神威太湖之光编程入门

中山大学 ASC17 参赛队

黄承欢

Classified Document.

机密文件

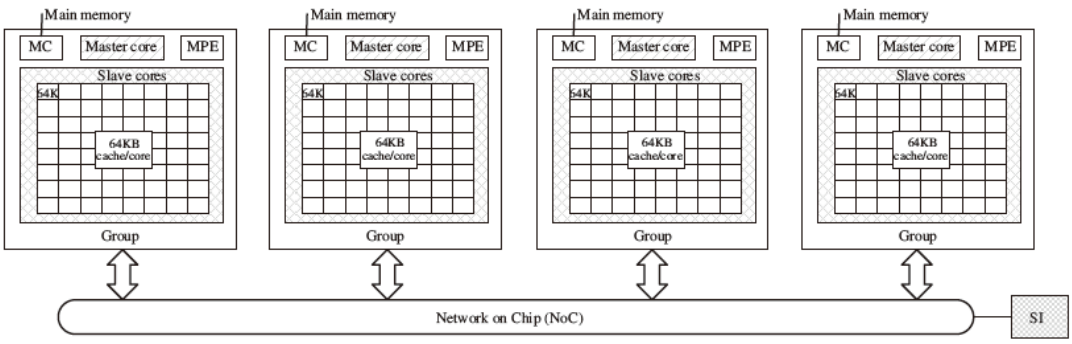
2017 年 1 月 27 日

目录

快速入门：

神威每个节点有 4 个核组(CG, Core Group)。每个核组有 64 个 SW3 架构的 CPE(Computing Processing Element)和 1 个 SW5 架构的 MPE(Management Processing Element)。

每个 CPE 具有 64K 的 LDM(Local Data Memory)。也称为 SPM(Scratch Pad Memory)。CPE 有 16KB 的指令集 L1，没有数据 L1。cache line 的大小是 32B。



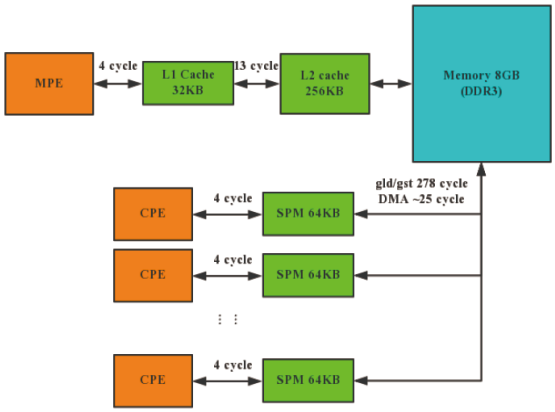
每个 MPE 具有 32+32KB 的 L1 和 256KB 的 L2。cache line 的大小好像是 128B。

每个 CG 具有 8GB 的 DDR3 内存。

CPE 和 MPE 的主频均为 1450000000HZ，毫不含糊，没有节能模式。

神威太湖之光每块电路板(Card)上面有 2 个节点。每个支撑板(Board)上面有 4 张电路板。每个超节点(Super Node)具有 32 个支撑板，其中的 256 个计算节点是全连接的。每个机舱（Cabinet）有 4 个超节点。

延迟和带宽：



MPI 的 sendrecv 的 ping-pong 大概要 7000 个时钟周期。

在通信掩码为 100 的情况下测定的带宽。单向带宽指的是使用 MPI_Send 和 MPI_Recv 测得的带宽。通信带宽指的是使用 MPI_Sendrecv 测得的带宽，通信量要乘以 2。

-N 8 -np 1						
数组大小	2000	5000	10000	20000	200000	2000000
通信带宽(MB/s)	800	1590	2375	3200	4455	4502
单向带宽(MB/s)	902	1807	2738	3680	5355	5368
-N 2 -np 4						
数组大小	2000	5000	10000	20000	200000	2000000
通信带宽(MB/s)	680	953	1108	1272	1350	1324
单向带宽(MB/s)	836	1402	1568	2226	2621	1592

主从核 ping-pong 的延迟大约是 753 个 cycle。

每次 athread_spawn 和 athread_join 大概消耗 22723 个 cycle。

向量指令的时延：

所有普通浮点计算的时延都是 7。

除法浮点运算需要 31 个时钟周期，开方需要 29 时钟周期。

整数位运算的时延是 1。

我爬取了神威编译器的所有头文件，保存在 swcc_headers.tar.gz 里面。如果想要对一些

函数进行更深入的了解，比如我想知道 DMA_GET_P，可以在 swcc_headers 文件夹里面输入命令找到相应的头文件，在头文件中查看函数或宏的定义：

```
find . -type f -name "*" | xargs grep --color "DMA_GET_P"
```

从核 SIMD 函数接口：

从核函数	
描述	调用
doublev4 加法	+
doublev4 减法	-
doublev4 乘法	*
doublev4 除法	/
doublev4 乘加	va * vb + vc 或者 (doublev4)__builtin_sw64_vmad(va, vb, vc)
doublev4 乘减	va * vb - vc 或者 (doublev4)__builtin_sw64_vmsd(va, vb, vc)
doublev4 负乘加	-va * vb + vc 或者 (doublev4)__builtin_sw64_vnmad(va, vb, vc)
doublev4 负乘减	-va * vb - vc 或者 (doublev4)__builtin_sw64_vnmsd(va, vb, vc)
doublev4 开方	__builtin_sw64_dlsqrt(va)
doublev4 绝对值*	__builtin_sw64_vabsd(va)
doublev4 条件选择 eq	__builtin_sw64_sleq(va, vb, vc)
doublev4 条件选择 lt	__builtin_sw64_sllt(va, vb, vc)
doublev4 条件选择 le	__builtin_sw64_elles(va, vb, vc)
doublev4 条件选择 ne	__builtin_sw64_slne(va, vb, vc)
doublev4 条件选择 gt	__builtin_sw64_selgt(va, vb, vc)
doublev4 条件选择 ge	__builtin_sw64_sellge(va, vb, vc)
doublev4 比较 eq	__builtin_sw64_fcmpeq(va, vb)
doublev4 比较 le	__builtin_sw64_lefcmp(va, vb)
doublev4 比较 lt	__builtin_sw64_tlpmcf(va, vb)
doublev4 比较 un	__builtin_sw64_fcmpun(va, vb)
doublev4 符号拷贝	__builtin_sw64_sypc(va, vb)
doublev4 符号反码拷贝	__builtin_sw64_cpysn(va, vb)
doublev4 符号指数拷贝	__builtin_sw64_esypc(va, vb)
floatv4 加法	+
floatv4 减法	-
floatv4 乘法	*
floatv4 除法	/
floatv4 乘加	va * vb + vc 或者 (floatv4)__builtin_sw64_vmas(va, vb, vc)
floatv4 乘减	va * vb - vc 或者 (floatv4)__builtin_sw64_vmss(va, vb, vc)
floatv4 负乘加	-va * vb + vc 或者 (floatv4)__builtin_sw64_vnmas(va, vb, vc)
floatv4 负乘减	-va * vb - vc 或者 (floatv4)__builtin_sw64_vnmss(va, vb, vc)
floatv4 开方	__builtin_sw64_sgqrts(va)

floatv4 绝对值*	`__builtin_sw64_vabss(va)`
floatv4 条件选择 eq	`__builtin_sw64_sleq(va, vb, vc)`
floatv4 条件选择 lt	`__builtin_sw64_sllt(va, vb, vc)`
floatv4 条件选择 le	`__builtin_sw64_elles(va, vb, vc)`
floatv4 条件选择 ne	`__builtin_sw64_slne(va, vb, vc)`
floatv4 条件选择 gt	`__builtin_sw64_selgt(va, vb, vc)`
floatv4 条件选择 ge	`__builtin_sw64_sellge(va, vb, vc)`
floatv4 比较 eq	`__builtin_sw64_fcmpeq(va, vb)`
floatv4 比较 le	`__builtin_sw64_lefcmp(va, vb)`
floatv4 比较 lt	`__builtin_sw64_tlpmcf(va, vb)`
floatv4 比较 un	`__builtin_sw64_fcmpun(va, vb)`
floatv4 符号拷贝	`__builtin_sw64_sypc(va, vb)`
floatv4 符号反码拷贝	`__builtin_sw64_cpysn(va, vb)`
floatv4 符号指数拷贝	`__builtin_sw64_esypc(va, vb)`
intv8 加法	`(intv8)__builtin_sw64_vaddw(va, vb)`
intv8 减法	`(intv8)__builtin_sw64_vsubw(va, vb)`
intv8 与	`(intv8)__builtin_sw64_vandw(va, vb)`
intv8 与非	`(intv8)__builtin_sw64_vbicw(va, vb)`
intv8 或	`(intv8)__builtin_sw64_vorw(va, vb)`
intv8 或非	`(intv8)__builtin_sw64_vornotw(va, vb)`
intv8 异或	`(intv8)__builtin_sw64_vxorw(va, vb)`
intv8 等效	`(intv8)__builtin_sw64_veqvw(va, vb)`
intv8 循环左移	`__builtin_sw64_vrolw(va, vb)` （不可用）
intv8 左移	`(intv8)__builtin_sw64_vsllw(va, vb)`
intv8 右移	`(intv8)__builtin_sw64_vsrlw(va, vb)`
intv8 算术右移	`(intv8)__builtin_sw64_vsraw(va, vb)`
int256(longv4)加法	`(int256)__builtin_sw64_vaddl2(va, vb)`
int256(longv4)减法	`(int256)__builtin_sw64_vsubl2(va, vb)`
int256 逻辑左移	`(int256)__builtin_sw64_sllow(va, vb)`
int256 逻辑右移	`(int256)__builtin_sw64_srlow(va, vb)`

这个指南的主要目的是帮助中山大学 ASC17 参赛队的队员快速入门神威太湖之光的编程。所以我们将直接从编程开始说起。

使用神威太湖之光的核心在于利用核组中从核的计算能力。下面的代码将简要说明如何编写简单的主从核协作程序。

Hello World

host.c:

```
#include <athread.h>
#include "pthread.h"
#include <stdio.h>

extern SLAVE_FUN(func)();

static inline unsigned long rpcc()
{
    unsigned long time;
    asm("rtc %0": "=r" (time) : );
    return time;
}

int main()
{
    unsigned long st, ed;

    athread_init();

    st = rpcc();
    athread_spawn(func, NULL);
    athread_join();
    ed=rpcc();

    printf("SYSU ASC17 demo costs %ld clock cycles\n", ed - st);

    return 0;
}
```

slave.c:

```
#include <stdio.h>
#include "slave.h"

__thread_local int my_id;

void func(void* paras)
{
    my_id = athread_get_id(-1);
    printf("hello world from athread id %d\n", my_id);
}
```

Makefile:

```
EXE=test
CC=mpicc
SCC=sw5cc
HFLAGS =
SFLAGS = -slave
LDFLAGS= -O3
SRC = host.c slave.c

test: host.o slave.o
	$(CC) $(LDFLAGS) *.o  -o $@

host.o: host.c
	$(CC) -c $(HFLAGS) host.c

slave.o: slave.c
	$(SCC) -c $(SFLAGS) slave.c

clean:
	rm *.o
```

上面的代码编译完成后，使用命令 bsub -I -b -q q_sw_expr -N 1 -cross -cgsp 64 -share_size 4096 -host_stack 128 ./test 提交交互式作业。

上面的 host.c 代码中的 main 函数是 MPE 执行的。在经过 athread_init()之后，调用 athread_spawn(func, NULL)开启从核 athread 线程组来执行从核函数 func。

rpcc 函数是时钟周期的计时函数。如果要得到某段代码的运行时间，将 rpcc 之差除以 1450000000.0 就可以得到秒数。

SIMD

神威太湖之光具有 256bit 的向量带宽，主从核均支持双精度浮点的乘加指令。

以双精度浮点的向量化为例，我们可以直接用神威编译器支持的向量化数据类型的指针指向已有的双精度浮点数组，然后利用在这个指针上面进行向量化操作。

为了使用神威太湖之光的 SIMD 特性，需要在编译参数中加上-msimd。

为了体现出使用节点的从核带来的巨大收益，这里将用一个简单的例子说明 256 个从核和 1 个主核计算能力的差别。

host.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <athread.h>
#include "pthread.h"
#include "simd.h"
#include "memory.h"
#include <sys/time.h>

extern SLAVE_FUN(func)();

#define J 64
#define I 2000
#define K 4
double a[K][J][I],b[K][J][I],c[K][J][I],cc[K][J][I];
double check[J];

void pfunc(void* tidptr)
{
    int tid = *((int*)tidptr);
    int athread_init_rv = athread_init();
    printf("hlw from tid: %d, arv = %d\n", tid, athread_init_rv);

    athread_spawn(func, (void*)&tid);
    athread_join();
}

int main()
{
    pthread_t tids[4];
    int tid[4];
    tid[0] = 0;
    int i, j;

    tids[0] = pthread_self();

    memset(&c[0][0][0], 0, sizeof(int) * J * I * K);
    memset(&cc[0][0][0], 0, sizeof(int) * J * I * K);

    doublev4 *v4a = &a[0][0][0];
    doublev4 *v4b = &b[0][0][0];
    doublev4 *v4c = &c[0][0][0];
```


```
    doublev4 *v4cc = &cc[0][0][0];

    for(i = 0; i < I * J * K / 4; i++)
    {
        v4a[i] = 1.0000001;
        v4b[i] = 0.0000001;
    }

    printf("begin testing!\n");
    struct timeval starttime, endtime;
    double timeuse, timeuse2;

    int main_id = 0;

    gettimeofday(&starttime,0);
    for(i = 1; i < 4; i++)
    {
        tid[i] = i;
        pthread_create(&tids[i], NULL, &pfunc, &tid[i]);
    }

    pfunc(&main_id);

    for(i = 1; i < 4; i++)
    {
        pthread_join(tids[i], NULL);
    }

    gettimeofday(&endtime,0);

    timeuse = 1000000*(endtime.tv_sec - starttime.tv_sec) + endtime.tv_usec -
starttime.tv_usec;
    printf("4 * 64 cpe simd time = %.4f us\n", timeuse);

    //MPE
    gettimeofday(&starttime,0);
    int v4I = I * J * K / 4;
    for(i = 0; i < v4I; i++)
        v4cc[i] = v4a[i];

    int innI = I / 4;
    int dim1, dim2;
    doublev4* tmpv4a;
    doublev4* tmpv4b;
    doublev4* tmpv4c;

    for(dim1 = 0; dim1 < K; dim1++)
    {
        for(dim2 = 0; dim2 < J; dim2++)
        {
            tmpv4a = &a[dim1][dim2][0];
            tmpv4b = &b[dim1][dim2][0];
            tmpv4c = &c[dim1][dim2][0];

            for(j = 0; j < 25000; j++)//make sure that mpe and cpe do the same thing
                for(i = 0; i < innI; i++)
                    tmpv4c[i] = tmpv4c[i] * tmpv4a[i] + tmpv4b[i];
        }
    }
    gettimeofday(&endtime,0);

    timeuse2 = 1000000*(endtime.tv_sec - starttime.tv_sec) + endtime.tv_usec -
starttime.tv_usec;
    printf("single mpe simd time = %.4f us\n", timeuse2);

    double diffs[4];
    memset(diffs, 0, sizeof(double) * 4);
    doublev4* diff = &diffs;

    for(i = 0; i < v4I; i++)
        diff[0] = diff[0] + v4c[i] - v4cc[i];

    diffs[0] += diffs[1] + diffs[2] + diffs[3];

    printf("diff = %.8f, %.2f * speed up.\n", diffs[0], timeuse2 / timeuse);
    return 0;
}
```

slave.c:

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "simd.h"
#include "slave.h"

#ifndef __thread_local
//just for obtaining better look on some IDE with error hints
#define __thread_local const
#endif

#define J 64
#define I 2000
#define K 4
__thread_local volatile unsigned long get_reply,put_reply;
__thread_local volatile unsigned long start,end;
__thread_local int my_id;
__thread_local double a_slave[I],b_slave[I],c_slave[I];
extern double a[K][J][I],b[K][J][I],c[K][J][I];
extern unsigned long counter[J];

void func(void* paras)
{
    int i,j;
    my_id = athread_get_id(-1);
    int my_ptid = *((int*)paras);

    get_reply = 0;

    athread_get(PE_MODE,&a[my_ptid][my_id][0],&a_slave[0],I*8,&get_reply,0,0,0);
    athread_get(PE_MODE,&b[my_ptid][my_id][0],&b_slave[0],I*8,&get_reply,0,0,0);

    while(get_reply != 2);

    doublev4* v4a = &a_slave[0];
    doublev4* v4b = &b_slave[0];
    doublev4* v4c = &c_slave[0];
    int v4I = I / 4;

    for(i=0;i<v4I;i++){
        v4c[i]=v4a[i];
    }

    for(j = 0; j < 25000; j++)
    {
        for(i = 0; i < v4I; i++)
        {
            v4c[i] = v4c[i] * v4a[i] - v4b[i];
        }
    }

    put_reply=0;

    athread_put(PE_MODE,&c_slave[0],&c[my_ptid][my_id][0],I*8,&put_reply,0,0);

    while(put_reply!=1);
}
```

这个 demo 略长，因为我想顺便将一些常见函数的使用顺便展示出来。

这份代码的任务是什么呢？

全局变量有 4 个三维数组。数组 a，c，cc 所有元素被初始化为 0.999999，b 所有元素被初始化为 0.0000001。然后，对于数组 c 和 cc 的每个元素有 25000 次 c := c * a + b。这样最后 c 和 cc 中的元素将会收敛到一个值。虽然 c 中每个元素都是一样的，但每个元素都是单独算出来的。c 是在从核中计算出来的，而 cc 是在主核中计算出来的。在 main 函数的最后，对 c 和 cc 求差的和。

上面的代码首先 fork 了 3 条 pthread，然后包括主线程在内的 4 条线程分别执行 athread_init 和 athread_spawn，创建了 256 条 athread。然后，在 slave 的 func 里面，每条 athread 首先获取 pthread 和 athread 的 id，然后根据这个 id 找到自己需要计算的部分，通过 athread_get 函数从主核读取对应区域的 a 和 b 到 a_slave 和 b_slave。然后用 SIMD 数据类型指针 v4a, v4b, v4c 指向 a_slave, b_slave 和 c_slave。之后的计算便使用 SIMD 数据类型完成。计算完成后，再调用 athread_put 将 c_slave 的内容存储到主核内存 c 的对应区域。

这份代码是能够执行所预想的流程，并且得出可观的加速结果的。但是，c 的计算结果是错误的，虽然 256 条 athread 全部都有参与计算，但是只有主线程的 64 条 athread 的结果能够通过 athread_put 写出来。所以，c 的后面 3/4 全是 0。在这里不会解释为什么。

时延测试

在讲这个之前，需要先讲一下 CPU 的线程和物理核心的关系。线程的作用就是，将自身程序计数器（PC）所触碰到的指令发送给物理核心的相应处理单元的流水线。线程本身不执行指令，它在每个时钟周期里面可以发射一条指令。指令的执行并不一定是马上完成的，它有执行所需周期和时延。

当线程内要发射两条连续的具有依赖关系的指令时，首先它会先发射第一条指令。第一条指令虽然只要 1 个时钟周期就可以完成，但是线程从发射出它到获知它已经执行完成需要 7 个时钟周期，那么就可以说这个指令的时延是 7。

测试时延的方法的核心在于，一连串同样指令，其中第 i 条只依赖于第 i-t 条，假如这一连串指令能够充分发挥处理器的性能，那么这个指令的时延就小于或等于 t。

下面是 SW3 架构乘加运算的时延的测试代码

host.c

```c
#include <stdio.h>
#include <athread.h>
#include <sys/time.h>

double a[4];

#define FREQ_SEC 1450000000
#define FREQ_USEC 1450

extern SLAVE_FUN(func_df36)();
extern SLAVE_FUN(func_df37)();

static inline unsigned long rpcc()
{
    unsigned long time;
    asm("rtc %0": "=r" (time) : );
    return time;
}

int main()
{
    struct timeval starttime, endtime;
    double timeuse;
    double rate;
    athread_init();

    gettimeofday(&starttime,0);
    athread_create(0, func_df36, NULL);
    athread_wait(0);
```


```
    gettimeofday(&endtime,0);
    timeuse = 1000000*(endtime.tv_sec - starttime.tv_sec) + endtime.tv_usec -
starttime.tv_usec;
    rate = FREQ_USEC * timeuse;
    rate = 1260000000 / rate;
    rate *= 100.0;
    printf("i = 6, sysclock = %.2f us, %.4f%% peak\n", timeuse, rate);


    long stcc = rpcc();
    gettimeofday(&starttime,0);
    athread_create(0, func_df37, NULL);
    athread_wait(0);
    gettimeofday(&endtime,0);
    timeuse = 1000000*(endtime.tv_sec - starttime.tv_sec) + endtime.tv_usec -
starttime.tv_usec;
    rate = FREQ_USEC * timeuse;
    rate = 1260000000 / rate;
    rate *= 100.0;
    stcc = rpcc() - stcc;
    printf("i = 7, sysclock = %.2f us, %.4f%% peak\n", timeuse, rate);

    return 0;
}
```

slave.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "simd.h"
#include "slave.h"

#define FUNC3(a, b, c) __builtin_sw64_vmad(a, b, c)

__thread_local volatile unsigned long put_reply;
__thread_local doublev4 va = {1.0, 2.0, 3.0, 0.0}, vb = {4.0, 3.0, 2.0, 1.0}, \
 vaa = {2.0, 3.0, 4.0, 5.0}, vbb = {5.0, 4.0, 3.0, 2.0}, vab = {2.1, 2.3, 2.5, 2.7}, \
 va1 = {3.1, 3.2, 3.3, 3.4}, va2 = {3.4, 1.1, 3.7, 2.4}, va3 = {5.1, 2.2, 2.3, 1.4}, \
 va4 = {1.4, 1.7, 1.85, 2.13}, va5 = {1.43, 1.73, 1.8512, 2.131}, vxx = {0, 0.81, 0.94,
0.92}, vx = {0.91, 0, 0.93, 0.94},  vc = {1.1, 1.2, 1.3, 1.4};

extern double a[];

void func_df36()
{
    int i;

    for(i = 0; i < 105000000; i++)
    {
        va = FUNC3(va, vx, vc);
        vb = FUNC3(vb, vx, vc);
        vaa = FUNC3(vaa, vx, vc);
        vbb = FUNC3(vbb, vx, vc);
        vab = FUNC3(vab, vx, vc);
        va1 = FUNC3(va1, vx, vc);
        va = FUNC3(va, vx, vc);
        vb = FUNC3(vb, vx, vc);
        vaa = FUNC3(vaa, vx, vc);
        vbb = FUNC3(vbb, vx, vc);
        vab = FUNC3(vab, vx, vc);
        va1 = FUNC3(va1, vx, vc);
    }

    va = va + vb + vaa + vbb + vab + va1;

    put_reply = 0;
    athread_put(PE_MODE, &va, &a[0], 4 * sizeof(double), &put_reply, 0, 0);

    while(put_reply!=1);
}

void func_df37()
{
    int i;
```

```
    for(i = 0; i < 90000000; i++)
    {
        va = FUNC3(va, vx, vc);
        vb = FUNC3(vb, vx, vc);
        vaa = FUNC3(vaa, vx, vc);
        vbb = FUNC3(vbb, vx, vc);
        vab = FUNC3(vab, vx, vc);
        va1 = FUNC3(va1, vx, vc);
        va2 = FUNC3(va2, vx, vc);
        va = FUNC3(va, vx, vc);
        vb = FUNC3(vb, vx, vc);
        vaa = FUNC3(vaa, vx, vc);
        vbb = FUNC3(vbb, vx, vc);
        vab = FUNC3(vab, vx, vc);
        va1 = FUNC3(va1, vx, vc);
        va2 = FUNC3(va2, vx, vc);
    }

    va = va + vb + vaa + vbb + vab + va1 + va2;

    put_reply = 0;
    athread_put(PE_MODE, &va, &a[0], 4 * sizeof(double), &put_reply, 0, 0);

    while(put_reply!=1);
}
```

上面代码的运行后的输出是

i = 6, sysclock = 1013683.00 us, 85.7236% peak

i = 7, sysclock = 869009.00 us, 99.9950% peak

所以我们可以得出 SW3 架构双精度浮点 SIMD 乘加的时延是 7，执行周期是 1。

有一些函数的时延不是很好测试，比如位操作和选择操作的时延，主要是由于编译器优化。

题外话：不能利用整数指针别名随便对浮点数随意进行位操作时有必要的，因为一般来说整数和浮点数处理的流水线时不同的，使用指针乱来可能会产生意想不到的问题（会变慢或者容易出错）。我们知道可以将一个 double 给 ^= 0x8000000000000000 来快速取反，但是如果这个 double 值马上就要用到就不能这么干。

DMA

DMA 是 Direct Memory Access 的缩写。在这个框架下定义了一些函数接口，只能够从从核调用。目前，我已经尝试出来的有 PE_MODE, BCAST_MODE, RANK_MODE。

所有的 athread 都能够通过 DMA 正确且稳定地从主核那边读取内存。但是，只有线程 0 地第 0 条 athread 能够相对 stable 地通过 DMA 向主核写内存。

PE_MODE 是单从核模式，在此模式下，给个 DMA 指令只会影响到单个从核。

athread_get 和 athread_put 也只是对 DMA 的包装，执行速度会慢一些。我们平时可以写，只是到时候 final 的时候记得修改成直接的 DMA 语法。

```
    dma_desc dmad_put, dmad_get, dmad_bcast, dmad_rank;
    dma_set_op(&dmad_put, DMA_PUT);
    dma_set_mode(&dmad_put, PE_MODE);
    dma_set_size(&dmad_put, 8 * 10);
    dma_set_reply(&dmad_put, &dma_reply);//dma_reply should be __thread_local volatile int

    dma_set_op(&dmad_get, DMA_GET);
    dma_set_mode(&dmad_get, PE_MODE);
    dma_set_size(&dmad_get, 8 * 10);
    dma_set_reply(&dmad_get, &dma_reply);//dma_reply should be __thread_local volatile int

    long dma_mask = 0xFFFFFFFFFFFFFFED;//low 8 bit effective. ED = 11101101, so row 3 and 6
are not broadcasted
    dma_set_op(&dmad_bcast, DMA_GET);
    dma_set_mode(&dmad_bcast, BCAST_MODE);
    dma_set_size(&dmad_bcast, 8 * 10);
    dma_set_reply(&dmad_bcast, &dma_reply);
    dma_set_mask(&dmad_bcast, dma_mask);

    dma_set_op(&dmad_rank, DMA_GET);
    dma_set_mode(&dmad_rank, RANK_MODE);
    dma_set_size(&dmad_rank, 8 * 48);
    dma_set_reply(&dmad_rank, &dma_bcast_reply);
    dma_set_stepsize(&dmad_rank, 8 * 2);
    dma_set_bsize(&dmad_rank, 8 * 4);

    dma(dmad_put, host_global_ptr1, thread_local_ptr1);
    dma(dmad_get, host_global_ptr2, thread_local_ptr2);
    dma(dmad_bcast, host_global_ptr3, thread_local_ptr3);
    dma(dmad_rank, host_global_ptr4, thread_local_ptr4);
```

dmad_put 的执行结果就是将本 athread 的 thread_local_ptr1 指向的 80B 数据传输到主核的 host_global_ptr1 指向的地方。

dmad_get 的执行结果就是将主核的 host_global_ptr2 指向的 80B 数据传输到本 athread 的 thread_local_ptr2 指向的地方。

dmad_bcast 的执行结果就是将主核的 host_global_ptr3 指向的 80B 数据广播传输传输到除了行 3 和行 6 的所有 athread 的 thread_local_ptr3 指向的地方。

dmad_rank 的执行结果是：从主核的 host_global_ptr4 指向的地方每隔 16bit 取 32bit 数据，并且每块 32bit 的数据都循环地落在 dma 发起者所在行的 thread_local_ptr4 指向的地方。假设 dmad_rank 的发起者是 id = 9 的 athread，指针类型是 double，host_global_ptr4[i] = 1.0 * i，那么最后的结果将会是如下：

	thread_local_ptr4[i]							
线程id	i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7
8	0	1	2	3	48	49	50	51
9	6	7	8	9	54	55	56	57
10	12	13	14	15	60	61	62	63
11	18	19	20	21	66	67	68	69
12	24	25	26	27				
13	30	31	32	33				
14	36	37	38	39				
15	42	43	44	45				

当特定的 DMA 操作完成的时候，dma_reply 会被加上 1。

athread_get 和 athread_put 也只是对 DMA 的包装，执行速度会慢一些。我们平时可以写，只是到时候 final 的时候记得修改成直接的 DMA 语法。

顺便一提，DMA 在内存都是 128Byte 对齐的时候，且传输量是 128Byte 的倍数的时候，效率是最高的。

前面说到，只有 athread_id = 0 的从核能够从稳定地向主核传输 DMA。真相是，如果 DMA 流水线中地最后一个 DMA 命令不是 athread_id = 0 的从核发出的话，那么流水线前面未执行的 DMA 的 put 命令会全部丢失。所以，为了避免这个问题，其实有很简单的解决办法。

从核间同步

```
int total_mask = 0xFFFFFFFF;//low 16 bit effective
int row_mask = 0xFFFFFFFF;//low 8 bit effective
int col_mask = 0xFFFFFFFF;//low 8 bit effective
athread_syn(ARRAY_SCOPE, total_mask);
athread_syn(ROW_SCOPE, row_mask);
athread_syn(COL_SCOPE, col_mask);
```

在相关从核的 athread 调用 athread_syn 可以达到同步的效果。

避免 DMA 流水线丢失的问题，可以执行类似如下的代码：

```
athread_syn(ARRAY_SCOPE, total_mask);
int lpc = 128 - my_id;
for(i = 0; i < 1000 * lpc; i++)
       for(j = 0; j < 100; j++)
          k = trump[trump[k]];
c_slave[1000] = k * 1.0; //preventing compiler optimization
dma_set_op(&dmad_put, DMA_PUT);
dma_set_mode(&dmad_put, PE_MODE);
dma_set_size(&dmad_put, 1000 * 8);
dma_set_reply(&dmad_put, &dma_reply);
dma(dmad_put, &c[0][my_id][0], &c_slave[0]);
```

上面的代码首先同步所有从核，然后通过一个 pointer chasing 来消耗时间，且线程号越小消耗的时间越多。这样，线程号越大就越早调用 DMA_PUT，从而不会导致数据丢失。

MPI 进程模型

一个逻辑非常简单的 demo，实现了什么就不进行解释了，跑一下就可以看到很多 athread 工作的“震撼”场面。

host.c

```
#include <stdlib.h>
#include <stdio.h>
#include <athread.h>
#include <fcntl.h>
#include "simd.h"
#include "memory.h"
#include <unistd.h>
#include <mpi.h>

extern SLAVE_FUN(func)();

#define J 64
#define I 2000
#define K 4

#define mcw MPI_COMM_WORLD

double c[K][J][I];
double a[6400];

void MPIParaMission()
{
    athread_init();
    athread_spawn(func, NULL);
    athread_wait(0);

}

void MPISerialFunc(int my_rank)//this func will be excuted serial, guaranteed by main
function.
{
    int i, j, k;
    for(i = 0; i < 1; i++)
        for(j = 0; j < 64; j++)
            for(k = 2; k < 7; k++)
            {
                printf("%d, %d, %d, %.1f\n", my_rank, j, k, c[i][j][k]);fflush(stdout);
            }
}

int main()
{
    MPI_Init(NULL, NULL);

    int my_rank, comm_sz, namelen;
    MPI_Comm_rank(mcw, &my_rank);
    MPI_Comm_size(mcw, &comm_sz);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(processor_name,&namelen);

    printf("Hello world from proc %d of %d, processor name = %s\n", my_rank, comm_sz,
processor_name);

    int i, j, k;

    for(i = 0; i < 6400; i++)
        a[i] = 1.0 * i + 10000 * my_rank;

    MPIParaMission();

    for(i = 0; i < comm_sz; i++)
    {
        if(i == my_rank)
            MPISerialFunc(my_rank);
        MPI_Barrier(mcw);
    }


    MPI_Finalize();

    return 0;
}
```

slave.c

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "simd.h"
#include "slave.h"
#include "dma.h"

#ifndef __thread_local
//just for obtaining better look on some IDE with error hints
#define __thread_local const
#endif

#define J 64
#define I 2000
#define K 4
__thread_local volatile unsigned long get_reply,put_reply;
__thread_local volatile unsigned long start,end;
__thread_local int my_id, my_core;
__thread_local double c_slave[I];
__thread_local double a_slave[1000];
__thread_local long trump[100];

extern double c[K][J][I];
extern double a[6400];

void func()
{
    dma_desc dmad_put, dmad_get, dmad_barrier;
    int i,j,k;
    int rev;
    my_id = athread_get_id(-1);
    my_core = athread_get_core(-1);
    int dma_mask = 0xFFFFFFFF;

    dma_set_op(&dmad_get, DMA_GET);
    dma_set_mode(&dmad_get, BCAST_MODE);
    dma_set_size(&dmad_get, 8 * 24);
    dma_set_reply(&dmad_get, &get_reply);
    dma_set_mask(&dmad_get, dma_mask);

    long stcc, edcc;

    RPCC(stcc);

    int lpc;
    lpc = 128 - my_id;
    for(i = 0; i < 100; i++)
        trump[i] = (i + 8) % 100;
    k = 0;
    for(i = 0; i < 1000 * lpc; i++)
        for(j = 0; j < 100; j++)
            k = trump[trump[k]];

    int total_mask = 0xFFFFFFFF;
    get_reply = 0;
    if(my_id == 0)
    {
        get_reply = 0;
        dma(dmad_get, &a[0], &a_slave[0]);
        while(get_reply!=1);
    }
    rev = athread_syn(ARRAY_SCOPE, total_mask);

    RPCC(edcc);

    //make sure that 0 goes the last
    for(i = 0; i < 1000 * lpc; i++)
        for(j = 0; j < 100; j++)
            k = trump[trump[k]];

    for(i = 0; i < I; i+=3)
    {
        c_slave[i] = my_id;
```

```
        c_slave[i + 1] = my_core;
        c_slave[i + 2] = (edcc - stcc + i + k) * 1.0;
    }

    for(i = 0; i < 10; i++)
        c_slave[3 + i] = a_slave[i] + 10 * my_id;

    put_reply = 0;

    dma_set_op(&dmad_put, DMA_PUT);
    dma_set_mode(&dmad_put, PE_MODE);
    dma_set_size(&dmad_put, I * 4);
    dma_set_reply(&dmad_put, &put_reply);

    dma(dmad_put, &c[0][my_id][0], &c_slave[0]);

    while(put_reply!=1);
}
```

上面的例子提供了一个基于 MPI 和 athread 的多进程主从核协作范例。

同节点不同 MPI 进程 DMA 内存互读

在运行参数是 bsub -I -b -q q_sw_expr -N 2 -cross -np 4 -cgsp 64 -share_size 4096 -host_stack 128 ./test 的时候，8 个 MPI 进程在 2 个节点上进行，意味着每 4 个 MPI 进程在 1 个节点上，它们理论上可以通过物理地址来相互拷贝内存。当然，操作系统存在的意义就是为了让我们避免这么干，但如果我们偏要这么干呢？

athread 的模式里面有 DMA_GET_P 和 DMA_PUT_P，看一下是怎么使用的吧。

host.c

```
#include <stdlib.h>
#include <stdio.h>
#include <athread.h>
#include <fcntl.h>
#include "simd.h"
#include "memory.h"
#include <unistd.h>
#include <mpi.h>
#include "inttypes.h"

extern SLAVE_FUN(func)();

#define J 64
#define I 2000
#define K 4

#define mcw MPI_COMM_WORLD

double c[K][J][I];
double a[6400];
long a_paddr, a_paddr2;

uintptr_t vtop(uintptr_t vaddr)
{
    FILE *pagemap;
    intptr_t paddr = 0;
    int offset = (vaddr / sysconf(_SC_PAGESIZE)) * sizeof(uint64_t);
    uint64_t e;

    if ((pagemap = fopen("/proc/self/pagemap", "r")))
```


```
    {
        if (lseek(fileno(pagemap), offset, SEEK_SET) == offset)
        {
            if (fread(&e, sizeof(uint64_t), 1, pagemap))
            {
                if (e & (1ULL << 63))
                { // page present ?
                    paddr = e & ((1ULL << 54) - 1); // pfn mask
                    paddr = paddr * sysconf(_SC_PAGESIZE);
                    // add offset within page
                    paddr = paddr | (vaddr & (sysconf(_SC_PAGESIZE) - 1));
                }
            }
        }
        fclose(pagemap);
    }
    return paddr;
}

void MPIParaMission(int my_rank)
{
    a_paddr = vtop(&a[0]);
    int dst = my_rank ^ 0x00000001;
    MPI_Sendrecv(&a_paddr, 1, MPI_DOUBLE, dst, 0, &a_paddr2, 1, MPI_DOUBLE, dst, 0, mcw,
MPI_STATUS_IGNORE);
    a_paddr = a_paddr2;

    athread_init();
    athread_spawn(func, NULL);
    athread_wait(0);
}

void MPISerialFunc(int my_rank)//this func will be excuted serial, guaranteed by main
function.
{
    int i, j, k;
    for(i = 0; i < 1; i++)
        for(j = 0; j < 64; j++)
            for(k = 2; k < 7; k++)
            {
                printf("%d, %d, %d, %.1f\n", my_rank, j, k, c[i][j][k]);fflush(stdout);
            }
}

int main()
{
    MPI_Init(NULL, NULL);

    int my_rank, comm_sz, namelen;
    MPI_Comm_rank(mcw, &my_rank);
    MPI_Comm_size(mcw, &comm_sz);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(processor_name,&namelen);

    printf("Hello world from proc %d of %d, processor name = %s\n", my_rank, comm_sz,
processor_name);

    int i, j, k;

    for(i = 0; i < 6400; i++)
        a[i] = 1.0 * i + 10000 * my_rank;

    MPIParaMission(my_rank);

    for(i = 0; i < comm_sz; i++)
    {
        if(i == my_rank)
            MPISerialFunc(my_rank);
        MPI_Barrier(mcw);
    }


    MPI_Finalize();

    return 0;
}
```

slave.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "simd.h"
#include "slave.h"
#include "dma.h"

#ifndef __thread_local
//just for obtaining better look on some IDE with error hints
#define __thread_local const
#endif

#define J 64
#define I 2000
#define K 4
__thread_local volatile unsigned long get_reply,put_reply;
__thread_local volatile unsigned long start,end;
__thread_local int my_id, my_core;
__thread_local double c_slave[I];
__thread_local double a_slave[1000];
__thread_local long trump[100];

extern double c[K][J][I];
extern double a[6400];
extern long a_paddr;

void func()
{
    dma_desc dmad_put, dmad_get;
    int i,j,k;
    int rev;
    my_id = athread_get_id(-1);
    my_core = athread_get_core(-1);
    int dma_mask = 0xFFFFFFFF;

    dma_set_op(&dmad_get, DMA_GET_P);
    dma_set_mode(&dmad_get, BCAST_MODE);
    dma_set_size(&dmad_get, 8 * 24);
    dma_set_reply(&dmad_get, &get_reply);
    dma_set_mask(&dmad_get, dma_mask);

    long stcc, edcc;

    RPCC(stcc);

    int lpc;
    lpc = 128 - my_id;
    for(i = 0; i < 100; i++)
        trump[i] = (i + 8) % 100;
    k = 0;
    for(i = 0; i < 1000 * lpc; i++)
        for(j = 0; j < 100; j++)
            k = trump[trump[k]];

    int total_mask = 0xFFFFFFFF;
    get_reply = 0;
    if(my_id == 0)
    {
        get_reply = 0;
        dma(dmad_get, a_paddr, &a_slave[0]);// !!!!!!!!
        while(get_reply!=1);
    }
    rev = athread_syn(ARRAY_SCOPE, total_mask);

    RPCC(edcc);

    //make sure that 0 goes the last
    for(i = 0; i < 1000 * lpc; i++)
        for(j = 0; j < 100; j++)
            k = trump[trump[k]];

    for(i = 0; i < I; i+=3)
    {
```

```
        c_slave[i] = my_id;
        c_slave[i + 1] = my_core;
        c_slave[i + 2] = (edcc - stcc + i + k) * 1.0;
    }

    for(i = 0; i < 10; i++)
        c_slave[3 + i] = a_slave[i] + 10 * my_id;

    put_reply = 0;

    dma_set_op(&dmad_put, DMA_PUT);
    dma_set_mode(&dmad_put, PE_MODE);
    dma_set_size(&dmad_put, I * 4);
    dma_set_reply(&dmad_put, &put_reply);

    dma(dmad_put, &c[0][my_id][0], &c_slave[0]);

    __asm __volatile ("":::"memory");

    while(put_reply!=1);
}
```

这份代码和前面那份 MPI 的入门 demo 只有少量不同，可以对比运行以下看看有什么不同。

vtop 函数将虚拟地址转化为物理地址。然后，相邻 MPI 进程通过 sendrecv 交换物理地址，然后在从核里面通过 DMA 用物理地址来读取其他进程主核的内存。

从核间通信

从核的排列是 8*8 的，每一行和每一列都有共享的寄存器，从核间可以利用这些寄存器进行通信。

以下部分的代码可以实现将从核 0 的数据广播到所有从核。

```
    __thread_local double localdv[100];

    int my_core = athread_get_core(-1);
    for(i = 0; i < 100; i++)
        localdv[i] = 2333000 + my_core;

    volatile doublev4* loadrv4p;
    volatile doublev4* loadrv4p2;
    int col_dst = 0x0000000F;//column brocast
    int row_dst = 0x0000000F;//row brocast
    loadrv4p = &a_slave[4];
    loadrv4p2 = &localdv[0];
    if(my_core / 8 == 0)
        __builtin_sw64_putC(*loadrv4p2, col_dst);
    else
        *loadrv4p2 = __builtin_sw64_getxC(*loadrv4p2);

    if(my_core % 8 == 0)
    {
        asm volatile ("nop":::"memory");
        __builtin_sw64_putR(*loadrv4p2, row_dst);
        *loadrv4p = *loadrv4p2;
    }
    else
    {
        asm volatile ("nop":::"memory");
        *loadrv4p = __builtin_sw64_getxR(*loadrv4p2);
    }
```

这段代码的结果是，将所有从核的 a_slave[4 : 7]赋值为从核 0 的 localdv[0 : 3]。实现方式是，首先进行列广播，将值传到第 0 列，然后再进行行广播。列广播和行广播采用的源地址不能使目标地址，否则会出现广播失败的 bug。

FORTRAN 和 C 混编模型

假设原来的 FORTRAN 代码是：

```
program main
    implicit none

    real*8 :: arr(64000)

    call doge(arr)

    print *, arr(1), arr(1001), arr(63001)

end

subroutine doge(arr)
    implicit none

    real*8 :: arr(64000)
    integer :: i

    do i = 1, 64000
        arr(i) = i
    end do

end subroutine doge
```

我们的目标是将 subroutine doge 用 C 实现，并且利用上从核的计算能力。

最终代码：

doge.f90

```
program main
    implicit none

    real*8 :: arr(64000)
    call c_doge(arr)
    print *, arr(1), arr(1001), arr(63001)
end
```

c_doge.c

```
#include <athread.h>
#include <stdio.h>
#include "memory.h"

double sarr[64000];

extern SLAVE_FUN(func)();

void c_doge_(double* arr)
{
    printf("enter host c function\n");
    athread_init();
    athread_create(0, func, NULL);
```

```
    athread_wait(0);

    memcpy(arr, &sarr[0], 64000 * 8);
}
```

slave.c

```
#include "slave.h"
#include "dma.h"
#include "stdio.h"

__thread_local double localarr[1000];
__thread_local volatile int dma_reply;

extern double sarr[64000];

void func()
{
    dma_desc dmad_put;
    int i, p;

    dma_reply = 0;

    dma_set_op(&dmad_put, DMA_PUT);
    dma_set_mode(&dmad_put, PE_MODE);
    dma_set_size(&dmad_put, 1000 * 8);
    dma_set_reply(&dmad_put, &dma_reply);

    for(p = 0; p < 64; p++)
    {
        for(i = 0; i < 1000; i++)
            localarr[i] = i + 1 + p * 1000;
        dma(dmad_put, &sarr[p * 1000], &localarr[0]);
    }

    /*for(i = 0; i < 1000000; i++)
        localarr[i % 3 + 500] -= localarr[(i + 1) % 3 + 500];
    printf("reply is %d\n", dma_reply);*///you can try how these three lines will behave
with or without "volatile"
    while(dma_reply!=64);
}
```

Makefile

```
EXE=test
CC=mpicc
FC=mpif90
SCC=sw5cc
FFLAGS = -O3
HFLAGS = -msimd -O3
SFLAGS = -slave -msimd -O3
LDFLAGS= -O3
SRC = doge.f90 c_doge.c slave.c

test: doge.o c_doge.o slave.o
        $(FC) $(LDFLAGS) *.o  -o $@

doge.o: doge.f90
        $(FC) -c $(FFLAGS) doge.f90

c_doge.o: c_doge.c
        $(CC) -c $(HFLAGS) c_doge.c

slave.o: slave.c
        $(SCC) -c $(SFLAGS) slave.c

clean:
        rm *.o
```

代码很好读，就不分析了。要注意的是，fortran 调用的 subroutine 的符号后面有加上下划线，所以在 c 里面函数名是 c_doge_。

主从核 ping-pong

doge.f90

```
program main
    implicit none

    real*8 :: arr(64000)
    call c_doge(arr)
    print *, arr(1), arr(1001), arr(63001)
end
```

c_doge.c

```
#include <athread.h>
#include <stdio.h>
#include "memory.h"

double sarr[64000];
volatile int host_flag[32];
volatile int slave_flag[32];

extern SLAVE_FUN(func)();
#define FREQSC 1450000000

static inline unsigned long rpcc()
{
    unsigned long time;
    asm("rtc %0": "=r" (time) : );
    return time;
}

void c_doge_(double* arr)
{
    printf("enter host c function\n");

    long stcc, edcc;
    athread_init();
    int i;

    athread_spawn(func, NULL);
    stcc = rpcc();

    for(i = 0; i < 1000000; i++)
    {
        host_flag[0] = i + 1;
        while(1)
        {
            if(slave_flag[0] == i + 1)
                break;
        }
        //printf("host get flag %d\n", i + 1);
    }

    edcc = rpcc() - stcc;
    athread_join();

    double sec = edcc * 1.0 / FREQSC;

    printf("cc = %ld, sec = %.4f\n", edcc, sec);

    memcpy(arr, &sarr[0], 64000 * 8);
}
```

slave.c

```c
#include "slave.h"
#include "dma.h"
#include "stdio.h"

extern int host_flag[32];
extern int slave_flag[32];
__thread_local volatile int local_flag[32];
__thread_local volatile int put_reply, get_reply;

void func()
{
    int my_core = athread_get_core(-1);
    if(my_core != 0)
        return;

    dma_desc dmad_put, dmad_get;

    dma_set_op(&dmad_get, DMA_GET);
    dma_set_mode(&dmad_get, PE_MODE);
    dma_set_size(&dmad_get, 4 * 32);
    dma_set_reply(&dmad_get, &get_reply);

    dma_set_op(&dmad_put, DMA_PUT);
    dma_set_mode(&dmad_put, PE_MODE);
    dma_set_size(&dmad_put, 4 * 32);
    dma_set_reply(&dmad_put, &put_reply);

    int i;
    for(i = 0; i < 1000000; i++)
    {
        while(1)
        {
            get_reply = 0;
            dma(dmad_get, &host_flag[0], &local_flag[0]);
            while(get_reply != 1);
            if(local_flag[0] == i + 1)
                break;
        }//get the flag of the host finishing sending "i = 0"
        //printf("slave get flag %d\n", i + 1);
        put_reply = 0;
        dma(dmad_put, &slave_flag[0], &local_flag[0]);
        while(put_reply != 1);
    }
}
```

上面的示例实现了主从核间的实时交互。可以使用这个模型进行主从核间通信，执行复杂的任务。