GPUDirect RDMA を用いたリモートホストの異常検知手法

金本 颯将[1]　光来 健一[1]

概要：システムの異常には，アプリケーションや OS の障害をはじめ，ヒューマンエラーが原因の障害や外部からの攻撃といった様々なものが存在する．従来，ソフトウェアやハードウェアによる検知手法が用いられてきたが，いずれの手法にも問題点があり，高信頼・低コスト・高性能の 3 つの要件を満たすのは難しかった．これらの要件を満たす手法として，システムから隔離された汎用 GPU を用いて異常検知を行う GPUSentinel が提案されている．しかし，GPUSentinel では検知結果を外部に通知するために OS のネットワーク通信機能を利用する必要があり，OS の内部で障害が発生したり，攻撃によってネットワーク通信を阻害されると，検知結果を通知することができなくなる可能性がある．そこで本稿では，OS を介さずに GPU が直接ネットワーク通信を行い，検知結果をリモートホストに通知することができるシステム GRASS を提案する．GRASS では，リモートホストが GPUDirect RDMA を用いて GPU メモリに直接アクセスし，リモートホストと GPU がポーリングを行うことで監視対象ホストの CPU を用いずに通信を行う．そのため，GPU とネットワークカードが正常に動作していれば通信が可能である．我々は CUDA を用いて GRASS を実装し，GPU との通信性能を計測した．

1. はじめに

システムの異常には，アプリケーションや OS の障害をはじめ，ヒューマンエラーが原因の障害や外部からの攻撃といった様々なものが存在する．このようなシステムの異常はできるだけ早く検知する必要があるため，従来，ソフトウェアやハードウェアによる様々な検知手法が用いられてきた．ソフトウェアを用いた異常検知手法は，OS 上や OS 内部で情報を取得して異常を検知する．ハードウェアを用いた異常検知手法は，専用ハードウェアや CPU の機能を用いて異常を検知する．しかし，いずれの手法にも問題点があり，高信頼・低コスト・高性能の 3 つの要件を満たすのは難しかった．

これらの要件を満たす手法として，汎用ハードウェアである GPU を用いて異常検知を行う GPUSentinel [1] が提案されている．GPUSentinel では，システム起動時に GPU 上で OS 監視システムの実行が開始され，メインメモリから取得した情報を基に異常検知を行う．GPU は CPU やメインメモリから隔離されているため，システム異常の影響を受けにくい．しかし，GPUSentinel は検知結果を外部に通知するために OS のネットワーク通信機能を利用する必要がある．そのため，OS の内部で障害が発生したり，攻撃によってネットワーク通信を阻害されたりすると，検知結果を通知することができなくなる可能性がある．

そこで本稿では，OS を介さずに GPU と直接ネットワーク通信を行い，検知結果を取得することができるシステム GRASS を提案する．GRASS は GPUDirect RDMA を用いて監視対象ホスト上の GPU メモリに直接アクセスすることで通信を行う．リモートホストが RDMA Write を用いて要求を GPU メモリに書き込むと，GPU はポーリングを行うことでそれを受け取る．GPU が検知結果を GPU メモリに書き込むと，リモートホストは RDMA Read を用いてポーリングを行うことでそれを取得する．このように，GPU およびネットワークカード（NIC）が正常に動作していれば通信が可能であり，検知結果の通知が障害や攻撃の影響を受けにくい．

我々は CUDA や RDMA ライブラリを用いて GRASS を実装した．実験の結果，監視対象ホストにおいて OS が異常停止しても正常に GPU との通信を行うことができることが確認できた．また，GPU 上で動作する OS 監視システムの異常を検知するためのハートビートにかかる時間を測定し，ping と比較して十分に短い時間で死活監視を行うことができることを確認した．さらに，GPU メモリを用いることによる通信性能への影響も小さいことを確認した．

以下，2 章ではシステムの異常に対する従来の異常検知手法について述べる．3 章では本稿で提案するシステムである，異常検知対象の OS 等を介さずに直接，監視対象ホストの GPU から検知結果を取得するシステム GRASS に

[1] 九州工業大学
Kyushu Institute of Technology

表 1: システム異常の分類

種類		具体例
障害	ハードウェア	機器の故障，停電や過電流による機能停止
	ソフトウェア	データ欠損，仕様外の動作
	ヒューマンエラー	思い込みや勘違い，過失による誤操作
性能低下		CPU の性能低下，通信の遅延
攻撃		セキュリティ上の脆弱性をついた攻撃

ついて述べる．4 章では GRASS の実装について述べ，5 章では GRASS の有効性を確かめるための実験について述べる．6 章ではこれまでに提案されてきた関連研究について述べ，7 章で本稿をまとめる．

2. システムの異常検知

2.1 従来の異常検知手法

システムの異常には，ハードウェアやソフトウェアの障害をはじめ，ヒューマンエラーが原因の障害や性能低下，外部からの攻撃といった様々なものが存在する．代表的なものとして，表 1 のようなシステム異常が挙げられる．本稿では，ハードウェア障害については異常検知機構に影響を与えないものだけを対象とする．これらシステムの異常はできるだけ早く検知して対処を行う必要がある．そのため，システム障害から復旧するための障害検知，システム性能を維持するための性能監視，攻撃の被害を最小化するための侵入検知などが行われている．

ソフトウェアを用いた異常検知手法では，OS 上や OS 内部で情報を取得してシステムの異常を検知する．例として，システムの状態を SNMP でリモートホストに送信して障害を検知する手法が挙げられる．また，アンチウィルスを用いてコンピュータウィルスへの感染を検知する手法も用いられる．ソフトウェアを用いた異常検知手法に共通する問題点として，OS の内部に異常が発生すると，検知を行うことができなくなることが挙げられる．OS が異常停止すると検知システムは動作しなくなる．また，OS の内部にカーネルルートキットがインストールされると，OS 上のセキュリティソフトウェアに偽の情報が返され，以降の攻撃を検知できなくなる．

ハードウェアを用いた異常検知手法では，専用ハードウェアや汎用 CPU の隔離実行のための機能を用いて異常を検知する．例として，PCI カード上でメインメモリ上の OS の整合性を安全に検査して外部に通知する手法 [6] や Intel 製 CPU のシステムマネジメントモード（SMM）を利用してハイパーバイザを安全に監視する手法 [7,8] などが挙げられる．ハードウェアを用いた異常検知手法はソフトウェアを用いる手法と比較すると信頼できるが，専用の PCI カードを用いる場合には高コストである点や SMM の実行は低速である点などに課題がある．

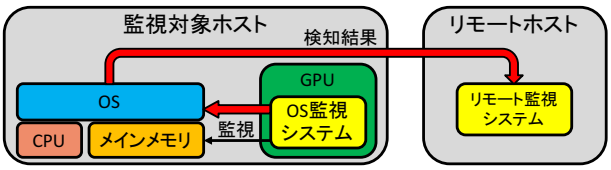

図 1: GPUSentinel のシステム構成

2.2 GPUSentinel

高信頼・低コスト・高性能の 3 つの要件を満たす手法として，GPU を用いた異常検知手法である GPUSentinel [1] が提案されている．GPUSentinel のシステム構成を図 1 に示す．GPUSentinel は，監視対象システムに搭載された GPU 上で OS 監視システムを動作させてシステムを監視し，システムの異常を検知する．GPUSentinel では OS 監視システムが GPU を占有して自律的に動作する．OS 監視システムは障害が発生する前のシステム起動時に起動され，その後 GPU 上で動作し続ける．GPU からシステムを監視するために，OS 監視システムはシステムのメインメモリ上のデータを取得して解析する．

GPU は OS が動作する CPU やメインメモリから物理的に隔離されており，OS の異常の影響を受けにくいため，異常検知の信頼性は高い．OS 側から GPU への攻撃についても PixelVault [10] で提案されている手法を用いて防ぐことができる．また，GPU は多くの計算機に標準的に搭載されており，一部の GPU を除いてコストは低い．その上，GPU は多数の演算コアを保持しており，それらを用いることで並列処理を行うことが可能であることから高性能であるといえる．

しかし，GPUSentinel では検知結果を外部に通知するために OS のネットワーク通信を用いる必要がある．GPU は能動的にネットワーク通信を行う機能を持たないためである．GPUSentinel を用いた異常検知の結果をリモートホストに通知する際には，OS 上のプロセスを介してリモートホストに検知結果を送信することになる．そのため，ソフトウェアを用いた異常検知手法と同様に，OS 内部に異常が発生したり，攻撃によってネットワーク通信を阻害されたりすると，監視結果を通知することができなくなる可能性がある．

3. GRASS

本稿では，OS を介さずに GPU が直接ネットワーク通信を行い，異常検知の結果をリモートホストに通知することができるシステム GRASS を提案する．GRASS のシステム構成を図 2 に示す．GRASS では，GPUDirect RDMA を用いることでリモートホストと GPU が直接通信を行う．リモートホストからの要求に基づいて，監視対象ホストの GPU 上の OS 監視システムがメインメモリ上の OS データを監視し，検知結果を GPU メモリに格納する．リモート

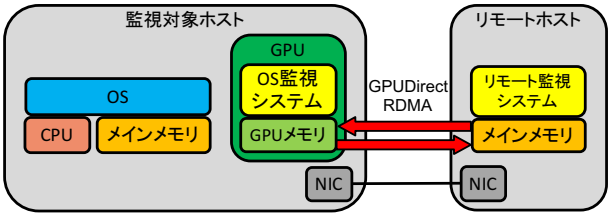

図 2: GRASS のシステム構成

ホストはこの GPU メモリから直接，検知結果を取得することができる．そのため，監視対象ホストにおいて OS 等に異常が発生しても GPU および NIC が正常に動作していれば，リモートホストから異常を検知することができる．GPU や NIC に異常が発生した場合には通信を行うことができなくなるが，応答がない場合には異常が発生していると判断することができる．ただし，その場合には異常に関する詳細な情報を取得することはできない．

GPUDirect RDMA は，CPU を介さずにリモートホスト上の GPU メモリに直接アクセスするためのハードウェア機構である．GPUDirect を用いて GPU メモリを物理メモリアドレス空間にマッピングし，NIC の RDMA 機能を用いてマッピングした GPU メモリにリモートホストから直接アクセスすることができる．具体的には，RDMA Write 機能を用いてリモートホストの GPU メモリに直接データを書き込み，RDMA Read 機能を用いてリモートホストの GPU メモリからデータを読み込む．

検知結果の要求を行う際には，リモートホストは RDMA Write を実行し，監視対象ホストの GPU メモリに要求を書き込む．このとき，GPU は演算コアを 1 つ専有し，RDMA Write による書き込みをポーリングによりチェックし続ける．ポーリングを行うのは，RDMA Write の完了を GPU に通知するハードウェア機構が存在しないためである．GPU は多数の演算コアを持つため，1 つを専有しても検知性能への影響はほとんどない．要求を受信した OS 監視システムは必要に応じてメインメモリからデータを取得し，GPU メモリに検知結果を格納する．

リモートホストは検知結果の取得を行うために，GPU メモリに対して，繰り返し RDMA Read を実行することによりポーリングを行う．これは GPU からリモートホストに対して RDMA Write を実行することができないためである．リモートホストではポーリングに CPU を用いるため，一定の間隔を開けてポーリングを行うことで，リモートホストの CPU 負荷を下げることが可能である．GPU メモリに検知結果が格納されると，ポーリングを終了して検知結果を取得する．

GRASS では，監視対象ホストの GPU メモリ上で稼働している OS 監視システムは GPUSentinel [1] を用いることで，メインメモリから情報を取得する．GPU からメインメモリを参照するために，CUDA が提供するマップトメモリ機能を利用する．GPUSentinel が提供する OS を用いることで，メインメモリ全体を GPU アドレス空間にマッピングすることができる．また，GPU 上で稼働している OS 監視システムが OS のデータを取得する際には，仮想アドレスを物理アドレスに変換し，物理アドレスを GPU アドレスに変換する必要がある．これらのアドレス変換を透過的に行えるように GPUSentinel が提供する LLView を用いてプログラム変換を行う．

GRASS は 3 種類の異常検知手法をサポートしている．1 つ目は，監視対象ホストの GPU 上で稼働している OS 監視システムが GPUSentinel を用いて定期的に OS の異常検知を行い，その結果を GRASS を用いてリモートホストに通知する手法である．2 つ目は，リモートホストが必要とするデータを GPU 上の OS 監視システムに要求し，取得したデータを用いて異常を検知する手法である．3 つ目は，リモートホストが監視対象ホストのメインメモリ全体の一括送信を要求し，異常検知を行う手法である．2 つ目および 3 つ目の方法は，リモートホストにおいて Transcall [2] を用いることで既存の IDS が実行可能である．また，GPU 上の OS 監視システム自体の異常を検知するために，定期的にハートビートを送ることもできる．

4. 実装

我々は CUDA 8.0，Verbs API，RDMA CM を用いて GRASS を実装した．GPUDirect RDMA を利用できるようにするために，Mellanox OFED 4.1 および nvidia-peer-memory 1.0.7 を用いた．また，監視対象 OS として GPUSentinel 向けに修正した Linux 4.4.64 を動作させ，OS 監視システムのコンパイルには LLVM 5.0.0 を用いた．

4.1 GPUDirect RDMA を用いた通信

リモートホストから監視対象ホストに対して RDMA 通信を行えるようにするために，それぞれのホストで Protection Domain（PD）を作成する．そして，PD の中に RDMA 通信の端点となる Queue Pair（QP）を作成する．PD を用いることで，外部からの RDMA アクセスの範囲を制限することができる．また，送受信の完了を知るために Completion Queue（CQ）を作成する．その後で，監視対象ホストからリモートホストに対してコネクションを確立する．

監視対象ホストではリモートホストとの RDMA 接続を

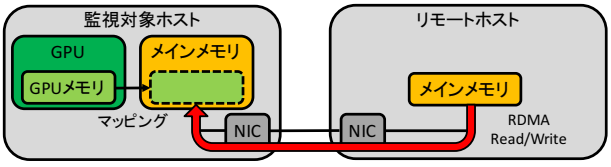

図 3: GPUDirect RDMA 機構

確立した後，図 3 のように GPU メモリ上に GPUDirect RDMA に使用するバッファを確保し，Memory Region を作成する．そのバッファのアドレスを RDMA Send を用いてリモートホストに送信する．GPU メモリ上のバッファはプロセスの統合仮想アドレス空間にマッピングされており，その仮想アドレスは NVIDIA ドライバによって物理アドレスに変換される．同時に，ホストチャネルアダプタ（HCA）によって生成されるリモートキーも送信する．リモートキーは RDMA Read/Write の際に使われ，値が一致しない場合にはアクセスが拒否される．

RDMA Write を用いることで，リモートホストは監視対象ホストの GPU メモリにデータを書き込む．リモートホストは書き込むデータが格納されたメインメモリのアドレスとサイズおよび，書き込み先の GPU メモリのアドレスを指定する．一方，RDMA Read を用いることで，リモートホストは監視対象ホストの GPU メモリからデータを読み込む．リモートホストは読み込むデータが格納された GPU メモリのアドレスとサイズおよび，書き込み先のメインメモリのアドレスを指定する．リモートホストは CQ を用いることで RDMA Read/Write の完了通知を受け取る．

4.2 検知結果の要求・取得

GRASS は，GPU 上に確保した RDMA 用メモリ領域を RDMA 受信バッファ，書き込み完了フラグ，RDMA 送信バッファ，読み込み許可フラグに分割する．RDMA 受信バッファはリモートホストからの送信データを格納するために用いられ，RDMA 送信バッファは監視対象ホストの GPU からの送信データを格納するために用いられる．書き込み完了フラグと読み込み許可フラグは RDMA 通信を行うホスト間で同期をとるために用いられる．

GPU との通信の流れを図 4 に示す．監視対象ホストの GPU に要求を送信する際に，リモートホストは RDMA Write を用いて，GPU メモリ上の RDMA 受信バッファに要求を書き込む．その後で，RDMA Write を用いて GPU メモリ上にある書き込み完了フラグをセットし，要求の送信完了を GPU 上の OS 監視システムに通知する．GPU 上の OS 監視システムは多数ある演算コアの 1 つを利用し，書き込み完了フラグがセットされるまでポーリングを用いてチェックを続ける．OS 監視システムは書き込み完了フラグがセットされたことを検知すると，GPU メモリ上の RDMA 受信バッファに格納された要求を取得し，書き込み完了フラグをクリアする．

OS 監視システムは受信した要求に従って異常検知を行い，検知結果を GPU メモリ上の RDMA 送信バッファに格納する．その後で GPU メモリ上にある読み込み許可フラグをセットし，検知結果の送信準備の完了をリモートホストに通知する．リモートホストはポーリングを用いて読み込み許可フラグに対して繰り返し RDMA Read を行う．

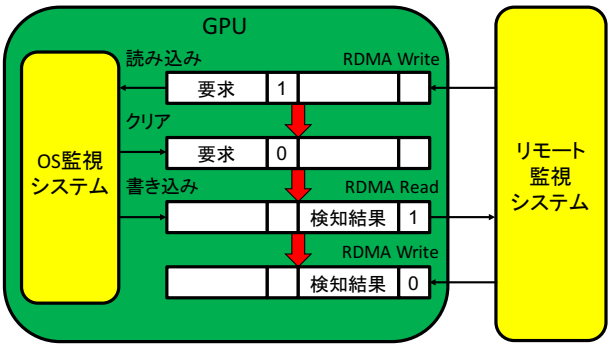

図 4: GPU との通信の流れ

リモートホストの負荷を下げるために，ポーリングは一定の間隔をあけて行う．読み込み許可フラグがセットされたことを検知すると，リモートホストは GPU メモリ上の RDMA 送信バッファから RDMA Read で検知結果を取得して，RDMA Write で読み込み許可フラグをクリアする．

4.3 GPU からメインメモリへのアクセス

GRASS ではメインメモリの内容を GPU から参照するために，CUDA が提供するマップトメモリ機能を利用する．マップトメモリはプロセスのメモリを GPU アドレス空間にマッピングし，GPU 上のプログラムから参照可能にする機能である．メインメモリを直接，GPU アドレス空間にマッピングすることはできないため，図 5 のように一旦，プロセスのアドレス空間にマッピングした上で GPU のアドレス空間にマッピングする．OS 等に異常が発生する前にメインメモリを GPU アドレス空間にマッピングしておくことで，異常発生後も GPU 上の OS 監視システムはメインメモリの情報を参照することができる．しかし，メインメモリ全体を GPU アドレス空間にマッピングすると CUDA によってページがピン留めされ，メインメモリ全体がロックされて使用中になってしまう．

そこで，GRASS では GPUSentinel [1] で提案されているメモリ管理機構を用いた．GPUSentinel は Linux カーネルのメモリ管理に修正を加え，/dev/pmem というデバイスファイルを用意している．/dev/pmem を mmap システムコールを用いてマッピングすることでメモリページの参照カウンタの増加を抑え，ピン留めを行う際にロックを行わないようにすることができる．また，メインメモリ全体

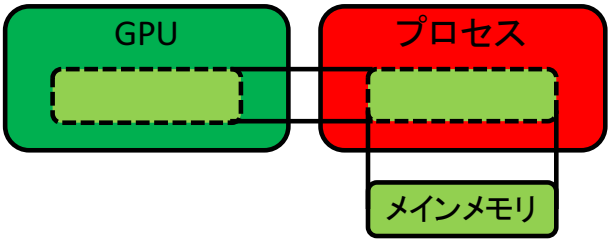

図 5: マップトメモリを用いたメインメモリのマッピング

のサイズのメモリをマップトメモリで利用できないというCUDA の制限を回避するために，sysinfo システムコールをフックし，メインメモリのサイズとして少し大きな値を返すようにする．

4.4 OS 監視システムの作成

GRASS では，OS 監視システムが GPU アドレス空間にマッピングされたメインメモリ上の OS データを取得する．そのためにまず，OS のページテーブルを用いて OS データの仮想アドレスを物理アドレスに変換し，変換した物理アドレスを GPU アドレスに変換する．GRASS は GPUSentinel で提案されている LLView [1] を用いて，これら一連のアドレス変換を透過的に行う．LLView は，GPU 上のプログラムを LLVM を用いてコンパイルし，生成された中間表現を変換することで透過的なアドレス変換を実現する．メモリからデータを読み込む際に実行される load 命令の直前にアドレス変換を行うコードを挿入することで，変換されたアドレスに対して load 命令を実行させることができる．また，中間表現で使われている OS の大域変数は対応する仮想アドレスに置換して，アドレス変換が行われるようにする．

5. 実験

GRASS の有効性を確かめるための実験を行った．監視対象ホストおよびリモートホストにはそれぞれ表 2 および表 3 のマシンを用いた．これらのホストは 100 ギガビットイーサネットで接続した．

5.1 OS の異常停止中の動作確認

監視対象ホストにおいて OS が異常停止しても正常に GPU との通信を行うことができることを確認する実験を行った．そのために，監視対象ホストで/proc/sysrq-trigger に”c”を書き込み，カーネルパニックを発生させた．実験の結果，監視対象ホストにおいてカーネルパニックが発生しても，正常に GPU 上の OS 監視システムとの通信を行うことができることを確認した．

表 2: 監視対象ホスト

OS	Linux 4.4.64
CPU	Intel Xeon E5-1603 v4
メモリ	8GB
GPU	NVIDIA Quadro M4000
NIC	Mellanox MCX455A-ECAT ConnectX-4 VPI Adapter Card
ソフトウェア	NVIDIA Graphics Driver 375.66, CUDA 8.0，Mellanox OFED 4.1 nvidia-peer-memory 1.0.7

表 3: リモートホスト

OS	Linux 4.10.0
CPU	Intel Xeon E3-1270 v3
メモリ	8GB
NIC	Mellanox MCX455A-ECAT ConnectX-4 VPI Adapter Card
ソフトウェア	Mellanox OFED 4.1，nvidia-peer-memory 1.0.7

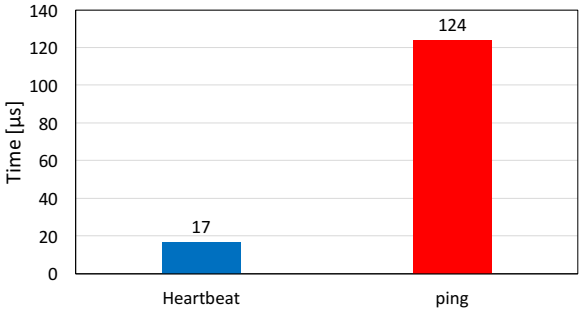

図 6: ハートビートの性能

5.2 ハートビートの性能

GPU 上の OS 監視プログラムの異常を検知するためにリモートホストから定期的に送信するハートビートにかかる時間を計測した．比較として，リモートホストから監視対象ホストの OS への ping にかかる時間の計測を行った．ハートビートでは，リモートホストは RDMA Write で書き込み完了フラグのみをセットし，要求は書き込まない．そして，GPU が読み込み許可フラグをセットするのを RDMA Read を用いたポーリングで待ち，検知結果の受信は行わない．ハートビートも ping も 10000 回繰り返して平均値および標準偏差を計算した．

計測結果を図 6 に示す．ping は 124 マイクロ秒かかるのに対して，ハートビートは 17 マイクロ秒で実行できることが分かった．また，ハートビートの標準偏差は 4 マイクロ秒と小さいことから安定していることも分かる．この結果より，GRASS では十分に短い時間で死活監視を行うことができることを確認できた．

5.3 検知結果の要求・取得の性能

GRASS において要求および検知結果のサイズを変化させて通信にかかる時間の計測を行った．RDMA 受信バッファと RDMA 送信バッファに書き込むデータのサイズをそれぞれ 1KB から 4MB まで増加させ，検知結果の要求・取得にかかる時間を計測した．この計測を 10000 回繰り返してスループットを計算した．なお，要求にかかる時間を測定する際には RDMA 送信バッファに書き込む検知結果のサイズを 1KB，取得にかかる時間を測定する際には RDMA 受信バッファに書き込む要求のサイズを 1KB に固定して計測を行った．

計測結果を図 7 に示す．実験結果から，RDMA 受信バッファに要求を書き込む性能よりも RDMA 送信バッファから検知結果を読み込む性能の方が高いことが読み取れる．

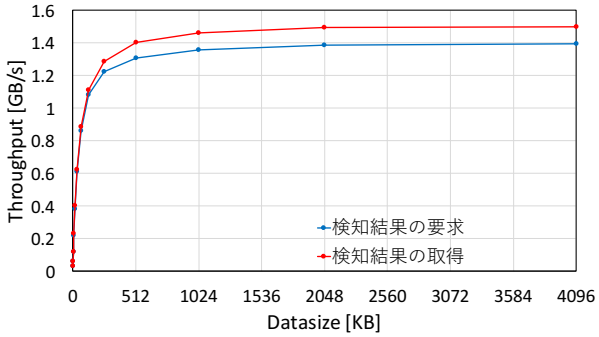

図 7: 検知結果の要求・取得の性能

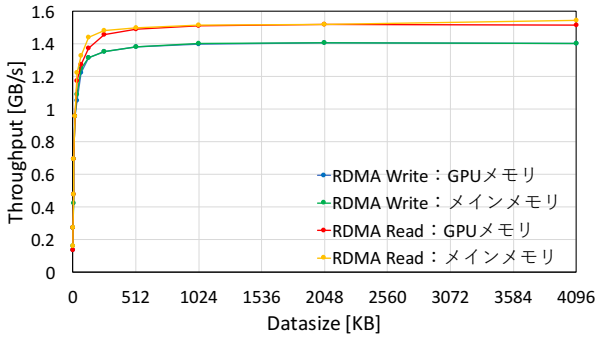

図 8: メモリの違いによる RDMA Read/Write への影響

いずれの場合もデータサイズが 1MB 程度でスループットがほぼ一定になった．

5.4 GPU メモリを用いる影響

GPU メモリに対して RDMA を行うと PCIe を経由してアクセスすることになるため，メインメモリに対する RDMA より性能が低下する可能性がある．そこで，RDMA 用メモリ領域を GPU メモリ上ではなくメインメモリ上に確保した場合のハートビートおよび RDMA の性能を計測した．ハートビートは，5.2 節と同様に計測し，RDMA は 1KB から 4MB までのデータの読み書きを 10000 回繰り返してスループットを計算した．

ハートビートにおいて，GPU メモリを使用した場合は図 6 に示すように 17 マイクロ秒で実行でき，メインメモリを用いた場合も同じく 17 マイクロ秒で実行できた．このことから，ハートビートでは，GPU メモリを使用する影響は見られなかった．次に，それぞれのメモリを使用した際の RDMA Read/Write の計測結果を図 8 に示す．実験結果から GPU メモリを使用することで，RDMA Write の性能に違いは見られなかったが，RDMA Read の性能はわずかに低下していることが分かった．メインメモリ上に RDMA 用メモリ領域を確保することで GPUDirect RDMA に非対応の GPU を利用することも可能になるが，OS 等の異常の影響を受けやすくなる．

```
kanamoto@z230-workstation:~/GRASS_1.0$ ./rdma_server
Server is listening successfully port: 20886
A new connection is accepted from 10.0.0.5
------------------------------------------------------------
buffer attr, addr: 0x7060c0000 , len: 9221 , stag : 0x722e
------------------------------------------------------------
Command: get
dst: 'comm:swapper/0
comm:systemd
comm:kthreadd
comm:ksoftirqd/0
comm:kworker/0:0H
```

図 9: プロセス情報の取得結果

5.5 プロセス情報の取得時間

GPUSentinel を用いて監視対象ホストのプロセス名の取得を行い，100 回繰り返して平均値および標準偏差を計算した．プロセス情報の取得結果は図 9 のようになった．取得時間は 88 マイクロ秒となり，標準偏差は 23 マイクロ秒であった．

6. 関連研究

GPUnet [3] は GPU プログラムにソケットと高水準ネットワーク API を提供する．GPUnet では，同じマシンか異なるマシンかに関わらず，GPU 内スレッドが他の GPU または CPU のスレッドと通信することができる．リモートホストが GPU プログラムにデータを送信する際には，GPUDirect RDMA を用いて GPU メモリに対してデータを書き込む．しかし，GPU プログラムがリモートホストにデータを送信する際には，CPU 経由で RDMA Write を実行するため，GPU だけでは通信を行うことができない．OS の内部に異常が発生してしまうと，リモートホストに監視結果を通知することができなくなる可能性がある．

Intel AMT [4] は，ネットワーク経由でホストの情報を取得したり，ホストをリモートコントロールしたりすることができるハードウェア機構であり，Intel vPro に対応したホストで利用できる．Intel AMT は OS を含むソフトウェアに依存せず，通常の NIC を用いて通信を行うことができる．IPMI などの同様のハードウェア機構もよく利用されている．しかし，監視することができる項目がハードウェア情報とソフトウェアによって登録された情報に限定されており，高度な異常検知を行うのは難しい．仮想マシン（VM）に対して Intel AMT の機能を提供するシステムとして vAMT [5] が提案されている．vAMT は，Intel AMT と同じインターフェースを提供するため．監視項目が限定される．

Copilot [6] は専用 PCI カードを用いて外部からカーネルの整合性を監視する．PCI カードがカーネルのメモリ内容をリモートホストに送信し，リモートホストでカーネルの改竄を検出する．攻撃者は PCI カードにアクセスできないため，監視結果を信頼することはできるが，専用の PCI カードが必要となるためコスト面に課題がある．

HyperCheck [7] はハイパーバイザの改竄を検出することができるシステムである．Intel 製 CPU のシステムマネジ

メントモード（SMM）と呼ばれる機能を用いて，NIC ドライバを動作させ，リモートホストにハイパーバイザのメモリを転送する．SMM で実行されるコードは BIOS のみが設定することができ，攻撃者はアクセスできないため信頼できる．また，CPU の標準機能であることからコストも低い．しかし，SMM での実行は低速であり，実行中はシステム全体が停止するなど，性能面に問題がある．

HyperSentry [8] は，IPMI を用いて監視対象ホストと通信し，SMM を用いてハイパーバイザの監視を行うシステムである．SMM を用いて割り込みを禁止し，ハイパーバイザ内で侵入検知システム（IDS）を安全に実行する．このシステムも HyperCheck と同様に実行は低速であり，実行中はシステム全体が停止するなど，性能面に問題がある．

RemoteTrans [9] は，監視対象の VM とは異なるリモートホストに IDS をオフロードし，ネットワーク経由で外部から VM を監視するシステムである．リモートホスト上の IDS はハイパーバイザとの間で暗号通信を行い，OS やディスク，ネットワークの整合性をチェックして異常を検出することができる．しかし，VM を用いたシステムに限定され，ハイパーバイザが攻撃を受けた場合には攻撃を検知できなくなる恐れがある．

7. まとめ

本稿では，異常検知の対象である OS 等を介さずに監視対象ホストの GPU とリモートホストとの間で直接通信を行い，検知結果を取得するシステム GRASS を提案した．GRASS は，GPUDirect RDMA を利用することにより，OS 等に異常が発生しても GPU および NIC が正常に動作していれば，検知結果を通知することができる．GRASS では，リモートホストが監視対象ホスト上の GPU メモリに直接アクセスし，双方でポーリングを行うことにより通信を行う．CUDA や RDMA ライブラリを用いて GRASS を実装し，実験結果から，監視対象ホストにおいて OS が異常停止しても正常に GPU との通信を行うことができることが確認できた．また，GPU 上の OS 監視システムの異常を検知するためのハートビートにかかる時間を測定し，ping と比較して十分に短い時間で死活監視を行うことができることを確認した．

今後の課題としては，GPUSentinel [1] を用いて取得した CPU 使用率やプロセス情報などをリモートホストに通知できるようにし，その性能を調べることが挙げられる．また，GPU で検知を行う代わりに，OS データをリモートホストに送信し，リモートホストで検知を行う手法の実装も計画している．その際には，データの送信量が増大するため，並列転送により転送性能を向上させる必要があると考えている．本稿ではカーネルパニック時の動作確認を行ったが，今後，様々な障害や外部からの攻撃時にも監視対象ホストとの通信を行うことができるかどうかを確認する必要がある．また，バックアップなど異常検知以外の用途への利用も検討している．

謝辞

本研究の一部は栢森情報科学振興財団の助成を受けたものである．

参考文献

[1] 尾崎雄一, 山本裕明, 光来健一：GPU を用いた OS レベルでの障害検知，情報処理学会研究報告，Vol. 2018-OS-142，2018.
[2] 飯田貴大, 光来健一：VM Shadow: 既存 IDS をオフロードするための実行環境，情報処理学会研究報告，Vol. 2011-OS-119，2011.
[3] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated and M. Silberstein：GPUnet: Networking Abstractions for GPU Programs，Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation，pp. 201-216，2014.
[4] Intel Corporation：https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/intel-active-management-technology.html.
[5] K. Kourai and K. Oozono：Virtual AMT for Unified Management of Physical and Virtual Desktops，In Proceedings of the 39th IEEE Computer Software and Applications Conference，2015.
[6] N. L. Petroni, Jr.，T. Fraser，J. Molina and W. A. Arbaug：Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor，Proceedings of the 13th USENIX Security Symposium，2004.
[7] J. Wang，A. Stavrou and A. Ghosh：HyperCheck: A Hardware-Assisted Integrity Monitor，Proceedings of International Symposium of Recent Advances in Intrusion Detection，2010.
[8] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang and N. C. Skalsky：HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity，Proceedings of the 17th ACM conference on Computer and communications security，pp. 38-49，2010.
[9] K. Kourai and K. Juda：Secure Offloading of Legacy IDSes Using Remote VM Introspection in Semi-trusted Clouds，In Proceedings of the 9th IEEE International Conference on Cloud Computing，2016.
[10] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis and S. Ioannidis：PixelVault: Using GPUs for Securing Cryptographic Operations，Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security，pp. 1131-1142，2014.