修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院情報システム学研究科 情報ネットワークシステム学専攻 博士前期課程		
氏　　　名	島 圭吾	学籍番号	1152019
論 文 題 目	FLAT: MPI を埋め込み可能な GPU プログラミングフレームワーク		

要　　旨

GPU は，高性能計算分野において広く使用されている．2012 年 11 月に発表されたスーパコンピュータのランキングである TOP500 では，1 位に GPU 搭載 PC クラスタが，単位電力あたりの性能ランキングである Green500 では，上位 5 件のうち 3 件が GPU 搭載 PC クラスタである．

しかし，GPU 搭載 PC クラスタで動作するプログラムの開発には多大な労力を要する．具体的な問題として，GPU-GPU 間の通信が挙げられる．GPU 搭載 PC クラスタで動作するプログラムは，GPU 上の処理を記述する GPU コードと通信処理を行う CPU のコードで構成される．一般に，ノード間通信に利用する MPI は CPU コードに記述する必要があり，GPU コードには直接記述できない．従ってプログラマは，GPU コードと MPI 処理を実行する CPU コード，さらに，GPU-CPU 間のデータ授受の 3 種類のコードおよびデータ構造を管理しなければならない．

そこで本研究では，GPU 同士のデータ授受を見通しよく記述するために，MPI を埋め込み可能な GPU プログラミングフレームワーク"FLAT"を提案する．FLAT は GPU コードに MPI 関数を記述可能であるため，GPU 間の通信で管理する必要があるのは，GPU コード内に記述する通信コードと転送対象のデータ構造のみである．従って，プログラミングコストを軽減することができる．GPU コードに埋め込まれた MPI は，コンパイル時に CPU での MPI 処理の要求に変換され，CPU 上のランタイムルーチンによって適切に実行される．

NVIDIA C1060 を搭載した PC クラスタを用いて，FLAT を用いた実装と MPI 関数を明示的に CPU コードに記述した通常のプログラミング手法による実装とを比較した．その結果 GPU コードの計算粒度が粗粒度の場合，FLAT を用いた実装の性能低下率は 3%以下であり，実行性能で遜色ない結果が得られた．

電気通信大学大学院情報システム学研究科

2013 Jan.

修　士　論　文

FLAT: MPIを埋め込み可能な GPUプログラミングフレームワーク

主任指導教員　吉永 努

指導教員　入江 英嗣

指導教員　荻野 長生

平成 25 年 1 月 24 日

提 出 者

所 属	大学院情報システム学研究科
	情報ネットワークシステム学専攻
学籍番号	1152019
氏 名	島 圭吾

(表紙裏)

目次

第 1 章　序論 **1**

第 2 章　MPI を埋め込み可能な
GPU プログラミングフレームワーク FLAT **3**

2.1　FLAT を用いた Livremore ループ Loop18 の実装 3

第 3 章　FLAT の設計 **8**

3.1　実行モデル 8

第 4 章　FLAT の実装 **10**

4.1　コード変換 11

4.1.1　ラベル生成 11

4.1.2　退避・復帰コードの挿入 11

4.1.3　スタブの挿入 11

4.2　実行制御 11

第 5 章　評価 **14**

5.1　プログラマビリティ 15

5.2　オーバヘッド 16

5.3　実プログラムの実行性能 17

第 6 章　関連研究 **20**

第 7 章　結論 **22**

謝辞 **23**

参考文献 **26**

発表論文 **26**

発表文献 **27**

付 録 A　用語説明 **28**

A.1　HPC 28

A.2　HPCC 28

A.3　GPU 28

A.4　MPI 29

A.5 CUDA 29

図目次

2.0.1 GPU 搭載ノードで構成する PC クラスタ (GPU クラスタ) 3
2.0.2 2 ノードにまたがる GPU 間通信 4
2.0.3 MPI 埋め込み GPU プログラミングモデル 4
2.1.1 livermore ループ Loop18 の GPU 間でデータ転送が必要な例 5
2.1.2 Livermore ループ Loop18 のオリジナルのコード 6
2.1.3 MPI を明示的に CPU コードに記述する通常のプログラミング手法を用いた GPU コード (Livermore ループ Loop18) 6
2.1.4 MPI を明示的に CPU コードに記述する通常のプログラミング手法を用いた CPU コード (Livermore ループ Loop18) 7
2.1.5 FLAT を用いた GPU コード (Livermore ループ Loop18) 7
2.1.6 FLAT を用いた CPU コード (Livermore ループ Loop18) 7

3.1.1 FLAT の実行モデル 9

4.2.1 MPI リクエスト・データにパラメータを格納する GPU コード 12
4.2.2 MPI 処理のためカーネル関数の実行を中断するコード 12
4.2.3 レジスタメモリの退避・復帰コード 13
4.2.4 MPI 関数の発行を待つ CPU コード 13
4.2.5 GPU が発行した MPI 関数に対応する CPU コード 13
4.2.6 カーネル関数再開のための switch 文 13

5.0.1 吉永研究室 16 ノード GPU 搭載 PC クラスタ 15
5.2.1 通信性能およびカーネル関数の中断/再開にともなうオーバヘッド 16
5.2.2 2^{24} 個のスレッドが走るプログラムの保存すべきレジスタ変数の個数と実行時間の関係 17
5.3.1 Livermore ループ Loop18 の実行性能 18
5.3.2 ステンシル計算の実行性能 18
5.3.3 オプティカルフロー計算の実行性能 19

表目次

2.1 FLAT がサポートする MPI 関数 4

4.1 MPI リクエスト・データのメンバ 10
4.2 GPU ステータス・データのメンバ 10

5.1 GPU クラスタを構成する各ノードの諸元 14
5.2 プログラマが管理すべき通信コード数とメモリ数 15

第1章　序論

スーパーコンピュータを中心としたHPC(High-Performance Computing)分野では，コスト対性能比の観点から，PCノードを複数台用いたHPCC(High-Performance Computing Cluster)が多く用いられている．HPCCは，多数の計算ノードをネットワークで相互接続し，並列に動作させることでシステムの単位時間あたりの計算量を高める並列コンピュータの設計手法の一つである．

HPCCの1ノードあたりの性能においては，複数の演算コアをもったCPUや，数十〜数百個の演算コアから成るメニーコアプロセッサのアクセラレータとしての搭載によって高い性能を得ている．特にHPCC環境においては，GPU(Graphic Processing Unit)をはじめとする，アクセラレータを搭載し，CPUからアクセラレータに対して演算処理をオフロードするヘテロジニアス型が注目を集めている．GPUはピーク性能が汎用CPUと比べて高く，コスト対性能比や電力性能比などに優れているため，汎用のベクトルプロセッサとしてグラフィック用途以外で一般目的への利用(General Purpose computation using on GPUs，以下GPGPU)として期待されている．2012年11月に発表されたスーパコンピュータのランキングであるTOP500 [1]では，1位にGPU搭載PCクラスタ(以降GPUクラスタ)が，単位電力あたりの性能ランキングであるGreen500[2]では，上位5件のうち3件がGPUクラスタである．GPUクラスタでは，タスクを各計算ノードに分割して並列に実行させ，さらに各ノードでプログラムの適切な部分をGPUを活用して高速に実行させることで，高い演算性能を得ることができる．下川辺らは天気予報シミュレーション[3]を，Jacobsenらは CFD[4]を，Komatitschらは地震波の伝達解析[5]がGPUクラスタで高性能計算できることを示している．

一方，HPCCにおけるプログラミング環境について着目すると，ノード間制御には，Ethernetによる構成においてはMPI(Message Passing Interface)等の並列プログラミングAPIが提供されている．また，GPU等のマルチコアアクセラレータの制御には，CUDA[6]やAPP(旧ATI Stream)[7], OpenCL[8]といった特殊な言語を記述する必要がある．GPUクラスタのプログラミングでは，計算ノード間およびノード内並列性を考慮する必要があり，プログラムの開発には多大な労力を要する．具体的な問題として，GPU間のデータ転送が挙げられる．一般に，ノード間通信に利用するMPIはCPUコード[1]に記述する必要があり，GPUコード[2]には直接記述できない．従ってプログラマは，GPUコードとMPI処理を実行するCPUコード，さらに，GPU-CPU間のデータ授受の3種類のコードおよびデータ構造を管理しなければならない．

この課題に対し，GPU間の直接データ転送を可能にする手法として，専用のPCIデバイスを利用しPCIパケットをEthernetを介して他ホストに転送するExpEther[9]がある．これにより，ホストCPUを介さずにデータをやり取りできるため，高速にデータの授受が行える他，データ転送のプログラミングが簡潔になる可能性を持つ．しかし，専用のハードウェアデバイスを利用する必要があることから導入コストが大きくなり，安価で高性能な計算環境を提供するというGPUクラスタの利点を損なってしまう．

[1]各ノード内のCPUで実行されるプログラム
[2]各ノード内のGPUで実行されるプログラム

一方，ソフトウェアによりGPU間の通信を可能にする手法として，NVIDIAのGPU-Direct™[10]がある．GPU-Directは既存ハードウェア環境にそのまま適用可能であり，かつGPU-CPU間のデータ転送を記述をする必要がなくなるという利点を持つ．しかし，データ転送の発行はCPUコードで記述する必要があり，転送対象のデータ構造はCPUコードで管理する必要がある．従って，煩雑なメモリ管理は依然として存在する．

これらの問題を解決するために，本論文では，GPUコード中にGPU同士のデータ授受を記述でき，安価なGPUクラスタにそのまま適用可能なFLATフレームワークを提案する．FLATはGPUコードにMPI関数を記述可能であるため，3種類のコードおよびデータ構造を管理する必要はない．GPU間の通信で管理する必要があるのは，GPUコード内に記述する通信コードと転送対象のデータ構造のみである．また，ソフトウェアによる解決のため，既存のハードウェア環境にそのまま適用できる．

本論文の構成は次の通りである，2章でFLATの概要について述べる．3章でFLATを実現するための実行モデルを示す．4章でコード変換手法及び実装手法を示す．5章でFLATを利用して記述されたプログラムのプログラマビリティと実行性能を評価する．6章に関連研究を示し，7章でまとめる．付録には本論文を読む上で必要なHPC分野に関する用語説明である．

第2章　MPIを埋め込み可能な GPUプログラミングフレームワークFLAT

図 2.0.1 は，GPU クラスタを図示したものである．一般的な GPU クラスタでは，ノード内のGPU-CPU 間の通信および NIC を介したノード間の CPU-CPU の通信を行うことができるが，GPU から直接 NIC を介して他ノードの GPU にアクセスすることはできない．従って，GPU_0 から GPU_n へデータ送信する場合，図 2.0.2 に示す手順が必要となる．まず，GPU_0 から CPU_0 へデータをコピーする．次に，CPU_0 から CPU_n へ MPI 通信をする．さらに，CPU_n から GPU_n へデータをコピーすることでデータ送信が完了する．GPU-GPU 間の通信をするために，MPI 関数を明示的に CPU コードに記述する通常のプログラミング手法を用いた実装の場合，GPU_0 から CPU_0，CPU_0 から CPU_n，CPU_n から GPU_n のデータ通信に相当する通信コードをプログラムに記述する必要がある．

一方，FLAT では，GPU-GPU 間の通信は GPU_0 から GPU_n の通信に相当する図 2.0.3 の flat_mpi_send と flat_mpi_recv を GPU コードに記述すれば通信可能である．実際の通信である図 2.0.3 の斜線部分は，FLAT フレームワークによって自動的に通信する．現在，FLAT がサポートする MPI 関数は，表 2.1 の通りである．

FLAT はコード変換器である．ソース to ソース変換によって，GPU コードに埋め込まれた通信関数は，CPU コードに MPI 関数として適切に置換される．その後，既存コンパイラを用いて，実行コードを生成する．

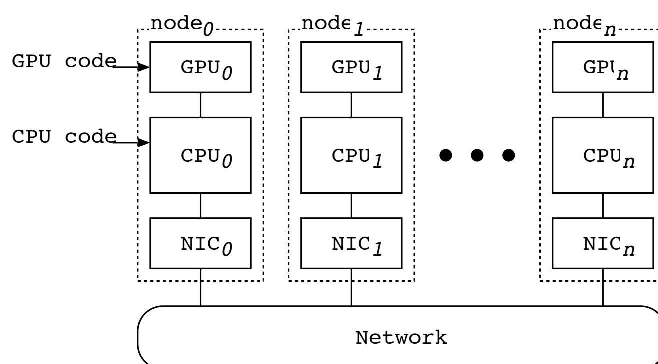


図 2.0.1: GPU 搭載ノードで構成する PC クラスタ (GPU クラスタ)

2.1　FLAT を用いた Livremore ループ Loop18 の実装

例として，ベンチマークプログラムの Livermore ループ [11] の Loop18 を考える．Livermore ループは，行列演算などの数値計算によく現れる 24 種類のループからなるベンチマークプログラムであり．Loop18 は二次元配列 $N \times N$ のデータに関する計算である．Loop18 の特徴として，計算途中

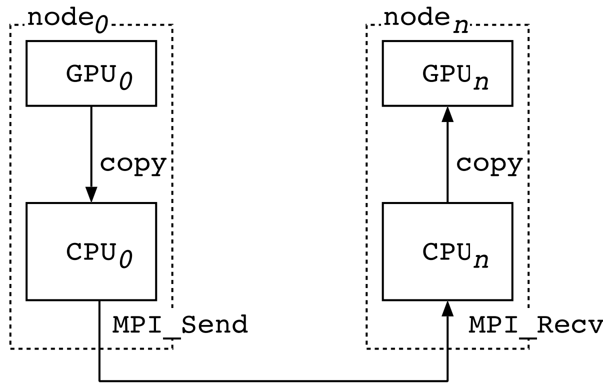


図 2.0.2: 2 ノードにまたがる GPU 間通信

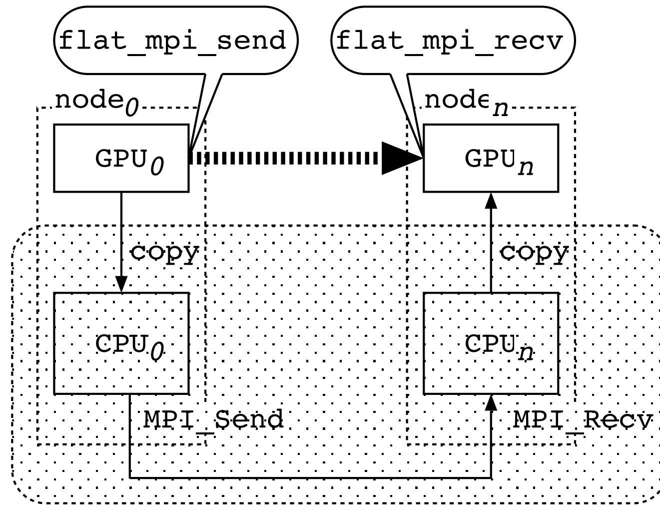


図 2.0.3: MPI 埋め込み GPU プログラミングモデル

表 2.1: FLAT がサポートする MPI 関数

name	corresponding MPI function
flat_mpi_send	MPI_Send
flat_mpi_recv	MPI_Recv
flat_mpi_isend	MPI_Isend
flat_mpi_irecv	MPI_Irecv
flat_mpi_wait	MPI_Wait
flat_mpi_sendrecv	MPI_Sendrecv
flat_mpi_barrier	MPI_Barrier
flat_mpi_scatter	MPI_Scatter
flat_mpi_gather	MPI_Gather
flat_mpi_bcast	MPI_Bcast

でデータ転送が必要になるということが挙げられる．GPU クラスタを用いて，計算対象となる二次元配列 $N \times N$ を図 2.1.1 に示すように縦にノード分割して実装する場合，ノード番号 id は，分割した境界部分のデータを MPI を用いて $id + 1$ から受信する必要があり，$id - 1$ に送信する必要がある．図 2.1.2 は C 言語で実装された Loop18 のオリジナルのコードである．図 2.1.2 に示した

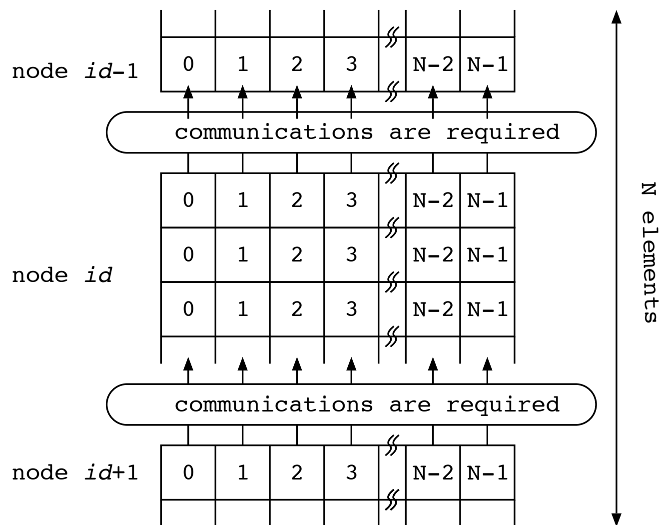


図 2.1.1: livermore ループ Loop18 の GPU 間でデータ転送が必要な例

プログラムを GPU クラスタで実行するための GPU コードと CPU コードを図 2.1.3 と図 2.1.4 に示す．GPU コードは CUDA で記述している．

Livermore ループの Loop18 を，図 2.1.3 と図 2.1.4 に示したような，MPI 関数を明示的に CPU コードに記述する通常のプログラミング手法を用いた実装には問題が 2 点ある．1 点目は，転送対象となる GPU 上のデータを，プログラマが CPU 上で実行する MPI 送信/受信に置換する必要がある点である．設計時に考慮していたデータのノード分割方法を変更する場合，GPU コードと CPU コードの両方を記述し直さなければならない．2 点目は，図 2.1.3 の GPU コードが，`kernel18_1` と `kernel18_2` に分断されていることにある．図 2.1.2 のオリジナルのコードは，2 カ所の二重ループ計算があり，1 回目のループと 2 回目のループの間に通信が必要になる．しかし GPU コードに MPI は記述できないため，プログラムの構造や計算のまとまりとは関係なく，MPI による通信の必要に応じて GPU コードが分断される．

FLAT はこれらの問題を解決する．FLAT を用いて図 2.1.2 に示したプログラムを記述することを考える．GPU コードを図 2.1.5 に，CPU コードを図 2.1.6 に示す．図 2.1.5 内の `flat_mpi_send()` および `flat_mpi_recv()` がそれぞれ MPI 関数の `MPI_Send()` と `MPI_Recv()` に相当する．GPU コード内に MPI 関数呼び出し及びその条件を直接記述することができるため見通しがよい．特に，転送の対象となるデータとして，CPU コード上の変数ではなく，実際に転送したい GPU コード上の変数を指定することができるため，分かり易いコードとなる．また，CPU コードは呼び出しのみで，GPU コードは分断する必要はない．オリジナルのコード同様，構造化された関数として記述可能である．

```
 1: void kernel18
    (float **za,float **zp,float **zq,float **zr,float **zm,
     float **zb,float **zu,float **zv,float **zz){
 2:  ...(snip)...
 3:  t = 0.0037; s = 0.0041; kn = NUM; jn = n;
 4:  for(k=1;k<kn;k++){
 5:    for ( j=1 ; j<jn ; j++ ) { /* calculation for za and zb */
 6:      za[k][j]=(zp[k+1][j-1]+zq[k+1][j-1]-zp[k][j-1]-zq[k][j-1])*
 7:               (zr[k][j]+zr[k][j-1])/(zm[k][j-1]+zm[k+1][j-1]);
 8:      zb[k][j]=(zp[k][j-1]+zq[k][j-1]-zp[k][j]-zq[k][j])*
 9:               (zr[k][j]+zr[k-1][j])/(zm[k][j]+zm[k][j-1]);
10:    }
11:  }
12:  for ( k=1 ; k<kn ; k++ ) {
13:    for ( j=1 ; j<jn ; j++ ) { /* calculation for zu and zv */
14:      zu[k][j]+=s*(za[k][j]*(zz[k][j]-zz[k][j+1])
                   -za[k][j-1]*(zz[k][j]-zz[k][j-1])
                   -zb[k][j]*(zz[k][j]-zz[k-1][j])
                   +zb[k+1][j]*(zz[k][j]-zz[k+1][j]));
15:      zv[k][j]+=s*(za[k][j]*(zr[k][j]-zr[k][j+1])
                   -za[k][j-1]*(zr[k][j]-zr[k][j-1])
                   -zb[k][j]*(zr[k][j]-zr[k-1][j])
                   +zb[k+1][j]*(zr[k][j]-zr[k+1][j]));
16:    }
17:  }
18:  ...(snip)...
19: }
```

図 2.1.2: Livermore ループ Loop18 のオリジナルのコード

```
 1: __global__ void kernel18_1
    (float *za,float *zp,float *zq,float *zr,float *zm,float *zb){
 2:   int i = blockIdx.y * blockDim.y + threadIdx.y+1;
 3:   int j = blockIdx.x * blockDim.x + threadIdx.x+1;
 4:   if( i < COL && j < N ){ /* calculation for za and zb */
 5:     za[i*ROW+j]= ...(snip)...;
 6:     zb[i*ROW+j]= ...(snip)...;
 7:  }
 8: }
 9:
10: __global__ void kernel18_2
    (float *za,float *zr,float *zb,float *zu,float *zv,float *zz){
11:   int i = blockIdx.y * blockDim.y + threadIdx.y+1;
12:   int j = blockIdx.x * blockDim.x + threadIdx.x+1;
13:   if( i < COL && j < N ){ /* calculation for zu and zv */
14:     zu[i*ROW+j]+= ...(snip)...;
15:     zv[i*ROW+j]+= ...(snip)...;
16:   }
17: }
```

図 2.1.3: MPI を明示的に CPU コードに記述する通常のプログラミング手法を用いた GPU コード (Livermore ループ Loop18)

```
 1: kernel18_1<<<dim3(N/BLOCKSIZE, N/NODES/BLOCKSIZE), dim3(BLOCKSIZE, BLOCKSIZE)>>>(zad,zpd,zqd,zrd,zmd,zbd);
 2: if(id > 0 ){
 3:   cudaMemcpy(buf, &zbd[ROW], sizeof(float) * ROW,cudaMemcpyDeviceToHost);
 4:   MPI_Send(buf, ROW, MPI_FLOAT, id-1 , 0, MPI_COMM_WORLD);
 5: }
 6: if(id < nCPU -1){
 7:   MPI_Recv(buf, ROW, MPI_FLOAT, id+1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
 8:   cudaMemcpy(&zbd[COL*ROW],buf, sizeof(float) * ROW,cudaMemcpyHostToDevice);
 9: }
10: kernel18_2<<<dim3(N/BLOCKSIZE, N/NODES/BLOCKSIZE), dim3(BLOCKSIZE, BLOCKSIZE)>>>(zad,zrd,zbd,zud,zvd,zzd);
```

図 2.1.4: MPI を明示的に CPU コードに記述する通常のプログラミング手法を用いた CPU コード (Livermore ループ Loop18)

```
 1: __global__ void kernel18
    (float *za,float *zp,float *zq,float *zr,float *zm,float *zb,
     float *zu,float *zv,float *zz, int id,int nCPU){
 2: int i = blockIdx.y * blockDim.y + threadIdx.y+1;
 3: int j = blockIdx.x * blockDim.x + threadIdx.x+1;
 4: float s = 0.0041; float t = 0.0037;
 5: if( i < COL && j < N ){ /* calculation for za and zb */
 6:   za[i*ROW+j] = ...(snip)...;
 7:   zb[i*ROW+j] = ...(snip)...;
 8:  }

 9:  if(id != 0)
       flat_mpi_send(&zb[ROW],sizeof(float)*ROW,id-1,
                   0,FLAT_MPI_COMM_WORLD);
10:  if(id!=nCPU-1)
       flat_mpi_recv(&zb[COL*ROW],sizeof(float)*ROW,id+1,
                   MPI_ANY_TAG,FLAT_MPI_COMM_WORLD);

11:  if( i < COL && j < N ){ /* calculation for zu and zv */
12:    zu[i*ROW+j]+= ...(snip)...;
13:    zv[i*ROW+j]+= ...(snip)...;
14:  }
15: ...(snip)...;
16:}
```

図 2.1.5: FLAT を用いた GPU コード (Livermore ループ Loop18)

```
 1: kernel18<<<dim3(N/BLOCKSIZE, N/NODES/BLOCKSIZE), dim3(BLOCKSIZE, BLOCKSIZE)>>>
                                          (zad,zpd,zqd,zrd,zmd,zbd,zud,zvd,zzd,id,nCPU);
```

図 2.1.6: FLAT を用いた CPU コード (Livermore ループ Loop18)

第3章　FLATの設計

GPU コードで MPI を記述可能にする既存技術は存在しない．既存ハードウェア環境にそのまま適用することを考えた場合，GPU コードに記述された MPI 関数の実際の処理を，CPU コード上での処理に置き換える手法が考えられる．この置き換え手法であれば，CUDA，APP(旧 ATI Stream), OpenCL など，本来 MPI による通信をサポートしない言語にも適用できる．FLAT では，MPI 関数の置き換え処理をプリプロセッサによるコード変換で実現している．コード変換後は，既存コンパイラによって実行ファイルを生成する．これにより，独自コンパイラを必要とせず，容易に FLAT フレームワークを実装することができる．

3.1　実行モデル

コード変換によって，GPU コードに記述された MPI 関数の実際の処理を，CPU コード上での処理に置き換えるため方法は 2 通り考えられる．一つ目は，GPU コードに記述された所望の MPI 関数を静的コード解析によって CPU コードに記述し直すことである．しかし，MPI 関数が GPU コード内のどこに記述されようとも実行できる必要がるため，制御構文の内側に MPI 関数が記述された場合には，制御構造ごと CPU コードに記述し直す必要があり，それは非常に困難である．二つ目の方法は，MPI 関数のデータのアドレスやサイズ，通信方式など，通信に必要な情報を CPU に通知するコードを，GPU コードと CPU コードの双方に挿入することである．この方法であれば，GPU コード内の制御構文をコード変換する必要はない．しかし，この手法を用いた場合，計算結果を保証するために，GPU コードに記述された MPI 関数の直前に，全 GPU スレッドの同期が必要である．FLAT では，二つ目の手法を用いて実行モデルを設計した．全 GPU スレッドの同期には，GPU コードを終了する事で実現している．実行モデルを図 3.1.1 に示す．各処理を以下に述べる．

1. CPU コードは必要な CPU 上，GPU 上のデータを初期化する．
2. GPU コードを起動する．
3. CPU コードは GPU コードの開始後 GPU コードからの MPI 処理のリクエスト発行を待機する．
4. GPU コードから MPI 処理を CPU にリクエストする．GPU コードは再実行のために必要なステータス情報を退避し，実行を中断する．
5. CPU コードは受理した MPI 処理のリクエストを実行する．
6. GPU コードを再起動する．GPU コードでは退避させたステータス情報を復帰し，MPI 関数後のコードから実行を再開する．
7. CPU コードは再び MPI 処理のリクエスト発行を待機する．
8. GPU コードは処理が完了すると，その旨を CPU コードに通知し，実行を終了する．

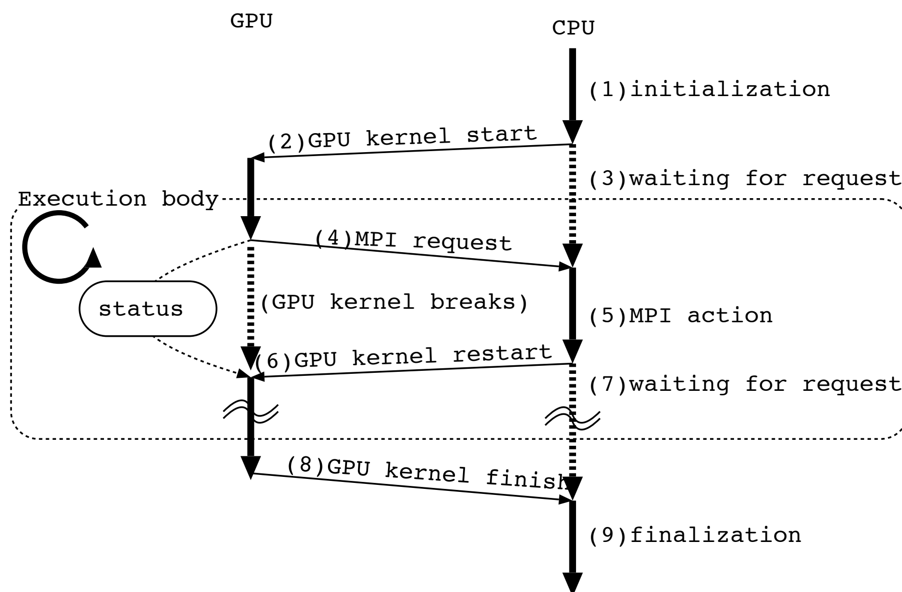


図 3.1.1: FLAT の実行モデル

9. CPU コードは後処理として確保したメモリ領域を解放する．

GPU でタスクを処理している間は，図 3.1.1 の (4) から (7) の処理を繰り返す．ここで，MPI 処理を行うために CPU で処理している間の中断から再実行までのブロックされた期間は，プログラマによって記述された GPU コードに対しては，隠蔽される．従って，プログラマは図 3.1.1 中に点線で囲った GPU 処理および MPI 処理を実行の本体として仮想的に取り扱うことができる．

第4章　FLATの実装

FLAT は，3 章で述べた実行モデルに従いコード変換を行う．実装する上で重要仕組みが 2 点ある．1 点目は，GPU から CPU へのリクエスト発行である．FLAT では，CPU で所望の MPI 関数を呼び出すために，GPU から CPU へリクエストを送る．リクエストを送るために，表 4.1 に示す MPI リクエスト・データ構造体のメモリを GPU と CPU で確保する．MPI リクエスト・データは MPI 処理をするためのパラメータを格納する．GPU で格納された MPI 処理のパラメータを，CPU へデータコピーすることでリクエストを発行する．表 4.1 の `request` には，FLAT がサポートする関数名，もしくは `GPU_DONE` の値を格納する．`GPU_DONE` は，カーネル関数が終了したことを，CPU に伝えるための値である．

2 点目は，全 GPU スレッドの同期をするための GPU コードの中断・再開である．FLAT は GPU コードの中断・再開を，GPU 関数（以後カーネル関数と呼ぶ）の終了と再起動で実現する．しかし，カーネル関数を終了することで，CUDA で使用されるレジスタメモリやシェアードメモリの値が失われる．従って，計算途中のデータと実行状態を保持する仕組みが必要である．

カーネル関数の中断の際，退避すべき情報は，表 4.2 に示す GPU ステータス・データ構造体によって保持する．`status` と `next_status` はそれぞれ，カーネル関数の実行コード位置と，再起動時のコード開始位置を保持している．また，カーネル関数の中断時に保存が必要なレジスタ変数はコード変換時に抽出され，GPU ステータス・データに加えられる．

表 4.1: MPI リクエスト・データのメンバ

name	type	description
`send_addr`	`void *`	top address of send data
`send_size`	`int`	the length of send data
`dest`	`int`	destination id to send
`recv_addr`	`void *`	top address of receive data
`recv_size`	`int`	the length of receive data
`src`	`int`	source id to receive
`comm`	`MPI_Comm`	MPI communicator
`request`	`request_t`	kind of requesting MPI function

表 4.2: GPU ステータス・データのメンバ

name	type	description
`status`	`int`	current position of GPU kernel to execute
`next_status`	`int`	next position of GPU kernel to restart

4.1　コード変換

FLAT は，ソースコードからソースコードへのコード変換器を用いて実現する．この変換は，CPU 側および GPU 側双方で行い，主に 3 点のコード変換を必要とする．1 点目は，MPI リクエスト時にカーネル関数の中断・再開の位置を制御するラベルの挿入である．2 点目は，保存が必要な揮発性メモリを抽出し GPU ステータス・データに加えること，また変数の退避・復帰を行うためのコード生成である．3 点目は，ユーザ定義のカーネル関数にスタブを挿入することである．

4.1.1　ラベル生成

FLAT はコード変換時に，カーネル関数の中断・再開に使用する固有のラベルを生成し，カーネル関数に挿入する．このラベルはカーネル関数再開の際，処理を再開する目印となる．従ってラベルは，プログラマによって埋め込まれた MPI 関数ごとに必要である．固有のラベルを割り振るために，カーネル関数名と番号を組み合わせた名前のラベルを生成する．ラベルの番号は，ラベル挿入箇所ごとに値を増加させる．

4.1.2　退避・復帰コードの挿入

FLAT は，変数の退避・復帰のためにカーネル関数内の MPI 関数の前後に `STORE_KERNEL_INFO()` と `RESTORE_KERNEL_INFO()` のマクロを挿入する．退避・復帰する変数がレジスタの場合，値を保持するスレッドが，デバイスメモリにコピーを行うことで退避する．復帰時は，デバイスメモリからレジスタ変数へコピーする．シェアードメモリの場合は，コピー直前にブロック同期を行い，ブロックごとにコピーする．シェアードメモリはコード解析時，配列の要素数が定数になることが決まってる．従って，ブロック内のスレッド数より，シェアードメモリの要素数が多い場合にでも，ループ文を用いることで，安全に退避・復帰することが可能である．

4.1.3　スタブの挿入

FLAT を実行するために，MPI リクエスト・データと GPU ステータス・データを GPU-CPU 間で共有する必要がある．FLAT は，これを実現するために，ユーザ定義のカーネル関数のスタブを生成する．このスタブは，プログラマが定義したカーネル関数をラップする．生成されたスタブの引数は，プログラマが記述した引数に加え，MPI リクエスト・データと GPU ステータス・データのポインタが加えられる．呼び出されたスタブ関数は，受け取った二つのポインタを保存し，プログラマが定義したカーネル関数を呼び出す．これにより，コード変換後に，GPU-CPU 間で MPI リクエスト・データと GPU ステータス・データの自動的な共有が可能となる．

4.2　実行制御

図 2.1.5 の 7 行目の `flat_mpi_send()` に着目し，カーネル関数の中断から，実際に MPI 関数が CPU によって呼び出され，GPU 処理を再開するまでを変換後のコードを使って説明する．

埋め込まれた MPI 関数はコード変換により，図 4.2.2 に変換される．図 4.2.2 の 2-3 行目の `STORE_KERNEL_INFO()` と `set_next_status()` は，それぞれデータの退避と再開時のコード開

```
 1: __device__ void
 2: flat_mpi_send(void *addr, int size, int id, MPI_Comm comm)
 3: {
 4:   if(threadIdx.x==0 && threadIdx.y==0 && threadIdx.z==0 &&
 5:      blockIdx.x==0  && blockIdx.y==0  && blockIdx.z== 0){
 6:     _info->send_addr = addr;
 7:     _info->send_size = size;
 8:     _info->dest = id;
 9:     _info->comm = comm;
10:     _info->request = MPI_SEND;
11:   }
12: }
```

図 4.2.1: MPI リクエスト・データにパラメータを格納する GPU コード

```
1: {
2:  STORE_KERNEL_INFO();
3:  set_next_status(1);
4:  flat_mpi_send(src_addr, sizeof(float), dest_id, tag);
5:  goto END_OF_KERNEL;
6:  LABEL_KERNEL_1: RESTORE_KERNEL_INFO();
7: }
```

図 4.2.2: MPI 処理のためカーネル関数の実行を中断するコード

始位置を設定するためのコードである，`set_next_status()` の引数 1 は，`LABEL_KERNEL_1` の 1 に対応する．4 行目の `flat_mpi_send()` によって，図 4.2.1 で示す通り MPI リクエスト・データにパラメータが格納される．パラメータはスレッド番号 ((0, 0, 0), (0, 0, 0)) によって格納する．従って現在の実装では，複数 GPU スレッドが別々の MPI 関数を呼ぶことができないため，MPI 関数は呼び出す数だけ記述する必要がある．

GPU コードを中断するために，図 4.2.2 の `goto END_OF_KERNEL` によって，強制的に GPU コードを終了する．MPI 関数実行後，再び呼び出されたカーネル関数は，`LABEL_KERNEL_1` まで処理をスキップし，計算を再開する．また，6 行目の `RESTORE_KERNEL_INFO()` によって，退避データを復帰する．`STORE_KERNEL_INFO()` と `RESTORE_KERNEL_INFO()` の実装を，図 2.1.5 の 2 行目の変数 i について図 4.2.3 に示す．

一方 CPU では，図 4.2.4 の 3 行目，`cudaMemcpy()` によってカーネル関数の中断もしくは終了を待っている．リクエストが発行された場合，MPI リクエスト・データを受け取り，図 4.2.4 の switch 文で，所望の MPI 関数が選択される．ここでは，図 4.2.5 に示す `host_mpi_send()` が呼び出される．この関数内で転送データを GPU から CPU へコピーし，その後 CPU によって `MPI_Send()` を実行する．CPU による MPI 関数の実行後，10 行目で，次のコード開始位置を特定し，カーネル関数を再起動する．

図 4.2.6 は，再起動されたカーネル関数が GPU ステータス・データをもとに中断位置から処理を再開するための switch 文である．この switch 文を介し，処理済のコードをスキップすることで，GPU 処理を再開する．

```
 1: #define STORE_KERNEL_INFO() \
 2:  { \
 3:    int _thread_id =   threadIdx.x \
 4:                     + threadIdx.y * blockDim.x \
 5:                     + threadIdx.z * blockDim.x * blockDim.y \
 6:                     + blockIdx.x * blockDim.x * blockDim.y * blockDim.z \
 7:                     + blockIdx.y * gridDim.x * blockDim.x * blockDim.y * blockDim.z; \
 8:    _status->i[_thread_id] = i; \
 9: #define RESTORE_KERNEL_INFO() \
10:  { \
11:    int _thread_id =  threadIdx.x \
12:                    + threadIdx.y * blockDim.x \
13:                    + threadIdx.z * blockDim.x * blockDim.y \
14:                    + blockIdx.x * blockDim.x * blockDim.y * blockDim.z \
15:                    + blockIdx.y * gridDim.x * blockDim.x * blockDim.y * blockDim.z; \
16:     i = _status->i[_thread_id]; \
17:  }
```

図 4.2.3: レジスタメモリの退避・復帰コード

```
 1:  do{
 2:   GPU_FUNCTION_stub<<<N,M>>>(....arguments...., info->info_dev, info->status_dev);
 3:   cudaMemcpy((void*)info->info_host, (void*)info->info_dev, sizeof(gpu_info), cudaMemcpyDeviceToHost);
 4:   switch(info->info_host->request){
 5:   case MPI_SEND:
        host_mpi_send((gpu_info*)info->info_host); break;
 6:   case MPI_RECV:
        host_mpi_recv((gpu_info*)info->info_host); break;
 7:   case MPI_BARRIER:
        host_mpi_barrier((gpu_info*)info->info_host); break;
 8:   case MPI_SENDRECV:
        host_mpi_sendrecv((gpu_info*)info->info_host); break;
 9:   }
10:   cudaMemcpy((void*)&(info->status_dev->status), (void*)&(info->status_dev->next_status),
                                                      sizeof(int), cudaMemcpyDeviceToDevice);
11:  }while(info->info_host->request != GPU_DONE);
```

図 4.2.4: MPI 関数の発行を待つ CPU コード

```
1: void host_mpi_send(volatile gpu_info *info)
2: {
3:   void *send_addr;
4:   send_addr=(void*)malloc(info->send_size);
5:   cudaMemcpy(send_addr, (void*)info->send_addr, info->send_size, cudaMemcpyDeviceToHost);
6:   MPI_Send(send_addr, info->send_size, MPI_CHAR, info->dest, 0, info->comm);
7:   free(send_addr);
8: }
```

図 4.2.5: GPU が発行した MPI 関数に対応する CPU コード

```
1:  switch(_status->status){
2:    case 0: goto LABEL_KERNEL_0;
3:    case 1: goto LABEL_KERNEL_1;
4:    case 2: goto LABEL_KERNEL_2;
5:    default:
6:      flat_error(); goto END_OF_KERNEL;
7:  }
```

図 4.2.6: カーネル関数再開のための switch 文

第5章　評価

本章では，第2章で用いたベンチマークの Livermore ループの Loop18，1次元ステンシル計算，実アプリケーションであるオプティカルフロー計算の三つのプログラムを，プログラマビリティと実行性能に関して評価する．

ステンシル計算は，科学計算領域における重要なプログラムの一つである．オプティカルフロー計算は，大村らが視覚神経系の数理モデルをもとに開発したアプリケーションである [12]．オプティカルフローとは，画像から観測される物体の動きをベクトルで表したものである．大村らは，工学的なオプティカルフローの推定法である，Lucas-Kanade 法 [13] を用いたモデルでシミュレーションを行った．アプリケーションは以下の流れに沿って実行される．

1. 入力画像のノード分割
2. GPU による1度目の計算
3. 袖領域のデータ交換
4. GPU による2度目の計算
5. ルートノードへの画像の集約

これら一連の流れをパイプライン処理によって高速化し，リアルタイムでオプティカルフローを計算可能にしている．FLAT を用いた実装には，3. 袖領域のデータ交換を GPU コードで記述している．また，オリジナルのプログラムと同様に，非同期通信を用いたパイプライン処理を実装している．

性能比較では，FLAT を用いた実装と，MPI 関数を明示的に CPU コードに記述した通常のプログラミング手法による実装とを比較する．オプティカルフロー計算については，大村らが実装したオリジナルのプログラムを通常のプログラミング手法とする．

実行時間の評価には，表 5.1 と図 5.0.1 に示すノード 16 台を Gigabit Ethernet で接続した GPU クラスタを用いる．

表 5.1: GPU クラスタを構成する各ノードの諸元

CPU	Intel(R) Xeon(R) CPU W3520 2.7GHz
メモリ	12GB
OS	CentOS Release 5.3 (Linux x86_64 2.6.18-128)
ネットワーク IF	Intel(R) PRO/1000 NIC
GPU	NVIDIA Tesla C1060

図 5.0.1: 吉永研究室 16 ノード GPU 搭載 PC クラスタ

5.1 プログラマビリティ

FLAT を用いた場合のプログラマビリティについて評価する．ここでは特に，コードの複雑さの観点からプログラマが管理すべきメモリ数[1]，およびプログラマが記述する通信コード数[2]を指標として評価する．表 5.2 は，Loop18，ステンシル計算，オプティカルフロー計算において，FLAT を利用した場合 (with FLAT) と，通常のプログラミング手法 (w/o FLAT) により実装した場合のプログラマが管理すべきメモリ数と記述が必要な通信コード数を表している．

表 5.2 より，両アプリケーションともに FLAT により管理すべきメモリ数と記述が必要な通信コード数が削減されていることが分かる．特に，オプティカルフロー計算の場合は w/o FLAT の場合は通信コードが 48 個，メモリは 64 個であったのに対し，FLAT を利用することで通信コードが 16 個，メモリは 32 個と大きく削減できている．これは，オプティカルフロー計算では 1 フレームをノード分割して 2 度の GPU 処理を行う必要がり，また，1 度目の GPU 処理で生成した中間データの袖領域の交換が必要であるため，PC ノードを 2 次元配置した場合，八方に双方向の袖領域交換が必要となるためである．このように，データ転送が多い場合は，プログラマが管理すべきメモリ数と，記述が必要な通信コード数は FLAT を利用することで大きく削減できるため，プログラマビリティ向上に貢献できると考えられる．

表 5.2: プログラマが管理すべき通信コード数とメモリ数

	Loop18		ステンシル計算		オプティカルフロー計算	
	with FLAT	w/o FLAT	with FLAT	w/o FLAT	with FLAT	w/o FLAT
通信コード数	2	4	4	8	16	48
メモリ数	1	3	1	7	32	64

[1]データを保存する配列や構造体数
[2]GPU-CPU 間，CPU-CPU 間における通信ライブラリの呼び出し回数

5.2 オーバヘッド

FLAT は主に 2 点のオーバヘッドが存在する．1 点目は，関数の中断・再開に伴うオーバヘッドである．FLAT は MPI 処理の度にカーネル関数の中断・再開を行う．このため MPI 関数が連続する場合，まとめて MPI 関数をリクエストするハンドコーディングに比べ，FLAT 実装では，カーネル関数の呼び出し回数が増える．そこで，カーネル関数の呼び出しにかかるオーバヘッドを測定した．測定用のプログラムでは，16 台のノードで，それぞれランク `(rank+1) & 0xF` のノードに指定したサイズのデータを送信し，ランク `(rank-1) & 0xF` のノードからデータを受信する．プログラムを CPU コード中で GPU-CPU 間のデータコピーと MPI 転送を記述する場合 (w/o FLAT) と，FLAT を利用して GPU コード中に MPI 転送埋め込む場合 (with FLAT) の実行時間を測定する．また，MPI でのデータ通信だけを行なうプログラム (MPI Data transfer(CPU only)) の実行時間も測定した．結果を図 5.2.1. に示す．横軸は通信の対象とするデータのサイズ (バイト数) である．図 5.2.1 の結果は，with FLAT，w/o FLAT ともにそれぞれデータサイズに依存しない同程度のオーバヘッドが上乗せされていることを示す．ここで GPU コードが起動される回数に着目する．GPU コードが起動される回数は MPI 通信だけを行うプログラムでは 0 回，CPU コード中に MPI 処理を記述する場合 (w/o FLAT) では 1 回，そして FLAT を用いて記述する場合 (with FLAT) では 3 回である．with FLAT の MPI でのデータ通信だけを行なうプログラムに対する増分と，w/o FLAT の MPI でのデータ通信だけを行なうプログラムに対する増分は，ほぼ 1:3 である．すなわち，GPU コードの起動にかかるオーバヘッドが，直接実行時間の増加に反映され，そのオーバヘッドは約 20μ 秒である．

2 点目のオーバヘッドは，揮発性メモリの退避・復帰にかかるオーバヘッドである．スレッドごとに保持しているレジスタの値を全て退避する必要があるため，スレッド数の増加に伴い，オーバヘッドも増加する．測定プログラムは，2^{24} 個のスレッドを生成するプログラムで中断・再開を一回行うものである．結果を図 5.2.2 に示す．保存するレジスタ数が実行時間の増加に大きく影響することが分かる．スレッド数 2^{24}，レジスタ変数が 6 個の場合，一回の GPU コードの中断/再実行にかかるオーバヘッドは約 0.26 秒となった．

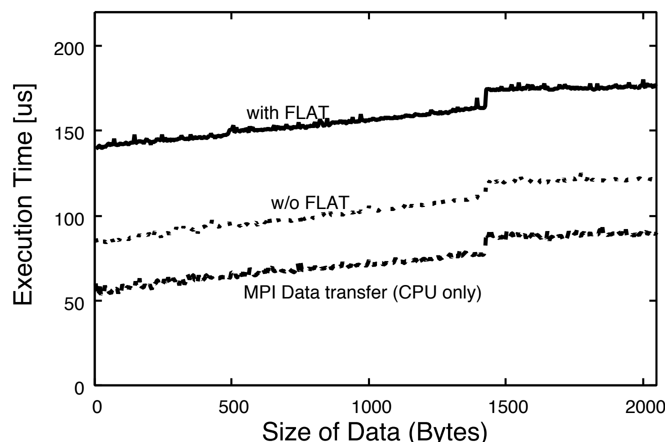


図 5.2.1: 通信性能およびカーネル関数の中断/再開にともなうオーバヘッド

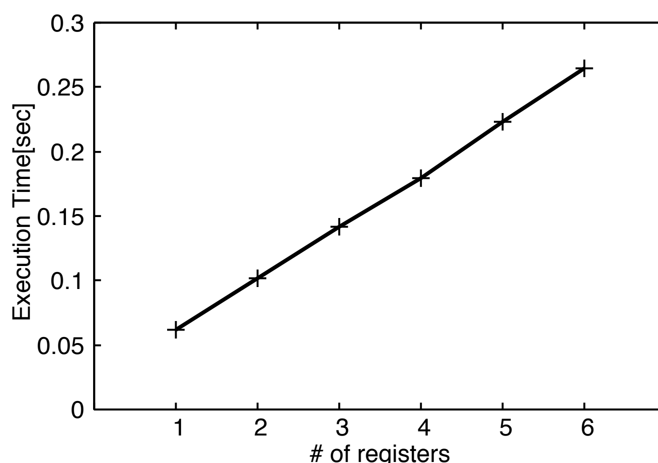


図 5.2.2: 2^{24} 個のスレッドが走るプログラムの保存すべきレジスタ変数の個数と実行時間の関係

5.3 実プログラムの実行性能

まず，livermore ループの Loop18 のについて考察する．プログラムは，1024×1024 の float 型の 2 次元配列で，データ交換を行う袖領域は 1024 ワードの設定である．実行結果を図 5.3.1 に示す．縦軸は実行時間であり，横軸はノード数である．with FLAT，w/o FLAT ともに台数効果が得られた．16 ノード実行で，w/o FLAT に対する with FLAT の性能低下率は 4%である．

次に，ステンシル計算の考察について述べる．図 5.3.2 は，総ステップ数を 2^{24} に設定し，256M ワード (1G) の float 型のデータを with FLAT と w/o FLAT で実行した結果である．データ交換を行う袖領域は 256 ワードである．with FLAT と w/o FLAT ともに，高いスケーラビリティを示し，16 ノード実行で，w/o FLAT に対する with FLAT の性能低下率は 3%である．わずかな差ではあるが，Loop18 の結果と比べ，FLAT の性能低下率が低い理由として，ステンシル計算は，Loop18 と比べて GPU による計算時間が長く，またデータ転送サイズが小さいため計算が粗粒度となることが挙げられる．Loop18 のループ毎の GPU 計算時間は 13 ミリ秒であったのに対し，ステンシル計算は 19 ミリ秒である．さらに，交換データサイズは，Loop18 が 1024 ワードであったのに対し，ステンシル計算は，128 ワードである．FLAT は MPI 通信に必要な CPU メモリを，通信の度に確保・解放する．そのため，通信データサイズによってもオーバヘッドが変わる．

最後に，オプティカルフロー計算プログラムについて考察する．図 5.3.3 に，処理する画像サイズを (a)320x240，(b)512x384，(c)640x480 で実行したときの性能を示す．横軸はノード数であり，縦軸は 1 秒間に処理したフレーム数 (FPS) である．処理データは float 型であり，交換袖領域は 16 ノード実行の場合 320x240，512x384，640x480 は，それぞれ 2156，3332，4116 ワードである．(a)320x240 については，1 フレームあたりの実行時間が他サイズに対して短く，100 フレーム平均の分散値が他サイズと比べて大きい，そのため，総数 500 フレームを 100 フレームずつ区切って FPS を算出し，トップスピードの値をグラフ化している．with FLAT は，画像サイズ 320x240 の 16 ノード実行において 60FPS を大きく越えており，512x384 で 30FPS を越えている．従って，w/o FLAT と同様，リアルタイムでオプティカルフロー計算を処理可能である．さらに，512x384 と 640x480 の画像サイズの 16 ノード実行では，w/o FLAT に対する，with FLAT の性能低下率は 3%以下である．with FLAT と w/o FLAT の性能差が画像サイズで異なる理由として，画像サイズが小さくなるにつれ GPU による計算が細粒度になることが挙げられる．1 フレームあたりの平均処

理時間は 640x480~320x240 で 50~8 ミリ秒に変化するのに対し，FLAT のオーバヘッドは画像サイズに関係なく 1 ミリ秒程度である．

以上をまとめると，GPU の計算量に対して FLAT のオーバヘッドは一定である. 従って，単純なベンチマークプログラムのような，一つのカーネル関数に必要な通信数や通信データサイズが少なく，カーネル関数の GPU 計算時間が 20 ミリ秒程度のものであれば FLAT の性能低下率は 3%以下になるため十分に適用可能である．また，オプティカルフロー計算のような実アプリケーションでは，データ転送の規模が大きくなり，また通信数が増えるため，カーネル関数の中断・再開の回数も多くなる．しかしこのような場合であっても，オプティカルフロー計算では，カーネル関数の GPU 計算時間が 50 ミリ秒程度であれば，FLAT の性能低下率は 3%以下であった．すなわち，FLAT はこれら条件を満たした粗粒度並列処理において，通常のプログラミング手法と比べて遜色ない性能が得られることが分かった．

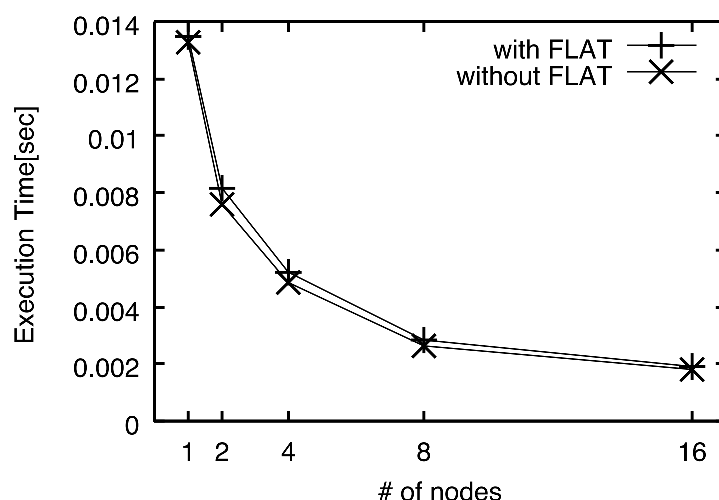


図 5.3.1: Livermore ループ Loop18 の実行性能

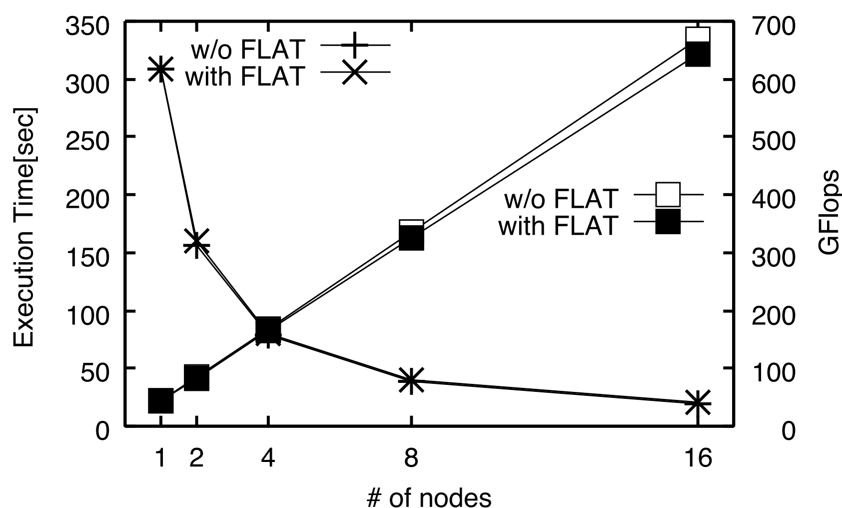
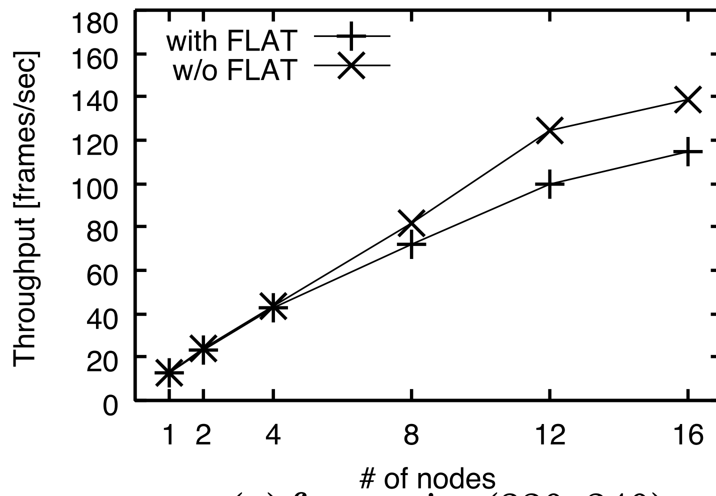
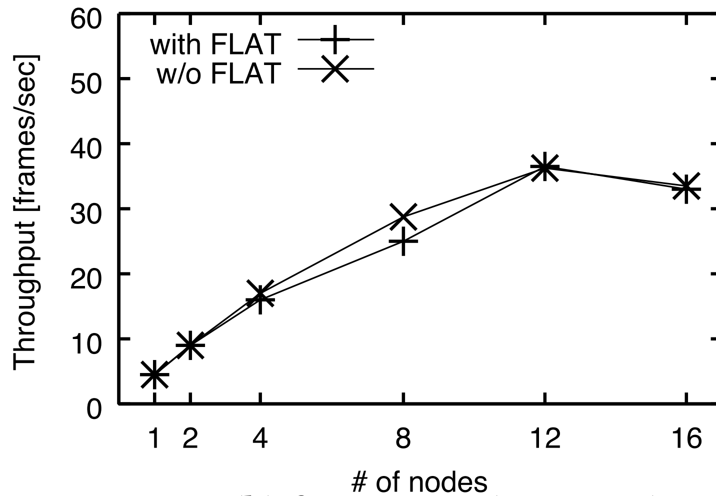


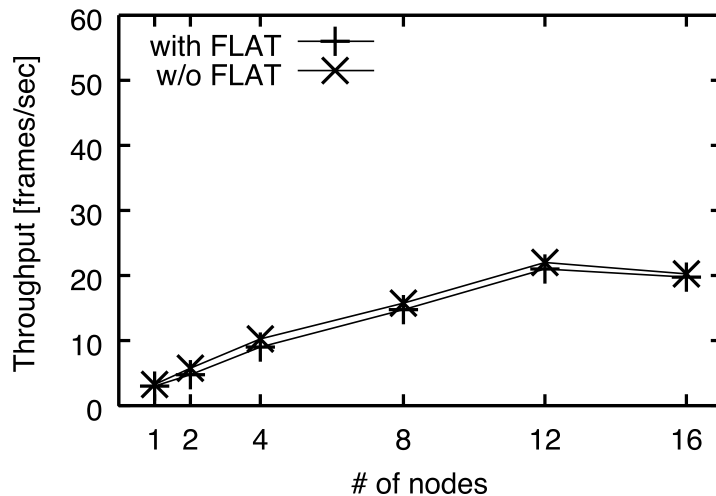
図 5.3.2: ステンシル計算の実行性能



(a):frame size (320x240)



(b):frame size (512x384)



(c):frame size (640x480)

図 5.3.3: オプティカルフロー計算の実行性能

第6章　関連研究

GPU クラスタを活用し，CUDA と MPI を容易にするプログラミングフレームワークを実現するために，Leung らは，R-Stream から複数 GPU アクセラレータへのソースコードからソースコードへの変換器を提案している [14]．このコンパイラでは，階層的な CPU 間，複数 GPU，GPU 内での並列化と分離を達成している．また，Lowlor は cudaMPI を提案している [15]．cudaMPI では GPU コードのために CPU コード中に記述すべき MPI 通信コードを簡潔に記述することができる．また，cudaMPI では，GPU-GPU 間のデータ転送で必要となる CPU 上の一時的なデータ転送用バッファを必要としない．しかし，この二つの研究では，あくまで CPU コード中に GPU コードのための MPI 処理を書く必要があり，GPU コードそのものに自然な形でデータ転送を実現するための MPI 処理の記述は実現できない．

Stuart らは GPU クラスタ上で GPMR の有用性を示した [16]. GPMR は MapReduce で書かれたプログラムに対し有用である．一方，FLAT は他の特定のメッセージパッシングを必要とするようなアプリケーションに重点をおいている．

HPCPE[17] では，CPU と GPU ベースのハイブリッドな並列計算プログラミング環境を提供している．HPCPE では与えられたプログラムを提唱する Two Level Model である計算タスクと制御タスクに分割し，それぞれ CPU と GPU に分割する．既存の並列プログラミングフレームワークである OpenMP や MPI の枠組みで GPU を活用するためのフレームワークとして，OpenMP から GPU コードに変換する手法 [18][19] や “軽量な”OpenMP と “重厚な”MPI の両方を使って複数 GPU を利用する [20]，および GPU や FPGA にタスクレベルで分割 [21] がある．さらに，既存の言語のフレームワークを利用して独自の言語を実現する DSL を，アクセラレータ用のプログラム記述に活用する研究もある [22][23]．これらは，プログラマにとって馴染み深いプログラミングインターフェイスを提供する．rCUDA[24] では，CUDA 言語を拡張し，専用ライブラリを用いることによって，ネットワークを介して複数 GPU を透過的に扱うことができる．rCUDA を除くこれらの文献に，複数アクセラレータの活用については言及されていないが，提案する MPI 埋め込み手法を用いることで，これらに対してデータ授受の仕組みを提供することができる．

NVIDIA GPU-Direct$^{\text{TM}}$[10] は，既存ハードウェア環境にそのまま適用可能である．GPU-Direct を用いることによってデータのコピー回数が減るため，高速なデータ転送を可能にする．また，GPU-CPU 間のデータ転送を記述をする必要がなくなるという利点を持つ．しかし，データ転送の発行は CPU コードで記述する必要があり，転送対象のデータ構造は CPU コードで管理する必要がある．従って，GPU コードでは転送対象であるデータを明示的に指定できないため，煩雑なメモリ管理は依然として存在する．FLAT の内部実装に GPU-Direct 用いることによって，メモリ管理コストが低減された高速な通信が可能になる．

文献 [9] では，ExpEther という仮想化技術を利用して，デバイス間を Ethernet を介して接続する，単一ホストでのマルチ GPU システムを提案している．しかし，システムを利用するために，設備を整える必要があるためコストが大きい．FLAT は MPI の埋め込みにコード変換を用いているため，安価な GPU クラスタシステムでもそのまま適用可能である．

複数 GPU を簡単に扱いたいという観点から、Kim らは GPU クラスタをあたかも 1 ノードのシステムのように扱う事ができるフレームワークを提案している [25]．このフレームワークは実行時，GPU にタスクを割り当てることで実現している．また，Bueno らは OpenMP のプラグマでタスクやタスク間のデータ入出力を与え，ランタイムシステムでタスク並列を実現する OmpSs[26] を提案している．プラグマは CPU コードに記述する必要があり，GPU コードそのものに自然な形でデータ転送を実現できない．従って FLAT は，これらフレームワークと類似するものではない．

第7章　結論

本研究では，MPIを埋め込み可能なGPUプログラミングフレームワークFLATについて開発した．FLATを利用することで，MPIによる通信の必要に応じてGPUコードが分断されることはないため，ひとまとまりの計算を一つの関数として定義可能になる．また，プログラマはCUDAによるGPUコードとMPIによるノード間でのデータ授受のコードの二種類のコードを管理する必要がなくなる．さらに，データ転送のためのCPUコード上での便宜的なメモリを意識する必要がないことから，プログラミングコストが軽減される．従って，我々の提案するFLATは，安価で高性能な計算環境を提供するGPUクラスタの利用を促進することができる．

本論文は，まず，FLATの実装手法を示した．次に，Livermoreループの Loop18，オプティカルフロー計算，ステンシル計算の三つの実プログラムをFLATで実装し，プログラマビリティと実行性能で評価した．FLATを用いることで，プログラマが管理すべきメモリ数，およびプログラマが記述する通信数が削減できることを示した．

実行性能では，粗粒度並列処理の場合，FLATのオーバヘッドは隠れるため，通常のプログラミング手法と比べて遜色ない性能が得られた．具体的には，オプティカルフロー計算における画像サイズが512x384と640x480の場合，またステンシル計算でw/o FLATに対するwith FLATの性能低下率は16ノード実行で3%以下であった．

今後の課題として，細粒度並列処理に対する最適化が考えられる．具体的には，カーネル関数の中断・再開のためのコンテキストスイッチの軽量化が挙げられる．FLATはMPI関数の度に，データ転送用のメモリを確保/解放する．そこで，FLATに制御が移る前に，あらかじめメモリを確保し，MPI関数でメモリが必要なときに確保したメモリを再利用するなど，最適化手法を検討する．

謝辞

本研究を進めるにあたり，ご指導，ご助言を頂きました，吉永努教授に感謝の意を表します．共著としてごご協力いただきました，情報システム基盤学専攻の本多弘樹教授，近藤正章准教授，情報ネットワークシステム学専攻の入江英嗣准教授，株式会社イーツリーズ・ジャパンの三好健文様，修士論文をご指導いただきました吉見真聡助教に深く感謝いたします．また，多くのアドバイスや議論をしていただいた吉永・入江研究室の皆さんに感謝します．

参考文献

[1] TOP500 Super Computing Sites. The TOP500 List - November 2012(1-100). http://www.top500.org/list/2012/11/100.

[2] Green500 Energy-Efficient Super Computers Sites. The Green500 List - November 2012(1-10). http://www.green500.org/lists/green201211.

[3] Takashi Shimokawabe, Takayuki Aoki, Chiashi Muroi, Junichi Ishida, Kohei Kawano, Toshio Endo, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, 2010.

[4] D. Jacobsen, J. C. Thibault, and I. Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. In *American Institute of Aeronautics and Astronautics (AIAA) 48th Aerospace Science Meeting Proceedings*, pp. –, 2010.

[5] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *J. Comput. Phys.*, Vol. 229, pp. 7692–7714, October 2010.

[6] NVIDIA CUDA Compute Unified Device Architecture. http://developer.nvidia.com/category/zone/cuda-zone.

[7] A.M.Devices Accelerated Parallel Processing (APP) SDK (formerly ATI Stream). http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx.

[8] OpenCL. http://www.khronos.org/opencl.

[9] 野村鎮平, 中浜徹也, 樋口淳一, 鈴木順, 吉川隆士, 天野英晴. Expether を用いたマルチ gpu システムの評価. 信学技報, 第 112 巻 of *CPSY2012-22*, pp. 79–84, 2012.

[10] NVIDIA GPUDirect™. http://developer.nvidia.com/gpudirect.

[11] Livermore Loops. http://www.netlib.org/benchmark/livermorec.

[12] Junichi Ohmura, Akira Egashira, Shunji Satoh, Takefumi Miyoshi, Hidetsugu Irie, and Tsutomu Yoshinaga. Multi-gpu acceleration of optical flow computation in visual functional simulation. In *Proceedings of the 2011 Second International Conference on Networking and Computing*, ICNC '11, pp. 228–234, 2011.

[13] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision (ijcai). In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pp. 674–679, April 1981.

[14] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 51–61, 2010.

[15] Orion S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *CLUSTER*, pp. 1–8. IEEE, 2009.

[16] J.A. Stuart and J.D. Owens. Multi-gpu mapreduce on gpu clusters. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1068 –1079, may 2011.

[17] Qing-kui Chen and Jia-kang Zhang. A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, pp. 86–89, 2009.

[18] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, Vol. 44, pp. 101–110, February 2009.

[19] 大島聡史, 平澤将一, 本多弘樹. OMPCUDA：GPU 向け OpenMP の実装. "2009 年ハイパフォーマンスコンピューティングと計算科学シンポジウム (HPCS2009)", pp. 131–138, 2009.

[20] Gabriel Noaje, Michael Krajecki, and Christophe Jaillet. MultiGPU computing using MPI or OpenMP. In *Proceedings of the Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing*, ICCP '10, pp. 347–354, 2010.

[21] Kuen Hung Tsoi, Anson H.T. Tse, Peter Pietzuch, and Wayne Luk. Programming framework for clusters with heterogeneous accelerators. *SIGARCH Comput. Archit. News*, Vol. 38, pp. 53–59, January 2011.

[22] 中里直人. アクセラレータを活用するためのプログラミング環境. 第 76 回情報処理学会プログラミング研究会, Vol. 3, No. 2, pp. 1–10, 10 2009.

[23] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pp. 835–847, 2010.

[24] J. Duato, A.J. Pena, F. Silla, J.C. Fernandez, R. Mayo, and E.S. Quintana-Orti. Enabling cuda acceleration within virtual machines using rcuda. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pp. 1 –10, dec. 2011.

[25] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pp. 341–352, 2012.

[26] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of gpu clusters with ompss. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 557 –568, may 2012.

[27] Message Passing Interface Forum. ”MPI: A Message-Passing Interface Standard Version 2.2”, Sep 2009.

[28] Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org/`.

[29] Intel MPI. http://software.intel.com/en-us/articles/intel-mpi-library/.

[30] HP MPI. http://welcome.hp.com/country/us/en/welcome.html.

発表文献

[1] 島 圭吾，三好 健文，近藤 正章，入江 英嗣，本多 弘樹，吉永 努，”MPI 埋め込み可能 GPU プログラミングフレームワーク適用可能性の評価”，並列／分散／協調処理に関する『鹿児島』サマー・ワークショップ（SWoPP 鹿児島 2011），2011 年 7 月 29 日

[2] Takefumi Miyoshi, Keigo Shima, Masaaki Kondo, Hidetsugu Irie, Hiroki Honda, and Tsutomu Yoshinaga. "FLAT: a GPU programming framework to provide embedded MPI", In Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5), David Kaeli, John Cavazos, and Enqiang Sun (Eds.). ACM, New York, NY, USA, 20-29.

付 録A　用語説明

A.1　HPC

High Performance Computing (HPC) とは，最新の技術を用いて性能の向上を図った計算機システムを用いた計算であり，主に，化学，電磁気，機械，金融などの工学分野における計算機を駆使した数値解析の分野で用いられる．HPC における計算性能は定量的なものであり，各時代間あるいは各システム間で相対的に評価される．また，HPC で扱うアプリケーションの計算量も 1 台のコンピュータ内で計算できるものから数千 ~ 数万の計算ノードを用いて構成されたスーパーコンピュータで計算されるものまで多岐にわたる．

計算処理を高性能化する方法は時代背景に依らず以下の二つに分類可能である．一つ目は計算量そのものを削減する方法である．スーパーコンピュータで扱われる問題では円周率の計算のような解答を求めるものではなく物理シミュレーションに代表されるような予測精度を求めるものが多い．したがって予測精度を高く維持しながら計算量そのものを削減するような計算アルゴリズムの最適化は計算処理の高速化に有効である．二つ目はシステムの単位時間あたりの計算量を向上させる方法である．計算スループットの向上方法についてはさらに二つに分類できる．

計算スループットの向上方法の一つ目は，単一の計算コアの性能を向上させる方法である．この方法にはプロセッサの設計の工夫や半導体プロセスの微細化に伴う動作周波数向上，専用命令を扱う計算コアの導入などが当てはまる．二つ目はタスクを分割し並列処理することでシステムにインストールされた計算ノード数，演算コア数に従った計算性能の向上を試みる手法である．

A.2　HPCC

コンピュータクラスタ (以下，クラスタ) は，多数のコンピュータをネットワークで相互接続したコンピュータシステムである．コンピュータアーキテクチャの観点から見たクラスタの利点は，主に二つである．一つはクラスタを構成するノード数の増加に従う計算性能の拡張性であり，もう一つは同じくノード数の増加に従う可用性の向上である．HPCC は特に前者の特性を活かし，多数の計算ノードを用いて並列計算を行うことで高い計算性能を実現するシステムである．

A.3　GPU

Graphics Processing Unit (GPU) は，コンピュータシステムにおいてコンピュータグラフィックス (CG) 等の画像処理を行うための専用プロセッサである．GPU は画像処理における並列度の高く大量のデータ処理に対応するために，シンプルな演算コアを数十 ~ 数百個もつメニー・コアプロセッサとして構成される．そのため，ピーク性能が汎用 CPU よりも高い．こうした GPU の特徴から，GPU を汎用のベクトル型プロセッサとして汎用目的に利用する General Purpose computation using on GPUs (GPGPU) が試みられている．

A.4 MPI

Message Passing Interface (MPI) は，分散メモリ型並列計算機における計算ノード間のメッセージ通信 API である [27]．MPI により，複数の CPU が情報をバイト列からなるメッセージとして送受信することで協調動作を行うことが可能である．

MPI には，各並列計算機ベンダが自社マシン向けに作成した実装や，様々なプラットフォームで利用可能なフリーウェアの実装も存在する．Ethernet 環境における MPI の実装例として，OpenMPI[28] や，MPITCH などがある．さらに，Intel は共有メモリ，TCP ソケット，InfiniBand 等の多数の DAPL ベースのインターコネクトでの使用を可能とする Intel MPI を提供している [29]．また，ヒューレットパッカードは自社の Linux，Windows サーバやワークステーション向けに HP MPI を提供している [30]．本研究の実験で用いた MPI は OpenMPI であり，バージョンは openmpi-1.3.3-gnu64-4.1.2 である．

A.5 CUDA

CPU よりも高いピーク性能を持つ GPU は，2000 年代よりプログラミングのできる演算コアのプログラマブルシェーダが GPU に搭載されるようになり，GPGPU の動きが起きた．しかし，当時の GPGPU 環境はグラフィックスパイプラインの知識が必要であった．そこで NVIDIA は，Compute Unified Device Architecture (CUDA) を発表した [6]．CUDA にはコンパイラ，ドライバ，デバッガ等のツール一式が提供されている．また，3D グラフィック処理の知識は不要で C ライクのマルチスレッドプログラミング環境である．さらに，CUDA プログラムは全ての CUDA 対応 GPU で実行可能である．本研究の実験で用いたバージョンは CUDA3.2 である．