Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 8, Number 1, pages 60–67, January 2019

Fast Algorithm of Inventory Management using Dynamic Programming on GPU

Makoto Motoka*, Hiroki Tokura*, Koji Nakano*, Yasuaki Ito*
*Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—In this paper, we propose an efficient GPU implementation of computing the inventory management problems solved by dynamic programming. In existing GPU implementations, repeat synchronization is necessary, and the overhead of calling the kernel for this necessitates a delay in calculation time. In addition, another method with different calculation order was also implemented, and a hybrid implementation that combines two kinds was also implemented to speed up. Therefore, in the proposed GPU implementation, by preparing flags for managing calculation order for each element in the table, we calculated the problem with one kernel calling and speeded up. As an evaluation experiment, we implemented the proposed method in NVIDIA Tesla V100 and measured the execution time. As a result, the proposed method achieves up to 3.41 times as much as the existing GPU implementation, and up to 1094.57 times faster than the sequential CPU implementation.**

***Index Terms*—inventory management, dynamic programming, GPU, CUDA**

I. はじめに

在庫管理問題とは，定期的に補充される原料を，任意の制約条件を満たしつつ複数の倉庫に格納するというもので，数理計画法における組み合わせ最適化問題に該当する [1].

ある 1 種の原料を格納する K 個の倉庫があるとする. 各倉庫からは毎日それぞれ決められた量だけ原料が消費され，一方で原料は別に毎日各倉庫のいずれかに搬入され補給される. この一連の流れが N 日間行われる. 各倉庫内の原料の充填率は 0% ～ 100% で毎日推移し，これは毎日どの倉庫に原料を搬入し補給するのが良いかというペナルティ関数で評価することができる. この充填率の値が，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，各日の倉庫への補充先を計画したい. この問題を数学的に定式化するため，以下のように定数，変数を定義する. なお添え字の範囲は，指定のない場合，$1 \leq k \leq K$，$1 \leq n \leq N$ とする.

$f_k^{(n)}$　倉庫 k の n 日目 $(0 \leq n \leq N)$ 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 日目に搬入される補給量

$g_k(f_k)$ 充填率が f_k のときの倉庫 k のペナルティ関数

$j(n)$　n 日目の補給先の倉庫番号

ここで，$f_k^{(0)}, v_k, c_k^{(n)}, a^{(n)}$ は与えられる定数 (入力) とする. 充填率は $0 \leq f_k^n \leq 1$ の実数であり，関数 $g_k(f_k)$ は，$f_k = 0$(空)，$f_k = 1$(満杯) で大きな値を取り，$f_k = 0.5$ で最小値を取るように与える.$j(n)$ は独立変数である. また，$f_k^{(n)} (1 \leq n \leq N)$ は次の式により定まる.

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

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 $\delta_{k,j(n)}$ は，$j(n) = k$ のときのみ $\delta_{k,j(n)} = 1$ となり，それ以外の場合は $\delta_{k,j(n)} = 0$ となる. 本論文で扱う在庫管理問題は，制約式 (1) の下で，目的関数

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

を最小化する問題として定式化される.

図 1 に例として表 I の入力の場合の補給計画の例を示す. 日数 $N = 3$，倉庫数 $K = 3$ とする.

TABLE I
入力例

	倉庫 A	倉庫 B	倉庫 C
総容量	15t	20t	10t
初期充填率	0.6	0.4	0.5

日数	補給量	搬出量		
		倉庫 A	倉庫 B	倉庫 C
1 日目	3.0t	1.5t	2.0t	2.0t
2 日目	6.0t	3.0t	6.0t	3.0t
3 日目	3.0t	4.5t	4.0t	1.0t

II. 動的計画法の適用

制約付き最小化問題 (1)，(2) は独立変数 $\{j(n)\}_{k=1}^K$ が有限個の値のみをとる組み合わせ最適化問題である. 既存研究 [1] 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，動的計画法を適用した.

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時の n 日目以降の部分目的関数を

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

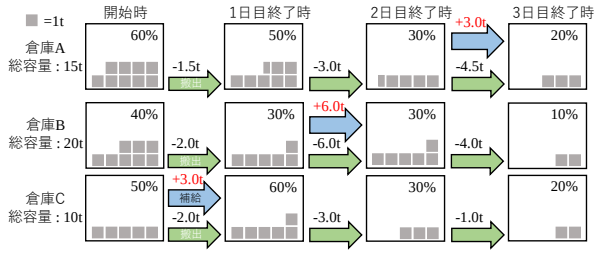

Fig. 1. 補給計画の例

と定義する. また，部分的な制約付き最小化問題

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

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

すると，式 (3) より明らかに，

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

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

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

を用いると，次の式が得られる.

$$\overline{G}^{(n)}(\{f_k^{(n-1)}\}_{k=1}^{K}) = \min_{j(n)}\left[\sum_{k=1}^{K} g_k(f_k^{(n)}) + \overline{G}^{(n+1)}(\{f_k^{(n)}\}_{k=1}^{K})\right] \quad (4)$$

この式は，もし $n+1$ 日目以降の部分目的関数 $\overline{G}^{(n+1)}(\{f_k^{(n)}\}_{k=1}^{K})$ 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 $\overline{G}^{(n)}(\{f_k^{(n-1)}\}_{k=1}^{K})$ 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 $j(n;\{f_k^{(n-1)}\}_{k=1}^{K})$ も求められる. そこで，終端条件 $\overline{G}^{(N+1)}(\{f_k^{(N)}\}_{k=1}^{K}) = 0$ から始め，式 (4) を用い $\overline{G}^{(n)}(\{f_k^{(n-1)}\}_{k=1}^{K})$ 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 $n = N, N-1, ..., 1$ と続けていけば，最終的に $\overline{G}^{(1)}(\{f_k^{(0)}\}_{k=1}^{K})$ 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**，$\overline{G}^{(n)}(\{f_k^{(n-1)}\}_{k=1}^{K})$ を**価値関数**と呼ぶ. また，この実装を後ろ向き実装と呼ぶ.

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.

$$\sum_{k=1}^{K} f_k^{(n)}v_k = \sum_{k=1}^{K} f_k^{(0)}v_k - \sum_{m=1}^{n}\sum_{k=1}^{K} c_k^{(n)} + \sum_{m=1}^{n} a^{(n)} \quad (5)$$

右辺は定数であることから，これは，$f_1^{(n)}, f_2^{(n)}, ..., f_K^{(n)}$ 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，この $K-1$ 次元空間を採用する.

以上で説明した動的計画法を用いてコンピュータ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，$[0,1]$ の連続値をある自然数 L で分割して $h = 1/L$ とし，各 f_k が離散値 $f_k = hl_k$ $(l_k = 0, 1, ..., L)$ のみをとるとする. また，

$$\overline{G}^{(n)}_{l_1,l_2,...,l_K} \equiv \overline{G}^{(n)}(\{hl_k\}_{k=1}^{K-1})$$

とおく. これを用いて式 (4) を書き換える.

ところが，いま $\{f_k^{(n-1)}\}_{k=1}^{K-1}$ が L 分割した $[0,1]$ 空間の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 $\{f_k^{(n)}\}_{k=1}^{K-1}$ が同様に格子点上にあるとは限らない. この場合，式 (4) における $\overline{G}^{(n+1)}$ の値が求められないという問題が生じる.

そこで，$\overline{G}^{(n+1)}$ の計算にあたっては，$K-1$ 次元空間の各点 $f_1^{(n)}, f_2^{(n)}, ..., f_{K-1}^{(n)}$ に対して $floor$ 関数を用い，L 分割した $[0,1]$ 空間の格子点上へと下に丸め込むことによって補間を行う. 以下のアルゴリズムの記述では，$\overline{G}^{(n+1)}(\{f_k\}_{k=1}^{K-1})$ の値を補間により求める関数を

$$Interpolate(\overline{G}^{(n+1)}, f_1, f_2, ..., f_{K-1})$$

と書く.

以上の定義のもとに，動的計画法を適用した在庫管理問題の解法をアルゴリズムとして書くと Algorithm1 のようになる. アルゴリズム中では簡単化のため $K=4$ としている

図 2 は出力が計算される際の参照先を順に示している. 倉庫数 $K=3$, 日数 $N=3$ の例である. テーブルの縦軸は日数であり，横軸は各倉庫の充填率の組み合わせとなる. 本来は日数と倉庫数で K 次元のテーブルになるはずだが，簡易化のため 2 次元のテーブルで示す. 左図は 3 行目の 1 要素の計算の際の参照を示しており, 右図は全要素が最終日から順に計算されることを示している.

また以下に，本アルゴリズムの入力と出力を記述する.

入力　$K, N, L, f_k^{(0)}, v_k, c_k^{(n)}, a^{(n)}$

出力　$minG(\{f_k^{(0)}\}_{k=1}^{K};\{j(n)\}_{k=1}^{K}), j(n)$

Algorithm 1 在庫管理問題の解法

```
for l_1 ← 0 to L do
  for l_2 ← 0 to L do
    for l_3 ← 0 to L do
      G^(N+1)_{l_1,l_2,l_3} = 0
    end for
  end for
end for
for n ← N to 1 do
  for l_1 ← 0 to L do
    for l_2 ← 0 to L do
      for l_3 ← 0 to L do
        G^(n)_{l_1,l_2,l_3} = +∞
        for j ← 1 to 4 do
          G_temp = 0
          for k ← 1 to 3 do
            f_k = (h l_k v_k − c_k^(n) + a(n)δ_kj)/v_k
            G_temp = G_temp + g_k(f_k)
          end for
          式 (5) により f_4 を計算
          G_temp = G_temp + g_4(f_4)
          G_temp = G_temp + Interpolate(G^(n+1), f_1, f_2, f_3)
          if G_temp < G^(n)_{l_1,l_2,l_3} then
            G^(n)_{l_1,l_2,l_3} = G_temp
            j(n, l_1, l_2, l_3) = j
          end if
        end for
      end for
    end for
  end for
end for
```

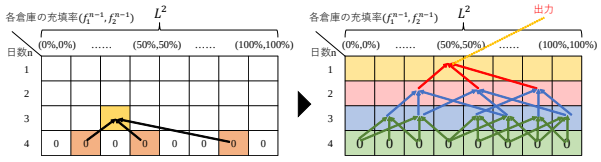

Fig. 2. 出力を計算する際のテーブル $\overline{G}$ の参照例

III. CUDA と GPU

本節では GPU と CUDA アーキテクチャについて述べる. GPU は複数の Streaming Multiprocessor(SM) と Global Memory と Shared Memory による階層的なメモリ構造を持っている [2].SM は GPU に複数搭載される演算ユニットでそれぞれ独立に演算を行い，多数の core と Shared Memory によって構成される.Global Memory は GPU 内の全ての core からアクセス可能なメモリである.SharedMemory と比べて容量は大きいがメモリアクセスに時間がかかる.Shared Memory は同一 SM 内にある core のみからアクセス可能なメモリである.Global Memory と比べて容量は小さいが高速なアクセスが可能となっている. これらの特性から頻繁にアクセスするデータを Shared Memory に格納することによって GPU プログラムの高速化につなげることができる. また，各 SM 内にはキャッシュ空間が存在する.GPU コンピューティングでは CPU と GPU の両方を用いて計算を行う. このとき CPU に実行させるプログラムをホストプログラム，GPU に実行させるプログラムをカーネルと呼ぶ. カーネルは C 関数のように記述することが可能で，ホストプログラムから呼び出すことで実行される. また，CPU と GPU のメモリは独立しており，データの転送 (cudaMemcpy) が必要である.

CUDA とは NVIDIA 社の提供する GPU のための統合開発環境である [2].C 言語を拡張した開発言語となっていて，GPU を用いた並列演算プログラムを容易に記述することができる.CUDA ソフトウェアアーキテクチャは階層的な構造となっている. プログラムの最小実行単位である thread，thread の集合である block，block の集合である Grid によって構成される (図 3). この thread，block，Grid はハードウェアアーキテクチャの core，SM，GPU にそれぞれ対応している. また 32 個の thread をまとめたものを warp と呼び，同一 warp 内の thread は同じ処理を同時に行う.GPU による演算は同一 warp 内の thread 以外は非同期なものとなっている. よって，thread 間で依存関係のある計算を実行するためには同期命令を呼び出すことで block 内の thread で同期をとる必要がある. 異なる block 間で同期をとる場合は 1 度カーネルを終了させ新しいカーネルを起動する必要がある. これらの同期はオーバーヘッドを伴うので処理の遅延の原因となる.

Device での thread 実行は stream 単位で管理されている.GPU のカーネル実行や cudaMemcpy も 1 つの stream である. 複数の stream を管理することで, 複数の kernel や cudaMemcpy を並列に実行可能である.

IV. 既存 GPU 実装

A. 概要

本章では，動的計画法を用いた在庫管理問題の既存 GPU 実装 [5] についての説明を行う. カーネルを呼び出す際には，block 数 B と thread 数 T という 2 つのパラメータを指定する. これによりカーネルは，それぞれが T 個の thread を持つ B 個の block により並列実行される. 各 thread は，自分の block 番号，block 内の thread 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.

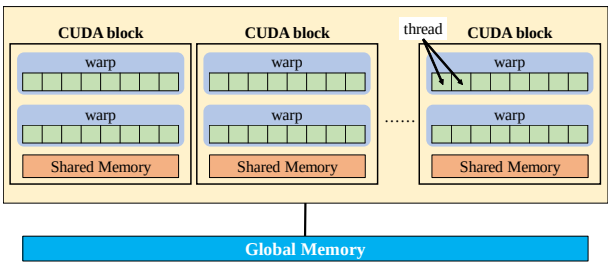

Fig. 3. GPU アーキテクチャ

B. GPU での実行部分

既存実装では，n に関する一回の繰り返しを実装する. つまり，配列 $\overline{G}^{(n+1)}$ の値を入力として配列 $\overline{G}^{(n)}$ の値を計算する部分を 1 個のカーネルとして実装する.$n = N, N-1, ..., 1$ の順に価値関数を求める部分の計算においては，この関数を N 回呼び出しアルゴリズム全体の計算を行う.

C. 各変数のメモリ空間への割り当て

前節で定めたカーネルを GPU で実行するにあたり，その中の各変数をどのメモリ空間に割り当てるかを決定する.2 章で各メモリ空間の特性に基づき，次のような割り当てを行った.

- 配列 $\overline{G}^{(n+1)}$，$\overline{G}^{(n)}$ は全 thread からアクセスする必要がある. また，$\overline{G}^{(n)}$ は各カーネルの終了時から次のカーネル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 Global Memory 上に置く. また，配列 $\overline{G}^{(n)}$ を計算する際に配列 $\overline{G}^{(n+2)}$ 以降の値は必要ではないため保持しない. そのため $\overline{G}^{(n-1)}$ の値は $\overline{G}^{(n+1)}$ の保存してあった領域に書き込まれ、配列 $\overline{G}$ は図 4 でいう 2 行分の領域のみがメモリ空間に必要である.
- 配列 j は CPU に転送する必要があるため Global Memory 上に置く.
- $a^{(n)}$，$c_k^{(n)}$，v_k は読み込み専用の値であり，かつ小容量なため，キャッシュによる高速アクセスが可能である.
- thread ごとに利用される変数はレジスタに置く.

D. block と thread による並列化

本論文のアルゴリズムは，$(K-1)$ 重の完全並列ループにより構成される. そこで本節では倉庫数 $K = 5$ と仮定した場合の実装を考える. 各カーネルは l_1, l_2, l_3, l_4 に関する 4 重のループで構成されるため，これらを block 及び thread に割り当てていく.

まず thread の並列化については，レジスタ数の制限の範囲で thread をできるだけ多く取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 thread 群がアクセスするメモリ領域が連続になるよう，l_3, l_4 の 2 つの添え字を thread 並列化に用いることにした. ただし，ここでは添え字 l_4 が変化するとき配列が連続になるように実装を行ったと仮定する. 具体的には block 内の thred 数を $T = blockDim.x * blockDim.y$ として 2 次元サイクリック分割を行い，添え字 l_3, l_4 を持つ要素が，番号

$$mod(l_3, blockDim.y) \times blockDim.x + mod(l_4, blockDim.x)$$

番の thread に割り当てられるようにする.

block に関する並列化も同様に，添え字 l_1, l_2 についての 2 次元サイクリック分割を用いて行う. また，パラメータ K の値が増えた際のループについては，各 thread について逐次的に実行する. 図 4 は既存実装における block と thread の割り当てにおける概略図である.

E. CPU と GPU 間のデータ転送

Algorithm 1 からわかる通り，カーネル内では，CPU から GPU に転送が必要なデータは $a^{(n)}$,$c_k^{(n)}$,v_k, L などの少数のパラメータのみである. 計算結果の配列 $\overline{G}^{(n)}$ は次のカーネルの呼び出しで $\overline{G}^{(n+1)}$ として用いるが，CPU に転送する必要はない. 全カーネル終了時に配列 $\overline{G}^{(1)}$ 内の 1 要素を初期充填率に対する目的関数の最小値として CPU に転送するが，転送時間はごく少量である. また，配列 j は CPU に転送する必要があるが，目的関数の最小値のみが必要な問題設定の場合は必要ではない.

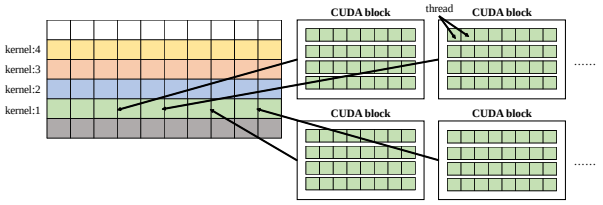

Fig. 4. 計算の割り当て

V. 提案 GPU 実装

A. 後ろ向き GPU 実装

1) 概要*:* 既存実装ではアルゴリズム全体を計算するために N 個のカーネルの起動が必要であった. 配列 $\overline{G}^{(n)}$ の計算に配列 $\overline{G}^{(n+1)}$ の値が必要であり，カーネルを切り替えることによって GPU 全体の同期を行っていたためである. しかし定期的な GPU 全体の同期は，GPU 内の thread の多くを休ませてしまうことになり，高速化の妨げになっていた.

提案実装では，アルゴリズム全体の実装を 1 カーネルで行う. 配列 $\overline{G}$ の各値の計算が終了しているかをグローバルメモリ内のフラグテーブルで管理することで，計算順制御を行い，GPU 全体の同期を避けることが可能である.

2) 配列 $\overline{G}$ 全体のフラグ管理*:* Algorithm 1 からわかるように，配列 $\overline{G}^{(n)}$ の 1 要素の計算には，配列 $\overline{G}^{(n+1)}$ の K 要素の値の参照が必要である. そのため既存実装ではアルゴリズム全体の実装のために N 回のカーネルの起動を行っていた. しかし本手法では，配列 $\overline{G}$ 全体の各要素の計算が終わっているかいないかをフラグテーブルを用いて管理する. 全要素の完了フラグをアルゴリズム実行の前に 0 で初期化し，実行の際各要素の計算が終わるとフラグに 1 を立てる. そして参照の際には，配列 $\overline{G}^{(n)}$ の 1 要素の計算を担当する thread は参照要素の計算が終わるまで待つような実装を行う. これにより計算順制御を行うことができ，配列 $\overline{G}$ の全要素を GPU 全体の同期なく計算可能である.

また，フラグテーブルは配列 j 内に格納することができる. 配列 j 内の値は適切な倉庫の選択先であり，$0, ..., K-1$ までの値しか入りえない. したがって，配列 j を char 型で実装した場合でも余分な bit が存在している. よって，配列 j の各値の最上位 bit をフラグとして活用することで，フラグテーブルを別の配列に用意することなく実装可能である.

3) カウンタを用いた *warp* 単位の割り当て*:* 本実装では，配列 $\overline{G}$ を計算する際 warp 単位での割り当てを行う. 具体的には，配列 $\overline{G}$ 全体を 32 要素毎に partition で区切る. 配列 $\overline{G}$ は 1 次元配列で実装し，配列の最後尾から 32 要素ずつ分割を行い，余った要素も 1partition としてカウントする. そして区切られた各 partition を 32-thread の 1warp で処理を行い，各 warp は担当 partition の処理が終わると次の partition の処理に移る. 各 warp が SM に割り当てられた順に partition の処理に移るために，また個別の ID がな

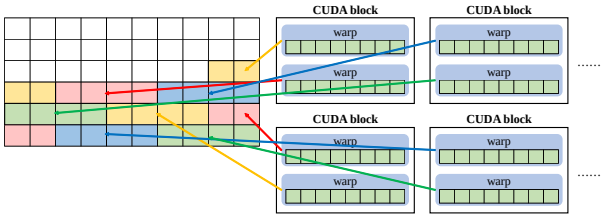

Fig. 5. warp 単位の割り当て

い warp の処理する箇所を一意に指定するために，Global Memory 上にカウンタを実装し割り当てを行う [3]. 以後，このカウンタをグローバルカウンタと呼ぶ. カウンタは初めに 0 として初期化しておき，各ワープの先頭 thread が atomic 演算を用いて値を 1 ずつ加算し，加算する前の値を返り値としてレジスタに保存する. その後同じ warp 内の全 thread に warp シャッフルを用いて値を伝搬することで，各 warp 内の thread は一意に担当 partition の処理を行うことができる. 図 5 は，warp の割り当て方を図にしたものである. 各区間を 1partition とし,1warp が担当する.

また，配列 $\overline{G}$ を分割する際，32 要素ごとの分割ではなく，64 要素や 128 要素など 32 の倍数での分割にすることができる. これにより一回の warp の処理要素数を増やすことで，atomic 演算の回数を減らし若干の高速化が望める.6 章の実験では 1warp の処理要素を 64 要素で固定して計測を行った.

また，配列 $\overline{G}$ の各要素を参照する際には，Index の値の計算が毎回必要である. しかし配列 $\overline{G}$ は 1 次元配列で実装を行っているため，配列の要素数が多くなると Index の計算が int 型では行えなくなる. その際にはポインタを用いるように実装が切り替わるようにした.

また，グローバルカウンタを用いずに各 partition の割り当てを行うと，warp の起動順序は SM により決められプログラマにはわからないため，先に計算が必要な領域に warp が割り当てられずに，他の領域に割り当てられた warp 全てが計算待ち状態になってしまいデッドロックが起こってしまう可能性がある.warp の割り当てをグローバルカウンタを用いることにより，先に計算が必要な領域に優先的に SM が warp を割り当てるようになりデッドロックを避けることができる. また，warp 全体の計算待ちの時間が少なくなるため高速化にもなる.

4) メモリ割り当て*:* $a^{(n)}$，$c_k^{(n)}$，v_k はコンスタントメモリに置いたあと，各 block で Shared Memory 内に格納する. この作業はカーネルの起動直後に行うが，既存実装ではカーネルを定期的に起動し直す必要があったため，カーネル起動のたびに Shared Memory への格納処理が入りこの戦略はあまり有効でなかった. しかし本実装は 1 カーネルの実装であり，一度格納が終わるとアルゴリズムの実装が終了するまで Shared Memory を使い続けることができ効率的である. また，4 章で述べた配列 $\overline{G}$ については，warp の処理が 2 行にまたがってしまうことがあるため 3 行分をメモリに保持するとする.

B. 前向き実装

これまでの実装では，$n = N, N-1, ..., 1$ の順に価値関数を求める計算を行ってきた。この章では，$n = 1, 2, ..., N$ の順に価値関数を求める，前向き実装を提案する.

前向き実装では，後ろ向き実装と同様のサイズの動的計画法テーブル（DP テーブル）G を用意する。テーブルの縦軸は日数であり，横軸は各倉庫の充填率の組み合わせとなる. この DP テーブルを $n = 1, 2, ..., N$ の順に埋めていき，一日毎の価値関数を合計していく. テーブル G に，日数と各倉庫の充填率の組み合わせを与えた場合，初期充填率からその組み合わせに至る際の価値関数の合計が得られる. よって, 最終的にテーブルの $N+1$ 行目に，初期充填率から，様々なパターンで補給を行った場合の価値関数の合計が 1 要素ずつ現れるため，その最小値を取り，解とする. また，一日毎の価値関数を求める際に，その際に補給した倉庫の番号を別のテーブル J の同アドレスに格納する. よって，最後に $n = N, N-1, ..., 1$ の順で最適解の場合の各日の倉庫への補給先も逆算で求められる.

この実装では，後ろ向き実装と比べて計算回数の上で利点がある. 後ろ向き実装では，DP テーブルの全要素の計算が行われる. よって，入力からはなり得ない日数と充填率の組み合わせの場合も，価値関数の計算は行われる. しかし前向き実装では，初期充填率から実際に推移しうる日数と充填率の組み合わせの場合でのみ価値関数の計算を行う. そのため大幅に計算回数の減少が見込める. ただし，計算を行うか行わないかはテーブル全体のフラグ管理で決定する. このフラグを計算決定フラグと呼ぶ.

1) アルゴリズム*:* 動的計画法を適用した在庫管理問題の前向き実装をアルゴリズムとして書くと Algorithm2 のようになる. アルゴリズム中では簡単化のため $K = 4$ としている. また，テーブル全体の初期化は省略している. これを基に説明を行う

まず，DP テーブル G の値は十分に大きな値で初期化される. 同様に，倉庫の補給先を格納するテーブル J，計算決定フラグを格納するテーブル F は 0 で初期化する. その後，1 行目の初期充填率の組み合わせに該当する要素に計算フラグ 1 を立て，テーブル G の該当要素は 0 とする.

n 行目の 1 要素の計算としては，2 の 8 行目から 25 行目が相当する. まず計算決定フラグを参照し，1 が立っている要素のみ計算が続けて行われる. 次にそのアドレスに格納されている目的関数の値を読み込む. その後，担当する充填率の組み合わせから k 番目の倉庫に補給した場合の目的関数の値を計算し $(k = 1, ..., K)$，読み込んだ目的関数の値と足し合わせ，$n+1$ 行目の要素に書き込む. 書き込まれる要素のアドレスは，先ほど補給した際に推移した各倉庫の充填率の組み合わせにより求められる. 同様に，同アドレスのテーブル J にはその際の補給先倉庫番号を，テーブル F には計算決定フラグを立てる.

この 1 要素の計算をテーブル全体に渡って $n = 1, 2, ..., N$ の順に実行する. これにより 1 行目の 1 要素 (初期充填率の組み合わせ) からのみ計算決定フラグと目的関数の値が伝搬していき，実際に到達しうる充填率の組み合わせでのみ計算が行われる.

最後にテーブルの $N+1$ 行目の要素から目的関数の最小値をとり出力とする. それと同時に, 最適な倉庫の補給先を

Algorithm 2 前向き実装での解法

```
l_1, l_2, l_3 = {f_k^(0)}_{k=1}^{K-1}
G^(1)_{l_1,l_2,l_3} = 0
F^(1)_{l_1,l_2,l_3} = 1
for n ← 1 to N do
  for l_1 ← 0 to L do
    for l_2 ← 0 to L do
      for l_3 ← 0 to L do
        if F^(n)_{l_1,l_2,l_3} == 1 then
          G_temp = G^(n)_{l_1,l_2,l_3}
          for j ← 1 to 4 do
            G_temp2 = 0
            for k ← 1 to 3 do
              f_k = (h l_k v_k − c_k^(n) + a(n)δ_kj)/v_k
              G_temp2 = G_temp2 + g_k(f_k)
            end for
            式 (5) により f_4 を計算
            G_temp2 = G_temp2 + g_4(f_4)
            G_temp2 = G_temp + G_temp2
            if G_temp2 < G^(n+1)_{l_1,l_2,l_3} then
              G^(n+1)_{l_1,l_2,l_3} = G_temp2
              J(n+1, l_1, l_2, l_3) = j
              F(n+1, l_1, l_2, l_3) = 1
            end if
          end for
        end if
      end for
    end for
  end for
end for
```

逆順に計算し出力とする. これらの処理は Algorithm2 では省略している.

このアルゴリズムでは一回で，1 要素から最大 K 要素への値の伝搬 (書き込み) が行われる. よって複数の n 行目の要素から $n+1$ 行目の 1 要素に書き込みが行われる場合がある. その場合，目的関数の値は最も小さいものを書き込む必要がある. この処理は逐次実行では問題にならないが，並列実装では書き込みの競合が起こってしまう. 次の節で説明するが，その際には排他的な書き込みと計算が必要である.

また, 図 6 は前向き実装について説明している. 矢印は計算後の書き込み要素を表しており, 色で塗られた部分はフラグが立った計算される要素を順に示している.

2) GPU 実装: 本節では，前向き実装での GPU 実装について述べる. これまでの実装のように，DP テーブルにおいて n 行目の計算を行うためには，その前の行である $n-1$ 行目の計算が必要である. 実際には，n 行目の 1 要素の計算に必要な要素は倉庫数である K 要素である.n 行目の 1 要素が担当する倉庫の充填率の組み合わせに，$n-1$ 行目で倉庫 1 に補給して到達する場合，倉庫 2 に補給して到達する場合... と K 通りの組み合わせが考えられる. そのため，毎回 $n-1$ 行目の K 要素に対してアドレスを計算して計算が完了しているかのルックバックを行えば，各行間で計算の同期を取らずに 1 カーネルでの実装が行える.

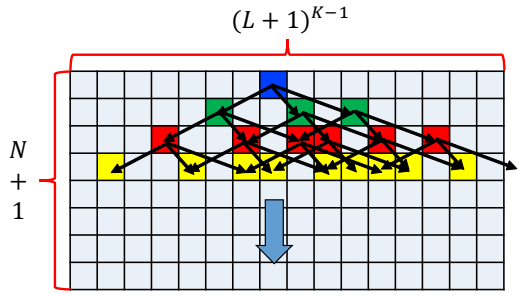

Fig. 6. 前向き実装

しかし本論文では，既存論文の手法のように行間同期をとる実装を行う. 後ろ向き実装では，フラグのルックバックは目的関数の読み込み操作のついでに行えて，新たにアドレス計算を行う必要はなかった. 前向き実装ではフラグのルックバックに前の行の要素のアドレス計算を行う必要がある. この操作とフラグルックバック自体のリードの遅延が合わさって，カーネル同期を伴う実装より遅くなってしまうためである.

カーネル同期を伴う実装について，説明を行う.1 カーネルでは 1 行分の計算が行われる. このカーネルを N 回起動することによりテーブル全体の計算が行われる. 各行内では計算の依存要素がないため並列実装が可能である.

しかし，書き込みの競合については対策が必要である. 前節で述べたように，n 行目の要素には最大 K 要素からの書き込み処理が $n-1$ 行目から行われる. その際には，より小さい目的関数の値だけではなく，補給された倉庫番号も書き込まなければいけない. つまり，2 要素の書き込みがクリティカルであり，通常の atomic 演算では対応できない. このため本実装では，目的関数を float 型 32bit のうち上位 29bit に格納するとともに，下位 3bit に補給先の倉庫番号 j を格納する. そして書き込み処理の際にはその 3bit を含んだまま atomicmin 演算を行う. この処理により高速かつ確実な競合対策が行える. ただし目的関数の値を扱う際には，下位 3bit の情報が損なわれないようにするため，計算のたびに上位 29bit をマスクで取り出す. そして bit 演算を行うために共用体を用いる.

C. *hybrid* 実装

本章では GPU の stream を用いた，前向き実装と後ろ向き実装のハイブリッドな実装について説明を行う.

通常，GPU では default stream と呼ばれる stream のみでカーネルの処理や GPU メモリへの処理を行う. この stream を複数用意し，stream1 で前向き実装を，stream2 で後ろ向き実装を行うように設計を行った. これにより例えば，前向き実装でカーネル同期をとっている時間の計算リソースを後ろ向き実装に回したりということができる.DP テーブル N 行の要素のうち，何行目までを前向き実装で行い，残りを後ろ向き実装で行うかはパラメータによって (パラメータ名を ReturnN とする) 決定可能である. 次章の計算機実験では，その ReturnN を変化させ，一番実行が早かったものを結果として載せている.

前向き実装では 1 行目から順に目的関数の値を合計して伝搬していく. 逆に後ろ向き実装では $N+1$ 行目から順に目

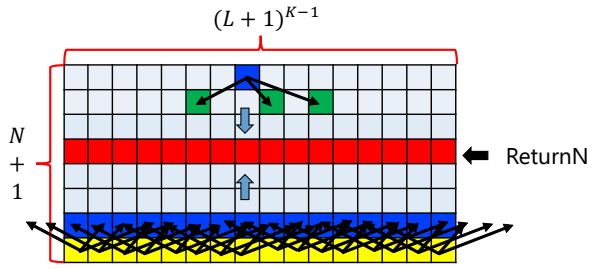

Fig. 7. hybrid 実装

的関数の値を伝搬していく. よって ReturnN 行目で双方から伝搬される目的関数の値を加算するために一度両 stream で同期をとる必要がある. その部分だけは別にカーネルを用意して値の結合処理を行う必要がある.

図 7 は hybrid 実装のイメージを示しており,ReturnN 行目以前まで前向き実装で処理し, その ReturnN 行目から後ろ向き実装で処理する.

VI. 計算機実験

本節では NVIDIA Tesla V100 (CUDAcore:5120, ブーストクロック:1.370GHz) [4] を用いて実装を行い，評価実験を行う. 入力として，倉庫数 K を $K=5$ で固定し，日数を $N=30,60$，分割数を $L=31,63$ と変化させ計測を行っている. ここで本実験では，既存実装の起動 block 数，1block 当たりの起動 thread 数はそれぞれ $(L+1)*(L+1)$block, $32*32$thread とした. これにより，$L+1$ が 32 の倍数の時 2 次元サイクリック分割による thread 並列化に望ましい.1warp:32thread が連続したメモリにアクセスでき，かつ block 内の thread が余すことなく 32 要素ずつの処理を担当できるからである.

提案実装について，起動 block 数はオキュパンシを考慮し，GPU に同時に割り当てられる最大の block 数とした. 本実験で使用した GPU である NVIDIA Tesla V100 は SM を 80 基搭載しており，各 SM には最大 64warp 割り当てることができる.

また，逐次処理との比較のため CPU による後ろ向き実装の実行時間も測定した.CPU による計算は，提案実装の手法と同様に配列を 1 次元で実装し，配列の最後尾から 1 要素ずつ連続にアクセスし処理を行う. 逐次実装においても thread が連続したメモリ内の要素にアクセスすることは重要で，そうでない場合とは実行時間の差が出てくる. また，Algorithm1 における l_1 から l_3 の 3 重ループは複合させて 1 重のループとして実装した.CPU は Intel(R) Xeon(R) CPU E7-8870 v4(2.10GHz) のものを使用した.

さらに，並列計算の比較材料として OpenMP [5] を用いたマルチスレッドによる実装も行った. こちらでは先ほどの Intel(R) Xeon(R) CPU E7-8870 v4 を 4 基用いた.Intel(R) Xeon(R) CPU E7-8870 v4 には 1 基あたり 20 個の core が用意されており，ハイパースレッディングにより 1core あたり 2thread の並列実装が行える. よって，$4*20*2=160$ より 160thread での実装を比較実験として行った.

また前向き実装では，後ろ向き実装と異なり，計算時間が異なる. 実際に入力から到達しうる倉庫充填率の組み合わせを持つ要素のみを計算するため，入力が複雑になり DP テーブル内の計算要素が増えると実行時間が遅くなる. 今回は，DP テーブルの計算要素がほとんどない疎な場合と，大部分を計算しなくてはならない密な場合の 2 ケースについて計測した.

同様に hybrid 実装についても，上の 2 ケースについて計測した.hybrid 実装については中継地点のパラメータ ReturnN を変化させ，一番早かったものを用いた.

表 II は以上の 6 実装を先ほど述べた各パラメータで実装した際の実行時間である. 提案後ろ向き GPU 実装は既存の実装と比べて最大 2.56 倍の高速化を達成した. しかし，分割数 L のパラメータを増やし配列サイズが大きくなると高速化率は低下する. 原因として atomic 関数のコストが考えられる. テーブルのサイズが大きくなると，グローバルカウンタをインクリメントする回数が多くなるためである. また逐次処理の CPU 実装と比較して最大 1094.57 倍の高速化を達成した.

DP テーブルの計算要素がほとんどない疎な場合では，前向き実装は後ろ向き実装と比較して当然高速である. しかし密な場合では，競合処理などを含んだ前向き実装が低速になる.

hybrid 実装は，計算要素が疎な場合最も高速である.$L=31$ の場合，テーブルの横幅が小さく，前向き実装は 1 カーネル内で計算リソースが余ってしまう. そのリソースを後ろ向き実装にも回すことで高速になる. この時，DP テーブルのほとんどが前向き実装で行われる. 逆に計算要素が密な場合，前向き実装の計算時間と後ろ向き実装の計算時間がほぼ等しくなるように ReturnN を定めたほうが効率が良くなる. この二つがうまくオーバーラップし高速となる.

図 8 は日数 N，倉庫数 K をそれぞれ $N=90, K=5$ で固定し，パラメータ L を $L=30,40,50,60,70,80,90$ と増やしていった場合の実行時間の変化を示している. 全ての提案後ろ向き実装の実行時間は既存実装の実行時間に対して高速である. パラメータ N,K を変えた場合においても、実行時間は同様の結果が得られた. さらに，提案後ろ向き実装の実行時間は L の値が増えるにつれて関数的に増加しているが，既存実装における実行時間は L の値の増加による増加幅が一定ではない. これは，既存実装がパラメータ L の値によってはアクティブな thread を余らせてしまう実装だからである. 提案後ろ向き実装ではパラメータ L の値によってアクティブな thread が余ることはないので有用である.

TABLE II
$K=5, N=30,60$ の場合の分割数 L の違いによる実行時間 [MS] と GPU:既存実装/提案実装の比較倍率

		N=30		N=60	
L		31	63	31	63
CPU	1thread	3064.11	47695.88	6166.9	95435.3
	160thread	219.3	1053.08	463.01	2083.91
GPU	既存実装	4.24	43.93	7.59	87.19
	後ろ向き実装	1.67	23.02	2.96	46.04
	前向き実装 (疎)	1.28	12.87	2.61	25.77
	前向き実装 (密)	1.8	14.4	4.01	36.74
	hybrid 実装 (疎)	1.11	14.24	2.12	26.85
	hybrid 実装 (密)	1.49	15.56	3.37	38.38
後ろ向き GPU 高速化率		2.54	1.91	2.56	1.89
前向き GPU 高速化率 (疎)		3.31	3.41	2.91	3.38

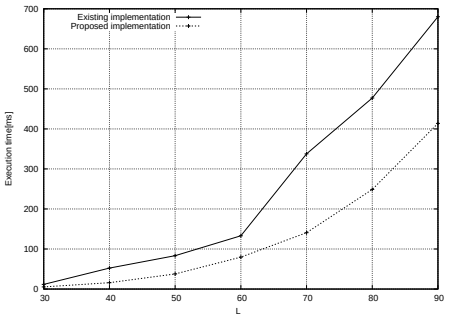

Fig. 8. $K = 5, N = 90$ での分割数 L の違いによる実行時間

VII. 結論

本論文では在庫管理問題の効率的な GPU 実装を提案した. 既存実装では定期的に同期をとるため複数のカーネルで計算を行っていたが、提案後ろ向き GPU 実装では要素ごとに計算が完了したかを示すフラグを用意することで、単一カーネルのみで在庫管理問題の計算を行い高速化を実現した. また，計算順序の異なる前向き実装を実装し，二つの実装を組み合わせた hybrid 実装も行った. 評価実験では NVIDIA Tesla V100 に実装した結果，提案した実装方法ではいずれのパラメータに対しても速い実行時間を示し，既存の GPU 実装と比べて最大 3.41 倍，逐次処理の CPU 実装と比較して最大 1094.57 倍の高速化を達成した.

References

[1] 李天，河畠工，山本有作，畝山多加志，張紹良：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，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
[2] NVIDIA Corporation : NVIDIA CUDA C *programming guide version* 9.0， 2018
[3] Duane Merrill，Michael Garland : Single pass Parallel Prefix Scan with Decoupled Look back，NVIDIA Technical Report NVR-2016-002， Mar 2016. CUB v1.0.1， Aug 2013
[4] NVIDIA TESLA V100: https://www.nvidia.com/en-us/data-center/tesla-v100/
[5] OpenMP. http://www.openmp.org/