Graphic processors to speed-up simulations for the design of high performance solar receptors

Caroline Collange, Marc Daumas, David Defour

HAL Id: hal-00135126

https://hal.science/hal-00135126v3

Submitted on 29 Aug 2007

Graphic processors to speed-up simulations
for the design of high performance solar receptors[*]

Sylvain Collange[a], Marc Daumas[a,b] and David Defour[a]

[a] ELIAUS, UPVD — [b] LIRMM, CNRS, UM2

52 avenue Paul Alduy — Perpignan 66860 — France

firstname.lastname@univ-perp.fr

Abstract

Graphics Processing Units (GPUs) are now powerful and flexible systems adapted and used for other purposes than graphics calculations (General Purpose computation on GPU — GPGPU). We present here a prototype to be integrated into simulation codes that estimate temperature, velocity and pressure to design next generations of solar receptors. Such codes will delegate to our contribution on GPUs the computation of heat transfers due to radiations. We use Monte-Carlo line-by-line ray-tracing through finite volumes. This means data-parallel arithmetic transformations on large data structures. Our prototype is inspired on the source code of GPUBench. Our performances on two recent graphics cards (Nvidia 7800GTX and ATI RX1800XL) show some speed-up higher than 400 compared to CPU implementations leaving most of CPU computing resources available. As there were some questions pending about the accuracy of the operators implemented in GPUs, we start this report with a survey and some contributed tests on the various floating point units available on GPUs.

1. Introduction

Graphics Processing Units (GPU) offer computing resources higher than the ones available on processors [8]. With the delivery of the latest generations of GPUs, they can be used for general processing (GPGPU, www.gpgpu.org) [7] and become application specific co-processors for regular and heavily data-parallel processing.

We strongly believe that the development of GPGPU will necessary pass through the identification of key applications that will benefit from the various hardwired functionalities

available on GPU. We describe the architecture of GPUs and properties of the implemented floating point arithmetic discovered with our tests in Section 2. Section 3 presents the accurate estimation of radiative heat transfers due to the filtering of incidental lines and the generation of heat induced lines. We elaborate on the performances of our prototype and we present perspectives in Section 4. We do not account for diffusion in this preliminary study as our medium does not contain particles.

To the best of our knowledge, there is no prior art in the implementation of the tasks reported here on GPUs. Monte-Carlo ray-tracing and line-by-line analysis are routinely performed on CPUs for simulations of radiative heat transfers though these tasks usually saturate CPUs leaving no opportunity to the coupling of convective and radiative phenomena on real applications. Other applications heavily rely on elaborate physical models [5, 6]. Most simulations are currently performed for simple reference cases (isothermal gas column at equilibrium). The description of gas spectrum is generally simplified in calculation with engineering interests leading to errors in the range of 5-15% or more.

2. Graphics Processing Units (GPU)

GPUs handle mostly geometrical objects and pixels. Images are created by applying geometrical transformations to vertices and by splitting objects into pixels. Calculations are carried out by various stages composing the graphics pipeline presented in Figure 1. Actual pipelines of existing GPUs differ slightly. Manufacturers move, share, duplicate or add resources depending on boards and processors. The figure shows the various stages on the example of a triangle. In this example, vertex shaders treat 3 vertices whereas pixel shaders treat 17 pixels. For most geometrical objects, the number of pixels is larger than the number of vertices. Modern architectures contain more pixel shaders than vertex shaders. The current ratio is commonly 24 against 8.

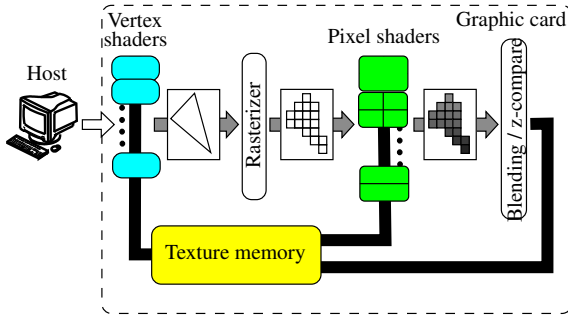The host sends vertices to position primitive geometrical
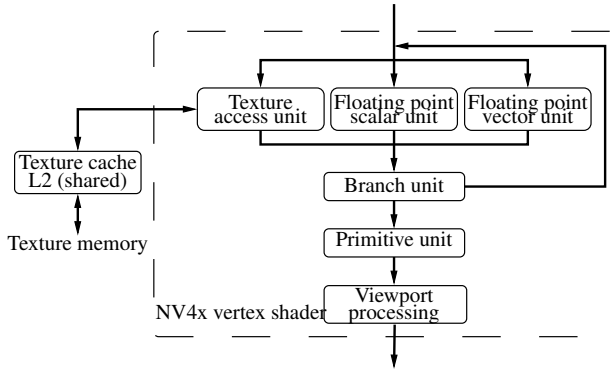
377

Figure 1. Model of the graphics pipeline.

Figure 2. Vertex shader of the Nvidia 7800GTX.

Figure 3. Pixel shader of the Nvidia 7800GTX.

objects (points, lines, polygons). Objects are transformed (rotation, translation, illumination...) and assembled to create more elaborate objects. These operations are carried out by *vertex shaders*.

At each cycle, each vertex shader (see Figure 2 adapted from [8]) is able to initiate a *Multiply and Add* (MAD) operation on 4 pieces of data in the vector unit and a *special* operation in the scalar unit. The implemented *special* operations are exponential functions (exp, log), trigonometric functions (sin, cos) and reciprocal functions ($1/x$ and $1/\sqrt{x}$). Since hardware support of DirectX 9.0, vertex shaders are able to address texture memory through a dedicated unit.

The first floating point unit of each pixel shader (see Figure 3 adapted from [8]) carries out 4 MADs or an access to texture via a dedicated unit. The result is then sent to the second floating unit which carries out 4 MADs. In the case of Nvidia 7800 GTX, each pixel shader includes a level 1 texture cache.

Table 1 presents the floating point formats implemented on GPUs and CPUs. A number is represented by its mantissa, its exponent e and its sign bit s. The first bit of the mantissa (left of the fraction point) can be set to 1 unless the number to be represented is very small. The re-
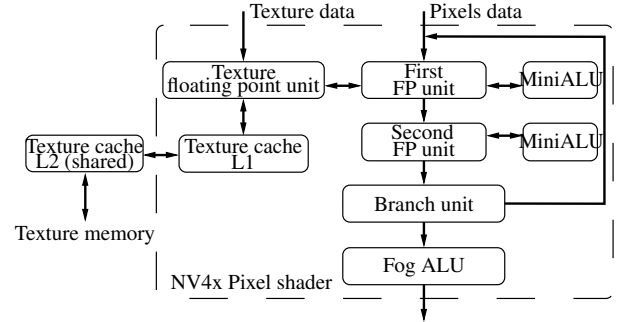
maining bits form the fraction f. A normal representation stores $(-1)^s \cdot 1.f \cdot 2^e$ and a subnormal one stores $(-1)^s \cdot 0.f \cdot 2^{e_{\min}}$ where $e_{\min}$ is the minimum allowed exponent. Single precision (32 bit) became available on GPUs with Shader Model 3.0. Manufacturers of GPUs do not claim full compatibility with ANSI-IEEE standard on floating point arithmetic.

Before porting our application to GPUs, we surveyed two pieces of software testing performances and implementations of floating point arithmetic on Nvidia 7800GTX and ATI RX1800XL [1, 4]. Tests have drawn the first following conclusions:

- Additions and multiplications are truncated.
- Subtractions seem to benefit from a guard bit with Nvidia but not with ATI.
- Multiplications attain faithful rounding.
- Errors on divisions indicate that divisions are based on multiplications by approximations of the reciprocal.

We wrote additional test vectors summarized in Table 2 where $\oplus, \ominus, \otimes$ are the addition, subtraction and multiplication operators implemented on GPU. $U[a, b]$ are uniformly distributed random variables on $[a, b)$. Random tests are performed on 2^{23} inputs, other tests are exhaustive.

We used OpenGL primitives and stored data in textures using *Frame Buffer Object* and respectively *texRECT* and *tex2D* for Nvidia and ATI chips. We set up vertex and pixel shaders for computation with the OpenGL shading language. We ran these tests on Nvidia 7800 GTX with driver ForceWare 81.98 and on ATI RX1800XL with driver Catalyst 6.3.

On some architectures, internal registers store numbers with a precision higher than the one used in memory or with a larger dynamics for the exponents. Sometimes MADs maintain larger accumulators or round results only once, after the addition. Our tests showed that no such things occur on GPUs but they revealed a surprising feature. It appears that the second pixel shader floating point unit on ATI and both units on Nvidia produce a mantissa with one extra bit.

Table 1. Representation format of floating point numbers on GPUs and CPUs.

Reference	Number of bits				Non numerical
	Total	Sign	Exponent	Fraction	values
Nvidia	16	1	5	10	NaN, Inf
	32	1	8	23	(as documented in [2])
ATI	16	1	5	10	Not implemented
	24	1	7	16	
	32	1	8	23	Not documented
ANSI-IEEE 754 [11]	32	1	8	23	NaN, Inf
	64	1	11	52	

Table 2. Arithmetic experimentations and results.

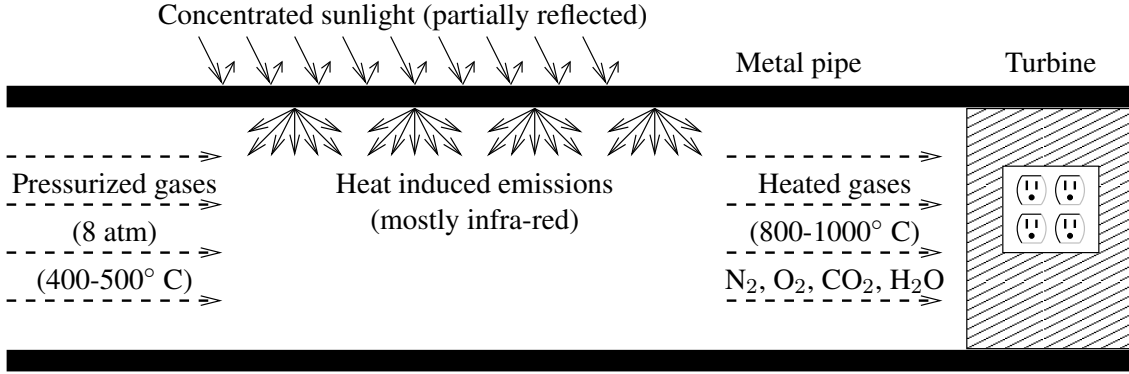| Operations	Shader	Observations
$(M \oplus M) \ominus M$	All	$M = 2^{127}(2 - 2^{-23}) \longrightarrow \infty$
$MAD(x, y, -x \otimes y)$	All	$x \sim U[1, 2) \wedge y \sim U[1, 2) \longrightarrow 0$
$1.5 \ominus 2^{-i}$	ATI-Pixel	$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$ $i = 24 \quad \longrightarrow 1.5 - 2^{-23}$ $25 \leq i \quad \longrightarrow 1.5$
	Nvidia-Pixel	$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$ $24 \leq i \leq 25 \quad \longrightarrow 1.5 - 2^{-23}$ $26 \leq i \quad \longrightarrow 1.5$
	ATI-Vertex	$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$ $24 \leq i \quad \longrightarrow 1.5 - 2^{-23}$
	Nvidia-Vertex	$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$ $i = 24 \quad \longrightarrow 1.5 - 2^{-23}$ $25 \leq i \quad \longrightarrow 1.5$
$(1 \oplus 0.5) \ominus 2^{-i}$	All-Pixel	$1 \leq i \leq 23 \quad \longrightarrow 1.5 - 2^{-i}$ $24 \leq i \leq 25 \quad \longrightarrow 1.5 - 2^{-23}$ $26 \leq i \quad \longrightarrow 1.5$
$(1.5 \ominus 2^{-i}) \ominus 1.5$	ATI-Pixel	$1 \leq i \leq 23 \quad \longrightarrow -2^{-i}$ $i = 24 \quad \longrightarrow -2^{-23}$ $25 \leq i \quad \longrightarrow 0$
	Nvidia-Pixel	$1 \leq i \leq 23 \quad \longrightarrow -2^{-i}$ $24 \leq i \leq 25 \quad \longrightarrow -2^{-23}$ $26 \leq i \quad \longrightarrow 0$
$x \otimes y + (\pm x) \otimes (\mp y)$	All	$x \sim U[1, 2) \wedge y \sim U[1, 2) \longrightarrow 0$
$x \otimes y - (-x) \otimes (-y)$	All	$x \sim U[1, 2) \wedge y \sim U[1, 2) \longrightarrow 0$
$x \otimes y - ((2 \cdot x) \otimes y)/2$	All	$x \sim U[1, 2) \wedge y \sim U[1, 2) \longrightarrow 0$
$(1 + 2^{-23}) \otimes (1 + 2^{-23}i)$	ATI-Pixel	$i \leq (2^{11} - 1) \cdot 2^{12} \longrightarrow$ correct
	Nvidia-Pixel	$i \leq 23 \cdot 2^{17} \quad \longrightarrow$ correct
	ATI-Vertex	$i \leq 2^{23} \quad \longrightarrow$ correct
	Nvidia-Vertex	$i \leq 2^{19} \quad \longrightarrow$ correct
$x \otimes y - x \times y$	ATI-Pixel	$x \in [1, 2) \wedge y \in [1, 2/x) \longrightarrow \{-1.00031 \text{ ulp} \cdots 0.00215 \text{ ulp}\}$ $x \in [1, 2) \wedge y \in [2/x, 2) \longrightarrow \{-1.00013 \text{ ulp} \cdots 0.00085 \text{ ulp}\}$
	Nvidia-Pixel	$x \in [1, 2) \wedge y \in [1, 2/x) \longrightarrow \{-0.51099 \text{ ulp} \cdots 0.64063 \text{ ulp}\}$ $x \in [1, 2) \wedge y \in [2/x, 2) \longrightarrow \{-0.76504 \text{ ulp} \cdots 0.32031 \text{ ulp}\}$
	ATI-Vertex	$x \in [1, 2) \wedge y \in [1, 2/x) \longrightarrow \{-1 \text{ ulp} \cdots 0\}$ $x \in [1, 2) \wedge y \in [2/x, 2) \longrightarrow \{-1 \text{ ulp} \cdots 0\}$
	Nvidia-Vertex	$x \in [1, 2) \wedge y \in [1, 2/x) \longrightarrow \{-0.82449 \text{ ulp} \cdots 0.93750 \text{ ulp}\}$ $x \in [1, 2) \wedge y \in [2/x, 2) \longrightarrow \{-0.91484 \text{ ulp} \cdots 0.46875 \text{ ulp}\}$

Figure 4. The solar receptor as simulated.

This extra bit forces modifications of some multiple precision tools [3] and we conjecture that it is implemented for backward compatibility.

Fast small multipliers usually ignore partial products below a given threshold and add a constant to correct the introduced statistical bias [10]. Results lead us to think that this constant is 2^{-35} on ATI and $41 \cdot 2^{-30}$ on Nvidia. The multipliers accumulate partial products on 9 extra rows on ATI and 6 extra rows on Nvidia. These figures do not include the extra bit left of the mantissa. Other tests indicate that multipliers use radix 2 sign-magnitude logic internally.

Additional tests showed that subnormal numbers are replaced by 0 during transfers even when no arithmetic operation is performed on GPUs meaning that drivers probably perform arithmetic operations on textures. Non numerical quantities are not modified except that sNaN (*signaling NaN*) is changed to qNaN (*quiet NaN*) on ATI.

3. Monte-Carlo line-by-line ray tracing

The experimental setting is presented in Figure 4. This device produces electricity from sunlight concentrated by a large reflector. Concentrated sunlight is used to heat a metal pipe that transfers heat through contact and infra-red radiations. The goal is to transfer as much energy as possible to the turbine. Dynamic and thermal phenomena are intricately interwoven as air temperature raises.

Though our approach is based on finite volumes used for example by Fluent (www.fluent.com) and Trio-U (www-trio-u.cea.fr), this work can also be applied to accurately instantiate source terms in software based on finite element methods such as ComSol (www.comsol.fr).

Combined optical depth τ of infrared participating gases CO_2 and H_2O (O_2 and N_2 are ignored) represents millions of lines that are functions of temperature T, pressure p, and density of absorbing molecule u_g in the following formulas copied from [9, Annex A.2] with the same notations.

$$\frac{S_{\eta\eta'}(T)}{S_{\eta\eta'}(T_{\text{ref}})} = \frac{Q(T_{\text{ref}})}{Q(T)} \frac{e^{-\frac{c_2 E_\eta}{T}}}{e^{-\frac{c_2 E_\eta}{T_{\text{ref}}}}} \frac{\left(1 - e^{-\frac{c_2 \nu_{\eta\eta'}}{T}}\right)}{\left(1 - e^{-\frac{c_2 \nu_{\eta\eta'}}{T_{\text{ref}}}}\right)} \quad (1)$$

$$\tau(\nu) = \sum_g u_g \sum_{\eta \to \eta'} S_{\eta\eta'}(T) f(\nu - \nu_{\eta\eta'}) \quad (2)$$

$$I_{\text{out}} = I_{\text{in}} e^{-\tau(\nu)l} + I(\nu, T) \left(1 - e^{-\tau(\nu)l}\right) \quad (3)$$

$$I(\nu, T) = \frac{2h\nu^3}{c^2} \cdot \frac{1}{\left(e^{\frac{c_2\nu}{T}} - 1\right)} \quad (4)$$

The first formula provides a ratio $S_{\eta\eta'}(T)/S_{\eta\eta'}(T_{\text{ref}})$ for the intensity of the line due to transition between lower and upper states η and η' of component gas g centered on wavenumber $\nu_{\eta\eta'}$. This ratio is applied to the 16 contributions of this line in the wavelength space around $\nu_{\eta\eta'}$. Once this transformation is performed for all the lines of all the gases, the contributions are cumulated to obtain $\tau(\nu)$ for all the considered wavenumbers ν. We apply Beer-Lamber law for absorption (first term of I_{out}) and Planck law for heat induced emissions (second term of I_{out}) for a ray passing through length l of an isothermal homogeneous finite volume of Figure 5. GPUs handle all data-parallel computations and Listing 1 presents how formulas (2), (3) and (4) are implemented.

After the GPU has computed I_{out} for all the considered wavenumbers the power of the total heat transfer is obtained by summing $I_{\text{in}} - I_{\text{out}}$ of up to $2^{24} \approx 16 \cdot 10^6$ values stored in a 2 dimensional square matrix. This task requires to sum all the data of a texture and we used a parallel reduction scheme adapted to GPUs [8]. The sum is evaluated with an iterative algorithm where each iteration sums of 4 pieces of data from the previous iteration.

Integrations with respect to space in the simulation of non-isothermal flows is obtained by Monte-Carlo line-by-line ray-tracing paradigm as presented Figure 5. The main

380

Listing 1. Parallel evaluation of (2)-(4)

```
!!ARBfp1.0

...
# sratio_g{1-2} = S_ηη′(T)/S_ηη′(T_ref) are computed
# in the omitted part from (1)
# and stored in a texture
TEX sratio_g1, sig_coords_1, texture[4], RECT;
TEX sratio_g2, sig_coords_2, texture[4], RECT;

# tref_g{1-2} = u_g S_ηη′(T_ref)f(ν − ν_ηη′) are constant
# textures computed on CPU and transfered to
# GPU memory on program initialization
TEX tref_g1, vnu_coords_1, texture[5], RECT;
TEX tref_g2, vnu_coords_2, texture[5], RECT;

# Suming up the contributions of the two gases
MUL tau, sratio_g1, tref_g1;
MAD tau, sratio_g2, tref_g2, tau;

MUL tau, tau, ll;
# ll = −l/ln(2) is a scalar
# set for each iteration
# Factors $1/ln(2)$ are introduced as GPUs
# currently only support base-2 exponentials

# Special functions need 4 invocations
EX2 exp_tau_l.x, tau.x;
EX2 exp_tau_l.y, tau.y;
EX2 exp_tau_l.z, tau.z;
EX2 exp_tau_l.w, tau.w;

MUL exponent, c2T, nu;
# c2T = c_2/(T ln(2))  is a scalar
# set for each iteration

EX2 den.x, exponent.x;
EX2 den.y, exponent.y;
EX2 den.z, exponent.z;
EX2 den.w, exponent.w;

SUB den, den, {1, 1, 1, 1};

RCP inv.x, den.x;
RCP inv.y, den.y;
RCP inv.z, den.z;
RCP inv.w, den.w;

MUL nu3, nu, nu;
MUL nu3, nu3, nu;
MUL nu3, nu3, hc2;
# hc2 = 2h/c^2 is a constant scalar


MUL factor1, inv, nu3;
SUB factor2, one, exp_tau_l;
MUL term, i_in, exp_tau_l;
# i_in = I_in is the I_out texture of
# the previous iteration

# Return result.color = I_out as the color of
# the pixel to be written
MAD result.color, factor1, factor2, term;

END
```
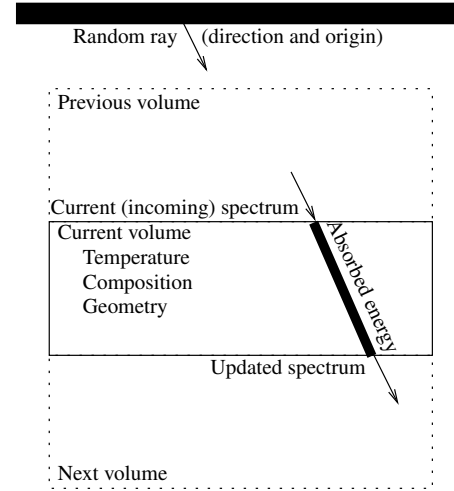

Figure 5. Monte-Carlo ray-tracing.

simulation code on CPU directs this process and averages the effect of individual rays.

We designed a program in two parts. The first part is executed by the CPU and represents 3500 lines of C++ code and OpenGL directives. The second part is executed by the pixel shaders of the GPU and represents 250 lines of OpenGL shading primitives (ARB fragment program). Listing 1 is extracted from these 250 lines and corresponds to the evaluation of formulas (2), (3) and (4). In addition, we wrote the same program in C to measure the benefit of the use of a GPU. This code was compiled with Microsoft Visual C++ 2005, optimizing for speed (/Ox /arch:SSE). We ran both programs on a Pentium 4 system with 1 GB of DDR2 memory and with a Nvidia 7800GTX and an ATI RX1800XL graphic card both with 256 MB of GDDR3.

We measured the number of lines evaluated per second depending on the number of lines per ray. The results are plotted in Figure 6 with logarithmic axes and show a speed-up as high as 420 compared to CPU. Timing is done on 100 rays treated sequentially. GPU performance loss around 10^6 lines per ray is due to data too large to fit in graphic memory and should disappear with newer GPU boards.

This impressive speed-up is partially due to the ability of GPUs to perform many complex operations per cycle. Each pixel shader can start one exponential per cycle thanks to dedicated hardware. As there are up to 24 shaders, 24 exponentials are initiated at 486 Mhz leading to $13.2\ 10^9$ exponentials per second. On CPUs, exponential functions are evaluated in software or in micro-code and require typically 100 cycles to complete. On a 3 Ghz Pentium 4 this means about $30\ 10^6$ exponentials per second. The second reason
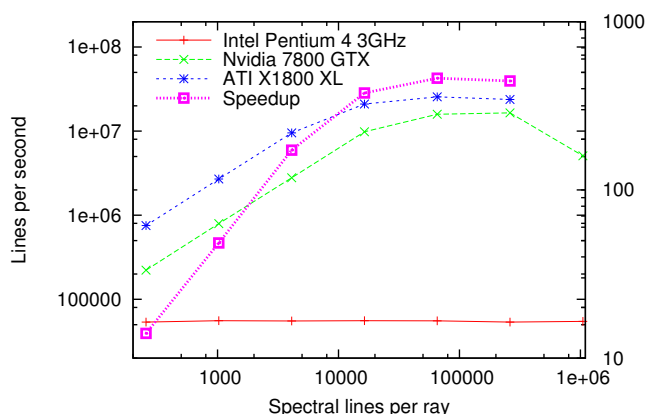
Figure 6. Number of lines treated by second.

for our speed-up lies in the fact that GPUs and drivers exploit regularity in the code to hide memory latency and execute floating point operations in parallel in pixel shaders.

4. Conclusion and perspectives

We started this report with test vectors aimed at the characterization of the floating point operators of GPUs. We showed that:

- Temporary results are computed to 32 bit format.

- Multipliers use constants to compensate for discarded partial products.

- Some Nvidia and ATI adders use an extra bit.

We will certainly set up more test vectors as we continue working on GPUs. Up to date tests are available from the authors upon request by email.

We accelerated the computation of radiation properties in order to simulate precisely, i.e. using line-by-line spectra of gases. Common speed-up brought by GPU start at 5 and may climb to 50 as some developments in the industry are claiming[1]. Our GPU implementation is 400 times faster than CPU evaluation. This performance almost preserves the computing resource available on CPU as we noticed a runtime increase below 1% when our test program saturates our CPU and GPU compared to the same program with no request to GPU.

These figures where obtained using a fixed number (16) of points of evaluation for each line. Our next version will dynamically adapt the number of points depending on the local temperature and the intensity of the line. This tasks will involve vertex shaders and blending units. Blending units starting with Nvidia 8800 operate on 32 bit floating

---

[1]See http://www.emphotonics.com/fastfdtd.html.

point data. Work on radiosity will be performed only if discrepancies between simulations and experimentations show that the effect of diffusion cannot be ignored.

The impressive speed-up reported here was due to the large number of lines for one single ray-tracing leading to a huge amount of data parallel transformations. Similar speed-ups may be obtained for other settings. One possible application of GPGPU with connection to the industry, is to speed-up simulations of elaborate surfaces of planar solar receptors. Software will average spectral effects to two bands of wavelength (infrared and visible) but it will consider a large number of independent rays to accurately account for anisotropic reflections and absorptions.

As we are building know-how on porting simulations for thermal sciences to GPUs we will explore automatic tools and build libraries of techniques to efficiently reuse parts of our developments.

References

[1] I. Buck, K. Fatahalian, and P. Hanrahan. GPUbench: evaluating gpu performance for numerical and scientifc application. In *Proceedings of the ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–20, Los Angeles, California, 2004.

[2] C. Cebenoyan. Floating point specials on the GPU. Technical report, Nvidia, february 2005.

[3] G. D. Graça and D. Defour. Implementation of float-float operators on graphics hardware. In *7th Real Numbers and Computers Conference*, pages 23–32, Nancy, France, 2006.

[4] K. Hillesland and A. Lastra. GPU floating-point paranoia. In *ACM Workshop on General Purpose Computing on Graphics Processors*, page C8, August 2004.

[5] L. Ibgui and J.-M. Hartmann. An optimized line by line code for plume signature calculations — I: model and data. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 75(3):273–295, 2002.

[6] K. A. Jensen, J.-F. Ripoll, A. A. Wray, D. Joseph, and M. E. Hafi. On various modeling approaches to radiative heat transfer in pool fires. *Combustion and Flame*, 148(4):263–279, 2007.

[7] D. Manocha. General purpose computations using graphics processors. *IEEE Computer*, 38(8):85–88, 2005.

[8] M. Pharr, editor. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[9] L. Rothman et al. The HITRAN molecular spectroscopic database and HAWKS (hitran atmospheric workstation): 1996 edition. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 60(5):665–710, 1998.

[10] M. J. Schulte and E. E. Swartzlander. Truncated multiplication with correction constant. In *Proceedings of the 6th IEEE Workshop on VLSI Signal Processing*, pages 388–396. IEEE Computer Society Press, 1993.

[11] D. Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.