University of North Carolina at Chapel Hill Department of Computer Science
Technical Report TR09-001
ftp://ftp.cs.unc.edu/pub/publications/techreports/09-001.pdf

Performance of 3D Deconvolution Algorithms on Multi-Core and Many-Core Architectures

Cory W. Quammen, David Feng, and Russell M. Taylor II

Department of Computer Science, University of North Carolina at Chapel Hill

Abstract
Deconvolution algorithms are commonly used to remove optical distortion from fluorescence microscopy images. Many such algorithms have been proposed, but those that produce the best image restoration results are iterative. Typically, each iteration involves one or more 3D convolutions, resulting in execution times of tens of seconds to several minutes for common image sizes on single-core computers. Fortunately, most of the constituent computational primitives in deconvolution algorithms are readily parallelized on shared memory architectures. In this paper, we analyze the performance of three deconvolution algorithms implemented on modern multi-core central processing units and on many-core graphics processing units. We discuss the computational primitives in the deconvolution algorithms and their implementations, and compare performance of the two implementations on recent parallel processing architectures.

1. Introduction

Deconvolution is a signal processing technique for removing signal distortion inherent in sensing systems. It is widely used in many fields including seismology [Sil79], astronomy [SPM02], single photon emission computed tomography [FJGC85], and fluorescence microscopy [SN06]. Though we focus on deconvolution algorithms commonly used in restoration of fluorescence microscopy imaging in this paper, many of the principles discussed can be applied to other imaging systems.

All sensing systems introduce distortion artifacts in the data they record. If the distortion introduced is linear and shift-invariant, the recorded signal can be modeled as a convolution of the underlying true signal with the point-spread function (PSF) of the system as follows:

$$o = i \otimes h. \tag{1}$$

Here, i is the true signal, $\otimes$ is the convolution operator, and h is the PSF. Inversion of this equation to yield the true signal i is desirable for data analysis. Various deconvolution algorithms approach inversion of (1) in different ways.

By the convolution theorem, we can rewrite (1) as

$$O = I \cdot H \tag{2}$$

where O, I, and H are the Fourier transforms of o, i, and h, respectively, and $\cdot$ represents point-wise multiplication of complex numbers. A naïve approach to inverting this equation is to directly solve for i:

$$i = \mathcal{F}^{-1}(O/H). \tag{3}$$

Unfortunately, h is band-limited in PSFs from fluorescence microscopes, so H has zero magnitude components. Additionally, high spatial frequencies introduced by noise in a h have small Fourier coefficients. Noise frequencies therefore become dominate in images restored with this linear inversion method [MKCC99].

Many approaches have been proposed that mitigate the problems with direct linear inversion. One approach, the Wiener filter, is a one-pass linear filtering method that reduces the impact of small coefficients in the Fourier transform. The Jansson-van Cittert and Maximum Likelihood Estimation deconvolution algorithms use an iterative approach to minimize the difference between the original distorted image o and the convolution of the estimated true signal i with the PSF.

Iterative deconvolution algorithms typically require 10-20 iterations to produce adequate restorations. On single-core central processing units (CPUs), they can take several minutes to run, introducing delay in the data analysis pipeline. Fortunately, deconvolution algorithms consist of several primitives that are readily parallelized on shared memory architectures. By exploiting multi- and many-core processors widely available today, deconvolution algorithms can be significantly accelerated, potentially reducing the time to scientific discovery.

In this paper, we analyze the performance of the Wiener filter, Jansson-van Cittert, and Maximum LIkelihood Estimation deconvolution algorithms on modern parallel computing hardware. Specifically, we analyze the performance of these algorithms on multi-core central processing units and many-core graphics processing units (GPUs). Our results show that the maximum achievable performance on the 16-way symmetric multi-processing (SMP) CPU architecture we used for testing is consistently better than a single GPU for all the deconvolution algorithms running on typical problem sizes, but performance is best when only 8 of the CPU cores are used.

2. Prior Work

Parallel deconvolution algorithms are available in various commercial software for fluorescence microscopy image analysis, including AutoQuant from MediaCybernetics (Bethesda, MD), Huygens from Scientific Volume Imaging (Hilversum, Netherlands), and DeconLIVE from Intelligent Imaging Innovations (Denver, CO). To our knowledge, no performance analysis has been undertaken on these multi-core CPU implementations. Furthermore, we are unaware of any prior GPU implementation of deconvolution algorithms.

3. Deconvolution Algorithms

In this section, we describe the three deconvolution algorithms we implemented.

3.1. Wiener Filtering

The Wiener filter is a linear filter that optimally minimizes the mean squared-error between the filtered image and the signal received by the imaging system [Wie49]. It assumes an imaging equation

$$o = i \otimes h + n \tag{4}$$

where n is additive Gaussian noise. Given the image distortion model in (4), a filter $\hat{h}$ can be constructed and applied to o to recover an estimated input signal $\hat{i}$, as shown in (5).

$$\hat{i} = o \otimes \hat{h} \tag{5}$$

In the frequency domain,

$$\hat{H} = \frac{H^*}{|H|^2 + (P_n/P_i)} \tag{6}$$

where * is the complex conjugate operator and P_n and P_i are the estimated power spectra of n and i respectively [SN06]. In practice, the P_n/P_i term is replaced by a constant ranging from 0.001 to 0.1 [AHSS89].

The Wiener filter can be computed and applied quickly, but it has several limitations. First, it is the optimal estimator for images with additive Gaussian noise, so it is not strictly appropriate for fluorescence microscopy where noise follows a Poisson distribution. Second, fluorescence images are non-stationary (image statistics change over space), violating one of the assumptions on which the optimality of Wiener filter is based. Finally, the Wiener filter is unable to restore the input signal at frequencies higher than the PSF bandwidth [SN06]. Nevertheless, the Wiener filter is fast and can be used to obtain a less noisy initialization for other deconvolution algorithms.

3.2. Jansson-van Cittert Deconvolution

The Jansson-van Cittert algorithm belongs to the constrained iterative family of deconvolution techniques. These techniques iteratively improve an estimate of i by adding a weighted difference between the original image and the convolution of the estimate with the PSF. Values in estimates of i must be non-negative because they represent the number of photons counted by a sensor. Projection of negative values to zero maintains the non-negativity constraint.

The Jansson-van Cittert algorithm entails repeated execution of several steps:

(a) $o^k = i^k \otimes h$
(b) $i^{k+1} = i^k + \gamma[o - o^k]$
(c) $i^{k+1} = \max(i^{k+1}, 0)$.

The initial guess is the unprocessed microscope image o. Each corrective factor is determined as the difference between o and the convolution of the kth true signal estimate and the PSF, o^k. If o^k is blurrier than o, the image will be sharpened by subtracting from the current guess image intensities proportional to the difference. As o^k converges to o, the magnitude of the corrective factor diminishes. The parameter γ is set to $1 - (o^k - A)^2/A^2$ where A is the maximum intensity value of o divided by 2; γ restricts intensities to the range $[0, 2A]$ [Aga84]. The algorithm can be run for a preset number of iterations or until some convergence criterion is met [SN06]. Empirical experiments with a synthetic example image and PSF show that between 10-20 iterations is usually sufficient for obtaining a restored image that closely matches a known input image.

This algorithm tends to emphasize contrast as well as noise in images. The noise enhancement is usually mitigated by applying a smoothing filter such as a Gaussian or Wiener filter before starting the algorithm, and smoothing intermediate results every few iterations. Smoothing the estimate at intermediate steps works against the goal of sharpening the image, but is often a necessary regularization step [Sib05].

3.3. Maximum Likelihood Estimation Deconvolution

Maximum Likelihood Estimation (MLE) deconvolution algorithms are designed to restore an image such that the likelihood that it could give rise to the observed image is maximized. The algorithms are derived from an image formation model that includes noise following a given probability distribution. In the case of fluorescence microscopy, the dominating noise source comes from the stochastic nature of photon emission and therefore follows a Poisson distribution.

The Richardson-Lucy MLE algorithm is commonly used in deconvolution of optical images. One iteration of the algorithm involves the following steps:

(a) $a = i^k \otimes h$
(b) $b = o/a$
(c) $i^{k+1} = ci^k(b \otimes h)$

where c is a normalization constant and multiplication and division are performed point-wise [Ric72, Luc74].

This deconvolution method does not require a projection step to maintain the non-negativity constraint because it involves only multiplication and division of positive numbers. Noise may be enhanced with this algorithm, but it can be reduced through Gaussian smoothing of the initial image or by terminating the algorithm before convergence [Sib05]. MLE deconvolution tends to produce better results than Jansson-van Cittert deconvolution, but each iteration requires two 3D convolutions instead of one.

4. Parallel Deconvolution Implementations

The deconvolution algorithms we implemented consist of three common computational primitive operations:

Mapping Various operations can be composed into a computational kernel that is mapped to one or more arrays of numbers yielding one or more result arrays. Pointwise operations in the deconvolution algorithms are implemented as mapping primitives.

Reduction Reduction applies an associative operator to an array of numbers yielding a single number. Normalization and energy preservation constants in the MLE deconvolution update equations are obtained through a reduction with the addition operation.

Fast Fourier Transform (FFT) The FFT algorithm computes the discrete Fourier transform of a signal. The FFT algorithm enables efficient computation of convolutions via multiplication in the Fourier domain.

In our implementation, these algorithms operate on image data stored as linear arrays of 32-bit floating point values. Arrays of complex values are stored with real and imaginary components interleaved. This data arrangement is required by the FFT libraries we used and is a convenient layout for other algorithms.

4.1. Multi-core CPU Implementation

We implemented the three deconvolution algorithms described in Section 3 on multi-core CPUs using C++ and OpenMP for parallelization [DM98]. The programs were compiled with the Intel C++ Compiler 11.0 which can automatically identify data-level parallelism and insert vector instructions. Streaming kernels were implemented as OpenMP `parallel for` regions using the default static scheduling. Summation reductions were implemented using the OpenMP `reduction` clause. We used the FFTW library for computing 3D discrete real-to-complex and complex-to-real Fourier transforms [FJ05]. FFTW was compiled with the `-with-openmp` option using the Intel compiler. FFTW uses the concept of *plans* to determine FFT algorithm parameters that reduce run time. We use the default `FFTW_ESTIMATE` option to avoid potentially costly initialization.

4.2. Many-core GPU Implementation

We used NVIDIA's CUDA programming environment to implement the three deconvolution algorithms on NVIDIA GPUs. Problem decomposition in the CUDA programming model involves several layers of granularity. At the coarsest level are individual tasks that correspond to steps in the algorithm. At a finer level are subproblems within a task that can be solved independently of each other but using the same set of instructions. At the finest level are pieces of the independent subproblems involving a small number of cooperative data-parallel threads that can communicate through a shared memory cache [NVI08a].

This model of problem decomposition is realized in CUDA through *grids*, *blocks*, and *threads*. A grid specifies a task that is broken down into a user-defined number of independent blocks. A block specifies the user-defined number of cooperative threads that can communicate through shared memory and which can synchronize with a thread barrier instruction. To split up work in the parallel task, each thread is assigned an index within the block and each block is assigned an index within the grid. Each thread has access to these indices and can use them to determine which part of the task to solve. Indices are 3D for potential convenience when accessing 3D structured data, but they can be treated as 1D or 2D indices by ignoring dimensions.

Implementing the deconvolution computational primitives in CUDA for best performance is more complex than the implementation on multi-core CPUs. In this section, we

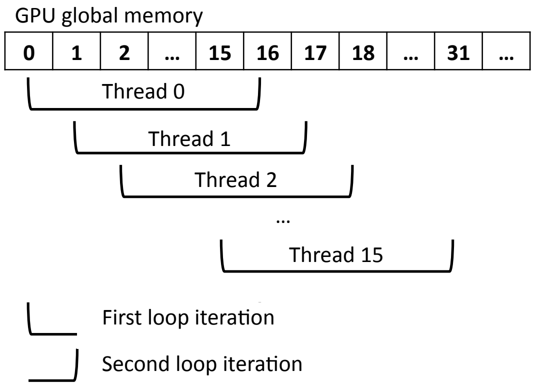


Figure 1: *Memory access pattern in a CUDA mapping primitive run with 16 total threads. Each thread iterates over a maximum of $\lfloor N/numThreads \rfloor + 1$ array elements, accessing the array with stride numThreads. This pattern assures that reads of 32-bit float-point numbers are coalesced because threads with contiguous indices access continuous memory addresses in order.*

describe the implementation of the computing primitives and discuss how the implementation choices best use the GPU hardware.

4.2.1. Mapping

Mapping a kernel onto a buffer containing 3D image data can be implemented in several ways using the CUDA decompositional model. One way is to specify 3D grid and block dimensions in such a way that the number of threads matches the number of voxels, and the number of threads in each dimension matches the dimensions of the image. Each thread then applies the kernel to the data for which it is responsible.

There are two problems with this approach. One problem is that the dimensions of the 3D data may not be even multiples of the grid and block dimensions. The grid and block dimensions could be factorized so that their products match the dimensions of the image, but doing so may result in grid and block sizes that do not make best use of the GPU resources. Alternatively, the grid size could be expanded so that it is slightly larger than the image. A conditional would then be necessary to handle the cases when the 3D thread index lies outside the image bounds. If the index is not valid, that thread will not do any work, reducing efficiency.

A more important problem in terms of performance occurs when the major dimension of the image (the x-dimension by our convention) is not a multiple of the size of an aligned memory segment on the hardware. Individual thread requests for memory can be coalesced into a single memory transaction when the pattern of memory accesses satisfies certain criteria. Coalesced memory accesses greatly increase memory bandwidth and are necessary for achieving maximal performance. For the most recent NVIDIA GPUs (those with CUDA Compute Capability 1.2), a separate memory transaction is issued for every memory segment accessed in the request. On older NVIDIA GPUs (Compute Capability 1.0 and 1.1), memory requests that do not fall with the same memory segment or which do not follow strict ordering requirements result in a separate transaction for every request, significantly reducing memory bandwidth. Thus, if the major dimension of the image is not a multiple of the size of a memory segment, many additional memory transactions will occur, hurting algorithm performance. The image data could be padded to a multiple of the segment size, but doing so wastes memory and can complicate indexing calculations in other algorithms that operate on the data.

A better approach is to treat the image data as a 1D buffer. To iterate over every voxel, we choose a fixed grid and block size appropriate for the number of streaming multiprocessors (SMs) on the GPU and independent of the image size. Each thread is responsible for processing a share of the voxels in the image. The kernel executed by each thread contains a `for` loop that iterates over the thread's share of voxels. Because memory accesses are coalesced when sequentially-indexed threads access sequentially-indexed memory locations, we interleave thread memory accesses to the image buffer. Specifically, the stride with which each thread accesses the image should equal the total number of threads in the grid to obtain this memory access pattern and thereby achieve maximum memory bandwidth [NVI09]. Figure 1 depicts this memory access pattern. For the best performance on current GPUs, the number of threads should be a multiple of 16 [NVI08a]. We have found that on an NVIDIA Quadro FX 5600, a grid of 128 blocks, each consisting of 64 threads, provides the best performance for the mapping operations we evaluated.

For complex-valued images where the real and imaginary components are interleaved, memory accesses can be coalesced when each complex number is requested as a single aligned 64-bit structure consisting of two 32-bit floats. In our implementation, we typecast a pointer to the `float` array containing the complex values to a `float2` data type pointer defined by the CUDA library [NVI08a]. This data type satisfies the 64-bit alignment criteria.

4.2.2. Reduction

In reduction primitives, we efficiently read image data following the same pattern as in the mapping primitive. Instead of computing a result and writing it back to memory, however, each thread collects a partial reduction of the elements it accesses and stores the results in the shared memory cache. This cache is as fast as registers when groups of threads access the memory in a way that produces no bank conflicts [NVI08a]. For reductions on real-valued images, the shared memory allocation is large enough to hold one 32-bit floating-point value per thread in the block.

After all partial results have been read into shared memory, we iteratively apply the reduction operation across threads. In each iteration, half of the values remaining in shared memory are combined with the other half through the reduction operator using a set of active threads. The indices of the active threads relative to the block always start at 0. We split the shared memory logically into upper and lower halves. Each active thread with index tid applies the reduction operator to elements with index $tid + lowerIndex$ and $tid + upperIndex$ where $lowerIndex$ is the starting index of the lower half (always 0) and $upperIndex$ is the starting index of the upper half. All the threads synchronize to ensure the shared memory contains the updated values, and the upper half of active threads are deactivated. The final reduction result for the block is stored in the first shared memory location [NVI09]. This result is written into a special global memory buffer allocated to hold the results of each block reduction. Because the number of blocks is relatively small, we read this buffer back into CPU memory and perform the final reduction pass serially on the CPU rather than performing another pass on the GPU. Figure 2 depicts data flow in reduction operations.

4.2.3. FFT

We use the CUFFT library provided by NVIDIA [NVI08b] to compute the discrete Fourier transform of a 3D image. Recent work has shown up to a three-fold performance improvement over CUFFT [GLD*08, NOEM08], but these libraries were not available for testing.

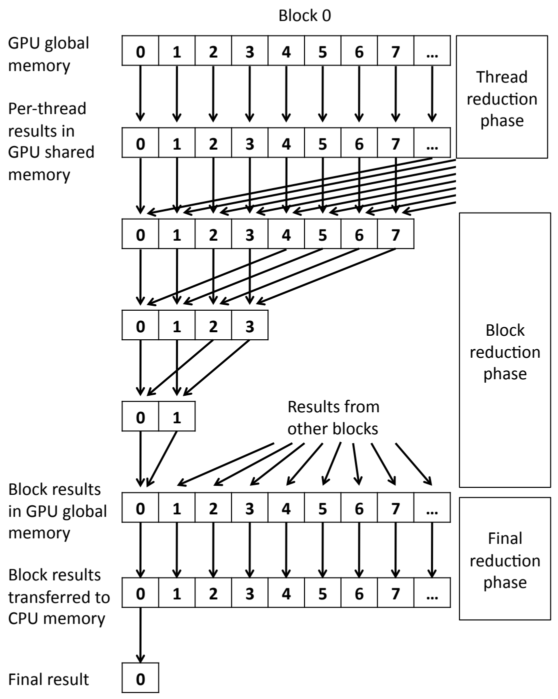

Block 0
GPU global memory
Per-thread results in GPU shared memory
Thread reduction phase
Block reduction phase
Results from other blocks
Block results in GPU global memory
Block results transferred to CPU memory
Final reduction phase
Final result

Figure 2: *Efficient reduction on the GPU. Per-thread partial reduction results are stored in the shared memory cache. Per-block partial reduction results are then computed and written to a specially-allocated global memory buffer on the GPU. A CPU core performs the final reduction after reading the per-block results from the GPU.*

5. Performance Analysis

In this section we compare the performance of the implemented deconvolution algorithms on two parallel computing architectures. One architecture consisted of a set of multi-core CPUs connected by a high-bandwidth interconnect. The other resource was a single modern GPU. We present results in three parts, first giving scalability results, then a breakdown of the performance of computational primitives in each algorithm, followed by a comparison of CPU and GPU performance.

5.1. Experimental Setup

We ran the algorithm CPU implementations on a Sun Fire X4600 M2 16-way symmetric multi-processing (SMP) node with 32 GB RAM on the Biomedical Analysis and Simulation Supercomputer at the University of North Carolina at Chapel Hill. The node hardware consists of 8 dual-core 2.8 GHz AMD Opteron processors connected by a HyperTransport network of direct processor-to-processor connections. The GPU implementations were evaluated on a single NVIDIA Quadro FX 5600 card in an NVIDIA QuadroPlex Model IV with sixteen 8-way streaming multiprocessors and 1.5GB RAM connected via 16× PCI Express to a node consisting of 4 dual-core 2.6 GHz Opteron processors with 8 GB RAM. We used version 2.0 of the CUDA Toolkit. All nodes ran RedHat Enterprise Linux 5.3.

All the times we report are averaged over ten runs. GPU performance numbers include the time taken for data transfer to and from the GPU. It is important to note that the input image and PSF image are padded to handle the cyclic nature of convolution via multiplication in the Fourier domain. Therefore the dimensions of an image processed by the deconvolution algorithms is actually the sum of the dimensions of the input image and PSF. The times reported here exclude the time required for padding the input image and PSF image and clipping the final image; these operations are common to both implementations. In all tests, the PSF had dimensions $64 \times 64 \times 32$.

5.2. SMP Scalability

To test the scalability of each algorithm on multi-core systems, we varied the number of cores from 1 to 16. Figure

3 shows scalability of the three algorithms applied to four different problem sizes. We use percentage of ideal linear speedup as the measure of scalability. All three algorithms exhibit a sharp drop-off in scalability beyond 8 cores. The drop-off is present in timings of FFTW's transform algorithms, but not in the mapping or reduction primitives. It is not clear whether the source of this drop-off is caused by resource contention in the hardware, by a software limitation, or by the operating system. In any case, the best wall clock time was achieved when all three algorithms were run with 8 cores (see Figure 4). All three algorithms scaled well for the largest image size ($512 \times 512 \times 64$), reaching from 82-88% of the ideal linear speedup.

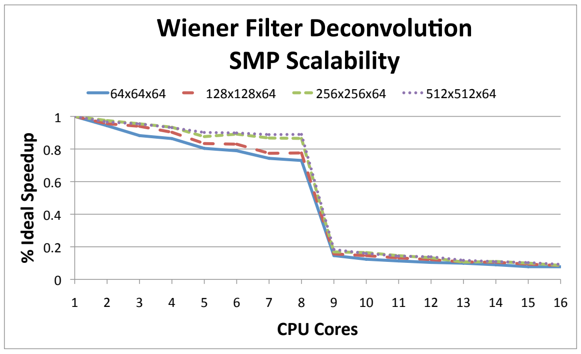
5.3. Performance of Computational Primitives

We measured the execution time of each computational primitive on both parallel processing architectures when deconvolving a $512 \times 512 \times 512$ image with a $64 \times 64 \times 32$ PSF. Tests of the SMP implementations were run with 8 cores. As shown in Table 1, real-to-complex forward (R2C) and complex-to-real inverse (C2R) FFT algorithms clearly dominate the running time in each algorithm implementation. The FFT algorithms take 91.4%, 80.6%, and 83.0% of the total execution time for the SMP implementations of the Wiener filter, Jansson-van Cittert algorithm, and MLE method, respectively, and 66.5%, 91.7%, and 93.8% of the GPU implementation execution times. Efforts to improve the performance of FFT algorithms is therefore the most effective way to further accelerate the deconvolution algorithms.

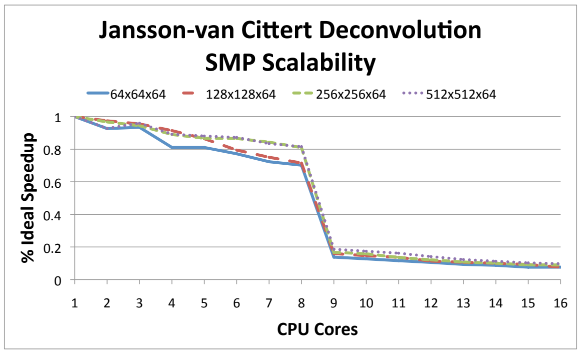
Mapping primitives in the SMP implementations make up a larger percentage of the running time than on the GPU implementations. Considering that these algorithms perform relatively little floating-point computation per memory access, this result is not surprising. GPUs are engineered to support higher memory bandwidth than CPUs (76.8 GB/s peak for the NVIDIA Quadro FX 5600 versus 42.8 GB/s peak for four dual-core processors in the Sun Fire X4600 M2 server).

Reduction operations in the MLE algorithm are 1.4 times faster on the GPU than on the SMP implementation. The GPU reduction operation runs more than twice as long as a mapping operation on the GPU that applies a scale factor to an array. This is somewhat surprising as the reduction operation reads the same amount of data from global memory as the mapping algorithm, but writes significantly less data. We attribute the slowdown to the logarithmic number of reduction steps in the algorithm as well as overhead from data transfer to CPU memory for calculating the final reduction value.

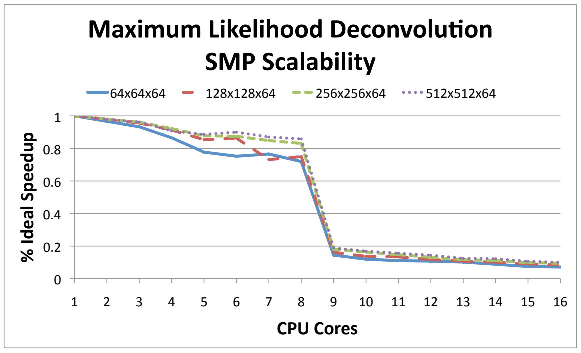
The GPU implementations suffer some overhead when transferring data from the CPU memory to the GPU. For the Wiener filter, the overhead is high (30.6%). In the iterative algorithms, however, the overhead is amortized over many



(a)



(b)



(c)

Figure 3: *Scaling behavior of the three deconvolution algorithms on multi-core CPUs. Performance is represented as the percentage of ideal linear speedup.*

iterations where no data transfer is required and accounts for less than 5% of the execution time.

5.4. Comparison of CPU and GPU

In our performance measurements, we found that 8 cores on the 16-way SMP node consistently outperforms a single GPU. The SMP implementation is 1.7-2.8$\times$ faster than the GPU on the Wiener filter. For the iterative algorithms, the

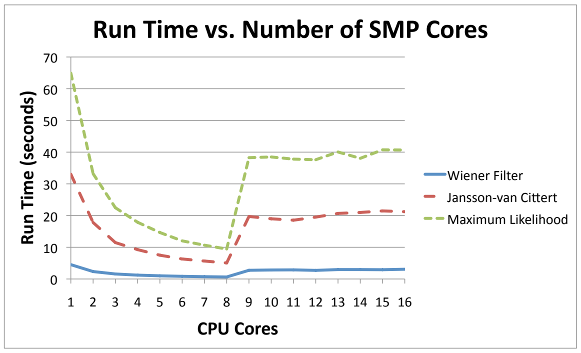


Figure 4: *Run time of three deconvolution algorithms versus number of cores for a* $512 \times 512 \times 64$ *image. The Jansson-van Cittert and MLE algorithms were each run with 10 iterations. Scalability peaks at 8 cores. Notably, the performance of 9-16 cores is worse than 2 cores for each algorithm.*

GPU is somewhat more competitive. The SMP implementation of the Jansson-van Cittert algorithm is 1.3-2.3× faster, and the Maximum Likelihood algorithm is 1.3-2.0× faster. For the large problems typically encountered in fluorescence microscopy, the SMP Wiener filter implementation is 1.9× faster while both iterative deconvolution algorithms are 1.3× faster. Figure 5 shows the speedup of 8 cores over one GPU for all three algorithms on a variety of problem sizes.

Another way to compare the CPU and GPU implementations is to determine how many CPU cores are needed to outperform a single GPU. For the Wiener filter, 3 cores are faster than the GPU for all problem sizes. For the Jansson-van Cittert and MLE algorithms, it takes 6 cores to beat the GPU on a large problem size ($512 \times 512 \times 64$). For smaller problem sizes, 4 or 5 cores outperform the GPU.

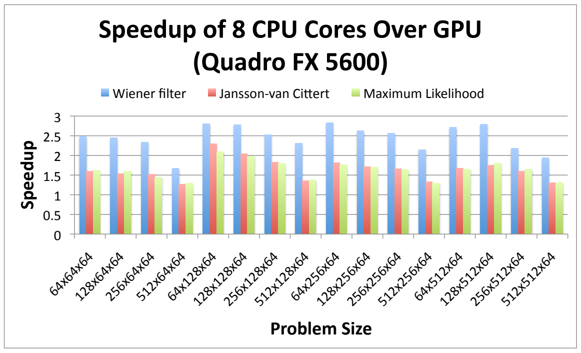


Figure 5: *Performance comparison of 8 CPU cores versus a single GPU. The 8 CPU cores are almost always twice as fast for the Wiener filter and are consistently faster than the single GPU.*

Computational Primitive	Time (% total) 8 CPU cores	Time (% total) GPU
Wiener Filter		
Total	636	1232
FFT R2C	387 (60.8%)	531 (43.0%)
FFT C2R	195 (30.6%)	290 (23.5%)
Map (kernel)	25 (4.0%)	6 (0.5%)
Copy to GPU	-	157 (12.7%)
Copy from GPU	-	221 (17.9%)
Jansson-van Cittert		
Total	4960	6171
FFT R2C	2126 (42.9%)	2842 (46.0%)
FFT C2R	1868 (37.7%)	2821 (45.7%)
Map (modulate)	256 (5.2%)	61 (1.0%)
Map (update kernel)	532 (10.7%)	105 (1.7%)
Copy to GPU	-	138 (2.2%)
Copy from GPU	-	219 (3.6%)
Maximum Likelihood Estimation		
Total	9325	11702
FFT R2C	4038 (43.3%)	5367 (45.9%)
FFT C2R	3706 (39.7%)	5600 (47.9%)
Map (modulate)	492 (5.3%)	122 (1.0%)
Map (division)	195 (2.1%)	53 (0.5%)
Map (scale)	119 (1.3%)	45 (0.4%)
Reduce (sum)	154 (1.7%)	107 (0.9%)
Copy to GPU	-	142 (1.2%)
Copy from GPU	-	203 (1.7%)

Table 1: *Time (in milliseconds) required for all invocations of each computational primitive on a* $512 \times 512 \times 64$ *image with a* $64 \times 64 \times 32$ *kernel. The percentage of total execution time is given in parentheses. FFT forward and inverse transforms clearly dominate the running time of all algorithm implementations.*

6. Conclusions and Future Work

We have analyzed the performance of 3D deconvolution algorithms on modern parallel computing architectures. The source code for these algorithms has been released in the Clarity Deconvolution Library, available at http://cismm.cs.unc.edu/downloads. The algorithms consist of parallel primitives capable of efficient operation on many-core architectures. On the architectures we tested, the best performance is available on 8 CPU cores, but a single GPU performs competitively with 4 or 5 CPU cores.

In the future, we would like to evaluate the GPU implementations with a faster FFT library, such as those developed by [GLD*08] or [NOEM08]. Furthermore, we would like to test the GPU implementation on more recent graphics hardware that features nearly twice as many cores as the Quadro FX 5600. Finally, We would also like to investigate whether parallelizing the algorithms across distributed memory nodes breaks the apparent scalability limitations on the SMP hardware we tested. Such a parallelization effort would also enable the use of several GPUs.

Acknowledgements

This work was supported in part by NIH grant P41 EB002025. The BASS supercomputer is funded through the NIH National Center for Research Resources, award number NIH 1S10RR023069-01.

References

[Aga84] AGARD D. A.: Optical sectioning microscopy: cellular architecture in three dimensions. *Annual Review of Biophysics and Bioengineering 13* (1984), 191–219.

[AHSS89] AGARD D. A., HIRAOKA Y., SHAW P., SEDAT J.: Fluorescence microscopy in three dimensions. *Methods in Cell Biology 30* (1989), 353–377.

[DM98] DAGUM L., MENON R.: OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering 5*, 1 (Jan-Mar 1998), 46–55.

[FJ05] FRIGO M., JOHNSON S.: The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231.

[FJGC85] FLOYD CAREY E. J., JASZCZAK R. J., GREER K. L., COLEMAN R. E.: Deconvolution of Compton Scatter in SPECT. *The Journal of Nuclear Medicine 26*, 4 (1985), 403–408.

[GLD*08] GOVINDARAJU N. K., LLOYD B., DOTSENKO Y., SMITH B., MANFERDELLI J.: High performance discrete Fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), IEEE Press, pp. 1–12.

[Luc74] LUCY L. B.: An iterative technique for the rectification of observed distributions. *Astronomical Journal 79* (June 1974), 745–+.

[MKCC99] MCNALLY J. G., KARPOVA T., COOPER J., CONCHELLO J. A.: Three-dimensional imaging by deconvolution microscopy. *Methods 19*, 3 (November 1999), 373–385.

[NOEM08] NUKADA A., OGATA Y., ENDO T., MATSUOKA S.: Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–11.

[NVI08a] NVIDIA: Compute Unified Device Architecture programming guide, June 2008.

[NVI08b] NVIDIA: CUFFT library user manual, April 2008.

[NVI09] NVIDIA: CUDA SDK Code Samples, Feb. 2009.

[Ric72] RICHARDSON W. H.: Bayesian-based iterative method of image restoration. *Journal of the Optical Society of America 62* (1972), 55–59.

[Sib05] SIBARITA J.-B.: *Deconvolution microscopy*, vol. 95 of *Advanced Biochemical Engineering/Biotechnology*. Springer-Verlag, Berlin/Heidelberg, 2005, pp. 201–243.

[Sil79] SILVIA M. T.: *Deconvolution of geophysical time series in the exploration for oil and gas*. Elsevier Scientific Publishers Co., New York, 1979.

[SN06] SARDER P., NEHORAI A.: Deconvolution methods for 3-D fluorescence microscopy images. *IEEE Signal Processing Magazine 23*, 3 (May 2006), 32–45.

[SPM02] STARCK J. L., PANTIN. E., MURTAGH F.: Deconvolution in astronomy: a review. *Publications of the Astronomical Society of the Pacific 114* (2002), 1051–1069.

[Wie49] WIENER N.: *Extrapolation, Interpolation, and Smoothing of Stationary Time Series*. Wiley, New York, 1949.