DogoRangsang Research Journal UGC Care Group I Journal
ISSN: 2347-7180 Vol-12 Issue-01 April 2022

Separating audio from its sources quickly by using nonnegative matrix factorization on many-core and multicore processors.

Dr D Arun Kumar[1,] A Valli Basha [2] S L Pratap Reddy[3], R V Sree Hari[4]

[2] Asst. Professor, Department of ECE, K. S. R. M College of Engineering(A), Kadapa

[1,3,4] Associate Professor, Department of ECE, K. S. R. M College of Engineering(A), Kadapa

ABSTRACT

However, iterative NMF algorithms are computationally demanding, making the time to convergence relatively sluggish on normal home computers, which has prevented their widespread application in audio source separation and parts-based analysis. Using OpenMP for shared-memory multicore systems and CUDA for many-core graphics processsores, we detail high-performance parallel implementations of NMF in this work. Our OpenMP and CUDA implementations cut processing time for 20 seconds of audio from 18.5 to 2.6 and 0.6 seconds, respectively. These speed improvements make it possible to execute source separation on whole songs in a matter of seconds, a task that was previously impractical due to its lengthy execution. We explain the techniques used to achieve these massive speedups and advocate for the widespread use of parallel music information retrieval programs.

INTRODUCTION

We have not yet seen significant implementation of MIR approaches in end-user applications, despite the rising relevance and popularity of MIR research. This may be in part because of the prevalence of hand-labelled data and collaborative filtering-based music recommendation services like Pandora and Last.fm online, but it is also likely due to the fact that many MIR techniques are computationally too complex to be used effectively outside of large-scale computing clusters. If the execution time of MIR approaches could be shortened, it would allow for faster assessment and adjusting of algorithm parameters and more frequent real-world deployment, both of which would considerably accelerate the pace of advancement in MIR research. Some effort has been made to focus on developing quick implementations, although it is far from adequate. Using the Maryssa audie processing system, Stamatakis produced submissions to MIREX 2007 that were orders of magnitude quicker than those of rivals while achieving equivalent results [1]. The multi-core Stamatakis implementation, for instance, finished the audio mood classossification work in 2 minutes, whereas other implementations took between 8 minutes and 3 hours. Such significant performance disparities may significantly affect the usability of MIR software, even for research implementations. Non-negative matrix faceatomization (NMF) is an unsupervised learning approach that has been applied in audio source separation and parts-based analysis [2, 3, 4, 5]. In this research, we report our work to speed up percusssave source separation using NMF. The composting time for such a source separation job is mostly determined by NMF.

Optimization is a crucial computational process. This work aims to encourage MIR researchers to create and reuse high-performance parallel implementations of crucial MIR processes by showcasing the huge speedup that can be gained by using multi-core and many-core versions of multimedia applications. In Section 2, we detail why it's so crucial to develop MIR applications in parallel. The section 3 focuses on the more tangible aspects of NMF-based audio source separation. OpenMP and CUDA are presented as parallel programming paradigms in Section 4. Our parallel implementations and several essential strategies for parallelizing MIR applications are described in Section 5. Finally, Section 6 wraps up with recommendations for making the most of parallel computing in MIR.

PARALLELIZING MULTIMEDIA APPLICATIONS

Percussive source separation is a useful first step in such MIR tasks as drum transcription, rhythm summarization, and beat tracking. By extracting an audio signal containing only percussive instruments, the task of rhythmic analysis can be greatly simplified. Helen and Virtanen [6] use NMF along with a support vector machine (SVM) to accomplish this. The drum track extractor we use as a target for performmince optimization is similar to that presented in [6] but includes additional complexity optimizations and percusssave features introduced in [7]. Computation time in this system is dominated by NMF, which makes up about 80% of the CPU time (18.5 seconds of the 23.1 seconds total) in a MATLAB implementation run on 20 seconds of audio. In order to increase throughput, the NMF step must be optimized. Because single-core CPU performance increases have been hindered by power concerns, limits on memory speed, and diminishing returns on instruction level parallelism, the focus of computer science research has turned strongly towards parallel architectures and programming models [8]. Applications programmers can no longer develop a sequintail implementation of their software and hope that future uniprocessor speedups will provide the necessary computting power to make their application useful. Instead, the exponentially increasing number of processing elements, or cores, in current architectures must be exploited to maxamaze performance. Multi-core CPU architectures are already commonplace in workstations, servers, and laptops, so parallelizing code to utilize available cores will lead to significant performmince increases for most users. In addition, the majority of personal computers today ship with many-core graphices processors contained on the system's video card. Current high-end graphics processors (GPUs) ship with tens of processors each capable of executing operations on large data vectors. The end result is a highly data-parallel Architexture that can be used for general computation (not just graphics rendering) thanks to programming frameworks like OpenCL [9] and Nvidia's CUDA [10]. CUDA has been successfully used to achieve very high performance on a variety of applications that rely on signal processing and machine learning. Examples include a fast GPU-based support vector machine implementation that achieves up to 135× speedup over LIBSVM [11], a large vocabulary speech recognition engine with 10× speedup over sequential versions [12], and an image contour detecttor that achieves 114× speedup [13]. To help put these numbers in perspective, the 114× speedup represents a reduction in runtime from 4 minutes to 2 seconds. We aim to achieve such dramatic performance gains with NMF-based source separation.

NON-NEGATIVE MATRIX FACTORIZATION FOR AUDIO SOURCE SEPARATION

Non-negative matrix factorization can be used for audio source separation by decomposing a spectrogram matrix into two matrices which contain source-wise spectral conattributions and time-varying gains. NMF can be phrased as the optimization problem:

Given an $M \times N$ non-negative matrix $\mathbf{X} \in \mathbb{R}_+^{M\times}$
find matrices $\mathbf{W} \in \mathbb{R}_+^{M\times K}$ and $\mathbf{H} \in \mathbb{R}_+^{K\times N}$ that min
mize the cost function $f(\mathbf{X}, \mathbf{WH})$.

Cost Function

Rather than using the mean-squared error between X and the product WH as the cost function, we use a matrix verySion of the Kullback-Leibler divergence:

$$D(\mathbf{X}\|\mathbf{WH}) = \sum_{ij}\left(\mathbf{X}_{ij}\log\frac{\mathbf{X}_{ij}}{(\mathbf{WH})_{ij}} - \mathbf{X}_{ij} + (\mathbf{WH})_{ij}\right)$$

It has been shown in [3] that this divergence cost function achieves better audio source separation results than mean-squared error.

Multiplicative Updates

Lee and Seung [14] have proposed an algorithm based on gradient-based multiplicative updates for minimizing the above optimization problem. For the divergence cost function, we alternate between updates on the two matrices using the following expressions

$$\mathbf{H} \leftarrow \mathbf{H}.*\frac{\mathbf{W}^{\mathrm{T}}\frac{\mathbf{X}}{\mathbf{WH}}}{\mathbf{W}^{\mathrm{T}}\mathbf{1}}, \qquad \mathbf{W} \leftarrow \mathbf{W}.*\frac{\frac{\mathbf{X}}{\mathbf{WH}}\mathbf{H}^{\mathrm{T}}}{\mathbf{1}\mathbf{H}^{\mathrm{T}}} \quad (2)$$

Where division is carried out element-wise, ".∗" is elementwise multiplication, and 1 represents an M × N matrix of ones and is used to compute row and column sums. It is important to note that, because the optimization problem is not convex in both W and H, the above updates do not necessarily converge to a global minimum. To address this problem, researchers typically use multiple random initializations and choose the best result. Adding extra computation time by running multiple trials cannot be done without significant justification since time to convergence can be in the minutes when operating on just seconds of audio

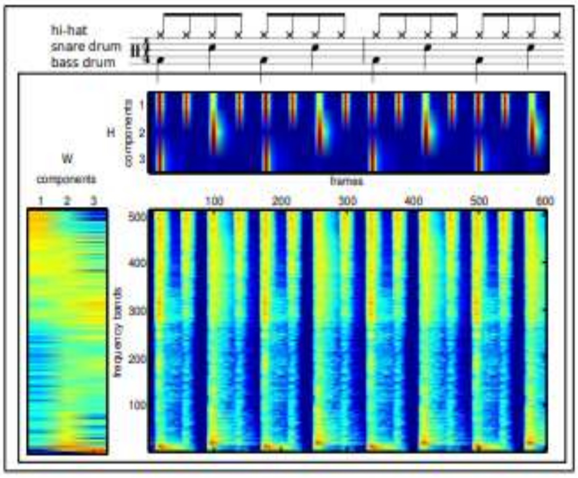


Figure 1. A spectrogram matrix for a basic rock beat surrounded by its factor matrices W and H computed using NMF. The component-wise gain matrix H has been aligned with the corresponding drum score.

Initialization

Other approaches use a deterministic initialization based on the structure or statistics of the matrix X or derived from knowledge about the domain. We use an approach based on the latter [7], which uses a subset of discrete cosine transform basis functions and typical drum spectra as the initial columns of W. For our purposes, the initializaton choice does not directly affect the speed with which the updates in eq. (2) are executed, but it can affect the overall number of iterations required for convergence. To eliminate this dependence, we will only focus on optimizeIng the speed of a set number of iterations rather than time to convergence.

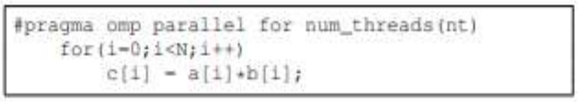
Matrix Dimensions

An additional consideration that must be made is the dimonotonality of the spectrogram matrix that is to be facetori zed. To adequately represent drum sounds in both time and frequency, a length 4096 Hann window is used to extract each analysis frame and a hop size of 256 is used to shift the window in time. For 20 seconds of audio Sampled at 44.1kHz, this gives us a matrix of size 2049×3445 (number of positive frequency bins × number of analysis frames). Since such high frequency resolution (~10Hz) is not required at higher frequencies, we use a Bark-based perceptual dimensionality reduction [7] on the columns of X to arrive at a matrix of size 512 × 3445. After NMF is carried out on this smaller matrix, we can interpolate to return to the original frequency scale if necessary. Lastly, we choose an inner dimension for the factor matrices W and H of K = 30. This represents the number of sources involved in the separation. Using these dimensions, our implementations require about 60MB of memory per minute of audio, making entiresong decomposition feasible from a memory standpoint. Next, we introduce the programming models that will be used to parallelize the NMF algorithm.

OPENMP AND CUDA

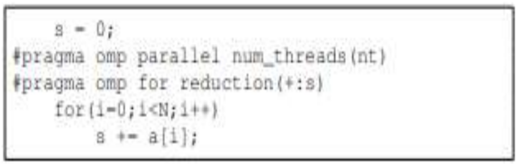
OpenMP

is a standardized API that enables parallel exection on shared-memory multi-core machines [15]. OpenMP has been implemented for C, C++, and Fortran and is supported in Visual C++ 2005, the Intel compiler, and gcc 4.2 and above.The beauty of OpenMP lies in its ability to parallelize existing sequential code by annotating it with compiler directives. OpenMP automatically forks threads that execute on separate processors according to the directives. OpenMP very conveniently parallelizes loops containing independent iterations using a single directive. The element-wise array multiplication shown below can be split amongst nt cores using a leading #pragma directive



```
#pragma omp parallel for num_threads(nt)
    for(i=0;i<N;i++)
        c[i] = a[i]*b[i];
```

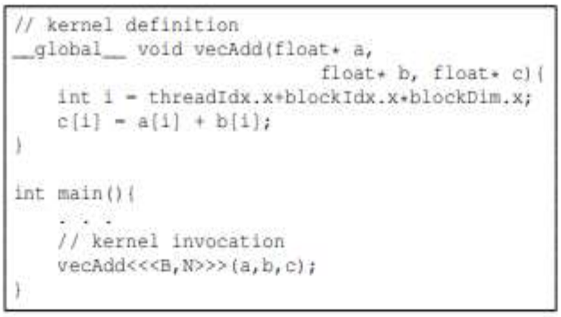
A reduction, which operates on multiple pieces of data and returns a single result, can be carried out using a reduction clause in the for pragma. In the example below, the reduction operator is addition, so we are returning the sum of an array. The first pragma creates a team of nt threads that are each assigned a chunk of the work in the for loop. After each thread completes its work, the values contained in each thread's private variable s are summed into a single final variable s.



```
    s = 0;
#pragma omp parallel num_threads(nt)
#pragma omp for reduction(+:s)
    for(i=0;i<N;i++)
        s += a[i];
```

CUDA

encompasses both the parallel device architecture used in newer Nvidia GPUs and the extensions to the C language used to program the CUDA architecture for general purpose computation. CUDA code compiled using Nvidia's nvcc is executed on the host, or CPU, which then issues instructions to the device or GPU. Host code typically contains control flow instructions and memory movement operations between host memory and device memory, while device code is made up of kernels, which are functions written to execute in a Single Program, Multiple Data (SPMD) fashion, i.e. each thread running on the device during kernel invocation executes the kernel code independently on whatever chunk of data is assigned to the thread. Teams of threads can also share memory. As of CUDA 2.1, threads can be grouped into thread blocks of up to size 512. Threads within the same block are executed on the same processor and can all access special on-chip shared memory, which is necessary for inter-thread communication. Because separate thread blocks cannot share data, they can be executed independently on separate processors. Therefore, a kernel that uses a large number of thread blocks should scale well on future GPUs with more processors. In the box below, we see a kernel that performs elementwise addition. Each thread runs the vecAdd function separately and computes an array index from its thread ID, block ID, and block size, and operates on the array elements located at that index. In the main function, the kernel is invoked with B thread blocks each containing N threads, so B × N should be equal to the size of the arrays.



```
// kernel definition
__global__ void vecAdd(float* a,
                              float* b, float* c){
    int i = threadIdx.x+blockIdx.x*blockDim.x;
    c[i] = a[i] + b[i];
}

int main(){
    . . .
    // kernel invocation
    vecAdd<<<B,N>>>(a,b,c);
}
```

Device kernels are physically executed in groups of 32 adjacent threads called warps. Warps are most efficient when the group of threads can be executed in a completely SIMD (Single Instruction, Multiple Data) manner, i.e. each thread in the warp does the exact same thing but to different data. Inserting control flow statements into a kernel that cause threads within the same warp to execute

different code (this is referred to as a “divergent” warp) forces the affected threads to be run sequentially rather than concurrently. Double-precision hardware support is currently lacking in CUDA, which is why we focus on single-precision implementations in this work. CUDA is designed to achieve high throughput on highly data-parallel computations. Luckily, most multimedia applications (especially music) exhibit a large amount of data parallelism.

PARALLEL IMPLEMENTATION

Important Kernels

To help organize our NMF implementation, we decompose the updates in eq. (2) into the most important computational kernels, including dense matrix multiplication, column and row sums, and element-wise vector arithmetic. Each of the kernels will be called sequentially, but individual kernels will be heavily parallelized and optimized. The kernel that will do the most work in terms of floating point operations (flops) is the Single-precision GEneral Matrix Multiply, or SGEMM. For the matrix dimensions listed at the end of Section 3.4, the four SGEMMs in eq. (2) require about 423 Mflops. The element-divides require about 3.6 Mflops, the sums about 0.1 Mflops, and the elementmultiplies about 0.1 Mflops. To prevent dividing by zero, a small constant (called EPS) is added to every element in each divisor matrix, which produces a non-trivial amount of work (3.6 Mflops). Also, in order to check for convergence, we compute the divergence cost function (1) every 25 iterations, which computes the sum of 1.8 × 106 logbased values. Even though the SGEMMs contain the vast majority of the work, other operations, namely the slow floating-point divides and the sums, can end up using a lot of compute time. Divides are inherently slow operations and can take tens of clock cycles on certain architectures. While the sums contain relatively few total operations, a parallelized sum will require inter-thread communication which can be very slow. Since a highly optimized SGEMM routine is available in most vendor BLAS libraries, our implementation goal was to tune the remaining kernels so that the SGEMMs dominate the overall computation time. Practically speaking, significantly outperforming our Matlab implementation (which takes 18.5 seconds to run 200 iterations on a Core 2 Duo T9300) was a more exciting goal.

OpenMP Implementation

As stated before, OpenMP makes it very easy to parallelize existing sequential code for a multi-core shared-memory machine. Using the two types of for pragmas from Section 4.1 we can parallelize the sums and element-wise arithmetic. Since the element divides are numerous, slow, and do not require inter-thread communication, it makes sense to parallelize their loop. The row and column sums, however, require a lot of communication for the amount of addition work done per core (since the partial sum computed by each core must be sent to another core), so parallelizing the reduction loop actually led to a slower kernel. The larger sum in the divergence cost function not only contains lots of addition but a slow log-based computation, so the work to communication ratio was befitting parallel speedup. For the SGEMMs, we use Intel’s Math Kernel Library (MKL) ver. 10.0.1.014, which is heavily optimized to take advantage of memory hierarchy and SIMD instructions. MKL uses OpenMP under the hood, so the number of threads used for the SGEMMs can be controlled in the same way as our parallel loops. Performance results for the OpenMP implementation are shown in Figure 2 for a dual-socket Intel Core i7 920 machine which has 8 cores and 16 hardware threads. The best performance is seen at 14 threads and is about 4.3× faster than the single-threaded run. The most significant speed up is seen in the SGEMM since it has the highest work to communication ratio, but other time-consuming kernels benefit as well. Running this implementation on the Core 2 Duo T9300 with 2 threads

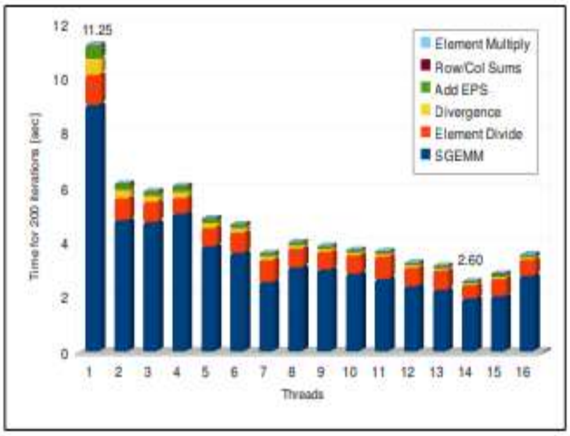


Figure 2. Performance results for the OpenMP implementation on a dual-socket Intel Core i7 920

CUDA Implementation

Writing a CUDA implementation takes a bit more thought. First, the matrices must be copied to GPU memory. Copies between CPU and GPU are relatively slow (ideally 3 GB/s over the PCI bus), and it's best to avoid them except during initialization or when returning results. This means that in our case it's better to perform all of the matrix computations on the GPU to avoid extra copies even if certain operations are better suited for the CPU. Element-wise arithmetic is completely data-parallel and is easily accomplished with code similar to that in Section 4.2. Other kernels, including the SGEMMs and sums, require a bit of inter-thread communication and are not so trivially parallelized on CUDA.

SGEMM

Luckily, an optimized SGEMM routine is available in the CUBLAS 2.1 library that achieves 60% of theoretical peak performance for large matrices on current GPUs [17]. For the Geforce GTX 280, 60% of peak amounts to 373 Gflops/s. For our particular matrix multiplications of dimensions[512× 30 × 3445], [30 × 512 × 3445], and [512 × 3445 × 30], the CUBLAS SGEMM achieves 117, 147, and 104 Gflops/s respectively on this GPU. Even though these are relatively small SGEMMs, we should still be able to do better. Upon inspection of the paper [17] that describes the methods used in the current CUBLAS SGEMM, we discovered that threads operate on matrix sub-blocks with dimensions 16 and 64. With this in mind, we tried zero padding our matrices to multiples of 16, 32, and 64. We found that simply padding the matrices to multiples of 32 resulted in an effective throughput (not counting operations on zero-padded areas) of 264, 196, and 85 Gflops/s for each SGEMM size. Since the NMF algorithm uses two SGEMMs of the first size, this results in an SGEMM running time reduction from 0.71 to 0.52 seconds for 200 iterations.

Reduction

Because parallel reductions, such as sums, mins, and maxes, are not included in standard libraries, we will have to write our own routines. A tutorial on optimizing reductions in CUDA is available in the CUDA SDK [18]. This overview presents optimization strategies that can be used to greatly improve the speed of large power-of-2-size reductions and shows how a 30× speedup can be achieved for a 4.2 × 106 length sum over a naive binary tree implementation. A binary tree reduction can be constructed in various ways. Using the shared memory of a thread block, we can perform a series of two-element reductions. Two ways to organize the overall reduction are shown in Figure 3. In both versions, each thread in the thread block starts by reading an array element from global memory into shared memory. Then threads are assigned to carry out two-element sums. The difference lies in which threads work on which array elements. Method 1 interleaves working and non-working threads which act on adjacent elements. Method 2 sequentially assigns working threads so there are contiguous blocks of working and non-working threads. This decreases the number of divergent warps. Also, the memory accesses are strided rather than adjacent to reduce the number of simultaneous memory bank accesses (since shared memory locations are cyclically assigned to memory banks) [16]. In addition to reorganizing the tree traversal, other optimizations –such as explicit loop unrolling and allowing each thread to read and sum multiple array elements into its shared memory location before the tree traversal begins– improve performance a bit. These techniques had to be adapted for non-power-of-2-size arrays, but they greatly improved the speed of the large 1.8 × 106 length divergence sum. For the smaller 512 and 3445 length column and row sums, these techniques were not quite enough, and the CUDA kernel ran much slower than a sequential CPU version. In

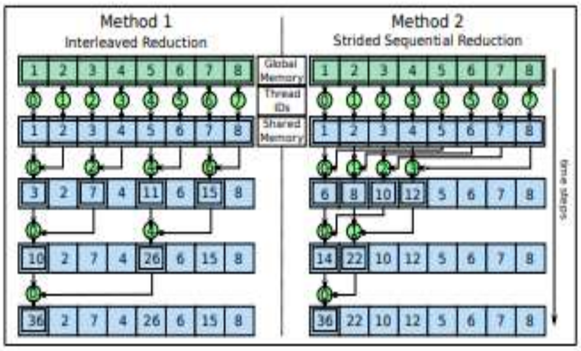


Figure 3. Two methods of shared memory reduction

order to produce more concurrent work (in terms of thread blocks), we can compute all 30 of the column or row sums simultaneously. This is accomplished by launching a 2D grid of thread blocks, in which the first dimension represents which of the 30 sums is being computed and the second dimension indexes the thread blocks within the individual sum. This final optimization produced staggering speedup for the 30 smaller sums as shown in Figure 4

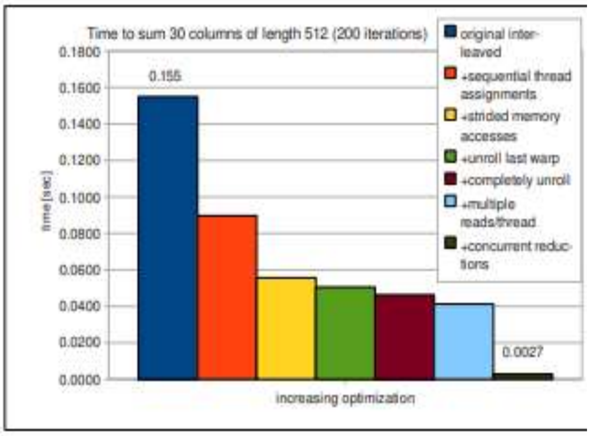


Figure 4. Cumulative effect of various optimizations on running time of 200 iterations of the 30 column sums

CUDA Performance Results

The results for the CUDA implementation compared to OpenMP and Matlab implementations are shown in Figure 5. The Matlab implementation is optimized for singleprecision vector operations and uses the dimensionality reduction technique mentioned in Section 3.4. Our Matlab implementation runs about 3× faster than a naive Matlab implementation that doesn't use dimensionality reduction. The OpenMP version runs more than twice as fast as the Matlab version on the same machine, and shows significant speedup when using more threads on the Core i7; however, the non-linear speedup between 1 and 14 threads suggests that the OpenMP version will not scale well to more cores. Our CUDA implementation shows great performance on the older Geforce 8600 GTS, which has 4 multiprocessors at 1.46 GHz. The newer Geforce GTX 280, with 30 multiprocessors at 1.3GHz, runs the CUDA implementation over 30× faster than the optimized Matlab implementation and 18× faster than the single-threaded OpenMP version on the Core i7 920. Both of these GPUs are marketed to consumers for desktop gaming and graphics so are quite affordable compared to many of the professionalgrade cards. Additional speedup is possible with future GPUs with more multiprocessors and greater memory bandwidth. As stated earlier, CUDA programs scale well if kernels have a large number of independent thread blocks. The relatively small size of the matrix operations doesn't guarantee strong scaling in the future, but in this case, additional speedup is not necessarily required. For audio source separation, the NMF already performs at 33× real-time on the GTX 280.

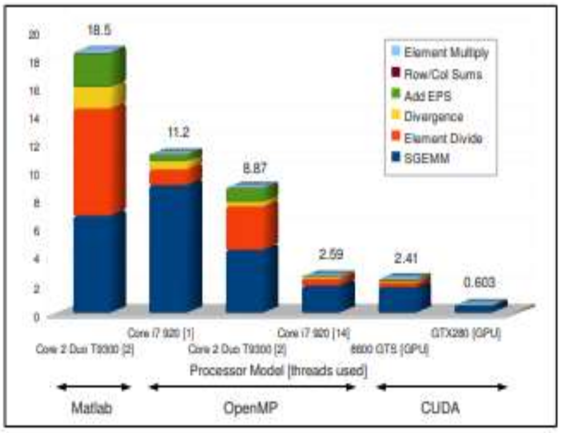


Figure 5. Running time comparison for 200 iterations of 512×30×3445 NMF using optimized implementations in Matlab, OpenMP, and CUDA on different architectures

DISCUSSION AND FUTURE WORK

After achieving such significant speedup on the NMF step of percussive source separation, the next step would be to parallelize the remaining pieces of the complete source separation process. As with the bulk of signal processing and machine learning routines, these steps are all very data-parallel (since individual audio frames can be processed independently) so would benefit from parallelization. When choosing between OpenMP and CUDA for programming MIR applications, it is important to note that while CUDA can achieve superior performance on newer GPUs, the programmer effort required is much greater than with OpenMP, which is a better starting point for those who already know how to program in C. We must also remember that parallel MIR applications do not necessarily have to be coded from scratch. Many MIR techniques can be assembled from basic building blocks that already have fast parallel implementations. In addition to standard libraries like MKL, fftw, and CUBLAS, many researchers have released parallel implementations of important routines. We will be releasing Python modules for the implementations described in this paper so that other researchers can benefit from the speed gains. We feel that sharing high-performance, user-friendly tools in order to encourage more widespread use of parallel implementations within the MIR community is an important step in increasing the practicality of MIR techniques.

REFERENCES

[1] G. Tzanetakis: "Marsyas submissions to MIREX 2007", MIREX 2007, 2007. URL: http://www.musicir.org/mirex/2008/abs/mirex2007.pdf

[2] D. Lee and H. Seung: "Learning the parts of objects by non-negative matrix factorization," Nature, Vol. 401, pp. 788–791, 1999.

[3] T. Virtanen: "Monaural sound source separation by nonnegative matrix factorization with temporal continuity and sparseness criteria," IEEE Transactions on Audio, Speech, and Language Processing, Vol. 15, No. 3, pp. 1066–1074, 2007.

[4] P. Smaragdis and J. Brown: "Non-negative matrix factorization for polyphonic music transcription," IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, pp. 177–180, 2003.

[5] A. Cont, S. Dubnov, D. Wessel: "Realtime Multiple-Pitch and Multiple-Instrument Recognition for Music Signals Using Sparse Non-Negative Constraints," Proceedings of the International Conference on Digital Audio Effects (DAFx), 2007.

[6] M. Helen and T. Virtanen: "Separation of drums from polyphonic music using nonnegative matrix factorization and support vector machine," Proc. EUSIPCO, 2005.

[7] E. Battenberg: "Improvements to Percussive Component Extraction Using Non-Negative Matrix Factorization and Support Vector Machines," Masters Thesis, University of California, Berkeley, December 2008. URL: http://cnmat.berkeley.edu/publications/author/Battenberg

[8] K. Asanovic, R. Bodik, et al.: "The landscape of parallel computing research: A view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December, 2006.

[9] A. Munschi: "OpenCL: Parallel computing on the GPU and CPU," SIGGRAPH08: ACM SIGGRAPH 2008 classes, 2008.

[10] J. Nickolls, I. Buck, et al.: "CUDA: Scalable parallel programming," ACM Queue, April, 2008.

[11] B. Catanzaro, N. Sundaram, and K. Keutzer: "Fast support vector machine training and classification on graphics processors," Proceedings of the 25th international conference on Machine learning, pp. 104– 111, 2008.

[12] J. Chong, Y. Yi, et al.: "Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors," Proceedings of the 1st Annual Workshop on Emerging Applications and Many Core Architecture (EAMA), pp. 23–25, 2008.

[13] B. Catanzaro, B. Su, et al.: "Efficient, high-quality image contour detection," International Conference on Computer Vision, 2009.

[14] D. Lee and H. Seung: "Algorithms for Non-negative Matrix Factorization'," Advances In Neural Information Processing Systems, pp. 556–562, 2001.

[15] Open MP Architecture Review Board: OpenMP application programming interface, Ver. 2.5, May 2005.

[16] "Nvidia CUDA Programming Guide," Ver. 2.1, URL: developer.download.nvidia.com/compute/cuda/2 1/toolkit/ docs/NVIDIA CUDA Programming Guide 2.1.pdf, 2008.

[17] V. Volkov and J. Demmel: "Benchmarking GPUs to tune dense linear algebra," Supercomputing 08, 2008.

[18] M. Harris: "Optimizing parallel reduction in CUDA," Nvidia Cuda SDK 2.1, URL: http://developer.download.nvidia.com/compute/cuda/ sdk/website/projects/reduction/doc/reduction.pdf, 2008.