Programmable accelerators have become commonplace in modern computing systems. Advances in programming models and the availability of unprecedented 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.

A Introduction

As the architectures of high-performance computing (HPC) evolves, there is a growing need to understand and quantify the performance and design benefits 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 [10, 11, 13] is indicative of the growing necessity of accelerator-based node architectures to obtain high compute throughputs. 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 (*e.g.* GPGPU-Sim [3]) to date has been on single-node performance. In order to truly study the problem at scale, and to permit larger workloads to be evaluated, a parallelizable, multi-node GPU simulator is necessary.

The Structural Simulation Toolkit (SST) [12] 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 and interconnect networks. What has been lacking, up to this point, has been a method to integrate accelerators into a node model in SST. This report builds upon previous work [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 components and will serve as a model for future accelerator integration studies.

B Balar Components

B.1 GPU Scheduler

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 types of discrete SST components – a CTA scheduler and SM groups [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). Table 1 enumerates the CUDA API calls currently intercepted and sent to the GPU components. 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 7 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 1: **CUDA API Calls Forwarded to the GPU components. Sched and SM represent CUDA calls sent to the scheduler or the SM groups.**

CUDACall	Sched	SM
`__cudaRegisterFatBinary`	Yes	Yes
`__cudaRegisterFunction`	Yes	Yes
`cudaMalloc`	Yes	No
`cudaMemcpy`	Yes	No
`cudaMemset`	Yes	No
`cudaConfigureCall`	Yes	Yes
`cudaSetupArgument`	Yes	Yes
`cudaFree`	Yes	No
`cudaLaunch`	Yes	Yes
`cudaGetLastError`	Yes	No
`cudaFuncSetCacheConfig`	Yes	No
`cudaSetDevice`	Yes	No
`cudaGetDeviceCount`	Yes	No
`cudaGetDeviceProperties`	Yes	No
`__cudaRegisterVar`	Yes	Yes
`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`	Yes	No

Aside from the basic functional model provided by GPU-SST, an initial performance model has also been developed. Figure 1 details the overall architecture. A CPU component (Ariel in the initial implementation) is connected via SST links to 2 types of GPU components: a centralized kernel and CTA scheduler (GPUSched) and SM Groups, which implement the timing and functional model for the GPU cores. When CUDA calls are intercepted from the CPU, API commands are sent to the CTA scheduler. When the scheduler launches a CTA for a kernel, CTA commands are sent to the corresponding SM groups to execute. CUDA calls related to queuing kernels and memory operations are handled by the scheduler, while execution related CUDA calls are redirect

to SM groups, since the functional model for executing the GPU CTAs lives inside the SM groups. Table 1 shows where CUDA calls send to.

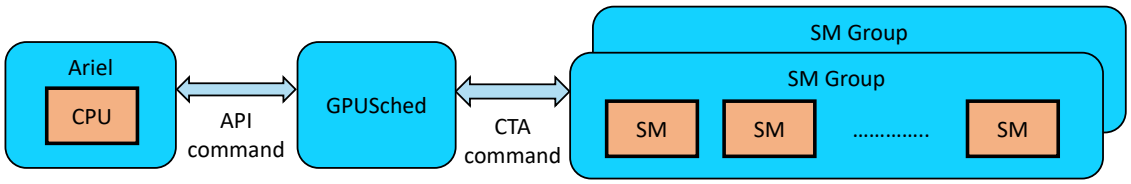

Figure 1: **SST component architecture for CTA scheduler and SM groups**

As CTAs complete on the SMs, commands are sent back to the GPU scheduler component, 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.

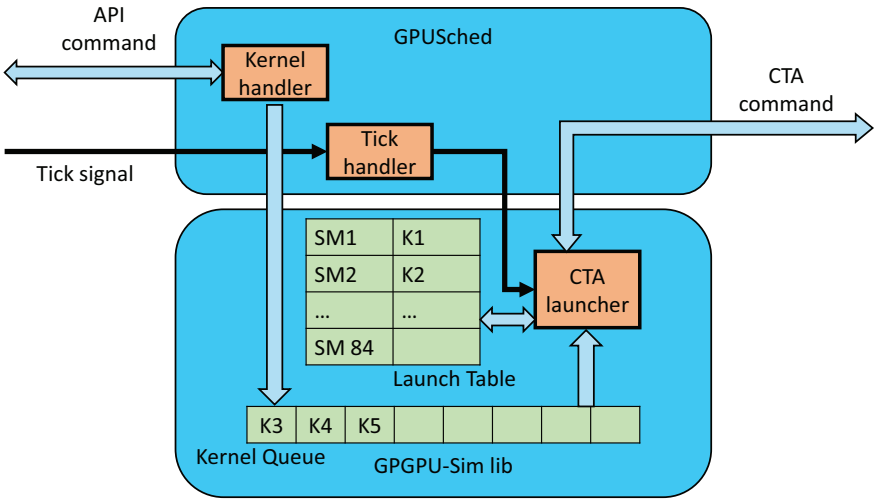

Figure 2: **Centralized GPU scheduler component**

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 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 launch 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 launch table can be freed.

B.2 SM Groups

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.

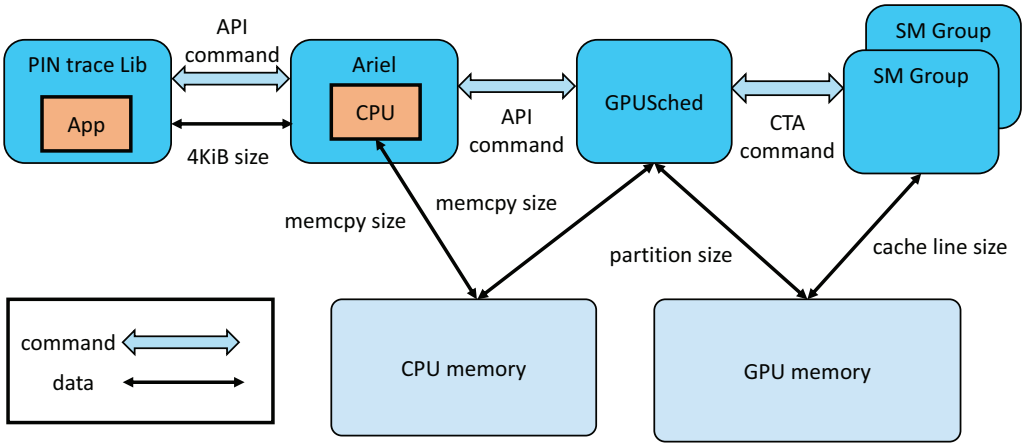

Figure 3: **Data transfer flow for functional simulation**

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 link between CPU/GPU and memory hierarchy from SST, as shown in 3. Data are transferred from the application to Ariel through inter-process communication tunnels. Ariel then communicates with the GPU scheduler through the CPU memory. The GPU schduler then write the data to the GPU memory. When an SM request a piece of data, the SM accesses the GPU memory for it. The tunnels utilize 4KiB size as the granularity, while CPU and GPU components employ larger size non-cacheable requests to access to CPU and GPU memories.

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 4 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, TimingDRAM, and CramSim [6]; 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 1.

Listing 1: **Sample SST-GPGPU Configuration**

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

[ariel]
executable: ./vectorAdd
gpu_mode: 2

[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
num_sm_blobs: 1

[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
```

B.3 GPGPU-Sim as a Library

We use GPGPU-Sim simulator as a library to provide CTA scheduling and GPU core functionality. GPU Scheduler and SM Group components invokes on GPGPU-Sim library (libcudart.so) with internal API calls.

To support SST-GPU to run on one thread, on multiple threads or on multiple processes on one node or multiple nodes, we refactor GPGPU-Sim simulator to avoid static and global variables. Instead, GPU components construct a GPU context data structure before using GPGPU-Sim library so that each component can keep an individual context. Moreover, we design an option inside the context data structure to manage the library in scheduler mode or SM mode. GPGPU-Sim library works as a CTA scheduler to issue CTAs in scheduler mode and works as a group of GPU cores in SM mode.

GPGPU-Sim simulator separates functional model with performance model, so the instruction operations are simulated on the issue pipeline stage. However, the load store unit (LSU) sends out memory requests to GPU memory hierarchy on the execution stage. This design breaks if SM groups need to access the GPU memory components. Therefore, we replay the memory instruction operations after the data returned to LSU in SM mode.

The SST-GPU components call to GPGPU-Sim library for functionality, while GPGPU-Sim library calls back to the GPU components for CTA command and memory accesses to GPU memory hierarchy. However, we separate the compilation of GPGPU-Sim simulator and SST components so that GPGPU-Sim can be compatible to mainstream. To achieve the independent compliation, we design the parent classes for scheduler and memory interface, so that the components can rewrite to utilize the SST infrastructure.

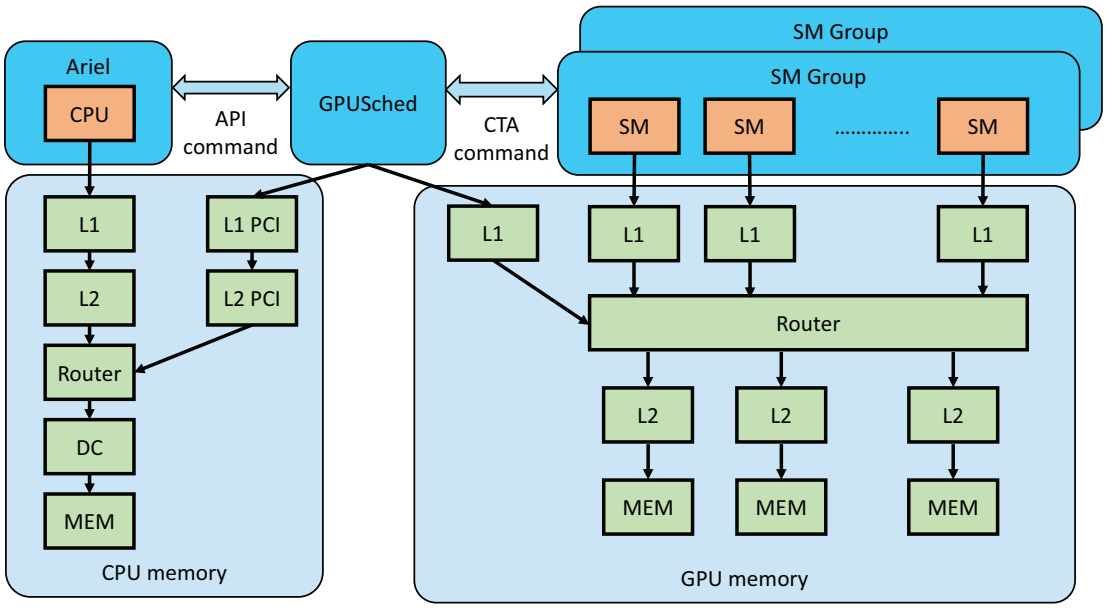

Figure 4: **Timing and memory model for SMs component**

C Evaluation

C.1 Correlation

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 2. 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 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 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 2: **CPU/V100 Model Parameters**

(a) CPU

Clock	2660MHz
DDR Clock	2666
DDR Capactiy	16384MiB
Mesh Frequency	800MHz
Mesh Input Ports	1
Mesh Output Ports	1
Data Link Latency	23840ps
Command Link Latency	23840ps

(b) GPU

Clock	1312MHz
SMs	84
L2 Slices	32
L2 Capactiy	192KiB per slice
HBM Capacity	16384MiB
HBM Stacks	4
Crossbar Frequency	1200MHz
Crossbar Input Ports	2
Crossbar Output Ports	1

Table 3: **SST-GPGPU Correlation**

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

C.2 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.

C.3 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.

C.4 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 4. 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.

C.5 Kokkos

The functional correctness of the model was validated using the unit tests from the Kokkos Kernels suite [14]. The unit tests were compiled using the parameters in Table 5. The target node architecture was assumed to be an Intel Broadwell attached to an NVIDIA Pascal GPUs. This target architecture was chosen based on hardware availability, specifically Sandia's Doom cluster, which

Table 4: **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

is based on the CTS-1 procurement. The SST model is derived from Figure **??** using the model parameters in Table‘6 to represent an NVIDIA P100 SXM2 [1].

Table 5: **Kokkos Build Parameters**

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

Table 6: **Broadwell/P100 Model Parameters**

(a) CPU

Clock	1200MHz
DDR Clock	2400
DDR Capactiy	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 Capactiy	512KiB per slice
HBM Capacity	16384MiB
HBM Stacks	4
Crossbar Frequency	1000MHz
Crossbar Input Ports	2
Crossbar Output Ports	1

Table 7 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 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.

Table 7: **Kokkos Kernels Unit Test Results**

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

C.6 Lulesh

A parameter sweep was performed using LULESH, described in Section C.4. The device clock was varied from 500MHz to 1312MHz to 1800MHz. The memory clock was varied from 877MHz to 1200MHz to 1600MHz. Figure 5 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 at this frequency the speedup is still small at 1.04x. 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.

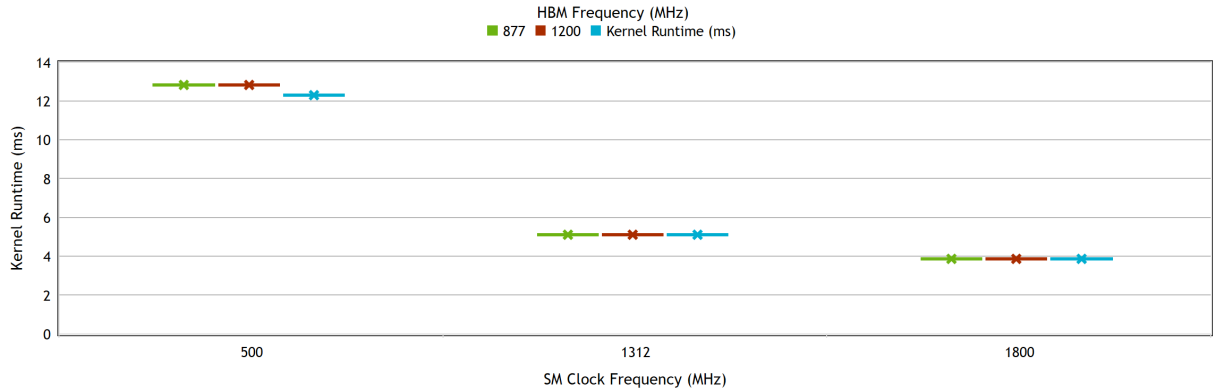

Figure 5: **GPU Parameter Sweep Using LULESH (Baseline was 1312MHz/877MHz)**

C.7 Parallel Simulation Performance

Fixme: *@Mengchi - put the new performance data and discussion in here. Make sure to run it through Grammarly and I will vett it too.*

D Conclusion

This report described the final integration of the SST-GPU project. Functional validation against the Kokkos Kernels unit tests shows that the GPU component can successfully run more than XX% of the tests with a path to reach a coverage of greater than XX%. Correlation with the Waterman V100 testbed is excellent, showing XX% error in the runtime for the applications considered. The final phase of the project has involved parallelizing the scheduler and groups of SMs, dubbed *SM Groups*, using both multi-threaded and multi-process SST. Initial performance results demonstrate good scalability using default scheduling policies with additional opportunities to improve parallel scheduling performance.

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.

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] 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.

[13] 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.

[14] 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.