Development of an Open ISA GPGPU for Edge Device Machine Learning Applications

Yu-Xiang Su, Jhi-Han Jheng, Dun-Jie Chen, and Chung-Ho Chen
Dept. of Electrical Engineering and Inst. of Computer & Communication Engineering
National Cheng Kung University, Tainan, Taiwan

***Abstract*—Hosting the deep learning model on the cloud may not be the best solution in many cases, for instance, IoT applications or autonomous system where low latency or enhanced security is desirable. Deep learning on the edge alleviates the above issues, and provides benefits of local computation. In this paper, we present the development of an open ISA (instruction set architecture) general purpose GPU aimed at edge computation. Our GPU, CASLab GPU, uses license-free, royalty-free HSAIL ISA specification and supports OpenCL1.2/2.0 APIs for heterogeneous computing. CASLab GPU also supports TensorFlow framework with CUDA-on-CL technology. CASLab GPU IP with configurable SIMT Core design tailors directly to the computing need of on-device learning and inference. The GPU is developed in ESL design methodology which incorporates GPU micro-architecture exploration, power modelling of the GPU, and the co-simulation of the GPU software stack.**

Keywords—Deep learning; Edge computation; ESL design methodology; GPGPU, SIMT operation

I. Introduction

The GPU has evolved from just a graphics chip into a core component of deep learning and machine learning where computation is usually taken in the form of single instruction multiple threading (SIMT). This type of GPU is often referred to as a general purpose GPU, GPGPU, or SIMT GPU. Taking an autonomous computer vision system as an example, the machine learning solution requires a very complete SoC chip with the full-fledged small form-factor embedded system, such as the Xavier SoC [1]. The Xavier chip has included the CPU complex, computing GPUs, deep learning accelerators, along with many other important system IP blocks. The development and integration of such a system, no doubt, requires tremendous man-year efforts.

Our GPU project focuses to develop an SIMT GPGPU core for the edge device applications. The GPU is designed in ESL (electronic system level) methodology that starts with functional GPU instruction set simulator (ISS) together with the development of the OpenCL runtime system. This ESL design methodology includes the micro-architecture exploration of the hardware GPU, the power modelling of the GPU, and the co-simulation of the GPU software stack.

The rest of the paper is organized as follows. In Section II, we introduce the background of SIMT GPU architecture and the GPU software runtime system, including Tensorflow and OpenCL. In Section III, we present our GPGPU micro-architecture and its platform, including the software stack. In Section IV, we show the experimental results and verification of the entire GPU design compared with real-world platforms. Finally, we make a conclusion for this paper in Section V.

II. Backgroud

A. SIMT architecture

Fig. 1 shows the block diagram of a typical GPGPU which composes of Single Instruction Multiple Thread execution cores or Streaming Multiprocessors (SM). Inside the GPU, a workgroup scheduler dispatches a workgroup, which includes a number of programmer specified threads, to an SM. Inside an SM, a warp scheduler decides and dispatches a warp's instruction to all of the lanes of the execution backend pipeline [2]. A number of threads, which is decided by hardware implementation, for instance 32 threads, are grouped into a warp. An SM may consist of a number of warps while each warp has its program counter for instruction fetching.

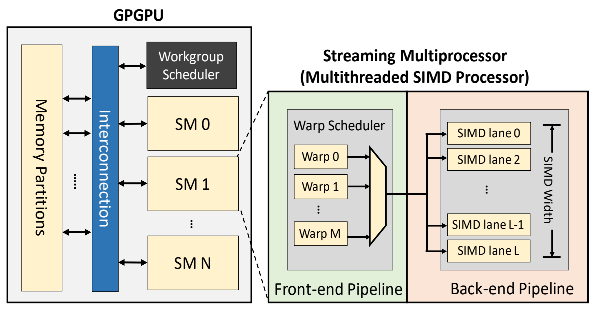

Fig. 1: Block diagram of a computing GPU.

B. GPU software stack: OpenCL runtime

An OpenCL application consists of kernel codes and a host code. The kernel codes implemented in OpenCL language are typically executed by the GPU while the host code typically written in C++ language, is executed by the host CPU. The kernel code is executed in the SIMT execution fashion via the OpenCL runtime APIs which interact with the GPU system.

C. GPU software stack: Tensorflow

Tensorflow is a cross-platform machine learning framework, which provides dataflow-programming style APIs for application developers to build their own machine learning models. The kernels of Tensorflow APIs are implemented by a Google-developed parallel programming model, Stream Executor, which enables runtime developer to implement a variety of software optimization according to different parallel target devices. For instance, the Stream Executor acts as the wrapper of the CUDA host-side programs responsible for memory allocation, memory resource management, and launching of the device-side program on the GPU device. Beside CUDA programming mode, some of the third party projects implement the interface of Stream Executor with OpenCL APIs to support OpenCL compatible devices and enable running Tensorflow applications on OpenCL devices.

III. CASLAB GPU MICRO-ARCHITECTURE/SOFTWARE STACK

Fig. 2 shows the internal micro-architecture of an SIMT core or SM of the CASLab GPU. In ESL methodology, we model the SM with pin cycle-accurate design in SystemC modules. The GPU front-end pipeline manages multiple instruction streams of fetching, decoding, etc., each of which is scheduled and executed as a warp independently. The warp scheduler in the front-end pipeline will only issue the warps, which have no instruction data dependency to the back-end pipeline. We have explored various warp scheduling policies for high instruction throughput [2].

A coalescer in the LSU unit helps the load-store unit to reduce memory traffic by merging the memory requests from threads into a single request, if possible, to the data cache. To evaluate the GPU power consumption, we implement the power model for each of the functional units in Fig. 2 so that power-hungry components or mechanisms can be examined and improved. In the following, we also briefly describe the runtime software stack, which is shown in Fig. 3, for our GPGPU system.

A. *Tensorflow enabling technology: TF-coriander*

To enable support of Tensorflow-based machine learning applications, we integrate the open-source TF-coriander system [4]. TF-coriander has used the OpenCL runtime APIs to implement the Stream Executor. Furthermore, TF-coriander makes use of CLBlast [5] library, a Basic Linear Algebra library written in OpenCL language, to implement the kernels of Tensorflow APIs for linear algebra operations. For the rest of APIs, TF-coriander has used a compiler, called coriander, to compile the OpenCL kernel code from the CUDA code.

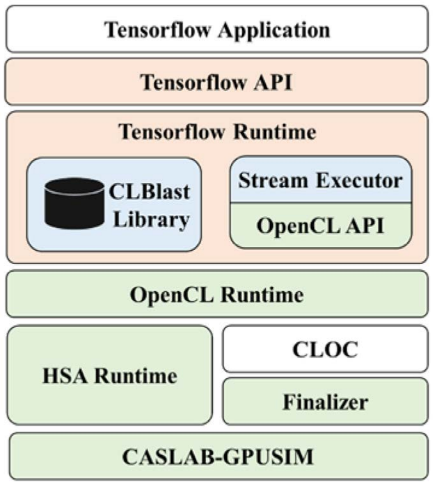

Fig. 3: Software stack for CASLab GPU

B. *OpenCL kernel compiling: CLOC*

Our GPU uses the open HSAIL (Heterogeneous System Architecture Intermediate Language) specification for GPU ISA [3]. We use OpenCL Offline Compiler (CLOC) to compile the OpenCL kernel codes to HSAIL. The compilation process is launched by the related API of OpenCL Stream Executor. Then, the output of the HSAIL intermediate representation (IR) is translated to GPU binary via a HSAIL finalizer.

C. *HSA Runtime*

The HSA runtime is responsible for the memory management of the GPU. Communication between CPU and GPU is constructed in HSA runtime via AQL packets. Specifically, we use the HSA runtime as the core of the OpenCL runtime implementation. To sum up, the kernels of Tensorflow API are implemented by

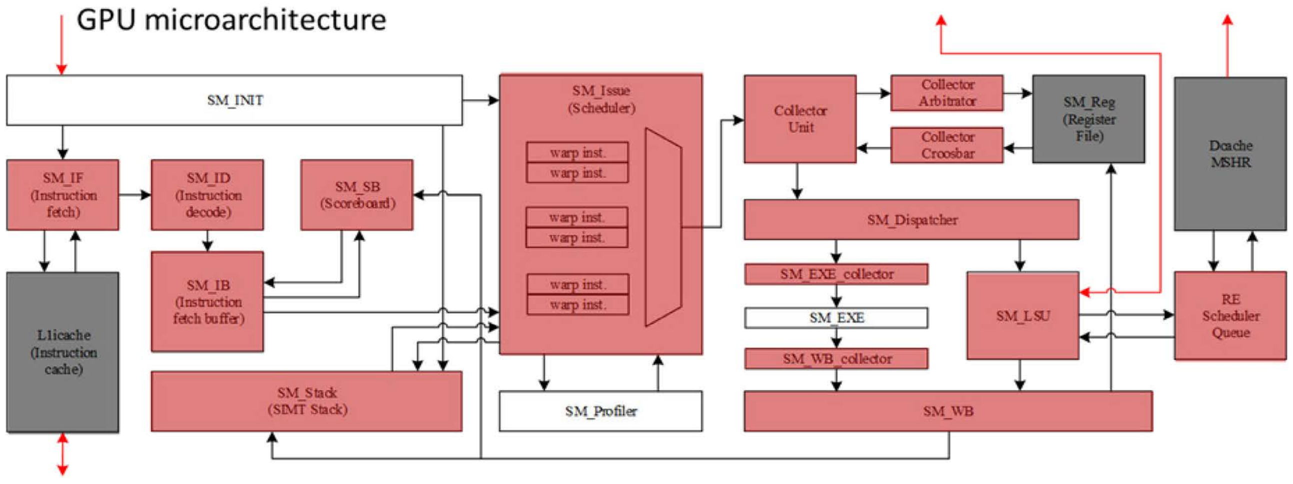

Fig. 2: The micro-architecture of an SM in CASLab GPU

TABLE I. GPU HARDWARE CONFIGURATION

Configuration of the CASLab GPU			
Frequency	200 MHz-1200MHz	Register File Size	1024 KB
# SM	4	# Register File Bank	8
Max # Warps / SM	48	D-Cache / SM	16KB
Max# Workgroups / SM	8	I-Cache / SM	32KB
SIMD Width	32	Shared Memory / SIMT-Core	32768 KB
# Stack Enrty	128	L2-Cache (shared)	256KB
# ALU Collector Entry	6	Memory Controller	FCFS
# LSU Collector Entry	4	DRAM (Latency)	DDR3/DDR4
ALU Consumed Cycle	1 cycle	NoC (interconnection)	Crossbar

several OpenCL kernel codes and then compiled and translated by CLOC compiler and finalizer, finally dispatched to the GPU through the HSA runtime.

IV. EXPERIMENTAL EVALUATIONS

Our GPU currently is implemented in SystemC with major modules in RTL for power modelling. Table I shows the GPU configuration for the experiment. We write a test bench in python that implements the MNIST training model in one fully-connected layer configuration using Tensorflow APIs. For simulation time purpose, the training model runs 10 training loops each with a batch size 100 for a total of 1000 28x28 images. The python program is run on three different platforms for performance comparisons. The output results from the two real-world systems can be used to verify the design of the entire CASLab GPU system, including the GPU hardware model, the OpenCL runtime, the support for the Tensorflow framework, etc.

TABLE II. PERFORMANCE BENCHMARKING PLATFORM

CASLab GPU	Intel Core i7-6700 (PC)	Jetson TX2
CASLab	Skylake	Pascal
4/8 SIMT cores	4 cores	8 SIMT cores
32 T/SIMT core	8	256 CUDA cores
200MHz –1.2 GHz	3.4 GHz	1.3 GHz
Host PC	--	ARM Cortex-A57
Tensorflow 0.11	Tensorflow 0.11	Tensorflow 1.3
OpenCL 1.2	Ubunt 16.04	Ubuntu 16.04
HSA1.0	C++Eigen lib	CuBlas/Cudnn

We evaluate the performance of the CASLab GPU and compare with the PC system of Corei7 processor and TX2 GPU for the same Tensorflow applications. Table II shows the details of the benchmarking platform. Note that the CASLab GPU is implemented in SystemC so it can be set to the desirable clock rate or configuration.

A. *Performance result*

Fig. 4 shows the performance result of running the same Tensorflow application on the three different platforms. We observe that CASLab GPU with DDR3 memory model either 4 SMs or 8 SMs has performed much worse than the version of DDR4 memory model. Yet with no optimization so far on the CASLab GPU system, we see its performance is close to the counterpart of the market veteran GPU, TX2 system. Future optimization can be done in various aspects, such as the memory controller, the CLBlast lib, the micro-architecture of the GPU.

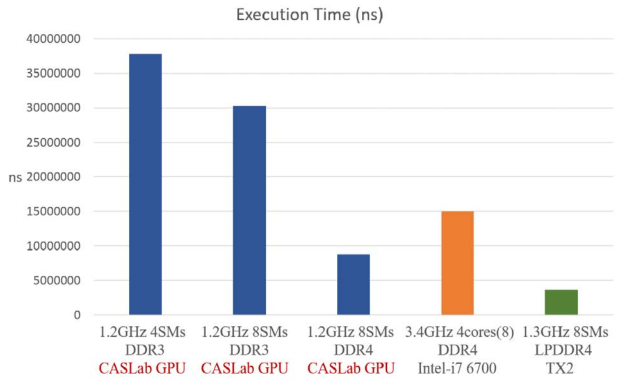

Fig. 4: Execution time of running the same MNIST training model

B. *Power modelling*

Our GPU is designed in ESL methodology which allows us to evaluate GPU micro-architecture and their power consumption at the same time. We have implemented major functional blocks of the GPU in RTL and get their power information from post-synthesis result. Fig. 5 shows the power breakdown for a 200MHz setting. Among the usage, we note that the stack mechanism used to support branch divergence operation has topped the list. This finding allows us to review and design the new micro-architecture support of the mechanism.

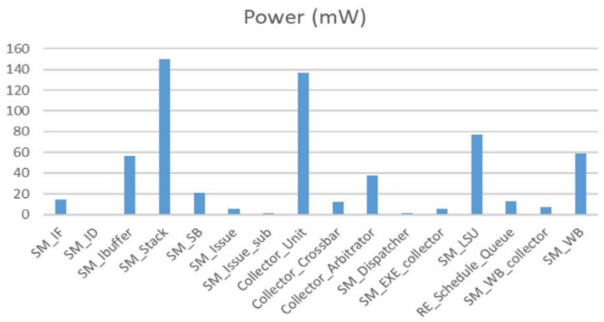

Fig. 5: Power consumption breakdown of a 200MHz clock rate GPU (40 nm process)

V. CONCLUSION

We have presented an open ISA general purpose GPU aimed at edge computation. Our GPU is developed in ESL methodology that allows us to explore the GPU micro-architecture, model power consumption of the GPU, and verify the design with the full-system of GPU software stack. The GPU supports TensorFlow framework with CUDA-on-CL technology. We show the preliminary performance of running the Tensorflow application and compare the results with real-world machine running the same python code. This approach enables us to optimize the GPU design for effective power and performance tradeoffs.

ACKNOWLEDGMENT

This work is supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 107-2634-F-006-002.

REFERENCE

[1] https://developer.nvidia.com/embedded/develop/hardware

[2] Chien-Ming Chiu, "GPU Warp Scheduling Using Memory Stall Sampling on CASLAB-GPUSIM," Master thesis, Institute of Computer and Communication Enginneering, National Cheng-Kung University, July, 2017.

[3] HSA Foundation, HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG) Version 1.0 Final, HSA Foundation, 2015.

[4] Perkins, Hugh. "CUDA-on-CL: a compiler and runtime for running NVIDIA® CUDA™ C++ 11 applications on OpenCL™ 1.2 Devices," Proceedings of the 5th International Workshop on OpenCL, 2017.

[5] Nugteren, Cedric. "Clblast: A tuned opencl BLAS library," arXiv preprint arXiv:1705.05249, 2017.

[6] Yang, Chun-Chieh, et al. "The support of an experimental OpenCL compiler on HSA environments," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2015.