Acceleration of Bilateral filtering algorithm for manycore and multicore architectures

Dinesh Agarwal, Sami Wilf, Abinashi Dhungel, and Sushil K. Prasad
Georgia State University
Department of Computer Science
Atlanta, Georgia 30303

***Abstract*—This work explores multicore and manycore acceleration for the embarrassingly parallel, compute-intensive Bilateral filtering kernel. For manycore architectures, we have created a pair-symmetric algorithm to avoid redundant calculations and for multicore architectures we improve the algorithm by use of low level single instruction multiple data (SIMD) parallelism across multiple threads. We present empirical data evidencing the performance gains we achieved over variety of hardware architectures including Nvidia GTX280, AMD Barcelona, AMD Shanghai, Intel Harpertown, AMD Phenom, Intel Core i7 quad core, and Intel Nehalem 32 core machines. The best speedup achieved was a 235.5x speedup by our CUDA-based implementation of our pair-symmetric algorithm running on Nvidia's GTX280. Our CPU multicore implementations resulted in a speedup of up to 38x using 16 cores of AMD Barcelona each with 4-stage vector pipeline, up to 23x using 8 cores for Intel Harpertown, and up to 26x using 32 cores of Intel Nehalem multicore machines compared to a compiler-optimized code.**

I. Introduction

The recent advances in parallel computing and architecture design have provided a series of avenues for application performance improvements. However, unlike the days when applications automatically became faster with increased clock frequency, newer chips require programmers and researchers to revisit their algorithm designs to harness the power of underlying architectures. Moreover, there is no well defined list of guidelines that can be followed for each application, which makes it challenging to deploy existing applications, even fundamental ones, to a myriad of available multicore and manycore architectures.

Our work explores efficient parallel implementation of Bilateral filtering kernel[1] proposed by Tomasi and Manduchi [2], on multicore and manycore architectures. Bilateral filtering kernel is used to smooth an image without blurring its edges. It is ubiquitous in various image processing applications in various contexts such as denoising [3], texture editing and relighting [4], tone management [5], stylization [6], optical-flow estimations [7], and demosaicking [8] etc.

A faster approximation to the original algorithm [9] extends the image to 3D space by treating the pixel intensity as the third dimension and using a linear shift-invariant convolution in 3D. A separable implementation [10] applies the filter kernel first to one of the image dimensions and the intermediate result is filtered again on other dimensions. This reduces the complexity from $O(N * (m^d))$ to $O(N * d)$, where N is the total number of pixels in the image and d is the dimensions of the image. Yang et al. [11] have reported a constant time algorithm by dividing the Bilateral filtering kernel into a constant number of time spatial kernels. However, we do not parallelize these methods since our interest is to develop a parallelization method that can be applied, in general, to any spatially invariant and inseparable filtering algorithm.

In this paper, we propose a set of optimizations and a non-naive algorithm for Bilateral filtering kernel. Specifically, we introduce a pair-symmetric algorithm which has the theoretical potential to reduce processing time to half. We propose architecture specific optimizations, such as exploiting the unique capabilities of special registers available in modern multicore architectures and the rearrangement of data access patterns as per the computations to exploit special purpose instructions. We also propose optimizations pertinent to Nvidia's CUDA, including utilization of CUDA's implicit synchronization capability and the maximization of single-instruction-multiple-thread efficiency.

In order to assess the efficacy of our optimization techniques, our experimental testbed includes a comprehensive set of multicore and manycore architecture chips including the Nvidia GTX280, AMD quad-core 2376 (Shanghai), AMD quad-core 8350 (Barcelona), Intel quad-core Xeon E5410 (Harpertown), Intel eight-core Xeon X7560 (Nehalem-EX), Intel quad-core Core i7, and AMD Phenom II six-core architectures.

Rest of the paper is organized as following: Section II defines the Bilateral filtering kernel and introduces a naive algorithm and our pair-symmetric algorithm for Bilateral filtering. Section III introduces the optimizations employed to devise the pair-symmetric algorithm for CUDA architectures. Section IV introduces the optimizations for CPU architectures and establishes the rationale behind ignoring traditional memory related optimizations for Bilateral filtering kernel. Our hardware and software testbed for experiments is detailed in Section V, immediately followed by Section VI that reports the performance of Bilateral filtering kernel on both multicore and manycore architectures. Section VII concludes this paper with remarks on general applicability of our optimizations.

II. Bilateral Filtering Kernel

Filtering, an essential application of image processing, is carried out by a relatively small window, known as the

[1]Our code is available under GNU GPL license from http://code.google.com/p/blfilter/.

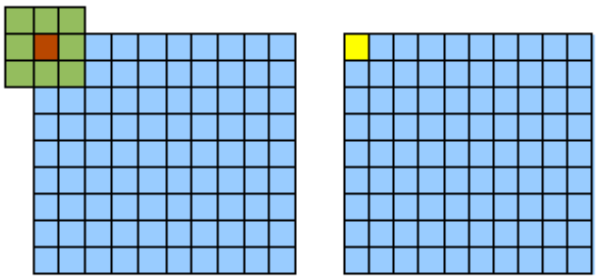


Fig. 1. Example of a filter at the starting position of the image

filter ("Filter" or "Window"), which iterates over the source image, performing the same execution at each step of the iteration. The elements that the Filters cover in a given iteration makeup the neighborhood ("Neighborhood") and are neighbors ("Neighbors" or "Neighboring Pixels") of the Filter's center element ("Center" or "Center Pixel").

Figure 1 illustrates the starting position of the Filter as commonly seen in standard, image processing algorithms; the source image is on the left and the destination image on the right is used to store the result of the Filter's computation at the respective Center's position.

Bilateral filtering combines two different kernels - the spatial kernel and the photometric kernel. With the photometric kernel, when the Center of the Filter rests on a pixel that boarders or is a part of a pictorial edge, e.g., an outline of an object or person against a contrasting background, the Neighbors that make up the other (contrasting) side of the edge have an insignificant impact on the Center Pixel's corresponding output pixel, which is stored in the destination array storing the processed image. Furthermore, when the Center is not bordering, and is not a part of a pictorial edge, the Neighbors contribute equally and the noise is filtered out. Hence, pictorial edges maintain sharpness while noise in the pictorial regions are filtered, resulting in more appealing images. Mathematically, the Bilateral filter can be defined in terms of the spatial kernel and photometric kernel as follows [2]:

Spatial kernel:

$$h(x) = k_d^{-1}(x) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\xi) c(\xi, x) d\xi \tag{1}$$

$$k_d(x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\xi, x) d\xi \tag{2}$$

Photometric kernel:

$$h(x) = k_r^{-1}(x) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\xi) s(f(\xi), f(x)) d\xi \tag{3}$$

$$k_r(x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} s(f(\xi), f(x)) d\xi \tag{4}$$

The combined kernel, which is the Bilateral filter, is the product of spatial kernel and photometric kernel and is described as:

$$h(x) = k^{-1}(x) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\xi) c(\xi, x) s(f(\xi), f(x)) d\xi \tag{5}$$

$$k(x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\xi, x) s(f(\xi), f(x)) d\xi \tag{6}$$

Function $c(\xi, x)$ measures the $geometric\ closeness$ between neighborhood Center x and a Neighbor ξ. Function $s(f(\xi), f(x))$ measures the $photometric\ similarity$ (similarity in pixel intensity) between the same. The influence of a Neighbor on the Filter's output computation, for a Center Pixel, is calculated as a product of spatial kernel and the photometric kernel. Thus, Neighbors with similar photometric properties influence the Center Pixel more than those with less similar photometric properties. This results in edge-preserved image smoothing, which is the purpose of the Bilateral filter. It should be noted, however, that the foregoing equations are mathematical representations for filtering of infinite-precision images. As infinite-precision is a concept more pertinent to the analog domain and does not exist in the digital domain, each double integral in equations 1-6 must be understood as the summation of a finite set of calculations, i.e. one calculation per Neighbor and one set of calculations per Center Pixel, or put another way, one set of calculations per Filter iteration. A formal expression is given below:

$$filtered\ pixel\ x = \frac{\sum_{i=1}^{N} f(\xi_i) \cdot c(\xi_i, x) \cdot s(f(\xi_i), f(x))}{\sum_{i=1}^{N} c(\xi_i, x) \cdot s(f(\xi_i), f(x))} \tag{7}$$

In addition to restating the Bilateral filter's mathematical model in terms of a finite summation, Equation 7 reveals the data dependency that exists in the Bilateral filter. Specifically, Equation 7 shows that before the filtered pixel can be calculated by dividing the numerator by the denominator, both the summation in the numerator and denominator must first be calculated. Consequently, the Bilateral filter has two data dependencies. As explained later in this paper, the pair-symmetric algorithm significantly increases the memory consumption in order to manage data dependencies but pays off by cutting, in half, the total number of executions.

Another observation taken into account is that the values of Equation 1, for the Neighborhood, can be computed in advance as the values depend on a simple Gaussian probability distribution function. However, the values of Equation 3 and consequently of Equation 5 cannot be known a priori as they depend on the similarity of signals between the current Center Pixel and the Neighborhood. Moreover, as the size of the Window increases, which only happens by choice of the user as the size is user defined, the number of computations per pixel grows at a rate of $O(FilterWindowWidth^2)$ where FilterWindowWidth is the diameter of the Window. By reason of the foregoing, the bulk of the time spent in processing an image using the Bilateral filter is in the arithmetic-intensive computations, because the number of writes to the destination image is constant as the rate of growth of Center-Neighbor computations with respect to the Window size increase.

A. A Naive Approach to Bilateral filtering kernel

The naive stencil algorithm is the standard model that is predominantly used in Bilateral filtering kernel. Previous

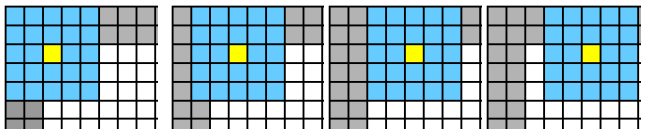


Fig. 2. Example of a naive Bilateral filter iterations

implementations [9], [10] of Bilateral filtering kernel improves the hardware dependent aspects and hence expend resources on achieving performance gains that can not match the effort taken. Here, we address a root problem of the standard naive algorithm by proposing a new algorithm, and then develop hardware dependent improvements for further speedup. Algorithm 1 is the pseudo-code for the standard naive Bilateral filtering kernel.

Algorithm 1 Naive Bilateral Filter Algorithm

for all element $s_{i,j}$ in source image s **do**
  $X_1 \leftarrow 0$
  $X_2 \leftarrow 0$
  for $k = -filter\_hw$ **to** $filter\_hw$ **do**
    for $l = -filter\_hw$ **to** $filter\_hw$ **do**
      $temp \leftarrow e^{C_1(k^2+l^2)+C_2(s[i,j]-s[i,j+l])^2}$
      $X_1 \leftarrow X_1 + temp \cdot s[i+k, j+l]$
      $X_2 \leftarrow X_2 + temp$
    end for
  end for
  output$[i,j] \leftarrow X_1/X_2$
end for

B. Pair-symmetric Bilateral filtering kernel

The pair-symmetric algorithm, shown in Algorithm 2, reduces the amount of Center-Neighbor computations by half. Thus, it can theoretically cut the processing time in half at the cost of increased memory requirement, which increases with the amount of dependencies. Its implementation is non-trivial due to potential race-conditions.

Our pair-symmetric Bilateral filtering algorithm is premised on the fact that each pixel assumes the role of both a Center Pixel and a Neighboring Pixel, and that the codes are symmetric with respect to Center-Neighbor paired computations. For example, in the Bilateral filter, $c(\xi, x)$ returns the same value as $c(x, \xi)$ and $s(f(\xi), f(x))$ returns the same value as $s(f(x), f(\xi))$. It then follows that the naive Bilateral filtering algorithm dictated in Algorithm 1 above, although simple, performs twice as many calculations than necessary.

The Pair-symmetric algorithm eliminates this redundancy that is inherent in the naive Bilateral filtering algorithm. Instead of reading from all Neighbors, our pair-symmetric algorithm reads from only half of the Neighbors. The result of each Center-Neighbor paired computation is then added to the X_n register, which eventually obtains the result of data dependency n. This is trivial as the naive Bilateral filtering algorithm already does this. What our pair-symmetric algorithm does unique is that the result of each Center-Neighbor paired computation is also added to and stored in the Neighbor's position in a temporary array ("Map") that is used for storing the results of a data dependency. So, there are n Maps for n number of dependencies.

After the data dependency conditions are met, the function that returns the output data from the dependencies iterates over the Maps. As applied to the Bilateral filter, the data dependency lies in calculating each pixel's numerator and denominator that is used to derive the pixel's output result. After the numerators and denominators are derived, a function iterates and divides the corresponding elements of the two arrays to return the output.

Algorithm 2 Pair-symmetric Bilateral Filtering Algorithm

/* D is the set of dependency arrays */
for all element $s_{i,j}$ in source image s **do**
  $D_1[i,j] \leftarrow 0$
  $D_2[i,j] \leftarrow 0$
end for
for all element $s_{i,j}$ in source image s **do**
  $X_1 \leftarrow 0$
  $X_2 \leftarrow 0$
  for $k = 1$ **to** $filter\_hw$ **do**
    for $l = -filter\_hw$ **to** $filter\_hw$ **do**
      $temp \leftarrow e^{C_1(k^2+l^2)+C_2(s[i,j]-s[i+k,j+l])^2}$
      $X_1 \leftarrow X_1 + temp \cdot s[i+k, j+l]$
      $X_2 \leftarrow X_2 + temp$
      $D_1[i+k, j+l] \leftarrow D_1[i+k, j+l] + temp \cdot s[i,j]$
      $D_2[i+k, j+l] \leftarrow D_2[i+k, j+l] + temp$
    end for
    syncthreads();
  end for
  for $l = -filter\_hw$ **to** 0 **do**
    $temp \leftarrow e^{C_1 l^2+C_2(s[i,j]-s[i,j+l])^2}$
    $X_1 \leftarrow X_1 + temp \cdot s[i, j+l]$
    $X_2 \leftarrow X_2 + temp$
    $D_1[i, j+l] \leftarrow D_1[i, j+l] + temp \cdot s[i,j]$
    $D_2[i, j+l] \leftarrow D_2[i, j+l] + temp$
  end for
  $D_1[i,j] \leftarrow D_1[i,j] + X_1$
  $D_2[i,j] \leftarrow D_2[i,j] + X_2$
end for
for all element $s_{i,j}$ in source image s **do**
  $output[i,j] \leftarrow D_1[i,j]/D_2[i,j]$
end for

III. CUDA-based Implementation Details

As Bilateral filtering algorithm belongs to the class of embarrassingly parallel algorithms, CUDA is well suited for its implementation. However, in order to effectively implement such an algorithm in CUDA, one must understand how CUDA operates, which is significantly different from any CPU.

A. Implicit Intra-Warp Synchronization and Memory Access Patterns

CUDA's intra-warp implicit synchronization capability is the cornerstone of our CUDA implementation of our pair-symmetric algorithm. Without implicit synchronization, our

pair-symmetric algorithm would suffer significantly over CUDA because it would require expensive synchronization calls per Center-Neighbor Filter computation.

In CUDA, threads are organized into groups called warps. Each warp executes one instruction at a time for its own group of threads. Because all threads in a warp share the same program counter and are therefore driven by the same sequential instruction stream, all threads in a warp are always in sync with each other without requiring explicit synchronization calls. Therefore, synchronization between threads in a warp is implicit.

To take full advantage of CUDA's implicit intra-warp synchronization feature, divergent branching between threads in the same warp must be avoided. This is so because all the threads in the same warp share only one program counter. A divergent branch will cause some or most of the threads to be idle as the program counter runs through the branch and drives only those threads that meet the branch condition to work. An example of a divergent branch can be a simple if-statement such as: $if\ (threadIdx.x < 5)\ \{...\}\ else\ \{...\}$.

Divergent branching will commonly occur in memory access patterns in which global memory operations are not coalesced and shared memory operations are not free of memory bank conflicts. For example, if eight threads were to read from eight non-contiguous segments of global memory - commonly seen in strided access patterns like array [threadIdx.x * 2] - divergent branching would occur because a warp of threads can only read from one segment of memory per instructional transaction, and that segment's word-length is hardware defined.

In other words, a memory transaction will be serialized unless all the threads in the warp to which the transaction belongs are aligned consecutively and fit within a segment of memory. Therefore, in order for all the threads of a warp to read/write from global memory in a single transaction, the threads must correspond read/write operations to aligned contiguous addresses within a segment of memory. For Bilateral filtering algorithm, warps should correspond to horizontal rows of pixels for an [i*jMax + j] representation of a 2D image, for if threads in a warp were to operate on pixels in a vertical order, there would be a separate transaction for each thread in the warp.

While intra-warp synchronization is guaranteed, the rate at which warps stream through their sequence of instructions is unpredictable. As a consequence, work must be arranged such that warps can work as independent from each other as possible, in order to minimize explicit synchronization calls as shown in Figure 5. This pooling of work is achieved by threads reading from addresses incrementally along the horizontal axis, and only making an explicit synchronization call per vertical shift to the next row once the work requiring the warp's access to the current row of input data has completed. This design works because horizontal shifts are implicitly synchronized as there is only one warp per row, but a vertical shift without an explicit synchronization call will result in a warp trespassing into the territory of another warp, thereby causing race conditions as threads from two different warps will consequently be prone to write to the same memory location.

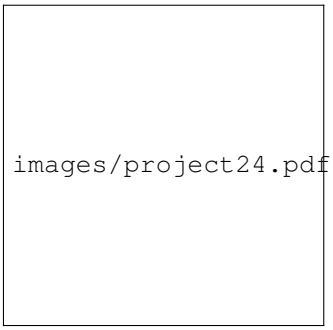


Fig. 3. Illustration of our CUDA implementation of pair-symmetric processing using implicit intra-wap synchronization for iterating through the pixel-cells for each thread-block's assigned tile.

Fig. 4. Illustration of our CUDA implemented pair-symmetric algorithm processing a 24 x 12 pixel image using 2 x 2 thread-blocks each with 4 x 3 threads and with the half-width parameter set to 3 pixels.

B. Shared Memory and Tile Processing

CUDA features three types of memory with read/write capability suitable for general-purpose computing on GPU: (i) global memory, which is accessible by all threads in all thread blocks, (ii) shared memory, which is provided separately for each block of threads to share among themselves, and (iii) registers, which are separate and distinct for each thread and of which each is only accessible by the thread to which it belongs. Of these three types of memory, the slowest is global memory, which unfortunately is where the host must first send the input data before the GPU can begin

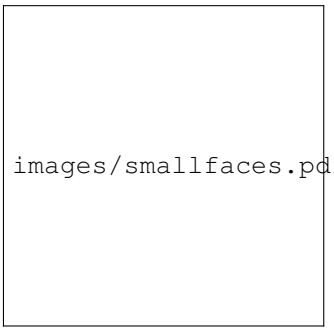


Fig. 5. An illustrated overview of the bilateral filtering process. The six raster images of processed tiles reflect the successive work of the four concurrent thread-blocks iteratively carried out over the padded source-image.

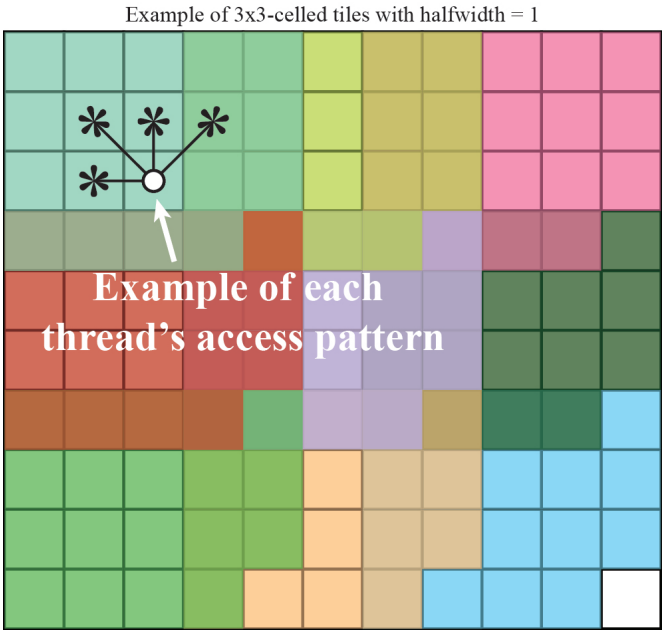


Fig. 6. Tiling for Pair-Symmetric Stencil Processing with each tile having a unique color and overlapping tiles have a resultant color that results from mixing colors of overlapping tiles. The pair-symmetric thread access pattern ensures exclusive memory writes.

its work. So, in order to see any meaningful performance in CUDA, data stored in global memory must be transferred to faster memory–i.e., shared memory, which is hundreds of times faster. However, the trade-off is that the shared memory size is much smaller than the global memory size, and this fact compels a tile-based approach to the implementation of Bilateral filtering algorithm.

Our tile-based processing approach is self-explanatory. Equal sized partitions of the input data are processed, and then any remaining data on and between the edge and the last partition that is closest to (but not falling over) the edge boundary of the 2-dimensional data representation is processed. With respect to the naive Bilateral filtering algorithm, tile-based processing is trivial and needs no further explanation because simply partitioning the space and processing it will suffice. However, for the pair-symmetric algorithm that we propose, tile-based Bilateral filtering algorithm requires adjustments in the order in which tiles are processed in order to guarantee the exclusion of race conditions.

If processes were assigned to adjacent tiles, race conditions would occur because the edge cells of the tiles would overlap each other due to intersecting half-width periphery regions, as illustrated in Figure 6. To get around this problem, simultaneously processed tiles can be spaced apart in strides such that no two tiles being processed at the same time intersect with one another. An example of such a technique is illustrated in Figure 7 (the tiles are flipped sideways in this rendering).

IV. CPU-based Implementation Details

A. Memory to computation ratio of Bilateral filtering kernel

Unlike traditional regular embarrassingly parallel stencil kernel codes where memory management plays a pivotal role

Fig. 7. Input image after one iteration of pair-symmetric algorithm. Tiles do not have perfect boundaries due to overlaps.

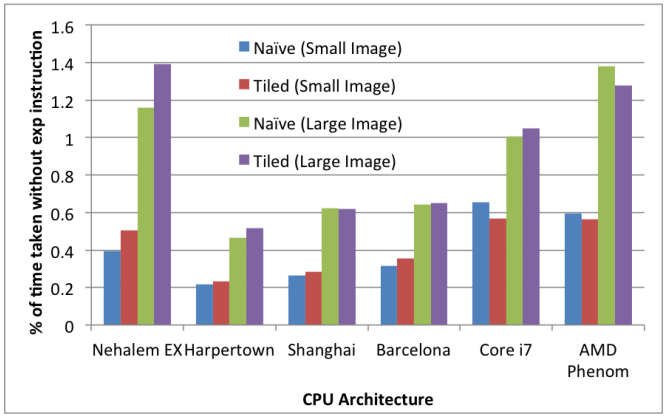


Fig. 8. Percentage of time taken by sequential kernel when run without exp instruction, $filter\ radius$=10.

in realizing performance, Bilateral filtering kernel is extremely computationally intensive with only a meager amount of time spent in memory accesses. Bilateral filtering kernel requires computation of $exponential$ function (exp function) which is a major component of time spent in processing. Figure 8 shows the percentage of time taken by Naive Bilateral filtering algorithm [2] and an improved algorithm employing tiled-memory-access pattern on different platforms for a 512x512 pixels small image and a 3000x3000 pixels large image after the exp function was commented out from the computation. The only purpose of this experiment is to justify the fact that traditional memory-access-pattern related optimizations can not improve the performance of Bilateral filtering kernel.

It can be seen that after taking the exp instruction out of the computation, the total sequential execution time is consistently less than 2% of the original execution time with exp instruction in place. For the smaller image it is strictly less than 1% of the original time with exp instruction in place. To

be precise it is 1.39% maximum on Intel Nehalem for the larger image and 0.65% for the smaller image on Core i7 processor. The lower memory transfer time for smaller image is due to the fact that the smaller image fits into the cache memory entirely and hence there are no repeated data transfers between the processor and the main memory.

Since the exp instruction does not involve transfer of data from main memory, it follows from this experiment that memory reuse and memory access pattern related optimizations are not of much help to speed up Bilateral filtering kernel. What Amdahl's law claims, for inherently sequential portion of a parallel algorithm, can be applied here.
The total speedup achieved by optimizations that deal with memory management is limited by the amount of time spent in memory transfers. Therefore, in this case, traditional optimizations such as tiling/blocking, cache efficient initializations, prefetching, and other NUMA optimizations can only result in a speed up of at most 2% and hence we have concentrated on optimizations that deal with the complexities of computational aspects of Bilateral filtering kernel.

B. SIMD optimizations

$Intel$'s Streaming SIMD Extension (SSE) is a technique to exploit micro-level parallelism in x86 based processors using SIMD capabilities. SSE can be ideal for tasks that involve repetitive computation such as the 3D heat equation or any other stencil kernels such as Jacobi iterations [12]. The matrix multiplication kernel was implemented in [13], [14] for matrices with integers and floating point numbers. SSE instructions have also been used for image or template matching algorithms such as normalized cross correlation ("NCC") [15].

Modern day CPUs use SSE that is an extension of MMX technology and is fundamentally similar to MMX. It has 8, 128-bit registers named XMM0 through XMM7. SSE instructions can operate in parallel on data packed in these registers thus providing a theoretically maximum speedup of 16 when working with 1 Byte data structures or 8 when working with 2 Bytes long data structures and so on. There are other extensions of SSE such as SSE2, SSE3, and SSE4.2. Moreover, recent extension of SSE, named Advanced Vector Extensions ("AVX"), uses 256-bit data registers and thus a theoretical peak speedup of 32 is possible.

Our CPU based implementation of Bilateral filtering algorithm explores the use of special purpose registers using both assembly level programming and SSE intrinsics. The advantage of using assembly language is that a programmer can have explicit control of underlying hardware. However, programmer productivity is affected and an in-depth knowledge of underlying hardware is essential.

Although SSE intrinsics are wrappers for assembly language, it is preferred to use SSE intrinsics due to ease of coding and also due to efficient utilization of available assembly level instructions without knowing the low level details of underlying architecture. A programmer, working with intrinsics, can specify the intrinsic and the appropriate instruction will automatically replace this intrinsic during compilation. Moreover, there are some operations for which there are no assembly instructions and hence in case of assembly level programming the code size can grow to such an extent that readability gets compromised.

Although the modern compilers usually try to improve the performance by automatically vectorizing, unrolling loops, prefetching, and SIMDization, we were still able to get significant performance improvements alongside compiler optimizations.

C. Reduction Methods

Sum by reduction is a technique to compute sum of multiple elements usually stored in contiguous memory locations. There are assembly instructions that can add horizontally packed, single precision, floating point numbers. We used $haddps$ assembly instruction that takes two (source and destination) 128 bit operands, and adds the single precision floating point numbers stored in first and second $dword$ of destination and source operand and stores it in first and third $dword$ of the destination operand respectively. Similarly, the third and fourth $dword$ of destination and source operand are added and stored in second and fourth $dword$ of the destination operand respectively. Both source and destination values must be stored in MMX registers. The SSE intrinsic to perform horizontal add is $\_mm\_hadd\_ps$. It takes two 128 bit operands, each of which is treated as four 32-bit floating point elements. The result of this operation on operand X(X0, X1, X2, X3) and Y(Y0, Y1, Y2, Y3) is (Y3 + Y2, Y1 + Y0, X3 + X2, X1 + X0).

V. Experimental Testbed

In order to evaluate our implementation we have tried it on a range of multicore and manycore chips. Table I summarizes architectural details of systems we used during our experiments.

A. Hardware

1) Intel Xeon: We have used two different Xeon chips, E5410 (Harpertown) and X7560 (Nehalem EX). Harpertown is a dual-socket, dual-processor relatively old system with a clock frequency of 2.33 GH_z. Harpertown does not have Intel's hyperthreading (HT) technology which means the maximum number of threads is equal to the number of cores available. Intel Nehalem EX is considerably newer chip having eight-cores processing at a clock frequency of 2.27 GH_z. We had access to a Linux cluster from Intel's Manycore Testing Lab (MTL) containing four sockets of Nehalem EX chips thereby making 32 cores available to experiment with. The processors are connected to each other via Intel QPI (Quickpath interconnect) links that can transfer data at 6.4 GT/s. Both of these machines support 64 bit instructions.

2) AMD Opteron: AMD Opteron 2376 (Shanghai) chip has private L1 and L2 caches of size 128KB (64 D + 64 I) and 512KB respectively per core along with a 6MB cache shared among all of the cores. It has a clock frequency of 2.30 GH_z and it is a quad-core architecture. Our compute node has two of Shanghai chips making it a total of eight cores to be used

Model	**Clock frequency**	**cores per chip**	**chips per node**	**cores per node**	**SIMD support**	**Cache/Memory**	Compiler
Intel Xeon E5410	2.33 GH_z	4	2	8	SSE, SSE2	12 MB	icpc 11.1
Intel Xeon X7560	2.27 GH_z	32	1	32	SSE, SSE2	24 MB	icpc 11.1
Intel Core i7-870	2.93 GH_z	4	1	4	SSE, SSE2, SSE3, SSE4.2	8MB	icpc 12.0.2
AMD Opteron 2376	2.3 GH_z	4	2	8	SSE, SSE2	6MB	icpc 11.1
AMD Opteron 8350	2.0 GH_z	4	4	16	SSE, SSE2	2MB	icpc 11.1
AMD Phenom II X6 1045T	2.7 GH_z	3	1	6	SSE, SSE2, SSE4a	6MB	icpc 12.0.2
NVidia GeForce GTX 280	1296 MHz	240	1	240	-	1 GB GDDR3	CUDA 2.0

TABLE I
ARCHITECTURAL DETAILS OF MULTICORE CHIPS EMPLOYED FOR EXPERIMENTS

for experiments. AMD Opteron 8350 (Barcelona) is similar to Shanghai in many aspects but it has a clock frequency of 2.0 GH_z and the shared cache is only 2MB. Our compute node contains 4 of Barcelona chips and thus we are able to run 16 threads on this node.

3) Intel Core i7: Intel Core i7 - 870 supports SSE 4.2 and has AVX technology. The clock frequency is an impressive 2.93 GH_z and it can operate at a maximum of 3.6 GH_z supported by $Intel$'s power boost technology. Its architecture supports hyperthreading which lets an application to execute 8 threads in total; however, we only used one thread per core. It contains 2 memory channels that provide a theoretical maximum memory bandwidth of 21 GB/s.

4) AMD Phenom II: AMD Phenom is a six core processor with each core capable of operating at 2.7 GH_z. If only three cores are in operation it can run at a maximum of 3.2 GH_z. It boasts 128 KB (64 D + 64 I) L1 cache, and 512 KB L2 cache per core in addition to a 6MB L3 cache shared among all six cores. The cache latencies are 3, 13, and 47 clock cycles for L1, L2, and L3 caches respectively. The cache line size is 64 bytes, L1 caches are 2 way set associative, L2 caches are 16-way set associative and L3 cache is 48-way set associative.

5) NVidia GeForce GTX 280: GTX 280 by NVidia is a GPU chip with 240 CUDA cores with 602 MH_z graphics clock and 1296 MH_z processor clock. GTX 280 is equipped with a 1 GB standard memory with 1107 MH_z memory clock and 141.7 GB/s memory bandwidth. GTX 280 boasts a memory interface width of 512-bit with texture fill rate of 48.2 billion/second.

B. Software Framework

Our implementation for multicores is written in $C++$ and uses OpenMP for parallelization; the GPU implementation uses CUDA. To enrich supported image formats, we use $CImg$ library.

1) Algorithm variations on multicore architectures: We have implemented a number of variations of Bilateral filtering kernel for multicore chips, results of which are reported in section VI. Our GPU implementation uses naive algorithm and the pair symmetric algorithm. For multicore chips we have created multiple variations of naive algorithm. These algorithms are labeled as follows:

- BL_Naive is the sequential version of the code, as described by Tomasi [2].
- BL_Tiles is an improved version of Bilateral filtering kernel that employs data blocking to leverage the availability of multi-level cache hierarchy in the modern multicore CPUs.
- $BL\_Assembly$ is the version of Bilateral filtering kernel implemented using assembly level code.
- $BL\_A\_Reduction$ is the version of Bilateral filtering that uses assembly language as well as reduction optimizations.
- $BL\_Intrinsic$ version uses SSE intrinsics instead of assembly language.
- $BL\_I\_Reduction$ is the extended version of $BL\_Intrinsic$ that uses the intrinsics that can compute sum of two 128 bit operands in parallel by exploiting the vector array processing.

BL_Naive has a redundant calculation inside the innermost loop which has been moved out of loop for all other versions. The original instruction involving this calculation looks like

$$gaussian\_ph = exp\left(\frac{-pd}{2 * sigma\_ph * sigma\_ph}\right);$$

After moving the redundant calculation out of the loop the same instruction becomes

$$gaussian\_ph = exp(pd * pre\_calc\_sigma\_sq);$$

Our Bilateral filtering code is open-source licensed under GPL and available from $Google\ Code$ repository http://code.google.com/p/blfilter through SVN version management system.

2) Input Image: In order to analyze the behavior of our kernel with varying image sizes, we have executed our kernel with two different images. One that fits into cache easily and the other that does not fit into cache but fits into main memory. Both of the images were converted to single spectrum images represented by single precision numbers for each pixel. The smaller image was 512 pixels wide and 512 pixels high, thus requiring 1MB memory to store it. As shown in Table I all of the multicore systems have cache memories bigger than 1 MB. The second image was 3000 pixels wide and 3000 pixels high and hence requires little more than 34 MB of memory to store it. None of the multicore cache memories were big enough to store this image entirely.

VI. Performance Results

A. Performance of Bilateral filtering kernel on CPU

For all of our experiments the filter radius was kept at 10, subsequently making the filter Window of size 21 x 21. The spatial spread and the photometric spread were also set to 10. The optimization flag $O3$ was used for all experiments in addition to SSE2 for SIMDization. For Core i7 chip compiler flag SSE4.2 was used instead of SSE2. We do not use hyperthreading, i.e., there are no more than one thread per core.

Figure 9 compares different algorithms on all three $Intel$ chips. In this experiment we are comparing the absolute speedup achieved by the respective algorithm compared to BL_Naive algorithm. It is important to note that these results are for micro level parallelism (use of 128 bit special purpose registers) only and do not consider speedup due to high level (OpenMP) parallelization. The parallel versions shown here are compared against the parallel version of BL_Naive algorithm. The speedup for all algorithms is still bounded by the theoretical peak speedup which happens to be four here since we work with 32bit floating point numbers. Interestingly, although we use compiler optimization flags such as -O3 and -xSSE2, i.e., there is automated vectorization taking place in all the algorithms, our hand coded vectorization is superior in many cases.

The speedup shown by BL_Tiles is mostly due to the computation reduction as discussed in section V-B because we do not use special purpose registers in this algorithm. It can be seen from figure 9 that the parallelism due to SIMDization is limited by four which is the theoretical maximum possible speedup due to SSE registers as discussed in section IV-B.

As seen in figure 9(a), the maximum speedup achieved on Nehalem architecture only using SSE optimization is 2.29x for the smaller image and 1.33x for the larger image. Smaller image generally performs better because the time to copy data to SSE registers is less as the entire image is always available in the cache memory. The BL_Naive algorithm outperforms our assembly level implementations but it is slower than our intrinsic based implementations in all cases. The reason for this is that the auto SIMDization by compiler is more efficient than our implementation for Nehalem architecture. Intel Harpertown, which is a relatively older chip, shows an impressive speedup of 3.54x (Figure 9(b)) even though compiler optimizations are still in place.

Another interesting observation is the results on Intel Core i7 architecture shown in Figure 9(c), where we have the absolute speedup of less than 1. This indicates the fact that the automated compiler optimizations on newer chips are superior than on older chips. These experiments were run with $SSE4.2$ compiler flag and hence the compiler is able to exploit SIMDization automatically thus making it difficult to surpass it using assembly level coding.

However, our algorithms that use SSE intrinsics and reduction, i.e. BL_A_Reduction, BL_Intrinsic, and BL_I_Reduction are able to provide better speedups than those achieved by compiler optimizations as shown in Figure 9 and Figure 10.

Figure 10 shows the similar comparison across AMD chips. The results are in tune with those for Intel platform. As shown in Figure 10(a) for Barcelona, which is a relatively older chip, the maximum speedup is 3.24x for BL_I_Reduction algorithm. Similarly, for AMD Shanghai, as shown in Figure 10(b), the maximum speedup achieved by exploiting SIMDization is 3.02x for BL_I_Reduction algorithm. The case of AMD Phenom is shown in Figure 10(c). The auto tuning applied by the compiler outperforms the manual tuning in case of BL_Assembly and BL_A_Reduction. However, BL_Intrinsic and BL_I_Reduction are still able to outperform compiler optimizations. The maximum speedup achieved in this case is 1.30x.

The next set of results demonstrate the total speedup due to OpenMP as well as due to SIMDization. It is interesting to see how speedup due to OpenMP multiplies with the speedup due to SSE optimizations to exhibit super linear speedup.

Figure 11(a) shows the performance of Bilateral filtering kernel on Intel Harpertown. This machine has 8 cores and the maximum combined speedup achieved is 20.91x. The speedup achieved only using only SSE optimization was 2.69x and the speedup achieved by BL_Naive algorithm using 8 cores is 7.73x.

Intel Nehalem chip with 32 cores does not show a super-linear speedup as shown in Figure 11(c). However, the reason behind this is the lack of enough parallelism that can be afforded by the kernel for this image. Both optimizations still multiply their individual speedup to result in the combined speedup. The total combined speedup is 26.46x using 32 cores, the speedup only due to SSE was 1.12x and the speedup for BL_Naive is 23.47x.

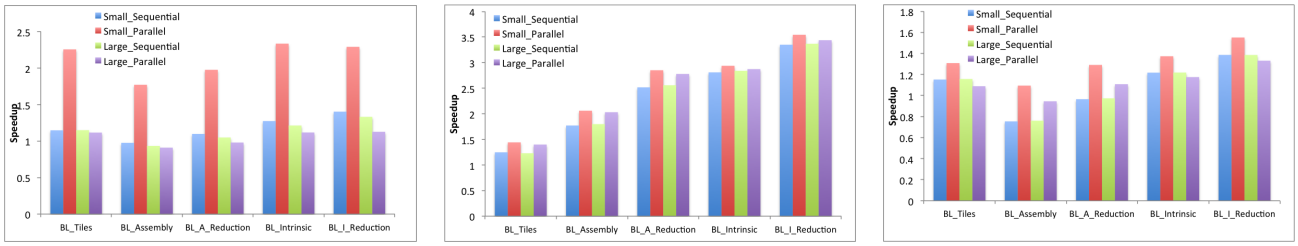
Intel Core i7 chip (Figure 11(b)) also demonstrates the similar behavior. The final speedup of 4.63x, using 4 cores, is a result of speedup due to SSE which is 1.33x and speedup due to OpenMP which is 3.48x.

Figure 12 shows a similar comparison over various AMD chips. Figure 12(a) shows the speedup for Barcelona chip. This chip has 16 cores with a combined maximum speedup being 38.03x. The speedup achieved using SSE optimizations was 2.48x and the speedup for BL_Naive was 15.34x and thus these both realize a speedup of 38.03x using 16 cores.

AMD Shanghai similarly shows an impressive speedup of 20.92x (Figure 12(b)). The interesting case is with AMD Phenom as shown in Figure 12(c). The speedup drops a little as number of cores increases beyond three. The reason behind this is the turbo core technology as discussed in section (V-A4). When there are a maximum of three cores being used the clock frequency scales up to 3.6 GH_z but as soon as more threads are created the clock frequency steps down to 2.7 GH_z.

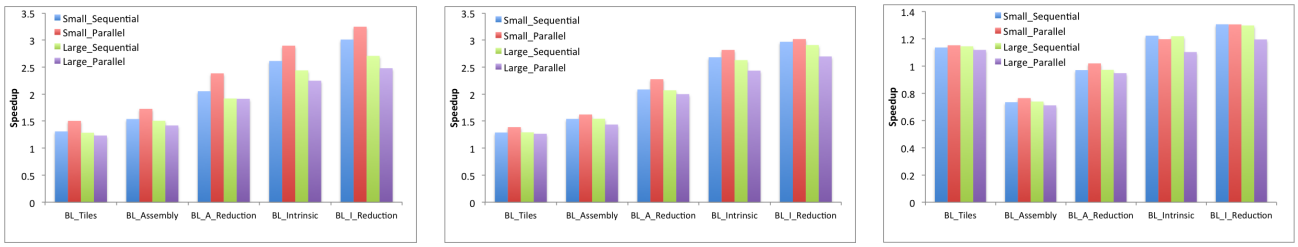
B. Performance of Bilateral filtering kernel on GPU

We used the NVIDIA GTX 280 for benchmarking the performance of our CUDA implementations. Our implementations' variable parameters were the image size, the filter window size, and the grid dimensions (# blocks horizontal * # of blocks vertical concurrently executed) and block dimensions (# of threads horizontal * # of threads vertical in each block).



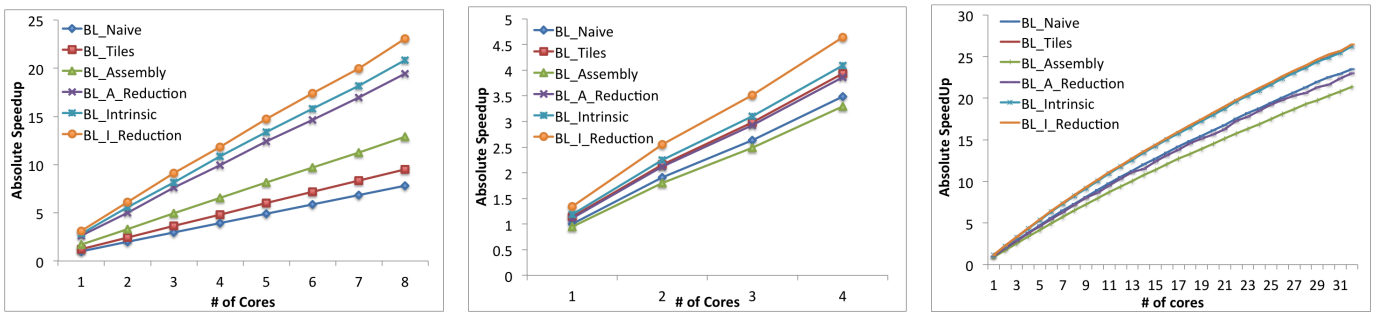
(a) Nehalem-EX Xeon X7650 (b) Harpertown Xeon E5410 (c) Core i7 - 870

Fig. 9. Bilateral filtering kernel on Intel chips. The baseline version is BL_Naive version, auto optimized by compiler with O3 and SSE flags.



(a) Barcelona AMD Opteron 8350 (b) Shanghai AMD Opteron 2376 (c) AMD Phenom II 1045T

Fig. 10. Bilateral filtering kernel on AMD chips. The comparison is with BL_Naive version, auto optimized by compiler with O3 and SSE flags.



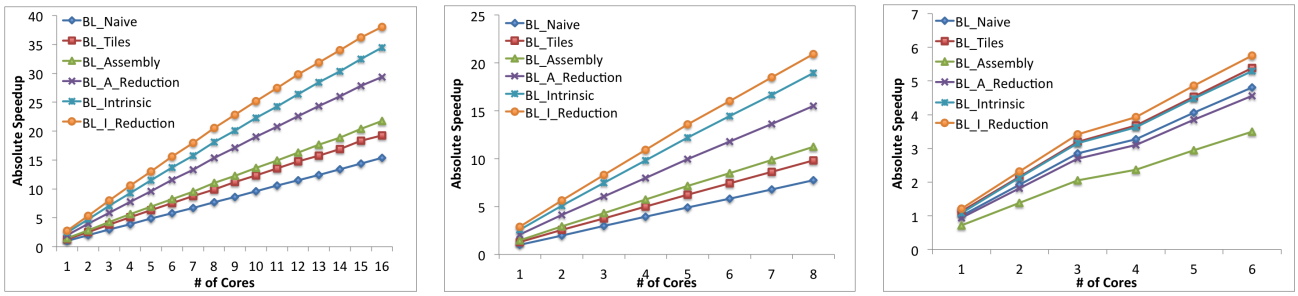
(a) Harpertown Xeon E5410 (b) Core i7 - 870 (c) Nehalem-EX Xeon X7650

Fig. 11. Comparison of combined speedup due to SIMDization and OpenMP parallelization on Intel chips for different algorithms

The tests used two images: a large image with 3000 x 3000 pixels and a small image with 768 x 512 pixels.

On the GPU, our pair-symmetric algorithm outperformed our naive algorithm for every set of tests. For example, Figure 13(a) shows the pair-symmetric algorithm's best timing to be roughly 75% of the naive algorithm's best timing for processing a 3000 x 3000 image using a 21 x 21 pixel filter window. The test that gave the closest difference in timing as shown in Figure 13(c) was processing a 768 x 712 pixel image using a 11 x 11 window size. In this test, the pair-symmetric algorithm's best timing was at a virtual tie with the naive algorithm's best timing at 0.00390 seconds, but upon further testing we found the pair-symmetric algorithm's optimal timing to be 0.00388 seconds when changing the grid dimension to 16 x 16 concurrent blocks, thus slightly outperforming the naive algorithm. Figures 13(a) and 13(c) show exponential rises in execution time for both algorithms as block vertical dimension decreased. Due to runtime complexity of the bilateral filter being the square of the filter width, Figures (b) and (d), which reflect tests that used a 5 pixel filter width, do not exhibit the steep curvature that Figures (a) and (c) exhibit.

While our tests returned a maximum 25% reduction in execution time, our pair-symmetric algorithm has the potential to reach a near 50% reduction in execution time. Furthermore, the benchmarks of our pair-symmetric algorithm reveal minor irregularities in the trajectory of performance derived from execution timings versus respective incremental adjustments of parameter values such as CUDA block and grid dimensions. The main cause of both of these observations is shared memory. To better understand the impact of the additional shared memory transactions that our pair-symmetric algorithm requires, whereas the naive algorithm does not require shared memory for output storage, we implemented a single-threaded CPU-based version of both the naive bilateral filter algorithm and pair-symmetric algorithm so that native-memory (RAM, L1, or L2 cache) could substitute for the GPU's shared memory. The results were remarkably illustrative of our pair-symmetric algorithm's potential. The naive algorithm running on the CPU with a filter width of 10 pixels processed a 3000 x 3000 image in an average of 1206.7 seconds while our pair-symmetric algorithm took 611.8 seconds - a 49.23% reduction in execution time. Moreover, when replacing shared



(a) Barcelona AMD Opteron 8350 (b) Shanghai AMD Opteron 2376 (c) AMD Phenom II 1045T

Fig. 12. Comparison of combined speedup due to SIMDization and OpenMP parallelization on AMD chips for different algorithms.

memory transactions with dummy register transactions in our CUDA implementation of the pair-symmetric algorithm, we observed a consistent 40% reduction in execution time. From these findings, we conclude that the inferior speed of GPU shared memory transactions to register transactions is what limited our pair-symmetric algorithm's performance gains to a maximum of 33%, i.e., achieving timings no less than 75% of the naive algorithm's timings. To further verify this inference, we meticulously stepped through the Parallel Thread Execution (PTX) code - CUDA's quasi-assembly language - of our CUDA implementations to ensure that no unintended optimizations were being made by the compiler when substituting shared memory transactions with dummy register transactions. Not only were no optimizations being made but the PTX code showed that the shared memory transactions required additional load and store instructions, further evidencing the cost of using shared memory.

Another challenge in interpreting our results was porting the concept of "speed-up" to general purpose GPU programming. Since speed-up is necessarily with respect to a reference timing, we used the same reference timing that our CPU optimizations used—that is how we calculated the 235.5x speed-up figure stated in the abstract of this paper—and we also made another reference timing by running our GPU implementations using a single thread. For example, using a single thread operating with filter width of 10 pixels to process a 768 x 512 pixel image, the time taken for the GTX280 to complete this execution was an average of 177.9 seconds for both of our algorithms; using a filter width of 5 pixels, the average execution time was 50.5 seconds for both algorithms. With respect to these reference timings, the speed-up is roughly 20,500x for the 10 pixel filter width and 12,440x for the 5 pixel filter width.

VII. Conclusion and Future Work

Modern day scientific applications demand higher computational capabilities than before and performance gains are no longer driven by clock speed. Chip manufacturers are trying to meet this demand by adding multiple cores on a chip equipped with multiple levels of cache hierarchy and enhanced SIMD extensions. In this paper we have tried to optimize Bilateral filtering kernel using both high level and low level parallelism. Our interest is to develop an efficient parallelization method for modern multicore architectures that can be applied to any spatially invariant and inseparable filter kernel and potentially to any compute-intensive regular numeric application. We have also documented why traditional optimization methods for multicores are not as effective for Bilateral filtering kernel compared to memory intensive kernels. It can be seen from Figure 9, and Figure 10 that a stencil kernel could be optimized by using SSE intrinsics even though the compiler optimizations are in place. Moreover, the reduction methods can provide a decent speedup where applicable. These findings will guide the auto optimizers and auto tuners to include SSE intrinsics as an essential optimization approach.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View From Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep., Dec. 2006.

[2] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Computer Vision, 1998. Sixth International Conference on*, Jan. 1998, pp. 839 –846.

[3] M. Zhang and B. Gunturk, "Multiresolution bilateral filtering for image denoising," *Image Processing, IEEE Transactions on*, vol. 17, no. 12, pp. 2324 –2333, dec. 2008.

[4] E. Eisemann and F. Durand, "Flash photography enhancement via intrinsic relighting," *ACM Trans. Graph.*, vol. 23, pp. 673–678, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015778

[5] S. Bae, S. Paris, and F. Durand, "Two-scale tone management for photographic look," in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006, pp. 637–645. [Online]. Available: http://doi.acm.org/10.1145/1179352.1141935

[6] D. DeCarlo and A. Santella, "Stylization and abstraction of photographs," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '02. New York, NY, USA: ACM, 2002, pp. 769–776. [Online]. Available: http://doi.acm.org/10.1145/566570.566650

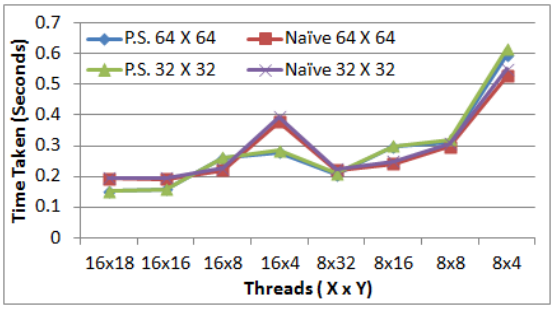
[7] J. Xiao, H. Cheng, H. Sawhney, C. Rao, M. Isnardi, and S. Corporation, "Bilateral filtering-based optical flow estimation with occlusion detection," in *In ECCV, volume I*, 2006, pp. 211–224.

[8] R. Ramanath and W. E. Snyder, "Adaptive demosaicking," *Journal of Electronic Imaging*, vol. 12, no. 4, 2003.

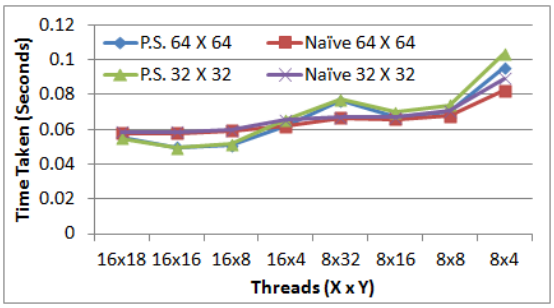
[9] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *Int. J. Comput. Vision*, vol. 81, pp. 24–52, January 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1487450.1487517

[10] T. Pham and L. van Vliet, "Separable bilateral filtering for fast video preprocessing," in *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, 2005, p. 4 pp.

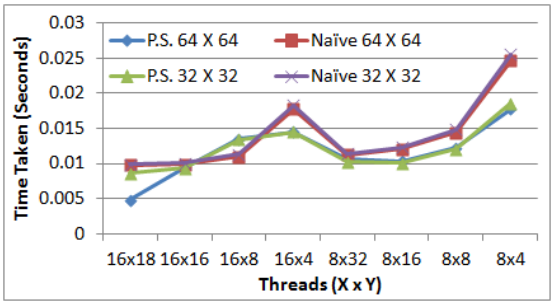
[11] Q. Yang, K.-H. Tan, and N. Ahuja, "Real-time o(1) bilateral filtering," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, june 2009, pp. 557 –564.



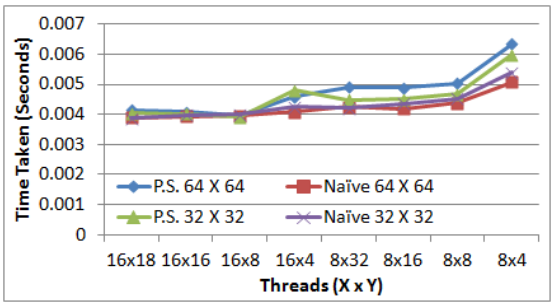
(a) Large Image, Filter Width = 10



(b) Large Image, Filter Width = 5



(c) Smaller Image, Filter Width = 10



(d) Smaller Image, Filter Width = 5

Fig. 13. Performance of Naive and pair-symmetric (P.S.) Bilateral filtering kernel algorithms on GTX 280 NVidia card for both smaller image and larger image.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1413370.1413375

[13] D. Aberdeen and J. Baxter, "General matrix-matrix multiplication using simd features of the piii," in *In European Conference on Parallel Processing*, 2000, pp. 980–983.

[14] A. Muezerie, R. J. Nakashima, R. J. Nakashima, J. Slaets, and G. Travieso, "Matrix calculations with simd floating point instructions on x 86 processors," 2001.

[15] L. D. Stefano, S. Mattoccia, and F. Tombari, "Speeding-up ncc-based template matching using parallel multimedia instructions."