Color Space Conversion

Application Note

Version 1.4

Confidential & Proprietary

Last modified: 12/06/2005

This preliminary publication is provided "AS IS." Stretch, Inc. (hereafter "Stretch") DOES NOT MAKE ANY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF TITLE, NONINFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Information in this document is provided solely to enable system and software developers to use Stretch S5000 processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder. Stretch does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Part #: AN-0000-0001-04

Color Space Conversion – An Optimization Example

One of the major benefits of using the Stretch Software Configurable Processor (SCP) is that it provides the ability to dramatically accelerate a C or C++ application using straightforward C and C++-based optimizations. Here we present the main development steps for optimizing any application to run on the SCP. These steps are:

1. Begin with a C or C++ implementation of the application.
2. Identify the key portions of the application that will benefit from acceleration.
3. Write custom instructions to accelerate the key portions, and modify the relevant code to take advantage of them.
4. Adjust the custom instructions to work on data in parallel.
5. Eliminate memory latency issues by using the SCP's high-performance memories.

This application note illustrates how to follow these steps using a color space conversion application as a model.

Color Space Conversion Application

The color space conversion algorithm is used for converting video data from one color space (RGB) to another color space (YCbCr).

The equations to convert pixel data from 8-bit RGB to 8-bit YCbCr are:

$$y = 0.299 \times r + 0.587 \times g + 0.114 \times b$$
$$cb = -0.169 \times r - 0.331 \times g + 0.5 \times b + 128$$
$$cr = 0.5 \times r - 0.419 \times g - 0.082 \times b + 128$$

These equations can be implemented in fixed-point arithmetic as follows:

$$y = (77 \times r + 150 \times g + 29 \times b)/256$$
$$cb = (-43 \times r - 85 \times g + 128 \times b + 32768)/256$$
$$cr = (128 \times r - 107 \times g - 21 \times b + 32768)/256$$

Color Space Conversion Implementation in C

The following code converts NP pixels from the RGB color space to the YCbCr color space:

```
void rgb2ycc(signed char *RGB, signed char *YCC)
{
  int i;
  signed char r, g, b;
  signed char y, cb, cr;
  for (i = 0; i < NP; i ++)
  {
      r = *RGB++;
      g = *RGB++;
      b = *RGB++;
      y  = ( 77*r + 150*g +  29*b        ) >> 8;
      cb = (-43*r -  85*g + 128*b + 32768) >> 8;
      cr = (128*r - 107*g -  21*b + 32768) >> 8;
      *YCC++ = y;
      *YCC++ = cb;
      *YCC++ = cr;
  }
}
```

NOTE: You can find the source code files for this example in the Documentation section of Stretch's website.

Identify Code for Acceleration

After the application is written, the next step is to understand its performance and to identify ways to speed up its performance. Stretch provides several tools to assist in this process. One such tool is the profiler. Profiling the code with the simulator produces output similar to that in Table 1. Because of the small data set, the reset handler ResetH and the function main consume a significant number of cycles. What we are interested in for this example, however, is the rgb2ycc function, which requires 26607 cycles. The example used in this note has 640 pixels (NP=640), so there are just over 41 (26607/640) cycles required to process each sample.

Table 1 Excerpt of performance statistics for C implementation

%	Cumulative cycles	Self cycles	Number of calls	Self cycles/call	Total cycles/call	Function Name
33.75	26607	26607	1	26607	26607	rgb2ycc

Table 1 Excerpt of performance statistics for C implementation

%	Cumulative cycles	Self cycles	Number of calls	Self cycles/call	Total cycles/call	Function Name
26.03	47130	20523	??	??	??	ResetH
14.28	63229	16099	1	16099	50685	main

Two other tools that are also useful when examining an application's performance are the timer and the performance counters. The timer may be used with the simulator as in the following code:

```
startCycles = sx_get_ccount();
rgb2ycc(RGB, ycc);
endCycles = sx_get_ccount();
printf("%d cycles.\n", endCycles - startCycles);
```

This code prints "26642 cycles.", which is consistent with the profiling data. The performance counters are added similarly, but must be run on a remote target. The code is further modified as follows:

```
sx_perf_init(SX_COUNTER_0, SX_COUNT_STALLS_ALL);
sx_perf_init(SX_COUNTER_1, SX_COUNT_STALLS_DCACHE);
sx_perf_enable(SX_COUNTER_0 | SX_COUNTER_1);
startCycles = sx_get_ccount();
rgb2ycc(RGB, ycc);
endCycles = sx_get_ccount();
sx_perf_disable(SX_COUNTER_0 | SX_COUNTER_1);
sx_perf_read(SX_COUNTER_0, &totalStalls);
sx_perf_read(SX_COUNTER_1, &dcacheStalls);
printf("%d cycles, of which %d are stalls, %d of which are
due to the data cache.\n",
   endCycles - startCycles, totalStalls, dcacheStalls);
```

When run on a development board using the Redboot framework, we see the output:

```
28822 cycles, of which 4530 are stalls, of which 4174 are
due to the data cache.
```

The code requires more cycles to run on the development board than it does with the simulator. The difference is due to the simulator's memory model containing only the on-chip SRAM. When running on a development board, the RGB and ycc arrays reside in the external DDR SDRAM, which has longer latency than the on-chip SRAM. As we optimize the code, we will eliminate most of these stalls and reduce the cycle count dramatically.

Use Custom Instructions

The Software Configurable Processor lets us write custom instructions to accelerate compute-intensive operations. We will begin the optimization process by converting the `rgb2ycc` function to use a custom instruction.

Data can be moved into and out of the ISEF most easily using the wide registers (WRs). Following is the `rgb2ycc` function rewritten to load and store data using these WRs, and to perform the RGB to YCbCr color conversion using the `RGB2YCC_ONEPIXEL` custom instruction.

```
void rgb2ycc_isef(signed char *RGB, signed char *YCC)
{
  int i;
  WRA rgb, ycc;
  WRGET0INIT(0, RGB);
  WRPUTINIT(0, YCC);
  for (i = 0; i < NP; i++)
  {
     WRAGET0I(&rgb, 3);           //  Get 1 pixel (3 bytes)
      RGB2YCC_ONEPIXEL(rgb, &ycc);   //  Convert 1 pixel
      WRAPUTI(ycc, 3);               //  Put 1 pixel
  }
  WRPUTFLUSH();
}
```

The custom instruction used in the for loop is specified as a C function:

```
SE_FUNC void RGB2YCC_ONEPIXEL(WRA rgb, WRA *ycc)
{
    se_sint<8> r, g, b;
    se_sint<8> y, cb, cr;
    //  Unpack input
    r = rgb;
    g = rgb >> 8;
    b = rgb >> 16;
    //  Compute one pixel
    y  = ( 77*r + 150*g +  29*b        ) >> 8;
    cb = (-43*r -  85*g + 128*b + 32768) >> 8;
    cr = (128*r - 107*g -  21*b + 32768) >> 8;
    //  Pack output
    *ycc = (cr, cb, y);
}
```

Running the preceding code in the simulator shows that the loop now takes 4235 cycles, more than a 6x improvement. We have, however, only begun the optimization process. In the Stretch Report file (`rgb2ycc_1pixel.xr`), we can examine the resources required by the custom instruction. The resource report lists the following information about the custom instruction:

```
//   Computational Resources
//     Arithmetic bits.................124
//     Logic bits.......................0
```

```
//     Mux bits.........................0
//     Register bits....................0
//     Pipeline bits....................16
//   AU total..........................140 out of 4096
//     Multiply bits....................448
//   MU total..........................448 out of 8192
//   Extension registers.................0 out of 4096
```

Notice that we are using less than 10% of the resources available in the ISEF. The next step in the optimization process is to maximize the use of these resources by converting the instruction to process several pixels simultaneously.

Perform Calculations in Parallel

The `RGB2YCC_ONEPIXEL` instruction performs the calculations to process one pixel in parallel, but we can do more than one pixel at a time by better utilizing the wide registers. The wide registers are 16 bytes wide, but when processing one pixel at a time only 3 of those bytes are used. We can easily modify the application code to load 15 bytes of data instead of 3 bytes, and thus process pixels five times faster:

```
void rgb2ycc_isef_widedata(signed char *RGB, signed char
*YCC)
{
  int i;
  WRA rgb, ycc;

  WRGET0INIT(0, RGB);
  WRPUTINIT(0, YCC);
  for (i = 0; i < NP/5; i ++)
  {
      WRAGET0I(&rgb, 15);    // Get 5 pixels (15 bytes)
      RGB2YCC_FIVEPIXELS(rgb, &ycc);  // Convert 5 pixels
      WRAPUTI(ycc, 15);               // Put 5 pixels
  }
  WRPUTFLUSH();
}
```

The Extension Instruction that converts five pixels at a time is just the `_ONEPIXEL` instruction wrapped in a `for` loop with the input and output packing adjusted as follows:

```
SE_FUNC void RGB2YCC_FIVEPIXELS(WRA rgb, WRA *ycc)
{
    se_sint<8> r, g, b;
    se_sint<8> y, cb, cr;
    int i;
    *ycc = 0;
    for(i=0; i<5; ++i)
    {
        //  Unpack input
```

```
            r = rgb >> (24*i);
            g = rgb >> ((24*i) + 8);
            b = rgb >> ((24*i) + 16);
            //  Compute one pixel
            y  = ( 77*r + 150*g +  29*b        ) >> 8;
            cb = (-43*r -  85*g + 128*b + 32768) >> 8;
            cr = (128*r - 107*g -  21*b + 32768) >> 8;
            //  Pack result into output
            *ycc |= ((WR)(cr, cb, y)) << (24*i);
        }
}
```

Running this code shows that the loop now executes in 2903 cycles. The .xr file now lists the following information about the custom instruction:

```
//   Computational Resources
//     Arithmetic bits.................620
//     Logic bits.....................0
//     Mux bits.......................0
//     Register bits..................0
//     Pipeline bits..................80
//   AU total.........................700 out of 4096
//     Multiply bits..................2240
//   MU total.........................2240 out of 8192
//   Extension registers..............0 out of 4096
```

We are still not using all the computational resources available in the ISEF, but we are now making almost full use of the wide registers. It is possible that we might achieve more optimization, but we have reached the point of diminishing returns. We will therefore settle for five pixels of conversion per cycle.

We do not, however, need to settle for this cycle count. Our previous optimization improved efficiency by a factor of 5, but the cycle count only dropped by a factor of about 1.5. There must be something other than the calculation that is taking up processor cycles.

Using High-Performance Memories

We profile the code again, this time with instruction-level profiling enabled:

%	Time	Total Cycles	Total Instr	Calls	Cycles/ Call	I$ Misses	D$ Misses	Loads Stores	Function
37.686	20545	4511	1	20545	15	3	3	0	ResetH
29.511	16088	14903	1	16088	10	119	3845	0	main
5.261	2868	393	1	2868	8	242	122	120	rgb2ycc…

We can also run the application on a remote target to measure the performance on the hardware:

```
4961 cycles, of which 4598 are stalls, 4028 of which are
due to the data cache.
```

We can see in both measurements the progress we have made – the profiler shows that the `rgb2ycc` function now requires only about 10% of the cycles it initially consumed. We can also see that our efforts to improve the computational efficiency are being thwarted by data cache misses.

The simulator shows that there are 242 data cache (D$) misses in the `rgb2ycc` function. Its memory model has all data reside in the cached on-chip SRAM by default. When data is read, if it is in the cache it can be accessed without stalls, but each cache miss costs approximately 10 cycles. Of the 2868 cycles required by the function, approximately 242*10 = 2420 cycles are being spent stalled while the cache is loaded with the required data.

When running on a remote target, the default location for data is in the off-chip DDR SDRAM. This memory has both longer and more variable latency than the on-chip SRAM. This increased latency means that the 242 data cache misses are responsible for over 4000 cycles when the application is run on a remote target.

Both measurements show that a large fraction of the cycles used to execute the `rgb2ycc` function are processor stalls, and that the data cache is responsible for most of those stalls. By moving the input and output arrays into the zero-latency internal data RAM, we can eliminate these data cache misses. Because of our small data set, we can simply declare the arrays to be in the internal RAM. To do this, we add an `__attribute__` to the variables. For example,

```
signed char ycc[3 * NP];
```

Becomes:

```
signed char ycc[3 * NP] __attribute__
((section(".dataram.data")));
```

Normally, much larger data sets are used. For those cases, buffers can still be declared in the data RAM. The S5's memory-to-memory direct memory access (MMDMA) hardware can be configured move new input data into the high-performance memory and to move the results from it. This data RAM has two ports; one for the processor and one for the MMDMA. The two ports allow for data movement and processing both to proceed without interfering with each other. Stretch provides SBIOS functions that facilitate the movement of data into and out of the dual-port data RAM.

Moving the input and output arrays into the internal data RAM reduces the cycle count to 525 in simulation. On a remote target, we see:

```
776 cycles, of which 414 are stalls, 0 of which are due to
the data cache.
```

We can also estimate the ideal performance of our application. We are processing 640 pixels, and we process five pixels every three cycles. The total cycles should therefore ideally be close to 640*3/5 = 384 cycles. The extra cycles include some loop overhead and instruction cache misses. Because this overhead is independent of the number of pixels processed, we can check the incremental number of cycles required to process each pixel in a steady-state. Reducing the number of pixels from 640 to 320 reduces the cycle count by 192. Each pixel thus requires 192/320 = 3/5 of a cycle. The performance of our code thus matches our expectations.

Optimization with Stretch Processors

In this example, we have reduced the execution time for this color conversion loop by a factor of 50. We gained this improvement by writing custom instructions, extending them to perform calculations in parallel, and using the S5's high-performance memories. These steps apply in general; they are the major steps involved in accelerating any performance-critical code on the Software Configurable Processor.

Throughout the optimization process, we considered the performance we expected. When we discovered to our surprise that processing the data five pixels at a time was only slightly better than processing it one pixel at a time, further investigation revealed that we could optimize the application further by using the high-performance memory. At the end of the process, we verified the time required to compute additional data indeed matches our expectations. Estimating the expected performance of a loop and checking it against the true performance is useful for identifying additional opportunities for optimization.

Another optimization technique is to consider what factor is limiting the performance. For example, when we processed only one pixel at a time, it was clear that neither ISEF resource availability nor the size of the wide registers was gating the performance. By expanding the load width to nearly the width of the WRs, we were able to accelerate the performance further.

The concepts presented for this example apply broadly; they are the same steps you will follow as you optimize your own C and C++ code. Each problem is different, but this example shows the important parts of the process and can be used as a model for your own optimization work.