

Fine-tuned high-speed implementation of a GPU-based median filter.

Gilles Perrot · Stéphane Domas · Raphaël Couturier

Received: date / Revised: date

Keywords median, filter, GPU

Abstract Median filtering is a well-known method used in a wide range of application frameworks as well as a standalone filter, especially for *salt-and-pepper* denoising. It is able to highly reduce the power of noise while minimizing edge blurring. Currently, existing algorithms and implementations are quite efficient but may be improved as far as processing speed is concerned, which has led us to further investigate the specificities of modern GPUs. In this paper, we propose the GPU implementation of fixed-size kernel median filters, able to output up to 1.85 billion pixels per second on C2070 Tesla cards. Based on a Branchless Vectorized Median class algorithm and implemented through memory fine tuning and the use of GPU registers, our median drastically outperforms existing implementations, resulting, as far as we know, in the fastest median filter to date.

1 Introduction

First introduced by Tukey in [?], median filtering has been widely studied since then, and many researchers have proposed efficient implementations of it, adapted to various hypothesis, architectures and processors. Originally, its main drawbacks were its compute complexity, its non linearity and its data-dependent runtime. Several researchers have addressed these issues and designed, for example, efficient histogram-based median filters featuring predictable runtimes [? ?]. More recently, authors have managed to take advantage of the newly opened perspectives offered by modern GPUs, to



develop CUDA-based filters such as the Branchless Vectorized Median filter (BVM) [? ?] which allows very interesting runtimes and the histogram-based, PCMF median filter [?] which was the fastest median filter implementation to our knowledge.

The use of a GPU as a general-purpose computing processor raises the issue of data transfers, especially when kernel runtime is fast and/or when large data sets are processed. In certain cases, data transfers between GPU and CPU are slower than the actual computation on GPU, even though global GPU processes can prove faster than similar ones run on CPU. In the following section, we propose the overall code structure to be used with our median kernels. For more concision and readability, our coding will be restricted to 8 or 16 bit gray-level input images whose height (H) and width (W) are both multiples of 512 pixels. Let us also point out that the following implementation, targeted on Nvidia Tesla GPU (Fermi architecture, compute capability 2.x), may easily be adapted to other models e.g. those of compute capability 1.3.

2 General structure

Algorithm 1 describes how data is handled in our code. Input image data is stored in the GPU's texture memory so as to benefit from the 2-D caching mechanism. After kernel execution, copying output image back to CPU memory is done by use of pinned memory, which drastically accelerates data transfer.

Algorithm 1: Global memory management on CPU and GPU sides.

```
1 allocate and populate CPU memory h_in;  
2 allocate CPU pinned-memory h_out;  
3 allocate GPU global memory d_out;  
4 declare GPU texture reference tex_img_in;  
5 allocate GPU array array_img_in;  
6 bind array_img_in to texture tex_img_in;  
7 copy data from h_in to array_img_in;  
8 kernel  gridDim,blockDim  /* to d_out  
   */;  
9 copy data from d_out to h_out ;
```
