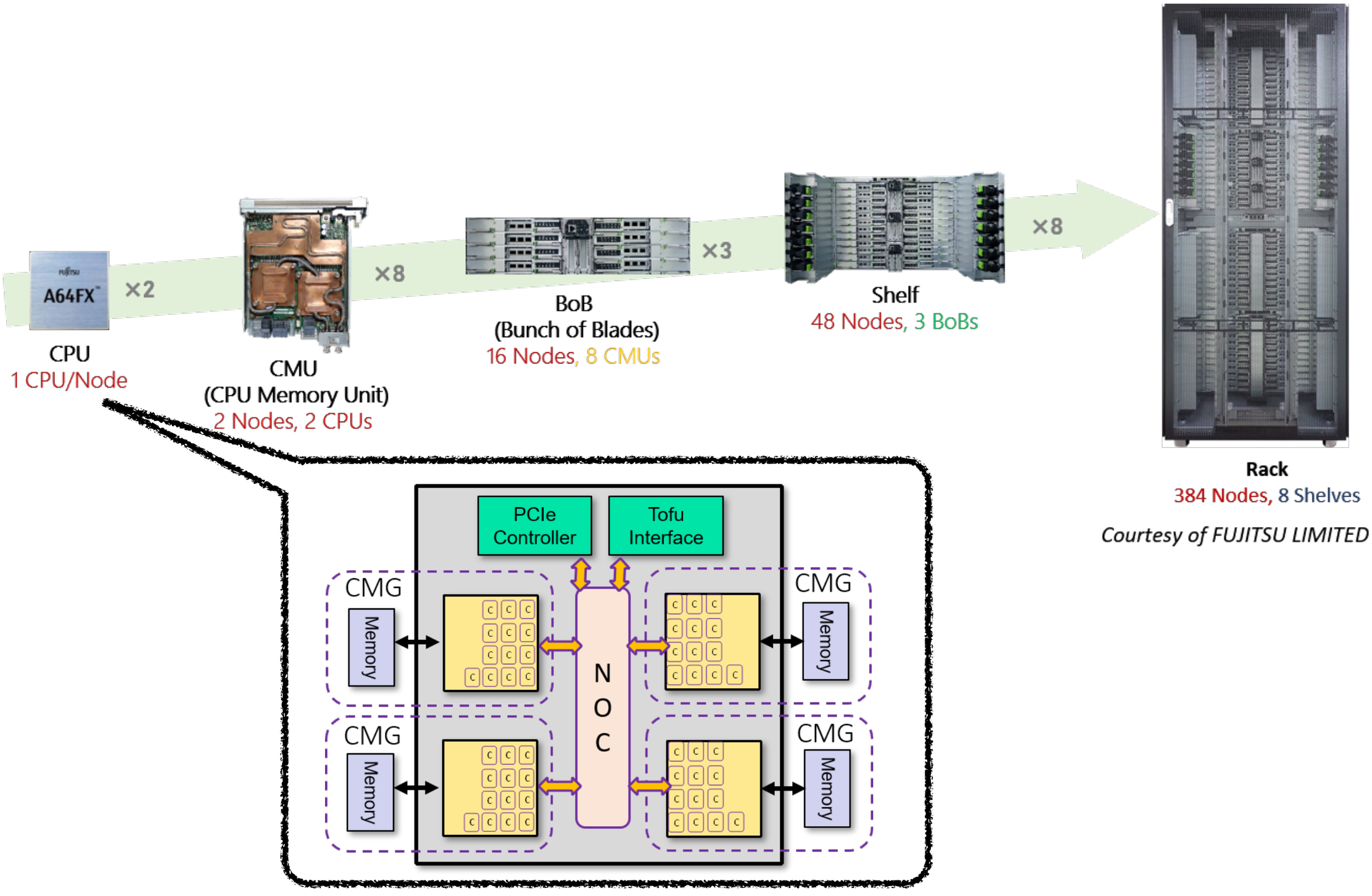
相互作用計算カーネルジェネレータPIKGの紹介

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

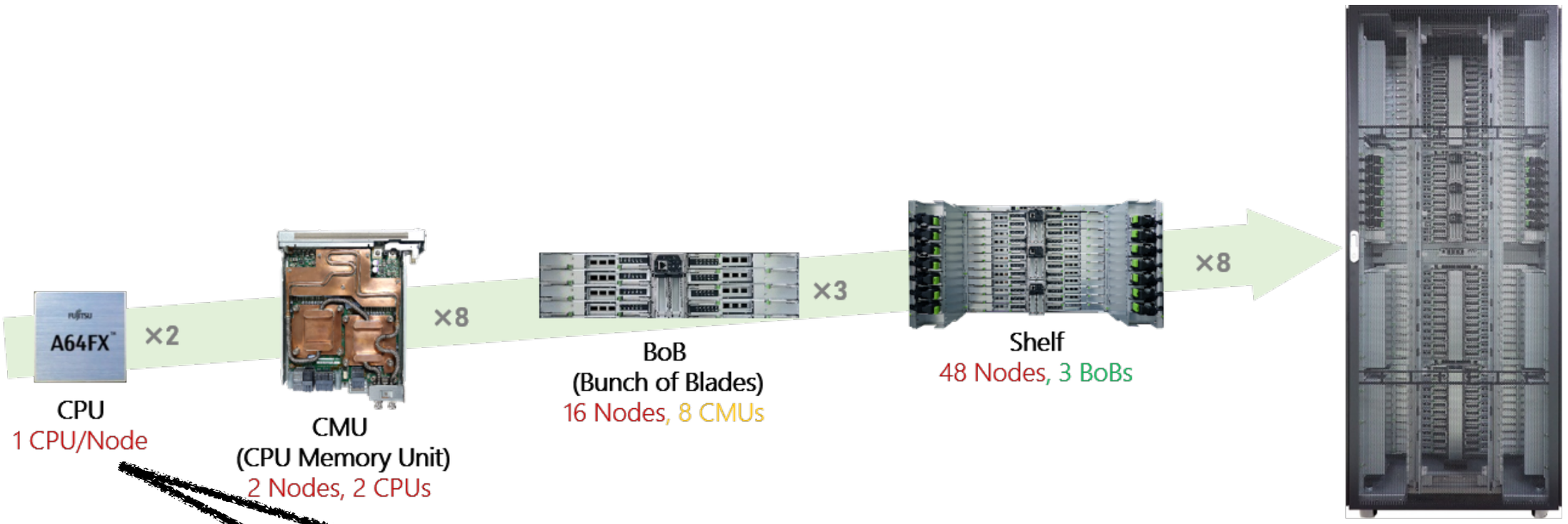
並列計算機向け最適化

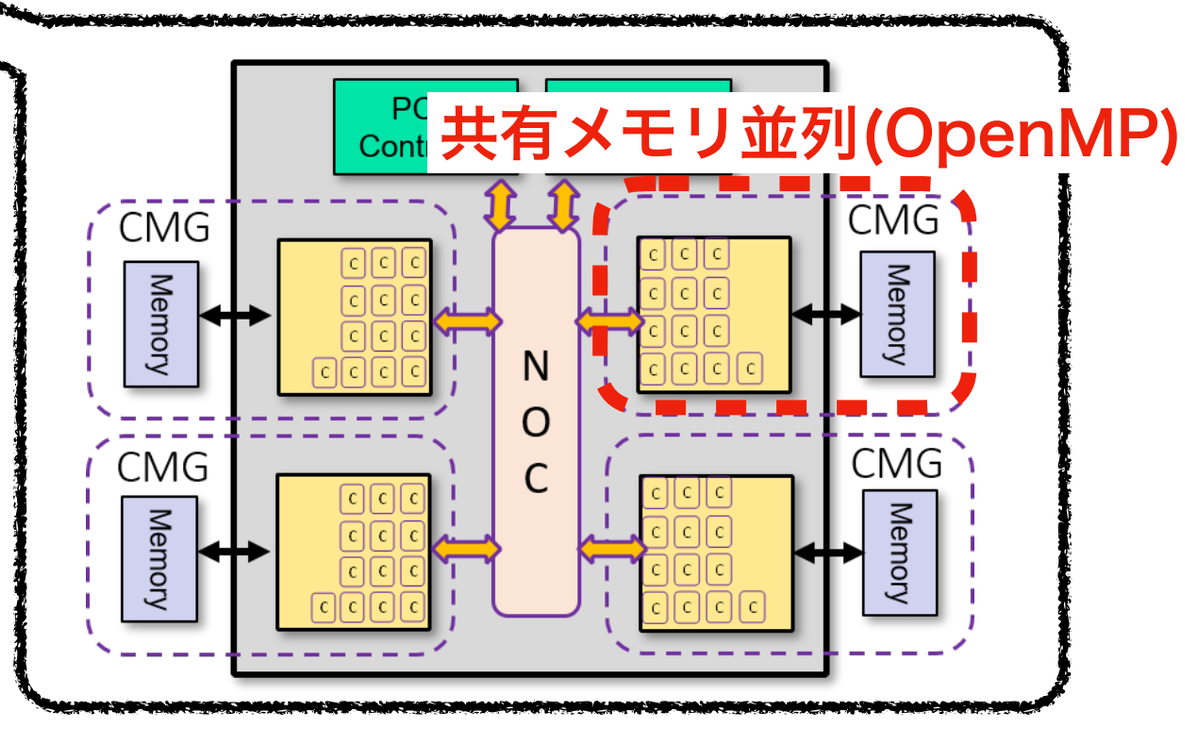


各図はhttps://postk-web.r-ccs.riken.jp/spec.htmlより

並列計算機向け最適化

分散メモリ並列(MPI)

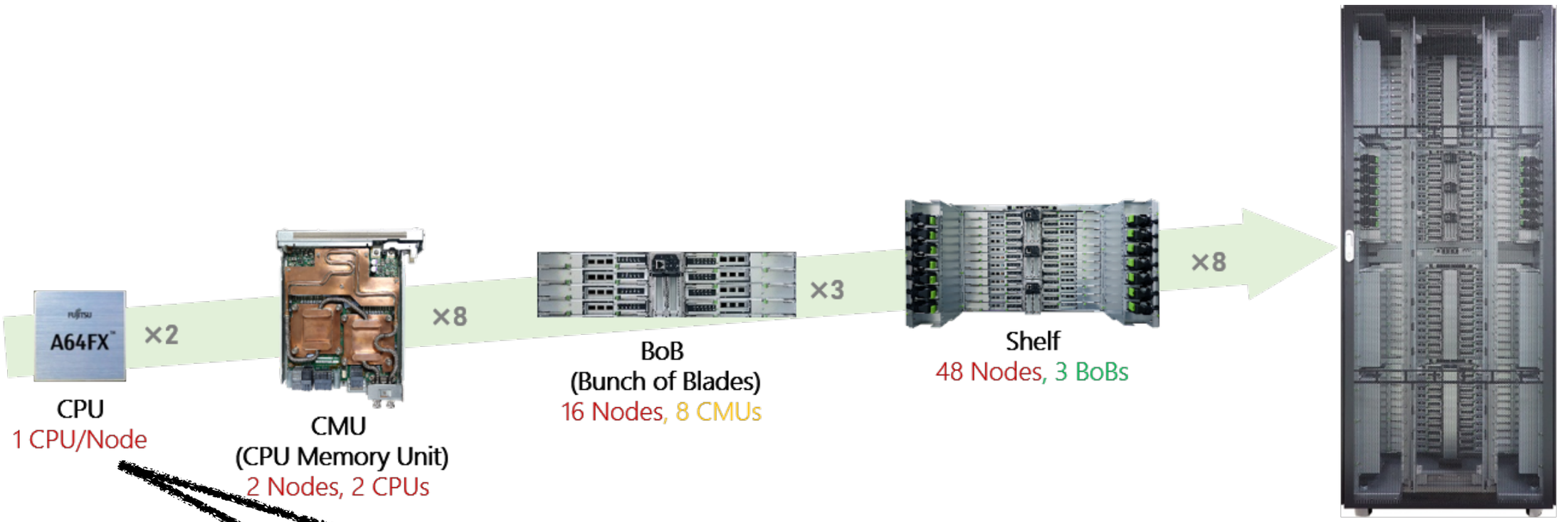




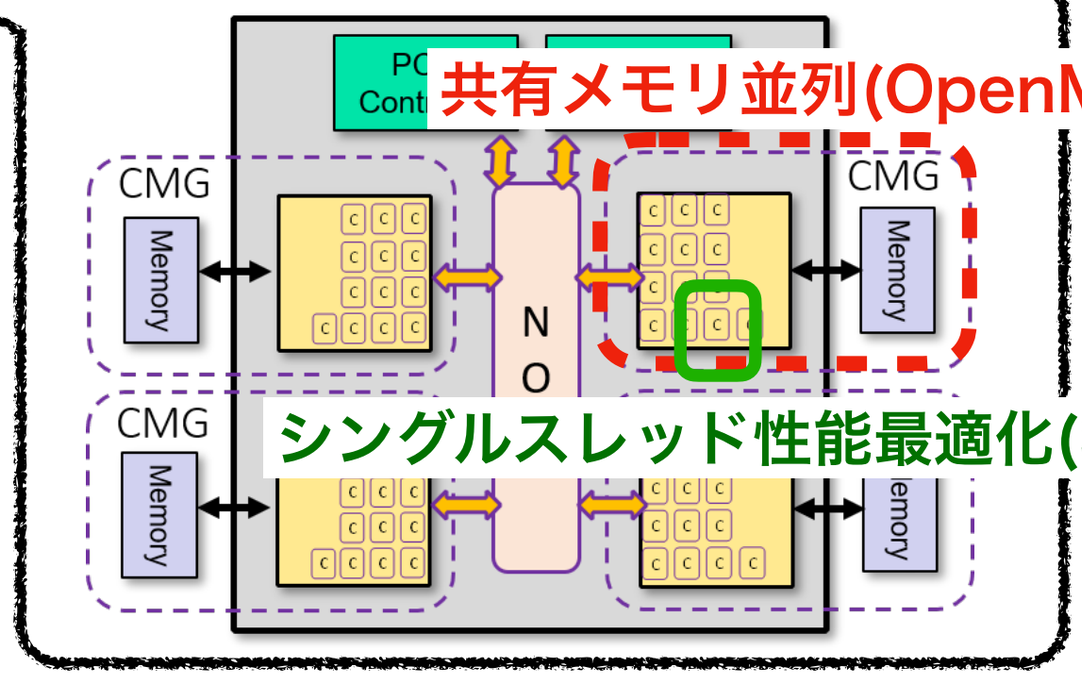
各図はhttps://postk-web.r-ccs.riken.jp/spec.htmlより

並列計算機向け最適化

分散メモリ並列(MPI)



Courtesy of FUJITSU LIMITED



各図はhttps://postk-web.r-ccs.riken.jp/spec.htmlより

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(最適化度合い)に大きく依存
- (大抵の場合)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，できあがるものは再利用できない
- ひとつコードを書いたらさまざまな計算機上で速く動いてほしい！

PIKG

- Paricle-particle Interaction Kernel Generatorの略
- ぱいくじーかぱいくって呼んでください
- 相互作用計算をDSL(ドメイン特化言語)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向け最適化コード生成

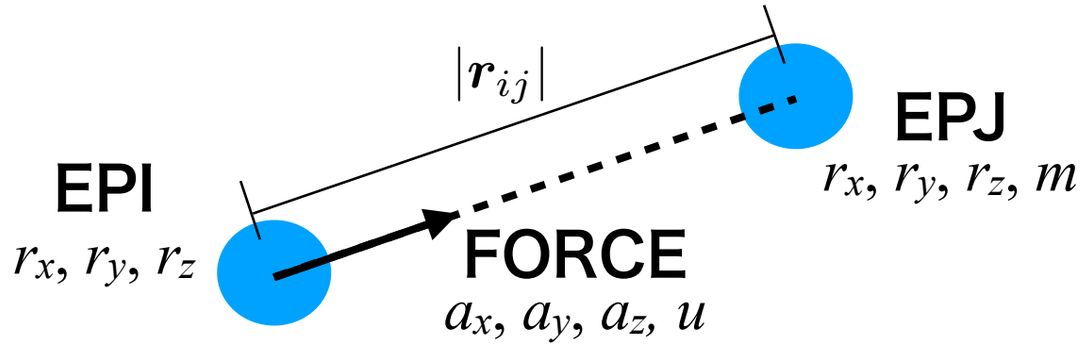
粒子間相互作用計算の一般化 (どんな関数が生成されるか)

```
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

まずはDSLの書き方から

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

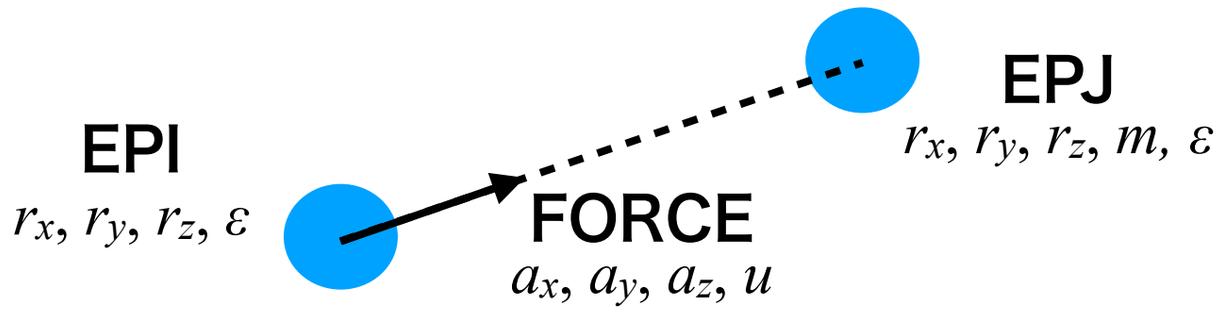


- ユーザは相互作用を及ぼされる粒子(EPI)と及ぼす粒子(EPJ)とEPIにかかる相互作用(FORCE)の変数を定義
  - 例えばEPは座標($\boldsymbol{r}$)，質量(m)，FORCEは加速度($\boldsymbol{a}$)
- 定義した変数と関数を用いて粒子間相互作用計算を記述
- オプション等を設定してジェネレータにかけるとカーネルができる

例(N体モノポール)

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

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



変数宣言部:

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

関数宣言部:

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

カーネル記述部:

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; }
};
```

使える型

- F32/F64：浮動小数点数(float, doubleに相当)
- S32/S64：符号付き整数(int32_t, int64_tに相当)
- U32/U64：符号なし整数(uint32_t, uint64_tに相当)
- F32vec/F64vec：浮動小数点数ベクトル型(3次元)
- F32vec2/F64vec2：浮動小数点数ベクトル型(2次元)
- F32vec3/F64vec3：浮動小数点数ベクトル型(3次元)
- F32vec4/F64vec4：浮動小数点数ベクトル型(4次元)
- C++のコード上では名前空間PIKG上で定義されている
  - PIKG::F64, PIKG::F64vecなど

使える演算子

- + - / * : 四則演算
- && || : 論理演算
- == != < > <= >= : 等号・比較
- ( ) : 括弧
- [ ] : 配列アクセス

予約関数(よく使うもの)

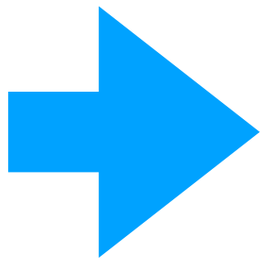
- sqrt : 平方根
- rsqrt ：逆数平方根
- inv ：逆数
- max / min : 最大/最小

変数宣言部

- EPI/EPJ/FORCEのどれに当たるかを宣言
- C++のクラスと1対1対応させる
  - 計算に使わない変数も宣言する

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

```
struct EPI{
  F64vec r;
};
struct EPJ{
  F64vec r;
  F64 m;
};
struct Force{
  F64vec a;
  F64 u;
};
```



```
EPI F64vec ri:r

EPJ F64vec rj:r
EPJ  F64 mj:m

FORCE F64vec a:a
FORCE F64 u:u
```

関数宣言部

- 相互作用記述部で使う関数を定義
- 型は推論されるので指定しなくて良い
- 必ず返り値を書く

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

```
function fmadd(a,b,c)
  ret = a*b + c
  return ret
end
```

相互作用記述部

- 一時変数やFORCE変数に値を代入していく
- EPI/EPJ/FORCE変数から型推論されるので一時変数の型は書く必要がない

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

```
rij = ri - rj
r2 = rij*rij
rinv = rsqrt(r2)
a -= (mj*rinv*rinv*rinv) * rij
u -= mj * rinv
```

条件分岐

```
if EXP1
  STATE1
elsif EXP2
  STATE2
else
  STATE3
endif
```

- Rubyの文法に準拠(してない)
- elsif以下は省略可能(endifは書く)

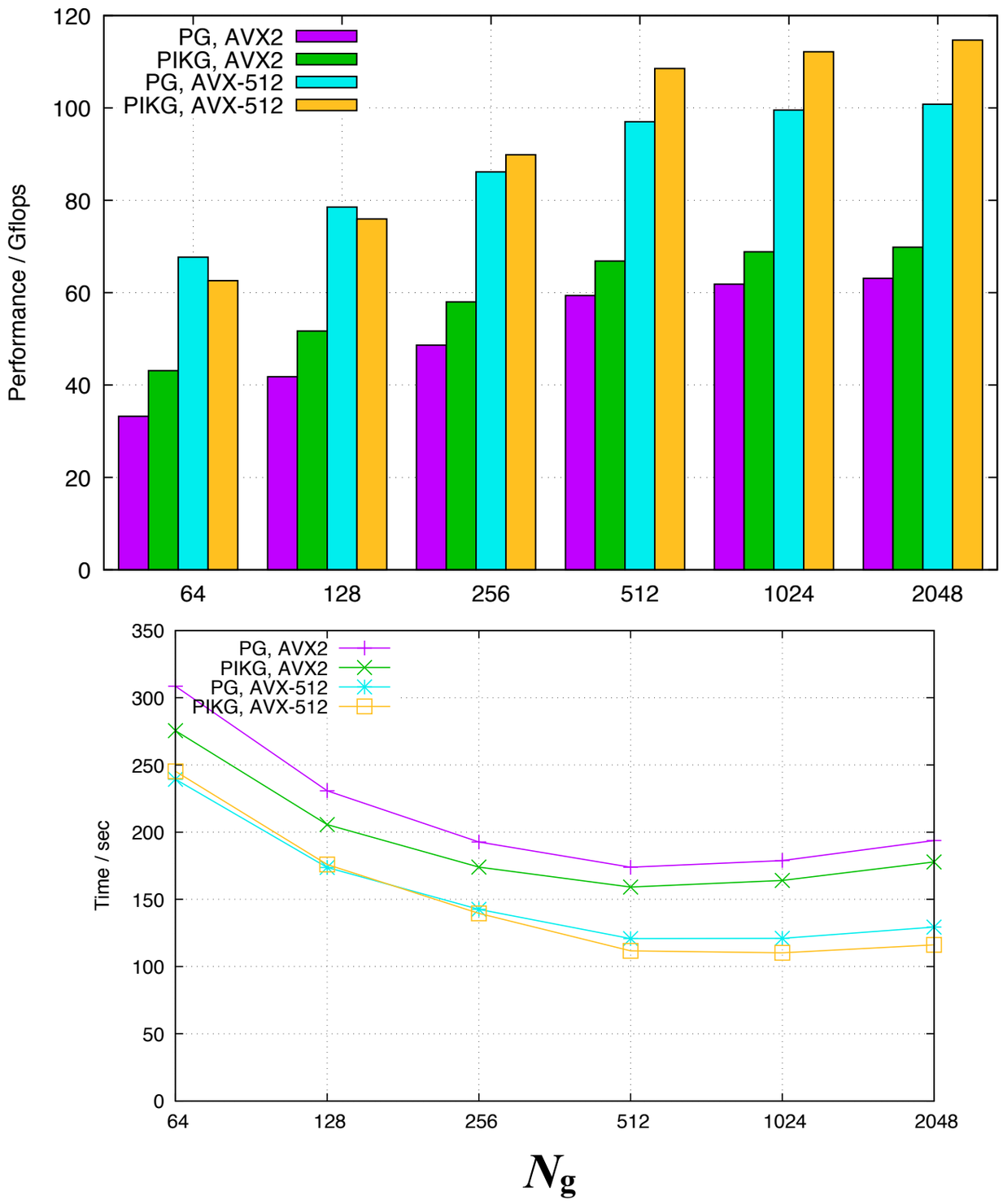
カーネル生成

- $(PIKG)/bin/pikg [options] -i INPUT -o OUTPUT
- よく使うオプション
  - —conversion-type ARCH : reference/AVX2/AVX-512/A64FX でターゲットアーキテクチャを指定
  - —epi-name NAME : EPIクラスのC++での名前を指定
  - —epj-name NAME : EPJクラスのC++での名前を指定
  - —force-name NAME : FORCEクラスの(以下略)

注意点

- 単精度と倍精度の演算が混ざるようなカーネルには対応していない
- 相互作用カーネルではFORCE変数が初期化されていると前提で計算を行うので事前の初期化を忘れない

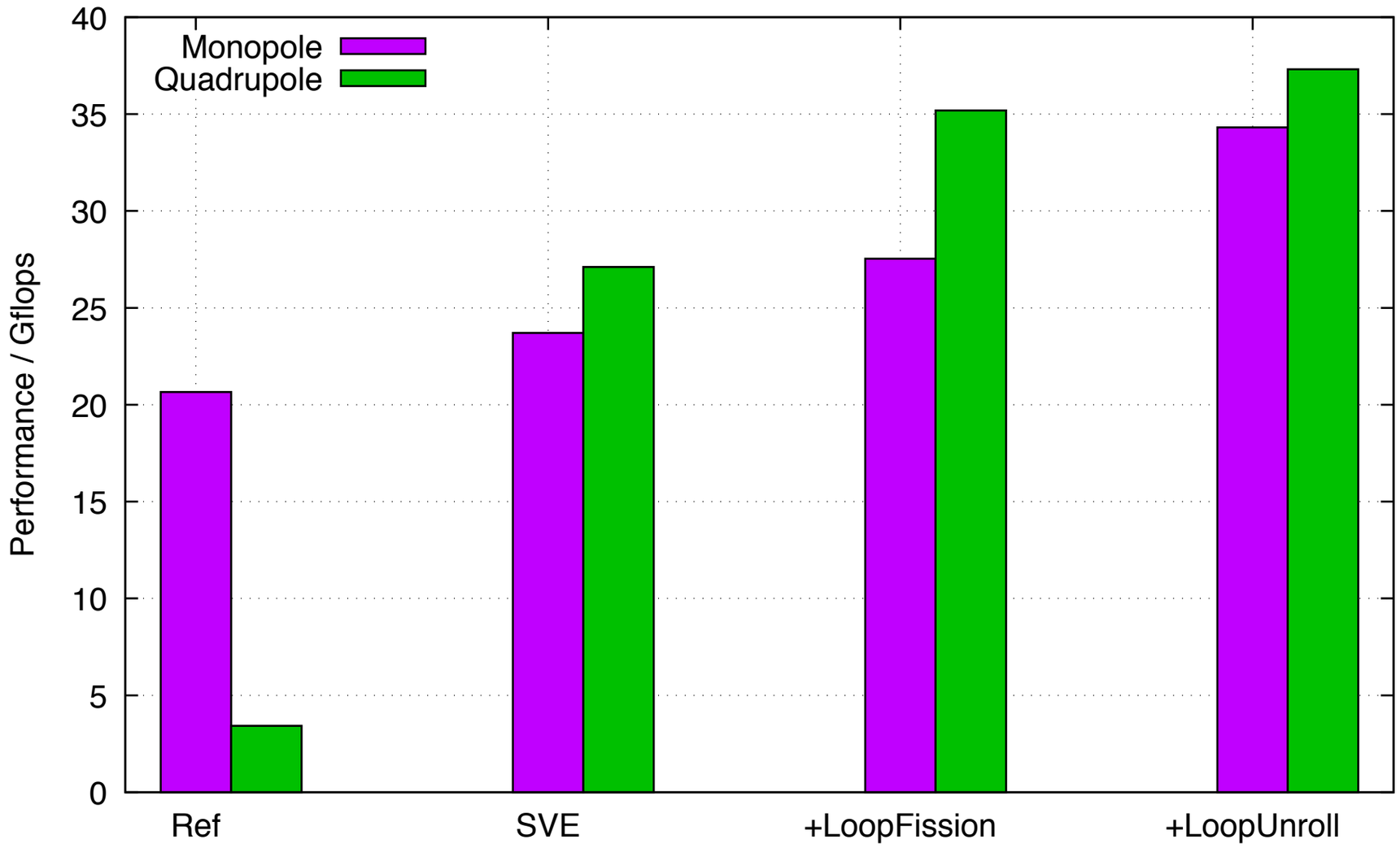
AVX2/AVX-512でのベンチマーク



- 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

富岳向け最適化

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

最後に

- PIKGの簡単な使い方を説明した
- PIKG単体は https://github.com/FDPS/PIKG からDL可能
- FDPSをDLするとPIKGも入っている
- $(FDPS)/sample/c++/nbodyにはPIKGを使ったサンプルがあるのでチュートリアルに従って試してみてください