The Application

• Audio source separation is an important part of Music Information Retrieval
• Drum track extraction is a specific example of source separation and is useful in rhythm summarization, drum transcription, and beat tracking.
• We use Non-negative Matrix Factorization (NMF) as the source separation technique.
• The process:
  - For a 512x3500 spectrogram representing 20 seconds of audio and 30-source NMF:
    - NMF takes 80% of the compute time (18.5 of 23.1 sec) in the Matlab implementation.
    - We will parallelize NMF using OpenMP for multi-core CPUs and CUDA for Nvidia GPUs.

Non-negative Matrix Factorization

• NMF is an optimization problem, and for music a divergence cost function works well:
  - Given an M x N matrix of non-negative X ∈ R^M x N, find matrices W ∈ R^M x K and H ∈ R^K x N that minimize the cost function f(X,W,H).
  - We use multiplicative gradient-based updates:

  \[ H \leftarrow H \cdot \frac{W^T \cdot X}{W^T \cdot H} \]
  \[ W \leftarrow W \cdot \frac{X \cdot H^T}{W^T \cdot H} \]

• Element-wise arithmetic is accomplished using a separate thread for each element.
• Reductions (sums) require most programming effort.
• Reorganize binary tree reduction to avoid divergent warps and memory bank conflicts (as in Method 2).
• Also, loop unrolling, and multiple global memory reads per thread.
• Most speedup comes from running the 30 sums concurrently.

Organizing with Design Patterns

• Example of a design pattern decomposition for one update step on CUDA
  - This helps us organize our code and communicate our computational needs.
  - GEMMs require ~400 MFlops per iteration, while other steps require less than 10 MFlops.
  - But sums require inter-thread communication, and divides are slow.

OpenMP Results

• Intel’s MKL is used for GEMM
• OpenMP for and reduction clauses are used for sums and element-wise arithmetic
• Scaling on dual-socket Nehalem show:
  - 4x speedup over sequential C
  - 7x speedup over Matlab

CUDA Results

• CUDA version runs over 30x faster than Matlab version.
  - 18.6x faster than OpenMP with 14 threads
  - 4.3x faster than sequential C
• Computation time down to 0.6 sec for 20 sec of audio which makes the app much more feasible.

• However, programming in CUDA requires much more effort than OpenMP and Matlab (especially when we need inter-thread communication).
• Programming in low-level CUDA is only worthwhile for important compute-intensive routines

Conclusions

• CUDA can achieve high performance for data-parallel music applications.
• Programmer effort in CUDA is much too great for music applications programmers.

Continued Work

• Developing Python modules of these implementations.
• Potential for Copperhead project to make CUDA more practical for writing music apps.
• Eventually building a DSL or framework to assist in constructing parallel music apps.