Vol.2015-HPC-150 No.31
2015/8/6

等間隔格子を用いたSPH粒子法プログラムのCUDAによる高速化

高田 貴正[1]　大野 和彦[1]

概要：物理シミュレーション手法の一つである SPH 粒子法は，流体解析など様々な応用が可能であり，幅広い分野での利用が期待されている．一方で，規模の拡大や高精度化に伴い計算コストが膨大になるため，高速化の研究が行われている．近年では GPU を汎用計算に用いる GPGPU を利用して，標準的な CPU 以上の処理の高速化を実現している．
GPU を用いた並列処理で高速な数値計算を行うためには，GPU アーキテクチャに合わせた低レベルなコーディングによる最適化が必要になる．しかし，最適化手法と GPU アーキテクチャの発展とともに，従来の並列化手法よりさらなる最適化の可能性が出てきた．
そこで本研究では，CUDA を使用し，等間隔格子を用いた空間分割法の SPH 粒子法プログラムに，配列を用いたハッシュの格子データ構造を使用し，さらに，粒子データのソートおよび SoA(Structures Of Array) を適用することで，コアレッシングアクセスによるデータアクセスの効率化を図った．本手法と従来手法を用いたプログラムとの実行時間の比較を行った結果，従来手法より実行時間を短縮することができた．

キーワード：GPU，CUDA，粒子法，SPH,

Acceleration of Uniform Grid-based SPH Particle Method using CUDA

Takada Kisei[1]　Ohno Kazuhiko[1]

Abstract: SPH particle method is widely used in various physical simulation fields such as fluid analysis. However, the demands for higher accuracy and larger problems increase the computation cost enormously. Thus the acceleration of the method have been studied. Recently, applying General Purpose computing on Graphics Processing Units (GPGPU) has achieved higher performance than using traditional CPU. High performance computing on GPU requires hand optimizations using low-level code considering the GPU hardware features. However, the recent advance on the GPU architecture and the optimization techniques enabled further improvement on implementing the method.
We implemented SPH particle method using space partitioning based on an uniform grid. Increasing *coalesced accesses* on GPU largely reduces the memory access cost and improve the performance. For this purpose, we introduced an array-based hash grid data structure with particle sorting and AoS(Array of Structure) to SoA(Structure of Array) conversion. As the result of evaluation, our scheme improved the performance of SPH particle method.

Keywords: GPU，CUDA，Particle method，SPH

1. はじめに

コンピュータシミュレーションは実現困難な実験を低コストで行うことができる．流体や固体といった連続体に関するシミュレーション手法の一つである粒子法は，連続体を粒子の集まりとして粒子同士の相互作用を計算することにより流体などをシミュレートする手法である．他の手法に対する利点として形状データの生成が容易であること，

[1] 三重大学大学院工学研究科
Mie University

大きな変形，ひずみを伴うシミュレーションを高精度に行えることなどが挙げられる [1]．代表的な粒子法に SPH 法がある．

SPH 法は流体解析以外にも構造解析や衝突解析などに用いる研究が進み，幅広い分野で利用されている．一方で問題規模の拡大や高精度化などに伴いシミュレーションにかかる計算コストが大きくなっている．

大量のプロセッサで並列に処理できる GPU は近年 CPU に比べて性能向上がめざましく，GPU に汎用的な計算を行わせる GPGPU では標準的な CPU 以上の処理の高速化を実現している [4]．これらのことから，GPU を用いた粒子法の高速化の研究が行われてきた [5]．しかし，GPU プログラミングでは GPU アーキテクチャに合わせたプログラミングをしなければその性能を十分に発揮できない．そこで本研究では配列を用いたハッシュの格子データ構造を用いデータアクセスの効率化を図り高速化を行った結果，従来手法より実行時間を短縮することができた．

2. 背景

2.1 GPU と CUDA

2.1.1 GPU

GPU は演算を行うコアを CPU よりも大量に搭載し多数の処理を並列に実行できる．

GPU ではいくつかのコアやレジスタ，キャッシュなどで構成された SM(streaming multiprocessor) を複数搭載しコアを分割管理している．大量に生成されたスレッドは 32 スレッド単位で分割され SM で実行される．この 32 スレッドのまとまりをワープという．SM 内ではワープ単位で SIMD 型の並列処理を行う．このとき，SM 内でロードストアなどによりストールした際に，実行ワープを切り替えることで他の演算などの命令を実行できる．これにより，メモリアクセスの大きなレイテンシを隠蔽することができる．

各コアがデータを要求し頻繁にアクセスを行うため，シェアードメモリやコンスタントメモリなど独自のメモリアクセス機構や，これらを利用した高速化手法がある [2]．その一つにコアレッシングアクセスがある．デバイスメモリのアクセスはキャッシュのラインサイズである 128 バイト単位で行われる．このため，ワープ内のスレッドが同時にアクセスを要求するアドレスが同一ライン内であれば，複数のデータを一度のアクセスでまとめて転送できる．このデータアクセスをコアレッシングアクセスという．

2.1.2 CUDA

CUDA は NVIDIA 社より提供されている GPGPU 用の SDK であり，C 言語を拡張した文法とライブラリ関数を用いて GPU プログラムを開発できる．

CUDA でのスレッドの管理モデルを図 1 に示す．スレッドはブロックにブロックはグリッドという単位でまとめられ管理される．ブロックサイズ (ブロック当たりのスレッド数)，グリッドサイズ (グリッド当たりのブロック数) はユーザーが指定する．

ブロック内のスレッドは全て単一の SM で実行される．ブロックサイズを大きくすることにより，一個の SM が並行実行するワープ数を増やすことでコアの稼働率を向上できる場合がある．

しかし，単一の SM で実行されるため，ブロック内のスレッドはレジスタ資源を共有しており，ブロックサイズを大きくするとスレッド当たりの利用できるレジスタ数が減少する．スレッド内で使用するレジスタ資源が不足した場合，レジスタの値は一時的にデバイスメモリに退避され必要になれば再びデバイスメモリからレジスタに戻される．このため，頻繁にデバイスメモリへの退避が発生するとかえって性能が低下する場合がある．

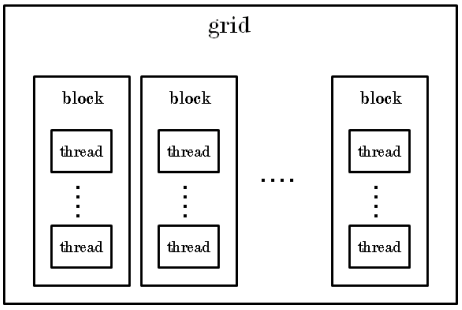


fig . 1 CUDA のスレッド管理モデル

2.2 SPH 粒子法

2.2.1 計算モデル

SPH 粒子法は元々は宇宙の銀河形成のシミュレーション手法として考案された [3]．この計算法を応用し，圧縮性流体や非圧縮性流体，構造解析など幅広い分野で利用されている．粒子は図 2 のような粒子データ配列 (粒子一個分の情報を表す構造体を要素に持つ配列) を用いて各粒子の相互作用を計算し，粒子の物理量を求める．

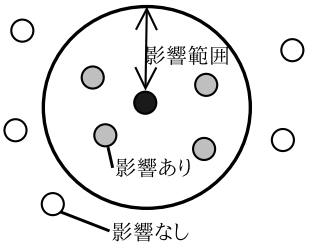
SPH 粒子法では図 3 に示すように粒子同士が相互に作用し合う一定の影響範囲があり，近傍粒子の探索で影響範囲内の粒子を見つけ，これらのデータを用い計算を行う．粒子は空間内を自由に移動できるため，タイムステップ毎に相互作用を及ぼす近傍粒子探索を行う必要がある．粒子が密集すると影響範囲内の粒子が増加するため，1 粒子当たりの計算のコストが大きくなる．

2.2.2 近傍粒子探索の効率化

近傍粒子探索の高速化手法として空間分割法がある．空間分割法では図 4 に示すように探索したい粒子がある空間を格子状に分割する．そして，計算対象となる粒子が含まれる分割領域と，その隣接する分割領域に含まれる粒子のみを探索することで，計算時間を大幅に減らすことがで

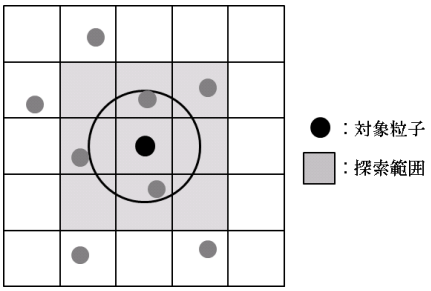
```
struct Particle{
 float posx, posy, posz;
 float velx, vely, velz;

 float density;
 float pressure;
 float force;
}particles[N];
```

fig．2 粒子のデータ構造



fig．3 粒子間で相互作用を及ぼす範囲

きる．空間分割法ではどのように空間を分割するかによりデータ構造や処理内容が決まる．代表的な手法として，すべての空間を同じ大きさの格子で分割する等間隔格子，格子のサイズを任意の大きさに調整できる階層格子などがある．等間隔格子を用いた空間分割法は利用できるデータ構造が幅広く，処理内容が単純であり最も並列化が容易であり，本研究ではこの手法を採用した．



fig．4 空間分割法による探索

2.3 GPU による粒子法

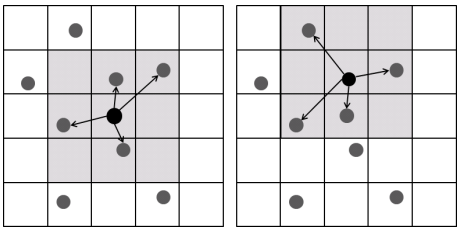
粒子法の並列化では，１つの粒子の物理量の計算を１つのコアに割り当て複数の粒子の計算を並列に行うことで高速化する．

空間分割法を用いた粒子法は計算の際，図 5 に示すように，一つの粒子の物理量を計算するために探索範囲に含まれるすべての粒子データへアクセスする．それぞれのコアが同様のアクセスを行うため，空間分割法を用いて探索を効率化しても，粒子データへのアクセスは頻繁に行われる．

3. 格子データの構造

3.1 従来手法

等間隔格子では分割した空間 (以降ボクセルと呼ぶ) 内に



fig．5 計算時のデータアクセス

存在する粒子を参照するためのボクセルデータに固定長の配列を用いる手法がある．図 6 は粒子が存在する二次元の空間を 4 分割した状態である．図 6 に対する固定長配列を用いたボクセルデータを図 7 に示す．図 7 では各ボクセルに対してサイズ 4 のポインタ配列を確保し，ポインタ配列にボクセル内の粒子データへのポインタを格納している．このポインタ配列によりボクセル内に存在する粒子を参照できる．粒子は空間内を自由に移動し，ボクセル内の粒子数は増減するためポインタ配列は十分な粒子数を許容できるサイズ (図 7 では 4) を確保しておく必要がある．

3.2 従来手法の問題点

固定長配列を用いた場合，ボクセルデータはシミュレーション空間の拡大やボクセル内に存在する粒子の最大数に比例して大きくなってしまう．また，粒子データ配列の並びは粒子が存在する空間とは独立しているため，各ボクセル内の粒子は粒子データ配列内にランダムに格納されている．このことより，以下の二点の問題が生じる．

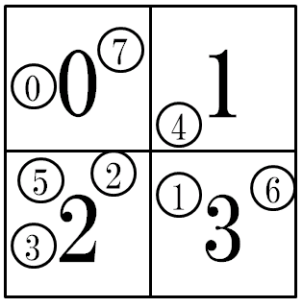
一点目は近傍粒子探索では同ボクセル内のすべての粒子にアクセスするため，探索のたび配列データの各要素にランダムアクセスすることになる．この場合，アクセスの際のキャッシュヒット率が低下し，レイテンシが大きくなってしまう．

二点目はワープ内の各スレッドが異なるデータを要求してしまうことである．スレッドが物理量の計算を担当する粒子の割り当ては，スレッドの識別 ID と粒子データ配列の要素番号に対応して割り当てられる．つまり，スレッド 0 が粒子データ配列の 0 番の要素，スレッド 1 が 1 番の要素というように割り当てられる．GPU ではスレッドはスレッドの ID が連続になるように分割される．前章で述べたようにワープ内のスレッドは同時に実行されるので，ワープ内のスレッドが要求するデータが近接している場合，重複した部分のメモリへのアクセスはまとめて行える．しかし，各ボクセル内の粒子は粒子データ配列内にランダムに存在するため，ワープ内の各スレッドが担当する粒子は別々のボクセルに存在する可能性が高い．このため，各スレッドが離れたアドレスのデータを要求することになり，まとめてアクセスできなくなる．

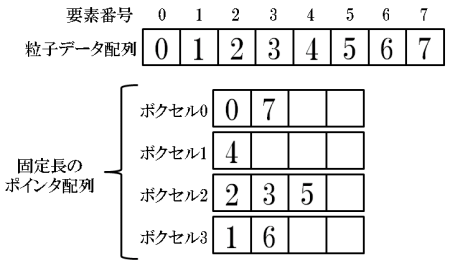
3.3 提案手法

従来手法ではボクセルデータの容量が大きく，粒子データ配列へランダムアクセスを行うという問題があった．そこで，メモリアクセスの効率化のために粒子データ配列を粒子が存在するボクセルの順にソートする．さらに，データサイズを抑えるために配列を用いたハッシュ格子を適用する．図 6 の状態に対し本手法を用いたボクセルデータを図 8 に示す．従来手法ではボクセル内の粒子の最大数に比例してポインタ配列を大きくする必要があり，さらに空間サイズにも比例して格子の構築に必要なデータが増加する．本手法ではボクセルデータは空間サイズに比例して大きくなるが，ボクセル内にどれだけの粒子が存在しても必要なデータが一定である．このため，従来手法に比べて少ないメモリで格子を構築できる．

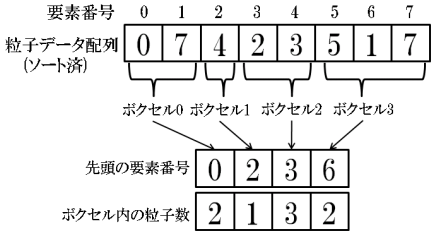
ボクセル内の粒子データを参照するには，粒子データ配列は粒子が存在するボクセルの順にソートされているので，ボクセル内の粒子データへのアクセスはボクセル内の粒子のうち粒子データ配列内で先頭となる要素からボクセル内の粒子の数だけ粒子データ配列をたどればよい．各ボクセルの先頭となる要素と各ボクセルの粒子数がわかっていればよい．このデータ構造では粒子データをソートする必要があるが，粒子の物理量計算の際，ワープ内のスレッドが要求するデータが同じになる可能性が高くなり，同ボクセル内の粒子へのアクセスが連続アクセスになるため，メモリへのアクセス効率が向上する．



fig．6 2 次元空間に存在する粒子



fig．7 固定長配列を用いた格子



fig．8 配列を用いたハッシュの格子

4. 実装方法

4.1 格子データの構築

格子データの構築は，各ボクセルの粒子数のカウント，カウント変数から各ボクセルの先頭粒子の要素番号算出，粒子データ配列のソートの 3 ステップで行う．まず，各ボクセルの粒子数のカウントは各ボクセルの粒子数カウント用の配列をすべて 0 で初期化した後，1 スレッドに 1 粒子を割り当て，各スレッドが担当した粒子が存在するボクセルのカウント配列の要素をインクリメントする．この時，別のスレッドが同様のボクセルのカウント変数に書き込むと競合するので，CUDA の atomic 関数による排他処理を用いた [6]．次に前ステップで求めたカウンタ変数の値からスキャンアルゴリズムを用い各ボクセルの先頭粒子の要素番号を算出する．スキャンアルゴリズムは GPU 上で並列実行可能なアルゴリスムである [7]．最後に各ボクセルの先頭となる要素番号が判明したので，その値を用いてバケツソートで粒子データ配列をソートする．各バケツに要素を追加する際のインクリメントでも競合が発生するので atomic 関数を用いる必要がある．このとき，粒子データ配列から同じ配列に書き込むとソート前のデータが上書きされてしまうので，同じサイズの配列を用意しそこに書き込む必要がある．

4.2 最適化

4.2.1 atomic 関数を用いない格子データの構築

CUDA の atomic 関数は排他処理ができるが性能低下を引き起こす要因となる．粒子法では atomic 関数を用いずに格子データを構築する手法が考案されている [8]．この手法ではインデックス配列やソート時の競合を回避するために同サイズのデータなどを用意する必要があり，1 粒子あたり合計 16 バイトのメモリが余分に必要となる．

4.2.2 SoA の適用

GPU では構造体の配列を使用する際，構造体の特定のメンバーへのアクセスを行うと，コアレッシングアクセスの効果が薄れる場合がある．このため，構造体の配列 (AoS) を配列の構造体に (SoA) に分割しコアレッシングアクセスの効果を向上させる手法がある．図 2 の粒子データ構造を SoA に変換した構造体を図 9 に示す．図 9 の構造体を

```
struct Particles{
 float posx[N], posy[N], posz[N];
 float velx[N], vely[N], velz[N];

 float density[N];
 float pressure[N];
 float force[N];
}particles;
```

fig．9 SoA に変換した粒子データ構造

用いることでコアレッシングによるアクセスの効率化が期待できる．また構造体の配列を用いる場合，粒子データのソートの際，データの上書きによる消失を避けるため同じデータサイズを用意する必要があったが，SPH 粒子法では図 2 のデータのうちソートする必要があるのは位置座標 (posx,posy,posz) と速度ベクトル (velx,vely,velz) のみである．SoA を適用しなかった場合，ソートする必要のないデータを含む配列をもう一つ確保するか，ソートする必要のあるデータサイズの配列のみ一つ確保し一時的にその配列に書き込んだ後，もとの粒子データ配列にその値を戻さなければならないが，SoA を適用した場合，ソートする必要のあるデータサイズの配列を確保しその配列に書き込んだ後，ポインタを付け替える高速かつ省メモリなソートができる．

4.2.3 ブロックサイズの変更による実行時間の変化

ブロックサイズは大きくしすぎるとスレッドあたりの使用できるレジスタ数が少なくなりレジスタの値をメモリに退避され性能低下を招く．しかし，ブロックサイズを小さくしすぎると SM 内でストールした際に切り替えるワープが少なくなり性能低下を招く．SPH 粒子法では複雑な計算を行うため，各スレッドが使用するレジスタの数が大きくなる．一方で，メモリアクセスを頻繁に行うため，レイテンシを隠蔽するにはブロックサイズを大きくすることが望ましい．そのため，適切なサイズを指定する必要がある．

5. 評価と考察

5.1 実験プログラムと環境

実験プログラムは SPH 法によるダム崩壊シミュレーションを行うプログラムで粒子数を 524,288 と 1,048,576 と二通りのシミュレーションを行った．評価環境はそれぞれ Intel Core i7-930，メモリ 6GB，Tesla K20c と Intel Xeon CPU E5-1620，メモリ 16GB，GeForce GTX980 を搭載した 2 台の計算機で行った．

5.2 固定長配列を用いた格子と配列を用いたハッシュ格子

固定長配列を用いた格子と配列を用いたハッシュ格子を使ったそれぞれの実装について，粒子数を変化させて実行した．それぞれの実行時間を表 1，2 に示す．いずれの場合においても配列のハッシュ格子を用いた方が実行時間が約 1/2 になっている．このことより，ソートによる計算コストを増加させてもデータアクセスの効率化により実行時間を大幅に短縮できることが分かる．

表 1 Tesla k20 での実行時間 (秒)

		固定長配列の格子	配列のハッシュ格子
粒子数	524,288	334.3	111.5
	1,048,576	656.3	227.5

表 2 GeForce GTX980 での実行時間 (秒)

		固定長配列の格子	配列のハッシュ格子
粒子数	524,288	122.7	53.9
	1,048,576	207.8	111.8

5.3 atomic 関数の使用/不使用の比較

atomic 関数を使用する格子データ構築法と使用しない手法の比較を行った．それぞれの実行時間を表 3，4 に示す．いずれの場合においても atomic 関数を使用しない場合，実行時間を約 10 %短縮できることが分かる．SPH 粒子法では粒子の物理量の計算コストが大きく格子データ構築にかかる時間は相対的に小さいが，データ構築の高速化により約 10 %の高速化が可能であることが分かった．

表 3 Tesla k20 での実行時間 (秒)

		atomic 関数有	atomic 関数無
粒子数	524,288	154.7	137.9
	1,048,576	464.7	411.8

表 4 GeForce GTX980 での実行時間 (秒)

		atomic 関数有	atomic 関数無
粒子数	524,288	53.6	48.8
	1,048,576	165.5	147.8

5.4 SoA の適用の有無

粒子データ構造に AoS，SoA を用いた実装を比較した．それぞれの実行時間を表 5, 6 に示す．全ての場合において，実行時間が遅くなっており，SoA の適用によるメモリアクセスの効率化によって性能向上が得られなかったことがわかる．この原因として，自身の担当する粒子への読み書きはコアレッシングアクセスにより効率化できるものの，周辺の粒子データへのアクセスについては多くの場合ワープ内スレッドで同一粒子へのアクセスになることが挙げられる．後者についてはもともとコアレッシングアクセスであり改善効果が得られない一方で，AoS では一つの粒子データ全体が連続していたものが SoA では分散してしまうため，キャッシュからあふれやすくなるためと考えられる．

表 5 Tesla k20 での実行時間 (秒)

		未適用	SoA
粒子数	524,288	137.9	180.6
	1,048,576	411.8	540.3

表 6 GeForceGTX980 での実行時間 (秒)

		未適用	SoA
粒子数	524,288	48.8	53.5
	1,048,576	147.8	162.0

5.5 ブロックサイズ変更による実行時間計測

ブロックサイズを変更し実行時間の計測を行った．粒子数は 524,288 の場合のみを比較した．それぞれの実行時間を表 7，8 に示す．ブロックサイズは 128 の時がもっとも実行時間が短くなった．SPH 粒子法では物理量の計算コストが，計算時に発生するメモリアクセスコストに比べて大きいこと，また，計算時に利用するレジスタが多いため，並行ワープ数が減少してもスレッド当たりの利用できるレジスタ資源を増やした方が実行時間を短縮できることが分かった．

表 7 Tesla k20 での実行時間 (秒)

		ブロックサイズ		
		128	256	512
粒子数	524,288	133.1	137.9	147.2

表 8 GeForce GTX980 での実行時間 (秒)

		ブロックサイズ		
		128	256	512
粒子数	524,288	47.6	48.8	50.4

6. おわりに

本研究では，GPU 上での空間分割法を用いた SPH 粒子法のさらなる高速化のため，配列を用いたハッシュ格子による実装を提案し評価を行った．さらに，格子データ構築の高速化，SoA によるアクセスの効率化により実行時間の短縮を図り，SPH 粒子法の計算コストとメモリアクセスコストの大きな処理に対してブロックサイズの変更による影響の調査を行った．その結果，配列を用いたハッシュ格子の適用，格子データ構築の高速化は実行時間短縮のため有用なことが分かった．さらに，SPH 粒子法ではブロックサイズを小さくして，スレッド当たりが利用できるレジスタを増やすことが効果的なことが分かった．

しかし，SoA の適用によるメモリアクセスの効率化は効果が得られなかった．これは配列のハッシュ格子の適用により，近傍粒子探索の際ワープ内のスレッドが同じ粒子データにアクセスする確率が高まる．粒子のデータ構造を SoA にした場合一つ当たりの粒子が持つ情報はメモリ上に分散されてしまうため，一つの粒子が持つ全フィールドへのアクセスにコアレッシングアクセスにならなくなったからだと考えられる．

今後の課題として，SoA 適用による利点とワープ内スレッドが同一粒子のデータにアクセスする際の効率を両立させる手法を実現する必要がある．

謝辞 本研究の一部は公益財団法人豊田理化学研究所の豊田理研スカラー助成による．

参考文献

[1] Reeves,W.T. *Particle Systems - a Technique for Modeling a Class of Fuzzy Objects.*, ACM Transactions on Graphics,pp.359-375, 1983.
[2] J. A. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. D. Liu, and W. Hwu. *timization and architecture effects on GPU computing workload performance.* In Proc.Innovative Parallel Computing 2012, InPar 2012, pages 1-10, 2012.
[3] J.J. Monaghan.Smoothed particle hydrodynamics. *Annu.Rev.Astrophys.*,Vol. 30, pp. 543-574, 1992.
[4] *GPGPU.org: General-Purpose computation on Graphics Processing Units*, 入手先 ⟨http://www.gpgpu.org/⟩,(2013.06.22).
[5] 原田隆宏，田中正幸，越塚誠一，河口洋一郎: グラフィックスハードウェアを用いた個別要素法の高速化，日本計算工学会論文集，Vol.2007，No.20070011(2007).
[6] *NVIDIA Developer CUDA Zone*, 入手先 ⟨http://developer.nvidia.com/category/zone/cuda-zone⟩, (2015.07.9).
[7] M.Harris. *Parallel Prefix Sum (Scan) with CUDA*,NVIDIA technical report,2007.
[8] S,Green. *Particle Simulation using Cuda*,NVIDIA technical report,2012.