Vol.2019-HPC-171 No.5
2019/9/20

量子計算機シミュレーションのNVLinkの機能を用いた性能評価

土井 淳[1,a)]　高橋 ひとみ[1,b)]　レイモンド ルディ[1,c)]　今道 貴司[1,d)]　堀井 洋[1,e)]

概要：量子計算機は組み合わせ最適化問題等の計算量が爆発的に増大する問題について現代の計算機に代わって解くことができる計算機として期待されている．近年実際に動作する量子計算機が登場しつつあるが，ノイズの問題などによって安定的にソフトウェアを実行するのは難しく，また計算機資源も限られており，実機によるアルゴリズム開発やデバッグは効率が良くない．そこで現行の計算機を用いてシミュレーションで量子計算を行うことで，ソフトウェアの開発を行うのが現実的である．しかしながら現行の計算機上で量子計算を行うには量子ビット数のべき乗のメモリと計算量が必要となり，GPU のような加速装置を用いて高速にシミュレーションをすることが望まれている．本研究では，NVLink2 に実装される高速な転送機能やメモリコヒーレンシーを利用して，複数の GPU を用いた量子計算機シミュレーションを実装した．特に複数 GPU と CPU のメモリ空間を統一的に扱える unified memory や Address Translation Service(ATS) の機能を用いた場合についての評価を行う．

Evaluations of Quantum Computing Simulation by Using Function of NVLink

1. はじめに

量子計算機は，組み合わせ最適化問題のように計算量が爆発的に増大するような，従来の計算機では扱うことが困難であったような問題を解くことができる可能性を持った計算機として次世代の計算機として期待されている．近年，扱える問題や規模に制限があるものの実際に動作する汎用ゲート型の量子計算機が登場しつつあり，研究開発が活発になってきている．しかしながら，現状の量子計算機は利用可能な量子ビット (qubit) 数も小さく，また非常にノイズが多く安定した結果を得るために工夫が必要であり，実用的にはまだ従来の計算機に代わるようなレベルには達していない．また，量子計算機の実機は従来の計算機のように潤沢に用意されているわけではなく非常に限られた計算機資源となっている．

[1] IBM Research - Tokyo
19-21, Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo 103–8510, Japan
[a)] doichan@jp.ibm.com
[b)] hitomi@jp.ibm.com
[c)] rudyhar@jp.ibm.com
[d)] imamichi@jp.ibm.com
[e)] horii@jp.ibm.com

このような量子計算機の現状では，量子計算機のためのアルゴリズムやソフトウェアの開発は効率よく行うことが困難であり，実機に合わせて，従来の計算機上で量子計算機シミュレーターを用いる方法がとられている．量子計算機シミュレーターは，従来の計算機のメモリ上に量子状態を記録し，量子ゲート演算による量子状態の変化をシミュレートする．このときノイズのない状態で計算が可能なため，理想的なアルゴリズムを開発/テストすることが可能である．このとき量子状態は倍精度の複素数の配列として記録し，n-qubit の量子計算をシミュレートするには長さ 2^n の配列が必要で，すなわち，16×2^n バイトのメモリが必要である．同時に量子ゲート演算の計算量も $O(2^n)$ であり，必要なメモリサイズと計算量ともに，量子ビット数に対して指数関数的に増大する．そのため，量子計算機シミュレーションの実用的な量子ビット数は高々数十ビット程度であると考えられる．

量子計算機シミュレーションを高速に実行するための一つの方法に GPU を利用する方法がある．GPU は消費電力量あたりの計算量の観点から近年多くの計算機に採用されており，多くの計算アルゴリズムを高速化するのに広く

用いられている．量子計算機シミュレーションでも，指数関数的に増大する計算量を扱うために GPU による高速化が適していると考えられるが，GPU に搭載されているメモリ容量は一般的な計算機のメモリ容量よりも比較的小さく，大きな量子ビット数の問題を扱うにはやや不利である．例えば NVIDIA の Tesla V100 [1] の場合 16GB または 32GB のメモリが使用できるが，16GB の場合最大で容量の半分までを量子状態を保存するために利用できるとして 29-qubit の量子計算まで実行可能である．

より大きな量子ビット数の量子計算を GPU を用いて高速化しようと思えば，複数の GPU を利用するか，さらに GPU クラスターを用いて複数のノードにまたがった分散メモリ並列化を行うかする必要がある．我々は，IBM Power System AC922 [2] のクラスターを用いて，分散メモリ並列化を行い，32 ノードを用いて 39-qubit の量子計算シミュレーションを実現している．[3]

しかしながら，複数の GPU やノードを用いて並列化を行うことは簡単ではなく，コードが複雑になりがちである．我々は，オープンソースソフトウェアである Qiskit Aer [4] を元に GPU による高速化コードを開発している．オープンソースソフトウェアでは，互換性や保守性が重視されるため，特定の環境に特化して最適化されたコードよりも，汎用的に書かれた見やすいコードの方が望まれる．そこで，本研究では，複数 GPU を用いるために Unified Memory [5] の仕組みを用い，比較的単純なプログラミングで保守性を高めた．最適化された複数 GPU を扱うプログラムと，簡易的に書かれたプログラムでどの程度性能に差が出るのかを評価した．

2. 量子計算機シミュレーション

2.1 量子計算とシミュレーション概要

量子回路は量子ゲートの組み合わせでアルゴリズムを作り量子状態を操作していく．量子状態は確率振幅の重ね合わせとして表現され，最終的にこれらを観測することで，それぞれの量子ビットが 0 か 1 として確立する．量子状態は n-qubit のテンソル積の形で表し，従来の計算機上では，図 1 のように 2^n の長さの複素数の配列，量子状態ベクトルとして保存する．

量子ゲートは量子状態を変化させるような演算で，シミュレーションでは配列内の複素数の確率振幅の組に作用させる．代表的な量子ゲートに次の式に示す u3 ゲートがあり，このゲートによってある量子ビットを回転させる効果がある．

$$u3(\theta,\psi,\lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ -e^{i\psi}\sin(\theta/2) & e^{\psi+\lambda}\cos(\theta/2) \end{pmatrix}. \quad (1)$$

シミュレーション上では，図 2 に示すように，2x2 の複

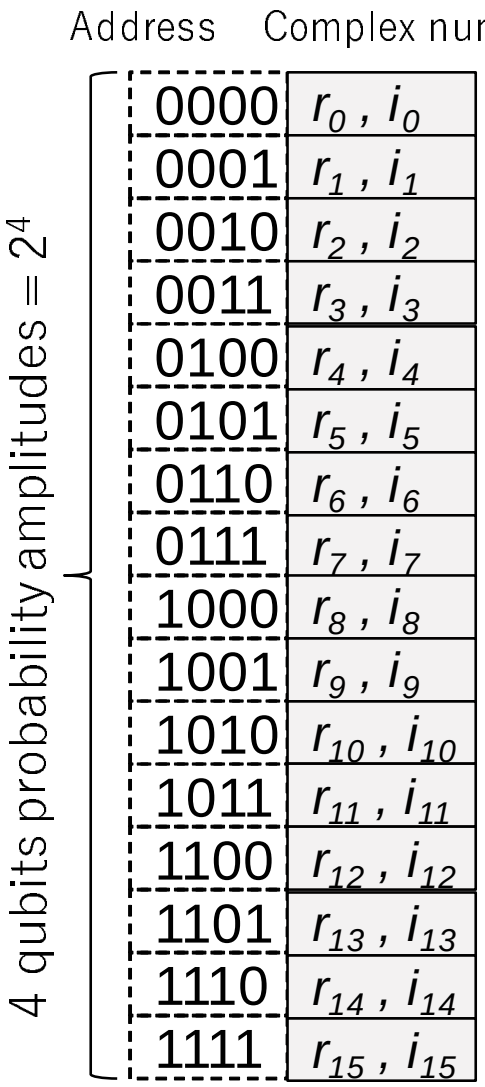

図 1 4 qubit の場合の量子状態を表す配列の例．長さ 16 の複素数配列として記憶する．

素数行列をベクトルに乗じる処理になる．ここで k-qubit について u3 ゲートを適用する場合，ベクトル成分となる 2 つの確率振幅の組は，配列のアドレスに 2^k の XOR をとることで求めることができる．

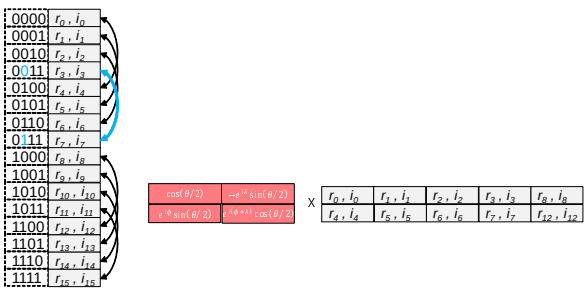

図 2 k=2 についての u3 ゲートの計算の例．2^k 離れた 2 つの確率振幅の組をベクトルとし 2x2 の行列を乗じる．

このように u3 ゲートは 2^n の長さの配列のすべての要素

を更新するため，1 つの量子ゲートをシミュレーションするためにはすべての配列要素についてアクセスする必要がある．u3 ゲートの byte/flop 値は約 2.29 であり，近年の多くの計算機ではメモリバンド幅律速な処理となる．

量子計算機の実機においては，異なる量子ビットを対象とする量子ゲートは並列して実行することが可能であるのに対して，シミュレーションでは，それぞれの量子ゲート計算がすべての要素を書き換えるため，同時に書き換えが起こる可能性があるため並列実行はできない．よって，シミュレーションによる並列化は，量子ゲートについての並列度ではなく 1 つの量子ゲート計算についての並列度を利用する．u3 ゲートでは，2^{n-1} の並列度があるので，GPU のような大量なスレッドを用いて並列化を行うのに向いている．

異なる量子ビットに対する量子ゲート演算は別々に同時に適用することはできないが，複数の量子ゲート同士を組み合わせたゲート演算を定義することは可能である．これは gate fusion と呼ばれるテクニックで，例えば 2 つの別々の量子ビットを対象とする 2x2 行列で計算できる量子ゲートは，4x4 行列で一度の計算にまとめることが可能である．つまり，m 個の量子ビットを対象とする量子ゲートの組み合わせは 2^mx2^m の行列にまとめることが可能である．このテクニックを使うことにとって，byte/flop 値を小さくすることが可能であり，メモリバンド幅の要求量を減らすことができる．また，複数 GPU や複数ノードを利用した並列化を行う場合に必要なデータ転送を減らす効果が得られる．

2.2 Qiskit Aer

Qiskit(Quantum Information Science Kit) [6] は，オープンソースな量子計算のための統合環境であり，量子計算を行うためのプログラミング環境，ライブラリ，実機およびシミュレーションを実行するためのツール群から構成され，誰でも簡単に量子計算を始めることができるようになっている．

Qiskit Aer [4] は従来の計算機の上で量子計算シミュレーションを行うためのフレームワークで，量子計算機実機で実行するのと同じプログラムを Qiskit Aer 上でも実行することが可能である．Qiskit Aer にはいくつかのシミュレーターの実装があるが，本研究では，ここまで説明してきた量子状態ベクトル型 (statevector) のシミュレーターについて GPU で高速化を行う．

3. 生産性と保守性のためのプログラミング環境

ここでは，Qiskit Aer による量子計算機シミュレーションを GPU で高速化するにあたり，生産性と保守性を高めるためのプログラミング環境についての概要を説明する．

3.1 NVLink

NVLink [7] は，NVIDIA 社によって提唱された GPU 間を高速に接続するための新しいインターコネクトであり，同社の GPU である，Tesla P100 に初めて搭載された．また，NVLink によって，GPU のみならず，PCI Express の代わりに GPU と CPU の間を接続し，より高速なデータ転送を行うことが可能となった．第二世代である NVLink2 は，一本あたり片方向 25GB/s のリンクを 6 本持ち最大で片方向 150GB/s の通信速度を持つ．図 3 は IBM Power System AC922 [2] の NVLink2 の接続例を示す．CPU ソケットあたり 3 つの GPU を接続する構成では，2 組ずつのリンクを用いて CPU と 3 つの GPU を相互結合することで，それぞれの間を双方向 100GB/s で接続している．これに対して現在の一般的な PC クラスター製品では，CPU と GPU の間は第三世代の PCI Express で接続するため，双方向で 32GB/s の転送速度となり，NVLink は 3.1 倍高速にデータ転送が行えることになる．ただし，これは PCI Express のリンクが十分にある場合の速度であり，PCI Switch 等を利用してリンクを共有する場合は，さらに速度差が大きくなる．NVLink のもう一つの利点は，ネットワークインターフェース等，他の PCI Express 機器と独立したインターコネクトで GPU にアクセスできることである．

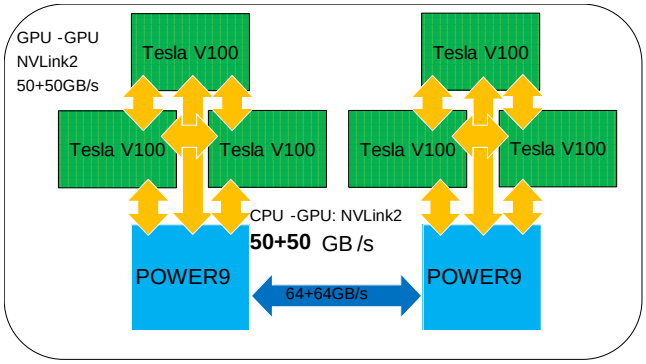

図 3 IBM Power System AC922 における NVLink2 の接続図．3 つの Tesla V100 GPU と 1 つの POWER9 CPU が相互に NVLinki2 で片方向 50GB/s で接続される．

3.2 Unified Memory

Unified Memory [5] は，CUDA [8] に実装されている仮想メモリの仕組みであり，ユーザープログラムから図 4 のように CPU と GPU で単一のメモリ空間を扱えるように，CPU-GPU 間のデータ転送を隠蔽した実装である．CPU と GPU それぞれにおいてページフォルトが発生した際に自動的にページ単位（64KB 単位）でデータ転送が実行される仕組みになっている．また，GPU に搭載されるメモリサイズよりも大きなメモリサイズを確保でき，GPU のメモリサイズに収まりきらないような問題を扱うことも可能である．また，複数の GPU 間でも相互に参照が可能な

メモリを提供し，簡単に複数 GPU による共有メモリ並列化を行うことが可能である．

プログラマーは，実データがどこにあるかを意識せずにプログラミングが行えるため，GPU へのコード移植が簡単になったり，プログラムコードの保守性が良くなったりといった利点が得られる．しかしながら，物理的にデータがどこにあるかを意識して書かれたプログラムに比べると，自動的に行われる CPU-GPU 間のデータ転送がボトルネックになる可能性がある．そのため，より転送速度の高速な NVLink と Unified Memory を組み合わせることで，比較的少ないオーバーヘッドで Unified Memory を利用できると考えられる．[9][10]

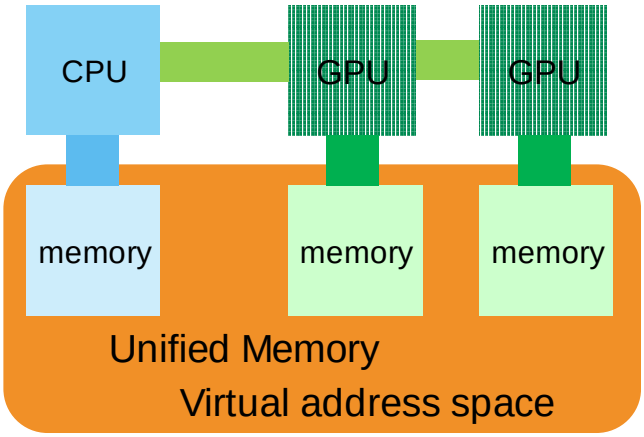

図 4　CUDA における Unified Memory の概念図．仮想メモリを用いて CPU と GPU の境なく同一のアドレス空間が利用できる．

3.3　Address Translation Service(ATS)

Address Translation Service(ATS) [11] は，IBM の POWER9 CPU と NVLink2 の独自の機能であり，CPU と GPU の間のキャッシュコヒーレンシーを利用して Unified Memory と同様に CPU と GPU 間で同一のメモリアドレス空間を利用可能とする．Unified Memory では cudaMallocManaged 関数等で Unified Memory として配列を確保することで利用するが，ATS では CPU 側で確保されたすべてのデータについて，CPU と GPU 双方からアクセスが可能となる．malloc 関数や new で確保されたものはもちろん，静的な配列や，スタックに確保された値も利用可能となる．

ATS は Unified Memory のようにまとめて確保された大きな配列を扱うというよりも，小さなデータを CPU と GPU で共有するような場面で特に有効である．例えば，GPU のカーネルプログラムに，GPU から参照可能な複数のポインタの入った配列を渡すような場合，従来は GPU 上に配列を確保してコピーしないといけなかったが，CPU 上の配列のまま渡すことが可能となりプログラムがシンプルになる．また，C++で書かれた非常に複雑なデータ構造を持つようなプログラムでは，まとめてデータを GPU 上にコピーしにくいものもあり，このようなものも明示的に GPU 上にデータをコピーしなくても実行できるようになる．

3.4　Thrust

Qiskit Aer は C++で実装されており，すべての処理はクラス内に定義される．通常の CUDA のプログラミングでは，GPU カーネルはグローバルな関数として書かれるため，Qiskit Aer の開発ポリシーには合わない．そこで，CUDA によって直接カーネルプログラムを書く代わりに，Thrust [12] を利用する．Thrust は Standard Template Library(STL) 互換の機能を持つクラスライブラリで，配列の管理や，GPU に最適化された配列に対する基本的な操作などの機能を持っている．比較的直感的にプログラミングが可能で，一般的な CUDA プログラムよりも可読性が高く，高い保守性が得られると思われる．筆者の感覚的には，CUDA FORTRAN に非常に近いプログラミングが可能で，とても簡単に GPU を利用できると感じた．

現時点で筆者が確認した制限として，Thrust で扱う配列自体には ATS が利用できない．これは，Thrust 内部で GPU に有効なアドレスかどうかのチェックが入っているためであり，Unified Memory の場合は利用可能である．

4.　量子計算機シミュレーションの GPU 化

4.1　Thrust による GPU 化

Thrust ではラムダ式を用いることで比較的簡単に GPU カーネルを実装することができる．単純な逐次アクセスな関数を作るだけであれば，配列を渡して式を書くだけで実装できるが，量子計算機シミュレーションでは与えられたインデックスから確率振幅の組を取り出して計算しなくてはならないため，逐次に与えられるインデックスから，アドレスに変換する部分を自分で書く必要がある．

逐次に計算を行うための `thrust::for_each` に渡す関数を自分で定義するためには，`unary_function` を継承し，operator に実装を書く．`operator` の引数に逐次インデックスを受け取るようにし，インデックスからアドレスを計算する．operator 内部からは継承したクラスのメンバ変数に設定した値のみ参照でき (CUDA におけるカーネル関数の引数にあたえられるイメージ)，ここに実際の量子状態ベクトルのポインタを持たせることで，逐次アクセスではなくても任意の位置のデータにアクセスができるようになる．実際にはこのポインタは Unified Memory として確保した配列である．また，u3 ゲートで用いる 2x2 の行列は，ここではそれぞれ 4 つの複素数の定数として渡す．これらはコンストラクタで設定することが可能である．

次のコードは，u3 ゲートの実装である．

```
struct u3_lambda : public unary_function<int,void>
{
 complex<double>* pVec;
 complex<double> m0,m1,m2,m3;
 int qubit;
 int add;
 int mask;
 u3_lambda(complex<double>* pV,
           complex<double>* pM,int q)
 {
  pVec = pV;
  qubit = q;
  m0 = pM[0];
  m1 = pM[1];
  m2 = pM[2];
  m3 = pM[3];
  add = 1ull << qubit;
  mask = add - 1;
 }
 __host__ __device__
 void operator()(const int &i) const
 {
  int i0,i1;
  complex<double> q0,q1;
  i1 = i & mask;
  i0 = (i - i1) << 1;
  i0 += i1;
  i1 = i0 + add;
  q0 = pVec[i0];
  q1 = pVec[i1];
  pVec[i0] = m0 * q0 + m2 * q1;
  pVec[i1] = m1 * q0 + m3 * q1;
 }
};
```

また，以下のコードは，u3 ゲートを GPU で実行する場合の実行方法を示す．

```
auto ci = thrust::counting_iterator<int>(0);
int n = 1 << (nqubits-1);
thrust::for_each(thrust::device,ci,ci+n,
                 u3_lambda(vec,mat,qubit));
```

他の量子ゲートについても同様に実装を行った．

4.2 複数の GPU による並列化

Unified Memory を用いて量子状態ベクトルを確保しているため，複数 GPU による並列化は非常に簡単である．GPU 間のデータ転送は自動的に行われるため，各 GPU が計算する範囲を指定してあげるだけで，並列化が完了する．次のコードのように OpenMP を利用して使用す GPU 数だけ並列に実行し，各 GPU で計算する範囲を与えるだけである．

```
 int iDev;
 int n = 1 << (nqubits-1);
#pragma omp parallel for
 for(iDev=0;iDev<nDev;iDev++){
  int is,ie;
  is = n * iDev / nDev;
  ie = n * (iDev+1) / nDev;
  auto ci = thrust::counting_iterator<uint_t>(0);
  cudaSetDevice(iDev);
  thrust::for_each(thrust::device,ci+is,ci+ie,
                   u3_lambda(vec,mat,qubit));
 }
}
```

4.3 Gate fusion の実装

u3 ゲートの実装では，行列の大きさは 2x2 であるので，行列の 4 成分を変数として渡すことができたが，gate fusion を行った場合，2^mx2^m 行列成分を変数としてカーネルに渡すのは現実的ではない．そこで，配列として行列を渡すことになるが，この比較的小さな配列を GPU 上に確保してコピーする処理を書くのは煩雑である．また，複数 GPU を使いたい場合，それぞれの GPU にコピーしなくてはならないため，さらにコードが複雑になってしまう．

そこで，ATS が利用可能であれば，CPU 上にある行列をそのまま渡し，ATS が利用できない場合は Unified Memory で確保した配列に行列をコピーすることで，複数 GPU にも対応できる実装を行った．

5. 性能評価

5.1 評価環境

性能評価には，IBM Power System AC922 で構成されるクラスタのうちの 1 ノードを使用した．今回の評価に使用するコードは，分散メモリ並列化には未対応である．表 1 に，使用した計算機環境についてまとめる．

5.2 性能評価に利用する量子回路

ここでは，Quantum Volume と呼ばれるランダムに生成された量子回路プログラムを性能評価に利用する．Quantum Volume は，すべての量子ビット間の量子もつれを含むような回路で，回路自体を最適化によって簡略化ができない（ずるができない）ような特徴がある．そのため，量子計算機自体の性能を公平に測定できるベンチマークである．今回はノイズが無い状態でのシミュレーションの実行で評価を行うが，ノイズのある実機において，量子ビット間のもつれを繰り返した時の安定性の評価にも役立つベンチマーク回路である．

表 1　性能評価に用いる計算機環境

Cluster node	IBM Power System AC922
CPU	POWER9
Number of sockets per node	2
Number of cores per socket	21
CPU memory size	512GB
GPU	NVIDIA Tesla V100
Number of GPUs per node	6
GPU memory size	16GB
CPU - GPU interconnect	NVLink2
OS	Red Hat Enterprise Linux Server 7.6
Compiler	GCC 8.3.0
CUDA Toolkit	CUDA 10.1

5.3　性能評価

まず, GPU を 1 つ使用した場合について, (i)CPU で実行, (ii)Thrust と Unified Memory を使用して実行, (iii)CUDA で最適化した実装で実行, の 3 つについて Quantum Volume 回路をシミュレートしたときの実行時間を比較する. なお, gate fusion を有効にし, 複数のゲートを同時に計算している. この時の実行結果を図 5 にまとめる.

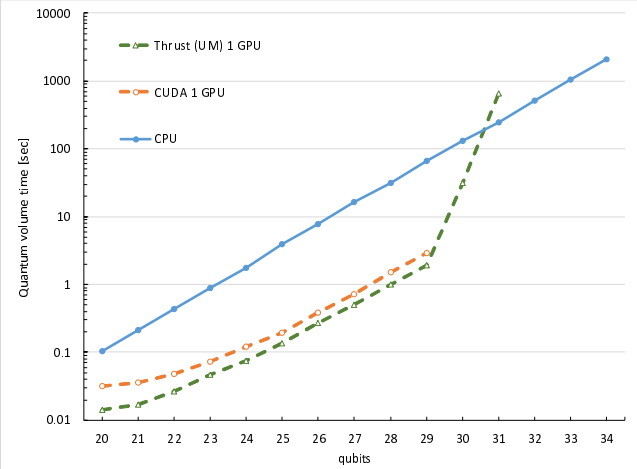

図 5　Quantum Volume 回路 (depth=10) を Qiskit Aer で実行した場合のシミュレーション実行時間の比較. IBM Power System AC922 において GPU を 1 枚利用した場合

図 5 では, Tesla V100 は 16GB のメモリを搭載しているため, GPU 上にメモリを確保する場合, 最大で 29 qubit までの量子回路の計算が可能である. よって, CUDA 版では最大で 29 qubit まで実行できるが, Thrust 版は Unified Memory を使用しているため, それ以上の qubit 数の量子回路の計算も可能となる. しかしながら, CPU と GPU の間でデータのスワッピングが発生するため, 実行時間は大きくなってしまう. CPU のみを利用した場合よりも低速な実行となる.

GPU 内に収まる場合 (29 qubit 以下の場合) 驚くべきことに, CUDA で最適化した実装よりも, Thrust と Unified Memory の実装の方が, 実行時間が短かった. この原因は, メモリの確保の仕方の違いが大きい. 図 6 に, 量子状態ベクトルの配列の確保を含む初期化の処理時間の比較をまとめる. CUDA で最適化したものは cudaMalloc 関数で直接 GPU 上にメモリを確保しているが, cudaMalloc 関数のオーバーヘッドがかなり大きく, 初期化にかかる時間が非常に大きいことが分かった. それに比べて, Unified Memory では cudaMallocManaged 関数で配列を確保するが, cudaMallocManaged 関数は確保時には GPU 内に実メモリを確保しないため, 非常に小さいオーバーヘッドとなっているようだ.

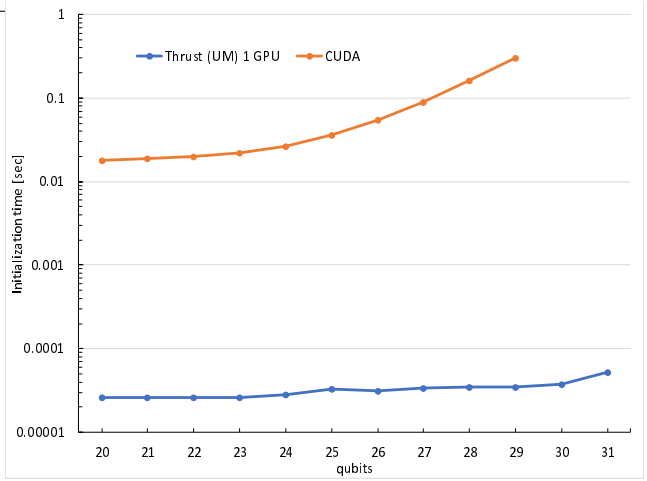

図 6　Quantum Volume 回路 (depth=10) を Qiskit Aer で実行した場合の初期化にかかる時間の比較. IBM Power System AC922 において GPU を 1 枚利用した場合

この違いは非常に大きく, 特に量子ビット数が小さい量子回路のシミュレーションをする場合, 計算時間が比較的小さくなるため, 初期化にかかる時間の割合は非常に大きくなってしまっている. CPU 版と比較した場合, CUDA 版の実行時間の加速率があまり良くないのはこれが原因であった. Unified Memory を使う利点として, GPU プログラミングが簡単になるというだけではなく, 少ないオーバーヘッドでメモリを確保できる, というのは非常に大きな発見である.

次に, 複数の GPU を利用する場合の比較を加えたものを図 7 に示す. 複数 GPU を利用する場合, GPU 間でデータを参照する必要がある場合にデータの転送のオーバーヘッドが加わる分, 単一 GPU で実行できる量子ビット数の場合は遅くなってしまう. 単一 GPU 実行の場合と同様に, CUDA 版は初期化にかかる時間が大きいため小さい量子ビット数の時には Thrust 版よりも遅い. しかしながら, 量子ビット数が大きくなると, 計算時間が初期時間に比べて相対的に大きくなるため, CUDA 版の方がデータ転送を最適化している分処理時間が短くなった. Thrsut 版は Unified Memory によって自動的にデータ転送が発生するためあまり効率が良くないが, それでも極端に遅いという

ほどではなく，簡単に実装できる割には十分健闘しているといえる．なお，33 qubit 以上になると，6GPU を使ってもすべての量子状態ベクトルを GPU メモリに保存できなくなるため，CUDA 版は CPU のメモリも利用するため，実行時間が増大している．[3]

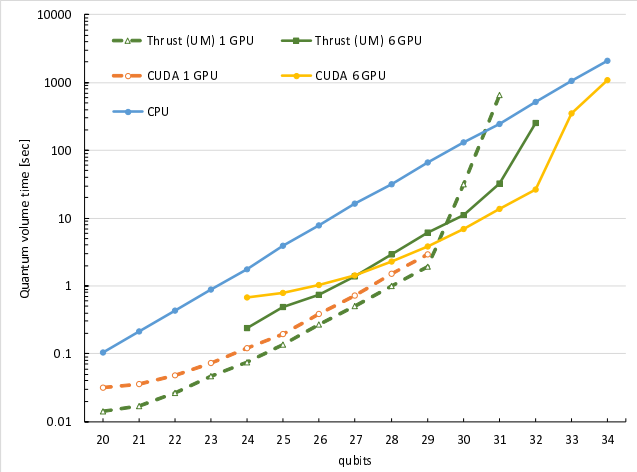

図 7 Quantum Volume 回路 (depth=10) を Qiskit Aer で実行した場合のシミュレーション実行時間の比較．IBM Power System AC922 を 1 ノード使用した場合

6. まとめ

量子計算機シミュレーションを GPU を用いて加速する方法として，CUDA を用いて最適なコードを生成する手法をとってきたが，オープンソースソフトウェアである Qiskit Aer に実装するにあたり，生産性と保守性を重視して，より単純なコードでどこまで性能が出せるかを検討してきた．本論文では，CUDA に付属する C++のテンプレートライブラリである Thrust を利用して少ない行数のコードで GPU によって量子計算機シミュレーションを高速化する方法を検討した．Thrust と Unified Memory を組み合わせることにより，非常に単純な実装で複数 GPU を用いた量子計算機シミュレーションを実装することができることが分かった．

Thrust と Unified Memory による実装を CUDA で最適化したものと性能を比較したところ，単一 GPU に収まる小さな量子ビット数の計算では，Thrust 版の方が初期化にかかるオーバーヘッドが小さいために高速であることが分かった．比較的短時間の計算処理を行うような場合，cudaMalloc による配列の確保のオーバーヘッドは馬鹿にならないくらい大きいことが分かり，Unified Memory を利用するメリットの一つとしてメモリ確保のオーバーヘッドが非常に小さいことを提案したい．複数 GPU を利用するような大きな量子ビット数の場合，さすがにデータ転送を明示的に行い最適化をしている CUDA 版の方が速度に分があるが，実装が難しく行数も非常に多いコードであり保守性という意味ではあまり良いコードではない．これに比べて Thrust 版では非常に可読性が高く短いコードにも関わらず，そこそこの性能が確認でき，十分に実用的であると考えられる．

今後は，Thrust 版についても複数 GPU 版の性能を改善する方法について検討していきたい．また，Thrust 版についても，分散メモリ並列化を実装し，複数ノードを利用してさらに大規模な量子ビットの量子計算機シミュレーションを実現したい．

参考文献

[1] Corporation, N.: NVIDIA TESLA V100 TENSOR CORE GPU, https://www.nvidia.com/en-us/data-center/tesla-v100/ (2019).
[2] Scott Vetter, Ritesh Nohria, G. S.: IBM Power System AC922 Technical Overview and Introduction, *An IBM Redpaper publication* (2018).
[3] Doi, J., Takahashi, H., Raymond, R., Imamichi, T. and Horii, H.: Quantum Computing Simulator on a Heterogenous HPC System, *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, New York, NY, USA, ACM, pp. 85–93 (online), DOI: 10.1145/3310273.3323053 (2019).
[4] Qiskit: Qiskit Aer: A High Performance Simulator Framework for Quantum Circuits (2019).
[5] Harris, M.: Unified Memory in CUDA 6, https://devblogs.nvidia.com/unified-memory-in-cuda-6/ (2016).
[6] Qiskit: Qiskit: An Open-source Framework for Quantum Computing (2017).
[7] Corporation, N.: NVLink and NVSwitch, https://www.nvidia.com/en-us/data-center/nvlink/ (2019).
[8] Corporation, N.: CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit (2019).
[9] 土井　淳: NVLink における Unified Memory と OpenACC によるプログラミングの性能評価，第 159 回 HPC 研究会，No. 5 (2017).
[10] 土井　淳: NVLink と Unified Memory を利用した OpenACC と OpenMP4.x による GPU プログラミング，第 161 回 HPC 研究会，No. 13 (2017).
[11] Grinberg, L.: Porting and Optimizing Applications for AC922 servers using OpenMP and Unified Memory, *OpenPOWER Summit Europe* (2019).
[12] : Thrust - Parallel Algorithms Library (2019).