1-1-2011

Analysis and architecture design of scalable fractional motion estimation for H.264 encoding

Jasmina Vasiljevic
Ryerson University

ANALYSIS AND ARCHITECTURE DESIGN OF SCALABLE FRACTIONAL MOTION ESTIMATION FOR H.264/AVC ENCODING

by

Jasmina Vasiljević

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science (MASc)

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2011

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Acknowledgments

I would like to thank my wonderful and strong family; my mother who selflessly encouraged, advised and fought for me every step of the way; my father who instilled in me an unquenchable thirst for science from a very young age; my brother who leads and inspires by example, continuously reminding me that no goal is out of reach; and finally to my soul mate, best friend and my *ljljic* – Davor Čapalija. I would also like to thank Professor Andy Ye for taking me on as a student and his help during the course of my study.

Abstract

ANALYSIS AND ARCHITECTURE DESIGN OF SCALABLE FRACTIONAL MOTION ESTIMATION FOR H.264/AVC ENCODING

Jasmina Vasiljević

Master of Applied Science (MASc)

Department of Electrical and Computer Engineering

Ryerson University

Fractional Motion Estimation (FME) is an important part of the H.264/AVC video encoding standard. FME can significantly increase the compression ratio achievable by video encoders while improving video quality. However, it is computationally expensive and can consist of over 45% of the total motion estimation runtime. To maximize the performance and hardware utilization of FME implementations on Field-Programmable Gate Arrays (FPGAs), one needs to effectively exploit the inherent parallelism in the algorithm. In this work we explore two approaches to FME algorithm parallelization in order to effectively increase the processing power of the computing hardware. The first method is referred to as vertical scaling and the second horizontal scaling. In total, we implemented six scaled FME designs on a Xilinx Virtex-5 FPGA. We found that our best vertically scaled FME design exhibited a speedup of *8x* over the horizontally scaled designs. Additionally, we conclude that scaling vertically within a 4x4 pixel sub-block is more efficient than scaling horizontally across several sub-blocks. As a result we were able to achieve higher video resolutions at lower hardware resource costs. In particular, it is shown that the best vertically scaled design can achieve 30 fps of QSXGA (2560x2048) video using 4 reference frames with only 25.5K LUTS and 28.7K registers.

Table of Contents

List of Tables

List of Equations

List of Figures

Chapter 1 – Introduction

Motion Estimation (ME) is a video compression algorithm that removes spatial and temporal data redundancy between frames by using motion vectors (MVs) to track the motion of objects within video footage. Unlike previous encoding algorithms, the H.264/AVC standard offers the use of Fractional Motion Estimation (FME). This is an engine which fine-tunes MVs to sub-pixel granularities in order to achieve an enhanced video quality and a better compression ratio which reduces transfer stream bit-rates as well as the amount of video memory storage. Compared with previous video compression standards, such as H.263 and MPEG-2, the H.264/AVC encoding standard can achieve 49% and 64% bit-rate reductions respectively [1]; as well as a compression rate of 50:1 [53].

There exists a quantitative formal video quality measurement called the Peak Signal to Noise Ratio – $PSNR_{dB}$. The $PSNR_{dB}$ depends on the mean squared error (MSE) between an impaired video frame and the original, based on the following equation:

$$PSNR_{dB} = 10\log_{10}\frac{(2^n - 1)^2}{MSE}$$

Equation 1 - Peak Signal to Noise Ratio

where n is the number of bits per pixel and MSE is the mean squared error [2]. $PSNR_{dB}$ can be calculated quickly and easily and is a widely used metric to compare the quality of compressed and decompressed video images. It has been measured that the FME can improve video quality by +4 dB [3].

The FME algorithm is an essential part of the H.264/AVC video compression standard. However, the FME algorithm is computationally expensive. Due to the high computing demand, many hardware architectures have been proposed to accelerate the computation of the FME algorithm. The large range of H.264/AVC engines vary from a free low-end Xilinx soft-core[4], to high-priced, high-performance designs aimed at the broadcasting industry, targeting custom FPGA/ASIC systems [5]. Further, the FME has been implemented on a variety of target architectures, for example: ASIC/FPGA [1][3][6]-[17], hybrid hardware/software systems [18] and GPUs[18][20].

The need for scalable high-performance encoding engines with efficient use of hardware resources have been previously recognized as a need [7][8][9][18][21][22]. These implementations use systolic arrays in order to achieve flexible scalable designs. They succeeded in providing a high degree of hardware utilization as well as data throughput rate. The parameterizable designs [7][8][9] allowed for setting of a variable which controls the size of the pixel block to be processed. While these designs are capable of handling a range of variable pixel array sizes, they require the user to commit to particular block dimensions at compile time. These works do not address the scalability and utilization challenges which arise when processing all 7 variable sized blocks at the same time. In addition, some designs evaluate MVs at a quarter-pixel resolution [18] in a single step whereas the refinement can be divided into two stages. A two stage refinement would involve Unnecessary computation can be saved by first evaluating the best half pixel, followed by refining it to a quarter pixel resolution.

Certain implementations focused on optimizing the pixel interpolation [23], while others on the distortion evaluation units [24]. Further, approximation algorithms have been implemented to simplify the computational complexity FME by abstracting away the need for

pixel interpolation [25], although they do suffer a video quality penalty.

A number of works have implemented full support of variable block sizes (VBS) [1][3][10][16]. They are capable of processing all 7 of the variable sized blocks. Processing of VBS gives rise to various hardware utilization problems and data flow trade-offs. In addition, these implementations are not scalable and easily parameterizable. They are capable of processing 4, 8 or 16 pixels from the same row of a macroblock, thus possessing horizontal parallelism. We found however, that expanding parallelism only along the row can potentially degrade hardware utilization. In this work, we explore an increase in parallelism along a column of pixels as well.

In this work, we perform a scalability study of FME with the goal of gaining performance while minimizing the use of additional hardware resources. Our target architecture is an FPGA which is capable of providing custom fine-grain optimizations with regards to parallelization. In our work, we describe the design and implementation of the Interpolation Engine (IE), as well as the distortion computation units (Processing Units - PUs). We implement a parameterizable, scalable design, capable of processing all 7 types of variable block sizes. We implement and analyze a total of six scalable FME architectures, with our formulated analysis method. In addition, we present results for both, the IE and the PU together, and analyze their interaction with respect to data flow, hardware utilization and data redundancy in order to identify the optimal high-performing FME engine.

Due to the high computing demand, many hardware architectures have been proposed to accelerate the computation of the FME algorithm. Most of the architectures, however, have been implemented in Application Specific Integrated Circuit (ASIC) technology. Except for the work in [15][26], which implements a non-scalable version of FME on FPGAs, limited information

exists on performance and algorithm-architecture mapping exploration of the FME onto reconfigurable technologies. In this work, we perform a scalability study of FME on FPGAs with the goal of gaining performance while minimizing the use of additional resources.

Scalability is a concept frequently used to indicate the *quality* of parallel systems. Many engineering issues emerge when scaling and implementing an algorithm. Many of these issues are best dealt with at the algorithmic level, rather than by manually modifying the code for each individual implementation. The fastest scaled implementation can depend on many factors, such as target architecture, amount of parallelism, software environment and memory hierarchy. A good understanding of the concept of scalability can be used to select the best algorithm-architecture combination for a problem. By determining the performance of an algorithm with a particular architecture and varying amounts of resources, one can determine the maximum speedup which can be obtained for a minimum amount of additional recourses.

This work provides an insight into the influence of the algorithm on the architecture and vice-versa to enable us to understand the scalability of the FME algorithm-architecture pair. We develop and present an empirical approach in order to quantify and evaluate two different scalability approaches and a total of six scaled FME implementations. We also present our new innovative approach to FME scalability, vertical scaling, and compare it against the traditional implementation of horizontal scaling.

The investigation of scalable FME implementations on FPGAs is particularly important since the programmability of FPGAs encourages design reuse and can greatly enhance the upgradability of digital systems. Scalability eases the creation of highly flexible FPGA-based encoding systems which can accommodate a multitude of existing standards as well as support the emergence of new standards. An efficient scalable FME implementation can be incorporated

into a single FPGA solution targeting low cost, low-resolution applications as well as multiple FPGA designs targeting high performance high-resolution applications.

1.1 Thesis Objectives

The objective of this work is to develop and implement a scalable FME engine. We aim to analyze the trade scalability trade-offs and quantify them in order to find a suitable scalability approach.

1.2 Thesis Contributions

In this work, we perform a scalability study of FME by increasing parallelism along a column of pixels as well as along a row of pixels. We define a set of overhead functions associated with the different algorithmic and architectural characteristics in order to quantify the scalability of the FME engine as a parallel system. Based on the results, we design and implement a scalable high-performance FME algorithm platform. We isolate the algorithmic and architectural overheads by examining their influence on the processing time and the use of hardware resources. Our design is then implemented on an FPGA which is capable of providing custom fine-grain optimizations with regards to parallelization. In summary, our contributions are four-fold:

1. Introduced a vertical scaling approach
2. Developed hardware utilization analysis metrics for scalable FME designs
3. Performed direct quantitative comparison between vertical and horizontal FME implementations

4. Implemented a parameterizable design, changeable at compile time to instantiate various scalable FME architectures

We describe the design and implement the Interpolation Engine (IE) and the distortion computation units (referred to as Processing Units PUs), capable of processing all 7 types of variable block sizes. The results for both the IE and the PU are presented. Their interaction between data flow, hardware utilization and data redundancy are analyzed in order to identify the best configuration for high-performing FME engines. Designs that implement a more complete H.264/AVC system typically choose an FME architecture that suits a pipeline work balance between the IME and FME. In order to reduce the clock cycles required to process the 41 sub-blocks. So far, designers have chosen a faster implementation by increasing parallelism horizontally. We argue and show that increasing parallelism vertically is significantly more efficient in terms of hardware resources and achieved throughput, and therefore urge designers to reconsider their choices taking into account analysis presented in this work. Looking at our scalability curve and cost-performance product derived in Chapter 5, we conclude that in terms of resource use the designer is better off choosing any of the three vertically scaled designs over the horizontally scaled versions. While previous works have implemented a more complete H.264/AVC system, our work was developed based on a bottom-up approach, starting at understanding all of the FME design trade-offs. Thus, in future work, the relationship between integer motion estimation (IME) and FME can be efficiently exploited through the application of the analytical methods developed in this work.

Finally, a four page conference paper version of this work was published in [54], and a journal was published in [55].

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 provides a summary of background information on video compression as well as a few of the motion estimation features which are specific and new to the H.264/AVC standard, including the motivation behind our work. Our newly developed scalable FME architecture is presented in Chapter 3, and Chapter 4 discusses the design space exploration where we formulate our design evaluation metrics. Chapter 5 presents details of the experimental setup and implementation results, and then moves on to describe our empirical study of the specific scalability challenges and analysis of performance trade-offs. Finally, Chapter 6 concludes by summarizing the presented work and related research, along with future work which can build upon the current results.

Chapter 2 – Background and Motivation

This chapter describes video and compression related background, including technical details required for estimation of computational complexity.

2.1 Introduction to Video

Many digital video formats exist, varying in their target resolutions, colour space, chrominance sub-sampling ratios and signal sampling frequencies. Figure 1 shows a sample map of the rich variety of available formats.

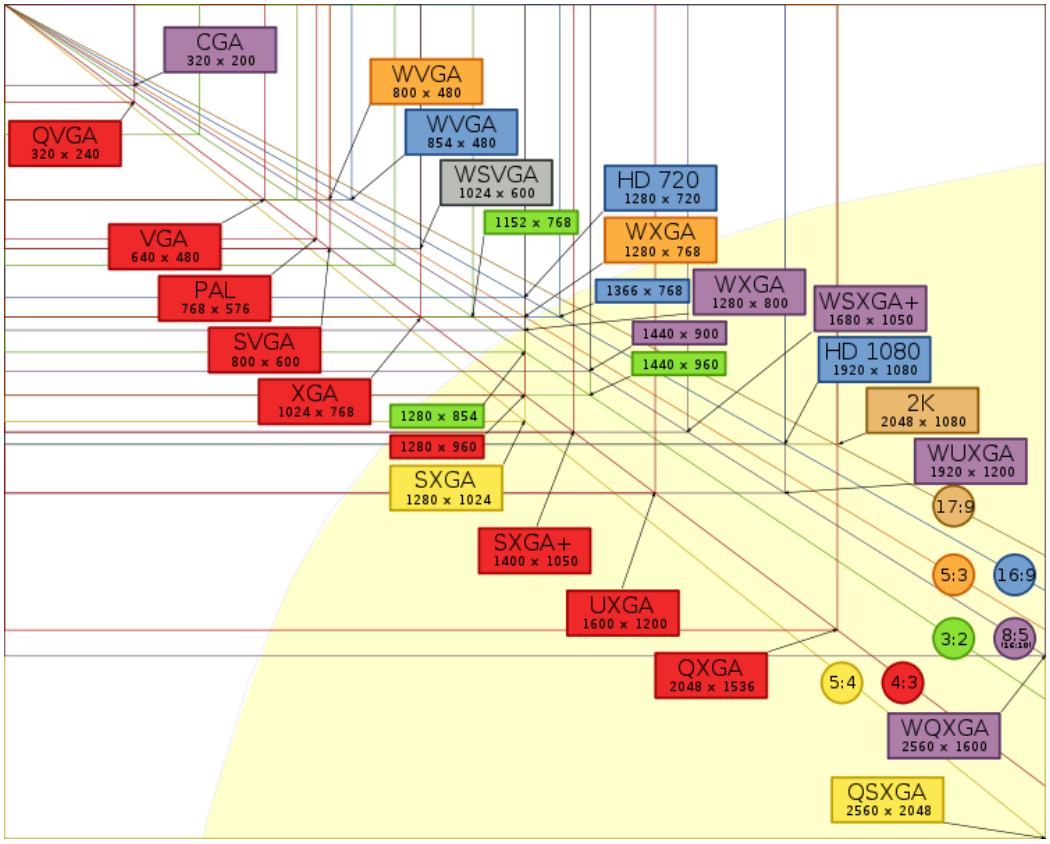


Figure 1 - Various Video Formats and Resolutions

The mathematical representation of a set of colours is called a colour space [27]. Popular models are RGB, used in computer graphics, YIQ, YUV or YCbCr, popular in video broadcasting systems, and CMYK, used in colour printing. The YCbCr format has one luminance (Y) and two chrominance components (Cb and Cr). The luminance and chrominance components are quantized as 8-bit data values. The 8-bit luminance value represents a single unsigned integer byte of data, ranging from 0 to 255, where 0 corresponds to a purely black pixel and 255 to a purely white pixel [27].

In this work we deal with the luminance component only which is what motion estimation engines utilize for encoding. This design choice stems from the observation that the human eye is more sensitive to changes in brightness then colour. What this means is that when

humans take in the view of an image as a whole, a lack of colour does not adversely affect the ability to discern motion.

2.2 Digital Video Compression

Digital video compression is performed by identifying and removing data redundancy in order to reduce transmission bandwidth and overall file size. This reduction can be achieved on two separate fronts: spatial, across a search region of a single frame, and temporal, across multiple frames.

For example, in a single frame, spatial redundancy can be performed on a large consistent background shade of blue sky. In this case the large repetitive area can be compressed through the use of various discreet cosine transformations which map an image in terms of its light or colour intensities [28]. Since this type of compression does not involve motion estimation, this topic is not examined further.

Temporal compression is achieved through the removal of data repeated over a given sequence of frames, for example background objects which are not likely to change over the course of a given footage. Their redundancy can be taken advantage of by transmitting the object's motion from frame to frame, which can result in considerable data compression. The recording of motion is done in the form of MVs. Consequently, the motion estimation algorithm is the process of deriving suitable MVs which best describe the movement of objects from one frame to the next.

Each video frame is processed in terms of macroblocks (MBs), which are a 16x16 pixel arrays. A frame that is in the process of being encoded is called a current frame. The goal of the

ME engine is to describe the current frame in terms of MBs already in the current frame or within a set of reference frames. The reference frames may come before or after the current frame as they are all buffered inside a decoder and can be referenced as needed. The ME algorithm scans potential reference MBs in order to find the most suitable one for the current MB. Once the ME engine decides on the optimal match, it encodes the current block of pixels as an MV. The MV is calculated as the spatial and temporal displacement between the current and reference MBs. Sample MVs are shown in Figure 2.

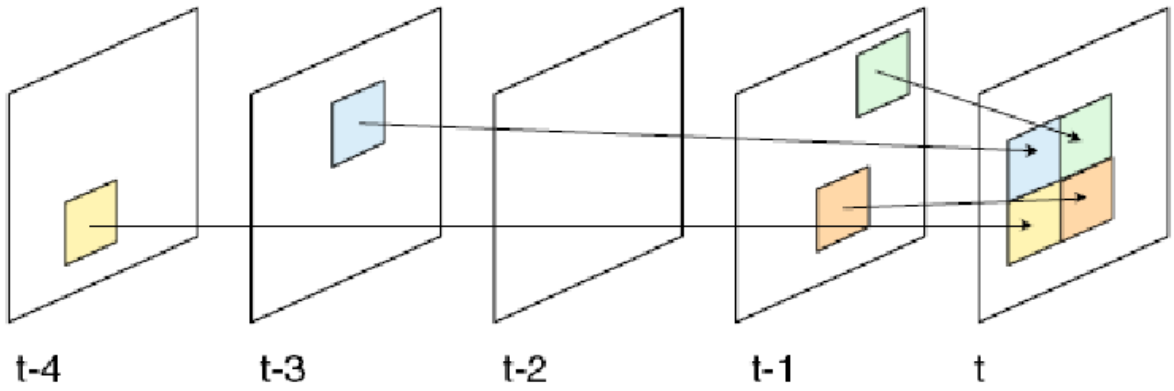


Figure 2 - Motion Estimation

Notable new features of the H.264/AVC compression standard include Variable Block Size Motion Estimation (VBSME) and Multiple Reference Frame (MRF) searching. Both of these features increase the computational complexity by expanding the search space and hence the required calculations. MRF enables the engine to capture periodic motion across a number of reference frames and from it extract data redundancy. VBSME has the capability of detecting motion of smaller objects but can increase the transmission bandwidth in order to deliver a higher quality stream and finer granularity of motion.

Currently, these techniques have been widely implemented as the basis of the H.264/AVC video standard. With the optimization and enhancement of these techniques, as well as others,

the standard has been able to achieve compression ratios as high as 60:1 as well as improved video quality [28]. The development of H.264/AVC was done by the International Telecommunications Union – Telecommunications Standardization Sector (ITU – T) Video Coding Experts Group (VCEG) and the International Organization for Standardization (ISO) MPEG committee.

2.3 Variable Block Size Motion Estimation (VBSME)

The technique of matching MBs was used in previous video compression standards, such as MPEG-2. However, the new H.264/AVC standard introduces VBSME which employs subblocks smaller than the MB in order to detect finer granularity of motion within a video sequence. For example, Figure 3 shows the same video frame encoded with three different thresholds of subblock granularity.

Figure 3 – Encoding a Singe Frame with Different VBSME settings

In general, larger blocks are well suited for representing large areas of consistent motion and patterns, such as backgrounds. Analysis and experimentation has concluded that a 16 pixel by 16 pixel square MB size is the best compromise between computational complexity and

accuracy [29]. On the other hand, an MB is less likely to be able to precisely describe fine grain objects which could move in several different directions. Reducing the subblock size increases the likelihood of all pixels within that subblock having uniform velocity, and hence the MV being a good fit for the motion of the block. On the other hand, finer granularity comes with an increase in the amount of required computations as well as the total final MV count contained in a single frame, which in turn can increase bandwidth.

The H.264/AVC standard specifies that the VBSME takes in an MB as input and subdivides the block six times to produce a total of 41 subblocks shown in Figure 4. Motion estimation is then performed on each subblock across multiple reference frames in order to produce 41 integer MVs.

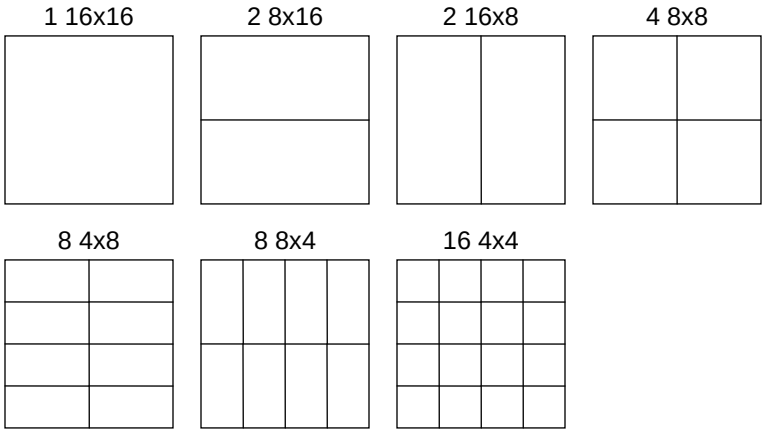


Figure 4 - MB and Subblocks Sizes in VBSME

2.4 Search Windows and Multiple Reference Frames

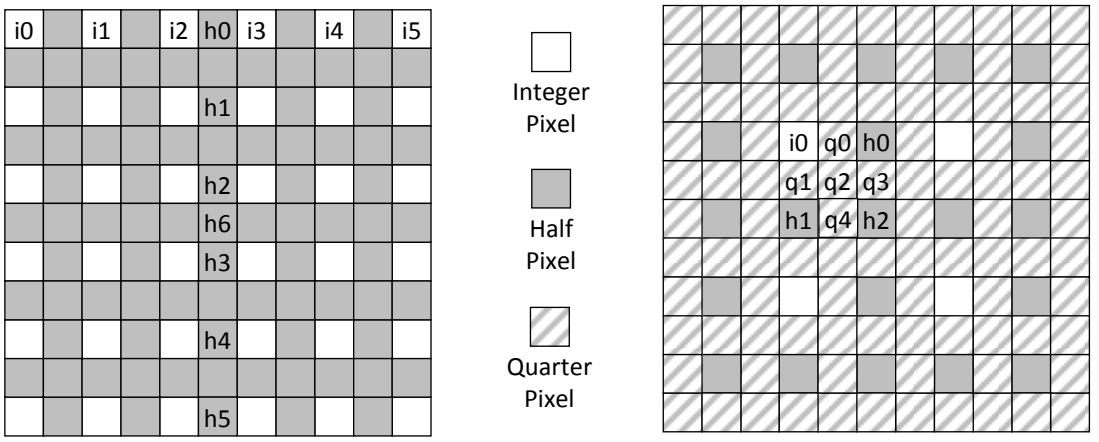
When searching for possible matches for an MB, the integer motion estimation (IME) searches through all of the provided reference frames within a designated search window. A

common search window size is 48 pixels high and 63 pixels across, with a corresponding search range of [-16, +16] vertically and [-24, +23] horizontally, centered on the position of the original current MB.

Further, since the H.264/AVC specification allows for the encoder to search within multiple reference frames (MRF), the MB is compared against one search window within each frame. Commonly, industry applications support the use of at least 4 reference frames, but the specification for allows up to 16. As such, the real-time design requirement used in this thesis is 4 reference frames for each target video resolution. The advantage of MRF is that it equips the encoder with the ability to potentially detect periodic motion. Optimal MB matches can be located within frames that are further away, prior to the immediately previous frame, where periodic motion occurs. This technique can increase the number of acceptable matches for a current MB, which in turn improves the compression ratio, as well as the video quality [30]. However, this advantage significantly increases the amount of computations required by the encoder.

2.5 The Fractional Motion Estimation Algorithm

The input to FME consists of a set of motion vectors from the integer VBSME algorithm [31].



(a) Pixels at Half Pixel Resolution

(b) Pixels at Quarter Pixel Resolution

Figure 5 - Half and Quarter Pixels

FME is used to refine the integer MVs at two finer granularities: at half pixel resolution and quarter pixel resolution. In particular, half pixels are created for reference frames based on Equation 2, where each pixel is defined as the weighted average of a row of six neighboring integer pixels or a column of six neighboring half pixels, as shown in Figure 5 - (a). Quarter pixels, on the other hand, are defined by Equation 3, where each quarter pixel is calculated as the average of two of its neighboring integer/half pixels as shown in Figure 5 - (b).

$$h_0 = \frac{\left(i_0 - 5 \times i_1 + 20 \times i_2 + 20 \times i_3 - 5 \times i_4 + i_5\right)}{32}$$

$$h_6 = \frac{\left(h_0 - 5 \times h_1 + 20 \times h_2 + 20 \times h_3 - 5 \times h_4 + h_5\right)}{32}$$

Equation 2 - Half Pixel Generation

$$q_0 = \frac{(i_0 + h_0 + 1)}{2}; \quad q1 = \frac{(i_0 + h_1 + 1)}{2}$$
$$q_2 = \frac{(h_0 + h_1 + 1)}{2}; \quad q3 = \frac{(h_0 + h_2 + 1)}{2}$$

Equation 3 – Quarter Pixel Generation

Half and quarter pixels are used to refine the integer motion vectors in two stages. First FME compares each integer motion vector to eight of its surrounding motion vectors at half pixel resolution. Here, the Sum of Absolute Transformed Differences (SATDs) is used as a metric which evaluates the suitability of the subblock. Equation 4 defines the SATD value for a 4x4 subblock. In the equation, Hadamard represents the 2-D Hadamard transform [24], *C(i, j)* represents the value of pixel (*i, j*) in the 4x4 block, *x* and *y* represent the horizontal and vertical displacement (due to the integer motion vector) of the pixel coordinates in the reference frame with respect to the pixel coordinates in the 4x4 block, *R(rx + x + i, ry + y + j)* represents the value of the corresponding pixel (*rx + x + i, ry + y + j*) in the reference frame, and *rx* and *ry* (as defined in Equation 6) represent the range of motion estimation at half pixel resolution.

The best motion vector at half pixel resolution is further refined at quarter pixel resolution based on Equation 3. Here *x* and *y* represent the displacement of the reference pixels with respect to pixels in the 4x4 block at half-pixel resolution, *rx* and *ry* (as defined in Equation 6) represent the search range at quarter pixel resolution, and the SATDs of the larger blocks are similarly defined as the sum of all SATDs of their constituting 4x4 blocks.

$$rx' = rx + x; \quad ry' = ry + y$$

$$R_M = \begin{pmatrix} R(rx'+0,ry'+0) & R(rx'+1,ry'+0) & R(rx'+2,ry'+0) & R(rx'+3,ry'+0) \\ R(rx'+0,ry'+1) & R(rx'+1,ry'+1) & R(rx'+2,ry'+1) & R(rx'+3,ry'+1) \\ R(rx'+0,ry'+2) & R(rx'+1,ry'+2) & R(rx'+2,ry'+2) & R(rx'+3,ry'+2) \\ R(rx'+0,ry'+3) & R(rx'+1,ry'+3) & R(rx'+2,ry'+2) & R(rx'+3,ry'+3) \end{pmatrix}$$

$$C_M = \begin{pmatrix} C(0,0) & C(1,0) & C(2,0) & C(3,0) \\ C(0,1) & C(1,1) & C(2,1) & C(3,1) \\ C(0,2) & C(1,2) & C(2,2) & C(3,2) \\ C(0,3) & C(1,3) & C(2,3) & C(3,3) \end{pmatrix}$$

$$M = Hadamard(R_M - C_M)$$

$$SATD = \sum_{i=0}^{3}\sum_{j=0}^{3}|M(i,j)|$$

Equation 4 – Generating the SATD Value

$$rx \in [-0.5 \quad 0 \quad 0.5] \quad ry \in [-0.5 \quad 0 \quad 0.5]$$

Equation 5 - Half Pixel Range

$$rx \in [-0.25 \quad 0 \quad 0.25] \quad ry \in [-0.25 \quad 0 \quad 0.25]$$

Equation 6 – Half and Quarter Pixel Range

2.6 The FME Architecture

The FME engine takes in as input a series of sets of 41 MVs, previously processed by an integer VBSME engine, shown in Figure 6. The 41 MVs point to the seven types of variable

sized subblocks previously shown in Figure 4. The blocks are sent to the FME engine and processed in various sized pixel array chunks per clock cycle, depending on the instantiated size of the scalable FME.

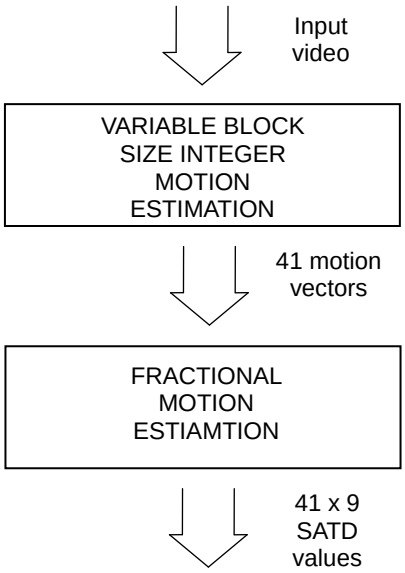


Figure 6 - Motion Estimation Block

2.7 Hardware Implementation

The following section describes two specific hardware implementation techniques used in this work.

2.7.1 Finite Impulse Response Filter

The FIR filter is the basic building block of the FME engine and is used to generate half-pixels based on a weighted sum of its neighboring six integer pixels. In this section we describe its hardware implementation. The original FIR filter weighted sum is shown in Equation 2. For the purpose of a hardware implementation, the filter was optimized by pipelining and sharing resources [3]. The logic is grouped so that the greatest common factors of each weight value are processed together. In addition, the logic is setup so that the multiplier values are a power of two and can be implemented as simple shift registers. Figure 7 shows the five stage pipelined data flow, defined by Equation 7 which is mathematically equivalent to Equation 2.

$$h_0 = \{[(i_0 + i_5) + (i_2 + i_3) \times 4 - (i_1 + i_4)] + [(i_2 + i_3) \times 4 - (i_1 + i_4)] \times 4\} \div 32$$

Equation 7 - FIR Filter

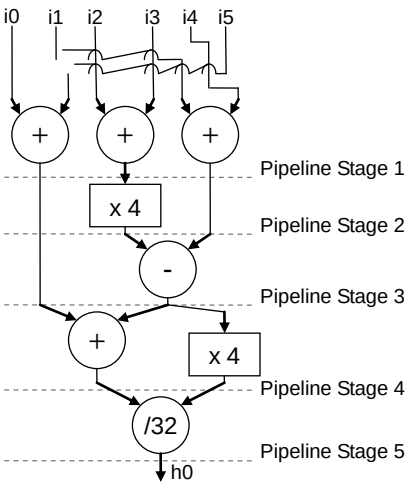


Figure 7 - Pipelined Hardware Implementation of the FIR Filter

2.7.2 The 2-D Hadamard Transform

The 2-D Hadamard transform yields a convenient and fast hardware implementation because all of its coefficients are either 1 or -1. The pixel bit vectors are transformed into a 2's compliment format. When multiplying by 1, the original pixel bit vector is preserved, and when multiplying by -1 the bit vector's most significant bit is inverted.

The original equation for the 1-D Hadamard transform is shown in Equation 8, where *a*, *b*, *c* and *d* are the four input pixels, and *had_out_m_0*, *had_out_m_1*, *had_out_m_2*, and *had_out_m_3*, are the four output pixels. The alike terms are grouped together in order to avoid redundant calculations resulting in Equation 9 which is mathematically equivalent to Equation 8. The hardware implementation is shown in Figure 8. The intermediate pipeline Stage 1 of adders ensures sharing of the like terms for the second stage of adders [24]. This way, only eight adders are required instead of 12 if the adder structure was designed corresponding to the original Equation 8.

$$
\begin{aligned}
had\_out\_m\_0 &= a + b + c + d \\
had\_out\_m\_1 &= a - b + c - d \\
had\_out\_m\_2 &= a + b - c - d \\
had\_out\_m\_3 &= a - b - c + d
\end{aligned}
$$

Equation 8 – 1-D Hadamard Transform

$$
\begin{aligned}
had\_out\_m\_0 &= (a + b) + (c + d) \\
had\_out\_m\_1 &= (a - b) + (c - d) \\
had\_out\_m\_2 &= (a + b) - (c + d) \\
had\_out\_m\_3 &= (a - b) - (c - d)
\end{aligned}
$$

Equation 9 – Hardware Optimized 1-D Hadamard Transform

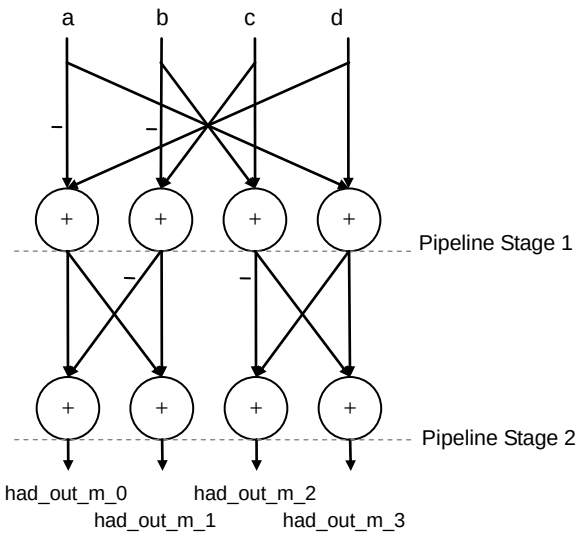


Figure 8 - The Hardware Implementation of the Optimized 1-D Hadamard Transform

2.8 Motivation

The motivation behind our work is rooted in the complexity requirements of video processing. This complexity appears on many fronts, such as large memory storage occupancy, high transmission bandwidth requirements, the number of GIPS for encoding and large amounts of needed hardware resources. Below we provide sample numbers and requirements for each of these, some corresponding to the entire H.264/AVC standard, and some for FME in particular. Next, we formulate a complexity expression with regards to the processing requirements. Our calculation is expressed in terms of *pixel comparisons*, which translates to multiple instructions, the number of which depends on the way the algorithm is coded as well as the targeted device. We also briefly explain the difference between full-search and fast-search FME algorithms. Finally, we present a brief introduction into the concept of our new FME scaling approach.

Raw digital video in its uncompressed form requires a considerable amount of storage space and transmission bandwidth. For example, uncompressed HD video requires 3.12 MB to store data belonging to a single frame. Capturing a video stream at a real-time rate of 30 fps requires 93.6 MB of storage for every second of video. This is a relatively large amount. In comparison, a Blu-ray disk with 100 Giga Bytes of storage capacity would be able to hold only 18 minutes of uncompressed HD video. Similarly, compression plays a large role in today's networked environments where limited bandwidths are present. The H.264/AVC codec can achieve a 64% bit-rate reduction [1]. To process a large amount of data in real-time video encoders are required to operate at the rate that digital video streams are captured. The processing requirement will intensify as high definition resolutions and frame rates increase in the future. This trend creates a requirement for continuously evolving processing hardware as

well as scalable solutions.

The computational requirement of the entire H.264/AVC codec, which FME is part of, has also been examined. In particular, experimental results using the Joint Video Team Reference Software (JM7.3) [32], have shown the video encoding algorithm to consume over 80 GIPS, with a baseline profile Level 2 with CIF[1] format, 5 reference frames and +/- 16SR[2] [3]. The FME has been measured to occupy 45% of the run-time in inter prediction [3]. Similarly, the computational complexity for the SDTV[3] and HDTV720p[4] standards, with 4 reference frames was measured by *iprof* software [33] (an instruction level software analyzer) to be 2470 and 3600 GIPS respectively [1], which is beyond the capability of general purpose processors, making dedicated hardware a necessity.

The FME occupies a large part of the overall H.264/AVC engine. One hardware implementation of the codec shows the FME consisting of over 400,000 gates, which amounts to 43% of the total chip resources used [1].

Next we explain the difference between full-search and fast-search FME algorithms, which is necessary to understand our formulation of the computational complexity. Hardware implementations of the FME algorithm can be classified into two types: *fast search* [34]-[43] and *full search* [1][3][10]-[16] algorithms. Fast search approaches can potentially reduce the computational complexity through algorithmic optimizations, at certain costs to video quality. These shortcuts result in irregular memory accesses because the optimizations are designed to be content dependent. This means that different video streams will result in a range of total processing times and complexities. As a result fast search is less computationally traceable,

[1] CIF resolution:352x288 in PAL
[2] SR – Search Range
[3] SDTV resolution: 856x480
[4] HDTV720p resolution: 1780x956

meaning that it is difficult to measure the exact efficiency of specific computational blocks because they will differ over various content. Full search algorithms eliminate the unpredictable data flow by fully exploring the motion estimation search space to produce higher compression ratios and video quality than fast search algorithms. The focus of this investigation is based on full search whose computational complexity is not dependent on the video content as it is fitting with our goal to quantify performance and identify the algorithmic scalability potential.

The following is our complexity analysis of the full-search half-pixel VBS-FME algorithm. We can define the running time, *T*, of the FME algorithm by calculating the number of operations that need to be performed per second, where each operation is considered to be a pixel comparison between two corresponding pixel locations in the reference and current subblock. The running time can be expresses in the following form:

$$T = P_{MB} \times i \times r_{xy} \times R_{frame} \times MB_{frame} \times fps$$

Equation 10 - The Running Time of FME

$$MB_{frame} = \frac{width_{frame}}{16} \times \frac{height_{frame}}{16}$$

Equation 11 – Total Number of Macroblocks per Video Frame

Where the complexity scales linearly with each of the following variables:

P_{MB} – number of pixels in a single MB

i – seven various subblock sizes

r_{xy} – nine half pixel candidate positions

R_{frame} – number of reference frames; range [1, 16]

fps – frames per second

MB_{frame} – total number of MBs in a single video frame

$width_{frame}$ – width of a video frame

$height_{frame}$ – height of a video frame

The full-search half-pixel VBS-FME algorithm has the following fixed values: P_{MB} = 256, i = 7, r_{xy} = 9. The remaining variables can be set within the encoder to suit the video stream to be processed as well as the desired compression ratio. A sample setting where R_{frame} = 4, ($width_{frame}$ x $height_{frame}$) = 1920x1080 (HDTV) and fps = 30 yields a total of approximately $\boldsymbol{15.7 \times 10^9}$ pixel comparisons per second.

Having expressed the intense computational complexity trend, it follows to set out to create algorithms with higher processing bandwidth, yet with efficient design implementation. Next we describe our approach at scaling the FME algorithms.

The size of macro/sub-blocks can range from 4x4 to 16x16 pixels, FME hardware must carefully balance parallelism with hardware utilization. In particular, previous work [1][3][10]-[16] has implemented designs that can simultaneously process 4, 8 or 16 pixels from a single row of a macro/subblock at a time. In this work we found that expanding parallelism only along the row, however, can quickly degrade utilization as the number of simultaneously processed pixels is increased. Since under-utilized hardware reduces overall efficiency, it is important to create designs that can increase parallelism while maintaining a high degree of hardware utilization.

We observe that the parallelism of FME not only can be increased horizontally across a row of pixels but also vertically along a column of pixels. Designs that increase parallelism in the vertical direction can potentially provide higher performances while maintaining high hardware utilization. In this work, we develop a scalable FME algorithm and compare the effect of horizontal with vertical scaling.

Table 1 - The FME Scaled Designs

Type of Scaling	Scaled FME Designs	Source of Design
Vertical	M=10, N=1	Our work
	M=4, N=1	Our work
	M=2, N=1	Our work
Base Design	M=1, N=1	[13]
Horizontal	M=1, N=2	[15]
	M=1, N=4	[16]

Table 1 lists six scaled FME designs addressed in this work. The base and horizontal design were previously implemented by [15] and [16] respectively. The three vertically scaled designs have been developed in this research. The new designs are primarily based on the base design, which was enhanced to form a scalable version. The vertically scaled designs were enhanced to perform vertical data processing.

2.9 Chapter Summary

In this chapter we presented the basics of video and video compression algorithms. In particular we covered the mathematical calculation necessary for the FME algorithm which is referenced in the later chapters where we introduce our hardware architecture design. The basics of FME calculations are strongly correlated with design decisions we faced when scaling the encoder. In addition we described a few techniques used during the hardware implementation as well as the non-scaled FME architecture. Finally, we conclude the chapter with the motivation behind our work which is rooted in the complexities of video encoding.

Chapter 3 - The Scalable FME Architecture

This chapter describes the scalable FME architecture core. The description is presented in a top down approach, starting with the overall data flow of the FME algorithm. We introduce the architectural components of the FME core followed by how each of them is scaled in the design.

In this work, we employ a scalable full-search FME engine on an FPGA in order to investigate the tradeoffs between used hardware resources and measured performance. We create several versions of the FME architecture by introducing two scalable factors, ***M*** and ***N***. By varying these scalable factors we in turn vary the number of pixels simultaneously processed by the engine per clock cycle, which translates to the amount of instantiated processing hardware.

M and N correspond to the height and width of the input pixel array taken in by the FME engine every clock cycle. Figure 9 shows the six various sizes of the input pixel array corresponding to the six scaled FME designs. The array sizes correspond to M x (4N+6) pixels. These variables determine the total number of pixels that are simultaneously processed by the system. This in turn determines the amount of resources instantiated for the hardware engine, trading area and hardware utilization for parallelism and performance. By simultaneously processing larger pixel arrays, we increase parallelization and reduce the total processing time of the 41 motion vectors. For example, the horizontally scaled design in Figure 9 - (c) or the vertically scaled design in (f) will process an 8x8 block in less clock cycles compared to the base design in Figure 9 - (a).

The base design was previously implemented by [13]. The two horizontally scaled

designs, [M=1, N=2] and [M=1, N=4] were implemented by [15] and [16] respectively. Instead of arbitrary increasing the width of the FME engine, the dimensions were chosen so that they correspond to the width of various subblocks. The width of the FME input array is incremented by 4 pixels at a time. This results in a corresponding block width of 4N pixels, which can be effectively mapped to N 4-pixel wide Hadamard transforms shown in Equation 4. An SATD value is defined over a 4-pixel wide subblock, which is the smallest common subblock size. For this reason, processing less than 4-pixels wide chunks in hardware would be inefficient. In our scalable design, when taking into account the surrounding pixels required for the 6-tap filter, the total width of the input array is expressed as 4N+6. Overall, the dimensions of each scalable input pixel array can be expressed ***M x (4N+6)***.

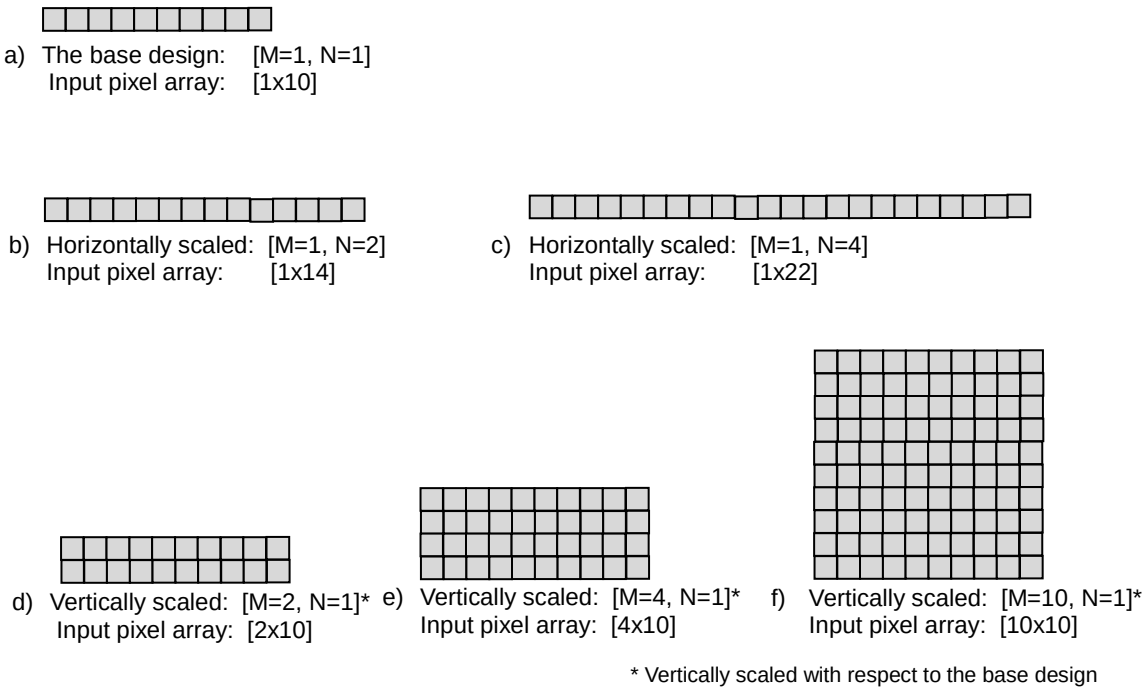


Figure 9 - The Input Array Sizes for 6 Scaled FME Designs

The three vertically scaled designs (d), (e) and (f) in Figure 9 were developed in this work. The vertically scaled designs increase parallelism by processing multiple rows at the same time. The design in (d) is capable of handling 20 pixels per clock cycle, whereas the design in (e) can handle 40. The design in (f) takes in 100 pixels per clock cycle, corresponding to the size 4x4 subblock including all of its surrounding pixels.

The naming convention for our scaling is derived with respect to the base design. When compared this way, we observe that the designs in (d), (e) and (f) are just as wide as the base design (a), therefore we consider them to be vertically scaled. Similarly, the designs in (b) and (c) are the same height as the base design but wider, therefore we consider them to be horizontally scaled.

The overall structure of the scalable FME architecture is shown in Figure 10. The FME contains two main components: *the Interpolation Engine* (IE) and *SATD Processing Units* (PUs). The IE is further divided into the horizontal (H-IPU) and vertical interpolation units (V-IPU), which generate half pixels. After interpolation, the PUs calculate SATD values for the nine candidate vector half pixel refinement positions.

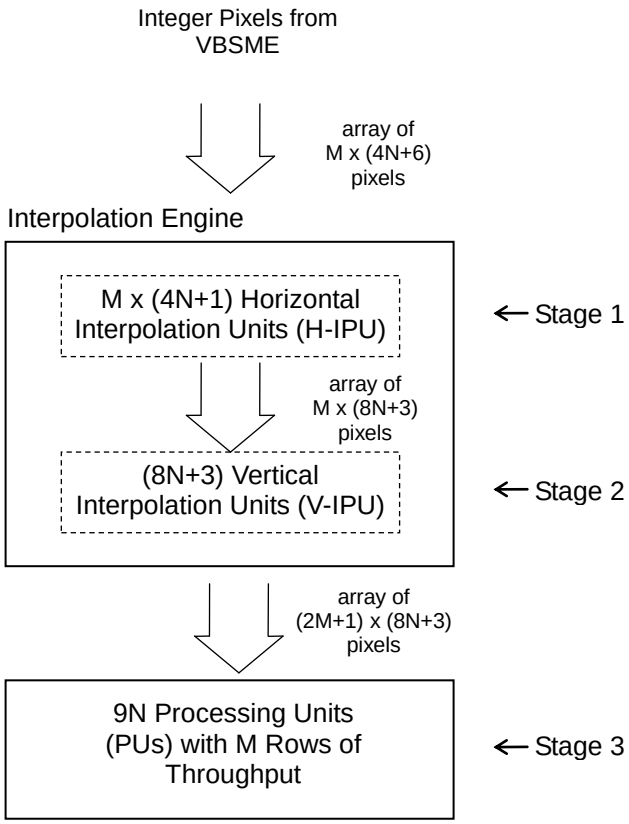


Figure 10 - The Scalable FME Architecture

The base design was extended in order to integrate scalability across two different axes. In particular, we modified the IE in order to vary it in width and height corresponding to the scalable variables. To achieve this we extended the connectivity of its components, such as the H-IPUs and V-IPUs. Further, we enhanced the V-IPUs in order to enable them to be used in vertically scaled designs and process multiple rows of pixels per clock cycle. Similarly, in order to match the vertically increased data throughput, we extended the PU design by adding resources and altering the operation of its internal structure.

Finally, the described design was written in VHDL with an *auto-tuning* feature, which enables the user to set the M and N scalable factors at compile time and then instantiates the resources for the corresponding FME engine, based on the designs shown in Figure 10.

3.1 Interpolation Engine

The IE takes integer pixels as input and generates interpolated half pixels as output. The basic building block of the IE is the six-tap Finite Impulse Response (FIR) filter. Multiple FIR filters are aggregated to compose Vertical (V-IPU) and Horizontal Interpolation Units (H-IPU) described next.

3.1.1 Horizontal Interpolation Unit

Structurally, our H-IPU design is based on the non-scaled design in [13]. Here each H-IPU contains a single FIR filter whose connectivity is shown in Figure 11. As previously described, each FIR filter takes in a row of six adjacent integer pixels as its input and produces one half pixel as its output. The generated half pixel is positioned between the third and fourth input pixel. The output pixel array from Stage 1 is passed onto Stage 2 for further half pixel generation. The entire pixel array output of Stage 1 consists of an array of M x (8N+3) pixels as shown in Figure 10. In order to align the generated half and original integer pixels in the Stage 1 output array, buffers are placed alongside each H-IPU. Each buffer receives integer pixels and delays them by five clock cycles, equivalent to the number of processing cycles required by the

FIR filters. This way the integer pixels and the half pixels experience the same logic delay and the row-alignment is preserved for Stage 2 processing.

In our design the number of H-IPUs in Stage 1 scales with M and N. Instead of being positioned in a single row, as in the base design, the vertically scalable H-IPU units are arranged in a 2D grid of size M x (4N+1). This allows the 2D grid to process M rows of pixels per clock cycle. In the input integer pixel array, each of the M rows of pixels is mapped onto its own row of H-IPUs for parallel processing. Further, within each row, the 4N+6 integer pixels are mapped onto 4N+1 H-IPUs.

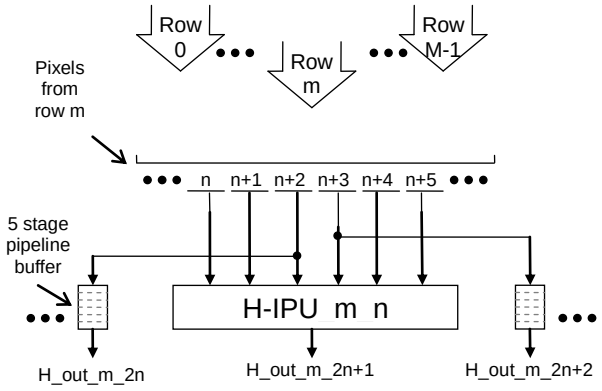
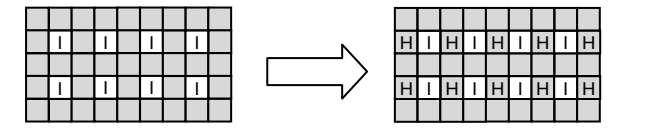


Figure 11 - The H-IPU Connectivity at Row m and Column n

The scalable 2D grid of H-IPUs makes up Stage 1 of the FME architecture. The H-IPUs are used to generate multiple variable length rows of half pixels as is shown in Figure 12. The remaining empty rows are generated by V-IPUs in Stage 2.



a) Input to Stage 1 of FME architecture

b) Output of Stage 1 of FME architecture

Figure 12 – Pixel Interpolation in Stage 1

3.1.2 Vertical Interpolation Unit

Our scalable V-IPU unit is based on the non-scaled V-IPU from the base design implemented in [13], which contains a single pixel wide FIFO and six FIR filters. Here the V-IPU receives a single pixel into an 8-bit wide FIFO, while the six FIRs calculate a single half pixel per clock cycle. The output pixel column is composed of three pixels, an integer pixel surrounded by two half pixels. One of the half pixels is the newly generated half pixel by the FIR filter, and the other is a buffered half-pixel from the previous clock cycle. This way every output column begins and ends with a half pixel which is required for mapping onto the Stage 3 PUs.

Our scalable V-IPU has enhancements which allow it to process multiple rows of pixels per clock cycle and is shown in Figure 13. First, it contains a variable width and depth FIFO, which can take in multiple pixels per clock cycle. Second, there are M FIR filters which simultaneously generate M half pixels. Finally we connect 8N+3 V-IPUs in a row, which make up Stage 2 of the FME architecture. Here the variable N allows for horizontal scaling while M varies the resources inside the V-IPU allowing for vertical scaling.

The scalable FIFO design is shown in Figure 13 - (b). Its size is determined by the number of input pixels to the V-IPU per clock cycle. In particular, the M filters need to access a total of M+5 pixels while a V-IPU receives only M input pixels per clock cycle. Consequently, the FIFO needs to buffer five pixels from the previous cycles – resulting in a total FIFO size of *ceil*((M+5)/M) x M pixels, shown in Figure 13 - (b). The width of the FIFO**,** *w*, is set to M, the number of input pixels per clock cycle. The depth, *d*, is set as the minimal height needed in order for the FIFO to hold *ceil*((M+5)/M pixels at width *w*.

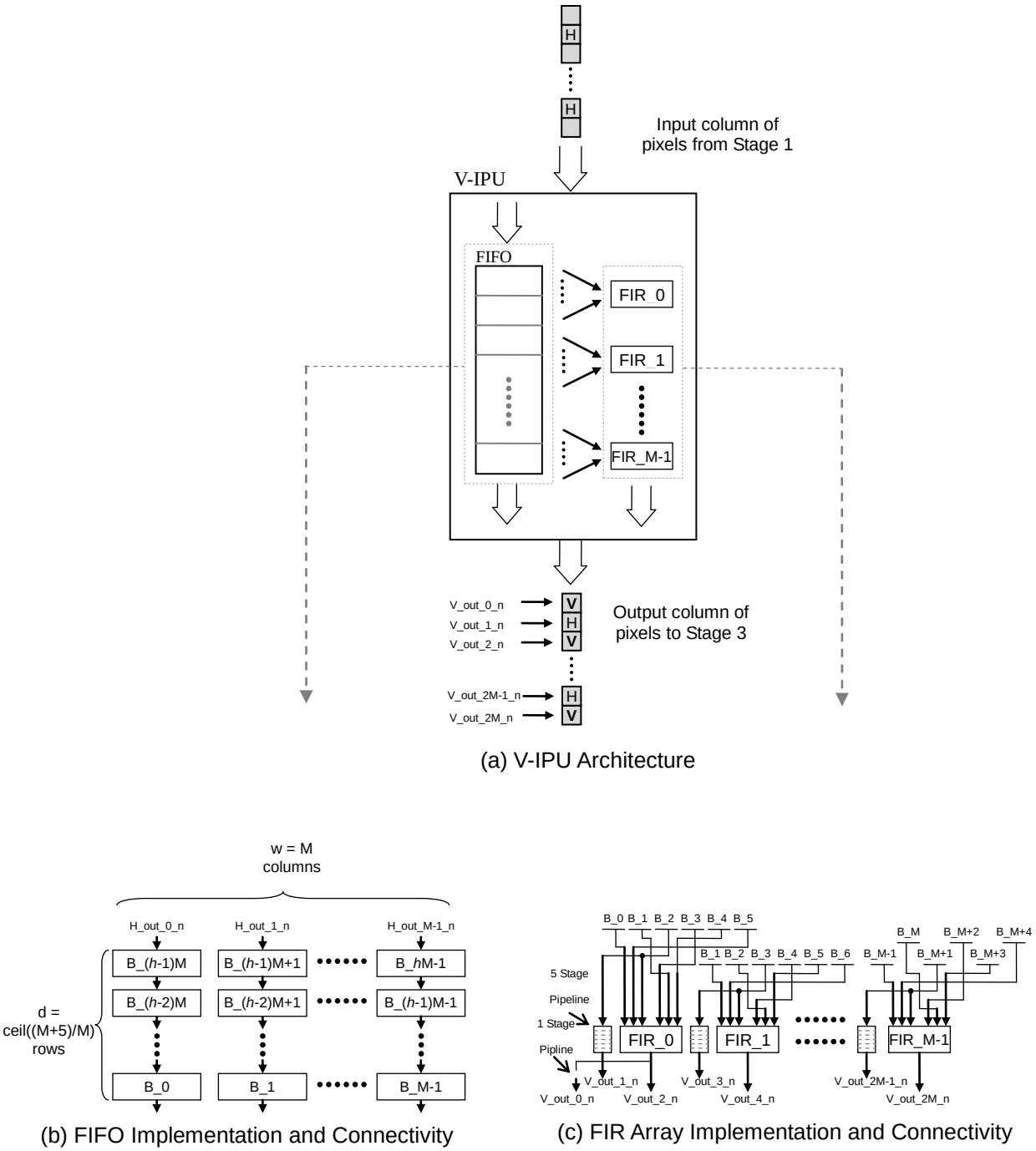
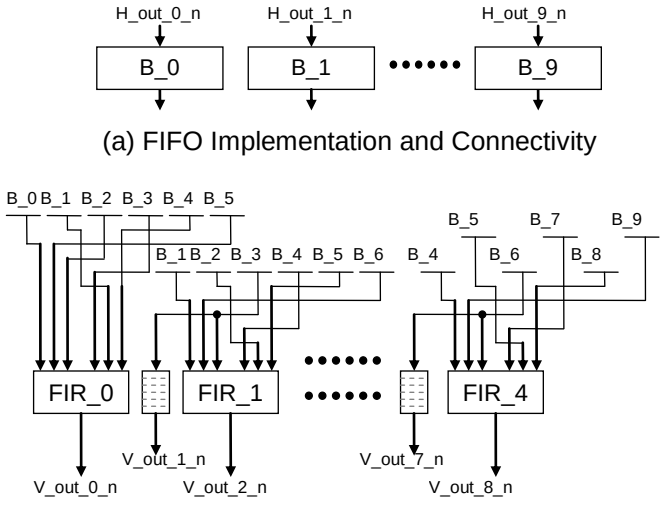


Figure 13 - The *n*th Scalable V-IPU

Each clock cycle, M new pixels are received by the FIFO, the rest of the pixels shift down, and the bottom pixels are sent out and become part of the V-IPU output column. The FIRs grab the necessary data from the FIFO and produce one half pixel each. The mapping of the pixels between the FIFO and the FIRs is shown in Figure 13 - (c). The first filter, FIR_0, takes in the top six pixels from the FIFO, and each subsequent FIR has an input pixel array which is shifted down by one pixel.

The V-IPU in the [M=10, N=1] FME design is an exception to the previously described scalable approach for the following reason. In this case the design takes in a 4x4 block of elementary pixels centered inside a 10x10 subblock of surrounding pixels. The V-IPU generates half pixels only around the elementary pixels, hence creating a total of only five half pixels. Therefore, each V-IPU can be reduced by using only five FIR filters, thus saving hardware resources while increasing data throughput. Similarly, for each clock cycle, within the FIFO the entire 10 pixels are flushed and replaced by 10 new ones; therefore the FIFO is reduced to a simple register, 10 pixels in size.



(a) FIFO Implementation and Connectivity

(b) FIR Array Implementation and Connectivity

Figure 14 - V-IPU for the [M=10, N=1] Design

The V-IPUs are used to generate the remaining half pixels which are located in between the rows of integer pixels, shown in Figure 15. The newly generated half pixels and the FIFO output pixels make up an output column produced by a single V-IPU. The aggregated columns of all the V-IPUs make up the output from Stage 2 which is the M x (8N +3) pixel array. Finally, the output pixel array from Stage 2 is forwarded to Stage 3, to the 9N Processing Units (PUs) for SATD calculations.

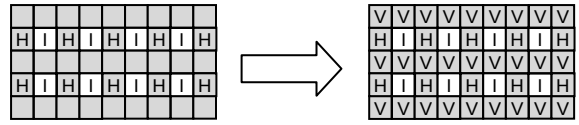


Figure 15 – Pixel Interpolation in Stage 2

3.2 SATD Processing Unit

Each PU has two input streams: the 4x4 subblocks from the *current* frame and the 4x4 subblocks from the *reference* frame. The current frame is the one to be encoded and the reference frames are the potential matches for the subblock from the current frame. The PUs function is to generate SATD values. The SATD represents how good of a match the reference 4x4 pixel subblock is to the current 4x4 subblock.

Our PU design is based on the non-scalable design previously implemented in [13], where a single set of 9 PUs processes one row of four pixels per clock cycle. Our scalable version also encompasses the horizontally scaled PU design implemented in [15] and [16]. Here,

2 and 4 sets of PUs were placed adjacently in order to process 8 or 16 elementary pixels at once. We denoted the horizontal scaling using the N scalable variable which instantiates N adjacent sets of 9 PUs. Next we enhanced the PU with additional resources which allow it to process multiple rows of pixels per clock cycle and as a result to be vertically scalable.

The resulting scalable PU architecture is shown in Figure 16. The PU is composed of three processing blocks: Residue Generators, a 2-D Hadamard Transform and an Absolute Function with Summation Units. The architecture of each of the parts is described in the next sections.

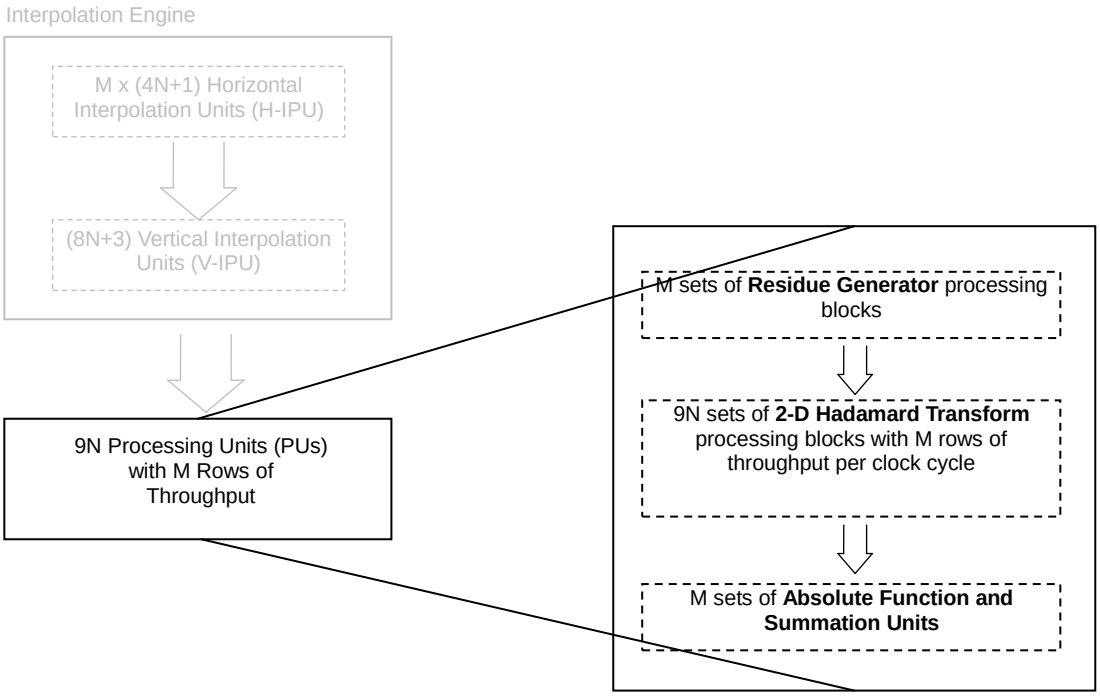
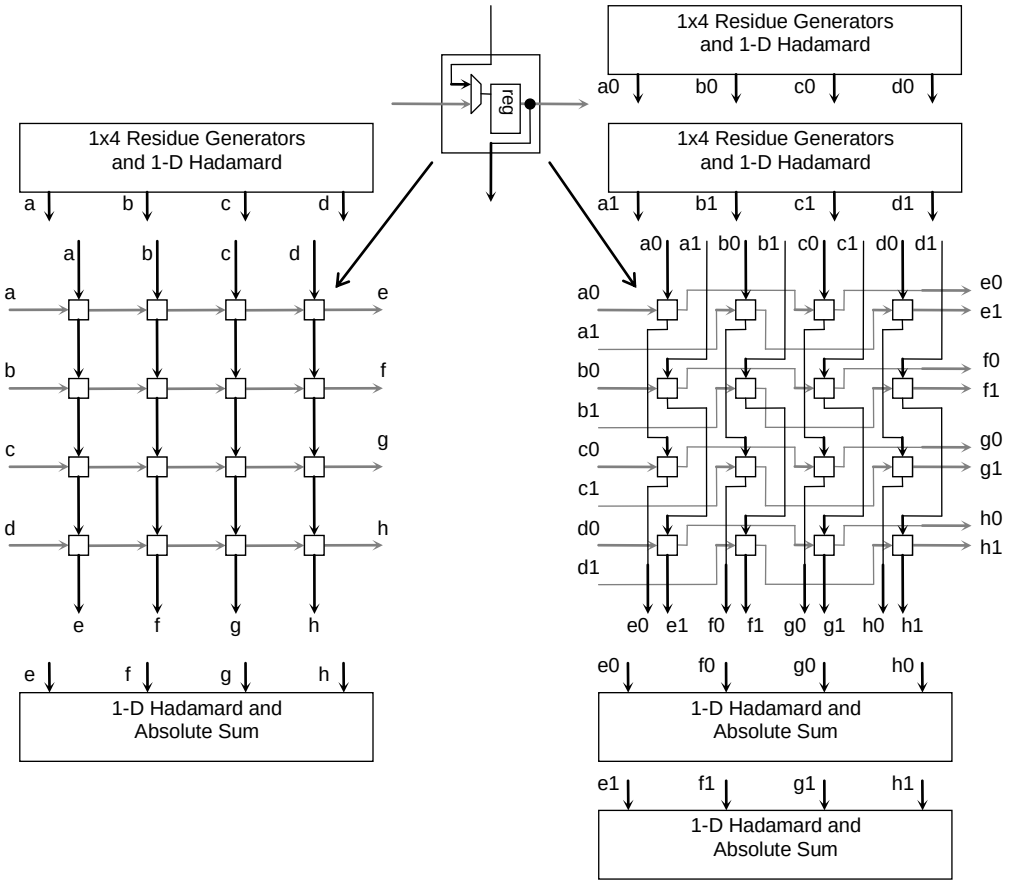


Figure 16 - The Processing Unit Architecture

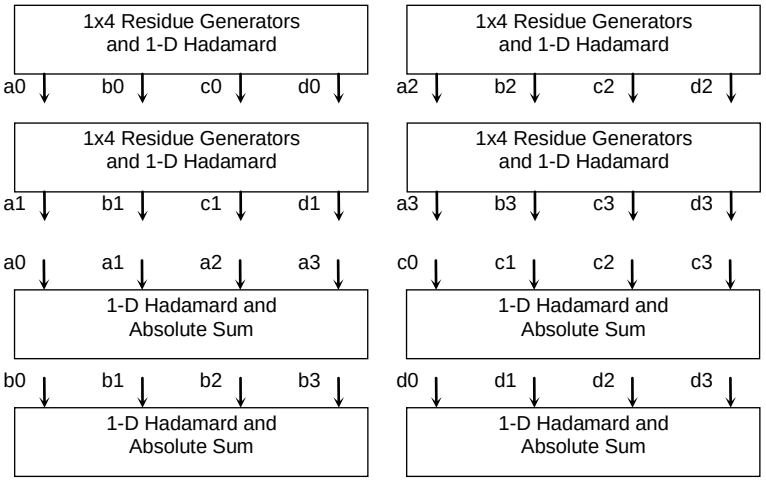
3.2.1 Vertical Scaling of the PU

Vertical scaling requires an increase in the PUs throughput. To allow for this the PUs are enriched with additional hardware in order to process the additional rows. In particular, for every additional row, each PU requires an extra set of residue generators, two extra 1-D Hadamard transform units, and one absolute value and summation unit. For example, a PU design with a throughput of two rows per clock cycle is shown in Figure 17 - (b). This PU corresponds to the [M=2, N=1] design and it calculates one SATD every two clock cycles. The PU design with a throughput of four rows per clock cycle is shown in Figure 17 - (c), corresponding to [M=4, N=1] and [M=10, N=1] designs, capable of producing an SATD value every clock cycle.



(a) Throughput – 1 Row per Cycle

(b) Throughput – 2 Rows per Cycle



(c) Throughput – 4 Rows per Cycle

Figure 17 - The Vertically Scaled PU Designs with Various Data Throughput Rates

3.2.2 Residue Generators

When we scale vertically we instantiate M sets of Residue Generators in parallel. Each set of residue generators processes a rows four pixels. Two sets of four pixels are inserted into the residue generator every clock cycle, as shown in Figure 18. One row of four pixels comes from the 4x4 subblock from the current frame, and the other from the 4x4 subblock from the reference frame. The residue generator is composed of four units, each calculating the difference between the corresponding current and reference frame pixels from the input rows. The output of the residue generators is passed onto the 2-D Hadamard transform unit.

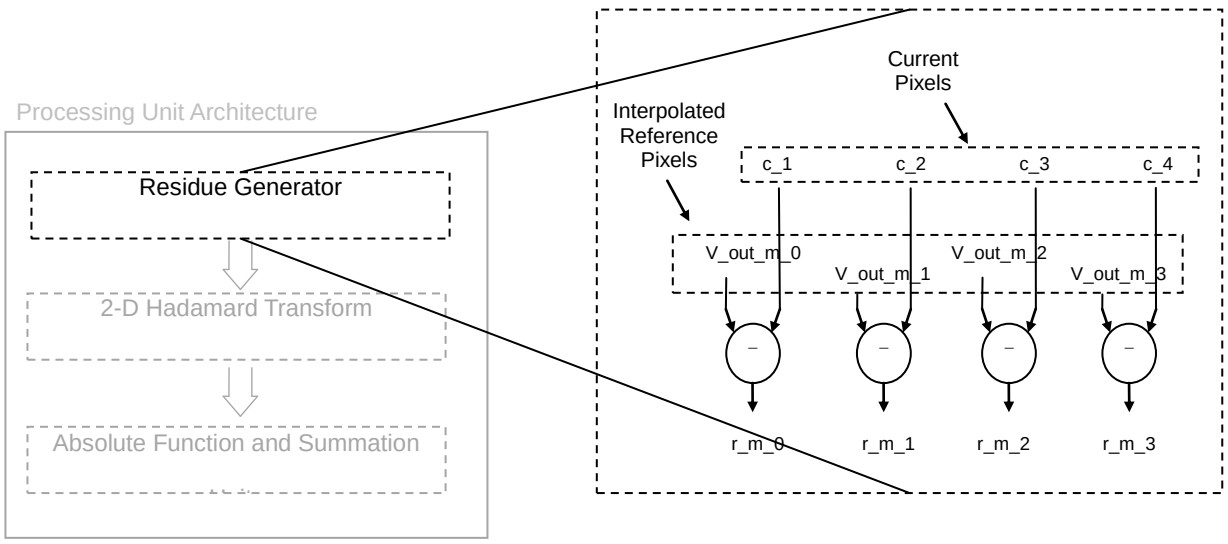


Figure 18 - The Residue Generator

3.2.3 The 2-D Hadamard Transform

The 2-D Hadamard transform is implemented in three parts: a single transpose shift register placed between two sets of 1-D Hadamard transform units, as shown in Figure 19. Each row of four residue pixels is sequentially processed by the three parts of the 2-D Hadamard Transform, and the output is passed onto the Absolute Function and Summation Units.

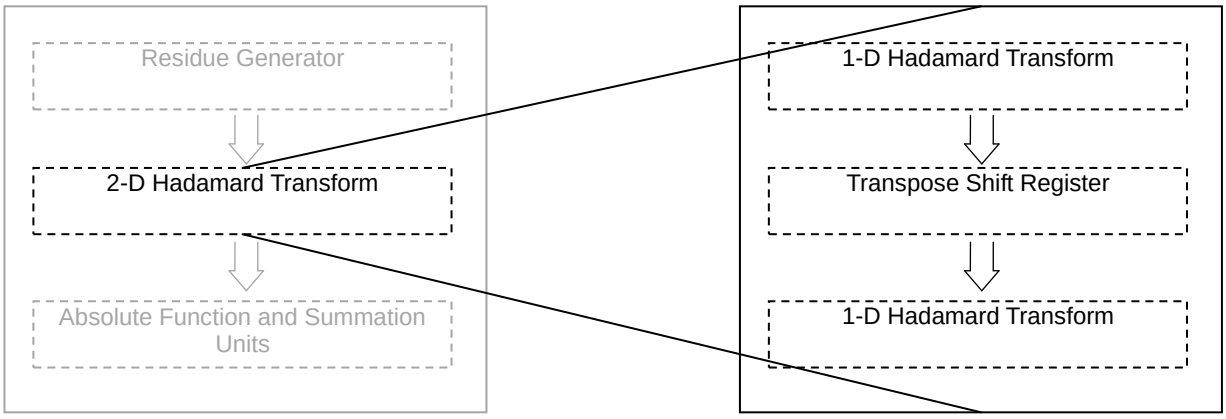


Figure 19 - 2-D Hadamard Transform

The Transpose Shift Register

The transpose shift register takes in a 4x4 pixel subblock and outputs its transpose which is then passed onto the second 1-D Hadamard Transform. Two sample input 4x4 subblocks, A, are shown in Figure 20 - (b), and the outputted transpose, A^T, is shown in Figure 20 - (c).

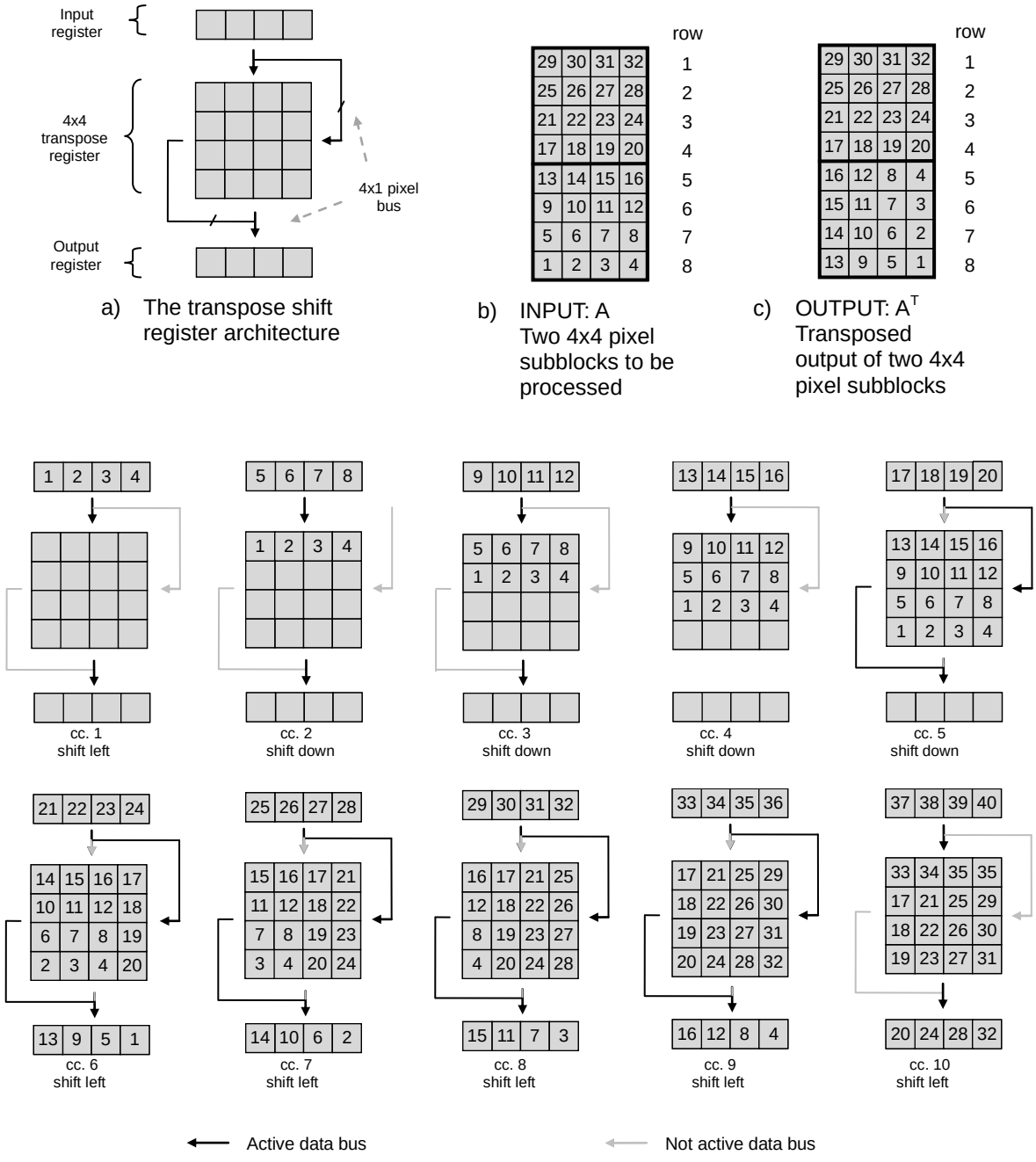


Figure 20 - Operation of the Transpose Shift Register for the [M=1, N=1] PU

Figure 20 - (c) shows a cycle by cycle sample data flow operation of the transpose shift register. The operation depicted is for a PU from the [M=1, N=1] design, which produces one SATD value every four clock cycles, and shifts a single row/column at a time.

The transpose shift register toggles between a *shift-down* and a *shift-left* mode every time an entire 4x4 subblock has been written into the register. In the shift-down mode, pixels from the 4x4 subblock are written into the register horizontally, as is depicted in Figure 20 - (d), in cc.1 to cc.5. In general, the register shifts in M rows per clock cycle, while the 4x4 block originally stored in the transpose shift unit is read out horizontally M rows per clock cycle. This continues until the entire 4x4 block originally stored in the transpose register unit is completely replaced by the new 4x4 block. At that time the transpose shift register switches to the shift-left mode. Now the next 4x4 block is written into the unit vertically 1 column per clock cycle, as is depicted in Figure 20 – (d), in cc. 6 to 10. In general, the register takes in M columns per clock cycle, while at the same time the stored 4x4 block is read out vertically M columns at a time.

The frequency of the switch between shift-down and shift-right modes is determined upon the throughput, M. The transpose register switches every 4 and 2 clock cycles for the [M=1, N=1] and [M=2, N=1] designs correspondingly. The [M=4, N=1] and [M=10, N=1] design process an entire 4x4 pixel array in a single clock cycle, therefore not requiring a transpose shift register and shift-down/left modes. Instead the 16 pixels are directly mapped between the 1-D Hadamard Transforms and the absolute function and summation units.

As we scale the FME architecture to increase data throughput and so the timing of the switch between the shift modes must be preserved with respect to the transition between adjacent 4x4 subblocks. The effects of this on the scaled designs of FME are discussed in terms of vertical data alignment in Chapter 4.3.3.

3.2.4 Absolute Function and Summation

We instantiate M sets of absolute function and summation units in parallel. After the Hadamard transform, the four output pixels pass though the absolute function units, followed by summation logic. The resulting output value is representative of a single row of four pixels. Four of these output values are accumulated and summed to produce the final SATD for a single 4x4 subblock. Similarly, to calculate the SATD value of a larger subblock, the SATD values of its composing 4x4 subblock are accumulated and summed.

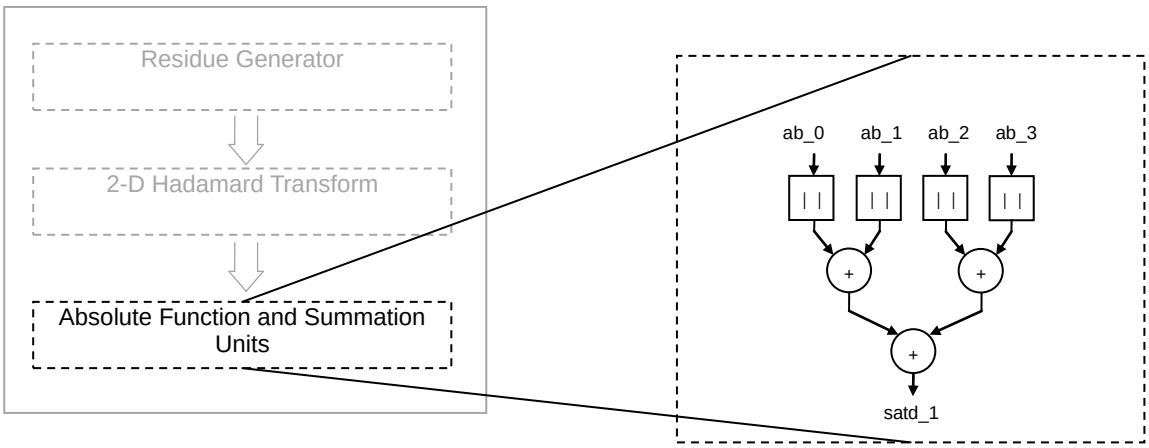


Figure 21 - The Absolute Function and Summation Units

3.2.5 Horizontal Scaling of the PU

The number of PUs required to simultaneously calculate all of the SATDs for an array of pixels is determined by the width of the Stage 3 input pixel array which is of (2M+1) x (8N+3) size. Rows of (2M+1) x (8N+1) pixels are forwarded to PUs in Stage 3 while the remaining left most and right most two columns can be stored for later use in the quarter pixel calculations, as shown in Figure 22. At half pixel resolution, each row of this array contains (8N+3) integer pixels, out of which 4N are elementary integer subblock pixels belonging to N 4x4 subblocks. Each SATD is defined over an array of 4x4 pixels was shown in Equation 4. To maximize parallelism a set of 9 PUs is used to handle the calculation of nine candidate refinement positions (shown in Figure 23) and their SATD values for one 4x4 block. When scaling horizontally, we simply instantiate N sets of 9 PUs, which are capable of processing multiple 4x4 subblocks side by side. The pixels for each 4x4 subblock are mapped onto a single set of nine PUs, and the nine candidate positions are calculated by the set.

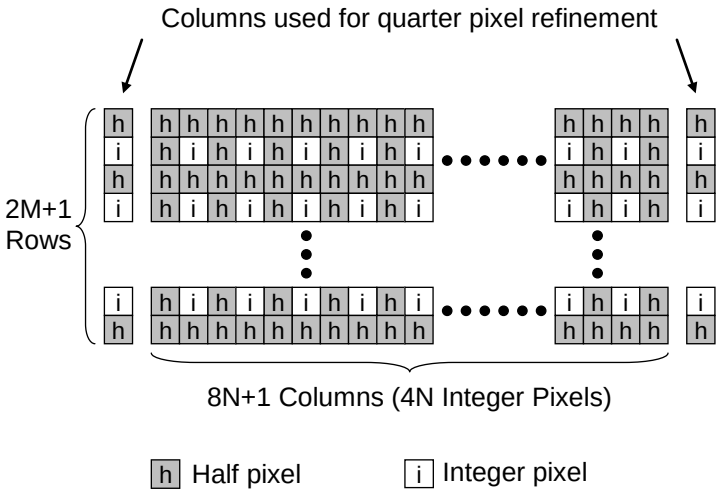


Figure 22 - Pixel Layout at the Output of Stage 2 (V-IPUs)

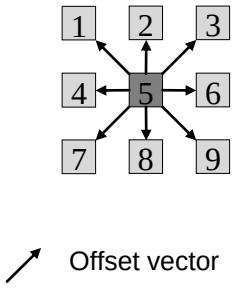


Figure 23 - Nine Half Pixel Candidate Positions and Their Offset Vectors for One Integer Pixel

In Figure 24, we show how a single 4x4 subblock with its interpolated half pixels is mapped onto a single set of nine PUs. Every one of the nine PUs receives an array of 4x4 pixels, shown in black. The middle PU, #5, receives the original integer pixels. The surrounding eight PUs receive the shifted and interpolated version of the 4x4 original subblock set. The offset vectors for the 4x4 block and the nine PUs correspond to the offset vectors for the single integer pixel and its eight surrounding half pixel candidate positions, shown in Figure 23. Depending on the vertical design in question, M rows of four pixels are forwarded to the array per clock cycle.

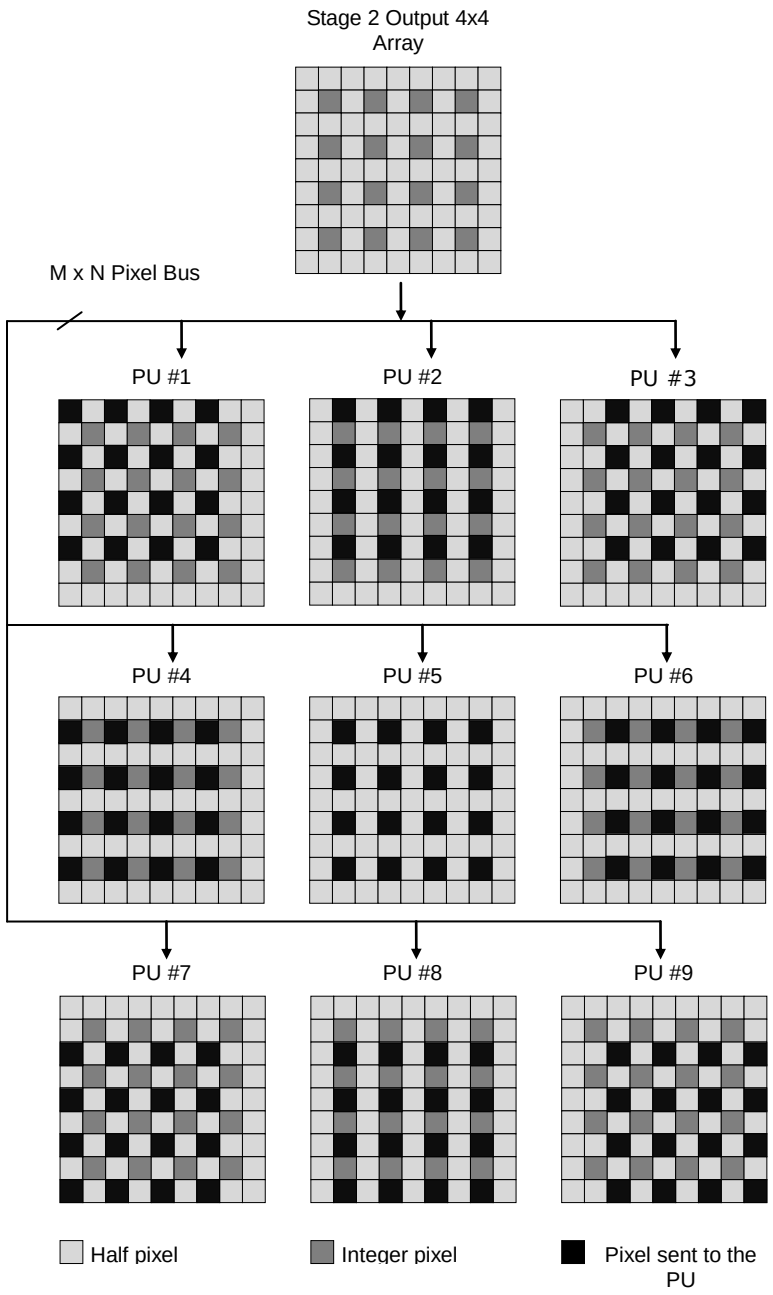


Figure 24 - Mapping a 4x4 Subblock to a Single Set of 9 PUs

3.3 Chapter Summary

In this chapter we have presented our scalable FME architecture and its two main components: the IE and the PU. We introduced the basic building block of the IE which is the FIR filter, and how it is implemented in order to create H-IPUs and V-IPUs. We showed how the PU is implemented by decomposition into three main parts: Residue Generators, 2-D Hadamard Transform, Absolute Function and Summation Units. We further explored the data flow between the IE and PU and the way the pixels map onto sets of 9 PU. We build upon the architecture details presented in this chapter in order to develop further concepts, such as data alignment with respect to vertical scaling. In particular we show how vertical alignment is influenced by the number of clock cycles it takes for the PU's 2-D Hadamard Transform to process a set of 4x4 pixels.

Chapter 4 - Design Space Exploration

The analysis techniques described in this chapter, along with the comparisons of the FME engines and their pixel processing schedules, set the stage for the next chapter which analyzes the overall scalability of each design. We describe factors used to compare the tradeoffs between the six scaled FME designs. First, we introduce the concept of *block decomposition*, which leads into a discussion of *data redundancy*. Next, we define three factors which influence *hardware utilization*: input data variance, throughput variance and vertical alignment, and derive equations for estimating the overall hardware resource usage. Finally, we calculate and the total number of clock cycles it takes each engine to process a set of 41 subblocks.

4.1 Block Decomposition

Block decomposition is a method of breaking up the macro/sub-blocks into smaller block sizes. If the processing hardware is narrower than the subblock width, the subblock must be decomposed into smaller chunks and the chunks processed sequentially. The next two sections describe block decomposition approaches for the vertically and horizontally scaled designs.

4.1.1 Block Decomposition for Vertically Scaled Designs

The block decomposition approach depends on the width of the FME unit. Consequently, the base design and two of the vertically scaled designs, [M=2, N=1] and [M=4, N=1], have the same decomposition schedule. The third vertically scaled design, [M=10, N=1], is an exception due to inherent additional design requirements, discussed later in this section.

The [M=1, N=1] design is wide enough to process rows of subblocks that are four pixels wide, plus the six surrounding pixels, three on each side. The macro/sub-blocks that are wider than four elementary pixels must be decomposed into a series of 4-pixel wide blocks. For example, Figure 25 shows a decomposition approach for a 16x16 MB. The block is shown with its surrounding pixels, required by the 6-tap FIR filters, resulting in a total of 22x22 a pixel array. When the MB is decomposed, each vertical strip, or sweep, contains some elementary block pixels encircled by surrounding pixels required by the FIR filter. The FIR filter requires six adjacent integer pixels therefore generating half pixels around the edges of the decomposed block requires padding of three extra pixels.

The total pixels in all four sweeps amount to a greater number then in the original 22x22 array, which results in redundant processing described in Section 4.2.

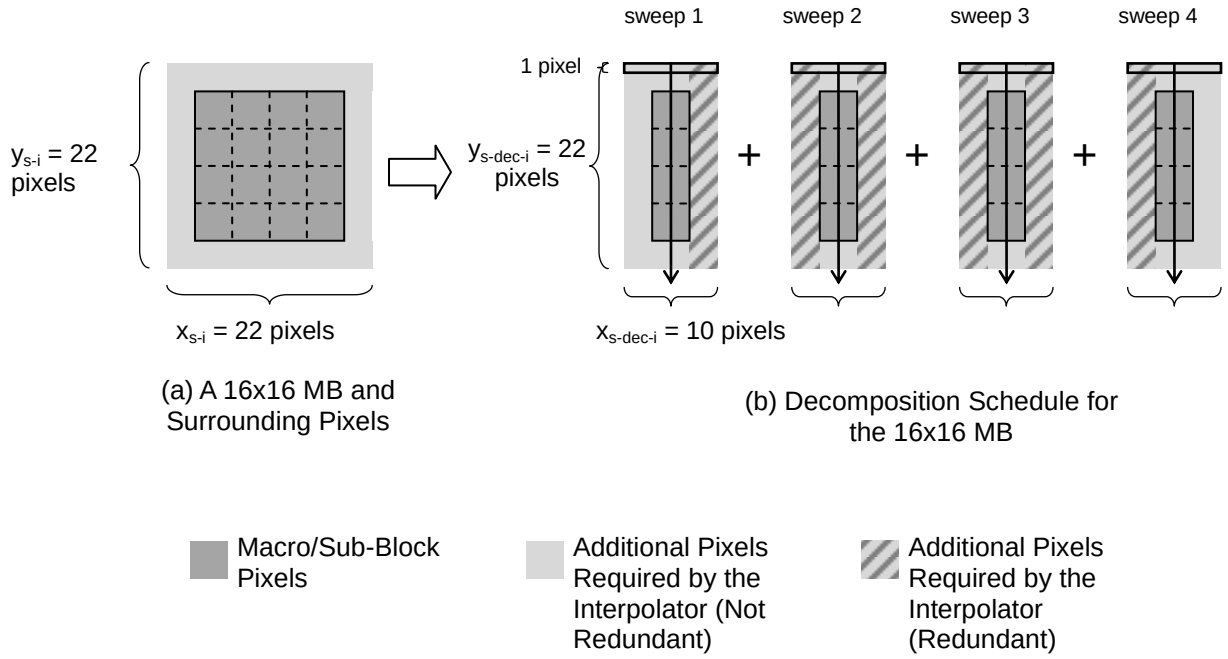


Figure 25 - Decomposition Schedule for a 16 x 16 Block for the [M=1, N=1] Design

In Table 2 we show the decomposition approach for all of the other subblock types, for the same [M=1, N=1] design. The left portion of the table, columns 2 and 3, contains information about the subblocks in their entirety, where x_{s-i} and y_{s-i} are subblocks dimensions including their surrounding pixels. The right side of the table, columns 4, 5 and 6, show details of the subblock's decomposition, where $x_{s-dec-i}$ and $y_{s-dec-i}$ represent the block's height and width respectively, also including the surrounding pixels. The s_i term is the number of decomposed smaller blocks, or the number of vertical sweeps performed by the IE.

Note that the $y_{s-dec-i}$ value is the same as the height of the block, which determines the length of each sweep. The $x_{s-dec-i}$ value is the same across all block types because the [M=1, N=1] FME engine is fit for processing rows that are 10 pixels wide. Consequently, block decomposition enables processing of wider subblocks with sequential multiple sweeps. For example, the 16x16 MB requires four sweeps, as shown in Figure 25. On the other hand, the

smallest 4x4 block requires only a single sweep.

Table 2 - The Block Decomposition Schedule for the [M=1, N=1], [M=2, N=1], [M=4, N=1] Designs

Subblock Type	Before Block Decomposition		After Block Decomposition		Number of Vertical Sweeps
	Subblock Dimensions		Subblock Dimensions		
	Width	Height	New Width	New Height	
i	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i
1	22	22	10	22	4
2	14	22	10	22	2
3	22	14	10	14	4
4	14	14	10	14	2
5	10	14	10	14	1
6	14	10	10	10	2
7	10	10	10	10	1

The [M=10, N=1] design has a different decomposition schedule than the other vertically scaled designs. The schedule breaks all macro/sub-blocks into 4x4 elementary pixel arrays, or 10x10 pixel arrays including the surrounding pixels. The interpolators process the 10x10 pixel arrays, one per clock cycle, where the subblock's elementary pixels are centered in the middle of the 10x10 array as shown in Figure 26. In this case, the data fed into the interpolator needs to match the exact throughput of the PU, due to a vertical alignment requirement explained in Section 4.3.3. This decomposition results in a significant increase in data redundancy, discussed in Section 4.2.

In Table 3, note that the $x_{s\text{-}dec\text{-}i}$ and $y_{s\text{-}dec\text{-}i}$ are now always 10, and only the s_i value varies between block types.

Table 3 - The Block Decomposition Schedule for the [M=10, N=1] Design

Subblock Type	Before Block Decomposition		After Block Decomposition		Number of Vertical Sweeps
	Subblock Dimensions		Subblock Dimensions		
	Width	Height	New Width	New Height	
i	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i
1	22	22	10	10	16
2	14	22	10	10	8
3	22	14	10	10	8
4	14	14	10	10	4
5	10	14	10	10	2
6	14	10	10	10	2
7	10	10	10	10	1

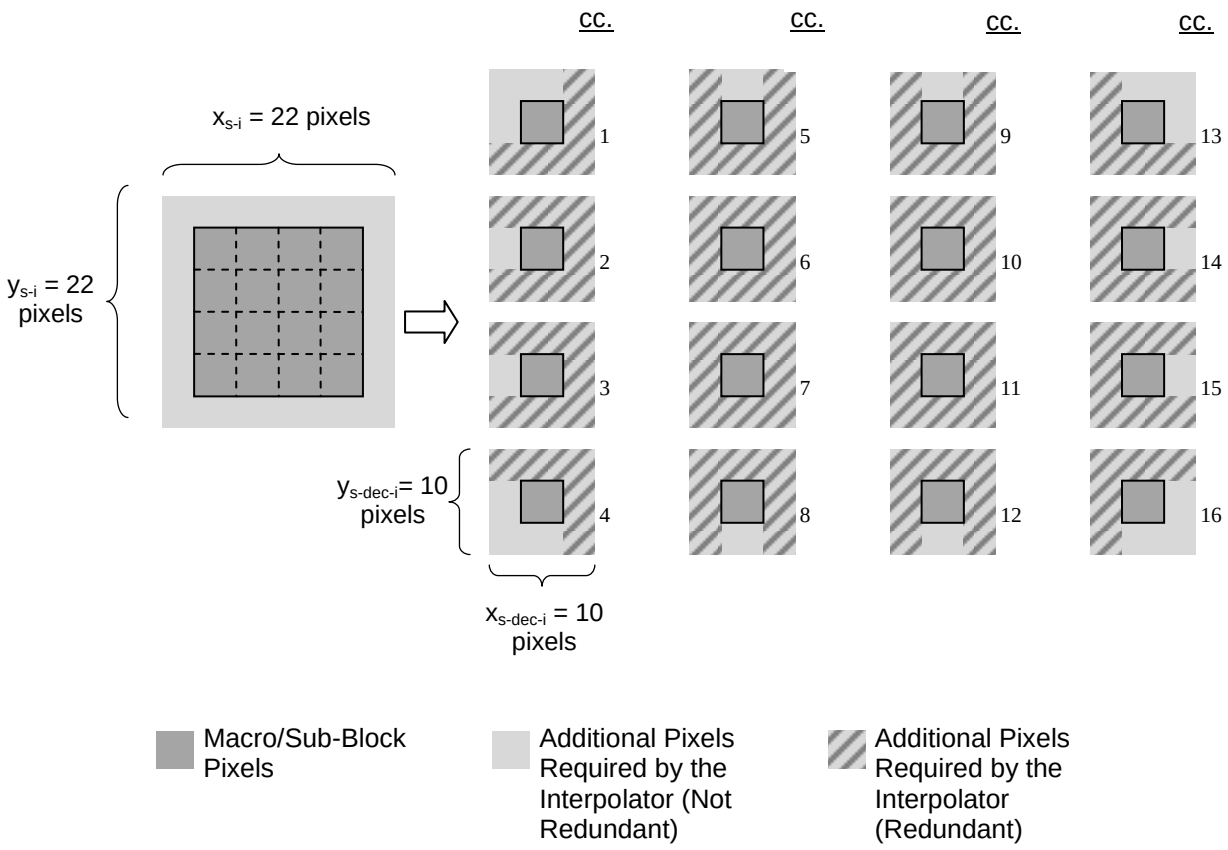


Figure 26 - 16x16 Block Decomposition and Data Redundancy for the [M=10, N=1] Design

4.1.2 Block Decomposition for Horizontally Scaled Designs

The two horizontally scaled FME designs have a different block decomposition schedule due to the increased width of the IE.

The widest FME design, [M= 1, N=4] can handle an input row size of 22 pixels, which is fit for the size of an MB (16x16 + surrounding pixels). Unlike in vertical scaling, in this design, the MB can be processed without requiring any block decomposition, as is shown in Figure 27 – (a). The remaining subblock types are also processed without requiring block decomposition. An additional feature in this design is that the 4-pixel wide subblocks are narrow enough to be placed side by side, two at a time, as is shown in Figure 27 – (c). This way the processing time for 4-pixel wide subblocks is halved, while the hardware utilization increased.

Table 4 - The Block Decomposition Schedule for the [M=1, N=4] Design

Subblock Type	Before Block Decomposition		After Block Decomposition		Number of Vertical Sweeps
	Subblock Dimensions		Subblock Dimensions		
	Width	Height	New Width	New Height	
i	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i
1	22	22	22	22	1
2	14	22	14	22	1
3	22	14	22	14	1
4	14	14	14	14	1
5	10	14	20	7	1
6	14	10	14	10	1
7	10	10	20	5	1

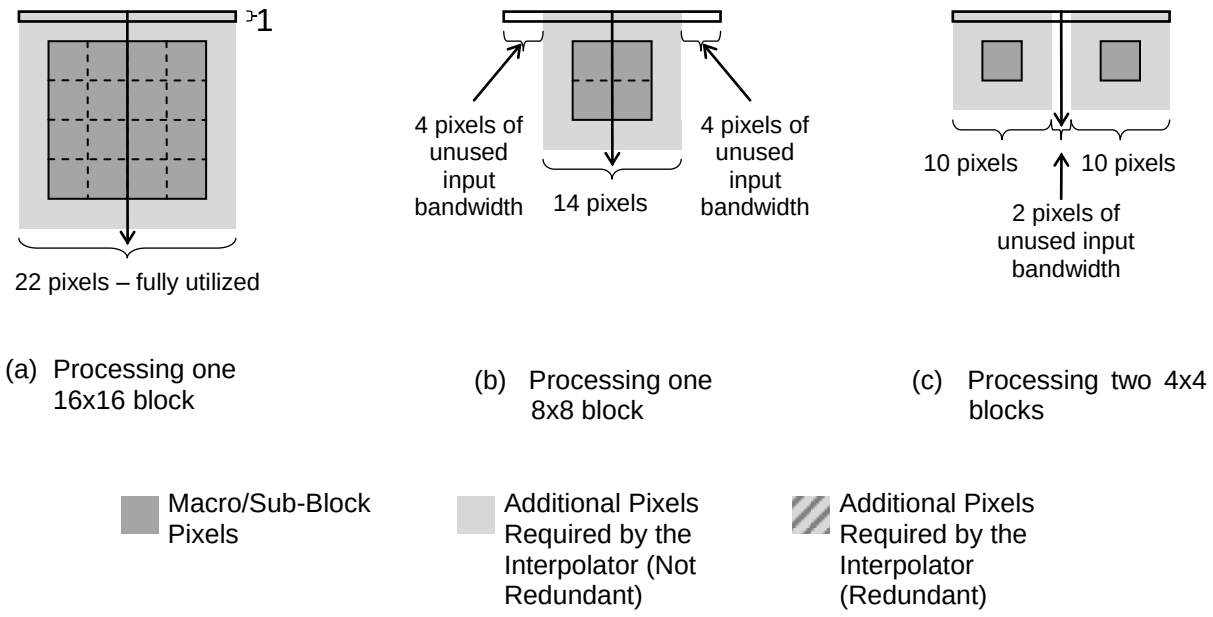


Figure 27 - Hardware utilization for the [M=1, N=4] design for 16x16, 8x8, and 4x4 blocks

Similarly, the horizontally scaled [M=1, N=2] FME design requires less block decomposition then the vertical designs. In this case, all subblocks can be processed in their entirety, except for the two 16 pixel wide blocks: 16x16 and 16x8. In this case the blocks have to be decomposed into two vertical sweeps. For example, the MB is decomposed into two parts, each 14 pixels wide, as is shown in Figure 28.

In general, both of the horizontally scaled designs benefit from the lack of decomposition by decreasing the amount of clock cycles required to process the subblocks. On the other hand, processing the less then fitting blocks has disadvantages, such as hardware utilization.

Table 5 - The Block Decomposition Schedule for the [M=1, N=2] Design

Subblock Type	Before Block Decomposition		After Block Decomposition		Number of Vertical Sweeps
	Subblock Dimensions		Subblock Dimensions		
	Width	Height	New Width	New Height	
i	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i
1	22	22	14	22	2
2	14	22	14	22	1
3	22	14	14	14	2
4	14	14	14	14	1
5	10	14	10	14	1
6	14	10	14	10	1
7	10	10	10	10	1

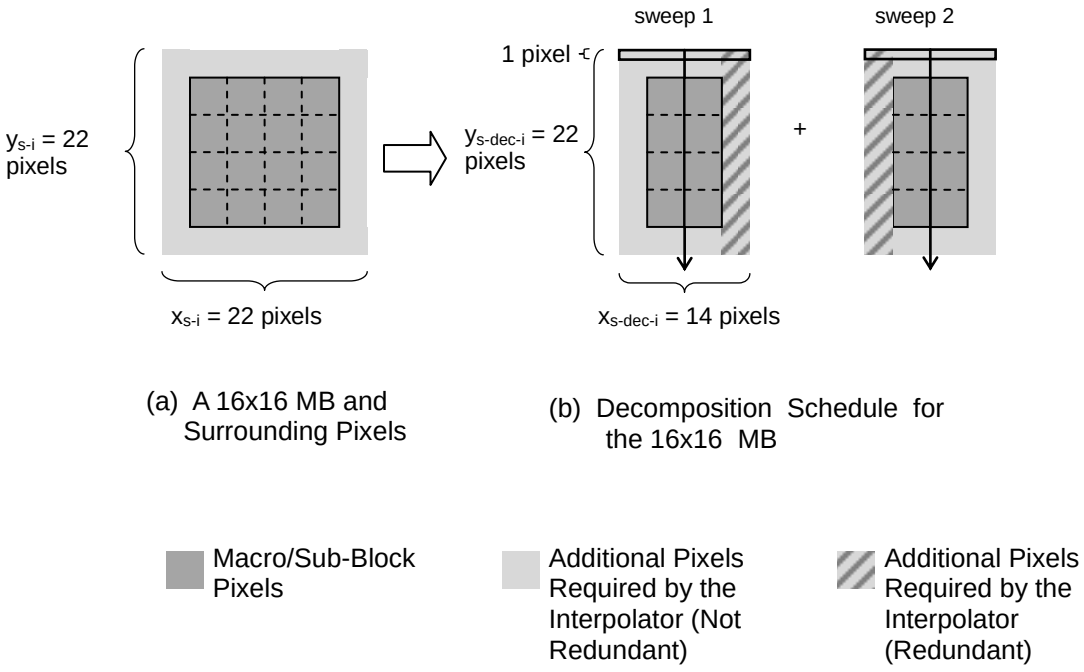


Figure 28 - Decomposition Schedule for a 16 x 16 Block for the [M=1, N=2] Design

4.2 Data Redundancy

The block decomposition approach leads to redundant calculations done by the IE on the surrounding pixels belonging to the decomposed sweeps. Data redundancy could potentially increase local memory bandwidth usage and waste hardware resources filtering the same pixels twice. The next section explains how we calculate the exact data redundancy percentages for each of the six FME designs, followed by a discussion on redundancy trends in vertical and horizontal scaling.

4.2.1 Calculating Data Redundancy

Data redundancy, $R_{M,,N}$, is the percentage difference between the total number of processed pixels when all the subblocks are in one piece, P_{total}, and the total number of pixels when the subblocks are decomposed, $P_{total\text{-}dec}$. For each scaled FME, we calculate percentage of data redundancy using the following equation:

$$R_{M,N} = \frac{P_{total-dec} - P_{total}}{P_{total}} \times 100\%$$

Equation 12 – Percent of Data Redundancy

Where,

$R_{M,N}$ – data redundancy for processing 41 subblocks (in pixels)

P_{total} – total number of pixels processed for all 41 subblocks before block decomposition

$P_{total\text{-}dec}$ – total number of pixels processed for all 41 subblocks after block decomposition

Equation 13 shows how to calculate P_{total}. $P_{block\text{-}i}$ is the total number of integer pixels for each subblock, in its entirety. It is calculated by multiplying the subblock dimensions which include the surrounding pixels. Next, we calculate $P_{type\text{-}i}$ which is the total number of pixels per subblock type. After block decomposition, we calculate $P_{block\text{-}dec\text{-}i}$ which is the total number of pixels when taking into account the number of sweeps required as well as the new $x_{s\text{-}dec\text{-}i}$ and $y_{s\text{-}dec\text{-}i}$ values. Similarly as in Equation 14, we calculate the new $P_{type\text{-}dec\text{-}i}$ and $P_{total\text{-}dec}$ values.

$$P_{block-i} = x_{s-i} \times y_{s-i}$$
$$P_{type-i} = P_{block-i} \times b_i$$
$$P_{total} = \sum_{i=1}^{7} P_{type-i}$$

Equation 13 – The Total Number of Pixels Processed for all Subblock Types

$$P_{block-dec-i} = x_{s-dec-i} \times y_{s-dec-i} \times s_i$$
$$P_{type-dec-i} = P_{block-dec-i} \times b_i$$
$$P_{total-dec} = \sum_{i=1}^{7} P_{type-dec-i}$$

Equation 14 - The Total Number of Pixels Processed for all Subblock Types After Block Decomposition

Where,

$x_{s\text{-}i}$ – width of an entire subblock, with surrounding pixels

$y_{s\text{-}i}$ – height of an entire subblock, with surrounding pixels

$x_{s\text{-}dec\text{-}i}$ – width of a decomposed subblock, with surrounding pixels

$y_{s\text{-}dec\text{-}i}$ – height of a decomposed subblock, with surrounding pixels

$P_{block\text{-}i}$ – number of pixels processed per single subblock *i*

$P_{type\text{-}i}$ – total number of pixels processed for an entire subblock type *i*

$P_{block\text{-}dec\text{-}i}$ – number of pixels processed per single subblock *i* after block decomposition

$P_{type\text{-}dec\text{-}i}$ – total number of pixels processed for an entire subblock type *i* after block decomposition

s_i – number of vertical sweeps required for subblock type *i*

b_i – total number of subblocks of type *i*

Finally, the total data redundancy is calculated as the percentage difference between the two totals: P_{total} and $P_{total\text{-}dec}$, based on Equation 12. Table 6 shows the total data redundancy percentages for each of the six scaled designs.

Table 6 - Data Redundancy for Six FME Scaled Designs

Type of Scaling	Scaled FME Designs	Data Redundancy $R_{M, N}$ (%)
Vertical	M=10, N=1	77%
	M=4, N=1	31%
	M=2, N=1	31%
Base Design	M=1, N=1	31%
Horizontal	M=1, N=2	5%
	M=1, N=4	0%

4.2.2 Data Redundancy in Vertically Scaled Designs

Generally, the block decomposition schedule depends on the width of the FME engine, although the [M=10, N=1] design is an exception to this trend. The same decomposition schedule for the [M=1, N=1] design is used for the [M=2, N=1] and [M=4, N=1] designs since their Interpolation Engines share the same input pixel array width of 10 pixels. Consequently, as shown in Table 6, all three designs share the same data redundancy of 31%.

Table 7 – Data Redundancy Analysis and Block Decomposition for the [M=1, N=1] Design

Before Block Decomposition						After Block Decomposition				
Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Pixels Per Subblock	Total Pixels Per Subblock Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Total Pixels Per Subblock	Total Pixels Per Subblock Per Type
		Width	Height			New Width	New Height			
i	n	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$P_{block\text{-}i}$	$P_{type\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$P_{block\text{-}dec\text{-}i}$	$P_{type\text{-}dec\text{-}i}$
1	1	22	22	484	484	10	22	4	880	880
2	2	14	22	308	616	10	22	2	440	880
3	2	22	14	308	616	10	14	4	560	1120
4	4	14	14	196	784	10	14	2	280	1120
5	8	10	14	140	1120	10	14	1	140	1120
6	8	14	10	140	1120	10	10	2	200	1600
7	16	10	10	100	1600	10	10	1	100	1600
				$\boldsymbol{P_{total}}$	**6340**				$\boldsymbol{P_{total\text{-}dec}}$	**8320**

In the case of [M=10, N=1], 396 out of 484 integer pixels from the 22x22 array must be processed twice. The redundancy values for the remaining five subblock types are shown in column 6 of Table 7. The data redundancy decreases as the size of the subblock decreases because the extent of the necessary decomposition is reduced. For example, the 8x4 subblock is

decomposed into two vertical sweeps of 4x4 subblocks, resulting in only 60 redundantly processed pixels.

The extensive block decomposition significantly increases the data redundancy. In this case, decomposition is performed both horizontally and vertically. In the case of the 16x16 MB, the number of redundantly processed pixels is 1116 as opposed to only 396 as in the [M=1, N=1] design. Overall, after decomposition, the IE performs a total of 4860 redundant pixel calculations, and the data redundancy is calculated to be 77%. This is the highest redundancy across all six scaled FME designs.

4.2.3 Data Redundancy in Horizontally Scaled Designs

As previously mentioned, the IE in the [M=1, N=4] design is wide enough to process all subblock sizes without any block decomposition. For this design, the data redundancy is 0%, as none of the pixels have to be processed twice. With respect to data redundancy, this is the best performing design.

In the case of the [M=1, N=2] design, only two out of seven types of subblocks require decomposition, resulting in a total of only 300 redundant pixels, and a total data redundancy is only 5%.

4.3 Hardware Utilization

We scale the FME design in hope to achieve an increase in data throughput and

processing speed. Hardware utilization tells us how efficiently we use the additionally instantiated hardware resources. When scaling the design and increasing the size of the hardware it is important to keep a high level of resource utilization and in turn achieve the greatest amount of speed up.

In our design space, the three factors which influence hardware utilization are *input data width variance, throughput variance* and *vertical alignment*. The next three sections explain the three factors in detail, followed by the derivation of equations used to calculate the hardware utilization numbers.

4.3.1 Input Data Width Variance

The IE can be potentially fully utilized when its input data bandwidth is fully utilized. This is the case when the Interpolation Engine's input pixel array width is smaller or equal to the subblock's width, as in the vertically scaled designs. For this reason, the Interpolation Engine's utilization for the vertically scaled designs can *potentially* be 100%. Vertical data alignment compromises this ideal result in the case of the [M=4, N=1] and [M=2, N=1] designs.

On the other hand, when the IE's input pixel array is wider than the subblock width, as is the case in horizontally scaled designs, some of the hardware is left idle during the processing of the smaller subblocks.

For example, some of the hardware from the [M=1, N=4] design must sit idle when processing subblocks that are less than 16 pixels wide. In particular, as shown in Figure 27 - (b), when processing 8-pixel wide subblocks, the IE (which are designed to process 22 pixels per

clock cycle) only process 14 pixels at a time. By only processing 14 pixels, the utilization of the interpolators, for that particular block, is reduced from 100% to 63%, according to the following calculation:

$$\frac{14\,pixels}{22\,pixels} \times 100\% = 63\%$$

Equation 15 – Interpolation Engine Utilization Example

In addition only two out of four sets of PUs are utilized at that time. For 4-pixel wide subblocks, as shown in Figure 27 - (c), the utilization can be increased by processing two subblocks at a time [16]. The total combined width (20 integer pixels) of each input array, however, is still less than the maximum capacity of the processing hardware (22 integer pixels per clock cycle). Similarly, even with the 20-pixel wide input arrays, only two out of four sets of PUs are utilized.

Overall, the utilization values for the horizontally scaled design's IE are decreased to 86% and 79% for [M=1, N=2] and [M=1, N=4] respectively. Their PUs also exhibit a significant drop in utilization, from the base design's 54% to 41% and 33%. Note that the PU utilization values are a result of a combination of decreased utilization causes, such as vertical alignment and throughput variance described next.

4.3.2 Throughput Variance

Due to the workload difference between the IE and the PUs, the PUs cannot be fully utilized even when the IE is. For example, when the [M=1, N=1] design is used to process a 4x4

subblock, the interpolators need to process an entire 10x10 array of pixels while the PUs only need to process the elementary 4x4 pixels. Consequently, the IE becomes the critical path and the PUs are forced to stall while the 6 rows of surrounding pixels are processed. Similarly, when processing an 8x4 subblock, the interpolators require 14 clock cycles, while the PUs require only 8 and sit idle for the remaining 6. The timing diagram in Figure 29 illustrates the idle cycles. The *x-axis* represents a row number, which is a single pixel height in the case of the [M=1, N=1] design, meaning that a single row corresponds to a single clock cycle.

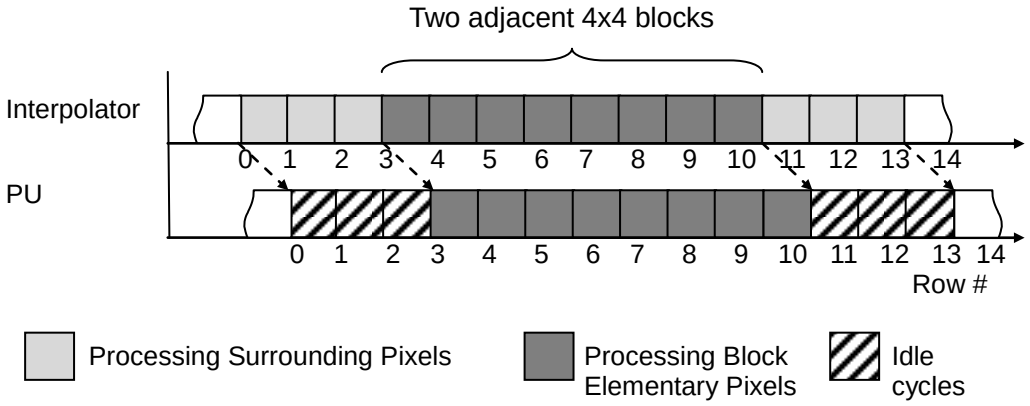


Figure 29 - Timing Diagram for Processing a 4x4 Block Using the [M=1, N=1] Design

The effect that throughput variance has on the utilization of the PU depends on the height of the vertical sweep for each subblock. For the 16x16 block, the vertical sweep has a height of 22 pixels, and during its processing the PU is used 73% of the time, according to the following calculation:

$$\frac{16\,pixels}{22\,pixels} \times 100\% = 73\%$$

Equation 16 - PU Utilization Example

When processing blocks with smaller vertical sweep heights, such as the 4x4, the utilization is only 40%. The PU utilization numbers for processing of all seven subblock types are shown in Table 8. Overall, the [M=1, N=1] design achieves a PU utilization of only 54%.

Table 8 – The Throughput Variance and PU Utilization of the [M=1, N=1] Base Design

Subblock Type	Subblock Dimensions (With Surrounding Pixels)		Subblock Dimensions (Elementary Pixels Only)		Number of Vertical Sweeps	Processing Time of the IE (cc.)	Processing Time of the PU (cc.)	PU Utilization per Subblock Type (%)
i	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$x_{e\text{-}i}$	$y_{e\text{-}i}$	s_i	$T_{cc}(y_{e\text{-}i})$	$T_{cc}(y_{s\text{-}i})$	$U_{PU\text{-}i}$
1	22	22	16	16	4	22 x 4	16 x 4	73
2	14	22	8	16	2	22 x 2	8 x 4	73
3	22	14	16	8	4	14 x 4	8 x 4	62
4	14	14	8	8	2	14 x 2	4 x 4	57
5	10	14	4	8	1	14 x 1	2 x 4	57
6	14	10	8	4	2	10 x 2	2 x 4	40
7	10	10	4	4	1	10 x 1	1 x 4	40

4.3.3 Vertical Alignment

The boundaries between the elementary subblock pixels and the surrounding pixels need to be aligned with the input pixel array in order to synchronize with the transpose operation of the various PU designs. Each clock cycle, the input pixel array must be aligned to contain either:

1. no elementary pixels in the entire input pixel array, or
2. some elementary pixels on all of the rows of the input pixel array.

For example, for the [M=2, N=1] design, Figure 30 - (b) shows the array of pixels that are

required to calculate the SATD value of a 4x4 subblock. The array is processed in six clock cycles. As shown, the first set of 2x10 pixels contains a row from the preceding subblock while the last set of 2x10 pixels contains a row from the subsequent subblock. Consequently pixels from the 4x4 subblock are completely covered by the two 2x10 input arrays at clock cycles 3 and 4.

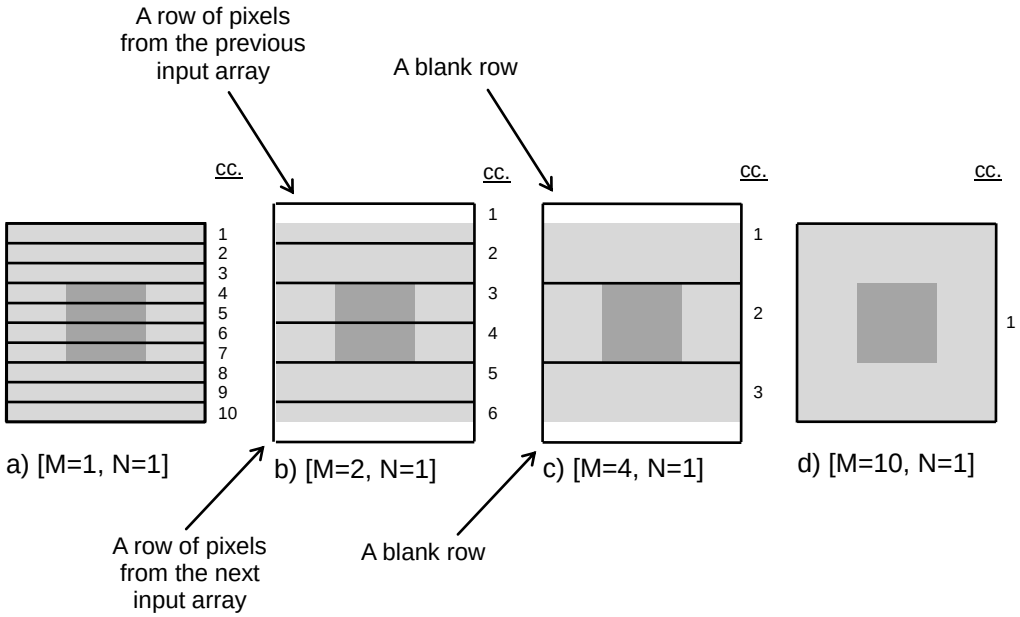


Figure 30 - Data Alignment for [M=1, N=1], [M=2, N=1], [M=4, N=1] and [M=10, N=1]

Similarly, for the [M=4, N=1] design, as shown in Figure 30 - (c), two blank rows are inserted, one into the top and one into the bottom input pixel arrays in order to properly align the middle 4x10 input pixel array onto the 4x4 elementary pixel subblock. Additional blank rows are also required when processing the 8x4 and 16x4 subblocks. As shown in Table 9, these additional rows reduce the interpolator utilization to 87% and PU utilization to 46%. The [M=1, N=1] design, as shown in Figure 30 - (a), processes one row of 10 pixels every clock cycle. Consequently, no alignment is necessary since the row either contains a row of subblock pixels

or it does not.

Finally, the [M=10, N=1] design in Figure 30 – (d) has a greater amount of block decomposition allowing its 10x10 input pixel array to be centered around the elementary pixels and ideally aligned with every clock cycle.

4.3.4 Calculating Hardware Utilization

The combined effect of the three above discussed factors results in the overall hardware utilization numbers which are shown in Table 9. The utilization of the IE and PUs is examined individually because the factors which influence them are different. Columns 4 and 6 in Table 9 summarize factors responsible for reducing the hardware utilization from the ideal 100%.

Table 9 - Hardware Utilization Summary

Type of Scaling	Scaled FME Designs	IE Hardware Utilization		PU Hardware Utilization	
		$U_{IE-M,N}$	Source	$U_{PU-M,N}$	Source
Vertical	M=10, N=1	100%	(FULL)	99%	(FULL)
	M=4, N=1	87%	Vertical alignment	46%	Throughput variance + vertical alignment
	M=2, N=1	99.8%	Vertical alignment	54%	Throughput variance
Base Design	M=1, N=1	100%	(FULL)	54%	Throughput variance
Horizontal	M=1, N=2	86%	Data width	41%	Data width + throughput variance
	M=1, N=4	79%	Data width	33%	Data width + throughput variance

Hardware Utilization of the Interpolation Engine

We calculate the hardware utilization of the IE by examining how full the input pixel array is every clock cycle, across all clock cycles required to process a single set of 41 subblocks. In this case, the input pixel array is M x (4N+6), as shown in Figure 10. The amount of utilization of the input pixel array maps directly to the utilization of the available IE hardware. For example, when the input pixel array is full, all of the IE processing hardware is 100% utilized. On the other hand, if only half of the input pixel array is populated with pixels to be processed, the hardware is only 50% utilized for that particular clock cycle.

To calculate the overall IE utilization, first we multiply each input pixel array utilization value by the number of vertical sweeps per block, s_i, and the number of blocks per type, b_i, in order to calculate the total number of clock cycles the particular utilization occurs for. Next, this value is summed across all seven subblock types, shown in the numerator of Equation 19. Further, the summation is divided by s_i, and b_i in order to achieve the overall average hardware utilization across all 41 subblocks.

To calculate the exact input pixel array utilization we evaluate the individual horizontal and vertical utilizations, as shown in Equation 19. The horizontal utilization takes into account input data width variance. It is calculated as the ratio between the input width of the decomposed subblock, $x_{s\text{-}dec\text{-}i}$, and the width of the IE, 4 x N+6, both in terms of number of pixels.

The vertical utilization approximation value takes into account the effect of vertical alignment. It is calculated by examining a single vertical sweep length, $y_{s\text{-}dec\text{-}i}$. $C_{ys\text{-}dec\text{-}i}$ is the

manually evaluated number of clock cycles it takes to process a single sweep of length $y_{s\text{-}dec\text{-}i}$ pixels. This number includes the blank rows used for vertical alignment purposes. The $y_{s\text{-}dec\text{-}i}$/M fraction is the theoretical number of clock cycles it could take the engine to process a sweep of length $y_{s\text{-}dec\text{-}i}$. For example, in Figure 30 - (c), the vertical utilization is the vertical length, $y_{s\text{-}dec\text{-}i}$ = 10, divided by the input array height, M = 4. This results in a value of 2.5, which is the number of clock cycles it could theoretically take the FME to process this block. We label this theoretical because it is not an integer value and therefore an illegal number of clock cycles, but in this case valid since it is used as an estimation of hardware utilization. Further, we divide 2.5 by $c_{ys\text{-}dec\text{-}i}$, which is in this case 3, to achieve a vertical utilization of:

$$\frac{2.5}{3} \times 100\% = 83\%$$

Equation 17 - IE Utilization for the [M= 2, N=1] Design

This percentage accounts for the two blank rows inserted for alignment purposes. An alternate way of calculating this is by realizing the over three clock cycles and 12 available rows, only 10 were filled with pixels, again resulting in the same value:

$$\frac{10}{12} \times 100\% = 83\%$$

Equation 18 – IE Utilization for the [M= 2, N=1] Design

$$U_{IE-M,N} = \frac{\sum_{i=1}^{7}\left(\begin{matrix}input\_pixel\_array \\ utilization\end{matrix} \times s_i \times b_i\right)}{\sum_{i=1}^{7} s_i \times b_i} \times 100\%$$

$$= \frac{\sum_{i=1}^{7}\left(\begin{matrix}horizontal \\ utilization\end{matrix} \times \begin{matrix}vertical \\ utilization\end{matrix} \times s_i \times b_i\right)}{\sum_{i=1}^{7} s_i \times b_i} \times 100\%$$

$$= \frac{\sum_{i=1}^{7}\left(\frac{x_{s-dec-i}}{4 \times N + 6} \times \frac{\frac{y_{s-dec-i}}{M}}{c_{ys-dec-i}} \times s_i \times b_i\right)}{\sum_{i=1}^{7} s_i \times b} \times 100\%$$

Equation 19 - Hardware Utilization Approximation of the Interpolation Engine

Where,

$U_{IE-M,N}$ – percentage of hardware utilization of the IE

$x_{s-dec-i}$ – width of a decomposed subblock, with surrounding pixels

$y_{s-dec-i}$ – height of a decomposed subblock, with surrounding pixels

s_i – number of vertical sweeps required for subblock type i

b_i – total number of subblocks of type i

$c_{ys-dec-i}$ – number of clock cycles required to process a single sweep of $y_{s-dec-i}$ pixels

The calculations of $U_{IE-M,N}$ for all six designs are shown in Appendix B.

Hardware Utilization of the Processing Unit

In FME pipeline, the IE is the critical path, while the PUs work and then have to wait for further data. In order to calculate PU utilization we have to take into account input throughput variance. In addition, the horizontally scaled FME designs experience input data width variance which causes certain PU sets to sit idle when processing smaller blocks.

In Equation 20, in the numerator we calculate the number of clock cycles it would take a single set of PUs to process a workload of 41 subblocks. We divide this value by N, the number of available PU sets in the particular design. Further, we divide this value by $T_{cc\text{-}M,N}$, which is the total processing time of the IE. This way we achieve the percentage of time the PUs are busy with respect to the critical path of the IE.

To calculate the workload, for each subblock we evaluate the number of 4x4 blocks it is composed of, $x_{e\text{-}i}/4$ x $y_{e\text{-}i}/4$. Next we multiply this value by, *ceil*(4/M), the number of clock cycles it takes a PU to process each 4x4 subblock based on its throughput.

$$U_{PU-M,N} = \frac{\sum_{i=1}^{7}\left(\frac{x_{e-i}}{4}\times\frac{y_{e-i}}{4}\right)\times ceil\left(\frac{4}{M}\right)}{T_{cc-M,N}\times N}\times 100\%$$

Equation 20 –Hardware Utilization Approximation for the Processing Unit

Where,

$U_{PU\text{-}M,N}$ – percentage of hardware utilization of the PU for an FME design M,N

$x_{e\text{-}i}$ – width of an entire subblock, elementary pixels only

$y_{e\text{-}i}$ – height of an entire subblock, elementary pixels only

$T_{cc\text{-}M,N}$ – total number of clock cycles required to process 41 subblocks

4.4 Processing Time

As we scale up the design we gain computational processing power and in turn decrease the total processing time which can be measured in terms of the number of clock cycles it takes an FME engine to process a single set of 41 motion vectors, shown in Equation 21. We choose to measure the processing time of a single set of 41 motion vectors because it is the basic building block of encoding, corresponding to a single MB. The $T_{cc\text{-}M,N}$ value can be used to further calculate performance numbers for various video sizes, such as target resolutions and frame rates. Table 10 lists the $T_{cc\text{-}M,N}$ values for the six scaled designs.

$$T_{cc-M,N} = \sum_{i=1}^{7} \left(c_{ys-dec-i} \times s_i \times b_i \right)$$

Equation 21 – Total Number of Clock Cycles

Where,

$T_{cc\text{-}M,N}$ – total number of clock cycles required to process 41 subblocks

s_i – number of vertical sweeps required for subblock type i

b_i – total number of subblocks of type i

$c_{ys\text{-}dec\text{-}i}$ – number of clock cycles required to process a single sweep of $y_{s\text{-}dec}$ pixels

Table 10 - Total Number of Clock Cycles Required for Processing 41 Subblocks

Type of Scaling	Scaled FME Designs	$T_{cc\text{-}M,N}$ (cc.)
Vertical	M=10, N=1	112
	M=4, N=1	240
	M=2, N=1	417
Base Design	M=1, N=1	832
Horizontal	M=1, N=2	552
	M=1, N=4	366

4.5 Chapter Summary

In this chapter we presented a detailed design space exploration of the scalable FME architecture. First we introduced the concept of block decomposition, which is elementary in understanding the processing dataflow of the various sized FME engines. Next we present how to derive the incurred data redundancy, a consequence of block decomposition. We define three causes which can vary hardware utilization and we then formulate them into a single equation. Finally, we calculate the processing time for a single MB in the current frame, in terms of clock cycles, which becomes one of the elementary components used to derive comparison medians for the six scaled designs, such as speedup, cost-performance product and relative hardware utilization.

Chapter 5 – Experimental Evaluation

In this chapter we analyze the scalability of the H.264/AVC full-search FME algorithm on FPGAs by exploring and interpreting the hardware implementation results. We analyze the hardware resources used, the maximum achievable clock frequency, and the total running times of each design, and perform comparative analysis by exploring the cost-performance product, speedup and utilization. Finally, we conclude our analysis by assigning each design an overall scalability ranking as well as an achievable target video resolution.

5.1 Target Device and Tools

All six designs shown were implemented in VHDL and synthesized using the Xilinx Synthesis Tool (XST), followed by MAP and PAR in the Xilinx Integrated Software Environment (ISE), Version 10. The device used was the Xilinx XC5VLX85T [44], *speedgrade* -2. The implementation was simulated and tested using custom generated test benches in VHDL Simili [45].

Xilinx [46] manufactures two main FPGA architecture types of FPGAs: the Virtex and the Spartan line. The Spartan FPGAs are marketed towards smaller low-cost smaller applications. The Virtex line is capable of handling computation heavy applications, and is geared towards more expensive applications requiring maximum performance. The Virtex line is further divided

into subcategories specialized in particular processing requirements: the LX, the SX and the FX [47]. The FX series poses an embedded PowerPC hard-core, meant for software-hardware co-processing. The SX line is designed for signal processing applications as these chips have a large number of DSP blocks. The LX series is geared towards raw processing power, containing the greatest amount of slices. The Virtex line was chosen because it is a very mature device, and it fits the requirements for real time video processing.

This work was developed on the LX platform. The FME algorithm was implemented entirely using raw distributed logic, instead of placing arithmetic operations into hard DSP blocks. This design decision was made based on a few under-utilization concerns. First, the Virtex 5 DSP48E block [48] is made up of a multiplier followed by an adder. In our design the FME's multipliers have been optimized into adders and shifters, as was shown in the case of the FIR filter. Similarly, the DSP blocks have 18 and 30 bit vectors and 48 bit output vectors, where the bit-width of the input pixels in the design is 8 bits, reaching a maximum of 14 bits. Thus using the DSP blocks would waste the available built in resources and decrease the overall utilization.

Further, migrating the algorithm implementation from distributed logic into DSP blocks would require extensive routing between the hard wired blocks which would waste distributed logic resources, offsetting the logic saved by the use of the hard blocks. A common optimization of the use of the DSP blocks is the transformation of the data flow from an adder tree to a carry chain [48] which would spare distributed logic use. But an implementation of this type would not be a fair comparison to our scaled designs since it requires an extensive algorithmic level changes as well as hand optimizations.

In addition, optimizing our design on a particular DSP block architecture does not

guarantee similar results on a different FPGA device. The hard-block architectures can vary significantly and even with forward compatibility they can be underutilized [49][50]. The goal of this project was to keep the design as portable as possible and to allow for deployment on an evolution on FPGAs.

5.2 Performance Analysis

The overheads in a parallel system that limit its scalability need to be identified in order to improve parallel algorithm design and the development of efficient high performance parallel systems. Such overheads may be broadly classified into two components. The first one is intrinsic to the algorithm itself and in the case of the FME it arises due to factors such as block decomposition and data redundancy. The second one is due to the interaction between the algorithm and the architecture it is employed on, and it arises due to work-imbalance, resource allocation and cost inefficiencies.

We define the notion of overhead functions associated with the different algorithmic and architectural characteristics in order to quantify the scalability of the FME engine as a parallel system. We design and implement a scalable high-performance FME algorithm platform that incorporates these methods for quantifying the overhead functions. We isolate the algorithmic and architectural overheads and examine their influence on the processing execution time. We use this system to study the scalability characteristics of six different instantiations on a Xilinx FPGA.

5.2.1 Hardware Implementation Results

First, we define fundamental limiting factors, such as resource use and data throughput. Next we will present our scalability analysis and evaluation methodology in order to gain insight into how the system performance is affected by these design tradeoffs.

Table 11- Implementation Results - Xilinx XC5VLX85T FPGA

Type of Scaling	Scaled FME Designs	Resource Cost		$f_{max\text{-}M,N}$ (MHz)	Data Throughput	
		$LUT_{M,N}$ (K)	$REG_{M,N}$ (K)		$T_{w\text{-}M,N}$ (µs)	MB/sec*
Vertical	M=10, N=1	25.5	28.7	350	0.3	3,125,000
	M=4, N=1	20.7	23.4	250	1	1,041,667
	M=2, N=1	12.8	14	337	1.2	808,153
Base Design	M=1, N=1	7.5	8.6	328	2.5	394,231
Horizontal	M=1, N=2	14.9	16.7	169	3.3	306,159
	M=1, N=4	29.5	32.8	140	2.6	382,514

* MB/sec – macroblocks per second

The ISE Place and Route (PAR) results are shown in Table 11. Columns 3 and 4 show the amount of hardware resources needed for each scaled design, in terms of Look-Up Table count ($LUT_{M,N}$) and registers ($REG_{M,N}$) and column 5 shows the variation in the maximum achievable clock frequency.

Data throughput in columns 6 and 7 is examined in order to quantify the performance and determine what kind of video resolution each design can handle. The computation time of the full-search FME is picture independent, therefore we define the total wall clock time, $T_{w\text{-}M,N}$, as the maximum running time an FME engine takes to process all 41 motion vectors for a single MB in the current frame and generate the corresponding SATDs. This time is dependent on the maximum achievable frequency, $f_{\text{max-M,N}}$, and the number of clock cycles it takes to encode a single MB, $T_{cc\text{-}M,N}$, derived in Section 4.4. We calculate $T_{w\text{-}M,N}$ using the following equation:

$$T_{w-M,N} = T_{cc-M,N} \times f_{\max-M,N}$$

Equation 22 – Total Wall Clock Time

We can calculate data throughput in terms of macroblocks, $MB_{M,N}$, and how many can be processed per second by each design. $MB_{M,N}$ depends on the maximum achievable frequency and the number of clock cycles each engine requires to process a single MB, as shown in Equation 23.

$$MB_{M,N} = \frac{f_{\max-M,N}}{T_{cc-M,N}}$$

Equation 23 - Macroblocks per Second

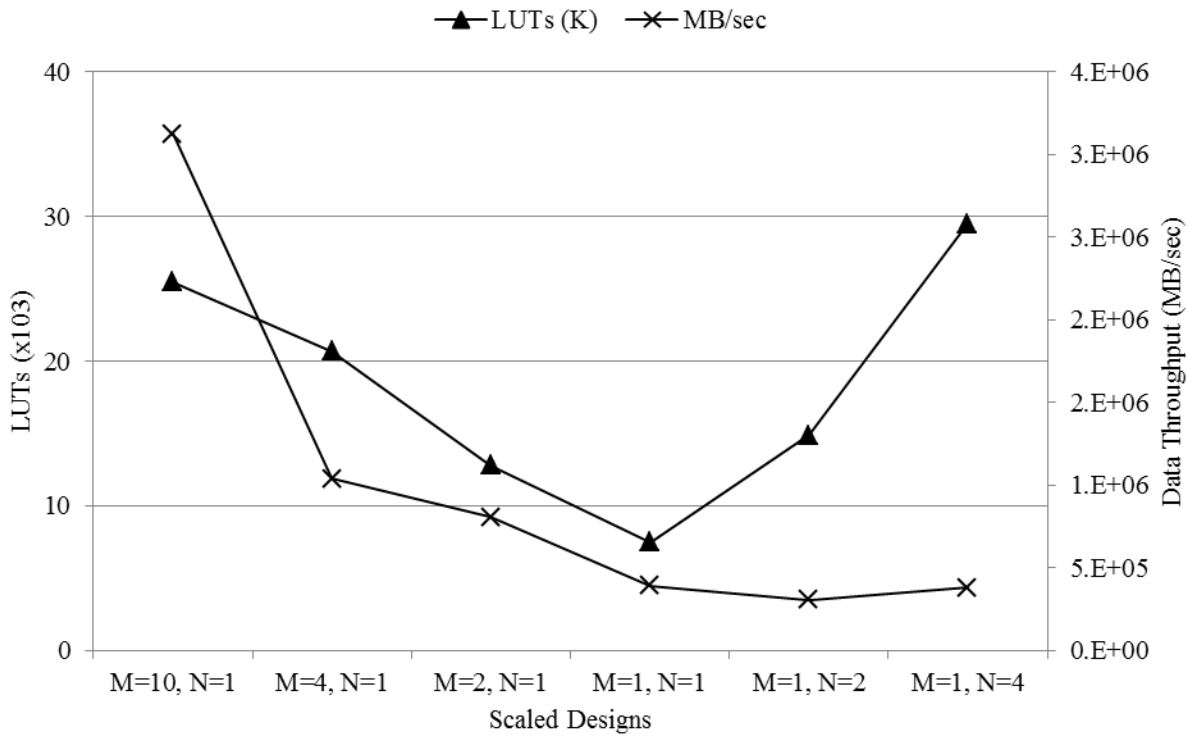


Figure 31 - LUT Count Compared with Data Throughput

Figure 31 shows the throughput compared to the resource use for the six scaled designs. We observe that the LUT count, rooted at the base design, increases in both directions with horizontal and vertical scaling. The throughput on the other hand plateaus with horizontal scaling

not providing much advantage despite being allocated the additional hardware resources. This is due to poor hardware utilization, discussed next.

5.2.2 Scalability Analysis

One approach to quantifying scalability is to examine the cost-performance product – $P_{w\text{-}M,N}$, the results of which are graphed in Figure 32 and calculated based on Equation 24. In this case, cost is the amount of hardware resources used, (number of LUTs), and performance is the total execution time of 41 vectors for each scaled design, $T_{w\text{-}M,N}$.

$$P_{cc-M,N} = LUT_{M,N} \times T_{cc-M,N}$$
$$P_{w-M,N} = LUT_{M,N} \times T_{w-M,N}$$

Equation 24 - Cost-Performance Product

Generally, we want to minimize the cost-performance product. The base design is used as the reference to which we compare our designs, and as we scale up, ideally, it is desired that the increase in cost to be equal to the increase in performance, or a decrease in total execution time. For example, if the LUT count was doubled and as a result the execution time was halved, the cost-performance product, would be the same for the base as well as for the particular scaled design. This is referred to as linear scaling.

$$P_{BASE} = LUT_{M,N} \times T_{w-M,N}$$
$$P_{SCALED} = \left(LUT_{M,N} \times 2\right) \times \left(\frac{T_{\text{sec}-M,N}}{2}\right) = P_{BASE}$$

Equation 25 - Linear Scaling in Terms of the Cost-Performace Product

On the other hand, if P_{SCALED} is greater than P_{BASE}, then the added resource cost does not yield a proportional amount of benefit in terms of execution time and we classify this as bad scaling. Similarly, if P_{SCALED} is less than P_{BASE}, then the achieved speedup is not proportional to what was paid for in terms of resource cost and we classify this as good scaling.

$$\begin{aligned} P_{M,N} &< P_{BASE} \quad && good\_scaling \\ P_{M,N} &= P_{BASE} && linear\_scaling \\ P_{M,N} &> P_{BASE} && bad\_scaling \end{aligned}$$

Equation 26 - Scalability Properties

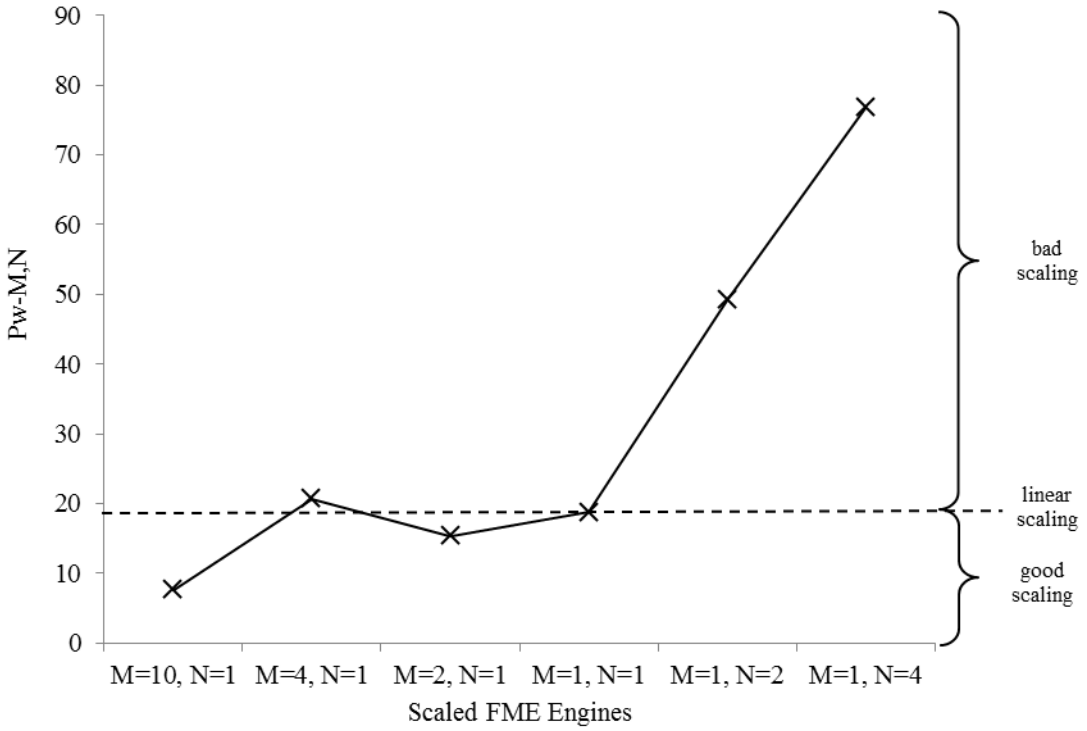


Figure 32 - Product of Cost and Total Execution Time

In Figure 32 we show graphed results of the scaled $P_{w\text{-}M,N}$. With respect to the base design, the two vertically scaled FMEs outperform linear scaling, while both of the horizontally scaled designs are distinctly above the linear scaling line and therefore scale poorly. Poor scaling is a

result of resource underutilization and inefficient use of the additional hardware resources. We can quantify utilization by examining cost ratios and processing speedup, defined next.

Relative cost is defined in terms of the FPGA hardware resources used, in terms of LUTs and registers. The ratio of the increase in cost, $C_{M,N}$, with respect to the base design [M=1, N=1], is calculated based on Equation 27 and is shown in Table 12, column 4.

$$C_{M,N} = \frac{LUT_{M,N}}{LUT_{M=1,N=1}}$$

Equation 27 – Relative Hardware Resource Cost

We define speedup, $S_{M,N}$, as the ratio between the processing time of the 41 motion vectors, $T_{w\text{-}M,N}$, and the processing time of the base design, based on the following equation:

$$S_{M,N} = \frac{T_{\text{sec}-M=1,N=1}}{T_{\text{sec}-M,N}}$$

Equation 28 - Speedup

Relative utilization is the ratio between the speedup and increase in cost:

$$U_{FME-M,N} = \frac{S_{M,N}}{C_{M,N}}$$

Equation 29 - Relative Efficiency

Table 12 - Performance Analysis

Type of Scaling	Scaled FME Designs	Speedup	Increase in Resources Ratio	Relative Utilization
		$S_{M,N}$	$C_{M,N}$	$U_{FME\text{-}M,N}$
Vertical	M=10, N=1	7.9	3.4	2.33
	M=4, N=1	2.7	2.8	0.97
	M=2, N=1	2	1.7	1.17
Base Design	M=1, N=1	1	1	1
Horizontal	M=1, N=2	0.8	2	0.38
	M=1, N=4	0.9	3.9	0.23

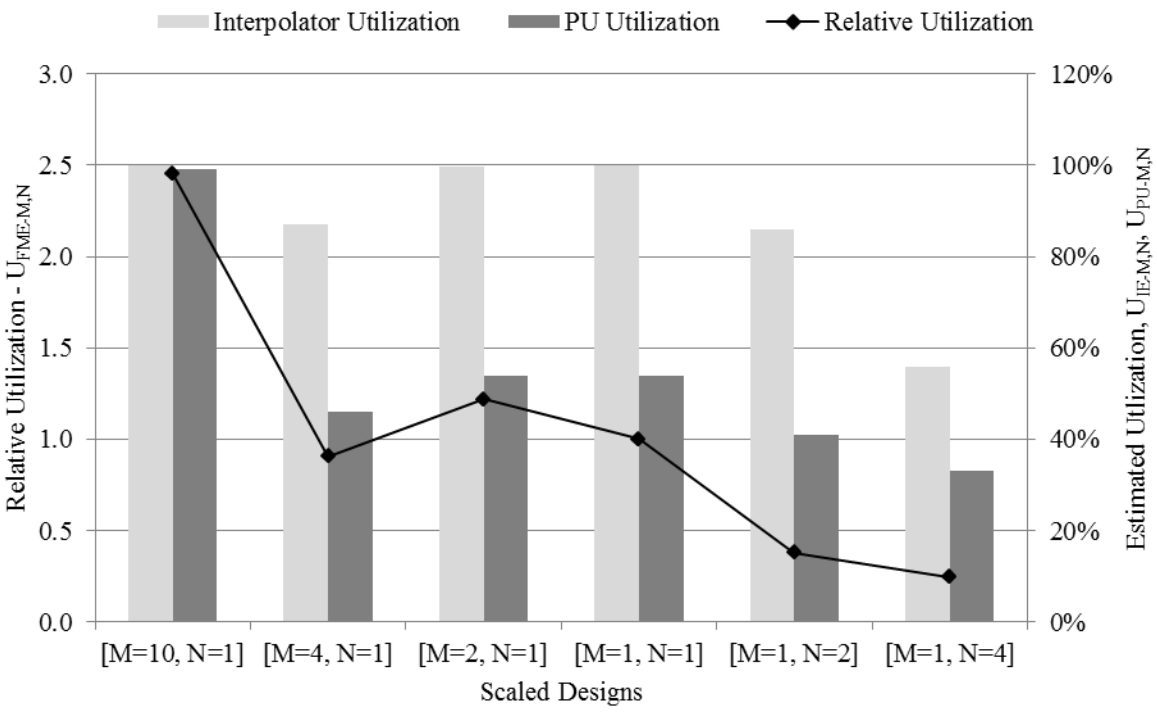


Figure 33 – Measured Overall Relative Utilization and Estimated Utilization

Figure 33 shows the observed implementation-based relative hardware utilization of the entire FME engine, $U_{FME\text{-}M,N}$, compared with the estimated percentage of utilization of the IE,

$U_{IE\text{-}M,N}$ and the PU units, $U_{PU\text{-}M,N}$ (Section 4.3.4). We observe that the overall utilization is influenced dominantly by the PU utilization rather than the IE's. The PU units consume more hardware resources in comparison to the IE, and therefore have a greater impact on the overall utilization results. Secondly, in terms of data flow, the IE is the bottleneck in the FME pipeline, hence the PUs are forced to stall thus lowering their utilization. Balanced workloads between the IE and PUs have a positive effect on overall utilization. In the case of the [M=10, N=1] design, the PUs do not have to stall and their throughput rate is equivalent to that of the IE, hence the overall design utilization is high.

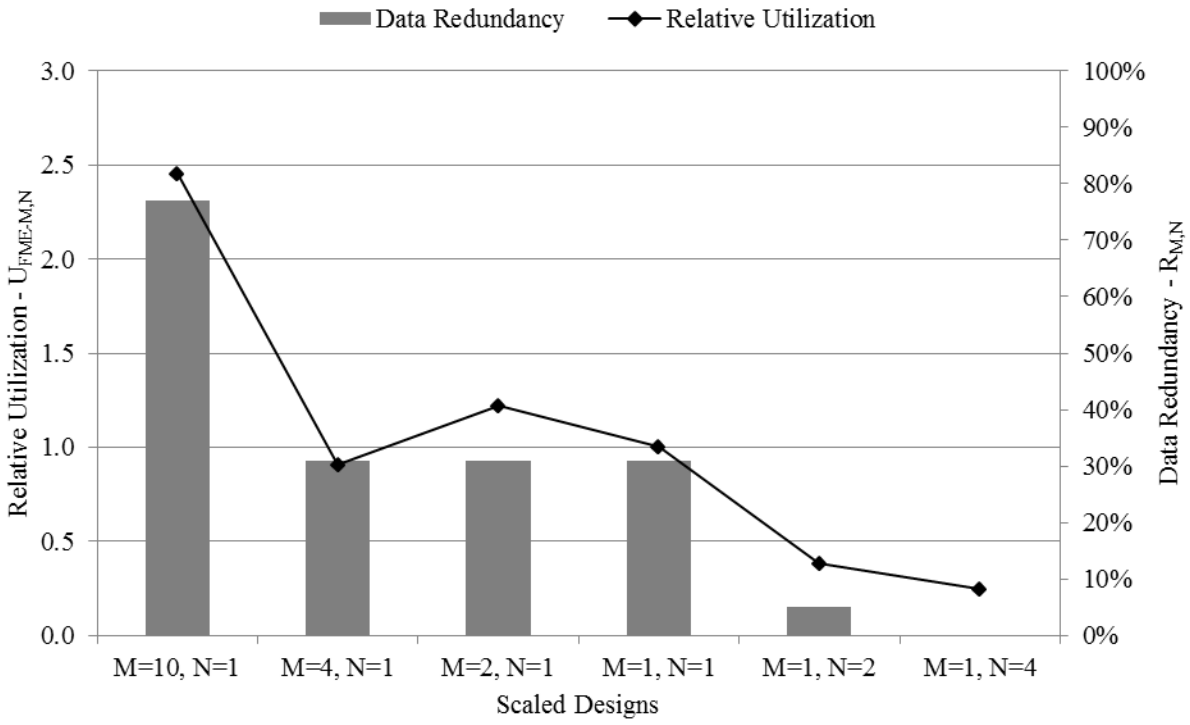


Figure 34 – Relevant Utilization and Data Redundancy

Figure 34 compares the relative utilization to the previously calculated data redundancy. We observe that eliminating the unwanted data redundancy is accompanied by an unwanted reduction in overall utilization. The horizontally scaled designs benefit from low data redundancy,

but do not scale as well as the vertical ones.

To further illustrate this trade off, Figure 35 shows the data throughput numbers. Despite having the highest data redundancy, the [M=10, N=1] design has a significantly larger data throughput. In particular, at an additional increase in cost of only 25%, this design gains close to three times the data throughput with respect to the [M=4, N=1] design.

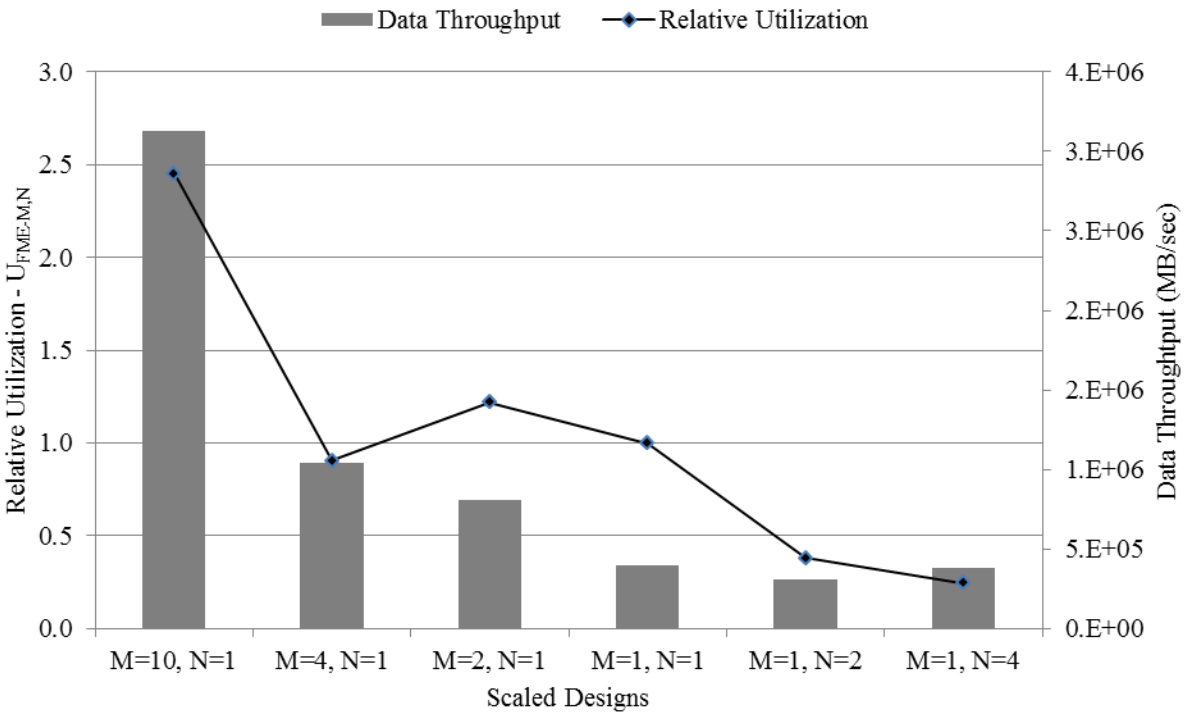


Figure 35 – Relative Utilization and Data Throughput

In conclusion, we assign a scalability ranking to each of the six designs, based on the relative utilization, $U_{FME\text{-}M,N}$. Here a value of '1' is assigned to the best scaled design [M=10, N=1], which makes the most use out of its additional resources and achieves the highest throughput processing performance.

Table 13 - Scalability Ranking

Type of Scaling	Scaled FME Designs	Scalability Ranking
Vertical	M=10, N=1	1
	M=4, N=1	4
	M=2, N=1	2
Base Design	M=1, N=1	3
Horizontal	M=1, N=2	5
	M=1, N=4	6

When scaling, the increase in performance should ideally be the same as the increase in cost. The vertically scaled designs outperform this expectation due to the more compact and efficient high-throughput PU designs. In particular, as discussed in Section 3.2.1, the design shown in Figure 17 - (c) completely eliminates the transpose shift registers while processing four rows of pixels per clock cycle. Similarly, the two-row-per-clock-cycle design shown in Figure 17 - (b) requires only half as many transpose shift registers in order to achieve the same performance as would two instantiations of the one-row-per-clock cycle design shown in Figure 17 - (a).

However, the increase in utilization is partially offset by an increase in data redundancy. As shown in Table 9, the [M=10, N=1] design has a 100% utilization of the IE as well as the PU. This design performs the most amount of work, some of which is redundant, but this increase in computation is compensated by the additional concurrency. Furthermore, the [M=10, N=1] design is used to process subblocks that are four pixels high, each V-IPU (as shown in Figure 14) requires only five FIRs (instead of ten) and the FIFO is reduced to 10 pixels. Overall, in terms of raw clock cycles, the [M=10, N=1] design is the fastest while also consuming the most area.

The horizontally scaled designs, on the other hand, show a decrease in utilization as

expanding parallelism only along the row quickly degrades utilization. Their PU sets do not simplify with the increase in parallelism, and thus increase in hardware resource use. Also, the smaller subblocks underutilize the wider instantiated hardware. These results show that it is important to create designs that can increase parallelism while maintaining a high degree of hardware utilization and clock frequency.

5.2.3 Frequency Analysis

The maximum achievable clock frequency, $f_{max\text{-}M,N}$ was recorded for each design and was previously shown in Table 11. The vertically scaled designs consistently achieved higher clock frequencies than the horizontally scaled ones, as measured by the ISE PAR.

Each scaled design, operating on a single clock frequency, was configured as a separate project and implemented individually on the FPGA. The XST *Optimization Goal* (OPT_MODE) [51] was set to optimize for *speed*, as opposed to area. This constraint prioritizes the reduction of logic levels in an attempt to increase frequency.

Within the VHDL code, the instantiation of the individual blocks was done using the *generic* statement and the scalable factors M and N. The block structure of the code is identical to the block structure of the FME algorithm described in Chapter 3. The generation of each of the six designs was done by simply changing the values of M and N before synthesis. This approach allowed for consistency between the scaled FME designs, removing the possibility for performance variations caused by coding style differences. The XST Synthesis Timing Report, for all six designs, had an identical estimated maximum frequency of 444 MHz, confirming the

intended code consistency and unbiased approach.

Figure 36 compares the two cost-performance products, $P_{w\text{-}M,N}$ and $P_{cc\text{-}M,N}$, where the former is calculated using the wall clock, $T_{w\text{-}M,N}$, which takes into account the total running time required for processing 41 subblocks, and the latter is clock frequency independent. The two lines show the effect of maximum achievable clock frequency on each design. We observe that the overall pattern of the two metrics is similar, with one exception: $P_{cc\text{-}M,N}$ monotonically increases, whereas $P_{w\text{-}M,N}$ has a slight bump in the case of the [M=4, N=1] design. This is due to a decrease in f_{max} and which results in a lesser score in terms of the scalability ranking then it would have had we not taken frequecy into account.

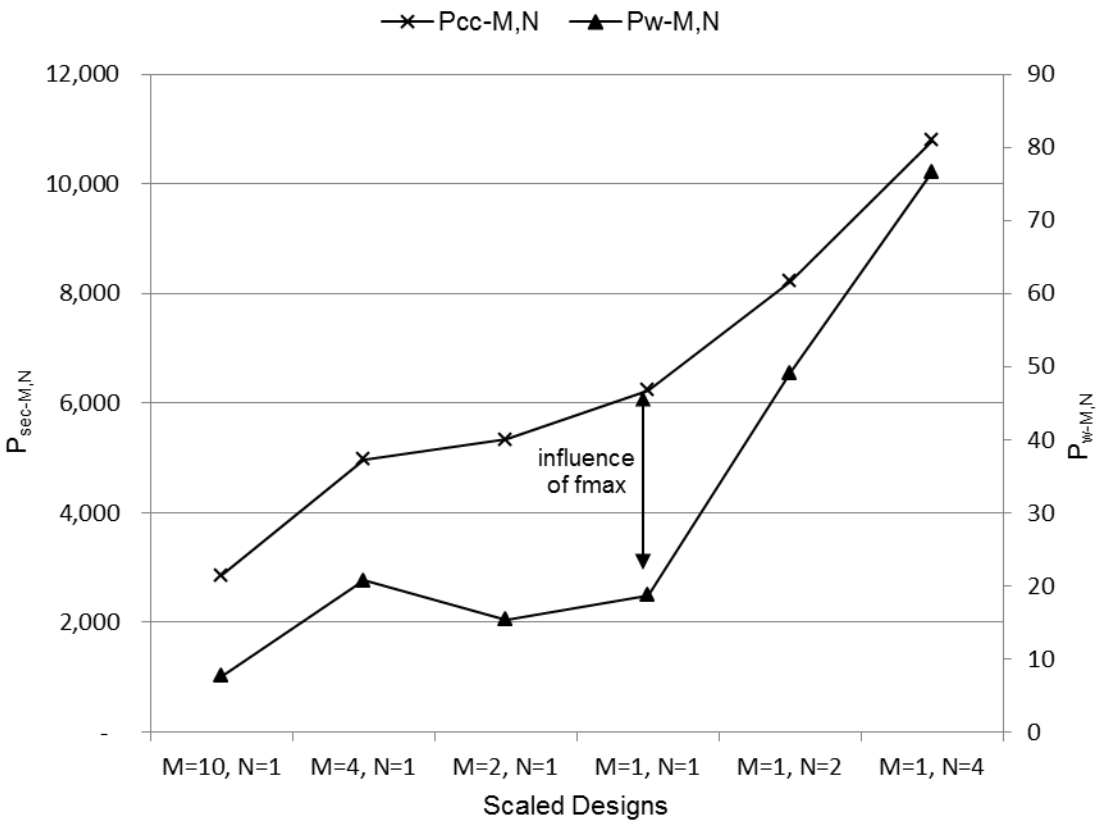


Figure 36 - Effect of the Maximum Achievable Frequency on the Design Scalability

The observed critical paths varied between the six designs. For example, the horizontally scaled designs have wider IEs. The nature of horizontal scaling extends the width of the total number of FIR filters, thus requiring more data sharing between them. For example, the [M=1, N=4] IE has 22 input pixels feeding into 17 H-IPUs, where each pixel, an eight bit vector, fans out to 6 adjacent H-IPUs. These pixel buses cross multiple boundaries of 4x4 subblocks, making routing a challenge at high clock speeds and large designs. It is difficult to place and route circuits with high fan-outs while maintaining closely packed logic. The horizontally scaled designs have a speedup of less the 1 due to the decreased maximum clock frequency. The vertically scaled designs maintain higher clock frequencies since as the design scales their signals do not increase in fan-out across the boundaries of the 4x4 subblocks. Vertical scaling of the IE is based on replication of the base design's H-IPU set, not requiring additional data sharing. Due to the higher clock frequencies and the more efficient use of hardware, the vertically scaled designs achieve an overall increase in performance.

5.3 Target Video Resolution Specifications

Each video standard has a different resolution as well as frame rate, which translates to a particular number of MBs that need to be processed per second. We calculate this by dividing the resolution dimensions by the dimensions of the MB, as is shown in Equation 30.

$$MB_{res} = \frac{width_{frame}}{width_{MB}} \times \frac{height_{frame}}{width_{MB}}$$

Equation 30 - Macroblocks per Second

Further, we take into account the time it takes each scaled design to process the 41 subblocks in reference to encoding a single MB, we can calculate the frames per second, $fps_{M,N}$, for each design, for various video resolutions. We also take into account the number of reference frames used, which in our case is set to 4, but commonly varies from 1 to 16.

$$fps_{M,N} = \frac{f_{\max -M,N}}{T_{w-M,N} \times MB_{res} \times R_{frame}}$$

Equation 31 - Frames per Second

Figure 37 shows the calculated $fps_{M,N}$ for four common video resolutions, within a reasonable range of less than 100 fps. The [M=10, N=1] design can handle the highest frame rates, even for the most demanding QSXGA standard.

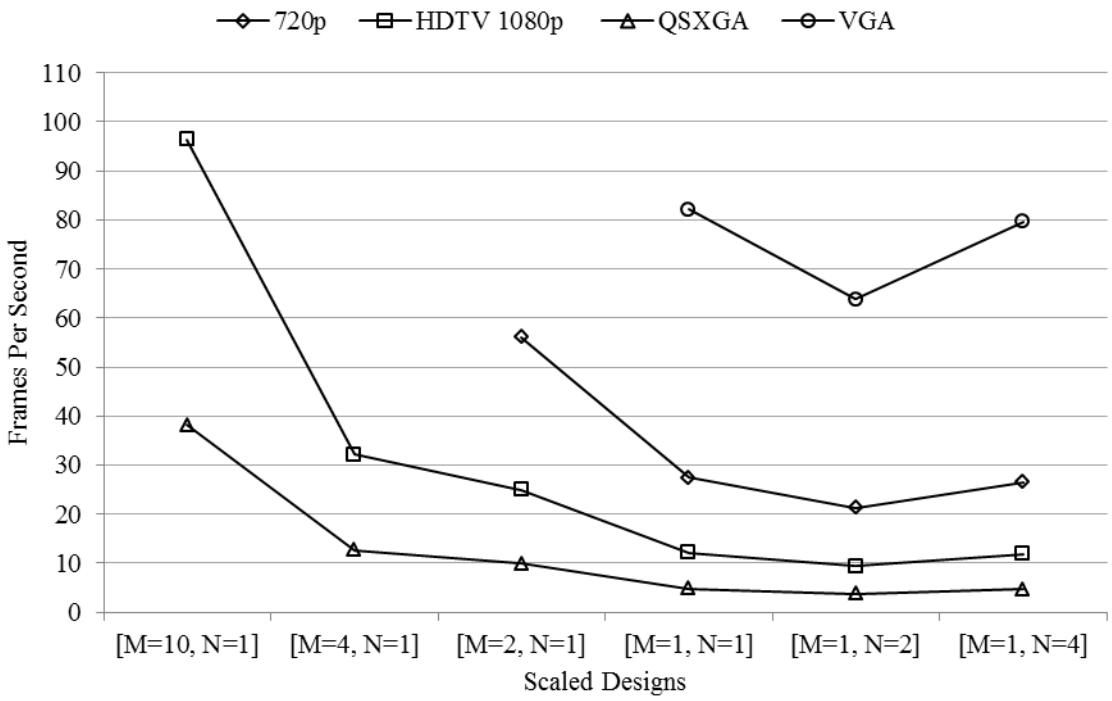


Figure 37 - Frames per Second at Various Video Resolutions

Finally, Table 14 shows the maximum common $fps_{M,N}$ and videos resolution standards suitable for each design.

Table 14 - Target Video Resolution

Type of Scaling	Scaled FME Designs	Target Video Resolution
Vertical	[M=10, N=1]	30fps QSXGA
	[M=4, N=1]	30fps 1080p
	[M=2, N=1]	50fps 720p
Base Design	[M=1, N=1]	60fps VGA
Horizontal	[M=1, N=2]	60fps VGA
	[M=1, N=4]	60fps VGA

5.4 Chapter Summary

In this chapter we present the recorded results from the implementation of all six FME algorithms on a Virtex 5 FPGA. We use these numbers to calculate scalability comparison metrics, such as the cost-performance product, speedup, relative hardware resource cost, and finally relative efficiency. Each of these metrics is meant to give us a further insight into the increase in resource cost and gained processing speed. The cost-performance product showed to be a valuable comparison metric where we saw how much increased performance we gained proportional to the increased cost. Here we identified each of the designs as bad, good or linear scaling. To further understand and quantify the scaling results we went on to define speedup and relative hardware resource cost, from which we derived the relative efficiency. This result was representative of the entire design, as opposed to the approximated efficiency of the IE and PU previously derived.

Chapter 6 – Concluding Summary

This chapter summarizes the presented work; it presents a condensed review of related works, and suggests potential directions for future work.

This work explores the scalability of FME, which is one of the essential components of the H.264/AVC standard. It has been shown that the H.264/AVC compression standard is capable of decreasing transfer stream bit-rates by up to 64% and exhibiting a compression ratio of 50:1. The advanced capability for compression is due to many factors, one of which is FME. Further, it has been shown that FME can improve video quality by up to +4dB.

6.1 Achieved Objectives

The motivation behind this work was to create a scalable and high performance FME engine and to develop an empirical approach by which to examine the scalability of the FME as a parallel system. The thesis defines formal criteria and methodology for evaluating design requirements such as data throughput, data redundancy, target video *fps* and resource costs and utilizations. In addition to examining existing scalable approaches, we developed a novel approach to scaling the FME engine: *vertical* scaling. The implemented designs show that vertical scaling approach can provide equal or better performance when compared with the traditional horizontal scaling approach. After examining the operational characteristics of the scaled designs in detail, we conclude overall the vertical scaled designs exhibit much better cost-performance results as well as overall relative resource efficiencies.

6.2 Comparative Study

Table 15 summarizes relevant and comparable FME implementations. The comparison is narrowed down to works which also implement processing of all 7 variable sized blocks, while performing a full-search of the MV refinement. In comparison to other implementations, we observe that our [M=10, N=1] design uses the greatest number of FIR filters, while a moderately low number of small-sized PUs.

Some works include an IME engine, and as well as a quarter precision FME engine. One of Chen's implementations [13] has four instantiations of the FME engine, one assigned to each of the four reference frames. Similarly, the designs also vary in implementation technology and maximum clock frequency. In our study we normalize the comparison by examining the isolated half-pixel precision unit, independent of the surrounding implementation circumstances.

Table 15 - Comparison to Previous Work

Design	System Overview	Scaling Strategy	Target Architecture	f_{max} *(MHz)*	Input Pixel Size	# of FIRs	# of PUs	# of FME cores	Video Format	*fps*
M=10, N=1 (Proposed)	FME (1/2 pixel)	Vertical Scaling	Virtex-5 FPGA, 25K LUTs	350	10x10	60	9 (4x4)	1	QSXGA	30
Chen [13] M=1, N=1	FME (1/2 pixel)	Base Design	UMC 0.18 µm, 405K gates	81	10x1	16	9 (4x4)	4	720x480	30
Mora-Campos [15] M=1, N=2	IME + FME (1/4 pixel)	Workload Balance	Virtex-4 FPGA, 8234 slices	100	14x1	44	3 x 9 (4x4)	2 sets of IEs & Pus	720p	30
Yang [16] M=1, N=4	FME (1/4 pixel)	Horizontal Scaling	TSMC 0.18µm 188K gates	200	22x1	-*	9 (16x16)	-*	1080p	30
Kao [52]	FME (1/4 pixel)	Workload Balance	TSMC 0.130 µm 311K gates	154	10x1	-*	2 x 9	3 IEs + 2PUs	1080p	30

** - not specified in the paper*

6.3 Future Work

Memory access is a common bottleneck in the H.264/AVC engine. A common design practice is to pipeline the data between the two motion estimation units, FME and IME [15][1]. It would be interesting to explore how suitable each of the scaled FME designs is with various IME blocks and how they fit into the entire H.264/AVC framework. In particular, in terms of the pipelined data flow it would be interesting to see if the data throughput rates of IME and FME can be closely matched and the data buffering between the stages minimized. The proposed FME architectures in this work can be evaluated based on how they perform with respect to memory efficiency.

Secondly, the frequency dip in the horizontally scaled designs can potentially be increased by careful handling and optimizations of the wide fan-outs and data sharing. Register replication can potentially help reduce the strain on the routing.

It would be interesting to examine power consumption of each of the six designs. Two primary sources of power consumption in FPGAs are dynamic power, dissipated due to switching logic, and static power, dissipated due to leaking current in the device. It would be interesting to see how the designs with lower utilization consume power, and weather they exhibit power saving characteristics.

Appendix A - Redundancy Analysis and Block Decomposition

Table 16 - Redundancy Analysis and Block Decomposition Calculations for the [M=1, N=1], [M=2, N=1] and [M=4, N=1] Design

Before Decomposition						After Decomposition				
Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Pixels Per Subblock	Total Pels Per Subblock Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Total Pixels Per Subblock	Total Pixels Per Subblock Per Type
		Width	Height			New Width	New Height			
i	n	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$P_{block\text{-}i}$	$P_{type\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$P_{block\text{-}dec\text{-}i}$	$P_{type\text{-}dec\text{-}i}$
1	1	22	22	484	484	10	22	4	880	880
2	2	14	22	308	616	10	22	2	440	880
3	2	22	14	308	616	10	14	4	560	1120
4	4	14	14	196	784	10	14	2	280	1120
5	8	10	14	140	1120	10	14	1	140	1120
6	8	14	10	140	1120	10	10	2	200	1600
7	16	10	10	100	1600	10	10	1	100	1600
				P_{total}	**6340**				**$P_{total\text{-}dec}$**	**8320**

Table 17 - Redundancy Analysis and Block Decomposition Calculations for the [M=10, N=1] Design

Before Decomposition						After Decomposition				
Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Pixels Per Subblock	Total Pels Per Subblock Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Total Pixels Per Subblock	Total Pixels Per Subblock Per Type
		Width	Height			New Width	New Height			
i	n	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$P_{block\text{-}i}$	$P_{type\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$P_{block\text{-}dec\text{-}i}$	$P_{type\text{-}dec\text{-}i}$
1	1	22	22	484	484	10	10	16	1600	1600
2	2	14	22	308	616	10	10	8	800	1600
3	2	22	14	308	616	10	10	8	800	1600
4	4	14	14	196	784	10	10	4	400	1600
5	8	10	14	140	1120	10	10	2	200	1600
6	8	14	10	140	1120	10	10	2	200	1600
7	16	10	10	100	1600	10	10	1	100	1600
				P_{total}	**6340**				**$P_{total\text{-}dec}$**	**11200**

Table 18 - Redundancy Analysis and Block Decomposition Calculations for the [M=1, N=2] Design

Before Decomposition						After Decomposition				
Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Pixels Per Subblock	Total Pels Per Subblock Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Total Pixels Per Subblock	Total Pixels Per Subblock Per Type
		Width	Height			New Width	New Height			
i	n	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$P_{block\text{-}i}$	$P_{type\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$P_{block\text{-}dec\text{-}i}$	$P_{type\text{-}dec\text{-}i}$
1	1	22	22	484	484	14	22	2	616	616
2	2	14	22	308	616	14	22	1	308	616
3	2	22	14	308	616	14	14	2	392	784
4	4	14	14	196	784	14	14	1	196	784
5	8	10	14	140	1120	10	14	1	140	1120
6	8	14	10	140	1120	14	10	1	140	1120
7	16	10	10	100	1600	10	10	1	100	1600
				P_{total}	**6340**				**$P_{total\text{-}dec}$**	**6640**

Table 19 - Redundancy Analysis and Block Decomposition Calculations for the [M=1, N=4] Design

Before Decomposition						After Decomposition				
Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Pixels Per Subblock	Total Pels Per Subblock Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Total Pixels Per Subblock	Total Pixels Per Subblock Per Type
		Width	Height			New Width	New Height			
i	n	$x_{s\text{-}i}$	$y_{s\text{-}i}$	$P_{block\text{-}i}$	$P_{type\text{-}i}$	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$P_{block\text{-}dec\text{-}i}$	$P_{type\text{-}dec\text{-}i}$
1	1	22	22	484	484	22	22	1	484	484
2	2	14	22	308	616	14	22	1	308	616
3	2	22	14	308	616	22	14	1	308	616
4	4	14	14	196	784	14	14	1	196	784
5	8	10	14	140	1120	20	7	1	140	1120
6	8	14	10	140	1120	14	10	1	140	1120
7	16	10	10	100	1600	20	5	1	100	1600
				P_{total}	**6340**				**$P_{total\text{-}dec}$**	**6340**

Appendix B – Hardware Utilization of the Interpolation Engine

Table 20 – Hardware Utilization of the Interpolation Engine for the [M=10, N=1] Design

Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Clock Cycles per ys-dec-i	Numerator (Eq. 18)	Denominator (Eq. 18)
i	b_i	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$c_{ys\text{-}dec\text{-}i}$		
1	1	10	10	16	1	16.0	16
2	2	10	10	8	1	16.0	16
3	2	10	10	8	1	16.0	16
4	4	10	10	4	1	16.0	16
5	8	10	10	2	1	16.0	16
6	8	10	10	2	1	16.0	16
7	16	10	10	1	1	16.0	16
						sum: 112	sum: 112

M = 10

4*N + 6 = 10

$U_{IE\text{-}M,N}$ = 100%

Table 21 - Hardware Utilization of the Interpolation Engine for the [M=4, N=1] Design

Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Clock Cycles per ys-dec-i	Numerator (Eq. 18)	Denominator (Eq. 18)
i	b_i	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$c_{ys\text{-}dec\text{-}i}$		
1	1	10	22	4	6	3.7	4
2	2	10	22	2	6	3.7	4
3	2	10	14	4	4	7.0	8
4	4	10	14	2	4	7.0	8
5	8	10	14	1	4	7.0	8
6	8	10	10	2	3	13.3	16
7	16	10	10	1	3	13.3	16
						sum: 55	sum: 64

M = 4

4*N + 6 = 10

$U_{IE\text{-}M,N}$ = 86%

Table 22 - Hardware Utilization of the Interpolation Engine for the [M=2, N=1] Design

Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Clock Cycles per ys-dec-i	Numerator (Eq. 18)	Denominator (Eq. 18)
i	b_i	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$c_{ys\text{-}dec\text{-}i}$		
1	1	10	22	4	11	4.0	4
2	2	10	22	2	11	4.0	4
3	2	10	14	4	7	8.0	8
4	4	10	14	2	7	8.0	8
5	8	10	14	1	7	8.0	8
6	8	10	10	2	5	16.0	16
7	16	10	10	1	5	16.0	16
						sum: 64	sum: 64

M = 2

4*N + 6 = 10

$U_{IE\text{-}M,N}$ = 100%

Table 23 - Hardware Utilization of the Interpolation Engine for the [M=1, N=1] Design

Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Clock Cycles per ys-dec-i	Numerator (Eq. 18)	Denominator (Eq. 18)
i	b_i	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$c_{ys\text{-}dec\text{-}i}$		
1	1	10	22	4	22	4	4
2	2	10	22	2	22	4	4
3	2	10	14	4	14	8	8
4	4	10	14	2	14	8	8
5	8	10	14	1	14	8	8
6	8	10	10	2	10	16	16
7	16	10	10	1	10	16	16
						sum: 64	sum: 64

M = 1

4*N + 6 = 10

$U_{IE\text{-}M,N}$ = 100%

Table 24 - Hardware Utilization of the Interpolation Engine for the [M=1, N=2] Design

Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Clock Cycles per ys-dec-i	Numerator (Eq. 18)	Denominator (Eq. 18)
i	b_i	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$c_{ys\text{-}dec\text{-}i}$		
1	1	14	22	2	22	2.0	2
2	2	14	22	1	22	2.0	2
3	2	14	14	2	14	4.0	4
4	4	14	14	1	14	4.0	4
5	8	10	14	1	14	5.7	8
6	8	14	10	1	10	8.0	8
7	16	10	10	1	10	11.4	16
						sum: 37.1	sum: 44

M = 1

4*N + 6 = 14

$U_{IE\text{-}M,N}$ = 84%

Table 25 - Hardware Utilization of the Interpolation Engine for the [M=1, N=4] Design

Subblock Type	Number of Subblocks Per Type	Subblock Dimensions (With Surrounding Pixels)		Number of Vertical Sweeps	Clock Cycles per ys-dec-i	Numerator (Eq. 18)	Denominator (Eq. 18)
i	b_i	$x_{s\text{-}dec\text{-}i}$	$y_{s\text{-}dec\text{-}i}$	s_i	$c_{ys\text{-}dec\text{-}i}$		
1	1	22	22	1	22	1.0	1
2	2	14	22	1	22	1.3	2
3	2	22	14	1	14	2.0	2
4	4	14	14	1	14	2.5	4
5	4*	20	14	1	14	3.6	4
6	8	14	10	1	10	5.1	8
7	8*	20	10	1	10	7.3	8
						sum: 22.8	sum: 29

M = 1

4*N + 6 = 22

$U_{IE\text{-}M,N}$ = 79%

* the number of these subblocks is halved because they are placed into the IE two at a time

Summary of Terms

P_{MB} – number of pixels in a single MB

i – seven various subblock sizes in VBSME

rxy – nine half pixel candidate positions

R_{frame} – number of reference frames; range: [1,16]

$width_{frame}$ – width of a single video frame

$height_{frame}$ – height of a single video frame

fps – frames per second

M – vertical scalable FME variable

N – horizontal scalable FME variable

$R_{M,N}$ – data redundancy for processing 41 subblocks (in pixels)

P_{total} – total number of pixels processed for all 41 subblocks before block decomposition

$P_{total\text{-}dec}$ – total number of pixels processed for all 41 subblock types after block decomposition

$x_{s\text{-}i}$ – width of an entire subblock, with surrounding pixels

$y_{s\text{-}i}$ – height of an entire subblock, with surrounding pixels

$x_{s\text{-}dec\text{-}i}$ – width of a decomposed subblock, with surrounding pixels

$y_{s\text{-}dec\text{-}i}$ – height of a decomposed subblock, with surrounding pixels

$P_{block\text{-}i}$ – number of pixels processed per single subblock i

$P_{type\text{-}i}$ – total number of pixels processed for an entire subblock type i

$P_{block\text{-}dec\text{-}i}$ – number of pixels processed per single subblock after block decomposition

$P_{type\text{-}dec\text{-}i}$ – total number of pixels processed for an entire subblock type *i* after block decomposition

s_i – number of vertical sweeps required for subblock type *i*

b_i – total number of subblocks of type *i*

$U_{PU\text{-}M,N}$ – percentage of hardware utilization of the PU for an FME design M,N

– percentage of hardware utilization of the IE for an FME design M,N

$x_{s\text{-}dec\text{-}i}$ – width of a decomposed subblock, with surrounding pixels

$y_{s\text{-}dec\text{-}i}$ – height of a decomposed subblock, with surrounding pixels

$c_{ys\text{-}dec\text{-}i}$ – number of clock cycles required to process a single sweep of $y_{s\text{-}dec}$ pixels

$x_{e\text{-}i}$ – width of an entire subblock, elementary pixels only

$y_{e\text{-}i}$ – height of an entire subblock, elementary pixels only

$T_{cc\text{-}M,N}$ – total number of clock cycles required to process 41 subblocks

$T_{w\text{-}M,N}$ – total wall clock time required to process 41 subblocks

$LUT_{M,N}$ – measured hardware cost in terms of FPGA LUTs

$REG_{M,N}$ – measured hardware cost in terms of FPGA registers

$f_{max\text{-}M,N}$ – maximum achievable clock frequency

$MB_{M,N}$ – data throughput in terms of MBs

$P_{w\text{-}M,N}$ – cost-performance product of based on the wall clock time it takes to process 41 subblocks

$P_{cc\text{-}M,N}$ – cost-performance product of based on the number of clock cycles it

takes to process 41 subblocks

P_{SCALED} – cost-performance product of a scaled design

P_{BASE} – cost-performance product of a base design

$C_{M,N}$ – increase in cost with respect to the base design

$S_{M,N}$ – speedup with respect to the base design

$U_{FME\text{-}M,N}$ – hardware utilization of the entire FME, relative to the base design

MB_{res} – the number of MBs in a single frame of a particular video resolution

Glossary

ME	– Motion Estimation
FME	– Fractional Motion Estimation
IME	– Integer Motion Estimation
PSNR	– Peak Signal to Noise Ratio
MSE	– Mean Squared Error
FPGA	– Field Programmable Gate Array
ASIC	– Application Specific Integrated Circuit
MV	– Motion Vector
VBSME	– Variable Block Size Motion Estimation
MRF	– Multiple Reference Frame
H.264/AVC	– video compression standard
AVC	– Advance Video Coding
MB	– macroBlock
subblock	– a block of pixels smaller then the MB
pel	– pixel
SATD	– Sum of Absolute Transformed Differences
HD	– High Definition
SDTV	– Standard Definition Television
HDTV	– High Definition Television

codec	– video compression engine
IE	– Interpolation Engine
PU	– Processing Unit
V-IPU`	– Vertical Interpolation Units
H-IPU	– Horizontal Interpolation Units
FIR	– Finite Impulse Response
FIFO	– First In First Out
cc.	– clock cycles

References

[1] T. Chen, S. Chien, Y. Huang, C. Tsai, C. Chen, T. Chen, and L. Chen, "*Analysis and Architecture Design of an HDTV720p 30 Frames/s H.264/AVC Encoder*," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 16, No. 6, June 2006, pp. 673-688.

[2] Iain, E. G. R.,: *H.264 and MPEG-4 Video Compression: Video Coding for Next generation Multimedia*. Wiley (2003)

[3] T. Chen, Y. Huang, and L. Chen, "*Fully Utilized and Reusable Architecture for Fractional Motion Estimation for H.264/AVC*," in Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, Canada, May 2004, pp. 9-12.

[4] Xilinx soft-core DS602, http://www.xilinx.com/support/documentation/ip_documentation/h264_cabac_prodbrief_ds602.pdf; last accessed 2010

[5] http://www.4i2i.com/; last accessed 2010

[6] Corrêa, M.M.; Schoenknecht, M.T.; Agostini, L.V, "*A High Performance Hardware Architecture for the H.264/AVC Halfpixel Motion Estimation Refinement*," 23rd Symposium on Integrated Circuits and Systems Design (SBCCI), 2010.

[7] T. Dias, N. Roma, and L. Sousa, "*Fully Parameterizable VLSI Architecture for Sub-Pixel Motion Estimation with Low Memory Bandwidth Requirements*," November 2005.

[8] N. Roma and L. Sousa, "*Parameterizable Hardware Architectures for Automatic Synthesis of Motion Estimation Processors*," in Proceedings of the IEEE Workshop on Signal Processing Systems - Design and Implementation (SiPS'01), Antwerpen - Belgium, Sept. 2001, pp. 428-439.

[9] L. De Vos and M. Stegherr, "*Parameterizable VLSI Architectures for the Full-Search Block-Matching Algorithm*", IEEE Trans. Circuits Syst., vol. 36 , pp.1309 - 1316 , 1989.

[10] T. Chen, Y. Huang, and L. Chen, "*Analysis and Design of Macroblock Pipelining for H.264/AVC VLSI Architecture*," in Proc. of IEEE International Symposium on Circuits and Systems, Vancouver, Canada, May 2004, pp. 273-276.

[11] T. Chen, C. Lian, and L. Chen, "*Hardware Architecture Design of an H.264/AVC Video Codec*," in Proc. of Asia and South Pacific Design Automation Conference, Yokohama, Japan, January 2006, pp. 750-757.

[12] Y. Chen, T. Chuang, C. Tsai, Y. Chen, and L. Chen, "*A Cost-Efficient Residual Prediction VLSI Architecture for H.264/AVC Scalable Extension*," in Proc. of Picture Coding Symposium, Lisboa, Portugal, November 2007, pp. 13-17.

[13] Y. Chen, T. Chen, S. Chien, Y. Huang, and L. Chen, "*VLSI Architecture Design of Fractional Motion Estimation for H.264/AVC*," Journal of Signal Processing Systems, Vol. 53, No. 3, December 2008, pp. 335-347.

[14] M. Shao, Z. Liu, S. Goto, T. Ikenaga, "*Lossless VLSI Oriented Full Computation Reusing Algorithm for H.264/AVC Fractional Motion Estimation*," IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences, Vol. E90-A, No. 4, April 2007, pp. 756-763.

[15] A. Maro-Campos, F. Ballester-Merelo, M. Martinez-Peiro, and J. Canals-Esteve, "*High Parallel-Pipeline Integer-Pel and Fractional-Pel Motion Estimation VLSI Architectures for H.264/AVC*," in Proc. of SPIE: VLSI Circuits and Systems III, May 2007, Maspalomas, Spain, pp. 659010-1--659010-11.

[16] C. Yang, S. Goto, and T. Ikenaga, "*High Performance VLSI Architecture of Fractional Motion Estimation in H.264 for HDTV*," in Proc. IEEE International Symposium on Circuits and Systems, Island of Kos, Greece, May 2006, pp. 2605-2608.

[17] S. Oktem, I. Hamzaoglu, "*An Efficient Hardware Architecture for Quarter-Pixel Accurate H.264 Motion Estimation*", in Proc of the 10th Euromicro Conference onDigital System Design Architectures, Methods and Tools, 2007, pp. 444-447

[18] F. Urban , R. Poullaouec , J.-F. Nezan and O. Deforges, "*H.264 Fractional Motion Estimation Refinement: A Real-Time and Low Complexity Hardware Solution for HD Sequences*", Proc. 15th EUSIPCO, pp. 836 2007.

[19] Urban F., Poulaouec R., Nezan J-F., Déforges O., "*A Flexible Heterogeneous Hardware/Software Solution for Real-Time High-Definition H.264 Motion Estimation*," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 18, Num. 12, Pp. 1781-1785, 2008.

[20] W.N. Chen, H.M. Hang, "*H.264/AVC Motion Estimation Implementation on Compute Unified Device Architecture (CUDA)*", Multimedia and Expo, 2008 IEEE International Conference on, pp.697-700, June 23 2008-April 26 2008

[21] Yeo, H., & Hu, Y.H. (1995). "*A Novel Modular Systolic Array Architecture for Full-Search Block Matching Motion Estimation*", IEEE Transactions on Circuits and Systems for Video Technology, 5(5), 407-416.

[22] M.-Y. Hsu, H.-C. Chang, Y.-C. Wang, L.-G. Chen, "*Scalable Module-Based Architecture for MPEG-4 BMA Motion Estimation*", 2000, Dept. Elect. Eng., National Taiwan Univ.

[23] Corrêa, M.M., Schoenknecht, M.T., Dornelles, R.S., Agostini, L.V., "*A High Performance Hardware Architecture for the H.264/AVC Half-Pixel Interpolation Unit*", Programmable Logic Conference , March 2010, pg 81 - 86.

[24] T. Wang, Y. Huang, H. Fang, and L. Chen, "*Parallel 4x4 2D Transform and Inverse Transform Architecture for MPEG-4 AVC/H.264*," in Proc. of IEEE International Symposium on Circuits and systems, May 2003, pp. 800-803.

[25] Sayed, Mohammed S. Badawy, Wael Jullien, Graham "*Interpolation-Free Fractional-Pixel Motion Estimation Algorithms with Efficient Hardware Implementation*", Journal of Signal Processing Systems, 2010

[26] S. Oktem, I. Hamzaoglu, "*An Efficient Hardware Architecture for Quarter-Pixel Accurate H.264 Motion Estimation*", in Proc of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2007, pp. 444-447

[27] K.Jack, "*A Handbook for the Digital Engineer: Video Demystified*", LLH Technology Publishing, Eagle Rock, VA, 2001

[28] Cliff Wootton, "*A Practical Guide to Video and Audio Compression From Sprockets and Rasters to Macro Blocks*", Elsevier Inc., 2003.

[29] Peter Kuhn, "*Algorithms, Cimplexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*", Kluwer Academic Publishers, 1999.

[30] LSI Logic Inc. , "*H.264/MPEG-4 AVC Video Compression Tutorial*", 2003

[31] T. Wiegand, G. Sullivan, G. Bjontegaard, A. Luthra, "*Overview of the H.264/AVC Video Coding Standard*," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, No. 7, July 2003, pp. 560-576.

[32] Joint Video Team Reference Software JM7.3, http://iphome.hhi.de/suehring/tml/download/; last accessed 2009

[33] Iprof ftp server. [Online]. Available: ttp://iphome.hhi.de/suehring/tml/download; last accessed 2009

[34] W. Chao, T. Chen, Y. Chang, C. Hsu, and L. Chen, "*Computationally Controllable Integer, Half, and Quarter-Pel Motion Estimation for MPEG-4 Advanced Simple Profile*" , in Proc. Of the International Symposium on Circuits and Systems, Bangkok, Thailand, May 2003, pp. II788-II791.

[35] J. Suh and J. Jeong, "*Fast Sub-pixel Motion Estimation Techniques Having Lower Computational Complexity*", IEEE Transactions on Consumer Electronics, Vol. 50, No. 3, August 2004, pp. 968-973.

[36] Z. Chen, J. Xu, Y. He, and J. Zheng, "*Fast Integer-Pel and Fractional-Pel Motion Estimation for H.264/AVC,*" Journal of Visual Communication and Image Representation, Vol. 17, No. 2, April 2006, pp. 264–290.

[37] L. Yang, K. Yu, J. Li, and S. Li, "*Prediction-Based Directional Fractional Pixel Motion Estimation for H.264 Video Coding,*" in Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Philadelphia, PA, March 2005, pp. II901–II904.

[38] L. Zhang, W. Gao, "*Improved FFSBM Algorithm and Its VLSI architecture for Variable Block Size Motion Estimation of H.264,*" in Proc. of the International Symposium on Intelligent Signal Processing and Communication Systems, Hong Kong, December 2005, pp. 445-448.

[39] J. Chang and J. Leou, "*A Quadratic Prediction Based Fractional-Pixel Motion Estimation Algorithm for H.264,*" in Proc. of the IEEE International Symposium on Multimedia, Irvine, CA, December 2005, pp. 491-498.

[40] Y. Wang, C. Cheng, and T. Chang, "*A Fast Algorithm and Its VLSI Architecture for Fractional Motion Estimation for H.264/MPEG-4 AVC Video Coding,*" IEEE Transactions on Circuit and Systems for Video Technology, Vol. 17, No. 5, May 2007, pp. 578-583.

[41] J. Jung, G. Jin, and H. Lee, "*Early Termination and Pipelining for Hardware Implementation of Fast H.264 Intraprediction Targeting Mobile HD Applications,*" EURASIP Journal on Advances in Signal processing, Vol.2008, pp. 1-19.

[42] Z. Liu, S. Goto, and T. Ikenaga, "*Content-Aware Fast Motion Estimation for H.264/AVC,*" IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences, Vol.E91-A, No.8, August 2008, pp. 1944-1952.

[43] Y. Song, Y. Ma, Z. Liu, T. Ikenaga, and S. Goto, "*Hardware-Oriented Direction-Based Fast Fractional Motion Estimation Algorithm in H.264/AVC,*" in Proc. of the IEEE International Conference on Multimedia and Expo, Hannover, Germany, June 2008, pp. 1009-1012.

[44] UG190 - Virtex-5 FPGA User Guide, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf; last accessed 2010

[45] http://www.symphonyeda.com/products.htm; last accessed 2010

[46] www.xilinx.com; last accessed 2010

[47] DS100 – Virtex-5 Family Overview, http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf; last accessed 2010

[48] Virtex-5 FPGA XtremeDSP Design Considerations http://www.xilinx.com/support/documentation/user_guides/ug193.pdf; last accessed 2010

[49] Virtex 6 FPGA DSP48E1 Slice http://www.xilinx.com/support/documentation/user_guides/ug369.pdf; last accessed 2010

[50] Sparten-6 FPGA DSP48A1 Slice http://www.xilinx.com/support/documentation/user_guides/ug389.pdf; last accessed 2010

[51] XST User Guide http://www.xilinx.com/itp/xilinx9/books/docs/xst/xst.pdf; last accessed 2010

[52] C.-Y. Kao, C.-L. Wu, Y.-L. Lin, "*A High-Performance Three Engine Architecture for H.264/AVC Fractional Motion Estimation*", IEEE Trans. Very Large Scale Integr. Syst. 18 (2010) 662–666.

[53] Kane Computing Ltd. Application Note http://www.kanecomputing.co.uk/pdfs/compression_ratio_rules_of_thumb.pdf; last accessed 2010

[54] J. Vasiljevic, A. Ye, "*A Scalability Study of Fractional Motion Estimation for H.264 Encoding*", in Proc. of the IEEE 18th Canadian Conference on Electrical and Computer Engineering, pp. 1–5.

[55] J.Vasiljevic, A. Ye, "*Analysis and Architecture Design of Scalable Fractional Motion Estimation for H.264 Encoding*", Integration, the VLSI Journal, 2011