关于格点QCD计算方法的新思考

宫明

中国科学院高能物理研究所

“非微扰方法及其在高能物理中的应用”研讨会

中国科学技术大学彭桓武中心

2024.10.25

问题的提出……

- 2017年，我参与一个科技部重点研发专项，在神威太湖之光上研发格点QCD代码
- 项目最终圆满成功，超额完成了各种考核指标
- 但我的目标远远没有完成：
  - 在申威架构上原生的代码只完成了最基本的Wilson费米子传播子求解，包装为Chroma的模块
  - 最慢的函数被加速之后，第二慢的函数便替代它成为最慢的……

问题的提出……

- 2017年，我参与一个科技部重点研发专项，在神威太湖之光上研发格点QCD代码
- 项目最终圆满成功，超额完成了各种考核指标
- 但我的目标远远没有完成：
  - 在申威架构上原生的代码只完成了最基本的Wilson费米子传播子求解，包装为Chroma的模块
  - 最慢的函数被加速之后，第二慢的函数便替代它成为最慢的……
- 在国产超算上研发格点QCD软件的进程，路漫漫其修远：
  - 有二十年历史的Chroma包含约55万行代码
  - 用于GPU加速的QUDA代码包含约13万行代码
    - 而且athread/hthread比CUDA罗嗦得多，写出来得更长
  - 把这么多代码，在神威和天河上分别重写一份并各自做优化？？

问题的提出……

- 2017年，我参与一个科技部重点研发专项，在神威太湖之光上研发格点QCD代码
- 项目最终圆满成功，超额完成了各种考核指标
- 但我的目标远远没有完成：
  - 在申威架构上原生的代码只完成了最基本的Wilson费米子传播子求解，包装为Chroma的模块
  - 最慢的函数被加速之后，第二慢的函数便替代它成为最慢的……
- 在国产超算上研发格点QCD软件的进程，路漫漫其修远：
  - 有二十年历史的Chroma包含约55万行代码
  - 用于GPU加速的QUDA代码包含约13万行代码
    - 而且athread/hthread比CUDA罗嗦得多，写出来得更长
  - 把这么多代码，在神威和天河上分别重写一份并各自做优化？？
- 不对！有什么**概念**没有辨析清楚！

关于格点QCD计算方法的新思考

宫明

中国科学院高能物理研究所

“非微扰方法及其在高能物理中的应用”研讨会

中国科学技术大学彭桓武中心

2024.10.25

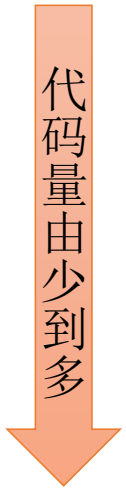
软件是计算方法的载体，但……

软件是计算方法的载体，但……

- 格点QCD用到的所有计算方法
  - 大约十几个课时就能全部讲清楚
  - 写成公式和伪代码，大约占几十页
- 几十万行代码包含：
  - 计算方法的实现
    - 我们真正关心的部分！
  - 软硬件平台的适配
    - 数据结构、并行化、向量化、数据传输、控制流、任务调度、……
  - 封装与接口
    - 数据对象、算法参数、IO、人机界面、……
  - 代码复用设计
    - 编程模型、设计模式、重载/继承/多态、……

软件是计算方法的载体，但……

- 格点QCD用到的所有计算方法
  - 大约十几个课时就能全部讲清楚
  - 写成公式和伪代码，大约占几十页
- 几十万行代码包含：
  - 计算方法的实现
    - 我们真正关心的部分！
  - 软硬件平台的适配
    - 数据结构、并行化、向量化、数据传输、控制流、任务调度、……
  - 封装与接口
    - 数据对象、算法参数、IO、人机界面、……
  - 代码复用设计
    - 编程模型、设计模式、重载/继承/多态、……



软件是计算方法的载体，但……

- 格点QCD用到的所有计算方法
  - 大约十几个课时就能全部讲清楚
  - 写成公式和伪代码，大约占几十页
- 几十万行代码包含：
  - 计算方法的实现
    - 我们真正关心的部分！

目的

  - 软硬件平台的适配
    - 数据结构、并行化、向量化、数据传输、控制流、任务调度、……
  - 封装与接口
    - 数据对象、算法参数、IO、人机界面、……
  - 代码复用设计
    - 编程模型、设计模式、重载/继承/多态、……

手段

软件是计算方法的载体，但……

- 格点QCD用到的所有计算方法
  - 大约十几个课时就能全部讲清楚
  - 写成公式和伪代码，大约占几十页
- 几十万行代码包含：
  - 计算方法的实现
    - 我们真正关心的部分！
  - 软硬件平台的适配
    - 数据结构、并行化、向量化、数据传输、控制流、任务调度、……
  - 封装与接口
    - 数据对象、算法参数、IO、人机界面、……
  - 代码复用设计
    - 编程模型、设计模式、重载/继承/多态、……

为研究者服务

为计算机服务

为程序员服务

软件是计算方法的载体，但……

- 格点QCD用到的所有计算方法
  - 大约十几个课时就能全部讲清楚
  - 写成公式和伪代码，大约占几十页
- 几十万行代码包含：
  - 计算方法的实现
    - 我们真正关心的部分！
  - 软硬件平台的适配
    - 数据结构、并行化、向量化、数据传输、控制流、任务调度、……
  - 封装与接口
    - 数据对象、算法参数、IO、人机界面、……
  - 代码复用设计
    - 编程模型、设计模式、重载/继承/多态、……

为研究者服务

为计算机服务

为~~程序员~~服务

软件是计算方法的载体，但……

- 格点QCD用到的所有计算方法
  - 大约十几个课时就能全部讲清楚
  - 写成公式和伪代码，大约占几十页
- 几十万行代码包含：
  - 计算方法的实现
    - 我们真正关心的部分！
  - 软硬件平台的适配
    - 数据结构、并行化、向量化、数据传输、控制流、任务调度、……
  - 封装与接口
    - 数据对象、算法参数
  - 代码复用设计
    - 编程模型、设计模式

为研究者服务

为计算机服务

为~~程序员~~服务

自动代码生成！

自动代码生成技术是未来科学计算的关键共性技术

- 跨平台代码的前置技能
  - 不同软硬件平台的差异再大，具体计算过程的代码也相差不多。能生成一个平台的代码，就意味着可以通过简单地调整，生成面向其他平台的代码。
  - 国产超算平台的编程模型互不兼容，但自动生成代码技术可以实现产生针对每个平台的代码，边际成本极低。
  - 这意味着国产超算有希望摆脱内卷气氛，实现合作共赢！
- 自动代码优化的前置技能
  - 人工智能是未来一段时间持续的热点技术，发展空间还有很大。
  - 自动代码优化可以用传统算法，也很有希望借人工智能的东风——虽然目前还不知道具体怎么做。
  - 人工智能是用大量数据喂出来的；自动优化也应是用大量程序喂出来的。
  - 以特定IR为参数批量生成“等价但不同”的程序，是未来这个方向发展的基础。
- 领域专家的人机界面
  - 面向领域专家的DSL通过IR对接代码生成，可以实现面向科研的敏捷开发平台！
  - 对于格点QCD而言，这个语言应直接面向物理公式，即张量表达式。
  - 因此，我们直接需求的自动代码生成技术是：张量表达式→程序源代码。

不需要自动生成所有代码！

- 如果能自动生成任意功能的代码，自然很好；研发目标过高可能难以实现，应退而求其次。
- 我们的确不需要任意功能，应当做一些减法：
  - 大规模高性能计算，不需要也不应该有运行时人机交互！
  - 带有副作用的代码片段应当被拆解为纯函数，方便自动优化环节的分析。
  - 带有间接引用的复杂数据结构，似乎没有看到这样的需求。
  - 其他各类不需要考虑的情况
- 我们明确需要的功能：
  - 物理公式翻译为计算代码
    - 这是自动代码生成的主要任务
  - 循环迭代的控制代码
    - 比较简单，可以自动生成，也可以考虑手写

不需要自动生成所有代码！

- 如果能自动生成任意功能的代码，自然很好；研发目标过高可能难以实现，应退而求其次。
- 我们的确不需要任意功能，应当做一些减法：
  - 大规模高性能计算，不需要也不应该有运行时人机交互！
  - 带有副作用的代码片段应当被拆解为纯函数，方便自动优化环节的分析。
  - 带有间接引用的复杂数据结构，似乎没有看到这样的需求。
  - 其他各类不需要考虑的情况
- 我们明确需要的功能：
  - 物理公式翻译为计算代码
    - 这是自动代码生成的主要任务
  - 循环迭代的控制代码
    - 比较简单，可以自动生成，也可以考虑手写
  - ↑只有这两类功能吗？

格点QCD的算法的热点解析

- 产生组态：
  - Heatbath：主要是小矩阵计算
  - MD：
    - 费米力：级数展开→线性方程求解→D-slash
    - 时间积分步进：费米子作用量→D-slash
  - Metropolis：
    - 费米子行列式→线性方程求解→D-slash
- 计算夸克传播子：
  - 线性方程求解→D-slash
  - 非局域费米子→线性方程求解→D-slash
  - 蒸馏向量→矩阵分解→矩阵计算
  - 预处理→各种矩阵计算
- 计算强子关联函数：
  - 计算张量表达式
- 拟合物理观测量
  - 二次型的优化

格点QCD的算法的热点解析

- 产生组态：
  - Heatbath：主要是小矩阵计算
  - MD：
    - 费米力：级数展开→线性方程求解→D-slash
    - 时间积分步进：费米子作用量→D-slash
  - Metropolis：
    - 费米子行列式→线性方程求解→D-slash
- 计算夸克传播子：
  - 线性方程求解→D-slash
  - 非局域费米子→线性方程求解→D-slash
  - 蒸馏向量→矩阵分解→矩阵计算
  - 预处理→各种矩阵计算
- 计算强子关联函数：
  - 计算张量表达式
- 拟合物理观测量
  - 二次型的优化

- 标绿色部分都可以写成张量表达式的形式
- 其余普通计算大多也可以写成张量表达式的形式
- 算法逻辑主要是循环迭代
  - “所有的计算都是优化；所有的优化都是迭代”
- 没有复杂的动态数据结构
- 没有复杂的算法流程

宽泛的“张量”概念：多维度、用指标索引的数据

$$\not{D} = \sum_{\mu} \left[(1^{(4)} - \gamma_\mu) U_\mu(\vec{x}) \delta_{\vec{x}+\mu,\vec{y}} + (1^{(4)} + \gamma_\mu) U_\mu^\dagger(\vec{x} - \mu) \delta_{\vec{x}-\mu,\vec{y}} \right]$$

宽泛的“张量”概念：多维度、用指标索引的数据

$$\not{\partial} = \sum_{\mu} \left[(1^{(4)} - \gamma_\mu) U_\mu(\vec{x}) \delta_{\vec{x}+\mu,\vec{y}} + (1^{(4)} + \gamma_\mu) U^{\dagger}_{\mu}(\vec{x}-\mu) \delta_{\vec{x}-\mu,\vec{y}} \right]$$

```
typedef struct
{
  Float re, im;
}complex;

typedef struct
{
  complex c1, c2, c3;
}su3_vector;

typedef struct
{
  complex c11, c12, c13;
  complex c21, c22, c23;
  complex c31, c32, c33;
}su3;

typedef struct
{
  su3_vector c1,c2,c3,c4;
}spinor;

// gauge and spinor field
typedef su3 su3_field[NT][NZ][NY][NX][4];
typedef spinor spinor_field[NT][NZ][NY][NX];
```

```
void dslash(spinor_field result, su3_field u, spinor_field input)
{
  spinor tmp1, tmp2;

  for(int it=0; it<NT; it++)
    for(int iz=0; iz<NZ; iz++)
      for(int iy=0; iy<NY; iy++)
        for(int ix=0; ix<NX; ix++)
          {
            init_spinor_zero(&tmp2);

            // t-direction, mu=0
            su3_spinor_mul(&tmp1, &(u[it][iz][iy][ix][0]),
                           &(input[INC_T(it)][iz][iy][ix]));
            one_pm_gamma_mul(&tmp2, 0, -1, &tmp1);
            su3_dag_spinor_mul(&tmp1, &(u[it][iz][iy][ix][0]),
                               &(input[DEC_T(it)][iz][iy][ix]));
            one_pm_gamma_mul(&tmp2, 0, 1, &tmp1);

            // x-direction, mu=1
            su3_spinor_mul(&tmp1, &(u[it][iz][iy][ix][1]),
                           &(input[it][iz][iy][INC_X(ix)]));
            one_pm_gamma_mul(&tmp2, 1, -1, &tmp1);
            su3_dag_spinor_mul(&tmp1, &(u[it][iz][iy][ix][1]),
                               &(input[it][iz][iy][DEC_X(ix)]));
            one_pm_gamma_mul(&tmp2, 1, 1, &tmp1);

            // y-direction, mu=2
            su3_spinor_mul(&tmp1, &(u[it][iz][iy][ix][2]),
                           &(input[it][iz][INC_Y(iy)][ix]));
            one_pm_gamma_mul(&tmp2, 2, -1, &tmp1);
            su3_dag_spinor_mul(&tmp1, &(u[it][iz][iy][ix][2]),
                               &(input[it][iz][DEC_Y(iy)][ix]));
            one_pm_gamma_mul(&tmp2, 2, 1, &tmp1);

            // z-direction, mu=3
            su3_spinor_mul(&tmp1, &(u[it][iz][iy][ix][3]),
                           &(input[it][INC_Z(iz)][iy][ix]));
            one_pm_gamma_mul(&tmp2, 3, -1, &tmp1);
            su3_dag_spinor_mul(&tmp1, &(u[it][iz][iy][ix][3]),
                               &(input[it][DEC_Z(iz)][iy][ix]));
            one_pm_gamma_mul(&tmp2, 3, 1, &tmp1);

            result[it][iz][iy][ix] = tmp2;

          }
}
```

稀疏张量
实现为代码

稀疏张量
实现为复杂的代码

稠密高维张量
实现为运行时数据

三维稀疏张量
实现为复杂的代码

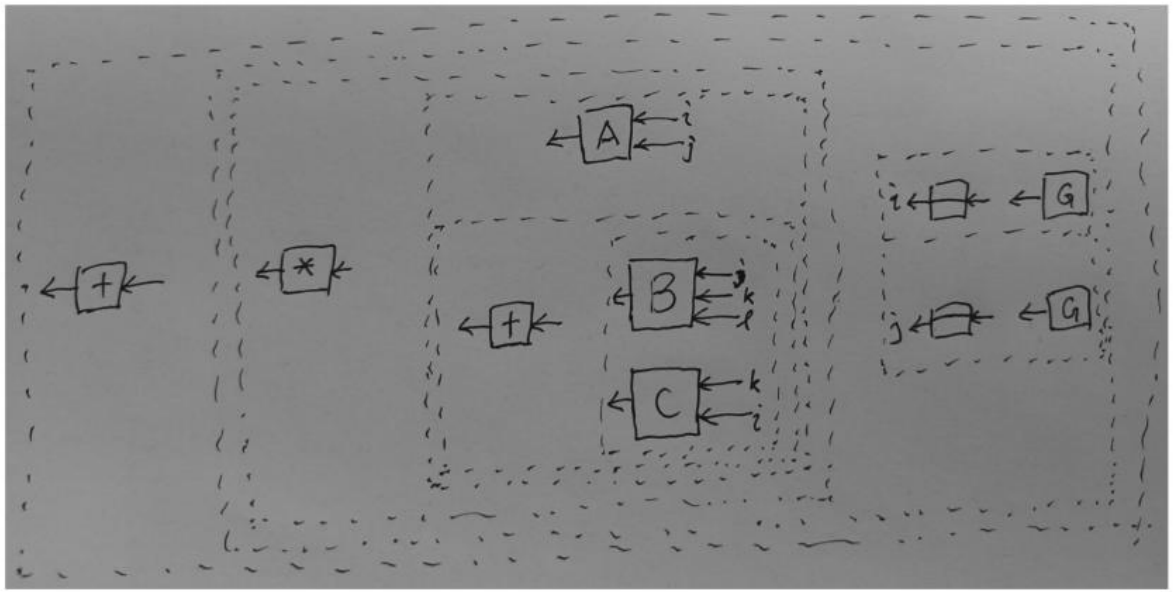
引用自孙玮的SimpleLQCD

MetaTensor软件：张量表达式→中间表示（x语言）→C源代码

需要计算的张量表达式：

$$result_{k,l} = \sum_{i,j} A_{i,j} * (B_{j,k,l} + C_{k,i})$$

用 x 语言转写为：



MetaTensor 把它编译到 C 语言：

```c
#include <stdint.h>
#include <stdlib.h>
void calc(void *res, void *a , void *b, void *c){
        double*var_5_=malloc(2048);
        {
                double*var_2_=(double*)((char *)(b)+0);
                double*var_3_=(double*)((char *)(c)+0);
                double*var_4_=(double*)((char *)(var_5_)+0);
                int32_t var_7_=0;
                int32_t var_8_=0;
                int32_t var_9_=0;
                const int32_t var_10_[4]={ 0,16,32,48};
                const int32_t var_11_[4]={ 0,64,128,192};
                for(int32_t var_12_=0; var_12_<4; var_12_++){
                        int32_t var_13_=var_7_+var_10_[var_12_];
                        int32_t var_14_=var_9_+var_11_[var_12_];
                        const int32_t var_15_[4]={ 0,4,8,12};
                        const int32_t var_16_[4]={ 0,4,8,12};
                        const int32_t var_17_[4]={ 0,16,32,48};
                        for(int32_t var_18_=0; var_18_<4; var_18_++){
                                int32_t var_19_=var_13_+var_15_[var_18_];
                                int32_t var_20_=var_8_+var_16_[var_18_];
                                int32_t var_21_=var_14_+var_17_[var_18_];
                                const int32_t var_22_[4]={ 0,1,2,3};
                                const int32_t var_23_[4]={ 0,4,8,12};
                                for(int32_t var_24_=0; var_24_<4; var_24_++){
                                        int32_t var_25_=var_19_+var_22_[var_24_];
                                        int32_t var_26_=var_21_+var_23_[var_24_];
                                        const int32_t var_27_[4]={ 0,1,2,3};
                                        const int32_t var_28_[4]={ 0,1,2,3};
                                        for(int32_t var_29_=0; var_29_<4; var_29_++){
                                                int32_t var_30_=var_20_+var_27_[var_29_];
                                                int32_t var_31_=var_26_+var_28_[var_29_];
                                                var_4_[var_31_]=0+var_2_[var_25_]+var_3_[var_30_];
                                        }}}}}
        double*var_37_=malloc(2048);
        {
                double*var_34_=(double*)((char *)(a)+0);
                double*var_35_=(double*)((char *)(var_5_)+0);
                double*var_36_=(double*)((char *)(var_37_)+0);
                int32_t var_39_=0;
                int32_t var_40_=0;
                int32_t var_41_=0;
                const int32_t var_42_[4]={ 0,4,8,12};
                const int32_t var_43_[4]={ 0,1,2,3};
                const int32_t var_44_[4]={ 0,64,128,192};
                for(int32_t var_45_=0; var_45_<4; var_45_++){
                        int32_t var_46_=var_39_+var_42_[var_45_];
                        int32_t var_47_=var_40_+var_43_[var_45_];
```

生成代码的优化与多平台适配

- MetaTensor是x语言到C语言的编译器
  - 一般编译器的优化被组织为“pass”，平台适配被组织为“后端”
  - MetaTensor的优化被组织为“插件”，平台适配被组织为“目标”
- 总共有且仅有4个插口
  - “复制”、“求值”、“优化”、“编译”
  - 每个插口上可以挂载一串插件
  - 单个插件可以调用其他插件，甚至调用整个插口
- 目标是这些插口挂载插件集合的方案
  - 针对不同平台，我们选择不同的插件进行优化和编译输出

生成代码的优化与多平台适配

- MetaTensor是x语言到C语言的编译器
  - 一般编译器的优化被组织为“pass”，平台适配被组织为“后端”
  - MetaTensor的优化被组织为“插件”，平台适配被组织为“目标”
- 总共有且仅有4个插口
  - “复制”、“求值”、“优化”、“编译”
  - 每个插口上可以挂载一串插件
  - 单个插件可以调用其他插件，甚至调用整个插口
- 目标是这些插口挂载插件集合的方案
  - 针对不同平台，我们选择不同的插件进行优化和编译输出

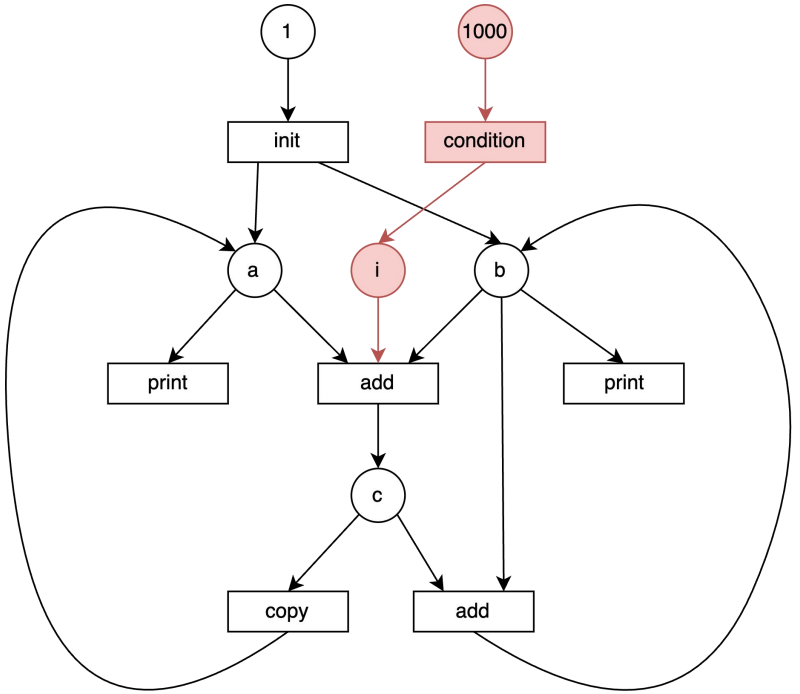
MetaTensor是可以无限扩展的！

代码的碎片化：局限或是革命？

- 自动生成代码在功能上是受限的，至少前期如此
- 不符合标准的算法，可以通过拆解算法流程，分别生成代码的各个部分
  - 这会导致代码的碎片化
  - 这意味着整个编程模型都会被颠覆！
- 传统的编程模型是从顶向下的
  - 先有“主程序”，它逐级向下调度功能模块
  - 影响了各种异构处理器相关的编程模型，它们大多是“主从结构”
- 由AI推动的新编程模型是从底向上的
  - 先有碎片化的“算子”，它逐级向上组装为程序
  - 代码是碎片化的，一般由通用的调度器负责管理，例如Spark
  - 高性能科学计算领域目前尚未有类似的编程模型——历史包袱太重
  - 可能是未来高性能计算的重要发展方向！

算子图模型

- AI领域的算子图模型是有向无环图(DAG)
  - 他们的计算模型简单且固定，因此够用
  - 因此其他领域并没有移植到AI相关的基础软件上
- 循环迭代需要有向有环图
  - 我们定义了对象的状态，以及状态转换规则
  - 我们研发了“调度器”(DDQ)
    - DDQ可以根据依赖关系安排算子的运行
    - DDQ可以自动安排互相无依赖的算子同时并发运行
    - DDQ负责管理所有对象的生存周期



- AI领域的算子图模型里，数据传输是边
- 在DDQ里，数据传输与计算都是算子，算子也是对象
  - 数据传输需要被提升地位！(这是被其他所有编程模型忽视的瓶颈)
  - 带来的好处之一：双缓冲优化技术被直接自动实现

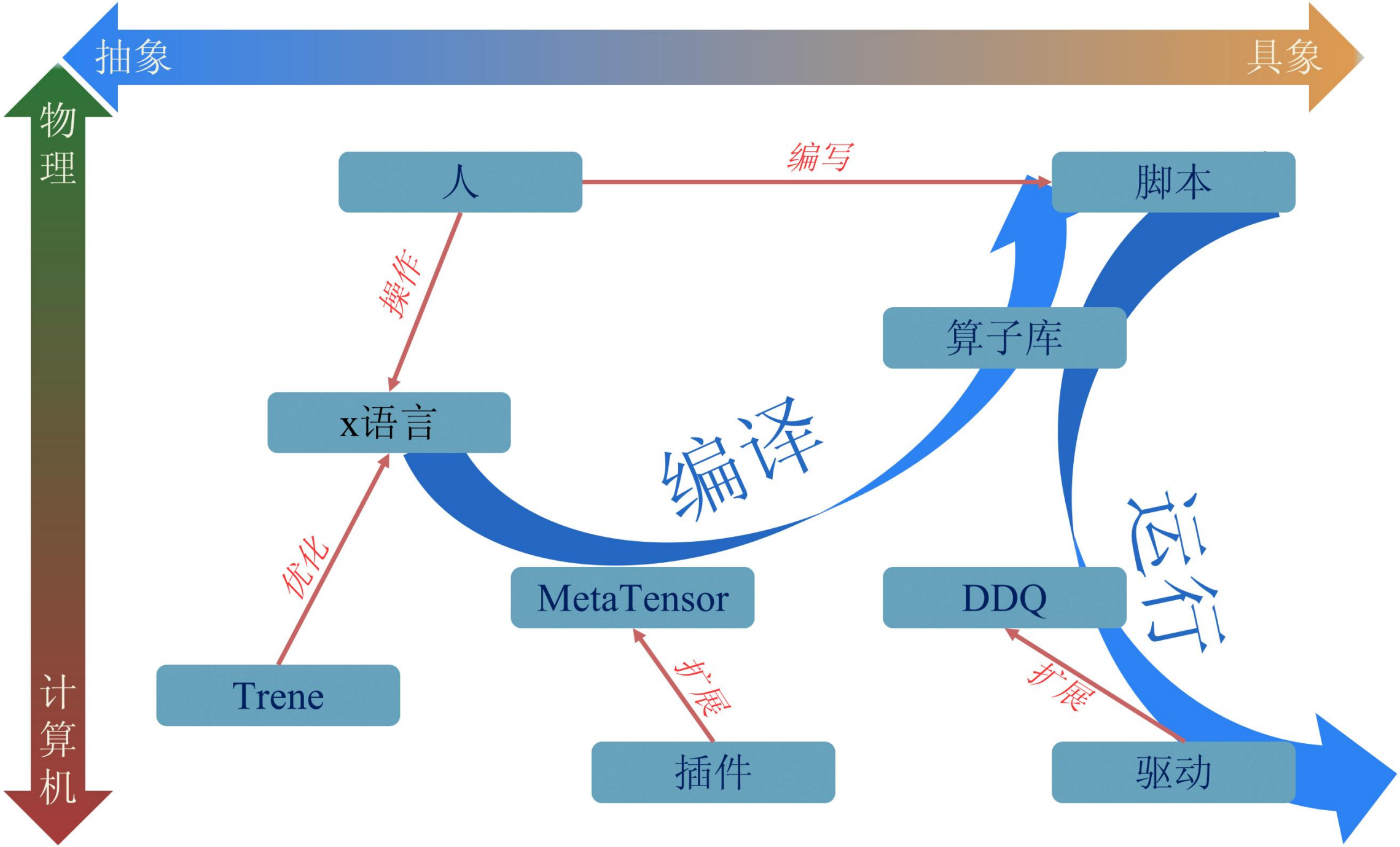
DDQ的分布式研发设计

- DDQ的主体、DDQ的脚本等都已基本成型
- DDQ的算子库
  - 按std、utl、sci、……等分类，以文件夹树的方式摆放
    - 形如：sci/lqcd/chroma/ferm/CalcProp
  - 子文件夹可以是git submodule
    - 不同的算子可以由不同的人进行分布式开发
    - 算子库负责汇集这些代码
- DDQ的驱动
  - DDQ通过驱动运行算子
    - 因此DDQ是跨平台调度器！
  - 针对各种软硬件环境，需要写对应的驱动
    - 驱动研发采用协程模型，非常简单
      - 支持每种新处理器大概只需要几百行代码
    - 驱动也可以类似地采用分布式开发

DDQ的分布式研发设计

- DDQ的主体、DDQ的脚本等都已基本成型
- DDQ的算子库
  - 按std、utl、sci、……等分类，以文件夹树的方式摆放
    - 形如：sci/lqcd/chroma/ferm/CalcProp
  - 子文件夹可以是git submodule
    - 不同的算子可以由不同的人进行分布式开发
    - 算子库负责汇集这些代码
- DDQ的驱动
  - DDQ通过驱动运行算子
    - 因此DDQ是跨平台调度器！
  - 针对各种软硬件环境，需要写对应的驱动
    - 驱动研发采用协程模型，非常简单
      - 支持每种新处理器大概只需要几百行代码
    - 驱动也可以类似地采用分布式开发

DDQ是可以无限扩展的！
而且支持第三方扩展！


抽象
具象
物理
计算机
人
编写
脚本
操作
算子库
x语言
编译
运行
优化
MetaTensor
DDQ
Trene
扩展
扩展
插件
驱动

XPT 研发贡献者（统计至：2024/10/22）

- 中国科学院高能物理研究所

  宫明、陈莹、刘朝峰、毕玉江、孙玮、施春江、蒋翔宇

- 北京航空航天大学

  栾钟治、韩斌、王御臣、肖敏毅、房歆哲、龚煜涵、马世清、刘星宇、李根、王逸杰、李亦白

- 中国科学院计算机网络信息中心

  徐顺、张克龙、韩秉豫、张术飞

- 中国科学院理论物理研究所

  王建成

感谢聆听！