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

Caroline Collange, Marc Daumas, David Defour

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

Sylvain Collange (ELIAUS), Marc Daumas (LIRMM-CNRS & ELIAUS) and David Defour (ELIAUS)
Université de Perpignan Via Domitia
52 avenue Paul Alduy — Perpignan 66860 — France
firstname.lastname@univ-perp.fr

Abstract

Graphics Processing Units (GPU) 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 temperatures, velocities and pressure to design next generations of solar receivers. Such codes will delegate to our contribution on GPUs the accurate computation of heat transfers due to radiations. We use Monte-Carlo line-by-line ray-tracing through finite volumes. That 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.

1 Introduction

Graphics Processing Units (GPU) offer computing resources higher than the ones available on general processing units [8]. With the delivery of the latest generations of GPUs, they can be used for general processing (GPGPU) [7][1] 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 of the various hardwired functionalities available on GPU. We describe the architecture of GPUs and properties of the implemented floating point arithmetic in Section 2. Section 3 presents the accurate estimation of radiative heat transfers due to the filtering of incidental rays and the generation of heat induced rays. We elaborate on the performances of our prototype and perspectives in Section 4. We do not account for diffusion of rays in this preliminary study as our medium does not contain particles and is relatively scattered. Such a task could be done by maintaining line by line radiosity and by partially hiding data transfers between GPU and main memory.

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] and references herein) leaving a large gap before direct numerical simulation (DNS) could be envisioned. Many simulations are currently performed only 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%

Although our approach is based on finite volumes used for example by Fluent and Trio-U, this work can also be applied to accurately instantiate source terms in software based on finite element methods such as Femlab.

2 Graphics Processing Units (GPU)

GPUs treat mainly geometrical objects and pixels. Images are created by applying geometrical transformations to vertices and by splitting objects into fragments or pixels. Calculations are carried out by various stages composing the graphics pipeline, as 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

*This work has been partially funded by the EVA-Flo project of the ANR and a STICS-UM2 multidisciplinary grant awarded to LIRMM, ELIAUS and PROMES laboratories. This work has been possible thanks to the kind help of Gilles Flamant, Pierre Neveu, Xavier Py and Régis Olives from PROMES laboratory (CNRS) and Frédéric André from CETHIL (CNRS-INSA Lyon).

[1] See `http://www.gpgpu.org/`.

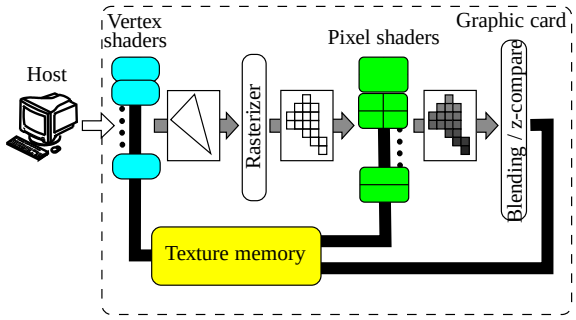

FIG. 1 – Model of the graphics pipeline.

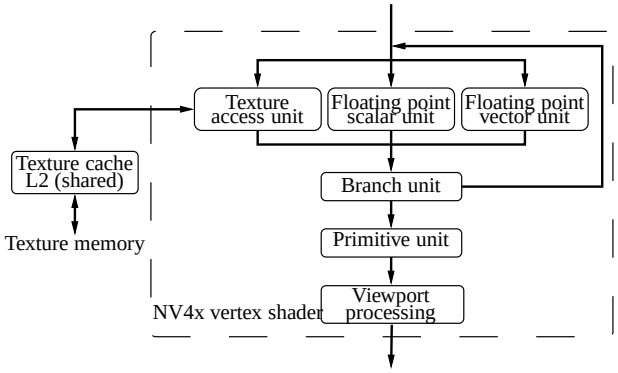

FIG. 2 – Vertex shader of the Nvidia 7800GTX.

on the example of a triangle. Vertex shaders treat 3 vertices whereas pixel shaders treat 17 pixels. For a given geometrical object, the number of pixels is usually 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 objects (polygons, lines, points). Objects are transformed (rotation, translation, illumination...) and assembled to create more elaborate objects. These operations are carried out by *vertex shaders* (see Figure 2 adapted from [8]). At each cycle, the vertex shader is able to initiate a *Multiply and Accumulate* (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.

When an object has reached its final position, form and lighting, it is split into fragments or pixels. An interpolation is applied to deduce properties of each pixel. Pixels are handled by *pixel shader*s to apply textures and set colors, for example (see Figure 3 adapted from [8]). The first floating

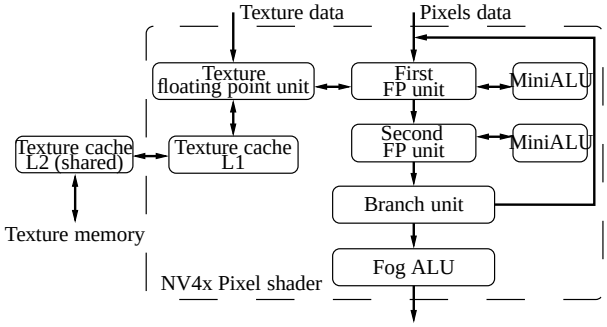

FIG. 3 – Pixel shader of the Nvidia 7800GTX.

point unit 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 floating point formats implemented on GPUs and CPUs. 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. We used OpenGL primitives and stored data in textures using *Frame Buffer Object* and respectively *texture rectangle* and texture 2D 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. We mostly targeted single precision (32 bits) though these tests can be adapted to smaller formats.

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.

Tests showed that no such things occur on GPUs but they revealed a surprising feature. It appears that the first floating point unit produces a mantissa with one extra bit that can be used by the second unit on pixel shaders of Nvidia. We conjecture that this extra bit is implemented for backward compatibility with some old functionality of graphic hardware.

This extra bit seems to be a nice feature to reduce round-off errors but the difference $x + y - x \oplus y$ can no longer be

TAB. 1 – Representation format of floating point numbers 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 remaining 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.

Reference	Number of bits				Non numerical values
	Total	Sign	Exponent	Fraction	
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	

represented in the working format for input that do not overflow. This implied modifications in some multiple precision tools [3].

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 $41 \cdot 2^{-30}$ and that 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. Absorption coefficients κ_ν of infrared participating gases CO_2 and H_2O (O_2 and N_2 are ignored) represent millions of lines that are functions of temperature T, pressure p, and composition x_g in the following formulas [9, Annex A.2]. GPUs handle all data-parallel computations in Listing 1 using constants of Table 3.

$$\frac{S_{ig}(T)}{S_{ig}(T_0)} = \frac{Q(T_0)}{Q(T)} \cdot \frac{e^{-\frac{hcE''_i}{k_B T}}}{e^{-\frac{hcE''_i}{k_B T_0}}} \cdot \frac{\left(1 - e^{-\frac{hc\nu_0}{k_B T}}\right)}{\left(1 - e^{-\frac{hc\nu_0}{k_B T_0}}\right)}$$

$$\kappa_\nu = \sum_{g \in \{g_1, g_2\}} \frac{x_g p}{k_B T} \sum_i S_{ig}(T) \Phi_i(\nu - \nu_{0,i})$$

$$I_{\text{out}} = I_{\text{in}} \cdot e^{-\kappa_\nu \cdot l} + \frac{2h\nu^3}{c^2} \cdot \frac{1}{\left(e^{\frac{hc\nu}{k_B T}} - 1\right)} \cdot \left(1 - e^{-\kappa_\nu \cdot l}\right)$$

The first formula provides a ratio $\frac{S_{ig}(T)}{S_{ig}(T_0)}$ for spectral ray i centered on wavenumber ν_0. This ratio is applied to the 16 contributions in the wavelength space of spectral ray i and gas g in κ_ν. Once this transformation is performed for all the spectral rays of all the gases, the contributions are cumulated to obtain κ_ν 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.

Wavelength integration computes the total heat transfer $\int (I_{\text{in}} - I_{\text{out}}) d\nu$ of about 2^{24} values stored in a matrix of size `original_size` and returns to main memory the total energy absorbed in the finite volume. This task requires to sum all the data of a texture. It is done by code in Listing 2 based on a parallel reduction scheme already published [8]. The total is evaluated with a multipass algorithm applied $\log_2($`original_size`$)$ times as during each pass the algorithm produces the sum of 4 pieces of data from the previous stage.

Simulation of non-isothermal flows needs, at least, integrations with respect to space. This is obtained by Monte-Carlo line-by-line ray-tracing paradigm as presented Figure 5. The main 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). Among these 250 lines, Listing 1 and 2 correspond to the

TAB. 2 – Arithmetic experimentations and results. $\oplus$, $\ominus$, $\otimes$ are the addition, subtraction and multiplication operators implemented on GPU. $M = 2^{127}(2 - 2^{-23})$. $U[a, b)$ are uniformly distributed random variables on $[a, b)$. {ATI;NV}-{P;V} corresponds to the Pixel or Vertex shader on ATI or Nvidia GPU. 'All' means all combinations. Random tests are performed on 2^{23} inputs, other tests are exhaustive.

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

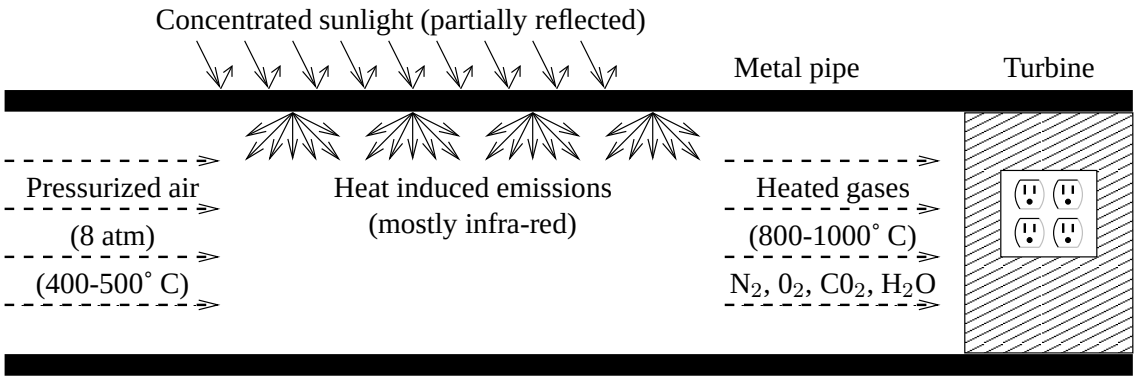

FIG. 4 – The solar receptor as simulated. This device produces electricity from sunlight concentrated by a large reflector. Concentrated sunlight is used to heat a metal pipe that transfers heat through contact and infra-red radiations. The goal is to transfer as much energy as possible to the turbine.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

Listing 1 – Parallel monochromatic intensity

```
!!ARBfp1.0
...
# Suming up the contributions of gases
MUL kappa_nu, sig_g1, vnu_g1;
MAD kappa_nu, sig_g2, vnu_g2, kappa_nu;

MUL kappa_nu, kappa_nu, ll;

# Special functions need 4 invocations
EX2 exp_kappa_nu_l.x, kappa_nu.x;
EX2 exp_kappa_nu_l.y, kappa_nu.y;
EX2 exp_kappa_nu_l.z, kappa_nu.z;
EX2 exp_kappa_nu_l.w, kappa_nu.w;

MUL exponent, hckbt, nu;
EX2 den.x, exponent.x;
EX2 den.y, exponent.y;
EX2 den.z, exponent.z;
EX2 den.w, exponent.w;

SUB den, den, one;

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;

MUL factor1, inv, nu3;
SUB factor2, one, exp_kappa_nu_l;
MUL term, i_in, exp_kappa_nu_l;

MAD i_out, factor1, factor2, term;

END
```

TAB. 3 – Constants of Listing 1. Vector constants of gases g_1 and g_2 are computed on CPU and transfered to graphic textures on program initialization. Other (scalar) constants are computed on CPU and transfered to GPU for each iteration. Factors $1/ln(2)$ are introduced as GPUs currently only support base-2 exponentials.

Scalar values computed on CPU and transfered to GPU		
`qr_g{1-2}`	$=$	$\frac{Q(T_0)}{Q(T)}$
`ll`	$=$	$-\frac{l}{\ln(2)}$
`hc2`	$=$	$\frac{2h}{c^2}$
`hckbt`	$=$	$\frac{hc}{k_B T \ln(2)}$
`xpkbt_g{1-2}`	$=$	$\frac{x_g p}{k_B T}$
Vector values stored in graphic textures		
`vnu_g{1-2}`	$=$	$S_{ig}(T_0)\Phi_i(\nu - \nu_{0,i})$
`Es_g{1-2}`	$=$	E_i''
`den_g{1-2}`	$=$	$e^{\frac{E_i''}{k_B T_0}} \left(1 - e^{-\frac{hc\nu_0}{k_B T_0}}\right)^{-1}$

Listing 2 – Sum of the monochromatic energy absorbed by the current finite volume

```
int src_size = original_size;
int dest_size = original_size / 2;
int levels = log2(original_size);

for(int i = 0; i < levels - 1; i++){
    // Reading from reduct[i]
    reduct[i]->AssignTexNum(8);
    // Drawing to reduct[i+1]
    reduct[i+1]->AttachRenderTarget(0);
    glBegin(GL_TRIANGLES);
    ...
!!ARBfp1.0

OUTPUT I_sum = result.color;
TEMP I0, I1, I2, I3, I_sum_1, I_sum_2;

TEX I0, fragment.texcoord[0], texture[8], RECT;
TEX I1, fragment.texcoord[1], texture[8], RECT;
TEX I2, fragment.texcoord[2], texture[8], RECT;
TEX I3, fragment.texcoord[3], texture[8], RECT;

ADD I_sum_1, I0, I1;
ADD I_sum_2, I2, I3;
ADD I_sum, I_sum_1, I_sum_2;

END
    ...
    glEnd();
    src_size = dest_size;
    dest_size = dest_size / 2;
}
```

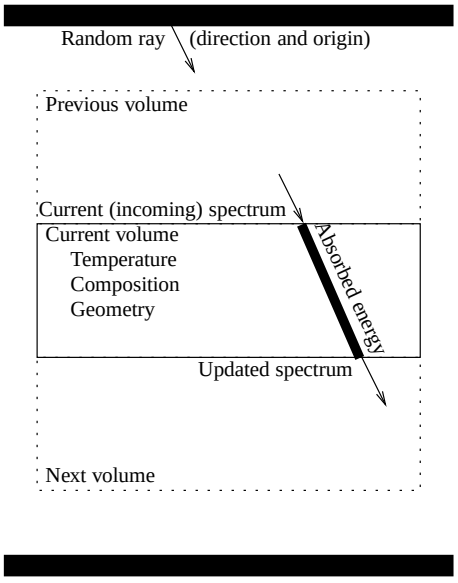

FIG. 5 – Monte-Carlo ray-tracing through finite volumes

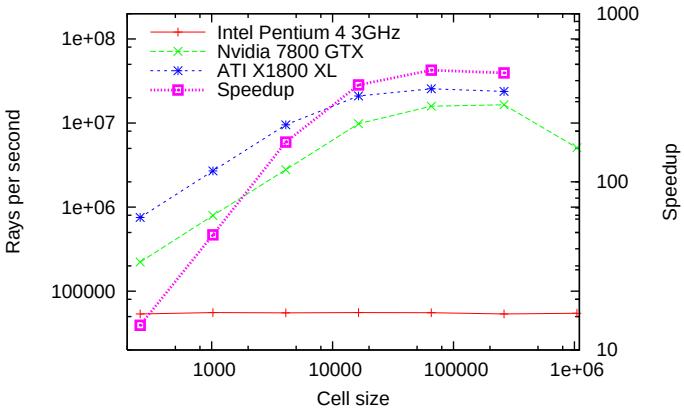

FIG. 6 – Number of ray treated by second. Both GPUs run 100 iterations. GPU performance loss around 10^6 rays is due to data to large to fit in graphic memory and should disappear with newer GPU boards.

core of the process we are simulating. In addition, we wrote the same program entirely using CPU to measure the benefit of the GPU. We ran these programs on a Pentium 4 system with 1 GB of memory and with a Nvidia 7800GTX and an ATI RX1800XL graphic card both with 256 MB. We measured the number of rays evaluated per second depending on the number of cells we are considering. The results are plotted in Figure 6 with logarithmic axis and show a speed-up as high as 420 compared to CPU.

This impressive speed-up is 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 the 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 for oour speed-up lies in 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 floating operators on GPUs that helped us in the development of simulation of solar receptor. We showed that :

- Temporary results are computed to 32 bit format.
- Multipliers uses constants to compensate for discarded partial products.
- Some Nvidia adders use an extra bit.

We will certainly set up more test vectors as we continue working on GPUs.

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[2]. 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% 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 ray. Our next version will dynamically adapt the number of rays depending on the local temperature and the intensity of the ray. This tasks will involve vertex shadders and blending units. Blending units starting with Nvidia 8800 operate on 32 bit floating 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 spectral rays 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 a prototype to speed-up simulations of complex receptor surfaces that average spectral effects to two bands of wavelength (infrared and visible) but require a large number of rays to accurately account for anisotropic reflections and absorptions. Such simulations could be the key to the design of home solar receptors with enhanced behavior during mornings and evenings when domestic activities use most heated water.

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.

Références

[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.

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

[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.