PARALLEL FLUX TENSOR ANALYSIS FOR EFFICIENT MOVING OBJECT DETECTION

Kannappan Palaniappan[1], *Ilker Ersoy*[1], *Guna Seetharaman*[2],
Shelby R. Davis[3], *Praveen Kumar*[1], *Raghuveer M. Rao*[4], *Richard Linderman*[2]

[1]Dept. of Computer Science, University of Missouri, Columbia, MO 65211, USA
email:{palaniappank, ersoy}@missouri.edu
[2]Air Force Research Laboratory, Information Directorate, Rome NY 13441
email:{Gunasekaran.Seetharaman, Richard.Linderman}@rl.af.mil
[3]Black River Systems Company, Utica, New York 13502, davis@brsc.com
[4]Army Research Laboratory, Image Processing Branch, Adelphi, MD 20783

ABSTRACT

The flux tensor motion flow algorithm is a versatile computer vision technique for robustly detecting moving objects in cluttered scenes. The flux tensor calculation has a high computational workload consisting of 3-D spatiotemporal filtering operations combined with 3-D weighted integration operations for estimating local averages of the flux tensor matrix trace. In order to achieve efficient real-time processing of high bandwidth video streams a data parallel multicore algorithm was developed for the Cell Broadband Engine (Cell/B.E.) processor and evaluated in terms of the energy to computation efficiency compared to a fast sequential CPU implementation. Our multicore implementation is 12 to 40 times faster than the sequential version for HD video using a single PS-3 Cell/B.E. processor and is faster than realtime for a range of filter configurations and video frame sizes. We report on the power efficiency measured in terms of performance per watt for the Cell/B.E. implementation which is at least 50 times better than the sequential version.

Index Terms— Parallel image processing, multicore Sony Toshiba IBM Cell/B.E. processor, realtime video/image processing, 3D convolution, power efficient

1. INTRODUCTION

Realtime persistent moving object detection and target tracking for surveillance and situational awareness applications providing MOVINT information is a computationally challenging problem. Current trends in distributed sensor networks, agile systems and autonomous intelligent systems favor the processing of large volumes of raw data closer to the sensor to reduce network transmission requirements, extract high priority scene information more rapidly and exchange integrated information for cooperative downstream processing, cross-cueing sensors, and decentralized information fusion. Energy efficiency is an important system architecture requirement for embedded processing and optimizing data management requirements using distributed platforms. Multicore parallel processing environments are widely available today for which energy efficient versions of image and video processing algorithms need to be designed that can dynamically adjust the active number of processors to modify the completion time based on the energy budget. Scheduling algorithms to minimize total energy across identical parallel processors has been shown to be NP-hard even for unit-sized jobs [1].

In this paper we characterize the workload of the flux tensor algorithm for moving object detection in high bandwidth video streams. The parallel flux tensor algorithm exhibits super-linear speed-up due to the vectorization, loop unrolling, short vector fused multiply add operations and double-buffering optimizations for the Cell/B.E. architecture which along with the power efficiency of the Cell/B.E. processor provides a tremendous improvement in the performance per watt metric compared to an optimized sequential implementation. Power efficient real-time flux tensor processing is required in a variety of operational scenarios including video-based net-centric exploitation and tracking on airborne platforms [2] and ground-based multi-sensor imaging for force protection. Net-centricity provides improved agility, collaborative distributed sensing and layered sensor fusion. Such agile sensor networks need to be further enhanced to minimize overall power consumption under the constraint of still yielding the best exploitable information in a timely manner. Embedded video processing requires efficient algorithms in terms of power-aware computing as well as parallelization to enable real time performance in analyzing complex video [3, 4].

There are a number of challenging computer vision problems that need to be solved for stabilizing, detecting, extracting, verifying and tracking moving objects in airborne video [5–9]. In this paper we focus on one part of the video processing pipeline, namely power-efficient realtime moving object

978-0-9824438-2-8/11/$26.00 ©2011 IEEE

detection that is robust to natural environmental conditions such as illumination variation, shadows, clutter, and noise. In order to reliably detect moving blobs in unconstrained video, we use the recently proposed *flux tensor* (J_F) operator [10, 11], which captures the temporal variations of the optical flow field within the local 3D spatiotemporal volume. The flux tensor detects only the moving structures, and is less sensitive to illumination, focus and related problems compared to other moving object detection algorithms including classical background subtraction, mixture of Gaussians and 3D structure tensor orientation estimation. The flux tensor motion detection results have, in general, better spatial coherency enabling more accurate motion-based object segmentation. The flux tensor is more efficient in comparison to the 3D grayscale structure tensor since motion information is more directly incorporated in the flux calculation which is less expensive than eigenvalue decompositions at each pixel in the image [11,12].

This paper describes a parallel implementation of the flux tensor optimized for the multicore Cell/B.E. processor for real-time processing of high-bandwidth video streams in power constrained environments. Some early supercomputing architectures like the SIMD MasPar were ideally suited for image analysis tasks like deformable motion estimation [13]. The PS-3 Cell/B.E. processor provides a modern power efficient single chip high performance computational platform, with seven heterogenous cores - one Power Processing Element (PPE) and six (of eight) active Synergistic Processing Elements (SPEs). The PPE is a 64-bit processor that is binary-compliant with the PowerPC 970 but with a simpler architecture supporting dual issue, in-order execution. Each SPE consists of a 3.2 GHz Synergistic Processing Unit (SPU), a large 128-entry 128-bit vector register file, a small 256 Kbytes of private local store memory, short pipelines, and a memory-flow controller (MFC) to access the 256 MB of shared main memory using non-blocking DMA commands at 25.6 Gbytes/s. The SPUs are in-order dual-issue statically scheduled short-vector number crunchers with support for SIMD instructions operating on packed multiple data value without dynamic branch prediction. The PS-3 version of the Cell/B.E. processor is optimized for single-precision arithmetic (double-precision peak is less than 11 GFLOP/s) with truncation rounding. Each SPE can perform 25.6 GFLOP/s single-precision floating point operations at 3.2GHz. The Cell/B.E. supports both single program multiple data (SPMD) and multiple program multiple data (MPMD) parallel programming models that is more flexible than the single instruction multiple data (SIMD) model for mapping heterogeneous multithreaded data flow execution onto SPEs.

The Cell/B.E. offered one of the first commercial implementations of a power efficient high performance single chip multiprocessor with a significant number of general-purpose programmable cores targeting a broad set of workloads [14]. A good description of scientific computing and programming on the Cell/B.E. is provided in [15] and other details of implementing scientific computing kernels and programming memory hierarchies can be found in [16, 17]. In [18], the authors discuss interesting code transformation techniques for moving scientific simulation codes to the Cell/B.E. and [19] describes the fastest Fourier transform for the Cell/B.E. processor (18.6 GFLOP/s). Many programming frameworks/platforms like RapidMind [20], MFC (Multicore Framework) by Mercury [21] have also emerged to support efficient programming for multicore processors. In order to reduce complexities of task management, multithreading and synchronization for programming the Cell/B.E. some tools for mapping serial code in a semi-automatic fashion are in development [22]. We first give a brief overview of the flux tensor method and discuss the sequential implementation along with the computation and memory characteristics. Then we discuss the parallel architecture issues involved in our Cell/B.E. implementation. A description of the data partitioning scheme and parallelization procedures to map the flux tensor algorithm onto the Cell/B.E. cores is followed by experimental results of speed-up and power efficiency ratios.

2. FLUX TENSOR-BASED MOTION DETECTION

The 3D flux tensor was shown to be a robust and computationally efficient method for coherent detection of moving regions in video [10–12]. The flux tensor is a computationally more efficient operator in comparison to the 3D grayscale structure tensor [23, 24] since motion information is more directly incorporated in the flux calculation without the necessity for computing eigenvalue decompositions as with the 3D grayscale structure tensor. We summarize the mathematical description of the flux tensor multidimensional orientation estimation method to describe the types of operators needed to compute the flux tensor quantity for robust motion estimation.

In order to reliably detect moving structures *without* performing expensive eigenvalue decompositions, the *flux tensor* has been shown to be a more robust operator in comparison to the more widely used structure or orientation tensor [11, 12]. The flux tensor is composed of the temporal variations in the optical flow field within the local 3D spatiotemporal volume. Computing temporal derivative of the optical flow equation under a constant illumination model and setting the image brightness acceleration to zero gives,

$$\frac{\partial}{\partial t}\left(\frac{dI(\mathbf{x})}{dt}\right) = I_{xt}\, v_x + I_{yt}\, v_y + I_{tt}\, v_t, \qquad (1)$$

where $I(\mathbf{x})$ is the spatiotemporal image volume, t is time, $\mathbf{v}(\mathbf{x}) = [v_x, v_y, v_t]$ is the optic-flow vector at $\mathbf{x}$, and the second derivative terms are defined as,

$$I_{xt} = \frac{\partial^2 I(\mathbf{x})}{\partial x\, \partial t}, \;\; I_{yt} = \frac{\partial^2 I(\mathbf{x})}{\partial y\, \partial t}, \;\; I_{tt} = \frac{\partial^2 I(\mathbf{x})}{\partial t\, \partial t} \qquad (2)$$

The I_{xt} and I_{yt} terms capture information about moving edges or gradients in the video while I_{tt} incorporates information on moving textures and temporal illumination

changes. A constrained total least squares solution for the velocity field using Eq. 1 leads to the structure tensor matrix, $\mathbf{J_F}(\mathbf{x}, W(\mathbf{x}, \mathbf{y}))$, with matrix elements given by,

$$J_F^{pq}(\mathbf{x}, W) = \int_\Omega W(\mathbf{x}-\mathbf{y})\left(\frac{\partial^2 I(\mathbf{y})}{\partial x_p \partial t}\frac{\partial^2 I(\mathbf{y})}{\partial x_q \partial t}\right)d\mathbf{y} \quad (3)$$

where $W(\mathbf{x}, \mathbf{y})$ is a windowed integration kernel. We use the trace of the flux tensor matrix, referred to as Tr_J_F, that is defined below,

$$Tr_J_F = \int_\Omega W(\mathbf{x}-\mathbf{y})(I_{xt}^2(\mathbf{y}) + I_{yt}^2(\mathbf{y}) + I_{tt}^2(\mathbf{y}))d\mathbf{y} \quad (4)$$

as the computational operator to reliably detect moving regions in video streams. Each term in Eq. 4 incorporates information about temporal gradients which leads to efficient filtering of moving image features. A spatially invariant integration kernel $W(\mathbf{x}-\mathbf{y})$, for multidimensional isotropic local averaging is used for low power operation (instead of a more expensive spatially varying kernel) and is applied after the derivative filtering stages of computation in the flux tensor trace are completed. A robust statistics formulation can be incorporated in the flux tensor computation to further improve performance in low signal-to-noise conditions [25, 26].

2.1. Numerical Computation of the Flux Tensor

The calculation of the second derivative operators needed to compute the trace of the flux tensor matrix are implemented as convolutions with appropriate kernel filters. Although general 3D convolution kernels can be used, separable kernels are preferred as the 3D convolutions then can be decomposed into a cascade of 1D convolutions with a substantial reduction in computational cost from $O(n_k^3)$ to $O(n_k)$ for an $n_k \times n_k \times n_k$ sized filter. For numerical stability as well as noise reduction, a smoothing filter is applied along the third dimension that is not involved in the specific second derivative filter. The calculation of the first component of the trace, I_{xt}, uses derivative filters in the x- and t-dimensions and smoothing along the y-dimension, whereas calculation of I_{yt} uses smoothing along the x-dimension. The final component of the flux tensor matrix trace, I_{tt}, is the second derivative along the temporal direction and in this case the smoothing is applied along both spatial dimensions. The integral operator is also implemented numerically as an averaging filter decomposed into three 1D filters. The operation flow is illustrated in Figure 1. The data flow objects $I_{D_x S_y}$, $I_{S_x S_y}$ and $I_{S_x D_y}$ represent the intermediate spatial convolution results required to calculate I_{xt}, I_{yt} and I_{tt}. The operator modules shown in Figure 1, are the spatial smoothing filters S_x and S_y, the spatial derivative filters D_x and D_y, both in the x- and y-directions respectively, and the temporal derivative operators D_t and D_{tt}, representing first and second derivative filters in t respectively. The final averaging filters are the integral part of the flux tensor operator with A_x, A_y and A_t representing averaging filters in x-, y- and t-directions respectively. The data flow shown in Figure 1 reflects optimizations for a sequential implementation. Specifically, the summation block is being done prior to the spatiotemporal averaging operators for improved computational efficiency but at the expense of increased task dependencies and reduced parallelism (see Eq. 5 discussion). Exchanging the order of the sum and averaging filters will increase parallelism but would require more memory or additional computation. For the implementation shown in Figure 1, calculating the flux tensor trace for each video pixel requires eight 1D convolutions for the three spatiotemporal derivatives and three 1D convolutions for local averaging filters within the corresponding spatiotemporal cubes. The number of temporal filtering operations is reduced by saving intermediate results using additional memory.

The filter lengths or tap sizes associated with the three kernels for computing the flux tensor trace, $(n_{S_x}, n_{S_y}, n_{D_x}, n_{D_y}, n_{D_t}, n_{D_{tt}}, n_{A_x}, n_{A_y}, n_{A_t})$ are the full set of filter parameters that would need to be specified for a given application. Since we use spatially isotropic filters, we have a reduced set of parameters to specify, with $n_{D_x} = n_{D_y} = n_{D_s}, n_{S_x} = n_{S_y} = n_{S_s}, n_{A_x} = n_{A_y} = n_{A_s}$. Typically we use the same filter lengths for the first and second temporal derivative kernels $(= n_{D_t})$ and for the spatial smoothing and derivative kernels, *i.e.*, $n_{S_s} = n_{D_s}$. Thus, there remains four main parameters of the flux tensor; $(n_{D_s}, n_{D_t}, n_{A_s}, n_{A_t})$ which are the 1D filter sizes of the spatial derivative filter, the temporal derivative filter, the spatial averaging filter, and the temporal averaging filter respectively. In medium to close view (indoor) shots, the choice of $(5, 5, 5, 5)$ for filter sizes works well for detection. For the very far view sequences, where the objects may be quite small and moving very slowly, a $(3, 9, 3, 3)$ size works best. The large temporal filter size helps to catch the slow motion, the small spatial filter size helps to detect small motion and keeps the smoothing to a minimum.

2.2. Sequential Implementation of Flux Tensor Operator

The flux tensor implementation uses just the luminance component of the RGB video (1920×1080 pixels). In our earlier work [10], we described a reference sequential implementation that has minimum memory requirement and used just a single input image First In First Out (FIFO) buffer of size $(n_{D_t} + n_{A_t} - 1)$ for storing the input frames but at the cost of recomputing all spatiotemporal derivatives and integrals for each new video frame. This can be a significant penalty in terms of time and power since many intermediate filtering results that can be reused now have to be recomputed. A more efficient sequential implementation that minimizes redundant computations using one larger FIFO buffer of size $4 * (n_{D_t} + n_{A_t} - 1)$, for the intermediate spatial and temporal derivatives, and storing these intermediate results to be reused across temporal stages is described in [11]. Here, we discuss a new alternative approach that further improves memory efficiency by using dual FIFO buffers with a smaller memory

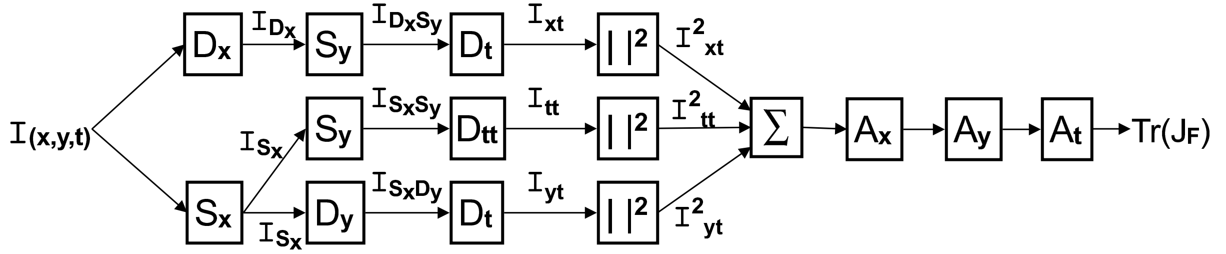

Fig. 1: Operator-centric data flow view of the various stages required to compute the flux tensor operator on a 3D spatiotemporal volume showing the task dependency relationships. Note that the magnitude squaring operators are explicitly shown. The summation stage is done prior to spatiotemporal averaging steps for memory efficiency at the cost of reduced parallelism.

footprint of $3 * n_{D_t} + n_{A_t}$ plus a few additional frames. Each frame of the input sequence is first convolved with spatial derivatives and smoothing filters. The intermediate results are stored as frames to be used in temporal convolutions, and pointers to these frames are stored in a FIFO buffer. The size of the first FIFO structure is of length n_{D_t} and for each input frame three spatial derivative frames $I_{D_x S_y}, I_{S_x D_y}$ and $I_{S_x S_y}$ are calculated and stored. Hence, the number of frames that need to be stored in the first FIFO structure is $3n_{D_t}$. Once n_{D_t} frames are processed and stored, the FIFO structure has enough frames for calculation of the temporal derivatives. Three frames of storage are needed to hold the temporal derivatives in memory for the current timestep.

Since averaging is distributive over addition for linear operators, the sum of squares $I_{xt}^2 + I_{yt}^2 + I_{tt}^2$, which is the trace of the flux tensor matrix is computed first, then spatial averaging is applied to this result and stored in a second FIFO structure of size n_{A_t}, to be used in the temporal part of averaging. The numerical expression that is being computed is,

$$Tr_J_F(\mathbf{x}) = \sum_{\mathbf{y} \in \mathcal{N}(x,y,t)} W(\mathbf{x} - \mathbf{y}) \big(I_{xt}^2(\mathbf{y}) + I_{yt}^2(\mathbf{y}) + I_{tt}^2(\mathbf{y}) \big) \quad (5)$$

where $\mathcal{N}$ is the local neighborhood over which the square of the second derivatives are summed. A weighted averaging filter, such as a Gaussian, can be used at the expense of additional computing cost. Typically box filters are used for a power efficient implementation. This temporal averaging FIFO keeps n_{A_t} frames at a time and produces the flux tensor trace after it is full. Once both FIFO's are full, processing a new input frame causes a shift of pointers in both FIFO's, reusing intermediate results from previous calculations and reducing the total computation per flux tensor output frame.

3. PARALLEL IMPLEMENTATION OF THE FLUX TENSOR OPERATOR ON THE CELL/B.E.

A power efficient multicore implementation needs to take full advantage of the intrinsic Cell/B.E. architecture specific hardware accelerations in order to use the best choice of data and task partitioning across SPEs, manage memory transfers, and take full advantage of vectorization and using local buffers. The 3.2 GHz SPEs deliver their peak performance while executing a fused short vector multiply add instruction (FMA) on each clock cycle that operates on a four floating-point element vector to complete eight floating point operations in SIMD fashion. Thus a peak peformance of 25.6 single-precision GFLOP/s per SPE can be obtained. An important point to note is that the SPEs only work on data staged in their local memory (local store). However, the SPE local storage is a limited resource as only 256 Kbytes is available for program, stack, local buffers and data structures. Making sure the SPEs efficiently receive and operate on current data without excessive buffering is critical to achieving high performance on the Cell/B.E. architecture. Rather than considering cache control and the impact of memory bandwidth, we focus on structuring data movement within the Cell/B.E. processor to keep the SPEs busy, and dividing the application into vectorized functions to make efficient use of the SPE hardware.

We implemented and tested our code on the SONY PS-3 which is a very energy efficient multicore processor with program access to six of the eight Cell/B.E. SPE processors and the main/external XDR memory on the PPE is limited to 256 MB of which about 200 MB is available to the Linux OS. To accommodate the required frame storage buffers and data structures for the parallel implementation, the data partitioning and grouping of the operations needed careful consideration. For main memory the parallel algorithm uses two FIFO buffers of size n_{D_t} and n_{A_t} for calculation of temporal derivatives and temporal averaging. In the sequential implementation, all the spatial derivatives are kept in the first FIFO buffer which requires $3n_{D_t}$ frames to be stored in main memory. Due to the limited shared (global) memory on the Cell/B.E. and taking advantage of the fast communication bus the parallel implementation of the flux tensor is memory efficient and stores only the input sequence of images in the first FIFO buffer of size n_{D_t} (instead of $3n_{D_t}$ for storing the spatial derivatives) and recalculates the spatiotemporal derivatives for each successive frame at the cost of extra calculations. We estimate the amount of redundant work to

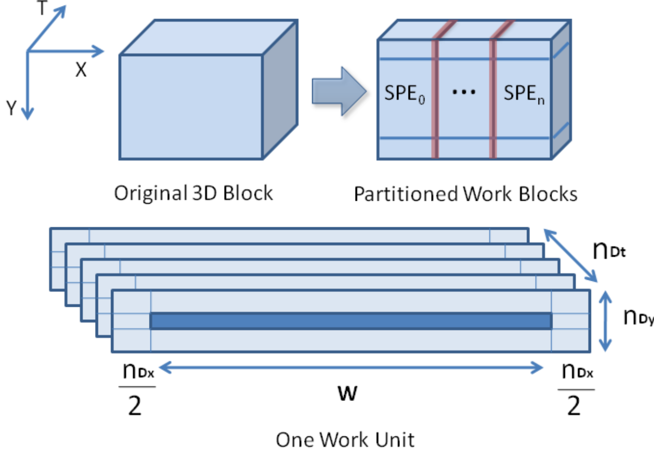

Fig. 2: Data partitioning scheme showing Work Unit (WU) block width in pixels on SPEs. The 3D block represents the spatiotemporal 3D grid of input data that needs to be processed to produce one flux tensor output frame. The 3D grid of data is chunked uniformly to distribute to each SPE. Since the partition is too large to fit into the limited SPE memory, each work block is further divided into smaller work units. The WU sizes are dependent on the filter sizes as labeled.

be between a factor of 2 and 5 depending on the filter sizes shown in Table 1, compared to the sequential version. There is no overlap in the work unit computation between adjacent SPEs, as shown by the vertical lines and faces marked in red in Figure 2 but there is an extra quad word data transfer (16 bytes) on left and right sides to provide pixel padding in the x-convolution direction. The second buffer FIFO2 operates the same as in the sequential case.

In order to parallelize across SPEs, the data needs to be partitioned into equal work blocks amongst different SPEs for optimal performance. Since there are convolutions in three dimensions (x, y, t), the whole work unit can be visualized as a 3D block. This is partitioned into as many overlapping blocks as there are number of active SPEs. The data is fetched and processed one row at a time. Due to finite size of the local store memory, each SPE may further subdivide the work block into smaller chunks and process a work unit width of WU columns each time. The data partitioning scheme and full work unit block is illustrated in Figure 2. The execution process on the PPE and SPE side is summarized in Algorithms 1 and 2 respectively. Optimized convolution operators are represented using the $\otimes$ symbol without the explicit loop unrolling and optimized FMA operations explicitly shown.

4. RESULTS AND DISCUSSION

The output of the flux tensor-based video object detection algorithm applied to a sample video sequence from the ARL Force Protection Surveillance System (FPSS) video collection [27] is shown in Figure 3. The first row shows color

Algorithm 1 Parallel Flux Tensor: PPE side

Input : Input Image sequence $I(x, y, t)$
Output : Flux Trace frame $Tr_J_F(x, y, t - \lfloor n_{D_t}/2 \rfloor - \lfloor n_{A_t}/2 \rfloor)$

1: **for** each time t **do**
2: Push($I(x, y, t)$, FIFO1)
3: Initialize number of intermediate flux frames, $N_m \leftarrow 0$
4: **if** FIFO1 contains n_{D_t} frames **then**
5: Partition data into blocks.
6: Put SPE control block information including work unit W and output location for intermediate flux F and final output Tr_J_F.
7: Set up SPE threads and wait for results F, Tr_J_F.
8: Push(F, FIFO2)
9: $N_m \leftarrow N_m + 1$
10: **if** $N_m > n_{A_t}$ **then**
11: Write output Tr_J_F
12: **end if**
13: **end if**
14: **end for**

Algorithm 2 Parallel Flux Tensor: SPE i

Input : Images in FIFO1, N_m, starting col C_i, and work block width W.
Output : Blocks of Intermediate Flux into FIFO2 and flux trace Tr_J_F

1: **for** each row r of Work Block **do**
2: Load from FIFO1, pixel data I_r from column $C_i - \lfloor n_{D_s}/2 \rfloor$ upto $C_i + W + \lfloor n_{D_s}/2 \rfloor$ into local store
3: Push($I_r \otimes S_x, I_{S_x}_buffer$);
Push($I_r \otimes D_x, I_{D_x}_buffer$);
4: **if** $I_{S_x}_buffer$ and $I_{D_x}_buffer$ have n_{D_s} rows **then**
5: $I_{S_x D_y} = I_{S_x} \otimes D_y$;
6: $I_{S_x S_y} = I_{S_x} \otimes S_y$;
$I_{D_x S_y} = I_{D_x} \otimes S_y$;
7: $I_{yt} = I_{S_x D_y} \otimes Dt$;
$I_{tt} = I_{S_x S_y} \otimes Dtt$;
$I_{xt} = I_{D_x S_y} \otimes Dt$;
$F_r = I_{xt}^2 + I_{yt}^2 + I_{tt}^2$;
8: Push(F_r, FIFO2);
Pop($I_{S_x}_buffer$);
Pop($I_{D_x}_buffer$);
9: **end if**
10: **end for**
11: **if** $N_m \geq n_{A_t}$ **then**
12: **for** each row r of Work Block **do**
13: Load from FIFO2, F_r data from column $C_i - \lfloor n_{A_s}/2 \rfloor$ upto $C_i + W + \lfloor n_{A_s}/2 \rfloor$ into local store
14: Push($F_r \otimes A_x$,$I_{A_x}_buffer$);
15: **if** $I_{A_x}_buffer$ has n_{A_s} rows **then**
16: $I_{A_x A_y} = I_{A_x} \otimes A_y$;
17: $Tr_J_F = I_{A_x A_y} \otimes A_t$;
Pop($I_{A_x}_buffer$);
18: **end if**
19: **end for**
20: **end if**

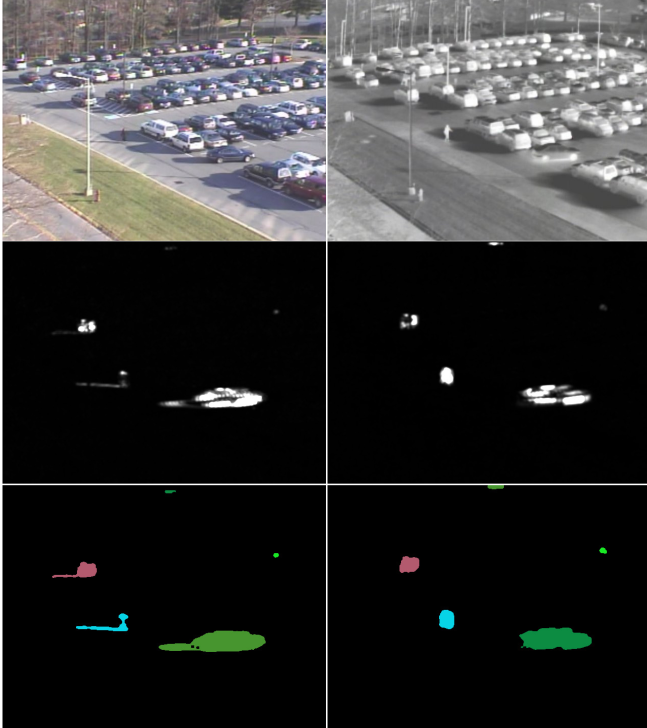

Fig. 3: Output of flux tensor motion estimation and blob extraction algorithm on selected frames of color visible and FLIR (forward looking long-wave infrared) data from the ARL FPSS dataset [27].

and long-wave infrared frames from the original video sequence. The second and third rows show the grayscale flux tensor response and the thresholded binary masks respectively using the flux tensor motion analysis with $(5, 5, 5, 5)$ filters, followed by grayscale closing (circular structuring element of radius 5) and using histogram based thresholding, adaptively switching between global Otsu and 80% cumulative histogram value. The colored blobs show the detected moving objects after post processing steps including morphological noise removal using area opening and connected component labeling to identify contiguous regions. The pink blobs are associated with two people walking in the far background. The FLIR channel is not affected by shadows and produces more compact blobs of moving objects suitable for tracking.

The sequential code was tested on a Dell PowerEdge 1850 server running CentOS Linux 5.4 using a single core of a dual CPU dual core Intel Xeon 2.8 GHz with 2 MB of cache per core, 4GB of memory and an 800MHz front side bus compiled using gcc -O3 version 4.1.2. The parallel code was tested on a PS3 Cell/B.E. with 6 SPEs using an appropriate SPE work unit for HD sized (1920×1080, WU=320 or smaller) and Standard Definition (SD) sized (640×480, WU=112 pixels) images. The PS-3 uses about 135 watts while the Dell PowerEdge 1850 uses about 550 watts for system operation including CPU, peripheral devices, operating system, multitasking, etc. Total system power was used to measure the performance to power efficiency ratios without doing detailed power measurements that can become complex to instrument and compare. The work unit size that can be accommodated by one SPE with 256 KB of local store depends on the size of the 3D convolution filters, especially the temporal filters, data alignment and partitioning requirements (usually multiples of 16 bytes). Parallel performance benchmarking done on an IBM QS20 and QS22 Blade servers with dual Cell/B.E.s both running Fedora Linux all compiled using gcc -O3 version 4.1.2 will be reported elsewhere.

Table 1: Speedup and performance to power ratios of the parallel multicore PS-3 Cell/B.E. implementation with 6 SPEs using 135 watts is compared to the optimized sequential implementation running on an Intel Xeon core in a Dell 1850 server using 550 watts. Sequential performance in frames per second are shown in the $(T_1^{\text{Seq}})^{-1}$ columns. Parallel PS-3 speed-up ($S_6^{\text{PS-3}}$) is compared to the sequential implementation running on an Intel Xeon CPU. The power efficiency improvement of the parallel implementation compared to the sequential implementation are shown in the *PPR* columns.

Filter Configuration				**HD video**			**SD video**		
n_{D_s}	n_{D_t}	n_{A_s}	n_{A_t}	$(T_1^{\text{Seq}})^{-1}$	$S_6^{\text{PS-3}}$	PPR	$(T_1^{\text{Seq}})^{-1}$	$S_6^{\text{PS-3}}$	PPR
3	3	3	3	1.75	40.1	164	17.32	18.2	74
5	3	5	3	1.68	38.6	157	14.35	20.0	82
7	3	7	3	1.54	38.1	155	13.02	19.3	79
9	3	9	3	1.37	39.6	161	11.52	19.6	80
3	5	3	5	1.53	30.8	125	13.99	14.7	60
5	5	5	5	1.42	29.6	120	12.28	14.9	61
7	5	7	5	1.34	28.7	117	10.99	15.2	62
9	5	9	5	1.23	28.3	115	10.05	14.8	60
3	7	3	7	1.34	25.6	104	12.01	13.5	55
5	7	5	7	1.25	24.7	101	10.44	13.2	54
7	7	7	7	1.18	23.6	96	9.88	11.9	48
9	**7**	**9**	**7**	**1.11**	**14.2**	**58**	9.03	12.0	49
3	9	3	9	1.24	22.0	89	10.45	12.0	49
5	9	5	9	1.14	21.3	87	9.34	11.3	46
7	**9**	**7**	**9**	**1.09**	**12.4**	**51**	8.63	11.1	45
9	**9**	**9**	**9**	**1.02**	**13.0**	**53**	7.97	10.7	43

We compared the performance between the sequential and parallel implementations of flux tensor for different filter configurations on two different frame sizes of video streams using 3D grids. Speed-up using p processors was calculated as, $S_p^{\text{PS-3}} = T_1^{\text{Seq}}/T_p^{\text{PS-3}}$ where T_p is the average time measured across p processors to complete the flux tensor computation for one frame and T_1^{Seq} is the time taken for the single core sequential implementation; $p = 6$ SPEs on the PS-3. The performance to power efficiency improvement ratio was calculated as, $PPR = S_p^{\text{PS-3}} \frac{P_{\text{Seq}}}{P_{\text{PS-3}}}$ where P_A is the system power used by architecture A and $S_p^{\text{PS-3}}$ is the speed-up ratio.

Table 1 shows the range of spatial and temporal sizes for both derivative and integral/averaging filters varying between

$3, 5, 7$ and 9 that were used for performance benchmarking. The sequential frame rate or inverse of the time to compute one frame on the Intel Xeon processor is given in column $(T_1^{Seq})^{-1}$ in frames per second, the speed-up measured on the PS-3 Cell/B.E. platform compared to the sequential performance is given in column $S_6^{\text{PS-3}}$. For the smallest derivative and integral filters of size 3 the speed-up of the parallel implementation compared to the sequential performance was more than a factor of 40 even though there are only six computational cores. The super-linear speed-up behavior is due to the use of extensive vectorization, loop unrolling and FMA operations to implement the flux tensor convolution kernels despite the additional work done by the SPEs recomputing intermediate results in the parallel implementation compared to the sequential version. As the filter sizes increase the speed-up gain decreases since the total volume of computations increases faster on the Cell (linearly with the size of the filter) since the parallel implementation needs to recompute all of the intermediate spatial derivatives, whereas the sequential implementation is able to store and reuse intermediate results at the cost of extra memory. For larger filter sizes the limited local store on the SPEs requires smaller data chunks (work unit width in Figure 2) that then requires more than six threads of execution and results in two stages of computation which reduces speed-up; the filter configurations needing two stages of execution are shown in bold font in Table 1. This can be partly mitigated by using the more expensive IBM Cell/B.E. Blade processors which have a lot more main memory but also higher power consumption.

The speed-up of the multicore flux tensor implementation ranged from a factor of 11 to 20 for the smaller SD video frame sizes to between 12 and 40 for the larger HD frame size video streams as shown in Table 1. The results for 16 different filter configurations for both HD and SD video frame sizes are compared and show substantial improvement in terms of both parallel speed-up as well as performance to power efficiency. Our implementation on the PS-3 Cell/B.E. was able to deliver 34.9 fr/s $\times 1.32$ GFLOPs/frame $= 46$ GFLOP/s which is 30% of single-precision peak performance (153.6 GFLOP/s) but significantly better than the expected memory-intensive peak of 12.8 GFLOP/s. The identical code reaches 39%, 35% and 24% of peak performance on the QS20 (410 GFLOP/s for 16 SPEs) with 6, 8 and 16 SPEs respectively for the same filter size configuration of $n_{D_s} = 9, n_{D_t} = 5, n_{A_s} = 9, n_{A_t} = 5$. An earlier implementation that used better data alignments but could only handle a limited number of filter sizes and image widths was able to reach 68 fr/s and 58% of peak on the QS20. We found that explicit memory management and some assembly coding on the Cell/B.E. is required to reach high performance even though this hand tuning incurs additional programming effort. Multicore GPU architectures are also well suited for computer vision algorithms [28]. In future work we will compare the power efficiency of the flux tensor on GPUs using CUDA or OpenCL.

5. CONCLUSIONS

The flux tensor operator estimates significant orientations in multidimensional gridded datasets and is an efficient technique for moving object detection in video datasets. The parallel algorithm implemented for the Cell/B.E. PS-3 architecture with six SPE computational cores achieved a speed-up improvement factor of 40 compared to the sequential algorithm using the smallest filter sizes which offers substantial energy efficiency optimization choices. The super-linear speed-up behavior is due to the extensive use of vectorized floating point operations, FMA instructions and double buffering to overlap computation and communication. Using larger filter sizes the speed-up gain decreased to a factor of 12 since the total volume of computations increases faster on the Cell/B.E. as the parallel implementation must recompute the intermediate spatial derivatives in comparison to the sequential implementation which uses significantly more memory. For larger filter sizes the limited local store on the SPEs also leads to smaller data partitioning sizes that requires more than six threads of execution which results in two stages of computation thus reducing speed-up. For all filter sizes tested the parallel flux tensor algorithm was able to exceed realtime performance requirements using a single PS-3 Cell/B.E. processor for SD sized video streams and for most of the filter sizes for HD sized video streams. The lower power requirements for the multicore PS-3 Cell/B.E. compared to an Intel Xeon processor makes the energy efficiency performance to power ratio of the flux tensor more than 160 times better for the smaller filter sizes and more than 50 times better for the larger filter sizes for processing HD video streams. The dependency of the energy efficiency factor on the flux tensor filter size provides an additional dimension for energy optimization based on output image quality, that is using slightly smaller filter sizes for a marginal reduction in performance.

6. ACKNOWLEDGMENTS

We thank Filiz Bunyak and Pieter Bellens (Barcelona Supercomputing Center) for discussions on timing and earlier drafts of the paper. This paper is based on work presented at the First Int. Workshop on Energy Efficient High-Performance Computing (EEHiPC10) at HiPC 2010. This research was partially supported by the Air Force Research Laboratory Visiting Faculty Research Program and AFOSR/ASEE Summer Faculty Fellowship Programs at Rome, NY, grants from the Air Force Research Laboratory under agreements FA8750-09-2-0198, FA8750-10-1-0182, from the Leonard Wood Institute (LWI 181223) in cooperation with the U.S. Army Research Laboratory (ARL) under Cooperative Agreement Number W911NF-07-2-0062 and by U.S. National Institute of Health award R33-EB00573. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of AFRL, ARL, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

7. REFERENCES

[1] S. Albers, F. Muller, and S. Schmelzer, "Speed scaling on parallel processors," in *Proc. 19^{t}h ACM Symposium on Parallelism in Algorithms and Architectures*, 2007, pp. 289–298.

[2] J. M. Metzler, M. H. Linderman, and L. M. Seversky, "N-CET: Network-centric exploitation and tracking," in *IEEE Military Communications Conf. (MILCOM)*, 2009, pp. 1–7.

[3] P. Kumar, K. Palaniappan, A. Mittal, and G. Seetharaman, "Parallel blob extraction using the multi-core Cell processor," *Lecture Notes in Computer Science (ACIVS)*, vol. 5807, pp. 320–332, 2009.

[4] S. Mehta, A. Misra, A. Singhal, P. Kumar, A. Mittal, and K. Palaniappan, "Parallel implementation of video surveillance algorithms on GPU architectures using CUDA," in *17th IEEE Int. Conf. Advanced Computing and Communications (ADCOM)*, 2009.

[5] Z. Yue, D. Guarino, and R. Chellappa, "Moving object verification in airborne video sequences," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 19, no. 1, pp. 77–89, Jan. 2009.

[6] A. Hafiane, K. Palaniappan, and G. Seetharaman, "UAV-video registration using block-based features," in *IEEE Int. Geoscience and Remote Sensing Symposium*, 2008, vol. II, pp. 1104–1107.

[7] S. Ali and M. Shah, "COCOA - Tracking in aerial imagery," in *SPIE Airborne Intelligence, Surveillance, Reconnaissance (ISR) Systems and Applications III*, Daniel J. Henry, Ed., 2006, number 6209 in Proceedings of the SPIE, p. Online.

[8] R.T. Collins, A.J. Lipton, H. Fujiyoshi, and T. Kanade, "Algorithms for cooperative multisensor surveillance," *Proc. of the IEEE*, vol. 89, pp. 1456–1477, October 2001.

[9] G. Seetharaman, G. Gasperas, and K. Palaniappan, "A piecewise affine model for image registration in 3-D motion analysis," in *IEEE Int. Conf. Image Processing*, 2000, pp. 561–564.

[10] F. Bunyak, K. Palaniappan, S.K. Nath, and G. Seetharaman, "Geodesic active contour based fusion of visible and infrared video for persistent object tracking," *IEEE Workshop Applications of Computer Vision (WACV 2007)*, p. Online, 2007.

[11] F. Bunyak, K. Palaniappan, S. K. Nath, and G. Seetharaman, "Flux tensor constrained geodesic active contours with sensor fusion for persistent object tracking," *J. Multimedia*, vol. 2, no. 4, pp. 20–33, August 2007.

[12] K. Palaniappan, I. Ersoy, and S. K. Nath, "Moving object segmentation using the flux tensor for biological video microscopy," *Lecture Notes in Computer Science (PCM)*, vol. 4810, pp. 483–493, 2007.

[13] K. Palaniappan, M. Faisal, C. Kambhamettu, and A. F. Hasler, "Implementation of an automatic semi-fluid motion analysis algorithm on a massively parallel computer," *10th IEEE Int. Parallel Processing Symp.*, pp. 864–872, 1996.

[14] M. Gschwind, "The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor," *International Journal of Parallel Programming*, vol. 35, no. 3, pp. 233–262, 2007.

[15] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca, "A rough guide to scientific computing on the Playstation 3," Tech. Rep. UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007.

[16] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. A. Yelick, "Scientific computing kernels on the Cell processor," *Int. J. Parallel Programming*, vol. 35, pp. 263–298, 2007.

[17] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J-Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy, p. online," *Proc. ACM/IEEE Conference Supercomputing*, 2006.

[18] P. R. Woodward, J. Jayaraj, P-H. Lin, and P-C. Yew, "Moving scientific codes to multicore microprocessor CPUs," *IEEE Computing in Science and Engineering*, vol. 10, no. 6, pp. 16–25, 2008.

[19] D. A. Bader and V. Agarwal, "FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine," *Lecture Notes in Computer Science (HiPC)*, vol. 4873, pp. 172–184, 2007.

[20] M. D. McCool, "Data-parallel programming on Cell BE and the GPU using the rapidmind development platform, p. online," *In GSPx Multicore Applications Conference*, 2006.

[21] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M-J. Prelle, "Multicore framework: An API for programming heterogeneous multicore processors," Tech. Rep., Mercury Computer Systems, Inc., 2006.

[22] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: making it easier to program the Cell broadband engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593–603, 2007.

[23] K. Palaniappan, H. S. Jiang, and T. I. Baskin, "Non-rigid motion estimation using the robust tensor method," in *IEEE CVPR Workshop on Articulated and Nonrigid Motion*, Washington DC, USA, June 27–July 2 2004, vol. 1, pp. 25–33.

[24] C. Weele, H. Jiang, and et al, "A new algorithm for computational image analysis of deformable motion at high spatial and temporal resolution applied to root growth," *Plant Physiology*, vol. 132, no. 3, pp. 1138–1148, July 2003.

[25] X. Zhuang, K. Palaniappan, and R. M. Haralick, "Highly robust statistical methods based on minimum-error bayesian classification," in *Visual Information Representation, Communication and Image Processing*, C. W. Chen and Ya-Qin Zhang, Eds., Optical Engineering, pp. 415–430. Marcel-Dekker, 1999.

[26] X. Zhuang, Y. Huang, K. Palaniappan, and Y. Zhao, "Gaussian mixture density modeling, decomposition and applications," *IEEE Trans. Image Processing*, vol. 5, no. 9, pp. 1293–1302, Sept. 1996.

[27] A. L. Chan, "A description on the second dataset of the U.S. Army Research Laboratory Force Protection Surveillance System," Tech. Rep. ARL-MR-0670, Army Research Laboratory, Adelphi, MD, 2007.

[28] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan, "GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction," in *5th IAPR Workshop on Pattern Recognition in Remote Sensing (ICPR)*, 2008, pp. 1–4.