procedure CUDAMULTIGPUCONVOLUTION($P$, $G$, $signal$, $kernel$, $K$, $L$, $M$, $norm$)

$signal' \leftarrow \text{Decompose}(signal, P)$

$kernel' \leftarrow \text{Decompose}(kernel, P)$

for $l \leftarrow 0, G - 1$ do

\quad selectDevice($l$)

\quad CudaDecomposedConvolution($P/G, signal'_l, kernel'_l, K/G, L, M, norm$)

end for

$signal \leftarrow \text{Compose}(signal', P)$

end procedure

Fig. 6. Multi-GPU Convolution. The for loop is performed on all the available graphic cards in parallel.

procedure CULOCKEDMEMCPY($to$, $from$, $direction$)

while $lock$ do

\quad Wait()

end while

$lock \leftarrow \text{true}$

$\text{cuMemcpy}(to, from, direction)$

$lock \leftarrow \text{false}$

end procedure

Fig. 7. Data transfer with the bus lock. A new (global) variable called $lock$ is introduced. This variable (or a flag) is shared across all the GPUs sharing the common data bus so that each time a GPU needs to send or receive data, it waits until the bus is free, then it locks the bus, transfers data and finally unlocks the bus.

specifically sized and the arbitrarily sized images. The results are presented in Tables IV and V and in Fig. 8(b) and 8(d).

The results reveal several facts. Again, the GPU implementation is up to eight times faster than the CPU c2c implementation (depending on image sizes) and up to three times faster than the CPU r2c implementation when comparing two GPUs with two CPU cores. Also a test with four CPU cores was made and the GPU implementation has performed still two times faster if the images were large enough. Generally, the GPU implementation becomes advantageous on images larger than 50 megavoxels.

Note that in some cases a single GPU has performed better than two GPUs (especially in the case of