Efficient distribution of video frames to achieve better throughput

Bhavik Vyas
Reliance Consulting
bhavik@gmail.com

Suruchi Jain
Marseille Networks
Suruchi.Jain@marseillenetworks.com

Kiran Maiya
Synopsys Inc.
Kiran.Maiya@synopsys.com

***Abstract*—Verification engineers are constantly looking to reap more simulation cycles with limited available resources. There are several methods to achieve this goal. One such method is to design the verification environment with multiple instances of identical design - frame driver pairs. A traffic flow controller picks the data packets or video frames from the image library and drives through these multi-channel drivers. The frames are reordered before being driven through these multiple channels. This reordering of data packets alone can give a better throughput.**

This paper also proposes an algorithm involved in reordering of the video frames before they are driven. This algorithm groups the frames in an intermediate repository, so as to have an equal simulation cycles in each group. This algorithm for deciding the optimal distribution of video frames was implemented using SystemVerilog constraints. Based on the solution generated by the constraint solver, a SystemVerilog ® based testbench was used to distribute and transmit the frames among available video channels. The results were promising and saw increased throughput.

Keywords-component;

HDMI, VCS®,SystemVerilog®, Constraints

A. *Abbreviations and Acronyms*

DUT – Device Under Test

HDMI – High Definition Multimedia Interface

SVTB – System Verilog Test Bench

VIP – Verification Intellectual Property

II. INTRODUCTION

With growing demand for a short time to market, engineers are bracing for solutions to run as many simulations that are possible in a short time span utilizing the available resource at its maximum efficiency. Typically, verifying graphics chips involves running multiple simulations while programming different aspects of the design. The DUT will be programmed either through register settings or using knobs. Frames having varying lengths and types are generated by the driver and are applied to the DUT under different programming modes. Once the processed image is available at the output of the DUT, the receiver recreates the image for further process depending on the verification goal. This driver, DUT and receiver combination is pretty much self contained, which can be instantiated multiple times to create multiple parallel channels that can process several frames simultaneously.

III. METHODOLOGY

A. *Typical setup*

Figure-1 shows a typical setup used by verification engineers for verifying graphics chip using Verification IPs. This setup shows different methods to generate frames.

The frames can be either a randomly generated image or image picked from library of images that typically fits the profile of the DUT being verified. Verification IPs like Synopsys HDMI VIP [2], can generate images having pixel values determined by constraints applied on them and also drive as per HDMI standard. These constraints are part of the testbench and can generate different frames by just changing the seed at runtime, without having to change the testbench. For testing the functionality of the design in the real world scenario, it is necessary to drive the DUT with actual images. A simple image to frame converter block can take the existing image tailored to test the functionality of the DUT, and split it into packets that will be transmitted using the VIP driver.

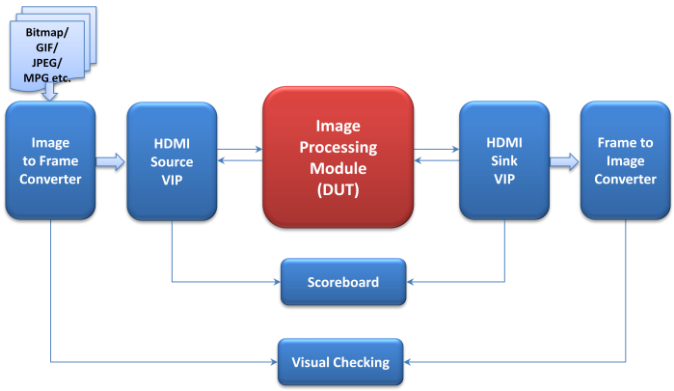

Figure 1. Typical setup for graphics device using VIPs

Depending on the functionality of the DUT, images undergo transformation or scaled (up-scaling or down-scaling) as it passes through the DUT. When this processed image arrive at the receiver, it will be forwarded for further operations, which may include displaying these images for a visual comparison with the source image or automatically checking the pixels for correctness by passing it to a scoreboard.

With the ever increasing image sizes, simulations involving video frames can consume significant amount of CPU cycles. Submitting simulation process to a large farm is the widely adopted methodology to speed up the simulation process. However there are several other ways one can adopt to achieve the same, with significantly lesser CPU cycles. One such method is to have a single simulation environment with multiple design instances inside it.

B. Methodology for parallel simulation

With designs having larger memory foot prints, simulation loading time can be significant. This amounts to significant boot up time when several simulations are to be run with different frame formats and DUT register settings. Under such circumstances, it is efficient to architect the verification environment as multiple copies of identical set of transmitter, DUT and receiver blocks creating one large image. Each of these copies acts as a standalone image processing pipe or channel. The entire image will be loaded only once in the beginning of the simulation and inside this image, multiple frames are processed in parallel.

This multiple identical design instance allows parallelism at runtime which can be subjected to various types of optimizations by simulators. With the advent of new technologies in simulators, it is more efficient to run two tests in one image rather than running two individual tests in two different simulations. Simulators like VCS® [1] can optimize large binaries by collapsing the duplicate logic in an efficient manner. Automatic module inlining allows grouping of identical logic and collapsing them to one group resulting in optimized simulation binary that is faster in simulation. Features such as Design Level Partitioning (DLP) in VCS®, allows partitioning of the design and create simulation threads that runs on multiple cores. These threads run in parallel across multiple cores leading to faster simulation.

Figure-2 shows a setup having multiple instances running in parallel. A master scheduler will retrieve the frames from frame database and feeds into multiple pipes as they become ready for processing the frames. Monitors at the output will recreate the frames and passes to comparators for checking for correctness.

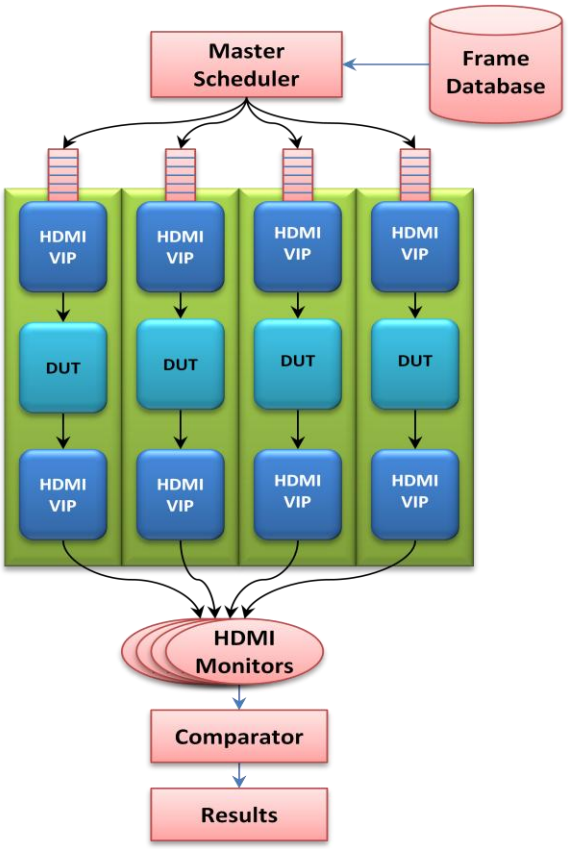

Figure 2. Setup for running multiple instances simultaneously

Main drawback of this methodology is that the overall turnaround time for a given set of frames. With each randomization, the frame order can vary and are fed in different order for processing. This change in order can cause some pipes to finish early than the others making the overall turnaround time unpredictable.

C. Reordering frames with uniform distribution

One of the major disadvantages in the parallel simulation setup described above is the under utilization of the design-stimulus channel causing inefficiency of resources. This under utilization is caused because of the random distribution of frames of varying length across multiple channels. This unequal distribution of frames can affect the overall turnaround time (TAT) for the simulation, especially when long running simulations are submitted at the end. In such cases the TAT is dictated by the longest running frame that runs till the end.

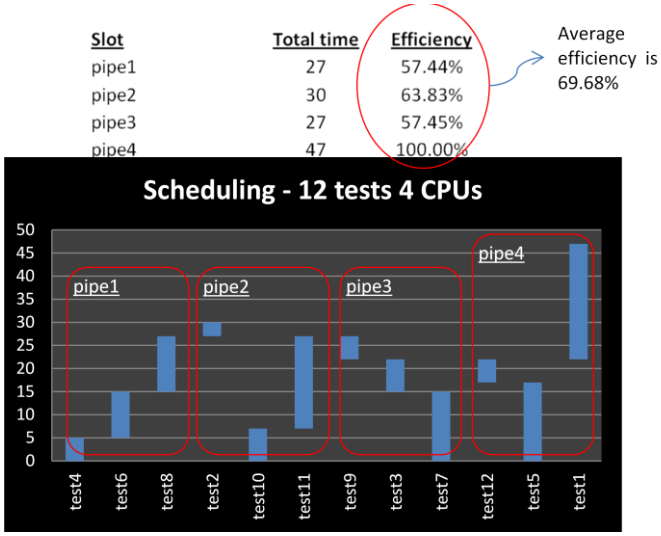

Figure 3. Turnaround time without distribution

Figure-3 shows a scenario where a long running simulation holds up one of the design instance reducing the efficiency of the other pipes. In this scenario pipe-4 runs for 45 time units, whereas the remaining pipes have completed their share of the frames and stay idle. Measuring the efficiency of the individual design instance shows that the distribution of efficiency is unequal among pipes. Except for the pipe-4 instance, that had the longest frame executing till the end, rest of the design instances were idle for significant amount of time. This reduced their efficiency significantly. Efficiency is determined as the ratio of simulation cycles executed by the design instance to the overall simulation cycles (i.e. longest running design instance).

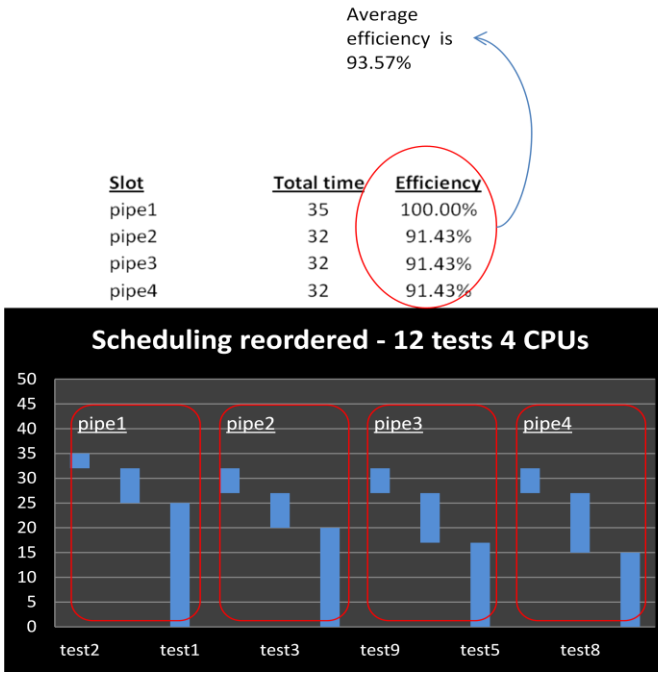

Figure 4. Turnaround time with equal distribution

Figure-4 shows how the distribution by rearranging the same set of images. Once the same frames are rearranged with long running tests equally distributed among each design instance, the simulation in each instance completes around the same time. This keeps all of the design instance fully occupied, resulting in higher efficiency. As seen in figure-4, the simulation is completed in 35 time units compared to 47 time units without redistribution. This redistribution of frames alone gives around 30% speed up.

For effective verification, every feature of the design needs its own set of images that should be used. Say, to verify the up scaling logic of a design, it makes sense to run simulations involving smaller images like 640x480 or 720x480 that gets up scaled, rather than picking larger images like 1920x1080p. When verification engineer wants to verify different features, image library needs to be changed accordingly. Under such circumstances, one cannot guarantee fixed order of frames that gives efficient distribution every time. There is a need for automation to determine efficient frame order, when the images in the library are constantly changing.

IV. Algorithm for Optimal Distribution

This section of the paper discusses the algorithm developed to achieve the automation. Native simulator's constraint solver was used to achieve this automation of finding the best possible distribution of frames. The logic to determine the distribution was written using SystemVerilog [3] constraints which are solved by the SystemVerilog simulator. The results obtained from the solver were used to determine the distribution.

Logic used here is a simple and straight forward one. The frame lengths of all the images were read into a *holder array*. Since frame length of each image is directly proportional to the simulation cycles that are needed to process that particular image, sum of this holder array should essentially give the rough estimate of total simulation time, if all the frames from *holder array* were to be run in serial in one pipe. In the SystemVerilog code, an array per simulation pipe is created and the frames from the *holder array* are distributed among these arrays. Constraints are applied such that the sum of frame lengths in each of these individual arrays are either equal or fall under certain threshold between each other. The individual frame buffer inside the master scheduler is then filled with these frames from corresponding array. This ensures that all frame buffers inside master scheduler are filled with equally distributed frames.

Following code snippets shows the logic used in distributing the frames equally among the available four arrays. For better readability of the code, a simplified version of the actual constraint code is shown here. In this code snippet, integral frame lengths are used as each item instead of the frame itself.

Following code snippet in Figure-5 shows the class structure containing the individual array that holds the frames (note that the frame lengths are used instead of actual frames) to be driven on each the pipe.

```
class Pipe ;
  rand int arr[] ; // Array holding frames lengths
  rand int sz ; // Variable to hold the array size
  rand int sm ; // Variable to hold the array sum

 constraint basic
   {
    sz == arr.size() ;
    sm == arr.sum() ;
    sz < 100;  // Pipe holds less than 100 frames
   }
 endclass: Pipe
```

Figure 5. Class structure showing individual buffers

A top level structure is created as shown in the code snippet in Figure-6, where deck of individual buffer classes are instantiated and constrained it to be as wide as the number of pipes in the design. Basic constraints are applied such that the individual arrays are no wider than the *holder array*. Here constraints are applied such that the sum of frame lengths inside each individual buffers are either equal or falls under certain threshold.

```
class Structure ;
  rand Pipe deck[] ;

  rand int wide ;  // Decides pipe count

  int hldr[];//Array to hold complete set of frames
  int hldr[];//Temporary array
  rand int ind[] ; //Array to hold indices
  rand int threshold ; // Threshold for individual
array sum variation

 constraint basic
   {
    deck.size == wide ; //no. of pipes
    hldr.size() == deck.sum with (item.sz) ;
    foreach (deck[i])
    {
     deck[i].arr.size < hldr.size() ;
    deck[i].arr.size > 0 ;
   } // deck[i]

   foreach (deck[i])
   {
    deck[i].sm >=(hldr.sum()/wide)-threshold ;
    deck[i].sm <=(hldr.sum()/wide)+threshold ;
   } // deck[i]
} // basic
```

Figure 6. Class structure to hold deck of individual buffer classes

The logic adopted to assign the frames from *holder array* into individual buffers, is by creating a temporary holder array, *t_arr*, whose contents are shuffled values from *holder array*. Figure 7 shows the code snippet to shuffle the *holder* array contents and assign it to the temporary array. There are several ways to shuffle an array, here is one such method where the indices of the array are maintained in an indices array. *ind[]*, and used to shuffle the contents.

```
constraint con_shuffle
{
 foreach(ind[i])
 {
  foreach(t_arr[j])
  {
   if(j == ind[i])
   t_arr[i] == hldr[j] ;
  } // t_arr[j]

 foreach (ind[j])
 {
 (i!=j) -> (ind[i] != ind[j]) ;
 }// ind[j]
 ind[i] inside {[0:ind.size() -1]} ;
 ind[i] >= 0 ;
} // ind[i]
} // con_shuffle
```

Figure 7. Constraints showing shuffling of *holder array* contents

The contents of this shuffled array are assigned to each individual buffers as shown in code snippet in figure-8.

```
constraint con_assign
{
 t_arr.size() == hldr.size() ;
     ind.size() == hldr.size() ;

     foreach (hldr[k])
     {
      foreach(hldr[j])
      {
       foreach(hldr[i])
       {
         if((i%wide) == k )
         {
          if (j == (i- (i%wide))/wide)
          {
           deck[k].arr[j] == t_arr[i] ;
          }
         } // if
        } // foreach hldr[i]
       } // foreach hldr[j]
      } // foreach hldr[k]
}
```

Figure 8. Constraints assigning *holder array* contents to individual buffers.

V. RESULTS

Experiments were run with multiple frames of varying size and numbers. In each of these experiments totally 12 frames were driven through the device. Each experiment had two sets of simulations runs. First, frames were generated randomly and applied to the pipes as they were available for simulation. This data is plotted as *Turnaround time without distribution* in Table 1. Second, frames were redistributed and applied to the pipes so as to get equal distribution as discussed in this paper. This data is plotted under *Turnaround with distribution* column in Table-1.

TABLE I. TURNAROUND TIME WITH AND WITHOUT TEST DISTRIBUTION

Expt. Number	Image Format	Number of Frames	Time (in normalized units)	
			Turnaround without distribution	*Turnaround with distribution*
1	640x480p	2	582	535
	1280x720p	4		
	1920x1080i	6		
2	640x480p	3	508	471
	1280x720p	5		
	1920x1080i	4		

In the first experiment there was 8% performance gain in total turnaround time by redistributing the frames for this given combination. In this experiment the overall utilization of the pipes were increased by 8%. Without the redistribution the pipes were occupied 86% of the time, whereas after the distribution they were occupied 94% on average.

In the second experiment there was more than 7% gain in total turnaround time with a similar redistribution. Here the pipe utilization was increased by 6%.

There can be a significant variation in the gain after distribution, depending on the frame length of the images that are used. In this section of the paper, a set of frames with modest gain were selected to plot these results. The actual time in secs were normalized while tabulating these results.

VI. CONCLUSION

Synopsys VCS simulator was used to simulate the design along with Synopsys-HDMI-VIP as the frame generator. Some of the key technologies available in the VCS simulator were used to further enhance the throughput of the simulation.

From the experiment it was concluded that having multiple instances of the design in parallel that are fed with evenly distributed frames gives a better throughput. Having an automated way of finding the equal distribution gives more flexibility in terms of picking random images from library and still able to benefit from this methodology.

As a future enhancement, we would like to plug the distribution algorithm into the design environment making the decision making process seamless. Currently the distribution algorithm is standalone and is not part of the overall flow. As per Amdahl's law, there is a limitation in terms of number of parallel channels that can be used before there is degradation in the overall performance. The optimal number for this setup needs to be measured.

VII. ACKNOWLEDGMENT

We would like to thank the engineering team from both Synopsys [4] and Marseille Networks [5] for helping in answering our questions and providing guidance in places where we needed help.

VIII. REFERENCES

Following materials and websites were referred during the course of this work.

[1] VCS Users guide
[2] HDMI VIP users guide
[3] SystemVerilog Language Reference Manual
[4] http://www.synopsys.com
[5] http://www.marseillenetworks.com/