

A GPU Implementation of the Complex Logarithmic Number System

Panagiotis Vouzis
Dept. of Chemical Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Email: pvouzis@cmu.edu

Mark Arnold
Dept. of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015, USA
Email: maab@lehigh.edu

Abstract—In this paper we present a technique to implement the Complex Logarithmic Number System (CLNS) on a Graphics Processing Unit (GPU). Although CLNS multiplication is a simple FP addition, CLNS addition involves evaluations of transcendental functions, which can be carried out in a few different ways by utilizing the GPU hardware resources, such as the special function units, the floating point units, or the texture memory. We propose the implementation of CLNS by using the linear interpolation capabilities of the, otherwise unused, GPU texture memory. In the context of an algorithm that involves complex arithmetic the CLNS additions can be overlapped with other computations by offloading them to the texture memory saving clock cycles. The proposed technique has been implemented on an Nvidia GT200 GPU with CUDA, and its clock-cycle requirements and error behavior are presented and analyzed.

I. INTRODUCTION

The use of Graphics Processing Units (GPUs) for general-purpose computing has gained broad attention in the last few years due to the speedup that can be achieved for many scientific-computing applications. Areas such as digital-signal processing, molecular-dynamic simulations, basic linear-algebra subroutines can benefit from the use of GPUs since they can be easily parallelized on a vector computer [1].

Initially, GPUs were custom-designed coprocessors dedicated to the acceleration of graphics algorithms. However, they were made more and more programmable and today they are general-purpose computing machines, without having lost their graphics rendering capabilities. The introduction of development environments, such as the Nvidia CUDA, expose to the programmer the memory hierarchy and the computational resources of a GPU, and give the freedom for their utilization in the best possible fashion to achieve the desired performance requirements.

Arnold et al. in [2] present a technique to carry out arithmetic operations in the Logarithmic Number System (LNS) [3] by utilizing the interpolation capabilities of the GPU texture memory. In this work we explore the use of the texture memory for the implementation of the Complex Logarithmic Number System (CLNS) [4], which is a generalization of the LNS. CLNS represents numbers with their logarithms simplifying the operations of multiplication and division, but makes more complicated the operations of addition and subtraction. The properties of LNS and CLNS have been shown to lend

themselves to certain kinds of applications. Stouraitis in [5] gives an overview of the application spectrum of LNS in signal processing, and Paliouras et. al in [6] show the potential reduction in power dissipation by using LNS compared to an equivalent linear representation. The most successful effort to incorporate LNS in general-purpose computing has been done in the context of the European Logarithmic Processor which encompasses a 32-bit LNS arithmetic logic unit that demonstrates a "better than FP" accuracy with improved speed [7]. A potential application of CLNS is the Fast Fourier Transform (FFT) since it requires the processing of complex numbers; thus the use of the CLNS is an attractive alternative, since it has been proven to be significantly more compact and requires fewer bits for the same accuracy than a comparable fixed-point representation [8]. Two different techniques to reduce the memory requirements of CLNS through transformations are presented in [9] and [10].

The CLNS addition and subtraction require the use of 2D Lookup Tables (LUTs) which can make a CLNS implementation with high-memory requirements slow and inefficient to implement on a limited-resource platform. GPUs, however, have a very efficient memory architecture to facilitate the heavy bandwidth requirements of graphics rendering, which can also be used for general-purpose applications, such as the implementation of the CLNS addition and subtraction operations. We present a technique to take advantage of the filtering capabilities inherent in a GPU texture memory, and the ways that this idea could be beneficial in an application.

The rest of the paper is organized as follows. The next section gives an overview of the GPU architecture, and Section III introduces the CLNS representation. Section IV presents the implementation of CLNS on the GPU, together with the clock-cycle and error-behavior results. Finally, conclusions are drawn in the final Section.

II. GPU ARCHITECTURE

GPUs are massively parallel computing machines that consist of basic building blocks that are repeated multiple times to create the degree of parallelism desired for a specific model. The GPU used in this paper is the GT200 from Nvidia which consists of ten Thread Processing Clusters (TPC) that consist

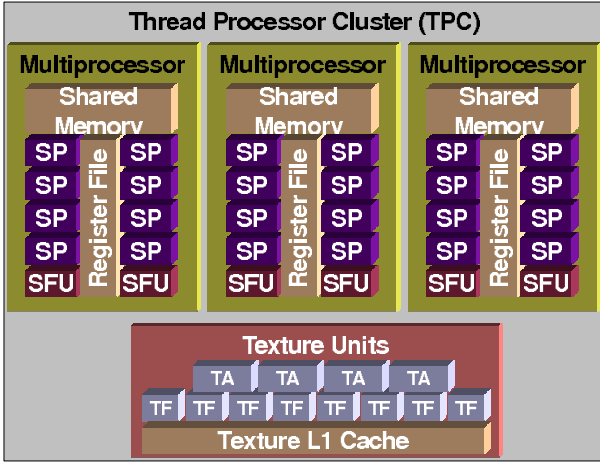


Fig. 1. Block diagram of one thread processor cluster of an Nvidia GT200.

of a load/store unit, one texture unit, and three multiprocessors as depicted in Fig. 1 [2].

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 using 32-way SIMD instructions. The shared memory present in each multiprocessor can be perceived as cache memory, but its effective utilization depends on the programmer; as opposed to modern CPUs where the cache functionality is implemented in hardware and is completely hidden from the programmer.

The texture unit or filtering unit is a dedicated unit used to accelerate memory accesses specific to graphics operations with mono-, 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 texel per clock or four FP16 or two FP32 per clock.

A. Bi-Linear Filtering

In graphics languages as well as in CUDA, a texture object has several attributes. A texture can be declared as a mono-, 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 modes 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 of the texture size.

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 is 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.

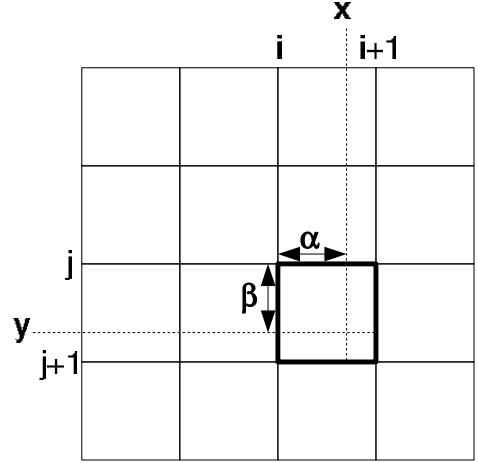


Fig. 2. Bilinear interpolation of a two-dimensional texture.

Let the two-dimensional texture T , depicted in Fig. 2, 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:

$$V(\alpha, \beta) = (1 - \alpha) \cdot (1 - \beta) \cdot T[i, j] + (1 - \alpha) \cdot \beta \cdot T[i + 1, j] + \alpha \cdot (1 - \beta) \cdot T[i, j + 1] + \alpha \cdot \beta \cdot T[i + 1, j + 1] \quad (1)$$

where

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

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

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.

III. LOGARITHMIC REPRESENTATIONS

In LNS, a real number, X , is represented by the logarithm of its absolute value,

$$x = \text{round}(\log_b(|X|)), \quad (2)$$

and an additional bit, s , denoting the sign of the number X where $s = 0$ for $X > 0$, $s = 1$ for $X < 0$, and b is the

base of the Logarithm (typically two). The $\text{round}()$ operation approximates x so that it is representable by $N = K + F$ bits in two's complement format. The number of integer K and fractional F bits is a design choice that determines the dynamic range and the accuracy respectively. In this work we assume that numbers are represented in single-precision Floating Point (FP) format $K + F = 23$.

In LNS real-number multiplication and division are simplified considerably, compared to a fixed-point and FP representation, since they are converted to addition and subtraction. The operations of logarithmic addition and subtraction, though, are more expensive, and they account for most of the delay and area cost of an LNS implementation. A simple algorithm usually used is described by:

$$\begin{aligned} \log_b(|X| + |Y|) &= \log_b(|X|(1 + \frac{|Y|}{|X|})) = x + s_b(z) \\ \log_b||X| - |Y|| &= \log_b(|X|(1 - \frac{|Y|}{|X|})) = x + d_b(z). \end{aligned} \quad (3)$$

where $z = |x - y|$, $s_b = \log_b(1 + b^z)$ and $d_b = \log_b(1 - b^z)$.

A naïve implementation for these functions is to calculate analytically their values for each input z . However, a technique that maps better on the available resources on a GPU is presented in [11] where the authors take advantage of the texture filtering units available on modern GPUs in order to evaluate $s_b(z)$ and $d_b(z)$.

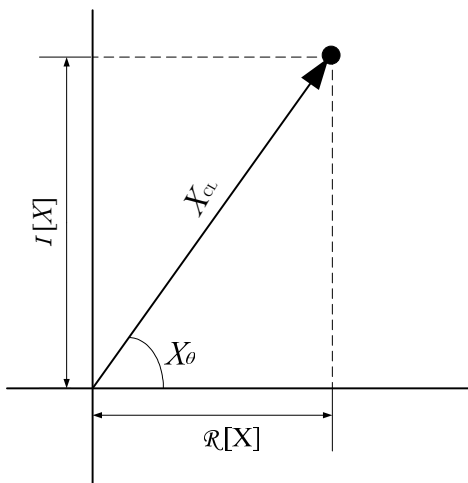


Fig. 3. The log/polar representation in CLNS.

This technique is extended here for the implementation of CLNS, which represents a complex value, X , with a log-polar pair, $X_{CP} = (X_{CL}, X_{\theta})$ as depicted in Fig. 3, whose components are:

$$\begin{aligned} X_{CL} &= 0.5 \cdot \log_b(\Re[X]^2 + \Im[X]^2) \\ X_{\theta} &= \arctan(\Re[X], \Im[X]). \end{aligned} \quad (4)$$

On output, this log-polar pair is converted back to rectangular coordinates that describe the same value:

$$\Re[X] = b^{X_{CL}} \cos(X_{\theta}), \quad \Im[X] = b^{X_{CL}} \sin(X_{\theta}). \quad (5)$$

CLNS multiplication and division are easy, e.g., to divide X by Y :

$$Z_{CL} = X_{CL} - Y_{CL}, \quad Z_{\theta} = (X_{\theta} - Y_{\theta}) \bmod 2\pi. \quad (6)$$

CLNS addition uses $T = X + Y = Y(Z + 1)$, where $Z = X/Y$. This needs two subtractions, two memory accesses, and two additions:

$$\begin{aligned} T_{CL} &= Y_{CL} + \Re[S_b(Z_{CP})] \\ T_{\theta} &= Y_{\theta} + \Im[S_b(Z_{CP})] \bmod 2\pi, \end{aligned} \quad (7)$$

where Z_{CP} is given by (6). $S_b(Z_{CP})$ implements i) conversion of Z_{CP} via (5); ii) incrementing this; and iii) conversion back via (4). These steps translate to the following expressions for the real and imaginary parts of $S_b(Z_{CP})$:

$$\Re[S_b(Z_{CP})] = 0.5 \cdot \log_b(1 + 2b^{Z_{CL}} \cos(Z_{\theta}) + b^{2 \cdot Z_{CL}}) \quad (8)$$

$$\Im[S_b(Z_{CP})] = \arctan(1 + b^{Z_{CL}} \cos(Z_{\theta}), b^{Z_{CL}} \sin(Z_{\theta})). \quad (9)$$

Because of commutativity, we may assume $0 \leq Z_{\theta} \leq \pi$.

IV. CLNS IMPLEMENTATION ON THE GPU

The straightforward implementation to carry out complex-number computations on a GPU is in rectangular coordinates which require two FP additions for complex addition, and four multiplications and three additions for complex multiplication, with each one requiring four clock-cycles. The alternative explored in this paper is the utilization of the texture memory for the CLNS implementation of the operation of addition and subtraction.

With the massive FP hardware available on GPUs, implementing CLNS may seem unnecessary, but like all chip sets, Nvidia chips have fixed resources. In their original intent for accelerating graphics, the designers at Nvidia committed silicon to texture to texture interpolation, hardware that typically goes unused in a GPGPU signal-processing application. This paper considers using the texture hardware to approximate $S_b(Z_{CP})$ that implement CLNS for portions of applications, like FFT, where CLNS has proven to be useful, even in the presence of FP hardware, and thereby improve throughput using CLNS format. The architectural complexity of GPUs make it difficult to predict when such CLNS will benefit a particular application; disclosing our proposed technique gives GPGPU programmers the option to experiment with CLNS.

For the interpolation of $S_b(Z_{CP})$, the function is evaluated at P_{CL} uniformly distributed points of $Z_{CL} \in [0, 1]$, and P_{θ} uniformly distributed points of $Z_{\theta} \in [0, \pi]$ since Z_{θ} is symmetric in $[0, \pi]$ with respect to π . If we assume that the input and the output of the texture memory are single-precision floating point numbers which require four bytes for their representation, the number of bytes needed to store both $\Re[S_b(Z_{CP})]$ and $\Im[S_b(Z_{CP})]$ is equal to $w = 2 \cdot 4 \cdot P_{CL} \cdot P_{\theta}$.

The implementation of this method consists mainly of storing the appropriate values of $S_b(Z_{CP})$ in the texture, and the evaluation step clock-cycle cost is just the memory fetch. Using the texture memory has a two-fold advantage: (a) the accessing is done through a read-only cache memory which exploits spatially locality provided that w is smaller than the

TABLE I
CLOCK-CYCLE COMPARISON OF THE PROPOSED METHODS AND OTHER
IMPLEMENTATIONS ON A GEFORCE GTX 280.

Operation	Clock Cycles per Warp
Complex FP addition	35.0
Complex FP multiplication	37.0
CLNS addition	42.0
CLNS multiplication	35.0

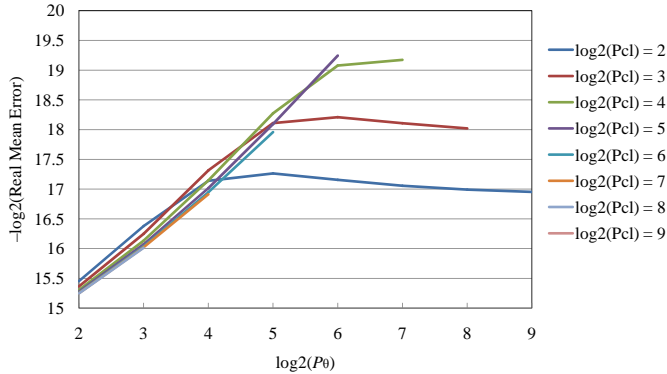


Fig. 4. The mean error of $\Re[S_b(Z_{CP})]$.

cache size, and (b) the interpolation comes at zero latency cost due to the dedicated TF units. Moreover, the stored values of $S_b(Z_{CP})$ are represented in the CUDA built-in vector type of float2 which has two components, x and y , which can be retrieved in one memory access saving an address generation and a memory access.

The results in Table I are from the implementation of both methods on a GTX 280 GPU, and they are expressed in shader clock cycles per warp, rounded to the nearest half-cycle, and they include the clock cycles of generating the addresses and other related functionalities necessary to carry out the computations. CLNS multiplication takes the same number of clock cycles with the complex FP addition, and the required clock cycles for CLNS addition are equal to 42 clock cycles per warp. We can see that CLNS addition takes more cycles than FP addition or multiplication; however, in an application that has enough FP operations to overlap with the evaluation of $S_b(Z_{CP})$ the total throughput of the computation can increase using hardware that would otherwise go unused. Increasing the number of points P_{CL} and P_θ increases the clock-cycle count per warp, but at the same time decreases the maximum and the mean error induced by the interpolation of $S_b(Z_{CP})$ as is shown in Fig. 4 and 5 for the real and the imaginary part of $S_b(Z_{CP})$. The presented CLNS clock cycles and error-behavior results are only for values of P_θ and P_{CL} that require a memory that fits in the texture cache, otherwise the clock-cycle count increases substantially, thus $w \leq 24KB \Rightarrow \log_2(P_\theta) + \log_2(P_{CL}) < 11$.

V. CONCLUSIONS

GPUs are parallel computing machines that have several types of computational units and a memory hierarchy which

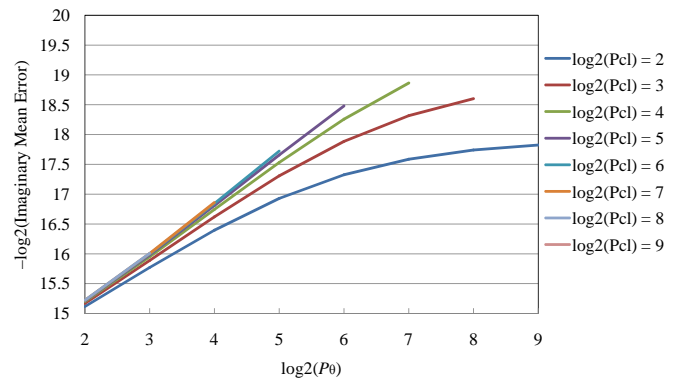


Fig. 5. The mean error of $\Im[S_b(Z_{CP})]$.

can be tailored to a broad range of applications. In this paper we presented a technique to utilize the texture memory of a GPU for the implementation of CLNS. Although there might be more straightforward ways to carry out complex arithmetic on a GPU than using CLNS, the proposed implementation method has the potential of overlapping with other computations since it is based on the texture units which are in a different clock domain than the rest of the GPU resources. A careful utilization of CLNS in the context of an algorithm can reduce the total clock-cycle budget and result in performance improvement.

REFERENCES

- [1] C. Lin and D. Manocha (eds.), "Special Issue on Cutting-Edge Computing: Using New Commodity Architecture," *IEEE Proceedings*, vol. 96, no. 5, pp. 753–899, 2008.
- [2] M. Arnold, S. Collange, and D. Defour, "Implementing LNS using filtering units of GPUs." [Online]. Available: <http://hal.archives-ouvertes.fr/hal-00423434>
- [3] E. E. Swartzlander and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1238–1242, Dec. 1975.
- [4] M. G. Arnold, T. A. Bailey, J. R. Cowles, and M. D. Winkel, "Arithmetic Co-transformations in the Real and Complex Logarithmic Number Systems," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 777–786, July 1998.
- [5] T. Stouraitis, "Logarithmic Number System Theory, Analysis, and Design," Ph.D. dissertation, Univ. of Florida, Gainesville, Florida, 1986.
- [6] V. Paliouras and T. Stouraitis, "Low-power properties of the Logarithmic Number System," in *Proceedings of 15th Symposium on Computer Arithmetic ARITH15*, Vail, CO, 11–13 June 2001, pp. 229–236.
- [7] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The European Logarithmic Microprocessor," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 532–546, April 2008.
- [8] M. Arnold, T. Bailey, J. Cowles, and C. Walter, "Fast Fourier Transform Using the Complex Logarithmic Number System," *Journal of VLSI Signal Processing*, vol. 33, no. 3, pp. 325–335, 2003.
- [9] M. G. Arnold and S. Collange, "A Dual-Purpose Real/Complex Logarithmic Number System ALU," in *19th IEEE Symposium on Computer Arithmetic*, Portland, OR, 8–10 March 2009.
- [10] P. Vouzis and M. G. Arnold, "A Parallel Search Algorithm for CLNS Addition Optimization," in *The IEEE International Symposium on Circuits and Systems*, Kos, Greece, 21–24 May 2006, pp. 2147–2420.
- [11] M. G. Arnold, S. Collange, and D. Defour, "Implementing LNS Using Filtering Units of GPUs," in *submitted to the IEEE International Conference on Acoustics, Speech and Signal Processing*, Dallas, TX, 14–19 March 2010.