Efficient scatter-based kernel superposition on GPU

Joakim da Silva[a,b,*], Richard Ansorge[a], Rajesh Jena[c]

[a] Cavendish Laboratory, University of Cambridge, UK

[b] Department of Oncology, University of Cambridge, UK

[c] Cambridge University Hospitals NHS Foundation Trust, UK

*Corresponding author:

jd491@cam.ac.uk

+44 1223 337010

BSS, Cavendish Laboratory

19 J J Thomson Avenue

Cambridge,

CB3 0HE

Abstract

Kernel superposition, where an image is convolved with a spatially varying kernel, is commonly used in optics, astronomy, medical imaging and radiotherapy. This operation is computationally expensive and generally cannot benefit from the mathematical simplifications available for true convolutions. We systematically evaluated the performance of a number of implementations of a 2D Gaussian kernel superposition on several graphics processing units of two recent architectures. The 2D Gaussian kernel was used because of its importance in real-life applications and representativeness of expensive-to-evaluate, separable kernels. The implementations were based both on the gather approach found in the literature and on the scatter approach presented here. Our results show that, over a range of kernel sizes, the scatter approach delivers speedups of 2.1–14.5 or 1.3–4.9 times, depending on the architecture. These numbers were further improved to 4.8–28.5 and 3.7–16.8 times, respectively, when only “exact” implementations were compared. Speedups similar to those presented are expected for other separable kernels and, we argue, will also remain applicable for problems of higher dimensionality.

Keywords

kernel superposition;

variable kernel convolution;

spatially varying;

point spread function;

scatter;

GPU;

Introduction

Kernel superposition vs. convolution

Image filtering finds use in a plethora of image processing, applications and can be used for example to suppress noise, enhance detail, and detect edges. Often, the filtering process consists of a convolution of the image data with a kernel, sometimes referred to as a filter or mask. In a true convolution, the kernel does not vary over the image and is a function only of the vector difference between the input and output coordinates (Eq. 1a). In the discrete case, this translates to element-wise multiplication and summation of a constant kernel matrix with the corresponding neighbourhood in the input (Eq. 1b). In some applications, however, the kernel is also dependent on its absolute spatial position in either input or output space (Eq. 2a and 2b, respectively). The resulting operation, referred to as kernel superposition (KS), variable kernel convolution, or convolution using a spatially-varying point spread function, appears in many different fields. Examples include radiotherapy dose calculation, both using photons [1] and charged particles [2], ultrasound imaging [3], computed tomography [4,5], positron emission tomography [6], photography [7,8], astronomy [9,10], and microscopy [11].

$$(f * g)(\mathbf{x}) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} f(\mathbf{x}')g(\mathbf{x} - \mathbf{x}')\mathrm{d}\mathbf{x}' \qquad \text{Eq. 1a}$$

$$O(\mathbf{x}) = \sum_{x'_n} \cdots \sum_{x'_1} I(\mathbf{x}')K(\mathbf{x} - \mathbf{x}') = \sum_{x'_n} \cdots \sum_{x'_1} \mathbf{I}_{x'_1,\cdots,x'_n} \mathbf{K}_{x_1 - x'_1,\cdots,x_n - x'_n} \qquad \text{Eq. 1b}$$

Direct evaluation of an n-dimensional convolution between an image and a kernel of side lengths M and N, respectively, requires a total number of operations proportional to $M^n N^n$. Several techniques can be employed to make the evaluation more efficient, especially when working with large kernels or images of higher dimensionality. These include carrying out the calculation in Fourier space; dividing an n-dimensional convolution into n 1D convolutions (for spatially separable kernels); and approximating more complicated kernels by repeated convolutions with simpler ones. GPU implementations employing these techniques, as well as direct evaluation, have all been described in detail [12–15].

$$K_{\text{in}}(\mathbf{x}, \mathbf{x}') = K_{\text{in}}(\mathbf{x} - \mathbf{x}', \mathbf{x}') \quad \text{Eq. 2a}$$

$$K_{\text{out}}(\mathbf{x}, \mathbf{x}') = K_{\text{out}}(\mathbf{x} - \mathbf{x}', \mathbf{x}) \quad \text{Eq. 2b}$$

Some of the techniques mentioned can be applied to special cases of KSs, such as fully spatially separable problems (Eq. 3a) [16]; regionally constant kernels or linear combinations thereof [17]; spatially varying kernels that can be made invariant by transforming the input image [18]; and approximate solutions for slowly varying Gaussian kernels [19]. However, this is not true for the general case, which thus requires direct evaluation. Further, good performance of direct evaluation techniques for true convolution on GPU rely on keeping the kernel values in memory and broadcasting one value at a time to multiple threads [15], which has no equivalent for KS.

$$K_{\text{FS}}(\mathbf{x} - \mathbf{x}', \mathbf{x}') = k_{\text{FS}}(x_1 - x'_1, x'_1) k_{\text{FS}}(x_2 - x'_2, x'_2) \cdots k_{\text{FS}}(x_n - x'_n, x'_n) \quad \text{Eq. 3a}$$

$$K_{\text{PS}}(\mathbf{x} - \mathbf{x}', \mathbf{x}') = k_{\text{PS}}(x_1 - x'_1, \mathbf{x}') k_{\text{PS}}(x_2 - x'_2, \mathbf{x}') \cdots k_{\text{PS}}(x_n - x'_n, \mathbf{x}') \quad \text{Eq. 3b}$$

Gather, scatter and system matrix approaches

The KS operation is in itself inherently parallel, since each output value can be calculated independently given the input image. In a parallel implementation, it is natural to assign threads either to elements in the input or output image. The former is referred to as the scatter approach, where each input element “scatters” results to its neighbouring elements in the output, whereas the latter is referred to as the gather approach, where each output “gathers” results from neighbouring elements in the input (Figure 1). Conventional wisdom from GPU programming, and in particular the true convolution case, states that the gather approach is preferable, since each thread can accumulate its output locally and avoid having to resolve write conflicts through costly synchronisation or atomic operations [20]. However, when the kernel is dependent on its position in the input image (Eq. 2a), which seems to be the case in most real-world applications, the scatter approach has the advantage that the kernel is fixed at the thread level. If the kernel is further spatially separable with respect to the difference between input and output coordinates according to Eq. 3b, this means that each thread only needs to calculate its 1D kernel values once and keep them locally. Therefore, the evaluation of the kernel at each of the N^n neighbours can be replaced by a multiplication of n pre-calculated 1D kernel

values. Such separability is common in image filters, perhaps most notably in the Gaussian kernel. (If the kernel is of the form of Eq. 2b and separable in the first argument, the problem becomes trivial; the gather approach is used to avoid any synchronisation issues and each thread pre-calculates its 1D kernel values as described.)

The gather and scatter approaches both perform the KS in a single step. Alternatively, the kernel evaluation and the multiplication can be divided into two separate steps, by first calculating the N^nM^n non-zero kernel values and storing them in a sparse system matrix. The KS is then performed by multiplying the system matrix with the vector containing the input image. If the same matrix can be used for multiple KSs, this effectively replaces each kernel evaluation with a load operation of the corresponding value from memory. If, however, the kernel parameters are different for each image, the benefit of using this technique is lost; it involves the same number of kernel evaluations and multiplications but, in addition, requires each of the N^nM^n kernel values to be stored and subsequently retrieved from memory. Further, for higher-dimensional problems, the amount of memory required for this approach may become a concern on GPUs.

Context of the presented work

The work presented here is intended as a systematic investigation of efficient, multi-dimensional KS implementation on GPU, similar to what has previously been done for field-programmable gate arrays [21]. Excluding early GPU work on now outdated architectures, e.g. by Wang et al. [22], four related studies have been identified in the literature. Two of these present GPU implementations of 1D KSs, as a part of image reconstruction workflows for 3D ultrasound imaging and positron emission tomography, respectively. Cui et al. started with a high-dimensional model but used symmetry to reduce the kernel to a 1D asymmetric Gaussian, which makes the evaluation technique not applicable to our case [23]. Gomersall et al. used the system matrix approach, which may be applicable also for higher-dimensional problems, but due to the lack of advantages for kernels that change with each image this technique will not be considered further [24]. The remaining two articles focus on 2D KS used to aid object discovery in astronomical images and in proton therapy dose calculation, respectively. For their GPU implementations, both these studies, by Hartung et al. [25] and Fujimoto

et al. [26], use the gather approach as recommended for true convolutions. Here we present several GPU implementations of the KS operation based on the gather approach and compare their performance to new scatter-based implementations on two common GPU architectures for a wide range of parameters. The study was carried out in the context of fast dose calculation for proton therapy [27], and hence the focus is on 2D Gaussian kernels. However, the results should be applicable also to other problems separable according to Eq. 3b and those of higher dimensionality.

Methods

General considerations

The presented implementations are written in CUDA, a proprietary extension to C/C++ used to program GPUs from Nvidia Corporation (Santa Clara, CA, USA). Although the discussion should be applicable to modern GPUs from any manufacturer, the terminology used throughout the paper will be that of CUDA. In CUDA parlance, a function that is executed on the GPU is known as a kernel function (KF), which is not to be confused with the mathematical kernels or filters used in convolution or superposition operations. A list of other terms used can be found in the Glossary section.

Since the Gaussian function has infinite support, a cut-off, beyond which kernel contributions are ignored, must be chosen. The cut-off is normally given as a multiple of the standard deviation, σ. Each value of σ, and hence each input pixel, therefore has an associated minimum necessary kernel radius, r, given as an integral number of pixels. Due to the lock-step execution, each execution time of a warp will be limited by the largest value of r among its threads, r_{warp}, which can thus be used as the cut-off for all threads within the warp without loss of performance. Here we choose to let all threads within a block (which could, in principle, be limited to one warp) use their largest value of r, r_{max}, as a common cut-off (giving $N=2r_{max}+1$ for each block). In doing so, we can implement r_{max} as a template parameter, which allows us to benefit from loop unrolling and compile-time evaluation of much of the expensive integer arithmetic. Generally, blocks of different r_{max} can be present in the same image, meaning several KFs have to be launched. For the case of many and/or large images, this can be done sequentially without loss of performance. For single, small images, blocks with r_{max} within specified

ranges can be batched together to increase the GPU usage at the expense of some redundant calculation or, on recent GPUs, KFs corresponding to different values of r_{max} could be launched concurrently.

The presented implementations assume that the input, σ-map, and output reside in global memory and are arranged to allow for coalesced memory operations. We use b_x and b_y to denote the width and height, respectively, of a thread block; t_x and t_y similarly refer to the width and height of a tile. It is assumed throughout that $t_x=b_x=32$, which equals the warp size on the considered GPU architectures, and that t_y is an integer multiple of b_y. If, for example, $t_y=2b_y$, each thread in a thread block handles two pixels of the input (scatter) or output (gather). Wherever a call to *__syncthreads()* was used, care was taken to ensure that all threads of a block were guaranteed to reach this statement.

Gather implementations

The backbone of the gather approach is given in the left column of Algorithm 1. The first thing to note with this approach is that for each of the N^2 elements in its neighbourhood, a thread must read a new pixel intensity and σ, and then calculate its contribution (Figure 1). A simple KF would read these values directly from consecutive (but not necessarily aligned with a cache-line boundary) global memory. A standard technique for improving performance in similar cases, where adjacent threads repeatedly access nearby memory locations, is using shared memory as an explicitly managed cache. All values required by the thread block are then read from global memory only once, with subsequent calculations relying on low-latency, high-bandwidth shared memory. The amount of shared memory is limited, however, and using large amounts might limit the occupancy, and thereby the opportunity for latency hiding elsewhere in the KF. Further, global memory is cached (in L1 for Fermi and in L2 for Kepler) which, together with the potential of higher occupancy, might limit the benefit of using shared memory. Therefore, all the gather approach KFs were implemented in three variants, using shared memory arrays for neither, one, or both of the input intensity and σ. The size, in number of elements, of each array is given by $(t_x+2r_{max})*(t_y+2r_{max})$, equivalent to a tile with an added halo of width r_{max} (Figure 1). After populating the shared memory, the threads within a block must be

synchronised through a call to *__syncthreads()* in order to ensure that the contribution from each thread is seen by all others before continuing the calculation.

The second thing to note is that using the gather approach, each thread has to perform $2N^2$ evaluations of the separable kernel (or nN^n for n dimensions). In the case of a Gaussian kernel, each evaluation is given by

$$K_{\text{Gauss}}(d,\sigma) = \frac{1}{2}\left[\text{erf}\left(\frac{d+0.5}{\sqrt{2}\sigma}\right) - \text{erf}\left(\frac{d-0.5}{\sqrt{2}\sigma}\right)\right] \qquad \text{Eq. 4}$$

where d is the distance in pixels along the current axis between the input and output pixels. Evaluating the error function four times for each neighbour becomes computationally expensive and, as pointed out by Fujimoto et al. [26], can be avoided by interpolating between pre-calculated kernel values. Two approaches for this were investigated. In the 1D approach, erf(x/(sqrt(2)*σ)) was pre-calculated for a range of values of x and stored as a 1D array. In the 2D approach, Eq. 4 was evaluated for integer values of d and for a range of σ, and stored as a 2D array in memory. To evaluate Eq. 4, the 1D approach requires two linear interpolations whereas the 2D approach requires one (since d is an integer, the interpolation is 1D in both cases). The accuracy and performance of either approach will be dependent on the sampling of the pre-calculated values. In both cases the functions were sampled as sparsely as possible whilst keeping the absolute mean of the relative error of Eq. 4 below 1%, when taken over all pixels within the cut-off for all values of σ in the considered range. The pre-calculated values can be kept in global memory, be duplicated in shared memory for each block, or be bound to texture memory. In the former two cases, linear interpolation must be carried out explicitly in the KF, whereas in the latter case we can make use of the texture pipeline's built-in hardware interpolation. Because shared memory was too small to fit all values for the 2D approach, we are left with six possible versions of the gather implementation: the naïve version explicitly evaluating the error function; versions of the 1D approach using global, shared, and texture memory; and versions of the 2D approach using global and texture memory. Multiplied by the three variants of each, this gives a total of eighteen implementations based on the gather approach.

Scatter implementations

The backbone of a scatter-based approach is shown in the right column of Algorithm 1. Since the intensity and σ remain constant as a thread loops over x and y, these have to be read only once from global memory, after which they are kept in registers (Figure 1). Based on the value of σ, each thread calculates its $r_{max}+1$ (or $2r_{max}+1$ for non-even kernels) values of the kernel which are stored in an array. Since the array length is known at compile-time and the indexing is constant, the compiler decides whether they are stored in registers or local memory.

The difficulty with implementing this approach lies in finding an efficient way of avoiding race conditions for the write operation in the innermost loop of Algorithm 1 (line 11). As a naïve approach we can make use of atomics to have each thread atomically add its result directly to the output in global memory. However, the degree of serialisation of the atomic operation due to overlap between threads of different warps is expected to be high. Better performance may therefore be achieved if the accumulation for each block is done locally in shared memory. As in the gather approach, the size of shared memory required by a single block is $(t_x+2r_{max})*(t_y+2r_{max})$, elements (Figure 1), although here one array is sufficient. Once a block finishes its calculations, the total result is transferred from shared to global memory, again using atomic addition to avoid race conditions in the overlapping halos of different blocks.

Five versions of a scatter-based KF were developed. In the global atomics version, each thread adds its partial result directly to global memory as discussed above. In the shared atomics version, each thread atomically adds its partial result to the array of shared memory in the innermost loop of Algorithm 1 (line 11). This is similar to the global atomics version, but is expected to achieve better performance for two reasons: lower latency of shared memory and less serialisation due to fewer write conflicts. In the explicit sync version, the threads within a block are synchronised by calling *__syncthreads()* after the write operation (between lines 11 and 12) in the innermost loop of Algorithm 1. This ensures synchronised execution of the block which, since for each combination of the loop variables x and y each thread within a block will write to its own unique memory location, avoids race conditions. The call to *__syncthreads()* further ensures that all write operations made to

shared memory are visible to all threads of the block before continuing execution. In the threadfence version, the call to *__syncthreads()* above is replaced with a call to *__threadfence_block()*. *__threadfence_block()* ensures that all memory operations carried out by a thread prior to the call are visible to all other threads in the same block before the thread is allowed to continue execution. Since we have assumed that $b_x = t_x$ is equal to the warp size, for a given value of the loop variable y, each warp writes to its own unique row of the shared memory array (Figure 1). Hence, for a given combination of x and y, we can be sure to avoid race conditions as long as y remains the same for all warps of a block. This can be ensured by inserting a call to *__syncthreads()* after the end statement of the innermost loop (between lines 12 and 13) of Algorithm 1. In the volatile version, we remove the call to *__threadfence_block()* introduced in the previous version, and instead declare the shared memory used to hold the result volatile. This ensures that accesses to the memory in question are compiled to explicit instructions rather than optimised to registers (in which case they are not visible to other threads). In the innermost loop of Algorithm 1, the threads of a warp will therefore explicitly read, add their contribution, and write back to adjacent positions in the same row of shared memory, before each shifting one step and repeating the procedure. Since only one warp writes to a specific row of the shared memory for a given value of y, keeping the call to *__syncthreads()* from the previous version is enough to avoid race conditions.

Benchmarking

Benchmarking was carried out on five Nvidia GPUs of the Fermi and Kepler architectures listed in Table 1. Detailed analysis was carried out for the highest-performance card of both architectures, the Geforce GTX 580 and GTX 680. All systems hosting the GPUs were running 64-bit Microsoft Windows environments and the KFs were compiled for each GPU type using CUDA 6.0.

All 23 implementations described above were templated with r_{max} as the argument. The KFs were compiled and benchmarked for r_{max} ranging from 1 to 32, corresponding to kernels of odd side length N between 3 and 65. (For variants where the shared memory requirement dictated a largest possible value of r_{max} smaller than 32, this number replaced 32 as the upper limit.) All calculations were carried out using single precision floating point operations and all implementations were compiled

with the flag *-use_fast_math*. A *#pragma unroll* statement was inserted just before the innermost loop of all kernels (between lines 2 and 3 and lines 8 and 9, respectively, in Algorithm 1) since this was seen to increase the performance of some KFs without negatively impacting that of others. The CUDA cache configuration was set to prefer shared memory (48KiB shared memory, 16KiB L1 cache) for all kernels with the exception of the six gather variants that do not hold input intensity or σ in shared memory and the naïve scatter implementation, for which the configuration was instead set to prefer L1 cache (16KiB shared memory, 48KiB L1 cache).

A 512x512 pixel image of evenly distributed pseudorandom floating point values on the interval [0, 1) was used as input intensities for the test case. The KFs corresponding to each $r_{\max}$ were tested individually using σ-maps where the values of σ were chosen pseudorandomly from the interval [0, $r_{\max}/n_\sigma$), where n_σ is the kernel cut-off expressed as a multiple of σ. n_σ =3 was chosen for the benchmarking. (The value of n_σ for a fixed $r_{\max}$ should not alter the performance of the KFs except indirectly by slightly changing the sampling in the 1D and 2D gather approaches.) The performance was measured by executing each combination of kernel implementation and $r_{\max}$ for each of the five feasible combinations of $b_y \in \{8, 16\}$ and $t_y \in \{8, 16, 32\}$. The KF timings were taken as the average execution time for ten identical KF executions, resulting in a grand total of 35,250 executions per GPU. Since the aim of this study is to compare GPU implementations, the reported execution times do not include memory transfers. Where not stated otherwise, the results are those obtained for the best-performing combination of b_y and t_y for each KF and each value of $r_{\max}$. Similarly, for the gather implementations, for each value of $r_{\max}$ the reported result corresponds to the shared memory variant that showed the best performance. In the final analysis, the different implementations were grouped together according to similarity, and within each group, for each value of $r_{\max}$, the best result was selected. The following five groups were considered: “exact” gather (identical to the naïve gather version); 1D interpolation gather; 2D interpolation gather; scatter not relying on warp-synchronous execution (i.e. the naïve, shared atomics and explicit sync versions); and scatter relying on warp-synchronous execution.

A single-threaded CPU implementation written in C++ was used to verify the output from the KFs. Since it does not suffer from race conditions, the scatter approach, as outlined in Algorithm 1 (right), was used. (Comparison confirmed that the scatter implementation performed considerably better on the CPU.) Although the CPU implementation was written with performance in mind, templating for r_{max} to allow for compile-time optimisations and ensuring regular memory access to reduce cache misses, it could likely be further improved by employing more advanced optimisation techniques [28]. Execution times are therefore provided only to give a rough idea of the performance of a single CPU core. The CPU implementation was compiled using the Microsoft Visual C++ 2013 compiler and executed on an Intel i7-3770K 3.5 GHz CPU.

Results

Figure 2 shows the execution times for the gather implementations on the Geforce GTX 580 and GTX 680. On the GTX 580, both 2D implementations showed better performance than the naïve implementation with the one relying on global memory being the fastest. Over the range of r_{max}, the best-performing 2D approach was 1.5–2.2 times faster than the naïve implementation. Surprisingly, all 1D implementations showed similar performance to the naïve implementation. On the GTX 680, using texture memory resulted in best performance both for the 1D and the 2D approaches, with all others showing similar or worse performance than the naïve approach. The fastest 1D and 2D implementations were, respectively, 1.2–2.2 and 1.9–3.9 times faster than the naïve implementation. The results of the other GPUs closely reproduced those of the corresponding architecture in Figure 2, apart from a vertical shift according to their base performance.

The execution times for the scatter implementations are shown in Figure 3. On the GTX 580, the global atomics version resulted in the slowest execution for all r_{max}, whereas the volatile version exhibited best performance, executing 3.7–33.0 times faster than the former. The threadfence and the explicit sync versions performed very similarly and were second fastest for r_{max} of 9 and smaller. For larger values of r_{max}, the shared atomics version was the second fastest. On the GTX 680, the general trend was the same: the global atomics version was slowest (or very close to slowest) for all values of r_{max} and the volatile version was fastest with a speedup of between 2.1 and 7.7 times. The

performance of the explicit sync and threadfence versions was again very similar, with the latter being the second fastest version for values of r_{max} of 17 and smaller. Above this number the shared atomics version performed second best. Again, the results seen in Figure 3 were mimicked by the other GPUs of the corresponding architectures, as exemplified for the volatile version in Figure 4.

The naïve gather implementation and all scatter implementations accurately reproduced the results obtained using the CPU implementation. The different 1D and 2D gather approaches showed larger errors which were dependent on the combination of intensity and σ, with the largest relative errors seen for points receiving contributions from fewer neighbours. Since the combined error is input-dependent, and therefore hard to characterise quantitatively, the maximum relative error evaluating Eq. 4 over the ranges of d and σ was measured. Using previously mentioned sampling, the maxima were 5.7% and 15.3% respectively for the 1D and 2D approaches.

Figure 5 shows the preferred configurations for b_y and t_y for all implementations and the preferred number of shared memory arrays for the gather implementations. The gather implementations tended to perform best when b_y=t_y, i.e. when each thread processes only one output, but no global preference for either b_y=t_y=8 or b_y=t_y=16 was seen. In terms of shared memory usage, the trend was to prefer using fewer arrays for smaller values of r_{max} and none for larger values, with implementations using texture memory generally using fewer arrays. On the GTX 580, the different scatter implementations preferred different configurations of b_y and t_y: global atomics b_y=8, t_y=32; shared atomics b_y=16, t_y=32; explicit sync and threadfence b_y=8, t_y=8; and volatile b_y=8, t_y=16. Curiously, on the GTX 680 the only trend for the scatter implementations was a preference for b_y=16, t_y=16, the only configuration not preferred by the GTX 580.

Figure 6 shows the best performance for different groups of implementations on the Geforce GTX 580 and GTX 680, with the "bumps" seen in Figure 3 flattened out as discussed in the next section. For both GPUs, the scatter implementations relying on warp-synchronous execution were fastest for all values of r_{max}. Further, both groups of scatter implementations were faster than any of the gather approach counterparts, except for the two largest values of r_{max} on the Geforce GTX 680. The range of

speedups over the values of r_{max} when comparing the different groups are summarised in Table 2. Table 3 lists the ranges of relative execution times for the different groups compared to the fastest scatter implementation for all benchmarked GPUs. For reference, the single-threaded CPU implementation running on the i7-3770K processor is also included in this table.

Discussion

The scatter-based approaches of the Gaussian KS implementation achieved considerably better performance than the gather-based ones. The fastest scatter implementation was 2.1–14.5 and 1.3–4.9 times faster than the fastest gather implementation, respectively, on the Geforce GTX 580 and GTX 680 (Table 2). For both GPUs, the lower end of these ranges corresponded to smaller kernel sizes, whereas for values of r_{max} larger than around five, the speedups were all in the upper halves of the ranges (Figure 6). This was true for all GPUs listed in Table 3 (but not when comparing with the CPU).

According to the CUDA Visual Profiler tool, all implementations in the best-performing configuration were bound by either memory bandwidth or latency, except for the naïve gather approach which was compute bound. Many of the presented results are thus explained simply by the number and type of memory operations used, e.g. the 2D gather versions requiring one linear interpolation were faster than the corresponding 1D versions requiring two; the implementations relying on L1 cache or shared memory were faster than those relying on L2 cache; and the scatter versions with fewer synchronisation events and less atomic serialisation were faster than those with more. It is not surprising that the volatile scatter version, which requires N^2 shared memory operations per pixel, was faster than other implementations requiring more memory operations, using slower memory, and/or requiring synchronisation. Such a scatter KF is therefore expected to perform better for any kernel on the form given by Eq. 3b for which the evaluation time is non-negligible.

The differences between the two architectures can largely be explained in a similar way. The faster texture memory on the GTX 680 resulted in better relative performance of the 1D and 2D gather versions using texture memory and the faster global atomic operations gave better performance of the

global atomics scatter version. Conversely, the L1 cached global memory operations on the GTX 580 resulted in better relative performance for the 1D and 2D gather versions using global memory. The worse absolute performance of the GTX 680 compared to the GTX 580 for some implementations (e.g. as seen in Figure 4) can further be explained by the number of cores in an SM sharing the same local resources. On Fermi, 32 or 48 cores have access to the same amount of shared memory as 192 cores on Kepler; when shared memory limits occupancy, there will be fewer threads per core on the GTX 680, and therefore less latency hiding.

Local maxima, where the execution time was longer for one $r_{\max}$ than for some larger $r_{\max}$, are seen in a few places in Figure 3. These bumps are not an effect of the implementations; a larger $r_{\max}$ always results in increased per-thread computations and use of GPU resources. Instead, it was caused by a jump in the number of registers allocated to each thread between one value of $r_{\max}$ and the next. Since there is no way of predicting the optimal trade-off between register usage and occupancy, the compiler bases the register assignment on heuristics, which in some cases leads to a suboptimal solution. There is no direct way to increase the register usage in CUDA. In our case, the easiest way to recover some of the lost performance is to use a slightly larger $r_{\max}$ where it performs better, which has been done in Figure 6. However, since it does not affect our argument or conclusions, this curiosity was not further investigated.

For problems of higher dimensionality and small N, the difference between the gather and scatter approaches may be even greater; the scatter implementations are limited by the number of shared memory operations, which is N^n per thread, whereas the gather implementations are limited by the number of kernel evaluations, which is proportional to nN^n per thread. However, for higher-dimensionality implementations, the amount of shared memory available will limit the possible values of $r_{\max}$ for all implementations relying on shared memory. (On current GPUs, the limit on $r_{\max}$ would be about six in the 3D case.) For both the gather and scatter approaches, problems requiring a larger $r_{\max}$ can be divided into N^{n-2} separate 2D problems. By exposing this new parallelism to the GPU, these problems can be solved simultaneously using any of the implementations presented here. Therefore, the same theoretical performance difference as seen for the 2D case is expected.

Using pre-calculated kernel values gave a clear performance boost for the gather approach but resulted in large errors for unfortunate combinations of r_{max} and σ. Although performance can be traded for accuracy by increasing the sampling density (which will increase the number of cache misses), using pre-calculated kernel values might not be suitable for applications where per-pixel accuracy is critical. In these cases the performance increase using a scatter approach was even more pronounced as seen in Table 2 (with the larger values of r_{max} again corresponding to the upper half of the range).

In the test case we have considered only kernels that are circularly symmetric. To accommodate kernels with different σ along the *x*- and *y*-axes, the scatter implementation would have to evaluate the kernel separately in the *x*- and *y*-directions. However, the required $3r_{max}+2$ evaluations (or $4r_{max}+2$ for non-even kernels) still compare favourably to the $(2r_{max}+1)^2$ evaluations for a gather approach.

Finally, we identified two indirect benefits of using the scatter approach in the intended radiotherapy application. First, r_{max} is given by the largest *r* among the input pixels handled by each block. In the scatter case, these are defined by the tile, whereas in the gather case they depend also on the value of r_{max} itself, requiring an iterative search to find each r_{max} before starting the KS. Second, it is easy to check (by calling *__syncthreads_or()* at the top of the KF) if all input intensities of a block are zero and, if so, stop the execution of the block to free up space for other blocks on the SM. To avoid similar redundant calculation with the gather approach, a conditional statement would have to be evaluated at every iteration of the innermost loop.

Conclusion

We have shown that using a scatter-based approach rather than a conventional, gather-based one results in significantly better performance for 2D Gaussian KS on modern GPUs. The improvement in performance was partly achieved through the use of volatile shared memory and warp-synchronous programming, both sometimes labelled as poor programming practices. Yet, the associated performance increase in certain cases, as illustrated here, makes these techniques worth considering. Based on our results and the widespread use of Gaussian and other separable kernels, we anticipate

that multi-dimensional KS employed in a range of fields could benefit considerably in terms of execution time from a scatter-based GPU implementation.

Acknowledgements

We would like to thank Victor da Silva for help with finalising the figures. This research was funded by the European Commission Seventh Framework People Programme through the ENTERVISION project, grant agreement number 264552. Dr Jena is funded in part by Cancer Research UK.

Glossary

coalesced memory access	memory access where consecutive threads access consecutive global GPU memory
global [GPU] memory	the main GPU random-access memory, typically between 1 and 12 GiB
kernel function (KF)	a function that executes on the GPU
kernel superposition (KS)	a convolution with spatially-varying kernel or point spread function
L1 cache	the highest level cache, non-coherent and local to each SM. Occupies the same space as the shared memory.
L2 cache	second level cache, fully coherent and global
occupancy	the number of threads simultaneously residing on an SM as a percentage of the maximum. Occupancy is limited by the number of registers per thread and the amount of shared memory per thread block for each KF. High occupancy can help hide the effects of low latency.
shared memory	low-latency memory that can be accessed by all threads within the same thread block. Occupies the same space as the L1 cache and is divided between blocks executing on the same SM.
streaming multiprocessor (SM)	group of cores on the GPU sharing the same resources, i.e. shared memory/L1 cache and registers. Consists of 32 or 48 cores on the Fermi architecture and 192 cores on the Kepler architecture.
texture memory	physically the same as the global memory but is read through its own cache. Optimised for spatial locality and supporting linear interpolation in hardware.
thread block	group of threads that share the same shared memory. Consists of one or more warps.
warp	group of threads executing in lock-step acting as a single instruction, multiple data machine. The warp consists of 32 threads on both the Fermi and Kepler architectures.

Algorithm 1. Pseudocode outlining the gather (left) and scatter (right) implementations of the KS operation from a thread perspective. *X* and *Y* indicate the global thread indices in the *x*- and *y*-directions, respectively, for the current thread. Kernel(d, σ) is the evaluation of the kernel, in our example a Gaussian as given in Equation 4.

Input: image[M, M]
Input: sigma [M, M]
Output: result[M+2*r_{max}, M+2*r_{max}]

```
GatherApproach(image, sigma, result)
1: res←0
2: for y=−r_max to r_max
3:     for x=−r_max to r_max
4:         im←image[X+x, Y+y]
5:         σ←sigma[X+x, Y+y]
6:         k_x←Kernel(x, σ)
7:         k_y←Kernel(y, σ)
8:         res←res+im*k_x*k_y
9:     end
10: end
11: result[X, Y] ←res
```

```
ScatterApproach(image, sigma, result)
1: im←image[X, Y]
2: σ←sigma[X, Y]
3: k[r_max+1]
4: for d=0 to r_max
5:     k[d] ←Kernel(d, σ)
6: end
7: for y=−r_max to r_max
8:     k_y←k[abs(y)]
9:     for x=−r_max to r_max
10:        k_x←k[abs(x)]
11:        result[X+x, Y+y] ← result[X+x, Y+y]+im*k_x*k_y
12:    end
13: end
```

Table 1. Overview of GPUs used for benchmarking.

Name	Arch.	# cores	Core clock [MHz]	# cores/SM	Max shared mem./SM [KiB]	Global mem. cache
Quadro 1000M	Fermi	96	1400	48	48	L1
Geforce GTX 580	Fermi	512	1544	32	48	L1
Quadro K1100M	Kepler	384	705	192	48	L2
Geforce GTX 680	Kepler	1536	1006	192	48	L2
Geforce GTX 760	Kepler	1152	980	192	48	L2

Table 2. Relative speedups between different groups of implementations. The upper right triangle corresponds to the Geforce GTX 580 and shows the speedup of columns relative to rows. The lower left triangle corresponds to the Geforce GTX 680 and shows the speedups of rows relative to columns.

GTX 580 / GTX 680	Naïve gather	1D gather	2D gather	Safe scatter	Warp-sync scatter
Naïve gather		1.0–1.1x	1.5–2.2x	4.8–11.9x	4.8–28.5x
1D gather	1.2–2.2x		1.5–2.0x	4.3–11.0x	4.4–26.2x
2D gather	1.9–3.9x	1.5–1.9x		2.1–6.7x	2.1–14.5x
Safe scatter	3.2–10.3x	1.5–7.4x	0.9–4.7x		1.0–2.8x
Warp-sync scatter	3.7–16.8x	2.4–8.5x	1.3–4.9x	1.0–3.4x	

Table 3. Relative calculation times for the different groups of implementations as compared with the fastest scatter implementation on each GPU. The relative calculation time for the single-threaded implementation on the i7-3770K CPU is also shown for each GPU.

	Naïve gather	1D gather	2D gather	Safe scatter	Warp-sync scatter	CPU (single-threaded)
Quadro 1000M	4.3–25.4	4.1–21.5	2.3–13.4	1.1–3.2	1.0	13.6–39.4
Geforce GTX 580	4.8–28.5	4.4–26.2	2.1–14.5	1.0–2.8	1.0	132–285
Quadro K1100M	4.0–17.2	2.5–8.7	1.2–5.0	1.0–3.3	1.0	12.8–57.7
Geforce GTX 680	3.7–16.8	3.2–10.3	1.9–3.9	1.2–2.2	1.0	76.4–294
Geforce GTX 760	3.6–17.2	2.3–8.5	1.2–4.9	1.0–3.4	1.0	61.2–216

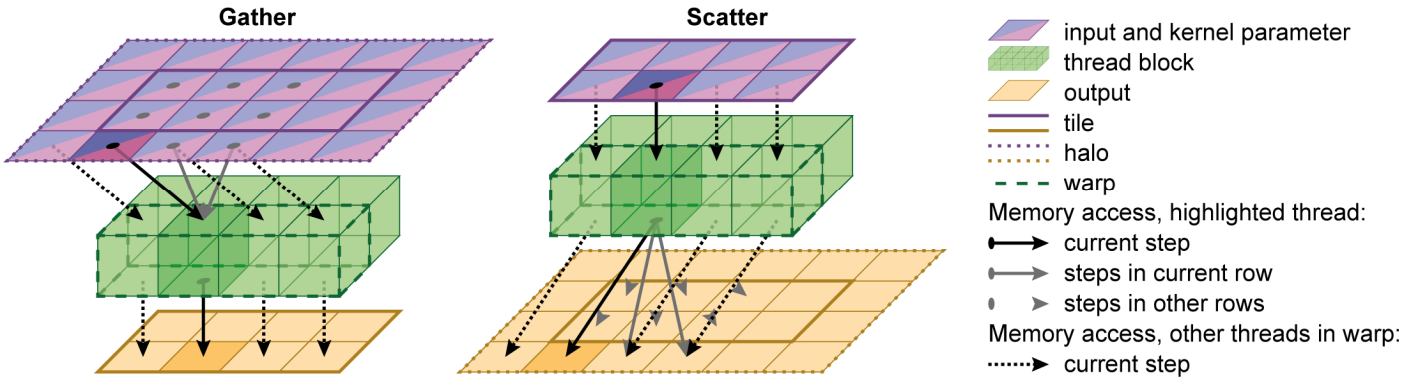

Figure 1. Overview of the gather (left) and scatter (right) approaches for a KF with r_{max}=1 and block size 4×2 (t_x=b_x=4, t_y=b_y=2). For illustrative purposes each warp is assumed to consist of only four threads.

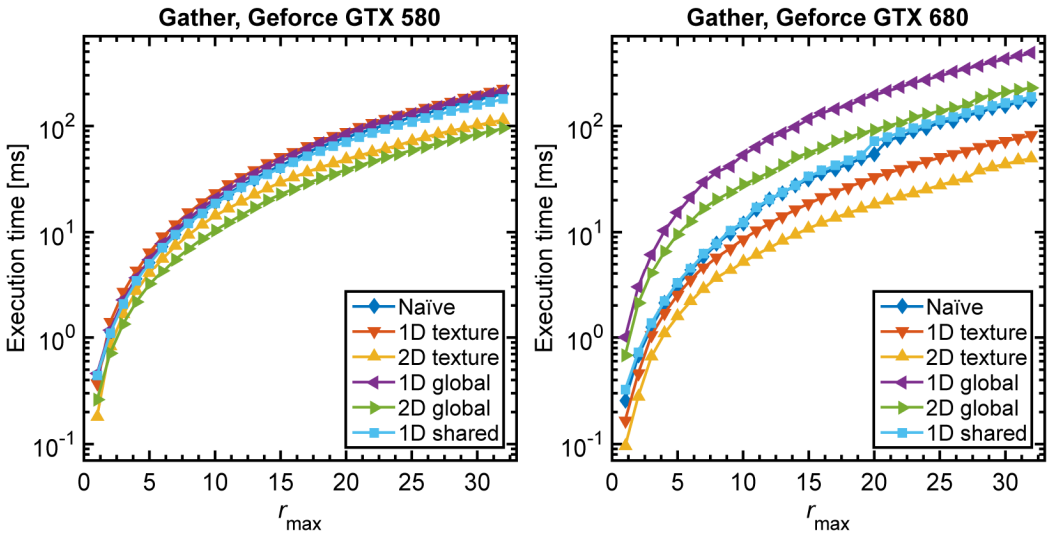

Figure 2. Execution times for the gather implementations for different r_{max} on the Geforce GTX 580 (left) and GTX 680 (right). Only the data points for the best-performing combination of t_y, b_y and number of shared memory arrays are shown for each implementation. Note the logarithmic scale on the *y*-axes.

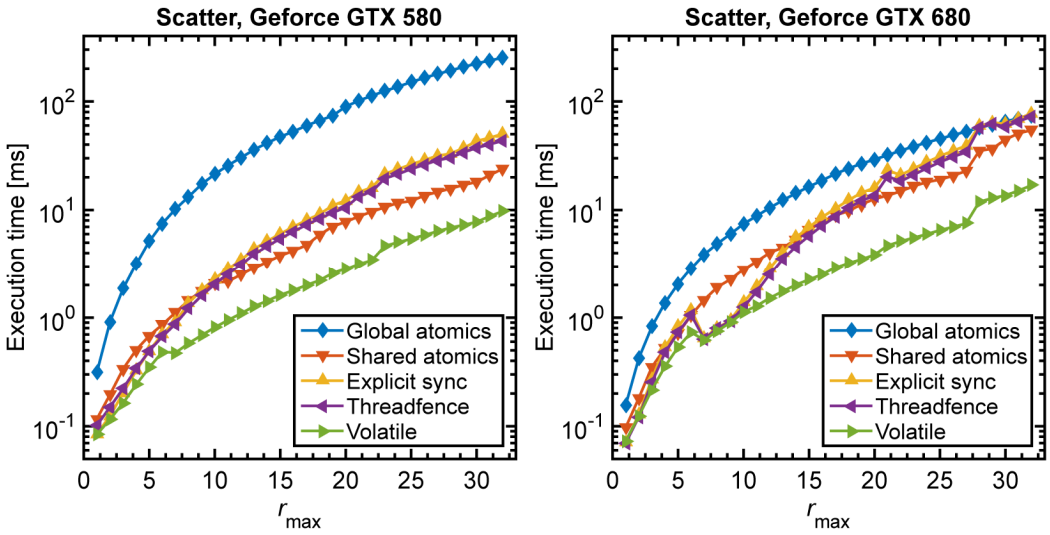

Figure 3. Execution times for the scatter implementations for different r_{max} on the Geforce GTX 580 (left) and GTX 680 (right). Only the data points for the best-performing combination of t_y and b_y are shown for each implementation.

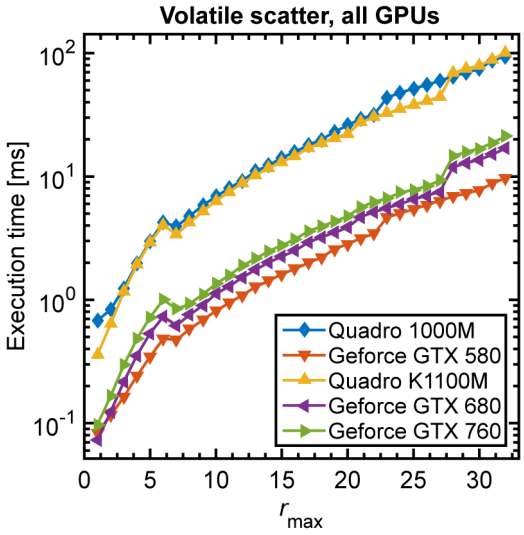

Figure 4. Comparison of the execution times for the volatile version of the scatter approach on all GPUs used. Curves corresponding to GPUs of the same architecture have the same shape but appear shifted along the y-axis.

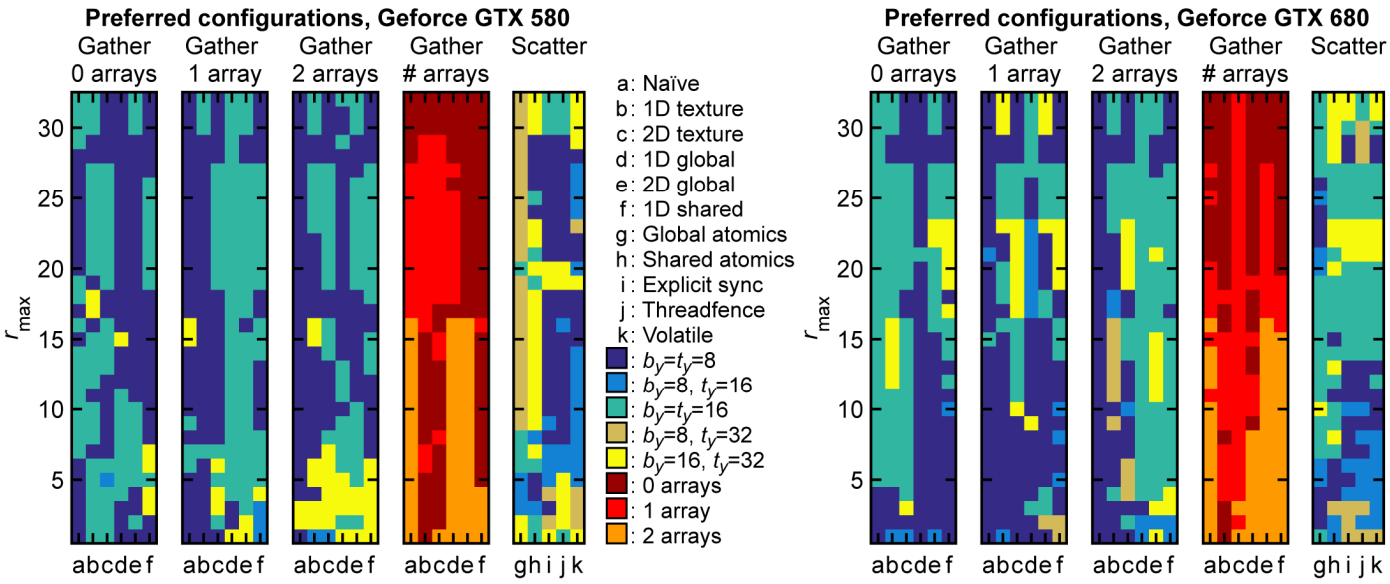

Figure 5. The preferred configurations for b_y and t_y for all implementations and the preferred number of shared memory arrays for the gather implementations on the Geforce GTX 580 (left) and GTX 680 (right) for different values of r_{max}.

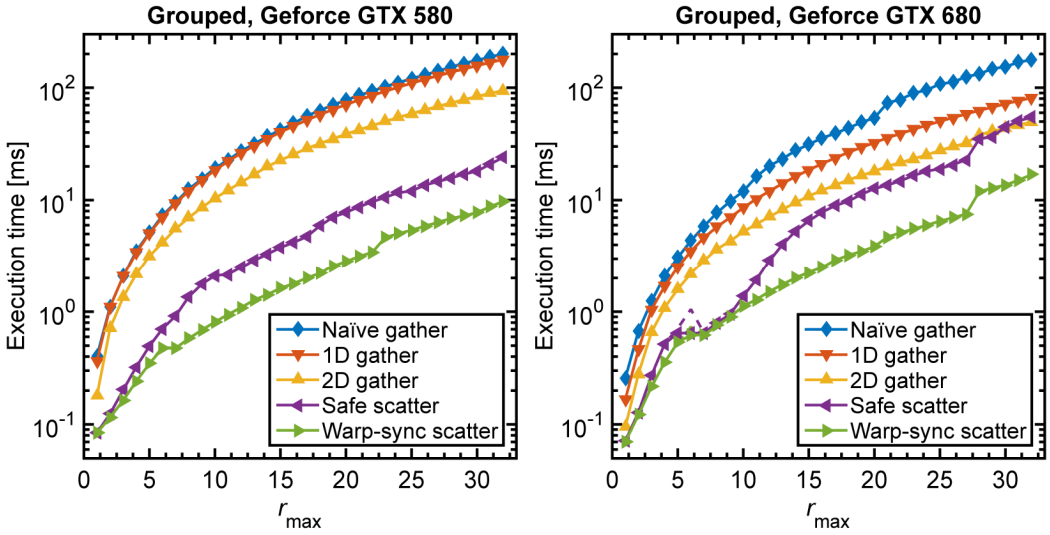

Figure 6. Execution times for the best-performing implementations in the different groups of gather and scatter implementations on the Geforce GTX 580 (left) and GTX 680 (right). Dashed lines show execution times for instantiations corresponding to the r_{max} indicated on the *x*-axis, solid lines show the fastest execution time for instantiations corresponding to r_{max} equal to or greater than the indicated value on the *x*-axis.

References

[1] Ahnesjö A. Collapsed cone convolution of radiant energy for photon dose calculation in heterogeneous media. Medical Physics 1989;16:577–92.

[2] Hong L, Goitein M, Bucciolini M, Comiskey R, Gottschalk B, Rosenthal S, et al. A pencil beam algorithm for proton dose calculations. Physics in Medicine and Biology 1996;41:1305–30.

[3] Ng J, Prager R, Kingsbury N, Treece G, Gee A. Modeling ultrasound imaging as a linear, shift-variant system. Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on 2006;53:549–63.

[4] Lauritsch G, Tam K, Sourbelle K. Solution to the long object problem by convolutions with spatially variant 1-D Hilbert transforms in spiral cone-beam computed tomography. Nuclear Science Symposium Conference Record, 2000 IEEE, vol. 2, 2000, p. 116–20.

[5] Kachelriess M, Watzke O, Kalender WA. Generalized multi-dimensional adaptive filtering for conventional and spiral single-slice, multi-slice, and cone-beam CT. Medical Physics 2001;28:475–90.

[6] Wiant D, Gersh J, Bennett M, Bourland J. Evaluation of the spatial dependence of the point spread function in 2D PET image reconstruction using LOR-OSEM. Medical Physics 2010;37:1169–82.

[7] Bitlis B, Jansson PA, Allebach JP. Parametric point spread function modeling and reduction of stray light effects in digital still cameras. Electronic Imaging 2007, 2007, p. 64980V–64980V.

[8] Lam EY. Image restoration in digital photography. Consumer Electronics, IEEE Transactions on 2003;49:269–74.

[9] Cobb ML, Hertz PL, Whaley RO, Hoffman EA. Space-variant point-spread-function deconvolution of Hubble imagery using the Connection Machine. SPIE's 1993 International Symposium on Optics, Imaging, and Instrumentation, 1993, p. 202–8.

[10] Alard C. Image subtraction using a space-varying kernel. Astronomy and Astrophysics Supplement Series 2000;144:363–70.

[11] Shaevitz JW, Fletcher DA. Enhanced three-dimensional deconvolution microscopy using a measured depth-varying point-spread function. Journal of the Optical Society of America A 2007;24:2622–7.

[12] Al Umairy SA, Van Amesfoort AS, Setija ID, Van Beurden MC, Sips HJ. On the use of small 2d convolutions on gpus. Computer Architecture, 2012, p. 52–64.

[13] Fialka O, Cadik M. FFT and convolution performance in image filtering on GPU. Information Visualization, 2006. IV 2006. Tenth International Conference on, 2006, p. 609–14.

[14] Terriberry TB, French LM, Helmsen J. GPU accelerating speeded-up robust features. Proceedings of 3DPVT, vol. 8, 2008, p. 355–62.

[15] Podlozhnyuk V. Image convolution with CUDA. NVIDIA Corporation 2007.

[16] Angel E, Jain AK. Restoration of images degraded by spatially varying pointspread functions by a conjugate gradient method. Applied Optics 1978;17:2186–90.

[17] Nagy JG, O'leary DP. Fast iterative image restoration with a spatially varying PSF. Optical Science, Engineering and Instrumentation'97, 1997, p. 388–99.

[18] Sawchuk AA. Space-variant image motion degradation and restoration. Proceedings of the IEEE 1972;60:854–61.

[19] Tan S, Dale JL, Johnston A. Performance of three recursive algorithms for fast space-variant Gaussian filtering. Real-Time Imaging 2003;9:215–28.

[20] Pratx G, Xing L. GPU computing in medical physics: a review. Medical Physics 2011;38:2685–97.

[21] Sriram V, Kearney D. A FPGA implementation of variable kernel convolution. Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT'07. Eighth International Conference on, 2007, p. 105–10.

[22] Wang Z, Han G, Li T, Liang Z. Speedup OS-EM image reconstruction by PC graphics card technologies for quantitative SPECT with varying focal-length fan-beam collimation. Nuclear Science, IEEE Transactions on 2005;52:1274–80.

[23] Cui J, Pratx G, Prevrhal S, Zhang B, Shao L, Levin CS. Measurement-based spatially-varying point spread function for list-mode PET reconstruction on GPU. Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2011 IEEE, 2011, p. 2593–6.

[24] Gomersall H, Hodgson D, Prager R, Kingsbury N, Treece G, Gee A. Efficient implementation of spatially-varying 3-D ultrasound deconvolution. Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on 2011;58:234–8.

[25] Hartung S, Shukla H, Miller JP, Pennypacker C. GPU acceleration of image convolution using spatially-varying kernel. Image Processing (ICIP), 2012 19th IEEE International Conference on, 2012, p. 1685–8.

[26] Fujimoto R, Kurihara T, Nagamine Y. GPU-based fast pencil beam algorithm for proton therapy. Physics in Medicine and Biology 2011;56:1319–28.

[27] J. Da Silva, R. Ansorge, R. Jena, Sub-second pencil beam dose calculation on GPU for adaptive proton therapy, Physics in Medicine and Biology 2015;60:4777–4795.

[28] Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ACM SIGARCH Computer Architecture News, vol. 38, 2010, p. 451–60.