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 SST's MemHierarchy to model a flexible memory system. 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 [**?**]. As the software stack and programming model of GPUs and their accelerator peers 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 that their techniques have using a GPU model. However, the focus of all publicly available cycle-level simulators (*e.g*GPGPU-Sim [**?**]) to date, has been on single-node simulation performance. In order to truly study the problem at scale, or for the simulation of large workloads, a parallelizable, multi-node GPU simulator is necessary.

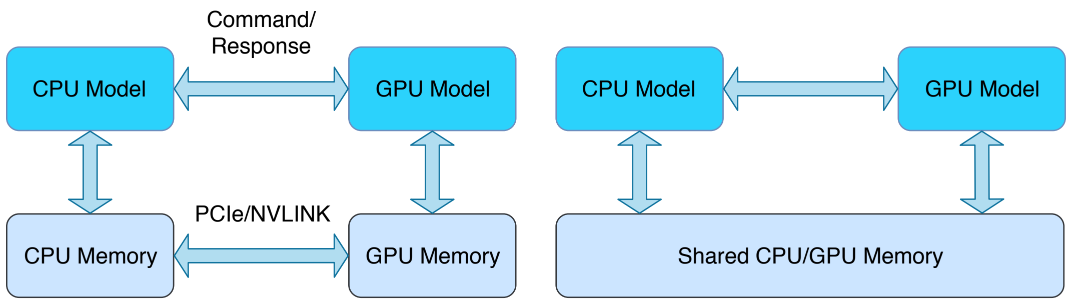

Figure 1: **High-level CPU/GPU interaction model**

Figure 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 (such as NVLink). The right shows an Accelerated Processing Unit (APU model where the CPU and GPU share a single memory. Note that in even in the discrete memory case, modern memory translation units allow the CPU and GPU to share the same virtual address space, although the memories themselves are discrete components.

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 [**?**]. This report details our efforts to integrate the functional and streaming multiprocessor core models from the open-source simulator GPGPU-Sim into SST. 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 [**?**]. 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 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 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: **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 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 launching of CTAs on the SMs, e.g. `cudaConfigureCall` and `cudaLaunch`.

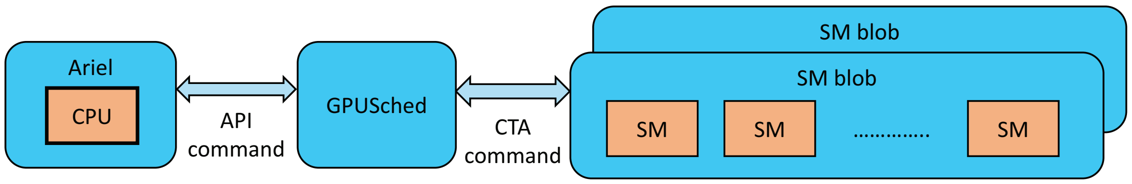

Figure 2: **SST Element architecture for kernel/CTA scheduler and SMs components**

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

Table 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

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.

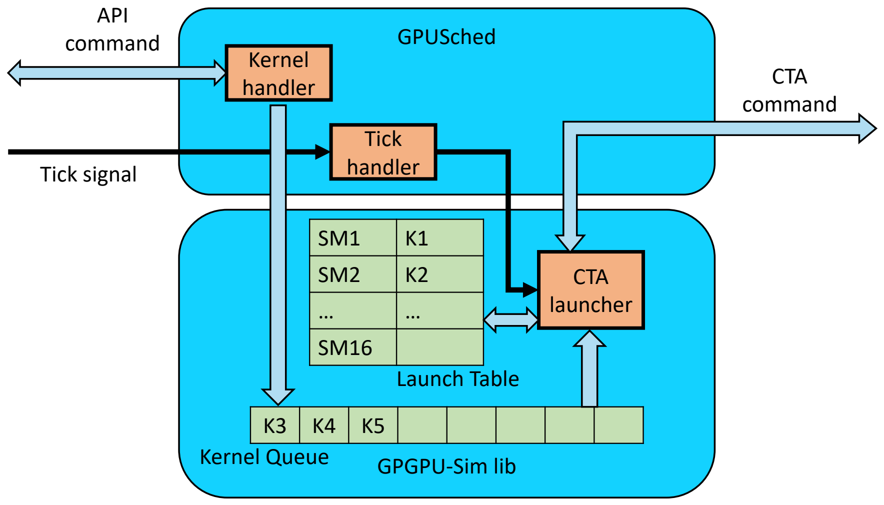

Figure 3: **Centralized GPU Scheduler component**

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 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. 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.

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 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.

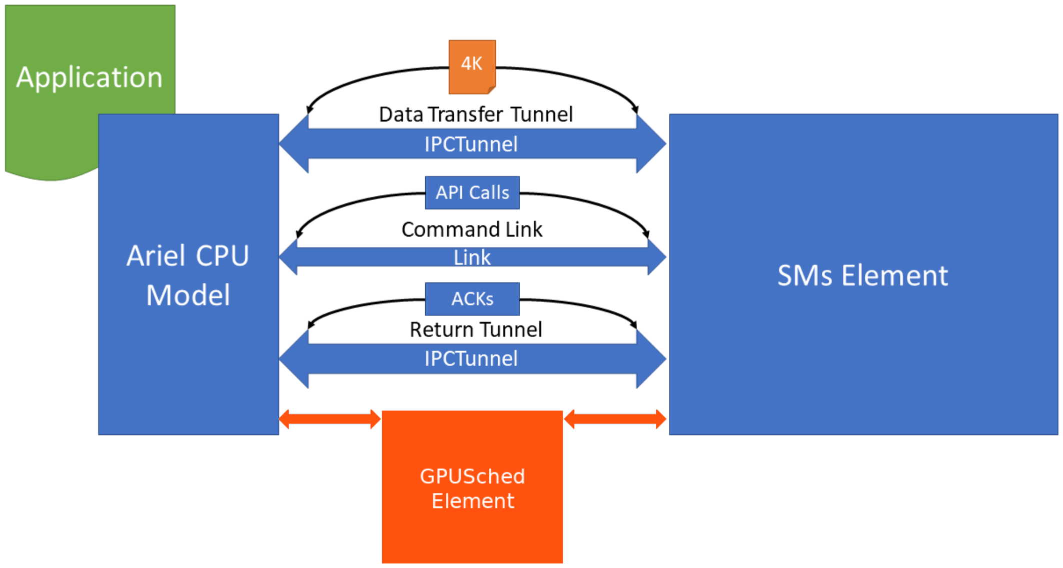

Figure 4: **SST Link and IPCTunnels for functional model support**

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 5 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 [**?**]; 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.

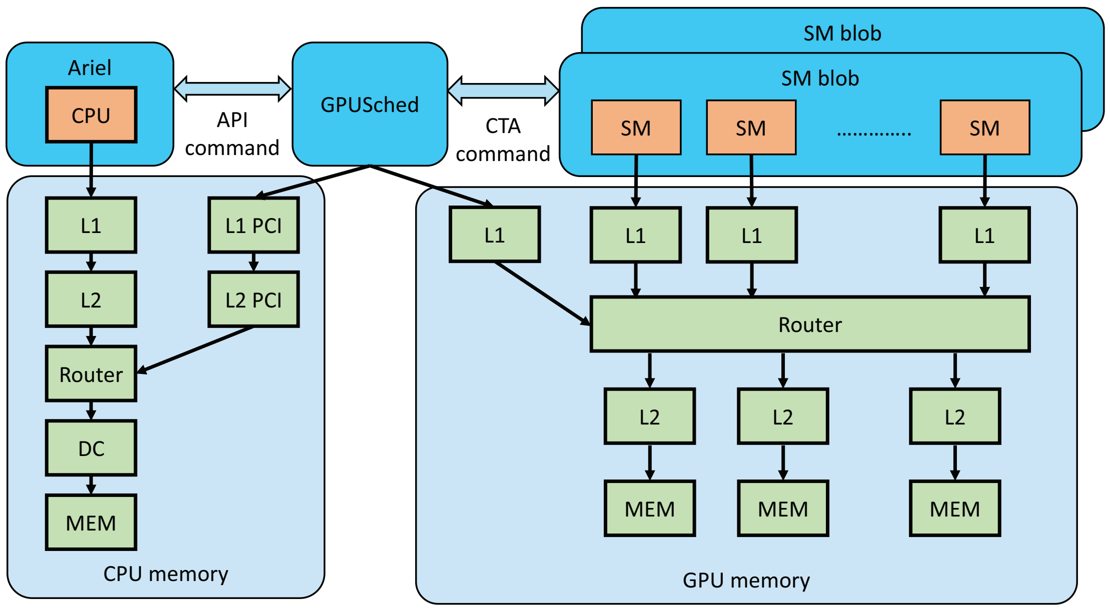

Figure 5: **Timing and memory model for SMs component**

***Listing 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
```

To evaluate the performance correlation of our GPU-SST model, versus both real hardware and the existing GPGPU-Sim memory system implementation an execution time correlation is done on the vectorAdd benchmark from the CUDA SDK. Figure 6 shows the results of this timing analysis. The most accurate GPGPU-Sim timing model is reasonably accurate (within 25% of the hardware results), however GPU-SST is much closer to real hardware, showing just and 8% deviation from a silicon Nvidia Titan V card.

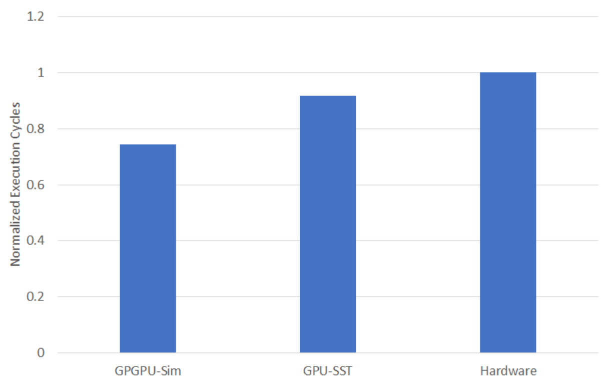

Figure 6: **Normalized execution time for 160k element vector addition kernel – SST-GPU is within 8% of the silicon of the Titan V**

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. 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.