Journal of Management & Entrepreneurship
ISSN 2229-5348

UGC Care Group I Journal
Vol-11 Issue-01 Jan 2022

Accelerating nonnegative matrix factorization on many-core and multicore systems

Mr. MOLUGUMATI PREMCHAND[1], Mrs. K. RAMA LAKSHMI[2] ,Mrs. DONGA HINDUJA[3] ,
ASST PROFESSOR[1] ,ASSISTANT PROFESSOR[2,3] , DEPARTMENT OF ECE,
SWARNANDHRA COLLEGE OF ENGINEERING AND TECHNOLOGY, NARASAPUR

ABSTRACT

Non-negative matrix factorization (NMF) has been extensively employed in parts-based analysis and audio source separation; however, the time to convergence of iterative NMF algorithms is very sluggish on standard personal computers due to their computational complexity. In this research, we describe high performance parallel implementations of NMF, created with CUDA for many-core graphics processors and OpenMP for shared-memory multicore systems. We reduce the running time for a 20-second audio clip from 18.5 seconds to 2.6 seconds with OpenMP and 0.6 seconds with CUDA. Source separation was previously impractical in terms of time, but these performance improvements make it possible to complete songs in a matter of seconds. We shed light on how such large speed increases were achieved and promote the advancement and

INTRODUCTION

Although research on music information retrieval (MIR) is becoming more and more important, MIR approaches are still not widely used in end-user applications. This might be partly explained by the widespread use of collaborative filtiring and hand-labeled data as the foundation for online music recommendation services like Pandora and Last.fm, but it's also likely because many MIR techniques are computationally too complex to use outside of large compute clusters. If the execution time of MIR techniques was sufficiently shortened to enable more frequent and efficient real-world application, as well as faster evaluation and adjustment of algorithm parameters, the rate of advancement of MIR research may be significantly increased. There has been some focus on the necessity of producing quick implementations, but not

Optimizing this computing process is crucial. In addition to encouraging MIR researchers to create and reuse high performance parallel implementations of significant MIR procedures, the purpose of this study is to highlight the huge speedup that can be realized by multi-core and many-core implementations of multimedia applications. We discuss the significance of generating parallel MIR applications in Section 2. The practical aspects of audio source separation based on NMF are discussed in Section 3. We present the OpenMP and CUDA parallel programming models in Section 4. In addition to providing information on methods crucial for parallelizing MIR applications, Section 5 describes in detail the architecture of our parallel implementations. Secton 6 ends with recommendations for maximizing the advantages of parallel computing for MIR.

PARALLELIZING MULTIMEDIA APPLICATIONS

For MIR tasks like beat tracking, rhythm summarization, and drum transcription, percussion source separation is a helpful initial step. The process of rhythmic analysis can be significantly streamlined by isolating an audio signal that only contains percussion instruments. To do this, Helen and Virtanen [6] combine NMF with a support vector machine (SVM). Similar to the one shown in [6], but with additional complexity improvements and percussive elements added in [7], is the drum track extractor we utilize as a goal for performance optimization. In a MATLAB implementation running on 20 seconds of audio, NMF accounts for almost 80% of the CPU time (18.5 seconds of the 23.1 seconds total). This indicates that NMF dominates computation time in this system. To boost the rate of throughput, the

NON-NEGATIVE MATRIX FACTORIZATION FOR AUDIO SOURCE SEPARATION

The process of decomposing a spectrogram matrix into two matrices that contain source-wise spectral conattributions and time-varying gains is known as non-negative matrix factorization, and it can be applied to audio source separation. NMF is sometimes referred to as the optimization problem.

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:

$$D(\mathbf{X}\|\mathbf{WH}) = \sum_{ij}\left(\mathbf{X}_{ij}\log\frac{\mathbf{X}_{ij}}{(\mathbf{WH})_{ij}} - \mathbf{X}_{ij} + (\mathbf{WH})_{ij}\right) \quad (1)$$

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

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

In element-wise division, elementwise multiplication is represented by ".∗," and row and column sums are calculated using 1 as a M × N matrix of ones. It is significant to note that the aforementioned updates may not converge to a global minimum because the optimization problem is not convex in both W and H. Researchers usually employ many random initializations and select the best outcome to overcome this issue. Since the time to convergence can be measured in minutes while working with just seconds of audio, it is not possible to add extra computing time by conducting multiple trials without a compelling reason.

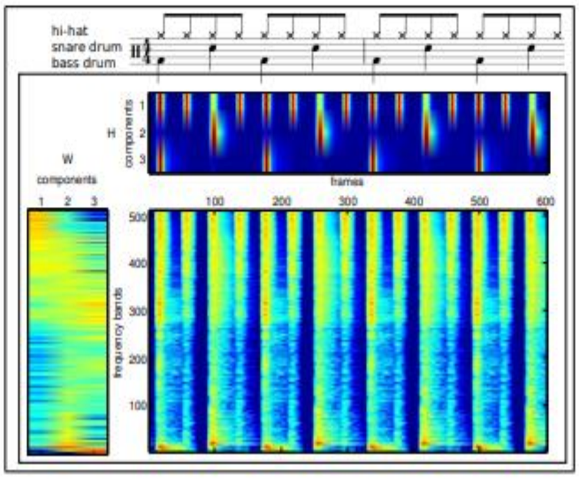

Figure 1. A spectrogram matrix for a basic rock beat surrounded by its factor matrices W and H computed using NMF. The component-wise gain matrix H has been aligned with the corresponding drum score.

Initialization

Alternative methods employ a deterministic initialization that is determined by the domain knowledge or the matrix X's structure or statistics. Our method is based on the latter [7], where the initial columns of W are a subset of discrete cosine transform basis functions and typical drum spectra. For our purposes, the initialization decision can influence the total number of iterations needed for convergence, but it has no direct impact on how quickly the updates in equation (2) are carried out. We will solely concentrate on optimizing the speed of a predetermined number of iterations rather than the time to convergence in order to get rid of this dependency.

Matrix Dimensions

The dimonotonality of the spectrogram matrix that is to be facetorized is another factor that needs to be taken into account. Each analysis frame is extracted using a length 4096 Hann window, and the window is shifted in time using a hop size of 256 in order to accurately represent drum sounds in both time and frequency. 20 seconds of audio This provides us with a matrix of size 2049×3445 (number of positive frequency bins × number of analysis frames), sampled at 44.1 kHz. At higher frequencies, we do not need such a high frequency resolution (~10Hz), so we use a Bark-based perceptual dimensionality reduction [7] on the columns of X to obtain a 512 × 3445 matrix. Following the execution of NMF on this reduced matrix, we

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];
```

is a standardized API that allows for simultaneous execution on multi-core, shared-memory computers [15]. OpenMP is supported in Visual C++ 2005, the Intel compiler, and gcc 4.2 and higher. It has been implemented for C, C++, and Fortran.The benefit of OpenMP is that it can parallelize sequential code that already exists by adding compiler directives to it. In accordance with the guidelines, OpenMP automatically forks threads that run on different processors. With just one directive, OpenMP extremely conveniently parallelizes loops with separate iterations. Using a leading #pragma directive, the element-wise array multiplication displayed below can be divided among nt cores.

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

CUDA

includes the extensions to the C programming language that are used to program the CUDA architecture for general purpose computation, as well as the parallel device architecture used in more recent Nvidia GPUs. The host, or CPU, runs CUDA code that has been compiled with Nvidia'snvcc and then sends commands to the device, or GPU. While device code consists of kernels—functions written to execute in a Single Program, Multiple Data (SPMD) fashion—each thread running on the device during kernel invocation executes the kernel code independently on whatever chunk of data is assigned to the thread, host code typically consists of control flow instructions and memory movement operations between host memory and device memory. Memory can also be shared by groups of threads. With

CUDA 2.1,

```
// 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 run in units called warps, which are collections of 32 neighboring threads. When a group of threads can execute in a fully SIMD (Single Instruction, Multiple Data) fashion—that is, with each thread in the warp doing the same action but on distinct pieces of data—warps operate at their most efficient. The impacted threads must run sequentially rather than concurrently when control flow instructions are inserted into a kernel to cause threads inside the same warp to execute separate code (this is known as a "divergent" warp). Since CUDA does not currently enable double-precision hardware, in this study we concentrate on single-precision implementations. High throughput on highly data-parallel computations is the goal of CUDA's design. Fortunately, the majority of multimedia apps, particularly those for music, show a wide

PARALLEL IMPLEMENTATION

Important Kernels

We break down the updates in eq. (2) into the key computing kernels, such as element-wise vector arithmetic, dense matrix multiplication, and column and row summing, in order to better organize our NMF implementation. Although each kernel will be called one after the other, each one will be highly parallelized and optimized. In terms of floating point operations (flops), the Single-precision GEneral Matrix Multiply, or SGEMM, kernel will perform the most work. Equation (2) requires approximately 423 Mflops for the four SGEMMs, given the matrix dimensions given at the end of Section 3.4. About 3.6 Mflops are needed for the element-divides, 0.1 Mflops for the sums, and 0.1 Mflops for the element-multiplies. Every element in the formula has a little constant (called EPS) added to it to prevent dividing by zero.

OpenMP Implementation

As previously mentioned, OpenMP simplifies the process of parallelizing sequential code for a multi-core shared memory system. We may parallelize the sums and element-wise arithmetic by utilizing the two kinds of for pragmas from Section 4.1. It makes sense to parallelize the element splits' loop because they are many, sluggish, and do not call for inter-thread communication. Parallelizing the reduction loop actually resulted in a slower kernel since the row and column sums demand a lot of communication for the amount of addition work done per core (because the partial sum computed by one core must be transferred to another core). In addition to having a lot of addition, the greater sum in the divergence cost function also involves a sluggish log-based computation, thus the effort to communication

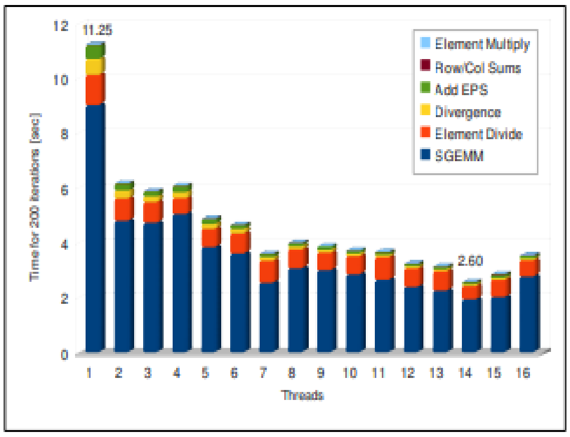

Figure 2. Performance results for the OpenMP implementation on a dual-socket Intel Core i7 920

CUDA Implementation

A CUDA implementation requires extra planning when writing. The matrices must first be copied to the GPU's memory. Avoid using copies between the CPU and GPU unless necessary for initialization or result return, as they are quite sluggish (preferably 3 GB/s over the PCI bus). This means that even while some operations are better suited for the CPU, in our scenario it's preferable to complete all of the matrix computations on the GPU to avoid unnecessary copies. Completely data-parallel, element-wise arithmetic can be performed with ease using code akin to that found in Section 4.2. Some kernels that are not as easily parallelized on CUDA, like the SGEMMs and sums, need some inter-thread communication.

SGEMM

Fortunately, the CUBLAS 2.1 library contains an optimized SGEMM function that, on modern GPUs, delivers 60% of theoretical peak performance for big matrices [17]. 60% of peak for the Geforce GTX 280 translates to 373 Gflops/s. The CUBLAS SGEMM on this GPU achieves 117, 147, and 104 Gflops/s, respectively, for The dimensions of our particular matrix multiplications are [512 × 30 × 3445], [30 × 512 × 3445], and [512 × 3445 × 30]. Even in these relatively modest SGEMM cases, we should be able to do better. Upon reviewing the publication [17], which describes the methods used in the present CUBLAS SGEMM, we discovered that threads function in the matrix sub-blocks of dimensions 16 and 64. Given this, we tried to zero pad our matrices to multiples of 16 so that for each SGEMM, we got 264, 196, and 85 Gflops/s (not including operations on zero-padded sections). 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

We will need to create our own procedures because standard libraries do not have parallel reductions like sums, mins, and maxes. The CUDA SDK contains a lesson on maximizing reductions in CUDA [18]. This overview demonstrates how to obtain a 30× speedup for a 4.2 × 106 length sum over a naive binary tree implementation, and discusses optimization methodologies that can be utilized to dramatically accelerate huge power-of-2-size reductions. There are several techniques to design a binary tree reduction. A thread block's shared memory allows us to carry out a number of two-element reductions. Figure 3 presents two approaches to organizing the overall decrease. Each thread in the thread block in both versions begins by

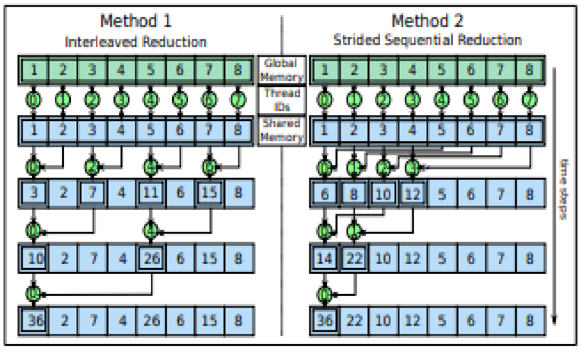

Figure 3. Two methods of shared memory reduction

We can compute all 30 column or row sums at once in order to generate more concurrent work (in terms of thread blocks). To achieve this, one launches a two-dimensional grid of thread blocks, where the thread blocks inside each individual sum are indexed in the second dimension, while the first dimension indicates which of the thirty sums is being computed. As seen in Figure 4, this last improvement resulted in an astounding speedup for the 30 smaller sums.

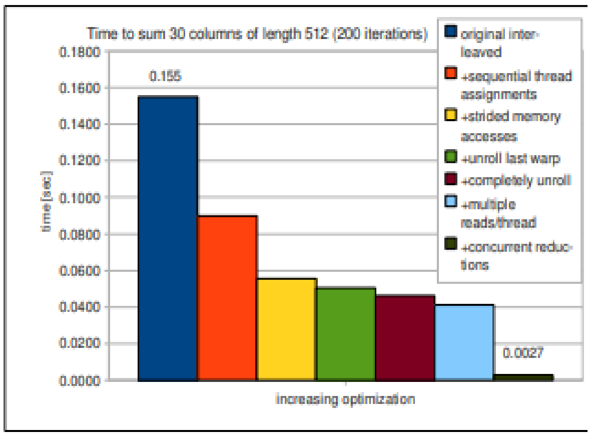

Figure 4. Cumulative effect of various optimizations on running time of 200 iterations of the 30 column sums

CUDA Performance Results

Figure 5 displays the outcomes of the CUDA implementation in comparison to the OpenMP and Matlab implementations. The dimensionality reduction method described in Section 3.4 is applied in the Matlab implementation, which is optimized for singleprecision vector operations. Our Matlab implementation outperforms a simple, non-dimensionality-reduction Matlab implementation by a factor of three. On the same system, the OpenMP version outperforms the Matlab version by a factor of two and exhibits a notable increase in speed when utilizing additional threads on the Core i7. Nevertheless, the non-linear speedup observed between 1 and 14 threads implies that the OpenMP version may not scale well to additional cores. Our CUDA implementation performs admirably on the older Geforce 8600 GTS, which sports four 1.46 GHz multiprocessors. The more recent Geforce

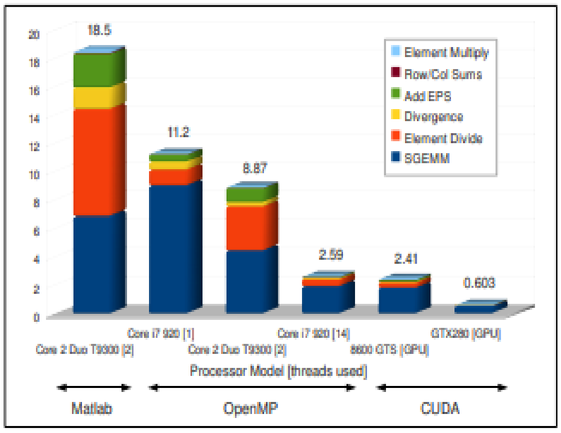

Figure 5. Running time comparison for 200 iterations of 512×30×3445 NMF using optimized implementations in Matlab, OpenMP, and CUDA on different architectures

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.

DISCUSSION AND FUTURE WORK

Parallelizing the remaining steps of the entire source separation process would be the next step after attaining such a large speedup on the NMF phase of percussive source separation. Since individual audio frames may be analyzed independently, these processes are all fairly data-parallel and would benefit from parallelization, much like the majority of signal processing and machine learning algorithms. It's crucial to keep in mind that while CUDA can achieve better performance on more recent GPUs, OpenMP is a better place to start for programmers who are already familiar with C programming. This is especially true when choosing between the two when writing MIR applications. Additionally, we must keep in mind that parallel MIR applications may not always require coding.

REFERENCES

Maryas contributions to MIREX 2007 [1] G. Tzanetakis, MIREX 2007, 2007. Visit this website: http://www.musicir.org/mirex/2008/abs/mirex2007.pdf

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

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

In the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, P. Smaragdis and J. Brown discussed "Non-negative matrix factorization for polyphonic music transcription," which was published in 2003.

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