

平成 25 年度

修 士 論 文

# コヒーレント X 線によるスペックル像の高速計算

平成 26 年 1 月提出

指導教員 雨宮 慶幸 教授

東京大学大学院新領域創成科学研究科

雨宮研究室      47-116042      太田 崇士

# 目次

第 1 章	序論	1
第 2 章	X 線散乱の基礎	2
2.1	小角 X 線散乱 . . . . .	2
2.2	コヒーレンス . . . . .	3
2.3	X 線光子相関分光法 . . . . .	5
第 3 章	高速計算の基礎	7
3.1	計算機の概要 . . . . .	7
3.2	コンピュータの高速化手法 . . . . .	10
3.3	現代のコンピュータ . . . . .	11
3.4	高速化を行う上での補足事項 . . . . .	13
第 4 章	計算機を用いたコヒーレント散乱像の計算	14
4.1	はじめに . . . . .	14
4.2	散乱像計算の高速計算例 . . . . .	14
4.3	スペックル散乱像計算アルゴリズム . . . . .	15
4.4	使用した計算機の詳細 . . . . .	16
4.5	プログラム . . . . .	18
4.6	計算例 . . . . .	19
4.7	高速化手法の検証 . . . . .	20
4.8	性能評価 . . . . .	27
4.9	ハードウェア性能の向上による高速化の検討 . . . . .	30
第 5 章	まとめと今後の展望	32
	謝辞	33
	参考文献	35



# 第 1 章

## 序論

コヒーレントな X 線が試料に入射すると、試料から生じる散乱波が干渉してスペックル像と呼ばれる散乱像が得られる [1]。これは小角 X 線散乱法で得られるような空間的に平均化されたなだらかな像と異なり、試料中の粒子配置などといった構造情報を強く反映している。こうして得られるスペックル像の時間変化を用いてサンプル中の粒子運動などのダイナミクスを測定する手法として X 線光子相関分光法 (X-ray Photon Correlation Spectroscopy : XPCS) がある。XPCS はスペックル像の強度の時間揺らぎから散乱強度の時間相関関数を試料のダイナミクスを測定する手法であるが、一部の単純な系を除いては、XPCS の結果から粒子の運動を定量的に議論するのは難しく、特にゴム中のナノ粒子の運動など、複雑な系に対しては、数値計算を利用した解析が求められている。

近年、コンピュータ技術の発展により、これまで困難であった大規模な物理シミュレーションが可能になりつつある。かつてはスーパーコンピュータに頼らざるを得なかった計算でも、近年普及が進む General-purpose Computing on Graphics Processing Units (GPU) やメニーコアプロセッサといった並列計算機を用いることで、研究室レベルでも容易に高速な計算が行われるようになった。並列計算とはプロセッサを複数用意することで複数の計算を一度に行う手法であり、スーパーコンピュータにも用いられている高速化手法である。一方で並列計算は全ての計算に対し適用できるわけではなく、計算機に適したアルゴリズム選択や最適化が不可欠である。

XPCS で測定されるデータは、ある瞬間の試料の構造から得られるスペックル像を長時間にわたって連続撮影したものである。これをコンピュータで計算するためには「撮影時間間隔ごとの構造変化のシミュレーション」「各時間における構造を反映したスペックル像の生成」の二つの過程が必要になる。将来的には前者について流体力学的相互作用を取り入れた分子動力学計算などの手法を取り入れることで、複雑な粒子運動の系で XPCS の解析が可能になると期待できる。本研究では後者のスペックル像計算について、並列計算機による高速計算の可能性について検討を行った。試料の規模が大きくなるほどスペックル像計算にかかる時間は増大することが予想され、これを高速に行うことは今後の XPCS の発展に重要な役割を果たすと考えられる。

本論文では、まず第 2 章で X 線散乱法の基礎として小角 X 線散乱および X 線光子相関分光法について述べる。次に第 3 章で計算機の基礎として一般に用いられているコンピュータの基礎的な解説や高速化手法を述べる。第 4 章では本研究で作成したスペックル像計算プログラムを示し、計算時間の振る舞いなどについて考察する。

## 第 2 章

# X 線散乱の基礎

本論文で扱う X 線光子相関分光法は、構造解析に用いられる小角 X 線散乱法を拡張し、試料のダイナミクスを測定する手法である。本節ではまず小角 X 線散乱法について述べた後、X 線光子相関分光法について解説する。

### 2.1 小角 X 線散乱

小角 X 線散乱 (Small-Angle X-ray Scattering : SAXS) は X 線を物質に入射した際に得られる散乱 X 線のうち散乱角が小さいものを測定する手法であり、主に非晶性物質の 1 ~ 100 nm の構造解析に用いられる [2]。

入射 X 線の波長を  $\lambda$  とする。散乱角  $2\theta$  に散乱される光は Bragg の式

$$\sin \theta = \frac{\lambda}{2d} \quad (2.1)$$

で決まる長さ  $d$  のスケールの構造情報を反映する。従って、大きな構造の情報は散乱角の小さい領域の散乱に現れる。

X 線が波数ベクトル  $k_0$  で入射し、 $k$  の方向に散乱することを考える。 $r$  だけ離れた 2 点からの散乱の位相差は、光路差  $\delta$ 、波長  $\lambda$  のとき

$$\Delta\phi = \frac{2\pi\delta}{\lambda} \quad (2.2)$$

となる。散乱ベクトルを

$$\mathbf{q} = \mathbf{k} - \mathbf{k}_0 \quad (2.3)$$

で定義すると、この式は次のように書ける。

$$\Delta\phi = -\mathbf{q} \cdot \mathbf{r} \quad (2.4)$$

・ 試料の電子密度が  $\rho(\mathbf{r})$  で表されるとき、試料全体からの散乱振幅は位相差を考慮して試料の体積  $V$  にわたって振幅を足し合わせた形で書くことができ

$$F(\mathbf{q}) = \int_V \rho(\mathbf{r}) \exp(-i\mathbf{q} \cdot \mathbf{r}) d\mathbf{r} \quad (2.5)$$

のようになる。

半径  $R$  の剛体球の場合、電子密度分布は球の中心からの距離  $r$  を用いて

$$\rho(r) = \begin{cases} \rho_0 & \text{for } r \leq R \\ 0 & \text{for } r > R \end{cases} \quad (2.6)$$

のように書け、これを Eq. (2.5) に代入すると

$$\begin{aligned} F(q) &= \frac{\rho_0}{q} \int_0^\infty 4\pi r \sin(qr) dr \\ &= \rho_0 v \frac{3(\sin qR - qR \cos qR)}{(qR)^3} \end{aligned} \quad (2.7)$$

のようになる。ここで  $v$  は球の体積である。

試料が希薄な場合には、散乱強度は各粒子からの散乱強度の和として表されるが、粒子の濃度が高い場合や凝集している場合など、粒子間の干渉効果が無視できないとき、散乱強度  $I(q)$  は粒子の形状に起因する形状因子  $F(q)$  と、粒子の配置に起因する構造因子  $S(q)$  に分けて考えなければならない。 $N$  個の粒子からなる系を考え、 $i$  番目の粒子の中心位置を  $r_i$  で表し、 $r = r_i + u$  とすると、散乱強度  $I(q)$  は

$$I(q) = \frac{1}{V} \left\{ \sum_{i=1}^N \exp(-iq \cdot r_i) \int \rho(u) \exp(-q \cdot u) du \right\} \times \frac{1}{V} \left\{ \sum_{j=1}^N \exp(-iq \cdot r_j) \int \rho(u) \exp(-q \cdot u) du \right\} \quad (2.8)$$

$$= \frac{N}{V} \left\{ \iint \rho(u) \rho(v) \exp(-iq \cdot (u - v)) du dv \right\} \times \left\{ \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^N \exp(-iq \cdot (r_j - r_i)) \right\} \quad (2.9)$$

となる。ただし積分範囲は粒子内である。Eq. (2.9) において、前半の項は粒子内の電子密度の自己相関関数であり、形状因子に対応する。後半の項は粒子の配置に起因する項であり、構造因子に対応している。粒子が単分散で等方的であれば、上の式は形状因子と構造因子の積、

$$I(q) = AF(q)S(q) \quad (2.10)$$

として書くことができる。

## 2.2 コヒーレンス

ここまでの議論では X 線が完全に平面波かつ単色であると仮定してきたが、現実に測定で用いる X 線は理想的な単色平面波ではなく、異なる波長、異なる方向の X 線を含んでいる。ここから X 線の干渉が起こる、すなわちコヒーレンスである距離に限界が生じる [3]。

まず、X 線が完全な単色光ではない場合の干渉への影響を考えるため、Fig. 2.1 に示す A と B という二つの波を考える。波 A と波 B は全く同じ方向に伝播しているが、波長が僅かに異なるものとし、波長をそれぞれ  $\lambda$ ,  $\lambda - \Delta\lambda$  とする。二つの波が波面 P において同位相であったとすると、二つの波が逆位相となる距離まではコヒーレンスが維持されているものと考えることができる。この距離を縦方向のコヒーレンス長 (時間コヒーレンス長)  $L_L$  と定義する。 $2L_L$  の距離を進むと 2 つの波は同位相になる。この  $2L_L$  の距離の中に波 A が  $N$  周期、波 B が  $N + 1$  周期だけ含まれていたとすると、

$$2L_L = N\lambda = (N + 1)(\lambda - \Delta\lambda) \quad (2.11)$$

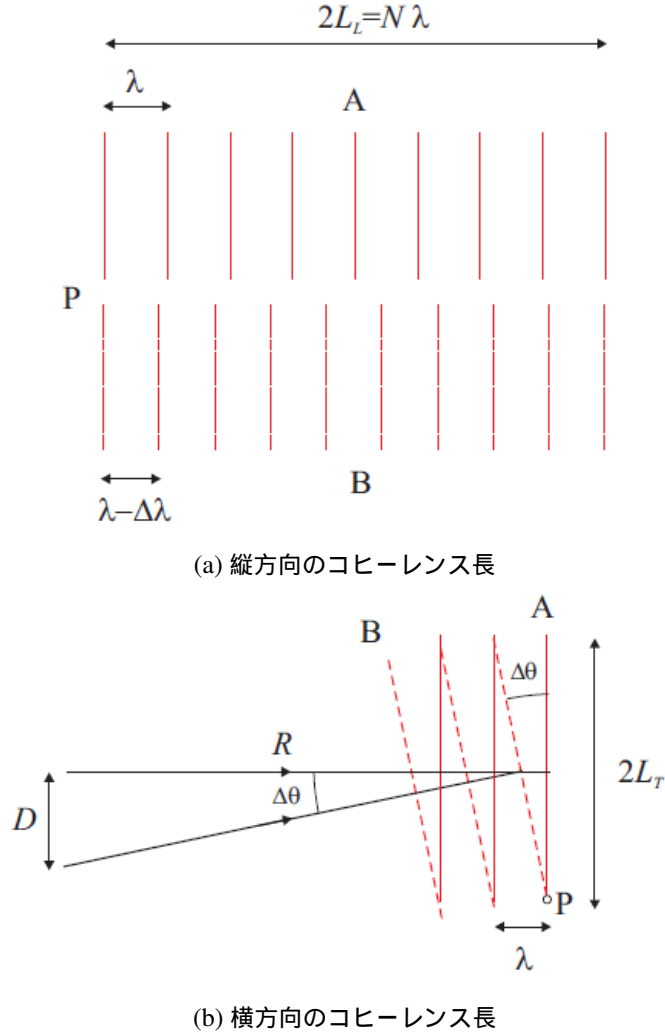


Fig. 2.1: コヒーレンス長の模式図 [3]

となる。2 つ目の等式から  $N \approx \lambda/\Delta\lambda$  となり、これを 1 つ目の等式と合わせると、縦方向のコヒーレンス長  $L_L$  は

$$L_L = \frac{1}{2} \frac{\lambda^2}{\Delta\lambda} \quad (2.12)$$

のように表せる。

次に X 線が平面波でないことの影響を見るために、Fig. 2.1 に示すもう一つの場合を考える。波 A、波 B は波長が等しく、伝播方向がわずかに異なっているものとする。点 P で 2 つの波の波面が揃っており、そこから角度  $\Delta\theta$  の角度だけずれて伝播すると考える。このとき点 P から離れていくと、ある距離で 2 つの波が逆位相になり、コヒーレンスが失われる。この距離を横方向のコヒーレンス長 (空間コヒーレンス長)  $L_T$  と定義される。  $2L_T$  の距離だけ離れると二つの波は再び同位相となり、

$$2L_T\Delta\theta = \lambda \quad (2.13)$$

$$\therefore L_T = \lambda/2\Delta\theta \quad (2.14)$$

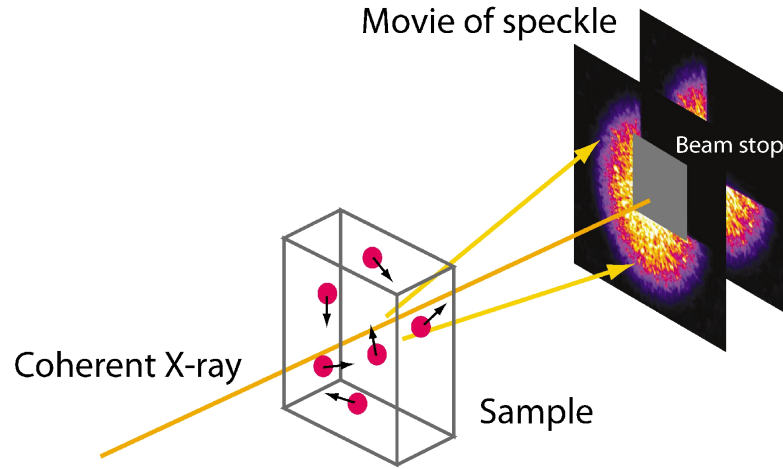


Fig. 2.2: XPCS の模式図 [4]

が成り立つ。単一の光源から発生する X 線のコヒーレンスを評価するため、波 A と波 B は同じ光源の距離  $D$  だけ離れた 2 点からそれぞれ放射されたと仮定する。観測点 P と光源の間の距離を  $R$  とすると、 $\Delta\theta = D/R$  となり、

$$L_T = \frac{1}{2} \frac{\lambda}{D/R} = \frac{\lambda R}{2 D} \quad (2.15)$$

が得られる。

コヒーレンス長の検討により、干渉を生じるような 2 点間の距離には上限が存在することがわかる。SPring-8 などの第 3 世代放射光源で光源サイズを評価すると、典型的には鉛直方向でおよそ  $100\mu\text{m}$  である。光源から 20 m の距離で波長 0.1 nm で実験を行う場合、横方向のコヒーレンス長  $L_T$  は鉛直方向でおよそ  $10\mu\text{m}$  である。同様に縦方向のコヒーレンス長  $L_L$  を評価する。 $L_L$  はモノクロメーターの単色性に依存し、例として完全結晶を用いる場合は単色度  $\Delta\lambda/\lambda$  は  $10^{-5}$  程度であり、 $L_T$  と同程度の長さになる。コヒーレンス長より遠く離れた 2 点間からの散乱はもはや散乱振幅で足し合わさることはなく、単純な強度和になる。

## 2.3 X 線光子相関分光法

X 線光子相関分光法 (X-ray Photon Correlation Spectroscopy : XPCS) は SAXS と同様、試料に X 線を照射して散乱される X 線を測定する手法であるが、照射する X 線がコヒーレントである点が SAXS と異なる [4]。コヒーレントな X 線を試料に照射すると、Fig. 2.2 に示すようなスペックル像が得られる。ビーム全体がコヒーレントな X 線を用いているため、平均化された情報しか得られない SAXS とは異なり、スペックル像には平均化されていない粒子位置が反映されている。そのため粒子位置が時間経過により揺らぐと、それに応じてスペックル像も変化する。従って、スペックル像の変化を解析することで、散乱角に応じたスケールの構造揺らぎに関する情報を得ることができる。

XPCS を応用した例として、ゴム中のフィラーのダイナミクス観察が挙げられる [5]。ゴムにカーボンブラックやシリカなどのナノ粒子を加えると、粘弾性特性などの物性が大きく変化し、タイヤなどへの応



用に欠かせないものとなっている。しかしフィラーの構造、ダイナミクスとマクロな粘弾性特性の変化との対応はついていない。ダイナミクスを観測する手法としては、可視光を用いて XPCS と同様の測定および解析を行う動的光散乱法 (Dynamic Light Scattering : DLS) がある [6] が、フィラー充填ゴムのように不透明な試料には応用することができない。そこで X 線を用いることで不透明な試料を観察可能な XPCS が、ゴム中でのナノ粒子のダイナミクスを解明する実験手法として期待されている。

XPCS で得られる情報について以下に示す。まず規格化された散乱光強度の時間相関関数を次のように定義する。

$$g_2(q, t) \equiv \frac{\langle I(q, t) \rangle \langle I(q, 0) \rangle}{\langle I(q, 0) \rangle^2} \quad (2.16)$$

ここで  $\langle \dots \rangle$  はアンサンブル平均を表す。測定により得られる散乱強度分布から  $g_2(q, t)$  を計算し、理論的なモデルから導出される関数でフィッティングを行うことにより、モデル中のフィッティングパラメータの決定を行う。

粒子の運動が Brown 運動であると仮定すると、この  $g_2(q, t)$  は次のように書くことができる。

$$g_2(q, t) = 1 + \exp(-2\Gamma t) \quad (2.17)$$

ここで粒子の拡散定数を  $D$  とするとき、 $\Gamma = Dq^2$  である。Eq. (2.16) で表される  $g_2(q, t)$  を測定し、Eq. (2.17) でフィッティングを行うことで、Brown 運動の拡散定数  $D$  を求めることができる。Stokes-Einstein の式により、 $D$  は Boltzmann 定数  $k_B$ 、温度  $T$ 、溶媒の粘性係数  $\eta$ 、粒子の流体力学的半径  $R_H$  を用いて

$$D = \frac{k_B T}{6\pi\eta R_H} \quad (2.18)$$

のように書ける。粒子系が既知であれば、この関係式から粘度  $\eta$  に関する情報が得られる。一方、DLS では主に粘度  $\eta$  を既知であるとして、測定結果から  $R_H$  の分布を求めるのに応用されている。

Brown 運動を示さない系については別の解釈が必要になるが、得られる情報は大きく制限されているのが現状である。例えば濃厚粒子系については  $g_2(q, t)$  の理論的な導出がなされていないため、経験的に Compressed Exponential Function と呼ばれる次の式でフィッティングを行っている [7]。

$$g_2(q, t) = g_\infty + A \exp \left[ -2 \left( \frac{t}{\tau} \right)^p \right] \quad (2.19)$$

ここで  $\tau$  は緩和時間を表す。また  $p$  は compressed exponent を表し、一般に  $1 < p < 2$  である。今のところ濃厚系で得られる情報は Eq. (2.19) のフィッティングで得られる  $\tau$  等に限られており、これらの解釈についても議論が分かれている。このような系については数値解析を利用した解析が必要となる。

## 第 3 章

# 高速計算の基礎

近年のコンピュータ技術の発展により、これまで困難であった大規模な物理シミュレーション等が可能になっており、科学技術の発展に大きな貢献をもたらしている。本章では高速計算に必要な計算機の基礎的事項や現代のコンピュータの成り立ちについて解説する [8]。

### 3.1 計算機の概要

#### 3.1.1 von Neumann 型コンピュータ

von Neumann 型コンピュータはコンピュータの動作モデルの一つであり、現在のコンピュータの基礎と言えるモデルである。次の特徴を持つ。

1. 記憶装置 (メモリ、主記憶) は 1 個だけあり、1 次元アドレスにより規定されるアドレス空間を構成する。
2. 演算を制御する命令がデータとともに記憶装置に記憶される。
3. 演算は命令の信号によって実行される。
4. 命令は、次に実行すべき命令のアドレスを保持する単一のプログラムカウンタにより、逐次的に実行される。

現在用いられているコンピュータは von Neumann 型コンピュータ (von Neumann architecture) として動作するように設計されている。そのためプログラムを作成する際は von Neumann 型コンピュータを前提に作成する。ただし後述するように現代のコンピュータは von Neumann 型から逸脱した設計を用いて高速化が図られていることに注意を要する。

von Neumann 型コンピュータはいわば裸のコンピュータであり、通常は von Neumann 型コンピュータの上で直接プログラムを実行することはせず、オペレーティングシステム (OS) の管理の下で動作させる。

Table 3.1: 浮動小数のフィールドの大きさおよび丸め誤差

	符号部	指数部	端数部	$\epsilon_M$
float	1	8	23	$2^{-24} \approx 10^{-7.2}$
double	1	11	52	$2^{-53} \approx 10^{-15.9}$

### 3.1.2 データ型と精度

コンピュータが取り扱う対象をデータと呼ぶ。データの種類は数、文字、画像、音声、動画など幅広い。ところがコンピュータが直接取り扱うことができる情報はビット列、すなわち二進数である。そのためデータは内部的に全て二進数で表現されており、必然的に情報量が制限されることから表現できる範囲の限界、精度の限界などを原理的に持つ。数値計算のためにコンピュータを用いる場合もこれらは無視できず、適切な評価を行う必要がある。

本節では本研究に特に関連の深いデータ表現である小数について、コンピュータ上でいかに表現されるかを解説する。本研究で扱うプログラムが C 言語で記述されていることを踏まえ、C 言語の基本型の名前を踏襲するものとする。

実数は数学的には連続量であり、無限に存在する。これをコンピュータ上で有限の大きさを持つデータとして表現するためにはある程度の誤差を伴う。

現在のほとんどのコンピュータでは IEEE754 と呼ばれる表現を用いている。これは 1985 年に制定された規格であり、IEEE 標準とも呼ばれる。IEEE 標準は浮動小数点数という形式を用いており、小数点の位置を固定せずに情報として保持することで広い範囲の実数を表現する。32bit 浮動小数と 64bit 浮動小数が主に用いられている。C 言語における取り扱いに従って 32bit 浮動小数を float、64bit 浮動小数を double と呼ぶこととする。float と double はどちらも次 3 つのフィールドを持つ。

1.  $s$  : 符号部 (sign)
2.  $e$  : 指数部 (exponent)
3.  $f$  : 端数部 (fraction)

各フィールドの大きさは float と double で異なる。大きさを Fig. 3.1 に示した。

浮動小数で表現できる数は数種類に分けられる。ここでは本研究に特に関連するものとしてゼロと正規化数について説明する。まず float を例に正規化数を説明する。float の場合は符号部  $s$  は 0 か 1、指数部  $e$  は 0 から 255 までの整数、端数部  $f$  は 23 桁の任意のビット列 (二進数) が格納されているものとする。  $0 < e < 255$  の場合について

$$(-1)^s 2^{e-127} (1.f) \quad (3.1)$$

の形で表される数を正規化数と呼ぶ。ここで  $1.f$  とは 1. の後ろに  $f$  のビットパターンを続けた二進小数を表す。ある小数を浮動小数で表す場合、最近接丸めすなわち表現可能な最も近くの値に丸めて表現するのが一般的である。正規化数で表現できない数としてゼロがあるが、 $e = 0$  かつ  $f = 0$  の場合を特別にゼロとして扱うことで対処する。以降、本論文においてコンピュータ上で小数を扱う際は、特に断らない限

り「正規化表現の浮動小数」または「ゼロ」であるものとする。

### 3.1.3 実数演算の誤差

実数を有限のビットで表現することで生じる誤差を丸め誤差と呼ぶ。 $x$  を正規化表現可能な実数とし、 $x$  の最近接丸めによる正規化表現を  $fl(x)$  と表すものとする。この二者の関係は

$$fl(x) = (1 + \delta)x \quad (3.2)$$

$$|\delta| \leq \epsilon_M \quad (3.3)$$

という式で表すことができる。ここで  $\epsilon_M$  は計算機イプシロンと呼ばれる数であり丸め誤差の限界を表している。float、double の  $\epsilon_M$  を Fig. 3.1 に示した。

実数を浮動小数として扱うことにより丸め誤差が生じるのであれば、演算により誤差がどのように伝播するのかを考える必要がある。有限精度演算を考える上で注意すべき点として、結合法則は必ずしも成り立たないという点がある。なぜなら計算順序によって演算誤差が変化するためである。ここでは基礎的な演算誤差として情報落ちと桁落ちを説明する。

まず 2 個の正規化数  $a$ 、 $b$  について、加減乗除のいずれかの演算  $\circ$  を行う場合の演算誤差を評価する。演算結果も正規化数であるとする。次のことが成り立つ。

$$fl(a \circ b) = (1 + \delta)(a \circ b) \quad (3.4)$$

$$|\delta| \leq \epsilon_M \quad (3.5)$$

1 回の演算で生じる演算誤差は相対誤差で  $\epsilon_M$  を超えないことがわかる。ところが複数回演算を行うことにより誤差が増大するということが起こりうる。例として 3 個の正規化数  $a$ 、 $b$ 、 $c$  の加算について考えると次のようになる。

$$\begin{aligned} fl(fl(a + b) + c) &= (1 + \delta_2)((1 + \delta_1)(a + b) + c) \\ &= a + b + c + \alpha \end{aligned} \quad (3.6)$$

$$|\alpha| \leq \epsilon_M(|a + b| + |a + b + c|) \quad (3.7)$$

ここで  $\delta_1$ 、 $\delta_2$  は 1 回目、2 回目の演算で生じる誤差を表す。また  $\delta$  の 2 次の項は無視した。この式からわかることとして、 $a$  の絶対値が小さく、 $b$  と  $c$  がほぼ相殺する場合は誤差が増大すると予想される。例えば

$$a = 1.01_2 \times 2^{-23}, b = 1.0, c = -1.0 \quad (3.8)$$

の場合について、データ型として float を用いる場合の誤差を考えてみる。計算順序を変えて計算すると

$$(a + b) + c = 1.0 \times 2^{-23} \quad (3.9)$$

$$a + (b + c) = 1.01 \times 2^{-23} \quad (3.10)$$

のようになる。Eq. (3.9) では有効数字の大部分が消えてしまっている。これは float の端数部が 24bit であることにより、 $a + b$  の計算において  $a$  の下位の情報が消えているためである。このように、絶対値の

大きさが異なる数の加算において小さい側の数の下位の情報が消えることを情報落ちと呼ぶ。一方で、値がほぼ等しい値の減算で生じる誤差の増大を桁落ちと呼ぶ。

誤差がどのように増大するかは計算によって異なり、それぞれ評価すべきである。演算誤差を低減するために、計算順序を工夫する等いくつかの方法を用いることができる。その一方で精度を高める工夫は計算時間の増加を招きがちであり、速度を優先し精度を許容するという選択も可能である。その場合も精度がどの程度まで確保されているかを評価しておくのが望ましい。

## 3.2 コンピュータの高速化手法

von Neumann 型コンピュータは命令を順番に実行していくことを基本とするが、現代のコンピュータはこれに忠実に従っているわけではなく、多数の高速化手法を組み合わせることで高速演算を実現している。本節ではコンピュータで広く用いられている高速化技術を、特定のハードウェアに限定しない形で解説する。

### 3.2.1 パイプライン処理

パイプライン処理とは、ある処理を実行している途中で次の処理の実行を開始してしまうことにより、複数の処理をオーバーラップさせることで高速化を行う手法である。本節では数値演算の実行をオーバーラップさせる技術である演算パイプラインについて、浮動小数 2 数の加算を例に説明する。浮動小数の加算で行われる処理は次の 4 つのステージに分割することができる。

指数部比較 (C)

加算を行う 2 数の指数部を比較し、差を求める

仮数部シフト (S)

指数部が小さい方の仮数部をシフトすることで指数部をもう一方に揃える

仮数部加算 (A)

仮数部同士を加える

正規化 (N)

加えた後の仮数部を正規化 ( $1.f$  の形に変形) し、指数部を修正する

この処理を配列に対して行う場合、すなわちある 2 つの配列を要素毎に加算し別の配列に代入することを考える。この場合、von Neumann 型コンピュータの動作に従うと、ある数について指数部比較を行えるのは一つ前の正規化が終了した後になる。ところが計算機の回路として見た場合は上記の 4 つの処理はそれぞれ回路として実装されており、例えば指数部比較を行っている間は他の回路は動作していないため、ある数の指数部比較を行った後、その数の仮数部シフトと次の数の指数部比較を同時に行う、といった形で流れ作業のように複数の計算を行うことができる。これが演算パイプラインの仕組みである。

配列に対する処理を演算パイプラインにより高速で行うことをベクトル処理と呼ぶ。またベクトル処理を行うプロセッサで構築された高速計算機をベクトルコンピュータと呼ぶ。1990 年代半ばまでのスーパーコンピュータはベクトルコンピュータが主流であり、地球シミュレータもその一例である。

### 3.2.2 命令の同時実行

von Neumann 型コンピュータはプログラムカウンタに従い命令を 1 つずつ実行するのが原則である。ところが依存関係のない計算は同時に行っても計算結果は変化しないため、同時に計算を行うことで高速化が実現できる。このような処理は von Neumann 型に従わないが、計算結果は von Neumann 型と変わらないように実装される。

命令の同時実行の例として

$$t_1 = a \times b \quad (3.11)$$

$$t_2 = c \times d \quad (3.12)$$

$$e = t_1 + t_2 \quad (3.13)$$

という計算を行う場合、2 つの乗算の間には依存関係が無いため同時実行が可能である。一方で乗算と和算の間には依存関係があり、和算を行うには乗算の結果が出るまで待たなければならない。このような依存性の判定は基本的にコンパイラが行うか、実行時にプロセッサが判断することになる。依存性がどのように表れるか、すなわち同時実行しやすいかどうかはプログラムによって異なり、高速演算のためにはハードウェアに適したアルゴリズム選択やプログラム設計が必要になる。

実装の例として SIMD(Single Instruction Multiple Data) 方式がある。SIMD 方式は 1 つの演算命令で複数データに対し同時に処理を行う方式である。SIMD 方式の命令に対応するプロセッサは SIMD 演算専用の回路を持つ。例として Intel のプロセッサに採用されている SSE(Streaming SIMD Extension) は 4 つの浮動小数に対して同一の命令を実行する。

### 3.2.3 並列処理

パイプライン処理や命令の同時実行がプロセッサの高速化技術であったのに対し、並列処理は複数のプロセッサを用いて高速演算を行う手法である。前述の高速化が実装された CPU(Central Processing Unit) を複数搭載して同時並行に動作させることを考える。この場合、複数の処理が原理的に同時に実行可能であるとき、その処理は各 CPU に割り振ることで CPU の搭載数に応じた高速化が可能になる。後述するマルチコア CPU や GPU(Graphical Processing Unit)、メニーコアプロセッサなどは、いずれも並列計算が可能ないように設計されている。

## 3.3 現代のコンピュータ

1946 年に世界初の汎用電子コンピュータとされる ENIAC が世に出て以来、技術革新によりコンピュータの性能は桁違いに向上した。コンピュータの進化の指標のひとつとして Moore の法則がある。これは元々は Intel の創業者の一人である Gordon Moore が Intel のプロセッサの複雑さ(部品数)がどのように増加するかを予測したものである。Moore の法則はあくまで将来予測でしかないが、現在は「集積回路上のトランジスタ数は 18 ヶ月ごとに倍になる」という形に一般化され、一般の製品性能の目標値として利用されている。プロセッサの成長の歴史上にはボトルネックの出現とそれを打破するブレイクスルーの登

場が何度も繰り返され、プロセッサ性能は 2014 年現在もおおむね Moore の法則に従っていると言える。

現在はパーソナルコンピュータという形で一般家庭にもコンピュータが普及しており、我々にとっても身近なものとなっている。一方で、スーパーコンピュータと呼ばれる高性能コンピュータも存在し、科学技術計算用としてパーソナルコンピュータとは桁違いの計算能力を持つ。この 2 つは目的も発展の歴史も異なるが、決して無関係なものではない。技術の進歩によっていわば「安価かつ小型なスーパーコンピュータ」とも言えるようなプロセッサ (またはコンピュータ) が登場し、かつてはスーパーコンピュータなしには不可能だった計算がパーソナルコンピュータ上で行えるようになりつつある。

### 3.3.1 マルチコアプロセッサ

近年、コンピュータはパーソナルコンピュータの名で一般家庭に普及している。計算を行う CPU はかつては単一のプロセッサが主流であったが、1990 年代後半にプロセッサ単体での性能向上が頭打ちとなった。当時はクロック周波数を引き上げることでプロセッサの性能を高めていたが、同時にプロセッサからの発熱が増加し冷却が困難になったのである。そのため、以降は別の高速化手段として一つのプロセッサ内に複数の演算装置を組み込むことで並列計算を行わせるという手法がとられている。この手法をマルチコアと呼ぶ。一般に一つのプロセッサあたり数個のコアを持つ。

### 3.3.2 GPGPU

パーソナルコンピュータの多くはグラフィック処理を専用に行うプロセッサを持ち、このプロセッサは GPU(Graphics Processing Unit) と呼ばれる。GPU が行う処理は、ある決まった演算を多数のデータについて行う、という処理が主であり、汎用プロセッサとして用いられる CPU とは扱う問題が異なる。このような事情から、GPU は CPU と異なり並列性を重視する方向で発展した。

GPGPU (General Purpose Computation on Graphics Processing Units) はグラフィック処理に用いられていた GPU を汎用計算目的に転用したものである [9]。GPU は CPU と比較してクロック周波数が数百 MHz と低く抑えられている一方で、数百のコアを用いた高い並列計算能力を持つ。

### 3.3.3 メニーコアプロセッサ

パーソナルコンピュータにおいて限界が見えつつあったクロック周波数の向上という手法にかわって採用されたマルチコアプロセッサであるが、この流れを極限まで推し進めたのがメニーコアプロセッサであり、数十個のコアを持つ。Intel はメニーコア製品として Larrabee[10] を開発した。本研究で用いる XeonPhi は Larrabee の後継である。XeonPhi は 1 GHz 程度のクロック周波数のコアを 60 個程度持つ。

### 3.3.4 スーパーコンピュータ

科学技術計算を主要目的とする大規模コンピュータをスーパーコンピュータと呼ぶ。スーパーコンピュータの高速化はパイプライン化と並列化の二種類の方向性で進められてきた。スーパーコンピュータのうち前者を用いたものをベクトルコンピュータ、後者を用いたものを並列コンピュータと呼ぶ。現在は

コストパフォーマンスの観点から並列コンピュータが主流になりつつある。

スーパーコンピュータの性能を表す指標として FLOPS (FLoating point Operations Per Second) 値がある。これは 1 秒間に実行可能な倍精度浮動小数演算の数を表しており、M(Mega)、G(Giga)、T(Tera)、P(Peta) といった接頭辞をつけて TFLOPS などという形で用いられる。

コンピュータシステムの性能を評価するにはプログラムがどのような状況で動作するかを規定しなければならない。理想的な状況を仮定した場合の性能をピーク性能、プログラムなどによって処理すべき内容を規定して測定する性能を実効性能という。プロセッサの周波数等から計算できる性能は前者を、いわゆるベンチマークにより得られる性能は後者を表す。ピーク性能は (浮動小数点演算器数) × (演算器が 1 サイクルに実行できる浮動小数点演算数) × (周波数) で計算される。

## 3.4 高速化を行う上での補足事項

本節では高速化を行う上でここまで触れてこなかった点であるキャッシュメモリについて補足する。

### 3.4.1 キャッシュメモリ

キャッシュメモリは多階層キャッシュとなっていることが多く、CPU に近い側からレベル 1(L1) キャッシュ、レベル 2(L2) キャッシュなどと呼ばれる。CPU に近いものほど読み書きの速度が速い一方で容量が小さい。これはメモリは高速かつ大容量であることが望ましいが、両方を同時に達成することが難しいことによる。以上の理由から、プログラムの設計を工夫してハードディスクやメモリへのアクセスを減らし、可能な限りキャッシュを有効活用するのが望ましい。CPU がデータにアクセスする場合、まずは L1 キャッシュを探し、あればそれを使い、無ければ L2 キャッシュを探す。L2 キャッシュになればメインメモリから読み出す。読み出したデータはキャッシュに保持される。この仕組みにより、使用頻度の高いデータほど CPU に近いキャッシュに保持されやすくなり、高速にデータが読み出される。

一方で、von Neumann 型コンピュータは本来メモリを 1 つしか持たないことになっている。実際にプログラムを作成する際はキャッシュメモリの存在は隠蔽され、開発者はメインメモリのみ存在するものとしてキャッシュメモリを意識せずにプログラミングを行うことができる。

### 3.4.2 キャッシュブロッキング

プログラムによってはプロセッサの演算でなくメモリの読み書き速度が律速になるケースがあり、これを改善する方法としてキャッシュブロッキングと呼ばれる手法が有効であることが知られている [11]。

キャッシュブロッキングは計算に必要な情報を可能な限り L2 キャッシュに留め、メインメモリへのアクセスを少なくすることで高速化を行う手法である。アクセスしなければならないデータが L2 キャッシュの容量を上回っている場合、それらのデータはメインメモリに置かれ、必要に応じて L2 キャッシュ、L1 キャッシュに読み込まれる。ここでメモリ上のデータを分割して L2 キャッシュサイズ以下のサイズを持つブロックを作成し、ブロック毎に計算を行うようにプログラムを書き換えると、各ブロックの計算を行っている間は必要なデータが L2 キャッシュに保持されるようになり、メインメモリへのアクセスが抑制される。以上がキャッシュブロッキングによる高速化の仕組みである。



## 第 4 章

# 計算機を用いたコヒーレント散乱像の計算

### 4.1 はじめに

本章では、コヒーレント散乱像の高速計算について述べる。はじめに、散乱像の高速計算例を紹介し、その次に我々の系で高速計算を実装した結果を述べる。

### 4.2 散乱像計算の高速計算例

スペックル像の計算に限らず、特定試料からの X 線散乱像を計算により求めるということは広く行われている。ここでは計算内容が類似している Grazing-Incidence SAXS (GI-SAXS) における散乱像の計算についての研究例を紹介する [12]。

まず DWBA を用いると散乱強度は次のように書ける。

$$I(q) = \frac{k_0^4}{16\pi^2} |\Delta n^2|^2 |\Phi(\mathbf{q}_{\parallel}, k_{zi}^0, k_{zf}^0)|^2 \quad (4.1)$$

$$\begin{aligned} \Phi(\mathbf{q}_{\parallel}, k_{zi}^0, k_{zf}^0) = & F(\mathbf{q}_{\parallel}, k_{zf}^0 - k_{zi}^0) \\ & + r_{0,1}^f F(\mathbf{q}_{\parallel}, -k_{zf}^0 - k_{zi}^0) \\ & + r_{0,1}^i F(\mathbf{q}_{\parallel}, k_{zf}^0 + k_{zi}^0) \\ & + r_{0,1}^i r_{0,1}^f F(\mathbf{q}_{\parallel}, -k_{zf}^0 + k_{zi}^0) \end{aligned} \quad (4.2)$$

ここで  $\Delta n$  は粒子と基板の複素屈折率の差を表す。 $F$  は粒子の形状因子を表す。形状因子は SAXS の場合と同様に体積  $V$  にわたる積分

$$F(q) = \int_V \exp(-i\mathbf{q} \cdot \mathbf{r}) d^3r \quad (4.3)$$

により表される。この式は Green の定理を用いることで表面積分の形で書き直せる。

$$\begin{aligned}
 F(q) &= \int_V \exp(-iq \cdot r) d^3r \\
 &= -\frac{1}{q^2} \int_V \nabla^2 [\exp(-iq \cdot r)] d^3r \\
 &= -\frac{1}{q^2} \int_S \frac{\partial}{\partial n} [\exp(-iq \cdot r)] d^2r
 \end{aligned} \tag{4.4}$$

ここで  $\frac{\partial}{\partial n}$  は表面  $S$  上の面積要素  $d^2r$  に垂直な単位ベクトル  $n$  の微分を表す。

GI-SAXS の高速計算では粒子の形状因子を計算する。粒子の形状因子  $F(q)$  は次のように書ける。[13]。

$$F(q) = \int_S \exp(-iq \cdot r) dr \tag{4.5}$$

すなわち、粒子の形の情報は粒子の表面のみの情報で記述される。表面を完全に記述するためには無限の情報量が必要になるが、表面を有限個数の三角形の集合で近似することで、有限時間で計算が可能になる。三角形分割により形状因子の式は次のように書き換えられる。

$$F(q) = \sum_{t=1}^N \exp(-iq \cdot r) s_t \tag{4.6}$$

$N$  は粒子表面を三角形分割した場合の三角形の数、 $s_t$  は三角形の面積を表す。サンプル粒子の形状を  $r$  と  $s_t$  で記述することでこの式から形状因子を計算することができる。

形状因子から散乱像を計算するためには多数の散乱ベクトルに対し形状因子を計算する必要があり、この計算が実行時間の大部分を占める。この計算は各々の三角形の座標、各々の  $q$  座標について計算が独立であるため容易に並列化が可能であり、並列コンピュータでの計算に適する。この先行研究では多数の GPU を組み合わせた GPU クラスタやマルチコア CPU を用いて実際に計算が行われている。計算例として理論ピーク性能 1.03TFLOPS の GPU である Nvidia Tesla M2050 を並列計算機として用いることで、並列化を用いないコードを CPU 上で動作させた場合と比較して 125 倍の高速化を実現したとされる。

## 4.3 スペックル散乱像計算アルゴリズム

XPCS で用いられるスペックル像の散乱振幅は電子密度分布  $\rho(r)$  を用いて

$$E(q) = \int \rho(r) \exp(-iq \cdot r) \tag{4.7}$$

のように書ける。この式を出発点としてコンピュータ上で散乱像を計算することを考える。この式をコンピュータ上で扱う際、 $\rho(r)$  をどのように表現するかが課題となる。大きく分けて二つの方法が考えられる。

まず、SAXS の「散乱の情報を形状因子と構造因子に分割する」という考え方を踏襲したやり方が考えられる。例えばサンプルが何らかの液体中に分散した懸濁液のような構造であり、粒子が粒径の全て等しい球であれば、サンプルの構造は「球の構造」と「球の位置」の二種類の情報で非常に簡潔に表すことが

Table 4.1: Intel Xeon Processor E5-2650 の主なスペック

コア数	8
コアクロック周波数	2.0GHz
L2 キャッシュサイズ	20MB
SIMD ベクトル化サイズ	256bit
理論ピーク性能	128GFLOPS

できる。この手法は粒径が分布を持つ場合についても粒径の情報を追加することで容易に拡張することができる。

もう一つの方法として、 $\rho(r)$  をサンプリングによって離散的な情報として取り扱うこともできる。サンプルが粒子などの単純な系でない場合はこの方法をとらざるを得ないケースもあるが、サンプルの記述に必要な情報量が莫大になる。式は典型的な離散 Fourier 変換の式になり、高速 Fourier 変換などのよく知られたアルゴリズムを用いるのが適切であると考えられる。

本研究では前者の方法を採用した。この場合、使用する情報はサンプル粒子の位置を表す  $r_n$  と、計算対象となる  $q$  空間座標  $q$  である。 $r_n$  は 3 次元の実空間座標を粒子数  $N$  だけ持つ。 $q$  の座標数は自由に設定可能であり、ここでは  $q_x$  方向、 $q_y$  方向の座標数をそれぞれ  $N_{qx}$ 、 $N_{qy}$  として、 $N_q = N_{qx} \times N_{qy}$  とする。散乱振幅は

$$F(q) = \sum_n^N \exp(-iq \cdot r_n) \quad (4.8)$$

と書ける。

現行の並列コンピュータで高速な計算を行うためには、プログラムのアルゴリズム自体の並列性が重要になってくる。本節で示したアルゴリズムは各々の粒子座標、各々の座標について計算が独立であり、結果は計算順序に依らない。そのため容易に並列化が可能であり、並列コンピュータにより高速に実行することができる。

## 4.4 使用した計算機の詳細

本研究では計算機として「FUJITSU Server PRIMERGY TX300 S7」を用いた。PRIMERGY TX300 S7 が演算に用いるプロセッサは 2 種類あり、1 つはメインプロセッサとして動作する 2 基の「Intel Xeon Processor E5-2650」である。これはマルチコア CPU を 2 つ並列動作させるもので、位置づけとしてはコンピュータにおける CPU と同様に考えてよい。これとは別に計算用並列計算用コプロセッサとして「Intel Xeon Phi 5110P」が搭載されている。こちらは必要に応じて補助的に用いるコプロセッサと呼ばれるプロセッサである。主なスペックを Table 4.1 と Table 4.2 に示す。

Xeon Phi は前述のマルチコアコプロセッサの一種であり、60 個のコアを用いて高速な並列計算を行う。nVidia が主導する GPGPU と同様の目的で用いられるものであるが、Xeon Phi の初出荷は 2013 年であり GPGPU に比べて新しい製品である。GPGPU との最大の違いはアーキテクチャがパーソナルコンピュータ向け CPU の x86 アーキテクチャ互換であり、プログラミングが容易である点である。

Table 4.2: Intel XeonPhi Coprocessor 5110p の主なスペック

コア数	60
コアクロック周波数	1.053GHz
L2 キャッシュサイズ	512KB × 60
メモリサイズ	8GB
SIMD ベクトル化サイズ	512bit
理論ピーク性能	1.011TFLOPS

Xeon Phi のアーキテクチャである MIC の上で動作するプログラムは Intel が提供する Intel Compiler でコンパイルでき、言語は C、C++、Fortran が使用可能である。本研究では C 言語を用いた。

一般に並列化プログラミングは設計が難しい。これは一般的に使われるプログラミング言語が並列化を前提としていないのが理由である。そのため、並列化プログラミングの際には並列化プログラミング言語を用いるか、既存の言語を拡張する形で並列化をサポートする仕組みを用いる。前者の例として CUDA、後者の例として OpenMP、MPI などがある。ハードウェアにより可能な手法が変わってくる他、性能や得意とする問題も手法により異なる。本研究では C 言語上で OpenMP を用いて並列化を行った。これは OpenMP が非常に簡潔な C 言語風の記述で用いることができるため習得が早いと判断したためである。

なお、XeonPhi の特徴の一つとしてネイティブ実行があるが、本研究では用いないものとする。ネイティブ実行とはコプロセッサである XeonPhi 上で動作している OS にログインし、その上でプログラムを実行するものである。GPGPU 等、他の手法では用いることができない手法であり、汎用性が損なわれると判断して採用せず、GPGPU と同様に演算の補助として用いることにした。ネイティブ実行に対してこの実行方法をオフロード実行と呼ぶ。

ここで、理論ピーク性能について説明する。一般に、各々のプロセッサは性能の公称値として理論ピーク性能と呼ばれる指標を公開している。これは von Neumann 型コンピュータが 1 サイクルに 1 回の計算を行うことを利用し、対象とするプロセッサが 1 秒間に計算可能な浮動小数演算の回数を性能の指標とするものである。これに対し、実際のプログラムがそのプロセッサ上でどの程度の速度で動作するかを表したものが実効性能である。実効性能はプログラムにも依存することに注意を要する。

本研究で使用した Xeon Phi 5110p の理論ピーク性能 1.011 TFLOPS の計算方法は以下ようになる。並列計算可能な計算機の場合、1 秒間のサイクル数がクロック周波数で決まり、1 サイクルあたりの計算回数が並列化手法による増加の積で決まる。データシートによれば、Xeon Phi 5110p のクロック周波数は 1.053GHz となっている。またコア 60 個による並列化、512bit のベクトル化が可能である。さらに Xeon Phi は FMA と呼ばれる小数の積和算命令を持ち、 $a \times b + c$  といった演算を 1 サイクルで実行可能である。これによりピーク性能は

$$(\text{クロック周波数}) \times (\text{並列化度}) \times (\text{ベクトル化度}) \times (\text{FMA}) \quad (4.9)$$

$$=(1.053 \times 10^9) \times 60 \times (512/64) \times 2 \quad (4.10)$$

$$=1.011 \times 10^{12} \quad (4.11)$$

のように計算できる。

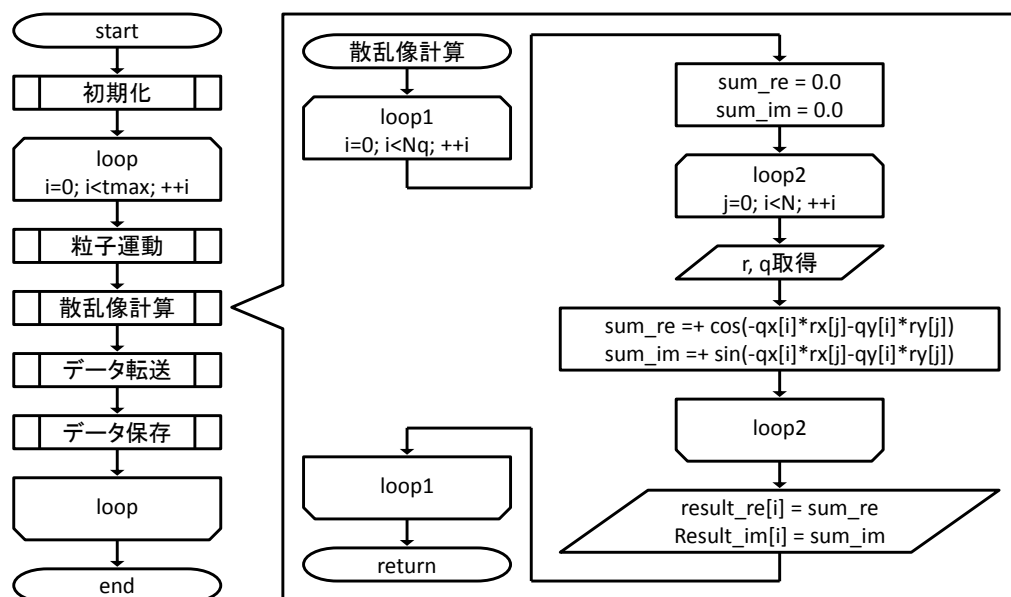


Fig. 4.1: プログラムフローチャート

以降、FUJITSU Server PRIMERGY TX300 S7 をホストと呼ぶことにする。

## 4.5 プログラム

これまでに示したアルゴリズムを元にプログラムを作成した。ここではプログラムのフローチャートのみ Fig. 4.1 に示す。

このプログラムでは入力として次のパラメータを受け付ける。

試料粒子数

試料中に含まれる粒子の数を表す。

試料サイズ (X)

試料の X 軸方向の大きさを表す。X 軸方向のビームサイズに対応する。単位は m とする。

試料サイズ (Y)

試料の Y 軸方向の大きさを表す。Y 軸方向のビームサイズに対応する。単位は m とする。

試料サイズ (Z)

試料の Z 軸方向の大きさを表す。試料の厚みに対応する。単位は m とする。

q 空間座標数 (X)

q 空間座標の X 軸方向のピクセル数を表す。

q 空間座標数 (Y)

q 空間座標の Y 軸方向のピクセル数を表す。

q 空間分割幅

q 空間座標の 1 ピクセルあたりの大きさを表す。単位は  $\text{m}^{-1}$  とする。

### ループ回数

粒子運動や散乱像計算等を行うループの試行回数を表す。

他、試料の粒子運動に関連するパラメータやデータ保存名等があるが省略する。

本プログラムは大まかに 4 箇所に分けて考えることができる。

### 粒子運動

XeonPhi 上であらかじめ定めた方法に基づき粒子運動をシミュレートする。

### 散乱像計算

XeonPhi 上で粒子運動計算結果の粒子位置を元に散乱像を計算する。

### データ転送

XeonPhi で得られた計算結果をホストに転送する。

### データ保存

ホストにデータ転送されたデータをハードディスクなどの記録媒体に保存する。

本研究では特に散乱像計算について重点的に検討を行った。粒子運動と散乱像計算は XeonPhi 上でオフロード実行される。データ転送はホストと XeonPhi の通信によって行われ、データ保存はホスト上で行われる。

散乱像計算について説明する。Eq. (4.8) に従って行う。この計算は Fig. 4.2 で示した。計算に用いるのは粒子位置  $r$  と  $q$  空間座標  $q$  であり、 $r$  は 3 次元座標が  $N$  個並んだ配列、 $q$  は 2 次元座標が  $N_q$  個並んだ配列である。 $r$  は粒子運動計算の部分で設定され、 $q$  はプログラムの初期化時に設定される。出力は散乱振幅  $F(q)$  であり、各  $r$  に対し定義され、実部と虚部を持つ。計算は各  $q$  について「全ての粒子に対して  $\exp(-iq \cdot r)$  を計算し、和をとる」という計算を行う。従って  $\exp(-iq \cdot r)$  の計算回数は  $N \times N_q$  回である。

## 4.6 計算例

計算の対象となる系は様々なものが考えられるが、まずは波長 0.1 nm、ビームサイズ  $5\mu\text{m} \times 5\mu\text{m}$  の正方形ビームを用いて直径 100 nm のランダム分散球サンプルを厚さ  $100\mu\text{m}$  で測定することを考える。体積分率は 10 % とする。この場合の粒子数はおよそ  $4.8 \times 10^5$  になる。

検出器の座標は実空間でなく  $q$  空間で与えるものとする。 $q$  空間の座標数と測定範囲は任意に定めることができるが、「スペックルが解像可能」「構造因子の第一ピークを測定可能」の二つの条件を満たすように設定するとしよう。ビームサイズを  $L$  とすると  $q$  空間におけるスペックルサイズ  $\Delta q$  は

$$\Delta q = \frac{2\pi}{L} \quad (4.12)$$

となる [14] ので、 $q$  空間の分解能はこれより小さくする必要がある。前述のサンプルの場合は  $\Delta q = 1.3 \times 10^6 \text{m}^{-1}$  となるため、ここではピクセルサイズを  $2 \times 10^5 \text{m}^{-1}$  とする。次に構造因子について、SAXS の理論を用いると、半径  $R$  の剛体球が体積分率  $f$  であるとき、構造因子  $S(q)$  は次のように書ける [2]。

$$S(q) = 1 - 8f \frac{3(\sin 2qR - 2qR \cos 2qR)}{(2qR)^3} \quad (4.13)$$

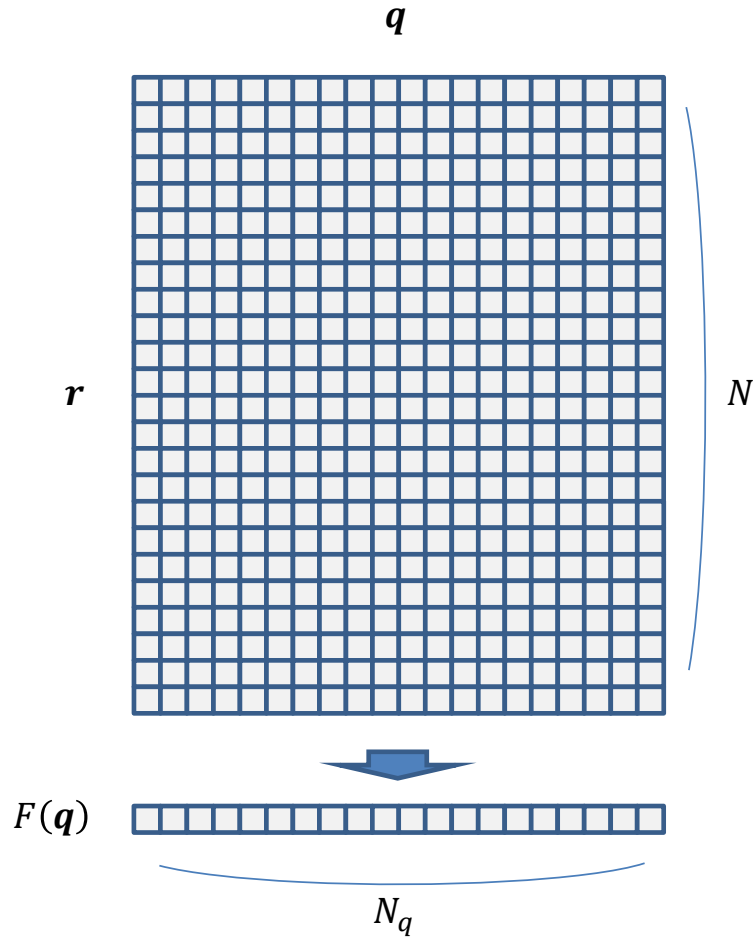


Fig. 4.2: 散乱像計算で用いられるデータの模式図。 $r$  は  $N$  個、 $q$  は  $N_q$  個存在する。 $r$  と  $q$  の組み合わせ毎に  $\exp(-iq \cdot r)$  を計算する必要があり、それを正方形で表した。同じ  $r$  のものは全て足し合わされ、 $F(q)$  に格納される。

これを前述のサンプルについて計算した結果を Fig. 4.3 に示す。粒子内干渉に起因したピークは  $6 \times 10^7 \text{m}^{-1}$  付近に現れる。ここでは 500 ピクセルに相当する  $1.0 \times 10^8 \text{m}^{-1}$  の範囲まで計算することにする。 $q$  空間座標数は  $2.5 \times 10^5$  となる。

以上を示した例を用いてスペックル像の計算を行った。結果を Fig. 4.4 に示す。プログラム内では電場の複素振幅が計算されるが、ここでは強度として示した。

## 4.7 高速化手法の検証

ここまでに作成したプログラムを対象として、実際に用いた、または利用を検討した高速化手法を以下に示す。特に重要なものとして、並列化とベクトル化がある。この 2 つは XeonPhi の重要な特徴と言えるものであり、高速化への寄与は極めて大きい。

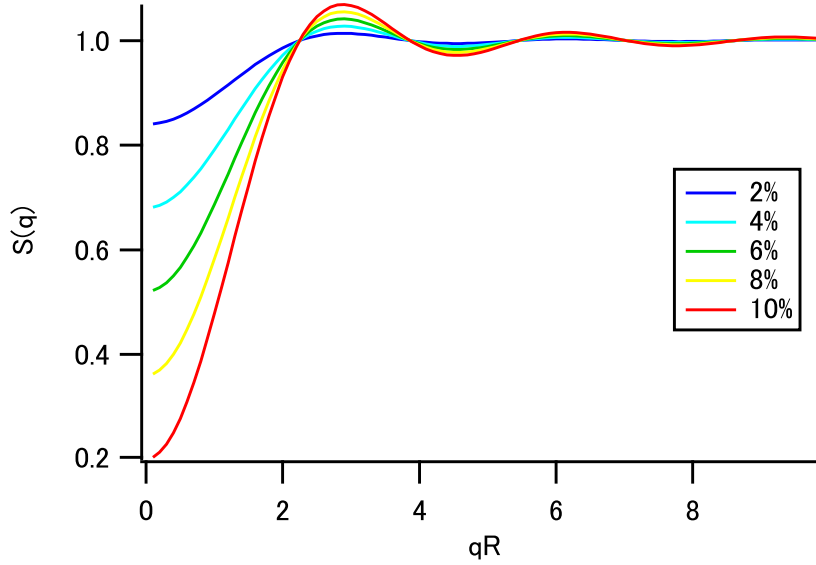


Fig. 4.3: Eq. (4.13) により計算した、体積分率を変化させた場合のサンプルの構造因子の計算結果。 $R = 5.0 \times 10^{-8}$  m の場合、近接粒子間距離に対応したピーク位置は  $q \approx 6.0 \times 10^7$  m になる。

#### 4.7.1 並列化

XeonPhi が持つ 60 個のコアを用いて並列化を行った。並列化プログラミングモデルとして OpenMP を採用した。OpenMP における並列化はプログラム内の for ループを対象に行われる。特定の for ループに「`#pragma openmp parallel for`」という指示文を加えることで、そのループで反復して実行される処理を各々のコアに割り振って計算を行う。本プログラムでは  $\exp(-iq \cdot r)$  の計算が 2 重ループになっており、外ループは計算対象の  $q$  座標を変化させている。この外ループを並列化に用いる。

#### 4.7.2 ベクトル化

XeonPhi は 512bit の浮動小数点レジスタを用いてベクトル化が可能である。これにより倍精度浮動小数数の場合は同時に 8 つの浮動小数数に対し計算を行うことができる。

本プログラムでは Intel C Compiler の自動ベクトル化を用いてベクトル化を行った。Intel C Compiler はベクトル化による最適化が指示された場合、最内ループのベクトル化を試みる。本プログラムでは  $\exp(-iq \cdot r)$  の 2 重ループの中で内ループ、すなわち計算対象の粒子を変化させるループをベクトル化する。

#### 4.7.3 並列化とベクトル化の影響

並列化とベクトル化を本プログラムに対し適用するとプログラムがどのように変化するかを以下に述べる、 $\exp(-iq \cdot r)$  の計算回数は粒子数  $N$  と  $q$  空間座標数  $N_q$  に依存し、 $N \times N_q$  回行われる。まず並列化を



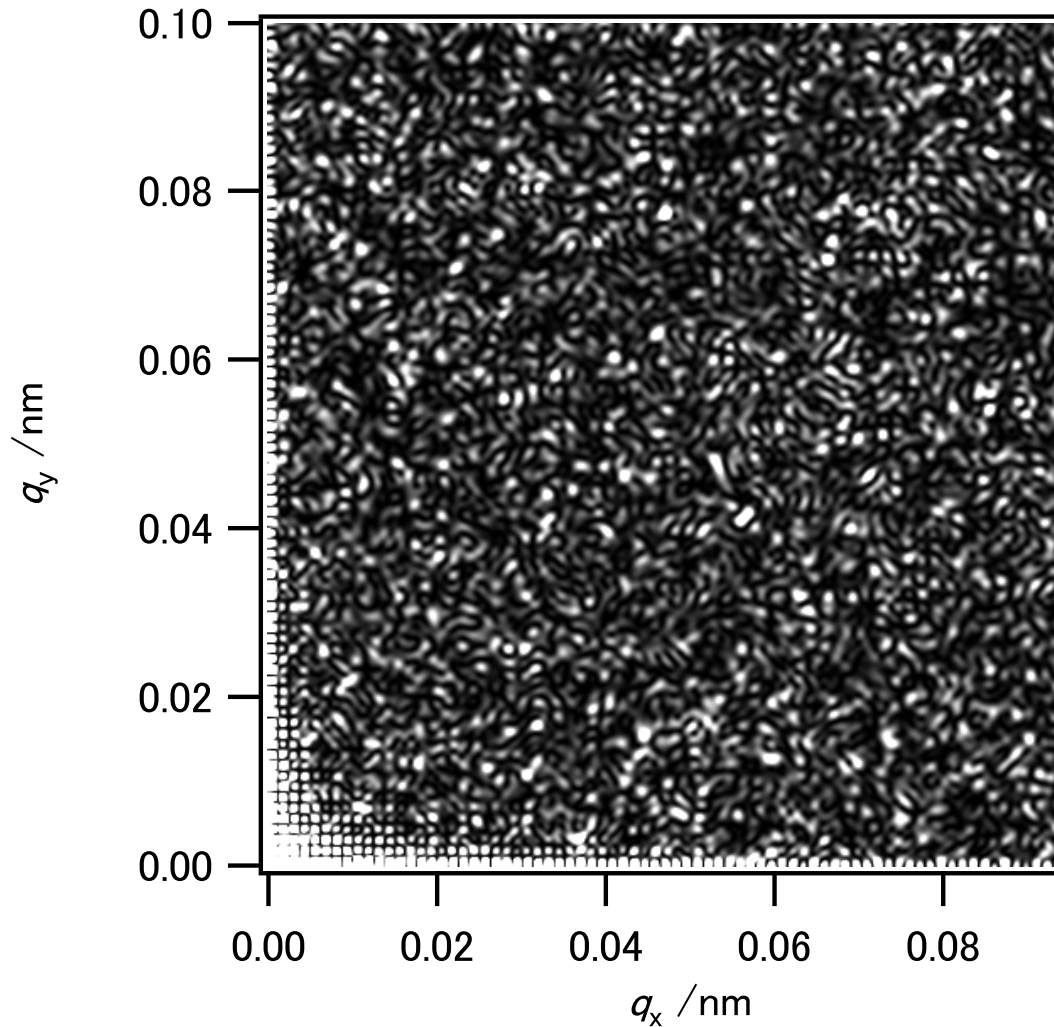


Fig. 4.4: スペックル像計算例

Table 4.3: スペックル像計算時間の比較。単位は秒。

	XeonPhi	ホスト
Parallel	$2.28 \times 10$	$1.0 \times 10^2$
Sequential	$2.437 \times 10^4$	$2.90 \times 10^3$

適用すると、各  $q$  空間座標について行う計算が各コアに割り振られ、1つのコアで扱う計算は「特定の  $q$  座標について全粒子の  $\exp(-iq \cdot r)$  の和を求める」という内容になる。各粒子の  $\exp(-iq \cdot r)$  を求める際にはベクトル化の適用により8個同時に計算される。

並列化とベクトル化を採用した場合の計算時間への影響を確認するため、XeonPhi およびホスト上で並列化やベクトル化を無効化して計算時間を測定した。並列化、ベクトル化を有効にした場合を「Parallel」、無効にした場合を「Sequential」と呼ぶこととする。測定結果は Table 4.3 に示した。結果を見ると、並

列化およびベクトル化を用いることで大幅な高速化が達成されたことがわかる。結果として XeonPhi を用いることでホストと比較して 4.1 倍の高速化が達成された。これは理論ピーク性能の差である 3.9 倍に近い値である。また、近年採用が進む並列化、ベクトル化等の技術を積極的に利用することの重要性も明らかになった。XeonPhi がメニーコアコプロセッサであるのに対し、ホストで利用されている Xeon はマルチコアプロセッサである。これはメニーコアコプロセッサや GPU ほど並列化を重視した製品ではないが、そのような環境でも高速化が見込めることから、コプロセッサを採用しない環境でも並列化は積極的に利用すべきであると言える。

Sequential と Parallel の違いであるが、XeonPhi では実行時間に 1000 倍以上の差がある。これは並列化やベクトル化の寄与から推測できる 480 倍と比較して 2 倍以上大きい。これは並列化やベクトル化を採用しない XeonPhi が何らかの理由で低速になっていると考えられるが、理由は本研究では明らかにできなかった。

仮に並列化やベクトル化を意識せずにプログラムを作成すると、散乱像計算は Xeon 上で Sequential で実行される。並列計算を最大限に活用することを考えると、XeonPhi 上で Parallel で実行することになる。この 2 つを比較すると実行速度の差は 127 倍に及ぶ。

#### 4.7.4 キャッシュブロッキング

先に解説したキャッシュブロッキングについて、本プログラムへの適用を試みる。

粒子数が  $N$  個のとき、粒子 1 個あたり 3 次元座標 1 個の情報を持つので、粒子の情報がメモリ内で占める領域は  $8B \times 3 \times N$  となる。 $N = 4.8 \times 10^5$  の場合は 11.5MB となり L2 キャッシュ容量を下回るが、粒子数が増していくと L2 キャッシュの容量を超えることになる。

キャッシュブロッキングによる高速化の検証のため、計算対象とする粒子位置の情報を一定サイズのブロックに分割し、そのブロックサイズを変化させて計算時間の変化を測定した。模式図を [?] に示す。今回は要素数が  $N$  個である  $r$  を  $b_r$  個ずつに分割し、また要素数が  $N_q$  である  $q$  を  $b_q$  ずつに分割した。これにより、Fig. 4.5 に示すように  $N \times N_q$  回行われる計算を 1 つあたり  $b_r \times b_q$  回分のブロックに分割し、各ブロックごとに分けて計算を行った。粒子数は  $N = 4.8 \times 10^6$  に、 $q$  空間座標数は  $N_q = 2.5 \times 10^5$  に固定した。さらに  $q$  空間座標のブロックサイズを  $5.0 \times 10^3$  に固定して粒子座標のブロックサイズを変化させて散乱像計算時間を計測した。結果を Fig. 4.6 に示す。この結果から、ブロックサイズによる計算時間の減少はほとんど無いことがわかる。これは本プログラムはメモリアクセスが律速でなく演算が律速であること示唆している。なおブロックサイズを小さくしていくことによる計算時間の増加はループ回数の増加による処理の増加が影響しているものと考えられる。

さらに検証を進めるため、本研究で作成したプログラムに変更を加えた別のプログラムを作成した。変更点は三角関数演算を省略したことである。このように変更すると、メモリアクセスの量はそのままに演算量が削減されるため、メモリアクセスが律速となる可能性が元のプログラムと比較して高い。この変更により目的を満たすプログラムではなくなっているが、ここではキャッシュブロッキングの効果を確認するために用いている。結果を Fig. 4.7 に示した。この結果を見ると、Fig. 4.7 と同様にブロックサイズが小さい場合の計算時間増大の傾向が見られる他、ブロックサイズが大きい部分でも計算時間の増大が見られ、キャッシュブロッキングのパラメータを適切に設定することで高速化が見込めることがわかった。

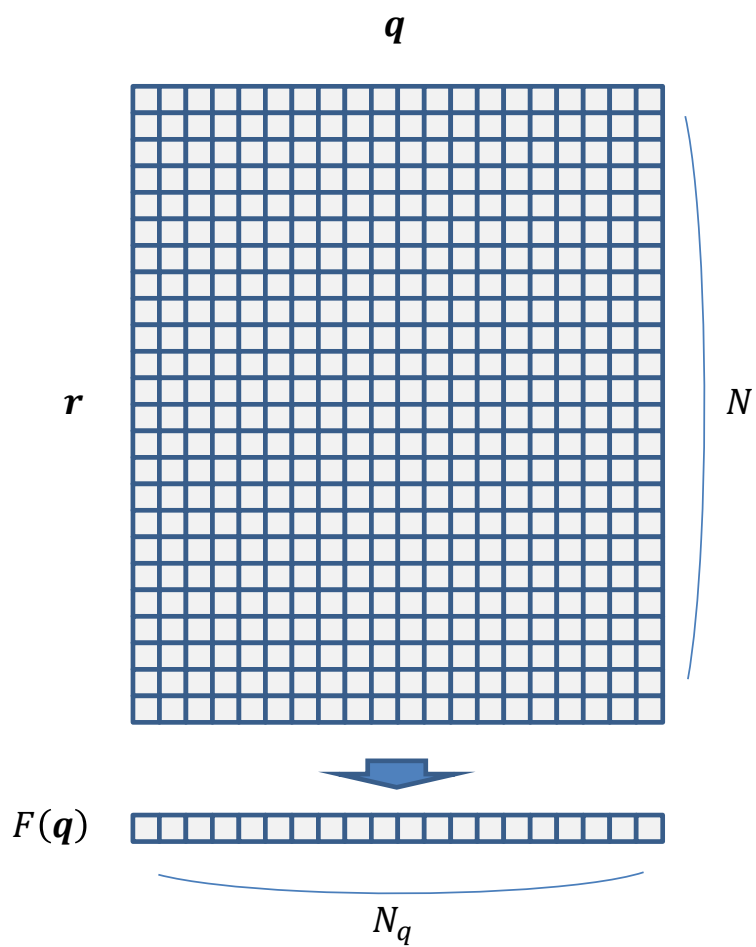


Fig. 4.5: キャッシュブロッキングの模式図

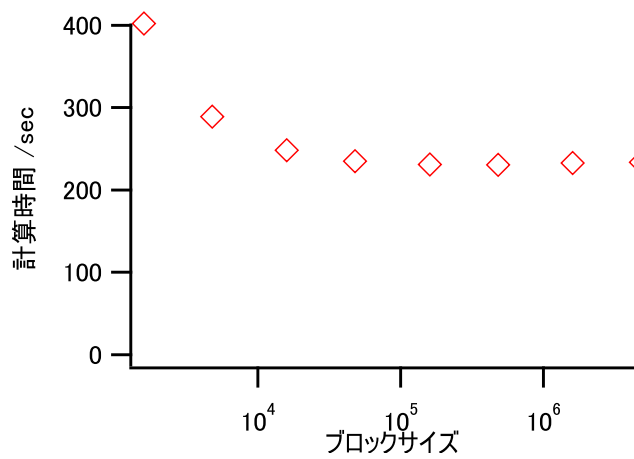


Fig. 4.6: キャッシュブロッキングのブロックサイズ変更による計算時間の変化

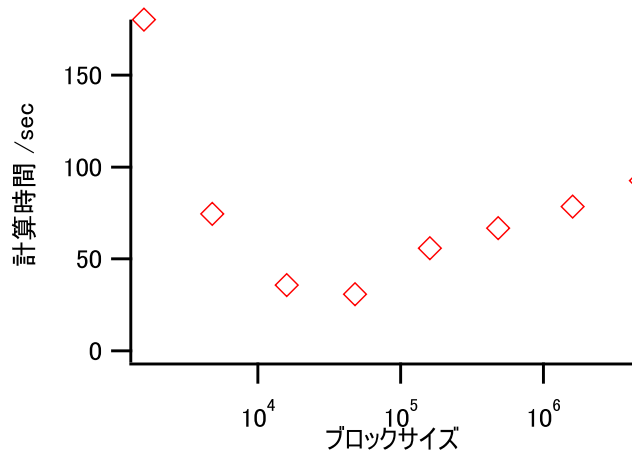


Fig. 4.7: キャッシュブロッキングのブロックサイズ変更による計算時間の変化。

ぎゃくに、本研究で作成したプログラムではこのような傾向が見られないことから、キャッシュブロッキングが有効でないことがわかる。

#### 4.7.5 精度低下による高速化

C 言語において浮動小数を扱う際、目的に応じて単精度と倍精度の 2 種の浮動小数を利用可能である。XeonPhi の場合、ベクトル化により同時に扱える処理の数はレジスタサイズで決まっているため、64bit の倍精度浮動小数から 32bit の単精度浮動小数に変更すると、ベクトル化による効果が 2 倍になり、高速化に繋がる。

一方で、丸め誤差が演算により蓄積、増大することにより生じる演算誤差は加算する数に応じて増加することが知られており、無作為な  $N$  個の和算における演算誤差は丸め誤差がランダムウォークを形成することから  $\sqrt{N}$  に比例する [15]。

ここでは浮動小数の型を倍精度から単精度に変更する場合、どの程度の誤差が生じるかを検証した。まず前述のプログラムを倍精度計算のプログラムとして用い、別に単精度計算のプログラムを作成した。単精度計算プログラムは粒子位置の情報を倍精度計算プログラムと同じ形で持ち、粒子位置から  $\exp(-iq \cdot r)$  を求める前に一度単精度浮動小数に変換し、以降は全て単精度で計算を行う。これにより単精度浮動小数を用いた場合の影響を評価した。パラメータは計算例のものを採用し、ビームが正方形であることから低  $q$  側に生じる散乱の影響を除くため、低  $q$  側の 100 ピクセルを除いた  $400 \times 400$  ピクセルを検証に用いた。

まず、倍精度で計算した散乱振幅の実数部についてヒストグラムを作成すると Fig. 4.8 のようになり、半値幅は  $6.9 \times 10^2$  となった。次に、単精度版と倍精度版で散乱振幅の実数部の差を取りヒストグラムを作成すると Fig. 4.9、半値幅は  $6.8 \times 10^{-2}$  となった。これらの結果から、倍精度から単精度への変更により  $9.9 \times 10^{-5}$  の相対誤差が生じると評価できる。

前述の通り、浮動小数演算において結合法則は必ずしも成立せず、計算順序を工夫することで精度低下を抑制できる場合がある。目的に応じて様々な加算アルゴリズムが提案されている [15]。一方で、こう

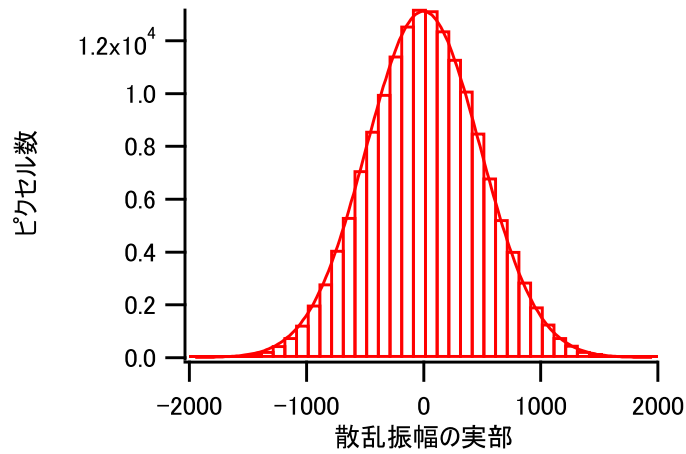


Fig. 4.8: 倍精度浮動小数による計算結果 (実数部) についてのヒストグラム。曲線は Gauss 関数によるフィッティング。

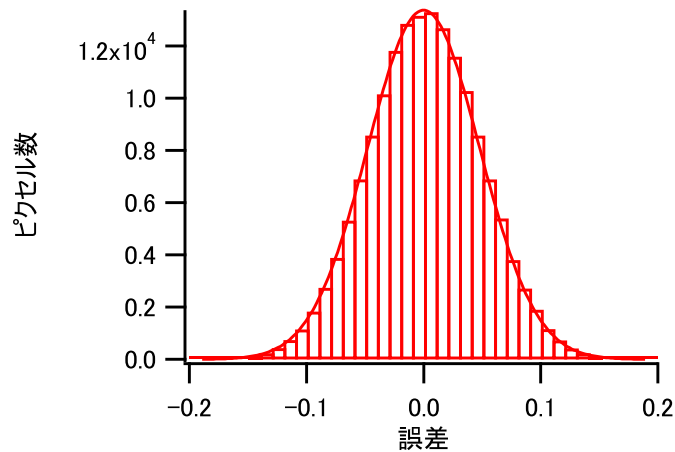


Fig. 4.9: 倍精度、単精度の計算結果 (実数部) の差についてのヒストグラム。曲線は Gauss 関数によるフィッティング。

いったアルゴリズムの採用は一般に計算速度を低下させるため、誤差が問題でないと判断できる場合には採用を見送るべきであると考えられる。なお、Intel Compiler は暗黙的に結合法則が成立するものとして最適化の際に並べ替えを行う [16]。誤差を低減する目的で何らかの加算アルゴリズムを採用する場合、コンパイラによって和をとる順序が変更されないよう、コンパイル時に順序変更を禁止することを明示する必要がある。

## 4.8 性能評価

### 4.8.1 パラメータ変化による実行時間変化の検証

計算時間のパラメータ依存性を確認した。パラメータの中で計算時間に関与すると考えられるパラメータは粒子数  $N$  と  $q$  空間座標数  $N_q$  である。この 2 つはプログラム内のループで反復回数を決めるため、プログラム全体の計算量を左右する。キャッシュブロッキングの例からわかる通り、計算量と計算時間が比例することは自明でない。

ここでは  $N$  や  $N_q$  を変化させたときに、実際に実行時間がどのように変化するかを検証した。まず  $N_q = 2.5 \times 10^5$  とし、 $N$  を変化させて散乱像計算、データ転送、データ保存にかかる時間を測定した。次に  $N = 4.8 \times 10^5$  とし、 $N_q$  を変化させて同様の測定を行った。これらの結果を Table 4.10、Table 4.11 に示す。

プログラムのアルゴリズムにより、散乱像計算時間は  $N \times N_q$  に、データ転送および保存時間は  $N_q$  に比例することが推測できる。測定結果もほぼ同様のものとなった。 $q$  空間座標数を変化させるときのデータ転送時間、データ保存時間は原点を通らないという結果が得られたが、これはデータ転送や保存のための前処理の時間が無視できないことによるものと考えられる。

計算時間のうち大部分を占めているのは散乱像計算時間であることにも注目しなければならない。これは計算時間のパラメータ依存性から自然なことである。

本章ですでに述べた高速化手法の多くは散乱像計算の高速化を目的として検討した。これは計算の高速化のためには計算時間が長い部分を重点的に高速化することが重要となるためである。このことについては後述する。

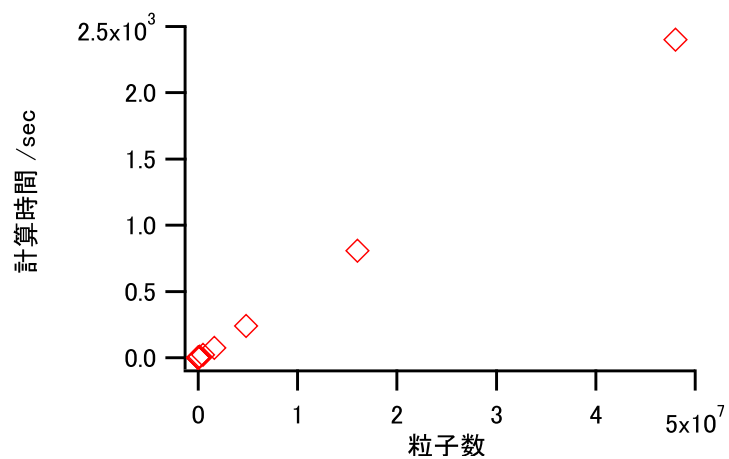
### 4.8.2 実効性能と理論ピーク性能

続いて本研究で作成したプログラムがどのような速度で動作しているかを考える。ここでは文献 [17] と同様の手法を用いるものとする。足し合わせる対象である指数関数を計算する上で必要な計算は積算が 2 回、和算が 3 回、三角関数演算が 1 回であり、粒子と  $q$  空間座標のペアあたり 6 回の浮動小数演算が必要になる。この計算を粒子数、 $q$  空間座標数の積だけ行うので、浮動小数演算の回数は  $7.2 \times 10^{11}$  回となる。この計算にかかる時間は 22.4 秒なので、実行性能は 0.032 TFLOPS となる。これは理論ピーク性能の 3.2 % にあたり、理想的な状態と比較して十分な性能が出ていないことがわかる。その原因と考えられるものを以下に述べる。

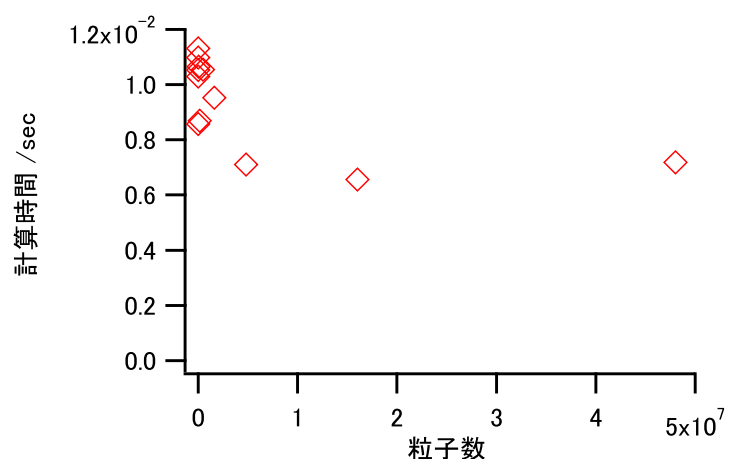
まず、並列化のオーバーヘッドが大きいことが挙げられる。並列化を行うと、各コアを制御するための処理など本来必要でなかった処理が入ってしまい、コアの数だけ高速化がされる訳ではない。

次に、FMA が過剰評価されていることも考えられる。FMA により処理速度が 2 倍になるのは浮動小数の積と和の演算を同時に行い続けるようなプログラムであり、本プログラムはそうには動作しない。そのため FMA の寄与は 2 ではなく 1 に近いと考えられ、理論ピーク性と実効性能が乖離する一つの要因であると言える。

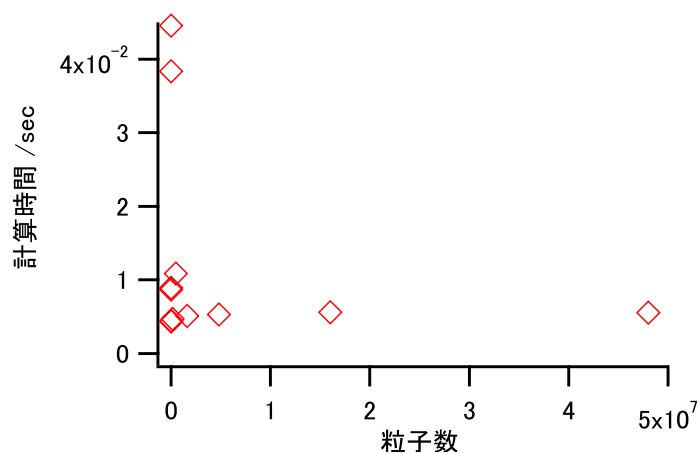
演算量が少なく見積もられている可能性も考えられる。三角関数演算は 1 サイクルで行われると仮定し



(a) 散乱像計算時間

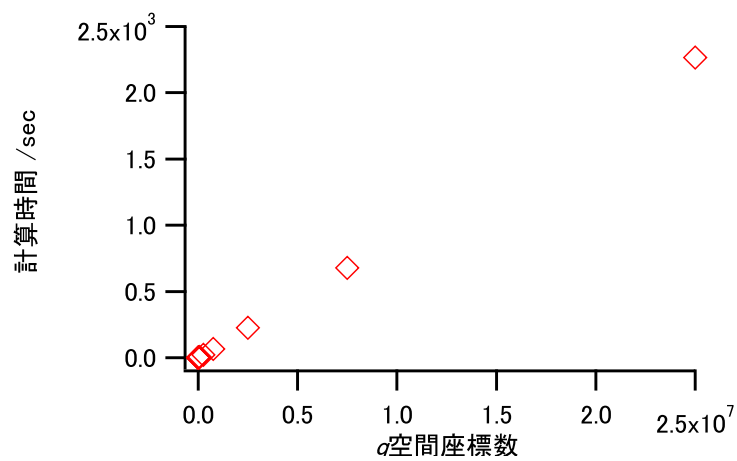


(b) データ転送時間

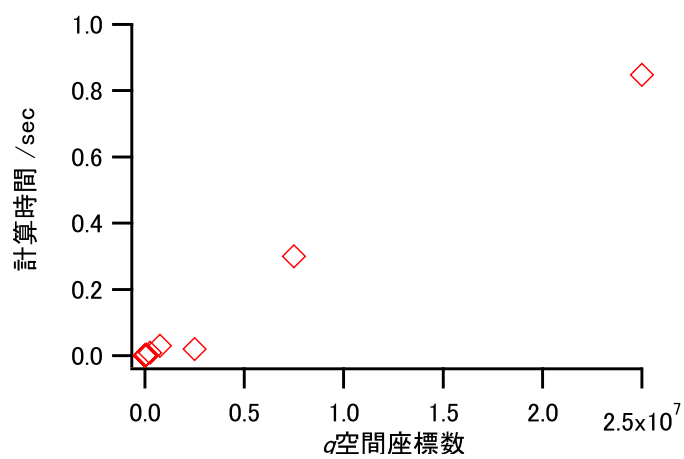


(c) データ保存時間

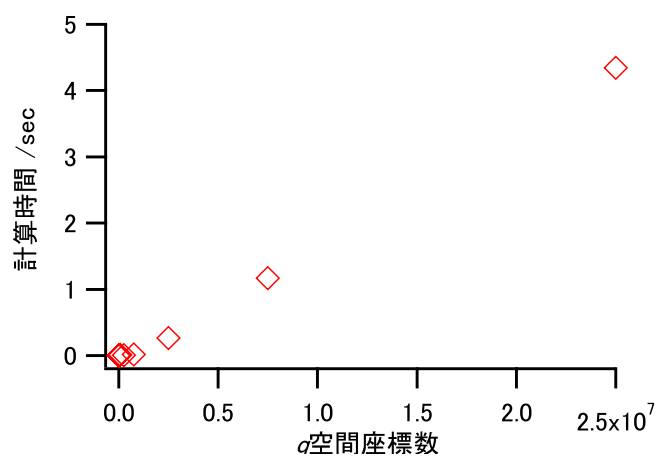
Fig. 4.10: 粒子数を変更したときの実行時間変化。 $q$  空間座標数は  $2.5 \times 10^5$  とした。



(a) 散乱像計算時間



(b) データ転送時間



(c) データ保存時間

Fig. 4.11:  $q$  空間座標数を変更したときの実行時間変化。粒子数は  $4.8 \times 10^5$  とした。



見積もりを行った。ところが三角関数演算は一般には4～8サイクルを要する [17] ため、三角関数演算に要する時間が大きいほど実際の実効性能は高くなるものと考えられる。仮に三角関数演算に6サイクル必要であると仮定すると、実効性能は0.059 TFLOPS となる。

#### 4.8.3 計算可能な条件

計算例として特定のサンプルに対する計算結果を示したが、計算対象とする試料の条件などによって計算量は大幅に変化する。例として、サンプルの体積分率を保ったまま粒子径を1/10にすると粒子数は1000倍になり、計算時間も1000倍になる。理想的には計算対象の系のスケールによらず計算可能であるべきだが、実際にはプログラム上で計算可能な範囲には上限が存在する。

上限の候補の一つにコプロセッサのメモリがある。Xeon Phi のメモリは8GB あるが、その他のストレージを持たないため、計算に用いるデータがこのメモリ容量を超えると計算不能となる。粒子位置の情報は1個あたり24Bで、 $3.3 \times 10^8$  ほどの粒子の情報を格納することができる。これを超えるとメモリ不足となり計算不能となる。実際に  $N = 3.3 \times 10^8$  で計算すると実行時にエラーが生じる。この問題はキャッシュブロッキングによく似た手法を用いることで解決する [12]。メモリに入る規模まで粒子をブロックに分割し、結果は最後にホスト上で足し合わせるのである。

他の上限の候補として、プログラム上のデータ型による制約がある。整数を表す int 型の表現可能な範囲は  $-2147483648 \sim 2147483647$  であり、粒子の番号や座標の番号として int 型を用いている場合、粒子数や座標数がこの範囲を超える場合は正常に計算することができなくなる。これを回避するためには、より広い範囲の整数を表現可能な long 型を用いる。

### 4.9 ハードウェア性能の向上による高速化の検討

本研究では計算機として並列計算機 XeonPhi を用いた。将来的にこのハードウェアでは性能不足になる場合、より性能の高いハードウェアを用いる必要がある。本節ではハードウェアを変更した場合にどのような性能向上が得られるかについて検討する。

直感的には、処理速度が  $x$  倍になったとき処理時間は  $1/x$  となるように思われる。しかし一般にこれは成立しない。なぜなら処理速度を向上させる時、プログラムの全ての計算が高速化する訳ではないからである。並列化による高速化を図る場合、プログラム中で並列処理できない部分は高速化されず、この処理時間が全体の高速化に制限を与える。

あるプログラムを並列化により高速化することを考える。プログラムの処理時間のうちで並列化可能な部分を  $r_p$ 、並列化不可能な部分を  $r_s$  とし、 $r_p + r_s$  が成り立つとする。このとき、並列化可能な部分の速度が  $n$  倍になったとすると、全体の速度向上  $S$  は Amdahl の法則 [18] に従い、次のようになる。

$$S = \frac{1}{r_s + \frac{r_p}{n}} \quad (4.14)$$

以下で本研究のプログラムについて処理速度と計算時間の関係を検証する。本プログラムでデータ転送およびデータ保存の処理は並列化できない処理であり、Fig. 4.11 の  $N_q = 2.5 \times 10^5$  の結果を用いると、計算例で示した条件で計算を行う場合は並列化不可能な処理が全体に占める割合は  $2 \times 10^{-4}$  程度である

ことがわかる。仮に理論性能 10 PFLOPS のスーパーコンピュータ京で計算する場合、処理速度の向上は  $1.0 \times 10^4$  となり、Amdahl の法則から  $3 \times 10^3$  程度の高速化が見込める。

Amdahl の法則はプログラムの高速化に対して 1 つの指針を与える。それは、プログラムの処理の中で時間がかかっている処理を高速化するのが効率的であるということである。この点が本研究で散乱像計算について重点的に高速化を図っている理由である。

## 第 5 章

# まとめと今後の展望

### まとめ

XPCS で用いられる、コヒーレント光によるスペックル像を並列計算機 XeonPhi を用いて高速に計算を行う手法を確立し、計算を行った。試料は球がランダムに試料中に分散した系を考えた。その結果、計算時間の大部分は散乱像計算により占められ、その時間は粒子数と  $q$  空間座標の数に比例することがわかった。

プログラムの高速化手法としては XeonPhi の性能を生かすため、並列化とベクトル化による高速化を行った。これにより実効性能 32 GFLOPS の性能が得られた。他の高速化手法であるキャッシュブロッキングについて検討したが、本プログラムを XeonPhi で実行する際は高速化に寄与しないことがわかった。精度を落とすことによる高速化についても検討したが、単精度浮動小数を用いる場合の精度の悪化は粒子数  $N = 4.8 \times 10^5$  のとき  $9.9 \times 10^{-5}$  の相対誤差が生じることが判明し、粒子数が増加すると相対誤差も増加するため、採用には注意を要する手法であることがわかった。

また、作成したプログラムの適用限界についても検討した。XeonPhi のメモリサイズに起因する適用限界があることがわかり、より大規模な系に本プログラムを適用するためには問題の分割によるメモリサイズ限界の回避が必要なことがわかった。

### 今後の展望

本研究では並列化による高速化を念頭に置いてプログラムを作成した。そのため同様に並列計算により高速化を行っているスーパーコンピュータにも適用可能であると考えられ、その検証が必要である。

また、本研究では粒子運動について深く触れなかった。XPCS の研究において仮定されてきた粒子運動は Brown 運動などの非常に簡単な系に限定されていたが、特定のポテンシャルに従うような複雑な粒子運動について計算することができれば、作成したプログラムを用いて散乱像を作成し、その結果を元に解析を行うことも可能になる。そのような粒子運動計算を取り込み、XPCS の解析に新たな可能性を与えることが今後の課題と言える。

## 謝辞

本研究は東京大学 新領域創成科学研究科 物質系専攻 雨宮研究室において行われたものです。

修士論文執筆にあたり、多くの方にお世話になりました。ここにお礼を申し上げたいと思います。

雨宮慶幸教授には研究方針から日常生活における様々な心構え等についても幅広いご指導をいただきました。研究室生活の間、教授は自分にとっての精神的な支えでした。悩んでいた時に声をかけていただいたことは一生忘れることはないと思います。

篠原佑也助教には研究内容について非常に詳しく相談に乗っていただきました。また研究室の環境整備を一手に担う姿に非常に多くのことを学びました。

秘書の榊原千晶さんには研究室生活の面でお世話になりました。辛かった時期の気遣いが非常に有り難く感じられました。

社会人博士の岸本浩通氏には博士の発表などを通じて社会人としてのあるべき姿を学びました。

社会人博士の松井和也氏には直向きに研究に打ち込む姿に多くのことを学びました。

博士1年の井上伊知郎氏には研究者としてのあるべき姿を学びました。研究に対して熱心な姿は自分もこうあるべきと強く感じさせるものでした。

修士2年の渡部慧氏には特に XPCS 関連で基礎的なことを教わりました。また井上氏が SPring-8 に常駐している中、学生のリーダーとしての姿勢に学ぶところも多かったように思います。

修士2年の名越健誠氏には研究についての相談の他、研究以外でもいろいろなことで話し相手になってもらいました。研究についてもっと相談に乗ればとよかったという点が心残りです。

修士1年の吉井輝明氏には研究内容に似た部分もあり特に研究について相談に乗ってもらいました。研究についてのディスカッションは自分の励みにもなりました。

修士1年の山本奈央子さんには研究室生活においていろいろな点を気遣ってもらいました。今後も健康に気をつかうことを肝に銘じていきたいと思います。

学部4年の安藤卓真氏、松木康裕氏には個人的な相談に乗ってもらいました。二人の元気さが自分にとっての励みにもなりました。

東北大学 多元物質科学研究所 百生研究室の百生敦教授には研究者としての振る舞いを学びました。実験、そして結果の考察に真剣に向き合う姿は今後の人生においても自分の目標であり続けると思います。

東北大学 多元物質科学研究所 百生研究室の矢代航准教授には研究に関して様々な面でお世話になりました。また私生活面にも気を遣っていただきました。

研究室に在籍中に卒業された雨宮研究室の八島恵子さん、清家はるかさん、百生研究室の Sebastien Harasse 氏、Margie P. Olbinado さん、深澤拓也氏、大田悠平氏、木林駿介氏、共に研究生活を歩んでい

けたことに感謝致します。お世話になりました。

最後になりますが、本論文は、研究生生活を支えて下さった皆様のご協力により書くことができました。お世話になった方々に深く感謝致します。ありがとうございました。

## 参考文献

- [1] M. Sutton, S. G. J. Mochrie, T. Greytak, S. E. Nagler, L. E. Berman, G. A. Held and G. B. Stephenson, *Nature*, **352**(6336), 608–610 (1991).
- [2] R. J. Roe, *Methods of X-Ray and Neutron Scattering in Polymer Science*, Oxford University Press (2000).
- [3] J. Als-Nielsen and D. McMorrow, *Elements of modern X-ray physics 2nd Edition*, Wiley (2011).
- [4] 篠原佑也, 放射光, 第 26 巻 第 2 号, 114–117 (2013).
- [5] Y. Shinohara, H. Kishimoto, T. Maejima, H. Nishikawa, N. Yagi and Y. Amemiya, *Soft Matter*, **8**(12), 3457–3462 (2012).
- [6] 早川禮之助, 伊藤耕三, 木村康之, 岡野光治, 非平衡系のダイナミクス動的物性の物理, 培風館 (2006).
- [7] L. Cipelletti, L. Ramos, S. Manley, E. Pitard, D. A. Weitz, E. E. Pashkovski and M. Johansson, *Faraday Discussions*, **123**, 237–251 (2003).
- [8] 小柳義夫, 中村宏, 佐藤三久, 松岡聡, 計算科学別巻 スーパーコンピュータ, 岩波書店 (2012).
- [9] J. Fung and S. Mann, Computer vision signal processing on graphics processing units, In *Proceedings of the Pattern Recognition 17th International Conference on (ICPR'04) Volume 1*, pp. 805–808 (2004).
- [10] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, *ACM Transactions on Graphics*, **27**(3), 18:1–18:15 (2008).
- [11] M. D. Lam, E. E. Rothberg and M. E. Wolf, *ACM Sigplan Notices*, **26**(4), 63–74 (1991).
- [12] A. Sarje, X. S. Li, S. Chourou, E. R. Chan and A. Hexemer, Massively parallel x-ray scattering simulations, In *SC'12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press (2012).
- [13] G. Renaud, R. Lazzari and F. Leroy, *Surface Science Reports*, **64**(8), 255–380 (2009).
- [14] Th. Zemb and P. Lindner (editors), *Neutron, X-rays and Light. Scattering Methods Applied to Soft Condensed Matter (North-Holland Delta Series)*, Elsevier (2002).
- [15] N. J. Higham, *SIAM Journal on Scientific Computing*, **14**(4), 783–799 (1993).
- [16] M. Corden, *Consistency of Floating Point Results or Why doesn't my application always give the same answer?* Intel Corporation Software Solutions Group (2008).
- [17] V. Favre-Nicolin, J. Coraux, M. I. Richard and H. Renevier, *Journal of Applied Crystallography*,

- 44**(3), 635–640 (2011).
- [18] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485 (1967).