Insights to Variable Block Size Motion Estimation by Design Space Exploration

Jani Boutellier
Machine Vision Group
University of Oulu
90014 Oulun yliopisto, Finland
Email: bow@ee.oulu.fi

Philip Brisk and Paolo Ienne
Processor Architecture Laboratory
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
Email: philip.brisk@epfl.ch, paolo.ienne@epfl.ch

Abstract—Numerous different implementations for H.264/AVC variable block size motion estimation have been proposed in the recent years to make this computationally challenging task more feasible for mobile devices with video encoding support. The variable block size motion estimation problem defined by the standard is complex and multidimensional, offering a wide variety of possibilities for efficient implementation. One of the most popular implementation architectures are systolic arrays.

In this paper we look at the full-search variable block size motion estimation problem on 1D systolic arrays from a high level by modeling the system with a software tool that enables design space exploration and cycle-accurate simulation. Our design space exploration tool has provided many interesting insights to the VBSME problem that give directions for making efficient designs.

I. INTRODUCTION

Motion estimation is a fundamental part of modern block-based hybrid video coders, such as in MPEG-4 video [1]. In block-based motion estimation a small part (template) of the present video frame is matched against a search area in a neighboring video frame in the hope that corresponding content can be found.

In the new h.264/AVC video coding standard the motion estimation is done by so-called variable block size motion estimation (VBSME), which is one of the most computationally demanding tasks of the encoding effort. In the last few years, numerous implementations have been proposed to make the computational burden lighter [2], [3]. These approaches can be divided into two main categories: approximative and full-search solutions.

An approximative algorithm for VBSME uses only a fraction of the available search space to find the best match for the template image data. This means that the globally best match cannot always be found, but there will be a sub-optimal solution with a considerably smaller computational effort. As an example, the algorithm described in [2] offers a 75% reduction in computational complexity with an image quality loss of only about 0.5dB (PSNR), which is visually hard to detect.

However, there are some occasions when the sub-optimal solution is not acceptable. In this case the motion estimation has to be done by a full-search method that searches all of the area within the limits or the search radius. Although the number of computations increases dramatically when compared to approximative solutions, there are fortunately a few features in the full-search approach that can make it computationally feasible. First of all, there is a considerable amount of redundancy in the computations. When utilized correctly, this means that the amount of data moved around can be reduced greatly. Second, the full search space makes the problem very regular, which promotes the use of simple and fast processing elements (PEs). Finally, the full-search solutions can be parallelized in many different ways, which opens further possibilities to reduce the latency of the algorithm.

Because of the aforementioned properties, the H.264/AVC VBSME problem has often been implemented in systolic array hardware. A systolic array architecture consists of a set of processing elements organized by a highly regular interconnection network (e.g., a mesh). During every clock cycle, each PE performs some computation and exchanges data with its immediate neighbors. The PE is typically controlled by a finite state machine (FSM) and the data flow is synchronous and regular. Both 1-dimensional (1D) systolic arrays and 2-dimensional (2D) solutions are used for VBSME. 1D systolic array architectures are typically used for portable electronics, where a premium is placed on power consumption and battery lifetime, rather than performance; 2D architectures are typically used for high-end systems, where a premium is placed on performance and power consumption is not a primary concern.

In this paper we describe a software tool for modeling 1D systolic array architectures that can be used for VSBME. The software tool has been written in C++ with the help of a general-purpose systolic array library. The software model can be used for design space exploration of VBSME solutions and it has given some insight on building efficient 1D systolic array architectures for VBSME.

In Section II we describe the general-purpose systolic array library that was used as a basis for our design space exploration tool. Section III describes the design space exploration tool in detail and Section IV describes the insights provided by our work.

II. SYSTOLIC ARRAY DEVELOPMENT LIBRARY

Our design space exploration tool was built on a systolic array library that is written in C++ in an object-oriented
manner. As a basis in the library are several primitive classes that are either instantiated directly by the designer, or used as base classes for custom design. The class hierarchy is depicted in Figure 1.

The library provides the primitive building blocks for a systolic array architecture, as well as the means to connect them. In addition to that, also convenient functions have been designed that help setting up meshes of PEs by a single function call. A feature worth mentioning is also the possibility of address auto-increment in memory blocks, which enables removing this mundane task from the actual processing loop.

Our library does not automatically map applications to a systolic array, but assumes that the designer has enough knowledge to do that. This also means that the synchronization of data streams from different inputs must be figured out by the designer.

The parent of all systolic array classes is `SynchronousObject` that provides a common clock to all objects that have a concrete counterpart in the system. `ContainerObject` is a virtual class that is derived from `SynchronousObject`. It is the parent class of all classes that can be used to store data. The final virtual class is `PE`, which contains the mandatory functionalities common to all kinds of processing elements. Our library does not offer any directly usable processing element classes, instead it is assumed that the user of the library derives a new class from `PE` and uses that for the application. After deriving a custom PE class, the user must program its actual behaviour in C++, which essentially means defining functionality between the input and output channels of the PE. The functionality between the input and outputs can be arbitrary, as long as it provides valid data to the outputs at each clock cycle and respects the width of the channel. Below is the class declaration of `PE`:

```cpp
class PE : public ContainerObject {
public:
  PE();
  ~PE();
  virtual void Process(int cycle) = 0;
  void ReadChannels(int cycle);
  void WriteChannels(int cycle);
  int AddInput(CommBuffer* source);
  int AddOutput(CommBuffer* sink);
};
```

Fig. 1. The systolic array primitive classes.

MemoryBlock is a class that is derived from `ContainerObject` and it is used to represent memories that can be written or read by processing elements.

`Commbuffer` is a stand-alone class without parents that is used to represent a communication channel between classes that are derived from `ContainerObject`. `Commbuffer` is capable of storing the incoming and outgoing data, as well as checking that the data does not exceed the channel width or that one channel is not written twice within one clock cycle. Instances of the `Commbuffer` class are not aware of the inputs and outputs that are connected to it. That information is found in the `ContainerObjects` attached to it.

Array is a class that can contain processing elements and communication channels, and serves as a wrapper that repre-
TABLE I
Usage example of our library.

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- mysg = new SynchronousGroup();</td>
<td>Derive a custom processing element class myPE from PE</td>
</tr>
<tr>
<td>- myarr = mysg.CreateArray();</td>
<td>Instantiate a new SystolicArray object</td>
</tr>
<tr>
<td>- myarr.CreatePEs(myPE)(1,8);</td>
<td>Instantiate 1 times 8 processing elements of type myPE to myarr</td>
</tr>
<tr>
<td>- myarr.CreateChannels(0,8);</td>
<td>Connect PEs by 8-bit channels in direction 0 (left-to-right)</td>
</tr>
<tr>
<td>- inmem = mysg.CreateMemoryBlock(size);</td>
<td>Instantiate new MemoryBlock</td>
</tr>
<tr>
<td>- inmem.AddOutput(arr.GetPE(0,0).GetInput(0));</td>
<td>Connect dangling leftmost channel of the array to inmem</td>
</tr>
<tr>
<td>- inmem.SetOutputStepping(0,1);</td>
<td>Set memory address auto-increment to 1</td>
</tr>
<tr>
<td>- outmem = mysg.CreateMemoryBlock(size);</td>
<td>Instantiate new MemoryBlock</td>
</tr>
<tr>
<td>- outmem.AddInput(arr.GetPE(0,7).GetOutput(0));</td>
<td>Connect dangling rightmost channel of the array to outmem</td>
</tr>
<tr>
<td>- outmem.SetInputStepping(0,1);</td>
<td>Set memory address auto-increment to 1</td>
</tr>
<tr>
<td>for(i = 0; i &lt; size; i++) mysg.Process();</td>
<td>Do cycle-by-cycle processing through the input data</td>
</tr>
</tbody>
</table>

III. THE DESIGN-SPACE EXPLORATION TOOL

Our design-space exploration tool was initially built as a high-level model that expresses two different, previously built 1D systolic array architectures as different design points of one general model. The two architectures that span the design space were: a part of the work done by Yap and McCanny [3] and another 1D design that is still unpublished work [4]. Before explaining our Design Space Exploration (DSE) tool in detail, we define the VBSME problem and present the two designs [3], [4] that motivated our work.

A. VBSME Concepts

In the h.264/AVC VBSME algorithm the 16x16 pixel template image is compared to the 31x31 pixel reference image at all 256 positions that the 16 pixel search range allows (See Figure 2). The comparison is done by computing a sum of absolute differences (SAD) between the overlapping pixels of the template and reference image at each specific position.

The 16x16 pixel template image is divided into 4x4 subblocks, which are combined into variable size blocks in seven different ways (depicted in Figure 3). As the SAD results are computed for each of the 7 block combinations that contain various numbers of blocks, the result of this scheme is 41 SAD (16+8+8+4+2+2+1) results for each search position, giving a total of 10496 SAD results. The VBSME search algorithm must finally return the position that gave the best SAD result, for each of the 41 partial SADs.

In addition to these standard concepts, we define an additional concept called cluster. A cluster is a rectangular shape of 16 pixels and it is used to describe the order in which the template image pixels are processed. Clusters come in 5 different shapes, as depicted in Figure 7. Possible cluster shapes include 1x16, 2x8, 4x4, 8x2 and 16x1. As each template image consists of 16 clusters, also the processing order of clusters must be agreed upon. For example, the traditional raster scan [3] can be implemented with the cluster shape 16x1.

For every array architecture presented in this paper, the input data feed was organized in the same way. The organization is named broadcasting reference frame data [5] and is depicted in Figure 4.
B. Implementation of the Architecture of Yap and McCanny

The architecture designed by Yap and McCanny [3] is depicted in Figure 5. It has a complex interconnection named the SAD Bus Network that connects two sets of processing elements. Most of the computations are done in the first set of PEs (labeled PE in Figure 5), whereas the second set of processing elements practically only serves as a set of comparators (labeled Min in Figure 5). As stated before, we used a part of the solution of Yap and McCanny as one initial design point while creating our DSE tool. The part that we used, was the array of processing elements labeled PE in Figure 5.

The PE data input and output schedules are not defined explicitly for the DSE tool. Instead, the DSE tool automatically computes a SAD production schedule after the array parameters have been set. For the array configuration that matches the work of Yap and McCanny, the resulting DSE PE schedules match exactly the ones defined in the paper [3].

The systolic array design of [3] was not followed outside the array named "PE" (see Figure 5), e.g., the SAD bus network is not reproduced by the DSE tool. Instead, the comparator functionality was implemented in the same fashion as in the other baseline work [4]. This is explained in the next subsection.

C. Improved 1D Systolic Array

The second design point that gave directions in creating our DSE tool was an internally developed architecture that dramatically simplifies the 1D systolic array design when compared to the work described in [3]. This design takes the comparator functionality that resides in a separate set of processing elements in [3], and moves them inside the primary set of PEs (shown in Figure 5). This eliminates the need of the complex SAD bus network.

This solution differs from the design of [3] also by an alternative organization of processing elements, which is depicted in Figure 6 under the name "4 PE pipeline". Moreover, the input order of the template image data (See Fig. 4) has been changed. We shall see in Section IV, how these changes affect the design. This architecture was designed on register transfer level before the work described in this paper was done.

D. Design Space Exploration Tool

As the two previous subsections already suggest, this design space exploration tool is actually a parameterized implementation of our internally developed VBSME systolic array [4] that can also be transformed into the design described in Subsection III-B, and numerous other designs by simple parameter changes.

The DSE tool allows changing the array configuration (Fig. 6) and template image cluster shape (Fig. 7), but the throughput remains fixed. These two parameters affect the number of registers in the system, as will be shown later. In addition to these two parameters, the cluster processing order can also be varied freely in the exploration tool, but we used only one fixed cluster order for each cluster shape.

Changing the design parameters does not change the number of computations that are done inside a PE: each PE computes a single pixel difference on each clock cycle. However, there is a difference in the rate that SAD results for sub-blocks and larger blocks are produced.

Based on the variables mentioned above, the DSE tool computes a cycle-accurate schedule for all the SAD computations of the block matching and allocates register space for results that have to be stored temporarily. At this point it is worthwhile to mention that our tool correctly reproduces the schedules of our internally developed VBSME systolic array [4] that can also be transformed into the design described in Subsection III-B, and numerous other designs by simple parameter changes.

To be specific: the results are available in 4096 + L clock cycles, where L is the pipeline length: 1...16.
IV. INSIGHTS PROVIDED BY THE DSE TOOL

Figure 8 shows the design space covered by our DSE tool. The vertical axis depicts the array configurations that are shown in Figure 6 and the horizontal axis depicts the cluster shapes (See Fig. 7). The letter A in the design space figure shows the design space position of the solution that was described in Subsection III-B. Respectively, the letter B shows where the solution of Subsection III-C lies in the search space. The other 23 design space points represent new solutions that have been brought up by this DSE tool. As we will later see, the DSE results point out that some of these designs might even be better than [4], [3], whereas others are most certainly worse.

The key result that the DSE tool produces is the schedule for SAD computations that tells at which clock cycles new results emerge and have to be processed by the comparator hardware. Evaluation of the SAD schedules brings up the first insight provided by this work: it is desirable that the SAD results are produced as a smooth stream instead of periodic bursts. If the SAD results occur as sudden bursts, the PEs need to contain more register space to save the results before sending them for further processing. According to the DSE output, the cluster shape is the variable that affects this issue most, as can be seen in Table II. The cluster shape 4x4, which has also been used in [4], provides the smallest SAD production variance and thus, register space requirement. This makes sense intuitively, since 4x4 is also the shape of sub-blocks in H.264/AVC. Again, this speaks against the use of the traditional raster scan that has been used, e.g., in [3] and [5].

The length of the systolic pipeline (See Fig. 6) is related to the reuse of template image data. With the pipeline of length 1 (array shape 1x16), the template image pixels are never reused inside the array, but instead are read 16 times more often from the template image memory than with pipeline length 16 (array shape 16x1). This means that the memory bandwidth increases as the pipeline grows shorter: with pipeline length 1 the system requires 16 new template image pixels (= 128 bits) each clock cycle, whereas the configuration with pipeline length 16 requires only one template image pixel value (8 bits) per clock cycle.

When we assume that we would create a design that uses the cluster shape 4x4, we see from Table II that the number of registers per PE remains generally around 6 with all pipeline lengths. The reason behind this is that during the VBSME computations, it happens for every PE at some point that 6 SAD results are ready at the same clock cycle. Therefore, each of the PEs must have this minimum number of temporary register space. This leads to our second insight: since the register space requirement per PE remains roughly the same for all pipeline lengths, this speaks strongly against long pipelines. This issue is shown in Table III that shows the total number of registers used by each configuration. In addition to the temporary register space, we have also added the necessary inter-PE communication registers to the register count here.
It is assumed that there is an 8-bit pipeline register for each parallel channel between PEs, and that temporary SAD results that are stored internally are 16 bits.

Related to the SAD register space, one might wonder if it would be possible to share the SAD registers between different PEs. With clever design and a small number of PEs, this could truly work, but it is evident that for longer pipelines this will raise complex interconnection issues such as the SAD Bus network in [3].

Another fact worth mentioning is that the cluster shape and processing order variables also affect the template image memory access patterns. The most simple memory access pattern is achieved with the cluster shape 16x1, which implies only a simple memory address incrementation each clock cycle. Other cluster shapes require more complex addressing, as Figure 7 shows. However, this particular issue can be compensated by the way how the template image is written to the memory – it is possible to write the pixels to the memory so that simple address incrementation can be applied to every cluster shape.

In a nutshell, the results indicate that longer pipeline lengths result in large numbers of registers (as well as other PE components) that are under-utilized. This speaks in favor of short 1...4 PE pipelines. Naturally, since our design space exploration tool does not provide a HW synthesis backend, it is not certain that a system like "Full Parallel" in Figure 6 would be feasible. Also, it is clear that the cluster shape (i.e. template image pixel processing order), should be set to 4x4, since it requires the least register space.

The DSE tool shows that there are potentially better array configurations than the ones that were known prior to building the DSE tool. It was known beforehand from our internally developed 1D array that the cluster shape 4x4 was better than the commonly used 16x1 raster scan. The design space exploration automated the non-trivial problem of determining schedules for the SAD calculations, which is strongly related to the register count required by the architecture.

As the systolic array library does not yet provide a possibility for hardware synthesis, it is not certain what kind of undiscovered drawbacks these newly found array configurations might have. This is a clear direction for future work.

V. RELATED WORK

Systolic arrays were popularized in the research community by Kung and Leiserson [6] in 1978. Historically, systolic arrays have been used for many different applications. Classic examples of algorithms that could easily parallelize on systolic arrays include matrix multiplication and evaluation of polynomial algorithms using Horner’s Rule.

In the domain of video coding, systolic arrays have also been used elsewhere than in motion estimation: Chiper [7] has published a systolic array implementation of the inverse discrete cosine transform and Li et al. [8] have published a systolic array implementation of 2-dimensional interpolation.

A considerable amount of work has been done in the development of the ALPHA language [9], which enables the automatic synthesis of systolic algorithms. ALPHA follows an equational approach, where a program is expressed as a collection of single assignment equations. This is a major difference when compared to our approach, as well as the fact that ALPHA does not convey implicitly the notion of time.

A work close to ours [10] has been done as a part of the Aries project. This technical memo describes a C++ framework name Sim that is intended for cycle-based simulations. Compared to our work, Sim employs a lower level of abstraction.

Saraswat et al. have performed design space exploration [11] on the same VBSME problem as presented in this paper. They used a tool called DesertFD and their work considered a much wider problem that involved choosing one of two different sub-optimal search patterns and also synthesizing two solutions to actual hardware. Their paper also contains a good survey of recent DSE papers related to VBSME.

VI. DISCUSSION AND FUTURE WORK

We have presented a variable block size motion estimation design space exploration tool for 1D systolic arrays. It is based on a general-purpose systolic array library that has been written in C++. The DSE tool correctly reproduces two already existing designs and uncovers a variety of new designs, some of which indicate the possibility of improved performance when compared to the existing ones.

For future work it would be interesting to build a back-end to the systolic array design library that would enable automatic hardware synthesis of the systems that have been designed with the library. Also, the design space exploration tool could be extended to encompass various throughputs and search radiiuses.

REFERENCES


