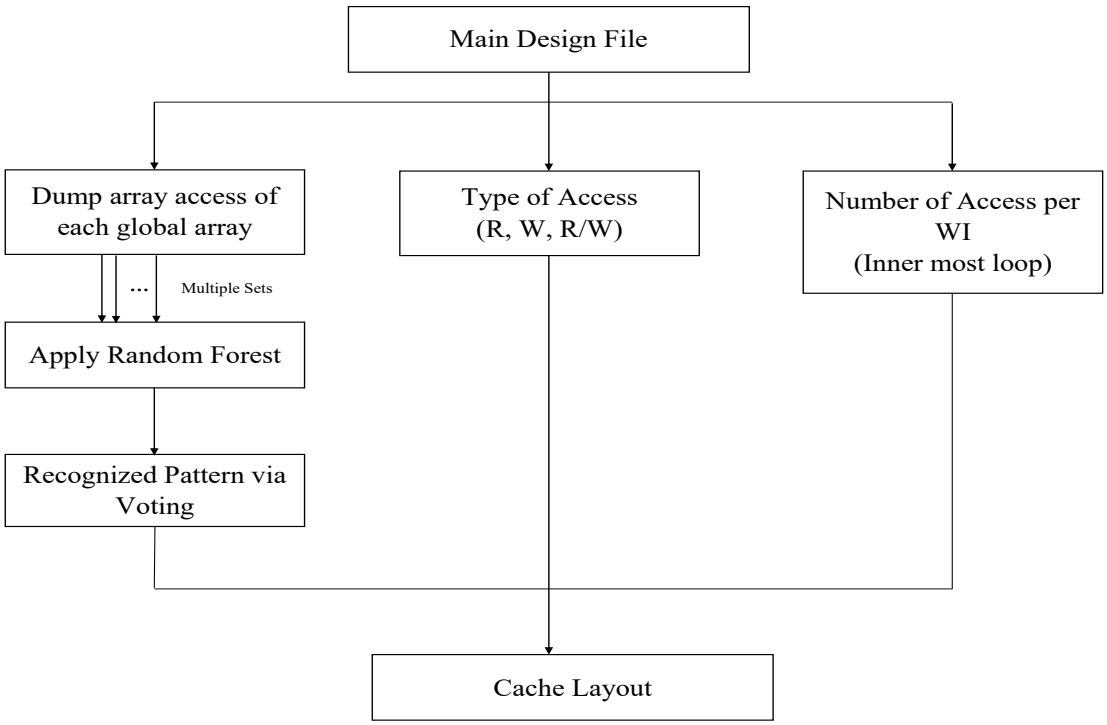


Figure 4.17: PEDAL Algorithm

Partools [40] is a tool set for C code, which looks for the dependencies in the code and not only give the performance estimation, but also advice for its parallel implementation. In this work, we used the C annotator build under partools for annotating C code. The annotator is build using OCaml which is based on Common Intermediate Language (CIL). After that we defined our own library to get the array traces from annotated C code.

The array access patterns are fed to trained machine learning algorithms. In our case the best results are achieved using Random forest algorithm which is explained in section 4.5.2. We used Weka software for implementing Random Forest algorithm. Weka is a collection of machine learning algorithms for data mining tasks. It contains tools for data preparation, classification, regression, clustering, association rules mining, and visualization. Weka is open source software issued under the GNU General Public License [79].

Here one important aspect of voting is implemented. The test is performed on at least 5 different samples extracted from the array access. This is necessary for two reasons:

- to make it error prone

- sometimes access pattern is not same for different loop iterations

so the samples are extracted from different access and then voted to get the best option.

The second piece of information required from the access pattern is the number of access in one iteration of the loop over work-items. This is also dumped in a file which is later used for the correct cache configuration. These files also contain the information about the types of accesses, whether only read, only write or mixture of the two. This information is needed to define a more optimal cache.

Based on the access pattern of array detected by Weka, number of accesses in one loop and type of accesses, PEDAL list a couple of Pareto optimal options and recommend the best one to be adopted in this scenario. Recommendations are based on the experimentation done in the section 4.3. Thus this script will provide you the best cache configuration for all the global arrays in the code automatically. The basic selection algorithm works as follows:

1. If the order of access is sequential, with no reuse, then the cache should have max line size. Only one line is enough and there is no advantage of having any assosiativity. The number of access in the inner most loop does not have much of an effect in this regard

2. If the order of access is sequential but it also reuses the previous data, i.e. overlap but no stride, then the line size should be maximum again but should have at least two lines.

3. If we have strided access without any reuse of data, then two factors are important, stride size and number of accesses in inner most loop. Here the Pareto optimal points have the number of access as product of assositivity and number of lines. Line size is also dependent on stride size.

4. If we have an overlapped strided access than the same rule as previous apply, just with the addition of line size being maximum possible this time.

5. For window access, it is essentially a combination of 1 and 3 (2 and 3 in case of sliding window). Here all the access in one row are sequential with or without overlap, while accesses between columns are strided. Here assositivity is necessary to avoid any conflict of mapping between different rows.

The suitable cache configurations are applied to the kernel file and the performance is evaluated.

4.5.2 Pattern Recognition using Random Forest

As stated in the section above, the basis of PEDAL is recognition of array patterns. Pattern recognition using Machine learning algorithms is not a new field. It has diverse applications from speech recognition to image recognition. Also there are a lot of applications in the field of Electronic Design Automation (EDA) for pattern recognition.

Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because its simplicity and the fact that it can be used for both classification and regression tasks.

Random Forest was proposed by Breiman [13] and defined as:

> *A random forest is a classifier consisting of a collection of tree-structured classifiers h(x, k), k = 1,... where the k are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x.*

Random Forest is a supervised learning algorithm. It builds a "forest" of Decision Trees. These trees are trained using "bagging" method. The bagging method is a combination of learning models which will increases the accuracy of overall result. Even better, random forests, during node splitting, looks for the best feature in a subset of features instead for the most important feature. To sum it up, Random forest algorithm is a collection of decision trees which are merged together for a more accurate and stable result.

Since it a combination of decision trees, hyperparameters for random forest are same as a decision tree or a bagging classifier. So, if we use random forest algorithm, we do not need to integrate a decision tree with a bagging classifier. Random Forest also have the capability to deal with Regression tasks.

In our application, we used a training set of 400000 samples (in groups of 200 samples per set). These sets are from different patterns explained in section 4.3. The samples train the network to detect the incoming test case among them. Moreover, all the samples in the training sets are normalized to 1024. It makes the sets independent of size of the array under consideration, as we are interested in the pattern of accesses.

4.6 Test case Implementations

To verify the results of PEDAL, we implemented that on 5 applications of Rosetta benchmark suite [85], one of our previous work on Lucas Kanade [8]. The five applications from Rosetta benchmark are Face Detection, 3D-Rendering, Optical Flow, Spam Filter and Digit recognition. They are not only computationally expensive but also are hungry for memory.

4.6.1 Face Detection

The face detection application used in Rosetta benchmark is adopted from [67]. The algorithm detects faces in a given picture. In their implementation, the input image size is 320x240 and is in gray scale. The output from the algorithm is the position and size of human faces detected in the image. Targeted throughput in this case is 30 frames per second so that it can be used with live video stream. To optimize the data transfer, in their implementation, they saved all the images on chip before processing. Similarly, all the results are written back to memory in form of bursts after finishing the computational part of the algorithm.

For the main computational file, the input is images with a bitwidth of 8 bit per pixel. There are four 32 bit integer outputs, namely x coordinate, y coordinate, width of face and height of face. All of the access in this case are sequential.

Following the PEDAL suggestion to use direct mapped caches, we used a direct mapped read cache with 32 lines of 512 bits each and a direct mapped write cache with 8 lines of 1024 bits each. Table 4.1 shows approximately the same execution time as the manually optimized implementation and more than 3.5x improvement over the unoptimized implementation of the code. Resource utilization of the cache is higher for a couple of reasons (that also effects the other results in this section):

- BRAM usage increases because we use wide lines to fully exploit DRAM interface bitwidth, and wide arrays are implemented by Vivado HLS using a large number of BRAMs [22]. In the future, we will explore the trade-off between BRAM usage and speed of loading a line from DRAM (which currently requires one clock cycle on the FPGA).
- LUT and FF usage increases due to cache logic, mostly the tag array which is read in parallel and hence must be fully partitioned into a register file.

Table 4.1: Results for Face Detection algorithm

Implementation	**Execution Time** (msec)	**Resource Utilization**			
		BRAM	**DSP**	**LUT**	**FF**
Unoptimized	83.7	42	79	49322	54222
Manually optimized	21.5	92	72	48217	54206
Cache	23.2	131	82	112857	157126

4.6.2 Digit Recognition

This benchmark from Rosetta suite work for the recognition of handwritten digits. It works on the principle of K-nearestneighbor (KNN) algorithm. In their implementation they used a subset of MNIST database [41]. They used 18000 training samples and 2000 test samples for their implementation. All the test and training cases are evenly split among all digits (0 to 9). For optimization purposes, they downsampled the image to 14x14 and represented each pixel using one bit. In this way they saved the whole image in 196 bit unsigned integer. Moreover, they also use burst read and write mode in order to store the information on chip.

For top level function, two input arguments are test and training set which are packed into 256 bit words and the output is the 8-bit integer as the decision of the algorithm. Both reading and writing in the algorithm is sequential.

In this case we also used direct mapped caches for input and output parameters. The best cache brings as many input data as possible on chip (i.e. 16384 training samples out of 20000), by using 8192 cache lines for the training data. The results in Table 4.2 show a performance improvement of two orders of magnitude with respect to the unoptimised version, but 6x worse than the manual implementation.

Table 4.2: Results for Digit Recognition algorithm

Implementation	**Execution Time** (msec)	**Resource Utilization**			
		BRAM	**DSP**	**LUT**	**FF**
Unoptimized	8669.6	2	2	22570	28100
Manually optimized	11.1	207	0	39971	33853
Cache (*8162 lines*)	65.2	284	0	152759	171889

4.6.3 Spam Filter

Spam filter benchmark in Rosetta suite is an implementation adopted from [60]. It uses stochastic gradient descent (SGD) to train a logistic regression (LR) model for spam email classification. The data set they used for training and testing of algorithm have 5000 emails, from which they used 4500 for training and 500 for testing. The representation of each email in the dataset is done as 1024 dimensional vector. These vectors have relative word frequencies in the format of 16 bit fixed point numbers. They used 5 epoches for training their network.

The manual optimization in terms of memory in this benchmark is again packing up the inputs and outputs in larger bitwidth words. The other optimization they

applied is that they copied all the required weights on chip using burst transfer. Similarly outputs were also writen back in form of bursts.

We used caches with 16 lines of 512 bits each for all three arrays. The results shown in Table 4.3 achieve 27x better performance than the unoptimized version, but 4x worse than the manual one.

Table 4.3: Results for Spam Filtering algorithm

Implementation	**Execution Time** (msec)	**Resource Utilization**			
		BRAM	**DSP**	**LUT**	**FF**
Unoptimized	2920.5	4	7	2280	3858
Manually optimized	25.1	90	224	7207	17434
Cache	105.5	65	224	57366	68470

4.6.4 3D-Rendering

This benchmark in Rosetta is taken from [55]. As the name suggests, the algorithm of 3D-Rendering take 3D triangular mesh models as input and provide 2D images as output. So basically, it takes vertices of 3D triangles and projects them on 2D images. The colors to the pixels are assigned according to altitude of triangle. In the Rosetta implementation of algorithm, the image size is 256x256 and all the pixels are 8-bit integers. The dataset used by Rosetta have 3192 triangle coordinates and they set a target of 30 fps throughput.

The kernel file have two arguments, an input which have the coordinates of triangles and the other is output. In Rosetta implementation, to optimize the data transfer, they packed the 8-bit input and output into 32 bits each.

We used two direct mapped caches with 2 lines of 512 bits for input data and 1 line of 512 bits for output data. In this case, we improved both the execution time 4x with respect to the unoptimized version and also outperformed by 25% the manually optimized implementation, as shown in Table 4.4. The same considerations as above apply for resource usage.

4.6.5 Optical Flow I

The Optical flow benchmark is an implementation of Lucas-Kanade Algorithm [42]. As explained in Chapter 3, this algorithm is used to calculate the displacement of the moving objects between consecutive image frames. This algorithm have a lot

Table 4.4: Results for 3D rendering algorithm

Implementation	Execution Time (msec)	Resource Utilization			
		BRAM	DSP	LUT	FF
Unoptimized	13.0	36	11	16865	21857
Manually optimized	4.4	36	11	6763	7916
Cache	3.3	51	11	18574	29551

of applications in different image/video processing tools. In Rosetta implementation, they are using MPI Sintel dataset [16] with an image resolution of 436x1024. The target throughput is 30 frames per second.

The kernel file have two arguments. The input arguments contain the image frames and the output have the calculated velocity in x and y direction. To optimize the memory transfers, they have packed the pixels of five frames together in large bitwidth datatype, i.e. first 8 bit pixel of all 5 frames are packed to form a datawidth of 40bits. They used a datawidth of 64 bits and rest of the bits are initialized to zero. Similarly, for output, two floats have packed into 64 bit wide datatype.

Our implementation uses two caches with 4 lines of 512 bits, both for input and output. As shown in Table 4.5, in this case we achieved 2x better performance than the unoptimized version, but 2x worse than the manual design.

Table 4.5: Results for Optical Flow algorithm

Implementation	Execution Time (msec)	Resource Utilization			
		BRAM	DSP	LUT	FF
Unoptimized	163.9	54	473	35473	64481
Manually optimized	42.0	55	484	38094	63483
Cache	86.6	98	473	45405	77271

4.6.6 Optical Flow II

The second implementation of Lucas Kanade Algorithm is adopted from the work done in Chapter 3 of this thesis for smart city application. Since in our implementation [8], we did not do any pre-conditioning of data, so the access pattern

is not sequential in our case. Therefore we used this case to verify the functionality of PEDAL in case of non sequential accesses.

This implementation takes two images as input, in addition to some camera and road coefficients. The accesses for images is window based while for coefficients is sequential. These characteristics were used for designing of the cache parameters. We implemented the code with no memory optimization using cache. Since the access pattern of the first frame is window-based, following PEDAL recommendation we used a 2-way associative cache with 16 lines per way. We used a 512-bit word (to access 15 8-bit elements from each row of the window). The second input frame uses strided access, so we implemented a direct mapped cache with 32 lines for it. For the sequential output we used a cache with 2 lines of 512 bits.

Note that in this case *the cache overhead in terms of BRAM is smaller than in the optimized one*, because a smaller associative cache contains all the data needed by the window access pattern, albeit with a lower performance.

Table 4.6: Results for Lucas Kanade algorithm

Implementation	**Execution Time** (msec)	**Resource Utilization**			
		BRAM	**DSP**	**LUT**	**FF**
Unoptimized	44210.0	31	56	37080	21367
Optimized	14883.87	122	51	24613	18410
Cache	17274.8	63	21	49773	67614

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis discusses several issues that appear during the optimization of memory intensive applications on FPGAs using high level synthesis. The activities are carried out as a part of doctoral studies in electronic design automation. To be more specific, this dissertation is a combination of two activities, one performing the optimizations manually on a specific application, while the other automates this aspect and selects the best layout for custom data caches to be used on FPGAs. The choice of FPGAs (over GPUs) is because of their high performance capabilities while consuming just a fraction of power. The only nuisance in the use of FPGAs is programming them in VHDL or Verilog, which now can be countered by using high level synthesis. High Level Synthesis provides an excellent platform for designers to exploit the capabilities of FPGAs without the long design times entailed by the use of Hardware Description Languages.

The first task is regarding the manual optimization of two memory intensive image processing algorithms embedded in a real life application. The application is proposed and developed in the context of smart city and provides the velocity and density of the vehicles on road in real time from the video stream captured by a camera. This information can be used by different stake holders such as public transportation, taxis and city planners. Real-time benefits of this data include less time spent on roads and can help to reduce pollution where in long run this data can be used for better planning of city and road infrastructure. The main optimization required in this application were related to the availability of data for algorithms from DRAM. Use of appropriate line buffers for each array mapped to external DRAM were used according to their data access pattern. The final implementation shows that computational optimizations do not achieve the required performance without carefully designing a custom memory architecture.

The second part of the research is regarding the use of custom data caches

for memory intensive algorithms. In this work, we focus on automating application profiling in order to select the best cache architecture for each DRAM array. We developed an algorithm to optimize inline caches that are synthesized from a C++ model onto an FPGA and have the opportunity to be tuned according to the memory access patterns. Caches are good substitutes for manually designed specialized on-chip buffers, but their architecture and parameters are usually application-specific. In particular, associative caches should be used very carefully, only when the access patterns require them, because even though they improve hit rate, they have a negative impact on the Initiation Interval or the clock cycle, i.e. on the throughput. Hence, caches must be architected and sized based on address trace analysis to get good performance benefits with reasonable resource costs. Firstly, we tuned the caches using heuristics algorithms. The heuristics can provide good estimation in certain cases while totally failing in others. Therefore our experimental results show that use of heuristics require manual intervention as automatic heuristic tuning cannot guarantee a good result every time. Thus, we developed a tool to find the most appropriate layout for the cache. As name suggests, PEDAL (Pattern Evinced Determination of Appropriate Layout), is based on memory access pattern identification for each DRAM-mapped application array. The use of PEDAL on benchmark applications shows that we can obtain performance results close to manual optimizations without virtually any designer effort.

Therefore, the bottom line is that memory intensive tasks can be performed optimally on FPGAs provided that the memory access are optimized manually or automatically. The computational capabilities and energy efficiency of FPGAs make them an excellent choice for computational as well as memory intensive tasks.

5.2 Future Work

As a future work, we would like to add more functionality to the PEDAL tool. On one hand, we will work to implement an *automated verification process* for the suggested layout, instead of verifying it manually. This feature will also help us to add all the verified test patterns into the training set for continuous learning. Secondly, we would also like to improve the use of the cache, by using information extracted by PEDAL. Since the tool knows the access pattern, and these patterns are fixed for the typical applications considered in this thesis, it can also help the cache in its internal bookkeeping activities. For example, it may be possible to optimize some accesses by knowing that they are always "hits", e.g. in the case of windows or unit strides, and hence increase the cache efficiency.

Nomenclature

Acronyms / Abbreviations

ALU Arithmetic and Logic Unit

CLB Configurable Logic Block

CPU Central Processing Unit

DSE Design Space Exploration

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High-Level Synthesis

HPC High Performance Computing

OpenCL Open Computing Language

SoC System On Chip

Bibliography

[1] Michael Adler et al. "Leap scratchpads: automatic memory and cache management for reconfigurable logic". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, pp. 25–28.

[2] M Tolga Akçura and S Burcu Avci. "How to make global cities: Information communication technologies and macro-level variables". In: *Technological Forecasting and Social Change* 89 (2014), pp. 68–79.

[3] Sam Allwinkle and Peter Cruickshank. "Creating smart-er cities: An overview". In: *Journal of urban technology* 18.2 (2011), pp. 1–16.

[4] J Anderson et al. *Getting Smart about Smart Cities: understanding the market opportunity in the cities of tomorrow*. 2012.

[5] Leo G Anthopoulos and Ioannis A Tsoukalas. "The implementation model of a Digital City. The case study of the Digital City of Trikala, Greece: e-Trikala". In: *Journal of e-Government* 2.2 (2006), pp. 91–109.

[6] Leonidas G Anthopoulos. "Understanding the smart city domain: A literature review". In: *Transforming city governments for successful smart cities*. Springer, 2015, pp. 9–21.

[7] Leonidas Anthopoulos and Panos Fitsilis. "Using Classification and Roadmapping techniques for Smart City viability's realization." In: *Electronic Journal of e-Government* 11.2 (2013).

[8] Arslan Arif et al. "Performance and energy-efficient implementation of a smart city application on FPGAs". In: *Journal of Real-Time Image Processing* (2018), pp. 1–15.

[9] Blaise Barney et al. "Introduction to parallel computing". In: *Lawrence Livermore National Laboratory* 6.13 (2010), p. 10.

[10] James Bell, David Casasent, and C Gordon Bell. "An investigation of alternative cache organizations". In: *IEEE Transactions on Computers* 100.4 (1974), pp. 346–351.

[11] Martin Blažević, Karla Brkić, and Tomislav Hrkać. "Towards Reversible De-Identification in Video Sequences Using 3D Avatars and Steganography". In: *arXiv preprint arXiv:1510.04861* (2015).

[12] Jean-Yves Bouguet. "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm". In: *Intel Corporation* 5.1-10 (2001), p. 4.

[13] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[14] Andre R Brodtkorb et al. "State-of-the-art in heterogeneous computing". In: *Scientific Programming* 18.1 (2010), pp. 1–33.

[15] Norbert Buch, Sergio A Velastin, and James Orwell. "A review of computer vision techniques for the analysis of urban traffic". In: *IEEE Transactions on Intelligent Transportation Systems* 12.3 (2011), pp. 920–939.

[16] Daniel J Butler et al. "A naturalistic open source movie for optical flow evaluation". In: *European Conference on Computer Vision*. Springer. 2012, pp. 611–625.

[17] Andrew Tzer-Yeu Chen et al. "Trusting the Computer in Computer Vision: A Privacy-Affirming Framework". In: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2017 IEEE Conference on*. IEEE. 2017, pp. 1360–1367.

[18] Shaoyi Cheng et al. "Exploiting memory-level parallelism in reconfigurable accelerators". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 157–160.

[19] Jongsok Choi et al. "Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 17–24.

[20] Robert T Collins et al. "A system for video surveillance and monitoring". In: *VSAM final report* (2000), pp. 1–68.

[21] Jason Cong et al. "An energy-efficient adaptive hybrid cache". In: *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press. 2011, pp. 67–72.

[22] Jason Cong et al. "Bandwidth optimization through on-chip memory restructuring for hls". In: *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE. 2017, pp. 1–6.

[23] Philippe Coussy et al. "An introduction to high-level synthesis". In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 8–17.

[24] Sokemi Rene Emmanuel Datondji et al. "A survey of vision-based traffic monitoring of road intersections". In: *IEEE transactions on intelligent transportation systems* 17.10 (2016), pp. 2681–2698.

[25] Christian De Schryver et al. "An energy efficient FPGA accelerator for monte carlo option pricing with the heston model". In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE. 2011, pp. 468–474.

[26] Yves Durand et al. "Euroserver: Energy efficient node for european microservers". In: *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE. 2014, pp. 206–213.

[27] *ECOSCALE Project*. http://www.ecoscale.eu/project-description.html. (Accessed on 01/11/2018).

[28] Juan Isaac Engel, Juan Martin, and Raquel Barco. "A Low-Complexity Vision-Based System for Real-Time Traffic Monitoring". In: *IEEE Transactions on Intelligent Transportation Systems* 18.5 (2017), pp. 1279–1288.

[29] Michael Fingeroff. *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[30] Benedict Gaster et al. *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Newnes, 2012.

[31] Ann Gordon-Ross et al. "Tuning caches to applications for low-energy embedded systems". In: *Ultra Low-Power Electronics and Design*. Springer, 2004, pp. 103–122.

[32] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323. URL: http://dx.doi.org/10.1109/ISSCC.2014.6757323.

[33] Mark Horowitz. "1.1 computing's energy problem (and what we can do about it)". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE. 2014, pp. 10–14.

[34] Lee Howes and Aaftab Munshi. *The OpenCL Specification*. 2015.

[35] Byunghyun Jang et al. "Exploiting memory access patterns to improve memory performance in data-parallel architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2011), pp. 105–118.

[36] Jani Jokinen, Tero Latvala, and José L Martinez Lastra. "Integrating smart city services using Arrowhead framework". In: *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*. IEEE. 2016, pp. 5568–5573.

[37] Norman P Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers". In: *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE. 1990, pp. 364–373.

[38] George Kalokerinos et al. "FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability". In: *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*. IEEE. 2009, pp. 149–156.

[39] Nicos Komninos. *Intelligent cities: innovation, knowledge systems, and digital spaces*. Taylor & Francis, 2002.

[40] Mihai T Lazarescu et al. "Energy-aware parallelization flow and toolset for C code". In: *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*. ACM. 2014, pp. 79–88.

[41] Yann LeCun. "The MNIST database of handwritten digits". In: *http://yann.lecun. com/exdb/mnist/* (1998).

[42] Bruce D Lucas, Takeo Kanade, et al. "An iterative image registration technique with an application to stereo vision". In: (1981).

[43] Liang Ma, Fahad Bin Muslim, and Luciano Lavagno. "High Performance and Low Power Monte Carlo Methods to Option Pricing Models via High Level Design and Synthesis". In: *European Modelling Symposium EMS2016 (EMS2016)*. Pisa, Italy, Nov. 2016.

[44] Liang Ma et al. "Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis". In: *IEEE Access* 5 (2017), pp. 18953–18974.

[45] Eric Matthews, Nicholas C. Doyle, and Lesley Shannon. "Design Space Exploration of L1 Data Caches for FPGA-Based Multiprocessor Systems". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: ACM, 2015, pp. 156–159. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689083. URL: http://doi.acm.org/10.1145/2684746.2689083.

[46] Iakovos Mavroidis et al. "ECOSCALE: Reconfigurable computing and runtime system for future exascale systems". In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 696–701.

[47] Brendan Tran Morris and Mohan Manubhai Trivedi. "A survey of vision-based trajectory learning and analysis for surveillance". In: *IEEE transactions on circuits and systems for video technology* 18.8 (2008), pp. 1114–1127.

[48] Brendan Tran Morris and Mohan Manubhai Trivedi. "Understanding vehicular traffic behavior from video: a survey of unsupervised approaches". In: *Journal of Electronic Imaging* 22.4 (2013), pp. 041113–041113.

[49] F. B. Muslim et al. “Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis”. In: *IEEE Access* 5 (2017), pp. 2747–2762. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2671881.

[50] Fahad Bin Muslim et al. “Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis”. In: *IEEE Access* 5 (2017), pp. 2747–2762.

[51] Elaine M Newton, Latanya Sweeney, and Bradley Malin. “Preserving privacy by de-identifying face images”. In: *IEEE transactions on Knowledge and Data Engineering* 17.2 (2005), pp. 232–243.

[52] *Optical Flow Design Example*. https://www.altera.com/support/support-resources/design-examples/design-software/opencl/optical-flow.html. (Accessed on 01/10/2018).

[53] Jian Ouyang et al. “SDA: Software-defined accelerator for large-scale DNN systems”. In: *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE. 2014, pp. 1–23.

[54] Kalin Ovtcharov et al. “Accelerating deep convolutional neural networks using specialized hardware”. In: *Microsoft Research Whitepaper* 2.11 (2015).

[55] Juan Pineda. “A parallel algorithm for polygon rasterization”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 22. 4. ACM. 1988, pp. 17–20.

[56] Andrew Putnam et al. “Performance and power of cache-based reconfigurable computing”. In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 395–405.

[57] Moinuddin K Qureshi, David Thompson, and Yale N Patt. “The V-Way cache: demand-based associativity via global replacement”. In: *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE. 2005, pp. 544–555.

[58] Hatem A Rashwan et al. “Understanding trust in privacy-aware video surveillance systems”. In: *International Journal of Information Security* 15.3 (2016), pp. 225–234.

[59] Nisarg Raval et al. “Markit: Privacy markers for protecting visual secrets”. In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*. ACM. 2014, pp. 1289–1295.

[60] Christian Robert. *Machine learning, a probabilistic perspective*. 2014.

[61] Franziska Roesner et al. “World-driven access control for continuous sensing”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1169–1181.

[62] Dyer Rolán, Basilio B Fraguela, and Ramón Doallo. "Adaptive line placement with the set balancing cache". In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE. 2009, pp. 529–540.

[63] *Scene 1.0 - Background subtraction and object tracking with TUIO*. http://scene.sourceforge.net/. (Accessed on 12/24/2018).

[64] *Scene 1.0 - Background subtraction and object tracking with TUIO*. http://scene.sourceforge.net/. (Accessed on 01/11/2018).

[65] Jeremy Schiff et al. "Respectful cameras: Detecting visual markers in real-time to address privacy concerns". In: *Protecting Privacy in Video Surveillance*. Springer, 2009, pp. 65–89.

[66] *SDAccel Environment Optimization Guide*. English. Version v2016.3. Xilinx Inc. 93 pp.

[67] Nitish Kumar Srivastava et al. "Accelerating Face Detection on Programmable SoC Using C-Based Synthesis". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 195–200.

[68] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.1-3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69. URL: http://dx.doi.org/10.1109/MCSE.2010.69.

[69] Prasanna Sundararajan. "High performance computing using FPGAs". In: *Xilinx White Paper: FPGAs* (2010), pp. 1–15.

[70] Bin Tian et al. "Hierarchical and networked vehicle surveillance in ITS: a survey". In: *IEEE transactions on intelligent transportation systems* 16.2 (2015), pp. 557–580.

[71] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[72] Berthold Ulmer. "VITA-an autonomous road vehicle (ARV) for collision avoidance in traffic". In: *Intelligent Vehicles' 92 Symposium., Proceedings of the*. IEEE. 1992, pp. 36–41.

[73] Kunfeng Wang et al. "A multi-view learning approach to foreground detection for traffic surveillance applications". In: *IEEE Transactions on Vehicular Technology* 65.6 (2016), pp. 4144–4158.

[74] Rick Weber et al. "Comparing hardware accelerators in scientific applications: A case study". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2011), pp. 58–68.

[75] M Williams. "The prometheus programme". In: *Towards Safer Road Transport-Engineering Solutions, IEE Colloquium on*. IET. 1992, pp. 4–1.

[76] Joseph G Wingbermuehle, Ron K Cytron, and Roger D Chamberlain. "Superoptimized memory subsystems for streaming applications". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, pp. 126–135.

[77] Felix Winterstein et al. "Custom-sized caches in application-specific memory hierarchies". In: *Field Programmable Technology (FPT), 2015 International Conference on*. IEEE. 2015, pp. 144–151.

[78] Felix Winterstein et al. "MATCHUP: memory abstractions for heap manipulating programs". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, pp. 136–145.

[79] Ian H Witten et al. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[80] Ding Xie, Jimmei Lai, and Jiarong Tong. "A high utilization rate routing algorithm for modern FPGA". In: *Solid-State and Integrated-Circuit Technology, 2008. ICSICT 2008. 9th International Conference on*. IEEE. 2008, pp. 2333–2336.

[81] Xilinx. *SDAccel Development Environment User Guide*. English. Version v2015.1. Xilinx. 95 pp.

[82] Alper Yilmaz, Omar Javed, and Mubarak Shah. "Object tracking: A survey". In: *Acm computing surveys (CSUR)* 38.4 (2006), p. 13.

[83] Yu. *Overload the Brackets Operator to Perform Complex Operations*. Nov. 2014. URL: https://argcv.com/articles/3228.c.

[84] Junping Zhang et al. "Data-driven intelligent transportation systems: A survey". In: *IEEE Transactions on Intelligent Transportation Systems* 12.4 (2011), pp. 1624–1639.

[85] Yuan Zhou et al. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs". In: *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb. 2018).

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.