Storm-1 Benchmarks

This document describes the Storm-1 benchmark program. It briefly discusses benchmark implementation. It explains how to build and run the benchmark program. It reports benchmark performance for Storm-1 processors and draws conclusions about stream program design and implementation from the performance data..

S1BK-00001-003

Stream Processors, Inc.
455 DeGuigne Drive
Sunnyvale, CA 94085-3890 USA
Telephone: +1.408.616.3338
Fax: +1.408.616.3337
Email: info@streamprocessors.com
Web: www.streamprocessors.com

Table of Contents

1 Introduction

This document describes the Storm-1 benchmark program. It explains how to build and run the program and reports benchmark performance for Storm-1 processors. It assumes that you are familiar with Stream programming and with the use of the Stream toolset, as described in *Stream User's Guide*.

The Storm-1 benchmark program performs several common digital image processing tasks. Image processing tasks are very compute-intensive: they perform very large numbers of arithmetic operations. They make good benchmarks for a stream processor because they perform identical operations on large numbers of data items, allowing very efficient implementation on a SIMD processor. Wikipedia article Image processing provides general information about digital image processing.

The Storm-1 benchmark program performs the following tasks:

- **color_convert** Color space conversion (RGB to/from YUV)
- **convolve_3x3_16** 3×3 convolution filter using 16-bit filter coefficients
- **convolve_3x3_8** 3×3 convolution filter using 8-bit filter coefficients
- **error_dispersion** Quantization error dispersion (Floyd-Steinberg dithering)
- **sad_16x16** Sum of absolute differences on 16x16 blocks

Section Benchmark overview below gives a brief high-level overview of each task. The Implementation section describes some details of the stream processor implementation of each task. Build the benchmark program describes how to build **benchmark**, and Run the benchmark program gives detailed information on how to run it. Command line options control its behavior, so executing the same program with different options provides performance information on different tasks. It runs either under the simulator **spsim** or on a Stream Processor hardware device. When the program completes execution of a task, it prints information on pipeline performance or on kernel performance. The Performance data section presents current benchmark performance data, and a final section draws some Conclusions from the data.

1.1 Document revision history

Document number	Date	Description	Release Version
S1BK-00001-001	December 2007	Initial release	Initial release
S1BK-00001-002	September 2008	Revision	RapiDev 1.0.4
S1BK-00001-003	May 2009	Major revision	Stream 2.3

2 Benchmark overview

The Storm-1 benchmark program, called **benchmark**, performs several digital image processing tasks.

2.1 Color conversion

benchmark can perform color conversion between RGB and YUV color spaces. Wikipedia article YUV gives information about color spaces and color space conversion. **benchmark** performs color conversion between 24-bit RGB (red/blue/green color components) and 24-bit YUV (luma and chrominance components) by performing a matrix multiplication on each pixel. Each 24-bit conversion requires nine multiplications and six additions, so a 1920 by 1080 image (2 megapixels) requires about 18 million multiply/accumulate operations.

A *bitmap* file, typically identified with extension **.bmp**, is a common image file format for RGB data. **benchmark** reads and writes bitmap files that contain uncompressed 24-bit RGB image data; it does not read or write other varieties of bitmap files. You can use most image viewers (for example, Linux **gimp** or Windows Picture and Fax Viewer) to view bitmap files. The benchmark distribution includes a 1920 by 1080 bitmap file **data/image.bmp** used for **benchmark** input.

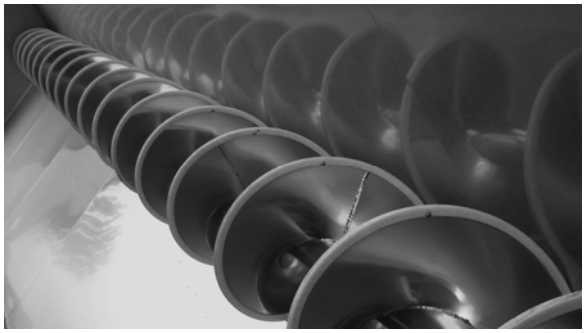
A *YUYV* file, typically identified with extension **.yuyv**, contains a YUV image in subsampled YUYV (also called YUV422) format. Under Windows, you can view a YUYV file with the Elecard YUV viewer, available for download in a free trial version.

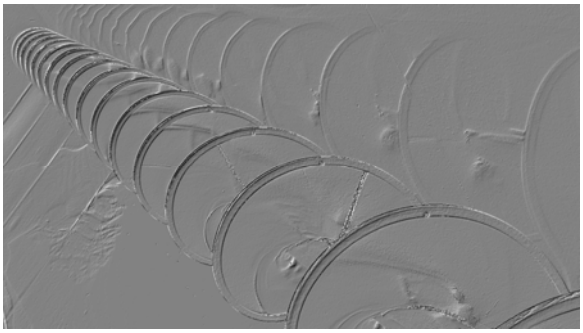
2.2 Convolution

benchmark can apply a 3x3 convolution filter to image data. A convolution filter finds a new value for each pixel of an image by multiplying the values of neighboring pixels by filter coefficients and summing the products. A 3x3 convolution computes $p'_{xy} = \sum_{j=y-1}^{y+1} \sum_{i=x-1}^{x+1} p_{ij} * z_{ij} + bias$, where **p** are pixel values, **z** are filter coefficients, and **bias** is a bias added to the result. Simple convolution filters blur or sharpen an image, find edges in an image, or produce effects like embossing. **benchmark** provides a variety of 3x3 convolution filters, selectable by command line option. The convolutions allow different degrees of precision (either 8-bit or 16-bit filter coefficients).

benchmark applies a 3x3 convolution filter to 8-bit raw image data (planar data), not to 24-bit RGB data. You cannot view 8-bit image data files directly with most image viewers, but you can use **benchmark** options to convert them to YUYV files (mode 3) or to bitmap files (mode 3, then mode 1). The benchmark distribution includes a 1920 by 1080 raw image file **data/image.gray** used for **benchmark** input.

The following images show the effect of an embossing filter (**benchmark -m=5 -z=8**).



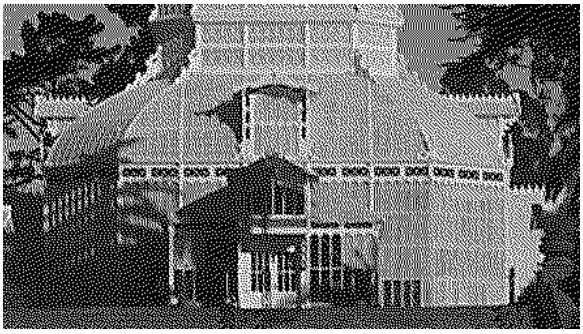


2.3 Error dispersion

benchmark can perform quantization error dispersion using the Floyd-Steinberg dithering algorithm. Wikipedia article Floyd-Steinberg dithering describes the algorithm. If a device can represent only certain pixel values, quantization error is the difference between the original pixel value and the represented pixel value. Floyd-Steinberg dithering distributes quantization error to adjacent pixels, resulting in better image rendering. The algorithm processes the image rowwise, left to right, top to bottom.

benchmark applies error dispersion to 8-bit raw image data (planar data), not 24-bit data. It quantizes the image by replacing each 8-bit pixel value (0 to 255) with either 0 or 255 (a 1-bit value, essentially).

The following images show the effect of applying error dispersion (**benchmark -m=6**). This is a portion of a larger image, cropped so that you can see the effect of the dithering in the reduced image shown here.



2.4 Sum of absolute differences

The sum of absolute differences (SAD) is commonly used in motion estimation for video compression, as described in Wikipedia. SAD measures the difference between two images, an input image and a reference image, by summing the absolute values of the differences between the values of corresponding pixels in the two images over a region. A 16x16 SAD computes $sad_{xy} = \sum_{j=16*y}^{y+15} \sum_{i=16*x}^{x+15} |p_{ij} - r_{ij}|$, where **p** are image pixel values, **r** are reference pixel values, and **sad** are block sums.

benchmark performs sum of absolute differences for 16x16 blocks (often called macroblocks) in images containing 8-bit raw data. Its output is an array of words containing block sums. Since each sum represents a 256-pixel macroblock, the output contains 1/256th as many items as the input. The computation requires one subtraction, one absolute value, and one addition per pixel, so computing SAD for a 1920 by 1080 (2 megapixel) image requires about 18 million operations.

3 Implementation

This section gives an brief discussion of some implementation issues and then describes some implementation details of benchmark tasks. Comments in the benchmark source files give additional implementation details.

3.1 Data flow analysis

Data flow analysis is a critical phase of stream program design. To use the stream processor efficiently, the programmer must decide what part of a task should be executed on the DPU and how to present the program data so the DPU can process it efficiently. The DPU processes data from the LRF, which is of limited size; typically, a program maps successive pieces of the program input and output data ("strips") through the LRF, using stream loads and stores. One approach might use indexed loads to reorder kernel input data, while an alternative approach uses block loads (faster than indexed loads) and lets the kernel do additional work (slower); which approach is faster often can be determined only empirically.

Program requirements are important program design considerations. If a pipeline requires one input stream and one output stream of the same size, requires double buffering for performance, and uses 256 words for local arrays, it has a maximum stream size of (**SPI_LRFSIZE** - 256) / 4 words per lane (960 on Storm-1). If the pipeline processes a complete row of an image in each lane with a single kernel call, it can process a maximum width of 3840 pixels if the image is represented with 1 byte per pixel, 1920 pixels if 2 bytes per pixel, or 960 pixels if 4 bytes per pixel. If the program must handle larger images, it must use a different data flow design.

benchmark processes images up to 1920 by 1080 in size. Smaller images might require padding before processing by **benchmark**; it is a benchmark program, not a general image manipulation program, so it does not handle all image sizes.

3.2 Fixed point arithmetic

Storm-1 processors do not support hardware floating point operations, so benchmark tasks use fixed point fractional arithmetic. Fixed point arithmetic uses efficient integer arithmetic operations on operands with known ranges; operands and results are implicitly scaled by factors of 2^*n*. *Stream User's Guide* provides a detailed discussion of fixed point arithmetic.

3.3 Performance analysis

There is no substitute for careful performance analysis during the implementation of stream programs. Build a program in profile mode, run it, and then examine the generated program profile to see its performance in detail; modify it, see how its performance changes, and repeat the cycle. Unlike a conventional sequential processor, a stream processor performs many different operations simultaneously; for example, DSP MIPS can execute DSP MIPS code, the stream controller can perform a load, the stream controller can perform a store, and the DPU can execute kernel code, all at the same time. The performance optimization chapter of *Stream User's Guide* gives some suggestions of how to approach performance optimization.

Deeply rooted programming habits can have unexpected results on a stream processor. For example, in conventional programming reducing the number of operations in a block improves performance, but in a stream processor code with fewer operations sometimes has more dependencies between operations, leading to a VLIW schedule less efficient than the schedule for similar code with more operations but fewer dependencies and hence a

shorter critical path. Similarly, adding operations to a loop that does not use all available ALUs might not change the time the loop requires.

Keep in mind the following hints:

- Use profile mode to generate a profile, then examine it carefully.
- Use double buffering.
  - Check the visualization; try to keep either the stream controller or the DPU fully occupied.
- Use software pipelining on kernel inner loops.
  - Check the profile kernel analysis and kernel visualization.
  - Shorten the critical path of kernel inner loop blocks.
  - Use kernel intrinsics for efficiency.
- Use loop unrolling on kernel inner loops.
  - Determine empirically whether unrolling improves performance.
- Use fixed point arithmetic.

3.4 Color conversion

Color conversion converts a bitmap file (RGB color space) to or from a YUYV file (YUV color space). Both files contain uncompressed color data in row major order, but a bitmap file orders image rows from bottom to top, while a YUYV file orders image rows from top to bottom. **benchmark** color conversion routines invert the row order during color conversion, not as a separate step. The color conversion pipelines use indexed loads (**spi_load_index**) to reorder data when loading it to the LRF. An alternative approach would be to perform the row reordering as a separate step, so the pipelines could use faster block loads (**spi_load_block**) instead. The color conversion of each pixel is independent of other pixel data, so pixels can be processed in any order.

A bitmap file contains 24-bit RGB pixel data, but the stream processor DPU processes data only in 32-bit words. The color conversion task could pad the 24-bit data for each pixel to 32 bits before processing it. In the current **benchmark** implementation, it instead processes RGB input by reading three successive words (12 bytes, containing 4 RGB pixels) in the color conversion kernel. An application that performed more complex processing of RGB data might find it more efficient to pad the data first instead.

A YUYV file contains 24-bit YUV pixel data in subsampled format; each 4 bytes of input contains data for two pixels, with the U and V chrominance data reused for both pixels. The **benchmark** color conversion performs the subsampling directly, converting two pixels from six bytes of YUV to four bytes of YUYV. An application that performed further processing of the YUV data might prefer to perform the subsampling at a later processing stage instead.

3.5 Convolution

For a 3x3 convolution filter, each computed result pixel value depends on the value of a pixel and its eight immediately adjacent pixels; equivalently, each pixel contributes to nine different results. The **benchmark** implementation processes data in rows. For each pixel in a row, it computes the contribution of the pixel to the previous row, the current row, and the following row. It writes out partial sums for pixels in the current row and the next row, and it writes a result for the previous row, since it finishes the previous row computation. The processing of the next row reads and updates the partial sums.

The convolution filter implementation also must deal with the edges of the image, as the computation for an edge pixel requires the value of a non-existent neighbor. The implementation pads the image by taking the value of the

edge pixel as the value of the non-existent neighbor pixel. It performs this padding in two different ways: the pipeline LRF loads duplicate the bottom and top rows, and the kernel pads the left and right columns.

The kernel that performs the convolution reads *n* rows of input and writes *n* rows of output, but only *n*-2 rows of the output contain valid data. The pipeline that invokes the kernel stores only the valid substream of the kernel's output stream. Many alternative approaches would work: the pipeline could pass partial sums from one kernel invocation to the next, for example.

3.6 Error dispersion

Error diffusion (Floyd-Steinberg dithering) is a particularly instructive Stream programming example. The algorithm disperses quantization error from each pixel to its adjacent neighbors to the right, bottom left, bottom center, and bottom right. As a result, an implementation can process the image by rows in a single pass, from left to right, top to bottom. But this presents a challenge to a VLIW implementation: each pixel requires dispersed error data from its immediate predecessors to compute its result. If each lane of the DPU receives adjacent pixel data, it loses the advantage of data parallelism, because each lane must wait for the dispersed error from a previous lane.

The **benchmark** error dispersion implementation avoids this problem by staging the computation in a way that keeps all lanes fully occupied most of the time. Each lane processes a row of the image, but data is loaded with a stagger between lanes. Thanks to the stagger, each lane has all the data required to complete its computation at every stage. Some lanes perform computation on lane padding at the beginning and the end of kernel execution, but most of the time the kernel is fully occupied doing useful work.

To illustrate, consider a 4-lane stream processor performing error diffusion on a 16 by 8 image. The input image contains 128 pixels, or 32 packed records. The raw image file stores records in row major order, starting from the top:

i0	**i1**	**i2**	**i3**
i4	**i5**	**i6**	**i7**
i8	**i9**	**i10**	**i11**
i12	**i13**	**i14**	**i15**
i16	**i17**	**i18**	**i19**
i20	**i21**	**i22**	**i23**
i24	**i25**	**i26**	**i27**
i28	**i29**	**i30**	**i31**

Error diffusion on a 4-lane stream processor processes the 8 image rows with two load/kernel/store cycles. It loads each row (4 records) into a single lane, with an additional **SPI_LANES** - 1 words of padding, for a total of 7 records per row. The padding words for the first load/kernel/store cycle (processing the top half of the image, records **i0** to **i15**) are shown here shaded:

Image row								
0		**i0**	**i1**	**i2**	**i3**			
1			**i4**	**i5**	**i6**	**i7**		
2			**i8**	**i9**	**i10**	**i11**		
3			**i12**	**i13**	**i14**	**i15**		

The start of the valid data in each row is offset by one word from the start of data in the previous row. **spi_load_stride** loads input data and padding into the LRF for the first kernel invocation so that successive **spi_read** calls in each kernel loop iteration process the records in the order shown here.

Image row	Lane	Iter.:	1	2	3	4	5	6	7
0	0		**i0**	**i1**	**i2**	**i3**	**i4**	**i5**	**i6**
1	1		**i3**	**i4**	**i5**	**i6**	**i7**	**i8**	**i9**
2	2		**i6**	**i7**	**i8**	**i9**	**i10**	**i11**	**i12**
3	3		**i9**	**i10**	**i11**	**i12**	**i13**	**i14**	**i15**

The error diffusion kernel reads LRF input data in the order shown above and writes result data to the LRF in the order shown below, where **o***n* is the *n*th output record and **x***n* is the *n*th padding record:

Image row	Lane	Iter.:	1	2	3	4	5	6	7
0	0		**o0**	**o1**	**o2**	**o3**	**x6**	**x7**	**x9**
1	1		**x0**	**o4**	**o5**	**o6**	**o7**	**x8**	**x10**
2	2		**x1**	**x3**	**o8**	**o9**	**o10**	**o11**	**x11**
3	3		**x2**	**x4**	**x5**	**o12**	**o13**	**o14**	**o15**

After the kernel finishes, **spi_store_indexed** stores the output stream from the LRF to the output image buffer, storing both padding words and valid output. The index stream rearranges the LRF data into the desired order:

o0	**o1**	**o2**	**o3**
o4	**o5**	**o6**	**o7**
o8	**o9**	**o10**	**o11**
o12	**o13**	**o14**	**o15**
x0	**x1**	**x2**	**x3**
x4	**x5**	**x6**	**x7**
x8	**x9**	**x10**	**x11**

The error diffusion pipeline then repeats the load/kernel/store process to process the bottom half of the image (input records **i16** through **i31**, overwriting the padding from the first indexed store with valid data from the second indexed store.

The diagram below shows kernel variables in two successive rows of the image during processing by the kernel. The diagram rows correspond to the image layout; each rectangular box represents a word (four pixels). The iteration number in the top left of the box shows when the kernel processes the word; here the kernel has completed iteration *i*-1 of its processing loop and is about to perform iteration *i*. **e***n* are accumulated errors and **n***n* are accumulated errors for the following row; each is a vector variable, with a different value in each lane. Each lane disperses errors **e0**, **e1**, **e2**, and **e3** to **e4** (to the right) and to **n***n* (to the next row). The computation uses the loop-carried values from iteration *i*-1 to compute new variable values. For example, the lane *n* value of **e0** in iteration *i* is the sum of error dispersed right from the pixel to its left (iteration *i*-1's value of **e4** from lane *n*) plus error dispersed down from pixels in the previous row (iteration *i*-1's value of **n0** from lane *n*-1). See the source code in **disperse.sc** for details.

Lane *n*-1	iteration		*i*-2				*i*-1				*i*
	row *n*-1 error						**e0**	**e1**	**e2**	**e3**	**e4**
	next row error					**nm1**	**n0**	**n1**	**n2**	**n3**	**n4**
Lane *n*	iteration		*i*-1				*i*				*i*+1
	row *n* error		**e0**	**e1**	**e2**	**e3**	**e4**				
	next row error	**nm1**	**n0**	**n1**	**n2**	**n3**	**n4**				

3.7 Sum of absolute differences

The **benchmark** sum of absolute differences pipeline performs an independent SAD computation in each lane (SIMD parallelism). It calls a kernel to generate an index stream and then uses indexed loads to load image data and reference data so that data for successive macroblocks go to successive lanes. After the loads, the pipeline calls an extremely efficient kernel to perform 16x16 SAD on 8-bit data. Since each lane contains all the data it needs for the computation of the SAD for a macroblock, no interlane communication is necessary. The kernel uses kernel intrinsics to perform byte operations (subword parallelism). The Conclusions section discusses SAD performance in more detail.

4 Build the benchmark program

You can build and run the benchmark program directly from the command line, or you can build and run it under the Stream Processors integrated development environment **spide**. The benchmark distribution contains:

- **data/** image data files
- **launch/** **spide** run configurations
- **src**/ source files
- **.project, .cproject** **spide** project files

The benchmark program is a DSP MIPS testbench executable. Benchmark distribution source directory **src/** contains sources (**.c** and .**sc**) and headers. You can compile the benchmark in functional mode (useful for detailed interactive stepping through program execution, but not useful for performance evaluation), in debug mode, in profile mode, or in release mode. The section below describes how to build the release mode version.

You must build two binaries if you wish to obtain both pipeline performance data and kernel performance data. Gathering accurate kernel performance data requires slowing down overall program performance, so a single binary cannot accurately produce both pipeline data and kernel data.

Compilation of pipelined kernel code may be slow (several minutes) in device modes; be patient.

4.1 From command line

To build a version of **benchmark** for Storm-1 SP-16 that reports pipeline performance data when you run it, **cd** to source directory **src/** and type:

```
$ spc -o benchmark_p *.c *.sc -m testbench
```

You can build a version that reports kernel performance data instead by compiling with **spc** option **-DKTIMER**:

```
$ spc -o benchmark_k *.c *.sc -m testbench -DKTIMER
```

You can compile Storm-1 SP-8 versions of **benchmark** by adding **spc** option **-m sp8** to the **spc** command line.

4.2 Under spide

The benchmark distribution includes **spide** project files. To import and build the **benchmark** project:

- Invoke **spide**.
- Select **File >> Import >> General >> Existing Projects into Workspace**.
- Navigate to the benchmark distribution.
  - Check the **Copy projects into workspace** checkbox if you wish to work with a copy of the distribution.
- Import the project.
- Select the desired machine and mode (for example, **SP-16 release**).
- Pull down the arrow next to the build icon (the hammer) and select **benchmark**.

spide will build two modules, **benchmark_p** and **benchmark_k**, for benchmarking pipeline performance and kernel performance respectively.

5 Run the benchmark program

The Storm-1 benchmark program usage is:

```
$ ./benchmark --help
Usage: benchmark [ option ... ] [ infile [ outfile ] ]
Options:
        -c      Use C reference implementations (not kernels)
        --help  Print help message
        -h=<n>  Set image height to <n> (default: 1080)
        -m=<n>  Set mode to <n>:
                0: 24-bit RGB (.bmp) to raw YUYV (default)
                1: Raw YUYV to 24-bit RGB (.bmp)
                2: Raw YUYV to 8-bit raw grayscale
                3: 8-bit raw grayscale to YUYV
                4: 3x3 convolution filter with 8-bit coefficients
                5: 3x3 convolution filter with 16-bit coefficients
                6: Error dispersion (Floyd-Steinberg dithering)
                7: Sum of absolute differences 16x16
        -r=<file> Use reference <file> for SAD
        -s      Silent: do not print performance stats
        -w=<n>  Set image width to <n> (default: 1920)
        -z=<n>  Use convolution filter <n>:
                0: identity
                1: blur1
                2: blur2
                3: sharpen1
                4: sharpen2
                5: edge enhance
                6: edge detect
                7: emboss1
                8: emboss2
```

The **-c** option tells **benchmark** to use reference C implementations rather than pipelines and kernels; it may be used to verify that the pipeline version produces the expected result. Not all modes have both C and Stream implementations; see source file **benchmark.c** for details. Modes 0, 1, 4, and 5 print performance data. The **-h** and **-w** options set the input image height and width for YUYV and raw 8-bit files; they are ignored for bitmap input files, as the bitmap file header defines the image height and width.

For 3x3 convolution filters (modes 4 and 5), the **-z** option selects from a variety of filters. The amount of computation required for a filter is independent of the filter choice, so the reported performance should be the same for all filters.

Sum of absolute differences (mode 7) requires a reference file in addition to the input file. Specify a reference file with the **-r** option.

5.1 From command line

The benchmark distribution includes data files in directory **data/**. The following commands use **benchmark_p** to perform color conversion (mode 0), 3x3 convolution with 8-bit filter coefficients (mode 4), 3x3 convolution with 16-bit filter coefficients (mode 5), error diffusion (mode 6), and sum of absolute differences (mode 7). The SAD command line specifies a reference file with option **-r** and specifies an image height with option **-i** (because the

default image height 1080 is not a multiple of 16). The performance data you see may differ from the data shown below. To run under the simulator **spsim**;

```
$ spsim benchmark_p -m=0      data/image.bmp data/result0.yuyv
...
SP-16 RGB to YUYV color conversion pipeline performance:
        Execution time:         3126750 ns for 2073600 24-bit pixels
        Average time:           1.51 ns per 24-bit pixel
...
$ spsim benchmark_p -m=4 -z=7 data/image.gray data/result47.gray
...
$ spsim benchmark_p -m=5 -z=7 data/image.gray data/result57.gray
...
$ spsim benchmark_p -m=6      data/image.gray data/result6.gray
...
$ spsim benchmark_p -m=7 -h=1072 -r=data/image2.gray \
                              data/image.gray data/result7.gray
...
```

Similarly, to run on a Stream Processor hardware device, use **sprun** rather than **spsim**:

```
$ sprun benchmark_p -- -m=0      data/image.bmp  data/result0.yuyv
...
$ sprun benchmark_p -- -m=4 -z=7 data/image.gray data/result47.gray
...
$ sprun benchmark_p -- -m=5 -z=7 data/image.gray data/result57.gray
...
$ sprun benchmark_p -- -m=6      data/image.gray data/result6.gray
...
$ sprun benchmark_p -- -m=7 -h=1072 -r=data/image2.gray \
                               data/image.gray data/result7.gray
...
```

Running these commands generates the pipeline performance data presented in the Performance section below. To obtain kernel performance data instead, simulate or run **benchmark_k** rather than **benchmark_p**, using the same options as above.

5.2 Under spide

The benchmark **spide** project includes run configurations to run under the simulator **spsim**. After you build the project, select run configuration **color_convert_k**, **convolve_3x3_8_k**, **convolve_3x3_16_k**, **error_disperse_k**, or **sad_16x16_k** to run **benchmark_k** under the simulator and print kernel performance data. Select run configuration **color_convert_p**, **convolve_3x3_8_p**, **convolve_3x3_16_p**, **error_disperse_p**, or **sad_16x16_p** to run **benchmark_p** under the simulator and print pipeline performance data.

To run on a Stream Processor hardware device, you must modify the run configurations. Make sure the run configuration you wish to modify is writable (run configurations are in directory **launch/**). In the **Run configurations** tab **Target**, select **Device** and then fill in the target IP address, user, and working directory. You must also specify input files and output files (for example, **benchmark/data/image.bmp** and **benchmark/data/result0.yuyv**) for each hardware run configuration.

6 Performance data

Performance data in this section results from execution of **benchmark_p** and **benchmark_k** on a Stream Processor hardware device. SP-8 performance data below is from execution of SP-8 versions (that is, versions compiled with **spc -m sp8**) on an SP-16 hardware device, not from execution on actual SP-8 hardware.

Performance data from simulation generally differs slightly from the hardware data, even with the use of **spsim** options **--dsp_mips_clock** and **--mem_clock** to specify clock frequencies comparable to the hardware device**.** The simulator times are typically around 5% slower than actual hardware performance times.

benchmark uses timers to obtain performance data. Kernel performance times include the total time required for all invocations of the benchmarked kernel. Kernel performance is the best measure of the computational power of the processor's DPU, with its VLIW SIMD design.

Pipeline performance times measure the time spent in the pipeline, not in the entire program. Pipelines operate on stream programming mode buffers, and pipeline execution does not include the loading or storing of the buffers. For example, color conversion (**benchmark** mode 0) loads a **.bmp** image file containing RGB data into a buffer, runs the color conversion pipeline to produce YUYV data in another buffer, and writes the resulting .**yuyv** file. The performance data includes only the pipeline portion of the program's execution time, not the file reads and writes.

The table below presents performance data below in nanoseconds (ns) per pixel, where a pixel is a 24-bit RGB pixel for **color_convert** and an 8-bit pixel for **convolve_3x3_8, convolve_3x3_16**, **error_disperse**, and **sad_16x16**. To convert **sad_16x16** times to ns per block, multiply by 256.

Date:	3/06/09
Toolset:	Stream 2.3 pre-release (build 786)
DSP MIPS clock:	250 MHz
DPU clock:	500 MHz
DDR:	203 MHz
Input data:	1920 x 1080 image (**data/image.bmp** or **data/image.gray**)

Benchmark	Pixel size	SP-16 kernel	SP-16 pipeline	SP-8 kernel	SP-8 pipeline
color_convert	24 bits	0.29 ns/pixel	1.32 ns/pixel	0.59 ns/pixel	1.46 ns/pixel
convolve_3x3_8	8 bits	0.84 ns/pixel	0.93 ns/pixel	2.06 ns/pixel	2.17 ns/pixel
convolve_3x3_16	8 bits	0.85 ns/pixel	0.94 ns/pixel	2.08 ns/pixel	2.19 ns/pixel
error_disperse	8 bits	2.91 ns/pixel	3.24 ns/pixel	5.73 ns/pixel	6.00 ns/pixel
sad_16x16	8 bits	0.04 ns/pixel	1.17 ns/pixel	0.08 ns/pixel	1.90 ns/pixel

The performance data above uses a 500 MHz DPU clock (i.e., 2 ns/cycle), so multiplying ns/pixel data by 0.5 restates kernel performance in units of DPU cycles/pixel. Kernel performance times scale linearly with DPU clock frequency. Pipeline performance times depend on both DSP MIPS and DPU clock frequencies; they will scale almost linearly with DPU clock frequency for DPU-limited tasks such as the convolution filters above, but not for other tasks.

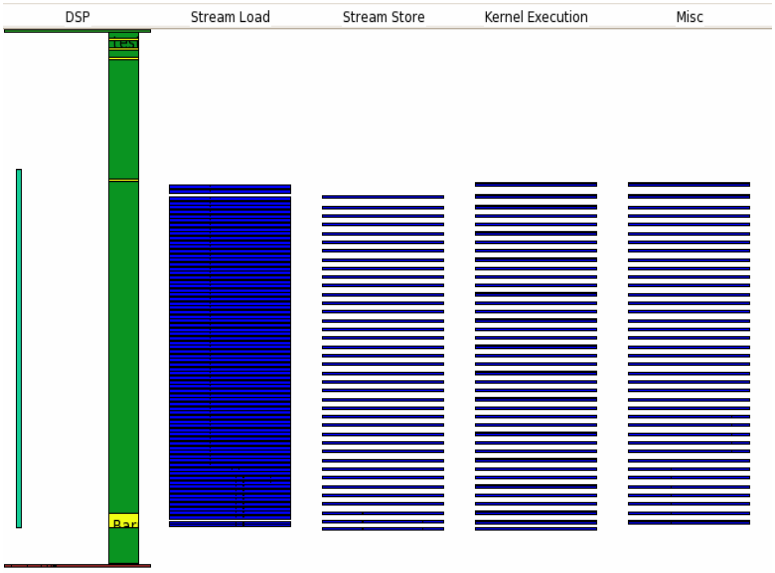
7 Conclusions

To examine benchmark performance in detail, build **benchmark** under **spide** in profile mode and run the version that reports pipeline performance. Open the generated profile to see the analysis and visualization of pipeline performance (LRF load/stores and kernel executions) over time. This section shows some examples of benchmark pipeline and kernel visualizations.

7.1 DMA-limited pipelines

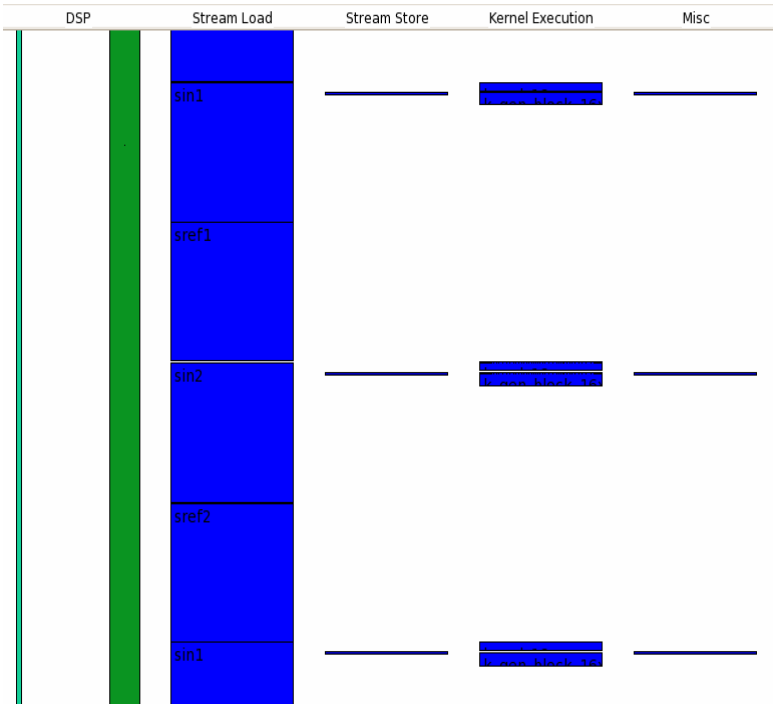
Color conversion and sum of absolute differences perform simple calculations, so the kernels that implement them are very fast. As a result, their pipeline performance is DMA-limited; the loads and stores that move pixel data to and from the LRF are much slower than the kernels. The performance data in the preceding section reflects this clearly: kernel performance is much faster than pipeline performance. Further improving kernel performance would have no effect on overall pipeline performance. DMA-limited pipelines are good candidates for merging with other tasks; the DPU has time available to perform more useful work.

A profile mode execution of **sad_16x16_p** produces the following pipeline visualization:



The longer green bar represents the entire DSP MIPS execution time of the program. The smaller bar to its left is the **sad_16x16** pipeline timer. The almost solid bar in the Stream Load column shows that load bandwidth is the limiting performance factor for this pipeline. The spaces between the horizontal bars in the Stream Store and Kernel Execution columns represent time that the stream controller store unit or the DPU is idle while waiting for a dependency to be fulfilled.

A magnified view from the middle of the pipeline shows more detail:

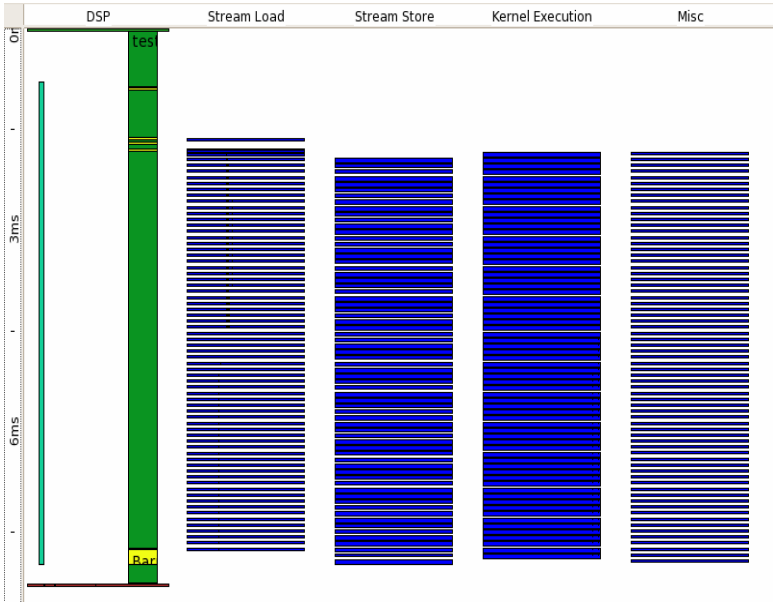


The double-buffered pipeline alternately loads streams **sin1** and **sref1**, then **sin2** and **sref2**. After the pipeline loads the input and reference data, it executes the SAD kernel **k_sad_16x16**, shown here in the Kernel Execution column as the very thin upper rectangle of each pair. The kernel runs very fast, as discussed in the Kernel performance section below. When the kernel finishes, it stores its result. The store in the Stream Store column is very fast in comparison to the loads because each load/kernel/store cycle loads 128 words of data per block (512 bytes, 16x16 bytes each of image and reference) but stores only one word per block. The lower rectangles of each pair in the Kernel Execution column represent index generation kernel executions.

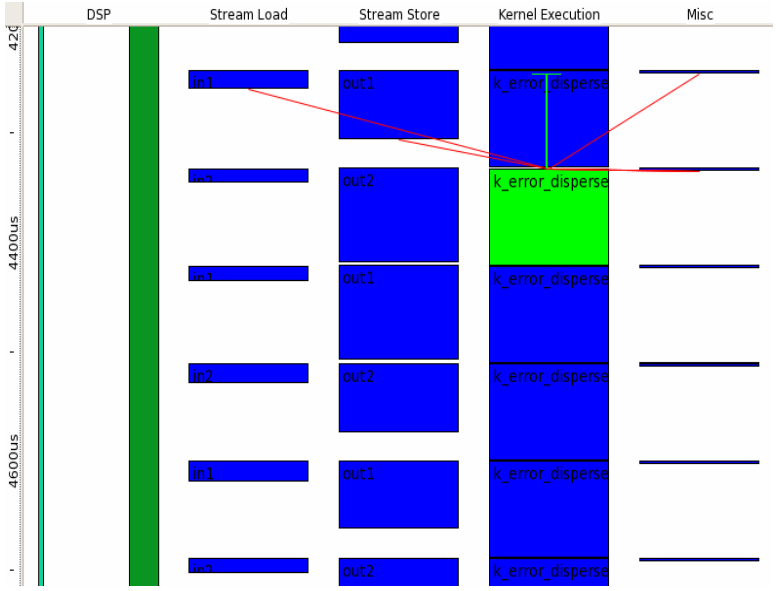
7.2 DPU-limited pipelines

Convolution filters and error dispersion perform more complex calculations. As a result, their pipeline performance is DPU-limited; the kernels take longer than the LRF loads and stores. For DPU-limited pipelines, improving kernel performance would further improve overall pipeline performance. These tasks keep the DPU fully occupied once the double buffered main loop of the pipeline begins (see the visualization below), but setting up the pipeline for the loop takes some time, creating pipeline overhead (the difference between kernel time and pipeline time). Performance of the 8-bit and 16-bit convolution filters is almost identical; there is no performance penalty for using the higher-precision 16-bit version.

Profile mode execution of **error_disperse_p** produces the following pipeline visualization:



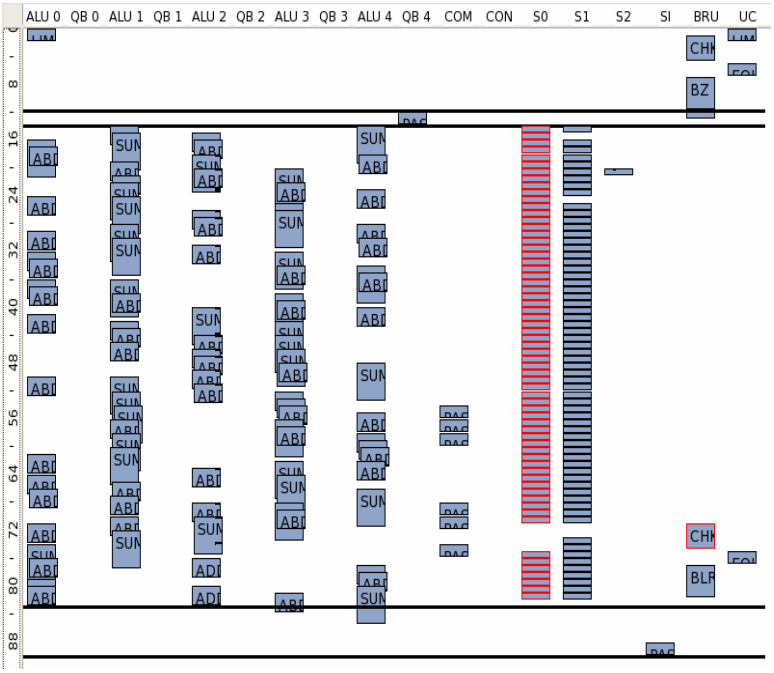
The longer green bar represents the DSP MIPS execution of the entire program, while the smaller bar to its left is the **error_disperse** pipeline timer. The almost solid bar in the Kernel Execution column shows that the DPU is the limiting performance factor for this pipeline. A magnified view from the middle of the pipeline shows more detail:



The relative height of stream loads and stores in this diagram reflects their efficiency. Each load and store moves the same amount of data to or from the LRF, but the strided load **spi_load_stride** is much faster than the indexed store **spi_store_index**. For the error diffusion pipeline, kernel execution time still limits the overall performance, but just barely. In a pipeline with a slightly faster kernel, indexed stream stores might be the limiting factor instead, and the program designer might choose a different data flow in order to avoid indexed stores.

7.3 Kernel performance

The kernel visualization below shows the SAD kernel **k_sad_16x16**:



The block starting at cycle 16 is the pipelined inner loop; it computes SAD for one 16x16 block. For each block, a 16x16 SAD requires 256 input pixels and 256 reference pixels, so the kernel must read 128 words (512 8-bit pixels). The DPU can read two words per cycle, so it requires 64 cycles to read the required input data; the reads are in columns S0 and S1 above, forming almost solid columns. Column SI shows the single write of the result word. The pipelined inner block performs the SAD computation in the ALUs almost entirely in parallel with the reads. The scheduled length of the pipelined block is 68 cycles, with the reads alone requiring 64 cycles.

On an SP-16 with a 500 MHz DPU, the minimum time required for the kernel just to read its input and reference data is 8 ns/block: 2 ns per cycle * .5 cycles per word * 256 pixels per block * .25 words per pixel * 2 [input + reference data] equals 128 ns for 16 blocks (one block per lane). The achieved SAD kernel performance is about 10 ns/block (.04 ns/pixel), so the kernel is operating very efficiently. Software pipelining overhead for priming and draining accounts for most of the difference between the inner block cycle count and the total kernel time.

For additional information or product support, please contact:
Stream Processors, Inc., 455 DeGuigne Drive, Sunnyvale, CA 94085-3890, USA
Telephone: +1.408.616.3338 • FAX: +1.408.616.3337 • Email: info@streamprocessors.com • Web: www.streamprocessors.com