Low-Power Lightweight Vision Supercomputer: Algorithms and Architectures

Presented by

Fouzhan Hosseini

To

Robotics, Brain, and Cognitive Sciense

Istituto Italiano di Tecnologia

and

Doctoral School of Life and Humanoid Technologies

Universit degli Studi di Genova

A thesis submitted in partial fulfillment of the requirements for

Doctor of Philosophy

April 2013

to my family and teachers

Those who went in pursuit of knowledge
Soared up so high, stretched the edge
Were still encaged by the same dark hedge
Brought us some tales ere life to death pledge.
Omar Khayyam

Acknowledgements

This not have been possible without the support, guidance, and help of many people who in one way or another extended their valuable assistance in the preparation and completion of this work.

First and foremost, a very special thank you goes to my supervisor Dr. Lorenzo Natale and also Dr. Chiara Bartolozzi for their support, guidance, and willingness to put up with me over the past few years. I am sure it has not always been easy for them, and I really appreciate it.

I would like to especially thank Prof. Giulio Sandini. Without his kind concern and consideration at a very difficult moment, this work definitely would have never come to an end.

Many thanks as well go to the technical support staff at the Robotics, Brain, and Cognitive Science (RBCS) department, the iCub support team, and the people in the Electronic Lab for all the help. I would like to specifically thank Marina Antonini, Ingrid Sica, and Francesca Cagnoni. They have always been great help.

I also would like to thank my previous supervisor Dr. Amir Fijany who I am sure did his best to help me in my way.

Special thanks also goes to my best friends and colleagues over the past few years, Dr. David Thomas Branson III, Shahrzad Latifi, Ali Paikan, Dr. Saeed Safari and Mandana Hamidi for their continuous support and encouragement. Not having such friends, I could not go all the way to the end.

Last but not the least, I would like to thank my family for their love, support and guidance throughout my life. Without them none of this would have been possible from the start.

Abstract

Mobile robots and humanoids represent an emerging and challenging example of embedded computing applications. On one hand, in order to achieve a large degree of autonomy and intelligent behavior, these systems require a very significant computational capability to perform a wide variety of complex tasks, some of them with real-time constraints. On the other hand, they are severely limited in terms of size, weight, and particularly power consumption of their embedded computing system since they should carry their own power supply. Moreover, since these systems need to implement a wide variety of applications, their computing systems should provide programmability and adaptability of a general purpose platform.

This thesis has followed two approaches to provide low-power, lightweight, high performance computing architectures for mobile robots and humanoids:

- exploiting new emerging parallel architectures which provide both high computational capability and low-power consumption, and
- extracting and processing only important data by using emerging bio-inspired sensors such as DVS vision sensor.

We proposed a low-power high performance vision architecture for mobile robots and humanoids which includes CSX SIMD and Tilera MIMD architectures. The estimated power consumption of such architecture is about 50 watt, while the peck performance is over 96 GFLOPs and 144 GOPs.

During this research, various parallel and event-based algorithms with different computational characteristics have been implemented on our proposed parallel architecture.

Currently there is a lot emphasis on the use of FPGAs and GPGPUs to achieve higher performance for image processing applications. However, our achieved results indicates that the proposed vision architecture provides higher level of programmability, adaptability, and computational efficiency comparing with both FPGAs and GPGPUs. These results can indeed further motivate the investigation and application of emerging highly parallel, low power, SIMD and MIMD architectures for mobile robots and humanoids.

In addition, the studies undertaken and the proposed solutions for parallel formulation and implementation of low, intermediate, and high-level image processing tasks on the CSX architecture along with parallel implementation of event-based vision algorithms on the Tilera architecture can be exploited for implementation of other image processing tasks with similar computational characteristics.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem of robot autonomy

Energy limitation is one of the most important challenges in robotic and especially for mobile robots. Most mobile robots have to carry batteries as their power sources, but batteries are heavy and large to carry and they have limited energy capacity. These days mobile robots are used in many applications, such as floor cleaning, pick up and delivery, entertainment or education, search and rescue. However, applications of mobile robots are limited by their level of autonomy. To achieve a large degree of behavior autonomy, these systems require a significant computational capability to perform a wide variety of tasks, some with real-time constraints, while they are limited in size, weight, and particularly power consumption of their embedded computing platform. Becoming more autonomous, mobile robots are going to be very valuable in more areas, e.g. they can be sent where humans cannot go or do not want to go, or they may be assigned to do dangerous and difficult tasks. Autonomous robots [1] are machines which are able to sense, think, and act, so they must have sensors, computing architectures and actuators. They need sensors to obtain information from the environment, computers to process this information and take decision, and finally, actuators to physically interact with the outside world. All these components consume energy

[1] In robotic, autonomy is defined in relation to human users/designers. It is quite clear that robots are not and will not be any time soon autonomous in the strong biological, autopoietic sense.

and should be considered to achieve a better energy efficiency, and eventually truly autonomous mobile robots.

In this thesis, however, the emphasis is on computation power reduction both in hardware and software level. Computing platform is the essential element of any autonomous robot and consumes a large portion of the robot energy budget. Mei et al. [3] have shown that embedded computer of their mobile robot, called PPRK, singly accounts for 33.3 % up to 65 % of the total power consumption. In addition, to become more autonomous, robots need to have more powerful computing platform which usually means more power consumption. By reducing computing power, robots will be able to perform more complex task with the same energy budget. To reduce computation power for mobile robots and humanoids, we have investigated two complementary solutions: (1) by exploiting new emerging parallel architectures, a low-power high performance computing architecture has been proposed and parallel implementation of different classes of image processing and computer vision applications on this architecture have been provided, and (2) by using emerging bio-inspired sensors such as DVS vision sensor [4], only important data was extracted and processed. In section 1.2, we discuss the approaches followed in this work to reduce computation power for mobile robots in more detail, but before that a short review of major works toward energy autonomy for mobile robots and humanoids is given in section 1.1.1.

1.1.1 Energy autonomy in the literature

Many research works have been conducted in order to cope with energy limitation for mobile robots and prolong their operating time. These works can be categorized in three classes.

One approach is to provide mobile robots with energy autonomy, i.e. developing robots which are able to extract energy from their environment through a self-reliant process. NASA/JPL solar-powered Mars rovers [5] are popular examples of energy autonomous mobile robots. Other successful examples of such robots are EcoBot-I and EcoBot-II developed by Melhuish et al. [6, 7] which mimic animal capabilities of collecting and digesting food. These robots power themselves by converting biomass into electrical energy by using on-board micro-

bial fuel cells. while Ecobot-I [6] was only fed by sugar, Ecobot-II [7] can catch and digest insects. The latest version of EcoBot-III [8] feeds off and is powered by sewage, and operated successfully for seven days in an enclosed environment. However, this technology is still young and these robots have limited energy budget. Therefore, while being fully energy autonomous, these robots are only able to perform simple tasks and show limited behavioral autonomy.

Another approach to build long-lived mobile robots is to enable them gain energy from recharging stations or other robots. Several techniques have been employed to develop autonomous docking mechanisms. The first mobile robots able to dock and recharge were developed by Grey Walter in the late 1940s [9]. These robots, known as tortoises, used a light following behavior to find their way into a hut containing a light beacon and a battery charger. Later on Hada and Yuta et al. [10] reported a week-long repetitive docking experiment with a mobile robot equipped with infrared sensor and reflective tape on the floor to guide the robot to the docking station. On the other hand, a docking station equipped with sensing and communication gadgets can easily guide robots to the station. Silverman et al. [11] and Seungjun Oh et al. [12] have proposed autonomous docking mechanisms using infrared proximity and laser range sensors to guide robot to go back and firmly connect with the station using landing technique of airplanes. Moreover, there are commercial robots equipped with auto-docking mechanisms available in market [13, 14]. Clearly, this techniques puts some demands on the sensing, computation and task scheduling of the robots. An alternative approach for a group of robots working together is to distribute energy among themselves. Zebrowski and Vaughan [15] have proposed the use of a special-purpose energy-transporting tanker robot in a system of autonomous worker robots. The tanker robot is responsible to find and recharge worker robots if demanded, while maintaining its own energy level. Ngo et al. [16] have presented a multi-robot energy-distributing system. In their proposal, robots are not only able of self-recharging energy but also capable to collect refreshed batteries at the charging station and carry batteries to share with other robots. Although these solutions, compared to conventional autonomous recharging schemes, can improve the overall efficiency of a team of robots, they have only been evaluated in simulation. In fact, they are not yet mature enough for real applications.

The last approach, widely followed in the literature, to improve the run-time of mobile robots is to reduce their energy consumption. This is done by considering energy efficiency either in software or hardware design. Most studies on energy reduction have considered motion power. Some studies have considered power consumption in motion control. For instance, Barili et al. [17] presented a method to control the speed of a mobile robot to avoid frequent speed changes and save energy. In another study, Yamasaki et al. [18] presented an energy consumption based control system for humanoid walking, which enables a robot to walk with arbitrary energy consumption. Michaud et al. [19] have proposed an energy efficient locomotion for a solar-powered rover, They also discussed the power management and daily task schedule of the rover based on its power budget. Another way to reduce motion power of a robot is to find the energy-efficient path considering the robot task [20, 21, 22]. Furthermore, there are studies on power management for mobile robots. For instance, Liu et al. [23] have proposed power-aware scheduling algorithm for mission-critical applications such as Mars rover. Mei et al. [3] have also demonstrated the benefit of energy-conservative techniques such as dynamic power management and real-time scheduling to save up robots' power consumption and prolong operating time of mobile robots. The advantage of energy efficient solutions is that first, they can be used in parallel with the other techniques discussed above, and second they allow robots to accomplish more complicated missions with the same energy budget. The work presented in this thesis is also aimed to reduce energy consumption of mobile robots by using energy efficient computing paradigms.

1.2 Research framework

As noted above, mobile robots and humanoids represent an emerging and challenging example of embedded computing applications. In order to achieve a large degree of autonomy and intelligent behavior, these systems not only require a very significant computational capability to perform a wide variety of tasks but they are also severely limited in terms of size, weight, and particularly power consumption of their embedded computing system since they should carry their own power supply. To provide low-power, lightweight, high performance com-

puting architectures for mobile robots and humanoids, two approaches can be considered.

The first approach is to exploit new emerging parallel architectures which provide both high computational capability and low-power consumption. An example is the ClearSpeed CSX massively parallel SIMD architecture which offers 192 Processing Elements with a peak computing power of about 96 GFOLPS while consuming under 9 watts. Another example is the Tilera parallel MIMD architecture which offers 64 general purpose processors with a peak computing power of 192 GOPS while consuming about 20 watts. Combining such architectures provides a heterogeneous light-weight parallel architecture with unprecedented computational capability and low-power consumption. Obviously, for employing parallel architectures in any applications, we should rethink about existing algorithms and design novel parallel ones.

The second solution would be to use bio-inspired information coding and computation. Despite its dramatic progress, information technology has not yet been able to deliver artificial systems that can compare with biology. This in fact, is in part due to a more efficient approach to extract and process important data employed by biological system. Toward this goal, the iCub humanoid robot[24] has been equipped with bio-inspired asynchronous vision sensors called DVS sensors. This sensor loosely models the transient pathway in biological retinas [25]. Unlike standard vision sensors which sample the scene at constant time and therefore produce massive amount of redundant data, the DVS output represents only the relative luminance changes. Consequently, exploiting output of DVS sensor can lead to image processing approaches which are more efficient in terms of bandwidth, memory size and especially computation time. However, to exploit features of this sensor, novel algorithms should be developed which are radically different from conventional image processing.

The work presented in this thesis puts together both complementary solutions mentioned above, to develop a low-power light-weight parallel computing platform for mobile robots and humanoids. On one hand low-power massively parallel architecture have been exploited to provide a small size, low weight, low-power embedded computing architecture for mobile robots. In addition, DVS sensor has been used on heavy computational tasks such as optical flow based navigation to

reduce computational cost. Next chapters describes the parallel and event-based algorithms developed during this research.

In the following section, we recall the current trend in microprocessor design, discuss the potential effect on robotics, and reveiw some state of the art parallel architectures, then in section 1.2.2 we propose a suitable computing platform for mobile robots, and finally in section 1.2.3 the important features of DVS sensor are briefly reviewed.

1.2.1 Trends in microprocessor design

Sequential processing has always been the dominated view in computer science and digital computing technology. However, given the industry's shift to multi-core computing in recent years, parallel computing is attracting growing attention.

From introduction of the Intel 4004 in early 1970's, each generation of processors grew faster and smaller while consuming more power and dissipating more heat. According to Moore's law the number of transistors that could be fabricated on a single chip was doubling each year. IC technology improvements not only was leading to more transistors on one chip but also was providing smaller transistors which operated faster than their predecessors, allowing the processor to run with higher clock frequency. In fact, in those decades, increasing the processor performance was synonymous with increasing frequency. Although smaller transistors consume less power, the number of transistors on one chip was rising faster than the falling amount of power per each transistor. Consequently, processors power consumption was increasing as fast as their performance. However, finally a point in processor design was reached that power-thermal issues, such as heat dissipation, has limited the increase of processor frequency[1]. Given the limited platform power and energy budget, parallel computing was the only way to deliver increased performance. Now, the new trend it to increase the efficiency and number of cores on a chip, to achieve higher performance without a

[1] In 2004, Intel announced it had canceled the development of Tejas, the successor of Pentium 4 because of the heat problem due to the power consumption of the core. From introduction of 8086 through Pentium 4, Intel had increased processor frequency from each generation to the next, but finally it changed its road map to multi-core processing.

corresponding increase in power consumption [26].

Multi-core technology has eventually become the dominant stream in CPU design. Intel and AMD, the mainstream manufactures of microprocessors, are producing processors with several cores for servers and even laptop and desktop environments. Each core also utilizes several Floating point unit. So, instead of raising frequency, they are focusing on increasing cores in one chip. In this manner, performance will increase much faster than power consumption. Besides, some massively parallel processors have emerged such as Cell, the general purpose graphics processing unit(GPGPUs), Tilera, and CSX. These highly parallel and low power architectures along with emerging parallel framework and programming models such as OpenCL and CUDA are the signs of a new era of computing.

As noted in section 1.1, computers are essential elements of robots. The continuous increases in performance, coupled with decreases in size and weight of microprocessors, had major effects on the development of mobile robots [27]. Being equipped with more powerful on-board computers, mobile robots have been able to perform more complex tasks. Attempt to improve the cognitive ability of these robots have been more focused on software level, since each generation of processors brought increased performance on existing applications with no extra effort. However, this is not true anymore unless the applications are written to run in parallel and to scale to an increasing number of cores [28]. In the next decades, we will witness the effect of new trend in processor design on robots and especially mobile robots. While the new multi-core technology can provide mobile robots with new massively parallel architectures, high computational capability and low-power consumption, it brings new challenges to robotics on how to exploit these new architectures. New parallel algorithms must be developed. In addition in parallel computing the right choice of parallel architecture according to characteristic and requirements of the application is very important to achieve the desired performance.

In the following, we first review the architecture of some state of the art parallel processors and then we discuss the choice of parallel architecture for mobile robot applications.

1.2.1.1 Parallel architectures & state of the art processors

Over years of continuous research on computer architectures, various forms of parallel architectures have evolved, and new forms will probably emerge in the future due to worldwide research to discover new architectures that can satisfy the increasing computation demands better than existing architectures. Each type of parallel processors has its own characteristics, advantages and disadvantages and hence is suitable for certain applications.

Given the wide range of parallel architecture types, it is not easy to develop a classification scheme for parallel architectures. In addition, various types of parallel architectures usually have overlapping features to some extents. Consequently, parallel architectures are usually classified under different factors such as, the internal characteristics of processors or processing elements (PEs), the communication mechanism among the PEs, interconnection network among PEs and memory modules, number of instruction and data streams, and even number of PEs.

One of the broad classification of parallel architectures was proposed by Flynn in 1966. Flynn's classification is based on number of simultaneous instruction and data streams which flow among PEs and main memory during program execution. He categorized computer architectures into four main classes described in the following.

- Single Instruction, Single Data Stream (SISD): These computers are actually sequential processors and are not able to perform parallel operations.
- Single Instruction, Multiple Data Stream (SIMD): In this configuration, a single control unit controls multiple execution units. This let the processor to execute one instruction stream on multiple data streams. In other words, one instruction stream is issued by the control unit, and multiple execution units perform the same instruction on different data streams.
- Multiple Instruction, Single Data Stream (MISD): This type of architectures allows multiple instruction stream operate on a single data stream. However, this type is an uncommon architecture and there are not many

examples of MISD architectures. Some pipeline architectures can be viewed as MISD architecture.

- Multiple Instruction, Multiple Data Stream (MIMD): In this configuration, multiple autonomous processors simultaneously execute different instructions on different data. This type of architecture provides more flexibility and consequently is the most common and widely used form of parallel architectures.

Extensions to the Flynn's classification has been proposed in recent years such as single program, multiple data (SPMD) class which combines the ease of SIMD programming with MIMD flexibility. In the following, we quickly review four currently available and more pervasive parallel architectures: Cell, GPGPU, CSX, and Tilera processors. They are very different in their architecture, and therefore each one has its own features, advantages, and disadvantages.

Cell Processor The Cell Broadband Engine (BE) [29] is a heterogeneous multi-core architecture which was jointly developed by Sony, IBM, and Toshiba. The Cell Architecture consists of one PowerPC Processor Element (PPE), eight Synergic Processing Elements (SPE), one Element Interconnection Bus (EIB),and Memory and I/O Controllers.

Two type of cores, PPE and SPE, are specially designed for their tasks and support different set of instructions. PPE, built on IBM's 64-bit Power Architecture, is optimized for control tasks. This is the core which runs the operation system, responds to interrupts, and also distributes the processing work among the SPEs and coordinates their operation, while SPE, 128-bit RISC vector processor, is optimized for data-intensive computing. In other words, each SPE provides data-level parallelism, while combining eight SPEs on one chip brings task-level parallelism.

EIB provides the infrastructure for inter-element communication, memory and I/O requests. The way of connecting elements together is very crucial in term of performance. The EIB consist of four rings, two of them carry data in the clockwise direction and the other two transfer data in the counter clockwise direction.

CSX The ClearSpeed's CSX SIMD architecture [1] has been developed to provide both high performance computing and low-power consumption. Each CSX core has a standard, RISC-like, control unit with instruction fetch, caches and IO mechanisms, and also has two execution units, the mono execution unit, which is dedicated to sequential processing, and poly execution unit.

Poly execution unit includes an array 96 PEs and provides parallel computation. The PEs, similar to other SIMD architectures, operates in a synchronous manner, i.e. all PEs execute the same instruction but on their own piece of data. Each PE includes an ALU, a floating point unit (FPU), an integer multiply-accumulate (MAC) unit, a register file, and private memory. In addition, the poly execution unit includes a programmable IO unit which is responsible for data transfer between external memory and PEs' memories. This unit can works in parallel with computational unit.

Moreover, the PEs are able to communicate with each other via a dedicated bus called *swazzle path*. Swazzle path connects the register file of each PE with the register files of its left and right neighbors, and allows PEs to perform a register-to-register data transfer in either left or right neighbor in each cycle.

Tilera The Tilera [30] MIMD architecture includes a two-dimensional grid of identical processing components called *tiles*. Each tile is a full featured processor with its associated cache hierarchy and a non-blocking switch which connects the tile to on-chip networks. Each tile processor is a 3-wide very long instruction world (VLIW) unit with two or three instructions per bundle, and can run programs independent of the other tiles.

The Tilera three-level cache scheme can store large portion of data and consequently reduce the memory communication time significantly. Each tile has its own L1 cache and L2 cache, while all the tiles' L2 caches are combined to provide on-chip L3 cache. This mechanism is highly effective because if one tile references its own cache and do not find the data, it would be looking for a neighbor may have it. Obviously, that is faster than going off chip to external memory by good ways.

Moreover, the Tilera *iMesh* network, which provides ample on-chip interconnect bandwidth, is responsible for all on-chip data communications. iMesh

consists of six different on-chip interconnection networks, each specialized for a different use; two of them are user accessible and can be used to transfer data between tiles, while four of the six networks are only system accessible. Two system accessible networks are used for data transfer between tiles, and between tiles and external memory, one interconnection network is used to provide cache coherency among tiles' caches, and the last one is used for communications to IO devices.

General Purpose Graphic Processing Unit The graphics processing unit (GPU), first introduced in 1999, has evolved from a fixed-function special purpose graphic processor into a fully-fledged general-purpose parallel processor. In fact, today, GPGPU is the most widely used parallel processor. Over the last decade, the GPU architecture and even the programming interface was changing markedly from generation to generation. Introduction of CUDA programming model in 2006 was a turning point in what today is called GPU Computing. Instead of programming dedicated graphics units with graphics APIs, the programmer could now write C-like programs and target the massively parallel GPGPU processors.

CUDA follows a SPMD programming model. It has introduced a three level hierarchy of threads: a thread, a thread block, and a grid of parallel thread blocks. A CUDA program calls parallel kernels. Each kernel executes across a set of parallel threads which are organized in thread blocks and grids of thread blocks. Each thread within a thread block executes an instance of the kernel, and has its own registers and private memory. Each thread block is a group of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory, and each grid is an array of thread blocks that execute the same kernel, read inputs from global memory, synchronize between kernel calls, write results to global memory [31].

The CUDA hierarchy of threads maps to a hierarchy of processors on the GPGPU. A GPGPU executes one or more kernels; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores in the SM execute threads. GPGPUs are different in number of CUDA cores, SMs, and CUDA cores per each SM. For instance, Fermi, a CUDA-based GPU, includes 512 CUDA cores which are organized in 16 SMs of 32 cores each [32]. Each SM executes threads in

groups of 32 threads called a warp. To make the best use of hardware, threads in a warp should execute the same code path and access memory in nearby addresses (similar to SIMD architectures).

1.2.2 Low-power high performance vision architecture

As noted in section 1.2.1.1, there are a vast number of parallel architecture types and each type is suitable for a certain application areas. Since visual input are primary sensory information for many mobile robots and humanoids, and on the other hand processing visual information usually requires intensive computational capabilities, this work is focused on image processing and computer vision tasks used in robot visual systems.

Conventional image processing algorithms are traditionally classified into three classes: *low-level*, *intermediate-level*, and *high-level*. The low-level image processing algorithms perform the same operation on all pixels of input image to produce the pixels of output image, and hence are the best candidate for data-level parallelism. The intermediate-level image processing algorithms perform operations on the input to produce a more abstract representation like a set of features. These operations are also appropriate candidate for data-level parallelism. The high-level image processing algorithms perform high-level analysis on the abstract data provided by intermediate-level algorithms. These algorithms are suitable to exploit instruction level parallelism due to their irregular structures. However, some high-level image processing tasks such as object classification/recognition include some form of matrix-vector operations which are suitable for exploiting data-level parallelism. SIMD or SPMD architectures such as CSX and GPGPUs are especially designed for data parallel applications. Hence, such architectures are excellent candidates for exploiting data-level parallelism, while to exploit instruction level parallelism, an MIMD architecture like Tilera or Cell processor is appropriate.

Furthermore, as mentioned in section 1.2, DVS sensor provides an asynchronous and sparse representation of the scene which only includes the relative luminance changes(see section 1.2.3 for more detail on how DVS sensor works). Due to this asynchronous and sparse representation, the novel event-based vision

algorithms, developed to work on the output of the DVS sensor, are more suited to be implemented on an MIMD architectures.

Consequently, the proposed supercomputer architecture, in this work, must include a combination of SIMD and MIMD architectures to provide high computing performance in wide range of computations involved in vision system.

On the other hand as already discussed, mobile robots and humanoids require a low-power light weight, but high performance embedded computing platform. While having a high performance computing platform allows them to accomplish more complex tasks, they are severely limited in terms of size, weight, and particularly power consumption of their embedded computing system. Therefore, satisfactory performance and acceptable performance per watt are the two important criteria a proper computing architecture for mobile robots must have. There are various technical factors that affect the overall performance of computing architectures. However, an important measure of computer performance is floating-point operations per second (FLOPS) or operations per second (OPS). Table 1.1 compares some state of the art processors in terms of architecture, performance, and power consumption. According to this table, considering architecture, performance and performance per watt, in other words giga FLOPs (GFLOPs) and GFLOPS per watt of available parallel processors, Clearspeed's CSX700 [1] SIMD architecture and Tilera's Tile64 [30] MIMD architecture are the most appropriate candidate for a low-power lightweight supercomputing architecture for mobile robots and humanoids.

1.2.3 Bio-inspired asynchronous even-based vision sensor

Conventional frame-based visual sensors have two drawbacks; they produce massive amounts of redundant data, and are limited in temporal resolution by their maximum frame rates. Consequently, conventional image processing methods are very costly in terms of computation time, memory usage and communication bandwidth, especially for high frame-rate applications, as they usually operate on the entire image in each frame. In contrast, research in biology/neuroscience has shown that high temporal resolution but sparse acquisition play an important role in the efficiency of the human visual processing system [33]. Inspired

Table 1.1: Multi-core Processors Performance Comparison

Processor	Core	Clock Speed (MHz)	Power (watt)	Performance (GFLOPs)	Performance per watt (GFLOPs / w)
Intel Core i7-965	4	3200	130	69.23	0.53
Tesla^{TM} C2050 *	448	1150	238	515	2.16
PowerXCell 8i	9	3200	92	200	2.17
CSX600	96	250	10	25	2.5
CSX700	192	250	< 9	96	$>$10.6
Tile64*Pro*^{TM} **	64	750	15-22	144	8

* the Kepler GPGPU architecture, introduced to the market in 2012, delivers up to 3x the performance per watt of Fermi architecture according to Nvidia numbers.

** This processor does not include FPU. The performance, therefore, is given in terms of giga OPS (GOPS). It should be mentioned that Tile-GX processor family, the successor of Tile64*Pro* announced in 2012, supports floating point hardware.

by biological evidence and after several attempts [34, 35, 36, 37] a novel visual sensor named Dynamic Vision Sensor (DVS) has been developed by Lichtsteiner et al.[4].

DVS is an asynchronous sensor which works based on temporal contrast in intensity. Hence, the output of DVS sensor has three key properties: it is event-based, sparse, and represents the relative luminance change. DVS includes an array of 128x128 pixels. These Pixels respond asynchronously to relative changes in intensity and generate spike events. Each event indicates a change in log intensity has occurred since the last event in the pixel i.e.

$$|\Delta \log I| > T \tag{1.1}$$

where I is the pixel illumination and T is a global chip threshold. Figure. 1.1 illustrates how one pixel of DVS responses to changes in the intensity. In this figure, the upper diagram represents the logarithm of sensed intensity in one pixel over time and the lower diagram shows the responses of the pixel over time. As can be seen, when the change in log intensity is bigger than the threshold, the pixel generates an event. As shown in Fig. 1.1, each event also specifies the sign of the change. In fact, there are two types of events: on and off which represent increasing or decreasing the intensity, respectively. By directly encoding

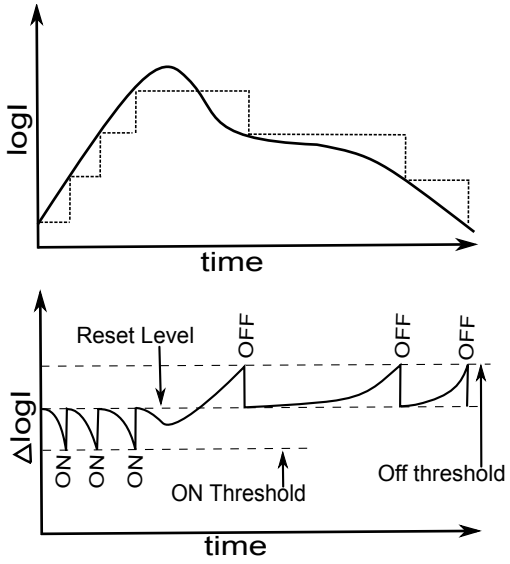


Figure 1.1: Principle of DVS vision sensor operation (one pixel)

scene reflectance changes, DVS reduces data redundancy while preserving precise timing information.

The final output of the sensor is an asynchronous stream of time-stamped address-events (AEs). AEs encode the x and y-address of the pixel in the array, in addition to the type of the event (ON or OFF). Timestamps are generated by a free-running counter outside the main sensor chip. All of the events that arrive within the same clock have the same timestamp On the other hand, since the pixels react independently and then they should send their data via a shared bus, the events which ideally should have the same timestamp may have different timestamps in practice, e.g. when a bar is moving to the right, the events generated by edges of the bar can have different timestamps.

1.3 Robot setup

As discussed in section 1.2.2, the proposed low-power high performance vision architecture includes CSX SIMD and Tilera MIMD architectures. An abstract view of the overall architecture, integrating Tilera, CSX, and a general purpose processor is shown in Fig. 1.2. In our prototype the interconnection network

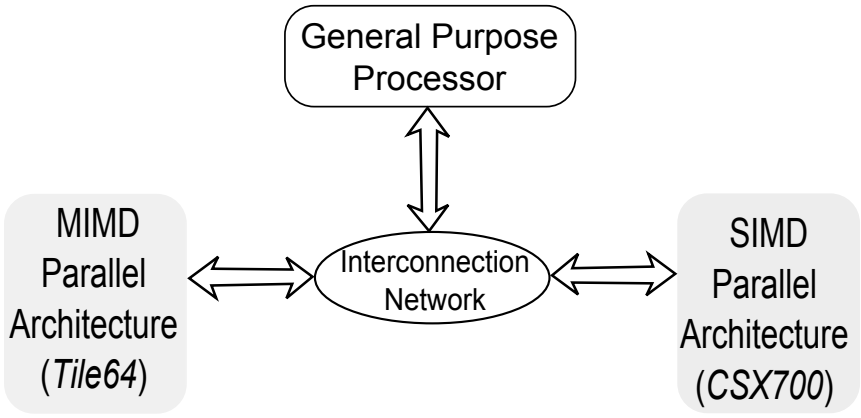


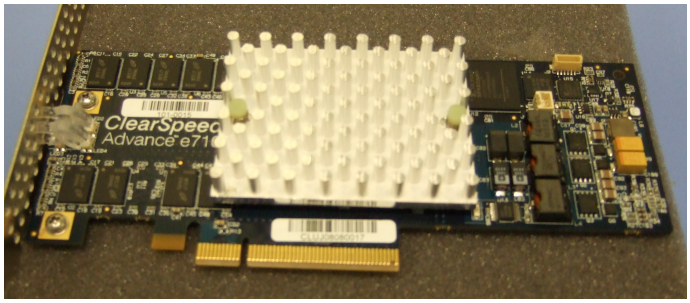
Figure 1.2: Overview of the proposed low-power high performance architecture

Figure 1.3: iCub humanoid robot standing on top of iKart mobile platform

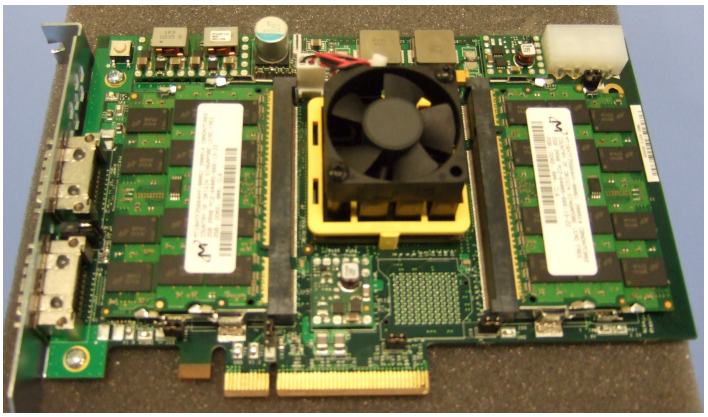
consists of the PCIe buses of the host system which includes the general purpose processor. The estimated power consumption of such architecture is about 50 watt (9 (CSX700) +22 (Tile64) + 20 (GPP) $<$ 50).

To have a prototype of the proposed architecture, we have exploited iKart mobile platform[38]. iKart is a six wheeled mobile platform originally designed to provide autonomous navigation capabilities for iCub humanoid robots. Figure 1.3 shows the iCub robot standing on top of the iKart platform. Moreover, the iKart is augmented by an Intel® Core^{TM} i7 processor which support on-board computation and server repository for the icub software. The platform is also equipped with a wireless bridge and a Lithium ion polymer battery.

Figure 1.4 shows ClearSpeed Advanced e710 Card [39] and Tilera® TileExpress-



(a)



(b)

Figure 1.4: (a) ClearSpeed Advanced e710 Card (b) Tilera® TileExpress-20G Card

20G card [40]. Advanced e710 Card includes one ClearSpeed CSX700 processor and 2 GBytes of ECC-protected DRAM. TileExpress-20G card features a TILE*Pro64* processor with clock frequency of 866 MHz, 8GB of DDR2 Memory with Two 10Gbps Ethernet ports (CX4). Both cards can be installed in a PCIe slot (x8 or x16) of a host machine and exploited as coproccessors. The required drivers are provided by manufacturers for Linux operationg system.

As a rapid working prototype of the proposed vision architecture for mobile robots and humanoids, Advanced e710 Card and TileExpress-20G were mounted on the iKart. Figure 1.5 shows a TileExpress-20G card installed on PCIe slot of the computing platform of iKart.

It should be mentioned that iCub and also iKart are equipped with both traditional RGB cameras and DVS vision sensor. Figure 1.6(a) shows a DVS vision

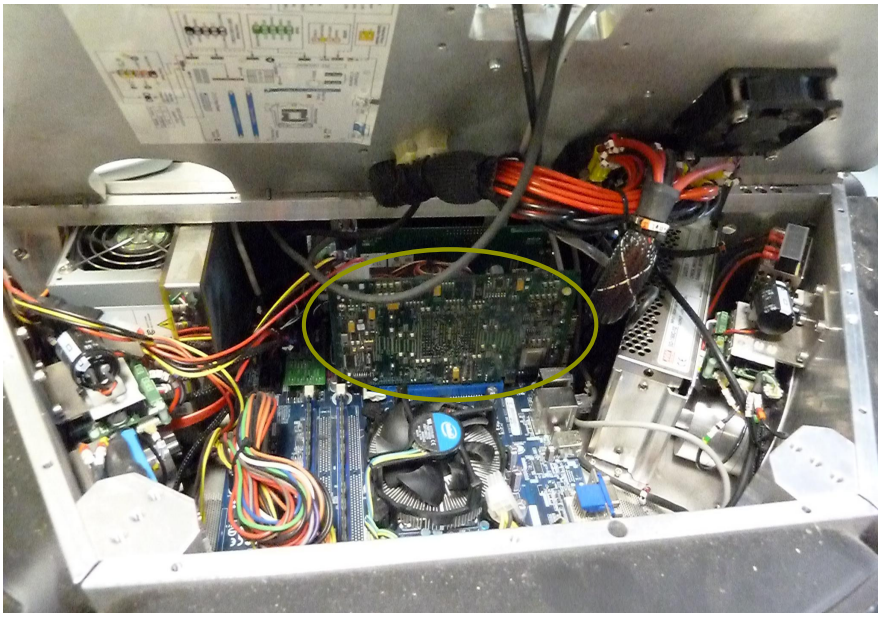


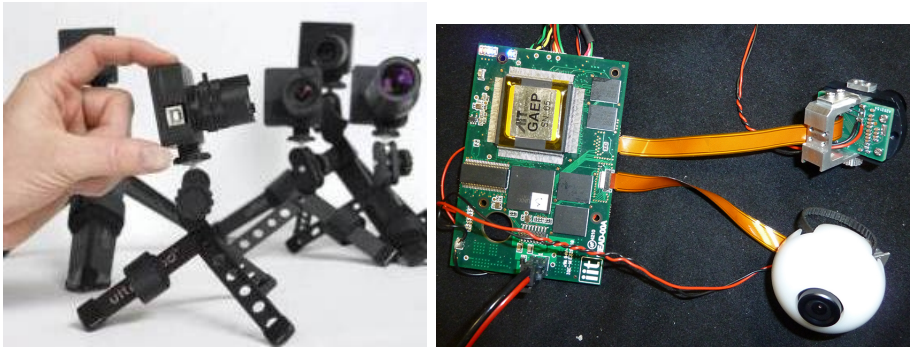
Figure 1.5: TileExpress-20G installed in iKart

sensor with a standard USB2.0 interface that can be mounted on any computation platform. Figure 1.6(b) shows a DVS vision sensor with the associated custom hardware components specifically designed to be installed inside eye balls of the iCub humanoid robots.

1.4 Layout of the thesis

The research undertaken is reported in six chapters.

Chapter 1 discusses the problem of energy autonomy in robotics and reviews some studies undertaken to prolong robot operating time. This chapter introduces the research framework followed in this work to reduce computation power of mobile robots and humanoids. A low-power light weight high performance computing architecture, including CSX SIMD and Tilera MIMD architectures, is proposed. This thesis covers some parallel vision algorithms developed for this architecture. Besides, DVS vision sensor is used. Exploiting output of DVS sensor can lead to image processing approaches which are more efficient in terms of bandwidth, memory size and especially computation time. This thesis also



(a) (b)

Figure 1.6: (a) DVS sensor with USB interface (b) DVS sensor with the associated hardware components designed for iCub eye balls

covers event-based vision algorithms developed to work on DVS output for robot navigation task.

Chapter 2 presents parallel implementation of low-level image processing tasks on the CSX SIMD architecture. It is particularly shown that the row-cyclic is the most efficient data decomposition scheme for data parallel computation of low-level image processing algorithms such as the local stereo vision algorithms and Harris corner detector. In addition, strategies are proposed to minimize parallel processing overhead. The performance of the parallel implementations are reported and discussed. For most cases, faster than real-time performance has been achieved.

Chapter 3 presents parallel formulation and fast implementation of HOG-based human detection on the CSX SIMD architecture. A detailed analysis of multi-level computational structure of the HOG-based object detection is given and it is shown that block level decomposition is the most efficient in terms of reducing the redundancy in the computation and communication. In addition, the mapping of multi-scale computation to 2D *strip packing problem* is described. Computation of HOG descriptors and HOG-based human detection are representative examples of intermediate and high-level image processing tasks, and the the ideas presented here especially for multi-scale computation and dealing with complex data dependency pattern can be easily exploited for parallel implementation of other object detection/classification algorithms. A performance of over

6 fps has been achieved for an image resolution of 640×480 while consuming only 9 watts. This performance can be further increased by deploying multiple CSX architectures.

Chapter 4 presents parallel formulation and massively parallel implementation of an event-based optical flow algorithm on the Tilera MIMD architecture. A hybrid parallel modes is used to achieve an efficient parallel implementation with minimum parallelization overhead on the Tilera MIMD architecture. In high level, the computation is structured in a four stage pipeline, and data decomposition technique is used to further exploit parallelism in computation. There are two important issues that should be considered in parallel implementation of event-based vision applications. First, commonly used data decomposition scheme can lead to unbalance computation, and consequently inefficient parallelization. Second, due to the sparse nature of data, it is more difficult to exploit the cache subsystem of computing platforms effectively, and hence the memory communication overhead may increase. Implementation of the event-based optical flow is provided on both GPP and Tilera architecture, and the experimental results for both implementations are presented. Even our GPP implementation is fast enough to satisfy performance requirements of some applications, and has been successfully used in a real-time obstacle avoidance system.

Chapter 5 proposes two event-based algorithms to estimate the focus of expansion (FOE) and time-to-contact (TTC) for a moving event-based vision sensor. These information are vital in vision-based navigation for mobile robots and humanoids to control movements and especially avoid obstacles. The proposed algorithms exploit the information provided by the event-based optical flow algorithm presented in chapter 3. The FOE is computed by pooling the information provided by all the flow vectors computed in fixed time intervals. Then, FOE estimation along with flow vectors are used to compute TTC. The proposed algorithms have been tested in real world scenarios, and the experimental results are presented. According to the results while estimation of FOE can be pretty accurate if there are enough number of events, the computation of TTC is vulnerable to the errors of optical flow magnitudes.

Chapter 2

Low Level Image Processing ON CSX SIMD Architecture

2.1 Introduction

In this chapter, parallel implementation of low-level image processing algorithms on the CSX SIMD architecture is explored. Two different classes of low-level image algorithms are considered: dense stereo vision and corner detector. These algorithms are usually performed as the first step in many vision-based robotic applications. While the real-time computation of some of these algorithms is possible with standard general purpose processors (GPPs), such an implementation especially for stereo vision algorithms taxes the system and does not leave enough resources for the next steps of computation [41]. These algorithms perform the same operation on all pixels of input image to produce the pixels of output image, and hence are best suited for exploiting data-level parallelism. This chapter extensively discuss the appropriate data decomposition schemes and details the parallel implementations issues on the CSX architecture for the considered algorithms.

In order to achieve real-time performance for low-level image processing tasks, two approaches have been mainly followed in the literature. In the first approach, special-purpose architectures such as ASICs and FPGAs are developed to achieve a better performance. While ASIC and FPGA architectures, due to their low

power consumption, are suitable for embedded applications, their relatively low-level programming model, compared with rather general purpose architectures, is a disadvantage for our target application, mobile robots and humanoids which need to implement a wide class of algorithms. In the second approach, high-performance architectures such as GPGPUs and cell processor are deployed. Although GPGPUs and cell processor have easier programming model, as stated earlier (see section 1.2.2), their high power consumption makes them impractical for embedded applications such as mobile robots and humanoids.

In the rest of the chapter, first a brief but comprehensive overview of the CSX architecture is given in section 2.2. Then, parallel implementation of the stereo vision algorithms and Harris corner detector on the CSX architecture along with the achieved results are presented in section 2.3 and 2.4, respectively. It the end, a summary of this chapter is given in section 2.5.

2.2 The CSX 700 architecture

In this section, we briefly review the ClearSpeed CSX 700 architecture with emphasis on some of its salient features which have been exploited in our implementation (see, for example, [1, 42] for more detailed discussions).

As illustrated in Fig. 2.1, the CSX700 has two similar cores; each core has a DDR2 memory interface and a 128-KB SRAM, called external memory. Each core also has a standard, RISC-like, control unit and two execution units, the *mono execution unit*, which is dedicated to processing mono (i.e., scalar or non-parallel) data, and the *poly execution unit*.

The poly execution unit includes 96 PEs and performs parallel computation (see Fig. 2.2). Each PE has a 128 bytes register file, 6KB of SRAM, high speed I/O channels to two adjacent PEs, as well as external I/O. It also includes an ALU, an integer multiply-accumulate (MAC) unit, and an IEEE 754 compliant floating point unit (FPU) with dual issue pipelined add and multiply, as well as support for floating point division and square root. The computational units within each PE can operate both in parallel and pipelined fashion. However, in order to better exploit these parallel and pipeline capabilities, specific instruction set called vector type operations should be used.

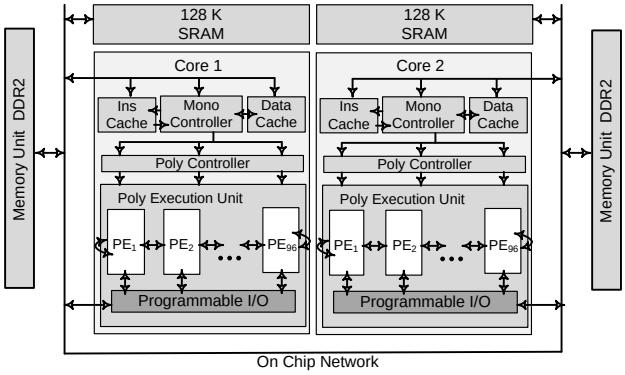


Figure 2.1: Simplified CSX chip architecture [1]

The CSX700 has a clock rate of $250MHz$[43]. Considering the add and the multiply FPUs working in parallel and each generating one result per clock cycle, the peak performance of each PE is then 500 MFLOPS, leading to a peak performance of 48 GFLOPS for one core and 96 GFLOPS for two cores (one chip). However, sequential (i.e., scalar) operations, wherein single add or multiply is performed, take 4 clock cycles to be performed [43]. This leads to a sequential peak performance of 62.5 MFLOPS for each PE, 6 GFLOPS for one core, and 12 GFLOPS for two cores (one chip). This indeed represents a drastic reduction in the peak, and hence, achievable performance. However, vector instructions allow greater throughput for operations. For example, vector instructions that operate on sets of 4 data are executed much faster, e.g, such vector add or multiply instructions take 4 cycles to be completed [43]. However, for some operations the code generated by the CSX compiler might not be optimized to enable achieving such a performance. Therefore, in order to make sure that the best performance is achieved, we have also written part of our codes in CSX assembly language.

Poly execution unit includes a Programmable IO (PIO) unit (Fig. 2.2) which is responsible for data transfer between external memory and PEs' memories, called poly memory. The PIO unit serially transfers data form each of 96 PEs' memories to the external memory and vice versa. As shown in Fig. 2.2, the PIO unit consists of a PIO engine and 96 IO buffers. Each IO buffer is connected to the register file and memory of one PE. In fact, serial data transfer is performed to/from each PE's IO buffer. It is important to note that this architecture enables

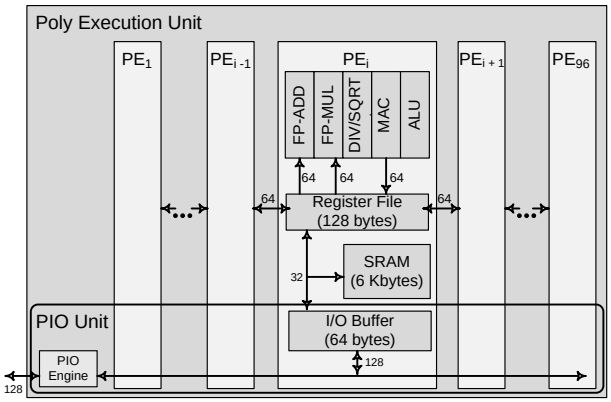


Figure 2.2: Simplified architecture of CSX poly execution unit[1]

the computational units and the PIO unit to work in parallel, thus enabling overlapping of communication with computation. This feature is fully exploited in our implementation to reduce IO overhead. Due to the IO buffer size, at each communication step the maximum size of the data that can be transferred between the external and each PE's memory is 64 bytes. Also, to better utilize the underlying bus bandwidth, the data size has to be at least 32 bytes, i.e. the time required to transfer 32 byte data or less is almost the same.

Moreover, each PE is capable of communicating with its two neighboring PEs by using a dedicated bus called *swazzle path*. As shown in Fig. 2.2, swazzle path connects the register file of each PE to the register files of its left and right neighbors (Note that the boundary PEs, i.e., PE_1 and PE_{96} are also connected to each other, thus forming a ring communication structure). Consequently, on each cycle, PEs are able to perform a register-to-register data transfer to either their left or right neighbor, while simultaneously receiving data from the other neighbor. The swazzle path can work in a pipeline fashion. There are assembly instructions that allow swazzling of sizes up to 32 bytes much faster than calling the swazzle functions on multiple of 8-byte objects. The swazzle path provides the facility for parallel data communication among PEs.

2.3 Parallel implementation of dense stereo vision algorithms

Stereo vision has been extensively investigated and a great variety of algorithms have been developed for its computation [44]. In general, dense stereo vision methods can be categorized into two classes: local and global. In local methods, the disparity map is computed using a winner-takes-all (WTA) strategy, i.e., the disparity of each pixel is calculated without considering disparity assignment of other pixels. In contrast, global methods formulate stereo matching as a global optimization problem. However, for real-time applications, local stereo vision algorithms have been usually considered, due to their rather low computational cost.

An extensive overview of stereo vision algorithms, with emphasis on the application for intelligent vehicles, is presented in [45]. The results in [45] demonstrate that using a local method such as sum of squared differences (SSD) algorithm along with a robust error rejection scheme, such as left-right check (i.e., using both images as reference) and multiple window computation, can lead to the best results. It should be emphasized that, in terms of choice of more accurate stereo vision algorithms for mobile robot applications (which share many commonalities with the intelligent vehicles), we rely on the analysis and benchmarking of the various algorithms reported in [45].

The class of stereo vision algorithms considered for our implementation on the CSX architecture is briefly presented in section 2.3.2. Data decomposition scheme and implementation issues are discussed in section 2.3.3 and 2.3.4, respectively. Finally, the performance of our parallel implementation is presented and discussed in section 2.3.5, but first of all, related work are reviewed in section 2.3.1.

2.3.1 Background and related works

This section briefly review various implementations of local stereo vision algorithms in the current literature.

Implementation of stereo vision algorithms on General Purpose Processors (GPPs) have been studied in [45, 46]. van der Mark et al. [45] have used the sum

of absolute differences (SAD) similarity measure and achieved a performance of 11.38 frame per second (fps) for 512 × 512 images with disparity of 48 by fully exploiting the SIMD computational features (i.e., SIMD SSE2) of a 3.2 GHz Pentium 4 processor. Di Stefano et al. [46] have also used SAD similarity measure and reported a performance of up to 25.94 fps for 320 × 240 image resolution with a disparity of 64 is reported. While the GPP provides the highest degree of flexibility and programmability, the results reported in [45, 46] clearly demonstrate that they cannot provide the adequate performance for larger images of practical interest, such as VGA (640 × 480) or emerging HDTV (1280 × 720) and/or more accurate algorithms.

A parallel implementation of a local stereo vision algorithm on the Cell Broadband Engine has been reported in [47], achieving a performance of 30 fps for VGA images with a disparity of 48. Yang et al. [48] have presented implementation of a local stereo vision algorithm on GPGPUs. Using an ATI Radeon 9,800 graphic card, they have achieved a performance of 13.86 fps for 512 × 512 images with a disparity of 94. Zhu et al. [49] have reported a CUDA-based implementation of SAD algorithm on GPGPU, and achieved a performance of 204 fps for 450 × 375 images with a disparity of 64. GPGPUs and cell processor usually offer a much higher peak computing power and hence can potentially achieve good performances. They also offer programmability (though less than GPPs but more than FPGAs and ASICs) but, as stated earlier, their high power consumption makes them impractical for embedded applications.

Various implementations of stereo vision algorithms have been also considered on special-purpose architectures. Chang et al. [50] have used DSP and achieved a performance of 50 fps for 384 × 288 images with a disparity of 16. Jia et al. [51] have presented an FPGA-based stereo vision system and, with a disparity range of 64, have achieved 30 fps and 120 fps for 640 × 480 and 320 × 240 image sizes, respectively. In [52], an FPGA implementation of a fuzzy hardware structure for disparity map computation has been proposed to achieve 440 fps for 640 × 480 images with disparity of 80. Woodfill et al. [53] have proposed an ASIC chip, called Tyzx DeepSea, and by using a different algorithm, i.e., the census transform, they have achieved a maximum performance of 200 fps for 512 × 480 images with a disparity of 52. In [54], a dataflow hardware architecture has been

developed and a performance of 50 fps for 256 × 192 images with a disparity of 25 has been achieved. In [55], an FPGA implementation of a stereo vision algorithm for automotive applications has been presented with a performance of 425 fps for 320 × 240 images with a maximum disparity of 100. ASICs and FPGAs can be used to design custom hardware and exploit the specific features of the computation to achieve a low-power high-performance implementation. However, they are not adequate for our objective due to their relatively low level programming model.

2.3.2 Overview of target stereo vision algorithms

In this section, we briefly describe the class of stereo vision algorithms considered for implementation on CSX architecture.

SSD algorithm

The SSD algorithm is a straightforward window-based approach to obtain the disparity map on a pair of rectified stereo images. To describe the algorithm, let $I_R(i,j)$ and $I_L(i,j)$ denote the intensity of pixels located at row i and column j in the right and left images, respectively. The input parameters of the algorithm are w the window size and β the maximum disparity. Assuming the right image as reference, the disparity for each pixel (i,j) in the right image is calculated as follows:

- Consider a window centered at (i,j) in the right image
- Consider a window centered at $(i,j+k)$ in the left image where $j \leq k < j+\beta$
- Calculate convolution of the windows in the left and right images as

$$S(i,j,k) = \sum_{l=i-\frac{w-1}{2}}^{i+\frac{w-1}{2}} \sum_{m=j-\frac{w-1}{2}}^{j+\frac{w-1}{2}} \Big[I_R(l,m) - I_L(l,m+k) \Big]^2 \qquad (2.1)$$

- The pixel that minimizes $S(i,j,k)$ is the best match. So,

$$k^* = \arg \min_{j \leq k < j+\beta} S(i,j,k), \tag{2.2}$$
$$d(i,j) = k^*$$

Briefly, the SSD algorithm consists of the following three steps:

1. calculating the squared differences of intensity values for a given disparity
2. summing the squared differences over square windows
3. selecting the minimal SSD value at each pixel.

SSD with multiple windows

To reduce the errors introduced by depth discontinuity, the use of multiple correlation windows with the same size and different centers have been proposed [56]. For example, by using a 5-window scheme, the disparity for each pixel (i,j) in the right image is calculated as follows:

First, convolution of the windows in the left and right image are calculated by using Eq. (2.1). Then, the two lowest SSD values of neighboring windows are added to the $S(i,j,k)$:

$$\begin{aligned} min_1 =& \min \Big\{ S(i-h, j-h, k), S(i+h, j-h, k), \\ & S(i-h, j+h, k), S(i+h, j+h, k) \Big\} \\ min_2 =& \text{second} \min \Big\{ S(i-h, j-h, k), \\ & S(i+h, j-h, k), \\ & S(i-h, j+h, k), S(i+h, j+h, k) \Big\} \\ S_5(i,j,k) =& S(i,j,k) + min_1 + min_2 \end{aligned} \tag{2.3}$$

where h is the correlation window half width, i.e. $w = 2h+1$, and $S_5(i,j,k)$ refers to image row i, image column j, and disparity k, by using 5 correlation windows.

Finally, using WTA technique, the pixel that minimizes $S_5(i, j, k)$ is the best match, i.e.

$$k^* = \arg \min_{j \leq k < j+\beta} S_5(i, j, k), \tag{2.4}$$
$$d(i, j) = k^*$$

In [56] other schemes by using 9 and 25 correlation windows have also been suggested.

SSD with left-right check

While using WTA techniques, a post processing step is needed to remove errors caused by occlusions. To detect and remove such errors in the estimation, left-right consistency check which exploits uniqueness constraint is often used [45]. In this technique, first both left to right and right to left disparities are calculated. Then for each point the corresponding results are compared. It is assumed that the results of left to right and right to left should be the same. Different results indicate an inconsistency, which might have been introduced by occlusion, and hence these results are removed. Left-right check technique lead to a significant increases in computation cost, as it needs the disparity map to be computed twice.

2.3.3 Data decomposition

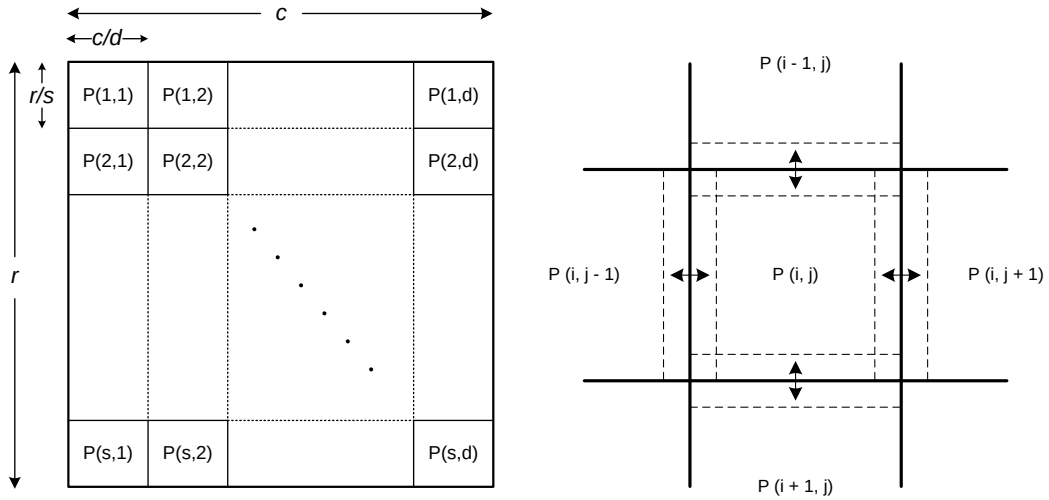
Data decomposition is the key factors for efficient implementations of data parallel algorithms. Data decomposition scheme has a direct impact on the required communications to external memory, the amount and distance of data exchanges among processors, and the required local memory space. In the CSX architecture, the size of PE's memory is rather limited and it can only store small segments of data. Consequently, PEs might need to receive data from external memory or other PEs. Data transfer from external memory to PEs' memory (poly memory) is much more expensive than inter-PE communication via swazzle path [1]. In fact, the cost of inter-PE communication among neighboring PEs is the same as simple arithmetic operations and hence should be exploited as much as possible.

For the CSX architecture, we analyze and compare various data decomposition schemes in terms of the following parameters: (a) required memory space for each PE, (b) size of data that needs to be transfered to PE's memory more than once, and (c) inter-PE communication time.

Having an image and a linear array of PEs, several data decomposition schemes can be employed including row (column)-stripe decomposition, block decomposition, and row (column)-cyclic decomposition [57]. Here, we analyze these schemes for parallel implementation of the SSD algorithms on the CSX architecture. In the following, c and r denote the number of columns and rows in the image, respectively. Also, w, β, and p indicate the SSD window size, the maximum disparity, and the number of PEs, respectively. In every memory communication, each PE reads or writes m bytes of data from/into the external memory. Finally, M_m is the memory needed to calculate SSD values for m pixels.

Block decomposition Block decomposition scheme is illustrated in Fig. 2.3(a). The image is divided into $p = d * s$ blocks, with each block having c/d columns and r/s rows. The first block is assigned to the first PE, the second one to the second PE, and so on. Each block is denoted by an ordered pair (i, j) where $1 \leq i \leq s$ and $1 \leq j \leq d$. In the same way, each ordered pair also denotes the PE that is responsible for processing of the corresponding block. In the following $P(i, j)$ refers to $\text{PE}_{(i-1)s+j}$.

Figure 2.3(b) depicts the boundary data needed for computation by $P(i, j)$ and its four immediate neighbors. To handle boundary data, needed by two neighboring PEs, there are two possible alternatives: first,transferring boundary data from external memory to both PEs, hence performing redundant data communication, or second, transferring to one PE and then using swazzle path to transfer it to the other PE. To process first rows and also columns, $P(i, j)$ requires the last rows and columns of $P(i-1, j)$ and $P(i, j-1)$, respectively. However, These PEs have not yet received data that $P(i, j)$ needs. Therefore, if the swazzle path is used then $P(i, j)$ should skip processing of the boundary data, until $P(i-1, j)$ and $P(i, j-1)$ provides the required data. Also, for processing the last rows and columns of data, $P(i, j)$ needs data which has already been sent to $P(i, j+1)$ and $P(i+1, j)$, respectively. For these PEs to provide the boundary data to



(a) (b)

Figure 2.3: (a) Block decomposition (b) Boundary data for each PE in block decomposition. P(i, j) refers to $PE_{(i-1)d+j}$

$P(i, j)$, they need to store this part of data in their memory which is a limited resource. The choice of PEs receiving the boundary data form external memory or via swazzle path depends on the trade-off between the required PE memory space and the cost of external memory communication. It should be noted that by using the block decomposition scheme on the CSX architecture, the distance between $P(i, j)$ and $P(i + 1, j)$ is equal to d, i.e. to exchange data between these PEs, the data must pass through d PEs.

Row-stripe decomposition Figure 2.4(a) illustrates the row-stripe decomposition. The rows are divided into several groups, each has r/p rows. Then, the first group is assigned to the first processor, the second one to the second processor, and so on.

The boundary data for PE_i is shown in Fig. 2.4(b). Boundary data is transfered from external memory to both PEs or from one PE to another via swazzle path. To process the first rows, PE_i requires last rows of PE_{i-1}, and to process its last rows, PE_i needs the first rows of PE_{i+1}. Using swazzle path, PE_i should skip processing the first rows, until PE_{i-1} receives its last rows. Also, PE_{i+1} should store data related to its first rows in its memory. In fact, like block

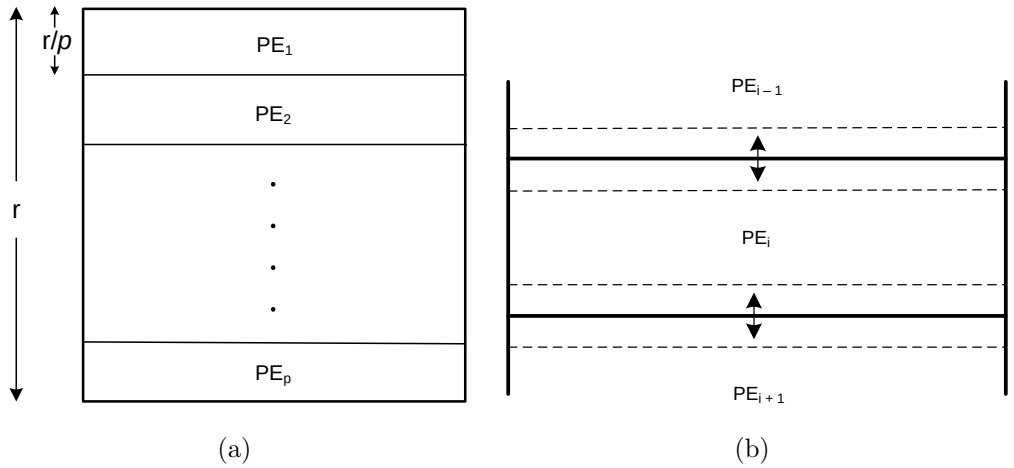


Figure 2.4: (a) Row-stripe decomposition (b) Boundary data for each PE in row-stripe decomposition

decomposition, neighboring PEs do not process data located in their boundary concurrently. Consequently, PEs may need data before their neighbors receive it, or data should be stored in limited memory of PEs.

Row-cyclic decomposition In this scheme, the first row is assigned to the first processor, the second row to the second processor, etc. Since one row is assigned to each PE, each PE needs to communicate with the PEs which are at most at the distance of $(w-1)/2$, as shown in Fig. 2.5. Here, each PE needs data just after its neighbor has finished processing that same data. So, swazzle path can be utilized without using extra poly memory space.

2.3.3.1 Comparison of data decomposition schemes

The parameters calculated for each data decomposition scheme are summarized in Table 2.1. As can be seen, block and row-stripe decomposition schemes require either more PE memory space or more redundant external memory communications. Interestingly, though the block decompositions scheme might seem a natural choice, it is further inefficient since it requires non nearest neighbor communication among PEs. Because of efficient usage of swazzle path and limited PEs' memory, row-cyclic decomposition scheme is the most efficient for imple-

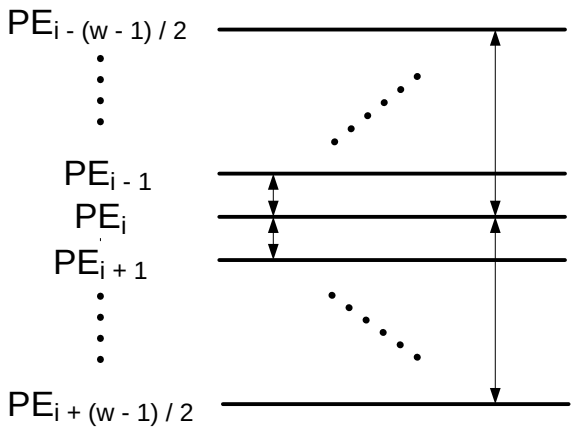


Figure 2.5: Boundary data for each PE in row-cyclic decomposition

menting SSD algorithm on the CSX architecture.

2.3.4 Proposed parallel implementation

In this section, we discuss parallel implementation of the SSD algorithms on the CSX architecture, based on the row-cyclic data decomposition scheme as discussed in the previous section. In order to efficiently utilize both cores of the CSX architecture, the input images are divided into two nearly equal parts. The first $\lceil r/2 \rceil + (w-1)/2$ rows are assigned to the first core, and the last $\lfloor r/2 \rfloor + (w-1)/2$ rows are assigned to the second core. Sending boundary lines to both cores represents a small overhead but enables each core to perform all computation locally and thus resulting in a near perfect speedup of two. In the following, the details of various implementations on a single core are discussed.

2.3.4.1 SSD algorithm

Outer and Inner Loop Iterations As mentioned in section. 2.2, each CSX core includes 96 PEs. To apply the row-cyclic decomposition scheme for computation, the input images are divided into groups of 96 rows and the computation is performed in several iterations (sweeps), denoted as *Outer Loop Iterations*. In each iteration, operations are performed on a group of 96 rows. To handle boundary conditions, two consecutive iterations are overlapped. For example, in the

Table 2.1: Figure of merit for different data decomposition schemes for SSD

Data decomposition	Boundary data	Redundant external memory Comm.	Inter-PE** comm.	PE memory space
Block	S *	-	$s\beta w + d^2\beta W$	$Q + m\beta W/2 + \beta$
	M	$rd(\beta + 2W) + 2csW$	-	Q
Row-stripe	S	-	$cs\beta W$	$Q + m\beta W/2$
	M	$2cpW$	-	Q
Row-cyclic		-	$rc\beta W$	$Q - M_m W$

In this table, $Q = 2m + \beta + wM_m$ and $W = (w - 1)$

* S indicates that boundary data is shared between PEs by using the swazzle path. M indicates that boundary data is transferred from external memory

** In calculating number of inter-PE communications for block and row-stripe decompositions, it is assumed that the w is smaller than number of rows which are assigned to the PEs.

first iteration, PE_{96} performs operations on row 96. However, the results are not correct since it does not have access to row 97 which is needed for its computation. To handle this boundary condition, rows 95 and 96 are also considered in the second sweep of rows. That is, in the first outer loop iteration, the results are correct for rows 1 to 94 and not for rows 95 and 96. This redundant calculation represents a small overhead, but it allows regular operations to be performed by all PEs.

In order to reduce memory communication overhead, computation and communication should be overlapped. To achieve maximum overlapping, each row is divided into a set of segments of size m. The computation for each row is then performed in several iterations (sweeps), denoted as *Inner Loop Iterations*. In each inner loop iteration, the computation is performed for a segment of data, i.e., m pixels. We have chosen $m = 32$ for two reasons. First, this size enables maximum utilization of the CSX bus bandwidth (see section 2.2). Second, it allows the best overlapping of the computation and communication (see below).

Memory Communications Pattern Figure 2.6 shows memory communication pattern for computation of one outer loop iteration consisting of $\lceil c/m \rceil$ inner loop iterations. In each inner loop iteration, the PIO unit (see section 2.2) trans-

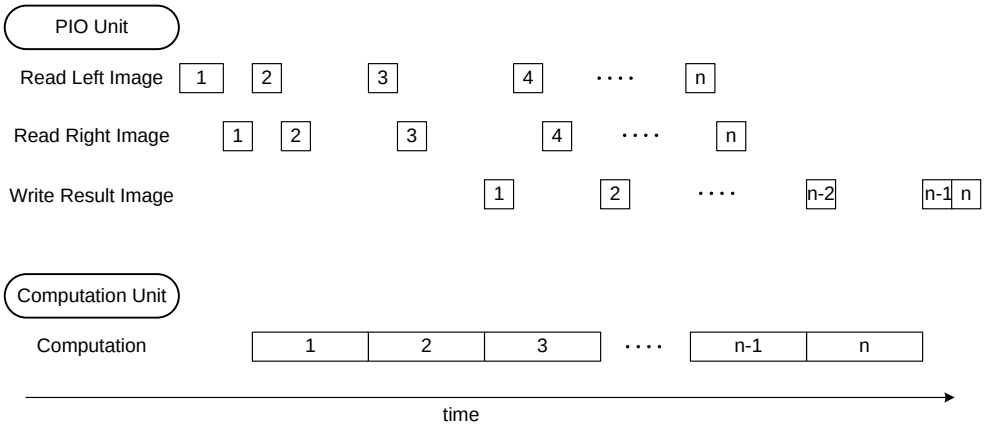


Figure 2.6: Parallelized SSD: communication pattern

fers two sets of 96 data segments, one set from right image and the other set from left image, from external memory to 96 PEs memory. Also, the PIO unit transfers 96 segments of the result from 96 PEs' memory to external memory. Data are transferred serially while the PEs process the received data in parallel. As shown in Fig. 2.6, in the initial phase, PEs are idle and PIO unit copies the first set of segments of data from external memory to PEs' memories. However, for processing of each pixel of the right image, β (maximum disparity) pixels ahead in the left image are also needed. Therefore, only in the initial phase $m + \beta$ pixels of the left image are transfered to PEs. At the end of this initial phase, all PEs have received the required data and start the processing of the first set of data segments while PIO unit start transferring the second set of data segments from external memory into 96 PEs memories. PEs start processing of the second segments, as soon as they have completed the computation of the first segments. While PEs process second data segments, PIO unit transfers the third segments of data from external memory to 96 PEs memories and transfers the first segment of the results from 96 PEs to external memory. This pattern continues till all the segments are processed. Finally, when the computation for one row is completed the PEs become idle and the PIO unit transfers the last segments of result to the external memory.

With our choice of m and by using this data communication pattern, transferring input data segments to and reading the results from 96 PEs' memory take

less time than processing one data segment. In fact, the PEs' computation units are busy all the time and therefore the communication time is fully overlapped with the computation.

Computation Steps Sequential SSD algorithm has three steps (see Sect. 2.3.2): evaluating square of differences, summing the squared differences over SSD windows, and selecting the minimum. Using the row-cyclic data decomposition scheme, each PE receives the data of one line of the input images and is responsible to calculate the same line of the output. Step 1 computes square of difference. The computation of Step 2 is divided into two sub steps: first, calculating summation over one line using local data, and then receiving the results of two neighboring PEs and calculating sum over the window. Finally, for each pixel, the minimum value among calculated SSD values is obtained. Thus, the parallelized SSD has four steps as follows: computing square of differences, summing SSD values over one line of the window (local-sum), summing SSD values over windows, and finally selecting the minimum. Algorithm 1 shows the computation steps of parallelized SSD for one segment of data.

As discussed in section 2.2, in order to achieve a better performance, the pipeline capability of PEs should be exploited as much as possible through vectorization of the computation. In section 2.3.5.1, the effect of vectorization on each step of the computation is presented.

2.3.4.2 Multiple window algorithm

SSD with multiple window is implemented the same as SSD with one window, just in the last step more inter-PE communications and comparison operations are needed to derive the final result. Since here more inter-PE communications are needed, and using the row-stripe data decomposition scheme decreases the inter-process communications, the question then raises as "is the row-cyclic data decomposition scheme still the best strategy?". In fact, the same argument as in section 2.3.3.1 can be applied here. Using 5 windows, the number of inter-PE communications increases four times for both row-stripe and row-cyclic schemes. Nevertheless, considering limited memory of PEs and the fact that in the CSX architecture, inter-PE communication takes just 2 cycles, row-cyclic decomposition

Algorithm 1 Parallelized SSD: Computation Steps

calculate_ssd(I_R, I_L, Res, m, d, w)
I_R: Segment from right image
I_L: Segment from left image
Res: Segment of the output
m: size of data segment
d: maximum disparity
w: window size

Step 1: Compute square of differences
  for i = 1 to m
    for j = 1 to d
      $SD[i,j] = (I_R[i] - I_L[i+j])^2$

Step 2: Sum the squared differences over lines
  for j = 1 to d
    for i = 1 to m
      $SSD[i,j] = SSD[i-1,j] - SD[i - \frac{w+1}{2}, j] + SD[i + \frac{w-1}{2}, j]$

Step 3: Compute SSD values over windows
  for i = 1 to m
    for j = 1 to d
      $up\_value = down\_value = SSD[i,j]$
      for k = 1 to $(w-1)/2$
        $up\_value = swazzle\_down(up\_value)$
        $down\_value = swazzle\_up(down\_value)$
        $SSD[i,j] = SSD[i,j] + up\_value + down\_value$

Step 4: Select the minimum
  for i = 1 to m
  $min\_value = SSD[i,1]$
  $min\_idx = 0$
  for j = 2 to d
    if $SSD[i,j] < min\_value$
      $min\_value = SSD[i,j]$
      $min\_idx = j$
  $Res[i] = min\_idx$
end

is still the best strategy.

As mentioned in Sec. 2.3.2, by using five windows, for each pixel and each disparity, the two lowest values among four neighboring window should be found. To achieve better performance it is necessary to reduce the number of operations as much as possible. To this end, the two lowest values can be found by using only 3 comparisons as follows. For pixel x the SSD values of pixels $x-h$ and $x+h$ are compared, where $w = 2h+1$. Then, each PE sends $\min\{(x-h),(x+h)\}$ and $\max\{(x-h),(x+h)\}$ to its two neighbors. Now, each PE has the minimum and maximum of up and down values. The two minimums can be found by comparing the maximum of up hand with minimum of down hand and vice versa.

2.3.4.3 Left-right check algorithm

Since CSX700 has two similar cores, task parallelism might seem to be the best approach for implementation of the left-right check algorithm. In this approach, one core performs the left to right search and the other one performs the right to left search and then the results are compared. Another possible approach is to use data parallelism as before, i.e. input images are divided between two cores, and each core performs both left to right and right to left search and subsequent comparison and produces the final results. The SSD values which should be computed for both left to right and right to left are the same. Therefore, performing left and right searches concurrently enables us to use the results of already computed SSD values. Moreover, in this approach the computation intensity increases. That is, once a part of data is read, all the computations are performed and the results are written back to the memory. However, in the former approach each part of data is read and write from memory 3 times. Moreover, each core should access the other core's memory which is more expensive. Thus, the second approach based on data parallelism provides a better performance. The only issue is that, due to the PEs' memory size limitation, the memory should be managed efficiently to keep all necessary data for doing both right to left and left to right searches simultaneously. We have implemented both approaches and the results, discussed in section 2.3.5.3, proves the better efficiency of the data parallelism approach over the task parallelism one.

2.3.5 Results and performance of parallel implementations

We have implemented the following algorithms on the CSX architecture:

- SSD: left to right search
- SSD_MV5: left to right search, SSD with 5 windows
- SSD_LR_TP: SSD with left-right check implemented according to task parallelism
- SSD_LR_DP: SSD with left-right check implemented according to data parallelism

The computation cost of the algorithms, in terms of number of operations per pixel and using 3×3 windows, is shown in Table 2.2. The computation time of the implemented algorithms for image sizes of 640×640 and 1280×720 are summarized in Table 2.3. Another considered performance measurement is the sustained number of FLOPS. The sustained GFLOPS and GFLOPS per watt of various implementations are presented in Table 2.4. The sustained GFLOPS results are obtained by dividing the number of operations of each algorithm by its computation time. The sustained GFLOPS/Watt is obtained by considering the maximum power consumption of 9 Watts for CSX architecture.

As can be seen from Table 2.3, we achieve real-time, and for most cases even faster than real-time, performance for all considered problem instances except for 1280x720 images with disparity of 32, for which only SSD algorithm achieves the real-time performance. However, for this case the real-time performance can still be achieved by using two CSX architectures and dividing the image, as it was done for two-core implementation.

Another important observation is that the computation time of different algorithms closely correlate with their computation cost in terms of the disparity range. In fact, as our practical results demonstrate the computation time almost linearly increases with the disparity range, e.g. the computation times almost double for a factor of 2 increase in the disparity range. This close correlation is due to the fact that the disparity range directly affects the amount of the

Table 2.2: Number of operations per pixel by using 3x3 windows

Algorithm	Number of operations
SSD	7β
SSD_MW5	12β
SSD_LR$_{DP}$	$8\beta + 1$
SSD_LR$_{TP}$	$7\beta + 1$

computation performed by each PE. The disparity range has a small impact on the initial phase of data transfer (see section 2.3.4.1) wherein $m + \beta$ pixels are needed to be transferred. However, as will be discussed below, the overall communication overhead is very small and hence increasing β will not significantly affect the overall performance. As shown in Table 2.2, the computation cost is a linear function of the maximum disparity, β, and consequently, by increasing the maximum disparity, the performance would decrease almost linearly.

Note that the sustained GFLOPS is a function of image size. That is, it is a function of the number of idle PEs in the last iteration of outer loop iterations. For example, for 1280x720 and 640x480 images the number of the idle PEs in the last outer loop iteration is 16 and 42, respectively. Therefore, due to a better utilization of PEs a better GFLOPS for 1280X720 images is achieved.

In the following, a more detailed analysis of performance and implementation issues of each algorithm is given.

2.3.5.1 SSD algorithm

The computation time of different steps of the implemented parallel SSD algorithm, for 640×480 input images with a disparity range of 16, is presented in Table 2.5. To better analyze the performance of different steps of the algorithm and particularly the memory communication overhead, we have also used the CSX visual profiler tool [58]. The tool is able to log the start and finish time of all the events which occur in one core such as poly computing, accessing to mono or poly memory, data transfer between mono and poly memory, etc. Fig. 2.7 shows the initial part of the log file related to the initial memory communication. As can be seen, while reading the first segment of data, the PEs are idle (in Fig. 2.7

Table 2.3: Computation time of the implemented algorithms on CSX700 architecture

Image	Algorithm	Latency (ms)		fps	
		$\beta = 16$	$\beta = 32$	$\beta = 16$	$\beta = 32$
640×480	SSD	5.57	10.56	179	94
	SSD_MW5	12.5	24.51	80	40
	$\text{SSD\_LR}_{\text{DP}}$	11.05	21.07	90	47
	$\text{SSD\_LR}_{\text{TP}}$	11.84	21.62	84	46
1280×720	SSD	14.72	28.04	67	35
	SSD_MW5	33.2	65.18	30	15
	$\text{SSD\_LR}_{\text{DP}}$	29.51	56.58	33	17
	$\text{SSD\_LR}_{\text{TP}}$	31.31	57.76	31	17

Table 2.4: Sustained GFLOPS and sustained GFLOPS/Watt on CS700 architecture

Image	Algorithm	GFLOPS		GFLOPS/watt	
		$\beta = 16$	$\beta = 32$	$\beta = 16$	$\beta = 32$
640×480	SSD	6.17	6.51	.685	.723
	SSD_MW5	4.71	4.81	.523	.534
	$\text{SSD\_LR}_{\text{DP}}$	3.58	3.74	.397	.415
	$\text{SSD\_LR}_{\text{TP}}$	2.93	3.19	.325	.354
1280×720	SSD	7.01	7.36	.778	.817
	SSD_MW5	5.32	5.42	.591	.602
	$\text{SSD\_LR}_{\text{DP}}$	4.02	4.18	.446	.464
	$\text{SSD\_LR}_{\text{TP}}$	3.32	3.59	.368	.398

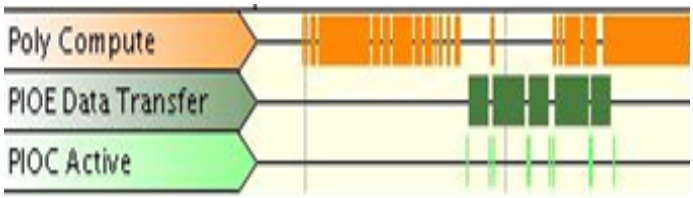


Figure 2.7: Running Profile of Parallelized SSD on CSX processor related to initial memory read. Poly Compute shows the time the poly execution unit(PEs) is busy. PIOE Data Transfer indicates data transfer between PEs' and external memory. Receiving the first data segment, PEs start computation, and computations and communications are overlapped

PIOE Data Transfer is active and *Poly Compute* is inactive). Upon receiving the first data segment, computation starts and the subsequent memory communications are fully overlapped with the computation. (both *PIOE Data Transfer* and *Poly Compute* are active). The output of the visual profiler is consistent with the scheme and model illustrated in Fig. 2.6 and discussed in section 2.3.4. It shows that ignoring the initial and final phases, the PEs (poly execution unit) are continuously performing the computation and never become idle, waiting for data from external memory. Furthermore, as can be seen from Table 2.5, the initial and final memory communications take only 16.3 μs, representing just 0.3% of the total computation time.

Such a small overhead clearly demonstrates the efficiency of our parallel implementation, i.e., the fact that the computation is fully parallelized with a minimum of overhead. Therefore, any further improvement in the computation can be only achieved by increasing the speed of PEs in the computation by employing an efficient vectorization strategy. The details of improvement achieved by vectorization of the computation are presented in Table 2.5. Note that, for some parts of the computation, we have directly developed our own assembly code to achieve a better performance since the CSX compiler does not always generate the most optimized assembly code. In particular, a more efficient use of the swazzle path for inter-PE communication can be achieved by directly using the assembly language of the CSX. As can be seen from Table 2.5, by using vectorization and assembly code, the total time of computation has been reduced from 9.6 ms to 5.57 ms, representing about 40% reduction.

Table 2.5: Computation time of parallelized SSD steps (optimized and non-optimized code) for images of 640 × 480 resolution and disparity range of 16

Step	Timing (ms)	Optimized Timing (ms)
Initial & Final Overhead	0.0163	-
Square-of-differences	3.36	1.16
Sum(over line)	1.974	0.88
Sum(over window)	2.27	1
Selecting the Min	2.26	-
Total	9.6	5.57

Table 2.6: Comparison of Computation time of SSD and SSD with multiple window on the CSX700 architecture for images of 640 × 480 resolution and disparity of 16

Step	SSD (ms)	SSD_MW5 (ms)
Square-of-differences	1.16	1.16
Sum(over line)	0.88	0.88
Sum(over window)	1	1
Selecting the Min	2.26	8.92
Total	5.57 ms	12.5 ms

2.3.5.2 Multiple window algorithm

Multiple window selection is implemented in 4 steps. Steps 1, 2, and 3, in which the SSD values are computed, are exactly the same as the single window implementation. In step 4, more swazzling and comparison operations are required. Due to the sequential nature of these operations, it is not possible to use vector instructions. Table 2.6 compares the computation time of SSD and SSD_MW5. We have observed that in the code generated by the CSX compiler, there are a lot of unnecessary move instructions in the functions for calculating the two lowest value of four SSD value of neighboring window. By writing these functions in assembly, the computation time was decreases from 16.84 ms to 12.5 ms, representing about 25% reduction.

2.3.5.3 Left-right check algorithm

We have implemented left-right check by using two approaches mentioned in section 2.3.4.3. The computation times of these two approaches for two image sizes are depicted in Table 2.3. The results prove that dividing image between two cores and doing left to right and right to left search on the same core lead to a slightly better performance.

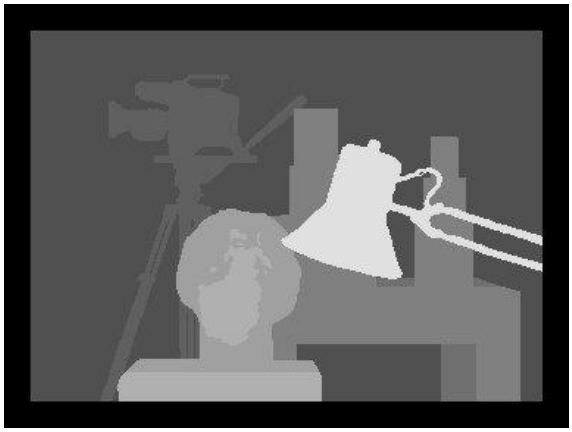
2.3.5.4 Disparity map output

In our work, we have used the same algorithms implemented by other researchers such as [44] and [45] and proposed parallel implementation of these algorithms on CSX SIMD architecture. The results of our parallel implementation is identical to the sequential ones. i.e., to achieve these frame rates, we have not made trade-offs between speed and accuracy in our parallel implementation. However, to evaluate the accuracy of the proposed approach, we used the Middlebury stereo vision benchmark [59]. The ground truth of Tsukuba image [59], along with the depth maps computed by using our implementation is presented in Fig. 2.8. Also, Table 2.8 compares accuracy of our implemented SSD with the literature in terms of the error rate calculated by Middlebury webpage for Tsukuba image.

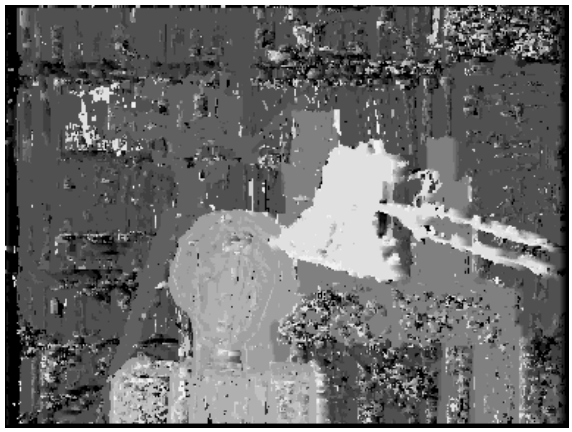
2.3.5.5 Comparison with published results

In this part, we compare performance of our implementation with previous implementation of WTA techniques for dense stereo vision, in terms of achieved performance and performance per Watt. Since various works have reported their results considering different image resolutions and maximum disparity range, the performance of each implementation is measured in terms of millions of disparities evaluated per second (MDEs).

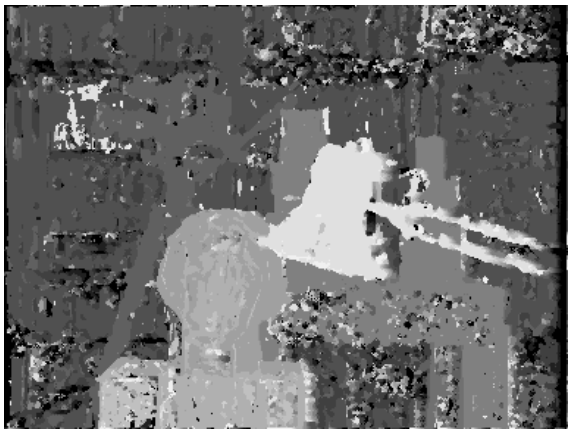
Table 2.7 compares the performance of the three implemented algorithms on the CSX700 and a Pentium 4 processor with a clock frequency of 3.2 GHz [45]. In [45], implementation of three algorithms, SAD, SAD with five multiple window, and SAD with left-right check by fully exploiting the SSE SIMD instructions set are discussed. As Table 2.7 shows, our implementation achieves a much



(a)



(b)



(c)

Figure 2.8: Tsukuba image (a) ground truth disparity map (b) depth map built from SSD (c) depth map built form SSD_MW5

Table 2.7: A comparison between the stereo vision implementations on the CSX700 and Pentium 4 processor with clock frequency of 3.2 GHz

Author	MDEs	Algorithm
This work	1,032	SSD
van der Mark [45]	143	SAD
This work	442	SSD_MW5
van der Mark [45]	75	SAD_MW5
This work	501	SSD_LR$_{DP}$
van der Mark [45]	114	SAD_LR

better performance than CPU implementation even with utilizing SSE SIMD instructions set.

Table 2.8 compares the performance of our SSD implementation on the CSX architecture with the current literature in terms of MDEs and MDEs per Watt. As also shown in Table 2.8, stereo vision algorithms have been implemented on different platforms which could be classified into two main categories: implementation on rather general purpose architectures and rather special-purpose hardwares. Our work should be classified as the former group due to the level of programmability of the CSX architecture. Conventional CPUs, GPGPUs, and Cell processor are also rather general purpose architectures which have been used to implement stereo vision algorithms. According to our knowledge, by using local approaches for stereo vision, the following are the fastest results reported on each of these architectures: CPUs [45], GPGPUs [48, 49], and cell processor [47]. As can be seen from Table 2.8, our results show a much better performance over these implementations in terms of both MDEs (except [49] which outperforms our approach in terms of MDEs) and particularly MDEs per Watt. Indeed, in terms of MDEs per Watt, our approach is one order of magnitude better than Cell and two orders of magnitude better than CPU and GPGPU implementations. On the other hand, some special purpose implementations outperform our approach in terms of MDEs and MDEs per Watt. For example, the ASIC [53] and the FPGA [55] implementations are one order of magnitude better than our implementation in terms of MDEs per Watt. In fact, ASIC and FPGA implementations could be the most efficient in terms of performance and

Table 2.8: A comparison with other implementations of local stereo vision algorithms in the literature

Author	Platform	Window	Power (watt)	Error*	MDEs	MDEs/Watt
van der Mark [45]	CPU **	9×9	82	n/a	143	1.74
Yang [48]	GPGPU	4×4	60	7.07	289	4.82
Zhu [49]	GPCPU	3×3	289	n/a	2,203	7.62
McCullagh [47]	Cell	4×4	92	n/a	961	10.45
This work	CSX700	3×3	9	23.6	1,032	114.6
Chang [50]	DSP	4×5	n/a***	21.7	88	-
Ambrosch [55]	FPGA	3×3	2.5	n/a	3,264	1,305.6
Woodfill [53]	ASIC	5×5	<1	n/a	2,555	2,555

* The reported error rates are calculated by Middlebury webpage for Tsukuba image [59]
** SSE Instruction are exploited for the implementation
*** This work is a software implementation

power consumption. FPGAs are more flexible than ASICs. However, they do not provide flexibility and programmability of a rather general purpose architecture. Table 2.8 illustrates that the performance of our approach fills the gap between general purpose architectures and special-purpose hardwares in terms of MDEs per Watt, and achieves a rather high performance in terms of MDEs. Indeed, our results clearly demonstrate that CSX architecture can provide excellent performance along with low power consumption and programmability of a rather more general purpose architecture for various embedded applications.

2.4 Parallel implementation of Harris corner detector

Feature detection is a low-level image processing task which is usually performed as the first step in many computer vision applications such as object tracking [60] and object recognition [61]. Harris Corner Detector (HCD) [62] is a popular feature detector due to its invariance to rotation, scale, illumination variation and image noises.

Fast implementation of HCD has been considered on various architectures. Teixeira et al. [63] have implemented HCD on a GPU. For an image of 640×480 resolution, the HCD is computed in 10.1 ms. They realized that the large number of memory accesses degrade performance. Therefore, by compressing each 2×2 pixels in the original image as one pixel, they reduced the computation time to 3.3 ms with one pixel imprecision. Saidani et al. [64] have employed Harris corner detector on Cell processor. Furthermore, Dietrich [65] has implemented HCD on a FPGA as part of a stereo vision system. The developed FPGA is capable of calculating HCD for images of the resolution 358×288 at the speed of 60 fps. Also, Cheng et al. [66] have proposed an ASIC implementation of HCD as part of a vision processor. The proposed architecture is capable of computing HCD for images of the resolution 128×128 at the speed of 1367 fps.

Rest of this chapter is dedicated to parallel implementation of HCD on the CSX architecture. First, HCD algorithm is briefly described in section 2.4.1. Then, data decomposition scheme and implementation details are discussed in section 2.4.2 and 2.4.3, respectively. Finally, experimental results are presented and discussed in section 2.4.4.

2.4.1 The Harris corner detector algorithm

To detect corners in a given image, the HCD algorithm [62] proceeds as following. Let $I(x, y)$ denote the intensity of a pixel located at row x and column y of the image.

1. For each pixel (x, y) in the input image compute the elements of the Harris matrix $G = \begin{bmatrix} g_{xx} & g_{xy} \\ g_{xy} & g_{yy} \end{bmatrix}$ as follows:

$$
\begin{aligned}
g_{xx} &= \left(\frac{\partial I}{\partial x}\right)^2 \otimes w \\
g_{xy} &= \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \otimes w \\
g_{yy} &= \left(\frac{\partial I}{\partial y}\right)^2 \otimes w,
\end{aligned} \tag{2.5}
$$

where $\otimes$ denotes convolution operator and w is the Gaussian filter.

2. For all pixel (x, y), compute Harris' criterion:

$$c(x, y) = \det(G) - k(trace(G))^2 \tag{2.6}$$

where

$$\begin{aligned} rcl\, \det(G) &= g_{xx}.g_{yy} - g_{xy}^2, \\ trace(G) &= g_{xx} + g_{yy} \end{aligned} \tag{2.7}$$

and k is a constant which should be determined empirically, and $trace(G) = g_{xx} + g_{yy}$.

3. Choose a threshold τ empirically, and set all $c(x, y)$ which are below τ to zero.

4. Non-maximum suppression, i.e. extract points (x, y), which have the maximum $c(x, y)$ in a window neighborhood. These points represents the corners.

2.4.2 Appropriate data decomposition scheme

In section 2.3.3, we described the three commonly used techniques for distributing arrays or matrices among processors: block, row-strip and row-cyclic decomposition schemes. Here, we analyze these data decomposition schemes for parallel implementation of HCD on CSX architecture. We recall the three important parameters that should be considered: (a) required memory space for each PE, (b) size of data that needs to be transfered to PE's memory more than once, and (c) inter-PE communication time (for detailed discussion see section 2.3.3).

According to the algorithm description in Section 2.4.1, HCD performs a set of operations in windows around each pixels. In fact, HCD uses windows which may have different sizes in 3 stages: calculating partial derivatives, Gaussian smoothing, and non-maximal suppression. Let ω be the sum of these window sizes. As before, p indicate the number of PEs. Also, c and r denote the number of columns and rows in image matrix, respectively. Finally, in each memory

Table 2.9: Figure of merit for different data decomposition schemes for HCD

Data decomp.	Boundary data	Redundant external memory comm.	Inter-PE comm.	PE memory space
Block	S	$cs(\omega-1)$	$r(\omega-1)$	Q
	M	-	$(cs+r)(\omega-1)$	$Q+M_m(\omega-1)/2$
Row-strip	S	$cp(\omega-1)$	-	Q
	M	-	$c(\omega-1)$	$Q+M_m(\omega-1)/2$
Row-cyclic		-	$c\omega(\omega-1)/2$	$Q-M_m(\omega-1)$

In this table, $Q=\omega M_m+m$, S indicates that boundary data is shared between PEs by using the swazzle path, and M indicates that boundary data is transferred from external memory
In calculating number of inter-PE communications for block and row-stripe decompositions, it is assumed that ω is smaller than number of rows which are assigned to each PE.

communication, each PE reads or writes m bytes of data (pixel) from/into the external memory. M_m is the memory space needed to calculate the elements of Harris matrix for m pixels.

The parameters calculated for each data decomposition scheme are summarized in Table 2.9. In block and row-strip decomposition schemes, the required poly memory space increases linearly with ω. Note that, the size of windows in HCD are determined empirically for each application. For larger ω, e.g. 7 or 11, using these data decomposition, the required PE memory will be larger than poly memory space. It is possible to decrease the required memory space in these decomposition schemes, but then redundant memory communications will increase. Row-cyclic decomposition needs less poly memory space and no redundant external memory communication. Although row-cyclic decomposition require inter-PE communication via swazzle path more than row-strip decomposition by a factor of $\omega/2$, this overhead will be negligible since communication via swazzle path is very fast (see section 2.2).

2.4.3 Proposed parallel implementation

Parallel implementation of HCD on the CSX architecture based on row-cyclic data decomposition is similar to implementation of SSD algorithm presented in

section 2.3.4. To efficiently exploit both cores of the CSX700, the input images are divided into two nearly equal parts. Besides, on each core, to apply the row-cyclic decomposition scheme, the input images are divided into groups of 96 rows and the computation is performed in several iterations. In each iteration, operations are performed on a group of 96 rows. To handle boundary conditions, two consecutive iterations are overlapped.

In addition, to overcome the overhead of external memory communication, communication and computation overlapping is greatly exploited in our implementation. To achieve maximum overlapping, each row is divided into a set of segments of size m. The computation for each row is then performed in several iterations over these data segments.

Computation Steps In this section, we present processing of one segment of data, i.e., m pixels. In our implementation of HCD, we have divided the algorithm into 5 steps: calculating partial derivation of I in direction x and y, Gaussian smoothing, computing Harris criterion, non-maximum suppression, followed by non-maximum suppression. Algorithm 2 shows the pseudocode for this processing.

To calculate partial derivation of I, we have used Prewitt operator. Prewitt operator uses two 3x3 kernels, P_X and P_Y, which are convolved with the original image to calculate approximations of the derivatives in x and y directions, respectively. In our implementation, we take advantages of the fact that convolution kernels used by Prewitt operator are separable, i.e. these kernels can be expressed as the outer product of two vectors.

$$
\begin{aligned}
P_X &= \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} * \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} \\
P_Y &= \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}
\end{aligned} \tag{2.8}
$$

So, the x and y derivation can be calculated by first convolving in one direction (using local data), then swazzling the computed data and convolving in the other direction.

Next step is Gaussian smoothing. Elements of Harris matrix, g_{xx}, g_{xy}, and g_{yy} are calculated using Eq. (2.5). As stated in section 2.2, the Gaussian smoothing can be performed using standard convolution methods. Gaussian kernel is also separable. Thus, the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then swzzling the calculated values and convolving with another 1-D Gaussian in the y direction. The y component is exactly the same as x component but is oriented vertically. Then, Harris' criterion is computed using Eq. (2.6).

In the next step, non-maximum suppression, the maximum value of Harris criterion in each 3x3 neighborhood is determined. First, each PE obtains the maximum value in 1x3 neighborhood. Then, each PE swazzle the maximum values to both its neighbors. Receiving the maximal values of two neighboring rows, the maximum value in 3x3 neighborhood can then be obtained. Using this strategy, the maximum value of 9 element in a 3x3 neighborhood is obtained by just 4 comparisons.

2.4.4 Results and performance of parallel implementation

To evaluate the performance, we have implemented the following HCDs on the CSX700 architecture: $HCD_{3\times3}$ and $HCD_{5\times5}$ which uses a 3×3 and 5×5 Gaussian kernel, respectively. Since our proposed parallel approach provides flexibility, it can be easily applied to images with different sizes, and various sizes of Gaussian filter or non-maximum suppression window. The performance of implemented algorithms in terms of latency, fps, and sustained GFLOPS for different image resolutions are summarized in Table 2.10. As Table 2.10 shows, for all tested image resolutions, even for resolution of 1280×720, our implementation is much faster than real-time.

The arithmetic intensity, i.e., number of operation per pixel, of $HCD_{3\times3}$ and $HCD_{5\times5}$ is 40 and 64, respectively. As Table 2.10 shows, the sustained GFLOP depends also on the image size. The reason is that in processing the last sweep

Algorithm 2 Pseudocode of Parallelized HCD

ω_1: Gaussian window size ω_2: NMS window size

PEs in parallel do

1. Derivation of I_x and I_y:

$I_x = I \otimes [-1\ 0\ 1]$, $\quad I_x = [swazzle\_down(I_x), I_x, swazzle\_up(I_x)] \otimes [1\ 1\ 1]$

$I_y = swazzle\_up(I) - swazzle\_down(I)$, $\quad I_y = I_y \otimes [1\ 1\ 1]$

2. Guassian Smoothing:

$g_{xx} = I_x^2 \otimes \mathrm{x} - \mathrm{Guassian}$, $\quad g_{xy} = (I_x I_y 2) \otimes \mathrm{x} - \mathrm{Guassian}$, $\quad g_{yy} = I_y^2 \otimes \mathrm{x} - \mathrm{Guassian}$

$g_{xx} = [swazzle\_down(g_{xx}), g_{xx}, swazzle\_up(g_{xx})] \otimes \mathrm{y} - \mathrm{Guassian}$

$g_{xy} = [swazzle\_down(g_{xy}), g_{xy}, swazzle\_up(g_{xy})] \otimes \mathrm{y} - \mathrm{Guassian}$

$g_{yy} = [swazzle\_down(g_{yy}), g_{yy}, swazzle\_up(g_{yy})] \otimes \mathrm{y} - \mathrm{Guassian}$

3. Computation Harris Criterion: $\quad c = g_{xx}g_{yy} - g_{xy}^2 - k(g_{xx} + g_{yy})$

4. Non-maximum suppression:

for $k = 1$ to m

$mx[k] = \max\{c[l] \mid k - (\omega_2 - 1)/2 \leq l \leq k + (\omega_2 - 1)/2\}$

for $k = 1$ to m

$mx[k] = \max\{mx[k],\ swazzle\_up(mx[k]),\ swazzle\_down(mx[k])\}$

5. Thresholding:

for $k = 1$ to m

if $c[k] \geq \tau$ and $c[k] == mxp[k]$

corresponding pixel is corner

$*$ *swazzle_up*() and *swazzle_down*() represent communication with left and right neighbors, respectively.

Table 2.10: Performance of HCD on CSX700 architecture using 3×3 and 5×5 Gaussian filter

Image	Latency (ms)		fps		Sustained GFLOPS	
Resolution	$HCD_{3\times3}$	$HCD_{5\times5}$	$HCD_{3\times3}$	$HCD_{5\times5}$	$HCD_{3\times3}$	$HCD_{5\times5}$
128×128	.165	.224	6060	4464	3.97	4.68
352×288	.8	1.22	1250	819	5.06	5.31
512×512	1.74	2.63	574	380	6.02	6.37
640×480	2.15	3.28	465	304	5.71	5.99
1280×720	7.04	10.89	142	91	5.23	5.41

Table 2.11: Comparison with other implementations in the literature

Image Resolution	fps reported in [ref]	fps achieved by our approach
128×128	1367 [66]	4464-6060
352×288	60 [65]	819
640×480	99 [63]	304

of data, some PEs may be idle, and the number of idle PEs depends on image size. For example, performing $HCD_{3\times3}$ for images of resolution 640×480 and 1280×720, the number of idle PEs are 4 and 32, respectively. Due to more utilization of PEs, better GFLOPS is achieved for images of 640×480 resolution.

Table 2.11 compares our implementation results with those reported in the literature. As can be seen, our approach provides much better performance in terms of latency or frame per second while providing a high degree of flexibility in terms of problem size and parameters.

2.5 Summary

This chapter presents fast parallel implementation of several dense stereo vision algorithms and HCD algorithm on the CSX SIMD architecture. Several data decomposition schemes were analyzed for an efficient parallel implementation with minimum communication overhead on the CSX architecture. In particular, it was shown that the row-cyclic is the most efficient data decomposition scheme for data parallel computation of low-level image processing algorithms such as the

SSD-based stereo vision and its variants and HCD. Moreover, it was shown that, by devising a careful strategy, it is possible to significantly reduce the memory communication overhead by almost fully overlapping computation and communication. In addition to an efficient parallelization, exploitation of the vector processing capability of the PEs was a key for achieving a better performance.

For stereo vision computation, SSD and its more robust, and hence more computationally expensive, variants have been considered. For most cases, even faster than real-time performance was achieved. The exceptions being multiple window and left-right check algorithms for HDTV images with disparity of 32. Our HCD results also represent a much faster than real-time implementation for all considered experiments.

The experimental results, presented in this chapter, clearly indicate that the CSX architecture is indeed a good candidate for achieving low-power high performance capability for low-level image processing tasks.

Next chapter explores parallel implementation of a more sophisticated and computationally expensive image processing algorithms on the CSX architecture.

Chapter 3

Implementation of Human Detection on CSX SIMD architecture

3.1 Introduction

This chapter studies parallel implementation of vision based human detection by using histogram of oriented gradients (HOG) descriptors [67] on the CSX SIMD architecture. Computation of HOG descriptors and HOG-based human detection are representative examples of intermediate and high-level image processing tasks, respectively. The ideas presented here especially for multi-scale computation and dealing with complex data dependency pattern can be easily exploited for parallel implementation of other object detection/classification algorithms. One example is computation of SIFT descriptors [68] which share similarities with HOG descriptor computation.

Furthermore, HOG descriptor [67] have become very popular and is being widely used in computer vision for object detection and in particular for human detection due to its excellent performance. On the other hand, detecting humans is required for a mobile robot or a humanoid: first for safety reasons and then to mimic human-like and social behavior. However, a major challenge in real-time applications of HOG descriptor is its computational complexity. In fact,

in the first original implementation of the HOG-based human detection on a conventional computer [67] only a performance of 1 fps could be achieved for a image of 320 × 240 resolution and with a rather small number of detection window of 800. Consequently, there have been several research works to improve the computational efficiency of the HOG descriptor.

Attempts to improve the computational efficiency of HOG-based object detection have followed two approaches: modifying and simplifying the original algorithm, or exploiting parallelism in its computation. In the former approach, most efforts are based on early rejection of scales or positions [69, 70]. For example [69] reports a performance of 5 fps for an image of 320 × 240 resolution and with 12800 detection windows.

The second and perhaps more promising approach is to speed up the HOG computation by exploiting parallelism. Cao et al. [71] have studied implementation of HOG-based human detector on a FPGA. However, they considered some modifications and simplifications to the original algorithm since computation of HOG descriptor is not suitable for FPGA implementation due to its rather complex communication and data dependency patterns. CUDA-based implementation of HOG descriptor has also been considered on GPGPU [72], [73]. Wojek et al. [72] have used a GeForce 8800 Ultra as their experimental platform and achieved a performance of 10 fps for colored images of 640 × 480 resolution. GeForce 8800 Ultra has the peak performance of 384 GFLOPS and power consumption of 157 watt. Prisacariu et al. [73] have used GTX 295 and achieved a performance of 12.5 fps for colored images of 640 × 480 resolution and 15 fps for grayscale images of 640 × 480 resolution. GTX 295 has the peak performance of 1788.48 GFLOPS and consumes 289 watt.

In section 3.3, our parallel implementation of HOG-based human detection on the CSX SIMD architecture is extensively discussed. Before that, in section 3.2, computation of HOG descriptors and HOG-based human detection are briefly described. In section 3.4, the achieved results are presented, and a summary of this chapter is given in section 3.5.

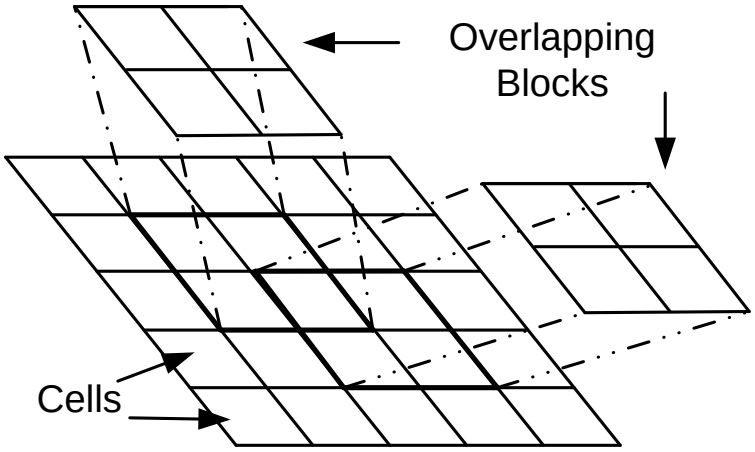


Figure 3.1: Division of the image in cells and blocks for HOG descriptor computation

3.2 HoG descriptor and object detection

In this section, we briefly discuss the computation of HOG descriptor [67], [74] and its application for object detection.

HOG Descriptor. To calculate HOG descriptor, the image is first divided into small spatial regions called *cell*s and the histogram of gradient orientation is calculated for each cell. In [67] both radial and rectangular cells have been discussed. Here, we consider each cell as a 2D array of $m \times m$ pixels. Then, histograms are normalized over groups of cells, called *block*s. Every block is a 2D array of $n \times n$ cells. There is overlapping among the blocks since each cell participates in several blocks. (Fig. 3.1)

In the following, each cell is denoted by an ordered pair $c(i, j)$ where $1 \leq i \leq r/m$ and $1 \leq j \leq c/m$ (r and c denote the number of rows and columns of the input image , respectively). Each block is also denoted by an ordered pair $b(i, j)$. The block $b(i, j)$ consists of cells $c(i, j), c(i, j + 1), \ldots, c(i + n, i + n)$.

Calculation of HOG descriptor consists of three steps: first, the magnitude and the orientation of gradient is calculated for each pixel in the image. To see this, let $I(x, y)$ be the intensity of pixel located at row x and column y. Then, the magnitude of gradient, $m(x, y)$, and the orientation of gradient, $\theta(x, y)$ are

calculated by:

$$m(x,y) = \sqrt{I_x(x,y)^2 + I_y(x,y)^2} \tag{3.1}$$

$$\theta(x,y) = \arctan \frac{I_y(x,y)}{I_x(x,y)}, \tag{3.2}$$

where $I_x(.,.)$ and $I_y(.,.)$ denote the partial derivation of I in direction x and y, respectively.

Second, the histograms are calculated for each cell. To calculate histograms, the orientation bin are evenly spaced over $0° - 180°$ or $0° - 360°$ into q bins. Each pixel calculates a vote for the histogram bin based on $\theta(x,y)$, the orientation of gradient on that pixel. The vote is also a function of $m(x,y)$, gradient magnitude at that pixel. Also to improve performance, the contribution of each pixel in the cell histogram is wighted by a Gaussian function centered in the middle of the block. Moreover, weighted votes are interpolated trilinearly between the neighboring bin centers in both orientation and position, to reduce aliasing effect.

The final step is normalization to form the block's descriptor which is calculated as follows. Let $\mathbf{C}_{i,j} = (c_1, c_2, \ldots, c_q)$ denote the q dimension histogram of cell $c(i,j)$. Then, the unnormalized descriptor vector for the block $b(i,j)$ is defined as $\mathbf{b}_{i,j} = (\mathbf{C}_{i,j}, \mathbf{C}_{i,j+1}, \ldots, \mathbf{C}_{i+n,j+n})$. Various normalization schemes could be considered. By using *L2-norm*, the HOG descriptor of the block $b(i,j)$, $\mathbf{B}_{i,j}$, is then given by:

$$\mathbf{B}_{i,j} = \frac{\mathbf{b}_{i,j}}{\sqrt{\|\mathbf{b}_{i,j}\|^2}}. \tag{3.3}$$

Object Detection. To detect objects, a sliding window based algorithm is used. First, a detection window is defined as a grid of $k \times l$ blocks. The detection windows are overlapping since a given block can belong to several detection windows. The descriptor of the detection window is then obtained by combining the vectors of HOG descriptors of its blocks. To this end, let $d(i,j)$ denote the detection window consisting of the blocks $b(i,j), b(i,j+1), \ldots, b(i+k, j+l)$. Then, the descriptor of this detection window is given as the vector $\mathbf{V}_{ij} = (\mathbf{B}_{i,j}, \mathbf{B}_{i,j+1}, \ldots, \mathbf{B}_{i+k,j+l})$. For object detection, the detection window is scanned

across the image at all positions and scales and the resulting descriptors are fed into a pre-trained linear SVM classifier to score each descriptor. It should be mentioned that the best results by using HOG descriptors have been achieved in human detection since HOG descriptors are not orientation invariant.

Human Detection. As reported in [67], the best results for human detection have been achieved by using the following set of parameters. Each cells is defined as a rectangular of 8x8 pixels and each block as a rectangular of 2x2 cells. Orientation of gradients are spaced over $0° - 180°$ into 9 bins. The detection window size is taken to be of 7x15 blocks or 64x128 pixels. The step size of sliding window is also taken to be of one cell (8 pixels) both horizontally and vertically. We have used this set of parameters in our parallel implementation.

3.3 Parallel formulation and implementation of HOG-based Human detection

In this section, we discuss parallel implementation of HOG-based object detection. The human detection problem is considered here, but that does not change the parallel formulation of the general algorithm. As mentioned in section 3.2, the computation is repeated for various scales, which represents repeated computation of the same problem but with reducing size. For example, consider a VGA image of 640x480 resolution as the first scale of computation. Given the parameters discussed in section 3.2, for this first scale the number of cells, blocks, and detection windows are 4800, 4664, and 3285, respectively. By using a scale ratio of 1.05 (as also used in [67]), the total number of images obtained by this scaling is 28 (including the original image). The smallest image is of 168x128 resolution with a number of 336 cells, 300 blocks, and 14 detection windows. Finding a mapping of computation on a SIMD architecture, which is efficient for various problem sizes, is challenging. Here, we first describe our technique to parallelize computation of one scale. Then, we discuss how to extend our solution for multi-scale computation.

3.3.1 One scale computation

We consider parallel computation of each scale on a single core (with 96 PEs) of the CSX architecture. As will be discussed in the following, both cores of the CSX architecture are then employed to exploit parallelism in the overall computation of all scales. The computation for object detection by using HOG descriptor, as described in section 3.2, can be analyzed at four levels of operations as follows:

- **Pixel Level Operations:** This level includes all preprocessing on the input image to generate images of gradient magnitude, $m(x, y)$, and gradient orientation, $\theta(x, y)$, according to Eq. (3.1) and Eq. (3.2), respectively, to be used as the input for next level.

- **Cell Level Operations:** This level includes computation of histogram for each cell $c(i, j)$. However, as discussed in section 3.2 each cell participates in several blocks. In each block, the contribution of each pixel in the cell histogram is weighted by a Gaussian function centered in the middle of block. Also, in each block, pixels of neighboring cells contribute to the histogram of the cell.

- **Block Level Operations:** This level includes calculation of HOG descriptor, $\mathbf{B}_{ij}$, for each block $b(i, j)$, as described in section 3.2. In our application for human detection, the vector of block descriptor has 36 elements.

- **Detection Level Operations:** This level includes linear SVM evaluation. For human detection, each detection window $d(i, j)$ consists of 7x15 overlapping blocks, $b(i, j), b(i, j+1), \ldots, b(i+14, j+6)$. Let $\mathbf{V}_{ij}$ denote the descriptor of detection window $d(i, j)$. Then, $\mathbf{V}_{ij} = (\mathbf{B}_{i,j}, \mathbf{B}_{i,j+1}, \ldots, \mathbf{B}_{i+14,j+6})$. To evaluate $d(i, j)$ for human presence, d_{ij} is computed as:

$$d_{ij} = \mathcal{W}^t \mathbf{V}_{ij} + a, \tag{3.4}$$

  where $\mathcal{W}$ is a vector with 3780 elements and a is a constant. Both $\mathcal{W}$ and a are determined in the offline learning phase for human detection [74][1].

[1] Here, the parallelization of the learning phase is not considered.

As can be seen, the computation involves several level of operations with different granularity. Therefore, the first critical decision to achieve optimal performance is the choice of grain size. For our parallel implementation on the CSX SIMD architecture, we have chosen the block level granularity. A pixel or cell level decomposition (finer grain parallelism) would result in a significant inter-PEs communications. With a detection level decomposition (coarser grain), the performance would suffer from lots of redundant computation of HOG descriptors, since overlapping detection windows include same blocks.

With a block level granularity, the input image is subdivided into blocks and PEs are assigned to compute the HOG descriptor of the blocks. Since, even for the smallest image scale the number of blocks (> 105) [1] is greater than the number of PEs (96), each PE has to calculate the HOG descriptor of several blocks. Hence, the next decision is to determine the assignment of the blocks to PEs. Since HOG descriptors of blocks are used as the input for the detection level, this assignment also affects the computation at the detection level. In the following, detection level computation and the optimal scheme for assignment of blocks to PEs to enable an efficient implementation of detection level computation are discussed.

At the detection level, for each detection window $d(i, j)$, d_{ij} is computed using Eq. (3.4). On one hand, Both vectors $\mathcal{W}$ and $\mathbf{V}_{ij}$ have 3780 elements. On the other hand, PE's memory space are rather limited. In fact, 1536 elements can be stored in whole PE's memory space. Therefore, some parts of data should be stored in external memory and this part of computation is memory bound.

To compute Eq. (3.4), $\mathcal{W}$ can be written as $\mathcal{W} = \{\mathbf{W}_{1,1}, \mathbf{W}_{1,2}, \ldots, \mathbf{W}_{7,15}\}$, where each $\mathbf{W}_{ij}$ is a vector of 36 elements corresponding to a block. Thus, Eq. (3.4) can now be computed as:

$$d_{ij} = a + \sum_{k=1}^{15} \sum_{l=1}^{7} \mathbf{W}_{kl}^{t} \mathbf{B}_{i+k-1, j+l-1}, \tag{3.5}$$

[1] Resolution of the smallest image scale is at least equal to detection window resolution (64×128)

then

$$d_{ij} = a + \sum_{k=1}^{15} \sum_{l=1}^{7} d_{ij}^{kl} \tag{3.6}$$

where

$$d_{ij}^{kl} = \mathbf{W}_{kl}^{t} \mathbf{B}_{i+k,j+l}. \tag{3.7}$$

Since detection windows are overlapped, each block contributes in several detection windows in 105 different positions of (k, l). Therefore, for each $b(m, n)$, we have to compute

$$d_{m-k+1,n-l+1}^{kl} = \mathbf{W}_{kl}^{t} \mathbf{B}_{m,n} \quad \forall k, l : 1 \leq k \leq 15, 1 \leq l \leq 7. \tag{3.8}$$

So, to reduce external memory communication, each PE computes the partial result of $d_{i,j}^{kl}$ using Eq. (3.8), just after computing HOG descriptor of one block. Since, HOG descriptors are already in PEs' memories, only vector $\mathcal{W}$ is transfered from external memory into PEs' memories. However, each block participates in 105 detection window, i.e. each block produces 105 partial results which also could not be stored in limited memory of PEs. So, to save memory space, each PE should store a summation of partial results. To perform summation, each PE requires partial results computed by other PEs.

Figure 3.2 illustrates our scheme for the assignment of blocks to PEs. As shown in this Figure, PE_j is assigned to calculate the HOG descriptors of all blocks $b(i, j)$. PE_j also calculate values of d_{ij} for all detection windows $d(i, j)$. This scheme permits efficient usage of swazzle path and limited PEs' memories.

This scheme allows Eq. (3.6) to be performed in two steps. First, PEs communicate their partial results, d_{ij}^{kl} to calculate partial summation d_{ij}^{k} as follows:

$$d_{ij}^{k} = \sum_{l=1}^{7} d_{ij}^{kl}, \tag{3.9}$$

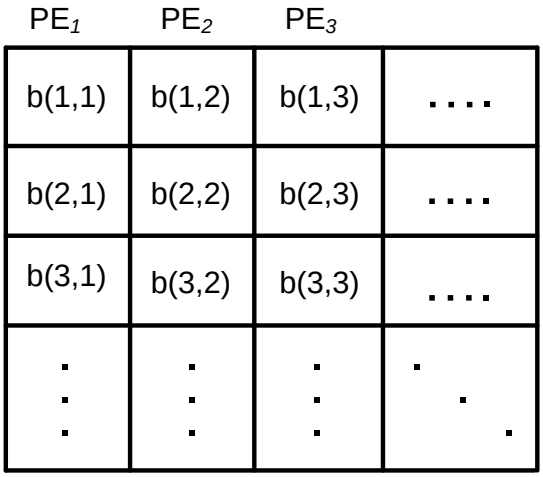


Figure 3.2: The Assignment of Blocks to PEs

then, each PE calculates final value of d_{ij} as:

$$d_{ij} = b + \sum_{k=1}^{15} d_{ij}^{k}. \tag{3.10}$$

For example, consider detection window $d(1,1)$. PE_1 is responsible to calculate $d_{1,1}$. First, PE_1 calculates $d_{1,1}^{1,1}$ and receives $d_{1,1}^{1,2}, \ldots, d_{1,1}^{1,7}$ from $\mathrm{PE}_2, \ldots, \mathrm{PE}_7$, respectively, and then, calculates $d_{1,1}^{1}$. Second, PE_1 calculates $d_{1,1}^{2,1}$ and receives $d_{1,1}^{2,2}, \ldots, d_{1,1}^{2,7}$ from $\mathrm{PE}_2, \ldots, \mathrm{PE}_7$, respectively, and then, calculates $d_{1,1}^{2}$ and add $d_{1,1}^{1}$ and $d_{1,1}^{2}$ together, and iteration continues.

Algorithm 3 presents the pseudocode for HOG descriptor computation and SVM evaluation for PE_j. Line 1 calculates the number of blocks. Line 2 allocates memory for array d which stores d_{ij} for each detection window $d(i,j)$. The first 14 elements are not valid data, and store results of some redundant computations on boundary blocks. This redundant calculation represents a small overhead, but it allows regular operations to be performed for all blocks by all PEs [1]. Lines 3-4 perform pixel level operations for pixels in cell $c(1,j)$. Since blocks $b(1,j)$ and $b(1,j+1)$ share the cell $c(1,j+1)$, PE_j receives the results of pixel level operations on cell $c(1,j+1)$ from its neighbor via swazzle path (line 5). Each

[1] This feature is used in multi-scale computation

iteration of **for** loop of Lines 6-19 computes HOG descriptor of one block and also the partial results for SVM evaluation. Since two consecutive blocks $b(i, j)$ and $b(i + 1, j)$ share two cells, in each iteration $m(x, y)$ and $\theta(x, y)$ for 2 new cells are required (Lines 7-9). Line 10 calculates HOG descriptor as discussed in section 3.2. Lines 11-19 handle SVM evaluation according to the Eq. (3.10).

Algorithm 3 HOG Descriptor & SVM Evaluation - PE_j

1. n = number of image row/8 − 1
   // the height of each cell (in pixels) is 8
2. d: Allocate an array of type float and size of n + (15 - 1)

3. **Receive** $I(x, y)$ for pixels in cell $c(1, j)$ from external memory
4. Calculate $m(x, y)$ and $\theta(x, y)$ for cell $c(1, j)$
5. **Receive** $m(x, y)$ and $\theta(x, y)$ for cell $c(1, j + 1)$ from PE_{j+1}

6. **for** i = 1 to n
7. **Receive** $I(x, y)$ for pixels in cell $c(i + 1, j)$ from external memory
8. Calculate $m(x, y)$ and $\theta(x, y)$ for cell $c(i + 1, j)$
9. **Receive** $m(x, y)$ and $\theta(x, y)$ for cell $c(i + 1, j + 1)$ from PE_{j+1}

10. Calculate HOG descriptor, B_{ij}

11. **for** k = 1 to 15
12. $d[(i + 15) - k] = \mathbf{W}_{1,1}\mathbf{B}_{i,j}$
13. **for** l = 2 to 7
14. $d = \mathbf{W}_{k,l}\mathbf{B}_{i,j}$

15. **send** d to PE_{j-l} // via swazzle path
16. d = **receive** data from PE_{j+l} // via swazzle path

17. $d[(i + 15) - k] = d[(i + 15) - k] + d$
18. **end**
19. **end**

20. **Send** $d[14 : i + 14]$ to the external memory as the result

As Fig. 3.2 illustrates, the number of required PEs to perform Algorithm 3 on a given image is determined by number of image columns. If the number of required PEs is greater than 96 available PEs, then the given image should be divided into smaller overlapping images which share some boundary columns. Then, each of these smaller images could be processed independently. Also, if the number of required PEs is less than 96, some PEs will be idle. Moreover, number

of iteration in Algorithm 3 (or number of blocks which are assigned to each PE) is determined by number of image rows. Since the computation of all blocks are exactly the same, all iterations take the same computation time. Hence, the computation time for the image is a linear function of iteration numbers.

3.3.2 Multi-scale computation

To calculate HOG descriptor for various image scales, first the input image is scaled down using a bilinear interpolation scheme (see below). Then, the computation of the HOG descriptor and SVM evaluation are performed for each downscaled image. Here, we first describe our parallelization approach to implement image downscaling on the CSX architecture. Then, we show that the efficient implementation of multi-scale computation on a fixed number of PEs can be mapped as the solution of the *strip packing problem*.

3.3.2.1 Image Downscaling

To scale down the input image, we use bilinear interpolation as follows. To estimate the value of an unknown pixel, bilinear interpolation considers the closest 4 known pixels surrounding the unknown pixel. Then, the value of unknown pixel is set to the weighted average of these 4 pixels. To see this, let r denote the scale ratio. To estimate the value of pixel at row x and column y in downscaled image, the four pixels (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2) in the input image are considered, where

$$x_1 = \lfloor x * r \rfloor, \qquad x_2 = x_1 + 1 \tag{3.11}$$

$$y_1 = \lfloor y * r \rfloor, \qquad y_2 = y_1 + 1. \tag{3.12}$$

Thus, to compute each row of output image, two rows of input image are required. Our parallel implementation is based on output data decomposition, i.e. output image is partitioned into rows, and each row is computed by one PE. The first row of output is assigned to the first PE, the second row is assigned to the second PE, and so on [1]. Since each CSX700 core has 96 PEs and the output

[1] This is the row-cyclic data decomposition of output data. In chapter 2, it was shown

images can have more than 96 rows, PE_i is assigned to compute all the rows j where $i = j \mod 96$. This parallelization approach minimizes both the external memory and the inter-PE communications.

Data decomposition scheme for image downscaling and HOG descriptor computation is different. To calculate each scale of the input image, first PEs receive the input image and the scale ratio, and then calculate the resulting downscaled image. The downscaled image is then transfered to external memory. Afterward, Algorithm 3 could be performed on the downscaled image.

3.3.2.2 Multi-Scale Optimization Problem

In section 3.3.1, we discussed the computation of HOG descriptor and SVM evaluation on one scale of image. The same algorithm can be applied for all scales in several rounds where in each round Algorithm 3 is applied to one scale of image. However, such a straightforward approach would be inefficient in term of PEs usage and hence the total computation time since the number of idle PEs increases as the scale of the image decreases. To clarify this issue, consider the example of 640×480 input image and scale ratio of 1.05 for which the computation is performed on 28 scales. The number of required PEs and iterations for each scale are shown in Table 3.1. As can be seen, to perform computation for scales 12 to 28, less than 50% of 96 available PEs are used. So, if in each round the computation is performed only for one scale, a large number of PEs would be idle. For example, the computation for scales 27 to 28 uses just 22 and 21 PEs, respectively.

A consequence of our parallelization strategy is that in Algorithm 3 regular operations are performed for all blocks by all PEs. Therefore, it is possible to perform the computation on several scales of input image simultaneously in each round to reduce the number of idle PEs and hence the total computation time. The only constraint is that the total number of required PEs to compute these scales should be less than or equal to 96. As an example, the total number of required PEs to perform computation on scales 25 to 28 is 90 PEs. To perform computation on these scales in one round, the first 24 PEs can operate on scale

that row-cyclic data decomposition is the most efficient scheme for parallel implementation of low-level image processing tasks on the CSX architecture

Table 3.1: The resources required for all scales of image of resolution 640×480 and scale ratio equal to 1.05. The first number denotes the required number of PEs and the second one shows number of iteration

Scale	1	2	3	4	5	6	7
Required PEs	80	76	72	69	65	62	59
Iterations	59	56	53	50	48	46	43
Scale	8	9	10	11	12	13	14
Required PEs	56	54	51	49	46	44	42
Iterations	41	39	37	35	34	32	30
Scale	15	16	17	18	19	20	21
Required PEs	40	38	36	34	33	31	30
Iterations	29	27	26	25	23	22	21
Scale	22	23	24	25	26	27	28
Required PEs	28	27	26	24	23	22	21
Iterations	20	19	18	17	16	15	15

25, the next 23 PEs (PE_{25} to PE_{47}) on scale 26, the next 22 PEs (PE_{48} to PE_{69}) on scale 27, and the next 21 PEs (PE_{70} to PE_{90}) on scale 28. If these computations are performed simultaneously in one round and in an overlapped fashion, then the number of required iterations will be 17, which is the number of iterations required by the biggest scale, i.e., scale 24. While, if the computations are performed in different rounds, then the number of required iterations will be 63 (sum of required iterations for the four scales). The key issue is then to find an optimal scheme for mapping the computations of various scales on the fixed number of PEs to minimize the total computation time. This optimization problem is indeed equivalent to the solution of the *strip packing problem*.

The *strip packing problem* consists of packing a set of rectangular items of width at most W on a strip of fixed width W and of infinite height (see, for example, [75]). The items may neither overlap nor be rotated. The objective is to minimize the height used.

Let $S = \{1, 2, \ldots, n\}$ where each $s \in S$ denotes one scale of input image. Assume I_s and P_s denote the number of iterations and the number of PEs required for computation of scale s, respectively. Since each CSX700 core has 96 PEs, it

can be considered as a strip of width 96. Also, as shown in Fig. 3.3(a), the computation for scale s of input image can be considered as a rectangle item of width P_s and height I_s. The objective is to minimize the computation time which could be considered as the height used to place all the rectangles (i.e, the computation of all different scales of image) into a strip (one CSX700 core).

The *strip packing problem* is NP-Hard [75]. Various methods have been proposed to solve *strip packing problem* exactly or approximately. A class of commonly used heuristic algorithms for strip packing problems are level algorithms. In all level algorithms, items are pre-sorted into decreasing order of height. Then, items are packed from left to right on a series of horizontal levels. Within the same level, all items are placed so that their bottoms are aligned. The first level is the bottom of the strip, and the subsequent levels are determined by the height of tallest items packed on the previous levels. For our current implementation, we have used a best-fit decreasing height (BFDH) heuristic to solve the mapping problem. This algorithm packs the next item on the level with minimum residual horizontal space.

Figure 3.3(b) illustrates the packing scheme produced by the best-fit algorithm for the example specified in Table 3.1. Each level in the produced packing represents one round of computation, i.e. all scales of input image which are placed in one level are computed in one round. For example, as shown in Fig. 3.2, the first round includes computation of only the first scale while the last round includes computation of scales 28 and 27.

It should be emphasized that the computations of rounds are totally independent and hence they can be performed in any order. Note that, this also includes the computation for image downscaling. Therefore, since each CSX700 has two cores, each core can be assigned to perform computation of half of the rounds. Likewise, if more than one CSX700 is used, rounds are divided into groups, and each group is assigned to one core. Number of groups is equal to number of available cores (number of chip $*2$).

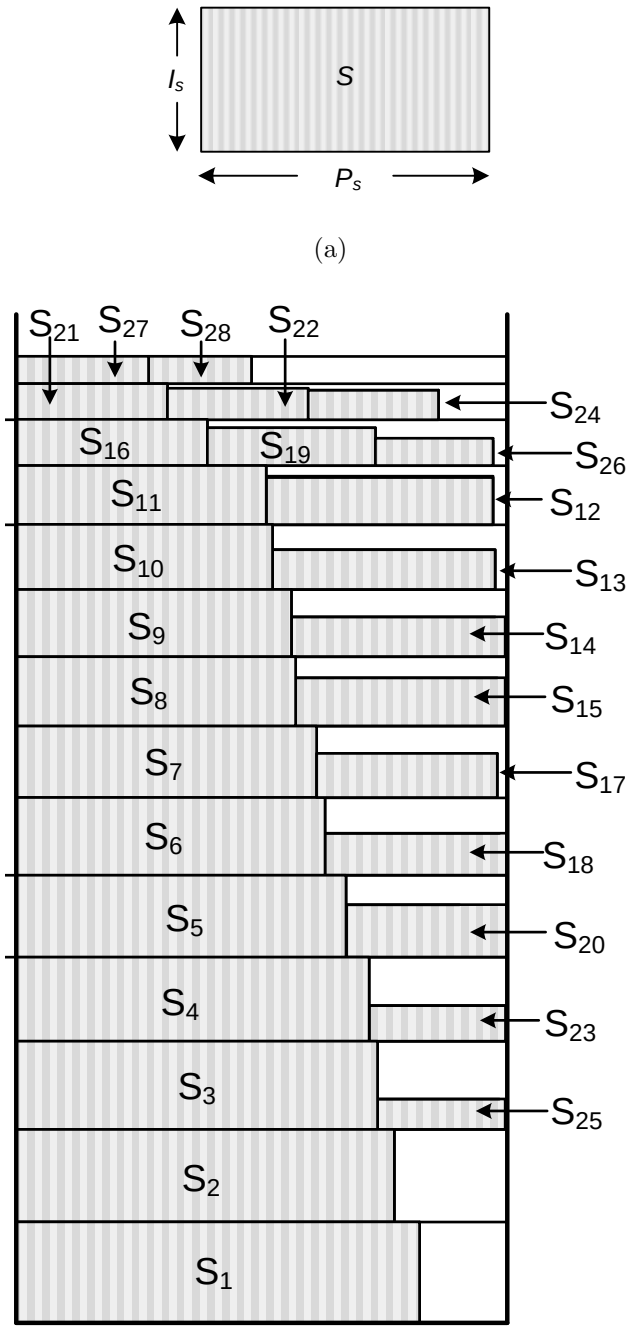


Figure 3.3: (a) Computation for each scale of image can be presented by a rectanglur of width P_s (number of required PEs) and height I_S (number of iterations or time) (b) Packing produced by the BFDH heuristic for the example specified in the Table 3.1

Table 3.2: Performance of HOG-based human detection algorithm on one core of CSX700 Architecture for images of 640 × 480 resolution and scale ratio of 1.05

Step	Computation time (ms)	Sustained GFLOPS
Image Downscaling	33.67	1.39
Gradient Computation	43.48	.7
Gradient Swazzling	2.51	-
HOG Descriptor Computation	100.68	1.72
SVM Evaluation	111.31	1.71
Total	292.03	1.51

3.4 Results and Performance of Parallel Implementation

We have implemented our proposed parallel algorithm for HOG-based human detection on the CSX700 SIMD architecture. As discussed in section 3.3.2, efficient mapping of the multi-scale computation for HOG-based human detection on the CSX700 architecture is equivalent to the solution of the *strip packing problem*. For the example of 640 × 480 input image and the scale ratio of 1.05 (given in Table 3.1) and by applying BFDH algorithm (shown in Fig. 3.3(b)), we have obtained a total computation time of 570 iterations. For this example, by using one core of the CSX700 processor, we have achieved a performance of 3.4 fps. The computation time of different steps of our parallel algorithm for grayscale images of 640 × 480 resolution and scale ratio equal to 1.05 is presented in Table 3.2.

Gradient computation (Lines 3-4 and 7-8 of Algorithm 3) takes 43.48 ms (14% of total computation time). As mentioned in Sec. 3.3.1, in our parallel implementation each PE performs gradient computation for half of the pixels of its assigned blocks and receives the result of gradient computation for the other half of the pixels of its blocks from its neighbor via swazzle path. As shown in Table 3.2, gradient swazzling (line 5 and 9 of Algorithm 3) takes just 2.51 ms. Therefore, this represents an efficient scheme for reducing the computation time since without swazzling the total time of this step would be of about 89 ms. The most

Table 3.3: Performance comparison of one core and two core implementation

	Latency (ms)	fps	Sustained GFLOPS
1 core	292.03	3.4	1.51
2 core	146.23	6.8	3

Table 3.4: Comparison with other implementations in the literature

	Computation time (ms)	Peak Performance (GFLOPS)	fps / watt
This Thesis	146.23	96	.75
Wojek [72]	99	384	.06
Prisacariu [73]	67	1788.48	.05

costly step is the SVM evaluation which has a computation time of 111.31 ms. Note that, although the operations in SVM evaluation are performed efficiently since they are fully vectorized, the PEs are idle for part of the computation of this step waiting to receive data (the vector $\mathcal{W}$) from the external memory. It should be emphasized that this idle time is somehow unavoidable since the vector $\mathcal{W}$ cannot be stored in PE's memory due to its limited size.

Next, we have implemented the above example by exploiting both cores of CSX700. As discussed in section 3.3.2, the computation of rounds are totally independent and can be performed in any order. So, the rounds are divided into two nearly equal groups and each group is assigned to one core of CSX700. Table 3.3 compares our implementation of one and two cores of CSX700. Our results illustrates that using both cores of CSX700, we achieved a near perfect speedup of two.

Table 3.4 compares our implementation with those reported in the literature. Although, in terms of computation time our implementation is slower but in terms of power consumption and particularly fps per watt, it represents a much better performance. To achieve better performance in terms of computation times, more CSX700 processors can be used. As discussed in section 3.3.2, the multi-scale computation can be divided among more cores, i.e. by using multiple CSX700, enabling almost a linear speedup. For example, by using 4 CSX boards a performance of about 25 fps can be achieved while consuming 36 watts.

3.5 Summary

This chapter presents parallel formulation and fast implementation of HOG-based human detection for mobile robot applications on the CSX SIMD architecture. A detailed analysis of multi-level computational structure of the HOG descriptors showed that the choice of block level grain size for parallel implementation is the most efficient in terms of reducing the redundancy in the computation and communication. Besides, the main challenge in parallel implementation of HOG-based human detection is the need for multi-scale computation. A consequence of our parallelization strategy is that regular operations are performed for all blocks by all PEs. Taking advantage of this regularity, it was shown that efficient multi-scale computation on a fixed number of PEs can be mapped as the solution of a 2D *strip packing problem*.

Our practical implementation results indicate that a performance of over 6 fps for an image resolution of 640×480 can be achieved while consuming only 9 watts. This performance can be further increased by deploying multiple CSX architectures. Such performance per watt enables novel capabilities in mobile robots and humanoid applications.

At the end of this chapter, it should be mentioned that for parallel computation of HOG-based object detection, the main challenges were on one hand, complex data dependency pattern, varying granularity (pixel, cell, block, and detection window), and multi-scale computation and on the other hand small size of PEs' memories in the CSX architecture. Other intermediate or high level image processing tasks especially object detection algorithms share similarities with HOG-based object detection in terms of computation needs. The solution proposed in this chapter can be exploited for parallel formulation and implementation of those tasks.

Chapter 4

Asynchronous Parallel Event-based Optical Flow

4.1 Introduction

This chapter first presents an event-based optical flow algorithm [76] and its serial implementation, and second, studies the parallel formulation and implementation of the algorithm on the Tilera MIMD architecture. Optical flow provides crucial information for various robotic applications such as navigation, obstacle avoidance, and tracking moving objects. Optical flow computation has been extensively studied over last decades. For a review on different optical flow computation methods and comprehensive study on performance of optical flow algorithms in terms of accuracy see [77, 78]. Actually, state-of-the-art methods can provide excellent and robust computation of flows for a variety of applications. However, they are very time consuming and power hungry, and consequently are not suitable for real-time and low-power applications such as mobile robots and humanoids. Fleury et al. [79] have proposed a general pipeline structure for parallel implementation of several optical flow techniques on a distributed-memory MIMD architecture, Transtech Paramid. While they have achieved speed up for all the cases, the reported computation times cannot satisfy the real-time application requirements.

As mentioned in chapter 1, event-based DVS vision sensor provides an sparse

representation of the scene. By only processing important data, the event-based vision algorithms can provide much faster solution compared with the conventional image processing algorithms. To reduce the computation cost in terms of both time and power, we have exploit DVS vision sensor for optical flow applications. At first step, the event-based optical flow algorithm proposed by Bensonman et al. [76] was implemented and tested on the iKart mobile robots and icub humanoid robots. Then, to further improve the run-time of the algorithm, we solved the problem in parallel and implement the parallel algorithm on the Tilera MIMD architecture.

In section 4.2, the event-based optical flow algorithm is reviewd. Then, a brief but comprehensive overview of the Tilera architecture is given in section 4.3. Parallel formulation of event-based optical flow and the implementation issues are discussed in section 4.4 and 4.5, respectively. Then, the results of both implementations on GPP and Tilera architecture are presented in section 4.6. Finally, a summary of this chapter is given in section 4.7.

4.2 Event-based optical flow algorithm

As mentioned above, we have exploited the event-based optical flow algorithm proposed by Benosman et al. [76]. Like the standard approach in computing optical flow [80, 81, 82], they have assumed the *brightness change constraint equation*, which is based on the hypothesis that the intensity structure of local time-varying regions of image are constant under motion for a short duration, so:

$$\frac{dI(x,y,t)}{dt} = 0, \tag{4.1}$$

where I represents the image gray level intensity. Expanding Eq. (4.1) leads to:

$$\begin{aligned} \frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial I}{\partial t} &= \begin{pmatrix} \frac{\partial I}{\partial x} & \frac{\partial I}{\partial y} \end{pmatrix}^T \begin{pmatrix} \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial t} \end{pmatrix} + \frac{\partial I}{\partial t} \\ &= \begin{pmatrix} \frac{\partial I}{\partial x} & \frac{\partial I}{\partial y} \end{pmatrix}^T \begin{pmatrix} v_x \\ v_y \end{pmatrix} + \frac{\partial I}{\partial t} = 0, \end{aligned} \tag{4.2}$$

where $(v_x, v_y)^T$ denotes the 2D velocity field. Equation (4.2) is a linear equation with two unknowns, and consequently it does not have a unique solution. One of the most popular techniques to overcome this problem is based on *local constant flow* assumption and was first proposed by Lucas and Kanade [81]. They assumed that $(v_x, v_y)^T$ is constant over an $n \times n$ neighborhood of the pixel (x, y). Then, the *optical flow equations* (i.e. Eq. (4.2)) for all pixel in a $n \times n$ neighborhood can form an equation system as follows:

$$\begin{pmatrix} grad^T(I(x_1, y_1)) \\ \vdots \\ grad^T(I(x_m, y_m)) \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} -I_{t_1} \\ \vdots \\ -I_{t_m} \end{pmatrix} \tag{4.3}$$

where $grad^T(I)$ and I_t denote spatial and temporal derivatives of I, respectively, and $m = n^2$. This equation system then can be solved using least square error minimization technique.

Benosman et al. [76] have suggested that the Eq. 4.3 can be formulated in an event-based manner. Since the event-based DVS sensor does not provide gray level intensity, the main difficulty would be the computation of the spatial derivatives. To provide an estimation of spatial derivatives, the activity of the neighboring pixels in a time interval of Δt are compared as follows:

$$\frac{\partial e(x, y, t)}{\partial x} \sim \sum_{t-\Delta t}^{t} e(x, y, t) - \sum_{t-\Delta t}^{t} e(x-1, y, t) \tag{4.4}$$

$$\frac{\partial e(x, y, t)}{\partial y} \sim \sum_{t-\Delta t}^{t} e(x, y, t) - \sum_{t-\Delta t}^{t} e(x, y-1, t). \tag{4.5}$$

Besides, the temporal derivative can be estimated as follows:

$$\begin{aligned} \frac{\partial e(x, y, t)}{\partial t} &\sim \frac{\sum_{t-\Delta t}^{t'} e(x, y, t) - \sum_{t-\Delta t}^{t} e(x, y, t)}{t - t'} \\ &= \frac{\sum_{t'}^{t} e(x, y, t)}{t - t'}, \qquad \text{with} \quad t - \Delta t < t' < t. \end{aligned} \tag{4.6}$$

The computation steps of event-based optical flow are illustrated in Fig. 4.1. First, for each event $e(x, y, t)$, all the events in a $n \times n \times t$ neighborhood are

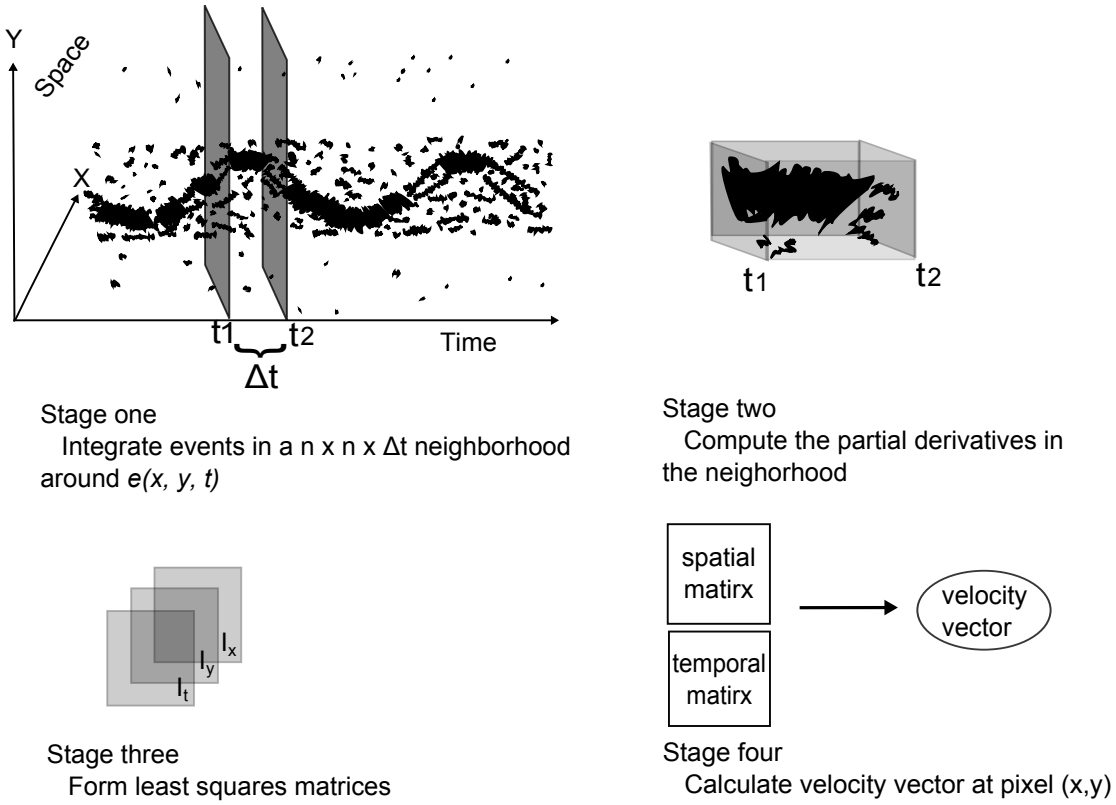


Figure 4.1: Event-based optical flow computation

indicated. Second, the partial derivatives are computed using Eq (4.4), (4.5), and (4.6). Third, the least squares matrices are built, and finally the velocity vectors are computed.

4.3 Tilera Architecture

In this section, we briefly review TILEPro64 [2] architecture with the emphasis on the key features, i.e. memory organization and on-chip interconnection network, which have been employed in our implementation[1].

As illustrated in Fig. 4.2, TILEPro64 is a many-core architecture including 64 processor cores (called *tiles*) organized in a two-dimensional mesh interconnected

[1]TILE-Gx^{TM} processor family, the successor of TILEPro^{TM} family are currently available in the Market. However, to test our parallel implementation, we only had access to the TILEPro64 processor

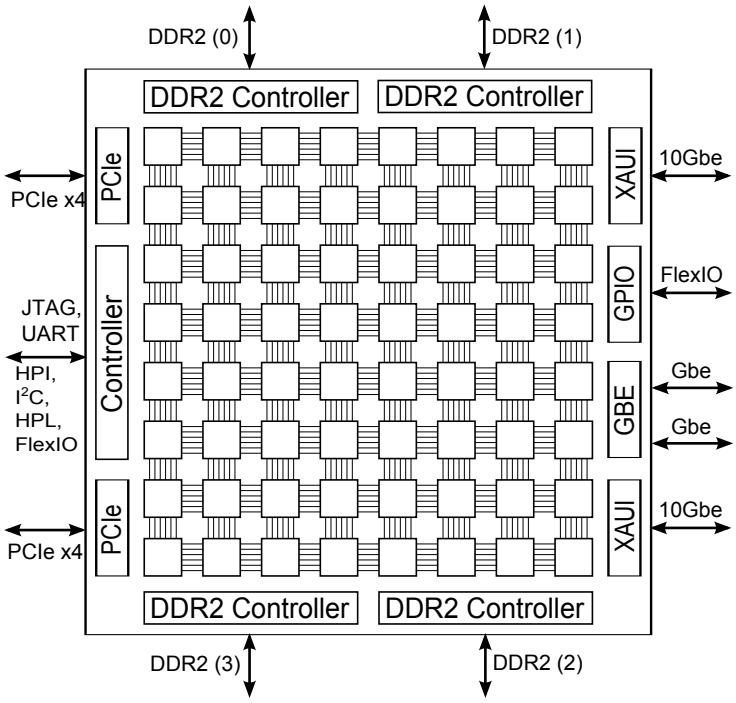


Figure 4.2: Simplified Tilera hardware architecture [2]

with Tilera's iMesh^{TM} on-chip network. Each *tile* is a full featured processor and as shown in Fig. 4.3 consists of three main parts: processor engine, cache engine, and switch engine. The processor engine is a 32-bit 3-way VLIW processor with two/three instructions per bundle. The cache engine provides L1 instruction cache, L1 data cache, and combined L2 cache for each tile. In addition, the cache engine is equipped with a DMA controller which provides facilities for fast data transfer between tiles and between tiles and external memory. The switch engine provides connection to the iMesh on-chip network. Since the tiles are laid out in a two dimensional mesh, the switch engine connects the *tile* to the north, south, east, and west neighbors. The switch engine directly connects to IO devices if the tile is adjacent to an IO device. In the following, more details on TILEPro64 memory organization and on-chip communication architecture is provided.

Memory organization TILEPro64 architecture provides a 36-bit physical address space which is globally shared between all 64 tiles. To balance memory bandwidth, the physical address space is distributed into four different DDR2

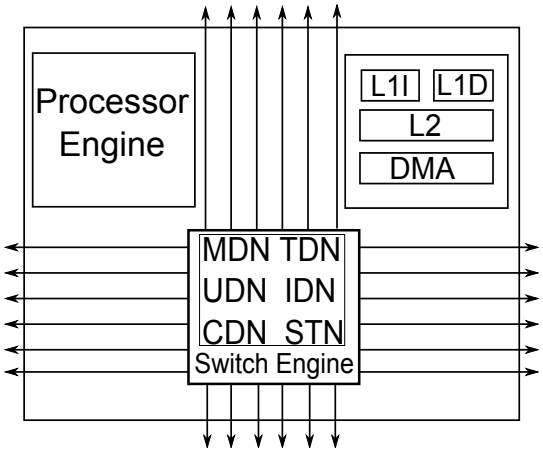


Figure 4.3: Simplified *Tile* architecture [2]

RAMs. This feature allows parallel read/write operations from different memory modules. In addition, locating tasks on the tiles near memory controllers they use can maximizes memory network bandwidth by avoiding unnecessary link congestion. Actually, being on the same left/right half of the chip as the memory controller is particularly helpful [83].

On the TILEPro64 processor, each tile cache system includes 16KB of L1 instruction cache, 8KB of L1 data cache, and 64KB of L2 combined cache. In addition, the processor provides facilities for a dynamic distributed cache (DDC) system. DDC system allows a page of shared memory to be homed on a specific tile or distributed across several tiles, then cached remotely by other tiles. Therefore, if the requested data is not found in local L1/L2 cache of the tile, at first the cache of the home tile is searched and in case of data miss, the request is passed to the external memory. This mechanism allows a tile to view the collection of on-chip caches of all tiles as a large shared, distributed coherent cache. The cache coherency of DDC is supported in hardware. Data access time for L1, L2, distributed coherent cache and external memory is equal to 2, 8, 35 and 69 cycles, respectively. Therefore, in order to achieve better performance, data distribution should be performed in such a way to maximize local memory access.

Finally, it should be mentioned that the Tile processor architecture defines a relaxed memory consistency model. Consequently, non-overlapping memory accesses to the shared pages of the memory from one tile can be reordered and

can become visible to other tiles sharing that page in an order different from the original program order with a few restrictions (for details see [2]). The Tile architecture provides the memory fence and test-and-set instructions. The former can be used to establish ordering among otherwise unordered instructions, and the latter can be used to read and write a memory location atomically.

Communication architecture The on-chip interconnection network, iMesh, is responsible for data transfers between tiles, between tiles and external memory, and between tiles and IO devices.

As shown in Fig. 4.3, the iMesh consists of a single static network and five dynamic networks. The static network uses circuit switching mechanism to establish a path between source and destination. This network is user accessible and is suitable for streaming scalar data between tiles. The dynamic networks can be classified into two groups: memory networks and messaging network. The memory networks handle all memory traffic such as cache misses, DDR2 requests, and so forth. The memory networks consists of the Memory Dynamic Network (MDN), the Tile Dynamic Network (TDN), and the Coherence Dynamic Network (CDN). The messaging networks are User Dynamic Network (UDN) and the IO Dynamic Network (IDN) and allow the user level program to control the network and transfer data between tiles or to/from IO devices. It should be mentioned that Tilera API library, iLib, provides a set of functions to use underlying interconnect network for data transfer between the tiles.

4.4 Parallel Formulation

In this section, the parallel formulation of event-based optical flow is presented. Decomposition, mapping, and minimizing interaction overhead are the fundamental steps to solve a problem in parallel [84]. To parallelize event-based optical flow, like any other problem, there were several choices for each of these steps. In the following, the main alternative choices along with their advantages and disadvantages are discussed.

Computing optical flow on the Tilera architecture in the abstract view can be seen as a three stage pipeline: *receiving events* on the board, *computing ve-*

locity vectors, and *sending the velocity vectors out* to the host machine. Since these stages produce and consume data at different rates, the producer-consumer pattern is best suited to model this computation. Producer-consumer pattern decouples processes that produce and consume data at different rate. In this pattern, queue data structures are used to communicate data between producer and consumer processes.

The main computation is performed in the second stage of the pipeline, computing velocity vectors. In the following we will discuss how to exploit parallelism in computation of velocity vectors. However, here suffice it to say that this process will be further divided into smaller tasks which will be assigned to multiple processors running in parallel, i.e. the events received in the first stage of the pipeline should be distributed among several tasks which compute velocity vectors. Hence, one stage must be added to the three stage pipeline, between the first and second stages. The parallel model of optical flow computation is illustrated in Fig. 4.4. The events are arrived to the board via PCIe or network ports. The *receiver* process is responsible to catch the arrived packets and save them in a circular queue. The *distributor* process processes the arrived packets and assigns each event to the corresponding task of the *velocity* process. Then, subprocesses of the *velocity* process compute velocity vectors and add the computed results to an output buffer. The output buffer is sent out to the host by the *sender* process through PCIe or network ports.

The parallel model presented in Fig. 4.4 is a high level model. The *distributor* and especially *velocity* processes can be further divided into smaller tasks which will be performed in parallel. In the following, first the shared data structures, which are used to communicate between processes in our parallel model, are described, then parallelization of the *velocity* and *distributor* processes are discusses.

4.4.1 The concurrent data structures

As mentioned above and can be seen in Fig. 4.4, three queue structures are used to communicate data between producer and consumer processes in our pipeline model. Circular queue structures are used to communicate data between the

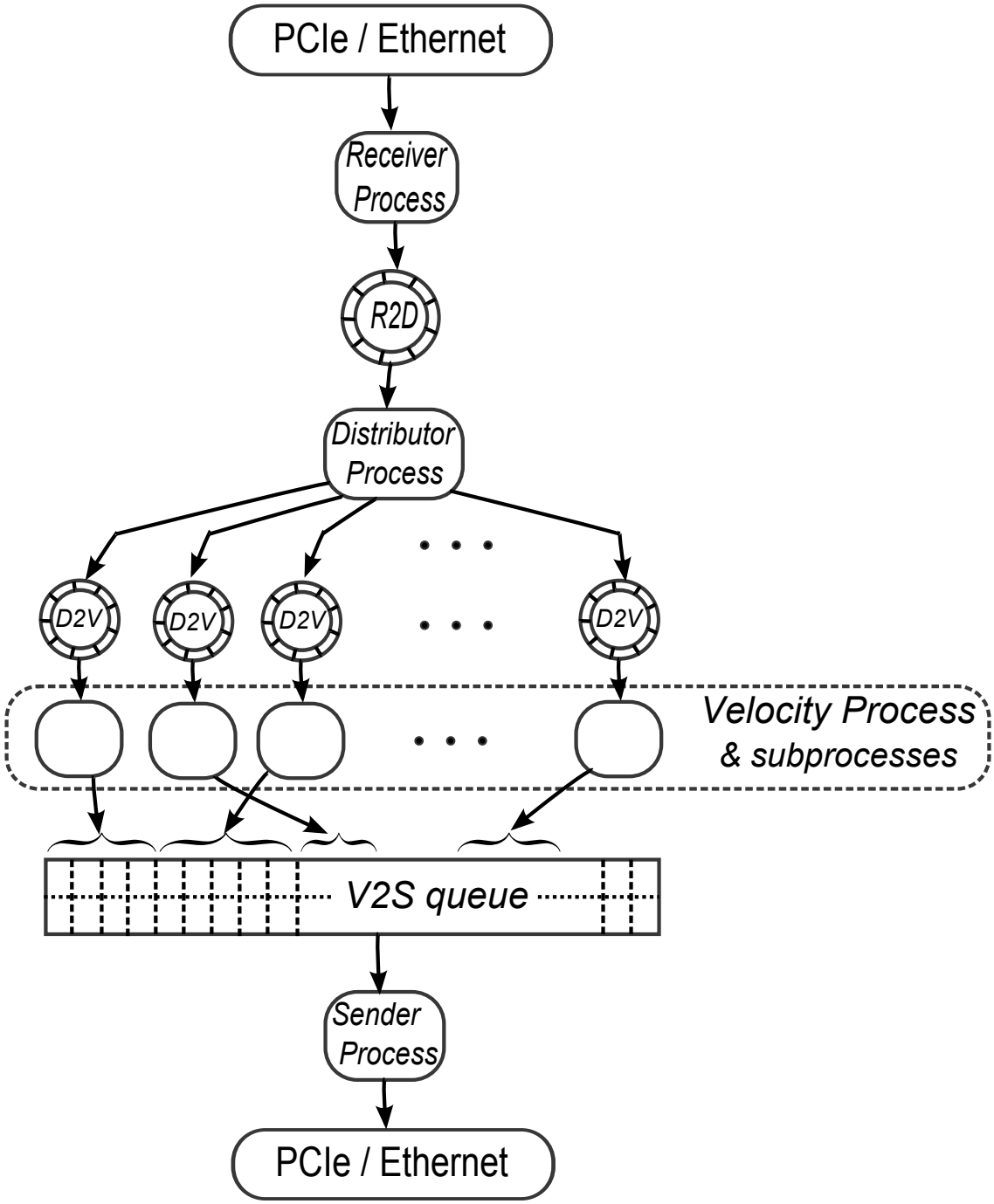


Figure 4.4: Parallel computation of event-based optical flow

receiver and the *distributor* processes and also between the *distributor* process and each subprocess of the *velocity* process. This circular queues are called *R2D* and *D2V*, respectively. Finally, an extended queue data structure, called *V2S* queue, is used to communicate between the *velocity* subprocesses and the *sender* process.

These queues are shared data structures. In classic lock-based multithreaded programming, the operations that change shared data must appear as atomic by locking and unlocking a mutex, such that no other thread intervenes to spoil the data's invariant, i.e. the access to the shared data must be serialized by locking and unlocking a mutex. However, this serialization can have a major impact on the performance and achievable speedup of the parallel application [85]. One solution is to use lock-free data structures[1], but it is very difficult to implement lock-free data structures and prove their correctness.

However, under sequential consistency memory model [86], Lamport [87] has proved that a wait-free single-producer single-consumer circular queue can easily be implemented without using explicit synchronization mechanisms between the producer and the consumer. Higham and Kavalsh [88] have formally proved that even under weakly ordered memory consistency model, the Lamport's algorithm with simple modification works correctly. The main idea of the algorithm is to couple control and data information into a single buffer operations by using a known value, which cannot be used by the application.

In our proposed parallel model, both *R2D* and *D2V* structures are single-producer single-consumer circular queues. Since, as mentioned in section 4.3, the Tilera architecture defines a relaxed memory consistency model, we have exploited the Higham and Kavalsh's algorithm [88] to implement a lock-free structure for *D2V*. However, since *R2D* circular queue is used to store the arrived packets, Highamand kavalsh's algorithm cannot be used to implement this queue. To write in *R2D* queue, *receiver* process first have to reserve a space in the queue and then start writing the data. Also, to read data, *distributor* process have to

[1] There are two types of non-blocking thread synchronization mechanisms: wait-free and lock-free. Wait-free guarantees the progress of each subtask or threads of the process, regardless of the behavior of other threads. Lock-free guarantees the progress of the process, i.e. while a given thread might be blocked by other threads, all CPUs can continue doing other useful work without stalls. Wait-free is a stronger condition.

lock the associated element of the queue.

Finally, lock-mechanism is used to implement *V2S* queue. By using double buffering techniques, however, the implementation of the *V2S* queue allows the *sender* thread to send out the computed result while the *velocity* subprocesses can write the computed velocity vectors on the other parts of the queue.

4.4.2 Velocity computation

In this section, parallelization of the *velocity* process is discussed. In the following, r and c denotes the number of rows and columns of the image plane, respectively. An ordered pair (x, y) represents the pixel located at row x and column y in the image plane. Also, $e(x, y, t)$ denotes an event arrived at pixel (x, y) at time t. Finally, the size of the neighborhood considered around each event to compute velocity vector is $n \times n$. In the following, we discuss different decomposition schemes for parallelization of the *velocity* process.

Event-based decomposition. The *even-based* decomposition is illustrated in Fig. 4.5. This decomposition scheme is based on partitioning the input data, i.e. one task is created for each event $e(x, y, t)$, and this task performs all the computation required to calculate velocity vector at position (e.i. pixel) (x, y) and time t. Arrived events can be related to any positions in the image plane. To compute velocity vectors for the events arrived in a $n \times n$ neighborhood, as can be seen in Fig. 4.5, the associated tasks need to access shared data. In this scheme, the data related to stage one and two of event-based optical flow computation (see Fig. 4.1), in fact a large portion of data, should be stored in the shared memory space. The local data of each task only includes the partial derivatives and least square measurement matrices.

Since tasks are generated dynamically upon arrival of events, a dynamic mapping technique is required. Actually, since all the tasks are of equal size, a simple dynamic mapping technique can provide overall load balancing among available processors. However, as already mentioned, the tasks related to neighboring events (in space and time) need to access the shared data to perform their computations, consequently synchronization among these tasks is required, i.e. some

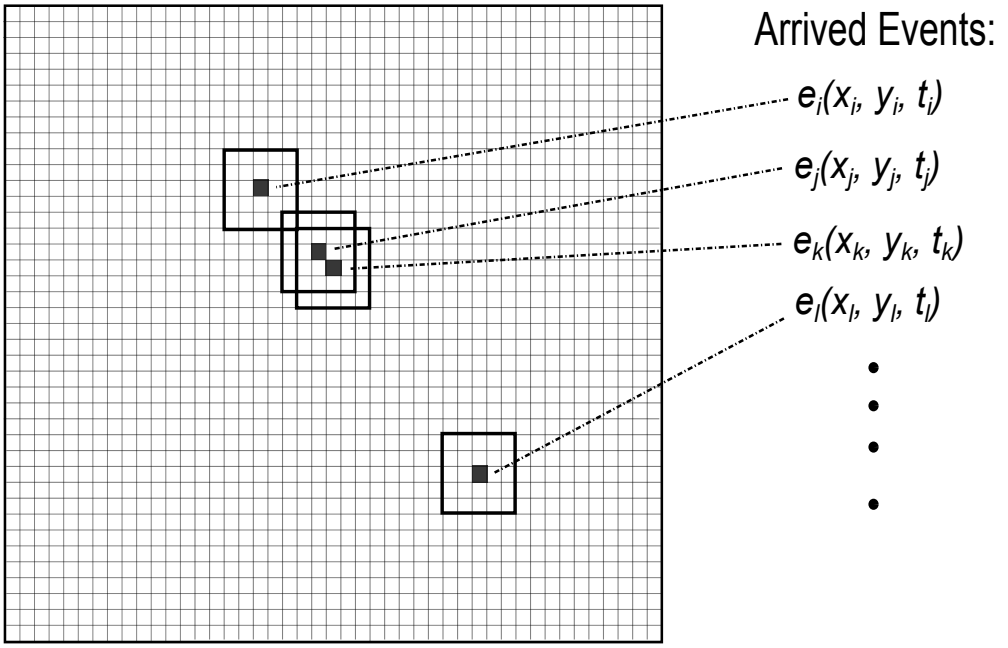


Figure 4.5: Event-based decomposition

task must wait for the other tasks to complete their process. Thus, the synchronization cost will increase the computation time and will outweigh the advantage of overall load balancing.

Velocity-based decomposition. As discussed in section 4.2, each velocity vector can be computed independently of others as a function of input events in a small neighborhood. This lead to velocity-based decomposition which actually partitions the output data. The *velocity-based* decomposition is illustrated in Fig. 4.6. In this scheme, a task is created for each pixel (x, y) at the image plane. This task is responsible to compute the velocity vector if an event arrives at this position. As shown in Fig. 4.6, each task keeps a local copy of events which arrive in the $n \times n$ neighborhood centered on its corresponding pixel. Consequently, all these tasks, even those assigned to neighboring pixels, can perform computation using only their local data, i.e. the synchronization and inter-process communication are minimized in cost of increasing memory usage. In fact, the required memory space for this scheme is $O(rcn^2)$, while it is only $O(rc)$ for the *event-based* decomposition scheme.

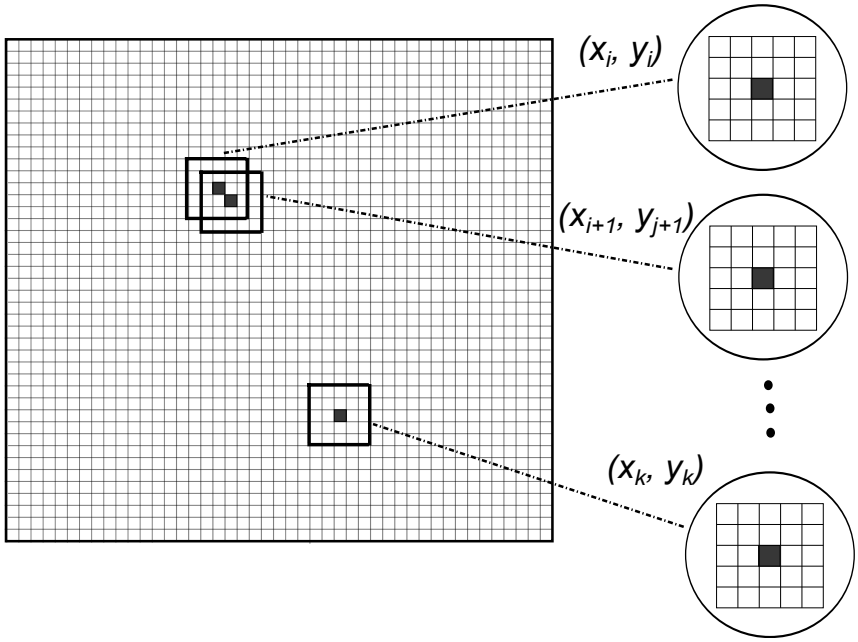


Figure 4.6: Velocity-based decomposition

Since the tasks are statically generated, either static or dynamic mapping can be used. The commonly used techniques for distributing arrays or matrices among processors such as block distribution can be used to statically map the tasks to the available processors. However, these techniques may lead to an unbalance computation especially when the input data has some special pattern. For example consider there is a set of 8 processors $\{T_1, T_2, \cdots T_8\}$ and a black distribution mapping as shown in Fig. 4.7 is used to map the tasks to the processors. If there are only events in the left side of the image, some processors (T_3, T_4, T_7, and T_8) will not get any work, while the processor on the left (T_1, T_2, T_5, and T_6) are assigned much of the work. A randomized distribution can improve the load balancing among processors [84]. While a dynamic mapping can provide overall load balancing, it entails moving the data associated with tasks among processors. This data movement[1] is costly and may render a static mapping more suitable.

[1]on Tilera architecture, the associated data have to move from one cache to another

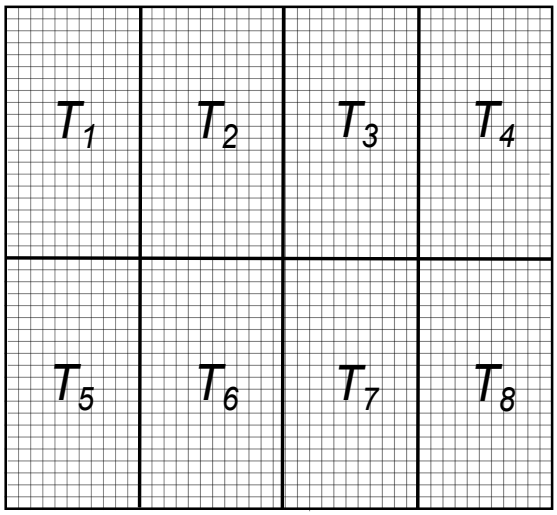


Figure 4.7: Velocity-based decomposition and mapping to eight processors

Image plane decomposition. *Image-plane* decomposition is illustrated in Fig. 4.8. This scheme is actually based on partitioning intermediate data (see Fig. 4.1, stage three of event-based optical flow computation). As shown in Fig. 4.8, the image plane is partitioned into 2-D blocks, and one task is assigned to each block. Each task is responsible to computes the velocity vectors at all pixels of the block, and only performs computation on local data. However, to compute velocity vector in boundary areas of each block, data from neighboring blocks are required. In Fig. 4.8 the gray shaded areas represent boundary area of the *block i*. To avoid data-exchange overhead among tasks, as also shown in Fig. 4.8 for the *black i*, each task keeps track of events arrived in the boundary area of its neighboring blocks, in addition to its own block data. Consequently, all the tasks can work asynchronously and minimize inter-process communication in cost of a small increase in memory space usage.

Since the tasks are known prior to the execution and on the other hand the size of data associated to each task is relatively large, static mapping is the best choice for this data decomposition scheme. Since for a real robotic application the tasks are expected to have almost the same size, a simple mapping technique such as equally distributing blocks among available processors can provide overall load balancing. However, as discussed for velocity-based decomposition, if the input has some special pattern, this general mapping technique may lead to unbalance

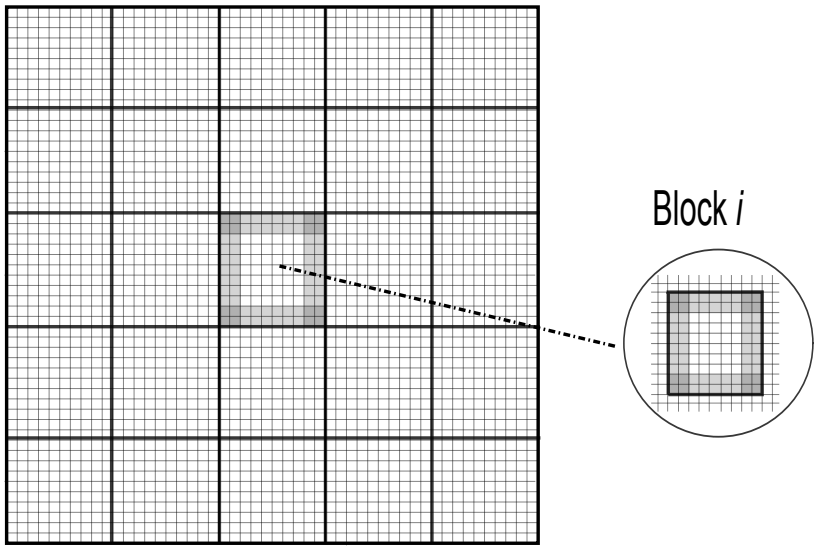


Figure 4.8: Image plane decomposition

computation. When the input has a fixed known pattern, it is possible to define a static mapping technique which provide load balancing. It should be mentioned that a randomized distribution will be helpful only if number of blocks is much more than number of available processors.

Appropriate decomposition scheme. As can be seen, different decomposition and mapping techniques can be considered for parallelization of the *velocity* process. For our parallel implementation on the Tilera MIMD architecture, we have chosen the image plane decomposition. Both Event and velocity vector based schemes are fine-grained decomposition and provide highest degree of concurrency. However, they have several drawbacks, especially they increase memory access time and perform redundant computation.

The events sent by DVS camera have spatial and temporal locality, i.e. when an event arrived at one pixel, it is likely that more events will arrive at that pixel and its neighbors. Actually an isolated event is considered as noise generated by the sensor and is dropped before further computations [25].In image-plane decomposition scheme, this spatial and temporal locality of events can be mapped to spatial and temporal locality assumption used in cache subsystem of computing platforms to decrease memory access time significantly.

In addition, locality of events also means that computation of derivatives, stage three of event-based optical flow (see Fig. 4.1), can be saved for neighboring pixels. However event and velocity-based decomposition cannot exploit this feature, since the computation related to neighboring pixels are assigned to different tasks[1].

Finally, the number of tasks generated by velocity based decomposition is equal to rc i.e. it depends on the image plane size (in case of DVS128 it is equal to $128 * 128 = 4094$), but our target hardware architecture includes only 64 processors. This means that we cannot benefit from very high level of concurrency provided by this decomposition technique.

Image-plane decomposition is a coarser grain scheme, however it still provide good degree of concurrency. Furthermore it reduces parallelization overheads (inter-process communication overhead), and saves computation and memory access time by exploiting temporal and spatial locality of events.

4.4.3 Event distribution

As shown in Fig. 4.4, the *distributor* process pops the arrived packets from the *R2D* circular queue, then it has to decode the events and assign them to the corresponding *velocity* subprocess to compute velocity. For parallelization of the *distributor* process, different decomposition schemes can be considered.

Like the *velocity* process, an image plane decomposition can be used. However, such scheme is not efficient for the implementation of *distributor* process, since the subprocesses have to perform redundant computation to decode the arrived events.

A coarse grain data decomposition may assign one task to each arrived packet in the queue. While this is an efficient scheme to exploit parallelism for *distributor* process, it will cause unordered computation of velocity vectors in the next stage of pipeline (the *velocity* process), and therefore, it is not acceptable.

A fine grain data decomposition scheme may distribute events of one packet among available processes based on their indexes in the packet. Assume d_n denotes the number of the subprocesses available for the *distributor* process. Then

[1] They can only save computation in cost of increasing inter-process communication.

subprocess j $(1 \leq j \leq d_n)$ is assigned to compute all the events in the packet at index i where $i = j \mod d_n$. This fine grain decomposition can be employed for parallel implementation of the *distributor* process.

However, the side-effect of this scheme is that in our parallel model, shown in Fig. 4.4, a multi-producer single-consumer circular queues structure will be required to communicate between the *distributor* and *velocity* subprocesses. As discussed in section 4.4.1, lock-based data structures can have a major impact on efficiency and performance of parallel applications, and look-free circular queues for multiple producer and consumers are much harder than single-producer single-consumer queues to implement. On the other hand, the *distributor* process does not perform intensive computations. Therefore, the better choice is to not decompose the *distributor* process in smaller subtasks, and instead have the possibility of using lock-free single-producer single consumer circular queues to communicate between the *distributor* and the *velocity* subprocesses[1].

4.5 The application architecture and implementation overview

In this section, we briefly describe the parallel implementation of event-based optical flow application on the Tilera architecture. Figure. 4.9 depicts the high level application architecture. The DVS vision sensor sends the events through Yarp network. a Yarp module running on the host machine is responsible to receive the events that are sent by the DVS sensor. This module continuously communicates the received events to the Tilera processor and receives the computed flow from Tilera. There are two ways to communicate with the Tilera processor: PCIe bus and Ethernet ports. In our implementation, the communication with Tilera is performed via PCIe bus. Finally, the host module provides the computed flows on a Yarp port, so the computed flows can be used as input by other modules (For example yarpview in Fig. 4.9).

[1] If the *distribute* process becomes a bottleneck in our parallel application, the implementation of the *distributor* process and the associated circular queues can be modified later. These modifications will not affect the other processes, since the producer-consumer pattern decouples the processes.

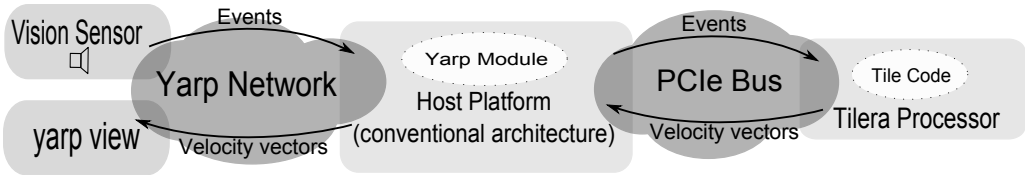


Figure 4.9: Application general overview

In the following we discuss two important issues in parallel implementation of event-based optical flow application on the Tilera architecture: host-tile communication, and memory allocation on the Tilera architecture.

4.5.1 Host-tile communication

The Tilera PCIe user space communication API provides several communication mechanisms, each suited for different usage model (for details see [89]). Among all, we have chosen zero-copy command copy API which provides a reasonable trade-off between simplicity and efficiency for our application. This API allows programmer to send commands directly to PCIe driver's scatter/gather engine by writing to a Linux device file associated with a PCIe channel. Reading from the device files returns a structure indicating the completion status of a previously written commands. However, this API relies on execution of significant Linux and hypervisor code (e.g. file operation and interrupts). Therefore, this data transfer process is not efficient for packets smaller than 8 KB.

Our tests performed on the DVS sensor showed that in the high load condition DVS sensor generates events with the rate of approximately 800,000 event per second. According to [89], it takes 20 and 10 μs to send 16KB and 8KB of data over PCIe channel, respectively. Since each event sent by the vision sensor requires 8 bytes of data, data can be sent to the Tilera with the rate of 100 mega-events per second. Therefore, the host-tile communication is not a bottleneck in the application performance.

4.5.2 Memory allocation strategy on Tilera

In the Tilera architecture, as mentioned in section 4.3, the union of the L2 caches serves as the distributed L3 cache. This has become possible by use of *home tile*

mechanism. The *home tile* is a tile designated to maintain and track sharing and coherence information for a particular physical address. In the Tilera architecture, each physical address of memory is associated with a home tile. The Tilera memory subsystem provides three different homing strategies: *local homing*, *remote homing*, and *hash-for-home*. With the *local homing* strategy, the entire page of memory is homed on the same tile that is accessing the memory. Therefore, when an access to address P misses in the L2 cache, a request is directly sent to DDR memory in order to retrieve the data. With the *remote homing* strategy, the entire page of memory is homed on a different tile than the tile accessing the data. In this scenario, when an access to address P misses in the L2 cache, a request is sent to the home tile (L3 cache). If the data is present in the home tile, then the request is serviced directly with the data from the home tile. If the data is not present, a request is sent to memory and the memory controller services the request. Finally, the *hash-for-home* strategy is very similar to the *remote homing* strategy. The difference is that the entire page of memory is hashed across a set of tiles within the system, at a cache line granularity (for more details see [83]).

The *home tile* strategy has an important impact on the data access time and consequently on the overall application performance. The *local homing* strategy is suitable for private data that would not benefit from using another core's cache as a backing L3 cache. The *remote homing* is suitable for shared memory FIFOs with single consumer. By homing the FIFO data on the consuming tile, the producer tiles can write directly to the consuming tile's cache. Therefore, the data structures can take advantage of two important principles: (1) loads are faster when issued to data that is homed on the issuing tile, (2) stores are fire-and-forget, i.e. storing to a remote cache has minimal performance impact because the main processor does not have to wait for a read result to come back. Finally, *Hash-for-home* enables an even traffic distribution through the on-chip networks, and effectively balance the cache traffic across a set of tiles. Consequently it tends to be the best for any data or instruction memory that is shared across multiple threads or processes.

Therefore, in our parallel implementation, the *remote homing* strategy should be used for the *R2D*, *D2V*, and *V2S* queues, while the *local homing* strategy should be used for the private data of each process or subprocess.

4.6 Results and performance of event-based optical flow implementations

We have implemented the event-based optical flow algorithm [76] on both general purpose processors(GPPs) and the Tilera architecture. In this section, first the output of both optical flow implementations for some tests performed on iCub humanoid robots are presented, then we discuss the performance of the two implementations in terms of speed. In terms of optical flow accuracy, we relay on the results presented in [76], i.e. we do not pursue this issue here.

Results of the GPP implementation of event-based optical flow

The GPP implementation of event-based optical flow has been extensively tested on the iCub humanoid robots and iKart mobile platform. In the following, the output of the implemented algorithm in different situation and with various stimuli is presented.

Rotating circle. The stimulus is shown in Fig. 4.10(a). This stimulus was rotating in front of the DVS vision sensor in a clockwise direction with fixed angular velocity. Rotation of this circle triggers events in the two edges of the black quadrant. Snapshots of the event-based optical flow output is presented in Fig. 4.10(b)-4.10(d). Even if this is a simple stimuli, it can generate flow vectors in many different directions and magnitudes. From the basic physics, it is known that the flows which are closer to the center of the circle should have smaller magnitudes than those which are close to the boundary of the circle. As can be seen in Fig. 4.10, in the output of event-based optical flow algorithm, the magnitude of flows increase from the center of the circle to the border.

iCub head movement. In this experiment, the environment is static and DVS vision sensor generates events since the iCub robot turns its head to the right and left as shown in Fig. 4.11(a). Field of view of iCub is shown in Fig. 4.11(b). Figure 4.11(c)-4.11(e) present the output of the algorithm when the head turns

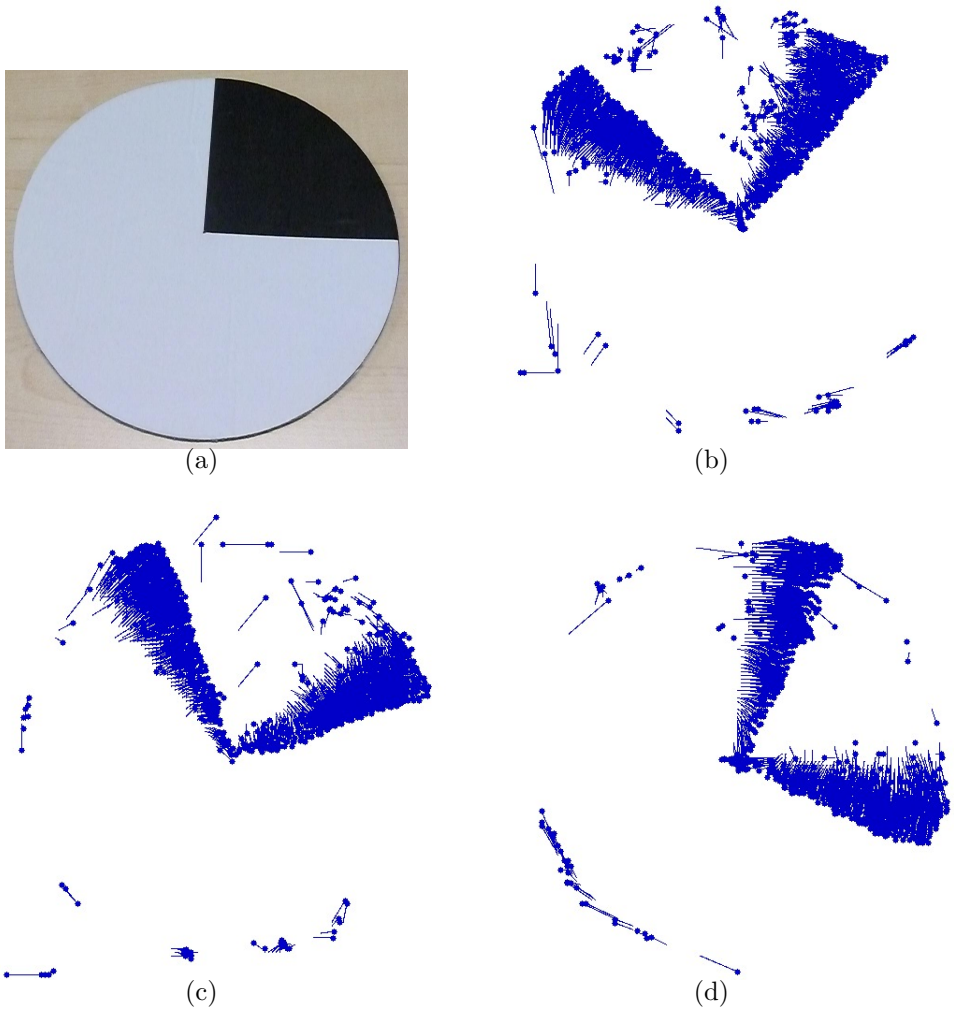


Figure 4.10: Results of GPP implementation of optical flow for a clockwise rotating circle

to the right, while Fig. 4.11(f)-4.11(h) illustrate the output when the head turns to the left. The magnitude of the flow vectors are almost of the same size, which is consistent with the experiment setup, since the curtain and the human which are in field of view of the iCub are almost in the same distance of the iCub and iCub turns its head with constant velocity.

Human walking. This experiment presents the output of the event-based optical flow when an human pass the iCub robot. In this experiment, the iCub head does not move. Figure 4.12(a)-4.12(h) illustrate the output of the algorithm when the human pass the iCub. As the human goes further, the number of events gradually decreases. Less number of events lead to an inaccurate estimation of optical flow. As can be seen in Fig. 4.12(h), at some point the computed flows does not provide the correct information.

Application of event-based optical flow in robot obstacle avoidance

The event-based optical flow implementation was exploited to avoid obstacle in the robot navigation task. In this experiment, we have used a DVS camera mounted on the iKart mobile platform. Knowing the movement and geometry of the system, an velocity field model was built. This model was continuously checked with the optical flow output. A difference between the model and optical flow field indicates an object. Snapshots of the conducted experiments are presented in Fig. 4.13. As shown in Figure 4.13(a)-4.13(c), iKart platform is moving that a human pass the robot (Fig. 4.13(d)-4.13(f)). The obstacle (the human) is detected and the iKart stop moving (Fig. 4.13(g)-4.13(i)). Then, in ig. 4.13(j)-4.13(m), iKart continues its movement by permission of human supervisor.

Results of the parallel implementation of event-based optical flow on the Tilera architecture

In this section, we present the output of parallel event-based optical flow for the same set of stimuli used to test the GPP implementation. Figure 4.14(a)-4.14(c)

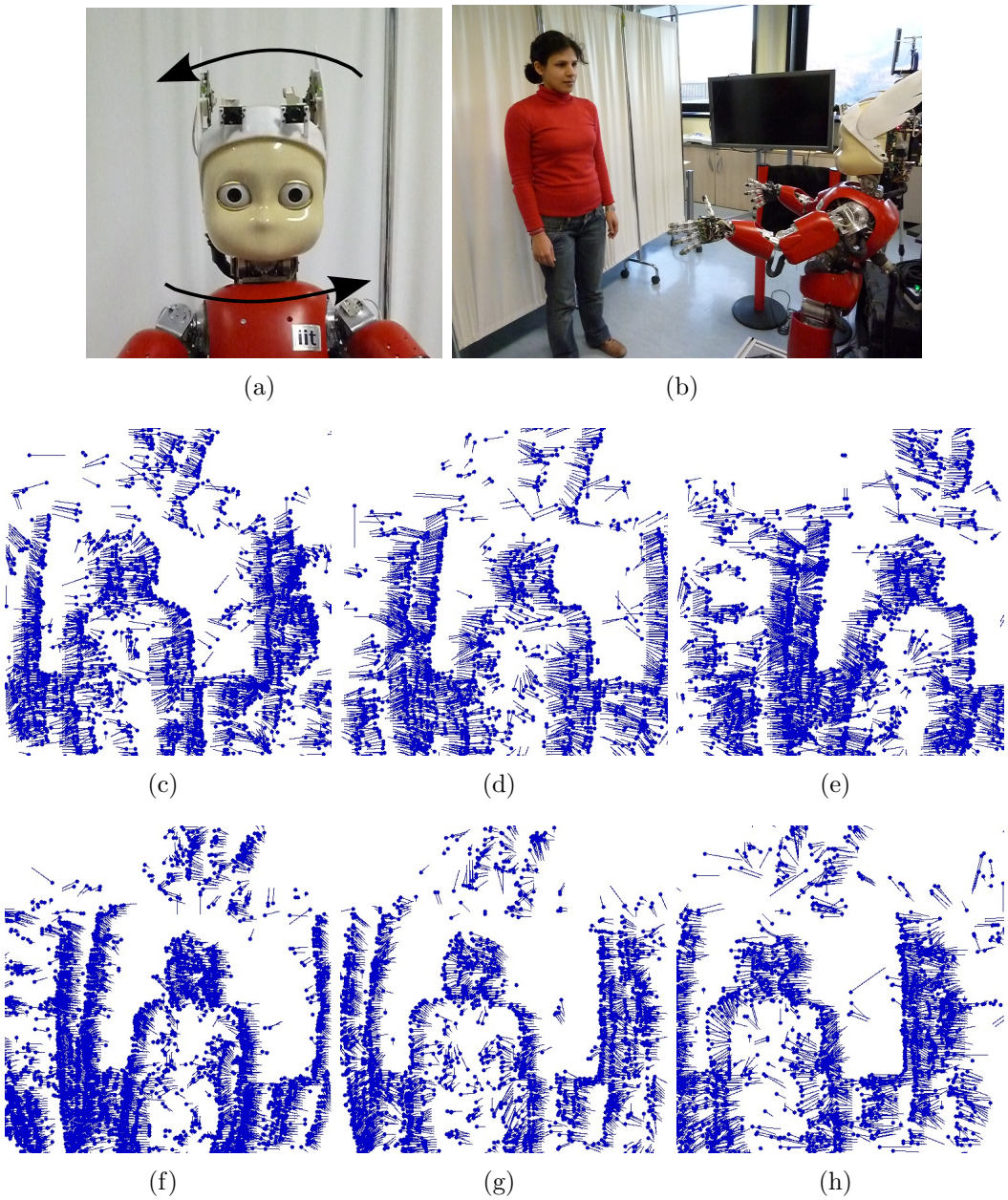


Figure 4.11: Results of GPP implementation of optical flow when iCub turns its head to the left and right

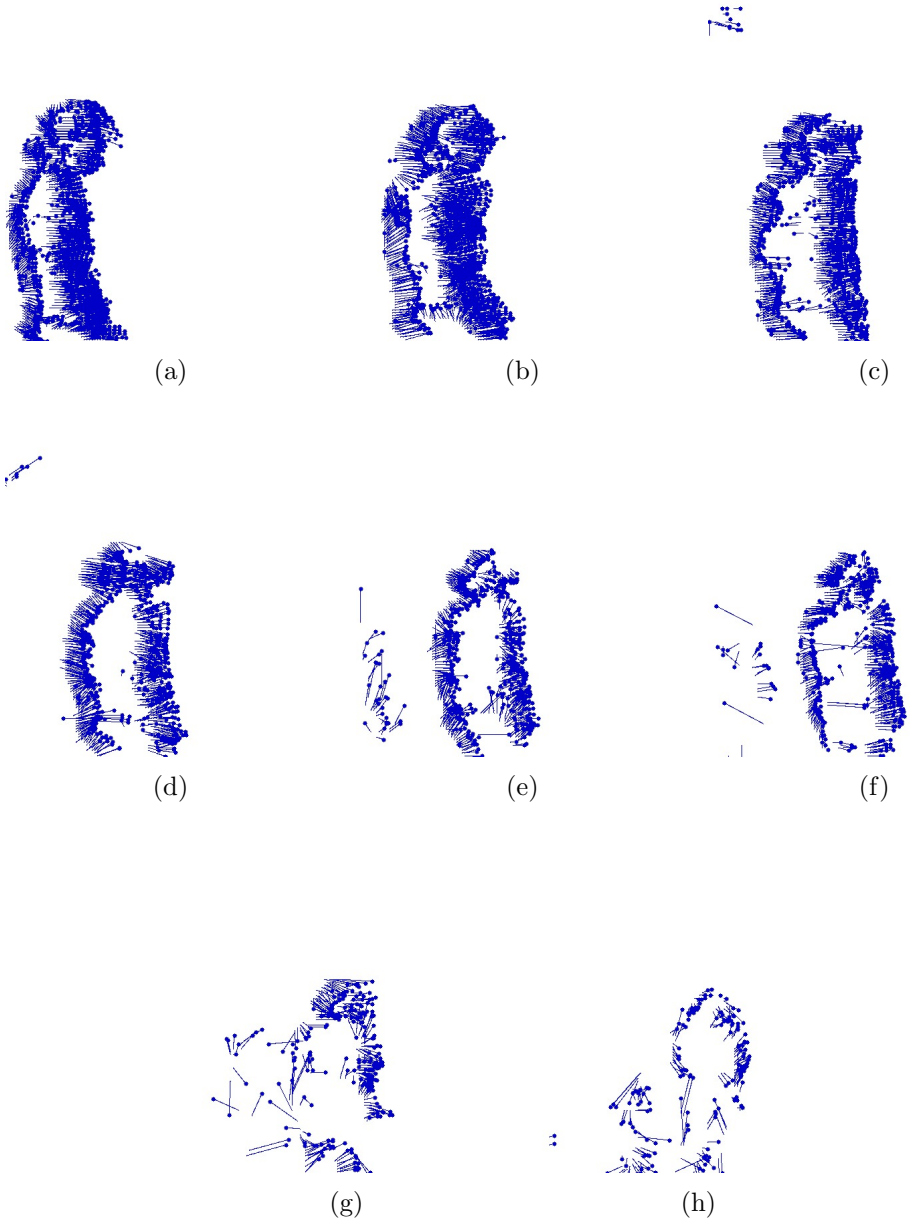


Figure 4.12: Results of GPP implementation of optical flow when a haman passes iCub

(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

(m) (n) (o)

Figure 4.13: Application of event-based optical flow in robot obstacle avoidance system

present the output of the rotating circle with a black quadrant. Figure 4.14(d)-4.14(f) present the output of the parallel optical flow algorithm when the icub turns its head to the left and right. The output of the parallel algorithm for the human passing the iCub are illustrated in Fig. 4.14(g)-4.14(i).

Performance comparison of GPP and Tilera implementations of event-based optical flow

To compare performance of the GPP and Tilera implementations of the event-based optical flow in terms of speed, we have computed the throughput of both systems for the rotating circle presented in Fig. 4.10(a). The throughput was computed as the number of computed velocity events per second. The GPP implementation was run on a 2.67 GHz Intel Core^{TM} i7-920 CPU which is of the state of the art conventional processors, while the Tilera implementation was run on 60 cores of a 860 MHz TilePro64 processor. Figure 4.15 compares the throughput of the GPP and Tilera implementations. As can be seen the throughput of the Tilera implementation is more than twice as the GPP implementation throughput. It should be mentioned that the power consumption of the TilePro64 processor is between 15-22 watt while Intel Core i7-965 power consumption is 130 watt.

Impact of block size on the performance of the Tilera implementation

As discussed in section 4.4.2, we have chosen the image plane decomposition scheme for parallelization of the *velocity* process. In this scheme, the image plane is partitioned into 2-D blocks, and one task is assigned to each block. However, we have not discuss the appropriate block size so far. The size of the 2-D blocks determines the number of required *velocity* subprocesses and can have a major impact on the performance. To investigate the impact of block size on the performance of the system, the computation of optical flow was performed on the same input but with different choices of block size. Figure 4.16 compares the throughput of different runs of the Tilera implementation of event-based optical

Figure 4.14: Results of Tilera implementation of event-based optical Flow

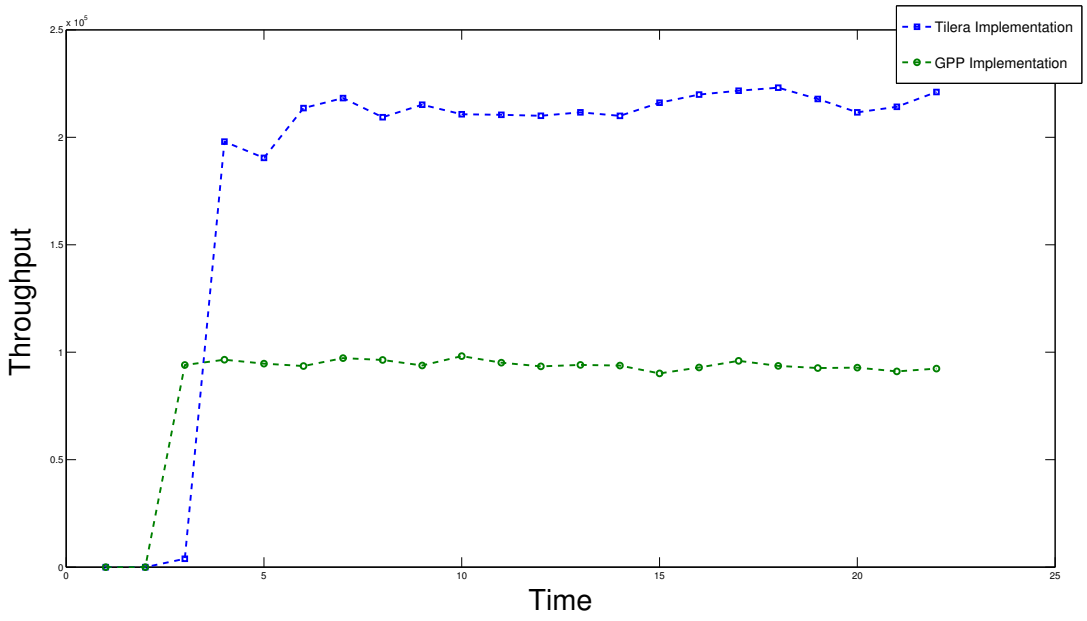


Figure 4.15: The throughput of GPP and Tilera implementations of event-based optical flow

flow algorithm for the rotation circle input (Fig. 4.10(a)) and the following block sizes: 16 × 16 (64 *velocity* subprocesses), 16 × 32 (32 *velocity* subprocesses), 32 × 16 (32 *velocity* subprocesses) 32 × 32 (16 *velocity* subprocesses), 16 × 64 (16 *velocity* subprocesses) 64 × 16 (16 *velocity* subprocesses), and 64 × 64 (4 *velocity* subprocesses).

The results illustrated in Fig. 4.16 indicate that by reducing the block size and consequently increasing the number of *velocity* subprocesses, the throughput increases. Moreover, as can be seen in Fig. 4.16 the throughput of the system is almost the same for the following cases: (a) block sizes of 16 × 32 and 32 × 16 (i.e. 32 *velocity* subprocesses), and (b) block size of 32 × 32, 16 × 64, and 64 × 16 (i.e. 16 *velocity* subprocesses). Thus, as long as the number of the subprocesses does not change, the block size change does not have an impact on the throughput.

Another important observation is that increasing the number of *velocity* subprocesses from 16 to 32 results in 50 % increase in the system throughput, while increasing the number of subprocesses from 32 to 64 leads to only 10% increase

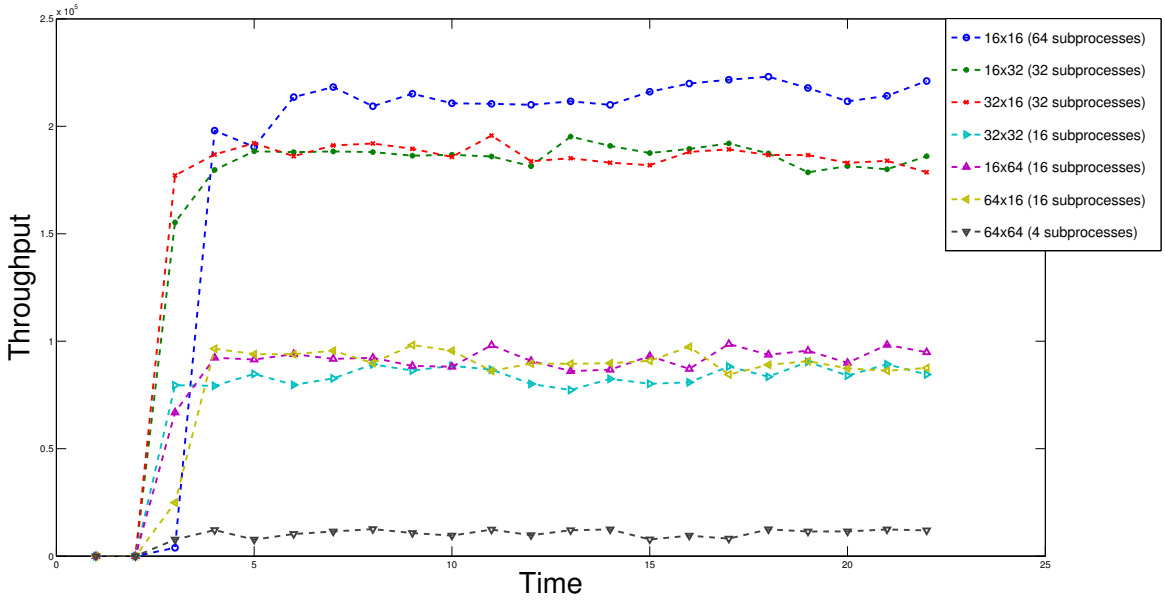


Figure 4.16: Impact of block size on throughput of Tilera implementation of event-based optical flow

in throughput of the system. Since maximum of 60 cores on the Tilera architecture can be used to run user programs, by increasing the number of *velocity* subprocesses more than the number of available cores, the cost of thread switching will increase. There are also other parallelization overheads such as storing the boundary data for the *velocity* subprocesses. This trend in throughput increase proves that a very fine grain computation is not suitable for our parallel implementation on the Tilera architecture.

4.7 Summary

This chapter studies parallel formulation of an event-based optical flow algorithm [76] as an example of event-based vision processing algorithms. Various decomposition and mapping schemes were analyzed for an efficient parallel implementation with minimum parallelization overhead on the Tilera MIMD architecture. It was shown that a pipeline or producer-consumer model is the suitable choice for par-

allel computation of event-based optical flow, and consequently the computation was modeled in a four stage pipeline: receiving events, distributing events, computing velocities, and sending the results back. Further decomposition of the pipeline stages was discussed and it was shown that image-plane decomposition is the most suitable scheme to exploit parallelism in velocity computing stage. This scheme on one hand reduces inter-process communication overhead and on the other hand saves computation and memory access time by exploiting temporal and spatial locality of events. A consequence of our parallelization strategy is that we can use lock free single-producer single consumer circular queue for data communication between stages of the pipeline.

The analyses undertaken in this chapter clearly shows that parallel formulation of event-based vision applications may be more complicated than conventional image processing tasks due to two reasons. First, commonly used data decomposition scheme can lead to unbalance computation, and consequently inefficient parallelization. Second, due to the sparse nature of data, it is more difficult to exploit the cache subsystem of computing platforms effectively, and hence the memory communication overhead may increase.

Along with parallel formulation and implementation of event-based optical flow on the Tilera MIMD architecture, a GPP implementation of even-based optical flow was provided for the mobile iKart platform and iCub robot. Although the GPP implementation compared with the Tilera implementation is more limited in terms of computed event per second, it can provide satisfactory performance for some applications. Actually, it has been successfully used in a real-time obstacle avoidance system.

Chapter 5

Optical Flow Visual Cues for Robot Navigation

5.1 Introduction

This chapter studies the application of the event-based optical flow, presented in chapter 4, in obstacle avoidance for mobile robots and humanoids. Patterns of optical flow contain information about self-motion, moving objects and 3-dimensional (3D) structure of the environment. The optical flow information is, therefore, very useful to guide a mobile robot through the environment. In fact, in a translation motion through a static environment, the radial pattern of flows depends only on the camera's heading and is independent of the 3D structure of the environment, while the magnitude of each flow vector depends on both heading and depth and decrease quickly with distance[1]. Hence, the information provided by optical flow can be used to perceive self-motion and estimate the distance to obstacles in the environment.

Our proposed method for obstacle avoidance by using even-based optical flow has two steps. The first step is to estimate focus of expansion (FOE) which contains self-motion information, and then in the second step, the time to reach an object, time-to contact[2](TTC), is estimated.

[1]In a dynamic environment, the relative motion between camera and the scene defines the direction of flows

[2]In the literature, it is also called time-to-impact or time-to-crash

The FOE represents the intersection of the camera translational velocity vector and the image plane, and plays a vital role in the extraction of information from the optical flow. In frame-based paradigm, various methods have been used to estimate the position of the FOE on the image plane and then TTC. The first group known as feature based methods use corresponding features such as points, lines, and curves between image frames to establish an estimation of the motion and the 3D structure of the scene[90, 91]. These methods use the local information from small regions, and consequently are error-prone. In addition, finding correspondence features is a difficult task. The second group known as motion field-based methods use optical flow information to approximate the motion and scene depth [92, 93, 94]. These methods usually solve a least square minimization problem based on optical flow vectors and consequently are computationally demanding. Finally, there are direct methods which are based on the brightness-change constraint equation [95, 96] and estimate position of the FOE by imposing more constraints such as *depth-is-positive* [97, 98, 99]. These methods work efficiently only for pure translational movements, but not when the movement involves rotational components.

An alternative approach to estimate TTC is based on the computation of the first order derivatives of optical flow field, i.e. the curl, divergence, and deformation components of the optical flow field. These methods usually use the expected form of the flow field (e.g. by applying known filter to the optical flow field) to estimate the TTC [100, 101, 102, 103, 104, 105, 106].

Besides, research in biology and neoroscience has shown the importance of motion cues in perception of the environment and controlling the behavior in different animal species and also humans [107, 108, 109]. For an extensive review on how optical flow are extracted by visual cortex and how this information is used to perceive self-motion and control behavior see [110].

In the rest of this chapter, first our camera model and the velocity field for a rigid body motion is reviewed in section 5.2. Then, our proposed algorithms for locating FOE and then estimating TTC by exploiting the event-based optical flow algorithm are described in section 5.3 and 5.4, respectively. The experimental results are presented in section 5.5, and finally a summary of this chapter is given in section 5.6.

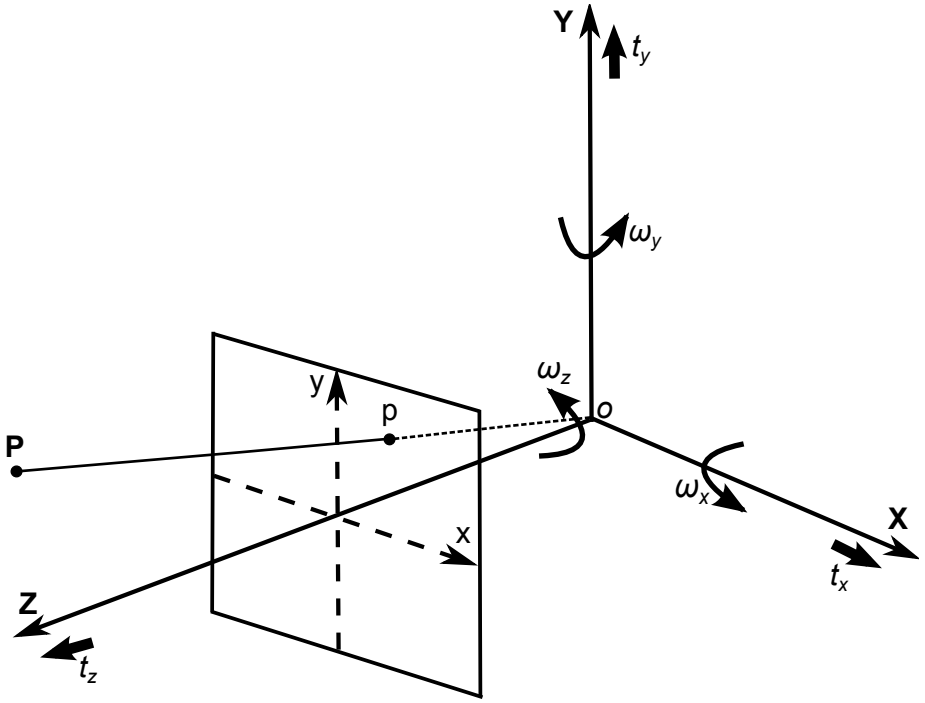


Figure 5.1: Camera coordinate system

5.2 The velocity field

Consider a camera-centered coordinate system where the z-axis aligns with the line of sight as shown in Fig. 5.1. If the camera moves with translational velocity $t = (t_x, t_y, t_x)$ and rotational velocity $\omega = (\omega_x, \omega_y, \omega_z)$ around its origin, then, the 3D velocity of a world point, $P = (X, Y, Z)$, is:

$$\dot{P} = -t - \omega \times P, \tag{5.1}$$

or, in components,

$$\dot{P} = \frac{dP}{dt} = \begin{pmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{pmatrix} = - \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} - \begin{pmatrix} \omega_y z - \omega_z y \\ \omega_z x - \omega_x z \\ \omega_x y - \omega_y x \end{pmatrix}. \tag{5.2}$$

Assume the world point $P = (X, Y, Z)$ is projected onto the point $p = (x, y)$ on the image plane. Using Pinhole camera model, points P and p are related via

perspective projection, so

$$x = f\frac{X}{Z}, \qquad y = f\frac{Y}{Z} \tag{5.3}$$

where f is the focal length of the camera [1]. Temporal differentiation of image coordinates results in the 2D motion field i.e. the velocity field induced in the image plane:

$$\dot{x} = \frac{\dot{X}Z - X\dot{Z}}{Z^2}, \qquad \dot{y} = \frac{\dot{Y}Z - Y\dot{Z}}{Z^2}. \tag{5.4}$$

By substituting $\dot{X}$ and $\dot{Y}$ from Eq. (5.2) into Eq. (5.4), the 2D motion filed, $\dot{p}$, can be written as follows:

$$\dot{p} = \frac{dp}{dt} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \frac{1}{Z}\begin{pmatrix} xt_z - t_x \\ yt_z - t_y \end{pmatrix} + \begin{pmatrix} xy & -(x^2+1) & y \\ (y^2+1) & -xy & -x \end{pmatrix}\begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}. \tag{5.5}$$

For a pure translational motion, i.e. when $\omega = (0, 0, 0)$, from Eq. (5.5), the velocity field at each point p can be written as

$$\dot{p} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \frac{t_z}{Z}\begin{pmatrix} x - \frac{t_x}{t_z} \\ y - \frac{t_y}{t_z} \end{pmatrix}. \tag{5.6}$$

As can be seen from Eq. (5.6), at location $(t_x/t_z, t_y/t_z)^T$ the velocity field is zero. This point indicates the position of the FOE. In addition, the variable Z/t_z is the time it takes for an object moving at constant velocity t_z to cross the distance Z, this represents TTC. Let τ and $(x_{foe}, y_{foe})^T$ denote TTC to world point P and location of FOE on the image plane, respectively. Equation (5.6) can be rewritten as:

$$\dot{p} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \frac{1}{\tau}\begin{pmatrix} x - x_{foe} \\ y - y_{foe} \end{pmatrix}, \tag{5.7}$$

[1] For sake of simplicity, we assume $f = 1$.

then,

$$\tau = \frac{d}{\|\dot{p}\|} \tag{5.8}$$

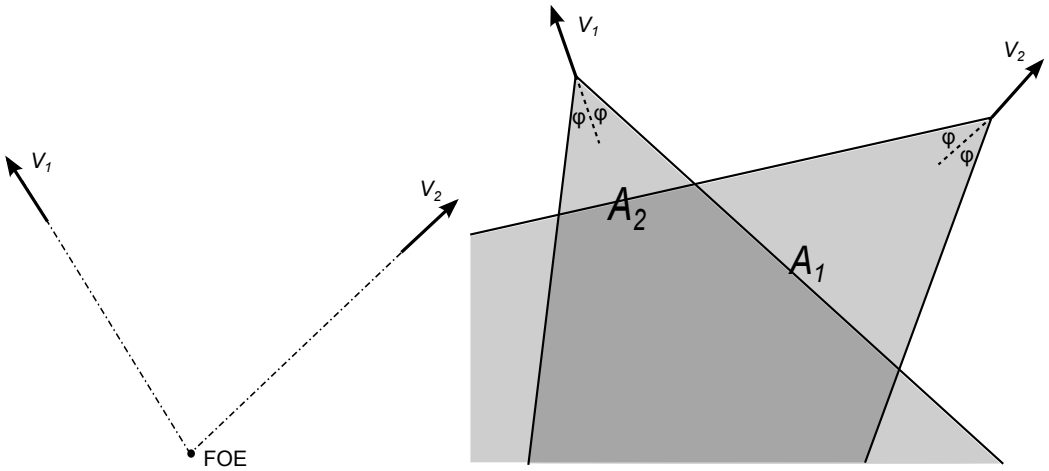
where d is the distance of point p to FOE on the image plane and $\|\dot{p}\|$ is the magnitude of velocity vector at point p.

The optical flow algorithm, presented in chapter 4, provides an estimation of the velocity filed that can be used to compute $\|\dot{p}\|$. In the following, we describe the algorithm proposed to compute FOE.

5.3 Locating the focus of expansion

As mentioned above, FOE is the projection of translation motion of the camera on the image plane, and in a pure translational movement, all the flow vectors diverge from the FOE. Therefore, in principle, FOE could be obtained by triangulation of two vectors in a radial flow pattern, as shown in Fig. 5.2(a). Obviously, such a method would be vulnerable due to errors in optical flow computation. However, since each computed flow provides an estimation of the FOE location, pooling all of them allows to reduce the triangulation error. Considering the possible errors in computation of flows, each flow indicates that FOE is located in an area in the opposite direction of the flow and with a angle of ϕ in either directions as shown in Fig. 5.2(b). In this figure, the two flow vectors V_1 and V_2 indicate that FOE is located in area A_1 and A_2, respectively. Consequently, the FOE would be located in the intersection of A_1 and A_2 areas. Considering more flow vectors from different part of the image will reduce the search space and lead to an estimation of the FOE position.

More formally, to estimate FOE position an image called *FOE-image* is built. *FOE-image* has the same size of the image plane, and each pixel (x, y) of the *FOE-image* represents the probability of the FOE being located on the corresponding pixel (x, y) of the image plane. The values of *FOE-image* pixels are updated according to the computed flow vectors using a leaky integration model. By exploiting the redundancy in the flow vectors and integrating many flow vectors, the patch of the *FOE-image* with the maximum value represents the FOE



(a) (b)

Figure 5.2: (a) In principle, FOE can be calculated by triangulation of two vectors in the velocity field (b) Each optical flow vector indicates that FOE is located in a certain area of the image plane

position.

In an event-based formulation, each velocity vector is represented by an ordered 5-tuple $v(x, y, t, v_x, v_y)$: the first three elements, x, y and t, represents the event $e(x, y, t)$ generated by the sensor, and $(v_x, v_y)^T$ is the velocity vector computed upon arrival of this event. The *FOE-image* map is updated in short time intervals of Δt in two steps as follows:

1. the values are gradually decreased over time as follows:

$$\begin{aligned} \textit{FOE-image}(n, m) =& \textit{FOE-image}(n, m) \\ & * e(-\lambda \Delta T_{(n,m)}) \quad \forall (n, m) \in \textit{FOE-image}, \end{aligned} \tag{5.9}$$

   where λ is the leak rate and ΔT is calculated according to the last update time of each pixel.

2. then, for each velocity event v_i arrived in Δt, the corresponding set of pixels, A_i (see Fig. 5.2(b)) are indicated and the value of these pixels in the *FOE-image* are increased, i.e. for each $v_i(x_i, y_i, t_i, v_{x_i}, v_{y_i})$ the A_i is indicated by

using (v_{x_i}, v_{y_i}) and then

$$FOE\text{-}image(n,m) = FOE\text{-}image(n,m) + c, \quad \forall (n,m) \in A_i, \tag{5.10}$$

where c is a constant value.

Then, after each update of *FOE-image*, the patch of *FOE-image* with the maximum value is found. Let $(x_m, y_m)^T$ denotes the center of the patch with maximum value on *FOE-image*. Finally, the new position of the FOE is computed by applying a low-pass filter as follows:

$$\begin{pmatrix} x_{foe} \\ y_{foe} \end{pmatrix} = \begin{pmatrix} x_{foe} \\ y_{foe} \end{pmatrix} + \alpha \left(\begin{pmatrix} x_m \\ y_m \end{pmatrix} - \begin{pmatrix} x_{foe} \\ y_{foe} \end{pmatrix} \right), \tag{5.11}$$

where α is a constant.

Algorithm 4 describes the computation of FOE. A snapshot of *FOE-image* taken form our algorithm running on a sequence is shown in Figure 5.3. The X and Y axes indicate the position of pixels on the image plane, and the *FOE probability* represents the *FOE-image*. This image shows that the search space for the FOE is reduced by integrating enough flow vectors form different part of the image, and then, with high probability, the patch with the maximum value represents the position of the FOE.

Algorithm 4 Focus of Expansion

For all the velocity events arrived in Δt

1. Update the *FOE-image* using Leaky integration model according to Eq. (5.9) and (5.10)
2. Locate (x_m, y_m), the patch of visual field with maximum value
3. Shift the FOE position toward the maximum patch using Eq. (5.11)

5.4 Estimation of time-to-contact

As discussed in section 5.2, in a pure translational motion, knowing the position of FOE and the flow vectors, it is possible to estimate TTC by using Eq. (5.8) at

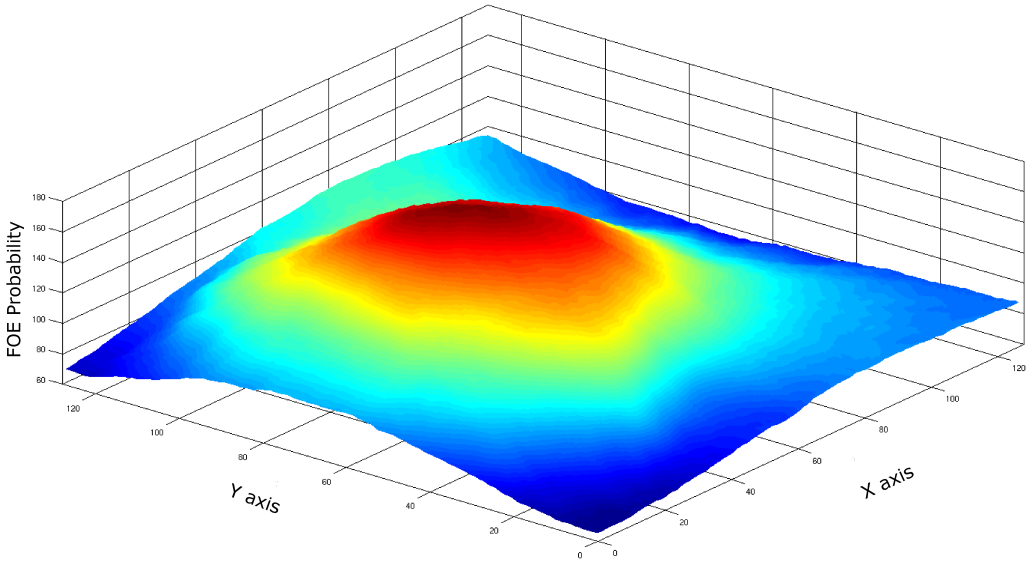


Figure 5.3: A snapshot of the *FOE-image*

each pixel.

To compute TTC, an image called *object-map* with the same size of the image plane is built. Each pixel (x, y) of the *object-map* represents the TTC to the corresponding pixel (x, y) of the image plane. In the event-based formulation, the *object-map* is updated in time intervals of Δt similar to the *FOE-image*. To update the *object-map*, for each velocity vector v_i that arrives in the time interval, the TTC at pixel (x_i, y_i) is computed using Eq. (5.8). Then, the pixels which has not updated for a long time are set to unknown. Finally, a spatial low-pass filter is applied to the *object-map* image.

Algorithm 5 summarize the computation of TTC.

Algorithm 5 Time-to-Contact

For all the velocity events arrived in Δt

1. Update the *object-map* with new arrived events using Eq. (5.8)
2. Reset the old values
3. Apply an average filter over $n * n$ neighborhoods

(a) (b)

Figure 5.4: (a) Experiment Setup (b) The experiment environment

5.5 Experiments and results

To test the proposed algorithms for computation of the FOE and TTC using event-based visual sensor (algorithm 4 and algorithm 5), we have used a set up presented in Fig. 5.4(a), where the DVS128 vision sensor was installed on the iKart platform. In this experiment, the iKart was moved in a lab-like environment towards an object with a chessboard-like front as shown in Fig. 5.4(b). The results of running our algorithms on the captured sequence are presented in Fig. 5.5(a)-5.5(e). The color map is shown is Fig. 5.5(f). The red color represents the smallest value for TTC. Figure 5.5(a)-5.5(e) show snapshot of the *object map* as the camera is moving towards the chessboard and getting closer to it, the red circle also represents FOE. It can be seen in Fig. 5.5(a)-5.5(e), that the general trend of TTC is decreasing.

Figure 5.5(a) shows objects in different dept and the computed TTC is consistent with the relative depth of the object: the chair in the left side of the image is the closest object and is red, the chair in the right side of the image is further and is green, and the chessboard which is the furthest object is blue. On the other hand, as can be seen in Fig. 5.5(c) and Fig. 5.5(d), the computed TTC for different parts of the chessboard are sometimes different. Actually, the points which are further form the FOE seems to have a smaller TTC. The reason is that

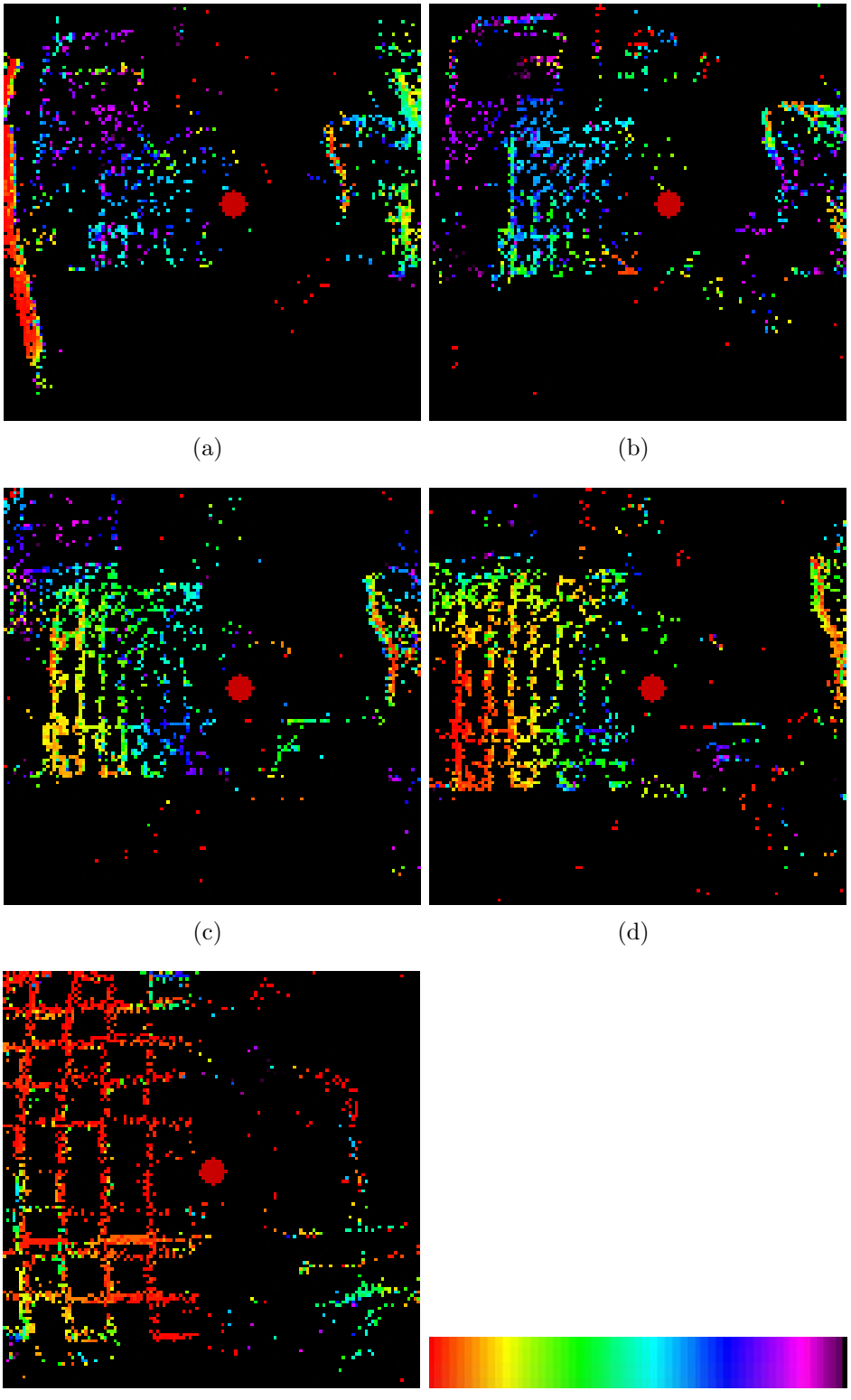
the distance to FOE is reduced while the magnitudes of optical flow for those parts have not decreased with the same ratio. Any improvement in accuracy of magnitude of optical flow will lead to more accurate estimation of TTC. In addition, other visual cues especially knowing that a set of events belong to the same object can improve TTC estimation.

Figure 5.6 represents the TTC estimation as the camera moves closer to the chessboard object. The values presented in this plot are computed by taking the average of the estimated TTC in a fixed patch of visual field every 10 ms. From the beginning to the end of the sequence, chessboard is the only object projected on this patch of visual field. The plot shows a decreasing trend which is consistent with the theory and the experiment set up.

Computation of FOE in outdoor environment

In this part, we present the output of algorithm 4 for locating FOE on a sequence captured form DVS128 vision sensor installed on a car, while the car was driving in the city in the normal day light condition. The advantage of using this sequence is that we can see how the algorithm behaves in real outdoor situations. Figure 5.7 presents snapshots of the sequence and the estimation of FOE by algorithm 4 when the car was driving in a street with trees in both sides of the street, and the DVS128 sensor, therefore, provided enough number of events to compute FOE. In Figure 5.7, left Images are reconstruction of scene (by accumulating received events), and right images show the output of FOE algorithm. The green and red circles represent the patch of *FOE-image* with maximum value and estimated location of FOE, respectively. Figure 5.8 presents the optical flow for the corresponding scenes. Figure 5.7(h) shows that the green and red circles have converged to the same location. Actually, in this part of the sequence, the green and red circles converge to the same location after a short time, and stay at the same position as long as the car is driving in the same direction. In this part of the sequence, there are objects in different part of the scene and they cause many optical flows in different part of the visual field, consequently the algorithm 4 can provide a stable and accurate estimation of the FOE position.

Figure 5.9 shows the computed FOE and optical flow when car is turning to



(a) (b) (c) (d) (e) (f)

Figure 5.5: (a)-(e) Snapshots of the *Object-map* in a translational movement towards a chessboard object (d) the color bar: red indicates smaller value of TTC

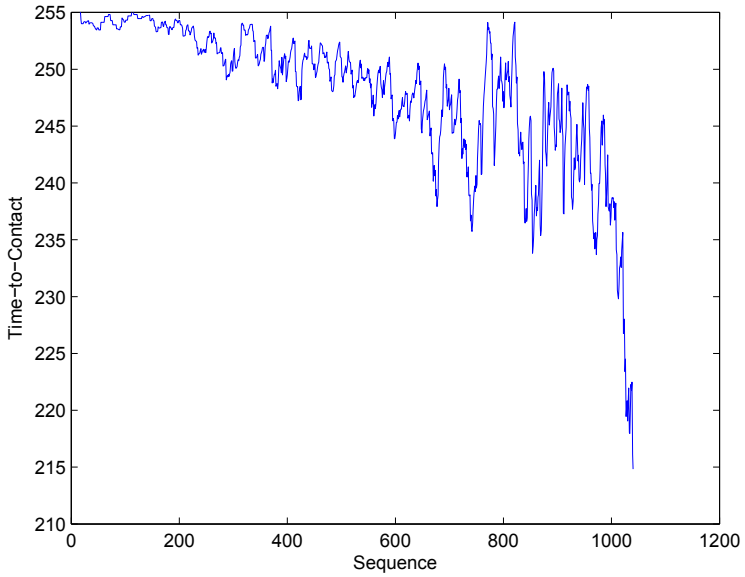
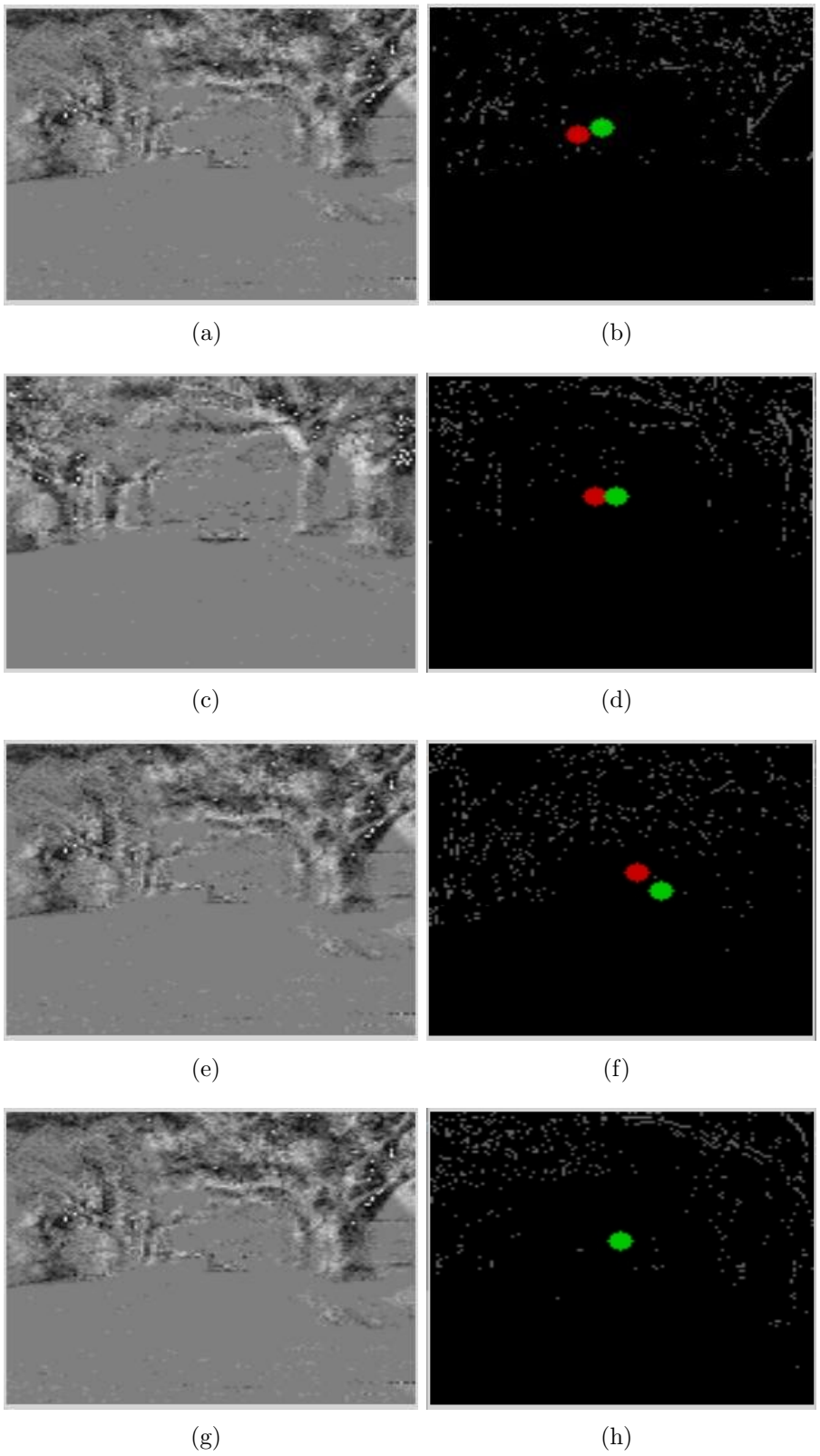


Figure 5.6: Computed TTC to the chessboard object as the camera gets closer to the object. The TTC values on are relative and do not represent real values

the left. The FOE is not defined for a rotational movement. However, algorithm 4 estimates that the FOE is located in the left side of visual field, since most flows are directed to the right. In other words, the output of algorithm 4 is not valid when the movement involves rotational components.

5.6 Summary

This chapter presents the estimation of FOE and TTC for a moving event-based vision sensor by exploiting the event-based optical flow algorithm presented in chapter 4. The FOE is computed by pooling the information provided by all the flow vectors computed in fixed time intervals. Our experimental results show that the estimation of FOE can be pretty accurate if enough number of events are provided by the vision sensor. The FOE estimation can be used to estimate the relative distances in the world and, hence, to reconstruct the 3D structure of the world. In addition, as shown in this chapter, FOE can be used to estimate TTC and, hence, to control movements especially avoiding obstacles.



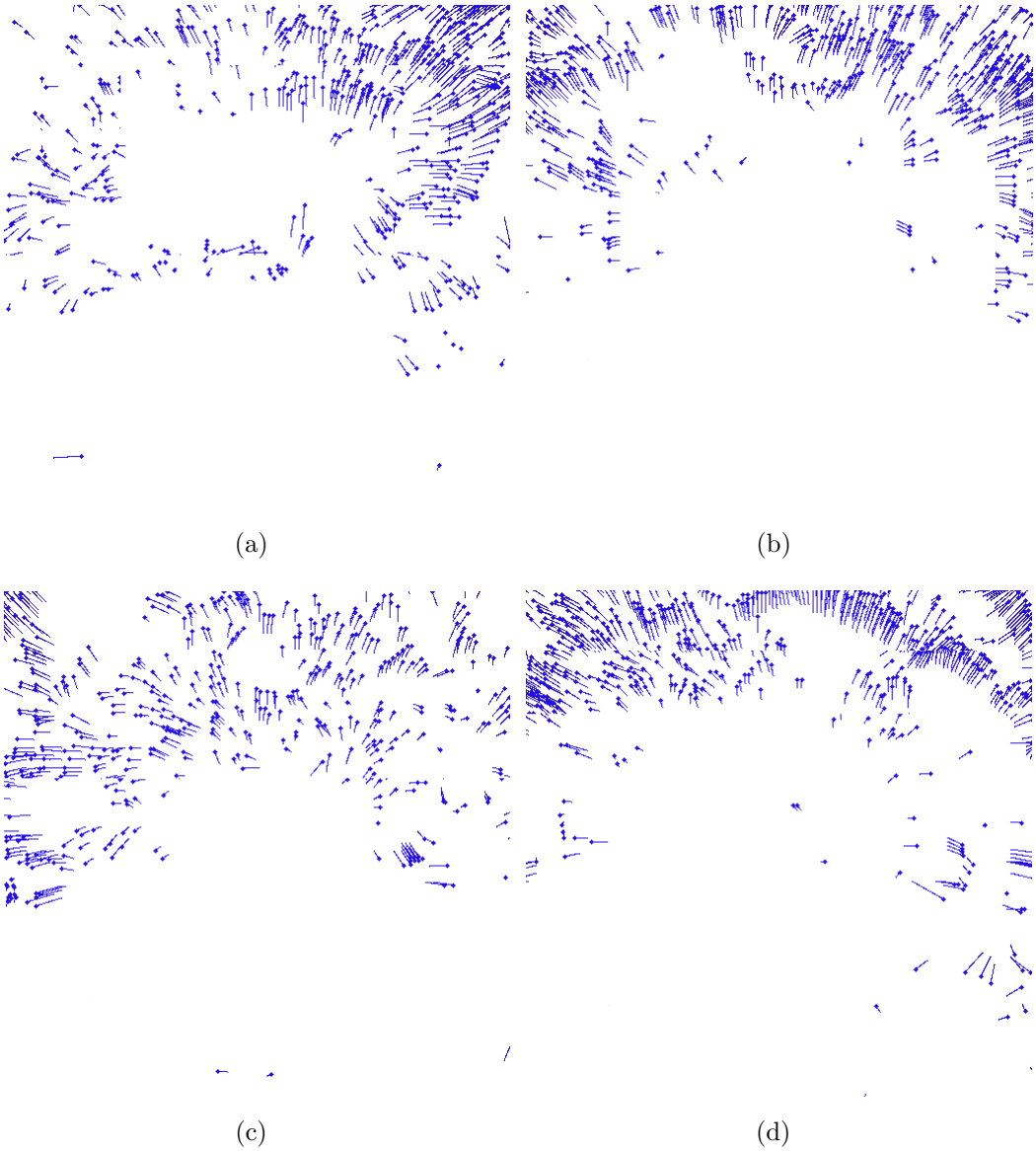
(a) (b)

(c) (d)

(e) (f)

(g) (h)

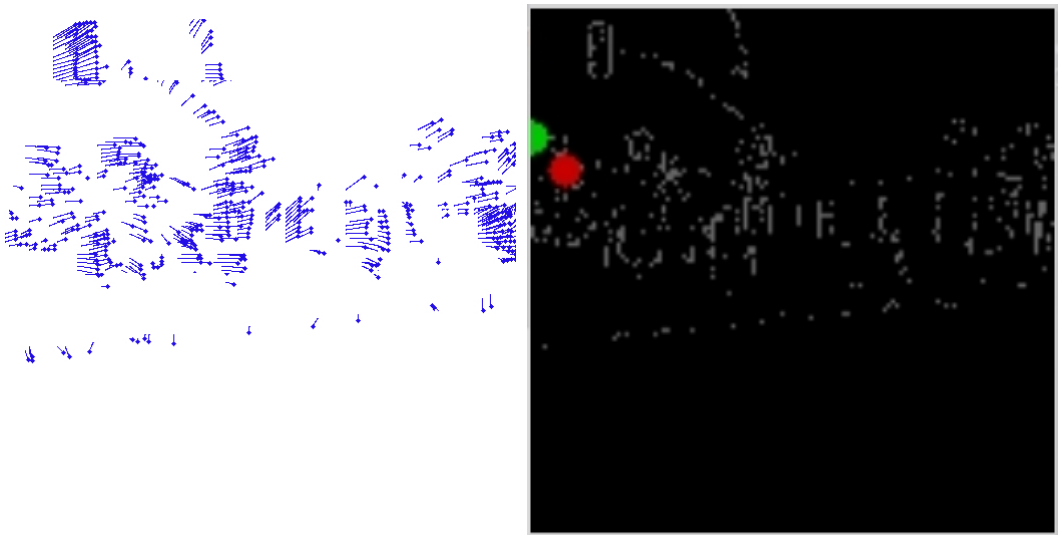
Figure 5.7: Driving car sequence: street snapshots and FOE estimation



(a) (b)

(c) (d)

Figure 5.8: Driving car sequence: computed optical flow



(a) (b)

Figure 5.9: Car turning left: FOE and optical flow

The proposed algorithm to compute the FOE only exploits the direction of computed flows, while to compute TTC the magnitudes of flows are also required. Therefore, computation of FOE is still accurate even if the optical flow algorithms does not provide a very accurate estimation of the flow magnitudes. The results presented in this chapter show that our algorithms work in real world situations. However, the computation of TTC is vulnerable to the errors of optical flow magnitudes. Consequently, more accurate optical flow algorithms results in a more accurate estimation of TTC. Another approach to improve estimation of TTC is to exploit other visual cues such as event clustering information.

Chapter 6

Conclusions and Future Work

6.1 Overview

Mobile robots and humanoids represent an emerging and challenging example of embedded computing applications. On one hand, in order to achieve a large degree of autonomy and intelligent behavior, these systems require a very significant computational capability to perform a wide variety of complex tasks, some of them with real-time constraints. On the other hand, they are severely limited in terms of size, weight, and particularly power consumption of their embedded computing system since they should carry their own power supply. Moreover, since these systems need to implement a wide variety of applications, their computing systems should provide programmability and adaptability of a general purpose platform. Programmability allows a single platform to support multiple applications, while adaptability shows the capability of the architecture to maintain efficiency even if the core computational characteristics of the applications change. Hence, their computing system should address three issues: computational efficiency[1], programmability, and adaptability [111].

This thesis has followed two approaches to provide low-power, lightweight, high performance computing architectures for mobile robots and humanoids:

- exploiting new emerging parallel architectures which provide both high computational capability and low-power consumption, and

[1] Computational efficiency represents the computation power in terms of power budget, i.e. performance per watt

- extracting and processing only important data by using emerging bio-inspired sensors such as DVS vision sensor.

Considering the three parameters mentioned above, computational efficiency, programmability, and adaptability, we proposed a low-power high performance vision architecture for mobile robots and humanoids which includes CSX SIMD and Tilera MIMD architectures. The estimated power consumption of such architecture is about 50 watt, while the peck performance is over 96 GFLOPs and 144 GOPs.

Toward the objective of this thesis, several parallel and event-based algorithms has been developed which can be classified as follows:

- parallelization of low-level image processing task on the CSX SIMD architecture
- parallelization of middle and high-level image processing tasks on the CSX SIMD architecture
- parallelization of event-based vision applications on the Tilera MIMD architecture
- even-based vision processing for mobile robots navigation

Currently there is a lot emphasis on the use of FPGAs and GPGPUs to achieve higher performance for image processing applications. In this research, various image processing algorithms with different computational characteristics have been implemented on our proposed parallel architecture. The results presented in chapter 2 and 3 indicates that the proposed vision architecture provides higher level of programmability, adaptability, and computational efficiency comparing with both FPGAs and GPGPUs. Although compared with GPGPU, our proposed architecture may provide lower computational performance for some application such as HOG-based human detection, but it provides a significantly better relative performance per watt.

The results presented in this thesis can indeed further motivate the investigation and application of emerging highly parallel, low power, SIMD and MIMD architectures for mobile robots and humanoids.

Moreover, the studies undertaken and the proposed solutions for parallel formulation and implementation of low, intermediate, and high-level image processing tasks on the CSX architecture (presented in chapter 2 and 3) along with parallel implementation of event-based optical flow algorithm on the Tilera architecture (chapter 4) can be exploited for implementation of other image processing tasks with similar computational characteristics.

6.2 Conclusions

Low-level image processing on CSX SIMD architecture

Parallel computation of several dense stereo vision algorithms and Harris corner detector (HCD) algorithm, as representative examples of low-level image processing tasks, on the CSX SIMD architecture was studied. It was shown that the row-cyclic data decomposition scheme is the most efficient data decomposition scheme for data parallel computation of low-level image processing algorithms such as the SSD-based stereo vision and its variants and HCD. It was also shown that, by devising a careful strategy, it is possible to significantly reduce the memory communication overhead by almost fully overlapping computation and communication. In addition to an efficient parallelization, exploitation of the vector processing capability of the PEs was a key for achieving a better performance.

The comparison of our results with the similar works in the literature demonstrated that our implementation provides a much better computational efficiency than GPGPUs and Cell processor. ASICs and FPGAs implementations can provide good absolute computational performance and computational efficiency for low-level image processing tasks. However, CSX architecture provides a much higher degree of programmability and adaptability compared to ASICs and FPGAs. The experimental results, presented in this chapter, clearly indicate that the CSX architecture is indeed a good candidate for achieving low-power high performance capability for low-level image processing tasks.

Intermediate and high-level image processing on CSX SIMD architecture

Parallel computation of HOG-based human detection, as an representative example of intermediate and high-level image processing tasks, on the CSX SIMD architecture was studied. For parallel computation of HOG-based object detection, the main challenges were on one hand, complex data dependency pattern, varying granularity (pixel, cell, block, and detection window), and especially the need for multi-scale computation and on the other hand small size of PEs' memories in the CSX architecture. A detailed analysis of multi-level computational structure of the HOG descriptors showed that the choice of block level grain size for parallel implementation is the most efficient in terms of reducing the redundancy in the computation and communication. A consequence of our parallelization strategy is that regular operations are performed for all blocks by all PEs. Taking advantage of this regularity, it was shown that efficient multi-scale computation on a fixed number of PEs can be mapped as the solution of a 2D *strip packing problem*. Other intermediate or high level image processing tasks especially object detection algorithms share similarities with HOG-based object detection in terms of computation needs. The solution proposed in this chapter can be exploited for parallel formulation and implementation of those tasks.

The parallel implementation of HOG-based human detection on GPGPU has been studied [72, 73]. Compared with GPGPU, we achieved a slightly lower computational performance but a significantly better relative computational efficiency (performance per watt). Our computational performance can be further increased by deploying multiple CSX architectures. As discussed in chapter 2, the performance increases almost linearly by using more CSX cores.

Asynchronous event-based optical flow: GPP and massively parallel implementations

Parallel formulation of an event-based optical flow algorithm [76] as an example of event-based vision processing algorithms was studied. The computation was

modeled in a pipeline (producer-consumer) manner, where the producers and consumers communicate through single-producer single consumer circular queues. The pipeline has four stages: receiving events, distributing events, computing velocities, and sending the results back. Further decomposition of the pipeline stages was discussed and it was shown that image-plane decomposition is the most suitable scheme to exploit parallelism in velocity computing stage. This scheme on one hand reduces inter-process communication overhead and on the other hand saves computation and memory access time by exploiting temporal and spatial locality of events.

The analyses undertaken for parallel formulation of event-based optical flow demonstrates that parallelization of event-based vision applications may be more complicated than conventional image processing tasks due to two reasons. First, commonly used data decomposition scheme can lead to unbalance computation, and consequently inefficient parallelization. Second, due to the sparse nature of data, it is more difficult to exploit the cache subsystem of computing platforms effectively, and hence the memory communication overhead may increase.

Along with parallel formulation and implementation of event-based optical flow on the Tilera MIMD architecture, a GPP implementation of even-based optical flow was provided for the mobile iKart platform and iCub robot. Although the GPP implementation compared with the parallel implementation is more limited in terms of computed event per second, it can provide satisfactory performance for some real-time applications, and it he been successfully used in a real-time obstacle avoidance system.

Event-based algorithms for obstacle avoidance

Event-based algorithms to estimate FOE and TTC for a moving event-based vision sensor were proposed. These algorithms exploits the information provided by the event-based optical flow algorithm presented in chapter 4. The proposed algorithm to compute the FOE only exploits the direction of computed flows, while to compute TTC the magnitudes of flows are also required. Therefore, computation of FOE is still accurate even if the optical flow algorithms does not provide an accurate estimation of the flow magnitudes. The results presented in

this chapter show that our algorithms work in real world situations. However, the computation of TTC is vulnerable to the errors of optical flow magnitudes.

6.3 Future work

Parallel implementation of intermediate and high level image processing algorithms on the Tilera architecture

Chapter 3 studied parallel implementation of HOG-based human detection as an example of intermediate and high level image processing tasks on the CSX architecture. Although, the computation is complex for an SIMD implementation, we have shown that by choosing the right size of granularity and regularizing the operation, it is possible to achieve high performance on the CSX SIMD architecture. One of the main bottleneck in computation of HOG-based human detection on CSX architecture was the memory communication overhead in the SVM evaluation step.

Tilera MIMD architecture provides higher adaptability than CSX SIMD architecture. This allows a wider choice for parallel formulation and implementation of HOG-based human detection and other similar image processing tasks on the Tilera MIMD architecture. A future work should consider parallel implementation of HOG-based human detection on the Tilera MIMD architecture. Such study would be necessary to understand which architecture is more suitable for low-power high performance implementation of intermediate and high level image processing tasks widely used in robotics such as object detection.

Studies on parallel formulation of other event-based algorithms

The parallel implementation of event-based optical flow algorithm on Tilera MIMD architecture was presented in chapter 4. Since the event-based DVS vision sensor provides a sparse and asynchronous stream of events in output, it might

seem that event based decomposition is the best strategy to exploit parallelism in event-based vision algorithms. However, as discussed in chapter 4, such techniques can increase memory communication overhead significantly since it cannot benefit from spatial and temporal locality assumption used in cache subsystem of computing platforms. On the other hand, as also discussed in chapter 4, an image-plane decomposition scheme (the commonly used techniques for distributing images in conventional parallel image processing) can lead to unbalance computation. The solutions proposed for parallel formulation of event-based optical flow can be exploited for parallel implementation of other event-based algorithms which are similar to the optical flow algorithm in terms of computational characteristics. However, future work on parallel implementation of other event-based algorithms (e.g. the FOE algorithm presented in this thesis, or the algorithms proposed in [112, 113]) would be advantageous to shed more light on exploiting parallelism for event-based vision processing tasks.

Integrating DVS and conventional frame-based cameras for mobile robot tasks

Chapter 4 and 5 present the research work undertaken to use DVS vision sensor for optical flow based navigation task to reduce computational cost. While the DVS vision sensor is beneficial in some aspects, it has its own limitations. One important limitation is the rather limited resolution. The other limitation is that it does not provide any color information. As mentioned in chapter 5, using other visual cues could improve our estimation of TTC. A future work would be to use conventional frame-based cameras to provide higher level information about the scene and integrate this information with the event-based processing to achieve more accurate algorithms.

References

[1] ClearSpeed Technology, "ClearSpeed Whitepaper: CSX Processor Architecture," tech. rep., 2007. viii, 10, 13, 22, 23, 24, 29

[2] Tilera Corporation., "Tile Processor User Architecture Manual (Doc. No. UG101)," tech. rep., 2011. viii, 77, 78, 79, 80

[3] Y. Mei, Y.-H. Lu, Y. C. Hu, and C. Lee, "A case study of mobile robot's energy consumption and conservation techniques," in *IEEE International Conference on Robotics and Automation (ICRA'05)*, pp. 492–497, IEEE, 2005. 2, 4

[4] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128 128 120dB 15us Latency Asynchronous Temporal Contrast Vision Sensor," *IEEE Journal of Solid State Circuits*, vol. 43, no. 2, pp. 566–576, 2008. 2, 14

[5] Mars Exploration Rover, "Mars Exploration Rover Mission," 2013. 2

[6] I. Ieropoulos, C. Melhuish, and J. Greenman, "Artificial metabolism: towards true energetic autonomy in artificial life," *Advances in Artificial Life*, pp. 792–799, 2003. 2, 3

[7] I. Ieropoulos, C. Melhuish, J. Greenman, and I. Horsfiel, "EcoBot-II: An artificial agent with a natural metabolism," *International Journal of Advanced Robotic Systems*, pp. 295–300, Dec. 2005. 2, 3

[8] I. Ieropoulos, J. Greenman, C. Melhuish, and I. Horsfield, "EcoBot-III: a robot with guts," in *Fellermann, H., Dörr, M., Hanczyc, M., Laursen,*

L., Maurer, S., Merkle, D., Monnard, P., Stoy, K. and Rasmussen, S., eds. Artificial Life XII, pp. 733–740, Massachusetts Institute of Technology Press, 2010. 3

[9] W. G. Walter, "An imitation of life," *Scientific American*, vol. 182, no. 5, pp. 42–45, 1950. 3

[10] Y. Hada and S. Yuta, "Robust navigation and battery re-charging system for long term activity of autonomous mobile robot," *Proceedings of the 9th International Conference on Advanced Robotics*, pp. 297–302, 1999. 3

[11] M. C. Silverman, D. Nies, B. Jung, and G. S. Sukhatme, "Staying alive: A docking station for autonomous robot recharging," in *IEEE International Conference on Robotics and Automation (ICRA'02)*, pp. 1050–1055, 2002. 3

[12] S. Oh, A. Zelinsky, and K. Taylor, "Autonomous battery recharging for indoor mobile robots," *Proceedings of the australian conference on Robotics and Automation*, 2000. 3

[13] IRobot, "iRobot Roomba Vacuum Cleaning Robot - Overview," 2012. 3

[14] Adept MobileRobots, "MobileRobots Charging Options," 2012. 3

[15] P. Zebrowski and R. T. Vaughan, "Recharging robot teams: A tanker approach," in *IEEE International Conference on Robotics and Automation (ICRA'05)*, pp. 803 – 810, 2005. 3

[16] T. D. Ngo, H. Raposo, and H. Schioler, "Potentially distributable energy: Towards energy autonomy in large population of mobile robots," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA'07)*, pp. 206–211, 2007. 3

[17] A. Barili, M. Ceresa, and C. Parisi, "Energy-saving motion control for an autonomous mobile robot," in *IEEE International Symposium on Industrial Electronics (ISIE'95)*, pp. 674–676 vol.2, 1995. 4

[18] F. Yamasaki, K. Hosoda, and M. Asada, "An energy consumption based control for humanoid walking," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'02)*, pp. 2473–2477, 2002. 4

[19] S. Michaud, A. Schneider, R. Bertrand, P. Lamon, R. Siegwart, M. Van Winnendael, and A. Schiele, "SOLERO: Solar powered exploration rover," *Proceedings of the 7th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, 2002. 4

[20] Z. Sun and J. Reif, "On energy-minimizing paths on terrains for a mobile robot," in *IEEE International Conference on Robotics and Automation (ICRA'03)*, pp. 3782–3788, 2003. 4

[21] T. Wang, B. Wang, and H. Wei, "Staying-alive and energy-efficient path planning for mobile robots," in *American Control Conference*, pp. 868–873, 2008. 4

[22] Y. Mei, Y.-H. Lu, Y. C. Hu, and C. Lee, "Energy-efficient motion planning for mobile robots," in *IEEE International Conference on Robotics and Automation (ICRA'04)*, pp. 4344–4349 Vol.5, IEEE, 2004. 4

[23] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi, "Power-aware scheduling under timing constraints for mission-critical embedded systems," in *Proceedings of the 38th annual conference on Design automation*, (New York, New York, USA), pp. 840–845, ACM Press, June 2001. 4

[24] "An Open Source Cognitive humanoid Robotic Platform," 2013. 5

[25] T. Delbruck, "Frame-free dynamic digital vision," *Proceedings of Intl. Symp. on Secure-Life Electronics, Advanced Electronics for Quality Life and Society*, pp. 21–26, 2008. 5, 88

[26] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-Scale Computing Research Overview," tech. rep., Research at Intel, White Paper, 2006. 7

[27] G. A. Bekey, *Autonomous Robots*. The MIT Press, 2005. 7

[28] B. Catanzaro, A. Fox, K. Keutzer, and D. Patterson, "Ubiquitous parallel computing from Berkeley, Illinois, and Stanford," *Micro, IEEE*, vol. 30, no. 2, pp. 41–55, 2010. 7

[29] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *Micro IEEE*, vol. 26, pp. 10–23, May 2006. 9

[30] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *Micro IEEE*, vol. 27, pp. 15–31, Sept. 2007. 10, 13

[31] J. Nickolls and W. J. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 30, pp. 56–69, Mar. 2010. 11

[32] NVIDIA Corporation, "NVIDIAs Next Generation CUDA Compute Architecture: Fermi," tech. rep., 2009. 11

[33] T. Gollisch and M. Meister, "Eye smarter than scientists believed: neural computations in circuits of the retina," *Neuron*, vol. 65, no. 2, pp. 150–164, 2010. 13

[34] P. Ruedi, P. Heim, F. Kaess, E. Grenet, F. Heitger, P. Burgi, S. Gyger, and P. Nussbaum, "A 128x128 pixel 120-dB dynamic-range vision-sensor chip for image contrast and orientation extraction," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 12, pp. 2325 – 2333, 2003. 14

[35] K. A. Zaghloul and K. Boahen, "Optic nerve signals in a neuromorphic chip II: Testing and results.," *IEEE transactions on bio-medical engineering*, vol. 51, pp. 667–675, Apr. 2004. 14

[36] U. Mallik, M. Clapp, E. Choi, G. Cauwenberghs, and R. Etienne-Cummings, "Temporal change threshold detection imager," in *IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*, pp. 362–363, IEEE, 2005. 14

[37] E. Culurciello and A. G. Andreou, "CMOS image sensors for sensor networks," *Analog Integrated Circuits and Signal Processing*, vol. 49, no. 1, pp. 39–51, 2006. 14

[38] "An Holonomic Mobile Platform for iCub Humanoid Robot," 2013. 16

[39] ClearSpeed Technology, "Advance e710 Board," 2013. 16

[40] Tilera Corporation., "TILExpress-20G Card," 2013. 17

[41] N. Kehtarnavaz and M. Gamadia, "Real-Time Image and Video Processing: From Research to Reality," *Synthesis Lectures on Image, Video, and Multimedia Processing*, vol. 2, pp. 1–108, Jan. 2006. 21

[42] ClearSpeed Technology, *CSX600 Hardware Programming Manual*. ClearSpeed, www.clearspeed.com, Jan. 2008. 22

[43] ClearSpeed Technology, *CSX600/CSX700 Instruction Set Reference Manual No. 06-RM-1137 Revision: 4.A*. ClearSpeed, www.clearspeed.com, Aug. 2008. 23

[44] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," *International Journal of Computer Vision*, vol. 47, no. 1-3, pp. 7–42, 2002. 25, 44

[45] W. Van Der Mark and D. M. Gavrila, "Real-time dense stereo for intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 7(1), pp. 38–50, 2006. 25, 26, 29, 44, 46, 47

[46] L. Di Stefano, M. Marchionni, and S. Mattoccia, "A PC-based real-time stereo vision system," *Journal of Machine Graphics & Vision*, vol. 13, no. 3, pp. 197–220, 2004. 25, 26

[47] B. McCullagh, "Real-Time Disparity Map Computation Using The Cell Broadband Engine," *Journal of Real-Time Image Processing*, vol. 7, pp. 87–93, Feb. 2012. 26, 46, 47

[48] R. Yang and M. Pollefeys, "A Versatile Stereo Implementation On Commodity Graphics Hardware," *Journal of Real-Time Imaging*, vol. 11, pp. 7–18, Feb. 2005. 26, 46, 47

[49] K. Zhu, M. Butenuth, and P. D'Angelo, "Comparison Of Dense Stereo Using CUDA," in *Workshop of Computer Vision on GPUs (CVGPU) in Conjucntion with ECCV*, 2010. 26, 46, 47

[50] N. Chang, T.-M. Lin, T.-H. Tsai, Y.-C. Tseng, and T.-S. Chang, "Real-Time DSP Implementation on Local Stereo Matching," in *IEEE International Conference on Multimedia and Expo*, pp. 2090–2093, IEEE, July 2007. 26, 47

[51] Y. Jia, X. Zhang, M. Li, and L. An, "A Miniature Stereo Vision Machine (MSVM-III) For Dense Disparity Mapping," in *Proceedings of the 17th International Conference on Pattern Recognition (ICPR'04)*, pp. 728–731 Vol.1, IEEE, 2004. 26

[52] C. Georgoulas and I. Andreadis, "A Real-Time Fuzzy Hardware Structure For Disparity Map Computation," *Journal of Real-Time Image Processing*, vol. 6, pp. 257–273, Mar. 2011. 26

[53] J. Woodfill, G. Gordon, and R. Buck, "Tyzx DeepSea High Speed Stereo Vision System," in *Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04)*, pp. 41–45, 2004. 26, 46, 47

[54] M. Kuhn, S. Moser, O. Isler, F. Gurkaynak, A. Burg, N. Felber, H. Kaeslin, and W. Fichtner, "Efficient ASIC Implementation Of A Real-Time Depth Mapping Stereo Vision System," in *IEEE 46th Midwest Symposium on Circuits and Systems*, vol. 3, pp. 1478–1481 Vol. 3, IEEE, 2003. 26

[55] K. Ambrosch, W. Kubinger, M. Humenberger, and A. Steininger, "Hardware Implementation Of An SAD Based Stereo Vision Algorithm," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'07)*, pp. 1–6, IEEE, June 2007. 27, 46, 47

[56] H. Hirschmüller, P. R. Innocent, and J. Garibaldi, "Real-time correlation-based stereo vision with reduced border errors," *International Journal of Computer Vision*, vol. 47(1), pp. 229–249, 2002. 28, 29

[57] C. Soviany, *Embedding data and task parallelism in image processing applications*. PhD thesis, Delft University of Technology, 2003. 30

[58] ClearSpeed Technology, *Visual Profiler. Document No.06-RM-1136 Revision:4.B.*, 2008. 40

[59] D. Scharstein and R. Szeliski, "Middlebury Stereo Vision Page," 2012. 44, 47

[60] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *ACM Computing Surveys*, vol. 38, no. 4, p. 13, 2006. 47

[61] P. M. Roth and M. Winter, "Survey of appearance-based methods for object recognition," Tech. Rep. ICG-TR-01/08, Inst. for Computer Graphics and Vision, Graz University of Technology, 2008. 47

[62] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *4th Alvey Vision Conference*, pp. 147–151, 1988. 47, 48

[63] L. Teixeira, W. Celes, and M. Gattass, "Accelerated Corner-Detector Algorithms," in *19th British Machine Vision Conference(BMVC '08)*, pp. 625–634, Springer-Verlag, 2008. 48, 54

[64] T. Saidani, L. Lacassagne, S. Bouaziz, and T. M. Khan, "Parallelization strategies for the points of interests algorithm on the cell processor," in *5th International symposium on Parallel and Distributed Processing and Applications (ISPA'07)*, pp. 104–112, 2007. 48

[65] B. Dietrich, "Design and Implementation of an FPGA-based Stereo Vision System for the {E}ye{B}ot {M6}." University of Western Australia, 2009. 48, 54

[66] C.-C. Cheng, C.-H. Lin, C.-T. Li, S. C. Chang, and L.-G. Chen, "{iVisual}: an intelligent visual sensor SoC with 2790fps CMOS image sensor and

205GOPS/W vision processor," in *45th annual Design Automation Conference(DAC '08)*, pp. 90–95, ACM, 2008. 48, 54

[67] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," in *Proc. International Conference on Computer Vision and Pattern Recognition ({CVPR}'05) - Volume 1*, pp. 886–893, June 2005. 56, 57, 58, 60

[68] D. G. Lowe, "Object Recognition from Local Scale-Invariant Features," in *Proceedings of the International Conference on Computer Vision (ICCV '99), Vol. 2*, pp. 1150–1157, IEEE Computer Society, 1999. 56

[69] Q. Zhu, M.-C. Yeh, K.-T. Cheng, and S. Avidan, "Fast Human Detection Using a Cascade of Histograms of Oriented Gradients," in *Proc. International Conference on Computer Vision and Pattern Recognition({CVPR}'06)-Volume 2*, pp. 1491–1498, June 2006. 57

[70] W. Zhang, G. Zelinsky, and D. Samaras, "Real-time Accurate Object Detection using Multiple Resolutions," in *Proc. IEEE International Conference on Computer Vision({ICCV}'07)*, pp. 1–8, Oct. 2007. 57

[71] T. P. Cao, G. Deng, and D. Mulligan, "Implementation of Real-time Pedestrian Detection on FPGA," in *23rd International Conference on Image and Vision Computing New Zealand({IVCNZ}'08)*, pp. 1–6, Nov. 2008. 57

[72] C. Wojek, G. Dorkó, A. Schulz, and B. Schiele, "Sliding-Windows for Rapid Object Class Localization: A Parallel Technique," in *Proc. of the 30th DAGM symposium on Pattern Recognition*, pp. 71–81, June 2008. 57, 72, 122

[73] V. Prisacariu and I. Reid, "fast HOG - a real-time GPU implementation of HOG," Tech. Rep. 2310/09, Department of Engineering Science, Oxford University, 2009. 57, 72, 122

[74] N. DALAL, *Finding People in Images and Videos*. PhD thesis, National Polytechnic Institute of Grenoble, July 2006. 58, 61

[75] N. Ntene and J. H. van Vuuren, "A Survey and Comparison of Guillotine Heuristics for the 2D Oriented Offline Strip Packing Problem," *Discrete Optimization*, pp. 174–188, 2009. 68, 69

[76] R. Benosman, S.-H. Ieng, C. Clercq, C. Bartolozzi, and M. Srinivasan, "Asynchronous Frameless Event-Based Optical Flow," *Neural Networks*, vol. 27, pp. 32–7, Mar. 2012. 74, 75, 76, 93, 102, 122

[77] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, pp. 43–77, Feb. 1994. 74

[78] D. Sun, S. Roth, and M. J. Black, "Secrets of optical flow estimation and their principles," in *Conference on Computer Vision and Pattern Recognition (CVPR'10)*, pp. 2432–2439, IEEE, June 2010. 74

[79] M. Fleury, A. Clark, and A. Downton, "Evaluating optical-flow algorithms on a parallel machine," *Image and Vision Computing*, vol. 19, pp. 131–143, Feb. 2001. 74

[80] B. K. Horn and B. G. Schunck, "Determining Optical Flow," *Artificial Intelligence*, vol. 17, pp. 185–203, Aug. 1981. 75

[81] B. Lucas and T. Kanade, "An Iterative Image Registration Technique With An Application to Stereo Vision," in *Proceedings of the 7th international joint conference on artificial intelligence (IJCAI'81)*, pp. 674–679, 1981. 75, 76

[82] H.-H. Nagel, "On Change Detection And Displacement Vector Estimation in Image Sequences," *Pattern Recognition Letters*, vol. 1, pp. 55–59, Oct. 1982. 75

[83] Tilera Corporation., "Programming The Tile Processor (Doc. No. UG205)," tech. rep., 2012. 79, 92

[84] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Addison-Wesley, 2003. 80, 86

[85] D. Ungar and S. S. Adams, "Harnessing Emergence For Manycore Programming," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, pp. 19–26, ACM Press, Oct. 2010. 83

[86] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, pp. 690–691, Sept. 1979. 83

[87] L. Lamport, "Concurrent Reading and Writing," *Communications of the ACM*, vol. 20, pp. 806–811, Nov. 1977. 83

[88] L. Higham and J. Kawash, "Critical sections and producer/consumer queues in weak memory systems," in *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, pp. 56–63, IEEE Comput. Soc, 1997. 83

[89] Tilera Corporation., "PCIE User Space Communication API (Doc. No. UG218)," tech. rep., 2011. 91

[90] R. Jain, "Direct computation of the focus of expansion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, pp. 58–64, Jan. 1983. 105

[91] L. Dron, "The multiscale veto model: a two-stage analog network for edge detection and image reconstruction," *International Journal of Computer Vision*, vol. 11, pp. 45–61, Aug. 1993. 105

[92] A. R. Bruss and B. K. Horn, "Passive navigation," *Computer Vision, Graphics, and Image Processing*, vol. 21, pp. 3–20, Jan. 1983. 105

[93] R. Sharma and Y. Aloimonos, "Early detection of independent motion from active control of normal image flow patterns.," *IEEE Transactions on Systems, Man, and Cybernetics. Part B*, vol. 26, pp. 42–52, Jan. 1996. 105

[94] D. Sazbon, H. Rotstein, and E. Rivlin, "Finding the focus of expansion and estimating range using optical flow images and a matched filter," *Machine Vision and Applications*, vol. 15, pp. 229–236, Oct. 2004. 105

[95] B. K. P. Horn and E. J. Weldon, "Direct methods for recovering motion," *International Journal of Computer Vision*, vol. 2, pp. 51–76, June 1988. 105

[96] S. Negahdaripour and B. K. P. Horn, "Direct passive navigation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, pp. 168–176, Jan. 1987. 105

[97] S. Negahdaripour and B. K. Horn, "A direct method for locating the focus of expansion," *Computer Vision, Graphics, and Image Processing*, vol. 46, pp. 303–326, June 1989. 105

[98] S. Negahdaripour and V. Ganesan, "Simple direct computation of the FOE with confidence measures," in *EEE Conference on Computer Vision and Pattern Recognition (CVPR'92)*, pp. 228–235, IEEE Comput. Soc. Press, 1992. 105

[99] I. S. McQuirk, B. K. Horn, H.-S. Lee, and J. L. Wyatt Jr., "Estimating the focus of expansion in analog VLSI," *International Journal of Computer Vision*, vol. 28, no. 3, pp. 261–277, 1998. 105

[100] R. Nelson and J. Aloimonos, "Obstacle avoidance using flow field divergence," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 10, pp. 1102–1106, 1989. 105

[101] N. Ancona and T. Poggio, "Optical flow from 1D correlation: Application to a simple time-to-crash detector," in *4th International Conference on Computer Vision*, pp. 209–214, IEEE Computer Society Press, 1993. 105

[102] F. Meyer, "Time-to-collision from first-order models of the motion field," *IEEE Transactions on Robotics and Automation*, vol. 10, no. 6, pp. 792–798, 1994. 105

[103] R. Cipolla and A. Blake, "Image divergence and deformation from closed curves," *International journal of Robotics Research*, vol. 16, pp. 77–96, 1997. 105

[104] D. Coombs, M. Herman, and M. Nashman, "Real-time obstacle avoidance using central flow divergence, and peripheral flow," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 49–59, 1998. 105

[105] T. Camus, D. Coombs, and M. Herman, "Real-time single-workstation obstacle avoidance using only wide-field flow divergence," in *International Conference on Pattern Recognition, vol. 3*, pp. 323–330, IEEE, 1996. 105

[106] S. Lakshmanan, N. Ramarathnam, and T. Yeo, "A side collision awareness method," in *IEEE 2002 Intelligent Vehicle Symposium*, vol. 2, pp. 640–645, IEEE, 2002. 105

[107] W. Schiff, J. Caviness, and J. Gibson, "Persistent fear responses in rhesus monkeys to the optical stimulus of looming," *Science*, vol. 136, pp. 982–983, 1962. 105

[108] D. Lee, "A theory of visual control of braking based on information about time-to-collision," *Perception*, vol. 5, no. 4, pp. 437–459, 1976. 105

[109] M. V. Srinivasan, S. W. Zhang, J. S. Chahl, E. Barth, and S. Venkatesh, "How honeybees make grazing landings on flat surfaces," *Biological Cybernetics*, vol. 83, pp. 171–183, Aug. 2000. 105

[110] W. Warren, "The state of flow," *High-level motion processing*, pp. 315–358, 1998. 105

[111] M. Woh, S. Mahlke, T. Mudge, and C. Chakrabarti, "Mobile supercomputers for the next-generation cell phone," *IEEE Computer*, vol. 43, pp. 81–85, Jan. 2010. 119

[112] J. A. Perez-Carrasco, B. Acha, C. Serrano, L. Camunas-Mesa, T. Serrano-Gotarredona, and B. Linares-Barranco, "Fast vision through frameless event-based sensing and convolutional processing: application to texture recognition.," *IEEE transactions on neural networks*, vol. 21, pp. 609–20, Apr. 2010. 125

[113] S. Schraml, A. N. Belbachir, N. Milosevic, and P. Schon, "Dynamic stereo vision system for real-time tracking," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 1409–1412, IEEE, May 2010. 125