Lucas-Kanade Optical Flow Estimation on the TI C66x Digital Signal Processor

Fan Zhang*, Yang Gao*, Jason D. Bakos†
* Department of Computer Science and Engineering, University of South Carolina
{zhangf, gao36}@email.sc.edu
†Department of Computer Science and Engineering, University of South Carolina
jbakos@cse.sc.edu

Abstract—**Optical flow is a computer vision operation that seeks to calculate the apparent motion of features across two consecutive frames of a video sequence. It is an important constituent kernel in many automated intelligence, surveillance, and reconnaissance applications. Different optical flow algorithms represent points in the trade off space of accuracy and cost, but in general all are extremely computationally expensive. In this paper we describe an implementation and tuning of the dense pyramidal Lucas-Kanade Optical Flow method on the Texas Instruments C66x, a 10 Watt embedded digital signal processor (DSP). By using aggressive manual optimization, we achieve 90% of its peak theoretical floating point throughput, resulting in an energy efficiency that is 8.2X that of a modern Intel CPU and 2.0X that of a modern NVIDIA GPU. We believe this is a major step toward the ability to deploy mobile systems that are capable of complex computer vision applications, and real-time optical flow in particular.**

I. Introduction

Optical flow estimation on embedded processors has a key role in robotic vision, surveillance system design, and other computer vision tasks. Optical flow seeks to calculate the motion of objects such as edges, corners, surfaces, or other complicated color patterns between consecutive frames in the video stream. It is widely used in motion-based image processing and when measuring the movement of the video background, tracking moving objects, and motion-based video compression.

The existing Optical Flow methods can be categorized into four types: block-based methods, spatiotemporal differential methods, frequency-based methods, and correlation-based methods [4], [5], [6], [7], [8] [1]. Lucas-Kanade is a spatiotemporal differential method and is perhaps the most well-known and most studied. This is perhaps due to it having fine-grain parallelism and reasonably high accuracy [2] [3]. In this paper, we focus on developing methodologies to accelerate Lucas-Kanade method on the TI C66x DSP, a relatively new microarchitecture that offers comparable peak floating point capability to a desktop CPU while having a comparable power envelope to a smartphone processor.

Floating point accelerators such as general purpose GPUs (GPGPUs) are widely applied as coprocessors for computer vision tasks. However, GPGPUs are generally not available in lightweight mobile applications (i.e. lighter weight than a high-end laptop) because mobile embedded SoC GPUs have not yet adopted general purpose programming models. This has provided an opportunity for digital signal processors such as the TI C66x to become a new generation of power efficient coprocessor technology for lightweight mobile platforms. The C66x DSP is based on TI's Keystone architecture and consists of eight 1.25 GHz VLIW/SIMD processor cores. Each core contains two register files (side A and side B), each connected to four single precision floating point functional units (eight functional units total). Each core also contains two on-chip memories that can be programmatically configured as cache, scratchpad, or a mixture of both. Like other VLIW architectures, the compiler must software pipeline each innermost loop in order to efficiently schedule the functional units and registers. This paper describes a Lucas-Kanade implementation for this architecture and provides a close examination of its performance and efficiency as compared to competing processor technologies in both the HPC and embedded space.

II. Background

The optical flow computation is based on Equation 1. In this equation, pixel intensity is represented as a function of its position (x, y) and the time t in the video stream.

$$f(x,y,t) = f(x+\Delta x, y+\Delta y, t+\Delta t) \quad (1)$$

Assuming the pixel displacement is small between consecutive frames, the right side of the equation can be approximated by its first-order Taylor expansion.

$$f(x,y,t) = f(x,y,t) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y + \frac{\partial f}{\partial t}\Delta t \quad (2)$$

This yields Equation 3

$$\frac{\partial f}{\partial x}v_x + \frac{\partial f}{\partial y}v_y = -\frac{\partial f}{\partial t} \quad (3)$$

In Equation 3, the partial derivatives $\partial f/\partial x$, $\partial f/\partial y$ and $\partial f/\partial t$ can be computed from subtracting the adjacent pixel intensity from the images or by using convolution methods such as a Sobel filter.

Equation 3 has two unknown variables that cannot be solved if no additional condition is provided. To solve this problem, Lucas-Kanade method assumes that the flow field is spatially preservative, i.e. the pixels within a certain window are assumed to have the identical velocity value. Thus the equation can be solved by taking a neighbor window around the advised pixel. Assuming that the window includes n pixels

$q_1, q_2, ...q_n$, the linear equation system are listed in Equation 4.

$$\begin{gathered}\frac{\partial f}{\partial x}(q_1)v_x + \frac{\partial f}{\partial y}(q_1)v_y = -\frac{\partial f}{\partial t}(q_1) \\ \frac{\partial f}{\partial x}(q_2)v_x + \frac{\partial f}{\partial y}(q_2)v_y = -\frac{\partial f}{\partial t}(q_2) \\ ... \\ \frac{\partial f}{\partial x}(q_n)v_x + \frac{\partial f}{\partial y}(q_n)v_y = -\frac{\partial f}{\partial t}(q_n)\end{gathered} \tag{4}$$

This gives an over determined linear system that can be potentially solved by the least squares method, where the velocity value v_x and v_y of each pixel can be computed by Equation 7.

$$A = \begin{bmatrix} \frac{\partial f}{\partial x}(q_1) & \frac{\partial f}{\partial y}(q_1) \\ \frac{\partial f}{\partial x}(q_2) & \frac{\partial f}{\partial y}(q_2) \\ ... & \\ \frac{\partial f}{\partial x}(q_n) & \frac{\partial f}{\partial y}(q_n) \end{bmatrix} \tag{5}$$

$$b = \begin{pmatrix} -\frac{\partial f}{\partial t}(q_1) \\ -\frac{\partial f}{\partial t}(q_2) \\ ... \\ -\frac{\partial f}{\partial t}(q_n) \end{pmatrix} \tag{6}$$

$$(v_x, v_y)^T = (A^T A)^{-1} A^T b \tag{7}$$

In order to improve the accuracy, the least squares method is performed iteratively on each pixel multiple times. The accumulated flow v_x, v_y are used to offset the center of the neighbor window in the second image.

1) Given a pixel x,y and let $v_x = 0$, $v_y = 0$.
2) Pick the neighbor window w_1 in the image1 centered at x,y, the neighbor window w_2 in the image2 centered at pixel $x + v_x$, $y + v_y$.
3) Compute the velocity update v'_x, v'_y from the least squares method. If the magnitude of v' is less than the given threshold or the number of iteration reaches the maximum number, stop. Otherwise let $v_x = v_x + v'_x$, $v_y = v_y + v'_y$ and go to 2.

The least squares method assumes that the inter-frame movement is constrained to a maximum distance given by the neighbor window. To correct for this, a coarse-to-fine methodology that uses Gaussian pyramid is added to track larger movements. A Gaussian pyramid is a series of images which are weighted down using a Gaussian blur and a 1/2 down sampling. When this technique is used multiple times, it creates a stack of successively half-sized images, with each pixel containing a local average that corresponds to a pixel neighborhood on a lower level of the pyramid. Higher level of Gaussian pyramid contains less detail and noise and shows a more abstract view. Gaussian blur is a separable 2D convolution that can be implemented as two passes of 1D convolution on x and y direction.

Using the Gaussian pyramid, Lucas-Kanade method is able to discover larger movement by tracking them in a hierarchical scaled vision space. The movement of a pixel that goes outside the neighbor window on the higher level of the pyramid becomes smaller so that it may be traceable. Two Gaussian pyramids are generated from the two input images with an user specified level number, and the initial optical flow on the top level of the pyramids is set to be zero. The algorithm starts computing optical flow using the Lucas-Kanade method on the top level of the pyramids and uses the result as the initial value of the next level. Because the size of optical flow computed from level n is half of the size of level $n+1$, it must be bilinear interpolated by a factor of two in both height and width. This procedure repeats until all the levels of the pyramids have been processed.

The Pyramidal Lucas-Kanade method is shown in Algorithm 1.

Algorithm 1 Pseudocode of Pyramidal Lucas-Kanade Method

```
1: Input: Two input images $im1$ and $im2$, pyramid level $L$
2: Output: Optical flow field $f$.
3: Generate Gaussian pyramids for $im1$ and $im2$
4: Initialize flow field $f$ with zero values
5: for $i = L \to 2$ do
6:     Compute the optical flow $f_i$ on pyramid level $i$ using iterative Lucas-Kanade method with an initial guess=$f$
7:     2X bilinear interpolate $f_i$ in both height and width and store the result in $f$
8: end for
9: Compute the optical flow $f_1$ on pyramid level 1 using iterative Lucas-Kanade method with an initial guess=$f$
10: $f = f_1$
```

III. Algorithm and Implementation

We implement and accelerate the Lucas-Kanade method using platform-specific optimization. Those methodologies involve both instruction-level and thread-level parallelization of the bottleneck components of the pyramidal Lucas-Kanade method.

A. Gaussian Pyramid Generation

After applying kernel separation, the procedure of 2D Gaussian blur can be separated to a horizontal 1x7 Gaussian followed by a vertical 1x7 Gaussian. As shown in Figure 1, a 1x7 Gaussian blur involves 7 multiplications and 7 additions for each pixel that can be organized into the SIMD instructions QMPYSP (4-way SIMD multiplication) and DADDSP (2-way SIMD addition) so that the utilization of the computation units on the DSP can be maximized. The last operand of the Gaussian blur is set to be NULL since only 7 computations are needed.

In the vertical pass of the Gaussian blur the input pixels are strided across different rows. For this we manually unrolled the outer loop (column loop) by a factor of 2 so that the data from two consecutive iterations can be integrated into a 2-way SIMD load. In each loop iteration, a 7x2 (7 rows, 2 columns) sub-block is loaded into 7 register pairs. The 7 lower halves and higher halves of the register pairs are processed in the same way as we perform SIMD multiplication and addition on the horizontal Gaussian blur.

B. X and Y Direction Derivative Computation

Two of the derivative values d_x and d_y can be generated for each pyramid level of the first input image and reused during

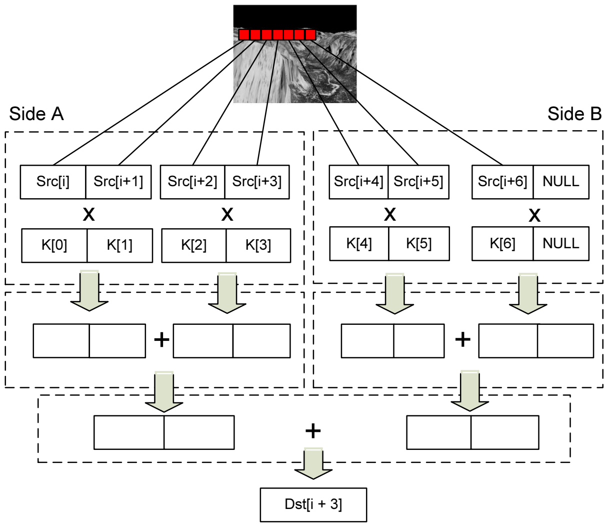

Fig. 1. Use of SIMD Instructions to Accelerate Gaussian Blur

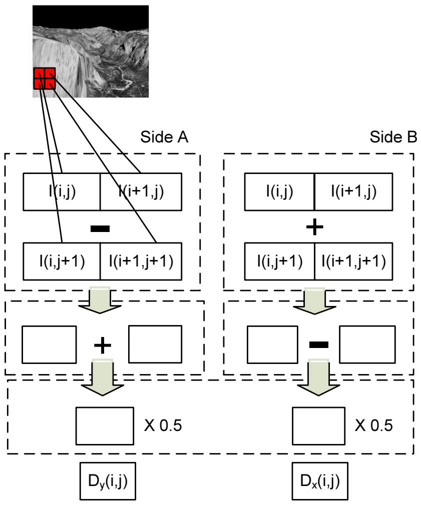

Fig. 2. Use of SIMD Instructions to Accelerate X and Y Derivative Computation

the entire procedure. Instead of storing d_x and d_y value into two matrices, we interleave them into a float2 vector type. This allows d_x and d_y values to be stored and loaded together and aligns the starting address of each pair of values with 64 bits so that aligned SIMD load and store can be utilized and achieve better memory performance. The optimization of derivative computation is implemented by applying SIMD addition and subtraction. An example of SIMD derivative computation is shown in Figure 2.

C. Least Squares Method

After the generation of the Gaussian pyramid and derivative matrices, the next step is to compute optical flow by least squares method on each level of the Gaussian pyramid. This part of the optimization consumes over 95% of the total execution time.

Algorithm 2 shows the most expensive component of the least squares method. It is a 2D loop that requires the computation of the summation of $\sum d_x^2$, $\sum d_x d_y$, $\sum d_y^2$, $\sum d_x d_t$, $\sum d_y d_t$ in the neighbor window. This computation is performed on each pixel on all the pyramid levels multiple times (iterative refinement). The d_x and d_y values are read from the partial derivative matrix, and d_t value is computed from subtraction of the corresponding pixel intensity $im2[i_2, j_2]$ and $im1[i_1, j_1]$. Without optimization, four floating point loads ($D[i_1, j_1].d_x$, $D[i_1, j_1].d_y$, $im2[i_2, j_2]$ and $im1[i_1, j_1]$) and five floating point multiplications and additions are required for processing each pixel.

Algorithm 2 Least Squares Method with Iterative Refinement

```
1: Input: Two input image im1 and im2, partial derivative matrix $D$, point ($x$, $y$) in im1, initial flow ($u_0$, $v_0$), neighbor window size $w$, iteration number $N$.
2: Output: Optical flow (u, v) for point (x, y).
3: $iter = 0, u = u_0, v = v_0$
4: for $iter = 0 \rightarrow N$ do
5:     $x_1 = x$, $y_1 = y$, $x_2 = x + u$, $y_2 = y + v$
6:     $a_{11} = 0, a_{12} = 0, a_{22} = 0, ab_1 = 0, ab_2 = 0$
7:     for $i_1 = y_1 - w \rightarrow y_1 + w$, $i_2 = y_2 - w \rightarrow y_2 + w$ do
8:         for $j_1 = x_1 - w \rightarrow x_1 + w$, $j_2 = x_2 - w \rightarrow x_2 + w$ do
9:             $d_x = D[i_1, j_1].d_x$, $d_y = D[i_1, j_1].d_y$
10:            $d_t = im2[i_2, j_2] - im1[i_1, j_1]$
11:            $a_{11} = a_{11} + d_x^2$
12:            $a_{12} = a_{12} + d_x \times d_y$
13:            $a_{22} = a_{22} + d_y^2$
14:            $ab_1 = ab_1 + d_x \times d_t$
15:            $ab_2 = ab_2 + d_y \times d_t$
16:        end for
17:    end for
18:    $det_a = a_{11}a_{22} - a_{12}^2$
19:    $ia_{11} = a_{22}/det_a$
20:    $ia_{12} = -a_{12}/det_a$
21:    $ia_{22} = a_{11}/det_a$
22:    $du = ia_{11}ab_1 + ia_{12}ab_2$
23:    $dv = ia_{12}ab_1 + ia_{22}ab_2$
24:    if $du, dv < threshold$ then
25:        break
26:    else
27:        $u = u + du$, $v = v + dv$
28:    end if
29: end for
```

Notice that the pixel intensity $im2[i_2, j_2]$ and $im1[i_1, j_1]$ are stored separately in two matrices, and $im2[i_2, j_2]$ is not a regular data access (the indices are computed from the previous flow update). In order to maximize the memory performance, we manually unroll the loop by a factor of two to enable SIMD load on the two input images. However since the starting address of $im2[i_2, j_2]$ is not predictable, we must use the unaligned SIMD loads for them. Correspondingly, two pair of d_x and d_y are read by aligned SIMD loads. On the C66, two aligned SIMD loads can be packed into a parallelized VLIW instruction and issued in one cycle, but when unaligned cannot be parallelized. Three cycles are required for each memory operation.

Our next optimization is to redesign the loop body for

SIMD arithmetic instructions. Algorithm 3 shows how complex multiply instructions can compute all five products in parallel. The product of d_x^2, d_y^2 and $d_x d_y$ can be computed using the complex number multiplication instruction CMPYSP. The CMPYSP instruction takes two register pairs ($src1\_o$, $src1\_e$) and ($src2\_o$, $src2\_e$), and stores the four results ($src1\_o * src2\_e$, $-src1\_o * src2\_o$, $src1\_e * src2\_o$, $src1\_e * src2\_e$) into a register quad. If both of the operands of CMPYSP are set to (d_x, d_y), we can retrieve the result of d_x^2, d_y^2 and $d_x d_y$ with a single instruction. The $d_x d_y$ value appears twice in the result but only one is needed. Since the loop is unrolled twice, we need to process two groups of d_x and d_y per iteration. A similar strategy is used to compute $d_x d_t$ and $d_y d_t$. Let the two consecutive d_t values be stored in s 2's array $d_t[2]$, the two groups of d_x and d_y values stored in $d_x d_y[2]$ and $d'_x d'_y[2]$. From line 13 and 14 of the algorithm we have $d_x d_t$ values stored in b_3 and c_4, $d_y d_t$ values stored in a_3 and d_4.

Algorithm 3 The Optimized Inner Loop of the Algorithm 2

```
1: **Input:** Two input image im1 and im2, partial derivative matrix $D$, point ($x$, $y$) in im1, initial flow ($u_0$, $v_0$), neighbor window size $w$, iteration number $N$.
2: **Output:** Optical flow (u, v) for point (x, y).
3: Perform line 3 to 6 of the Algorithm 2...
4: for $i_1 = y_1 - w \to y_1 + w$, $i_2 = y_2 - w \to y_2 + w$ do
5:     for $j_1 = x_1 - w \to x_1 + w$, $j_2 = x_2 - w \to x_2 + w$, $j_1 += 2$, $j_2 += 2$ do
6:         $d_x d_y[2] = D[i_1, j_1]$ (aligned)
7:         $d'_x d'_y[2] = D[i_1, j_1 + 1]$ (aligned)
8:         $p_0[2] = (im2[i_2, j_2], im2[i_2, j_2 + 1])$ (unaligned)
9:         $p_1[2] = (im1[i_2, j_2], im1[i_1, j_1 + 1])$ (unaligned)
10:        $d_t[2] = p_1[2] - p_0[2]$
11:        $(a_1, b_1, c_1, d_1) = CMPYSP(d_x d_y[2], d_x d_y[2])$
12:        $(a_2, b_2, c_2, d_2) = CMPYSP(d'_x d'_y[2], d'_x d'_y[2])$
13:        $(a_3, b_3, c_3, d_3) = CMPYSP(d_t[2], d_x d_y[2])$
14:        $(a_4, b_4, c_4, d_4) = CMPYSP(d_t[2], d'_x d'_y[2])$
15:        $a_{11} = a_{12} - b_1 - b_2$
16:        $a_{12} = a_{12} + c_1 + c_2$
17:        $a_{22} = a_{22} + d_1 + d_2$
18:        $ab_1 = ab_1 + b_3 + c_4$
19:        $ab_2 = ab_2 + a_3 + d_4$
20:    end for
21:    Handle the loop boundary condition...
22: end for
23: Perform line 18 to 29 of the Algorithm 2...
```

The size of the loop in Algorithm 3, line 5 is two times the windows size. However, the iterations of this innermost loop is software pipelined, so the iteration number must be large enough to offset so the prologue and epilogue overhead. We use a technique called loop flattening when the window size is smaller than 20. The loop flattening transforms the nested loops into a single loop to make the software pipeline longer.

Algorithm 4 shows the pseudo code of the loop flattening procedure. Instead of using i, and j as the loop variable, it updates the value of i and j inside the loop iteration.

Algorithm 4 Loop Flattening

```
1: **Input:** Two input image im1 and im2, partial derivative matrix $D$, point ($x$, $y$) in im1, initial flow ($u_0$, $v_0$), neighbor window size $w$, iteration number $N$.
2: **Output:** Optical flow (u, v) for point (x, y).
3: Perform line 3 to 6 of the Algorithm 2...
4: Initialize $i_1$, $j_1$, $i_2$, $j_2$
5: for $n = 0 \to 2w^2$ do
6:     Perform computation of $a_{11}$, $a_{12}$, $a_{22}$, $ab_1$, $ab_2$
7:     Update the value of $i_1$, $j_1$, $i_2$, $j_2$
8: end for
9: Perform line 18 to 29 of the Algorithm 2...
```

TABLE I. RUN TIME (SECONDS) OF PYRAMID OPTICAL FLOW COMPONENTS

time(ms)	win size =4	win size =8	win size = 16
Gaussian Blur	2.1	2.1	2.1
Derivative Computation	0.9	0.9	0.9
Bilinear Interpolation	0.7	0.7	0.7
Least Squares Method	179	414	956

D. Multicore Utilization

As shown in Table I, the least squares method occupies nearly all the running time. We swept the neighbor window size while setting the pyramid level = 4 and iteration number = 10. Since the running time of Gaussian blur, derivative computation and bilinear interpolation is negligible compared with the least squares method so they can be executed on single core without impacting performance.

The master core (core 0) initializes data structure, loads the images, and generates Gaussian pyramids and derivative matrices. Then, eight cores will begin performing the least squares method for the first level of the Gaussian pyramid. The workload is uniformly distributed along the rows. A synchronization is performed after each pyramid level is finished, then core 0 will interpolate the Optical Flow field onto the next level of the pyramid and then all the cores begin the least squares computation on the next level.

IV. RESULTS AND ANALYSIS

In this section we summarize the scalability, performance, and power efficiency results.

A. Experimental Setup

The accuracy of Pyramidal Lucas-Kanade method is dependent on the parameter selection: window size, iteration number, and number of pyramid levels. Marzat studied two error metrics, average angular error and norm error on and determined that an ideal parameter set is window size = 12, iteration number = 6 and pyramid level = 4 [9]. We use these values as our default parameters in our experiments.

We tested on an NVIDIA K20 GPU using the dense Pyramidal Lucas-Kanade implementation in OpenCV 2.4.9. Since OpenCV only provides a single-core sparse version of Pyramidal Lucas-Kanade implementation on its CPU-based implementation, we implemented our own multicore version of Pyramidal Lucas-Kanade method with C++ and OpenMP for the ARM Cortex-A9 and Intel i7-2600 CPU.

TABLE II. PERFORMANCE RESULT FROM DIFFERENT PLATFORMS

Platform/core#	C66x/8	CortexA9/2	Intel i7-2600/4	Tesla K20
Gflops	15.42	0.77	17.19	**108.61**
core power (W)	5.73	**4.85**	52.50	79.00
Gflops/W	**2.69**	0.16	0.33	1.37

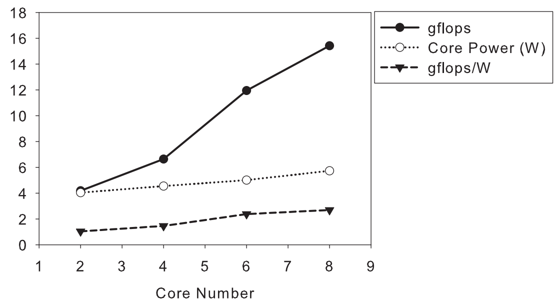

Fig. 3. Performance Study on Multicore

B. Performance and Power Efficiency Results

Table II lists the performance results of Pyramidal Lucas-Kanade method on different platforms. The power consumption of the DSP core is collected using the TI GPIO-USB module, which reads voltage and current directly from the onboard voltage regulator. For Intel CPU we use power results collected from Intel RAPL and for GPU we use the NVIDIA System Management Interface (nvidia-smi). The power consumption results of the Cortex-A9 processor are collected from a YOKOGAWA WT500 power analyzer.

Figure 3 shows the strong scalability of our DSP implementation relative to number of cores (each the test is performed on 1920x1080 video). The performance and power efficiency scales linearly, with eight cores providing 16 Gflops and 2.7 Gflops/W.

Figure 4 shows the system performance with different frame sizes. As shown in this figure, the floating point throughput is consistent except for 584x388 because the width is not aligned with the L2 cache line size (64 bytes), which increases the cache conflict miss rate.

Floating point throughput is not affected by the iteration number or the pyramid level as they only affect the outermost loop, but the window size controls the innermost loop and thus affects the relative overhead imposed by the prologue and epilogue of the software pipeline loop body. As shown in Figure 5, floating point throughput scales with window size, and loop flattening improves the performance when the window size is small.

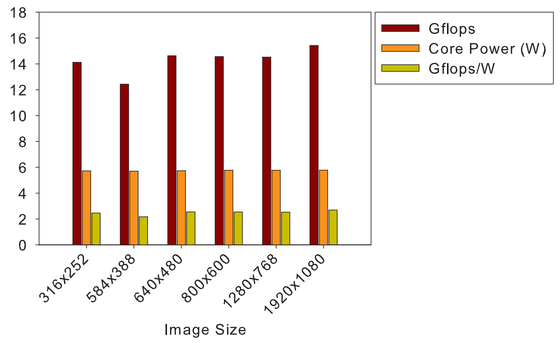

Fig. 4. Performance vs. Input Size

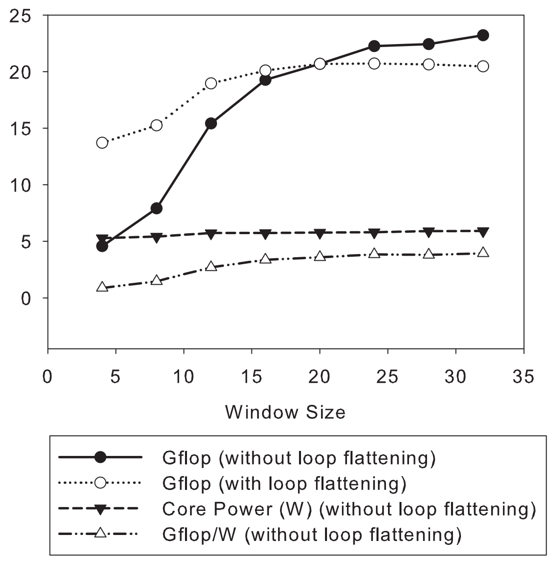

Fig. 5. Performance vs. Window Size

Let the image size be (m, n), pyramid level = p, iteration number = t, window size = w, The total number of flop operation per pixel (assuming we use separated 7x7 Gaussian) can be approximated by Equation 8. For example, if the window size = 12, pyramid level = 4, iteration number = 6. the flop per pixel is $76{+}10{\times}12{\times}12{\times}6{\times}(1{+}1/4{+}1/16{+}1/64) = 11551$. Figure 6 shows the number of C66x DSPs needed to reach a processing speed of 30 real time fps based on extrapolating our observed multicore scalability beyond eight cores.

$$\begin{aligned} &fpp = 76 + 10 \times w^2 \times t \times \textstyle\sum_{i=0}^{p-1}(1/4)^i \\ &pps = Gflops/fpp \times 10^9 \\ &fps = pps/(m \times n) \end{aligned} \tag{8}$$

C. DSP Performance Analysis

In this section we perform an analysis of the performance efficiency of the least squares method. Our calculations are based on 1920x1080 frame, pyramidal level = 4, iteration number = 5, and window size = 16. We collect the running time of the least squares method on single DSP core using a performance counter, which is 13.5×10^9 cycles.

The most inner loop of the least squares method (Algorithm 3, line 6 to 9) requires two aligned and two unaligned SIMD loads. Because the aligned SIMD load is parallelizable, together they require 3 cycles of latency. The total number of iterations of the most inner loop of the Algorithm 3 (line 5) can be computed by $16 \times 8 \times 5 \times (1920 - 8) \times (1080 - 8) \times (1 + 1/4 + 1/16 + 1/64) = 1.74 \times 10^9$. 16 and 8 are

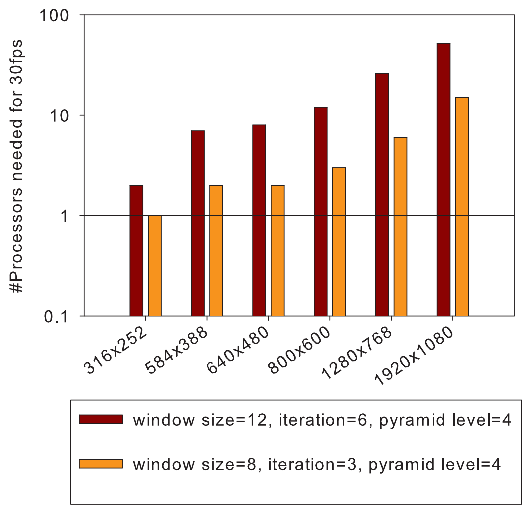

Fig. 6. Number of Cores Needed for Real-time 30 FPS

the trip count (cycles per iteration) of the nested loop and there are 5 iterations. Each level of the pyramid requires one-fourth the workload of the level below, so for example four levels requires $(1 + 1/4 + 1/16 + 1/64) = 1.3281$ of the workload of the first level. The actual cycles per iteration = $13.5 \times 10^9/(1.74 \times 10^9) = 7.75$. From the assembler output we know that after software pipeline optimization, each loop iteration (Algorithm 3, from line 5 to line 20) requires 3 cycles on average. The difference between this number and the actual cycles per iteration computed previously is the memory stall cycles, which is 7.75 - 3 = 4.75.

A C66x DSP core can execute two 2-way single precision SIMD loads per cycle. The peak memory bandwidth is reached when all the data read are from L1 cache with 1 cycle delay (the cores do not support out of order execution). This gives a maximum memory bandwidth of four single precision floating point values per cycle. However, since the memory access of the least squares method is composed of 50% of aligned loads and 50% of unaligned loads and the unaligned load is half of the bandwidth of aligned load (Algorithm 3, line 6 to 9), the maximum bandwidth of the memory operation is (4 + 2) / 2 = 3 single precision floating point values per cycle.

Each iteration of the loop on line 5 of Algorithm [**?**] performs 20 floating point operations and accesses 8 single precision floating point values. The compute-to-memory ratio is 20 / 8 = 2.5. Since the actual cycles for issuing memory operation per iteration is 3 and the memory stall cycles is 4.75, the memory achieves 3 / (4.75 + 3) = 38% of its maximum bandwidth. Combined together with the compute-to-memory ratio = 2.5 and maximum number of single precision floating point value reads per cycle is 3, we get the theoretical peak performance = $0.38 \times 3 \times 2.5 = 2.85$ flops/cycle. At 1 GHz this is 2.85 Gflops. We compute the total number of flop operations in the least squares method, which is $1.74 \times 10^9 \times 20 = 34.8 \times 10^9$. The actual Gflops is calculated by total number of flops divided by the actual running time in cycle, which is $34.8 \times 10^9/(13.5 \times 10^9) = 2.57$ Gflops. The utilization ratio is 2.57 / 2.85 = 90%. This value shows the efficiency of our implementation in terms of hardware utilization.

V. Related Work

Efficiently mapping Lucas-Kanade method has been studied for many years [3], [10]. Among those methods, Marzat et al. implement Pyramidal Lucas-Kanade method on Tesla C870 GPU workstation [9]. Their baseline algorithm is the same as the one targeted in this paper. Their performance test is performed with parameters set to image size = 640x480, iteration number = 3, neighbor window size = 10, pyramid level = 4 and are able to achieve an overall 68 ms per frame. We compute from their time performance results the performance of their implementation = 18.4 Gflops and power efficiency = 0.11 Gflop/W, which is 8% of our DPS implementation. Related work on hardware acceleration of other optical flow methods can be found in [11], [12], [13], [14].

VI. Acknowledgements

We would like to thank Arnon Friedmann, Murtaza Ali, and Alan Ward from Texas Instruments and Emily Teng from Advantech Corporation for their support of this work. This material is based upon work supported by Texas Instruments and the National Science Foundation under grant No. 0844951.

References

[1] S. S. Beauchemin and J. L. Barron, "The computation of optical flow," *ACM Computing Surveys*, vol. 27, pp. 433—466, 1995.

[2] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pp. 674–679, 1981.

[3] D. Javier and R. Eduardo, "Superpipelined high-performance optical-flow computation architecture," *Comput. Vis. Image Underst.*, vol. 112, no. 3, pp. 262–273, 2008.

[4] E. H. Adelson and J. R. Bergen, "Spatiotemporal energy models for the perception of motion," *Journ. Opt. Soc. Am.*, pp. 284–299, 1985.

[5] D. J. Fleet and A. D. Jepson, "Computation of component image velocity from local phase information," *IJCV*, pp. 3057–3079, 1995.

[6] B. Horn and B. Schunck, "Determine optical flow," *Artificial Intelligence*, vol. 17, pp. 185—203, 1981.

[7] R. Kories and G. Zimmerman, "A versatile method for the estimation of displacement vector fields from image sequences," *IEEE Proc. of Workshop on Motion-Representation and Analysis*, pp. 101—106, 1986.

[8] D. S. Kalivas and A. A. Sawchuk, "A region matching motion estimation algorithm," *CVGIP*, pp. 275–288, 1991.

[9] J. Marzat and Y. Dumortier, "Real-time dense and accurate parallel optical flow using cuda," *WSCG*, pp. 105–111, 2009.

[10] M. Anguita, J. Diaz, E. Ros, and F. Fernandez-Baldomero, "Optimization strategies for high-performance computing of optical-flow in general-purpose processors," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 19, no. 10, pp. 1475–1488, Oct 2009.

[11] Y. Mizukami and K. Tadamura, "Optical flow computation on compute unified device architecture," *International Conference on Image Analysis and Processing*, pp. 179–184, 2007.

[12] A. Bruhn and J. Weickert, "Variational optical flow computation in real time," *IEEE Transaction on Image Processing*, pp. 608–615, 2005.

[13] J. L. Martn and A. Zuloaga, "Hardware implementation of optical flow constraint equation using fpgas," *Computer Vision and Image Understanding*, vol. 98, no. 3, pp. 462—490, 2005.

[14] A. Plyer, G. Besnerais, and F. Champagnat, "Massively parallel lucas kanade optical flow for real-time video processing applications," *Journal of Real-Time Image Processing*, pp. 1–18, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11554-014-0423-0