Application Report
ZHCA568 – June 2013

OpenEM 简介和基于 OpenEM 的大矩阵乘实现

James Li
Multi-core DSP / FAE

摘要

OpenEM 的全称是 Open Event Machine。是 TI 针对嵌入式应用开发的 multicore runtime system library。OpenEM 可以在多核上有效的调度，分发任务。它把任务调度给负载轻的核，进而实现动态的负载平衡。OpenEM 是基于 TI Keystone 系列芯片的 multicore Navigator 构建的，具有开销小，效率高的特点。本文首先对 OpenEM 的原理做了简单的介绍。然后结合一个大矩阵乘的演示用例详细介绍了 OpenEM 的使用。最后通过量化分析这个演示用例的执行 cycle 数，总结了 OpenEM 的效率和局限。希望本文能成为学习 OpenEM 的读者的一个有用的参考。

文档修改纪录

Version	Date	Author	Notes
1.0	April 2013	James Li	First release
1.2	June 2013	James Li	Updated base on the review comments

目录

OpenEM 简介和基于 OpenEM 的大矩阵乘实现 1
1 OpenEM 简介 3
1.1 OpenEM 的软件对象 3
1.2 OpenEM 的两个重要概念 4
1.3 OpenEM 的常用 API cycle 数 5
2 基于 OpenEM 的大矩阵乘实现 5
2.1 基于 OpenEM 的大矩阵乘方案设计 6
2.1.1 Memory 使用 6
2.1.2 处理流程 7
2.2 基于 OpenEM 的大矩阵乘实现 9
2.2.1 OpenEM 的 Global 初始化 9
2.2.2 创建生产者/消费者场景 10
2.2.3 产生 event 12
2.2.4 运行和 exit 13
2.3 基于 OpenEM 的大矩阵乘性能测试结果 14
2.3.1 算法代码和 cycle 数的理论极限 14
2.3.2 基于 OpenEM 的性能测试结果 14
3 总结 15

表

表 1 OPENEM 的主要软件对象列表 3
表 2 OPENEM 常用 API 的 CYCLE 数 5

图

图 1 OPENEM 软件对象关系图 4
图 2 大矩阵乘示意图 6
图 3 大矩阵乘数据缓冲区位置示意图 7
图 4 大矩阵乘处理流程图 8
图 5 大矩阵乘 EVENT 到 EXECUTE OBJECT 的映射关系图 11

1 OpenEM 简介

OpenEM 的全称是 Open Event Machine。它是 TI 开发的可应用于 Keystone 多核 DSP 的 multicore runtime system library。OpenEM 的目的是在多核上有效的调度，分发任务，实现动态的负载平衡。基于 OpenEM，用户可以很容易的把原来的单核应用移植到 Keystone 多核芯片。需要注意的是 OpenEM 目前只能把任务调度分发到同一个 DSP 的多个核上，不能跨 DSP 调度分发。 OpenEM 不依赖于 BIOS。它可以在芯片上裸跑，代码精简，效率高。而且，OpenEM 不同于业界已经有 OpenMP 和 OpenCL 等开放式的 multi-core runtime systems。它是针对嵌入式系统的设计，更能满足嵌入式设计的实时性要求。TI 的 keystone 架构多核芯片中有 Multicore Navigator。它由 Queue Manager(简称为 QMSS)和一系列 Packet DMA engine 构成。OpenEM 就是基于这套硬件系统构建的。例如，OpenEM 的 scheduler 是运行在 QMSS 的 PDSP(QMSS 内部的 RISC 处理器)上的。OpenEM 的 preload 功能是通过 QMSS 的 packet DMA 实现的。熟悉 QMSS 的编程对学习 OpenEM 很有帮助。OpenEM 是 MCSDK 的一个组件。它还在不断的发展改进中。本文对 OpenEM 的介绍以及演示用例都是基于 BIOS MCSDK 2.01.02 的 OpenEM 1.0.0.2。

1.1 OpenEM 的软件对象

下面通过列表和图示介绍了 OpenEM 的主要软件对象。表 1 是 OpenEM 的主要软件对象的列表。

表 1 OpenEM 的主要软件对象列表

软件对象名	含义
Event	Event 表征的是一个待处理的数据块。在 OpenEM 中，event 在物理上是一个 host descriptor 以及它指向的 buffer
Queue group	每个 Queue group 有一个 core mask，core mask 表征哪些 DSP 核可以用来处理属于这个 queue group 的 queue 输入的 event
Queue	每个 Queue 都属于一个 queue group。Queue 的两个最重要属性是优先级和 atomic 属性。Queue 的优先级决定了从这个 queue 输入 event 被调度的先后。如果一个 queue 是 Atomic queue，那么任何时候系统中都只能有一个来自这个 queue 的 event 被处理。与 atomic queue 相对的是 parallel queue。parallel queue 的多个 event 可以在多个 DSP 核上并行处理。需要注意的是：这里所说的 queue 是一个软件对象，而不是 QMSS 的 hardware queue。这里的 queue 也不是一个软件链表。它的主要作用是把 event 和 execution object 映射起来。这样发送到 queue 的 event 就被所映射的 execution object 的 receive 函数处理。从图 1 可以看出，一个 queue 只能对应一个 execution object，但是一个 execution object 可以被多个 queue 映射。例如 execution object 1 被映射到 queue 1，2，3。
Scheduler	调度器，可以运行在 PDSP(QMSS 内部的 RSIC 处理器，6678 的 QMSS 有 2 个 PDSP)，也可以运行在 C66x 核上。如果调度器运行在 PDSP，那么称为异步调度，如果调度器运行在 C66x 核上，那么称为同步调度。
Dispatcher	Dispatcher 运行在每个 DSP 核上，负责查询待处理的 event 并调用相应的 receiver 函数。 Dispatcher 有 co-operative 和 run to completion 两种运行模式。co-operative 模式的 dispatcher 可以调用 suspend API 把自己挂起，run to completion 没有这个功能。
Execution object	Execution object 是个结构体，主要用来封装处理 event 的 receive 函数的指针。

Receive function	用来处理接收到的 event 数据的用户函数，由应用实现

需要注意的是，本文介绍的 OpenEM 的运行模式是：Scheduler 运行在 PDSP，Dispatcher 是“run to completion ”模式。

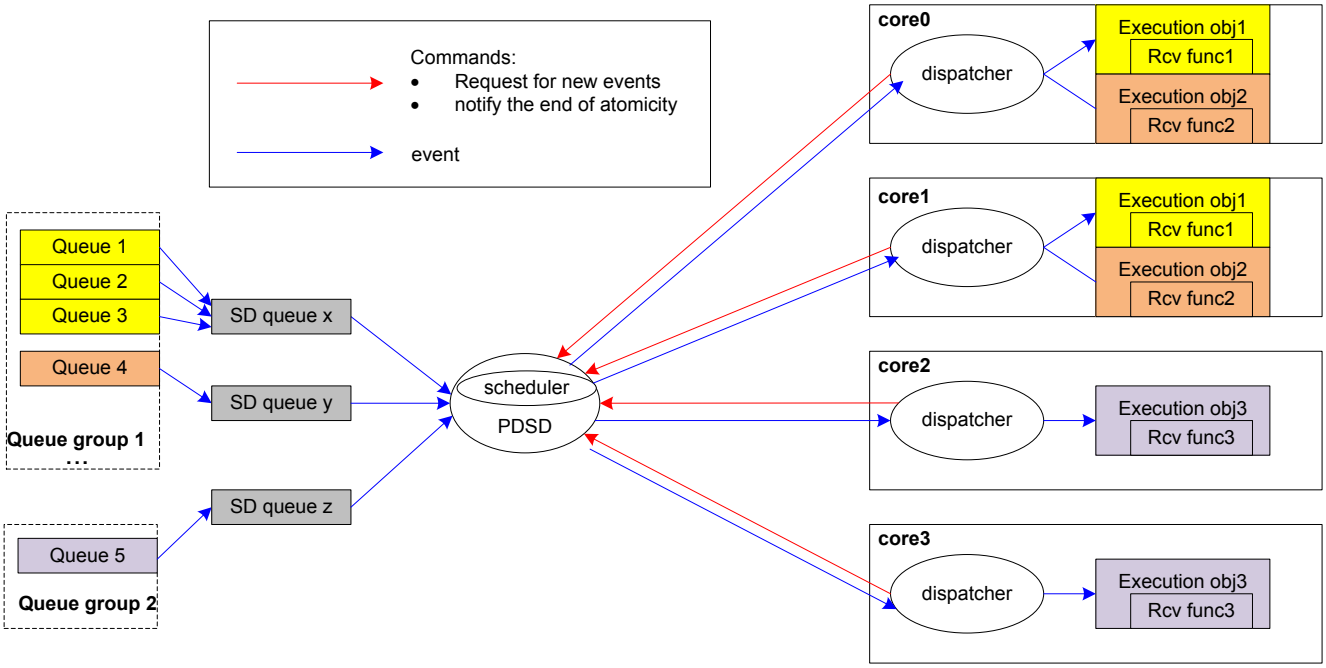

图 1 OpenEM 软件对象关系图

图 1 是一个软件对象关系图，显示出了表 1 中列举的软件对象。定义了 2 个 queue group，5 个 queue 和 3 个 execution object。Queue group1 的 core mask 对应核 0 和 1。所以来自 queue1，2，3，4 的 event 只能在核 0 和核 1 上执行，因为这些 queue 属于 queue group1。Queue group2 的 core mask 对应核 2 和 3。所以来自 queue5 的 event 只能在核 2 和核 3 上执行，因为 queue5 属于 queue group2。execution object 1 和 queue 1,2,3 映射关联。execution object 2 和 queue 4 映射关联。execution object 3 和 queue 5 映射关联。图中的蓝线表示了 event 的行径，红线表示 command 的行径。图中的 SD queue 是 hardware queue，它不是一个软件对象而是 OpenEM 内部的组件。

1.2 OpenEM 的两个重要概念

OpenEM 中有两个容易混淆的重要概念：prefetch 和 preload。

- Prefetch 是指每个 DSP 核向 scheduler 发命令，告诉 scheduler “本核已经空闲了，可以分配新的工作给本核了”。只有收到一个核的 prefetch 命令，scheduler 才会调度新的 event 给这个核。如果 DSP 核不发出 prefetch 命令，它就不会被分派任务。这是 OpenEM 的 scheduler 的基本调度原则。

- Preload 和 event 的属性有关。通常，event 的数据是位于 DDR 的。如果 DSP 核直接访问 DDR 效率会比较低。所以，OpenEM 可以把 event 的数据通过 QMSS 的 packet DMA 搬到 DSP 核的 local L2。这个搬移的过程就是 preload。每个 event 的数据是否做 preload 是可配的。每个 event 在创建的时候都可以指定一个 preload 属性。Event 的 preload 属性可以是：

- Preload disable, 即不做预搬移
- Preload up to sizeA，即做预搬移，但是最多只搬 sizeA bytes
- Preload up to sizeB，即做预搬移，但是最多只搬 sizeB bytes
- Preload up to sizeC，即做预搬移，但是最多只搬 sizeC bytes
- 其中 SizeA，SizeB 和 SizeC 是常数，在 OpenEM 初始化的时候可以配置。

1.3 OpenEM 的常用 API cycle 数

OpenEM 的附带开销是应用最关注特性之一。所以我们实测了 OpenEM 常用 API 的 cycle 数如表 2。需要注意的是：由于 OpenEM 会负责 cache 一致性的维护，而有些 API 的处理过程中含有 cache 一致性的维护操作。所以这些 API 的调用 cycle 数很大程度上取决于它对多大的数据缓冲区做了 cache 一致性的维护。本文测试这些 cycle 的场景使用的数据缓冲区的大小是是 4096 words(32bit)。

表 2 OpenEM 常用 API 的 cycle 数

API 函数名	C66x Cycle 数	备注
ti_em_preschedule	40	发 prefetch 命令给 scheduler
em_alloc	200~300	从 free pool 申请一个空的 event
em_free	40	释放 local event，不需要维护 cahe 一致性
	2000	释放 global event，要 invalidate 4096 words。
em_send	5000	需要 write back 并且 invalidate 整个数据缓冲区（4096 words）。因为 em_send 要负责维护 cache 一致性。
从发出 prefetch command 到收到新调度的 event 的延迟	39000	Preload 了 4096 words 的数据缓冲区
	1200	没有 preload

2 基于 OpenEM 的大矩阵乘实现

大矩阵相乘的目的是计算 X*Y = Z

矩阵 X 是(100 × 2048)的浮点实数矩阵。

矩阵 Y 是(2048 × 2048)的浮点实数矩阵。

矩阵 Z 是(100 × 2048)的浮点实数矩阵。

由于矩阵 Y 的数据量很大，所以在多核 DSP 上可以把它拆分成多个子块，交给多个 DSP 核并行计算。如图 2 所示。

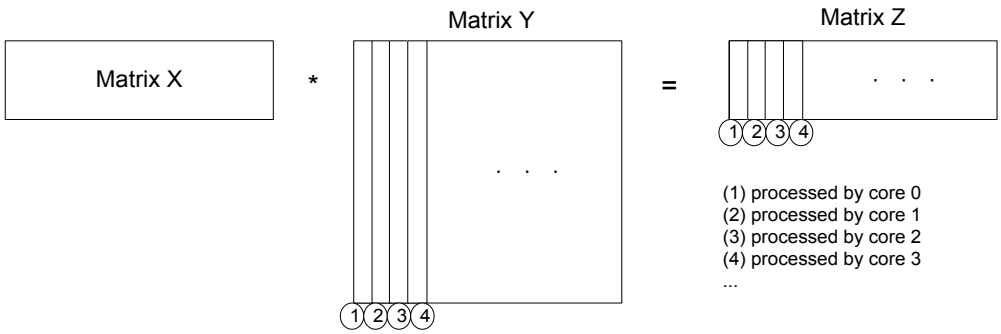

图 2 大矩阵乘示意图

2.1 基于 OpenEM 的大矩阵乘方案设计

2.1.1 Memory 使用

Shannon DSP (6678)的内存系统包括片内的 LL2(local L2)和 SL2(shared L2)。加上片外的 DDR。LL2 的 size 是 512 Kbytes，每个核有一份 LL2。 SL2 的 size 是 4Mbytes，8 个核共享 SL2。DDR size 和硬件板卡设计有关，一般在 1G bytes 以上。 C66x 核对 LL2 的访问效率最高，对 SL2 的访问效率稍差，对 DDR 的访问效率最低。基于多种存储区间的不同特性，我们对数据存储位置按如下规划(参见图 3)：

- 矩阵 X 的 size 是 800 Kbytes，存储是 shared L2
- 矩阵 Y 的 size 是 16 Mbytes，存储是 DDR
- 矩阵 Z 的 size 是 800 Kbytes，存储是 shared L2

虽然矩阵 Y 存储在 DDR，但是我们启用了 OpenEM 的 preload 功能。Preload 就是通过 QMSS 的 packet DMA 把待处理的 event 数据(通常位于 DDR)搬到被调度 core 的 LL2。所以 DSP 核运行的时候不直接从 DDR 取数。这保证了 DSP 核的数据访问效率。

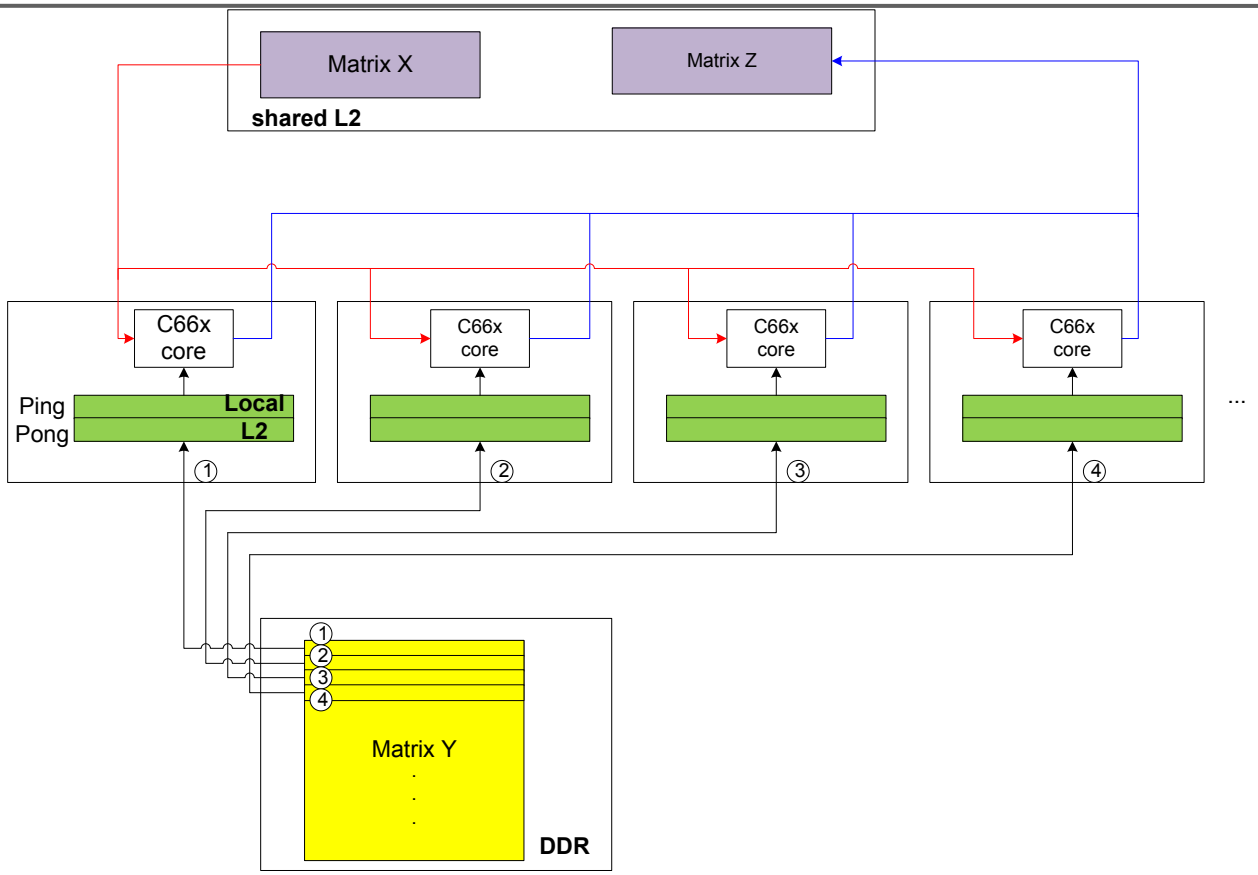

图 3 大矩阵乘数据缓冲区位置示意图

2.1.2 处理流程

OpenEM 中要有一个 DSP 核作为主核，其他核就是从核，主核要完成的工作较多。本文的演示用例中，核 0 是主核，核 1～7 是从核。主从核的分工差异如图 4：

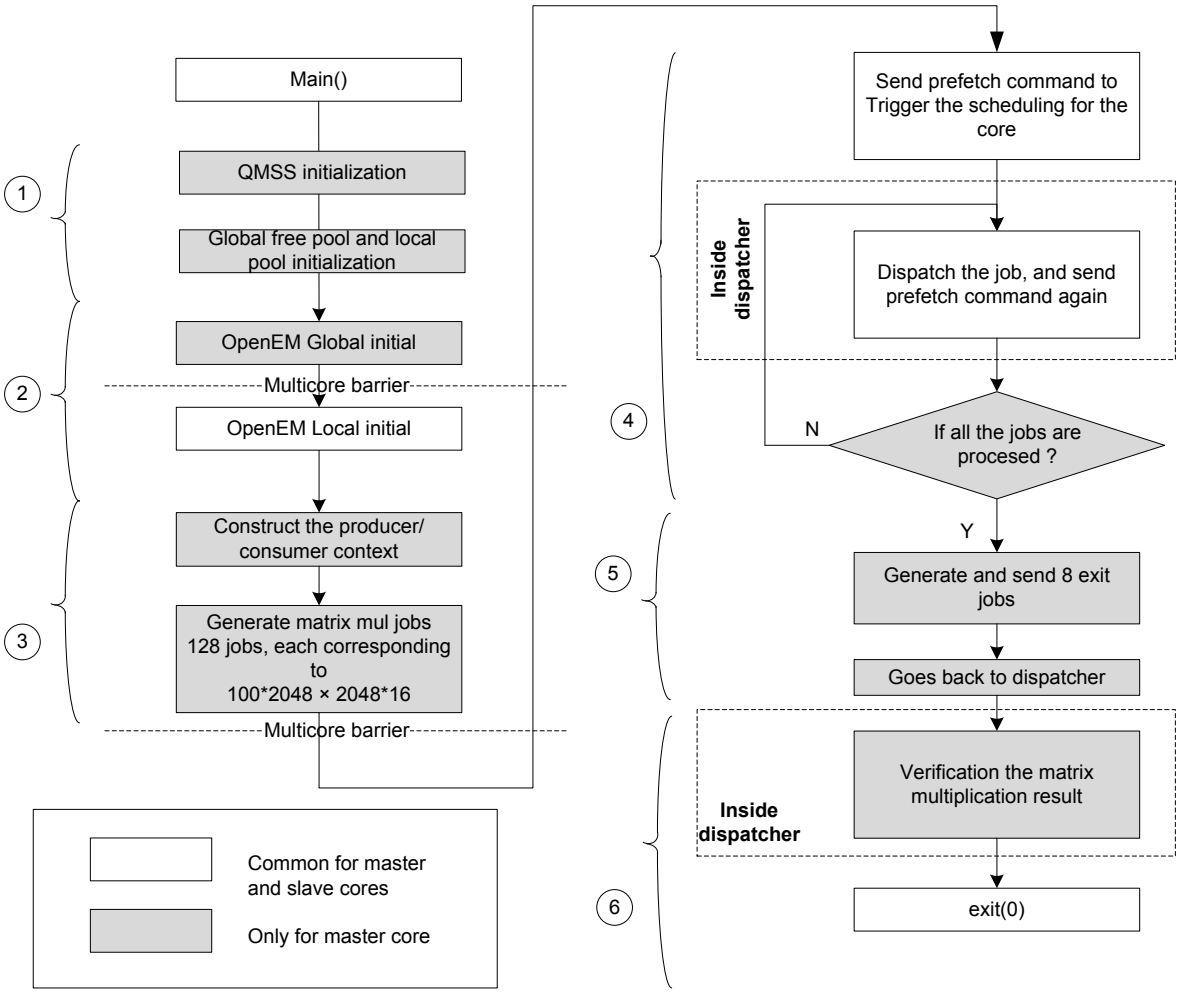

图 4 大矩阵乘处理流程图

1. 初始化 QMSS 和 free pool。
2. OpenEM 的 global 初始化和 local 初始化。global 初始化是主核执行。local 初始化是每个核各自执行。Local 初始化要等 global 初始化完成才能开始。所以，中间需要加一个 barrier。Barrier 可以理解成一个同步点，所有 DSP 核在这个点完成一次同步再继续向下执行。本演示用例的 Barrier 是通过共享内存的软件信号量实现的。
3. 主核构造生产者/消费者场景并产生待处理的 event。生产者在 OpenEM 中不是一个软件对象。我们可以把产生 event 并发送到 queue 的函数认为是生产者。消费者就是 execution object，沟通生产者和消费者的管道就是 queue。构造生产者/消费者场景就是创建 execution object 和 queue 并且把它们关联起来。
4. 主核和从核进入 event 处理的过程。
5. 主核检测到所有 event 都处理完成后为每个 DSP 核（包括它自己）产一个 exit job。
6. 主核和从核处理 exit job。从核直接调用 exit(0)退出。主核先做结果验证然后调用 exit(0)退出。

本文演示用例实现的几个特点是：

- OpenEM 的 free pool 是由用户初始化的。在初始化 free pool 的时候 event 描述符不指向数据缓冲区。等分配了一个 event 的时候再在这个 event 对应的描述符上挂数据缓冲区。这样可以避免不必要的数据拷贝(从 global buffer 拷贝到 event buffer)。
- 主核通过查询 free pool 中的 event 个数是否恢复回初始值来判断是否所有“矩阵乘 event”都处理。因为：
  - Free pool 在初始化以后有 N 个 free event，
  - 从中分配了若干个 event 后，free event 就减少了相应的个数，
  - 每个 core 每处理完一个 event 就把这个 event 回收到 free pool，free pool 的 event 个数就加一。当 free pool 的 event 个数恢复回 N，就说明所有 event 都处理完了。

2.2 基于 OpenEM 的大矩阵乘实现

在初始化 OpenEM 之前首先要做 multicore Navigator 的初始化。包括：PDSP firmware 的 download， Link RAM 的初始化， Memory region 的初始化还有 free pool (也就是 free descriptor queue)的初始化。这不属于本文介绍的范畴，本文直接介绍 OpenEM 的初始化。

2.2.1 OpenEM 的 Global 初始化

OpenEM 的 global 初始化通过调用 API 函数 ti_em_init_global()完成的。这个 API 的入参是下面所示的结构体。其中所列的参数是本文的演示用例使用的配置参数。本文针对每个参数的作用做了注释。了解了参数了含义，就能了解 OpenEM 的 global 初始化的大致做了些什么。

```
ti_em_hw_config_t my_hwCfg =
{
    1024,                           /* hw_queue_base_idx，参见注释 1   */
    3,                              /* hw_sem_idx           参见注释 2 */
    Qmss_MemRegion_MEMORY_REGION8,  /* private_region_idx   */
    0,                              /* dma_idx              参见注释 3 */
    800,                            /* dma_queue_base_idx 参见注释 4 */

    /* local_free_queue_idx_tbl[TI_EM_CORE_NUM] */
    {
        core0_localPoolFdqIdx,  /* 参见注释 5 */
        core1_localPoolFdqIdx,
        core2_localPoolFdqIdx,
        core3_localPoolFdqIdx,
        core4_localPoolFdqIdx,
        core5_localPoolFdqIdx,
        core6_localPoolFdqIdx,
        core7_localPoolFdqIdx
    },

    2,                              /* pool_num 参见注释 6 */
                                    /* pool_config_tbl[TI_EM_POOL_NUM] */
    {
/*free_queue_idx*/     /*coh_mode*/      /*buf_mode*/       /*free_push_policy*/
  { globalFreePoolFdqIdx, TI_EM_COH_MODE_ON,  TI_EM_BUF_MODE_GLOBAL_TIGHT,      0},
  { exitPoolFdqIdx      , TI_EM_COH_MODE_OFF, TI_EM_BUF_MODE_GLOBAL_TIGHT,      0}
    },
        256,                            /* preload_size_a，参见注释 7 */
```

```
        2048,                           /* preload_size_b */
        LOCAL_POOL_BUFSIZE              /* preload_size_c */
    };
    typedef enum{
        globalFreePoolFdqIdx  = 2048,
        exitPoolFdqIdx        = 2049,
        core0_localPoolFdqIdx = 2050,
        core1_localPoolFdqIdx = 2051,
        core2_localPoolFdqIdx = 2052,
        core3_localPoolFdqIdx = 2053,
        core4_localPoolFdqIdx = 2054,
        core5_localPoolFdqIdx = 2055,
        core6_localPoolFdqIdx = 2056,
        core7_localPoolFdqIdx = 2057
    }SEnumFdqIdx;
```

注释：

1. OpenEM 要使用 hardware queue 资源。hw_queue_base_idx 用来指定 OpenEM 从哪个 hardware queue 开始可用。

2. OpenEM 的少量操作需要多 DSP 核访问共享的数据结构。是通过 hardware semaphore 实现多核 lock/unclock 的。所以通过 hw_sem_idx 告诉 OpenEM 该使用哪一个 hardware semaphore。

3. 指定 preload 使用的 QMSS packet DMA 的通道的起始索引。QMSS packet DMA 有 32 个 RX/TX channel。在 OpenEM 中，每个 DSP core 要占用一个 TX/RX channel。

4. 指定 preload 使用的 QMSS Tx queues 的起始索引。要和 dma_idx 对应起来。QMSS 有 32 个 TX queue，索引是 800～831。对应 QMSS packet DMA 的 TX channel 0～31。所以，如果前面配置的 dma_idx 是 0，那么这里配置的 dma_queue_base_idx 应该是 800。

5. 指定 OpenEM local free pool 对应的 free queue index。Local free pool 是和 preload 相关的。local free pool 在物理上是一个 free descriptor queue。里面存储着 2 个 host 描述符。每个描述符对应一个 local L2 buffer。如果发生 preload，packet DMA 就从 free descriptor queue pop 描述符，然后把数据传到描述符指向的 local L2 buffer。每个 DSP 核有一个 local free pool。例如，在我们的演示用例中 core0～7 对应的 free descriptor queue 索引是 2050～2057。

6. 指定 OpenEM global free pool 的个数。每个 global free pool 包括 4 个初始化参数，例如 `{ globalFreePoolFdqIdx, TI_EM_COH_MODE_ON,TI_EM_BUF_MODE_GLOBAL_TIGHT,0}`。参数 1 是这个 global free pool 对应的 free queue index。接下来几项是这个 pool 中的 buffer 的属性。Global free pool 是用来从中分配 free event 的。调用 em_alloc()的入参之一就是 free pool index。

7. 配置 preload 门限，参见本文 1.2 节的叙述。

2.2.2 创建生产者/消费者场景

前面介绍过，在 OpenEM 中，消费者就是 execution object，沟通生产者和消费者的管道就是 queue。本小节介绍怎样创建 execution object 和 queue 以及怎样把它们关联起来。 关于怎样产生 event，本文在下一小节描述。OpenEM 有下列 API 供应用调用：

- 调用 em_eo_create()可以创建 execution object

- 调用 em_queue_create()可以创建 queue
- 调用 em_eo_add_queue()可以把 queue 和 execution object 映射起来

本演示用例通过参数配置表列出 execution object, queue group object 和 queue object 的参数，然后通过解析函数解析配置表再调用 OpenEM 的 API，这样各个软件对象的参数在配置表中一目了然，代码的可读性较好。图 5 是本演示用例的映射关系。

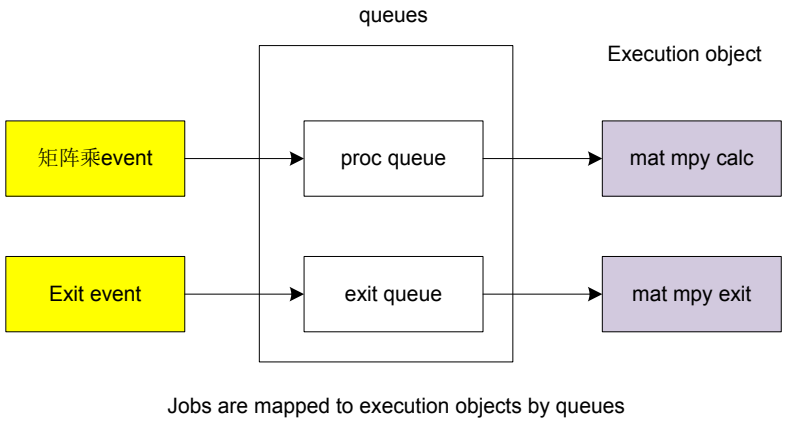

图 5 大矩阵乘 Event 到 Execute Object 的映射关系图

```
/* queue group 配置参数列表 */
SDQueGrpObjParams QueGrpObjList[]={
       /*name*/                /*mask*/
    {"core0123_grp", {0x00,0x00,0x00,0x00,0x0f,0x00,0x00,0x00}},
    {"core4567_grp", {0x00,0x00,0x00,0x00,0xf0,0x00,0x00,0x00}},
    {"allCores_grp", {0x00,0x00,0x00,0x00,0xff,0x00,0x00,0x00}}
};
```

需要注意的是 coremask 总共有 64 个比特，但是目前 6678 最多也只有 8 个 DSP 核。所以大量 mask 比特是用不到的，目前。核 0～7 对应的 mask 比特是位于 byte[4]的 bit0：7

```
/* queue 配置参数列表 */
SDQueueObjParams  QueueObjList[]={
       /*name*/        /*type*/           /*prio*/  /*group*/
    {"proc queue", EM_QUEUE_TYPE_PARALLEL,   14,   queGrpIdx_coreAll,
    {"exit queue", EM_QUEUE_TYPE_PARALLEL,   12,   queGrpIdx_coreAll ,

      /*receiver*/      /* queue handler */
    (void*)mat_mpy_calc , NULL},
    (void*)mat_mpy_exit , NULL}
};
```

```
/* execution object 配置参数列表 */
SDExecuteObjParams executeObjList[]={
     /* name */     /*start func*/   /*loc start func*/ /*stop func*/
    {"mat mpy calc ",  NULL,             NULL,              NULL,
    {"mat mpy exit ",  NULL,             NULL,              NULL,

   /*loc stop func*/ /*receiver*/  /*context*/ /*EO handler*/
        NULL,          mat_mpy_calc,  NULL,       NULL },
        NULL,          mat_mpy_exit,  NULL,       NULL }
};
```

需要注意的是 queue 到 execution object 的映射是通过 receiver 函数关联起来，如红色高亮显示部分。

初始化job的伪代码如下：

```
int init_jobs()
{
    /* Create queue group */
    numQueGrp = sizeof(QueGrpObjList)/sizeof(SDQueGrpObjParams);
    for( i =0; i< numQueGrp; i++ )
    {
        em_queue_group_create(QueGrpObjList[i].name,
                    &QueGrpObjList[i].mask,0,0);
    }

            /* create execution object */
            numEO = sizeof(executeObjList)/sizeof(SDExecuteObjParams);
    for( i =0; i< numEO; i++ )
    {
        executeObjList[i].eo_handler = em_eo_create(
              executeObjList[i].name,         // name
              executeObjList[i].start,        // start func
              executeObjList[i].local_start, // start loca func
              executeObjList[i].stop,         // stop func
              executeObjList[i].local_stop,  // stop local func
              executeObjList[i].receive,      // receiver func
              executeObjList[i].eo_ctx);      // context
    }

    /* creat queue object */
    numQue = sizeof(QueueObjList)/sizeof(SDQueueObjParams);
    for( i =0; i< numQue; i++ )
    {
        QueueObjList[i].que_handler = em_queue_create(
              QueueObjList[i].name,
              QueueObjList[i].type,
              QueueObjList[i].prio,
              QueueObjList[i].group);

        /* map the queue with the EO */
        {
            /* search for the corresponding execution object */
            for( j = 0; j < numEO; j++ )
            {
                if( executeObjList[j].receive == QueueObjList[i].receive )
                    break;
            }

            em_eo_add_queue(
              executeObjList[j].eo_handler,
              QueueObjList  [i].que_handler);
        }
    }
}
```

2.2.3 产生 event

本文的演示用例把 matrix Y 切分成了 128 个 2048*16 的子块，每个 event 对应一个子块。Event 被发送给 execution object 以后，receive 函数计算 Matrix X 乘与 matrix Y block，即 100*2048 × 2048*16 的矩阵乘，产生 100*16 个输出。event 的产生包括下面几个简单步骤：

- 调用 em_alloc 函数，从 public pool 获取 free 的 event 描述符并且 enable preloading。

- 把待处理的数据缓冲区挂到描述符上，也就是把描述符的 buffer 指针指向这个数据缓冲区。
- 在描述符的 software info 域填上 job index。
- 调用 em_send，把 event 发送到对应的 queue，也就是 proc queue。

下面是产生 event 的代码：

```
void source_job()
{
    for( i = 0; i< numJob; i++ )
    {
        lvEventHdl = em_alloc(eventBufSize,
TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_C,
                              pubPoolIdx);

        descPtr = (HostPacketDescriptor *)(lvEventHdl&0xfffffff0);

        /* hook the buffer to the descriptor */
        descPtr-> buffer_ptr = GLOBAL_ADDR(&ptr_transX2[i*(eventBufSize>>2)]);
        descPtr-> userCfgInfo = i; /*fill the job index into desc soft info field */
        em_send(lvEventHdl,QueueObjList[procQueueIdx].que_handler);
    }
}
```

需要注意的是 Event 产生的时候，它被哪一个 execution object 处理还没有确定。因为 execution object 只是和 queue 关联的。当把 event 发送到一个 queue 的时候，负责处理 event 的 execution object 就确定了。所以在调用 em_send()发送 event 到 queue 的时候参数之一就是要发送到的 queue 的 handler。

2.2.4 运行和 exit

如前所述，“矩阵乘 event”是通过 proc queue 发给 scheduler 的，所以它被 proc queue 映射到 mat_mpy calc 这个 execution object 上。Dispatcher 收到这个 event 后就调用“mat_mpy calc”对应的 receiver 函数计算矩阵相乘。因为 proc queue 所属的 queue group 是映射到所有 DSP 核的，所以 128 个“矩阵乘 event”是在所有核上并行处理的。每个核处理完 event 后就把它释放回 global free pool。这样这个 event 又成为一个 free 的 event。

如 2.2.3 节所述，主核可以通过查询 global free pool 的描述符个数是否恢复来判断是否所有“矩阵乘 event”已经处理完。

当所有“矩阵乘 event”处理完后，主核再产生 8 个“exit event”发送到 exit queue。理论上 scheduler 可以把 exit job 调度给任意一个核，而不会保证每个核一个 exit job。所以 exit job 中的处理比较特殊。exit job 的 receiver 函数直接执行系统调用 exit(0)。这样就不会返回到 Dispatcher，也不会再发出 prefetch command。而另一方面，scheduler 是在收到 DSP 核的 prefetch command 以后才把 event 调度给这个核的。这个机制保证了每个核收到且仅收到一个“exit event”。

在 exit job 的 receiver 函数中，主核执行的分支稍有差异。主核需要先做完结果的校验再执行系统调用 exit(0)。所以在板上运行是会观察到其他核很快(小于 1s)就从 run 状态转换到 abort 状态，而主核保持 run 了很长时间(大约 50s)才进入 abort 状态。原因是：在主核上执行结果验证工作时产生校验结果的函数计算耗时比较长。

下面是 exit job 的 receiver 函数的代码主干：

```
void mat_mpy_exit(
      void* eoCtxtPtr,
      em_event_t eventHdl,
      em_event_type_t eventType,
      em_queue_t queueHdl,
      void* queueCtxtPtr)
{
    if( DNUM == MY_EM_INIT_CORE_IDX )
    {
        verification();
        exit(0);
    }
    else
          exit(0);
}
```

2.3 基于 OpenEM 的大矩阵乘性能测试结果

2.3.1 算法代码和 cycle 数的理论极限

设 r1 是 X 矩阵的行数，c1 是 X 矩阵的列数，c2 是 Y 矩阵的列数。在我们的演示用例中 r1 =100, c1 = 2048, c2 = 2048。如前所述，Receiver 函数要计算 100*2048 × 2048*16 的矩阵乘，对应下面的伪代码：

```
for (i = 0; i < r1; i+=2)
{
    for( kk = 0; kk < 16; kk += 2 )
    {
         /* loop kernel 计算 2*2048 × 2048*2 的矩阵乘 */
         for (k = 0; k < c1; k+=4)
         {
              … /* loop kernel is 4 cycles on C66 */
         }
    }
}
```

循环内核是 4 个 cycle。 如果只考虑循环内核消耗的 cycle 数，计算 100*2048 × 2048*16 的矩阵乘需要的 cycle 数是 100/2*16/2*2048/4*4 = 819,200 cycle。整个 X*Y=Z 包括计算 128 个这样的矩阵乘。所以总的 cycle 数是 819,200*128 = 104,857,600 cycles。在 1Ghz 的 C66 核上这相当于 104.8ms。但是我们的上述理论计算没有考虑循环的前后缀消耗的 cycle 数，也没有考虑 cache miss stall 的等待时间。在 6678EVM 板的单个 DSP 核上实测，计算 X*Y=Z 消耗的实际时间是 190,574,214 cycles。相当于 190ms。

2.3.2 基于 OpenEM 的性能测试结果

基于 OpenEM 的演示用例实现过程中，DSP 代码中嵌入了少量测试代码收集运行的 cycle 信息。每个核把自己处理每个 event 的起始和结束时间记录在内存(我们通过一个全局 timer 来保证所有 DSP 核记录的时间戳在时间轴上是同步的)。这些时间戳用 CCS 存到主机做后处理分析。通过分析，我们可以得到 8 个 DSP 核并行处理消耗的时间。还可以分析每个 DSP 核的忙/闲区间。

测试结果是，从第一个 event 开始处理到最后一个 event 处理完，总时间是 31,433,438 cycle，也就是 31.4ms。也就是说，通过 OpenEM 把单 DSP 核的工作负载平衡到 8 个 DSP 核上能达到的 DSP 核利用率是 190,574,214/(**31,433,438*8)= 76%。**

通过对时间戳的处理我们得到下面的运行图，“-”表示 receiver 函数处理 event 的区间，本文称之为有效时间。“**#**”表示 receiver 之外的区间（也就是代码在 dispatcher 中执行的区间），本文称之为调度开销。每个“-”和“**#**”刻度表示 100,000 CPU cycle。

```
      <---------------------->
core0: -----------------#####----------------####-----------------###
core1: -----------------####-----------------####-----------------##--
core2: -----------------#####----------------####-----------------###
core3: -----------------#####----------------####-----------------###
core4: -----------------#####----------------####-----------------###
core5: -----------------#####----------------####-----------------###
core6: -----------------#####----------------####-----------------###
core7: -----------------#####----------------####-----------------###
```

从上面的执行图看，调度开销不小，占了大约 15～20%的时间。但是这只是表面的现象。实际上，调度开销的大部分时间里，Dispatcher 是在查询 hardware queue，等待新的 event。这是因为 preload 没能及时完成导致的。因为同时给 8 个核做 preload 需要很大的数据搬移的流量。根据以往的测试结果。使用 QMSS 的 packet DMA 从 DDR3 输入数据到 local L2 的流量大约是 4G bytes 每秒。那么 preload 8 个 event 总的数据量是 4byte * 2048 rows * 16 columns * 8 core = 1M bytes，需要的时间是 1/4 ms。因为每个“-”和“**#**”刻度表示 100,000 CPU cycle，运行图中红线长度就代表 preload 8 个 event 的时间，它非常接近 250,000 cycle。理论计算和实际值基本吻合，所以我们认为调度延迟是 packet DMA 的传输流量不足导致的。

我们也测试了不使用 `pre-load` 的场景。观测到 scheduler 调度一个 event 的延迟大约是 1200 个 C66 CPU cycle。但是 `DSP` 核处理一个 `event` 的耗时增大到原来的 `10` 倍。所以，pre-load 虽然会导致 QMSS packet DMA 流量不足成为凸显的瓶颈，但是从总体效率来看还是非常必要的。

细心的读者可能会发现 76% + 20% = 96%，并不是 100%。我们分析时间戳发现，8 个 DSP 核同时运行的场景下，每个核处理一个 100*2048 × 2048*16 的矩阵乘的时间比只有一个 DSP 核运行的场景下的时间稍长。原因是： 我们的演示用例中 X 矩阵和 Z 矩阵是存储在 shared L2 的， 8 个核同时运行就会同时读写这两个 buffer，导致产生 shared L2 的 bank 冲突。 所以性能下降了。

3 总结

OpenEM 具有使用简单，功能实用，执行高效的特点。能在 KeyStone 多核 DSP 上实现动态的负载平衡。它一方面提供了强大的功能，另一方面也给应用留出了很大的灵活性。例如，通过让应用初始化 free pool 方便了 buffer 的管理。OpenEM 的现有功能已经能够支持基本的应用。随着版本更新功能还将不断完善。

Reference

Ref[1] ti.openem.white.paper.pdf 位于 OpenEM 安装目录

重要声明

德州仪器(TI) 及其下属子公司有权根据 JESD46 最新标准, 对所提供的产品和服务进行更正、修改、增强、改进或其它更改， 并有权根据 JESD48 最新标准中止提供任何产品和服务。客户在下订单前应获取最新的相关信息, 并验证这些信息是否完整且是最新的。所有产品的销售都遵循在订单确认时所提供的TI 销售条款与条件。

TI 保证其所销售的组件的性能符合产品销售时 TI 半导体产品销售条件与条款的适用规范。仅在 TI 保证的范围内，且 TI 认为 有必要时才会使用测试或其它质量控制技术。除非适用法律做出了硬性规定，否则没有必要对每种组件的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。客户应对其使用 TI 组件的产品和应用自行负责。为尽量减小与客户产品和应 用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 组件或服务的组合设备、机器或流程相关的 TI 知识产权中授予 的直接或隐含权限作出任何保证或解释。TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服 务的许可、授权、或认可。使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它 知识产权方面的许可。

对于 TI 的产品手册或数据表中 TI 信息的重要部分，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况 下才允许进行复制。TI 对此类篡改过的文件不承担任何责任或义务。复制第三方的信息可能需要服从额外的限制条件。

在转售 TI 组件或服务时，如果对该组件或服务参数的陈述与 TI 标明的参数相比存在差异或虚假成分，则会失去相关 TI 组件 或服务的所有明示或暗示授权，且这是不正当的、欺诈性商业行为。TI 对任何此类虚假陈述均不承担任何责任或义务。

客户认可并同意，尽管任何应用相关信息或支持仍可能由 TI 提供，但他们将独力负责满足与其产品及在其应用中使用 TI 产品 相关的所有法律、法规和安全相关要求。客户声明并同意，他们具备制定与实施安全措施所需的全部专业技术和知识，可预见 故障的危险后果、监测故障及其后果、降低有可能造成人身伤害的故障的发生机率并采取适当的补救措施。客户将全额赔偿因 在此类安全关键应用中使用任何 TI 组件而对 TI 及其代理造成的任何损失。

在某些场合中，为了推进安全相关应用有可能对 TI 组件进行特别的促销。TI 的目标是利用此类组件帮助客户设计和创立其特 有的可满足适用的功能安全性标准和要求的终端产品解决方案。尽管如此，此类组件仍然服从这些条款。

TI 组件未获得用于 FDA Class III（或类似的生命攸关医疗设备）的授权许可，除非各方授权官员已经达成了专门管控此类使 用的特别协议。

只有那些 TI 特别注明属于军用等级或“增强型塑料”的 TI 组件才是设计或专门用于军事/航空应用或环境的。购买者认可并同 意，对并非指定面向军事或航空航天用途的 TI 组件进行军事或航空航天方面的应用，其风险由客户单独承担，并且由客户独 力负责满足与此类使用相关的所有法律和法规要求。

TI 已明确指定符合 ISO/TS16949 要求的产品，这些产品主要用于汽车。在任何情况下，因使用非指定产品而无法达到 ISO/TS16949 要求，TI不承担任何责任。

	产品		应用
数字音频	www.ti.com.cn/audio	通信与电信	www.ti.com.cn/telecom
放大器和线性器件	www.ti.com.cn/amplifiers	计算机及周边	www.ti.com.cn/computer
数据转换器	www.ti.com.cn/dataconverters	消费电子	www.ti.com/consumer-apps
DLP® 产品	www.dlp.com	能源	www.ti.com/energy
DSP - 数字信号处理器	www.ti.com.cn/dsp	工业应用	www.ti.com.cn/industrial
时钟和计时器	www.ti.com.cn/clockandtimers	医疗电子	www.ti.com.cn/medical
接口	www.ti.com.cn/interface	安防应用	www.ti.com.cn/security
逻辑	www.ti.com.cn/logic	汽车电子	www.ti.com.cn/automotive
电源管理	www.ti.com.cn/power	视频和影像	www.ti.com.cn/video
微控制器 (MCU)	www.ti.com.cn/microcontrollers		
RFID 系统	www.ti.com.cn/rfidsys		
OMAP应用处理器	www.ti.com/omap		
无线连通性	www.ti.com.cn/wirelessconnectivity	德州仪器在线技术支持社区	www.deyisupport.com

邮寄地址： 上海市浦东新区世纪大道 1568 号，中建大厦 32 楼 邮政编码： 200122