Implementing LNS using filtering units of GPUs

Mark G. Arnold, David Defour, Caroline Collange, David Defour Eliaus

▶ To cite this version:

Mark G. Arnold, David Defour, Caroline Collange, David Defour Eliaus. Implementing LNS using filtering units of GPUs. International Conference on Acoustics Speech and Signal Processing (ICASSP), Mar 2010, Dallas, TX, United States. pp.1542–1545, 10.1109/ICASSP.2010.5495516 . hal-00423434

HAL Id: hal-00423434
https://hal.science/hal-00423434

Submitted on 9 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IMPLEMENTING LNS USING FILTERING UNITS OF GPUS

Mark Arnold

Department of Computer Science
and Engineering,
Lehigh University
markgarnold@yahoo.com

Caroline Collange, David Defour

ELIAUS,
Université de Perpignan,
66860 Perpignan, France
david.defour@univ-perp.fr

ABSTRACT

Current GPUs offer specialized graphics hardware in addition to generic floating-point processing units. We propose a method which reuses specialized texture filtering units to perform piecewise polynomial evaluations, which helps accelerate LNS computations and can be used in combination with hardware-based transcendental functions.

Index Terms— Graphics processing units, logarithmic number system, texture filtering, polynomial approximation

1. INTRODUCTION

While the primary design goal of GPUs is efficient execution of graphics rendering, the massive parallelism available in these chips has opened up lately the possibility using them for more general signal-processing applications.

Current GPUs are mostly vector computers which focus on single-precision arithmetic, but they also offer graphics-related specialized units in hardware, such as texture filtering units and transcendental evaluation units. Development environments such as NVIDIA CUDA make it possible to program GPUs for various non-graphics applications. However, few such applications make use of texture filtering. We propose a novel approach to take advantage of these otherwise-idle texture filtering units to evaluate functions, which can be used to implement the Logarithmic Number System (LNS) on GPUs. In this paper we will consider the CUDA framework and the GT200 processor from NVIDIA, although our approach is generalizable to other GPUs.

We first give an overview of the GT200 GPU and LNS in the rest of this section, then present the proposed scheme in section 2, and discuss performance and accuracy in section 3.

1.1. Computational units in modern GPU

Figure 1 describes the hardware organization of computational units in the GT200 processor. The first level of the computational hierarchy corresponds to the Thread Processor Cluster (TPC). Up to 10 TPC are embedded in this GPU. Each TPC is made of one load/store unit (not represented on the figure), 1 texture unit and 3 multiprocessors.

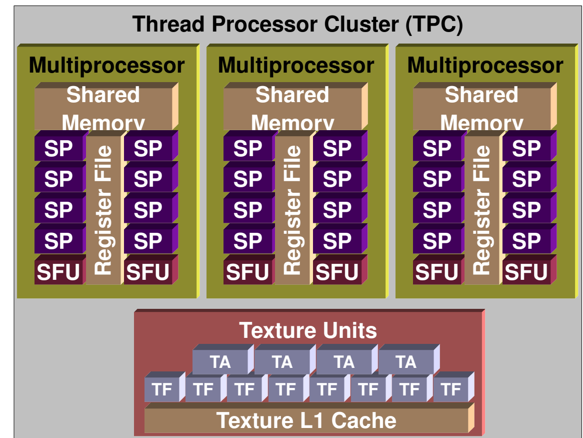

Fig. 1. Block diagram of one thread processor cluster of a NVIDIA GT200.

1.1.1. Multiprocessor units

Each multiprocessor embeds three kinds of vector units. Eight SP units perform single-precision multiply-and-adds. Two special function units (SFUs) compute reciprocals, reciprocal square roots, base-2 logarithms and exponentials, sines and cosines. Finally, one double-precision unit computes fused multiply-and-adds. All units are optimized for throughput and are accessed through 32-way SIMD instructions.

1.1.2. Texture Units

The texture unit or filtering unit is dedicated to accelerate memory accesses specific to graphics operations with uni-, two- or three-dimensional spatial locality. This unit is located in a different clock domain than the rest of the processor and is composed of 4 Texture Address units (TA) and 8 Texture Filtering units (TF). Spatial locality is exploited thanks to a dedicated 24 KB read-only texture cache.

The 4 TAs and 8 TFs are used to access texture elements called *texels* and optionally apply filtering (linear, bilinear,

and anisotropic). For example, a bilinear filter can be applied to eight 8-bit texels per clock or four FP16 or two FP32 per clock.

1.1.3. Bi-Linear filtering

In graphics languages as well as in CUDA, a texture object has several attributes. A texture can be declared as a uni-, two- or three-dimensional array that can be accessed with one, two or three texture coordinates. Fetched data can be 8-bit, 16-bit integer, 32 bit floating point numbers or a vector made of 1, 2 or 4 elements. The programmer can also choose between two addressing mode to access the texture by specifying whether texture coordinates are normalized (between 0 and 1) or not. Normalized textures are useful when the application requires texture addresses to be independent on the texture dimensions.

When a texture is configured to return floating-point data, the programmer can define the filtering mode to apply: nearest-point sampling or linear filtering. Linear filtering is a low-precision interpolation between neighboring texture data. When interpolation in enabled, texels surrounding a texture fetch location are read and used to interpolate values based on where the texture coordinates fall between the texels. Linear interpolation is performed for 1D, bilinear interpolation for 2D and trilinear for 3D texture.

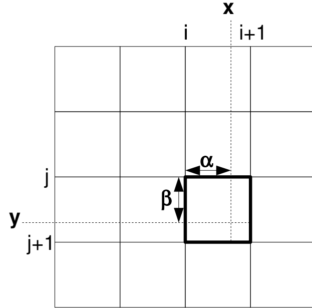

Fig. 2. Bilinear filtering of a two dimensional texture.

Let the two-dimensional texture T represent a $N \times M$ array of texels fetched using normalized floating-point texture coordinates x and y such that $x \in [0, N]$ and $y \in [0, M]$. The filtering unit returns the value V such that:

$$\begin{aligned} V \;=\; & (1-\alpha) \cdot (1-\beta) \cdot T[i,j] \\ & +(1-\alpha) \cdot \beta \cdot T[i+1,j] \\ & +\alpha \cdot (1-\beta) \cdot T[i,j+1] \\ & +\alpha \cdot \beta \cdot T[i+1,j+1] \end{aligned} \quad (1)$$

where

$$i = floor(x - 0.5) \text{ and } \alpha = frac(x - 0.5)$$

$$j = floor(y - 0.5) \text{ and } \beta = frac(y - 0.5)$$

It should be noted that α and β are manipulated in a fixed-point format with an 8-bit fractional part on NVIDIA GPUs. The hardwired linear texture filter is such that internal computations are performed with application-tailored precision, which limits the accuracy to the least significant bit of the texture data format.

1.2. Logarithmic Number System

The Logarithmic Number System represents a number, X, by its base-b logarithm, $x = \log_b |X|$, rather than the exponent and mantissa used in FP systems. Normally x is manipulated in fixed-point format giving precision and range like that of a similar FP system. In this paper, x itself may be stored in FP format, making the range of the LNS so astronomical that the problems of under/overflow disappear. For algorithms involving only positive real numbers, x by itself is sufficient to represent X; algorithms needing $X < 0$ require a sign bit like that in FP [1]. Zero is treated specially, and often in signal processing may be ignored. Products, quotients, roots and powers are trivial in LNS; sums and differences involve the functions $s_b(x) = \log_b(1+b^x)$ and $d_b(x) = \log_b |1-b^x|$. For example, since $Z + Y = Y(1 + Z/Y)$, LNS computes the sum as $\log(Z + Y) = y + s_b(z - y)$. This approach has been generalized to complex values [2] using logcos(θ), logsin(θ) and $\tan^{-1}(b^x)$ in addition to s_b and d_b. Complex LNS (CLNS) has been shown to be an efficient representation for the Fast Fourier Transform (FFT) [3].

Before the widespread adoption of DSP chips, LNS offered attractive VLSI implementations for early signal processing hardware with limited-precision lookup of s_b and d_b from ROMs. A bibliography [4] lists around twenty papers from that era appearing in ICASSP and the *Transactions* on signal processing using LNS. Stouraitis [5] gives a good summary of the state of the art in LNS for signal processing in the 1980s. Later, LNS found other niche applications, such as the massively-parallel N-body computation that won the Gordon Bell prize in 1999 [6]. LNS continues to be an attractive alternative number system for FPGAs [7] because VHDL libraries [8, 9] are now available that implement polynomial approximations to s_b and d_b, and LNS is attractive for complex signal-processing algorithms [10]. LNS is common in computing Hidden Markov Models (HMMs) [11], even on conventional CPUs, because a summation of exponentials is easier to compute in LNS than FP.

With the massive FP hardware available on GPUs, implementing LNS may seem unnecessary, but like all chip sets, NVIDIA chips have fixed resources. In their original intent for accelerating graphics, the designers at NVIDIA felt it worthwhile to commit silicon to texture interpolation, hardware that typically goes unused in a GPGPU signal-processing application. This paper considers using the texture hardware to approximate s_b and related functions that implement either real or complex LNS for portions of applications, like HMM or N-body, where LNS has proven to be useful even in the presence of FP hardware, and thereby improve

throughput using LNS format. The architectural complexity of GPUs make it difficult to predict when such LNS will benefit a particular application; disclosing our proposed technique gives GPGPU programmers the option to experiment with LNS.

2. PROPOSED METHOD

We want to evaluate $f(x)$, for instance s_b or d_b. We assume $x \in [0,1]$ is in IEEE-754 single precision format. There exist several techniques to evaluate a function on a given interval. Due to the presence of filtering on GPUs, spline and B-Spline are very common. Sigg et al. described in [12] how to execute fast third order texture filtering while taking advantage of bilinear filtering. Another technique consists in approximating a function by one or several polynomials evaluated with SP units. The main advantage of this technique is that it is very accurate as every operation is done using floating-point arithmetic. However, it consumes computational resources to perform the higher-order interpolation.

2.1. Order-2 polynomial approximation

The proposed method consists in performing polynomial evaluation inside the texture units, whenever low input precision with high output accuracy is necessary. We propose to approximate $f(x)$ with a piecewise degree 2 polynomial approximation on $[0,1]$ with k polynomials. Let P_w be the polynomial used to approximate $f(x)$ on $[w.2^{-k},(w+1).2^{-k}]$:

$$P_w(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 \quad (2)$$

The novelty of this methods lies in setting the correct parameters in the texture T and the address used to fetch data such that equation 1 representing a bilinear filtering operation, is similar to the evaluation of the polynomials P_w.

If we perform a texture fetch at coordinates $(x+0.5,(x-floor(x))+0.5)$ then (1) is equivalent to:

$$\begin{aligned} V &= (1-\gamma)^2 \cdot T[i,0] + (\gamma-\gamma^2)\cdot T[i+1,0] \\ &\quad +(\gamma-\gamma^2)\cdot T[i,1] + \gamma^2 \cdot T[i+1,1] \end{aligned} \quad (3)$$

with $\gamma = frac(x), i = floor(x)$

Now, let us focus on texture coefficient such that the texture T store $T[i,0] = a_0$, $T[i+1,0]+T[i,1] = 2\cdot a_0 + a_1$ and $T[i+1,1] = a_0+a_1+a_2$. We have:

$$\begin{aligned} V &= (1-\gamma)^2.a_0 + (\gamma-\gamma^2).(2.a_0+a_1) \\ &\quad +\gamma^2.a_0 + a_1 + a_2 \\ &= a_0 + a_1.\gamma + a_2.\gamma^2 \end{aligned} \quad (4)$$

Therefore $T[i,0]$, $T[i+1,0]$, $T[i,1]$ and $T[i+1,1]$ are used to store the coefficients of one polynomial. In order to store the coefficients of k polynomial approximations of $f(x)$ on k interval, the $4.k$ coefficients will be stored on a $2.k \times 2$ texture. In that case, fetch will be performed at the coordinates illustrated by this CUDA code:

```
__device__ float text_eval_deg2(float x, int
   k){
   x = x * k;
   float xe = floor(x);
   float xf = x - xe + .5f;

   return tex2D(texRef, (2.0f*xe) + xf, xf);
}
```

2.2. Discussion

Even though texture units have their own texture address unit, polynomial coefficients are stored in a way that requests some adjustment performed with the help of SP units. However with this technique, the execution cost of the evaluation of one function at x is almost equivalent to the evaluation cost of 4 completely different functions at the point x thanks to texture filtering units that works either on one floating point number or a vector of up to 4 floating-point numbers. In addition, one may observe that the address shifting by 0.5 can be avoided by including this bias into the polynomial approximation rather than including it in address calculations.

3. RESULTS AND VALIDATION

We implemented the degree-2 method described above, using the Sollya tool to generate optimized minimax polynomials with single-precision coefficients [13]. For comparison purposes, we also implemented an order-1 method that uses only linear texture filtering, a computation that takes advantage of the hardware-based exp and log in the SFU, and a direct polynomial approximation evaluated using the Horner scheme, also generated using Sollya.

Figure 3 describes the worst-case error on the s_b function for the methods implemented. We consider an input precision f such that $k = 2^{f-8}$, so the 8-bit quantized fractions of coordinates γ are exact.

We remark that storing 64 segments is necessary to provide 23 bits of output accuracy. Such storage requirements are low enough that coefficients can all fit inside the texture cache.

We evaluated the performance of each method, and combinations of them, to evaluate either the s_b function or both s_b and d_b at the same point on a GeForce GTX 280 GPU. Results are normalized and expressed in shader clock cycles per warp, rounded to the nearest half-cycle and presented in Table 1.

When using hardware transcendentals to compute both s_b and d_b as $\log_2(1 \pm 2^x)$, compiler optimizations enable the common 2^x calculation to be computed only once. Though order-2 texture interpolation provides no benefit when used by itself or to evaluate single functions, it increases the computation throughput of s_b and d_b when used in combination with the SFU units.

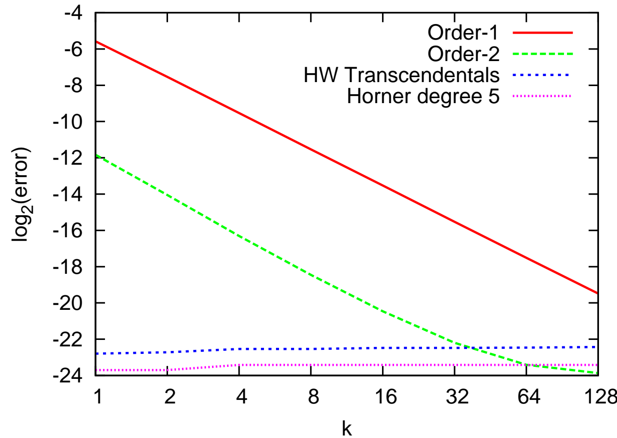

Fig. 3. Comparison of the worst-case error of the proposed methods on the s_b function, with $k = 2^{f-8}$.

Table 1. Performance comparison of the proposed methods and other implementations on a GeForce GTX 280, in clock cycles.

Method	s_b	s_b and d_b
Order-1	42.5	42.5
Order-2	49.0	49.0
Order-3	49.5	49.5
Hardware transcendentals	32.0	33.0
Horner degree 5	35.0	44.0
Order-2 + HW transcendentals	46.0	52.5
2× HW transcendentals	40.0	56.0

4. CONCLUSION

We presented a scheme that leverages the specialized texture-filtering units on GPUs to perform function evaluation, enabling a higher throughput for the functions that form the basic blocks of LNS. Other applications that require function approximation with a high output accuracy for a low input precision could also be accelerated.

The most prominent obstacle for accurate evaluation being the quantization of the inputs of the filtering unit, we make a case for the exposure of the hardware attribute interpolation unit in the CUDA environment. Direct access to the attribute interpolator, which does not suffer from the accuracy limitations of the filtering unit, would allow accurate evaluation of arbitrary functions, as already implemented in hardware for a limited set of functions [14].

5. REFERENCES

[1] E.E. Swartzlander et al., "Sign/logarithm arithmetic for FFT implementation," *IEEE Transactions on Computers*, vol. C-32, pp. 526–534, 1983.

[2] M. G. Arnold and C. Collange, "A dual-purpose real/complex logarithmic number system ALU," in *19th IEEE Symposium on Computer Arithmetic*, 2009, p. 15–24.

[3] M. Arnold et al., "Fast fourier transforms using the complex logarithm number system," *J. VLSI Signal Proc.*, vol. 33, pp. 325–335, 2003.

[4] ," `www.xlnsresearch.com`.

[5] T. Stouraitis, *Logarithmic Number System Theory, Analysis, and Design*, Ph.D. thesis, Univ. of Florida, Gainesville, 1986.

[6] J. Makino and M. Taiji, *Scientific Simulations with Special-Purpose Computers: The GRAPE Systems*, John Wiley & Son Ltd., 1998.

[7] C. Collange, F. de Dinechin, and J. Detrey, "Floating point or LNS: Choosing the right arithmetic on an application basis," in *EuroMicro Digital System Design DSD*, 2006, pp. 197–203.

[8] J. Detrey and F. de Dinechin, "A VHDL library of LNS operators," in *37th Asilomar Conference on Signals, Systems, and Computers*, Nov 2003, vol. 2, pp. 2227–2231, `www.ens-lyon.fr/LIP/Arenaire`.

[9] P. Vouzis, C. Collange, and M. G. Arnold, "LNS subtraction using novel cotransformation and/or interpolation," in IEEE 18th International Conference on Application-specific Systems, Architectures and Processors, 2007, pp. 107–114.

[10] F. Albu et al., "Pipelined implementations of the a priori error-feedback LSL algorithm using logarithmic number system," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2002, vol. 3, pp. 2681–2684.

[11] S. Young et al., *The HTK Book (for HTK Version 3.4.1)*, Cambridge University Engineering Dept., 2009, `http://htk.eng.cam.ac.uk/`.

[12] Matt Pharr, Ed., *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, 2005.

[13] N. Brisebarre and S. Chevillard, "Efficient polynomial L-∞ approximations," in *18th IEEE Symposium on Computer Arithmetic ARITH '07*, June 2007, pp. 169–176.

[14] Stuart F. Oberman and Michael Siu, "A high-performance area-efficient multifunction interpolator," in *17th IEEE Symposium on Computer Arithmetic*, Koren and Kornerup, Eds., Los Alamitos, CA, July 2005, pp. 272–279.