National University of Ireland, Galway

Doctoral Thesis

# IMPAIR: Massively Parallel Regularised Richardson-Lucy Image Deconvolution on Heterogeneous Hardware

*Author:*
Michael Sherry

*Supervisor:*
Prof. Andy Shearer

*A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy*

*in the*

Centre for Astronomy
School of Physics

October 2017

# Contents

# List of Figures

# List of Tables

# *Abstract*

This thesis investigates the comparative performance of multicore CPU and general purpose GPU on a commodity desktop computer. To investigate this, an image deconvolution software package (IMPAIR) was updated from its original cluster-computing design to support both of these parallel architectures. The IMPAIR software was chosen for this investigation due to the high memory and computational demands of the image restoration algorithms it implements, coupled with these algorithms' natural amenity to highly parallelised solutions.

IMPAIR performs the image deconvolution operation by parallelising either the unregularised Richardson Lucy algorithm (RL) or a wavelet regularised variant of Richardson Lucy (WRL), which carries a significantly higher computational cost but is more robust to the presence of high levels of noise in the algorithm's input image. In order to support this WRL algorithm, general use wavelet shrinking libraries were developed for both the GPU and CPU, where a $\times 2$ –$\times 3$ speedup of the GPU wavelet shrinking to the CPU wavelet shrinking was achieved.

In total, eight parallelisation strategies for the IMPAIR deconvolution algorithms have been implemented and their runtime performance on a commodity desktop hardware is presented. Of the strategies presented, the "Topdown" multicore CPU strategy and the "Streaming" GPU strategy achieve similar runtimes, but the reduced memory footprint of the GPU Streaming strategy permits scaling up to image data over ten times the maximum capacity of the multicore CPU Topdown strategy, for both the regularised and unregularised Richardson Lucy algorithms.

# *Acknowledgements*

I am heavily in debt to the following people for the aid, advice, and infinite patience they made available to me over these last six years:

Firstly, my supervisor, Professor Shearer. He has pulled me back from starting down a winding path to nowhere and pointed me back in the direction I should be going more times than I can count. Thank you.

Secondly, Professor David Gregg and Doctor Ray Butler, who played the exhausting roles of external and internal examiner with a methodical patience and understanding I am in awe of. Thank you both.

Thirdly, the staff and postgrads of the Centre for Astronomy, NUI Galway, in particular for the inviting welcome offered to me by them in my first year, when I arrived a stranger to the department. Thank you all.

To those people with whom I have shared living space, be it an office, or a house, or corner of a sitting room. I've wrecked all their heads with my ramblings, and none have ever held it against me. To Catherine, Simon, Sean, Susan, Gillian, Diarmaid, Paul, Navtej, Ronan, Paul, Lisa, Eoin, Mags, Gordon, Eamon, Kieran, Nicola, Aisling, Chris, Hazem, Camile, Siobhán, James, Barry, Ben, Donal, John, Jimmy, Suzi, and many, many, many others: Thanks a million.

Finally, my parents, Mary and Tom, and my siblings, Peter and Clara. Thanks for bearing with me through this.

PS. Emergency thank-yous to Lisa, Laura, and Eoin for dropping everything to help me out when time was of the essence. Thanks!

# Declaration of Authorship

I, the Candidate, certify that the Thesis is all my own work and that I have not obtained a degree in this University or elsewhere on the basis of any of this work.

Signed: _____

Michael Sherry

Date: _____

# Chapter 1

# Heterogeneous Computing on the Modern Desktop Computer

## 1.1 Introduction

A modern Desktop PC is a vastly heterogeneous beast. The combination of CPU model and manufacturer, Operating System API, RAM caching properties and strategies, virtualisation environment, persistent storage facilities, and special purpose hardware on both the mother board and expansion ports varies greatly between machines. While writing software that will run on all of the various configurations that exist is possible, writing software that will take advantage of any relevant optimisations available requires direct intervention and design decisions on the part of the developer. Unfortunately there is both the danger and expectation that since these decisions require insight into the underlying working of the machines, any optimisations will result in source code that is neither clearly written from an application domain point of view, nor easily maintained or ported to other systems.

In particular, the wide variation in multicore CPU architectures currently in circulation, including dual core, quad core and hyperthreaded 8 core CPUs, as well as the older generation of single core CPUs, has made it increasingly difficult to write software that will run evenly across this variable landscape of hardware [1]. The move from serial, single core CPUs to multicore CPUs and the introduction of programmable GPU cards in the standard Desktop PC initially left popular compilers, their programming languages, and runtime environments unable to take advantage of hardware at their disposal, unlike the previous transition from scalar to superscalar CPUs for desktop computing, which provided a transparent boost in performance [2]. Though the situation has gradually improved with the introduction of cross platform library suites compatible and tuned to

wider ranges of these architectures [3][4][5], taking on the responsibility of managing the hardware resources manually or limiting the portability of the software via a non-standard framework is not an attractive choice for a software developer.

However, an exciting consequence of this fundamental shift in the make up of the common Desktop PC is a further closing of the gap between High Performance Computing platforms and the ubiquitous Desktop PC platform. Already this has allowed for a more seamless migration of software in both directions between these two domains, in a much more straightforward manner than previous generations of hardware were able to permit. Since the Desktop PC can be considered a physical miniaturisation of a computational cluster [6], Desktop software (provided it correctly exploits this miniaturised HPC architecture) can now scale up satisfactorily in terms of both throughput and latency when deployed on a modern HPC node or cluster without any source code modifications. Similarly, HPC software can be expected to scale downward gracefully to the Desktop PC.

## 1.2 Concurrency and Parallelism

Early advances in parallelism found success in the automatic parallelisation of serial software to better make use of the ability of the internal functional units of a CPU to operate in parallel. This automatic parallelisation of CPU instructions was achieved at the compiler level, where the compiler statically scheduled multiple functional units to operate simultaneously; and at the CPU level, where the CPU dynamically scheduled multiple functional units to operate simultaneously in the hope that this would better parallelise software with more non-deterministic control flow. As a data-processing application spends a disproportionately large amount of its runtime within looping structures, emphasis was put on parallelising the sequences of instructions if they are within a loop, and on simultaneously executing the instructions from successive iterations of a loop if the iterations are found to be safely re-orderable [7]. By the end of the $20^{th}$ century, the limits of automatic parallelisation at the instruction-level had been reached [2].

Significant research into the behaviour of parallel computers began in the 1960s with the introduction of the overlapped fetch-decode-exec cycle CPU design [8] and array-processor machines [9][10], and during this period a simple model of parallel computing now known as the Flynn taxonomy [11] was put forward. Unlike the more theoretical models of parallel computing (such as Hoare's Communicating-Sequential-Processes [12], Hewitt's Actor Model [13], and numerous others [14][15]), which aim to cover all manner of forms of concurrent and distributed algorithms, this model was developed only to

address the implementation of parallel execution in a single centralised processor, by covering the range of all possible configurations of the externally presented interfaces.

## 1.3 The Flynn Taxonomy

The Flynn taxonomy [11] divides all forms of computational systems into four basic categories. These describe the interfaces, rather than the implementations, of service-like systems. Consequently, the taxonomy can be applied generally to any computational system—whether purely hardware, purely software, or any combination of the two—by treating the system as a black-box virtual machine, and identifying its place in the taxonomy based on the interface the virtual machine provides. This allows the taxonomy to sensibly categorise systems that are physically parallel, logically parallel (but physically serial), and systems that exhibit various combinations of both of these traits without having to take these details into account. Similarly, distinctions between shared and distributed memory address space systems, or hierarchical or otherwise non-uniform memory architectures, do not require special consideration under this taxonomy.

The four categories of the Flynn taxonomy are: Single Instruction Stream, Single Data Stream (SISD); Single Instruction Stream, Multiple Data Streams (SIMD); Multiple Instruction Streams, Single Data Stream (MISD); Multiple Instruction Streams, Multiple Data Streams (MIMD).

**Single Instruction Stream, Single Data Stream (SISD)**
The SISD interface describes the classic Von Neumann architecture (where the incoming instruction stream and data stream are logically separate, but travel from the same external store over the same transfer bus), and the Harvard architecture (where the incoming instruction stream and data stream are physically separated, traveling from distinct external stores over distinct buses). Processors with separate $L_1$ instruction-cache and data-cache spaces, but combined $L_2$–$L_L$ caches similarly present an SISD.

A system that exposes an SISD interface can be trivially built on an underlying SIMD system by only utilising one of the SIMD system's data streams, and its only instruction stream. This is the standard approach in GPU, multicore, and cluster computing to perform the inherently serial operations that result in the bottlenecks for Amdahl's law [16]. In situations such as this, there is no performance benefit from building an SISD system on top of an SIMD system.

Examples Systems:

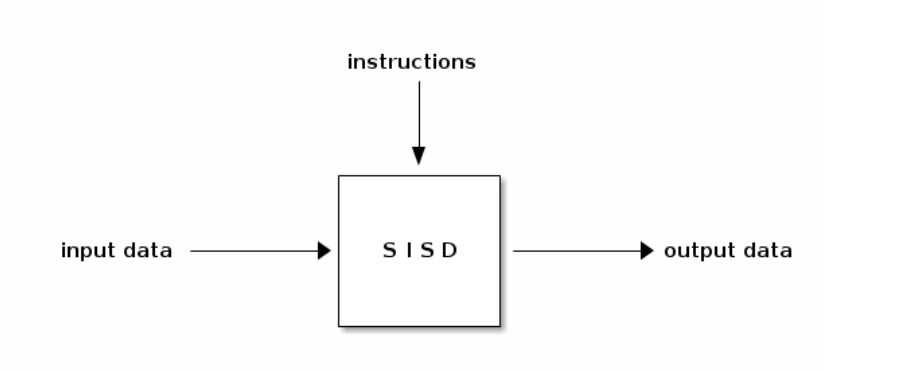



FIGURE 1.1

- Scalar Processors
- Pipelined Scalar Processors [8]

  (present an SISD interface, though execution is parallelised by overlapping)
- Superscalar Processors [17]

  (present an SISD interface, though execution is parallelised by overlapping and dependency tracking)

SISD over MISD is far more common, as this configuration describes the pipelined or overlapped fetch-exec cycle that has been available in CPUs for decades. Taking advantage of pipelined parallelism in this fashion to process a single datastream occurs frequently in software as a pattern to address IO limited problems.

SISD over MIMD can be achieved in either a similar fashion to SISD/SIMD (by only utilising one of the many instruction stream-data stream pairs), or by means of a global synchronisation lock between all the instruction streams, so only one instruction stream is ever progressing at any one time. A system constructed in this fashion is capable of out-of-order execution of the elements of the instruction stream, and complex branch prediction operations—though with comparatively less of a speedup than that which is introduced by instruction pipelining.

**Single Instruction Stream, Multiple Data Streams (SIMD)**

The SIMD interface describes a system that processes a set of parallel data streams according to the same instruction stream. It is defined as a set of parallel data streams that pass through the CPU, controlled by only one instruction stream. A sufficiently large SIMD processor is capable of operating on N arguments in the length of time that an SISD processor can operate on 1, giving an average latency of less than the length of

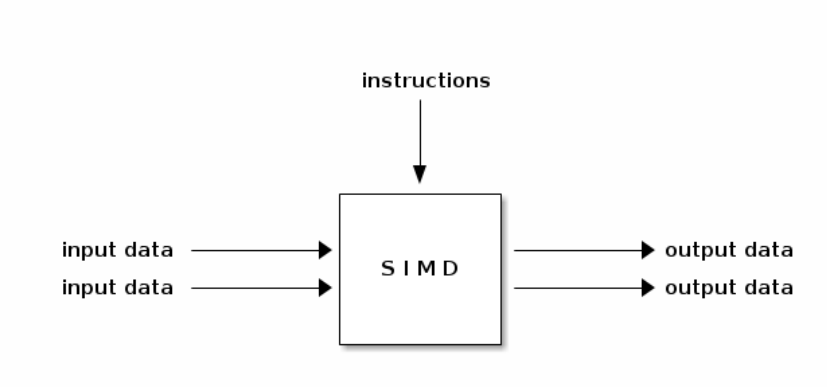



FIGURE 1.2

time it takes to process an instruction on an SISD machine. It appears as if the first instruction was delayed, but the successive instructions were performed instantly.

Example Systems:

- Array Processors [9][10]
- Vector Processors [18][19]
- Array Languages [20][21]
- Map-Reduce Frameworks [22]

The SIMD interface is often implemented over SISD systems due to the concise description this interface affords for operations on collections of data [20]. The popular *map* and *reduce* operations of functional and big data programming languages are both SIMD operations, as are the element-by-element vector operations of Matlab, Octave and Python that remove the need for the programmer to explicitly loop through collection data in order to perform primitive arithmetic operations. Similarly, CPU SIMD register operations can be executed using scalar functional units operating without the delay of decoding N instructions for each of the N arguments [19].

SIMD/MIMD is achieved by duplicating the instruction stream so it is passed with each data stream. This method is the fundamental technique behind massively multicore and cluster computing software, where concurrent hardware is utilised in a deterministic fashion.

**Multiple Instruction Streams, Single Data Stream (MISD)**

The MISD interface describes a system that processes a single data stream according to a set of parallel instruction streams. It is the Flynn taxonomy category that shares

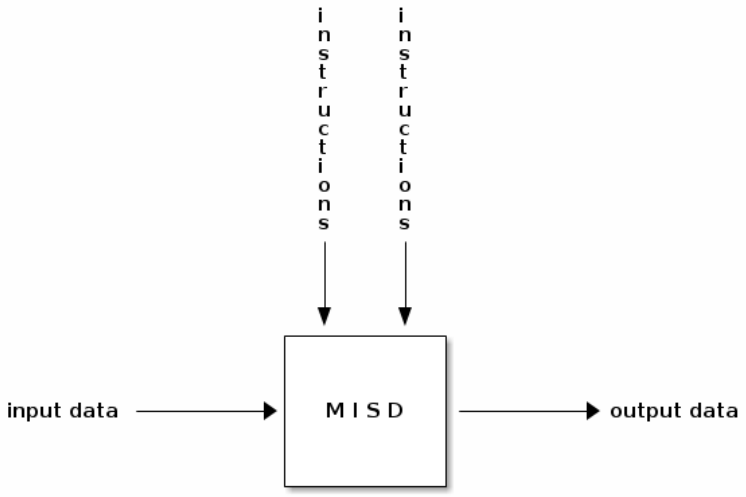



FIGURE 1.3

much in common with data-flow programming, stream processing, neural-network-like hardware and software and, in particular, hardware and software pipelines of IO limited processing systems, as this approach hides the latency of processing large batches of data by staggering the output instead of delivering it in one go.

While MISD hardware typically underlies all single core CPUs and their SIMD register operations, MISD pipelines occur throughout the software stack in order to hide the latency or increase the responsiveness of IO limited processes. A classic example is the shell pipeline on multitasking OS: data streamed from a single source (typically an IO bottleneck) is processed by a set of SISD applications running in parallel—MISD/MIMD —allowing disk IO and computation to be performed in parallel, without any explicit requests or the complications involved in asynchronous communications.

Example Systems:

- Dataflow Processors [23]

  (where each datum is (potentially) associated with many instructions, and is evaluated as soon as the arguments are available. Dataflow processors also exhibit MIMD behaviour, depending on the nature of the associations.)

- Neural Network Processes and Processors [24]

  (where each datum is 'fanned-out' to many nodes in the network, which correspond to instructions.)

**Multiple Instruction Streams, Multiple Data Streams (MIMD)**
Examples:

- Very-Long-Instruction-Word Processors [25]

  (Where multiple instructions are explicitly requested to be performed simultaneously. This is MIMD at the Instruction-Level.)

- Dataflow Processors [23]

  (Where multiple instructions are implicitly requested to be performed simultaneously, due to the "independence of their dependencies." This is MIMD at the Instruction-Level, though dataflow processors also exhibit MISD behaviour.)

- Reduction Processors [26]

  (Lazily-Evaluated Dataflow processors. No or low-priority speculative execution [27]. Each instruction is processed once its result is required by an executing instruction.)

- Temporal Multithreading Processors [11]

  (Where multiple data-instruction stream pairs are performed in an overlapping fashion. This is MIMD at the Thread-Level.)

- Simultaneous Multithreading Processors [28]

  (Where multiple data-instruction stream pairs are performed simultaneously. This is MIMD at the Thread-Level.)

- Multicore Processors [29]

  (Where multiple SISD or MIMD processors operate on their own data-instruction stream pairs simultaneously. A miniaturisation of a Symmetric Multiprocessor Computer. This is MIMD at the Thread-Level.)

- Symmetric Multiprocessor Computers [2]

  (Where multiple SISD or MIMD processors operate on their own data-instruction stream pairs simultaneously. A predecessor of Multicore Processors. This is MIMD at the Thread-Level.)

- Cluster Computers [30]

  (Where multiple SISD or MIMD processors operate on their own data-instruction stream pairs simultaneously. A magnification of a Symmetric Multiprocessor Computer. This is MIMD at the Thread-Level.)

- Cloud Computing Infrastructure [31]

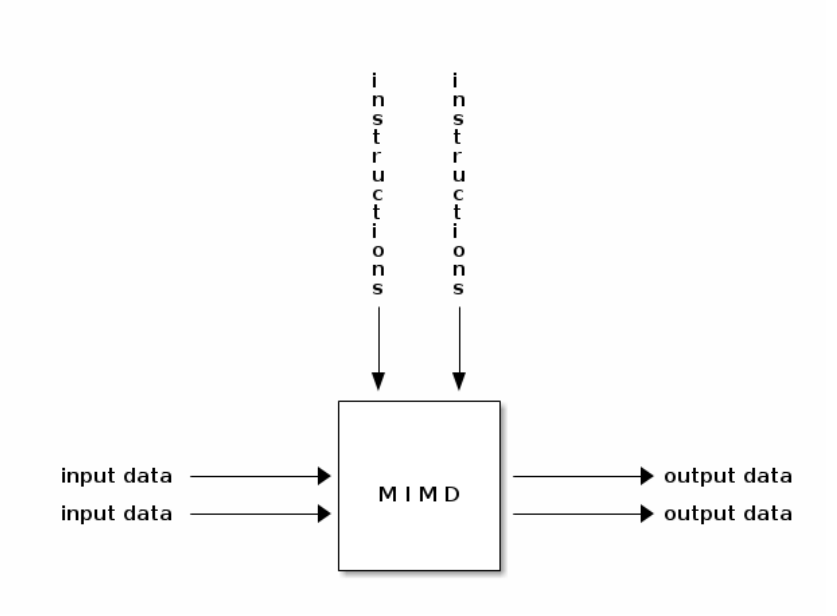



FIGURE 1.4

The MIMD model covers all distributed computing architectures, and corresponds to the more general concurrency based models of computation, including multicore architectures and network clusters, active threads and processes in a multitasking OS, and the Internet as a whole.

While MIMD hardware systems are typically implemented on top of sets of MIMD systems (networked multicore systems) or sets of SISD systems (the set of cores in a multicore CPU), the GPU architecture is constructed from an MIMD system built over a set of SIMD systems.

MIMD systems are advantageous in that they support an overlaid SIMD or MISD system without *needing* to internally serialise any parallel operations —and so without necessarily incurring any system slowdown.

**Discussion**

Since the Flynn taxonomy categorises parallel architectures based on a single aspect of their interface (the number of parallel streams), many of the internal mechanisms of parallelism that have been explored over the last 60–70 years of computer history appear indistinguishable from this point of view, or straddle multiple categories. For example, a processor that contains a single instruction stream, but internally contains multiple Arithmetic Logic Units or Multiple Floating Point Units that are capable of simultaneous operation on distinct registers, does not present as a parallel processor under the Flynn taxonomy. Similarly, a processor that contains several instruction and

data streams, but which internally is restricted to operating on no more than one pair of registers simultaneously, presents as a parallel processor under the Flynn Taxonomy.

Behind the instruction-stream/data-stream interface, a processor is composed of independent logic units and state registers, and an instruction-stream "interpreter", that engages or disengages the underlying logic units on specific registers. This "interpreter" can act on an abstract language, that dynamically assigns operations to unused logic units [32] and blocks processing until these logic units have completed [17], or on a concrete language, where every instruction is capable of controlling all internal logic units simultaneously, and the processor takes no action to prevent the same unit being engaged while it is still in process [25]. While the latter option allows for more expressive use of the inherently parallel components of the CPU, there are software development and deployment costs to such an architecture that limit its applicability [2].

An intermediate design between these two extremes is the dataflow processor [23]. This processor "interprets" a serialised dependency graph of instructions, rather than a stream, which is then evaluated by dynamically scheduling the independent logic units of the processor as they become available. As a consequence of this approach the begin-time and end-time of the operations described in the instruction stream are performed in an order that may bear no relation to the ordering in the serial representation of the dependency graph. Approaches put forward as hybrid Von-Neumann and dataflow architecture [33] attempted to take advantage of the scenarios where superscalar processors unexpectedly outperformed dataflow processors [34] due to reduced ('free') synchronisation costs, and where dataflow processors allowed for multithreading behaviour at a lesser cost than superscalar processors could provide. Commodity desktop computers like the Intel i7 family of processors avail of this functionality in a limited form, where a small, sliding window of the instruction stream is de-serialised into a dependency graph, and then executed out-of-order. The Intel processor logical threading approach is built on top of this mechanism, where the dependency graphs of the two logical-thread instruction streams are treated as a single graph, allowing for inter-thread dependencies to exist and be performed meaningfully, while also allowing for operations in the two threads to be performed without any interdependent operations within the execution window.

Speculative execution is another approach to parallelism that is difficult view in terms of the interfaces of the Flynn taxonomy. For speculative execution, not only are the register operations performed in a different ordering to how they appeared in the serialised form of the dependency graph, but also out-of-order to how they should be performed when evaluating the dependency graph. This is possible for subgraphs of conditional expressions that do not internally make use of the value that lead to the conditional branch. In these situations, portions of the consequences of both outcomes of the branch

can be executed before the value of the condition is known, and one set of consequences discarded once the condition has been evaluated. Like logical hyperthreading, this behaviour falls out naturally from the construction of a dependency graph from the instruction stream of the processor. In best-case circumstances the consequences of both outcomes of the branch can be evaluated simultaneously, as there will necessarily be no further inter-dependencies between their sub-graphs, and sufficient computational units that the operations of both subgraphs can be scheduled simultaneously.

Parallel cache hierarchies that feed parallel cores do not feature in any way in the Flynn taxonomy, since they appear as either external to the incoming datastreams to an SIMD processor, or as slower internal registers available to an MIMD processor.

The modern commodity desktop computer contains elements of parallelism inherited from pipelined processors [8], superscalar processors [17], streaming and vector processors [35][36], dataflow processors [23], simultaneous multithreading [28] and temporal multitasking processors [11], as well as being a miniaturisation of symmetric or homogeneous multiprocessor machines [2]. Various other parallel architectures exist, such as heterogeneous system-on-a-chip processors [37], dynamically reconfigurable Field Programmable Gate Array (FPGA) processors [38], and memristor based computation-in-memory decentralised computers [39], and the introduction of these new paradigms are expected to be the next necessary step in overcoming declining Moore's Law behaviour of each new generation of the commodity desktop computer.

## 1.4 Parallelism in the Modern Desktop Computer

The modern desktop computer is composed of a hierarchy of parallel systems of various forms—some special purpose, some general purpose, some logically parallel, some physically parallel—and realising the benefits of each component requires an overview of the depth and extent of this hierarchy.

At the topmost level, the desktop computer is a single node in a vast MIMD networked system: the Internet. This provides the grossest level of parallelism that is leveraged by the computational cluster [30]. Latencies at this scale are high, and bandwidth is both limited and frequently severely non-deterministic—a factor which has driven the development of such tools as the Hadoop framework [22][40][6] as an approach to managing this complexity.

Within the box, so to speak, there are four main components capable of performing independent tasks, interacting via Direct Memory Access (DMA) regions of a shared volatile store. These are the CPU chips, the GPU devices, the network interface, and

the persistent storage system. The independent operation of these devices—two general purpose, two special purpose—is sufficient to provide high level MISD parallelism for the system as a whole.

Both the persistent and volatile stores avail of batch SIMD parallelism, at the scale of RAID devices [41], the internal hard disk platters, and the movement of data in fixed sized lines between RAM and cache, though as these are single or special purpose devices, this overview will not cover them further.

Both the multicore CPU chip and GPU device follow a similar, converging internal structure, with slight variations to tune their performance in favour of one use case over another, as is described below.

### Multicore CPU

#### CPU Core

The CPU core contains the arithmetic logic units (ALU), floating point units (FPU), and embedded vector co-processor units, known as SIMD streaming extensions (SSE) on Intel CPUs. These three units are capable of operating independently of one another, allowing parallel operations to take place on separate CPU registers.

#### GPU

The GPU has a similar structure to the multicore CPU, but with different terminology for equivalent functionality [42].

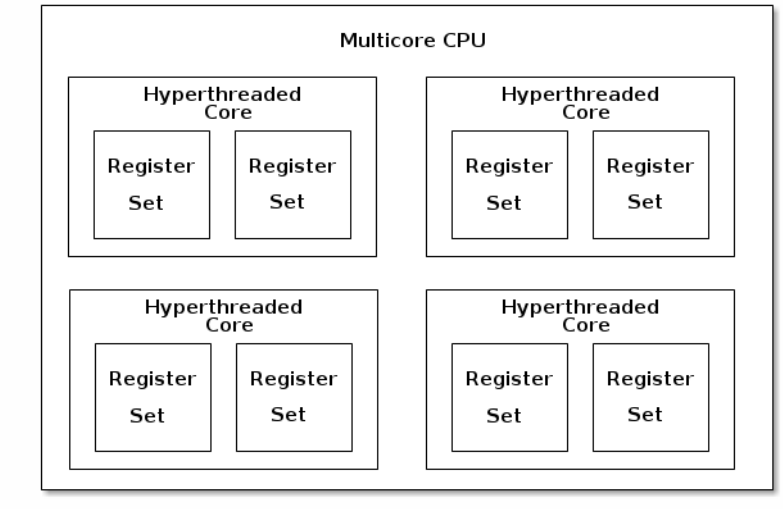



Figure 1.5

The GPU equivalent of the per-core SIMD CPU vector unit is known as a *warp*. A warp provides conceptual lock-step SIMD execution of 32 threads. In current implementations, each 16 thread 'half-warp' is executed in lock-step, and each half-warp is executed in sequence, but this behaviour is not fixed in Nvidia's GPU road map, and reliance on this behaviour is discouraged as it may cease to hold for future GPU devices.

A set of warps is known as a block, and is equivalent to a set of hyperthreaded cores in a multicore CPU. A number of warps running in parallel within a block are implicitly synchronised at the beginning and end of a GPU routine, known as a *kernel*, but can also explicitly synchronise with one another during runtime.

At the coarsest level, a set of blocks running in parallel is known as a *grid*—which can be considered to be equivalent to either the physical cores in a multicore CPU, or a set of multicore CPUs running in parallel. A grid has the coarsest level of synchronisation, which only occurs at the beginning and end of a kernel's execution. There is no mechanism for explicitly synchronising blocks in a grid during a kernel's runtime, but since the blocks in a grid are synchronised at the end of a kernel's execution, constructing a grid-wide kernel as a pair of *before* and *after* kernels that are invoked in sequence allows for a grid-wide synchronisation to occur mid-way through the grid-wide kernel.

## 1.5 Synchronisation & Lockstep Execution

Parallel systems built over more general concurrent systems (whether physically as a network of independent nodes in a cluster, or conceptually like SIMD parallelism built over a threading framework) inherit a certain degree of non-deterministic behaviour from the underlying concurrent machinery. In such systems it is necessary to have a concept of synchronisation, which specifies single points where the state of the parallel system is guaranteed to be predictable, even though between these synchronisation points the state may be unpredictable. At the software-level, synchronisation points can be explicit (via a command called by each thread), or implicit—which usually occurs when a program switches from one organisation of parallel threads to another.

The use of synchronisation points hides the underlying reliance on thread semaphores, shared and mutex locks, or Compare-And-Swap style atomic transactions at the thread-level, and memory-barrier superscalar processor operations at the instruction-level.

With this concept in mind, parallel systems like the per core SIMD vector registers can be thought of as parallel systems that synchronise after each operation—this can be thought of as progressing in *lockstep*. Conversely, CPU thread or GPU block based

multicore CPU or GPU SIMD software can be considered to move in lockstep at the resolution of the synchronisation points, rather than each operation.

## 1.6 Parallel Speedup

Amdahl's Law [16] and Gustafson's Law [43] are two principles that describe how parallelism can improve the runtime of a parallel program. These rules deal separately with either improving the runtime by reducing the latency of the inner operations of a program, or by increasing the throughput or bandwidth of the inner operations.

**Parallel Limits– Amdahl's Law**

$$Speedup = \frac{1}{1 - f + \frac{f}{P}} \tag{1.1}$$

***where***

$f$ = fraction of program that is parallelisable

$P$ = number of available processors

Amdahl's law gives the limit to the speedup the same program can exhibit when run on any number of cores. It is concerned with applications that wish to avail of parallel speedup to reduce the system's latency by as much as possible. Systems subject to Amdahl's Law initially receive a boost in speed as new cores are provided, but soon hit a point of diminishing returns where adding subsequent cores provides less and less of a speedup, until the runtime cannot be reduced to less than a base constant. This constant is the runtime of the inherently serial portions of the program—such as synchronisation points, initialisation procedures, or IO operations that cannot be parallelised. The proportion of time spent during these necessarily serial portions compared to the time spent in parallelisable portions is the maximum speedup that the software can achieve through parallelisation.

Amdahl's Law is formulated for processing datasets of a fixed size, and can be used to determine what fraction of a system's computational resources should be allocated to the program in order to get the best gains from their use [44]. When viewed in this fashion, it is used for determining what the optimal core-to-data ratio should be when designing a system that takes advantage of the behaviour characterised by Gustafson's Law.

**Parallel Speed-up– Gustafson's Law**

$$ScaledSpeedup = P + (1 - P) \times (1 - f) \tag{1.2}$$

***where***

$f$ = the fraction of the program that is parallelisable

$P$ = the number of processors

Gustafson's Law applies to achieving parallel speedup by increasing the system's bandwidth by adding more nodes to a cluster, or memory and cores to software running on a multicore system. While this approach does not actually increase the speed at which the individual data are processed, it allows more data to be processed in the same period of time.

Gustafson's Law provides a mechanism for avoiding the point of diminishing returns predicted by Amdahl's Law, since in many parallel algorithms the inherently serial bottlenecks that dominate Amdahl's Law are of a fixed cost, that is neither proportional to the size of the dataset being processed, nor the number of processors being employed. In these cases Amdahl's point of diminishing returns is unreachable, as the software will always be spending a larger proportion of its time processing the data rather than performing synchronisation or initialisation operations.

Gustafson's Law is the primary justification for GPU based computing and Cloud Computing, where a reasonable runtime time is achieved by acquiring a suitable number of cores to process the dataset, rather than tuning the software to make maximum use of the cores at hand.

## 1.7 Parallel Slowdown

Parallel slowdown manifests as an inverse form of Gustafson's Law, where as more cores are made available to the software, the overall throughput of the software decreases, until the runtime is slower than the same software running with less cores. This behaviour is due to the underlying behaviour of the hardware, or conflict with the wider ecosystem of OS daemons, services, and background user processes, rather than an error in the algorithm or implementation.

The most common cause of parallel slowdown for a multicore or GPU system is due to the behaviour of the memory hierarchy hidden behind the flat virtual memory model that is exposed to the programmer. The two primary forms this takes are thrashing—which is a well known issue in serial software as well as parallel software; and cache contention —which only occurs in parallel or concurrent systems [45][46].

**Thrashing the Cache**

The cache hierarchy that exists between the CPU and RAM is an attempt to smooth out the growing disparity between CPU speed and RAM speed. Unlike in the GPU, where transporting data up and down through the memory hierarchy must be done manually, CPU caching is an entirely hidden and automatic process, which uses a set of hardwired heuristics known as eviction strategies to determine how and when data should be pulled from RAM into the cache, or pushed from the cache back to RAM.

A fully-associate cache works analogously to virtual memory demand-paging, where an initial reference to a cache line replaces the least-recently-used (or some other heuristic) line of the cache with the newly referenced line, and subsequent references operate on the cached line in this location.

Cache thrashing is a similar problem to the hard disk thrashing that occurs when a program uses more virtual memory than there is physical memory available, and spends the bulk of its runtime copying data from the hard disk to ram and back to the hard disk. As the RAM size of computers grew since the initial introduction of virtual memory, this problem became less and less apparent, but it has since resurfaced due to the widening gap between RAM speed and CPU cache speed.

In both single core and multicore CPUs this behaviour occurs when the program operates on a larger region of main memory than can fit inside the CPU cache. The most straight forward approach to dealing with this problem is to re-organise the algorithm so it only operates on short contiguous chunks of main memory at a time. This memory access pattern can be easily accommodated by the current caching strategies employed in hardware. Due to the drastic difference in IO speed between the cache and RAM, algorithms can afford a significant amount of computational overhead in order to achieve this new memory access pattern, while still improving runtime over an unoptimised version.

For multicore CPUs there is a further complication in that the cache-to-core ratio is not constant between different CPU models. Some models have a completely unique set of level 1, level 2, and level 3 caches for each core, some hyperthreaded multicore CPUs shared the same level 1 cache between each pair of logical processors, and some models share the level 3 cache between all cores. This causes problems in two circumstances:

1. Multicore CPU software developed for one CPU model is run on a different model that shares cache between cores.

2. Scaling the software from a small number of CPU cores to all CPU cores. If the cache was already fully occupied for $\frac{N}{2}$ cores, scaling out to $N$ cores will result in a

dramatic slowdown with a lower throughput per core ratio than the less parallelised version.

This situation can occur for multicore CPU programs, even if the algorithm has been modified to operate on chunks of RAM, if the chunks are too large. This problem can be avoided by either configuring the software with a different chunk size for each CPU model it is run on, or by choosing an arbitrarily small chunk size, that is guaranteed to fit several times over into a core's cache, whether it is shared or not.

**Cache Contention**

Cache contention is the term used to describe the effect of multiple threads of execution attempting to fully utilise the available cache [46][45]. This phenomenon occurs in situations where multiple processes or threads are sharing at least one level of the same intermediate memory hierarchy. For SIMD multithreaded software, it can be managed by constraining each thread of the software to only make use of $\frac{1}{N}^{th}$ of the available cache space. Managing the situation in cases of MIMD software is more involved, as the rate at which cache lines are re-used can result in threads *over-utilising* or *under-utilising* the cache, based on the relative rates at which they reference their respective memories, rather than their individual memory access patterns.

## 1.8 This Work

This thesis presents an investigation into the effectiveness of two related mechanisms of parallel computation that are readily available on a commodity personal computer: Multicore CPU hardware and general purpose GPU hardware. The aim of this thesis is to address the question of what computational tasks are the generic CPU and GPU processor types more suited for, in the context of desktop scientific computing. This investigation is conducted by means of an exemplar image processing package, IMPAIR, which presents a set of parallelised image processing algorithms. The performances of this set of image processing algorithms are evaluated in terms of their total running times, resource requirements, scalability, and to what extent the strategies are in a position to address the medical and astronomical imaging specific problems of achieving gigapixel deconvolution, realtime deconvolution, and kilopixel-to-megapixel blind deconvolution in a practical fashion on a commodity desktop computer.

The implementation of these algorithms attempts to take advantage of the presence of the independent physical cores, the shared logical cores, and the floating point vector registers of the Multicore CPU, and the similar structures found in the GPU architecture, referred to as blocks, warps, and half-warps respectively in the CUDA

language specification. The effectiveness of this attempt is discussed in detail in Chapter 6, particularly with respect to the scalability of the strategies for the CPU implementations, where the runtime performance is subject to worst-case interaction with the memory hierarchy.

This evaluation found that the GPU Naive strategy is capable of 24 frames per second unregularised deconvolution of a 1 megapixel ($1024 \times 1024$ pixel) image, and 24 frames per second wavelet regularised deconvolution for a 200 kilopixel ($512 \times 512$ pixel) image, while the CPU Naive strategy achieved 24 frames per second deconvolution for 60 kilopixel ($256 \times 256$ pixel) images.

The GPU streaming strategy was found to be capable of deconvolving gigapixel images at a rate of less than 15 seconds per iteration for the wavelet regularised algorithm, and 6 seconds per iteration for the unregularised algorithm.

The larger memory requirements of the CPU Topdown strategy are an obstacle to scaling to the gigapixel range, but for images in the megapixel range the CPU Topdown strategy was found to perform within $5\times$ the speed of the fastest GPU strategy.

This thesis presents two parallelised implementations of the Wavelet-Regularised Richardson Lucy algorithm, suitable for spatially variant image deconvolution of gigapixel images on commodity desktop hardware and the realtime deconvolution of a confocal microscopy image acquisition stream. To the best of the author's knowledge, the GPU and Multicore CPU performance of this Wavelet-Regularised Richardson Lucy algorithm has not previously been investigated.

Through the implementation of the parallelised wavelet-transform algorithms, aspects of the results of the theoretically optimal discrete wavelet transform algorithm on the GPU presented by Song et al [47] have been tested; IMPAIR's implementation of the Song et al Block-Based DWT performs as well as IMPAIR's previous transpose-based DWT algorithm in all cases except for the GPU Streaming Strategy, where the Block-Based DWT out-performed the transpose-based DWT. As an exact re-implementation of the Block-Based DWT was not implemented by IMPAIR, future work will include a second comparison between these two approaches.

Based on the wavelet transform library, a parallelised wavelet-shrinking library has been developed for the GPU and CPU and its behaviour profiled.

The novel contributions of this work are the Multicore CPU and GPU implementations of the wavelet-regularised deconvolution algorithm, and the investigation of the performance of both these algorithms and the unregularised Richardson-Lucy deconvolution algorithm on the Multicore CPU and GPU. This investigation has laid the groundwork

for the future developement of a spatially variant wavelet regularised deconvolution image processing tool, and for the developement of a wavelet regularised blind deconvolution algorithm, which would have been impractical in practice without the underlying Multicore and GPU parallelisation.

The runtime performance of an earlier version of the regularised and unregularised CPU Naive and Topdown strategies, and an earlier version of the unregularised, and in-progress version of the regularised GPU Streaming strategy were presented at the SPIE Image Processing: Algorithms and Systems XI Conference in February 2013 [48].

The structure of this thesis is as follows:

Chapter 2 will present a broad overview of the field of computer vision, and its relationship to image restoration, concluding with an overview of the various approaches to solving the ill-posed inverse problem of image deconvolution.

Chapter 3 will detail the domain-specific opportunities for the parallelisation and optimisation of image processing primitives that are required by the image restoration algorithms discussed in Chapter 2.

Chapter 4 will present the development and testing of the parallel wavelet transform and wavelet shrinking libraries, using the techniques of Franco et al, van der Lann et al, and Song et al as described in the literature[49][47][50].

Chapter 5 will present the image restoration performance of the algorithms described in chapter 3, and the runtime performance benchmarks of the parallelisation strategies used in their implementation.

Chapter 6 will present an analysis of the results of two of these strategies in terms of the Multicore CPU algorithm's interaction with the Multicore CPU's cache-hierarchy.

Chapter 7 will then finish with an outline of the intended future developments for the software.

# Chapter 2

# Image Processing

## 2.1 Introduction

As it stands, the modern desktop computer is a massively parallel machine, capable of processing large datasets via a hierarchy composed of coarse to fine grained parallel hardware with a corresponding memory hierarchy. Large scale image processing tasks are particularly responsive to parallel hardware [51] such as this, and are now within the grasp of a portable application software for a wide range of commodity hardware.

Tasks that have until recently required batch processing on small cluster computing frameworks, can now be comfortably performed interactively on large datasets, or even in realtime as the image dataset is acquired [52]—this enables general and transferable solutions to common image processing tasks to be employed across the board in desktop applications of computer vision.

## 2.2 Computer Vision, The Image Processing Pipeline & The Semantic Gap

This thesis is concerned with the development of parallelised deconvolution software for astronomical and medical imaging, as the deconvolution algorithm employed by the software has a history of use in these areas [53][54]. However, there are similar image and signal processing based domains that feature deconvolution problems, to which any deconvolution software might find application. For example, image deconvolution has been used in the areas of long range imaging, to restore images in the presence of atmospheric turbulence [55][56][57], motion blur caused by a hand shake during a hand-held camera's exposure [58][59], and the artificial deepening of the depth of field

of an image [60][61]. The interaction of a deconvolution process with other processes in the general domain of computer vision is beyond the scope of this thesis, but a brief description is presented here as an introductory section.

The term *the semantic gap* [62] is used to describe the discrepancy between the large amount of useful knowledge that is present in a digital image, and the small amount of domain specific knowledge that a software system presented with the image can process. The field of computer vision (CV) is the area of research devoted to closing, shrinking, or bridging this gap, and allowing software productive access to the vast, and exponentially increasing amount of digital image data that is currently acquired and archived by humanity.

Computer vision algorithms can be generally divided into two coarse categories [63]: those that slot into a stack of general purpose computer vision algorithms, producing image data to be viewed or processed by another algorithm; and those with a specific domain, that consume image data and produce non-image data (for example Optical-Character-Recognition algorithms), or associations between image data and non-image data that can direct domain specific image processing tasks (for example the iterative Optical Character Recognition image-preprossing approach described in [64]).

The approaches used in the solution of one domain-specific CV problem are at risk of being of little use to the solutions to similar problems in related domains, or of responding so differently to pre-processing steps as to render them counter-productive. For example, the pre-processing approach described in [64] relies upon the assumptions of the Optical Character Recognition of scanned and faxed documents, to the point that its applications to the classification of non-character images are not obvious or likely. However, the recent progress in the application of convolutional-neural-network software to such knowledge extraction problems [65] has found success in cases where techniques used in one domain-specific CV problem can be successfully repurposed and transfered to another [66].

A simplified version of the stack described in [63] is as follows:

- The optical system
- The acquisition system

---

- Image restoration processes
- Image segmentation processes

- Feature extraction processes

---

- Feature identification system
- Knowledge extraction system

where each element in the pipeline only interacts with its neighbouring elements in a feedforward manner (in the simplest case). The creation of a more expressive taxonomy of image analysis algorithms is described in [67] and [68]. The three central elements of this pipeline cover computational image processing related tasks, into which IMPAIR falls, while the first two hardware stages, and the final two information-system stages, are significantly different fields of study.

## 2.3 Image Restoration

The image restoration process in the image processing pipeline aims to minimise the degradation that has occurred to the theoretical *true image* as it is processed and transformed through the optical and acquisition systems [63]. This degradation takes the form of blurring or warping of the true image by the optical system, and sampling errors—additive noise—introduced by the acquisition system.

Under certain conditions the observed image can be 'restored' to a spatial resolution that exceeds that of the acquisition [69]. These conditions are available to the restoration of astronomical images [70][71] (a star-field has a number of small foreground objects on a uniform background of values close to 0).

As it is an early process of the image processing pipeline, any improvements, whether in terms of runtime, reduced resource requirements, or restorative ability, will cascade onwards towards all processes that depend on its output [72][73][74][75], whether they be applied in the domains of realtime image processing, smartphone or embedded image processing, or the large scale mining of image datasets, like those found in medical imaging [76].

## 2.4 Metrics of Image Quality

For image restoration purposes, the two fundamental measurements of the degradation or quality of an image are the signal to noise ratio (SNR) of the image pixels, and the spatial resolution of the image. These aspects describe the behaviour of the sensor and optics of

the acquisition system, respectively. The maximum spatial resolution that an image can represent is determined by the nature of the impulse response, or point-spread function (PSF), of the acquisition system. While an estimate of the SNR of the image data is not necessary for the application of the image restoration processes described in this chapter, an estimation of the acquisition system's PSF is required by the Richardson-Lucy [77][78][79] algorithms discussed in sections 2.8 and 2.11. This data can be obtained either from measurement, or derived analytically from a model of the image acquisition system [63].

The Modulation Transfer Function (MTF) is a representation of the quality of an imaging system, in terms of the dampening effect the system has on the amplitudes of its input spatial frequencies. The information displayed is in terms of the contrast between the light and dark regions of the image for the range of possible spatial frequencies, where a fully discernible spatial frequency will have a maximum contrast, and an unresolvable spatial frequency will have a contrast of zero.

## 2.5 Denoising

Denoising signal data, or designing algorithms that are tolerant of the presence of noise in signal data, is an unavoidable problem for all computer vision applications [80][81]. Various techniques, linear and non-linear approaches, are employed to this end which range from spatial filtering and Wiener techniques to common Wavelet approaches [82][83][84][85][86]. Which of these is chosen will depend upon the computer-vision process, for example whether or not the additive noise in the input data is Gaussian distributed, and the requirements of the processes that will follow from the denoising step, for example whether or not photometry will be performed on the output data.

Denoising via crude global thresholding is most appropriate to signal data taken under highly controlled conditions, such as evenly illuminated printed text. More expressive variants of thresholding, where noise levels are reduced or shrunk, but not flattened, across the image have more general applications, and are more suited to adaptive methods, such as wavelet based denoising [87].

Linear filter based approaches perform denoising by convolving the signal with a filter designed to suppress variations in the image with a high spatial frequency. Depending on the choice of filter used—for example a Gaussian or box filter—this denoising approach can result in significant loss of contrast, and loss of spatial resolution in the signal. Consequently, these approaches are potentially counterproductive for image restoration processes that require a deconvolution operation to restore contrast or improve the spatial

resolution of the signal [88]. The behaviour of non-linear filters are more suitable for situations where features such as object edges must be preserved [89][90].

For an approach to generating iterative denoising algorithms from a 'black-box' denoising operation (in a similar fashion to the blind deconvolution algorithm discussed in 2.12), see [91].

Both thresholding and filter based approaches are dependent to some degree on the choice of specific parameters—either the threshold level, or the size and type of image filter. Adaptive approaches to denoising reduce or remove the need for such tunable parameters, removing the need for human supervision during the denoising process [92]. This behaviour is particularly desirable in situations where the denoising process is run frequently over a changing set of signal data where the same parameter value will not be applicable to each instance—for example, within an inner loop of an iterative image processing algorithm.

## 2.6 Deconvolution & The Deconvolution Problem

Deconvolution is the term given to the reversing or undoing the convolution of a signal with a kernel. It is an ill-posed inverse problem—meaning the deconvolution of a signal with respect to a kernel can only be estimated, as there is no guarantee of a unique nor stable solution [70].

The success of a deconvolution algorithm depends entirely on the properties of the kernel and the properties of the original signal. For certain input signals satisfactory deconvolution will not be possible, and likewise for certain convolution kernels. The clearest example of this is a low-pass filter kernel (like an averaging filter) that removes the presence of any high spatial frequencies in an image.

For signals and kernels with the appropriate properties, full reconstruction of the signal is possible via the deconvolution technique referred to as *inverse filtering* [63], or by Van Cittert deconvolution [93].

The deconvolution of images acquired by a detector system is complicated due to the presence of additive noise in the acquired image. This noise is amplified by inverse filtering or similar straight forward techniques for pure deconvolution. See Figure 2.2 for an example image.

Similarly, in a real world detector system any measurement of the PSF will contain imperfections, since it itself will be subject to additive noise and minor spatial variations. The problem of deconvolution for the purpose of image restoration suits algorithms that

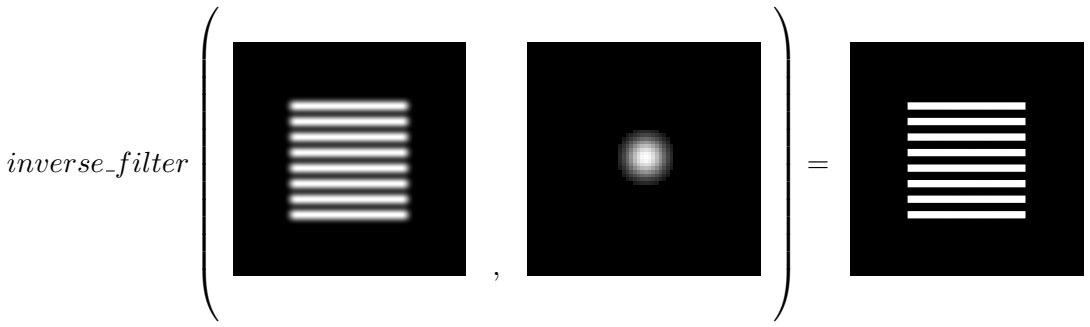



FIGURE 2.1: IMPAIR Inverse Filtering Example Images. Examples of a 'perfect' inverse filtering operation performed by IMPAIR.

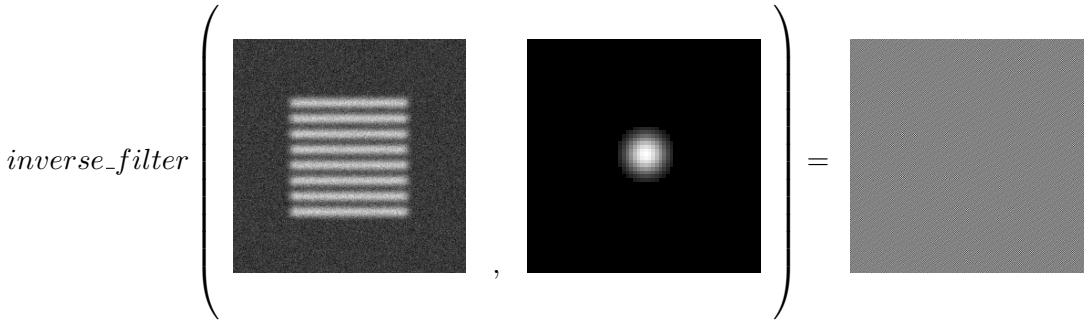



FIGURE 2.2: IMPAIR Inverse Filtering Noise Amplification Example Images. Extreme noise amplification via inverse-filtering a synthetic image with additive noise, showing the unsuitability of this technique for real world images which will contain some amount of unavoidable additive noise due to the nature of the acquisition system.

function acceptably with less than perfect knowledge of the PSF, and in the presence of varying levels of additive noise [70].

## 2.7 The Zoo of Deconvolution Algorithms

Deconvolution algorithms for image restoration range from a historical set of filter based approaches (such as image sharpening, unsharp masking, and the Wiener filter), to a broad and growing toolset of iterative algorithms [70].

Filter based approaches have the advantage of lower computational costs and memory requirements, and—due to their single-pass nature—more deterministic runtimes, making them well suited to problems of realtime image processing. Iterative methods, though more powerful, are more costly, and so historically found less use outside of special purpose hardware.

The typical filter based approaches to image restoration are:

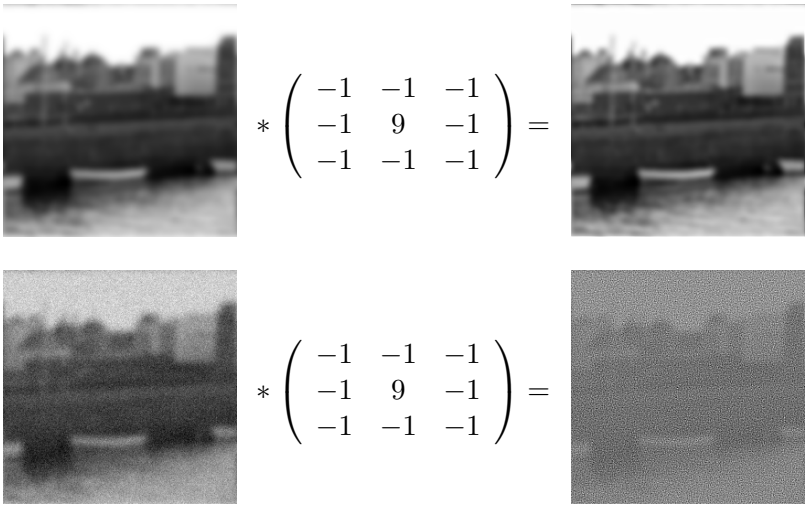



FIGURE 2.3: Image sharpening with a $3 \times 3$ pixel kernel.
Top: Sharpening of a low noise image.
Bottom: Sharpening of a high noise image. Noise amplification in restored image clearly visible.

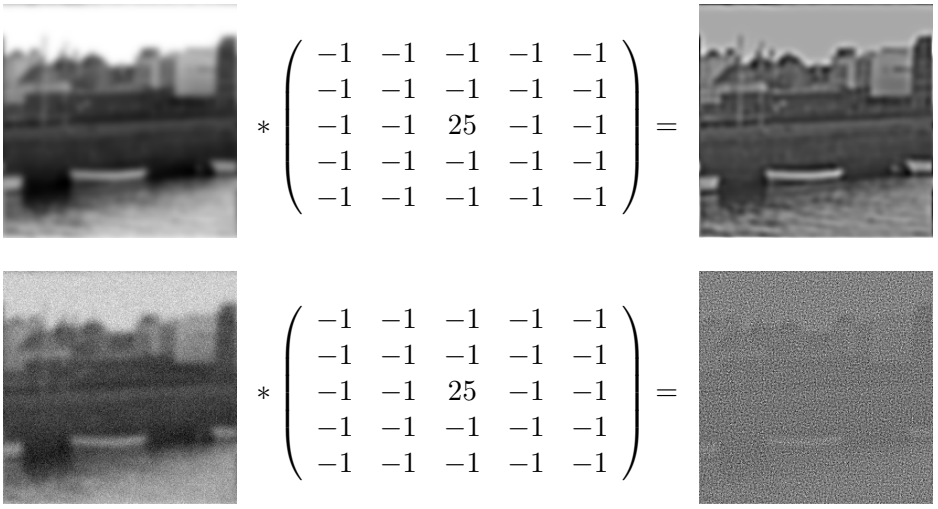



FIGURE 2.4: Image sharpening with a $5 \times 5$ pixel kernel.
Top: Sharpening of a low noise Image.
Bottom: Sharpening of a high-noise Image. Noise amplification in restored image clearly visible.

**Image Sharpening**

This is the convolution of a filter designed to increase the contrast of the image over a small window (typically $3 \times 3$ or $5 \times 5$ pixel regions), and is prone to noise amplification.

**Inverse Filtering**

This is the algorithmic inverse of the convolution operation, defined in the Fourier domain as:

$$deconvolve(a, b) = \mathcal{F}^{-1}\left(\frac{\mathcal{F}(a)}{\mathcal{F}(b)}\right) \tag{2.1}$$

The success of this approach is highly dependent on the level of additive noise in the input signal, and the accuracy of the knowledge of the acquisition system's PSF. See Figures 2.1 and 2.2 for example images.

**The Wiener Filter**

The Wiener filter [63] straddles the border between the above filter based approaches and the more complex iterative methods. It consists of an image filter generated by an iterative algorithm trained on a set of images, that can then be applied to deconvolve and denoise images with similar characteristics as the training set, with the speed and low memory demands common to filter based approaches.

**Iterative Deconvolution**

The most primitive approach to iterative deconvolution is the Van-Cittert algorithm, which is rarely used due to its slow rate of convergence and similar noise amplification properties as the inverse-filter, as it involves no statistical estimation step to guide its convergence. Regularised variants of the algorithm exist [94].

The iterative algorithms commonly employed for deconvolution in astronomy are: [69]

- Expectation-Maximisation Algorithms (EM)
- Maximum-Likelihood Algorithms (ML)
- Maximum A-Priori Algorithms (MAP)
- Maximum Entropy Method (MEM)

These groups classify iterative deconvolution algorithms in terms of the procedure the algorithm uses to determine how it progresses.

## 2.8 Richardson-Lucy Deconvolution

The Richardson-Lucy algorithm is an iterative Bayesian image deconvolution technique, originally designed to correct for the combined problems of image noise and image smoothing, or blurring, that necessarily occur during the process of digital image acquisition in astronomy [77], and later shown to converge at the ML solution [78]. The unregularised algorithm is robust to small errors in the PSF [70] and Poisson noise [79].

However, the algorithm is quite general and can be applied to deconvolution and denoising problems in areas of digital image processing outside of the domain of astronomical image restoration, due in part to the Bayesian basis of the algorithm– which depends on only a single a-priori assumption as to the nature of the deconvolved image.

Since its introduction various extensions and modifications to the RL algorithm have been demonstrated; from regularisation methods [95], which improve the quality of the deconvolved image in the presence of high levels of noise, to software techniques to take advantage of parallel computing [96], to approaches for dealing with a wider range of image restoration problems such as 3D medical imaging [97], correcting for motion blur in consumer grade digital cameras [58], deconvolution with a spatially variant point spread function [96], or blind-deconvolution [98]: where there is severely limited, or a complete absence of, a-priori knowledge of the point spread function.

**Algorithm**

$$e_{n+1} = e_n \left[ \left( \frac{o}{e_n \ * \ psf} \right) \star psf \right] \tag{2.2}$$

where

$$e_0 = o \ * \ psf$$

The initial estimate is the observed image convolved with the PSF. This action removes any of the discontinuities caused by noise in the observed image, producing a smoother image.

At this point the estimate image is less noisy than the observed image, but significantly more degraded. The iterative work of the algorithm is then to gradually reintroduce deconvolved features of the input image, without reintroducing the noise content.

## 2.9 Noise Amplification Problem

While the RL algorithm performs significantly better in the presence of additive noise in the input image than the inverse filtering approach, it is not immune to the problem of noise amplification. After a relatively few number of iterations of the RL algorithm, the only changes made with each following estimation are to improve how well the estimate image matches the noise present in this input image.

*from left: original image, degraded image, degraded image corrupted with additive noise*

*for iterations 0, 2, 4 detail in the input image is restored*

*for iterations 8, 10, 12 detail in the input image remains the same, while artefacts due to noise amplification become more pronounced*

FIGURE 2.5: IMPAIR Noise Amplification Example Images

This can be further exacerbated by the ringing artefacts (Gibbs oscillations) that echo around sharp discontinuities in the input image, which—although not generated as a consequence of the additive noise—are not a part of the true signal being reconstructed, and so contribute to the overall noisy appearance of the image to the human eye.

In order to further suppress the interference of noise in the image restoration process, various extensions and modifications of the Richardson-Lucy algorithm have been developed.

## 2.10 Spatially Variant Richardson-Lucy Deconvolution

Deconvolution with a Spatially Variant PSF using the Richardson-Lucy algorithm is approached in two ways: globally-variant/locally-invariant deconvolution, where the PSF is treated as being invariant for various subregions of the image, and these sub-images are processed as independent images; and locally-variant deconvolution, where a unique PSF is provided for every pixel of the degraded image, which must be processed as independent images.

Both these algorithms are well suited to parallelisation, as they must be performed as the deconvolution of independent subregions even if implemented in a serial fashion [53][54]. Due to the robustness of RL to slight errors in the PSF, this locally-variant deconvolution is often unnecessary in practice.

## 2.11 Regularised Richardson-Lucy Deconvolution

Each iteration of the above RL algorithm converges on its estimate of the true image by comparing its current estimate to the observed image. Any noise present in this observed image is naively treated as part of the signal by this process. Regularised variants of the RL algorithm, such as the Total-Variation regularisation [95], compensate for this naivety by replacing each use of the observed image in the RL algorithm with a synthetic image that is derived from the observed image, but intended to contain less additive noise. These synthetic images are generally produced by constraining the individual pixel values of each iteration's residual image according to an extra input parameter.

The variant of the RL algorithm proposed for astronomy [99] and microscopy [100] avoids the need for an extra parameter by relying on an adaptive technique to constrain the pixel values of each residual image. While this technique is more computationally expensive than the parameter based techniques mentioned above, the lack of an extra parameter makes this variant an attractive algorithm for general use. The original implementation of this technique was shown by [99] to improve the performance of investigated downstream high-level image processing tasks such as feature extraction or feature identification when this image restoration step was performed at an early stage in a larger image processing pipeline.

Wavelet regularisation has also been found to dampen the extent of ringing artefacts on the restored image that are introduced by the deconvolution [101], by tackling the local pixel intensity variation due to ringing as if it were per-pixel additive noise.

The RL or WRL algorithm can be used as a stand-alone image restoration tool or as a component in a more complex image restoration operation, such as image restoration where the PSF varies dramatically across the observed image, or blind deconvolution—where the algorithm iteratively estimates both the PSF image, and the un-degraded input image.

## 2.12 PSF recovery and Blind Deconvolution

So far, the convolution operation has been described as a function that takes two parameters: an input signal, and the convolution kernel or PSF. While this reflects how the convolution operation is commonly employed in image processing, where the kernel is typically far smaller, and a specifically chosen analytical function, it suggest a strict distinction between the kernel and the input signal that does not exist. The convolution $A \star B$ is equivalent to $B \star A$, and $A$ can be recovered by deconvolving $A \star B$ with $B$ just as easily as $B$ can be recovered from $A \star B$ by deconvolving with $A$.

One application of this is to use an un-degraded version of the input image to generate the PSF used in degrading it. Understandably, this finds little use in image restoration problems, as an estimate of the PSF is more easily or accurately obtained by other means, and a suitable un-degraded version of the input image may not exist. However, this property can be used to form the basis of various blind deconvolution algorithms [102][103][104], a problem which has a much wider range of applications than straightforward PSF recovery, and deconvolution with a known PSF.

The blind Richardson Lucy algorithm of Fish et al [98] is constructed from two interleaved applications of the Richardson Lucy algorithm—playing the RL algorithm off against itself, in order to incrementally improve the estimation of both the PSF image and the un-degraded input image simultaneously.

**Algorithm**

$$
\begin{aligned}
psf_{n+1} &= psf_n \left[ \left( \frac{o}{psf_n \ * \ e_n} \right) \star e_n \right] \\
e_{n+1} &= e_n \left[ \left( \frac{o}{e_n \ * \ psf_n} \right) \star psf_n \right]
\end{aligned}
\tag{2.3}
$$

where

$$e_0 = < an\ initial\ guess\ of\ the\ input\ image >$$

$$psf_0 = < an\ initial\ guess\ of\ the\ PSF\ image >$$

This algorithm has the attractive quality of requiring no further image operations or tuning parameters than the original Richardson Lucy algorithm, and so can be constructed from an existing Richardson Lucy implementation in a black-box fashion. Furthermore, this component-like construction of the blind algorithm easily allows regularised variations of the RL algorithm to be used in the inner deconvolution steps, such as the wavelet regularised RL algorithm of Starck and Murtagh [99]. Unfortunately, the choice of an initial estimate for both the PSF and the input image is no longer as straight forward as a uniform prior or $e_0 = o * psf$, as it is for the original RL, and must be provided on a case-by-case basis.

While this algorithm has the advantage of being easily implemented on top of existing RL implementations, the computational demands of a single iteration of the blind algorithm are comparatively enormous, with the PSF estimation step requiring hundreds of iterations, per iteration of the blind algorithm. The expense of this requirement cannot be mitigated by coarse-grained parallelism, and must be addressed by a highly optimised inner Richardson Lucy implementation in order to provide a blind deconvolution algorithm suitable for common use.

The possibility that the algorithm could be implemented with only a single iteration of each inner-loop exists, if certain conditions are met [70][105].

Variations of this algorithm have been proposed, using adaptive denoising processes for regularisation [88].

## 2.13 Related Work

Image processing algorithms lend themselves well to parallel implementations due to the ability to meaningfully represent images as collections of homogeneous, and independent, sub-images, down to the level of single pixels [51]. Consequently, some of the earliest applications of SIMD parallelism in consumer grade hardware have been in the area of image processing. The SSE registers on the Intel x86 processors were originally introduced to aid with the lossy encoding of images and video [35]. The GPU platform, which originated as a special-purpose device for specific image processing operations, first found use as a more general purpose computational device with implementations of numerical algorithms commonly relied on by image and video codecs, such as the Fast Fourier Transform algorithm, and the Discrete Wavelet Transform algorithm.

The computational expense of these transforms becomes a limiting factor to the scale at which any algorithms that rely on them can be used. Consequently, the need for widely available high performance implementations of these algorithms is recognised [21]. The multicore CPU FFTW and the CUDA GPU CuFFT libraries address this for problems that rely on discrete Fourier transforms of image data. However, no equivalent Multicore and GPU discrete wavelet transform library has emerged.

The remainder of the section presents an overview of related work in the specific area of the discrete wavelet transform parallelisation techniques and implementations, and the parallel implementation of more general deconvolution algorithms.

### 2.13.1 The Discrete Wavelet Transform

Efforts to avail of the GPU in the calculation of the discrete wavelet transform predate the introduction of general-purpose GPU devices. Initial investigations implemented the DWT using the GPU's shader-language, and focused on determining whether the filter-scheme or lifting-scheme DWT was more suited to the constraints of the hardware [106]. Due to the write-once-read-many memory limitations found on these early devices, the filter-scheme was found to outperform the lifting scheme [106].

With the introduction of the CUDA compatible GPU devices, this memory constraint was removed, and the lifting-scheme was found to outperform the filter-scheme, due to its ability to perform the DWT in-place [49].

The remaining efforts focused on the performance of various parallelisation strategies and domain-specific optimisations for the lifting-scheme DWT.

These strategies are categorised by Song et al [47] into three groups:

- Transpose-Based DWT (called RC or Row-Column)
- Sliding-Window DWT (called LB or Line-Based)
- Block-Based DWT (called BB or Block-Based)

These groups each take a different approach to which aspects of the DWT are parallelised in a lock-step fashion, and which aspects are parallelised independently.

The Transpose-Based DWT treates a 2D image as a set of independent 1D signals that must be transformed, and is the earliest technique employed to implement the 2D DWT. Each pixel in a row is transformed in lock-step, while each row is transformed

independently. The GPU Transpose-Based DWT has been shown to outperform CPU Transpose-Based implementations with a $> 20\times$ speedup [49].

The Sliding-Window DWT treats the horizontal transform of the 2D image as a set of independent 1D signals, identically to the Transpose-Based DWT. The vertical transform is then calculated incrementally, with the $n^{th}$ pixel of each column processed in lock-step. This approach guarantees a memory access pattern for the vertical transform that is suited to a cache hierarchy, and performs at the same rate as the horizontal transform. Implementing the 2D DWT in this fashion avoids the cost of the transpose operation, but imposes a device-specific limit on the maximum width of the image data.

The GPU Sliding-Window DWT has been shown to achieve a $12\times$ speedup over the GPU Transpose-Based DWT, and achieve theoretically optimal performance on the GPU device [50].

The GPU Block-Based DWT is a cache-oblivious DWT algorithm, that is not subject to the Sliding-Window's limit on the maximum image width. This algorithm divides the image into a grid of overlapping blocks, and processes the blocks independently, while processing the pixels in each block in lock-step. This technique achieves a $2\times$ speedup over the GPU Sliding-Window DWT, and is the fastest DWT parallelisation strategy currently available [47].

### 2.13.2 Deconvolution

Image and volume deconvolution is a well studied application of parallel and HPC computing [107][108][109]. The emergence of commercial grade-parallel hardware with the Multicore CPU and programmable GPU a decade ago led to various investigations into the suitability of these parallel architectures towards the problem of deconvolution in the areas of astronomy [107][108][109], medical imaging [79][110], microscopy[111][112], and digital photography [58].

This review focuses on recent efforts that pushed the boundaries of what deconvolution can be practically applied to, particularly with respect to the until-now impractical aspects such as real-time deconvolution, 3D deconvolution, and blind deconvolution techniques.

Realtime image deconvolution at video stream rates was first achieved on the GPU by Klosowski and Krishnan in 2011 [52], with an implementation of the Krishnan-Fergus deconvolution algorithm [113] built from the CUDA CuFFT library, and lookup-table optimisations. The implementation ran at a speed of 40fps for a $710 \times 470$ pixel (approximately 480p definition video) single-channel image stream, and achieved a $27\times$ speedup over the original CPU implementation of the Krishnan-Fergus algorithm.

Investigations into the performance of 2D and 3D unregularised Richardson-Lucy deconvolution conducted by Domanski et al [114][115] were directed towards identifying the most efficient use of the CPU and GPU node components of a heterogeneous cluster. Their system was based on a top-level run-time load balancing queue that dispatched sections of the dataset to individual nodes, which were then responsible for their own internal parallelisation across the GPU and CPU cores. 3D deconvolution was found to be more responsive than 2D deconvolution to the number of available computational resources, while negative scaling effects were found for 2D deconvolution for high numbers of computational resources due to the communication bottleneck of the top-level load balancing queue.

Multicore CPU and GPU implementations of the Scaled-Gradient Projection deconvolution algorithm [116], a variant of Richardson-Lucy that converges in fewer iterations, though with a higher per-iteration computational cost, have been investigated in the context of astronomical imaging [117] and confocal microscopy [111]. Efficient implementations of this algorithm have been developed and released to the public for astronomical image processing (*http://www.unife.it/prin/software*). Ongoing work is being carried out on selecting optimal values for the various tuning parameters, which determine the rate at which the algorithm converges.

GPU implementations of the 3D scaled gradient projection deconvolution algorithm developed by Zanella et a l[111] have resulted in a 100× speedup over the multicore CPU implementation, and peak 690× speedup over their unregularised 3D Richardson-Lucy deconvolution CPU implementation, due to the combined effects of a faster rate of convergence, and the surplus of computational cores provided by the GPU. This performance boost allows for realtime rates of volume deconvolution for confocal microscopy, which has a 3 minute acquisition time per 33 megavoxel volume.

A realtime unregularised 3D Richardson-Lucy implementation developed by Bruce and Butte [112] achieved deconvolution rates of 10× faster than volume acquisition times for two-colour kilovoxel volumes. This software was made available for free of charge academic use, on request, at the time of publication.

## 2.14 Conclusion

The large scale parallelism that is now available in a modern desktop computer makes a range of iterative image processing algorithms more readily accessible for interactive use. These algorithms have high requirements for the computational and storage facilities of

a machine, and as such were traditionally limited to either special purpose systems, or computational clusters.

These intensive image processing activities fall naturally into a stack or pipeline, that flows from the raw image acquisition, to high level knowledge extraction from heavily preprocessed data. The overall behaviour of such a software stack is consequently heavily dependent on the early stages of this pipeline, as these will be shared most frequently between a broad range of higher level components. Robust deconvolution, a step early on in this pipeline, is now feasible in situations where it had been traditionally written off as prohibitive. The implementation of such a process will trigger far reaching benefits in higher level image processing software—both in transparently improving the quality of these processes due to providing higher quality input data, and by relaxing the requirement for higher level processes to be immune to the effects of poor-quality data.

# Chapter 3

# IMPAIR

## 3.1 Introduction

As shown in Chapter 2, deconvolution is an embarrassingly parallel problem, that is well suited to the parallel hardware of a modern desktop computer. What follows is a description of the algorithms and parallelisation strategies for the development of the C and CUDA Richardson Lucy deconvolution library IMPAIR, first developed as an MPI based cluster computing software tool, and used for both astronomical [118] and radiologica l[96][119][120] image restoration.

Under its original incarnation, IMPAIR consisted of a single manager process, and one or more worker processes that persisted for the lifetime of the manager process. The manager process divided all input images into tiles according to a strategy that attempted to maximise the number of power-of-two square tiles that would be processed, with the smallest number of non-square, non power-of-two sized tiles remaining. For spatially variant images, the tiles and per-tile PSF images were specified explicitly. These tiles formed a job queue, that was fed to the worker pool as workers became available. As the tile sizes were irregular, the execution time of the worker processes was subject to change throughout the lifetime of the manager process. Each worker process accepted jobs from the manager process, composed of pairs of degraded images and PSF images. The worker processes deconvolved these degraded images with the PSF image and transmitted the image back to the manager process.

The deconvolution of the tiles was performed serially, with the convolution operations accelerated by the FFTW2 library in a single-thread configuration, and the wavelet shrinking operation performed using an early wavelet transform library, based on the filter scheme approach, that performed a number of heap allocations per pixel of the tile per iteration of the deconvolution.

This approach to parallelising the deconvolution used the same array-tiling strategy as is needed for serial or parallel spatially variant deconvolution, and the next generation Multicore and GPU incarnations of IMPAIR does so also. The GPU and Multicore CPU update of IMPAIR attempts to address the irregular runtime of the original IMPAIR by strictly adhering to a given tile size for its array-tiling policy.

This update of IMPAIR contains a new wavelet shrinking implementation, capable of scaling to 100 megapixel images with predictable runtimes. Further discussion of this wavelet shrinking implementation is given in Chapter 4.

This update of IMPAIR investigates 3 parallelisation strategies: the Naive strategy, that does not involve any image tiling; the Topdown strategy that is an extreme form of array tiling that chooses the largest possible tiles and processes them on a one-tile-per-core basis; the Streaming strategy, that processes the image one tile at a time, in an N-cores-per-tile fashion; and the Queuing strategy, that processes the image N tiles at a time, in a one-tile-per-core basis. It is expected that the physical and logical core parallelism behaviours of these strategies will be different, due to the granularity of the parallelism each strategy employs. A parallelisation strategy more closely aligned with that of the original IMPAIR was investigated with an earlier version of the wavelet shrinking library, and found to be less effective than the Streaming and Topdown strategies. Its performance is presented in Appendix C, but not discussed further in this thesis.

## 3.2 Convolution of 1D, 2D, and 3D Signal Data

The bulk of the computational load of the plain Richardson Lucy deconvolution algorithm is due to the two convolution operations that are performed at every iteration. Consequently, an optimal plain RL implementation will hinge around the performance of the underlying convolutions, and the choice of which discrete convolution algorithm is chosen to implement them.

Discrete convolution takes two signals as input, and produces a new signal that is a combination of the data from the two input signals. For digital signal processing, the two input signals are typically of unequal sizes or lengths, with the smaller of the two signals referred to as the convolution kernel, finite impulse response (FIR) filter, or Point Spread Function (PSF) for 2D or 3D convolution; while the larger signal is referred to as the input image, or even simply the image. The convolution operation is associative: the convolution of the kernel with an input image is equal to the convolution of the input image with the kernel. The conventional order of the operands is $image_{ij} * kernel_{uv}$, where the first left hand side operand is always the larger of the two signals. Convolution

is also distributive over addition, which will be taken advantage of in the divide and conquer based convolution algorithms.

The size of the output signal is determined by the sizes of the two input signals in the following way:

For 1D convolution:

$$signal_M * signal_O \rightarrow signal_{(M+O-1)} \tag{3.1}$$

For 2D convolution:

$$image_{M\times N} * image_{O\times P} \rightarrow image_{(M+O-1)\times(O+P-1)} \tag{3.2}$$

For 3D convolution:

$$volume_{M\times N\times O} * volume_{P\times Q\times R} \rightarrow volume_{(M+P-1)\times(N+Q-1)\times(O+R-1)} \tag{3.3}$$

In practice, this lengthening of the output signal is often not performed strictly as described above. Instead, only $M$, or $M - O$ samples of the $M + O - 1$ output signal are calculated, and convolutions that require larger output signals preprocess the input signal to fascilitate this. Typically, this is achieved by padding the input signal with either zeros (zero-padded convolution), with the data from the other end of the signal (cyclic convolution), with a mirror image of signal (as occurs when using the Discrete Cosine Transform to accelerate convolution), or by repeating the last element of the signal infinitely (clamping)[121]. Zero-padded convolution is often a sensible choice of border condition for practical work, as it matches the behaviour of systems as they exist in nature, while cyclic convolution lends itself to efficient Fourier based implementations without the overhead of a larger input image. Since it is less expensive to get the effect of zero-padded convolution behaviour from a cyclic convolution implementation than the other way around, for algorithms like Richardson Lucy—which rely on many convolution operations per iteration—it is sensible to base the implementation on a cyclical convolution operation and leave zero-padded convolution as an exceptional case that is employed only when necessary.

The algorithms which now follow detail several approaches to calculating the convolution of two images without any regard for concurrency or parallelism. These are effectively the algorithms for convolution on a single-core machine, but form the basis

for both the top-down and bottom-up parallel deconvolution algorithms for the SIMD register, multicore, and GPU convolution operations.

## 3.3 Discrete Convolution in the spatial domain

The simplest algorithm for the convolution of two images is to process them in the spatial domain. In practice, this algorithm is only useful for the convolution of an input image with an extremely small PSF or kernel image [69], or more rarely in the unusual case where the PSF image is larger than the input image—a situation that can occur when the input image is decomposed into a large number of tiles in order to simulate an optical system with a locally spatially variant PSF [54].

Algebraically, the convolution operation is defined as follows:

$$(Image_{(k,l)} * PSF_{(u,v)})_{(x,y)} = \sum_{i=0}^{I} \sum_{j=0}^{J} Image_{(x-i,y-j)} PSF_{(i,j)} \tag{3.4}$$

$$where$$
$$x > I > 1,\ y > J > 1$$

which corresponds to the following pseudo-code description:

```
for each pixel (i,j) in output_image
  output_image(i,j) = 0
end for

for each pixel (i,j) in output_image
  for each pixel (u,v) in psf_image
    output_image(i,j) += input_image(i-u,j-v)*Psf_image(u,v)
  end for
end for
```

This algorithm requires an $N$ sized input and output buffer, and an $M$ sized PSF buffer, giving a total memory requirement of $2N + M$. Runtime will be proportional to the sizes of the input buffer and the PSF buffer, $N \times M$. Assuming that the input and PSF buffers are of equal size, this gives a runtime of $N^2$. In reality, this will not always be the case, and cases such as $M = 3 \times 3 = 9$ or $M = 4 \times 4 = 16$ will often occur for many filtering operations (such as sharpening, edge-detection, or blurring). The cache-miss penalty for this algorithm will be dominated by the size of the PSF buffer $M$.

## 3.4 Discrete Convolution in the frequency domain

Frequency domain convolution is achieved by a complex-multiplication of the corresponding elements in the input and kernel signals. This results in a significant speed-up over the above spatial domain convolution algorithm for all but the smallest kernel sizes [69]. The performance of the frequency domain algorithm is dominated by the $O(nlog\ n)$ computational expense of the forward and inverse discrete Fourier transforms (DFT).

This algorithm implements cyclic convolution without any explicit modification of the input signal. The forward and inverse DFT algorithms treat the input signal as an infinitely repeating cycle of itself, so the frequency domain representation that is produced is actually a representation of an infinitely repeating cycle of the elements of the input signal, and not a representation of a finite sequence of these elements. This behaviour allows convolution with a cyclic border condition to be performed without preprocessing the input image or increasing the memory requirements. Algebraically, this is defined as:

$$Image_{(k,l)} * PaddedPSF_{(u,v)})_{(x,y)} = \mathcal{F}^{-1}\Big(\mathcal{F}(Image_{(x,y)})\mathcal{F}(PaddedPSF_{(x,y)})\Big)$$

$$where$$

$$u < U,\ v < V \tag{3.5}$$

$$PaddedPSF_{(u,v)} = \begin{cases} PSF_{(u,v)} & : u < U,\ v < V \\ 0 & : u > U,\ v > V \end{cases}$$

which corresponds to the following pseudo-code description:

```
for each pixel (i,j) in padded_psf
  padded_psf(i,j) = 0
end for

for each pixel (u,v) in psf
  padded_psf(u,v) = psf(u,v)
end for

input_fourier = forward_dft(input_image)
padded_fourier = forward_dft(padded_psf)
```

```
for each pixel (i,j) in input_fourier
  output_fourier(i,j) = input_fourier(i,j) * padded_fourier(i,j)
end for

output_image = inverse_dft(output_fourier)
```

The runtime behaviour of this formulation is dominated by the $O(nlog\ n)$ runtime behaviour of the discrete Fourier transform algorithm [122][123]. Since the PSF image is zero padded to match the size of the input image, the amount of data that must be transformed by the DFT is solely dependent on the size of the input image. For an $N \times M$ pixel input image, the convolution operation consists of 3 discrete Fourier transforms, each with a runtime proportional to $M(Nlog\ N) + N(Mlog\ M)$ For a sufficiently parallel machine, or a sufficiently small size of input image, two of these DFT operations can be performed simultaneously, the third operation can only be performed after both the other two have completed.

Achieving any performance improvement through parallelisation is also dependent on how well the DFT algorithm responds to an increased number of parallel data streams. For an $M \times N$ 2D transform, the speed-up is more linear, as each 1D row or column can be transformed in parallel. For a sufficiently parallel machine, or a sufficiently small size of input image, this results in a transform roughly proportional to $\frac{M}{P}(Nlog\ N) + \frac{N}{P}(Mlog\ M)$ where $P$ is the number of parallel processors.

While the zero padding of the kernel signal significantly raises the amount of data that must be processed in order to perform the convolution, by using this Fourier technique in combination with the divide and conquer convolution algorithms described below, the effect of this can be reduced.

## 3.5 Discrete Convolution defined recursively

These definitions define the convolution operation in terms of itself, and are used in a divide and conquer approach to break a large deconvolution operation down into a set of smaller deconvolution problems. They form the basis of the out-of-core convolution algorithms, the parallel convolution algorithms, and the algorithm for convolution with a spatially variant PSF [124] which extends spatially invariant deconvolution to the problem of spatially variant deconvolution with a local-invariance assumption. This is a common approach for accelerating convolution performed in the frequency domain [125], as it trades transforming a large amount of data to a large number of independent transformations of a small amount of data.

The two image tiling techniques described in Sections 3.6 and 3.7 are exactly equivalent in both runtimes and memory demands for serial execution without considering the performance of the cache hierarchy. When executed in parallel, the operations in the final stages of the Overlap-Add method involve writing to a shared resource, which will result in either the intermittent forced serialisation of the parallel threads when two or more threads attempt to add their values to the same pixel in the destination tile, or require a more involved algorithm for scheduling the convolution of each individual tile in order to avoid such a scenario. When considering the behaviour with respect to a writeback memory hierarchy, where the cost of writing to the cache is greater than the cost of reading from the cache, as writes to the cache must be immediately committed to main memory, the Overlap-Add approach's requirement of multiple threads writing to the same pixel means that there will be more than $M \times N$ pixel writes for processing an $M \times N$ pixel image. The Overlap-Save approach has more than $M \times N$ pixel reads for processing an $M \times N$ pixel image, and produces exactly $M \times N$ pixel writes.

These algorithms can also be stacked on top of each other, so an Overlap-Save convolution can delegate the convolution of each tile to an inner Overlap-Save convolution—though this amplifies the amount of data from overlapping tile borders that the algorithm must process, and so is only useful in exceptional circumstances. In general, the image should be divided into the smallest sized tiles that will be required all at once.

## 3.6 Overlap-Add Image Tiling

The overlap-add [69] method breaks a single convolution operation down into a set of zero-padded convolution operations of smaller signals, which are convolved individually, and then recombined into a single signal that is identical to the convolution of the original input signal with the kernel.

Algebraically, this is defined as:

$$\begin{aligned} (Tile_{(i,j)(k,l)} * PSF_{(u,v)})_{(x,y)} &= \\ \left(\sum_{i=0}^{I}\sum_{j=0}^{J} Tile_{(i,j)(x,y)}\right) * PSF_{u,v} &= \sum_{i=0}^{I}\sum_{j=0}^{J}\left(Tile_{(i,j)(x,y)} * PSF_{u,v}\right) \end{aligned} \tag{3.6}$$

$$where$$

$$Tile_{(i,j)(x,y)} = \begin{cases} Image_{(x,y)} & : \frac{(i-1)x}{I} < x < \frac{ix}{I},\ \frac{(j-1)x}{I} y < \frac{jy}{J} \\ 0 & : otherwise \end{cases}$$

$$0 < u < U,\ 0 < v < V$$
$$0 < i < I,\ 0 < j < J$$
$$2U < I,\ 2V < J$$

Which corresponds to the following pseudo-code description:

```
divide the input image into N rectangular tiles, numbered 0 through to N
create N copies of the input image, numbered 0 through to N
for each number N
 in image number N, set all pixels of all tiles except tile number N to 0
for each image N
 convolve image N with the kernel
for each pixel(i,j) in the output image
 pixel(i,j) = sum of pixel(i,j) of each tile
```

However, implementing this algorithm exactly as above will neither reduce the computational expense nor reduce the memory demands of the convolution. Instead the algorithm is sensibly implemented in a way that takes advantage of the identity $zero_{M\times N} * kernel_{O\times P} = zero_{M\times N}$, which means that each tile only needs a border of zeros around it the width and height of the kernel image, since all pixels beyond that region will remain 0 after the convolution operation is performed.

After this modification, this divide and conquer approach performs the convolution in $O(i\frac{n}{j}log\ \frac{n}{j})$, where $i$ is the number of tiles that are produced, $j$ the ratio of the tile size to the input signal size, and $n$ the size of the input signal. Note that $j > \frac{n}{i}$ due to the overlapping border region of each tile.

## 3.7 Overlap-Save Image Tiling

The Overlap-Save [69] method is a minor modification of the Overlap-Add technique that removes the need for each independent thread to write to a shared resource, and restricts the amount of output data that was written, in exchange for increasing the amount of input data that must be read.

The algorithm utilizes a scatter and gather function to divide the input image into tiles, and combine the tiles into the output image.

The scatter function divides the input image into a set of regions of interest, and copies each region of interest, along with a border region as wide and as tall as the PSF

image, into a tile. This border region overlaps with the neighbouring regions of interest, and will be disregarded by the gather function. Consequently, this border should be relatively small compared to the size of the region of interest for algorithm's runtime not to be dominated by the calculation of the overlapping region. The tile is then convolved with the PSF.

The gather function copies the region of interest from the tile into the appropriate location in the output image. By favouring overlapping reads during the scatter rather than overlapping write during the gather, this approach can avail of course grained parallelism at the per tile level. How available computational cores are divided between this outer algorithm and the inner convolution algorithm will be discussed in more detail in chapter 5.

## 3.8 Convolution with a Spatially Variant PSF

The above four algorithms demonstrated a number of ways of performing the same operation: straight forward 2D convolution of an input signal with a PSF or kernel signal. This mathematical operation corresponds to the impulse response of a linear system, such as a shock wave's transmission through a material, or light rays through a lens. For situations where the impulse response of a system is invariant with respect to other parameters, such as time or position, convolution is sufficient to model the behaviour of the system. However, the algorithm must be adapted in order to be applied to situations where the impulse response varies with one or more parameters, such as a system with a spatially variant PSF [124].

The most pedantic approach to performing convolution with a spatially variant PSF is to either calculate or measure a PSF image for each pixel of the input image, and process the input image using the following pseudo code:

```
for each pixel (i,j) in output_image
  output_image(i,j) = 0
end for

for each pixel(i,j) in input_image
  calculate psf_set(i,j).psf_image
end for

for each pixel (i,j) in output_image
  for each pixel (u,v) in psf_set(i,j).psf_image
```

```
    output_image(i,j) += input_image(i-u,j-v) *
                         psf_set(i,j).psf_image(u,v)
  end for
end for
```

Specifying a unique PSF image for each pixel is uncommon in practice. It requires $M \times N$ unique PSF buffers to be present in memory, and can only be implemented via the $O(n^2)$ spatial convolution algorithm.

The Overlap-Add or Overlap-Save convolution algorithms provide a workable compromise between pixel-perfect accuracy and low resource/runtime costs. By measuring or calculating the PSF for each tile in the divide and conquer convolution algorithm the more economical $O(nlog\ n)$ Fourier convolution algorithm can be used, along with a drastically reduced number of PSF image buffers. Using this technique, the sum total of space required for the PSF image buffers will be less than or equal to the size of the input image—an acceptable level of overhead, since the overall memory footprint remains dependent on just the input image.

The following pseudo-code describes an approximate spatially variant convolution algorithm built over a Fourier convolution routine and the Overlap-Save divide and conquer technique:

```
for each (i,j) in tileset
  input_tile = input_image.get_tile(i,j)
  output_tile = FourierConvolve(input_tile, psf_set(i,j).image)
  output_image.put_tile(i,j, output_tile)
end for
```

where `psf_set(i,j)` is an array of PSF images, and `get_tile()` and `put_tile()` are functions for copying tiles from the input image to the output image, in accordance with either Overlap-Add or Overlap-Save semantics. The algorithm can be parallelised using the same technique as the parallelised Overlap-Add or Overlap-Save convolution, with the same increase in memory requirements and processing overhead.

## 3.9 Richardson Lucy Algorithms

Building the plain Richardson Lucy deconvolution algorithm (RL) or wavelet regularised variant (WRL) on top of these convolution algorithms is a straight forward task, since

the bulk of the complexity due to parallelism and optimisation is handled by the above convolution techniques.

As described in Chapter 2, the RL algorithm is the following:

$$e_{n+1} = e_n \left[ \left( \frac{o}{e_n \ * \ psf} \right) \star psf \right] \tag{3.7}$$

where

$$e_0 = o \ * \ psf$$

Which corresponds to the following pseudo-code description:

```
est_image = convolution(input_image,psf_image)

for n = 1 to N
  tmp_image = convolution(est_image,psf_image)

  for each pixel (i,j) in input_image
    tmp_image(i,j) = input_image(i,j) / tmp_image(i,j)
  end for

  tmp_image = correlation(tmp_image,psf_image)

  for each pixel(i,j) in est_image
    est_image(i,j) = est_image(i,j) * tmp_image(i,j)
  end for
end for
```

The RL algorithm is the same for spatially variant convolution, with the exception that the `psf_image` becomes the `psf_set` array, and the convolution function call becomes a call to a spatially variant convolution function.

## 3.10 Parallel RL for cluster-based computing

The above algorithm must synchronise each thread of execution four times per iteration: after the convolution operation, after the pixel-by-pixel division operation, after the cross-correlation operation, and after the pixel-by-pixel multiplication operation. For a GPU or multicore CPU, these synchronisations are inexpensive, and do not impact on the performance of the algorithm regardless of the size of the input or PSF images.

The mapping, reduction, and synchronisation of multiple threads of execution on a computation cluster is more costly, however, as it requires the image data to be copied over the cluster's internal network to and from the worker nodes. In these situations, the RL algorithm is modified to reduce the number of per-iteration synchronisation operations in favour of a single scatter and gather operation as the first and last steps of the algorithm. This approach is a trade off between runtime costs and algorithm correctness, and is prone to introducing tiling artifacts in the deconvolved image after many iterations, though for a small number of iterations these effects are largely undetectable.

Constructed from the Overlap-Save convolution algorithm, this modified algorithm is described by the following pseudo-code:

```
for each tile(i,j) in input_image
  est_tile = tile(i,j) * psf_image
  for n = 0 to N
    tmp_tile = convolution(est_tile,psf_image)
    for each pixel(u,v) in tile(i,j)
      tmp_tile(u,v) = tile(u,v) / tmp_tile(u,v)
    end for
    tmp_tile = correlation(tmp_tile,psf_image)
    for each pixel(u,v) in tmp_tile(u,v)
      est_tile(u,v) = tmp_tile(u,v) * est_tile(u,v)
  end for
end for
```

Where the convolution function is the serial or parallel Fourier convolution algorithm, or even an inner Overlap-Save convolution. The cost of the overlapping border calculations being offset in this exceptional case by sending larger tiles with a comparatively smaller overlapping border over the network.

With this formulation, the outer loop can be distributed across many nodes in a cluster without incurring any per-iteration communication overhead. If executed on a

multicore node, parallelised Fourier convolution and image arithmetic in the inner loops still have the same synchronisation requirements, but with a constant, rather than $O(n)$, runtime expense.

### 3.10.1 Richardson Lucy Border Conditions

As the RL algorithm involves two convolution operations, performed sequentially, the 'reach' of a tiled implementation of this algorithm is double that of a tiled convolution or cross-correlation algorithm. Consequently, the image to be deconvolved must be zero-padded with twice the width and twice the height of the PSF image.

The hard threshold that exists between the image tile and the zero padding will produce ringing artifacts if processed as-is. One approach to reduce these artifacts is to rely on the regularisation step, another is to modify the tile's image data to resemble how the tile would look if it were a standalone image captured by the optical system [121]. In the second case, the edge pixels of the image would be blurred into the zero-valued border pixels according to the PSF, similarly to how the pixels of the acquired image are blurred into each other. This can be achieved by rescaling the pixels on either side of the threshold by the normalised PSF pixel values—an inexpensive operation that must only be performed once, and can be parallelised down to the granularity of each individual border pixel.

This preprocessing is necessary for avoiding tiling artifacts in the spatially variant deconvolution operations, and the Topdown and Streaming strategy deconvolution strategies described in 5.

### 3.10.2 Parallelising The Blind Richardson-Lucy Algorithm

The blind RL algorithm discussed in 2.12 contains two independent RL operations per loop, with each iteration of the loop depending on the results of the previous iteration. Each of the RL operations can be transparently parallelised internally according to any strategy, but the opportunity exists to perform the image estimation and PSF estimation steps in parallel with each other as well. As this approach requires only one synchronisation per loop, it provides a potential $\times 2$ speedup for a Dual-Core processor over a Single-Core processor . For GPU based deconvolution, the larger memory requirements of the blind deconvolution algorithm mean it will be less capable of taking advantage of the Loop-Level parallelism—but the opportunity to assign one of the loop-level parallel operations to the CPU and one of the loop-level parallel operations to the GPU naturally presents itself.

# Chapter 4

# The Wavelet Transform Library for the GPU and CPU

## 4.1 Wavelet Regularised Richardson Lucy

The wavelet-regularised Richardson-Lucy algorithm (WRL) is a modification of the RL algorithm with superior image restoration abilities in the presence of high levels of noise in the input image [99]. The IMPAIR library provides an implementation of this variant algorithm, via a custom Discrete Wavelet Transform (DWT) library implemented in ISO C99 with OpenMP for the multicore CPU, and CUDA for the GPU. The DWT Library provides forward and inverse transforms for the Daubechie family of wavelets, implemented via the lifting scheme [126] as well as API calls for performing wavelet transforms with custom filter coefficients.

The DWT library provides two wavelet shrinking operations: *k-sigma clipping* and Donoho's *universal thresholding* algorithm—an adaptive denoising algorithm that requires no external parameters other than the input image [127].

This chapter details the implementation and performance of Multicore CPU and GPU variations of a discrete wavelet transform library and wavelet shrinking library for use by IMPAIR. The GPU implementations outperform the CPU implementations in all cases.

## 4.2 Daubechie Wavelets

The Daubechie "family" of wavelets [126] is a set of related high-pass and low-pass finite-impulse-response filters, that are applied at descending scales to a signal in order to

provide a wavelet-domain representation of the signal. For the discrete wavelet transform, the low-pass filter and the high-pass filter of the Haar wavelet are related to one another by a simple mechanism: the low-pass filter's values are the absolute values of the highpass filter, in reverse order.

This chapter discusses the Daub-2, or Haar, and Daub-8 wavelet transforms. Images of the high and low pass wavelet filters used by IMPAIR are shown in Figure 4.1.

## 4.3 Algorithm Overview

This section discusses the *decimating* wavelet transform [128], so called because with each level of the transform the signal length is reduced by half, resulting in the transformed image having precisely the same size as the untransformed image. The original approach to the wavelet transform, the *a trous* transform [128], does not have this memory-conserving property, and so is not implemented by IMPAIR.

**1D DWT**

The forward DWT algorithm is composed of two steps: the filtering step, which generates the high and low frequency representations of the input signal with respect to a wavelet filter, and the recursive step, which performs a DWT on the the low frequency representation of the input image. This produces a set of signals which represent the input signal at various scales or resolutions .

The inverse DWT algorithm iteratively recombines the low and high frequency representations of the input signal, beginning with the high and low frequency signal pairs for the lowest scale, and introducing the higher frequency signals as each low frequency representation is reconstructed. This results in a runtime complexity of $\theta(n)$ for the forward and inverse 1D DWTs for signals consisting of $n$ samples.

As these iterative stages are dependent on the output of the previous stage, they cannot be parallelised in an SIMD fashion (though they are amenable to pipelining parallelism, to a limited degree). Effective parallelisation of the 1D DWT transforms hang on parallelising the internal operations within each iteration.

Each individual level of the DWT can be parallelised over $n$ processors for a signal of $n$ samples. Consequently, if the number of available processors exceeds the number of samples, a single level of the DWT has a runtime complexity of $\theta(1)$. As $log\ n$ iterations of this single level of the DWT must be performed in sequence in order to perform the full transform, the full DWT has a runtime complexity of $\theta(log\ n)$ in this case.

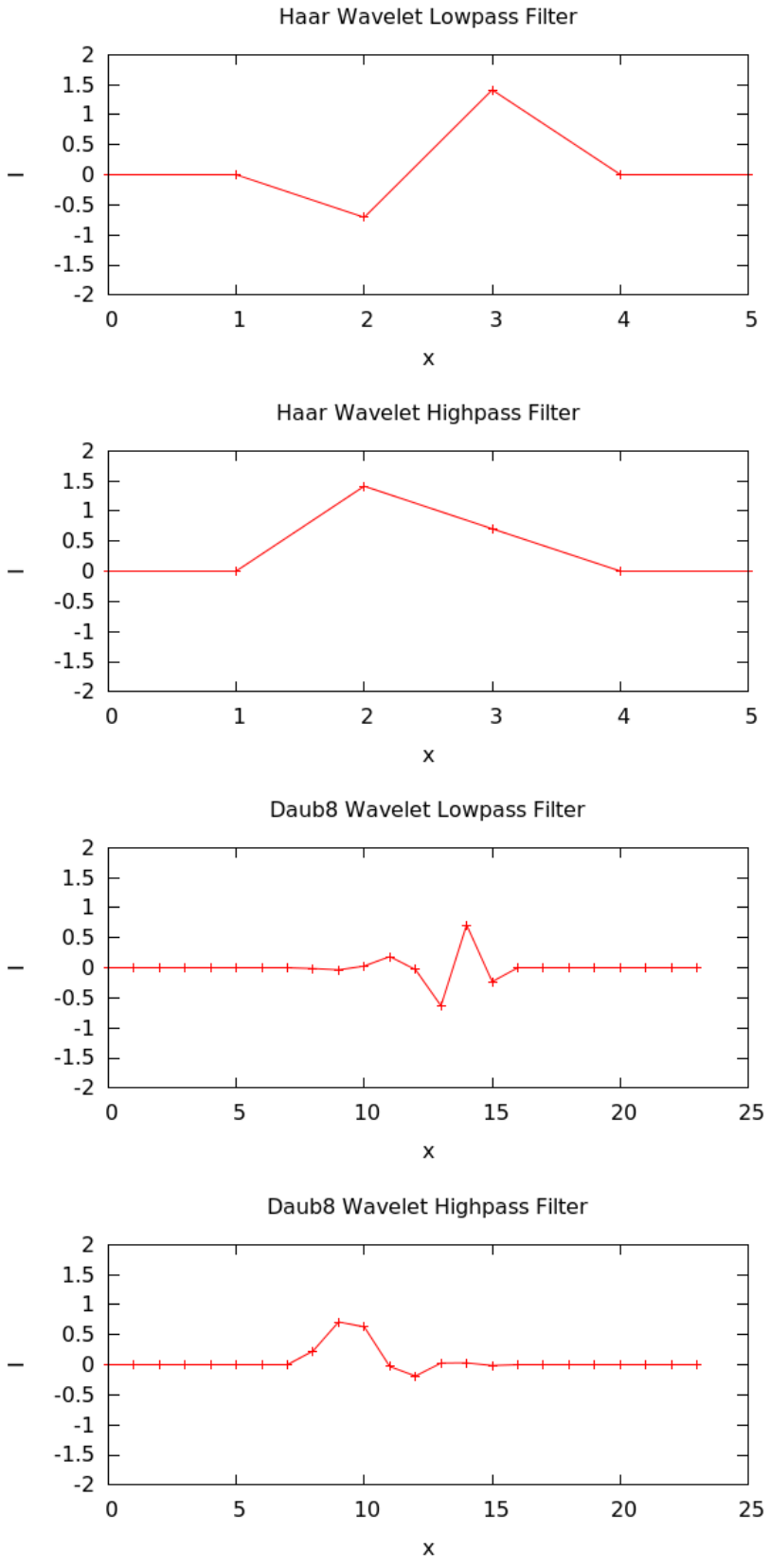



FIGURE 4.1: Example Wavelet Filters.

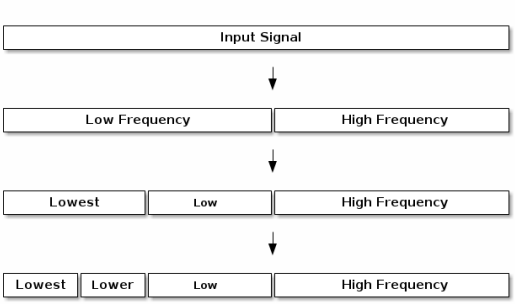



FIGURE 4.2: The One Dimensional Discrete Wavelet Transform

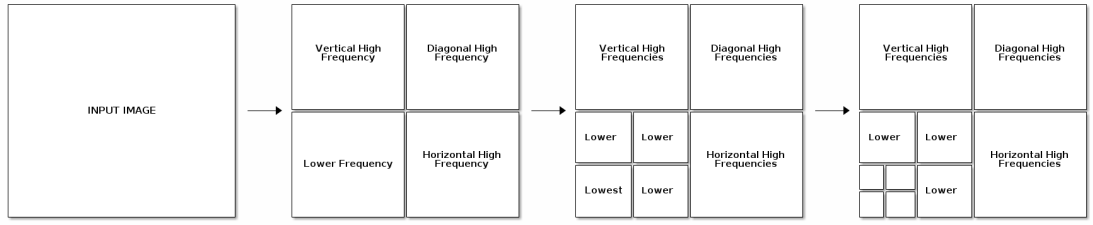



FIGURE 4.3: The Two Dimensional Discrete Wavelet Transform

If the number of available processors $p$ does not exceed the number of samples $n$ in the signal, a single level of the DWT will have a runtime complexity of $\theta(\frac{n}{p})$. The runtime complexity of the full DWT involves $log\ n$ single levels on successively smaller signals, which will lower the number of samples to beneath the number of available processors after $(log\ n - log\ p)$ iterations. This gives a runtime complexity of $\theta(\frac{n}{p})$ where $n > p$, and a runtime complexity of $\theta(log\ p)$ where $n \leq p$.

**2D DWT**

The 2D wavelet follows the same gross structure, a single-level 2D DWT applied iteratively to the low frequency output of previous 2D DWT, and similarly, the parallelisation of the 2D DWT must be done within each iteration.

Each iteration of the 2D DWT—known as a level—corresponds to a single-level 1D DWT of each row in the image, and a single-level 1D DWT of each column of the resulting image. This produces an image of four quadrants, three of which contain the high frequency wavelet coefficients of the input signal, and one containing the low frequency coefficients. As with the 1D transform, the low frequency quadrant is then transformed with a 2D DWT, producing a series of lower and lower resolution representations of the input signal.

The two dimensional nature of this algorithm does not introduce significant runtime expenses over a batch of 1D transforms, and so has a similar runtime proportional to $nlog\ n$ and can be parallelised to result in a runtime proportional to $\frac{n}{c} log\ n$, where $c$ is

the number of parallel threads. For a sufficiently parallel machine, or a sufficiently small input image, runtimes proportional to $log\ n$ can be achieved.

IMPAIR provides two alternative approaches to performing the 2D DWT operation in terms of batches of 1D DWT of rows or columns of the input image: transpose-based DWT and interleaved-batch DWT.

The transpose-based DWT performs batches of 1D DWT operations on a contiguous vector of vectors, corresponding to a vector of rows of an image. Since each 1D DWT requires access to the neighbouring samples of each vector, this results in memory access pattern where all out-of-order reads are restricted to being within a sliding window of fixed length, regardless of the length of each row, which will coalesce into a single memory access to serve all threads in the warp. In order to perform the DWT of the image columns, the image is transposed and the same vector of rows algorithm is performed. This introduces unavoidable latency to the 2D DWT, as the transpose operation requires synchronisation between all threads and has an efficient, but not optimal, memory access pattern.

The interleaved-batch DWT (introduced and described in [47] as a block-based DWT) performs batches of 1D DWT operations on an interleaved vector of vectors, corresponding to a vector of columns of an image. Since each 1D DWT requires access to the neighbouring samples of each vector, this results in worst-case cache performance if each vector is transformed completely before proceeding to the next. By incrementally transforming each additional sample of each vector in a round-robin fashion, this worst-case performance can be avoided. This results in a DWT that can be performed without requiring an intermediate transpose operation, and without suffering under a deep cache hierarchy. This strategy can be understood as the entire batch being processed as if deterministically multitasked by as many lock-step threads as there are columns in the input batch.

## 4.4 Parallelisation Strategies

The IMPAIR DWT library explores three parallelisation strategies for the 1D and 2D discrete wavelet transform: the DWT-Naive strategy, the DWT-Topdown strategy, and the DWT-Recursive strategy.

**DWT-Naive Strategy**

The DWT-Naive strategy is an iterative out-of-place operation that attempts to treat each sample in the output signal, or each pixel in the output image, as an independent entity, whose values are calculated in parallel with each other. Due to the inherent locality

of the DWT transform, this DWT-Naive approach results in cache-friendly memory access patterns if the output pixels are generated in their in-memory order. Since the strategy performs an out-of-place transform, input data can be shared between all parallel threads without any costs associated with ensuring cache consistency, and output data is restricted to being changed by only one thread.

The DWT-Naive strategy requires $log\ n$ iterations of the single-level transform to perform a full DWT. These iterations are necessarily serial operations, and require all threads to synchronise before proceeding. Due to the $log\ n$ decay in the size of the dataset being processed for each subsequent iteration, the cost of these synchronisation steps is similarly reduced.

**DWT-Topdown Strategy**

The DWT-Topdown strategy is a less general form of the the DWT-Naive strategy, that partitions the workload across all threads at the granularity of a 1D input signal. This approach has no advantage over an unparallelised DWT for the DWT of a single input signal, and is of limited advantage for calculating the 2D DWT of an extremely oblong image. Due to this limitation, a full implementation and testing of this DWT-Topdown strategy is not considered in this thesis, but is left as future work.

**DWT-Recursive Strategy**

The DWT-Recursive strategy takes a different approach. Instead of describing a 1D or 2D DWT as the iterative application of a 1-Level DWT, it composes a 1D or 2D algorithm out of the iterative application of an N-Level DWT (where N is an arbitrary value greater than the length of the wavelet filter). This approach divides the input signal or image into overlapping sub-signals or sub-images, whose size is lower-bounded by the length of the wavelet filter, and performs the fixed-level transform on the batch of tiles. The resulting signals are then scattered into the correct places in the image, so that the low-frequency pixels of each tile form the low frequency image that must be transformed by the next step.

This approach reduces the number of synchronisation steps that are required for parallelising the DWT from once every level to once every N levels, and follows a memory access pattern suited to a cache hierarchy. The processing of the input image in fixed sized tiles allows for batches of input images to be pipelined in a straightforward fashion, where unused threads can begin processing tiles from the next image as the number of threads to process the current image are halved. This is a loop-tiling approach to the discrete wavelet transform, and is employed by the Song et al implementation. A full investigation into the runtime performance of this strategy is left as future work, as the DWT-Naive strategy already provides a signficant speedup over the previous DWT transform.

## 4.5 DWT Benchmarks

Figure 4.4 shows the performance of the forward and inverse Haar wavelet transform distributed over the cores of an 8 core, hyperthreaded machine. The benchmarks were run on an 8-core i7 Intel machine, running OpenSuSE 12.3, with 16GB of RAM. The transform times scale regularly with the size of the image in pixels, and are not sensitive to the aspect ratio of the image, and the inverse transform outperforms the forward transform for all image sizes by a marginal amount. The hyperthreaded 8 core runs of the transform achieve no speedup over the 4 core runs, the low cost of the Haar wavelet.

Figure 4.5 shows the performance of the forward and inverse Daub-8 wavelet transform. The 8 core case experiences a slowdown due to hyperthreading once the image size passes a certain threshold, due to the increased cost of the Daub-8 wavelet with the same cache-friendly memory access pattern of the DWT.

Figure 4.6 shows the per-core speedup of the Haar and Daub-8 DWTs for a $4096 \times 4096$ image. The Daub-8 DWTs experience a larger boost in performance from parallelisation than the Haar DWTs, due to the higher per-pixel cost of calculating the Daub-8 DWT. For both the wavelet filters, the inverse transform experiences less speedup than the forward transform, though performing faster overall. The inverse transforms experience a comparative slowdown for the 8-core hyperthreaded configurations.

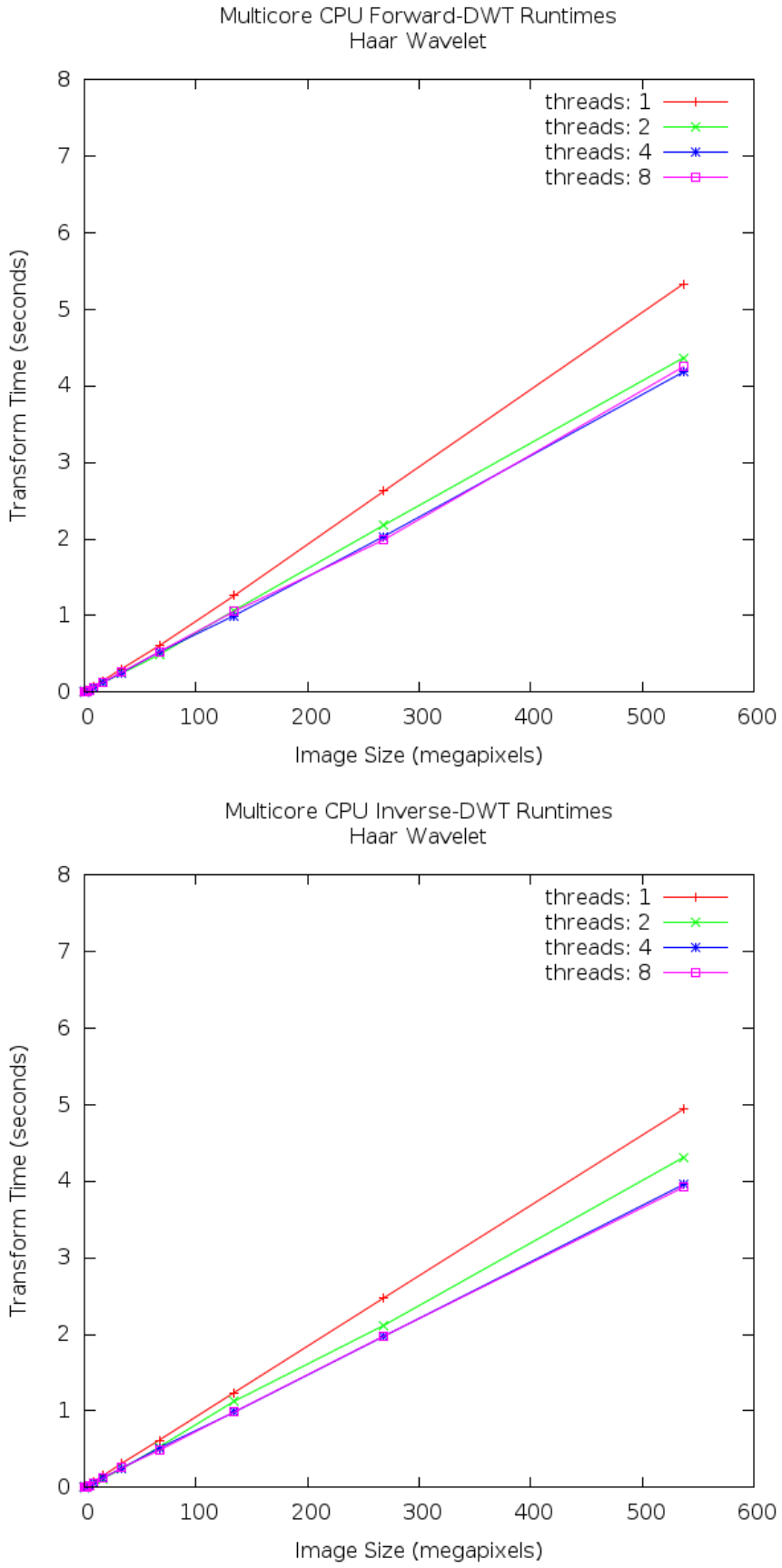



FIGURE 4.4: The forward & inverse transform run times for the Daub-2 (least computational overhead) wavelets, for a range of image sizes, running with 1 to 8 threads on an 8 core machine

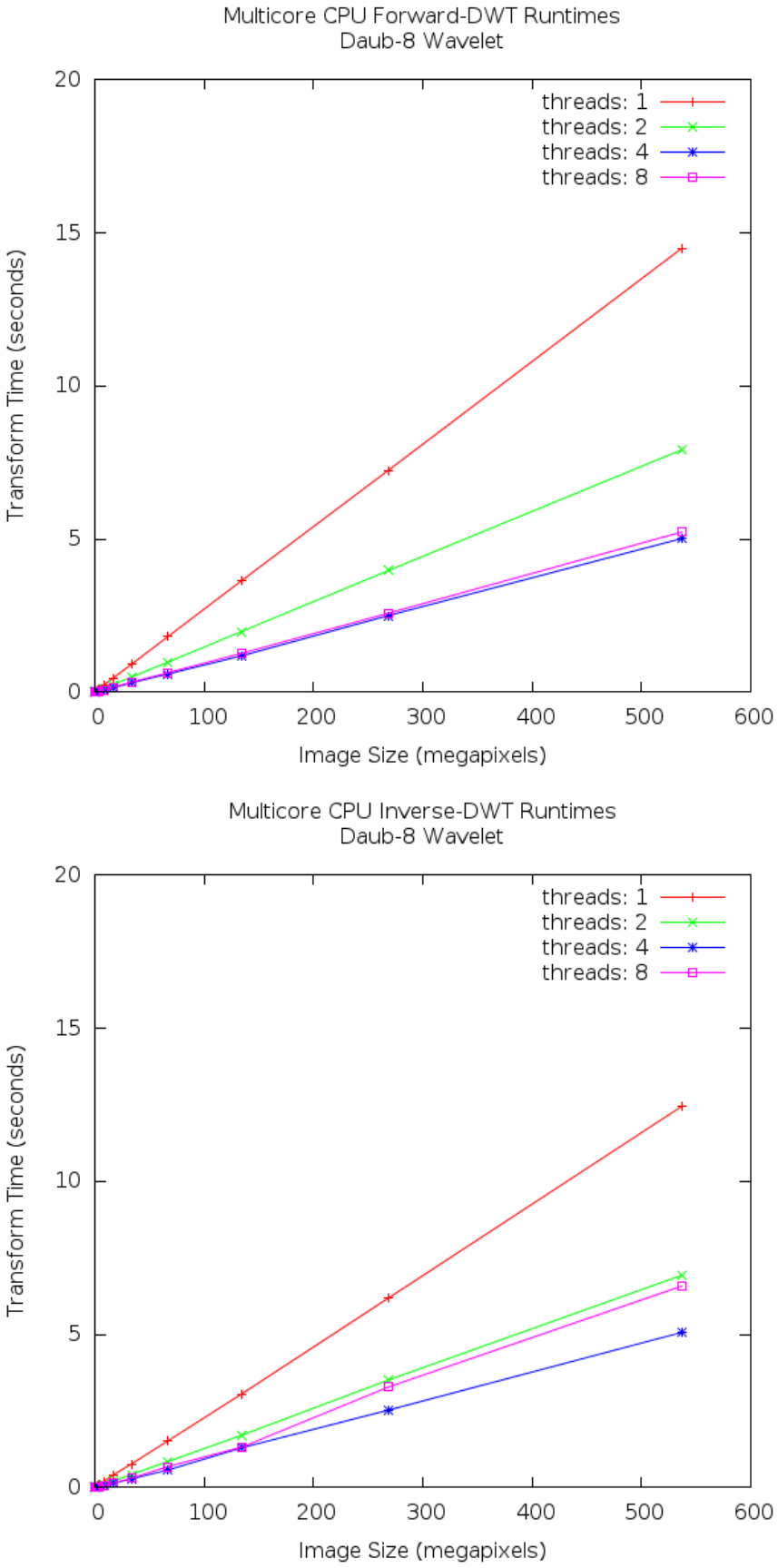



FIGURE 4.5: The forward & inverse transform run times for the Daub-8 (most computational overhead) wavelets, for a range of image sizes, running with 1 to 8 threads on an 8 core machine

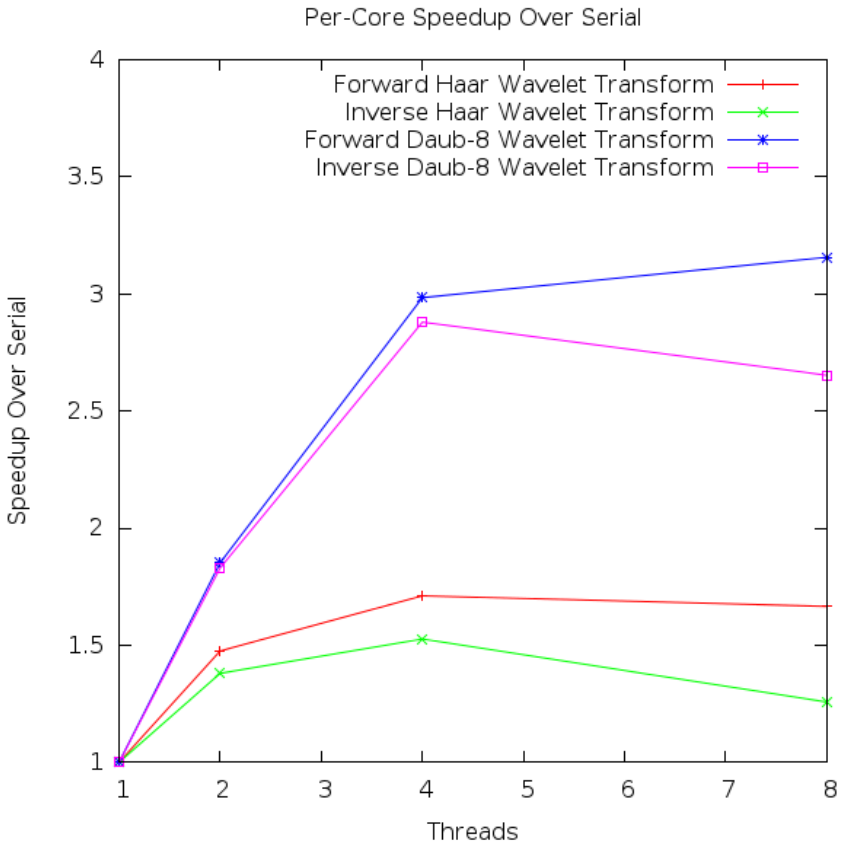



FIGURE 4.6: Speedup chart for Daub-2 and Daub-8 forward and inverse wavelet transforms on the multicore CPU.

Figure 4.7 shows the performance of a forward wavelet transform running on an NVIDIA GeForce 660 GPU, on an OpenSuSE 12.3 machine. As with the CPU performance, the inverse transform out-performs the forward transform. The extent to which this occurs is more subject to the length of the wavelet filter for these GPU transforms than it is for the CPU transforms.

Figure 4.8 shows the comparative performance of the forward wavelet transform on the CPU and GPU. The GPU transform's greater sensitivity to the length of the wavelet filter results in the parallelised CPU Daub-8 transform achieving a comparative speedup over the CPU Haar transform.

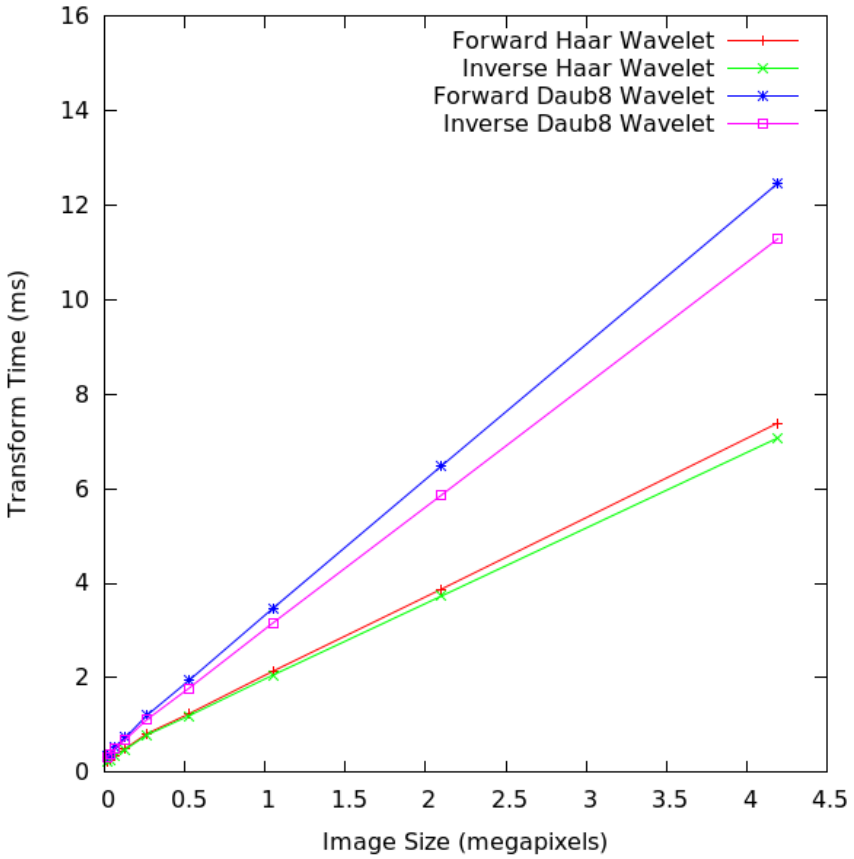



FIGURE 4.7: The forward & inverse transform run times for the Haar & Daub-8 wavelets, for a range of image sizes, on the GPU

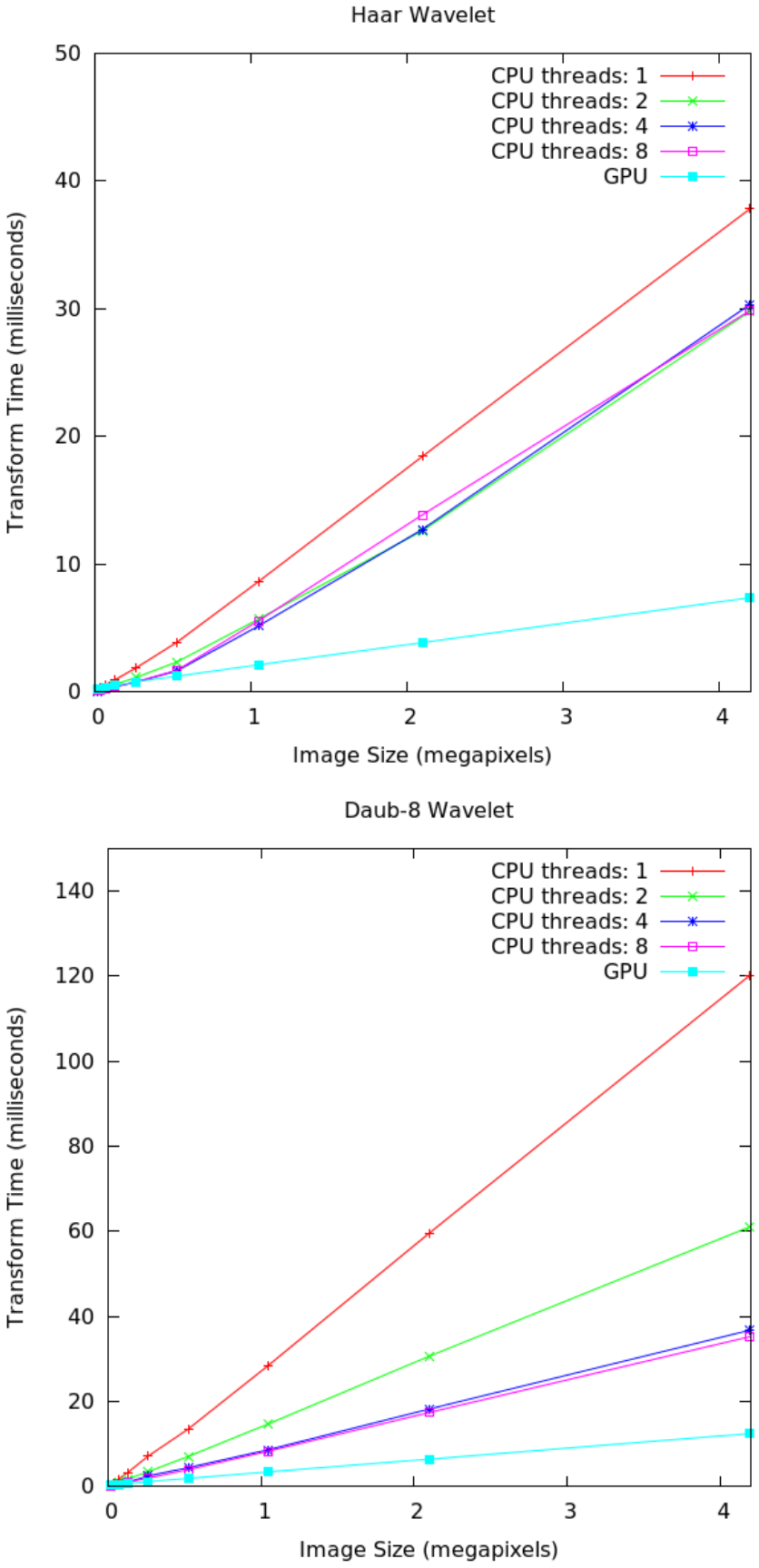



FIGURE 4.8: GPU and CPU Forward DWT Runtimes

## 4.6 Wavelet Shrinking Benchmarks

As the wavelet shrinking operation is employed as the regularisation step of the Richardson Lucy algorithm, the performance of these operations in isolation will give the minimum extra cost of the wavelet regularised Richardson Lucy over the unregularised algorithm. This section presents a description of the wavelet shrinking denoising algorithm implemented by the IMPAIR software, and gives a set of runtime benchmarks for the performance of this algorithm on the GPU and CPU platforms. For the CPU wavelet shrinking operations, we provide runtimes for image sizes up to 500 megapixels. This is significantly larger than the maximum image size profiled for deconvolution on the CPU in Chapter 5. For the GPU, we present image sizes up to 4 megapixels, as a 4 megapixel image is the maximum tile size that the GPU deconvolution will employ. Future work will include extending the GPU wavelet shrinking algorithm to scale to 100 megapixel images, using the streaming mechanism employed by the GPU deconvolution operations. We then present the comparative performance of the GPU and CPU for the 4 megapixel tile-sized images, showing the comparative speedup of the CPU's Daub-8 shrinking operation over its Haar shrinking operation compared to the GPU.

Denoising approaches like Gaussian smoothing as described in Chapter 2 are susceptible to classifying any high-frequency variation of an image as noise content. This behaviour leads to high-frequency content of the underlying signal—such as edges in an image—experiencing notable degradation as a result of the denoising process. Denoising approaches that seek to avoid this unwelcome side effect are referred to as *edge-preserving* denoising algorithms [89].

Wavelet Shrinking is an approach to denoising that takes advantage of how the wavelet transform separates high-frequency spatial variations in a signal—typically additive noise—from variations with lower frequency components—typically edges. By separating the noise from the image in this fashion the high frequency coefficients can be selectively erased or reduced, rather than distributed into the lower frequency coefficients of the signal as is done by smoothing operations.

The ability of the wavelet transform to automatically separate the noise from the intended signal in the input data allows for adaptive denoising procedures that do not require an initial estimate of the noise content of an image. IMPAIR implements one such adaptive denoising process—universal thresholding [127] —as part of the regularised deconvolution. For situations where the automatic detection of noise levels is not required, IMPAIR provides a separate denoising algorithm (K-Sigma clipping) which accepts a parameter that acts as multiplier threshold.

The Universal Thresholding algorithm is an adaptive technique for shrinking rather than smoothing noise from an input signal, and performed as follows:

- perform a high-pass wavelet filtering of the input signal

  perform a forward wavelet transform of the input signal

  wipe all co-efficients below the highest level

  perform an inverse wavelet transform of the input signal

- calculate the standard deviation of the signal samples from the signal samples' mean

- if any wavelet co-efficients surpass a value (that is a function of the standard deviation of the highpass filtered signal), decrease those co-efficients by this value.

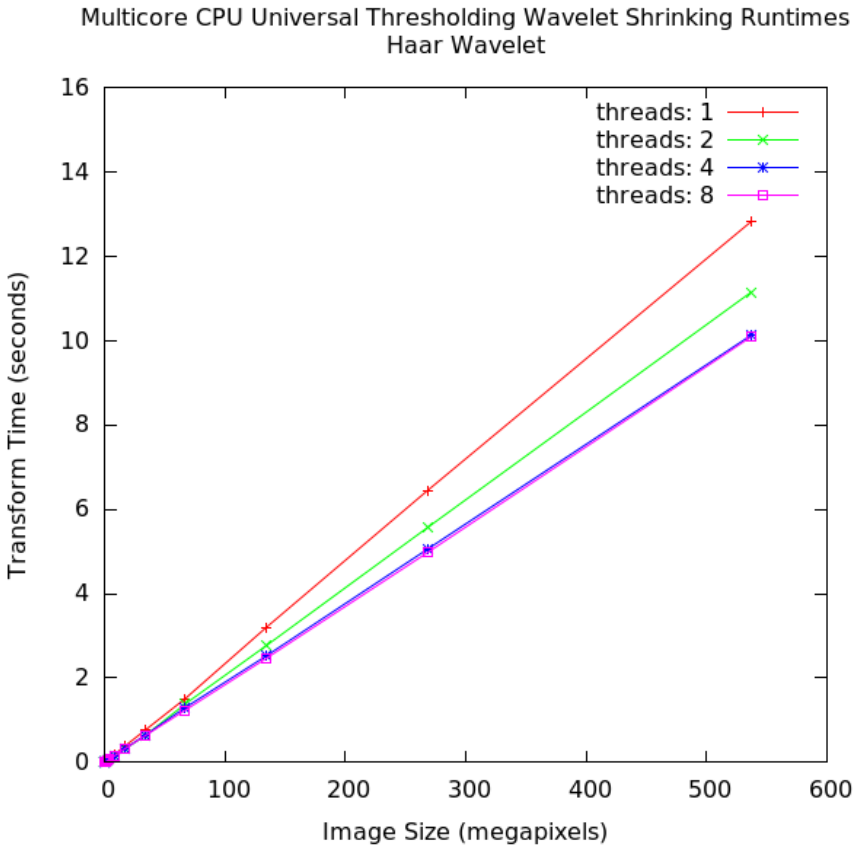
  for the universal thresholding algorithm, the threshold value is given by $\sigma\sqrt{2log(signal\_length)}$ [127]

A parallel implementation of this algorithm requires a parallel wavelet-transform operation, a parallel operation for calculating the standard-deviation, and a parallel operation to apply the thresholding formula to each wavelet coefficient. A two-pass procedure is used for the calculation of the standard-deviation, where first the mean sample value is calculated, in parallel, and then the standard deviation from the mean is calculated in parallel. For the GPU implementation, these statistics are calculated recursively, using an external memory buffer. For the CPU implementation, each parallel core calculates its share of the image sequentially.

The wavelet-shrinking library builds on the GPU and CPU DWT implementations to provide the regularisation step for IMPAIR's deconvolution operations. The runtime of the library is largely determined by the performance of the underlying forward and inverse wavelet transforms.

Figure 4.9 shows the runtimes for the Universal Thresholding wavelet shrinking operations for megapixel images. All configurations scale evenly with the image sizes, and are insensitive to the image aspect ratio. As with the Daub-8 inverse DWT, the 8 core run experiences a slowdown for image sizes above a certain threshold.

Figure 4.10 shows the per-core speedup for the Universal Thresholding wavelet shrinking operation for a $4096 \times 4096$ image. The per-core speedup behaviour of this operation is limited by the underlying inverse-wavelet transform, particularly in the case of the Haar wavelet operation, which experiences poor per-core speedup due to its minimal computational expense.



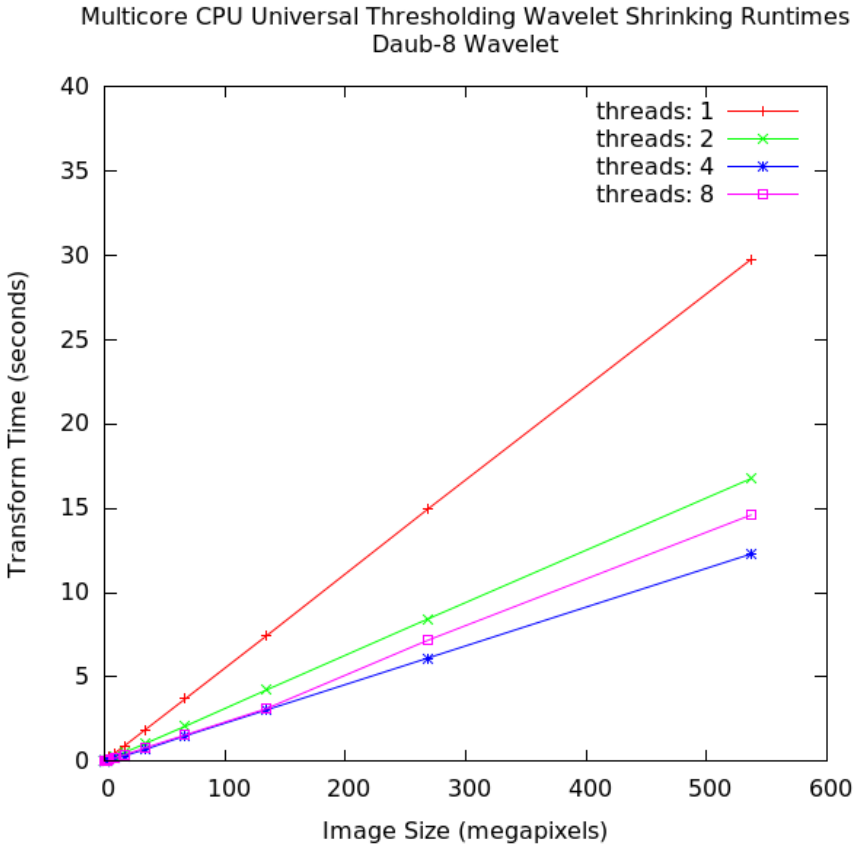




FIGURE 4.9: The universal-thresholding wavelet-shrinking run times for the Daub-2 (least computational overhead) and Daub-8 (most computational overhead) wavelets, for a range of image sizes, running with 1 to 8 threads on an 8 core machine

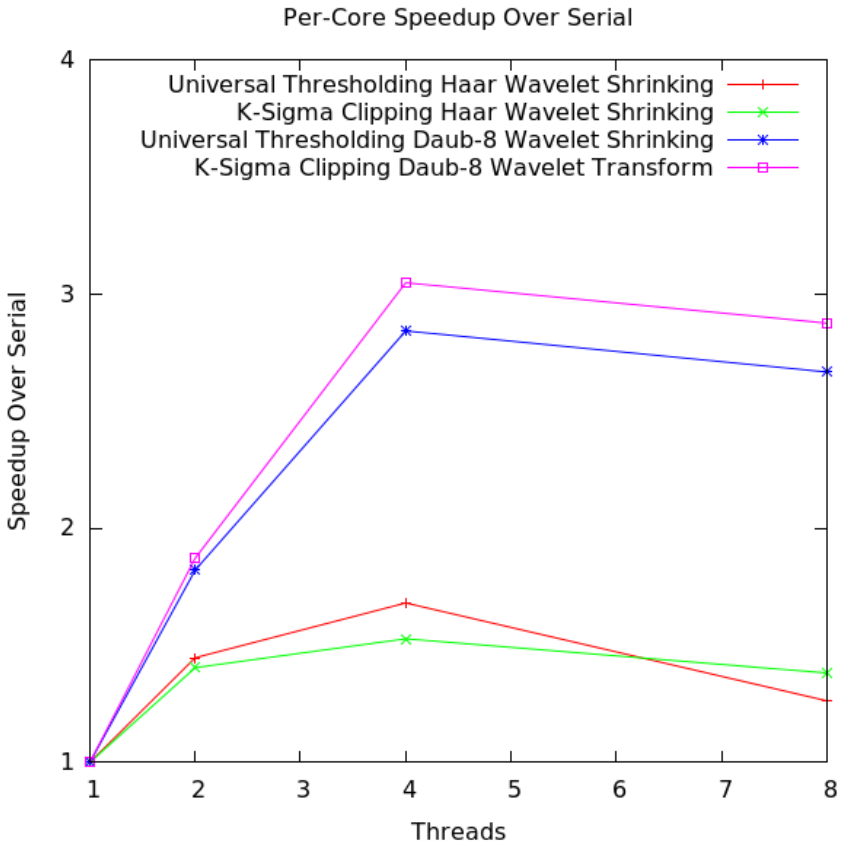



FIGURE 4.10: Per-thread speedup for the universal-thresholding wavelet-shrinking run times for the Daub-2 (least computational overhead) and Daub-8 (most computational overhead) wavelets.

Figure 4.11 shows the runtime performance of the Universal Thresholding operation for image sizes up to $2048 \times 2048$ pixels. The runtime scales regularly with the image size in pixels. The runtime is largely determined by the combined runtimes of the underlying forward and inverse DWT runtimes for images up to 8 megapixels, the wavelet-domain shrinking operation taking up a negligible proportion of the time.

Figure 4.12 shows the comparative performance for the CPU and GPU wavelet shrinking algorithms. The same as was seen in Figure 4.8 is shown, with the GPU out-performing the CPU for all threading configurations, but the GPU runtime's greater sensitivity to the length of the wavelet filter making the 4 and 8 threaded Daub-8 CPU shrinking operations comparatively faster than the CPU Haar shrinking operations.

## 4.7 Conclusion

We have implemented Multicore CPU and GPU Discrete Wavelet Transform libraries, according to the parallelisation strategy outlined by Song et al [47]. Built on top of

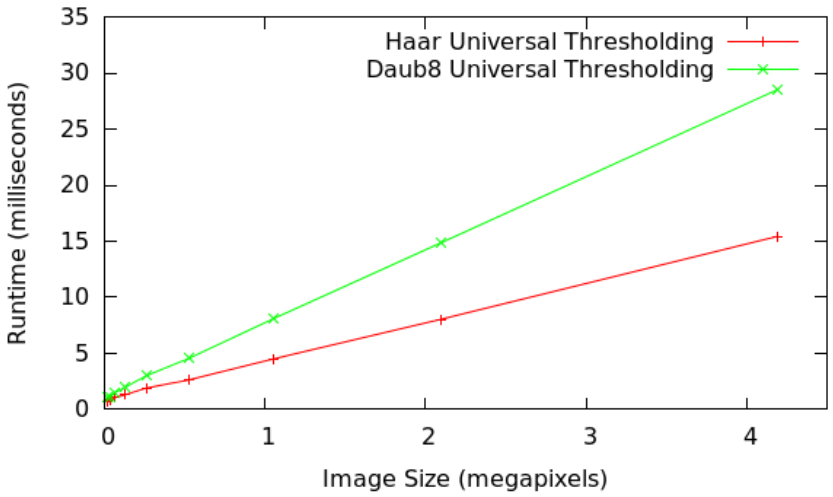



FIGURE 4.11: The GPU universal-thresholding wavelet-shrinking run times for the Haar & Daub-8 wavelets

these libraries, parallel CPU and GPU Universal Thresholding algorithms have been implemented, and their runtime behaviour measured. The runtime of the Universal Thresholding algorithm is determined by the runtime of the underlying forward and inverse Discrete Wavelet Transform algorithms. The GPU implementation of the Discrete Wavelet Transform achieves a $\times 2$—$\times 4$ speedup over the fastest Multicore CPU implementation, but the runtime is sensitive to the length of the wavelet filter. The Multicore CPU implementation runtimes have no similar sensitivity, and the runtime behaviour of this pair of implementations suggests that for a marginally larger filter, like a Daub-12 or Daub-16, the runtimes of the GPU and CPU implementations could be equivalent to one another.

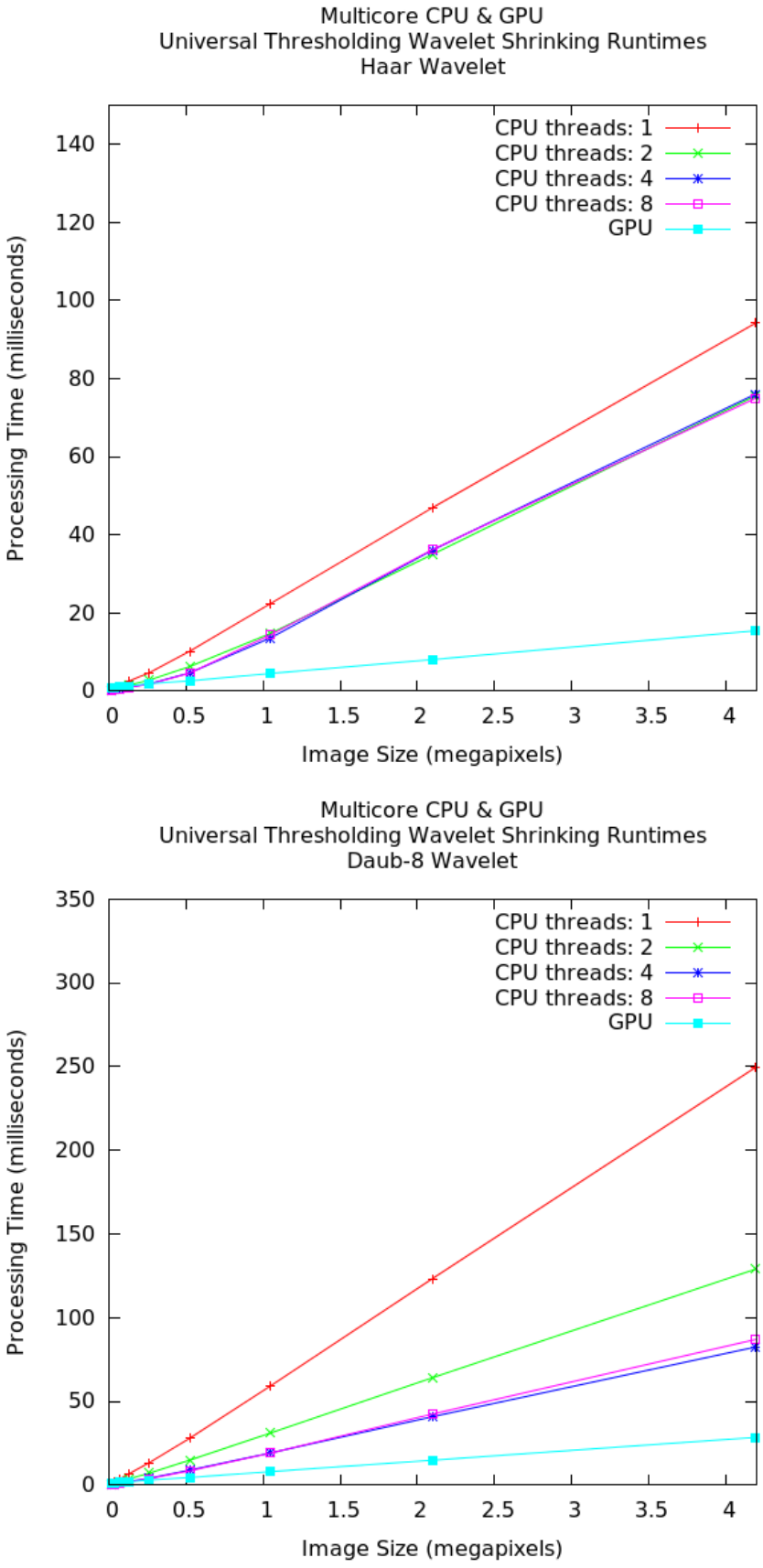



FIGURE 4.12: The Universal Thresholding processing times for both the CPU and GPU implementations, for both the Haar & Daub-8 wavelets

# Chapter 5

# The Richardson-Lucy Deconvolution Library for the GPU and CPU

## 5.1 Introduction

This chapter presents the performance of the IMPAIR software in terms of both its image restoration ability and its runtime performance. The Richardson-Lucy deconvolution algorithms and parallelisation strategies that were laid out in Chapter 3 are profiled on both the multicore CPU and the CUDA-compatible GPU. Section 5.2 illustrates the image restoration ability of both the original unregularised Richardson-Lucy algorithm (RL), and the wavelet based Richardson-Lucy algorithm with a universal-thresholding based regularisation component (WRL). Section 5.4 illustrates the runtime performance of the multicore CPU IMPAIR software. The three major parallelisation strategies (Naive, Streaming, Topdown) are compared for both the regularised and unregularised deconvolution of megapixel images. The response of each of these algorithms to the number of available processor cores is illustrated in terms of the improvement gained over a singlecore or serial deployment. Section 5.6 illustrates the runtime performance of the CUDA compatible GPU IMPAIR software. This section follows the same structure as the multicore CPU section, but does not include any speedup-over-serial measurements, due to the fixed threads-per-element ratio of the GPU threading model.

Section 5.7 presents the comparison between the GPU and CPU implementations. For the regularised and unregularised RL operations, the GPU implementations achieve an approximately $\times 10$ speedup over the Naive Multicore CPU implementations, an improvement over the GPU-CPU speedup of the wavelet shrinking algorithm shown

in Section 4.6. The fastest Multicore CPU implementations experience less than a $\times 4$ slowdown compared to the GPU for the regularised RL algorithm using all available threads, in keeping with the GPU-CPU speedup of the wavelet shrinking operation, with a particular case experiencing less than a $\times 2$ slowdown for this same reason.

## 5.2 Image Restoration Performance

For optical systems, the degree to which the amplification or deadening of the contrast ratio is present at various spatial frequencies is known as the system's Modulation Transfer Function (MTF), where it is typically used to illustrate the blurring or loss of effective resolution that the optics imposes on the output image. The restorative power of the Richardson Lucy algorithm can be illustrated in terms of the improvement it brings to the MTF of a known system.

The measurements of IMPAIR's contrast restorative strength were generated from a set of test images of a sinusoidal ramp with increasing spatial frequencies, like the ones shown in Figure 5.1.

This image is degraded by a PSF, which has the effect of reducing the contrast of the ramp as the spatial frequency increases, until the highest frequency bars become a uniform field. The degraded image is then corrupted with additive Gaussian noise, and treated as the observed output image of an optical system.

Figures 5.2, 5.3, and 5.4 illustrate the degradation of the spatial frequency ramp image with synthetic Gaussian PSF images, and restorative effects of the RL algorithm after 0, 4, 8, and 12 iterations. In this case, the image produced after 0 iterations is the initial estimate from which the RL algorithm begins to work, which contains less high frequency noise at the expense of degrading the spatial resolution of the image further.

For an exceptionally thin PSF image, like the one used to generate Figure 5.2 , perfect restoration of the input image is achieved in twelve iterations, with the contrast ratio of the highest frequency components more than doubled in only four iterations.

For broader PSF images, perfect restoration is dramatically slower, if possible at all, as shown in Figure 5.3, due to the complete flattening of the highest frequency components of the input image by the degradation process. However, all frequencies that are not utterly destroyed by the degradation process do experience a contrast restoration, for instance: the next highest frequency component is restored to the level of the initial, undegraded image. In Figure 5.4 , for an extremely broad PSF image, four of the six

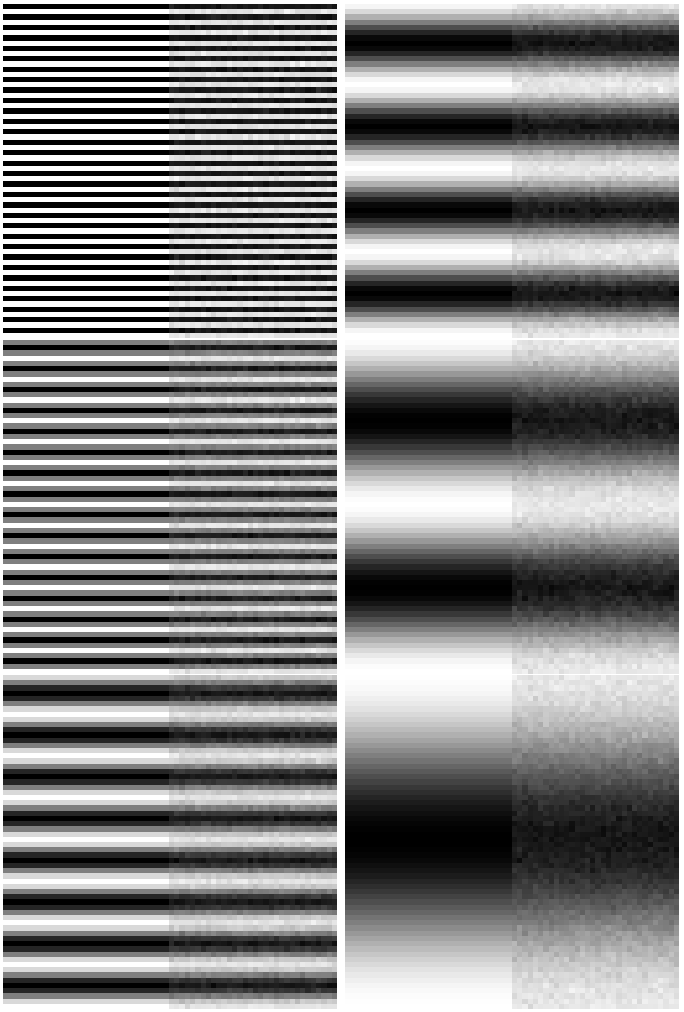


FIGURE 5.1: Spatial Frequency Test Charts. Example original and degraded testchart images, for line widths doubling from 2 to 64 pixels.

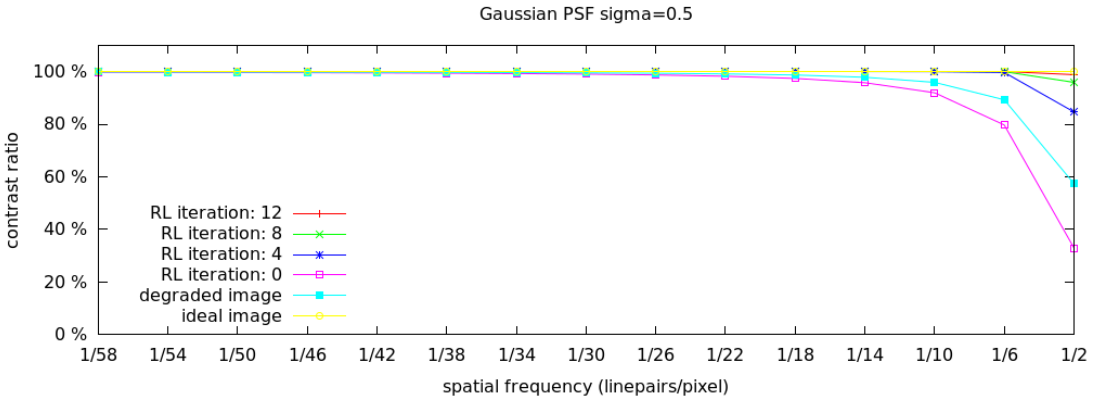



FIGURE 5.2: Noiseless Image Restoration MTF Plot

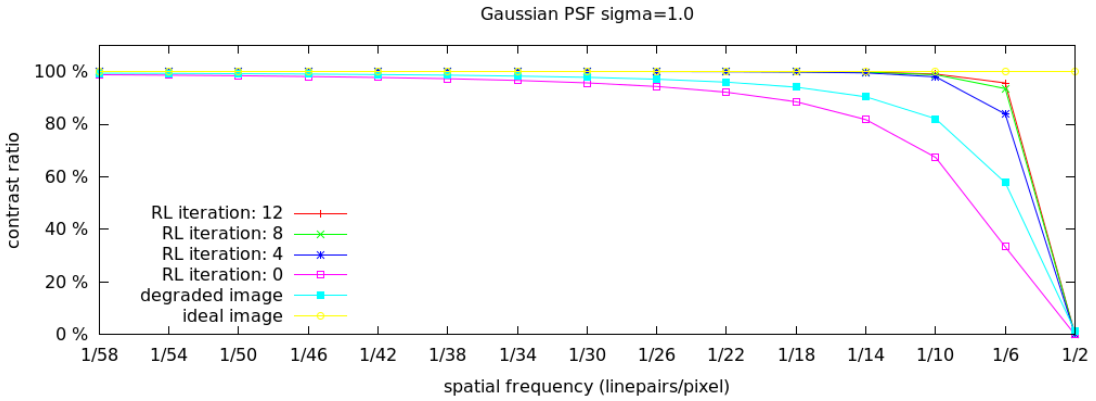



FIGURE 5.3: Noiseless Image Restoration MTF Plot

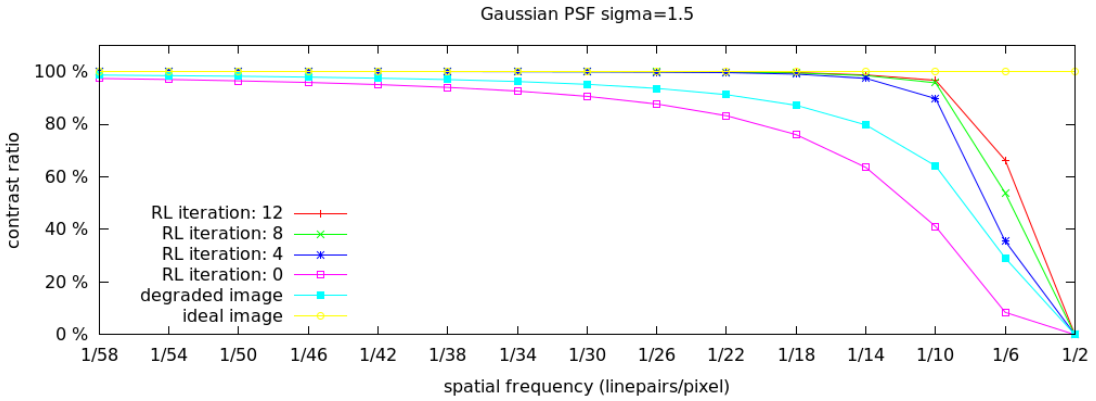



FIGURE 5.4: Noiseless Image Restoration MTF Plot

degraded frequencies are restored to their undegraded levels, and the highest frequency that has not been lost completely has its contrast ratio doubled by twelve iterations.

The following three MTF plots in Figure 5.5 illustrate the image restoration performance of the RL algorithm for test charts corrupted by varying levels of additive noise. These show the same per-pixel contrast restoration behaviour as was found with the non-noisy ramp charts for the degraded high spatial frequencies. However, a hint of the effects of the noise amplification of the RL algorithm can be seen as artifacts in the lower spatial frequencies of the MTF curve, which are perturbed slightly above and below the line of perfect contrast restoration.

In this set of MTF curves the noise amplification artifacts are noticeable, particularly for the small $\sigma = 0.5$ PSF, which is not broad enough to contribute to suppressing the noise amplification via its per-iteration smoothing effects.

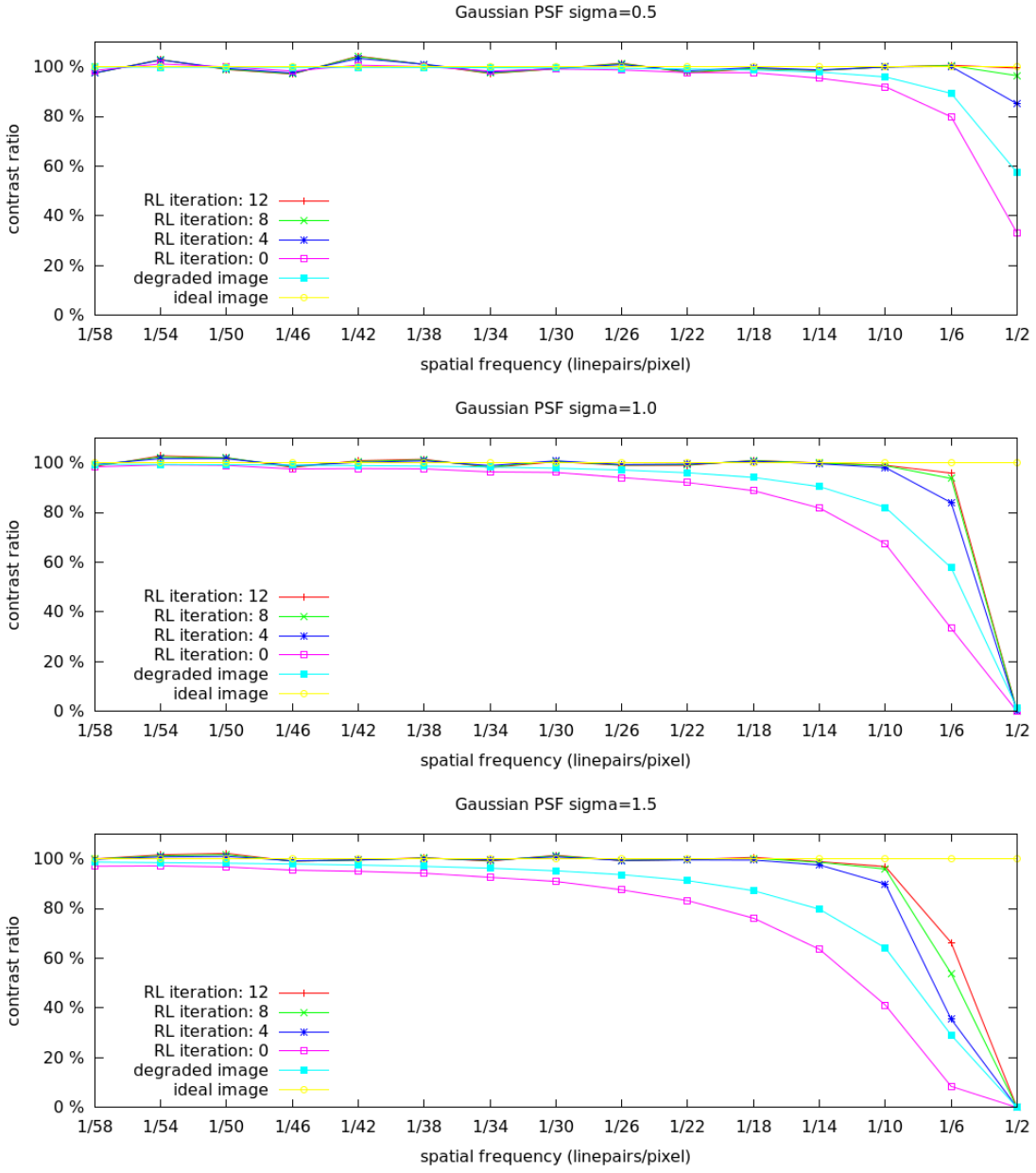



FIGURE 5.5: Noisy Image Restoration MTF Plot. Noise amplification effects visible as contrast is restored to over 100% for some line-pairs.

## 5.3 IMPAIR Images

While noise amplification artifacts can be detected in the MTF curves, the full extent of the noise-amplification properties of the Richardson-Lucy algorithm is more clearly illustrated by comparing the degraded and restored images by eye.

Figure 5.6's left panel shows a text degraded with a broad Gaussian PSF ($\sigma = 2.5$) and corrupted with additive Gaussian noise ($\sigma = 2.0$). The right panel shows the result of an unregularised Richardson Lucy deconvolution. For this level of additive noise, the ringing artifacts introduced by the deconvolution are more visible than the noise amplification.

Figure 5.7 shows the restoration of an image degraded with a higher level of additive Gaussian noise ($\sigma = 8.0$), where noise amplification is now visible.

Figure 5.8 shows the restoration of an image degraded with a broad PSF and corrupted with additive Gaussian noise ($\sigma = 8.0$), but restored with a thinner Gaussian PSF ($\sigma = 1.5$), rather than the full PSF used in the degradation. Ringing artifacts are not present under this approach. Noise amplification is still present, but less invasive on the overall structure of the image.

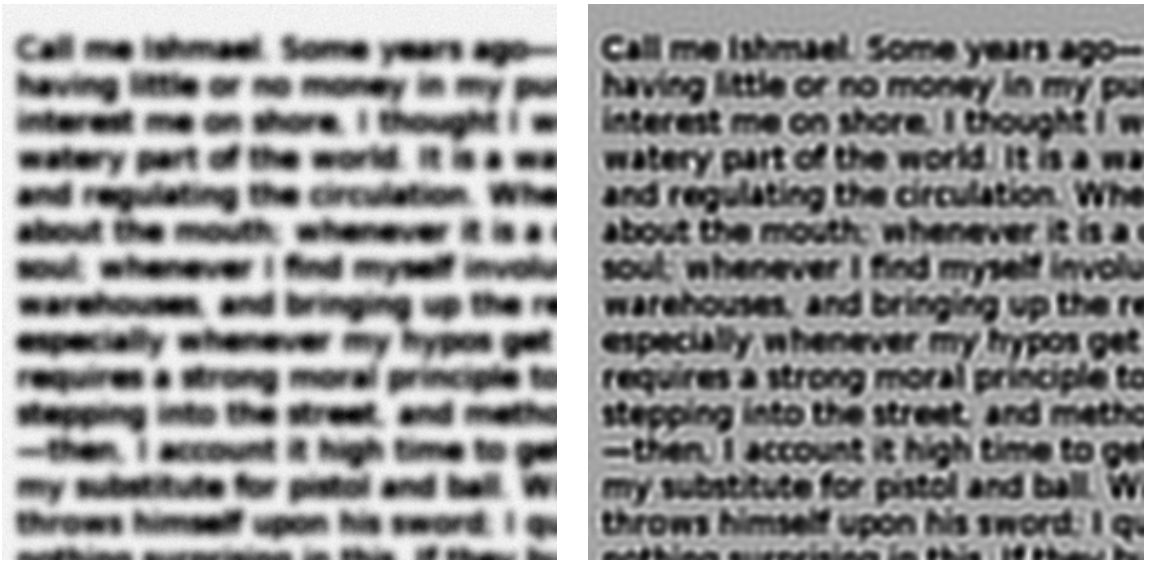



FIGURE 5.6: Low Noise Restoration with Unregularised Richardson-Lucy.
Left: The Image, degraded by a known PSF and corrupted with a lower level of additive Gaussian noise than used in Figure 5.7.
Right: Image restored with 128 iterations of unregularised Richardson-Lucy deconvolution with the known PSF. The steep discontinuity between the uniform background and solid text produces slight ringing artifacts that can be seen outlining each word. Noise amplification artifacts are more present over the uniform background than the text.

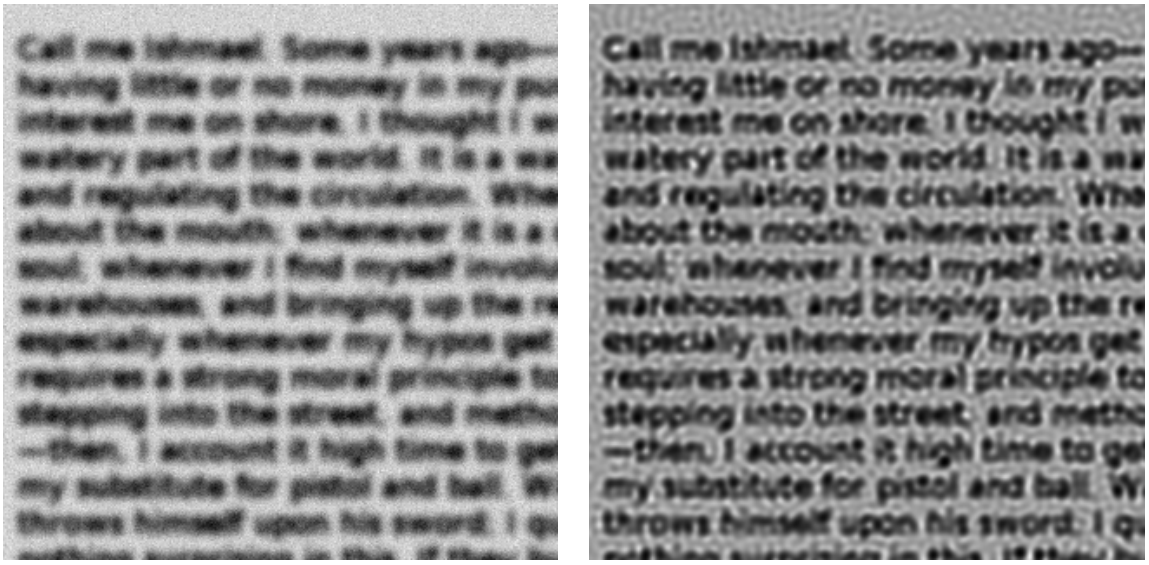


FIGURE 5.7: High Noise Restoration with Unregularised Richardson-Lucy.
Left: The Image, degraded by a known PSF and corrupted with a higher level of additive Gaussian noise than used in Figure 5.6.
Right: Image restored with 128 iterations of unregularised Richardson-Lucy deconvolution with the known PSF. The higher level of noise in the restored suppress ringing artifacts that were present in Figure 5.6, but the noise amplification artifacts are now visible over the text as well as the uniform background.

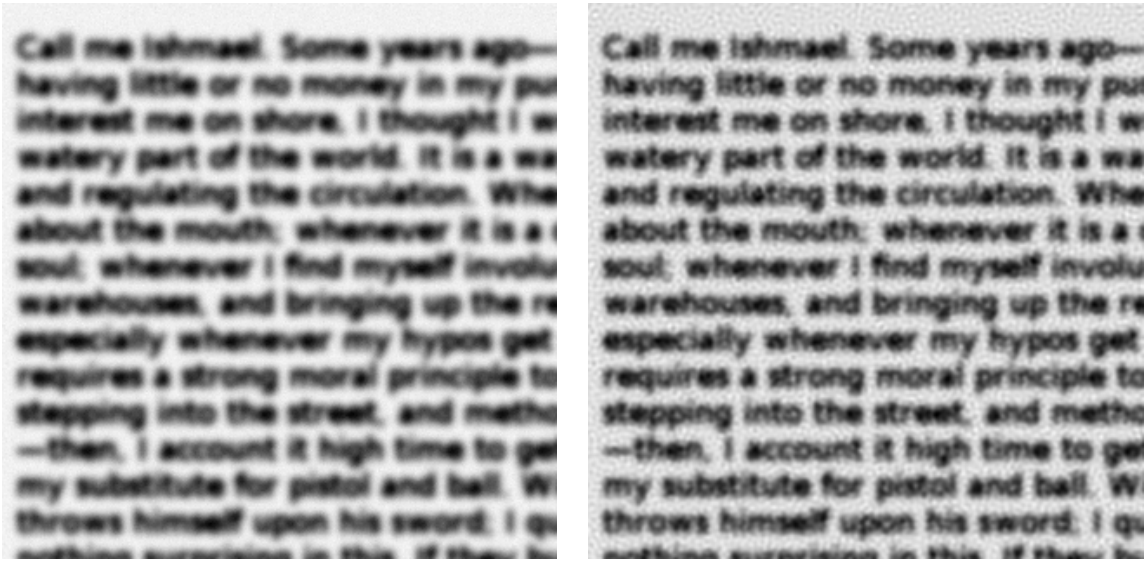


FIGURE 5.8: Unregularised Richardson-Lucy Restoration with thinner PSF.
Left: The Image, degraded by a known PSF and corrupted with a low level of additive Gaussian noise..
Right: The Image restored with unregularised Richardson-Lucy deconvolution, using a thinner PSF than was used for the degradation. This approach significantly reduces ringing artifacts, while still introducing a restorative effect on the image. Noise amplification is still present, but of a different quality to that seen in Figure 5.6. Noise amplification is most visible on the uniform background of the image, but of a finer granularity than seen in Figure 5.6, due to the use of a thinner PSF.

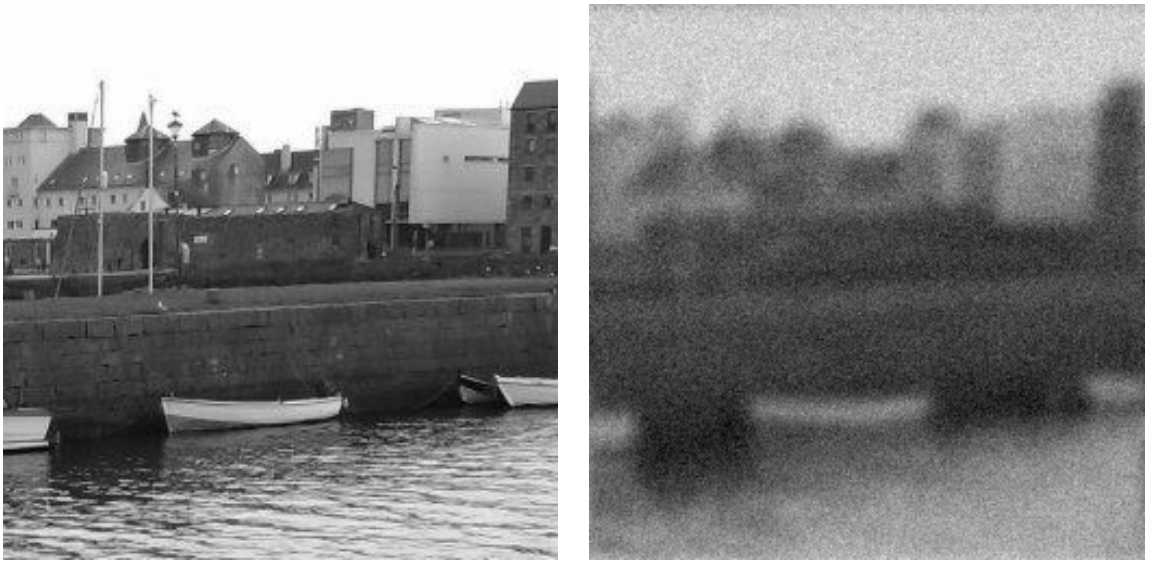


FIGURE 5.9: Uncorrupted and Corrupted Example Image.
Left: The original input image.
Right: The input image degraded with a broad Gaussian PSF ($\sigma = 2.5$) and corrupted with a high level of additive Gaussian noise ($\sigma = 8.0$)

FIGURE 5.10: Unregularised and Wavelet-Regularised Richardson-Lucy Restoration.
Left: Image restored with unregularised RL deconvolution. No ringing artifacts present, due to the lack of extremely sharp discontinuities in the original image, but noise amplification artifacts are clearly visible, and ringing artifacts would emerge with further iterations.
Right: Image restored with wavelet regularised RL deconvolution. Noise amplification has been greatly suppressed in comparison to the unregularised restoration.

## 5.4 Computational Performance

IMPAIR implements four underlying parallelisation strategies: the Naive strategy, a baseline comparison strategy that delegates all parallelism; the Topdown strategy, an array-tiling strategy that processes the image in one go, with one tile per core; the Streaming strategy, a loop-tiling strategy that uses N cores per tile, and processes one tile at a time; and the Queuing strategy, a loop-tiling strategy that processess N tiles at a time, with one core per tile. Other variations, including an N:M tiling strategy, where N cores deconvolve M tiles at a time, were investigated, but found to be out performed by the Topdown, Streaming, and Queuing strategies in all cases. Runtime profiles of an earlier version's Tiling strategy are included in Appendix C.

The Topdown strategy has the largest memory footprint, and the fewest number of synchronisations per deconvolution. The Streaming strategy has the smallest memory footprint, and the Queuing strategy has a larger memory footprint than the Streaming strategy by a smaller factor than the Topdown strategy. The Naive strategy has the largest number of synchronisations per deconvolution, followed by the Streaming strategy. The Topdown and Queuing strategies avoid per-iteration synchronisations and so have the least.

The benchmarks presented in sections 5.5 and 5.6 combined cover the following configurations:

- the size of the image to be deconvolved
- the algorithm that performs the deconvolution

  Unregularised Richardson Lucy

  Haar Wavelet Regularised Richardson Lucy

  Daub-8 Wavelet Regularised Richardson Lucy
- the parallelisation strategy used for the deconvolution algorithm

  Naive

  Streaming

  Topdown

  Queuing
- the platform the deconvolution is parallelised on: either Multicore CPU or GPU

  the number of Multicore CPU cores used by the parallelisation strategy

Multicore CPU benchmarks are presented in section 5.5. These cover the combinations of the three algorithms against the Naive and Topdown strategies, for a range of image sizes and numbers of cores. GPU benchmarks are presented in section 5.6. These cover the combinations of the three algorithms against the Naive and Streaming strategies, for a wider range of image sizes, but with the number of GPU cores fixed to the absolute image size in pixels. Section 5.7 presents results from sections 5.5 and 5.6 in comparison with each other.

All datapoints plotted are the average of 100 runs with the largest standard errors observed as 0.06 and 0.03. As error-bars that cover this range are not visibly distinguishable, none are plotted.

## 5.5 CPU IMPAIR Runtimes

This section details the runtime behaviour of various CPU parallel deconvolution strategies for megapixel sized image data sets. The performance of the algorithms are illustrated in terms of overall runtime, the response of each strategy to a doubling of the available computational cores, and the sensitivity of each strategy to changes in image aspect ratio for images of the same total pixel count for the two strategies with the most erratic performance. The conditions under which one parallelisation strategy becomes more effective than another in terms of overall runtime are highlighted.

### 5.5.1 CPU Naive

The Naive strategy delegates responsibility for parallelising the deconvolution to batch implementations of the WRL algorithm components. These components operate in isolation from one another, synchronising all threads of execution before and after their execution, as well as during the operation if required. Each iteration operates in isolation from the previous, which introduces per-iteration memory copy costs during the preprocessing and post-processing steps of the convolution and shrinking operations.

The cache misses introduced by this behaviour mean that, for suitably large images, the hyperthreaded logical cores are available to bring an improvement over distributing wavelet regularised deconvolution over just the physical cores, as logical threads spend a greater amount of time stalled due to cache misses. This can be seen in the Daub-8 runs of Figure 5.12 where the gradual increases in runtime of the single-threaded deconvolutions for image data with an aspect ratio of $2:1$ correspond with continued linear scaling in runtime for the hyperthreaded 8 core deconvolutions. Figure 5.11 shows an improvement

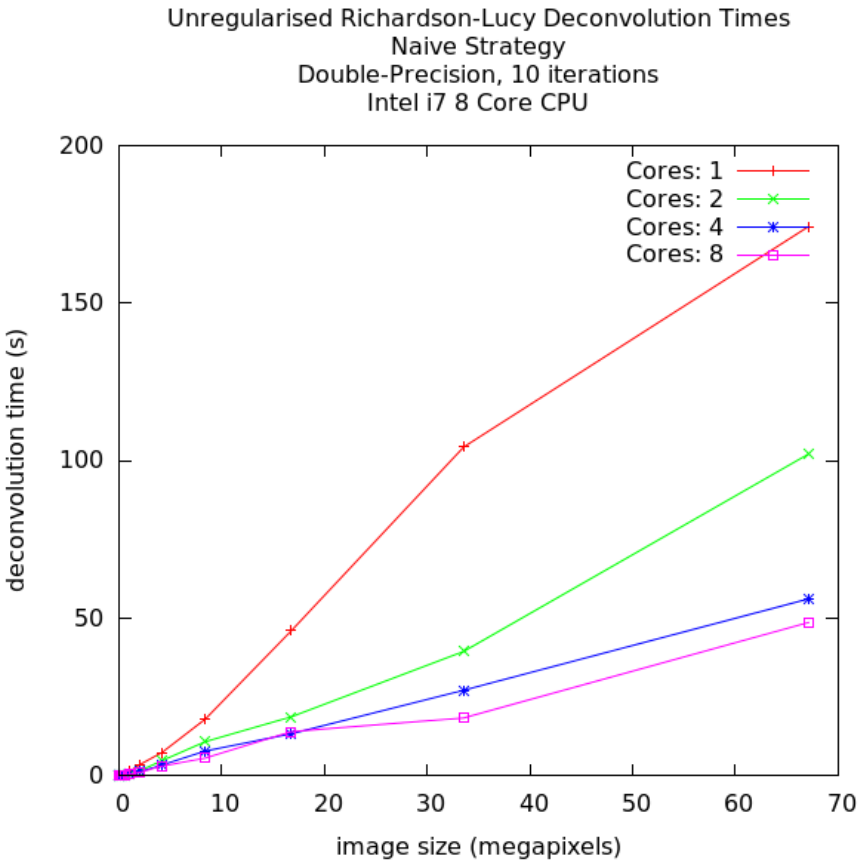
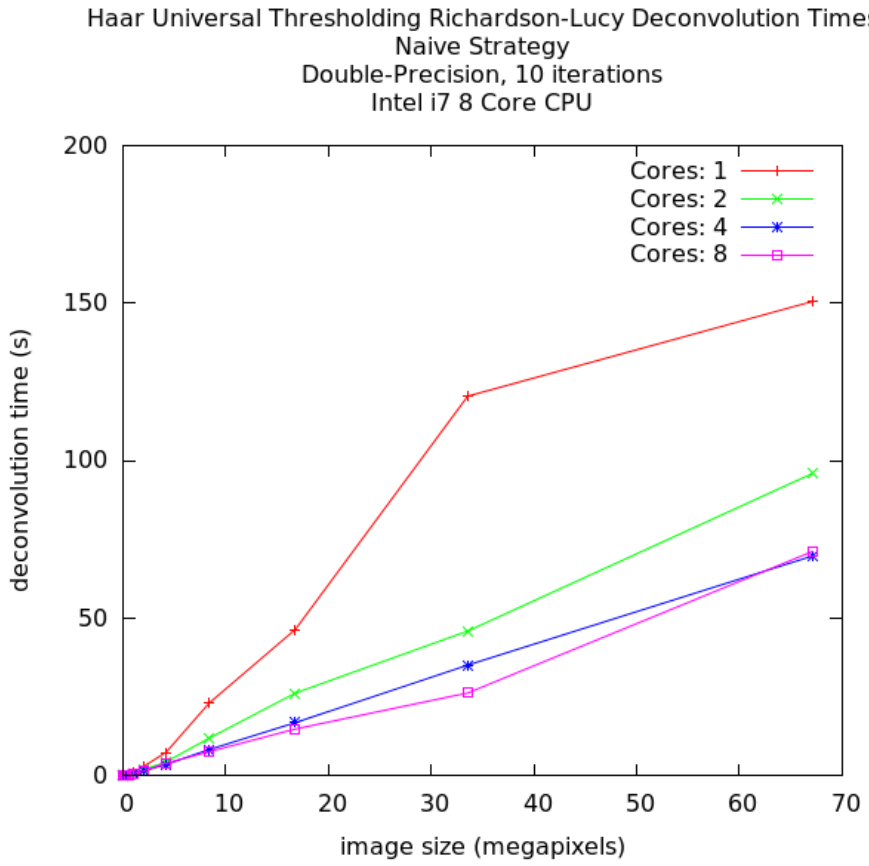



FIGURE 5.11: Unregularised Naive Megapixel Image Performance. The hyperthreaded 8 core configuration does not begin to provide an advantage over the non-hyperthreaded 4 core configuration until the input image size exceeds 32 megapixels. See Section 5.5.1 for further discussion.

of the hyperthreaded configuration over the 4 core physical configuration, but in this case there is no corresponding drop in performance for the 4 core physical configuration (unlike the crossing over and back behaviour seen for the Topdown strategy in Figures 5.18 and 5.19). If the speedup of the hyperthreading configuration was due to an increase in the cache misses of the physical configuration, the physical configuration would be expected to progress at a lower rate than when the hyperthreaded and physical configurations performed identically.



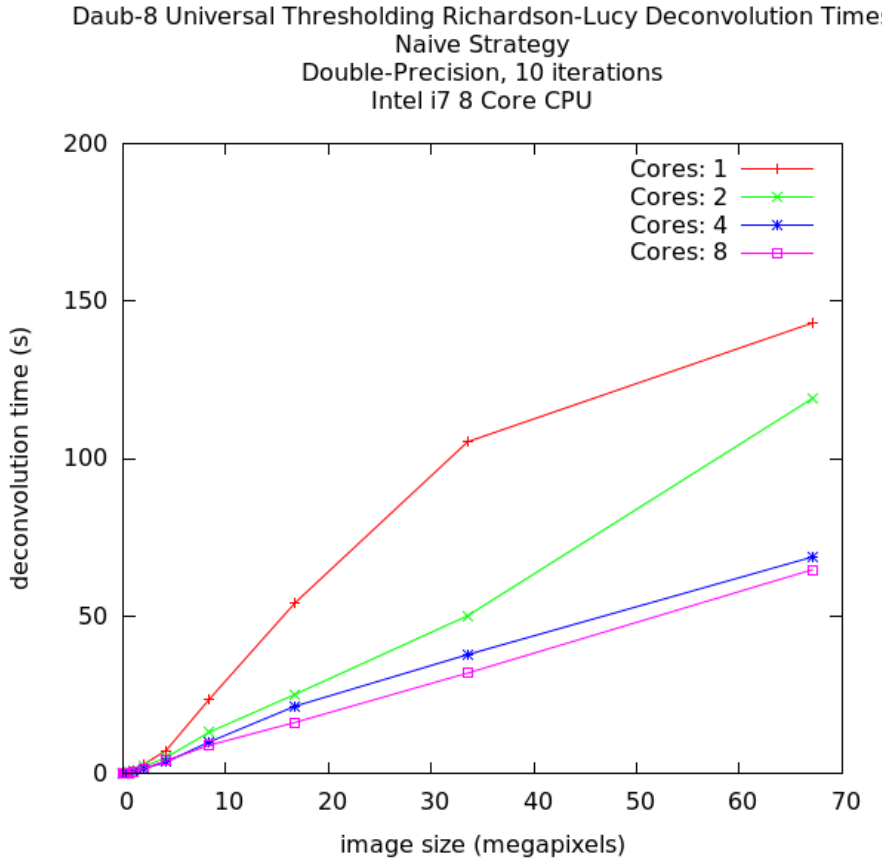




FIGURE 5.12: Regularised Naive Megapixel Image Performance. The hyperthreaded 8 core configuration provides less of an advantage over the non-hyperthreaded 4 core configuration, though for the Daub-8 run, this advantage occurs earlier, after the input image size exceeds 8 megapixels. See Section 5.5.1.

Since the performance of the Naive strategy is sensitive to the aspect ratio of the image, as well as the overall size in megapixels, as shown later in Figures 5.15 and 5.17, the hyperthreaded cores provide a mechanism for smoothing out these irregularities once the latencies that would be introduced surpass the threshold defined by the time to fetch a cache line.

Figures 5.13 and 5.14 show the per-core speedup for $512 \times 512$ and $2048 \times 2048$ pixel images, for the unregularised and regularised RL algorithms. The unregularised algorithm responds well to each additional physical core for the $512 \times 512$, achieving almost a $\times 1$ and $\times 4$ speedup over the serial configuration. The additional hyperthreaded cores provide negligible benefit as expected for an image of this size, which will not be subject to cache-miss penalties. For the $2048 \times 2048$ configuration, the algorithm responds poorly to each additional core, requiring $4\times$ the number of physical cores in order to achieve a $\times 2$ speedup.

Figures 5.15 and 5.17 show the variation in runtimes for small changes in image

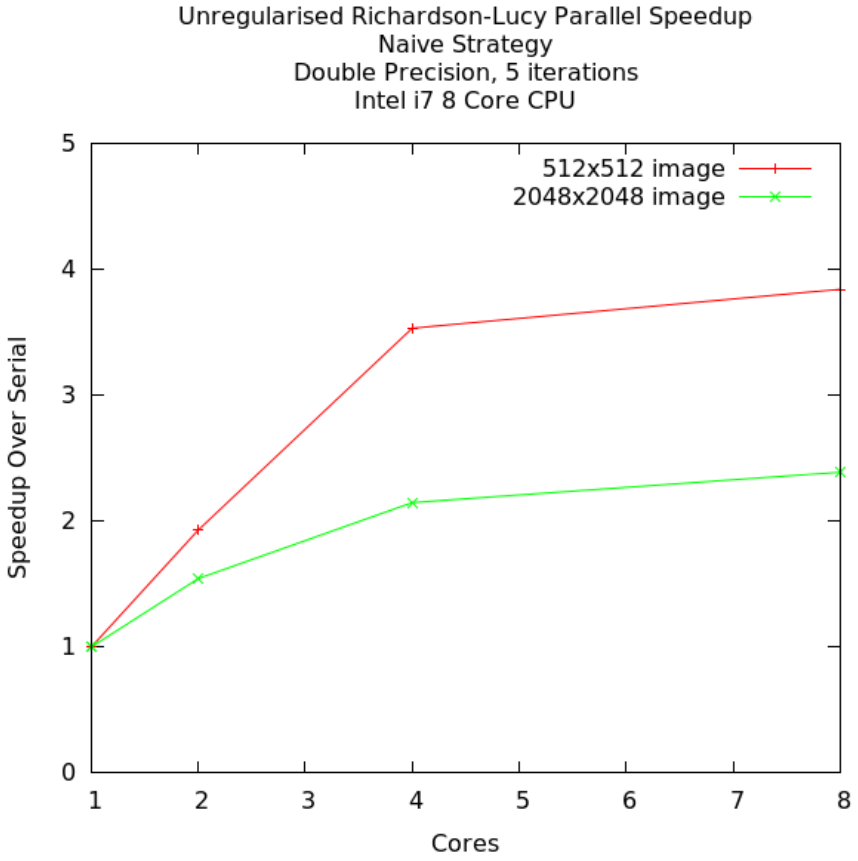



FIGURE 5.13: Unregularised Naive Per-Core Speedups. The unregularised Naive strategy achieves the greatest return on each additional core for input images in the kilopixel range ($512 \times 512$ pixels) than input images in the megapixel range ($2048 \times 2048$ pixels). See Section 5.5.1 for further discussion.

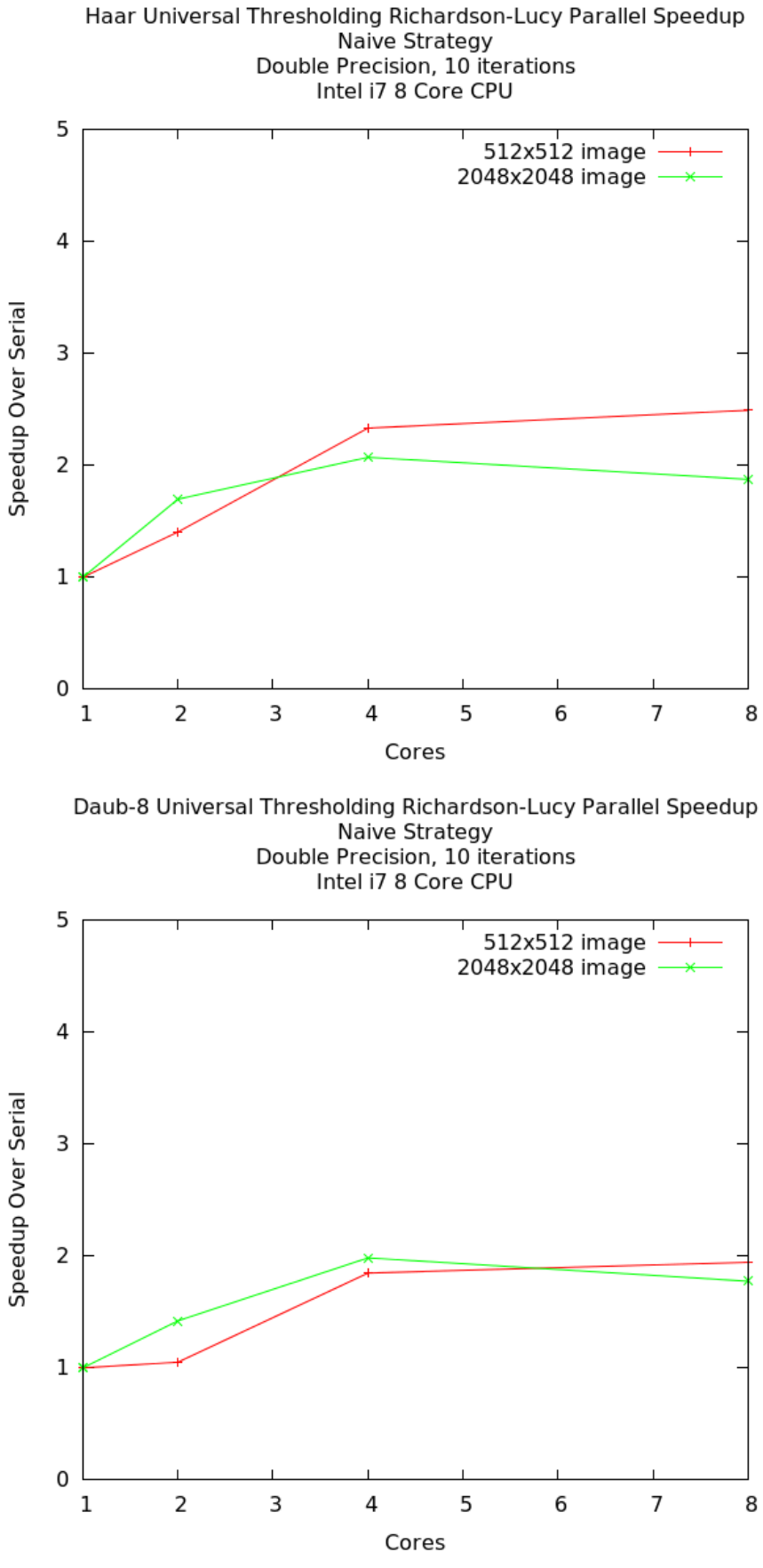



FIGURE 5.14: Regularised Naive Per-Core Speedups. The regularised Naive strategy achieves less return on each additional core than the unregularised runs shown in Figure 5.13. See Section 5.5.1.

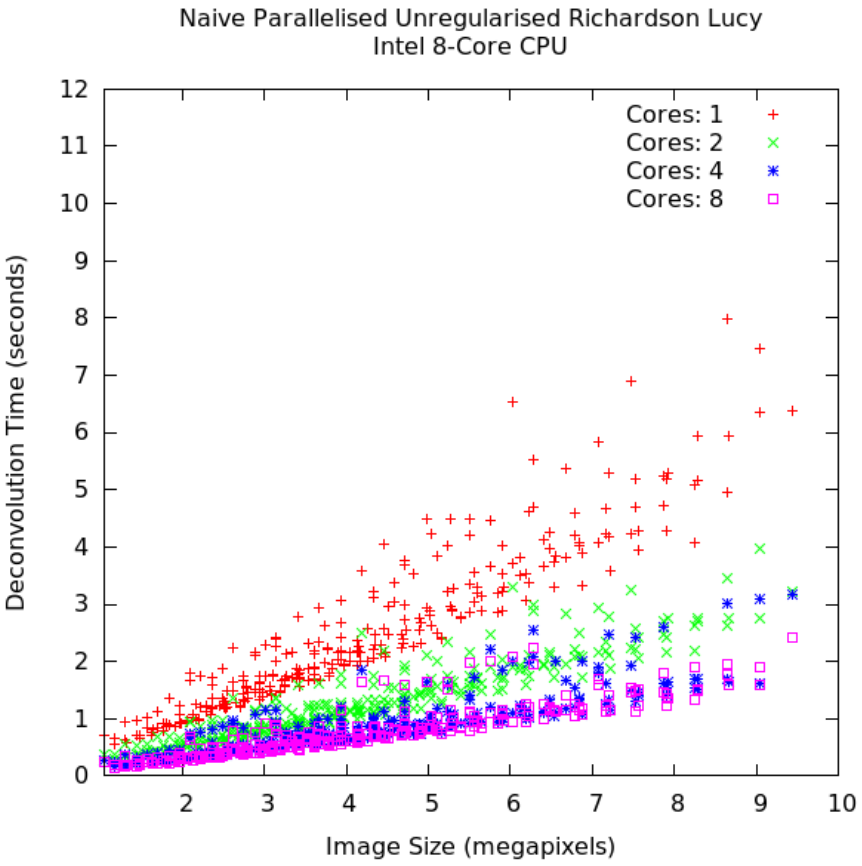



FIGURE 5.15: Unregularised Naive Runtime Variations for similar image sizes

size, and aspect ratio. These variations can be large, particularly for the unregularised algorithm, and influence the runtime performance of the CPU Topdown parallelisation strategy discussed in Section 5.5.2.

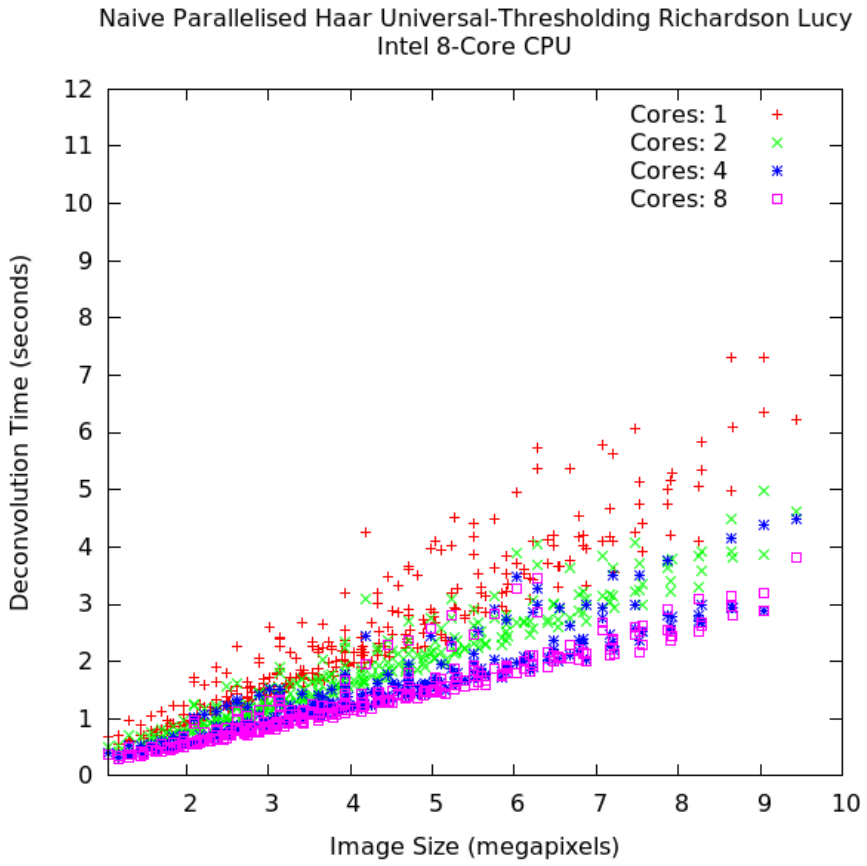



FIGURE 5.16: Haar Universal Thresholding Naive Runtime Variations for similar image sizes

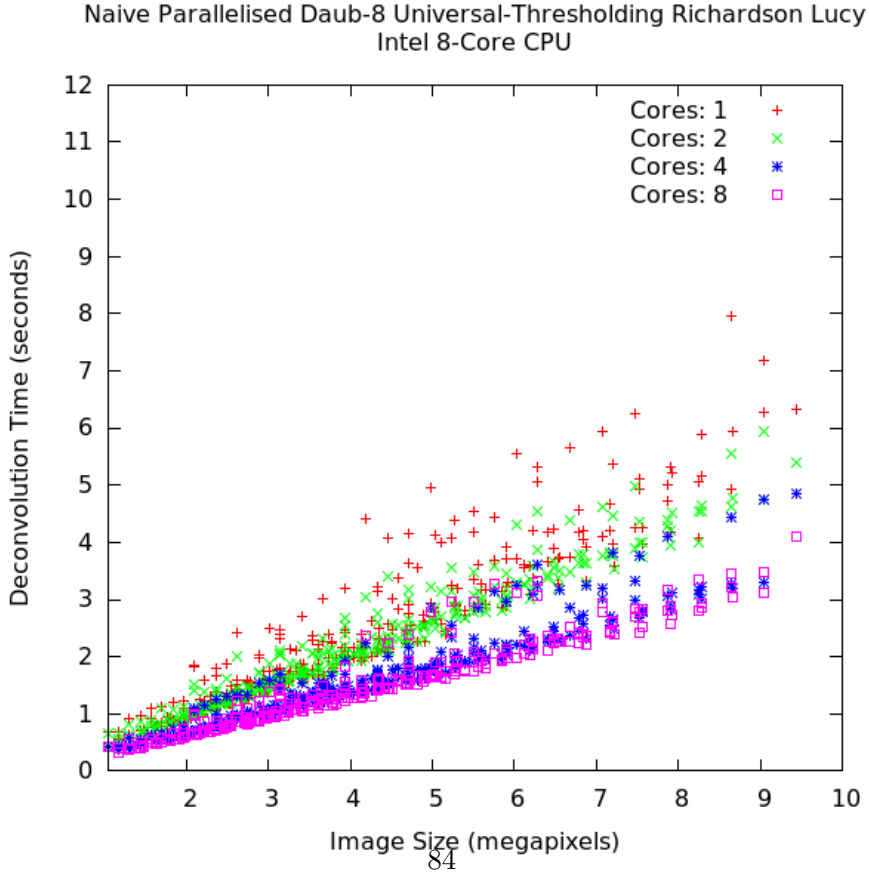





FIGURE 5.17: Daub-8 Universal Thresholding Naive Runtime Variations for similar image sizes

### 5.5.2 CPU Topdown

The Topdown parallelisation strategy divides the image deconvolution into a set of independent tiles, which are deconvolved in isolation from each other. By constraining all the parallelisation to the coarsest level, this strategy avoids the per-iteration synchronisation costs of the Naive strategy, but with the trade-off of an increased number of pixels that must be processed per iteration. As the Topdown strategy uses the number of available cores to determine how many tiles the input image is divided into, the single-core configuration of the Topdown strategy is identical to the Naive strategy's single-core case.

Figures 5.18 and 5.19 show the performance of the parallelisation strategy for megapixel images. For the single-threaded case, the Topdown strategy and the Naive strategy are identical. These profiles show a growing discrepancy between the performance of the single threaded case and the multicore case as image sizes increase towards 100 megapixels. This behaviour is highlighted later in Figures 5.21 and 5.22. In contrast to the behaviour of the Naive strategy, the 8 core configuration of the Topdown strategy does not consistently out-perform the 4 core configuration for larger images, suggesting that the Topdown strategy exhibits more streamlined use of the CPU cache hierarchy than the Naive strategy, giving the hyperthreaded cores less opportunities to overlap their execution with a cache miss. This possibility is discussed in further detail in Chapter 6 Section 6.3.

Figure 5.20 shows the variation in runtimes for small changes in image sizes, including contributions from the variability in the underlying Naive algorithm. The consequences of these variations are for particular image sizes or aspect ratios, deconvolution can take almost double the time as for a larger image, with a non-worst-case aspect ratio.

Figures 5.21 and 5.22 show the per-core speedup for the unregularised and regularised algorithms. Unlike the Naive parallelisation strategy shown in 5.5.1, the Topdown strategy does not respond with a speedup to additional cores for small image sizes.

The Topdown strategy is shown in these figures to experience super-linear speedups for each additional physical core that is added to the configuration. This is unexpected as speedup of the Topdown strategy with each doubling of the cores should closely correspond to the speedup of the single-threaded Naive strategy as the image size in megapixels is halved, particularly for large images, where the overlapping regions will occupy a negligible proportion of the tile size.

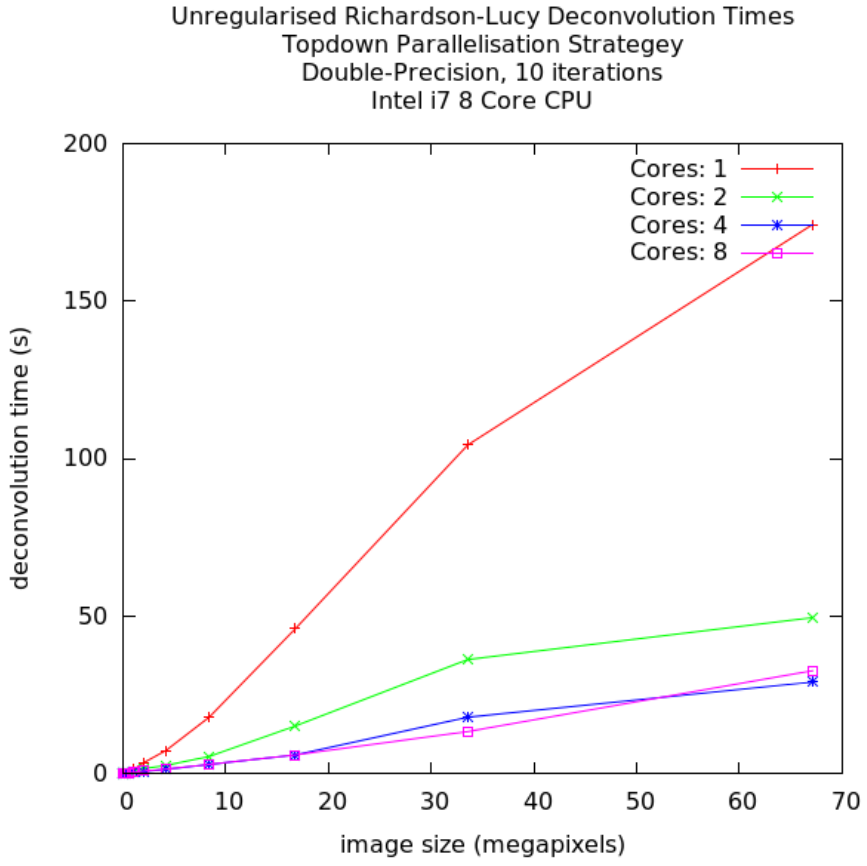



FIGURE 5.18: Unregularised Topdown Megapixel Image Performance. The 8 core hyper-threaded configurations does maintain an advantage over the 4 core non-hyperthreaded configuration, in contrast to the behaviour of the Naive strategy shown in Figure 5.11. This is discussed in more detail in Section 5.5.2.

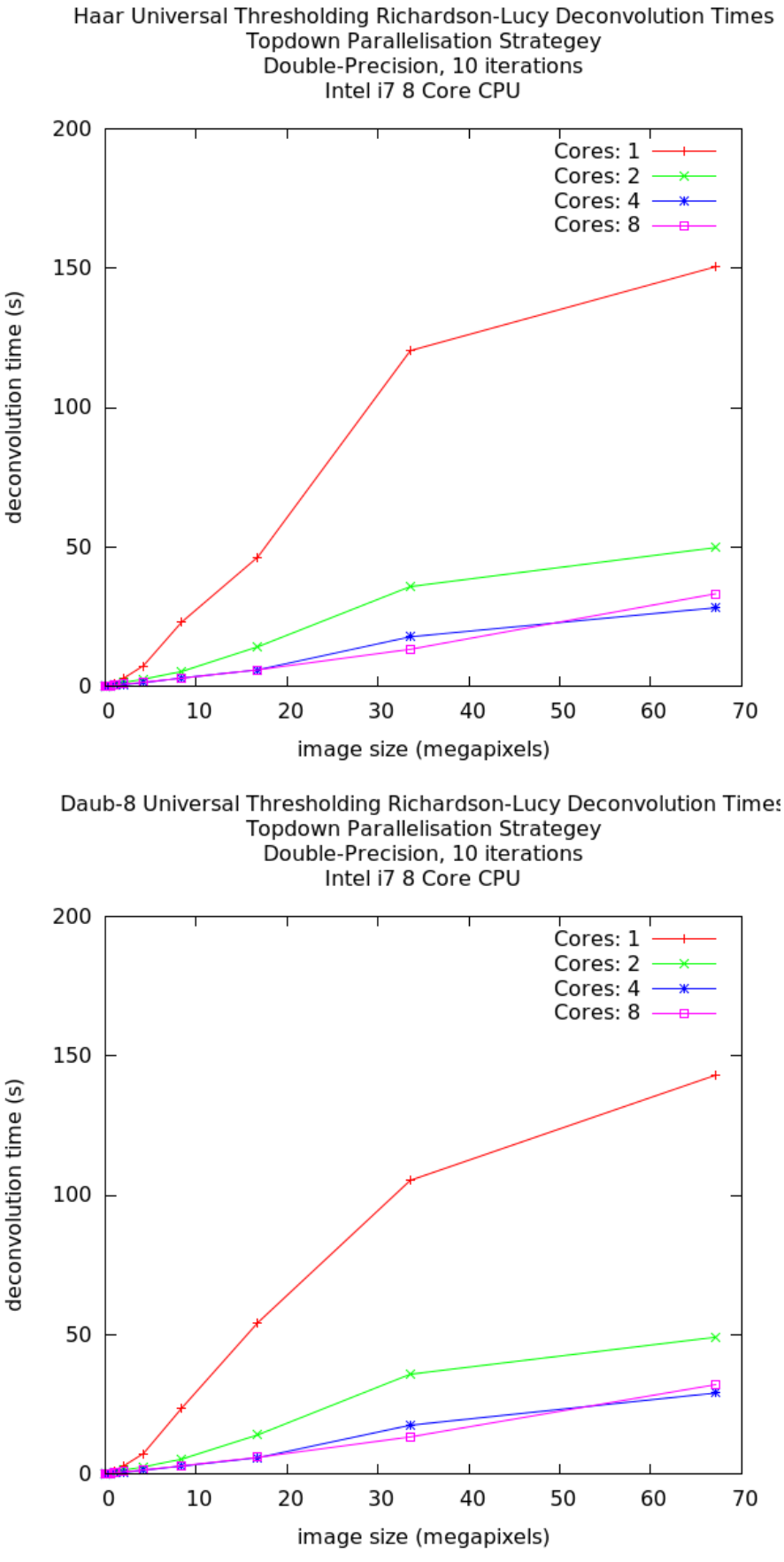



FIGURE 5.19: Regularised Topdown Megapixel Image Performance. The regularised Topdown strategy runs exhibit little difference in runtime when compared to the unregularised Topdown strategy shown in Figure 5.18, including the cross-over between the runtimes of the 4 core and 8 core configurations. See Section 5.5.2.

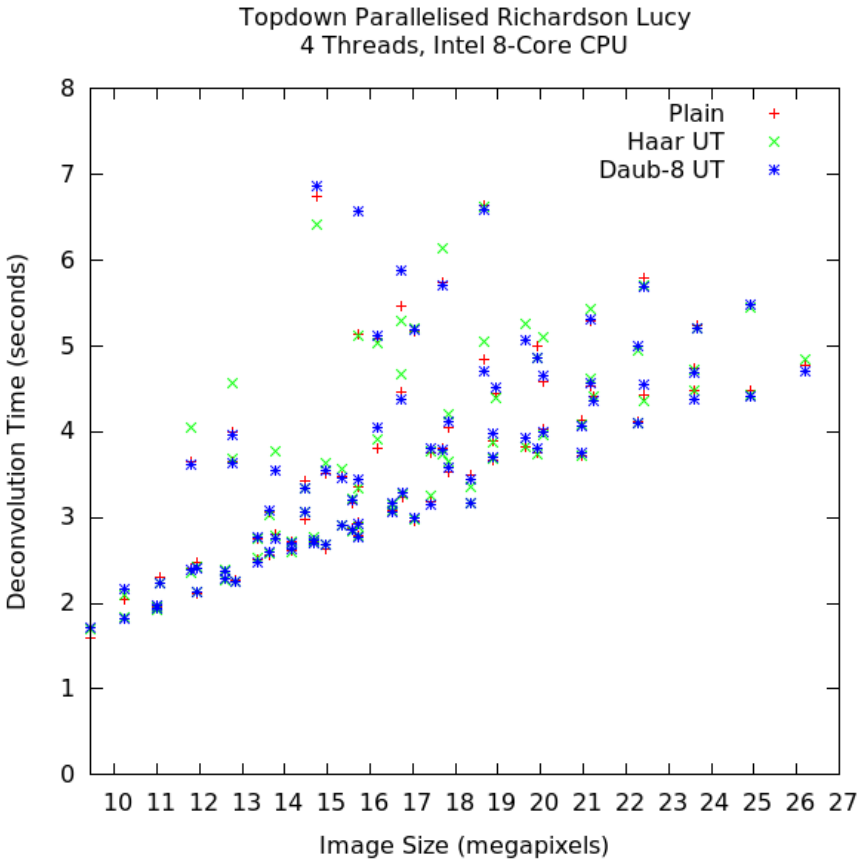



FIGURE 5.20: Topdown Image Size Runtime Variations

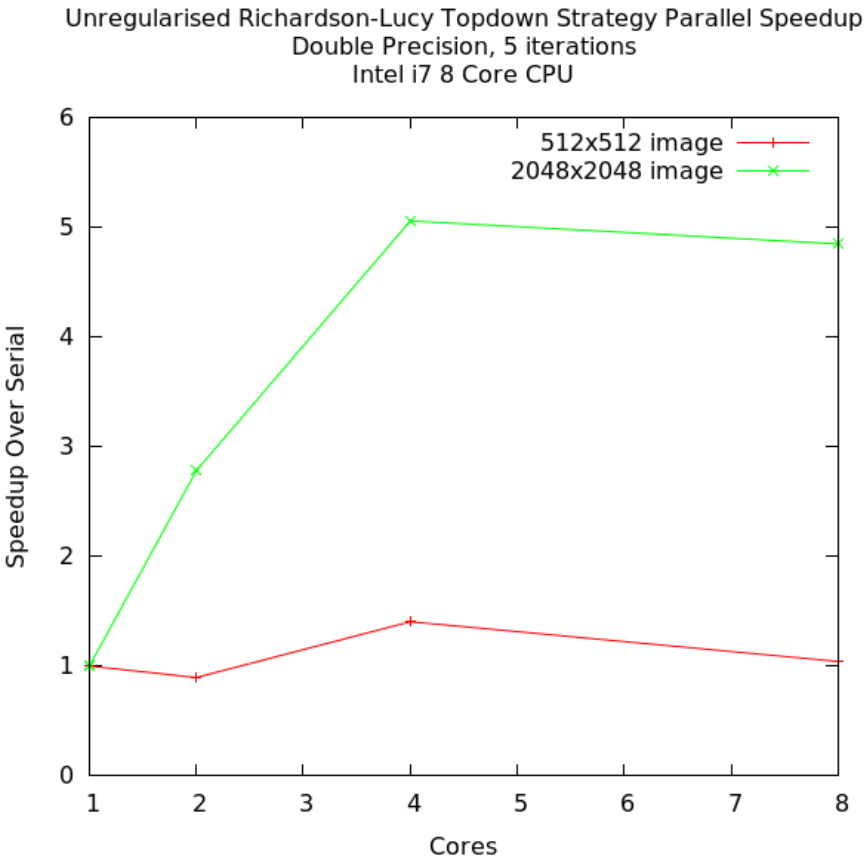
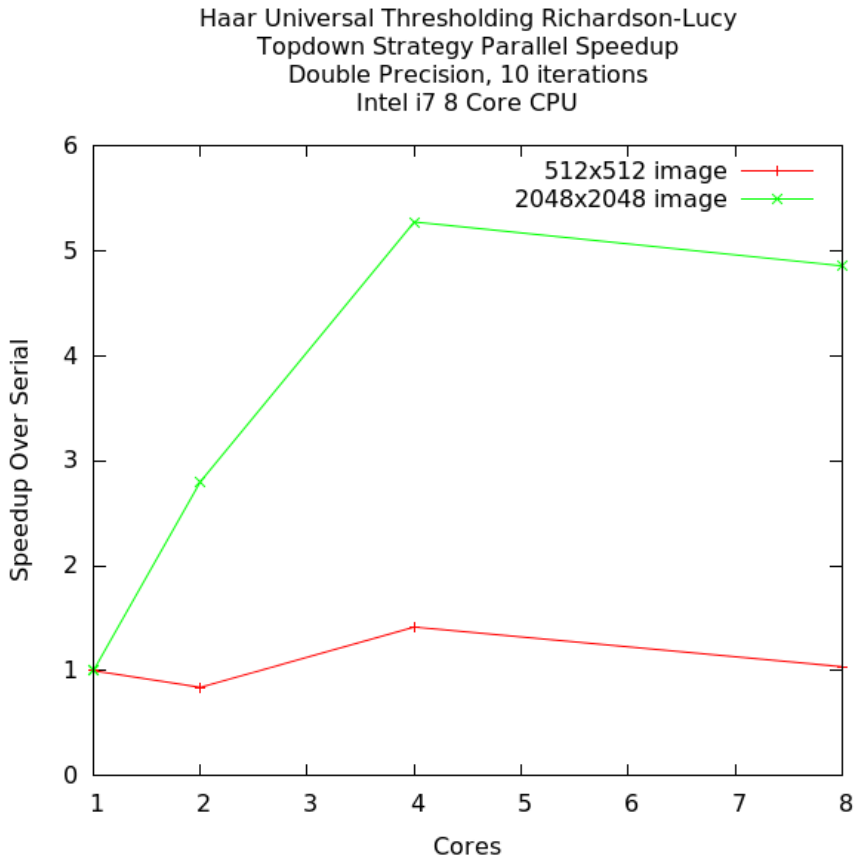



FIGURE 5.21: Unregularised Topdown Per-Core Speedups. The unregularised Topdown strategy, along with the regularised Topdown strategy profiled in Figure 5.22, exhibits a super-linear speedup during the transition from the 1 core to 2 core configuration. The doubling between 2 core to 4 cores gives a halving of the runtime of the two core configuration, as can be seen in Figure 5.18, though this too presents as a super-linear speedup when scaled to the single-core configuration.



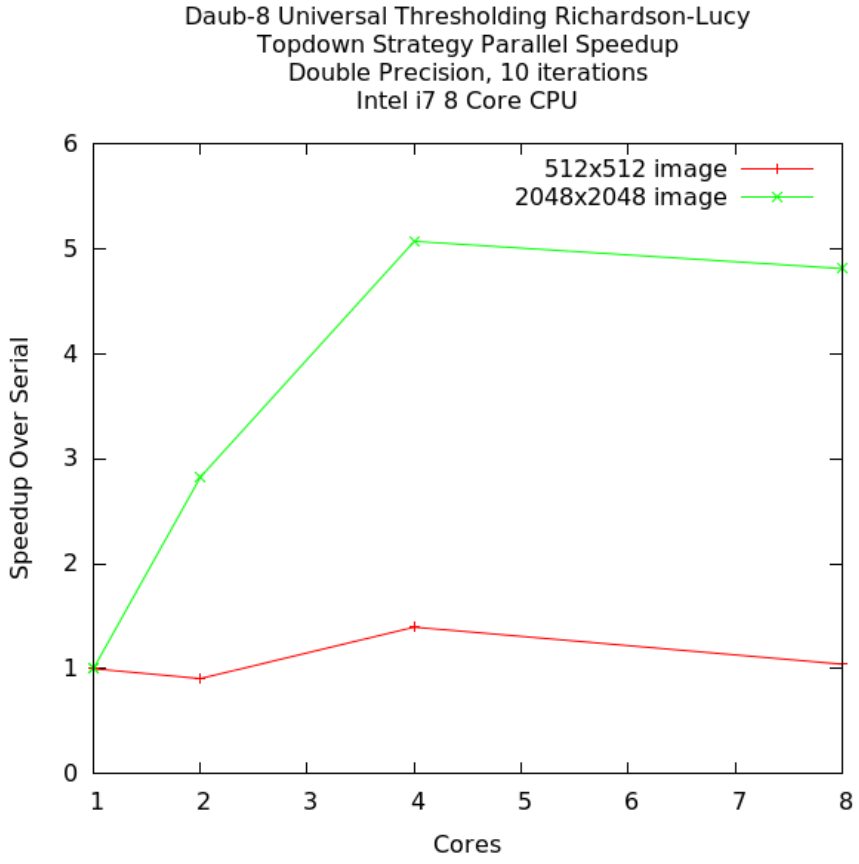




FIGURE 5.22: Regularised Topdown Per-Core Speedups. The regularised Topdown strategy, along with the unregularised Topdown strategy profiled in Figure 5.22, exhibits a super-linear speedup during the transition from the 1 core to 2 core configuration. The doubling between 2 core to 4 cores gives a halving of the runtime of the two core configuration, as can be seen in Figure 5.19, though this too presents as a super-linear speedup when compared to the single-core configuration.

### 5.5.3 CPU Streaming

The CPU Streaming Strategy builds on top of the CPU Naive Strategy in the same way that the GPU Streaming Strategy builds on top of the GPU Naive Strategy, discussed in Section 5.6.2. The CPU Streaming Strategies divide the input image into fixed sized tiles, which are then processed one-at-a-time, using all available cores. For the CPU Streaming Strategy, the tile dimensions are determined by the dimensions of the PSF image, such that the pixels in the overlapping region of the tile do not make up more than 50% of the tile's total pixel count.

The CPU Streaming Strategy's parallelism is delegated to the CPU Naive Strategy that is processing each individual tile. Consequently, there are numerous synchronisation operations during each iteration of the deconvolution. The single-core configuration CPU Streaming Strategy is not subject to these synchronisation costs.

This strategy has the lowest memory requirements of all the CPU strategies implemented by IMPAIR, and will naturally employ more cache-friendly memory access patterns for the deconvolution of a large image and a small PSF than the Naive or Topdown Strategies.

Figures 5.23 and 5.24 show the runtime performance of the CPU Streaming Strategy for a range of image sizes and a fixed tile size. The runtime performance of the Streaming Strategy scales regularly with the size of the input image, and is capable of scaling to beyond 70 megapixels, due to its reduced memory requirements (See Section 5.7.3 for further details). For images sized in the tens of megapixels the hyperthreaded 8 core configuration visibly outperforms all others. This suggests that the reduced data locality of the Streaming Strategy's tiles allows the underlying Naive Strategy's regular cache misses to be taken advantage of by the CPU hyperthreading.

Figures 5.25 and 5.26 show the per-core speedup of the CPU Streaming Strategy for a set of image sizes. These figures show equivalent speedups regardless of the size of the input image. For the regularised deconvolution runtimes, the parallel configurations experience significant overhead due to the cost of the regularisation step, while the serial configuration does not appear to (See Figure 5.24). For the Daub-8 two-core configuration, this overhead is enough to slow the deconvolution to behind that of the serial configuration.

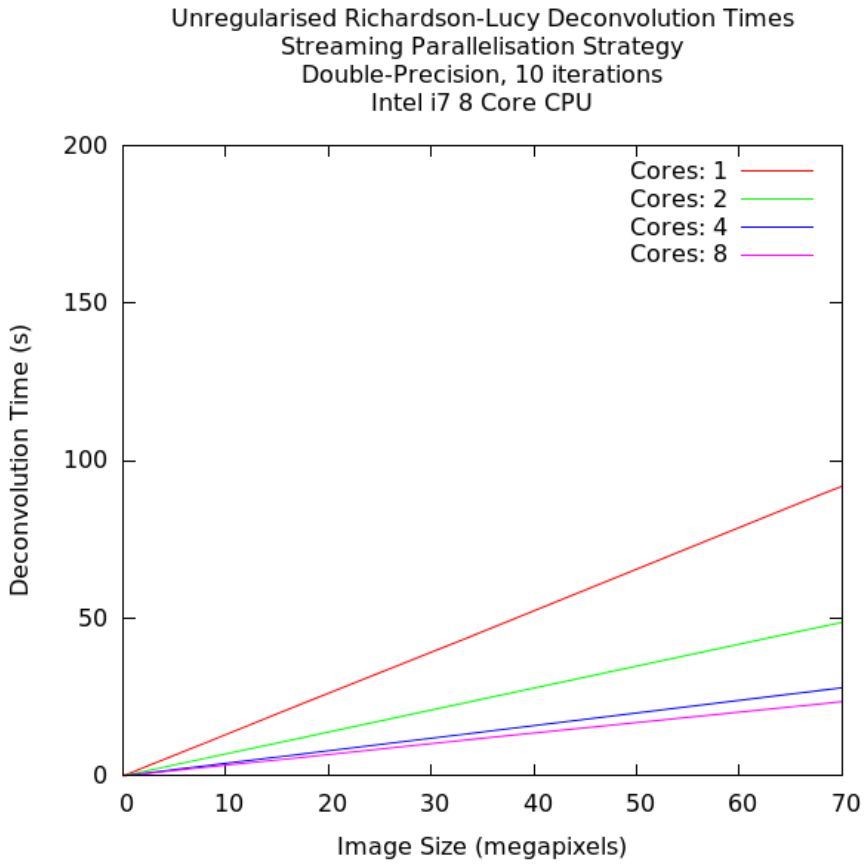
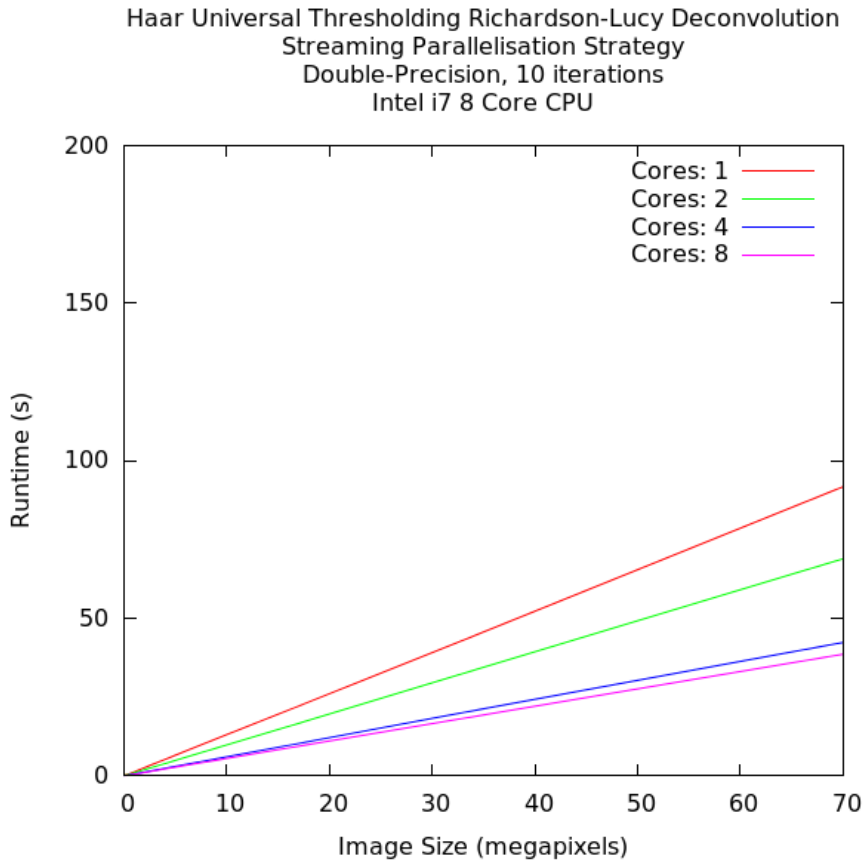



FIGURE 5.23: Unregularised Streaming Megapixel Image Performance. The CPU Streaming strategy scales in a far more deterministic fashion than the Naive or Topdown strategies (see Figures 5.11 & 5.18). The hyperthreaded 8 core configuration consistently outperforms the 1,2,4 core configurations for larger image sizes. See Section 5.5.3 for further discussion.



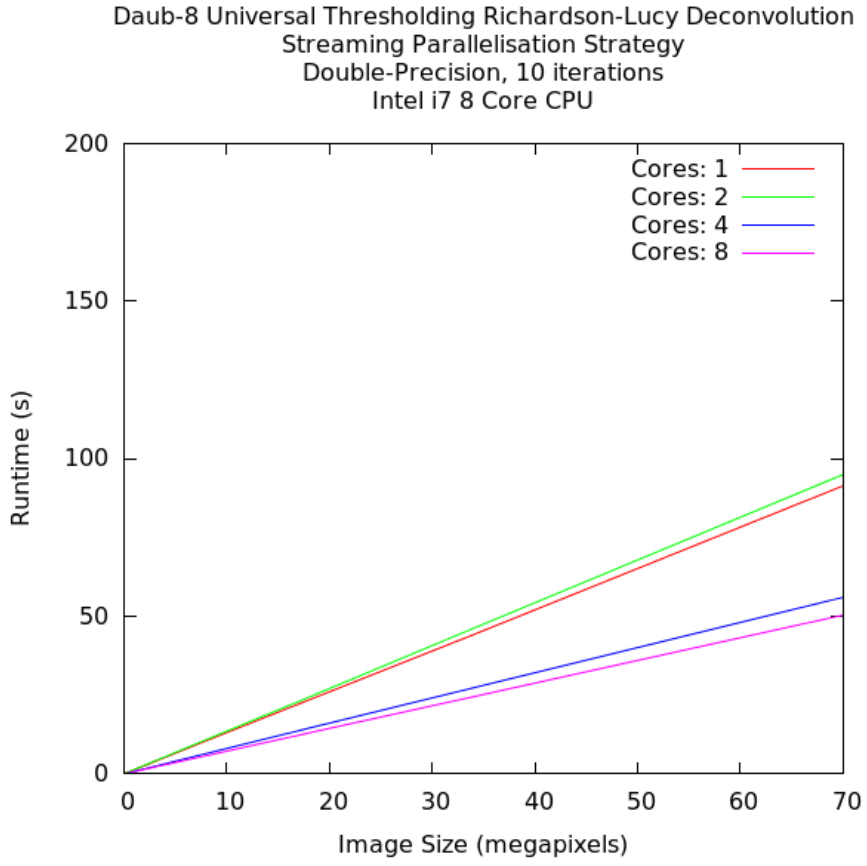




FIGURE 5.24: Regularised Naive Megapixel Image Performance. The hyperthreaded 8 core configuration outperforms the 1,2,4 core configurations for larger image sizes, similar to what is shown in Figure 5.23. For regularisation with the Daub-8 wavelet, the number of per-iteration synchronisations required by the wavelet filter for the 2-core configuration slows this configuration to the point that it is overtaken by the single-core configuration. See Section 5.5.3 for further discussion.

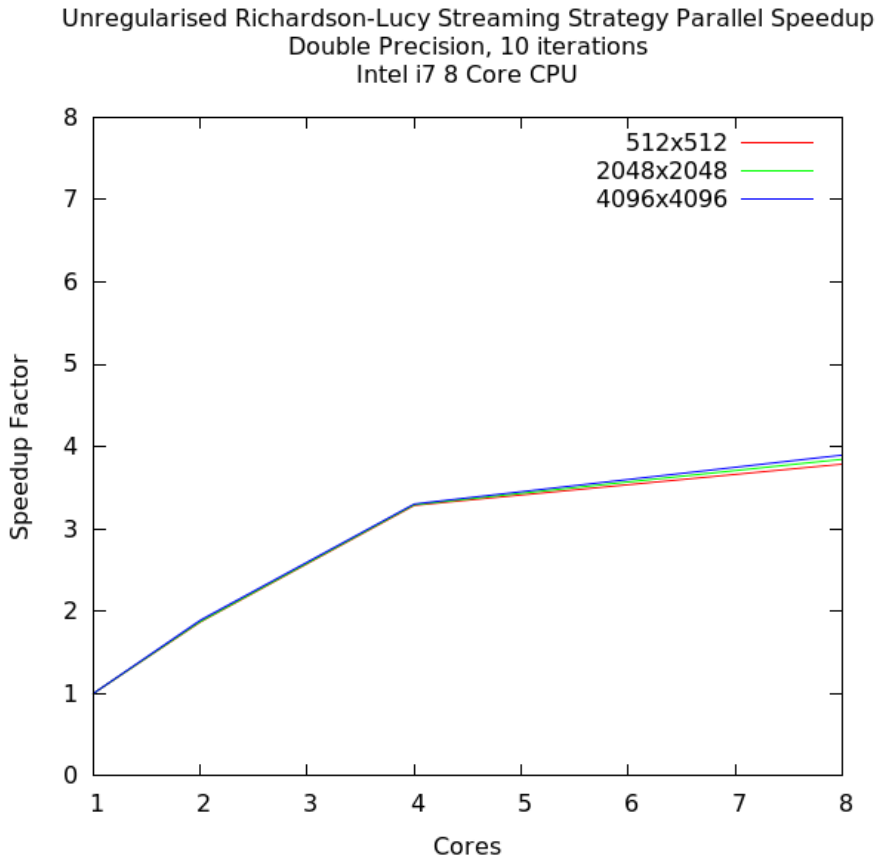



FIGURE 5.25: Unregularised Streaming Per-Core Speedups. The unregularised streaming strategy achieves equivalent speedups with each additional core for images in the kilopixel ($512 \times 512$ pixels) and megapixel ($2048 \times 2048$ & $4096 \times 4096$) range. A slight separation can be seen for the hyperthreaded 8 core configuration, with the larger images acheiving marginally larger speedups. See Section 5.5.3 for further discussion.

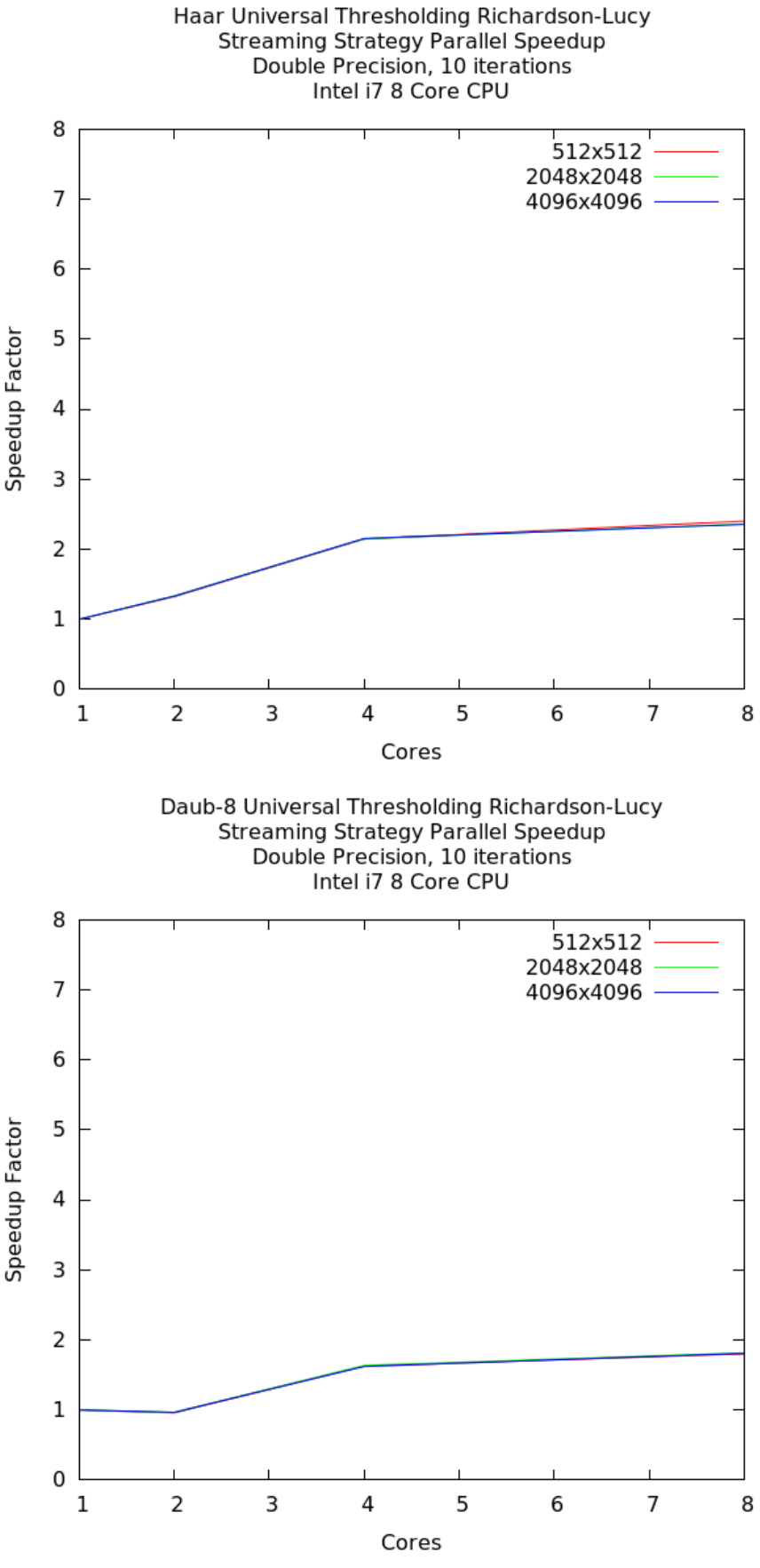



FIGURE 5.26: Regularised Streaming Per-Core Speedups. The regularised streaming strategies show less improvement than was seen in Figure 5.25 with each additional core, with the Daub-8 wavelet regularisation experiencing a slow-down in the two-core case. Both the Haar and Daub-8 wavelets show equivalent speedups for each additional core for images in the kilopixel ($512 \times 512$ pixels) and megapixel ($2048 \times 2048$ & $4096 \times 4096$) range. See Section 5.5.3 for further discussion.

### 5.5.4 CPU Queuing

The CPU Queuing Strategy creates a Streaming Strategy for each core, and assigns an equal portion of the input image to each. As each underlying Streaming Strategy is executed in its single-core configuration, there are no per-iteration or per-tile synchronisations during the deconvolution operation—similar to the Topdown Strategy. The CPU Queuing Strategy has larger memory requirements than the CPU Streaming Strategy, but still less than the Naive Strategy. For the single-core configuration, the CPU Queuing Strategy and the CPU Streaming Strategy are identical.

Figures 5.27 and 5.28 show the runtime performance of the CPU Queuing Strategy for a range of image sizes and a fixed tilesize. The unregularised deconvolution strategy scales with the input image size just as well as the Streaming Strategy scales, and outperforms the Streaming Strategy for the regularised deconvolution. Like the Streaming Strategy, the Queuing Strategy is capable of scaling to image sizes above 70 megapixels (See Section 5.7.3 for further details).

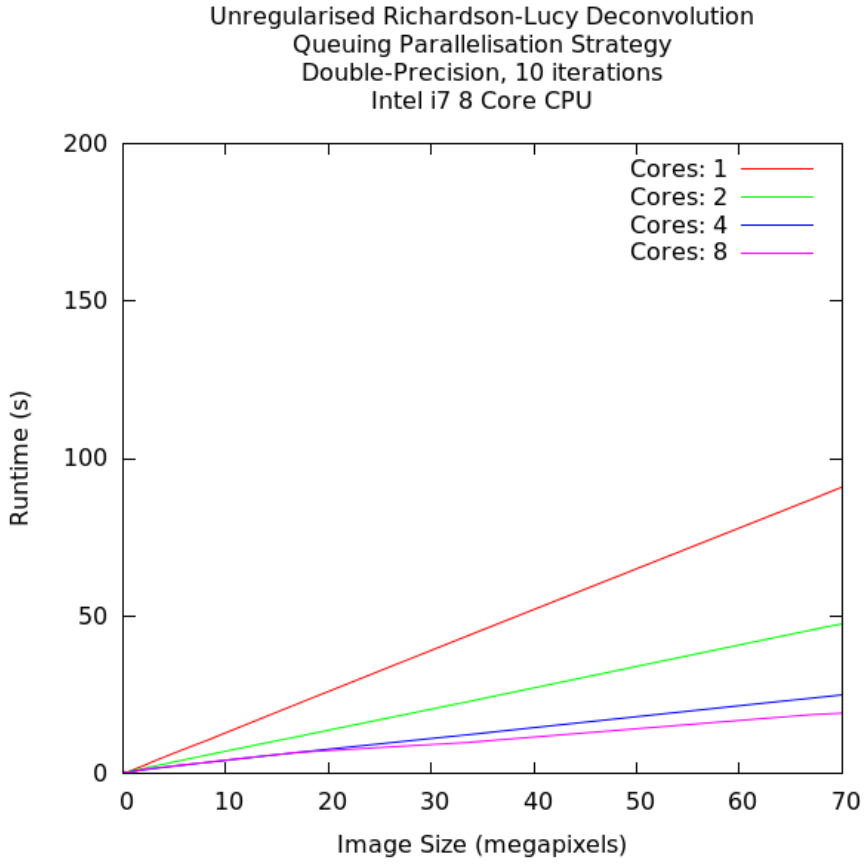
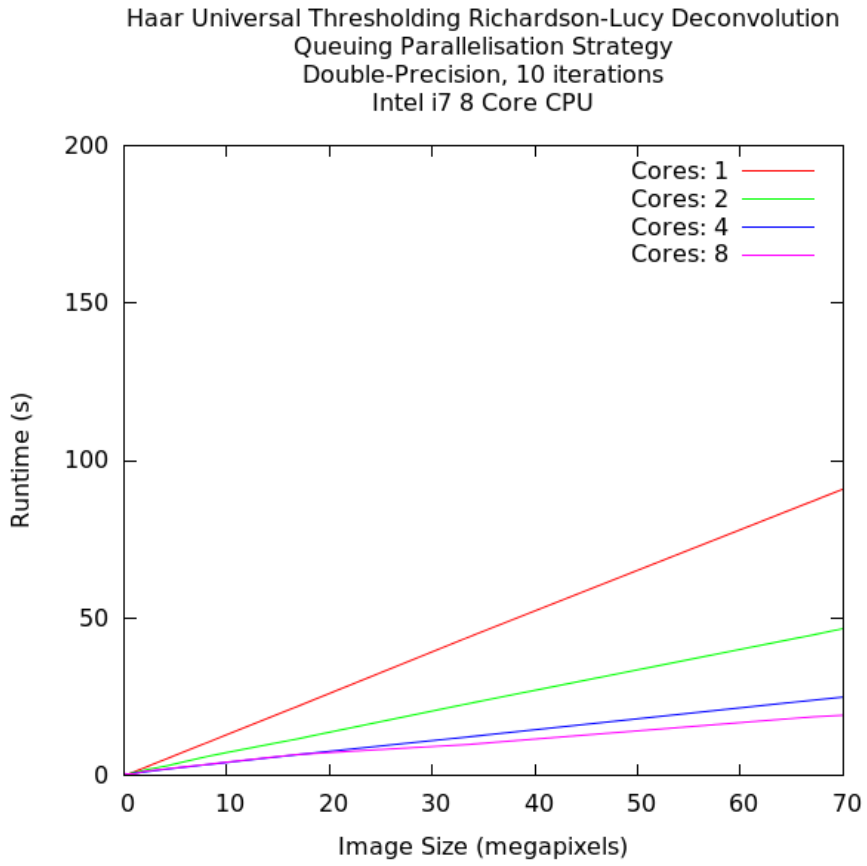



FIGURE 5.27: Unregularised Queuing Megapixel Image Performance. The unregularised Queuing parallelisation strategy scales in a deterministic fashion, similar to the Streaming parallelisation strategy shown in Figure 5.23. For the single-core case, these two strategies are identical. See Section 5.5.4 for further discussion.



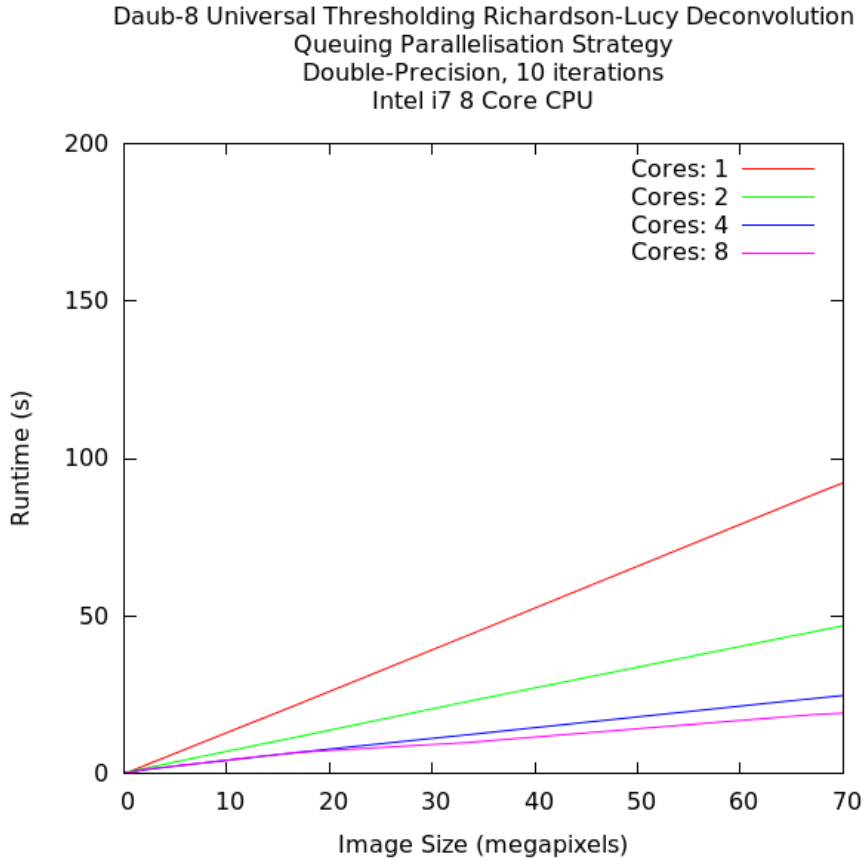




FIGURE 5.28: Regularised Queuing Megapixel Image Performance. The Regularised Queuing parallelisation strategy scales equivalently to the unregularised Queuing strategy, and does not experience the slowdown observed in the Daub-8 case of Figure 5.24. See Section 5.5.4 for further discussion.

Figures 5.29 and 5.30 show the per-core speedup of the CPU Queuing Strategy for a set of image sizes. For the profiled image sizes, the speedup-per-core is proportional to the size of the image—though this behaviour does not hold above a relatively low cut-off point. Unlike the Streaming Strategy, which experiences parallelisation costs for regularised deconvolution that it does not experience for serial configurations, the Queuing Strategy achieves similar performances for the regularised deconvolution as it does for the unregularised deconvolution, due to it relying on an underlying set Streaming Strategies which also exhibit this behaviour in their single-core configurations.

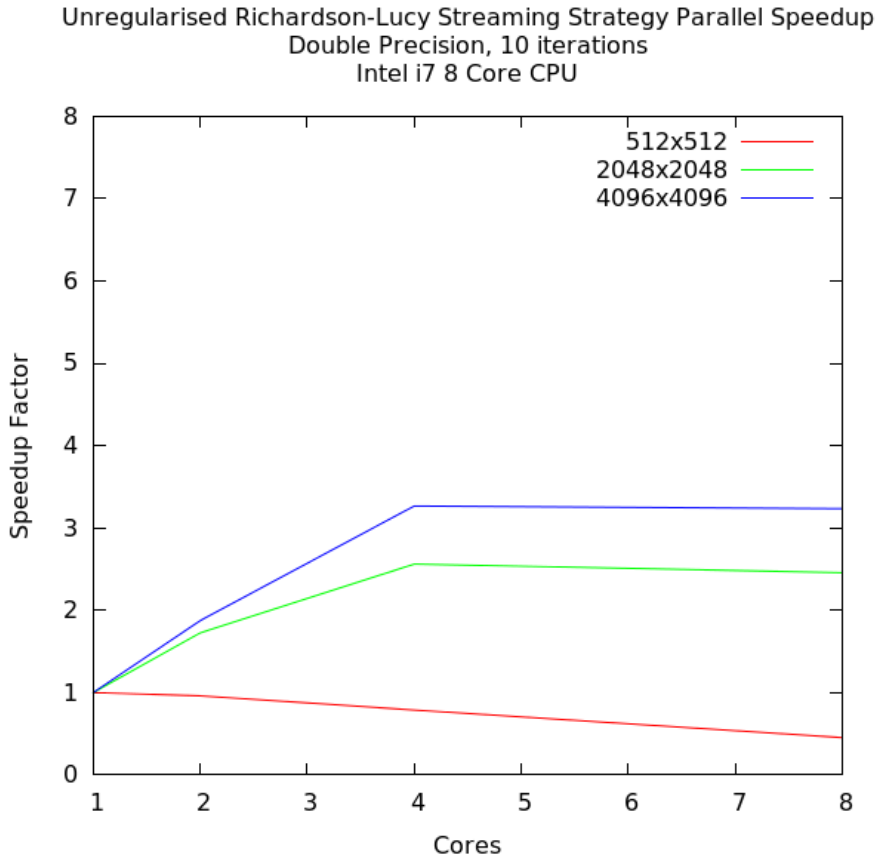



FIGURE 5.29: Unregularised Queuing Per-Core Speedups. The extent of the observed speedup is proportional to the size of the image: for the kilopixel ($512 \times 512$) image, a slowdown is observed; for the $2048 \times 2048$ pixel image the speedup with each additional core is less than that experienced by the $4096 \times 4096$ pixel image. For the megapixel images, a slowdown only occurs with the hyperthreaded 8 core configuration. As the image sizes increase, this 8 core slowdown is no longer observed, as can be seen in Figure 5.27, where the hyperthreaded 8 core configuration begins to achieve a speedup at roughly the 16 megapixel mark. See Section 5.5.4 for further discussion.

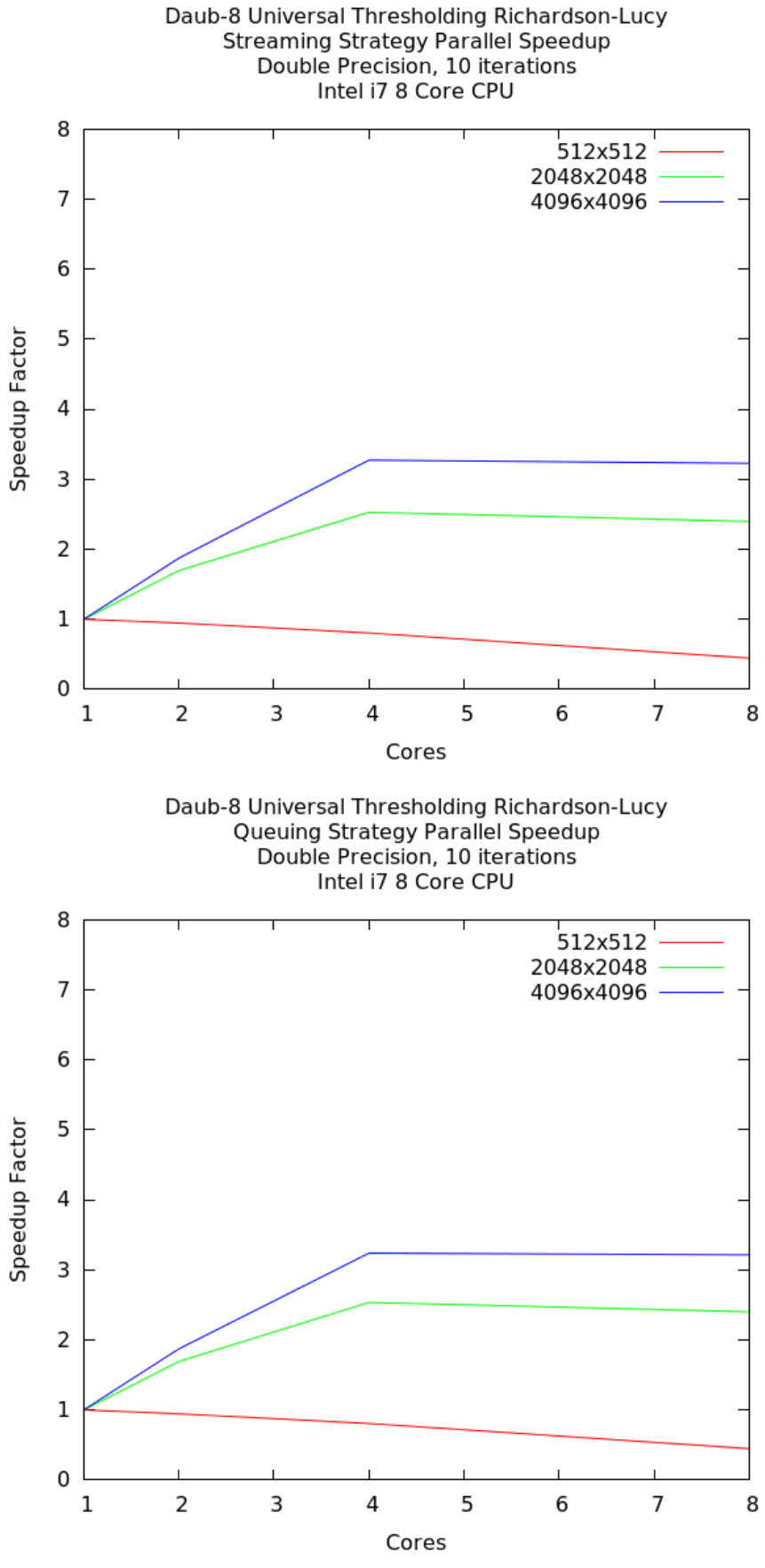



FIGURE 5.30: Regularised Queuing Per-Core Speedups. Like Figure 5.29, the extent of the observed speedup is proportional to the size of the image. Due to the low number of per-iteration synchronisations in the Queuing parallelisation strategy compared to the Streaming parallelisation strategy, the regularised deconvolution experiences superior speedups to those seen in Figure 5.26, and achieves an equivalent speedup with each additional core regardless of the size of the wavelet filter. See Section 5.5.4 for further discussion.

## 5.6 GPU IMPAIR Runtimes

This section details the runtime behaviour of various GPU parallel deconvolution strategies for megapixel sized image data sets. The performance of the algorithms are illustrated in terms of overall runtime, without an estimate of each algorithm's response to more computational cores, due to the GPU model of directly coupling logical and physical cores to the size of the input dataset.

### 5.6.1 GPU Naive

The Naive GPU implementation quickly becomes subject to the memory limitations of the device, and so the IMPAIR GPU streaming implementation imposes a maximum image size of $2048 \times 2048$ pixels. This exact value of this limitation is graphics-card specific, and will increase as hardware improves. Figure 5.31 benchmarks images up to double this size, which became possible late in the development of the IMPAIR software,

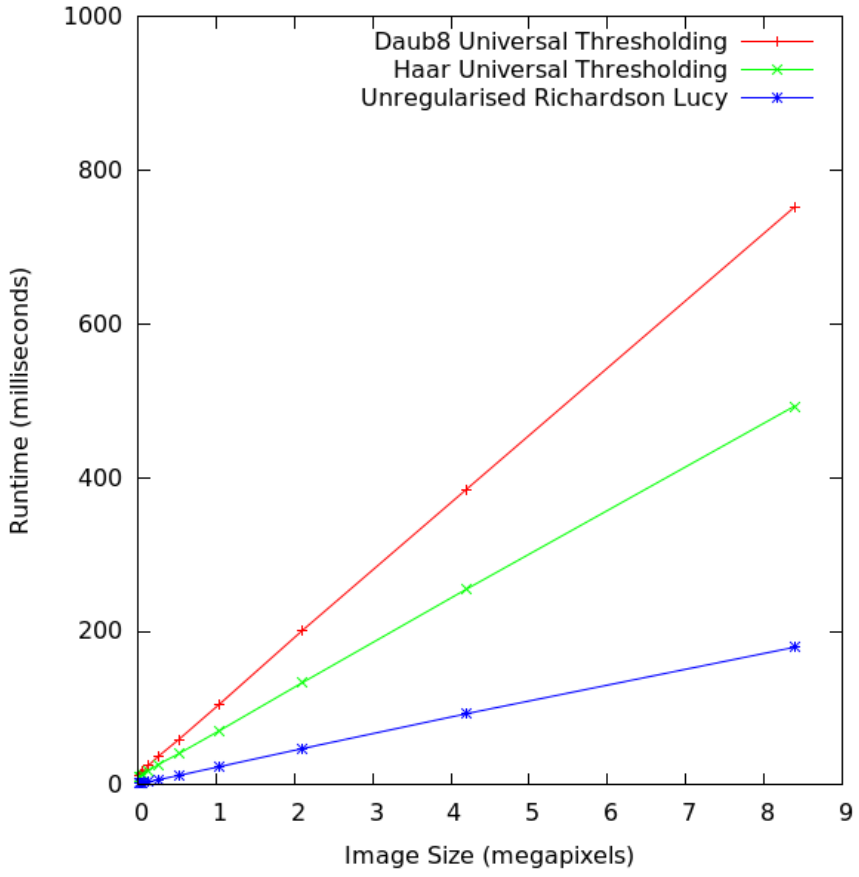



FIGURE 5.31: GPU Naive Megapixel Image Performance. The regularisation overhead of the Haar and Daub-8 Universal Thresholding algorithm amounts to over 50% of the runtime for the GPU Naive strategy implementation, see section 5.6.1 for further discussion.

with an improvement in the underlying wavelet transform algorithm's implementation. Future work will include transitioning the GPU streaming strategy to using this larger internal tile-size, which will halve the number of tiles that must be transferred between the GPU and CPU during runtime.

Like the multicore CPU Naive strategy, the Naive GPU deconvolution delegates all responsibility for parallelisation to the underlying CuFFT [123] and wavelet shrinking libraries, discussed in Chapter 4.

For the range of image sizes that can be run entirely within GPU memory, the GPU Naive strategy scales regularly whether running a regularised or unregularised deconvolution.

The overhead of the wavelet shrinking step in the regularised algorithm runs is in line with the wavelet shrinking the runtimes of the wavelet shrinking operations shown in chapter 4, scaled by the number of iterations the regularised algorithm was run for, with a roughly $+150ms$ and $+250ms$ overhead for the Haar and Daub-8 wavelet regularised operations for a 4 megapixel image.

### 5.6.2 GPU Streaming

The GPU Streaming strategy scales as regularly as the Naive strategy for images in the megapixel range, as can be seen in Figure 5.32. The overall overhead of the wavelet shrinking operation is marginally less than the overhead experienced by the Naive strategy. The extent to which the regularity that the Streaming strategy scales with the size of the input image is discussed further in section 5.7.2.

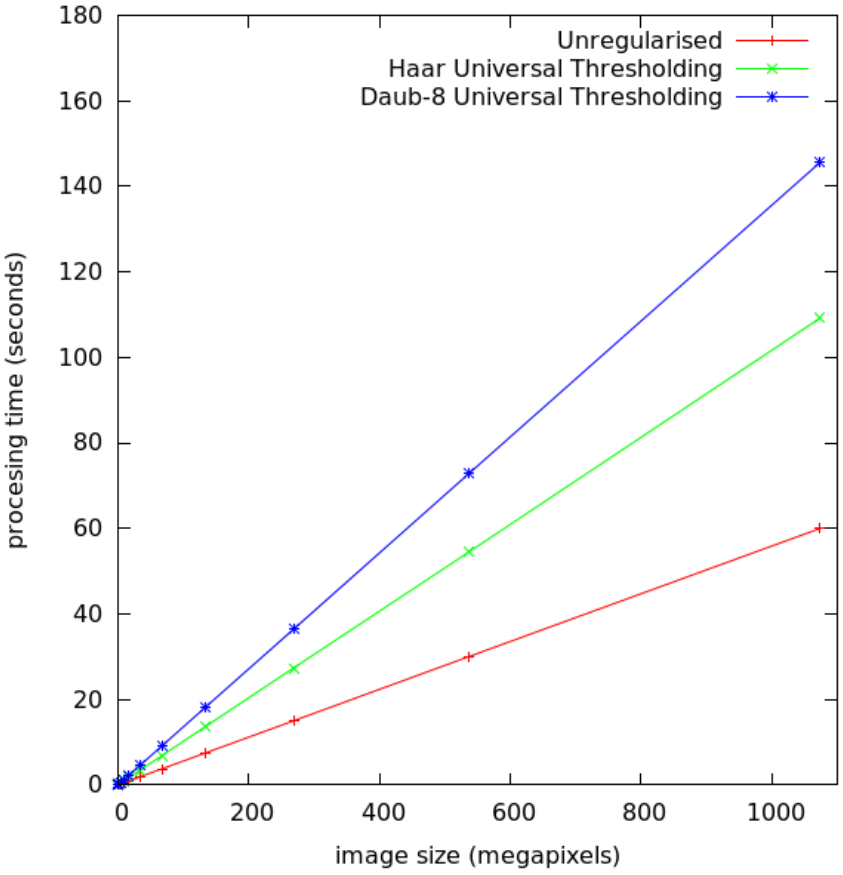



FIGURE 5.32: GPU Streaming Megapixel-Gigapixel Image Performance. The GPU Streaming strategy scales regularly up to gigapixel images, with the regularised algorithm incurring the expected proportional runtime overhead compared to the unregularised algorithm. This behaviour is not observed in the Multicore CPU profiles shown in Figures 5.18 and 5.19. For further discussion on this behaviour, see Section 5.7.2.

## 5.7 CPU-GPU Comparison

The following two sections show the CPU runtimes for megapixel images in terms of the GPU runtimes for the same set of images. The difference in runtimes between the CPU and GPU approaches a constant at roughly the 4 megapixel point (the tilesize for the GPU Streaming Strategy), for both the Naive and Topdown parallelisation strategies. The slowdown is greater for unregularised deconvolution than for the two profiled regularised deconvolution runs, due to the CPU implementations experiencing no significant runtime overhead for the regularisation step over the unregularised algorithm.

The two sections following those show the CPU and GPU runtimes for megapixel images, as the performance of the strategies covered in these sections (CPU Streaming & CPU Queuing) are more regular than the performance of the Naive & Topdown strategies. The GPU and CPU Streaming strategy, and the CPU Queuing strategy are all capable

of scaling 100 megapixel image sizes, and so the deconvolution time for such image sizes is presented.

### 5.7.1 Naive

The comparisons presented in Figures 5.33 and 5.34 of the Naive GPU and CPU parallelisation strategies cover image sizes up to 10 megapixels, due to the GPU memory limitation. For all regularisation algorithms, the 9 megapixel image is deconvolved over 10× faster on the GPU than on the CPU. For lower image sizes, particularly sub-megapixel, the CPU and GPU implementations are more competitive.

For the Universal Thresholding Daub-8 configuration, shown in Figure 5.34, the higher expense of the algorithm incurred on the GPU results in a greatly reduced slowdown with respect to the GPU, compared to the Haar and Unregularised configurations. Both regularised CPU configurations are on the order of twice as competitive with their GPU counterpart as the unregularised configuration.

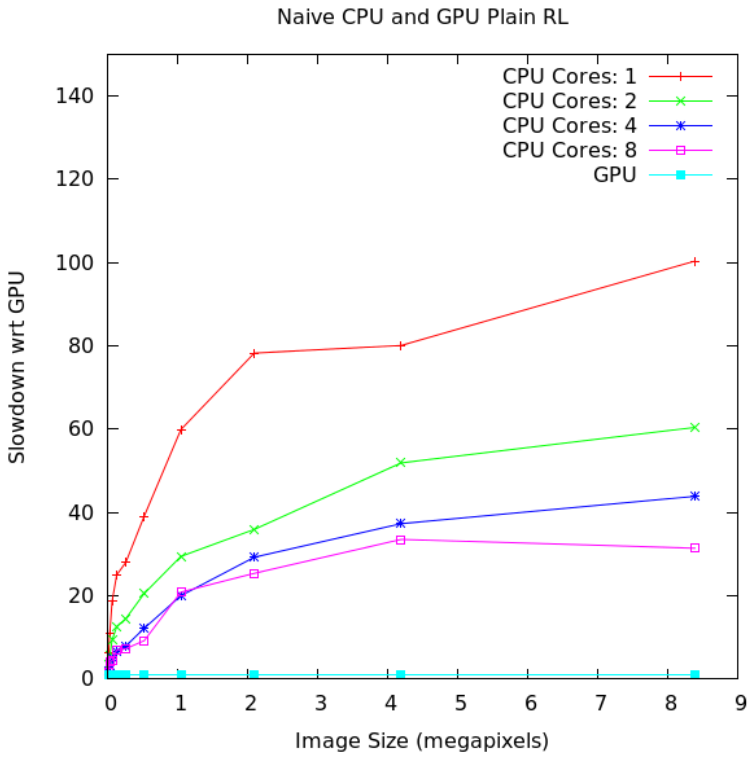



FIGURE 5.33: GPU-CPU Slowdown Naive Unregularised RL. The unregularised Richardson-Lucy runtimes for the Multicore CPU configurations perform between 20—60 slower than the GPU Naive strategy for image sizes over 1 megapixel. This slowdown is dramatically reduced when considering the regularised Richardson-Lucy algorithms, shown in Figure 5.34.

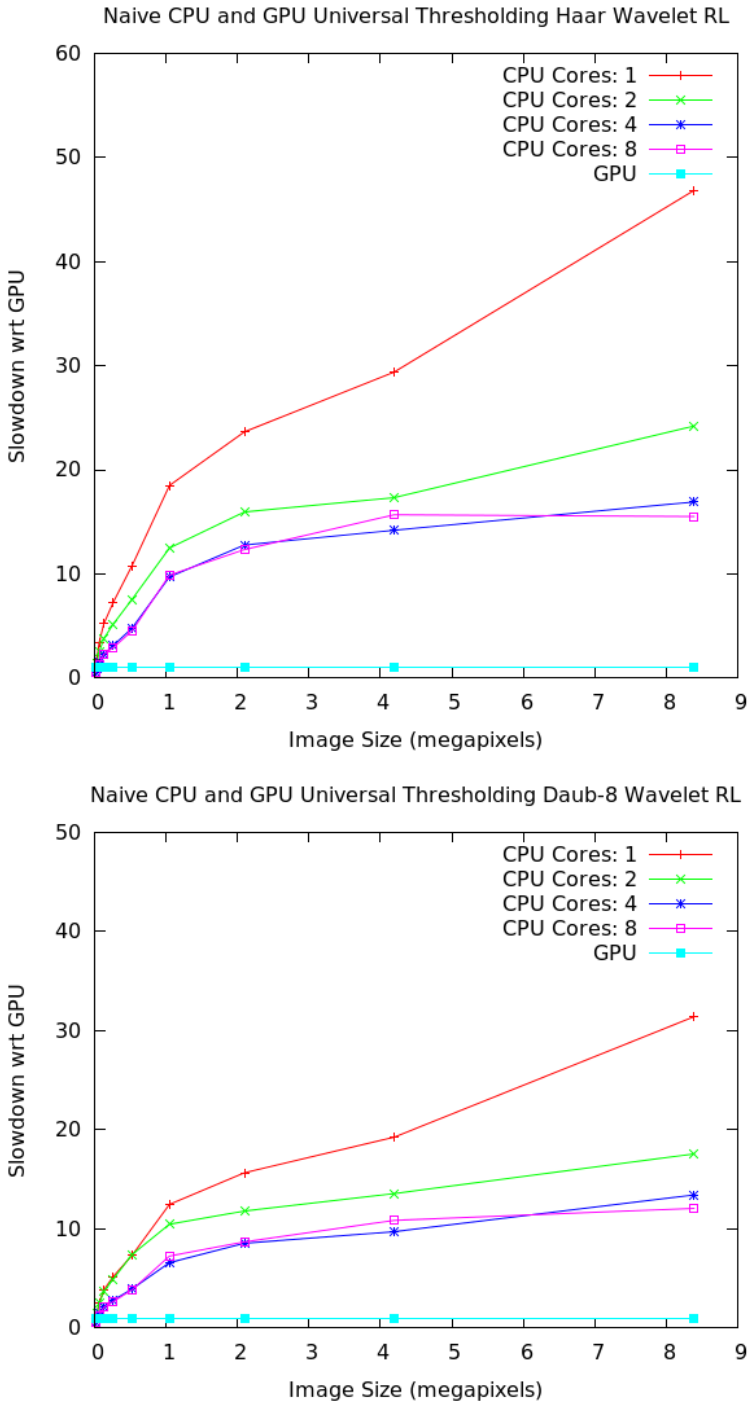



FIGURE 5.34: GPU-CPU Slowdown Naive Universal Thresholding Haar & Daub-8 RL. The regularised Richardson-Lucy runtimes for the Multicore CPU configurations perform between 10—20 slower than the GPU. A comparative improvement shown over the unregularised slowdown shown in Figure 5.33.

### 5.7.2 GPU Streaming & CPU Topdown

The CPU Topdown strategy's sensitivity to image aspect ratio results in a more erratic slowdown factor compared to the Naive strategy's, in particular larger variations between the performance of the full physically threaded runs (4-threaded) and the hyperthreaded runs (8-threaded), as shown in Figure 5.35. In Figure 5.36, for the 4 and 8 threaded runs, both the CPU regularised configurations are less than 5× slower than their GPU counterparts. This is over twice as fast, comparatively, as the slowdown between the CPU and GPU Naive parallelisation strategy.

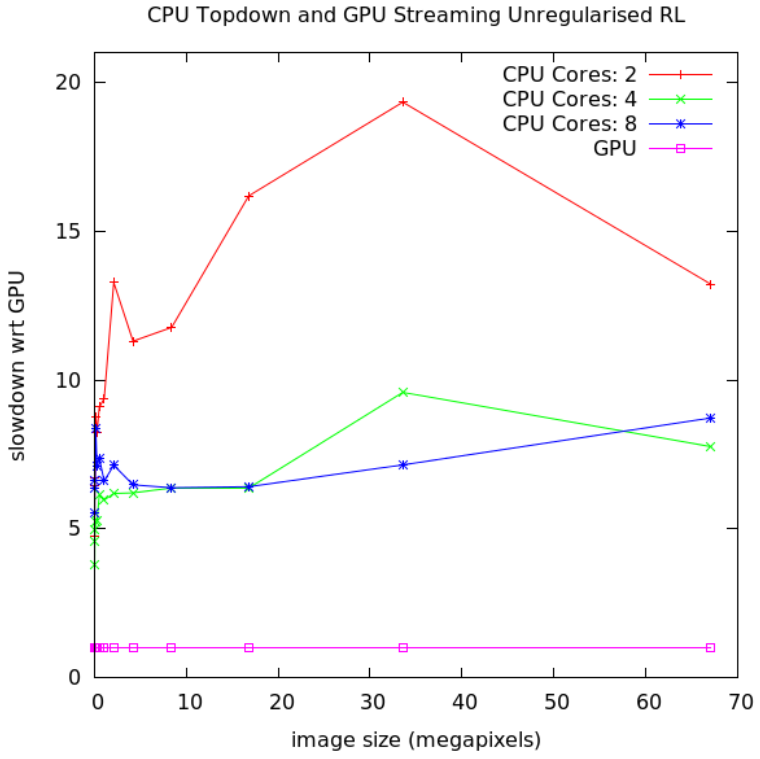



FIGURE 5.35: GPU-CPU Slowdown Unregularised RL. The unregularised Richardson-Lucy runtimes for the Topdown Multicore CPU configurations perform between 5—10 times slower than the GPU Streaming strategy.

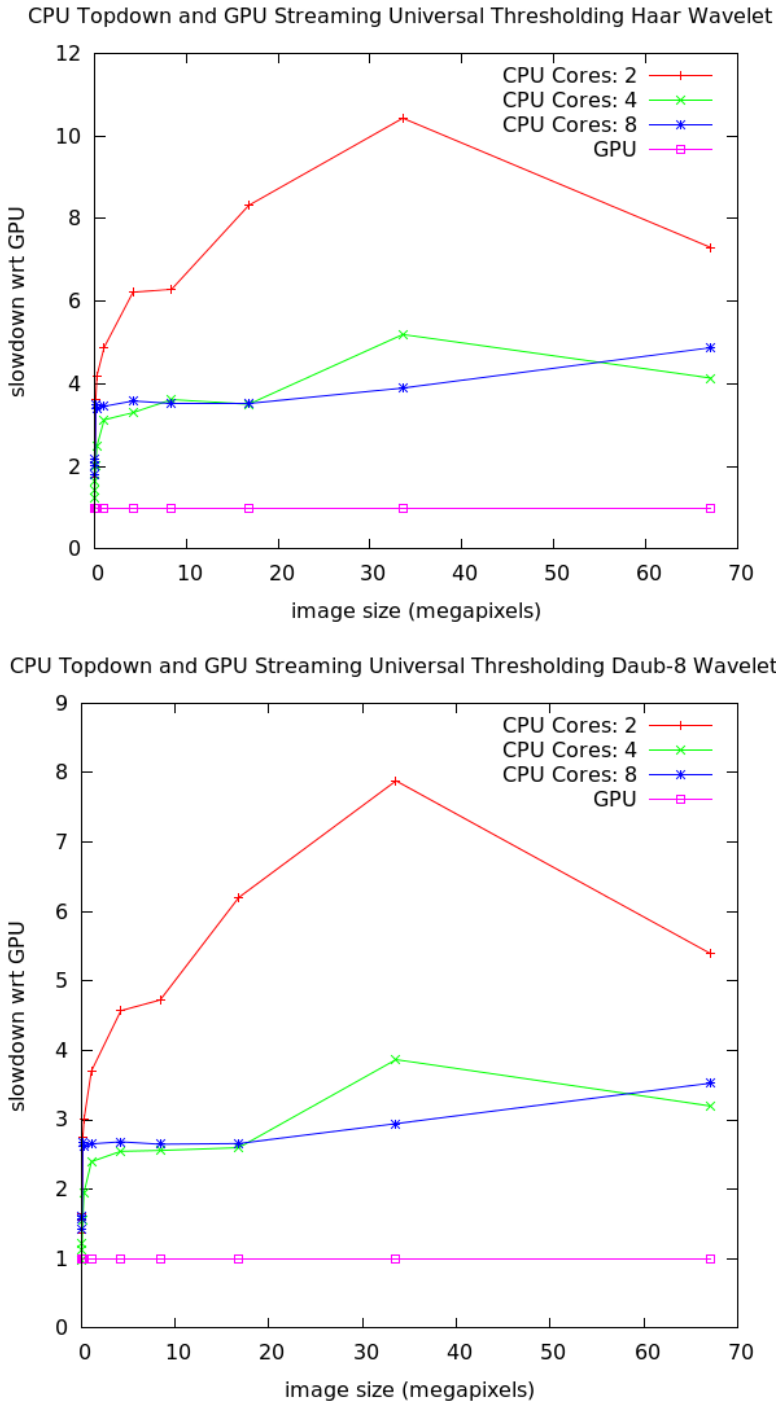



FIGURE 5.36: GPU-CPU Slowdown Universal Thresholding Haar & Daub-8 RL. The regularised Richardson-Lucy runtimes for the Topdown Multicore CPU configurations perform between 2—4 times slower than the GPU streaming strategy for images over 10 megapixels. The comparative slowdown is greater for the less expensive Haar wavelet based algorithm (which experiences a $\times 4$ slowdown), than for the more expensive Daub-8 wavelet based algorithm, (which experiences a $\times 3$ slowdown).

### 5.7.3 GPU Streaming & CPU Streaming & CPU Queuing

The CPU Streaming strategy and CPU Queuing strategy are both capable of scaling to to larger image sizes than the Naive and Topdown strategies, into the 100 megapixel range. Though as is shown in Figures 5.37 and 5.38, the CPU Streaming strategy begins to experience a dramatic slowdown as the size of the input image increases beyond the 300 megapixel threshold. Figure 5.37 shows the comparative performance of the unregularised deconvolution between the fastest CPU configurations and the GPU. Figure 5.38 shows the comparative performance of the Haar and Daub-8 regularised deconvolution between the fastest CPU configurations and the GPU, during which the Queuing strategy achieves an equivalent runtime regardless of the length of the wavelet filter in line with what was seen in Figure 5.28.

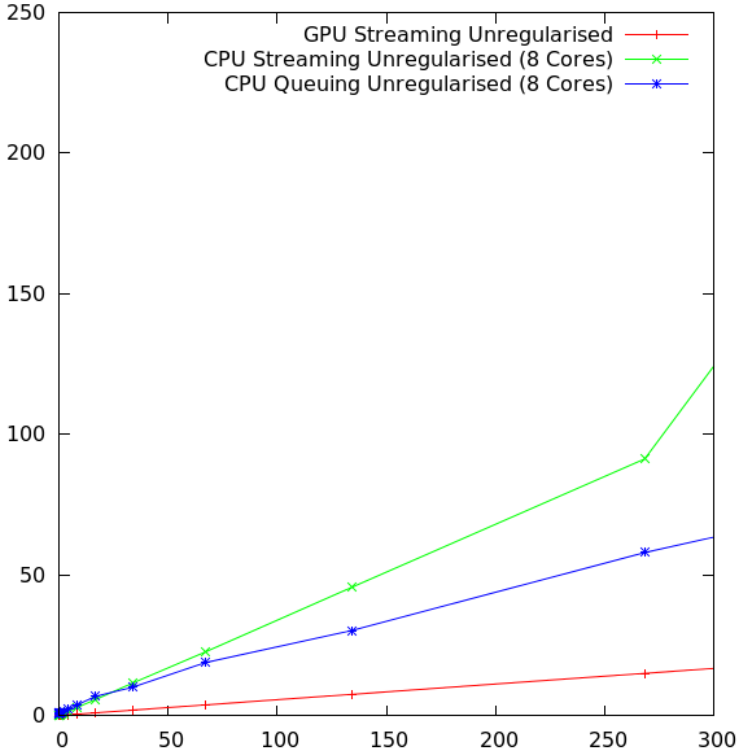



FIGURE 5.37: GPU-CPU Unregularised 100 megapixel deconvolution. The Streaming and Queuing Strategies' lower memory requirements allow them to scale to images in the hundred-megapixel range, with runtimes of less than 10× the runtime of the GPU Streaming Strategy. See Section 5.7.3 for further discussion.

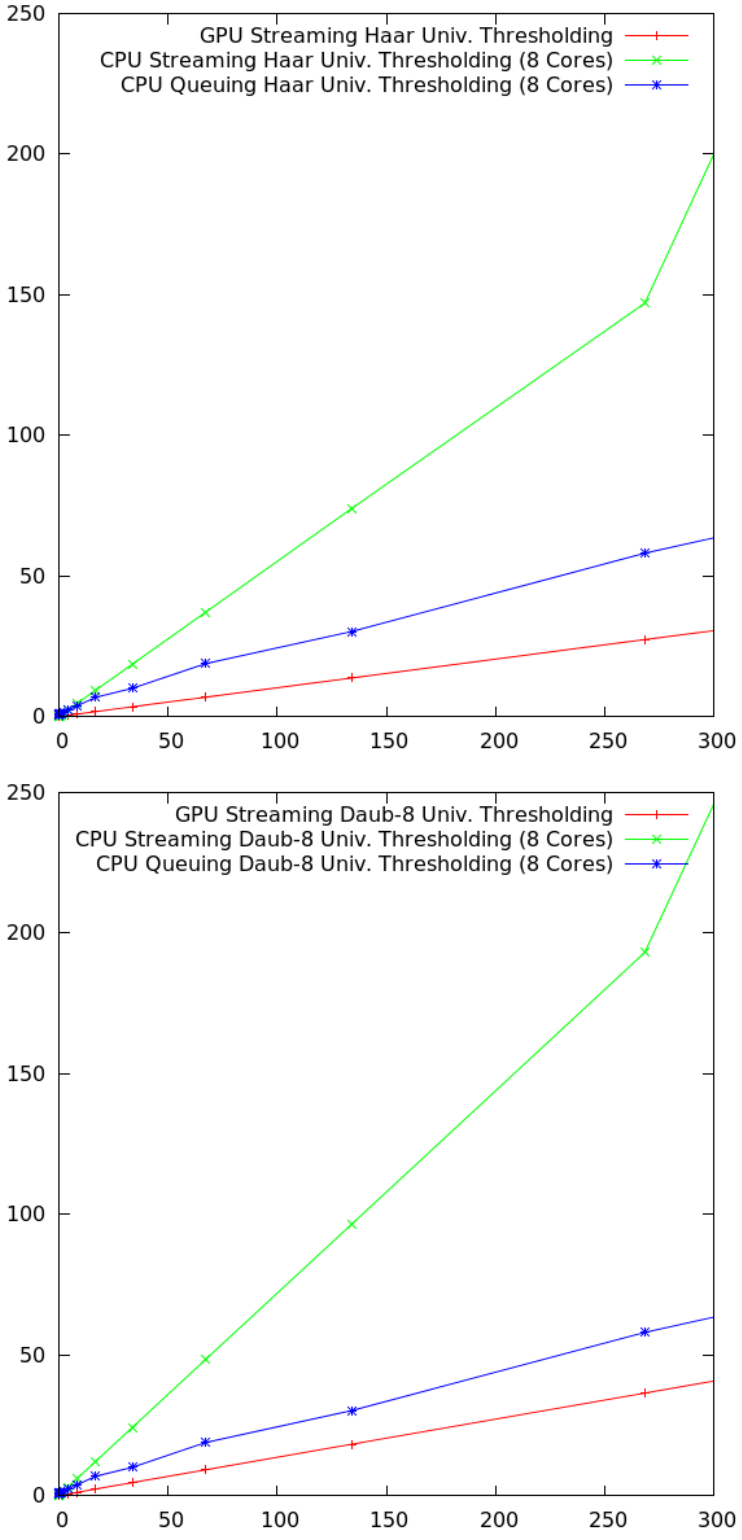



FIGURE 5.38: GPU-CPU Regularised 100 megapixel deconvolution. The Streaming and Queuing Strategies' lower memory requirements allow them to scale to images in the hundred-megapixel range. For the regularised deconvolution, both the GPU and CPU Streaming Strategies experience a slowdown as the size of wavelet filter increases, due to the increase in the number of per-iteration synchronisations that must occur to accommodate this. The CPU Queuing Strategy maintains the same performance as seen in Figure 5.37. For the Daub-8 case, the slowdown of the GPU leads to the CPU Queuing Strategy running in less than 2× the runtime of the GPU Streaming Strategy. See Section 5.7.3 for further discussion.

## 5.8 Conclusion

We have implemented Multicore CPU and GPU regularised and unregularised Richardson-Lucy deconvolution libraries, parallelised according to three related strategies, and presented their runtime performances, and confirmed that the implementations perform comparably with the original implementations of these algorithms from an image restoration perspective. We have found that for the Multicore CPU algorithm runtimes, the addition of the wavelet shrinking step does not incur any significant overhead when compared to the Multicore unregularised algorithm runtime, while the GPU algorithm runtimes present the expected overhead of the extra computational expense. If the CPU Multicore algorithms followed the same behaviour as the GPU algorithms, a +20 to +50 second overhead should be visible for images in the region of 70 megapixels in size. Due to the coarse granularity of the CPU Multicore algorithms compared to the lock-step GPU algorithms, the Multicore CPU has more freedom to schedule the computation across its cores in an advantageous manner. The likelihood that this behaviour accounts for the invisible cost of the regularised algorithm over the unregularised algorithms on the Multicore CPU is investigated further in Chapter 6.

# Chapter 6

# Analysis

## 6.1 Introduction

Chapter 5 illustrated the runtime performance of the IMPAIR deconvolution algorithms for the CPU and GPU platforms. While the GPU performance scales regularly with the input image size, the CPU performance is more erratic in the response both to doubling the number of threads and to doubling the image size. This analysis aims to address the possible causes of the observed irregular occasions of parallel slowdown of the benchmarked CPU deconvolution, and to identify the extent to which this behaviour is due to the interaction with the cache hierarchy.

Section 6.2 details the memory requirements of each of the parallelisation strategies IMPAIR employs with respect to the size of the input image data and number of parallel cores. This defines the upper limit of the image data that IMPAIR can process on a given machine, but does not have an immediate effect on the overall latency or throughput of an image deconvolution.

Section 6.3 details the behaviour of each method with respect to the memory hierarchy (either RAM, $L_3$, $L_2$, $L_1$ caches for the CPU strategies, or RAM, Global Memory, Shared Memory, Register Memory for the GPU strategies). Profiling data from both simulated (using the Cachegrind [129] simulator) and live runs (using the i7's hardware counter registers) for each strategy are presented and discussed.

## 6.2 Memory Footprint

For images of size $M \times N$ pixels deconvolved with a PSF kernel of size $K \times L$, the total memory footprint of the Naive strategy deconvolution is as follows:

| **Naive Strategy** | |
|---|---|
| Input Image: | $N \times M$ pixels |
| Output Image: | $N \times M$ pixels |
| Estimate Image: | $N \times M$ pixels |
| Temporary Image: | $N \times M$ pixels |
| PSF Image: | $N \times M$ pixels |
| Convolution Buffer: | $N \times M$ pixels |
| **Unregularised Total:** | $6(M \times N)$ **pixels** |
| DWT Buffer: | $N \times M$ pixels |
| Shrinking Buffer: | $N \times M$ pixels |
| **Universal-Thresholding Regularised Total:** | $8(M \times N)$ **pixels** |

The number of cores the deconvolution is parallelised over does not influence the memory footprint of the Naive strategy in a per-pixel fashion. The memory footprint of the Naive Strategy is not influenced by the size of the PSF kernel image, as long as the input image is cyclically deconvolved. A zero padded deconvolution can be achieved by manually zero padding the input image to be of size $(N+2K) \times (M+2L)$ pixels. In this case, the total memory footprint of the deconvolution becomes $6(N+2K) \times (M+2L)$ pixels for the unregularised deconvolution, and $8(N+2K) \times (M+2L)$ pixels for the universal-thresholding regularised deconvolution.

For images of size $M \times N$ pixels deconvolved with a PSF kernel of size $K \times L$ over $P$ parallel cores, the total memory footprint of the Topdown strategy deconvolution is as follows:

| **Topdown Strategy** | |
|---|---|
| Input Image: | $N \times M$ pixels |
| Output Image: | $N \times M$ pixels |
| PSF Image: | $(PN+2K) \times (\frac{M}{P}+2L)$ pixels |
| Estimate Image: | $(PN+2K) \times (\frac{M}{P}+2L)$ pixels |
| Temporary Image: | $(PN+2K) \times (\frac{M}{P}+2L)$ pixels |
| Convolution Buffer: | $(PN+2K) \times (\frac{M}{P}+2L)$ pixels |
| **Unregularised Total:** | $4 \times (PN+2K) \times (\frac{M}{P}+2L) + 2(N \times M)$ **pixels** |
| DWT Buffer: | $(PN+2K) \times (\frac{M}{P}+2L)$ pixels |
| Shrinking Buffer: | $(PN+2K) \times (\frac{M}{P}+2L)$ pixels |
| **Universal-Thresholding Regularised Total:** | $6 \times (PN+2K) \times (\frac{M}{P}+2L) + 2(N \times M)$ **pixels** |

The division of the image into $P$ $(N+2K) \times (\frac{M}{P}+2L)$ pixel tiles is the worst-case of the Topdown strategy's memory footprint, square numbers of parallel cores are divided

into $P$ $(\frac{N}{\sqrt{P}} + 2K) \times (\frac{M}{\sqrt{P}} + 2L)$ pixel tiles, and the 8 parallel core case is handled by $(\frac{N}{\sqrt{\frac{P}{2}}} + 2K) \times (\frac{M}{\sqrt{2P}} + 2L)$ by IMPAIR.

For images of size $M \times N$ pixels deconvolved with a PSF kernel of size $K \times L$, processed as a sequence of $Q \times R$ pixel tiles, the total memory footprint of the Streaming strategy deconvolution is as follows:

| **Streaming Strategy** | |
|---|---|
| Input Image: | $N \times M$ pixels |
| Output Image: | $N \times M$ pixels |
| PSF Image: | $(Q + 2K) \times (R + 2L)$ pixels |
| Tile Image: | $(Q + 2K) \times (R + 2L)$ pixels |
| Estimate Image: | $(Q + 2K) \times (R + 2L)$ pixels |
| Temporary Image: | $(Q + 2K) \times (R + 2L)$ pixels |
| Convolution Buffer: | $(Q + 2K) \times (R + 2L)$ pixels |
| **Unregularised Total:** | $5 \times (Q + 2K) \times (R + 2L) + 2(N \times M)$ **pixels** |
| DWT Buffer: | $(Q + 2K) \times (R + 2L)$ pixels |
| Shrinking Buffer: | $(Q + 2K) \times (R + 2L)$ pixels |
| **Universal-Thresholding Regularised Total:** | $7 \times (Q + 2K) \times (R + 2L) + 2(N \times M)$ **pixels** |

For the GPU implementations, the $Q$ and $R$ values are fixed by the hardware limitations of the test machine's GPU card to $2048 \times 2048$.

The memory footprints of the above parallelisation strategies determine the maximum size of an image that can be deconvolved by each as a process must have access to enough memory to hold the intermediate buffers in memory for the deconvolution operation. The interaction of each of these strategies with the memory hierarchy is more involved, as some buffers are more frequently accessed than others. Furthermore, the operations performed on each buffer follow different patterns of access, which will have their own effect on the overall performance. As the IMPAIR deconvolution algorithms operate in a straightforward looping pattern, with no unpredictable control flow branching, the number of cache references that can be expected to occur can be estimated from these memory footprints.

By implementing and profiling a set of parallelisation strategies with these global memory footprints, the GPU and CPU platforms' behaviours can be investigated for processes with smaller and larger memory footprints and correspondingly lesser and greater likelihoods of local reuse, regardless of the memory access patterns of the operations on the individual buffers. For the parallelisation strategies with memory footprints that do not depend on the number of cores over which the algorithm is

being run, the overall memory access counts can be expected to remain constant as the algorithm is profiled with 1 to 8 threads. For the Topdown parallelisation strategy, whose memory footprint is influenced by the number of cores that are in use, a corresponding increase or decrease in the absolute memory access counts is expected to be observed.

## 6.3 Memory Hierarchy

The Intel i7 Multicore CPU [130] cache contains three layers, the lowest-level cache ($L_3$) is shared amongst all CPU cores, while the intermediate ($L_2$) and highest ($L_1$) level caches are shared between each pair of hyperthreaded logical cores. All cache lines are present in the $L_3$ cache, and duplicated in the $L_2$ and $L_1$ caches, limiting the total available memory in the cache hierarchy to the size of the $L_3$ cache. This allows the $L_3$ cache to be used to share memory between physical processor cores with less expense than loading a cache line from main memory, but more of an expense than unshared $L_3$ cache reference, as any changes to the cache line must be first committed to the $L_3$ cache before they can be exposed to another physical core. This lesser expense allows the $L_3$ cache to be used to share read-only memory between physical cores with the cost of an $L_1$ or $L_2$ cache miss. As the cost of a read from a shared cache line is marginally more expensive than a read from the $L_3$ cache, future work will included an investigation into whether it is better to duplicate the read-only data among all cores, rather than share access.

The i7's cache hierarchy follows a *least-recently-used* policy, where the longer a cache line has been resident in the cache, the higher the likelihood that it will be replaced by a new cache line. As the cache is shared between multiple cores, with multiple threads of execution simultaneously using the cache, whether or not any cache line is the least recently used cannot be determined by examining the behaviour of a single thread of execution, or a single core [46].

For threads that operate on sequential regions of memory, sequential cache-lines are loaded automatically from main memory by the hardware prefetcher to prevent frequent stalls due to $L_3$ cache misses [131]. This lowers the runtime cost of an $L_3$ cache miss for long single dimensional arrays, as the anticipated misses are performed in parallel with the SIMD register calculations. The i7 hardware prefetcher can also detect when a sequence of reads with a regular offset are performed, and can similarly prefetch. This reduces some of the penalties for accessing image data by column vectors rather than row vectors, but will never reduce the penalty to as low as that of accessing image data by row vectors, since a single cache fetch 'prefetches' the next several pixel's data, while a stride-prefetch will only 'prefetch' the next pixel's data.

Due to the prefetching behaviour of the $L_3$ cache, a thread which operates on rows of pixels that span a number of cache lines is blocked for less time on the transition between two cache lines than a thread which is not operating on sequential data, as the process of fetching the following cache line from main memory has either already begun, or already completed, by the time the transition occurs. This is a similar behaviour to the coalesced global memory access that occurs when the threads in a GPU warp reference a sequential region of global GPU memory simultaneously. Consequently, operations on rows of pixels on the CPU can be expected to find benefit from the same memory access patterns as the GPU; the cost of proceeding to the next pixel in an image row is on average lower than the cost of $L_3$ cache miss, while the cost of accessing the next pixel in an image column can vary unpredictably, up to the point of incurring the maximum penalty (as is the case with an $L_3$ cache miss, or non-coalesced GPU global memory access).

The memory access pattern of the deconvolution algorithm implementations is such that the runtime cache miss events will be either capacity misses (where the amount of closely-located memory being frequently accessed exceeds the available space at a cache level) or conflict misses (where the cache level has not been filled, but all the possible slots for the required line are in use). The streaming nature of the batch arithmetic implementations means that the cache will fill with recently-used data that will not be re-used. These previously used lines can be safely ejected, though this will result in the cache constantly filling rather than reworking the same subset of cached data. The streaming nature of the operations may result in a high number of collision-based cache misses [132], due to the eight-way associativity of the cache, and the requirement that each thread operate on the data from several image buffers. Depending on the alignment of this data with respect to memory page boundaries, the referencing of cache lines from separate threads may cause the recently-used lines of the other thread to be rejected. This behaviour is prone to occur as the number of working threads grows above 2, as procedures within the deconvolution operation require the use of 3 buffers simultaneously. Depending on the alignment of this data with respect to memory page boundaries, this behaviour will be guaranteed as the number of working threads is equal to the N-way associativity of the cache—in the case of the Intel i7 processor, this happens to be the same as the maximum number of threads. This kind of maximal-threading is only suitable for constant scaling or thresholding of pixel data under these memory alignment circumstances. This worst case behaviour is the result of a design flaw of the SIMD vectoring parallelisation of the IMPAIR Naive strategy, that underlies all others, and future work will include adhering to a memory access pattern that avoids this worst-case behaviour in the event of page-aligned image data.

The Intel i7 Multicore CPU provides hardware cache counters for counting the total number of $L_1$ references, $L_2$ references (as $L_1$ cache misses), and $L_3$ references. Since

the $L_1$ cache is divided into separate instruction and data caches, independent counters are provided for these. The $L_3$ cache does not distinguish between instruction or data cache lines, and so these counters include speculative instruction fetch measurements.

The values presented in Tables 6.1, 6.2, and 6.3 are presented in terms of millions-of-counts, and are the averages of 10 repeated runs, with the largest observed standard deviation of any of these values being at approximately 5%. A cache reference in these tables refers to an attempt to access memory in a level of cache. An $L_1$ reference is an attempt to access a memory address from the $L_1$ cache. This is a count of every memory access that was performed over the course of the deconvolution. An $L_2$ reference is an attempt to access a memory address from the $L_2$ cache, because it was not present in the $L_1$ cache. This is a count of the number of $L_1$ cache-misses that occurred over the course of the deconvolution. An $L_3$ reference is an attempt to access a memory address from the $L_3$ cache, because it was not present in either the $L_1$ or $L_2$ caches. This is a count of the number of $L_2$ cache-misses that occurred over the course of the deconvolution. The i7 processor does not provide hardware counters for physical memory accesses— which correspond to $L_3$ cache misses— so an estimate of the number of physical memory accesses per deconvolution is presented based on the Cachegrind profiler, which simulates a 3 level inclusive cache similar to the i7 processors. Neither the i7 processor nor Cachegrind provide counts for $L_1$ references that miss in the physical core's local $L_1$ cache, but hit in another core's $L_1$ cache. This scenario does not incur the same penalty as an $L_1$ miss that hits in the $L_2$ or lower caches, but it is more expensive than a hit in a physical core's local $L_1$ cache. The design of the IMPAIR DWT transform is such that this scenario will occur when sharing the wavelet-filter coefficients between physical cores.

There are four categories of cache-miss [2] that the Intel i7 Multicore Processor is subject to:

- Compulsory Misses

  Compulsory misses occur the first time an area of main memory is accessed, and some number will always occur.

- Capacity Misses

  Capacity misses occur when the entire cache is not large enough to hold all previously accessed areas of main memory, and so main memory (or a higher level of the cache hierarchy) must be accessed again.

- Conflict Misses

  Conflict misses occur if a cache has a restricted number of locations in which it can store an area of main memory, and the number of these locations is too few to

hold all previously accessed areas that are associated with it, and so main memory (or a higher level of the cache hierarchy) must be accessed again.

Conflict misses do not occur in fully-associative caches, which have an unrestricted number of locations where an area of main memory can be stored.

- Coherence Misses

  Coherence misses occur if a cache holds an out of date copy of an area of main memory that has been changed since it was last accessed, and so main memory (or a higher level of the cache hierarchy) must be accessed again.

  Coherence misses only occur in parallel systems with a shared cache, like multiprocessor computers or multicore CPUs.

Since the algorithm is constructed using repeated sequences of in-order passes over pairs of buffers, after the first iteration all compulsory cache misses will have already occured. As the parallel implementation does not rely on any shared writable memory, cache lines can be present in multiple $L_1$ and $L_2$ caches without incurring any cost for maintaining the coherency of the copies with one another, as unmodified shared cache lines are flagged as such. An investigation into whether marking these regions of memory explicitly read-only has any noticeable effect of the runtime could be conducted.

The cache misses presented below will be predominatly capacity cache misses or conflict cache misses, depending on the number of cores in use. Conflict cache misses will be unlikely to occur in the single core case, as the 8-way associative cache can accommodate two cache lines that collide without evicting either of them. This behaviour should also holds for the two core case, where two threads will each attempt to load two lines that all conflict with one another, taking up four of the available eight 'slots' in $L_3$ cache. The four core case will fill all available 'slots' in this worst case scenario, and the eight core case will result in continuous evictions as the cores progress through the pair of buffers.

**Naive Plain Deconvolution**

(Millions of line-references and percentage of higher-level's line-reference count)

| **8 Megapixel, 2:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 2500 | | 2671 | | 2676 | | 2662 | |
| $L_2$ Cache Load References | 1364 | (55%) | 1371 | (51%) | 1371 | (51%) | 1578 | (59%) |
| $L_3$ Cache Load References | 456 | (33%) | 468 | (34%) | 512 | (37%) | 539 | (34%) |
| $L_1$ Cache Store References | 795 | | 793 | | 793 | | 798 | |
| $L_2$ Cache Store References | 613 | (77%) | 632 | (79%) | 629 | (79%) | 721 | (90%) |
| $L_3$ Cache Store References | 387 | (63%) | 389 | (62%) | 395 | (63%) | 412 | (57%) |

| **16 Megapixel, 1:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 5276 | | 5618 | | 5633 | | 5579 | |
| $L_2$ Cache Load References | 2788 | (53%) | 2830 | (50%) | 2830 | (50%) | 3179 | (57%) |
| $L_3$ Cache Load References | 894 | (32%) | 958 | (34%) | 1063 | (38%) | 1107 | (35%) |
| $L_1$ Cache Store References | 1677 | | 1675 | | 1675 | | 1680 | |
| $L_2$ Cache Store References | 1263 | (75%) | 1213 | (72%) | 1258 | (75%) | 1429 | (85%) |
| $L_3$ Cache Store References | 794 | (63%) | 798 | (66%) | 810 | (64%) | 867 | (61%) |

Table 6.1:

**Naive Haar Universal Thresholding Deconvolution**

(Millions of line-references and percentage of higher cache level's line-reference count)

| **8 Megapixel, 2:1 Image** | Threads:1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 2498 | | 3474 | | 3483 | | 3474 | |
| $L_2$ Cache Load References | 1341 | (31%) | 1860 | (54%) | 1861 | (53%) | 2094 | (60%) |
| $L_3$ Cache Load References | 433 | (32%) | 708 | (38%) | 791 | (43%) | 821 | (39%) |
| $L_1$ Cache Store References | 793 | | 1025 | | 1025 | | 1030 | |
| $L_2$ Cache Store References | 632 | (79%) | 891 | (87%) | 894 | (87%) | 1017 | (98%) |
| $L_3$ Cache Store References | 388 | (61%) | 504 | (57%) | 510 | (57%) | 524 | (52%) |

| **16 Megapixel, 1:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 5281 | | 7199 | | 7216 | | 7192 | |
| $L_2$ Cache Load References | 2860 | (54%) | 3769 | (52%) | 3777 | (52%) | 4199 | (58%) |
| $L_3$ Cache Load References | 962 | (34%) | 1402 | (37%) | 1577 | (42%) | 1672 | (39%) |
| $L_1$ Cache Store References | 1675 | | 2133 | | 2137 | | 2143 | |
| $L_2$ Cache Store References | 1164 | (69%) | 1784 | (83%) | 1803 | (84%) | 2026 | (95%) |
| $L_3$ Cache Store References | 795 | (68%) | 1025 | (57%) | 1036 | (57%) | 1087 | (54%) |

Table 6.2

**Naive Daub-8 Universal Thresholding Deconvolution**
(Millions of line-references, percentages of higher cache level's line-reference count)

| **8 Megapixel, 2:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 2501 | | 4680 | | 4699 | | 4698 | |
| $L_2$ Cache Load References | 1384 | (55%) | 2694 | (58%) | 2700 | (57%) | 3059 | (65%) |
| $L_3$ Cache Load References | 476 | (34%) | 714 | (27%) | 805 | (30%) | 939 | (31%) |
| $L_1$ Cache Store References | 794 | | 1361 | | 1362 | | 1372 | |
| $L_2$ Cache Store References | 612 | (77%) | 1234 | (90%) | 1232 | (90%) | 1450 | (106%) |
| $L_3$ Cache Store References | 389 | (64%) | 506 | (41%) | 510 | (41%) | 590 | (41%) |

| **16 Megapixel, 1:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 5280 | | 9621 | | 9652 | | 9627 | |
| $L_2$ Cache Load References | 2782 | (53%) | 5433 | (56%) | 5418 | (56%) | 6110 | (63%) |
| $L_3$ Cache Load References | 793 | (29%) | 1398 | (26%) | 1562 | (29%) | 1775 | (29%) |
| $L_1$ Cache Store References | 1674 | | 2808 | | 2809 | | 2816 | |
| $L_2$ Cache Store References | 1253 | (75%) | 2461 | (88%) | 2507 | (89%) | 2843 | (100%) |
| $L_3$ Cache Store References | 891 | (71%) | 1023 | (42%) | 1032 | (41%) | 1086 | (38%) |

TABLE 6.3: Note regarding the 106% and 100% reference ratios:
The $L_2$ store references for the 8 threaded cases of both images amount to a greater number of line-references than the $L_1$ store references. The absolute counts for these images are 10976 million $L_1$ word store-references for the 8 megapixel image, and 22528 million $L_1$ word store-references for the 16 megapixel image. In the 8 threaded configurations it appears that the assumption that 8 $L_1$ references will correspond to at most one $L_2$ reference is incorrect, which could be due to either the alignment of the words within a cache line— where what is treated as a continuous run of 8 words by the software actually straddles two cache lines, or due to a canceled or restarted operations, a situation that is likely to occur in the 8 threaded case, when both the logical threads of each physical core are vying for the same resources.

Table 6.1 presents the cache reference data for 10 iterations of unregularised deconvolution parallelised according to the Naive strategy. The $L_1$ counts record the number of cache-line references made by the processor, while the $L_2$ counts reflect the number of searches of the $2^{nd}$ level cache for a line, and the $L_3$ counts reflect the number of searches in the $3^{rd}$ level. $L_1$ line-reference counts are reported as units of pixel data that are being operated on in order using the intel SIMD registers (which correspond to a single cache line in size). The $L_1$ line-reference counts are estimated from the profiled $L_1$ word-reference counts, assuming all memory accesses were to the neighbouring eight-consecutive words in a cache line.

The $L_1$ cache experiences less than double the references of the $L_2$ cache, meaning more than half the memory references made by the process were not found in the $L_1$ cache. This $L_1$ hit rate is in keeping with the high spatial-locality but low temporal-locality of the Naive strategy, where the ordered spatially-local references are bundled into a single batch. In this scenario, the prefetching of a successive $L_1$ cache line can explain why each $L_1$ line-reference is not a compulsory miss. The $L_2$ cache experiences only $\times 2$–$\times 3$ more references than the $L_3$ cache, meaning 1 out every 2–3 memory references not found in the $L_1$ cache were not found in the $L_2$ cache either. This low hit rate is similarly in keeping with the high spatial-locality but low temporal-locality exhibited in the $L_1$ access pattern. For both the 8 and 16 Megapixel images, the load memory references have a better chance of hitting the $L_1$ or $L_2$ caches than the store references. This is in keeping with the higher rate of load-accesses compared to store-accesses, and the behaviour where no memory that has been recently read will be soon written to, and no memory that was recently written will be soon read. Together, these give the read operations proportionally more influence over the population of the cache than the write operations. The behaviour is also compatible with a cache hierarchy that prioritises caching read-only lines over writable lines, or prioritises read operations over write operations.

The data in Tables 6.2 and 6.3 show the cache-line reference counts under the same configurations for the regularised deconvolution. The number of $L_2$ stores range from between 60%–100% of the number of $L_1$ stores, suggesting that almost every new line that was required was not prefetched into $L_1$, as it appears to be in the unregularised deconvolution profiles. The counts for $L_2$ are less than $\times 2$ the counts for $L_3$, meaning over half the required lines were not available in $L_2$, and were searched for in $L_3$. It seems reasonable to project that this pattern holds for the $L_3$–main-memory counts, with over half the attempts to pull a line from $L_3$ falling back to main memory access. Single-threaded cache profiles with the Valgrind simulator give main memory references of 57% for the unregularised, Haar and Daub-8 $4096 \times 4096$ images and 60% for the

$4096 \times 2048$ images, though the absolute counts gathered differ greatly from the profiles shown in these tables.

The multithreaded configurations experience an increase in the the number of $L_1$ references compared to the single threaded configuration, for all image sizes and wavelet filters. This overhead does not appear to depend on the number of threads, only on whether or not the multithreaded functions were called.

The ratio of load-accesses to store-accesses is not constant, suggesting the presence of speculative or restarted cache accesses in the profiled $L_1$ counts. As the $L_3$ cache is shared between the $L_1$ instruction and data caches, all $L_3$ counts also include load-accesses for the instruction caches.

For the 8 megapixel images, this overhead is approximately 700 million counts for the Haar filter, and 1700 for the Daub-8 filter. For the 16 megapixel images, this overhead is approximately 1500 million counts for the Haar filter, and 3000 million counts for the Daub-8 filter. Though not visible from these tables, the extent of this overhead varies with the number of iterations performed, suggesting it is associated with the repeated creation and synchronisation of the parallelised whole-image operations within each iteration. The Naive deconvolution, profiled in Table 6.1, shows a similar single-threaded to multithreaded overhead of approximately 2 million counts and similar counts for the single threaded configuration unregularised deconvolution as seen for the regularised deconvolutions, suggesting the Naive strategy's parallelisation of the wavelet shrinking operation is responsible for this large increase in memory accesses, rather than the calculation of the wavelet shrinking operation.

All the Naive strategy runs shown in the previous tables follow a similar pattern of $L_2$ references of over 50%, and $L_3$ references of under 50%. This overall pattern is expected to be observed if the range of memory required by a process greatly exceeds the size of the $L_1$ cache, but is more in line with the size of the $L_2$ cache. As the inner convolution operations of the Naive strategy have a minimum range throughout which they must regularly reference pixel data, this suggests that the minimum memory range is greater than twice the size of the $L_1$ cache, but less than twice the size of the $L_2$ cache, making the in-order revisits to recently used $L_2$ pixels more likely to succeed than for recently used $L_1$ pixels. For the hyperthreaded configurations, the size of the per-thread $L_1$ cache is effectively halved compared to the other runs, as it is shared between both logical threads. This would be expected to produce an increase in the number of $L_1$ cache misses, as the minimum range is comparatively doubled. This increase is observed for the $L_1$ store misses, and due to the higher number of load to store references, it results in $L_1$ store misses reaching approximately 100%.

The Topdown strategy's approach to parallelisation, profiled in Tables 6.4, 6.5, and 6.6, tightens the ranges throughout which the threads must each regularly reference pixel data. This can be seen as the number of $L_2$ references falls below half of the number of $L_1$ references for all but the hyperthreaded configurations once the image is divided amongst more than one thread. The 100% $L_1$ store-miss rate of the hyperthreaded configurations is also avoided.

**Topdown Unregularised Deconvolution**
(Millions of Line-References, Percentages of higher cache level's line-reference count)

| **8 Megapixel, 2:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 2501 | | 4221 | | 3814 | | 3221 | |
| $L_2$ Cache Load References | 1372 | (55%) | 937 | (22%) | 824 | (22%) | 978 | (30%) |
| $L_3$ Cache Load References | 465 | (34%) | 411 | (44%) | 333 | (40%) | 477 | (49%) |
| $L_1$ Cache Store References | 765 | | 1125 | | 1147 | | 1170 | |
| $L_2$ Cache Store References | 612 | (80%) | 373 | (33%) | 361 | (32%) | 446 | (38%) |
| $L_3$ Cache Store References | 389 | (64%) | 134 | (36%) | 122 | (34%) | 202 | (45%) |

| **16 Megapixel, 1:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 5282 | | 7538 | | 8446 | | 7390 | |
| $L_2$ Cache Load References | 2810 | (53%) | 2617 | (35%) | 1870 | (22%) | 2074 | (28%) |
| $L_3$ Cache Load References | 920 | (33%) | 926 | (35%) | 875 | (47%) | 1128 | (54%) |
| $L_1$ Cache Store References | 1674 | | 2246 | | 2250 | | 2292 | |
| $L_2$ Cache Store References | 1230 | (73%) | 1329 | (59%) | 746 | (33%) | 857 | (37%) |
| $L_3$ Cache Store References | 794 | (65%) | 492 | (37%) | 275 | (37%) | 400 | (47%) |

Table 6.4

**Topdown Haar Universal-Thresholding Deconvolution**
(Millions of Line-References, Percentages of higher cache level's line-reference count)

| **8 Megapixel, 2:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 2501 | | 4223 | | 3813 | | 3220 | |
| $L_2$ Cache Load References | 1376 | (55%) | 939 | (22%) | 826 | (22%) | 977 | (30%) |
| $L_3$ Cache Load References | 469 | (45%) | 412 | (44%) | 320 | (39%) | 477 | (49%) |
| $L_1$ Cache Store References | 795 | | 1125 | | 1147 | | 1170 | |
| $L_2$ Cache Store References | 616 | (77%) | 375 | (33%) | 361 | (31%) | 445 | (38%) |
| $L_3$ Cache Store References | 389 | (63%) | 134 | (36%) | 122 | (34%) | 202 | (45%) |

| **16 Megapixel, 1:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 5282 | | 7538 | | 6446 | | 7390 | |
| $L_2$ Cache Load References | 2795 | (53%) | 2619 | (35%) | 1869 | (29%) | 2074 | (28%) |
| $L_3$ Cache Load References | 910 | (33%) | 926 | (35%) | 887 | (47%) | 1129 | (54%) |
| $L_1$ Cache Store References | 1674 | | 2246 | | 2250 | | 2292 | |
| $L_2$ Cache Store References | 1243 | (74%) | 1330 | (50%) | 748 | (33%) | 857 | (37%) |
| $L_3$ Cache Store References | 794 | (64%) | 492 | (37%) | 278 | (37%) | 398 | (46%) |

Table 6.5

**Topdown Daub-8 Universal-Thresholding Deconvolution**
(Millions of Line-References, Percentages of higher cache level's line-reference count)

| **8 Megapixel, 2:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 2501 | | 4223 | | 3813 | | 3221 | |
| $L_2$ Cache Load References | 1379 | (55%) | 939 | (22%) | 826 | (21%) | 976 | (30%) |
| $L_3$ Cache Load References | 471 | (34%) | 412 | (44%) | 322 | (39%) | 482 | (49%) |
| $L_1$ Cache Store References | 795 | | 1125 | | 1147 | | 1170 | |
| $L_2$ Cache Store References | 608 | (76%) | 375 | (33%) | 362 | (32%) | 445 | (38%) |
| $L_3$ Cache Store References | 389 | (64%) | 134 | (36%) | 122 | (31%) | 201 | (45%) |

| **16 Megapixel, 1:1 Image** | Threads: 1 | | Threads: 2 | | Threads: 4 | | Threads: 8 | |
|---|---|---|---|---|---|---|---|---|
| $L_1$ Cache Load References | 5281 | | 7538 | | 8445 | | 7391 | |
| $L_2$ Cache Load References | 2796 | (53%) | 2619 | (35%) | 1869 | (22%) | 2070 | (28%) |
| $L_3$ Cache Load References | 910 | (33%) | 926 | (35%) | 884 | (47%) | 1135 | (55%) |
| $L_1$ Cache Store References | 1674 | | 2246 | | 2249 | | 2292 | |
| $L_2$ Cache Store References | 1243 | (74%) | 1330 | (59%) | 747 | (33%) | 857 | (37%) |
| $L_3$ Cache Store References | 795 | (64%) | 492 | (37%) | 278 | (37%) | 397 | (46%) |

Table 6.6

## 6.4 Conclusion

We have investigated the comparative effectiveness of the Multicore CPU and Generally Programmable GPU by implementing and profiling the behaviour of an image processing task that fits into the critical path of a wide range of image processing and computer vision operations, but has historically been prohibitive to widespread adoption due to its computational expense. We have implemented four parallelisation strategies for this algorithm: a Naive base case, which is equivalent for both platforms, and three complementary array-tiling strategies that focus on utilising the underlying available parallelism in situations of a high core-to-memory ratio (the Multicore CPU), and a low core-to-memory ratio (the GPU). Array-tiling strategies are commonly employed for image deconvolution as such approaches correspond to the necessary memory access patterns of spatially variant deconvolution. The novel contributions of this work are the Multicore CPU and GPU implementations of the wavelet-regularised deconvolution algorithm, suitable for spatially variant wavelet regularised deconvolution and for a wavelet regularised blind deconvolution algorithm, which would have been impractical in practice without the underlying Multicore and GPU parallelisation.

In this chapter we have analysed the behaviour of these parallel algorithms with respect to their static memory requirements, the utilisation of the various computational units provided by the Multicore and GPU, and their positive and negative interactions with each platform's memory hierarchies in terms of the number of full in-order passes over the data which determines the minimum possible number of cache misses per pass. We have found that the inclusive model of the Multicore memory hierarchy imposes an upper limit (which happens to cover the transition to gigapixel deconvolution) on the size of the images that can be processed, which the exclusive model of the GPU memory hierarchy is not subject to. We have found that the CPU Queuing and Streaming strategies exhibit the best and most deterministic use of the logical hyperthreading capabilities available on the Multicore CPU, followed by the Naive strategy—despite the fact that this strategy proceeds at a finer granularity with more regular synchronisation intervals than the largely unsynchronised Topdown Multicore strategy. We have found that the differences in design between the array-tiling strategies were insufficient to produce more than a 5× difference in deconvolution time. As Multicore CPU strategies have operated on double-precision pixel data, and the GPU strategies on single-precision pixel data, this 5× difference is will likely close when the GPU strategies are benchmarked on double-precision capable hardware.

The opportunities for further investigation that have been identified in this chapter are discussed in more detail in Chapter 7.

# Chapter 7

# Future Directions: Towards a general, high-throughput, Blind Deconvolution library

## 7.1 GPU Queuing Strategy

Investigate the feasibility of implementing a GPU Queuing Strategy, which has a larger memory requirement than the GPU Streaming Strategy. The GPU Queuing Strategy will be limited to PSF sizes of a certain range, due to the GPU device memory restrictions.

## 7.2 Implement a Simultaneous Forward and Inverse Wavelet Transform

The forwards and inverse wavelet transforms are mirror images of one another. The forward wavelet transform begins using the maximum number of threads, and halves its thread count with each iteration until the transform is complete. The inverse transform begins using a single thread, and doubles its thread count with each iteration until the transform is complete. While each transform leaves cores unused for the bulk of its runtime, the simultaneous performance of a forward and inverse transform (on independent input data) will consistently occupy all available cores for the duration of both transforms. This functionality is expected to be of use only to the GPU wavelet transform implementation, as a Multicore CPU will naturally schedule a simultaneous forward and inverse transform in such a fashion. As the IMPAIR wavelet shrinking

algorithms cannot make use of this overlapped behaviour, it has been left unimplemented for the time being.

## 7.3 Investigate Cache-Share penalty

Currently, IMPAIR attempts to the share all immutable memory between all threads of execution, under the assumption that since the cost of sharing an immutable cache line between physical cores is less than the cost to a physical core of a cache miss, it is best to leave as much of the cache available for unshared mutable use as possible. Whether or not this is the case has not been directly investigated. An indirect measurement can be seen by the comparative performance of a spatially variant Topdown convolution (which does not share the PSF image between cores) and a spatially invariant Topdown convolution (which does).

## 7.4 Single and Multi-dimensional DWT & WRL

Currently, the IMPAIR libraries provide operations for the deconvolution of two dimensional image data, as well as the more low level two dimensional image data operations based around the wavelet transform. Modifying the IMPAIR libraries to provide one dimensional variants of these operations is an obvious next step, allowing the Richardson Lucy algorithm to be applied to the deconvolution of timeseries data, which the algorithms implemented in IMPAIR will be able to support into the scale of billions of datapoints.

Extending IMPAIR to handle three dimensional volumetric data, and datasets of an arbitrarily higher dimensionality, will only require extensions to the wavelet shrinking operation. The largest volume processable by IMPAIR's current parallelisation strategies will be limited in 3D to 64 megavoxels, and in 8D to an $8^8$ hypercube.

## 7.5 Additional Wavelet Shrinking Algorithms

Currently the IMPAIR DWT library only provides coefficients for the *Daub-2* to *Daub-8* wavelets. Extending this to further Daubechie family wavelets only requires adding these coefficients to the library source. Furthermore, additional wavelet coefficients such as the CDF 9/7 and CDF 5/3 wavelets used by the JPEG 2000 standard [133], and recognised for their superior performance for both lossy and lossless image compression [134], can be introduced with no changes to the software logic.

The introduction of support for additional wavelet families should lead to the implementation of additional wavelet shrinking algorithms [135][136], beyond the Universal-Thresholding and K-Sigma clipping techniques currently employed by IMPAIR. Adaptive strategies, with few or no tuning parameters [82][83][84], will be considered to be the most ideal approach to regularising the parameter-less Richardson Lucy deconvolution algorithm.

## 7.6 Ports

IMPAIR's Multicore CPU implementation currently relies exclusively on the OpenMP compiler extension, developed for shared memory HPC. While this extension is supported by the major C compilers in current use (the GNU gcc compiler, the LLVM Clang, and Microsoft's MSVC), the OpenMP thread-pool framework underperforms when it is used in conjunction with threaded software that is not based on OpenMP, such as POSIX threads or Windows threads. In order for the IMPAIR libraries to be compatible with other OpenMP based libraries, it must not rely on any other threading back-end. Conversely, in order for the IMPAIR library to be compatible with POSIX threaded software, it must *not* rely on an OpenMP back-end. As most computational software is OpenMP based, the OpenMP back-end was chosen for the first implementation, but future work must include extending IMPAIR to support POSIX threads explicitly.

## 7.7 Bindings

The IMPAIR libraries will find most use as a suite of functions that can be invoked from higher level interactive languages, like Python, Octave [137], Matlab, R [138], or the more recent HPC interactive language Julia [139]. The IMPAIR API was designed from the outset to be easily exposed to this set of languages, and does not rely on any C preprocessor macros or "magic-number" constants for correct use. This allows any high level language with a minimal foreign-function call interface to avail of the IMPAIR libraries without requiring any code generation or header file parsing operations.

Providing IMPAIR bindings to a high level language will allow for interactive image processing sessions, and a GUI front end to be created for IMPAIR with minimal development effort.

## 7.8 Runtime Profiles of Deconvolution for PSF Recovery

The runtime profiles shown in Chapter 5 scale the size of the input image but keep the size of the PSF image fixed at $64 \times 64$ megapixels. Deconvolution for PSF recovery and the blind RL algorithm deconvolution require deconvolution with a PSF of equal size to the input image, which will have consequences for the effectiveness of the parallelisation strategies, and may find benefit from the blind deconvolution being performed assuming a spatially-variant PSF, even in cases where the PSF is known to be spatially invariant.

## 7.9 Blind Deconvolution on the GPU and CPU

The blind Richardson Lucy deconvolution algorithm of Fish et al [98] describes a simple approach to blind deconvolution based on a pair of interacting Richardson Lucy deconvolution steps, with one step incrementally improving the PSF estimate, and the other incrementally improving the output image estimate. This approach provides a blind deconvolution that can be constructed entirely out of the components provided by the IMPAIR library, and used by other software as easily as IMPAIR's non-blind deconvolution operations, due to it being a similarly tuning-parameter-free deconvolution algorithm.

While the unregularised Richardson Lucy algorithm achieves acceptable restoration after approximately 10 iterations, the Fish algorithm requires an orders of magnitude more, due to the nesting of one RL process within another. Over one hundred iterations of the PSF estimation process are recommended per iteration of the image estimation process, requiring over a thousand Richardson Lucy operations to achieve blind image restoration.

With IMPAIR's current performance, the unregularised blind deconvolution of a $2048 \times 2048$ pixel image should be achieved in roughly 500 seconds, which allows for 10 iterations of the outer, image-estimating, deconvolution, each containing 100 iterations of the inner PSF-estimating, deconvolution, with regularised blind deconvolution of a $2048 \times 2048$ pixel image likely to take less than 600 seconds.

The inner loops of the blind deconvolution algorithm will benefit from a well defined stopping criterion, like that provided by [88], in order to maintain the total number of iterations as the only tuning parameter. An investigation as to whether it is practical to allow the blind deconvolution algorithm to converge given an arbitrary fixed number of inner iterations will be carried out [70][105].

# Appendix A

# IMPAIR Deconvolution Library API

The design of the IMPAIR WRL library covers two loosely coupled domains: the underlying deconvolution engine, and the user-facing library API. These two concerns are of roughly equal importance, and any defects in one or other of the domains will have negative effects on the utility of the library as a whole.

The primary focus of the design of the library API is to provide as clean and straight forward a user interface as possible to the underlying deconvolution engine. This means that the primary use-cases of the library must be easily performed, with as close to a single API function call as is sensible, and with few, if any, configuration parameters required in order to get the best results from the underlying engine. Less and less common use-cases should require proportionally more complex API function calls, either in terms of the number of API function calls required, or the number of configuration parameters that must be supplied.

There are three reasons for pushing for this behaviour:

1. A library API is a user interface in a very real sense to the programmers who will use the library in their software, and whether or not they decide to adopt the use of the library will depend as much on how complicated it is for them to use it as on how well it does it's job.

2. The clearer and simpler the API is, the easier it will be to be port the library to other operating systems and hardware, and the easier it will be to incorporate the library into software written in other programming languages. As with the first reason, this will increase the overall utility of the library to any prospective programmers.

3. The interface to the library will influence the interface to any software that is built on it, and—particularly for user-facing (as opposed to programmer-facing) application programs—a more complicated library API will increase the likelihood of a more complicated GUI or command line interface in the calling application.

IMPAIR takes two approaches to this, implementating a high-level object oriented style API, where objects are used to hide the allocation of working buffers and algorithm configuration. This follows the design precident set by the FFTW library, and similar software like the CUDA CuFFT API. IMPAIR also provides a low-level API, where working buffers of the appropriate size must be provided by the calling function, this is in keeping with the more low level BLAS and GNU Scientific Library design principles.

The first approach results in more straight forward code, and a more user friendly/easier to understand API, that doesn't ask the user to have an understanding of the internal workings. This is particularly advantageous for parallel programming, as it prevents complicated resource-sharing issues coming into play.

The second allows for the user take advantage of their understanding of the inner workings in a limited way, by re-using the same buffer for multiple unrelated transforms. This can be detrimental in a parallel system, since the user must manually manage resource-sharing—but advantageous in situations where memory is in short supply, and buffer re-use will provide a performance gain by avoiding paging induced latency.

IMPAIR provides both of these interfaces, the default interface is the FFTW style resource hiding interface, but a more advanced `_using_buffer()` allows the residual buffer to be externally allocated by the user, so it can be re-used between IMPAIR calls. This buffer must be allocated on a per-thread basis, and must be large enough to hold the input image batch.

IMPAIR WRL uses this feature of the IMPAIR DWT library, and exposes it back through it's own API. IMPAIR WRL only allows *one* buffer to be user allocated (the DWT/residual buffer), since the implementation of this is just to pass the call through to the DWT library. The other IMPAIR WRL buffers do not allow externally allocation, for the sole reason of keeping the number API functions to a reasonable limit. This DWT/residual buffer is not used for plain RL deconvolution, so using this advanced constructor in that case will not result in a reduced memory footprint.

## A.1 Use Cases

The basic use-cases of the library are organised along three axes, with a fourth axis (Floating Point Precision) that is specified at compile time rather then run time. The three run time axes are: the deconvolution algorithm axis, which contains the various deconvolution algorithms that the library provides; the image processing axis, which contains the various tasks the library is required to perform; the data management axis, which contains the basic operations on image datasets that the library is required to perform efficiently. The design of the IMPAIR library is such that each of these axes may be configured independently, while optimal default configurations are exposed in a straightforward manner.

### Floating Point Precision Axis

- Single Precision (32bit IEEE Floating Point)
- Single Precision (32bit Legacy CUDA Floating Point)
- Double Precision (64bit IEEE Floating Point)

### Algorithm Axis

- "Plain" Richardson-Lucy Deconvolution
- Universal Thresholding Regularised Richardson-Lucy Deconvolution
- K-Sigma Clipping Regularised Richardson-Lucy Deconvolution

### Image Processing Axis

- Spatially Invariant Deconvolution
- Spatially Variant Deconvolution

  Course Grained SPV (high throughput, low/reduced quality image)

  Fine Grained SPV (reduced throughput, higher quality image)

### Data Management Axis

The data management options have been chosen to allow IMPAIR to be used efficiently and effectively as part of a larger, more complex image processing operation, with the

predominant usage pattern of single human user processing individual images treated as being a special case of the above. Less common human centric usage patterns (such as deconvolving small sets of images) are not treated with as high a priority. This trade off can be mitigated by working with the IMPAIR library via a language that natively supports more expressive collections of arbitrary data structures then C tolerates.

No interface is provided for dealing with alternative x, y, or z strides, or for in-place manipulation of a region-of-interest in a larger image.

IMPAIR data management use-cases:

- Images that share the same PSF or PSF array

  Deconvolve a single small image

  Deconvolve a set of small images

  Deconvolve a single large image

  Deconvolve a set of large images

  Deconvolve a set of arbitrary images

- Images that do not share the same PSF or PSF array

  Deconvolve a set of small images

  Deconvolve a set of large images

  Deconvolve a set of arbitrary images

In this case large means greater than four times the size of the PSF image or PSF element in a PSF array, and small means less than or equal to four times the size of the PSF image or PSF element in a PSF array. This is used as an estimate as to whether or not the deconvolution should be processed using an out-of-core algorithm.

Of course, for extremely large or extremely small PSF images or PSF elements this estimate will backfire. For the former case the outcome will still result in a sensible decision, since IMPAIR does not provide a specialised deconvolution function for large PSF images or PSF elements, and relies on the fine grained parallelism optimisations of the in-core algorithm to achieve optimal performance. The latter case can be overcome by specifying an arbitrary minimum value for a PSF's width and height, or simply ignored and allowed to invoke the out-of-core algorithm. Comparisons between these two behaviours are detailed in chapter 6.

IMPAIR provides specialised implementations of the following data management related operations:

- Deconvolve a single small image
- Deconvolve a set of small images that share the same PSF
- Deconvolve a set of small images the do not share the same PSF
- Mapping a large image into a set of small images

These operations are used to implement the six of the above 8 data management use-cases in an optimised manner. Deconvolving sets of arbitrary images is not supported directly by the IMPAIR library, but will be made available through other languages as described in chapter 7.

The next two sections describe the how the first two data management use cases are handled directly by the in terms of the specialised data management operations.

#### Single Small Image

From the data management perspective, the deconvolution of a single small image is no-op. The responsibility for optimisation is delegated to underlying deconvolution function and it's components. The underlying function is instructed to process the data at the maximum speed, regardless of the resource overhead. This is the simple base-case, and amounts to configuring the DFT and DWT libraries to avail of the largest number of cores that will not create contentions in cache access. For example: exceptionally small images will be processed by a single core, while images on the order of a megabyte in size will be distributed across all cores. For machines with a high number of cores (ie. more then eight) the gains of doing this will not be much greater then distributing the computation across half of these cores. IMPAIR's treatment of sets of small images is more sensible with regards to this kind of "over-optimisation".

#### Sets of Small Images

IMPAIR's treatment of sets of small images is to first divide the set up into batches of images which will be processed simultaneously, and then to allocate all available cores to processing each batch in turn. The number of images per batch is based on the size of an image from the set, the number of logical cores that are available, and a guess of number of physical cores that back them. Currently this is that on machines with more then 8 logical cores each physical core backs two logical cores, and on machines with less then 8 logical cores each logical core is backed by one physical core. In situations where there are more images in the set then logical cores available, the batch size is chosen so that

there is roughly one image per physical core, which amounts to two or more logical cores on a hyper-threaded machine. In situations where the number of images is less then the number of logical cores available, the computation will be distributed over the largest number of cores that will not create cache contentions.

This treatment is identical for sets of images that share the same PSF, and for sets of images that have a unique PSF image per input image.

#### Single Large Image

IMPAIR's treatment of single large images is to create a set of small images that share the same PSF, and process this set according that strategy. This mapping operation is similar to the mapping operation performed by the Spatially Variant Deconvolution function, but uses a different set of criteria to determine the width and height of each tile in the image. The choice of tile width and height in this case is determined by the number of available logical cores, and the width and height of the PSF image. Depending on the number of logical cores available, and the ratio of the PSF image size to the larege image size, either the Naive or Topdown strategy is chosen at runtime to perform the deconvolution.

#### Sets of Large Images

IMPAIR's treatment of sets of large images is to process each large image in sequence, one at a time. This approach ensures the memory footprint of the operation does not rise further than—and thus begin to stymie—the throughput gains brought on by the above optimisations. This policy of lazy evaluation allows IMPAIR's performance to scale gracefully and predictably to large datasets. One concern regarding this approach is that by providing an interface for deconvolving batches of large images suggests that IMPAIR will performing a more sensitive or intelligent operation then it is. There is a risk that this may tempt or encourage the user to avail of this interface instead of developing their own streaming or pipeline model for processing the large dataset, which is far more efficient in terms of the memory overhead incurred by using this batch interface, and the high latency incurred due to large image disk/network IO.

## A.2 Implementations

There are two base implementations of the IMPAIR deconvolution algorithms: the CPU implementation written in ISO C90, and the GPU implementation written in a combination of ISO C90 and NVIDIA CUDA.

The CPU engine provides course grained multicore SIMD parallelism (via the OpenMP runtime) and a single core implementation that involves no thread calls whatsoever, which is suitable for both serial execution and simultaneous execution from multiple OS threads in an MIMD or MISD fashion. Both these variants exploit non-thread-based parallelism via per core SIMD CPU registers if available.

The GPU implementation exploits both course grained and fine grained SIMD parallel operations on a much larger scale, and exposes both a CPU oriented API that transparently manages CPU–GPU communications, and a GPU oriented API that allows the caller code to manage the CPU–GPU communications manually.

## A.3 Library API

### A.3.1 API Design

The IMPAIR library API provides an interface to two underlying deconvolution engines: the CPU engine and the the GPU engine. The API is structured in such a fashion that only a handful of differences exist between API for each underlying engine, which are kept clustered around the setup code. The function calls that actually perform the various deconvolution operations that IMPAIR supports do not vary between the CPU and GPU engines.

### A.3.2 The IMPAIR Library Helper Functions and Datatypes

These datatypes and functions are used to specify details of how a discrete wavelet transform is to be performed. They are compatible with both the CPU and GPU libraries.

`impair_real`

and

`impair_real*`

All signal and filter data in the IMPAIR library perform operations on arrays of impair_real data. The impair_real datatype is a typedef wrapper of either the float, double, or long double datatypes. An instance of the IMPAIR DWT Library can be compiled to support operations on one and only one of these floating-point datatypes.

By default, impair_real wraps the double ISO90 C dataype. However, IMPAIR can be compiled so that `impair_real` wraps either single or quad precision floating point values if this is required. Certain CUDA enabled GPU devices have limited to no support for double-precision floating point operations. In these cases, IMPAIR must be compiled to use single-precision floating point operations.

`impair_wrl_plan*`

This datatype represents a specific deconvolution operation to be executed, configured for a signal of specific dimensions and size, with a specific wavelet basis. The datatype is returned by calling any of the IMPAIR WRL plan creation functions. The plan contains the pre-allocated temporary storage necessary to perform the deconvolution. Consequently, plans must be created on a per-thread basis, and not shared between two or more threads in simultaneous execution. A plan can be used any number of times before it is destroyed. The calling function is responsible for destroying this datatype with the `impair_wrl_destroy()` function.

This plan is compatible with both plain Richardson-Lucy and wavelet-regularised Richardson-Lucy deconvolution.

This plan is compatible with both CPU and GPU IMPAIR WRL plan constructors.

### A.3.3 C Multicore API

**Functions —Constructors**

The following functions create plans for executing the deconvolution of images and image batches of fixed dimensions on the CPU. Once these plans have been created they can be re-used for any images or image batches with the same dimensions. A WRL plan will allocate and hold all CPU resources necessary to perform the WRL at this point, so that no memory management is performed during the actual WRL operation. The constructor functions must only be called from the main thread, due to the non-thread-safe nature of the constructors of some of IMPAIR's dependencies.

WRL plans should not be executed from multiple threads simultaneously, as the pre-allocated resources will force the execution of the multiple threads to be serialised. If simultaneous WRL transforms are required to be performed from different calling threads, the WRL plans must be constructed as single core operations, and on a per-thread basis.

```
impair_plan*
impair_rl_create(int width, int height,
                 int psf_width, int psf_height, impair_real*psf_image)

impair_plan*
impair_wrl_create(int width, int height,
                  int psf_width, int psf_height, impair_real*psf_image,
                  impair_dwt_wavelet* wavelet)
```

`impair_rl_create2d` is a quick interface to unregularised deconvolution. To plan unregularised deconvolution using the more advanced functions, pass a NULL in as the impair_dwt_wavelet.

```
impair_plan*
impair_wrl_create2d_batch(int batch, int width, int height
                          int psf_width, int psf_height,
                          impair_real*psf_image,
                          impair_dwt_wavelet* wavelet)

impair_plan*
impair_rl_create2d_variant(int batch, int width, int height,
                           impair_real*psf_array_image,
                           int psf_count_across, int psf_count_down)

impair_plan*
impair_wrl_create2d_variant(int batch, int width, int height,
                            impair_real*psf_array_image,
                            int psf_count_across, int psf_count_down,
                            impair_dwt_wavelet* wavelet)
```

(an image of all the PSFs, centred in their tiles, and the number of PSFs across and down the array)

there is no plan for batches of deconvolutions using separate PSFs for each image. To deconvolve a batch of separate PSFs-image pairs, create a plan for each pair, and run them sequentially or using your own parallelism:

```
impair_wrl_plan plan[BATCH];

/* plans are all created by a single thread */
for(i = 0; i < BATCH; ++i)
{
  plan[i] = impair_wrl_create(input_hor[i],input_ver[i],
                              psf_image[i],psf_hor[i],psf_ver[i]);
}

/* plans are all executed on separate threads */
#pragma omp parallel for
for(i = 0; i < BATCH; ++i)
{
    impair_wrl_exec(plan[i], input_image[i], output_image[i]);
}
```

**Functions —Evaluators**

```
void impair_wrl_exec(impair_plan* plan,
                     impair_real* input,
                     impair_real* output,
                     int iterations)

void impair_wrl_exec_clip(impair_plan* plan,
                          impair_real* input,
                          impair_real* output,
                          int iterations,
                          impair_real threshold)
```

Performs a planned deconvolution operation on the data pointed to by input, storing the results in the buffer pointed to by output. This evaluator is compatible with planned deconvolutions of any batch size.

The input and output buffers must be of the size specified when the plan was created, and may overlap. However, if either buffer has been previously supplied as the scratch buffer for the transform, the transform will not be performed.

`impair_rl_exec` deconvolves the input image via the unregularised Richardson-Lucy algorithm.

`impair_wrl_exec` deconvolves the input image via the wavelet-regularised Richardson-Lucy algorithm, using the universal thresholding technique.

`impair_wrl_exec_clip` deconvolves the input image via the wavelet-regularised Richardson-Lucy algorithm, using the k-sigma clipping technique. The `impair_real` argument `threshold` is used to control

**Functions —Destructors**
There is a single destructor function for all WRL plans, created by any of the constructor functions.

```
void impair_destroy(impair_plan* plan)
```

Frees all memory associated with a WRL plan. Compatible with planned deconvolutions of any batch size. A plan cannot be passed to an evaluator or destructor function after it has been previously destroyed.

### A.3.4 C GPU API

### A.3.5 C CUDA API

The C CUDA IMPAIR API is a special interface to the GPU engine that allows the library to be used in conjunction with other CUDA libraries, such as NVIDIA's Fourier transform library CUFFT, or NVIDIA's BLAS implementation CUBLAS, without hiding the CPU–GPU communications.

All functions in the the C CUDA API work with image buffers allocated in GPU global memory via the `cudaMalloc()` or similar function, and cannot take CPU RAM memory buffers as arguments.

The C GPU and C CUDA APIs can be used interchangeably, so plans created with the C GPU library can be executed with the C CUDA API and vice verse.

**Functions —Constructors**

```
impair_plan*
impair_wrl_create_cuda(int batch, int width, int height
                       int psf_width, int psf_height,
                       impair_real*psf_image,
                       int wavelet_id);

impair_plan*
impair_wrl_create_cuda_using_buffer(int batch, int width, int height
                                    int psf_width, int psf_height,
                                    impair_real*psf_image,
                                    int wavelet_id, impair_real*buffer);
```

The `impair_wrl_create_cuda()` constructor creates an `impair_plan` for deconvolution using a `psf_image` buffer in GPU Global Memory. Otherwise, it performs exactly the same as the counterpart GPU constructor function.

The `impair_wrl_create_cuda_using_buffer()` constructor creates an `impair_plan` using an caller allocated GPU Global Memory buffer. This buffer must be large enough to store the input image batch, and must not be freed until after the `impair_plan` has been destroyed.

**Functions—Evaluators**

```
void impair_wrl_exec_cuda(impair_plan* plan,
                          impair_real* input,
                          impair_real* output,
                          int iterations);

void impair_wrl_exec_cuda_clip(impair_plan* plan,
                               impair_real* input,
                               impair_real* output,
                               int iterations,
                               impair_real threshold);
```

The functions perform exactly as their counterpart GPU and CPU functions, except the `input` and `output` buffers are located in GPU Global Memory, not CPU Main Memory.

**Functions —Destructors**

The GPU destructor `impair_wrl_destroy()` should be used for destroying an `impair_plan` created with the C CUDA API constructors.

# Appendix B

# IMPAIR Discrete Wavelet Transform Library API

## B.1 Use Cases

As with the IMPAIR WRL library, the DWT library use cases are organised along several axes. The precision axis, which deals with DWT the floating point precision of the transform; The wavelet basis axis, which deals with each of the various wavelet bases that are available; the discrete transform axes, with a category for the forward discrete wavelet transform and the inverse discrete wavelet transform; the decimating or recursive transform axis, which deals with the parameters specific to the decimating or recursive wavelet transform;

- Floating Point Precision Axis

  Single Precision (32bit IEEE Floating Point)

  Single Precision (32bit CUDA Compute XX.YY or earlier Floating Point)

  Double Precision (64bit IEEE Floating Point)

- Wavelet Basis Axis

  Haar

  Daub4 - Daub20

- Discrete Transform Axis

  Forward DWT

  Inverse DWT

- Decimating or Recursive Transform Axis

  Single Level DWT

  N Level DWT

  Full DWT

- Signal Data Axis

  1 Dimensional Transforms

  2 Dimensional Transforms

  2 Dimensional so called "non-standard" or Mallet Transforms

- Data Management Axis

  Transform a single signal or image

  Transform a set of signals or images

## B.2 API

### B.2.1 The DWT Library Helper Functions and Datatypes

These datatypes and functions are used to specify details of how a discrete wavelet transform is to be performed. They are compatible with both the CPU and GPU libraries.

`impair_real`

and

`impair_real*`

All signal and filter data in the IMPAIR DWT Library perform operations on arrays of impair_real data. The impair_real datatype is a typedef wrapper of either the float, double, or long double datatypes. An instance of the IMPAIR DWT Library can be compiled to support operations on one and only one of these floating-point datatypes.

By default, impair_real wraps the double ISO90 C dataype. However, certain CUDA enabled GPU devices have limited to no support for double-precision floating point operations. In these cases, it is advisable to compile the IMPAIR DWT library to use single-precision floating point operations.

`impair_wavelet*`

This datatype represents a wavelet basis to be used during a discrete wavelet transform. The datatype is returned by calling either:

impair_dwt_wavelet_haar()
impair_dwt_wavelet_daub()
impair_dwt_wavelet_custom()

Once this datatype has been passed to the appropriate create function, it will not be needed again. The calling function is not required to destroy or otherwise de-allocate this datatype.

`impair_wavelet* impair_wavelet_haar(void)`

Creates a wavelet object to be passed to the create functions to specify a haar wavelet. The haar wavelet is equivalent to the daub2 wavelet, created by impair_dwt_wavelet_daub(2).

`impair_wavelet* impair_wavelet_daub(moments)`

Creates a wavelet object to be passed to the _create functions to specify a Daubechie wavelet with N vanishing moments. Currently only 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 vanishing moments are supported. If an unsupported number of vanishing moments is requested, impair_dwt_wavelet_daub() will return NULL.

### B.2.2 The CPU Library

#### Datatypes

`impair_dwt_plan_cpu*`

This datatype represents a specific wavelet transform to be executed on the CPU, configured for a signal of specific dimensions and size, with a specific wavelet basis. The datatype is returned by calling any of the IMPAIR DWT CPU plan creation functions. The plan contains the pre-allocated temporary storage nessesary to perform the discrete wavelet transform. Consequently, plans must be created on a per-thread basis, and not shared between two or more threads in simultaneous execution. A plan can be used any

number of times before it is destroyed. The calling function is responsible for destroying this datatype with the `impair_dwt_cpu_destroy()` function.

This function is compatible with both one and two dimensional, and both forwards and inverse discrete wavelet transforms.

This function is only compatible with the IMPAIR DWT CPU functions of the IMPAIR DWT Library.

**Functions —Constructors**

The following functions create plans for executing discrete wavelet transforms of images and image batches of fixed dimensions. Once these plans have been created they can be re-used for any images or image batches with the same dimensions. A DWT plan will allocate and hold all resources nessesary to perform the DWT at this point, so that no memory management is performed during the actual calculation of the DWT. DWT plans should not be called from multiple calling threads simultaneously, as the pre-allocated resources will force the execution of the multiple threads to be serialised. If simultaneous DWT transforms are required to be performed from different calling threads, the DWT plans must be allocated on a per-thread basis.

```
impair_dwt_plan_cpu*
impair_dwt_create_cpu(int length, impair_dwt_wavelet* wavelet)
```

Create a plan for transforming a 1 dimensional signal on the CPU. The calculation of the transform will be automatically distributed across all available threads. This function allocates the nessesary resources for performing the 1D DWT, and collects them in the `impair_dwt_cpu_plan` structure. A 1D plan can only be used on input signals with the exact length that was specified during the plan's creation.

```
impair_dwt_plan_cpu*
impair_dwt_create_batch_cpu(int batch, int length,
                            impair_dwt_wavelet* wavelet,
                            int threads)
```

Create a plan for transforming a batch of 1 dimensional signals on the CPU, requesting a that the calculation be distributed over the specified number of threads. The threads parameter is a suggestion, and may be thresholded if it exceeds the maximum number of threads OpenMP has been configured to create. The value 0 creates the maximum number of threads. Use `impair_dwt_cpu_threads_max()` to examine this value at runtime.

When executing the transform, the signal data must be stored contiguously in the input buffer. `signal[i + j*length]` should return the $i^{th}$ element of the $j^{th}$ signal in the batch.

```
impair_dwt_plan_cpu*
impair_dwt_create_advanced_cpu(int batch, int length,
                               impair_dwt_wavelet* wavelet,
                               int threads,
                               impair_real* scratch)
```

This is an extension of the above function `impair_dwt_cpu_create_batch()` which allows an externally allocated buffer to be used as temporary storage when performing the wavelet transform. This buffer must be an array of at least batch*length texttttimpair_real elements. The buffer will not be freed when `impair_dwt_cpu_destory()` is called, and so must be managed by the calling code. The buffer must not be freed until after all wavelet transforms using this plan have been performed. The shared use of an external buffer between multiple plans breaks the thread safety of the `impair_dwt_cpu_plan`. If this is done, all plans that share the buffer can only be safely executed from the same thread.

```
impair_dwt_plan_cpu*
impair_dwt_create2d_cpu(int width, int height,
                        impair_dwt_wavelet* wavelet)
```

Create a plan for transforming a two dimensional signal on the CPU. The calculation of the transform will be automatically distributed across all available threads. This function allocates the nessesary resources for performing the 2D DWT, and collects them in the `impair_dwt_cpu_plan` structure. A 2D plan can only be used on input signals with the exact width and height that were specified during the plan's creation.

```
impair_dwt_plan_cpu*
impair_dwt_create2d_batch_cpu(int batch,
                              int width, int height,
                              impair_dwt_wavelet* wavelet,
                              int threads)
```

Create a plan for transforming a batch of two dimensional signals on the CPU, requesting a that the calculation be distributed over a number of threads. The threads parameter

is a suggestion, and may be thresholded if it exceeds the maximum number of threads OpenMP has been configured to create. The value 0 creates the maximum number of threads. Use `impair_dwt_cpu_threads_max()` to examine this value at runtime.

The signals must be stored contiguously in memory. `signal[i + j*width + k*width*height]` should return the $(i^{th}, j^{th})$ element of the $k^{th}$ signal in the batch.

```
impair_dwt_plan_cpu*
impair_dwt_create2d_advanced_cpu(int batch,
                                 int width, int height,
                                 impair_dwt_wavelet* wavelet,
                                 int threads,
                                 impair_real* scratch)
```

This is an extension of the above function `impair_dwt_cpu_create2d_batch()` which allows an externally allocated buffer to be used as temporary storage when performing the wavelet transform. This buffer must be an array of at least batch*width*height `impair_real` elements. The buffer will not be freed when `impair_dwt_cpu_destory()` is called, and so must be managed by the calling code. The buffer must not be freed until after all wavelet transforms using this plan have been performed. The shared use of an external buffer between multiple plans breaks the thread safety of the impair_dwt_cpu_plan. If this is done, the use of the plan must be restricted to a single CPU thread.

**Functions —Evaluators**

The below evaluator functions allow a plan to be used any number of times to perform both forwards and inverse transforms.

```
void impair_dwt_forward_exec_cpu(impair_dwt_plan_cpu* plan,
                                 impair_real* input,
                                 impair_real* output)
```

Performs a planned forward wavelet transform on the data pointed to by input, storing the results in the buffer pointed to by output. This evaluator is compatible with both the one and two dimensional transforms of any batch size.

The input and output buffers must be of the size specified when the plan was created, and may overlap. However, if either buffer has been previously supplied as the scratch buffer for the transform, the transform will not be performed.

```
void impair_dwt_inverse_exec_cpu(impair_dwt_plan_cpu* plan,
                                 impair_real* input,
                                 impair_real* output)
```

Performs a planned inverse wavelet transform on the data pointed to by input, storing the results in the buffer pointed to by output. This evaluator is compatible with both one and two dimensional transforms of any batch size.

The input and output buffers must be of the size specified when the plan was created, and may overlap. However, if either buffer has been previously supplied as the scratch buffer for the transform, the transform will not be performed.

**Functions —Destructors**

There is a single destructor function for all DWT CPU plans, created by any of the constructor functions.

```
void impair_dwt_destroy_cpu(impair_dwt_plan_cpu* plan)
```

Frees all memory associated with a CPU plan. Compatible with both one and two dimensional transforms of any batch size. A plan cannot be passed to an evaluator or destructor function after it has been previously destroyed.

### B.2.3 The GPU Library

**Overview** The GPU library is contains both a highlevel GPU API, and a lower level CUDA API. The GPU API follows the same structure as the CPU API, and is designed to be used by CPU based software. The CUDA API provides extra functions that CUDA based software will require in order to use the impair dwt library effectivly.

**GPU Datatypes**

```
impair_dwt_plan_gpu*
```

This datatype represents a specific wavelet transform to be executed on the GPU, configured for a signal of specific dimensions and size, with a specific wavelet basis. The datatype is returned by calling any of the IMPAIR DWT GPU plan creation functions. The plan contains the pre-allocated temporary storage nessesary to perform the discrete wavelet transform. Consequently, plans must be created on a per-thread basis, and not shared between two or more threads in simultaneous execution. A plan can be used any

number of times before it is destroyed. The calling function is responsible for destroying this datatype with the impair_dwt_destroy_gpu() function.

This datatype is compatible with both one and two dimensional, and both forwards and inverse discrete wavelet transforms.

This datatype is compatible with the GPU functions and the CUDA fucntions of the IMPAIR DWT library.

**GPU Functions —Constructors**

```
impair_dwt_plan_gpu*
impair_dwt_gpu_create(int length, impair_dwt_wavelet* wavelet)
```

Performs exactly as impair_dwt_create_cpu().

This datatype is compatible with the GPU functions and the CUDA fucntions of the IMPAIR DWT library.

```
impair_dwt_plan_gpu*
impair_dwt_create_batch_gpu(int batch, int length,
                            impair_dwt_wavelet* wavelet)
```

Performs exactly as impair_dwt_create_batch_cpu(), with the exception that this function does not accept a threads parameter, as the GPU execution is automatically parallelised based on the number of input data elements.

This datatype is compatible with the GPU functions and the CUDA fucntions of the IMPAIR DWT library.

```
impair_dwt_plan_gpu*
impair_dwt_create2d_gpu(int width, int height,
                        impair_dwt_wavelet* wavelet)
```

Performs exactly as impair_dwt_create2d_cpu().

```
impair_dwt_plan_gpu*
impair_dwt_create2d_batch_gpu(int batch,
                              int width, int height,
                              impair_dwt_wavelet* wavelet)
```

Performs exactly as impair_dwt_create2d_batch_cpu(), with the exception that this function does not accept a threads parameter, as the GPU execution is automatically parallelised based on the number of input data elements.

This datatype is compatible with the GPU functions and the CUDA fucntions of the IMPAIR DWT library.

**CPU Functions —Evaluators**

```
void impair_dwt_forward_exec_gpu(impair_dwt_plan_gpu* plan,
                                 impair_real* input,
                                 impair_real* output)
```

Performs exactly as impair_dwt_forward_exec_cpu(),

```
void impair_dwt_inverse_exec_gpu(impair_dwt_plan_gpu* plan,
  impair_real* input,
  impair_real* output)
```

Performs exactly as impair_dwt_inverse_exec_cpu(),

**GPU Functions —Destructors**

```
void impair_dwt_destroy_gpu(impair_dwt_plan_gpu* plan)
```

Performs exactly as impair_dwt_destroy_cpu().

**CUDA Functions —Constructors**

```
impair_dwt_plan_gpu*
impair_dwt_create_advanced_cuda(int batch, int length,
                                pair_dwt_wavelet* wavelet,
                                impair_real* scratch)
```

The scratch buffer must be a CUDA allocated buffer on the GPU device—as created with cudaMalloc() or similar function. It must not be a CPU buffer: doing so will result in corrupted output data of any transforms using this plan. The scratch buffer must be large enough to hold the input image or image batch that will be transformed.

This function does not accept a threads parameter, as the GPU execution is automatically parallelised based on the number of input data elements.

Otherwise, performs exactly as impair_dwt_create_advanced_cpu().

```
impair_dwt_plan_gpu*
impair_dwt_create2d_advanced_cuda(int batch,
                                  int width, int height,
                                  impair_dwt_wavelet* wavelet,
                                  impair_real* scratch)
```

The scratch buffer must be a CUDA allocated buffer on the GPU device—as created with cudaMalloc() or similar function. It must not be a CPU buffer: doing so will result in corrupted output data of any transforms using this plan. The scratch buffer must be large enough to hold the input image or image batch that will be transformed.

This function does not accept a threads parameter, as the GPU execution is automatically parallelised based on the number of input data elements.

Otherwise, performs exactly as impair_dwt_create2d_advanced_cpu(),

```
void impair_dwt_forward_exec_cuda(impair_dwt_plan_gpu* plan,
                                  impair_real* input, impair_real* output)
```

The input and output buffers must be a CUDA allocated buffers on the GPU device–as created with cudaMalloc() or similar function. They must not be CPU buffers: doing so will result in corrupted output data of any transforms using this plan.

Performs exactly as impair_dwt_forward_exec_cpu().

```
void impair_dwt_inverse_exec_cuda(impair_dwt_plan_cpu* plan,
                                  impair_real* input, impair_real* output)
```

The input and output buffers must be a CUDA allocated buffers on the GPU device—as created with cudaMalloc() or similar function. They must not be CPU buffers: doing so will result in corrupted output data of any transforms using this plan.

Performs exactly as impair_dwt_inverse_exec_cpu().

# Appendix C

# Appendix C

Figure C.1 (overleaf) shows the runtimes of a previous IMPAIR verion's N-tiles-to-M-cores Tiling strategy, for the Multicore CPU.

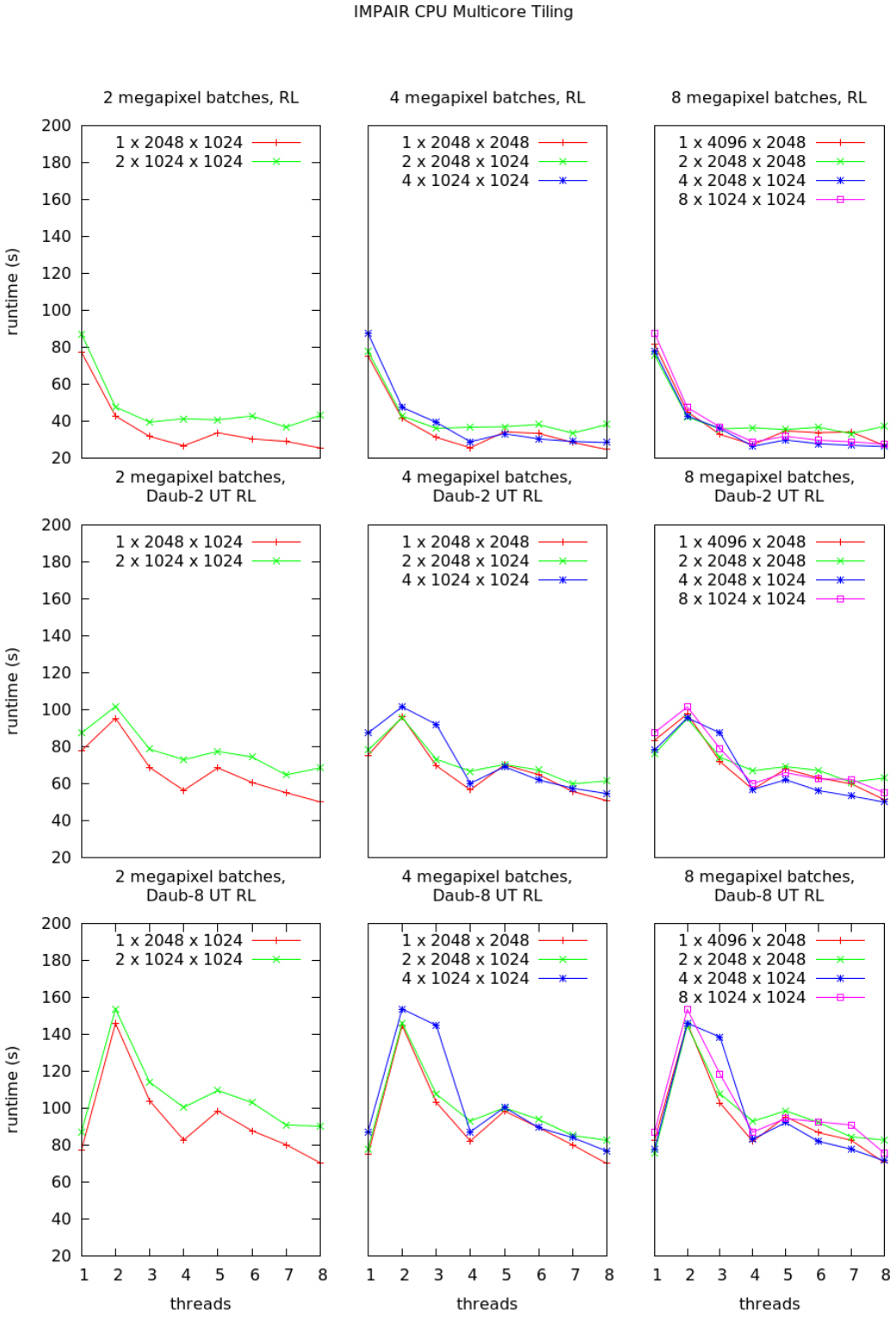

IMPAIR CPU Multicore Tiling
2 megapixel batches, RL
4 megapixel batches, RL
8 megapixel batches, RL
1 x 2048 x 1024
2 x 1024 x 1024
1 x 2048 x 2048
2 x 2048 x 1024
4 x 1024 x 1024
1 x 4096 x 2048
2 x 2048 x 2048
4 x 2048 x 1024
8 x 1024 x 1024
2 megapixel batches, Daub-2 UT RL
4 megapixel batches, Daub-2 UT RL
8 megapixel batches, Daub-2 UT RL
2 megapixel batches, Daub-8 UT RL
4 megapixel batches, Daub-8 UT RL
8 megapixel batches, Daub-8 UT RL
runtime (s)
threads

FIGURE C.1

# Bibliography


[1] E Wes Bethel and Mark Howison. Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning. *The International Journal of High Performance Computing Applications*, 26(4):399–412, 2012.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[3] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13:22–30, 2011.

[4] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[5] NVIDIA Corp. *CUBLAS Library*, 2012.

[6] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1918–1927. IEEE Computer Society, 2013. ISBN 978-0-7695-4979-8.

[7] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[8] C. V. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Comput. Surv.*, 9 (1):61–102, 1977.

[9] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The solomon computer. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, AFIPS '62 (Fall), pages 97–107, New York, NY, USA, 1962. ACM.

[10] S. F. Reddaway. Dap—a distributed array processor. *SIGARCH Comput. Archit. News*, 2(4):61–65, 1973.

[11] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, 1978.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[14] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, New York, NY, USA, 2009.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

[16] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[17] Mike Johnson. *Superscalar microprocessor design*. Prentice Hall, 1991.

[18] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: Past, present and future. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 425–432, 1998. ISBN 0-89791-998-X.

[19] Charles J. Purcell. The control data star-100: Performance measurements. In *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, AFIPS '74, pages 385–387, New York, NY, USA, 1974. ACM.

[20] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

[21] Robert Bernecky. The role of apl and j in high-performance computation. *SIGAPL APL Quote Quad*, 24(1):17–32.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[23] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, 1974.

[24] M. L. Mumford, D. K. Andes, and L. L. Kern. The mod 2 neurocomputer system design. *IEEE Transactions on Neural Networks*, 3(3):423–433, 1992.

[25] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[26] M. Greenberg and V. Woods. The flagship parallel reduction machine. In *IEEE Colloquium on VLSI and Architectures for Symbolic Processing*, 1989.

[27] A. S. Partridge and A. H. Dekker. Speculative parallelism in a distributed graph reduction machine. In *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, volume 2, pages 771–779 vol.2, 1989.

[28] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354, 1997.

[29] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5): 2–11, 1996.

[30] Sterling T. et al. Becker, D. Beowulf: A parallel workstation for scientific computation. pages 11–14, 1995.

[31] Thomas Ludescher, Thomas Feilhauer, and Peter Brezany. Cloud-based code execution framework for scientific problem solving environments. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1), 2013.

[32] B.R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspective.

[33] R. S. Nikhil. Can dataflow subsume von neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, 1989.

[34] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS*, pages 177–186, 2005.

[35] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[36] Stuart Oberman, Greg Favor, and Fred Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[37] Sanjeev Jahagirdar Tanveer Khondker Robert Milstrey Sanjib Sarkar Scott Siers Isreal Stolero Arun Subbiah Satish Damaraju, Varghese George. A 22nm ia multi-cpu and gpu system-on-chip. In *IEEE Interneational Solid-State Circuits Conference*, 2012.

[38] Tyrone Tai-On Kwok and Yu-Kwong Kwok. On the design, control, and use of a reconfigurable heterogeneous multi-core system-on-a-chip. In *In IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008.

[39] Said Hamdioui, Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, Henk Corporaal, Hailong Jiao, Francky Catthoor, Dirk Wouters, Linn Eike, and Jan van Lunteren. Memristor based computation-in-memory architecture for data-intensive applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1718–1725. EDA Consortium, 2015.

[40] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[41] G. A. Gibson Patterson, D. A. and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). pages 109–106, 1988.

[42] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[43] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31: 532–533, 1988.

[44] David J. Kuck. A survey of parallel machine organization and programming. *ACM Comput. Surv.*, 9(1):29–59, 1977.

[45] Fazal Hameed, Lars Bauer, and Jörg Henkel. Reducing inter-core cache contention with an adaptive bank mapping policy in dram cache. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1:1–1:8, Piscataway, NJ, USA, 2013. IEEE Press.

[46] Hsien-Kai Kuo, Bo-Cheng Charles Lai, and Jing-Yang Jou. Reducing contention in shared last-level cache for throughput processors. *ACM Trans. Des. Autom. Electron. Syst.*, 20(1):12:1–12:28, 2014.

[47] Changhe Song, Yunsong Li, Jie Guo, and Jie Lei. Block-based two-dimensional wavelet transform running on graphics processing unit. *Computers Digital Techniques, IET*, 8(5):229–236, 2014.

[48] Michael Sherry and Andy Shearer. Impair: massively parallel deconvolution on the gpu. volume 8655, pages 86550Q–86550Q–7, 2013.

[49] Joaquín Franco, Gregorio Bernabé, Juan Fernández, and Manuel Ujaldón. The 2d wavelet transform on emerging architectures: Gpus and multicores. *Journal of Real-Time Image Processing*, 7(3):145–152, 2011.

[50] W.J. van der Laan, A.C. Jalba, and J.B.T.M. Roerdink. Accelerating wavelet lifting on graphics hardware using cuda. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):132–146, 2011.

[51] Kenneth E. Batcher. Architecture of a massively parallel processor. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 174–179, New York, NY, USA, 1998. ACM.

[52] James T. Klosowski and Shankar Krishnan. Real-time image deconvolution on the gpu. volume 7872, pages 78720H–78720H–15, 2011.

[53] A. F. Boden, R. J. Hanisch, J. Mo, and D. C. Redding. Massively parallel spatially variant maximum-likelihood restoration of hubble space telescope imagery. *J. Opt. Soc. Am. A*, 13(7):1537–1545, 1996.

[54] Mohammad Faisal, Richard L. White, Aaron D. Lanterman, and Donald L. Snyder. Implementation of a modified richardson-lucy method for image restoration on a massively parallel computer to compensate for space-variant point spread of a charge-coupled-device camera. *J. Opt. Soc. Am. A*, 12(12):2593–2603, 1995.

[55] Adam W. M. van Eekeren, Klamer Schutte, Judith Dijk, Piet B. W. Schwering, Miranda van Iersel, and Niek J. Doelman. Turbulence compensation: an overview. volume 8355, pages 83550Q–83550Q–10, 2012.

[56] Bo Chen, Ze-xun Geng, Tian-Shuang Shen, and Yang Yang. Wavelet-based adaptive regularization deconvolution for turbulence-degraded image. volume 7157, pages 71570I–71570I–11, 2008.

[57] Oren Haik and Yitzhak Yitzhaky. Improvement of automatic acquisition of moving objects in long-distance imaging by blind image restoration. volume 6737, pages 67370O–67370O–11, 2007.

[58] Yu-Wing Tai, Ping Tan, and M.S. Brown. Richardson-lucy deblurring for scenes under a projective motion path. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(8):1603–1618, 2011.

[59] Shijie Sun, Huaici Zhao, Bo Li, Mingguo Hao, and Jinfeng Lv. Kernel estimation for robust motion deblurring of noisy and blurry images. *Journal of Electronic Imaging*, 25(3), 2016.

[60] Basel Salahieh, Jeffrey J. Rodriguez, Sean Stetson, and Rongguang Liang. Single-image full-focus reconstruction using depth-based deconvolution. *Optical Engineering*, 56(4), 2016.

[61] H. Navarro, G. Saavedra, M. Martinez-Corral, M. Sjöström, and R. Olsson. Extended depth-of-field in integral imaging by depth-dependent deconvolution. volume 8648, pages 86481H–86481H–8, 2013.

[62] Arnold W. M. Smeulders, Marcel Worring, Simone Santini, Amarnath Gupta, and Ramesh Jain. Content-based image retrieval at the end of the early years. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(12):1349–1380, 2000.

[63] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013168728X.

[64] Mysore Y. Jaisimha, Eve A. Riskin, Richard E. Ladner, and Werner Stuetzle. Model-based restoration of document images for ocr. volume 2660, pages 297–308, 1996.

[65] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[66] Holger R. Roth, Amal Farag, Le Lu, Evrim B. Turkbey, and Ronald M. Summers. Deep convolutional networks for pancreas segmentation in ct imaging. volume 9413, pages 94131G–94131G–8, 2015.

[67] I. B. Gurevich, O. Salvetti, and Yu. O. Trusova. Fundamental concepts and elements of image analysis ontology. *Pattern Recognition and Image Analysis*, 19 (4), 2009.

[68] Sara Colantonio, Igor Gurevich, Massimo Martinelli, Ovidio Salvetti, and Yulia Trusova. Thesaurus-based ontology on image analysis. In *Semantic Multimedia: Second International Conference on Semantic and Digital Media Technologies, SAMT 2007, Genoa, Italy, December 5-7, 2007. Proceedings*, pages 113–116, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[69] J.L. Starck and F. Murtagh. *Astronomical Image and Data Analysis*. Springer-Verlag, 2002.

[70] M Bertero, P Boccacci, G Desiderà, and G Vicidomini. Image deblurring with poisson data: from cells to galaxies. *Inverse Problems*, 25(12), 2009.

[71] Anconelli, B., Bertero, M., Boccacci, P., and Carbillet, M. Restoration of interferometric images - iv. an algorithm for super-resolution of stellar systems. *A&A*, 431 (2):747–755, 2005.

[72] I Irum, MA Shahid, M Sharif, and M Raza. A review of image denoising methods. *Journal of Engineering Science and Technology Review*, 8(5):41–48, 2015.

[73] C Xu O Haeberlé J J Meyer A Chomik, A Dieterlen and S Jacquey. Quantification in optical sectioning microscopy: a comparison of some deconvolution algorithms in view of 3d image segmentation. *Journal of Optics*, 28.

[74] Eli Chen, Oren Haik, and Yitzhak Yitzhaky. Classification of moving objects in atmospherically degraded video. *Optical Engineering*, 51(10):101710–1–101710–14, 2012.

[75] Eli Chen, Oren Haik, and Yitzhak Yitzhaky. Classification of moving objects in atmospherically degraded video. *Optical Engineering*, 51(10):101710–1–101710–14, 2012.

[76] M. Hawrylycz, D. Feng, C. Lau, C. Kuan, J. Miller, C. Dang, and L. Ng. Large scale digital atlases in neuroscience. volume 9034, pages 90340Z–90340Z–12, 2014.

[77] William Hadley Richardson. Bayesian-based iterative method of image restoration. *Journal of the Optical Society of America*, 62(1):55–59, 1972.

[78] L. B. Lucy. An iterative technique for the rectification of observed distributions. *Astronomical Journal*, 1974.

[79] L.A. Shepp and Y. Vardi. Maximum likelihood reconstruction for emission tomography. *Medical Imaging, IEEE Transactions on*, 1(2):113–122, 1982.

[80] I Irum, MA Shahid, M Sharif, and M Raza. A review of image denoising methods. *Journal of Engineering Science and Technology Review*, 8(5):41–48, 2015.

[81] Mukesh C Motwani, Mukesh C Gadiya, Rakhi C Motwani, and Frederick C Harris. Survey of image denoising techniques. In *Proceedings of GSPX*, pages 27–30, 2004.

[82] David L Donoho and Iain M Johnstone. Ideal spatial adaptation by wavelet shrinkage. *biometrika*, pages 425–455, 1994.

[83] David L Donoho and Iain M Johnstone. Adapting to unknown smoothness via wavelet shrinkage. *Journal of the american statistical association*, 90(432):1200–1224, 1995.

[84] S. G. Chang, Bin Yu, and M. Vetterli. Adaptive wavelet thresholding for image denoising and compression. *IEEE Transactions on Image Processing*, 9(9):1532–1546, 2000.

[85] V. Strela, P. N. Heller, G. Strang, P. Topiwala, and C. Heil. The application of multiwavelet filterbanks to image processing. *IEEE Transactions on Image Processing*, 8(4):548–563, 1999.

[86] A. Pizurica, W. Philips, I. Lemahieu, and M. Acheroy. A joint inter- and intrascale statistical model for bayesian wavelet based image denoising. *IEEE Transactions on Image Processing*, 11(5):545–557, 2002.

[87] Samuel P. Kozaitis and Tim Young. Thresholding for higher-order statistical denoising. volume 6979, pages 69790O–69790O–10, 2008.

[88] Brian M. Northan. Fuzzy logic components for iterative deconvolution systems. volume 8589, pages 858903–858903–12, 2013.

[89] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, Washington, DC, USA, 1998. IEEE Computer Society.

[90] Akira Taguchi, Hironori Takashima, and Yutaka Murata. Fuzzy filters for image smoothing. volume 2180, pages 332–339, 1994.

[91] Yaniv Romano and Michael Elad. Boosting of image denoising algorithms. *SIAM Journal on Imaging Sciences*, pages 1187–1219, 2015.

[92] Imola K. Fodor and Chandrika Kamath. Denoising through wavelet shrinkage: an empirical study. *Journal of Electronic Imaging*, 12(1):151–160, 2003.

[93] P. H. van Cittert. Zum einfluß der spaltbreite auf die intensitätsverteilung in spektrallinien. ii. *Zeitschrift für Physik*, 69(5):298–308, 1931.

[94] David A. Agard, Yasushi Hiraoka, and John W. Sedat. Three-dimensional microscopy: Image processing for high resolution subcellular imaging. volume 1161, pages 24–30, 1989.

[95] L I Rudin, S Osher, and E Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1-4):259–268, 1992.

[96] Andrew Shearer, Gerard Gorman, Triona O'Doherty, Wilhelm J. van der Putten, Peter McCarthy, and Lukasz Jelen. Parallel image restoration with spatially variant point spread function: description and first clinical results. *Proc. SPIE*, pages 787–795, 2001.

[97] Nicolas Dey, Laure Blanc-Feraud, Christophe Zimmer, Pascal Roux, Zvi Kam, Jean-Christophe Olivo-Marin, and Josiane Zerubia. Richardson–lucy algorithm with total variation regularization for 3d confocal microscope deconvolution. *Microscopy Research and Technique*, 69(4):260–266, 2006.

[98] Blind deconvolution by means of the richardson–lucy algorithm. *Journal of the Optical Society of America*, 12(1), 1995.

[99] J.-L. Starck and F. Murtagh. Image restoration with noise suppression using the wavelet transform. *Journal of Astronomy and Astrophysics*, 288:342–348, 1994.

[100] Jacques Boutet de Monvel, Sophie Le Calvez, and Mats Ulfendahl. Image restoration for confocal microscopy: Improving the limits of deconvolution, with application to the visualization of the mammalian hearing organ. *Biophysical Journal*, 80(5): 2455–2470, 2001.

[101] Ali Mosleh, J. M. Pierre Langlois, and Paul Green. *Image Deconvolution Ringing Artifact Detection and Removal via PSF Frequency Analysis*, pages 247–262. Springer International Publishing, Cham, 2014.

[102] D. Kundur and D. Hatzinakos. Blind image deconvolution. *IEEE Signal Processing Magazine*, 13(3):43–64, 1996.

[103] D. Kundur and D. Hatzinakos. Blind image deconvolution revisited. *IEEE Signal Processing Magazine*, 13(6):61–63, 1996.

[104] G. R. Ayers and J. C. Dainty. Iterative blind deconvolution method and its applications. *Opt. Lett.*, 13(7):547–549, 1988.

[105] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In *In NIPS*, pages 556–562. MIT Press, 2000.

[106] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, and F. Tirado. Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. *Parallel and Distributed Systems, IEEE Transactions on*, 19(3): 299–310, 2008.

[107] A. F. Boden, R. J. Hanisch, J. Mo, and D. C. Redding. Massively parallel spatially variant maximum-likelihood restoration of hubble space telescope imagery. *J. Opt. Soc. Am. A*, 13(7):1537–1545, Jul 1996.

[108] Mohammad Faisal, Richard L. White, Aaron D. Lanterman, and Donald L. Snyder. Implementation of a modified richardson-lucy method for image restoration on a massively parallel computer to compensate for space-variant point spread of a charge-coupled-device camera. *J. Opt. Soc. Am. A*, 12:2593–2603, Dec 1995.

[109] Michael L. Cobb, Paul L. Hertz, Robert O. Whaley, and Eric A. Hoffman. Space-variant point-spread-function deconvolution of hubble imagery using the connection machine. volume 2029, pages 202–208, 1993.

[110] Zheng Huang, Jingxin Zhang, and Cishen Zhang. Ultrasound image reconstruction by two-dimensional blind total variation deconvolution. Dec 2009.

[111] R. Zanella, G. Zanghirati, R. Cavicchioli, L. Zanni, P. Boccacci, M. Bertero, and G. Vicidomini. Towards real-time image deconvolution: application to confocal and sted microscopy. *Scientific Reports*, 3, 2013.

[112] Marc A. Bruce and Manish J. Butte. Real-time gpu-based 3d deconvolution. *Opt. Express*, 21(4):4766–4773, 2013.

[113] D. Krishnan and R. Fergus. Fast image deconvolution using hyper-laplacian priors. *NIPS*, pages 1033–1041, 2009.

[114] L. Domanski, P. Vallotton, and D. Wang. Two and three-dimensional image deconvolution on graphics hardware. In *Proceedings of the 18th World IMACS/MODSIM Congress*, pages 1010–1016, 2009.

[115] L. Domanski, T. Bednarz, P. Vallotton, and J. Taylor. Heterogeneous parallel 3d image deconvolution on a cluster of gpus and cpus. In *19th International Congress on Modelling and Simulation*, pages 613–619. Modelling and Simulation Society of Australia and New Zealand, 2011.

[116] S Bonettini, R Zanella, and L Zanni. A scaled gradient projection method for constrained image deblurring. *Inverse Problems*, 25(1), 2009.

[117] Prato, M., Cavicchioli, R., Zanni, L., Boccacci, P., and Bertero, M. Efficient deconvolution methods for astronomical imaging: algorithms and idl-gpu codes. *A&A*, 539:A133, 2012.

[118] Symeon Charalabides, Andrew Shearer, and Raymond F. Butler. Application of deconvolution to images from the egret gamma-ray telescope. *Proc. SPIE*, pages 213–220, 2003.

[119] Triona O'Doherty, Andrew Shearer, Wilhelm J. van der Putten, and Phillip Abbott. Image deconvolution as an aid to feature identification: a clinical trial. *Proc. SPIE*, pages 1469–1475, 2000.

[120] Phillip Abbott, Andrew Shearer, Triona O'Doherty, and Wil van der Putten. Image deconvolution as an aid to mammographic artifact identification: I. basic techniques. *Proc. SPIE*, pages 698–709, 1999.

[121] M. Bertero and P. Boccacci. A simple method for the reduction of boundary effects in the richardson-lucy approach to image deconvolution. *Astronomy and Astrophysics*, 437:369–374, 2005.

[122] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[123] NVIDIA Corp. *CUDA CUFFT Library*, 2007.

[124] H.J. Trussell and B.R. Hunt. Sectioned methods for image restoration. *IEEE Trans. Acoust., Speech, Signal Process.; (United States)*, ASSP-26:2, 1978.

[125] J. Proakis and D. Manolakis. *Digital Signal Processing.* Prentice-Hall, 1996.

[126] Y. Nieverge. *Wavelets made easy.* Birkhäuser,, Boston, 2001.

[127] David L. Donoho and Iain M. Johnstone. Ideal spatial adaptation by wavelet shrinkage. *Biometrika*, 81:425–455, 1994.

[128] M. J. Shensa. The discrete wavelet transform: wedding the a trous and mallat algorithms. *IEEE Transactions on Signal Processing*, 40(10):2464–2482, 1992.

[129] N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, University of Cambridge, 2004.

[130] Intel 64 and ia-32 architectures optimization reference manual, order number: 248966-033. pages 53–55, Jun 2016.

[131] Intel 64 and ia-32 architectures optimization reference manual, order number: 248966-033. pages 59–60, 277, Jun 2016.

[132] Intel 64 and ia-32 architectures optimization reference manual, order number: 248966-033. pages 140, 278, Jun 2016.

[133] David S. Taubman and Michael W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[134] J.D. Villasenor, B. Belzer, and J. Liao. Wavelet filter evaluation for image compression. *Image Processing, IEEE Transactions on*, 4(8):1053–1060, 1995.

[135] Jean-Luc Starck, Mai K. Nguyen, and Fionn Murtagh. Wavelets and curvelets for image deconvolution: A combined approach. *Signal Process.*, 83(10):2279–2283, 2003.

[136] T. Blu and F. Luisier. The sure-let approach to image denoising. *IEEE Transactions on Image Processing*, 16(11):2778–2786, 2007.

[137] Soren Hauberg John W. Eaton, David Bateman and Rik Wehbring. *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations.* 2015.

[138] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008.

[139] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.