# General-Purpose Computation Using Graphics Hardware for Fast HDR Image Processing

Dawid Pająk*

Institute of Computer Graphics
Technical University of Szczecin
Szczecin / Poland

## Abstract

This paper presents a new approach to hardware aided image processing and analysis, primarily focused on HDR imaging. In order to achieve interactive frame rates and great processing speeds we propose a library model which is able to utilize efficiently most powerful of the underlying vector hardware. The resulting library architecture was implemented and tested on GPUs and SIMD capable multicore CPUs.

**Keywords:** GPGPU, GLSL programming, SSE, SIMD, optimization, HDR image processing, tone reproduction

## 1 Introduction

In recent years, graphics applications and algorithms have increased their computational complexity greatly. The problem of effective image processing arises more when applied to HDR (high dynamic range) images. Comparing to regular 8-bit per channel LDR images, HDR images store color and luminance data with much greater precision. Greater precision implies more data being processed per pixel. Also, as the HDR image reflects real world luminance values, displaying it on a LDR (low dynamic range) device is not a trivial problem. LDR devices like CRT monitors or LCD panels are able to represent two orders of absolute dynamic range, while HDR images often span over ten orders of magnitude. There is a class of software (games, image editors, face/human recognition programs etc.) which require HDR image processing in real-time or at least interactive rates. Obviously computational power of today's computers increased to meet the demands of the industry. A low cost mass market PC is equipped with multicore SIMD (SSE, Altivec) capable CPU and fully programmable graphics accelerator (GPU). Additionally PCs can be augmented with dedicated boards used for game physics (Ageia PhysX) or general calculation acceleration (IBM/Sony/Toshiba Cell processor). All these advances in technology put a heavy burden on a programmer, a lot of platform dependent know-how is required to implement algorithms effectively on certain hardware. There are some examples [3] [5] [6] of GPU being used effectively in HDR image processing, tone mapping [12] in particular. Tone mapping is an operation which converts HDR luminance values to LDR range - [0,1] usually. Some of the tone mapping algorithms are considered to be among the most computationally expensive operations that can be performed on HDR images. A more general approach to image processing using vector parallel hardware can be found in Cornwall et al. [2] and some commercial products like Matlab, Nvidia CUDA, Intel Performance Primitives library.

In this paper we present an efficient and flexible way of processing and analyzing HDR images by the best hardware available (GPU or SIMD CPUs). A result of our work is a system independent library equipped with a set of mathematical functions each characterized by different kind of operation and type of performed computations (local, global or accumulative). Programmer does not need to know the API/architecture details of execution environment nor he needs to know the execution hardware itself.

The paper is organized as follows. In Section 2 we explain the basic ideas behind the library architecture and API constraints that need to be set in order to efficiently implement the library abstract objects on vector enabled parallel hardware. In Section 3 we discuss all the implementation details on target architectures. We present the results of library working on photographic tone mapping operator in Section 4. Finally we conclude our paper in Section 5.

## 2 Hardware aided HDR image processing library

A library which is able to work fast on multiple hardware platforms requires a special approach in the early design stage. Public API structure is a careful trade off between performance, flexibility and portability on vector performers. HDR image processing and analysis is basically very similar to general purpose floating point data processing, so what we need is a library that could process two dimensional float arrays (image data) in a parallel manner.

*dpajak@wi.ps.pl

## 2.1 Basic library features

Listed below are the basic design requirements that need to be met in order to allow implementation of vast majority of image processing and analysis algorithms.

- Ability to perform simple (e.g. addition, multiplication, reciprocal) and complex (e.g. logarithm, power) math operations on float arrays (in atomic manner)
- High performance HDR image processing oriented functions (e.g. log average of image luminance)
- Ability to perform conditional operations ($if\ x > 1\ then\ y = z$)
- Ability to perform accumulation/gather operations (e.g. min/max of elements in array, sum of elements, convolution)
- Ability to group many simple operations in a complex one (having the general input, output and intermediate arrays provided we build up a new operation which the library engine should execute much faster comparing to combined time of simple operations)
- Clear and straightforward API
- Ability to detect at run-time the best hardware component to execute operations on (in most cases the GPU will be the best performer)

## 2.2 Built-in abstract types

Having the GPU/CPU architecture limitations in mind we came up with an idea (see Figure 1) of 4 built-in abstract types that would let us:

- Store and manage floating point streams (arrays)
- Execute one/many operations on arrays in an effective way
- Hardware idependent architecture which performs the computations

### 2.2.1 Array

This type is a container for our data. Functionally *Array* is similar to a 2D float array with dimensions set up during the construction.

### 2.2.2 Queue

*Queue* is the main acceleration structure of the library. By using it we can group many simple operations into one complex command and execute it efficiently one or more times. However there are some limitations on how *Queue* instance should be constructed. In order to exploit vector processing and parallel computing capabilities of hardware we have set some constraints on queued operations:

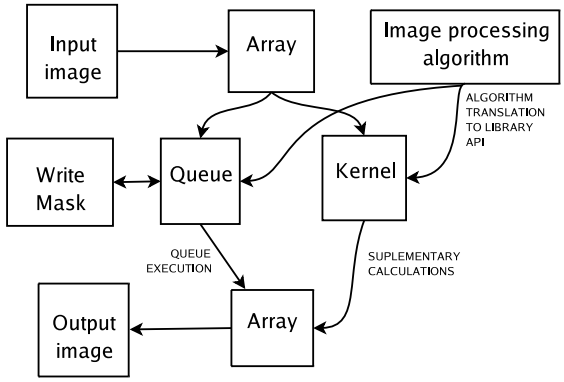


Figure 1: Overall model of computation using proposed library architecture. Image data is stored inside *Array* objects. Image processing algorithm is implemented by *Queue* and/or *Kernel* computation engine

- All input *Array* objects should have same dimensions
- *Queue* instance has one output *Array* which queued operations may write to
- Operations are allowed to write the result either to a temporary variable (available only for queued operations internally) or to *Queue* output
- Operations cannot read from *Queue* output
- Some operations cannot read from intermediate variables (gathering operations e.g. 1D convolution - temporary variables are local scalar types)
- No accumulation (e.g. sum of all elements in certain input *Array*) operations are allowed
- Once the operation with certain arguments is added to *Queue* it cannot be removed or modified in any way

With these limitations set we can be sure that all the calculations for each element of *Array* output are independent, thus can be directly mapped on SIMD or parallel hardware. Also, because the *Array* instances used by *Queue* can be read-only or write-only the synchronization in gathering operations is unnecessary. The increased performance of *Queue* execution comes from greatly increased locality of calculations (e.g. we use intermediate scalar variables and perform many operations on single element of input array) and reduced memory bandwidth usage. The *Queue* is able to perform writes to output conditionally e.g. if some condition is met then the result of the calculation will not be written to *Queue* output *Array* (so called "break if").

### 2.2.3 WriteMask

The *WriteMask* type is a help structure that can be seen as a bit array with the same dimensions as *Queue* output. During the *Queue* execution, element at index *i* will have

its value computed only if *Queue* current $WriteMask$ at index $i$ is set to $false$. Also writing to *Queue* output *Array* at index $i$ sets the $WriteMask$ at index $i$ to $true$ (only if $WriteMask$ is writable by *Queue* object). This way we are able to compute *Queue* output conditionally between subsequent execution method calls. In other words, multi-pass conditional computation is possible. It's worth noticing that one $WriteMask$ object can be shared among multiple *Queue* instances.

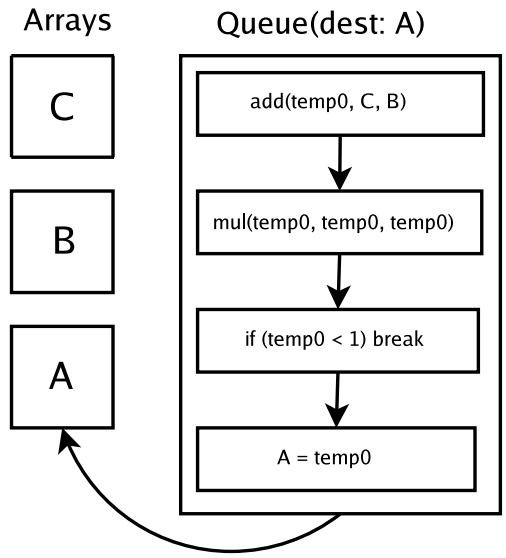


Figure 2: Example of a *Queue* instance with conditional instruction execution. The *Queue* type can be imagined as a container for a set of operations. Executing a *Queue* means executing serially all the operations inside, beginning from the first one added.

The following pseudo code describes the overall algorithm of how *Queue* from Figure 2 works during the execution (loop body is performed in parallel/vector manner).

```
int n = width * height;
for (int i = 0; i < n; i++) {
  if (writeMask != 0 && writeMask[i])
  {
    continue;
  }
  float temp0 = C[i] + B[i];
  temp0 = temp0 * temp0;
  if (temp0 < 1.0f) continue;
  A[i] = temp0;
  if (writeMask != 0 && writeMaskWritable)
  {
    writeMask[i] = 1;
  }
}
```

#### 2.2.4 Kernel

The *Kernel* class is able to invoke accumulating functions and most of *Queue* functions. It is not capable of grouping commands or executing conditional computation, the operations are complete just after method call. Also the *Kernel* class is a manager (Factory) of *Array* and *Queue* abstracts.

## 3 Implementation

In this section we present the implementation details of library base types for two target architectures: GPU and SIMD capable CPUs. CPU implementation was developed as a proof-of-concept, reference implementation.

### 3.1 Mapping the library computation model on SIMD CPU

The idea behind SIMD computation is to perform single operation on many data elements at the same time. Many current compilers are equipped with features like intrinsic and auto vectorization to help the programmer utilize the vector unit of modern CPU. However, using automated vectorization feature will result in much smaller performance gains than using fine tuned hand-written intrinsic code for SSE unit.

#### 3.1.1 Math operations on SIMD architectures

When we assume that the *Array* value at index $i$ is independent of any other value in the same *Array* the transformation between the SISD and SIMD code becomes simple.

$$y = \frac{2 * a}{b - 3} \tag{1}$$

The following code segment shows a SIMD implementation of an Equation 1.

```
#include <xmmintrin.h>
const __m128 a2 = _mm_set1_ps(2.0f);
const __m128 a3 = _mm_set1_ps(3.0f);

__m128 fun(__m128 a, __m128 b) {
  __m128 num = _mm_mul_ps(a2, a);
  __m128 den = _mm_sub_ps(b, a3);
  return _mm_div_ps(num, den);
}
```

Efficient form of this code allows us to accelerate the execution of the function by theoretical factor of four. HDR image processing requires some complex math operations e.g. $log_2(x)$, $2^x$ to be able to execute on vector data. As we put more pressure on performance than precision, an approximation of these functions will fit our needs completely.

To compute $log_2(x)$ we use some specific features of single precision floating-point number representation. As defined in IEEE754 specification single precision number is described by equation: $(-1)^s 2^e m$ where $s$ is a sign bit, $e$ is 8-bit exponent and $m$ is 24-bit normalized mantissa.

$$log_2(x) = log_2(2^e m) = e + log_2(m), x > 0. \quad (2)$$

We calculate the $log_2(x)$ value by extracting the exponent from the number representation and adding it to the approximation of $log_2(x)$ function in [1, 2] interval (the value of extracted mantissa). In our implementation we use Chebyshev mini-max fifth degree polynomial to approximate the function. This results in small relative error ($10^{-6}$) and overall good quality of the function values.

We use a similar approach with $2^x$ evaluation.

$$2^x = 2^{R(x)+F(x)} = 2^{R(x)} * 2^{F(x)} \quad (3)$$

$$2^{R(x)} * 2^{F(x)} = 2^{R(x)} * (2^e m) = 2^{e+R(x)} m \quad (4)$$

, where $R(x)$ is truncated integer part of $x$ and $F(x)$ is fractional part of the input value. To compute $2^x$ value first we approximate the $2^{F(x)}$ (input range [0, 1]) using polynomial and add the $R(x)$ value directly to the exponent field of approximated value. Both techniques used (vector bit manipulation and vector polynomial value evaluation) can be implemented very efficiently on SIMD architectures.

#### 3.1.2 Implementation of *Array*, *WriteMask*, *Kernel* classes

The mapping of *Array* and *WriteMask* abstract types is quite straightforward. Both are represented as single precision floating point arrays stored in main memory of computer.

#### 3.1.3 Implementation of *Queue* class

To make use of CPU cache coherency we put all the vector operations into a queue and let them run on small data chunks at a time. Also when multicore/SMP system is detected then the work on all chunks is split across available CPUs in interleaved manner (see Figure 3). To avoid unnecessary branching[1] we use SIMD compare instructions to get comparison result(bit mask) and do a mask dependent write to *Queue* output. The downside of this approach is that we compute all the queued calculations on the element even if it is already discarded by *WriteMask* or condition mask value. However it proved to be much more profitable it terms of performance than doing separate branching for each element.

### 3.2 Mapping the library computation model on GPU

Within past few years GPUs have emerged as a powerful computational platforms. Equipped with extremely powerful memory system and many programmable pipelines

[1] modern processors suffer great performance penalties when branching is used, especially when branch prediction was incorrect

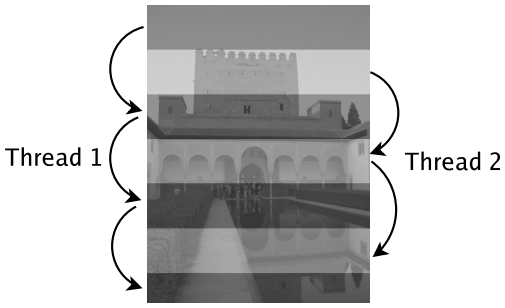


Figure 3: *Queue* execution on multicore CPU. Data is divided into small chunks. Each chunk is assigned to execution *Thread*. On this example *Thread* 1 is processing chunks 0, 2, 4, 6 and *Thread* 2 the reminder, chunks 1, 3 and 5.

the GPUs have added another level of parallelism into computation area. Effectively utilizing the wealth of this computational resource requires a different programming model to be used (stream computation, stream programming model). Below we show a simplified algorithm of a function calculation for vector input on graphics hardware.

1. CPU uploads input data to GPU in the form of floating point 2D texture
2. CPU uploads to GPU an appropriate fragment program (equivalent to *Queue* execution loop body) which will be executed for each pixel of input/output texture
3. GPU does the calculation, which is equal to drawing a textured rectangle into destination buffer/texture
4. CPU downloads output texture with calculated results back to system memory.

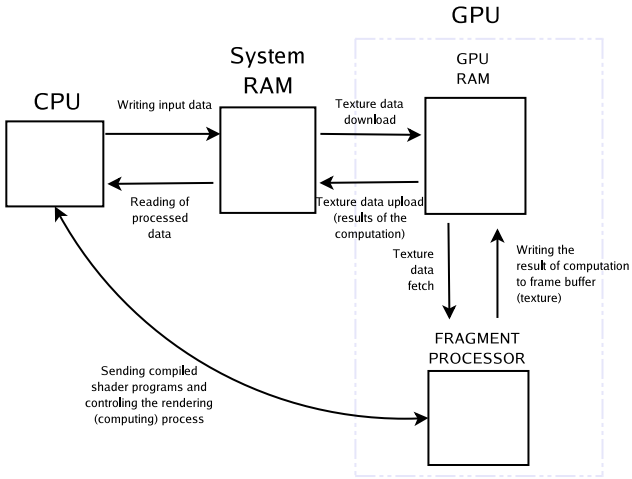


Figure 4: CPU-GPU interaction in computation.

In order to perform general purpose computations on GPUs we need to map our library base types to their GPU counterparts.

#### 3.2.1 Implementation of *Array* class

*Array* type is mapped directly to 2D one component floating point texture. The textures are used as source and destination for the rendering (off-screen rendering to texture). The data is stored in two locations: locally (system RAM) and remotely (GPU RAM). To save bandwidth and increase performance synchronization is done only when necessary, meaning when GPU or programmer (CPU) requests the access to data and their copy is obsolete (because of the changes made in the other copy).

#### 3.2.2 Implementation of *Queue* class

At the beginning of *Queue* execution the group of queued (added) operations is transformed to GLSL fragment program source. After the shader is compiled (which might take some of the CPU time, however it is a one time action) the drawing to *Queue* destination *Array* (texture) is performed. Example *Queue* and GLSL shader:

```
Array *dest = kernel->createArray(10, 10);
Array *a = dest->clone();
Array *b = dest->clone();
Queue *q = kernel->createQueue(dest);
q->add(Q_OUTPUT, a, b);
q->log(Q_OUTPUT, Q_OUTPUT);
q->sub(Q_OUTPUT, Q_OUTPUT, a);
q->execute();
```

GLSL:

```
uniform float param0;
uniform samplerRect array0;
uniform samplerRect array1;

void main(void) {
  float tx0 = textureRect(array0,
    gl_TexCoord[0].st).r;
  float tx1 = textureRect(array1,
    gl_TexCoord[0].st).r;
  gl_FragColor = log(tx0 + tx1) - tx0;
}
```

#### 3.2.3 Implementation of *WriteMask* class

*WriteMask* class is realized as a floating point array attachable to FBO (Frame Buffer Object) as a depth buffer. When *Queue* has *WriteMask* set, the rendering is done with depth buffer set and testing turned on (pixels which do not pass the test are not rendered to destination texture).

#### 3.2.4 Implementation of *Kernel* class

The GPU implementation of *Kernel* class is responsible for new *Queue* and *Array* allocation as well as for invocation of simple math methods (actual implementation internally creates one element *Queue* with modifiable arguments) and gathering methods. The fragment program is not capable of writing to global variables (shared across all the shader units), so evaluating sum of an array is not a trivial task. It is accomplished by applying a fragment program which adds up four adjacent pixels and writes the output to destination texture (which is 2 times smaller in each dimension). We repeat the process (switching output with input at the end of each iteration) until we get $1x1$ output texture. The complexity of this algorithm is $O(nlog_2(n))$ comparing to $O(n)$ when calculating the sum on the CPU. Also it has tendencies to be memory bandwidth consuming. Similar algorithm can be applied when computing minimum/maximum value of *Array*.

## 4 Results

In the following section we demonstrate our library at work. The objective is to prove that API combined with appropriate hardware gives great speedups in HDR image processing and the library itself is portable and efficient on multiple platforms. Our example algorithm is the photographic tone mapping operator [12] which we describe in Section 4.1. We have tested five HDR images (see Figure 5) each characterized by different resolution, luminance and contrasts values. The tests were mostly focused on performance and quality of calculations.

In Section 4.2 we include details on the implementation of the local variance of the operator. Section 4.4 with test results is preceded by detailed specification of the test environment (Section 4.3).

### 4.1 Photographic tone reproduction operator

The process of photographic tone mapping begins with a linear scaling by key value which is equal to setting exposure in a camera. The key of a scene is an indicator of how light or dark the overall impression of the scene is. A good approximation of scene key is log average of scene luminance (Equation 5 and 6). A typical value of $\alpha$ is 0.18.

$$L_{avg} = e^{\frac{1}{n}\sum_{i=1}^{n}\log(0.00001+L_{wi})} \tag{5}$$

$$L_m(x,y) = \frac{\alpha}{L_{avg}} L_w(x,y) \tag{6}$$

Transfer function from Equation 7 predominantly compresses high luminance values and preserves low ones. The function has asymptote at 1, but usually mapped luminance is not infinitely big to reach this value. Because of that a modified form of Equation 7 exists which introduces $L_{white}$ parameter, a maximum luminance value (after pre-scaling) mapped to white (Equation 8). This is analogous to burning process.

$$L_d(x,y) = \frac{L_m(x,y)}{1+L_m(x,y)} \tag{7}$$

$$L_d(x,y) = \frac{L_m(x,y)(1+\frac{L_m(x,y)}{L_{white}^2})}{1+L_m(x,y)} \tag{8}$$

The photographic tone mapping operator also has a local variant which uses a different luminance compression function for each pixel (global variant uses the same function for whole image). This modification was introduced in order to let the algorithm imitate photographic dodging and burning. It means that each pixel receives a different exposure value estimated on the reminder of the image bounded by sharp contrast. This is accomplished by finding the largest pixel neighborhood without significant contrasts (Equation 10 - difference between gaussian blurred pixels is close to zero).

$$L_s^{blur}(x,y) = L_m(x,y) * R_s(x,y) \tag{9}$$

$R_s$ is a gaussian kernel.

$$V_s(x,y) = \frac{L_s^{blur}(x,y) - L_{s+1}^{blur}(x,y)}{2^{\phi}\alpha/s^2 + L_s^{blur}(x,y)} \tag{10}$$

The denominator of Equation 10 ensures that $V_s$ is independent of absolute luminance values and kernel size. A $\phi$ parameter might be viewed as sharpening parameter (setting it to 8 gives good results).

To find the largest area with low contrast we seek the largest scale value $s_{max}$ for which the Equation 11 remains true.

$$s_{max} : |V_{s_{max}}(x,y)| < \varepsilon \tag{11}$$

The local operator final luminance value is given by Equation 12.

$$L_d(x,y) = \frac{L_m(x,y)}{1 + L_{s_{max}}^{blur}(x,y)} \tag{12}$$

We have implemented successfully both variants of the operator using our library.

## 4.2 Local tone mapping operator implementation

Global variant of photographic operator is simple and straightforward to implement. The local version performs condition check for each element and stops when it is false ($V_s$ exceeds threshold value). We implement similar behavior using *WriteMask* objects. A *Queue* which computes $V_s$ value writes $L_s^{blur}$ to output only if $V_s$ is greater than threshold ($\varepsilon$). After this no further writes to output are performed as *WriteMask* at this position is true. One execution of this *Queue* is analogous to checking if a condition from Equation 11 is true for particular $s$. We need to execute the *Queue* $n$-times, where $n$ is the number of different gaussian kernel sizes (we start at the smallest 1-pixel neighborhood). The shortcoming of this method is that we have to calculate all the blur levels (we use eight different gaussian masks, varying in size from 1 to 43) even if all the pixels have their final $V_s$ set. Another side effect of this approach is that mapping operation is insensitive to pixel input neighborhood contrast values.

## 4.3 Testing environment

Our test system was based on FC5 Linux AMD64 distribution. Programs were compiled as native 64-bit software using GCC 4.1.1. Hardware configuration:

- AMD Athlon X2 3800+ (2.0Ghz clocked)
- 2048MB DDR RAM (128-bit bus, 400Mhz clock)
- GPU 1 - NVidia Geforce 7600GT 256MB RAM[2]
- GPU 2 - NVidia Geforce 8800GTS 640MB RAM[3]

Both GPUs used NVidia driver 1.0-9746 for x86-64 systems. One of the issues we had to deal with was timing computation when using GPU. The driver issues rendering calls fully asynchronously so to measure time properly we had to force drawing by downloading the destination texture. The measured download speed is about 780MB/s on our system. As we want to make CPU/GPU tests reliable, all the results in this section are without texture data download/upload times counted. Our assumption was that after tone mapping operation image luminance stays locally at GPU memory for further processing/displaying.

The CPU implementation by default uses all the available execution units (cores) during the computation. We used *pfstools* [11] package as an example of classic approach to HDR image processing and analysis.

## 4.4 Tests results

Figure 5: Images used in test. Image names (in top-bottom, left-to-right fashion): TuolumneTree, BurnedRedwood, BoyScoutTrail6, BristolBridge, memorial.

[2] 128-bit bus, 1400Mhz memory clock, 650Mhz core clock

[3] 320-bit bus, 1600Mhz memory clock, 500Mhz core clock, 1200Mhz for shader units

| Image | Resolution | Time [ms] | | | |
|---|---|---|---|---|---|
| | | PFS | CPU | 7600 | 8800 |
| memorial | 512x768 | 107 | 4.8 | $< 1$ | $< 1$ |
| BoyScoutTrail6 | 1000x1504 | 421 | 15 | 4.1 | $< 1$ |
| BristolBridge | 2048x1536 | 947 | 30 | 12 | 6.2 |
| TuolumneTree | 2272x1704 | 1222 | 36 | 15 | 8.5 |
| BurnedRedwood | 2000x3008 | 2641 | 59 | 36 | 11 |

Table 1: Global operator benchmark

| Resolution | Speed-up | | |
|---|---|---|---|
| | CPU | 7600GT | 8800GTS |
| 512x768 | 22.2x | *NA* | *NA* |
| 1000x1504 | 28x | 102x | *NA* |
| 2048x1536 | 31.5x | 78x | 152x |
| 2272x1704 | 33.9x | 81x | 143x |
| 2000x3008 | 44x | 73x | 240x |

Table 2: Increase in computation performance of global tone mapping compared to pfs tools.

Tests showed huge speedups when computing global TMO (Tone Mapping Operator) (Table 2). Both GPU and CPU were able to tone-map the biggest image at interactive speeds. Although even a powerful dual-core CPU working with SIMD unit is not able to compete in terms of processing power and memory bandwidth with a mid-end GPU like 7600GT.

| Image | Resolution | Time [ms] | | | |
|---|---|---|---|---|---|
| | | PFS | CPU | 7600 | 8800 |
| memorial | 512x768 | 430 | 279 | 27 | 17 |
| BoyScoutTrail6 | 1000x1504 | 1563 | 875 | 106 | 33 |
| BristolBridge | 2048x1536 | 5912 | 2612 | 221 | 82 |
| TuolumneTree | 2272x1704 | 7782 | 2152 | 259 | 98 |
| BurnedRedwood | 2000x3008 | 8226 | 3273 | 376 | 144 |

Table 3: Local operator benchmark

The computational power of GPUs are even more exposed when it comes to local variance of TMO (Table 4). CPU implementation suffers from low memory bandwidth and is not able to perform image blurs at reasonable speeds (see Section 5). The fragment programs used for local TMO were fairly simple[4] so mapping is purely memory bound process - which can be observed by comparing memory bandwidths of tested GPUs with the results (ratio is close to 3:1 in favour of 8800GTS board).

The speedup of local tone mapping for largest image is smaller than predicted (this applies to GPU and CPU performer). The cause of this is the image content itself. It has some high contrast, noise areas in which the $pfstools$ TMO usually applies only the smallest kernel. It appears that our performers slow down, but in fact it is the $pfstools$ TMO which increased its processing speed in this particular case.

| Resolution | Speed-up | | |
|---|---|---|---|
| | CPU | 7600GT | 8800GTS |
| 512x768 | 1.54x | 15.9x | 25.2x |
| 1000x1504 | 1.79x | 14.7x | 47.3x |
| 2048x1536 | 2.26x | 26.7x | 72.1x |
| 2272x1704 | 3.62x | 30x | 79.4x |
| 2000x3008 | 2.51x | 21.8x | 57.1x |

Table 4: Increase in computation performance of local tone mapping compared to $pfstools$.

| Resolution | Time [ms] | | |
|---|---|---|---|
| | CPU | 7600GT | 8800GTS |
| 512x768 | 1.5 | $< 1$ | $< 1$ |
| 1000x1504 | 6.2 | 3.6 | $< 1$ |
| 2048x1536 | 12 | 6.3 | $< 1$ |
| 2272x1704 | 15 | 8.1 | 2.3 |
| 2000x3008 | 24 | 11 | 2.8 |

Table 5: The calculation time of log average of image luminance.

The computation speed of log average on GPU is noticeably faster, however as the GPU uses bandwidth consuming algorithm (see Section 3.2.4) for data accumulation the calculation scales much better on 8800GTS. It is worth noticing that CPU (single core) is calculating the log average at the speed of 960MB/s.

The resulting tone mapped images from both performers were compared using VDP [8] algorithm. Image from CPU performer was used as reference[5]. There were no visible differences which was expected as calculations done on GPU were also in single precision floating point format. The images were not exact though. The way the log function is approximated (when calculating the luminance log average) is different on both GPU and CPU implementations. The relative error between approximations is close to $10^{-5}$. Images with high luminance values might aggregate this error into further processing.

# 5 Conclusions and future work

In order to solve HDR image processing problems efficiently we have presented and implemented a framework that provides hardware processing power. Unlike other hardware aided image processing software, our library is flexible enough to execute any kind of algorithm and run it on many vector/parallel architectures. Both example implementations showed great speedups (quite often two orders of magnitude).

[4] in terms of floating point operations done per memory word read

[5] as the FPU unit is fully IEEE754 compatible

The author of the implementation is aware that a lot of development is still needed. Here is a list of some improvements that could increase the performance of certain algorithms even more:

- CPU: Rewrite convolution code to make use of SSE unit - the test program used unoptimized convolution code, the result was that 80% of the tone mapping time was spent on blurring the images
- GPU/CPU: Implement automatic Gauss pyramid building - another optimization opportunity which HDR processing (tone mapping in particular) would benefit from
- GPU/CPU: Implement different kinds of branching in *Queue* code. Now only "break if" type is available
- Make specialized ports for current high-end graphics hardware like G80 - in our opinion porting the computation code of library to NVidia CUDA platform would give another noticeable performance gain

## 6 Acknowledgments

Thanks to Radoslaw Mantiuk for providing the initial library concept and lots of help during writing process of this article.

The research work, which results are presented in this paper, was sponsored by Polish Ministry of Science and Higher Education (years 2006-2007).

## References

[1] R. Bogart, F. Kainz, and D. Hess. OpenEXR image file format. In *ACM SIGGRAPH 2003, Sketches & Applications*, 2003.

[2] Jay L.T. Cornwall, Olav Beckmann, and Paul H.J. Kelly. Accelerating a c++ image processing library with a gpu. 2006.

[3] Frédéric Drago, Karol Myszkowski, Thomas Annen, and Norishige Chiba. Adaptive logarithmic mapping for displaying high contrast scenes. *Computer Graphics Forum, proceedings of Eurographics 2003*, 22(3):419–426, 2003.

[4] F. Durand and J. Dorsey. Interactive tone mapping. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 219–230, 2000.

[5] N. Goodnight, R. Wang, C. Woolley, and G. Humphreys. Interactive time-dependent tone mapping using programmable graphics hardware. In *Rendering Techniques 2003: 14th Eurographics Symposium on Rendering*, pages 26–37, 2003.

[6] Grzegorz Krawczyk, Karol Myszkowski, and Hand-Peter Seidel. Perceptual effects in real-time tone mapping. 2005.

[7] Rafal Mantiuk, Grzegorz Krawczyk, and Radoslaw Mantiuk. High dynamic range imaging pipeline: Merging computer graphics, physics, photography and visual perception. In *In Spring Conference on Computer Graphics 2006 Posters and Conference Materials*, pages 37–40, 2006.

[8] Rafal Mantiuk, Karol Myszkowski, and Hans-Peter Seidel. Visible difference predicator for high dynamic range images. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pages 2763–2769, 2004.

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[10] S. Pattanaik, J.E. Tumblin, H Yee, and D.P. Greenberg. Time-dependent visual adaptation for realistic image display. In *Proceedings of ACM SIGGRAPH 2000*, pages 47–54, 2000.

[11] Radoslaw Mantiuk Rafal Mantiuk, Grzegorz Krawczyk. High dynamic range imaging pipeline: Perception-motivated representation of visual content. In *To be published in Proc. of Human Vision and Electronic Imaging XII, IS&T/SPIE's Annual Symposium on Electronic Imaging*, SPIE Proceedings Series, San Jose, California USA, January 2007.

[12] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. on Graph.*, 21(3):267–276, 2002.

[13] G. Ward Larson. LogLuv encoding for full-gamut, high-dynamic range images. *Journal of Graphics Tools*, 3(1):815–30, 1998.