

連続画面配信における GPU および並列演算による画像差分圧縮の高速化

Performance Improvement of Differential Image Compression for Screen Distribution System using GPU and Parallel Computing

深井 裕二* 河合 洋明* 仲野 修*

Yuji Fukai, Hiroaki Kawai and Osamu Nakano

Abstract

The use of a screen distribution system for simultaneous distribution of the operation screen via a network is an effective approach of explaining software operations during information education lessons. In this case, it is important to enhance the reproducibility of the image and restrict time lag to avoid causing a sense of discomfort or misunderstanding to those who are attending the class. Achieving a minimal CPU load is also desirable in order to avoid hindrances to practical training. We attempted to enhance efficiency and processing speed and reduce CPU usage via differential image compression techniques by utilizing GPU processing capabilities based on DirectX and parallel computing. In this study, we report on our implementation of the abovementioned techniques and the performance of the system using these techniques.

1. はじめに

情報教育の授業では、ソフトウェア活用スキル習得のために、実際の操作学習が欠かせない。ソフトウェアの GUI (Graphical User Interface) は、直感的な操作誘導や状態認識によって操作効率を高める役割がある。近年の GUI やソフトウェアの表示では、画面要素のスライド、グロー、フェードといった動きの効果が用いられる。それらのデザインは、利用者の満足や経験を指すユーザエクスペリエンス (UX: User Experience, ISO9241-210) ⁽¹⁾ を高めるものでもあり、操作意欲や学習意欲に関与するものであると考えられる。

授業でプロジェクトを用いたソフトウェア操作説明をする際、大人数教室では視認性低下の問題が深刻である。GUI のフォントは 9pt 程度と小さく、作図や描画などのグラフィカルな操作も詳細部分は視認性が低い。この問題解決として、教員の PC 画面を受講者 PC に一斉ネットワーク配信する画面配信システムが挙げられる。しかし、GUI や画面要素の滑らかで素早い動きに対し、再現性や遅延および CPU 負荷などのシステム性能が悪ければ、違和感や誤解、ストレスなどで学習

の意欲や効率を低下させる可能性がある。

本研究では、授業に用いる画面配信システムに対し、受講者へのストレスを抑えるための高性能化を目的とし、GPU (Graphics Processing Unit) の処理能力を活用したシステムの実装法を提案する。本システムの高性能化にあたり、画面画像の差分圧縮を処理の基本とし、GPU の処理性能を活用できる開発ライブラリである Direct3D11 (DirectX11) ⁽²⁾ および並列演算アルゴリズムを適用する。本稿では、それらのシステム開発実践および得られた性能について報告する。

2. 関連研究と技術背景

授業向けの画面配信には、遠隔 PC 操作ツールを活用した画面共有⁽³⁾やデスクトップ配信システム⁽⁴⁾がある。これらは、受講者数の増加に対し、複数のサーバやクライアントによる中継でトラフィックを抑えているが、そのため遅延時間の不均衡な増加が予想される。一方、遠隔授業では、学内遠隔教育⁽⁵⁾の画面共有活用例もある。一般に、遠隔授業システムは、導入コストや特別な機器を必要とし、多様な入力映像用に、圧縮率の高い非

* 北海道科学大学高等教育支援センター学士課程教育支援部門

可逆圧縮を用いるため、画面要素や文字などの劣化が目立ち、圧縮と展開の負荷も大きい。また、フレームレートは 30fps 以下で遅延は数百 ms～数 s 以上と、遠隔授業では十分な性能であっても、対面授業では不向きな場合がある。対話向き映像配信⁽⁶⁾では、低遅延とされる既存システムの遅延測定結果は 233ms である。また、聴覚と視覚が同時と感じる時間差⁽⁷⁾は約 100ms までとされている。画面上の動きに対し、配信での再現性低下や、操作説明における発話と配信画面のタイムラグは、受講時のストレスとなる可能性がある。

北海道科学大学では、授業に使用する学生 PC に、大学や社会で使用実績が高い Windows プラットフォームを採用している。その開発元である Microsoft 社は、高速グラフィックス API (Application Programming Interface) として、DirectX を提供し、GPU の著しい進歩に対応し発展させている。DirectX では、直接的なハードウェア制御により、GPU の高速な処理能力が利用できる。

GPU は特に 3D グラフィックスの高速処理のために、多数の演算ユニットで並列演算を行う。この演算能力を汎用的に利用する技術として、GPGPU (General-Purpose computing on Graphics Processing Units) がある。DirectX は GPGPU 用に演算シェーダー (CS : Compute Shader) ⁽⁸⁾ 機能を用意している。CS は GPU をプログラムで動作させるプログラマブルシェーダーであり、Intel, AMD, NVIDIA など主要メーカーの GPU に対応し利用環境が多い。DirectX によるアプリケーション開発は、グラフィックス処理と GPGPU による並列演算の連携が容易であり、画面画像の処理に対し効率や性能の向上が期待できる。

GPU による並列プログラミングモデルは、SPMD (Single Program Multiple Data) モデルに該当し、一つのプログラムで複数のデータを並列に処理する。これは、画像処理における大量の画素に、同一演算を施す処理に適している。CS では、複数のスレッドおよび複数のスレッドグループによる 2 段階の並列処理構造を持ち、本システムのように画像データの画素単位およびブロック単位の処理効率を高める際にも有効である。

本研究で開発する画面配信システムの重要な性能として以下が挙げられる。これらの性能を高めるために、既存システム^(3,4)と異なり、DirectX を用いた開発技法および CS を用いた並列演算手

法によって本システムを実装する。

- (1) 画面変化に対する高フレームレートと無劣化画像配信による高い再現性。
- (2) 対面授業での発話とのタイムラグによる違和感をなくす低遅延。
- (3) 配信対象となるソフトウェアの稼働や操作に影響を与えない低 CPU 負荷。

3. 画面配信システムの概要と環境

本システムは、教員の PC 画面を受講者の PC 画面へネットワーク経由で配信する授業支援システムであり、図 1 に実行画面を示す。システムは教員 PC のサーバと複数の受講者 PC のクライアントで構成される。サーバでは、画面上の任意の位置に設定した矩形範囲を配信対象エリアとし、エリア内の画面を画像として連続的に配信する。エリアは、さらに矩形のブロックに分割し、変動のあったブロックのみを配信対象とし、データ圧縮した後に送信する。このとき全クライアントにマルチキャスト通信によって一斉配信する。クライアントでは、受信したブロックを展開し、ウィンドウにリアルタイムに描画する。

本システムの開発環境およびプログラミング言語には Visual C++ 2015 を用いた。グラフィックス処理に DirectX11 の API を用い、並列演算処理に CS5.0 仕様によるプログラミング言語 HLSL (High Level Shading Language) を用いた。HLSL はプログラマブルシェーダー専用の高水準言語である。C++ プログラムは CPU 上で、また HLSL プログラムは GPU 上で、それぞれコンパイルされたマシン語コードとして実行される。本システムの動作環境は、DirectX11 に対応した Intel Core

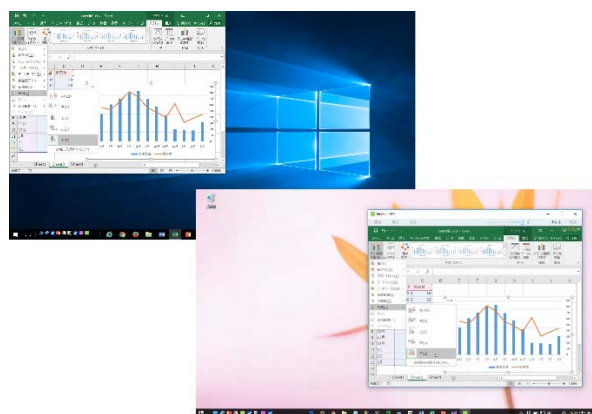


図 1. 画面配信の実行例 (左上 : サーバ, 右下 : クライアント)

i3/i5/i7, Celeron や AMD APU などの CPU 内蔵型の GPU や、非内蔵型の GPU の搭載マシンであり、OS は Windows8/10 を、また、ネットワーク環境は 100BASE-TX 以上の有線 LAN を対象とする。

4. システムの設計と開発

4.1 処理の構成と性能への影響

本研究では、2.で挙げた(1)～(3)の性能を向上させるために、GPU 処理を適用すべき処理内容を検討した。図 2 は、本システムにおけるデータ構造の概略であり、本システムを構成する主要な処理について、基本的な仕様と性質を以下で説明する。

(1) 画面キャプチャ処理

API を用いてデスクトップ画面をキャプチャし、フレーム (f) としてメモリに保存する。素早く変動する画面を配信する際、クライアント上での滑らかな画面再現のために、高フレームレートが必要となる。本処理では、液晶モニタの一般的な垂直同期周波数 60Hz を描画周期の最速値と捉え、60fps でキャプチャを繰り返す。本処理は、メモリ上の画像転送などを要するため、フレームサイズに応じて処理時間および CPU 負荷が生じる。通常 Windows で画面キャプチャを行うには API を用い、処理の負荷はその仕様に依存する。

(2) 差分ブロック検出処理

フレーム内の 8×8 px の矩形領域を 1 ブロックとし、フレームをブロックの集まりとして構成する。現在のフレーム f_t と 1 回前に記録された f_{t-1} を比較し、変動のあったブロックを差分ブロックとして検出する。差分のみを配信対象とすることで、後続処理の処理時間を抑える。本処理はフレーム内の全ブロックに対して行う必要があり、フレームサイズに応じて処理時間と CPU 負荷が生じる。

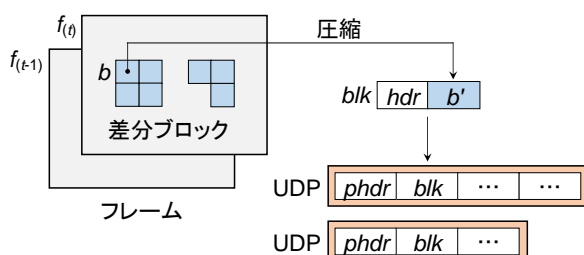


図 2. 画面配信データの構造

(3) ブロック圧縮処理

各差分ブロック (b) に対し、連長圧縮 (RLE : Run-Length Encoding) によって画像圧縮したブロック (b') を得る。RLE は同じ色のピクセルが連続しやすい画面画像に適しており、文字などが劣化しない可逆圧縮である。例えば単色の画素で構成されるブロックは圧縮率が最も高く、文字やグラデーションなどで同色の連続性が低い場合は圧縮率が低い。また圧縮が高速なため、高フレームレートおよび低遅延時間を実現しやすい。そして圧縮後の各 b' に対し、ブロック位置と圧縮サイズをブロックヘッダ (hdr) に付加し、送信用ブロック構造 (blk) を形成する。本処理は変動する差分ブロック数に応じて処理時間および CPU 負荷が増加する。

(4) データ送信処理

blk は UDP パケットに收容し、IPv4 マルチキャスト (RFC1112) で送信する。このとき UDP パケットの最大データサイズに収まる最大数の blk およびフレームのタイムスタンプなどを含むパケットヘッダ ($phdr$) を格納し、1 個以上のパケットで送信する。本処理では、差分ブロック数および圧縮率に応じて送信データ量が決定し、データ量にほぼ比例した処理時間を要する。CPU 負荷は、主にパケット構築およびネットワーク API 呼び出し時に発生する。

ブロック送信処理の通信プロトコルである UDP には TCP におけるパケット到達の確認応答や損失パケットの再送機能はない。高フレームレートによる連続送信処理では、これらの機能は遅延時間を増加させる要因でもある。そのため、本システムでは損失パケットの再送は独自に実装し、TCP のような送信待ちの発生を回避している。また、既存システム^(3,4)のユニキャスト通信では、大人数教室の授業において、接続クライアント数に比例して送信コストが増加する。本システムのマルチキャスト通信では、クライアント数の増加はサーバの配信性能に影響を及ぼさない。

送信処理では、送信データ量以外に処理時間に影響する要因として、LAN の伝送速度および伝送遅延時間が挙げられるが、これらはアルゴリズムや API の選択によって性能を改善できない。また、送信バッファや送信間隔の最適化を行えば、パケット送信のバースト性を高め処理時間が短縮できるが、パケット損失による画面劣化のリスクと

のトレードオフとなるため、検討から除外した。

4.2 実装方法

本研究では、GPU 処理の適用で性能向上が見込めるものとして、4.1 で示した(1)画面キャプチャ、(2)差分ブロック検出、(3)ブロック圧縮の処理に対して高速化を検討した。なお、(4)データ送信処理は、主メモリからネットワーク I/O へのデータ転送が主体となり、GPU 処理を適用する利点が見当たらない。(1)~(3)に対する実装方法の組み合わせとして、表 1 に示す①GDI、②DX、③DX+CS（提案システム）の 3 手法を比較する。

画面キャプチャには、①では標準的グラフィックス機能である GDI(Graphics Device Interface) の API を、②と③では GPU 上で動作する DXGI(DirectX Graphics Infrastructure)⁽⁹⁾ の API を用いた。DXGI は DirectX の一部として低水準に位置する高速なハードウェア制御機能である。差分ブロックの検出および圧縮には、①と②では CPU 上のプログラム処理(CPU)を、③では GPU 上のプログラム処理(GPU)を用いた。GPU では、CS による並列演算を行う。

画面キャプチャにおいて、GDI 手法では画面上の矩形範囲を指定し、CPU 側に画像データとして取得する。この API 処理は画面変動を検知できず、また定常的に数十 ms の処理時間を要する。一方 DXGI の API 処理は高速で処理時間は数 ms であり、画面変動を検知できるため後続処理の必要性を判断して効率を高めることができる。さらに、画面範囲内で前回から変動のあったウィンドウ更新領域に限定して GPU 側の画面リソースをテクスチャリソースへコピーすることで画像を取得する。テクスチャの確保場所は、後続の差分ブロック検出の場に応じ、DX 手法では CPU 側、また DX+CS 手法では GPU 側に確保する。

差分ブロック検出について、全対象ブロックお

表 1. 各手法における処理方法の組み合わせ

対象処理	手 法		
	GDI	DX	DX+CS
画面キャプチャ	GDI API	DXGI API	DXGI API
差分ブロック検出	CPU*	CPU*	GPU†
ブロック圧縮	CPU*	CPU*	GPU†

* CPU 上のプログラム処理

† GPU 上の並列プログラム処理

よびブロック内の画素に対するループ処理は、CPU 上では逐次的に処理し、GPU 上では並列的に処理する。ブロック圧縮についても、差分ブロックおよびブロック内の画素に対するループ処理は、CPU 上で逐次的に処理し、GPU 上で並列的に処理する。また、圧縮後の送信用ブロック構造の構築については、GPU 上で行うことで CPU 側の処理を省力化する。

GPU と CPU の連携処理について、図 3 に処理の流れを示す。DX 手法と DX+CS 手法を比較すると、API 呼び出しが主体となる画面キャプチャは共に GPU 上で処理される。一方、アルゴリズムによるデータ処理が主体となる差分ブロック検出およびブロック圧縮は、処理の場が CPU 上か GPU 上かが異なる。GPU 上で行う場合、並列アルゴリズムによる高速化と CPU 負荷の低減が期待できる。さらに、GPU-CPU 間のデータ転送でも、更新領域全体の転送に比べ、差分のみの転送は効率がよい。例えば、ドロップダウンメニューやダイアログが表示された場合、それらの位置とサイズが更新領域となるが、領域内の背景色が更新前後で同じならば、差分ブロックは更新領域より実質的に少なく、データ転送量は少ない。

4.3 並列演算アルゴリズム

DS+CS 手法における GPU 上の処理手順について説明する。CS によるシェーダープログラムは主に差分ブロック検出とブロック圧縮で構成され、SPMD の並列演算モデルによって処理される。

アルゴリズム 1 は、並列差分ブロック検出処理の疑似コードである。1 行目ではスレッドグループ数 $GX \times GY$ で並列ループが構成され、2 行目では、 $TX \times TY$ の複数のブロックごとおよびブロッ

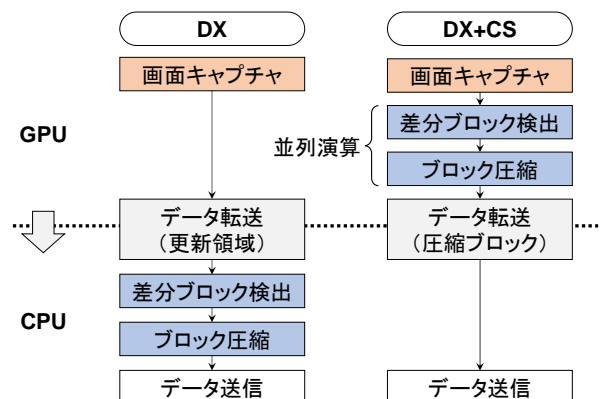


図 3. GPU と CPU による処理の構成

ク内画素 ($P=64\text{px}$) ごとのスレッドで並列ループが構成される。各添え字として p はブロック内のピクセル位置, idx はグループ内のブロック位置, pos は配信画面の画素位置となる。

9 行目ではフレームテクスチャである *Last*, *Curr* を比較し, 変動ブロックを判定し, 結果を共有メモリ上の *diff* に格納する。*diff* への書き込みはスレッド間で競合するが, CRCW (Concurrent Read Concurrent Write) モデルにおける同一値の書き込みを許容した Common CRCW により悪影響はない。13 行目は, 関数 *IncrementCounter* のアトミック操作によるスレッド間での排他的なカウントを行い, 共有メモリ上の *counter* に差分ブロックの番号として追加記録する。

最終的に 17 行目では, 全スレッドにより差分ブロック *Block* に画素を保存する。なお, 共有メモリはテクスチャリソースに比べ高速にアクセス可能であるため使用しており, グループ内の各スレッドから共有アクセスできる。また, *sync* はスレッド間の実行同期命令であり, 共有メモリへのアクセス順序を制御している。

アルゴリズム 2 は, 並列ブロック圧縮処理を行うものであり, 差分ブロック検出と連携させ, それによって得られた要素数 B 個の *Block* を GPU メモリ上で受け継ぎ, 1 行目の並列ループを構成

アルゴリズム1 並列差分ブロック検出処理

```

diff[i], i ← 0..TX*TY-1    ▷ 共有差分フラグ
counter[i], i ← 0..TX*TY-1  ▷ 共有差分カウンタ
Last                      ▷ フレーム画素(t-1)
Curr                      ▷ フレーム画素(t)
Block                     ▷ 差分ブロック
1: parallel for gx ∈ 0..GX-1, gy ∈ 0..GY-1 do
2:   parallel for tx ∈ 0..TX-1, ty ∈ 0..TY-1,
      p ∈ 0..P-1 do
3:     (gtx, gty) ← (gx * TX + tx, gy * TY + ty)
4:     idx ← ty * TX + tx
5:     pos ← (gtx * 8 + p % 8, gty * 8 + p / 8)
6:     if p = 0 then
7:       diff[idx] ← 0
8:     sync
9:     if Last[pos] ≠ Curr[pos] then
10:      diff[idx] ← 1
11:    sync
12:    if p = 0 ∧ diff[idx] = 1 then
13:      counter[idx] ← IncrementCounter()
14:      Block[counter[idx]].idx ← gtx * GX * TX + gtx
15:    sync
16:    if diff[idx] = 1 then
17:      Block[counter[idx]].pixel[p] ← Curr[pos]
```

する。2 行目はブロック画素ごとの並列ループである。なお, *boundary*, *sum*, *runPos* は共有メモリ上に確保している。本処理では 3~22 行目に並列 RLE アルゴリズム⁽¹⁰⁾を実装し, 図 4 のように, 11 行目で画素の不連続性を *boundary* に記録し, 14 行目の排他接頭部和 (exclusive prefix sum) によって *sum* に *boundary* の累計を求める。18 行目で連続画素以外の位置を *runPos* に記録し, そこから圧縮画素の連続数が取得可能となる。

24 行目以降では, 送信用ブロック構造を並列に構築する。関数 *Format1* は圧縮データに対する処

アルゴリズム2 並列ブロック圧縮処理

```

boundary[i], i ← 0..P-1    ▷ 共有境界フラグ
sum[i], i ← 0..P           ▷ 共有排他接頭部和
runPos[i], i ← 0..P-1     ▷ 共有境界位置
Block[i], i ← 0..B-1      ▷ 差分ブロック
R[i], i ← 0..B-1         ▷ 圧縮ブロック
1: parallel for b ∈ 0..B-1 do
2:   parallel for p ∈ 0..P-1 do
3:     if p = 0 then
4:       block ← Block[b]
5:       sum[0] ← 0
6:     sync
7:     if p = P-1 then
8:       flag ← 1
9:     else
10:      flag ← block.pixel[p] ≠ block.pixel[p+1]
11:    boundary[p] ← flag
12:    sum[p+1] ← flag
13:    sync
14:    for i ∈ 1..log2P do
15:      if p+1 > 2i-1 then
16:        sum[p+1] ← sum[p+1] + sum[p+1-2i-1]
17:      sync
18:    if boundary[p] = 1 then
19:      idx ← sum[p]
20:      runPos[idx] ← p+1
21:    else
22:      idx ← 0
23:    sync
24:    size ← sum[P]
25:    if size ≤ P * 3 / 4 then
26:      if p = 0 then
27:        R[b].inf ← F < 24 | (size*4) < 16 | block.idx
28:      if boundary[p] = 1 then
29:        if idx = 0 then
30:          len ← runPos[idx]
31:        else
32:          len ← runPos[idx] - runPos[idx-1]
33:        R[b].run[idx] ← Format1(block, p, len)
34:      else
35:        if p = 0 then
36:          R[b].inf ← block.idx | (P * 3) < 16
37:        R[b].run[p] ← Format2(block, p)
```

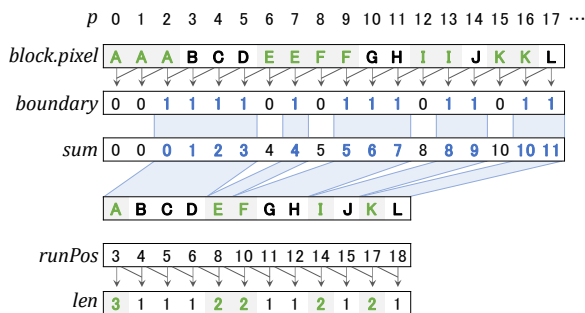



図 4. 並列 RLE 処理

理を定義し、また、*Format2* は RLE の性質上、圧縮サイズが元サイズより増加した際の処理を定義している。これらの関数は、テクスチャ画素の ARGB (32bit) 形式から圧縮後の RGB (24bit) 形式の並びにビット演算によって編成するものである。*R* はメンバ *inf* および *run* で構成された構造体であり、*inf* には圧縮フラグ *F*、圧縮サイズ、ブロック位置を格納し、*run* には圧縮画素を格納する。*R* のメモリレイアウトは UDP パケットに単純格納できる形にしてあり、差分ブロック数の分だけ GPU 側から CPU 側にデータ転送する。

5. 実験と考察

5.1 性能比較実験

3 つの手法に対し、性能比較実験を実施した。実験に用いたサーバマシンは CPU Intel Core i7 3.4GHz、メモリ 8GB、GPU AMD Radeon HD7950 (コアクロック 1000MHz、メモリ 3GB、メモリ帯域幅 240GB/s、Stream Processor 1792 基、GCN Compute Unit 28 基、バス PCI-E3.0)、モニタ解像度 1920×1080、LAN 規格 1000BASE-T である。

配信対象は、800×600px の配信エリア内に GUI と同じサイズのフォントで文字描画と背景色による消去を 100ms 間隔で交互に繰り返すものである。ブロック変動率として 1%、5% および 10% ~100% まで 10% 刻みで変化するように文字数を調整して各 1 分間継続する。なお、ブロック圧縮率は全ブロックで均一とし、文字描画時で 52.08%、消去時で 2.08%、平均して 27.08% である。

計測項目は、①画面キャプチャ時間、②差分ブロックの検出および圧縮時間、③データ送信時間、④CPU 使用率である。①~③は 1μs 精度の高分解能パフォーマンスカウンタ API を用いて、1 フレームごとに計測し、④は PDH (Performance

Data Helper) API を用いて 1s 間隔で取得する。計測値の記録処理では、計測への影響を抑えるためにマルチスレッドによる非同期処理でログファイルに出力している。結果は 3 つの手法による計測を各 10 回実施し、ブロック変動のないフレームを除いて平均値を求めた。

5.2 実験結果

図 5 はブロック変動率に対する画面キャプチャ時間、差分ブロック検出時間、およびブロック圧縮の合計時間の平均値を示している。全ブロック変動率での平均値は、GDI 15.68±0.64ms、DX 3.57±0.80ms、DX+CS 2.13±0.61ms であった。GDI 手法と DX 手法では、0.23 倍に短縮されており、これは、API の性能差と 4.2 で述べた画面変動の検知による効率性の効果と考えられる。また、DX 手法と DX+CS 手法では、差分ブロック検出およびブロック圧縮の演算手法が異なる。並列演算を用いた DX+CS 手法では 0.60 倍に短縮されており、提案手法が最も高速な結果となった。

並列演算による計算量を見ると、差分ブロック検出における 1 ブロック (*n* px) の処理の計算量は、逐次演算での $O(n)$ に対し、並列演算ではプロセッサ *n* 個を使用して $O(1)$ である。また、ブロック圧縮における 1 ブロックの処理の計算量は、逐次演算での $O(n)$ に対し、並列演算では、接頭部和のループに着目すると、プロセッサ *n* 個を使用して $O(\log n)$ である。よって DX 手法の逐次演算を DX+CS で並列化したことで計算量が減り、DX 手法に比べ DX+CS 手法では、差分ブロック検出+圧縮の処理時間は平均 0.58 倍に短縮された。

圧縮データを得る際、GPU へのデータ転送命令の発行直後に、CPU 側でデータにアクセスするためのメモリマッピング (メモリロック) を行う。このとき、GPU 上の圧縮処理およびデータ転送が完了するまでの間 CPU と GPU が同期状態となる。本システムでは、遅延を最小にするために画面キャプチャからデータ送信までを連続処理する必要があるが、同期によって CPU と GPU の並列性が失われる欠点がある。この影響を軽減するために、GPU-CPU 間のデータ転送量を最小限にすることが考えられ、DX+CS 手法では差分ブロックのみを対象とした圧縮データの転送を実施している。DX+CS 手法でブロック変動率に応じて処理時間が上昇しているのは、処理対象ブロックの数が多いため、並列ブロック圧縮に対するスレッ

ドの並列性が低下し、さらに GPU-CPU 間のデータ転送量の影響を受けているものと考えられる。

図 6 はデータ送信処理時間の平均値であり、処理時間は送信データ量に比例している。全手法で処理内容が同一であり、処理時間は主に単純なパケット構築時間、ネットワーク API の呼び出しとその I/O 処理の待ち時間で構成される。

図 7 は CPU 平均使用率である。平均すると、

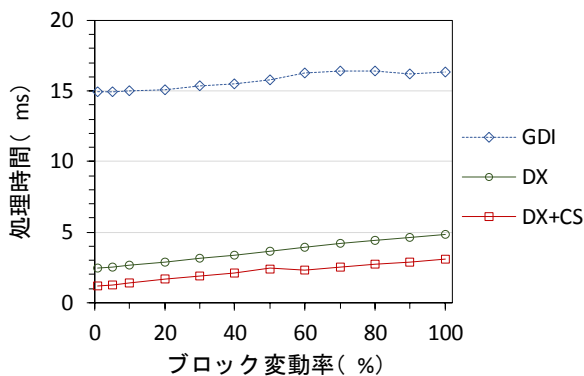


図 5. 画面キャプチャ・差分ブロック検出・ブロック圧縮の合計処理時間

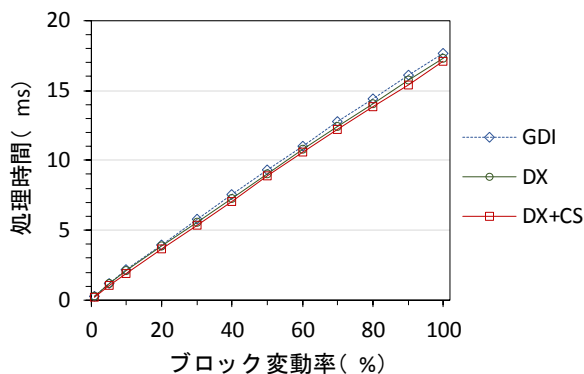


図 6. データ送信処理時間

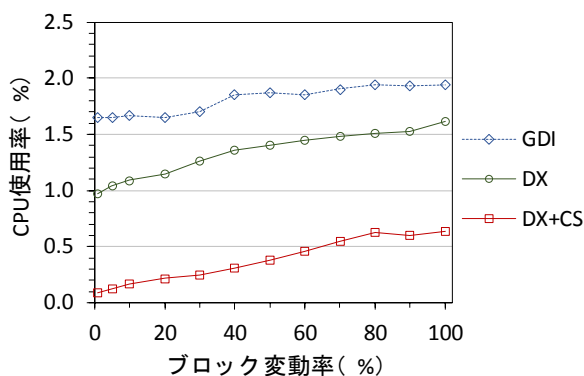


図 7. CPU 負荷

GDI 1.80%, DX 1.32%, DX+CS 0.37%である。GDI 手法に対し DX 手法が低いのは、主に GPU 側で処理される DXGI API の性能および、画面変動を検知できる機能性による、後続処理の効率的な実行制御のためと考えられる。

DX+CS 手法ではブロック変動率 1%~100%に対し CPU 使用率は 0.08%~0.64%と非常に低い。DX 手法との比較において、DX+CS 手法が低い理由は、データ処理の主要なアルゴリズム処理に適用した並列演算は、GPU 上での演算処理中に CPU 処理時間をほとんど消費しないためである。これによって、本システムを情報基礎教育授業で利用する際、対象ソフトウェアの多くが CPU 処理主体であるため、使用ソフトウェアへの影響が小さい低負荷システムが実現できる。

5.3 ソフトウェア操作画面による配信実験

前節の性能比較実験は、テストパターンを用いた画面配信であるが、今度は授業に近い画面内容を用い、実際に 1000BASE-T の LAN 上でクライアントに配信しながら計測した。クライアントにはサーバと同じスペックの PC を使用している。配信画面内容として表 2 に示す Word, Excel, PowerPoint による一連の操作を用いて、サーバにおけるフレームレート、処理時間（キャプチャ＋差分ブロック検出＋ブロック圧縮＋データ送信の合計時間）、CPU 使用率およびクライアントの処理時間（ブロック展開＋描画）について、性能比較実験と同様の方法で 10 回計測した。フレームレートは画面キャプチャ開始周期からフレームごとに算出している。

表 2. 配信対象のソフトウェア操作

Word (段落書式設定)	<ol style="list-style-type: none"> 1. 段落の選択と箇条書きの適用 2. 段落の選択と段落番号の適用 3. 段落の選択と行間隔の変更 4. 文字列の選択とフォント種類の変更 5. 文字列の選択とフォントサイズの変更
Excel (グラフ作成)	<ol style="list-style-type: none"> 1. セル範囲の選択と円グラフの挿入 2. セル範囲の選択と散布図グラフの挿入 3. セル範囲の選択と棒グラフの挿入 4. データ系列の折れ線グラフへの種類変更 5. 折れ線グラフ系列の第 2 軸設定
PowerPoint (作図)	<ol style="list-style-type: none"> 1. フリーフォームによる多角形の描画 2. 図形頂点のスモージング 3. 直線の挿入 4. 全図形の選択とグループ化 5. 図形グループのコピーと貼り付け 6. 図形グループの回転と反転

表 3. ソフトウェア操作画面の配信性能

	GDI	DX	DX+CS
フレームレート (fps)	54.90 (4.36)	59.76 (1.74)	59.67 (1.74)
処理時間 (ms)	14.78 (5.10)	3.11 (2.01)	2.01 (1.97)
CPU 使用率 (%)	1.56 (0.43)	0.85 (0.44)	0.09 (0.14)

()内は標準偏差

表 3 に、サーバにおける各計測平均値を示す。計測中の平均ブロック変動率は 0.43%、平均圧縮率は 35.99%であった。GDI 手法では、処理時間が $14.78 \pm 5.10\text{ms}$ であり、1 フレーム時間の超過によりフレームレートが低下している。また CPU 使用率も他と比べ高い。DX 手法および DX+CS 手法では、GPU の効果で処理時間が短い。システムの配信遅延時間を、サーバとクライアントの各処理時間の合計とした場合、DX が $5.03 \pm 2.72\text{ms}$ 、DX+CS が $3.93 \pm 2.69\text{ms}$ となる。また、CPU 使用率は、特に DX+CS 手法で 0.09%と低かった。

以上の結果より、GDI 手法では、1 フレーム周期の 16.67ms に対し余裕が厳しいため、安定した再現性が得られにくく、変動ブロックが増加した場合に悪化する可能性が高い。DX 手法および DX+CS 手法では、処理時間が短く低遅延時間とフレームレートの安定が得られている。両手法の比較では、サーバにおいて処理時間 0.64 倍、CPU 使用率 0.10 倍と DX+CS 手法が優れている。

6. まとめ

情報教育や大人数授業の問題解決として、PC 画面の詳細かつ素早い動きに対応できる画面配信システムの開発にあたり、GPU の処理能力を活用した手法を適用した。特に、DX+CS の提案手法による GPU と並列演算の活用で、画面の再現性、遅延時間、CPU 負荷で高い性能が得られた。

GPU の性能向上は今後も続くと予想されており、ノート PC における GPU 性能も進歩しつつある。このような状況は GPU 性能に依存する本システムにとって有益である。一方でディスプレイの高解像度化が進むなか、ノート PC でもフルハイビジョン以上の解像度が増加している。これは画面配信エリアの拡大にもつながりやすく、処理時間の影響はさらに深刻化し、並列演算処理の

最適化や圧縮率の高い圧縮アルゴリズムの適用も視野に入れる必要がある。本システムの性能改善は、学習者のストレスを軽減し、モチベーション低下を防ぐものと考えられ、それらは授業支援システムの実用性要因として重要である。

7. 参考文献

- (1) 黒須正明, “ユーザエクスペリエンスと満足度,” 放送大学研究年報, No.28, 2010, pp.71-83.
- (2) Microsoft, “Graphics APIs in Windows,” <https://msdn.microsoft.com/en-us/library/windows/desktop/ee417756%28v=vs.85%29.aspx>, 2016.
- (3) 谷成雄, 大城信康, 河野真治, “VNC を用いた講義用画面共有システムの設計・開発,” 情報処理学会研究報告, Vol.2012-OS-121, No.19, 2012, pp.1-5.
- (4) 山之上卓, 小荒田裕理, 片桐太樹, 小田謙太郎, 下園幸一, “HTML5 技術を利用した授業や会議向けデスクトップ画面実時間配信システムとその管理システムの試作,” 情報処理学会研究報告, Vol.2014-IOT-26, No.11, 2014, pp.1-8.
- (5) 後藤正幸, 大野明彦, 萩原拓郎, 横井利彰, “初級プログラミング科目を対象とした学内遠隔教育とその効果,” 武蔵工業大学環境情報学部情報メディアセンタージャナル, Vol.6, 2005, pp.30-37.
- (6) 徳差雄太, 松谷健史, 空閑洋平, 村井純, “低遅延により自然な遠隔コミュニケーションを実現する映像配信システムの提案,” マルチメディア, 分散, 協調とモバイルシンポジウム論文集, 2013, pp.911-917.
- (7) 小峯立也, 北島律之, “感覚間の同時性,” 電子情報通信学会技術研究報告 HIP, Vol.106, No.143, 2006, pp.39-43.
- (8) Microsoft, “Compute Shader Overview,” <https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx>, 2016.
- (9) Microsoft, “DXGI Overview,” <https://msdn.microsoft.com/en-us/library/windows/desktop/bb205075%28v=vs.85%29.aspx>, 2016.
- (10) Fang, W., He, B., Luo, Q., “Database compression on graphics processors,” Proceedings of the VLDB Endowment, Vol.3, No.1, 2010, pp.670-680.