Vol.2009-ARC-186 No.9
Vol.2009-HPC-123 No.9
2009/12/1

GPU向けソフトウェアキャッシュ機構の実装と評価

平澤将一[†,††] 下田和明[†,††]
大島聡史[†,††] 本多弘樹[†,††]

高性能コンピューティングにおいて GPU が注目されている．NVIDIA 製 GPU は CUDA において高性能なシェアードメモリを有効に用いるプログラミング技術により各種アプリケーションで非常に高いピーク性能が得られている一方，プログラミングの容易さ，汎用性に問題を残している．本研究においては CUDA においてユーザが明示的に使用するシェアードメモリの一部をデバイスメモリのキャッシュとするソフトウェアキャッシュ機構を提案する．本機構によりデバイスメモリからシェアードメモリへ暗黙的にデータ転送が行われ汎用計算の高速化が達成される．

A Software Cache Implementation for GPU

Shoichi Hirasawa,[†,††] Kazuaki Shimoda,[†,††] Satoshi Ohshima[†,††]
and Hiroki Honda[†,††]

In HPC, GPU attracts attention. Although programming difficulty still remains, very high peak performance can be achieved using NVIDIA GPUs. In this research, we propose a software cache mechanism which caches the device memory of CUDA with the shared memory. User data can be transfered implicitly with the software cache and performance improvement of general-purpose computation benchmark programs can be achieved.

1. はじめに

近年，画像処理用のハードウェアである GPU(Graphics Processing Unit) の性能が著しく向上した．GPU は多数の画素に対し同様の処理を行う 3D 画像処理に特化したハードウェアであり，高い浮動小数点演算性能を備えている．

NVIDIA 社は GPGPU プログラミング環境 CUDA (Compute Unified Device Architecture) を開発し，2007 年に提供を開始した．C/C++の拡張言語である C for CUDA はグラフィックス API の知識を必要とせず，GPGPU プログラミングの煩雑さを軽減している．

しかしながら，CUDA を用いて GPU の性能を引き出すためには，GPU のハードウェアアーキテクチャに適したプログラムの最適化を行う必要がある．CUDA を用いたプログラムの最適化を難しくする要因のひとつに，GPU が複数のメモリを持った分散メモリ構造をとっていることが挙げられる．

CUDA のメモリ構造には、大容量だがアクセスレイテンシが非常に大きいデバイスメモリと，容量が少ないがアクセスレイテンシが非常に小さい shared memory がある．CUDA を用いてアプリケーションの高速化を図るためには高速共有メモリを有効に利用する必要があるが，メモリの特性を理解しプログラムの最適化を行うことはプログラマの負担となる．

† 電気通信大学
The University of Electro-Communications
†† 独立行政法人科学技術振興機構, CREST

本研究では，プログラマが明示的に用いていた shared memory の一部をデバイスメモリのキャッシュとして扱うソフトウェアキャッシュ機構を提案する．ソフトウェアキャッシュ機構は，ソフトウェアによりデバイスメモリから高速共有メモリへのデータ転送を行うキャッシュの動作をエミュレートする．本機構を用いることにより，プログラマはデバイスメモリから高速共有メモリへのデータ転送を明示的に行う必要が無くなり，汎用計算の高速化を図ることが出来る．

2. GPU 上の高速共有メモリによるソフトウェアキャッシュ機構の提案と実装

2.1 CUDA プログラミングにおける問題点

CUDA を用いてアプリケーションの高速化を図るためには，プログラマは各メモリの特徴を理解し，複数のメモリを利用しなければならない．プログラマはそのメモリの中で特に低レイテンシの shared memory を有効に利用する必要がある．shared memory を用いるために，プログラマは明示的にデータ転送を行う．この処理の記述はプログラマの負担となるため、shared memory を容易に利用することを可能とする機構が望まれる．

2.2 shared memory を用いるソフトウェアキャッシュ機構の提案

shared memory を容易に利用することを可能とする環境を用意するべく，shared memory を global memory のキャッシュとして扱うソフトウェアキャッシュ機構を提案する．ソフトウェアキャッシュ機構は，ソフトウェアによってキャッシュの動作をエミュレートし，global memory か

ら shared memory へのデータ転送を行う．この機構を使うことにより，プログラマは global memory から shared memory へ明示的にデータの転送の処理を行う必要が無くなり，プログラマは shared memory を容易に利用することができる．

2.3 ソフトウェアキャッシュ機構の構成

ソフトウェアキャッシュは参照する global memory のアドレスを格納するタグと，global memory の参照データを格納するデータから構成される．タグとデータの一組をキャッシュラインと呼ぶ．1 ライン辺りのデータは複数のワードから構成されている．1 ラインあたりのデータのワード数をキャッシュラインサイズと定義する．図 1 はキャッシュライン数を 4 本，キャッシュラインサイズを 16 ワードとした場合の例を示している．

また，shared memory から構成されるソフトウェアキャッシュへのデータの格納方式にダイレクトマップ方式を採用した．

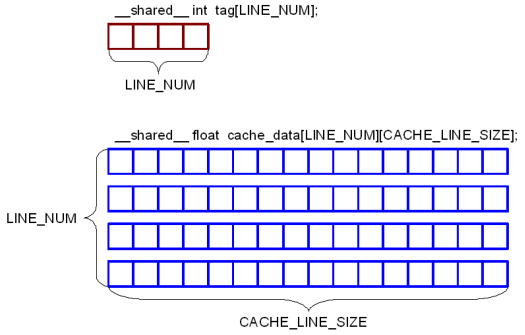


図 1 shared memory によるキャッシュの構成

2.4 ソフトウェアキャッシュ機構の仕様

ソフトウェアキャッシュ機構の仕様を以下にまとめる．

- リードオンリーキャッシュとする．
  本研究のソフトウェアキャッシュ機構は global memory 上のデータのリードオンリーキャッシュとして設計しており，キャッシュのデータを更新することを対象としていない．そのため，本機構はキャッシュに対して書き込みがなされたか否かを示すダーティフラグを持っていない．
- 本機構が対象とする global memory 上の変数は配列変数のみとする．
  ソフトウェアキャッシュ機構が提供する関数を用いて，プログラマは global memory 上に格納されている配列のインデックスを元にデータを得る．
- キャッシュ構造体のメモリ領域の確保を行う．
  CUDA ではカーネル実行中に shared memory 上のメモリ領域を動的に確保することが出来ない．よって，ソフトウェアキャッシュのメモリ領域を確保する必要がある．そのため，ソフトウェアキャッシュ機構は，ソフトウェアキャッシュを構成するタグ，データ配列，キャッシュに乗せる global memory の配列変数の先頭アドレスを記憶するポインタの 3 変数をメンバとする構造体を提供する．この構造体をソースコード中に宣言することにより，ソフトウェアキャッシュのメモリ領域を確保する．

2.5 キャッシュの動作

本節ではソフトウェアキャッシュの動作について述べる．ソフトウェアキャッシュ機構が提供する，キャッシュにデータを格納し対象のデータを読み出す関数（以下キャッシュリード関数）はキャッシュの動作をエミュレートする以下の動作を行う．

2.5.1 キャッシュヒット，ミスを判定するための処理

キャッシュリード関数はキャッシュヒット，キャッシュミスを判定する以下の処理を行う．

(1) キャッシュブロックアドレスを求める．
global memory 上のデータ配列はキャッシュラインサイズで分割されたブロックごとにキャッシュラインに転送される．このブロックのアドレスをキャッシュブロックアドレスと定義する．キャッシュリード関数は対象の global memory のデータ配列のインデックスと，キャッシュラインサイズからキャッシュブロックアドレスを求める．キャッシュブロックアドレスは global memory のデータ配列のインデックスをキャッシュラインサイズで割った商から求められる．

(2) 格納する対象のキャッシュラインを求める．
キャッシュリード関数は 1 で求めたキャッシュブロックアドレスとキャッシュライン数の剰余を計算することにより，global memory のデータを格納する対象のキャッシュラインを求める．

(3) キャッシュブロックアドレスとタグを比較し，キャッシュヒット/ミスの判定を行う．
キャッシュリード関数は対象のキャッシュラインのタグと 1 で求めたキャッシュブロックアドレスを比較することにより，キャッシュヒット/ミスを判定する．一致していればキャッシュヒット，一致しなければキャッシュミスと判定され，

(4) 対象のラインがリプレースされる．

図 2 はキャッシュの動作例を示す．図 2 では，global memory のデータ配列を 64 ワードの配列としている（図中の global_data）．またソフトウェアキャッシュ機構のキャッシュライン数を 4 本，キャッシュラインサイズを 8 ワードとしている．global_data[41] にアクセスする場合はインデックスの 41 をキャッシュラインサイズ 8 で割った商の 5 がブロックアドレスとなる．また global_data[40] 〜global_data[47] が同一ブロックに属する．

2.5.2 キャッシュミス時の動作

キャッシュミスと判定した場合，キャッシュリード関数は以下の処理を行う．

(1) Thread を用いてタグの値，キャッシュライン内のデータを更新する．
キャッシュリード関数は global memory のデータ配列から対象のキャッシュラインへのデータの転送を行う．また対象のキャッシュラインのタグに，以前に求めたキャッシュブロックアドレスを格納する．データ転送の処理時間を減らすため，キャッシュリード関数は複数の Thread を用いて global memory のデータ配列をキャッシュラインへ並列に転送する．1Thread が global memory のデータ配列の 1 ワー

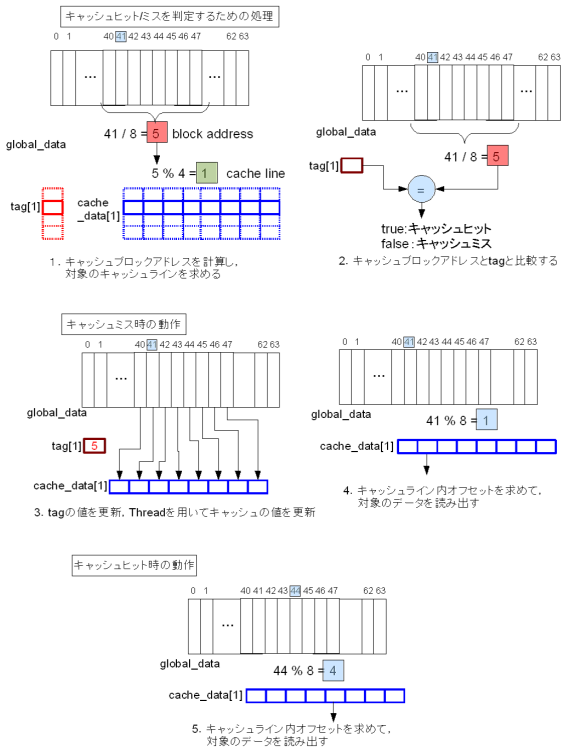


図 2 キャッシュの動作

ドをキャッシュへ転送することを担当し，キャッシュラインサイズと同数の Thread を用いて転送する．

(2) キャッシュラインから対象のデータを読み出す．
キャッシュリード関数は global memory のデータ配列のインデックスと対応したキャッシュライン内のデータを読み出す．そのため，キャッシュライン内のデータ配列の先頭からの距離を示すキャッシュライン内オフセットを求めて対象のデータを返す処理を行う．キャッシュライン内オフセットは global memory のインデックスと

(3) キャッシュラインサイズの剰余から求められる．

図 2 の例では tag[1] にキャッシュブロックアドレスの 5 を格納し，8Thread を用いて global_data[40]〜global_data[47] のデータを cache_data[1][0]〜cache_data[1][7] へ転送する様子を示している．global memory 上のデータ配列のインデックスは 41 であるため，キャッシュライン内オフセットは 1 と決まり，Thread は cache_data[1][1] をリードする．

2.5.3 キャッシュヒット時の動作

キャッシュヒットと判定した場合，キャッシュリード関数は以下の処理を行う．

(1) キャッシュラインから対象のデータを読み出す
キャッシュリード関数は global memory のデータ配列のインデックスと対応したキャッシュライン内のデータを読み出す．そのために，キャッシュライン内オフセットを求めてデータを返す処理を行う．

図 2 は，Thread が global_data[41] をアクセスした後，続いて global_data[44] をアクセスした場合の様子を示している．この場合，global memory 上のデータ配列 global_data[44] から求められるキャッシュブロックアドレスは対象のラインのタグと等しいのでキャッシュヒットと判定される．global memory 上のデータ配列のインデックスは 44 であるため，キャッシュ内オフセットは 4 と決まり，Thread は cache_data[1][4] をリードする．

3. 提案機構の評価

本節では提案，実装を行ったソフトウェアキャッシュ機構を評価する．ソフトウェアキャッシュ機構を評価するアプリケーションとして行列積と離散フーリエ変換を用いた．

3.1 対象アプリケーション 1：行列積

行列積は，演算に用いるデータ量 N に対して $O(N^3)$ という計算量が多い問題であるが，その一方で並列処理などによる高速化を行いやすい問題でもある．またキャッシュによる高速化も期待できる．行列積 $C = A \times B$ では，行列 A は特に多数回アクセスされることから，キャッシュに乗せると高速化が期待できる．

行列は全て正方行列とし，行列一辺の長さを問題サイズと定義する．

CUDA を用いた行列積計算のアルゴリズムを説明する．図 3 は問題サイズ 2048 の行列積の並列アルゴリズムの図である．行列積 $C = A \times B$ の C の行を Block 数でブロック分割し，その Block 内を 256 個の Thread を用いて並列計算を行う．

図 3 は Block 数を 64 とした場合のアルゴリズムである．問題サイズは 2048 のため，各 Block は行列 C の 32×2048 の計算を担当する．Block 内の計算は以下のアルゴリズムで計算を行う．

各 Block は割り当てられた分割箇所の 1 行目を計算する．キャッシュに乗せて効果があると考えられる行列 A の 1 行をソフトウェアキャッシュ機構のキャッシュリード関数を用いてキャッシュに乗せてデータを取り出す．Block 内 Thread は 2 つのベクトルの内積を求める．Block 辺りの Thread 数を 256 としたので 1 列目を Thread0，2 列目を Thread1，…，256 列目を Thread255 が担当する 1 行 ×256 列の計算を一度に行う．次のブロックに移り，257 列目を Thread0，258 列目を Thread1，…，512 列目を Thread255 が担当する．この計算を繰り返すことにより行列 C の 1 行分が計算される．1 行目の計算が終了すると，2 行目の計算に移り，同様の計算を行う．割り当てられた分割箇所の最終行まで同じ計算を繰り返す．

3.2 対象アプリケーション 2：離散フーリエ変換

離散フーリエ変換（Discrete Fourier Transform，以下 DFT）は，連続周期信号をサンプリングし，周波数に変換する計算を行う．N 個の離散データ x_n（$n = 0, 1, \cdots N-1$）の DFT は，

$$X(k) = \sum_{n=0}^{N-1} W^{kn} x(n) \quad (1)$$

$$W = e^{-j\frac{2\pi}{N}} \quad (2)$$

で定義される．

CUDA を用いた DFT 計算のアルゴリズムを説明する．

Vol.2009-ARC-186 No.9
Vol.2009-HPC-123 No.9
2009/12/1

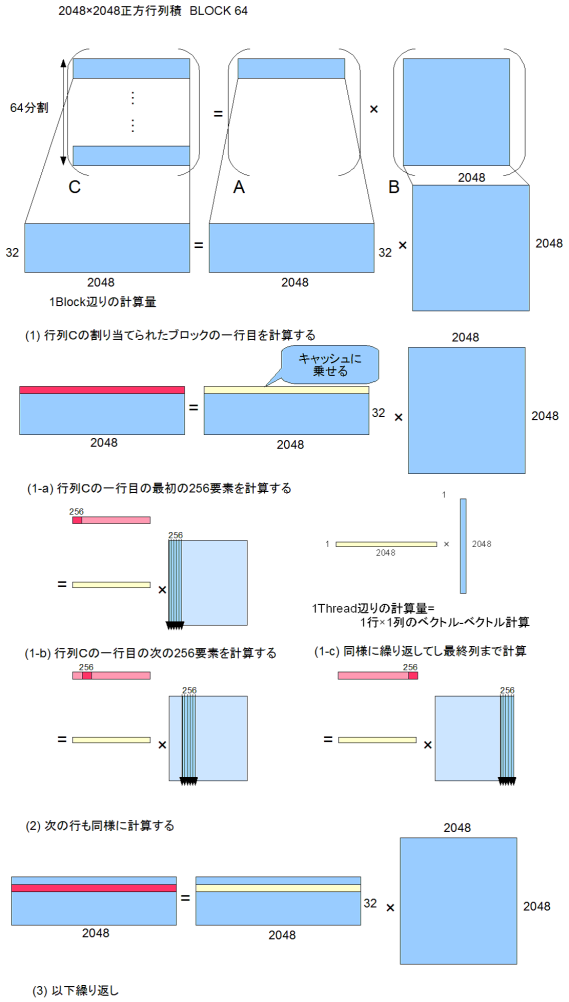


図 3 CUDA による行列積の実装

図 4 は離散データ数 N を 65535 とした場合の並列アルゴリズムの図である．求める周波数配列 X を複数の Block に分ける．Block 内の X の各要素を Thread を用いて並列に計算する．Block 内 Thread 数を 256 とした．図 4 では Block 数を 128 としている．離散データ数 N は 65536 であるため，各 Block あたり担当する要素数は 512 要素となる．各 Block は割り当てられた要素のうち，最初の 256 要素を Thread を用いて並列に計算を行う．ベクトル部 x をソフトウェアキャッシュ機構のキャッシュリード関数を用いてキャッシュに乗せ，データを取り出す．各 Thread は 2 つのベクトルの内積を求める（図 4（1））．割り当てられた Block の最初の 256 要素を計算したのち，残りの 256 個の要素を計算する（図 4（2））．

3.3 評 価 環 境

表 1 の評価環境を用いて，ソフトウェアキャッシュ機構の評価を行った．2 種類の GPU を用いてソフトウェアキャッシュ機構の有用性を確かめる．各評価環境に搭載されている GPU の性能を表 2 にまとめた．評価環境 1 の GPU，GeForce GTX 280（以下 GTX280）は GeForce 200 Series の GPU である．また評価環境 2 の GPU，GeForce 8600GTS 512MB（以下 8600GTS）は GeForce 8 Series

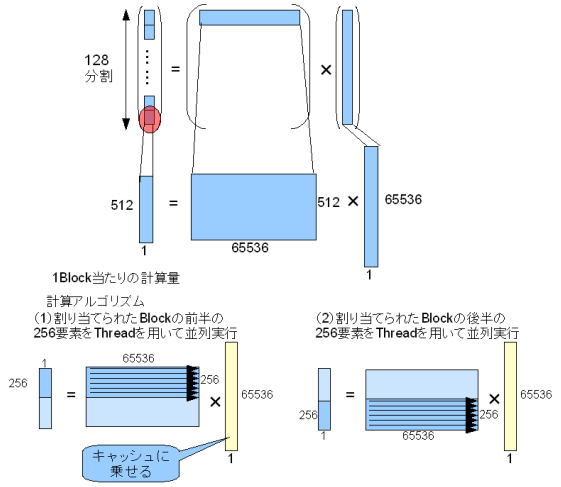


図 4 CUDA による DFT 計算の実装

の GPU である．GTX280 の SP 数は 240 個であり，一方 8600GTS は 32 個と GTX280 は 8600GTS より演算性能が高い GPU である．

評価はアプリケーションの実行時間を測定し行う．今回の評価では行列積（DFT）を行う Kernel が Host から呼ばれ，計算が終了するまでを実行時間の測定範囲としており，Host と Device 間の転送時間は含めないものとする．

表 1 評価環境

	評価環境 1	評価環境 2
GPU	GTX 280	8600GTS
CPU	Xeon E5345 2.33GHz	Pentium 4 3.00GHz
Memory	4.0GB	1.0GB
OS	CentOS 5.0	CentOS 5.0
Software	CUDA2.0	CUDA 2.0

表 2 評価環境の GPU のスペック

	GTX 280	8600GTS
MP 数	30	4
SP 数	240(30 × 8)	32(4 × 8)
ビデオメモリ	1GB GDDR3	512MB GDDR3
shared memory/MP	16KB	16KB
register/MP	16384	8192
SP クロック	1.296GHz	1.45GHz
メモリクロック	2.214GHz	2.0GHz
メモリバンド幅	512bit	128bit

3.4 キャッシュ構成の適正値の調査

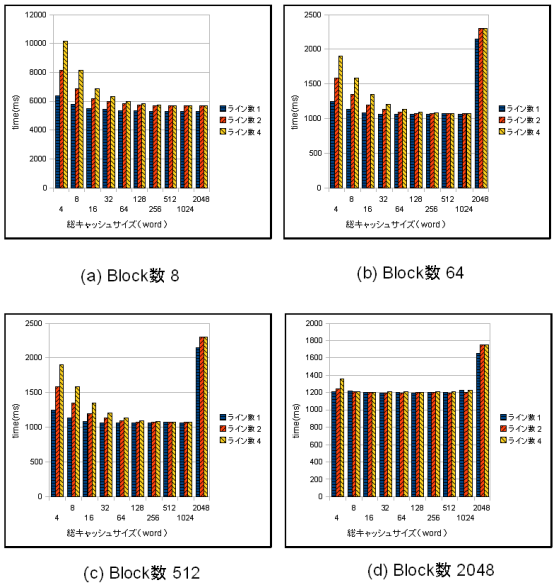
3.4.1 キャッシュライン数の適正値の調査

提案するソフトウェアキャッシュ機構はキャッシュラインサイズだけではなく，キャッシュラインの数も自由に指定することが出来るようにした．そこで本節では最適なキャッシュライン数を調べることとし，具体的にはライン数 1，2，4 本の 3 種類について実行時間を測定する．またアプリケーションは行列積を対象とし，Block 数を変えてライン数の影響を見る．評価環境 1 を用いて評価を行った．

結果を図 5 に示す．それぞれの Block 数におけるグラフは全ラインのキャッシュサイズの合計値を横軸，実行時間を縦軸にとり，キャッシュライン数を 1 本とした場合の実行時間とキャッシュライン数を 2 本とした場合の実行

時間，そしてキャッシュライン数を 4 本とした場合の実行時間の 3 つのグラフを左から重ねたグラフとなっている．Block 数は 8，64，512，2048 としている．

これらのグラフから読み取れることは，キャッシュライン数は 1 本の場合が最善であるということである．図 5 (a) の Block 数 8 のグラフを見ると，キャッシュライン数を 1 本とした場合の実行時間より 2，4 本とした場合の実行時間の方が長くなっている．異なる Block 数の図 5 (b)，(c)，(d) を見ても，それぞれのキャッシュライン数の実行時間はほぼ等しいか，ライン数を増やすと実行時間が長くなっている．これはキャッシュのリプレース時に起こるキャッシュラインを選択する際の条件分岐のオーバーヘッドの影響が出ているからと考えられる．



(a) Block数 8　(b) Block数 64

(c) Block数 512　(d) Block数 2048

図 5　キャッシュライン数を変化させた場合の実行時間（対象アプリケーション：行列積，評価環境 1）

3.4.2　キャッシュラインサイズの適正値の調査

本節ではソフトウェアキャッシュ機構の最適なキャッシュラインサイズを調べる．3.4.1 節の議論からキャッシュライン数を 1 本とする．キャッシュライン数を 1 本とした場合のソフトウェアキャッシュのキャッシュラインサイズは $2^0 = 1$ ワードから $2^{11} = 2048$ ワードまでの値を指定することが出来る．

2 種類の評価環境を用いて，それぞれの環境におけるキャッシュラインサイズの最適値を調べた．

3.4.3　行列積を用いたキャッシュラインサイズの適正値の調査

評価環境 1（GeForce GTX 280）での調査

GTX280 を用いてソフトウェアキャッシュ機構の評価を行う．問題サイズを 2048 とした場合の行列積の実行時間を測定した．

行列積におけるソフトウェアキャッシュ機構の適切なキャッシュラインサイズを調べるため，キャッシュラインサイズを変更し，その実行時間を調べる．結果を図 6 に示す．このグラフはキャッシュラインサイズを横軸，実行時間を縦軸としている．Block 数を 2^0〜2^{11} まで変更し，実行時間を測定しそれぞれの結果をグラフにまとめた．図 6 において Block 数が 16 以上のグラフを拡大したものが図 6 下のグラフである．

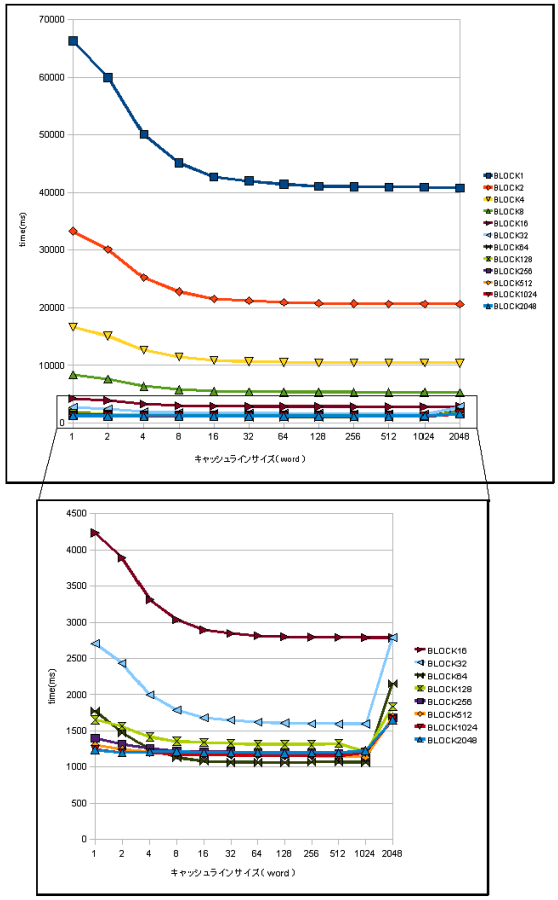


図 6　ソフトウェアキャッシュ機構を用いた場合の実行時間（対象アプリケーション：行列積，評価環境 1）

各々の Block のグラフを見ると，キャッシュラインサイズ 1〜64 ワードではキャッシュラインサイズが大きくなるごとに実行時間が短くなっており，キャッシュの効果が現れていることが分かる．しかしキャッシュラインサイズが 128 ワード以上となる場合には，キャッシュの効果が頭打ちになっている．

ここで注目したいのは，キャッシュラインサイズを 2048 ワードとした場合の実行時間である．Block 数が 1〜16 の場合には，キャッシュラインサイズが 1024 ワードの場合の実行時間とほぼ変わらないが，Block 数 32 以上の場合にはキャッシュラインサイズを 2048 ワードとした場合に急激に実行時間が長くなっている．

キャッシュラインサイズ 2048 ワードでは，問題サイズ 2048 の場合には 1 回のキャッシュミスで行列 A の 1 行のデータが全て乗り，キャッシュミスの回数は確実に減っており，実行時間が伸びることは不可解なことと考えられる．

この理由は MP に同時に割り当てられる Block 数に起因する．Block 数が MP 数より多い場合には，1 つの MP に複数個の Block が割り当てられ，タイムスライスごとに処理する Block を切り替える．同時に割り当てることが出来る Block 数は MP 内の資源によって制限される．そ

の資源の一つとして shared memory が挙げられる．

キャッシュラインサイズを 2048 ワードとした場合の 1Block 辺りが確保しなければならない shared memory の容量は tag の容量を考慮すると，4（Byte）×2048＋4（Byte）＝8196（Byte）となる．1Block 当たり用いることが出来る shared memory の容量は 16KB であることから，キャッシュラインサイズが 2048 ワードの場合には MP が同時に担当できる Block 数が 1 個に限定されてしまう．GTX280 は MP 数が 30 であるため，Block 数が 32 以上の場合にこの影響が出てしまう．

一方 Block 数が 16 以下の場合には，各 Block を一度に各 MP に割り当てることが可能であり，Block 数 32 以上の場合に起こったオーバーヘッドは現れないが，全ての MP を用いることが出来ないのでトータルの性能は落ちる．

以上のことから，評価環境 1 ではキャッシュラインサイズが 128 ワード〜1024 ワードとした場合に高い実行性能が得られていることが分かる．

評価環境 2（GeForce 8600GTS 512MB）での調査

8600GTS を用いてソフトウェアキャッシュ機構の評価を行う．問題サイズを 2048 とした場合の行列積の実行時間を測定した．

行列積におけるソフトウェアキャッシュ機構の適切なキャッシュラインサイズを調べるため，評価環境 1 の場合と同様にキャッシュラインサイズを変更し，その実行時間を調べる．Block 数を 2^0〜2^{11} まで変更し，実行時間を測定しそれぞれの結果をグラフにまとめた．

結果を図 7 に示す．このグラフはキャッシュラインサイズを横軸，実行時間を縦軸としている．図 7 において Block 数 4 以上のグラフを拡大したものが図 7 下のグラフである．

各々のグラフを見ると，キャッシュラインサイズが 1 ワード〜16 ワードの間ではキャッシュの効果が現れている．しかしキャッシュラインサイズが 32 ワードを越えると，キャッシュの効果が頭打ちになっている．

キャッシュラインサイズ 256 ワード以降，Block 数 1〜128 のグラフは実行時間の変化が現れないことに対し，Block 数 256〜2048 のグラフはキャッシュラインサイズを増やすと実行時間が長くなる．これは MP 数に対する Block 数が非常に多いことに起因していると考えられる．

以上のことから，評価環境 2 ではキャッシュラインサイズが 32 ワード〜128 ワードとした場合に高い実行性能が得られていることが分かる．

3.4.4 DFT を用いたキャッシュラインサイズの適正値の調査

評価環境 1（GeForce GTX 280）での調査

GTX280 を用いて，ソフトウェアキャッシュ機構の評価を行う．

ソフトウェアキャッシュのキャッシュラインサイズを変更し，最適なラインサイズを求める．Block 数を 2^0〜2^8 と変更し，実行時間を測定しそれぞれの結果のグラフにまとめた．

結果を図 8 に示す．このグラフはキャッシュラインサイズを横軸，実行時間を縦軸としている．図 8 において

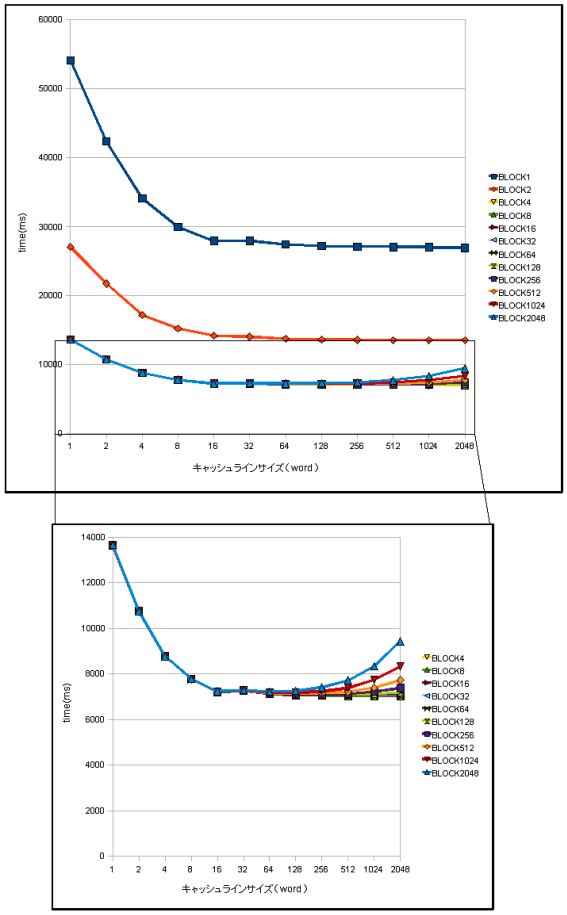


図 7 ソフトウェアキャッシュ機構を用いた場合の実行時間（対象アプリケーション：行列積，評価環境 2）

Block 数 16 以上のグラフを拡大したものが図 8 下のグラフである．

各々の Block のグラフを見ると，キャッシュラインサイズが 1 ワード〜16 ワード間ではサイズが大きくなるごとに実行時間が短くなっており，キャッシュの効果が現れていることが分かる．しかしキャッシュラインサイズが 16 ワードより大きい場合には，キャッシュの効果が頭打ちになっている．

キャッシュラインサイズを 2048 ワードとした場合の実行時間に急激な変化が見られる．Block 数 1〜16 の場合はキャッシュラインサイズを 1024 ワードとした場合の実行時間とほぼ変わらないが，Block 数 32 以降ではキャッシュラインサイズを 2048 ワードとした場合の実行時間が長くなっている．これは行列積の場合と同様，1MP あたりに同時に割り当てることが出来る Block 数が shared memory の容量の制限より 1 個に制限されるためと考えられる．

以上のことから，評価環境 1 ではキャッシュラインサイズを 32 ワード〜1024 ワードとした場合に高い実行性能が得られていることが分かる．

評価環境 2（GeForce 8600GTS 512MB）での調査

8600GTS を用いてソフトウェアキャッシュ機構の評価を行う．

ソフトウェアキャッシュのキャッシュラインサイズを変

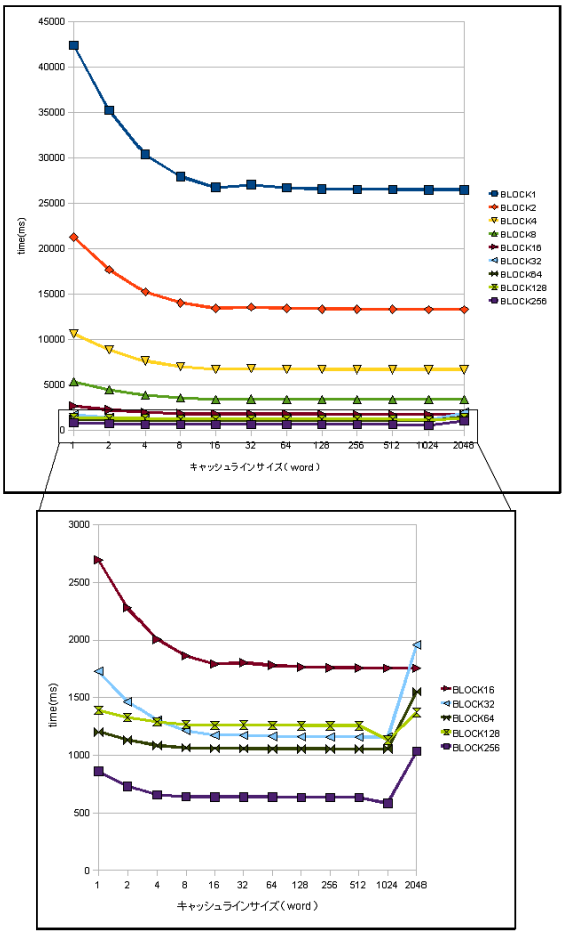


図 8 ソフトウェアキャッシュ機構を用いた場合の実行時間（対象アプリケーション：DFT，評価環境 1）

更し，最適なラインサイズを求める．Block 数を 2^0〜2^8 と変更し，実行時間を測定しそれぞれの結果をグラフにまとめた．

結果を図 9 に示す．このグラフはキャッシュラインサイズを横軸，実行時間を縦軸としている．図 9 において Block 数 4 以上のグラフを拡大したものが図 9 下のグラフである．

各々のグラフを見るとキャッシュラインサイズが 1 ワード〜16 ワードではサイズを増やすごとに実行時間が短くなっており，キャッシュの効果が明確に現れていると言える．しかしキャッシュラインサイズが 32 ワードより大きい場合には，キャッシュの効果が頭打ちになっている．

キャッシュラインサイズを 2048 ワードとした場合，Block 数 8 以上のグラフが急激に実行時間が長くなっている．キャッシュラインサイズを 2048 ワードにすることにより，1MP あたりの Block 数が 1 個に制限されてしまうからである．そのため，キャッシュラインサイズ 2048 の場合には 8600GTS の MP 数 4 より大きい Block 数を割り当ててもそれ以上の高速化を図ることが出来ない．

以上のことから，評価環境 2 ではキャッシュラインサイズを 32 ワード〜1024 ワードとした場合に，高い実行性能が得られていることが分かる．

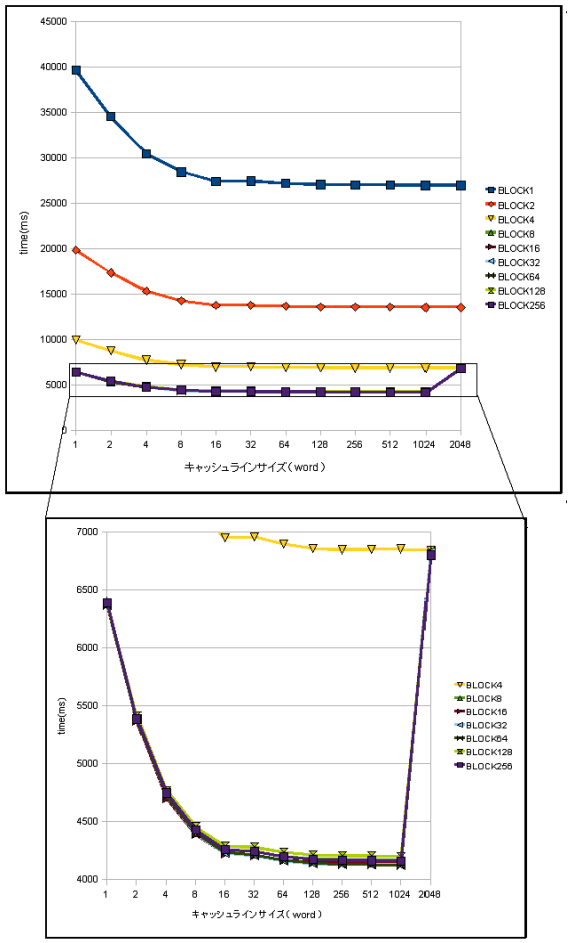


図 9 ソフトウェアキャッシュ機構を用いた場合の実行時間（対象アプリケーション：DFT，評価環境 2）

3.5 性能評価

本節では 3.4 節で求めたキャッシュの適正値を用いたソフトウェアキャッシュ機構を，global meomry だけを用いる場合および shared memory を明示的に用いた場合と比較し，ソフトウェアキャッシュ機構の有用性を確認する．

3.5.1 行列積を用いた評価

評価環境 1（GeForce GTX 280）での評価

3.4 節の結果適切だと分かったキャッシュラインサイズである 128 ワードの場合のソフトウェアキャッシュ機構を用いた場合と，shared memory を用いずに global memory だけを用いる場合，そして明示的に shared memory に乗せて計算を行う場合の実行時間を比較する．shared memory は 256 要素の配列を用いてブロック化を行い，計算を行った．

結果を図 10 に示す．このグラフの横軸は Block 数を示し，また縦軸は実行時間を示している．図 10 の各棒グラフは，

- shared memory を用いず global memory だけを用いた場合の実行時間（図 10「global」）
- shared memory を明示的に用いた場合の実行時間（図 10「shared」）
- キャッシュラインサイズを 128 ワードとしたソフト

ウェアキャッシュ機構を用いた場合の実行時間（図 10「software cache」）
を示している．

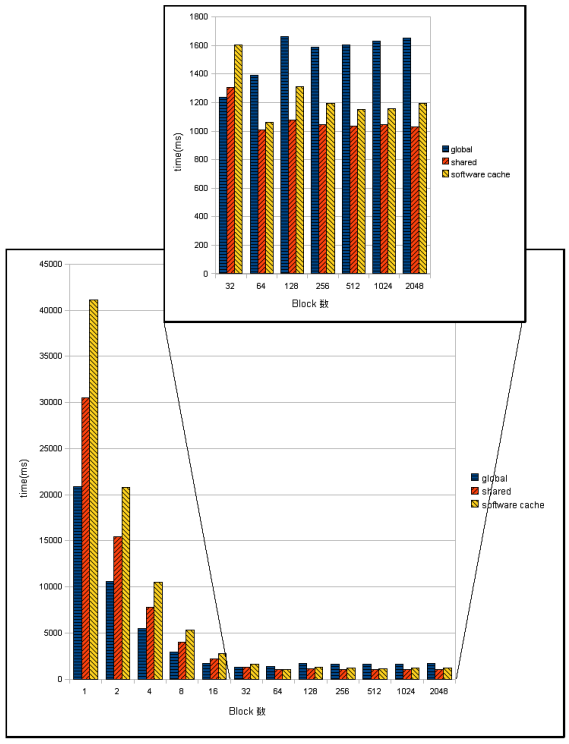


図 10 global memory，shared memory，ソフトウェアキャッシュ機構の正方行列積の実行時間（対象アプリケーション：行列積，評価環境 1）

Block 数が 32 までの場合は global memory だけを用いた場合が最も高速であることを示している．これは GTX280 では global memory のメモリクロックとメモリバンド幅が大きく，shared memory の転送量に密接に関係している SP クロックがそれほど高速ではないことから，Block 数が少ない場合には shared memory をデータ転送に用いる場合のオーバーヘッドが大きいためであると考えられる．しかし，Block 数 64 以降でその関係は逆転しており，shared memory を明示的に用いた場合が最も高速であり，続いてソフトウェアキャッシュ機構，global memory と続いている．以上から，GTX280 では Block 数が多い場合に，ソフトウェアキャッシュ機構が有効であることが分かる．

評価環境 2（GeForce 8600GTS 512MB）での評価

3.4 節の結果適切だと分かったキャッシュラインサイズである 128 ワードの場合のソフトウェアキャッシュ機構を用いた場合，shared memory を用いず global memory だけを用いる場合，そして明示的に shared memory に乗せて計算を行う場合の実行時間を比較する．評価環境 1 と同様，shared memory は 256 要素の配列を用いてブロック化を行い，計算を行う．

結果を図 11 に示す．このグラフの横軸は Block 数を示し，また縦軸は実行時間を示している．図 11 の各棒グラフは，

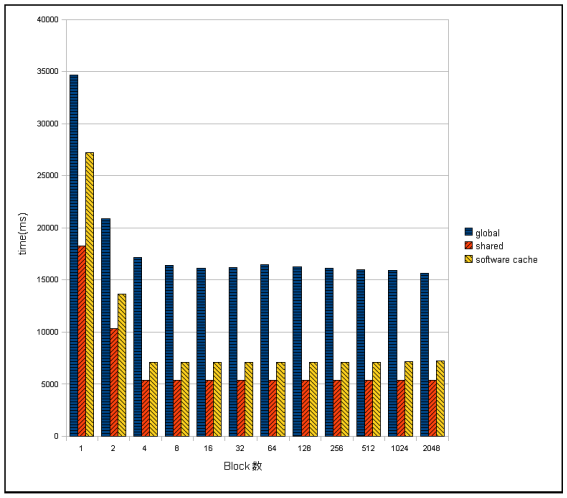


図 11 global memory，shared memory，ソフトウェアキャッシュ機構の正方行列積の実行時間（対象アプリケーション：行列積，評価環境 2）

- shared memory を用いず global memory だけを用いた場合の実行時間（図 11「global」）
- shared memory を明示的に用いた場合の実行時間（図 11「shared」）
- キャッシュラインサイズを 128 ワードとしたソフトウェアキャッシュ機構を用いた場合の実行時間（図 11「software cache」）

を示している．

Block 数を一定数以上とした場合にソフトウェアキャッシュ機構の効果が見られた評価環境 1 と異なり，評価環境 2 では Block 数 1 から global memory だけを用いた場合より，ソフトウェアキャッシュ機構の効果が現れている．この理由はメモリクロックとメモリバンド幅が 8600GTS は GTX280 より小さく，また shared memory の転送量に関係している SP クロックが GTX280 のそれよりも大きいためであると考えられる．以上から，8600GTS ではソフトウェアキャッシュ機構が有効であることがわかる．

3.5.2 DFT を用いた評価

評価環境 1（GeForce GTX 280）での評価

3.4 節の結果適切だと分かったキャッシュラインサイズである 256 ワードの場合のソフトウェアキャッシュ機構を用いた場合と，shared memory を用いずに global memory だけを用いる場合，そして明示的に shared memory に乗せて計算を行う場合の実行時間を比較する．shared memory は 256 要素の配列を用いて，ブロック化を行い計算を行った．

結果を図 12 に示す．このグラフの横軸は Block 数を示し，また縦軸は実行時間を示している．図 12 の各棒グラフは，

- global memory だけを用いた場合の実行時間（図 12「global」）
- shared memory を明示的に用いた場合の実行時間（図 12「shared」）
- キャッシュラインサイズを 256 ワードとしたソフトウェアキャッシュ機構を用いた場合の実行時間（図 12

「software cache」）
を示している．

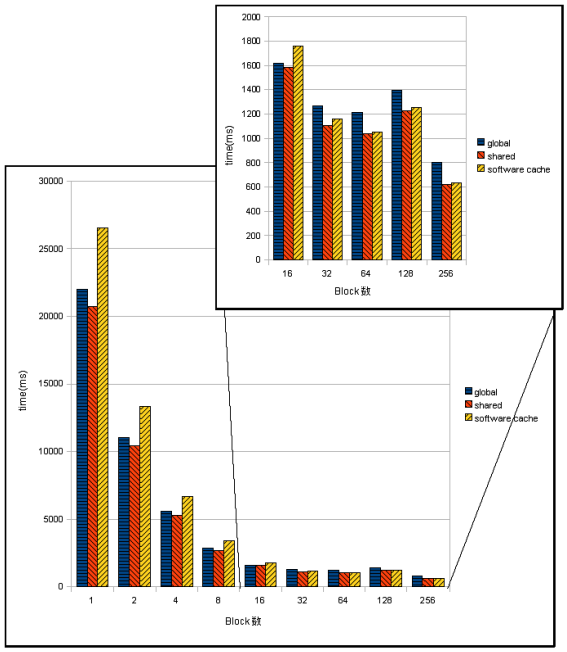


図 12 global memory，shared memory，ソフトウェアキャッシュ機構の DFT の実行時間（対象アプリケーション：DFT，評価環境 1）

Block 数が 16 までの場合は global memory だけを用いた場合より，ソフトウェアキャッシュ機構を用いた場合のほうが実行時間が長い．これは行列積を用いた評価で考察したように，メモリクロックと SP クロックの差が小さいことから，Block 数が少ない場合には shared memory へデータ転送するオーバーヘッドの影響が大きいためであると考えられる．一方，Block 数 32 以降ではその関係は逆転している．shared memory が最も高速であり，ソフトウェアキャッシュ機構がその次であり，global memory と続いている．以上から，GTX280 では Block 数が大きいとより機構を有効に利用出来ることがわかる．

評価環境 2（8600GTS）での評価

3.4 節の結果適切だと分かったキャッシュラインサイズである 256 ワードの場合のソフトウェアキャッシュ機構を用いた場合と，shared memory を用いずに global memory だけを用いる場合，そして明示的に shared memory に乗せて計算を行う場合の実行時間を比較する．shared memory は 256 要素の配列を用いて，ブロック化を行い計算を行う．

結果を図 13 に示す．このグラフの横軸は Block 数を示し，また縦軸は実行時間を示している．図 13 の各棒グラフは，

- global memory だけを用いた場合の実行時間（図 13「global」）
- shared memory を明示的に用いた場合の実行時間（図 13「shared」）
- キャッシュラインサイズを 256 ワードとしたソフトウェアキャッシュ機構を用いた場合の実行時間（図 13「software cache」）

を示している．

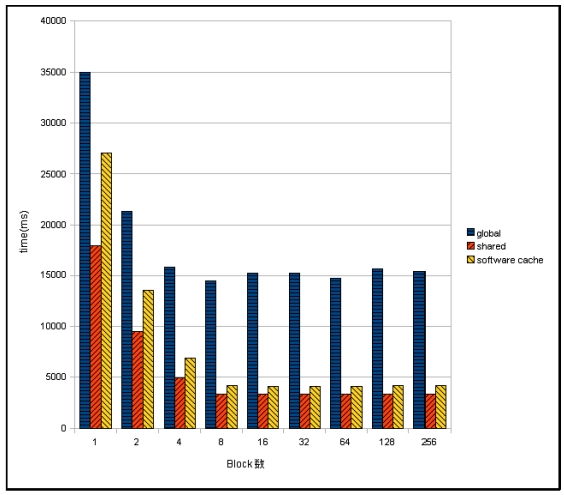


図 13 global memory，shared memory，ソフトウェアキャッシュ機構の正方行列積の実行時間（対象アプリケーション：DFT，評価環境 2）

Block 数を一定数以上とした場合にソフトウェアキャッシュ機構の効果が見られた評価環境 1 と異なり，評価環境 2 では Block 数 1 から global memory だけを用いた場合より，ソフトウェアキャッシュ機構の効果が現れている．この理由はメモリクロックとメモリバンド幅が 8600GTS は GTX280 より小さく，また shared memory の転送量に関係している SP クロックが GTX280 のそれよりも大きいためであると考えられる．図 13 より，ソフトウェアキャッシュ機構は有効であることがわかる．

3.6 ま と め

本節では提案機構の有効性の検証のため，キャッシュの効果が期待されるアプリケーションである行列積，DFT に対してソフトウェアキャッシュ機構を適用した．実験環境はハイエンド GPU を搭載した計算機とミドルレンジ GPU を搭載した計算機の 2 種類の計算機を用いた．

まず，ソフトウェアキャッシュ機構の最適な構成を調べた．最適な構成はキャッシュライン数を 1 本，キャッシュラインサイズを 128 ワード〜256 ワードとした場合であることが分かった．

また，キャッシュの容量を大きくすることによる速度低下が見られた．これは MP に同時に割り当てられる Block 数がキャッシュの容量を大きくすることにより制限されてしまったからであると考えられる．

次に，global memory だけを用いた場合，明示的に shared memory を用いた場合と適切な構成のソフトウェアキャッシュ機構を用いた場合の実行時間を比較し，ソフトウェアキャッシュ機構の有用性を確認した．2 つの実験環境において global memory だけを用いた場合より実行時間が短くなっており，ソフトウェアキャッシュ機構による速度向上が図れていることが分かった．shared memory を明示的に用いた場合より実行時間が長くなっているの

Vol.2009-ARC-186 No.9
Vol.2009-HPC-123 No.9
2009/12/1

は，キャッシュミスによるストールやキャッシュアクセスのオーバーヘッドが発生しているからと考えられる．

また，SP クロックと global memory のメモリクロックの差が大きい GPU のほうがソフトウェアキャッシュ機構の効果が大きかった．これはメモリクロックが global memory からデータを呼び出す際の性能に起因し，SP クロックが shared memory からデータをリードする際の性能に起因するからであると考えられる．

4. 関連研究

共有メモリを持たない並列分散環境において，仮想的な共有アドレス空間を提供するソフトウェア分散共有メモリ（以下 S-DSM）の研究が行われている[1)2)]．S-DSM[3)] では，アクセスレイテンシの削減のため遠隔ノードのデータを自ノードのメモリにキャッシュする必要があり、コンパイル時にデータを自ノードにキャッシュするプリフェッチを行う．本研究で提案するソフトウェアキャッシュ機構はプログラム実行中にデータにアクセスする時点で shared memory 中のデータの有無の判定を行い，必要に応じて global memory から shared memory にデータを転送する点で異なる．

ストリーミング言語を用いて GPU のメモリアクセス性能を引き出すことが可能なコードへ変換する手法が提案されている[4)5)]．本研究で提案したソフトウェアキャッシュ機構は既存の GPGPU プログラミング環境 CUDA 上で用いることとしているが，文献[4)5)] の研究は新たなストリーミング言語を提案している点で本研究と異なる．

GPU と並ぶアクセラレータとして利用されている Cell Broadband Enigne（以下 Cell プロセッサ）向けのソフトウェアキャッシュ機構が提案されている[6)7)]．SPE 上のスクラッチパッドメモリ LS は CUDA における shared memory に対応し，On-chip memory を用いたソフトウェアキャッシュ機構を作成した点において本研究と文献[6)7)] の研究は共通している．しかし，文献[7)] で提案されているソフトウェアキャッシュ機構は，他プロセッサへのデータ供給を支援するキャッシュコアを提案している．本研究ではこのような他プロセッサへのデータ供給を支援するソフトウェアキャッシュ機構を想定していない．

5. 結　　論

本研究では，高速共有メモリの shared memory を global memory のキャッシュとして用いるソフトウェアキャッシュ機構を提案した．このソフトウェアキャッシュ機構で提供される関数を用いることにより，プログラマは容量制限のある shared memory を用いるためにブロック化を行い，shared memory のデータのリプレースを行う必要が無くなる．

キャッシュの効果が期待される行列積，DFT 計算に対し，ソフトウェアキャッシュ機構を適用し評価を行った．行列積，DFT 計算におけるキャッシュラインサイズの最適値を求め，その最適値におけるソフトウェアキャッシュ機構を適用した場合と，shared memory を明示的に用いた場合，shared memory を用いず大容量メモリの global memory 上にデータが存在する場合の 3 通りについて 2 種類の実験環境で動作させ，実行時間を比較した．

キャッシュラインサイズの最適値を求める過程で，キャッシュラインサイズを大きくすると実行性能が著しく落ちる現象が確認できた．これは CUDA のスケジューリング方法に起因する．プログラマが指定する並列処理単位の Block は，GPU の MP に割り当てられ処理される．同時に割り当てることが可能な Block 数は shared memory の量に関係している．用いる shared memory の量が多すぎると，割り当てられる Block 数が減り実行性能が低下する．適切なキャッシュラインサイズの場合のソフトウェアキャッシュ機構を用いると，global memory 上にデータが存在する場合と比べて実行時間が低下し，速度向上を図ることが出来た．

ソフトウェアキャッシュ機構を用いる場合，プログラマはキャッシュラインサイズを指定することが出来るが，今回の実験結果で得られたキャッシュラインサイズをデフォルト値として設定することにより，速度向上を図ることが可能となる．

参 考 文 献

1) Pete Keleher, Alan L. Cox, Hya Dwarkadas, Willy Zwaenepoel:TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proc. of the Winter 1994 USENIX Conference, pp.115-131, 1994.
2) 田邊 浩志，本多 弘樹，弓場 敏嗣：ソフトウェア分散共有メモリを用いたマクロデータフロー処理，情報処理学会論文誌 Vol.46 No.SIG_4(ACS_9)，pp. 56-68, 2005.
3) 丹羽純平：コンパイラが支援するソフトウェア DSM におけるプリフェッチ機構，情報処理学会研究報告 2004-ARC-156，pp.7-12，2004.
4) 滝沢寛之，白取寛貴，佐藤功人，小林宏明：SPRAT：実行時自動チューニング機能を備えるストリーム処理記述用言語，先進的計算基盤システムシンポジウム（SACSIS2008），pp.139-148，2008.
5) 佐藤功人，滝沢寛之，小林弘明：GPU を効率的に利用するための言語拡張と自動最適化手法，情報処理学会研究報告 2008-HPC-116，pp.199-204，2008.
6) 佐藤 芳紀，神酒 勤：Cell Broadband Engine への SPE ソフトウェアデータキャッシュの実装，情報処理学会研究報告 2008-HPC-110，pp.13-18，2007.
7) 森谷 章，藤枝 直輝，佐藤 真平，吉瀬 謙二：メニーコアプロセッサに向けたデータ供給を支援する多機能キャッシュコア，先進的計算基盤システムシンポジウム SACSIS2008，pp.421-430，2008.