A Multi-Core Framework for Non-Block Based Image Processing

B Ravi Kiran, Y Senthil Kumar and Anoop K P, Member IEEE.

1. Introduction

With increasing computational requirements for state-of-art computer vision and graphics algorithms, multi-core processor implementations have picked up momentum. Adapting single core image processing algorithms for multi-core environment is difficult. Different platforms require a hand tailored approach to optimally fit the algorithm into a given multi-core architecture. An automatic compiler which could split and compile the code for any platform is desirable. Such a compiler doesn’t exist and this kind of automation is tough to achieve. To aid existing compilers, we propose a simple software architecture framework for multi-core systems that aims to provide a uniform, scalable, and portable solution.

There have been several attempts to provide a multi-core framework for image processing algorithms. A few of these include pipelined decomposition tree (PDT) [Ko, 2006], data flow graphs, task graphs, etc. These frameworks were inherently designed to handle local (block based) image processing operations. Non-block based algorithms were not dealt efficiently within these frameworks. In this paper, we attempt to provide insights for solving a much wider class of algorithms on multi-core systems using a homogeneous framework.

2. Multi-core Frameworks

2.1 Software framework description

In general, a framework is a basic conceptual structure used to solve or address complex issues. In software design, framework is a reusable design with its associated implementations. The design is a model or an entity of an application which abstracts out the relevant characteristics. The associated implementations define how this model will be applied. Developing a good framework requires a deep understanding of the application domain and experience in development. In other words, the framework represents the software architecture and its implementation designed to capture all the common details and flexible for capturing the uncommon details.

A software framework has the following benefits. The main benefit obtained is design and code reuse. Also development time is saved as it reduces the effort of design and coding. Depending upon the flexibility and exhaustiveness of the framework, more reuse can be obtained. Portability and maintenance becomes easier as the key design and higher level implementations are abstracted out. To add, the applications developed on top of the framework will be less buggy due to the homogeneity of the coding and reuse. In industry perspective, a good framework gives the developer achieve better productivity and shorter time to market.

2.2 Multi-core Systems

A multi-core system is used obtain more performance by making use of the same dye area on the silicon to split multiple processors of lower clock rates, as opposed to a single processors of high clock rate on the same dye area. This is what is described in literature as the hitting the “clock
wall”. Any increment in clock rate would impact in significant power consumption. Thus attempts are being made to switch to a multi-core environment. In this environment, many architectures get developed with different feature sets. Integration of such heterogeneous systems is the current developer’s and system engineer’s challenge.

A multi-core framework is a requirement in this upcoming industry which is a split between optimal solutions obtained using symmetric multiprocessing (SMP) versus heterogeneous system on chip (SoC). SMP implementations conventionally involve an HLOS implementation that performs a variety of operations: inter-processor communication, binary code split amongst processors, split or load optimization, memory management and bookkeeping, developer abstraction layers – these are features that are put in to make the framework more generic and less user dependent. This is primary to drive implementations that parallelize any user code. Architectures can be classified in multiple ways to describe could be explained in terms of what is called the Flynn’s taxonomy. There is no exhaustive characterization of the different types of parallel systems. The most popular taxonomy was defined by Flynn in 1966. The classification is based on the notion of a stream of information. Two types of information flow into a processor: instructions and data. Conceptually these can be separated into two independent streams, whether or not the information actually arrives on a different set of wires. Flynn’s taxonomy classifies machines according to whether they have one stream or more than one stream of each type.

The four combinations are SISD (single instruction stream, single data stream), SIMD (single instruction stream, multiple data streams), MISD (multiple instruction streams, single data stream), and MIMD (multiple instruction streams, multiple data streams).

Similar features are required in a generic multi-core system where the differences occur in architecture, number of available processors, internal memory of each processor, memory hierarchy organization(distributed memory model, shared memory model,...) processing MHz(speed), IPC mechanisms (interconnect based, Shared memory communication etc) and some external factors like compiler specifics, multi-linker coordination. These are multitudes of settings that need to be hand tailored by the system engineer to extract the maximum out of a given heterogeneous multi-core array. This process becomes tedious when it should happen afresh for every new system.

### 2.2.1 Data level & task level parallelism

In a multiprocessor system executing a single set of instructions (SIMD), data parallelism is achieved when each processor performs the same task on different pieces of data. A single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication takes place usually to pass data from one thread to the next as part of a workflow. Data parallelism emphasizes the distributed (parallelized) nature of the data, while task parallelism emphasizes the distributed (parallelized) nature of the processes or functions. Most real programs fall somewhere on a continuum between task parallelism and data parallelism.
2.2.2 Pipeline Level Parallelism

Pipeline parallelism is when multiple steps depend on each other, but the execution can overlap and the output of one step is streamed as input to the next step. The pipeline can be extended to include any number of steps and can even extend between different physical machines. Pipeline parallelism is possible when Step B requires output from Step A, but it does not need all the output before it can begin.

A limitation of piping is that it supports single-pass, sequential data processing. Because piping stores data for reading and writing in ports (transport port can be TCP/IP for physically different machines or it can be Interconnect port abstractions) instead of disks, the data is never permanently stored. Instead, after the data is read from a port, the data is removed entirely from that port and the data cannot be read again. If your data requires multiple passes for processing, piping cannot be used. The benefits of piping should be weighed against the cost of potential CPU or I/O bottlenecks. If execution time for a procedure or statement is relatively short, piping is probably counterproductive.

2.2.3 Static and Dynamic Dataflow

Dataflow architecture is an alternate to Von Neumann/control flow based architecture. Data flow architectures do not have conventionally heard architecture details such as: program counter, the execution of the instructions is determined by the availability of the input arguments to the instruction. Synchronous dataflow architectures tune to match the workload presented by real-time data path applications such as wire speed packet forwarding. Dataflow architectures that are deterministic in nature enable programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources.

Designs that use conventional memory addresses as data dependency tags are called static dataflow machines. These machines did not allow multiple instances of the same routines to be executed simultaneously because the simple tags could not differentiate between them. Designs that use Content-addressable memory (CAM) are called dynamic dataflow machines. They use tags in memory to facilitate parallelism.

Any instruction to be executed after the result of a previous operation are serialized based on the CAM mechanism where the output of the previous operation would trigger a memory dependency tag. This tag would fire the current instruction. The trigger is achieved through a transient message. In contrast such messages are permanently stored in memory in case of Von Neumann architectures.

2.3 Motivation for multi-core framework

There is a wide variety of multi-core architectures, instruction set types, memory models and interconnects. It is difficult for a developer who is well versed in one such configuration to start developing seamlessly in another. For example, a GPU developer can’t develop code on a hypercube processor directly. Especially in the field of computer vision, the researchers have very little knowledge on exploiting parallel computing. Thus it is important to give the developer an interface where the architecture differences are abstracted out. Here the challenge is to generate optimized code using the interface. We illustrate two use-cases for a multi-core image
processing framework. The first use case, depicted in Fig. 1, is a hard coded accelerator which comprises of a section of the multi-processor system for which the generic framework library is compiled and burnt in ROM one time. Now this section behaves as an accelerator for non-block based functions. The second use case shown in Fig. 2, involves parsing any image processing application. The user has to write the code in a specific way to give compiler hints that a particular code section is block based or non-block based. And if it is non-block based, the implementation has to be in a particular way according to the framework. The compiler will now

Figure 1: One time compile architecture with single framework library.

Figure 2: Multi-Architecture intelligent compiler
be able to build this code for a particular platform. The framework abstracts out the higher level information and compiler takes it on from that level. The role of algorithmic parser common to both use cases is to identify the local, global and globalocal parts of a generic image processing algorithm. This can also be achieved manually by user tagging of the code sections.

2.4 Survey of multi-core frameworks

2.4.1 Pipelined Decomposition Tree

Pipeline decomposition tree (PDT) is a novel and recent multi-core framework proposed by [Ko, 2006]. PDT with an associated scheduling framework is used for mapping image processing applications onto embedded multiprocessor systems. PDT scheduling is based on a model of the target implementation as a coarse-grained (task-level), pipelined architecture. PDT scheduling spreads functional operations over the underlying pipeline through construction and iterative analysis of the PDT. Intuitively, the PDT can be viewed as a kind of depth first search tree whose nodes are mapped to stages of the targeted pipeline. Any number of nodes of the PDT can be mapped to a single stage of the pipeline. PDT scheduling ultimately generates schedules with different latency/throughput trade-offs to effectively explore the multidimensional space of signal processing performance considerations. Furthermore, the PDT scheduling process can take into consideration various scheduling constraints, such as constraints on the number of available processors, and the amounts of on-chip and off-chip memory, as well as performance-related constraints (i.e., constraints involving latency and throughput). The PDT scheduling approach places special emphasis on distinguishing and taking into account different modes of parallelism — task-level parallelism, as well as homogeneous and heterogeneous data parallelism — that must be exploited carefully to achieve efficient implementation of image processing applications.

2.4.2 Explicit Data-Parallel Representation of Image-Processing Programs

In this work, [Baumstark, 2005] proposes a framework to extract a high-level specification with explicit data-parallel semantics from a sequential C code. A dataflow representation for sequential programs written in C, an extended MDSDF (multi-synchronous dataflow) representation for representing explicit data-parallelism, and a pattern-matching system for abstracting the former into the latter is employed in the framework. MDSDF is an extension of SDF. In SDF models, production and consumption rates of (scalar) tokens are constant integers, leading to statically determinable execution rates for each node (process). MDSDF augments SDF with multi-element, multidimensional tokens. These tokens provide a useful representation for image processing algorithms, which often operate on regular sub-regions of an image, such as rows, columns, and tiled blocks. They also enhance by explicitly specifying the bounds (as ranges of values) of the multi-dimensional tokens, by providing a relative orientation of tokens when this is not implicit in the bounds, and by allowing symbolic expressions in dimensions.

2.4.3 Software architecture for user transparent parallel image processing

The core of the architecture is a library containing a set of abstract data types and associated pixel level operations executing in data parallel fashion [Seinstra, 2000]. Domain specific performance models are used as a basis for automatic optimization of applications implemented using the library. The library implementation has two independent steps namely modeling the image processing operations and parallel extensions. A set of operation classes can be identified that covers the bulk of all commonly applied image operations. Each operation that maps onto the functionality as provided by a generic algorithm is implemented by instantiating the generic
algorithm with the proper parameters, including the function to be applied to the individual data elements. For Parallel Extensions, three classes of routines are implemented (using MPI) that introduce the parallelism into the library: (1) data partitioning routines, to indicate which data parts should be processed by each processing unit, (2) distribution and redistribution routines, to scatter, gather, broadcast, and redistribute data structures, and (3) overlap communication routines, to exchange shadow regions, such as image borders in neighborhood operations.

3. Non-block based Algorithms

Image processing operators are commonly classified as:

- **Point Operators**: These operators act on input pixels to produce output pixels without requiring data from any other pixel.

- **Local Operators**: These operators need data from a restricted neighborhood of pixels to produce the output for a single pixel. These operators can be viewed as an operation of a local kernel on the input image.

- **Global Operators**: Processing of even a single pixel depends on the entire image.

Apart from the above set of operators, there exists a class of low level image processing operators, which heavily depends on the characteristics of the input image, that cannot strictly be classified *a priori* as a local or a global operator. This scenario occurs due to the fact that, for this class of operators, the output at a particular pixel may not always depend on all other pixels in the image (hence not strictly global) and also we cannot put a hard bound on the neighborhood of pixels that might be required to obtain the output (hence not local too). We introduce a new terminology - **Globalocal operators**, to refer to this class of image processing operators.

Point and local image processing operators can be implemented efficiently on multi-core systems using a simple data parallel (block based) image processing framework wherein each core works on a different chunk (one of the different non-overlapping blocks) of the original image in a parallel fashion [Nicolescu, 2000], [Bräun, 2001, 2000]. Most parallel image processing packages and algorithms handles only this kind of parallelism. Global and globalocal operators, both of which lead to algorithmic implementations which are inherently non-block based, cannot be implemented efficiently in the block-based framework. The use of a block-based framework for these classes of algorithms leads to enormous amount of communication (IPC) between the various cores. The amount of IPC required for global operators cannot be reduced any further, but for the class of globalocal operations this is not the case. We explore the use of a proper framework to optimize the inter processor communication overhead for the globalocal operators.

Examples of non-block based algorithms for globalocal operators includes many practically relevant algorithms like connected component labeling (CCL), Hysterisis thresholding, Chain coding, Crack coding, Hough transform, Integral image, Image segmentation, Contour following etc. In the rest of this section, we take a close look at some of these algorithms to highlight the common characteristics of the globalocal operators and to motivate the design of the multi-core framework for non-block based algorithms.
3.1. Connected Component Labeling (CCL)

Extracting and labeling of various disjoint and connected components in an image is central to many automated image analysis applications. Connected component labeling is used in computer vision to detect connected/unconnected regions in binary digital images, although color images and data with higher-dimensionality can also be processed. When integrated into an image recognition system or human-computer interaction interface, connected component labeling can operate on a variety of information.

Connected components labeling scans an image and groups its pixels into components based on pixel connectivity, i.e. all pixels in a connected component share similar pixel intensity values and are in some way connected with each other. The notion of connectedness depends on the set of neighboring pixels that are checked for similarity. Most commonly used notions of connectivity are the 4-connected or 8-connected pixels as depicted in Fig. 3. Once all groups have been determined, each pixel is labeled with a gray level or a color (color labeling) according to the component it was assigned to.

![Figure 3: Most commonly employed notions of connectedness: (a) 4-connected and (b) 8-connected pixels. The green pixel is the current pixel being analyzed for connectivity and the gray pixels depict the neighborhood defined in each case. The current pixel is compared with the neighboring pixels for similarity in pixel intensity.](image)

Here we give a brief description of the relatively simple two-pass algorithm originally proposed by [Rosenfeld, 1966] and later modified by [Lumia, 1983]. The algorithm iterates through 2-dimensional, binary data to label the 1-valued pixels into connected components. The algorithm makes two passes over the image: one pass to record equivalences and assign temporary labels and the second to replace each temporary label by the label of its equivalence class. During each pass, the algorithm iterates through each element of the data by column, then by row. On the first pass, if an element is not the background (0 pixel value):

1. Get the neighboring elements of the current element
2. If there are no neighbors, uniquely label the current element and continue
3. Otherwise, find the neighbor with the smallest label and assign it to the current element
4. Store the equivalence between neighboring labels

On the second pass, if an element is not the background

1. Relabel the element with the lowest equivalent label

An illustration of the above algorithm is shown in Fig. 4.

3.2 Hysteresis Thresholding

Hysteresis thresholding is a variant of the connected component labeling problem and plays a significant role in the highly popular canny edge detector. In order to eliminate the edges which represent noise in the image, each candidate edge is labeled with its gradient norm value and a hysteresis-based thresholding method is applied. The method is based on the assumption that
important edges should be along continuous curves in the image. Noisy pixels that do not constitute a line but have produced large gradients are ignored. Thus this algorithm gives well-connected edges while eliminating isolated noisy edges.

The hysteresis thresholding method can be decomposed into two steps – a double thresholding and an edge relaxation step. Double thresholding step uses two threshold values – high and low.
During double thresholding, values in the input image which are lesser than the low threshold are marked as background (0); those higher than the high threshold are marked as strong edge (2) and the once lying between the two thresholds are treated as weak edges (1). The Hysteresis thresholding operation first performs this double thresholding to give a tri-valued (output pixel can assume values 0, 1 or 2) image.

The edge relaxation step acts on the above tri-valued image to produce a binary image where each pixel is marked as either an edge pixel or a non-edge pixel. All strong edge pixels are treated as edge pixels. The weak edge pixels that are connected to strong edge pixels are also considered as edge pixels. Further, the weak edge pixels that have neighboring weak edge pixels that were converted to edge pixel is also to be treated as an edge pixel. All remaining pixels are marked as non-edge.

Though the double thresholding step is a point operation, the edge relaxation step is not block based as it involves traversing through the image in an arbitrary fashion.

3.3 Chain coding

When dealing with a region or object, we often require information about the contour of the objects to facilitate manipulation and measurements on the object. Chain coding refers to a method used to arrive at a compressed representation of such connected contours in an image.

This representation is based upon the work of [Freeman, 1961]. We follow the contour in a clockwise manner and keep track of the directions as we go from one contour pixel to the next. For the standard implementation of the chain code we consider a contour pixel to be an object pixel that has a background (non-object) pixel as one or more of its 4-connected neighbors. The codes associated with eight possible directions are the chain codes and, with $x$ as the current contour pixel position, the codes are generally defined as:

$$\begin{align*}
3 & \quad 2 & \quad 1 \\
4 & \quad x & \quad 0 \\
5 & \quad 6 & \quad 7
\end{align*}$$

See Fig. 6 for an example of chain coding of the contour of a given region in an image using the above chain codes.

![Figure 6: Contour (dark pixels) of a region (shaded) and its chain coding from an initial reference pixel.](image-url)
3.4 Other Similar Algorithms

In computer vision, segmentation refers to the process of partitioning a digital image into multiple segments (sets of pixels) (Also known as superpixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze [Shapiro, 2001]. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain visual characteristics.

A summed area table (also known as an integral image) is an algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a grid. As the name suggests, the value at any point \((x, y)\) in the summed area table is just the sum of all the pixels above and to the left of \((x, y)\), inclusive: [Crow, 1984], [Viola, 2002].

3.5 Generalization of Non-block based algorithms

The following section examines each non-block based algorithm and highlights the challenge involved in optimally splitting these algorithms amongst different processors of different characteristics (heterogeneous multi-core environment).

The first algorithm to be considered is the connected component labeling. The initial step involved in the splitting this algorithm is allocation of blocks of the original image to different processors based on various factors – internal memory, speed (MHz) of each processor etc. Overlap processing: The blocks allocated to various cores can in general have an overlap – usually one pixel overlap is sufficient for neighborhood operations.

Once block allocation is done, the next logical operation to achieve is merging or synchronizing operations across processors. Figure 7-1 shows the blocks of an input image being allocated different processor. It also depicts that the overlap regions being passed to both processors to be used to communicate any border information to notify the other core of a temporary label that is to be taken into account to label an object in the image that is split across processors. This is a classic characteristic in any global/local operation as will be seen in further examples.

**Figure 7:** Global information to be passed across cores for local scope operators.

In Fig. 7-2, representing the hysteresis thresholding algorithm, we see that the algorithm requires some border information like the connected component labeling. This information is an
indication of whether a strong edge (on core 0) is going to cause a weak edge (on core 1). In Fig. 7-3, the chain coding algorithm is depicted. When the blocks are split amongst different cores the following **boundary information** needs to be conveyed across to other cores – overlap pixel’s reference point and the direction delta of the of the corresponding pixel from the other core where the pixel is being resolved.

Segmentation is a similar problem to connected component labeling, except it is being generalized to grayscale or in a more general sense for any color, with a particular parameter used to group the pixels (absolute difference in the pixel gray scale intensities, similarly in color - RGB values or YUV values). This is what needs to be passed as **border information** when segmentation is to be split amongst different cores.

### 4. Multi-Core Framework

A multi-core framework needs to be envisaged taking into account all the above factors. The framework design can be segregated into following sections in an attempt to see the full data flow. This data flow primarily depicts a critical **2 pass methodology** (Fig. 8) followed in forming the framework. In general the number of passes required can be more than 2. The first pass has two primary functions:
- Block processing : labeling, marking reference points, marking weak/strong edge links
- Obtain border information to be passed on to global table that shall be processed by host or master processor.
- The 1st pass produces some intermediate results in the blocks that need to be written back into the same block

**Higher level operations or Global resolutions**
- Resolution processes like union find algorithm, link list merging and other process that is required to complete the global processing so that it can be applied in local scopes. This is the stage where multiplicity of decisions is made based on **boundary information**.

The 2nd pass constitutes of the following tasks
- The second pass involves applying the finalized global updates and writing the final block of the output image.

These two passes to an extent of efficiency has created a block independent processing capability.

PASS 1 overlap with global processing determines the efficiency in computation of global information vs. full global information required to make hierarchical decision.

PASS 2 overlap with global processing determines the efficiency in how much global info is required to process an image block locally. More this overlap is lesser the global information impact

**Figure 8**: The two-pass example of the framework

Framework flow diagram in Fig. 9, displays all the data flow paths involved between hardware components and software (abstraction) entities. The block allocator is a software mechanism that does basic bookkeeping in tracking what block number is being processed and what boundaries are being considered. It optimally distributes to all the cores in the system (based on internal memory, MHz etc described earlier).
The implementation of the local operation is algorithm specific – for example it might be a labeling process in case of connected component labeling, or calculation of direction delta in case of chain coding.

In Fig. 9 the image area and data area refer to logical sections in the shared memory region that the multiprocessor arrays share. Image area refers to the section of memory where blocks of image are stored in contiguous fashion. While the data are refers to a managed section of any global information required to be communicated to other cores – this is done by the activity of the host on the global tables. Apart from the global tables there are tables maintained to reflect global/border information changes – local tables – they store results from every local operation. This is stored and modified in between passes by the host. This same table is then applied during the 2nd pass to reflect all the border processing/decisions made into every block to complete the processing flow.

**FRAMEWORK FLOW DIAGRAM**

![Diagram](image)

**Figure 9: Framework flow**

### 4.1 Scalability

Scalability here refers to the frameworks capability to adapt to multiple facets of the multi-core heterogeneous SoC environment. It describes the efficiency of the framework to extend to different number of processors (N), different processing speeds (F₁ to Fₙ), different internal memory sizes (M₁ to Mₙ), this can also be extended to different interconnect delays (t₁ to tₙ), different Instruction Set Architecture (ISA) of the Core.

Factors affecting scalability:
- Ability to map to ISA that is not a part of the input criterion to the intelligent SoC multi-architecture compiler
- Ability to map to a multi-core system acting like a single core processor. (quotient of abstraction of system architecture internals)
Rate analysis of block processing – taking into account available internal memory – memory hierarchy organization within a processor – and the processing speed MHz of the core.

4.2 Block allocation

Block allocation algorithm should also take into account the following issues:

- Difference in the complexity of image in the input block: a comparative study can be done by examining the extremes – an image block full of zeros (no processing) – Vs – and an image block filled with groups of ones. The block allocation algorithm should take care that no core is left untended without a block to process.
- Blocks allocated to different cores occur as neighbours then the border global information exchanged can be managed more efficiently in the global table since this tends more spatial locality to the global data stored in the data area of the shared memory in the system.

5. Conclusion

In this paper, we propose a multi-core software design framework for globalocal image processing operations which are treated as corner cases and not usually covered in existing frameworks. The parallelism is split into three abstractions namely block allocation, overlap processing and higher level resolution. This framework is scalable across the number of processors and heterogeneity of the multi-core system. A hardware abstraction model is assumed and software architecture is built upon it.

Existing compilers are incapable of splitting a non-block based algorithm, which conventionally is handled by a system engineer (user) aware of target multi-core environment. The framework helps provide intelligence to compilers to efficiently distribute these algorithms amongst various cores.

The framework currently dealt with homogeneity and design re-use. Optimization of the framework for a target platform has to be studied. All globalocal image processing functions will fit into this framework. But there may be algorithms which are global and won’t fit into this framework. Thus the framework has to be extended to capture these functions.

References


