大規模ステンシル計算のための Flash SSD 向け テンポラルブロッキングの性能評価

緑川 博子[†1]　　丹 英之[†1]

計算サーバの主メモリ(DRAM)サイズを超える規模のステンシル計算を処理する場合に，Flash SSD を DRAM メモリの下位にある大容量の遅いメモリとして用いるための手法を検討，評価した． DRAM と Flash SSD のアクセス性能ギャップを埋めるために，新たに Flash 向けのテンポラルブロッキングアルゴリズムを導入し， fastswap を実装したカーネル 3.13.0 において， swap，mmap，aio の 3 種の手法による性能を調査した． その結果，3 次元格子データの近傍 7 点ステンシル計算において，主メモリ（32GiB）サイズの 16 倍程度（512GiB 超）の問題を， DRAM のみを用いて処理した場合の 40%から 70%程度の性能で処理できることがわかった．

An Evaluation of Flash-based Temporal Blocking Algorithms for Large-scale Stencil Computations

HIROKO MIDORIKAWA[†1]　HIDEYUKI TAN[†2]

A locality-aware, hierarchical out-of-core computation algorithm by employing data structure blocking techniques in stencil computations is proposed to bridge the DRAM-flash latency divide. Three different implementations are compared on Linux kernel 3.13.0. It reveals that a 7-point stencil computation for a 512-GiB problem size (16 times that of the DRAM), using only a 32-GiB DRAM and a flash SSD, in Mflops attains form 40% to 70% of the performance achieved in execution using only DRAM.

1. はじめに

ステンシル計算は, 様々な工学･科学技術計算において，広く使われ，最も重要な基本計算カーネルの一つである．典型的には，時間ステップ毎に，対象とする領域（たとえば 3 次元格子データ）全体を走査して, 各データに対して，一定の更新処理（たとえば，ある格子点データを，近傍格子点データを用いた計算により更新する）を繰り返し行うものである．多くの科学技術計算がそうであるように，ステンシル計算を含む応用処理分野では，より大きな規模の問題，より高い解像度の処理が，求められるのが常で，データ処理のための大容量メモリへの要求も留まるところを知らない．これまで，HPC 向けクラスタにおけるノード当たりのメモリ容量とノード数の増強という方向で，この要求を満たしてきた．しかし，今後，多数ノードからなるシステムにおいて, 各ノードに大量の DRAM を搭載することは消費電力の観点からも難しい．

現在，様々な不揮発性メモリ（NVM）の開発，実用化が盛んになっている．Resistive Ram (ReRAM), Spin Transfer Torque RAM (STT-RAM)，Phase-Change Memory（PCM）などの様々な不揮発性メモリが研究・実用化されつつあり，今後主メモリを構成する大きな担い手となると考えられる．すでに広く普及した Flash メモリは NVM の中ではアクセス性能が速いほうとはいえないが，従来のハードディスクに比べると 3 桁くらい高速であり，大容量のハードディスクと高速なフラッシュメモリを組み合わせたハイブリッド型のデバイスも利用されている．また，ReRAM や STT-RAM と Flash メモリを組み合わせて，Flash の弱点である耐久性（書き込み劣化による寿命）を向上させるデバイスなどの実用化も始まっている．また，メモリを組み込んだ CPU チップの開発や，GPU と CPU のメモリを統一的に扱える仕組みなども作られつつある．

本報告では，現在最も広く普及した Flash SSD を大規模ステンシル計算に利用するための手法について検討する．PCIe バス接続型の Flash SSD であっても，図 1 に示すように DRAM との性能ギャップは 1000 倍近くあり，Flash SSD を「遅いが大容量のメモリ」として利用するには，汎用的な手法だけその差を埋めることは難しい．このため，ステンシル計算の時間ステップの繰り返し更新処理に着目し，データアクセスの時間的局所性を高めるためのテンポラルブロッキングをステンシル計算に取り入れる．

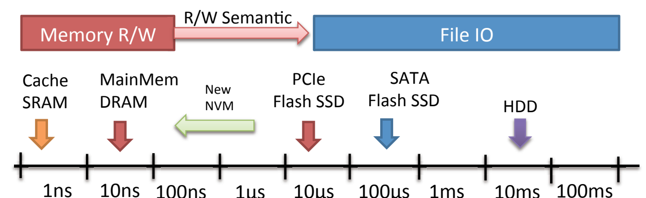


図 1　アクセス性能

†1 成蹊大学，JST CREST
Seikei University, JST CREST
†2 (株)アルファシステムズ，JST CREST
Alpha Systems Inc.，JST CREST

Figure 1　Latency in various devices.

ステンシル計算にテンポラルブロッキングを用いる試みは新しいものではなく，これまでも，キャッシュとDRAM 間，GPU メモリとホストメモリの間[1][2][3]，クラスタシステムにおけるローカルノードとリモートノード間[4]など，様々な階層おいて，アクセス性能のギャップを埋めるために，研究されている．中には NUMA とマルチコアなど特定なアーキテクチャに依存した詳細なチューニングを行うもの[4][5][6]や，テンポラルブロッキングに伴う冗長計算や余分なメモリ領域の使用の低減や，中間結果の再利用などの様々な工夫を行っているものも提案されている．

本報告では，あらたに DRAM と Flash SSD 間に対応したテンポラルブロッキングのアルゴリズムを提案する．ここで示すのは基本的なアルゴリズムで，性能向上のための詳細なチューニングは行っていない．本報告では，この基本的なテンポラルブロッキングによるステンシル計算を，３つの実装方式により性能を比較した．これまでの OS はハードディスクをはじめとするこれまでの遅い IO 装置を想定したつくりになっており，現在，改良されつつあるものの，高速な Flash などのストレージに十分に対応しているとは言い難い．ここでは，現状で利用できる３つの機能を使って，その性能について調査した．

2. Flash 向けブロッキングアルゴリズム

2.1 データアクセス局所性を意識したステンシル計算

ここで提案する Flash SSD 向けステンシル計算アルゴリズムには，２つのブロッキング手法を用いている．その一つは，行列乗算などでもおなじみの空間ブロッキング（Spatial Blocking）で，小さな空間領域に処理を分割して計算を進め，空間的なデータアクセス局所性を高めることで，キャッシュヒットを上げる．図２は，2 次元格子に対する近傍５点を用いるステンシル計算の例で，空間ブロッキングの有無による処理順序の違いを示している．空間ブロッキングでは，図２の例のように，一つの小ブロックの更新が終わると，次の小ブロックに処理を移していく．①から⑨の順にすべての小ブロックの更新が終了すると，１回の時間ステップの処理が終了する．典型的な実装としては，図３のように，領域全体が入るバッファ領域を２つ用意し，時間ステップ毎に，読み出し・書き込みバッファを交換しながら，領域全体を繰り返し更新する．

もう一つブロッキング手法は，ステンシル計算の時系列時間ステップ毎のデータ更新，時間的繰り返しにおける局所性を高める時間ブロッキング（Temporal Blocking，テンポラルブロッキング）である． これは，空間ブロッキングで分割された小ブロックにおいて，図２のように，一つの小ブロック更新が１回終わると，次の小ブロックに処理を移すのではなく，一つの小ブロックに対して複数回の時間ステップ（たとえば bt 回分）の更新をまとめて行ってから，次の小ブロックへ処理を移すという手法である．これにより，空間ブロッキングだけでは，例えば nt 回，全領域を走査していたものが，nt/bt 回の走査に減らすことができる．ただし，計算過程で bt 回の更新に必要な小ブロックの周囲の近傍データが必要なため，通常の空間ブロッキング処理よりも，大きい領域をリードして，小ブロックの計算に利用する必要がある．

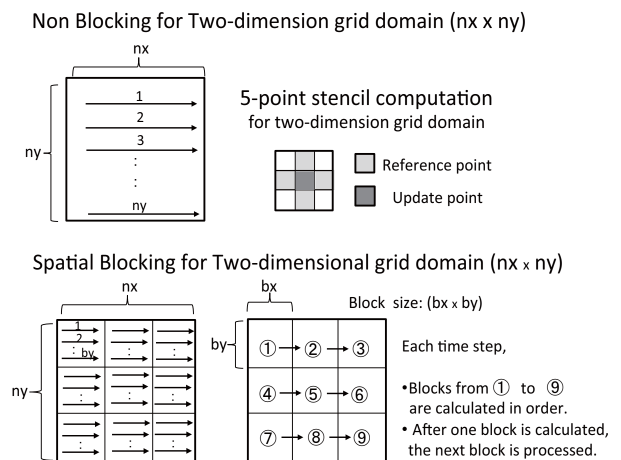


図２　２次元領域に対する空間ブロッキング処理

Figure 2 Non-blocking and spatial blocking

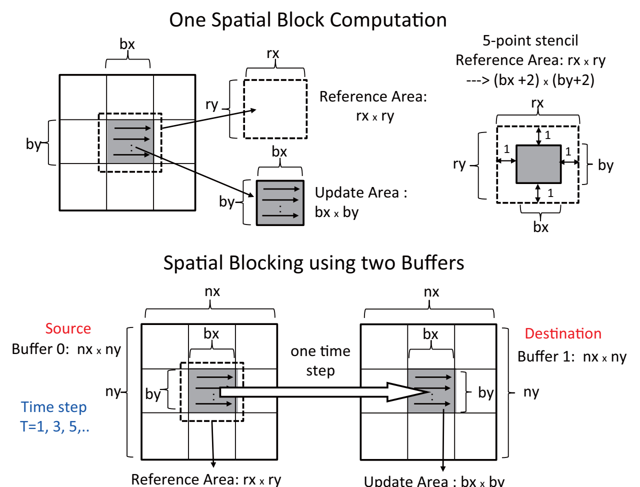


図３　ダブルバッファを用いた空間ブロッキング処理

Figure 3 Spatial blocking using two buffers

図４は bt が２（時間ステップ）の場合の一つの小ブロックでの計算過程を示している．内部時間ステップとして，２ステップの時間更新が行われる．図５は，典型的な実装を示している．領域全体を格納する２つのバッファ領域に

加え，時間ブロッキング処理に必要な小ブロックの近傍データを含む２つのブロックバッファを用いて，小ブロック内の更新計算を行う．

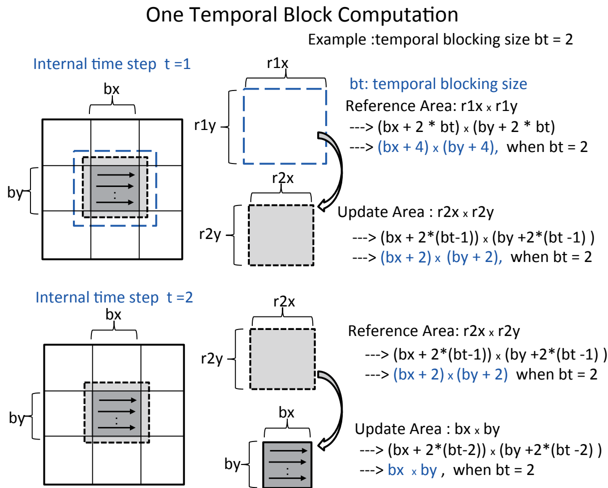


図４　小ブロックに対する時間ブロッキング処理

Figure 4 One temporal block computation:

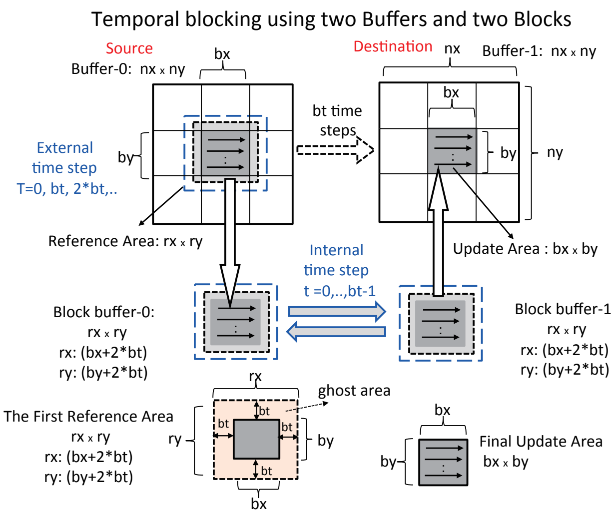


図５　バッファとブロックバッファを用いた時間ブロッキング処理

Figure 5 Temporal blocking using two buffers and two block buffers

2.2 Flash－DRAM テンポラルブロッキングアルゴリズム

本報告で用いる Flash 向けのテンポラルブロッキングによるステンシル計算では，３次元格子領域を想定しており，図６のようなデータ構造を用いている．２つのバッファ領域（Buffer-0 と Buffer-1）が用意されており，１つのバッファ（たとえば Buffer-0）に対象とする３次元領域データをセットして計算を開始する．主メモリサイズを超える問題サイズを扱うので，２つのバッファ領域は，今回 Flash SSD に置かれることを想定している．一方，テンポラルブロックによる計算を行うための小ブロック領域としては，２つのブロックバッファ（Block-0 と Block-1）が用意されており，これは DRAM メモリ上におく．スワップデバイスとして Flash を用いる場合には，このブロック領域は mlock してメモリに固定する．この２つのブロックバッファを用いて，テンポラルブロックとして，bt 回の時間ステップの更新を行った後，読みだしてきたバッファとは別のバッファに書き戻す．

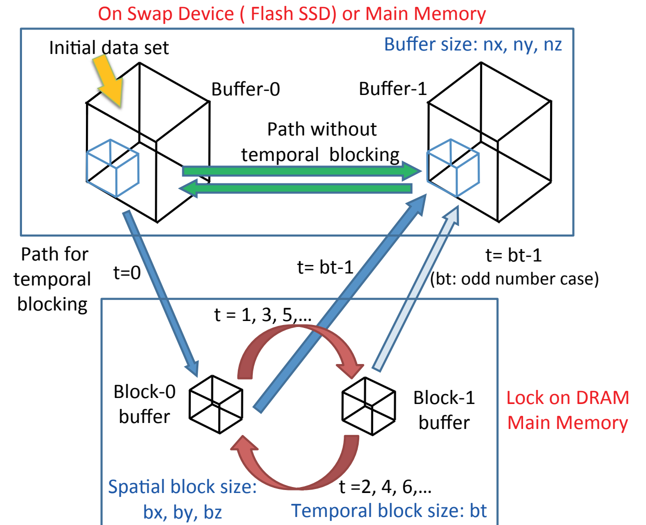


図６ DRAM と FlashSSD を用いた時間ブロッキング計算

Figure 6　Calculating data flow in temporal blocking for flash and DRAM tiers.

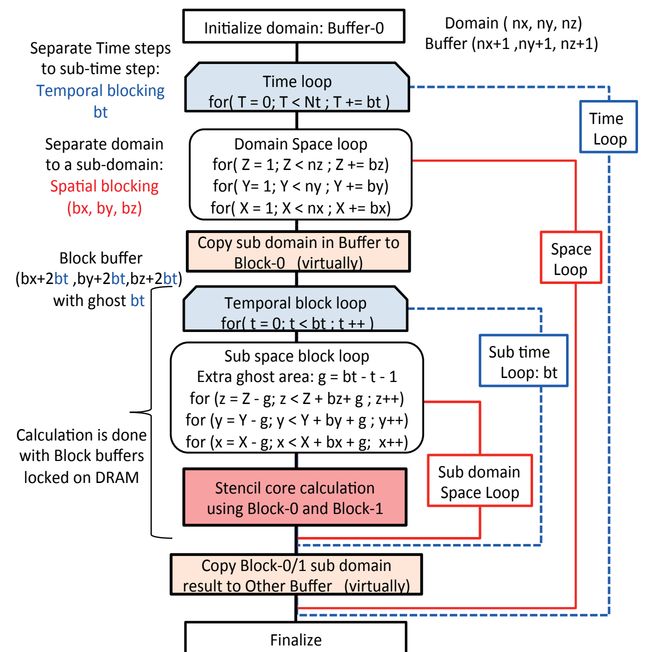


図７　３次元領域に対する時間ブロッキング計算

Figure 7 1-level temporal blocking algorithm: pseudo codes for a 3D domain.

全体の処理の流れは図７に示す．空間，時間ともに２重ル

ⓒ 2014 Information Processing Society of Japan

ープとなり小空間ブロック，小時間ブロックに分割されて処理されることがわかる．図７は，テンポラルブロッキングによる１レベルのブロッキング処理を示し，DRAM と Flash 間の性能ギャップを埋めることを目的としている．

3. Flash 向けステンシル計算の予備実験

Flash と DRAM 間に，テンポラルブロッキングアルゴリズムを適用するにあたり，パラメタ設定のための予備実験をおこなった[7]．表１に実験環境を示す．Flash SSD としては，PCIe バス接続型の ioDrive2[10]を用いている．本報告では，３次元格子データにおける近傍７点のステンシル計算を扱うことにする．

表 1　実験環境

Table 1　Experimental Environment.

CPU	Xeon E5-2650 2.00GHz x 1socket (8cores)
Memory	128GiB, 16GiB(DDR3 1600MHz ECC Reg) x 8 32GiB/128GiB boot
OS kernel	Linux kernel 3.13.0
Compiler	gcc 4.4.7 20120313　 -O3
Flash SSD	ioDrive2 MLC , 1.2TB, PCIe2.0 x 4 (FusionIO)

3.1　空間ブロックサイズと時間ブロックサイズ

テンポラルブロッキング処理では，時間局所性を高めるために，小ブロックに対し，bt 回分の時間ステップ更新処理を行うので，図５に示すように本来の更新データ部分（小空間ブロック）周りに次元ごとに両側に bt ずつの幅の近傍データ（袖領域）が必要である．このため，時間ブロックサイズ bt が大きければ，時間的メモリアクセス局所性が向上する性能上のメリットがあるが，一方で，袖領域のための余分なメモリ量と冗長計算が増加するというデメリットもあり，bt の設定にはトレードオフが存在する．

また，限られたサイズの DRAM メモリにのせるブロックバッファサイズに対して，時間ブロックサイズ bt と空間ブロックサイズ（bx,by,bz）をどのような割合にするかによっても，性能が変わってくる．また，同じサイズの空間ブロックであっても，その形状によりアドレス連続性などから性能に違いが出る．

図８は，近傍７点の３次元ステンシル計算で 64GiB 超の問題（2048x2048x1024，繰り返し 256 回，８スレッド)を，128GiB のメモリ上で処理した場合の，時間ブロックサイズ（bt：1~256）と空間ブロックサイズ（bx,by,bz，１辺 32~1024 の立方体形状）の違いによる性能の変化を示している．これによると，32 x 32 x 32 ～64 x 64 x 64 程度の空間ブロックサイズで時間ブロックサイズ 4~8 程度が最適であることがわかる．この実験は，表１の環境を用いており，L3 キャッシュ 20MB と DRAM 間での最適なサイズを示していることになる．

同じ問題を，今度は DRAM メモリを 32GiB に制限し，Flash SSD をスワップデバイスとして用いて実験した場合の結果を図９に示す．DRAM とキャッシュ間の場合とは異なり，冗長計算のオーバヘッドに比べて Flash SSD への入出力は非常に時間がかかるため，空間ブロックサイズは，メモリに収まる最大限のサイズが有利で，時間ブロックも大きいほど良いが，32GiB のメモリ入る最大の空間ブロックサイズは，1024x1024x1024 (double)で，これに時間ブロック bt＝128 を用いた時が最高性能となった．bt=256 になると，袖領域が増えて 32GiB のメモリに２つのブロックが入らなくなり，ブロックアクセスにもスワップが稼働して，急激に性能が低下する．（図９の右端，bx x by x bz =1024，bt=256 の点）

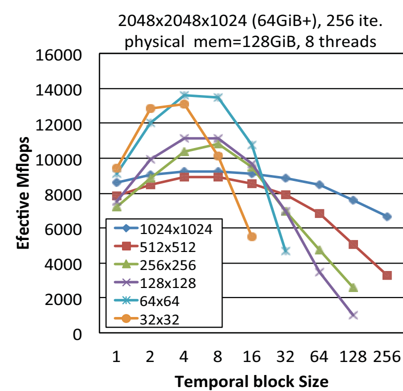


図８　空間ブロックサイズと時間ブロックサイズのトレードオフ（メモリ 128GiB：DRAM－キャッシュ間）

Figure 8　Spatial block size vs. temporal block size on full memory

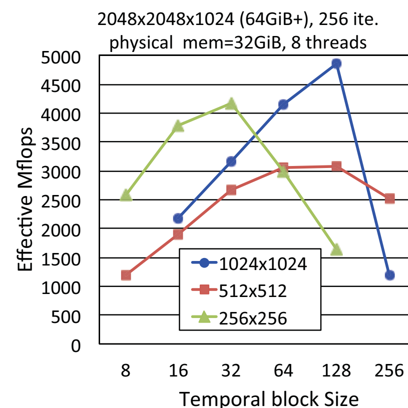


図９　空間ブロックサイズと時間ブロックサイズのトレードオフ（メモリ 32GiB＋swap：DRAM－Flash SSD 間）

Figure 9　Spatial block size vs. temporal block size on full memory

これらの予備実験により，Flash むけテンポラルブロッキングアルゴリズムでは，利用可能な DRAM メモリサイズに入る最大限の２の累乗のサイズの空間ブロックサイズと，それにつけることのできる最大の袖領域サイズを考慮して時間ブロックサイズを決め

ⓒ 2014 Information Processing Society of Japan

ることとした．また，ここでは省略しているが，小ブロックのアクセスができるだけ連続アドレスになるように，小ブロックの x 方向サイズをもとのデータ領域（バッファ）の x 方向サイズと等しくなるようにして，形状は立方体にこだわらない．ただし，各辺とも，時間ブロックサイズの少なくとも２倍以上であることとしている．

3.2 マルチレベルのブロッキング

前節に述べたように, Flash 向けのテンポラルブロッキングアルゴリズムでは，空間ブロックサイズをできるだけ大きくする必要があることから，メモリ上にあるこのブロックバッファの更新処理自体に, 今度は DRAM とキャッシュとの間の局所性を生かす必要性が生じる．図８で示したように，この実験環境では，キャッシュを意識した空間ブロックサイズとしては，32～64 程度立方体が有利である．このため, メモリ上の２つのブロックバッファの更新処理に, キャッシュヒットを意識したさらに小さい空間ブロッキングをもう１段導入し，合計，２レベルのブロッキング処理を導入することとした．そこで，128GiB のメモリ上で，１レベルのテンポラルブロッキング処理(空間ブロックサイズ 1024x1024x1024: 時間ブロックサイズ 128) と，内部に空間ブロッキング（空間ブロックサイズ 32x32x32）をさらに導入した２レベルのテンポラル＋空間ブロッキングを用いた場合の処理時間を調べた．図 10 は，8 スレッドの１レベルブロッキングの結果を基準とした相対実行時間を示している．マルチスレッドになるほど，２レベル目の空間ブロッキングにより，キャッシュの共同利用が進み，性能差が出ることがわかる．

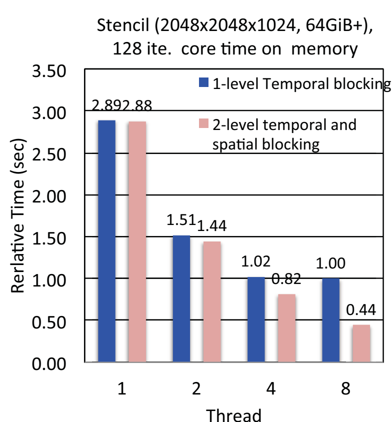


図 10 1 レベルブロッキングと 2 レベルブロッキングの比較（メモリ 128GiB：DRAM－キャッシュ間）

Figure 10 One-level blocking vs. two-level blocking on full memory (128GiB), 7-point stencil (2048 x 2048 x 1024 double 64 GB+, 256 item. 8 threads), (temporal block size = 128, spatial: 1st block size = 1024, 2nd block size = 32).

今回の環境では，メモリ上のブロックバッファの更新処理に，２レベル目のテンポラルブロッキングを導入しても，オーバヘッドが高くほとんど効果がみられなかったため，メモリ上のブロックバッファの更新計算には，空間ブロッキングのみを導入している．

4. Flash 向けステンシル計算の３つの実装方式

前節までで説明した Flash 向けテンポラルブロッキングによるステンシル計算において，領域全体データを持つ Flash 上のバッファブロックから, メモリ上にあるブロックバッファへ小ブロックデータを持ってくる手法として，３つの方式を比較した．それぞれの方式で Flash SSD をどのように利用するかを以下に示す.

(1) swap 方式：スワップデバイスとして用いる
(2) mmap 方式：ext4 ファイルシステムとして用いる
(3) aio 方式：直接 IO 用ブロックデバイスとして用いる

(1) swap 方式

Flash SSD を swap デバイスとして利用する方式である．図６の２つのバッファと２つのブロックバッファともvalloc で動的にメモリ確保する．ただし，２つのブロックバッファ（Block-0, Block-1）は，mlock することにより，スワップアウトされないようにしている．

従来の swap システムは，どちらかといえばメモリ欠乏時の緊急退避手段として実装されており，積極的に高性能なデマンドページングとして用いるためのものではなかった．マルチスレッドにも十分に対応しておらず，従来の回転式ハードディスクを意識した実装で，非常に低い性能であり，高性能計算においては，事実上使ってはいけないものと考えられることもあった．しかし，高性能な Flash ストレージの台頭を背景に，昨年 FusionIO 社により OpenNVM プロジェクトの一環として提案された fastswap[11][12][13]は，マルチスレッドでのスループット向上や，従来のハードディスクを意識したレガシー部分を排除し, Flash などを意識したデマンドページングとしてのスワップシステムとして提案され，Linux kernel 3.6.0 のパッチとして配布された．これまで，我々はこのパッチバージョンを用いた実験を行ってきた[7][8][9]が，現在は，新しい kernel に取り込まれており，今回は kernel 3.13.0 を用いて，Flash SSD をスワップデバイスとして用いて，DRAM サイズを超える問題サイズのステンシル計算を行う．

スワップ方式の利点は，アプリケーションの変更がいっさいいらない点であるが，スワップデバイスへのページの出し入れを制御することはできないため，かならずしも，アプリケーションに適したページの入出力がされると限らない．fastswap では，アプリケーションの特徴に応じて，madvise を指定すると性能向上がある場合もあるが，反対

に，性能悪化を引き起こすこともあり，指定が難しい．

(2) mmap 方式

mmap 方式とは，DRAM に入りきらないデータ領域（今回の例ではデータ領域全体を保持する２つのバッファ領域）を同じサイズのファイルとして作成し，このファイルを mmap する方式である．(1)の swap 方式の valloc 部分を上記の処理でおきかえれば，ほぼ同等のプログラムでメモリを超えるデータ処理ができる．通常，主メモリの空き領域は，他の要求がないかぎり，めいっぱいファイルのページキャッシュとして利用されるので，利用できるメモリのサイズに応じて Flash 上のファイル（今回は ext4 のファイルシステム上に作成した２つのバッファに対応する２つファイル）が展開されるため，多くのアクセスはメモリ上で行うことができて，高速である．これまでのわれわれの評価では多くのアプリケーションで，swap 方式にくらべ，mmap 方式のほうが，主メモリを超えたサイズのデータ処理は概して優位性があった．この方式の実行環境では，スワップデバイスなしとして，スワップは起動されないように設定しており，mlock しなくても，ブロックバッファは常にメモリ上にあることが保障されている．

この方式の場合も，アプリケーションの特徴に応じてページキャッシュとファイルの入出力（同期）がされるわけではない点は，swap 方式と同じであるが，利用可能メモリに多くのファイルのページが常に展開されているので，swap 方式にくらべ優位であることが多い．

（３）aio 方式

ここでは，Flash を前述２つの手法のようにあたかもメモリであるかのようにアクセスして利用するのではなく，明示的に全体領域データをもつファイル（Buffer-0, Buffer-1）から，ブロックバッファ（Block-0,Block-1）にファイルを読み込み，計算結果をファイルに書き込むという処理を行う．この方式は，アプリケーションに必要なデータだけを Flash との間でやりとりするというものである．さらに，カーネルのページキャッシュやファイルシステムのレイヤのオーバヘッドを削除するため，Flash をブロックデバイスとして O_DIRECT を指定し open して用いる．さらにマルチスレッドによる IO の並列性を高めるため，小ブロックの更新計算とのオーバーラップを行う．IO には，Linux kernel 実装による非同期 IO[14]を用いる．これは POSIX 準拠の非同期 AIO 関数とは異なり，カーネルレベルのシステムコールとして実装されており，POSIX のユーザレベルのスレッド実行とは実装が異なっており，マルチスレッドによる IO 性能の点でより優れている．

マルチスレッドが並列に複数の IO 要求を投入できるように IO キューサイズを大きくし，各スレッドが担当領域のデータをブロックバッファ（Block-0）へ読み込ませるための読み込み要求をキューにすべて投入した後，いっせいに起動をかける．ブロックの前半部分の領域の読み込みが完了した時点で，前半ブロックのステンシル計算を開始する．前半の計算が終わったところで，後半ブロックの読み込み完了を確認し，続いて後半部分の計算を行う．そのあとの時間ブロックサイズ bt の間はブロックバッファのみの更新計算になるため，ファイルの IO は全くおきない．そして bt ステップ後の最終ステップ時の更新では，前半の計算終了とともに，前半部分のファイル（buffer-1）への書き込みを起動し，並行して後半部分の計算をおこなう．すべての計算が終わってから，後半部分をファイル（buffer-1）へ書き込む．したがって，メモリ上にあるブロック間の更新計算の最初と最後の部分で，明示的な入出力部分が付加される．しかし，不要なデータの読み書きが全くないこと．さらに，ブロックバッファの形状を連続で４KB にアラインしたものにすることにより，各スレッドの１つの IO 要求を大きなサイズにすることで，Flash の限界スループットまでの多量なリード・ライト要求を投入できる．

しかし，アプリケーションには明示的なファイル入出力の記述が必要であり，計算とのオーバーラップを行う場合にはそのための記述も必要となる．また，並列 IO の効果が発揮されるにはブロックデバイスとして直接アクセスし，データの先頭はアラインされねばならないという制限などが生じる．

5. Flash 向けステンシル計算の性能評価

5.1 テンポラルブロッキングアルゴリズムの効果

図 11 は，Flash むけテンポラルブロッキング手法による効果と，手法の差を示している．64GiB 超の問題（2046x2048x2048，256 繰り返し，8 スレッド）の処理を，128GiB の十分なメモリを使って実行した場合（右の２つ）と，用いるプログラムサイズの半分の DRAM メモリ(32GiB)で実行した場合の実行時間を示す．右から２番目の NORMAL-SWAP（メモリ 128GiB）は，テンポラルブロッキングを用いないアルゴリズムを通常実行した場合を示す．すべてメモリ上で実行され，キャッシュだけを意識した 32x32x32 の小ブロックによる空間ブロッキングのみを用いて処理した場合で，1222sec で終了する．本報告で提案する２レベルのテンポラルブロッキングアルゴリズムを用いた手法を，同じく 128GiB のメモリで処理を行うと，冗長計算などによるオーバヘッドにより，少し実行時間が増えて 1373sec となる．（図中右端）

次に左の３つは，メモリを 32GiB に制限した場合で，扱う問題サイズの半分以下しかメモリがない状況で，３つの実装方式で実行した場合の実行時間を示す．通常実行（NORMAL-SWAP，メモリ 128GiB）と比較すると，swap 方式が 2.22 倍，mmap 方式が 1.58 倍，aio 方式が 1.51 倍の

ⓒ 2014 Information Processing Society of Japan

時間がかかる．このグラフに示していないが，テンポラルブロッキングを用いない NORMAL-SWAP を同じく 32GiB のメモリ上で実行すると，65715sec（18 時間 16 分）かかり，通常実行 1222sec の 53.8 倍にも達する．テンポラルブロッキングの効果が，いかに大きいかがわかる．

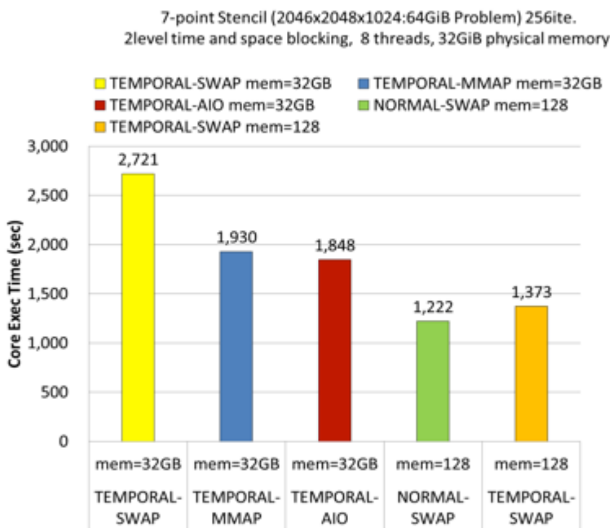


図 11　ステンシル計算における各手法の実行時間
Figure 11　Stencil Computation Time comparison

5.2　３つの実装手法の性能の比較

次に，メモリサイズを 32GiB に固定して，問題サイズを大きくしていった場合の相対実効性能を図 12 に示す. 各問題におけるバッファとブロックのサイズを表 2 に示す．実際にプログラムで使用するメモリ領域は袖領域なども含み，図 12 の横軸に示す問題サイズよりもさらに大きなサイズとなっている．図 12 の Effective Mflops 値とは，ステンシル更新部分の実際にかかった計算処理部分の時間で，本来行うべき double の演算数を除算したもので，テンポラルブロッキングアルゴリズムに伴う冗長計算による演算数の増加を考慮せずに計算した性能である．問題サイズが大きくなるので, 総実行時間は大きくなるが, この性能をもとに，問題サイズの増加によって，どの程度性能が劣化するかを調べることができる. 図 12 は, 十分にメモリがある場合（左端の 32GiB のメモリで 16GiB サイズの問題の処理）の swap 方式の実効性能を基本とした相対性能を示している．これによると，512GiB（メモリ 32GiB の 16 倍）を超えるサイズの問題であっても，メモリが十分にある場合の実効性能の約 40%から 65%程度の性能で処理ができることを示す．

swap 方式は, メモリが十分ある場合（16GiB 問題サイズ）から，少しでもスワップデバイスを利用する状況（32GiB 問題サイズ）になると性能が半分程度（0.52）に落ちてしまう. それにくらべ, mmap の性能劣化はゆるやかである．aio 方式は一定量のファイル IO が明示的に行われるため，メモリ上に問題がのっている場合（左端）であっても，一定のオーバヘッドが常にかかる．しかし，問題サイズの増加に伴う性能劣化は一番少ない．

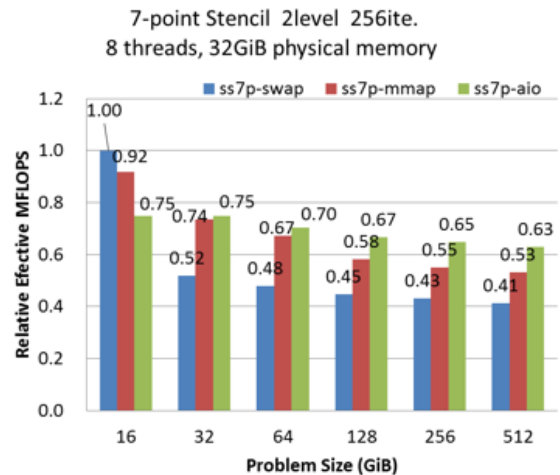


図 12　問題サイズの変化にたいする各手法の相対性能
Figure 12　Relative performance of three methods.

表 12　問題サイズと実際のデータサイズ
Table 2 Problem size and actual data size

	Spatial block size		2046 x 512 x 512		
	Temporal block size		128		
Problem Size (GiB)	Domain Shape	2 Buff Size (GiB)	Actual one Block Size (GiB)	Actual one Buffer Size (GiB)	Actual Total Size(GiB)
16	1022 x 1024 x 1024	16*	5.62	8.03	27.34
32	2046 x 1024 x 1024	32	10.12	16.06	52.36
64	2046 x 2048 x 1024	64	10.12	32.09	84.42
128	2046 x 2048 x 2048	128	10.12	64.13	148.48
256	2046 x 2048 x 4096	256	10.12	128.19	276.61
512	2046 x 409 6x 4096	512	10.12	256.25	532.73

16* : Spatial block size 1022 x 512 x 512

5.3　Flash　SSD に対する入出力状況の比較

３つの実装方式を用いた場合の Flash SSD に対する入出力状況はどのようになっているか, iostat コマンドを用いて調査したのが図 13，図 14 である．扱った問題は，図 11 で用いたのと同じ 64GiB 超の問題である．図 13 は，各手法における Flash に対するリードとライトのバンド幅を時系列で示している．図 14 は，それぞれの手法における CPU 利用率を時系列で示しており，user がほぼステンシル計算の部分に対応し, iowait が Flash 間からの入力待ちに対応する．これらの図をみると，あきらかに，それぞれの手法による Flash へのアクセスが違うことがわかる．定期的に表れている計算部分（ユーザ利用率 100%の部分）は，毎回バッファ領域からブロック領域へデータを読み込んだあとに，テンポラルブロッキングによる計算を行っている部分で，山の数は小ブロックの数に対応している．

これをみると明らかに, swap はだらだらと入出力が行われており，mmap に比べると，ピークのバンド幅も低い．また，実行中のリードデータの総サイズも，swap 方式は

mmap 方式の約３倍，aio 方式の６倍弱と多い．

一方，必要なものだけを並列に IO している aio 方式では，FlashSSD の最大バンド幅に近い性能が得られており，非常に短時間で入出力が終わっていることがわかる．IO 部分の時間をみると，時間ブロックサイズによる計算山の時間幅との兼ね合いもあるが，時間繰り返し数やブロック数が多いほど，aio 方式が有利になっていくものと思われる．

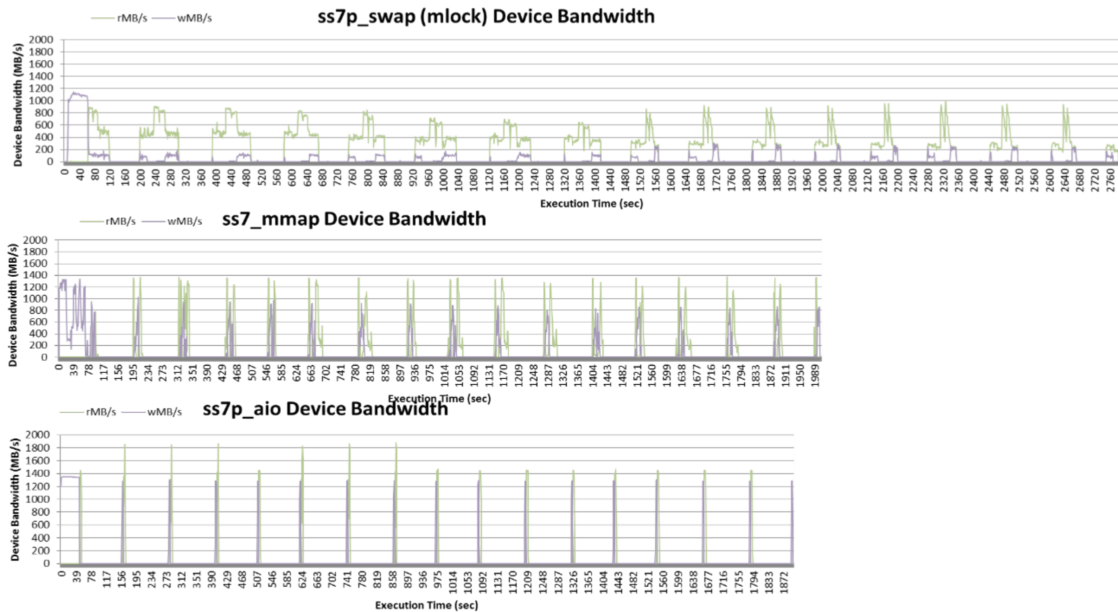
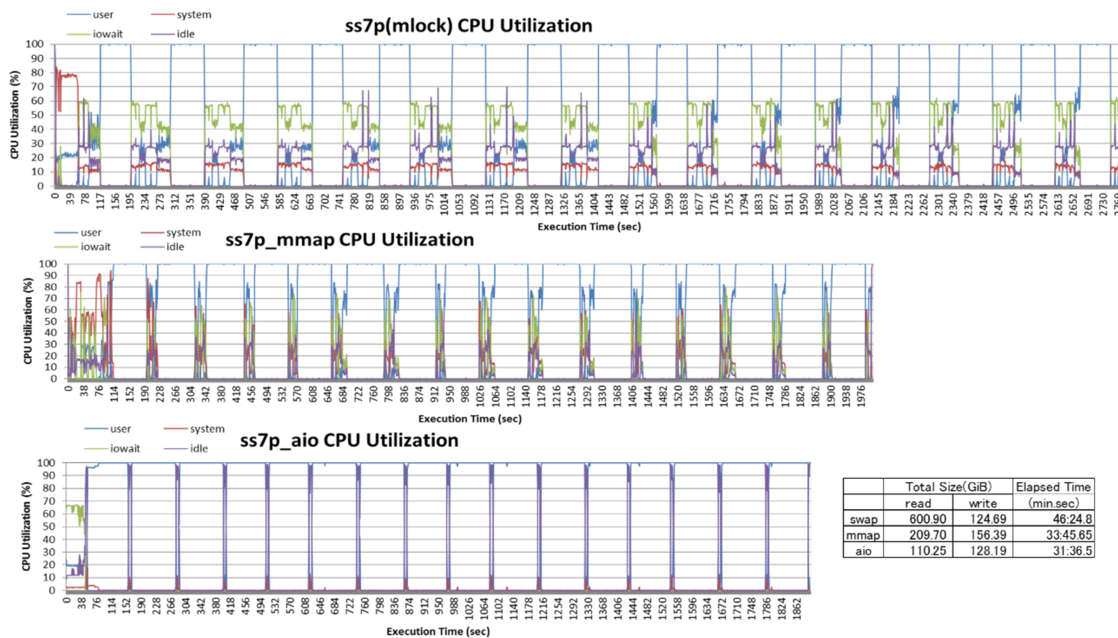


図 13　テンポラルブロッキングステンシル計算における Flash SSD へのリードライトバンド幅

Figure 13　Read/Write Bandwidth in Stencil Computations



	Total Size(GiB)		Elapsed Time
	read	write	(min.sec)
swap	600.90	124.69	46:24.8
mmap	209.70	156.39	33:45.65
aio	110.25	128.19	31:36.5

図 14　テンポラルブロッキングステンシル計算における CPU 利用状況と総データリード・ライト量

Figure 14　CPU Utilization in Stencil Computations

6. おわりに

本報告では，ステンシル計算において Flash 向けのテンポラルブロッキングアルゴリズムを提案し，３つの異なる手法を用いて実装し，比較した．Flash SSD は他の ReRAM や STT-RAM などの NVM に比べ，アクセス性能は高くない．したがって応用プログラム側になんらかの工夫を行って，アクセス局所性を高め，大規模データを処理する一つのデバイスとして用いることが，現実的な利用形態である．また，この報告では，カーネルの様々なオーバヘッドをスキップして，ブロックデバイスとして非同期 IO による並列 IO を行うことにより，デバイスの最大限の性能を引き出すことが可能であることを示した．このままの形で一般ユーザに利用させるには難があるが，応用に特化した入出力をユーザに隠ぺいした形で提供できれば，一つの有力な選択肢になると考えられる．

参考文献

1) Nguyen, A. ; Satish, N. ; Chhugani, J. ; Changkyu Kim ; Dubey, P. , “3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs”, High Performance Computing, Networking, Storage and Analysis (SC), 2010, DOI: 10.1109/SC.2010.2, 2010 , pp. 1 – 13
2) Guanghao Jin, Toshio Endo and Satoshi Matsuoka, “A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU,” Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, pp. 1080 - 1087 , 2013, DOI: 10.1109/IPDPSW.2013.58
3) Guanghao Jin, Toshio Endo and Satoshi Matsuoka, “A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs”, IEEE Cluster2013, 2013, 10.1109/CLUSTER.2013.6702633
4) M. Wittmann, G. Hager, and G. Wellein, “Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory”, Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages, April 2010, DOI: 10.1109/IPDPSW.2010.5470813
5) Shaheen, M., Strzodka, R., “NUMA Aware Iterative Stencil Computations on Many-Core Systems ”, Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, DOI: 10.1109/IPDPS.2012.50, 2012 , Page(s): 461 – 473
6) Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann and Holger Fehske, “Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization”, Computer Software and Applications Conference, vol.1, pp. 579 – 586, 2009.
7) Hiroko Midorikawa, Tan,Hideyuki, Toshio Endo, ”An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations”, IEEE The 12th International Conference on High Performance Computing & Simulation, HPCS2014, 2014
8) 丹英之,緑川博子:"フラッシュ SSD をメモリセマンティクス API で用いるための予備調査", ハイパフォーマンスコンピューティングと計算科学シンポジウム HPCS2014, HPCS2014 論文集, (2014,1)
9) 丹英之,緑川博子:"フラッシュ向け Linux スワップシステムの評価", 電子情報通信学会，コンピュータシステム研究会 Vol.113, No.282, pp.61-66, (2013,11)
10) ioDrive2（FusionIO 社）
https://www.fusionio.com/products/iodrive2/
11) Improve Linux swap for High speed Flash Storage
http://events.linuxfoundation.org/sites/events/files/lcjpcojp13_shaohua.pdf.
12) Nisha Talagala, “Creating Flash-Aware Applications”, Flash Memory Summit 2013,
http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130814_203B_Talagala.pdf.
13) OpenNVM, FusionIO, http://opennvm.github.io.
14) Kernel Asynchronous I/O (AIO) Support for Linux
http://lse.sourceforge.net/io/aio.html