Accelerating Adaptive Background Subtraction with GPU and CBEA Architecture

Matthew Poremba
Pennsylvania State University
University Park, PA 16802
mrp5060@psu.edu

Yuan Xie
Pennsylvania State University
University Park, PA 16802
yuanxie@cse.psu.edu

Marilyn Wolf
Georgia Institute of Technology
Atlanta, GA 30332
wolf@ece.gatech.edu

ABSTRACT

Background subtraction is an important problem in computer vision and is a fundamental task for many applications. In the past, background subtraction has been limited by the amount of computing power available. The task was performed on small frames and, in the case of adaptive algorithms, with relatively small models to achieve real-time performance. With the introduction of multi- and many-core chip-multiprocessors (CMP), more computing resources are available to handle this important task. The advent of specialized CMP, such as NVIDIA's Compute Unified Device Architecture (CUDA) and IBM's Cell Broadband Engine Architecture (CBEA), provides new opportunities to accelerate real-time video applications. In this paper, we evaluate the acceleration of background subtraction with these two different chip-multiprocessor (CMP) architectures (CUDA and CBEA), such that larger image frames can be processed with more models while still achieving real-time performance. Our analysis results show impressive performance improvement over a baseline implementation that uses a multi-threaded dual-core CPU. Specifically, the CUDA implementation and CBEA implementation can achieve up to 17.82X and 2.77X improvement, respectively.

1. INTRODUCTION

Background subtraction is an important task in computer vision, and serves as a basis for many computer vision applications, such visual flow, object tracking, and surveillance [1] [2] [3]. This task, however, requires large amounts of computing resources in order to process large frames, with high frame-rates, and/or provide scene adaptivity. Since the application operates on pixels or groups of pixels, a large amount of data needs to be stored and moved in memory, while providing a high rate of processing.

Background subtraction, as well as other computer vision and real-time video processing tasks, has been largely limited by these amounts of computing power available since its advent. Over time, the complexity of these applications has increased in parallel with the amount of computing power available. The rise of general purpose chip-multiprocessors (CMP) and many-core graphics processing units (GPU) has provided new opportunities to accelerate the performance of such real-time video applications. The introduction of multi- and many-core chip-multiprocessors (CMP), such as Intel Core2 Duo architecture, can help the application acceleration by paralleling tasks. The improvement, however, could be limited due to the fact that such CMP architectures are for general-purpose computing.

The advent of specialized many-core CMPs, such as NVIDIA's *Compute Unified Device Architecture (CUDA)* framework [6] as well as IBM's *Cell Broadband Engine Architecture (CBEA)* [7], provide new opportunities to accelerate the performance of video applications. Compared to general-purpose CMP architecture, CUDA provides advantages over other architectures due to the large amount of memory that can be transferred back and forth at a high rate of speed, and the large amount of memory available locally to the GPU cores. The CBEA provides a multi-core architecture with novel methods of solving problems being faced by conventional CPUs.

The rest of this paper discusses various research already completed on techniques related to background subtraction. A variety of methods for solving this problem are presented in section 2. A description of the algorithm, architecture backgrounds, and architecture specific program flows are discussed in section 3. The test results of the chosen two implementations are presented in section 4, as well as descriptions of tweaks preformed for this specific application to obtain the best results. Finally, some notes on future work are given in section 5 and a conclusion is given in section 6.

2. BACKGROUND AND RELATED WORK

Background subtraction is used to identify objects as being in either the foreground or the background. As the name implies, background subtraction removes the background leaving only the foreground objects. An example of this is shown in Figure 1. With this information the data may be used in conjunction with other applications in image processing and computer vision including object tracking, visual flow, and surveillance. Several techniques may be employed to determine the background of an image. These techniques are described in detail in this section.

The most basic technique of performing background subtraction is by measuring the Euclidean distance of the current frame's pixel with the same coordinate pixel in the background model, and removing the pixel if it is below a certain threshold range. The technique is shared by both adaptive and non-adaptive background subtraction algorithms. The difference between the two techniques is how the background is modeled.

In *non-adaptive modeling*, a single background model reference is used for each pixel. This background model can be created by various means, as shown in the related work below. Non-adaptive modeling has the advantage of being simple to understand, simple to implement, and it uses relatively low computing resources. *Adaptive modeling*, on the other hand, uses multiple gaussian models to improve the accuracy. Adaptive modeling is more difficult to understand and implement, and the computing resources increase linearly with the amount of gaussian models desired with respect to non-adaptive modeling. Adaptive modeling, how-

This work was supported in part by NSF 0905365, 0903432, 0643902.

Figure 1: Example of a Background Subtracted Image

ever, provides the best results, as it can adjust itself to environmental changes, and is thus more robust and useful for a wider range of applications such as outdoor scenes.

Non-Adaptive Background Subtraction. Horprasert et al. proposed a *non-adaptive* approach for background subtraction [1]. The approach uses a 3-space cube with each axis representing a color (R, G or B). The mean and standard deviation are taken for each pixel in this so-called RGB space. The difference distances in expected color and brightness are determined by finding the orthogonal projection and its perpendicular component in the RGB space. A geometric mean of each of the distances is calculated and used to normalize the difference distances so that a single threshold value can be used for all pixels. The approach uses both of these values to differentiate between background, illuminated background, background in shadows, and foreground. Using this approach, a background model must be created at the beginning of the algorithm, using the first N frames, which are assumed to not contain any foreground objects. This approach has a certain disadvantages, since it is not adaptive. For example, without the use of multiple models, the approach will not be able to adjust itself for objects consistently moving, such as swaying trees or fountains.

Simplified Adaptive with Mixture Models. Schlessman et al. [2] proposed an adaptive approach using a weighed gaussian mixture model. This paper discusses the design of a multi-processor system on chip (MPSoC), with a gaussian mixture model. The background model of this paper is based on that of [1], but in Y, Cb, Cr colors channels. For each gaussian model in the mixture model, the mean and standard deviation of the pixel are stored. The difference between the current pixel and the background model pixel is determined by the number of standard deviations the current pixel is from the background model. All the differences are divided by a threshold value, summed, and tested against an overall threshold. This can be seen as an ellipsoid in Y, Cb, Cr space, with pixels inside the ellipsoid being background, and pixels outside being foreground. This approach differs from the full approach of using the complete probability density function, since the goal is ease of implementation of the MPSoC architecture.

Full Adaptive with Mixture Models. Several papers [3] [4] [5] proposed a complete gaussian mixture model, determining if a pixel is in the foreground or background using the probability density function, the calculated means and standard deviations. A good mathematical description of the process can be found in [3]. Even in these works, the full probability density function is often not used, as it can introduce computational complexity [5]. The previous paper [2] can be seen as further reduction of the probability density

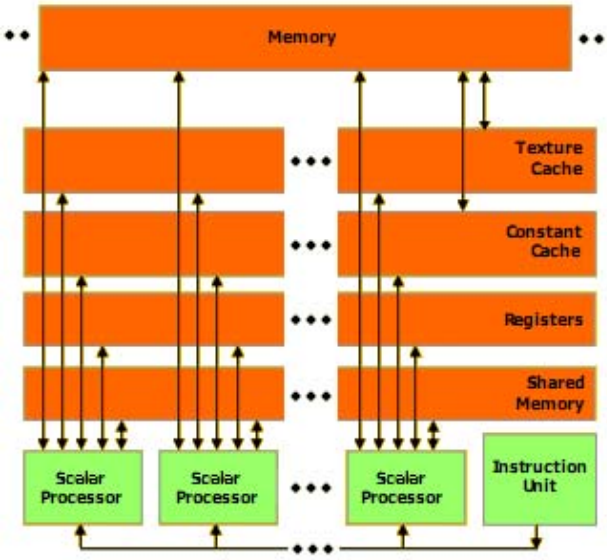

Figure 2: The Conceptual View of the CUDA Architecture: A Streaming Multiprocessor

function, while still providing the ability to match gaussians, so this algorithm is used as a basis for the algorithm in this paper.

3. ALGORITHM IMPLEMENTATION

This section discusses writing the adaptive background subtraction algorithm in CUDA and on the Cell Broadband Engine. The section first gives background information about the CUDA and CBEA architectures, followed by background information on the test algorithm. Next a complete description of the algorithm is given, and finally program flow for both architectures is discussed.

3.1 CUDA Background

CUDA [6] is NVIDIA's general purpose parallel computing architecture enabling developers to write massively parallel code to run on desktop systems with NVIDIA GPUs. The CUDA SDK provides extensions to the C programming language allowing code to be easily produced by programmers already familiar with the C language.

Figure 2 shows the conceptual view of the Nvidia GPU with CUDA architecture. The GPU many-core architecture organizes hundreds of cores into streaming multiprocessors. The architecture can therefore execute hundreds of threads and manage thread counts in the thousands. Hardware support is included for thread scheduling to provide zero-overhead thread switching and therefore provides very fine-grained granularity parallelism. The so-called single-instruction, multiple-thread (SIMT) architecture makes this possible.

3.2 CBEA Background

The Cell Broadband Engine Architecture is a collective term for Sony, Toshiba and IBM's Cell Processor consisting of a complex PowerPC based core and several simple Synergistic Processing Elements (SPEs) connected with a high-bandwidth element interconnect bus (EIB). A graphical view is depicted in figure 3. CBEA attempts to alle-

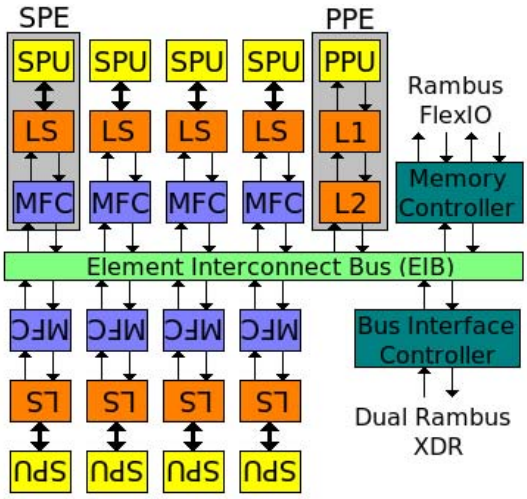

Figure 3: Graphical Depiction of CBEA

viate many of the problems faced by processors today such as power, clock speed, and the memory wall [7]. The cell processor allows for more general types of parallelism than CUDA and has already been shown to have promise in the supercomputing world [8].

The architecture attempts to solve the memory wall problem by using a local store (LS) for each SPE. Data is transferred asynchronously between the main memory and local stores, all controlled by software. In a sense, the programmer is defining what data will be processed and controls the cache contents.

An SPE processor uses a new instruction set optimized for power and performance with SIMD capabilities. Branch prediction is controlled by software hints provided by the programmer or compiler. Each SPE has its own memory flow controller (MFC) to control and queue DMA transfers. Since DMA transfers can be queued and are asynchronous, transfers can be, and often are, double- or multi-buffered for performance. The MFC controller is connected to a ring network connecting all of the SPEs, the PPE and IO, which includes main memory.

IBM provides an SDK package for the CELL, running exclusively on linux at the time of writing, with libraries for the C programming language and extensions to the make tool chain. Running of programs on the CBEA as well as scheduling and context switching are controlled by the operating system.

3.3 Algorithm Description

Due to the large amount of main memory available in all three test architectures, the number of gaussian models can be increased as necessary. Consequently, the adaptive background subtraction algorithm can be employed. In the case of real-time processing, increasing the number of gaussian models decreases the size a frame may be, but provides better response to scene changes. Too many gaussian models, however, may be considered a waste, as they may be very rarely used [4]. If a modest number of gaussian models are chosen, around 3 to 5, the next metric of interest is the size of the frames that can be processed in real-time.

The background subtraction approach is based on that of [2]. Background pixels are modeled using a pre-defined number of gaussians G. For each gaussian model, a mean and standard deviation are stored in memory, as well as weights and the number of times the gaussian has been seen.

Gaussian matches are tested using a single ellipsoid with half-axis lengths in units of the current pixel's standard deviation and center at the current pixel's mean, where σ is standard deviation, μ is the mean and X is the current pixel value. The a-values are thresholds in standard deviations.

$$(\frac{\alpha_y}{a_y})^2 + (\frac{\alpha_{cb}}{a_{cb}})^2 + (\frac{\alpha_{cr}}{a_{cr}})^2 < 1 \tag{1}$$

$$\alpha_x = \sqrt{\frac{|X - \mu_x|}{\sigma_x}} \tag{2}$$

Matched gaussians are updated to include the current pixel in the model. This means the mean and standard deviation values are updated iteratively. The number of times the pixel has been seen, N is also updated.

$$\mu_{x_n} = \mu_{x_{n-1}} + \frac{X_n - \mu_{X_{n-1}}}{N} \tag{3}$$

$$\sigma_{X_n} = \sigma_{X_n} + \frac{(X_n - \mu_{X_{n-1}})(X_n - \mu_{X_n}) - \sigma_{X_{n-1}}}{N} \tag{4}$$

$$N = \begin{cases} n, & \text{if } n < N_{max} \\ N_{max}, & \text{if } n \geq N_{max} \end{cases} \tag{5}$$

After each pixel is tested, all gaussian weights must be updated to determine the usefulness of the model. A model that is not seen is will have its weight lowered, while a model that is seen will have its weight raised. The variable ρ is set if the model is matched in the current frame.

$$M = \begin{cases} m, & \text{if } m < M_{max} \\ M_{max}, & \text{if } m \geq M_{max} \end{cases} \tag{6}$$

$$\omega_m = \omega_{m-1} + \frac{\rho - \omega_{m-1}}{M} \tag{7}$$

3.4 Program Flow

To use the algorithm described, the RGB colors are first converted to YCbCr channels. Next, for each of the predefined number of gaussians, the pixel is tested whether or not it is in the ellipsoid defined by the formulas (1) and (2), where μ is the mean and σ is the standard deviation of the color channel of the current pixel.

If a gaussian match is found, the pixel is marked as a background pixel. In the case of background subtraction, it is removed. The gaussian models of matched pixels must also be updated. Upon a gaussian match, the number of times the gaussian has been seen is incremented as per equation (5). New mean and standard deviation values are also calculated using equation (3) and (4).

If no match is found, a new gaussian model must be created. If there are no gaussians remaining unused, the gaussian model with the lowest weight is replaced. A newly created gaussian is modeled using the current pixel's value as the mean, a high variance, and a low weight.

After all of the gaussian models are tested, the weights on all G gaussians are updated using formulas (6) and (7), where ρ is 1 if a gaussian match was found and 0 if no match was found.

This same algorithm is used for all three test architectures. Slight changes are made for each for architecture-specific

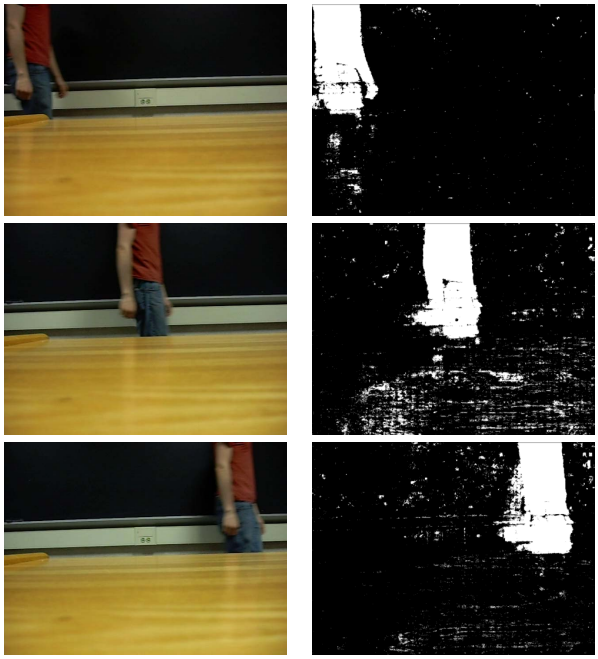

Figure 4: Frames Subtracted with Test Algorithm

control and partitioning of the pixel data to be processed. For testing the architectures, the frames of a pre-recorded video were converted into a series of bitmap images. The bitmap data is read in as raw RGB data. In all three architectures, a separate I/O thread is created to perform disk reads and writes with two buffers. The I/O thread reads the entire image frame into a buffer, signals to the main thread the read is complete, and reads into the next buffer. Once the processing is complete, the data is written back to the first buffer. Once all data is written to the buffer, the entire buffer is flushed to disk, and the next frame is read in and so forth.

3.5 CPU Implementation

A general purpose multi-core CPU with two cores is first tested to implement the algorithm and used as a baseline to compare with other architectures. Since the algorithm operates on a single pixel, it is simply encased in two for loops to iterate over the entire frame. The organization of this test splits the frame into equal sized pieces for each thread for the number of cores available. In this case, the frame is split into two pieces height-wise, so the first thread processes the top half, and the second thread processes the bottom half. In other words, the threads operate on separate contiguous chunks of data, and do not interpolate between even/odd pixels. The reasoning behind processing using height-wise chunks is due to better spatial locality of the pixel and model data.

3.6 CUDA Implementation

In the CUDA architecture, program execution takes place partly on the host CPU and partly on the GPU. The reading of the frame and the writing of the processed frames must be done by the host CPU. This means that large amount of data need to be moved between the GPU memory and the host main memory. In this work CUDA 2.1 is the most current version. CUDA 2.1 did not allow overlapping of kernel execution and memory transfer, which was implemented in CUDA 2.2. The program flow is thus wait for data; send data to GPU; run CUDA kernel; receive data from GPU; repeat.

The kernel itself is implemented as the algorithm running on a single pixel. In CUDA, each processor has a unique x, y, and z ID assigned to it. In this algorithm, only x and y are used. The IDs represent the x,y coordinate pixel to operate on. This particular algorithm is very simple since no data is shared. Processing only one pixel per kernel utilizes the least amount of registers, important for maximizing occupancy in the processors.

Table 1: Baseline Results

Size	Models	Time (ms) Single Core	Time (ms) Dual Core
320x240	3	40.38	23.06
	5	43.71	23.52
640x480	3	128.37	66.39
	5	175.33	71.1

3.7 CBEA Implementation

On the Cell Processor, the algorithm is run on the SPEs, while the PPE performs disk reads and sets up execution regions for the SPEs. Much like the CPU implementation, the frame is split height-wise into equal sized chunks to be processed based on the number of SPE cores available. The PPE determines these regions once for both disk buffers at the start of the program, and sends the address to each SPE along with the number of pixels in the chunk. This homogenous approach is done for simplicity.

Since the size of the local store in each SPU is limited, as is the size of each DMA transfer, pixel and gaussian model data is packed into a structure the size of a DMA transfer. This means that for more gaussians, less pixels are transferred per DMA. To speed up transfers, each SPE queues 8 incoming transfers and 8 out-going transfers, determined by the maximum amount of DMA transfer slots in the MFC queue. Each SPE iterates over the chunk data using for loops, and writes the result to the original pixel data in the DMA structure. The whole structure is transferred back to the main memory buffer.

The DMA structure also contains flags to signal to the PPE that processing is complete. The completed flag is set when the SPE has finished processing the entire chunk. The PPE main loop waits for all SPEs to complete before writing data back to disk and reading in the next frame.

4. RESULTS

The algorithms described in the previous sections are implemented and evaluated on an Nvidia GeForce 9800 GPU, an Nvidia GTX 280, and IBM Cell processors, respectively. The performance is compared against a baseline multi-thread implementation running on an Intel Core2 Duo dual-core CPU. The experimental results show a significant speed-up in terms of performance (up to 17.82 times for CUDA and 2.77 times for CEBA, respectively).

Results for the dual-core and were measured using built-in timers in the respective architectures. For CUDA, the CUDA utilities timer was used. For the CPU and Cell processor versions, the linux timing utilities are used.

4.1 Baseline Results

As a baseline test for the background subtraction, an iden-

Table 2: CUDA results GF 9800 and GTX 280

Size	Models	Time (ms)	Time + Mem (ms)	Time + Mem + Write (ms)
320x240	3	1.8	2.34	3.32
	5	2.14	2.66	3.35
640x480	3	6.81	8.82	12.56
	5	8.19	10.2	14.8
320x240	3	0.57	0.86	1.29
	5	0.92	1.22	1.61
640x480	3	2.17	3.34	4.30
	5	3.61	4.78	5.72

tical algorithm was developed for a general-purpose x86 processor. The test system was equipped with a dual core 2.16 GHz Core 2 Duo processor with 2GB of main memory. For each of the test scenarios described below, the test was run on the CPU version of the code, as well as the GPU code version.

In order to create a fairer comparison with current technologies, a multi-threaded version of the baseline test was developed. The multi-threaded version has a processing thread count equal to the number of physical CPU cores times the number of simultaneous multithreading issue slots to maximize memory bandwidth. An additional helper thread is launched to read and write image frames to disk. Results from both the single threaded and multi-threaded versions are shown in Table 1.

Test frames from this run are shown in Figure 4. The test video was recorded with an inexpensive camera resting on a table, and a person walking in front of the camera several times. The test frames replace foreground pixels with white and background pixels with black for easier visibility of the output results.

4.2 CUDA Results

The GPU based results were performed on a GeForce 9800 with CUDA version 1.1. Results were also measured on a GTX 280 with CUDA version 1.3. The CUDA SDK is built for linux, and was run on Fedora 9 with the 2.6.27 32-bit kernel. The results from both runs are shown in Table 2, with GeForce 9800 first and GTX 280 second. All times are in milliseconds per frame and are the averages of a few dozen frames.

Performance can be evaluated using the CUDA profiling tool. Nvidia's occupancy calculator [9] and variables output from the CUDA compiler can also provide some insight as to how well the GPUs resources are being utilized, but is not as detailed the profiler. Using the occupancy calculator spreadsheet, the occupancy is only 75% for CUDA versions 1.2 – 1.3, and 33% for versions 1.0 and 1.1. This is due to the fact that 17 registers are used in the program's kernel, limiting the number of registers available per multiprocessor.

CUDA Observations. This leads to the observation, that although the occupancy is not 100%, performance is NOT dramatically improved by forcing the compiler to use only 16 registers. A register count of 16 is the optimal number of registers for CUDA versions 1.2 – 1.3, and occupancy was increased to 100%. For older CUDA versions, occupancy was increased to 75%. Register count can be forced to a limit by saving registers in local memory. Local memory is thread-specific and has the same latency as global memory.

The previous observation outlines one of the key features of the CUDA architecture. Even though less of the processor resources are being utilized, there are still sufficient threads available to run while other threads are handling global memory accesses. In combination with the no-overhead switching between thread warps and the sheer number of threads, there is almost always a thread in a runnable state. Therefore, the global memory accesses are still hidden by these runnable threads. The number of threads is maximized in the CUDA implementation by having each thread process only one pixel.

Table 3: CBEA code results

Size	Models	Time – 6 SPUs
320x240	3	14.77
	5	16.37
640x480	3	56.08
	5	61.47

Table 4: CBEA code with various SPU counts

Size	Models	SPUs	Time
320x240	5	2	36.79
		3	26.48
		4	21.47
		5	18.24
		6	16.37
		7*	10.36
		8*	8.49

* Estimated

4.3 CBEA Results

Cell processors results were gathered on a playstation 3 running Fedora Core Linux version 6 with kernel 2.6.16. Using the libspe API calls, 6 available SPUs were reported as being available. Since the algorithm is based on the number of SPUs available, this number can easily be edited to perform tests on lower numbers of SPUs and extrapolate the results for larger SPU counts. Results can be seen in Table 3 for 6 SPUs. Results for 320x240 using various numbers of SPUs can be seen in Table 4. Using these numbers we can estimate speedups from additional SPUs, also shown in Table 4. The estimated speed-ups were determined by graphing the results in Excel and adding a best fit line, then inserting the number of SPUs into the equation generated. Estimates are denoted with an asterisk.

Performance of the Cell architecture can be evaluated using the systemsim full system simulator. The systemsim simulator allows for dynamic profiling of statistics such as total cycles, DMA channel stall cycles, and branch hint hits [10]. Since the test code utilizes multi-buffered read and writes with the maximum number of transfers in the MFC queue, the channel stalls were minimal. The simulator also helped with tuning branch prediction hints, since they may not always be as obvious as they seem – even to the programmer! For example, the programmer may assume that the first gaussian model sorted by weight will be matched, branching out of the for loop. However, this may not always be true. Using the simulator, branch statistics for several different test cases can be analyzed.

CBEA Observations. Unlike the CPU and CUDA versions of the algorithm, the cell processor version is more sensitive to the number of gaussian models used. This sensitivity was observed when using a number of gaussians greater

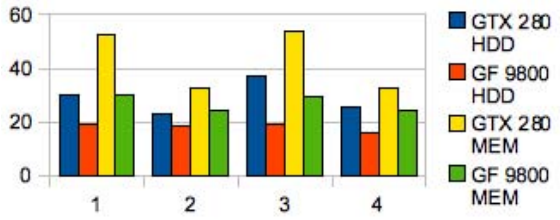

Figure 5: Speedup using Streams

than 15 on the cell processor when compared to the CUDA or CPU versions. This happens because data must be explicitly transferred to the SPU local stores for processing, and the number of variables associated with each pixel increases with the number of gaussians. This can result in more time waiting for DMA transfers, since the main processing loop of the SPU spends less time processing and the number of transfers increases with more gaussian models. Even with the maximum DMA transfer size only a few tens of pixels can be sent for processing in one transfer for a very high number of gaussian models.

Based on this observation, a modified version of the CBEA algorithm was implemented. In this modified version, only pixel data is sent, and the gaussian matching loop explicitly requests gaussian models. The idea is that less unused data will be transferred to the SPU local store. For example, if there are 10 gaussian models, and the first gaussian model is a match, the remaining 9 models are never transferred to the CPU, with the exception of the weights that must be updated. However, this modified version had other problems. Using small DMA transfers for the gaussian models is not the most efficient use of the interconnect bus, and the transfers suffer from the Convoy effect by getting placed in the MFC queue after pixel data, which is the maximum DMA transfer size.

5. FUTURE WORK

The CUDA code developed for the algorithm was implemented using API version 2.1. Starting with version 2.2, Nvidia introduced the ability to create streams. Streams allow memory to be copied asynchronously, so memory transfers and kernel executions can occur at the same time. This can be used within the CPU-side code to overlap memory transfers and disk reads with kernel executions. This can effectively hide disk latency or memory transfer latency for diskless implementations of this algorithm. At the time of writing, this algorithm was not implemented, so the results shown are estimated based on results gathered from the CUDA implementation in section 4.2.

Using the results obtained from the CUDA program, we can determine the speedup of introducing this technique. Subtracting Time from the Time+Mem+Write results in table 2, the time of the memory transfer and disk reads can be estimated. Since the kernel execution runs at the same time as memory transfers and disk reads, a stream sequence will be complete at the speed of the slower of the two. Using this number we can compare with the original results, and approximate the speedup of using this approach. Likewise for diskless implementations we can determine the memory transfer time by subtracting the Time from the Time+Mem result, and use the maximum as the time for one sequence in a stream to complete. Results of the two are shown in Figure 5. In the figure HDD denotes a hard-drive based system while MEM denotes a memory only implementation.

6. CONCLUSION

In this paper we explored the usage of NVIDIA's parallel programming language CUDA and IBM's Cell Processor to improve the speed of a background subtraction algorithm used in various computer vision applications. Based on the optimized results, a speedup of up to 17.82 times was seen using CUDA when compared to the results of the multi-threaded algorithm running on a multi-core CPU. With the addition of CUDA streams, the algorithm can provide even more speed up over the CPU version of the algorithm. Using CBEA a speedup of up to 56% was measured for 6 SPUs, and an estimated speedup of 2.77 times for 8 SPUs.

Although these results are not as significant as other algorithms ported to CUDA, they do provide enough speed to process relatively large frames, 640x480 and even larger, with more gaussian models at 30 frames per second than possible with the CPU application. The bottleneck of the acceleration is mainly the large amounts of memory transfers that are unavoidable in this application. Since the algorithm itself is not extremely computationally intensive, memory movement results in a noticeable amount of the kernel execution time.

7. REFERENCES

[1] T. Horprasert, D. Harwood, and L. S. David, "A robust background subtraction and shadow detection," in *Proceedings of the 1st Asian Conference on Computer Vision*, 2000.

[2] J. Schlessman, M. Lodato, B. Ozer, and W. Wolf, "Heterogeneous mpsoc architectures for embedded computer vision," in *International Conference on Multimedia and Expo*, 2007.

[3] N. Friedman and S. Russell, "Image segmentation in video sequences: A probabilistic approach," in *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, 1997.

[4] P. KaewTraKulPong and R. Bowden, "An improved adaptive background mixture model for real-time tracking with shadow detection," in *Proceedings of the 2nd European Workshop on Advanced Video Based Surveillance Systems*, 2001.

[5] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1999.

[6] N. Corporation, "Nvidia compute unified device architecture programming guide 2.1," in *http://www.nvidia.com/object/cuda$_d$evelop.html*.

[7] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, July/September 2005.

[8] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. A. Yelick, "The potential of the cell processor for scientific computing," in *Conf. Computing Frontiers*. ACM, 2006, pp. 9–20.

[9] N. Corporation, "Cuda occupancy calculator," in *http://developer.nvidia.com/*.

[10] M. Kistler, D. Murrell, and V. Sachdeva, "Using systemsim to guide application transformation and tuning for the cell broadband engine," 2006.