Installation of HASP-enabled GPGPU-Sim

I. Installation and Run

Firstly, use git to clone our code to your computer. (*git clone https://gitee.com/lhy_giytee/gpgpu-sim_hasp.git*) We use Ubuntu 20.04 to perform the evaluation with GPGPU-Sim V4.0.1.

After installation, load the environment variables of CUDA with the following commands. In the example, the CUDA version is 11.1, users can change the CUDA path of their own environment (in **basic_env.sh**).

```
source basic_env.sh
source setup_environment
```

Then the build process is as same as the original GPGPU-Sim project (refer to **README.md**). To run the HASP-enabled demo, please build the workloads and then execute the ELF file:

```
cd workspace/mstream
build
./mstream
```

The case **mstream** is an example case to use the programming interface of multi-stream to simulate the execution of multi-NN. In the current version, we do not use two processes, because GPGPU-Sim is currently wrapped as a function in the libcudart library to interact with applications, so it is difficult to support context management for multiple application processes at the same time.

We have designed the configuration interface related to standalone HASP. This includes a **haspSet** that allocates computational resources to each task (stream), where **haspSet_th1_sh19_mem6** means that stream 1 is bound to 19 shaders and 6 and memory partitions. **haspMalloc** is used to bind correspond **cudaMalloc** function to certain stream and **haspUnset** is used to release the resource of certain stream, enabling dynamic resource allocation.

```
// User-Level Specific Function Interface
__global__ void haspSet_th1_sh19_mem6(){}
__global__ void haspSet_th2_sh11_mem6(){}
__global__ void haspMalloc_th1(){}
__global__ void haspMalloc_th2(){}
__global__ void haspUnset_th1(){}
__global__ void haspUnset_th2(){}
```

The following code shows how these three special interfaces can be applied. These three interface functions are called CUDA kernel functions. The only thing to note is that **haspMalloc** needs to be called before the **cudaMalloc** function is bound.

```
// Create the HASP Env Group 1 & 2
haspSet_th1_sh19_mem6<<<1, 1, 0, stream1>>>();
haspSet_th2_sh11_mem6<<<1, 1, 0, stream2>>>();
// Memory Allocation Before cudaMalloc
haspMalloc_th1<<<1, 1, 0, stream1>>>();
cudaMalloc(&d_a1, N1 * sizeof(int));
// Details Omitted (Kernel Execution)
// ......
cudaMemcpyAsync(h_c, d_c, N1 * sizeof(int), cudaMemcpyDeviceToHost, stream1);
// HASP Env Group 1 Free
haspUnset_th1<<<1, 1, 0, stream1>>>();
cudaMemcpyAsync(h_d, d_d, N2 * sizeof(int), cudaMemcpyDeviceToHost, stream2);
// HASP Env Group 2 Free
haspUnset_th2<<<1, 1, 0, stream2>>>();
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

II. Design Introduction

At the execution level, Figure S1 illustrates the difference between SIMT and MIMD architecture. Here each SIMT core (one shader), can execute 32 threads concurrently. Several SIMT cores are connected to the memory partition through the interconnection network, where the memory partition contains the global L2 cache and DRAM (GDDR) access interfaces.

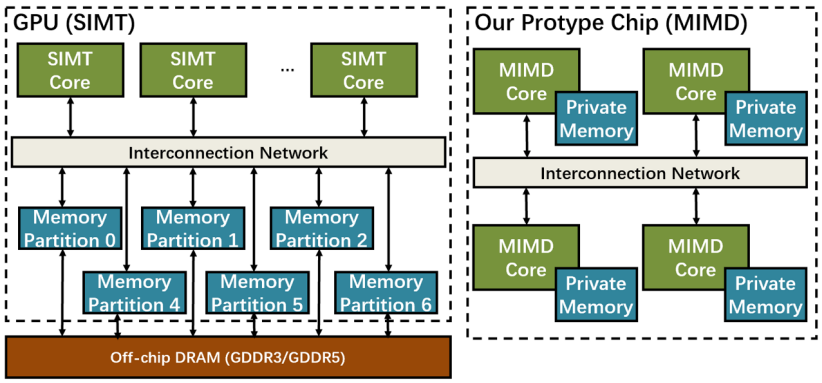

Fig. S 1. Architecture differences between SIMT and MIMD.

The CPU assigns tasks to the GPU in CUDA kernel units. The assigned tasks are stored in a task queue, and the GPU takes the required tasks from the queue for execution according to its own runtime. A proposed solution is to provide a function-specific interface in the CUDA run-time library, where the CUDA programmer explicitly specifies the number of shaders and memory partitions (MP) required for each task, and then calls the corresponding interface before task allocation. At the same time, the trigger module records the occupancy status of each shader and memory partition (shown in Figure S2), so that resources can be allocated in real-time, thus solving the problem of dynamic changes in the target core.

HASP function configuration table

func_name	task_id	shader_num	memory_partition_num	func_addr
haspSet_th1	1	19	4	0x4fc4a0
haspSet_th2	2	11	8	0x4fc4c8

HASP mem-partition runtime table

task_id	addr	size
1	0x100	48
1	0x300	56
.....		
2	0x3300	256
/	/	/

Shader / SM Core Number (N)

cudaMalloc Blocks (M)

HASP shader runtime table

shader_id	task_id
1	1
2	1
.....	
N-1	2
N	2

Fig. S 2. HASP run-time table used for shader/memory allocation in GPU.

In this example, the HASP trigger module is a special hardware module that registers the corresponding configuration information when **haspSet** is detected, and allocates the required shader and memory partition according to the corresponding configuration whenever the CUDA kernel arrives.

The reason we consider this a semi-transparent implementation is the flexibility between letting software specify a certain shader for each function and leaving it entirely to hardware scheduling. With hardware-guaranteed synchronous isolation, this decoupling fully balances the tension between isolation and high performance, and the compiler can get enough information at compile time to make compilation optimizations. As an additional note, the allocation of the corresponding shader and access memory to the CUDA kernel itself is implicitly synchronized, so a perfect isolation strategy is equivalent to unbundling the corresponding synchronization.