計算数理工学論文集 Vol. 11 (2011 年 12 月), 論文 No. 06-111216 JASCOME

GPU 上の多倍長数値計算環境による複素逆 Laplace 変換の高速化

ACCELERATION OF NUMERICAL COMPUTATION OF THE COMPLEX INVERSE LAPLACE TRANSFORM USING A MULTIPLE PRECISION SYSTEM ON GPU

眞鍋 秀悟 [1)], 藤原 宏志 [2)]

Shugo MANABE and Hiroshi FUJIWARA

1) 京都大学大学院 情報学研究科 (〒 606-8501 京都市左京区吉田本町, E-mail: shugo@acs.i.kyoto-u.ac.jp)
2) 京都大学大学院 情報学研究科 (〒 606-8501 京都市左京区吉田本町, E-mail: fujiwara@acs.i.kyoto-u.ac.jp)

Multiple precision systems are applied to numerical computation of the complex inverse Laplace transform based on the Bromwich integral because of the numerical instability. We design and implement a multiple precision system for vector operations on GPU (Graphics Processing Unit). This paper represents acceleration of the inverse Laplace transform using the proposed GPU system.

Key Words: GPGPU, 逆 Laplace 変換, 多倍長計算

1. 序

本論文は, GPU (Graphics Processing Unit) を利用した高速なベクトル演算をもつ多倍長数値計算環境の設計および実装を行い, ベクトル演算に適したスキームである複素逆 Laplace 変換の高速化を示すものである.

Laplace 変換

$$\mathcal{L}f(s) = F(s) = \int_0^\infty e^{-st} f(t)\, dt$$

は力学, 工学, 情報学, 数理経済学など多くの分野で現れる. 同時に逆 Laplace 変換も重要な役割を果たしており [(1)], その数値計算法も幾つか提案されている [(2, 3, 4)]. これらの方法は大別して, Laplace 変換像 $F(s)$ の $s > 0$ における値を直接利用する実逆変換 [(5)] と Bromwich 積分に基づく複素逆変換に分類できる.

原像 f が $M > 0$ に対して $|f(t)| \leq e^{Mt}$ を満たすとき, Laplace 変換像 $F(s)$ は複素平面における $\mathrm{Re}\, z > M$ で正則な正則函数 $F(s)$ に解析接続される. $\sigma > M$ とすれば, Bromwich 積分による逆変換

$$f(t) = \frac{1}{2\pi i} \lim_{T\to\infty} \int_{\sigma-iT}^{\sigma+iT} e^{tz} F(z)\, dz \qquad (1)$$

が成立する [(2)].

複素逆 Laplace 変換は数値的に不安定であり, 丸め誤差の取扱いに注意が必要である [(6)]. そのため多倍長数値計算環境の適用により, 丸め誤差の影響を抑えた. しかし現存の多倍長数値計算環境の演算速度は遅く, 本論文では GPU を利用することで, 従来よりベクトル演算が高速な環境を構築した.

2011 年 9 月 30 日受付, 2011 年 11 月 10 日受理

本論文は GPU のもつ高い並列処理能力に着目し, ベクトル演算に適した多倍長数値計算環境の実現を行ったものである. GPU は画像描画用途のプロセッサであり, その用途から, 高い並列処理能力をもつプロセッサである. GPU の性能の進展や, GPU に対するプログラミング環境の整備の進展から近年 GPU を用いた科学技術計算の高速化が行われている. こうした GPU を利用した汎用計算を GPGPU (general-purpose computing on graphics processing unit) という. 本研究はこの GPGPU に着目した研究である.

以下, 2 節で複素逆 Laplace 変換スキームの一つである細野の方法の紹介を行う. 続いて 3, 4 節では GPU を用いた多倍長数値計算環境の設計と実装, その特徴について論じる. 最終 5 節では, 複素逆 Laplace 変換はベクトル演算に適したスキームであることを示し, GPU を用いて構築した環境を利用することで複素逆 Laplace 変換の高速化を行う. ここでは複素逆 Laplace 変換として細野の方法 [(7)] を例に示した.

2. 細野の方法

Bromwich 積分 (1) の核函数 e^s は

$$E_{ec}(s, \sigma_0) = \frac{e^{\sigma_0}}{2\cosh(\sigma_0 - s)}$$

と定めると,

$$e^s = \lim_{\sigma_0\to\infty} E_{ec}(s, \sigma_0) = \lim_{\sigma_0\to\infty} \frac{e^{\sigma_0}}{2\cosh(\sigma_0 - s)}$$

を満たす. また,

$$E_{ec}(s, \sigma_0) = \frac{ie^{\sigma_0}}{2} \lim_{N\to\infty} \sum_{n=-N}^{N} \frac{(-1)^n}{s - \left(\sigma_0 + i\left(n - \frac{1}{2}\right)\pi\right)}$$

であることから, 留数定理を用いて,

$$\frac{1}{2\pi i}\lim_{T\to\infty}\int_{\sigma-iT}^{\sigma+iT} E_{ec}(st,\sigma_0)F(z)\,dz$$
$$=\frac{e^{\sigma_0}}{t}\sum_{n=1}^{\infty}(-1)^n \operatorname{Im} F\left(\frac{1}{t}\left\{\sigma_0+i\left(n-\frac{1}{2}\right)\pi\right\}\right) \qquad (2)$$

を得る. 細野の方法は核関数 e^s を $E_{ec}(s,\sigma_0)$ で近似した (2) に対して Euler 変換を適用して得られるスキームであり, 原像 f は

$$f(t)\approx\frac{e^{\sigma_0}}{2}\left(\sum_{n=1}^{k-1}F_n(t)+\frac{1}{2^{\mu+1}}\sum_{\nu=0}^{\mu}A_{\mu,\nu}F_{k+\nu}(t)\right) \qquad (3)$$

で与えられる. ただし,

$$F_n(t)=(-1)^n\operatorname{Im}F\big(z_n(t)\big)$$
$$z_n(t)=\frac{1}{t}\left\{\sigma_0+\left(n-\frac{1}{2}\right)\pi i\right\}$$
$$A_{\mu,\nu}=1,\quad A_{\mu,\nu-1}=A_{\mu,\nu}+\binom{\mu+1}{\nu}$$
$$N=k+\mu$$

である. N は打ち切りのパラメータであり, μ は Euler 変換に現れるパラメータである.

細野の方法などの複素逆 Laplace 変換は [6] で論じられるように数値不安定性を持つ. これに対し, 多倍長数値計算環境を適用することにより丸め誤差の影響を軽減し, 不安定スキームの直接計算が実現される [6].

3. GPU を用いた多倍長数値計算環境の設計と実装

複素逆 Laplace 変換の数値的不安定性のため, 本論文は GPU を用いた多倍長数値計算環境を構築した. 本節では, この環境の設計および実装を論じる.

多倍長数の表現方法としては, 大別して固定小数点方式と浮動小数点方式があるが, 本研究は多倍長数として固定小数点方式を採用した. 浮動小数点演算は条件分岐が必要な処理が多く, 分岐命令により GPU のパフォーマンスが著しく落ちる. このため GPU は多倍長の浮動小数点演算に向かないと判断した. また, 多倍長数の負の数の表現方法として符号ビットによる方式を用いた.

多倍長数は n 個の 64 ビット符号なし整数型の配列により GPU 内のメモリに保持される. Fig. 1 の形で保持され, 配列要素 $x[0],x[1],\ldots,x[n-1]$ により多倍長数は構成される. $x[n-1]$ の MSB (最上位ビット) を除いた 63 ビットを w, $x[n-1]$ の MSB を s で表すことにする. このとき, 多倍長数は s により符号を定め, $x[0],\ldots,x[m-1]$ の m 個の配列要素により固定小数点の小数部を, $x[m],\ldots,x[n-2]$ の $n-m-1$ 個の要素と w により整数部分を表す. すなわち Fig. 1 で保持された多倍長数は

$$(-1)^s\times\left(w\times\beta^{n-m-1}+\sum_{j=0}^{n-m-2}x[m+j]\times\beta^j+\sum_{i=1}^{m}x[m-i]\times\frac{1}{\beta^i}\right)$$

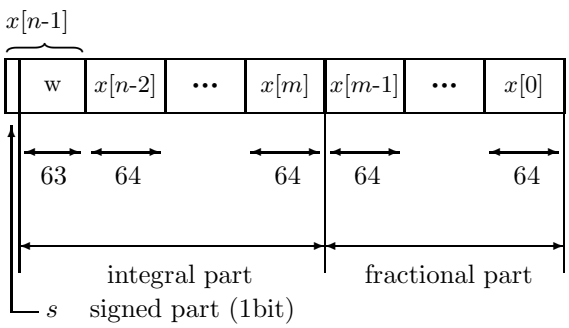

Fig. 1 data structure of a multiple precision number on the proposed GPU system

を表現する. ただし, $\beta=2^{64}$ である.

ユーザが 10 進法により整数部に要求する精度, 小数部に要求する精度をコンパイル時に決定することで n,m の値が決まり, 多倍長数のデータ構造が決定する仕組みをとっている.

上述の多倍長数型に対し, スカラー演算, ベクトル演算, 10 進法の入出力, 組込みの整数型および浮動小数点型の型変換の実装を行った. 実装したスカラー演算は多倍長数と多倍長数, 多倍長数と組込みの整数型および浮動小数点型との四則演算である. また, 本論文はベクトル演算に着目した研究であり, 実装したベクトル演算としては, ベクトル要素間の四則演算, 総和・標準内積演算である. ここで述べるベクトルは多倍長数の配列, または組み込み型の配列を指す.

ベクトル要素間の四則演算として

$$\begin{pmatrix}Z_0\\Z_1\\\vdots\\Z_{N-1}\end{pmatrix}=\begin{pmatrix}X_0\\X_1\\\vdots\\X_{N-1}\end{pmatrix}\triangle\begin{pmatrix}Y_0\\Y_1\\\vdots\\Y_{N-1}\end{pmatrix},$$

$$\begin{pmatrix}Z_0\\Z_1\\\vdots\\Z_{N-1}\end{pmatrix}=\begin{pmatrix}X_0\\X_1\\\vdots\\X_{N-1}\end{pmatrix}\triangle Y\ ,\quad\begin{pmatrix}Z_0\\Z_1\\\vdots\\Z_{N-1}\end{pmatrix}=X\triangle\begin{pmatrix}Y_0\\Y_1\\\vdots\\Y_{N-1}\end{pmatrix}$$

の実装を行った. ただし, $\triangle\in\{+,-,\times,/\}$, N はベクトルの要素数を, 大文字は多倍長数または組込みの整数型および浮動小数点型である. 乗除算の定義について注意が必要であり, 例えば

$$\begin{pmatrix}X_0\\X_1\\\vdots\\X_{N-1}\end{pmatrix}\times\begin{pmatrix}Y_0\\Y_1\\\vdots\\Y_{N-1}\end{pmatrix}:=\begin{pmatrix}X_0\times Y_0\\X_1\times Y_1\\\vdots\\X_{N-1}\times Y_{N-1}\end{pmatrix},$$

$$X\ /\begin{pmatrix}Y_0\\Y_1\\\vdots\\Y_{N-1}\end{pmatrix}:=\begin{pmatrix}X_0\ /\ Y_0\\X_1\ /\ Y_1\\\vdots\\X_{N-1}\ /\ Y_{N-1}\end{pmatrix}$$

と要素間の演算として定義している.

また, 総和演算

$$Z = \mathrm{SUMMATION}(\begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{pmatrix}) := \sum_{j=0}^{N-1} X_j \ ,$$

標準内積演算

$$Z = \mathrm{DOT\_PRODUCT}(\begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{pmatrix}, \begin{pmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{N-1} \end{pmatrix}) := \sum_{j=0}^{N-1} X_j \times Y_j$$

を実装した.

4. GPU を用いた本多倍長計算環境の特徴

本多倍長数値計算環境は, 100-10000 桁程度の精度に最適化したものである. ユーザはコンパイル時に精度を指定することで利用できるが, 計算の途中で多倍長環境の精度は変更できない.

GPU を利用した高速化において主メモリと GPU 内のメモリのやりとりはパフォーマンス低下を招く. 本研究では, 多倍長数のデータは演算中を除き, すべて GPU 内のメモリに確保することで, 主メモリとのやりとりを軽減した.

高速な多倍長演算の実装に NVIDIA の Compute Compute 2.0 以上 [8] の GPU デバイスで提供される命令を利用した. この命令は PTX コード [8] と呼ばれるアセンブラに相当するコードにより利用可能であり, 本研究は PTX コードを利用し, 演算の高速化を行った.

GPU は命令を実行するための CUDA コア [8] と呼ばれるコアが複数存在し, 高速化においてその取扱いが重要となる. ベクトル演算の各要素間の四則演算はそれぞれ 1 つの CUDA コアを用いて計算され, それぞれの各要素間の四則演算を各コアへ割り当てることで, 複数のコアを利用し, 高速化を図った. そのため, スカラー演算は 1 つの CUDA コアのみで実現され, GPU の持つコア数の点から無駄が多いことに注意が必要である.

上述の CUDA コアの割り当て方については, 演算の種類, 多倍長数の精度, ベクトルの要素数の 3 つから成るテーブル表をもち, このテーブル表参照することで適切な割当て方を実現している. これにより, 任意の要素数, 上述の精度の範囲においてパフォーマンスを引き出した.

ここで, 演算のパフォーマンスについて論じる. GPU を用いて構築した環境と　多倍長数値計算ライブラリ exflib[9, 10] との演算速度の比較を Fig. 2 に示す. Fig. 2 は

$$\begin{pmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{N-1} \end{pmatrix} = \begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{pmatrix} + \begin{pmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{N-1} \end{pmatrix} \tag{4}$$

を用いたベクトル演算の加法についてのパフォーマンスである. 尚, exflib の速度は参考文献の [11] による.

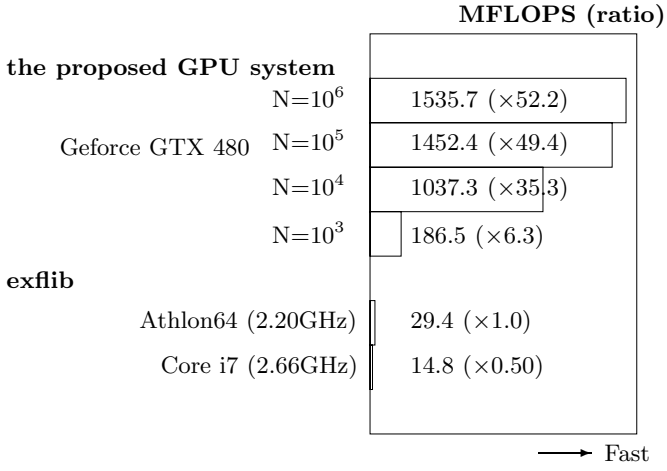

Fig. 2　comparison of speed of addition with 100 decimal precision

Fig. 2 に示す通り本多倍長数値計算環境ではベクトルの要素数 N に応じてパフォーマンスが異なる. これは上述のテーブル表を用いて各要素数ごとに最適な CUDA コアへの割当てを行い, GPU の利用方法がそれぞれ異なることによる. また, 要素数の増加に応じてパフォーマンスの向上が見られ, 要素数 $N = 10^6$ 以上でのパフォーマンスについては $N = 10^6$ のものと同程度である.

GPU を用いた本多倍長数値計算環境は C++言語を利用したインターフェースを提供している. 多倍長数, 多倍長数のベクトルのクラスを実装し, ユーザはこのクラスを利用することで, 3 節で述べた演算を実行することが可能である. 例えばベクトル演算の加法 (4) は

$$\mathrm{vectZ} = \mathrm{vectX} + \mathrm{vectY};$$

を記述し, コンパイルすることで, GPU の高速化手法を知ることなく, 上述のパフォーマンスのベクトル演算を利用できる. ただし, vectX, vectY, vectZ は上述の多倍長数のベクトル型である.

5. 複素逆 Laplace 変換への構築した環境の適用

複素逆 Laplace 変換を用いた M 個の点における原像 f の値を求める問題について, その高速化方法について論じる. すなわち $t_0, t_1, \ldots, t_{M-1}$ が与えられ, $\big(f(t_0), f(t_1), \ldots, f(t_{M-1})\big)^T$ の値を数値計算により求める問題について本節では考える.

複素逆 Laplace 変換は, 原像の値 $f(t_i)$ が別の原像の値 $f(t_{i-1})$ に依存することなく, 各 t_i ごとに Bromwich 積分から導出されるスキームを並列に計算できる. このことを細野の方法を例にとり詳述する.

本問題に対して細野の方法 (3) は

$$\begin{pmatrix} f(t_0) \\ f(t_1) \\ \vdots \\ f(t_{M-1}) \end{pmatrix} = \frac{e^\sigma}{t} \left(\sum_{n=1}^{k-1} \begin{pmatrix} F_n(t_0) \\ F_n(t_1) \\ \vdots \\ F_n(t_{M-1}) \end{pmatrix} + \right.$$

Table 1 computational time using Hosono's method ($F(s) = \exp(-s), \sigma_0 = 40, N = 80, \mu = 40$)

	numerical system	
M	exflib	the proposed GPU system
10^3	4.1 (sec)	5.0 (sec)
10^4	40.8 (sec)	7.5 (sec)
10^5	407.3 (sec)	39.4 (sec)

$$\frac{1}{2^{\mu+1}} \sum_{\nu=0}^{\mu} A_{\mu,\nu} \times \begin{pmatrix} F_{k+\nu}(t_0) \\ F_{k+\nu}(t_1) \\ \vdots \\ F_{k+\nu}(t_{M-1}) \end{pmatrix} \Bigg)$$

と表せる. これにより, ベクトル $\big(F_n(t_0), F_n(t_1), \ldots, F_n(t_{M-1})\big)^T$ $(n = 1, 2, \ldots, k+\mu)$ を得ることで, 3 節で述べたベクトル演算を用いて細野の方法は実現できる.

また, このベクトル $\big(F_n(t_0), F_n(t_1), \ldots, F_n(t_{M-1})\big)^T$ を計算する際にも 3 節のベクトル演算を利用でき, 例えば, $F(s) = \exp(-s)$ では,

$$\begin{aligned} F_n(t_j) &= (-1)^n \operatorname{Im} F\big(z_n(t_j)\big) \\ &= (-1)^{n+1} \exp\left(-\frac{\sigma_0}{t_j}\right) \sin\left(\left(n - \frac{1}{2}\right)\frac{\pi}{t_j}\right) \end{aligned}$$

と表される. この式に対し, 指数函数, 三角函数を Taylor の定理を利用して多項式で近似することで, $\big(F_n(t_0), F_n(t_1), \ldots, F_n(t_{M-1})\big)^T$ を実装したベクトル演算の四則演算を用い, 並列に求めることが可能である.

ここでは, $F(s)$ として上述の $\exp(-s)$ を, $\{t_j\}_{j=0}^{M-1}$ は $[0, 2]$ の等分点を採用した. また, 細野の方法のパラメータとして $\sigma_0 = 40, N = 80, \mu = 40$ を用いた.

数値計算結果を Table 1 に示す. 10 進 100 桁の浮動小数点環境と数値計算結果に対して同じ計算精度を得るのに 10 進 145 桁の固定小数点環境 (整数部 10 進 35 桁, 小数部 10 進 110 桁) が必要であり, この計算精度で比較を行った. ただし, exflib の結果は CPU のうち 1 コアのみを利用したものである. また, CPU は Opteron 280 を GPU は Geforce GTX 580 を利用した.

まず, $M = 10^3$ のときは, GPU を用いた環境での数値計算は exflib を用いた数値計算に比べ, およそ 1.2 倍の計算時間を要した. これは高並列性をもつ GPU にとって問題が小規模であり, GPU の演算コアの大部分が使われなかったためと考えられる. また, $M = 10^5$ のときの exflib を用いた数値計算と GPU を用いた環境を用いた数値計算ではおよそ 10 倍程度の計算速度の向上が実現された.

さらに, GPU を利用した環境では M の増加と共にパフォーマンスが向上する. Table 2 は計算時間/M を表した表であり, 原像の値を 1 つ求めるのに要した平均時間を表す. exflib を用いた計算環境では 4.1×10^{-3} と一定である. これに対し, GPU を利用した本環境は M の増加と共に平均時間は減少している.

Table 2 mean computational time for each f(t) using Hosono's method ($F(s) = \exp(-s), \sigma_0 = 40, N = 80, \mu = 40$)

	numerical system	
M	exflib	the proposed GPU system
10^3	4.1×10^{-3}(sec)	5.0×10^{-3}(sec)
10^4	4.1×10^{-3}(sec)	7.5×10^{-4}(sec)
10^5	4.1×10^{-3}(sec)	3.9×10^{-4}(sec)

本論文は, 複素逆 Laplace 変換の高速化を $F(s) = \exp(-s)$ に関する細野の方法を例に示した. 特に問題の規模が大きい場合には, GPU を用いた本環境により高いパフォーマンスが得られることを論じた.

参考文献

(1) 荒井政大, 林久志, 三宅達也, 長秀雄, 内山友成：レーザー超音波を用いた薄膜の密着強度評価に関する境界要素解析, 計算数理工学論文集, **9**(2009), pp. 25–30.

(2) Cohen, A. M. : Numerical methods for Laplace transform inversion, (2007), Springer.

(3) Davies, B. and Martin, B. ：Numerical inversion of the Laplace transform: a survey and comparison of methods, *J. Comput. Phys.*, **33**(1979), pp. 1–32.

(4) Duffy, D. G.：On the numerical inversion of Laplace transforms: comparison of three new methods on characteristic problems from applications, *ACM Trans. Math. Software*, **19**(1993) pp. 333–359.

(5) Fujiwara, H.：Numerical real inversion of the Laplace transform by reproducing kernel and multiple-precision arithmetic, *Proceedings of the 7th International ISAAC Congress*, (2010), pp. 289–295.

(6) 藤原宏志：複素逆 Laplace 変換の数値計算における注意, 計算数理工学論文集, **10**(2010), pp. 7–10.

(7) 細野敏夫：数値ラプラス変換, 電気学会論文誌 A, **99**(1979), pp. 494–500.

(8) NVIDIA Corporation, NVIDIA CUDA C Programming Guide version 4.0, (2011).

(9) 藤原宏志：高速な多倍長計算環境の PC・WS 上での実現, 計算数理工学レビュー No.2005–1(2005), pp. 33–40.

(10) exflib home page： `http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/index.html`

(11) 藤原宏志：科学技術のための高速多倍長数値計算環境の構築, 計測と制御, **49**(2010), pp. 285–290.