Vol.2022-HPC-183 No.9
2022/3/17

HBM2搭載FPGAのためのAddressable Cacheを用いたHPC向けメモリシステムの性能評価

藤田 典久[1,2]　小林 諒平[1,2]　山口 佳樹[2,1]　朴 泰祐[1,2]

概要：高性能計算の分野で Field Programmable Gate Array (FPGA) が新たなるアクセラレータとして注目されている．他のアクセラレータと比較して，FPGA は外部メモリ帯域が弱いという弱点があり，HPC における FPGA 利用の障壁のひとつである．最新の高性能 FPGA では，High Bandwidth Memory 2 (HBM2) を搭載する FPGA があり，これを使うことで HPC における FPGA 利用が広がると考えられる．しかしながら，FPGA は固定機能としてのメモリネットワークやキャッシュを持たず，HBM2 の性能を発揮できるメモリ回路を別途開発しなければならない問題がある．本稿では，我々が研究開発している HPC 向け HBM2 メモリシステムの実装と性能評価を示す．また，本システムを扱うための API の設計と実装についても報告を行う．FPGA は自律動作できるアクセラレータであり，本システムを扱う API はこの特徴を活かしたものである．

1. はじめに

Field Programmable Gate Array (FPGA) が新たなるアクセラレータとして注目されている．FPGA のプログラムにはハードウェア記述言語 (Hardware Description Language: HDL) が用いられていたた．HDL では，クロックサイクルレベルの粒度での記述をするため，FPGA を利用するにはハードウェアの知識が求められており，利用する障壁が高いという問題があった．近年，高位合成 (High Level Syntheiss: HLS) と呼ばれる開発環境が発展しつつあり，FPGA を利用することが容易になりつつある．HLS は，ソフトウェアで用いられる言語 (例えば C, C++, OpenCL) を用いてハードウェアの動作を記述するものであり，FPGA に関する知識がなくとも FPGA を扱えるようになりつつある．

これまで，FPGA メモリ帯域は他のアクセラレータとくらべて低く，性能のボトルネックとなることが多かった．DDR4 メモリを実装した FPGA ボードが一般的であり，DDR4-2400 メモリを 4 チャンネル持つ場合 76.8GB/s の帯域となる．NVIDIA 社の Graphics Processing Unit (GPU) である A100 80GB[1] が 2TB/s のメモリ帯域を持つことと比べると，約 25 倍の性能差が存在していた．

高性能な FPGA として High Bandwidth Memory 2 (HBM2) を搭載した FPGA チップがベンダーからリリースされ始めており，最大で 512GB/s のメモリ帯域を有する．前述した A100 GPU と比較すると 1/4 のメモリ帯域でしかないものの，10 倍やそれ以上の帯域差がある状況よりは改善されている．しかしながら，HBM2 は従来式メモリとは異なるアーキテクチャを有しており，HBM2 に適した FPGA メモリシステムが求められている．

本研究の目的は，HPC アプリケーションに適する HBM2 メモリシステムの提案と実装を行うことである．提案するシステムでは，FPGA に内蔵されているメモリを Addressable Cache として用いるそして，HBM2 とキャッシュ間をクロスバで接続することで，HBM2 が持つ多数のチャンネルの同時利用とメモリアクセスに対する柔軟性の両立を狙う．

本稿の貢献は以下の通りである．

- HBM2 搭載 FPGA のためのメモリシステムの提案を行う
- システムのプロトタイプを FPGA 上に実装し，実際に計算が行えることを示す
- 本システム向けに設計した API の設計と実装について示す

2. 関連研究

FPGA に搭載されている HBM2 を活用する研究としては，従来型のメモリ帯域が必要なアプリケーション [2] に加えて，ニューラルネットワークに適用した研究 [3], [4] が知られている．また，本研究会においても [5] の報告があ

[1] 筑波大学 計算科学研究センター
[2] 筑波大学 システム情報系

る．これらの研究では，演算回路が特定の HBM2 メモリ Channel に接続されており，それ以外の channel にアクセスすることはできない．

[6] で Choi らは Xilinx 製 FPGA ボードである Alveo U280 を用いて，Bucket Sort と Merge Sort を実装し，性能評価を行った．Vivado HLS で実装した Sorter 部分と彼らが HBM Connect と呼んでいるメモリネットワークが実装されている．HBM Connect によって Sorter 回路とメモリ間が接続されており，Sorter 回路が全ての HBM2 Channel にアクセスできる．ボード開発環境の制限により，ボードにある HBM2 の半分でしか性能評価ができていないものの，理論ピークの 9 割程度の性能が達成されている．性能の代償として，HBM Connect を構成するために回路リソースを消費しており，その分アプリケーションのために使える回路リソースが減っている．

本稿で提案するシステムは，HBM2 を外部メモリとして用いることを前提としており，HBM2 が持つ数十のメモリチャンネルを前提として設計を行う．また，アプリケーションと HBM2 メモリを直接接続するのではなく，間にメモリネットワークおよび FPGA に内蔵されているメモリをキャッシュとして配置するところに本研究の新規性がある．このようなメモリシステムを実装することで，性能とアプリケーションの自由度のバランスがとれた実装が行えるものと考えている．

3. これまでの研究

本研究会において，我々は Intel FPGA における HBM2 のメモリシステムおよび性能評価に関する報告を行った [7], [8], [9]．[7] は，Intel FPGA における HBM2 の性能評価を行った報告である．ここでは，FPGA に搭載されている HBM2 コントローラを直接制御するマイクロベンチマークを FPGA 上に実装し，HBM2 の基本的な性能について評価した．メモリコントローラの動作周波数が仕様上の最大値より低くしか動作できていない問題があったものの，動作周波数に応じた想定通りの性能が得られることがわかった．しかしながら，HBM2 が持つ多数のメモリチャンネルを並列に扱うことが，FPGA で HBM2 を利用する際の大きな問題であることを明らかにした．

[8], [9] では，より実用的なメモリシステムの実現を目指し，DMA コントローラやクロスバを組み込んだメモリシステムを提案し，そのプロトタイプ実装を示した．この実装では，FPGA が持つ 32 のメモリチャンネルうち，2 チャンネル (全体の 1/16) しか実装できておらず，全チャンネルに対する実装が課題として残った．また，性能面では 71.0%の効率しか得られず，システム全体の最適化，特にホストからの制御について最適化が十分に行えていなかった．本稿では，これら先行研究を踏まえ，[8], [9] で実装したシステムをさらに改善・発展させたものを示す．特に，どのように FPGA に対して指示を与えるのか，API の設計と実装について示す．

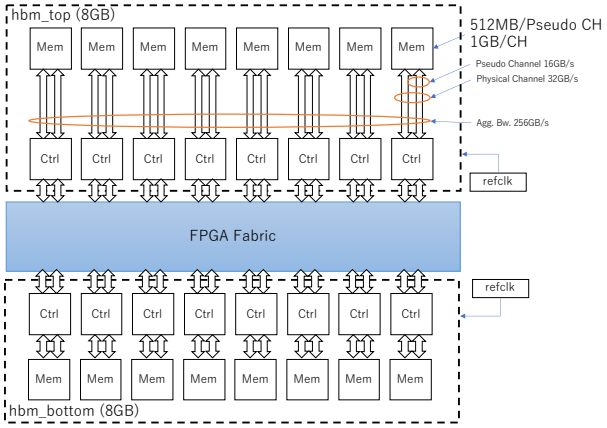

図 1: HBM2 メモリの構造.

4. Intel FPGA における HBM2 の構造

Intel FPGA における HBM2 については，我々の先行研究 [7] で既に報告している．したがって，内容の重複を避けるため，本稿ではその中でも重要な部分について述べるにとどめる．詳細な構造や，メモリ単体としての性能について興味がある方は [7] を参照して頂きたい．

図 1 に Intel Stratix 10 MX FPGA における HBM2 のアーキテクチャを示す．2 つの HBM2 ダイが接続され，容量は 8GB (4GBx2) もしくは 16GB (8GBx2) であり，メモリ帯域 (Aggregated) は最大で 512GB/s に達する．HBM2 は，遅いメモリを多数並列に駆動することで高い性能を実現している．図 1 からわかるように，1 チャンネルあたり 16GB/s の帯域しかなく，数チャンネルだけ利用するのであれば，DDR4 のような従来式のメモリを用いた方が帯域が高い．したがって，多数のメモリチャンネルを並列に駆動することが求められる．

FPGA では，固定機能としての高性能なキャッシュやインターコネクトといったメモリシステムは存在せず，必要に応じて FPGA 内に回路として実装しなければならない．HPC アプリケーションをターゲットとして考えると，実プリケーションにおいてはメモリアクセスは複雑であり，ある程度の自由度があるメモリシステムは必須であると考える．FPGA で CPU などと同様の回路を構築することはもちろん可能であるが，複雑なシステムは回路リソースを多く必要とする．本来，回路リソースはすべて演算に使いたいところであるため，メモリシステムにすべてのリソースを割り当てられる訳ではない．したがって，我々は性能とコストのバランスをとったメモリシステムを FPGA に実装する必要があると考えている．

5. 提案するメモリシステム

5.1 概要

本稿で提案するメモリシステムの基本的な設計思想は [8], [9] のものをベースとしている．図 1 で示した様に，Intel FPGA は最大で 32 個の HBM2 メモリチャンネルを持つ．それらのメモリチャンネルとアプリケーションを直接説した場合，複雑なアプリケーションでそれらを効率よく扱うプログラミングは非常に困難であり，事実上不可能であると我々は考えている．加えて，HBM2 は DRAM であるため，ページを跨ぐランダムアクセスでは高い性能を発揮しづらい．

以上の理由から，我々は Block RAM (BRAM) をキャッシュとして HBM2 とアプリケーションの間に配置するのが良い実装方針であると考えている．BRAM は FPGA に内蔵されているメモリであり，広帯域・低レイテンシでアクセスが可能であるが，内蔵であるがゆえに容量が限られている．BRAM をキャッシュとして使うことで，HBM2 と BRAM 間の転送はバルク転送となり，ある程度の塊でデータ転送を行えるようになり性能低下を避けられる．また，BRAM であれば，高速にランダムアクセス可能であるため，アプリケーションの実装が容易になる．

一般的にキャッシュと言うと，ハードウェアで自動で制御される実装が多い．しかしながら，FPGA でそのような実装を行うと，本来演算で使うべきハードウェアリソースをキャッシュシステムで消費してしまう．したがって，HBM2 と BRAM キャッシュ間のデータ転送はプログラマが手動で指定する方式をとる．この方式は，プログラミングコストが増加するという問題があるが，アクセラレータを利用してプログラムを記述する場合，CPU とアクセラレータ間のデータ転送を手動で記述することは一般的である．したがって，データ転送を手動で管理するプログラミングモデルの考え方は受け入れられていると考えている．

5.2 ハードウェア構成

図 2 にメモリシステムの概要図を示す．ただし，本システムは開発中のため，現時点では 8 メモリチャンネル分しか実装ができていない．そのため，残りの 3 メモリグループは実装予定として，図 2 では点線で示す．今後，点線で示された領域にも左下と同様の構造を追加する予定である．

本システムは, PCI Express (PCIe) IP, Local Store (LS), HBM2 Memory Controller，Crossbar から構成され，8 つの HBM2 メモリチャンネルを 1 グループとして扱う．グループ分け設計を行う理由は，クロスバの回路コストが $O(n^2)$ であるためポート数を増やすことが難しいことと，HBM2 と FPGA 内部ファブリックの間の接続が 8x4 に分けられているためである．そして，メモリグループ間は全体を司る全体クロスバによって接続される．また，ホストと FPGA 間の通信を担う PCIe バスから来る通信も，全体クロスバに接続され，それぞれのメモリグループに転送されていく．

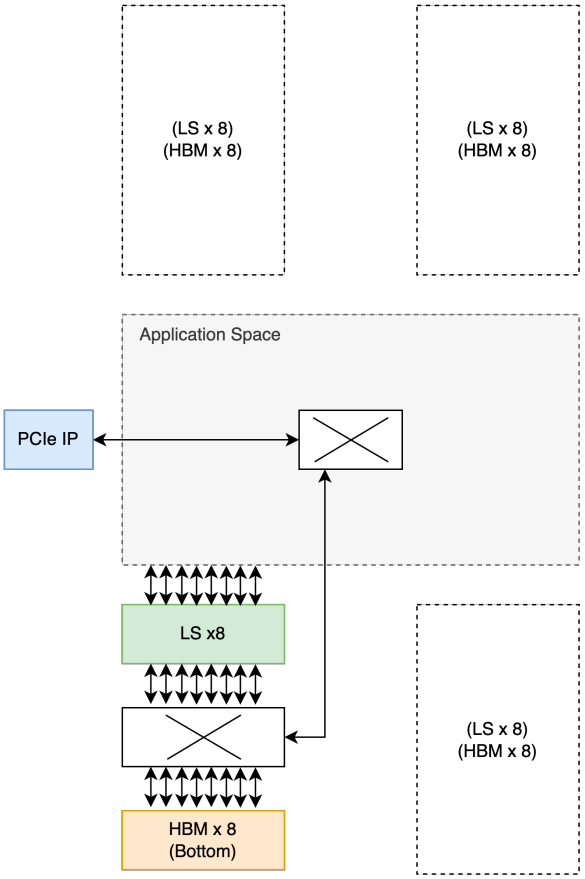

図 2: 提案メモリシステムの概要図.

本システムの大部分は Chisel[10] で記述されている．Chisel は Scala 上に構築される Domain Specific Language (DSL) であり，Verilog HDL と同等の RTL の抽象度で記述する．Chisel によって生成された Verilog HDL を Intel 社が提供する FPGA 開発環境を用いて回路合成を行う．ただし，非同期な動作が必要な箇所や Intel が提供する IP を用いる箇所は Chisel での記述は向かないため，Verilog HDL で直接記述している．

5.3 クロスバ

図 2 は概要を示した図であり，LS と HBM 間にあるクロスバは 17 ポートを持つように見える．しかしながら，そのような多ポートのクロスバを実装することは現実的ではないため，実際の構成は**図 3** にあるように，9 ポートクロスバを 2 台として実装されている．なお，図において，赤矢印は Crossbar 1 への接続を，黒矢印は Crossbar 2 への接続をそれぞれ表す．この場合，LS → HBM，HBM → LS 間はフル帯域を有するが，LS 同士や HBM 同士の直接通信は行えず，全体クロスバ経由となり帯域が制限される．し

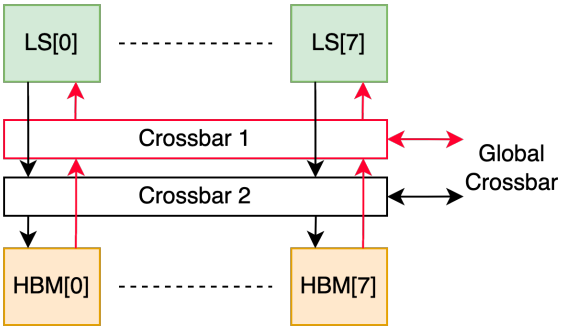

図 3: クロスバの構成．赤矢印は Crossbar 1 に対する，黒矢印は Crossbar 2 に対する接続を表す．

かしながら，LS 同士や HBM 同士の通信における転送性能は重要ではなく，制御用パケットが流れる程度であり，性能上のボトルネックにはならない．

本システムのクロスバは 613bit のバス幅を持つ (101bit のヘッダ部+512bit のデータ部)．Virtual Output Queue (VOQ) を実装しており，異なる宛先への通信は，混雑状況に応じて追い越しが可能である．また，スケジューリングアルゴリズムとして DRRM[11] を用いるが，動作周波数を高めるために，スケジューリングは 2 サイクルかかる実装としている．あるサイクルで開通した経路は，次のサイクルでも必ず通信できる．送るデータが 1 サイクル分しかない場合でも必ず同じ経路が選択されるため，通信パターンによっては通信効率が低下する．しかしながら，本システムでは主にバースト転送を扱うため，スケジューリングに 2 サイクルかかることの性能に対する影響は小さいと考える．クロスバのスケジューリングアルゴリズムは多数提案されており，他に PIM[12]，iSLIP[13]，EDRRM[14]，Combined Parallel Round Robin Arbiter[15] などが知られている．アルゴリズムの複雑さが低く高い動作周波数でも利用できることと，入力間の公平性を担保できることから DRRM を用いる．

5.4 LS の構成

図 2 において “LS” と表されているコンポーネントの詳細を**図 4** に示す．それぞれの LS には，キャッシュ用 BRAM だけでなく，制御用のコンポーネントも実装されている．各 LS には，128KB のキャッシュメモリ，Direct Memory Access Controller (DMAC) が 2 基，RISC-V[16] コアが搭載されている．また，キャッシュメモリはアプリケーションカーネルとも接続され，アプリケーションからのアクセスも行える．

DMAC は 2 基あるがメモリアクセスの方向が固定されており，それぞれ LS → HBM と HBM → LS 方向専用である．2 つの DMAC はそれぞれ独立しており，同時に動作することができる．

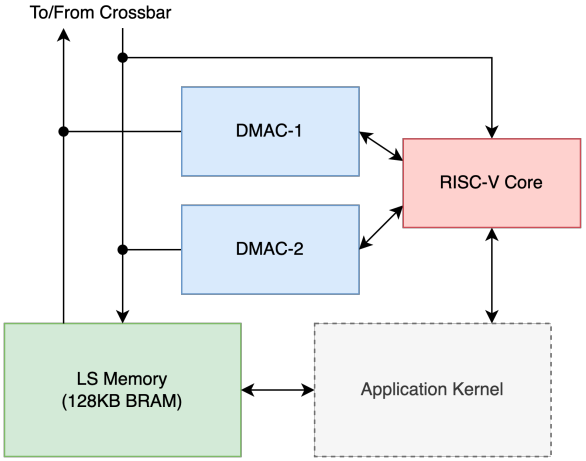

図 4: ls.png

LS に実装されている RISC-V のコアはシステム制御に用いる．ただし，このコアは RV32I で定義される命令のうち必要な一部のみを実装したものであり，RISC-V の仕様に完全に準拠したものではない．また，このコアはあくまで制御用であり，これ自体に演算性能は求めないため，2 サイクルに 1 回のみ命令が実行される設計とし，リソースの節約をしている．

本システムでは，この RISC-V コアを用いて，システム全体の制御を行う．LS ⇔ HBM のデータ転送に加えて，カーネルの起動に関する制御も RISC-V が担う．本システムでは，FPGA の動作は RISC-V のコードとして表現される．したがって，柔軟な動作を FPGA にさせることができ，デバイス上での繰り返しや分岐を実現できることが本システムの大きな特徴である．なお，このシステムを扱う API は 6 章で述べる．

6. 制御用 API の設計と実装

6.1 概要

高性能計算の分野においては，CUDA や OpenCL がアクセラレータの制御用 API としてよく用いられる．これらの API では，アクセラレータはホスト CPU から PCIe などのバスを通じて制御される動作モデルが用いられる．言い換えると，アクセラレータは単体で動作することはできず，常に CPU からの指示を受けて演算やデータ転送などの動作を行うものである．一方，FPGA はその再構成可能な点を活かして，自律動作できるアクセラレータにすることも可能である．

本システムでは，5 章で述べたように，データを FPGA 内にあるバッファ領域 (LS) に転送し，その上で演算を行う．LS には FPGA 内にある BRAM を用いるため高い性能を得られるが，容量に限りがあり，大きな計算を行う際は小さな複数の計算ステップに分解して行う必要がある．HBM2 の動作周波数と BRAM 容量から計算すると，メモ

リバンド幅な計算を行う際は部分計算 1 ステップにかけられる時間は約 20μs となる．加えて，転送時間を隠蔽するためにデータ転送と演算をオーバーラップしダブルバッファリングを行うと考えると，1 ステップあたり約 10μs の時間となる．ホスト CPU から 10μs 間隔で PCIe バスを通して各ステップの動作に関する制御を行うことは非現実的である．よって，本システムでは FPGA が細粒度かつ自律的に動作できるように設計開発を行う．

6.2 RISC-V コアを用いたシステム制御

本システムでは，FPGA 内に実装された RISC-V コアが演算やデータ転送の制御を行う．この RISC-V コアが実行するコードは，ホストで実行されている専用の C++ API を通じて RISC-V コードを実行時に生成する．そして，生成したコードを RISC-V コアの命令メモリに転送して実行する．

本システムでは，メモリデータ転送と演算をオーバーラップしてレイテンシやオーバーヘッドを隠蔽するために，ダブルバッファリングを行うことを前提としている．ダブルバッファリングの動作記述を容易にするために，CUDA における cudaStream のような実行形態を採用している．各コントローラコアは最大で 2 つの Stream を実行することができる．同時に動作する Stream は 1 つのみであるが，DMAC の動作完了やカーネルの実行完了時に発生する割り込みを用いて実行する Stream を切り替えられる．

6.3 RISC-V コード生成

データ転送や制御に関する記述は，ホスト CPU で実行されるコードに記述されている (**図 5**-(1))．これらの記述は，ホスト CPU ではなく，FPGA 側で実行されなければならない．そのため，ホスト用コード上にある FPGA で実行されるべき API 呼び出しを構文木として抽象化し，中間形式である Single Static Assignment (SSA) 形式を通じて，最終的に RISC-V コードを生成する．

Abstract Syntax Tree (AST) を生成には，C++の Expression Template (ET) と呼ばれるテクニックを用いる (図 5-(2))．ET の処理には，Boost C++ Library[17] に含まれている Boost.YAP library[18] を用いる．Boost.YAP は C++の Operator Overload を用いて AST を生成するためのライブラリであり，木の情報を型として保持できる．また，構築した AST を評価 (eval) するためのインターフェイスも持つ．

次に，ET を用いて生成した AST を SSA 形式に変換する (図 5-(3))．次段階で SSA の最適化のために LibFirm[19] を用いるため，LibFirm の SSA 構築用 API を用いて SSA を構築する．本システム専用の一部処理は，LibFirm のカスタム命令として実装を行い，SSA 形式に含めている．Control Flow Graph (CFG)，依存関係解析，CFG に基づく ϕ 関数の挿入，Dead Code Elimination (DCE)，定数 fold，整数積とビットシフト変換といった SSA 形式上で行う最適化を LibFirm を用いて適用する (図 5-(4))．

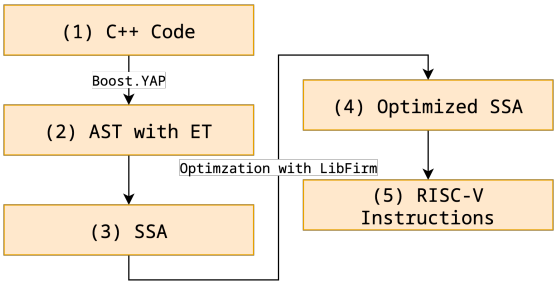

図 5: API を用いた RISC-V コード生成の流れ．

最後に，SSA 形式から RISC-V の命令を生成する (図 5-(5))．レジスタ割当には [20] のアルゴリズムを用いる．CFG から作られた Dominator Tree (支配木) の Perfect Elimination Order (PEO) の順序でレジスタを割り当てる．Dominator Tree は LibFirm を用いて求めることができる．なお，本システムにある RISC-V コアは，データ用メモリを持たずレジスタスピルが行えない．したがって，レジスタが不足した場合はエラーとする．

6.4 シミュレーション環境

一般的に，FPGA の回路合成には長い時間を要する．Intel Stratix 10 FPGA の全体を使う複雑な回路では 24 時間を超えることもある．これではデバッグが困難であるため，FPGA のシステムを開発する際は Register Transfer Level (RTL) のシミュレータを用いることが一般的である．また，シミュレータであれば，FPGA 内部の信号変化をログとして保存することができるため，デバッグが容易である．FPGA 実機では情報収集手段が限られデバッグが困難である．

本システムでは，Verilator を用いたシミュレーションをサポートしている．Verilator は Verilog HDL で書かれたハードウェアを C++でモデル化し，CPU 上で実行できるようにするソフトウェアである．Verilator で変換された C++モデルは非常に高速に動作し，C++であることから既存のコードに組み込むことが容易でにできる．ただし，Verilator は FPGA ベンダーが提供するライブラリを扱うことができないため，高位合成で生成された Verilog HDL コードが扱えない．そのため，将来的には Model SIM などの RTL シミュレータをサポートする予定である．

本システム用の API は実機環境でもシミュレータ環境でも同じものが使えるように設計を行う．CPU が FPGA と通信する際に使うプロトコルは PCIe バス経由の Memory Mapped I/O (MMIO) のみであり，PCIe 割り込みは用いていない．したがって，**図 6** にあるように MMIO アクセスを抽象化し，実機環境とシミュレータ環境を実行時のオ

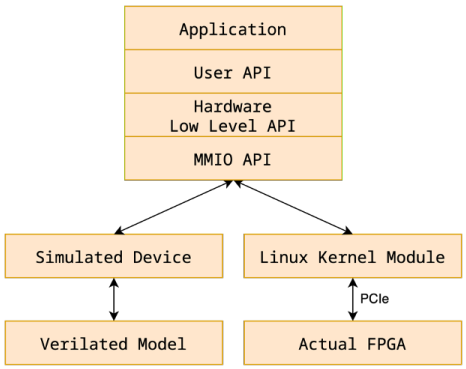

図 6: ソフトウェアスタックの階層構造.

表 1: 評価環境 (PPX)

CPU	Intel Xeon E5-2690 v4 × 2
CPU Memory	DDR4 2400 MHz 64 GB (8 GB × 8)
Infiniband	Mellanox ConnectX-4 EDR
Host OS	CentOS 7.9
Host Compiler	gcc 9.1.0
FPGA	インテル Stratix 10 MX FPGA 開発キット (1SM21CHU2F53E1VG)
FPGA Memory	HBM2 16GB (8GB × 2)
Quartus Prime	Quartus Prime Pro 20.4.0.72

プションで切り替えられるように設計をしている．実機動作時は，専用 Kernel Module を通して FPGA の PCIe アドレス空間を `mmap(2)` し MMIO を行う．シミュレータ動作時は，仮想的に作成した PCIe Transport Leyer Packet (TLP) をシミュレータ上に構築されたハードウェアに対して送信して MMIO を模擬する．ただし，すべての MMIO アクセスをシミュレーションすると時間がかかりすぎるため，HBM2 に対するアクセスはハードウェアのシミュレーションをバイパスし，シミュレータ上に確保している HBM2 を模すメモリ領域へ直接アクセスして高速化するオプションを有する．

7. 性能評価

7.1 評価環境

本稿では，筑波大学計算科学研究センターで運用中の実験クラスタ Pre-PACS-X (PPX) を性能評価に用いる．PPX は開発用クラスタであるため，様々な仕様のノードが混在しているが，その中の 1 ノード (**表 1**) にインテル Stratix 10 MX FPGA 開発キット [21] を搭載し，性能評価を行う．開発キットは，1SM21CHU2F53E1VG を搭載しており，Speedgrade-1 のロジック部と，16GB の HBM2 を搭載した MCM となっている．Intel FPGA の HBM2 帯域はチップの Speedgrade によって決まり，Speedgrade-1 の場合，最大で 512GB/s（メモリ動作周波数 1000MHz）まで対応する．

今回の実験では HBM2 メモリ周波数を 1000MHz, HBM2 FPGA 側の制御ロジックを 400MHz，他のシステム部分を 250MHz で駆動する．HBM2 のコントローラは固定機能として実装されているが，FPGA 側にもコントローラとの通信を行う制御ロジックが実装される．過去の研究 [7] で，この部分を 500MHz で駆動するには細かいチューニングが必要なことがわかっているため，HBM2 制御ロジックは 400MHz 動作とする．なお，HBM2 メモリコントローラは固定機能として実装されているため，仕様上の最大周波数である 1000MHz 駆動とする．

Stratix 10 FPGA で 400MHz や 500MHz で動作する回路を設計することは難しく，特に C 言語や OpenCL を用いる高位合成環境では達成困難である．アプリケーションが HBM2 メモリよりも動作周波数が遅い場合でもメモリ帯域を使い切れるように，クロスバのデータ部バス幅は HBM2 のデータ部バス幅 の倍である 512bit にし，かわりに動作周波数を半分にする．本稿では HBM2 コントローラは 400MH 動作としているが，最大動作周波数が 500MHz 駆動であるため，他のシステム部分をその半分の 250MHz 動作とする．

7.2 評価用プログラム

性能評価に用いたプログラムを擬似コードを**図 7** に示す．ホストが FPGA 上の配列に対して乱数を書き込み，それを FPGA がビット反転させ，最後にホストで値が正しいか検証するプログラムである．本稿では，メモリシステムの性能評価に注目するため，FPGA 上で実行されるカーネル部は Verilog HDL で実装する．高位合成を用いると，生成されたハードウェアの構造を把握しづらくなり，性能ボトルネックがどこにあるのかの判断が難しくなる．なお，今後実アプリケーションを本システムで動作させる際には，Intel HLS Compiler (i++) を用いてアプリケーションカーネルを記述する予定である．i++と HBM2 を組み合わせて支障なく利用できることは，過去の研究 [8] で明らかにしている．

図 8 に，ホスト側のコードの一部抜粋を示す．この部分が本研究で提案する API を用いている箇所である．LS あたりのキャッシュメモリサイズが 128KB であるため，16K 要素 (64KB) を単位として 2 Stream を用いて反復を行う．ここで，`var` 型変数や `array_view` 型変数は FPGA 上の領域を示すハンドルである．`var` 型変数を用いることで，転送や計算の範囲を示すことができる (16, 28 行目)．また，`array_view` 型変数に対して代入 (19, 26 行目) することで，FPGA 上でのデータ転送を記述できる．そして，`stream_for` (18 行目) を用いることで，デバイス上での反復を表現できる．通常の `for` 文と同じように動作するが，`stream_for` の実態は C プリプロセッサのマクロであるた

```
uint32_t data[N];
/* Host */
for (int i = 0; i < N; i++) {
    data[i] = random();
}

/* FPGA */
for (int i = 0; i < N; i++) {
    data[i] = ~data[i];
}

/* Host */
for (int i = 0; i < N; i++) {
    verify(data[i]);
}
```

図 7: 実験に使用したプログラムの擬似コード．random() 関数は 32 ビット整数の疑似乱数生成関数，verify() 関数は引数の値が生成した乱数列と一致しているかを検証する関数．

表 2: 全体のリソース消費量と FPGA 全リソースに対する使用率．

ALM	Registers	M20K	DSP
97,602	174,409	1,243	120
13.89%	6.20%	18.15%	3.03%

め，区切り文字は “;” ではなく “,” である．

7.3 リソース消費量

システム全体の FPGA のリソース消費量を**表 2** に，それの詳細値を**表 3** に示す．Adaptive Logic Module (ALM) は論理回路を構成する Look Up Table (LUT) とレジスタを含むモジュール，M20K は FPGA 内蔵 BRAM，Digital Signal Processor (DSP) は整数乗算器を表す．表 3 の値は，Fitter Placement のレポートファイル (.fit.place.rpt) から求めた．ただし，Intel FPGA の合成ソフトウェアはモジュールを跨いだ最適化を行うため，表 3 の値は変動する可能性があるが，おおまかなリソース消費量を把握するには十分である．

表 2 から FPGA 18%分のリソースを消費していることがわかる．表 3 からわかるように，今回の実験で用いるカーネルは単純なビット演算しか行わないため非常に小さく，表 2 のリソースのほとんどはメモリシステムで消費していると考えて良い．最も使われているリソース種別は M20K であるが，18%中キャッシュ用のメモリが 6.54%を消費している．この分を除くと，ALM が最も使われているリソースとなり，メモリの制御系＋ネットワークの消費量で ALM 13.89%消費している．

```
auto ctrl = device->open_control(id);
auto ls = device->open_ls(id);
device_array buffer1 = array_alloc_1d(ls,
    CHUNK_BYTE);
device_array buffer2 = array_alloc_1d(ls,
    CHUNK_BYTE);

ctrl->begin_config();

for (int s = 0; s < N_STREAMS; s++) {
  define_stream(ctrl, s) {
    var i;
    var start;
    array_view<uint32_t> buffer(s == 0 ?
        buffer1 : buffer2);
    array_view<uint32_t> data(d_data);

    i = 0;
    start = NOC_LOCAL_STREAM_ID(id, s) *
        CHUNK_SIZE;

    stream_for(i = 0, i < N_CHUNK / N_WORKERS,
        i = i + 1) {
      buffer(0, CHUNK_SIZE) = data(start,
          start + CHUNK_SIZE);

      if (s == 0) {
        kernel(0, 0, LOOP_LEN);
      } else {
        kernel(LOOP_LEN, LOOP_LEN, LOOP_LEN);
      }
      data(start, start + CHUNK_SIZE) = buffer
          (0, CHUNK_SIZE);

      start = start + N_STREAMS * NOC_GRP_SIZE
          (id) * CHUNK_SIZE;
    }
  }
}

ctrl->stream_start_and_sync();
ctrl->end_config();
```

図 8: 本システム用の API を用いたコードの一部．ただし， N_CHUNK, CHUNK_SIZE はデータ量から求められる定数，N_WORKERS はシステム全体で何個の Stream が動作するかを表す定数である．d_data は HBM2 上のメモリを表すハンドルを表す．

7.4 メモリ転送の不具合

本稿の実装はバグがあり，一部のデータ転送が正常に動作していないことが判明しており，現在デバッグを行っている．本実験では，ホストで乱数を用いて初期値を与え，FPGA でビット反転をし，ホストで結果を読み出して比較している．しかしながら，正常に動作していない場合は，一部領域が初期値から変化していないことを確認している．

問題が発生した場合は，Write リクエスト 1 つに相当す

表 3: FPGA リソース消費量の詳細.

	ALM		Register		M20K		DSP	
HBM2 Controller	6653.5	0.95%	20829	0.74%	248	3.62%	0	0.00%
Global Crossbar	4323	0.62%	10467	0.37%	62	0.91%	0	0.00%
LS-HBM Crossbar (x2)	44659.7	6.36%	72338	2.57%	280	4.09%	0	0.00%
Kernel (x8)	435.6	0.06%	1017	0.04%	0	0.00%	0	0.00%
LS (x8)	27636.6	3.93%	32124	1.14%	448	6.54%	120	3.03%
MC (x8)	8005.8	1.14%	22891	0.81%	192	2.80%	0	0.00%
PCIe	2251.2	0.32%	4638	0.17%	13	0.19%	0	0.00%
Sys. Mgr. + Perf. Counter	2956.1	0.42%	8185	0.29%	0	0.00%	0	0.00%
others	244.9	0.03%	903	0.03%	0	0.00%	0	0.00%

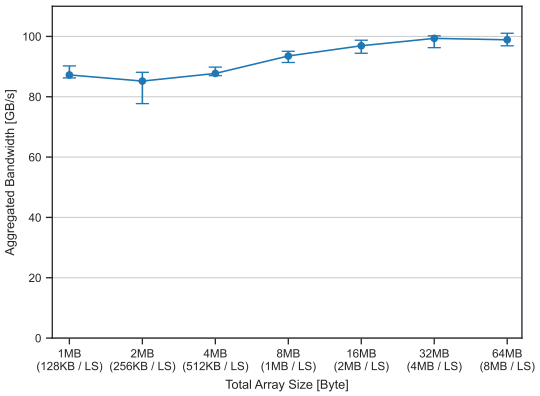

図 9: 性能評価の結果.

る 64byte の領域でデータが初期値から変化していない．したがって，Write リクエストが正しく発行できていないと思われる．LS とパフォーマンスカウンタでは，HBM2 Controller へのリクエスト数と完了レスポンスの数を数えており，何かしらのリクエストが発行できてることは確認しており，結果が正しくない場合であっても，書き込みリクエストは HBM2 Controller に届いているとみられる．よって，何らかの不具合によって書き込みリクエストが正しいアドレスに発行されていないものと推測し，デバッグを行っている．

なお，64MB の領域にアクセスした場合でも，データ異常が発生する領域は 1〜4 リクエスト (64 Byte〜256 Byte) 程度であり発生率は低い．また，必ず不具合が発生するというわけではなく，全結果が問題ないケースも多い．次節で示す評価結果は，すべての結果が正常に転送された場合の結果を示す．

7.5 評価結果

メモリ転送性能は各メモリバスに接続したパフォーマンスカウンタを用いて FPGA 側で計測を行う．ホストから計算開始の命令を受信してから，すべての LS が計算完了状態になるまでの時間をクロックサイクル単位で計測を行う．LS は，自身が発行したすべての Write, Read リクエストの完了応答が HBM2 Controller から帰ってきてくるのを待機してから，完了フラグを立てる．なお，パフォーマンスカウンタは 250MHz で駆動されているため，4ns の粒度で計測が行える．

配列サイズを 1MB から 64MB まで変化させたときの性能評価の結果を**図 9** に示す．ベンチマークプログラムを 10 回実行して性能を測定し，エラーバーは 10 回の実行中の最大値・最小値を表し，折れ線は中央値を表す．7.4 節で述べたように，本システムはバグがあり，計算が正しく行えない場合がある．そのため，10 回の実行を 1 セットとして測定用のシェルスクリプトを作成し，1 セット内の 10 回全ての実行で問題がないケースを結果として示す．

図 9 より，配列サイズ 64MB のときが最も性能がよく，最大で 101.05GB/s の帯域が得らていることがわかる．理論ピークが 102.4GB/s (= 400MHz × 256bit × 8) であるため，この結果は 98%の効率に相当する．

8. 考察

キャッシュ用のメモリを除くと，メモリの制御系＋ネットワークで ALM 13.89%消費している．ほとんどの回路リソースはメモリチャンネル数に応じて増えるため，32 チャンネルへ実装を拡大することを考えると，ほとんどのリソースは 4 倍になると考えられる．したがって，32 チャンネル実装時のリソース消費量は 55%と見積もれる．今後，実用的なアプリケーションを実装することを考えると，いくらメモリ性能を使い切るためとはいえ，半分を超えるリソースをメモリネットワークで消費するのは問題があると考える．

ALM の消費量が多いモジュールはクロスバである．ALM は特に出力データを選択する Multiplexer で消費される．Multiplexer は基本的かつ必須の機能であるため，このリソース消費量を最適化することは難しい．したがって，スループットを維持しつつリソース消費量を減らすには，動作周波数をあげてデータバス幅を減らすしかないと考えられる．

HBM2 は DRAM の一種であるため，リフレッシュに伴う停止期間が存在するにもかかわらず，図 9 より最大で 98%の効率が達成されている．HBM2 Controller IP のドキュメント [22] によると，ハード IP として実装されている HBM Controller はメモリクロックの半分の周波数で動作するとされており，今回の回路では 500MHz で動作している．FPGA 側に実装され，コントローラとデータのやり取りをする部分は 400MHz で動作しており，理論ピーク性能は 400MHz 側のスループットに性能が律速されている．しかしながら，リフレッシュ動作は高速で動作している HBM コントローラ側で実行されるため，リフレッシュ動作が隠蔽され，400MHz 動作の理論ピークに近い性能が得られていると考えられる．

今回の実験では 1/4 のメモリチャンネルしか用いることができていないが，得られた 101GB/s の帯域は既に従来型メモリである DDR4 を採用したの FPGA の理論ピーク性能 (DDR4-2400 × 4ch: 76.8GB/s) を上回っており，HBM2 が非常に高い性能を持っていることを示せている．

9. まとめと今後の課題

本稿では，HBM2 搭載 FPGA のための Addressable Cache を用いたメモリシステムの提案と実装を行った．本システムでは，FPGA 内蔵メモリを使うため，キャッシュサイズが小さく細粒度の制御が求められる．この問題を解決するために，RISC-V コアを制御用 CPU として FPGA に搭載し，前述した API を用いて RISC-V コードを生成・実行する API の実装を行った．これによって，繰り返しや分岐を含む処理を FPGA に実行させることが可能になり，ホストの制御がなくとも複雑な処理を実行できる．メモリバンド幅を測定するプログラムを用いて提案システムの性能を測定し，最大で 98%の効率が達成できていることを示した．しかしながら，7.4 節で述べた不具合が残っており，今後も研究開発を進めていく．今回は単純なマイクロベンチマークでの性能評価にとどまるが，幅広いベンチマークやアプリケーションを対象に性能評価を行いたいと考えている．

Intel FPGA の HBM2 は全体で 32 のメモリチャンネルを持つが，現時点では 8 チャンネルに対してしか回路を実装できておらず，32 チャンネル実装が今後の課題である．しかしながら，8 章で考察したように，現状の設計のまま 32 チャンネル分のシステムを実装すると，FPGA リソースの半分を消費してしまうと予測される．このままでは FPGA に搭載できるアプリケーションの規模が制限されてしまうがめ，動作周波数の最適化を行いクロスバのリソース消費量を軽減していくことが今後の課題の一つである．

謝辞 本研究の一部は，「高性能汎用計算機高度利用事業」における課題「次世代演算通信融合型スーパーコンピュータの開発」及び，文部科学省研究予算「次世代計算技術開拓による学際計算科学連携拠点の創出」による．本研究の一部は, JSPS 科研費 21H04869 の助成を受けたものである．また，本研究の一部は，「Intel University Program」を通じてハードウェアおよびソフトウェアの提供を受けており，Intel 社の支援に謝意を表する．

参考文献

[1] NVIDIA: VIDIA A100 | NVIDIA, `https://www.nvidia.com/ja-jp/data-center/a100/`.
[2] Meyer, M., Kenter, T. and Plessl, C.: Evaluating FPGA Accelerator Performance with a Parameterized OpenCL Adaptation of Selected Benchmarks of the HPCChallenge Benchmark Suite, *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pp. 10–18 (online), DOI: 10.1109/H2RC51942.2020.00007 (2020).
[3] Venkataramanaiah, S. K., Suh, H. S., Yin, S., Nurvitadhi, E., Dasu, A., Cao, Y. and Seo, J. S.: FPGA-based Low-Batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory, *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–8 (2020).
[4] Kuramochi, R. and Nakahara, H.: An FPGA-Based Low-Latency Accelerator for Randomly Wired Neural Networks, *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 298–303 (online), DOI: 10.1109/FPL50879.2020.00056 (2020).
[5] 塙 敏博，三木洋平：宇宙物理アプリケーションのための FPGA 演算オフローディングの検討，研究報告ハイパフォーマンスコンピューティング（HPC）, 2020-HPC-172 (2019).
[6] kyu Choi, Y., Chi, Y., Qiao, W., Samardzic, N. and Cong, J.: HBM Connect: High-Performance HLS Interconnect for FPGA HBM, *FPGA '21* (2021).
[7] 藤田典久，小林諒平，山口佳樹，朴 泰祐：HBM2 メモリを持つ FPGA ボードの性能評価, 研究報告ハイパフォーマンスコンピューティング（HPC）, 2021-HPC-178 (2021).
[8] 藤田典久，小林諒平，山口佳樹，朴 泰祐：FPGA における HPC アプリケーション向け HBM2 メモリシステムの提案と実装，研究報告ハイパフォーマンスコンピューティング（HPC）, 2021-HPC-180 (2021).
[9] Fujita, N., Kobayashi, R., Yamaguchi, Y. and Boku, T.: HBM2 Memory System for HPC Applications on an FPGA, *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 783–786 (online), DOI: 10.1109/Cluster48925.2021.00116 (2021).
[10] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J. and Asanović, K.: Chisel: Constructing hardware in a Scala embedded language, *DAC Design Automation Conference 2012*, pp. 1212–1221 (online), DOI: 10.1145/2228360.2228584 (2012).
[11] Chao, J.: Saturn: a terabit packet switch using dual round robin, *IEEE Communications Magazine*, Vol. 38, No. 12, pp. 78–84 (online), DOI: 10.1109/35.888261 (2000).
[12] Anderson, T. E., Owicki, S. S., Saxe, J. B. and Thacker, C. P.: High-Speed Switch Scheduling for Local-Area Networks, *ACM Trans. Comput. Syst.*, Vol. 11, No. 4, p. 319–352 (online), DOI: 10.1145/161541.161736 (1993).

[13] McKeown, N.: The iSLIP scheduling algorithm for input-queued switches, *IEEE/ACM Transactions on Networking*, Vol. 7, No. 2, pp. 188–201 (online), DOI: 10.1109/90.769767 (1999).
[14] Li, Y., Panwar, S. and Chao, H.: The dual round robin matching switch with exhaustive service, *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie*, pp. 58–63 (online), DOI: 10.1109/HPSR.2002.1024209 (2002).
[15] Papaphilippou, P., Meng, J. and Luk, W.: High-Performance FPGA Network Switch Architecture, *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, New York, NY, USA, Association for Computing Machinery, p. 76–85 (online), DOI: 10.1145/3373087.3375299 (2020).
[16] RISC-V International: `https://riscv.org/`.
[17] Boost C++ Libraries: `https://www.boost.org/`.
[18] Boost.YAP Library: `https://www.boost.org/doc/libs/release/doc/html/yap.html`.
[19] LibFirm: `https://pp.ipd.kit.edu/firm/`.
[20] Hack, S., Grund, D. and Goos, G.: Register Allocation for Programs in SSA-Form, *Compiler Construction* (Mycroft, A. and Zeller, A., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 247–262 (2006).
[21] Intel: インテル (R) Stratix(R) 10 MX FPGA 開発キット, `https://www.intel.co.jp/content/www/jp/ja/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html`.
[22] Intel: High Bandwidth Memory (HBM2) Interface Intel® FPGA IP User Guide, `https://www.intel.co.jp/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-20031.pdf`.