Vol.2014-ARC-208 No.19
2014/1/24

GPGPU による多数桁計算を用いた円周率を求める級数計算の並列処理

木原弘貴[1]　　山内長承[2]

級数を使う近似計算を多数桁精度で行う場合、級数において加算する項数が増加すると計算量も増加し、また多数桁での乗除算は計算量が多くなる。そこで、本研究では多数の演算処理ユニットを持つ GPGPU を用いて、多数桁での円周率を求める級数計算を並列に処理する手法を開発し、級数の高速計算を目指す。本研究では、円周率を求める BBP 公式を選択し、並列に分割した結果十分な並列性を確保し高速な計算が実現できるかを検討する。GPGPU では高速アクセス可能なメモリの容量が小さく、容量の大きいメモリへのデータ転送速度が限定されるため、多数桁計算でのメモリ使用量が小さい、途中結果のメモリ間移動が限定され少ないというメリットを持つ BBP 公式を用いて円周率計算を行う。予備実験にて、BBP 公式の処理が、多数桁計算の並列化によって約 3.0 倍、公式の 4 項をそれぞれ並列に処理することで約 3.8 倍の速度向上を確認した。

1. はじめに

円周率を求める際に、無限級数計算が使用される。また、無限級数を使う近似計算を多数桁精度で行う場合、級数において加算する項数が増加すると計算量も増加し、多数桁での乗除算の処理についても計算量が多くなる。そこで、本研究では多数の演算処理ユニットを持つ GPGPU を用いて、多数桁精度での円周率を求める級数計算を並列に処理する手法を開発し、級数の高速計算を目指す。

従来、円周率計算はコンピュータ・クラスタなどでの並列処理を念頭に置いた手法が考えられてきた。Ramanujan[1][2]や Chudnovsky 公式[1][3]をもとに、Binary Splitting 法[3]や DRM[1][4]などの計算手法が用いられている。これらの手法は通分処理を行なうことで、除算回数を減らし計算速度の向上を図っている。しかし、円周率の桁を多く算出する場合、通分処理によって分母、分子にとても大きな桁数の整数を保持しなければならない。この様な、膨大なデータの保持と使用は、高速アクセス可能なメモリの量が小さく、容量の大きいメモリへのデータ転送速度が限定されているという性質を持つ GPGPU には適さない。そこで、GPGPU での処理では、メモリアクセス速度とメモリ容量の性質を念頭に置いた計算アルゴリズムの選択が求められる。級数による近似計算では、従来、収束の高速性のために用いられてきた Ramanujan などの公式の他、計算結果の任意桁の数値が得られる BBP 公式[1][5]が考えられてきた。前者に比べて後者は、多数桁精度計算でのメモリ使用量が小さい、途中結果のメモリ間移動が限定され少ないという点で GPGPU に適している。本研究では、円周率計算の後者の方式である BBP 公式を選択し、並列に分割した結果十分な並列性を確保し高速な計算が実現できるかを検討する。

1 東邦大学大学院理学研究科情報科学専攻
2 東邦大学理学部情報科学科

2. GPGPU を用いた BBP 公式の並列処理手法と多数桁精度計算

本章では、円周率計算として BBP 公式を選択した理由、BBP 公式の並列処理手法、使用している多数桁精度計算のアルゴリズムについて記述する。

2-1. BBP 公式の選択理由

GPGPU の性質から、Ramanujan や Chudnovsky などの公式と Binary Splitting 法や DRM の計算手法は、GPGPU での処理には適していない。そこで、GPGPU で処理しやすい性質を持つ BBP 公式を、円周率を求める公式として選択した。

円周率計算には、Chudnovsky 公式などの収束の速い式が使われている。2011 年 11 月に 10 兆桁を算出した Yee、近藤らも円周率を求めるのに Chudnovsky 公式を用いている[4]。また、級数の各項で除算を行うことによる計算速度の低下を防ぐために、Binary Splitting 法や DRM などの通分処理を行い、除算回数を減らすアルゴリズムが使われている。しかし、円周率の桁を多く算出する場合、通分処理によって分母、分子にとても大きな桁数の整数を保持しなければならない。この様な、膨大なデータの保持と使用は GPGPU には適さない。

GPU では、高速アクセス可能なメモリの量が小さく、容量の大きいメモリへのデータ転送速度が限定されているという性質がある。図 1 は、CUDA のメモリモデルである。CUDA では、チップ上にあるレジスタ、シェアードメモリへのアクセス速度は速く、チップ外にあるグローバルメモリへのアクセス速度は遅い。また、NVIDIA GeForce GTX TAITAN において、レジスタは 1Block あたり 256KB、シェアードメモリは 1Block あたり 48KB とメモリ容量が小さく、グローバルメモリは、6144 MByte と容量が大きい。他の GPU についても、アクセス速度の速いメモリは容量が小さく、アクセス速度の遅いメモリは容量が大きい。

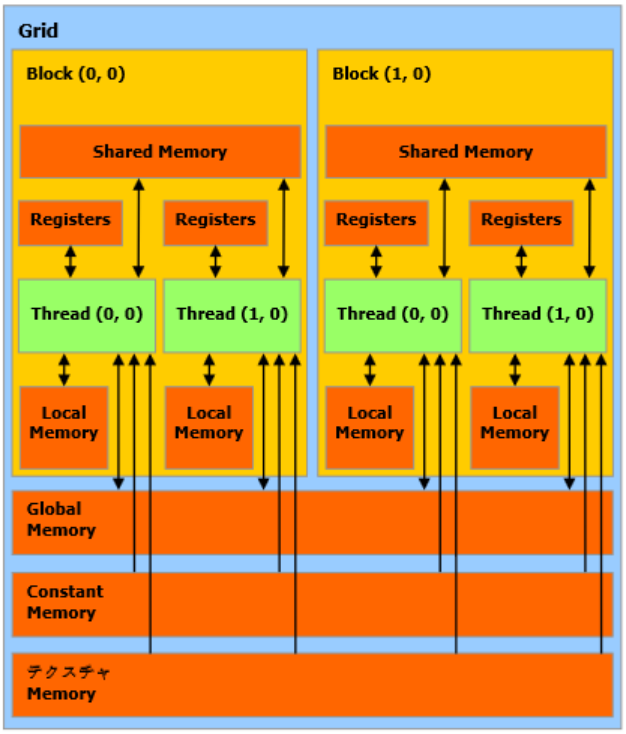

図 1. CUDA のメモリモデル[6]

円周率計算において、桁を多く算出する場合、多数桁精度計算を行う必要がある。そのため、扱う数値の桁が大きい場合、高速アクセス可能なメモリにデータが入りきらない。入りきらないデータは、容量の大きいメモリで保持することになり、計算に必要になったときに、適宜、高速アクセス可能なメモリへデータを移すこととなる。よって、扱う数値の桁が大きくなればなるほど、メモリ間でのデータ移動が増えることになる。

そこで、円周率の任意桁以降の数値が得られるという特徴を持つ BBP 公式を使用する。公式は、式 1 のように表される。

$$\pi = \sum_{n=0}^{\infty}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right) \quad (式\ 1)$$

式 1 により、16 進数での円周率を求めることが出来る。16 進数での円周率 d 桁目（d の値は任意）を求めるとき、式 1 に16^dを掛け式 2、式 3 の様にすることで計算結果の小数第 1 位に d+1 桁目の値が出力される。

$$16^d\pi = \{4(16^dS_1) - 2(16^dS_4) - (16^dS_5) - (16^dS_6)\} \quad (式\ 2)$$

$$16^dS_j = \left(\sum_{k=0}^{d}\frac{16^{d-k}}{8k+j}\right) + \left(\sum_{k=d+1}^{\infty}\frac{16^{d-k}}{8k+j}\right) \quad (式\ 3)$$

d+1 桁目以降の値を求めるので、整数部分は必要ない。そこで、式 3 を式 4 にする。これにより、計算の途中結果の桁数を減らすことができるため、保持する必要のあるデータ量を減らすことができる。

$$16^dS_j = \left(\sum_{k=0}^{d}\frac{16^{d-k}\ mod\ 8k+j}{8k+j}\right) + \left(\sum_{k=d+1}^{\infty}\frac{16^{d-k}}{8k+j}\right) \quad (式\ 4)$$

式 4 は、d+1 桁目以降の数桁分の精度の値を出力する。そのため、円周率全体は、d の値を数桁分ずつ加算し（例えば、計算結果を 7 桁精度で出力する場合は、d に 7 ずつ加算していく）、それぞれ式 4 に代入して解き、計算結果を繋ぎ合わせることで求められる。この計算方法は、GPGPU の多数並列のメリットを生かすことが出来る。

2-2. BBP 公式の並列処理手法

並列 Thread は、図 2 の様に CUDA により階層的にグループ化されている[6]。ここで、BBP 公式の並列計算にあたり、BBP 公式を Block 単位で分割し、多数桁精度計算を Thread 単位で処理する。BBP 公式の d の値の入力と出力結果のイメージは、表 1 の様になる。7 桁の精度で出力するとすれば、1 桁目から 7 桁目、8 桁目から 14 桁目、15 桁目から 21 桁目について、それぞれ BBP 公式を解くこととなる。

d	0	7	14	・・・
桁	1～7	8～14	15～21	・・・
値	0x243f6a8	0x885a308	0xd313198	・・・

表 1. BBP 公式の d の値の入力と出力結果のイメージ

まず、式 2 の16^dS_1、16^dS_4、16^dS_5、16^dS_6をそれぞれ並列に計算する。さらに、式 3 の$\sum_{k=0}^{d}\frac{16^{d-k}}{8k+j}$についても、d の値に応じて分割し並列処理する。また、式 3 の$\sum_{k=d+1}^{\infty}\frac{16^{d-k}}{8k+j}$については、現在、項の計算

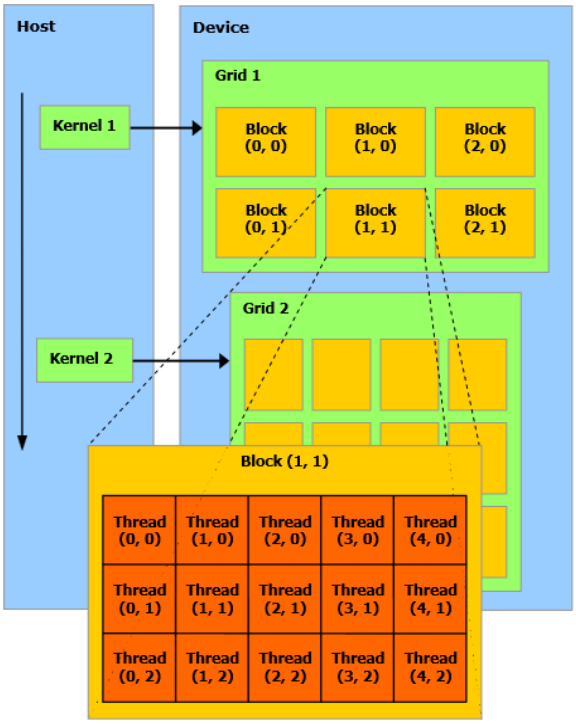

図 2. CUDA での並列 Thread の階層[6]

が一定精度以下になったときに計算を打ち切っているため、並列処理は行っていない。例えば、式 3 の $\sum_{k=0}^{d}\frac{16^{d-k}}{8k+j}$について、4 分割し並列処理するとすれば、式 2 の$16^{d}S_1$を Block 番号 (0, 0) (1, 0) (2, 0) (3, 0) が処理する。他も同様に、$16^{d}S_4$を Block (0, 1) (1, 1) (2, 1) (3, 1)、$16^{d}S_5$を Block (0, 2) (1, 2) (2, 2) (3, 2)、$16^{d}S_6$を Block (0, 3) (1, 3) (2, 3) (3, 3) に処理させる。合計 16Block 使用し、式 2、式 3 をそれぞれ 4 並列ずつ処理する。各 Block 内での多数桁精度計算は Thread で並列処理する。なお、GPU の Block は多数用意されている。例えば、GeForce GTX TITAN には、1024 x 1024 x 64 の Block が存在する。そのため、同時に複数の BBP 公式の計算が可能である。

2-3. 現在使用している 多数桁精度計算アルゴリズム

現在、多数桁精度計算は固定小数点法にて実装しており、Thread による並列処理を行っている。乗算については、一般的な総当りで計算している。除算については、商または余りを出力する除算と小数を出力する除算の 2 つの関数を使用しているが、ともに引き放し法[7]を用いている。後述の予備実験の章でも述べるが、除算の計算速度は遅く、アルゴリズムも並列化しにくいので、Newton-Raphson 法を用いた除算に改良を行う予定である。また、その改良に伴い、多数桁精度計算を固定小数点法から浮動小数点法に変更する予定である。乗算についても、より高速な FFT を用いたアルゴリズムへの改良を行う予定である。また、予備実験にて比較している GPU 上での逐次処理にて使用しているアルゴリズムは、乗算がブース法であり、除算については、ともに引き放し法を用いている。

3. 予備実験

3-1. 予備実験の目的と対象

提案手法による処理速度の向上を確認することを目的とする。乗算と商または余りを出力する除算、小数出力用除算、多数桁精度計算を含めた BBP 公式全体の処理を対象とし、GPU 上での逐次処理と並列処理での処理速度を比較する。多数桁精度計算の並列処理での使用 Thread 数は、32Thread である。また、現時点で、円周率全体の計算については実装していないため、1 つの BBP 公式の処理を対象とする。多数桁精度計算は、それぞれ2^{31}進数 32 桁での計算となる。それぞれの処理時間の取得データ数は、20 個である。実験環境を表 2 に示す。

GPU	NVIDIA GeForce GTX TITAN 837MHz 2688 CUDA Cores
Video RAM	GDDR5 384bit 6Ghz 6GB
OS	Windows7
Language	C++, CUDA

表 2. 実験環境

3-2. 実験結果

多数桁精度乗算について、逐次処理と並列処理の平均処理時間を図 3 に示す。逐次処理では、ブース法[7]を用いている。並列処理では、一般的な総当たりでの方法を採っている。平均処理時間は、逐次処理が 19.52ms で並列処理が 1.53ms で、逐次処理に比べ 12.72 倍の速度向上となった。計算量は N 桁精度の場合、逐次処理では$O(31N)$であり、並列処理では$O(N)$となる。そのため、12.72 倍の速度向上は予想より低い。現在、乗算は、円周率の値の出力結果を、16 進数から 10 進数へ変換する際に多く用いられている。BBP 公式内での乗算は、全て 2 の累乗の値との乗算なので、左シフト演算を用いている。そのため、BBP 公式の処理では、速度向上による影響がない。しかし、2 章 2-3 でも述べたように、多数桁精度の小数出力用除算を Newton-Raphson 法を用いた除算に改良を行う予定である。その際、乗算が頻繁に使われるため、多数桁精度乗算の速度が大きく影響すると考えている。

多数桁精度の商または余りを出力する除算について、逐次処理と並列処理の平均処理時間を図 4 に示す。逐次、並列処理ともに、引き放し法を用いている。平均処理時間は、逐次処理が 34.00ms で並列処理が 12.69ms で、逐次処理に比べ 2.68 倍の速度向上となった。現在、商または余りを出力する除算については、式 4 の$16^{d-k} \ mod \ 8k+j$を繰り返し計算するため頻繁に用いられている。そのため、BBP 公式の処理では、速度向上による影響が大きい。

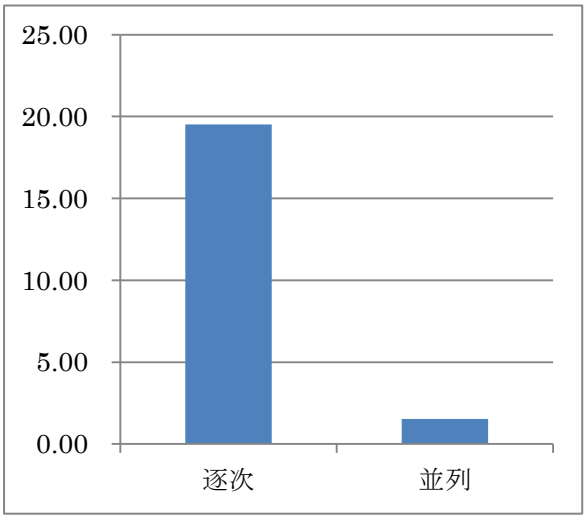

図 3. 多数桁精度乗算の逐次処理と並列処理の平均処理時間

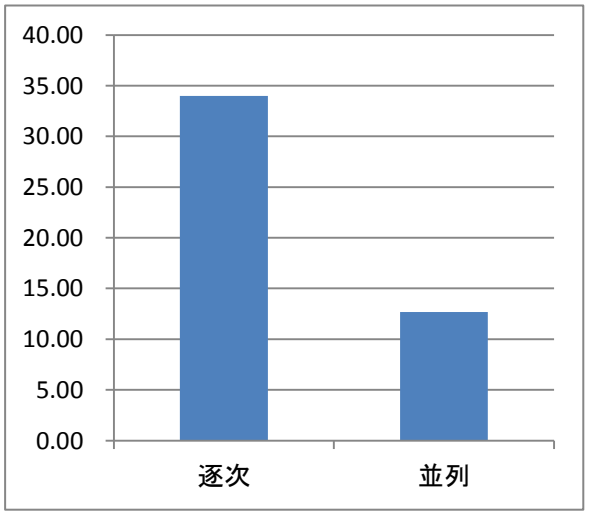

図 4. 多数桁精度の商または余りを出力する除算の逐次処理と並列処理の平均処理時間

多数桁精度の小数出力用除算について、逐次処理と並列処理の平均処理時間を図 5 に示す。逐次、並列処理ともに、引き放し法を用いている。平均処理時間は、逐次処理が 1061.04ms で並列処理が 1003.89ms で、大きな変化はなかった。現在、小数出力用除算については、式 4 の$\sum_{k=0}^{d}\frac{16^{d-k}\ mod\ 8k+j}{8k+j}$の各項にて用いられている。BBP 公式の処理では、d の値が大きくなると、用いられる回数が増加する。式 4 の$\sum_{k=d+1}^{\infty}\frac{16^{d-k}}{8k+j}$での使用回数については、計算を打ち切る精度によるが、d=0 かつ 16 進数での円周率の値 7 桁分の精度で出力をする場合、小数出力用除算は 9 回使用される。d の値が大きくなると、使用回数は減ってゆく。

多数桁精度計算を含めた BBP 公式全体の処理について、逐次処理と Block 数 1Block、2Block、4Block での並列処理の平均処理時間を図 6 に示す。逐次処理では、多数桁精度計算、BBP 公式の処理ともに逐次処理を行っており、Block 数 1Block の並列処理では、多数桁精度計算が 32Thread での並列処理であり、BBP 公式の処理は逐次処理となる。平均処理時間は、逐次処理で 151728.51ms で、1Block のとき 50412.25ms、2Block のとき 25263.03ms、4Block のとき 13267.32ms となった。逐次処理と比較して、それぞれ、3.01 倍、6.01 倍、11.44 倍の速度向上となった。また、1Block に対して 2Block、4Block での速度向上率は、それぞれ、1.99 倍、3.80 倍となった。多数桁精度計算での並列処理は、乗除算だけでなく、加減算や他のメモリへの値の代入なども並列化しているため、逐次処理に比べ、多数桁精度計算での 32Thread、BBP 公式 1Block での並列処理の速度向上が 3.01 倍の速度向上は妥当と言える。また、式 2 の16^dS_1、16^dS_4、16^dS_5、16^dS_6をそれぞれ Block 単位で並列に計算しているので、1Block での並列処理に対し、2Block、4Block での並列処理の速度向上が、1.99 倍、3.80 倍の速度向上は妥当と言える。

多数桁精度計算については、改良する必要が大いにあるが、BBP 公式全体の並列化については、良い結果が得られた。今後、更なる並列化により処理速度の向上が見込める。

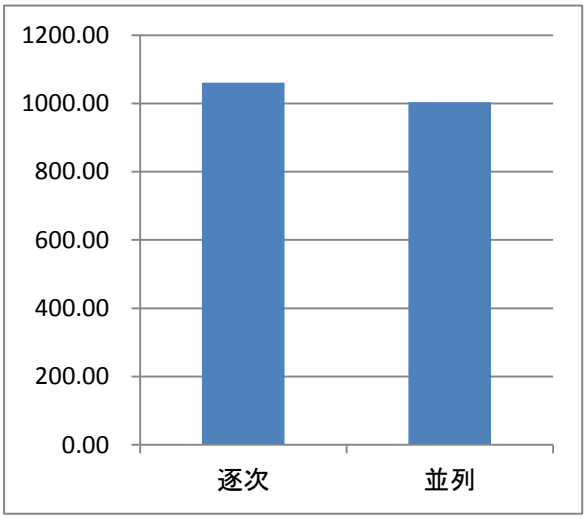

図 5. 多数桁精度の小数出力用除算の逐次処理と並列処理の平均処理時間

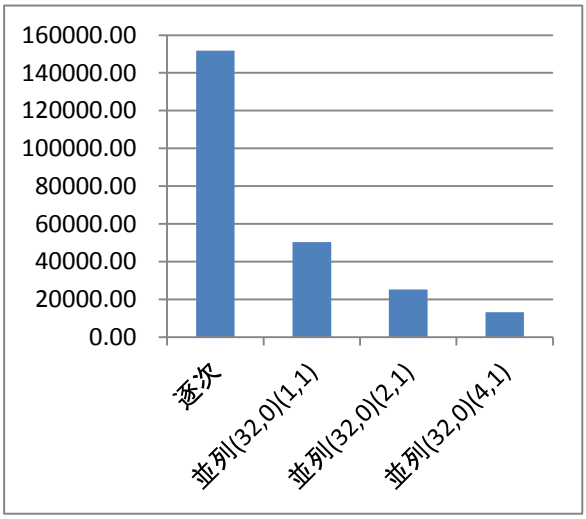

図 6. 多数桁精度計算を含めた BBP 公式全体の処理の逐次処理と並列処理の平均処理時間

3-3. 考察

実験の結果から、多数桁精度乗算において、並列処理による処理速度の向上を確認した。しかし、予想より速度向上は見込めなかった。複数の Thread が、同じメモリへアクセスするバンクコンフリクトが原因と考えられる。多数桁精度での商または余りを出力する除算において、並列処理による処理速度の向上を確認した。しかし、同じアルゴリズムを使用している多数桁精度での小数出力用除算にて、速度向上が見込めないことから、テストデータが適切でない可能性がある。今後、複数のパターンでテストしなおす必要がある。多数桁精度での小数出力用除算については、速度向上は確認できず、処理時間も非常に長い。原因として、引き放し法が並列化しにくいアルゴリズムのため、速度向上が見られなかった。そのため、2 章 2-3 でも述べたが、Newton-Raphson 法を用いた除算に改良を行う予定である。また、BBP 公式全体の処理について、BBP 公式の式 2 の 4 項の Block での分割による処理速度の向上を確認した。これにより、BBP 公式が並列処理しやすいことがわかる。

4. まとめ

本稿では、GPGPU を用いて、多数桁精度での円周率を求める級数計算を、並列に処理する手法を示した。現時点で、多数桁精度計算の実装と 1 つの BBP 公式の計算の実装が完了している。円周率については、任意桁から 7 桁分の精度の値の計算ができている。

BBP 公式の性質から、GPGPU を用いた並列処理に最適との仮説を立てた。そして、実験により、BBP 公式が並列処理しやすいことが確認された。今後、多数桁精度計算の高速化などの改良と、複数の BBP 公式を同時に計算し、円周率の最初の桁から任意桁までの値の出力ができるようにしたいと考えている。

参考文献

[1] 円周率.jp (online), http://xn--w6q13e505b.jp/

[2] J.M.Borwein, P.B.Borwein, D.H.Bailey, ″Ramanujan, Modular Equations, and Approximations to Pi or How to Compute One Billion″, The American Mathematical Monthly, Vol.96, No.3, (Mar.,1989), pp.201-219

[3] Alexander J.Yee, Shigeru Kondo (online), ″10 Trillion Digits of Pi: A Case Study of summing Hypergeometric Series to high precision on Multicore Systems″, IDEALS https://www.ideals.illinois.edu/handle/2142/28348

[4] 後保範, 金田康正, 高橋大介, ″無限級数に基づく多数桁計算の演算量削減を実現する分割有理数化法″, 数理解析研究所講究録 1084 巻 1999 年 pp.60-71 http://www.kurims.kyoto-u.ac.jp/~kyodo/kokyuroku/contents/1084.html

[5] David Bailey, Peter Borwein, Simon Plouffe, “On the rapid computation of various polylogarithmic constants”, MATHEMATICS OF COMPUTATION, Vol.66, Number.218, April 1997, pp.903-913

[6] NVIDIA Corporation (online), ″NVIDIA CUDA Programming Guide ver1.1″, http://www.nvidia.co.jp/docs/IO/51174/NVIDIA_CUDA_Programming_Guide_1.1_JPN.pdf

[7] 堀桂太郎, ″図解コンピュータアーキテクチャ入門″, 森北出版株式会社 pp.43-55