Developments in Performance and Portability of BlockGen

E Bothmann^3, JT Childers^1, W Giele^2, S Höche^2, J Isaacson^2, M Knobbe^3, R Wang^1

^{1}Argonne National Laboratory, Lemont, IL, 60439, USA
^{2}Fermi National Accelerator Laboratory, Batavia, IL 60510, USA
^{3}Institut für Theoretische Physik, Georg-August-Universität Göttingen, 37077 Göttingen, Germany

Abstract. For more than a decade Monte Carlo event generators with the current matrix element algorithms have been used for generating hard scattering events on CPU platforms, with excellent flexibility and good efficiency. While the HL-LHC is approaching and precision requirements are becoming more demanding, many studies have been made to solve the bottleneck in the current Monte Carlo event generator tool chains. The novel BlockGen family of fast matrix element algorithms shown in this report, is one of the new developments that are more suitable for GPU acceleration. We report the development experience of porting BlockGen using Kokkos. Moreover, we discuss the performance of the Kokkos version in comparison with the dedicated GPU version in CUDA.

1. Introduction

Monte Carlo (MC) event generators are the first step in the simulation chain of LHC experiments, typically followed by detector response simulation and reconstruction of the simulated physics objects. The HL-LHC will facilitate measurements at an unprecedented precision by producing an order of magnitude more collision events, requiring even larger MC event samples. This is expected to put pressure on computing resources [1, 2]. In addition, the computing architecture landscape has become more heterogeneous with resources including more dependence on accelerators for computational power and some including custom hardware designed specifically for AI training.

The computing and theory communities are working toward addressing these challenges by developing event generators that run on supercomputers and utilize accelerators effectively [3, 4]. The BlockGen algorithm calculates tree-level amplitudes and was presented in [5] with results for a C/C++ and a CUDA implementation. This proceedings presents experience and performance metrics for a Kokkos [6] implementation of the same algorithm.

1.1. Portability Frameworks

The physics computing community has begun investigating portability frameworks as a method of addressing the increasing diversity in hardware being deployed [7, 8, 9, 10]. Frameworks, like Kokkos, Sycl, and Alpaka, provide tools for writing algorithms once that can be compiled to target different architectures. For example, the same algorithm written in Kokkos can be compiled to run on Intel CPUs and GPUs, CUDA GPUs, and AMD CPUs and GPUs.

The primary kernel of BlockGen calculates the amplitude for an independent, randomly generated set of particle momenta for a given physics process, which makes it a clean target for parallelization. With this in mind, one can use the `parallel_for` methods from Kokkos to wrap the targeted kernel and run it in parallel with each kernel operating on different particle momenta. In the serial C++ version of BlockGen there are two types of data objects, static and dynamic. Static objects are filled at run start and do not change from one phase-space configuration to the next, whereas the dynamic data can change. In the Kokkos implementation, a copy of the dynamic data is needed for each parallel execution while the static data can be shared as read-only by all threads. Kokkos provides a class object, named `View`, that manage data objects so that the user can perform data copies to/from devices such as NVidia GPUs.

2. Measuring Computing Performance

The measure of computing performance for Blockgen is throughput, that is events per second. Users typically need a fixed number of events for a given physical process, making this a weak scaling problem, i.e. the more events one can process in parallel the quicker the task is completed.

The original C++ and CUDA implementations will be compared with the Kokkos implementation to show differences between native implementations and portability frameworks. To do this, we utilized CPU and GPU nodes at the Argonne Joint Laboratory, Perlmutter at National Energy Research Scientific Computing Center (NERSC), and the LXPLUS (Linux Public Login User Service) batch system at CERN for System Evaluation. The evaluation has been made with the following hardware:

- two Intel Skylake (8180) CPUs each with 28 cores (56 cores total),
- AMD EPIC 7713 CPU with 64 cores,
- NVidia A100 GPU with 6,912 CUDA cores and 432 Tensor cores,
- NVidia V100 GPU with 5,120 CUDA cores and 640 Tensor cores, and
- AMD MI100 with 120 compute units totaling 7,680 stream processors.

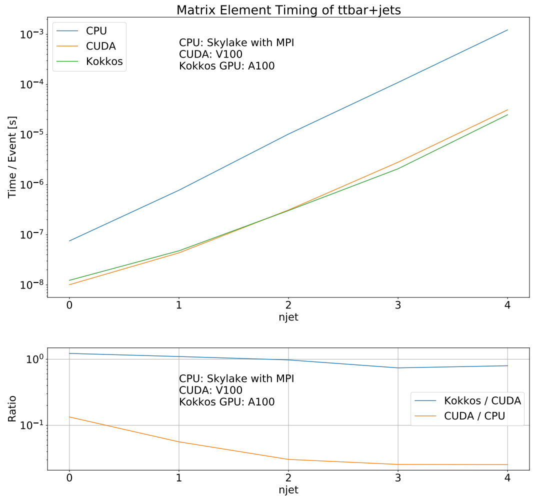

Figure 1. Native C++ running on Skylake using MPI to fill CPU, compared with native CUDA version running on V100 and Kokkos version running on NVidia A100. Time to calculate one event is shown above, with comparison ratios shown below.

We use the Intel Skylakes to compare the Kokkos version compiled with OpenMP to the serial C++ version, and the NVidia devices are used to compare Kokkos (compiled with CUDA) with the native CUDA version. While the Kokkos version compiled with OpenMP can utilize all CPU cores in a parallel manor, the serial C++ version will only utilize one CPU core. In order to make a fair comparison of throughput in the plots to follow, the throughput values for the serial C++ version were measured by running it with MPI to launch parallel processes on the CPU. The combined throughput across the parallel MPI processes is used as the throughput.

3. Results

Measurements are shown for the $t\bar{t}$+jets and W+jets processes as representative benchmarks. Both of them are the major backgrounds of the Higgs productions in the p-p collision. In general,

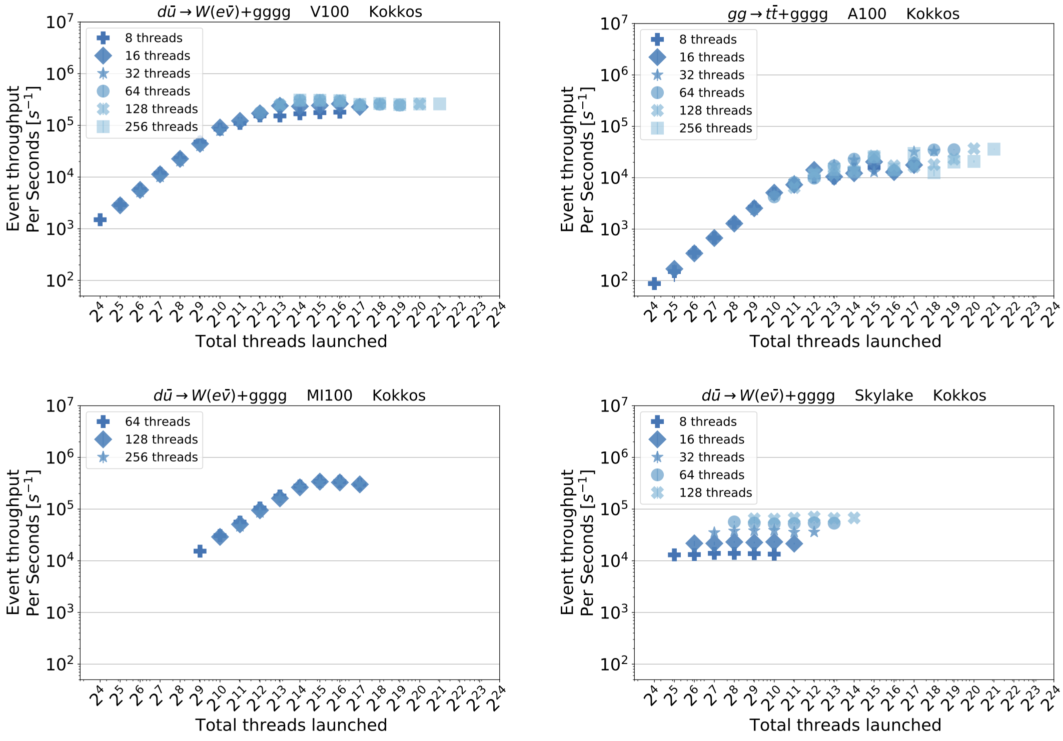

Figure 2. Event throughput as a function of the total number of parallel threads grouped by the number of "threads per block" in CUDA terminology. Results are shown for W+jets and $t\bar{t}$+jets.

$t\bar{t}$ is more compute intensive than W production, and more outgoing particles is more compute intensive than fewer.

Figure 1 shows measurements of $t\bar{t}$ production with a varying number of jets using the native C++ and CUDA version with the Kokkos version compiled for CUDA. The time to calculate a single event is shown as well as the ratios for comparison. As the number of jets increases, the compute time increases due to increasing complexity. It is important to note that the Kokkos+CUDA and the native CUDA implementations show nearly identical performance.

Figure 2 shows the event throughput versus total number of parallel threads on different devices. It demonstrates the usefulness of a portability framework to enable running on many modern architectures. The performance on GPUs plateaus as the number of threads increases showing the GPU is saturated. These plots survey the possible run configurations and are used to find settings with the peak throughput for a given process. The results for this peak value are shown in Fig. 3. The measurements are grouped by hardware on the x-axis with throughput on the vertical axis. The serial C++ version running on an Intel Skylake CPU is also included for comparison which was run in a trivially parallel way using MPI to spawn one process per CPU core. The total throughput summed across these parallel processes can be compared to the Kokkos version which uses OpenMP to launch many parallel threads. For compute intensive cases the Kokkos version outperforms the serial C++ version with MPI.

4. Summary

The Kokkos kernel performs comparably to the native CUDA and serial-C++ version on NVidia and Intel devices for more intensive computations, e.g. for more than 2 additional jets in the

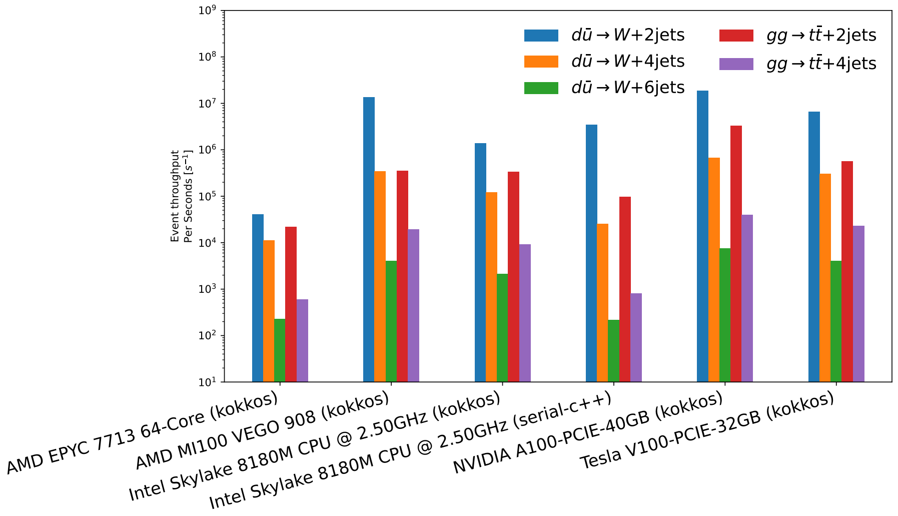

Figure 3. Hardware Comparison of Kokkos performance with CUDA backend on Nvidia GPUs, OpenMP backend on AMD and Intel CPUs and ROCM/HIP backend on AMD GPUs. Benchmarking processes $t\bar{t}$+jets and W+jets are used.

$t\bar{t}$ case. In addition, writing the kernel with Kokkos enabled utilization of AMD/Intel CPUs with OpenMP and AMD GPUs with ROCM/HIP with no change to the source code. These measurements were made using only the amplitude calculation with a simple integrator and no event output. Next steps include implementing a proper VEGAS integration framework with events written to disk and studying algorithmic performance on the different architectures. The goal being to produce a leading-order event generator that can run on most modern systems with good performance.

5. Acknowledgments

This work was supported by the DOE HEP Center for Computational Excellence at Lawrence Berkeley National Laboratory under B&R KA2401045. This work was also done as part of the offline software research and development programme of the ATLAS Collaboration, and we thank the collaboration for its support and cooperation. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, and resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. E.B. and M.K. acknowledge support from BMBF (contract 05H21MGCAB). Their research is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 456104544; 510810461.

6. References

[1] ATLAS Collaboration 2022 ATLAS Software and Computing HL-LHC Roadmap Tech. rep. CERN Geneva URL `https://cds.cern.ch/record/2802918`

[2] CMS Offline Software and Computing 2022 CMS Phase-2 Computing Model: Update Document Tech. rep. CERN Geneva URL `https://cds.cern.ch/record/2815292`

[3] The HSF Physics Event Generator WG, Valassi A, Yazgan E *et al.* 2021 *Computing Software for Big Science* **5** URL `doi.org/10.1007/s41781-021-00055-1`

[4] Benjamin D, Childers J, Hoeche S, LeCompte T and Uram T 2017 *Journal of Physics: Conference Series* **898** 072044 URL `https://dx.doi.org/10.1088/1742-6596/898/7/072044`

[5] Bothmann E, Giele W, Hoeche S, Isaacson J and Knobbe M 2021 Many-gluon tree amplitudes on modern gpus: A case study for novel event generators URL `https://arxiv.org/abs/2106.06507`

[6] Trott C R, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman D S, Ibanez D, Liber N, Madsen J, Miles J, Poliakoff D, Powell A, Rajamanickam S, Simberg M, Sunderland D, Turcksin B and Wilke J 2022 *IEEE Transactions on Parallel and Distributed Systems* **33** 805–817

[7] Kortelainen, Matti J, Kwok, Martin, (on behalf of the CMS Collaboration), Childers, Taylor, Strelchenko, Alexei and Wang, Yunsong 2021 *EPJ Web Conf.* **251** 03034 URL `https://doi.org/10.1051/epjconf/202125103034`

[8] Kortelainen, Matti J, Kwok, Martin and on behalf of the CMS Collaboration 2021 *EPJ Web Conf.* **251** 03035 URL `https://doi.org/10.1051/epjconf/202125103035`

[9] Yu, Haiwang, Dong, Zhihua, Knoepfel, Kyle, Lin, Meifeng, Viren, Brett and Yu, Kwangmin 2021 *EPJ Web Conf.* **251** 03032 URL `https://doi.org/10.1051/epjconf/202125103032`

[10] Pascuzzi V R and Goli M 2021 *CoRR* **abs/2109.01329** (*Preprint* `2109.01329`) URL `https://arxiv.org/abs/2109.01329`