加速库指令加速-SVE指令技术

张彬彬

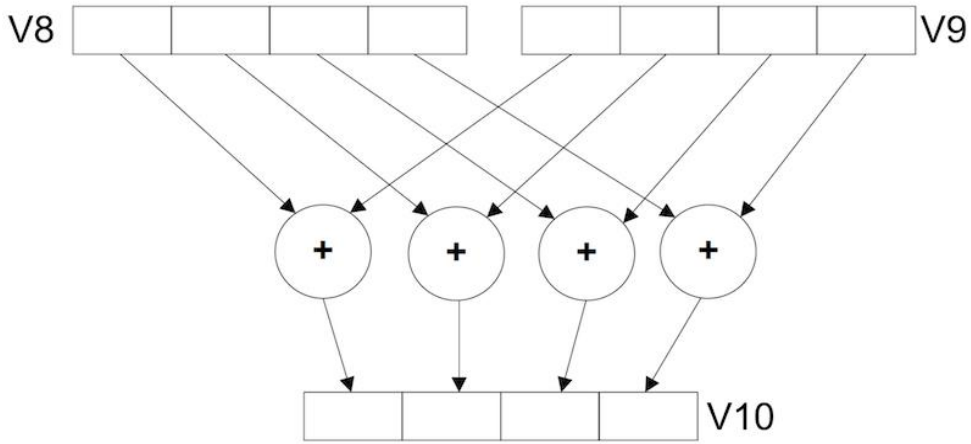
2022.04

目录

引子---从SISD到SIMD

```
// n源输入长度，da初始值， dx源数据指针
double AddReduction(int64_t n, double da, double *dx)
{
    double res = da;
    for (int64_t i = 0; i < n; ++i) {
        res += dx[i];
    }
    return res;
}
```

```
double AddReductionNeon(int64_t n, double da, double *dx)
{
    double res = da;
    float64x2_t res_64f = vdupq_n_f64(0.0);
    int64_t i;
    int64_t loopLen = n & 0xFFFFFFFFFFFFFFFE;
    // vector loop
    for (i = 0; i < loopLen; i += 2) {
        float64x2_t src_64f = vld1q_f64(dx + i);
        res_64f = vaddq_f64(res_64f, src_64f);
    }
    res += vaddvq_f64(res_64f);
    // loop tail
    for (; i < n; ++i) {
        res += dx[i];
    }
    return res;
}
```



ARM向量计算方式

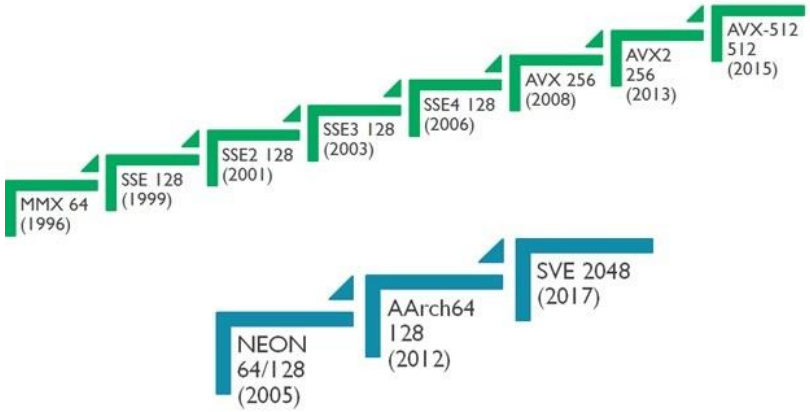
*本研讨材料图片均摘自网络，
如有侵权可联系删除

SVE简介

随着 Neon 架构扩展（其指令集具有固定的 128 位向量长度）的开发，Arm 设计了可伸缩向量扩展 (Scalable Vector Extension SVE) 作为 AArch64 的下一代 SIMD 扩展。SVE 允许灵活的向量长度实现，在 CPU 实现中具有一系列可能的值。向量长度可以从最小 128 位到最大 2048 位不等，以 128 位为增量。SVE 设计保证了相同的应用程序可以在支持 SVE 的不同实现上运行，而无需重新编译代码。SVE 提高了架构对高性能计算 (HPC) 和机器学习 (ML) 应用程序的适用性，这些应用程序需要大量数据处理。

SVE 引入了以下主要功能：

- 可伸缩向量（128bit~2048bit，可取128的整数倍）。
- 推测向量化，即P寄存器。
- 每通道预测，通过P寄存器按通道是否活动进行操作。
- 聚集加载和分散存储。
- 水平和序列化向量操作。



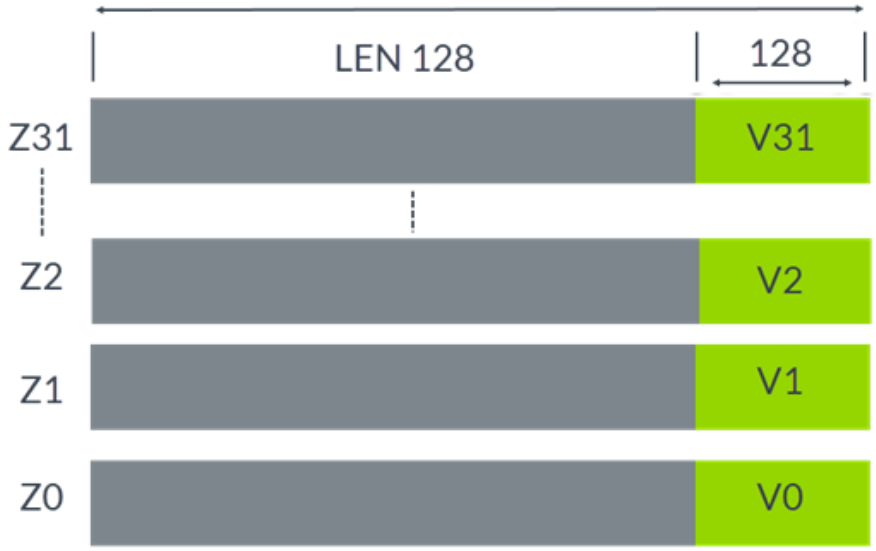
ARM开发者网站：
https://developer.arm.com/

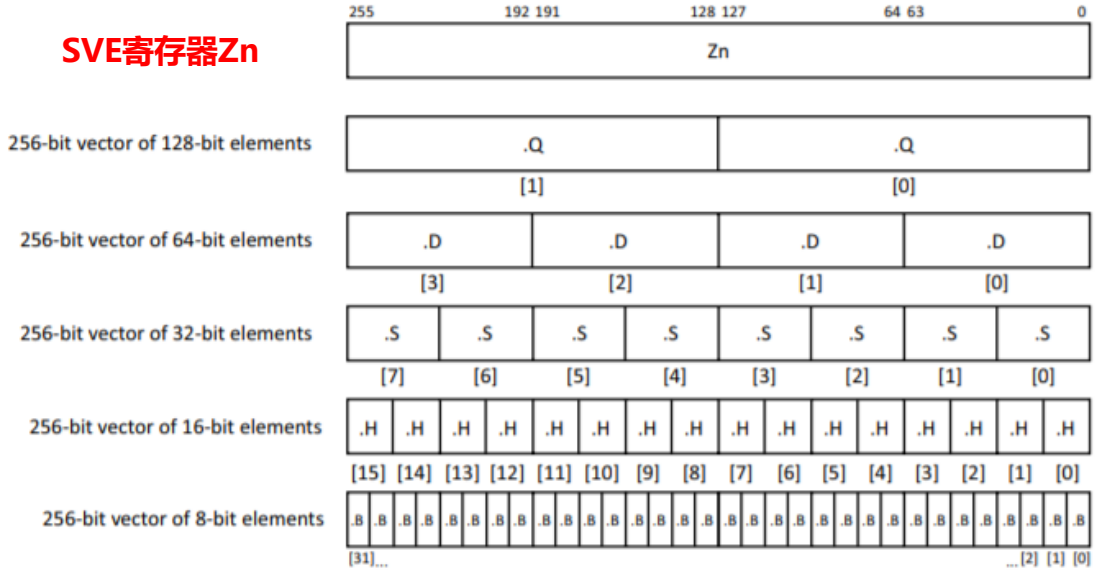
AMR指令集文档下载：
https://developer.arm.com/documentation/ddi0487/ha/?lang=en

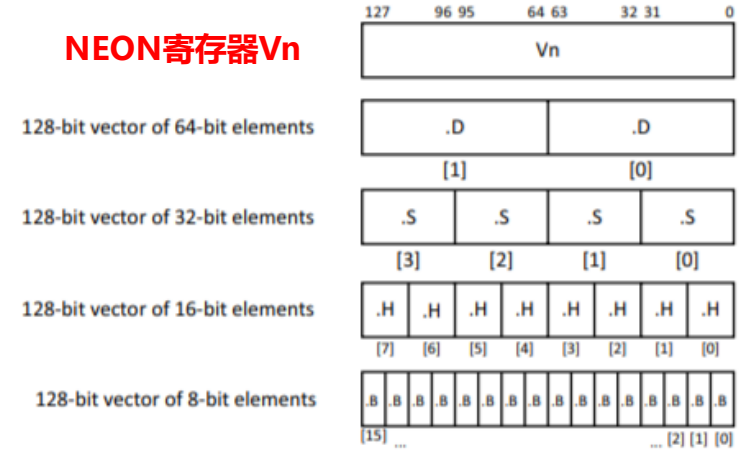
SVE指令集说明：
https://developer.arm.com/documentation/ddi0602/2022-03/SVE-Instructions

SVE简介---向量寄存器Z

- SVE有32个Z寄存器，Z0-Z31可以在微架构中使用 128-2048 位来实现，可取128的整数倍。
- 其可容纳 8、16、32和64位元素，使用后缀.b/.h/.s/.d/.q来标识通道位宽，支持整数、双精度、单精度和半精度浮点元素。
- SVE的Z寄存器与NEON的V寄存器共享低128bit，即Vn寄存器对应Zn寄存器的[127:0]。

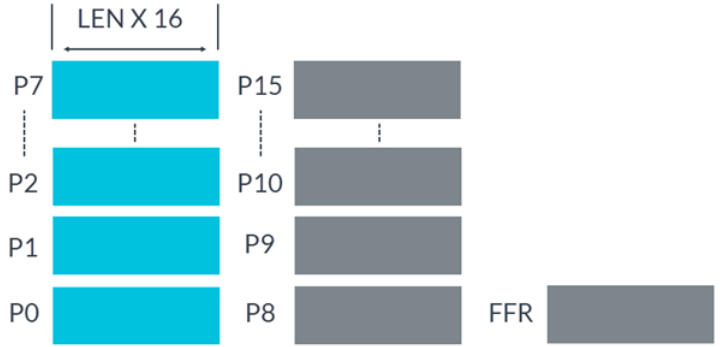


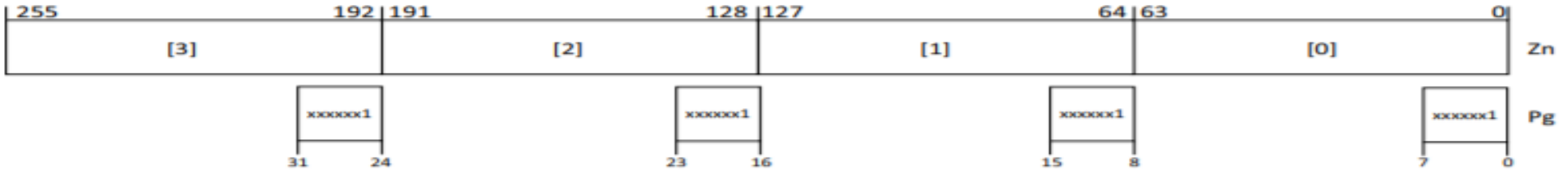




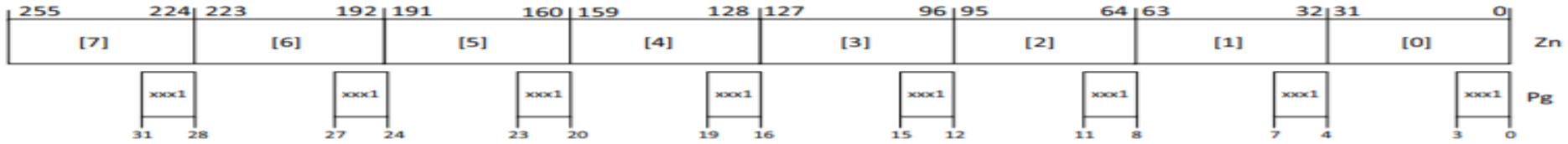
SVE简介---预测寄存器P

- SVE有16个P寄存器P0~P15，预测寄存器通常用作数据操作的位掩码，如果预测寄存器不用作位掩码，则将它们用作操作数。
- P寄存器的长度为Z寄存器的1/8，P寄存器每bit对应Z寄存器每byte。
- 支持预测的指令叫预测指令，P寄存器可以指定Zn寄存器中哪些通道是活动的，哪些是不活动的，操作指令只对活动的通道有效。
- 对于不活动的通道，预测指令有两种处理：
  - 清零，用法为p0/Z，将Zn寄存器不活动的通道清零。
  - 保留，用法为p0/M ，保留Zn寄存器不活动的通道的值。
- 预测寄存器的通道划分取决于指令指定的位宽：
  - Zn一个通道64bit，P寄存器为8bit一个通道，Zn活动状态通过P每个最低位bit判断。
  - Zn一个通道32bit，p寄存器为4bit一个通道，Zn活动状态通过P每个最低位bit判断。





Zn寄存器通道64位



Zn寄存器通道32位

SVE简介--- FFR寄存器

FFR寄存器（first fault register）：

- FFR是一个专用寄存器，其大小与格式和P寄存器相同。
- 主要用于获取向量寄存器加载时的异常和状态。
- SVE新增相关指令：如读指令：RDFFR，RDFFRS；写指令：WRFFR，SETFFR等等。
- 可用于推测性内存访问，例如strcmp判断尾部’ \0’ 。

[4] - 初始化P0寄存器，标志位全置为true。
[6] - 初始化FFR寄存器，标志位全置为true。
[7] - 将[e]内存数据加载到 Z0寄存器中，非首位元素故障时，通过FFR先记录加载故障位置，P0寄存器不会更新。
[8] - 获取FFR寄存器的内容保存到P1寄存器以记录加载失败的标志位。
[9] - 通过P1寄存器对比Z0活动元素值与0之间是否相等（即加载失败的数据不对比），并保存对比结果到P2寄存器。（元素非0时，寄存器值为false）
[10] - 将P2寄存器置为true，但第一个活动true元素及后续元素都置为false。并且在第一个true条件之前中断，设置!Last (C) 条件标志和 V 标志设置为零。
[11] - 计算x1 + 偏移量（P2活跃元素数目）并赋值给x1，作为新一轮循环的起始地址。

```c
int strlen(const char *s) {
    const char *e = s;
    while (*e) e++;
    return e - s;
}
```

(a) Strlen C code

```asm
// x0 = s
strlen:
  mov x1, x0          // e=s
.loop:
  ldrb x2, [x1], #1 // x2=*e++
  cbnz x2, .loop     // while(*e)
.done:
  sub x0, x1, x0     // e-s
  sub x0, x0, #1     // return e-s-1
  ret
```

(b) Unoptimized ARMv8 scalar strlen

```asm
// x0 = s
strlen:
  mov    x1, x0                  // e=s
  ptrue p0.b                     // p0=true
.loop:
  setffr                         // ffr=true
  ldff1b z0.b, p0/z, [x1]        // p0:z0=ldff(e)
  rdffr  p1.b, p0/z              // p0:p1=ffr
  cmpeq  p2.b, p1/z, z0.b, #0 // p1:p2=(*e==0)
  brkbs  p2.b, p1/z, p2.b        // p1:p2=until(*e==0)
  incp   x1, p2.b                // e+=popcnt(p2)
  b.last .loop                   // last=>!break
  sub    x0, x1, x0              // return e-s
  ret
```

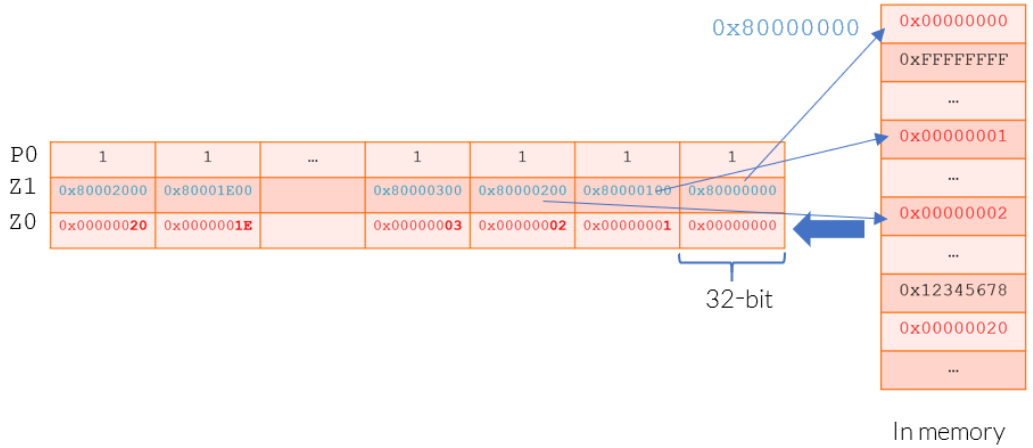
(c) Unoptimized ARMv8 SVE strlen

SVE特性简介---聚集加载和分散存储

SVE 中灵活的寻址模式允许向量基于地址或向量偏移量，这样可以从非连续内存位置加载或者存储单个Zn寄存器。

示例：从Z1寄存器活动通道的32位值为地址指向的内存，聚集加载对应的数据，并存储在Z0寄存器的活动通道中。
并将Z0未活动的通道置0。

LD1SB Z0.S, P0/Z, [Z1.S]

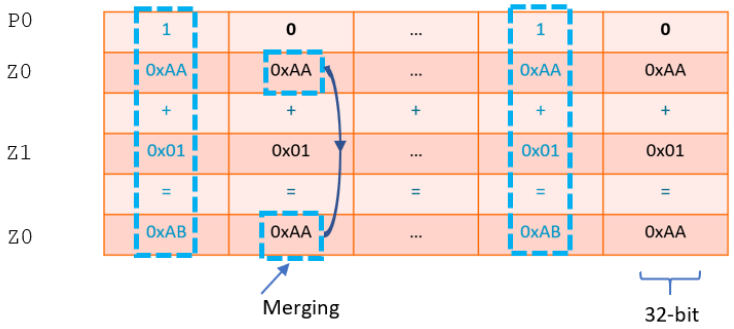


SVE特性简介---每通道预测

为了允许对选定元素进行灵活操作，SVE 和 SVE2 引入了 16 个Pg寄存器 ，以指示Zn寄存器的活动通道上的有效操作。

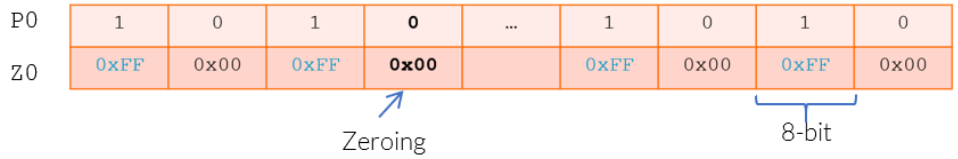
示例一：将P0寄存器指定的活动的Z1和Z2寄存器的通道元素相加，并将结果放入Z0中。 “M”表示将合并非活动元素，这意味着Z0非活动元素在ADD操作之后将保持其原始值。

ADD Z0.D, P0/M, Z1.D, Z0.D



示例二：将有符号整数0xFF复制到Z0的所有活动通道中。 “Z”表示将Z0的非活动通道元素将设置为零。

CPY Z0.B, P0/Z, #0xFF

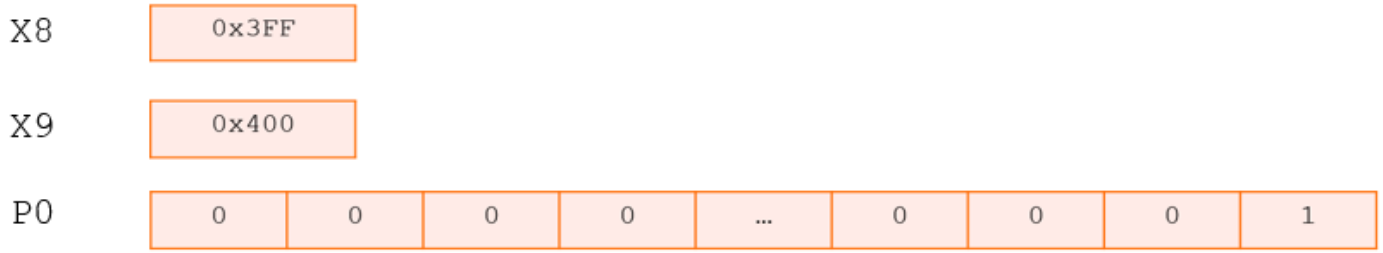


SVE特性简介---预测P寄存器驱动的循环控制

预测不仅提供了控制向量运算的各个元素的灵活性，而且还实现了预测驱动的循环控制。预测驱动的循环控制和管理使循环控制高效灵活。此功能通过在预测寄存器中注册活动和非活动元素索引，消除了处理部分向量的额外循环头和尾的开销。预测驱动的循环控制和管理意味着，在接下来的循环迭代中，只有活动元素执行预期的选项。

示例：通过第一个无符号标量X8操作数低于第二个标量操作数X9的增量值，在P0中生成一个预测，该预测从编号最低的元素开始为true，数目有x9-x8的差值，此后为false，直到编号最高的元素。

WHILELO P0.S, x8, x9

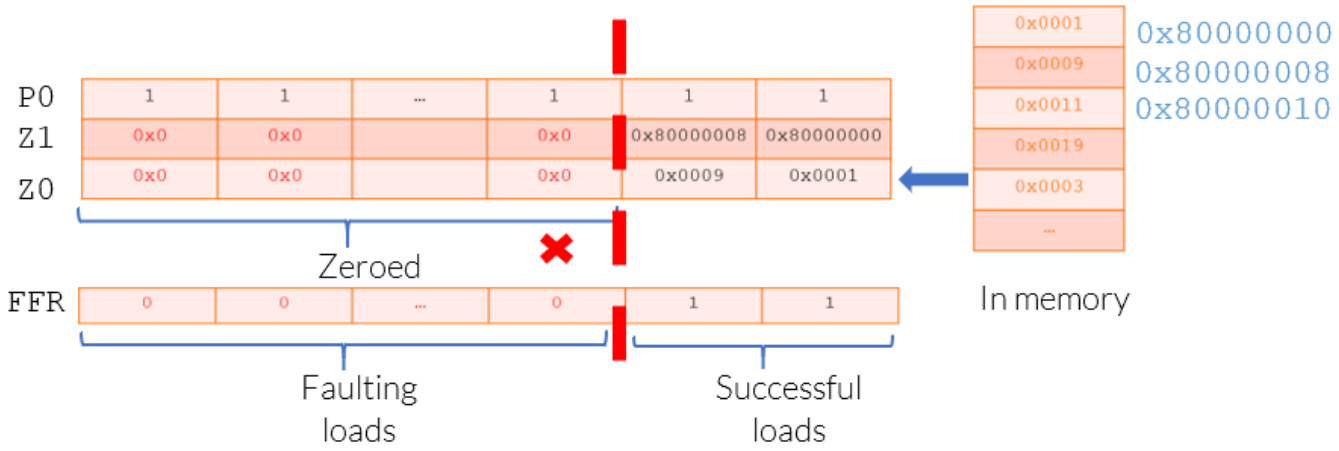


SVE特性简介---用于软件管理推测的向量分区

SVE 改进了对推测负载的 Neon 向量化限制。SVE 引入了first-fault向量加载指令，例如LDRFF，以及first-fault预测寄存器 (FFR) 以允许向量访问跨入无效页面。

示例：从基向量Z1加0生成的内存地址指向的数据，以双字节的首次故障行为聚集加载到Z0的活动元素。非活动元素不会读取设备内存或信号故障，并在目标向量中设置为零。成功加载到有效内存将为first-fault寄存器（FFR）设置true，第一个故障加载将为FFR中的相应元素和其余元素设置false。

LDFF1D Z0.D, P0/Z, [Z1.D, #0]

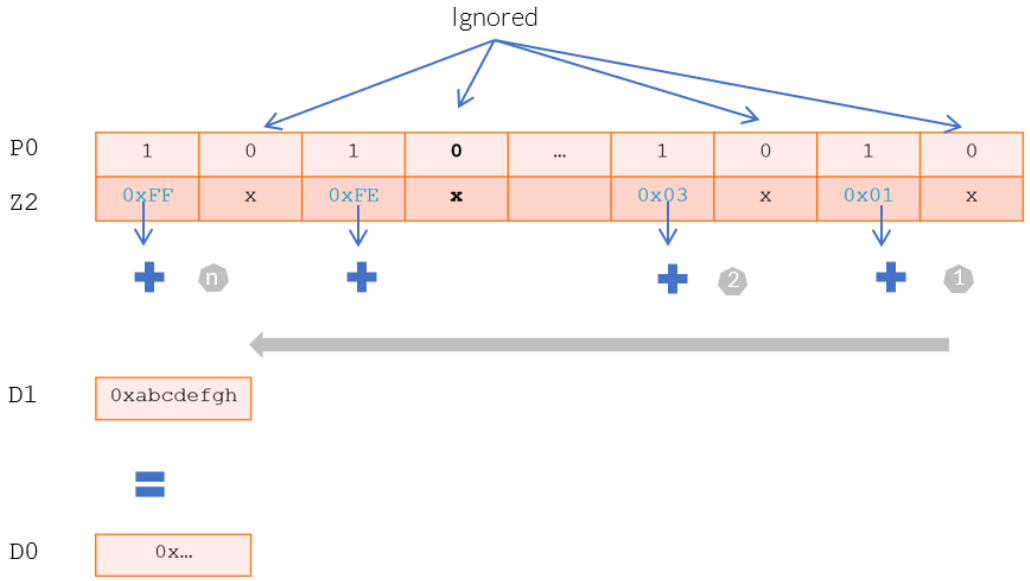


SVE特性简介---浮点按位水平归约

为了在向量中实现高效的归约操作，并满足对精度的不同要求，SVE 增强了浮点和水平归约操作。指令可能具有按顺序（从低到高）或基于树（成对）的浮点归约排序，其中操作排序可能会导致不同的舍入结果。这些操作权衡可重复性和性能。

示例：将D1和Z2.D的所有活动元素相加，并将结果放置在标量寄存器D0中。向量元素严格按照从低到高的顺序处理（即向量中活动元素的累加），标量源D1提供初始值。源向量中的非活动元素将被忽略。而FADDV将执行不包含标量源初始值的重复累加，并将结果放入标量寄存器中。

FADDA D0, P0/M, D1, Z2.D



SVE C/C++扩展开发

SVE 内在函数是编译器支持的函数，可以用相应的指令替换。程序员可以直接调用C、C++等高级语言的指令函数。SVE 的 ACLE（Arm C 语言扩展）定义了哪些 SVE 指令函数可用、它们的参数以及它们的作用。支持 ACLE 的编译器可以在编译期间用映射的 SVE 指令替换内在函数。要使用 ACLE 内在函数，必须包含头文件“arm_sve.h”，其中包含可在 C/C++ 中使用的向量类型和指令函数（针对 SVE）的列表。

SVE的Zn数据向量类型如下：

- svint8_t svuint8_t
- svint16_t svuint16_t svfloat16_t
- svint32_t svuint32_t svfloat32_t
- svint64_t svuint64_t svfloat64_t

比如svint64_t 表示 64 位有符号整数的向量，svfloat16_t 表示半精度浮点数的向量。

SVE 的 ACLE还定义了一个无大小的P寄存器预测类型svbool_t，它有足够的位来控制对字节向量的操作。
预测的主要用途是选择向量中的活动元素。当向量中的元素有N个字节时，只有N个预测位序列中的低位是有效的，如下表所示：

Vector type	Element selected by each svbool_t bit									
svint8_t	0	1	2	3	4	5	6	7	8	...
svint16_t	0		1		2		3		4	...
svint32_t	0				1				2	...
svint64_t	0								1	...

SVE C/C++扩展开发---优势

SVE指令开发的优势：

- 代码更加简洁，数据长度即使不是一次循环处理的整倍数，也不需要在单独处理。
- 在处理大量数据的场景，增加了计算的整体性能。
- 稀疏数据计算性能更加突出。
- 向量中数据，执行带有条件语句的能力有很大改善。
- ……

ACLE函数接口文档下载：
https://developer.arm.com/documentation/100987/latest/

SVE C/C++扩展开发---函数形式

svbase[_disambiguator][_type0][_type1]...[_predication]

SVE函数都是以sv开头，后接指令名称base。
base名对于大多数函数，这是SVE指令的小写名称。有时，指示正在操作的数据类型或大小的字母会被删除，在这些字母中可以从参数类型中暗示。
无符号扩展加载添加一个u，以指示数据将为零扩展，以更明确地将它们与有符号的等效项区分开来。

disambiguator该字段区分函数的不同模式，例如：

- offset区分寻址模式
- index区分以标量而不是向量作为最终参数的模式

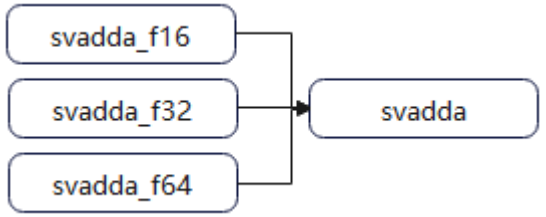
type0 type1 ...向量和预测的类型列表，从返回类型开始，然后从每个参数类型开始：

- 向量类型，_s8、_u32和_f32，它们分别表示有符号8位整数、无符号32位整数和单精度32位浮点类型等等。
- 预测类型由_b8、_b16等表示，分别适用于8位和16位类型的预测。适用于所有元素类型的预测类型由_b表示。如果不需要类型来消除base函数的变体之间的歧义，则省略它。

predication此后缀描述了预测操作结果中的非活动元素。它可以是以下之一：

- z - 零预测：将结果的所有非活动元素设置为零。
- m - 合并预测：从第一个向量参数中复制所有非活动元素。
- x - “不在乎”预测。当不关心非活动元素时，使用此后缀。

编译器可以自由地在归零、合并或非预测形式之间做出选择，以提供最佳的代码质量，但不能保证哪些数据将留在非活动元素中。同时，[_type]可以舍弃，由编译器自行匹配。



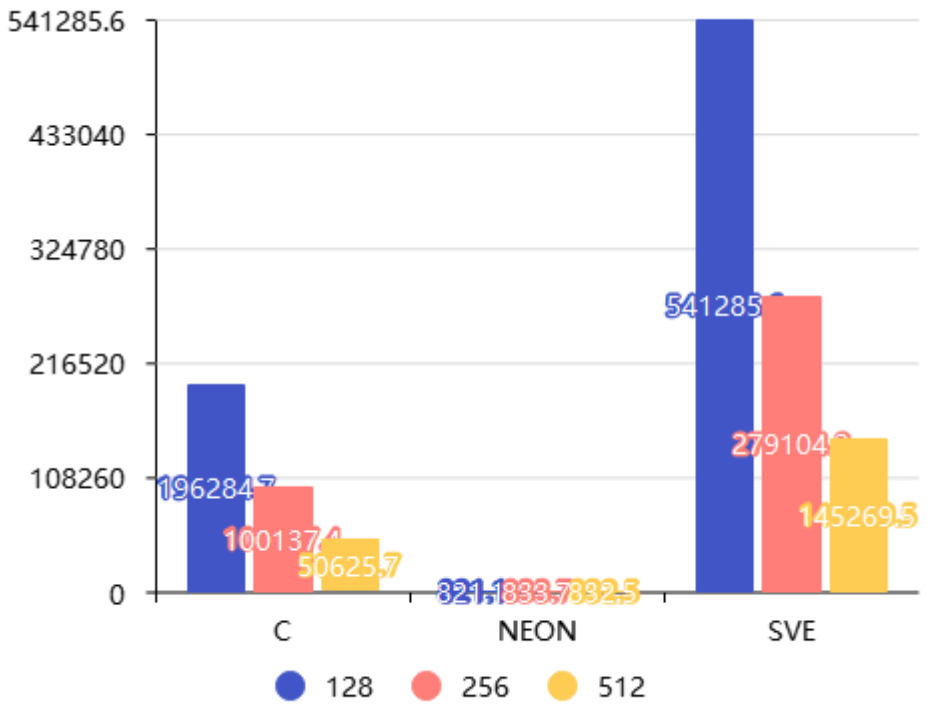
SVE C/C++扩展开发---示例

```
double AddReductionSVE(int64_t n, double da, double *dx) {
    int64_t i = 0;
    double res = da;
    svbool_t pg = svwhilelt_b64(i, n); // [1]
    do {
        svfloat64_t vsrc = svld1 (pg, dx + i); // [2]

        res = svadda(pg, res, vsrc); // [3]


        pg = svwhilelt_b64(i += svcntd(), n); // [4]
    } while (svptest_any(svptrue_b64(), pg)); // [5]
    return res;
}
```

从NEON到SVE更多示例：
https://developer.arm.com/documentation/101726/0400/Coding-for-Scalable-Vector-Extension--SVE-/Migrate-from-Neon-to-SVE/Part-4--Migrate-your-Neon-code-to-SVE/

[1] - 初始化一个预测寄存器来控制循环。_b64指定 64 位元素的预测。从概念上讲，此操作创建一个整数向量，该向量从i1 开始并在每个后续通道中递增。如果该值小于n，则预测通道处于活动状态。因此，即使n≤ 0，即使效率低下，此循环也是安全的。在循环底部使用相同的操作来更新下一次迭代的预测。
[2] - 将[dx + i]内存数据加载到 SVE 向量中，由循环预测保护。此预测为假的通道不执行任何加载（因此不会产生故障），并将结果值设置为 0.0。加载的通道数取决于向量宽度，这仅在运行时才知道。
[3] - 从浮点种子值res开始，依次与向量的每个活动元素执行加法，按索引顺序进行。如果没有元素处于活动状态，则结果是种子值。
[4] - 重新计算索引i，并重新计算预测向量通道的数量。
[5] - ptest如果更新的预测向量的任何通道处于活动状态，则返回 true，如果还有任何工作要做，这将导致控制返回到 while 循环的起始。

SVE C/C++扩展开发---仿真性能



基于鲲鹏服务器920仿真ARMIE环境测试HMPPS_Add_32f接口的性能结果图示

HmppResult HMPPS_Add_32f(const float *src1, const float *src2, float *dst, int32_t len)；

鲲鹏 BoostKit 加速库向量化计算的应用

鲲鹏BoostKit加速库提供基于ARM指令深度优化和基于鲲鹏KAE（鲲鹏硬件加速引擎）开发的加速库，覆盖系统库、压缩、加解密、媒体、数学库、存储、网络等7类加速库，为大数据加解密、分布式存储压缩、视频转码等应用场景提供高性能加速。

鲲鹏 BoostKit 加速库---HMPP

软件介绍

鲲鹏超媒体性能库HMPP（Hyper Media Performance Primitives）包括向量缓冲区的分配与释放、向量初始化、向量数学运算与统计学运算、向量采样与向量变换、滤波函数、变换函数（快速傅里叶变换），支持IEEE 754浮点数运算标准，支持鲲鹏平台下使用。

应用场景

HMPP涉及信号处理、图像处理、颜色转换、滤波、变换、数学运算，为计算机视觉运算、向量运算、统计、信号滤波、信号变换和固定精度运算等提供了丰富的功能接口和极致性能优化。

HMPP提供的接口可分为4类，HMPP（基础函数）、HMPPI（图像库）、HMPPS（信号库）或HMPPA（音频库）。

可适用于如下相关领域：

- 数字媒体
- 数据通信
- 生物医学
- 航空航天

SVE开发仿真环境及相关资源

OpenEuler

编译器

Arm GNU 工具版本 8.0+ 支持 SVE指令，gcc10.0+支持ACLE。
Arm GUN工具链官方说明：
https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain
gcc开源地址：
https://github.com/gcc-mirror/gcc

指令仿真器ArmIE

目前支持SVE指令的真机并未普及，SVE指令仿真器可以提供模拟开发和执行环境。
ArmIE官方说明：
https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator

OpenEuler

我参与 我做主

下载体验

您可以通过以下四种方式体验openEuler操作系统

公有云　虚拟机　硬件　树莓派

加入贡献

请根据您的参与身份，选择签署：

个人CLA　企业CLA　员工CLA

浅谈Fugaku HPL-AI benchmark实现技术研讨

华为

付姚姚

目录

HPC科学计算的基本方法

OpenEuler

科学研究的基本目的

- 寻求基本规律

例如：行星运动的三大定律，量子力学基本方程

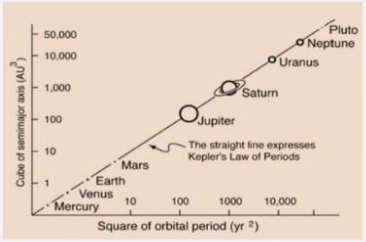
- 解决实际问题

工程学科，制造行业，材料，航天航空，……

科学研究的基本方法 I

- **开普勒范式**：从数据中直接总结出规律并解决实际问题。

例如：行星运动的第三定律



- 最成功的案例：生物信息学 (人类基因组工程)
- 最成功的工具：统计方法、机器学习、计算方法

开普勒模式： 高效，但知其然不知其所以然

牛顿模式：深刻，但难以用来解决实际问题

科学研究的基本方法 II

- **牛顿范式**：寻求基本原理并用以解决实际问题。

例如：行星运动的第三定律

1. 牛顿第二定律：加速度与力（引力）成正比
2. 万有引力定律：引力和距离的平方成反比

行星运动规律变成一个数学（微分方程）问题！

解微分方程，得到第三定律。

- 最成功的例子： 物理学（成为自然科学，工程科学的基础）
  - 牛顿力学 (牛顿方程)
  - 电磁场理论（Maxwell 方程组）
  - 量子力学（Schrodinger 方程）

把物理（科学）问题变成数学问题

- 最成功的工具：微分方程

其它基本原理的例子：

- 空气动力学（Euler 方程）
- 流体力学（Navier-Stokes 方程）
- 弹性力学（Lame 方程等）
- 电磁场问题（Maxwell 方程）

有效数学方法出现之前，科学家们为解决实际问题，唯一能够做的事情就是简化模型。比方说Thomas-Fermi密度泛函理论

薛定谔方程：

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V\psi = E\psi$$

- 多体（高维）问题： 维数= 3 x 粒子的个数
- 对于电子体系，波函数必须是反对称函数

第一个时代（50年代到现在）：电子计算机+数值方法

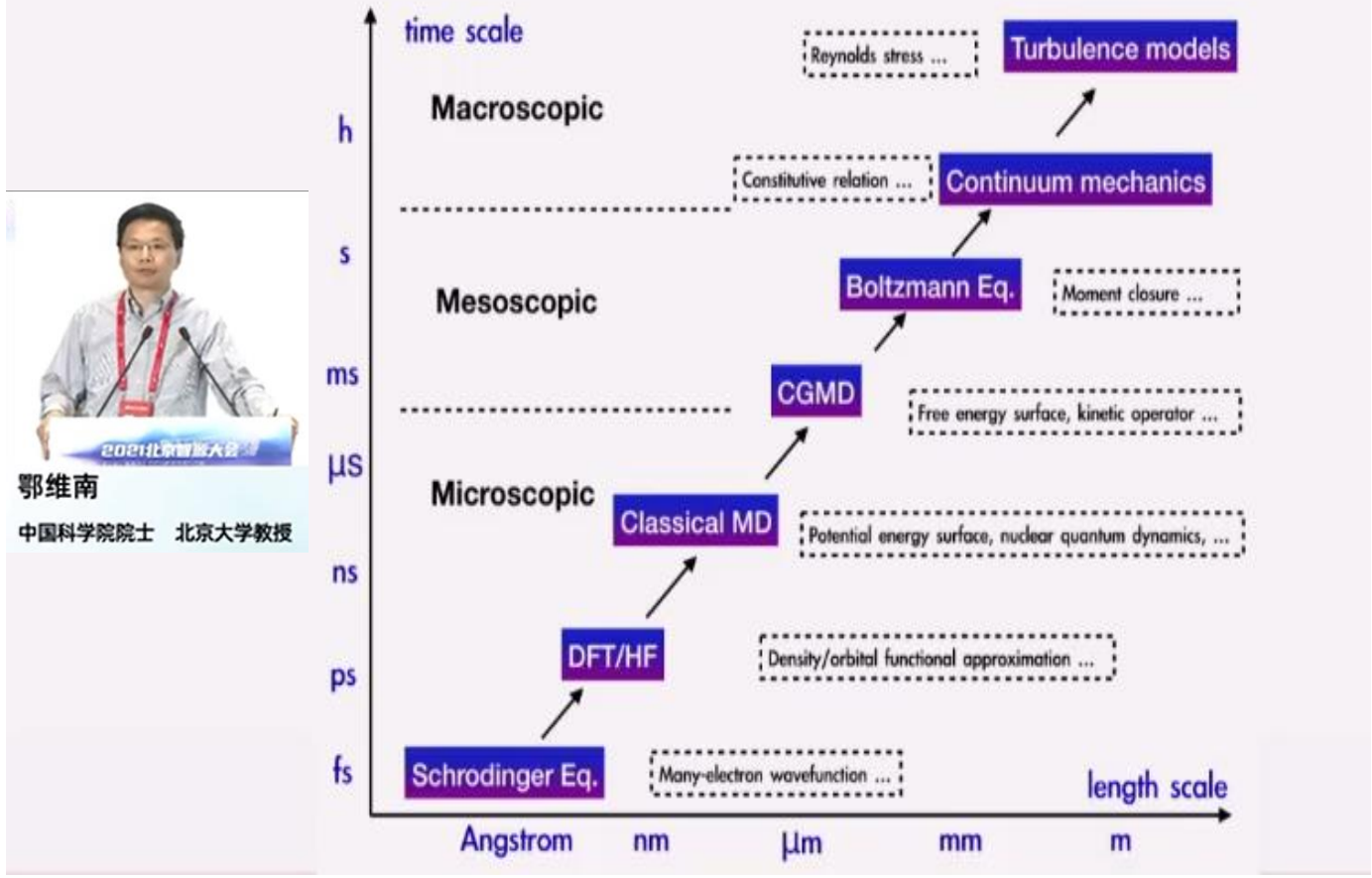
- 差分方法，有限元方法，谱方法等
- 第一次大规模地实现了直接用基本原理解决实际问题
- 基本出发点：（分片）多项式可以有效逼近一般函数

工具1：多项式　工具2：多尺度　工具3：机器学习

HPC科学计算的经典物理方程

OpenEuler

科学计算（物理模型）的生态链



鄂维南

中国科学院院士　北京大学教授

例子1: 高维控制问题

控制论面临的困境：
（基于基本原理的）理论研究和实际问题距离太远

其结果：
为解决实际问题，必须作各种近似（放弃基本原理）

一个重要观察：控制问题和深度学习问题之间有着惊人的相似 (Han and E, 2016)

用深度学习方法解决（高维）控制论问题

用控制论方法解决深度学习问题

1. 传统的科研领域应该成为人工智能的主战场！

化学、材料、电子工程、化学工程、机械工程等

➢全面提升科研能力，推动尽快进入“智能化科研”时代

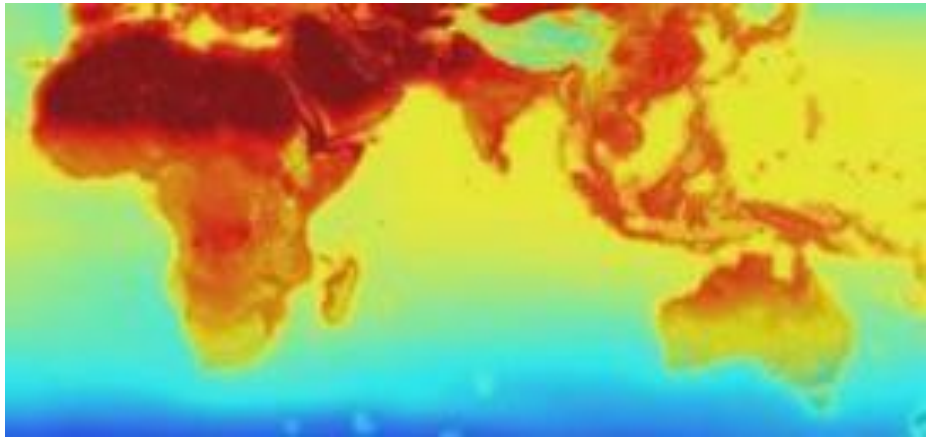
➢推动对当下工业和技术的升级

2. 人工智能算法和模型的基础研究

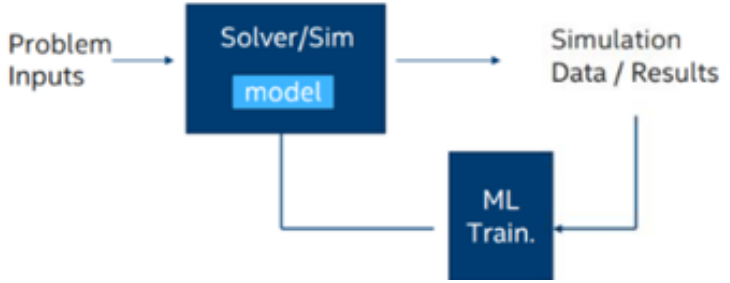
- **共同的基础：**机器学习
- **重要主题的数学模型：**
  - 图像
  - 自然语言处理
  - 机器人

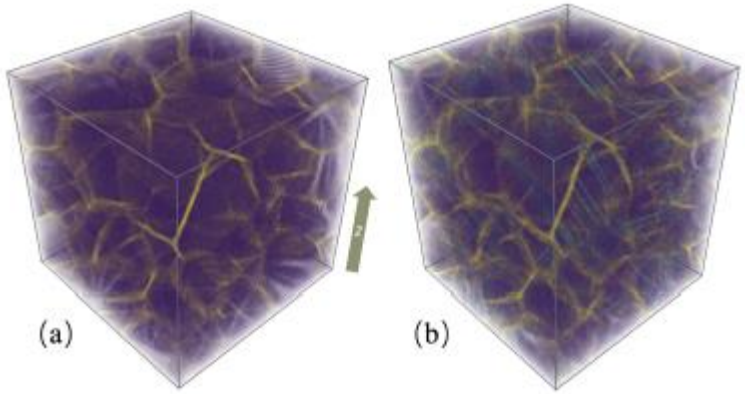
问题背景

- 当前典型HPC问题：
  - ✓ 大气物理：极端气象模拟
  - ✓ 凝聚态物理中最核心和最具挑战性的问题：量子多体模拟
- Gordon Bell Prize(2018)：Exascale Deep Learning for Climate Analytics
  -- Kurth T, Treichler S, Romero J, et al. IEEE, 2018: 649-660.



- 面对未来E级计算的挑战
  - ✓ 第一性原理计算 → 维数灾难 → 算力挑战
  - ✓ 数据表示 → 物理法则 → AI的理论边界
- Gordon Bell Prize(2020)：Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning.
  -- Jia W, Wang H, Chen M, et al. IEEE, 2020: 1-14.





目录

数学背景知识

- 线性方程组：Ax=b；A为m*n阶矩阵，x为n阶列向量，b为m阶列向量
- LU分解：A=LU --> 回代法求解线性方程组： $\mathrm{LUx}=\mathrm{b}\begin{cases} Ly=b \\ Ux=y \end{cases}$

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{L_0} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{L_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{L_2} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

$$A \qquad L_0A \qquad L_1L_0A \qquad L_2L_1L_0A=U$$

其中

$$L_j=\begin{bmatrix} 1 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & 0 & \dots & 0 \\ 0 & \dots & -A_{j+1,j}/A_{jj} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & -A_{n,j}/A_{jj} & 0 & \dots & 1 \end{bmatrix}$$

整理 $U=L_2L_1L_0A$ 可得:

$$A=LU$$
$$L=L_0^{-1}L_1^{-1}L_2^{-1}$$

- 部分选主元的LU分解：

$$PA=LU,\ where\ P=\prod_{i=0}^{n-2}P_i$$

其中P_i为置换矩阵

- 浮点计算次数：

$$N_{float}=\left(\frac{2*N^3}{3}-\frac{1*N^2}{2}\right)_{LU}+(2*N^2)_{回代}$$

数学背景知识

- 线性方程组：Ax=b；A为m*n阶矩阵，x为n阶列向量，b为m阶列向量
- GMRES子空间迭代法：求解非对称线性方程组最常用的算法之一

(1) 令 $m = 1$

(2) 定义 Krylov 子空间 $\mathcal{K}_m(A, r_0)$;

(3) 找出仿射空间 $x^{(0)} + \mathcal{K}_m$ 中的“最佳近似”解;

(4) 如果这个近似解满足精度要求, 则迭代结束;
否则令 $m \leftarrow m + 1$, 返回第 (2) 步.

“最佳近似”解的判别方法为 使得 $\|r_m\|_2 = \|b - Ax^{(m)}\|_2$ 最小

对任意向量 $x \in x^{(0)} + \mathcal{K}_m$, 可设 $x = x^{(0)} + V_m y$, 其中 $y \in \mathbb{R}^m$. 于是

$$r = b - Ax = r_0 - AV_m y = V_{m+1}\left(\beta e_1 - H_{m+1,m} y\right),$$

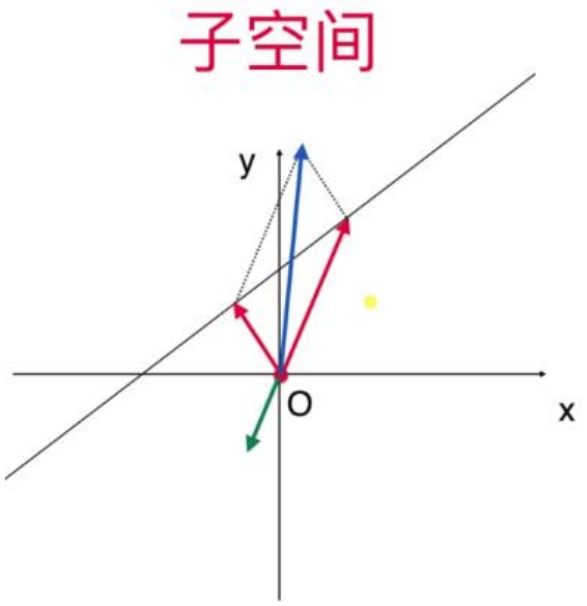
这里 $\beta = \|r_0\|_2$. 由于 V_{m+1} 列正交, 所以

$$\|r\|_2 = \|V_{m+1}(\beta e_1 - H_{m+1,m} y)\|_2 = \|\beta e_1 - H_{m+1,m} y\|_2.$$

于是最优性条件就转化为

$$y^{(m)} = \arg\min_{y \in \mathbb{R}^m} \|\beta e_1 - H_{m+1,m} y\|_2.$$

用基于 Givens 变换的 QR 分解来求解即可.



迭代法：CG、GCR、GMRES、Jacobi、AMG、SOR等

HPL介绍

- High Performance Linpack (HPL)是一种在分布式内存计算机上解决双精度（64位）算法的（随机）密集线性系统的软件包，HPL是评测高性能计算机性能重要benchmark，TOP500就是依据HPL实测性能进行排名
- HPL是求解N维的线性方程组Ax=b的程序，通过列主元高斯消元实现增广矩阵A的LU分解(双精度)，并回代最终求解x

$$\mathrm{Ax} = \mathrm{b} \;\; -> \;\; \mathrm{LUx} = \mathrm{Pb} \quad \begin{cases} Ly = \mathrm{P}b \\ Ux = y \end{cases}$$

优化人员可以根据硬件架构调整HPL问题规模、代码实现

Hpl-harness后向误差：

$$\frac{\|Ax-b\|_\infty}{\|A\|_\infty\|x\|_\infty + \|b\|_\infty} \times (n \times \epsilon)^{-1},$$

其中Double类型机器精度：
$\varepsilon = 2^{-53} = 1.1 * 10^{-16}$，且后向误差小于16即认为收敛

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

HPL实测浮点峰值 = $(\frac{2}{3}n^3 + \frac{3}{2}n^2)$ /（整个系统求解的总时间）* 10^{-9} GFLOPS
理论浮点峰值 = CPU主频×CPU每个时钟周期执行浮点运算次数×CPU数量

鲲鹏920理论峰值计算：
1、48核：　2.6 GHz * 4 FLOPs/cycle * 2 * 48cores = 998.4 GFLOPS
2、64核：　2.6 GHz * 4 FLOPs/cycle * 2 * 64cores = 1331.2 GFLOPS

```
The following parameter values will be used:

N      :  233984
NB     :     256
PMAP   : Row-major process mapping
P      :       1
Q      :       2
PFACT  :   Right
NBMIN  :       4
NDIV   :       2
RFACT  :   Crout
BCAST  :  1ringM
DEPTH  :       1
SWAP   : Mix (threshold = 64)
L1     : transposed form
U      : transposed form
EQUIL  : yes
ALIGN  : 8 double precision words

--------------------------------------------------------------------------------

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be           1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11C2R4      233984   256     1     2           19602.63              4.3567e+02
HPL_pdgesv() start time Fri Mar 27 15:24:39 2020

HPL_pdgesv() end time   Fri Mar 27 20:51:22 2020

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        3.88296473e-03 ..... PASSED
================================================================================
```

HPL-AI介绍

- HPL-AI benchmark分两部分：LU分解和迭代细化求解过程（IR过程：GMRES使用LU分解因子作为预条件子）
- LU分解浮点理论计算量$\frac{2}{3}n^3+\frac{3}{2}n^2$，但可以使用任何精度；IR迭代求解线性方程组要求具有64位精度
- 后向误差最后必须小于16，性能的计算=($\frac{2}{3}n^3+\frac{3}{2}n^2$) / （整个系统求解的总时间）

RESULTS

HPL-AI

Since 2019, various supercomputing installations started reporting performance results bi-annually. Below are the currently available results.

November 2021

Rank	Site	Computer	Cores	HPL-AI (Eflop/s)	TOP500 Rank	HPL Rmax (Eflop/s)	Speedup
1	RIKEN	Fugaku	7,630,848	2.000	1	0.4420	4.5
2	DOE/SC/ORNL	Summit	2,414,592	1.411	2	0.1486	9.5
3	NVIDIA	Selene	555,520	0.630	6	0.0630	9.9
4	DOE/SC/LBNL	Perlmutter	761,856	0.590	5	0.0709	8.3
5	FZJ	JUWELS BM	449,280	0.470	8	0.0440	10.0
6	University of Florida	HiPerGator	138,880	0.170	31	0.0170	9.9
7	SberCloud	Christofari Neo	98,208	0.123	44	0.0120	10.3
8	DOE/SC/ANL	Polaris	259,840	0.114	13	0.0238	4.8
9	ITC	Wisteria	368,640	0.100	18	0.0220	4.5
10	NSC	Berzelius	59,520	0.050	95	0.0053	9.5
11	Nagoya	Flow Type I	110,592	0.030	74	0.0066	4.5
12	NVIDIA	Tethys	19,840	0.024	297	0.0023	10.8
13	NVIDIA	DGX Saturn V	87,040	0.022	118	0.0040	5.5

TOP500 The List.

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70,870.0	93,750.0	2,589
6	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
7	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482

0.442EFlops

HPL-AI计算流程

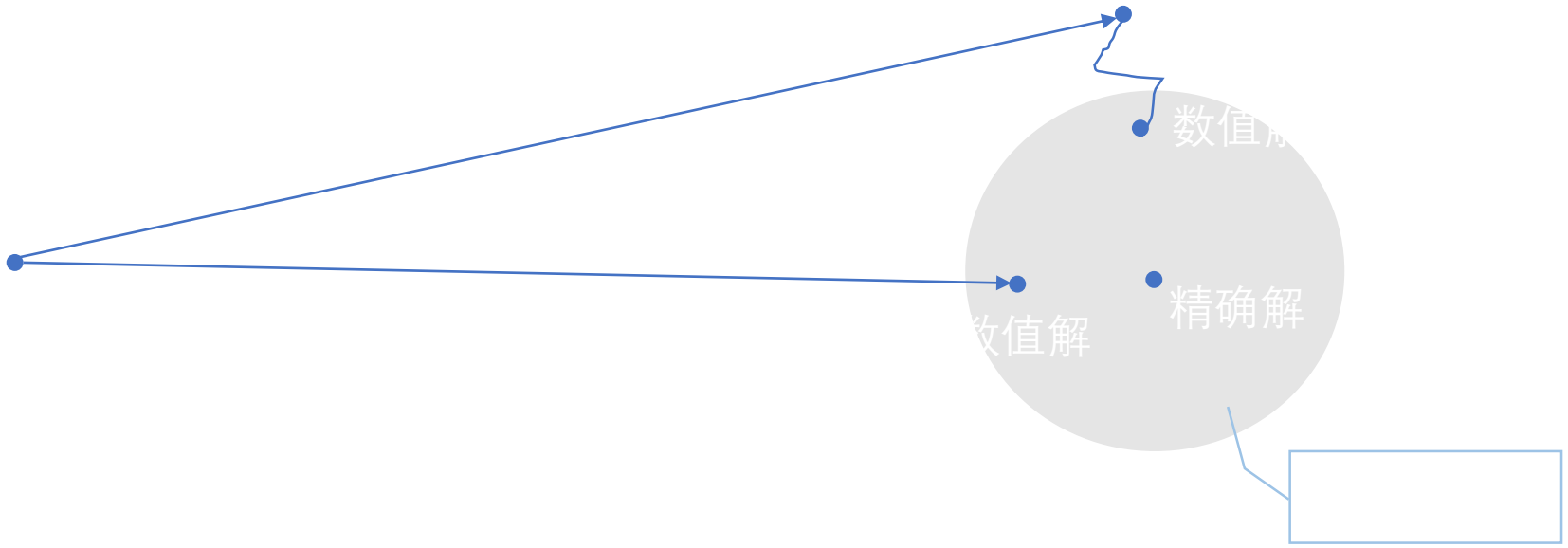
- 官方标准的HPL-AI计算流程及程序运行结果示意图

官方标准HPL-AI计算流程图

```
[root@localhost hpl-ai]# ./hpl-ai 1000 2
================================================================================
                        HPL-AI Mixed-Precision Benchmark
      Written by Yaohung Mike Tsai, Innovative Computing Laboratory, UTK
================================================================================

This is a reference implementation with the matrix generator, an example
mixed-precision solver with LU factorization in single and GMRES in double,
as well as the scaled residual check.
Please visit http://www.icl.utk.edu/research/hpl-ai for more details.

Time spent in conversion to single:        0.003 second
Time spent in factorization       :        0.324 second
Time spent in solve               :        0.001 second
Time spent in conversion to double:        0.003 second
Residual norm at the beginning of GMRES: 3.448479e-09
Estimated residual norm at the 1-th iteration of GMRES: 2.198013e-15
Estimated residual norm at the 2-th iteration of GMRES: 2.002036e-21
Time spent in GMRES               :        0.013 second
Total time                        :        0.341 second
Effective operation per sec       :     1.958859 GFLOPs
The following scaled residual check will be computed:
||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
The relative machine precision (eps) is taken to be: 1.110223e-16
Computational tests pass if scaled residuals are less than 16.0
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 0.011570 ...PASSED
```

HPL-AI vs HPL

- 目的：利用LU分解求解稠密线性方程组Ax=b，计算浮点性能



目录

富岳(Fugaku)超级计算机介绍

- 富士通与日本理化学研究所共同开发的超级电脑，由400台计算机组成，采用富士通A64FX 48C 2.2GHz处理器，单节点FP64、FP16峰值性能3.07TFlops、12.28TFlops；富岳有158976个节点（共7630848核），内存5087232GB
- Tofu interconnect D是嵌入在A64FX微处理器中的网络接口，提供高带宽（每节点带宽40.8GB/s）和低时延，具有硬件卸载通信功能

Architecture Features

- Armv8.2-A (AArch64 only)
- SVE 512-bit wide SIMD
- 48 computing cores + 4 assistant cores*

*All the cores are identical

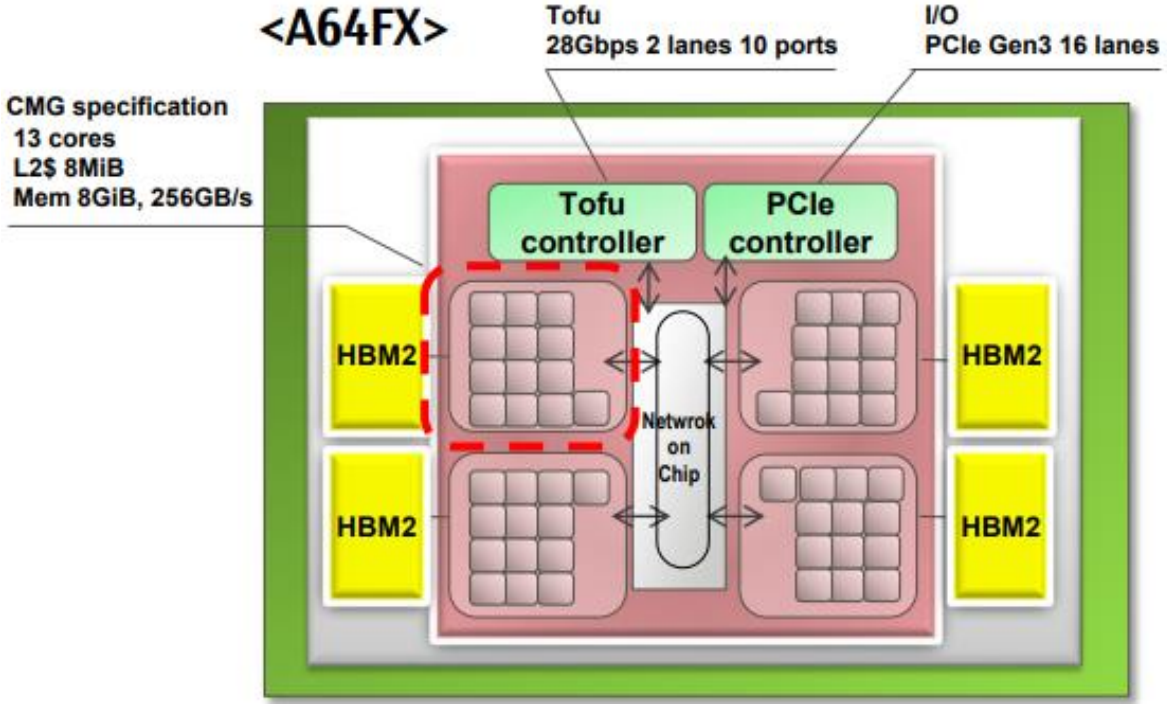
- HBM2 32GiB
- Tofu 6D Mesh/Torus
  28Gbps x 2 lanes x 10 ports
- PCIe Gen3 16 lanes

7nm FinFET

- 8,786M transistors
- 594 package signal pins

Peak Performance (Efficiency)

- >2.7TFLOPS (>90%@DGEMM)
- Memory B/W 1024GB/s (>80%@Stream Triad)



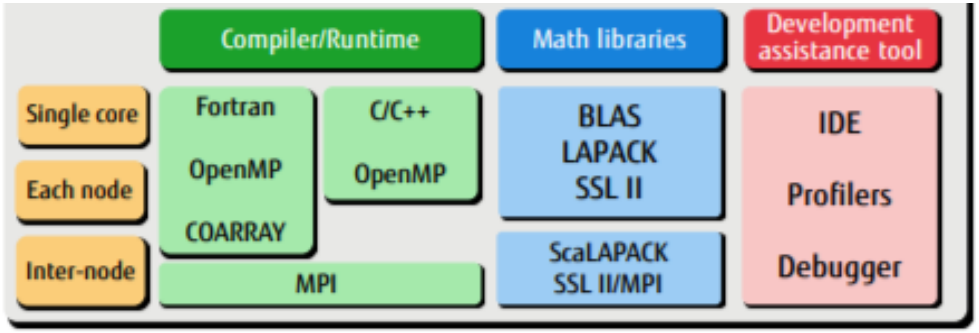
富岳(Fugaku)超级计算机介绍

- 全球超级计算机Top500榜单连续4次蝉联第一，LINPACK benchmark达到理论峰值的82.3%（0.442EFlops）
- 2021年度“戈登贝尔奖新冠特别奖”被授予一个由6人组成的日本团队，他们因在“富岳”超级计算机上的“飞沫/气溶胶感染风险评估的数字化模拟”项目荣获这一殊荣

Performance	
Linpack Performance (Rmax)	442,010 TFlop/s
Theoretical Peak (Rpeak)	537,212 TFlop/s
Nmax	21,288,960
HPCG [TFlop/s]	16,004.5
Power Consumption	
Power:	29,899.23 kW (Optimized: **26248.36** kW)
Power Measurement Level:	2
Software	
Operating System:	Red Hat Enterprise Linux
Compiler:	FUJITSU Software Technical Computing Suite V4.0
Math Library:	FUJITSU Software Technical Computing Suite V4.0
MPI:	FUJITSU Software Technical Computing Suite V4.0

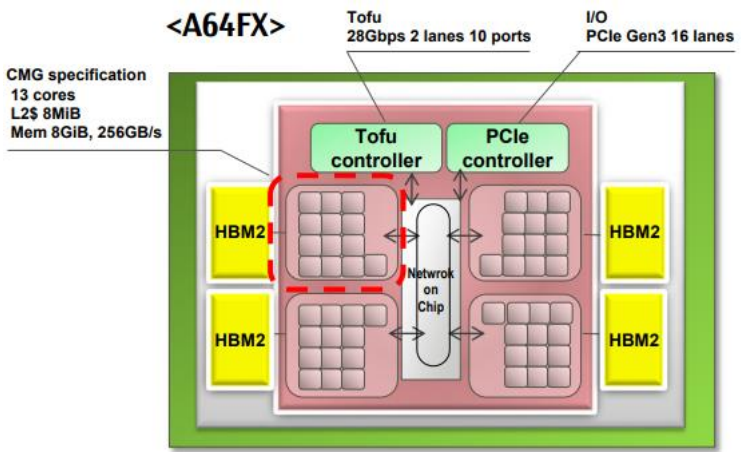
RANKING

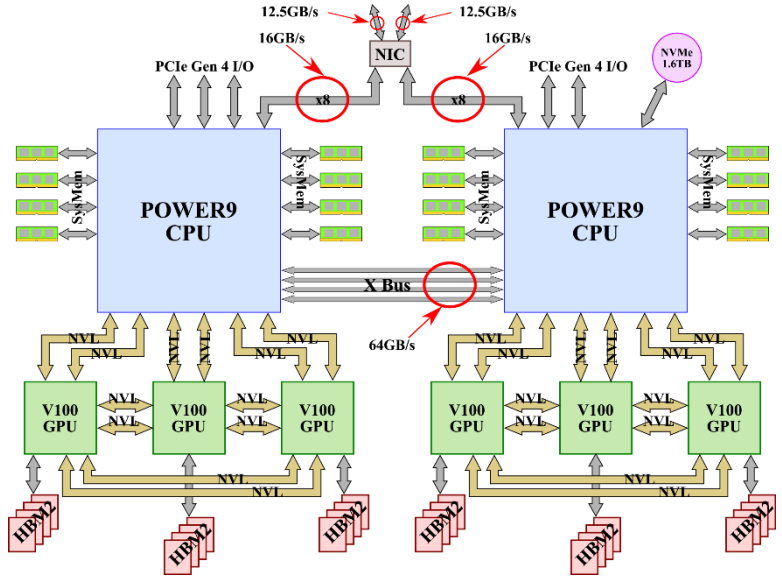
List	Rank	System	Vendor	Total Cores	Rmax (TFlops)	Rpeak (TFlops)	Power (kW)
11/2021	1	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Fujitsu	7,630,848	442,010.0	537,212.0	29,899.23
06/2021	1	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Fujitsu	7,630,848	442,010.0	537,212.0	29,899.23
11/2020	1	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Fujitsu	7,630,848	442,010.0	537,212.0	29,899.23
06/2020	1	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Fujitsu	7,299,072	415,530.0	513,854.7	28,334.50



Fugaku vs Summit

名称	Fuguku	Summit
架构	A64X 48C 2.2GHz	IBM POWER9 22C 3.07GHz + V100
节点数	158976节点 (48CPU)	4608节点(2CPU +6V100)
核数	7630848核	2414592核
内存	5087232GB	2801664
带宽	40.8GBs带宽	64GBs带宽（CPU之间）50GBs（CPU与GPU之间）
双精度浮点性能	3.07TFlops	19.5TFlops
半精度浮点性能	12.28TFlops	312TFlops
HPL-AI benchmark	2.0EFlops	1.411EFlops
LINPACK benchmark	0.4420EFlops	0.1486EFlops





HPL-AI矩阵

- HPL-AI矩阵：对角占优，a_{ii}为第i行其他元素的绝对值之和， a_{ij} 为伪随机数生成

性质

1、对于严格对角占优矩阵，矩阵可逆且LU分解的增长因子很小
2、由于元素的随机性，HPL-AI矩阵基本都是严格对角占优矩阵，且生长因子和严格对角占优矩阵一样小；
3、HPL-AI中LU分解不需要选主元；只需要做正常的Gauss消去即可

HPL-AI矩阵对角线元素是独立同分布的随机变量求和，利用中心极限定理，随着n增加，分布收敛到正态分布（$\frac{n-1}{4}$，$\frac{n-1}{24}$），该分布的取值范围涵盖$O\left(\frac{1}{n}\right)$~$O(n)$，对于中等矩阵大小n = 2^{16}=65536，FP16(6e-8~65504)涵盖范围太小，从而需要在计算中引入缩放等其他技术

$$a_{i,j} := \begin{cases} \sum_{k \neq j} |a_{i,k}|, & \text{if } i = j \\ \text{drandlcg}(n(j-1)+i)), & \text{otherwise} \end{cases}$$

```
uint64_t lcg(uint64_t n){
    if(n==0u) return 1;
    else return lcg(n-1)*6364136223846793005+1;
}
double drandlcg(uint64_t n){
    return 0x1.fffffffffffffP-65*(int64_t)lcg(n-1)
}
```

Fig. 1. A slow, recursive code for generating linear congruent generator (LCG) pseudo-random numbers in the range $[-0.5, 0.5)$. We ignored the small bias $\leq 2^{-65}$ to $-\infty$ for performance.

$$\rho_W^{(k)} = \frac{\max_{i,j} |a_{ij}^{(k)}|}{\max_{i,j} |a_{ij}^{(1)}|}, \quad \rho_W = \max_k \rho_W^{(k)}.$$

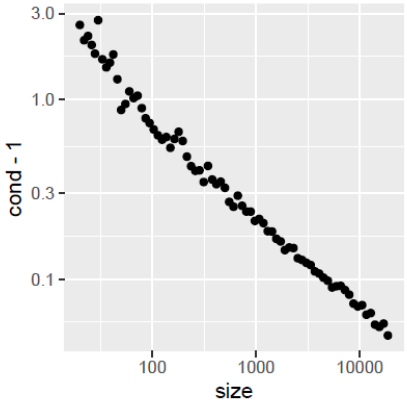
$$A^{(1)} = L_1 L_2 \cdots L_k A^{(k+1)} = L_1 L_2 \cdots L_{n-1} A^{(n)} = LU,$$

HPL-AI矩阵的近视分解技术

- HPL-AI矩阵：严格对角占优
- 2种快速近似LU分解：A = D + S + R;

其中D是对角矩阵, S为严格下三角矩阵，R为严格上三角矩阵

$$\kappa_2(A) \le \frac{|\sigma_1(D) + \sigma_1(S+R)|}{|\sigma_n(D) - \sigma_1(S+R)|},$$

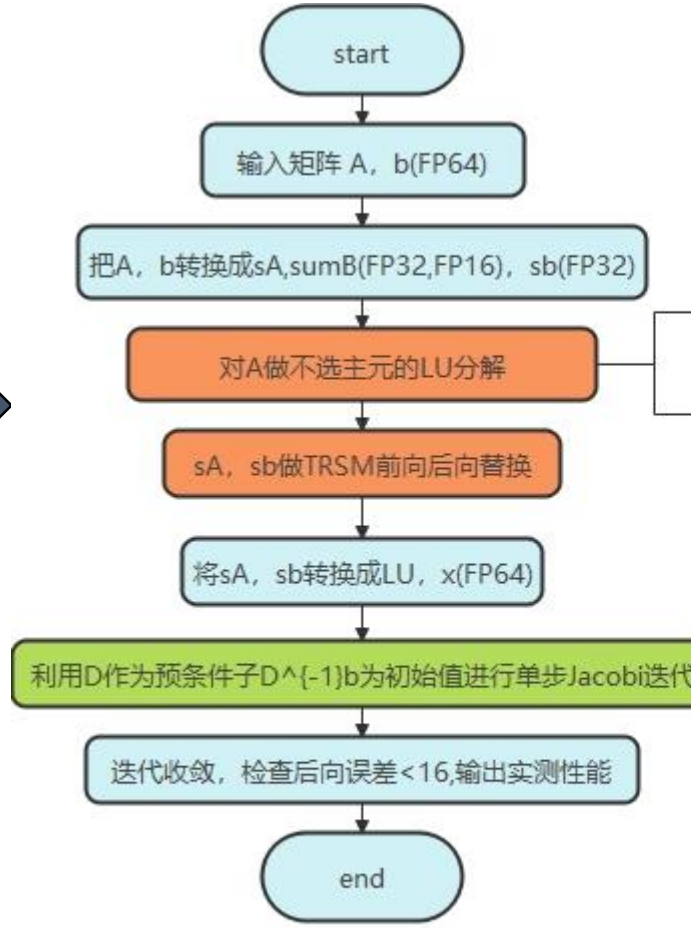


快速LU名称	Jacobi近似分解	O1近似分解
分解形式	$A = LU \approx L_{\mathrm{J}}U_{\mathrm{J}} = I_n D,$	$A = LU \approx L_{\mathrm{O1}}U_{\mathrm{O1}} = (SD^{-1} + I_n)(D + R).$
计算量	O(1)	$O(n^2)$
相对残差范数	$r_{\mathrm{J}} := \frac{\|A - L_{\mathrm{J}}U_{\mathrm{J}}\|_2}{\|A\|_2} = \frac{\|S+R\|_2}{\|A\|_2}$	$r_{\mathrm{O1}} := \frac{\|A - L_{\mathrm{O1}}U_{\mathrm{O1}}\|_2}{\|A\|_2} = \frac{\|SD^{-1}R\|_2}{\|A\|_2}$
残差尺寸	$r_J \sim O(\frac{1}{\sqrt{n}})$	$r_{O1} \sim O(\frac{1}{n})$
残差误差（n > 2^{24}）	$2^{-12} < 2^{-10}$（FP16单元舍入误差）	$2^{-24} < 2^{-23}$（FP32单元舍入误差）

Fugaku HPL-AI计算流程

- Fugaku混合精度：FP64求解IR确保小的向后误差，FP32计算Panel分解避免上/下溢，FP16计算HGEMM获得高性能

官方标准HPL-AI计算流程图



Fugaku HPL-AI计算流程图及技术应用

技术2：延迟初始化　技术3：静态缩放
技术4：动态矩阵生成　技术5：双层布局
技术6：数据打包　技术7：卸载广播
技术1：单步迭代格式

2) Iterative Refinement: The IR technique improves the accuracy of a computed solution, $\hat{x}$, for a linear system of equations, $Ax = b$. Here, we let the initial solution be $x_0 = 0$, and the iterative refinement is a series of iterations:

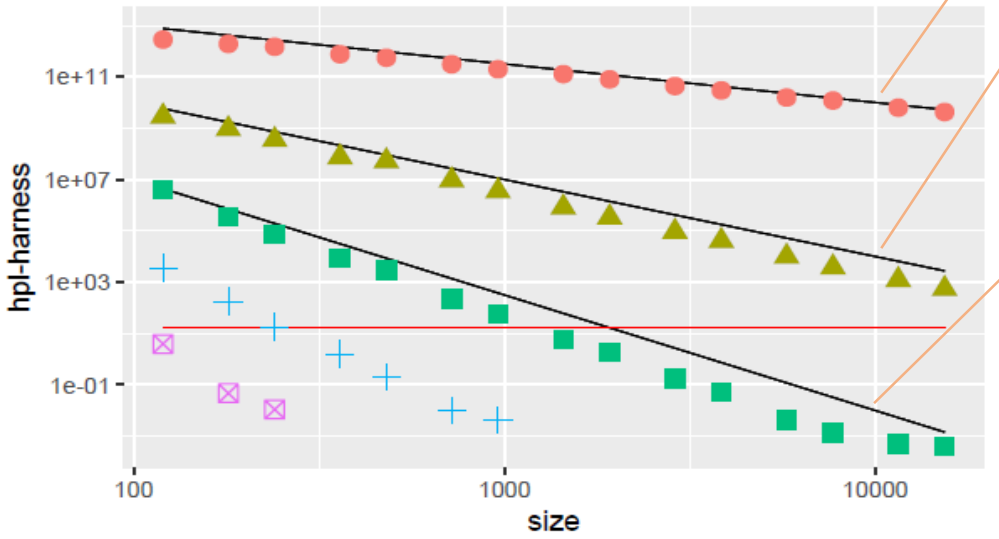
1) Residual: Compute the residual $r = b - Ax_i$.
2) Correction: Solve $Ac = r$ (e.g., using an initial LU factorization).
3) Update: Correct the current solution $x_{i+1} = x_i + c$.

技术1：单步迭代格式（SIIR）

- 基于HPL-AI矩阵的特殊性，可以做近似的LU分解（Jacobi、O1近似）并选取LU作为预条件子，选取合适的初始值 $x_0 = D^{-1}b$，初始误差$error_0 < r_J = O(\frac{1}{\sqrt{n}})$（定理1），当n大于$1.0e5$的话，迭代一步误差大概为$error = 10 < 16$，则迭代一步即可收敛

Theorem 1. *Assume A and D are nonsingular, $r_J < 0.1$, and let $\hat{x} := A^{-1}b$ be a non-zero vector. The relative error of the initial guess $e_0 := \|x_0 - \hat{x}\|_2 / \|\hat{x}\|_2 < 1.12 r_J$.*

$$\frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty} \times (n \times \epsilon)^{-1},$$



$$A = LU \approx L_J U_J = I_n D,$$

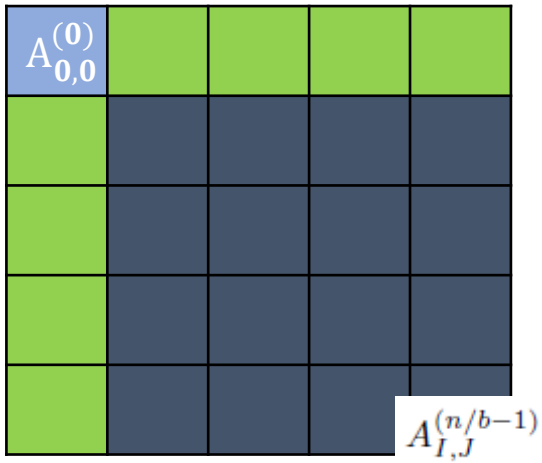
$$r_J := \frac{\|A - L_J U_J\|_2}{\|A\|_2} = \frac{\|S + R\|_2}{\|A\|_2},$$

技术2：延迟初始化

- 更改求和顺序，让初始矩阵A（FP32，LU-O(1)，D-O(n)）最后求和，确保求和项矩阵（FP16求和结果转FP32，L-$O(\frac{1}{n})$）和初始矩阵有着相同的尺度（类似GEMM中循环分块技巧）
- 缺点1：FP16求和会有数值溢出的风险，解决1：使用静态缩放技术；
- 缺点2：需要双重存储空间来存储初始矩阵和部分和，解决2：动态矩阵生成和双重布局技术

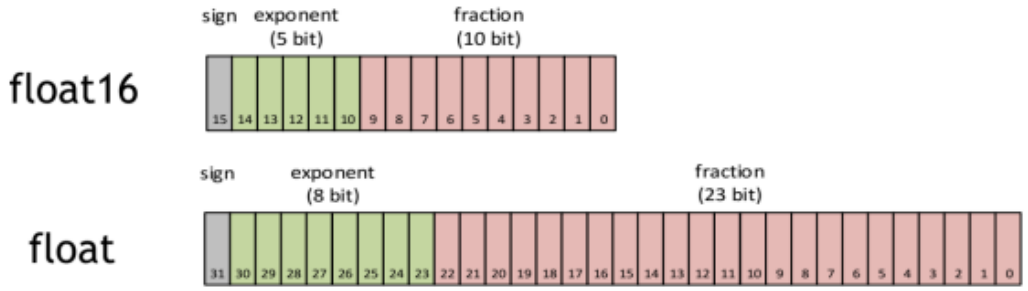
$$A_{I,J}^{(n/b-1)} = \begin{cases} \mathrm{lu}\left(A_{I,I}^{(0)} - \sum_{k=1}^{I-1} A_{I,k}^{(k)} A_{k,J}^{(k)}\right), & \text{if } I = J \\ \left(B_{I,J}^{(I)}\right)^{-1}\left(A_{I,J}^{(0)} - \sum_{k=1}^{I-1} A_{I,k}^{(k)} A_{k,J}^{(k)}\right), & \text{if } I < J \\ \left(C_{I,J}^{(J)}\right)^{-1}\left(A_{I,J}^{(0)} - \sum_{k=1}^{J-1} A_{I,k}^{(k)} A_{k,J}^{(k)}\right), & \text{otherwise} \end{cases} \tag{12}$$

$$B_{I,J}^{(I)} = \mathrm{triu}\left(A_{I,I}^{(I)}\right), \text{ and } C_{I,J}^{(J)} = \left(\mathrm{stril}\left(A_{J,J}^{(J)}\right) + I_b\right)$$



技术3：静态缩放

- 基于HPL-AI矩阵的对角优势和简单结构，使用单因子缩放Css $= O(n^{\frac{3}{4}})$，从而求和项$O(n^{-1})$变成$O(n^{-\frac{1}{4}})$ 会落在FP16范围内（当n<2^{60}）,使原来的范围$O(n^{-1}) \sim O(n^{1})$ 变成了$O(n^{-\frac{1}{4}}) \sim O(n^{\frac{1}{4}})$，对于足够大的n，数值都会落在FP16内



FP16的动态范围(6x10^{-8} ~ 65504) **远低**于 FP32的动态范围(1.4x10^{-45} ~ 1.7x10^{+38})

FP16的精度(2^{-10}) **远粗**于 FP32的精度(2^{-23})

技术4：动态矩阵生成

- 优点1：防止将FP64矩阵（IR过程）保留在内存中；
- 优点2：避免延迟初始化矩阵所需的额外存储
- 方法：使用64位并行LCG生成器可以实现动态矩阵的快速生成
- 缺点1：仍然需要两个区域来存储2个大矩阵，解决1：使用双层布局技术实现共享内存

线性同余发生器（LCG）是一种伪随机序列生成器算法，能产生具有不连续计算的伪随机序列的分段线性方程。生成器由循环关系定义 [1]：

$$X_{n+1} = (aX_n + c) \mod m.$$

这里，（1）X 是伪随机序列；

（2）m, $0 < m$ 表示模量；

（3）a, $0 < a < m$ 表示乘数；

（4） 表示增量；

（5）X_0, $0 \le X_0 < m$ 表示初始值。

```
subroutine MPLU(n,b,A)
integer, intent(IN)  :: n               ! matrix size
integer, intent(IN)  :: b               ! block size
real(4), intent(OUT) :: A(b,b,n/b,n/b) ! result in FP32
integer :: i, j, k, nb
real(4) :: s, is
real(2) :: B(b,b,n/b,n/b), L(b,b,n/b) U(b,b,n/b)
nb = n / b
B(:,:,1:nb,1:nb) = 0
s = sqrt(n*sqrt(n))
is = 1. / s
do k=1,nb
   ! static scaling and lazy initialization
   A(:,:,k:nb,k) = MG(b,k,nb,k,k+1) - is*real(B(:,:,k:nb,k),4)
   A(:,:,k,k+1:nb) = MG(b,k,k+1,k+1,nb) - is*real(B(:,:,k,k+1:nb),4)
   A(:,:,k,k) = LU(A(:,:,k,k))
   if(k.eq.nb) exit
   A(:,:,k+1:nb,k) = TRSM("R","U","N","N",A(:,:,k,k),A(:,:,k+1:nb,k))
   A(:,:,k,k+1:nb) = TRSM("L","L","N","U",A(:,:,k,k),A(:,:,k,k+1:nb))
   L(:,:,k+1:nb) = real(s*A(:,:,k+1:nb,k),2)
   U(:,:,k+1:nb) = real(A(:,:,k,k+1:nb),2)
   do j=k+1:nb
      do i=k+1:nb
         B(:,:,i,j) = B(:,:,i,j) + L(:,:,i)*U(:,:,j)
      end do
   end do
end do
end subroutine
```

动态矩阵生成MG

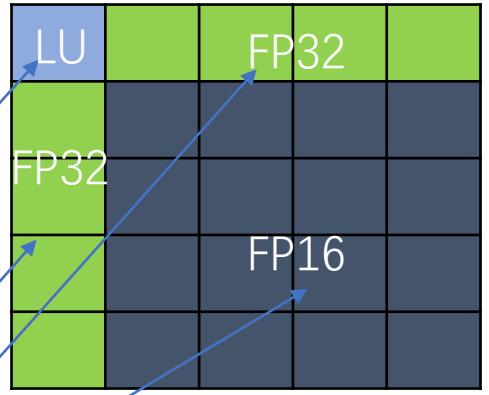


Fig. 5. A pseudo-program of the mixed-precision LU factorization with lazy initialization, static scaling and on-the-fly matrix generation techniques. MG generates part of the input matrix, TRSM corresponds to the BLAS's xTRSM, and LU factorizes a small part of the matrix.

```
uint64_t lcg(uint64_t n){
   if(n==0u) return 1;
   else return lcg(n-1)*6364136223846793005+1;
}
double drandlcg(uint64_t n){
   return 0x1.fffffffffffffP -65*(int64_t)lcg(n-1)
}
```

Fig. 1. A slow, recursive code for generating linear congruent generator (LCG) pseudo-random numbers in the range $[-0.5, 0.5)$. We ignored the small bias $\le 2^{-65}$ to $-\infty$ for performance.

技术5：双层布局

- 运行过程中，基于初始矩阵A（FP32）和求和矩阵B（FP16）使用的先后顺序，使用矩阵A的下半部分存储矩阵B实现共享内存，减少1/3内存占用空间
- 每个矩阵元素的内存消耗从8+4+2（FP64+32+16）到4字节（FP32），从而允许将矩阵的维度大致翻一番

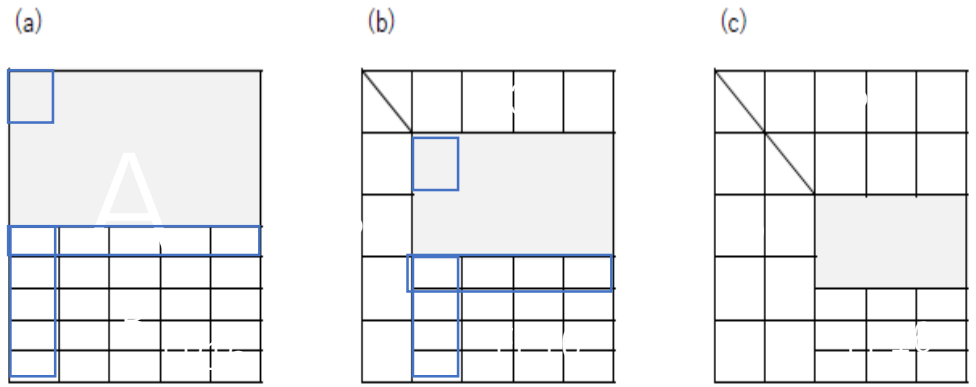


Fig. 6. Schematic view of the mixed-precision LU decomposition, FP16 panels in rectangular, FP32 in square, and unused regions are filled in gray. (a) The initial state. All FP16 matrix elements are stored in the lower half of each column and zero initialized. (b) After the first step. The first column/row panels are converted to FP32 and initial random matrix elements are added. The diagonal panel is factorized by LU, TRSM for the column/row panels, and the rest are updated by GEMM. (c) After the second step.

技术6：数据打包

- 数据打包是GEMM中的一种技术

✓ 打包成本低，打包的数据在矩阵乘法中被重用

✓ 如果GEMM之外具有可重用性，可以在GEMM之前打包以提高额外的可重用性并降低打包成本，例如intel xgemm_pack

- 混合精度LU分解的MPI版本中，LU分解在单个进程中生成，在广播前对LU打包以便多个进程中重用打包数据；Intel在Xeon Phi运行LINPACK中使用了相同的技术

- Fukagu新打包技术是执行三种数据移动，即同步打包、FP32转换FP16和静态缩放；该技术可以提升HGEMM的性能2%，同时打包的成本占总时间小于0.1%

技术7：HW卸载广播--基于硬件卸载的集体通信

- 前面6个技术都是可移植或者修改之后可复制的，除了通信部分，它使用了Fugaku的网络接口TofuD的功能
- MPI库支持持久通信的硬件卸载功能，修改算法实现了可更改广播的大小和根，算法原理和HPL中实现的1-Ring通信本质相通，这是Fugaku与Summit的主要区别

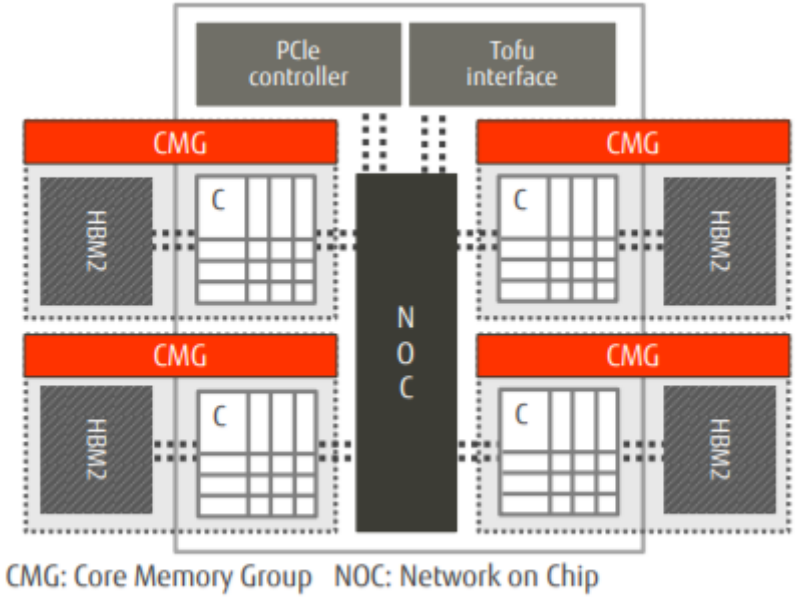


Figure 2 PRIMEHPC FX1000 compute nodes

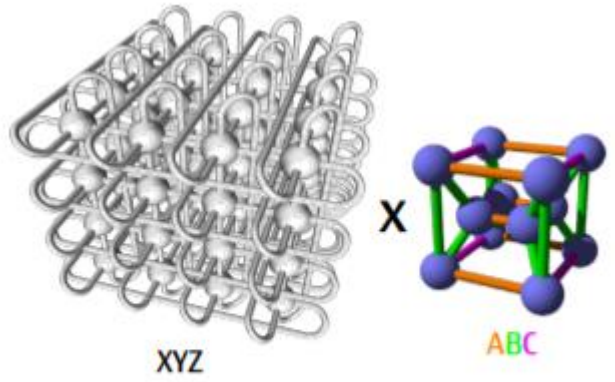


Figure 3 Tofu interconnect D

参考论文中的文献[1] Y. Ajima et al. “The Tofu Interconnect D”. In: IEEE Cluster ’18. 2018, pp. 646–654.

Fugaku实现的性能结果

- 第一阶段，测试了2个不同大小的矩阵，大小矩阵内存占用分别等于23.7GB和5.9GB

PERFORMANCE ON 6912 NODES, FOR TINY AND LARGE MATRIX SIZES, AND VARYING BLOCK SIZE b. EFFICIENCIES OF THE HGEMM KERNEL PART ONLY ARE SHOWN IN THE LAST COLUMN.

n	b	T (sec)	PFlop/s	Eff. (%)	(hgemm)
3,317,760	288	311.8	78.07	91.92	95.35
"	320	310.8	78.34	92.24	95.86
"	576	312.6	77.88	91.70	97.31
"	640	313.2	77.73	91.51	97.77
6,635,520	288	2427	80.26	94.49	96.05
"	320	2416	80.61	94.91	96.53
"	576	2403	81.07	95.45	97.89
"	640	2400	81.15	95.54	98.25

- 第二个阶段，使用126720个节点

PERFORMANCE ON 126,720 (528 × 240) NODES

n	b	T (sec)	PFlop/s	Eff. (%)	(hgemm)
13,516,800	320	1158	1421	91.27	95.87

June 2020

Rank	Site	Computer	Cores	HPL-AI (Eflop/s)	TOP500 Rank	HPL Rmax (Eflop/s)	Speedup
1	RIKEN, Japan	Fugaku	7,299,072	1.42	1	0.416	3.42
2	ORNL, USA	Summit	2,414,592	0.55	2	0.144	3.83

```
!REMAP FOR 330 RACKS
jobid=567937
n=13516800 b=320 r=1056 c=480
2dbc lazy rdma full pack
numasize=4 numamap=CONT2D nbuf=3
epoch_size = 1689600
#BEGIN: Fri May 15 08:03:02 2020
!epoch 0/8: elapsed=0.000321, 0.000000 Pflops (estimate)
!epoch 1/8: elapsed=370.133893, 1468.210388 Pflops (estimate)
!epoch 2/8: elapsed=650.033814, 1464.253280 Pflops (estimate)
!epoch 3/8: elapsed=851.582777, 1461.317344 Pflops (estimate)
!epoch 4/8: elapsed=988.277782, 1457.670705 Pflops (estimate)
!epoch 5/8: elapsed=1072.277294, 1454.437530 Pflops (estimate)
!epoch 6/8: elapsed=1117.625903, 1450.088542 Pflops (estimate)
!epoch 7/8: elapsed=1143.143441, 1437.409844 Pflops (estimate)
# iterative refinement: step= 0, residual=5.0168606685474515e-04 hpl-harness=222677.730166
# iterative refinement: step= 1, residual=1.5618974863462753e-11 hpl-harness=0.006931
#END_: Fri May 15 08:22:50 2020
1158.483898010 sec. 1421151817.927741528 GFlop/s resid = 1.561897486346275e-11 hpl-harness = 0.006931424
1421.151818 Pflops, 91.267070 %
```

Fig. 7. The dump list of the output of the benchmark with 126,720 nodes. Because an A64FX has four numa groups in a node, the number of MPI processes is 506,880, approximately, half million. One epoch is equivalent to 5,280 steps of the algorithm in this setting.

小结：

- 混合精度计算：低精度计算提高性能，高精度迭代保证计算精度
- 对角占优矩阵的LU分解不需要使用传统的Gauss消去法，利用近视分解，计算量只需O1，O(n^2)，即可以利用混合精度实现和原方法精度一样的效果
- 混合精度计算过程中需要涉及到的问题有：数值越界、内存翻倍、MPI通信
- 使用的技术包括：延迟初始化、静态缩放、动态矩阵生成、双层布局、数据打包、卸载广播和单步迭代格式（针对特殊情况的算法优化）

- 混合精度LU分解对数学库的LAPACK、SOLVER的LU分解具有借鉴意义
- 前6个技术主要是解决和优化FP16引入导致的问题，对混合精度计算具有参考价值
- HGEMM是半精度的GEMM实现，数据打包是GEMM常用优化手段之一，帮助HGEMM性能提升2%左右

- 开源数学库：MAGMA（异构平台数学库，为混合多核多GPU系统设计线性代数算法和框架）、SLATE等

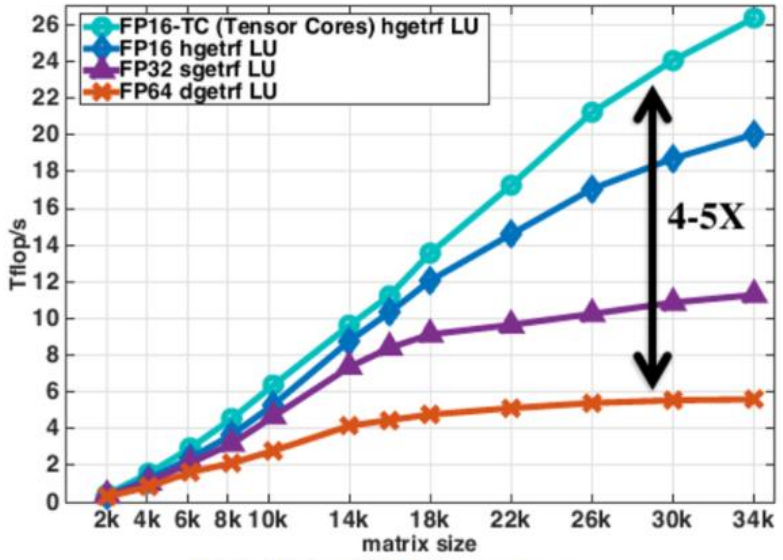


图2 混合精度LU分解的测试(Fig 5 in [2])

混合精度稠密迭代求解器LU和GMRES实现：基于FP16，FP32，FP64的Tensor Core提供4倍的加速，文章：Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers

OpenEuler

我参与 我做主

下载体验

您可以通过以下四种方式体验openEuler操作系统

公有云　虚拟机　硬件　树莓派

加入贡献

请根据您的参与身份，选择签署：

个人CLA　企业CLA　员工CLA

vsva

Zhangfei Gao Linaro

2022.4

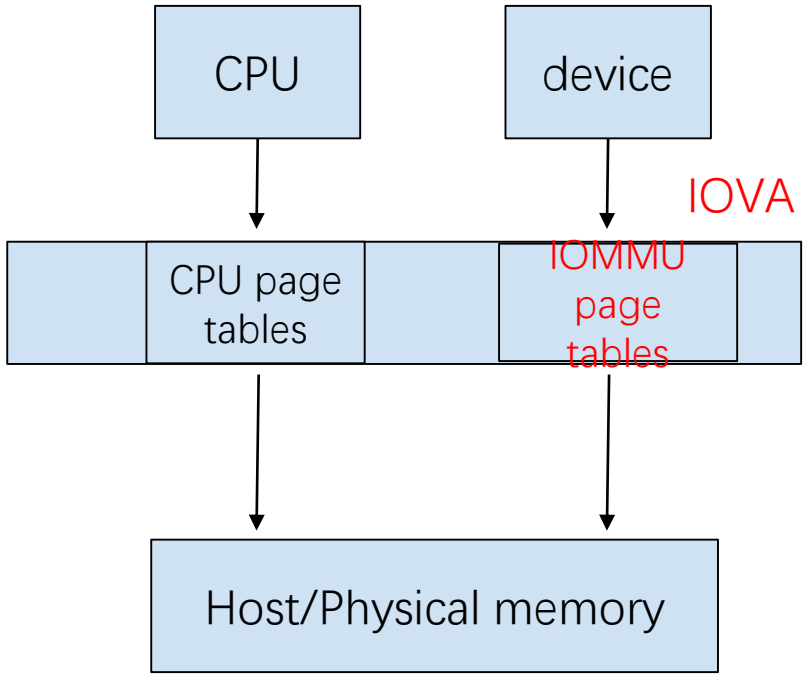
iommu function

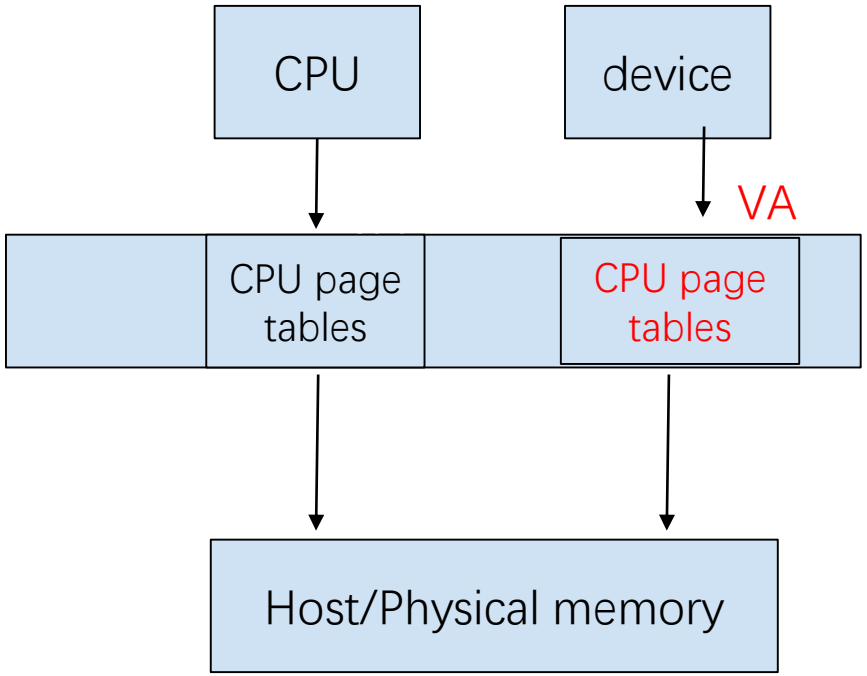
- Isolation
  - dev1 -> pa 1 && dev2 -> pa2 ==> dev1 !-> pa2
- Translate
  - iova -> pa

sva

- cpu & iommu share virtual address iova = va
  - Translate, why not VA?
- Goal: User space can use va for dma, no memcpy

sva

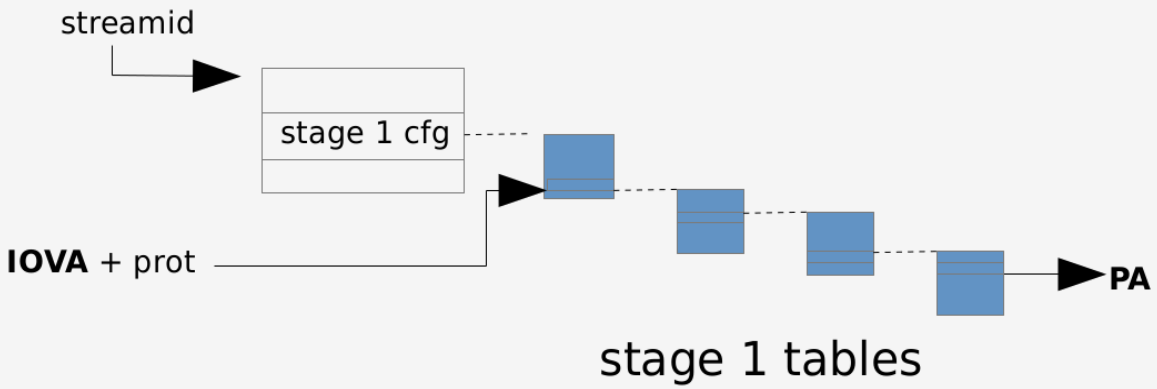

CPU
device
IOVA
CPU page tables
IOMMU page tables
Host/Physical memory


CPU
device
VA
CPU page tables
CPU page tables
Host/Physical memory

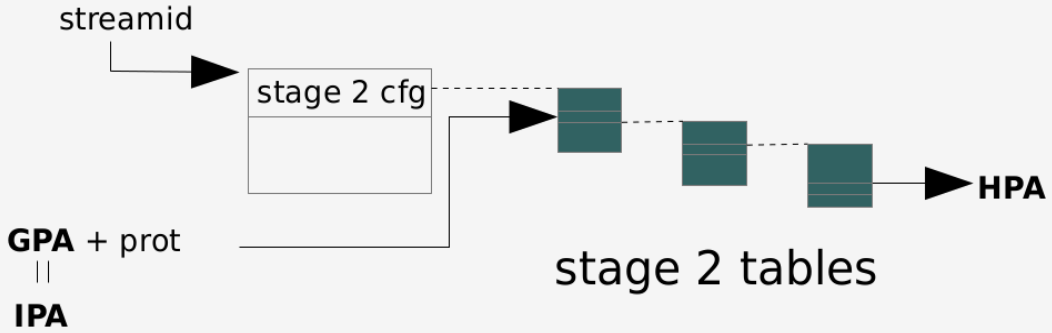
vsva

- Goal: Guest user space use va for dma
- How:
  - Translate: iova -> gpa -> hpa
  - Page fault:
    - host trigger page fault -> guest
    - guest page response -> host

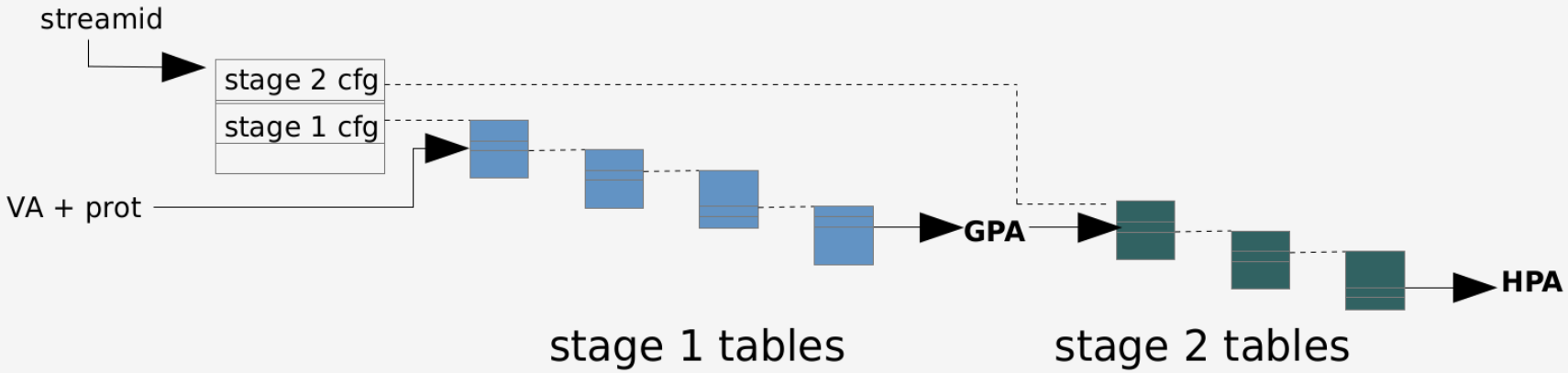
Translation Stages and HW Nested Paging



stage 1 Only (OS)

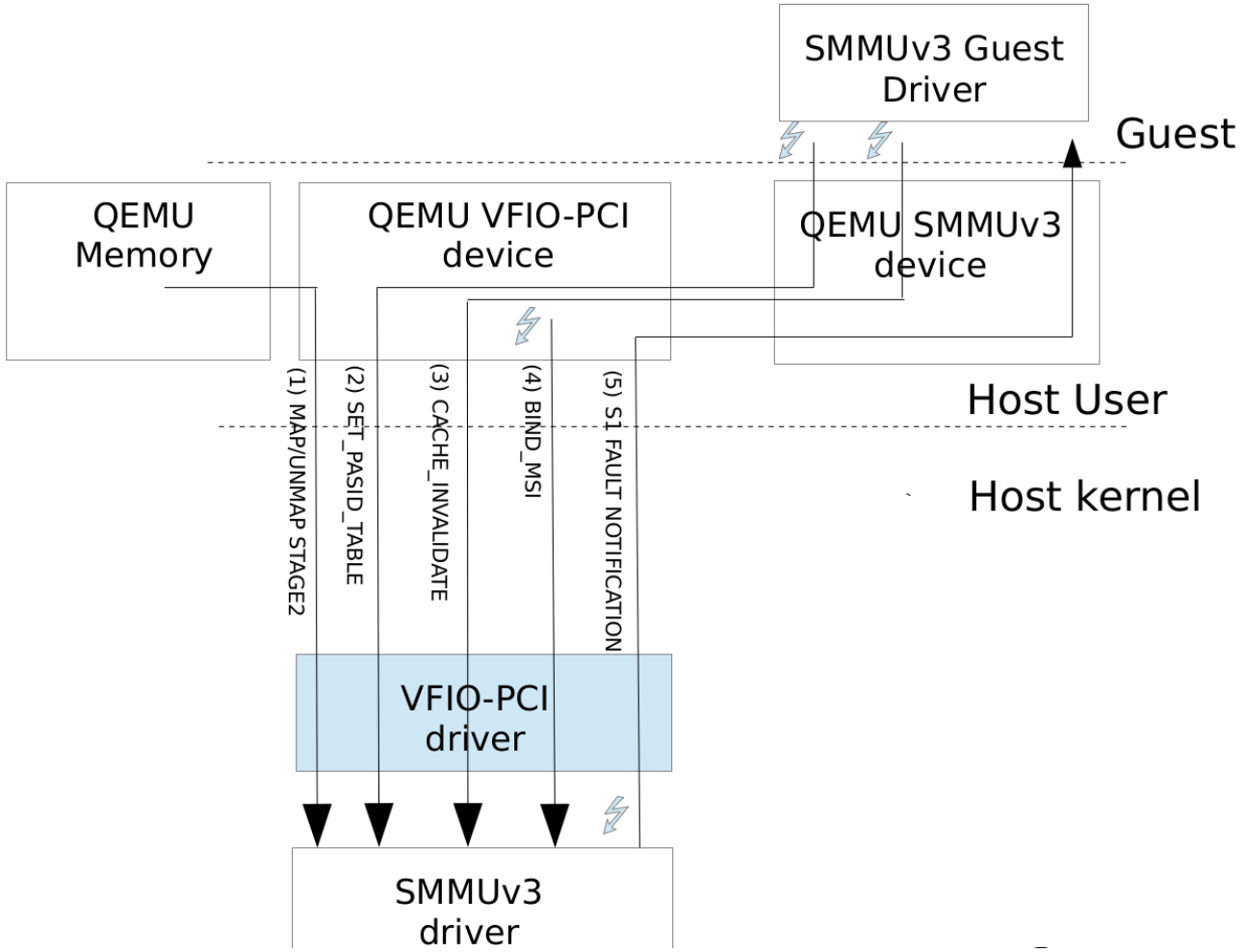


stage 2 Only (hyp)



v1: via vfio -- From Eric

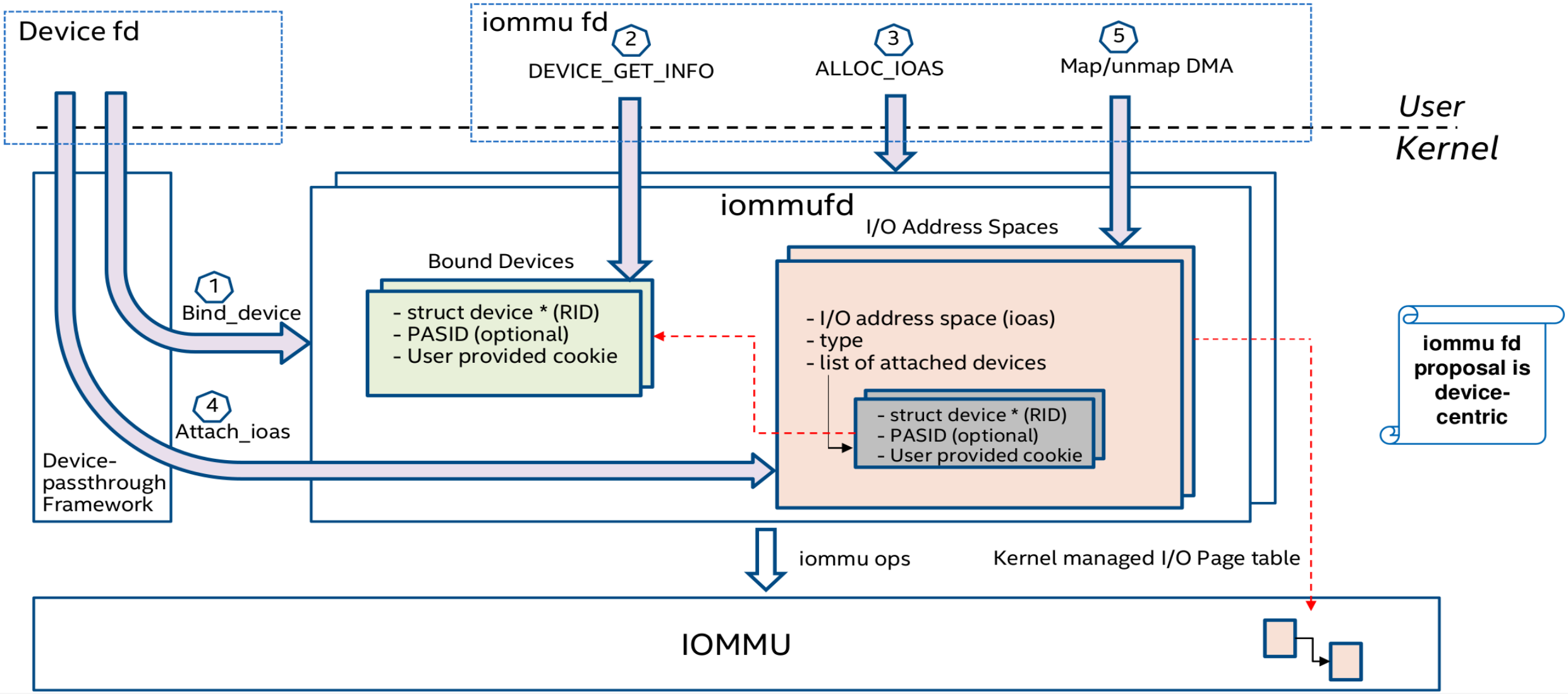
1	A Guest RAM region is added	Build stage2 mapping. Force HW stage2 to be used.
2	Guest config invalidation commands	Propagate stage 1 guest config to the host
3	Guest sends TLB/PASID cache invalidation commans	Propagate invalidations to Host
4	MSI Enable	Propagate stage1 MSI binding from guest to host
5	Stage 1 related fault	Propagate stage 1 faults from host to guest



v1: via vfio

- Status
  - Ready for one year, but not accepted.
  - Already supported
    - kernel: https://github.com/Linaro/linux-kernel-uadk/tree/uacce-devel-5.16
    - qemu: https://github.com/Linaro/qemu/tree/v6.1.0-rmr-v2-nested_smmuv3_v10

v2: via /dev/iommu From Yi Liu



v2: via /dev/iommu

- Status: in developing
  - kernel:
    - https://github.com/luxis1999/iommufd/commits/iommufd-v5.17-rc6
  - qemu:
    - https://github.com/eauger/qemu/commits/qemu-for-5.17-rc4-vm-rfcv11
    - Want to support both legacy vfio interface and /dev/iommu interface
  - Readme draft
    - https://lore.kernel.org/linux-iommu/BN9PR11MB5433B1E4AE5B0480369F97178C189@BN9PR11MB5433.namprd11.prod.outlook.com/