Porting an MPEG-2 Decoder to the Cell Architecture

Troy Brant, Jonathan Clark, Brian Davidson, Nick Merryman
Advisor: David Bader

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0250
troy.brant@gatech.edu, antimatter3009@gmail.com, gtg884r@mail.gatech.edu,
nick.merryman@gmail.com

ABSTRACT

In this paper, a method to port an existing MPEG-2 decoder to the Cell architecture is presented. The proposed method consists of selecting an existing MPEG-2 implementation, parallelizing the MPEG-2 video stream at the slice level, and changing the code base to take advantage of the nine total processors on the Cell chip.

1. INTRODUCTION

The Cell architecture consists of one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). These processors give the Cell the ability to execute up to nine threads of instructions simultaneously. The PPE is a 4 GHz, 64-bit IBM PowerPC processor that is primarily used to coordinate the work done on the SPEs. Each SPE is a 4GHz processor that generally performs the brunt of the work for any given program run on the Cell.

2. MPEG-2

MPEG-2 is video compression format used for many types of video media, including DVDs, Internet video, and digital cable [1]. MPEG-2 is an extension of the MPEG-1 specification and can handle both progressive and interlaced videos.

The MPEG-2 standard instructs how to both encode and decode videos, but we will be focusing solely on decoding in this paper.

2.1 Format

An MPEG-2 video file consists of an hierarchy of layers [2]. The highest layer is the sequence layer, and there is one sequence per video file. A sequence consists of one or more groups of pictures, each of which contain one or more frames. Each frame is then broken down into horizontal slices, where each slice contains multiple macroblocks, which are themselves made up of 8 x 8 pixel blocks.

2.2 Parallelization

To optimize an MPEG-2 library for Cell, the decoding process needs to be parallelized in order to take advantage of the multiple SPEs on the Cell processor.

There are several layers of MPEG-2 where you could implement parallelization, including the groups of pictures (GOP) layer, the frame layer, the slice layer, the macroblock layer, and the block layer. The GOP layer would not be a good choice for parallelization because small videos will probably only have a couple GOPs and would not take advantage of all the SPEs on the Cell. There is no way to parallelize at the frame layer because most frames require sequential decoding. Again, there is no way to parallelize at the macroblock layer because macroblock decoding depends on previous macroblocks for motion compensation. The block level would not be a feasible option because of the amount of time that would be spent communicating between the PPE and SPEs would far outweigh any benefits gained from parallelization.

The slice layer is the best choice for parallelization in the MPEG-2 decoder because each slice in a frame can be decoded independently, unlike almost every other layer. The slice layer has also been the choice of parallelization in several other research papers, including [3], [4], and [5].

2.3 Libraries

There are many libraries available that implement MPEG-2 on several platforms. ffmpeg is the most

popular implementation and is used in a wide variety of Linux media players. libmpeg2 is also widely used in Linux media players and has performance comparable to ffmpeg. We were largely unsuccessful in attempting to port both of these libraries to the Cell. The problem we ran into with each of the libraries was that they were highly optimized for performance, and thus the code was highly unreadable. With the limited amount of time we had to finish the research project, we determined our time would be better spent finding a more understandable code base to work from.

For our implementation library, we chose mpeg2play. The mpeg2play library is an optimized version of a prototype MPEG-2 library. The prototype library was written as an educational tool to show developers implementing MPEG-2 how the standard was supposed to work. This library is a great choice to port to the Cell because the code is commented extensively and can be easily restructured, unlike the other highly optimized MPEG-2 libraries.

3. PORTING MPEG-2 TO CELL

The mpeg2play library is single threaded and needed massive restructuring to take advantage of the Cell's multi-core capabilities.

3.1 Slice Parallelization

The first step to porting the library to Cell was to change the logical structure of the program in order to support parallelization. The mpeg2play decoder originally worked by loading a 2KB chunk of a file into a buffer, and advancing through the bitstream and filling back up the buffer as necessary until the end of the file was reached. For parallelization, however, we needed to split the bitstream into slice-sized chunks and pass them to the SPEs. So, we changed the program so that instead of just advancing through the bitstream and working on whatever data is there, the program works on the bitstream like normal until a slice start code is found. Then, the bitstream is stored in a buffer until the next slice start code is found. At that point, the bitstream is set to point to the beginning of the slice buffer, and the decoding function `getMBs()` works on the data in the slice buffer. When it's done, the original bitstream with the file data is restored, and the process continues until all slices in a frame are decoded. This change was necessary for parallelization because the PPE on the Cell will store the slices in buffers and send them to SPE for decoding.

3.2 SPE Support

The next step was to convert the code to compile on the Cell architecture by splitting code into portions that will run on the PPE and portions that will run on the SPE. The general work-flow is that the PPE splits the bitstream into slices, sends the slices to the SPEs, and the SPEs decode the slices and return the result to the PPE. To achieve this, the PPE needs to DMA transfer to the SPE a structure that contains the address of the slice buffer to decode, all the global variables the SPE needs to perform the decoding, and the destination addresses for the decoded blocks. The SPE needs to have all the code required to perform the decoding. Thus, we added a global_data structure that contains the global information that needs to be passed from the PPE to the SPE, and we also added an "spu" directory to the code base that contains all the files required to perform the decoding of a slice. After making these changes, we integrated the mpeg2play Makefile with the generic Cell Makefile, and compiled the program for successfully for the Cell.

3.3 Work Queue

Now we are able to send the data to the SPE for decoding purposes, but at this stage we can only offload the work to one SPE and have to wait for that one to finish. To fully optimize the decoder, we would need a way to dispatch SPEs to do work and wait for them to finish and give them more work.

This led us to the development of the Work Queue / Wait Queue Synchronization System. The PPU calls into this system to send a slice decode request to an available SPE and doesn't worry about it being handled and instead just moves to dispatch the next decode request. When all available SPEs have been assigned work, the remaining requests are pushed onto the end of the Work Queue and are available when an SPE is freed up after decoding its slice. When an SPE gets done with their work, it sends back its response in a delayed verification fashion and contacts the PPE for more work. The sending of the work in a delayed verification fashion is a form of double buffering. We tell the processor to start copying the memory back, but we don't verify it was actually sent until we are starting two decoding iterations from now. This allows the DMA transfer to take place while the next slice is being decoded.

When the SPE is done with its work and there are no more requests on the Work Queue, the SPE is put on the Wait Queue until more work comes or until the PPE signals we are done with the frame. Once the entire frame is decoded, the PPE sends a synchronization signal to the SPEs. This allows all the SPEs to verify that all of their delayed verification responses have been written back, and we are then able to move on to the next frame.

3.4 High-Resolution Video Support

The main challenge with using an SPE to decode MPEG-2 video is determining which portions of the reference frame to keep in the local storage of the SPE at any given time for the decoding of P and B frames. For these frames, the motion vectors choose nearby pixels from the reference frames on which to base the decoding of the current frame. These motion vectors can select a reference pixel between 0 and 15 pixels away in any direction – this means for a given slice, you must have 46 horizontal pixel rows from the reference frames in the local storage of an SPE.

For high-resolution videos, this is far too much data to be held in local storage all at once. The strategy used to overcome this problem is to logically divide the decoding of each slice on an SPE into chunks such that the leftmost part of the slice is decoded, then additional reference frame information is transferred to the SPE before continuing to the next logical chunk. Double buffering of data is implemented within a slice on this chunk-level.

4. Performance

In general, we found the Cell implementation of the mpeg2play library to be 4 times faster than the original x86 version. The Cell speedup is essential for larger videos that take more time to decode each frame and may cause noticeable skipping in the single-threaded version of mpeg2play.

[3] presents a slice optimization for decoding B frames in the MPEG-2 stream without synchronizing after each frame, but we did not implement the optimization in our system. Our results are reasonable considering that synchronization is required after decoding each frame.

Name	Resolution	Frames	Pattern
short.mpg	352 x 240	7	IBBPBB
dein_cact	704 x 480	250	IPPPPPPPPPPPPP

Table 1: Test videos used in performance analysis

Table 1 shows the two primary videos we used to test the performance of our system. The frames in the dein_cact video are two times as large as the short.mpg video, thus they also contain roughly two times as many slices.

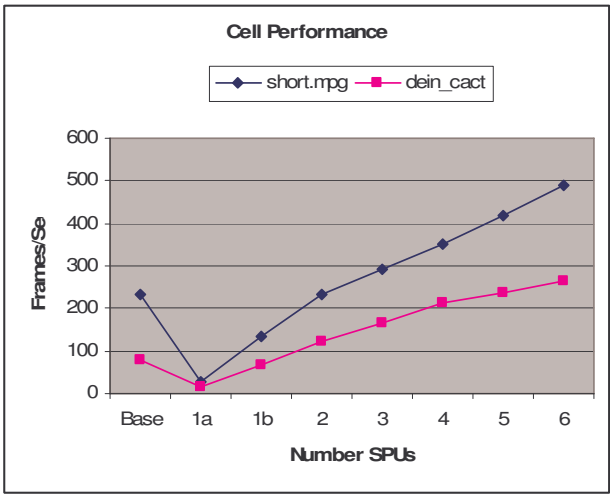

Figure 1: Frames per second of Cell version compared to single threaded version

In figure 1, we performed a comparison of our Cell implementation to the single-threaded MPEG-2 implementation. The 'Base' label on the x-axis represents the single-threaded implementation. '1a' labels the test version of our Cell implementation that starts and stops a single SPE each time a slice needs to be decoded. '1b' labels our final Cell implementation that uses one SPU that is started once and receives slices from the PPE via mailboxing.

In figure 1, the initial decrease in performance when one SPE is enabled is expected. Enabling only one SPE is equivalent to running the program as a single thread, but with the additional overhead of performing DMA transfers for every slice in the video.

Our results show a general linear improvement in performance as more SPEs are added. This is a great property of the system because as more SPEs are fit onto future Cell processors, our system's performance will scale linearly with the number of SPEs enabled during decoding.

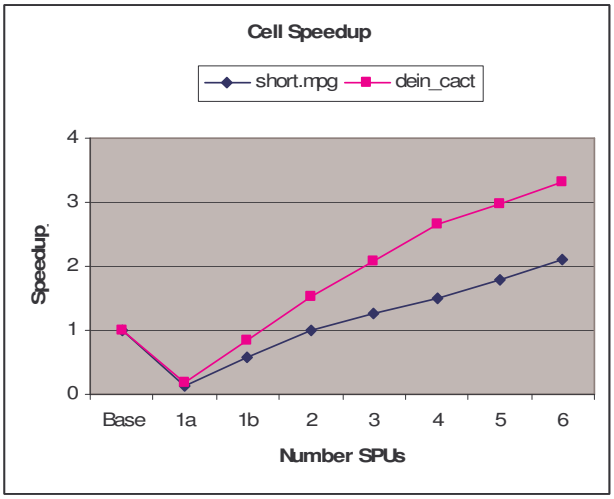

Figure 2: Speedup of Cell version compared to single threaded version

Figure 2 shows the speedup of the Cell implementation over the single threaded version of mpeg2play. As more SPEs are enabled, the dein_cact video has a more impressive speedup than the short.mpg video. Each frame in the dein_cact video is two times as large as a frame in the short.mpg video, and thus it has two times as many slices as a frame in the short.mpg video. This increase in the number of slices means that the SPEs can continuously do more work before needing to be synchronized at the end of a frame, and so the dein_cact video takes advantage of the parallelization of the system more than the short.mpg video does.

5. CONCLUSION

Our research has produced an MPEG-2 decoder optimized for the Cell architecture. We implemented a work queue on the PPE that contained the slice chunks to be passed to the next available SPE. This work queue idea may help to inspire similar designs in future projects porting libraries to the Cell processor. The decoder we developed isn't perfect and doesn't work with every type of MPEG-2 video, but it works for most videos and provides a high level of speedup over single-threaded MPEG-2 libraries.

5.1 Future Work

There are several optimizations and improvements that could be made to the current MPEG-2 Cell decoder. For optimization, each SPE can execute vectorized code using SIMD instructions. SIMD instructions give you parallelization at the instruction level, allowing you to execute four instructions per cycle instead of one.

Another optimization is providing double buffering, pre-caching, for the DMA transfers needed in a slice decode request. Our attempts at adding double buffering were unsuccessful due to memory constraints. There is a tradeoff, however, in that if double buffering is implemented, you wouldn't be able to transfer as much of the slice over to the SPE for decoding and will need more DMA transfers than our current single-buffered solution.

The Cell decoder can also be made more robust by handling system stream MPEG-2 videos, which use a different bitstream format than typical MPEG-2 videos. Also, the decoder needs to handle videos that only have a single slice per frame, a video type that we came across experimentally.

6. CONTRIBUTIONS

This research project was conducted by three undergraduates (Troy Brant, Jonathan Clark, and Brian Davidson) and one graduate student (Nick Merryman). Everyone in the group performed the following tasks:

- Learned Cell programming techniques by implementing the RC5 compression algorithm on Cell
- Researched the MPEG-2 standard
- Researched MPEG-2 implementations, such as ffmpeg, libmpeg2, and mpeg2play
- Cross-compiled the various MPEG-2 implementations for Cell

Our individual contributions to the project are listed as follows:

6.1 Troy Brant

Troy was responsible for making the initial changes to the mpeg2play library detailed in section 3.1 of this document. In particular, he altered the library so that instead of just progressing sequentially through the MPEG-2 bitstream, the program broke the bitstream into slice buffers that were decoded individually. He was also contributed to the overall design and development of the MPEG-2 Cell implementation.

6.2 Jonathan Clark

Jonathan was responsible for getting our code to write PPM formatted images. These were necessary for testing as they allowed us to run the code on the simulator and check the output. We could also check the output more closely with static images rather than a playing video. Additionally, Jonathan helped with the initial move of the code to the Cell, including writing the data structure necessary to pass information between the PPU and the SPUs. Finally, he worked on trying to get the IDCT portion of the code to use Altivec, but it never ended up working.

6.3 Brian Davidson

Brian was responsible for the development of the Work Queue / Wait Queue Synchronization System and the integration of this system with the necessary main memory detached slice level decoding efforts doing basically the canonical example of parallelization by dealing with starting up some SPE threads dispatching the work and collecting the results. Additionally, Brian made an attempt at further optimization by trying to implement a pre-caching double buffering schema when used in conjunction would mitigate the delay of DMA transfers, as the last response memory transfer and the next request memory transfer would happen in the background while the SPE was busy perform the current request, but getting successful implementation was derailed by the fact that each SPE has only a 256K local store, which corresponds to a little less space than needed to hold the current slice and pre-cache the next one, thus we were unable to allocate enough memory to do the work.

6.4 Nick Merryman

Nick was responsible for all aspects of decoding an individual slice on a single SPE. The code he developed was designed to run in parallel on as many SPEs are available. This included all data transfers to and from the SPE and modifications to the decoding structure necessary due to the limited local memory space of each SPE. He also implemented double buffering of memory on the SPEs.

7. REFERNCES

[1] P. N. Tudor. *MPEG-2 Video Compression*. In Electronics & Communication Engineering Journal, Volume 7, Issue 6, December 1995, pages 257 – 264.

[2] ISO/IEC 13818-2. MPEG-2 video coding standard, March 1995.

[3] A. Bilas, J. Fritts, and J. P. Singh. *Real-Time Parallel MPEG-2 Decoding in Software*. Proceedings of the 11th International Parallel Processing Symposium, 1997.

[4] E. Iwata and K. Olukotun. *Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm*. In Proceedings of the International Conference on Supercomputing (ICS), 1999.

[5] A. Bala, D. Shah, W. Feng, D. K. Panda. *Experiences with Software MPEG-2 Video Decompression on an SMP PC*. In Proceedings of the 1998 International Conference on Parallel Processing Workshops (ICPPW), 1998.