ADVANCES IN THE PARALLELIZATION OF MUSIC AND AUDIO APPLICATIONS

Eric Battenberg, Adrian Freed, & David Wessel
The Center for New Music and Audio Technologies and
The Parallel Computing Laboratory
University of California Berkeley
{eric, adrian, wessel}@cnmat.berkeley.edu

ABSTRACT

Multi-core processors are now common but musical and audio applications that take advantage of multiple cores are rare. The most popular music software programming environments are sequential in character and provide only a modicum of support for the efficiencies to be gained from parallelization. We provide a brief summary of existing facilities in the most popular languages and provide examples of parallel implementations of some key algorithms in computer music such as partitioned convolution and non-negative matrix factorization NMF. We follow with a brief description of the SEJITS approach to providing support between the productivity layer languages used by musicians and related domain experts and efficient parallel implementations. We also consider the importance of I/O in computer architectures for music and audio application. We lament the fact that current GPU architectures as delivered in desk and laptop processors are not properly harnessed for low-latency real-time audio applications.

1. INTRODUCTION

Concerns with energy consumption and heat dissipation have put the squeeze on processor clock rates and the move to multi- and many-core computer systems appears to be the only viable way to improve computing performance. Multi-core CPUs and many-core GPUs are now widely available in consumer desk and lap-top systems, yet music applications, as with nearly the entire landscape of applications, have yet to provide much in the way of efficiency via parallelism.

The most widely used software environments in the interactive computer music community, Max/MSP/Jitter, PD, SuperCollider, and CHUCK are fundamentally sequential programming paradigms. These environments have exploited multi-core systems by running copies of the program in separate threads. The operating system in turn then schedules the threads on the different cores.

With the client-server architecture of the SuperCollider one core is typically identified as the client and the remaining cores are servers thus providing a rudimentary mechanism for coordination.

In the PD environment Miller Puckette (Puckette 2009) has provided a PD~ abstraction which allows one to embed a number of PD patches, each running on separate threads, into a master patch that handles audio I/O.

Max/MSP provides some facilities for parallelism within its **poly~** abstraction mechanism. The **poly~** object allows one to encapsulate a single copy of a patcher inside of its object box and it automatically generates copies that can be assigned to a user specified number of threads which in turn are allocated to different cores by the operating system. This mechanism has the flavor of data parallelism and indeed the traditional musical abstractions of voices, tracks, lines, and channels can make effective use of it.

We also hasten to mention Yann Orlarey's efforts to automatically parallelize FAUST (2009).

We now consider two example parallel implementations, one with real-time constraints and another that uses a GPU.

2. PARALLEL PARTITIONED CONVOLUTION

2.1. Background

Partitioned convolution is an efficient method of performing low-latency convolution with a long FIR filter. A typically use is in convolution-based reverb where the impulse response represents the reverberations of a large hall, small room, or other interesting environment. More creative uses exists as well, such as applying a drum sound as the impulse response to create rich, percussive sounds out of recordings of mundane tapping (such as on a table or chair).

An algorithmic approach to partitioned convolution was introduced by Gardner in (Gardner 1995) with a set partitioning of exponentially increasing block sizes. While this method is much more efficient than direct convolution, it misses out on the benefits of reusing the FFTs taken for each block size. Garcia (2002) demonstrates the use of a frequency delay line (FDL) for each block size, which encourages FFT reuse among blocks of the same size. He also develops an algorithm to arrive at a theoretically optimal partitioning given a desired latency and impulse response length. Adriaensen (2006) in his *jconv* has used

non-uniform block sizes for low latency convolution reverb applications.

2.2. Our Approach

The implementation of partitioned convolution that we are developing puts more emphasis on real-time guarantees and efficiency along with utilizing the multiple cores of parallel processors. **Figure 1** shows an example partitioning with 3 FDLs. The first FDL contains blocks equal to the desired latency. The bottom graph shows how each FDL is scheduled in a separate thread and allowed to compute for the amount of time associated with its block size. This gives a total latency of 2N/fs, where fs is the sample rate, since the input buffer of the first FDL must be filled with N samples then we compute for time N/fs.

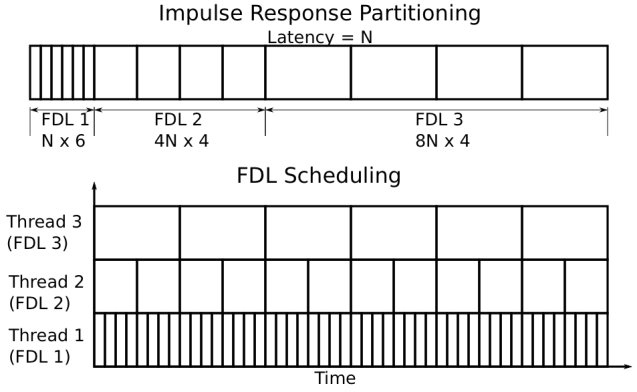

Figure 1. Top - An example of slicing an impulse response up into multiple FDLs. Bottom - The thread for each FDL is allowed to compute for as long as it takes to fill its next buffer.

The real-time threads of each FDL can run in parallel on a distinct processor core, until synchronization and buffering is required at each horizontal boundary in the scheduling diagram. In addition to the parallelism afforded by FDL threading, many-core system can exploit data-parallelism to compute individual FFTs and complex vector multiplies on multiple cores.

Our approach to coming up with a "best" partitioning does not focus on reducing the total number of operations or even reducing overall compute time. This is because we focus on making real-time (as opposed to throughput) guarantees for our implementation. Therefore, we measure a worst-case execution time (WCET) for each size of FDL by polluting the cache or measuring while other processes are running. Only then can we reason about what partitioning scheme would behave best in real-time for a given latency, impulse response length, and machine.

We have developed an Audio Unit, a Max/MSP external, and a portable Portaudio implementation. We expect this work to enable us to apply long impulse responses to many (100+) channels in real-time.

3. ACCELERATING NON-NEGATIVE MATRIX FACTORIZATION

3.1. The Algorithm

Here we describe our efforts to speed up non-negative matrix factorization (NMF), an unsupervised learning technique that has been used in audio source separation and parts-based analysis (Lee *et al* 1999, Virtanen 2007). NMF can be used for audio source separation by decomposing a spectrogram matrix into two matrices which contain source-wise spectral contributions and time-varying gains.

NMF can be phrased as an optimization problem in which we attempt to find two matrices, **W** and **H**, with non-negative values whose product approximates the input matrix, **X**, with minimal error, i.e. $\mathbf{X} \approx \mathbf{WH}$.

We use a matrix divergence cost function to calculate the error and minimize this cost function with respect to **W** and **H** using multiplicative updates (eq. 1), which are introduced in (Lee *et al* 1999).

$$\mathbf{H} \leftarrow \mathbf{H} .* \frac{\mathbf{W}^T \frac{\mathbf{X}}{\mathbf{WH}}}{\mathbf{W}^T \mathbf{1}} \qquad \mathbf{W} \leftarrow \mathbf{W} .* \frac{\frac{\mathbf{X}}{\mathbf{WH}} \mathbf{H}^T}{\mathbf{1} \mathbf{H}^T} \tag{1}$$

In the above, division is carried out element-wise, ".*" is element-wise multiplication, and **1** represents a matrix of ones and is used to compute row and column sums.

The details of our approach to drum track extraction using the above updates are covered in (Battenberg 2008). Although these updates are simple to implement and efficient compared to more complicated optimization strategies, they can still take a very long time to converge and completely dominate the computation time of a source-separation task. Therefore, NMF is an important computational procedure to optimize. We have implemented NMF using CUDA to target highly parallel graphic processors and OpenMP to target multi-core CPUs.

3.2. Implementations

3.2.1. CUDA

CUDA is the API used to program Nvidia GPUs for general-purpose computations. The hardware on a CUDA-compatible GPU can greatly accelerate data-parallel computations. For example, for a single-precision matrix-matrix multiply, the current CUDA BLAS (CUBLAS) can achieve up to 375 Gflops/s on the GTX 280 GPU, while the Intel MKL BLAS only achieves 38 Gflops/s on a Core2 Duo 2.67GHz CPU (Volkov *et al* 2008).

In our CUDA implementation of NMF, we use the highly tuned CUBLAS for matrix multiplication (which makes up the bulk of the computation) and an optimized routine to compute array sums.

3.2.2. OpenMP

OpenMP is an API that simplifies the process of parallelizing C code for multi-core CPUs. We use OpenMP pragmas to parallelize *for*-loops and the multi-threaded Intel MKL matrix multiply routines in our C with OpenMP implementation.

The details of both implementations are covered in (Battenberg 2009).

3.3. Performance Results

The chart in **Figure 2** shows the execution time (smaller is better) for various implementations of NMF on different architectures. The CUDA version runs over 30 times faster on a GTX 280 GPU than the Matlab version runs on a Core 2 Duo T9300. The OpenMP version runs over 7 times faster on a dual-socket Core i7 920 than the Matlab version.

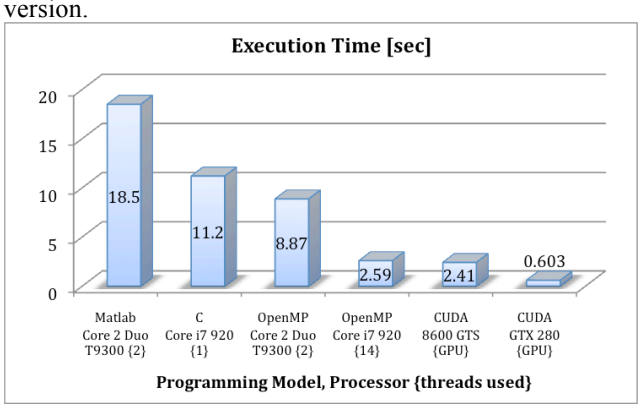

Figure 2. Running time comparison for 200 iterations of 30-source NMF run on 20 seconds of audio data.

Although the CUDA version performs much better, it is important to point out that writing the CUDA implementation took almost 10 times as long as writing the OpenMP version. Therefore, for most programs, CUDA should be reserved for only the most compute-intensive routines.

4. BRIDGING THE GAP

As we just noted, efficient parallel code is not easy to develop. Our music and audio languages should provide mechanisms that help exploit parallelization without the encumbering details of the efficiency layer implementation. In the Par Lab at UC Berkeley there is promising research on an idea know as **S**elective, **E**mbedded **J**ust-**I**n **T**ime **S**pecialization **(SEJITS)** (Catanzaro *et al* 2009)**.** Let's use the example above that implemented, in CUDA, non-negative matrix factorization, a key component in music information retrieval (MIR) systems.

A music researcher would write MIR software in a productive and expressive scripting language such as PYTHON or RUBY and express operations on time-frequency representations like the multiplication and division in the example above in the scripting language. This scripted version of the software would remain as the portable source code. When the script is run on a particular architecture SEJITS determines if efficient implementations of selected code segments are available and if so generates the appropriate code in an efficiency layer language like C, Cilk, C++, OpenCL, or CUDA as in this example.

5. BALANCED ARCHITECTURE COMPUTING

Balancing Load/Store rates and arithmetic unit rates is key to achieving high performance in modern computer architectures. This problem is commonly framed in terms of load/store throughput to memory. For current music and media processing applications especially where low-latency is important it is necessary to also consider balancing data flows from the I/O system. The industry is currently focused on optimizing systems with video I/O, usually with emphasis on output and typically on a small number of outputs corresponding to monitor ports. The resulting GPU architectures aren't very effective in audio applications with a high input and output stream count typical of wave-field, beam-forming microphone and speaker array applications. They also don't address the trend towards higher gesture information rates from high resolution multitouch surfaces, and time-of-flight 3D cameras.

We recognized this need for balanced architectures from the beginning of CNMAT in 1989 and developed our early real-time performance software on the Reson8 machine (Barrière *et al* 1989) which was optimized for musical applications. The Reson8 employed 7 24-bit processors connected via dual-port RAM to a master processor. This processor could read I/O from any processor in a single instruction cycle time and broadcast to all with a single cycle. Each processor had its own SSI port capable for 8-channels of digital I/O.

When the clock rates of DSP processors started to lag mainstream general purpose processors we moved to develop on SGI Indigo and Indy which had high performance multichannel audio I/O and a unique bus that synchronized all video and audio peripherals to the system clock. The REACT hard real-time scheduling of SGI IRIX provided the means to achieve the low latency high performance potential of the hardware. We explored multiprocessor musical applications using the two processor SGI Octane which also had high performance I/O to support digital video, computer graphics and audio simultaneously.

As SGI's CPU performance fell behind mainstream processors we moved to mainstream PowerPC and Intel

desktop and laptop machines. Noticing performance bottlenecks and resource contention on the buses proposed with these machines we developed our own multichannel audio I/O hardware using FPGA's and Ethernet connectivity. These were extended to include gesture acquisition and clock synchronization and we have demonstrated scaling comfortably to hundreds of channels.

The recent interest in multicore machines presents new challenges because I/O is now bottlenecked at the boundary between the CPU core and off chip I/O sources and sinks. We would like to see more per CPU support for timely I/O and more pins allocated to I/O and are building demonstration application to explore the associated design tradeoffs. We will explore the idea that direct I/O into each core may be a useful way of avoiding cache interaction and contentions problems as we observed in our work on the Reson8.

6. CONCLUSIONS

On the horizon is a diversity of many-core architectures with varying amounts of homogeneity. The SEJITS approach has promise for providing portability across the rapidly evolving and likely rough terrain of new computer architectures, compilers, and code generation strategies. For SEJITS to do its job the scripting language must have mechanisms for introspection and meta-programming. We need to examine our languages in this regard.

The example MIR application of SEJITS we provide is a non-real-time one and in no way can we stop there. We need to bring temporal criteria to efficiency layer programming and pass performance behavior stats to a SEJITS like environments to find optimal implementations.

In the work on parallel partitioned convolution we provide the opportunity to auto-tune not only raw performance but also temporal performance and timely behavior. The accelerated non-negative matrix factorization demonstrated the power of the current generation of GPUs to provide significant speedups. These results along with many other non-video applications of GPUs call out for better I/O facilities on GPU architectures that would better support low-latency performance. We need GPU CPU combinations to be more flexible with I/O. The reality is that we will need more I/O pins to our tiles of CPUs.

7. ACKNOWLEDGEMENTS

Supported by a grant from Intel and Microsoft (Award #20080469), and the UC Discovery Program (Award #DIG07-10227). Special thanks to Andy Schmeder, Rimas Avizienis, and John MacCallum for discussions.

8. REFERENCES

F. Adriaensen (2006) “Design of a convolution engine for reverb” International Linux Audio Conference, Karlsruhe 2006.

E. Battenberg and D. Wessel, (2009) “Accelerating non-negative matrix factorization for audio source separation on multi-core and many-core architectures” International Society for Music Information Retrieval Conference, 2009.

E. Battenberg, (2008) “Improvements to percussive component extraction using non-negative matrix factorization and support vector machines,” Masters Thesis, University of California, Berkeley, Berkeley, CA, 12 2008.

J. B. Barrière, P. F. Baisnee, A. Freed, and M. D. Baudot. (1989) “A digital signal multiprocessor and its musical application” *Proceedings of the 15th International Computer Music Conference,* ICMA, pp 17-20, 1989.

Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf (LBL), Kathy Yelick, Armando Fox. (2009) *SEJITS: Getting Productivity And Performance With Selective, Just-In-Time Specialization.* Proc. 1st Workshop on Programming Models for Emerging Architectures (PMEA’09), Raleigh, NC, Sept. 2009.

G. Garcia, (2002) "Optimal filter partition for efficient convolution with short input/output delay," Proceedings of the Audio Engineering Society Convention, 2002

W. Gardner, (1995) "Efficient convolution without input-output delay," Journal of the Audio Engineering Society, vol. 43 (3) pp. 127-136, 1995.

D. Lee and H. Seung, (1999) “Learning the parts of objects by non-negative matrix factorization,” Nature, vol. 401, no. 6755, pp. 788–791, 1999.

D. Lee and H. Seung, (2001) “Algorithms for non-negative matrix factorization,” Advances in neural information processing systems, pp. 556–562, 2001.

Y. Orlarey (2009) “Adding automatic parallelization to Faust” International Linux Audio Conference, 2009.

M. Puckette, (2009) “Multiprocessing in PD” *Proceedings of the 3rd International Pure Data Convention Sao Paulo Brazil* 2009.

T. Virtanen, (2007) “Monaural sound source separation by non-negative matrix factorization with temporal continuity and sparseness criteria,” IEEE Transactions on Audio, Speech, and Language Processing, vol. 15, no. 3, pp. 1066–1074, 2007.

V. Volkov and J. Demmel, (2008) “Benchmarking GPUs to tune dense linear algebra,” in Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, 2008, pp. 1–11