SANDIA REPORT

SAND2019-1967
Unlimited Release
Printed February 22, 2019

SST-GPU: An Execution-Driven CUDA Kernel Scheduler and Streaming-Multiprocessor Compute Model

M. Khairy, M. Zhang, R. Green, and T. Rogers
Accelerator Architecture Lab
Purdue University
West Lafayette, IN 47907
khairy2011@gmail.com, zhan2308@purdue.edu, rgreen.dev@gmail.com, timrogers@purdue.edu

S.D. Hammond, R.J. Hoekstra, and C. Hughes
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM 87185
{sdhammo, rjhoeks, chughes}@sandia.gov

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2019-1967
Unlimited Release
Printed February 22, 2019

SST-GPU: An Execution-Driven CUDA Kernel Scheduler and Streaming-Multiprocessor Compute Model

M. Khairy[1], M. Zhang[1], R. Green[1],
S. Hammond[2], R.J. Hoekstra[2], T. Rogers[1], and C. Hughes[2]
[1]AALP Research Group, Purdue University, West Lafayette, IN 47907
[2]Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185

Abstract

Programmable accelerators have become commonplace in modern computing systems. Advances in programming models and the availability of massive amounts of data have created a space for massively parallel acceleration where the context for thousands of concurrent threads are resident on-chip. These threads are grouped and interleaved on a cycle-by-cycle basis among several massively parallel computing cores. The design of future supercomputers relies on an ability to model the performance of these massively parallel cores at scale.

To address the need for a scalable, decentralized GPU model that can model large GPUs, chiplet-based GPUs and multi-node GPUs, this report details the first steps in integrating the open-source, execution driven GPGPU-Sim into the SST framework. The first stage of this project, creates two elements: a kernel scheduler SST element accepts work from SST CPU models and schedules it to an SM-collection element that performs cycle-by-cycle timing using SSTs MemHierarchy to model a flexible memory system.

Acknowledgment

We would like to thank Gwen Voskuilen for her help with MemHierarchy and recommendations on debugging problems with the NIC and interconnect. We would also like to thank Arun Rodrigues and Scott Hemmert for their support and help in defining the scope of the project.

Contents

1 Introduction **7**

2 Scheduler Component **9**

3 Streaming-Multiprocessor Component **13**

4 Conclusion **17**

References **18**

List of Figures

1.1 High-level CPU/GPU interaction model 7

2.1 SST Element architecture for kernel/CTA scheduler and SMs components 11

2.2 Centralized GPU Scheduler component 11

3.1 SST Link and IPCTunnels for functional model support 13

3.2 Timing and memory model for SMs component 14

Chapter 1

Introduction

With the rise of General-Purpose Graphics Processing Unit (GPGPU) computing and compute-heavy workloads like machine-learning, compute accelerators have become a necessary component in both high-performance supercomputers and datacenter-scale systems. The first exascale machines are expected to heavily leverage the massively parallel compute capabilities of GPUs or other highly parallel accelerators [4]. As the software stack and programming model of GPUs and their peer accelerators continue to improve, there is every indication that this trend will continue. As a result, architects that wish to study the design of large-scale systems will need to evaluate the effect their techniques have using a GPU model. However, the focus of all publicly available cycle-level simulators like GPGPU-Sim [2] is on single-node performance. In order to truly study the problem at scale, a parallelizable, multi-node GPU simulator is necessary.

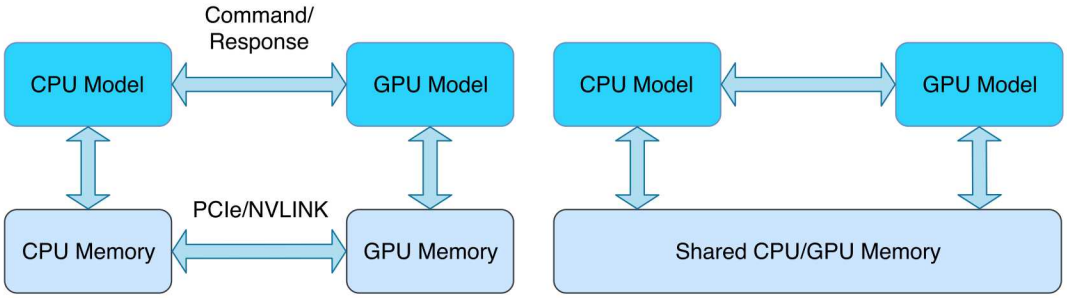

Figure 1.1: High-level CPU/GPU interaction model

Figure 1.1 depicts the current CPU/GPU model co-processor model. On the left is the common high-performance, discrete GPU configuration, where the CPU and GPU have separate memory spaces and are connected via either PCIe or a high-bandwidth link, like NVLink. The right shows the APU model where the CPU and GPU share the same memory. Note that in even in the discrete memory case, modern memory translation units allow the CPU and GPU to share the same address space, although the memories themselves are discrete.

In this report we will detail a model that is capable of simulating both discrete and unified memory spaces by leveraging the MemHeirarchy interface in SST [5]. This report details our efforts to integrate the functional and streaming multiprocessor core models from the open-source simulator GPGPU-Sim into SST.

This page intentionally left blank.

Chapter 2

Scheduler Component

The first step in integrating GPGPU-Sim into SST is to handle the interaction with an SST CPU component. Since GPUs today function solely as co-processors, functionally executing GPU-enabled binaries requires the CPU to initialize and launch kernels of work to the GPU. In our model, the GPU is constructed out of two discrete SST components – a scheduler and a SM block [1]. When CUDA functions are called from the CPU component, they are intercepted and translated into messages that are sent over SST links to the GPU (along with the associated parameters). Table 2.1 enumerates the CUDA API calls currently intercepted and sent to the GPU elements. These calls are enough to enable the execution of a number of CUDA SDK kernels, DoE proxy apps as well as a collection of Kokkos Unit tests. Table 2.2 lists the number of Kokkos unit tests that pass with our current implementation of SST-GPU, which is about 60%. There is ongoing work with the PTX parser to increase the number of running kernels.

Table 2.1: Intercepted CUDA API Calls Forwarded to GPU Model

`__cudaRegisterFatBinary`
`__cudaRegisterFunction`
`cudaMalloc`
`cudaMemcpy`
`cudaConfigureCall`
`cudaSetupArgument`
`cudaFree`
`cudaLaunch`
`cudaGetLastError`
`__cudaRegisterVar`
`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`

Aside from the basic functional model provided by GPU-SST, an initial performance model has also been developed. Figure 2.1 details the overall architecture. A CPU component (Ariel in the initial implementation) is connected via SST links to 2 GPU components: the SMs, which implement the timing and functional model for the GPU cores, and a centralized kernel and CTA scheduler (GPUSched). When CUDA calls are intercepted from the CPU, messages are sent to both the SMs and the GPU scheduler. Messages related to memory copies and other information necessary to populate the GPU functional model are sent directly to the SMs element, since the functional model for executing the GPU kernels lives inside the SMs element. Calls related to enqueuing kernels for execution are sent to the GPU scheduler element, which co-ordinates the

Table 2.2: Functionally Passing Kokkos Unit Tests

Kernel Name	GPGPU-Sim	GPGPU-Sim/SST
abs_double	OK	OK
abs_mv_double	OK	OK
asum_double	OK	OK
axpby_double	OK	OK
axpby_mv_double	OK	OK
axpy_double	OK	OK
axpy_mv_double	OK	OK
dot_double	OK	OK
dot_mv_double	OK	OK
mult_double	OK	OK
mult_mv_double	OK	OK
nrm1_double	OK	OK
nrm1_mv_double	OK	OK
nrm2_double	OK	OK
nrm2_mv_double	OK	OK
nrm2_squared_double	OK	OK
nrm2_squared_mv_double	OK	OK
nrminf_double	FAILED	PREVIOUS FAILED
nrminf_mv_double	FAILED	PREVIOUS FAILED
reciprocal_double	FAILED	PREVIOUS FAILED
reciprocal_mv_double	FAILED	PREVIOUS FAILED
scal_double	OK	OK
scal_mv_double	OK	OK
sum_double	OK	OK
sum_mv_double	OK	OK
update_double	OK	OK
update_mv_double	OK	OK
gemv_double	FAILED	PREVIOUS FAILED
gemm_double	FAILED	PREVIOUS FAILED
sparse_spgemm_double_int_int_TestExecSpace	FAILED	PREVIOUS FAILED
sparse_spadd_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_gauss_seidel_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_block_gauss_seidel_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_crsmatrix_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_blkcrsmatrix_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_replaceSumIntoLonger_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_replaceSumInto_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_graph_color_double_int_int_TestExecSpace	NOT PARSED	PREVIOUS FAILED
sparse_graph_color_d2_double_int_int_TestExecSpace	FAILED	PREVIOUS FAILED
common_ArithTraits	NOT PARSED	PREVIOUS FAILED
common_set_bit_count	FAILED	PREVIOUS FAILED
common_ffs	OK	OK
batched_scalar_serial_set_double_double	OK	FAILED
batched_scalar_serial_scale_double_double	OK	OK
batched_scalar_serial_gemm_nt_nt_double_double	OK	OK
batched_scalar_serial_gemm_t_nt_double_double	OK	OK
batched_scalar_serial_gemm_nt_t_double_double	OK	OK
batched_scalar_serial_gemm_t_t_double_double	OK	OK
batched_scalar_serial_trsm_l_l_nt_u_double_double	OK	OK
batched_scalar_serial_trsm_l_l_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_serial_trsm_l_u_nt_u_double_double	OK	OK
batched_scalar_serial_trsm_l_u_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_serial_trsm_r_u_nt_u_double_double	OK	OK
batched_scalar_serial_trsm_r_u_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_serial_lu_double	OK	FAILED
batched_scalar_serial_gemv_nt_double_double	OK	OK
batched_scalar_serial_gemv_t_double_double	OK	OK
batched_scalar_serial_trsv_l_nt_u_double_double	OK	FAILED
batched_scalar_serial_trsv_l_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_serial_trsv_u_nt_u_double_double	OK	FAILED
batched_scalar_serial_trsv_u_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_team_set_double_double	OK	FAILED
batched_scalar_team_scale_double_double	OK	OK
batched_scalar_team_gemm_nt_nt_double_double	OK	OK
batched_scalar_team_gemm_t_nt_double_double	OK	OK
batched_scalar_team_gemm_nt_t_double_double	OK	OK
batched_scalar_team_gemm_t_t_double_double	OK	OK
batched_scalar_team_trsm_l_l_nt_u_double_double	OK	OK
batched_scalar_team_trsm_l_l_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_team_trsm_l_u_nt_u_double_double	OK	OK
batched_scalar_team_trsm_l_u_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_team_trsm_r_u_nt_u_double_double	OK	OK
batched_scalar_team_trsm_r_u_nt_n_double_double	FAILED	PREVIOUS FAILED
batched_scalar_team_lu_double	OK	FAILED
batched_scalar_team_gemv_nt_double_double	OK	OK
batched_scalar_team_gemv_t_double_double	OK	OK

launching of CTAs on the SMs, e.g. `cudaConfigureCall` and `cudaLaunch`.

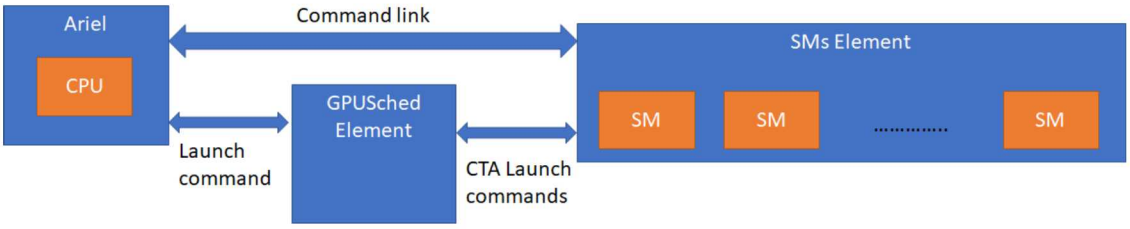

Figure 2.1: SST Element architecture for kernel/CTA scheduler and SMs components

As CTAs complete on the SMs, messages are sent back to the GPU scheduler element, which pushes new work to the SMs from enqueued kernels as needed. Memory copies from the CPU to GPU address space are handled on a configurable page-size granularity, similar to how conventional CUDA unified memory handles the transfer of data from CPU to GPU memories.

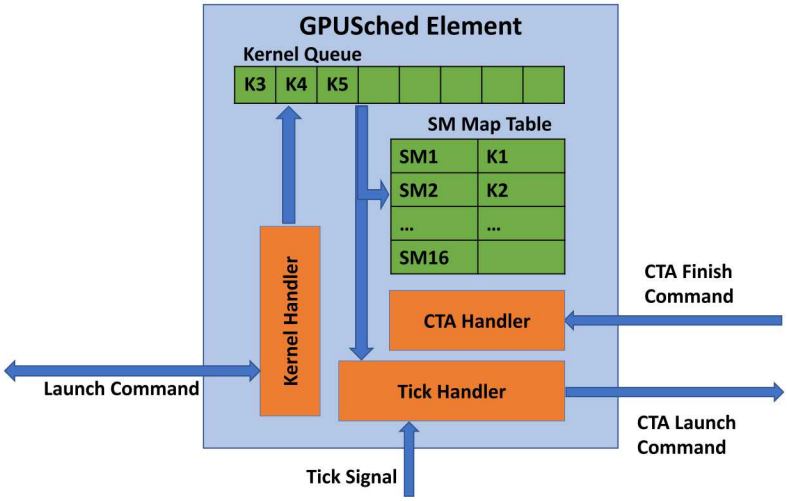

Figure 2.2: Centralized GPU Scheduler component

The centralized GPU scheduler receives kernel launch commands from the CPU, then issues CTA launch commands to the SMs. The scheduler also receives notifications from the SMs when the CTAs finish. The reception of kernel launch and CTA complete notifications are independent, therefore we designed a different handler for each type of message. Figure 2.2 shows the design of the centralized kernel and CTA Scheduler. The kernel handler listens to calls from a CPU component and pushes kernel launch information to the kernel queue when it receives kernel configure and launch commands. The SM map table contains CTA slots for each of the SMs, which is reserved when launching a CTA and released when a message indicating that a CTA has finished is received from the SMs. The scheduler clock ticks trigger CTA launches to SMs, when space is available and there is a pending kernel. On every tick, the scheduler issues a CTA launch command

for currently unfinished kernels if any CTA slot is available or tries to fetch a new kernel launch from kernel queue. The CTA handler also waits for SMs to reply the CTA finish message, so that CTA slots in the SM map table may be freed.

Chapter 3

Streaming-Multiprocessor Component

To support the GPGPU-Sim functional model, a number of the simulator's overloaded CUDA Runtime API calls were updated. A number of functions that originally assumed the application and simulator were within same address space now support them being decoupled. Initialization functions, such as `__cudaRegisterFatBinary`, now take paths to the original application to obtain the PTX assembly of CUDA kernels.

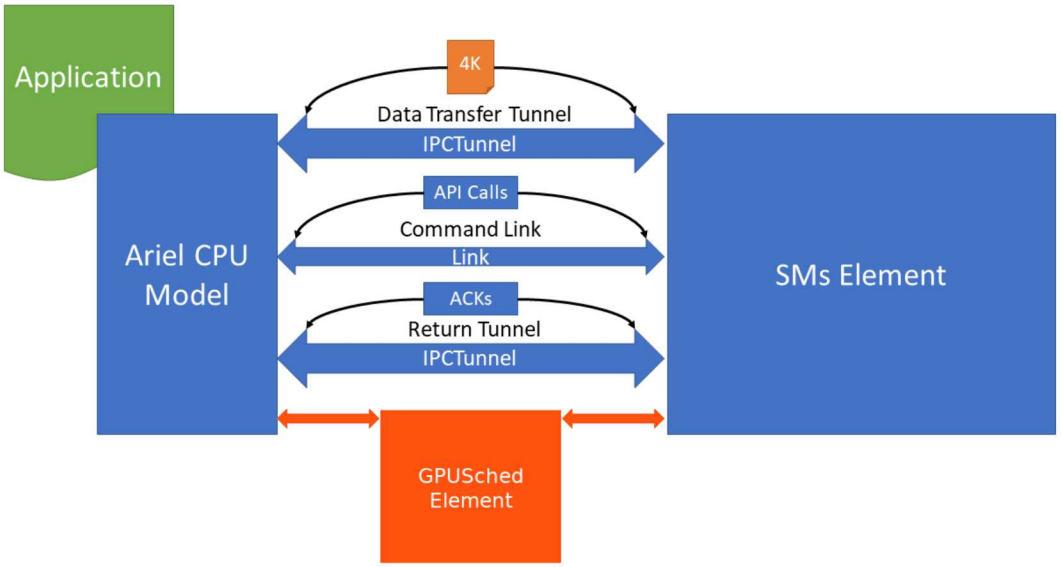

Figure 3.1: SST Link and IPCTunnels for functional model support

Supporting the functional model of GPGPU-Sim also requires transferring values from the CPU application to the GPU memory system. This is solved by leveraging the inter-process communication tunnel framework from SST-Core, as shown in 3.1. Chunks of memory are transferred from the CPU application to the GPU memory system at the granularity of a page (4KiB). The transfer of pages is a blocking operation, therefore all stores to the GPU memory system must be completed before another page is transferred or another API call is processed.

To model GPU performance, the memory system of the public GPGPU-Sim is completely removed. Instead, all accesses to GPU memory are sent though SST links to the MemHierarchy interface. As Figure 3.2 shows, a multi-level cache hierarchy is simulated with the shared L2

sliced between different memory partitions, each with its own memory controller. Several backend timing models have been configured and tested, including SimpleMem, SimpleDRAM, Timing-DRAM, and CramSim [3]; CramSim will be used to model the HBM stacks in the more detailed performance models. We have created an initial model for the GPU system similar to that found in an Nvidia Volta. The configuration for the GPU, CramSim and Network components is shown in Listing 3.1.

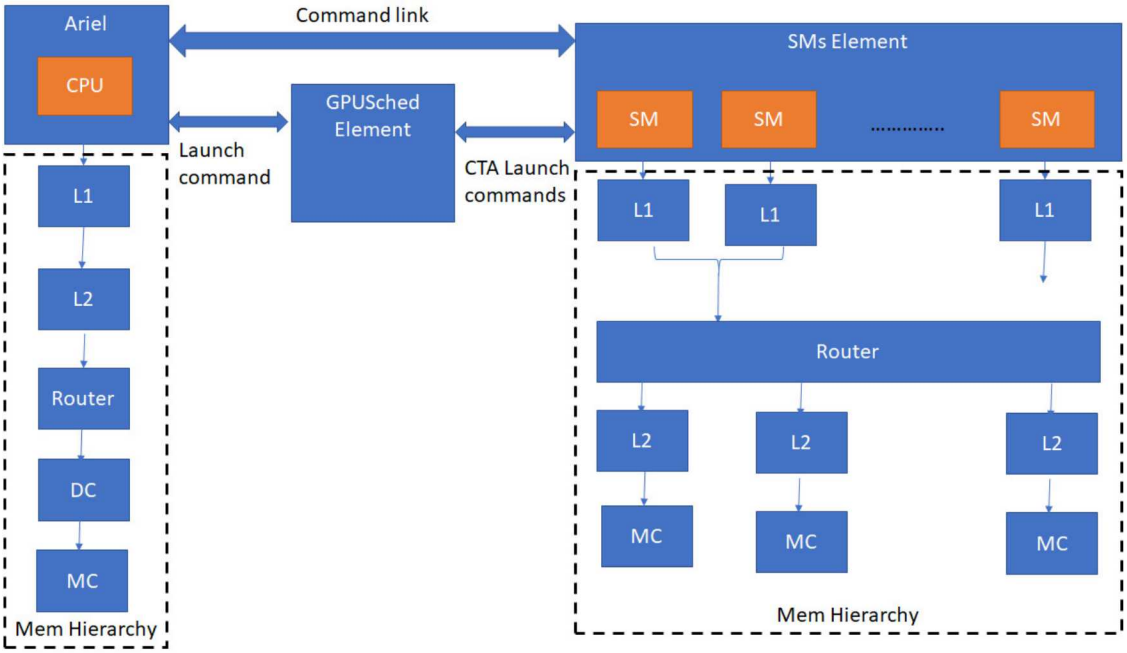

Figure 3.2: Timing and memory model for SMs component

Listing 3.1: Sample SST-GPGPU Configuration

```
[CPU]
clock: 2660MHz
num_cores: 1
application: ariel
max_reqs_cycle: 3

[ariel]
executable: ./vectorAdd
gpu_enabled: 1

[Memory]
clock: 200MHz
network_bw: 96GB/s
capacity: 16384MiB

[Network]
latency: 300ps
bandwidth: 96GB/s
flit_size: 8B

[GPU]
clock: 1200MHz
gpu_cores: 80
gpu_l2_parts: 32
gpu_l2_capacity: 192KiB
gpu_cpu_latency: 23840ps
gpu_cpu_bandwidth: 16GB/s

[GPUMemory]
clock: 1GHz
network_bw: 32GB/s
capacity: 16384MiB
memControllers: 2
hbmStacks: 4
hbmChan: 4
hbmRows: 16384

[GPUNetwork]
latency: 750ps
bandwidth: 4800GB/s
linkbandwidth: 37.5GB/s
flit_size: 40B
```

This page intentionally left blank.

Chapter 4

Conclusion

This report has detailed the first phase of the SST-GPU project, where the execution-driven functional and performance model of a GPU had been integrated SST. Initial results demonstrate significant coverage of applications. The next phase of the project will focus on further disaggregating the GPU to enable truly scaled GPU performance in a multi-process MPI simulation.

This page intentionally left blank.

References

[1] Volta v100 white paper. Technical report, Nvidia, 2017.

[2] Tor M. Aamodt, Wilson W. L. Fung, Inderpreet Singh, Ahmed El-Shafiey, Jimmy Kwa, Tayler Hetherington, Ayub Gubran, Andrew Boktor, Tim Rogers, Ali Bakhoda, and Hadi Jooybar. Gpgpu-sim 3.x manual. http://gpgpu- sim.org/manual/index.php/Main, June 2016.

[3] Michael B. Healy and Seokin Hong. Cramsim: Controller and memory simulator. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, pages 83–85, New York, NY, USA, 2017. ACM.

[4] Timothy Prickett Morgan. The roadmap ahead for exascale hpc in the us. https://www.nextplatform.com/2018/03/06/roadmap-ahead-exascale-hpc-us, March 2018.

[5] Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. The structural simulation toolkit: A tool for bridging the architectural/microarchitectural evaluation gap. Internal Report SAND2004-6238C, 2004.

[6] Christian Trott, Mark Hoemmen, Mehmet Deveci, and Kyungjoo Kim. Kokkos c++ performance portability programming ecosystem: Math kernels - provides blas, sparse blas and graph kernels. https://github.com/github/open-source-survey, 2019.

DISTRIBUTION:

1	MS 1318	Robert J. Hoekstra, 01422
1	MS 1319	Simon D. Hammond, 01422
1	MS 1319	Arun F. Rodrigues, 01422
1	MS 1319	Gwendolyn R. Voskuilen, 01422
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories