基于多芯粒集成的GPU共享存储式仿真器说明文档

1. 简介

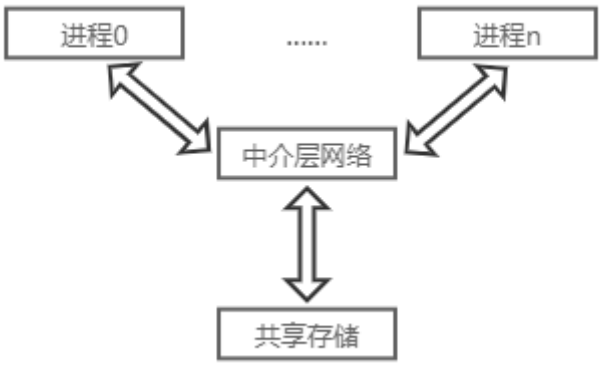
1.1 背景与目的

单一芯片规模增大会面临着成本上升与良率下降的问题。为解决此问题，Intel，NVIDIA与AMD等公司将多芯粒集成技术作为主流的处理器设计模式。但当前多芯粒集成技术还面临如下挑战：多芯粒集成技术中的设计空间复杂，设计选项多，前期设计空间探索中，需要仿真器进行仿真。当前的开源芯片仿真器有CMP[1]， sniper[2]，gem5[3]，GPGPU-Sim[4]，gem5+GPGPU-Sim仿真器[5]等。前三者为CPU仿真器，GPGPU-Sim为GPU仿真器，gem5+GPGPU-Sim为CPU与GPU异构型片上系统(system-on-chip, SoC)仿真器。但是这些仿真器存在几个问题：缺乏详细和准确的中介层互连延迟和功耗模型，因此不足以对片间的互连进行准确的建模；无法进行大规模并行仿真以加快仿真速度，采用多芯粒集成技术所构建的系统中可能具有大量内核，其需要大规模并行模拟器来加快仿真速度；无法支持各种存储模型。

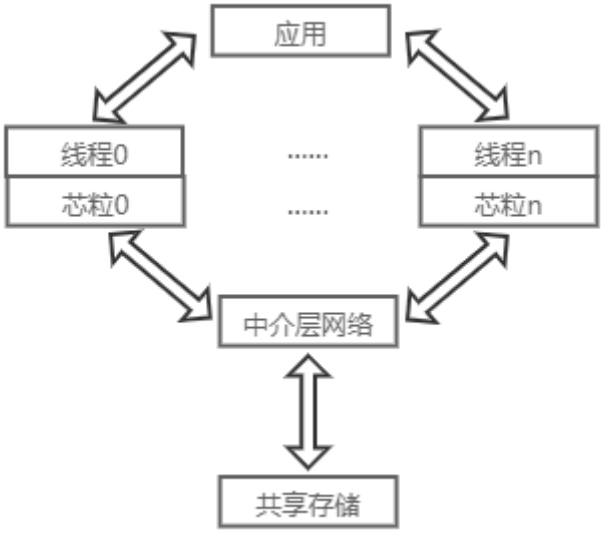
对此，本文提出了一种基于多芯粒集成技术的多GPU仿真器。其具有如下几个关键部分：第一，片间使用共享存储以及相应的编程模型；第二、针对进程间的通信，设计了相应的通信协议；第三，通过将芯粒同步转化为进程同步，并使用进程屏障实现进程同步，使得多个模拟器进程可以同步它们的通信。

1.2 基于多芯粒的多GPU仿真器构建

1.2.1 总体架构



a) 基本架构



b) 应用运行于仿真器之上

图1 总体架构

图1所示为系统的目标体系结构。如图1 a) 所示，每个进程通过中介层网络(network-on-interposer, NoI)与共享存储区域相连。图1 b) 展示了系统运行时的架构图。在该系统中，用进程模拟芯粒，当系统执行某个应用时，一个大的应用分解成多个线程，分配于各个芯粒。该系统为需要使用高计算能力、芯粒间可高效通信、易于计算延迟与能耗的仿真器的多芯粒研究者构建一个基于多芯粒的多GPU系统仿真器，该仿真器可在不同GPU芯粒运行计算任务，相互之间通过通信协议进行交互，并由NoI模拟器得出通信延迟等信息，同时将所需传递的数据存储至共享存储之中。

1.2.2 共享存储模型

基于多芯粒集成技术构建的系统的存储模型大致有两大类，一是分布式，二是共享式。本文采用的是共享式。

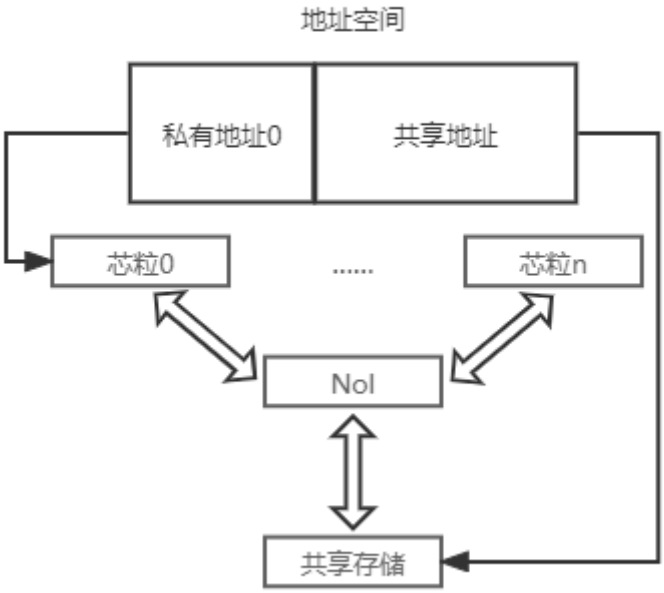


图2 芯粒地址空间

如图2所示，图2展示的是芯粒0角度的地址空间，私有地址0映射到芯粒0，共享地址映射到共享存储，芯粒0能够访问自身的私有地址0以及共享地址，无法访问其他芯粒的私有地址。

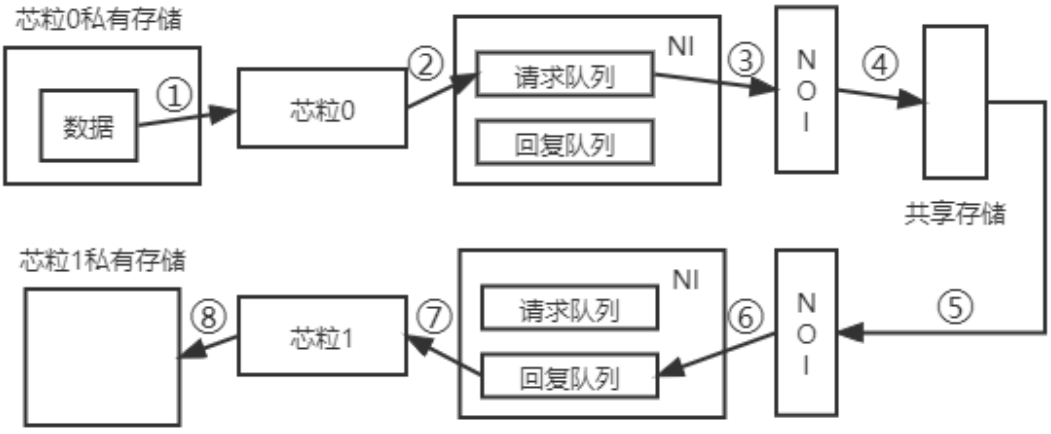


图3 共享存储传递数据

图3表示了通过共享存储传递数据的过程，其分四步：①芯粒0将数据从芯粒0私有存储取出；②芯粒0发送写请求，数据发送至请求队列；③数据通过请求队列发送至NoI；④芯粒0将数据写入共享存储；⑤芯粒1从共享存储取出数据；⑥芯粒1发送读请求，NoI通过回复队列发送数据；⑦芯粒1通过回复队列获得数据；⑧中芯粒1将数据存入芯粒1私有存储。共享存储和芯粒通信实现如下，首先判断芯粒进行的是读操作还是写操作，根据操作的不同确定不同的trace字段，如果是写操作，才将trace写入共享存储memory中，代码位于文件worksapce/controller.c：

```
if(readOrWrite==1){//判断是读/写以确定trace字段
        inputChipID=atoi(argv[1]);
        //因为进行的是timing model的仿真，所以，暂时在进行写操作时，目的芯粒用的是随机生成的
        int i=Random(numOfChip);
        if(outputChipID==inputChipID){
            i=(i+1)%numOfChip;
        }
        outputChipID=chipID[i];
    }
    else{
        inputChipID=atoi(argv[4]);
        outputChipID=atoi(argv[1]);
    }

    char bench[20]="bench";
    trace_writeFile(inputChipID,outputChipID,time,bench);

    system("sort -n -k 1 bench -o bench");
    char cmd[256];
    if(readOrWrite==1){
        system("tail -1 bench >> pre_memory.txt");//如果是写操作，才将trace写入memory
    }
    sprintf(cmd,"tail -%d pre_memory.txt > memory.txt", MEM_SIZE);
    system(cmd);
```

1.2.3 仿真器进程间同步

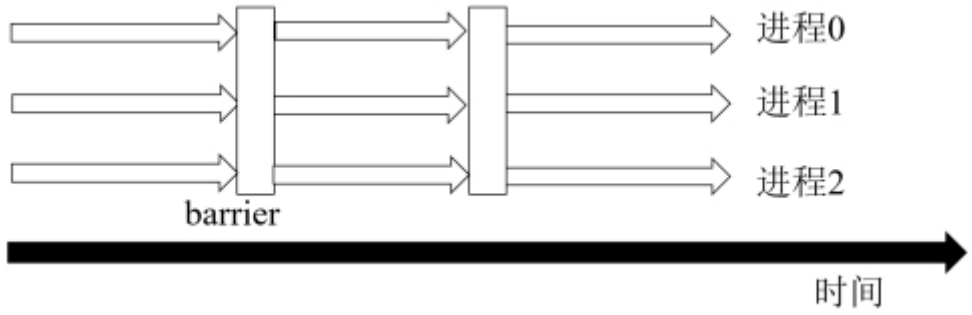


图4 进程同步

芯粒间同步问题转化成进程间同步问题，本文的解决方法采用了多核CPU仿真器CMP中采用的屏障(barrier)的思想。如图4，在某些时间节点设置针对进程的barrier，当进程到达barrier所在的时间节点时，将计数加一，当计数器的数等于所有进程数时，则通知所有的进程继续往下执行，否则令其阻塞在barrier前。其执行操作流程图如图5所示，首先置计数器为0，若计数器读数不等于总进程数，令到达的进程阻塞，当有进程到达时，增加计数器读数，至计数器读数与总进程数相等时，屏障移除，所有进程通行。

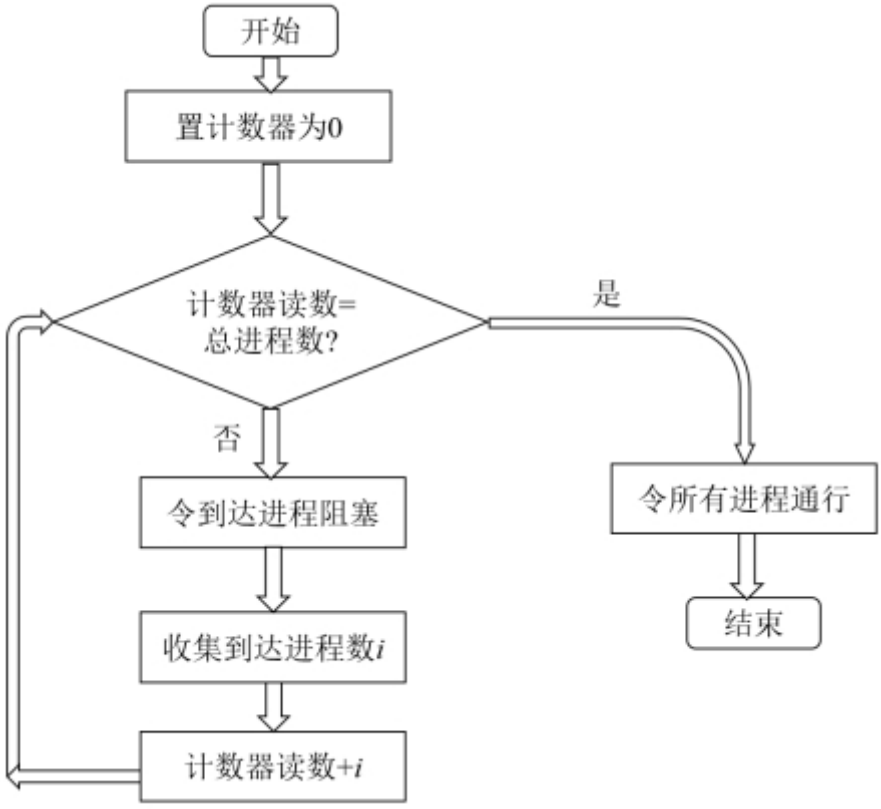


图5 barrier操作流程图

仿真器进程间同步的具体实现如下，芯粒进行读写时，调用synchronization函数，将当前进程时间、到达的芯粒号写入synchronization.txt，接着读取当前barrier时间，若进程的时间小于barrier时间，则令其继续运行。代码位于文件worksapce/controller.c

```
void synchronization(float time,char * chipID,int numOfChip){

    char filename_string[64];
    char time_string[10];
    strcpy (filename_string,"../workspace/synchronization/");
    int time_int=(int)time;

```

```
    //将当前进程时间、到达的芯粒号写入synchronization.txt
    strcat (filename_string,"synchronization.txt");

    /*read file get the time*/
    int syn_time=syn_fileReadFirstLine(filename_string);//读取当前barrier时间
    if (time_int<syn_time)return ;//若小于barrier时间，则令其继续运行
    int writeFail_flag=0;
    /*check if this chip is ready*/
    int row=fileRowCount(filename_string);
    int exist_flag=syn_fileSearch(filename_string,chipID);

    if(exist_flag==1){}
    else{/*write the file*/
        FILE *fp_out;
        if ((fp_out = fopen((filename_string), "aw+")) == NULL)
                writeFail_flag=1;
        if (0 == flock(fileno(fp_out), LOCK_EX)){

            fputs(chipID, fp_out);//将当前芯粒号写入文件
            fputs("\n", fp_out);
            fclose(fp_out);
            flock(fileno(fp_out), LOCK_UN);
        }
        else{
                printf("lock failed\n");
            }
    }
```

1.2.4 芯粒与NoI仿真器的修改

基于多芯粒的多GPU仿真器是基于GPGPU-Sim和popnet[6]进行改造的，支持共享存储、消息传递等编程模型，并支持GPU之间的通信，包括数据读写；其中，GPGPU-Sim作为芯粒仿真器，popnet作为NoI仿真器。

其中GPGPU-Sim的架构图如图6示。其有多个单指令多线程(single instruction multiple threads, SIMT)核簇，每个SIMT核簇里有多个SIMT核，保证了高并行以提升计算能力。核簇通过与booksim模拟的NoC相连来进行访存。在存储部分，分别有原子操作执行器，以及L2缓存和片外动态随机存取存储器(Dynamic Random Access Memory, DRAM)。

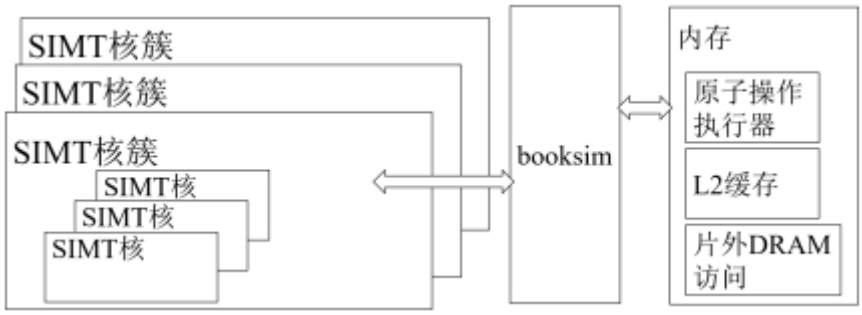


图6 GPGPU-Sim架构

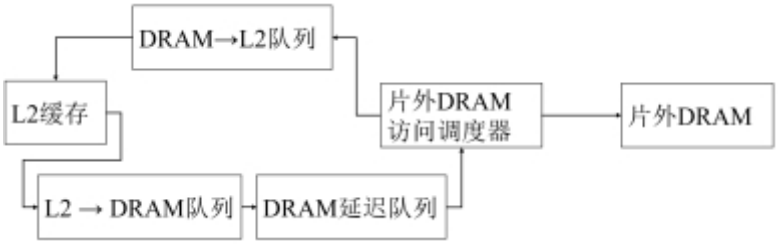


图7 GPGPU-Sim 片外DRAM与L2缓存连接细节

为了保证GPGPU-Sim功能的正常执行，在片外DRAM访问这里将每个GPGPU-Sim接入NoI以作为芯粒仿真器。图7所展示的是GPGPU-Sim 片外DRAM与L2缓存连接的细节。片外DRAM访问由一个调度器进行调度。调度器与L2缓存之间通过三个队列来连接：DRAM→L2队列、L2→DRAM队列、DRAM延迟队列，其中DRAM延迟队列用以模拟DRAM访问所带来的延迟。

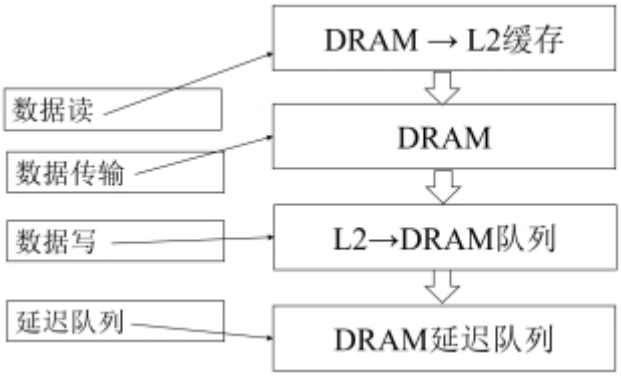


图8 映射关系

为了保证GPGPU-Sim的正常运行，本文选择将片外DRAM替换为NoI，而不是从NoC接入NoI。这样就得到了一个映射关系，如图8。其中，数据读对应了DRAM→L2缓存部分，也即原来从DRAM中读取数据的操作转化成了通过NoI从共享或私有存储中读取数据；原有的DRAM时钟周期转化成了数据传输周期；原有的L2→DRAM缓存部分转化成了数据写操作；原有的延迟队列通过修改，将访问DRAM的延迟替换成访问共享或私有存储的延迟。

除了对用作芯粒仿真器的GPGPU-Sim进行修改之外，还需对用作NoI仿真器的popnet进行修改。用作NoI仿真器的popnet原本是运行在Linux系统上的NoC仿真器，使用C++语言进行构建。因NoI与NoC在许多方面有着相同之处，故将其改造作为NoI仿真器，将各个芯粒连接起来。其可以通过配置文件或命令参数更改虚通道数、拓扑结构、路由器数量等，便于用户使用。

仿真器popnet接受trace作为输入，这就需要在GPGPU-Sim进行通信时，将这些通信的元组记录保存下来，经过格式转换后作为popnet的输入。Popnet中的trace格式为：源节点编号，目的节点编号，时间。在接受了输入的trace后，popnet运行得到延迟，再反馈给GPGPU-Sim。

仿真器运行的架构如图9，基准测试程序作为应用运行在多个GPGPU-Sim上，GPGPU-Sim产生trace，输入到popnet，popnet反馈延迟给GPGPU-Sim。

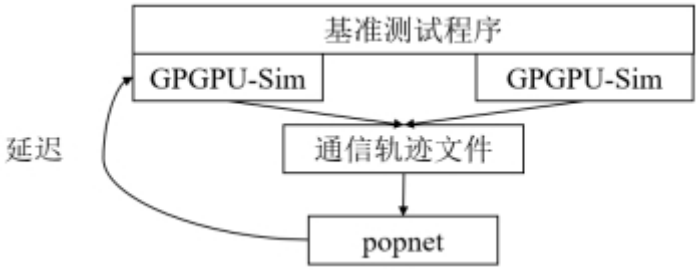


图9 仿真器系统运行架构

2. 文件标准

2.1 GPGPU-Sim的输入输出

2.1.1 输入文件

输入文件包括配置文件和由CUDA编译出的可执行文件。配置文件主要用于规定gpgpu-sim所仿真的GPU的各项指标，包括其架构、计算单元数量等内容。gpgpu-sim所需的配置文件共3个，分别是：

a) gpgpusim.config 主要规定了仿真器运行所需要的各项参数，以及单个芯粒的基本配置。

b) gpuwattch_*.xml主要规定了计算功耗时所需要的各项参数内容。

c) config*islip.icnt主要规定了芯片内部的互联网络架构。

这三项配置文件gpgpu-sim已经给出，用户可以通过修改这三项文件内容达到对芯粒配置的修改。

可执行文件是本次仿真需要执行的程序，需要由用户编写源码，并使用CUDA进行编译。

2.1.2 输出文件

```
---------------------------END-of-Interconnect-DETAILS-------------------------


gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 3 sec (3 sec)
gpgpu_simulation_rate = 405000 (inst/sec)
gpgpu_simulation_rate = 4876 (cycle/sec)
total time is 3124 ms
```

图10 GPGPU-Sim运行结果

如图10所示，第一行代表本次仿真所花费的现实时间。第二行代表本次仿真中，仿真器每秒处理的指令数量。第三行代表本次仿真中，仿真器每秒的时钟周期数。一般来说，仿真结果中最被关注的是总时钟周期数，因此将第一行数据与第三行数据相乘，即可得到单个芯粒在一次仿真中所耗费的总时钟周期数。

2.2.popnet的输入输出

2.2.1 输入文件

Popnet的输入文件包括**trace文件**与**启动脚本文件**。

trace文件记录了不同芯粒之间通信记录，它由多行组成，每一行的结构如下：

T sx sy dx dy n

T: 数据发送的时间

sx sy: 发送数据的芯粒在网络中的坐标

dx dy: 接收数据的芯粒在网络中的坐标

n: 本次发送数据的数据包大小

启动脚本文件记录了启动Popnet的各项配置，它的内容如下：

./popnet -A 9 -c 2 -V 3 -B 12 -O 12 -F 4 -L 1000 -T 20000 -r 1 -I ./random_trace/bench -R 0

-A 9: 互联网络的大小，代表每个维度上的芯粒数

-c 2: 互联网络的维数，2代表网络是2维的，3代表网络是3维的

-B 12: 输入缓冲区的大小

-O 12: 输出缓冲区的大小

-F 4: flit大小

-L 1000: 线路长度，以um为单位

-T 20000: 仿真周期

-r 1: 随机数种子

-I ./random-trace/bench: trace文件位置

-R 0: 选择拓扑结构，0，1，2分别代表不同的拓扑结构

2.2.2 输出文件

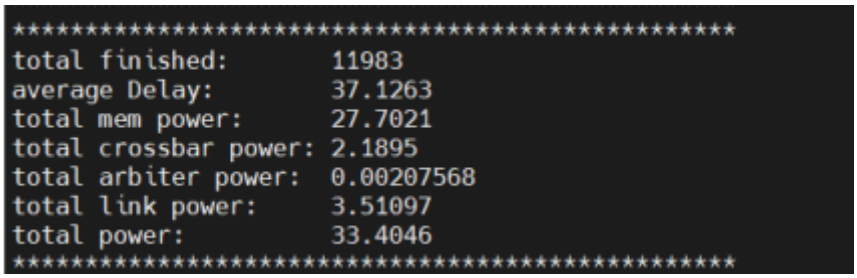


图11 popnet运行结果

如图11所示，第一行total finished表示总的数据发送量（以数据包为单位），第二行average Delay表示每个包平均经过多少个时钟周期数的延时到达目标芯粒。第三行至第六行表示在不同环节的功耗，第七行表示总功耗。一般来说，我们关注的是average Delay。

2.3 计算公式

我们需要的是gpgpusim的gpgpu_simulation_rate（gpgpusim每秒的时钟周期数，见图10）和gpgpu_simulation_time（gpgpusim仿真花费时间，见图10）以及popnet的average delay（popnet每个包的平均延迟，见图11）。计算公式如下：

$$total\ delay = gpgpu\_simulation\_rate * gpgpu\_simulation\_time + average\ delay$$

3.安装与使用

3.1 环境要求

Linux版本	Ubuntu 14.04
gcc/g++版本	4.4.7
python版本	2.7.6
boost库版本	1.54.0
nvcc版本	4.0
openmpi版本	1.8.8

3.2 下载安装NVIDIA CUDA 4.0

3.2.1 下载ubuntu linux 10.10 cuda toolkit和GPU Computing SDK code samples

Https://developer.nvidia.com/cuda-toolkit-40\
GPGPU-Sim只支持到cuda 4

3.2.2 安装CUDA toolkit

```
chmod +x cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
sudo ./cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
```

默认安装在/usr/local/cuda，不用管他，直接enter。

3.2.3 增加CUDA toolkit到~/.bashrc中，添加环境变量

.bashrc在根目录下，是隐藏文件，按control+H可看到

```
echo 'export PATH=$PATH:/usr/local/cuda/bin' >> ~/.bashrc
echo 'export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib:/usr/local/cuda/lib64'
>> ~/.bashrc
source ~/.bashrc
```

可用vim查看：

```
sudo vim ~/.bashrc
```

底部两行已加入路径即可。

3.2.4 安装GPU Computing SDK code samples

```
chmod +x gpucomputingsdk_4.0.17_linux.run
sudo ./gpucomputingsdk_4.0.17_linux.run
```

默认安装在~/NVIDIA_GPU_Computing_SDK路径中，直接enter。

3.2.5 安装gcc-4.4和g+±4.4(CUDA 4.0只支持gcc版本到4.4)

```
sudo apt-get install gcc-4.4 g++-4.4
```

由于Ubuntu 18.04自带7.4.0版本gcc，所以无法安装，可通过以下方法修改：

```
sudo vim /etc/apt/sources.list
```

底部插入两行代码：

```
deb http://dk.archive.ubuntu.com/ubuntu/ trusty main universe
deb http://dk.archive.ubuntu.com/ubuntu/ trusty-updates main universe
```

添加好后，保存退出。

更新apt源：

```
sudo apt=get update
```

再重新安装gcc-4.4和g++-4.4即可。

```
sudo apt-get install gcc-4.4 g++-4.4
```

3.2.6 改变系统中的gcc/g++为gcc-4.4/g+±4.4

```
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 150
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.4 100
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-7 150
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.4 100
```

用update-alternatives选择4.4版本：

```
sudo update-alternatives --config gcc
```

输入4.4版本前面对应的序号，然后enter。

3.3 下载和运行GPGPU-Sim

3.3.1 从GitHub下载GPGPU-Sim

```
sudo apt-get install git
git clone https://github.com/gpgpu-sim/gpgpu-sim_distribution.git
```

3.3.2 安装依赖

```
sudo apt-get install build-essential xutils-dev bison zlib1g-dev flex
libglu1-mesa-dev
sudo apt-get install doxygen graphviz
sudo apt-get install python-pmw python-ply python-numpy libpng12-dev python-
matplotlib
sudo apt-get install libxi-dev libxmu-dev freeglut3-dev
```

3.3.3 添加CUDA_INSTALL_PATH到~/.bashrc中

```
echo 'export CUDA_INSTALL_PATH=/usr/local/cuda' >> ~/.bashrc
source ~/.bashrc
```

3.3.4 编译GPGPU_Sim

在编译之前，先将Chiplet_GPGPU-Sim_SharedMemory文件夹下的Souce Code文件夹重命名为SourceCode，如果路径有空格可能会影响后续操作。

```
source setup_environment
make
make docs
```

如果make结束出现错误，如图12

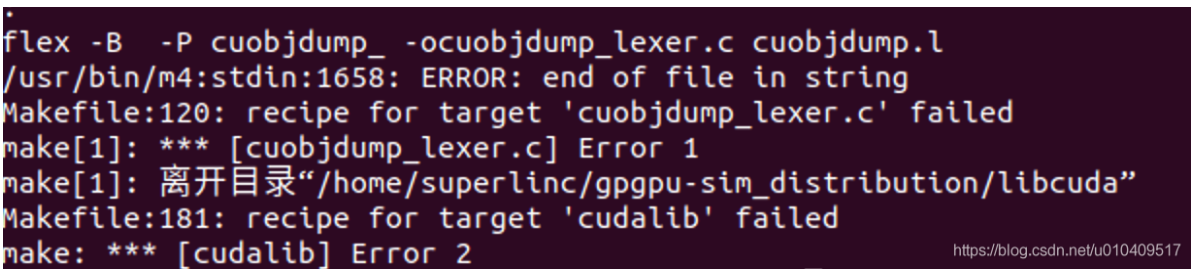


图12 编译gpgpu-sim出现错误

移除cuobjdump.l:109-111行，再make就不会出现错误了。

3.3.5 运行GPGPU_Sim

在gpgpu-sim_distribution-master下运行cuda程序，

程序示例：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void kernel(void) {

}

int main() {

    kernel << <1, 1 >> > ();
    printf("Hello world!\n");
    return 0;

}
```

保存为hello.cu格式。

终端运行：

```
nvcc  hello.cu -o hello.out
```

生成一个hello.out文件

```
./hello.out
```

但还不能运行GPGPU_Sim，要将/configs/GTX480文件夹下的三个文件都复制到程序中，如图13所示

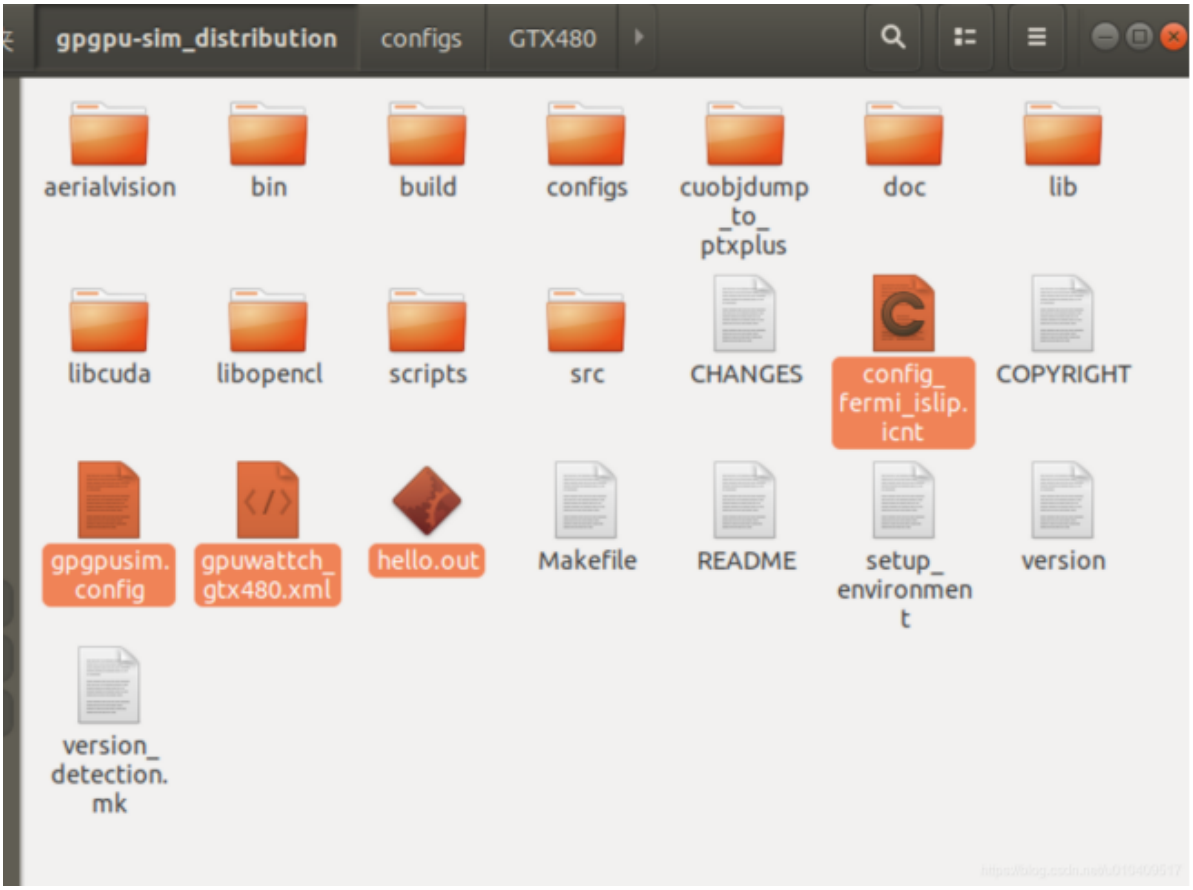


图13 运行GPGPU_Sim所需的文件

在此路径中运行：

```
source setup_environment
```

会发现出现一大堆信息，最后可以看到运行时间，速率等信息，以及最后的输出。至此，GPGPU_Sim安装运行完毕。

3.4 运行ispass-2009 benchmarks

进入Chiplet_GPGPU-Sim_SharedMemory/benchmark/，并给文件夹内的文件赋予可执行权限。

```
sudo chmod -R 777 ispass2009-benchmarks-master/
cd ispass2009=benchmarks-master/
```

3.4.1 编译ispass2009-benchmarks

打开Makefile.ispass-2009文件在文件开头添加：

```
CUDA_INSTALL_PATH=/usr/local/cuda
NVIDIA_COMPUTE_SDK_LOCATION=/[PATH]/NVIDIA_GPU_Computing_SDK
```

[PATH] 替换为安装 CUDA 过程中个人自定义的 CUDA SDK 路径。

比如我的路径为：

```
NVIDIA_COMPUTE_SDK_LOCATION=/home/fj5/NVIDIA_GPU_Computing_SDK
```

然后用编辑器打开Makefile.ispass-2009文件，修改OPENMPI_BINDIR路径：

```
change
export OPENMPI_BINDIR=/usr/lib64/mpi/gcc/openmpi/bin/;
to
export OPENMPI_BINDIR=/usr/bin/;
```

保存并退出。

3.4.2 AES 错误修正

进入AES目录下，用编辑器打开Makefile文件修改LINKFLAGS。

```
change
LINKFLAGS       := -L$(BOOST_LIB) -lboost_filesystem$(BOOST_VER)
to
LINKFLAGS       := -L$(BOOST_LIB) -lboost_filesystem$(BOOST_VER) -
lboost_system
```

3.4.3 DG 错误修正

进入DG目录下，用编辑器打开Makefile文件，找到第54和56行INCLUDES：

```
// line 54
change
INCLUDES = -Dp_N=$(N) -DNDG3d -DCUDA -I/opt/local/include -
I/usr/include/malloc -I$(HDRDIR) -I/opt/mpich/include
to
INCLUDES = -Dp_N=$(N) -DNDG3d -DCUDA -I/opt/local/include -
I/usr/include/malloc -I$(HDRDIR) -I/opt/mpich/include -
I/usr/lib/openmpi/include

// line 56
change
INCLUDES = -Dp_N=$(N) -DNDG3d -DCUDA -I/opt/local/include -
I/usr/include/malloc -I$(HDRDIR)
to
INCLUDES = -Dp_N=$(N) -DNDG3d -DCUDA -I/opt/local/include -
I/usr/include/malloc -I$(HDRDIR) -I/usr/lib/openmpi/include
```

3.4.4 WP 错误修正

进入WP根目录，用编辑器打开makefile文件，修改第75行。

```
// line 75
change
NVOPT = $(DEVICEEMU_NVCC) $(PROMOTE) $(DEBUGDEBUG) $(DEBUGOUTPUT) \
                -DXXX=$(XXX) -DYYY=$(YYY) -DMKX=$(MKX) --host-compilation
'C++' --use_fast_math
to
NVOPT = $(DEVICEEMU_NVCC) $(PROMOTE) $(DEBUGDEBUG) $(DEBUGOUTPUT) \
                -DXXX=$(XXX) -DYYY=$(YYY) -DMKX=$(MKX) --use_fast_math
```

然后找到GPGPULINK：

```
change
#GPGPULINK = -L$(CUDAHOME)/lib64/ -lcudart -
L$(NVIDIA_COMPUTE_SDK_LOCATION)/C/lib/ -lcutil_x86_64 -lm -lz -ldl -lGL -
lstdc++ $(NEWLIBDIR) $(LIB) # /usr/lib64/libstdc++.so.6
to
GPGPULINK = -L$(CUDAHOME)/lib64/ -lcudart -
L$(NVIDIA_COMPUTE_SDK_LOCATION)/C/lib/ -L$(NVIDIA_COMPUTE_SDK_LOCATION)/C/lib
-lm -lz -ldl -lGL -lstdc++ $(NEWLIBDIR) $(LIB) # /usr/lib64/libstdc++.so.6
```

3.4.5 执行编译命令：

```
make -f Makefile.ispass-2009
```

如果修改错误后还是编译不了的，就注释掉：

```
#$(SETENV) make noinline=$(noinline) -C AES
#$(SETENV) make noinline=$(noinline) -C DG/3rdParty/ParMetis-3.1
#$(SETENV) make noinline=$(noinline) -C DG
#$(SETENV) make noinline=$(noinline) -C WP
```

编译生成的二进制文件在.../bin/release/中。

3.4.6 链接GPU配置文件

```
cd /home/gpgpu-sim_distribution
source setup_environment
cd ispass2009-benchmarks/
./setup_config.sh GTX480
```

3.4.7 运行基准测试，比如NN

```
cd NN/
sh README.GPGPU-Sim
```

3.5 popnet的使用

3.5.1 对popnet进行编译

```
cd SourceCode/popnet-master
make
```

3.5.2 通信延迟的计算

在每次gpgpu-sim进行片外dram访问时，都在popnet发一个包，就当作往其他芯粒发包。即仿真器会于trace文件夹中的文件bench.src_x.src_y在添加一条trace记录。包的格式是：T sx sy dx dy n，

T: 表示数据包发出时的时间

sx sy: 表示数据源Chiplet的地址

dx dy: 表示数据目标Chiplet的地址

n: 表示包大小

在程序运行完成之后，将所有bench.*.*文件中的trace记录合并，并根据数据包发出时间进行排序，生成文件命名为bench并置于本目录下。然后进入popnet所在目录，执行指令得到通信延迟。

```
./popnet -A 9 -c 2 -V 3 -B 12 -O 12 -F 4 -L 1000 -T 20000 -r 1 -I ./address -R 0
```

各参数含义详见popnet文件夹下的README文件。

3.6 workspace文件夹的使用

先编译好GPGPU-Sim和popnet，然后再用compile.sh编译workspace文件夹下的几个文件，starter.c文件是作为启动各个benchmark的文件，controller.c/controller2.c是作为通信协议。然后就使用命令：

```
./S AES BFS MM mesh
```

前面几个是benchmark（AES,BFS,MM），最后一个是拓扑结构。

4. Benchmark的编写

4.1 芯粒-进程映射

在构建仿真器时，需要多个GPGPU-Sim作为芯粒，一种方法是将仿真器复制多份。然而一份GPGPU-Sim的大小为12兆字节(Megabytes, MB)，若有256个芯粒，则需要12×256=3072MB, 即3吉字节(Gigabyte, GB)。在实际应用中不可能使用这种大开销的方法。因此，本文采用了另外一种解决办法。

如图14，基准程序调用一个GPGPU-Sim的可执行文件时，会产生一个进程，每个进程都有其独有的进程标识符(process identification, PID)，程序运行时不会改变。于是本文所述工作用一个进程来代表一个芯粒，将其PID与芯粒号进行映射。因此需要一张PID-芯粒号映射表，每一行就是一个元组(PID，芯粒号)。

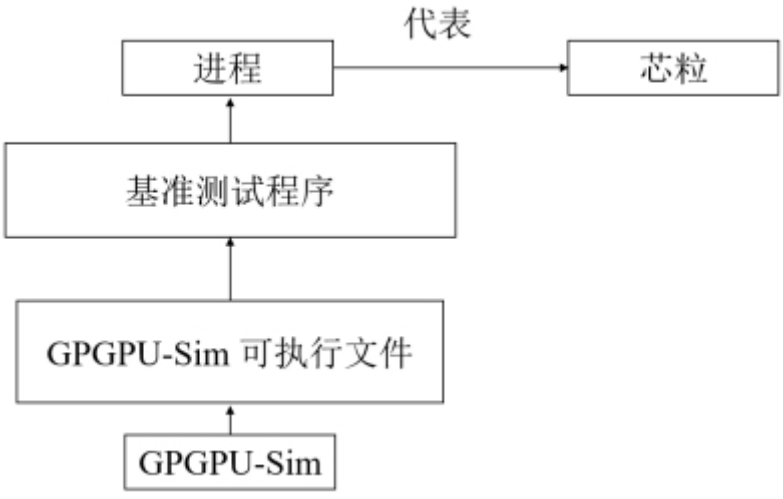


图14 GPGPU-Sim运行架构

4.2 多机应用矩阵乘（MM）示例

4.2.1 仿真器系统运行架构

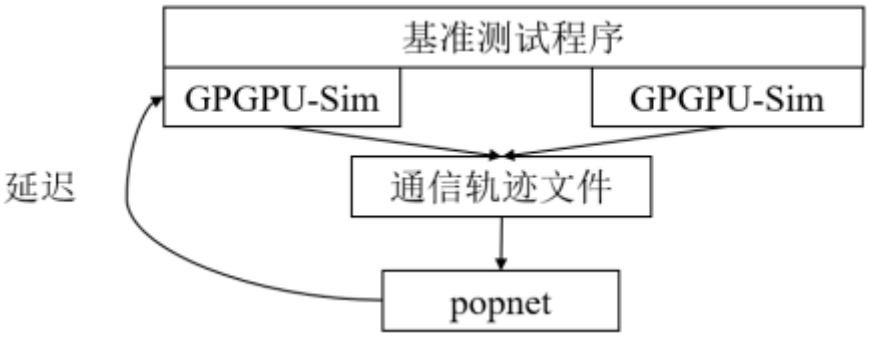


图15 系统架构

仿真器运行的架构如图15，基准测试程序作为应用运行在多个GPGPU-Sim上，GPGPU-Sim产生trace，输入到popnet，popnet反馈延迟给GPGPU-Sim。

4.2.2 矩阵乘编译与运行

$$AB = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

图16 矩阵乘示例图

```
cd Chiplet_GPGPU-Sim_SharedMemory/benchmark/ispass2009-benchmarks-master/MM
```

如图16所示，假设矩阵A和B大小400×400，则将mm.cu中的第9、10行设为400

```
vim mm.cu
//line 9 10
#define Row 400
#define Col 400
```

运行mm.cu得到A×B的运行时间，记为t_0，输出文件nohup.out在MM目录下。

```
cd Chiplet_GPGPU-Sim_SharedMemory/SourceCode/workspace/
sh compile.sh
./S MM mesh
```

然后由图16所示，容易得到A_{00}、B_{00}等小矩阵的大小为200×200，则则将mm.cu中的第9、10行设为200

```
cd Chiplet_GPGPU-Sim_SharedMemory/benchmark/ispass2009-benchmarks-master/MM
vim mm.cu
//line 9 10
#define Row 200
#define Col 200
```

假设开启四个gpgpusim（命令为./S MM MM MM MM mesh），则每个gpgpusim需要运行两次200×200的矩阵乘法，另外，矩阵加法也可以并行。矩阵加法函数如下：

```
__global__ void matrix_add_gpu(int *M,int *N,int *P,int width)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = threadIdx.y + blockDim.y * blockIdx.y;

    if (i < width && j < width)
    {
        P[i][j] = M[i][j] + N[i][j]
    }
}
```

运行方法同上，结束后将四个gpgpusim的总耗时相加，再加上popnet得到的通信延迟，得到矩阵并行计算花费的时间，记为t_1，将t_0除以t_1可得到加速比。矩阵乘法的划分和到各芯粒的映射见图17：

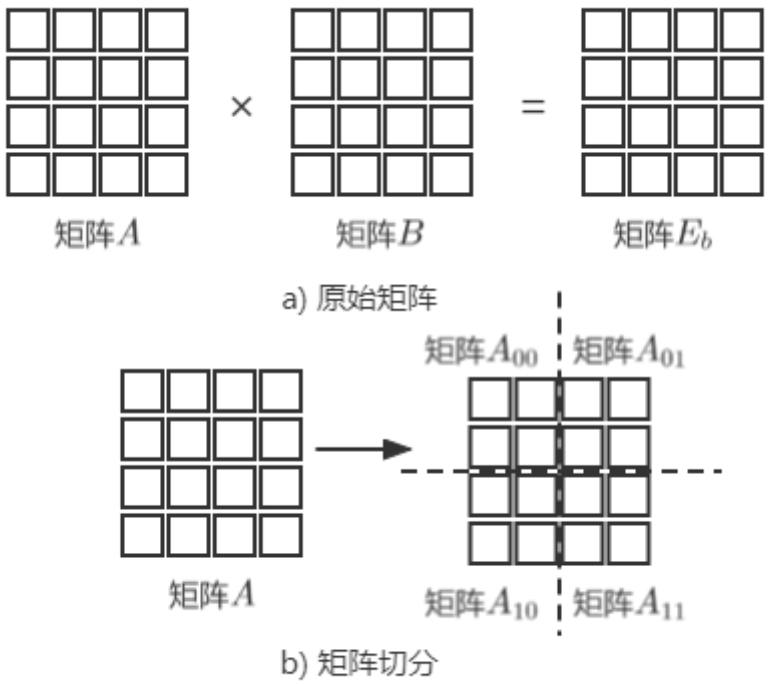
步骤一，原始大矩阵A、B相乘得到最初结果矩阵E_b，如图17 a) 所示；

步骤二，将大矩阵A、B进行划分后各得到四个小矩阵，矩阵A分解为矩阵A_{00}、A_{01}、A_{10}、A_{11}，矩阵B分解为矩阵B_{00}、B_{01}、B_{10}、B_{11}，如图17 b) 所示；

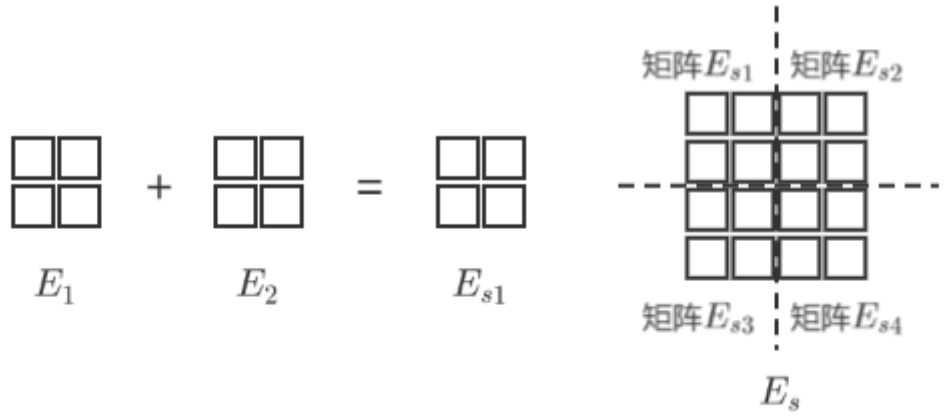
步骤三，切分后的小矩阵A_{00}和B_{00}进行相乘得到乘法结果E_1，另外，小矩阵A_{01}和B_{10}进行相乘得到乘法结果E_2，小矩阵A_{00}和B_{01}进行相乘得到乘法结果E_3，小矩阵A_{01}和B_{11}进行相乘得到乘法结果E_4，小矩阵A_{10}和B_{00}进行相乘得到乘法结果E_5，小矩阵A_{11}和B_{10}进行相乘得到乘法结果E_6，小矩阵A_{10}和B_{01}进行相乘得到乘法结果E_7，小矩阵A_{11}和B_{11}进行相乘得到乘法结果E_8，如图17 c) 所示；

步骤四，乘法结果E_1、E_2相加后得到加法结果E_{S1}，乘法结果E_3、E_4相加后得到加法结果E_{S2}，乘法结果E_5、E_6相加后得到加法结果E_{S3}，乘法结果E_7、E_8相加后得到加法结果E_{S4}，E_S由E_{S1}、E_{S2}、E_{S3}、E_{S4}构成，与E_b是相同的，如图17 d) 所示；

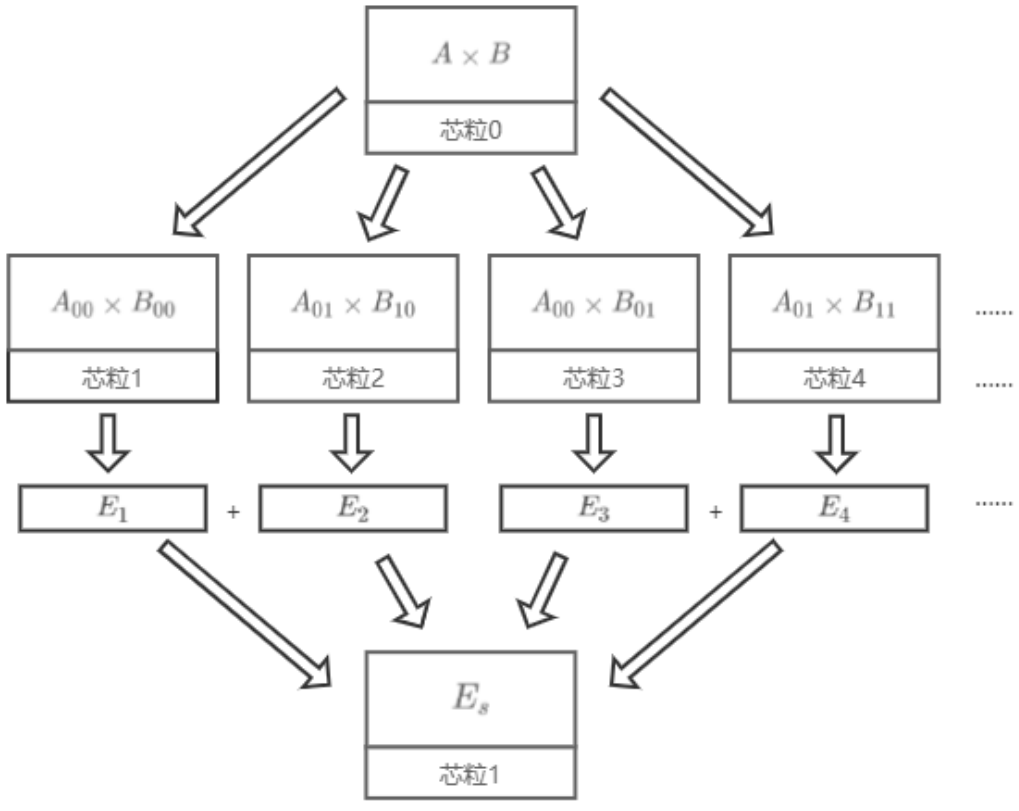
步骤五，图17 e) 展示了矩阵乘到各芯粒的映射，A×B映射到芯粒0，计算得到E_b，A_{00}×B_{00}映射到芯粒1，计算得到E_1，A_{01}×B_{10}映射到芯粒2，计算得到E_2，A_{00}×B_{01}映射到芯粒3，计算得到E_3，A_{01}×B_{11}映射到芯粒4，计算得到E_4，A_{10}×B_{00}映射到芯粒1，计算得到E_5，A_{11}×B_{10}映射到芯粒2，计算得到E_6，A_{10}×B_{01}映射到芯粒3，计算得到E_7，A_{11}×B_{11}映射到芯粒4，计算得到E_8，最后将E_1、E_2相加，E_3、E_4相加，E_5、E_6相加，E_7、E_8相加得到E_s，E_s映射到芯粒1。



c) 小矩阵相乘



d) 小矩阵相乘结果之和等于正确结果



e) 到各芯粒的映射

图17 矩阵乘法的划分和到各芯粒的映射

参考资料：

[1] Xiaohang Wang, Mei Yang, Yingtao Jiang, et al. On self-tuning networks-on-chip for dynamic network-flow dominance adaptation[J]. ACM transactions on embedded computing systems, 2014: 1-2.

[2] Trevor Carlson, Wim Heirman, Stijn Eyerman, et al. An evaluation of high-level mechanistic core models[J]. ACM transactions on architecture and code optimization, 2014: 28.

[3] Binkert Nathan, Bradford Beckmann, Gabriel Black, et al. The gem5 simulator[J]. ACM computer architecture news, 2011:1-7.

[4] Wilson Fung, Ivan Sham, George Yuan, et al. Dynamic warp formation and scheduling for efficient GPU control flow[C]. IEEE/ACM international symposium on microarchitecture, 2007: 407-420.http://www.gpgpu-sim.org/

[5] Hao Wang, Vijay Sathish, Ripudaman Singh, et al. Workload and power budget partitioning for single-chip heterogeneous processors[J]. Parallel architectures and compilation techniques - conference proceedings, 2012:401-410.

[6] Shang Li, Lishiuan Peh, Niraj Jha, et al. Dynamic voltage scaling with links for power optimization of interconnection networks[C]. High-performance computer architecture, 2003: 91-102.https://gitee.com/wlos/popnet_origial

博客：https://blog.csdn.net/u010409517/article/details/91050129

博客：https://learnku.com/articles/39866