相互作用計算カーネルジェネレータPIKGの開発とN体およびSPHカーネルの富岳向け最適化

野村 昂太郎 / Kentaro Nomura
神戸大学理学研究科惑星科学研究センター

目次

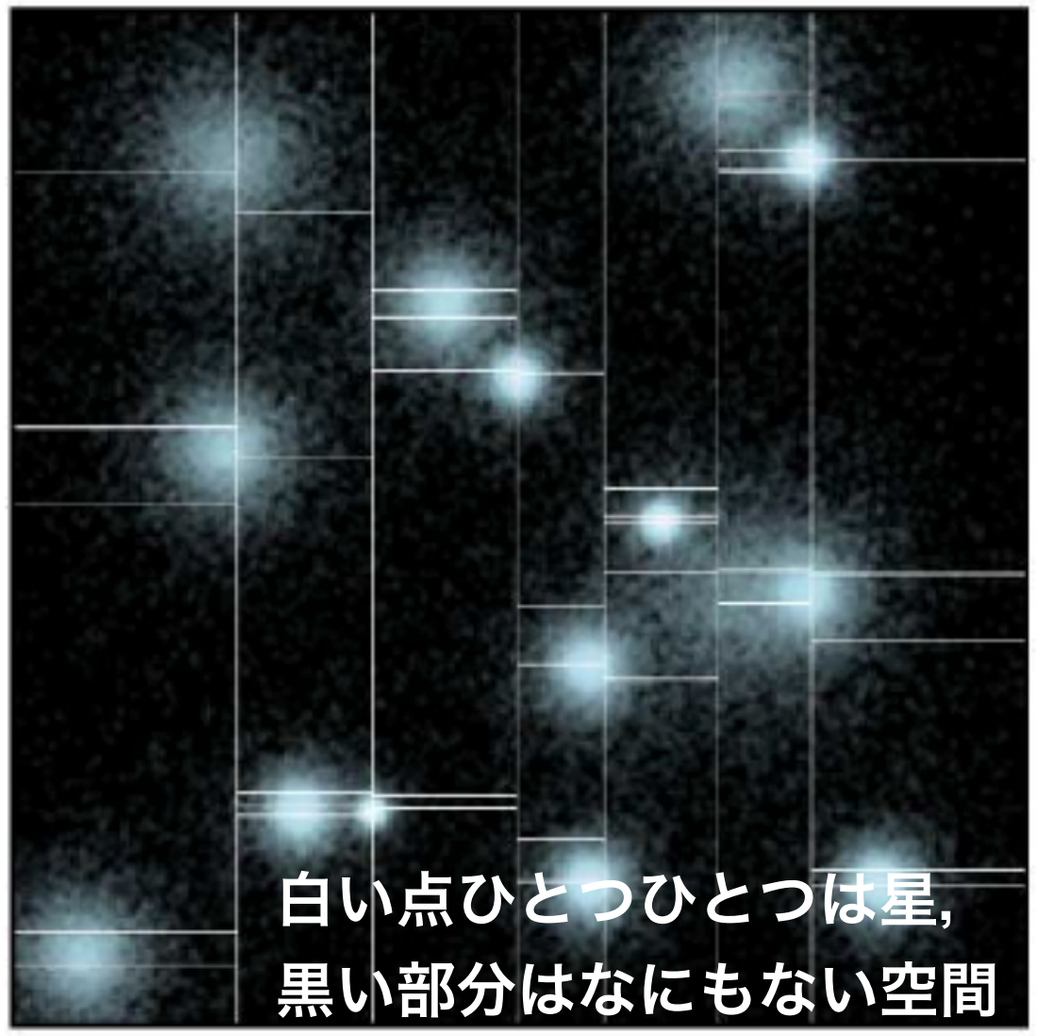
- 背景
- PIKGについて
- 富岳向けの最適化ついて
- まとめ/富岳の感想など

背景

- 分子動力学やN体，SPHなどの粒子系シミュレーションコードの性能は(多くの場合)粒子間相互作用計算の実装(最適化)に大きく依存
- ユーザはそれぞれの計算機向けに最適化された相互作用カーネルを書かないといけなく，できあがるものは(大抵の場合)再利用できない
- 単一のコードでさまざまな計算機上で十分な性能を引き出したい！

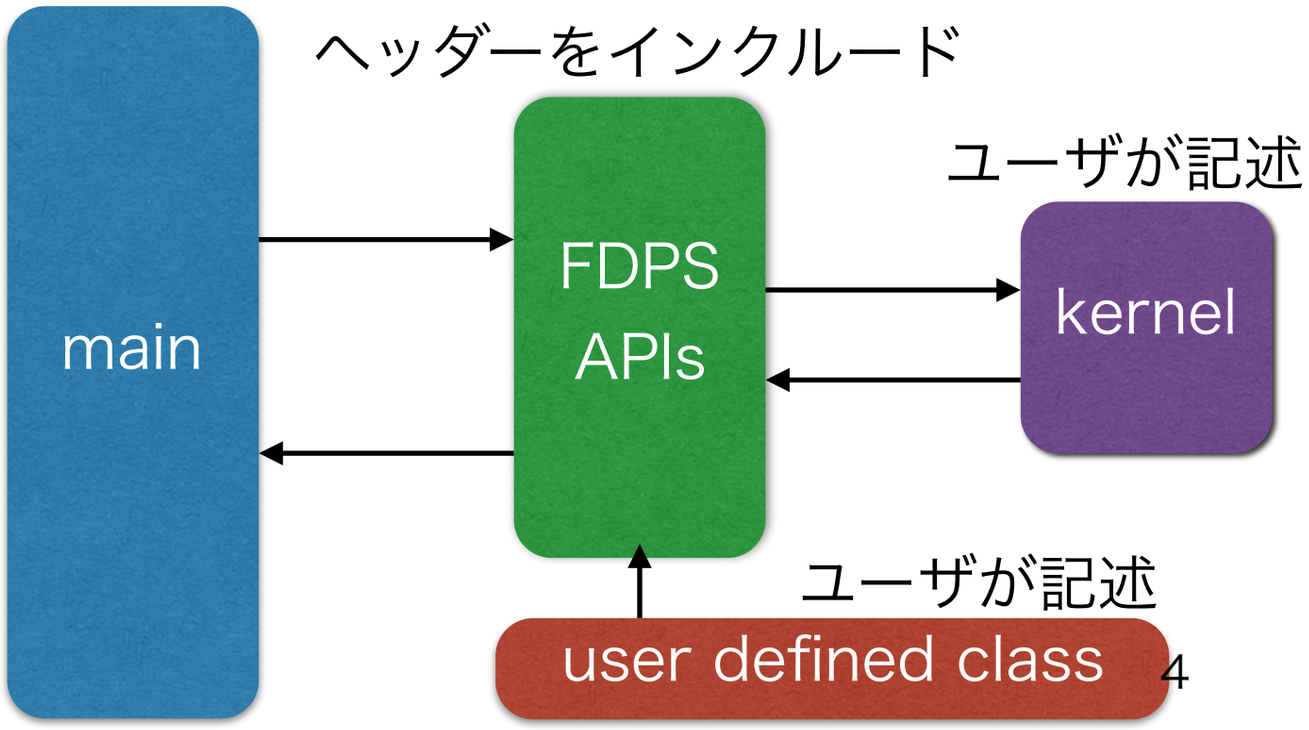
FDPS (Framework for Developing Particle Simulators)

- 粒子同士が相互作用しながら時間発展するシミュレーションの開発支援フレームワーク
- FDPSはノード間通信・系の分割・相互作用リスト作成など面倒なことを引き受けてくれる
- ユーザはユーザー定義クラス, 時間発展(main関数)と相互作用カーネルを記述
- 粗密にあわせて動的に領域を分割(右図)
- ツリー構造を使った相互作用リストの作成
- Fortran/Cインターフェース
- アクセラレータの利用



Iwasawa+, PASJ(2016)

ユーザが記述



FDPSの目指すところ

- ユーザは粒子の情報と時間発展と粒子間相互作用を記述するだけ
- 用意されたAPIを使いながらシリアルコードに近いものを書くと勝手に並列化される
- ユーザは(ほとんど)並列化については考えない
- MPI/OpenMPの利用はコンパイルオプションだけで変更
- アクセラレータのあるヘテロジニアスな構成でも利用可能
- さまざまな言語から利用可能

FDPSの計算の流れ

1．領域分割

ロードバランスを考慮しながら計算領域をサブ領域に分割して各MPIプロセスに粒子を割り当て

2．粒子情報の交換

粒子の移動などにあわせてそれぞれのプロセスが持つ粒子の情報を交換

3．相互作用情報の交換

相互作用に必要な情報を隣接するプロセスから取得

並列化
ノード間通信

APIを提供

4．相互作用の計算

各MPIプロセス内で相互作用を計算

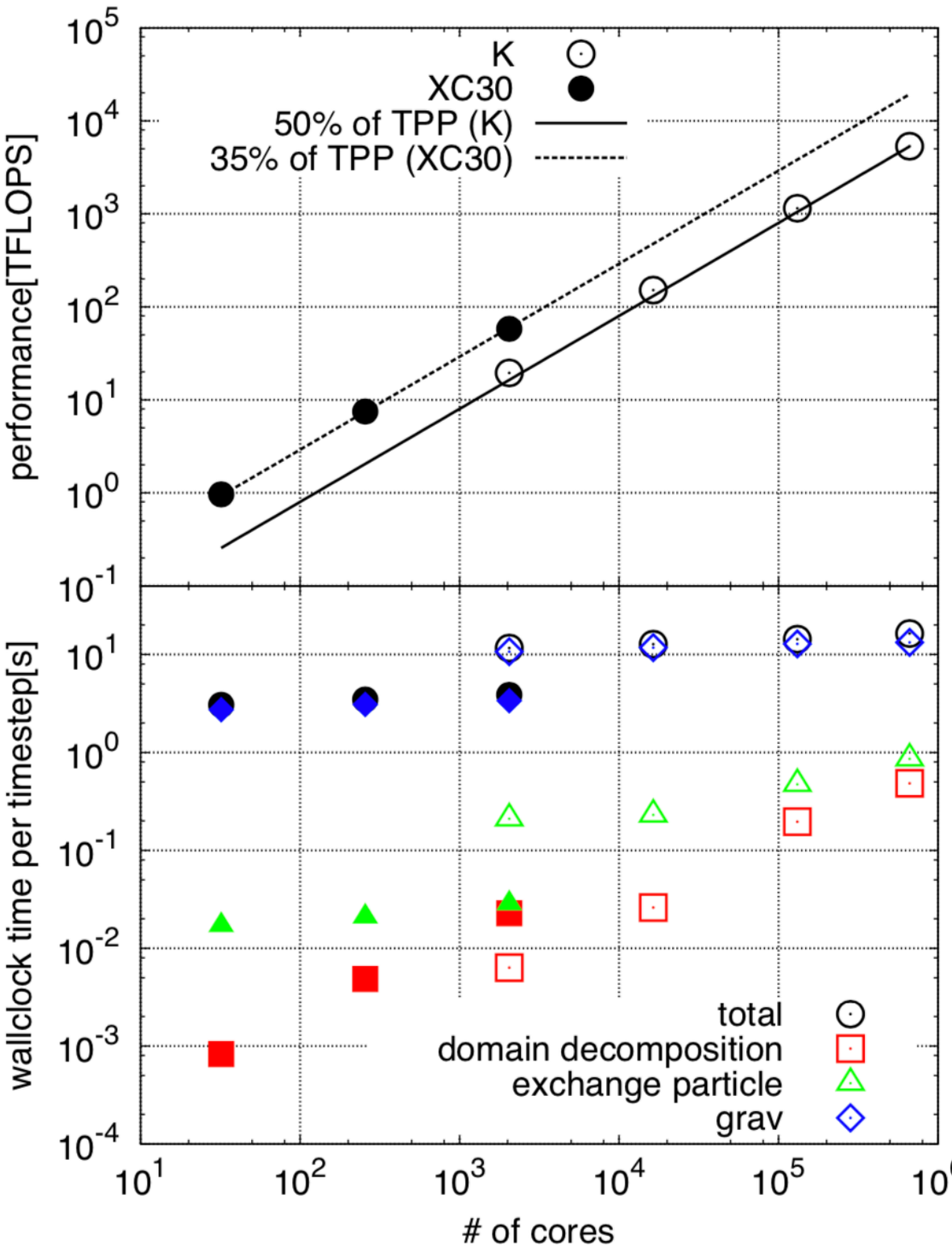
相互作用リストを提供し
計算を高速に実行

5．粒子の時間発展

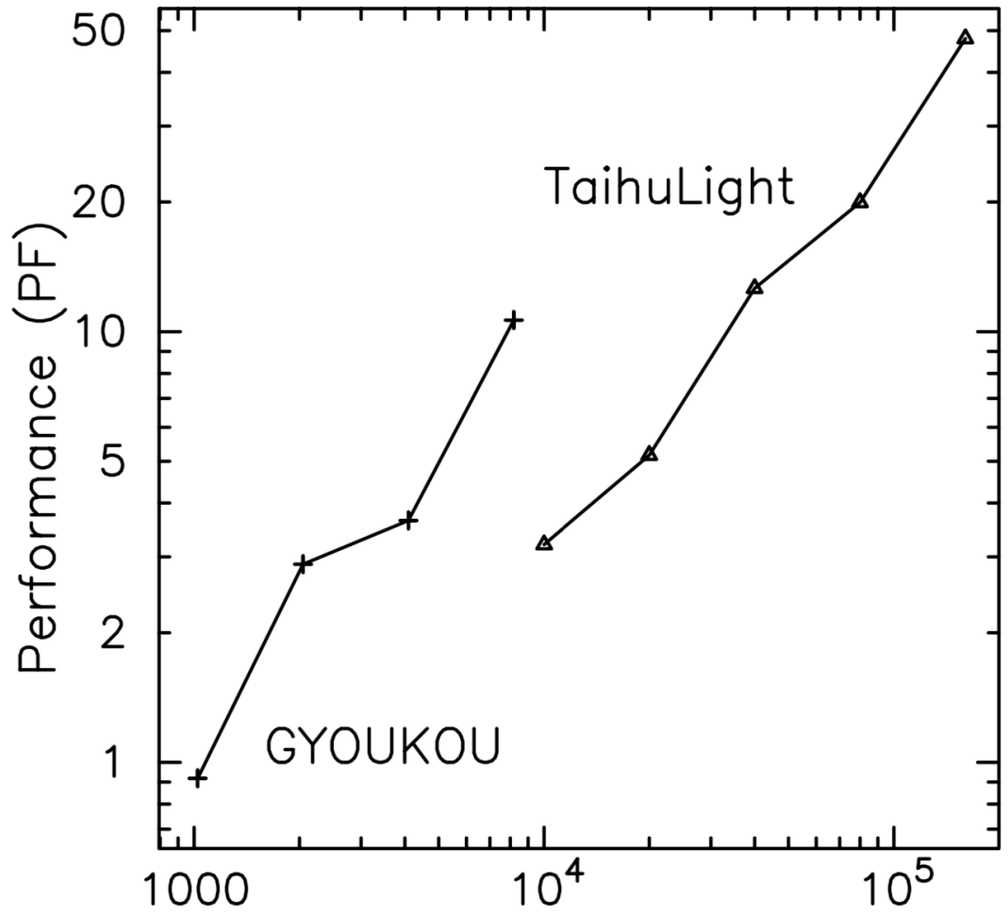
それぞれのプロセスが担当するの相互作用を計算

ユーザーが自由に記述

並列化性能



Iwasawa+, PASJ(2016)



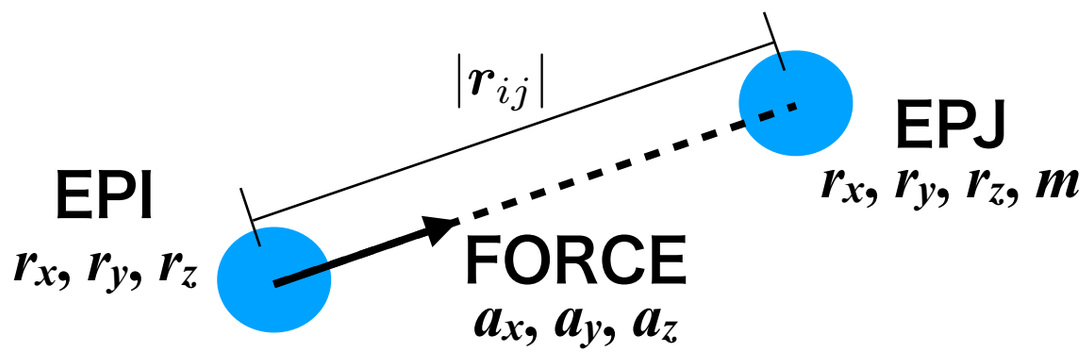
- 様々なスパコンで高い並列化効率
- 相互作用カーネルの速度が性能に大きく影響
- これまでは似鳥さんらエキスパートの手によって素晴らしいカーネルが作られてきた
- カーネルひとつくらいなら頑張ってもいいが…

PIKG

- (Paricle-particle Interaction Kernel Generatorの略)
- 相互作用計算をDSLで簡単に記述
- ジェネレータにDSLコードとオプション(アーキテクチャの指定とか)を与えると最適化された粒子間相互作用カーネルコードがでてくる
- ユーザーは(ほとんど)最適化については考えない
- 単一コードから複数アーキテクチャの最適化コードを生成

まずは使い方から

$$\boldsymbol{a}_i = \frac{m_j}{|\boldsymbol{r}_{ij}|^3}\boldsymbol{r}_{ij}$$



- ユーザは相互作用を及ぼされる粒子(EPI)と及ぼす粒子(EPJ)とEPIにかかる相互作用(FORCE)の変数を定義
- 定義した関数などを用いて2粒子間相互作用の計算を記述
- オプション等を設定してジェネレータにかけるとカーネルができる

例(N体モノポール)

$$\boldsymbol{a}_i = \frac{m_j}{(e_i + e_j + |\boldsymbol{r}_{ij}|)^3}\boldsymbol{r}_{ij}$$

$$\frac{u_i}{m_i} = \frac{m_j}{e_i + e_j + |\boldsymbol{r}_{ij}|}$$

Variable definition:

[*class_type*] *type* var*name* [: *member_name*]

Function definition:

function *name*(*variables*...)
  [*statements*...]
  return *val*
end

Kernel definition:

variable (=|+=|-=) *expression*

```
EPI F32vec xi:pos
EPI F32    epsi:eps
EPJ F32vec xj:pos
EPJ F32    mj:mass
EPJ F32    epsj:eps
FORCE F32vec f:acc
FORCE F32 phi:pot
```

```
function sub(a,b)
  return a-b
end
```

```
rij = xi - xj
r2 = epsi*epsi + rij*rij
rinv = rsqrt(r2)
mrinv = mj*rinv
f -= mrinv*rinv*rinv * rij
phi -= mrinv
```

生成コード(non-SIMD)

```
template<typename Tepi,typename Tepj,typename Tforce>
struct Kernel{
  Kernel(){}
  void operator()(const Tepi* epi,const int ni,const Tepj* epj,const int nj,Tforce *force){
    for(int i=0;i<ni;i++){
      PS::F32vec xi = epi[i].pos;
      PS::F32 epsi = epi[i].eps;
      PS::F32vec f = force[i].acc;
      PS::F32 phi = force[i].pot;
      for(int j=0;j<nj;j++){
        PS::F32vec xj = epj[j].pos;
        PS::F32 mj = epj[j].mass;
        PS::F32 epsj = epj[j].eps;
        PS::F32vec rij;
        rij.x = (xi.x-xj.x);
        rij.y = (xi.y-xj.y);
        rij.z = (xi.z-xj.z);
        PS::F32 r2 = madd<PS::F32,PS::F32,PS::F32,PS::F32>(epsi,epsi,madd<PS::F32,PS::F32,PS::
F32,PS::F32>(rij.x,rij.x,madd<PS::F32,PS::F32,PS::F32,PS::F32>(rij.y,rij.y,(rij.z*rij.z))));
        PS::F32 rinv = rsqrt<PS::F32,PS::F32>(r2);
        PS::F32 mrinv = (mj*rinv);
        PS::F32 __fkg_tmp0 = ((mrinv*rinv)*rinv);
        f.x = nmsub<PS::F32,PS::F32,PS::F32,PS::F32>(__fkg_tmp0,rij.x,f.x);
        f.y = nmsub<PS::F32,PS::F32,PS::F32,PS::F32>(__fkg_tmp0,rij.y,f.y);
        f.z = nmsub<PS::F32,PS::F32,PS::F32,PS::F32>(__fkg_tmp0,rij.z,f.z);
        phi = (phi-mrinv);
      }
      force[i].acc = f;
      force[i].pot = phi;
    }
  }
  template<typename Tret,typename Ta,typename Tb>
  Tret sub(Ta a,Tb b){
    return (a-b);
  }
  template<typename Tret,typename Top>
  Tret rsqrt(Top op){ return (Tret)1.0/std::sqrt(op); }
  template<typename Tret,typename Top>
  Tret sqrt(Top op){ return std::sqrt(op); }
  template<typename Tret,typename Top>
  Tret inv(Top op){ return 1.0/op; }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret madd(Ta a,Tb b,Tc c){ return a*b+c; }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret msub(Ta a,Tb b,Tc c){ return a*b-c; }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret nmadd(Ta a,Tb b,Tc c){ return -(a*b+c); }
  template<typename Tret,typename Ta,typename Tb,typename Tc>
  Tret nmsub(Ta a,Tb b,Tc c){ return c-a*b; }
};
```

PIKGの中身に入る前に…

- Single Instruction Multiple Data(SIMD)
  - 複数のデータに対して同じ命令を実行
  - 近年プログラムのシングルコア性能をピークに近づけるためにはコードはSIMD化され(るように書か)なくてはならない
  - ロード/ストアもまとめて行うのでデータ形状が大事
- Array of Structure(AoS)とStructure of Array(SoA)
  - 例えば各粒子が(x,y,z,w)の4要素を持っているとき
  - (AoS)$x_0y_0z_0w_0x_1y_1z_1w_1\cdots$ (SoA)$x_0x_1x_2x_3y_0y_1y_2y_3\cdots$
  - 同じ命令を各データに当てはめるのでSIMD演算はSoA構造がよい(場合がほとんど)

どうやって最適化コードを生成するか

- 粒子間相互作用計算の一般化
- DSLの設計
- DSLの読込(一般的なものを使っているので今回は割愛)
- オプションから適切なループ構造を選択
- アーキテクチャ(命令セット)に合わせてカーネルを生成

2粒子間相互作用計算の一般化

```
void kernel_body(const EPI* epi,const int ni,
                 const EPJ* epj,const int nj,
                 FORCE* force){
  (1) preprocess
  for (int i=0; i < ni; i++){
    (2) load EPI variables
    (3) load FORCE variables
    for(int j=0; j < nj; j++){
      (4) load EPJ variables
      (5) calc interaction for FORCE variable
    }
    (6) store FORCE variables
  }
}
```

1. 前処理が必要であればここで行う
2. EPI変数及び一時変数でカーネル計算で使われる変数をロード
3. 相互作用のアキュムレートに必要なFORCE変数のロード
4. EPJ変数のロード．(2)と同様
5. 相互作用の計算を行い，FORCE変数にアキュムレート
6. アキュムレートしたFORCE変数をもとの配列にストア

一般的なSIMD化作業

```
void kernel_body(const EPI* epi,const int ni,
                 const EPJ* epj,const int nj,
                 FORCE* force){
  (1) preprocess
  for (int i=0; i < ni; i++){
    (2) load EPI variables
    (3) load FORCE variables
    for(int j=0; j < nj; j++){
      (4) load EPJ variables
      (5) calc interaction for FORCE variable
    }
    (6) store FORCE variables
  }
}
```

1. i/jループ形状をSIMD幅に合わせて変更
2. ロードでSoAからAoSにする
3. SIMD化されたカーネル計算を書く
4. ストアでAoSからSoAにする

DSL設計の要件

- SoAを入力をもらったら相互作用を計算してSoAで出力する
  - 入出力の構造体の情報が必要
- 2重ループをユーザに書かせない
  - 最適化で大事な要因で，アーキテクチャごとに変わるループ形状はPIKG側で決める
- 計算中の精度の管理はユーザに任せる
  - といっても入出力のデータの型を指定するだけ
  - 混合精度にも対応(まだできてない)
  - 入出力以外の一時変数は型推論される
- 最適化で重要となる関数(sqrt/rsqrt/invとか)は予約語にしてPIKGが最適なものを選んで使う

実際のDSL(再掲)

$$\boldsymbol{a}_i = \frac{m_j}{(e_i + e_j + |\boldsymbol{r}_{ij}|)^3}\boldsymbol{r}_{ij}$$

$$\frac{u_i}{m_i} = \frac{m_j}{\epsilon_i + \epsilon_j + |\boldsymbol{r}_{ij}|}$$

Variable definition:

[*class_type*] *type* var*name* [: *member_name*]

Function definition:

function *name*(*variables*…)
  [*statements*…]
  return *val*
end

Kernel definition:

variable (=|+=|-=) *expression*

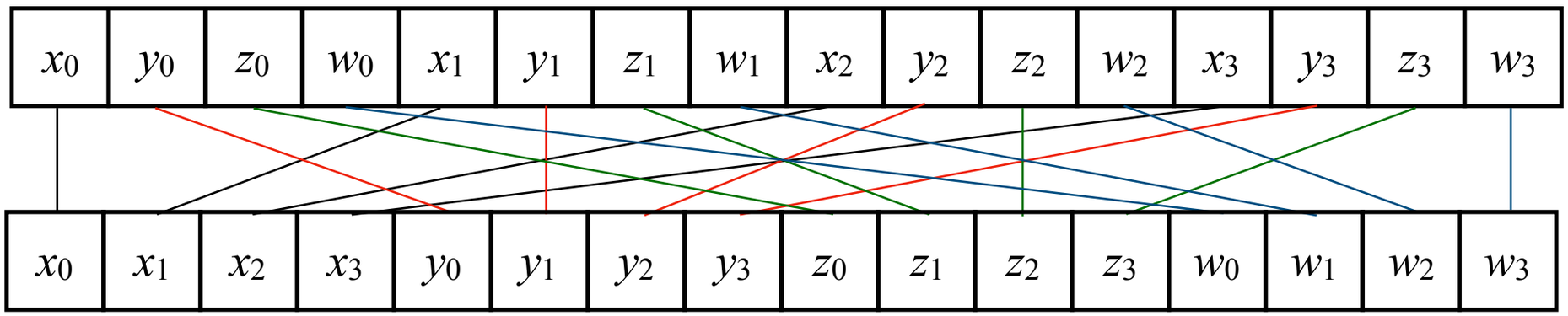
```
EPI F32vec xi:pos
EPI F32    epsi:eps
EPJ F32vec xj:pos
EPJ F32    mj:mass
EPJ F32    epsj:eps
FORCE F32vec f:acc
FORCE F32 phi:pot
```

```
function sub(a,b)
  return a-b
end
```

```
rij = xi - xj
r2 = epsi*epsi + rij*rij
rinv = rsqrt(r2)
mrinv = mj*rinv
f -= mrinv*rinv*rinv * rij
phi -= mrinv
```

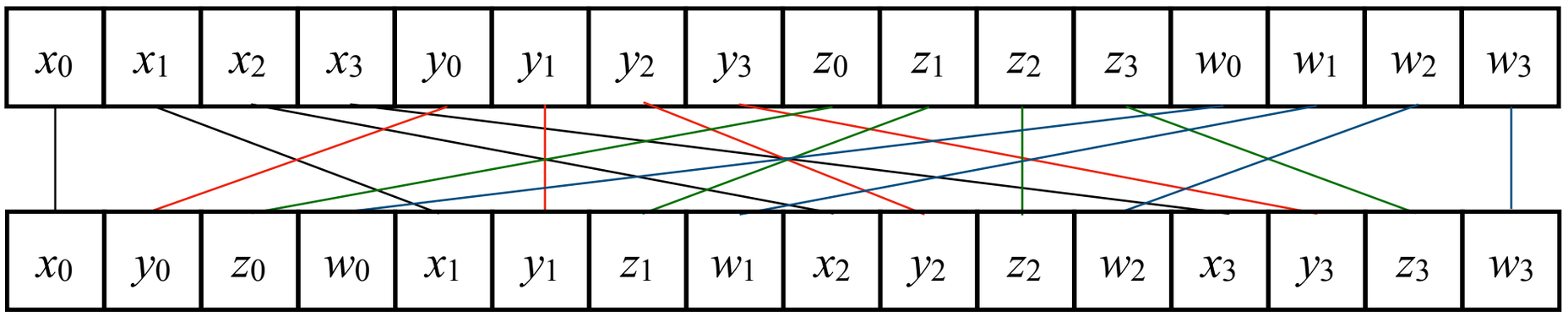
AoS <-->SoA変換

AoS -> SoA



Gather load / Structure load / Load+Transpose

SoA -> AoS



Scatter store / Structure store / Transpose+Store

カーネルを翻訳

- PIKG内ではDSLの式をそれぞれ入れ子構造で保持
  - C = A + B だと(=, C, (+, A B)) のようなイメージ
- 与えられたオプションによってAVX2/AVX-512/ARM SVEなどのイントリンジックスに変換
  - (例)C = A + B
    - reference: C = (A + B);
    - AVX2 :　　C = _mm256_add_ps(A,B);
    - AVX-512: C = _mm512_add_ps(A,B);
    - ARM SVE: C = svadd_f32_z(pg,A,B);

生成コード(最内ループ)

リファレンス

```
for(;j < nj;++j){
PS::F32 ejloc;

ejloc = ejloc_tmp[j];
PS::F32 mjloc;

mjloc = mjloc_tmp[j];
PS::F32vec xjloc;

xjloc.x = xjloc_tmp_x[j];
xjloc.y = xjloc_tmp_y[j];
xjloc.z = xjloc_tmp_z[j];
rij.x = (xiloc.x-xjloc.x);
rij.y = (xiloc.y-xjloc.y);
rij.z = (xiloc.z-xjloc.z);
__fkg_tmp2 = (eiloc+ejloc);
__fkg_tmp1 = (__fkg_tmp2 + rij.x*rij.x);
__fkg_tmp0 = (__fkg_tmp1 + rij.y*rij.y);
r2 = (__fkg_tmp0 + rij.z*rij.z);
rinv = rsqrt(r2);
mrinv = (mjloc*rinv);
__fkg_tmp3 = (mrinv*rinv);
mr3inv = (__fkg_tmp3*rinv);
f.x = (f.x - mr3inv*rij.x);
f.y = (f.y - mr3inv*rij.y);
f.z = (f.z - mr3inv*rij.z);
phi = (phi-mrinv);
}
```

ARM SVE

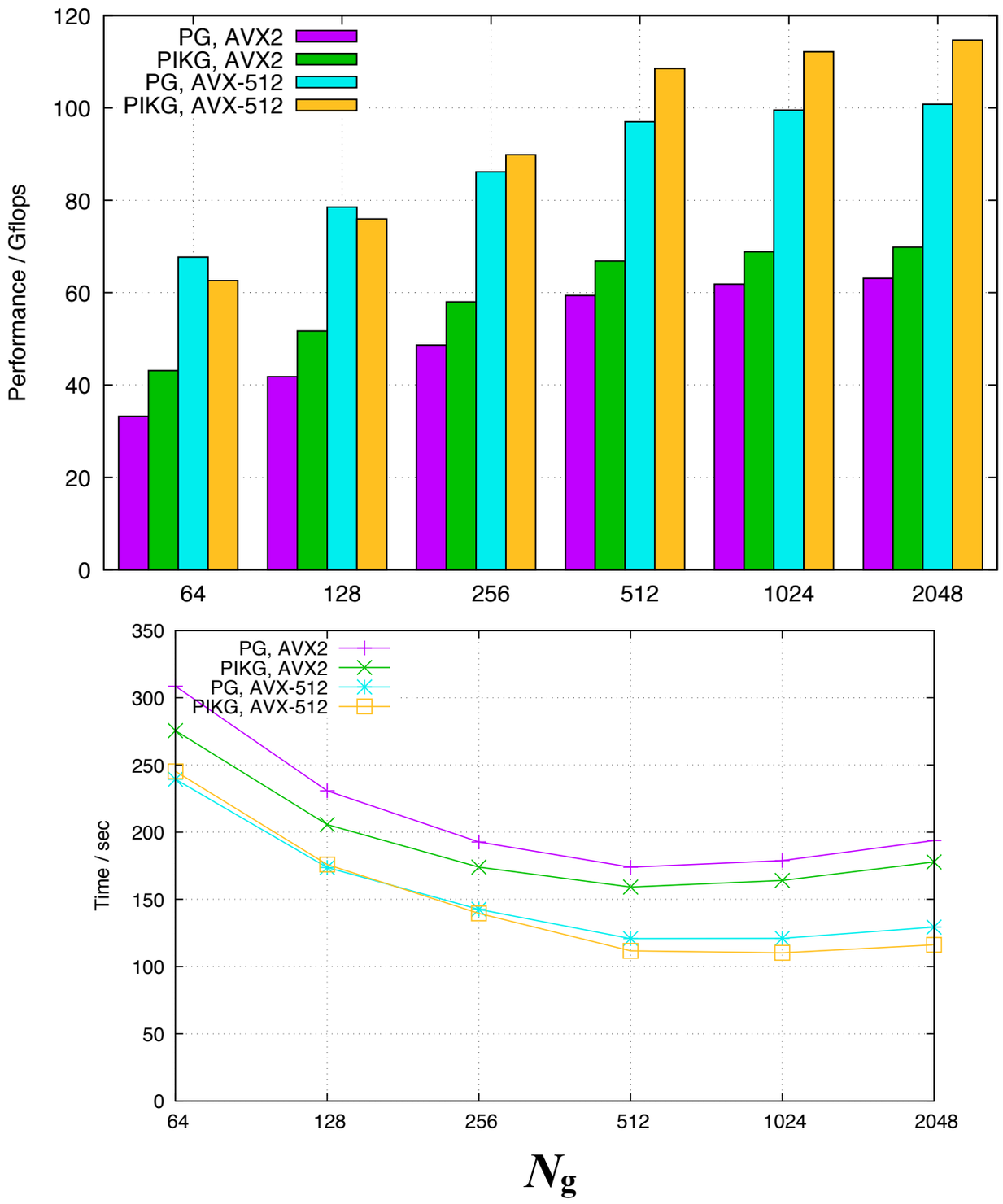
```
for(;j < nj;++j){
svfloat32_t ejloc;

ejloc = svdup_n_f32(ejloc_tmp[j]);
svfloat32_t mjloc;

mjloc = svdup_n_f32(mjloc_tmp[j]);
svfloat32x3_t xjloc;

xjloc.v0 = svdup_n_f32(xjloc_tmp_x[j]);
xjloc.v1 = svdup_n_f32(xjloc_tmp_y[j]);
xjloc.v2 = svdup_n_f32(xjloc_tmp_z[j]);
rij.v0 = svsub_f32_z(pg0,xiloc.v0,xjloc.v0);
rij.v1 = svsub_f32_z(pg0,xiloc.v1,xjloc.v1);
rij.v2 = svsub_f32_z(pg0,xiloc.v2,xjloc.v2);
__fkg_tmp2 = svadd_f32_z(pg0,eiloc,ejloc);
__fkg_tmp1 = svmad_f32_z(pg0,rij.v0,rij.v0,__fkg_tmp2);
__fkg_tmp0 = svmad_f32_z(pg0,rij.v1,rij.v1,__fkg_tmp1);
r2 = svmad_f32_z(pg0,rij.v2,rij.v2,__fkg_tmp0);
rinv = rsqrt(pg0,r2);
mrinv = svmul_f32_z(pg0,mjloc,rinv);
__fkg_tmp3 = svmul_f32_z(pg0,mrinv,rinv);
mr3inv = svmul_f32_z(pg0,__fkg_tmp3,rinv);
f.v0 = svmsb_f32_z(pg0,mr3inv,rij.v0,f.v0);
f.v1 = svmsb_f32_z(pg0,mr3inv,rij.v1,f.v1);
f.v2 = svmsb_f32_z(pg0,mr3inv,rij.v2,f.v2);
phi = svsub_f32_z(pg0,phi,mrinv);
}
```

AVX2/AVX-512でのベンチマーク



- FDPSのN体サンプルでPhantom-GRAPEと比較(https://bitbucket.org/kohji/phantom-grape/src/master/)
- 10万粒子，θ =0.5
- Skylake Xeon 1コア計測
- パフォーマンスはカーネル計算部分のみ，計算時間は相互作用計算全体
- 横軸は何粒子まで同じ相互作用リストを使うかという指標．大きいほど余分な計算が多くなる

富岳概要

https://news.livedoor.com/article/detail/18423524/

総ノード数

総ノード数	158,976ノード 384ノードx 396 ラック= 152,064 192 ノードx 36 ラック= 6,912

総理論性能

総演算性能	通常モード（CPU動作クロック周波数2GHz）	• 倍精度理論最高値（64bit）488ペタフロップス • 単精度理論最高値（32bit）977ペタフロップス • 半精度（AI学習）理論最高値（16bit）1.95エクサフロップス • 整数（AI推論）理論最高値（8bit）3.90 エクサオップス
	ブーストモード（CPU動作クロック周波数2.2GHz）	• 倍精度理論最高値（64bit）537ペタフロップス • 単精度理論最高値（32bit）1.07エクサフロップス • 半精度（AI学習）理論最高値（16bit）2.15エクサフロップス • 整数（AI推論）理論最高値（8bit）4.30 エクサオップス
総メモリ容量		4.85 PiB
総メモリバンド幅		163 PB/s

ノード単体性能

命令セットアーキテクチャ		Armv8.2-A SVE 512bit 富士通拡張:ハードウェアバリア、セクタキャッシュ、プリフェッチ
計算コア数		48 + 2アシスタントコア 4 CMG (Core Memory Group, NUMA nodeのこと)
演算性能	通常モード（CPU動作クロック周波数2GHz）	倍精度: 3.072 TF, 単精度: 6.144 TF, 半精度: 12.288 TF
	ブーストモード（CPU動作クロック周波数2.2GHz）	倍精度: 3.3792 TF, 単精度: 6.7584 TF, 半精度: 13.5168 TF
キャッシュ*1 *2		L1D/core: 64 KiB, 4way, 256 GB/s (load), 128 GB/s (store)
		L2/CMG: 8 MiB, 16way L2/node: 4 TB/s (load), 2 TB/s (store) L2/core: 128 GB/s (load), 64 GB/s (store)
メモリ		HBM2 32 GiB, 1024 GB/s
インターコネクト		Tofu Interconnect D (28 Gbps x 2 lane x 10 port)
I/O		PCIe Gen3 x16
テクノロジー		7nm FinFET

https://www.r-ccs.riken.jp/jp/post-k/overview.html

富岳の最適化で知るべき数字

- A64FX_Microarchitecture_Manual参照
- SIMD演算に使えるレジスタは32本
- SVEの演算レイテンシは基本9
    - コアあたりの演算器は2つなので(雑に考えると)演算ばかりでも18個依存関係のない命令が連続で発行されないと演算レイテンシが見えてその分性能が落ちる
- L1$レイテンシはSVEは11，その他は8．読込は2ライン，書き込みは1ライン
- ほかにもあるが相互作用カーネル作るときはこのくらい

富岳向け最適化

- ループ分割
  - 長い(というわけでもないものも)ループを分割してレジスタ負荷を減らす
- ストリップマイニング
  - L1$にのるようループ回転数を分割
  - 最内ループを定数(n)回転にできる
- ループアンロール
  - 長い命令レイテンシを隠蔽
  - アンロール段数(m)は分割したループごとに最適値がある

元コード

```
for(int i=0;i<N;i++){ A(i); B(i);}
```

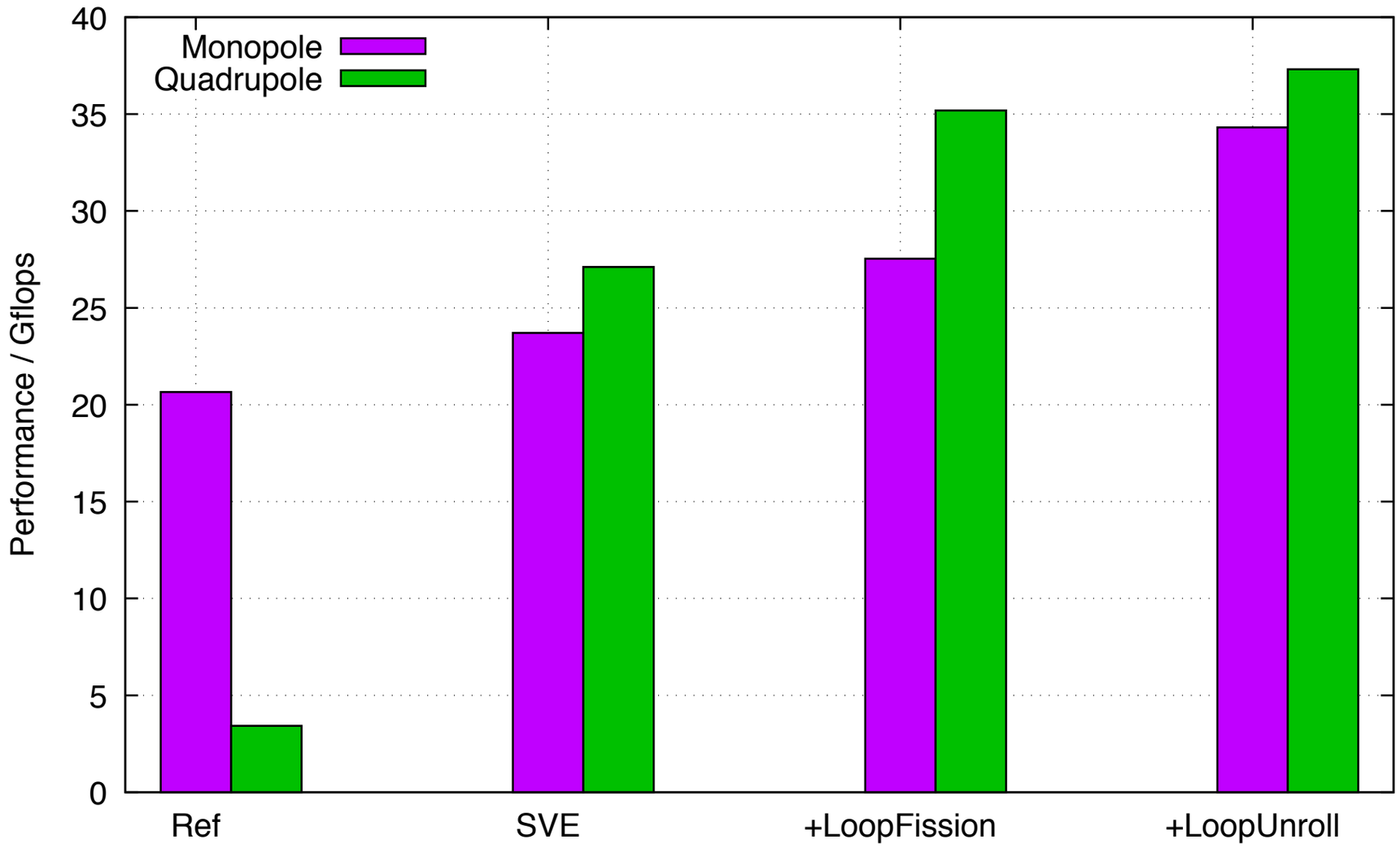
```
for(int i=0;i<N;i++){ A(i);}
for(int i=0;i<N;i++){ B(i);}
```

```
for(int i=0;i<N/n;i++){
  for(int j=0;j<n;j++) A(n*i+j);
  for(int j=0;j<n;j++) B(n*i+j);
}
```

```
for(int i=0;i<N/n;i++){
  for(int j=0;j<n/m;j+=m){
    A(n*i+j+0); … A(n*i+j+m-1);
  }
  for(int j=0;j<n/m;j+=m){
    B(n*i+j+0); … B(n*i+j+m-1);
  }
}
```

最適化の効果

A64FX 2.0 GHz (1コア, 単精度ピーク128Gflops)
コンパイラ: FCC 4.1.0 20200415
オプション: -Kfast -Nclang
条件: 512 x 512のN²ループ



※富岳共用開始後の性能を保証するものではありません

区間多項式近似

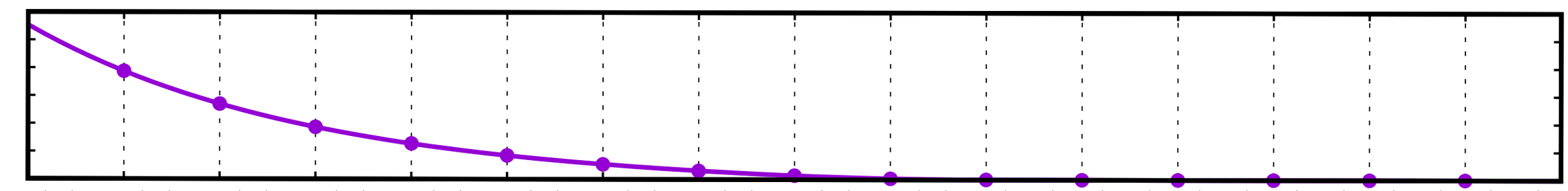
- 近似したい関数の定義域をm個の区間に分割
- それぞれの区間についてn次のminimax多項式で近似
- SVE(AVX-512も)のテーブル参照命令でレジスタをテーブルに使える

  ➡例えば単精度で$m = 16$にすると, n+1個のレジスタを使う

- 近似精度は分割数と定義域の分割の仕方，次数で変わる
- 除算やルートを減らせる / FMAリッチ

$$f^{\mathrm{PPA}}(x;k) = \sum_{l=0}^{n} a_{k,l}\left(x - x_0^{(k)}\right)^l$$

$a_{k,l}$: k番目の領域のl次の係数, $x_0^{(k)}$: k番目の領域の始点



コードで見る区間多項式近似(SPHカーネルを例に)

リファレンスコード

```
PS::F64 dWdh(const PS::F64 r2, const PS::F64 hi) {
  const PS::F63 r2 = sqrt(r2);
  const PS::F64 u = r*hi;
  const PS::F64 p1u = std::max(0.0, 1.0 - u);
  const PS::F64 coeff = (21.0 / 2.0 * math_const::pi) * hi*hi*hi*hi;
  return - coeff * p1u*p1u*p1u * (3.0 + u * (9.0 - 32.0*u));
}
```

シリアル区間多項式近似コード

```
PS::F64 dWdh(cost PS::F64 r2,const PS::F64 h2i){
  PS::F64 u = r2 * h2i;
  PS::F64 du = std::min(1.0,u);
  PS::F64 index_f = 16.0 * du;
  PS::S32 index = (int)index;
  du = du - 0.0625 * (float)index;
  PS::F64 c0 = tab0[index];
  PS::F64 c1 = tab1[index];
  PS::F64 c2 = tab2[index];
  PS::F64 c3 = tab3[index];
  PS::F64 c4 = tab4[index];
  return c0 + du*(c1 + du*(c2 + du*(c3 + du*c4)));
}
```

SIMD化区間多項式近似コード

```
svfloat32_t dWdh(const svfloat32_t r2, const svfloat32_t h2i){
  svbool_t pg = svptrue_b32();
  svfloat32_t u = svmul_f32_z(pg,r2,h2i);
  svfloat32_t du = svmin_n_f32_z(pg,u,1.0f);
  svfloat32_t index_f = svmul_n_f32_z(pg,du,16.0f);
  svuint32_t index = svcvt_u32_f32_z(pg,index_f);
  du = svmsb_f32_z(pg0,svdup_n_f32(0.0625f),svcvt_f32_u32_z(pg0,index),du);
  svfloat32_t c0 = svtbl_f32(tab0,index);
  svfloat32_t c1 = svtbl_f32(tab1,index);
  svfloat32_t c2 = svtbl_f32(tab2,index);
  svfloat32_t c3 = svtbl_f32(tab3,index);
  svfloat32_t c4 = svtbl_f32(tab4,index);
  svfloat32_t ret = svmad_f32_z(pg,du,c4,c3);
  ret = svmad_f32_z(pg,du,ret,c2);
  ret = svmad_f32_z(pg,du,ret,c1);
  ret = svmad_f32_z(pg,du,ret,c0);
  return ret;
}
```

- sqrtが消える(sqrtのレイテンシは98)
- 上記は4次多項式で近似しているが，ここは3次でも精度に影響がない(必要な精度にあわせて演算を削れる)
- テーブル命令のレイテンシは6(加減乗算の2/3)

まとめ

- 粒子間相互作用計算カーネルジェネレータPIKGの紹介
  - DSLで粒子間相互作用計算を記述してジェネレータにかけると，オプションによってさまざまなアーキテクチャに最適化されたカーネルが生成される
  - 現在ARM SVE/AVX-512/AVX2向けの最適化が利用可能
  - 富岳向けに最適化オプションを追加し，およそ20から30% の効率のカーネルを生成できている
  - GPU(CUDA)やPEZY-SC2などにも対応予定
  - 近日githubで公開予定

謝辞

本研究は，文部科学省「富岳」成果創出加速プログラム「宇宙の構造形成と進化から惑星表層環境変動までの統一的描像の構築」の一環として実施したものです。

その他富岳を使って

- 使い勝手としては基本的に京と変わらない(らしい)
- SIMD化は手でやったほうが性能は出る(のでPIKG作ったと言える)
- できたてのシステムではよくあることでしょうが，実装の過程でコンパイラのバグを何個か踏んだ．対処法はあるが直ってなかったりするのでポータルのレポートやマニュアル(既知の問題とか)はしっかり読みましょう
- (頑張って最適化したい人向け)A64FXの気持ちになるためには理研シミュレータ＋gem5可視化ソフト(Konata)が便利
  - ロボ太先生(@kaityo256)の記事を読もう
  - https://qiita.com/kaityo256/items/00fc50221d86ce3ff2ea