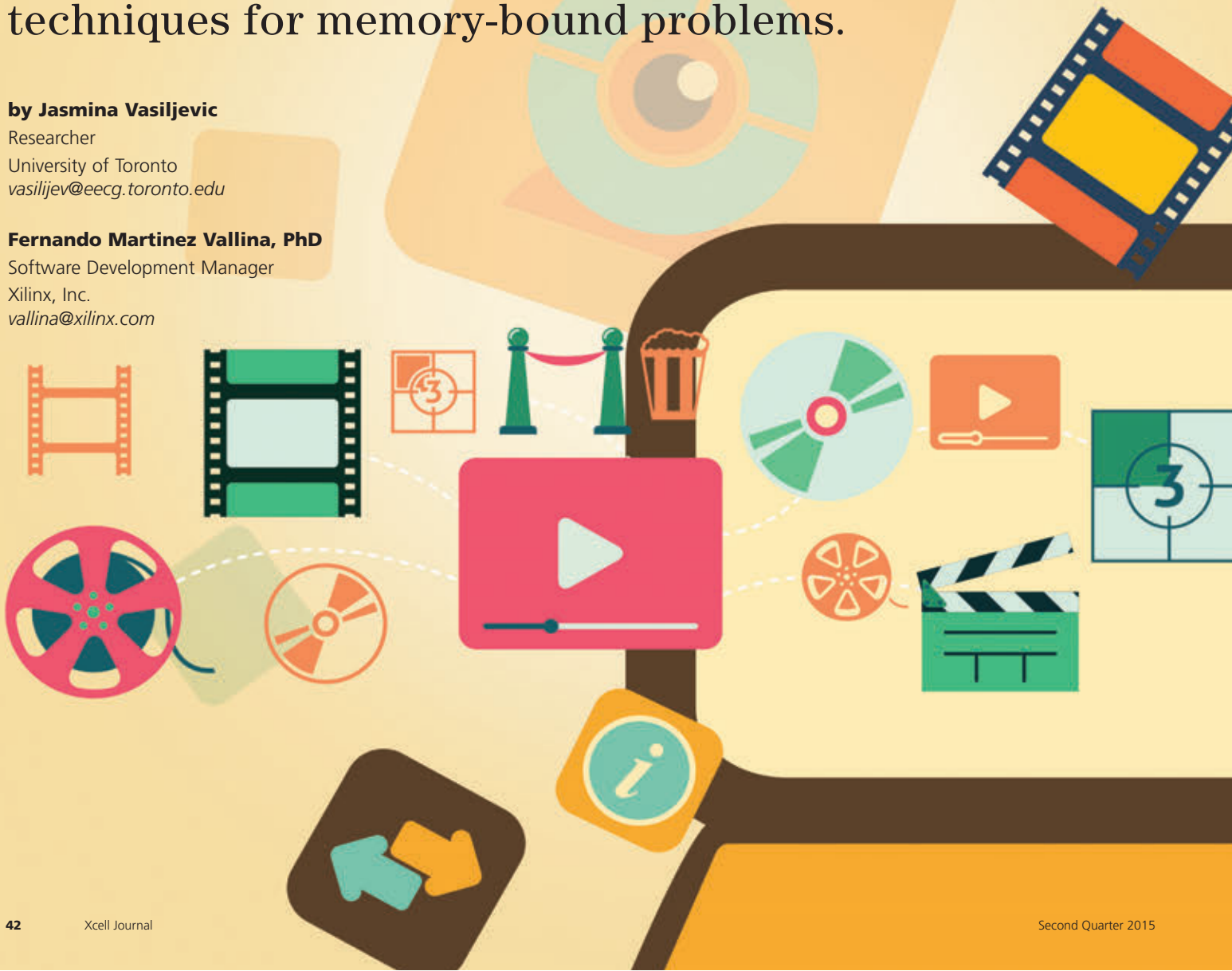
Optimizing an OpenCL Application for Video Watermarking in FPGAs

Xilinx's SDAccel development environment offers optimization techniques for memory-bound problems.

by Jasmina Vasiljevic
Researcher
University of Toronto
vasilijev@eecg.toronto.edu

Fernando Martinez Vallina, PhD
Software Development Manager
Xilinx, Inc.
vallina@xilinx.com



Video streaming and downloading account for the majority of consumer Internet traffic and are a driving force behind cloud computing. The continually growing demand for this type of content is pushing video-processing applications out of specialized systems and into the data center. This shift in the deployment paradigm allows for the rapid scaling of computation nodes to accommodate the needs of the different compute-intensive stages of video content preparation and distribution, such as transcoding and watermarking.

We recently used Xilinx®'s SDAccel™ development environment to compile and optimize a video-watermarking application written in OpenCL™ for an FPGA accelerator card. Video content providers use watermarking to brand and protect their content. Our goal was to design a watermarking application that would process high-definition (HD) video at a 1080p resolution with a target throughput of 30 frames per second (fps) running on an Alpha Data ADM-PCIE-7V3 card.

The SDAccel development environment enables designers to take applications captured in OpenCL and compile them to an FPGA without requiring knowledge of the underlying FPGA implementation tools. The video-watermarking application serves as a perfect way to introduce the main optimization techniques available in SDAccel.

VIDEO WATERMARKING WITH LOGO INSERTION

The main function of the video-watermarking algorithm is to overlay a logo at a specific location on a video stream. The logo used for the watermark can be either active or passive. An active logo is typically represented by a short, repeating video clip, while a passive logo is a still image.

The most common technique among broadcasting companies that brand their video streams is to use a company logo as a passive watermark, so that was the aim of our example design. The application inserts a passive logo on a pixel-by-pixel level of granularity based on the operations of the following equation:

```
out_y[x][y]  = (255-mask[x][y]) * in_y[x][y]  + mask[x][y] * logo_y[x][y]
out_cr[x][y] = (255-mask[x][y]) * in_cr[x][y] + mask[x][y] * logo_cr[x][y]
out_cb[x][y] = (255-mask[x][y]) * in_cb[x][y] + mask[x][y] * logo_cb[x][y]
```

The input and output frames are two-dimensional arrays in which pixels are expressed using the YCbCr color space. In this color space, each pixel is represented in three components: Y is the luma component, Cb is the chroma blue-difference component and Cr is the chroma red-difference component. Each component is represented by an 8-bit value, resulting in a total of 24 bits per pixel.

The logo is a two-dimensional image containing the content to be inserted. The mask is also an image, but it contains only the contour of the logo. The pixels in the mask are either white or black. White pixels in the mask indicate the logo insertion location, while black pixels indicate that the original pixel remains untouched. Figure 1 shows an example of the operation of the video-watermarking algorithm.

TARGET SYSTEM AND INITIAL IMPLEMENTATION

The system on which we executed the application is shown in Figure 2. It is composed of an Alpha Data ADM-PCIE-7V3 card communicating with an x86 processor over a PCIe® link. In this system, the host processor retrieves the input video stream from disk and transfers it to the device global memory. The device global memory is the memory on the FPGA card that is directly accessible from the FPGA. In addition to placing the video frames in device global memory, the logo and mask are transferred from the host to the accelerator card and placed in on-chip memory to take advantage of the low latency of BRAM memories. Since this application uses a passive logo, only the data for the still image and placement location needs to be stored in the on-chip memories.

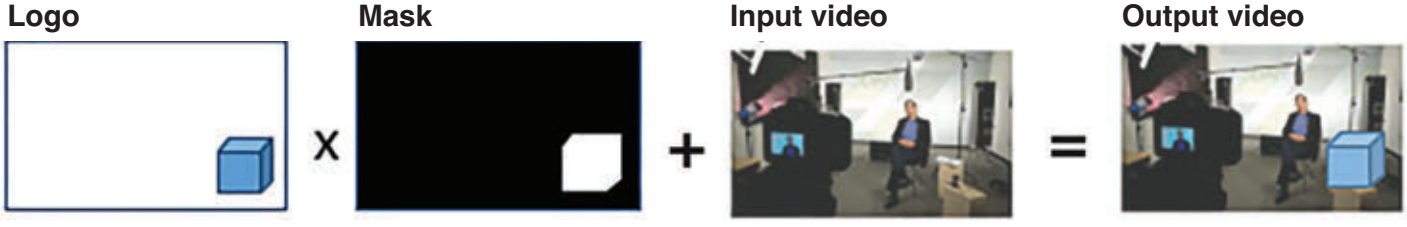


Figure 1 – The video-watermarking algorithm in action

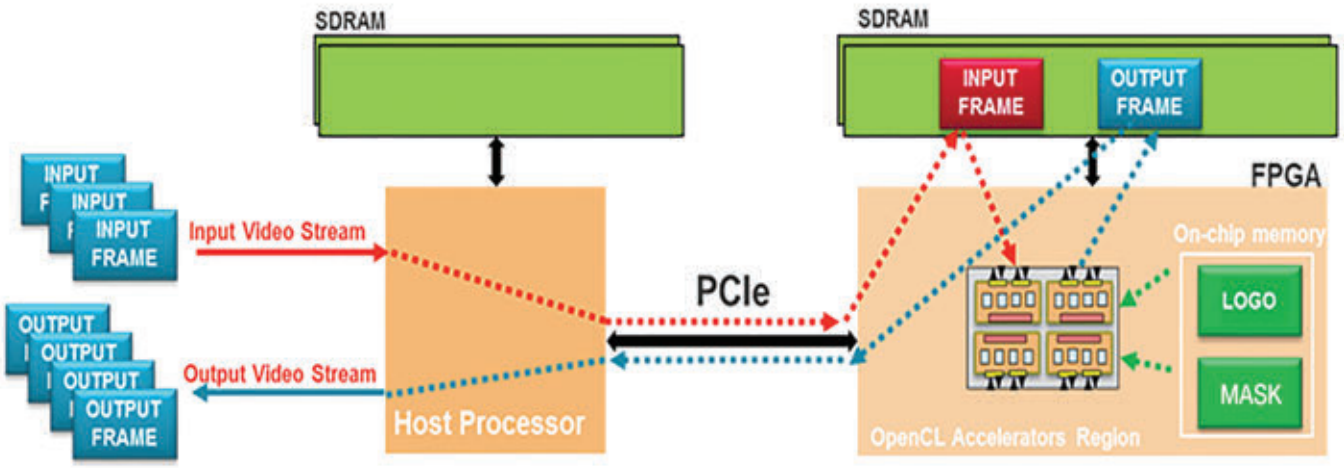


Figure 2 – System overview for the video-watermarking application

Once data has been set up, the host processor sends a start signal to the watermarking kernel in the FPGA fabric. This signal triggers the kernel to do three things: start fetching input video frames from device global memory, insert the logo at the location defined by the mask and place the processed frames back in device global memory to be fetched by the processor.

The coordination of data transfer and computation for every frame in the video stream is achieved by means of the code in Figure 3.

This code, which runs on the host processor, is responsible for sending a video frame to the FPGA accelerator card, launching the accelerator and then retrieving the processed frame from the FPGA accelerator card.

The first implementation of the watermarking algorithm for the FPGA is shown in Figure 4. This is a functionally correct implementation of the application but does not have any performance optimization or consideration for the capabilities of the FPGA fabric. As is, this code compiles in SDAccel and can be run on the Alpha Data card for a maximum throughput of 0.5 fps.

As you can see in the code of Figure 4, the watermarking algorithm is not a compute-intensive design. Most of the time is spent accessing memory to read and write video frames. Therefore, we focused on memory bandwidth when optimizing our example design.

OPTIMIZING MEMORY ACCESS USING VECTORIZATION

One of the advantages of the FPGA fabric over other software-programmable fabrics is the flexibility and configuration of interconnect buses to memory. SDAccel creates custom-size datapaths and architectures to memory based on the application kernels. A higher memory bandwidth from the kernel can be inferred by modifying the code to consume multiple pixels at a time, a procedure referred to as vectorization.

The level of vectorization that is appropriate depends on the application and the FPGA accelerator card being used. In the case of the Alpha Data card, the interface to device global memory has a width of 512 bits, which matches the maximum AXI interconnect width available to a kernel in SDAccel. Given this boundary of 512 bits, the application is modified to process 20 pixels at a time (24 bits/pixel x 20 pixels = 504 bits). SDAccel offers full support for vector data types. Therefore, for this application, vectorizing the code is as simple as changing the data type of all arrays to char20, as shown in Figure 5, which results in a throughput of 12 fps.

OPTIMIZING MEMORY ACCESS USING BURSTS

Although vectorization significantly improves the performance of the application, it was not enough to reach the goal of 30 fps. The application remains memory bound, because the kernel is issuing memory transfers of only 20 pixels at a time. To reduce the impact of memory constraints on the application, we had to modify the kernel code to generate burst read/write operations to memory for a data set larger than 20 pixels. The modified kernel code is shown in Figure 6.

The first modification to the kernel code is to define on-chip storage in the kernel to store a block of pixels at a time. The on-chip memories are defined by arrays declared in the kernel code. To state a burst transaction to memory, the code instantiates a memcpy command to move a block of data from DDR to BRAM storage inside of the kernel. Based on the size of on-chip memory resources and the amount of data to

be processed, a video frame is divided into 20 blocks of 1920 x 54 pixels, as shown in Figure 7.

Once the memcpy operations have placed the data for a block into the kernel arrays, the algorithm executes the watermarking algorithm on the block of data and places the results back into kernel arrays. The results of the block processing are transferred back to DDR memory using memcpy operations. This sequence of operations repeats for 20 times until all blocks in a given frame have been processed. As a result of this modification to the kernel code, the system performance is 38 fps, which exceeds the original goal of 30 fps.

Host Code

SEND FRAME / COMPUTE FRAME / RECEIVE FRAME

```
for (i=0; i<FRAMES; i++) {
  // Send a video frame to the FPGA device
  err = clEnqueueWriteBuffer(commands, d_frin, CL_TRUE, 0,
        sizeof(int) * LENGTH_FRAN, h_frin, 0, NULL, &writeEvent0;
  clWaitForEvents(1, &writeEvent);

  // Run logo insertion on the input frame
  err = clEnqueueTask(commands, kernel_load_block, 0, NULL,
        &kernelEvent);
  clWaitForEvents(1, &kernelEVent);

  // Read the output frame back to CPU
  err = clEnqueueReadBuffer( commands, d_frout, CL_TRUE, 0,
        sizeof(int) * LENGTH_FROUT, h_frout, 0, NULL, &readEvent);
  clWritForEvents(1, &readEvent);
}
```

Figure 3 – Code to coordinate each frame's data transfer and computation

```
for (y=0; y<FRAMES_HEIGHT; y++) {
  for (x=0; x<FRAMES_WIDTH/20; x++) {

        i = y*FRAME_WIDTH/20 + x;
        out_y[i]   = (255-mask[i]) * in_y[i]   + mask[i] * logo_y[i];
        out_cr[i]  = (255-mask[i]) * in_cr[i]  + mask[i] * logo_cr[i];
        out_cb[i]  = (255-mask[i]) * in_cb[i]  + mask[i] * logo_cb[i];
  }
}
```

Figure 4 – Initial implementation of the watermarking kernel

```
for (y=0; y<FRAMES_HEIGHT; y++) {
  for (x=0; x<FRAMES_WIDTH/20; x++) {

        i = y*FRAME_WIDTH/20 + x;

        out_y[i]   = (255-mask[i]) * in_y[i]   + mask[i] * logo_y[i];
        out_cr[i]  = (255-mask[i]) * in_cr[i]  + mask[i] * logo_cr[i];
        out_cb[i]  = (255-mask[i]) * in_cb[i]  + mask[i] * logo_cb[i];
  }
}
```

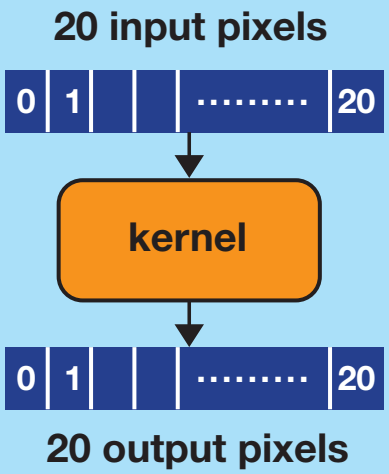


Figure 5 – Kernel code after vectorization

```
char20 l_in_y[BLOCK_WIDHT*BLOCK_HEIGHT/20];
char20 l_in_cr[BLOCK_WIDHT*BLOCK_HEIGHT/20];
char20 l_in_cb[BLOCK_WIDHT*BLOCK_HEIGHT/20];

char20 l_out_y[BLOCK_WIDHT*BLOCK_HEIGHT/20];
char20 l_out_cr[BLOCK_WIDHT*BLOCK_HEIGHT/20];
char20 l_out_cb[BLOCK_WIDHT*BLOCK_HEIGHT/20];

for (block_id=0; block_id<FRAME_HEIGHT/BLOCK_HEIGHT; block_id++){

  memcpy(l_in_y, in_y + block_id*BLOCK_WIDHT*BLOCK_HEIGHT/20, BLOCK_WIDHT*BLOCK_HEIGHT*3);
  memcpy(l_in_cr, in_cr + block_id*BLOCK_WIDHT*BLOCK_HEIGHT/20, BLOCK_WIDHT*BLOCK_HEIGHT*3);
  memcpy(l_in_cb, in_cb + block_id*BLOCK_WIDHT*BLOCK_HEIGHT/20, BLOCK_WIDHT*BLOCK_HEIGHT*3);

  for (y=0; y<BLOCK_HEIGHT; y++) {
    for (x=0; x<BLOCK_WIDTH/20; x++) {

      i = y*BLOCK_WIDTH/20 + x;

      l_out_y[i]  = (255-mask[i]) * l_in_y[i]  + mask[i] * logo_y[i];
      l_out_cr[i] = (255-mask[i]) * l_in_cr[i] + mask[i] * logo_cr[i];
      l_out_cb[i] = (255-mask[i]) * l_in_cb[i] + mask[i] * logo_cb[i];
  }

  memcpy(out_y, l_out_y + block_id*BLOCK_WIDHT*BLOCK_HEIGHT/20, BLOCK_WIDHT*BLOCK_HEIGHT*3);
  memcpy(out_cr, l_out_cr + block_id*BLOCK_WIDHT*BLOCK_HEIGHT/20, BLOCK_WIDHT*BLOCK_HEIGHT*3);
  memcpy(out_cb, l_out_cb + block_id*BLOCK_WIDHT*BLOCK_HEIGHT/20, BLOCK_WIDHT*BLOCK_HEIGHT*3);
}
```

On-chip memory storage for the input block

On-chip memory storage for the ioutput block

Loop through 20 blocks

PHASE 1

PHASE 2

PHASE 3

Figure 6 – Kernel code optimized for burst data transfers

BROADLY APPLICABLE

The optimizations necessary when creating applications like this one using SDAccel are software optimizations. Thus, these optimizations are similar to the ones required to extract performance from other processing fabrics, such as GPUs. As a result of using SDAccel, the details of getting the PCIe link to work, drivers, IP placement and interconnect became a non-issue, allowing us as designers to focus solely on the target application.

The optimizations we made in our watermarking application are applicable to all designs compiled using SDAccel. In fact, video watermarking provides a great "how to" introduction to the optimization methods Xilinx has made available in SDAccel.

Figure 7 – Video frame partitioning into blocks