オンチップ並列処理を利用した密行列 LU 分解の実装

里城 晴紀　 小長谷 明彦
東京工業大学 情報理工学研究科 計算工学専攻

本稿では Cell Speed Challenge 2008 規定課題である連立一次方程式を解くために開発した left-looking 法による 32×32 のブロック LU 分解について報告する. 規定課題では, 与えられる係数行列のサイズは 32 の倍数という制約が課せられていた. 32×32 のブロック LU 分解は計算量に対してデータ転送量の比率が大きい. このため, 高い演算性能を持つ Cell BE においては, 頻繁なメモリアクセスを必要とする right-looking 法はメモリアクセスが性能ボトルネックとなる. 一方,left-looking 法はデータの時間的局所性が高く, ローカルストア内にキャッシュを用意することで, メモリアクセス回数を大幅に減少させることが期待できる.Left-looking 法において, 参照の局所性を活かしたタスクスケジューリング法を採用することで最大実行性能 137.8Gflops(理論性能の 76.9 %) を達成した.

1 はじめに

本稿では, Cell Speed Challenge 2008 規定課題にて開発した連立一次方程式求解プログラムに用いた最適化技法について報告する. 規定課題のために作成したプログラムは, LU 分解, 前進消去および後退消去のモジュールからなる. 各モジュールの最適化においては, 自由課題部門で報告した PDS 最適化サイクル [1] を適用し, 常にメモリアクセスと計算量のバランスを確認しながら最適化した [2]. 前進消去, 後退消去については SIMD 化とループアンローリング, ダブルバッファリングを施した. これらの最適化技法については [3, 4] が詳しい. LU 分解においては,32×32 のブロック LU 分解を採用したため, これまで報告されていた 64×64 のブロック LU 分解の実装 [5, 6] では生じなかったメモリアクセスボトルネックを解決することが性能向上のポイントとなった.

2 性能向上の課題

Cell BE は非常に高い浮動小数点数演算性能を持つため, メモリアクセスがボトルネックとなりやすい. ブロック LU 分解で最も計算量の多い演算は更新行列 $\mathbf{A}$ から $\mathbf{L}$ 成分と $\mathbf{U}$ 成分の積を引いて更新行列 $\mathbf{A}$ に戻す行列積差（$\mathbf{A} \leftarrow \mathbf{A} - \mathbf{LU}$）であり（図 1）, 各 SPE の行列積差演算の逐次実行性能の差がそのまま Cell BE 全体での性能差となる. 32×32 のブロック分割では, 計算に必要な 4 つのブロック ($\mathbf{A}$,$\mathbf{L}$,$\mathbf{U}$, および更新後の $\mathbf{A}$）のデータ転送時間は計算時間の 2 倍であり, 通常のタスクスケジューリング法では, ダブルバッファリングを用いても隠蔽不可能なメモリアクセスボトルネックが生じる. この問題を解決するためには, メモリアクセス回数そのものを減少させる方法の開発が必要である.

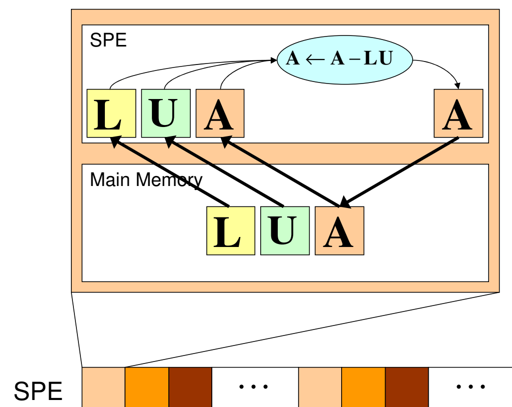

図 1: タスクの内容とタスクの待ち行列. 同色のタスクは同じ更新行列 $\mathbf{A}_{ij}$ を読み込む.

3 設計思想

メモリアクセス回数を減少させることが性能向上に必要である. そこで, 更新領域の時間的局所性が高い left-looking 法を採用し, キャッシングを行うことでメモリアクセス回数を減少させる. キャッシングの効果を確実に得るためのタスクの割り当て方も工夫する. 本節ではこれらを実現するための left-looking 法, キャッシング, ロードバランシングについて述べる.

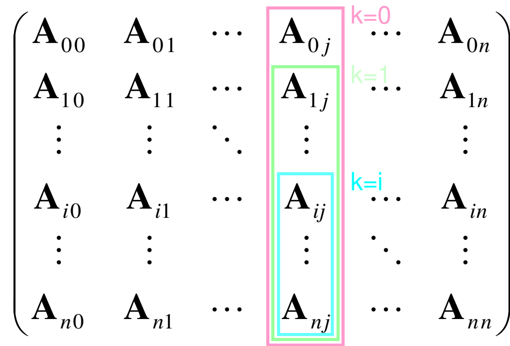

図 2: Left-looking 法の更新領域.1 列ずつ集中して計算を終わらせる. データの依存関係により SPE 間通信が増加するが, 時間的局所性が高い.

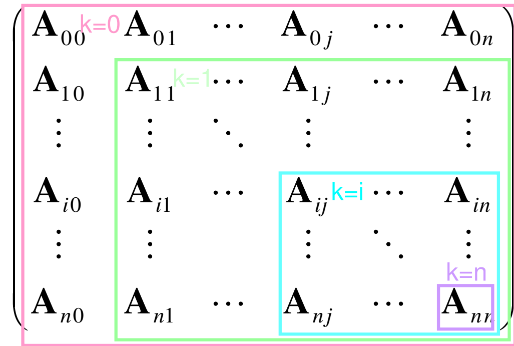

図 3: Right-looking 法の更新領域. データの依存関係のない広い範囲を計算する. データの依存関係はないが, 時間的局所性もない.

3.1 Left-looking 法

Left-looking 法は更新領域の時間的局所性が高く, データを一時的に保持することができれば, メモリアクセス回数を大幅に減少させることが期待できる. 一方, 最大並列度が確保できる right-looking 法は広い範囲の更新を同時に行うため, メモリアクセスがボトルネックとなり性能が向上しない.

図 2,3 に left-looking 法と right-looking 法のそれぞれの更新領域を示す.Right-looking 法は更新領域が広く並列度が高い. しかし, データの依存関係が少ないために通信回数は $O(n)$ 回で済むが, 一度使った更新領域を保持しても次に同じものが使われるまでには $O(n^2)$ の時間がかかる. 一方,left-looking 法は更新領域が 1 ブロック幅に限定されているため同じ更新領域が使われるまでの時間は $O(n)$ となる. よって,right-looking 法よりも left-looking 法を採用したほうがデータキャッシングに必要なローカルストアの容量を少なくできる. ローカルストアの容量は十分にあるので, データキャッシングによりメモリアクセスの大幅な減少が期待できる. ただし,left-looking 法ではクリティカルとなるデータ更新を他の SPE にブロードキャストする必要があるため, 通信回数が $O(n^2)$ に増えるという問題を持つ. しかし,Cell BE においては,SPE 間通信は非常に速いため係数行列が十分大きい場合は性能律速要因とはならない.

3.2 キャッシング

Left-looking 法においては更新領域の時間的局所性が高いため, キャッシュのヒット率が高く, メモリアクセス回数を 3 分の 1 まで減らすことができる. キャッシュがヒットすると更新領域 **A** の読み書き 2 回分のアクセスは不要になる.**U** 行列成分は他の SPE から転送されるものを連続して利用できるため, タスク中のメモリアクセスは **L** 行列成分を取得するときのみになる. Right-looking 法では更新領域が広いためキャッシングするのは困難であり, 連続で利用できるデータは **L** 行列成分, または **U** 行列成分のどちらか一方しかない. そのため right-looking 法ではタスク中のメモリアクセス回数は 3 回必要となる.

現行の Cell BE のメモリ転送速度は 25.6GB/秒であるが,Right-looking 法ではタスク中にブロックの転送を 3 回行うので必要な転送速度は 33.6GB/秒となりメモリアクセスボトルネックが発生する. 一方, キャッシングを用いた left-looking 法ではタスク中のブロックの転送回数は 1 回であるから必要な転送速度は 11.2GB/秒になり, メモリアクセスボトルネックは生じない.

3.3 ロードバランシング

キャッシングの効果を得るため, また, 各 SPE の計算量をバランスさせるために, SPE の担当領域は 32 行ごとに巡回させる（図 4）. キャッシュの効果を得るためには SPE の保持するデータとそれを参照するタスクが一致しなければならないので, 担当領域を固定する必要がある. また, タスクの数は係数行列の右下ほど多くなるため, 計算量を均等に配分するために担当領域は巡回させる必要がある.32 $\times$ 32 のブロック分割で計算する場合は, 担当領域を 32 行ごとに巡回させて各 SPE に振り分ければよい.

$$
\begin{array}{l}
\text{SPE0} \\
\text{SPE1} \\
\\
\text{SPE(i\%7)} \\
\\
\text{SPE(n\%7)}
\end{array}
\begin{pmatrix}
\mathbf{A}_{00} & \mathbf{A}_{01} & \cdots & \mathbf{A}_{0j} & \cdots & \mathbf{A}_{0n} \\
\mathbf{A}_{10} & \mathbf{A}_{11} & \cdots & \mathbf{A}_{1j} & \cdots & \mathbf{A}_{1n} \\
\vdots & \vdots & \ddots & \vdots & & \vdots \\
\mathbf{A}_{i0} & \mathbf{A}_{i1} & \cdots & \mathbf{A}_{ij} & \cdots & \mathbf{A}_{in} \\
\vdots & \vdots & & \vdots & & \vdots \\
\mathbf{A}_{n0} & \mathbf{A}_{n1} & \cdots & \mathbf{A}_{nj} & \cdots & \mathbf{A}_{nn}
\end{pmatrix}
$$

図 4: SPE の担当領域. ここで $\mathbf{A}_{ij}$ は 32 × 32 の行列である. 担当領域が明確に分かれているので, それぞれの SPE の更新作業は他の SPE のキャッシュに影響を与えない.

4 実装方法

本節では left-looking 法による 32 × 32 のブロック LU 分解の実装についてを述べる.

4.1 ソフトウェアキャッシュ

各 SPE のローカルストア内に 20 ブロック（80KB）分のキャッシュ領域を用意し, 読み込んだ更新領域はこのキャッシュ領域に保持する.Left-looking 法では更新領域が 1 ブロック幅であるから, 7 基の SPE をあわせると係数行列のサイズが 4480 までならば更新領域をキャッシュに格納できる. データはダイレクトマップ方式で格納し, スラッシングを防ぐために置換は行わない. キャッシュに格納したデータは更新がすべて終わった後にメモリに書き戻す. このとき, 各 SPE の担当領域は重なっていないためキャッシュコヒーレンシを考える必要はない.

4.2 タスクスケジューリング

U 行列成分の更新を含むタスクがクリティカルパスとなるため, 各 SPE において最優先に実行する. 各 SPE は他に計算可能なタスクを割り当てられているので, ブロードキャストにかかる時間は隠蔽できる（図 5）.

4.3 行列積差演算の最適化

行列積差演算には SIMD 化, ループアンローリング, パイプライン化を施し, さらにループカウンタの計算をシャッフル演算で代用することで, 最内ループで理論性能と同じ性能を引き出した. 表 1 は最内ループの命令列の一部である. 演算パイプラインが 1 本しかないため,2 命令同時発行比率は $59/128 = 0.46\cdots$ となっている. ロード, ストア,

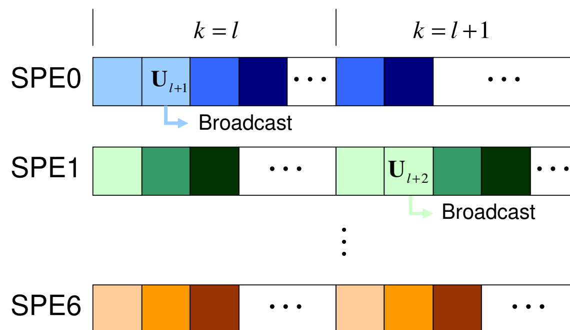

図 5: 各 SPE が実行するタスク.$k = l + 1$ のときのタスクには $\mathbf{U}_{l+1}$ が必要であるから, 計算可能になった直後に計算し, ブロードキャストを行う. ブロードキャストにかかる時間は他のタスクを実行することで隠蔽できる.

表 1: 最内ループの命令列の一部.D は 2 命令同時発行を示す. 浮動小数点数演算命令 fnms がすべてのサイクルで発行されている.

D	fnms	$37,$65,$77,$37
D	lqd	$77,320($55)
D	fnms	$38,$67,$7,$9
D	shufb	$67,$8,$8,$101
D	fnms	$43,$66,$7,$4
D	shufb	$66,$94,$94,$101
D	fnms	$46,$65,$7,$6
D	shufb	$65,$113,$113,$101
	fnms	$17,$64,$93,$88
	fnms	$16,$64,$27,$89
	fnms	$54,$64,$25,$3
	fnms	$82,$65,$25,$85
	fnms	$26,$67,$25,$87
	fnms	$24,$66,$25,$86
	fnms	$22,$66,$27,$84
	fnms	$15,$65,$27,$83

シャッフル命令などはすべて浮動小数点数演算との 2 命令同時発行になり, 余った浮動小数点数演算は 1 命令発行になっている. SIMD 命令の記述は [7] を参考にした.SPU 命令セットは [8] で確認できる.

5 性能評価

図 6 に係数行列の大きさごとの実行性能を示す. ただし, すべての場合で右辺ベクトルは 1 本である. 係数行列が 4096 × 4096 の密行列のとき,7 基の SPE を用いたときの理論性能 179.2Gflops の 76.9 パーセントに相当する 137.8Gflops を達成した. 図 7 に係数行列を 4096 × 4096 に固定したときの SPE 数と性能の関係を示す. メモリアクセスボトルネックが解消され,SPE 数に比例して性能が

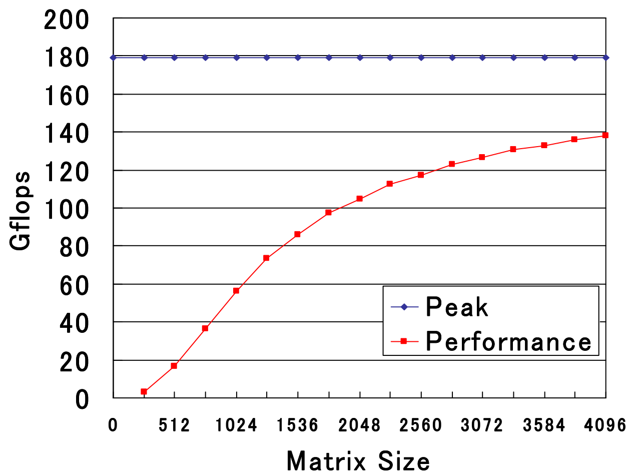

図 6: 問題サイズごとの実行性能. 係数行列が 4096 × 4096 のとき理論性能の 76.9 パーセントに当たる 137.8Gflops を達成した.

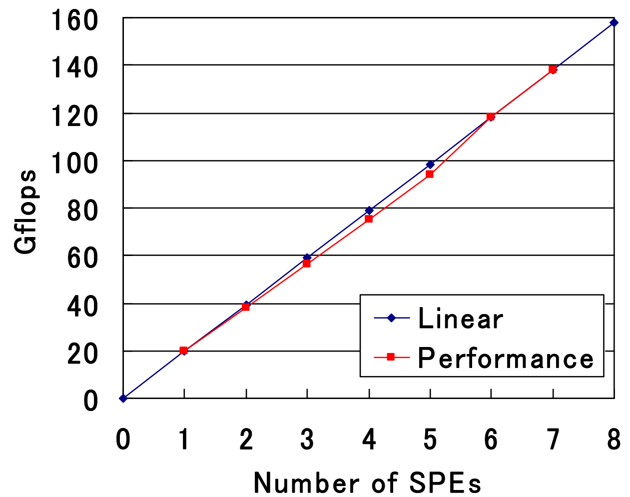

図 7: 係数行列を 4096 × 4096 に固定したときの SPE 数ごとの性能. メモリアクセスボトルネックの解消と SPE 間通信の隠蔽に成功したので, SPE 数に比例した性能の向上が見られる.

向上していることがわかる.

6 まとめ

データの時間的局所性を利用することで,Cell BE の性能を引き出せることを示した. 本プログラムの実行に必要なメモリ転送速度は 11.2GB/秒であり, ハードウェア性能の半分以下となっている. したがって,SPE の演算性能が 2 倍になってもメモリアクセスボトルネックは発生しない. このことは現行の Cell BE アーキテクチャには性能改善の余地があることを示唆している.

参考文献

[1] Haruki Satoshiro, Akihiko Konagaya. PDS Optimization Cycle: New Programming Optimization Methodology for Cell BE Architecture. SACSIS 2008.

[2] 里城晴紀, 小長谷明彦. オンチップ並列処理を利用した密行列 LU 分解の高速化. SACSIS 2008.

[3] IBM. Cell Broadband Engine Programming Tutorial Version 3.0, 2007.

[4] IBM. Cell Broadband Engine Programming Handbook Version 1.1, 2006.

[5] Thomas Chen, Ram Raghavan, Jason Dale, Eiji Iwata. Cell Broadband Engine Architecture and its first implementation, 2005.

[6] Jakub Kurzak, Jack Dongarra. Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the CELL Processor. Concurrency and Computation: Practice and Experience, Volume 19, Issue 10, Jul 2007.

[7] Sony Computer Entertainment Inc. SPU C/C++ 言語拡張 Version 2.3, 2006.

[8] Sony Computer Entertainment Inc. Synergistic Processor Unit 命令セット・アーキテクチャ Version 1.2, 2007.