DEPARTMENT OF ELECTRONIC AND ELECTRICAL ENGINEERING

UCL

Software-based Approximate Computation Of Signal Processing Tasks

Davide Anastasia

A thesis submitted for the degree of Doctor of Philosophy

Department of Electronic and Electrical Engineering
University College London

May 2012

Statement of Originality

I declare that the work presented in this thesis and the thesis itself was composed and originated by myself in the Department of Electronic and Electrical Engineering, University College London. The work of other persons is appropriately acknowledged.

Davide Anastasia

Abstract

This thesis introduces a new dimension in performance scaling of signal processing systems by proposing software frameworks that achieve increased processing throughput when producing approximate results.

The first contribution of this work is a new theory for accelerated computation of multimedia processing based on the concept of tight packing (Chapter 2). Usage of this theory accelerates small-dynamic-range linear signal processing tasks (such as convolution and transform decomposition) that map integers to integers, without incurring any accuracy loss.

The concept of tight packing is combined with incremental computation that processes inputs in a bitplane-by-bitplane manner (Chapter 3), thereby leading to substantial throughput/distortion scalability within filtering, transform-decomposition and motion-estimation tasks. This framework also provides for region-of-interest computation and has inherent robustness to arbitrary termination of processing, imposed, for example, by a task scheduler.

Finally, the concept of packed processing is extended to floating-point (lossy) matrix computations, with particular focus on the generic matrix multiplication (GEMM) routine of BLAS-3 (Chapters 4 and 5). This routine is a fundamental building block for several linear algebra and digital signal processing systems, such as face recognition and neural-network training for metadata-based retrieval systems. In order to compete with the best-performing software designs for GEMM, an implementation using single instruction, multiple data (SIMD) instructions is presented and analysed. The proposed approach demonstrates substantial performance scaling in practice; specifically, it is shown to achieve up to twice the processing throughput of the best designs for GEMM when producing approximate results (under the same hardware).

In summary, the proposed approximate computation of signal processing tasks can be selectively disabled thereby producing conventional full-precision/lower-throughput processing when deemed necessary. Importantly, the proposed software designs run on off-the-shelf computer hardware and provide for on-demand reconfiguration, depending on the input data and the precision specification (from full precision to noisy computation). Thus, the proposed approximate computation framework allows for backward compatibility and can be offered as an add-on service, creating significant competitive advantages for application developers. It can be used in mobile or high-performance computing systems when the precision of computation is not of critical importance (error-tolerant systems), or when the input data is intrinsically noisy.

Acknowledgements

This thesis is the end of a journey that began many years ago. From a small city in South Italy, this journey brought me first to an important university of North Italy and then to one of the most vibrant universities in the world. The last four years have been full of rewarding moments, as well as tough ones. However, thanks to the support of family, friends and special people, I have never given up. Hence I wish to express my sincere appreciation to those who have contributed to this thesis and supported me in one way or another during this amazing journey.

Foremost, I would like to express my sincere gratitude to my supervisor, Professor Yiannis Andreopoulos. It has been an honour to be his first PhD student, and it is only thanks to his guidance, enthusiasm and technical knowledge that I have achieved results that I otherwise wouldn't have. His help has been important in many moments during these years, especially when friendship was more needed than technical advice.

I would also like to thank Fabio Verdicchio, whose support was especially important when I had just moved to London and I needed it most. His technical help was always enlightening and precise making him a valuable advisor and more than that, over time, a good friend.

A mention goes to my lifetime friend Giusi: she convinced me to continue my academic career when I was close to giving up, to fill in applications and, at the end, to start my PhD research.

I would like to thank my parents: when I told them that I was leaving Italy, they thought I was completely crazy, but they supported me every day and they can now say to have a “Doctor” in the family (but I am not sure whether they still think I am crazy or not).

The last mention, the most important one, goes to my wonderful girlfriend Laura. She paid the highest price of my decision to move to London, but never failed to support and cheer me up when I was down, or to celebrate the good moments with me. While she might not understand most of this pile of formulas, charts and tables, this thesis is dedicated to her nonetheless.

Contents

List of Figures

List of Tables

Acronyms

CIF common interchange format. 76, 77

DCT discrete cosine transform. 25

DFT discrete Fourier transform. 27

DSP digital signal processing. 110, 128

DVS dynamic voltage scaling. 26

FFT fast Fourier transform. 29

FPGA field-programmable gate array. 30, 31

FPU floating-point processor unit. 49

FXP fixed-point. 77

GEMM generic matrix multiplication. 2, 23, 31, 34–36, 102, 103, 127, 130, 131, 133, 137, 148

GPU graphics processing unit. 30, 31

HPC high performance computing. 30, 33

i.i.d. independent and identically distributed. 35, 115, 118, 120, 151

KLT Karhunen-Loeve transform. 25

LSB least significant bit. 57, 58

MAC multiplier-accumulator. 106, 108, 110

MLP multi-layer perceptron. 146, 147

MSB most significant bit. 57, 58

MSE mean squared error. 23, 114, 115, 119, 136, 142

MSSIM mean SSIM. 84

PCA principal component analysis. 139, 143, 145

PSNR peak signal-to-noise ratio. 76, 77

QCIF quarter-CIF. 76, 77

ROI region-of-interest. 91

RV random variable. 115, 118

SIMD single instruction, multiple data. 2, 108, 110, 128, 130

SNR signal-to-noise ratio. 23, 76, 77, 84, 100, 120, 123

SSE streaming SIMD extensions. 102, 104, 127, 130, 133, 134, 148

SSIM structural similarity index measure. 84, 100

STFT short-time Fourier transform. 27

SVD singular value decomposition. 103

TLB translation lookaside buffer. 31, 130

Nomenclature

$\overline{\mathbf{A}}$ Packed block. 107, 108

$\tilde{a}$ Companded value of the quantity a based on the companding coefficient $c_{\mathbf{A}}$. The same notation is extended to matrices. 106

a^n_{bit} Computed value of a when only bitplane n of the input is used, with $0 \leq n < N-1$. The same notation is extended to matrices. 57

a^n_{full} Computed value of a when the input consists of bitplanes $N-1$ down to (and including) bitplane n, with $0 \leq n < N-1$. The same notation is extended to matrices. 57

$c_{\mathbf{A}}$ Companding coefficient for the matrix $\mathbf{A}$. 106

d Packing coefficient. 47–49, 73

g Number of bitplanes packed in each increment layer n, e.g. $g = 1$ when a single biplane is packed in each increment layer. 62, 65

R_{max} Maximum possible value of the result of the operation op applied to operands of known dynamic range. 41, 73

R_{min} Minimum possible value of the result of the operation op applied to operands with of known dynamic range. 41

R_{range} Range of the possible value of the result of the operation op applied to operands of known dynamic range. 42

r_{type} Representation type: equal to -1 for 64-bit floating-point and to 1 for unsigned integer. 47

$\hat{u}_{\text{safe}}$ Experimental value for u^*_{safe}. 49

u^*_{safe} Theoretical minimum safe padding that assures errorfree packed processing. 43

u_{sys} Relative relative precision of the computer used for the implementation. 42, 49

W Packing capability. 39, 42, 49, 50, 81, 100

w_{type} Representation width: approximately equal to 50 for 64-bit floating-point and equal to 31 for unsigned integer. 47

z Packing factor. 39, 40, 42, 47, 110

Chapter 1

Introduction and Literature Review

One of the fundamental areas of Information and Communications Technologies (ICT) is computer-based information processing. Several popular ICT applications, such as media players, computer graphics, image and video post-processing, and motion estimation and compensation, are implemented today via software solutions in programmable processors.

Advances in this area hinge on the premise of inexpensive doubling of the processing capability of microprocessors every $18 \sim 24$ months. However, today it is widely acknowledged that this is threatened by fundamental limitations of silicon-based transistor integration, which lead to excessive energy dissipation and unacceptable fault rates for future microprocessors [1, 2]. In a last strive to avoid such limitations, the microprocessor industry has extended conventional single-processor architectures to networks of processors (cores), ranging from 2 to 8 large cores (multi-core) up to 4096 small core units (many-core) [2].

Despite this increase in core numbers and core diversity, today there is very little synergy between the system layer (software design, processor, task manager) and the multimedia application layer (e.g. image processing task, such as filtering). For example, if one is watching a movie on a portable video player (e.g. [3]) and this is draining the system resources (battery), current systems do not allow for seamless trade-offs in visual quality vs. battery life (execution time per task). In such cases, the user is practically facing the on/off approach of digital systems, while one would strongly opt for a best-effort approach, often found in analogue systems, where energy autonomy would be increased with graceful degradation in the decoded video quality.

At the other end of the service deployment spectrum we have ultra-large scale multimedia content analysis, indexing and retrieval services like Google Image Search, social networking and automated media tagging, webpage ranking algorithms, etc. These are deployed in large server-based clusters [4, 5]. While all such algorithms aim for average error and not for the worst-case (e.g. expected recall rate of misclassification percentage), all systems today implement such algorithms without considering their precision aspects. Nevertheless, while an individual user would not notice a drop on the image recall rate by a few percentile points, he/she *will notice* server outages caused by exceeding the system's processing capacity.

The work described in this thesis aims to systematically trade-off quality of information processing services when the target application can cope with lower accuracy. It goes beyond traditional throughput scaling approaches by parallelisation, by introducing a new scaling dimension for a range of signal processing and linear algebra computations: novel computational frameworks are proposed that achieve increased processing throughput by producing approximate outputs [6, 7, 8, 9, 10]. These can be used in mobile or high-performance computing systems when the precision of computation is not of critical importance (error-tolerant systems), or when the data is intrinsically noisy [9]. Importantly, such computational approaches run on off-the-shelf computer hardware and they provide for on-demand reconfiguration, depending on the input data and the precision specification (from full precision to noisy computation). Finally, it has also been established that this approach can run on a fault-generating computer system, such as a modern processor under aggressive voltage scaling or under an operating environment enforcing aggressive thread scheduling.

Beyond the theoretical and concept study, this research is brought to a proof-of-concept software design for *image processing* operations within a variety of applications, like block decomposition and convolution [7, 11], with really encouraging results in throughput scalability, power consumption and real-time scheduling. Essentially, this approach treats processing resources as a *communication channel*: when lower signal quality is required from the application, the computation is accelerated (useful in high performance computing applications or multimedia) or requires less power (useful in high-end server clusters or mobile computing environments).

Beyond multimedia-oriented software designs, this method has then been developed for GEMM [8, 9], which is a fundamental building block for several *digital signal processing* routines, such as transform decomposition, LU factorisation and linear system solvers [12]. GEMM is also the core element within more complex systems, such as face recognition software [13]. This newly designed implementation has then been used inside multimedia and computer vision applications (like video processing and face recognition), which require high performance but can tolerate errors and approximate results. For instance, by adjusting the imprecision introduced by the approximate GEMM to match the inherent acquisition noise of images within a state-of-the-art face recognition system, we demonstrate up to 100% increase in the processing throughput with virtually no effect in the recognition accuracy. This means that a high-performance computing cluster using the proposed GEMM design for face recognition can handle twice the number of input images per second in comparison to using the conventional GEMM of a high-performance library like ATLAS or GOTO [14, 15]. Other systems of this type can also take advantage of similar performance gains (a face recognition system is a particular case of an object identification system [16, 17]).

The proposed approximate computation can be selectively disabled thereby producing conventional full-precision/lower-throughput processing when considered necessary. Thus, such an approximate computation framework allows for backward compatibility and can be offered as an add-on service, creating significant competitive advantages for application developers.

Overall, the proposed frameworks of this thesis can be used to target a particular signal-to-noise ratio (SNR) or mean squared error (MSE) – against the result computed at full precision – for a linear operation used within a multimedia or numerical processing system. The impact of this approximation on the final results of each application is assessed on a case-by-case basis via experimentation with representative inputs and outputs. This is directly analogous to rate reduction via lossy source coding for multimedia, where all lossy source coding algorithms are optimized for SNR or MSE distortion within particular subsets of the input (macroblocks in video, image pixels after quantization, etc.) and the impact of such distortion on the visual quality of each algorithm is assessed on a case-by-case basis within each coding application (e.g. video streaming, video playback, etc.). Hence, the proposed software-based approaches for approximate computation take as input the required approximation in SNR or MSE, or the portion of the input bitplanes to process, and derive the best possible acceleration under these constraints. The impact of such approximate computation on the results of several multimedia processing algorithms is assessed on a case-by-case basis via experiments with representative inputs and it is not linked to the precision of the approximate computation in an analytic manner.

In a practical deployment within a particular system and application, this would entail the empirical matching of the acceptable accuracy of the application and the acceleration obtained from the utilized system with the operational settings of the approximate computation being used. This is commonly done in practical deployments of lossy source coding standards, where visual quality and rate requirements of the application are linked in an empirical manner to the operational settings of the rate-distortion optimized image or video coding algorithm being used.

1.1 Literature Review

The proposals of this thesis involve a variety of methods, hence reviewing the research literature related to each method is necessary.

This section starts by reviewing algorithm-specific implementations with complexity/precision scalability in Subsection 1.1.1; this will serve as reference for the proposals of Chapter 3. We then review different approaches for approximate computation, incremental computation and memory compression schemes in Subsection 1.1.2 and 1.1.3; these approaches are related to the basic ideas proposed in Chapter 2 and 4 of this thesis. In Subsection 1.1.3 we summarise different data representation schemes; such schemes comprise the foundation of the description of the loose and tight packing used in Chapter 2 and Chapter 4. Finally a review of high performance computing techniques and software systems is given in Subsection 1.1.4 and 1.1.5, which strongly relates to the experimental development of the proposals of Chapter 4 and Chapter 5 of this thesis.

1.1.1 Algorithm-specific implementations with complexity/precision scalability

Existing algorithm-oriented research focuses on complexity reduction [18, 19, 20] or complexity scalability for image processing tasks [21, 22, 23], where computational complexity is decreased and approximate results are produced.

In complexity/distortion research, Goyal and Vetterli [18] study the relationship between computational complexity and coding performance in a systematic way. They focused on the Karhunen-Loeve transform (KLT) and the discrete cosine transform (DCT), two well known block decomposition transforms to code a Gauss-Markov source and concluded that it is possible to create a framework for complexity/distortion processing analogous to a framework for rate-distortion coding.

A content-based approach is proposed in [19], where the complexity of the encoding is calculated based on the estimation of the video scene content. Results for the Dynamic Closest Checking Point (DCCP) algorithm are presented, demonstrating six-fold increase in execution time (in the best case) compared with the full search. In [20] a different metric for quantifying motion estimation complexity is proposed, in order to decrease execution time. Lengwehasatit and Ortega [21] propose an algorithm specific implementation for the DCT, which is a hybrid between the frequency selection (only a subset of

DCT coefficients is considered) and the accuracy selection (DCT coefficients are computed at reduced accuracy). In a similar way, [22] presents a complexity-scalable scheme for motion estimation using different coding mode and spatio-temporal decomposition structures. Finally, [23] introduces the concept of incremental refinement using different portion of the input, but it does not show how it can be realised in a real system and what kind of performance scaling can be achieved. Similar works can be found in the literature on fine-tuning of well-known algorithms, like DCT decomposition algorithms [24, 25], or block motion estimation [26], in order to achieve higher performance. While significant performance scaling can be achieved in some cases, these techniques are algorithm-specific. As such, they require significant algorithm-based customisation in order to provide complexity scalability, which can be cumbersome in a general service deployment environment (e.g. multicore processors).

Hardware-oriented research on algorithm complexity scalability focuses on multimedia-driven energy scaling of processors via dynamic voltage scaling (DVS) in an attempt to provide energy consumption scaling with approximate results. For instance, Yuan and Nahrstedt [27] propose a voltage-scaling method to minimise the total energy consumption while still meeting multimedia timing requirements. Similarly, Akyol and van der Schaar [28] present a model to adapt voltage/frequency in order to adapt dynamically to the workload. In their results, a DVS method for video decoding was used, demonstrating that a specialised version of the algorithm achieves reduction in energy consumption compared to conventional DVS algorithms that do not consider the precision of multimedia processing tasks.

Overall, for all existing approaches: (*a*) algorithm-specific and/or system-specific customisations are required, which limit the applicability of the proposed techniques; (*b*) only one operational point in the complexity/distortion curve [18, 22] can be obtained, i.e. one is not able to seamlessly increment the quality of the output with increased computation. The latter means that complex hardware and software reconfigurations are required when different throughput in frames-per-second (fps) is required. Hence, application scalability and robustness is not obtained instantaneously and in a natural and straightforward manner.

1.1.2 Approximate and Incremental Computation

The concept of incremental computation is well known for many years now (e.g. iterative system solvers [12]). The term is used in different areas with slightly different meanings, but it always refers to the capability of a certain algorithm to improve the accuracy of its result progressively, via a number of iterations. There is an important difference between incremental computation and incremental processing that is discussed in the following two paragraphs.

Theoretical proposals for *incremental computation* have been made for signal transforms and salient point detection algorithms [29, 30, 31], where the main principle is: under a refinement of the input source description (e.g. image pixel refinement), the computation of the image processing task refines the previously-computed result. Winograd and Nawab [29] present a theoretical framework for generating discrete Fourier transform (DFT) and short-time Fourier transform (STFT), Andreopoulos and van der Schaar [30] present the discrete wavelet transform computed in an incremental way, Andreopoulos and Patras [31] show a similar approach applied to salient point detection algorithms in images. However, these works are only using arithmetic complexity estimates, such as the expected bit switching activity for additions and multiplications, and no practical realisations are proposed. An exception to this is the work of Chandrakasan et al. [32, 33, 34] on hardware designs for data-driven computation where incremental computation of the DCT and FIR filters can be supported by hardware designs based on distributed arithmetic. Nevertheless, such approaches require custom-hardware designs and cannot be deployed on commodity processors. However, in some cases, a trade-off between custom-hardware designs and a pure software-based approach can be found using advanced FPGA synthesis techniques. Constantinides et al. [35] propose an approach to the wordlength allocation and optimization problem for linear digital signal processing systems. This approach allow the user to trade-off implementation area (between 6% and 45%) with a speed increase compared to the optimum uniform wordlength design. More recently, a method to exploit larger 6-LUTs in FPGAs has been proposed by Hutton et al. [36], against the belief that 4-LUTs are the most efficient for area/delay trade-off: this method instead shows a 15% performance increase with a 12% area decrease [36, 37]. Nevertheless, while these approaches can derive scalable resource-distortion computation, they cannot provide for incremental computation, where the quality of the result is improved with continued processing.

It is important to distinguish *incremental computation* and *incremental processing*. *Incremental processing* [38, 39] refers to progressive quantisation (and potentially entropy coding) which may or may not lead to an incrementally computable algorithm. Practical systems, such as progressive JPEG decoding [40] used in digital cameras, require additional resources to decode and display complete image bitstream in comparison to non-progressive (conventional) processing. However, they remain useful because they can decode a low-resolution result with low latency, which is useful for fast image browsing in digital cameras, for example.

The concept of incremental processing is also well known in the design of compilers: Sundaresh and Hudak [41] study how to derive an incremental version of an algorithm from a non-incremental description using a framework that could be applicable to other domains. It also aims to find a methodology to prove the correctness of the incremental approach given the non-incremental one. Continuing on the same route, Hoover [42] explains how the ideas of incremental processing and dynamic dependency analysis have been used to develop a program transformation tool.

However, this technique starts by assuming that global algorithms are applied to an entire dataset, even though this dataset can already be in a state that would facilitate the execution of a localised algorithm. For example, it assumes a global sort algorithm is executed after every insertion in a vector, even though a localised sort would suffice under the knowledge of the insertion point. This means that such code transformation tools will not lead to competitive solutions against the state-of-the-art implementation of the non-incremental version of the algorithm.

Many of these approaches have to solve the problem of the storage of the intermediate results. One method that has been proposed, called "static caching" [43, 44, 45], allows the storage of not only the final result of each intermediate state, but also of intermediate results for each stage that could be useful in the following ones, thanks to static program analysis and semantics-preserving program transformations. Overall, while all such approaches may offer incremental processing for certain algorithmic classes, this usually comes at significant implementation and memory cost that makes these approaches unsuitable for compute and memory-intensive processing [43, 44, 45].

1.1.3 Memory Compression and Data Representation

Starting from studies on what the best storage hierarchy could be for a particular algorithm [46, 47], many research works have focused on memory compression schemes. The bulk of the proposed methods target current high performance systems with the aim to narrow the gap between processors' performance and memory bandwidth [48], which has historically been a bottleneck in computing performance. Some of these schemes apply at the cache level [49, 50] and imply a slight redesign of the CPU in order to properly manage the cache. Specifically, Alameldeen and Wood [49] and de Castro et al. [50] propose an adaptive decompression strategy for the memory data cache in order to overcome the limitation in speed of a fully-compressed cache: while for low-cache-miss benchmark such a scheme degrades the performance up to 17%, for memory-intensive application the gain can be up to 18% [49]. Other methods are instead applied at the virtual memory level [51, 52]: they offer advantages for applications that need to move large quantities of data in order to perform their tasks (memory bounded applications), because they decrease the bandwidth necessary to copy from the swap memory (usually disks) into the main memory (or into the first level of the cache hierarchy): however, while in general this method works well with a multitude of applications, exceptions can be experienced in the several common application, like fast Fourier transform (FFT) [52]. Finally, a multitude of methods have been proposed for the compression of the main memory: Tuduce and Gross [53] present a memory compression solution for applications that use large data sets frequently and exhibit poor performance due to the excessive page faults. This method adapts the allocation of memory between uncompressed and compressed pages and also manages fragmentation without user involvement. The solution proposed by Ekman and Stenstrom [54] instead exploits a simple compression scheme, a highly-efficient structure for locating a compressed block in memory, and a hierarchical memory layout that allows compressibility of blocks to vary with a low fragmentation overhead. In this same area, Kjelso et al. [55] present a new design for main memory hardware data compression, showing its performance applied to a commonly used Unix application. Finally, by proposing an unified compression scheme for the entire memory hierarchy, Hallnor and Reinhardt [56] claim an effective increase of the on-chip cache capacity, off-chip bandwidth and main memory size, while avoiding compression and decompression overheads between levels. A similar mechanism can also be found in the Memory Expansion Technology (MXT)[1] developed by IBM [57, 58, 59]. A complete comparison of this hardware main memory compression is performed in [59] using SPEC CPU2000 [60], which is a complete set of computer programme benchmarks. A similar methodology has been used in [61] to measure the performance of Linux with a customised memory management module (developed by HP). Operating system support is in fact necessary, either if the memory compression is performed in hardware or in software [62].

In conclusion, it is important to mention that even the way data is stored is a key aspect for the efficiency of a certain algorithm. Many studies have been made in order to find out what could be the best method to store data in order to minimise read and write implementation complexity (e.g. simplify address calculation and minimise latency in data access): for example, chip producers like Motorola and Intel ended up using different technical solutions (big endian and little endian) for the memory organisation used by their processors.

[1]`http://mxt.sourceforge.net/`

1.1.4 High Performance Computing

High performance computing (HPC) refers to the use of supercomputers or computer clusters to solve advanced computational problems. The concept of the supercomputer has evolved in time, thanks to the advancements in new technologies, like multi-core processors [63], graphics processing units (GPUs) [64] and field-programmable gate arrays (FPGAs) [65]. Although compilers and software development methodologies have been proposed to try and harness the power of multicore and GPU [66], there is a point where adding more processor cores will not improve performance due to the cost of the interconnection and communication which cannot be parallelised (Amdahl's Law).

In terms of performance analysis, *SPEC CPU2000* has been created in order to define an industry standard to measure the computation performance of a certain machine across a whole range of stimulus [60]. Before the introduction of this standard (and others), throughput of supercomputers was measured using LINPACK [67], which is heavily based on BLAS (Basic Linear Algebra Subprograms)[2], and in particular on GEMM. LINPACK is still used within the measurement methodology performed to the get a new supercomputer listed in the TOP500 list (list of the fastest 500 supercomputers in the world)[3]. For this reason, GEMM still has an important place in the measurement of system performance [68]. Thus, many chip manufacturer invest heavily in fine tuning of BLAS routines in order to achieve the best possible performance with their processors [69, 70, 71]. At the same time, mathematicians and computer scientists contributed with several new ways of improve performance of GEMM [14, 15, 72, 73, 74] (hence BLAS and LINPACK and all the applications that use a set of those).

The majority of performance gains in GEMM computation came from two open source projects, ATLAS [14] and GOTO [15][4]: the first aims to create a

[2]A more extensive coverage of BLAS will be made in Chapter 5

[3]`http://www.top500.org/lists`

[4]However, even though ATLAS and BLAS are the most important projects in this field, many others contributed as well, like PHiPAC [75].

BLAS implementation that, during compilation time, selects the best internal structure for the target processor and memory architecture in order to achieve the maximum throughput; the second one instead used manually fine-tuned assembly code plus a novel approach that takes into account the translation lookaside buffer (TLB) into the memory hierarchy [72]. However, other approaches have also been proposed: Fatahalian et al. [76] explain what are the strengths and the weakness of a matrix-matrix multiplication performed with GPU because of GPU's different architecture compared to a mainstream general purpose processor; Underwood and Hemmert [77] and Dou et al. [78] show how FPGAs can be used in place of CPUs to perform matrix multiplication and BLAS routines.

However, instead of changing the architecture of the underlying hardware, in literature we can find a multitude of mathematical methods for fast matrix multiplication [12, 79]. The method proposed by Strassen [80] is surely the best known [81], as the one presented by Coppersmith and Winograd [82]. Both of these methods try to reduce the number of multiplications required for the exact computation of the matrix multiplication, in favour of additions and subtractions, thus decreasing the final complexity, with the aim to achieve the limit described by Coppersmith and Winograd [83][5].

A different branch of the search for a faster matrix multiplication is the one that goes in the direction of approximated result. For instance, one presented by Drineas and Kannan [85] comprises a fast Monte Carlo algorithm that down-samples the number of rows (and columns) of the input matrices before performing the matrix multiplication, hence decreases the number of operation performed and consequently the execution time. The authors also provides a proof for the error bound. This work is then extended in [86], where the authors present an extension of the original row-wise method that decreases the number of operations necessary to select the best subset of rows (or columns), called pass-efficient model, plus a new element-wise algorithm that randomly selects elements of the original matrices, scales them accordingly and zeroes

[5]However, the method described by Coppersmith and Winograd [82] only achieves a practical performance gain for matrices with dimensions larger than 1500 [81, 84].

all the others.

Many algorithms for HPC also adopt mixed-precision techniques: by running different portions of the same algorithm with different precision, the processing throughput can be increased in comparison to the same routine implemented in double-precision [87]. This method works particularly well when the difference in throughput between double-precision and single-precision implementation is high, as proven by the experiments conducted in [87]. Similar methods have been proposed in [88, 89]: Buttari et al. [88] study how mixed-precision iterative refinement techniques can be used to speed up the solution of dense linear systems, while maintaining the same precision, while Demmel et al. [89] instead apply a similar technique to overdetermined linear least squares (LLS) problems.

1.1.5 Fault-tolerant Computation

As technology continues to scale, and transistors dimension become smaller, they also become increasingly vulnerable to soft errors [2]. Soft errors comprise the phenomenon of an erroneous change in the logical value of a transistor, and can be caused by several effects, including fluctuations in signal voltage, noise in the power supply, inductive coupling effects, particle strikes [90, 91], etc. Soft errors can result in incorrect results, segmentation faults, application or system crash, or even the system entering an infinite loop. Solutions already proposed belong to different categories [92, 93], such as: (*a*) hardware detection and correction of soft errors [94, 95]; (*b*) cross-layer error resilience [96, 97]; (*c*) compiler techniques can reduce the impact of soft errors by changing the computation to use processor resources that are protected, and approaches that perform computation in a redundant fashion [98].

Overall, due to the predicted evolution of CMOS integration, techniques that facilitate fault tolerance in conjunction with high performance are of increasing importance in signal processing and linear algebra routines. As such, the proposals of this thesis that allow for decreased execution time under approximate results can become enablers for high-performance, fault-tolerant, computation. This can be performed either by trading off some of the obtained performance gains for resilience to computing faults, or by allowing for graceful degradation in the results' quality under computing faults. Some initial work concerning the latter is presented in Section 3.9.

1.2 Thesis Structure and Research Contributions

1.2.1 General Structure

The remaining chapters of this thesis are divided into two parts: the first part addresses the concept of progressive computations by proposing a unified software framework for image processing tasks exhibiting incremental refinement of computation. The proposed software designs of transform decompositions, two-dimensional (2D) convolution and block-matching operations combine incremental computation with a recently-proposed packing approach that enables the calculation of multiple limited dynamic-range integer operations via one 32-bit or 64-bit arithmetic operation. The proposed software designs are validated in two different systems and are also provided online [99]. This effort is reported in detail in our published work in [6, 7, 11].

In the second part, inspired by the concept of incremental processing and packing, a software-based approximate computation approach has been developed for GEMM, which is a fundamental building block for several digital signal processing routines (factorisation and system solving just to name a few). GEMM is also the core element within more complex systems, such as face recognition software. By systematically trading off precision in favour of processing throughput, this method is able to achieve different operational points in real-time. This effort is reported in details in our published work in [9, 10].

1.2.2 Detailed Structure

Because operational packing forms a key part of the proposals of this thesis, Chapter 2 presents the theoretical background on the *packing theory* as well as a novel proposal for operational *tight packing*, a result not previously known in literature and reported in our work in [6]. Packing theory provides a practical approach to calculate numerical signal processing algorithms using concurrent arithmetic processing. This concurrency is achieved with a pure software-based approach without the need for machine-specific customisation or custom hardware requirements.

Chapter 3 merges packing theory with recently-proposed approaches for incremental computation. This chapter focuses on the conversion of three basic building blocks for signal processing (convolution, block transformation and block matching) into an incremental computation model using operational packing. This synergy of packing and incremental computation enables new interesting perspectives:

- from a performance point of view, it allows for progressive calculation with virtually no loss in execution-time performance for the full-quality (entire) result as compared to the conventional (non progressive) calculation;
- from a distortion/complexity point of view, it can obtain a coarse (but useful) result of the computation at a fraction of the execution time required for the full calculation;
- from a functionality point of view, new adaptive computational models are enabled; one can compute in regions of interest or refine the computation based on previously-produced results.

The second point is particularly important in a real-time environment where a scheduler may allocate only a certain amount of execution time for the calculation of a single frame. These points are elaborated in Section 3.6 of this thesis, where a full set of experiments is presented specifically focusing on this topic.

A limitation of incremental computation as proposed in Chapter 3 is that it works only for integer-to-integer processing. This constraint is alleviated in Chapter 4 where, by dropping the constraints imposed by lossy-to-lossless progressive computations we propose a novel way to combine input companding and packing approaches. By selecting the correct operational settings (e.g. the number of input samples to be squeezed together using the theoretical results of Chapter 2, their "distance" within the packed representation and the loss of precision from companding in an analytical manner), a substantial increase in processing throughput can be achieved by trading off precision, hence the concept of *lossy packed processing* is proposed. The entire chapter focuses on the mathematical background behind the implementation of this newly developed method in a GEMM routine: a new revised lossy packed processing theory is presented first (starting from the lossless packed processing first presented in Chapter 2); then new packing techniques are presented, showing their relative strength and weakness; finally, a stochastic model of the interaction between the lossy packed processing and the internal GEMM structure is presented for independent and identically distributed (i.i.d.) inputs, thus enabling the analytic study of the distortion/throughput trade off and the analytic estimation of the best operational settings.

Chapter 5 complements the theory presented in Chapter 4 by presenting the key software techniques used for the implementation of this proof-of-concept approximate GEMM, along with a complete set of experiments. The experiments performed demonstrate both the stand-alone performance of the proposed approximate GEMM computation, as well as how such a component performs inside bigger systems (such as a face recognition system).

Finally Chapter 6 presents the overall conclusions of this thesis.

1.2.3 Research Publications

The work presented in this thesis has led to 3 journal publications, 4 conference publications and presentations, 1 best paper award and 1 shortlist for the EPSRC UK ICT Pioneers Competition 2011, plus some invited talks.

Conference publications:

- D. Anastasia and Y. Andreopoulos, “Operational Refinement of Image Processing: ORIP v1.0, ” in *Proc. of London Communications Symposium (LCS)*, London, UK, September, 2009
- D. Anastasia and Y. Andreopoulos, “Software designs of image processing tasks with incremental refinement of computation,” in *Proc. IEEE Workshop on Signal Processing Systems (SiPS), 2009*, Tampere, Finland, October 2009, pp. 249 –254.
- D. Anastasia and Y. Andreopoulos, “Scheduling and energy-distortion tradeoffs with operational refinement of image processing,” in *Proc. Design, Automation & Test in Europe (DATE)*, Dresden, Germany, March 2010, pp. 1719 –1724.
- D. Anastasia and Y. Andreopoulos, “Throughput-precision computation for generic matrix multiplication: Toward a computation channel for high-performance digital signal processing,” in *17th International Conference on Digital Signal Processing (DSP)*, Corfù, Greece, July 2011, pp. 1 –6.

Journal publications:

- D. Anastasia and Y. Andreopoulos, “Linear image processing operations with operational tight packing,” *IEEE Signal Processing Letters*, vol. 17, no. 4, pp. 375 –378, April 2010.
- D. Anastasia and Y. Andreopoulos, “Software designs of image processing tasks with incremental refinement of computation,” *IEEE Transactions on Image Processing*, vol. 19, no. 8, pp. 2099 –2114, August 2010.
- D. Anastasia and Y. Andreopoulos, “Throughput-distortion computation of generic matrix multiplication: Toward a computation channel for digital signal processing systems,” *IEEE Transactions on Signal Processing*, vol. 60, pp. 2024–2037, April 2012.

Invited talks:

- Invited presentation "Software Designs of Image Processing Tasks with Incremental Refinement of Computation," at *UCL Communications and Information Systems Group Seminars*, University College London, London, UK, Dec 11, 2009
- Shortlisted poster "Software Designs Of Image Processing Tasks With Incremental Refinement Of Computation", *University Booth 2010* (co-located with *Design, Automation & Test in Europe*), Dresden, Germany, March 2010
- Invited presentation "Operational Refinement of Image Processing," at *Computer Laboratory Systems Research Group Seminars*, University of Cambridge, Cambridge, UK, April 20, 2010
- Invited presentation "Software Designs for Progressive Multimedia Processing with Graceful Degradation," at *Electronics & Optical Engineering Research Group Seminars*, University of Aberdeen, Aberdeen, UK, Nov 18, 2010

Awards and recognitions:

- *Bob Owens Best Student Paper Award* (1 best paper award out of – approximately – 50 accepted papers) for the paper "*Software Designs of Image Processing Tasks with Incremental Refinement of Computations*" at IEEE Workshop on Signal Processing Systems SiPS 2009: http://www.sips09.org/paperawards.html
- Shortlisted for the *EPSRC UK ICT Pioneers Competition 2011* as one of the top-20 PhD research projects in ICT in the UK for the category "Innovation for Sustainability" with the project "Error tolerant software adaptation of signal processing systems": http://www.epsrc.ac.uk/newsevents/news/2011/Pages/ukictpioneers.aspx; A brief presentation of the presented work is available online: http://bit.ly/hWC7rP

Chapter 2

Packing: a Method for Concurrent Calculation of Image Processing Operators in Software

If we consider the dynamic range that a modern 32-bit or 64-bit floating-point or integer number can attain, we find out that in most practical applications we fail to take advantage of this range because we perform small dynamic range calculations. Thus, it was recognised by prior work that computer hardware with native support for large-bitwidth operations can be used for the concurrent calculation of multiple independent linear image processing operations. This is achieved by packing multiple input samples in one large bitwidth number [6, 11, 100], performing a linear operation in the packed representation and extracting the results.

Thus, packed linear image processing hinges on the idea that the dynamic range of a 32-bit or 64-bit numerical representation can be used for the concurrent calculation of multiple small dynamic range integer operations by stacking multiple operands using the correct *packing factor* z to avoid overflow (or "invading" [100]) of samples outside their interval in the packed representation. Hence, the *packing factor* z is the key aspect of packing theory, since it controls the *packing capability* W, i.e. the maximum number of operands. Ideally we would like to maximise the packing capability in order to perform as many operations simultaneously as possible [6, 100].

The value of z can be calculated following two different methods: using a *loose* or a *tight* approach. Even though they give the same packing capability in most cases, there are cases where an improvement can be obtained with tight packing [6]. We will discuss the maximum theoretical packing capability in Section 2.1, which is one of the theoretical proposals of this thesis. Section 2.2 reviews the alternative approach, which is loose packing. Since most general purpose processors and graphics processing units offer higher speed for the floating-point representation, in comparison to the integer representation of the same bitwidth, Section 2.3 describes how to use the full capability of the double-precision floating point representation to maximise the packing capability. Section 2.5 summarises the exposition and concludes this chapter.

2.1 Tight Packing

In this section we focus on the case of operational[1] packing with real number representations ($0 < z < 1$) and in particular with floating-point since: (a) the parameters for the best-possible packing and unpacking with integer representations are a direct extension of this approach as it will shown in the Section 2.2; (b) unlike integer representations, floating-point representations preserve the sign information for each packed number [7, 100]; (c) programmable processors can offer better native support for floating-point representations in comparison to integer representations thereby enabling higher speed [7] .

Consider a linear operation op that can be applied to W image blocks $\mathbf{B}_p$ concurrently[2] ($p \in \{1, \ldots, W\}, W \geq 1$), using integer operator matrix[3] $\mathbf{K}$:

$$\mathbf{U}_p = (\mathbf{B}_p \, op \, \mathbf{K}). \tag{2.1}$$

[1]The term operational refers to an algorithm or representation realisable by a computer.
[2]The W blocks can be parts of different images that are processed concurrently, or parts of the same image.
[3]Boldface capital letters indicate matrices; the corresponding italicised letters indicate individual matrix elements, e.g. $\mathbf{A}$ and $A[i, j]$; all indices are integers.

For instance, **K** can be the 4×4 H.264/AVC DCT kernel [101], defined as:

$$\mathbf{K} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

and op describes a block decomposition algorithm:

$$\mathbf{U}_p = \mathbf{K}\mathbf{B}_p\mathbf{K}^T$$

where $\mathbf{B}_p$ is an 4×4 matrix taken from the input image.

Operational packing first forms a single block **D** by:

$$\mathbf{D} = \sum_{p=1}^{W} \mathbf{B}_p\, z^{p-1} \tag{2.2}$$

with $z > 0$ an appropriate *packing factor*. Then, the concurrent processing takes place by

$$\mathbf{R} = (\mathbf{D}\, op\, \mathbf{K}). \tag{2.3}$$

Considering the use of an operational real-number representation, such as floating-point, the results can be unpacked sequentially [100]. To do so, all packed results are shifted to the non-negative region of zero by:

$$\mathbf{R}^+ = \mathbf{R} - L_{\min} \cdot \mathbf{J} \tag{2.4}$$

with: $L_{\min} = R_{\min} \sum_{p=1}^{W} z^{p-1}$, $R_{\min}$ being the minimum possible value of the results[4] of (2.1) and **J** the unit matrix (matrix of ones). Each result is subsequently unpacked from $\mathbf{R}^+$ by:

[4]The minimum and maximum possible values of the output ($R_{\min}$ and $R_{\max}$) can be calculated a-priori for given op and **K**, under the known dynamic range of the input.

$$p = 1 : \begin{cases} \mathbf{R}_1^+ & \equiv \mathbf{R}^+ \\ \mathbf{U}_1^+ & = \lfloor \mathbf{R}_1^+ \rfloor \end{cases} \tag{2.5a}$$

$$\forall p \in \{2, \ldots, W\} : \begin{cases} \mathbf{R}_p^+ & = \frac{1}{z}(\mathbf{R}_{p-1}^+ - \mathbf{U}_{p-1}^+) \\ \mathbf{U}_p^+ & = \lfloor \mathbf{R}_p^+ \rfloor \end{cases} \tag{2.5b}$$

where $\mathbf{R}_p^+$ indicates the contents of $\mathbf{R}^+$ during the pth unpacking and $\lfloor a \rfloor$ the largest integer smaller or equal to a. Finally, the results are derived from $\mathbf{U}_1^+, \ldots, \mathbf{U}_W^+$ by offsetting to their original range:

$$\forall p \in \{1, \ldots, W\} : \mathbf{U}_p = U_p^+ + R_{\text{min}}. \tag{2.6}$$

The higher the value of W , the higher the execution time reduction offered by operational packing, since more results are calculated concurrently [7, 100].

If an appropriate factor z is chosen for (2.2), it can be shown [6, 100] that the results $\mathbf{U}_p$ can be extracted correctly (i.e. $\mathbf{U}_p$ of (2.6) is equal to $\mathbf{U}_p$ of (2.1)), if:

1. the processing kernel $\mathbf{K}$ contains integers;
2. op is a linear operation.

The selection of the appropriate packing factor z depends on the dynamic range of the specific algorithm being considered, as elaborated in the following section.

2.1.1 Theory of Operational Tight Packing

Bounds for z and W can be derived from the dynamic range of the output R_{range}, $R_{\text{range}} = R_{\text{max}} - R_{\text{min}}$, that the image processing operation can produce, and from the precision of the operational framework. This range can be calculated if op and $\mathbf{K}$ are known.

The following proposition presents the bound for tight packing based on R_{range} and u_{sys}, with the later being the *relative precision of the computer used for the im-*

plementation[5]. For block decomposition and convolution, a full exposition of the calculation algorithms will be presented in the Section 3.3 and 3.4 respectively.

Proposition 1 (Error-free Packed Processing [6]). *Packing W integers via (2.2) for linear integer-to-integer processing with output range R_{range}, followed by unpacking by (2.4)-(2.6), requires:*

$$z < \frac{1}{R_{\text{range}} + u^*_{\text{safe}}} \tag{2.7}$$

and, under $z < 1$,

$$W \leq \left\lfloor log_z[(R_{\text{range}} + 1)u_{\text{sys}}] + 1 \right\rfloor \tag{2.8}$$

with

$$u^*_{\text{safe}} = \arg \min_{\forall u_{\text{safe}} \in \mathbb{R}^+} \left\{ \left| \left(R_{\text{range}} + u_{\text{safe}}\right)^{W-1} (1 - u_{\text{safe}}) - R_{\text{range}} \right| \right\} \tag{2.9}$$

and u_{sys} the relative precision.

Proof of (2.7)*:* Expanding any element (i,j) of $\mathbf{R}^+$ we have:

$$\begin{aligned} R^+[i,j] = (U_1[i,j] - R_{\min}) + (U_2[i,j] - R_{\min})\, z + \dots \\ + (U_W[i,j] - R_{\min})\, z^{W-1} \end{aligned} \tag{2.10}$$

with $U_p[i,j]$, $p \in \{1, \dots, W\}$, the (i,j)th result for the pth packed block. In order to recover $U_1[i,j]$ correctly via (2.5a):

$$0 \leq \sum_{p=2}^{W} \left(U_p[i,j] - R_{\min}\right) z^{p-1} < 1. \tag{2.11}$$

The upper bound is approached when the linear processing derives $\forall p \in \{2, \dots, W\}$: $U_p[i,j] = R_{\max}$, i.e. the maximum value for each packed result:

$$\sum_{p=2}^{W} z^{p-1} - \frac{1}{R_{\text{range}}} < 0 \Leftrightarrow -R_{\text{range}}\, z^W + (R_{\text{range}} + 1)\, z - 1 < 0 \tag{2.12}$$

[5]Since u_{sys} stems from the finite precision of the implementation, it can be calculated offline by a simple numerical experiment with the target implementation platform [100].

Furthermore, $\forall p \in \{2,\ldots,W\} : U_p[i,j] = R_{\min}$, the lower bound of (2.11) is achieved, regardless of z. Hence, the allowed values of z can be bounded solely based on (2.12) by:

$$z \in \left(0, \frac{1}{R_{\text{range}} + u^*_{\text{safe}}}\right) \tag{2.13}$$

with $u^*_{\text{safe}} > 0$ derived from the solution of (2.12) under the marginal condition of equality to zero. The analytic expression deriving the exact value for u^*_{safe} from (2.12) under this marginal condition can be simplified to:

$$(R_{\text{range}} + u^*_{\text{safe}})^{W-1}(1 - u^*_{\text{safe}}) - R_{\text{range}} = 0. \tag{2.14}$$

Since the last equation has no closed-form solution for u^*_{safe} when $W \geq 4$, we can express u^*_{safe} as the argument minimising the magnitude of (2.14), i.e. (2.9), and use numerical methods (e.g. bisection) to find u^*_{safe}. When unpacking any $U_p[i,j]$, $p \in \{3,\ldots,W\}$, all admissible solutions for z fall within the interval of (2.13) with u^*_{safe} given by (2.9) because $\sum_{p=k}^{W} z^{p-1}$ in (2.12) ($k \geq 2, 0 < z < 1$) is maximised when $k = 2$. As a result, the upper bound for z, which ensures all unpacked samples are mathematically correct, is controlled from the first unpacking. $\square$

Proof of (2.8)*:* Assuming (2.10) under the worst case, i.e. with the maximum value for each element of $\mathbf{R}^+$ ($\forall p \in \{1,\ldots,W\} : U_p[i,j] = R_{\max}$) and relative machine precision u_{sys}, we have[6]:

$$R[i,j] = R_{\text{range}} \sum_{p=1}^{W} z^{p-1} + (R_{\text{range}} + 1)\, u_{\text{sys}} \tag{2.15}$$

In order to recover all $U_p[i,j]$ correctly via (2.5)-(2.6), with $p \in \{1,\ldots,W\}$,

[6]This includes the term $(R_{\text{range}} + 1)u_{\text{sys}}$ to account for the maximum possible numerical error, which is upper bounded by the maximum value of the calculation, $(R_{\text{range}} + 1)$, scaled by the relative precision u_{sys}.

unpacking via (2.5b) imposes:

$$\frac{R_{\text{range}}\left(\sum_{n=p+1}^{W} z^{n-1} + u_{\text{sys}}\right)}{z^{p-1}} < 1. \tag{2.16}$$

For the last unpacking, i.e. $p = W$, we have $\sum_{n=p+1}^{W} z^{n-1} = 0$ and, hence, we reach (2.9) after rounding down to the nearest integer. When any other unpacking $p = M - k$, $k \in \{1, \ldots, W - 1\}$ is considered, we have $\sum_{n=W-k+1}^{W} z^{(n-1)} > 0$ and hence we reach bounds for W that are larger or equal to the one of (2.9). As a result, the tightest upper bound for W is derived by the last unpacking. □

Remark 2.1 (Solution of (2.14) when $W \leq 3$)**.** If $2 \leq W \leq 3$, (2.14) has the following analytic solution ($u_{\text{safe}} \in \mathbb{R}^{+}$):

$$W = 2 : u^{*}_{\text{safe}} = 1 - R_{\text{range}} \Rightarrow z \in (0, 1) \tag{2.17a}$$

$$W = 3 : u^{*}_{\text{safe}} = \frac{1}{2}\left(\sqrt{R^{2}_{\text{range}} + 4R_{\text{range}}} - R_{\text{range}}\right) \tag{2.17b}$$

The solutions of (2.14) for $W = 2$ and $W = 3$ address common cases in the experimental instantiations. We also note that the solution for $W = 1$ is meaningless because (2.10) contains only one element (i.e. no packing), which does not depend on the value of z. ■

Remark 2.2 (Effect of machine precision on (2.7))**.** The upper bound of (2.7) did not consider the machine precision. Unlike (2.8), where the finite precision of the machine (represented by u_{sys}) is the reason that makes W a finite number, the upper bound of (2.7) is imposed by the unpacking process itself and it is valid even under infinite precision. Finite-precision effects will decrease the practical value of z slightly in some cases, as it will be shown experimentally in Subsection 2.3.1. Since this effect is marginal and it depends heavily on the implementation, the next remark proposes a practical, low-complex, framework for setting the operational parameters for tight packing via Proposition 1. ■

Remark 2.3 (Practical usage)**.** The practical calculation of the bounds is done as follows.

Step 0 (Initialisation) Set $L = 1$. Set $W_{\{0\}} \equiv 1, u^*_{\text{safe}\{0\}} \equiv \varnothing, z^*_{\{0\}} \equiv \varnothing$ (the default is no packing capability).

Step 1 (Increment of packing) Set $W_{\{L\}} = L + 1$.

Step 2 (Parameters calculation): Calculate $u^*_{\text{safe}\{L\}}$ from (2.9) and $z_{\{L\}}$ set equal to the bound of (2.7).

Step 3 (Packing bound check) If $W_{\{L\}}$ satisfies (2.8), increment L by 1 and go to Step 1. Otherwise, the tight packing parameters are settled to: $W_{\{L-1\}}$, $u^*_{\text{safe}\{L-1\}}, z_{\{L-1\}}$. ■

Remark 2.4 (Link to prior work)**.** In their work on tight packing, Kadyrov and Petrou [100] propose rules for tight packing, which, under the operational scenario of (2.2)-(2.6), are expressed by [(20), [100]]:

$$z \leq \frac{1}{R_{\text{range}} + 1} \tag{2.18a}$$

$$z^{W-1} > 2g \tag{2.18b}$$

with g the maximum numerical error during the packing and processing of (2.2) and (2.3), and $R_{\min}$, $R_{\max}$ the minimum and maximum possible value of the results of (2.1), respectively. Parameter g is expressed as [100, p. 1883]: $g = \max\{|R_{\max}|, |R_{\min}|\}u_{\text{sys}}$, i.e. the maximum absolute value produced during the processing, multiplied by parameter u_{sys} that represents the relative precision of the computer arithmetic hardware.

The results of Proposition 1 are similar to the previously-proposed rules given by (2.18) [100], but not identical. For example, under non-negative input and kernel values, we have $R_{\min} = 0$, which leads to:

$$W \leq \lfloor \log_z \lceil (R_{\max} + 1) u_{\text{sys}} \rceil + 1 \rfloor$$

under Proposition 1, instead of:

$$W \leq \lfloor \log_z \lceil 2R_{\max}\, u_{\text{sys}} \rceil + 1 \rfloor$$

of (2.18). In general, (2.18) will approximate the bounds of Proposition 1 only under symmetric dynamic range, i.e. when $R_{\max} = -R_{\min}$. Finally, even though it is proposed in [100] to utilise the remaining space beyond the last packing (i.e. beyond $W_{\{L-1\}}$ from Remark 2.3) by reducing the range of the last packing, or by the introduction of error in the results of the last block $\mathbf{U}_W$, this is not applicable when all packed numbers are under the same dynamic range $[R_{\min}, R_{\max}]$, since the same operation is performed in all input blocks, or when only error-free operation is considered. ■

2.2 Loose Packing

A simplified version of the tight packing theory can be used in order to achieve packing with integer representation. This different type of packing involves *precise* division or multiplication by z and, for this reason, if z is defined as a power of two, the resultant values are visibly placed in the packed sample. This packing has the advantage that it consists of bit shift operations and hence packing and unpacking can be performed with a fast procedure under appropriate software or hardware platform-specific support. This simplified version is called *loose packing* [100]. We can define z as:

$$z \equiv 2^{r_{\text{type}} \cdot d} \tag{2.19}$$

with d an integer greater than zero, $r_{\text{type}} = -1$ for 64-bit floating-point representation or $r_{\text{type}} = 1$ for 32-bit unsigned integer representation. The dynamic range of the packing obtained with the maximum packing coefficient cannot be smaller than 2^{-50} for 64-bit floating-point representation[7], and it cannot be larger than 2^{31} for 32-bit unsigned integer representation, which leads to

$$\lceil W + 0.5(r_{\text{type}} - 1) \rceil d \leq w_{\text{type}}, \tag{2.20}$$

[7]For typical floating-point precision, $u_{\text{sys}} = 1.3417592 \times 10^{-16}$ (see details in Section 2.3.1), which leads to $\log_2 u_{\text{sys}} \simeq -50$.

with $w_{\text{type}} = 50$ or $w_{\text{type}} = 31$ respectively[8].

If the range of all outputs of (2.3) is contained in the interval[9] $[R_{\min}, R_{\max}]$, then, following loose packing theory [100], we have:

$$d \geq \lceil \log_2 R_{\max} \rceil + 1. \tag{2.21}$$

Selecting the minimum value of d satisfying the inequality, we reach:

$$W \leq \left\lfloor \frac{w_{\text{type}}}{\lceil \log_2 R_{\max} \rceil + 1} \right\rfloor - 0.5(r_{\text{type}} - 1) \tag{2.22}$$

Loose packing is performed by (2.2) with d selected by (2.19). For floating-point representation ($r_{\text{type}} = -1$), unpacking is performed by [7, 11]:

$$p = 1 : \begin{cases} R_1[i,j] & \equiv R_1[i,j] \\ U_1[i,j] & = \lfloor R_1[i,j] \rceil \end{cases} \tag{2.23a}$$

$$\forall p \in \{2, \ldots, M\} : \begin{cases} R_p[i,j] & = 2^d (R_{p-1}[i,j] - U_{p-1}[i,j]) \\ U_p[i,j] & = \lfloor R_p[i,j] \rceil \end{cases} \tag{2.23b}$$

where: $\mathbf{U}_p$ is the output increment of the result for block p, $\mathbf{R}_p$ is the $\mathbf{R}$ matrix at the pth unpacking and $\lfloor a \rceil$ performs rounding to the nearest integer.

For integer representation ($r_{\text{type}} = 1$), the unpacking is performed by:

$$p = 1 : \begin{cases} R_1[i,j] & \equiv R_1[i,j] \\ U_1[i,j] & = R_1[i,j] \pmod{2^d} \end{cases} \tag{2.24a}$$

$$\forall p \in \{2, \ldots, M\} : \begin{cases} R_p[i,j] & = (R_{p-1}[i,j] \gg d) \\ U_p[i,j] & = R_p[i,j] \pmod{2^d} \end{cases} \tag{2.24b}$$

where $(a \gg d)$ shifts a down by d bits and $a \pmod{2^d} = a - \lfloor a/2^d \rfloor 2^d$ is the modulo operation.

[8]The term $W + 0.5(r_{\text{type}} - 1)$ presents the fact that when $r_{\text{type}} = -1$ (floating point), there is an additional packing at the mantissa which is not included within $w_{\text{type}} = 50$; when $r_{\text{type}} = 1$ (integer representation), the entire packing is achieved within $w_{\text{type}} = 31$.

[9]For simplicity in the exposition and without loss of generality, in the following we assume $|R_{\max}| \geq |R_{\min}|$

In order to ensure there is no numerical error in the calculation when packing with floating-point arithmetic, the magnitude of the maximum possible error [100] must allow for correct rounding by (2.23a) and (2.23b), i.e.:

$$\frac{2^{-w_{\text{type}}} R_{\max}}{2^{-(W-1)d}} < 0.5. \tag{2.25}$$

In our designs, W is initially derived by (2.22) and then decreased (if needed) so that (2.25) holds. Note that no numerical error is possible in the integer packing if the output of each result remains within $[R_{\min}, R_{\max}]$.

2.3 Floating-Point Aspects in Tight Packing

Loose packing will achieve error-free operation under a floating-point representation if (2.25) is satisfied for d and W. However, tight packing, as formalised by Proposition 1, depends explicitly on the machine relative precision u_{sys}. In this section we investigate in more detail the practical implications of the machine precision on tight packing under floating point.

It is well known that the mapping of the floating-point representation (with single or double precision) in the IEEE standard is not linear [102]. Floating–point processor units (FPUs) are designed to have increasingly-finer sampling around zero. Consequently, in an operational environment with an FPU, the experimental value for u^*_{safe}, denoted by $\hat{u}_{\text{safe}}$, may be larger than the theoretical estimate of (2.9) and, hence, the practical value of z may need to be decreased for correct packing and unpacking in floating-point. This is due to the fact that, under the packing of (2.2), the working region of each packed sample (i, j) becomes $(U_1[i, j] - 0.5, U_1[i, j] + 0.5)$, i.e. centers around $U_1[i, j]$ instead of the high-precision region around zero. Naturally, we can translate this region to the high-precision region of $(-0.5, 0.5)$ if we sacrifice the packing performed in the operand (base) of the floating-point number, i.e. if we set $\mathbf{B}_1 \equiv 0$ in (2.2). We perform a related experiment in the next section to demonstrate the practical relevance of the derived bounds and the impact of the precision of FPUs.

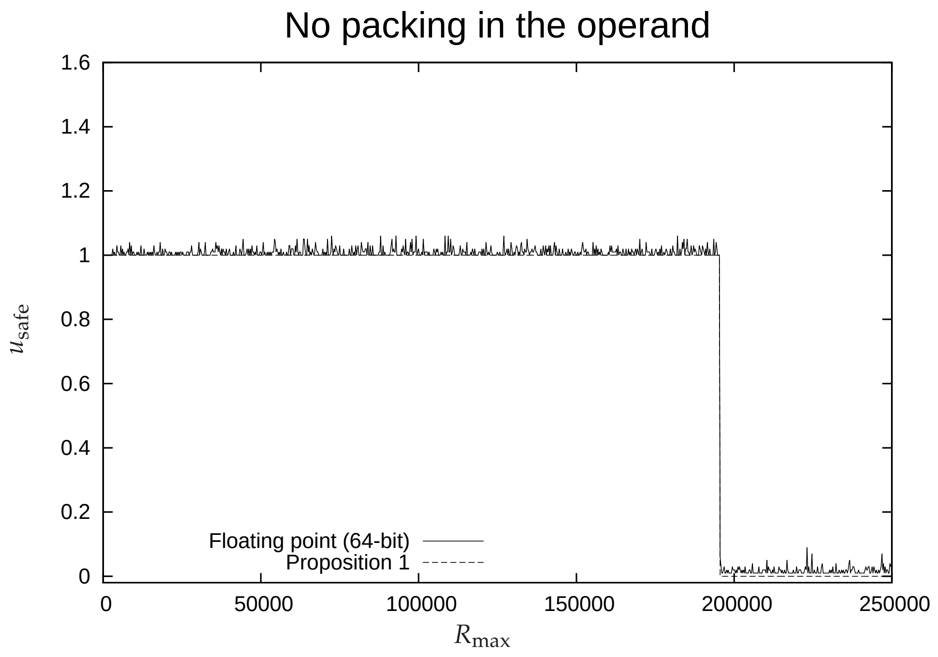
2.3.1 Validation of Proposition 1 in a Floating-Point Unit

We examine the popular case of convolution operations $\mathbf{U}_p = (\mathbf{B}_p \, op \, \mathbf{K})$, $p \in \{1, \ldots, W\}$, with unsigned 8-bit input samples and non-negative convolution[10] kernels $\mathbf{K}$ ($R_{\text{min}} = 0$) deriving increasing values for R_{max}. We used 25 convolution kernels representing various realistic examples such as: 2D Gaussian smoothing filters converted to fixed-point (integer) representation [7], image processing kernels (e.g. kernels simulating camera motion effects or smoothing from Matlab's `fspecial()` function), kernels derived from image templates (for template matching via cross-correlation [7, 100]), etc. Our experiments cover: $R_{\text{max}} = \{10, \ldots, 1000\} \times (2^8 - 1)$. For each kernel, we calculate u^*_{safe} by solving (2.9) with numerical methods (bisection) for each W admissible by (2.8) and selected the corresponding to the maximum admissible W. The calculated relative machine precision in the implementation hardware (double-precision floating point realisation using an Intel Core Duo 2 processor under Microsoft Visual C++ 9.0) was found to be $u_{\text{sys}} = 1.3417592 \times 10^{-16}$ using the test of [(28), [100]]. Per kernel (i.e. per R_{max} value), the experimental value for u^*_{safe} was found by iteratively increasing u^*_{safe}: $\hat{u}_{\text{safe}} = u^*_{\text{safe}} + 0.01i$, $i = 0, 1, \ldots,$ until the packed convolution results can be unpacked without error under the operational tight packing framework of (2.2)-(2.6) in two different cases: (a) no sample packed in the FPU operand (base), i.e. $\mathbf{B}_1 \equiv 0$ in (2.2); (b) one sample packed in the FPU operand, which is the commonly-utilised setting [7, 100].

Figure 2.1 presents the experimental points ($\hat{u}_{\text{safe}}$) vs. theoretical prediction (u^*_{safe}) for case (a) and (b). The floating-point precision causes slight deviations of $\hat{u}_{\text{safe}}$ from the theoretically-predicted value, which, as expected, are slightly higher in the case of one packing in the operand. Importantly, the solution of (2.9) predicts the transition point for $\hat{u}_{\text{safe}}$ accurately. Hence, Proposition 1 offers a more precise characterisation of the experimental results than the packing rules of (2.18b), which suggest that $\forall R_{\text{max}} : u^*_{\text{safe}} = 1$.

For the more noisy case of one packing in the operand (which offers higher packing and is used in practice [7, 100]), Figure 2.2 and Figure 2.3 shows the

[10] A full description of the convolution algorithm will be made in the Section 3.4.



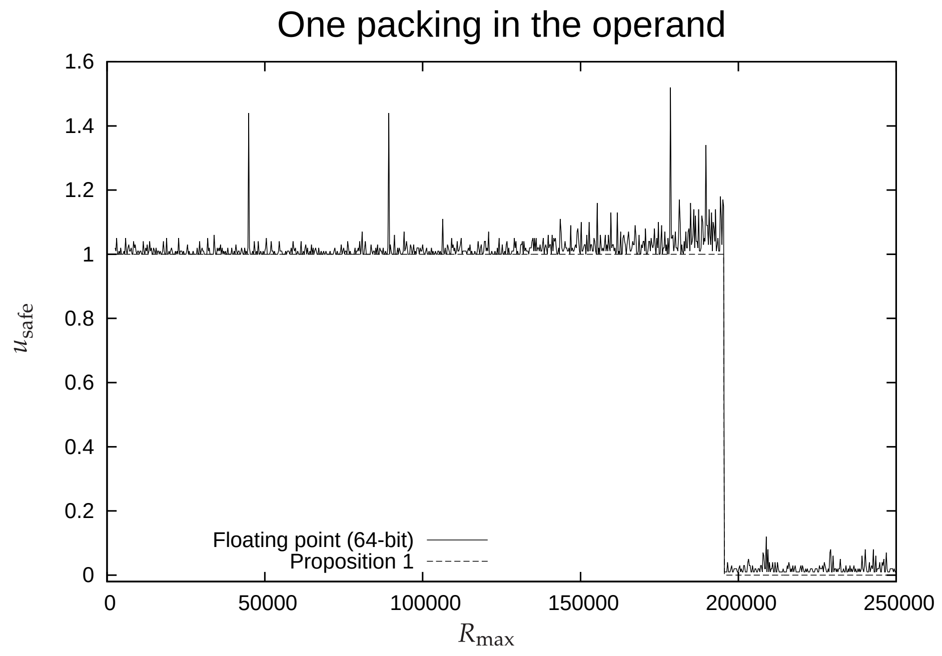


Figure 2.1: Graph of $\hat{u}_{\text{safe}}$ (under double-precision floating-point representation) vs. u^*_{safe} (derived by Proposition 1).

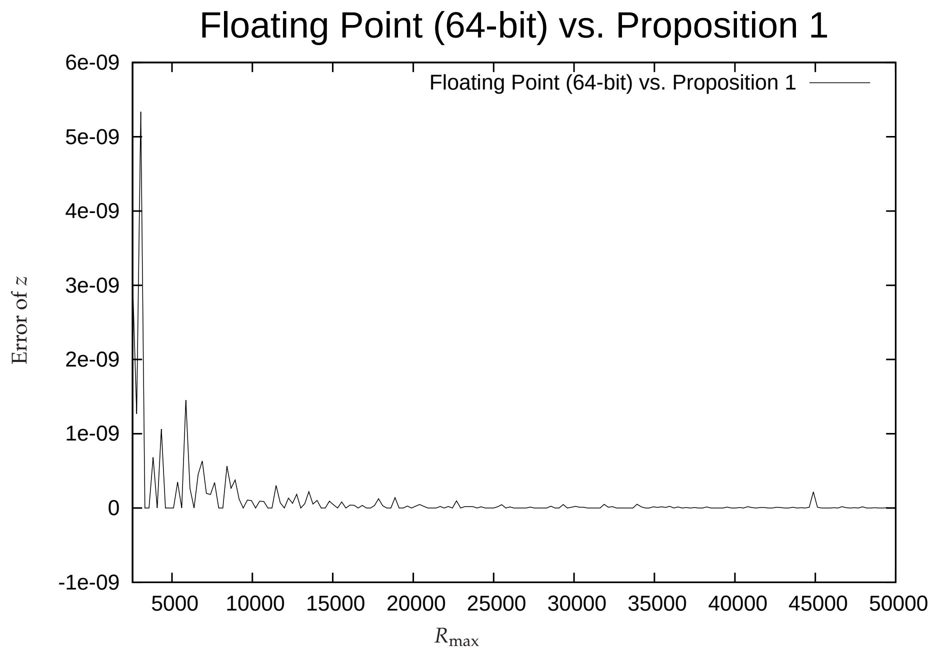
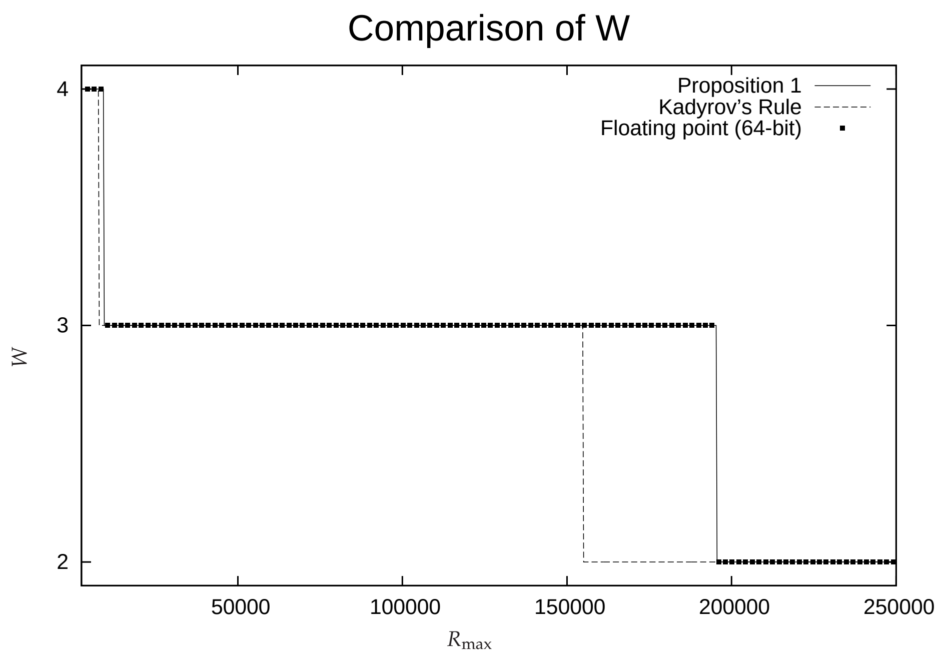


Figure 2.2: Experimentally derived $\hat{z}$ (using double-precision floating-point representation) versus its theoretical bound from Proposition 1.

corresponding experimentally-derived $\hat{z}$ and $\hat{W}$ versus the theoretical bounds of Proposition 1 and the rule of (2.18b) from [100]. For Proposition 1, we measured: $\sum_{\forall R_{\max}} |z - \hat{z}| = 2.15 \times 10^{-8}$, while for the rule of (2.18b): $\sum_{\forall R_{\max}} |z - \hat{z}| = 3.97 \times 10^{-8}$, i.e. both theoretical estimates are very close to the measured values. Finally, the bound of (2.9) predicts the experimentally observed number of packings, $\hat{W}$, for all cases. Notice that the rule of (2.18b) from [100] does not match the transition points from $W = 4$ to $W = 3$ and from $W = 3$ to $W = 2$, since it does not provide the upper bound for the *packing capability* W.

2.3.2 Setting of Operational Parameters for Tight Packing

Following the results of Figure 2.1, 2.2 and 2.3, it is evident that only minor experimental adjustment is needed to derive the operational parameters for tight packing. Under the knowledge of the algorithmic parameters (op and $\mathbf{K}$), we first set $\hat{u}_{\text{safe}} = u^*_{\text{safe}\{L-1\}}$, $\hat{z} = z^*_{\{L-1\}}$ and $\hat{W} = W^*_{\{L-1\}}$ following Re-



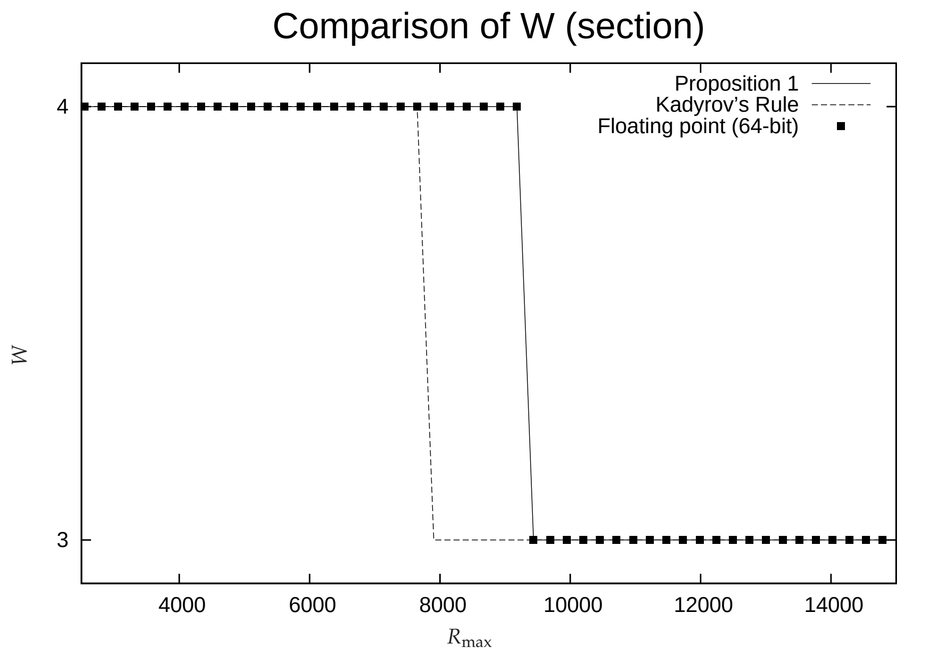


Figure 2.3: Experimentally derived $\hat{W}$ versus its theoretical bound from Proposition 1 and the packing rule of (2.18b). The bottom graph is a zoomed section of the top one.

mark 2.3. We then pack $\hat{W}$ values equal to the maximum and minimum input (e.g. 255 and 0, respectively, for 8-bit images), process them with op and $\mathbf{K}$, and unpack them checking whether the unpacked results are correct (they should all be equal to $R_{\max}$ and $R_{\min}$, respectively). If not, we increase $\hat{u}_{\text{safe}}$ by 0.1, recalculate $\hat{z}$ and $\hat{W}$, and repeat the process until the results are correct. For all our experiments, this derived the tight operational parameters requiring (maximally) two iterations, since the bounds of Proposition 1 provide a very precise characterisation of the operational parameters. This process has negligible complexity and can be performed at an initialisation phase, as described in Remark 2.3. Furthermore, it is completely realisable in software without requiring access to hardware specifications of a particular system.

2.4 Loose Packing vs. Tight Packing

In order to demonstrate the impact of the tight packing framework in applications, we present example results with two convolution kernels, a 12×12 Gaussian kernel converted to fixed-point assuming nine fractional bits, and a 5×9 integer kernel simulating camera-motion effects derived from the Matlab's `fspecial('motion')`. Both kernels derive $R_{\max}$ within $\{15.5, \ldots, 19.0\} \times 10^5 \times (2^8 - 1)$, which allows for $\hat{W} = 3$ in the proposed packing, while tight packing via (2.18b) (denoted as "Kadyrov's Packing") and the operational loose packing environment, denoted as "Loose packing", achieve $\hat{W} = 2$. Testing environment will be described in more details in Section 3.6, where the entire set of experiments performed will be presented (Dell Latitude D630 with Intel Core 2 Duo 2.5 GHz, 2 Gb RAM, Microsoft Windows XP, Microsoft Visual Studio 2008, with all default optimisations of `-O2`). All programs were executed using a convolution kernel to operate using (2.1)-(2.6) and, also, using all 8 input bitplanes directly, in order to operate without the incremental computation features. The input content consisted of the luminance frames of several 704x576 YUV progressive video sequences (at 30 frames-per-second). We report the average processing throughput in frames/second for all cases in Table 2.1. Similar results were obtained for several convolution or cross-

Kernel Type	Conventional	Kadyrov's Packing and Loose Packing	Tight Packing
5×9 Camera motion blurring	11.7	47.5	58.8
12×12 Gaussian fixed-point	9.2	30.3	36.9

Table 2.1: Throughput expressed in frames/second for two convolution examples. The tight packing increases the operational packing from $\hat{W} = 2$ to $\hat{W} = 3$.

correlation experiments where an increase of the packing capability was obtained, i.e. in the regions of Figure 2.3 where the proposed method offers increased $\hat{W}$ in comparison to operational tight packing of [100].

2.5 Remarks on the Usage of Packing for Image Processing Applications

Packing works particularly well when the the input has small dynamic range as indicated by Proposition 1. Detailed experimental results supporting this statement are presented in Section 3.6. This can also be intuitively explained based on the fact that, because the output dynamic range grows due to the performed operations, the number of operands that can be packed successfully is small when the input already has high dynamic range. We can then begin to think of an approach that would split the input samples into "portions" of smaller width and then process these portions through the use of packing and accumulate the unpacked results to obtain the final result. The process of separation of the input source samples into increments is known in the literature as incremental computation [29]. As a result, we can envision the merging of incremental computation with the packing theory for two important reasons: (*a*) increase the packing capability thanks to the smaller input dynamic range; (*b*) decrease the required computation of incremental refinement, which, in its original form [29], requires more operations than the conventional computation. The proposal for the merging of packing with incremental computation is the topic of the next chapter.

Chapter 3

Incremental Computation using Packing and Unpacking

The packing theory provides a powerful approach for small dynamic range integer-to-integer data processing. This feature can be used in a progressive computation environment to keep the execution time comparable with the execution of the same algorithm computed with the conventional approach.

3.1 Introduction

If we modify an algorithm to make it progressive in execution through a series of calculation increments (or steps), the execution time will increase by a certain factor (which is linearly dependent on the number of steps within the progressive execution). This effect can be compensated by using packed processing, so that we can obtain a progressive algorithm with small overhead (or even zero) in execution time in comparison to the conventional (non-progressive) execution. To examine this approach, a novel framework is proposed in this chapter for incremental computation of three different well-known multimedia processing tools: filtering, block transformation and block matching. These three algorithms cover numerous potential applications, such as image transform-based coding [103, 104], de-noising by filtering or by decomposition [105], motion estimation and compensation [20], and image and video analysis algorithms. As a result, these algorithms are the perfect candidates to demonstrate the idea in a operational environment.

This chapter describes the framework, how it works and how the packing theory is merged with the progressive calculation. The notational conventions used for the packing theory are expanded in order to accommodate the exposition of the progressive calculation proposed in this chapter. For each algorithm, data-partitioning, packing, processing and unpacking are analysed both mathematically and operationally.

3.2 Loose Packing in a Incremental Framework

A pictorial depiction of the operational framework for incremental computation is given in Figure 3.1. In this chapter, we discuss this framework in more detail, while the next chapter presents the basic tradeoffs of the proposed packing approach and a full set of experimental results.

The proposed framework is built under the notion of processing of bitplanes n, starting from the most significant bit (MSB) of the input ($n = N - 1$) and going down to the least significant bit (LSB), which is bitplane $n = 0$. For non-negative 8-bit images considered in this work, $N = 8$. Two useful definitions of quantities used in the remainder of this chapter are given below.

Definition 1. For any quantity a used in the computation of an algorithm, a^n_{full}, $0 \leq n < N - 1$, is the computed value of a when the input consists of bitplanes $N - 1$ down to (and including) bitplane n. ■

Definition 2. For any quantity a used in the computation of an algorithm, a^n_{bit}, $0 \leq n < N - 1$, is the computed value of a when only bitplane n of the input is used. ■

The notational conventions of Definition 1 and Definition 2 are extended to

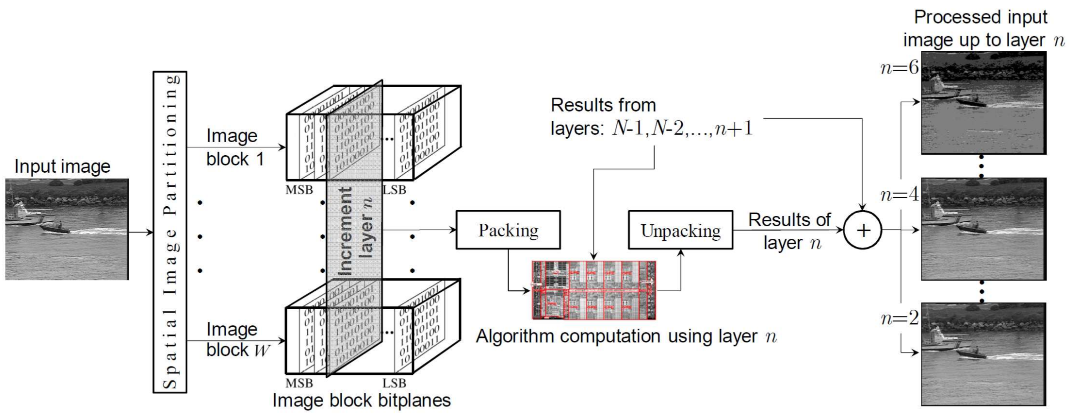

Processed input image up to layer n
n=6
n=4
n=2
Results from layers: N-1,N-2,...,n+1
Input image
Spatial Image Partitioning
Image block 1
Image block W
MSB
LSB
Increment layer n
Packing
Unpacking
Results of layer n
Algorithm computation using layer n
Image block bitplanes

Figure 3.1: Incremental refinement of computation using packing and unpacking of increment layers extracted progressively from the input image data. The output result is progressively refined via the computation of more increment layers. The computation of each layer can also utilise results from previous layers to reduce complexity or focus the computation on regions of interest.

matrices[1], e.g. $\mathbf{A}^n_{\text{bit}}$ is the matrix containing the computed coefficients of $\mathbf{A}$ when only bitplane n of the input image is used.

As shown in Figure 3.1, an input image is initially partitioned into M non-overlapping blocks, whose binary (bitplane-by-bitplane) representation is shown in the middle of the figure, from MSB to the LSB. A total of N increment layers are formed by grouping together the nth bitplane of all blocks ("Increment layer n" in Figure 3.1), $0 \leq n < N-1$. Each increment layer n is also a layer of computation.

First, all W blocks $\mathbf{B}^n_{1,\text{bit}}, \ldots, \mathbf{B}^n_{W,\text{bit}}$ of one layer are stacked together in one block $\mathbf{D}^n_{\text{bit}}$ by:

$$D^n_{\text{bit}}[i,j] = \sum_{p=1}^{W} B^n_{p,\text{bit}}[i,j] 2^{r_{\text{type}}(p-1)d} \tag{3.1}$$

where $B^n_{p,\text{bit}}[i,j]$ is the (i,j)th value of block $\mathbf{B}^n_{p,\text{bit}}$ $(p \in \{1,\ldots,W\})$ that contains parts of increment layer n belonging to the mth spatial block. The last equation is the extension of (2.2) using the "loose packing" described in the Section 2.2

[1]Superscripts in matrices or scalars indicate the bitplane number and, for the case of block matching (Section 3.5) the video frame index (except for superscript T that indicates transposition); the distinction between bitplane and frame is identifiable from the contest.

in order to accommodate the new incremental notation; in particular, it shows that the nth bitplane of the pth block is scaled by $2^{r_{\text{type}}(p-1)d}$ and is then added to the sum of the previous blocks $\{1, \ldots, p-1\}$ of the same increment layer. This leads to a packed increment layer having all W blocks placed on one block $\mathbf{D}^n_{\text{bit}}$ and using integer or floating-point representation[2].

After the packing approach, the desired image processing task op is applied to $\mathbf{D}^n_{\text{bit}}$ for each layer n, $0 \leq n < N$, e.g. convolution with kernel $\mathbf{K}$ is performed by (extension of (2.3)):

$$\mathbf{R}^n_{\text{bit}} = (\mathbf{D}^n_{\text{bit}}\, op\, \mathbf{K}) \tag{3.2}$$

Unpacking is performed by (2.23) with $\mathbf{R}_p$ and $\mathbf{U}_p$ replaced by $\mathbf{R}^n_{p,\text{bit}}$ and $\mathbf{U}^n_{p,\text{bit}}$. For floating-point representation ($r_{\text{type}} = -1$), unpacking is performed by [7] (extension of (2.23)):

$$p = 1 : \begin{cases} R^n_1[i,j] & \equiv R^n_1[i,j] \\ U^n_1[i,j] & = \lfloor R^n_1[i,j] \rceil \end{cases} \tag{3.3a}$$

$$\forall p \in \{2, \ldots, W\} : \begin{cases} R^n_p[i,j] & = 2^d (R^n_{p-1}[i,j] - U^n_{p-1}[i,j]) \\ U^n_p[i,j] & = \lfloor R^n_p[i,j] \rceil \end{cases} \tag{3.3b}$$

where: $\mathbf{U}_p$ is the output increment of the result for block p, $\mathbf{R}_p$ is the $\mathbf{R}$ matrix at the pth unpacking and $\lfloor a \rceil$ performs rounding to the nearest integer.
For integer representation ($r_{\text{type}} = 1$), the unpacking is performed by (extension of (2.24)):

$$p = 1 : \begin{cases} R^n_1[i,j] & \equiv R^n_1[i,j] \\ U^n_1[i,j] & = R^n_1[i,j] \pmod{2^d} \end{cases} \tag{3.4a}$$

$$\forall p \in \{2, \ldots, W\} : \begin{cases} R^n_p[i,j] & = (R^n_{p-1}[i,j] \gg d) \\ U^n_p[i,j] & = R^n_p[i,j] \pmod{2^d} \end{cases} \tag{3.4b}$$

where $(a \gg d)$ shifts a down by d bits and $a \pmod{2^d} = a - \lfloor a/2^d \rfloor 2^d$ is the modulo operation.

[2]The best choice for the utilised representation (integer or floating-point) is system dependent, as it will be shown by our experiments.

After unpacking, the final stage of the proposed computation increments the previously-computed results of increment layers $N-1,\ldots,n+1$ by adding to them the results of the current layer, $\mathbf{U}^n_{1,\text{bit}},\ldots,\mathbf{U}^n_{W,\text{bit}}$:

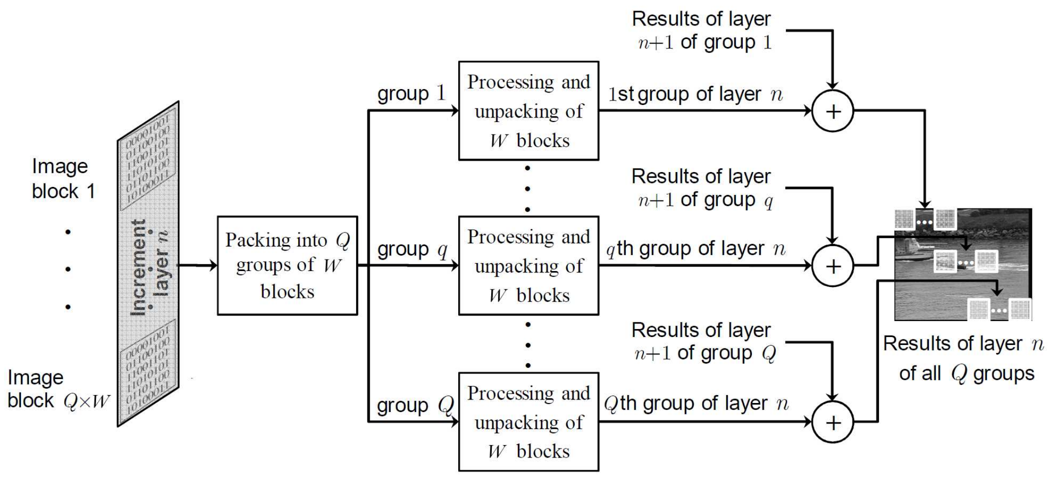
$$\forall p \in \{1,\ldots,M\} : \mathbf{U}^n_{p,\text{full}} = \mathbf{U}^{n+1}_{p,\text{full}} + \mathbf{U}^n_{p,\text{bit}} \tag{3.5}$$

with $\mathbf{U}^{N-1}_{p,\text{full}} \equiv 0$.

Depending on the algorithm of interest, one could localise the calculation of (3.2) around areas of interest based on the previously-computed increment layers (as indicated in Figure 3.1). This will be used in the block matching task and in our experiments with region-of-interest based computation in the following chapter.

Due to the utilisation of the packing technique, the results of all W blocks are computed concurrently by (3.2). Depending on the overhead of packing and unpacking, a comparable number of operations maybe performed in comparison to the direct non-incremental computation of each block.

For notational simplicity, this work discusses bitwise inputs and $N=8$ increment layers for 8-bit images; however, one can combine a number or bitplanes into one layer in order to reduce the increments required to obtain the full-precision result. This is enabled by the implementation of the proposed approach [99] and it is utilised in the Chapter 3.6 in order to make the execution time of the proposed approach comparable to the equivalent non-incremental design of each algorithm of interest. Finally, even though Figure 3.1 indicates that all image blocks are packed together (i.e. the algorithm splits the image into W blocks), in an operational environment the number of blocks that can be packed together may not suffice to cover the entire image. Hence, after packing Q groups of W blocks (where $Q \times W$ gives the total number of image blocks), the processing and unpacking are interleaved. This is shown in the schematic of Figure 3.2. Once the first increment is computed for all groups, the interleaving allows for arbitrary termination of the algorithm even *in-between increment layers*: this is a feature that allows for virtually seamless quality improvement with increased computation within each increment layer.



Results of layer n+1 of group 1
Processing and unpacking of W blocks
group 1
1st group of layer n
Image block 1
Increment layer n
Packing into Q groups of W blocks
Results of layer n+1 of group q
group q
qth group of layer n
Results of layer n+1 of group Q
Results of layer n of all Q groups
Image block Q×W
group Q
Qth group of layer n

Figure 3.2: Packing, processing, unpacking and incrementing the result with Q groups of W blocks.

3.3 Transform Decomposition

The input of this case is Q groups of W blocks of $C \times C$ input pixels, $\mathbf{B}^0_{p,\text{full}}$, $1 \leq p \leq W$, with $C = \{4, 8, 16\}$ for typical cases of block transforms found in the literature [103, 104, 106]. The transform matrix is given by a $C \times C$ integer kernel $\mathbf{T}_{\text{for}}$, e.g. the H.264/AVC 4×4 transform [103]. Transform kernels with non-integer coefficients can be approximated by a fixed-point (FXP) representation with the appropriate number of fractional bits [107]. Hence, they can be computed with an integer kernel followed by inverse scaling after the termination of the calculation and can be accommodated by our framework. The following describe the proposed incremental computation for all W blocks of each group of blocks q, $1 \leq q \leq Q$.

Under an integer transform kernel, the decomposition of the pth block is performed by:

$$\forall p \in \{1, \ldots, W\} : \mathbf{U}^0_{p,\text{full}} = \mathbf{T}_{\text{for}} \mathbf{B}^0_{p,\text{full}} \mathbf{T}^T_{\text{for}}. \tag{3.6}$$

When bitplanes of the input are used for the transform decomposition, the process can be performed for each bitplane n (from $n = N - 1$ down to $n = 0$)

of the pth block by:

$$\forall p \in \{1,\ldots,W\} : \mathbf{U}^n_{p,\text{bit}} = \mathbf{T}_{\text{for}}\mathbf{B}^n_{p,\text{bit}}\mathbf{T}^T_{\text{for}}. \tag{3.7}$$

and the results are added to the previously-computed results by (3.5).

The above process was already proposed within transform-specific formulations for incremental computation of the discrete Fourier transform [29] and the lifting-based discrete wavelet transform [30]. Here, we consider packing the results in order to accelerate the incremental computation in software. We form $\mathbf{D}^n_{\text{bit}}$ by (3.1) and it is used to compute the packed result of all W blocks by:

$$\mathbf{R}^n_{\text{bit}} = \mathbf{T}_{\text{for}}\mathbf{D}^n_{\text{bit}}\mathbf{T}^T_{\text{for}}. \tag{3.8}$$

The results are unpacked from $\mathbf{R}^n_{\text{bit}}$ using (3.3a) and (3.3b) [or (3.4a) and (3.4b) if integer packing is performed] and the final results per bitplane n are derived by (3.5). Notice that only one transform decomposition with block $\mathbf{D}^n_{\text{bit}}$ is performed by (3.8) instead of W block decompositions performed by (3.7). This is expected to save operations by combining blocks together via the incremental packing approach. As shown by (8) and (9), the total number of blocks combined (packing capability), W, depends on the worst-case dynamic range R_{range} of (3.7). This range can be found by assuming the worst-case block:

$$B_s[i,j] = \begin{cases} 1, & \text{if } (-1)^s T_{\text{for}}[i,j] > 0 \\ 0, & \text{if } (-1)^s T_{\text{for}}[i,j] \leq 0 \end{cases} \quad \text{for } 0 \leq i,j < C, s \in \{0,1\}. \tag{3.9}$$

$$R_{\max} = (2^g - 1) \max_{\forall s}\left\{ \max_{\forall j}\left\{\sum_{i=0}^{C-1} T_{\text{for}}[i,j]B_s[i,j]\right\} \cdot \max_{\forall i}\left\{\sum_{j=0}^{C-1} T_{\text{for}}[i,j]B_s[i,j]\right\}\right\} \tag{3.10}$$

where g is the number of bitplanes packed in each increment layer n, e.g. $g = 1$ when we pack a single biplane in each increment layer.

Concerning the transform reconstruction (synthesis) process, the same approach can be followed, where the synthesis is given by

$$\forall p \in \{1, \ldots, W\} : \mathbf{B}^0_{p,\text{full}} = \mathbf{T}_{\text{inv}} \mathbf{R}^0_{p,\text{full}} \mathbf{T}^T_{\text{inv}}, \tag{3.11}$$

with $\mathbf{T}_{\text{inv}} = \mathbf{T}^{-1}_{\text{for}}$. In this case the maximum bitplane of the input is changed depending on the dynamic range expansion of the forward transform. When using packing with integer representation, the incremental approach as presented so far only covers the use of transform kernels with non-negative coefficients because the sign information is not preserved via integer packing. In order to cover the general case of arbitrary transform kernels, we need to convert all transform coefficients to non-negative numbers by:

$$\mathbf{T}_{\text{for}+} = \mathbf{T}_{\text{for}} + \mathbf{P}, \tag{3.12}$$

with $P = -\min_{\forall i,j} \{T_{\text{for}}[i, j]\} \cdot \mathbf{J}$ and $\mathbf{J}$ a matrix of ones. After the incrementally-computed decomposition is performed for each input block $\mathbf{D}^n_{\text{bit}}$ using $\mathbf{T}_{\text{for}+}$, we need compensate for the added component of the kernel of (3.12) during the derivation of the final results per bitplane. However, simple linear algebra shows that several multiplications and additions are needed in order to derive the correct result since the decomposition with the transform kernel of (3.12) derives:

$$\mathbf{R}^n_{\text{bit}+} = \mathbf{T}_{\text{for}} \mathbf{D}^n_{\text{bit}} \mathbf{T}^T_{\text{for}} + \mathbf{P} \mathbf{D}^n_{\text{bit}} \mathbf{T}^T_{\text{for}} + \mathbf{T}_{\text{for}} \mathbf{D}^n_{\text{bit}} \mathbf{P}^T + \mathbf{P} \mathbf{D}^n_{\text{bit}} \mathbf{P}^T \tag{3.13}$$

out of which only the term $\mathbf{T}_{\text{for}} \mathbf{D}^n_{\text{bit}} \mathbf{T}^T_{\text{for}}$ is the desired increment. Hence, we do not investigate this option in this work and restrict our approach to floating-point representation for the transform decomposition case; as mentioned in Chapter 2, packet representation in floating point maintain the sign information and can handle positive and negative operands.

3.4 Two-Dimensional Convolution

For an image consisting of $R_{\text{in}} \times C_{\text{in}}$ pixels, the block partitioning of this case separates the image into W partially overlapping horizontal "stripes", each of which is the considered to be the input block of samples, $\mathbf{B}_p^0$ $(1 \leq p \leq W)$, each having C_{in} columns (as shown in Figure 3.3). The number of rows in each block is controlled by the input image rows and the packing capability (i.e. the value of W).

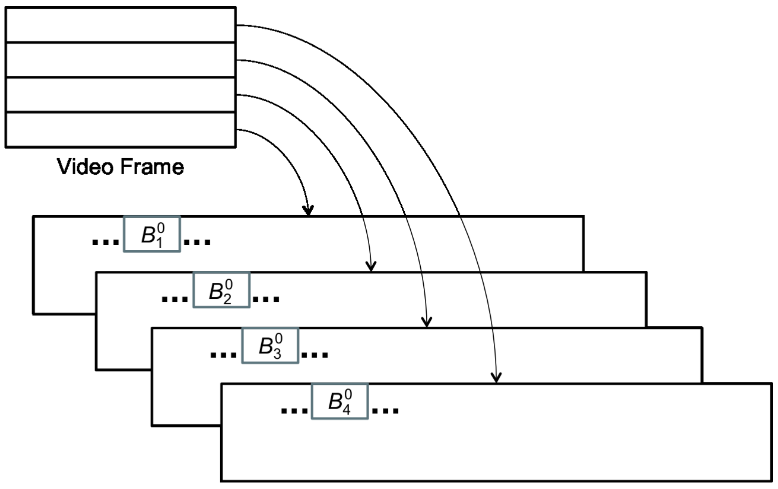


Figure 3.3: Horizontal stripes in the implementation of the packed Two-Dimensional Convolution.

The convolution filter is given by a $V_{\text{kernel}} \times C_{\text{kernel}}$ coefficient kernel $\mathbf{T}_{\text{conv}}$ and convolution of the pth block is performed by:

$$\forall p \in \{1, \ldots, W\} : \mathbf{U}_{p,\text{full}}^0 = \mathbf{B}_{p,\text{full}}^0 * \mathbf{T}_{\text{conv}}. \tag{3.14}$$

In order to produce the correct result with the block-based calculation of (3.14), consecutive blocks share a common subset of rows $V_{\text{overlap}} = \lfloor V_{\text{kernel}}/2 \rfloor$, i.e. the first block ("stripe") is overlapping with the second block vertically by V_{overlap} rows, all subsequent blocks overlap with their previous and next blocks by V_{overlap} rows (above and below the block), and the last block overlaps with its previous block by V_{overlap} rows. When bitplanes of the input are used, the process can be performed for each bitplane n of the pth block by:

$$\forall p \in \{1,\ldots,W\} : \mathbf{U}^n_{p,\mathrm{bit}} = \mathbf{B}^n_{p,\mathrm{bit}} * \mathbf{T}_{\mathrm{conv}}, \tag{3.15}$$

and the results are added to the previously-computed outputs by (3.5).

If we consider packing the results in order to accelerate the incremental computation, then $\mathbf{D}^n_{\mathrm{bit}}$ is formed by (3.1) and it is used to compute the packed result of all W blocks by:

$$\mathbf{R}^n_{\mathrm{bit}} = \mathbf{D}^n_{\mathrm{bit}} * \mathbf{T}_{\mathrm{conv}}. \tag{3.16}$$

The results are unpacked from $\mathbf{R}^n_{\mathrm{bit}}$ using (3.3a) and (3.3b) [or (3.4a) and (3.4b) if integer packing is performed] and the final results per bitplane n are derived by (3.5). Visual examples of Gaussian filtering when $n \in \{6,4,2\}$ are given in Figure 3.1; similar examples with $n \in \{5,2,0\}$ are given in Figure 3.9. As in Section 3.3, the packing capability depends on the worst-case dynamic range, which is calculated using $\mathbf{T}_{\mathrm{conv}}$ in (3.9) and then:

$$R_{\max} = (2^g - 1) \max_{\forall s} \left\{ \left| \sum_{i=0}^{V_{\mathrm{kernel}}-1} \sum_{j=0}^{C_{\mathrm{kernel}}-1} B_s[i,j] T_{\mathrm{conv}}[i,j] \right| \right\}. \tag{3.17}$$

In addition, similarly to the transform decomposition case, kernels with non-integer coefficients can be approximated by an FXP representation. When using packing with integer representation and the convolution kernel contains negative coefficients, we apply (3.12) using $\mathbf{T}_{\mathrm{conv}}$ and then, after unpacking, we increment the result by:

$$\begin{aligned}\forall p \in \{1,\ldots,W\} : \mathbf{U}^n_{p,\mathrm{full}} &= \mathbf{U}^{n+1}_{p,\mathrm{full}} + \mathbf{U}^n_{p,\mathrm{bit}} \\ &+ \min_{\forall i,j} \{T_{\mathrm{conv}}[i,j]\} \sum_{i=0}^{V_{\mathrm{kernel}}-1} \sum_{j=0}^{C_{\mathrm{kernel}}-1} B^n_{p,\mathrm{bit}}[i,j]\end{aligned} \tag{3.18}$$

in order to compensate for the added element $P = -\min_{\forall i,j}\{T_{\mathrm{conv}}[i,j]\} \cdot \mathbf{J}$. Finally, in order to permit incremental computation even within an increment layer, the calculation of (3.16) and the unpacking and incrementation of results are interleaved for each output coefficient $R^n_{\mathrm{bit}}[i,j]$. This permits virtually seamless quality improvement with increased computation within each increment layer.

3.5 Block Matching

The problem of block matching between two successive images $\mathbf{I}_{\text{full}}^{0,t-1}$ and $\mathbf{I}_{\text{full}}^{0,t}$ (of $R_{\text{in}} \times C_{\text{in}}$ pixels) can be abstracted as follows. Given the qth non-overlapping block $\mathbf{B}_{q,\text{full}}^{0,t}$ of $C \times C$ pixels in $\mathbf{I}_{\text{full}}^{0,t}$ ($1 \leq q \leq Q$, assuming Q blocks in total) and a corresponding search area $\mathbf{S}_{q,\text{full}}^{0,t-1}$ of $2S \times 2S$ overlapping blocks in $\mathbf{I}_{\text{full}}^{0,t-1}$, find the $C \times C$ block in the search area that is closest to the qth block. Conventional search algorithms use non-linear distance criteria, such as the sum squared (SS) error or the sum of absolute differences (SAD) error [20]. In this work, we propose an approach to perform incremental block matching using the SS error criterion. However, since the framework of (3.1)-(3.5) works with linear processing, careful handling of the packing, processing and unpacking is required.

The first problem to be addressed is the packing itself. There are several ways one can consider using incremental processing with packing in the block matching case. One could consider two consecutive image blocks in frame and pack increments of these blocks together to compute a single distance criterion for both blocks. On the other hand, it is also possible to pack sample of the same block and the same search area, in order to decrease the number of required operations for each match.

Here we present both approaches. In Subsection 3.5.1 we present the block matching algorithm with a bitwise matching criterion. In this case the packing is performed using different blocks and search areas at the same time. In Subsection 3.5.2 a completely different approach is described that uses the packing *inside* the block itself and uses the SS error as distance metric.

3.5.1 Bitwise Matching Criterion

The first approach is to consider two consecutive image blocks in frame $\mathbf{I}_{\text{full}}^{0,t}$ and pack increments of these blocks together to compute a single distance criterion for both blocks, i.e. following the generic overview of Figure 3.1. The target in this case is to achieve both the maximum packing capability W and, at the same time, be able to check an early termination condition. However, due to the fact that the positions of the best match within the search ranges in frame $\mathbf{I}_{\text{full}}^{0,t-1}$ will be different, this has two important detriments: it makes early termination difficult to apply for block matching[3] and, for increments beyond the first one, it complicates the localisation of the calculation around the previously-established match. However, we shall still pursue this case not only to show a complementary block matching criterion to the one described in the Section 3.5.2, but also to show how different types of packing pattern can be applied to the same algorithm.

In particular, we consider here the popular one-bit motion estimation of Natarajan et al. [108], where a binarisation of the input image is performed prior to block matching and the exclusive-OR (XOR) function is used as a matching criterion.

This block matching method starts with the application of an integer high-pass 2D mask $\mathbf{T}_{\text{high}}$ to the input images:

$$\forall \tau \in \{t-1, t\} : \mathbf{R}_{\text{high,full}}^{0,\tau} = \mathbf{I}_{\text{full}}^{0,\tau} * \mathbf{T}_{\text{high}} \tag{3.19}$$

[3]Early termination stops the calculation of the distance for a candidate match once the distance value has exceeded the one of the already-found best match [20]. This becomes cumbersome for concurrent processing of two (or more) blocks as they have different matches with different minimum distances.

where we use the 19×19 mask proposed by Erturk [109]:

$$T_{\text{high}}[i,j] = \begin{cases} \frac{1}{16}, \text{if}(i,j) \in \left\{ \begin{array}{c} (0,9),(3,6),(3,12), \\ (6,3),(6,9),(6,15), \\ (9,0),(9,6),(9,12),(9,18), \\ (12,3),(12,9),(12,15), \\ (15,6),(15,12),(18,9) \end{array} \right\} \\ 0, \text{otherwise} \end{cases} \tag{3.20}$$

After filtering, the one-bit representation of the filtered image is formed by thresholding the high-frequency images [108]:

$$\begin{array}{c} 0 \leq i < R_{in} \\ 0 \leq j < C_{in} \end{array} : R^{0,\tau}_{\text{bin,full}}[i,j] = \begin{cases} 1, & \text{if} \quad R^{0,\tau}_{\text{high,full}}[i,j] \geq I^{0,\tau}_{\text{full}} \\ 0, & \text{otherwise} \end{cases} \tag{3.21}$$

Then, block matching is performed between all W non-overlapping blocks of $C \times C$ (binarised) pixels in $\mathbf{R}^{0,t}_{\text{bin,full}}$ $(1 \leq p \leq W)$ and their corresponding (binarised) search areas in $\mathbf{R}^{0,t-1}_{\text{bin,full}}$. For each block p at position (i_p, j_p) within image $\mathbf{R}^{0,t}_{\text{bin,full}}$, this derives the optimal location $\{x^{0,*}_{p,\text{full}}, y^{0,*}_{p,\text{full}}\}$ of the matching block within its search area $(-S \leq x, y < S)$ in frame $\mathbf{R}^{0,t-1}_{\text{bin,full}}$ by:

$$\left\{x^{0,*}_{p,\text{full}}, y^{0,*}_{p,\text{full}}\right\} = \arg\min_{\forall x,y} \sum_{i=0}^{R-1} \sum_{j=0}^{C-1} \left\{ R^{0,t}_{\text{bin,full}}[i_p + i, j_p + j] \oplus \right. \\ \left. R^{0,t-1}_{\text{bin,full}}[i_p + i + x, j_p + j + y] \right\} \tag{3.22}$$

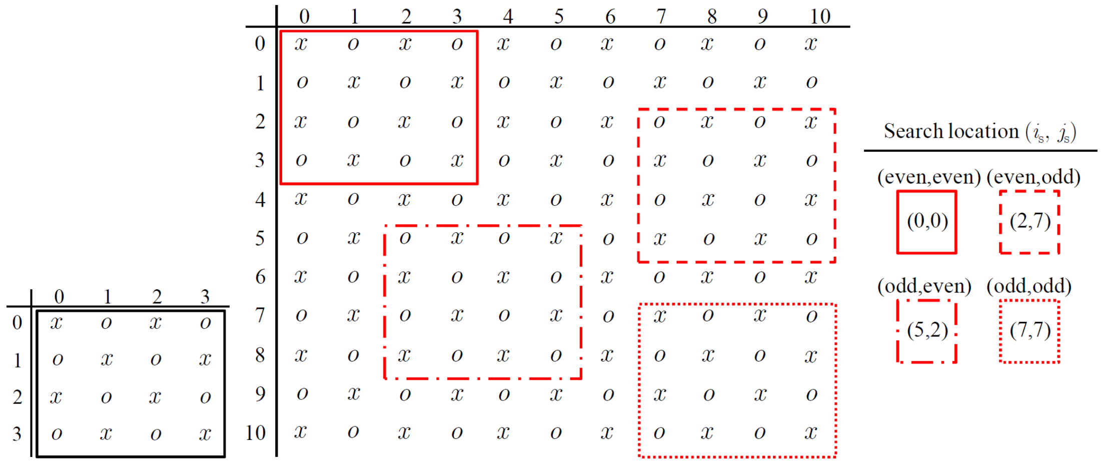
The distance function of (3.22) is simply the summation of the result of the XOR operation between the current block and each block of the search area, which is the Hamming weight. This can be parallelised by packing 32 values of $\mathbf{R}^{0,t}_{\text{bin,full}}$ and $\mathbf{R}^{0,t-1}_{\text{bin,full}}$ in two unsigned 32-bit operands and using a specific processor instruction or a few low-cost operations for the calculation of their Hamming weight [20].

When all bitplanes $n \in \{N-1, N-2, \ldots, 0\}$ of the input images $\mathbf{I}^{n,\tau}_{\text{full}}$ are processed independently, the first part can be performed by the incremental convolution approach of the previous section. Once the results $\mathbf{R}^{n,\tau}_{\text{high,full}}$ of the convolution have been produced, the binarisation of (3.21) is applied in order to produce $\mathbf{R}^{n,\tau}_{\text{bin,full}}$. Then, for every n, the best match per block is found by (3.22) using $\mathbf{R}^{n,\tau}_{\text{bin,full}}$ and 32-bit integer packing.

However, the above technique is expected to increase the execution time for the incremental block matching process, as each increment layer applies the full search algorithm. In order to accelerate this case, we utilise the knowledge of the best match found for each block during the previous increment layers $N-1, ..., n+1$. This is performed as follows. For the first increment layer $n = N-1$, we perform a fast search using logarithmic-step search [20]. Subsequently, for each block p we only search in the neighbourhood of the previously-found best match for this block. The localised search pattern is a spiral search with a fixed distance limit of S_{spiral} pixels horizontally and vertically (see [99] for more details). The use of log-search and the localisation of the search around the previously-found best match will produce approximate results per increment layer.

3.5.2 Sum of Squared Error Criterion

Another way to consider the incremental processing for the block matching is to pack two sets of samples of a *single* block together in order to compute the distance criterion on both sets concurrently. The search area can also be packed in the same way in order to allow for comparisons between packed incremental representations. This is detailed in the following subsection. Subsection 3.5.2.2 explains how the (non-linear) SS error criterion can be calculated using the packed representations. The overall block matching algorithm is summarised in Subsection 3.5.2.4.



(a) Example of quincunx lattice for a 4×4 block.

(b) Example of quincunx lattice of an 8×8 search area with indicative search positions highlighted. Any sub block of the search area within the coordinates $\{(0,0), \ldots, (7,7)\}$ can be selected as a match.

Figure 3.4: Quincunx lattice

3.5.2.1 Packing of Incremental Block and Search-Area Samples using the Quincunx Lattice

We split the block and search-area samples into two non-overlapping sets using the quincunx lattice, whose samples $x[i,j]$ and $o[i,j]$ are shown in Figure 3.4 for an example 4×4 block and its corresponding 8×8 search area. For each new increment n of the block and search area, the packing within each block $\mathbf{B}^{0,t}_{q,\text{full}}$ is done by[4] $(0 \le i < C, 0 \le j < C/2)$:

$$\mathcal{B}^{n,t}_{q,\text{full}}[i,j] = x^{n,t}_{q,\text{full}}[i,j] + o^{n,t}_{q,\text{full}}[i,j]2^{-d} \tag{3.23}$$

with $x^{n,t}_{q,\text{full}}[\cdot,\cdot]$, $o^{n,t}_{q,\text{full}}[\cdot,\cdot]$ the samples of $\mathbf{B}^{0,t}_{q,\text{full}}$ up to (and including) the nth bitplane and following the quincunx lattice of Figure 3.4a and d the packing coefficient, whose setting is discussed in Subsection 3.5.2.2.

For the corresponding search area $\mathbf{S}^{0,t-1}_{q,\text{full}}$, we form four packings by $(0 \le i <$

[4]For exposition simplicity we focus on the case of floating-point packing, i.e. $r_{\text{type}} = -1$.

$2S, 0 \leq j < S$):

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,0,0] = x_{q,\text{full}}^{n,t-1}[i,j] + o_{q,\text{full}}^{n,t-1}[i,j]2^{-d} \tag{3.24a}$$

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,1,0] = o_{q,\text{full}}^{n,t-1}[i,j] + x_{q,\text{full}}^{n,t-1}[i,j]2^{-d} \tag{3.24b}$$

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,0,1] = o_{q,\text{full}}^{n,t-1}[i,j] + x_{q,\text{full}}^{n,t-1}[i,j+1]2^{-d} \tag{3.24c}$$

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,1,1] = x_{q,\text{full}}^{n,t-1}[i,j] + o_{q,\text{full}}^{n,t-1}[i,j+1]2^{-d} \tag{3.24d}$$

with $x_{q,\text{full}}^{n,t-1}[\cdot,\cdot]$, $o_{q,\text{full}}^{n,t-1}[\cdot,\cdot]$ the samples of $\mathbf{S}_{q,\text{full}}^{0,t-1}$ up to (and including) the nth bitplane and following the quincunx lattice of Figure 3.4b. Notice that the packing rules of (3.23) and (3.24) correspond to the case of $W = 2$ of (3.1) but, instead of using two blocks, we use the two lattice sample sets. The need for the four separate packings given by (3.24) becomes evident once we examine the samples that will be compared in the packed representation for every possible combination of search positions. In particular, the packings of (3.24a) and (3.24b) are used when the block is compared to blocks located at (even,even) and (odd,even) positions in the search grid, respectively. The packings of (3.24c) and (3.24d) are used when the block is compared to blocks located at (even,odd) and (odd,odd) positions in the search grid, respectively. Examples for all four cases are given in Figure 3.4b. Hence, for each search location $S_{q,\text{full}}^{0,t-1}(i_s, j_s)$ we use $\mathcal{S}_{q,\text{full}}^{n,t-1}[i_s, \lfloor j_s/2 \rfloor, x, y]$ with $x = mod(i_s, 2)$ and $y = mod(j_s, 2)$. To keep the required memory footprint for (3.23) and (3.24) small, the packing and block matching is performed separately for each block and its search area. In this way, even for 16×16 blocks with ± 16 pixels search range, less than 20Kb are required per block match, i.e. an amount of memory that can easily fit in the level-one cache of all modern processors. In the following subsection, we examine the approach we follow to calculate the packed SS error.

3.5.2.2 SS Error Calculation using Packed Representations

The block SS error calculation with packed representations using $W = 2$ is performed as follows. Assume the packed block samples $\mathcal{B}_{q,\text{full}}^{n,t}[i,j]$ and a candidate block in the packed search area $\mathcal{S}^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j, x, y]$, which corresponds to search location (i_s, j_s) in $\mathbf{S}^{0,t-1}_{q,\text{full}}$. We calculate the packed SS error by:

$$\mathcal{D}_{\text{SS}} = \sum_{i=0}^{C-1} \sum_{j=0}^{C/2-1} \left(\mathcal{B}^{n,t}_{q,\text{full}}[i,j] - \mathcal{S}^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j, x, y] \right)^2 \tag{3.25}$$

Per block position (i, j), (3.25) performs the squared difference between the packing of (3.23) and one of (3.24). In the remainder of this subsection, we analyse the case when (3.23) and (3.24a) are used in (3.25), i.e. $x = 0$ and $y = 0$, since all other cases of (x, y) are examined in the same manner.

By replacing the packed representations using (3.23) and (3.24a), and expanding the square we have:

$$\begin{aligned}
\mathcal{D}_{\text{SS}} = \sum_{i=0}^{C-1} \sum_{j=0}^{C/2-1} \Bigg[& \left(x^{n,t}_{q,\text{full}}[i,j] - x^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j] \right) \\
& + \left(o^{n,t}_{q,\text{full}}[i,j] - o^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j] \right) 2^{-d} \Bigg]^2 \\
= \sum_{i=0}^{C-1} \sum_{j=0}^{C/2-1} & \left(x^{n,t}_{q,\text{full}}[i,j] - x^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j] \right)^2 \\
+ \sum_{i=0}^{C-1} \sum_{j=0}^{C/2-1} & \left(o^{n,t}_{q,\text{full}}[i,j] - o^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j] \right)^2 \times 2^{-2d} \\
+2 \sum_{i=0}^{C-1} \sum_{j=0}^{C/2-1} & \left(x^{n,t}_{q,\text{full}}[i,j] - x^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j] \right) \\
& \left(o^{n,t}_{q,\text{full}}[i,j] - o^{n,t-1}_{q,\text{full}}[i_s+i, \lfloor j_s/2 \rfloor + j] \right) \times 2^{-d}
\end{aligned} \tag{3.26}$$

The three terms of (3.26) show that we can unpack the SS error of each sampling grid and discard the unwanted cross-product term (which is scaled by 2^{-d}). This is done following the unpacking process of (3.3a) and (3.3b): $U_1 = \lfloor \mathcal{D}_{SS} + 0.5 \rfloor$, $\mathcal{D}_{\text{SS},1} = (\mathcal{D}_{\text{SS}} - U_1)2^{-d}$, $U_2 = \lfloor \mathcal{D}_{\text{SS},1} + 0.5 \rfloor$, $U_3 = \lfloor (\mathcal{D}_{\text{SS},1} - U_2)2^{-d} + 0.5 \rfloor$. The total SS error of both grids is $\mathcal{D}_{\text{SS,grid}} = U_1 + U_3$ and U_2 is the unpacked cross-product term. Hence, the packed SS error of (3.25) "carries" within it the result of both incremental grids. Notice that, even though we packed two inputs, we need to perform three unpacking because of the unwanted cross-product term.

3.5.2.3 Setting of the Packing Coefficient d

As explained in Section 2.2, d can be set based on the worst-case dynamic range $R_{\max}$ of the computed results within the packed representation. The worst case for (3.26) occurs in the cross-product term, where we have $R_{\max} = C^2 \times \left(2^{(N-n)}\right)^2$. When we have large blocks (e.g. when $C = 16$) or when we reach the least significant bits ($n = 0$) this range may be prohibitively large to permit $W = 3$ correct unpacking. However, during the calculation of (3.25) we check at the end of every odd-numbered row for early termination (i.e. whether the SS error exceeds the previously-found best one). Hence, we can also set a worst-case dynamic range $R_{\max}^{\text{early}}$ which, if exceeded, we enforce early termination because this will most-likely not correspond to a good match. Using (2.22) and (2.25) with $W = 3$, we find that loose packing can accommodate $R_{\max} = 32768$. Based on experiments with numerous real-world video sequences, we set $R_{\max}^{\text{early}} = \frac{R_{\max}}{2}$ as the threshold for early termination.

3.5.2.4 Overall Block Matching Algorithm

The basic algorithm performed for each increment is given in Algorithm 3.1. In particular, each increment layer applies the search algorithm with a row-by-row scan pattern using early termination. The search area grid to be used is selected via `step 5`, which is performed before the loop that calculates the packed SS error. This simplifies the indexing of the software implementation. This algorithm can be readily extended to consider interpolation grids, and multi-frame motion estimation.

Incremental block matching can benefit from the knowledge of the best match found for each block during the previous increment layers $N-1, \dots, n+1$ in order to speed up the execution. This is performed as follows. For the first

Algorithm 3.1 Incremental block matching using sum squared error

```
$R_{\max}^{\text{early}} \leftarrow 2C \times 2^{N^2}$
$d = \lceil \log_2 R_{\max}^{\text{early}} \rceil + 1$ {Set packing coefficient to be used}
$i_{s,q}^{n,*} \leftarrow S - 1$, $j_{s,q}^{n,*} \leftarrow S - 1$ {Coordinates of best match are set to the center of the search area}
{*Incremental block matching using SS error for each input block* $\mathbf{B}_{q,\text{full}}^{n,t}$ *and search area* $\mathbf{S}_{q,\text{full}}^{n,t-1}$}
**for all** $n = N-1, \ldots, 0$ **do** {For each increment $n$}
  Extract $\mathbf{B}_{q,\text{full}}^{n,t}$ and $\mathbf{S}_{q,\text{full}}^{n,t-1}$
  Calculate $\mathcal{B}_{q,\text{full}}^{n,t}$ and $\mathcal{B}_{q,\text{full}}^{n,t-1}$ using (3.23) and (3.24)
  Set $\mathcal{D}_{\text{SS}}^{*} = \infty$ {The minimum distance will go in $\mathcal{D}_{\text{SS}}^{*}$}
  **for all** $i_s = 0, \ldots, 2W-1$ **do** {For each search row $i_s$}
    **for all** $j_s = 0, \ldots, 2W-1$ **do** {For each search column $j_s$}
      $x \leftarrow i_s \pmod 2$
      $y \leftarrow i_j \pmod 2$
      $\mathcal{D}_{\text{SS}} \leftarrow 0$
      **for all** $i = 0, \ldots, C-1$ **do** {For each block row $i$}
        **for all** $j = 0, \ldots, C/2-1$ **do** {For each block column $j$}
          $\mathcal{D}_{\text{SS}} \leftarrow \mathcal{D}_{\text{SS}} + (\mathcal{B}_{q,\text{full}}^{n,t}[i,j] - \mathcal{S}_{q,\text{full}}^{n,t-1}[i_s+i, \lfloor j_s/2 \rfloor + j, x, y])^2$
        **end for**
        **if** $i \pmod 2 = 1$ **then**
          $U_1 \leftarrow \lfloor \mathcal{D}_{SS} + 0.5 \rfloor$
          $\mathcal{D}_{\text{SS},1} \leftarrow (\mathcal{D}_{\text{SS}} - U_1)2^{-d}$
          $U_2 \leftarrow \lfloor \mathcal{D}_{\text{SS},1} + 0.5 \rfloor$
          $U_3 \leftarrow \lfloor (\mathcal{D}_{\text{SS},1} - U_2)2^{-d} + 0.5 \rfloor$
          $\mathcal{D}_{\text{SS,grid}} \leftarrow U_1 + U_3$ {Unpack $\mathcal{D}_{\text{SS}}$ values of both grids and add them}
          **if** $\mathcal{D}_{\text{SS,grid}} > \mathcal{D}_{\text{SS}}^{*}$ **or** $\mathcal{D}_{\text{SS,grid}} > R_{\max}^{\text{early}}$ **then** {Early termination}
            Goto **Early-Termination**
          **end if**
        **end if**
      **end for**
      Early-Termination:
      **if** $\mathcal{D}_{\text{SS,grid}} < \mathcal{D}_{\text{SS}}^{*}$ **then**
        $\mathcal{D}_{\text{SS,grid}} \leftarrow \mathcal{D}_{\text{SS}}^{*}$
        $i_{s,q}^{n,*} \leftarrow i_s$, $j_{s,q}^{n,*} \leftarrow j_s$
      **end if**
    **end for**
  **end for**
  Store coordinates of best match for increment $n$ of block $q$
**end for**
```

increment layer $n = N - 1$, we perform a fast search using the logarithmic-step search followed by a spiral search pattern around the location of the best match [20]. For subsequent increments of each block q, we only search in the neighbourhood of the previously-found best match for this block. This is performed by performing a spiral search within a fixed distance limit of S_{spiral} pixels horizontally and vertically (see [99] for full details on the implementation). The use of log-search and the localisation of the search around the previously-found best match will produce approximate results per increment layer. Comparisons against the conventional (non-incremental) full search algorithm in terms of prediction quality vs. execution time are given in the Section 3.8.2.

3.6 Experimental Results

For the experiments, the xo-laptop of the OLPC foundation (detailed specification can be found in [110]) running its native Linux operating system (denoted as ″low-end‟ profile) and a Dell Latitude D630 mainstream laptop with an Intel Core 2 Duo processor (clocked at 2.5 GHz with 2 Gb RAM) running Microsoft Windows XP (denoted as ″mainstream‟ profile) are used. All programs were written in C++ and compiled with the GCC 4.1.2 compiler in Linux and with the Microsoft Visual Studio 2008 compiler in Windows, with all default optimisations of `-O2` (maximise speed) mode in both cases. To achieve stable execution-time measurements with high precision in both platforms, the Windows `QueryPerformanceCounter()` function and the Linux `gettimeofday()` function are used and all programs run on the highest priority. Only the execution time required for the computation was measured to display the results of this section (and converted to milliseconds based on system-specific timing measurement). All I/O time from/to the disk was excluded, since it produced the same overhead for both the conventional and the incremental approaches.

The common interchange format (CIF) version of the “Coastguard”, “Foreman”, “Mobile”, “Silent” and “Stefan” video sequences were used as input video frames at 30 fps. Every sequence consists of 300 frames. This set of sequences creates a 1500-frame video with diverse content. For the low-end profile, downsampled sequences to quarter-CIF (QCIF) format at 10 fps were used in order to achieve real-time (or near real-time) processing with the xo-laptop.

The SNR or the peak signal-to-noise ratio (PSNR) measurements presented in the results utilise only the Y (luminance) channel. SNR was measured for all transform and convolution experiments using as reference (noise-free) the result when processing up to the LSBs of each frame (full precision, $n = 0$). P-SNR was measured for the block matching experiment by using the prediction error of frame-by-frame motion compensation (using the original frames) with the motion vector of each block produced from the location of the best match found within the search area.

For a $R \times C$ Y-channel of a video frame, SNR is measured by:

$$\mathrm{SNR}(Y_{\mathrm{current}}) = 10\log_{10}\left(\sum_{i=0}^{R-1}\sum_{j=0}^{C-1}\frac{(Y_{\mathrm{ref}}[i,j])^2}{(Y_{\mathrm{ref}}[i,j] - Y_{\mathrm{current}}[i,j])^2}\right) \tag{3.27}$$

where Y_{current} is the current frame under analysis and Y_{ref} is the reference frame. PSNR is measured by:

$$\mathrm{PSNR}(Y_{\mathrm{current}}) = 10\log_{10}\left(\frac{255^2}{\frac{1}{R\cdot C}\sum_{i=0}^{R-1}\sum_{j=0}^{C-1}(Y_{\mathrm{ref}}[i,j] - Y_{\mathrm{current}}[i,j])^2}\right). \tag{3.28}$$

The results are averages over all the tested video frames, unless is specified otherwise in the particular experiment.

3.7 Incremental Transform Decomposition and 2D Convolution Experiments

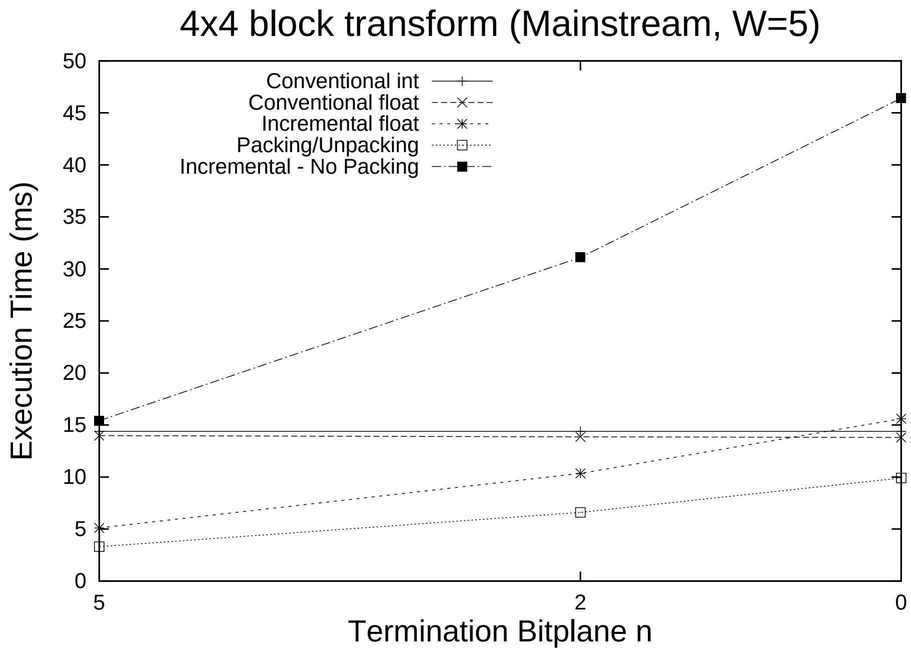
We present results with the 4×4 H.264/AVC block transform [103] and with the fidelity-range extension (FRExt) 8×8 block transform kernel [104] in order to cover two different transform sizes that are used in practice. For the 2D convolution case, we present results with 12×12 and 6×6 Gaussian kernels with their coefficients approximated by fixed-point (FXP) representation with fractional part set to 8 bits and 6 bits, respectively, with the final results rounded to 8-bit integers for display purposes. The selection of the number of bits for the fractional part of the FXP representation ensured that SNR above 58 dB was obtained for all our filtering experiments in comparison to the results obtained with the floating-point representation of the filter kernels. The small kernel is applied on the QCIF content in the low-end profile and the large one on the CIF content in the mainstream profile. We also performed an experiment of block cross-correlation using random image blocks of 8×8 pixels as kernel $\mathbf{T}_{\mathrm{conv}}$ for the two profiles. The results are shown in Figure 3.5-Figure 3.8, where we also report the number of packed blocks W achieved by the incremental approach following (2.22) and (2.25). The corresponding average SNR results are given in Table 3.1. Visual examples of outputs of the 12×12 Gaussian filtering at different precisions are given in Figure 3.9.

3.7.1 Results Exposition

For the results of the incremental approach, instead of inserting each bitplane separately in the incremental computation, we inserted groups of bitplanes together following the pattern $\{3, 3, 2\}$, i.e. the three most significant bitplanes, followed by the 3 intermediate bitplanes, followed by the two least-significant bitplanes. Per video frame, this provides for three quality-driven termination points for the algorithm's execution, which are indicated by the terminating bitplanes of the figures. Conversely, the conventional (non-incremental) approach was executed three times, each time using the source precision indicated by the terminating bitplanes in the figures. Even though the proposed incremental approach can also terminate at arbitrary points *in-between* increment layers, we do not demonstrate this in the results of Figure 3.5-Figure 3.8 since the conventional approach cannot provide for arbitrary termination. Instead, this feature is explored in detail in Section 3.9.

3.7.2 Comparisons Performed

In order to examine the impact of the utilised numerical representation, Figure 3.5-Figure 3.8 show execution time results for both conventional and incremental approaches when using floating-point and integer representation. The only exception are in Figure 3.5 and 3.6 (transform decomposition), where Section 3.3 demonstrated [via (3.13)] that integer representations are impractical when the processing kernel has negative coefficients. In addition, in order to demonstrate the impact of using packed processing, Figure 3.5-Figure 3.8 include the execution time required for packing and unpacking (without processing). This time is included within the reported results for the incremental approaches. We also present the performance of the incremental approach



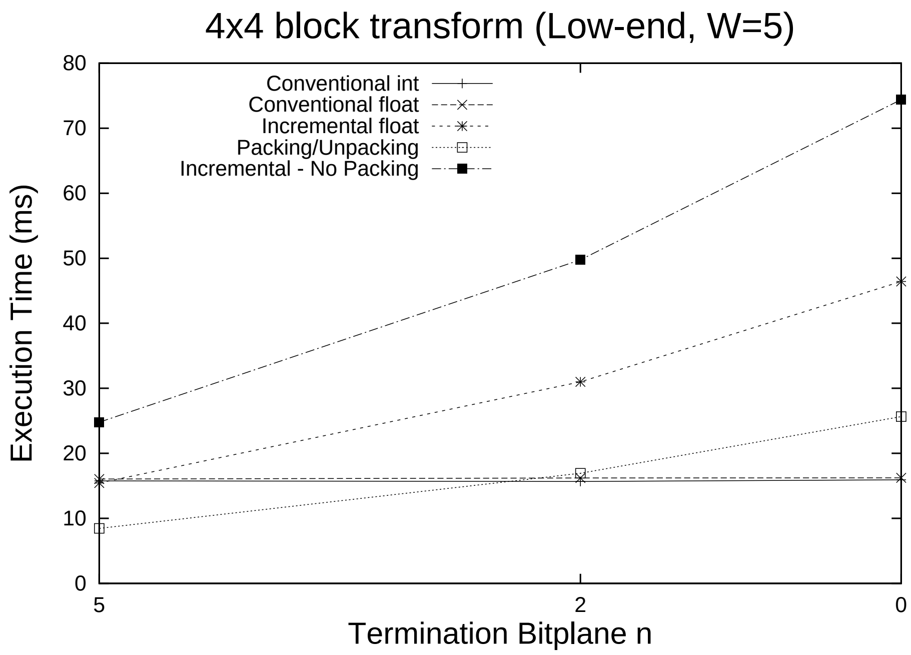
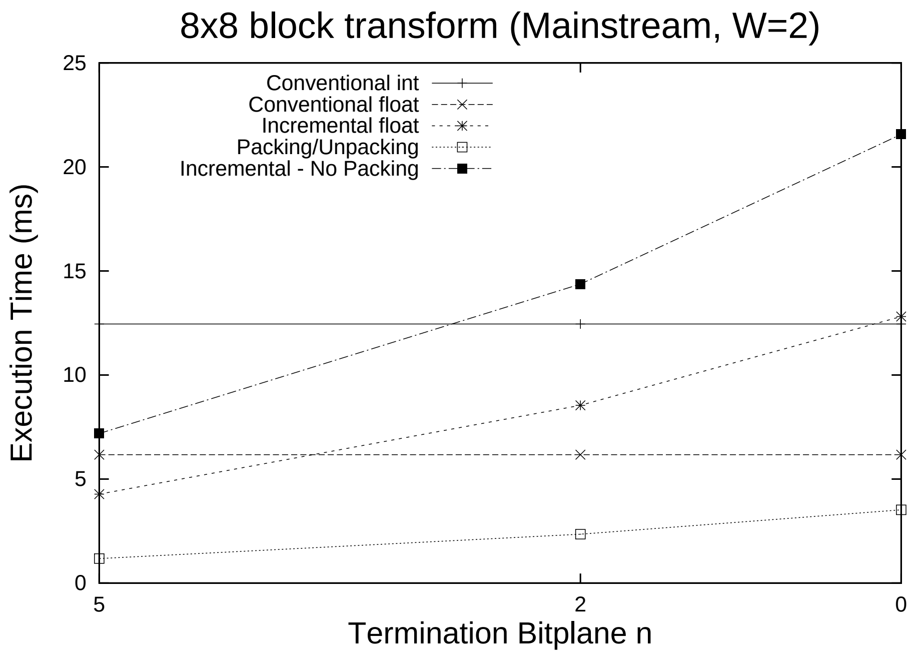


Figure 3.5: Transform decomposition results: 4×4 AVC transform.



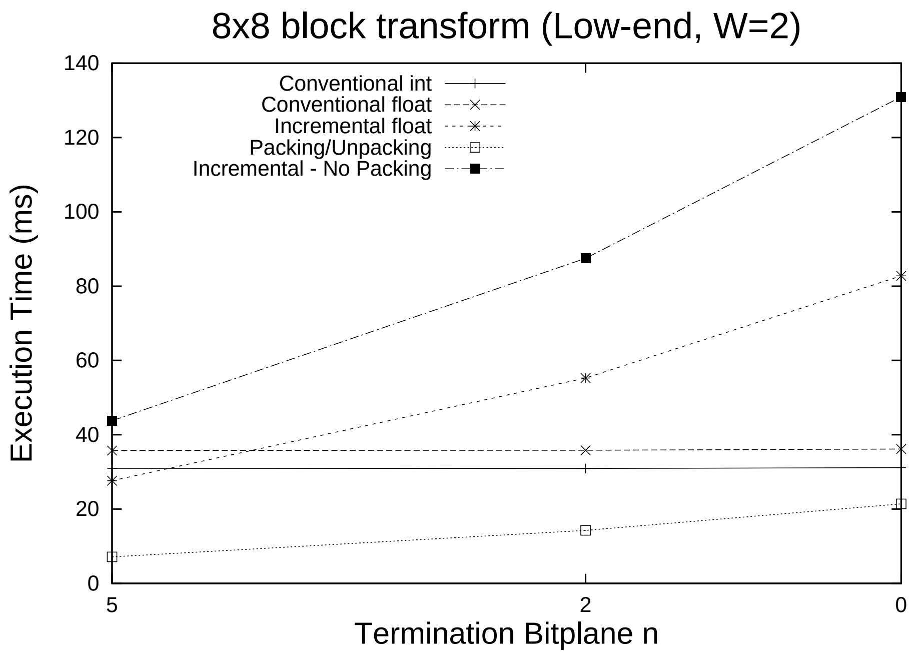


Figure 3.6: Transform decomposition results: 8×8 FRExt kernel.

when packing is not used, i.e. each increment of each block is computed separately.

3.7.3 Analysis of Execution Efficiency

The experiments summarised in Figure 3.5-Figure 3.8 demonstrate that the 32-bit integer representation executes faster than double-precision floating-point in the low-end profile. The mainstream profile exhibits the reverse behaviour. The two profiles analysed lead to the following generic rules for the proposed approach:

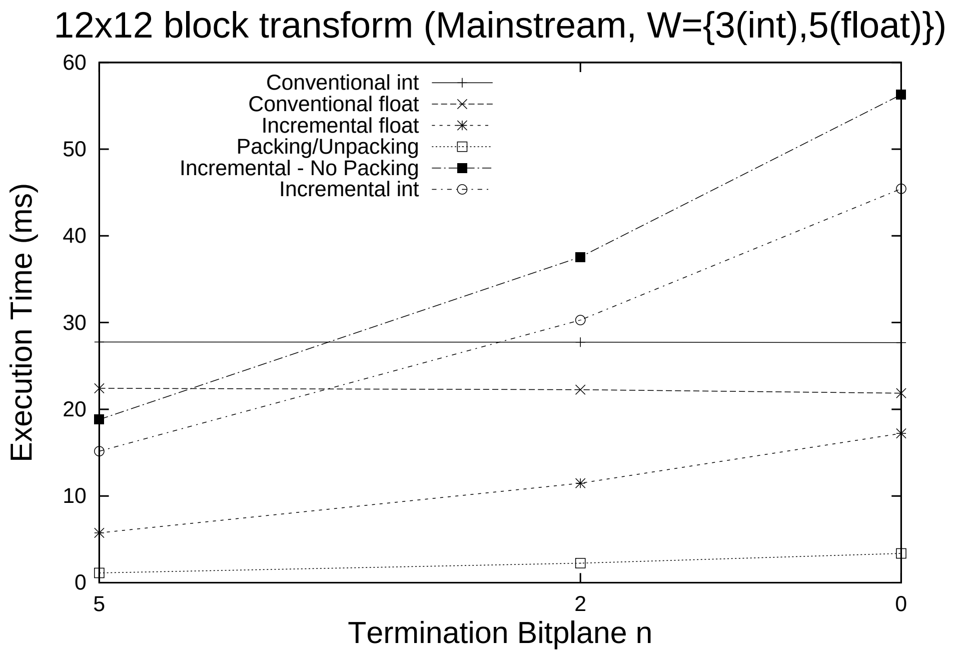
- Representations with larger bitwidth are advantageous for the proposed approach because they increase the packing capability, as shown in the results of Figure 3.7 and 3.8;

- Use of packing is always beneficial for the proposed approach; incremental processing without packing is consistently found to run slower in all experiments;

- When the packing capability W is lower or equal to the number of terminating bitplanes, the proposed approach tends to be inefficient. This is particularly evident in the low-end profile results of Figure 3.6. Conversely, if W is high, the proposed approach becomes very efficient, *unless* it uses a representation that is not fast in the implementation hardware (e.g. low-end profile of Figure 3.7 with floating-point representation).

- When the packing/unpacking cost requires more than 30% of the execution time of the conventional (non-incremental) approach, the proposed approach becomes inefficient [e.g. low-end profile of Figure 3.6]. The exception to this rule is when high packing capability is achieved using a fast numerical representation in the utilised hardware, as seen in the mainstream profile of Figure 3.6.

- The average execution time of the proposed approach is increasing linearly when the source is processed with increased precision (lower ter-



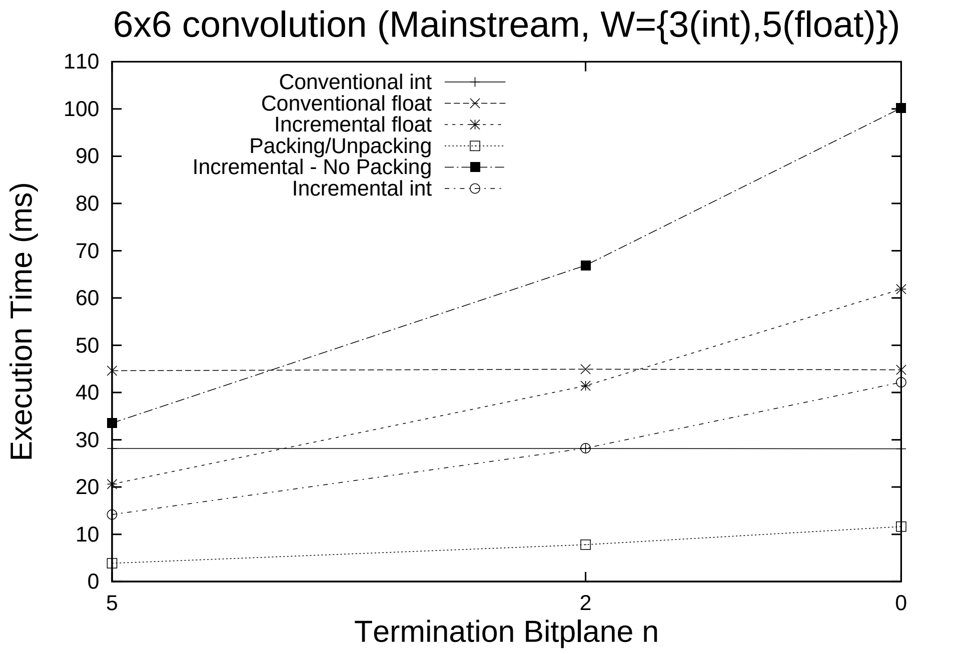
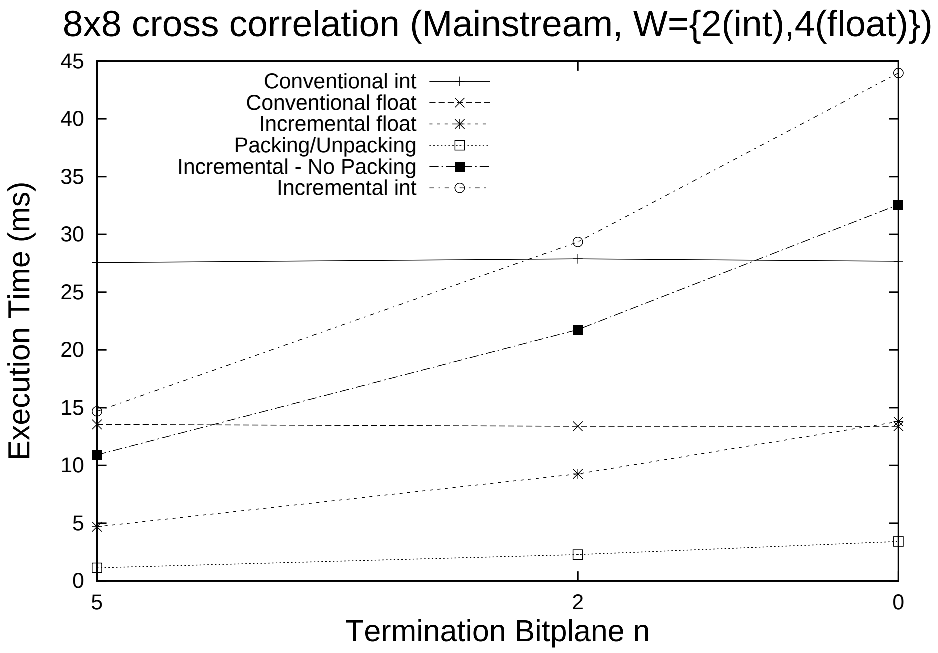


Figure 3.7: 2D convolution results with 12×12 (mainstream profile) and 6×6 (low-end profile) Gaussian kernels approximated with fixed-point representation.



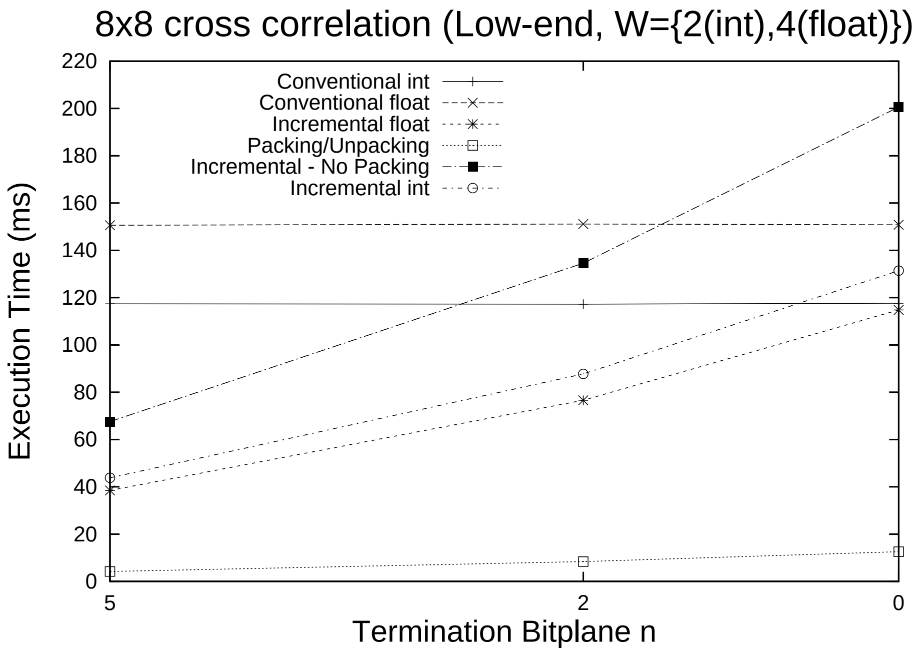


Figure 3.8: Block cross-correlation results.

	Transform Decomposition SNR (dB)		2D Filtering SNR (dB)	
Terminating Bitplane	4x4	16x16	12x12 gaussian	8x8 cross-correlation
$n = 5$	16.74	19.12	18.88	18.88
$n = 2$	35.45	39.05	39.35	39.34

Table 3.1: Average signal-to-noise ratio for the terminating bitplanes of the mainstream-profile experiments of Figure 3.5-Figure 3.8. SNR was infinity for all cases when $n = 0$. Both conventional and incremental algorithms achieved identical SNR for each terminating bitplane.

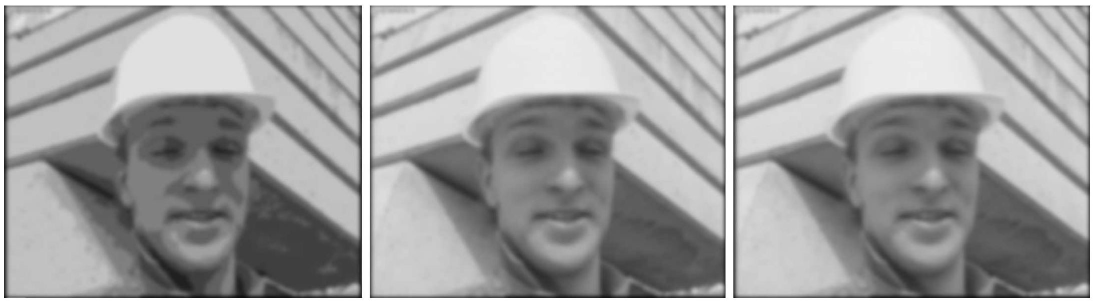


Figure 3.9: Representative output frame for terminating the computation at $n = \{5, 2, 0\}$ bitplanes (shown from left to right) for the 12×12 Gaussian filtering.

minating bitplanes). This contrasts with the conventional approach that requires constant execution time regardless of the input precision. Once two increments have been processed, this feature can be used to establish the average execution time of subsequent increments of the proposed approach.

These five rules encapsulate all our experimental observations. They also form useful guidelines for deciding if and how to deploy the proposed approach: which numerical representation to use, how many terminating bitplanes are possible without significant loss in efficiency, whether the algorithm is not complex enough to outweigh the cost of packing and unpacking, are all factors that affect the deployment of the proposed approach.

3.7.4 Analysis of Visual Quality

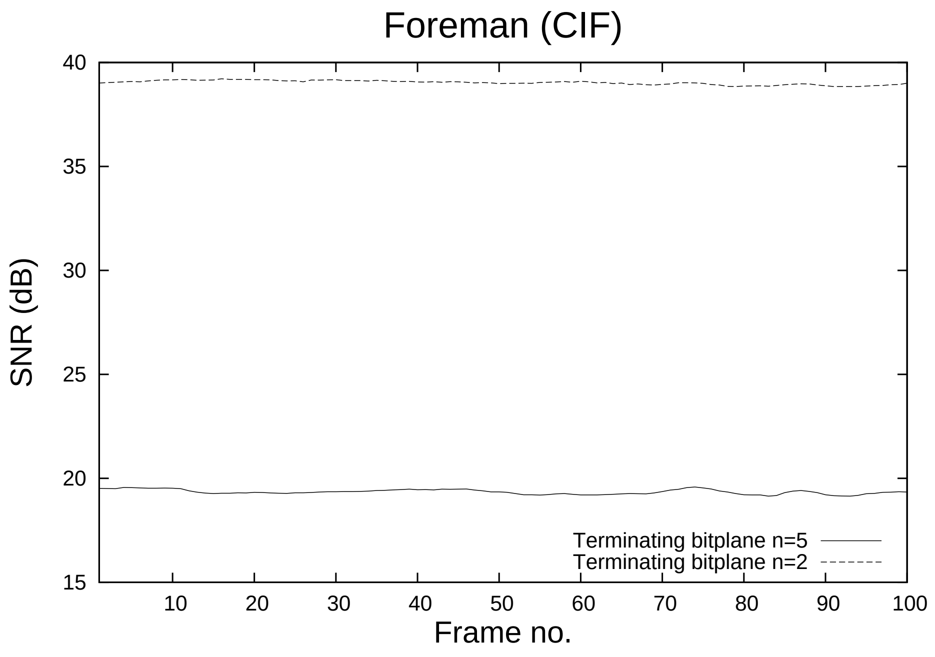
Identical SNR results were obtained for both conventional and incremental algorithms in all cases (Table 3.1). Importantly, SNR per frame is monotonically increased when processing more increments (lower terminating bitplanes). An example is given in Figure 3.10a by inverting the results of the 4×4 transform decomposition back to the image domain and comparing with the original video frames (since the transform is lossless at full precision). Since SNR comparisons may not reflect the visual distortions caused by terminating the processing to higher bitplanes, we have also performed tests with the structural similarity index measure (SSIM) of Wang et al. [111] using the related Matlab source code[5] with the suggested parameter settings. We used the Y-frames of each sequence for this purpose and provide an example in Figure 3.10b for the transform decomposition. Indeed, the comparison between Figure 3.10a and 3.10b shows that even though a significant drop occurs in SNR, the output results are visually meaningful since the mean SSIM (MSSIM) remains around 0.8 [11]. For terminating bitplane $n = 0$, SNR is infinite and the MSSIM is one.

As a final remark, it is important to emphasise that the incremental approach produces *all execution-time vs. distortion measurements via one single execution*. In other words, if, for any frame, the computation is terminated arbitrarily at a given point by a task scheduler, the results based on the already computed bitplanes of that frame are readily available in the program's allocated memory.

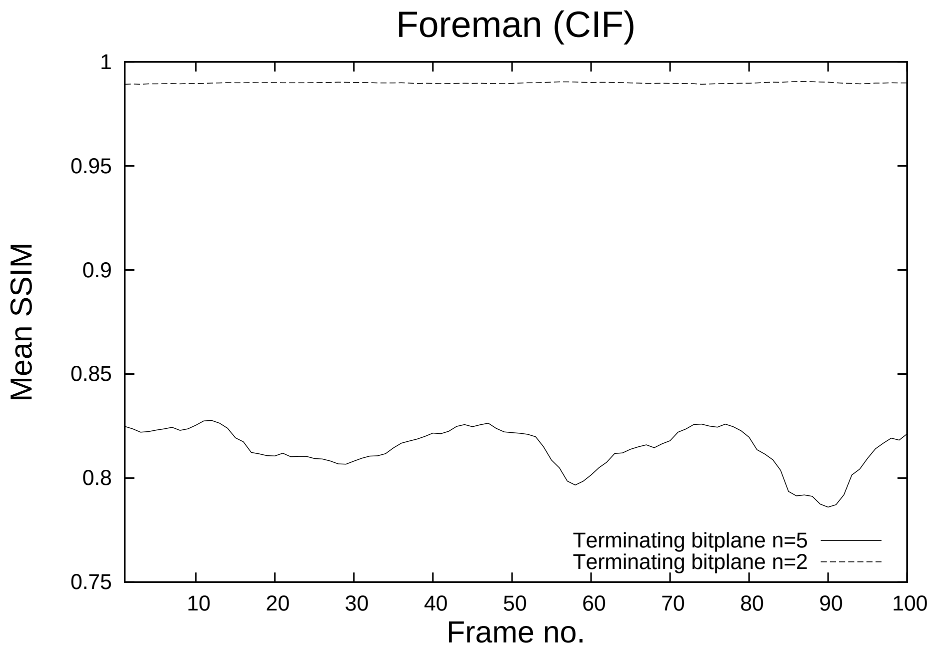
3.8 Incremental Block Matching Experiments

Two set of experiments were made for the two different solutions of the block matching algorithm described in the Section 3.5. We present the case of $C = S = 16$ for both profiles and both implementations.

[5]available online at http://www.cns.nyu.edu/ lcv/ssim/



(a) SNR comparison



(b) MSSIM comparison

Figure 3.10: Frame-by-frame comparison for the reconstruction of the incrementally computed 4×4 transform decomposition of Figure 3.5; Only the first 100 frames are shown.

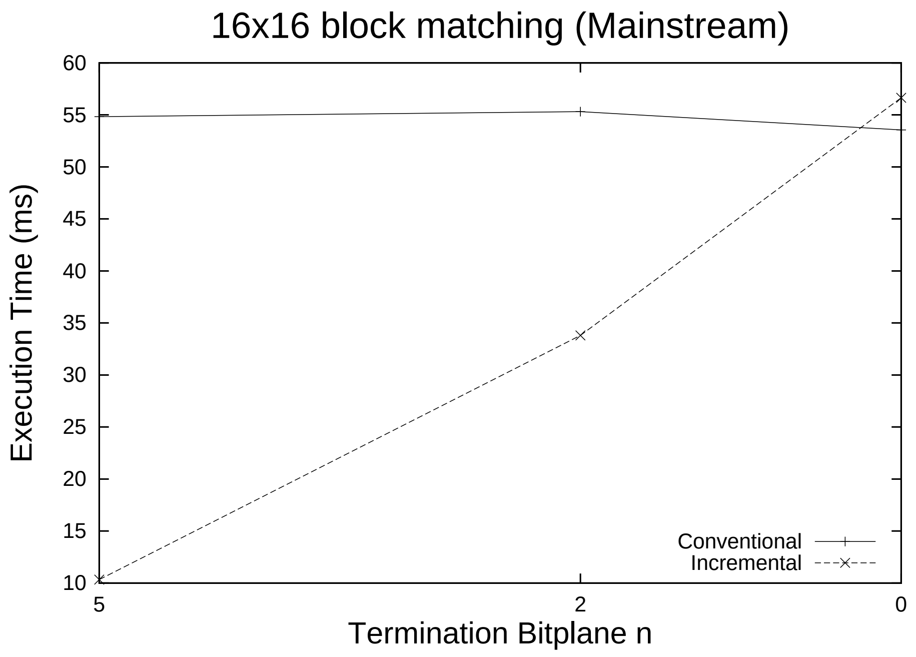
	Mainstream Profile PSNR (dB)		Low-end profile PSNR (dB)	
Bitplane	Incremental	Conventional	Incremental	Conventional
$n = 5$	27.32	28.32	23.57	24.67
$n = 2$	28.48	28.52	24.77	24.83
$n = 0$	28.51	28.57	24.79	24.84

Table 3.2: Average peak-signal-to-noise ratio for the terminating bitplanes of the experiments of Figure 3.11.

3.8.1 Bitwise Matching Criterion

Indicative results for the block matching algorithms are shown in Figure 3.11. The corresponding PSNR results are shown in Table 3.2. Since the convolution kernel $\mathbf{T}_{\text{high}}$ of Erturk [109] only requires 16 additions, it was found experimentally that the incremental algorithm can perform the convolution directly per input set of bitplanes (rather than use the packing approach) without significant overhead. Similar to the previous cases, instead of always inserting individual bitplanes, we inserted the input-image bitplanes following the pattern $\{3, 3, 2\}$ (as indicated by the terminating bitplanes of Figure 3.11).

The PSNR results of Table 3.2 demonstrate that the log-search performed for the first terminating biplane ($n = 5$) provides significantly inferior prediction result for the incremental method as compared to the conventional approach that performs full search (approximately 1 dB loss). However, the prediction quality of the incremental algorithm becomes virtually identical to the conventional approach once more bitplanes are processed and the spiral search refines the motion vector per block. In particular, incremental motion estimation provides only 0.06 dB loss of prediction precision at full precision, $n = 0$, with $S_{\text{spiral}} = 9$ in the mainstream profile and $S_{\text{spiral}} = 4$ in the low-end profile. In addition, since the performance seems to saturate when $n < 2$, the proposed approach can terminate the computation earlier and achieve near real-time performance, something that the conventional approach cannot take advantage of, since its execution time does not scale down with decreased precision.



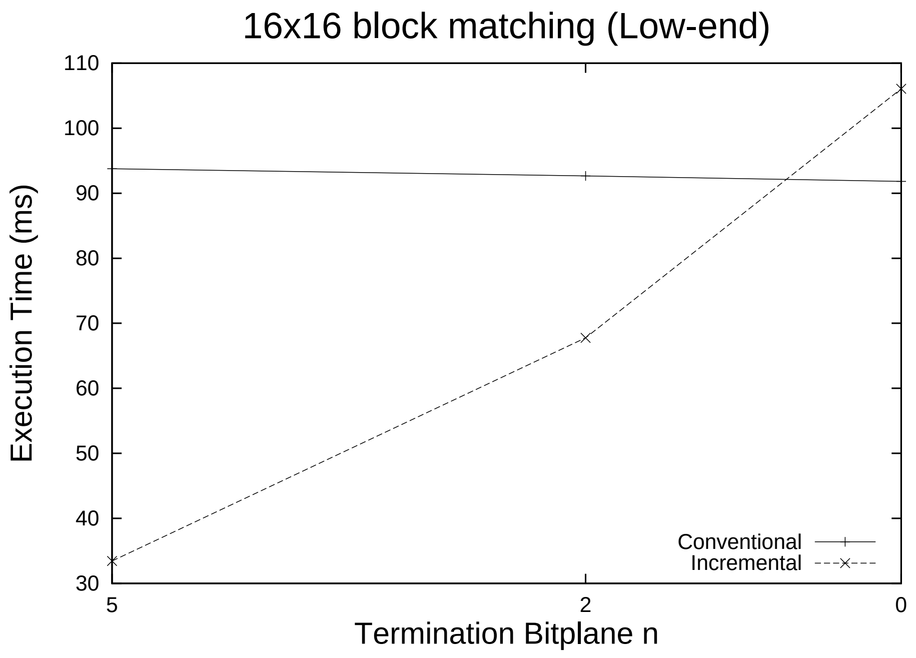


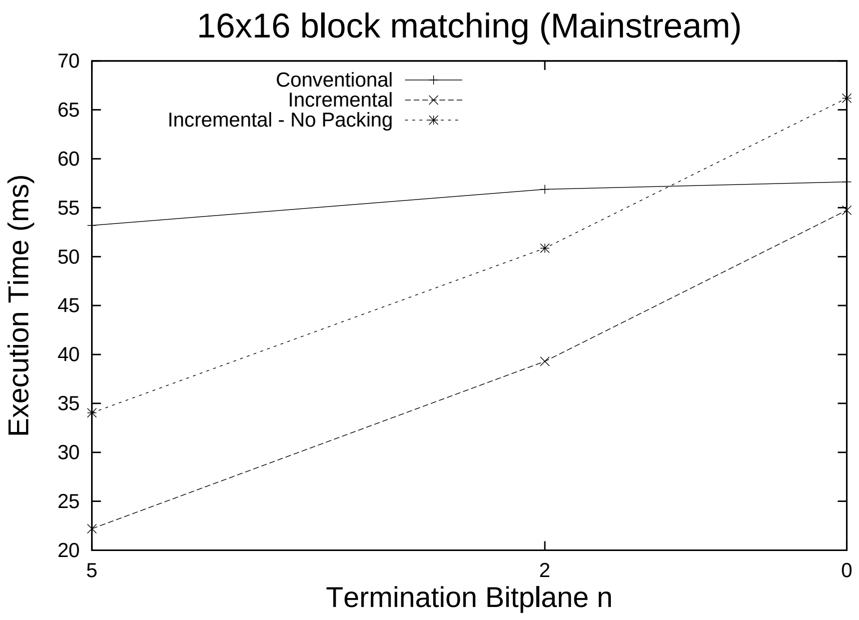
Figure 3.11: Block matching results with bitwise criterion.

3.8.2 Sum of Square Error Criterion

For this second case, the packing structure and the weight function are different. As described in the Section 3.5.2, in this case packing is made *inside* both block and search area and for this reason both refinement of the previously-found best match and early termination can be performed without any additional algorithmic cost.

The average execution times obtained for the block matching algorithms are shown in Figure 3.12. The corresponding PSNR results are shown in Table 3.3. The conventional approach is using SAD-based matching in order to correspond to the common full-search algorithm found in the literature. We also include the proposed incremental block matching scheme without the use of packing for comparison purposes. Similar to the previous cases, instead of always inserting individual bitplanes, we inserted the input-image bitplanes following the pattern $\{3, 3, 2\}$ (as indicated by the terminating bitplanes of Figure 3.12).

The PSNR results of Table 3.3 demonstrate that the log-search performed for the first terminating biplane ($n = 5$) provides significantly inferior prediction result for the incremental method as compared to the conventional approach that performs full search (approximately 0.6 dB loss in performance). However, the prediction quality of the incremental algorithm approaches the conventional approach once more bitplanes are processed and the spiral search refines the best match location found per block. In particular, over the larger range of video content tested (1500 frames from 5 sequences), incremental block matching leads to only 0.2 dB loss of prediction efficiency at full precision ($n = 0$). We used $s_{\text{spiral}} = 9$ in the mainstream profile and $S_{\text{spiral}} = 8$ in the low-end profile. In addition, since the performance seems to saturate when $n < 2$, the proposed approach can terminate the computation earlier and achieve near real-time performance, something that the conventional approach cannot take advantage of, since its execution time does not scale down with decreased precision.



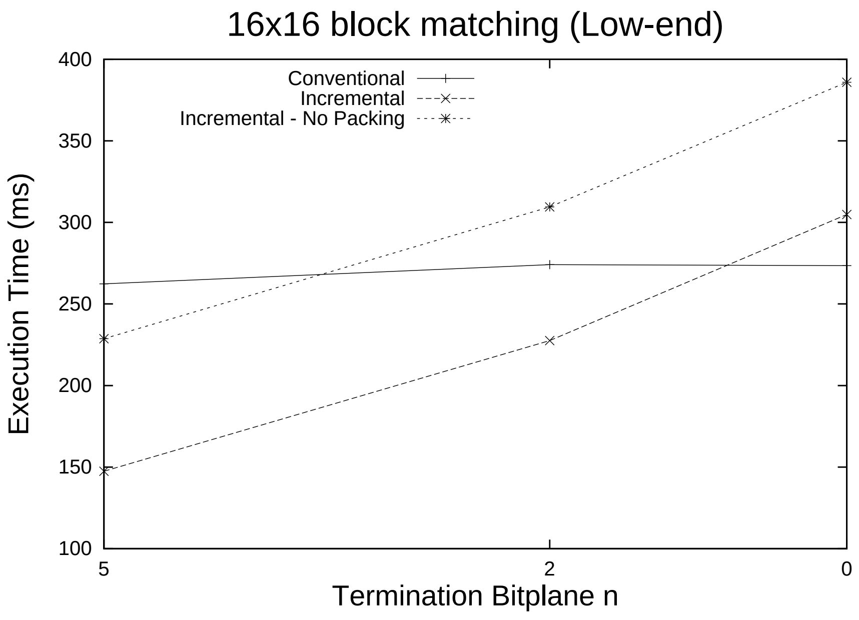


Figure 3.12: Block matching results.

	Mainstream Profile PSNR (dB)		Low-end profile PSNR (dB)	
Bitplane	Incremental	Conventional	Incremental	Conventional
$n = 5$	28.43	29.18	24.38	24.88
$n = 2$	29.10	29.46	24.70	25.01
$n = 0$	29.25	29.46	24.78	25.01

Table 3.3: Average peak-signal-to-noise ratio for the terminating bitplanes of the experiments of Figure 3.12.

3.9 Applications

In this section, we exploit the incremental and scalable nature of the proposed incremental computation in order to show its usefulness in applications. We first present a simple example of how one can use the proposed framework for region-of-interest computation. Subsection 3.9.2 presents results with a real-time scheduling framework, while Subsection 3.9.3 presents indicative results for the energy-distortion trade-offs enabled by the proposed software-based incremental computation of image processing on the ultra low-power xo-laptop. Since the last two application examples use multi-process execution, the reported timing measurements therein include both the computation time as well as all I/O time from/to the disk.

3.9.1 Region-of-Interest Based Incremental Computation

The proposed approach can selectively refine parts of the computation for a given input video depending on a preselected region-of-interest (ROI) mask. To demonstrate this, we selected the “Silent” sequence that involves a sign-language presenter at a static location in each frame and defined the arbitrary ROI mask shown in Figure 3.13 that focuses on the presenters face and hands region. The 4×4 AVC block transform decomposition was used as an indicative processing algorithm (running on the mainstream profile). The decomposition occurred progressively for each terminating bitplane $n \in \{5, 2, 0\}$ within the ROI. However, the decomposition outside of the ROI terminated at $n = 5$

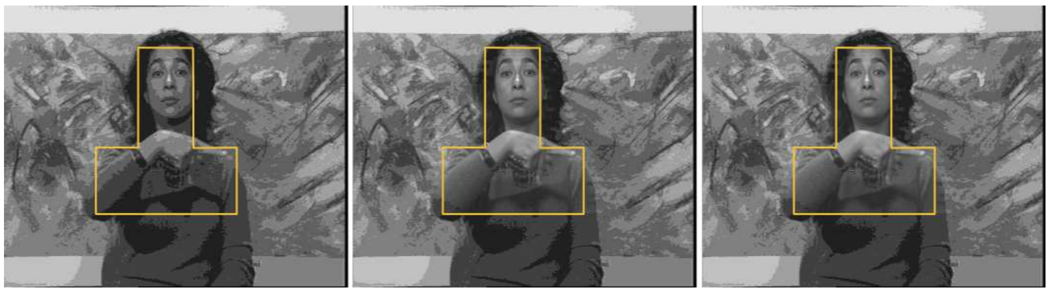


Figure 3.13: ROI-based incremental transform decomposition. An example frame with terminating bitplanes $5, 2, 0$ is shown (from left to right).

(first increment only). The average execution times per frame were: 5 ms for $n = 5$, 6ms for $n = 2$ and 7 ms for $n = 0$. Incremental computation without the ROI required 5 ms, 10 ms and 16 ms, respectively. Conventional (non incremental) computation required 15 ms for all cases since the execution time does not scale down with decreased precision. Indicative visual results of this process are shown in Figure 3.13 by reconstructing the video from each calculated decomposition.

3.9.2 Time-Driven Computation

Conventional real-time software for image processing tasks operates under worst-case assumptions, e.g. see [19]. Here, we want to investigate what happens when scheduling deadlines do not comply with the worst case. To this end, we consider the scenario where, for each video frame, the image processing task of interest is controlled by a scheduler (timer), which terminates the task after the scheduled time per video frame elapses. When the termination signal is received, the task *immediately* provides the already computed results for the input frame, before proceeding to the next video frame. We illustrate this approach in Figure 3.14. In order to implement this design, we have used the cross-platform OpenMP framework [112] where two independent threads (timer and main thread) are concurrently executed. The two threads share the common memory element `flag_int` to realise the signalling: when `flag_int` is set to true by the timer thread, the application thread terminates the processing

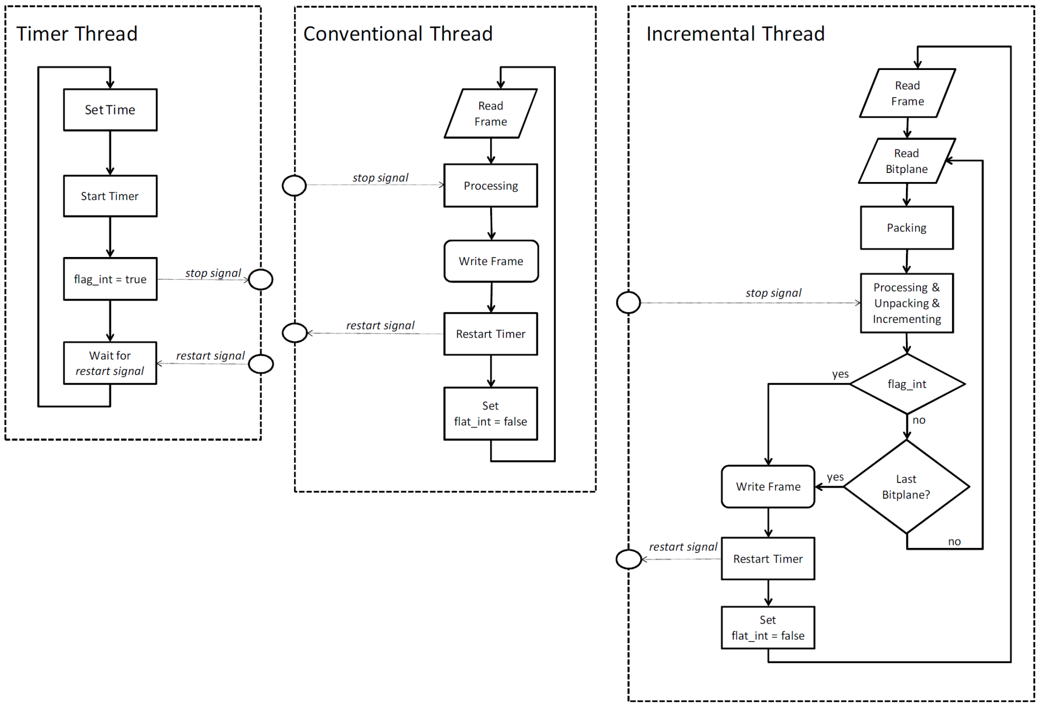


Figure 3.14: Time-driven computation of image processing tasks. The timer thread sends the stop signal to the application thread in order to terminate their execution for each frame. The application thread initiates the timer thread by the `restart signal`. The signalling is achieved via checking and setting/resetting `flag_int`.

of the current frame and resets `flag_int` to `false`.

In our first experiment, the termination signal is generated by the timer thread using an average value `A` with `D%` of variability around the average value. Two cases are considered: (*a*) "regular-variability" scheduling, where `A=100%` of the average frame completion time for each task and `D=30%` of A, and (*b*) "aggressive-variability" scheduling, where `A=80%` of the average frame completion time for each task and `D=50%` of `A`. In order to report results for both conventional and incremental versions of the algorithms, we measure two aspects: (*a*) the percentage of *uncovered frames*; these are frames with areas within them that have not been processed (covered) at all (i.e. areas with no decomposition or filtering, or no block matching for some blocks); (*b*) the percentage of *fully-completed frames*; these are fully-covered frames *and* with the result computed at full pre-

Scheduling Type	Regular-variability (A=100% of each method, D=30%)		Aggressive-variability (A=80% of each method, D=50%)	
Measurement	Uncovered	Fully-completed	Uncovered	Fully-completed
Transform Decomposition				
Conventional	33.90%	66.10%	42.99%	57.01%
Incremental	0.19%	91.06%	5.61%	81.86%
2D Filtering				
Conventional	4.41%	95.59%	6.31%	93.69%
Incremental	0.06%	99.87%	0.56%	97.59%
Block Matching				
Conventional	75.24%	24.76%	79.09%	20.91%
Incremental	0.10%	76.85%	10.88%	67.58%

Table 3.4: Percentage of uncovered and fully-completed frames for 4×4 and 16×16 integer block transforms (top part), 8×8 cross-correlation and 12×12 convolution (middle part), 8×8 and 16×16 block matching (bottom).

cision. Naturally, for optimal performance, the first percentage should be as close to zero as possible, while the second should be as close to 100% as possible. The results are given in Table 3.4.

The proposed approach also has an intermediate case, which is *covered frames but not fully-completed*, i.e. not all increments have been computed. Representative visual examples of the artefacts observed are given in Figure 3.15. Post-processing with error concealment could potentially reduce the distortion caused by uncovered areas in both conventional and incremental processing at the cost of additional complexity. However, the results of Table 3.4 show that the proposed incremental approach rarely requires this, since the percentage of uncovered frames remains well below 1% in all but two experiments. This is an order of magnitude difference with the conventional approach that typically leaves more than 10% of the frames with uncovered areas when operating under scheduling. This demonstrates that, unlike the conventional implementations, the proposed approach obtains reasonable quality even when the scheduler does not provide for the worst-case. It is interesting to observe that, apart from this advantage, the proposed method also provides significantly-higher

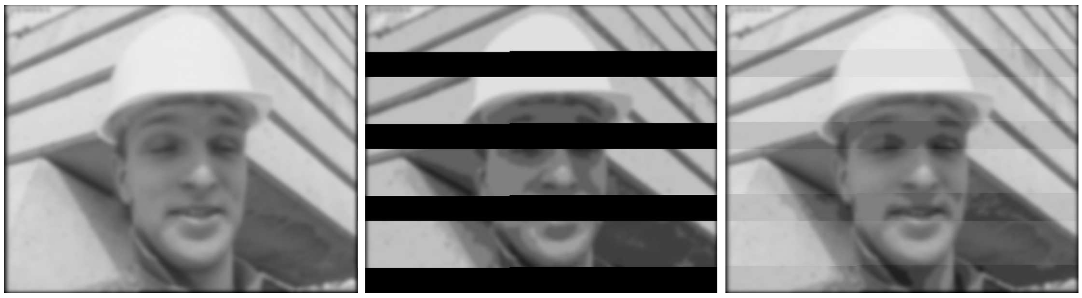


Figure 3.15: Visual example of video frame; from left to right: fully-completed frame, uncovered frame, covered frame but not fully completed (i.e. the result is not computed to full precision).

percentage of fully-completed frames under both scheduling provisions. We observed that the execution time of the proposed incremental approach fluctuates less across different frames in comparison to the conventional approach. This allows for successful completion of more frames for this method when the scheduling time fluctuates around the mean execution time.

In a second scheduling experiment, we want to explore the throughput vs. quality tradeoffs enabled by the proposed approach via execution with fixed deadline per frame. Figure 3.16a shows typical SNR versus throughput results (in terms of frame-per-second – fps) obtained with the incremental 2D convolution with the 12×12 Gaussian mask (mainstream profile). We gradually decreased the scheduling deadline (without variability) from `A=31ms` to `A=19ms` per frame[6], which leads to increased throughput, from 32.3fps to 52.6fps respectively, with a corresponding drop in SNR from infinity (full precision) to 19.36 dB. Representative visual results are shown in Figure 3.9: from left to right the displayed frames represent typical outputs from highest fps to lowest fps, i.e. from stopping at increment layer $n = 5$ to stopping at $n = 0$, respectively. It is important to remark that, for all results reported in Figure 3.16, no uncovered frames were produced, i.e. there were no sudden blanks in the filtered video apart from the gradual quality reduction. This straightfor-

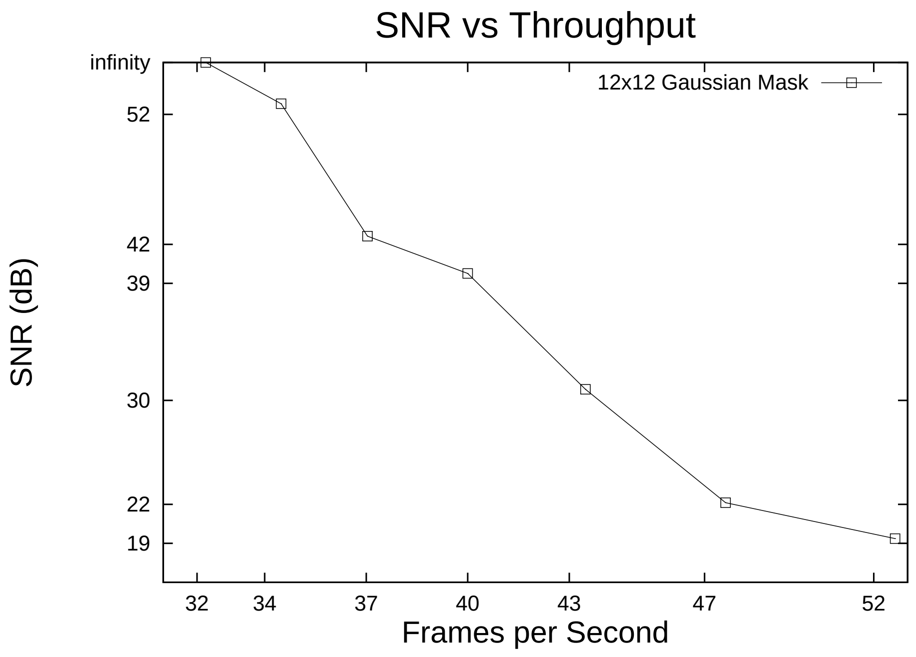
[6]Notice that the scheduling deadline includes I/O time, therefore the scheduling deadlines are higher than the timing measurements of Figure 3.7 that are reporting only the average computation time per frame.

ward quality-complexity scalability provides a very efficient tool when the processing requirements need to be scaled on-the-fly to match throughput requirements. Notice that constant-time execution is complementary to bitplane-based execution (shown in Figure 3.7) where constant-quality processing is achieved but the execution time per frame can vary. The SNR results per frame shown in Figure 3.16b demonstrate this difference; there, the constant-quality execution was terminated at bitplane $n = 2$ per frame, while the constant-time execution imposed `A=24ms` per frame (41.7fps); both methods required virtually the same average time per frame (the difference was within a 5% margin). However, constant-time execution produces occasional drops in SNR in certain frames, while constant-quality execution provides near-constant SNR with occasional bursts of execution time due to differences in the execution flow caused by time-varying processor or operating-system interrupts.

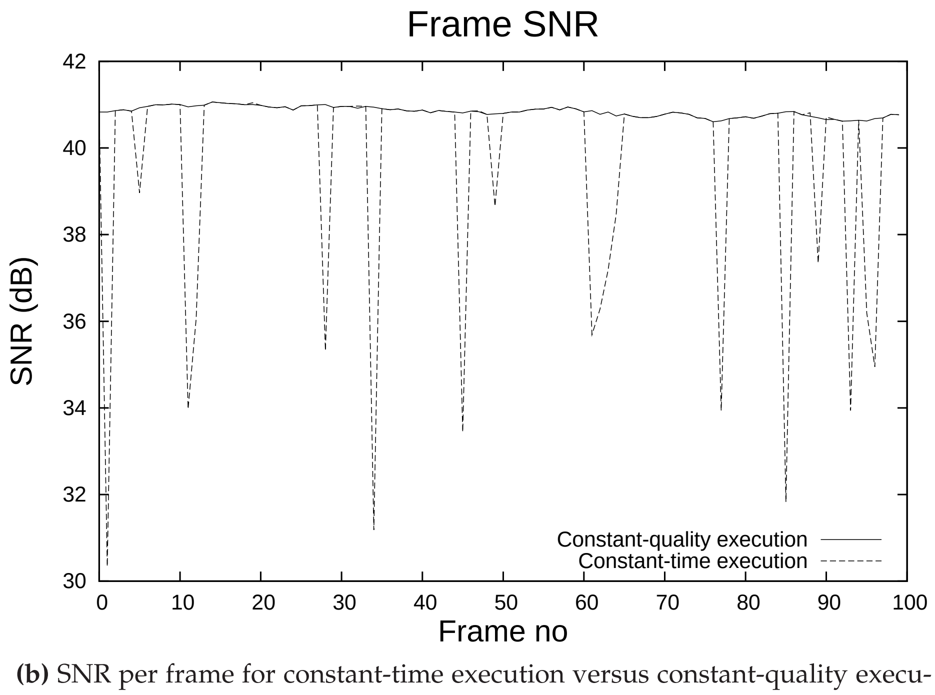
3.9.3 Energy-Distortion Efficiency of Software-based Incremental Computation for Real-Time Video Processing on the XO Laptop

The proposed incremental computation is aiming for accelerated processing by handing only a subset of the input bitplanes. As such, when accelerated processing is obtained, this leads to energy consumption reduction, which is a side-effect stemming from the obtained acceleration. To examine the effect of incremental computation on the energy consumption, in this experiment [8] we use the on-board camera of the xo-laptop to capture still images in raw YCbCr format (640×480 pixels) and apply the 8×8 cross-correlation algorithm with floating-point packing using a high-pass filter kernel. This corresponds to an image capturing and filtering scheme for edge detection in a live monitoring application. We used the Linux Hardware Abstraction Layer[7] to periodically read the xo-laptop's battery status during the algorithm execution. Our goal is to measure the power-level reduction when computing the high-pass filtering with different accuracies (in terms of terminating bitplane

[7]http://www.freedesktop.org/wiki/Software/hal



(a) Experimental results on mainstream profile with reduced scheduling deadline



(b) SNR per frame for constant-time execution versus constant-quality execution

Figure 3.16: SNR comparison in a constant time and a constant quality environment. Example with 100 frames from the"Foreman" sequence.

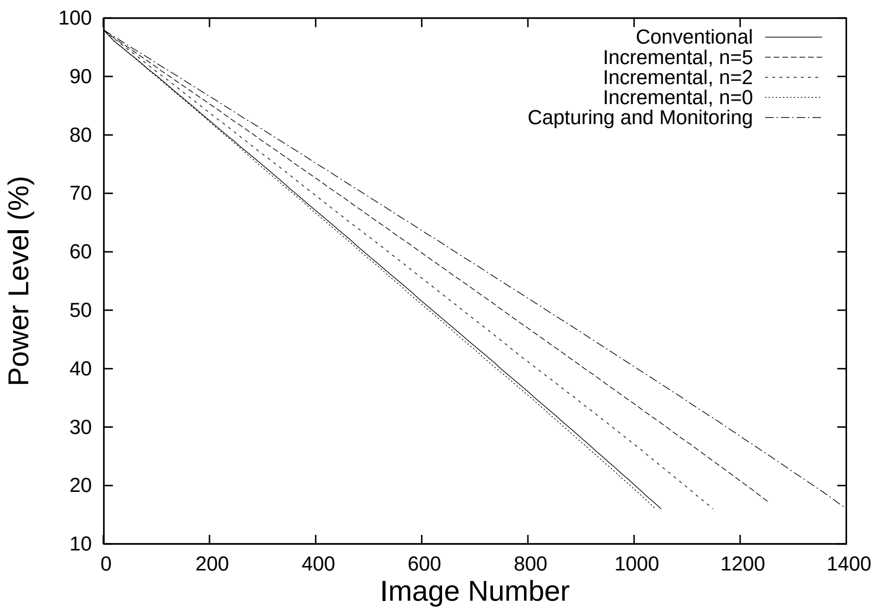


Figure 3.17: Measured battery power-level reduction versus images captured and processed with different precision (different terminating bitplanes n) in the xo-laptop. The dotted line indicates the power-level reduction when operating the frame capturing and battery monitoring only (without any processing).

n). To this end, we switched the xo-laptop to terminal mode and converted the monitor to low-power (reflective) mode [110]. Live image capturing was realised with the gstreamer framework [110]. We run the image capturing and filtering algorithm continuously from battery power level 97% down to power level 17%. Typical output results are shown in Figure 3.17 in terms of battery power level and number of images captured and processed for conventional full-precision processing and incremental processing with different terminating bitplanes.The results demonstrate that the proposed software achieves up to 20% more images processed for the same reduction in battery power level when reducing precision from $n = 0$ to $n = 5$. Conversely, once the experiment passes the 700th image, significant difference in power level can be observed for the same number of images in all approaches. For example, for 900 images, the power level goes from 28% for the conventional and the incremental approach with $n = 0$, to 35% for the incremental approach with $n = 2$, and to 41% for $n = 5$. For a low-power surveillance and monitoring scenario where for most of the time no activity is observed and hence the system can predominantly operate in lowest-precision mode, this indicates that there is potential of offering increased energy autonomy without requiring any specific customisations. Additional techniques, such as platform-specific optimisation to reduce the power consumed by the image capturing process, could provide further improvements.

3.10 Concluding Remarks

Incremental computation is a well-known method to achieve different quality levels from the same input. However, due to the nature of the process, the calculation has to be executed multiple times, i.e. once for each increment of the image. This increases the execution time of the method by a factor that depends on the number of increments. The packing theory presented in the Chapter 2 shows how to perform small dynamic range integer-to-integer linear data processing concurrently. This can be used to compensate for the increase in execution time when performing incremental processing, especially based on the fact that incremental computation tends to decrease the dynamic range of the operations for each increment in comparison to the conventional computation.

This chapter proposed the merging of the packing with the incremental computation. In Section 3.2, a generalisation of the loose packing was proposed in order to accommodate the progressive calculation (and the notation needed). Then transform decomposition (Section 3.3), two-dimensional convolution (Section 3.4) and block matching (Section 3.5) are presented as real examples of the merging of the packing and the incremental computation.

This chapter also contains a full set of experiments: they describe the framework from different angles, such as execution time, time-driven computation or energy-distortion efficiency. Moreover, to demonstrate the generality of the derived software solution, different architectures were used: a common laptop as a mainstream profile and a small device as low-end profile.

From the experiments it is possible to notice that:

- Large bitwidth representations allow for higher packing capability, hence lower execution time;
- Packing is always beneficial because it allows for lower execution time, as shown in Figure 3.5, 3.6, 3.7 and 3.8;
- The packing capability, W, is inherently dependent on the number of terminating bitplanes used in the incremental processing: together they determine if the algorithm is efficient or not for a certain instantiation. If W is higher, the proposed approach becomes very efficient; instead, if W is lower than the number of terminating bitplanes, this approach tends to be inefficient;
- The packing and unpacking functions have to be constructed carefully in software, because they lead to fixed overhead on the execution time and for this reason they have to incur the smallest possible overhead;
- When increasing the input precision (number of bitplanes), the average execution time increases linearly, rather than staying constant, as it happens for the conventional approach.

The objective (SNR) and visual (SSIM) quality of transform decomposition and filtering was presented in Section 3.7.3. For the block matching, a similar set of experiments were presented: execution time can be found in Figure 3.11 and 3.12, while the quality results measured in PSNR are presented in Table 3.2 and 3.3.

Apart for the quality and performance results presented, several applications were described: a region of interest localised execution is analysed in Section 3.9.1, where it is shown how this feature can improve performance without a significant penalty in visual results; a time-driven computation (Section 3.9.2) exploits how the scalable execution can fit different time constraints; and finally, the last application (deployed on the XO Laptop of the OLPC Foundation [110]) shows that setting different quality levels affects the power consumption significantly (Section 3.9.3) [8].

All the information about the code, a short manual and all the references are available online at http://www.ee.ucl.ac.uk/~iandreop/ORIP.html. The project is open source and available for download [99].

Chapter 4

Beyond Lossless Tight Packing

Even though the previous chapters demonstrated scaling of the execution time (or, equivalently, of the processing throughput), the proposal of incremental packed processing has certain limitations:

- It is limited to integer-to-integer incremental linear processing or, in other words, performs fixed-point processing with the input refined progressively;
- Only the packing of one operand (e.g. input image pixels) is performed: it is possible to gain in performance and/or operational flexibility if both operands (e.g. input image and filter kernel) are packed;
- While generic signal processing primitives are converted to incremental form, it would be interesting to explore the processing throughput vs. distortion scaling possibilities of *generic linear algebra* primitives, such as matrix-vector or matrix-matrix processing in BLAS [113];
- Last but not least, while the proposals of the previous chapters can be performed with streaming SIMD extensions (SSE) instructions, this is not demonstrated in practice.

The remaining chapters of this thesis address these issues. In particular, this chapter describes the theory behind *lossy tight packing* when it is applied to a Generic Matrix Multiplication (GEMM) subprogram[1]. An acceleration technique for GEMM based on *dynamically* adjusting the *imprecision* (distortion) of computation is proposed. This technique employs adaptive scalar companding and rounding to input matrix blocks followed by two forms of tight packing in floating-point that allow for concurrent calculation of multiple results. Since the adaptive companding process controls the increase of concurrency (via packing), the increase in processing throughput (and the corresponding increase in distortion) depends on the input data statistics. To demonstrate this, an optimal throughput-distortion control framework for GEMM is derived for the broad class of zero-mean, independent identically distributed, input sources. This approach thus converts matrix multiplication in programmable processors into a *computation channel* since, when increasing the processing throughput, the output noise (error) increases due to: (i) coarser quantisation and (ii) computational errors caused by exceeding the machine-precision limitations

Why is the focus on GEMM? It is well known that GEMM forms the core component of all computationally intensive routines of BLAS Level-3 [114] (e.g. singular value decomposition (SVD), LU, linear solver), as most routines can be written to use GEMM as the problem size scales. Moreover, GEMM has important uses as a stand-alone function as we shall show by the applications of the next chapter.

4.1 Generic Matrix Multiplication (GEMM)

This section describes the preliminaries required for the understanding of the theory behind the lossy tight packing applied to a Generic Matrix Multiplication (GEMM). A detailed description of the implementation and experiments within applications will be presented in Chapter 5.

Consider the GEMM design depicted in Figure 4.1a: the application calls sGEMM or dGEMM (depending on whether single or double precision is required) for

[1] An extended description of the implementation will be discussed in the Chapter 5

an $M \times K$ by $K \times N$ matrix multiplication, [top-level of Figure 4.1b] which is further subdivided into $L \times L$ "inner-kernel" matrix multiplications.

The *inner-kernel* result $\mathbf{R}_{2,1}$ of the example shown in Figure 4.1a comprises multiple subblock multiplications $\mathbf{A}_{2,l}\mathbf{B}_{l,1}$:

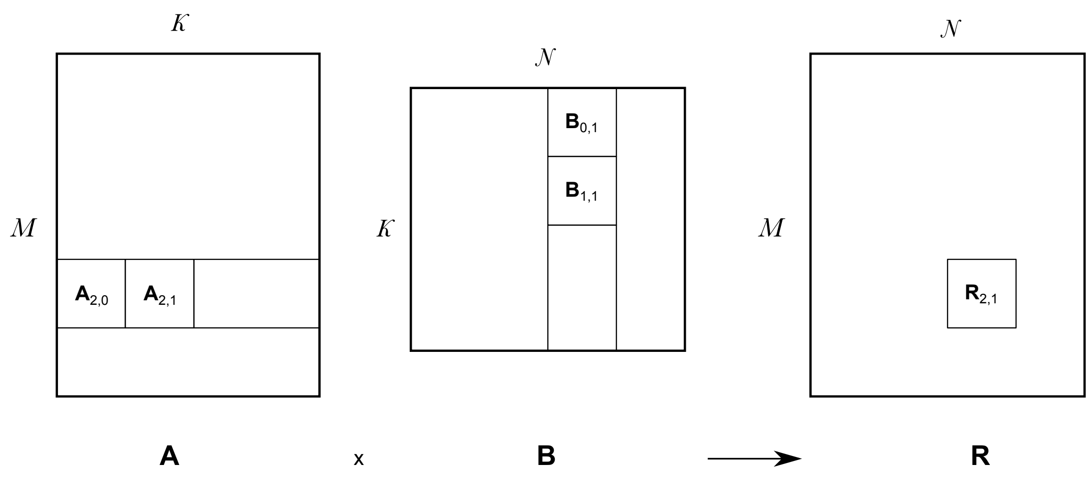
$$\mathbf{R}_{2,1} = \sum_{l=0}^{\frac{K}{L}-1} \mathbf{A}_{2,l}\mathbf{B}_{l,1}. \tag{4.1}$$

If the matrices' dimensions are not multiples of L, some "cleanup" code [14] is applied at the borders to complete an inner-kernel result of the overall matrix multiplication.

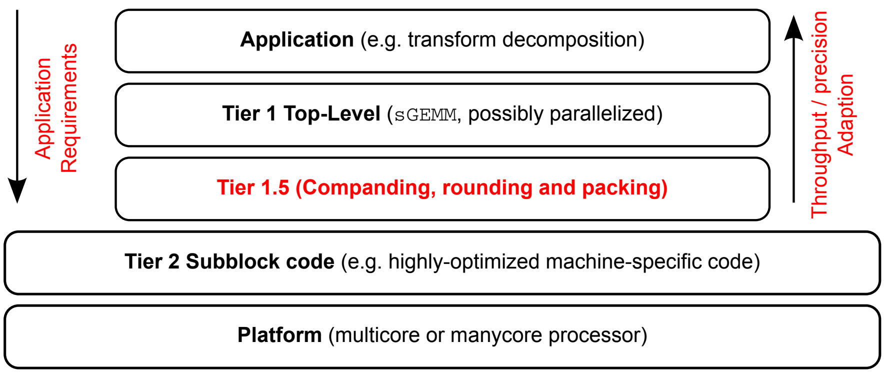
This separation into top-level processing and subblock-level processing is done for efficient cache utilisation and for efficient parallelisation in multithreaded multicore processing environments. Specifically, during the initial data access of sGEMM for top-level processing, data in matrix **A** and **B** is reordered into block major format: for each $L \times L$ pair of subblocks $\mathbf{A}_{i,l}$ and $\mathbf{B}_{l,j}$ multiplied to produce inner-kernel result $\mathbf{R}_{i,j}$, $0 \leq l < \frac{K}{L}$, the input data within $\mathbf{A}_{i,l}$ and $\mathbf{B}_{l,j}$ is scanned in rowwise and columnwise raster manner, respectively. In this way, sequential data accesses are performed during the actual computation and this enables the use of SIMD instructions for each subblock multiplication, thereby leading to significant acceleration[2].

As shown in Figure 4.1b, our proposal creates an intermediate level (Tier 1.5) that performs certain pre-processing for companding, rounding and packing before calling the subblock matrix multiplication code. Once the calculation is completed for each subblock multiplication, post-processing is applied to retrieve the results. The subblock-specific adjustment of the companding factors in our approach allows for data-driven (adaptive) packing and concurrency in the calculation of the inner-kernel results - thus, sGEMM and dGEMM are accelerated according to the input statistics of each subblock. Since our modifications are applied externally to the $L \times L$ subblock matrix multiplication, in the next section we focus on the subblock processing stage. If the matrix sizes

[2]A detailed discussion of SSE instruction will be presented in Section 5.1.1



(a) Illustration of outer and inner-kernel processing of GEMM in BLAS.


Application Requirements
Application (e.g. transform decomposition)
Tier 1 Top-Level (sGEMM, possibly parallelized)
Tier 1.5 (Companding, rounding and packing)
Tier 2 Subblock code (e.g. highly-optimized machine-specific code)
Platform (multicore or manycore processor)
Throughput / precision Adaption

(b) Positioning of the proposed work within the execution environment of signal processing tasks based on optimised linear algebra libraries.

Figure 4.1: Generic Matrix Multiplication (GEMM)

are not multiples of L, trivial size modifications are required for the cleanup code at the borders [14].

4.2 Quantisation via Companding & Rounding

In Chapter 3 we discussed how tight packing can be used in order to increase performance of common image processing algorithms, like convolution and block decomposition. These algorithms map integers to integers, with small input bitwidth (e.g. 8 bit images). Unfortunately, many applications require processing of floating point data and this breaks one of the hypothesis we set in the Section 2.1. At the same time, in many applications a definition of the maximum amplitude of the input dynamic range can not be made in a reliable manner at runtime, which makes impossible to calculate the proper output dynamic range, necessary to safely pack samples.

To solve this problem, an intermediate level performs certain pre-processing for companding, rounding and packing before the actual calculation. Once the calculation is completed, post-processing is applied to each dataset to retrieve the results.

Quantisation is adjusted according to the input's numerical range, which is calculated by reading each input block and storing its maximum amplitude. This operation is performed during each input block reordering to minimise its impact in the overall runtime (i.e. in the overall runtime of the matrix multiplication routine), as it will discuss in detail in Chapter 5.

In the case of the matrix multiplication, each input block is an $L \times L$ subblock of the input matrices. Uniform companding and rounding within each subblock $\mathbf{A}$ and $\mathbf{B}$ is performed, assuming input ranges $[-\max \mathbf{A}, \max \mathbf{A}]$ and $[-\max \mathbf{B}, \max \mathbf{B}]$ symmetric. For any element $a_{m,n}, b_{m,n}$ of $\mathbf{A}$, $\mathbf{B}$, $0 \leq m, n < L$, we have:

$$\begin{aligned} \tilde{a}_{m,n} &= \lfloor c_{\mathbf{A}} a_{m,n} \rceil \\ \tilde{b}_{m,n} &= \lfloor c_{\mathbf{B}} b_{m,n} \rceil \end{aligned}. \tag{4.2}$$

Hence, $\tilde{a}_{m,n}$ are integers within $\{\lfloor -c_{\mathbf{A}} \max \mathbf{A} \rceil, \ldots, \lfloor c_{\mathbf{A}} \max \mathbf{A} \rceil\}$, with $c_{\mathbf{A}}$ the companding coefficient ($c_{\mathbf{A}} > 0$) that is designed according to precision requirements for the final result (equivalently for $\tilde{b}_{m,n}$ and $c_{\mathbf{B}}$). Those values are then used in the core of the calculation, which, for each output element, is a vector inner product with L multiplier-accumulator (MAC) operations:

$$\tilde{r}_{m,n} = \sum_{l=0}^{L-1} \tilde{a}_{m,l} \tilde{b}_{l,n}. \tag{4.3}$$

The final result is recoverd after reverse companding:

$$\hat{r}_{m,n} = \frac{1}{c_{\mathbf{A}} c_{\mathbf{B}}} \tilde{r}_{m,n}. \tag{4.4}$$

Despite the distortion $\left\| \tilde{\mathbf{R}} - \mathbf{R} \right\|_F^2$ due to quantisation, using (4.3) for the actual computation of $\tilde{r}_{m,n}$ would not lead to any acceleration in a programmable processor; however, once the inputs have been companded and rounded, we can create reduced-size input blocks by packing multiple inputs together via two different methods. Both methods aim for accelerated processing.

4.3 Packing Techniques

In Chapter 2 we presented a complete discussion on how the packing is performed for three class of problems: block decomposition, convolution and block matching. The following two subsections will expand further this topic, showing two different methods to perform packing inside the matrix multiplication routine.

4.3.1 Asymmetric Packing

In the first method, only one input block (namely $\tilde{\mathbf{A}}$) is packed. This is following the packing described in Chapter 2 and also proposed in previous work [6, 100]. Specifically, once the inputs have been companded and rounded by (4.2), the packing process creates block $\overline{\mathbf{A}}$ with $\frac{L}{W} \times L$ coefficients given by $(\forall m, n : 0 \leq m, n < L, \overline{m} = \lfloor \frac{m}{W} \rfloor)$:

$$\overline{a}_{\overline{m},n} = \sum_{p=0}^{W-1} z^p \tilde{a}_{W\overline{m}+p,n}, \tag{4.5}$$

where z, $0 < z < 1$, is the utilised packing coefficient. Once (4.5) is completed, processing occurs via $\overline{\mathbf{R}} = \overline{\mathbf{A}}\tilde{\mathbf{B}}$ ($\forall m, n : 0 \leq m, n < L, \overline{m} = \lfloor \frac{m}{W} \rfloor$):

$$\bar{r}_{\overline{m},n} = \sum_{j=0}^{L-1} \bar{a}_{\overline{m},j}\tilde{b}_{j,n} = \sum_{j=0}^{L-1}\sum_{p=0}^{W-1} z^p \, \tilde{a}_{W\overline{m}+p,j}\tilde{b}_{j,n} \tag{4.6}$$

The packed result of (4.6) contains the output of groups of W rows packed together. Since the processing is performed in the function's native representation, any high-performance $\frac{L}{W} \times L$ by $L \times L$ software kernel for sGEMM can be used for (4.6). Following the completion of the processing, unpacking of the results can be performed by the following iterative process, as already shown in Subsection 2.1 and Subsection 2.2 [6] ($\forall m, n : 0 \leq m, n < L, \overline{m} = \lfloor \frac{m}{W} \rfloor$):

$$\tilde{r}_{\overline{m},n} = \lfloor \bar{r}_{\overline{m},n} \rceil \tag{4.7a}$$

$$\forall p \in \{1, \ldots, W-1\} : \begin{cases} \bar{r}_{\overline{m}+p,n} = \frac{1}{z}(\bar{r}_{\overline{m}+p-1,n} - \tilde{r}_{\overline{m}+p-1,n}) \\ \tilde{r}_{\overline{m}+p,n} = \lfloor \bar{r}_{\overline{m}+p,n} \rceil \end{cases} \tag{4.7b}$$

Reverse companding can be applied to each $\tilde{r}_{m,n}$ via (4.4).

4.3.2 Symmetric Packing

In the second method, once the inputs have been companded and rounded, the packing process creates two blocks $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$ with $L \times \frac{L}{W}$ and $\frac{L}{W} \times L$ coefficients (respectively) given by ($\forall m, n : 0 \leq m, n < L, \overline{m} = \lfloor \frac{m}{W} \rfloor, \overline{n} = \lfloor \frac{n}{W} \rfloor$):

$$\bar{a}_{m,\overline{n}} = \sum_{p=0}^{W-1} z^p \, \tilde{a}_{m,W\overline{n}+p} \tag{4.8a}$$

$$\bar{b}_{\overline{m},n} = \sum_{p=0}^{W-1} z^{-p} \, \tilde{b}_{W\overline{m}+p,n} \tag{4.8b}$$

where z, $0 < z < 1$, is the utilised packing coefficient that controls the allocated space for each packed input[3] within the floating point representation

[3]Details on the setting of z were discussed in Subsection 2.1.

and W the number of packings performed. Notice that, unlike previous work on block-based packed processing [6, 11, 100], the elements of *both* $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$ are packed along input rows and columns (respectively). Due to the block major-format reordering, (4.8a) and (4.8b) perform MAC operations in sequential memory elements, thereby allowing the use of SIMD instructions for accelerated processing.

Processing occurs using the packed data, i.e. $\overline{\mathbf{R}} = \overline{\mathbf{AB}}$:

$$\begin{aligned}\forall m,n : \bar{r}_{m,n} &= \sum_{j=0}^{\frac{L}{W}-1} \bar{a}_{m,j}\bar{b}_{j,n} = \\ &= \sum_{j=0}^{\frac{L}{W}-1}\left(\sum_{p=0}^{W-1} \tilde{a}_{m,Wj+p}\,\tilde{b}_{Wj+p,n} + \sum_{p=0}^{W-1}\sum_{\substack{h=0\\h\neq p}}^{W-1} z^{p-h}\tilde{a}_{m,Wj+p}\tilde{b}_{Wj+h,n}\right)\end{aligned} \tag{4.9}$$

The packed result of (4.9) contains the required output as well as $(W^2 - W)$ "side" outputs: $o_{p,h} = z^{p-h}\,\tilde{a}_{m,Wj+h}\,\tilde{b}_{Wj+h,n}, \forall p,h : 1 \leq p,h < W \;\&\; p \neq h$. Notice that (4.9) is performed in the function's native representation. As such, any high-performance $L \times \frac{L}{W}$ by $\frac{L}{W} \times L$ software kernel for sGEMM or dGEMM can be used for (4.9), as indicated in the subblock processing of Figure 4.1b. Due to companding and packing, (4.9) performs W times the operations of conventional SIMD-based matrix multiplication; we term this approach as *turbo SIMD*.

Following the completion of the processing, unpacking of the results can be performed by $(\forall m,n)$:

$$u_{m,n} = \lfloor \bar{r}_{m,n} \rceil \tag{4.10a}$$

$$\tilde{r}_{m,n} = u_{m,n} - (z^{-1}\lfloor z u_{m,n} \rceil) \tag{4.10b}$$

The unpacking process extracts the useful result from the packed output by: (a) the rounding operation to remove the first unneeded set of results, $o_{p,h}$ with $p > h$, of (4.9); (b) removing the second unneeded set of results, $o_{p,h}$ with $p < h$, of (4.9) by (4.10b). Reverse companding can be applied to each $\tilde{r}_{m,n}$ via (4.4).

4.3.3 Differences between symmetric and asymmetric packing

Remark 4.1 (Packing Trade-off)**.** The symmetric packing approach of Subsection 4.3.2 produces $L \times L$ outputs, each requiring L/W MAC operations. It requires $2L^2W^{-1}b_{\text{repr}}$ bytes for storage of the packed input data with b_{repr} respectively equal to 4 and 8 for single-precision and double-precision. On the other hand, the asymmetric packing of Subsection 4.3.1 produces $(L/W) \times L$ outputs (that are then unpacked to the final $L \times L$ outputs), each requiring L MAC operations. It requires $L^2(1 + W^{-1})b_{\text{repr}}$ bytes for storage of the input data. Evidently, both approaches have the same complexity in terms of MAC operations. They are both memory efficient in comparison to the conventional approach that requires $2L^2b_{\text{repr}}$ bytes of memory for the input data. Between them, the symmetric packing requires the least amount of memory for the computation. However, the disadvantage of the symmetric packing approach is that the second group of side results of the packed output (i.e. $o_{i,h}$ with $i < h$) is occupying space in the numerical representation even though it is not used in the results. ■

Remark 4.2 (Integer Processing)**.** Even though one may consider avoiding the use of packing and instead construct matrix multiplication in 16-bit integer representations (integer SIMD instructions exist for all mainstream processors), this has the following detriments:

1. integer conversion to and from floating point (which is the general input required by linear algebra and digital signal processing (DSP) applications) leads to significant overhead since it cannot be performed with SIMD instructions;
2. unlike floating point representations where the maximum packed value can be flexible with graceful degradation in the results, a strict limit is set on the quantised values in integers in order to avoid overflow. ■

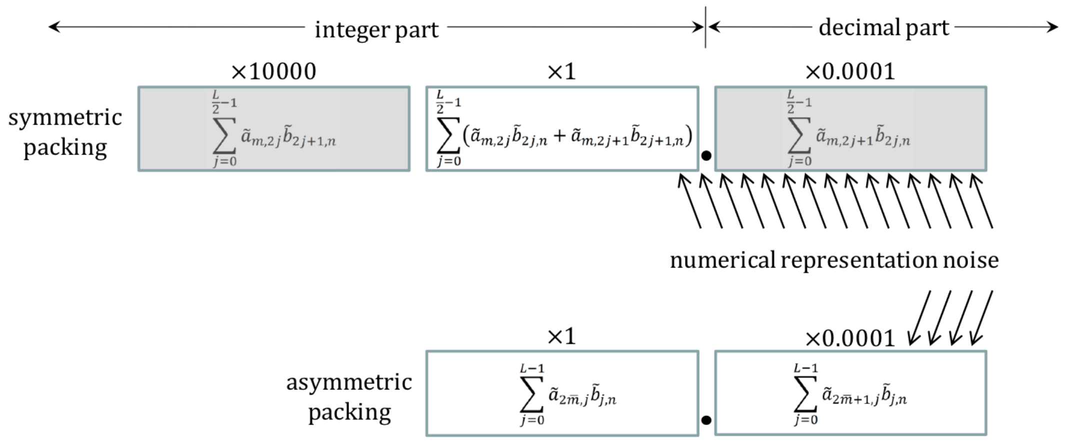

integer part
decimal part
×10000
×1
×0.0001
symmetric packing
$\sum_{j=0}^{\frac{L}{2}-1} \tilde{a}_{m,2j}\tilde{b}_{2j+1,n}$
$\sum_{j=0}^{\frac{L}{2}-1} (\tilde{a}_{m,2j}\tilde{b}_{2j,n} + \tilde{a}_{m,2j+1}\tilde{b}_{2j+1,n})$
$\sum_{j=0}^{\frac{L}{2}-1} \tilde{a}_{m,2j+1}\tilde{b}_{2j,n}$
numerical representation noise
×1
×0.0001
asymmetric packing
$\sum_{j=0}^{L-1} \tilde{a}_{2\bar{m},j}\tilde{b}_{j,n}$
$\sum_{j=0}^{L-1} \tilde{a}_{2\bar{m}+1,j}\tilde{b}_{j,n}$

Figure 4.2: Conceptual example of $W = 2$ packings with $z = 0.0001$. Top: result of (4.9); bottom: result of (4.6). Shaded blocks contain side results that are produced during the packed processing but not used in GEMM.

The last point of Remark 4.2 identifies that, unlike integer representations, floating-point representations are lossy by construction, in order to allow for a more flexible representation of real numbers without running into overflow problems for well-conditioned numerical computations [102]. In the proposed packing approaches of the last two subsections, this can be exploited by appropriate adjustment of packing factor, z, in order to increase the space allowed for the quantised inputs and thus decrease the distortion incurred by the proposed packed computation. We shall in fact show that significantly-higher quantisation accuracy can be achieved within packing of $W = 2$ in single-precision floating point representation than within the 16-bit integer representation.

A conceptual illustration of the packed representation of (4.9) and (4.6) and how the floating-point representation noise affects the packed results is given in Figure 4.2. Symmetric packing is better protected from the numerical representation noise, because the“lower” side result (multiplied by $z = 0.0001$) is not used; this is despite the fact that noise is amplified in this representation as the “higher” side result (multiplied by $z^{-1} = 10000$) takes the number further away from the high-precision region around zero [102]. This representation noise creates the notion of computational capacity in our approach:

for given quantisation distortion, there is a limit on the throughput increase achieved via increased packing (i.e. increased values for W, surpassing $W = 2$ shown in Figure 4.2), beyond which the distortion stemming from the floating point computation surpasses the quantisation distortion. The interdependency between throughput and distortion and the notion of computational capacity make our approach a computation channel for generic matrix multiplication.

4.4 GEMM as a computation channel

In the proposed framework, noise stems from companding and rounding, but it can also stem from erroneous unpacking of the packed results.

4.4.1 Summary of known results on operational tight packing

The quantisation of (4.2) may not be the only noise source in the proposed packed processing. The real-number space is not mapped linearly in the floating-point representation, thereby resulting in higher accuracy around zero [102]. Limits for error-free packing of integers have been established within the tight packing theory in [6, 100].

Proposition 1 (Subsection 2.1.1) states that error-free unpacking of W_{ef} signed integers [6] by (4.7a) and (4.7b) or (4.10a) and (4.10b) after performing (4.6) or (4.9), respectively, requires[4]:

$$z < \frac{1}{2R_{\max} + u^*_{\text{safe}}} \tag{4.11}$$

$$W_{\text{ef}} \leq \lfloor \log_z \left[(2R_{\max} + 1) u_{\text{sys}} \right] + 1 \rfloor, \tag{4.12}$$

with: u_{sys} the relative precision of the computer hardware/software used for

[4]Proof was given in Subsection 2.1.1.

implementation [6],

$$u_{\text{safe}}^{*} = \arg \min_{\forall u_{\text{safe}} \in \mathbb{R}^{+}} \left| (R_{\max} + u_{\text{safe}})^{W_{\text{ef}}-1} (1 - u_{\text{safe}}) - R_{\max} \right| \tag{4.13}$$

and, within the context of the proposed companding process of (4.2), the maximum amplitude of the elements of the packed matrix $\overline{\mathbf{R}}$, $R_{\max}$, given by:

$$R_{\max} = \lceil c_{\mathbf{A}} c_{\mathbf{B}} L \max \mathbf{A} \max \mathbf{B} \rceil. \tag{4.14}$$

In practice, one can set $u_{\text{safe}}^{*} = 50$ to cover all possible scenarios with no practical loss in the packing capability. A simple algorithm to calculate z and W_{ef} for any processor is provided in Subsection 3.5.2.3 [6, 100].

Can we go beyond the limits of error-free packed processing of Proposition 1 under the proposed throughput/distortion framework?

There are two ways this may be possible. Firstly, one can attempt to increase z beyond the limit of (4.11) in order to "squeeze in" more data in the floating-point representation. In such a case, the output results from the quantised processing of (4.6) or (4.9) may "invade" each other causing catastrophic error during unpacking [6, 100]. Because of the severity of the errors caused, this is clearly an undesired option. As an alternative, one can attempt to utilise values for packing beyond W_{ef} that is the limit set by (4.12). If one does apply such increased packing, distortion will gradually be introduced in the unpacked results of (4.10b) and (4.7a), (4.7b). However, this may be acceptable since quantisation already introduces approximation. To this end, by modifying $R_{\max}$ for every W, $W > 1$, one can systematically investigate the trade-off between quantisation-induced error and representation induced error: high values for $R_{\max}$ reduce the quantisation error [since $c_{\mathbf{A}}$ and $c_{\mathbf{B}}$ can be increased in (4.14)] but may lead to significant representation induced error if the bound of (4.12) is violated (i.e. if we use $W > W_{\text{ef}}$ for the selected $R_{\max}$ value); low values for $R_{\max}$ cause the reverse effect. Thus, to establish the optimal operational conditions for throughput/distortion processing via the use of companding and packing, we must identify the appropriate value of $R_{\max}$ for each packing W.

We investigate this via a statistical characterisation for the combined impact of quantisation-induced and representation induced error.

4.4.2 Noise of Packed Results in Floating-point Representations

In order to collect statistics from the representation induced error under packed processing, we use integer samples for $\mathbf{A}$ and $\mathbf{B}$ with $L = 288$ and we set $c_{\mathbf{A}} = c_{\mathbf{B}} = 1$ in all these experiments, i.e. no loss is caused from companding and rounding. The experiments are performed with $\max \mathbf{A} = 22$ and $\max \mathbf{B} \in \{1, 2, \ldots, 63\}$ using five random instantiations for $\mathbf{A}$ and $\mathbf{B}$ for each combination of maximum input values[5]. Using (4.14), the selected ranges and subblock size lead to output range $R_{\max} \in \{6336, 12672, \ldots, 399168\}$, which encompasses the range where $W_{\text{ef}} \in \{2, 3, 4\}$ is obtained in single- or double-precision floating point representations. For each instantiation of each combination of $R_{\max}$ and W, we measure the mean error and the MSE of each matrix multiplication in packed form by:

$$\forall \max \mathbf{A}, \max \mathbf{B}, W : \begin{cases} m(R_{\max}, W) = \frac{1}{L^2} \sum_{m=0}^{L-1} \sum_{n=0}^{L-1} (\hat{r}_{m,n} - r_{m,n}) \\ v(R_{\max}, W) = \frac{1}{L^2} \left\| \hat{\mathbf{R}} - \mathbf{R} \right\|_F^2 \end{cases} \tag{4.15}$$

respectively, with: $W \in \{2, 3, 4\}$ packings, z set for each W according to (4.11) and R computed with $W = 1$ (conventional computation) under the same data type (single or double-precision floating point representation).

When $m(R_{\max}, W) \cong 0$ (approximately zero bias), $v(R_{\max}, W)$ approaches the sample variance of the error. Furthermore, given that in this experiment there is no quantisation noise, any mismatch in the results stems solely from the imprecision caused by the numerical representation. Specifically, depending on the value of W and the input dynamic range, packed processing may not

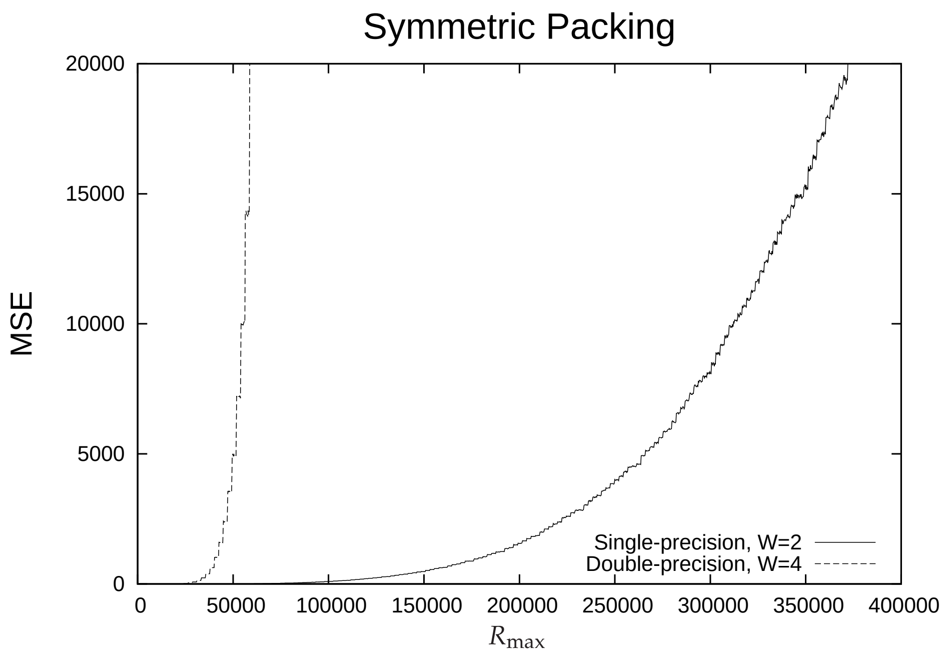
[5]The ranges for $\mathbf{A}$ and $\mathbf{B}$ were chosen so as to produce $R_{\max}$ values within the range required to cover the numerical representation limits for each number of packings W; the moments of the representation induced error are invariant to the specific ranges and depend solely on $R_{\max}$ and W.

induce any error [6, 100]. However, once $W > W_{\mathrm{ef}}$ of (4.12) for the setting of $R_{\max}$ of a particular measurement, the output will contain numerical errors.

Figure 4.3 shows the relationship of MSE with $R_{\max}$ for single and double-precision floating-point representations in an Intel Core 2 Duo P8800 processor (with $W = 2$ and $W = 4$, respectively). Figure 4.4 presents the average error for each case. This experiment indicates that, depending on the numerical representation (single or double-precision) and the value of W, one can select a range of $R_{\max}$ values and assume zero average error $m(R_{\max}, W)$, i.e. no systematic error. In such a case, the sample variance of the representation induced error is monotonically increasing with $R_{\max}$.

Since each value of $m(R_{\max}, W)$ and $v(R_{\max}, W)$ reported in Figure 4.3 and Figure 4.4 corresponds to hundreds of thousands of independent inner-product calculations [via (4.9) or (4.6)], the results of Figure 4.3 and Figure 4.4 are a good approximation of the ensemble statistics. Thus, they are independent of the actual data being used for the matrix multiplication and stem solely from the numerical representation limitations and the utilised packing.

Interestingly, for symmetric packing in single-precision floating point representation with $W = 2$, we can use up to $R_{\max} = 320000$ and obtain: $m(R_{\max}, 2)/R_{\max} < 0.0001\%$ and $\sqrt{v(R_{\max}, 2)}/R_{\max} < 0.04\%$. This indicates that one can utilise $R_{\max}$ values that are up to an order of magnitude higher than the maximum range of 16-bit signed integer representations (where $R_{\max} < 32768$ is imposed in order to avoid overflow) with very small representation induced error. Overall, the proposed packed processing in floating-point representations allows for significantly increased quantisation range at the cost of gradually increased representation induced noise [i.e. increased noise variance $v(R_{\max}, W)$ for increased $R_{\max}$]. This noise is significantly smaller for the case of symmetric packing, as shown by the comparison between the two graphs of Figure 4.3. This occurs because this packing does not use outputs $o_{p,h}$ of (4.9) with $p > h$, which are the outputs affected the most by the representation induced noise as they correspond to the decimal part of the packed result. This part is indeed used in the asymmetric packing that actually extracts all outputs of (4.6) via the iterative unpacking of (4.7a), (4.7b).



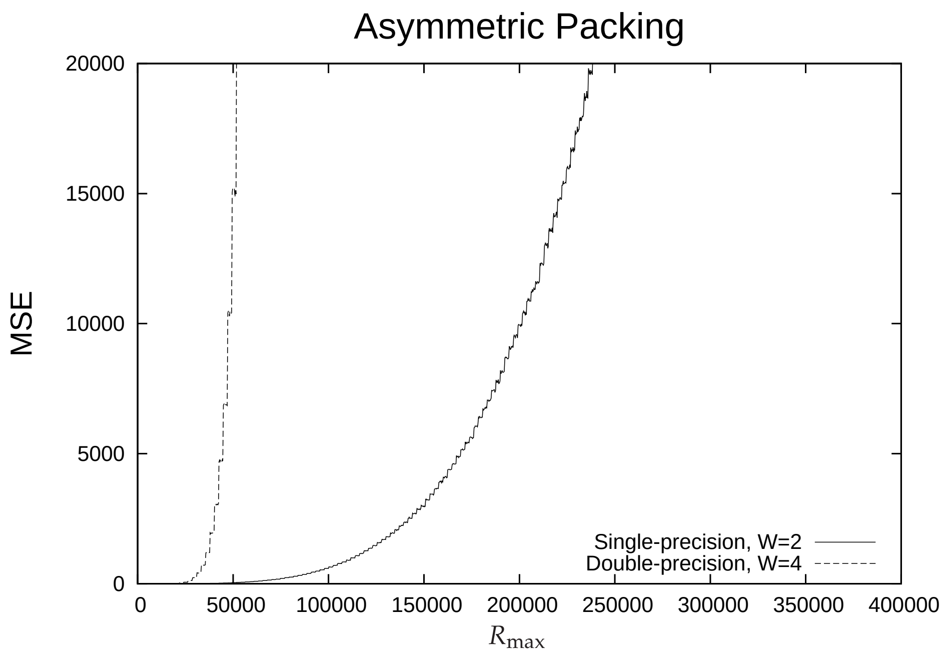
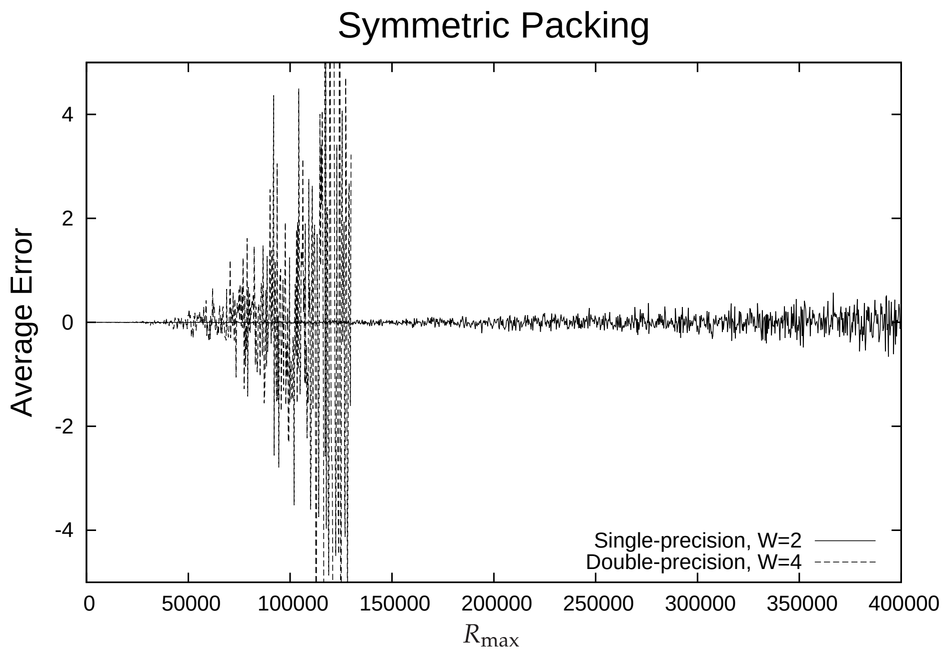


Figure 4.3: *Mean squared error measurements* for matrix multiplication of integer inputs leading to different values of R_{max} within single-precision and double-precision representation and without quantisation ($c_{\mathbf{A}} = c_{\mathbf{B}} = 1$).



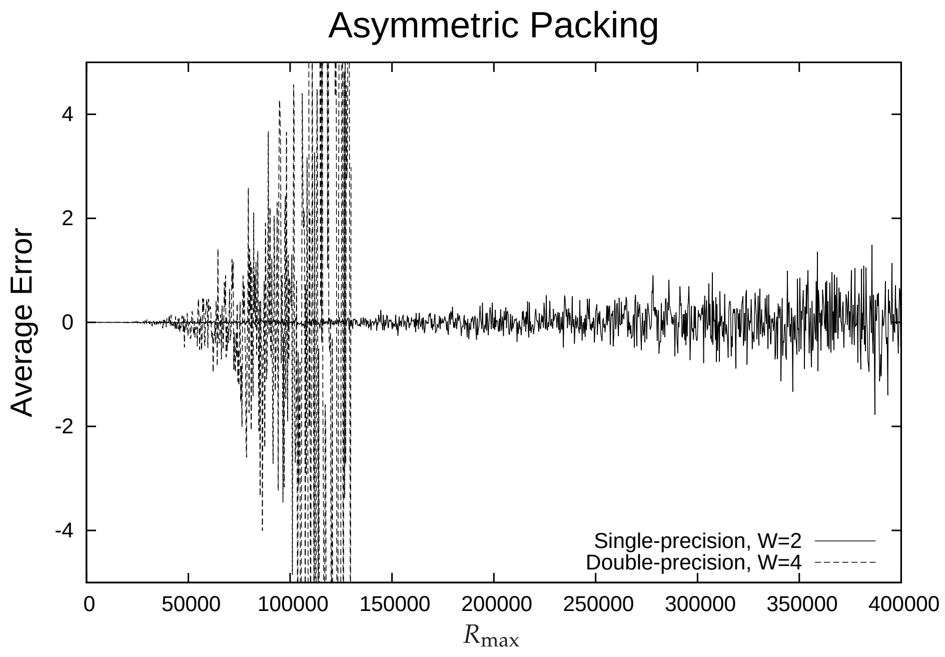


Figure 4.4: *Average error measurements* for matrix multiplication of integer inputs leading to different values of $R_{\max}$ within single-precision and double-precision representation and without quantisation ($c_{\mathbf{A}} = c_{\mathbf{B}} = 1$). The results of double precision with $W = 4$ are not displayed beyond $R_{\max} > 120000$ as they clearly exceed acceptable limits.

4.4.3 Quantisation Noise Model

We now present a statistical model of the quantisation noise introduced via companding. Inputs $a_{m,n}$ and $b_{m,n}$ are modeled as zero-mean i.i.d., random variables (RVs) $\alpha \sim \mathrm{P}_{\alpha}(\sigma_{\alpha})$, $\beta \sim \mathrm{P}_{\beta}(\sigma_{\beta})$, respectively. Quantisation noise in $\tilde{a}_{m,n}$, $\tilde{b}_{m,n}$ is modeled as additive, zero-mean, i.i.d. white [115], RVs $\nu_{\alpha} \sim \mathrm{P}_{\nu_{\alpha}}(\sigma_{\nu_{\alpha}})$, $\nu_{\beta} \sim \mathrm{P}_{\nu_{\beta}}(\sigma_{\nu_{\beta}})$ with $\sigma_{\nu_{\alpha}} = 1/(c_{\mathbf{A}}\sqrt{12})$, $\sigma_{\nu_{\beta}} = 1/(c_{\mathbf{B}}\sqrt{12})$, i.e. the standard deviation of the noise per matrix is scaled according to the companding applied. Finally, the output results $\hat{a}_{m,n}$ are modeled by zero-mean i.i.d. RVs $\hat{\rho} \sim \mathrm{P}_{\hat{\rho}}(\sigma_{\hat{\rho}})$.

Proposition 2 (Quantisation Noise Power)**.** *Under i.i.d. inputs modeled by $\alpha \sim \mathrm{P}_{\alpha}(\sigma_{\alpha})$, $\beta \sim \mathrm{P}_{\beta}(\sigma_{\beta})$ and i.i.d. quantisation noise modeled by $\nu_{\alpha} \sim \mathrm{P}_{\nu_{\alpha}}(\sigma_{\nu_{\alpha}})$, $\nu_{\beta} \sim \mathrm{P}_{\nu_{\beta}}(\sigma_{\nu_{\beta}})$, the expected noise power of the output results of the matrix multiplication under error-free unpacking is $(0 \leq m, n < L)$:*

$$E\left\{(\rho_{m,n} - \hat{\rho}_{m,n})^2\right\} = L\left[(\sigma_{\alpha}\,\sigma_{\nu_{\beta}})^2 + (\sigma_{\beta}\,\sigma_{\nu_{\alpha}})^2 + (\sigma_{\nu_{\alpha}}\,\sigma_{\nu_{\beta}})^2\right]. \tag{4.16}$$

Proof. From (4.3) and under i.i.d. input statistics, the resulting output random variable $\hat{\rho}_{m,n}$ is:

$$\hat{\rho}_{m,n} = \sum_{l=0}^{L-1} \hat{\alpha}_{m,l}\hat{\beta}_{l,n} \tag{4.17}$$

where, $\forall m,l,n : \mu_{\hat{\alpha}_{m,l}} = \mu_{\hat{\beta}_{l,n}} = 0$, $\sigma_{\hat{\alpha}_{m,l}} = \sigma_{\alpha} + \sigma_{\nu_{\alpha}}, \sigma_{\hat{\beta}_{l,n}} = \sigma_{\beta} + \sigma_{\nu_{\beta}}$. We can express $\hat{\rho}_{m,n}$ in affine form [116]:

$$\hat{\rho}_{m,n} = \sum_{l=0}^{L-1}\left(\mu_{\hat{\alpha}_{m,l}}\mu_{\hat{\beta}_{l,n}} + \mu_{\hat{\alpha}_{m,l}}\sigma_{\hat{\beta}_{l,n}}\chi_l + \mu_{\hat{\beta}_{l,n}}\sigma_{\hat{\alpha}_{m,l}}\chi_{L+l} + \sigma_{\hat{\alpha}_{m,l}}\sigma_{\hat{\beta}_{l,n}}\chi_{2L+l}\right) \tag{4.18}$$

with $\chi_0, \ldots, \chi_{3L-1} \sim \mathrm{P}_{\chi}(1)$ zero-mean i.i.d. RVs with unit standard deviation. Expanding on (4.18), we have:

$$\hat{\rho}_{m,n} = \sum_{l=0}^{L-1}\left(\sigma_{\alpha}\sigma_{\beta}\psi_l + \sigma_{\alpha}\sigma_{\nu_{\beta}}\psi_{L+l} + \sigma_{\beta}\sigma_{\nu_{\alpha}}\psi_{2L+l} + \sigma_{\nu_{\alpha}}\sigma_{\nu_{\beta}}\psi_{3L+l}\right) \tag{4.19}$$

with $\psi_0, \ldots, \psi_{4L-1} \sim \mathrm{P}_{\psi}(1)$ zero-mean i.i.d. RVs with unit standard deviation.

The equivalent expression for the inner product of a row of **A** with a column of **B** is:

$$\rho_{m,n} = \sum_{l=0}^{L-1} \sigma_\alpha \sigma_\beta \psi_l. \tag{4.20}$$

Hence, the noise is expressed by:

$$\rho_{m,n} - \hat{\rho}_{m,n} = \sum_{l=0}^{L-1} \left(\sigma_\alpha \sigma_{v_\beta} \psi_{L+l} + \sigma_\beta \sigma_{v_\alpha} \psi_{2L+l} + \sigma_{v_\alpha} \sigma_{v_\beta} \psi_{3L+l} \right) \tag{4.21}$$

and the expected noise power, $E\left\{(\rho_{m,n} - \hat{\rho}_{m,n})^2\right\}$, is given by (4.16). □

The expected power of the (error-free) output **R** is $L(\sigma_\alpha \sigma_\beta)^2$. Hence, the expected SNR of $\hat{\mathbf{R}}$ versus **R** is:

$$E\left\{S_{\text{subblock}}\right\} = 10 \log_{10} \frac{L(\sigma_\alpha \sigma_\beta)^2}{E\left\{(\rho_{m,n} - \hat{\rho}_{m,n})^2\right\}}. \tag{4.22}$$

Notice that, if the statistics of the input matrices are known (i.e. assuming known or estimated P_α and P_β), (4.16) and (4.22) are parameterised solely by $c_\mathbf{A}$ and $c_\mathbf{B}$.

4.4.4 Combined Noise Model

Since the quantisation noise of Proposition 2 and the representation induced noise of Subsection 4.4.2 stem from physically independent processes, we can assume they are statistically independent. The expected noise power of the output results is then:

$$\begin{aligned} D_W\left(R_{\max}, c_\mathbf{A}, c_\mathbf{B}\right) = L &\left[(\sigma_\alpha \sigma_{v_\beta})^2 + (\sigma_\beta \sigma_{v_\alpha})^2 + (\sigma_{v_\alpha} \sigma_{v_\beta})^2 \right] \\ &+ \frac{1}{(c_\mathbf{A} c_\mathbf{B})^2} v(R_{\max}, W) \end{aligned} \tag{4.23}$$

where $v(R_{\max}, W)$ is the experimentally-measured MSE of the representation induced noise for values of $R_{\max}$ and W that correspond to $m(R_{\max}, W) \cong 0$

[examples for $W = 2$ and $W = 4$ are given in Figure 4.3], and factor $(c_{\mathbf{A}} c_{\mathbf{B}})^{-2}$ maps the representation induced noise from the quantisation index domain to the output value domain [via the reverse companding of (4.4)]. For a given choice of packing W, $W > 1$, we can express the intuitive trade-off between quantisation and representation induced noise by combining (4.23) and (4.14): increasing $R_{\max}$ leads to increased values for $c_{\mathbf{A}}$ and $c_{\mathbf{B}}$ and therefore reduced quantisation noise variances σ_{ν_α} and σ_{ν_β} and reduced quantisation noise power from Proposition 2; however, as shown in Figure 4.3, the representation induced noise $v(R_{\max}, W)$ increases monotonically with $R_{\max}$. Hence, D_W, as expressed by (4.23), becomes the mechanism for adjusting the desired throughput and distortion of GEMM according to user-specified constraints for: (i) percentile throughput increase against the conventional (full precision) computation $\mathbf{R} = \mathbf{A}\,\mathbf{B}$ and (ii) the SNR of $\hat{\mathbf{R}}$ versus $\mathbf{R}$.

4.5 Distortion-Controlled Throughput Scaling of Subblock Multiplication

When attempting to accelerate GEMM with the proposed approach, it is imperative to minimise D_W for each $L \times L$ subblock multiplication of each $L \times L$ inner-kernel calculation of the $M \times K$ by $K \times N$ matrix multiplication [e.g. $\forall l : \mathbf{A}_{2,l}\mathbf{B}_{l,1}$ of $\mathbf{R}_{2,1}$ of Figure 4.1 and (4.1)] via the optimal use of companding and packing. Once the optimal configuration and minimum distortion is established for each packing W of each subblock, the configuration for companding and packing can be decided for the overall matrix multiplication.

4.5.1 Theoretical Calculation of Optimal Companders and Experimental Validation

The following proposition provides the general form of admissible companders for each $L \times L$ subblock under i.i.d. inputs.

Proposition 3 (General Form of Companders)**.** *For packed subblock multiplication $\tilde{\mathbf{A}}\tilde{\mathbf{B}}$ of i.i.d. inputs within $[-\max \mathbf{A}, \max \mathbf{A}]$ and $[-\max \mathbf{B}, \max \mathbf{B}]$ modeled by $\alpha \sim \mathrm{P}_\alpha(\sigma_\alpha)$, $\beta \sim \mathrm{P}_\beta(\sigma_\beta)$, respectively, and quantisation noise modelled by $\nu_\alpha \sim \mathrm{P}_{\nu_\alpha}(\sigma_{\nu_\alpha})$, $\nu_\beta \sim \mathrm{P}_{\nu_\beta}(\sigma_{\nu_\beta})$, the companders achieving expected SNR of $E\{S_{\text{subblock}}\}$ dB against $\mathbf{AB}$ calculated with floating-point precision are:*

$$c_{\mathbf{A}} = \frac{1}{\sqrt{2}\sigma_\alpha c_{\text{tot}}} \sqrt{D_{\text{QvR}} \pm \sqrt{D_{\text{QvR}}^2 - 4\sigma_\alpha^2 \sigma_\beta^2 c_{\text{tot}}^2}} \tag{4.24a}$$

$$c_{\text{B}} = \frac{\sqrt{2}\sigma_\alpha}{\sqrt{D_{\text{QvR}} \pm \sqrt{D_{\text{QvR}}^2 - 4\sigma_\alpha^2 \sigma_\beta^2 c_{\text{tot}}^2}}} \tag{4.24b}$$

with D_{QvR} expressing the quantisation versus representation induced distortion, given by:

$$D_{\text{QvR}} = \frac{12\sigma_\alpha^2\sigma_\beta^2}{10^{0.1\,E\{S_{\text{subblock}}\}}} + \frac{1 - 144\nu(R_{\max}, W)}{12c_{\text{tot}}^2} \tag{4.25}$$

with

$$c_{\text{tot}} = \frac{L \max \mathbf{A} \max \mathbf{B}}{R_{\max}} \tag{4.26}$$

and

$$D_{\text{QvR}} \geq 2\sigma_\alpha\sigma_\beta c_{\text{tot}}. \tag{4.27}$$

Proof. We express $c_{\mathbf{B}}$ in function of $c_{\mathbf{A}}$ from (4.14). We then link $c_{\mathbf{A}}$ with the expected SNR via (4.23), given that

$$E\left\{(\rho_{m,n}) - \hat{\rho}_{m,n})^2\right\} = \frac{L(\sigma_\alpha\sigma_\beta)^2}{10^{0.1\,E\{S_{\text{subblock}}\}}}.$$

Replacing (4.23) in the last equation and after a few straightforward algebraic manipulations, we reach:

$$\sigma_\alpha^2 c_{\text{tot}}^2 c_{\mathbf{A}}^4 + \left[\frac{1 + 144\,v(R_{\max}, W)}{12c_{\text{tot}}^2} - \frac{12\sigma_\alpha^2\sigma_\beta^2}{10^{0.1\,E\{S_{\text{subblock}}\}}}\right] c_{\text{A}}^2 + \sigma_\beta^2 = 0 \tag{4.28}$$

with c_{tot} defined by (4.26). Solving (4.28) for $c_{\mathbf{A}}^2$ provides:

$$c_{\mathbf{A}}^2 = \frac{D_{\text{QvR}} \pm \sqrt{D_{\text{QvR}}^2 - 4\sigma_\alpha^2\sigma_\beta^2 c_{\text{tot}}^2}}{2\sigma_\alpha^2 c_{\text{tot}}^2} \tag{4.29}$$

with D_{QvR} representing the quantisation-versus-representation noise, defined by (4.25). From (4.29), in order for $c_{\mathbf{A}}$ to be real, $|D_{\text{QvR}}| \geq 2\sigma_\alpha\sigma_\beta c_{\text{tot}}$ and $D_{\text{QvR}} > 0$. This leads to companders defined by (4.24). □

Given the input statistics, Proposition 3 demonstrates that various pairs of companders provide for $E\{S_{\text{subblock}}\}$ dB as long as they lead to $R_{\max}$ [via (4.14)] that satisfies (4.27). This complicates the selection process for the operational parameters since, per $L \times L$ subblock of the matrix multiplication, one must select:

1. the number of packings used (W),
2. the desired value of $R_{\max}$ in order to find $v(R_{\max}, W)$ for this choice of packed processing via Figure 4.3,
3. the desired value for $E\{S_{\text{subblock}}\}$ for the particular subblock [leading to D_{QvR} that satisfies (4.27)], and
4. the specific set of companders from the permissible options of (4.24).

Fortunately, in the following we show that, for each subblock and each choice of packing, $W \in \{2, 3, 4\}$, there exists a unique value for each of: $R_{\max}$, $c_{\mathbf{A}}$, and $c_{\mathbf{B}}$, which maximises the expected SNR $E\{S_{\text{subblock}}\}$. An expression for the obtained (maximum) value of $E\{S_{\text{subblock}}\}$ for each W is also provided. This facilitates the parameter selection to a great extent as there is a unique (optimal) parameter configuration for each packing W of each subblock.

Proposition 4 (Minimum-error Companders)**.** *Under the settings of Proposition 3 for subblock multiplication, for each packing $W > 1$, the companders providing the* *maximum expected SNR of* $E\left\{S^*_{\text{subblock}}\right\}$ dB *are:*

$$\begin{aligned} c^*_{\mathbf{A}} &= \sqrt{\frac{\sigma_\beta}{\sigma_\alpha c_{\text{tot}}}} \\ c^*_{\mathbf{B}} &= \sqrt{\frac{\sigma_\alpha}{\sigma_\beta c_{\text{tot}}}} \end{aligned} \tag{4.30}$$

with

$$\{R^*_{\max}, E\{S^*_{\text{subblock}}\}\}_W = \arg\max_{\forall R_{\max}} \left\{ E\{S_{\text{subblock}}\} = -10\log_{10}\left[\frac{144\,v(R_{\max},W)+1}{144\,\sigma_\alpha^2\sigma_\beta^2 c_{\text{tot}}^2} + \frac{c_{\text{tot}}}{6\sigma_\alpha\sigma_\beta}\right]\right\} \tag{4.31}$$

for every $W > 1$ *and* c_{tot} *given by* (4.26).

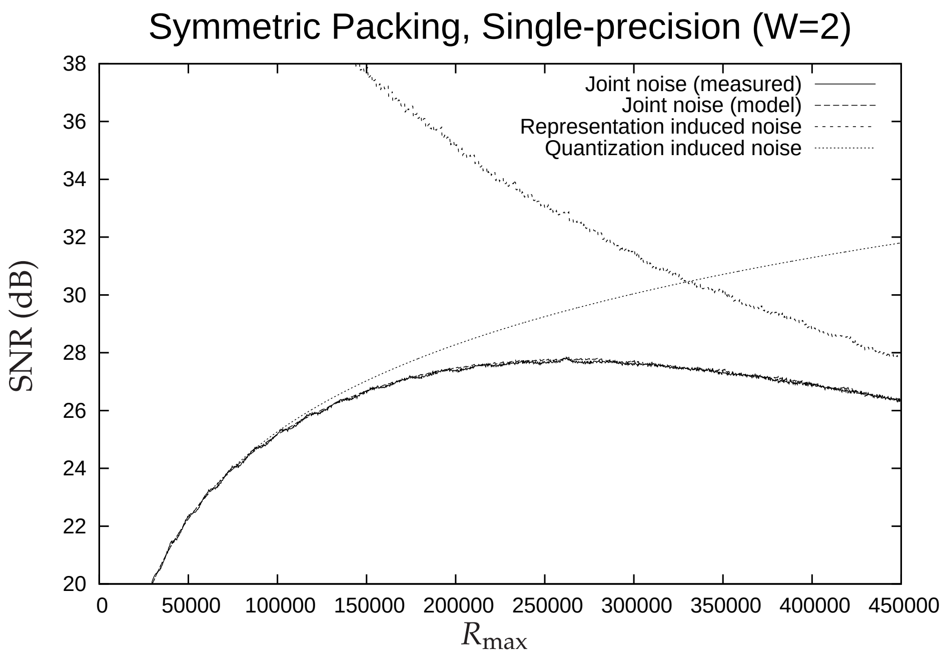
Proof. The term D_{QvR} of (4.25) expresses the quantisation versus the representation noise. In particular, when quantisation is refined via the use of larger $R_{\max}$, the first term of (4.25), $12\sigma_\alpha^2\sigma_\beta^2 10^{-0.1\,E\{S_{\text{subblock}}\}}$, is monotonically decreasing as $E\left\{S_{\text{subblock}}\right\}$ (obtained SNR) is monotonically increasing. However, the second part of (4.25) is monotonically increasing for increased $R_{\max}$, since c_{tot}^{-2} is proportional to $R^2_{\max}$ and $v(R_{\max}, W)$ is monotonically increasing with $R_{\max}$ [something that was also verified experimentally in Figure 4.3]. Hence, the optimal point is found at the value of $R_{\max}$ for which $D_{\text{QvR}} = 2\sigma_\alpha\sigma_\beta c_{\text{tot}}$, i.e. the maximum possible SNR is obtained with companders that remain marginally admissible by (4.27). This condition leads to the companders shown in (4.30). Furthermore, under this condition, solving (4.26) for $E\left\{S_{\text{subblock}}\right\}$ derives the form for $E\left\{S_{\text{subblock}}\right\}$ given in (4.31). Since this expression depends on $v(R_{\max}, W)$, for which no clear analytic model exists, we can solve this equation numerically to derive the optimal (maximum) value for $E\left\{S_{\text{subblock}}\right\}$ and $R_{\max}$ under the input data statistics and the given packing W. □

Besides the simple form of the optimal companders, it may seem surprising that such a unique solution should exist in the first place. This can be explained as follows. For each value of W, increasing $R_{\max}$ initially leads to decreased error due to the decrease of quantisation error. However, after a certain point, the representation induced error, $v(R_{\max}, W)$, begins to rise exponentially, as seen in Figure 4.3. This counteracts the quantisation error reduction.

We demonstrate indicative experimental versus theoretical (model) curves of $E\left\{S_{\text{subblock}}\right\}$ in Figure 4.5. Per $R_{\max}$ value, the experimental curves are produced by measuring S_{subblock} numerically from the output of multiple runs under the experimental settings of Subsection 4.4.2 but this time using floating-point inputs instead of integers, and companders set via (4.30). For the theoretical calculation for each $R_{\max}$ value, we use the expression of (4.31) for $E\left\{S_{\text{subblock}}\right\}$ with $v(R_{\max}, W)$ taken from the results of Figure 4.3. The very good agreement between the theoretical and experimental results validates both the independence assumption between quantisation-induced and representation induced error, as well as the accuracy of the proposed SNR estimation model of (4.31). Finally, the unique maximum SNR observed in the results of Figure 4.5 shows that, under the conditions of Proposition 3, there is indeed a unique solution for companders per subblock (and per W) that minimises the produced error, which is given by (4.31) of Proposition 4.

4.5.2 Practical GEMM Configuration for Optimised Throughput/Distortion Processing

Proposition 4 simplifies the optimum selection of operational settings for the overall matrix multiplication under throughput/distortion constraints. This is achieved by first computing $\max \mathbf{A}, \max \mathbf{B}, \sigma_\alpha, \sigma_\beta$ during the subblock data accesses for reordering to block-major format. Subsequently, for every $L \times L$ subblock multiplication of every $L \times L$ inner-kernel of the overall $(M \times K) \times (K \times N)$ matrix multiplication, $c^*_{\mathbf{A}}$, $c^*_{\mathbf{B}}$, $\left\{R^*_{\max}, E\left\{S^*_{\text{subblock}}\right\}\right\}$ are computed via Proposition 4 for every packing W, $W \in \{2, 3, 4\}$. For each W, the expected percentile throughput scaling, F_W, versus the plain processing ($W = 1$) can be calculated by off-line experiments on the target platform since depends only



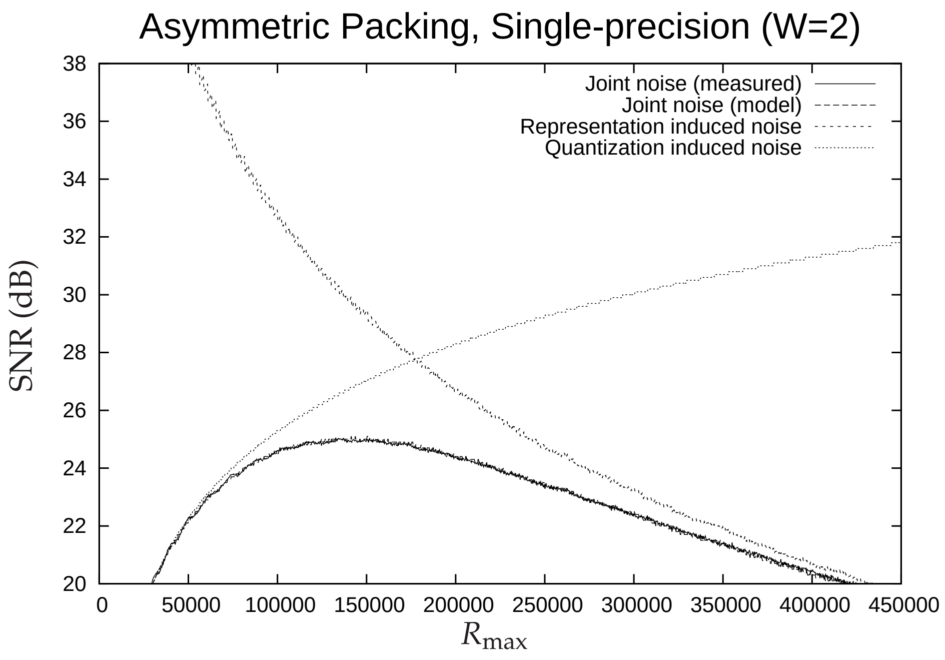


Figure 4.5: Indicative experiments for experimentally-obtained output SNR versus the expected SNR by Proposition 4 under the use of the optimal companders per $R_{\max}$ value.

on the subblock size and the implementation of the inner-kernel processing. These optimal parameters for each packing configuration W of each subblock, along with F_W, are kept in a data structure in order to prune out the best possible combination of subblock multiplication options according to distortion or throughput constraints, as discussed in the next section. The only complex part of this process is obtaining the numerical solution of (4.31) per subblock. However, Figure 4.5 shows that obtaining the exact solution for $R^*_{\max}$ is not of critical importance, since there is a wide range of $R_{\max}$ values attaining near maximum SNR per subblock. Hence, we pre-compute an approximation for all possible solutions of (4.31) offline, by using a representative set $\mathcal{B}_{\text{offline}}$ of cardinality $|\mathcal{B}_{\text{offline}}|$, consisting of input block standard deviations expected from each application, i.e.

$$\begin{array}{c} \{\sigma_\alpha, \sigma_\beta\}|_{b_{\text{offline}}} \in \mathcal{B}_{\text{offline}} \\ 1 \leq b_{\text{offline}} \leq |\mathcal{B}_{\text{offline}}| \end{array}. \tag{4.32}$$

We keep the corresponding solutions, $\forall b_{\text{offline}} : \left\{R^*_{\max}, E\left\{S^*_{\text{subblock}}\right\}\right\}|_{b_{\text{offline}}}$, obtained by (4.31) in a data structure. During the GEMM operation, for each pair of subblocks $\mathbf{A}_{i,l}$ and $\mathbf{B}_{l,j}$ $(0 \leq l < K/L)$ with standard deviations $\sigma_{\alpha,i,l}$ and $\sigma_{\beta,l,j}$, once the actual values for $\{\sigma_{\alpha,i,l}, \sigma_{\beta,l,j}\}$ are found during the block major format reordering, the solution $\forall W : \left\{R^*_{\max}, E\left\{S^*_{\text{subblock}}\right\}\right\}|_{b^*_{\text{offline}}}$ corresponding to the closest element within $\mathcal{B}_{\text{offline}}$ is used, i.e.:

$$b^*_{\text{offline}} = \arg \min_{1 \leq b_{\text{offline}} \leq |\mathcal{B}_{\text{offline}}|} \left\| \{\sigma_{\alpha,i,l}, \sigma_{\beta,l,j}\} - \{\sigma_\alpha, \sigma_\beta\}|_{b_{\text{offline}}} \right\|_F^2 \tag{4.33}$$

Naturally, since the set of input values contains only certain values out of the possible combinations of the two standard deviations, the produced GEMM configuration becomes slightly less accurate in comparison to the theoretical maximum. For values of σ_α,σ_β selected with step equal to 20% of their maximum possible range for each application, this loss of accuracy was found to be negligible and (4.33) requires minimal effort.

4.6 Concluding Remarks

In the introduction of this chapter, we presented what the limitations of the approach presented in Chapter 2 are. Even though this proposed approach brought to positive experimental results (presented in Chapter 3), bounding this approach into the family of integer-to-integer linear processing is a huge constraint for many applications.

This chapter addresses this important problem, proposing an extension of the theory presented in Chapter 2: using a technique that pushes quantised data (by companding and rouding, as presented in the Subsection 4.2) inside the packing module, multiple throughput/distortion trade-offs can be obtained and input values can be floating-point.

However, to fully exploit the capability of this new quantised approach, packing theory has been extended to obtain the packed format for both the operands of the performed calculation. This responds to the second limitation envisaged in the introduction of this chapter.

Next chapter will further extend this newly revised theory with a real world example: a GEMM subprogram implementation. Moreover, to compete with state-of-the art implementations of GEMM, a reference implementation based on SSE instructions will be presented. This implementation will be also used inside real world application that require GEMM: a face recognition system and a neural network.

Chapter 5

BLAS and Generic Matrix Multiplication

Basic Linear Algebra Subprograms (BLAS) is a *de facto* application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication. First published in 1979 as a set of FORTRAN routines [113, 117], they were written to address common matrix manipulation problems and, during the years, they became a standard choice for all the applications that need matrix calculation and they were used to build larger packages, such as LAPACK, Matlab and others. These routines are heavily used in high-performance computing and highly optimised implementations of the BLAS interface have been developed by hardware vendors through the years, such as Intel MKL (Math Kernel Library), ACML (AMD Core Math Library), Apple Accelerate and many others, as well as by other authors, such as Goto BLAS and ATLAS [14, 15]. At the same, multiple C or C++ wrappers (and other languages) have been developed around the original FORTRAN interface.

The Generic Matrix Multiply routine is the core element of high-performance linear algebra libraries used in many computationally-demanding DSP operations, such as covariance scatter matrix calculation [13], noise cancellation [118], back-propagation learning [119], and two-dimensional (2D) transform analysis and synthesis of large data sets [120]. Optimised realisations of such operations in programmable processors are based on BLAS [14, 15], which is tailored to the particular hardware via the use of assembly kernels or SIMD operations [121].

As discussed in Chapter 4, single and double-precision floating-point matrix multiplication are realised in BLAS by the *Generic Matrix Multiply* (GEMM) routines. Since most BLAS functions can be rewritten to use GEMM as the dominant operation as the problem size scales [114], GEMM throughput measurements have traditionally been considered important enough to form a core part of processor benchmarking efforts.

BLAS functionalities are divided into three levels: 1, 2 and 3. BLAS Level 1 contains *vector operations* of the form $y \leftarrow \alpha\mathbf{x} + \mathbf{y}$ as well as scalar dot products and vector norms. BLAS Level 2 contains matrix-vector operations of the form $\mathbf{y} \leftarrow \alpha\mathbf{A}\,\mathbf{x} + \beta\mathbf{y}$ as well as solving $\mathbf{T}\,\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ with $\mathbf{T}$ being triangular. BLAS Level 3 [122] contains *matrix-matrix operations* of the form $\mathbf{C} \leftarrow \alpha\mathbf{A}\,\mathbf{B} + \beta\mathbf{C}$ as well as solving $\mathbf{B} \leftarrow \alpha\mathbf{T}^{-1}\,\mathbf{B}$ for triangular matrices $\mathbf{T}$. This level contains the GEMM routine.

5.1 Generic Matrix Multiply Routine (GEMM)

Matrix multiply is written in terms of a lower-level building block, usually called *kernel*. Kernels perform matrix multiplication with fixed input dimension, e.g.: $M = N = K = L$, where the block size L is chosen in order to maximise L1 cache reuse. Matrices are taken as input in *row-major storage* and then reordered into *block-major* storage. For example, if the input matrix is defined as:

$$\begin{bmatrix} 10 & 23 & 4 & 5 & 6 & 20 \\ 7 & 4 & 23 & 1 & 7 & 23 \\ 10 & 22 & 31 & 10 & 7 & 6 \\ 2 & 9 & 4 & 54 & 32 & 1 \\ 0 & 9 & 43 & 2 & 1 & 3 \\ 0 & 7 & 23 & 1 & 3 & 4 \end{bmatrix}$$

and $L = 3$, the row-major storage will be:

$$[10\ 23\ 4\ 5\ 6\ 20\ 7\ 4\ 23\ 1\ 7\ 23\ 10\ 22\ 31\ 10\ 7\ 6\ 2\ 9\ 4\ 54\ 32\ 1\ 0\ 9\ 43\ 2\ 1\ 3\ 0\ 7\ 23\ 1\ 3\ 4]$$

while the block-major storage will be:

$$[10\ 23\ 4\ 7\ 4\ 23\ 10\ 22\ 31\ 5\ 6\ 20\ 1\ 7\ 23\ 10\ 7\ 6\ 2\ 9\ 4\ 0\ 9\ 43\ 0\ 7\ 23\ 54\ 32\ 1\ 2\ 1\ 3\ 1\ 3\ 4]$$

In block-major storage, the $L \times L$ blocks operated on by the kernel are actually contiguous in memory. This optimisation prevents unnecessary cache misses, cache conflicts and TLB problems [14, 15, 72].

5.1.1 Streaming SIMD Extensions and High Performance Computing

Streaming SIMD Extension (SSE) is a SIMD instruction set designed by Intel and introduced in 1999. SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers. SSE was subsequently expanded by Intel to SSE2 (extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers [123]), SSE3 (provides new instructions that can accelerate application performance in specific areas, such as video processing, complex arithmetic operations, and thread synchronisation and complements SSE and SSE2 with instructions that process data asymmetrically and facilitate horizontal computation [123]), SSSE3 (provides additional enhancement for SIMD computation with new instructions on digital video and signal processing [123]), SSE4 (targeted to improve the performance of media, imaging, and 3D workloads [124]) and SSE4.1 (adds instructions that improve compiler vectorisation [124]).

SSE instructions can greatly increase performance when the same operations are to be performed on multiple data, which often happens in digital signal processing and graphic processing, as well as in linear algebra routines (like GEMM). Figure 5.1 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

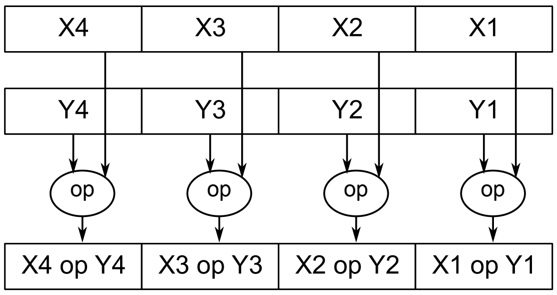


Figure 5.1: Typical SIMD Operations.

5.1.2 Matrix Reordering

Matrix reordering is performed using a technique called "loop blocking" [123, 125], a useful technique for memory performance optimisation. Loop blocking obtains blocks to be operated on by the kernel that are actually contiguous in memory. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain of a given problem into smaller contiguous portions rather than sequentially traversing through the entire memory domain. Each portion should be small enough to fit all the data for a given computation into the cache, thereby maximising data reuse.

Listing 5.1 shows a basic implementation of the matrix reordering algorithm, which is illustrated in Figure 5.2.

It is important to note that, given the different role of **A** and **B** in the matrix multiply, reordering algorithm swaps the lines and rows accordingly, as depicted in Figure 5.2. We also note that in the asymmetric packing approach

```
// block reording of A
int idx = 0;
for (int oi = 0; oi < M; oi+=KERNEL_SIZE)
  for (int oj = 0; oj < K; oj+=KERNEL_SIZE)
    for (int ai = 0; ai < KERNEL_SIZE; ai++)
      for (int aj = 0; aj < KERNEL_SIZE; aj++)
      {
        At[idx] = A[(oi+ai)*K + oj + aj];
        idx++;
      }

// block reordering and transposition of B
idx = 0;
for (int oj = 0; oj < N; oj+=KERNEL_SIZE)
  for (int oi = 0; oi < K; oi+=KERNEL_SIZE)
    for (int bj = 0; bj < KERNEL_SIZE; bj++)
      for (int bi = 0; bi < KERNEL_SIZE; bi++)
      {
        Bt[idx] = B[(oi + bi)*N + oj + bj];
        idx++;
      }
```

Listing 5.1: Basic reordering algorithms

(Subsection 4.3.1), either $\tilde{\mathbf{A}}$ or $\tilde{\mathbf{B}}$ can be packed after matrix reordering has been performed: this selection does not affect the performance in the case of GEMM as both blocks have been reordered in block major format.

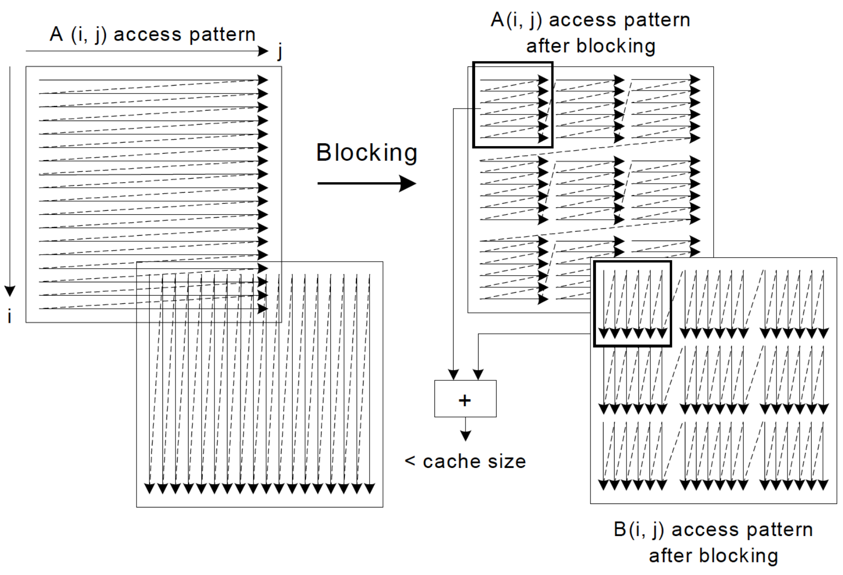

A (i, j) access pattern
j
i
Blocking
A(i, j) access pattern after blocking
+
< cache size
B(i, j) access pattern after blocking

Figure 5.2: Matrix reordering using loop blocking.

5.1.3 Top-Level of GEMM

In Section 4.1 we have seen that *inner-kernel* result $\mathbf{R}_{2,1}$ of the example shown in Figure 4.1a comprises multiple subblock multiplications $\mathbf{A}_{2,l}\mathbf{B}_{l,1}$, as formulated in (4.1).

A code snippets that shows how this algorithm can be implemented is presented in Listing 5.2. This implementation relies on the fact that the input matrices have been reordered in block-major format (as shown in Subsection 5.1.2), so that blocks are contiguous in memory, in order to minimise the number of cache misses.

```
const int M_LOOPS = M/KERNEL_SIZE;
const int N_LOOPS = N/KERNEL_SIZE;
const int K_LOOPS = K/KERNEL_SIZE;

float *pA, *pB, *pC;

for (int ii = 0; ii < M_LOOPS; ii++)  // blocks of A
{
  for (int jj = 0; jj < N_LOOPS; jj++)  // blocks of B
  {
    pC = &C[ii*N_LOOPS*KERNEL_ELEMS + jj*KERNEL_SIZE];

    pA = &At[(ii*K_LOOPS)*KERNEL_ELEMS];
    pB = &Bt[(jj*K_LOOPS)*KERNEL_ELEMS];

    KernelGemm(pA, pB, 0, pC, N);

    for (int kk = 1; kk < K_LOOPS; kk++)
    {
      pA = &At[(ii*K_LOOPS + kk)*KERNEL_ELEMS];
      pB = &Bt[(jj*K_LOOPS + kk)*KERNEL_ELEMS];

      KernelGemm(pA, pB, 1, pC, N);
    }
  }
}
```

Listing 5.2: Basic GEMM structure

5.1.4 GEMM Kernels and Optimisation Techniques

Different optimisation techniques can be used in the implementation of the inner-kernel, usually tailoring the implementation for a specific architecture.

In fact, this is the portion of the GEMM subroutine where most of the low-level optimisation is made. These optimisation can be divided in two groups: low-level (SSE or assembly) instructions and instruction reordering, which also includes *loop unrolling techniques*.

The inner-kernel implementation used for the experiments in Section 5.2 and Section 5.3 uses a synergy of SSE instructions and loop unrolling, in order to obtain the maximum parallelism achievable using XMM registers. Listing 5.3 shows a simplified version of this kernel. This particular implementation uses *Intel SSE Intrinsic* (C calls that maps one-to-one with assembler instructions) in order to exploit the maximum parallelism from the architecture. At the same time, loops are unrolled in order to fill completely every XMM register during every phase of the execution: four rows of **B** (index `jj` is then incremented by four every loop) are multiplied against two rows **A** (index `ii` is then incremented by two every loop), in order to fill completely two register that will be stored inside **C**. Index `kk` is incremented by four every loop because every 128-bit SSE register holds four single-precision floating point samples.

```
void KernelGemm(const float* A,
                const float* B,
                const float beta,
                float* C,
                const int ldc)
{
  // M = N = K = KERNEL_SIZE
  __m128 a, b0, b1, b2, b3, B0, B1, B2, B3;
  __m128 c00, c01, c02, c03, c10, c11, c12, c13;

  const __m128 BETA = _mm_set1_ps(beta);

  for (int ii = 0; ii < KERNEL_SIZE; ii+=2)  // rows of A
  {
    for (int jj = 0; jj < KERNEL_SIZE; jj+=4)  // cols of B
    {
      b0 = _mm_load_ps(&B[jj*KERNEL_SIZE]);
      b1 = _mm_load_ps(&B[(jj+1)*KERNEL_SIZE]);
      b2 = _mm_load_ps(&B[(jj+2)*KERNEL_SIZE]);
      b3 = _mm_load_ps(&B[(jj+3)*KERNEL_SIZE]);

      a = _mm_load_ps(&A[ii*KERNEL_SIZE]);

      c00 = _mm_mul_ps(b0, a);
      c01 = _mm_mul_ps(b1, a);
      c02 = _mm_mul_ps(b2, a);
      c03 = _mm_mul_ps(b3, a);

      a = _mm_load_ps(&A[(ii+1)*KERNEL_SIZE]);
      c10 = _mm_mul_ps(b0, a);
      c11 = _mm_mul_ps(b1, a);
      c12 = _mm_mul_ps(b2, a);
      c13 = _mm_mul_ps(b3, a);

      for (int kk = 4; kk < KERNEL_SIZE; kk+=4)
      {
        b0 = _mm_load_ps(&B[jj*KERNEL_SIZE + kk]);
        b1 = _mm_load_ps(&B[(jj+1)*KERNEL_SIZE + kk]);
        b2 = _mm_load_ps(&B[(jj+2)*KERNEL_SIZE + kk]);
        b3 = _mm_load_ps(&B[(jj+3)*KERNEL_SIZE + kk]);

        a = _mm_load_ps(&A[ii*KERNEL_SIZE + kk]);

        B0 = b0;
        B0 = _mm_mul_ps(B0, a);    // a = a*b0
        c00 = _mm_add_ps(c00, B0);  // c = c + a*b0
        B1 = b1;
        B1 = _mm_mul_ps(B1, a);
        c01 = _mm_add_ps(c01, B1);
        B2 = b2;
        B2 = _mm_mul_ps(B2, a);
        c02 = _mm_add_ps(c02, B2);
        B3 = b3;
        B3 = _mm_mul_ps(B3, a);
        c03 = _mm_add_ps(c03, B3);

        a = _mm_load_ps(&A[(ii+1)*KERNEL_SIZE + kk]);

        b0 = _mm_mul_ps(b0, a);    // a = a*b0
        c10 = _mm_add_ps(c10, b0);  // c = c + a*b0
        b1 = _mm_mul_ps(b1, a);
        c11 = _mm_add_ps(c11, b1);
        b2 = _mm_mul_ps(b2, a);
        c12 = _mm_add_ps(c12, b2);
        b3 = _mm_mul_ps(b3, a);
        c13 = _mm_add_ps(c13, b3);
      }

      c00 = _mm_hadd_ps(c00, c01);
      c02 = _mm_hadd_ps(c02, c03);
      c00 = _mm_hadd_ps(c00, c02);

      c03 = _mm_load_ps(&C[ii*ldc + jj]);
      c03 = _mm_mul_ps(c03, BETA);
      c00 = _mm_add_ps(c00, c03);
      _mm_store_ps(&C[ii*ldc + jj], c00);

      c10 = _mm_hadd_ps(c10, c11);
      c12 = _mm_hadd_ps(c12, c13);
      c10 = _mm_hadd_ps(c10, c12);

      c13 = _mm_load_ps(&C[(ii+1)*ldc + jj]);
      c13 = _mm_mul_ps(c13, BETA);
      c10 = _mm_add_ps(c10, c13);
      _mm_store_ps(&C[(ii+1)*ldc + jj], c10);
    }
  }
}
```

Listing 5.3: GEMM Inner-kernel

5.1.5 Throughput/distortion Optimisation of Inner-Kernel Multiplication

We propose throughput/distortion acceleration, controlled at the subblock level of the overall matrix multiplication. Consider the example of the inner-kernel result $\mathbf{R}_{2,1}$ shown in Figure 4.1a and computed by (4.1). As elaborated in the previous section, for each individual subblock multiplication of (4.1), i.e. $\mathbf{A}_{2,l}\ \mathbf{B}_{l,1}$ with $0 \leq l < \frac{K}{L}$, we have precomputed parameters: $\forall W, l : \left\{R^*_{\max}, E\left\{S^*_{\text{subblock}}\right\}\right\}\big|_{b^*_{\text{offline}}}$ and F_W, and for each packing we can compute $c^*_{\mathbf{A}}$,$c^*_{\mathbf{B}}$ by (4.30) at runtime [and from that we can readjust S^*_{subblock} via the expression of (4.31) for additional accuracy]. The aim is to select (per subblock) the best possible packing, W, along with the best possible parameters, such that the resulting inner-kernel, $\mathrm{R}_{2,1}$, is computed with:

1. the highest percentile acceleration possible under an SNR constraint, or
2. the highest SNR possible under a percentile acceleration constraint.

Controlling the throughput/distortion optimisation process at the inner-kernel level allows for flexibility within practical applications. For example, if matrix multiplication is used for transform decomposition, some inner-kernels of the resulting matrix corresponding to transform coefficients that are required at the smallest possible distortion can be computed with $W = 1$, i.e. with native floating point accuracy of `sGEMM` or `dGEMM`, while others can be accelerated via the use of $W = \{2, 3, 4\}$ and produce approximate results. The application must only specify the required SNR $S_{\text{kernel}}(i, j)$ per $L \times L$ inner-kernel $\mathrm{R}_{i,j}$ against the result at the native precision of GEMM, or the required percentile acceleration, $F_{\text{kernel}}(i, j)$, in comparison to computing with $W = 1$ and the pruning process described in the following will derive the appropriate settings per subblock multiplication in order to meet this specification. For each inner kernel $\mathbf{R}_{i,j}$, $S_{\text{kernel}}(i, j)$ is converted to MSE by $D_{\text{kernel}}(i, j) = 10^{-0.1 S_{\text{kernel}}(i,j)}\, L \sum_{l=0}^{K/(L-1)} (\sigma_{\alpha,i,l}\sigma_{\beta,l,j})^2$, with $\sigma_{\alpha,i,l}$ and $\sigma_{\beta,l,j}$ the standard deviation of $\mathbf{A}_{i,l}$ and $\mathbf{B}_{l,j}$, respectively.

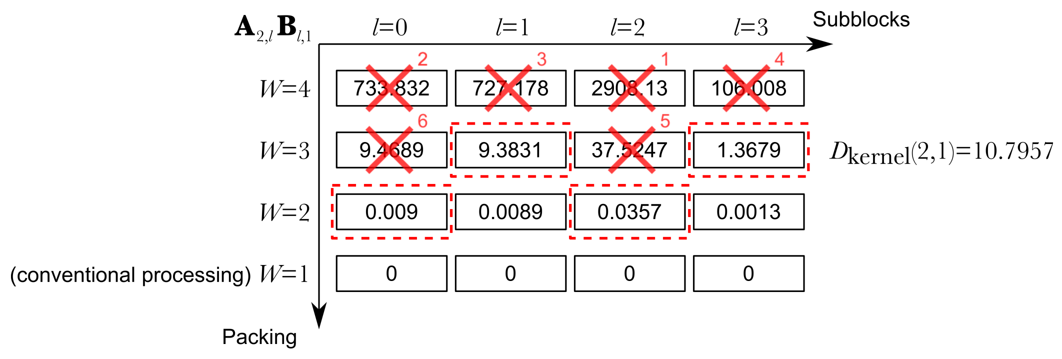


Figure 5.3: Example distortion (mean squared error (MSE)) for each of the subblocks of (4.1) with $K/L = 4$ and pruning steps enumerated until $D_{\text{kernel}}(2,1) \leq 11.0$ is achieved. The dashed rectangles indicate the final packing selection for each subblock.

The utilised pruning is a top-down approach where, starting from the maximum acceleration, the selection is pruned by removing the outcome with the highest distortion, until the distortion constraint, $D_{\text{kernel}}(i,j)$, or the percentile throughput acceleration constraint, $F_{\text{kernel}}(i,j)$, is met. This is illustrated for distortion-constrained processing of inner-kernel $\mathbf{R}_{2,1}$ in the example of Figure 5.3, where we set $S_{\text{kernel}}(i,j)$ corresponding to $D_{\text{kernel}}(2,1) = 11.0$ and we assume $\frac{K}{L} = 4$. For every calculation $\mathbf{A}_{2,l}\mathbf{B}_{l,1}$, $0 \leq l < 4$, the algorithm starts from $W = 4$ and successively removes the subblock result with the highest distortion (the removal steps are enumerated in the figure), until the distortion (or throughput) constraint is met.

The resulting operational settings utilise the maximum packing possible (i.e. offer the maximum acceleration) and provide the minimum distortion possible under the proposed framework. These settings are guaranteed to provide the best solution under iid statistics because distortion decreases monotonically with increased packing due to the required increase of companding coefficients.

5.2 Experimental Results

The proposed GEMM has been implemented in its entirety using SSE3 in an Intel Core 2 Duo P8800 processor operating at $c_{\mathrm{freq}} = 2.66\,\mathrm{GHz}$ (to ensure maximum performance, single-threaded execution, CPU throttling disabled and `gcc4.4.1 -O3 -march=native -fomit-frame-pointer`). In our experiments we selected: $W \in 1, 2$ for single precision floating point; $W \in 1, 2, 3, 4$ for double precision; and $L = 288$ as a representative inner-kernel size.

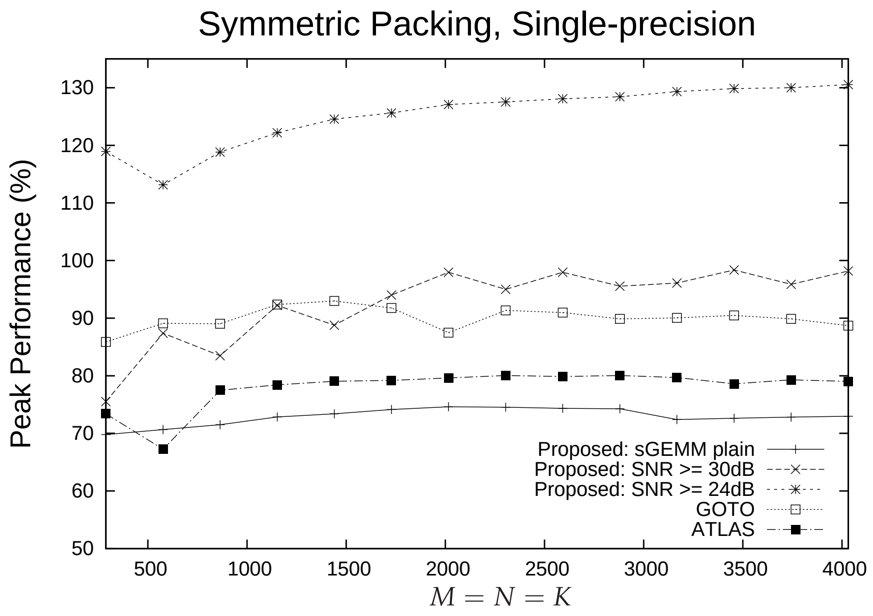
In our implementation, the top-level processing of GEMM follows the well-known reordering techniques of other optimised packages [14]. For the generic experiments of this section, we set input matrices **A** and **B** to contain uniformly distributed floating-point inputs selected from $[-\max_e, \max_e]$ within subblocks of 288×288, with $\max_e$ selected randomly for each subblock from the set $4.0, 5.0, 6.0, \ldots, 2048.0$. By disabling the processor throttling and running the proposed approach in maximum priority, we set various SNR requirements $S_{\mathrm{kernel}}(i, j)$ for each inner-kernel processing and obtain the results for $M = K = N$ shown in Figure 5.4 and Figure 5.5. As an external comparison, we provide the performance of the utilised kernel without the proposed approach ("sGEMM plain") as well as the performance of the state-of-the-art ATLAS [14] and GOTO packages [15]. With the selected input dimensions, all packages avoid "cleanup" code for the borders of the matrix multiplication.

The results reported in Figure 5.4 and Figure 5.5 show that the proposed approach provides for processing throughput that changes according to the required SNR value and it can in fact exceed 130% and 175% of the peak performance for single-precision and double-precision representation, respectively. The results of Figure 5.4 and Figure 5.5 show that exceeding 100% of peak performance is indeed possible in practice, due to the utilised companding and packing. Notice that the companders for each inner kernel are found at run-time and the performance figures reported in Figure 5.4 and Figure 5.5 for our approach include the entire process and the control framework described in Chapter 4 and Subsection 5.1.5. Finally, we validated that, even when the SNR setting leads to $W = 1$ for all inner-kernel processing, no loss in performance is observed against the conventional "sGEMM plain" and "dGEMM plain" approaches.

Beyond distortion-controlled execution, we present an example of throughput-controlled acceleration in Figure 5.6 for the case of sGEMM, where $W \in \{1, 2\}$. The 11 points reported in the figure were obtained for size $M = N = K = 4032$ matrix multiplication by increasing the percentage of accelerated inner-kernel blocks in steps of 10%: the (leftmost) maximum-SNR point (infinity) corresponds to 0% of inner-kernel blocks accelerated (i.e. $W = 1$ for all), while the (rightmost) lowest-SNR point (27.8 dB for symmetric packing and 23.9dB for asymmetric packing) corresponds to 100% of inner-kernel blocks accelerated (i.e. $W = 2$ for all). Evidently, the symmetric packing provides for lower distortion along the operational points. Goto's throughput was 19.7 GFLOPS while ATLAS achieved 18.0 GFLOPS for this example.

Overall, due to the highly optimised nature of GOTO and ATLAS, our "sGEMM plain" software kernel for matrix multiplication is 25% less efficient than GOTO. In addition, our plain software kernel is approximately 10% less efficient than ATLAS. In double precision, the equivalent loss in performance is 46% and 34%, respectively. This indicates that there is room for further improvement: if our approach were to be deployed within GOTO (or any other) high-performance kernel code, landmark performance of beyond 150% of peak performance in single precision and beyond 200% of peak performance in double-precision representation could be achieved under throughput/distortion scaling.

The gains in processing throughput can be exchanged for fault detection and correction functionalities under error-generating operating systems or processors [1, 93, 126]. Alternatively, one can reduce the operating processor frequency and still obtain comparable performance to using "sGEMM plain" (or "dGEMM plain") at a higher processor frequency, albeit at higher distortion. This approach effectively translates the throughput/distortion scaling into power/distortion. Indicative applications of the proposed framework are presented in the following section.



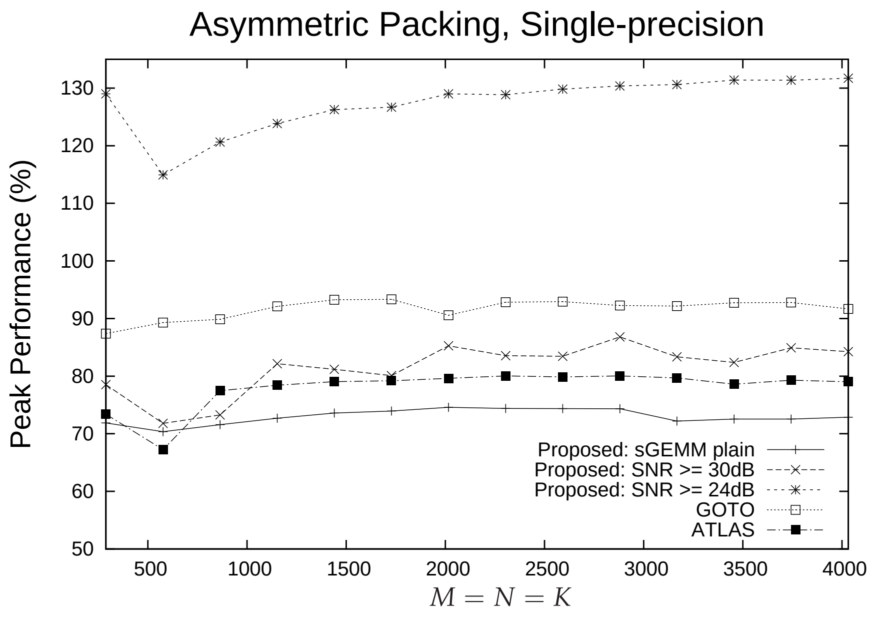
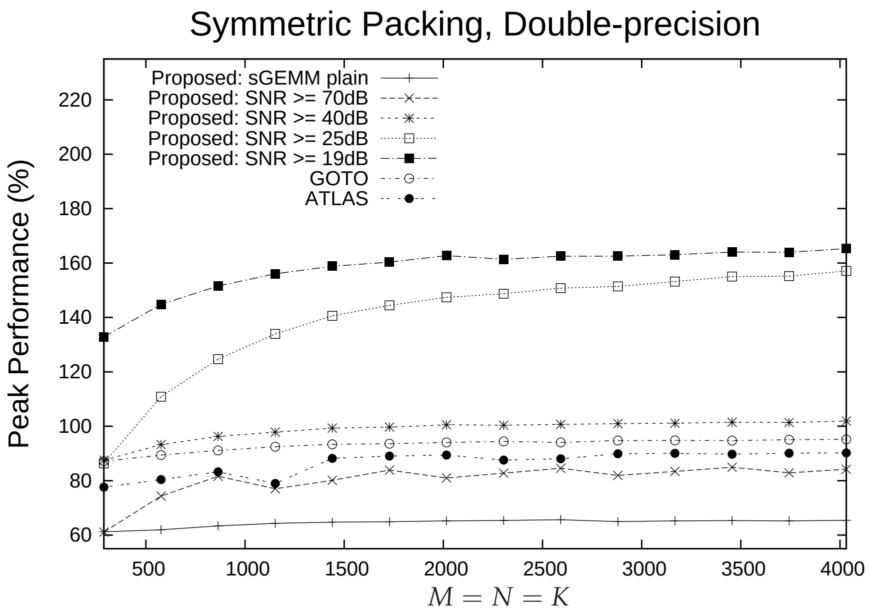


Figure 5.4: Percentage of peak performance of sGEMM (higher is better) under different distortion settings $\forall i, j : S_{kernel}(i, j)$ for the inner kernel processing; 100% of peak performance corresponds to $8 \times c_{\text{freq}} = 21.28\,$GFLOPS (giga floating-point operations per second).



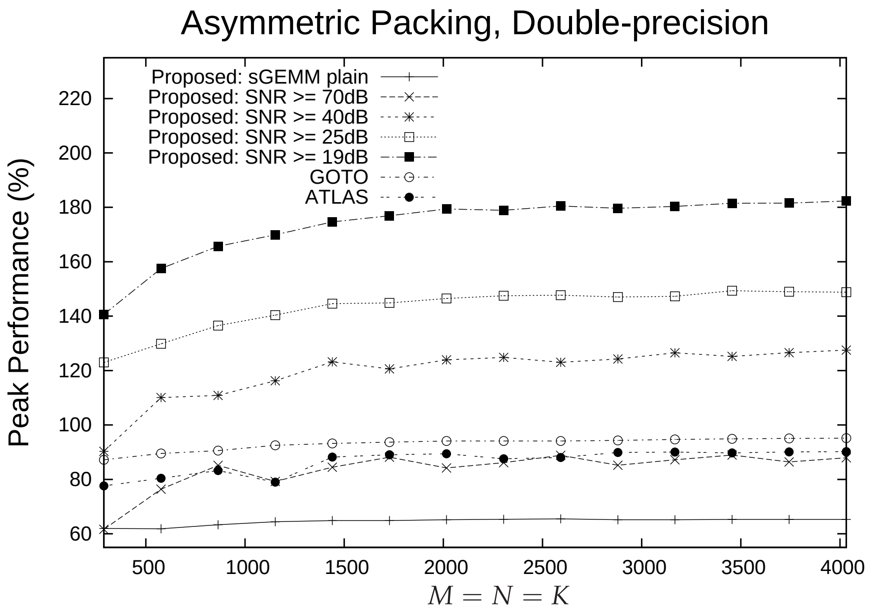


Figure 5.5: Percentage of peak performance of dGEMM (higher is better) under different distortion settings $\forall i, j : S_{kernel}(i, j)$ for the inner kernel processing; 100% of peak performance corresponds to $4 \times c_{\text{freq}} = 10.64$ GFLOPS for dGEMM.

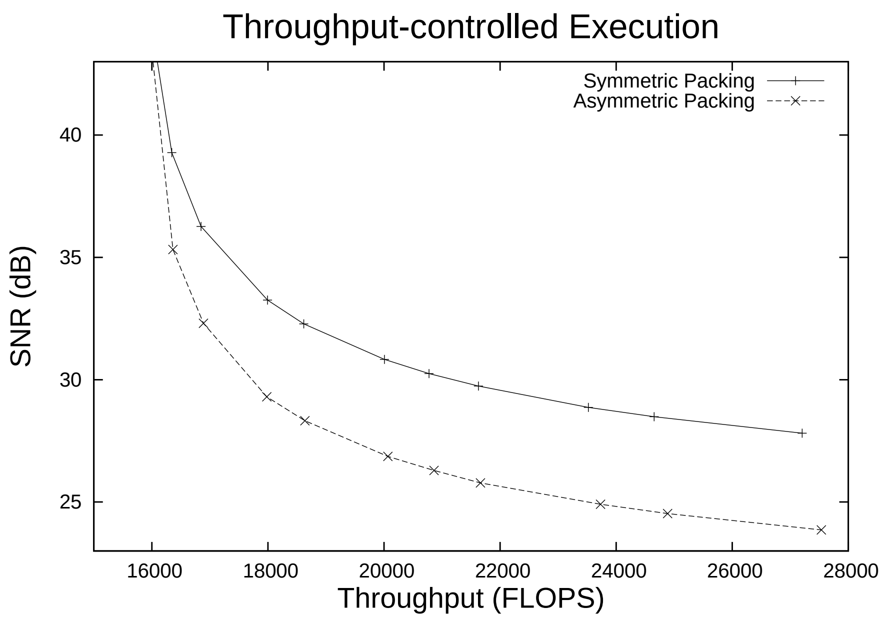


Figure 5.6: SNR vs. throughput in giga floating-point operations per second for sGEMM under acceleration control.

5.3 Applications

The proposed approach can bring important benefits to high-performance signal processing systems when the precision of computation is not of critical importance (error-tolerant systems) or when the data is intrinsically noisy. In this section, the benefits of the proposed framework is validated within three DSP applications of increasing complexity: a conceptual example of a noise cancellation system, a face recognition application based on principal component analysis (PCA), and an example of artificial neural network training for metadata learning from a large music feature database.

5.3.1 Conceptual Example: Disturbance Cancellation under Estimation Uncertainty

Assume a 576-parameter system is linearly disturbed by well-conditioned random disturbance matrix **G**, $\forall i,j \in \{0, 575\} : g_{i,j} \in [-0.5, 0.5]$. Due to computation, measurement and estimation inaccuracies, we assume that both $\mathbf{G}$ and the disturbance cancellation matrix, $\mathbf{G}^{-1}$, are corrupted by additive noise uniformly distributed within $[-0.05, 0.05]$. We examined the accuracy of disturbance cancellation, $\hat{\mathbf{I}} = \mathbf{G}^{-1}\mathbf{G}$, by calculating the MSE: $\text{MSE} = \frac{1}{576}\|\mathbf{I} - \hat{\mathbf{I}}\|_F^2$ under multiple additive noise instantiations using: (i) the proposed approach with $W = 2$ for every subblock via $c_\mathbf{A} = c_\mathbf{B} = 32$; (ii) sGEMM via "sGEMM plain" and via the GOTO software kernel. We obtained $MSE_{(i)} = 0.065$, $MSE_{(ii)} = 0.060$. Hence, even though the companders were not matching the additive noise range, similar precision is achieved, and the proposed approach is 56% more efficient "sGEMM plain" and 28% more efficient than GOTO. The throughput scaling in this case can turn into a power-scaling advantage. In particular, we can reduce the CPU frequency to 1.60 GHz (instead of the peak frequency of 2.66 GHz). As shown by the results of Figure 5.7, despite the significantly lower clock frequency, this allows for throughput comparable to "sGEMM plain" (and only 12% less efficient that GOTO clocked at 2.66 GHz). Alternatively, one can consider our approach as a way to use older processors (operating at lower frequency) to compute results with comparable throughput to newer processors. This would allow graceful degradation of hardware: older hardware continues to be used to produce results at comparable throughput to newer instantiations, albeit at lower precision, by a simple distortion or throughput parameter setting in the numerical processing library.

5.3.2 Accelerated Processing in State-of-the-art Face Recognition

State-of-the-art techniques for robust image recognition systems derive feature matrices and use 2D decomposition schemes via matrix multiplication in order to match features between a new image and an existing database of images (e.g. for automatic identification of human faces [13]. Large-scale deployments of such systems run in high-performance workstations or in cloud computing infrastructures, such as Amazon's EC2. In such deployments, it is not uncommon to expect that thousands of training and recognition tasks

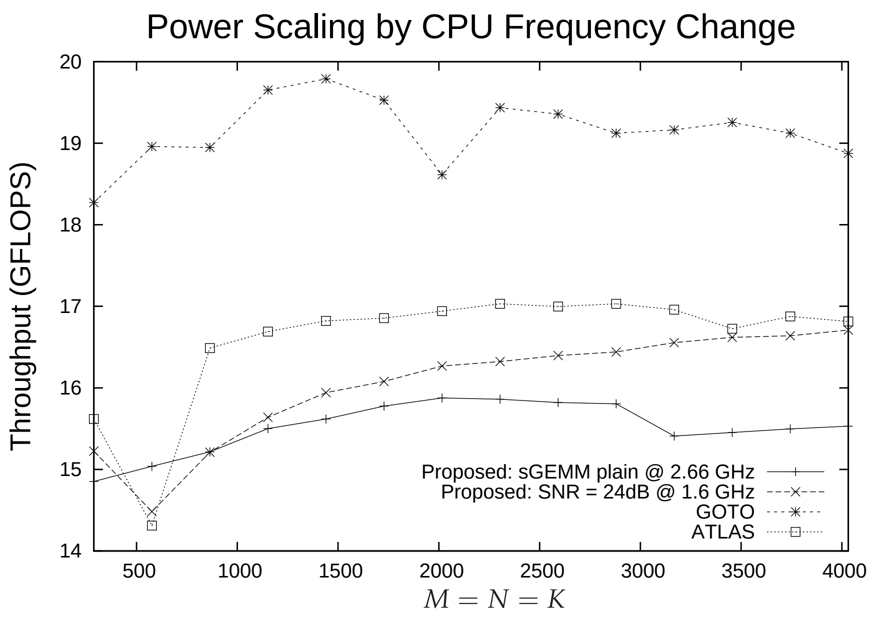


Figure 5.7: Lowering the processor clock frequency for low-power GEMM computation at comparable throughput, albeit at increased distortion.

should be computed with the highest-possible throughput/precision capability of each system in order to maximise the processors' or cloud utilisation. An example would be a real-time matching engine for human faces formed by continuous (incremental) 2D PCA training via hundreds of thousands of pictures uploaded by its users, while simultaneously handling new face matching queries. Using the proposed approach, one can accelerate the real-time training and matching process. Specifically, the accelerated GEMM can be used for the image covariance matrix calculation and for the input image projection to the feature matrix [13]. This application is suitable for approximate GEMM results as the feature-selection process is anyway approximate, since only the feature matrices corresponding to the most significant eigenvalues of the training images are selected for the matching process [13]. In the following, we provide details of such a deployment for the prominent 2D PCA system of [13] under different SNR values for the GEMM computations.

The algorithm consists of three stages: training, feature extraction and classification. The training stage uses a number of *training input images* and first calculates the image covariance scatter matrix from J_{set} zero-mean input images, $\mathbf{A}_j$, by: $\mathbf{G}_J = \sum_{j=1}^{J_{\text{set}}} \mathbf{A}_j \mathbf{A}_j^{\text{T}}$. Based on this input training set, it then calculates the projection matrix comprising a series of projection axes (eigenvectors), $\mathbf{X} = [\mathbf{x}_1 | \ldots | \mathbf{x}_d]$, with $\mathbf{x}_i$, $1 \leq i \leq d$ the orthonormal eigenvectors of $\mathbf{G}_J$ corresponding to its d largest eigenvalues [13]. Each training-set image is mapped to $\mathbf{X}$ via: $\mathbf{Y}_{set,j} = \mathbf{A}_j \mathbf{X}$.

For the feature extraction stage, each new input image $\mathbf{B}_i$ (*test image*) is projected to all $\mathbf{x}_1, \ldots, \mathbf{x}_d$, via: $\mathbf{Y}_{\text{test},i} = \mathbf{B}_i \mathbf{X}$. Finally, the classification stage determines for each test image $\mathbf{B}_i$ the test-set image j^* with the smallest distance in their projections:

$$j^*_{\mathbf{B}_i} = \arg \min_{\forall j} \|\mathbf{Y}_{\text{test},i} - \mathbf{Y}_{\text{set},j}\|_F \tag{5.1}$$

From the algorithm description, it is evident that the complexity of 2D PCA is predominantly concentrated in the matrix multiplication operations required for the construction of $\mathbf{G}_J$ and the mapping during the feature extraction, i.e. $\mathbf{Y}_{\text{test},i}$. This is because the eigenvalue decomposition required for the creation of $\mathbf{X}$ is only performed once every J_{set} training images, and fast algorithms for the quick estimation of the best match via (5.1) have been known for several years[1] [127].

To examine the effects of throughput/distortion within this application, we utilise the proposed single-precision GEMM design for the matrix multiplication operations of 2D PCA. The Yale database of images[2] was used for our experiments and, following prior work [13], each image was cropped to 288×288 pixels (that includes the face portion) and the mean value was subtracted prior to processing. Results from performing all matrix multiplication operations with reduced $S_{\text{kernel}}(0,0)$ are presented in Table 5.1. Following [13], 75 images were used for the training set and 90 images were used as test images. The table demonstrates that decreasing $S_{\text{kernel}}(0,0)$ leads to increased processing throughput with the recognition accuracy remaining equal to the one obtained with the full-precision computation. In fact, for $S_{\text{kernel}}(0,0) = 20\,\text{dB}$, we ob-

[1]Examples are early termination techniques and bounding via the Cauchy-Schwarz inequality.

[2]http://www.face-rec.org/databases/

Method	Recognition (%)	Throughput (GFLOPS)
GOTO [15]	78.4	18.27
Proposed: `sGEMM plain`	78.4	14.85
Proposed: $S_{\text{kernel}}(0,0) = 50\,dB$	78.4	16.11
Proposed: $S_{\text{kernel}}(0,0) = 20\,dB$	78.8	25.31

Table 5.1: Recognition accuracy versus requested SNR for the matrix operations of 2D PCA and versus the obtained throughput for matrix multiplication (higher throughput is better).

served a slight increase in the recognition accuracy due to the companding acting as a noise removal mechanism. Importantly, due to the flexibility of the proposed framework, we obtain the results of Table 1 without any application-specific tailoring of the computation; instead, only a simple adjustment of the distortion (or throughput) specification is required.

5.3.3 Accelerated Supervised Training of Multi-layer Perceptron (MLP) based Learning System

As a final example, we examine the benefits of the proposed approach in a large deployment of a multi-layer perceptron (MLP) based learning system [119]. MLP-based learning uses back-propagation to train a neural network to create connections between input features and outputs. We follow the OpenCV implementation of the back-propagation learning algorithm operating in bunch mode: instead of using one training pattern at a time to update the weight matrices, the design uses n_p training patterns. When $n_p = 1$, only matrix-vector operations are involved in the training algorithm. However, when $n_p > 1$, matrix multiplications are used and comprise the dominant part of the execution. In order to derive test results, we utilised the Million-song dataset, a "freely-available collection of audio features and metadata for a million contemporary popular music tracks" from Columbia University, available via the UCI Machine Learning Repository [128]. Our target was to predict the publication year of each song (between 1920-2010) based on the provided set of 90

Method	Average number of epochs	Average recognition accuracy (correct/total)	Total time for GEMM computations (hours)
Proposed: `sGEMM plain`	510240	14324/24576 (58%)	474.6
GOTO [15]	510240	14324/24576 (58%)	376.6
Proposed: `sGEMM` with $W = 2$	543460	14324/24576 (58%)	375.5

Table 5.2: Summary results of MLP algorithm. Smaller time values are better.

features per song. MLP-based approaches are appropriate for such problems as there is no clear methodology to connect song features and publication year and the hope is that the learning algorithm will discover such connections automatically.

For our experiment, the bunch size was set to $n_p = 384$ and training was repeated in groups of 24576 songs randomly chosen from the training subset of the database. Validation was done on groups of 24576 songs from the validation subset of the database [128]. The only modification performed in the MLP implementation was the replacement of the matrix multiplication with: sGEMM from the GOTO library [15], sGEMM plain, and sGEMM with $W = 2$ packings. All experiments were executed on a high-performance server comprising two quad-core processors with the Intel Xeon X5460 at 3.16 GHz. Table 5.2 reports summary results, showing that `sGEMM` with $W = 2$ packings achieves the same recognition accuracy as the conventional (full-precision) approaches. In this case, companding and rounding the inputs corresponds to quantising to 13-15 bits. While it is well known that the back-propagation algorithm is robust to quantisation [129], the quantisation noise can increase the number of epochs required for training, as shown by Table 5.2. Importantly, `sGEMM` with $W = 2$ achieves 21% execution time reduction in comparison to `sGEMM plain` (i.e. approximately 4 days less). Despite this reduction, in this case `sGEMM` with $W = 2$ does not outperform GOTO due to its highly-optimised software implementation as compared to our own `sGEMM plain` design. However, a deployment of our approach using GOTO as the utilised `sGEMM plain` software kernel would indeed benefit from the demonstrated execution time speedup.

5.4 Concluding Remarks

In Chapter 2 we presented a method for concurrent computation using the packing technique. Chapter 4 further extended this theory, trying to address the shortcomings described in the introduction of Chapter 4. Finally, Chapter 4 addresses how to perform algorithms that fall outside of the integer-to-integer calculation and explains how the way the packing is performed has an impact on the achievable speed up and the overall distortion. However, no explanation was given on how to actually implement this newly revised theory into a linear algebra routine and what kind of benefit can be measured.

This chapter addresses these final shortcomings: we presented an implementation of GEMM that uses SSE intrinsics which is competitive with the state-of-the-art implementation of GEMM, like ATLAS [14] and GOTO [15], in order to prove the throughput increase. Then we presented the achievable throughput/distortion scalability and validated the proposed distortion model of Chapter 4 against the measured distortion incurred from input data matching the model assumptions. We then used our GEMM implementation inside three well known applications (disturbance cancellation, face recognition and neural network training), demonstrating that some applications are more resilient than others to approximation in one of their components.

Chapter 6

Conclusions

The first part of this thesis presented a novel software framework for image processing tasks that processes input increments in an accelerated manner and refines previously-computed results. This framework is based on the mathematical theory of the packed processing, presented and extended in Chapter 2. Specifically our proposal was realised by the synergy of the packing theory and incremental computation of linear operations, as described in the Chapter 3. Packed processing works well when the packed samples have small dynamic range. This can be achieved by using "portions" of the input samples, which leads to incremental computation.

The resulting framework was validated in Section 3.6. By implementing this approach for transform decompositions, two dimensional convolution and cross correlation, and frame by frame block matching. The results with bitplane-based computation indicate that the proposed approach can be comparable or superior to conventional (non incremental) computation for several cases. The scheduling results (Section 3.9.2) demonstrate that, by exploiting the incremental nature of the proposed computation, the worst-case ("digital world") approach of: '*Can this image processing task be performed in X frames/second?*' changes to the best-effort ("analogue world") approach of: 'What is the achieved quality when this task is performed in X frames/second?'. Similarly, the power/distortion results (Section 3.9.3) highlight that the proposed software-based incremental computation allows for seamless prolongation of the battery life of a low-power device with a simple change of output quality level.

After exploring all the aspects related to the progressive computation for image processing tasks using packing processing, this thesis introduced the concept of “lossy packed processing” in Chapter 4. This new concept, not previously known in literature, allows the creation of on-the-fly precision/throughput trade off in many computationally-intensive algorithms. In fact, by introducing the trade-off between *Companding and Packing* (Section 4.2), this work exploits the maximum numerical precision of the floating point unit (in either single or double precision) to accelerate subsets of computations where the maximum precision is not required. In order to test this new theory in a real world scenario (as done previously for image processing tasks), the development of the theory was coupled with the implementation of an ad-hoc Generic Matrix Multiplication (GEMM) routine that uses the state-of-the-art programming techniques for high performance computing development. A detailed study was conducted on the impact of the quantisation on the precision of the computation and the packing capability, which brought to the results summarised in Subsection 4.4.2. At the same time, based on this results, it was possible to formalise a stochastic model to describe the precision/throughput trade off achievable for iid input sources (Subsection 4.4.4).

Chapter 5 showed how the theory and the practical implementation were merged together: Subsection 5.1 describes all the details of the implementation, showing simplified portions of code[1], that focus on the memory optimisation, code organisation and SSE usage. The proposed approach was then compared against state-of-the-art free BLAS (Basic Linear Algebra Subroutines) implementations (namely, ATLAS and GOTO libraries), showing the increase in processing throughput that can be achieved when different distortions (in dB) are required by the application (from full precision floating-point to noisy computation).

In order to show the potential of this kind of approach, this newly design

[1]This code is currently release in the ORIP project: `http://code.google.com/p/orip/`, `http://www.ee.ucl.ac.uk/~iandreop/ORIP.html`.

GEMM subroutine was then used within several systems where matrix multiplication plays an important role, such as matrix inversion, face recognition and MLP training. In all of them, after a careful selection of the correct trade off for the specific application, we have observed significant acceleration with minimal or no impact in the results accuracy.

The amount of throughput acceleration achievable depends strongly on the type of application that the system handles: for instance in the face recognition system, the throughput increase was higher than the one of the MLP training; in the first the images submitted to the system are inherently noisy and the quantisation behaves like a low-pass filter, removing what could be considered noise; in the later, however, because of the perturbations that the network is subject to during the training with approximate inputs, even though there is a measurable speed up in the evaluation of a single epoch, more epochs are necessary to obtain the same level of prediction accuracy. Both these results provide some initial insights on how to find the perfect balance between the imprecision of the computation and throughput acceleration for a specific algorithm.

Further studies are necessary to understand how this method can be used in fault-tolerant environments, by investigating its capability for fault detection and recovery and perhaps establishing new theoretical results.

While the initial steps for modelling the noise introduced by the approximated GEMM in function of the input source statistics have been done in Section 4.4, extending such work to non i.i.d. sources would be interesting. The model can be extended for different distribution of the input or, alternatively, this could be pursued by matching the input source model to the marginal distribution of the input data. This can be coupled with specific applications. Recent work along these lines is reported by [130] for cross-correlation in music matching systems and metadata calculation for audio sources.

Further work can also be done on the implementation side, for instance applying the proposed techniques to other domains, such as GPU-based or FPGA-based hardware designs. This is a promising avenue that can demonstrate computational scalability and acceleration by approximate processing in different domains. It is also possible that even higher acceleration could be achieved if some of the pre and post-processing tasks had hardware support (e.g. packing/unpacking).

References

[1] D. Lammers, "The era of error-tolerant computing," *IEEE Spectrum*, Nov. 2010.

[2] D. Patterson, "The trouble with multicore," *IEEE Spectrum*, Jul. 2010.

[3] M. Macedonia, "Power from the edge [apple video ipod]," *Computer*, vol. 38, no. 12, pp. 123–125, Dec. 2005.

[4] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. Berman, and P. Maechling, "Scientific workflow applications on amazon EC2," in *E-Science Workshops, 2009 5th IEEE International Conference on*, dec. 2009, pp. 59 –66.

[5] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *Cloud Computing*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2010, vol. 34, pp. 115–131.

[6] D. Anastasia and Y. Andreopoulos, "Linear image processing operations with operational tight packing," *IEEE Signal Processing Letters*, vol. 17, no. 4, pp. 375 –378, April 2010.

[7] ——, "Software designs of image processing tasks with incremental refinement of computation," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS), 2009*, Tampere, Finland, October 2009, pp. 249 –254.

[8] ——, “Scheduling and energy-distortion tradeoffs with operational refinement of image processing,” in *Proc. Design, Automation & Test in Europe (DATE)*, Dresden, Germany, March 2010, pp. 1719 –1724.

[9] ——, “Throughput-precision computation for generic matrix multiplication: Toward a computation channel for high-performance digital signal processing,” in *17th International Conference on Digital Signal Processing (DSP)*, July 2011, pp. 1 –6.

[10] ——, “Throughput-distortion computation of generic matrix multiplication: Toward a computation channel for digital signal processing systems,” *IEEE Transactions on Signal Processing*, vol. 60, pp. 2024–2037, April 2012.

[11] ——, “Software designs of image processing tasks with incremental refinement of computation,” *IEEE Transactions on Image Processing*, vol. 19, no. 8, pp. 2099 –2114, August 2010.

[12] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[13] J. Yang, D. Zhang, A. F. Frangi, and J. yu Yang, “Two-dimensional PCA: A new approach to appearance-based face representation and recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 131–137, 2004.

[14] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, ser. Supercomputing ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27.

[15] K. Goto and R. A. v. d. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, pp. 12:1–12:25, May 2008.

[16] S. Belongie, J. Malik, and J. Puzicha, “Shape matching and object recognition using shape contexts,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 4, pp. 509 –522, apr 2002.

[17] Y. LeCun, F. J. Huang, and L. Bottou, “Learning methods for generic object recognition with invariance to pose and lighting,” in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 2, june-2 july 2004, pp. II–97 – 104 Vol.2.

[18] V. Goyal and M. Vetterli, “Computation-distortion characteristics of block transform coding,” in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-97)*, vol. 4, Apr 1997, pp. 2729–2732 vol.4.

[19] J. Valentim, P. Nunes, and F. Pereira, “Evaluating MPEG-4 video decoding complexity for an alternative video complexity verifier model,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 11, pp. 1034–1044, Nov 2002.

[20] B. Zeng, R. Li, and M. L. Liou, “Optimization of fast block motion estimation algorithms,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 6, pp. 833–844, Dec 1997.

[21] K. Lengwehasatit and A. Ortega, “Scalable variable complexity approximate forward DCT,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 11, pp. 1236–1248, Nov. 2004.

[22] D. Turaga, M. van der Schaar, and B. Pesquet-Popescu, “Complexity scalable motion compensated wavelet video encoding,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 8, pp. 982–993, Aug. 2005.

[23] S. H. Nawab, A. V. Oppenheim, A. P. Chandrakasan, J. M.Winograd, and J. T.Ludwig, “Approximate signal processing,” *J. VLSI Signal Process. Syst.*, vol. 15, no. 1/2, pp. 177–200, 1997.

[24] T. Tran, “The binDCT: fast multiplierless approximation of the DCT,” *IEEE Signal Processing Letters*, vol. 7, no. 6, pp. 141 –144, jun 2000.

[25] J. Liang and T. Tran, “Fast multiplierless approximations of the DCT with the lifting scheme,” *IEEE Transactions on Signal Processing*, vol. 49, no. 12, pp. 3032 –3044, dec 2001.

[26] L.-M. Po and W.-C. Ma, “A novel four-step search algorithm for fast block motion estimation,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 313 –317, jun 1996.

[27] W. Yuan and K. Nahrstedt, “Practical voltage scaling for mobile multimedia devices,” in *MULTIMEDIA ’04: Proceedings of the 12th annual ACM international conference on Multimedia*. New York, NY, USA: ACM, 2004, pp. 924–931.

[28] E. Akyol and M. van der Schaar, “Complexity model based proactive dynamic voltage scaling for video decoding systems,” *IEEE Transactions on Multimedia*, vol. 9, no. 7, pp. 1475–1492, Nov. 2007.

[29] J. Winograd and S. Nawab, “Incremental refinement of DFT and STFT approximations,” *IEEE Signal Processing Letters*, vol. 2, no. 2, pp. 25–27, Feb 1995.

[30] Y. Andreopoulos and M. van der Schaar, “Incremental refinement of computation for the discrete wavelet transform,” *IEEE Transactions on Signal Processing*, vol. 56, no. 1, pp. 140–157, Jan. 2008.

[31] Y. Andreopoulos and I. Patras, “Incremental refinement of image salient-point detection,” *IEEE Transactions on Image Processing*, vol. 17, no. 9, pp. 1685–1699, Sept. 2008.

[32] A. Chandrakasan and T. Xanthopoulos, “A low-power IDCT macrocell for MPEG-2 MP@ML exploiting data distribution properties for minimal activity,” *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 5, pp. 693 –703, may 1999.

[33] ——, "A low-power DCT core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization," *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 5, pp. 740 –750, may 2000.

[34] A. Chandrakasan, A. Sinha, and A. Wang, "Energy scalable system design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 2, pp. 135 –145, april 2002.

[35] G. Constantinides, P. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1432 – 1442, oct. 2003.

[36] M. Hutton, J. Schleicher, D. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bourgeault, A. Lee, H. Kim, and R. Saini, "Improving fpga performance and area using an adaptive logic module," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3203, pp. 135–144.

[37] D. Dickin and L. Shannon, "Exploring FPGA technology mapping for fracturable LUT minimization," in *Field-Programmable Technology (FPT), 2011 International Conference on*, dec. 2011, pp. 1 –8.

[38] P. Sherwood and K. Zeger, "Progressive image coding on noisy channels," in *Proceedings Data Compression Conference, 1997. DCC '97.*, mar 1997, pp. 72 –81.

[39] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Transactions on Image Processing*, vol. 1, no. 2, pp. 205 –220, apr 1992.

[40] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 still image coding system: an overview," *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4, pp. 1103 –1127, nov 2000.

[41] R. Sundaresh and P. Hudak, "Incremental computation via partial evaluation," in *Proceedings 18th Symposium on Principles of Programming Languages*. ACM, Jan. 1991, pp. 1–13.

[42] R. Hoover, "Alphonse: incremental computation as a programming abstraction," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, ser. PLDI '92. New York, NY, USA: ACM, 1992, pp. 261–272.

[43] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 315–328.

[44] Y. A. Liu, S. D. Stoller, and T. Teitelbaum, "Static caching for incremental computation," *ACM Trans. Program. Lang. Syst.*, vol. 20, pp. 546–585, May 1998.

[45] W. W. Pugh, "Incremental computation and the incremental evaluation of functional programs," Ph.D. dissertation, Cornell University, August 1988.

[46] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78 –117, 1970.

[47] "Multifacet project," 1998 –2012. [Online]. Available: http://www.cs.wisc.edu/multifacet/

[48] D. Citron and L. Rudolph, "Creating a wider bus using caching techniques," in *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, 1995, pp. 90 –99.

[49] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 212–.

[50] R. S. de Castro, A. P. do Lago, and D. D. Silva, "Adaptive compressed caching: Design and implementation," *Computer Architecture and High Performance Computing, Symposium on*, vol. 0, p. 10, 2003.

[51] T. Cortes, Y. Becerra, and R. Cervera, "Swap compression: resurrecting old ideas," *Softw. Pract. Exper.*, vol. 30, pp. 567–587, April 2000.

[52] R. Cervera, T. Cortes, and Y. Becerra, "Improving application performance through swap compression," in *Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1999, pp. 46–46.

[53] I. C. Tuduce and T. Gross, "Adaptive main memory compression," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 29–29.

[54] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 74–85.

[55] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," *EUROMICRO Conference*, vol. 0, p. 0423, 1996.

[56] E. Hallnor and S. Reinhardt, "A unified compressed memory hierarchy," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, feb. 2005, pp. 201 – 212.

[57] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271 –285, march 2001.

[58] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory expansion technology (MXT): Software sup-

port and performance," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 287 –301, march 2001.

[59] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith, "Performance of hardware compressed main memory," in *The Seventh International Symposium on High-Performance Computer Architecture*, 2000, pp. 73–81.

[60] J. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28 –35, jul 2000.

[61] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, apr 2001, p. 7 pp.

[62] B. Abali and H. Franke, "Operating system support for fast hardware compression of main memory contents," in *Main MemoryâĂİ, Memory Wall Workshop, International Symposium on Computer Architecture (ISCA2000*, 2000.

[63] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[64] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879 –899, may 2008.

[65] R. Dimond, S. RacanieÌĂ andre, and O. Pell, "Accelerating large-scale HPC applications using FPGAs," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, july 2011, pp. 191 –192.

[66] A. Bhattacharjee, G. Contreras, and M. Martonosi, "Parallelization libraries: Characterizing and reducing overheads," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 5:1–5:29, February 2011.

[67] G. Liu and X. Hu, "Performance and efficiency evaluation and analysis of supercomputers," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 4, july 2010, pp. 642 –646.

[68] B. Greer, "The most important technical library in the world," *SIGPLAN Fortran Forum*, vol. 17, pp. 25–32, December 1998.

[69] Intel Corporation, *Intel®Math Kernel Library, Reference Manual*. [Online]. Available: http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf

[70] ——, *Intel®64 and IA-32 Architectures Optimization Reference Manual*, June 2011. [Online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[71] Advanced Micro Devices Inc., *Software Optimization Guide for AMD64 Processors*, September 2005. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/25112.PDF

[72] K. Goto and R. van de Geijn, "On reducing TLB misses in matrix multiplication," 2002.

[73] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum," *ACM Trans. Math. Softw.*, vol. 28, pp. 239–267, June 2002.

[74] J. Choi, J. Dongarra, and D. Walker, "PB-BLAS: a set of parallel block basic linear algebra subprograms," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, may 1994, pp. 534 –541.

[75] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," in *Proceedings of the 11th international conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 340–347.

[76] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '04. New York, NY, USA: ACM, 2004, pp. 133–137.

[77] K. Underwood and K. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, april 2004, pp. 219 – 228.

[78] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 86–95.

[79] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[80] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354–356, 1969.

[81] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull, "Implementation of strassen's algorithm for matrix multiplication," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, 1996.*, 1996, p. 32.

[82] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 1–6.

[83] ——, "On the asymptotic complexity of matrix multiplication," in *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, ser. SFCS '81. Washington, DC, USA: IEEE Computer Society, 1981, pp. 82–90.

[84] S. Robinson, "Toward an optimal algorithm for matrix multiplication," SIAM News, November 2005. [Online]. Available: http://www.siam.org/news/news.php?id=174

[85] P. Drineas and R. Kannan, "Fast monte-carlo algorithms for approximate matrix multiplication," in *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, 2001, pp. 452 –459.

[86] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast monte carlo algorithms for matrices I: Approximating matrix multiplication," *SIAM J. Comput.*, vol. 36, pp. 132–157, July 2006.

[87] M. Baboulin, A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," 2008.

[88] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed precision iterative refinement techniques for the solution of dense linear systems," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 457–466, November 2007.

[89] J. Demmel, Y. Hida, E. J. Riedy, and X. S. Li, "Extra-precise iterative refinement for overdetermined least squares problems," *ACM Trans. Math. Softw.*, vol. 35, pp. 28:1–28:32, February 2009.

[90] P. Hazucha and C. Svensson, "Impact of CMOS technology scaling on the atmospheric neutron soft error rate," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2586 –2594, dec 2000.

[91] R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," in *International Electron Devices Meeting, 2002 (IEDM '02)*, 2002, pp. 329 – 332.

[92] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11.*, feb. 2005, pp. 243 – 247.

[93] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap: (invited paper)," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, 2010, pp. 1011–1016.

[94] S. Mitra, M. Zhang, N. Seifert, T. Mak, and K. S. Kim, “Built-in soft error resilience for robust system design,” in *IEEE International Conference on Integrated Circuit Design and Technology, 2007 (ICICDT ’07)*, 30 2007-june 1 2007, pp. 1 –6.

[95] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, “Soft-error resilience of the IBM POWER6 processor,” *IBM J. Res. Dev.*, vol. 52, pp. 275–284, May 2008.

[96] N. P. Carter, H. Naeimi, and D. S. Gardner, “Design techniques for cross-layer resilience,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE ’10, 2010, pp. 1023–1028.

[97] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, “Scalable stochastic processors,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE ’10, 2010, pp. 335–338.

[98] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, “Soft-error detection using control flow assertions,” in *Proceedings. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003.*, Nov. 2003, pp. 581 – 588.

[99] “Operation refinement of image processing (ORIP) tasks project,” 2008-2012. [Online]. Available: http://www.ee.ucl.ac.uk/~iandreop/ORIP.html

[100] A. Kadyrov and M. Petrou, “The “invaders” algorithm: Range of values modulation for accelerated correlation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1882–1886, Nov. 2006.

[101] A. K. Prasoon and K. Rajan, “4 $\times$ 4 2-D DCT for H.264/AVC,” in *Proceedings of the International Conference on Advances in Computing, Communication and Control*, ser. ICAC3 ’09. New York, NY, USA: ACM, 2009, pp. 573–577.

[102] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[103] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity transform and quantization in H.264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 598–603, July 2003.

[104] D. Marpe, T. Wiegand, and S. Gordon, "H.264/MPEG4-AVC fidelity range extensions: tools, profiles, performance, and application areas," in *IEEE International Conference on Image Processing, 2005. ICIP 2005.*, vol. 1, Sept. 2005, pp. I–593–6.

[105] D. Donoho, "De-noising by soft-thresholding," *IEEE Transactions on Information Theory*, vol. 41, no. 3, pp. 613 –627, may 1995.

[106] H. Sun and W. Kwok, "Concealment of damaged block transform coded images using projections onto convex sets," *IEEE Transactions on Image Processing*, vol. 4, no. 4, pp. 470–477, Apr 1995.

[107] W. Sung and K.-I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Transactions on Signal Processing*, vol. 43, no. 12, pp. 3087–3090, Dec 1995.

[108] B. Natarajan, V. Bhaskaran, and K. Konstantinides, "Low-complexity block-based motion estimation via one-bit transforms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 4, pp. 702 –706, Aug 1997.

[109] S. Erturk, "Multiplication-free one-bit transform for low-complexity block-based motion estimation," *IEEE Signal Processing Letters*, vol. 14, no. 2, pp. 109 –112, Feb. 2007.

[110] "The olpc wiki." [Online]. Available: http://wiki.laptop.org/

[111] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, April 2004.

[112] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[113] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, pp. 308–323, September 1979.

[114] J. L. Gustafson and B. S. Greer, "A hardware accelerator for the Intel Math Kernel," White Paper, ClearSpeed Technology Inc., 2006.

[115] R. Gray and D. Neuhoff, "Quantization," *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2325 –2383, oct 1998.

[116] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, pp. 147–158, 2004.

[117] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, pp. 1–17, March 1988.

[118] F. Abdelkefi, P. Duhamel, and F. Alberge, "Impulsive noise cancellation in multicarrier transmission," *IEEE Transactions on Communications*, vol. 53, no. 1, pp. 94 – 106, jan. 2005.

[119] G. Bradski and A. Kaehler, *Learning OpenCV*. O'Reilly Media, 2008.

[120] Q. Du and J. Fowler, "Hyperspectral image compression using JPEG2000 and principal component analysis," *IEEE Geoscience and Remote Sensing Letters*, vol. 4, no. 2, pp. 201 –205, april 2007.

[121] D. Aberdeen and J. Baxter, "Emmerald: a fast matrixâĂŞmatrix multiply using intel's SSE instructions," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 2, pp. 103–119, 2001.

[122] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, pp. 1–17, March 1990.

[123] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, November 2009, reference Number: 248966-020.

[124] *Intel® SSE4 Programming Reference*, Intel Corporation, July 2007, reference Number: D91561-003.

[125] "Loop blocking to optimize memory use on 32-bit architecture." [Online]. Available: http://software.intel.com/en-us/articles/loop-blocking-to-optimize-memory-use-on-32-bit-architecture

[126] N. R. Shanbhag, R. A. Abdallah, R. Kumar, and D. L. Jones, "Stochastic computation," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 859–864.

[127] Y.-C. Lin and S.-C. Tai, "Fast full-search block-matching algorithm for motion-compensated video compression," *IEEE Transactions on Communications*, vol. 45, no. 5, pp. 527 –531, may 1997.

[128] A. Frank and A. Asuncion, "YearPredictionMSD data set (a subset of the data from the million song dataset)," UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, Tech. Rep., 2011. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD

[129] K. Asanovic and N. Morgan, "Experimental determination of precision requirements for back-propagation training of artificial neural networks," in *In Proceedings 2nd International Conference on Microelectronics for Neural Networks*, 1991, pp. 9–15.

[130] M. Anam and Y. Andreopoulos, "Throughput scaling of convolution for error-tolerant multimedia applications," *IEEE Transactions on Multimedia*, vol. 14, no. 3, pp. 797 –804, june 2012.