

消息传递式 gpu 仿真器

综述：消息传递式 **gpu** 仿真器是基于 **GPGPU-Sim** 仿真器修改的 **GPU** 模拟器，它的主要修改点是在 **GPGPU-Sim** 仿真器功能的基础上加入了 **GPU** 之间进行消息传递的接口函数，与 **popnet** 仿真器协作可以从功能和时间两方面进行多芯粒的仿真。

1. Cuda4.0 的安装

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

1.2 安装 CUDA toolkit

```
superlinc@superlinc: /media/psf/Home/Downloads/gpgpu-sim
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
17_linux_64_ubuntu10.10.run
[sudo] superlinc 的密码:
Verifying archive integrity... All good.
Uncompressing NVIDIA CUDA.....
.....
.....
Enter install path (default /usr/local/cuda, '/cuda' will be appended)
```

指令：

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

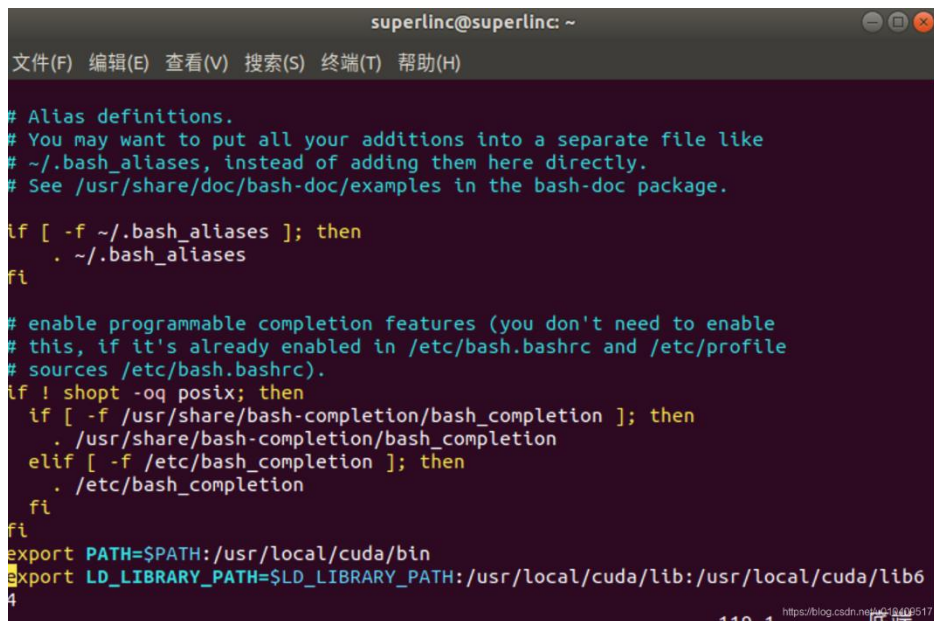
默认安装在/usr/local/cuda。

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。

A terminal window titled 'superlinc@superlinc: ~' showing the content of the ~/.bashrc file. The text is as follows:

```
# 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 1. ~/.bashrc 文件更改后内容

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。

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

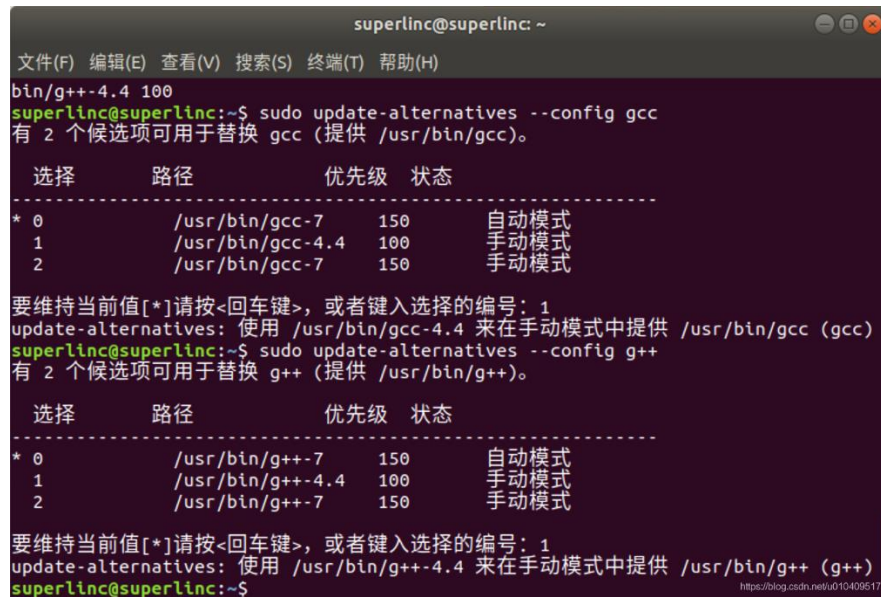
```
sudo apt-get install gcc-4.4 g++-4.4
```

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 2. 修改 gcc 与 g++ 版本

2. 下载安装消息传递式 gpu 仿真器

2.1. 下载源码

GitHub 链接: <https://github.com/FCAS-SCUT/Chiplet-GPGPU-Sim-MessagePassing>

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
```

2.3. 添加 CUDA_INSTALL_PATH 到 ~/.bashrc 中

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

2. 4. 编译消息传递式 gpu 仿真器

```
source setup_environment
make
make docs
```

3. CUDA 编程

3. 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。

3. 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。

3. 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);
```

3.4. kernel 函数

一个简单的 cuda kernel 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 线程的层次结构如下图所示：

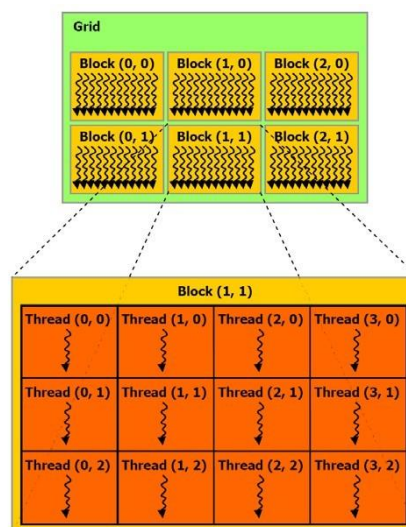


图 3. GPU 线程的层次结构

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 kernel 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*blockID.y)]*Layer1_Weights_GPU[weightBegin+i];
    }

    result=(1.7159*tanhf(0.6666667*result));

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

图 4. kernel 函数示例

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);
```

图 5. kernel 函数的调用示例

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

4. GPGPU-Sim 的简单使用

4.1. 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

4. 2. 使用 GPGPU-Sim 运行 CUDA 程序

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。

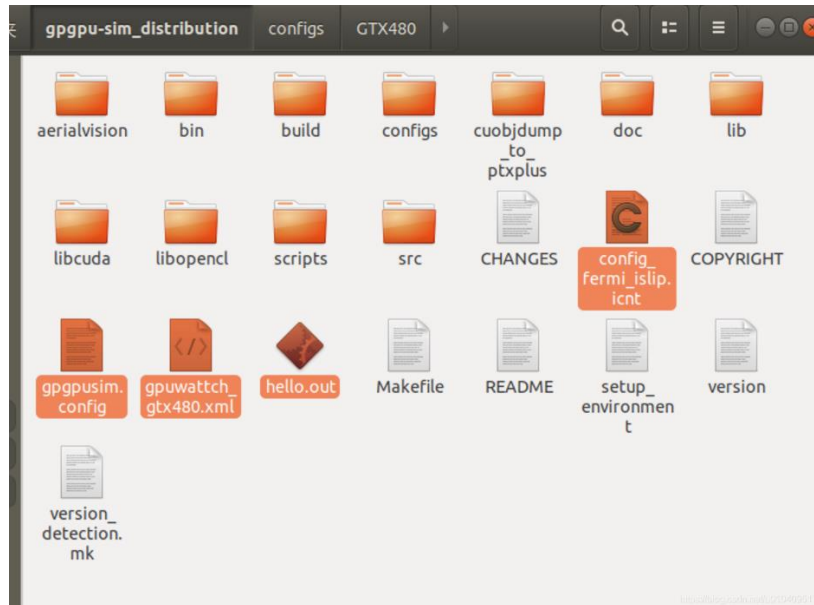


图 6. gugpu-sim_distribution 文件夹下要新增的文件在此路径中运行

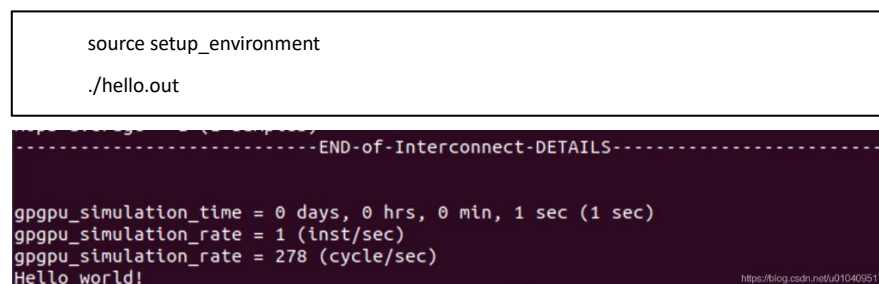


图 7. 仿真程序运行完成图

如顺利运行，则 GPGPU-Sim 安装完毕。

5. 通讯接口函数

5. 1. 函数的参数含义

消息传递式 gpu 仿真器中 Chiplet 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` 发生数据还是读取其他 `chiplet` 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 Chiplet 发送了一个数据，数据内容为 1111。

发送的数据文件名为“buffer_%d1_%d2”，d1 为数据目标 Chiplet 的 x 坐标，d2 为数据目标 Chiplet 的 y 坐标。

5. 2. 函数的使用方法。

5.2.1. 本函数需要放在 kernel 函数中进行使用。

5.2.2. 执行发送功能的函数和执行接收功能的函数应该成对出现。

6. 运行多机消息传递式 gpu 仿真器

6. 1. 多机架构图（以 4 Chiplet 为例）

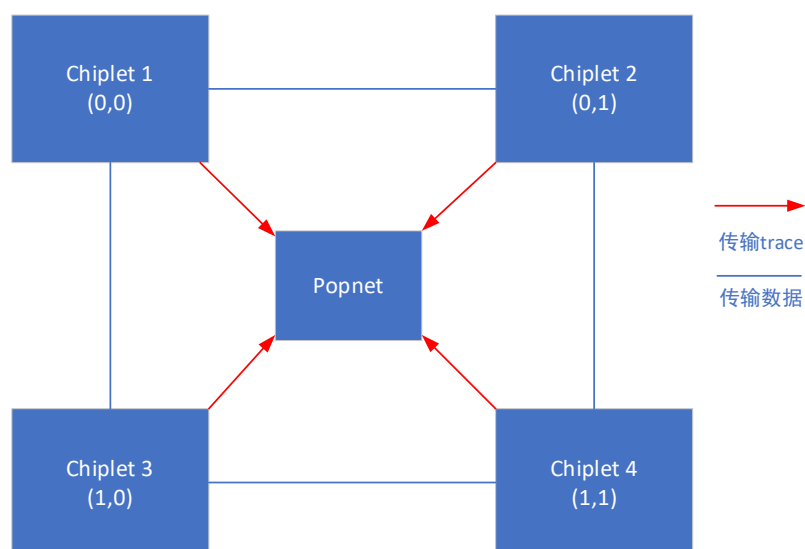


图 8. 多机系统架构示意图

多机模式下，各个 Chiplet 之间主要传递数据，数据传输延迟由 Popnet 根据 trace 记录进行计算。

6. 2. 运行多机程序的主要步骤

6.2.1. 编写多机版本的程序。

6.2.2. 将刚才生成的可执行文件复制到 gugu-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。

6.2.3. 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。

trace 记录的格式是：

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

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

其中各参数含义如下：

-A 9: the size of the network in each dimension. In this example, there are nine routers on each dimension.

-c 2: 2D network

-B 12: input buffer size

-O 12: output buffer size

-F 4: flit size. We assume 64-bit flit unit, the actual flit size is then

$$64 \times 4 = 128$$

-L 1000: link length in um

-T 20000: simulation cycles

-r 1: random seed

-l ./random-trace/bench: trace file

-R 0: choose dimension routing

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，与消息传递式 gpu 仿真器运行完成后所显示的总 cycle 数相加，即为本次仿真的总耗时。

6.3. 以矩阵乘法为例的多机程序运行示例

6.3.1. Step 1: 编写多机版本的程序，本示例程序代码如下：

其文件名为 Matrix.cu，位于目录

~/gpgpu-sim_distribution-master/MatrixM

```

#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]));
    }
}

/**
 * 用于接收其他 chiplet 发送的数据的函数
 */
void readMessage( int srcX,int srcY,int dstX,int dstY,int*data,int dataSize){

```

```

char * fileName = new char[100];
sprintf(fileName, "./buffer%d_%d_%d_%d", srcX, srcY, dstX, dstY);
std::ifstream file(fileName);
int tmpdata = 0;
for(int i = 0; i < dataSize; i++)
{
    file >> tmpdata;
    data[i] += tmpdata;
}
file.close();
}

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

    struct timeval start, end;
    gettimeofday( &start, NULL );

    //malloc local memory
    int *A = (int *)malloc(sizeof(int) * Row * Col);
    int *B = (int *)malloc(sizeof(int) * Row * Col);
    int *C = (int *)malloc(sizeof(int) * Row * Col);
    //malloc device memory
    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) * Row * Col);
    //set value
    for (int i = 0; i < Row * Col; i++) {
        A[i] = rand() % 51;
        B[i] = rand() % 51;
    }

    cudaMemcpy(d_dataA, A, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
    cudaMemcpy(d_dataB, B, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);

    //calculate
    dim3 threadPerBlock(10, 10);

```

```

    dim3 blockNumber(1);
    matrix_mul_gpu << <blockNumber, threadPerBlock >> > (d_dataA, d_dataB, d_dataC,
Col);
    cudaMemcpy(C, d_dataC, sizeof(int) * Row * Col, cudaMemcpyDeviceToHost);

    //如果本 chiplet 不是 (0, 0) 号, 则向 (0, 0) 号发送数据
    if(srcX != 0 || srcY != 0)
    {
        passMessage << <1,1>> > (0,0,srcX,srcY,d_dataC,100);
    }
    else    //如果本 chiplet 是 (0, 0) 号, 则接收其他 chiplet 发送来的数据
    {
        char ready = '0';
        //需要输入任意字符来继续程序, 主要是为了保证依赖,
        //即为了保证其他 chiplet 已经发送完毕数据后, 在开始接收数据
        std::cin >>ready;
        readMessage(srcX,srcY,0,1,C,100);
        readMessage(srcX,srcY,1,0,C,100);
        readMessage(srcX,srcY,1,1,C,100);
    }

    //拷贝计算数据-一级数据指针

    //释放内存
    free(A);
    free(B);
    free(C);
    cudaFree(d_dataA);
    cudaFree(d_dataB);
    cudaFree(d_dataC);

    gettimeofday( &end, NULL );
    int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("total time is %d ms\n", timeuse/1000);

    return 0;
}

```

6.3.2 编译程序

假设终端目前位于目录~/ gpgpu-sim_distribution-master 目录下。

首先，确定 Linux 的 gcc 与 g++ 版本

```
sudo update-alternatives --config gcc
sudo update-alternatives --config g++
```

此处需要确保 gcc 与 g++ 版本为 4.4。

接着，进入 Matrix.cu 所在目录，使用 nvcc 进行编译

```
cd MatrixM
nvcc Matrix.cu -o Matrix
```

编译成功后，Matrix.cu 目录下出现可执行文件 Matrix

6.3.3 准备运行程序

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。如果需要修改仿真配置，则可以在此处修改配置文件。

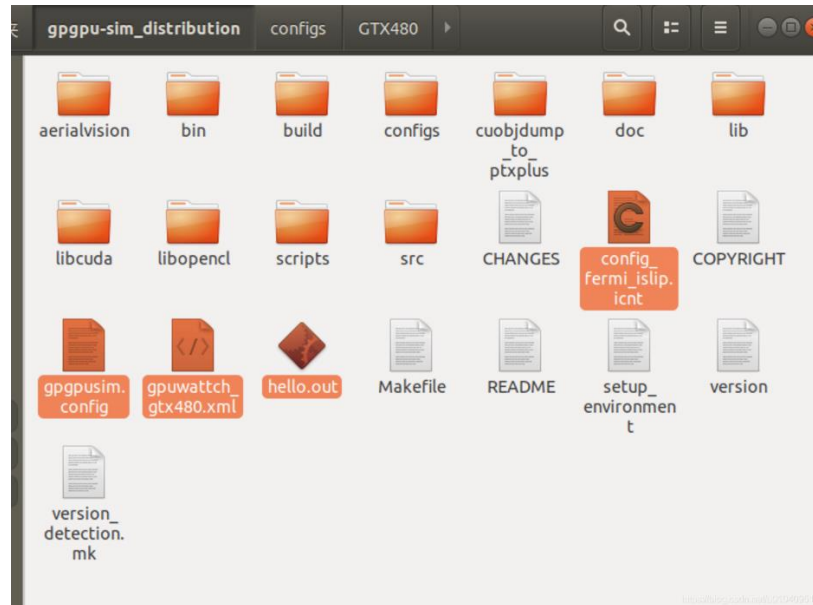


图 9. 示例程序运行前的准备

将可执行文件 Matrix 复制到 ~/ gpgpu-sim_distribution-master 目录。

```
cp Matrix ..
```

6.3.4 运行程序

打开四个终端，并且全部使其处于 ~/ gpgpu-sim_distribution-master 目录下。四个终端依次下方命令以启动 gpgpu-sim

```
source setup_environment
```

```
GPGPU-Sim version 3.2.2 (build ) configured with GPUWattch.
setup_environment succeeded
```

图 10. 环境检查无误，准备仿真

待 4 个终端全部如图 10 所示，则仿真可以进行。

在四个终端中依次启动程序：

终端一：进程对应 chiplet (0, 0)

```
./Matrix 0 0
```

终端二：进程对应 chiplet (0, 1)

```
./Matrix 0 1
```

终端三：进程对应 chiplet (1, 0)

```
./Matrix 1 0
```

终端四：进程对应 chiplet (1, 1)

```
./Matrix 1 1
```

```
-----END-of-Interconnect-DETAILS-----  
  
gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 1 sec (1 sec)  
gpgpu_simulation_rate = 93200 (inst/sec)  
gpgpu_simulation_rate = 32209 (cycle/sec)  
1  
total time is 16030 ms
```

图 11. GPGPU-Sim 运行完毕

待四个进程全部如图 11 所示，则 GPGPU-Sim 运行完毕。因为 chiplet(0, 0) 执行最后的结果汇总，因此我们记录 chiplet (0, 0) 的运行 cycle 数为 T_1 。

此时 \sim /gpgpu-sim_distribution-master 文件夹下出现 trace 目录。

6.3.5 运行 popnet 计算片间通信耗时

假设 popnet 所在目录为 \sim /popnet-master。

将 trace 程序复制到 popnet-master 目录下，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。指令如下：

```
cp trace ~/popnet-master  
cd ~/popnet-master/trace  
touch bench  
cat bench.0.1 >> bench  
cat bench.1.0 >> bench  
cat bench.1.1 >> bench  
sort -n -k 1 bench -o bench
```

运行 popnet, 指令如下：

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

此处 -T 参数可能需要多次调整，直至所有数据包发送运行完毕。

假设所有数据包发送运行完毕时的 cycle 数为 T_2

至此，多机矩阵乘法运行完毕，bench 仿真的总耗时为 T_1+T_2 。