情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　221–230 (Aug. 2008)

ある在庫管理問題の動的計画法による解法とCUDAを用いた高速化

李　天[†1]　河畠　工[†1]　山本有作[†1]
畝山多加志[†2]　張　紹良[†1]

本論文では，ある在庫管理計画問題の GPU による高速化について報告する．この問題は組合せ最適化問題であり，動的計画法を用いて解けるが，状態空間は連続かつ高次元であり，多大な計算量が必要である．本問題の解法を分析したところ，計算の並列性がきわめて高いこと，単精度で十分であること，メモリバンド幅への要求が高いことなどから，GPU 向きであることが分かった．そこで，この解法を GPU による汎用計算のための統合環境 CUDA を用いて GPU に移植した．NVIDIA 社 GeForce8800GTX 上で評価した結果，CPU（Intel Core2 Duo，1.86 GHz，2 コア使用）で 1.5 時間を要する最大規模の問題を約 6 分で解くことができ，15 倍の高速化が達成できた．

Solution of a Stock Management Problem by Dynamic Programming and Its Acceleration Using CUDA

Tian Li,[†1] Takumi Kawahata,[†1] Yusaku Yamamoto,[†1]
Takashi Uneyama[†2] and Shao-Liang Zhang[†1]

We report the result of speeding up the solution of a certain stock management problem using GPU. Our problem is a combinatorial optimization problem and can be solved by dynamic programming. However, the state space is continuous and high dimensional, hence the solution requires much computational work. We analyzed the solution algorithm and found that it is suited for GPU computing, since it has enormous parallelism, requires only single precision, and requires high memory bandwidth. We therefore ported the algorithm to NVIDIA's GeForce8800GTX using CUDA, an integrated environment for GPU computing. Numerical experiments show that for the largest problem that requires 1.5 hours on a CPU (Intel Core2 Duo, 1.86 GHz, using 2 cores), the GPU can find the solution in 6 minutes, achieving 15 times speedup.

1. はじめに

近年，GPU（Graphics Processing Unit）の進化は著しく，CPU よりも速いスピードで高性能化が進んでいる．特に，NVIDIA 社 GeForce8800GTX，AMD 社 R600 などの最新 GPU は，単精度演算で 500 GFLOPS 程度と，汎用 CPU の数十倍の演算能力を持つに至っている．このような GPU の演算能力を，グラフィックス演算に限らない一般の高性能計算に活用する GPGPU（General Purpose GPU）という利用法が，ここ数年注目を集めている[1]．GPGPU は，大量の計算が必要な分野，大量のデータを扱う分野で特に有効であり，流体計算，分子動力学，データマイニングなどの分野で有効性が確認されている．

しかし，従来の GPGPU では，グラフィックス API を利用した特殊なプログラミング[2]，あるいはストリーム言語によるプログラミング[3] が必要であった．そのため，一般のプログラム開発者にとっては敷居が高く，GPGPU の広範な普及の障害となっていた．

これに対して NVIDIA 社は 2006 年に，GPGPU のための統合環境 CUDA [4] を発表した．CUDA では，標準の C 言語に簡単な拡張を加えることで，NVIDIA 社の GPU を汎用的な計算に利用することを可能にしている．また，GeForce8800GTX をはじめとする最新 GPU 向けの最適化機能を備えており，チューニングのための情報もマニュアル[5] や web 上のフォーラム[4] を通じて広く公開されている．そのため，GPU の性能を引き出すプログラムの作成が従来に比べて格段に容易になっている．これらの利点により，CUDA は急速に普及し，重力多体計算[6]，分子軌道計算[7]，バイオインフォマティクス[8]，気象予報[9] など，様々な分野で活用が始まっている．

一方，我々は，企業の現場で現れる種々の最適化問題を高性能計算技術を用いて解く試みを行っている．その 1 つとして，ある在庫管理計画問題がある．この問題では，定期的に搬入されてくる原料を，ある制約条件を満たすよう複数の倉庫に振り分ける計画を立案する．これは組合せ最適化問題であり，動的計画法を用いて解けるが，汎用の PC ではデュアルコアマシンを用いても計算に 1.5 時間以上を要し，高速化が求められていた．

本問題を解くアルゴリズムを分析したところ，計算の並列性，必要な計算精度，メモリア

†1 名古屋大学大学院工学研究科計算理工学専攻
Department of Computational Science and Engineering, Graduate School of Engineering, Nagoya University
†2 京都大学大学院理学研究科物理学・宇宙物理学専攻
Department of Physics, Graduate School of Science, Kyoto University

クセスのバンド幅に対する要求などの点で，GPU による計算にきわめて適していることが明らかになった．

そこで本論文では，このアルゴリズムを CUDA を用いて GPU に移植し，性能評価を行った結果を報告する．GPU として GeForce8800GTX を用いた場合，最大規模の問題で Intel Core2 Duo（1.86 GHz，2 コア使用）に比べて約 15 倍の高速化が得られ，GPU 利用が大きな効果をあげる問題であることが確認できた．

本論文の構成は以下のとおりである．まず 2 章で，本研究で扱う在庫管理計画問題と動的計画法による解法を示す．3 章では統合環境 CUDA を紹介し，チューニングのための指針について触れる．4 章で本問題の解法の CUDA による実装法を示す．5 章で性能評価の結果を報告し，最後に 6 章でまとめと今後の課題を述べる．

2. 在庫管理計画問題と動的計画法による解法

2.1 本論文で扱う在庫管理計画問題

本節では，本論文で扱う在庫管理計画問題について述べ，制約付き最小化問題として定式化を行う．

いま，ある原料を格納する K 個の倉庫があるとする．各倉庫からは，毎日それぞれ決まった量だけ原料が搬出される．一方，原料を積載したトラックが毎日 1 台やってきて，K 個の倉庫のどれか 1 つにその原料を搬入する．これを N 日の期間にわたって行う．すると，各倉庫中の原料の充填率（0%〜100%）は毎日変化するが，これは，各日においてトラックが運んでくる原料をどの倉庫に搬入するかという選択の関数となる．この充填率が，0%からも 100%からもなるべく離れた中間の領域で推移するよう，各日のトラックの行き先を期間の最初の時点で計画したいというのが問題である．この問題を数学的に定式化するため，次のように定数・変数を定義する．なお，添字の範囲は，指定のない場合，$1 \le k \le K$，$1 \le n \le N$ とする．

$f_k^{(n)}$ ： 倉庫 k の n 日目（$0 \le n \le N$）最終時点における充填率
v_k ： 倉庫 k の容量
$c_k^{(n)}$ ： 倉庫 k からの n 日目における搬出量
$a^{(n)}$ ： n 日目に来るトラックの積載量
$g_k(f_k)$ ： 充填率が f_k のときの倉庫 k のペナルティ関数
$j(n)$ ： n 日目に来るトラックの行き先の倉庫番号

ここで，$f_k^{(0)}$，v_k，$c_k^{(n)}$，$a^{(n)}$ は与えられた定数である．関数 $g_k(f_k)$ は，$f_k = 0$ (空)，$f_k = 1$ (満杯）で大きな値をとり，$f_k = 0.5$ 付近で最小値をとる与えられた関数である．$j(n)$ は独立変数である．また，$f_k^{(n)}$（$1 \le n \le N$）は次の式により定まる．

$$f_k^{(n)} v_k = f_k^{(n-1)} v_k - c_k^{(n)} + a^{(n)} \delta_{k,j(n)} \quad (1 \le k \le K,\ 1 \le n \le N). \tag{1}$$

この式は，第 n 日目の倉庫 k における原料の保存則を表す．右辺第 3 項において δ_{kj} はクロネッカーのデルタであり，$j(n) = k$ のときのみ，倉庫 k 中の原料が $a^{(n)}$ だけ増えることを示す．本論文で扱う在庫最適化問題は，制約式 (1) の下で，目的関数

$$G\left(\{f_k^{(0)}\}_{k=1}^K; \{j(n)\}_{n=1}^N\right) \equiv \sum_{k=1}^{K} \sum_{n=1}^{N} g_k\left(f_k^{(n)}\right) \tag{2}$$

を最小化する問題として定式化される．

なお，本問題は実際に企業の現場で高速解法が求められている実問題である．実際の問題では，1 日に 2 台のトラックが来る場合，日によって特定の倉庫には搬入できない場合などがあり，より複雑であるが，それらへの対処は比較的容易であるため，本論文では上記の簡単化した問題を扱う．

2.2 動的計画法の適用

制約付き最小化問題 (1)，(2) は独立変数 $\{j(n)\}_{n=1}^N$ が有限個の値のみをとる組合せ最適化問題である．そのため，0-1 整数計画問題に帰着させて解く方法，メタヒューリスティクスを使う方法などが考えられる．しかし，前者の場合は目的関数の非線形性が問題となる．一方，後者については遺伝的アルゴリズムの適用を検討したが，$\{j(n)\}_{n=1}^N$ を染色体とする自然なコーディングを採用した場合，両方の親の解の持つ良い性質を引き継ぐ交叉演算子を定義するのが容易ではないことが分かった．そこで本研究では，この問題が多段決定問題の形をしていることに着目し，動的計画法[10),11)] を適用する．

まず，第 $n-1$ 日目の最終時点における各倉庫の充填率が与えられたときの n 日目以降の部分目的関数を

$$G^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^K; \{j(m)\}_{m=n}^N\right) \equiv \sum_{k=1}^{K} \sum_{m=n}^{N} g_k\left(f_k^{(m)}\right) \tag{3}$$

と定義する．また，部分的な制約付き最小化問題

$$\min_{\{j(m)\}_{m=n}^{N}} G^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K};\{j(m)\}_{m=n}^{N}\right)$$
$$\text{subject to}$$
$$f_k^{(m)} v_k = f_k^{(m-1)} v_k - c_k^{(m)} + a^{(m)}\delta_{k,j(m)}$$
$$(1 \le k \le K,\ n \le m \le N). \tag{4}$$

を考え，その最小値を $\bar{G}^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K}\right)$ とする．

すると，式 (3) より明らかに，

$$G^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K};\{j(m)\}_{m=n}^{N}\right) = \sum_{k=1}^{K} g_k\left(f_k^{(n)}\right) + G^{(n+1)}\left(\{f_k^{(n)}\}_{k=1}^{K};\{j(m)\}_{m=n+1}^{N}\right) \tag{5}$$

が成り立つ．ここで，両辺の $\{j(m)\}_{m=n}^{N}$ に関する最小値をとり，右辺において等式

$$\min_{\{j(m)\}_{m=n}^{N}} = \min_{j(n)}\left[\min_{\{j(m)\}_{m=n+1}^{N}}\right] \tag{6}$$

を用いると，次の式（Bellman 方程式[10),11)]）が得られる．

$$\bar{G}^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K}\right) = \min_{j(n)}\left[\sum_{k=1}^{K} g_k\left(f_k^{(n)}\right) + \bar{G}^{(n+1)}\left(\{f_k^{(n)}\}_{k=1}^{K}\right)\right]. \tag{7}$$

この式は，もし $n+1$ 日目以降の部分目的関数 $\bar{G}^{(n+1)}\left(\{f_k^{(n)}\}_{k=1}^{K}\right)$ が $\{f_k^{(n)}\}_{k=1}^{K}$ のあらゆる値に対して与えられているならば，$j(n)$ に関する 1 期間の最適化を行うことで，n 日目以降の部分目的関数 $\bar{G}^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K}\right)$ が $\{f_k^{(n-1)}\}_{k=1}^{K}$ のあらゆる値に対して計算できることを意味する．また，この 1 期間の最適化により，$\{f_k^{(n-1)}\}_{k=1}^{K}$ が与えられたときの n 日目の最適な選択 $j\left(n;\{f_k^{(n-1)}\}_{k=1}^{K}\right)$ も求められる．そこで，終端条件 $\bar{G}^{(N+1)}\left(\{f_k^{(N)}\}_{k=1}^{K}\right) = 0$ から始めて，式 (7) を使って $\bar{G}^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K}\right)$ を $\{f_k^{(n-1)}\}_{k=1}^{K}$ のあらゆる値に対して計算する処理を $n = N, N-1, \ldots, 1$ と続けていけば，最終的に $\bar{G}^{(1)}\left(\{f_k^{(0)}\}_{k=1}^{K}\right) = 0$ が求められる．定義より，これが目的関数の最小値となる．また，与えられた初期値 $\{f_k^{(0)}\}_{k=1}^{K}$ に対する最適な選択 $\{j(n)\}_{n=1}^{N}$ を求めるには，$n = 1$ から始めて，$j(n) = j\left(n;\{f_k^{(n-1)}\}_{k=1}^{K}\right)$ と式 (1) を交互に使って $j(n)$ と $\{f_k^{(n)}\}_{k=1}^{K}$ を順次計算していけばよい．以上が本問題に対する動的計画法の適用である．

以下では，動的計画法の用語に従い，各日において充填率 $\{f_k^{(n)}\}_{k=1}^{K}$ のなす K 次元空間を**状態空間**，$\bar{G}^{(n)}\left(\{f_k^{(n-1)}\}_{k=1}^{K}\right)$ を**価値関数**と呼ぶ．また，$n = N, N-1, \ldots, 1$ の順に価値関数を求める部分の計算を**後ろ向き計算**，$n = 1, 2, \ldots, N$ の順に最適な選択を求める部分の計算を**前向き計算**と呼ぶ．

2.3　状態空間の次元縮小

本問題の特性を利用すると，状態空間の次元を 1 だけ縮小できる．実際，式 (1) を辺々1 から n まで加え，さらに k について 1 から K まで和をとって $\sum_{k=1}^{K}\delta_{k,j(n)} = 1$ を用いると，次式が得られる．

$$\sum_{k=1}^{K} f_k^{(n)} v_k = \sum_{k=1}^{K} f_k^{(0)} v_k - \sum_{m=1}^{n}\sum_{k=1}^{K} c_k^{(n)} + \sum_{m=1}^{n} a^{(n)}. \tag{8}$$

右辺は定数であるから，これは $f_1^{(n)}, f_2^{(n)}, \ldots, f_K^{(n)}$ のうち $K-1$ 個が決まれば残りの 1 個も決まることを示している．したがって，状態空間としては $\{f_k^{(n)}\}_{k=1}^{K-1}$ のなす $K-1$ 次元空間を考えればよい．以下では，この方法を採用する．

2.4　アルゴリズム

以上で説明した動的計画法を用いて計算機上で目的関数の最小化を行うには，状態空間を離散化する必要がある．そこで，$[0,1]$ を L（L はある自然数）分割して $h = 1/L$ とし，各 f_k が離散値 $f_k = hl_k$（$l_k = 0, 1, \ldots, L$）のみをとるとする．また，

$$\bar{G}^{(n)}_{l_1,l_2,\ldots,l_K} \equiv \bar{G}^{(n)}\left(\{hl_k\}_{k=1}^{K-1}\right) \tag{9}$$

とおく．これを用いて Bellman 方程式 (7) を書き換えることを考える．

ところが，いま $\{f_k^{(n-1)}\}_{k=1}^{K-1}$ が格子点上にあったとしても，これから式 (1) を用いて計算した $\{f_k^{(n)}\}_{k=1}^{K-1}$ が格子点上にあるとは限らない．この場合，式 (7) における $\bar{G}^{(n+1)}$ の値が求められないという問題が生じる．

そこで，$\bar{G}^{(n+1)}$ の計算にあたっては，$K-1$ 次元空間の点 $f_1^{(n)}, f_2^{(n)}, \ldots, f_{K-1}^{(n)}$ を含み，格子点を頂点とする最小の $K-1$ 次元超立方体を考え，その 2^{K-1} 個の頂点での $\bar{G}^{(n+1)}$ の値を用いて多重線形補間（各方向の座標について 1 次式であるような補間式を使う補間）を行うことにする．多重線形補間を使う理由は，動的計画法の価値関数は C^0 級という以上の滑らかさを持たないため，高次の補間式を使う利点がないからである．以下のアルゴリズムの記述では，$\bar{G}^{(n+1)}\left(\{f_k\}_{k=1}^{K-1}\right)$ の値を多重線形補間により求める関数を

$$\text{Interpolate}(\bar{G}^{(n+1)}, f_1, f_2, \ldots, f_{K-1}) \tag{10}$$

と書く．

以上の準備のもとに，2.2 節，2.3 節で述べた動的計画法による在庫管理問題の解法をアルゴリズムとして書くと次のようになる．

［アルゴリズム 1：在庫管理問題の解法］
do $l_1 = 0, L$
　do $l_2 = 0, L$
　　do $l_3 = 0, L$
　　$\bar{G}^{(N+1)}_{l_1,l_2,l_3} = 0$
　　end do
　end do
end do
do $n = N, 1, -1$
　do $l_1 = 0, L$
　　do $l_2 = 0, L$
　　　do $l_3 = 0, L$
　　　$\bar{G}^{(n)}_{l_1,l_2,l_3} = +\infty$
　　　do $j = 1, 4$
　　　$\bar{G}_{\text{tmp}} = 0$
　　　do $k = 1, 3$
　　　$f_k = (hl_kv_k - c^{(n)}_k + a^{(n)}\delta_{kj})/v_k$
　　　$\bar{G}_{\text{tmp}} = \bar{G}_{\text{tmp}} + g_k(f_k)$
　　　end do
　　　式 (8) により f_4 を計算
　　　$\bar{G}_{\text{tmp}} = \bar{G}_{\text{tmp}} + g_4(f_4)$
　　　$\bar{G}_{\text{tmp}} = \bar{G}_{\text{tmp}}$
　　　$+\text{Interpolate}(\bar{G}^{(n+1)}, f_1, f_2, f_3)$
　　　if $\bar{G}_{\text{tmp}} < \bar{G}^{(n)}_{l_1,l_2,l_3}$ **then**
　　　$\bar{G}^{(n)}_{l_1,l_2,l_3} = \bar{G}_{\text{tmp}}$
　　　$j(n, l_1, l_2, l_3) = j$
　　　end if
　　end do
　end do
end do
end do

なお，前向き計算の部分は，2.2 節の説明から容易に分かるとおり，計算量が $O(NK)$ と少ない．そのため，本論文での高速化の対象から外し，以降では後ろ向き計算の部分のみを考える．また，アルゴリズム中では，簡単のため $K = 4$ としている．

本アルゴリズムの計算量は，容易に分かるように $O(NL^{K-1}K^2)$ である．一方，可能な K^N 通りの組合せ全部について目的関数値を比較する全探索法の計算量は $O(K^{N+1}N)$ であるから，本アルゴリズムは K が小さく，N が大きい場合に全探索法より有利である．また，配列として $\bar{G}^{(n)}_{l_1,\ldots,l_{K-1}}$，$\bar{G}^{(n+1)}_{l_1,\ldots,l_{K-1}}$，$j(n, l_1, \ldots, l_{K-1})$ をメモリ上に格納することが必要であり，そのため $O(L^{K-1})$ の記録領域が必要である．なお，$\bar{G}^{(m)}_{l_1,\ldots,l_{K-1}}$ $(m \geq n+2)$ については，n 日目の計算を行う時点ではもう使わないので，捨ててしまってよい．また，$j(m, l_1, \ldots, l_{K-1})$（$m \geq n+1$）については，後に前向き計算を行うまでは使わないので，ディスクに退避すればよい．

なお，目的関数 $g_k(f_k)$ として，本問題では区分 1 次関数を用いている．

2.5　アルゴリズムの特徴

本アルゴリズムの特徴を述べると次のようになる．

(a) 計算量，記憶領域がそれぞれ $O(NL^{K-1}K^2)$，$O(L^{K-1})$ と多い．
(b) 各時間ステップにおいては L^{K-1} 個の格子点に対する計算を行うが，これらは完全に独立であり，並列に計算可能である．
(c) 計算精度は単精度で十分である．これは，誤差の主な原因が $\bar{G}^{(n+1)}$ の補間であり，それに比べて丸め誤差は十分小さいからである．
(d) メモリバンド幅に対する要求が大きい．実際，j に関するループの 1 回の繰返しに注目すると，最も演算量が多いのは多重線形補間を行う関数 Interpolate であるが，ここでは 2^{K-1} 個の点での $\bar{G}^{(n+1)}$ での値を使い，$O(2^{K-1})$ 回の演算により補間を行う．したがって，演算とメモリアクセスの比は小さな定数となっている．

問題サイズを決めるパラメータとして，適用対象の実問題では，倉庫数 K が最大 5，期間の長さ N が最大 90 日となっている．一方，各方向の分割数 L をどのくらい大きくとれば十分良い解が得られるかは，入力パラメータ値により異なる．事前の数値実験の結果から，充填率が全体的に 0%または 100%近くで推移する場合など，制御が難しい場合は L を大きくとる必要があり，最大で 80 程度にとる必要があることが分かっている．$K = 5$，$N = 90$，$L = 80$ の場合，Intel Core2 Duo（1.86 GHz，2 コア使用）の PC では求解に 1.5 時間程度を要し，所要メモリは 400 MB 程度である．実際の使用時には，得られた解を見ながらパラメータを調整して繰り返し計算を行うため，計算時間を数分程度に抑えることが望ましい．

3. GPGPU のための統合環境 CUDA

3.1 CUDA の概要

本章では，前節で説明したアルゴリズムを GPU を用いて高速化するための準備として，CUDA について簡単に紹介する[4),5)]．CUDA は NVIDIA 社の GPU を利用して汎用的な計算を行うための統合プログラミング環境であり，nvcc コンパイラ，BLAS や FFT などのライブラリ，シミュレータなどから構成される．nvcc コンパイラは，C 言語の拡張で記述されたソースを入力とし，CPU と GPU で実行されるコードを出力する．本研究では，この nvcc を使用する．

3.2 プログラミングモデル

3.2.1 GPU の利用法

CUDA では，main 関数は CPU で実行され，そこから GPU で実行される関数（グローバル関数）を呼び出す形で GPU を利用する．CPU と GPU は別々のメモリ空間を持ち，両者の間でのデータ転送は，CPU プログラムの側から cudaMemcpy 関数を実行することにより行う．

3.2.2 ブロックとスレッドによる並列化

グローバル関数を呼び出す際には，ブロック数 N_{block} とスレッド数 N_{thread} という 2 つのパラメータを指定する．これにより，グローバル関数は，それぞれが N_{thread} 個のスレッドを持つ N_{block} 個のブロックにより並列実行される．各スレッドは，自分のブロック番号，ブロック内のスレッド番号を参照でき，それを利用して自分の担当する処理を行う．また，ブロック内のスレッド全体で同期をとる関数が用意されている．

3.2.3 メモリモデル

GPU のメモリ空間としては，いくつかの種類がある．まず，全ブロックの全スレッドからアクセスできるグローバルメモリ空間がある．次に，各スレッドのみからアクセスできるローカル空間がある．さらに，同一ブロックのスレッドのみからアクセスできる共有メモリ空間がある．グローバル関数あるいはそこから呼ばれる関数の中で宣言される自動変数はローカル空間にとられる．一方，CPU プログラムから cudaMemcpy 関数により転送できる領域はグローバルメモリ空間である．グローバルメモリ空間における配列は，CPU プログラムから cudaMalloc 関数により確保され，グローバル関数の開始/終了にかかわらず，cudaFree 関数により解放されるまで保持される．また，グローバル空間と同様，すべてのスレッドからアクセスできる変数として，コンスタントとテクスチャという 2 つのメモリ空間がある．これらは GPU からは読み出し専用となるが，その代わり，キャッシュによる高速アクセスが可能であるという利点を持つ．

3.3 本研究で用いる GPU

本研究では，GPU として NVIDIA 社の GeForce8800 GTX を搭載したグラフィックボードを用いる．GeForce8800 GTX は，各々が 8 個のプロセッサ，8,192 個のレジスタ，16 KB の共有メモリを持つ SIMD 型並列計算ユニット 16 個を 1 チップ上に搭載し，1.35 GHz のクロックで動作する．CUDA を使った場合，利用可能なピーク性能は単精度で 345.6 GFLOPS である．倍精度演算は行えない．また，ボード上には 768 MB の DRAM があり，GPU との間で最大 86.4 GB/s の速度でデータ転送を行える．ただし，レイテンシは 400〜600 サイクルと大きい．また，ボード上の DRAM と PC の主メモリ間の転送速度は PCI-Express バスの速度により制約されるため，2 GB/s 程度と小さい．

CUDA を使う場合，各ブロックは 1 個の SIMD 型並列計算ユニットにより実行される[⋆1]．一般にスレッド数はプロセッサ数より多く設定し，時分割で実行する．ローカルメモリ空間の変数はまずレジスタ，ついで DRAM に置かれ，共有メモリ空間の変数は共有メモリに，それ以外の変数は DRAM に置かれる．8,192 個のレジスタはブロック内のすべてのスレッドで分割されるため，スレッド数の上限は，アルゴリズムの並列性に加え，各スレッドが使うレジスタ数により定まる．

3.4 チューニングの指針

CUDA を使って GeForce8800 GTX の性能を引き出すには，ボード上の DRAM の転送速度は大きいがレイテンシも大きいこと，PC の主メモリとの転送速度が小さいこと，それぞれの並列計算ユニットが SIMD 型であることなどを考慮する必要がある．具体的には，次

⋆1 複数のブロックが 1 個の SIMD 型並列計算ユニットにより実行される場合もあるが，以下では考えない．

の条件を満たすようにプログラミングを行う[5]．

(A) データ参照の局所性を高め，できるだけチップ上のレジスタおよび共有メモリ内で演算を行う．また，キャッシュが利用できるコンスタント，テクスチャメモリを積極的に利用する．

(B) 各ブロックのスレッド数をできるだけ大きくする．これにより，1 個のプロセッサが多くのスレッドを時分割で実行できるようになり，ボード上 DRAM のアクセスのレイテンシを隠蔽しやすくなる．

(C) できるだけボード内で演算を行い，PC の主メモリとのデータ転送は最小限にする．

(D) 同一ブロック内[⋆1]のスレッドが if 文の異なる分岐を実行するようなプログラムはなるべく避ける．これは，SIMD 型並列計算機では 1 度に 1 つの命令列しか実行できず，あるスレッド群が if 文の片方の分岐を実行している間，残りのスレッド群はアイドル状態になるからである．

(E) ボード上の DRAM をアクセスする場合，ブロック内のスレッド群がアクセスするメモリ領域は連続になるようにする．また，アクセス開始位置がアラインメントに関する条件を満たすようにする．

次章ではこれらの条件に基づき，在庫管理計画問題に対する動的計画法の CUDA による高速化を検討する．

4. 在庫管理計画問題に対する動的計画法の CUDA による高速化

4.1 アルゴリズムの特徴と GPU への親和性

まず，2.5 節で述べたアルゴリズムの特徴 (a)〜(d) に基づき，本アルゴリズムが GPU による実行に適しているかどうかを検討する．

(a) 計算量が多くて高速化が必要なのは，先に述べたとおりである．一方，所要メモリは最大規模の問題で 400 MB 程度であるが，これは今回使うグラフィックボードの搭載メモリより小さい．したがって，CPU からのデータ転送を最小限に抑え，GPU ボード内でほとんどの計算を行える可能性がある．

(b) 並列度は，最大規模の問題で 81^4 と十分大きい．これは，レジスタ数の制限内であれば，スレッド数を十分大きくできることを意味する．

(c) 計算は単精度でよいため，一般に単精度の計算しかできない GPU での計算に適している．

(d) 本アルゴリズムはメモリバンド幅に対する要求が大きいが，GPU では，演算性能に対するボード内のメモリバンド幅が比較的大きく，たとえば今回の GeForce8800GTX では 345.6GFLOPS に対し 86.4 GB/s である．これに対し，たとえば浮動小数点アクセラレータである ClearSpeed CSX600 では，96GFLOPS に対し 6.4 GB/s と，ボード内のメモリバンド幅がずっと小さい．

以上の点を考慮すると，本アルゴリズムはきわめて GPU に適していると考えられる．

4.2 CUDA による実装

4.2.1 GPU での実行部分

本研究では，アルゴリズム 1 における n に関するループの 1 回の繰返し，すなわち配列 $\bar{G}^{(n+1)}$ の値を入力として配列 $\bar{G}^{(n)}$ および配列 j の値を計算する部分を 1 個のグローバル関数として GPU で実行することにする．以下，このグローバル関数を BackwardOneDay と名付ける．後ろ向き計算においては，この関数を N 回呼び出すことになる．

4.2.2 各変数のメモリ空間への割当て

関数 BackwardOneDay を GPU で実行するにあたり，その中の各変数，配列を GPU のどのメモリ空間に割り当てるかを決定する必要がある．ここでは，3.2.3 項で述べた各メモリ空間の特性に基づき，次のような割当てを行った．

- 配列 $\bar{G}^{(n+1)}$，$\bar{G}^{(n)}$ は全スレッドからアクセスする必要がある．また，$\bar{G}^{(n)}$ は，1 回の BackwardOneDay の終了から次の呼び出しまで，値を保持する必要がある．したがって，これらの配列はグローバルメモリ空間に置く．
- 配列 j は，CPU に転送する必要があるため，グローバルメモリ空間に置く．
- パラメータ $a^{(n)}$，$c_k^{(n)}$，v_k はコンスタントメモリ空間に置く．
- 配列 δ_{ij} は共有メモリ空間に置く．これは，コンスタントメモリ空間でもよいが，共有メモリ空間の方が 2 次元配列を扱いやすいので，そのようにした．
- スレッドごとの中間変数はレジスタに置く．

4.2.3 ブロックとスレッドによる並列化

最大規模の問題では，倉庫数 K が 5 であるから，関数 BackwardOneDay は l_1，l_2，l_3，l_4 に関する 4 重の完全並列ループにより構成される．そこで，これらのループをブロックおよびスレッドに割り当てる．

まず，スレッド並列化については，レジスタ数の制限の範囲でスレッドをできるだけ多くとれるようにするため，2 方向を用いる．さらに，連続する番号のスレッド群がアクセスす

⋆1 正確には，スレッド番号を 16 で割った商が同じ値になるスレッド群．

るメモリ領域が連続になるよう，l_3，l_4 の 2 つの添字をスレッド並列化に用いることにする．ただし，ここでは添字 l_4 が変化するとき配列が連続アクセスになると仮定した．具体的な割当て法としては，スレッド数を $N_{\mathrm{thread}} = N_{\mathrm{thread},x} \times N_{\mathrm{thread},y}$ として 2 次元サイクリック分割を行い，添字 l_3，l_4 を持つ要素が，番号

$$\mathrm{mod}(l_3, N_{\mathrm{thread},y}) \times N_{\mathrm{thread},x} + \mathrm{mod}(l_4, N_{\mathrm{thread},x}) \quad (11)$$

番のスレッドに割り当てられるようにする．

一方，ブロックに関する並列化は，添字 l_2 について 1 次元サイクリック分割を用いて行う．l_1 に関するループについては，各スレッドで逐次的に実行する．

4.2.4　CPU と GPU 間のデータ転送

アルゴリズム 1 から分かるとおり，BackwardOneDay では，CPU から GPU に転送が必要なデータは $a^{(n)}$，$c_k^{(n)}$，v_k，L，h などの少数のパラメータのみである．計算結果の配列 $\bar{G}^{(n)}$ は次の BackwardOneDay の呼び出しで $\bar{G}^{(n+1)}$ として用いるが，CPU に転送する必要はない．

一方，配列 j は CPU に転送する必要がある．j の各要素を 4 ビットでコーディングした場合，81^4 メッシュでは 21 MB 程度であり，90 日間の後ろ向き計算をした場合でも，全転送量は 1.8 GB 程度である．したがって，CPU と GPU の転送速度を考慮すると，転送時間は数秒〜数十秒のオーダと考えられる．これは，CPU のみでこの問題を解いた場合の計算時間である 1.5 時間に比べてきわめて小さい．以上より，CPU と GPU 間のデータ転送は，本アルゴリズムにおいて高速化のネックとはならないと考えられる．

5. 性能評価

5.1 評価条件

前章の方針に基づいて CUDA プログラムを作成し，C 言語による CPU プログラムとの性能比較を行った．計算機環境は**表 1** のとおりである．CPU はデュアルコアであるため，pthread ライブラリを用いて並列化し，1 コア使用時と 2 コア使用時の両方の性能を測定した．並列化は，最外側の添字 l_1 のループについて行った．なお，CPU での計算も単精度で行っている．

評価例題としては，倉庫数 $K = 5$，期間 $N = 90$ の最大規模の問題を用い，分割数 L を 40 から 80 まで変えて計算を行った．

表 1　実験に使用した計算機環境
Table 1　Computational environments used in the experiments.

項目	条件
CPU	Intel Core2Duo 1.86 GHz
メモリ	2 GB
OS	openSUSE Linux 10.2（64 ビット版）
コンパイラ	gcc ver. 4.1.2（オプション -O3）
GPU	NVIDIA GeForce8800GTX
GPU メモリ	768 MB
CUDA コンパイラ	nvcc ver. 1.0

表 2　CPU と GPU の実行時間（秒）
Table 2　Computational time (in sec.) on the CPU and GPU.

L	40	50	60	70	80
CPU (1 コア)	745.2	1,768.6	3,612.5	6,617.7	11,197.1
CPU (2 コア)	384.1	904.1	1,841.7	3,367.4	5,745.0
GPU	23.4	61.7	107.0	227.5	367.4
加速 (1 コア比)	31.8	28.7	33.8	29.1	30.5
加速 (2 コア比)	16.4	14.7	17.2	14.8	15.6
N_{thread}	41×3	51×1	61×1	32×4	32×2
目的関数値	21,783.6	22,376.6	21,813.3	21,724.6	21,659.5

5.2 評価結果

5.2.1　GPU による速度向上効果

まず，GPU 利用による速度向上効果を見るため，CUDA におけるスレッド数 N_{thread} を最適化した場合について，CPU のみのプログラムと GPU を使ったプログラムの実行時間を比較する．ここで，各スレッドのレジスタ使用本数は nvcc の-cubin オプション[5] より 46 本と判明したため，N_{thread} が $8{,}192/46 = 178$ 以下となるようにし，$N_{\mathrm{thread},x}$，$N_{\mathrm{thread},y}$ については次項で述べる組合せの中で最適なものを選択した．

結果を**表 2** および**図 1** に示す．目的関数値は，CPU でも GPU でも同一であった．また，$L = 80$ の場合の充填率の時間変化を**図 2** に示す．図より，充填率が 0%からも 100%からも離れた領域で推移する良い解が得られていることが分かる．

表 2 より，GPU の使用によって最高で 33.8 倍（対 1 コア比）ないし 17.2 倍（対 2 コア比），最大規模の $L = 80$ の問題の場合に 30.5 倍（対 1 コア比）ないし 15.6 倍（対 2 コア比）の速度向上が得られていることが分かる．また，後者の場合，GPU での実行時間は約

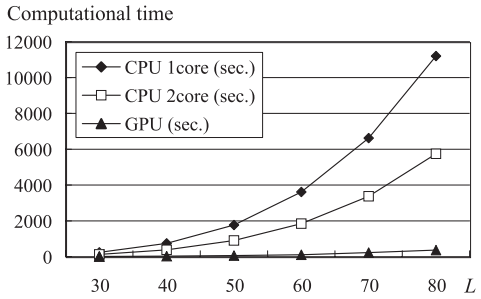


図 1　CPU と GPU の実行時間（秒）
Fig. 1　Computational time (in sec.) on the CPU and GPU.

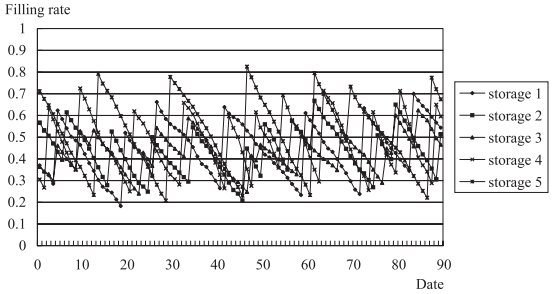


図 2　$L = 80$ の場合の各倉庫の充填率の時間変化
Fig. 2　Time evolution of the filling rates of each storage for the $L = 80$ case.

6 分であり，数分程度という実問題の要求を満たしている．$L = 60$ の場合に特に GPU の性能が良いのは，L が 32 の倍数に近く，スレッド群が無駄なく使われているのが原因ではないかと考えられる．

なお，上記の問題のパラメータは比較的条件が良いため，$L = 40$ でもたまたま目的関数の小さい良い解が得られているが，条件の悪い問題では，倉庫のあふれ/払底を防ぐために $L = 80$ 程度の分割数による精度の良い計算が必要である．このような問題の例として，倉庫の容量を最大限に使うよう搬入量を定めた結果，充填率が高い水準で推移する場合があげられる．このような問題に対し，$L = 60$ および $L = 80$ として計算した結果をそれぞれ**図 3**，**図 4** に示す．$L = 60$ の場合，充填率の最大値は 96%以上で倉庫あふれの危険があるが，$L = 80$ とすることで最大値が 90%に抑えられ，解が大きく改善されていることが分かる．

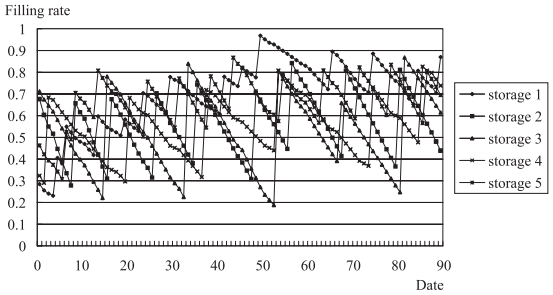


図 3　条件の悪い問題での充填率の時間変化（$L = 60$）
Fig. 3　Time evolution of the filling rates in an ill-conditioned problem for the $L = 60$ case.

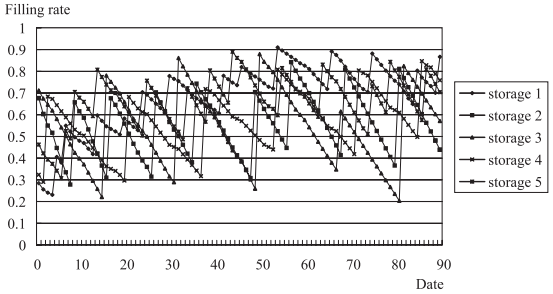


図 4　条件の悪い問題での充填率の時間変化（$L = 80$）
Fig. 4　Time evolution of the filling rates in an ill-conditioned problem for the $L = 80$ case.

5.2.2　スレッド数による性能変化

次に，スレッド数を 2 通りのやり方で変えた場合の GPU の実行時間の変化を調べた．第 1 の方法としては，$N_{\mathrm{thread},x} = L + 1$ とし，$N_{\mathrm{thread},y}$ を 1，2，3 と変化させた．この方法では，無駄になるスレッドが少ないが，連続した 32 個のスレッドが連続するアドレスをアクセスするという条件は満たされない．また，L が大きくなると，スレッド数の上限から，スレッド数の自由度が少なくなる．一方，第 2 の方法では，$N_{\mathrm{thread},x} = 32$ と固定し，$N_{\mathrm{thread},y}$ を 1，2，3，4 と変化させた．この方法では，L の値によっては無駄になるスレッドが多いが，アクセスの連続性は満たされる．

両者を比較した結果を**表 3** に示す．表中での太字は，実行時間が最も短いケース，横線はリソース不足で実行できなかったケースを表している．表より，最適なスレッド数の決め方は L により変化し，どちらの方法が最適かは一概にいえないことが分かる．ただし，ス

表 3　GPU 実行時間のスレッド数による変化（秒）
Table 3　Computational time (in sec.) on the GPU as a function of the number of threads.

$N_{\text{thread}} \setminus L$	40	50	60	70	80
$(L+1)\times 1$	35.8	**61.7**	**107.0**	262.6	426.5
$(L+1)\times 2$	28.4	65.0	107.2	–	–
$(L+1)\times 3$	**23.4**	–	–	–	–
32×1	59.5	96.3	146.9	407.6	549.7
32×2	35.3	61.8	107.9	258.0	**367.4**
32×3	33.6	73.1	120.7	260.5	421.6
32×4	29.3	64.3	108.6	**227.5**	380.6

レッド数を 32×4 として 128 スレッドを確保し，かつ連続アクセスができるようにした場合は，どの L においても最適値に準じる性能が出ている．したがって，リソースの許す範囲でスレッド数を 32 のできるだけ大きな倍数にとることは，1 つの有効な戦略ではないかと考えられる．

6. おわりに

本研究では，ある在庫管理計画問題の動的計画法による解法を，GPU による汎用計算のための統合環境 CUDA を用いて高速化した結果について報告した．本解法は GPU での実行に適した特性を持っており，その特性を生かすよう実装を行った結果，デュアルコア CPU で 1.5 時間かかっていた大規模問題を GeForce8800GTX を用いて 6 分で解くことができた．

今後の課題としては，より倉庫数の大きい大規模問題への適用があげられる．本論文で採用した格子を用いた動的計画法では，計算量が倉庫数に関して指数的に増えるため，そのままでは倉庫数の大きい問題を扱うことは困難である．そのため，関数近似やモンテカルロ法を用いた動的計画法[12)]，強化学習[11),12)] などの利用が考えられる．また，大規模問題では，データが GPU ボードのメモリ上に収まらなくなり，CPU と GPU 間でデータを密にやりとりしながら計算を行わなければならない場合も考えられる．その場合の効率的なデータ転送法も検討課題である．

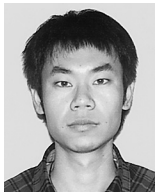
本問題のような高次元の状態空間を持つ動的計画法は，複数資産に対するポートフォリオ最適化[13)]，複数資産に依存するオプションの価格計算[14)]，実行時自動チューニングにおける逐次実験計画[15)] など，様々な問題で必要となる．これらは計算量が大きな問題として知られているが，本論文の解法と同様の特徴を持つため，GPU 利用により大幅な高速化が図れる可能性がある．この点に関する検討も今後の課題である．

謝辞　本論文を丁寧に査読してくださり，有益な助言をくださった査読者の方々に感謝いたします．また，日頃から高性能計算に関してご議論いただいている自動チューニング研究会の皆様に感謝いたします．なお，本研究は名古屋大学 21 世紀 COE プログラム「計算科学フロンティア」，科学研究費補助金基盤研究（C）（課題番号 18560058），および科学研究費補助金特定領域研究「i-explosion」（課題番号 18049014）の補助を受けている．

参 考 文 献

1) http://www.gpgpu.org/
2) SIGGRAPH 2005 GPGPU Course. http://www.gpgpu.org/s2005/
3) BrookGPU. http://graphics.stanford.edu/projects/brookgpu/
4) NVIDIA CUDA. http://www.nvidia.com/object/cuda_home.html
5) CUDA Programming Guide 1.1. http://www.nvidia.com/object/cuda_develop.html
6) Belleman, R.G., Bedorf, J. and Zwart, S.P.: High Performance Direct Gravitational N-body Simulations on Graphics Processing Units – II: An implementation in CUDA, to appear in *New Astronomy*.
7) Yasuda, K.: Two-Electron Integral Evaluation on the Graphics Processor Unit, *Journal of Computational Chemistry*, Vol.29, No.3, pp.334–342 (2007).
8) Manavski, S.A. and Valle, G.: CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment, preprint. http://www.manavski.com/downloads/SWcuda01-11-pics.pdf
9) Michalakes, J. and Vachharajani, M.: GPU Acceleration of Numerical Weather Prediction, preprint. http://www.mmm.ucar.edu/wrf/WG2/michalakes_lspp.pdf
10) Bellman, R.: *Dynamic Programming*, Dover Publications (2003).
11) Bertsekas, D.P. and Tsitsiklis, J.N.: *Neuro-Dynamic Programming*, Athena Scientific (1996).
12) Sutton, R.S. and Barto, A.G.: *Reinforcement Learning*, MIT Press (1998).
13) Garlappi, L., Naik, V. and Slive, J.: Portfolio Selection with Multiple Assets and Capital Gains Taxes, preprint. http://ssrn.com/abstract=274939
14) 高柳健一郎，山本有作，杉原正顯：2 資産に依存するオプションの高速・高精度価格計算手法，情報処理学会論文誌：コンピューティングシステム，Vol.46, No.SIG12 (ACS11), pp.289–298 (2005).
15) 須田礼仁：実行時自動チューニングのための逐次実験計画—分散が共通な 2 つの正規分布の場合，情報処理学会研究報告，2006-HPC-106, pp.13–18 (2006).

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月 5 日採録)



李　　天

1983 年生．2008 年名古屋大学工学部物理工学科（応用物理学コース）卒業．現在，東京大学大学院工学系研究科精密機械工学専攻修士課程在学中．名古屋大学在学中は GPU を用いた数値計算アルゴリズムの高速化に関する研究に従事．現在はロボットに関する研究に従事．

河畠　　工

1984 年生．2007 年名古屋大学工学部物理工学科（応用物理学コース）卒業．動的計画法による最適化問題の求解アルゴリズムの研究開発に従事．

山本　有作（正会員）

1966 年生．1990 年東京大学工学部計数工学科（数理工学コース）卒業．1992 年同大学院工学系研究科物理工学専攻修士課程修了．同年（株）日立製作所中央研究所入所．2003 年名古屋大学大学院工学研究科計算理工学専攻助手．現在，同准教授．並列計算機向け行列計算アルゴリズムおよび金融工学向け高速計算アルゴリズムの研究開発に従事．高性能計算とその応用に興味を持つ．博士（工学）．SIAM，INFORMS，日本応用数理学会各会員．

畝山多加志

1980 年生．2003 年名古屋大学工学部物理工学科卒業．2005 年同大学院工学研究科計算理工学専攻修士課程修了．2008 年京都大学大学院理学研究科物理学・宇宙物理学専攻修了．現在，同大学化学研究所特定助教．高分子物理学の理論の開発およびシミュレーションに従事．高速計算アルゴリズムの物理シミュレーションへの応用に興味を持つ．日本物理学会，高分子学会，日本レオロジー学会各会員．

張　　紹良（正会員）

1990 年 3 月筑波大学大学院工学研究科博士課程修了．工学博士．現在名古屋大学大学院工学研究科計算理工学専攻教授．大規模行列計算における高速解法の開発および並列計算のアルゴリズムの研究に従事．日本応用数理学会，SIAM 各会員．