International Journal of Computer Science and Applications, ©2009 Technomathematics Research Foundation
Vol. 6, No. 1, pp. 13 – 25

PARALLEL VIDEO PROCESSING PERFORMANCE EVALUATION ON THE IBM CELL BROADBAND ENGINE PROCESSOR

HASHIR KARIM KIDWAI

College of Information Technology, UAE University, P.O. BOX 17555
Alain, Abu-Dhabi, UAE
hkidwai@uaeu.ac.ae

TAMER RABIE

College of Information Technology, UAE University, P.O. BOX 17555
Alain, Abu-Dhabi, UAE
tamer@uaeu.ac.ae

FADI N. SIBAI

College of Information Technology, UAE University, P.O. BOX 17555
Alain, Abu-Dhabi, UAE
fadi.sibai@uaeu.ac.ae

The Cell Broadband Engine (BE) is a heterogeneous 9-core microprocessor which initially saw the light in the Sony PlayStation 3. This paper describes the parallelization of a video processing application on the Cell BE, and the programming model chosen for this application. Serial implementations on PPE only and parallel implementations on PPE-SPE with 8 SPEs are described. This is followed by the presentation of the speedup comparisons with and without DMA and thread creation overhead times. The results presented in this paper demonstrate that the Cell BE processor can achieve a speedup of 10x on this application and shows good scalability with number of SPEs. When the input data size is at least 512x512, we showed that the speedup becomes limited by the number of memory transfer.

Keywords: Cell multi-core computing, video processing, optical flow parallelization

1. Introduction

The field of computer architecture is a fast paced one that has witnessed major innovations and has taken dramatic turns in recent years. In the world of desktop computers dominated by x86-based processors, the super pipelined processor design approach that pushes the limits of performance by increasing the pipeline length has hit the power wall paving the way for multi-core and/or multithreaded architectures to invade this market segment. In the world of mobile computers, novel low power design techniques have been adopted in the processor, chipset, and system to maximize the battery life while keeping the performance at acceptable levels. Also, virtualization and security support are now visible in many product offerings. In the server space, scalable interconnects, multiple cores, increasing cache sizes, and RAS techniques have enhanced the performance scalability and availability of servers. In the embedded world, VLIW

designs with multiple execution units and innovative SoC designs based on standard ARM or MIPS embedded processor cores have emerged.

In the ultra fast pace of graphic processor designs, graphics processors (GPUs) are turned out at a faster rate than CPUs. Although running at lower frequencies than CPUs, but with multiple vertex and pixel shader (e.g. nVIDIA 7900 has 24 pipes x 1 multitexturing unit x 8 vertex shader processing units) and high memory bandwidth (nVIDIA 7900 has 256 bit memory bus at 51.2 GB/s memory bandwidth), GPUs have pushed the limits of real time cinematographic rendering and have outperformed their higher frequency CPU counterparts on common math tests such as matrix multiplication. For instance the Geforce4 with 128MB VRAM was shown to outperform by a factor of 6x a Pentium4 1.5Ghz machine on common math tests. A 3GHz Pentium 4 processor and a GeForce 5900 Ultra can execute 6 GFlops and 20 GFlops, respectively, making the GeForce 5900 Ultra roughly equivalent to a 10GHz Pentium 4. Conferences and University classes teaching the usage of GPUs for General Purpose (GP) computation (GP GPU) are held annually [http://www.gpgpu.org].

GP GPUs take advantage of the GPU architecture's stream model of computation. The IBM cell processor is another leading commercial processor based on the stream model of computation. The Stanford IMAGINE processor is based on a stream processor architecture developed by a team of researchers led by Professor William Dally. Multimedia and graphics applications and database queries are examples of applications which fit the stream model of computation. In general, applications with large data sets and applications amenable to vector processing will perform well under this model of computation.

The stream model of computation is based on the concepts of streams and kernels. A stream is a set of sequential data that require related operations to be performed on them. A stream is created by appending data elements to the tail of the stream. A stream is consumed by taking data elements from its head. Streams can be combined with other streams via operations which append or index into other streams. In the stream model of computation, whereas streams relate to the data, the kernels relate to the instructions or operations to be performed on the data. By definition, a kernel is a short program which takes one or more streams for input and generates an output stream as a result of the execution of the kernel instructions on the input streams' elements.

Stream processors perform extremely well on media, graphics, applications with large data sets requiring the execution of similar operation on their data elements such as vector processing applications because they trade large portions of CPU die area traditionally devoted to superscalar out-of-order scheduling and issue logic (register rename, reservation stations or scoreboard, and reorder buffer) and large on-chip centralized caches to massive numbers of SIMD execution units, large register files, and large memory units distributed among the processor's processing elements. On the late dual core desktop CPUs such as the Pentium 840 or the AMD Athlon X2 or FX60, large on-die caches (2MB L2) help alleviate the memory latency and the processor-memory performance gap that is still widening with time. Stream processors do not need large L2 caches as the stream model of computation takes advantage of the data locality property

of streams for which much smaller memory areas suffice, the output stream is used up (e.g. rendered) and rarely needs to be saved in caches for future reuse, and the data stream set is so large anyway that traditional L2 caches are not big enough nor efficient enough to hold it. Between the stream processor's execution units where one execution unit operates on a stream that becomes an input stream to another execution unit, caching of the stream is not needed in this consumer/producer model. Instead, memory units distributed among the processing elements in charge of SIMD/vector computation hold portions of the code and the portions of the data streams on which the code operates. Stream processors' processing elements typically access their local memories/storage areas for temporary storage and reuse of data, but rarely access main memory to get hold of sparse data. Without various levels of large caches and with local memory serving the computation needs of the processing elements, the L0 cache to L1 cache to L2 cache to main memory latency observed on traditional CPUs is avoided on stream architectures. In fact it is estimated that stream processors exhibit a ratio of memory to arithmetic operations that is 25x smaller than exhibited by scalar processors.

While significantly reducing access to main memory than traditional processors, stream processors exploit data parallelism (in their SIMD execution units), instruction level parallelism (in the superscalar pipeline of their processing elements or across pipelines in the various processing elements or in the host processor), and thread level parallelism. The latter one can be achieved by either: a. running 2 or more kernels simultaneously on 2 or more processing elements (throughput computing); or b. dividing a kernel among processing elements and running the various pieces of the kernel at the same time (parallel programming).

With its unique capabilities for accelerating applications requiring video, 3D graphics, for imaging, security, visualization, health care, surveillance, and more, Cell Broadband Engine (BE) technology is a promising step forward in the pursuit of real-time processing of highly sophisticated algorithms. Based on a high-bandwidth memory architecture and multicore technology, the Cell BE is a processor optimized for compute-intensive and broadband rich media applications, jointly developed by Sony Group, Toshiba, and IBM.

The Cell BE is a heterogeneous multi-core microprocessor whose objective is to provide high performance computation for graphics, imaging and visualization, and to a wide scope of data parallel applications. It is composed of one 64-bit PowerPC Processing Element (PPE) serving as host processor, eight specialized co-processors called Synergistic Processing Elements (SPE), and one internal high speed bus called Element Interconnect Bus (EIB) which links PPE and SPEs together. The host PPE supports the 64-bit PowerPC AS instruction set architecture, and the VMX (AltiVec) vector instruction set architecture [AltiVec™] to parallelize arithmetic operations. Each SPE consists of a Synergistic Processing Unit (SPU), and a Synergistic Memory Flow Controller (SMF) unit providing DMA, memory management, and bus operations. A SPE is a RISC processor with a 128-bit SIMD organization for single and double precision instructions. Each SPE contains a 256KB instruction and data local memory area, known as the local store (LS), which is visible to the PPE and can be addressed directly by

software. The LS does not operate like a superscalar CPU cache since it is neither transparent to software nor does it contain hardware structures that predict what data to load. The EIB is a circular bus made of two channels in opposite directions and allows for communication between the PPE and SPEs. The EIB also connects to the L2 cache, the memory controller, and external communication. The Cell BE can handle 10 simultaneous threads and over 128 outstanding memory requests.

The following sections describe the video processing application that we parallelized and developed for the Cell BE, and the programming model chosen for this application. Serial implementations on PPE only and parallel implementations on PPE-SPE with 8 SPEs are described. This is followed by the presentation of the speedup comparisons with and without DMA and thread creation overhead times.

2. Video Processing

Given a video sequence of time-varying images, points on the image appear to move because of the relative motion between the camera and objects in the scene [Gibson, 1979]. The instantaneous velocity vector field (magnitude and direction, shown in figure 1) of this apparent motion is usually called optical flow [Ballard and Brown, 1982].

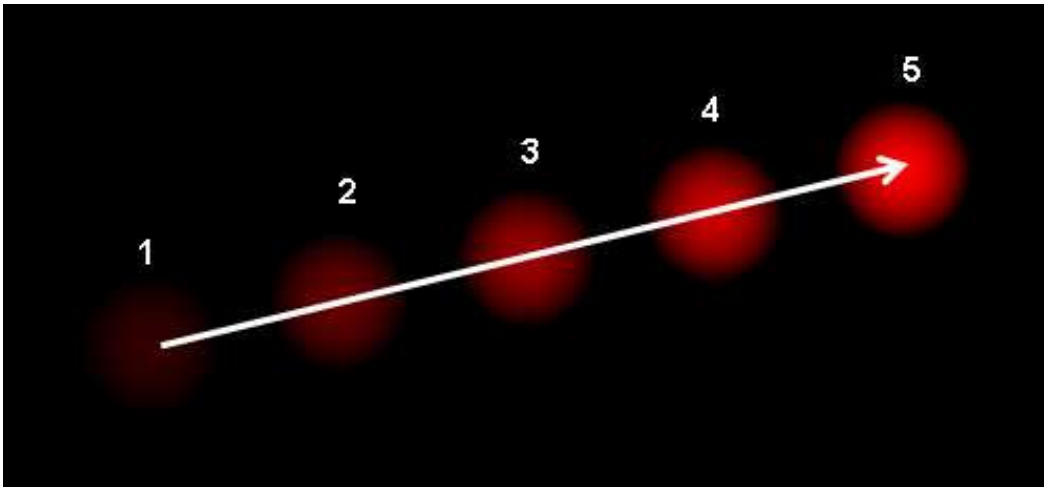

Fig.1. Example showing an optical flow motion vector of an object moving from frame 1 to frame 5.

Optical flow can provide important information about the spatial arrangement of objects viewed (refer to figure 2) and the rate of change of this arrangement [Horn, 1986]. Optical flow, once reliably estimated can be very useful in various computer vision applications. Discontinuities in the optical flow can be used in segmenting images into moving objects [Irani and Peleg, 1992]. Navigation using optical flow and estimation

of time-to-collision maps have been discussed in [Campani et al., 1995] and [Meyer and Bouthemy, 1992].

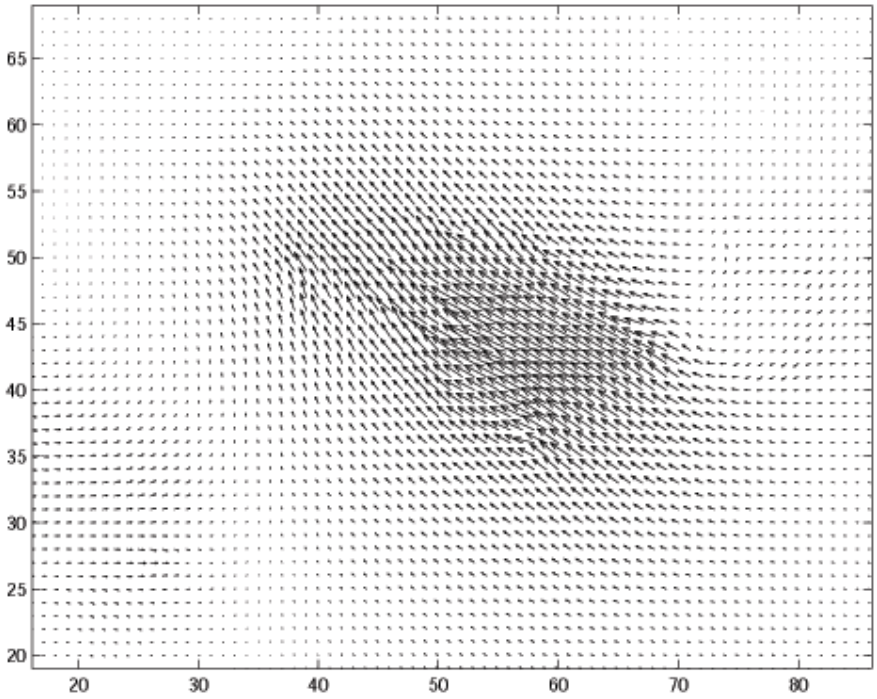

Fig.2. Sample Optical Flow Field from a taxi driving down a street. The larger the motion vector, the faster the pixel it corresponds to is moving between frames.

(Source: http://www.kfunigraz.ac.at/imawww/invcon/pictures/flowfield.gif)

The optical flow motion vectors are essential for accurate generation of MPEG4 video encoding. This is performed through the process of analyzing previous and future frames in the video sequence to identify fixed-size windows that were unchanged or have only changed location between frames. Motion vectors are then stored in place of these window locations in the frame. This is very compute-intensive and usually causes visual errors and artifacts when the motion vector estimation process is subject to inaccuracies [Richardson, 2004].

Optical flow estimation for each pixel of an image in a sequence of two consecutive images is achieved by computing the motion vector (u,v) between the current and previous pixel inside a window of size NxN, where N depends on the magnitude of the motion (e.g. N=5 for a 5x5 window neighborhood around the current pixel of interest). The motion vector is computed as a translational offset in the window's coordinate

system by a least squares minimization of the optical flow constraint equation between image frames at times (t) and (t-1) [Burt et al., 1989].

The optical flow constraint equation is given by [Horn, 1986]

$$uI_x + vI_y + I_t = 0, \tag{1}$$

where Ix , Iy , It is the spatiotemporal intensity gradient given as

$$\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}, \frac{\partial I}{\partial t} \tag{2}$$

Values of the motion vector (u,v) satisfying this constraint equation lie on a straight line in velocity space. The error function for each pixel

$$E(u,v) = \sum_{x,y \in N \times N} (uI_x + vI_y + I_t)^2 \tag{3}$$

is minimized by simultaneously solving the two equations

$$\frac{\partial E}{\partial v} = 0 \quad \text{and} \quad \frac{\partial E}{\partial u} = 0 \tag{4}$$

for the image motion vectors (u,v) of the current pixel at (x,y).

3. Algorithm parallelization and programming model

Our parallelization approach of the video processing algorithm is based on the data parallel model in which each SPE core performs the same computation on different parts of the data.

The video processing algorithm was first implemented on the simple single PPE architecture model and then recoded for the parallel PPE-SPE architecture model. In the parallel model, the PPE is responsible for SPE thread creation and I/O functions and SPEs perform the video processing computation. The computational part is uniformly distributed on all enabled SPEs. The number of SPEs therefore determines the number of times the SPE module gets replicated and executed. The PPE initiates the SPE threads (a total of 8 SPE threads in our experiment). The data is grouped and kept ready for distribution to 8 SPEs. Each SPE can directly access only its local store memory of 256 KB. The SPEs process the retrieved data from the main memory through DMA transfer and once finished, write back the data to the same memory location again via DMA transfer.

4. Implementation

We created two implementation sets i. A serial PowerPC/PPE-only implementation following the first above model; and ii. A parallel PPE-SPE (aka embedded) implementation following the second model.

4.1. *Overview*

In the serial PPE-only implementation, we input two consecutive images with sizes (128x128 and 128x128), (256x256 and 256x256) and (512x512 and 512x512). Once the data from both the input images was retrieved, we store it into multi-dimensional arrays. We then apply the optical flow algorithm, described in section 2, using 5x5 windows. We then calculate the execution time and write the processed data to the output vector files.

In the Embedded implementation, we again retrieved the data from the input files via the PPE code, and store it into multi-dimensional arrays. In all our implementations, PPE has to distribute unequal workload among different SPEs because of the nature of the application. On the SPE part we again have to verify which SPE is handling the job as there were different workloads for all of them.

Our parallel implementation is based on the pthread library which creates separate threads and allocates those threads to the context of every SPE, thus every thread is associated with a SPE and can completely run simultaneously. Also we are running our experiments on a single blade with a single Cell processor not requiring any message communication via MPI as would inter-blade communication require.

4.2. *Data Partitioning*

The data represents two consecutive images of a video sequence. We tested the images of different sizes as we have mentioned before in overview. For 128x128 input data, PPE was responsible for partitioning the data into 16x128 rows of the first image and 16x128 rows of the second image which makes a total of 16K worth of input data to process. Similarly for 256x256 matrices, the PPE code was partitioning the data into 32 rows and 256 columns of the first image and 32 rows and 256 columns of the second image which makes a total of 32*256*4= 32KB*2= 64kB blocks of data to process, similarly for 512x512, We distributed 64x512x4x2= 128K worth of data. We would also like to mention one additional thing and that is we in fact distributed more rows then we mentioned above because of the nature of the application. Every pixel which is currently processed is dependent on its neighboring pixels in all directions thus every SPE based on the algorithm has to have additional rows in both forward and backward direction to process the pixel under consideration correctly. Except for the first SPE, every SPE will be given the starting address of the previous two rows. For example in case of 128x128 matrix, SPE1 to SPE 7 will get the starting address of the 14th row rather than the 16th row and so on.

4.3. *Data Transfer*

Data was transferred from main memory to the SPEs' LS and vice versa. For efficient data transfer, the data has to be 128 Bytes aligned. A PPE starts a SPE module by creating a thread on the SPE by calling the spe_create_thread function. Some of the arguments to this function are the SPE program to load, and a pointer to a data structure. Since SPEs do not have any direct access to main memory, they can only access it through the Memory Flow Controller (MFC) DMA calls. The primary functions of the MFC are to connect the SPUs to the EIB and support DMA transfers between main storage and the LS.

The DMA calls once issued by the SPEs bring the data structure into LS. The maximum data size which can be transferred between PPE and SPE is 16KB per DMA call.

4.3. *Data Processing*

Once the data was moved from the main storage into the LS it was then processed through the Optical flow technique under 5x5 window which produced the two different vector files of the same sizes as the input images as discussed in the application section. As mentioned before that all SPEs except the first SPE due to the nature of the application were given different workload. For 128x128 implementation, although every SPE will process 16 rows but SPE1 to SPE7 will have to have extra rows present to make sure that every pixel under consideration process correctly. In case of 128x128 once the processing is done the output vector files representing the processed images were then transferred to PPE. Due to small input data sizes we did not see any size problem and thus techniques like double although tried didn't help us much. For 256x256 input data sizes we again processed the input data and vectors files then transferred to PPE. The output vector files were of sizes 32x256x4x2= 64K.

For 512x512, the input data size was of 128K and we follow the same processing technique as we employed in case of 128x128 and 256x256 then we end up having the two output files of size 128K and thus cross the local store limit. For this reason we had to tweak our code a lot and we also implemented double buffering technique. In case of 512x512, we processed the input data files and keep one output file size intact but we reduce the size of the other vector file to half i.e. ((64*512*4)/2) = 64K and thus we were able to accommodate our code and data in the local store at the same time. But the drawback was that this technique had a diminishing effect, since we were processing and transferring the second output data in the serial fashion which increases the algorithm execution time and thus results in a reduction of speedup.

4.4. *Video Processing Algorithm*

The following loop illustrates our optical flow estimation algorithm on every SPE

```
for (i = 0; i < rows_allocated_to_every_spe; i++) {
   for (j = 0; j < col; j++) {

     a = 0.0;
     b = 0.0;
     c = 0.0;
     d = 0.0;
     e = 0.0;

 // computes dense flow field within 5x5 window.
     for (m = -mm[l]; m <= mm[l]; m++) {
        for (n = -nn[l]; n <= nn[l]; n++) {
       x = ilimit(0, i+m, (int)row-2);
       y = ilimit(0, j+n, (int)col-2);

       Ix = matrix_1[x+1][y] - matrix_1[x][y];
       Iy = matrix_1[x][y+1] - matrix_1[x][y];
       It = matrix_2[x][y]   - matrix_1[x][y];

       a += Ix*Ix;
       b += Iy*Iy;
       c += Ix*Iy;
       d += -Ix*It;
       e += -Iy*It;
       }

   }
matrix_u[i][j] = (fabs(a*b - c*c) > mv) ? (a*e - c*d)/(a*b - c*c) : 0.0;
//horizontal motion component
 matrix_v[i][j] = (fabs(a*b - c*c) > mv) ? (b*d - c*e)/(a*b - c*c) : 0.0;
//vertical motion component

   } // j col.
} // i rows
```

5. Experiments

We ran our experiments on an IBM Cell machine with 1 Cell blade containing one 9-core Cell processor package. In order to obtain the more accurate timing information, we calculated the number of ticks on the serial PPE-only model through the mftb function which is available through PPU intrinsic header files along with gettimeofday function. In the embedded (PPE+SPE) version, we invoked spu_read_decrementer and spu_write_decrementer functions to get the total number of clock ticks and passed on that information to the PPE code by updating the counter. The reason for using mftb or spu_read_decrementer and spu_write_decrementer functions was to get more accurate

and reliable timing information as compared to the information provided by the gettimeofday function. On the Cell blades, the time base frequency is set to 14.318 MHz with a 3.2 GHz microprocessor. In order to get the execution time, we divided the number of clock ticks by the time base frequency.

We first calculated the time it took to run our Video Processing algorithm by varying the input data files sizes on the single PowerPC or PPE-only model as shown in figure 3. We then calculated the time it took to execute the same algorithm by our embedded model also shown in figure 3 by taking the following steps: a) Calculate the entire input data transfer time from PPE to SPEs, complete execution time of the entire data, and transfer time of the output data back to main memory by the SPEs to PPE; b) Calculate the execution time only, omitting the input and output data transfer times; and c) Calculate the execution time by varying the number of SPEs.

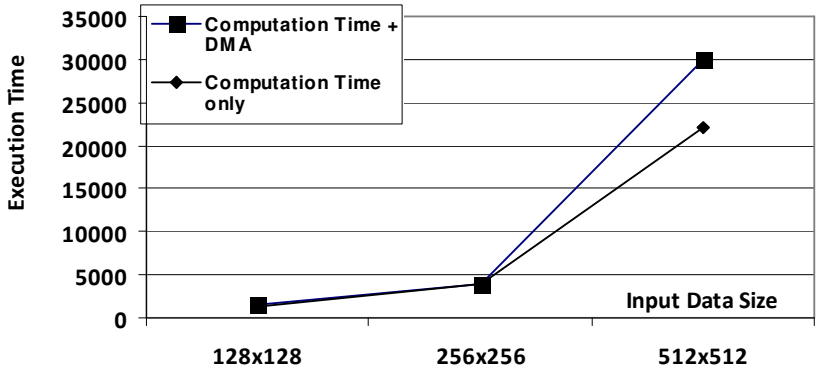

Fig.3. Execution time vs. input data size.

From figure 3 we noticed a constant increase in the execution time with the increasing input data size. We also compare the execution time of the algorithm with and without the DMA and find that this algorithm effects more with the DMA transfer in case of large data sets as compared to small data sets.

Table 1. Execution Time (in microseconds)

Data Size / Mode	128x128	256x256	512x512
Embedded (Comp. Time + DMA	1493.86	4022.15	29936.38
Embedded (Comp. Time only)	1400.0	3924.09	22106.22
PPE-only Application	5412.3	22745.64	91961.91

From Table 1 we notice that there is significant difference in the execution time in case of 512x512 input image frame matrices which validates our assumption that significant communication impacts this particular case. For 128x128 we did not see much communication impact. The difference in the execution time between DMA and non-DMA in case of 512x512 algorithm implementation can be reduced with the help of double buffering which is discussed in the conclusion section.

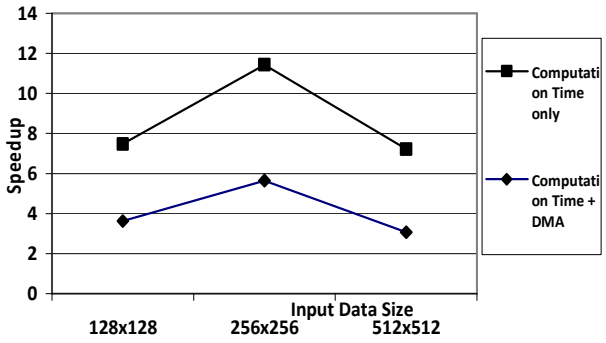

Fig.4. Plot of speedup vs. input data size.

We notice from the plot of speedup shown in figure 4 that this particular algorithm benefits most from multi-core computing when the input data size is 256x256. We also observe for the 512x512 case that this particular implementation is affected most with the additional memory interaction along with serial computation. The reason for the decline in speedup in the case of size 512x512 inputs is the DMA effect on the algorithm along with additional conditional branches which were introduced in the algorithm i.e. different if-else statements. For the 128x128 case, we did not notice much speedup based on the execution time differences shown in Table 1. Our assumption for 512x512 is that speedup can be achieved by employing a double buffering technique.

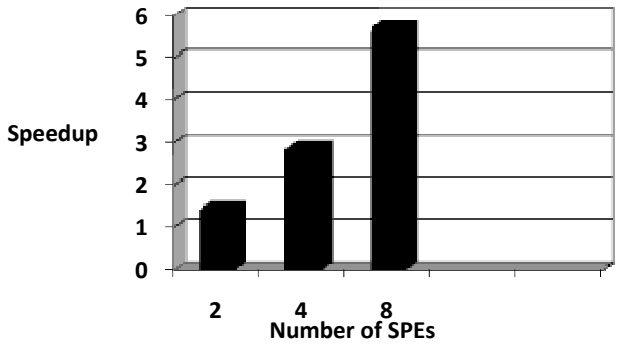

Fig.5. Plot of Speedup vs. the number of SPEs for input size of 256x256

We also analyzed the effect of varying SPEs on the overall speedup as shown in figure 5. Our results are based on input data size of 256x256 matrices. We observed an increase in the overall speedup with an increase in the number of SPEs, indicating good scalability.

6. Conclusion

The results presented in this paper demonstrate that the Cell BE processors can perform particularly well in cases where the application is compute intensive and not memory intensive. Speedups of 10x were observed on this application. Although for this particular algorithm, further speedup could be achieved by employing double buffering techniques, increasing the optical flow window from 5x5 to 11x11 based on the input data size and magnitude of the image motion makes the application more compute intensive. Also as the application becomes more memory intensive one observes a diminishing effect on the speeds. We will focus on building the 512x512 code with loop unrolling and double buffering techniques in our future work and comparing it with other compute intensive video processing algorithms.

Acknowledgments

We acknowledge generous hardware equipment, software, and service support from IBM Corporation as a result of a 2006 IBM Shared University Research award.

References

[AltiVec™] AltiVec™ Technology Programming Environments Manual, http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf

[Gibson, 1979] J. J. Gibson. The Ecological Approach to Visual Perception. Houghton Mifflin, Boston, MA, 1979.

[Ballard and Brown, 1982] D.H. Ballard and C.M. Brown. Computer Vision. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[Burt et al., 1989] P.J. Burt, J.R. Bergen, R. Hingorani, R. Kolczynski, W.A. Lee, A. Leung, J. Lubin, and H. Shvaytser. Object tracking with a moving camera: An application of dynamic motion analysis. Proc. IEEE Workshop Visual Motion, pages 2-12, March 1989.

[Horn, 1986] B.K.P. Horn. Robot Vision. MIT Press, Cambridge, MA, 1986.

[Irani and Peleg, 1992] M. Irani and S. Peleg. Image sequence Enhancemenmt using multiple motion analysis. In Proc. 11th IAPR, Int. Conf. on Pattern Recognition, pages 216-221, 1992.

[Campani et al., 1995] M. Campani, A. Giachetti, and V. Torre. Optic flow and autonomous navigation. Perception, 24:253-267, 1995.

[Meyer and Bouthemy, 1992] F. Meyer and P. Bouthemy. Estimation of time-to-collision maps from first order motion models and normal flows. In Proc. 11th IAPR, Int. Conf. on Pattern Recognition, pages 78-82, 1992.

[Richardson, 2004] Iain E.G. Richardson. H.264 and MPEG-4 Video Compression. John Wiley & Sons, England, 2004.