Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

SHAPE *AUDIONAMIX*: *A benchmark of linear algebra libraries for HPC*

Sébastien Eggenspieler[a], Guillaume Vincke[b], Pierre Leveau[c]

Audionamix

Abstract

The source separation technology developed by Audionamix requires a significant amount of computation. To increase the speed of the technology, several High Performance Computing libraries for linear algebra have been benchmarked with two applications: a simple matrix multiplication and a Non-Negative Matrix Factorization. The benchmarked libraries were Eigen, Armadillo, MKL and the GPU library Magma. The benchmark returned that for both applications the GPU-based solution performed better than the other solutions. But among CPU-based solutions, Eigen performed better on a complex task than MKL. Thus, making the Magma library as an Eigen back-end seems to have a good potential for HPC-based linear algebra, since this solution would benefit at the same time from the GPU-accelerated computation of Magma and from Eigen's optimized data transfer.

1. Introduction

Audionamix is a technology company developing audio unmixing technologies, which rely on computationally intensive optimization algorithms. The low speed is an impediment for the application of the technology in a number of business cases.
Audionamix is exploring the latest hardware and software solutions to speed up its technology. In order to do so we broke down the task into steps. First, we tried to confirm the relevance of GPU-based computation on more recent hardware. Then, we assessed the most recent solutions (OpenCL, CUDA, Mic/MKL) with respect to the algorithm structure.

2. Preliminary research

The execution time of the Audionamix source separation algorithm can easily take from 2 to 5 times the duration of the input audio file. As a consequence the number of application is limited. Most of the used algorithms are iterative methods that update matrices representing sources (instruments) from an input audio file.

The structure of our algorithm can be roughly represented as such:

a* sebastien.eggenspieler@audionamix.com
b* guillaume.vincke@audionamix.com
c* Pierre.leveau@audionamix.com

```
for iteration in iterationNumberPerStep
        for Matrix in modelMatrices
                UpdateMatrix()
        end
end
```

Figure 1: Structure of the Audionamix source separation algorithm

Considering the structure of the algorithm (*Figure 1*), we found out that the only part where a HPC solution could be used efficiently would be on the linear algebra function applied to the matrices. Indeed, the update rules are composed of basic linear algebra functions such as matrix products, element wise products or matrix sums. These operations are applied to dense matrices and are independent from each other, thus it is perfectly possible to run them in parallel.

Another possibility is to investigate the CPU level parallelization techniques for increasing the speed of the algorithm execution.

For instance the OpenMP/MPI technology enables sequential code dealing with a large amount of data to share the work between all computation unit available in order to improve its execution speed. However, considering the relative computation speed and the amount of data to process, the transfer time between CPU over a network is most likely to slow down the execution speed of the source separation algorithm.

The Intel MIC technology can also be considered. MIC is a multi-core-hardware close to the GPU architecture. This technology however is still at an early stage and its community not quite developed yet.
Considering this, we decided to restrict our tests on CPU technologies to the Intel® Math Kernel Library [4].

When dealing with GPUs, OpenCL and Nvidia CUDA are the main candidates. GPU programming allows to send data to a graphic card in order to exploit its significant power of computation. OpenCl and CUDA are quite similar in terms of performances. OpenCL asset remains in its portability and Nvidia CUDA in its specialization on Nvidia GPUs and its relative easy accessibility. Luckily, the Magma library [2] gives an easy-to-use interface for CUDA with an ongoing support for OpenCL.

Finally, some libraries handle the linear algebra operations at the matrix level, such as Eigen [3] and Armadillo [1]. The behavior of these libraries has also been studied.

3. Benchmarking

The benchmark is composed of two test applications. Each library is tested on both applications and the elapsed time of execution measured.

The libraries used for the benchmark are the CPUbased Eigen and Armadillo libraries, and a Intel MKL wrapped in higher-level functions (matrix multiplications, element-wise operations). For the GPU libraries benchmark, we used Nvidia cuBLAS through the linear algebra library Magma.

The test environment for the benchmark is the following:

- CPU: 2 x Quad-Core Xeon 2,66Ghz - 8Go ram
- GPU: Nvidia GeForce GTX 560ti

The Intel Xeon processors are optimized to run efficiently with the Intel MKL. Considering this specificity, we expect the MKL to produce honourable results.

3.1 First test application: Matrix Multiplication

The first test application is a set of square matrix multiplication of growing dimensions by steps of 1000.

```
C(M,M) = A(M,M) * B(M,M)
```

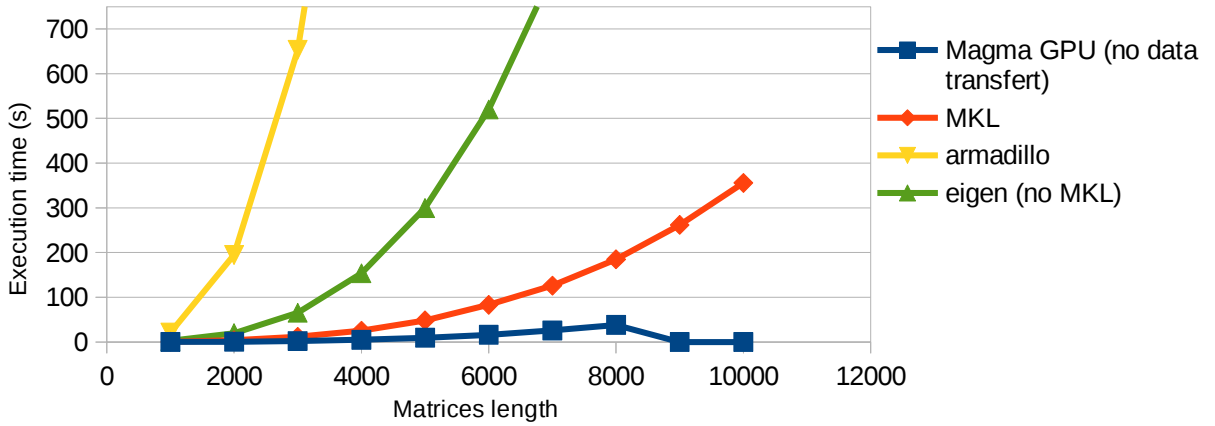

Figure 2: Performances of the matrix multiplication operation

As expected, the Intel MKL appears to be the fastest CPU based solution compared to the native Armadillo and Eigen operations, but it is quickly surpassed by the GPU-based library Magma (*Figure 2*).

We however have to consider that we are only measuring the computational speed. This first benchmark does not take into account the communication time generated by the data transfer to and from the GPU with Magma-CUDA calls.

Furthermore Magma seems to break when the matrices dimensions become too large, between 8000^2 and 9000^2. This issue is due to the fact that the matrices are too large to be allocated on the GPU, so the function cannot be executed properly. Considering the targeted use case (one dimension relates to the length of an audio file), this memory limitation could be an issue since it will force to call supplementary data transfer procedures, thus slowing down overall computational speed.

3.2 Second test application: Non-Negative Matrix Factorization

The second test consists of running a classical algorithm involved in source separation: the Non-Negative Matrix Factorization [6].

The following presents the structure of the second benchmark, a common iterative part and the called functions, adapted for each libraries used in the benchmark.

`For each:`

```
unsigned int nbIteration = 50;
{
   for(unsigned int iteration = 0; iteration < nbIteration; iteration++){
      updateW(H, X, &W);
      updateH(W, X, &H);
   }
}
```

The update rules functions are then adapted to each of the libraries.

`Custom wrapping of MKL and Magma:`

```
namespace mkl{ // replace mkl by magma to get magma call
    template<class Mat>
    void updateW(const Mat& H, const Mat& X, Mat* W){
        Mat tot;
        {
            Mat numerator, denumerator;
            Mat temp;
            MatMulMat(*W, H, &temp);
            MatElemPowerMat(temp, -2, &numerator);
            MatElemPowerMat(temp, -1, &denumerator);
            MatElemMulMat(numerator, X, &temp);
            MatMulMatT(temp, H, &numerator);
            MatMulMatT(denumerator, H, &temp);
            MatElemDivMat(numerator, temp, &tot);
        }
        MatElemMulMat(*W, tot, W);
    }
```

```
    template<class Mat>
    void updateH(const Mat& W, const Mat& X, Mat* H){
        Mat tot;
        {
            Mat numerator, denumerator;
            Mat temp;
            MatMulMat(W, *H, &temp);
            MatElemPowerMat(temp, -2, &numerator);
            MatElemPowerMat(temp, -1, &denumerator);
            MatElemMulMat(numerator, X, &temp);
            MatTMulMat(W, temp, &numerator);
            MatTMulMat(W, denumerator, &temp);
            MatElemDivMat(numerator, temp, &tot);
        }
        MatElemMulMat(*H, tot, H);
    }
}
```

Armadillo:

```
namespace armadillo{
    template<class Mat>
    void updateW(const Mat& H, const Mat& X, Mat* W){
        Mat WH = (*W)*H;
        *W = (*W) % (((arma::pow(WH, -2) % X) * H.t()) / (arma::pow(WH, -1) * H.t()));
    }

    template<class Mat>
    void updateH(const Mat& W, const Mat& X, Mat* H){
        Mat WH = W*(*H);
        *H = (*H) % (((W.t() * (arma::pow(WH, -2) % X)) / (W.t() * arma::pow(WH, -1))));
    }
}
```

Eigen:

```
namespace eigen{
    template<class Mat>
    void updateW(const Mat& H, const Mat& X, Mat* W){
        Mat WH = ((*W)*H);
        *W = W->array() * (((WH.array().pow(-2).array() * X.array()).matrix() *
H.transpose()).array() / (WH.array().pow(-1).matrix() * H.transpose()).array());
    }

    template<class Mat>
    void updateH(const Mat& W, const Mat& X, Mat* H){
        Mat WH = (W*(*H));
        *H = H->array() * ((W.transpose() * (WH.array().pow(-2).array() *
X.array()).matrix()).array() / (W.transpose() * WH.array().pow(-1).matrix()).array());
    }
}
```

The results of this second benchmark are presented below.

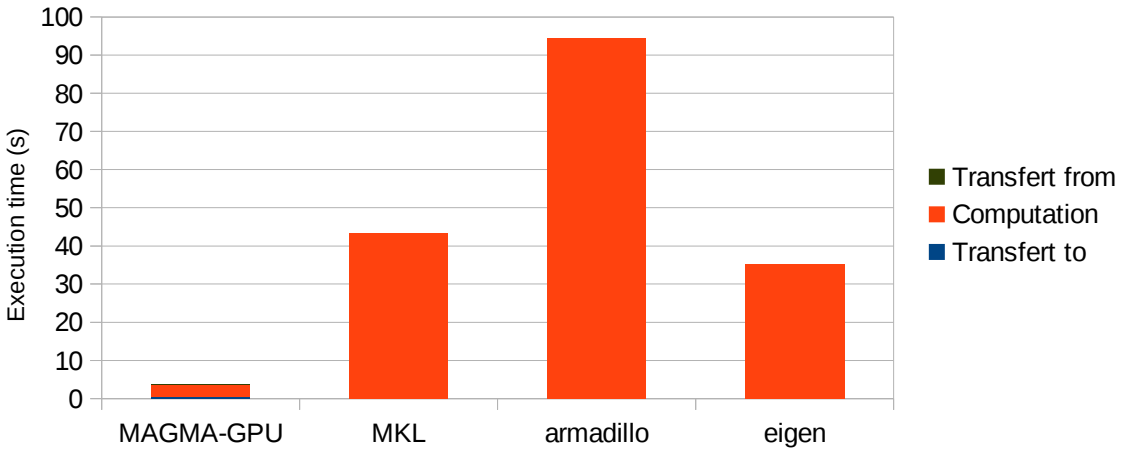

Figure 3: Performances of the NMF update

This time, the Eigen library gives the best results among the CPU solutions. We can guess that Eigen has been better optimized in its memory handling compared to the custom wrapper. But even when the memory handling is optimized, here again, GPU Magma outperforms the other libraries.

To get a better idea of Magma performances, here is a close-up of the algorithm:

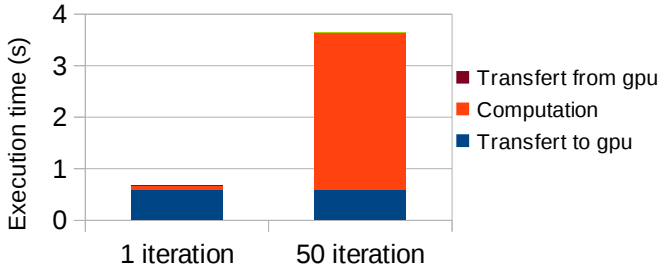

Figure 4:Execution time with Magma for 1 and 50 iterations

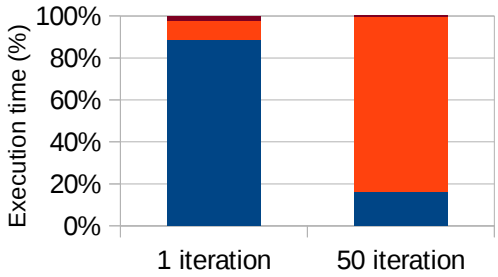

Figure 5: Magma execution ratio by steps

The time taken to transfer data to and from the GPU shows a considerable loss of computation time that cannot be ignored (*Figure 4 and 5*). Regardless of the iteration number, the data transfer time remains unchanged. However the ratio “data transfer time / execution time” decreases when increasing the iteration number (*Figure 5*). This proves that it is possible to reduce the impact of such a loss by reducing the data transfer calls to the minimum and perform as much computation as possible on the GPU data before sending them back. But the memory capacity of the GPU also needs to be taken into account since it will limit the amount of data that can be store on the GPU as observed from the first test application.

4. Conclusion

Considering the results of the benchmark (*Figure 3*), it appears that a Magma-GPU based solution would be the answer to our speed expectation. However, the Magma syntax is heavy, even though it appears to be a little more convenient than CUDA's. On the other hand, Eigen is a high-level optimized library that proposes a syntax close to Matlab. There is a fork called Eigen-Magma [5] that interfaces Eigen in order to use GPU Magma functions instead of the CPU ones. However this same interfacing of libraries was not optimized since for each operation the matrices used were sent to the GPU, used in the function then sent back to the CPU. The transfer process is so time-consuming, as previously explained, that it may hinder the interest of using GPUs. For example, the benchmark of the library found on Eigen-Magma's webpage shows that this implementation is slower that the CPU implementation on some functions and doesn't give as impressive results as the ones we got in our benchmark. Nevertheless, the perspective of using the Eigen syntax with Magma's performances seems to be the most promising approach to get a perfect balance between progamming paradigm and high speed.

References

[1] Armadillo project site: http://arma.sourceforge.net/
[2] Magma project site: http://icl.cs.utk.edu/magma/index.html
[3] TuxFamily's Eigen project site: http://eigen.tuxfamily.org/index.php?title=Main_Page
[4] Intel MKL: https://software.intel.com/en-us/intel-mkl
[5] Eigen-Magma fork: https://github.com/bravegag/eigen-magma
[6] Lee, Daniel D., and H. Sebastian Seung. "Learning the parts of objects by non-negative matrix factorization." *Nature* 401.6755 (1999): 788-791.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.