

# 基于多芯粒集成的

## CPU-GPU 异构消息传递式仿真器说明文档

### 1. 简介

#### 1.1 多芯粒系统及其仿真

多芯粒系统是一种能降低芯片设计成本并提高复杂片上系统良率的新设计范式。多芯粒系统的设计空间比单芯片的片上系统大得多。为了支持早期的设计空间探索，仿真器显得尤为重要。目前较为常用的仿真器有 x86 架构的 sniper, Gem5 和费米架构的 gpgpu-sim 等。然而，由于以下原因，大规模的多芯粒系统无法使用现有的开源多核/众核仿真器进行模拟：1) 缺乏精确的芯粒间互连模型；2) 无法支持精确互连模型的大规模并行模拟。

因此，本文提出了一种由 gpgpu-sim, sniper 和 popnet 构建的基于多芯粒集成的 GPU 消息传递式仿真器，能够支持精确互连模型的大规模并行模拟。

#### 1.2 基于多芯粒集成的 CPU-GPU 异构的消息传递式仿真器

基于多芯粒集成的 CPU-GPU 异构的消息传递式仿真器（后文简称“本仿真器”）是一款用于模拟由 CPU 芯粒和 GPU 芯粒组成的多芯粒系统的仿真软件。它的主要功能是：模拟在多芯粒系统中一个应用程序运行的过程，给出运行结果以及各项性能指标，如：用时、能耗等。

#### 1.3 仿真器系统架构

##### 1.3.1 本仿真器的功能

对多芯粒系统进行仿真，仿真器需要实现两方面的功能：功能仿真与时序仿真。

功能仿真，对于给出的程序，仿真器能够正确运行并给出与在真实系统上运行相近的结果；时序仿真，即从时序的角度，仿真器能够模拟出真实系统执行一个程序时，所需要消耗时间（时钟周期数）、功耗等参数。

从多芯粒系统的特点来看，对多芯粒系统进行仿真的工作可以分为对单个芯粒的仿真和对芯粒之间交互的仿真。

单个芯粒作为一颗 CPU 或 GPU 芯片，自身内部拥有计算单元、存储单元和网络结构，能够完成分配到的运算任务。对单个芯粒的仿真需要保证单个芯粒能够完成计算任务，并且得到每个计算任务的用时与功耗。

芯粒之间交互主要指不同芯粒之间的数据传输。对芯粒之间交互的仿真需要保证各个芯粒之间的数据传输功能可以被正确使用，并且对数据传输所需要的时间进行计算。

##### 1.3.2 仿真器的系统架构

本仿真器建立在三个开源仿真器的基础上——gpgpu-sim 仿真器、sniper 仿真器和 popnet 仿真器，下面先对它们分别做简要介绍，再介绍整个仿真器的系统架构。

##### 1.3.2.1 gpgpu-sim 简介

*gpgpu-sim* 官网: <http://www.gpgpu-sim.org/>

*gpgpu-sim* 提供了一个运行由 CUDA 编译器生成的 PTX 内核的当代 GPU 的详细仿真模型（支持多种 GPU 架构），以及一个集成的功耗模型。本仿真器目前使

用的 gpgpu-sim 版本是 3.2.0。

简言之，面对一个由 CUDA 编译器编译出的可执行文件，gpgpu-sim 能够根据要求的架构以及其他配置执行，执行过程中能够完成对真实 GPU 的功能的仿真，以及对真实 GPU 完成此项工作所需要的时间与功耗的计算。

在此基础上，本仿真器中的 gpgpu-sim 功能已被扩展（具体实现方法将在第三章介绍），在保留了原有的对单个 GPU 的仿真功能基础上，增加了向外发送数据与接收数据的 API 接口，使其能够实现对芯粒之间数据传输的功能仿真。

gpgpu-sim 主要包含三个主要模块（每一个在它的单独的文件夹中）：

- cuda-sim 模块，执行由 NVCC 或 OpenCL 编译器生成的 PTX 内核的功能模拟器
- gpu-sim 模块，模拟 GPU（或其他许多核心加速器架构）的时序行为的性能模拟器
- intersim 模块，采用 booksim 作为互连网络模拟器

对于 Gpgpu-sim 的工作细节以及对代码文件的简要说明可以参考网址：

[https://blog.csdn.net/qq\\_32535783/article/details/93487567?spm=1001.2101.3001.6650.10&utm\\_medium=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-10.no\\_search\\_link&depth\\_1-utm\\_source=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-10.no\\_search\\_link](https://blog.csdn.net/qq_32535783/article/details/93487567?spm=1001.2101.3001.6650.10&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-10.no_search_link&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-10.no_search_link)

#### 1.3.2.2 sniper 简介

Sniper 模拟器是 Trevor E. Carlson 等人开发的并行、高速且精确的 x86 模拟器。模拟器分为功能模块（又称为前端）和时间模块（又称为后端），前者执行程序并输出事件流，后者根据事件流模拟处理器内部的动作，估算时间。由于被模拟的程序是在系统中直接运行的，只是被插入了一些代码来做输出事件流、系统调用模拟等工作，速度相对较快。

Sniper 官方网站：<http://snipersim.org/>

代码中的 sniper-manual.pdf 是 Sniper 官方的用户手册。

#### 1.3.2.3 popnet 简介

popnet 是一款开源的互连网络模拟器，能够根据网络节点间的通信记录信息（trace 文件）计算出网络传递数据包的平均延迟以及总能耗。

#### 1.3.2.4 基于多芯粒集成 GPU 消息传递式仿真器系统架构

对于本仿真器来讲，一个 gpgpu-sim 进程完成对多芯粒系统中一个芯粒的仿真任务，并且通过拓展功能完成芯粒之间交互的功能仿真。Popnet 负责模拟芯粒间的网络，完成芯粒之间交互的时序仿真。具体的仿真任务如图 1 所示。

	功能仿真	时序仿真
芯粒间网络层	<ul style="list-style-type: none"> <li>Sniper和gpgpu-sim的芯粒间通信模块</li> </ul>	<ul style="list-style-type: none"> <li>Popnet</li> </ul>
芯粒层	<ul style="list-style-type: none"> <li>gpgpu-sim的cuda-sim 模块</li> <li>Sniper的function模块</li> </ul>	<ul style="list-style-type: none"> <li>gpgpu-sim的gpu-sim 模块</li> <li>gpgpu-sim的intersim模块</li> <li>Sniper的performance模块</li> </ul>

图 1. 仿真器中不同组件所负担的仿真任务

在本仿真器工作时，gpgpu-sim 或 sniper 作为基础的工作单元需要根据仿真需求同时启动多个进程，popnet 作为芯粒间网络层的时序仿真工具只需要启动一个进程。图 2 展示了在仿真四个芯粒集成的多核系统时，本仿真器不同部件之间如何协同工作。

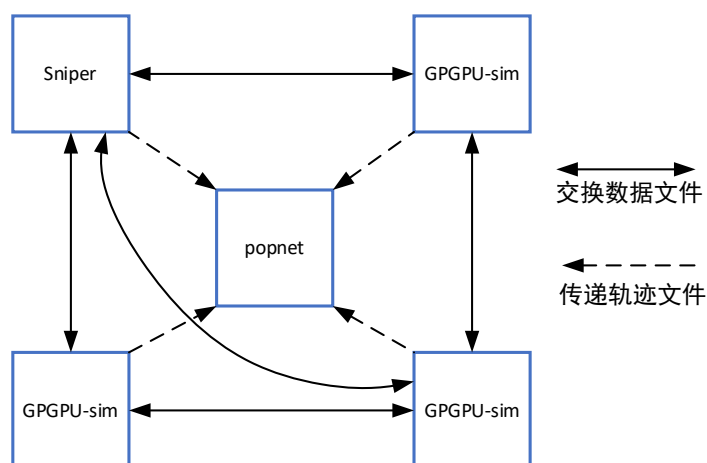


图 2. 以四芯粒为例，仿真器不同构成部分间的依赖关系

## 1.4 仿真器的工作原理

本小节从仿真器执行程序的角度介绍仿真器的工作原理。

### 1.4.1 单个芯粒的仿真原理

#### 1.4.1.1 单个 GPU 芯粒的仿真原理

gpgpu-sim 仿真器的基础单元是单指令多线程（SIMT）内核，一个 SIMT 内核的功能相当于 GPU 中 NVIDIA 称之为流式多处理器(SM)或 AMD 称为计算单元(CU)的处理器，可以执行访问存储器指令和 ALU 指令。

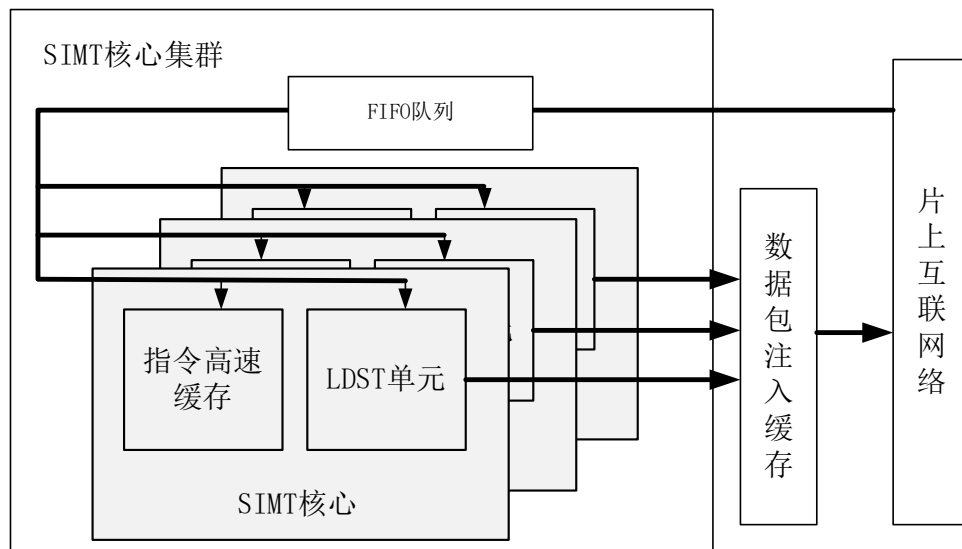


图 3. 单个 SIMT 核心集群的基础架构

SIMT 核心被组织为 SIMT 核心集群。每个 SIMT 核心集群中的 SIMT 核心共享一个连接到互连网络的公共端口，一个 SIMT 核心集群的架构如图 3 所示。每个 SIMT 核心集群都有一个响应 FIFO 队列，用于保存从互连网络中弹出的数据包。数据包被定向送到 SIMT 核心的指令高速缓存或负责访问存储的内存流水线单元（LDST 单元）。为了在 LDST 单元处生成访问存储器请求，每个 SIMT 核心具有其到互连网络中注入端口。但是，注入端口缓存由集群中的所有 SIMT 核共享。

仿真器中同时具有多个 SIMT 核心集群，他们直接并行工作，并独立的与片上网络和内存交换数据，SIMT 核心集群的组织方式如图 4 所示。这些 SIMT 核心集群通过片上互连网络连接到内存。

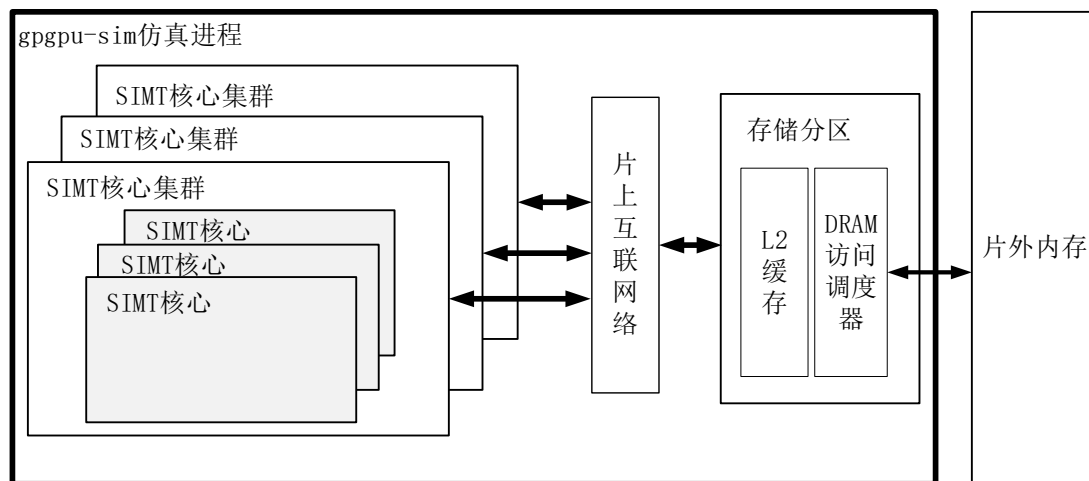


图 4. gpgpu-sim 仿真器的顶层设计

仿真器工作时，首先由 nvcc 编译器将程序编译为 PTX 指令构成的可执行文件，并将其载入 GPU 内存中。片上互连网络将指令载入 SIMT 核心集群，SIMT 核心集群将指令分发至其下所有的 SIMT 核心进行执行。

#### 1.4.2 多核系统的仿真原理

对于多核系统来说，其顶层设计如图 5 所示。首先，由 sniper 进程仿真的 CPU 芯粒向所有的 gpgpu-sim 进程仿真的 GPU 芯粒分发任务。并将数据从系统内

存中装载入 GPU 内存。由 gpgpu-sim 仿真的 GPU 芯粒之间通过片上互连网络相连。每个 gpgpu-sim 的 SIMT 核心集群可以直接与片上网络进行数据交换。

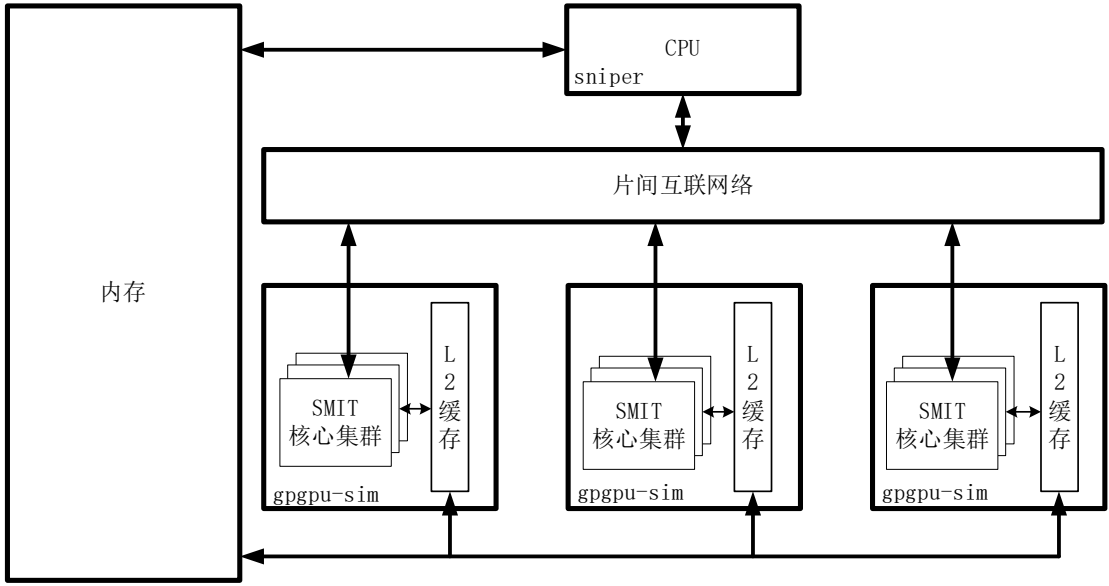


图 5. 多芯粒仿真的顶层设计

芯粒间通过片间互连网络收发消息的工作原理如图 6 所示。每个芯粒拥有一片独立的存储区域用于接收其他芯粒发送的数据。当一个芯粒向目标芯粒发送数据时，数据包经过片间互连网络写入目标芯粒对应的存储中，并使用自身的芯粒号标记数据。当一个芯粒需要从其他芯粒接收数据时，它在自己对应的片上网络存储中根据数据来源芯粒的标记号对数据进行检索，检索到对应数据后将其读入自己片内存储，并将缓存清空。

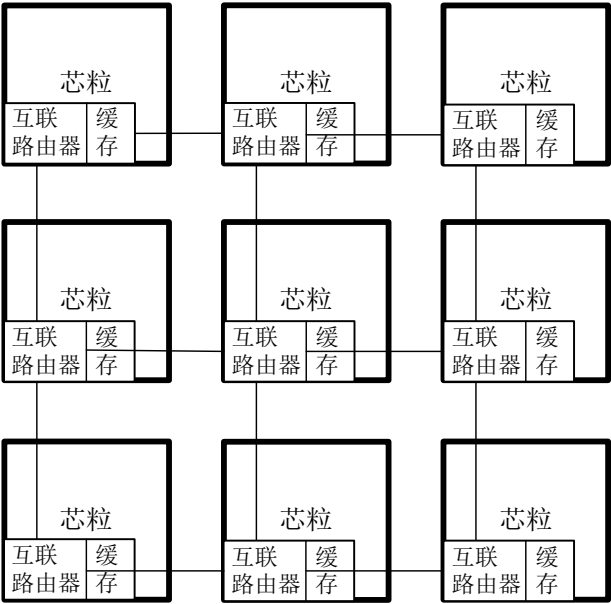


图 6. 芯粒间通过片间互连网络进行数据传输示意图

片间互连网络主要通过仿真器进程之间的文件读写实现。文件主要包括三种，trace 文件、数据文件和同步文件。

仿真器运行过程中，trace 文件的主要功能是保存多个芯粒之间的通信记录。每个芯粒（sniper 进程或 gpgpu-sim 进程）负责维护一个单独的 trace 文件。

Trace 文件的具体内容包括：通信发起的时间，发起通信的芯粒编号，通信目标的芯粒编号，以及本次通信的数据大小。每当一个芯粒试图向其他芯粒发起通信时，Trace 文件就会被写入一条新的通信记录。在仿真器功能仿真结束之后，trace 文件经过处理之后作为时序仿真的输入文件。

数据文件的主要功能是暂存芯粒之间通信的数据内容。每当一个芯粒向其他某个芯粒发起通信时，它的仿真进程将产生一个数据文件并将通信内容写入数据文件中。在一次通信中，负责接收信息的芯粒的仿真进程通过识别并读取数据文件来接收数据。

同步文件的主要功能是保证多个仿真器进程能够以一致的速度进行工作，以保证仿真的精确度。

1.4.2.1 数据文件

数据文件由一个芯粒（gpgpu-sim 进程或 sniper 进程）第一次向其他芯粒发送数据时产生，并在每次数据发送时更新。主要功能是从功能仿真的角度，实现芯粒间的数据发送与接收。数据文件的文件名格式为：

“buffer\_%d<sub>1</sub>\_%d<sub>2</sub>\_%d<sub>3</sub>\_%d<sub>4</sub>”，d<sub>1</sub>为数据目标芯粒的 x 坐标，d<sub>2</sub>为数据目标芯粒的 y 坐标，d<sub>3</sub>为本芯粒的 x 坐标，d<sub>4</sub>为本芯粒的 y 坐标。

芯粒主要通过识别文件名来判断自己是否是数据的接收方，以及辨识数据的发送方。在数据文件内部，数据按写入顺序排列，每个数据占一行，数据大小为 32 字节。一个典型的数据文件如下：

文件名: buffer_0_3_2_2
199
2003
4447
.....
3330

该文件的含义为：本文件是标号为（0，3）的芯粒向标号为（2，2）的芯粒发送的数据文件。（0，3）号芯粒可以在整个仿真过程中不断的向此文件中写入数据，而（2，2）号芯粒在每次读取完文件内容后会将此文件清空。

1.4.2.2 trace 文件

trace 文件由一个芯粒（gpgpu-sim 进程或 sniper 进程）第一次向外发送数据时创建，并在每次发送数据时不断更新。主要功能是从时序仿真的角度，记录本芯粒向外发送数据的时间、目标与数据大小。trace 文件的文件名格式为：

“bench.%d<sub>1</sub>\_%d<sub>2</sub>”，d<sub>1</sub>为本芯粒的 x 坐标，d<sub>2</sub>为本芯粒的 y 坐标。

trace 文件内部的一行表示一次数据发送记录，格式如下：T sx sy dx dy n  
T 表示本次数据发送的时间，sx 和 sy 表示本芯粒的 x 坐标与 y 坐标，dx 和 dy 表示目标芯粒的 x 坐标与 y 坐标，n 表示发送数据的大小。

一个典型的 trace 文件如下所示：

文件名: bench.1.3
16788 1 3 6 3 4
18812 1 3 0 2 4
.....
22012 1 3 1 2 4

该文件记录了自仿真开始依赖，标号为（1，3）的芯粒所有向外发送数据的记录。例如文件第一行的含义是：在芯粒的第 16788 个时钟周期向标号为（6，3）的芯粒发送了大小为 32 字节的数据。

芯粒每次向外发送 32 字节数据时，它先将数据内容写入数据文件中，再在 trace 文件中添加一行内容。

#### 1.4.2.3 读/写操作流程

芯粒在片间互连网络中有两种主要行为：写操作和发出读请求。写操作表示芯粒要向其他芯粒传输数据；读操作表示芯粒需要获取其他芯粒传来的数据。

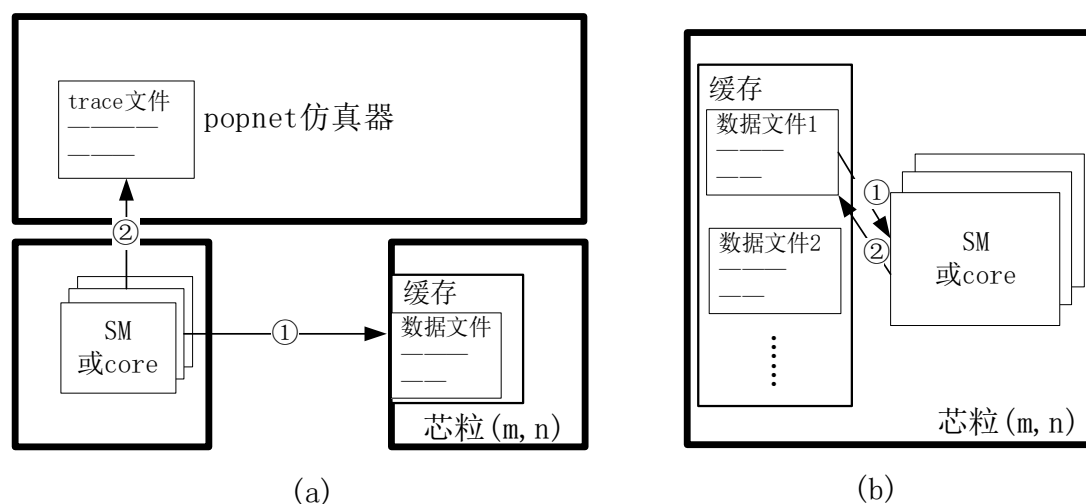


图 7. (a) 芯粒间写操作流程示意图; (b) 芯粒读操作流程示意图

当芯粒进行写操作时，仿真器工作流程如图 7（a）所示。

- ① 芯粒 i 的 SMIT 核心集群（GPU 中）或 core（CPU 中）将数据写入位于芯粒 j 缓存中的数据文件“buffer\_m\_n\_x\_y”。
- ② 芯粒 i 的 SMIT 核心集群或 core 将本次数据发送记录写入由 popnet 处理的 trace 文件“bench. x. y”中。

当芯粒进行读操作时，仿真器工作流程如图 7（b）所示。

- ① 芯粒的 SMIT 核心集群或 core 在本芯粒的缓存中根据文件名称进行检索，查找到正确的数据源芯粒发送来的文件，
- ② 芯粒的 SMIT 核心集群或 core 读取其中内容后，清空该数据文件。

#### 1.4.2.4 同步文件

同步文件由所有芯粒共同维护，主要功能是同步所有芯粒的时钟周期数，以保证它们之间的数据依赖。通过同步文件同步各仿真进程的时钟周期在本版本暂未实现，将在未来版本在更新。对于同步文件的规定如下：同步文件在全部仿真进程中只有一个，其文件名为“synchronization”。

同步文件的文件格式如下所示：

文件名: synFile
12000
20000
0, 2
1, 2
.....
2, 1

第一行是一个整数，表示当前的同步时钟周期。第二行表示距离下一次同步的时钟周期间隔。第三行及之后各行代表芯粒标号。假设当前仿真系统的总芯粒数位 9 个，那么 synchronization 至多有 11 行。

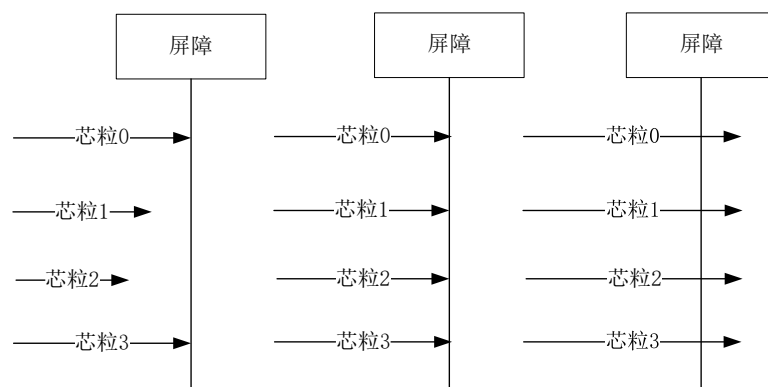


图 8. 同步文件的作用示意

同步文件的作用如同在芯粒运行过程中设置一个屏障，如图 7 所示。所有芯粒的仿真进程由箭头表示，箭头从左至右增长，表示时钟周期数增加。时钟周期数增加较快仿真进程在碰到屏障时将被阻拦并暂停运行，直至所有芯粒仿真进程的时钟周期数全部到达屏障后，所有进程再一同向下运行。

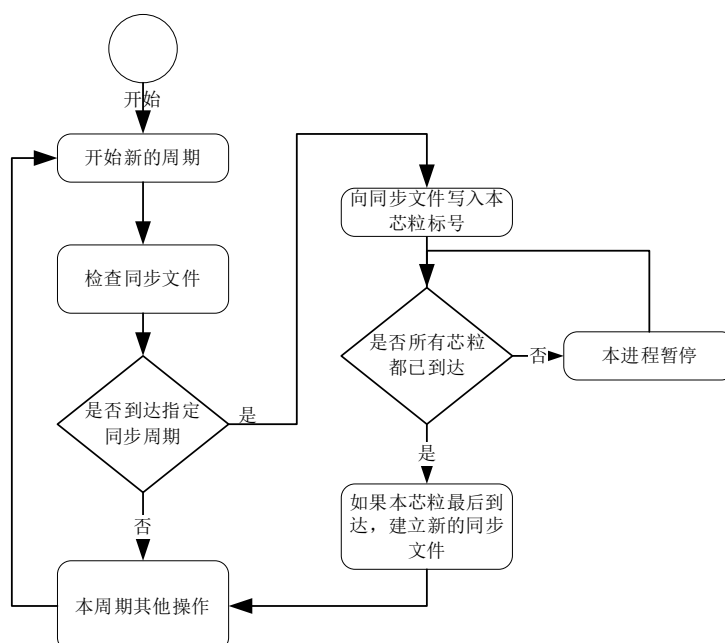


图 9. 芯粒仿真进程同步流程图

各芯粒在运行时，会在每个时钟周期检查是否到达同步文件要求的时钟同步



周期。如果达到，本芯粒暂停运行，并向同步文件尾部写入一行本芯粒标号。如未达到，本芯粒继续运行。暂停运行的芯粒会不断检查文件行数是否达到最大，达到最大则说明所有芯粒都已运行至要求的同步时钟周期，则本芯粒继续运行，最后一个到达同步时钟周期的芯粒在开始运行后清空同步文件，并向第一行写入新的时钟同步周期，向第二行写入下一次时钟同步周期。

## 2. 仿真器安装与使用

仿真器的安装过程包括如下步骤：（1）安装 gpgpu-sim；（2）安装 popnet。（3）安装 sniper，在本章中，约定所有组件安装在目录 ROOT/目录下：

gpgpu-sim 安装在目录 ROOT/gugpu-sim\_distribution

popnet 安装在目录 ROOT/popnet

sniper 安装在目录 ROOT/ intersim/sniper

### 2.1 gpgpu-sim 的安装

#### 2.1.1 cuda4.0 的安装

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

下载链接：<https://developer.nvidia.com/cuda-toolkit-40>

本文使用的 Gpgpu-sim 版本只支持到 cuda 4.0

#### 2.1.1.2 安装 CUDA toolkit

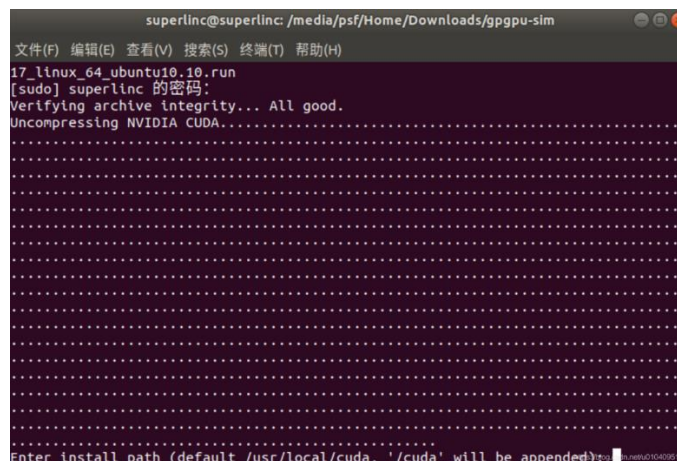


图 10. CUDA toolkit 的安装过程

赋予 cudatoolkit 文件可执行权限并执行，指令：

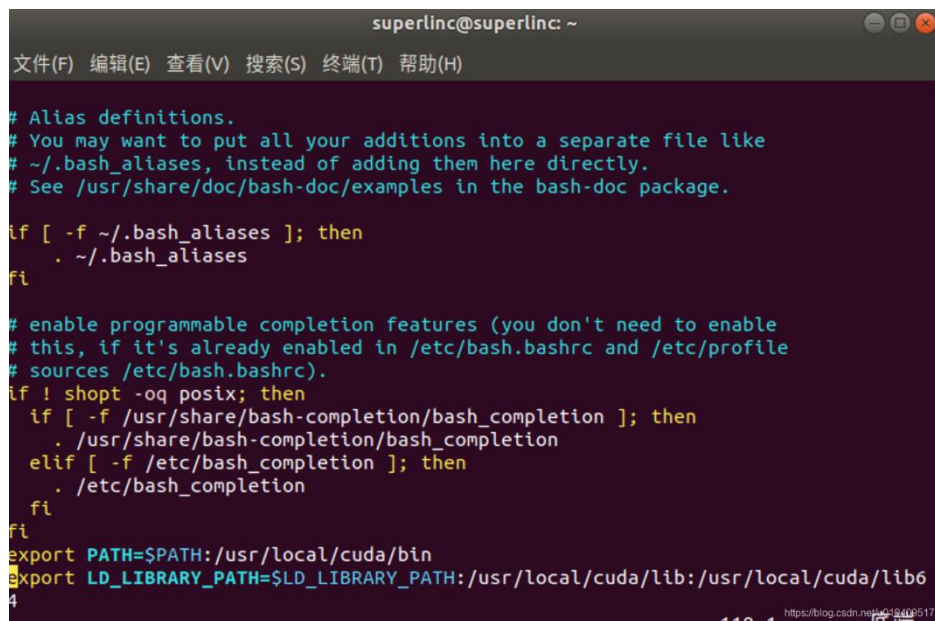
```
chmod +x cudatoolkit_4.0.17_linux_64_ubuntu10.10.run  
sudo ./cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
```

默认安装在/usr/local/cuda。

2.1.1.3 增加 CUDA toolkit 到 ~/.bashrc 中，添加环境变量  
指令：

```
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
```

可看到`~/.bashrc` 底部两行已加入路径。



```
superlinc@superlinc: ~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
  
# Alias definitions.  
# You may want to put all your additions into a separate file like  
# ~/.bash_aliases, instead of adding them here directly.  
# See /usr/share/doc/bash-doc/examples in the bash-doc package.  
  
if [ -f ~/.bash_aliases ]; then  
    . ~/.bash_aliases  
fi  
  
# enable programmable completion features (you don't need to enable  
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile  
# sources /etc/bash.bashrc).  
if ! shopt -oq posix; then  
    if [ -f /usr/share/bash-completion/bash_completion ]; then  
        . /usr/share/bash-completion/bash_completion  
    elif [ -f /etc/bash_completion ]; then  
        . /etc/bash_completion  
    fi  
fi  
export PATH=$PATH:/usr/local/cuda/bin  
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib:/usr/local/cuda/lib64
```

图 11. `~/.bashrc` 文件更改后内容

#### 2.1.1.4 安装 GPU Computing SDK code samples

指令:

```
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。

#### 2.1.1.5 安装 gcc-4.4 和 g++-4.4 (CUDA 4.0 只支持 gcc 版本到 4.4)

由于 Ubuntu 18.04 自带 7.4.0 版本 gcc, 无法直接安装 4.4 版本的 gcc 可通过以下方法修改:

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

底部增加两行代码, 按 I 插入:

```
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, 按 `esc`, 然后按 `:wq`, 保存退出。

更新 apt 源:

```
sudo apt-get update
```

再重新安装 gcc-4.4 和 g++-4.4 就可以了

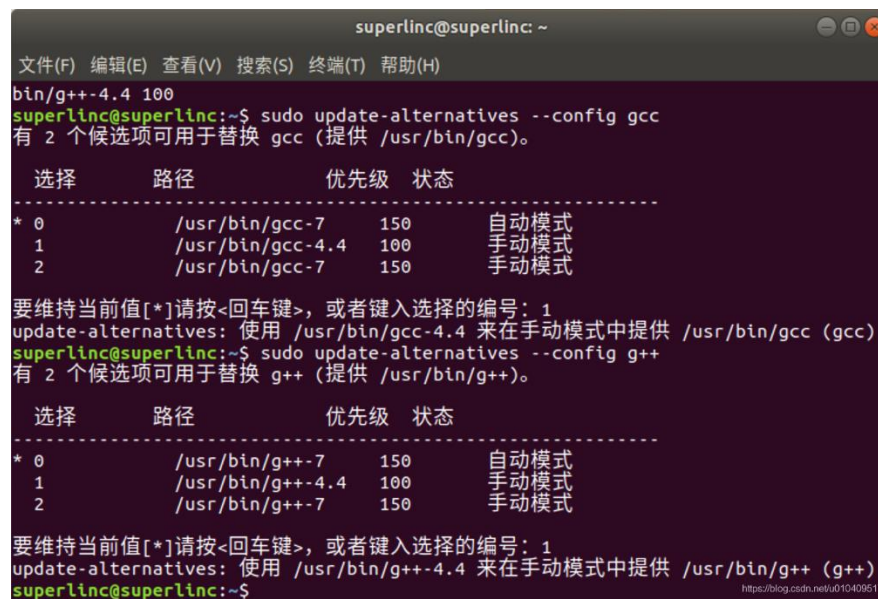
```
sudo apt-get install gcc-4.4 g++-4.4
```

### 2.1.1.6 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
```



```
superlinc@superlinc: ~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
bin/g++-4.4 100
superlinc@superlinc:~$ sudo update-alternatives --config gcc
有 2 个候选项可用于替换 gcc (提供 /usr/bin/gcc)。

  选择      路径              优先级  状态
-----
* 0          /usr/bin/gcc-7          150     自动模式
  1          /usr/bin/gcc-4.4        100     手动模式
  2          /usr/bin/gcc-7          150     手动模式

要维持当前值[*]请按<回车键>, 或者键入选择的编号: 1
update-alternatives: 使用 /usr/bin/gcc-4.4 来在手动模式中提供 /usr/bin/gcc (gcc)
superlinc@superlinc:~$ sudo update-alternatives --config g++
有 2 个候选项可用于替换 g++ (提供 /usr/bin/g++)。

  选择      路径              优先级  状态
-----
* 0          /usr/bin/g++-7          150     自动模式
  1          /usr/bin/g++-4.4        100     手动模式
  2          /usr/bin/g++-7          150     手动模式

要维持当前值[*]请按<回车键>, 或者键入选择的编号: 1
update-alternatives: 使用 /usr/bin/g++-4.4 来在手动模式中提供 /usr/bin/g++ (g++)
superlinc@superlinc:~$
```

图 12. 修改 gcc 与 g++ 版本

### 2.1.2 下载安装 gpgpu-sim

#### 2.1.2.1 下载源码

GitHub 链接: <https://github.com/FCAS-SCUT/Chiplet-Gpgpu-sim-MessagePassing>

#### 2.1.2.2 安装依赖项

```
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
```

#### 2.1.2.3 添加 CUDA\_INSTALL\_PATH 到 ~/.bashrc 中

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

#### 2.1.2.4 编译 gpgpu-sim

```
source setup_environment  
  
make  
  
make docs
```

#### 2.1.3 使用 gpgpu-sim 测试运行 CUDA 程序

此小节是为了测试 gpgpu-sim 安装无误，读者暂且不必深究各项操作的含义，它们将在使用部分进行详细的介绍。

##### 2.1.3.1 编写 CUDA 程序并编译

一个简单的示例如下，其文件后缀需要为 .cu：

```
#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;  
}
```

在文件夹下运行命令，将其编译为可执行文件。

```
nvcc 文件名.cu -o 文件名.out
```

将刚才生成的可执行文件复制到 gpgpu-sim\_distribution 文件夹下，并将 /configs/GTX480 中的三个文件复制出来。/configs/GTX480 中的三个文件为预先设定好的单个 GPU 芯粒的配置文件。

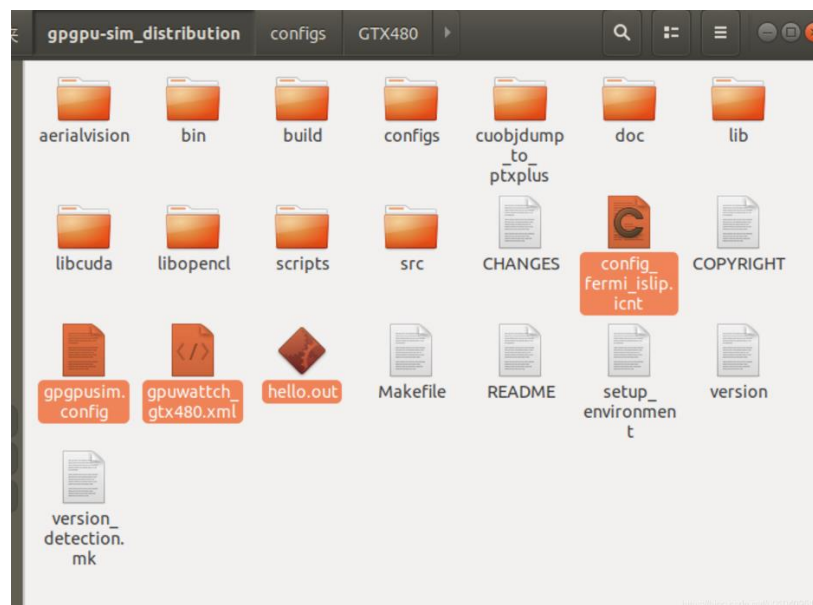


图 13. gpgpu-sim\_distribution 文件夹下要新增的文件

在此路径中运行

```
source setup_environment
./hello.out

-----END-of-Interconnect-DETAILS-----
gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 1 sec (1 sec)
gpgpu_simulation_rate = 1 (inst/sec)
gpgpu_simulation_rate = 278 (cycle/sec)
Hello world!
```

图 14. 仿真程序运行完成图

如顺利运行，则 gpgpu-sim 安装完毕。

## 2.2 popnet 的安装

### 2.2.1 下载 popnet 源码：

```
git clone https://gitee.com/hic_0757/popnet_modified.git
```

### 2.2.2 编译 popnet 仿真器

```
make
```

## 2.3 sniper 的安装

注意：本节需要管理员权限。请确保g++和gcc 的版本为7。

### 2.3.1 编译安装依赖库，依次运行以下命令：

#### A. cmake:

```
sudo apt install cmake
```

#### B. Sniper 的依赖库：

```
sudo dpkg --add-architecture i386

sudo apt-get install binutils build-essential curl git libboost-dev libbz2-dev libc6:i386 libncurses5:i386
libsqlite3-dev libstdc++6:i386 python wget zlib1g-dev
```

### 2.3.2 编译Sniper

#### A. 从以下网址下载PIN 工具包：

<https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html>

#### B. 将压缩包解压，复制到ROOT/intersim/sniper 中，并重命名为pin\_kit。

#### C. 在ROOT/intersim/sniper 执行编译命令：

```
make
```

#### D. 在ROOT/intersim/sniper/interchiptlet\_app 下运行。

```
sh build.sh
```

## 2.3 仿真器的输入文件与输出结果

### 2.3.1 仿真器的输入文件

2.3.1.1. gpgpu-sim 的输入文件

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：  
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。  
gpuwattch\_\*.xml 主要规定了计算功耗时所需要的各项参数内容。  
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。  
对于 gpgpu-sim 的输入文件如何准备，即如何使用 gpgpu-sim 的配置文件以及如何编译代码生成可执行文件将在第四章进行详细说明。

2.3.1.2. Popnet 的输入文件

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

启动脚本文件需要本仿真器用户根据所需要仿真的多芯粒系统进行确定。如何得到 trace 文件将在使用部分进行详细说明。

2.3.2 仿真器的时序仿真结果

2.3.2.1 gpgpu-sim 的时序仿真结果

gpgpu-sim 的仿真结果包括两部分，第一部分是本芯粒的功耗报告文件，第二部分是芯粒执行程序所消耗的时钟周期数。



在仿真结束后，功耗报告文件自动生成于 Gpgpu-sim 仿真器的根目录中。芯粒执行程序所消耗的时钟周期数由控制台打印输出，内容如图 15 所示：

```
gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 22 sec (22 sec)
gpgpu_simulation_rate = 127163 (inst/sec)
gpgpu_simulation_rate = 5083 (cycle/sec)
```

图 15. Gpgpu-sim 进程运行结束时控制台输出

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。

#### 2.3.2.2 Popnet 的时序仿真结果

Popnet 的仿真结果由控制台打印输出，如图 16 所示：

```
*****
total finished:      11983
average Delay:       37.1263
total mem power:     27.7021
total crossbar power: 2.1895
total arbiter power: 0.00207568
total link power:    3.51097
total power:         33.4046
*****
```

图 16. Popnet 运行结束时控制台输出

第一行 total finished 表示总的的数据发送量（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，仿真结果中被关注的是 average Delay。

#### 2.3.2.3 Sniper 的时序仿真结果

#### 2.3.2.4 本仿真器总的时序仿真结果

本次仿真的总功耗计算：每个 gpgpu-sim 功耗报告文件中的功耗加上 sniper 运行结果中的功耗加上 Popnet 输出的总功耗。

本次仿真的总时间周期数计算：本次仿真所需要的同步次数乘以 Popnet 输出的 average Delay 加上所有 gpgpu-sim 进程中最大的总时钟周期数。

### 2.4 使用本仿真器前的准备

使用本仿真器需要了解两方面知识：第一，如何编写本仿真器能够执行的程序；第二，如何使用本仿真器执行仿真。下面首先介绍编写本仿真器能够执行的程序的基础知识，随后介绍本仿真器的仿真流程和使用步骤。仿真器的编程模型将在第四章进行介绍。

#### 2.4.1 CUDA 编程的简单介绍

CUDA 编程模型假设系统是由一个主机（CPU）和一个设备（GPU）组成的，而且各自拥有独立的内存。程序员需要做的就是编写运行在主机和设备上的代码，并且根据代码的需要为主机和设备分配内存空间以及拷贝数据。而其中，运行在设备上的代码，我们一般称之为核函数（Kernel），核函数将会由大量硬件线程并行执行。

一个典型的 CUDA 程序是按这样的步骤执行的：

- 把数据从 CPU 内存拷贝到 GPU 内存。
- 调用核函数对存储在 GPU 内存中的数据进行操作。
- 将数据从 GPU 内存传送回 CPU 内存。

下面先简单介绍 CUDA 编程中比较重要基础的两个函数。

#### 2.4.2 cudaMalloc() 函数

cudaMalloc() 的主要作用是在向 GPU 内存申请空间，它的函数声明如下：

**cudaMalloc( (void\*\*) &data\_in\_CPU, sizeof(datatype)\*data\_size)**

其中第一个参数是存储在 cpu 内存中的指针变量的地址，第二个参数是需要申请的内存空间大小。

一个简单的示例如下：

```
float *device_data=NULL;

size_t size = 1024*sizeof(float);

cudaMalloc((void**)&device_data, size);
```

上面这个例子中我在显存中申请了一个包含 1024 个单精度浮点数的一维数组。而 device\_data 这个指针是存储在主存上的。之所以取 device\_data 的地址，是为了将 cudaMalloc 在显存上获得的数组首地址赋值给 device\_data。

#### 2.4.3 cudaMemcpy() 函数

cudaMemcpy() 函数的主要作用是将 CPU 内存中的数据传递给 GPU 内存，或者将 GPU 内存中的数据传递回 CPU 内存。它的函数声明如下：

**cudaError\_t CUDARTAPI cudaMemcpy(void \*dst, const void \*src, size\_t count, enum cudaMemcpyKind kind);**

首先介绍第四个参数，第四个参数常用有两种取值：**cudaMemcpyHostToDevice** 或者 **cudaMemcpyDeviceToHost**。**cudaMemcpyHostToDevice** 表示本次函数调用将数据从 CPU 内存传递至 GPU 内存，**cudaMemcpyDeviceToHost** 表示本次函数调用将数据从 GPU 内存传递至 CPU 内存。

第三个参数表示传递数据的大小。

第二个参数 src 表示数据源地址，第一个参数 dst 表示数据目标地址。

一个简单示例如下：

```
int Layer1_Weights_CPU[156]; //初始化部分省略

//向 GPU 内存申请数据

int *Layer1_Weights_GPU;

cudaMalloc((void**) &Layer1_Weights_GPU, sizeof(int)*156);

//将 CPU 内存中的数据传递给 GPU 内存

cudaMemcpy(Layer1_Weights_GPU, Layer1_Weights_CPU, sizeof(int)*156, cudaMemcpyHostToDevice);
```

#### 2.4.4 核函数(kernel function)



一个简单的 cuda 核函数如下：

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;

    c[i] = a[i] + b[i];
}
```

函数 addKernel 在最前有一个修饰符 “\_\_global\_\_”，这个修饰符告诉编译器，被修饰的函数应该编译为在 GPU 而不是在 CPU 上运行，所以这个函数将被交给编译设备代码的编译器——NVCC 编译器来处理，其他普通的函数或语句将交给主机编译器处理。

这个核函数里有一个陌生的 threadIdx.x，表示的是 thread 在 x 方向上的索引号，GPU 线程的层次结构如图 17 所示：

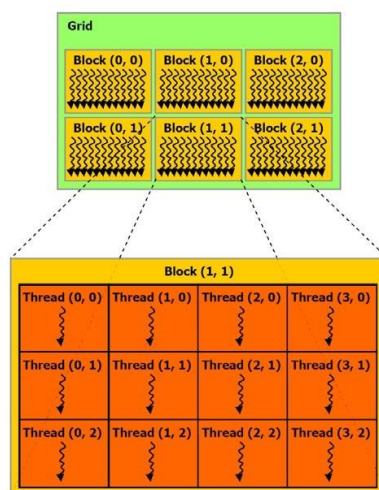


图 17. GPU 线程的层次结构

CUDA 中的线程（thread）是设备中并行运算结构中的最小单位，类似于主机中的线程的概念，thread 可以以一维、二维、三维的形式组织在一起，threadIdx.x 表示的是 thread 在 x 方向的索引号，还可能存在 thread 在 y 和 z 方向的索引号 threadIdx.y 和 threadIdx.z。

一维、二维或三维的 thread 组成一个线程块（Block），一维、二维或三维的线程块（Block）组合成一个线程块网格（Grid），线程块网格（Grid）可以是一维或二维的。通过网格块（Grid）->线程块（Block）->线程（thread）的顺序可以定位到每一个并且唯一的线程。

一个更为复杂的核函数如下：

```
__global__ void executeFirstLayer(float *Layer1_Neurons_GPU,float *Layer1_Weights_GPU,float *Layer2_Neurons_GPU)
{
    int blockID=blockIdx.x;

    int pixelX=threadIdx.x;

    int pixelY=threadIdx.y;

    int weightBegin=blockID*26;

    int windowX=pixelX*2;

    int windowY=pixelY*2;

    float result=0;

    result+=Layer1_Weights_GPU[weightBegin];

    ++weightBegin;

    for(int i=0;i<25;++i)
    {
        result+=Layer1_Neurons_GPU[(windowY*29+windowX+kernelTemplate[i])+(29*29*blockIdx.y)]*Layer1_Weights_GPU[weight
Begin+i];
    }

    result=(1.7159*tanhf(0.66666667*result));

    Layer2_Neurons_GPU[(13*13*blockID+pixelY*13+pixelX)+(13*13*6*blockIdx.y)]=result;
}
```

在 main 函数中，对于 kernel 函数的调用方法如下：

```
dim3 Layer1_Block(6,NUM,1);

dim3 Layer1_Thread(13,13);

executeFirstLayer<<<Layer1_Block,Layer1_Thread>>>>(Layer1_Neurons_GPU,Layer1_Weights_GPU,Layer2_Neurons_GPU);
```

“<<<>>>”表示运行时配置符号，“<<<>>>”中的参数并不是备代码的参数，而是定义主机代码运行时如何启动设备代码。

#### 2.4.5 gpgpu-sim 仿真器芯粒间通讯函数

本函数的主要功能是实现多芯粒系统中 gpu 芯粒之间的通信，gpgpu-sim 中芯粒间的通讯接口函数主要通过 gpgpu-sim 仿真器没有实现的 PTX 指令实现（类似于汇编指令）。选择的具体函数如下：

```
asm("addc.u32 %0, %1, %2;" : "=r"(*n) : "r"(*m) , "r"(*n));
```

函数接收两个参数，参数 m 是一个 9 位数，假设九位从大到小分别为 abcdefghi，其不同位的含义如下：

ab 表示 src 的 x 坐标，  
cd 表示 src 的 y 坐标，  
ef 表示 dst 的 x 坐标，  
gh 表示 dst 的 y 坐标，

i 表示是向其他 chiplet 发送数据还是读取其他芯粒传递来的数据。i=0 表示发送数据，i=1 表示接收数据。

参数 n 是要发送的数据，数据大小为一个 int。

举例说明：

```
__global__ void kernel(void){  
    int aaa=101010;  
    int bbb=1111;  
    int *m=&aaa;  
    int *n=&bbb;  
    asm("addc.u32 %0, %1, %2;" : "=r"(*n) : "r"(*m), "r"(*n));  
}
```

上方例子表示，坐标为（0，1）的 Chiplet 向坐标为（1，1）的芯粒发送了一个数据，数据内容为 1111。

发送的数据文件名为“buffer\_ $d_1$ \_ $d_2$ \_ $d_3$ \_ $d_4$ ”， $d_1$ 为数据目标芯粒的 x 坐标， $d_2$ 为数据目标芯粒的 y 坐标， $d_3$ 为本芯粒的 x 坐标， $d_4$ 为本芯粒的 y 坐标。

#### 2.4.6 sniper 仿真器芯粒间通信函数

Sniper 仿真器的片间通讯函数主要用于实现多芯粒系统中 CPU 芯粒向 GPU 芯粒的通信，sniper 中的芯粒间的通讯接口函数主要通过文件读写函数实现。

函数一：

```
void passGpuMessage(int dstX, int dstY, int srcX, int srcY, int* data, int dataSize)
```

该函数接受一个存有待发送数据的数组，并将其发送到目标芯粒，同时生成或维护 trace 文件。参数含义如下：

dstX, dstY 表示本次通信的目标芯粒坐标，srcX, srcY 表示系统中 CPU 芯粒的坐标，data 为存放数据的数组地址，dataSize 表示数据的个数。

函数二：

```
void readGpuMessage( int srcX, int srcY, int dstX, int dstY, int*data, int dataSize)
```

该函数接收其他芯粒传输给本芯粒的数据，并将其存入制定的地址中。参数含义如下：

srcX, srcY 表示本次通信的数据来源芯粒坐标，dstX, dstY 表示系统中 CPU 芯粒的坐标，data 为存放数据的数组地址，dataSize 表示数据的个数。

## 2.5 仿真器的使用

### 2.5.1 仿真器工作流程

本仿真器的工作流程如图 18 所示

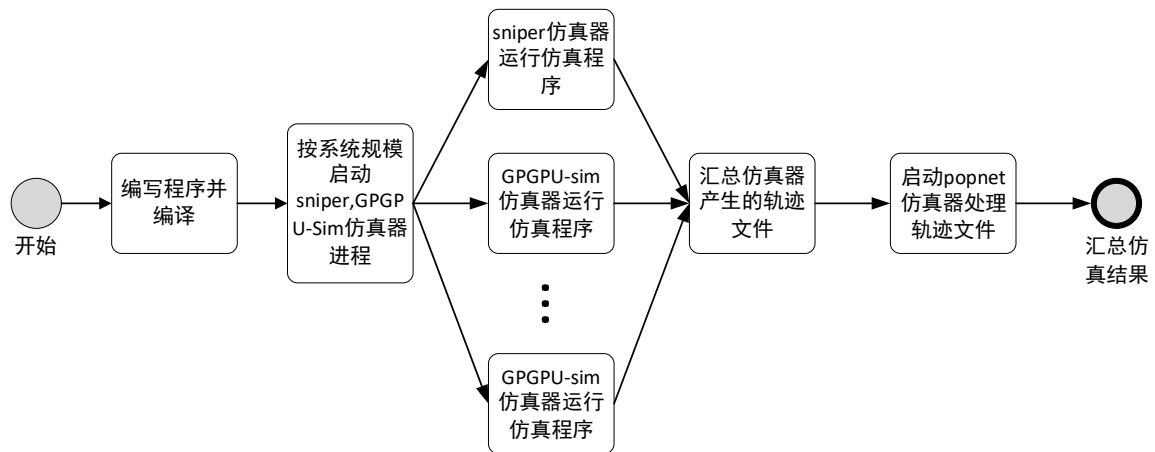


图 18. 仿真器工作流程图

Step1:编写多机版本的程序。

Step2:将刚才生成的可执行文件复制到 `gugpu-sim_distribution` 文件夹下，并将 `/configs/GTX480` 中的三个文件复制出来。`/configs/GTX480` 中的三个文件为预先设定好的单个 GPU 芯粒的配置文件。此处的配置文件可以根据实验需求进行修改。

Step3:分别启动运行于 sniper 仿真器上的 CPU 运行程序和 `gpgpu-sim` 上的 GPU 运行程序。

Step4:使用 popnet 计算片间通信耗时 (cycle 数)

在每次调用通讯接口函数之后，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。

当 popnet 将所有 `trace` 记录处理完成后，会显示仿真结果。将仿真结果中的 `cycle` 数乘以 16 后，与 `gpgpu-sim` 运行完成后所显示的总 `cycle` 数相加，即为本次仿真的总耗时。

### 3. 程序的编写与工作负载的映射

矩阵乘法是神经网络中的基础操作之一，应用范围广泛，本章节将以矩阵乘法为例，说明在设计本仿真器能够运行的多芯粒程序的过程中的编程模型，负载映射与注意事项。

#### 3.1 分析程序的任务如何并行化

设计程序的第一步是分析程序的任务如何并行化。在由 1 个 CPU 和多个 GPU 组成的系统中，CPU 通常负责任务和数据的分发，以及接收 GPU 传送回的运行结果。GPU 芯粒负责完成大部分计算任务，并且这些计算任务应该进行并行化以适应 GPU 的结构特点。并行化分为两个层级：如何并行化程序使其适合多芯粒系统运行；如何并行化程序使其负载能够映射到每个芯粒上。

##### 3.1.1 如何并行化程序使其适合多芯粒系统运行

根据 2.3.4 小节介绍 GPU 的架构特点，我们需要将一个程序划分为众多操作相同的较小的线程，同时这些线程最好在时序是不存在数据依赖。以对矩阵乘法的分析为例：

系统架构：1 CPU 芯粒（负责分发任务，汇总结果） + 4 GPU 芯粒（负责矩阵计算）

程序的输入：大小为  $m \times n$  的输入矩阵  $Z$  与大小为  $n \times k$  的输入矩阵  $W$

矩阵的输出：结果矩阵 E  
 矩阵乘法的操作过程如图 22 所示：

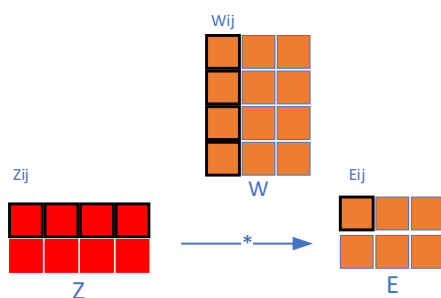


图 22. 矩阵乘法的计算过程

操作过程的两个特点如下：首先，对于结果矩阵 E 来讲，位于  $(i, j)$  位置的数值与位于其他位置，如  $(i+1, j)$ ，的数据没有依赖关系。其次，计算结果矩阵中每个位置的数值时，操作完全相同。

因此，我们可以根据结果矩阵 E 中的大小来确定线程数，并将计算结果矩阵中每个数值的任务映射到线程上，我们将得到  $m \times k$  个线程。每个线程的工作任务是计算结果矩阵中的一个值。

### 3. 1. 2 如何并行化程序使其负载映射到每个芯粒

适用于本部分的并行化主要有两种：按范围分解与按功能分解。

按范围分解：使用这种方法程序所需的数据会被分解；不同的芯粒会使用同一个程序处理不同的数据。这个方法一般在需要处理大量数据的情况下使用。

按功能分解：使用这个分解方法会将问题分解为几个任务，每个任务会对可利用的数据执行不同的操作。执行不同任务的芯粒可以想流水线一样进行工作。

分析矩阵乘法可知它有如下特点：如图 23 (a) 所示，对于两个较大的矩阵，可以使用图 23 (b) 的方法将它们平均切分为几个较小的部分。再分别计算若干个较小的矩阵乘法，并将结果相加，得到的结果与直接进行较大的矩阵相等。

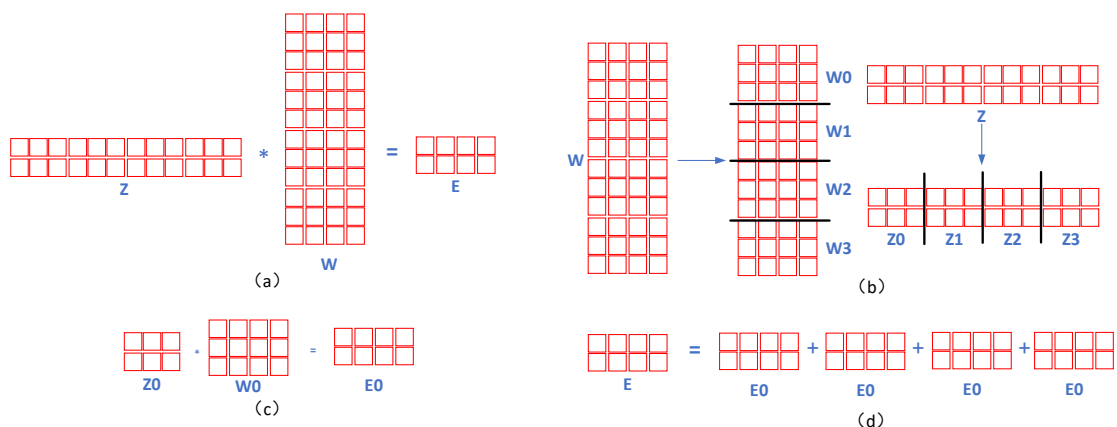


图 23. (a) 未被分解的原始矩阵乘法，数据规模较大；(b) 对输入矩阵的切分；

(c) 分解过后，较小的矩阵乘法；(d) 较小的矩阵乘法结果之和等于正确结果

综上，矩阵乘法是十分适合使用按范围分解的程序。我们将计算切分过后较小的矩阵乘法的任务映射到不同的 GPU 芯粒上，并最终将其结果相加。

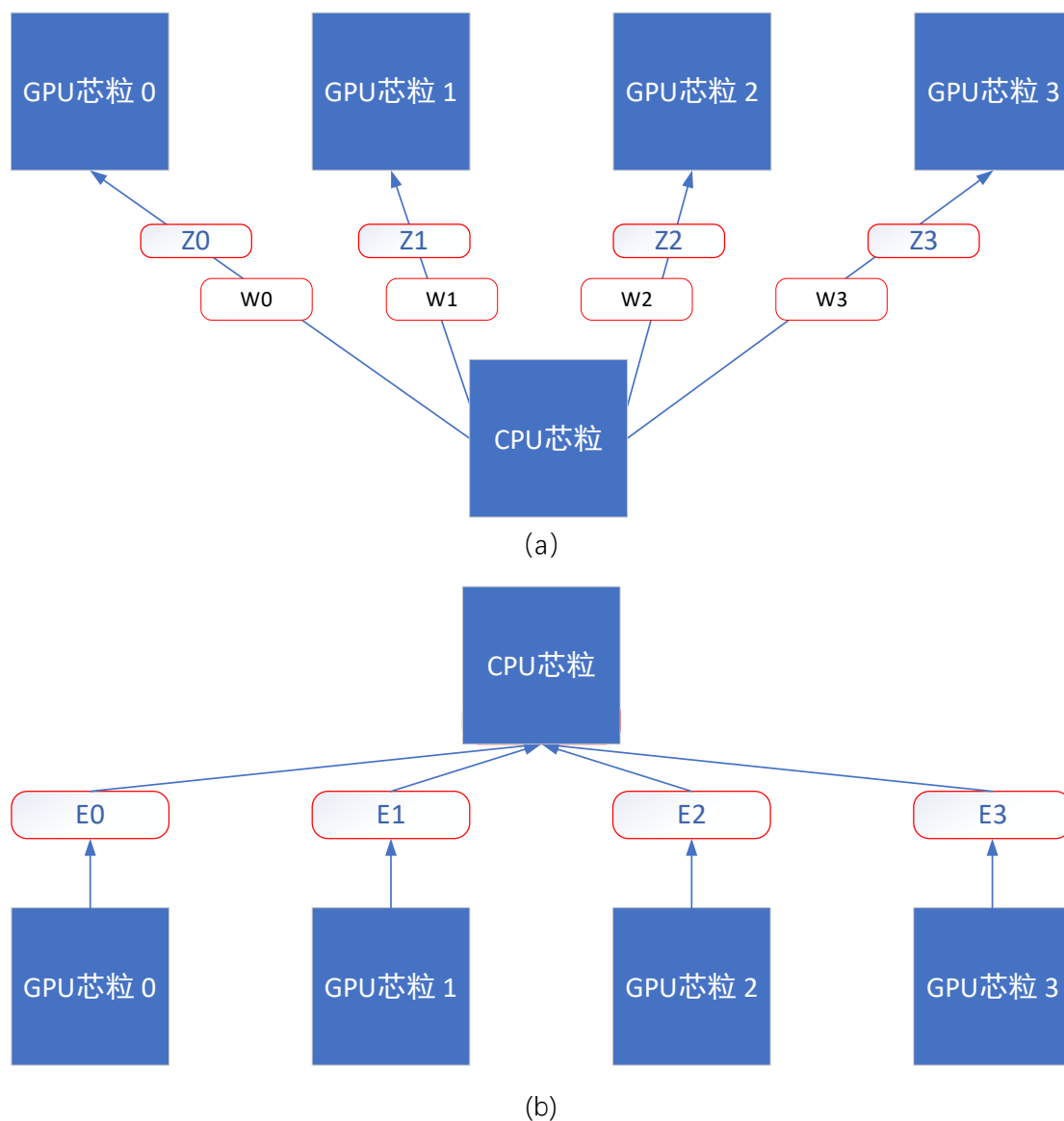


图 24. 矩阵乘法的仿真工作过程。(a) CPU 芯粒向 GPU 芯粒分发数据；  
(b) GPU 芯粒完成计算任务，并将计算结果反馈给 CPU 芯粒

### 3.1.3 编程模型与注意事项

我们将分解给每个芯粒需要完成的任务对应的代码称为一个代码段。一个完整的程序应该包括一个或若干个代码段，这些代码段完成了整个程序中的大部分运算任务。程序的编写模型包括两个部分：一、代码段内部如何编写；二、代码段之间如何组织，即如何将代码段映射到芯粒。

#### 3.1.3.1 代码段的编写

一个代码段应有的流程如图 25 所示。

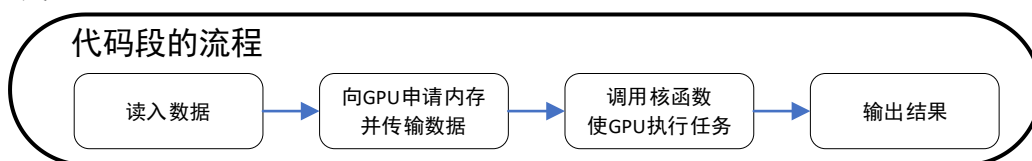


图 25. 代码的主要组织方法

#### (1) 读入数据

读入数据部分的主要任务是准备输入数据，包括在 CPU 内存空间对数据进行

声明、初始化、切分等处理。

对于示例的矩阵乘法程序来说，在读入数据部分我们省略对原矩阵的切分过程，使用随机数直接初始化切分过后的较小的输入矩阵。

```
//声明数据
int *A = (int *)malloc(sizeof(int) * Row * Col);
int *B = (int *)malloc(sizeof(int) * Row * Col);
int *C = (int *)malloc(sizeof(int) * Row * Col);
//初始化数据
for (int i = 0; i < Row*Col; i++) {
    A[i] = rand() % 51;
    B[i] = rand() % 51;
}
```

### (2) 向 GPU 申请内存并传输数据

本部分的代码形式较为固定，主要是通过 `cudaMalloc()` 函数和 `cudaMemcpy()` 函数将数据传递给 GPU 芯粒。

```
//向 GPU 申请内存空间
cudaMalloc((void**)&d_dataA, sizeof(int) *Row*Col);
cudaMalloc((void**)&d_dataB, sizeof(int) *Row*Col);
cudaMalloc((void**)&d_dataC, sizeof(int) *Row*Col);
//向 GPU 传递数据
cudaMemcpy(d_dataA, A, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
cudaMemcpy(d_dataB, B, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
```

### (3) 调用核函数

本部分是程序设计中的核心部分，核函数即 GPU 运行时每个线程的执行内容。核函数的设计主要依靠于程序分析过程中“如何并行化程序使其适合 GPU 运行”部分的分析结果。

根据 3.1.1 小节的分析结果，矩阵乘法的核函数编写如下：

```

/**
 * 矩阵乘法的核心函数，由每个线程都会运行一次本函数，
 * 根据线程编号不同计算出位于结果矩阵不同位置的数据。
 */
__global__ void matrix_mul_gpu(int *M, int* N, int* P, int width)
{
    int sumNum = threadIdx.x + threadIdx.y*10;
    int i = threadIdx.x;
    int j = threadIdx.y;
    int sum = 0;
    for(int k=0;k<width;k++)
    {
        int a = M[j*width+k];
        int b = N[k*width+i];
        sum += a*b;
    }
    P[sumNum] = sum;
}

```

#### (4) 输出结果

数据结果主要包括将结果转移至 CPU 内存和向其他芯粒发送计算结果两种。下面结合示例程序来说明本部分代码应该如何编写。

如果需要向其它芯粒发送 GPU 内存中的数据，应该首先编写一个核函数，调用 2.3.5 小节的芯粒间通讯函数执行数据发送。

```

/**
 * 用于传递单个 chiplet 计算结果的 kernel 函数
 */
__global__ void passMessage(int dstX, int dstY, int srcX,int srcY,int* data, int dataSize){
    int para1 = srcX *10000000 + srcY*100000 + dstX*1000+dstY * 10;
    for(int i = 0; i<dataSize;i++){
        asm("addc.s32 %0, %1, %2;" : "=r"(data[i]) : "r"(para1), "r"(data[i]));
    }
}

```

随后，在代码段的输出结果部分调用该核函数：

```
passMessage << <1,1>>> (0,0,srcX,srcY,d_dataC,100);
```

如果需要将 GPU 内存中的数据拷贝到 CPU 内存中，则调用 `cudaMalloc()` 函数。

```
cudaMemcpy(C, d_dataC, sizeof(int) * Row * Col, cudaMemcpyDeviceToHost);
```



### 3.1.4 代码段的组织与映射

完成对代码段的编写意味着我们已经完成了对单个芯粒要执行的任务的编程。根据在 3.1.2 小节得到的分析结果，我们可以使用以下方法将代码段映射到不同的芯粒上。

对于每个仿真器进程来讲，它都会完整运行我们代码文件中的所有内容。因此我们将芯粒编号设置为启动程序所必须的参数，这样仿真器进程可以通过自己的芯粒编号来判断自己应当执行的代码段。

```
int srcX,srcY;
int main(int argc, char** argv)
{
    //读取本进程所代表的芯粒编号
    srcX=atoi(argv[1]);
    srcY=atoi(argv[2]);
```

在矩阵乘法程序中，每个芯粒都应该执行自己的乘法运算，运算结束后，有一个芯粒负责完成对分结果的相加，得出最终结果。因此，我们需要指定一个芯粒负责接收数据，而其他芯粒向它发送数据。

在本部分需要注意的关键问题是时序上各个芯粒的依赖关系，这两个问题不仅出现在矩阵乘法程序中。

假设芯粒 A 的下一步任务依赖与芯粒 B 发送给它的的数据，那么它必须等待芯粒 B 完成数据传输之后才能进行下一步任务。如果它过早的尝试读取 B 传来的信息，会导致运算出现错误或者程序崩溃。解决办法有多种，例如芯粒 A 的仿真程序不断的读取芯粒 B 传输给它的文件内容，直到识别到一个代表传输已经完毕的信号。

矩阵乘法的示例程序中，因为所有芯粒仿真进程之间只存在一次数据传输过程，所以选取较为简单的处理方法解决依赖问题，需要测试人员确定数据传输完毕后手动控制程序继续运行。

### 3.1.5 数据的映射

经过图 23 的分析可知，不同的芯粒直接虽然执行相同的运算操作，但是操作的数据内容并不相同。根据图 23（b）中设计的数据切分方法，将不同的数据映射到芯粒上的方法主要有两种：将数据预处理为多个文件，不同的仿真进程读取不同的数据文件；仿真进程一次读入全部文件，从中选取映射到本芯粒部分的数据。

第一种数据映射方法的源码如下：

```
char * fileName = new char[100];
//根据芯粒编号来读取不同的数据文件
sprintf(fileName, "./data%d_%d ", srcX, srcY);
std::ifstream file(fileName);
int data[40000];
for(int i = 0; i < dataSize/4; i++)
{
    file >> data[i];
}
```

```

char * fileName = new char[100];

//根据芯粒编号来读取不同的数据文件
sprintf(fileName, "./data");
std::ifstream file(fileName);
int data[40000];
for(int i = 0; i < dataSize; i++)
{
    if(i > 10000 && i <= 20000)
        file >> data[i];
}

```

第二种数据映射方法的源码如下：

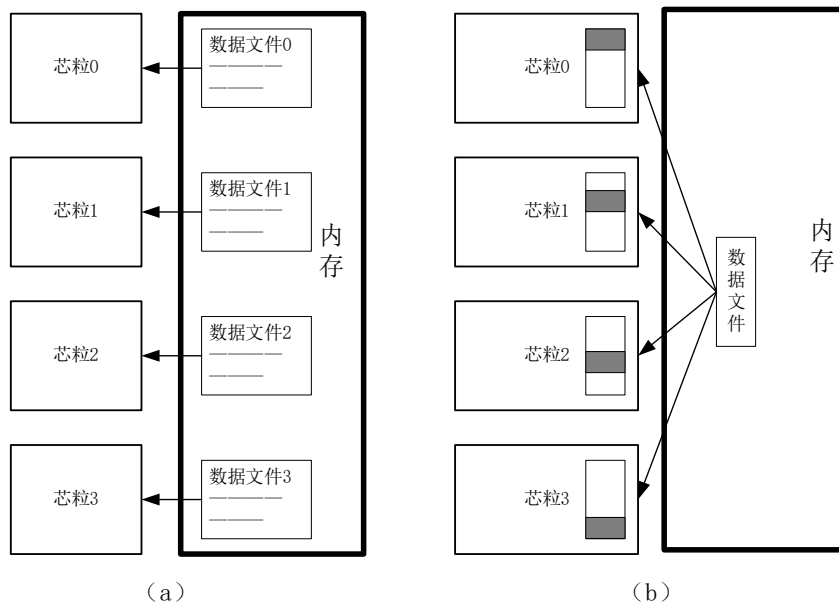


图 24. 两种数据映射方法：(a) 不同芯粒从内存中取得不同数据  
(b) 不同芯粒取得相同数据后提取其中部分

至此，能够在基于多芯粒集成的 GPU 消息传递式仿真器上运行矩阵乘法程序编写完毕。

附录：基于多芯粒集成的 GPU 消息传递式仿真器的矩阵乘法示例程序  
由 gpgpu-sim 运行的程序：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <sys/time.h>
#include <stdio.h>
#include <math.h>
#include <string>
#include <iostream>
#include <fstream>

/**
 * 本示例程序为：通过 4 个 GPU chiplet
 * 计算随机数矩阵 A (400 * 100) 与随机数矩阵 B (100 * 400) 相乘结果。
 * 由矩阵乘法原理可知，我们可将计算任务划分为 4 个 100*100 的矩阵相乘，并将结果相加。
 */

#define Row 100
#define Col 100

/**
 * 矩阵乘法的核心函数，由每个线程都会运行一次本函数，
 * 根据线程编号不同计算出位于结果矩阵不同位置的数据。
 */
__global__ void matrix_mul_gpu(int *M, int* N, int* P, int width)
{
    int sumNum = threadIdx.x + threadIdx.y*10 ;
    int i = threadIdx.x;
    int j = threadIdx.y;
    int sum = 0;
    for(int k=0;k<width;k++)
    {
        int a = M[j*width+k];
        int b = N[k*width+i];
        sum += a*b;
    }
    P[sumNum] = sum;
}

/**
 * 用于传递单个 chiplet 计算结果的 kernel 函数
 */
__global__ void passMessage(int dstX, int dstY, int srcX,int srcY,int* data, int
dataSize){
    int para1 = srcX *10000000 + srcY*100000 + dstX*1000+dstY * 10 ;
    for(int i = 0; i<dataSize;i++){
```

```

        asm("addc.s32 %0, %1, %2;" : "=r"(data[i]) : "r"(para1) , "r"(data[i]));
    }
}

__global__ void readMessage(int dstX, int dstY, int srcX,int srcY,int* data, int dataSize)
{
    int para1 = srcX *10000000 + srcY*100000 + dstX*1000+dstY * 10 + 1 ;

    for(int i = 0; i<dataSize;i++){
        data[i]=i;
        asm("addc.s32 %0, %1, %2;" : "=r"(data[i]) : "r"(para1) , "r"(data[i]));
    }
}

int main(int argc, char** argv)
{
    //读取本进程所代表的 chiplet 编号
    int srcX=atoi(argv[1]);
    int srcY=atoi(argv[2]);
    int *d_dataA, *d_dataB, *d_dataC;
    cudaMalloc((void**)&d_dataA, sizeof(int) *Row*Col);
    cudaMalloc((void**)&d_dataB, sizeof(int) *Row*Col);
    cudaMalloc((void**)&d_dataC, sizeof(int) *Col);

    readMessage <<<1,1>>> (0,0,srcX,srcY,d_dataA,10000);
    readMessage <<<1,1>>> (0,0,srcX,srcY,d_dataB,10000);

    //calculate
    dim3 threadPerBlock(10,10);
    dim3 blockNumber(1);
    matrix_mul_gpu << <blockNumber, threadPerBlock >> > (d_dataA, d_dataB, d_dataC, Col);

    passMessage << <1,1>> > (srcX,srcY,0,0,d_dataC,100);
    cudaFree(d_dataA);
    cudaFree(d_dataB);
    cudaFree(d_dataC);
    return 0;
}

```

由 sniper 运行的程序：

```
#include <fstream>
#include "../interchiptet_app/interchiptet_app.h"
#define Row 100
#define Col 100
int srcX,srcY;
using namespace std;

using namespace nsInterchiptet;

int main(int argc, char** argv)
{
    srcX=atoi(argv[1]);
    srcY=atoi(argv[2]);
    int64_t *A = (int64_t *)malloc(sizeof(int64_t) * Row * Col);
    int64_t *B = (int64_t *)malloc(sizeof(int64_t) * Row * Col);
    int64_t *C1 = (int64_t *)malloc(sizeof(int64_t) * Col);
    int64_t *C2 = (int64_t *)malloc(sizeof(int64_t) * Col);
    int64_t *C3 = (int64_t *)malloc(sizeof(int64_t) * Col);
    for (int i = 0; i < Row*Col; i++) {
        A[i] = rand() % 51;
        B[i] = rand() % 51;
    }
    cout<<"aaa"<<endl;
    sendGpuMessage(0,1,srcX,srcY,A,10000);
    sendGpuMessage(1,0,srcX,srcY,A,10000);
    sendGpuMessage(1,1,srcX,srcY,A,10000);

    readGpuMessage(0,1,srcX,srcY,C1,100);
    readGpuMessage(1,0,srcX,srcY,C2,100);
    readGpuMessage(1,1,srcX,srcY,C3,100);

    for(int i=0;i<100;i++)
    {
        C1[i] += C2[i];
        C1[i] += C3[i];
    }
}
```