Date: Mon, 22 Nov 2010 18:12:45 +0900
Subject: hp2tg_t=multi-thread-disk-I/O+CBP-engine-for-CUDA
To: Kentaro UESUGI
Cc: Masayuki Uesugi , Satoshi Okumura
Attached: ~~hp2tg_sec.e~~, ~~cbp.eu.none.~~, hp2tg_time.txt, cbps.pdf, hp2tg.pdf

---

うえすぎさま、

GSJ/AIST のなかのです。10/23 付のお知らせの E-mail から間があいてしまいました。CUDA GPU 用のプログラム hp2tg_t の話をします。これは sinogram と tomogram のデータのディスク I/O を CPU のマルチスレッド処理で、それらの変換（画像再構成）を CBP engine for CUDA（ver.8）の並列処理で行います。

(0) 復習

以前からある通常 CPU 用の hp2tg_t（これはデータのディスク I/O ではなく、画像再構成をマルチスレッドで処理する）や、1 枚のスライス画像だけを再構成するプログラム sg2tg などでは以下の手順で処理を行っています：

[0] 関数 InitCBP() を呼んで CBP 法による画像再構成の準備をする。
--- ここから個々のスライス画像の再構成処理
[1] 測定画像（dark.img や q*.img）を読み込み、指定したスライスの sinogam を作成する。なお、そのスライス番号を z として、以下ではこの処理を仮想的な関数 read_SG(z) で行っていることにします。
[2] 関数 CBP(z) を呼び出してそのスライスの画像を再構成する。
[3] そのスライス画像を TIFF ファイルに書き込む。ただし、先と同様に、この処理も仮想的な関数 store_TG(z) で行っていることにします。
--- ここまで
[4] 関数 TermCBP() で処理の後始末（使用したメモリの解放）を行う。

N を測定画像の横画素数 (== 再構成画像の横・縦画素数)、M を投影数として、関数 CBP() を使った 1 枚のスライスの画像再構成の処理時間は概ね N×N×M に比例します。NVIDIA Tesla C1060 GPU を使った場合、その比例定数（時定数；τ と呼びます ）は 0.1 nano sec. 程度の値になりました。つまり、1800 投影、2000×2000 画素の画像なら 1 秒以下（計算上は 0.72 秒）で再構成できます。

このように CUDA GPU を使うと CBP() の処理が高速化されたので、read_SG() や store_TG() によるディスク I/O の処理時間が問題になってきました。また、CBP() の最後の部分で実行している逆投影（Back Projection）演算では CUDA GPU の上で走っている関数 BP*() の処理（これが CBP() の処理時間の大部分を占めます）の終了を CPU は何もせずに待っているだけです。そこで、複数スライスを再構成する場合の全体としての処理時間の短縮を期して、この待ち時間に GPU と並列に CPU が read_SG() や store_TG() を実行するプログラム hp2tg を書きました。そのためにまず、CBP engine for CUDA の関数 CBP()のコードを以下のように２個に分割しました（ただし、sg2tg などの従来の画像再構成用メインプログラムのコードをそのまま使えるように、CBP engine for CUDA には関数 CBP() = BeginCBP() + EndCBP() も残してあります ）:

関数 BeginCBP(z)
　CBP(z) の最初から BP*(z) の呼び出しまでの処理を行う。
関数 EndCBP(z)
　GPU による BP*(z) の処理の終了を待ち、再構成画像を取得する。

そして、これらを用いた hp2tg では以下のような手続きによって、BeginCBP() と EndCBP() の間の待ち時間に（CPU が）それ以前に得た tomogram や次の画像再構成に使う sinogram に対する処理を行うようにしました：

```
for   z = z1 ～ z2 + 1 ← 再構成するスライスは z1～z2 だが、...
    if z<=z2 then read_SG(z)          ← 次の sinogram を読み込む
    if z>z1 then EndCBP(z-1)          ← 前の tomogram を GPU から取得
    if z<=z2 then BeginCBP(z)         ← 次の sinogram を GPU に渡す
    if z>z1 then store_TG(z-1)        ← 前の tomogram を書き込む
```

２年前にこのような hp2tg のコードを書き、当時使っていた CUDA toolkit 2.0（CUDA 2.0）の環境で色々と動作テストをしました。その結果、hp2tg を使った複数スライスの画像再構成でも、全体として前記の１枚のスライスだけを再構成した場合と同程度の時定数（τ =0.1 nano sec. 程度）で処理を行えていることを確認できました。

注
　ぼくは、read_SG()、CBP()、store_TG() を順に呼び出している従来の画像再構成プログラムによる処理を「sequential processing」、CUDA GPU 用の hp2tg による処理（CPU と GPU の並列処理）を「parallel processing」と呼んでいます。

その後ちょくちょく、今年の夏には本格的にCBP engine for CUDAを改良しました。特に、最新版のCBP engine for CUDAは~~最新の安定版である~~CUDA 3.1の環境に適合したコードになっています。そして、これらの改良により、１枚のスライスだけを再構成した場合の時定数は0.08 nano sec. 以下になりました。

ところが、先月になって、最新版のCBP engine for CUDAを組み込んだhp2tgをCUDA 2.1や3.1の環境で走らせると以前にCUDA 2.0で走らせていた場合よりも長い処理時間を要すること（があること）に気づきました。CUDA 3.1 がインストールされているSPring-8の計算機sp8ct（+ NVIDIA C1060 GPU）では以前と同じか若干遅く、CUDA 2.1を使っていたぼくの計算機gsjkic（+ C1060）では以前の倍以上の処理時間になることもありました。その後、gsjkicにCUDA 3.1をインストールしましたが、状況は変わりません。要約すると以下のことがぼくを悩ませています：

[0] CBP engine for CUDA は高速化したのにhp2tgが遅いのはなぜか？
[1] それは CUDA toolkit の問題なのか？
[2] それを改善する方法は？

これらのうち [0] と[1] の理由は未だ明確にわかっていません。[2] について考えた結果がプログラムhp2tg_tです。

(1) hp2tgの実行状況の調査

hp2tg_tのコードを書く前に２種類の方法で hp2tg の実行状況を調査しました。

まずCUDA toolkit（3.1 を使いました）が何らかの「悪さ」をしていないかを調べるため、Linuxコマンドstrace を使ってhp2tgが呼び出しているsystem call を追跡してみました。hp2tgにN=2000、M=3600、Z=1312（スライス数）の測定080411aのデータを食わせてstraceを実行したので、主にディスクI/Oに関する膨大な回数のsystem callが検知されました。ところで、１枚目のスライス（z=0）付近でhp2tgの画像再構成の手続きは以下のようになっています。

```
read_SG(0)
BeginCBP(0)
read_SG(1)
EndCBP(0)
...
```

strace で調べると read_SG(0) と read_SG(1) のそれぞれから (M=3600 に対応した) 3600 回づつ呼び出されたファイル読み込み用の system call "read" に挟まれて、system call "sched_yield" が複数回呼び出されていました。これは BeginCBP(0)から呼び出されたのだと思われます。"sched_yield" はプロセスやスレッドの実行の優先順位を下げる system call で、BeginCBP() の最後に GPU 用の関数 BP*()を起動した後、その処理待ちに備えて CUDA 3.1 が呼び出したようです。そのため、この後に hp2tg のメインスレッドとして CPU が実行した read_SG() などの処理が遅くなった可能性があります。この真偽は不明ですが、もしそうでも、read_SG() などを hp2tg のメインスレッドとは別のスレッドで実行すればそのような処理速度の低下は生じないはずです。

次に、hp2tg で画像再構成の処理をしない場合の read_SG() と store_TG() の正味の実行時間（経過時間）を調べました。これには、~~この E-mail に添付した~~（hp2tg のコード "hp2tg.c" を改造した）"hp2tg_sec.c" と（CBP engine for CUDA の "cbp.cu" に対応する）"cbp.cu.none" を使いました。"hp2tg_sec.c" は以前に報告したテストでも使いました。そして、"cbp.cu.none" のコードでは以下の関数が通常の CBP engine for CUDA のものと異なります：

BeginCBP()
　何もせずにリターンする。
EndCBP()
　画像中央の画素のみ CT 値１、それ以外は CT 値０の再構成画像を返す。

添付したテキストファイル"hp2tg_time.txt "にはこのようなコードを使ったプログラムを計算機 gsjkic と sp8ct のそれぞれで実行して、測定 080411a のデータを処理した時の以下の情報が記されています：

[0] Linux の time コマンドで測ったプログラム全体の経過時間など
[1] read_SG()、BeginCBP()、EndCBP()、store_TG()のそれぞれの経過時間

また、このファイルには以前の版の CBP engine for CUDA（"cbp.cu.7"；最新版との実質的な違いはない）を組み込んだ hp2tg で普通に画像再構成した時の情報も記されています。ここでは以下の項目の「表」に注目して下さい：

function
　関数名。ただし、read_SG()と store_TG()は read_raw と store_tg になっています（"hp2tg_sec.c"を修正するのを忘れていました）。

total sec.

測定 081011a の 1312 枚のスライスすべてに対する処理で費やされた、その関数の正味の実行時間の合計（単位は秒）

min. & max. sec.

それぞれのスライスの処理に要した正味の実行時間の最小値と最大値

mean & SD sec.

スライスごとの正味の実行時間の平均値とその標準偏差

"hp2tg_time.txt"の１ページ目に記されているように、こちらの計算機 gsjkic では画像再構成の処理をした・しないに関わらずスライスごとの read_SG()の正味の実行時間が大きくばらついています（0.61～35.5 秒）。時々生じている read_SG()の長時間処理のため hp2tg 全体の処理速度が遅くなっていることが明らかです。gsjkic では CUDA GPU と関係のない異常が生じているようです。この gsjkic の問題について考えるのは後回しにします。

"hp2tg_time.txt"の２ページ目より、計算機 sp8ct では再構成した・しないに関わらずスライスごとの read_SG()の正味の処理時間は一定値（平均 0.66 秒程度）です。また、再構成した時の BeginCBP()と EndCBP()の処理時間（それぞれ平均 0.52 と 0.04 秒程度）も同様です。そして、再構成した・しないで store_TG()の処理時間（0.30 と 0.10 秒程度）が異なっているのは、tomogram を LZW 圧縮して TIFF ファイルに書き込んでいるためだ思われます（再構成の処理をしない "cbp.cu.none" の EndCBP()が返す画像では１画素以外はすべて CT 値０なので、LZW 圧縮によってファイルサイズがべらぼうに小さくなる)。つまり、N=2000、M=3600 の複数スライスを hp2tg on sp8ct +C1060 で再構成した時のスライス当たりの正味の処理時間は以下のような値になります：

関数	平均値	最小値（「最速値」）
read_SG()	0.66 秒	0.62 秒
BeginCBP()	0.52	0.52
EndCBP()	0.04	0.04
store_TG()	0.30	0.24

さて、hp2tg では BeginCBP()と EndCBP()の間に GPU で関数 BP*()が走っており、その待ち時間に CPU が read_SG()と store_TG() を並列に実行しています。080411a では他と比べて EndCBP() の処理時間が十分小さい（EndCBP() は GPU による BP* ()の処理待ちをしていないように見える）ので、

read_SG() と store_TG() の正味の処理時間の和 > BP*() の処理時間

のはずです。また、「sequential processing」の動作テストで得た画像再構成の処理全体 (== BeginCBP()+BP*()+EndCBP()) の時定数 τ は 0.08 nano sec. 程度なので、081011a のスライス 1 枚当たりの正味の処理時間は計算上は 0.08×2000×2000×3600/10^9=1.152 秒程度になります。これより、080411a では

GPU 用の関数 BP*() の正味の処理時間 = 1.152-(0.52+0.04)=0.592 秒程度

と予想されます（この値は確かに read_SG() と store_TG() の正味の処理時間の和 0.62 + 0.24 = 0.86 秒よりも小さいです)。

注

画像再構成の処理全体の τ の値は「理論値」に近いものなので、上の計算ではそれぞれの関数の正味の処理時間として 080411a の「最速値」 を使いました。

(2) hp2tg_t

やっと hp2tg_t の話です。まず、CBP engine for CUDA の関数 BeginCBP() のコードを以下のように 2 分割しました（ただし、従来の画像再構成用のメインプログラムのコードをそのまま使えるように、CBP engine for CUDA には関数 BeginCBP() = PrepareCBP() + ExecuteCBP() も残してあります）:

関数 PrepareCBP()

関数 InitCBP() が返す共有メモリ（P）に格納されている投影データを GPU にコピーする（コピーするだけで他に何もしない)。

関数 ExecuteCBP()

GPU を用いた畳み込み（Convolution）演算を行い、関数 BP*()を起動して GPU に逆投影（Back Projection）演算を依頼する。

また、以下の関数は以前のものと同じですが、念のため説明しておきます :

関数 EndCBP()

関数 BP*()の処理の終了を待ち、GPU から再構成画像を共有メモリにコピーする。その後、その共有メモリ（F）のアドレスを返す。

注

CBP engine for CUDA を含むぼくが書いた CBP 法のライブラリ関数の変遷については添付ファイル "cbps.pdf" をご覧ください。

この最新版の CBP engine for CUDA のソースファイル（"cbp.h"と"cbp.cu"）とプログラム hp2tg_t のソースファイル"hp2tg_t.c"は、以前に紹介した書庫ファイルに入れてあります：

http://www-bl20.spring8.or.jp/~sp8ct/tmp/cuda_cbp.taz

"hp2tg.c" と比較すればわかりますが、"hp2tg_t.c" ではメインスレッド（関数 main()）の中で行っていた read_SG() と store_TG() のそれぞれの処理を別のスレッド(ThreadP() と ThreadF()）で実行しています。そのために main()の中にあった局所変数のいくつかを外に出して（"hp2tg_t.c"でのみ有効な）静的変数にしました。また、POSIX thread（pthread）のライブラリを使ってマルチスレッドの制御を行っています。main() にあったコードが減った分、hp2tg_t では hp2tg に比べてプログラムの可読性は良くなったような気がします。

hp2tg_t の 3 個のスレッドの処理内容は概ね以下の通りです：

スレッド ThreadP()

for z = z1 ～ z2

共有メモリ P に以前に置いた投影データが残っていないかを調べ、もし残っていれば、それが使われるまで待つ。

read_SG(z) によって P に新しいデータをコピーする。

P にデータを置いたことを記録し、それを知らせるシグナルを発信。

スレッド ThreadF()

for z = z1 ～ z2

共有メモリ F に新しい再構成画像データが用意されているか調べ、もしなければ、それが用意されるまで待つ。

store_TG(z) によって F のデータを TIFF 画像ファイルに格納。

F のデータを使ったことを記録し、それを知らせるシグナルを発信。

メインスレッド main()

マルチスレッド処理とそれらの「同期」処理の初期化

for z = z1 ～ z2

P に新しい投影データが用意されているかを調べ、もしなければ、それが用意されるまで待つ。

PrePareCBP() によって P のデータを GPU にコピーする。

P のデータを使ったことを記録し、それを知らせるシグナルを発信。

ExecuteCBP() によって GPU を使った画像再構成を行う。

F に以前に置いた再構成画像のデータが残っていないかを調べ、もし残っていれば、それが使われるまで待つ。

EndCBP() によって GPU から F に新しいデータをコピーする。

F にデータを置いたことを記録し、それを知らせるシグナルを発信。

マルチスレッド処理とそれらの「同期」処理の後始末

上記のようにメインスレッドと ThreadP およびメインスレッドと ThreadF は共有メモリ P と F のそれぞれを介してデータ（投影データと再構成画像データ）のやりとりをします。実は、hp2tg_t で行っている P と F のそれぞれに関する 2 組の「同期」処理はいずれも、並列処理で有名な「生産者・消費者問題」そのものなので、ここでは説明を省略します。

以前と同様にして測定 080411a のデータを hp2tg_t で処理してみました。Linux の time コマンドで測った処理時間などは以下の通りでした：

sp8ct + C1060 + CUDA 3.1 + cbp.cu.8 + SDD / HDD（time_h_s_t.log）

3597.565u 165.238s 26:37.82 235.4% 0+0k 960+30757736io 0pf+0w

sp8ct + C1060 + CUDA 3.1 + cbp.cu.8 + HDD / SDD（time_s_h_t.log）

3598.755u 165.412s 26:38.35 235.5%　0+0k 960+30758024io 0pf+0w

このように hp2tg_t の正味の処理時間は 26 分 38 秒でした。これは添付したファイル "hp2tg_time.txt" に記されている最速値（32 分 40 秒）よりも十分高速で、また、以前に CUDA 2.0 の環境で hp2tg のテストをした時に最速だった run d の値（25 分 9 秒）に近い

です。なお、hp2tg_t による 080411a のスライス１枚あたりの正味の処理時間は 1598 / 1312 = 1.218 秒であり、これは時定数 τ が 0.08 nano sec. の場合の計算値（1.152 秒）と概ね一致します。

(3) hp2tg と hp2tg_t の並列処理の速度に関する考察

hp2tg と hp2tg_t の処理速度についてもう少し深く考えるため、添付した PDF ファイル "hp2tg.pdf" の図を書いてみました。これは sgs2tgs で行っている「sequential processing」、hp2tg による「pararell processing」、そして、hp2tg_t を使った「multi-threading + pararell processing」によって４枚のスライスを処理した場合を示す模式図です。その１ページ目に示したものは read_SG()と store_TG() による正味の処理時間の和が BP*() の値より小さい場合、２ページ目はその逆の場合で、処理時間を示す縦方向の長さは適当ですが、同じ関数を囲んだ箱の縦辺長（== 処理時間）はページごとに揃えてあります。

これらの図から以下のことが明らかです：

"hp2tg.pdf" の１ページ目に示した
read_SG() と store_TG() の処理時間が BP*() の値より小さい場合
　hp2tg による処理時間と hp2tg_t による処理時間は同程度になる。

"hp2tg.pdf" の２ページ目に示した
read_SG () と store_TG() の処理時間が BP*() の値より大きい場合
　hp2tg よりも hp2tg_t の方がずっと高速に処理を行える。

つまり、いずれの場合でも hp2tg_t の処理は hp2tg よりも高速になります。

次に、read_SG() と store_TG()の処理時間が BP*()の値と同じになる状況を考えてみます。read_SG()は sinogram の読み込み、store_TG は tomogram の書き込み処理なので、それぞれの処理時間以下のようになるはずです：

Tr = read_SG()の処理時間 = sinogram の画素数に比例 = tr×N×M
Ts = store_TG()の処理時間 = tomogram の画素数に比例 = ts×N×N

また、BP*()の処理時間は画像再構成全体と同じ比例関係になるはずです：

Tb = BP*()の処理時間 = 画像再構成全体の場合と同じ = tb×N×N×M

前記の 080411a のデータ処理で得た値（「最速値」や「計算値」）を使うと、これらの比例定数 tr、ts および tb の値は以下のようになります。

tr = 0.62 × 10^9 / (2000×3600) = 86.11 nano sec.
ts = 0.24 × 10^9 / (2000×2000) = 60 nano sec.
tb = 0.592 × 10^9/(2000×2000×3600) = 0.041 nano sec.

そして、Tr + Ts = Tb となる状況では、

tr × M + ts × N = tb × N × M

M = N の場合を考えると上式は解けて、

N = M = L = (tr + ts) / tb = (86.11 + 60) / 0.041 = 3554

結局、N = M なら Tr + Ts は N^2 に、Tb は N^3 に比例するので、

N < L なら Tr + Ts > Tb（"hp2tg.pdf" の２ページ目に示した場合）
N > L なら Tr + Ts < Tb（"hp2tg.pdf" の１ページ目に示した場合）

言い換えると、N が 3500 画素程度以下の画像再構成では hp2tg_t による処理の方が hp2tg による処理よりずっと速くなり、N がそれ以上の画素数でもこの hp2tg_t と hp2tg の処理速度の関係が逆転することはないと思われます。

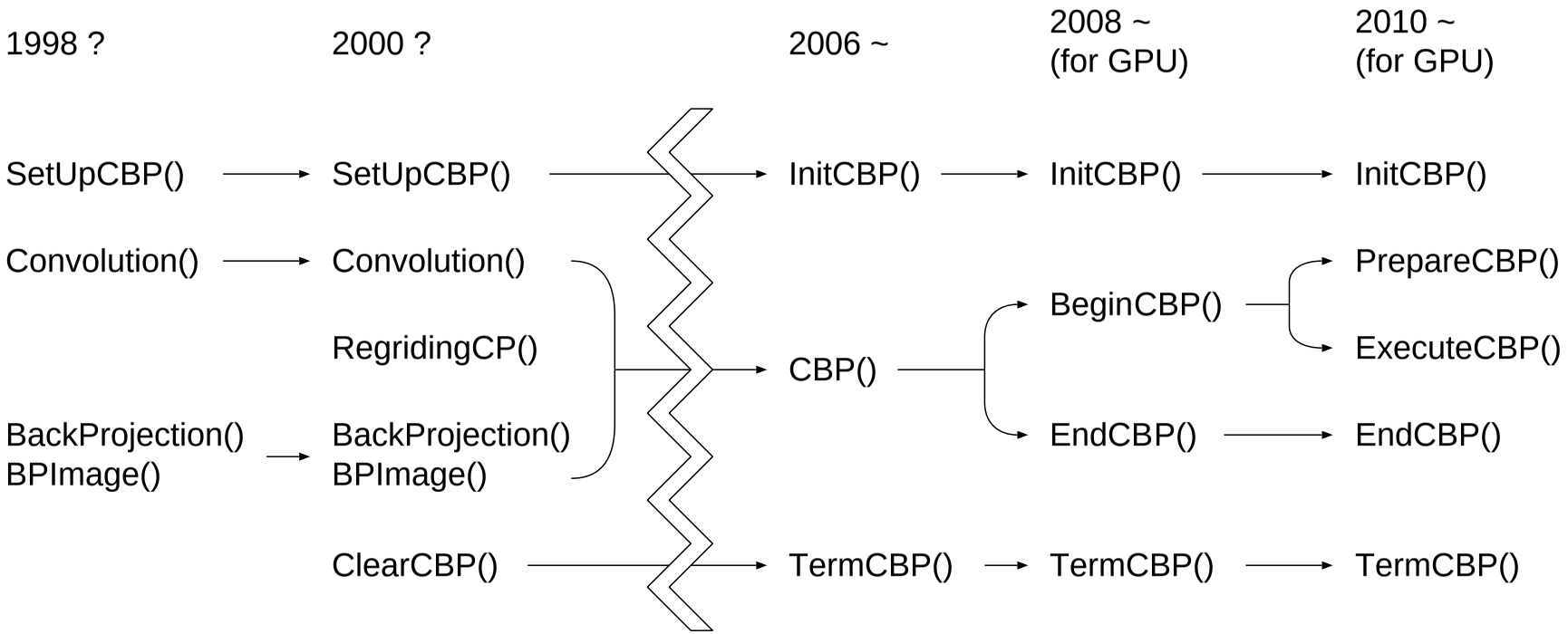
長い E-mail になりました。とりあえず、以上です。

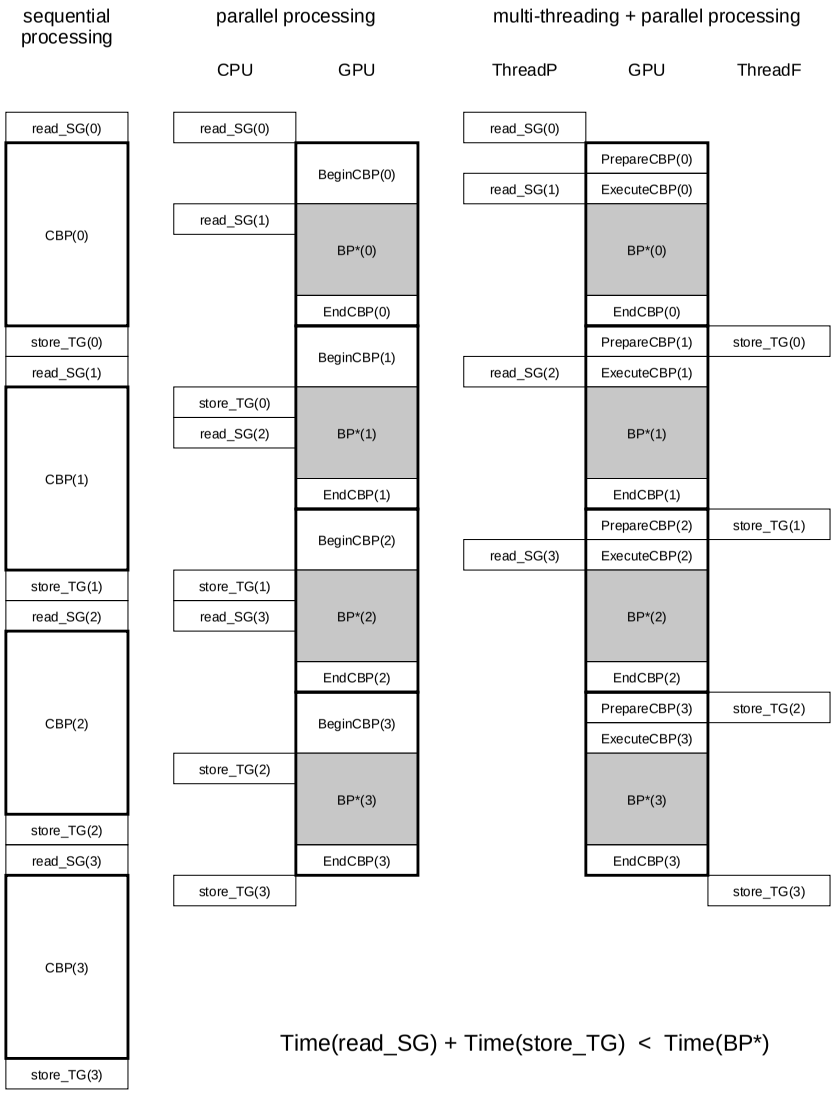
添付ファイル “hp2tg_time.txt” :
添付ファイル “cbps.pdf” :
添付ファイル “hp2tg.pdf” :

```
time of hp2tg for 080411a (N=2000, M=3600, Z=1312)

  gsjkic + C1060 + CUDA 3.1 + cbp.cu.7 + HDD_work2 / HDD_tmp (hp2tg_sec.log)
    1821.486u 193.349s 1:20:46.25 41.5% 0+0k 37741240+15377384io 5pf+0w

    function    total sec.      min. & max. sec.        mean & SD sec.
    read_raw    3686.335732     0.615601 35.418172      2.809707 4.102648
    BeginCBP()  679.268560      0.517292 0.521510       0.517735 0.000432
    EndCBP()    51.377560       0.038459 0.344469       0.039160 0.008439
    store_tg    385.760919      0.243003 0.409148       0.294025 0.024402
----
  gsjkic + cbp.cu.none + HDD_work2 / HDD_tmp (hdd_none_sec.log)
    989.709u 30.899s 1:02:31.29 27.2%   0+0k 37770440+399112io 11pf+0w

    read_raw    3585.656138     0.613004 35.415992      2.732970 3.852002
    BeginCBP()  0.000377        0.000000 0.000002       0.000000 0.000000
    EndCBP()    0.000453        0.000000 0.000001       0.000000 0.000000
    store_tg    122.027976      0.092703 0.132195       0.093009 0.001117

  gsjkic + cbp.cu.none + SSD / HDD_tmp (ssd_none_sec.log)
    967.065u 62.884s 30:32.64 56.1%     0+0k 37745273+399048io 4pf+0w

    read_raw    1684.986602     0.607598 22.415495      1.284289 1.729157
    BeginCBP()  0.000361        0.000000 0.000001       0.000000 0.000000
    EndCBP()    0.000477        0.000000 0.000001       0.000000 0.000000
    store_tg    123.251268      0.093505 0.132727       0.093942 0.001340
```

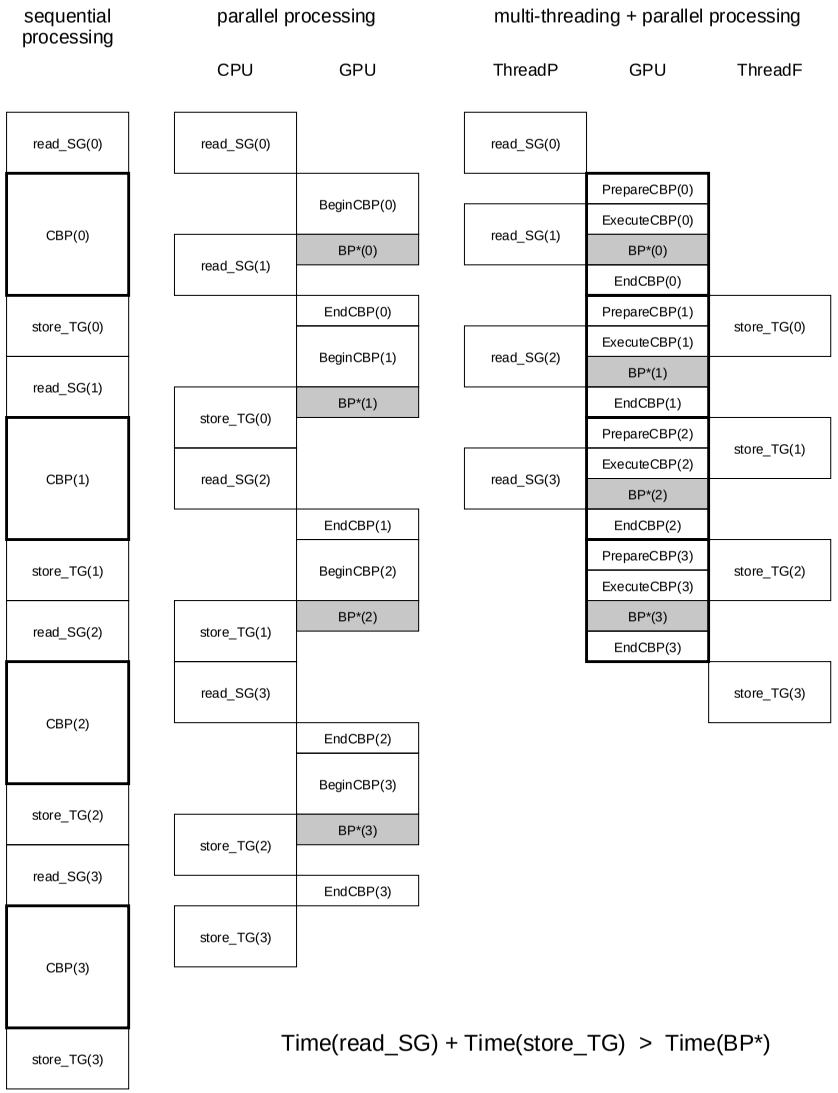
```
  sp8ct + C1060 + CUDA 3.1 + cbp.cu.7 + MD / LV (hp2tg_sec.log)
    1793.125u 165.817s 32:39.45 99.9%   0+0k 0+15377024io 0pf+0w

    function    total sec.      min. & max. sec.        mean & SD sec.
    read_raw    856.880225      0.615148 0.702900       0.653110 0.016404
    BeginCBP()  673.195652      0.511831 0.515236       0.513106 0.000457
    EndCBP()    45.769866       0.034230 0.338838       0.034886 0.008399
    store_tg    382.821357      0.239903 0.426920       0.291785 0.026144
----
  sp8ct + C1060 + CUDA 3.1 + cbp.cu.7 + SDD / HDD (s_h_sec.log)
    1792.632u 168.417s 32:42.22 99.9%   0+0k 488+15376848io 0pf+0w

    function    total sec.      min. & max. sec.        mean & SD sec.
    read_raw    860.024623      0.616573 0.708213       0.655507 0.016152
    BeginCBP()  673.282261      0.511760 0.514284       0.513172 0.000352
    EndCBP()    45.696611       0.034207 0.337969       0.034830 0.008378
    store_tg    382.403638      0.238261 0.343409       0.291466 0.026037

  sp8ct + C1060 + CUDA 3.1 + cbp.cu.7 + HDD / SDD (h_s_sec.log)
    1794.231u 167.882s 32:42.43 99.9%   0+0k 480+15376848io 0pf+0w

    function    total sec.      min. & max. sec.        mean & SD sec.
    read_raw    860.331965      0.615874 0.703047       0.655741 0.016119
    BeginCBP()  673.313020      0.512127 0.514257       0.513196 0.000354
    EndCBP()    46.221293       0.034358 0.338518       0.035230 0.008384
    store_tg    381.765339      0.238669 0.328004       0.290980 0.025852

  sp8ct + C1060 + CUDA 3.1 + cbp.cu.7 + SDD / SDD (s_s_sec.log)
    1793.465u 167.693s 32:41.46 99.9%   0+0k 488+15376848io 0pf+0w

    function    total sec.      min. & max. sec.        mean & SD sec.
    read_raw    860.080371      0.617604 0.709312       0.655549 0.016315
    BeginCBP()  673.246158      0.511836 0.514314       0.513145 0.000338
    EndCBP()    45.742374       0.034264 0.339641       0.034865 0.008424
    store_tg    381.594849      0.238919 0.328048       0.290850 0.025831

  sp8ct + C1060 + CUDA 3.1 + cbp.cu.7 + HDD / HDD (h_h_sec.log)
    1792.305u 169.292s 32:42.87 99.9%   0+0k 480+15376856io 0pf+0w

    function    total sec.      min. & max. sec.        mean & SD sec.
    read_raw    860.257024      0.616400 0.708385       0.655684 0.016151
    BeginCBP()  673.320824      0.511942 0.514749       0.513202 0.000349
    EndCBP()    45.789928       0.034260 0.338335       0.034901 0.008387
    store_tg    382.705669      0.239470 0.349653       0.291696 0.026239
----
  sp8ct + cbp.cu.none + SDD / LV (none_sec.log)
    969.672u 21.922s 16:31.79 99.9%     0+0k 16+398968io 0pf+0w

    read_raw    860.175387      0.616798 0.700756       0.655621 0.015874
    BeginCBP()  0.000466        0.000000 0.000006       0.000000 0.000001
    EndCBP()    0.000585        0.000000 0.000006       0.000000 0.000001
    store_tg    130.981645      0.098801 0.124678       0.099834 0.001001
```

Changing CBP functions





Time(read_SG) + Time(store_TG) < Time(BP*)



Time(read_SG) + Time(store_TG) > Time(BP*)