Heterogeneous Computing for Signal and Data Processing

E4750_2017Fall_SRBISR_report
Xiaotong Qiu xq2174,Yameng Xiao yx2417,Xiaotian Hu xh2332
Columbia University

Abstract:
Super resolution(SR) is one of the popular image reconstruction technology. One of the approach to it is sparse presentation method. This method learns a low-resolution dictionary and high-resolution dictionary pair to store the high-resolution image. Out team implemented the dictionary learning algorithm for super resolution. We focused on the most time-consuming part of the dictionary-learning algorithm, the K-SVD algorithm and tried to parallel the core part of the K-SVD: Orthogonal Matching Pursuit. The parallelization faced two challenges: the computation scale of the OMP is not fit into a single thread nor a whole GPU. And the memory access in the computation process of OMP is random. We used the batch technology and write series of kernel intended for loading data to solve these two problems. Our program succeeded in getting dictionary that can be used in super resolution. And our parallelization version K-SVD shows a 10 times improvement in speed when compared with sequential version K-SVD. In this report, we discuss the implementation of our program and show some experiment results to show the performance improvement we achieved.

1 1. Overview

1.1 Problem in a Nutshell

Super resolution(SR)[3, 7, 4, 5]is a popular image reconstruction problem, it reconstructs the high-resolution image from the low-resolution image. This technology has vast application field including medical imaging, satellite imaging, pattern recognition, etc.

Dictionary-based method[3, 4] is one of the widely talked method to this problem. It takes a training dataset consists of known low-resolution and high-resolution image pairs and get image patch signals with partition and some successive image processing. Based on this patch signals, it tries to learn a low-resolution dictionary A_l and high-resolution dictionary A_h which can represent image patch signals as:

$$p_k^h = A_h q_k \tag{1}$$

$$p_k^l = A_l q_k \tag{2}$$

where x_l and x_h are the low-resolution and high-resolution image patch signal and α is the sparse representation. α is a sparse vector with only a few non-zero elements. A_r and A_l are consisted of vectors of the same length of patch signals and these vectors are called atom. Patch signals are actually represented as the linear combination of those atoms and α is the combination coefficients. Since alpha is sparse[3], only a few atoms will be used to form the patch signal, hence, the representation alpha is called spares representation. After getting the dictionary pair A_r and A_l, with the assumption that the corresponding low-resolution patch signal and high-resolution patch signal can be represented as the same sparse representation α with A_r and A_l respectively. The high-resolution image can be reconstructed by computing the spare representation of the low-resolution image patches, mapping their sparse representation, getting the corresponding high-resolution patches and stitching them to get the final output.

The key step of this method is learning the dictionary pairs A_r and A_l, one of the algorithm is K-SVD[1]. It iteratively learns the dictionary from the training data. For each iteration, K-SVD firstly compute the spares representation with current dictionary for training signals with some pursuit algorithm and then uses the representation to refine the dictionary to decrease the error of representation. The refining process involves SVD, which gives it the name, K-SVD. In SR problem, K-SVD needs massive computation because the training process usually uses millions of patch signals.

The computation of K-SVD can be reduced by replacing SVD computation in every step with simple but coarse computation in every iteration.[2, 6]. It accelerate the K-SVD method with only a little precision loss.

Our project focus on parallelizing the key procedure the dictionary learning process: K-SVD. And to achieve better speed performance, the approximated K-SVD[6] is used. And the other part of the dictionary-based super resolution algorithm, which only take a small part of whole computation is implemented in sequential fashion.

1.2 Prior Work

Article [3] gives the general framework of dictionary-based approach to the super resolution. And the K-SVD algorithm is detailed described in the article [1]. And the article [6] comes up with the approximated K-SVD algorithm.

2 Description

In this section, we elaborate the details about the implementation of the dictionary learning program. We will first describe the algorithm and then point out the most time-consuming part of the program, the K-SVD, by analyzing the computational complexity. Finally we will discuss in detail about how to parallelize the K-SVD.

2.1 Objectives and Technical Challenges

Implement the parallelized version of dictionary learning algorithm for SR, including Two objectives are derived:

1) contdesign the dictionary learning program whose routines include extracting patches signals, training the low-resolution dictionary and computing the high-resolution dictionary.
2) parallel the K-SVD part of the dictionary learning program. It involves in parallelizing Orthogonal Matching Pursuit algorithm and handling random memory ac-

cessing issues.

2.2 Problem Formulation and Design

The dictionary learning algorithm tries to find the optimal dictionary pair Al and Ar and sparse representation for training signals that minimize the representation error. It can be formulated as the optimization problem:

$$\underset{A_l, q_k}{\operatorname{argmin}} \quad \sum_k \|A_l q_k - h_k^l\|_2 \tag{3}$$

$$p_k^h = A_h q_k \quad and \quad \|q_k\|_0 << T \tag{4}$$

Where p_k^h and p_k^l is the patch signal from high-resolution images and low-resolution images, and the T is the sparse coefficients which constrains the largest number of non-elements in the sparse representation q_k

Dictionary learning routine can be generally divided into 3 steps, which is depicted by flow chart figure 1

1) extract the patch signals from the training images
2) train the low-resolution dictionary using K-SVD
3) compute the high-resolution dictionary from the low-resolution dictionary.

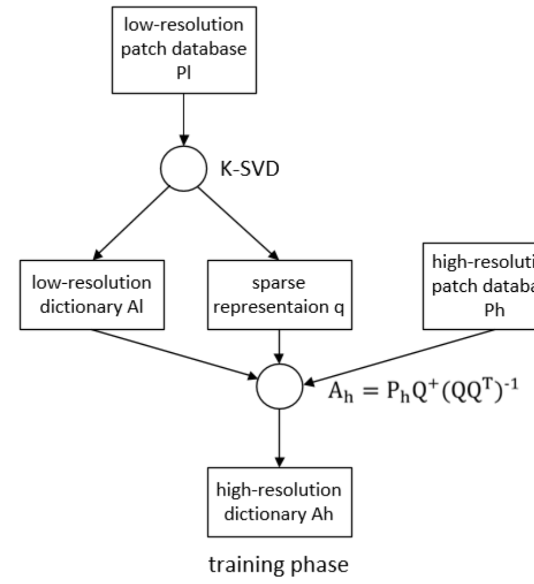
low-resolution patch database Pl
K-SVD
low-resolution dictionary Al
sparse representaion q
high-resolution patch database Ph
$A_h = P_h Q^+ (QQ^T)^{-1}$
high-resolution dictionary Ah
training phase

Figure 1: The flow chart of the dictionary learning process. It firstly extract the patches signals, and then learn the low-resolution dictionary. Finally, compute the high-resolution dictionary with a close-form expresssion.

The training process takes around 20 images and extract over 20,000 patch signals for training. This process needs a great amount of computation, most of which is caused by K-SVD. The K-SVD is used to get the low-resolution dictionary, in other word, to solve the sub-problem of problem equation 5

$$\underset{A_l, q_k}{\operatorname{argmin}} \quad \sum_k \|A_l q_k - h_k^l\|_2 \tag{5}$$

In each iteration of K-SVD, it computes the sparse representations of all training signals under current dictionary with Orthogonal Matching Pursuit(OMP) algorithm and then use the sparse representation to refine the dictionary. The pseudo code for K-SVD is shown algorithm 1

Algorithm 1 Approximate K-SVD

```
Input: Singal set $\mathbf{X}$, initial dictionary $\mathbf{D_0}$, target sparsity $K$, number of iterations $k$
Output: Dictionary $\mathbf{D}$ and sparse matrix $\mathbf{\Gamma}$ such that $\mathbf{X} \approx \mathbf{D\Gamma}$
Init Set $\mathbf{D} \leftarrow \mathbf{D_0}$
for $n = 1, \ldots, k$ do
    $\mathbf{\Gamma} = \mathrm{OMP}(\mathbf{X}, \mathbf{D})$
    for $j = 1, \ldots, L$ do
        $\mathbf{D}_j \leftarrow 0$
        $I \leftarrow \{$ indices of the signals in $\mathbf{\Gamma}_i$ whose $i$ element is non-zero $\}$
        $g \leftarrow \mathbf{\Gamma}_{j,I}^T$
        $d \leftarrow \mathbf{X}_I g - \mathbf{D\Gamma}_I g$
        $d \leftarrow d / \|d\|_2$
        $g \leftarrow \mathbf{X}_I^T d - (\mathbf{D\Gamma_I}^T) d$
        $\mathbf{D}_j \leftarrow d$
        $\mathbf{\Gamma}_{j,I} \leftarrow g^T$
    end for
end for
```

The main cause of the great amount of computation of K-SVD is that it will perform over 10,000 OMP in each iteration. Hence, the acceleration of the OMP procedure is the key to accelerate the entire K-SVD algorithm. In the next paragraph, the computation complexity of OMP will be analyzed in detail.

Algorithm 2 Orthogonal Matching Pursuit(OMP)

```
Input: Dictionary $\mathbf{D_0}$, batch of signals $X = \{x_i\}$, target sparsity $T$
Output: Sparse Representations $\Gamma$
Init Set $I \leftarrow \{\}$, $R \leftarrow X, \Gamma \leftarrow 0$
for $i = 1, \ldots, T$ do
    $\mathbf{K} \leftarrow \mathbf{D}R$
    $k^* \leftarrow \max\{\mathbf{K}\}$
    $I \leftarrow (I, k^*)$
    in batch solve $x_i = \mathbf{D}_I r_i$
    in batch compute $r_i \leftarrow x_i - \mathbf{D}_I \gamma_I$ where $R = r_i$
end for
```

Algorithm 2 is the pseudo code of OMP algorithm. The operations of iteration include computing serval times of inner products, solving a overcomplete linear system and performing a matrix multiplication and subtraction. For a typical super resolution problem, low-resolution is consisted of over 1000 atoms and the size of each atom vector is around 50. Besides, each sparse representation will contain less than 5 non-zeros elements. Hence, for each iteration of OMP, there are over 1000 inner products, a linear system with size of 30×5 and matrix multiplication and subtraction with matrices size of 30×5 matrices in total.

The scale of computation for OMP brings a challenge: it is too large to fit into a single CUDA thread, is also not large enough to use a CUDA grid to compute, which will cause a great waste of GPU resources. Therefore, the batch technology is used. With batch, GPU resource can be shared by more than one streaming each of which computes a linear system solver or matrix multiplication for one signal OMP operation.

Besides the resource allocation, the memory access pattern of OMP operation also raises a challenge. For each iteration of single OMP, the program loads the dictionary atom corresponding to the largest inner product, which means which atoms to be loaded is not determined until the run time, depicted by figure 2

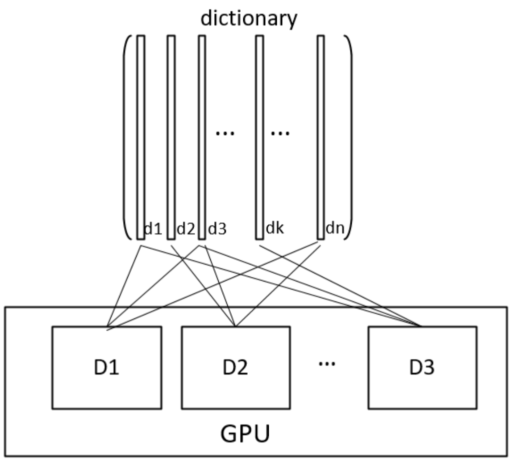

Figure 2: The memory access pattern of OMP. The dictionary atoms will be loaded into device and which atoms will be loaded is not determined until the running time.

The atom selecting and loading tasks are executed by CPU with cudaMemcpy function in traditional way. It will cause a great number of memory access between host and device, which is an intolerably time-consuming. To mitigate the impact of random memory access, we write series of dedicated GPU kernel programs to do the atom loading.

2.3 Software Design

To make the report concise, the pseudo code for the whole dictionary learning program is placed in the appendix.A, and we focus the parallel part of our code in this section. The algorithm 1 describes K-SVD and the algorithm 3 shows the parallelized version OMP.

Algorithm 3 Parallelized Orthogonal Matching Pursuit(OMP)

```
Input: Dictionary $\mathbf{D_0}$, batch of signals $X = \{x_i\}$, target sparsity $T$
Output: Sparse Representations $\Gamma$
Init Set $I \leftarrow \{\}$, $R \leftarrow X, \Gamma \leftarrow 0$
for $i = 1, \ldots, T$ do
    $\mathbf{K} \leftarrow \mathbf{D}R$
    $k^* \leftarrow \max\{\mathbf{K}\}$
    $I \leftarrow (I, k^*)$
    in batch solve $x_i = \mathbf{D}_I r_i$
    in batch compute $r_i \leftarrow x_i - \mathbf{D}_I \gamma_I$ where $R = r_i$
end for
```

The parallelized OMP processes one batch of signals for each run. It has 5 steps for each iteration. We will discuss the implementation details of these 5 steps respectively.

The first step is to compute the inner products between the dictionary atoms and the signal vectors. This operation can be realized with a single matrix multiplication as shown in the pseudo code, where $\mathbf{X}$ is the combination of batch signals whoes column corresponding to those patch signal.

The second and third steps are to find the position of the atom with the max inner production and assign the found indices into the atom loading pointer I. These two steps are implemented within a kernel program, which find the max element of each column of input matrix and set the corresponding element to 1 in the output matrix.

Step 4 solves linear system for each signal in batching fashion. Before performing the computation, the corresponding matrix $\mathbf{D}_I$ and x_i should be loaded into device according to the atom loading pointer I.

Step 5 calculates the current residues for each signal between its representation and itself, and, the same as the fourth step, it also should load the related data into device.

We write the dedicated kernel to do the data loading job for step 4 and step 5, and use the cuBlas API to solve the linear system.

At the end of the OMP iteration, the output should be converted from dense format into sparse format. Another kernel has been written to perform this conversion.

3 Results

3.1 Dictionary Learning Algorithm Result

In our project, we firstly implemented the pure python version of dictionary learning algorithm. It takes 20 images as training dataset and output a dictionary with 1024 atoms each of which is a vector of size 30. The dictionary can used to restore the high -resolution images, shown as the following figure 3

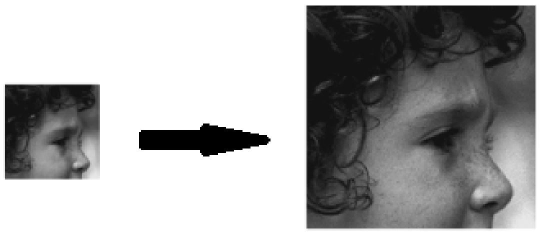

Figure 3: The high-resolution image is successfully restores from low-resolution image with our dictionary

3.2 Performace Comparison of K-SVD

We tested our parallelized K-SVD algorithm on the tesseract server. This server equipped an 8-cores 8 Intel(R) Xeon(R) CPU of clock frequency 1.8GHz, and Tesla k40c GPU. We feed the K-SVD program with the data extracted from out dictionary learning program. In the testing scenario, the K-SVD takes 8172 patches signals vector of size 30 and output a dictionary of size 30×1024. It takes the CPU version program 63.044 seconds while takes GPU version only 6.664 seconds, which means the parallelization brings around 10 times performance improvement in speed.

3.3 impact of batching and dedicated kernel

During our programming work, the first version of parallelized OMP did not use the batch technology and dedicated data loading kernel. This program used CPU to select the atom to load and transferred the atom from CPU to GPU. It achieves a poor performance with only 2 times improvement in speed. The nvvp analyze result for the program is shown below in figure 4

It can be observed the running time of this program is divided into 3 segments which corresponds to 3 iteration. The memory copy from host to device and from device to host takes the most time of computation, which means the frequent memory interaction between the host and device lowered the program speed to a great extent.

Beside the cudaMemcpy API, the imax_kernel took the second largest portion of the program running times. This kernel was used to find the atom corresponding the max inner product for every signal. It needs to be ran thousands of times in each iteration in our case, and, even worse, it needs call cudaMemcpy API for each run.

This program was modified with batching and dedicated kernel. We write a new kernel to find the maximum for all signal simultaneously and designed series of kernel to move data between different bucks of device memory. The performance of the program dramatically changed after the modification. It achieves around 180 times of improvement in speed when compared with the CPU version OMP. The nvvp analysis is shown in below figure 5

It can be seen that with those new kernels, all computations are performed on the GPU and therefore no data interaction between device and host happens during the iterations. The maximum finder kernel costs much less computation time and is not the bottleneck of the speed anymore.

In conclusion, by using batching kernel and dedicated data loading kernel and put all computation onto the GPU, the parallelized program achieves a significant speed acceleration.

4 Discussion and Further Work

We succeed in parallelizing the K-SVD and achieve an around 10 times improvement in speed. However, this program can be further optimized with respect of speed. On the one hand, besides the OMP part of K-SVD, the atom refinement phase can also be parallelized. It has the similar operation like OMP and can also be speeded up with batching technology and dedicated kernel for loading. On the other hand, there exists parallelized code outside the K-SVD in the dictionary learning program including principle component analysis(PCA) process in the patch- signal- extracting routine.

5 Conclusion

In our project, we succeed in implementing the dictionary learning algorithm and parallelizing the core of this algorithm K-SVD. With the parallelized version of K-SVD, the speed of the dictionary training process was improved to a large extent. To parallelize the K-SVD, we learned the batching technology that can make GPU run several kernels simultaneously. After implementing and comparing the performance of parallel version programs implemented with cudaMemcpy function and dedicated kernel respectively, we noticed the memory access between host and device is usually the most time-consuming part of the whole program. To achieve better performance, the CUDA program should avoid the data interaction between host and device.

Although our work has improved the speed of dictionary training algorithm, there exists other processes in the whole program which can be parallelized to achieve further improvement such as the principle component analysis in the patch- signal- extraction phase.

6 Appendix.A

The pseudo code for the overall dictionary learning

Algorithm 4 Dictionary-Learning Process

```
STEP 1. Load training images.
STEP 2. Converts images from RGB color space to YCbCr color space and save the illuminance(Y) value.
STEP 3. Crop the images and save as high resolution images set.
STEP 4. Downsampling the high resolution images to get low resolution images set.
STEP 5. Upsampling the low resolution images to get middle resolution images set.
STEP 6. Call collect function to extract features for each image in middle resolution set and get features matrix.
STEP 7. Upsampling the low resolution images to get interpolated images set.
STEP 8. Subtract each images in high resolution set and in interpolated images set to get patches set.
STEP 9. Call collect function to extract features for each image in patches set and get patches matrix.
STEP 10. Implement dimensionality reduction on features matrix based on PCA and get features_pca matrix.
STEP 11. Call ksvd function using features_pca to train for the low resolution dictionary as well as gamma matrix.
STEP 12. Calculate high resolution dictionary $D_h$ using formula : $D_h = PQ^T(QQ^T)^{-1}$ (P: patches matrix Q: gamma matrix)
```

References

[1] M. Aharon, M. Elad, and A. Bruckstein. "$rm K$-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation." In: *IEEE Transactions on Signal Processing* 54.11 (Nov. 2006), pp. 4311–4322. ISSN: 1053-587X. DOI: 10.1109/TSP.2006.881199. URL: http://ieeexplore.ieee.org/document/1710377/ (visited on 10/31/2017).

[2] Paul Irofti and Bogdan Dumitrescu. "GPU parallel implementation of the approximate K-SVD algorithm using OpenCL." In: *Signal Processing Conference*. 2014, pp. 271–275. URL: http://ieeexplore.ieee.org/document/6952053/.

[3] Jianchao Yang et al. "Image Super-Resolution Via Sparse Representation." In: *IEEE Transactions on Image Processing* 19.11 (Nov. 2010), pp. 2861–2873. ISSN: 1057-7149. DOI: 10.1109/TIP.2010.2050625. URL: http://ieeexplore.ieee.org/document/5466111/ (visited on 12/17/2017).

[4] Kaibing Zhang et al. "Multi-scale dictionary for single image super-resolution." In: IEEE, June 2012, pp. 1114–1121. ISBN: 978-1-4673-1228-8 978-1-4673-1226-4 978-1-4673-1227-1. DOI: 10.1109/CVPR.2012.6247791. URL: http://ieeexplore.ieee.org/document/6247791/ (visited on 12/17/2017).

[5] Kamal Nasrollahi and Thomas B. Moeslund. "Super-resolution: a comprehensive survey." In: *Machine Vision & Applications* 25.6 (2014), pp. 1423–1468. DOI: 10.1007/s00138-014-0623-4. URL: http://link.springer.com/article/10.1007/s00138-014-0623-4.

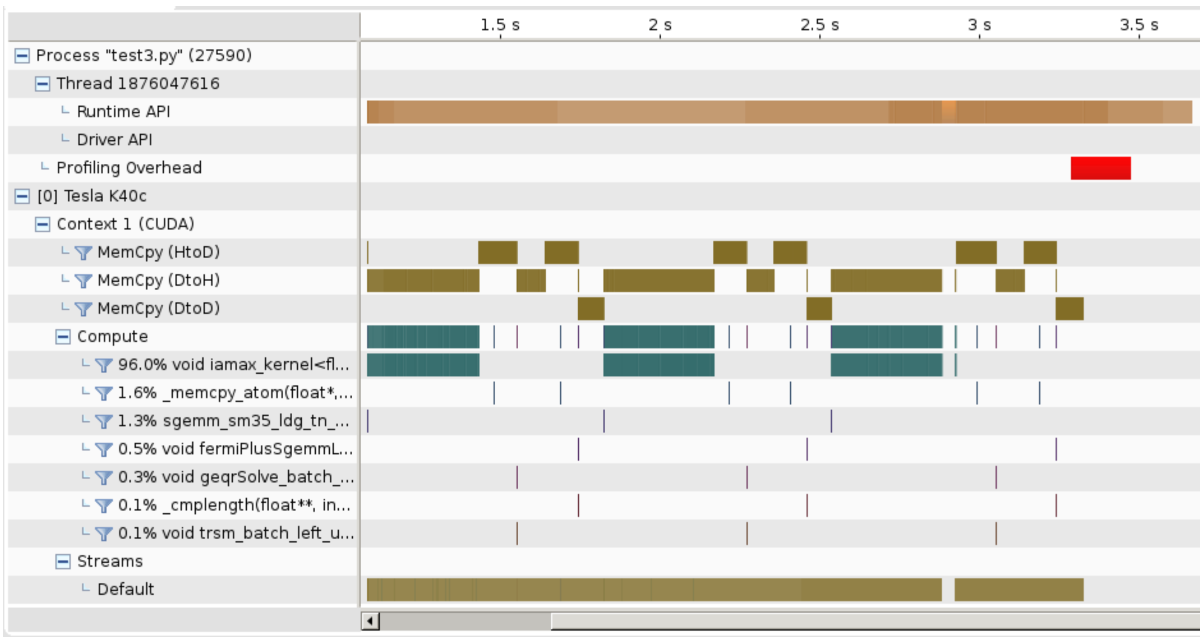
1.5 s
2 s
2.5 s
3 s
3.5 s
Process "test3.py" (27590)
Thread 1876047616
Runtime API
Driver API
Profiling Overhead
[0] Tesla K40c
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
96.0% void iamax_kernel<fl...
1.6% _memcpy_atom(float*,...
1.3% sgemm_sm35_ldg_tn_...
0.5% void fermiPlusSgemmL...
0.3% void geqrSolve_batch_...
0.1% _cmplength(float**, in...
0.1% void trsm_batch_left_u...
Streams
Default

Figure 4: The nvvp analysis of the naive parallelized OMP

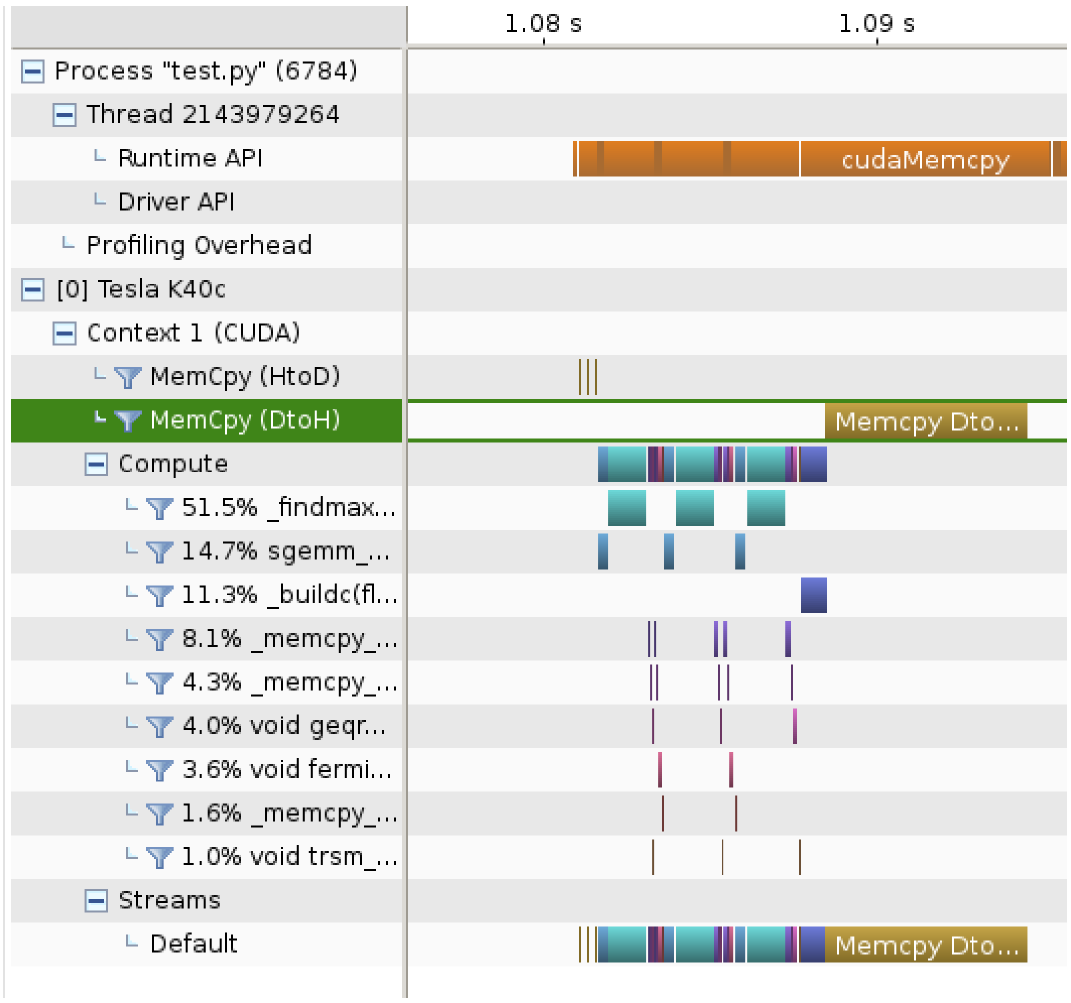

Figure 5: The nvvp analysis of the imporved parallelized OMP

[6] Ron Rubinstein, Michael Zibulevsky, and Michael Elad. "Efficient Implementation of the K-SVD Algorithm Using Batch Orthogonal Matching Pursuit." In: *Cs Technion* 40 (2009). URL: http://www.researchgate.net/publication/251229200_Efficie.

[7] Radu Timofte, Vincent De, and Luc Van Gool. "Anchored Neighborhood Regression for Fast Example-Based Super-

Resolution." In: IEEE, Dec. 2013, pp. 1920–1927. ISBN: 978-1-4799-2840-8. DOI: 10.1109/ICCV.2013.241. URL: http://ieeexplore.ieee.org/document/6751349/ (visited on 10/11/2017).