Interleaved Pixel Lookup for Embedded Computer Vision

Kota Yamaguchi, Yoshihiro Watanabe, Takashi Komuro, and Masatoshi Ishikawa
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
{Kota_Yamaguchi, Yoshihiro_Watanabe, Takashi_Komuro,
Masatoshi_Ishikawa}@ipc.i.u-tokyo.ac.jp

Abstract

This paper describes an in-depth investigation and implementation of interleaved memory for pixel lookup operations in computer vision. Pixel lookup, mapping between coordinates and pixels, is a common operation in computer vision, but is also a potential bottleneck due to formidable bandwidth requirements for real-time operation. We focus on the acceleration of pixel lookup operations through parallelizing memory banks by interleaving. The key to applying interleaving for pixel lookup is 2D block data partitioning and support for unaligned access. With this optimization of interleaving, pixel lookup operations can output a block of pixels at once without major overhead for unaligned access. An example implementation of our optimized interleaved memory for affine motion tracking shows that the pixel lookup operations can achieve 12.8 Gbps for random lookup of a 4x4 size block of 8-bit pixels under 100 MHz operation. Interleaving can be a cost-effective solution for fast pixel lookup in embedded computer vision.

1. Introduction

One of the issues in designing embedded hardware for computer vision is the arrangement of memory for pixel lookup operations, where coordinates are mapped to pixels. System designers have to carefully design a lookup memory while balancing the requirement of performance, hardware cost, and power consumption in an embedded system.

The first difficulty in arranging memory for pixel lookup operations comes from the capacity requirements in computer vision. For example, more than 2.45 Mbit is required to store a VGA size 8-bit gray scale image. It takes almost 1.3 million gates if we use a 0.5 gates/bit SRAM macro in an ASIC process. In contrast to the unconstrained use of redundant memory modules in PC-based systems, we cannot easily deploy many of such area-consuming modules in embedded hardware.

In addition to the capacity requirement, computer vision tasks require memory throughput that is speedy enough to handle video streams. Computer vision tasks often require a system to repeatedly perform computation on massive amounts of video data at a specified frame rate (30 fps or more). Despite the limitation in hardware resources, enormous memory throughput is necessary to supply source pixels to internal computation for real-time operation. How to efficiently arrange memory for fast lookup operations is an important consideration in an embedded computer vision system.

In this paper, we focus on the acceleration of pixel lookup operations by interleaving, a technique to parallelize memory access by partitioning a long data word and distribute it into multiple banks. It can improve access throughput without doubling total storage capacity. So far, interleaving has been successfully utilized in applications which requires extreme memory performance, such as computer graphics where enormous memory bandwidth is required for texture mapping[3][8]. Although interleaving has been used in computationally intensive tasks like graphics, there has been made little discussion of its applicability to computer vision. We found that computer vision applications could best take advantage of interleaving utilizing spatial locality in a sequence of memory accesses.

The remaining portions of this paper are organized as follows; in the section 2, we will briefly discuss the reasons why we focused on interleaving for pixel lookup operations in embedded computer vision. Section 3 presents the details of our interleaved memory architecture optimized for pixel lookup operations. Section 4 describes an example implementation of our interleaved memory for affine tracking application. The results show that pixel lookup bandwidth could reach 12.8 Gbps with an FPGA operating at 100MHz. Finally, we will conclude this paper in the section 5.

978-1-4244-2340-8/08/$25.00 ©2008 IEEE

2. Motivation

In computer vision, spatial locality is implicitly present in a sequence of pixel lookup operations. Locality in memory access provides an opportunity to parallelize system memory operation. This section briefly discuss the spatial locality in pixel lookup operations and the benefit of interleaving in embedded computer vision along with the similarity to texture memory in computer graphics.

2.1. Locality in Pixel Lookup

A lookup operation is used in almost all embedded computer vision systems as a function to convert coordinates to pixels. Consider if we want to retrieve the data of a single pixel at the coordinates of (i, j) from an input image I. To obtain the pixel $I(i, j)$, it is necessary to implement $I(\cdot, \cdot)$, a "pixel mapping function," which converts a coordinates (i, j) to a pixel $I(i, j)$. Random access memory is used to implement this pixel lookup operation, because it is practically impossible to implement this function with wired logic.

The pertinent characteristic of pixel lookup is that spatial locality is present within a sequence of operations. Spatial locality appears if lookup operation follows some spatial filtering process, such as interpolation or 2D convolution. Assume we have a random sequence of lookup indices $\{(i_0, j_0), (i_1, j_1), ...\}$. With a 2×2 post-filtering process, the index sequence changes to have a regular pattern of access to adjacent pixels, such as $\{(i_0, j_0), (i_0 + 1, j_0), (i_0, j_0 + 1), (i_0 + 1, j_0 + 1), (i_1, j_1), (i_1 + 1, j_1), (i_1, j_1 + 1), (i_1 + 1, j_1 + 1), ...\}$. Figure 1 illustrates this sequence of accesses. The pattern is dependent on the spatial attributes of the window used in the following stage of operation. Importantly, this locality in pixel lookup is commonly used in many computer vision algorithms, since filtering pixels is fundamental to almost all image processing tasks. We can utilize this spatial locality to parallelize memory operation with interleaving, as we later explain.

Unfortunately, temporal locality is not as apparent as spatial locality in pixel lookup. A sequence of lookup indices $\{(i_0, j_0), (i_1, j_1), ...\}$ usually points to different locations except for some overlap between a adjacent pixels of consecutive lookup operations. Instead, temporal locality appears is a relatively long term effect, as operations performed on the next video frame may repeat the same pattern. This makes it difficult to accelerate computer vision tasks using small cache memory, because it is necessary to store the whole data stream in a cache to have an effect. In contrast, using the spatial locality is a quick and effective approach to parallelization of memory access.

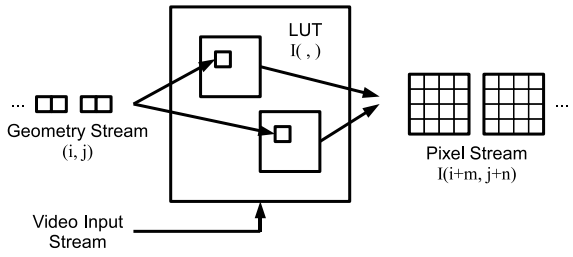

Figure 1. Pixel lookup operations

2.2. Benefits of Interleaving

Interleaving can parallelize a memory access if there exists some spatially regular pattern in a sequence of accesses. Interleaving first divides consecutive data into several partitions, so that multiple memory banks can be allocated for each partition. This ensures that every memory access can be parallelized, since data stored in different banks can be read out at the same time.

The benefit of interleaving compared to other approaches to parallelize memory operation is throughput gain without an additional requirement for storage capacity. Doubling performance of lookup operations usually requires doubling hardware cost to duplicate the memory table, since memory access overlapped to the same memory bank cannot be made at the same time. We need to mirror the same data volume to another bank for completely parallelized random access. Or, it is necessary to double operational frequency to double random access throughput, with large penalty in power consumption. These penalties can be critical for embedded systems. However, interleaving does not require additional memory nor frequency increase. Since each memory banks only needs to store partitioned data, total capacity requirement does not increase in the case of interleaving. This characteristic is most suitable for the memory requirements of embedded systems.

The drawback of interleaving is its applicability. Interleaving is in essence a method to broaden the format of a data word. To take advantage of it, memory access must be in regular sequential order, otherwise overhead in data alignment severely degrades the throughput gain. In other word, application has to contain some spatial locality in memory access. Fortunately, computer vision tasks present regular pattern in a sequence of pixel lookup operations and can benefit from interleaving.

2.3. Previous Work in Graphics Hardware

Although little discussion was made for the applicability of interleaving in embedded computer vision, computer graphics has successfully utilized it for texture mapping[3]. In texture mapping, fragments of texture must access multiple texels for interpolation or texture filtering[9]. Texture

memory, a dedicated memory for texture mapping, usually has interleaved banks to meet the requirement of extreme memory bandwidth[8][4] in computer graphics.

Pixel lookup operations in computer vision are similar to texture mapping in computer graphics except for a slight difference in requirements for capacity and functionality. Computer vision does not require extremely large memory capacity, as opposed to texture memory which has to store many texture images.

Instead, computer vision requires more flexibility in write operations. In computer vision, pixels stored in a lookup table are constantly updated as the system receives video stream, while texels in graphics are not always updated dynamically. Recent graphics hardware has introduced the capability of flexible write operations[7], however, limitations in memory access prevents it from being fully utilized in a computer vision system. In a sense, computer vision requires texture memory to receive video stream from a frame buffer while operating as a texture table for internal computation. In computer vision, memory has to be more flexible for both write and read operations.

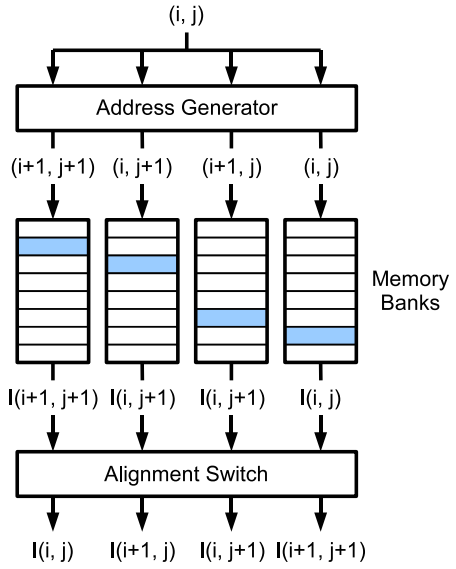

Figure 2. Interleaved memory for pixel lookup

3. Interleaved Memory for Pixel Lookup

Although interleaving can improve the throughput of memory operations, we need to optimize the memory structure for pixel lookup operations to solve the alignment issue. An access not aligned to address boundary severely degrade the throughput of memory operations, causing the overhead due to adjustment of data format. To avoid the effect of unaligned access, we designed dedicated hardware modules to support data alignment in pixel lookup operations. This section describes our method of data partitioning and alignment support.

3.1. Architecture

Figure 2 shows an example of interleaved memory architecture arranged for 2×2 pixel lookup operations. The architecture consists of an address generator, memory banks, and a switch to align data. With this arrangement, input of a single index (i, j) can produce parallel output of pixels $\{I(i,j), I(i+1,j), I(i,j+1), I(i+1,j+1)\}$ without overhead in throughput for adjustment of unaligned data. The number of memory banks is scalable depending on the requirement of the application and the availability of hardware for these three modules. In theory, we can arrange $M_x \times M_y$ memory banks to output $M_x \times M_y$ pixels at each lookup.

3.2. Data Partitioning

The first step to design interleaved memory is to partition data volume into data words and allocate memory banks to them. There exists several strategies to partition data, such as per-line partitioning or per-block partitioning. However, in computer vision, the generally effective approach is to partition data into 2D blocks. Figure 3 shows an example of this partitioning. Actually, the optimal scheme depends on the shape of window used in post-filtering processes. Hariyama discusses in-depth the optimization of memory allocation in [5]. In this paper, we adopt a regular square 2D block under the assumption that in most cases window shape becomes square in post-filtering.

In this 2D partitioning scheme, data elements in a block are interleaved by allocating different memory banks to them with the same address. Address columns in the figure 3 are allocated to different memory banks. The row address of the block to which a pixel at coordinates (i, j) belongs is identified by $(i/M_x, j/M_y)$, if each block has $M_x \times M_y$ elements.

The block size is determined by how much data-parallelism is necessary for post-filtering processing. The maximum size will be $M_x \times M_y$ to supply all the pixels for a filtering process which has an $M_x \times M_y$ size kernel. For example, 2×2 block partitioning is appropriate for fully parallelized 2×2 bilinear interpolation. Of course, we can arrange smaller block sizes if we do not need to supply all the source pixels at once depending on performance requirement.

This partitioning scheme is effective not only for specialized parallel hardware but also for DSPs or general purpose processors with SIMD extensions, such as MMX/SSE in the Intel Architecture[6]. Compared to 1D partitioning of pixel rows, 2D partitioning reduces the latency necessary to get source pixels for 2D filtering.

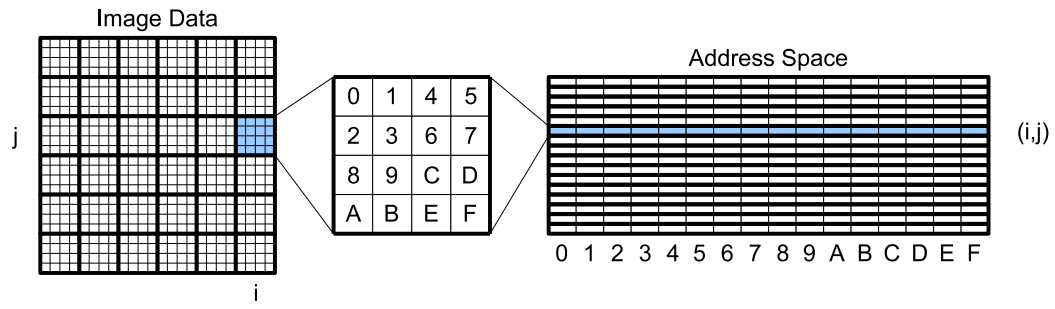

Figure 3. 2D block partitioning

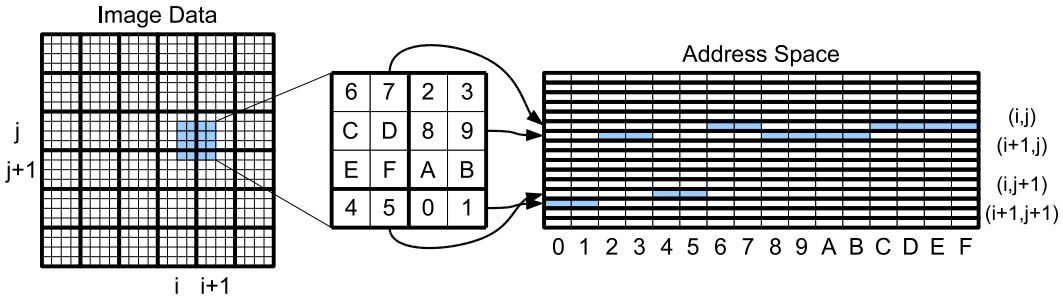

Figure 4. Unaligned access

3.3. Alignment Support

Data alignment is a major problem for parallel architectures that deal with packed data[10]. Alvarez reports the severe influence of alignment for SIMD processing in video codec applications[1]. Parallel architectures can achieve the highest performance if all the memory access is regular and aligned, however, it is often the case that data are not aligned to address boundaries. Unaligned access requires multiple memory accesses and adjustment of data format. For the 2D partitioning scheme, an unaligned access to four adjacent blocks requires four memory accesses and data alignment of these output. This overhead severely degrade memory throughput (almost half in case of 2×2 block partitioning), and is proportional to the size of data block.

The overhead of multiple accesses can be avoided by implementing indirect access to each interleaved memory bank. In the figure 2, the address generator provides the functionality for indirect access. With the 2D partitioning scheme, interleaving guarantees that each data element in a block is stored in a different memory bank regardless of alignment of the block. Even if a parallel access overlaps block boundary, no overlap occurs in an access to the same memory bank. As long as access is to data packed as a block, we can always read out the block at once by pointing to different address for each memory bank. Figure 4 shows an example of unaligned access. An overlapped access never occurs within the same memory bank.

To maximize throughput, additional support is necessary for adjustment of data format. Access not aligned to address boundary shuffles the data format of pixels output from memory banks. To adjust the format, we added a switch after the output of memory banks. Figure 2 illustrates the functionality of the switch. The structure of switch depends on data format. In our case of 2D blocks, the switch will be a cascade of 1D rotators. Although the switch adds slight latency and hardware cost, it can help maximize memory throughput.

To sum up, we can avoid the performance penalty of unaligned access by preparing an address generator for indirect access and a switch to adjust data format of raw output from memory banks.

This direct implementation of data alignment can effectively reduce the overhead of alignment present in SIMD instructions in general purpose processors[1]. For general purpose systems, the effective approach is to implement the data alignment into a DMA controller so that more cycles are available for computation.

4. Example Implementation: Affine Tracking

To evaluate the effectiveness of the optimized interleaved memory in embedded computer vision, we implemented an affine motion tracker with an FPGA as an example application. We chose the Lucas-Kanade algorithm[2] for affine

tracking, because it is known for fast computation compared to full search algorithms such as regular block matching. On implementing the algorithm, memory bandwidth required for pixel lookup tends to limit the overall performance even if enough hardware resources can be arranged for computation.

In this section, we will explain the details of Lucas-Kanade algorithm and our hardware implementation with interleaved memory.

4.1. Lucas-Kanade Algorithm

The Lucas-Kanade Algorithm[2] uses Gauss-Newton gradient descent to perform non-linear optimization to solve image alignment problems formulated as least squares. The algorithm is widely used in computer vision applications, such as optical flow estimation, tracking, or mosaic construction. For affine tracking, we will use Lucas-Kanade to solve optimal parameters of affine transformation between input video and template image patches every frame.

Given a template image $T(\mathbf{x})$, the goal of Lucas-Kanade is to align $T(\mathbf{x})$ to an input image $I(\mathbf{x})$, where $\mathbf{x} = (x, y)^T$ is a column vector containing the pixel coordinates. For formulation as least squares, the error function to be minimized will be given:

$$E(\mathbf{p}) = \sum [I(\mathbf{W}(\mathbf{x};\mathbf{p})) - T(\mathbf{x})]^2 \tag{1}$$

where $\mathbf{W}(\mathbf{x};\mathbf{p})$ represent the parametrized set of geometrical warps and $\mathbf{p} = (p_1, p_2, ..., p_n)^T$ is a vector of its parameters. In the case of affine warps, $\mathbf{p}$ has six parameters, and so $\mathbf{W}(\mathbf{x};\mathbf{p})$ will be:

$$\mathbf{W}(\mathbf{x};\mathbf{p}) = \begin{bmatrix} p_1 & p_3 & p_5 \\ p_2 & p_4 & p_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{2}$$

The basic strategy is to find a solution without searching every possible parameter. In practice the strategy is to iteratively solve increments of the parameters $\Delta\mathbf{p}$ under a known current estimate of $\mathbf{p}$; the error function $E(\mathbf{p}+\Delta\mathbf{p})$ is minimized with respect to $\Delta\mathbf{p}$, and the parameters are updated:

$$\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}. \tag{3}$$

By linearizing the error function $E(\mathbf{p}+\Delta\mathbf{p})$ with respect to $\Delta\mathbf{p}$, the parameter update $\Delta\mathbf{p}$ can be obtained by solving the following simultaneous equations:

$$H \cdot \Delta\mathbf{p} = \mathbf{s} \tag{4}$$

where H is Hessian matrix:

$$H = \sum_{\mathbf{x}} \left[\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}}\right]^{\mathrm{T}} \left[\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}}\right] \tag{5}$$

and $\mathbf{s}$ is the steepest decent parameter updates (SDPU):

$$\mathbf{s} = \sum_{\mathbf{x}} \left[\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}}\right]^{\mathrm{T}} [T(\mathbf{x}) - I(\mathbf{W}(\mathbf{x};\mathbf{p}))]. \tag{6}$$

In Equation (5) and (6), $\nabla I = (\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y})$ is the gradient of image I, and the term $\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ is the Jacobian of the warp $\mathbf{W}(\mathbf{x};\mathbf{p})$. For affine transformation, the Jacobian will be expressed as:

$$\frac{\partial \mathbf{W}}{\partial \mathbf{p}} = \begin{pmatrix} x & 0 & y & 0 & 1 & 0 \\ 0 & x & 0 & y & 0 & 1 \end{pmatrix}. \tag{7}$$

The update $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$ is iterated until the estimate of $\mathbf{p}$ converges. Convergence can be tested by thresholding the norm at each frame:

$$|\Delta\mathbf{p}| \leq \epsilon. \tag{8}$$

For real-time operation, it may be necessary to halt iteration if it takes too much time to converge.

4.2. Hardware Design

The most computationally intensive part in Lucas-Kanade algorithm is the calculation of gradient and interpolation. The throughput of pixel lookup operations to supply source pixels to these filtering processes determines the achievable tracking performance. We examined the effectiveness of interleaving on this bottleneck.

Figure 5 shows the overall architecture of our hardware implementation of the Lucas-Kanade algorithm. To balance the computational load, we divided the implementation of Lucas-Kanade algorithm into two modules: Hessian pipeline and floating point unit (FPU).

4.2.1 Hessian Pipeline

The Hessian pipeline is a dedicated hardware module that computes Hessian matrix H and SDPU $\mathbf{s}$ given respectively in equations (5) and (6) from the input of the current approximation of affine parameters $\mathbf{p}$. Because of the intensive computational cost of these processes, we maximized their throughput using the interleaved memory.

Each module in Hessian pipeline shown in Figure 5 provides the following functions.

- The affine warp calculator generates a stream of warped coordinates $\mathbf{W}(\mathbf{x};\mathbf{p})$ from the input of the current approximation of affine parameters $\mathbf{p}$.
- The filter kernel generator calculates bilinear operator from the warped coordinates $\mathbf{W}(\mathbf{x};\mathbf{p})$. In addition, the generator convolves it with Sobel operators $\nabla = (\nabla_x, \nabla_y)$ to calculate a cascaded filter kernel for

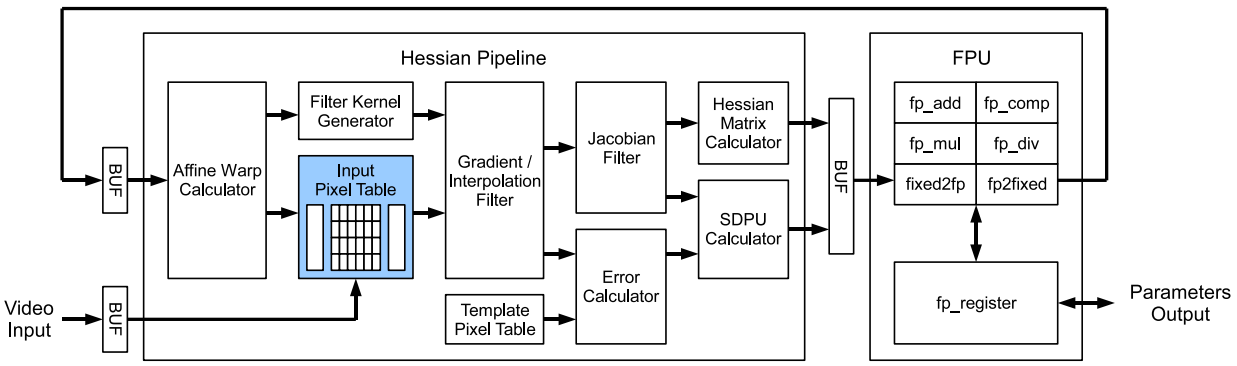

Figure 5. Lucas-Kanade pipeline

image gradients in advance; If we choose $f(\mathbf{x})$ as a bilinear operator, this stage generates $f(\mathbf{W}(\mathbf{x};\mathbf{p}))$ and $\nabla * f = (\nabla_x * f(\mathbf{W}(\mathbf{x};\mathbf{p})), \nabla_y * f(\mathbf{W}(\mathbf{x};\mathbf{p})))$.

- The input pixel table outputs a parallel stream of source pixels with interleaved memory. It also receives a video stream to update pixels every frame.
- The template pixel table stores template images and outputs source pixels for error calculation. It is not necessary to interleave it because we can align template's coordinates to integer and no interpolation is required.
- The Gradient/Interpolation filter generates $I(\mathbf{W}(\mathbf{x};\mathbf{p}))$ and $\nabla I(\mathbf{W}(\mathbf{x};\mathbf{p}))$ from the source pixels I and generated filter kernels $\{f, \nabla * f\}$.
- The Jacobian filter multiplies $\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ with a stream of gradient output.
- The error calculator computes the difference of $T(\mathbf{x})$ and the warped input $I(\mathbf{W}(\mathbf{x};\mathbf{p}))$.
- The Hessian matrix calculator and SDPU calculator multiplies and accumulates input streams to obtain H and $\mathbf{s}$.

We applied the interleaved memory to the input pixel lookup table where source pixels are supplied to interpolation and gradient filters. This operation converts output of affine transformation $\mathbf{W}(\mathbf{x};\mathbf{p})$ to $\mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p}))$ using the input pixel table. To fully parallelize the cascaded filter operation in next stage, we arranged interleaved memory banks to correspond to the size of the filter kernel; since we adopted 2×2 bilinear operator f and 3×3 Sobel operator, the input pixel table has 4×4 memory banks so that it can supply all pixels necessary to output one filtered pixel at once. The acceleration in pixel lookup can effectively give an increase in performance of the Lucas-Kanade algorithm, which requires the computation of these filtered pixels over each image patch at each iteration. With interleaving, the throughput become 4×4 times faster than non-interleaved sequential access at the same operational frequency.

Actually, there is another approach to accelerate pixel lookup operations other than interleaving in a Hessian pipeline. It is possible to compute image gradients in advance and store them into other pixel tables, because the Sobel operator is not dependent to coordinates $\mathbf{x}$ and static over iterative computation in each frame. This pre-filtering approach can reduce the latency in the post-filtering approach, which applies Sobel filters at each iteration. However, pre-filtering requires additional memory space to store pixels, which tends to be costly in embedded implementation such as ASIC or FPGA. In our post-filtering implementation, we only need to store one frame of input pixels $I(\mathbf{x})$ in the pixel table. In contrast, in pre-filtering approach, we need to store image gradients $\nabla I(\mathbf{x}) = (\nabla_x I(\mathbf{x}), \nabla_y I(\mathbf{x}))$ in addition to $I(\mathbf{x})$ resulting in a three fold increase of cost for memory resources. Instead of this expense, we chose the post-filtering approach for the gradient computation. It is more beneficial to decrease memory requirement in embedded implementation, even if there is a slight increase in requirement for logic to align output of interleaved memory and calculate filtered pixels every iteration. The pre-filtering approach would be advantageous when hardware cost for logic exceeds the hardware cost for memory, or a bottleneck exists in computation.

In addition to interleaving, we implemented double buffering into the input pixel table utilizing the dual-port functionality of RAM resources in an FPGA. Although it raises the capacity requirement of memory, memory operations can be overlapped for data transfers and internal pixel lookup operations. This overlap is extremely useful for computer vision, because video input is transferred almost constantly and it can allow more time for computation. Of course, we can remove it if cost limitations do not permit additional memory resources.

4.2.2 FPU

In the FPU, the current approximation of affine parameters $\mathbf{p}$ is updated with increments $\Delta\mathbf{p}$ calculated from the input of Hessian matrix H and SDPU $\mathbf{s}$. Also, the FPU takes the responsibility to determine whether to continue iteration or not by evaluating the equation (8). We adopted a general-purpose floating point hardware in this case for the following reasons.

First, numerical precision is essential for the computation of the equations (4) and (8). An impractical amount of dynamic range is necessary to accurately solve the equation (4) with fixed point format; In the most complex path, we have to compute a cascade of 10 subtractions, 10 multiplications, and 11 divisions. Floating point operation provides more cost-effective implementation for this kind of numerical computation.

Also, cycles available to solve the equations (4) and (8) are less tight than those in Hessian pipeline under a macro architecture consisting of an FPU and Hessian pipeline. Compared to the size of data volume the Hessian pipeline deals with, small fragments of streams are computed in the equations (4) and (8). In this case, we can reduce the consumption of hardware resources by sharing an arithmetic unit among several processes.

In addition, complexity in data flow is more critical than the data volume to solve the equations (4) and (8). Because of their irregular and deep data dependency, we cannot take much advantage by designing an inflexible hardware pipeline. Rather, programmable hardware best suits this case, with minimum hardware resources.

4.3. FPGA Synthesis

Table 1 shows the synthesis result of our Lucas-Kanade pipeline. The target device is Xilinx Virtex-4 FPGA XC4VLX200, and the synthesis is done in Xilinx ISE 9.2i software. Every module is implemented on a single Virtex-4 device.

For the implementation of the interleaved pixel table, we arranged dual-port RAM blocks in the FPGA device. The table can store the maximum of two 512×256 size images with 8-bit gray scale pixels for video input, enabling double buffering of video input. It occupied 256 RAM blocks out of 336 in the FPGA device. This high occupancy of memory resources provides a good example of how important techniques to reduce the memory consumption are in embedded implementation.

The results show that the device can operate at 264 MHz thanks to our finely pipelined architecture. Although it is difficult to operate the device at 264 MHz, the estimated performance of pixel lookup operations reaches 12.8 Gpixel/s even under the condition of operation at 100 MHz. The throughput is 16 times larger than those of regular single port configuration of pixel lookup, and about 3 times larger than those of simple interleaving without alignment support depending on the tracked object in video data. The acceleration enables the overall pipeline to fully utilize parallel processing at data level, resulting in the ability to apply 172 arithmetic operations in the same cycle time and a total throughput of 344.1 Gbps at 100MHz operation. Thanks to the performance gain by interleaved pixel lookup, it is estimated that we can track the affine motion of 5 pieces of 64×64 size template patches at 200 fps if all of the Lucas-Kanade computation converges within less than 10 iteration at each frame.

FPGA	Virtex-4 XC4VLX200
Maximum Frequency	264.890 MHz
Slices	3,108 / 890,833 (3%)
DSP Slices	75 / 96 (79%)
RAM Blocks	266 / 336 (78%)
	(4,788 Kb)

Table 1. Synthesis results

Extending beyond ASIC or FPGA implementation, our interleaved architecture can be also beneficial for other implementations such as a DSP system. To take advantage of interleaving in a DSP system, the approach is the arrangement of 2D data partition and alignment support in a DMA controller. Unless a performance bottleneck exists in arithmetic operations, the system would benefit from the acceleration of pixel lookup operations by interleaving, without an additional data storage.

5. Conclusion

This paper described an interleaved memory architecture optimized for pixel lookup operations in computer vision tasks. We discussed spatial locality present in a sequence of pixel lookup operations and the applicability of interleaving for them along with the similarity to texture memory in graphics hardware. To fully take advantage of interleaved access in pixel lookup operations, we implemented 2D data partitioning and alignment support as dedicated hardware. The interleaved memory architecture enables parallel output of a block of pixels without overhead in throughput due to unaligned access. Also, the implementation requires only a slight increase in logic for data alignment without additional storage capacity in the memory. The example implementation for affine tracking on an FPGA demonstrated that the throughput of pixel lookup operations could reach 12.8 Gbps, providing enough performance for real-time operation. Interleaved pixel lookup operations will be beneficial to many of embedded computer vision applications.

References

[1] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance impact of unaligned memory operations in simd extensions for video codec applications. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 62–71, April 2007.

[2] S. Baker and I. Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal of Computer Vision*, 56(3):221 – 255, March 2004.

[3] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1995.

[4] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. *SIGARCH Computer Architecture News*, 25(2):108–120, 1997.

[5] M. Hariyama, M. Kameyama, and Y. Kobayashi. Optimal periodical memory allocation for logic-in-memory image processors. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 193, Washington, DC, USA, 2006. IEEE Computer Society.

[6] A. Peleg, S. Wilkie, and U. Weiser. Intel mmx for multimedia pcs. *Commun. ACM*, 40(1):24–38, 1997.

[7] M. Pharr. *GPU Gems 2*. Addison-Wesley Professional, 2005.

[8] A. Schilling, G. Knittel, and W. Strasser. Texram: a smart memory for texturing. *Computer Graphics and Applications, IEEE*, 16(3):32–41, May 1996.

[9] H.-C. Shin, J.-A. Lee, and L.-S. Kim. A cost-effective vlsi architecture for anisotropic texture filtering in limited memory bandwidth. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(3):254–267, March 2006.

[10] J. Tanskanen, T. Sihvo, and J. Niittylahti. Byte and modulo addressable parallel memory architecture for video coding. *Circuits and Systems for Video Technology, IEEE Transactions on*, 14(11):1270–1276, November 2004.