Software Designs Of Image Processing Tasks With Incremental Refinement Of Computation

Davide Anastasia and Yiannis Andreopoulos*

ABSTRACT

Software realizations of computationally-demanding image processing tasks (e.g. image transforms and convolution) do not currently provide graceful degradation when their clock-cycles budgets are reduced, e.g. when delay deadlines are imposed in a multi-tasking environment to meet throughput requirements. This is an important obstacle in the quest for full utilization of modern programmable platforms' capabilities since worst-case considerations must be in place for reasonable quality of results. In this paper, we propose (and make available online) platform-independent software designs performing bitplane-based computation combined with an incremental packing framework in order to realize block transforms, 2D convolution and frame-by-frame block matching. The proposed framework realizes *incremental computation*: progressive processing of input-source increments improves the output quality monotonically. Comparisons with the equivalent non-incremental software realization of each algorithm reveal that, for the same precision of the result, the proposed approach can lead to comparable or faster execution, while it can be arbitrarily terminated and provide the result up to the computed precision. Application examples with region-of-interest based incremental computation, task scheduling per frame, and energy-distortion scalability verify that our proposal provides significant performance scalability with graceful degradation.

Index Terms— complexity scalable image processing, incremental refinement of computation, programmable processors

EDICS CODE: ELI-HDW

I. INTRODUCTION

Several popular applications, such as media players, computer graphics, image and video

*Corresponding author. The authors are with the University College London, Dept. of Electronic & Electrical Engineering, Torrington Place, WC1E 7JE, London, UK; Tel: +44-20-76797303; fax: +44-20-73889325; email: d.anastasia@ee.ucl.ac.uk (D. Anastasia), iandreop@ee.ucl.ac.uk (Y. Andreopoulos). The authors acknowledge the support of the EPSRC, grant EP/F020015/1.

post-processing, and motion estimation and compensation, are implemented today via software solutions in general-purpose processors. New generations of processors are increasingly powerful and enable more dedicated resource allocation to real-time multimedia tasks due to multi-core designs [1] and higher levels of parallelism. At the same time, state-of-the-art software compilers now automatically generate platform-specific optimized assembly code [2], thereby enabling platform-independent C++ solutions to achieve a high degree of utilization of the processor's resources.

However, 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 tradeoffs 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 analog systems, where energy autonomy would be increased with graceful degradation in the decoded video quality.

Existing algorithmic-oriented research focuses on complexity reduction [4]-[6] or complexity scalability for image processing tasks [7]-[9], where computational complexity is decreased and approximate results are produced. Implementation-oriented research focuses on multimedia-driven energy scaling of processors via dynamic voltage scaling [10] [11] in an attempt to provide computational scalability with approximate results. Overall, for all existing approaches: *(i)* algorithm-specific and/or system-specific customizations are required, which limit the applicability of the proposed techniques; *(ii)* only *one operational point* in the complexity-distortion curve [4] [8] 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.

An exception is found in theoretical proposals for incremental computation of transforms and salient point detection algorithms [12]-[14], where the main principle is: *under a refinement of the source description, the computation of the image processing task refines the previously-computed result*. However, existing work [12]-[14] is only using arithmetic complexity estimates and no practical realizations are proposed.

In this paper we address this aspect by proposing a unified software framework for image processing tasks exhibiting incremental refinement of computation. Our 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 [15]. Our initial efforts are reported in [16]. In this paper we are providing the following additional contributions: (*i*) we present the case of transform decompositions; (*ii*) a new approach is proposed for incremental block matching using the sum squared error criterion; (*iii*) we present applications of the proposed approach with: region-of-interest computation, parameterized real-time task scheduling with arbitrary variability, and video capturing and processing exhibiting power-distortion tradeoffs.

Section II presents our framework. Sections III-V detail the application of this approach for transform decompositions, 2D convolution and block matching. Section VI presents the experimental comparisons and Section VII presents applications demonstrating the advantages of the proposed approach in practical systems. Finally, Section VIII concludes the paper.

II. Image Processing Tasks Realized via Incremental Packing and Unpacking

A general depiction of the proposed framework for incremental computation based on source refinements is given in Figure 1. In the following subsection we discuss this framework in more detail, while Subsection B presents the basic tradeoffs of the proposed packing approach. The proposed framework is built under the notion of processing of bitplanes n, starting from the most significant bitplane (MSB) of the input ($n = N - 1$) and going down to the least-significant bitplane (LSB), which is bitplane $n = 0$. For non-negative 8-bit images considered in this paper, $N = 8$. Two useful definitions of quantities used in the remainder of the paper are given below.

Definition 1: For any quantity a used in the computation of an algorithm, a^n_{full}, $0 \leq n < N$, 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}, $0 \leq n < N$, 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 matrices[1], e.g. $\mathbf{A}_{\text{bit}}^{n}$ is the matrix containing the computed coefficients of $\mathbf{A}$ when only bitplane n of the input image is used.

A. Overall Framework

As shown in Figure 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 n th bitplane of all blocks ("Increment layer n" in Figure 1), $0 \le n < N$. Each increment layer n is also a layer of computation. We calculate the results of all M blocks of each layer concurrently using an incremental packing approach, as described in the following.

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

$$D_{\text{bit}}^{n}[i,j] = \sum_{m=1}^{M} B_{m,\text{bit}}^{n}[i,j] \times 2^{\lambda_{\text{type}}(m-1)\rho} \tag{1}$$

where $B_{m,\text{bit}}^{n}[i,j]$ is the (i,j) th value of block $\mathbf{B}_{m,\text{bit}}^{n}$ $(1 \le m \le M)$ that contains parts of increment layer n belonging to the m th spatial block, $\lambda_{\text{type}} = -1$ if 64-bit floating-point representation is used or $\lambda_{\text{type}} = 1$ if 32-bit unsigned integer representation is used, and $\rho > 0$ is the *packing coefficient*, whose explanation and setting will be discussed in detail in the following subsection (II.B). The last equation shows that the n th bitplane of the m th block is scaled by $2^{\lambda_{\text{type}}(m-1)\rho}$ and is then added to the sum of the previous blocks $1,\ldots,m-1$ of the same increment layer. This leads to a packed increment layer having all M blocks placed on one block $\mathbf{D}_{\text{bit}}^{n}$ and using integer or floating-point representation. The best choice for the utilized representation (integer or floating-point) is system dependent, as it will be shown by our experiments.

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

$$\mathbf{R}_{\text{bit}}^{n} = \left(\mathbf{D}_{\text{bit}}^{n} \; op \; \mathbf{K}\right) \tag{2}$$

Depending on the algorithm of interest, one could localize the calculation of (2) around areas of interest based on the previously-computed increment layers (as indicated in Figure 1). This will be used in the

[1] Boldface capital letters indicate matrices; the corresponding italicized letters indicate individual matrix elements, e.g. $\mathbf{A}$ and $A[i,j]$; all indices are integers; superscripts in matrices or scalars indicate the bitplane number and the frame index (except for superscript T that indicates transposition), the distinction between the two is identifiable from the context.

block matching task and in our experiments with region-of-interest based computation.

If an appropriate coefficient ρ is chosen for (1), it can be shown [17] that the results of all the blocks within increment layer n can be extracted from $\mathbf{R}^n_{\text{bit}}$ if:

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

This is based on the so-called "invaders" approach [17], where any integer linear operation can be performed by packing multiple operations together [see (1)], and then unpacking them by the reverse operation performed recursively for all values of all blocks. For floating-point representation ($\lambda_{\text{type}} = -1$), unpacking is performed by [16]:

$$m = 1: \quad \begin{array}{c} R^n_{1,\text{bit}}[i,j] \equiv R^n_{\text{bit}}[i,j], \\ U^n_{1,\text{bit}}[i,j] = \left\lfloor R^n_{1,\text{bit}}[i,j] + 0.5 \right\rfloor \end{array} \tag{3}$$

$$\forall m \in \{2,\ldots,M\}: \quad \begin{array}{c} R^n_{m,\text{bit}}[i,j] = 2^\rho \times \left(R^n_{m-1,\text{bit}}[i,j] - U^n_{m-1,\text{bit}}[i,j] \right), \\ U^n_{m,\text{bit}}[i,j] = \left\lfloor R^n_{m,\text{bit}}[i,j] + 0.5 \right\rfloor \end{array} \tag{4}$$

where: $\mathbf{U}^n_{m,\text{bit}}$ is the output increment of the result for block m, $\mathbf{R}^n_{m,\text{bit}}$ is the $\mathbf{R}^n_{\text{bit}}$ matrix at the mth unpacking and $\lfloor a + 0.5 \rfloor$ performs rounding to the nearest integer. For integer representation ($\lambda_{\text{type}} = 1$), the unpacking is performed by:

$$m = 1: \quad \begin{array}{c} R^n_{1,\text{bit}}[i,j] = R^n_{\text{bit}}[i,j], \\ U^n_{1,\text{bit}}[i,j] = \text{mod}\left(R^n_{1,\text{bit}}[i,j], 2^\rho \right) \end{array} \tag{5}$$

$$\forall m \in \{2,\ldots,M\}: \quad \begin{array}{c} R^n_{m,\text{bit}}[i,j] = \left(R^n_{m-1,\text{bit}}[i,j] \gg \rho \right), \\ U^n_{m,\text{bit}}[i,j] = \text{mod}\left(R^n_{m-1,\text{bit}}[i,j], 2^\rho \right) \end{array} \tag{6}$$

where $\left(a \gg \rho \right)$ shifts a down by ρ bits and $\text{mod}(a, 2^\rho) = a - \left\lfloor a \big/ 2^\rho \right\rfloor 2^\rho$ is the modulo operation. The selection of the appropriate packing coefficient ρ depends on the specific algorithm being considered. In addition, even though Figure 1 shows all blocks of the input image being packed together, in practice the value of M depends on the dynamic range of the result of each increment layer. These aspects are elaborated further in the following sections.

As shown in Figure 1, 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, $\mathbf{U}^n_{1,\text{bit}},\ldots,\mathbf{U}^n_{M,\text{bit}}$:

[2] In Section V, we are also considering the case where op is a quadratic operation. It is shown that such non-linear operations are possible, but they require careful handling within a packing framework.

$$\forall m \in \{1,\ldots,M\} : \mathbf{U}^{n}_{m,\text{full}} = \mathbf{U}^{n+1}_{m,\text{full}} + \mathbf{U}^{n}_{m,\text{bit}} \tag{7}$$

with $\mathbf{U}^{N}_{m,\text{full}} \equiv \mathbf{0}$. This leads to computation of the processing task with increased precision for increased number of increment layers, as shown in the visual examples of Figure 1. Due to the utilization of the packing technique, the results of all M blocks are computed *concurrently* by (2). Depending on the overhead of packing and unpacking, we expect to save operations in comparison to the direct computation of each layer.

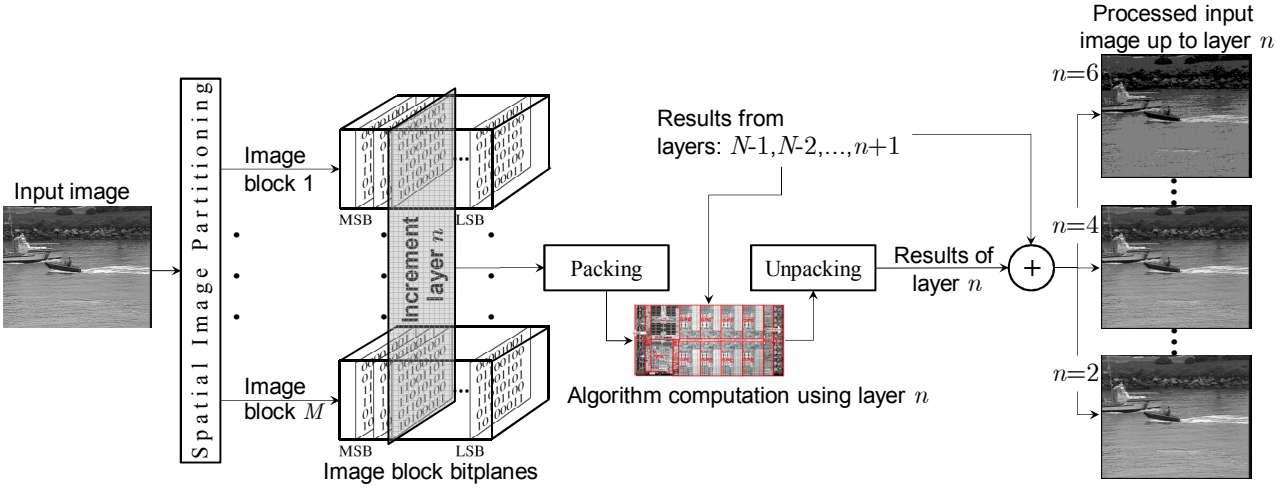


Figure 1. Incremental refinement of computation using packing and unpacking of increment layers extracted progressively from the input image data. The output result is progressively refined via the computation of more increment layers. The computation of each layer can also utilize results from previous layers to reduce complexity or focus the computation on regions of interest.

B. Controlling Parameters and Practical Extensions

The parameters controlling the proposed approach of Figure 1 are: the total number of increment layers (N), the total number of blocks (M) and the packing coefficient (ρ) that controls the stacking of multiple increment layers in one operand $D^{n}_{\text{bit}}[i,j]$. Ideally, we would like to maximize the packing capability in order to perform as many operations simultaneously as possible [17].

As analyzed in the original "invaders" algorithm, the packing capability depends on the dynamic range of the operations. Furthermore, if packing with integer representation is desired, ρ has to be integer. The dynamic range of the packing obtained with the maximum packing coefficient cannot be smaller than 2^{-50} for 64-bit floating-point representation [17], and it cannot be larger than 2^{31} for 32-bit unsigned integer representation, which leads to $\left[M + 0.5\left(\lambda_{\text{type}} - 1\right)\right]\rho \le \omega_{\text{type}}$, with $\omega_{\text{type}} = 50$ or $\omega_{\text{type}} = 31$,

respectively[3]. If the range of all outputs $\left(\mathbf{B}_{m,\text{bit}}^{n}\ op\ \mathbf{K}\right)$ (for every bitplane n and block m) is contained in the interval $\left\{-A_{\max}, A_{\max}\right\}$, then, following loose packing theory [17], we have $\rho \geq \left\lceil \log_2 A_{\max} \right\rceil + 1$. Selecting the minimum value of ρ satisfying the inequality, we reach

$$M \leq \left\lfloor \frac{\omega_{\text{type}}}{\left\lceil \log_2 A_{\max} \right\rceil + 1} \right\rfloor - 0.5\left(\lambda_{\text{type}} - 1\right) \tag{8}$$

As expected, the number of packed blocks decreases with the increase of the output's dynamic range. The output dynamic range of each layer depends on the algorithm of interest and it will be discussed separately in the following sections. 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 [17] must allow for correct rounding by (3) and (4), i.e.:

$$\frac{2^{-\omega_{\text{type}}} A_{\max}}{2^{-(M-1)\rho}} < 0.5\,. \tag{9}$$

In our designs, M is initially derived by (8) and then decreased (if needed) so that (9) holds.

For notational simplicity, this paper 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 [15] and it is utilized in the experimental section of the paper 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 1 indicates that all image blocks are packed together (i.e. the algorithm splits the image into M blocks), there are practical cases where the desired number of blocks is larger than the value of M calculated by (8) and (9), e.g. in a 4×4 block transform decomposition of a 352x288-pixel image. For those cases, after packing Q groups of M blocks (where $Q \times M$ gives the total number of image blocks), the processing, unpacking and result-incrementing processes are interleaved. This is shown in the schematic of Figure 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.

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

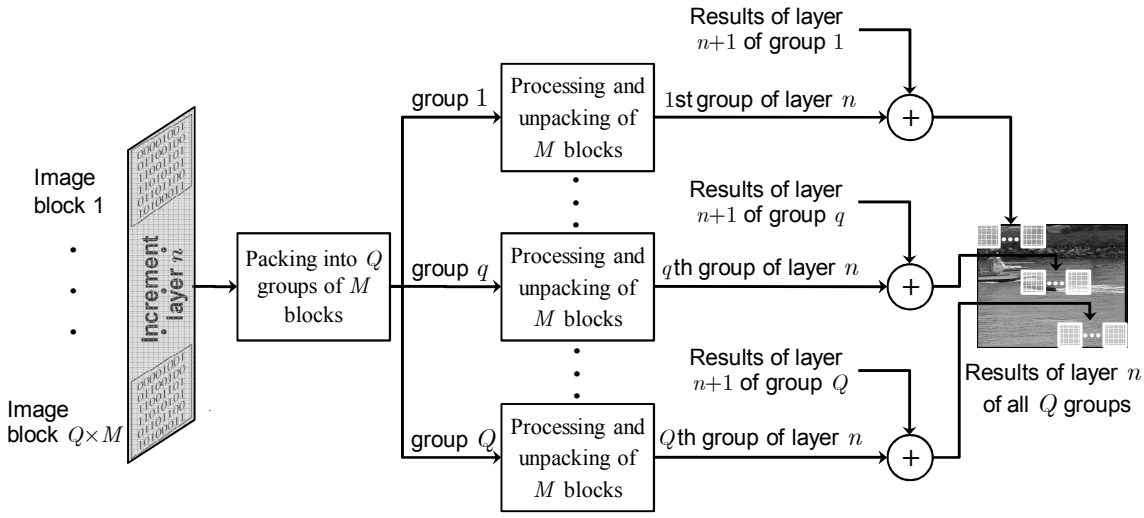


Figure 2. Packing, processing, unpacking and incrementing the result with Q groups of M blocks.

III. Incremental Transform Decomposition

The input of this case is Q groups of M blocks of $C \times C$ input pixels, $\mathbf{B}^0_{m,\text{full}}$, $1 \le m \le M$, with $C = \{4, 8, 16\}$ for typical cases of block transforms found in the literature [18]-[20]. The transform matrix is given by a $C \times C$ integer kernel $\mathbf{T}_{\text{for}}$, e.g. the H.264 4×4 transform [18]. Transforms kernels with non-integer coefficients can be approximated by a fixed-point (FXP) representation with the appropriate number of fractional bits [21]. 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 M blocks of each group of blocks q, $1 \le q \le Q$.

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

$$\forall m \in \{1, \ldots, M\} : \mathbf{U}^0_{m,\text{full}} = \mathbf{T}_{\text{for}} \mathbf{B}^0_{m,\text{full}} \mathbf{T}^T_{\text{for}}. \tag{10}$$

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 m th block by:

$$\forall m \in \{1, \ldots, M\} : \mathbf{U}^n_{m,\text{bit}} = \mathbf{T}_{\text{for}} \mathbf{B}^n_{m,\text{bit}} \mathbf{T}^T_{\text{for}}. \tag{11}$$

and the results are added to the previously-computed results by (7).

The above process was already proposed within transform-specific formulations for incremental computation of the discrete Fourier transform [12] and the lifting-based discrete wavelet transform [13]. Here, we consider packing the results in order to accelerate the incremental computation in software. We form $\mathbf{D}^n_{\text{bit}}$ by (1) and it is used to compute the packed result of all M blocks by:

$$\mathbf{R}_{\text{bit}}^{n} = \mathbf{T}_{\text{for}}\mathbf{D}_{\text{bit}}^{n}\mathbf{T}_{\text{for}}^{T}. \tag{12}$$

The results are unpacked from $\mathbf{R}_{\text{bit}}^{n}$ using (3) and (4) [or (5) and (6) if integer packing is performed] and the final results per bitplane n are derived by (7). Notice that only one transform decomposition with block $\mathbf{D}_{\text{bit}}^{n}$ is performed by (12) instead of M block decompositions performed by (11). 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), M, depends on the worst-case dynamic range $A_{\max}$ of (11). This range can be found by assuming the worst-case block:

$$B_s[i,j] = \begin{cases} 1, & \text{if } (-1)^s T_{\text{for}}[i,j] > 0 \\ 0, & \text{if } (-1)^s T_{\text{for}}[i,j] \le 0 \end{cases} \text{ for } 0 \le i,j < C, s \in \{0,1\}. \tag{13}$$

$$A_{\max} = \left(2^p - 1\right) \max_{\forall s}\left\{ \max_{\forall j}\left\{ \sum_{i=0}^{C-1} T_{\text{for}}[i.j] \cdot B_s[i,j] \right\} \times \max_{\forall i}\left\{ \sum_{j=0}^{C-1} T_{\text{for}}[i.j] \cdot B_s[i,j] \right\} \right\}. \tag{14}$$

where p is the number of bitplanes packed in each increment layer n, e.g. $p = 1$ 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

$$\forall m \in \{1,\ldots,M\} : \mathbf{B}_{m,\text{full}}^{0} = \mathbf{T}_{\text{inv}}\mathbf{R}_{m,\text{full}}^{0}\mathbf{T}_{\text{inv}}^{T}, \tag{15}$$

with $\mathbf{T}_{\text{inv}} = \mathbf{T}_{\text{for}}^{-1}$. 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:

$$\mathbf{T}_{\text{for}+} = \mathbf{T}_{\text{for}} + \mathbf{P}, \tag{16}$$

with $\mathbf{P} = -\min_{\forall i,j}\left\{T_{\text{for}}[i,j]\right\}\mathbf{1}_{C\times C}$ and $\mathbf{1}_{C\times C}$ a $C \times C$ matrix of ones. After the incrementally-computed decomposition is performed for each input block $\mathbf{D}_{\text{bit}}^{n}$ using $\mathbf{T}_{\text{for}+}$, we need compensate for the added component of the kernel of (16) 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 (16) derives:

$$\mathbf{R}_{\text{bit}+}^{n} = \mathbf{T}_{\text{for}}\mathbf{D}_{\text{bit}}^{n}\mathbf{T}_{\text{for}}^{T} + \mathbf{P}\,\mathbf{D}_{\text{bit}}^{n}\mathbf{T}_{\text{for}}^{T} + \mathbf{T}_{\text{for}}\mathbf{D}_{\text{bit}}^{n}\mathbf{P}^{T} + \mathbf{P}\,\mathbf{D}_{\text{bit}}^{n}\mathbf{P}^{T} \tag{17}$$

out of which only the term $\mathbf{T}_{\text{for}}\mathbf{D}_{\text{bit}}^{n}\mathbf{T}_{\text{for}}^{T}$ is the desired increment. Hence, we do not investigate this option in

this paper and restrict our approach to floating-point representation for the transform decomposition case.

IV. Incremental 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 M partially overlapping horizontal "stripes", each of which is the considered to be the input block of samples, $\mathbf{B}_m^0$ ($1 \le m \le M$), having C_{in} columns. The number of rows in each block is controlled by the input image rows and the packing capability (i.e. the value of M). The convolution filter is given by a $V_{\text{kernel}} \times C_{\text{kernel}}$-coefficient kernel $\mathbf{T}_{\text{conv}}$ and convolution of the m th block is performed by:

$$\forall m \in \{1,\ldots,M\} : \mathbf{U}_{m,\text{full}}^0 = \mathbf{B}_{m,\text{full}}^0 * \mathbf{T}_{\text{conv}}. \tag{18}$$

In order to produce the correct result with the block-based calculation of (18), consecutive blocks share a common subset of rows $V_{\text{overlap}} = \left\lfloor V_{\text{kernel}} / 2 \right\rfloor$, 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 m th block by:

$$\forall m \in \{1,\ldots,M\} : \mathbf{U}_{m,\text{bit}}^n = \mathbf{B}_{m,\text{bit}}^n * \mathbf{T}_{\text{conv}}, \tag{19}$$

and the results are added to the previously-computed outputs by (7).

If we consider packing the results in order to accelerate the incremental computation, then $\mathbf{D}_{\text{bit}}^n$ is formed by (1) and it is used to compute the packed result of all M blocks by:

$$\mathbf{R}_{\text{bit}}^n = \mathbf{D}_{\text{bit}}^n * \mathbf{T}_{\text{conv}}, \tag{20}$$

The results are unpacked from $\mathbf{R}_{\text{bit}}^n$ using (3) and (4) [or (5) and (6) if integer packing is performed] and the final results per bitplane n are derived by (7). Visual examples of Gaussian filtering when $n \in \{6,4,2\}$ are given in Figure 1; similar examples with $n \in \{5,2,0\}$ are given in Figure 8. As in Section III, the packing capability depends on the worst-case dynamic range, which is calculated using $\mathbf{T}_{\text{conv}}$ in (13) and then:

$$A_{\max} = \left(2^p - 1\right) \max_{\forall s} \left\{ \left| \sum_{i=0}^{V_{\text{kernel}}-1} \sum_{j=0}^{C_{\text{kernel}}-1} B_s[i,j] \cdot T_{\text{conv}}[i,j] \right| \right\}. \tag{21}$$

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 (16) using $\mathbf{T}_{\text{conv}}$ and then, after unpacking, we increment the result by:

$$\forall m \in \{1,\ldots,M\} : \mathbf{U}^{n}_{m,\text{full}} = \mathbf{U}^{n+1}_{m,\text{full}} + \mathbf{U}^{n}_{m,\text{bit}} + \min_{\forall i,j}\left\{T_{\text{conv}}[i,j]\right\} \sum_{i=0}^{V_{\text{kernel}}-1} \sum_{j=0}^{C_{\text{kernel}}-1} B^{n}_{m,\text{bit}}[i,j] \qquad (22)$$

in order to compensate for the added element $\mathbf{P} = -\min_{\forall i,j}\left\{T_{\text{conv}}[i,j]\right\}\mathbf{1}_{C\times C}$. Finally, in order to permit incremental computation even *within* an increment layer, the calculation of (20) and the unpacking and incrementation of results are interleaved for each output coefficient $R^{n}_{\text{bit}}[i,j]$. This permits virtually seamless quality improvement with increased computation within each increment layer.

V. Incremental Block Matching

The problem of block matching between two successive images $\mathbf{I}^{0,t-1}_{\text{full}}$ and $\mathbf{I}^{0,t}_{\text{full}}$ (of $R_{\text{in}} \times C_{\text{in}}$ pixels) can be abstracted as follows. Given the q th non-overlapping block $\mathbf{B}^{0,t}_{q,\text{full}}$ of $C \times C$ pixels in $\mathbf{I}^{0,t}_{\text{full}}$ ($1 \le q \le Q$, assuming Q blocks in total) and a corresponding search area $\mathbf{S}^{0,t-1}_{q,\text{full}}$ of $2W \times 2W$ overlapping blocks in $\mathbf{I}^{0,t-1}_{\text{full}}$, find the $C \times C$ block in the search area that is closest to the q th block. Conventional search algorithms use non-linear distance criteria, such as the sum squared error (SSE) or the sum of absolute differences (SAD) [6]. In this paper, we propose an approach to perform incremental block matching using the SSE criterion. However, since the framework of (1)-(7) 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. The first, most straightforward, approach is to consider two consecutive image blocks in frame $\mathbf{I}^{0,t}_{\text{full}}$ and pack increments of these blocks together to compute a single distance criterion for both blocks, i.e. following the generic overview of Figure 1. However, due to the fact that the positions of the best match within the search ranges in frame $\mathbf{I}^{0,t-1}_{\text{full}}$ will be different, this has two important detriments: it makes early termination difficult to apply for block matching[4] and, for increments beyond the first one, it complicates the localization of the calculation around the previously-established match.

Another way to consider incremental processing for this case is to pack two sets of samples of a *single*

[4] 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 [6]. This becomes cumbersome for concurrent processing of two (or more) blocks as they have different matches with different minimum distances.

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 B explains how the (non-linear) SSE criterion can be calculated using the packed representations. The overall block matching algorithm is summarized in Subsection C.

A. 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 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 $\mathbf{B}_{q,\text{full}}^{0,t}$ is done by[5] ($0\le i<C$, $0\le j<\frac{C}{2}$):

$$\mathcal{B}_{q,\text{full}}^{n,t}[i,j]=x_{q,\text{full}}^{n,t}[i,j]+o_{q,\text{full}}^{n,t}[i,j]\times 2^{-\rho} \tag{23}$$

with $x_{q,\text{full}}^{n,t}[\cdot,\cdot]$, $o_{q,\text{full}}^{n,t}[\cdot,\cdot]$ the samples of $\mathbf{B}_{q,\text{full}}^{0,t}$ up to (and including) the nth bitplane and following the quincunx lattice of Figure 3(a) and ρ the packing coefficient, whose setting is discussed in Subsection B. For the corresponding search area $\mathbf{S}_{q,\text{full}}^{0,t-1}$, we form four packings by ($0\le i<2W$, $0\le j<W$):

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,0,0]=x_{q,\text{full}}^{n,t-1}[i,j]+o_{q,\text{full}}^{n,t-1}[i,j]\times 2^{-\rho} \tag{24}$$

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,1,0]=o_{q,\text{full}}^{n,t-1}[i,j]+x_{q,\text{full}}^{n,t-1}[i,j]\times 2^{-\rho} \tag{25}$$

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,0,1]=o_{q,\text{full}}^{n,t-1}[i,j]+x_{q,\text{full}}^{n,t-1}[i,j+1]\times 2^{-\rho} \tag{26}$$

$$\mathcal{S}_{q,\text{full}}^{n,t-1}[i,j,1,1]=x_{q,\text{full}}^{n,t-1}[i,j]+o_{q,\text{full}}^{n,t-1}[i,j+1]\times 2^{-\rho} \tag{27}$$

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(b). Notice that the packing rules of (23)-(27) correspond to the case of $M=2$ of (1) but, instead of using two blocks, we use the two lattice sample sets. The need for the four separate packings given by (24)-(27) 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 (24) and (25) 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 (26) and (27) are used when the block is compared to blocks located at (even,odd) and (odd,odd) positions in the search grid, respectively.

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

Examples for all four cases are given in Figure 3(b). Hence, for each search location $S_{q,\text{full}}^{0,t-1}(i_\text{s}, j_\text{s})$ we use $\mathcal{S}_{q,\text{full}}^{n,t-1}[i_\text{s}, \lfloor j_\text{s}/2 \rfloor, g, z]$ with $g = \text{mod}(i_\text{s}, 2)$ and $z = \text{mod}(j_\text{s}, 2)$. To keep the required memory footprint for (23)-(27) 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 is 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 SSE.

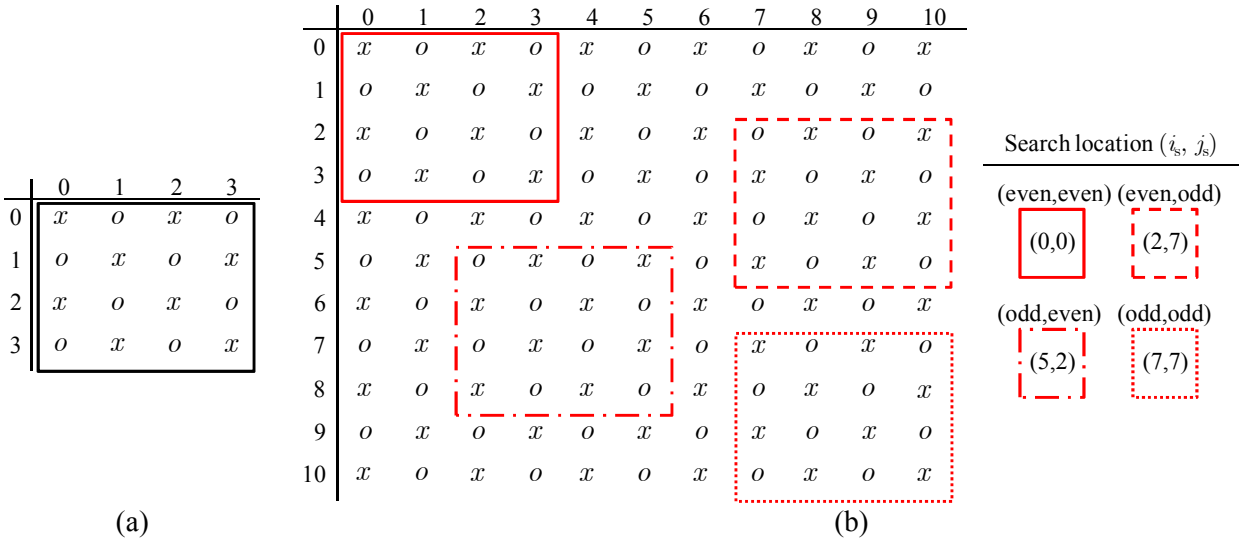


Figure 3; (a) Example of quincunx lattice for a 4×4 block; (b) Example of quincunx lattice of an 8×8 search area with indicative search positions highlighted. Any subblock of the search area within the coordinates $\{(0,0), \ldots (7,7)\}$ can be selected as a match.

B. SSE Calculation using Packed Representations

The block SSE calculation with packed representations using $M = 2$ 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 $\mathcal{S}_{q,\text{full}}^{n,t-1}[i_\text{s} + i, \lfloor j_\text{s}/2 \rfloor + j, g, z]$, which corresponds to search location (i_s, j_s) in $\mathbf{S}_{q,\text{full}}^{0,t-1}$. We calculate the packed SSE by:

$$\mathcal{D}_{\text{SSE}} = \sum_{i=0}^{C-1} \sum_{j=0}^{C/2-1} (\mathcal{B}_{q,\text{full}}^{n,t}[i, j] - \mathcal{S}_{q,\text{full}}^{n,t-1}[i_\text{s} + i, \lfloor j_\text{s}/2 \rfloor + j, g, z])^2 \,. \tag{28}$$

Per block position (i, j), (28) performs the squared difference between the packing of (23) and one of the packings of (24)-(27). In the remainder of this subsection, we analyze the case when (23) and (24) are used in (28), i.e. $g = 0$ and $z = 0$, since all other cases of (g, z) are examined in the same manner.

By replacing the packed representations using (23) and (24), and expanding the square we have:

$$\begin{aligned}\mathcal{D}_{\mathrm{SSE}} &= \sum_{i=0}^{C-1}\sum_{j=0}^{C/2-1}\left[\left(x_{q,\mathrm{full}}^{n,t}[i,j]-x_{q,\mathrm{full}}^{n,t-1}[i_{\mathrm{s}}+i,\left\lfloor j_{\mathrm{s}}/2\right\rfloor+j]\right)+\left(o_{q,\mathrm{full}}^{n,t}[i,j]-o_{q,\mathrm{full}}^{n,t-1}[i_{\mathrm{s}}+i,\left\lfloor j_{\mathrm{s}}/2\right\rfloor+j]\right)\times 2^{-\rho}\right]^2 \\ &= \sum_{i=0}^{C-1}\sum_{j=0}^{C/2-1}\left(x_{q,\mathrm{full}}^{n,t}[i,j]-x_{q,\mathrm{full}}^{n,t-1}[i_{\mathrm{s}}+i,\left\lfloor j_{\mathrm{s}}/2\right\rfloor+j]\right)^2+\sum_{i=0}^{C-1}\sum_{j=0}^{C/2-1}\left(o_{q,\mathrm{full}}^{n,t}[i,j]-o_{q,\mathrm{full}}^{n,t-1}[i_{\mathrm{s}}+i,\left\lfloor j_{\mathrm{s}}/2\right\rfloor+j]\right)^2\times 2^{-2\rho} \\ &\quad +2\sum_{i=0}^{C-1}\sum_{j=0}^{C/2-1}\left(x_{q,\mathrm{full}}^{n,t}[i,j]-x_{q,\mathrm{full}}^{n,t-1}[i_{\mathrm{s}}+i,\left\lfloor j_{\mathrm{s}}/2\right\rfloor+j]\right)\left(o_{q,\mathrm{full}}^{n,t}[i,j]-o_{q,\mathrm{full}}^{n,t-1}[i_{\mathrm{s}}+i,\left\lfloor j_{\mathrm{s}}/2\right\rfloor+j]\right)\times 2^{-\rho}\end{aligned} \tag{29}$$

The three terms of (29) show that we can unpack the SSE of each sampling grid and discard the unwanted cross-product term (which is scaled by $2^{-\rho}$). This is done following the unpacking process of (3) and (4): $U_1=\left\lfloor \mathcal{D}_{\mathrm{SSE}}+0.5\right\rfloor$, $\mathcal{D}_{\mathrm{SSE},1}=2^{-\rho}\left(\mathcal{D}_{\mathrm{SSE}}-U_1\right), U_2=\left\lfloor \mathcal{D}_{\mathrm{SSE},1}+0.5\right\rfloor$, $U_3=\left\lfloor 2^{-\rho}\left(\mathcal{D}_{\mathrm{SSE},1}-U_2\right)+0.5\right\rfloor$. The total SSE of both grids is $\mathcal{D}_{\mathrm{SSE,grid}}=U_1+U_3$ and U_2 is the unpacked cross-product term. Hence, the packed SSE of (28) "carries" within it the result of both incremental grids. Notice that, even though we packed two inputs, we need to perform *three* unpackings because of the unwanted cross-product term.

Setting of packing coefficient ρ: As explained in Subsection II.B, ρ can be set based on the worst-case dynamic range ($A_{\max}$) of the computed results within the packed representation. The worst case for (29) occurs in the cross-product term, where we have $A_{\max}=C^2\times\left(2^{N-n}\right)^2$. 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 $M=3$ correct unpackings. However, during the calculation of (28) we check at the end of every odd-numbered row for early termination (i.e. whether the SSE exceeds the previously-found best one). Hence, we can also set a worst-case dynamic range $A_{\max}^{\mathrm{early}}$ which, if exceeded, we enforce early termination because this will most-likely not correspond to a good match. Using (8) and (9) with $M=3$, we find that loose packing can accommodate $A_{\max}=32768$. Based on experiments with numerous real-world video sequences, we set $A_{\max}^{\mathrm{early}}=\frac{A_{\max}}{2}$ as the threshold for early termination.

C. Overall Block Matching Algorithm

The basic algorithm performed for each increment is given in Figure 4. 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 SSE. This simplifies the indexing of the software implementation. This algorithm can be readily extended to consider interpolation grids, and multi-frame motion estimation.

Setup: Set $A_{\max}^{\text{early}} = 2C \times \left(2^N\right)^2$. Set $\rho = \left\lceil \log_2 A_{\max}^{\text{early}} \right\rceil + 1$. // set packing coefficient to be used

Set $(i_{\text{s},q}^{n,*}, j_{\text{s},q}^{n,*}) = (W-1, W-1)$. // coordinates of best match are set to the center of the search area

Basic Algorithm: Incremental Block Matching using SSE for each input block $\mathbf{B}_{q,\text{full}}^{0,t}$ and search area $\mathbf{S}_{q,\text{full}}^{0,t-1}$

For each increment n, $n = N-1,\ldots,0$ {

1. Extract $\mathbf{B}_{q,\text{full}}^{n,t}$ and $\mathbf{S}_{q,\text{full}}^{n,t-1}$.
2. Calculate $\boldsymbol{\mathcal{B}}_{q,\text{full}}^{n,t}$ and $\boldsymbol{\mathcal{S}}_{q,\text{full}}^{n,t-1}$ using (23)-(27). Set $\mathcal{D}_{\text{SSE}}^{*} = \infty$. // the minimum distance will go in $\mathcal{D}_{\text{SSE}}^{*}$,
3. For each search row i_{s}, $i_{\text{s}} = 0,\ldots,2W-1$ {
4. For each search column j_{s}, $j_{\text{s}} = 0,\ldots,2W-1$ {
5. Set: $g = \text{mod}(i_{\text{s}},2)$, $z = \text{mod}(j_{\text{s}},2)$, $\mathcal{D}_{\text{SSE}} = 0$.
6. For each block row i, $i = 0,\ldots,C-1$ {
7. For each block column j, $j = 0,\ldots,C/2-1$ {
8. $\mathcal{D}_{\text{SSE}} \leftarrow \mathcal{D}_{\text{SSE}} + (\boldsymbol{\mathcal{B}}_{q,\text{full}}^{n,t}[i,j] - \boldsymbol{\mathcal{S}}_{q,\text{full}}^{n,t-1}[i_{\text{s}}+i, \lfloor j_{\text{s}}/2 \rfloor + j, g, z])^2$ // $a \leftarrow b$ assigns b to a
}
9. If $\text{mod}(i,2) = 1$ {
10. $U_1 = \lfloor \mathcal{D}_{\text{SSE}} + 0.5 \rfloor$, $\mathcal{D}_{\text{SSE},1} = 2^{-\rho}(\mathcal{D}_{\text{SSE}} - U_1)$, $U_2 = \lfloor \mathcal{D}_{\text{SSE},1} + 0.5 \rfloor$, $U_3 = \lfloor 2^{-\rho}(\mathcal{D}_{\text{SSE},1} - U_2) + 0.5 \rfloor$
Set $\mathcal{D}_{\text{SSE,grid}} = U_1 + U_3$. // unpack the $\mathcal{D}_{\text{SSE}}$ values of both grids and add them
11. If ($\mathcal{D}_{\text{SSE,grid}} > \mathcal{D}_{\text{SSE}}^{*}$ or $\mathcal{D}_{\text{SSE,grid}} > A_{\max}^{\text{early}}$) Then Go to Step 12 // early termination
}
}
12. If $\mathcal{D}_{\text{SSE,grid}} < \mathcal{D}_{\text{SSE}}^{*}$ Then { Set $\mathcal{D}_{\text{SSE}}^{*} = \mathcal{D}_{\text{SSE,grid}}$. Set $(i_{\text{s},q}^{n,*}, j_{\text{s},q}^{n,*}) = (i_{\text{s}}, j_{\text{s}})$ }
}
}
13. Store coordinates of best match for increment n of block q : $(i_{\text{s},q}^{n,*}, j_{\text{s},q}^{n,*}) = (i_{\text{s}}, j_{\text{s}})$
}

Figure 4. Pseudocode of incremental block matching using sum squared error.

Incremental block matching can benefit from the knowledge of the best match found for each block during the previous increment layers $N-1,\ldots,n+1$ in order to speed up the execution. This is performed as follows. For the first 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 [6]. For subsequent increments of each block q, we only search in the neighborhood of the previously-found best match for this block. This is performed by performing a spiral search within a fixed distance limit of W_{spiral} pixels horizontally and vertically (see [15] for full details on the implementation). The use of log-search and the localization 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 next section.

VI. EXPERIMENTAL RESULTS FOR BITPLANE-DRIVEN INCREMENTAL COMPUTATION

For our experiments, we used the xo-laptop of the OLPC foundation (detailed specification can be found in [22]) 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 2.5GHz with 2Gb RAM) running Microsoft Windows XP (denoted as "mainstream" profile). All programs were written in C++ and compiled with the gcc4.1.2 compiler in Linux and with the Microsoft Visual Studio 2008 compiler in Windows, with all default optimizations of "-o2 (maximize speed)" mode in both cases. To achieve stable execution-time measurements with high precision in both platforms, we used the Windows `QueryPerformanceCounter()` function and the Linux `gettimeofday()` function and run all programs in highest priority. Only the execution time required for the computation was measured for 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.

We utilized the Common Interchange Format (CIF) video sequences "Coastguard", "Foreman" "Mobile", "Silent" and "Stefan" as input video frames at 30fps. The sequences consist of 300 frames each and provide a 1500-frame video with diverse content. For the low-end profile, we downsampled the sequences to quarter-CIF (QCIF) format at 10fps in order to achieve real-time (or near real-time) processing with the xo-laptop. The signal-to-noise ratio (SNR) or the peak-signal-to-noise ratio (PSNR) measurements presented in the results utilize 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, $n = 0$). PSNR 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.

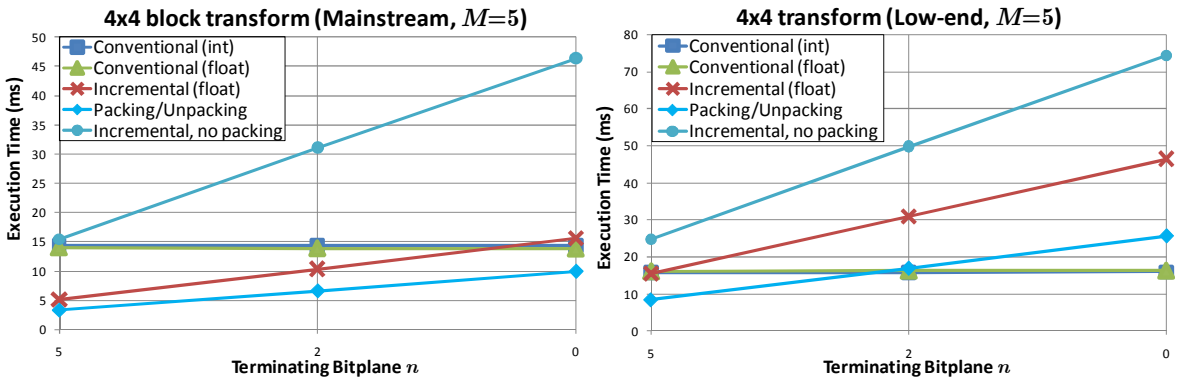
A. Incremental Transform Decomposition and 2D Convolution Experiments

We present results with the 4×4 H.264/AVC block transform [18] and with the fidelity-range extension (FRExt) 16×16 block transform kernel [19] 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 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

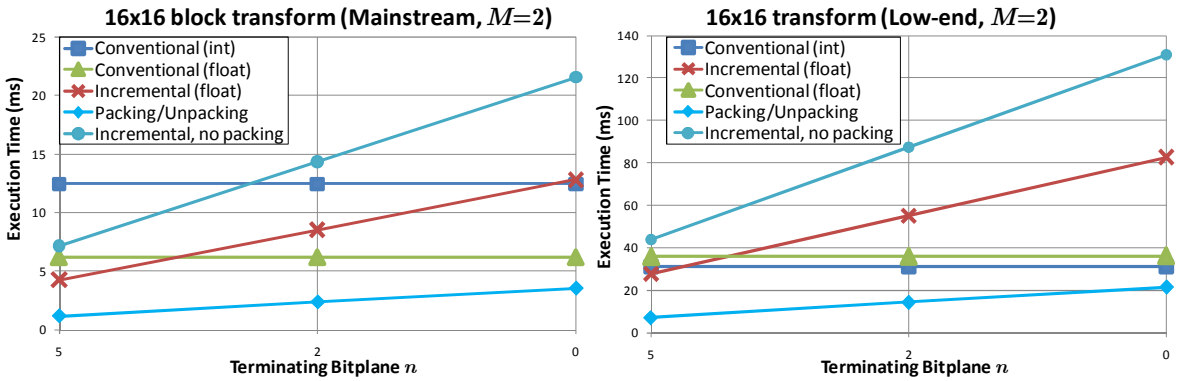
number of bits for the fractional part of the FXP representation ensured that SNR above 58dB was obtained for all our filtering experiments in comparison to the results obtained with the floating-point representation of the filter kernels. The small kernel is applied on the QCIF content in the low-end 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 $\mathbf{T}_{\text{conv}}$ for the two profiles. The results are shown in Figure 5-Figure 7, where we also report the number of packed blocks (M) achieved by the incremental approach following (8) and (9). The corresponding average SNR results are given in Table 1. Visual examples of outputs of the 12×12 Gaussian filtering at different precisions are given in Figure 8.

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 5-Figure 7 since the conventional approach cannot provide for arbitrary termination. Instead, this feature is explored in detail in Section VII.

Comparisons performed: In order to examine the impact of the utilized numerical representation, Figure 5-Figure 7 show execution time results for both conventional and incremental approaches when using floating-point and integer representation. The only exception is in Figure 5 (transform decomposition), where Section III demonstrated [via (17)] 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 5-Figure 7 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 when *packing is not used*, i.e. each increment of each block is computed separately.



(a)



(b)

Figure 5. Transform decomposition results; (a) 4×4 AVC transform ; (b) 16×16 FRExt kernel.

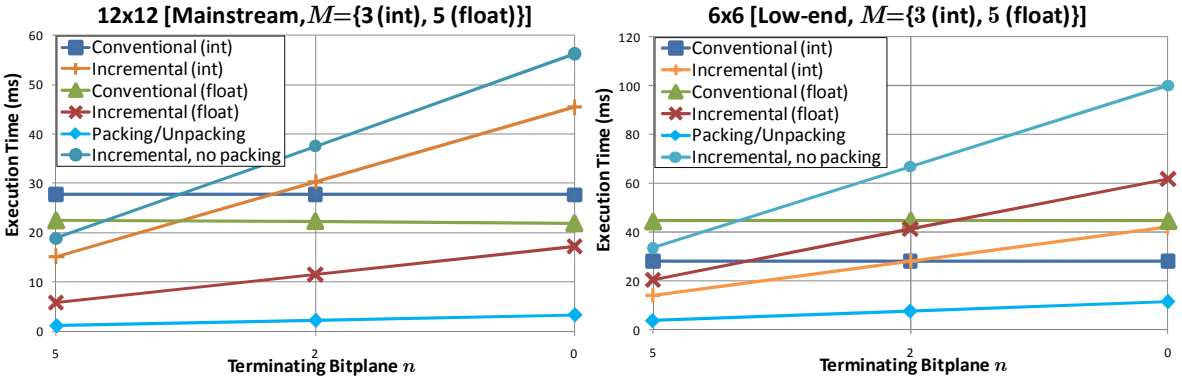


Figure 6. 2D convolution results with 12×12 (mainstream profile) and 6×6 (low-end profile) Gaussian kernels approximated with fixed-point representation.

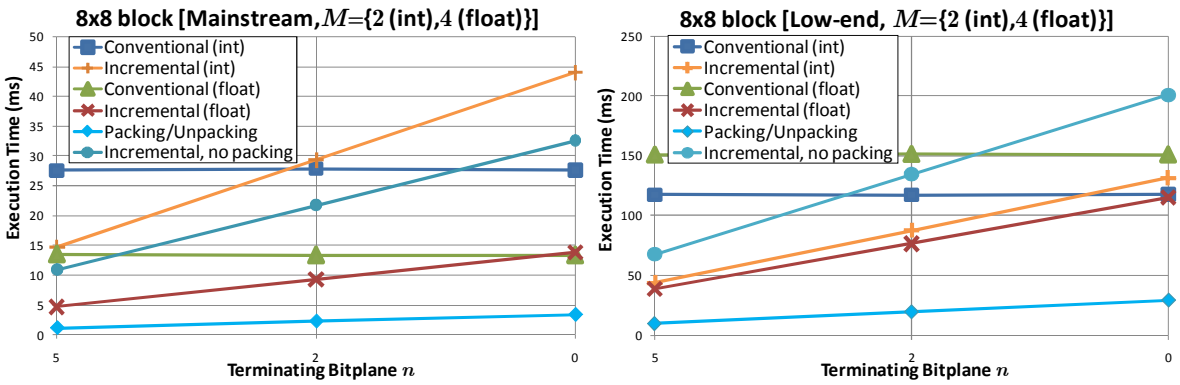


Figure 7. Block cross-correlation results.

Terminating Bitplane	Transform Decomposition SNR (dB)		2D Filtering SNR (dB)	
	4×4	16×16	12×12 **Gaussian**	8×8 **cross-correlation**
$n = 5$	16.74	19.12	18.88	18.88
$n = 2$	35.45	39.05	39.35	39.34

Table 1. Average signal-to-noise ratio for the terminating bitplanes of the mainstream-profile experiments of Figure 5-Figure 7. SNR was infinity for all cases when $n = 0$. Both conventional and incremental algorithms achieved identical SNR for each terminating bitplane.

Figure 8. Representative output frame for terminating the computation at $n = \{5, 2, 0\}$ bitplanes (shown from left to right) for the 12×12 Gaussian filtering.

Analysis of execution efficiency: The experiments summarized in Figure 5-Figure 7 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 behavior. The two profiles analyzed lead to the following generic rules for the proposed approach:

(i) Representations with larger bitwidth are advantageous for the proposed approach because they increase the packing capability, as shown in the results of Figure 6 and Figure 7.

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

(iii) When the packing capability (M) 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 5(b). Conversely, if M is high, the proposed approach becomes very efficient, *unless* if it uses a representation that is not fast in the implementation hardware (e.g. low-end profile of Figure 6 with floating-point representation).

(iv) 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 5(a)]. The exception to this rule is when high packing capability is achieved using a fast numerical representation in the utilized hardware, as seen in the mainstream profile of Figure 5(a).

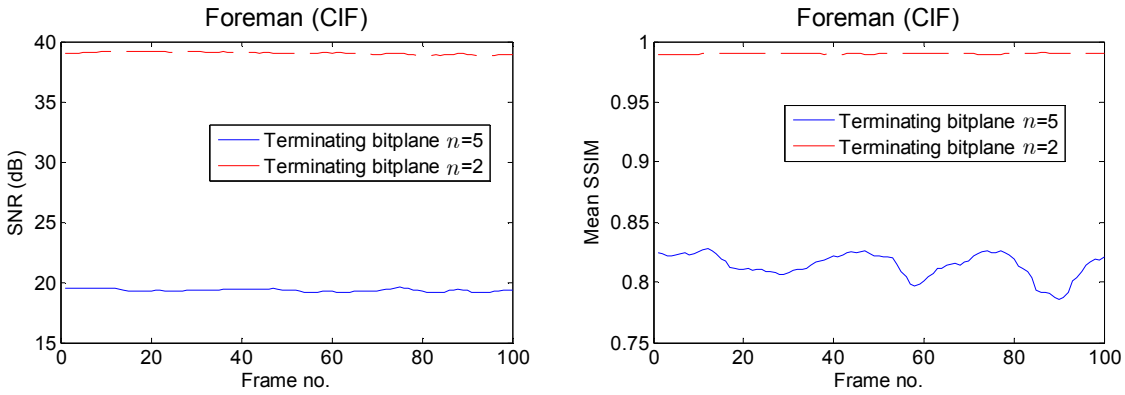
(v) The average execution time of the proposed approach is increasing linearly when the source is processed with increased precision (lower terminating bitplanes). This contrasts with the conventional approach that requires constant execution time regardless of the input precision. Once two increments 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.

Analysis of visual quality: Identical SNR results were obtained for both conventional and incremental algorithms in all cases (Table 1). Importantly, SNR per frame is monotonically increased when processing more increments (lower terminating bitplanes). An example is given in Figure 9(a) 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* [23] using the related Matlab source code[6] with the suggested parameter settings. We used the Y-frames of each sequence for this purpose and provide an example in Figure 9(b) for the transform decomposition. Indeed, the comparison between Figure 9 (a) and

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

(b) shows that even though a significant drop occurs in SNR, the output results are visually meaningful since the mean SSIM remains around 0.8.



(a) (b)

Figure 9. Frame-by-frame comparison for the reconstruction of the incrementally-computed 4×4 transform decomposition of Figure 5(a); (a) SNR comparison; (b) MSSIM comparison. Only the first 100 frames are shown. For terminating bitplane $n = 0$ SNR is infinite and the MSSIM is one.

As a final remark, it is important to emphasize 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.

B. Incremental Block Matching Experiments

The average execution times obtained for the block matching algorithms are shown in Figure 10. The corresponding PSNR results are shown in Table 2. We present the case of $C = W = 16$ for both profiles. 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 case, 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 10).

The PSNR results of Table 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 0.6dB 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 0.2dB loss of prediction efficiency at full precision ($n = 0$). We used $W_{\text{spiral}} = 9$ in the mainstream profile and $W_{\text{spiral}} = 8$ 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.

Although the results of Figure 10 can be viewed as a fast block matching algorithm incurring some loss in prediction performance, this is not the novelty of our proposal; instead, our approach presents *successively-refined precision of block matching with additional computation as more bitplanes are processed*, without the need to re-compute the result for each new increment. This enables the arbitrary termination of block matching per frame when delay constraints are met (or when resources suddenly become unavailable) and retaining the vectors of all blocks with the already-computed precision.

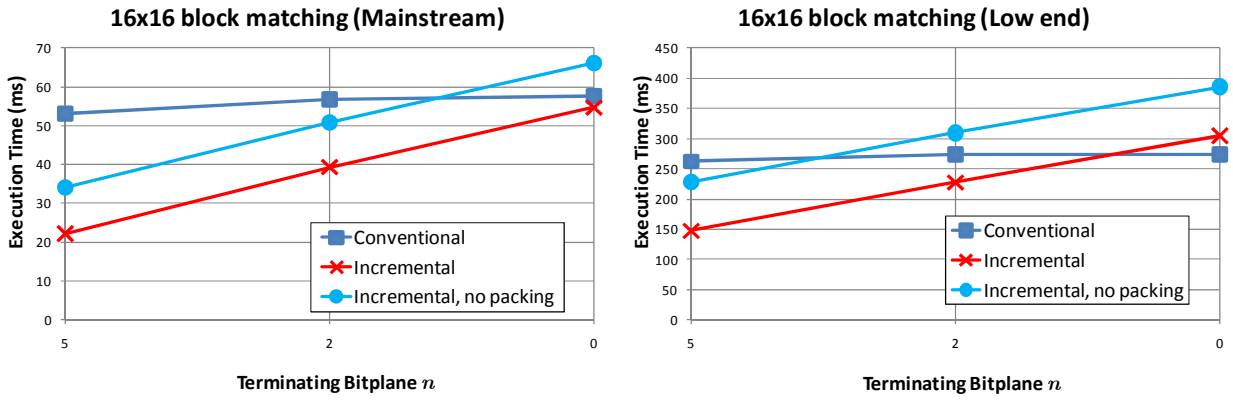


Figure 10. Block matching results.

Terminating Bitplane	Mainstream profile PSNR (dB)		Low-end profile PSNR (dB)	
	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 2. Average peak-signal-to-noise ratio for the terminating bitplanes of the experiments of Figure 10.

VII. Applications: Region-of-Interest Computation, Scheduling and Energy-Distortion Tradeoffs

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 B presents results with a real-time scheduling framework, while Subsection C presents indicative results for the energy-distortion tradeoffs 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.

A. Region-of-Interest based Incremental Computation of Image Processing Tasks

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 11 that focuses on the presenters face and hands region. The 4x4 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 $n = 5$ (first increment only). The average execution times per frame were: 5ms for $n = 5$, 6ms for $n = 2$ and 7ms for $n = 0$. Incremental computation without the ROI required 5ms, 10ms and 16ms, respectively. Conventional (non incremental) computation required 15ms for all cases since the execution time does not scale down with decreased precision. Indicative visual results of this process are shown in Figure 11 by reconstructing the video from each calculated decomposition.

Figure 11. ROI-based incremental transform decomposition. An example frame with terminating bitplanes 5, 2, 0 is shown (from left to right).

B. Time-driven Computation of Image Processing Tasks

Conventional real-time software for image processing tasks operates under worst-case assumptions, e.g. see [5]. 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 12. In order to implement this design, we have used the cross-platform OpenMP framework [24] where two independent threads (timer and main thread) are concurrently executed. The two threads share the common memory element flag_int to realize the signaling: when flag_int is set to true by the timer thread, the application thread terminates the processing of the current frame and resets flag_int to false.

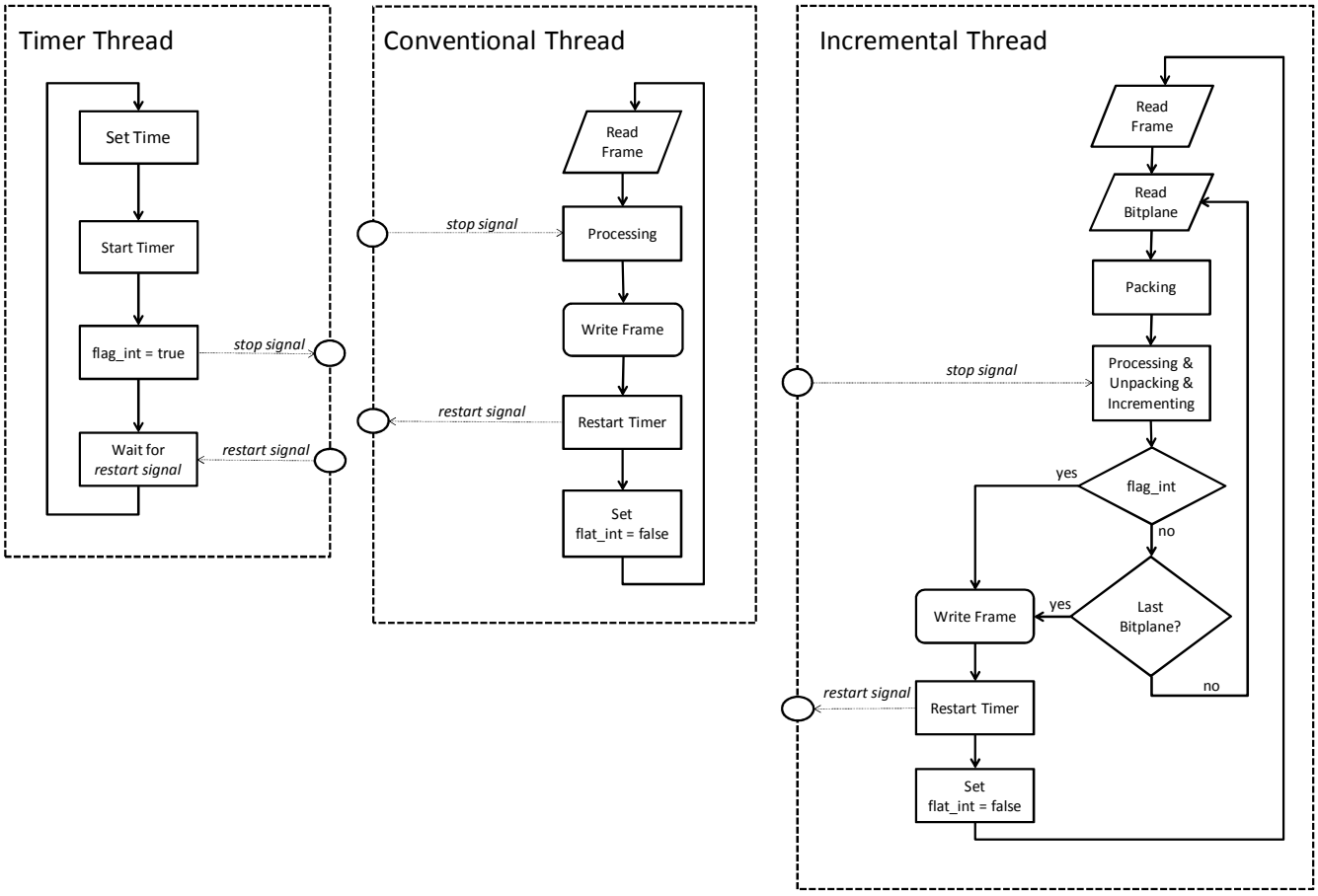


Figure 12. Time-driven computation of image processing tasks. The timer thread sends the *stop signal* to the application thread in order to terminate their execution for each frame. The application thread initiates the timer thread by the *restart signal*. The signaling is achieved via checking and setting/resetting flag_int.

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: *(i)* "regular-variability" scheduling, where A=100% of the average frame completion time for each task and D=30% of A, and *(ii)* "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: *(i)* 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); *(ii)* the percentage of *fully-completed frames*; these are fully-covered frames *and* with the result computed at full precision. 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.

Transform Decomposition				
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
Conventional	33.90%	66.10%	42.99%	57.01%
Incremental	0.19%	91.06%	5.61%	81.86%
2D Filtering				
Measurement	Uncovered	Fully-completed	Uncovered	Fully-completed
Conventional	4.41%	95.59%	6.31%	93.69%
Incremental	0.06%	99.87%	0.56%	97.59%
Block Matching				
Measurement	Uncovered	Fully-completed	Uncovered	Fully-completed
Conventional	75.24%	24.76%	79.09%	20.91%
Incremental	0.10%	76.85%	10.88%	67.58%

Table 3. 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).

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 artifacts observed are given in Figure 13. 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 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 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.

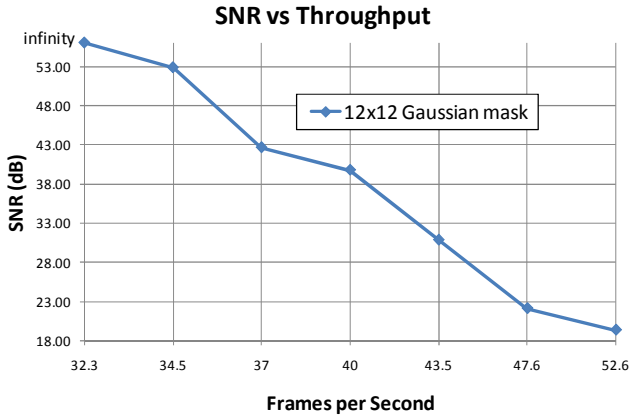
Figure 13. Visual example of video frame; from left to right: fully-completed frame, uncovered frame, covered frame but not fully completed (i.e. the result is not computed to full precision).

In a second scheduling experiment, we want to explore the throughput/quality tradeoffs enabled by the proposed approach via execution with fixed deadline per frame. Figure 14(a) shows typical SNR versus throughput results (in terms of 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[7], which leads to increased throughput, from 32.3fps to 52.6fps respectively, with a corresponding drop in SNR from infinity (full precision) to 19.36dB. Representative visual results are shown in Figure 8: 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 14, no uncovered frames were produced, i.e. there were no sudden blanks in the filtered video apart from the gradual quality reduction. This

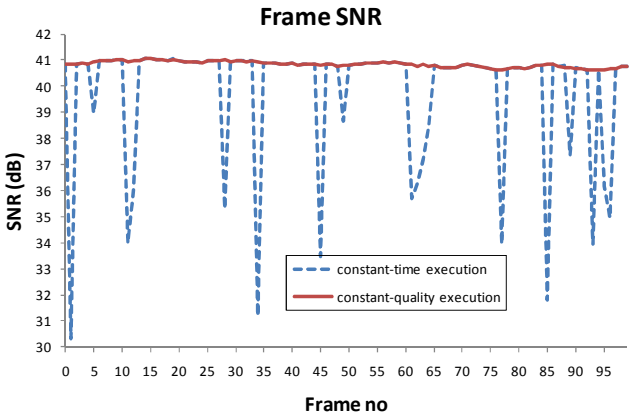
[7] Notice that the scheduling deadline includes I/O time, therefore the scheduling deadlines are higher than the timing measurements of Figure 6 that are reporting only the average computation time per frame.

straightforward 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 6) where constant-quality processing is achieved but the execution time per frame can vary. The SNR results per frame shown in Figure 14(b) 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.



(a)



(b)

Figure 14: (a) Experimental results on mainstream profile with reduced scheduling deadline; (b) SNR per frame for constant-time execution versus constant-quality execution (example with 100 frames from the "Foreman" sequence).

C. Energy-Distortion Efficiency of Software-based Incremental Computation for Real-time Video Processing on the xo-laptop

In this experiment, we use the on-board camera of the xo-laptop to capture still images in raw YCbCr format (640x480 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[8] to periodically read the xo-laptop's battery status during the algorithm execution. Our goal is to measure the power-level reduction

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

when computing the high-pass filtering with different accuracies (in terms of terminating bitplane n). To this end, we switched the xo-laptop to terminal mode and converted the monitor to low-power (reflective) mode [22]. Live image capturing was realized with the gstreamer framework [22]. 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 15 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 customizations. Additional techniques, such as platform-specific optimization to reduce the power consumed by the image capturing process, could provide further improvements.

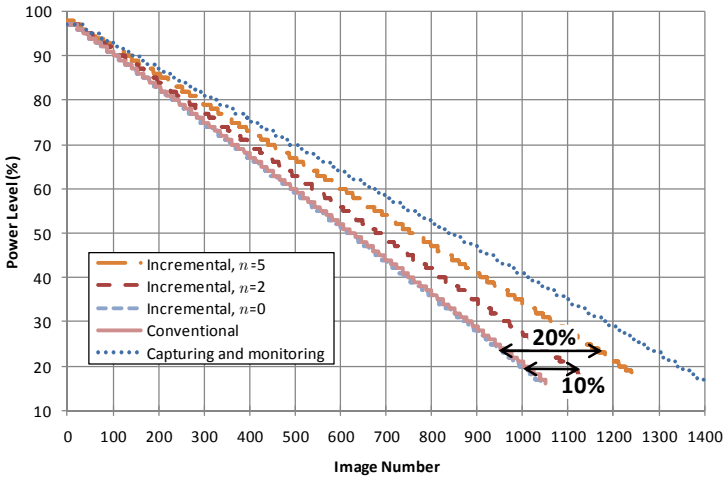


Figure 15. Measured battery power-level reduction versus images captured and processed with different precision (different terminating bitplanes n) in the xo-laptop. The dotted line indicates the power-level reduction when operating the frame capturing and battery monitoring only (without any processing).

VIII. CONCLUSION

We present a novel software framework for image processing tasks that combines packing of input bitplanes in order to process multiple increments of the input together and increment previously-computed results. The resulting framework is validated using transform decompositions, 2D 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 of this paper 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-per-second?*' changes to the best-effort ("analog world") approach of: '*What is the achieved quality when this task is performed in X frames-per-second?*'. Similarly, the power-distortion results demonstrate 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. Future research on better exploiting the packing capabilities of certain integer or floating-point representations [25], as well as system-specific customizations of the available software [15], such as computing using the graphics processing units, may permit further improvements in throughput and energy scalability with reduced output precision. Finally, the proposed framework deals with linear or quadratic operations, but the possibility of application to other (non-linear) types operations is non-trivial and requires further research.

ACKNOWLEDGEMENT

The authors thank the Associate Editor, Prof. Evans, for his meticulous handling of the review process and the six anonymous reviewers for their constructive remarks.

REFERENCES

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

[2] A. J. C. Bik, D. L. Kreitzer, and X. Tian, "A case study on compiler optimizations for the Intel Core 2 Duo processor," *Int. J. Parallel Prog.*, vol. 36, pp. 571-591, Apr. 2008.

[3] M. Macedonia, "Power from the edge [Apple video iPod]" *IEEE Computer*, vol. 38, no. 12, pp. 123-125 Dec. 2005.

[4] V. K. Goyal, and M. Vetterli, "Computation-distortion characteristics of block transform coding," *Proc. IEEE Int. Conf. on Accoust., Speech, and Signal Proc.*, vol. 4, pp. 2729-2732, April 1997.

[5] J. Valentim, P. Nunes and F. Pereira, "Evaluating MPEG-4 video decoding complexity for an alternative video complexity verifier model," *IEEE Trans. Circuits and Syst. for Video Technol.*, vol. 12, no. 11, pp. 1034-1044, Nov. 2002.

[6] B. Zeng, *et al*, "Optimization of fast block motion estimation algorithms," *IEEE Trans. Circ. and Syst. Video Technol.*, vol. 7, no. 6, Dec. 1997.

[7] K. Lengwehasatit and A. Ortega, "Scalable variable complexity approximate forward DCT," *IEEE Trans. Circ. and Syst. Video Technol.*, vol. 14, no. 11, pp. 1236-1248, Nov. 2004.

[8] D. Turaga, M. van der Schaar, and B. Pesquet-Popescu, "Complexity scalable motion compensated wavelet video encoding," *IEEE Trans. Circ. Syst. Video Technol.*, vol. 15, no. 8, Aug. 2005.

[9] S. H. Nawab, A. V. Oppenheim, A. Chandrakasan, J. Winograd, J. T. Ludwig, "Approximate Signal Processing," *J. of VLSI Signal Process.*, vol. 15, no. pp. 177-200, 1997.

[10] W. Yuan and K. Nahrstedt, "Practical voltage scaling for mobile multimedia devices," *ACM Internat. Conf. Multimedia*, pp. 924-931, 2004.

[11] E. Akyol and M. van der Schaar, "Complexity model based proactive dynamic voltage scaling for video decoding systems," *IEEE Trans. on Multimedia*, vol. 9, no. 7, pp. 1475-1492, Nov. 2007.

[12] J. Winograd and S. H. Nawab, "Incremental refinement of DFT and STFT approximations," *IEEE Signal Process. Letters*, vol. 2, no. 2, pp. 25-27, Feb. 1995.

[13] Y. Andreopoulos and M. van der Schaar, "Incremental refinement of computation for the discrete wavelet transform," *IEEE Trans. on Signal Process.*, vol. 56, no. 1, pp. 140-157, Jan. 2008.

[14] Y. Andreopoulos and I. Patras, "Incremental refinement of image salient-point detection," *IEEE Trans. on Image Process.*, vol. 17, no. 9, pp. 1685-1699, Sept. 2008.

[15] http://www.ee.ucl.ac.uk/~iandreop/ORIP.html.

[16] D. Anastasia and Y. Andreopoulos, "Software designs of image processing tasks with incremental refinement of computation," *IEEE Workshop on Signal Process. Systems* (SIPS-09), Tampere, Finland.

[17] A. Kadyrov and M. Petrou, "The "Invaders" algorithm: range of values modulation for accelerated correlation," *IEEE Trans. Pattern Anal. Machine Intel.*, vol. 28, no. 11, pp. 1882-1886, Nov. 2006.

[18] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity transform and quantization in H.264/AVC," *IEEE Trans. on Circ. Syst. for Video Technol.*, vol. 13, no. 7, July 2003.

[19] D. Marpe, T. Wiegand, and S. Gordon, "H.264/MPEG4-AVC fidelity range extensions: tools, profiles, performance, and application areas," *IEEE Int. Conf. on Image Proc.*, vol. 2, pp. 870-873, Sept. 2005.

[20] H. Sun and W. Kwok, "Concealment of damaged block transform coded images using projections onto convex sets," *IEEE Trans. on Image Process.*, vol. 4, no. 4, pp. 470-477, April. 1995.

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

[22] http://wiki.laptop.org/

[23] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Trans. on Image Process.*, vol. 13, no. 4, pp. 600-612, April 2004.

[24] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP*, MIT Press, 2007, ISBN-13: 978-0-262-53302-7.

[25] D. Anastasia and Y. Andreopoulos, "Linear image processing operations with operational tight packing," *IEEE Signal Processing Letters*, to appear.