Table 1: *Properties of the presented algorithms, for row and column processing of an $h \times w$ image with causal and anticausal recursive filters of order r, assuming block-size b, and p SMs with c cores each. For each algorithm, we show an estimate of the number of steps required, the maximum number of parallel independent threads, and the required memory bandwidth.*

Alg.	Step complexity	Max. # of threads	Bandwidth
RT	$\frac{hw}{cp} 4r$	h, w	$8hw$
2	$\frac{hw}{cp}\left(8r + 4\frac{1}{b}(r^2+r)\right)$	$\frac{1}{b}hw$	$(9 + 16\frac{r}{b})hw$
4	$\frac{hw}{cp}\left(8r + 6\frac{1}{b}(r^2+r)\right)$	$\frac{1}{b}hw$	$(5 + 18\frac{r}{b})hw$
5	$\frac{hw}{cp}\left(8r + \frac{1}{b}(18r^2+10r)\right)$	$\frac{1}{b}hw$	$(3 + 22\frac{r}{b})hw$
SAT	$\frac{hw}{cp}(8 + \frac{3}{b})$	$\frac{1}{b}hw$	$(3 + \frac{8}{b} + \frac{2}{b^2})hw$

Recursive doubling [Stone 1973] is a well known strategy for first-order recursive filter parallelization we can use to perform intra-block computations. The idea maps well to GPU architectures, and is related to the tree-reduction optimization employed by efficient one-dimensional parallel scan algorithms [Sengupta et al. 2007; Dotsenko et al. 2008; Merrill and Grimshaw 2009]. Using a block size b that matches the number of processing cores c, the idea is to break the computation into steps in which each entry is modified by a different core. Using recursive doubling, computation of b elements completes in $O(\log_2 b)$ steps.

The extension of recursive doubling to higher-order recursive filters has been described by Kooge and Stone [1973]. The key idea is to group input and output elements into r-vectors and consider equation (1) in the matrix form of (52) in appendix A. Since the algebraic structure of this form is the same as that of a first-order filter, the same recursive doubling structure can be reused.

8 Results

Table 1 summarizes the main characteristics of all the algorithms that we evaluated, in terms of number of required steps, the progression in parallelism, and the reduction in memory bandwidth.

Our test hardware consisted of an NVIDIA GTX 480 with 1.5GB of RAM (480 CUDA cores, $p = 15$ SMs, $c = 32$ cores/SM). All algorithms were implemented in *C for CUDA*, under CUDA 4.0. All our experiments ran on *single-channel* 32-bit floating-point images. Image sizes ranged from 64^2 to 4096^2 pixels, in 64-pixel increments. Measurements were repeated 100 times to reduce variation. Note that small images are solved in sequence, *not* in batches that could be processed independently for added parallelism and performance.

First-order filters As an example of combined row-column causal-anticausal first-order filter, we solve the bicubic B-spline interpolation problem (see also figure 7(top)). Algorithm RT is the original implementation by Ruijters and Thévenaz [2010] (available on-line). Algorithm 2 adds blocking for memory coalescing, inter-block parallelism, and kernel fusion. (Algorithms 1 and 3 work on 1D input and are omitted from our tests.) Algorithm 4 employs overlapped causal-anticausal processing and fused row-column processing. Finally, algorithm 5 is fully overlapped. Performance numbers in figure 4(top) show the progression in throughput described throughout the text. As an example, the throughput of algorithm 5 solving the bicubic B-spline interpolation problem for 1024^2 images is 4.7GiP/s (gibi-pixels per second). Each image is transformed in just 0.21ms or equivalently at more than 4800fps. The algorithm appears to be compute-bound. Eliminating computation and keeping data movements, algorithm 5 attains 13GiP/s (152GB/s) on large images, whereas with computation it reaches 6GiP/s (72GB/s).

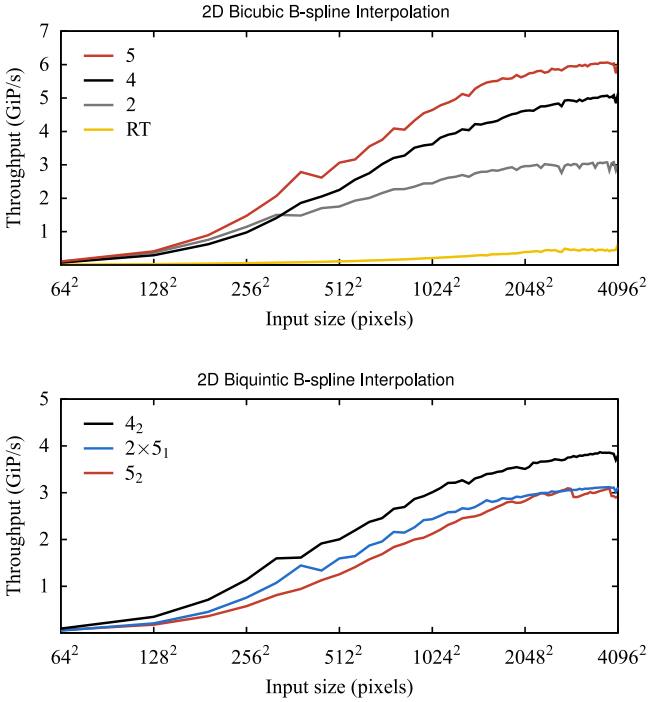

Figure 4: *Throughput of the various algorithms for row-column causal-anticausal recursive filtering. (Top plot) First-order filter (e.g. bicubic B-spline interpolation). (Bottom plot) Second-order filter (e.g. biquintic B-spline interpolation).*

Second-order filters The second-order, causal-anticausal, row-column separable recursive filter used in our tests solves the biquintic B-spline interpolation problem. Figure 4(bottom) compares three alternative structures: $2\times5_1$ is a cascaded implementation using two fused fully-overlapped passes of first-order algorithm 5, 4_2 is a direct second-order implementation of algorithm 4, and 5_2 is a direct fully-overlapped second-order implementation of algorithm 5. Our implementation of 4_2 is the fastest, despite using more bandwidth than 5_2. The higher complexity of second-order equations slows down stages 5.4 and 5.5 substantially. We believe this is an optimization issue that may be resolved with a future hardware, compiler, or implementation. Until then, the best alternative is to use the simpler and faster 4_2. It runs at 3.1GiP/s for 1024^2 images, processing each image in less than 0.32ms, or equivalently at more than 3200fps.

Precision A useful measure of numerical precision in the solution of a linear system such as the bicubic B-spline interpolation problem is the *relative residual*. Using random input images with entries in the interval $[0, 1]$, the relative residual was less than 2×10^{-7} for all algorithms and for all image sizes.

Summed-area tables Our overlapped summed-area table algorithm was compared with the algorithm of Harris et al. [2008] implemented in the CUDPP library [2011], and with the multi-wave method of Hensley [2010]. We also compare against a version of Hensley's method improved by two new optimizations: fusion of processing across rows and columns, and storage of just "carries" (e.g. $P_{m,n}(\bar{Y})$) between intermediate stages to reduce bandwidth. As expected, the results in figure 5 confirm that our specialized overlapped summed-area table algorithm outperforms the others.

Recursive Gaussian filters As mentioned in section 1, Gaussian filters of wide support are well approximated by recursive filters (see also figure 7(bottom)). To that end, we implemented the third-order