CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–15
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

An optimized GPU-based 2D convolution implementation

Gilles Perrot, Stéphane Domas, Raphaël Couturier

FEMTO-ST institute, Rue Engel Gros - 90000 Belfort, France. forename.name@univ-fcomte.fr

SUMMARY

With the increasing sophistication of image processing algorithms, and due to its low computation complexity, convolution should fully benefit from the ever-increasing capacities of state-of-the-art GPUS, such as Nvidia's Kepler and Maxwell family cards. Currently, it tends to be used as a preprocessing stage within more intricate image manipulations and has recently been implemented quite efficiently by several teams. However, either their implementations do not come near hardware's peak performance or are unable to process large mask sizes.
Such limitations are overrun by our original PCRF (Parallel Register-only Convolution Filter) implementation of 2D convolution filters that can process 32-bit floating-point images on an NVidia K40 card using mask sizes up to 127x127 and at the same time achieving pixel throughputs over 29GPs, which is, as far as we know, the highest rate known to date. Such results were obtained by using registers sparingly and by designing memory access patterns that cancel both load and store replays at warp levels, along with optimizing cache use. Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: convolution, filter, GPU

1. INTRODUCTION

According to [1], convolution is one, if not the, most widely used operation in image processing, especially in the field of object recognition. The scope of 2D convolution ranges from simple Gaussian denoising to edge detection via a lot of other feature extraction filters. Also, with the ever-increasing complexity of image processing algorithms, the convolution filter tends to be used as a pre-processing stage within more complex image manipulation sequences. Moreover, the fact that it can be easily broken down into as many independent execution threads as pixels to be processed, has resulted in efficient GPU implementations.

Implementing the convolution operator on GPU results in kernels which are computationally lightweight and whose intrinsic performances are limited by the GPU's available *memory bandwidth*. Additionally, though it could be objected that the time costs of CPU-GPU data transfers are likely to exceed those of the actual kernel executions, it remains worthwhile to optimize their execution time as much as possible, the more so as convolution is most often just one single step out of a complex GPU processing pipeline.

Until recently, a vast majority of GPU implementations followed two mainstream principles: first, maximizing the *occupancy* (thread level parallelism) as a means to hide latencies, and second, pre-fetching data from global memory into shared memory in order to minimize global memory bandwidth usage and high-latency accesses.

However, and in contradiction to the above recommendations, Volkov proved in [2] that high performance can be achieved even *at low occupancy*, by increasing the ILP (Instruction Level Parallelism) and assigning more computations to each thread. In addition, as shown in [3] and also in our prior work on the 2D median filter ([4]), a further step toward high performance consists in privileging the use of GPU registers against that of shared memory, and in unrolling loops as much as possible.

Indeed, assuming the occurrence of a *perfect* pre-fetching stage, *ie.* through coalescent loads from global memory and optimal stores into shared memory (no bank conflicts), its time cost would still be equal to that of a corresponding *perfect* fetch into registers (instead of shared memory). Moreover, as register loads always feature much lower latency than shared memory loads (1 cycle against 38), reading data from the register file during kernel actual computations still appears to be faster. This remains the case provided no single value has to be fetched from global memory more than once, even if this clearly represents some loss of versatility against shared memory use and makes it necessary to organize kernel computations accordingly.

Eventually, by applying the above principles, our implementation proves both significantly faster than the speediest GPU implementations known to date and able to process large mask sizes, an interesting feature for various processing pipelines that involve large images or video sequences, as in [5] where 21×21 convolutions are part of a video summarization process.

Through the rest of this paper, we will first give definitions and paper-wide notations in section 2, before detailing some key points about Nvidia's Fermi and Kepler architectures in section 3. Section 4 will then present and evaluate the other most relevant implementations to date (Nvidia NPP [6], ArrayFire [7], Iandola's [8]). Section 5 will detail our proposed implementation, followed by section 6 which will present our results and performance comparisons against the other cited implementations. Section 7 will introduce the online generator that helps using our kernels and finally the conclusion will synthesize our approach and define prospects for future development.

2. 2D CONVOLUTION: DEFINITION, PROPERTIES, NOTATIONS

Given a digital image I of dimensions H and L (resp. height and width, in pixels), the 2D convolution operation is performed between image I and convolution mask[*] h and is defined for each pixel of coordinates (j,i) by

$$I'(j,i) = (I * h) = \sum_{(y<H)} \sum_{(x<L)} I(j-x, i-y)h(x,y) \tag{1}$$

While processing an image, function h is often bounded by a square window[†] of edge size $k = 2r + 1$, i.e., an uneven number, to ensure there is a center that helps in defining the mask radius r. The gray-level value of each pixel of output image I' is the weighted sum of pixels included in the $k \times k$ neighborhood defined around the corresponding pixel in the input image.

The first property to be noted is that, unlike for example the median filter, the output values of the convolution filter do not belong to input space values, which often leads to coding them with *floating-point-types*, so as to ensure sufficient precision to subsequent processing stages.

Additionally, if the sum of all coefficients in the mask is not 1, image brightness gets altered so a normalization stage has to be performed. In most cases, that involves a GPU time-costly division operation for each pixel. To solve this, we use a simple well-known workaround that consists in normalizing mask values before the GPU kernel uses them.

Implementing a convolution operation raises the issue of the output values of near-edge pixels that are not computable by using the generic formula of equation 1. One frequent technique to deal with it is to allocate a data input memory area larger than the image to process, that defines a r-pixel wide *halo* all around the image, as represented in Figure 1 for a 5×5 mask ($r = 2$) and a H$\times$W pixel image.

[*] To avoid confusion with other GPU functions referred to as kernels, the term *convolution mask* will be used instead of *convolution kernel*.

[†] We shall also point out that the square shape is not a limiting factor to the process, as any shape can be inscribed into a square. In the case of a more complex shape, the remaining space is filled by null values (padding).

For each pixel, the convolution computation involves a k×k window. In our implementation, the computation of several pixels is assigned to each thread, combined into what we call a *packet* of pixels. Such a *packet* contains P adjacent pixels, arranged either horizontally or vertically. Performing the P convolution computations of the whole packet involves a $(P + 2r)(2r + 1)$ pixel window referred to as the *ROI* (Region Of Interest). Figure 1 shows an horizontal packet of eight pixels and its ROI for a 5×5 mask, while Figure 3 shows two vertical packets of respectively eight and four pixels along with their ROIs for the same mask size.

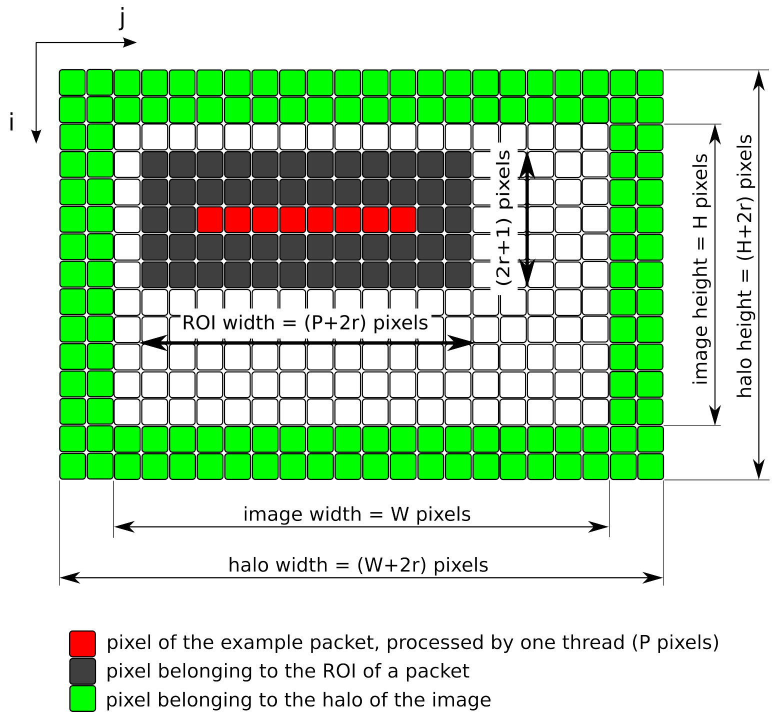

j
i
(2r+1) pixels
image height = H pixels
halo height = (H+2r) pixels
ROI width = (P+2r) pixels
image width = W pixels
halo width = (W+2r) pixels
pixel of the example packet, processed by one thread (P pixels)
pixel belonging to the ROI of a packet
pixel belonging to the halo of the image

Figure 1. Definitions and notations used in our GPU-based 2D convolution, illustrated for a $P = 8$ horizontal pixel packet and a 5×5 mask ($r = 2$).

3. GPU KEY ISSUES

Nvidia have implemented a massively parallel SIMT (Single Instruction Multiple Threads) model in all their successive GPU generations, including Fermi, Kepler and, more recently, Maxwell families.

We had access to one C2050 (Fermi family, arch. GF100) and one K40c (Kepler family, arch. GK110). The C2050 features 14 Symmetric Multiprocessors (SMs) including a total of 448 cores, while the K40c features 15 SMs and a total of 2880 cores. On every Nvidia GPU architectures, threads are grouped and run in 32-thread *warps*, the difference being that Kepler models schedule four warp executions concurrently while Fermi can only issue 2 warps at a time.

As our implementations rely mostly on GPU registers, it is worth noting that the Kepler GK110 architecture brings 4 times as many available registers per thread (255) as the Fermi GF100 (63), within a limit of 64 K per SM (32 K on Fermi). The Kepler family also introduces a 48 KB cache for data in global memory *that is known to be read-only* and was until now exclusively accessible through the texture unit (quoted from [9]).

On Kepler, when a warp reads from *read-only* global memory, all accessed data has to be contained within '*four* 32-byte segments. If not, supplementary transactions occur, called *global load replays*, and are liable to negatively impact kernel efficiency by consuming too much memory bandwidth. A global read instruction generating no load replay is often referred to by Nvidia as a *coalescent* access.

Global memory write instructions follow a very similar process and may also lead to unwanted *global store replays*.

Both global *load* and *store replays* can be examined by running kernels through the CUDA profiler and are associated with two of its internal events[‡]. Any load or store transaction that does not generate *replays* corresponds to what Nvidia call a *coalescent* memory access.

During our design and benchmark process, we also watched two other parameters that are closely related with how memory accesses fit GPU constraints: the *non-coherent cache global hit rate* and the *L2 cache hit rate*[§]. They are helpful in understanding performance gaps between kernels, especially when their source code is not available.

Additionally, in order to evaluate the relative performances of our implementations, we experimentally determined the *maximum effective pixel throughput values* for each of our configurations. For this purpose, we first measured data transfer times between GPU and CPU for several amounts of data and several types of memory. It appears that the fastest configuration on the CPU's side always consists in using a page-locked memory area to store output data from the GPU's global memory. On the GPU's side, loading input data either from pure global memory or through the texture cache leads to similar transfer rates for both Fermi and Kepler models. Unlike what we and other authors claimed in previous publications, using global memory to store input data *always* leads to the fastest processes, even on Fermi architecture, thanks to a now optimal use of the non-coherent cache that will be detailed later.

On the basis of this memory combination represented in Figure 2, we designed and measured the speed of dummy kernels called *identity kernels* that just fetch data from input memory and write them out into global memory, each thread processing one pixel or more.

The maximum throughput values are respectively 31.1 GP/s, on K40 (4 pixels processed per each thread) and 14.9 GP/s on C2050 (2 pixels per thread), both obtained on 9216$\times$9216 images, with pixels stored as 32-bit floats and ECC switched off. Those throughput values are consistent with the results of the CUDA implementation ([10], [11]) of the original *Stream* memory bandwidth presented in [12].

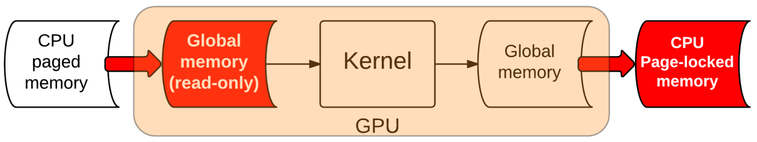


Figure 2. The optimal combination of CPU and GPU memories used in our 2D convolution implementation.

4. RELATED WORK

4.1. The Nvidia NPP library

Nvidia provide some implementations through SDK samples, and also through their NPP binary library which features a set of convolution functions adapted to various data types. Each SDK sample follows the classical approach of reading data either directly from texture or through a pre-fetch into shared memory. As for the NPP library, although it is provided as a binary, we were able, through profiling and benchmarking, to establish the following facts:

‡ *global_ld_mem_divergence_replays*, *global_st_mem_divergence_replays*
§ named *nc_cache_global_hit_rate* and *l2_l1_read_hit_rate* by the profiler

- Building a convolution operation with NPP involves many objects and preliminary instructions. Nevertheless, functions are available for each possible combination of data type and channel count, along with a clear naming scheme.
- None of the NPP convolution kernels make use of shared memory.
- Mask sizes of 3×3 and 5×5 are processed by one kernel named *forEachTupleByteQuad*, suggesting that it processes four pixels per thread. It combines efficient L2 cache usage and global memory write management (no store replays). On the other hand, read accesses generate memory load replays ranging from 7% to 12% of the overall number of pixels.
- Mask sizes of 7×7 and over are processed by another kernel named *forEachPixelByte* whose overall performance is severely impacted by 77% up to 1100% of global load replays, which means that at worst, 11 global load replays occur for each processed pixel.

4.2. The ArrayFire Library

Profiling the convolution functions provided by the *ArrayFire* library suggests that their implementations are very intricate for simple operations such as convolutions, which was confirmed by our test and profiling results:

- The largest possible mask size that we were able to process without errors was 13×13. Larger sizes always output the same results, corresponding to the inner 13×13 mask, without producing any error message. We did not further investigate the reasons of this incorrect behavior.
- Processing one 13×13 2D convolution requires the execution of no less than 17 GPU kernels.
- On a 9216×9216 pixel image (84 Million pixels), we noted over 84 Million load and store replays.
- L2 cache usage is poorly efficient with only 33% to 77% cache hits, except for kernel *spradix* (100%).
- Its image processing functions are easy to use.

4.3. Iandola's et al. implementation

Iandola *et.al.* recently proposed in [8] a more efficient implementation inspired by Volkov's early work in [2], that uses GPU registers to store input data, as recommended in one of our previous contributions ([4]).

Besides processing more than 1 pixel per thread, their implementation actually pre-fetches an entire neighborhood in registers, using one register for each pixel, with the consequence that the per-thread register count limitation is reached with a 7×7 mask size, beyond which no processing is possible (see Figure 10). Moreover, for mask sizes under 7×7, global performance is impacted by excessive block resource consumption (32k registers by thread block) which impairs thread level parallelism. It is also interesting to note that, in [8], a maximum effective throughput is mentionned and measured around 12.5 GP/s, while our own measurements show that this maximum throughput is around 15.0 GP/s. On Kepler some of their kernels are also reported to output over than 100% of it, which is attributed to texture cache use. As far as we know, such surprising results are more likely explained by experimental underestimation of the maximum effective throughput.

5. PARALLEL REGISTER-ONLY CONVOLUTION FILTER (PRCF)

It is now established that higher performances on GPU may be obtained through loop unrolling and register use, among other techniques. When designing GPU code with low computational load, one important parameter to optimize is memory bandwidth consumption, which may be reduced either by increasing per-thread data output volume or by optimizing data access patterns.

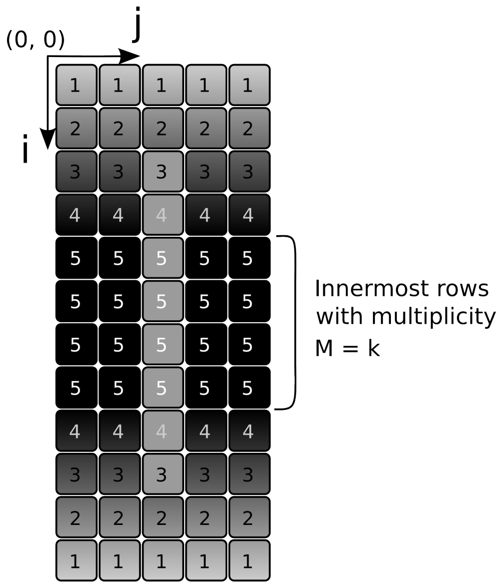
On Kepler, our proposed implementation benefits from the *read-only* functionality, but still uses global memory for input on Fermi as it turned out to be faster than texture memory.

We also observed that most of the time, the prevailing idea of pre-fetching data from global to shared memory takes too much time and does not actually lead to higher performance. That is true at least each time the ROIs of adjacent pixels overlap and the computation of the output values needs to re-use overlapping data. In such situations, our own experiments always confirmed that storing data in registers is the best performance choice.

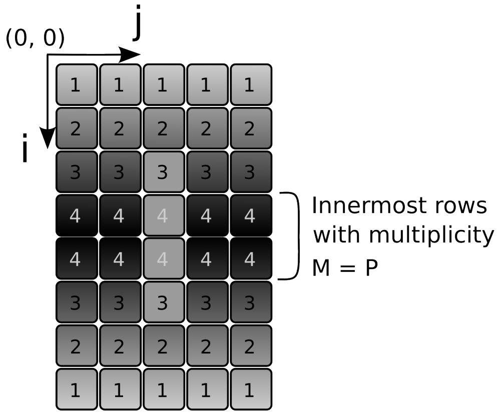
In our implementation, instead of dedicating one register to each pixel of the ROI (like in [8]), we dedicate one register to each pixel of the *packet*. This saves a lot of registers by allowing to process larger mask sizes and reducing thread block resource consumption, liable to impair global performance.

Figure 3 shows ROIs of vertical packets whose sizes are respectively eight and four pixels, both processed by a 5×5 mask. Using ROI-relative coordinates, top-left pixels are at position $(0, 0)$ while bottom-right pixels are at position $(P + 2r - 1, 2r)$. Pixels belonging to the packet are at positions $(p + r, r)$, with $p \in [0; P - 1]$. Additionally, the number in each pixel's cell represents the number of times it contributes to the computation of an output value, and is referred to as multiplicity (M). For example, pixels on row $i = 0$ only contribute to the computation of the first pixel of the packet $(p = 0)$, and value 1 is displayed in corresponding cells, while pixels on row $i = 2$ are involved in the computations of the first three pixel of the packet ($p \in [0; 2]$) and thus value 3 is displayed.

Consequently, depending on its vertical position in the ROI, each pixel may be involved in at most $M = 2r + 1$ computations for pixels belonging to the inner block, and at least $M = 1$ computation for pixels belonging to the outermost rows. The absolute maximum value of M is reached when $p \geq 2(r + 1)$ ($M \in [1; 2r + 1]$) as shown in Figure 3a, else $M \in [1; p]$ as shown in Figure 3b. Each ROI pixel can then be read only once before its contributions are computed and added to each concerned output value.



(a) $k = 5, p = 8$



(b) $k = 5, p = 4$

Figure 3. The M multiplicity of each pixel in a packet's ROI. Its value is written inside each pixel's cell.
(a) Packet size $P = 8$. The highest value $M = k = 2r + 1$ is reached if $P \geq 2(r + 1)$
(b) Packet size $P = 4$. The maximum here is $M = P$ as $P < 2(r + 1)$

Algorithm 1 represents this process for vertical packets only, beginning with the initialization of convolution parameters and output values (lines 1 to 7), while the last three lines output final values into global memory. The actual computation is done from line 8 to line 15, following the above described method where global memory loads are performed row by row from top to bottom and from left to right. On line 11, we ensure that contributions are added only to the concerned pixels of the packet, whose vertical coordinates, *a priori* ranges from $(i - r)$ to $(i + r)$, without exceeding both inferior and superior limits (resp. r and $P - r$). Array h contains mask values and is stored in constant memory, as its size is comparatively small.

Algorithm 1: PRCF algorithm run by each thread, for vertical packets

```
1  $r \longleftarrow$ mask radius ;
2  $P \longleftarrow$ packet size ;                          // Each thread processes $P$ pixels
3  $(j_{base}, i_{base}) \longleftarrow$ position, in the image, of the first pixel of the packet (base pixel) ;
4  $h \longleftarrow (2r+1) \times (2r+1)$ mask values ;
5  foreach $p \in [0; P-1]$ do                           // Output values initialization
6      $outval_p = 0.0$
7  end
8  foreach $i \in [0; P+2r-1]$ do                          // ROI-relative row position
9      foreach $j \in [0; 2r]$ do                        // ROI-relative column position
10         $pixVal \longleftarrow I(j_{base} - r + j, i_{base} - r + i)$ ;       // Read pixel from input
           image
11         foreach $p \in [max(i-r, r) - r; min(i+r, P+r) - r]$ do
12             $outval_p += h(j - (j_p - r), i - (i_p - r))pixVal$ ;
13         end
14     end
15 end
16 foreach $p \in [0; P-1]$ do         // Write output values into global memory
17     $outpixel_p(j_{base}, i_{base} + p) \longleftarrow outval_p$ ;
18 end
```

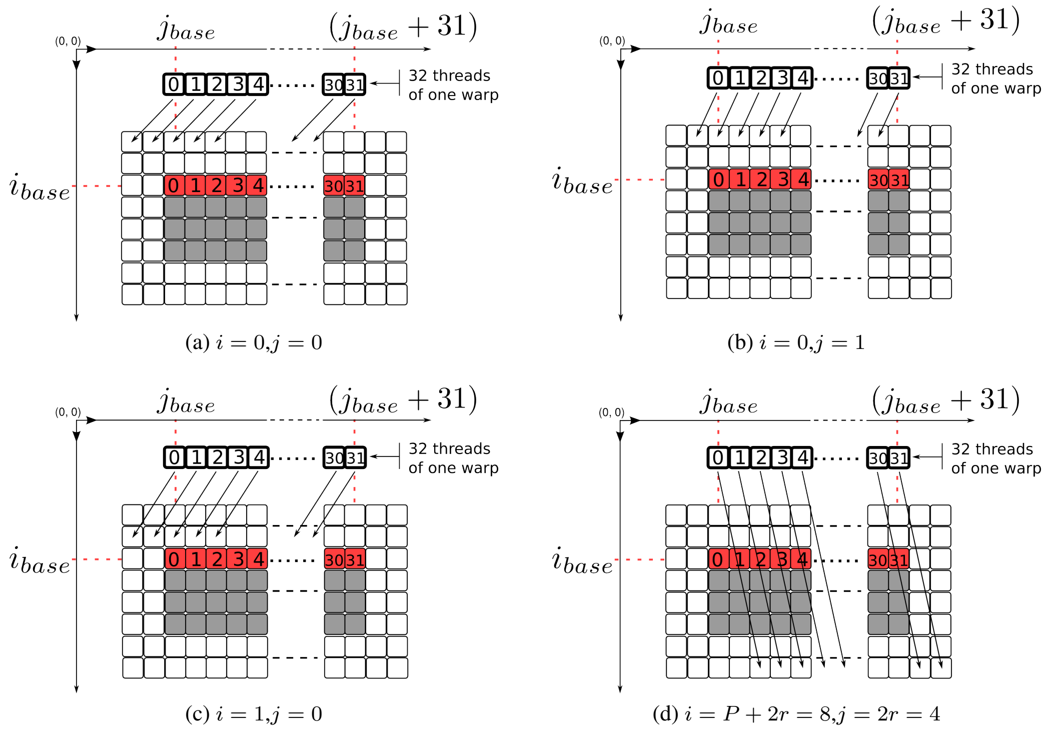
Each thread follows algorithm 1 on a different packet within the image and thus, performs $(2r+1)(P+2r)$ global memory loads where the most naive methods, *ie.* that do not exploit the window overlapping, would need $P(2r+1)^2$ loads. Within the limits of our experiments, this saves at least 33% loads with smaller mask and packet sizes ($P = 2$, $k = 3$), up to 90% with larger ones ($P = 8$, $k = 21$), as represented in detail in Figure 5. Together with cache hit maximization, this strongly reduces memory traffic and leads to high execution speeds. Eventually, to further optimize the execution speed and the GPU register file use, all loops are unrolled.

In order fo figure out how memory traffic is reduced, let us consider the execution sequence of one 32-thread warp. If we assume that one thread processes one vertical packet (P pixels), the warp it belongs to will process the convolution computations of a $32 \times P$ pixel-wide area. For this purpose, an area of $(32 + 2r) \times (P + 2r)$ pixels wide will be accessed by the warp.

Figure 4 displays the execution sequence of such a warp, following Algorithm 1 and computing the convolution values of every pixel within a 32×4-pixel area, starting at (i_{base}, j_{base}) in the input image coordinate system. This warp actually accesses a 36×8 pixel area starting at $(i_{base} - 2, j_{base} - 2)$ which corresponds to vertical packets of size $P = 4$ and a 5 convolution mask ($r = 2$). Figure 4 shows global memory loads issued by a warp for four different combinations of the loop counters i and j, *ie* rows and column counters of lines 8-9 in Algorithm 1.

As each pixel is 32-bit coded (single precision real value), each couple of coordinates i and j triggers $32 \times 4 = 128$ consecutive byte load from global memory.

At first, $i = 0$ and $j = 0$ (fig. 4a) and the warp reads the 32-pixel segment located at $(i_{base} - r, j_{base} - r)$. Provided that the input image is properly padded and aligned, that fulfills the coalescence requirement of loading 128-byte aligned 128-byte segments. Moreover, at least the next 128-byte segment is fetched into the L1 cache and thus, the following global loads are to be read from cache, while $j \leq 2r$. Further cache misses are likely to happen with each row's first pixel, but the corresponding global loads will still be coalescent. The last global load instruction is issued when $(i, j) = (P + 2r, 2r)$ as shown in Figure 4d. As a consequence, the whole global read sequence detailed above only generates $P + 2r$ cache misses and no load replays.



🟥 Base pixel of the packet processed by thread n in the warp.
▨ Pixel belonging to the packet whose base is the red pixel of its column.
□ Pixel whose value is to be read at least by one thread of the warp.

Figure 4. Coalescence of global memory loads, for mask size $k = 5$ $(r = 2)$ and packet size $P = 4$ at different stages of the 2D-loop detailed at lines 8-9 in Algorithm 1. Values of i and j displayed in the captions are related to those of Algorithm 1.

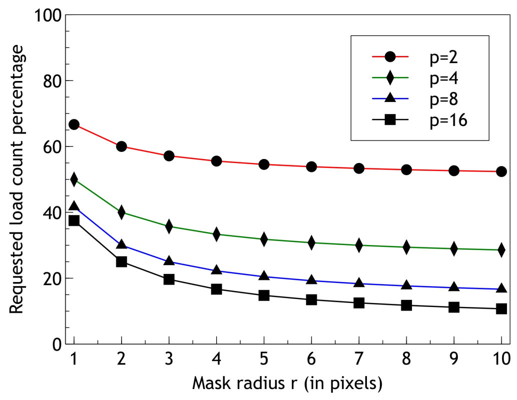


Figure 5. Reduction of global memory load count. Reference value (100%) is the total load count required by methods not exploiting window overlapping (max load count). Depending on mask and packet sizes, our method requires 10% to 66% of the max load count.

From a global point of view, one can observe that each pixel is actually read several times, but this has no impact on overall performances as each load instruction actually triggers 32 parallel global memory accesses, one for each thread of the warp. Thus, in terms of execution speed, it does not make a difference whether one thread loads one single pixel value or if one warp coalescently loads as many as 32 pixel values. This approach is obviously faster than using shared memory that would involve a pre-fetching stage of the entire ROI of each block (placed before line 8 of Algorithm 1). Assuming that no bank conflicts occur, this would take the same time as our global loads of the ROI into registers (line 10 of Algorithm 1). Nevertheless, the actual computation (loops at lines 8-9) would then need to read data from shared memory, which would necessarily imply extra execution time.

As for the global stores, the optimal access rules are similar to the loading rules, and no store replay is generated if all the threads of a warp store their data within four 32-bytes aligned segments, not necessarily consecutive. Each iteration at line 16 of Algorithm 1 triggers the store of 32 consecutive pixel values, 128-byte aligned, which fulfills the no store replay requirement.

6. RESULTS AND PERFORMANCE COMPARISON

Our experiments focus on 32-bit floating point gray-level images of size 9216$\times$9216 pixels, which is neither a hardware nor a software limit, but the consequence of our choice to compare our implementation with Iandola's *et al.*. For the same reason, we ran our kernels on an older Fermi C2050 in addition to our initial Kepler K40 target. All results have been obtained with the 6.5 version of the Nvidia software development kit and the following GPU settings:

- **K40** graphics 875 MHz, memory 3004 MHz, ECC off.
- **C2050** graphics 573 MHz, memory 1494 MHz, ECC off.

In our implementation, no significant additional memory area is required, which means that the size of the processed image can be extended up to half of the GPU's global memory size. In addition, the amount of available constant memory, in which mask values are stored allows up to 127$\times$127 mask sizes.

As for the register count required by the NVCC compiler, it is not easily predictable and always seems to be overestimated. Surprisingly, we observed that register count hardly depends on packet size. As plotted in Figure 6, the register count, which remains constant up to 61$\times$61 masks, rises to unexpected high values on both Fermi or Kepler cards. As a result, on Kepler K40, our kernels are actually able to process mask sizes up to 107$\times$107 without generating any register spilling in local memory, while the maximum size is 59$\times$59 on Fermi. Above these thresholds, every two-pixel increase of the mask size leads to spill four or five values into local memory, but this hardly impairs overall performance, those values beeing L1-cached.

Whatever the values of mask and packet sizes, execution speed always follows the same global variations against image size. These variations are reproduced in Figure 7 for sizes from 512$\times$512 to 9216$\times$9216 on both GPU models. We actually computed and displayed pixel throughput values of each execution, with reference value 1 defined as the highest throughput value obtained when processing the 9216$\times$9216 pixel image. This will be, from now on, the only size presented in our results and comparisons. The charts of Figure 7 can be used as look-up functions to estimate the actual throughputs expected with intermediate sizes.

Figure 8 shows pixel throughputs achieved, on K40, by our PRCF kernels (Parallel Register-only Convolution Filter) for various packet sizes, along with the optimal execution settings of thread block dimensions *blockDim.x* and *blockDim.y*, determined experimentally through exhaustive search. Our kernels achieve their best throughputs with $P = 8$ or $P = 16$, while the optimal thread block dimensions is most often 512$\times$1, except for mask sizes 5,7 and 9 where it appears to be respectively 128$\times$1, 128$\times$1 and 256$\times$1. Beyond $P = 16$ pixels per thread, as performance decreases, the corresponding results have not been plotted.

The comparison with Nvidia's NPP and ArrayFire's implementations is shown in Figure 9. For a given mask size, the best throughput obtained when the packet size varies ($P \in 1, 2, 4, 8, 16$ Cf. Figure 8a) is displayed in Figure 9a as the PRCF value. Speedups achieved by PRCF against ArrayFire are plotted in Figure 9b and range from $\times$2.5 to $\times$11.0; against NPP, they range from $\times$6.8 to $\times$26.0. As for NPP, its plot may seem unusual, but is easily explained by the two distinct types of kernels involved depending on mask size. On K40, our fastest kernel (3$\times$3 mask) achieves 90% of the maximum effective throughput allowed by the GPU, as defined in section 3.

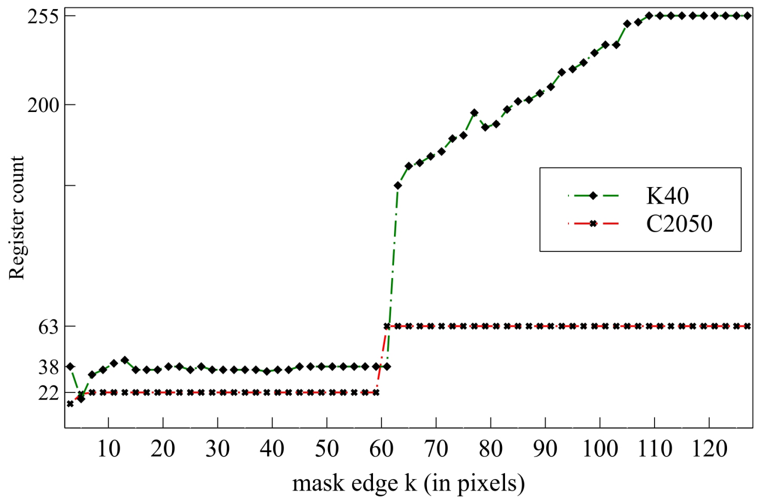


Figure 6. The register count required by our kernels. Register spilling occur when the requested register count exceeds the maximum allowed by the GPU, which is 255 on Kepler and 63 on Fermi. This situation can be clearly identified on the Figure for both GPU family, above $k = 60$ for Fermi and above $k = 107$ for Kepler.

Figure 10 shows compared performances, on C2050, of all studied implementations, including Iandola's. While throughput values achieved by NPP, ArrayFire and PRCF follow a similar and usual $(1/k)$ curve, Iandola's performance decrease is close to linear $-\alpha k$ for $k \leq 7$. Speedups achieved by our kernels against all others are plotted in Figure 10b and range from $\times 1.8$ to $\times 5.0$ against Iandola's, and from $\times 3.5$ to $\times 28.2$ against NPP and ArrayFire.

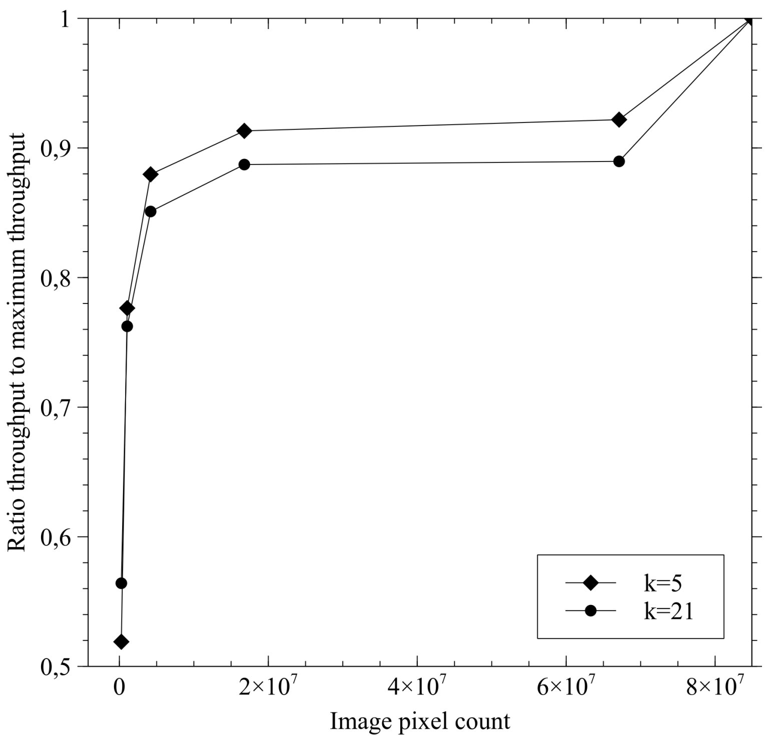
The Kepler GTX680 card (arch. GK104) on which Iandola *et al.* conducted their experiments is supposed to be faster than our K40 for such simple-precision computations and *relatively small* images, according to recent reports such as [13]. On GTX680, their kernels processed a 3$\times$3 2D convolution at the speed of 17 GP/s (32-bit floating-point 9126$\times$9216 image). In the same configuration, our PRCF achieves over 29 GP/s on K40 ($P = 8$, 256 threads per block). We shall not, however, further develop comparisons, as no GTX680 was made available to us.

7. AUTOMATIC KERNEL GENERATOR

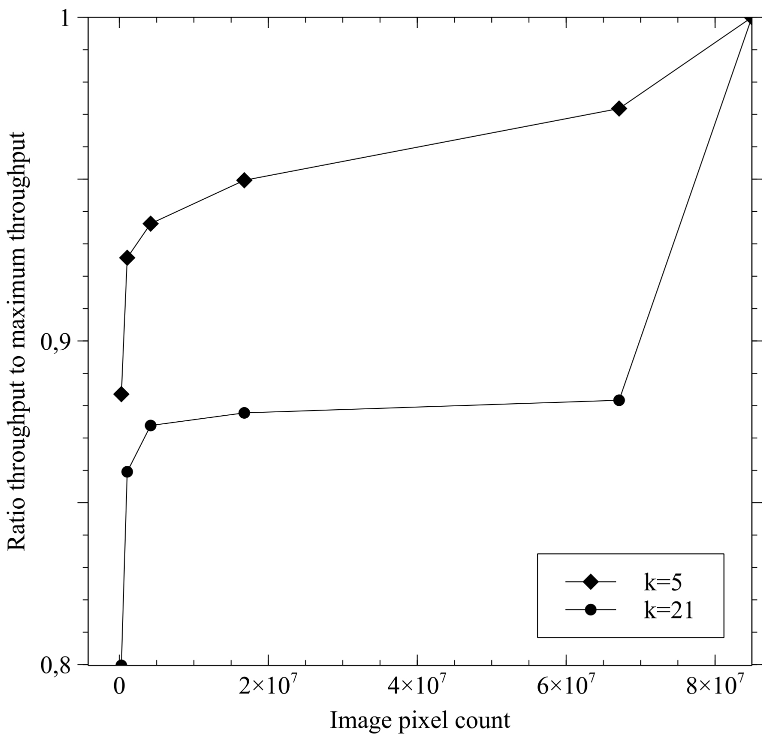
The high performance level of our PRCF has its counterpart, as one specific kernel processes only one mask size and one packet size. To override this constraint and help users or developers who are willing to use our kernels, we designed an online application ([14]) that collects parameters and generates the requested kernels on the fly, together with an appropriate *main* file. We also provide the necessary *Makefile* and a sample image in order to supply a fully functional set, ready to compile and run.

The generated *main* file does not currently implement the extensive search process of the optimal thread block dimensions. Instead, it assigns to thread blocks a constant size of 512$\times$1, as this represents the opimal size in most situations, except for mask sizes 5, 7 and 9, where the performance gap against the actual optimum is of very limited impact.

In addition to the fondamental parameters,*ie* the mask size k and the paquet size P, the generator allows to choose the input memory type (global or texture) and whether the convolution is to be considered as separable or not. The separable version of the PRCF relies on two specific 1D optimized implementations of the generic 2D non-separable convolution described in this paper, each of them processing respectively the 1D horizontal and vertical stage.

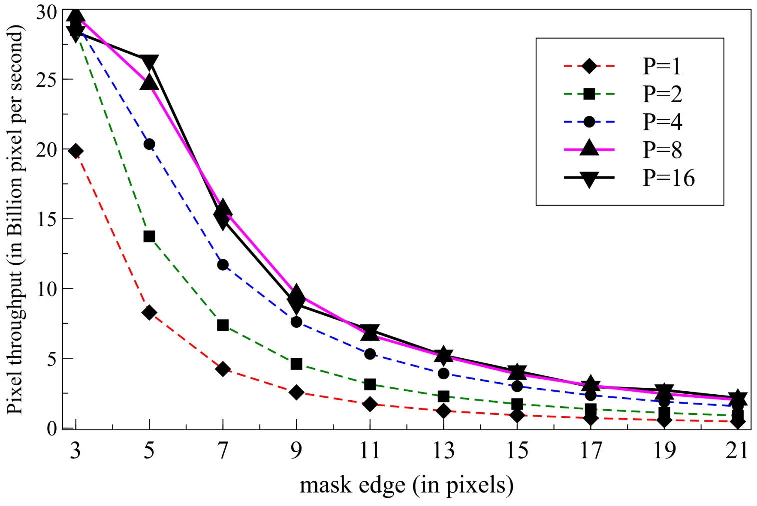


(a) Kepler K40c

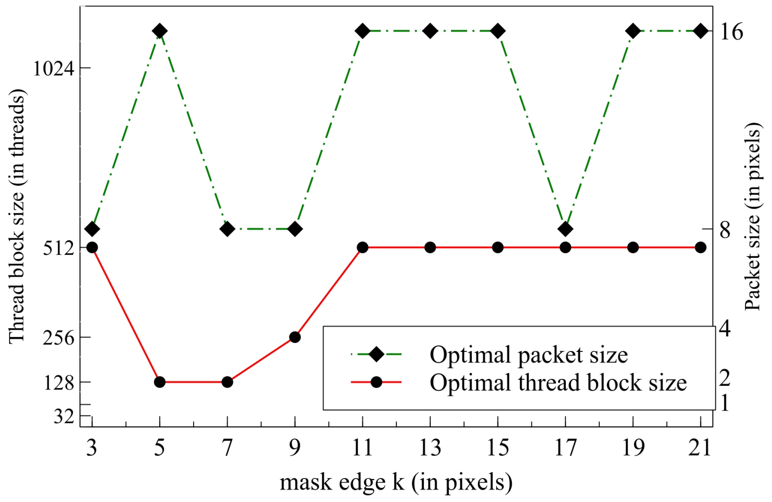


(b) Fermi C2050

Figure 7. Impact of the image size on the pixel throughput. The impact is measured as the ratio of the actual throughtput to the maximum throughput on images of size ranging from 512×512 to 9216×9216, on both (a) K40 and (a) C2050 GPU models, for mask sizes $k = 5$ and $k = 21$. The reference maximum throughput is computed for a 9216×9216 pixel image.



(a) Performance of our kernels for various packet sizes.



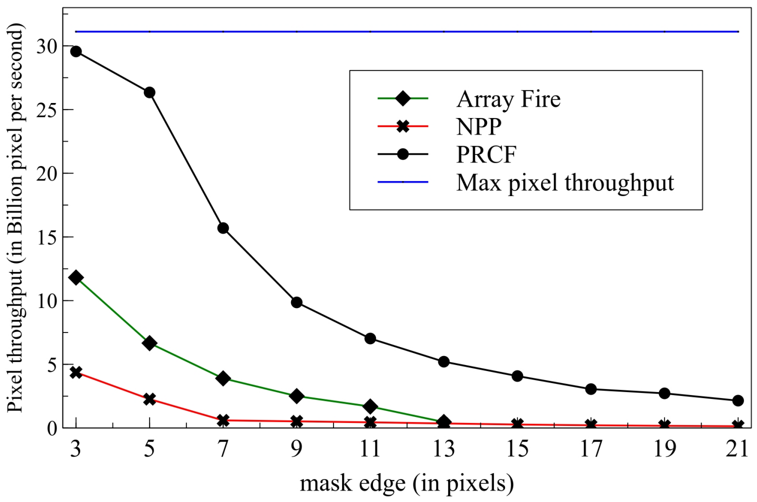
(b) Optimal settings.

Figure 8. Performances and optimal settings of our proposed PRCF kernels, when processing a 9216×9216 pixel image on K40 with mask edge value ranging from $k = 3$ to $k = 21$. (a) Pixel throughputs of our kernels when packet size $P \in 1, 2, 4, 8, 16$. (b) Optimal values of the packet size P and the thread block size (*blockDim.x*×*blockDim.y*).

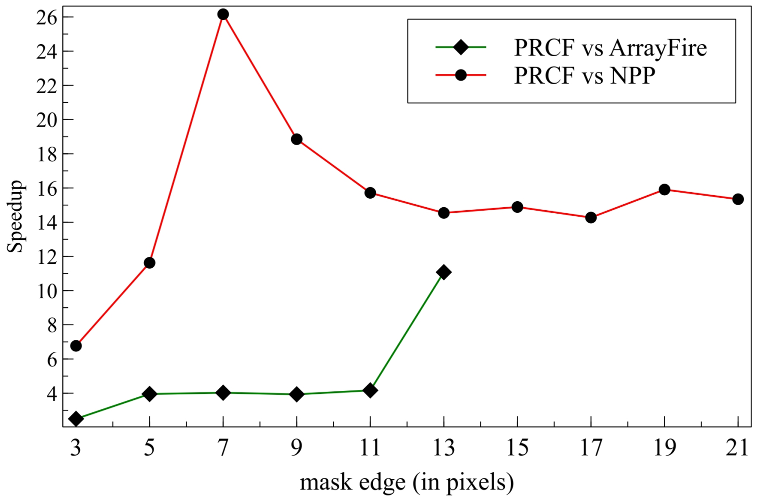
8. CONCLUSION

In the proposed 2D convolution implementation, our guideline has been both to perform all computation in registers and to keep register use well below hardware limitations by assigning one register to each pixel processed by a thread. On Kepler GPUs, our kernels feature a constant register count for mask sizes up to 61×61 and can process mask sizes up to 107×107. For smaller mask sizes, this low register consumption preserves high thread parallelism, allowing to expect high-efficiency.

Besides limited register use, higher instruction level parallelism is achieved for all mask sizes by having each thread process several adjacent pixels. We have also taken advantage of this multiplicity by using window mask overlapping between adjacent pixels to design optimal memory access patterns. Consequently, the coalescence of all load and store transactions is strictly preserved while the number of transactions itself is minimized.



(a) Performances of PRCF, NPP and ArrayFire kernels.

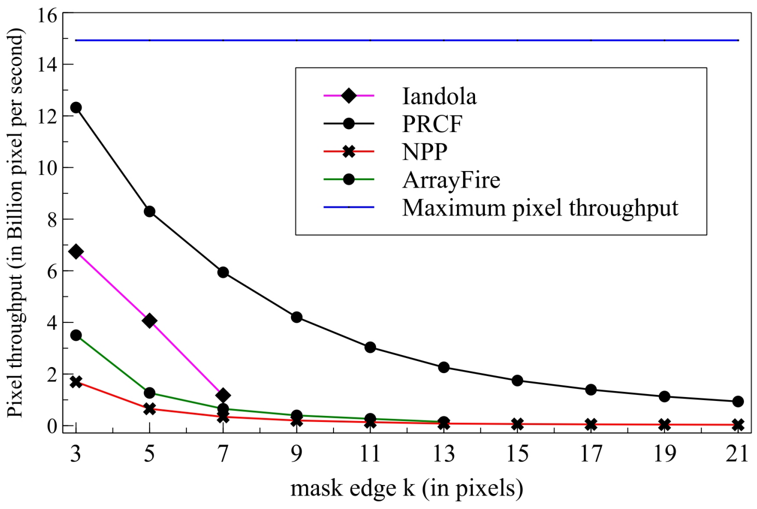


(b) Speedups achieved by our PRCF against NPP and ArrayFire.

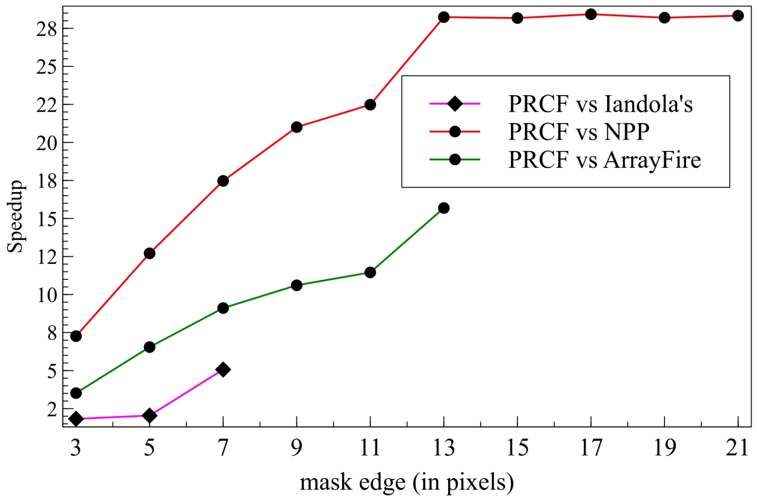
Figure 9. Comparison of our best kernels against NPP and ArrayFire, for mask size ranging from 3×3 to 21×21 on a 9216×9216 pixel image. Note that ArrayFire provides erroneous results above 13×13. (a) Throughputs values of our best PRCF kernels compared to NPP and ArrayFire. (b) Speedups achieved by our PRCF implementation against ArrayFire and NPP.

All the above optimizations have led us to propose a 2D convolution implementation that outperforms all implementations known to date. We also provide an online software that generates user customized kernels on the fly, together with all the necessary material making the requested kernel ready to run. However, we are working on a more user-friendly interface and improved capabilities, such as the ability to choose a specific GPU model and to run the optimized generated kernel on one of our own cards if it appears to be available.

In [4], we successfully applied similar techniques to generate fast median filter kernels and are now planning to extend them to speed up more complex processes. We specifically target recent *per-patch* image processing algorithms in which the ability of our kernels to process large masks with a high parallelism level is likely to be of interest.



(a) Performance of studied kernels on Fermi C2050.



(b) Speedups achieved by our PRCF against the others.

Figure 10. Performance comparison of all studied kernels, when processing a 9216×9216 image on C2050, with mask edge values ranging from $k = 3$ to $k = 21$. Iandola's is limited to 7×7 (*Cf.* section 4.3) and ArrayFire to 13×13.
(a) Pixel throughputs.
(a) Speedups.

ACKNOWLEDGMENTS

This work received the support of Nvidia and the Franche-Comté regional parliament through the supply of Kepler K40 cards. We also thank Daniel Cuney and Ingrid Couturier for their help in writing the English version.

REFERENCES

1. Su BY. Parallel application library for object recognition. PhD Thesis, University of California Berkeley 2012. URL http://www.escholarship.org/uc/item/52t70776.
2. Volkov V. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC* 2010; **10**.
3. Volkov V, Demmel JW. Benchmarking gpus to tune dense linear algebra. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, IEEE Press: Piscataway, NJ, USA, 2008; 31:1–31:11. URL http://dl.acm.org/citation.cfm?id=1413370.1413402.
4. Perrot G, Domas S, Couturier R. Fine-tuned high-speed implementation of a gpu-based median filter. *Journal of Signal Processing Systems* 2013; :1–6.
5. Gurrin C, Hopfgartner F, Hurst W, Johansen H, Lee H, OConnor N. *MultiMedia Modeling: 20th Anniversary International Conference, MMM 2014, Dublin, Ireland, January 6-10, 2014, Proceedings*, vol. 8325. Springer, 2014.
6. Nvidia. Nvidia performance primitives. URL http://developer.nvidia.com/npp.
7. ArrayFire. URL http://arrayfire.com.
8. Iandola F, Sheffield D, Anderson M, Phothilimthana P, Keutzer K. Communication-minimizing 2d convolution in gpu registers. *Image Processing (ICIP), 2013 20th IEEE International Conference on*, 2013; 2116–2120, doi:10.1109/ICIP.2013.6738436.
9. Nvidia. Nvidias next generation cuda compute architecture: Kepler tm gk110 2012. URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.
10. Cumming B. cuda-stream software. URL https://github.com/bcumming/cuda-stream.
11. Philipps E. Optimizing the high performance conjugate gradient benchmark on gpus 2014. URL http://devblogs.nvidia.com/parallelforall/optimizing-high-performance-conjugate-gradient-benchmark-gpus/.
12. McCalpin JD. Memory bandwidth and machine balance in current high performance computers 1995; URL http://www.researchgate.net/profile/John_Mccalpin2/publication/213876927_Memory_Bandwidth_and_Machine_Balance_in_Current_High_Performance_Computers/links/541083180cf2d8daaad3d254.pdf.
13. Gupta P. Performance analysis of cula on different nvidia gpu architectures. *Technical Report*, Simulation Based Engeneering Lab - University of Wisconsin may 2014.
14. Perrot G. Median and convolution kernel generator 2013. URL http://info.iut-bm.univ-fcomte.fr/staff/perrot/convomed.