

SANDIA REPORT

SAND2019-10389

Unclassified Unlimited Release

Printed September 3, 2019



Sandia
National
Laboratories

Balar: A SST GPU Component for Performance Modeling and Profiling

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

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

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

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & 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@osti.gov
Online ordering: <http://www.osti.gov/scitech>

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: <https://classic.ntis.gov/help/order-methods>



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 accelerators capable of maintaining context for thousands of concurrent threads resident on-chip. These threads are grouped and interleaved on a cycle-by-cycle basis among several massively parallel computing cores. One path for the design of future supercomputers relies on an ability to model the performance of these massively parallel cores at scale.

The SST framework has been proven to scale up to run simulations containing tens of thousands of nodes. A previous report described the initial integration of the open-source, execution-driven GPU simulator, GPGPU-Sim, into the SST framework. This report discusses the results of the integration and how to use the new GPU component in SST. It also provides examples of what it can be used to analyze and a correlation study showing how closely the execution matches that of a Nvidia V100 GPU when running kernels and mini-apps.

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	9
2. GPGPU-Sim Integration With SST	10
2.1. Scheduler	10
2.2. Streaming-Multiprocessor	12
3. Performance Analysis	14
3.1. Functional Testing	14
3.2. Correlation with Volta	15
3.2.1. Vector Addition	17
3.2.2. LU Decomposition	18
3.2.3. LULESH	18
3.3. Lulesh Performance Study	19
4. Conclusion	20
References	21

LIST OF FIGURES

Figure 2-1. Overall GPGPU-Sim Architecture	10
Figure 2-2. SST Element Architecture for Kernel/CTA Scheduler and SMs Components	11
Figure 2-3. Centralized GPU Scheduler Component	11
Figure 2-4. SST Link and IPCTunnels for Functional Model Support	12
Figure 2-5. SST GPGPU-Sim Integration	13
Figure 3-1. SST Model of CPU With GPU Attached	14
Figure 3-2. GPU Parameter Sweep Using LULESH	19

LIST OF TABLES

Table 3-1. Kokkos Build Parameters	15
Table 3-2. Broadwell/P100 Model Parameters	15
Table 3-3. Kokkos Kernels Unit Test Results	16
Table 3-4. CPU/V100 Model Parameters.....	17
Table 3-5. SST-GPGPU Correlation.....	17
Table 3-6. LULESH Instruction Count Percentages	18

This page intentionally left blank.

1. INTRODUCTION

As the science of high-performance computing (HPC) evolves, there is a growing need to understand and quantify the performance and architectural value of emerging technologies. To complicate the design space, the rise of General-Purpose Graphics Processing Units (GPGPUs) and other compute accelerators, which are needed to handle the growing demands of compute-heavy workloads, have become a necessary component in both high-performance supercomputers and datacenter-scale systems. That the first exascale machines will leverage the massively parallel compute capabilities of GPUs [11, 10, 14] is indicative of the growing necessity of accelerator-based node architectures. As the software stack and programming model of GPUs and their peer accelerators continue to improve, there is every indication that this trend of accelerator integration will continue, leading to a diverse ecosystem of technologies. GPUs are likely to continue to play a role as discrete accelerators or integrated as a part of an SOC. As a result, architects who wish to study the design of large-scale systems will need to evaluate system and software designs using a GPU model. However, the focus of all publicly available cycle-level simulators like GPGPU-Sim [3] is on single-node performance. In order to truly study the problem at scale, a parallelizable, multi-node GPU simulator is necessary.

The Structural Simulation Toolkit (SST) [13] is a parallel discrete event-driven simulation framework that provides an infrastructure capable of modeling a variety of high performance computing systems at many different scales. Currently used by a wide variety of government agencies and computer manufacturers to design and simulate HPC architectures and supported by a Python and C++ code base with a large array of customization options, SST offers the HPC community powerful, highly customizable, tools to create and integrate models for evaluating current and future HPC node architectures, interconnect networks, and infrastructures. What has been lacking, up to this point, was a way to integrate accelerators into a node model in SST. This report builds upon a previous report [9], providing more details on our efforts to integrate an open-source GPGPU simulator, GPGPU-Sim, into SST. This integration effort will provide SST users the ability to run GPGPU-based simulations using the Balar GPU component and will serve as the model for future accelerator integration.

2. GPGPU-SIM INTEGRATION WITH SST

The latest version of GPGPU-Sim is a (mostly) object-oriented design written in C++ with a clean delineation between the functional model and the timing model. A high-level overview of the organization is shown in Figure 2-1. The SIMT blocks can be thought of as Nvidia Streaming Multiprocessors (SMs) or AMD Compute Units (CUs). The interconnect is a modified version of BookSim [12], which can model a variety of topologies, and the memory partition is a custom model. The remainder of this section describes the integration of GPGPU-Sim with SST.

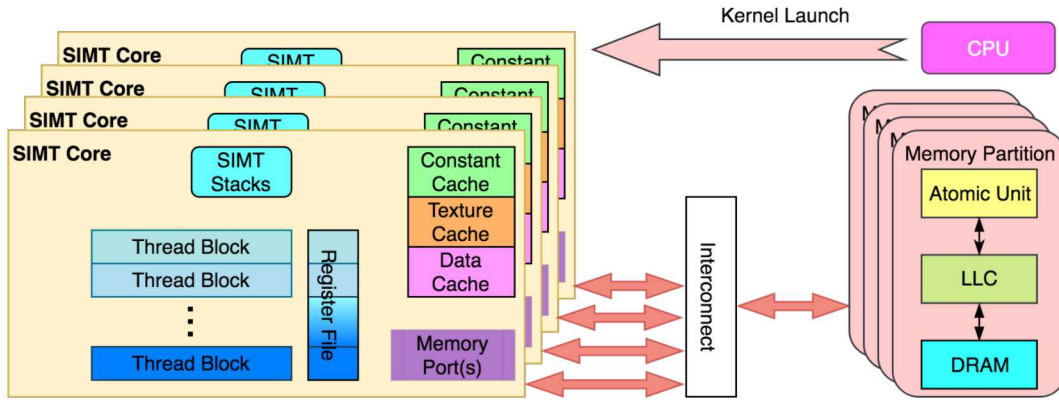


Figure 2-1. Overall GPGPU-Sim Architecture [3]

2.1. SCHEDULER

The first step in integrating GPGPU-Sim into SST is to handle the interaction with a 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 [2]. A block diagram of the model is shown in Figure 2-2. 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).

The 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 enqueueing kernels for execution are sent to the GPU scheduler element, which co-ordinates the launching of CTAs on the SMs, e.g. `cudaConfigureCall` and `cudaLaunch`.

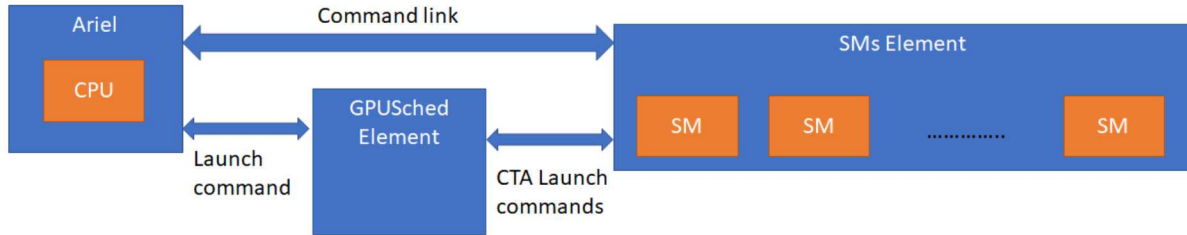


Figure 2-2. SST Element Architecture for Kernel/CTA Scheduler and SMs Components

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.

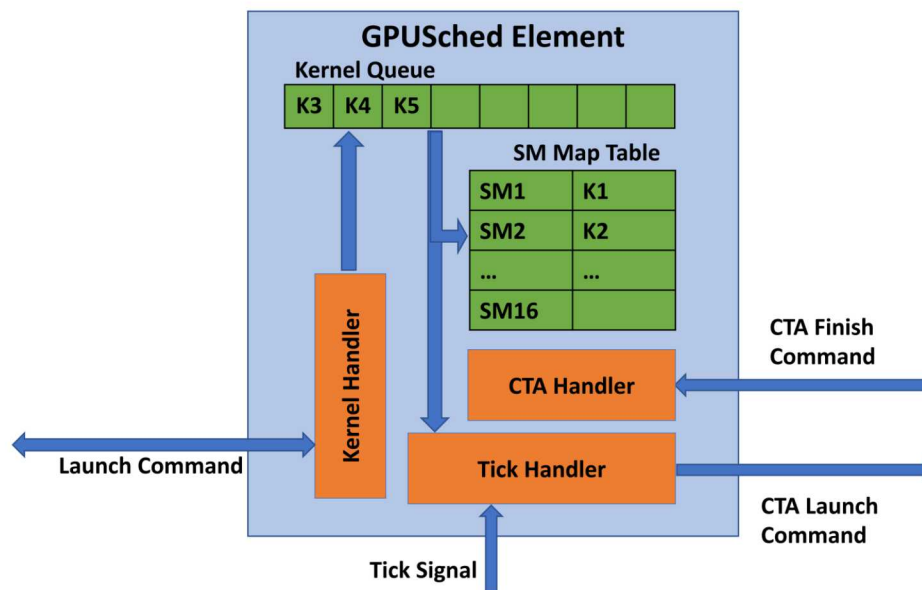


Figure 2-3. Centralized GPU Scheduler Component

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 2-3 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.

2.2. STREAMING-MULTIPROCESSOR

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.

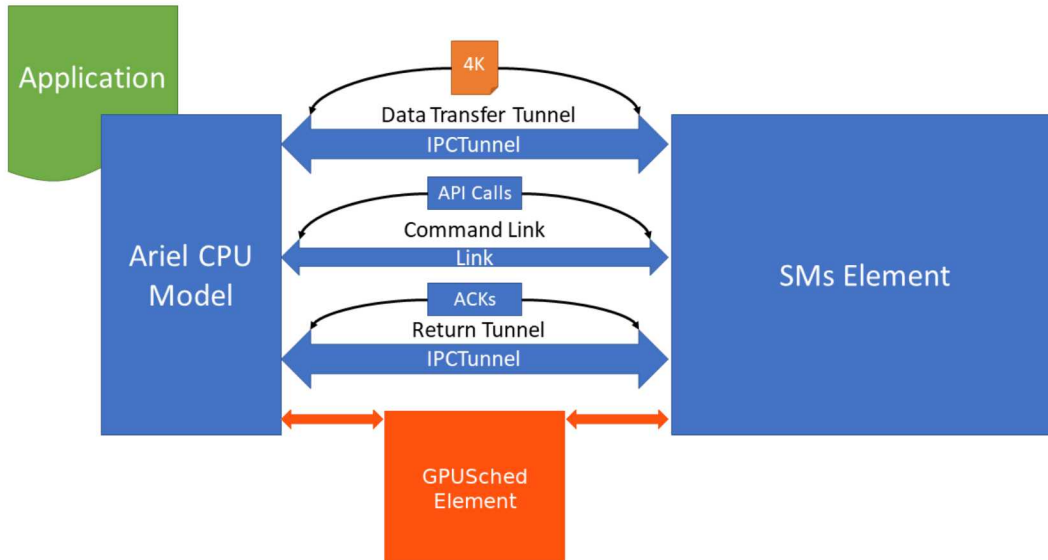


Figure 2-4. SST Link and IPCTunnels for Functional Model Support

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 Figure 2-4. 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 complete the integration with SST, everything except the SIMT units are replaced with SST components, as shown in Figure 2-5. The host CPU is swapped for a SST execution component, such as Ariel or Juno. The interconnect is swapped for a SST networking component, such as Shogun, Kingsley, or Merlin. The memory partition, caches, and ports is swapped for SST

memHierarchy components – caches and backing store models such as TimingDRAM, Simple-Mem, or Cramsim. A detailed description of how to use the GPU component can be found in [9]. The remainder of this paper is dedicated to a discussion of performance.

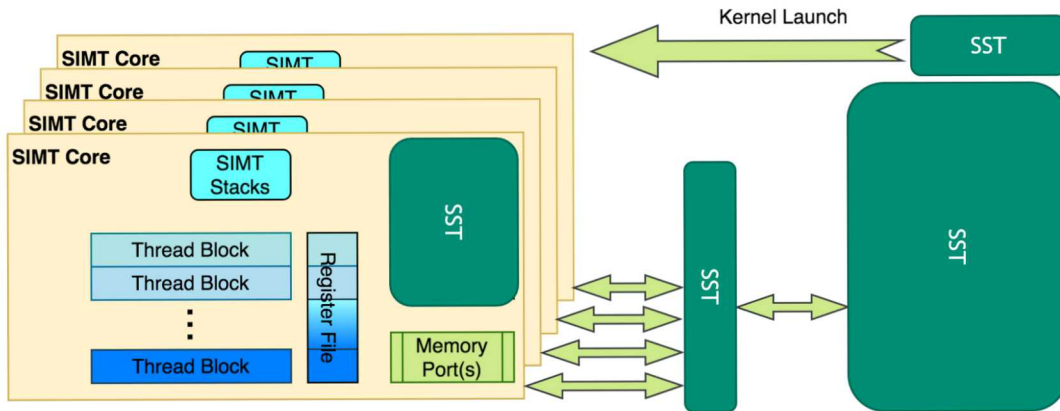


Figure 2-5. SST GPGPU-Sim Integration

3. PERFORMANCE ANALYSIS

This chapter is devoted to an analysis of the functional and timing models in SST when using the GPU component as a part of a generalized compute node. Since GPUs in HPC function solely as co-processors, functionally executing GPU-enabled binaries requires the CPU to initialize and launch kernels of work to the GPU. Figure 3-1 shows the general simulation model used for the evaluation. The Command Link and Data Link serve as the transport mechanism for the CPU (host) to launch and coordinate work with the GPU (device). The other components in each model can be tailored to fit any host and device that one wishes to model.

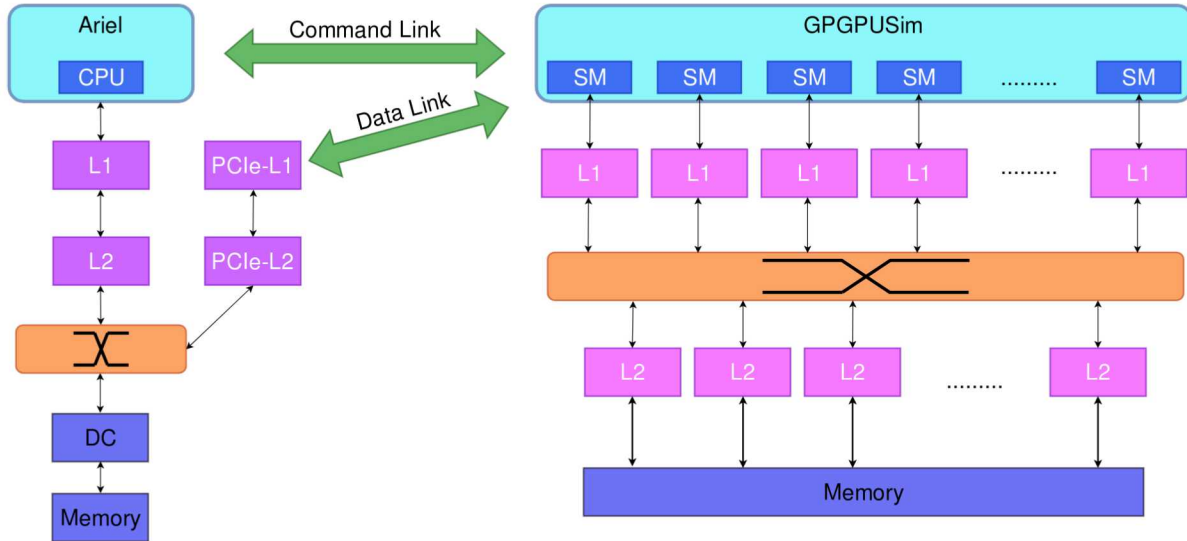


Figure 3-1. SST Model of CPU With GPU Attached

3.1. FUNCTIONAL TESTING

The functional correctness of the model was validated using the unit tests from the Kokkos Kernels suite [15]. The unit tests were compiled using the parameters in Table 3-1. The target node architecture was assumed to be an Intel Broadwell attached to Nvidia Pascal GPUs. This target architecture was chosen based on hardware availability, specifically Sandia's Doom cluster, which is based on the CTS-1/TLCC3 procurement. The SST model is derived from Figure 3-1 using the model parameters in Table 3-2 to represent a Nvidia P100 SXM2 [1].

Table 3-3 shows the Kokkos Kernels unit tests that were run. With the current implementation of SST-GPU, 54 out of 89 tests run to completion and pass. The passing tests are highlighted in

Table 3-1. Kokkos Build Parameters

KOKKOSKERNELS_SCALARS=double
KOKKOSKERNELS_LAYOUTS=left
KOKKOSKERNELS_ORDINALS=int
KOKKOSKERNELS_OFFSETS=int
KOKKOSKERNELS_DEVICES=Cuda,Serial
KOKKOS_ARCHS=BDW,Pascal60

Table 3-2. Broadwell/P100 Model Parameters**(a) CPU**

Clock	1200MHz
DDR Clock	2400
DDR Capacity	16384MiB
Mesh Frequency	800MHz
Mesh Input Ports	1
Mesh Output Ports	1
Data Link Latency	23840ps
Command Link Latency	23840ps

(b) GPU

Clock	1328MHz
SMs	56
L2 Slices	8
L2 Capacity	512KiB per slice
HBM Capacity	16384MiB
HBM Stacks	4
Crossbar Frequency	1000MHz
Crossbar Input Ports	2
Crossbar Output Ports	1

green. Of the remaining tests, all but the yellow tests fail in both SST-GPGPU and GPGPU-Sim. The tests in pink fail because the PTX parser cannot locate a post-dominator. This can happen when the control flow is too complex for the parser and a path from the point of divergence to a convergence point for all of the threads cannot be found. There are plans to work with the Kokkos Kernels developers to find a solution. The tests in purple fail because of a bug in the parser that creates an empty file, which should contain the PTX for the kernel, but the simulator crashes when it attempts to read from it. Neither the SST developers nor the GPGPU-Sim developers have been able to locate the problem. The tests in red fail because the problem size is overflowing the address space. Both teams are actively engaged on a fix for this. The two remaining tests, in yellow, run to completion and pass in GPGPU-Sim but have run for more than 10 days without completion in Balar. It is believed that they would complete successfully if given more run time.

3.2. CORRELATION WITH VOLTA

A validation sweep was run using two kernels and a mini-app. The three applications were run using an SST model that approximates a Nvidia V100 attached to a CPU. The simulation parameters are shown in Table 3-4. The overall kernel runtime was compared with the results of running the three applications through nvprof on Sandia's Waterman testbed, which is comprised of IBM Power9 CPUs and Nvidia Volta GPUs. Table 3-5 shows the total number cycles that each application took on the SST-GPU model and on the native V100. Note that this is only cycles where a kernel was running and does not include host execution time. There are challenges isolating

Table 3-3. Kokkos Kernels Unit Test Results

1	abs_double	46	batched_scalar_serial_gemm_nt_nt_double_double
2	abs_mv_double	47	batched_scalar_serial_gemm_t_nt_double_double
3	asum_double	48	batched_scalar_serial_gemm_nt_t_double_double
4	axpby_double	49	batched_scalar_serial_gemm_t_t_double_double
5	axpby_mv_double	50	batched_scalar_serial_trsm_l_l_nt_u_double_double
6	axpy_double	51	batched_scalar_serial_trsm_l_l_nt_n_double_double
7	axpy_mv_double	52	batched_scalar_serial_trsm_l_u_nt_u_double_double
8	dot_double	53	batched_scalar_serial_trsm_l_u_nt_n_double_double
9	dot_mv_double	54	batched_scalar_serial_trsm_r_u_nt_u_double_double
10	mult_double	55	batched_scalar_serial_trsm_r_u_nt_n_double_double
11	mult_mv_double	56	batched_scalar_serial_trsm_l_l_t_u_double_double
12	nrm1_double	57	batched_scalar_serial_trsm_l_l_t_n_double_double
13	nrm1_mv_double	58	batched_scalar_serial_trsm_l_u_t_u_double_double
14	nrm2_double	59	batched_scalar_serial_trsm_l_u_t_n_double_double
15	nrm2_mv_double	60	batched_scalar_serial_gemv_nt_double_double
16	nrm2_squared_double	61	batched_scalar_serial_gemv_t_double_double
17	nrm2_squared_mv_double	62	batched_scalar_serial_trsv_l_nt_u_double_double
18	nrminf_double	63	batched_scalar_serial_trsv_l_nt_n_double_double
19	nrminf_mv_double	64	batched_scalar_serial_trsv_u_nt_u_double_double
20	reciprocal_double	65	batched_scalar_serial_trsv_u_nt_n_double_double
21	reciprocal_mv_double	66	batched_scalar_team_set_double_double
22	scal_double	67	batched_scalar_team_scale_double_double
23	scal_mv_double	68	batched_scalar_team_gemm_nt_nt_double_double
24	sum_double	69	batched_scalar_team_gemm_t_nt_double_double
25	sum_mv_double	70	batched_scalar_team_gemm_nt_t_double_double
26	update_double	71	batched_scalar_team_gemm_t_t_double_double
27	update_mv_double	72	batched_scalar_team_trsm_l_l_nt_u_double_double
28	gemv_double	73	batched_scalar_team_trsm_l_l_t_n_double_double
29	gemm_double	74	batched_scalar_team_trsm_l_u_nt_u_double_double
30	sparse_spghemm_double_int_int_TestExecSpace	75	batched_scalar_team_trsm_l_u_nt_n_double_double
31	sparse_spadd_double_int_int_TestExecSpace	76	batched_scalar_team_trsm_r_u_nt_u_double_double
32	sparse_gauss_seidel_double_int_int_TestExecSpace	77	batched_scalar_team_trsm_r_u_nt_n_double_double
33	sparse_block_gauss_seidel_double_int_int_TestExecSpace	78	batched_scalar_team_trsm_l_l_t_u_double_double
34	sparse_crsmatrix_double_int_int_TestExecSpace	79	batched_scalar_team_trsm_l_l_t_n_double_double
35	sparse_blkcrsmatrix_double_int_int_TestExecSpace	80	batched_scalar_team_trsm_l_u_t_u_double_double
36	sparse_replaceSumIntoLonger_double_int_int_TestExecSpace	81	batched_scalar_team_trsm_l_u_t_n_double_double
37	sparse_replaceSumInto_double_int_int_TestExecSpace	82	batched_scalar_team_gemv_nt_double_double
38	graph_graph_color_double_int_int_TestExecSpace	83	batched_scalar_team_gemv_t_double_double
39	graph_graph_color_deterministic_double_int_int_TestExecSpace	84	batched_scalar_serial_lu_double
40	graph_graph_color_d2_double_int_int_TestExecSpace	85	batched_scalar_serial_inverselu_double
41	common_ArithTraits	86	batched_scalar_serial_solvelu_double
42	common_set_bit_count	87	batched_scalar_team_lu_double
43	common_ffs	88	batched_scalar_team_inverselu_double
44	batched_scalar_serial_set_double_double	89	batched_scalar_team_solvelu_double
45	batched_scalar_serial_scale_double_double		

the cause of the performance gaps. This is one of the largest, if not the largest, node simulation that has been run with 139 unique components and 906 links (the statistics output contains nearly 20k unique entries). The complex model interactions and scale make it difficult to pinpoint where models are lacking in detail or are incorrect. Turning on debug for even a small run can produce multi-terabyte output files. That being said, the authors do have some intuition into why there are gaps and how to close them.

Table 3-4. CPU/V100 Model Parameters

(a) CPU		(b) GPU	
Clock	2660MHz	Clock	1312MHz
DDR Clock	2666	SMs	84
DDR Capacity	16384MiB	L2 Slices	32
Mesh Frequency	800MHz	L2 Capacity	192KiB per slice
Mesh Input Ports	1	HBM Capacity	16384MiB
Mesh Output Ports	1	HBM Stacks	4
Data Link Latency	23840ps	Crossbar Frequency	1200MHz
Command Link Latency	23840ps	Crossbar Input Ports	2
		Crossbar Output Ports	1

Table 3-5. SST-GPGPU Correlation

	P9/V100	SST-GPGPU	Error
vectorAdd	5271	5751	9.09
lud	494519	605685	22.48
lulesh	12454750	11896477	4.48

3.2.1. Vector Addition

The vectorAdd application is from the Cuda SDK with error checking removed. It implements element by element vector addition using an array with 163840 elements.

vectorAdd contains a single kernel with a single invocation that, essentially, streams through memory performing integer operations. It was expected that this would have a higher correlation, but the fact that there are so many memory dependencies and memory operations make the results highly dependent on the model for the backing store. A number of models were tried and flaws were found in all of them. With the exception of Cramsim, all of the models are derived from simple DRAM models and are unable to accurately replicate the behavior of HBM. It is believed that there is an issue in the memory controller that Cramsim uses and that when this is solved, it will serve as a good model for HBM2. However, the timingDRAM model clearly provides enough detail for kernels that are not bottle-necked by memory bandwidth.

3.2.2. LU Decomposition

The lud application is from the Rodinia benchmark suite [4][5] and implements the LU decomposition algorithm to solve a set of linear equations using a 256x256 element matrix.

The lud application from Rodinia contains 3 kernels with 46 total kernel launches. lud has the worst correlation. The `perimeter` and `diagonal` kernels occupy the majority of the compute time – `diagonal` has 16 invocations and consumes 63% of the time; `perimeter` has 15 invocations and consumes 22% of the time; `internal` has 15 invocations and consumes 14% of the time. `perimeter` and `diagonal` spend 50% and 80% of their time inactive, respectively, due to the number of divergences. Given that LULESH has a much greater diversity of instructions, including FP64, and the previously reported issues determining control flow, it's unlikely that the problem lies in the ALU models and more likely that the issues stem from how the GPU model handles divergences or complex issues exposed by the differences in using PTX verses SASS.

3.2.3. LULESH

LULESH is one of the most widely used mini-applications developed by the US Department of Energy. The code was originally developed by Lawrence Livermore National Laboratory to represent challenging hydrodynamics algorithms that are performed over unstructured meshes [7][8]. Such algorithms are common in many high-performance computing centers and are particularly prevalent within the NNSA laboratories. In the original LULESH specification, the authors state that such algorithms routinely count in the top ten application codes in terms of CPU hours utilized [7].

The unstructured nature of LULESH presents challenges for the design of memory subsystems, not least because operands are gathered from a fairly limited locale but are done so sparsely. This makes efficient streaming and vectorization of the data operations difficult and places additional pressure on the memory subsystem (typically the L2 caches) to provide operands quickly.

For this experiment, the problem size was set to 22 with 50 iterations, leading to an application that contains 26 kernels with 1400 total invocations. The top three kernels, in terms of execution time, provided a good mix of operations, shown in Table 3-6. The diversity of operations in lulesh, compared to the other too applications, obfuscates the areas where the simulation is lacking, leading to higher correlation with the V100 target platform.

It's clear that a more detailed study is needed to isolate the weaknesses in the models.

Table 3-6. LULESH Instruction Count Percentages (nvprof)

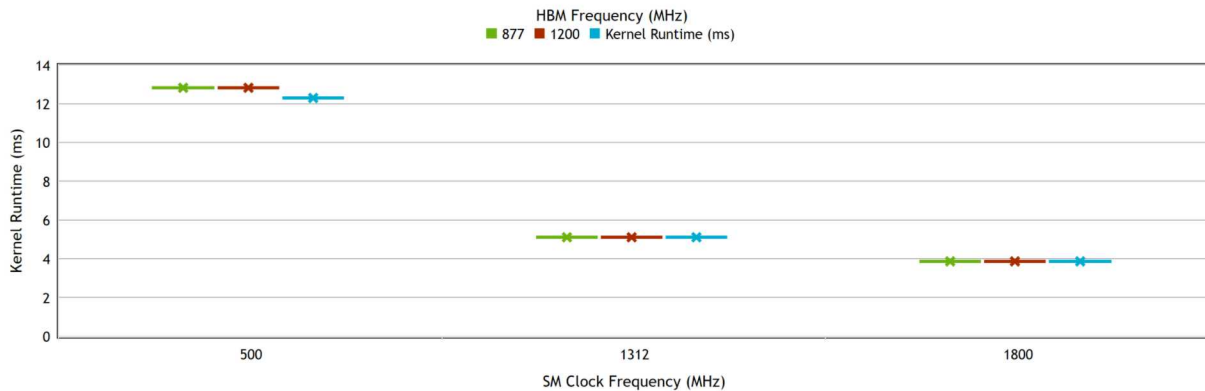
	FP32	FP64	INT	CTRL	L/S	MISC	INACTIVE
CalcFBHourglassForceForElems	1	10	11	10	12	31	23
CalcPressureForElems	5	17	27	2	19	16	15
CalcHourglassControlForElems	0	25	21	3	38	9	1

3.3. LULESH PERFORMANCE STUDY

A parameter sweep was performed using LULESH, described in Section 3.2.3. The device clock was varied from 500MHz to 1312MHz to 1800MHz. The memory clock was varied from 877MHz to 1200MHz to 1600MHz. Figure 3-2 shows the results, where lower runtime time is better.

As expected, changing the frequency of the backing store has little effect on LULESH for this problem size because it is not memory bandwidth bound. The most improvement is seen at the low device clock frequency, but even there it's only a 1.04x speedup. However, increasing the frequency of the SMs does improve the performance noticeably. Going from 500MHz to 1312MHz shows a 2.5x speedup; going from 1312MHz to 1800MHz shows a further 1.3x speedup.

Although this was a small study, one can imagine being able to run a more complete parameter sweep over any of the Balar parameters.



**Figure 3-2. GPU Parameter Sweep Using LULESH
(Baseline was 1312MHz/877MHz)**

4. CONCLUSION

This report described the final integration of phase one of the SST-GPU project. Functional validation against the Kokkos Kernels unit tests shows that the GPU component can successfully run more than 60% of the tests with a path to reach a coverage of greater than 90%. Correlation with the Waterman V100 testbed is excellent, showing 4-22% error in the runtime for the applications considered. 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.

REFERENCES

- [1] NVIDIA Tesla P100 White Paper. Technical report, Nvidia, 2016.
- [2] NVIDIA Volta V100 White Paper. Technical report, Nvidia, 2017.
- [3] 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.
- [4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.
- [5] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, Dec 2010.
- [6] 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.
- [7] R.D. Hornung, , J.A. Keasler, and M.B. Gokhale. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory, CA, United States, 2011.
- [8] I. Karlin, J.A. Keasler, and J.R. Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, Livermore, CA, United States, August 2013.
- [9] Mahmoud Khairy, Mengchi Zhang, Roland Green, Simon David Hammond, Robert J. Hoekstra, Timothy Rogers, and Clayton Hughes. SST-GPU: An Execution-Driven CUDA Kernel Scheduler and Streaming-Multiprocessor Compute Model. Internal Report SAND2019-1967, 2019.
- [10] Morgan L McCorkle. U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL. <https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl>, May 2019.
- [11] 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.

- [12] Nan Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96, April 2013.
- [13] 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.
- [14] Tiffany Trader. U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL. <https://www.hpcwire.com/2019/08/13/cray-wins-nnsa-livermore-el-capitan-exascale-award>, August 2019.
- [15] 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

Email—██

Name	Org.	Sandia Email Address
Robert J. Hoekstra	01422	rjhoeks@sandia.gov
Simon D. Hammond	01422	sdhammo@sandia.gov
David Z. Poliakoff	01422	dzpolia@sandia.gov
Arun F. Rodrigues	01422	afrodri@sandia.gov
Gwendolyn R. Voskuilen	01422	grvosku@sandia.gov
Sivasankaran Rajamanickam	01465	srajama@sandia.gov
Daniel Sunderland	01465	dsunder@sandia.gov
Micheal W. Glass	01545	mwglass@sandia.gov
Technical Library	01177	libref@sandia.gov

This page intentionally left blank.



Sandia
National
Laboratories

Sandia National Laboratories is a
multimission laboratory managed
and operated by National
Technology & 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.