Line-by-line spectroscopic simulations on graphics processing units

Caroline Collange, Marc Daumas, David Defour

Available online at www.sciencedirect.com

ScienceDirect

Computer Physics Communications 178 (2008) 135–143

Computer Physics Communications

www.elsevier.com/locate/cpc

Line-by-line spectroscopic simulations on graphics processing units [☆,☆☆]

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

[a] *ELIAUS, UPVD, 52 avenue Paul Alduy, 66860 Perpignan, France*
[b] *LIRMM, CNRS, UM2, 161 rue Ada, 34392 Montpellier, France*

Received 8 June 2007; received in revised form 29 July 2007; accepted 17 August 2007

Available online 14 September 2007

Abstract

We report here on software that performs line-by-line spectroscopic simulations on gases. Elaborate models (such as narrow band and correlated-K) are accurate and efficient for bands where various components are not simultaneously and significantly active. Line-by-line is probably the most accurate model in the infrared for blends of gases that contain high proportions of H_2O and CO_2 as this was the case for our prototype simulation. Our implementation on graphics processing units sustains a speedup close to 330 on computation-intensive tasks and 12 on memory intensive tasks compared to implementations on one core of high-end processors. This speedup is due to data parallelism, efficient memory access for specific patterns and some dedicated hardware operators only available in graphics processing units. It is obtained leaving most of processor resources available and it would scale linearly with the number of graphics processing units in parallel machines. Line-by-line simulation coupled with simulation of fluid dynamics was long believed to be economically intractable but our work shows that it could be done with some affordable additional resources compared to what is necessary to perform simulations on fluid dynamics alone.

Program summary

Program title: GPU4RE
Catalogue identifier: ADZY_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADZY_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html
No. of lines in distributed program, including test data, etc.: 62 776
No. of bytes in distributed program, including test data, etc.: 1 513 247
Distribution format: tar.gz
Programming language: C++
Computer: x86 PC
Operating system: Linux, Microsoft Windows. Compilation requires either gcc/g++ under Linux or Visual C++ 2003/2005 and Cygwin under Windows. It has been tested using gcc 4.1.2 under Ubuntu Linux 7.04 and using Visual C++ 2005 with Cygwin 1.5.24 under Windows XP.
RAM: 1 gigabyte
Classification: 21.2
External routines: OpenGL (http://www.opengl.org)
Nature of problem: Simulating radiative transfer on high-temperature high-pressure gases.
Solution method: Line-by-line Monte-Carlo ray-tracing.
Unusual features: Parallel computations are moved to the GPU.

[☆] 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).
[☆☆] This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
[*] Corresponding author at: ELIAUS, UPVD, 52 avenue Paul Alduy, 66860 Perpignan, France.
E-mail address: marc.daumas@univ-perp.fr (M. Daumas).

0010-4655/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2007.08.013

Additional comments: nVidia GeForce 7000 or ATI Radeon X1000 series graphics processing unit is required.
Running time: A few minutes.

PACS: 44; 44.40.+a; 07.05.Tp; 89.80.+h

Keywords: Radiative heat transfer; Line-by-line; Simulation; Graphics processing unit; GPU; Parallelism; Data parallel

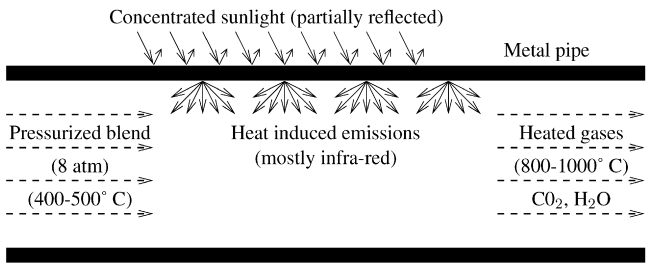

Fig. 1. Experimental setting of our prototype simulation.

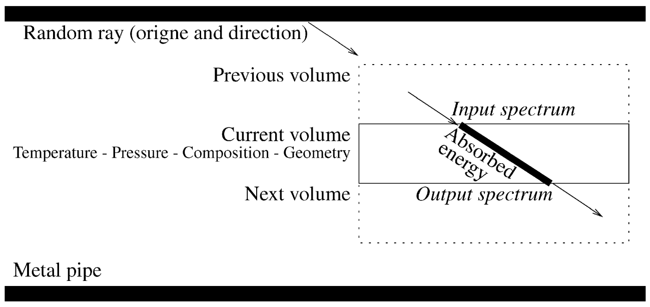

Fig. 2. Monte-Carlo ray-tracing through finite volumes.

1. Introduction

Spectroscopic databases such as HITRAN[1] (HIgh-resolution TRANsmission) [1] allow one to compute intensity diagrams, absorption coefficient profiles, transmittance, absorption, and radiance spectra on any mixture of gases. High temperature conditions are covered by specific databases including CDSD[2] (Carbon Dioxide Spectroscopic Databank) and HITEMP[1] (HIgh-TEMPerature) amongst others [2–4].

Most databases can be used by proprietary software such as JavaHAWKS[1] or PcLnWin[3] as well as server side software.[2] These codes apply Lorentz or Voigt profiles [5, Appendix A.2], approximations to the partition functions [6] and well known laws such as Beer–Lamber law for absorption and Planck law for heat induced emissions.

The experimental setting of prototype simulation is presented in Fig. 1. Concentrated sunlight is used to heat a metal pipe that transfers heat through contact and infrared radiations to a pressurized mixture of infrared participating gases. We consider wavenumbers between $\nu_{\min} = 512\ \text{cm}^{-1}$ and $\nu_{\max} = 16{,}895\ \text{cm}^{-1}$ for a mixture of H_2O and CO_2. Line-by-line simulation is time consuming. Though real applications require more accuracy we start with lines for which the intensity is higher than 0.1% of the most intense line. This leads to 21,879 lines at 1000 K. To be consistently accurate, we sampled the wavenumber axis down to $4096^{-1}\ \text{cm}^{-1}$. This choice yields 2,561,660 points at 1000 K and 8 atm.

In an anisotropic setting discretized in finite volumes such as the one that we simulate we cannot produce analytical expressions for the mean effect of Beer–Lamber law. We obtained the mean effects using Monte Carlo ray tracing as presented in Fig. 2 on over 1000 rays where each ray follows its path through an average of 6 volumes before it hits the opposite side of the pipe.

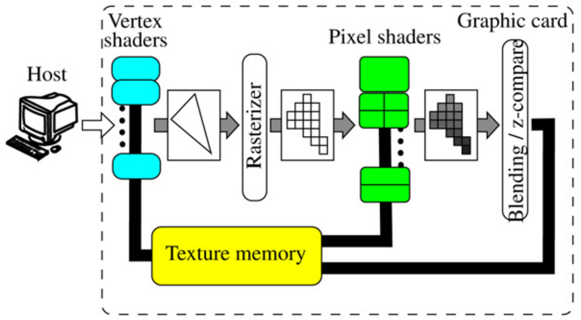

Fig. 3. Simplified rendering pipeline.

These figures amount to a minimum total (sequential) running time close to 7 hours and 40 minutes. As our application is embarrassingly parallel, we could reduce the running time using (expensive) distributed machines. We decided to use graphics processing units instead. To the best of our knowledge, there is no prior art in the implementation of these tasks on such units.

Graphics Processing Units (GPU) offer computing resources higher than the ones available on processors [7]. With the delivery of the latest generations of GPUs, they can be used for general processing[4] (GPGPU) [8] and become application specific co-processors for regular and heavily data-parallel processing.

GPUs are graphics oriented units manipulating streams of data through what is called a rendering pipeline (see Fig. 3). The *host* sends vertices to position primitive geometrical objects (polygons, lines, points). *Vertex shader*s transform objects (rotation, translation, illumination...) and assemble them to create more elaborate objects. When an object has reached its final position, form and lighting, it is split into pixels. The *rasterizer* interpolates properties of the vertices on each pixel. *Pixel shader*s apply *texture*s and assign colors to pixels, for example. *Blending and z-compare* affect objects that are not directly

1 http://cfa-www.harvard.edu/hitran/.
2 http://cdsd.iao.ru/.
3 http://ontar.com/.
4 http://www.gpgpu.org/.

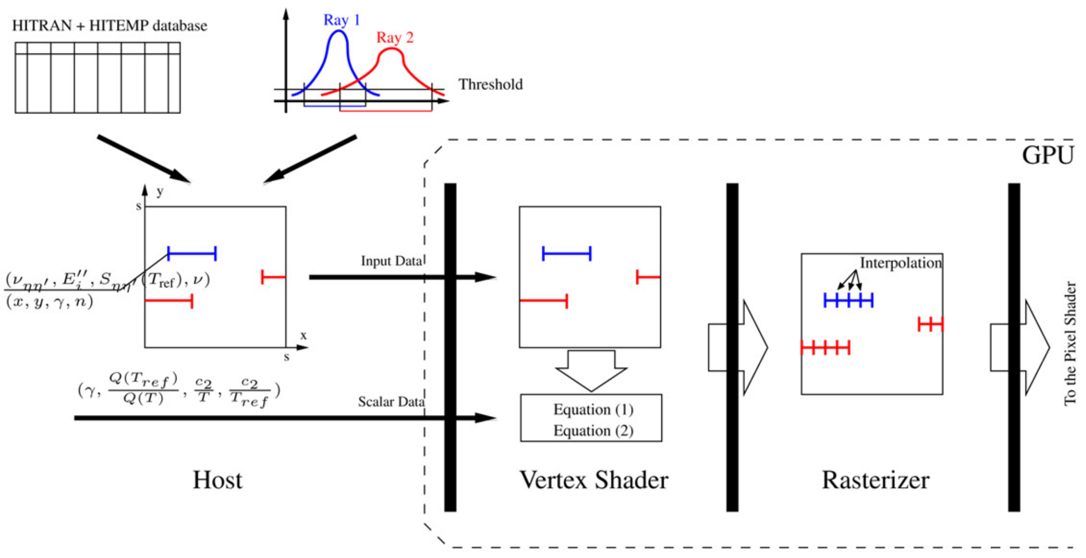

Fig. 4. Description of data generation on the host and temperature and pressure corrections in the first steps of the graphics pipeline.

visible but partially hidden or seen through other objects. This process ends with the images ready be displayed.

GPUs can be programmed following various paradigms but we obtain efficiency, portability and re-usability by implementing our application as if it were a graphics application using OpenGL [9]. We apply temperature and pressure correction on line intensity and line half-width in the *vertex shader*s as described in Section 2. Discretization on the wavenumber axis is later performed by the *rasterizer*. The *pixel shader*s compute Lorentz profile on the discretized lines as presented in Section 3 and the optical depth of various lines and component gases are summed in the *blending* unit. When this process is over, the *pixel shader*s compute the output spectrum of the ray based on its input spectrum and it yields one component of the power absorbed by the finite volume as described in Section 4.

Although our approach is based on finite volumes used by Fluent [10] and Trio-U,[5] this work can also be applied to accurately instantiate source terms in software based on finite element methods such as ComSol.[6] Coupling is obtained by running the simulation of fluid dynamics on the processor and estimating radiative source terms on the GPU. Simulations where the code for fluid dynamics runs on parallel machines [11] will certainly benefit from quasi-linear speedups on GPUs as presented for a similar embarrassingly parallel application of GPGPU [12].

2. Temperature and pressure corrections on line intensity and line half-width

The first operation to be done before a ray enters a finite volume consists in temperature and pressure corrections on line intensity and line half-width. Similar corrections are applied to each line of each gas component of the volume considered. They are based on homogenized temperature and pressure (total and partial for each component). The temperature and pressure fields are provided in intermediate files by the simulation on fluid dynamics. We use the notation of [5, Appendix A.2] in Eqs. (1), (2) and the rest of this text.

$$S_{\eta\eta'}(T) = S_{\eta\eta'}(T_{\text{ref}}) \cdot \frac{Q(T_{\text{ref}})}{Q(T)} \cdot \frac{e^{-\frac{c_2 E_\eta}{T}}}{e^{-\frac{c_2 E_\eta}{T_{\text{ref}}}}} \cdot \frac{(1 - e^{-\frac{c_2 \nu_{\eta\eta'}}{T}})}{(1 - e^{-\frac{c_2 \nu_{\eta\eta'}}{T_{\text{ref}}}})}, \tag{1}$$

$$\gamma(T, p, p_s) = \left(\frac{T_{\text{ref}}}{T}\right)^n \big(\gamma_{\text{air}}(p_{\text{ref}}, T_{\text{ref}})(p - p_s) + \gamma_{\text{self}}(p_{\text{ref}}, T_{\text{ref}}) p_s\big). \tag{2}$$

The description of the operations involved by the execution of these corrections on the GPU is given in Fig. 4. These operations are carried out by the vertex shaders. Such units usually compute on (x, y, z, w) attributes in a 3-dimensional homogeneous space. The last component w helps operate non-linear transformations on vertices.

In our application, vertex shaders receive two attributes, $(\nu_{\eta\eta'}, E_i'', S_{\eta\eta'}(T_{\text{ref}}), \nu)$ and (x, y, γ, n), for each line with the notation of Eqs. (1) and (2). We define $\gamma = \gamma(T_{\text{ref}}, p_{\text{ref}}, p_{s,\text{ref}}) / p_{\text{ref}}$ assuming that there is neither chemical reaction nor phase change. The other quantities used in Eqs. (1) and (2) do not depend on the line considered though they may depend on the gas component considered. They are transmitted and stored as scalar constants for each volume. Constant subexpressions are precomputed only once on the processor.

Vertex shaders cannot store their results to memory. Results are aggregated into attribute groups and passed to the next step of the rendering pipeline. The z coordinate is usually handled in the very last steps of the rendering pipeline (blending and z-compare) and the rasterizer unit is well suited to address data organized on a 2 dimensional array. The x and y coordinates introduced in the previous paragraph are used to encode the wavenumber ν in a discrete square of length s (represented on

[5] http://www-trio-u.cea.fr/.
[6] http://www.comsol.fr/.

the left side in Fig. 4) with the following transformations:

$$\nu(x, y) = \nu_{\min} + (x + s \times y) \times \nu_{\text{step}}, \tag{3}$$

$$x(\nu) = (\nu - \nu_{\min})/\nu_{\text{step}} \bmod s, \tag{4}$$

$$y(\nu) = \bigl((\nu - \nu_{\min})/\nu_{\text{step}} - x(\nu)\bigr)/s. \tag{5}$$

Minimum and maximum wavenumbers are slightly shifted so that s and $\nu_{\text{step}} = (\nu_{\max} - \nu_{\min})/s^2$ are powers of 2.

In our square representation of the wavenumber axis, each line is coded by at least two points. The first point is located at the wavenumber where the line intensity rises above a predefined threshold and the second point is located at the wavenumber where the line intensity falls below the predefined threshold. In most cases, the line is a discrete segment of our square representation. The rasterizer creates the discrete samples of this segment and these samples are routed to the pixel shaders with their corrected line intensity and line half-width.

The wavenumbers where the intensity of some spectral lines is above the threshold may cross the border of our square representation. We represent such lines by two segments or more, one segment for each row of the square where the line is above the threshold. In our current software, such lines are detected statically on the processor by considering the worst case. Thanks to the new hardware and software capabilities of the geometric shader defined by Direct3D 10 that can generate new data set from existing ones, future versions of our software will handle this step dynamically on the GPU.

3. Monochromatic optical depth for each line of each gas

The second operation needed before a ray enters a finite volume consists in computing the discretized optical depth of each line of each component gas to a given wavenumber. The details of this step are given in Fig. 5.

At that point in the rendering pipeline, the rasterizer has generated two attribute groups for each discretized wavenumber of each line and each gas above the threshold (see Section 2). The wavenumber ν of each sample is linearly interpolated, while the other attributes are copied unaltered. To fully exploit most resources of GPUs operating on four-component attribute groups, the wavenumber axis is oversampled four times corresponding to sub-pixel interpolation, resulting in our target resolution of $4096^{-1}\ \text{cm}^{-1}$.

For each sample, the line intensity is multiplied by the shape function. As our application simulates fluids around 8 atm, we neglect the pressure-shift correction and we use Lorentz profile as shape function:

$$f(\nu, \nu_{\eta\eta'}, T, p) = \frac{1}{\pi} \frac{\gamma(T, p)}{\gamma(T, p)^2 + (\nu - \nu_{\eta\eta'})^2}. \tag{6}$$

Contribution of transition between lower and upper states η and η' of component gas g to the monochromatic absorption coefficient is

$$k_{\eta\eta'}(\nu, T, p) = S_{\eta\eta'}(T) f(\nu, \nu_{\eta\eta'}, T, p). \tag{7}$$

We obtain the contribution to the optical depth $\tau_{\eta\eta'} = uk_{\eta\eta'}$ where u is the density of absorbing molecules per unit path length.

All the formulas above are evaluated on *pixel shaders*. Listing 1 is the source code of a Lorentz profile in Cg language. It can be easily adapted to compute other profiles, such as Voigt profiles.

For a given geometrical object, the number of pixels is usually larger than the number of vertices and in our application the number of samples is much larger than the number of lines considered. GPUs typically contain more pixel shaders than vertex shaders. The current ratio is commonly 24 against 8. This is the reason why we used the pixel shaders for this step.

Pixel shaders as well as vertex shaders cannot control the location of their results. Results are just passed to the next step of the rendering pipeline. The blending units or ROPs "blend" the components of new pixels with the previous values of the pixels at the same (x, y) location. Both GPUs used in out tests contain 16 blending units. Such units are parametrized to compute $aP + bQ$ on the components of pixels P and Q, where a and b are constants. We sum overlapping monochromatic optical depth with $a = b = 1$. At this point in the rendering pipeline, we have a discretization of the τ function.

Most blending units of GPUs shipped in 2006 operate on a 16-bit reduced-precision format that does not provide enough

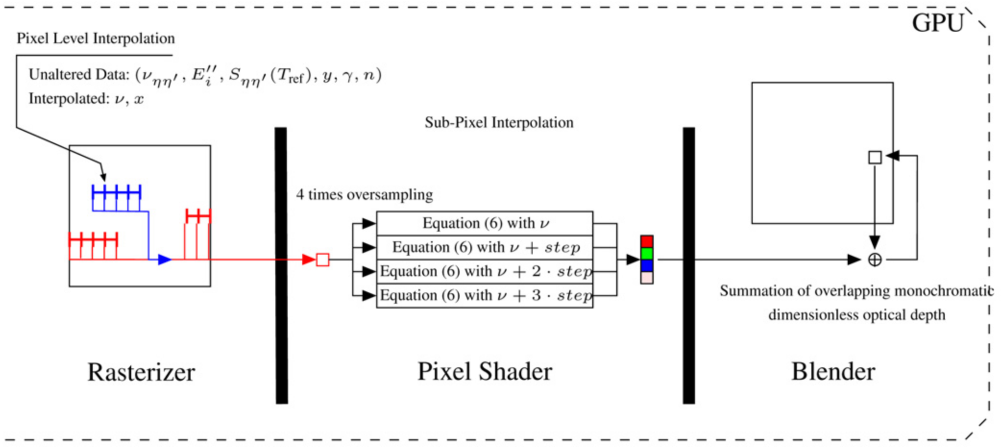
GPU
Pixel Level Interpolation
Unaltered Data: $(\nu_{\eta\eta'}, E_i'', S_{\eta\eta'}(T_{\text{ref}}), y, \gamma, n)$
Interpolated: ν, x
Sub-Pixel Interpolation
4 times oversampling
Equation (6) with ν
Equation (6) with $\nu + step$
Equation (6) with $\nu + 2 \cdot step$
Equation (6) with $\nu + 3 \cdot step$
Summation of overlapping monochromatic dimensionless optical depth
Rasterizer
Pixel Shader
Blender

Fig. 5. Description of monochromatic optical depth for each line of each gas.

```
float4 main (
     // Inputs:
         float4 params : TEXCOORD0, // (ν_ηη', 0, S_ηη'(T), ν)
         float4 coords : TEXCOORD1, // (x, y, γ(T, p, p_s), 0)
     // Constants:
         uniform float piinv,        // 1/π
         uniform float4 nu_bias      // (0, 1/4 ν_step, 2/4 ν_step, 3/4 ν_step)
) : COLOR {
        float delta_nu = params.x - params.w;
        float4 nu = delta_nu.xxxx - nu_bias;
           // oversample to 4-component vector
        float gamma = coords.z;
        float4 denom = nu * nu + (gamma * gamma).xxxx;
        return (piinv * gamma * params.z).xxxx / denom;
}
```

Listing 1. Parallel monochromatic intensity.

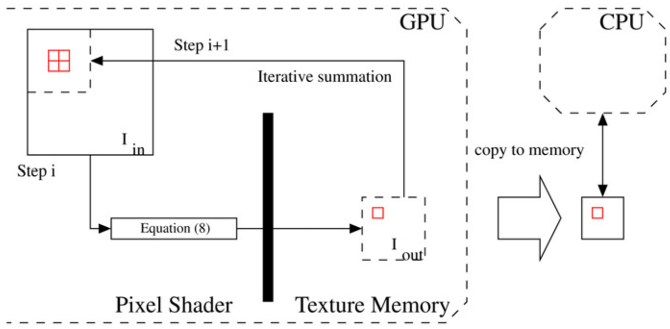

Fig. 6. Description of monochromatic and total sources of power from radiative heat transfers.

accuracy. Alternative methods [7] can be used on these GPUs, at the price of reduced performances. GPUs starting with nVidia GeForce 8000 and ATI Radeon 2000 series can perform blending on single precision data.

4. Mono- and polychromatic sources of power from radiative heat transfers

When steps described in Sections 2 and 3 are over, the GPU computes the monochromatic output intensity I_{out} for a ray passing through length l of an isothermal homogeneous finite volume of Fig. 2. It is based on the monochromatic input intensity I_{in}, Beer–Lamber law for absorption (first term of I_{out} below) and Planck law for heat induced emissions (second term of I_{out}).

$$I_{\text{out}} = I_{\text{in}} \cdot e^{-\tau(\nu)\cdot l} + \frac{2h\nu^3}{c^2} \cdot \frac{1}{(e^{c_2\nu/T} - 1)} \cdot \left(1 - e^{-\tau(\nu)\cdot l}\right). \tag{8}$$

This step is performed in the pixel shaders. Monochromatic input intensities I_{in} are stored in a texture "image" and results I_{out} are stored in a new texture image. The power absorbed by the finite volume is the integral of the monochromatic intensities absorbed by the volume, and it is approximated by the sum of the differences $I_{\text{in}} - I_{\text{out}}$ scaled by ν_{step}. This power has to be scaled by the fraction of the finite volume that is affected by the ray based on the solid angle of the ray.

The architecture of GPUs prevents the use of a global accumulator for this sum. We use an iterative reduction technique [7]. For each iteration, data are summed locally reducing four pixels into one producing a square texture of size reduced by 2. Four memory accesses are performed to compute the value of one pixel that is stored in the reduced texture. When the size of the iterated texture is no longer sufficient to exhibit enough parallelism, the texture is copied to main memory and the sum is finished by the processor.

When all computations needed for one finite volume are over, buffers I_{out} and I_{in} are swapped and the process starts with a new volume. This process ends when the ray hits the opposite wall and I_{in} is reinitialized based solely on Planck law and properties of the metal pipe.

5. Tests and results

Single-precision floating-point only covers limited exponents. We converted HITRAN/HITEMP data to different units so that all intermediate quantities remain between $\pm 10^{-38}$ and $\pm 10^{38}$. These conversions are given in Table 1.

HITEMP database contains 1,283,468 lines for the most common isotope of H_2O and 1,032,269 for the most common isotope of CO_2. Respectively 1,193,874 and 741,495 lines are within the infrared area considerer. We used the following parameters and the thresholds of Table 2 to compare performance at various line densities.

$$s = 2^{11}, \tag{9}$$

$$\nu_{\text{step}} = 2^{-12}\ \mu\text{m}^{-1}, \tag{10}$$

$$\nu_{\text{min}} = 0.0512\ \mu\text{m}^{-1}, \tag{11}$$

$$\nu_{\text{max}} = 1.6895\ \mu\text{m}^{-1}, \tag{12}$$

$$T = 1200\ \text{K}, \tag{13}$$

$$p = 8\ \text{atm}, \tag{14}$$

$$p_s = 0.4 \times p \quad \text{(for infrared participating gases)}. \tag{15}$$

Generating Table 2, we noticed that

$$\max\left(S_{\eta\eta'}(T_{\text{ref}})\right) = 3.5 \times 10^{-20}\ \text{m}^{-1}/(\text{molecule} \cdot \text{m}^{-2}), \tag{16}$$

Table 1
Units used in the HITRAN/HITEMP database and in our application

Variable	ν	γ	p	$S_{\eta\eta'}$	$\tau(\nu)$	$E_{\eta\eta'}$
HITRAN/HITEMP units	cm^{-1}	cm^{-1}/atm	atm	$cm^{-1}/(molecule\ cm^{-2})$	cm^{-1}	cm^{-1}
Units used in GPU	μm^{-1}	$\mu m^{-1}/atm$	atm	$m^{-1}/(molecule\ m^{-2})$	m^{-1}	m^{-1}

Table 2
Statistics on the effort required for a line-by-line computation for different values of the threshold on $k_{\eta\eta'}(\nu, T, p)$

Threshold (m^{-1})	Lines		Samples		Max # of overlap		Max # of samples in a line	
	H_2O	CO_2	H_2O	CO_2	H_2O	CO_2	H_2O	CO_2
1×10^{-13}	353	4051	7324	366,156	2	5	19	145
1×10^{-14}	2060	19,819	126,552	2,435,108	3	15	113	461
1×10^{-15}	7117	75,620	927,628	12,617,948	4	22	395	1457
1×10^{-16}	20,883	222,660	4,678,780	54,189,736	4	28	1259	4609
1×10^{-17}	58,039	525,544	20,700,852	204,068,100	6	54	3987	14,573
1×10^{-18}	155,565	686,791	81,948,032	684,901,816	10	122	12,609	46,085

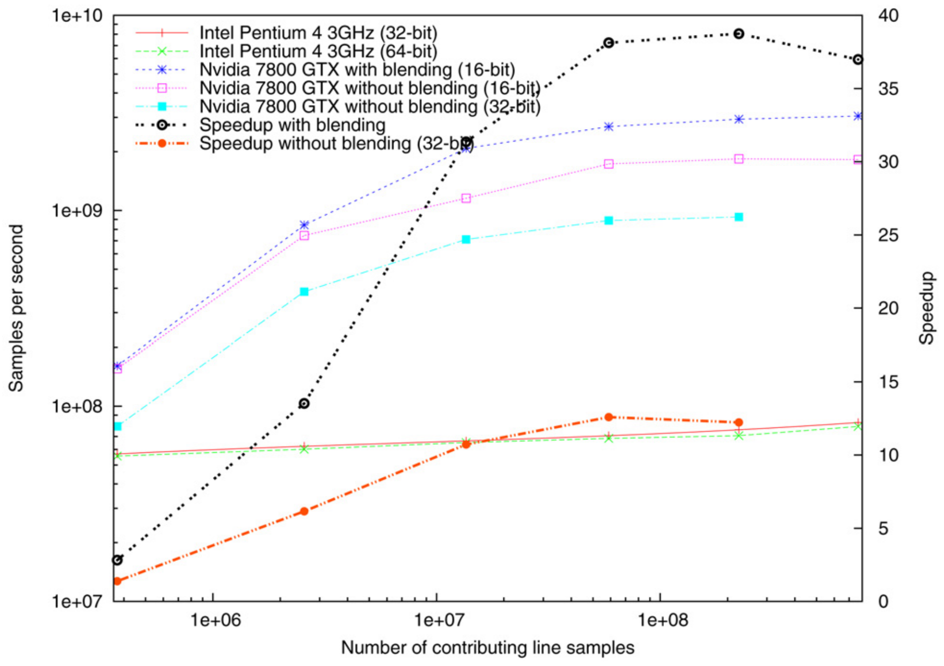

Fig. 7. First pass performance results on our first test configuration.

$$\max\left(k_{\eta\eta'}(\nu, T, p)\right) = 1.26 \times 10^{-11}\ m^{-1}/(molecule \cdot m^{-2}). \tag{17}$$

The first threshold of Table 2 is clearly too high. The next one is probably too high and a good threshold seems to be the third or the fourth one. We will validate this empirical results with interval arithmetic in future work.

Tests were carried out on two workstations: one equipped with a nVidia GeForce 7800 GTX graphics card and running Linux and another equipped with an ATI Radeon X1800 XL graphics card and running Windows XP.

A C++ program performing the same computations was written for purposes of validation and performance comparisons. Under Linux, it was compiled with g++ 4.1.2 with the following flags: `-O3 -march=prescott -mfpmath =sse`. Under Windows, we used Microsoft Visual C++ 2005 with flags `/Ox /arch:sse`.

The first pass through the rendering pipeline is based on Sections 2 and 3. It performs temperature and pressure corrections in vertex shaders, computes optical depth computation in pixel shaders, and merges lines either in pixel shaders or blending units. Fig. 7 shows the relative performance of a Pentium 4 desktop processor and our first GPU.

All computations are carried out in single precision allowing the use of the SSE instruction set on the processor. To discover possible memory contentions, we considered several memory layouts. The processor uses either 32-bit single precision and 64-bit double precision arrays. On GPU, we used single precision except for the last steps of the process where we used either single precision and 16-bit half precision.

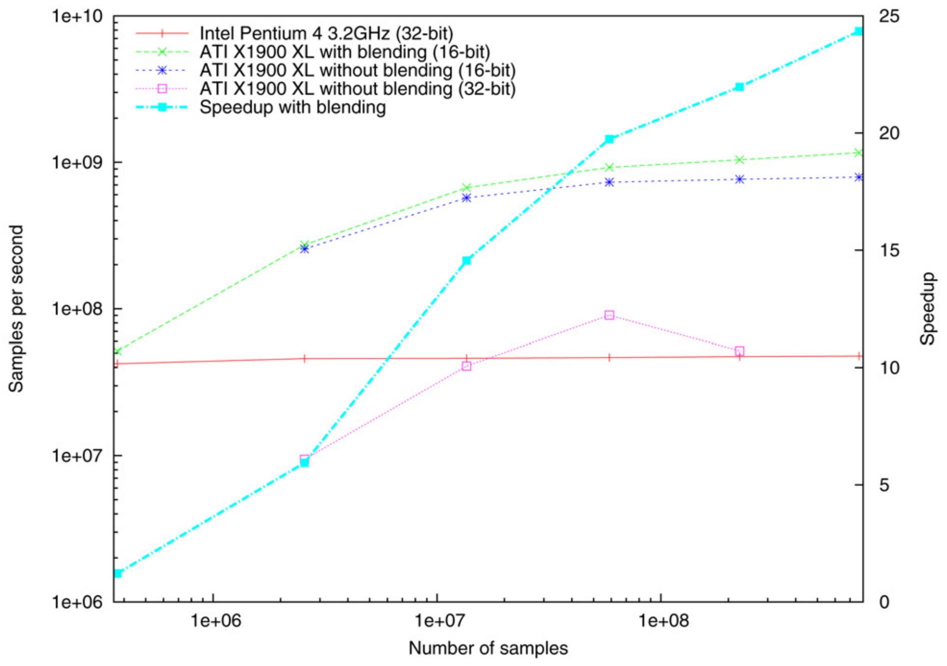

Fig. 8. First pass performance results on our second test configuration.

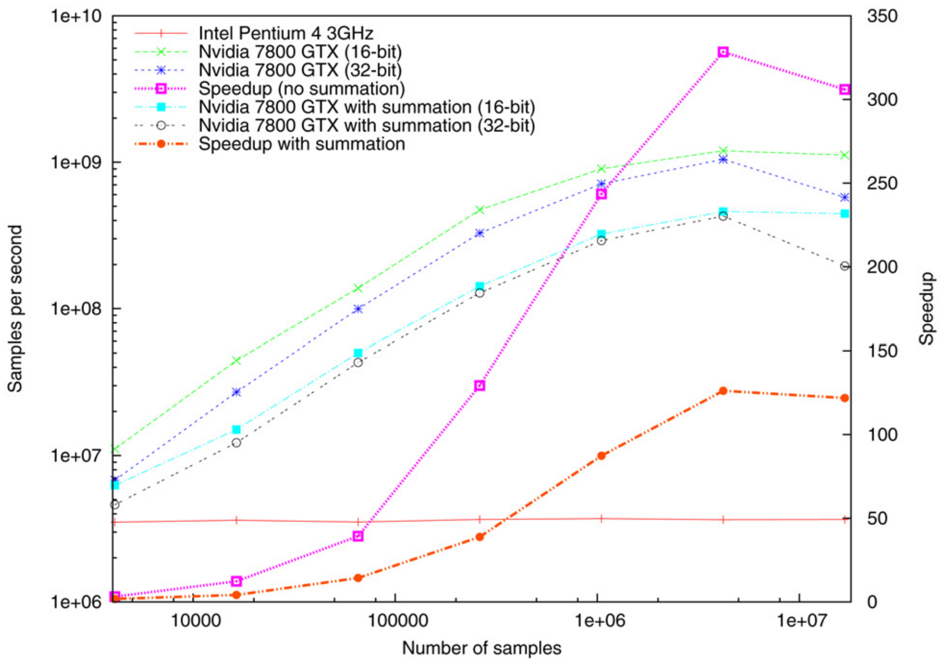

Fig. 9. Second pass performance results.

On one hand, GPUs are optimized for throughput over latency. The massive parallelism induced by the input data is used to maintain enough operations in flight such that dependencies between instructions do not cause pipeline stalls. On the other hand, higher data density also benefits the processor. It yields more spacial and temporal localities thus decreasing the ratio of cache misses.

The two-fold performance gap between the same computation performed in single precision and half precision textures reveals that this step is bounded by memory bandwidth on GPUs. Changing formats only causes marginal changes on performances of processors. We may assume that computing time on processor is bounded either by the computation time itself, by the memory latency, or by a combination of both.

This first pass involves divisions. Recent shaders contain a pipelined scalar unit responsible for evaluating special functions such as reciprocal, reciprocal square root, exponential, logarithm and trigonometric functions. This is not the case for most current processors. The reciprocal function is computed by fully-pipelined specialized units on GPUs delivering a result every cycle, whereas division is performed in microcode on the Pentium 4 processor and requires 40 cycles. This can cause the CPU to be computation-bound.

Emulating the blending operation in pixel shaders requires roughly twice as much bandwidth. This explains the twofold difference obtained between both methods for large datasets (more than 2×10^6 samples).

With these entirely bandwidth-dependent results in mind, we can assume a GPU with blending units operating on 32-bit data would operate roughly half as fast as one operating in 16-bit mode. This would represent a speedup around 20 to 30 compared to the CPU implementation.

Our second test configuration shows similar results when using 16-bit textures as seen Fig. 8. However, using 32-bit textures causes a major slowdown, GPU performance becoming even with processor performance. This is likely caused by a lack of graphics memory, which leads GPU and driver to swap graphics memory to main memory.

Likewise, neither of our two workstations was able to complete the last test with 32-bit textures, both experiments resulting in a crash of the application. This is due this time to the lack of system memory. Both test configurations contain 1 GB main DDR2 RAM and 256 MB graphics GDDR3 RAM. As more recent systems already feature up to 4 GB system RAM and 1 GB graphics RAM, these limitations will be pushed further as technology advances.

The second pass through the rendering pipeline performs operations presented in Section 4. Performance comparison between implementation on pixel shaders of nVidia GeForce 7800 GTX and our reference CPU are depicted in Fig. 9. The number of computed samples per second is plotted against the number of samples after merging. To obtain such timings, we changed the size of the texture used.

The second pass yields a speedup near 330 for 5×10^6 samples. One reason for these impressive results are the presence in GPUs of hardwired exponential evaluation units. On processors, transcendental functions such as the exponential are evaluated in software or microcode and typically require 100 clock cycles to compute. The nVidia GeForce 7800 GTX has 24 fully-pipelined function evaluation units and each unit can deliver one result per clock cycle. As with the reciprocal in the first pass, this allows the computation units to run at a speed approaching memory bandwidth.

However, the summation iteration brings the balance back toward the processor. Instead of needing a bandwidth-consuming multi-pass reduction scheme like the GPU, the processor only need an extra floating-point register. This register is used as an accumulator and the summation is almost an invisible operation.

For more than 2×10^7 samples, the complete data-set does not fit into graphics memory, especially when we use 32-bit textures. This leads to a noticeable performance decrease for the rightmost points of our figures.

6. Conclusion

Line-by-line simulation on real applications was long believed to be too expensive. We presented in this report our port of this task to graphics processing units so that line-by-line simulation performed on GPU can easily be coupled with simulation on fluid dynamics on processor. During our effort to port this application to GPU, we obtained a speedup close to 330 on computation intensive tasks and 12 on memory intensive tasks. Line-by-line simulation that requires more than 7 hours to complete on processor now runs in less than 10 minutes on GPU.

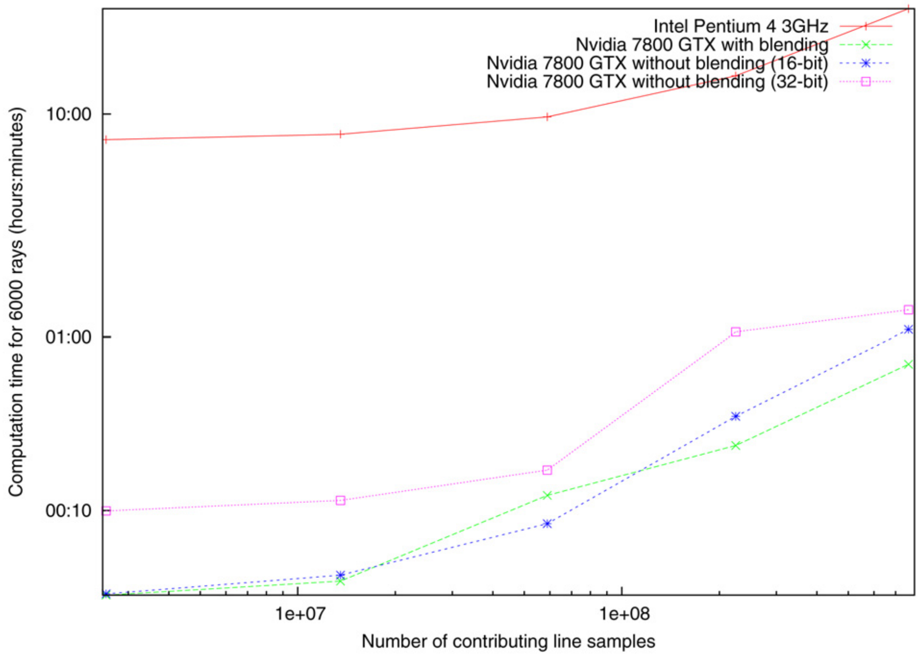

Fig. 10. Complete computation performance results.

Fig. 10 also presents running time for larger and more accurate simulations on 6000 rays entering various volumes.

As a conclusion, simulation of radiative phenomena coupled with simulation of fluid dynamics is now possible on real applications with some affordable additional resources compared to what is necessary to perform simulations on fluid dynamics alone. Future work will present new simulation and insight into physics obtained thanks to this new piece of software.

References

[1] L. Rothman, et al., Journal of Quantitative Spectroscopy and Radiative Transfer 96 (2005) 139.
[2] H. Partridge, D.W. Schwenke, The Journal of Chemical Physics 106 (1997) 4618.
[3] S.A. Tashkun, V.I. Perevalov, J.-L. Teffo, A.D. Bykov, N.N. Lavrentieva, Journal of Quantitative Spectroscopy and Radiative Transfer 82 (2003) 165.
[4] P. Perez, A. Boischot, L. Ibgui, A. Roblin, Journal of Quantitative Spectroscopy and Radiative Transfer 103 (2007) 231.
[5] L. Rothman, et al., Journal of Quantitative Spectroscopy and Radiative Transfer 60 (1998) 665.
[6] R.R. Gamache, S. Kennedy, R. Hawkins, L.S. Rothman, Journal of Molecular Structure (2000) 407.
[7] M. Pharr (Ed.), GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005.
[8] D. Manocha, IEEE Computer 38 (2005) 85.
[9] D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, fifth ed., Addison-Wesley Professional, 2005.
[10] Fluent, Fluent 5 User's Guide, 1998.
[11] Fluent, Fluent News 8 (1999) 9, 13.
[12] D.R. Horn, M. Houston, P. Hanrahan, ClawHMMER: a streaming HMMer-search implementation, in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Seattle, Washington, 2005, p. 9.