Enhancing Monte Carlo proxy applications on GPUs

Forrest Shriver*, Seyong Lee†, Steven Hamilton‡, Jeffrey Vetter† and Justin Watson*
*Materials Science & Nuclear Engineering Dept.
University of Florida
†Future Technologies Group
Oak Ridge National Laboratory
‡Radiation Transport Group
Oak Ridge National Laboratory

***Abstract*—In continuous-energy Monte Carlo neutron transport simulations, a computational routine commonly known as the "cross-section lookup" has been identified as being the most computationally expensive part of these applications. A tool which is commonly used as a proxy application for these routines, named "XSBench", was created to simulate popular algorithms used in these routines on CPUs. Currently, however, as GPU-based HPC resources have become more widely available, there has been significant interest and efforts invested in moving these traditionally CPU-based simulations to GPUs. Unfortunately, the algorithms commonly used in the cross-section lookup routine were originally devised and developed for CPU-based platforms, and have seen limited study on GPUs to date. Additionally, platforms such as XSBench implement approximations which may have a negligible effect on CPUs, but may be quite impactful to performance on GPUs given the more resource-limited nature of the latter. As a result, we have created VEXS, a new tool for modeling the cross-section lookup routine which removes or at least reduces the approximations made by XSBench in order to provide a more realistic prediction of algorithm performance on GPUs. In this paper, we detail our efforts to remove and reduce these approximations, show the resulting improvement in performance prediction in comparison to a reference production code, Shift, and provide some basic profiling analysis of the resulting application.**

***Index Terms*—Monte Carlo, GPU, XSBench, VEXS**

I. Introduction

Continuous-energy Monte Carlo neutron transport simulations are a class of simulation codes used often in the field of nuclear engineering to analyze scenarios of interest, such as reactor core design and radiation dose calculations. These codes have gained a reputation of being mostly "embarrassingly parallel" [1], and are known to be the most accurate simulation method available for core design purposes. However, a specific routine which is common to all continuous-energy neutron transport simulations, the cross-section lookup routine, has been identified as composing a significant fraction of problem runtime [2]. These routines require a search for a given neutron energy be performed on a given energy grid, with the end goal being the retrieval of the upper and lower indices that a given energy is between so that the resulting cross section value can be interpolated. This routine has been found to be generally memory-bound, and often latency-bound in particular. This routine must be performed potentially many billions of times over the course of a single simulation [3], resulting in a significant portion of simulation runtime being spent waiting for data to reach compute devices. As a result of this memory-bound behavior and the critical nature of this routine to neutron transport simulations, several algorithms have been developed to reduce the time-to-solution for these routines. These algorithms have had great success reducing computation time on their original architectures, CPUs, however they have not seen as significant study or even implementation on GPUs. As GPU-based HPC resources continue to be built and developed [4], the question of whether these algorithms perform as well on GPUs as they do on CPUs is open for study and experimentation.

In an effort to broaden contributions from the scientific community external to nuclear engineering, the proxy application XSBench was written in 2013 [5]. This application has come to be widely cited for CPU-based studies of random memory accesses [6] [7] [8] [9] and is generally held to be representative of the behavior expressed by OpenMP-based continuous-energy Monte Carlo codes. However, several approximations are made in XSBench which enable ease-of-use and performance portability. While these approximations may be reasonable for CPUs, GPUs represent significantly different compute models and even resource availability [10], such that these approximations may result in false or insufficient prediction of performance improvements. As a result, we have created a new application, called VEXS (**V**ery **E**asy **XS**), which removes or reduces these approximations in order to bring performance prediction closer to those available from real-world production codes.

A. Contributions

The scientific contributions made by this work are as follows:

1) A detailed description and analysis of the computational scenario that XSBench and VEXS seek to emulate, as well as an analysis of the reference code Shift [11].
2) Introduction to implementation details of the XSBench and VEXS applications, and explanation of the motive behind removing certain approximations.

978-1-7281-5977-5/19/$31.00 ©2019 IEEE
DOI 10.1109/PMBS49563.2019.00009

3) Detailing of the algorithms of interest expressed in all three codes, as well as an initial analysis of potential algorithm behavior.
4) Comparison of performance results from XSBench and VEXS vs. Shift.
5) Comparison of profiling results from XSBench and VEXS vs. Shift.

II. Related Work

Existing work in this specific domain (continuous-energy cross section lookup algorithms) is largely limited to XSBench and VEXS, which both study the cross-section lookup problem in isolation. Other mini-apps do exist, however, which attempt to emulate the program flow commonly found in Monte Carlo neutron transport codes. This section discusses some of these programs, explains why they were not used for this research, and (in brief) explains how VEXS and XSBench are useful for this study.

A. Quicksilver

The Quicksilver mini-app is a proxy app developed and maintained by Lawrence Livermore National Laboratory (LLNL) as an open-science reflection of the workload of their in-house development Monte Carlo neutron transport code, MERCURY [12]. Quicksilver performs an approximation of the critical physical routines that would be required of a full production neutron transport code, although in a simplified manner with only some of the most common physical interactions used. This mini-app was not studied in closer detail in this paper due to it using multi-group (as opposed to continuous-energy) cross-sections, which are usually averaged in different ways over the "most accurate" continuous-energy data that is available for each energy grid. In this way, these codes avoid the need to perform lookups over potentially hundreds of thousands of points for one energy grid; however, the aforementioned averaging does come with a loss in accuracy of results. The energy ranges in question are usually subdivided into a few hundred groups at most in multi-group schemes; thus, the cross-section lookup acceleration algorithms which are the focus of this paper are not particularly applicable and indeed were not developed with multi-group applications in mind.

B. Neutral

Neutral is also a proxy app for most of the physical routines needed in a Monte Carlo neutron transport code [13], in the same vein as the Quicksilver application. One significant difference is that Neutral uses synthetic continuous-energy data, which would in theory allow it to better simulate different continuous-energy cross-section lookup acceleration algorithms. This mini-app was not used further in this study for two reasons; one is that the cross-section data used in Neutral is limited in scope (more details on the "full" computational scenario and how VEXS improves upon current codes are given in Section III). Because of this, the Neutral application itself is only designed to use this data in the case of a single energy grid being present, where virtually every scenario that we would wish to simulate would have multiple energy grids. The second reason is that some cross-section lookup acceleration schemes (not shown here but well-documented in other studies) require multiple energy grids to achieve any benefit at all; as we will study implementation of these algorithms on GPUs in the future, we elected to build our own application which was able to handle the different physical scenarios we wished to study.

C. Profugus

Profugus is a proxy app developed at Oak Ridge National Laboratory for the purpose of providing an open-science reflection of their Exnihilo code suite, which is designed for neutron transport and different reactor applications [14]. One of the available kernels under the mini-app is the MC package, which is also supposed to represent the general code layout and physical functions involved in a Monte Carlo neutron transport code. However, similar to the Quicksilver application, this package uses strictly multi-group data, and therefore the amount of data present and used in simulations will not be representative of the actual amount of data used in continuous-energy Monte Carlo simulations. For this reason, this mini-app was not used as a basis for this research.

D. VEXS/XSBench

Further detailing of the computational scenario simulated by VEXS/XSBench will be given in Section III. In brief, however, both mini-apps use continuous-energy data, resolving the issues with using the Quicksilver and Profugus mini-apps. They both also attempt to simulate "multi-material" scenarios, meaning potentially hundreds of different energy grids are present and utilized in both codes, resolving the issue with using the Neutral mini-app. Finally, both mini-apps specifically attempt to study the cross-section lookup problem in isolation, whereas all mini-apps detailed so far attempt to simulate the wider neutron transport program flow. As the cross-section lookup algorithm is known to represent a significant portion of the computational workload of continuous-energy Monte Carlo neutron transport, study of these algorithms in isolation is what both of these mini-apps are especially useful for, and thus were the targets for this paper.

III. Application details

A. Production-code scenario

Continuous-energy Monte Carlo neutron transport codes rely on the averaging of billions of random interactions provided by simulated neutrons to produce an estimate of realistic particle behavior in a given reactor configuration. In order to capture how neutrons behave in aggregate through these spaces, these codes utilize cross-sections, which are basic indicators of the probability of a neutron with a given energy interacting with a given material's member elements. In a normal interaction at least a few cross-sections must be computed independently, with many interactions requiring the computation of a few hundred cross-sections in the case of

those materials which are composed of many independent elements. As neutrons most commonly decrease in energy as they interact with, and lose energy to, the simulated environment, and can also dynamically change material during the course of the simulation, the cross-section values which represent their location and probability of interaction must be constantly recalculated and updated. As this procedure must be done for every particle, the number of values that must be stored becomes impossible to reliably cache and results in significant latency-bound behavior for many simulations. The computational challenges that are present with this routine boil down to a few key challenge points shown below:

1) The element of randomness inherent to Monte Carlo simulations predictably introduces significant issues for compiler-based optimization methods geared towards prediction of application behavior.
2) The energy grids that must be searched over are from a computational perspective very distinct from each other, with some energy grids containing only a few hundred points and some grids containing upwards of a few hundred thousand points. Additionally, most of these energy grids contain entirely different distributions of points that cannot easily be correlated to each other.
3) Many simulations that are of relevance require many hundreds of these energy grids be available for search, meaning any algorithm that hopes to provide a performance improvement that is of interest to the broader community needs to essentially handle at-scale scenarios as well as small-scale scenarios.
4) Any computed cross-sections must be discarded almost as soon as they are computed, as the interaction determined by their computation will immediately prompt a change of neutron energy and possibly a change in material.

B. XSBench

The XSBench mini-app was written to address items 1, 3 and 4 of the challenge points shown in Section III-A. Originally written in C with OpenMP instrumentation, it has very recently been extended to also support OpenACC and SYCL instrumentation as well as hand-written CUDA implementations. The fundamental unit of almost all XSBench's data structures is a struct containing an energy value as well as five cross-section values, as shown in Figure 1. The energy value as well as all cross-section values associated with that energy are randomly selected to be between zero and one, with the energy value being the quantity of importance (as this will be the value search methods will compare against) and the cross-section values only being used to simulate memory consumption for Monte Carlo simulations as well as provide a small amount of computational work that must be done. The struct shown in Figure 1 is used to build uniformly-sized sorted grids of 11,303 points; this model of placing data in memory is best described as 'Array-of-Structs'.

This approach works well for CPUs, however there are a few key approximations made which are not accurate to real-life scenarios. One such assumption is the use of uniformly-sized energy grids, which as stated in item 2 of the challenge points shown in Section III-A, is not close to actual Monte Carlo simulations other than being an average of the number of points across all grids. The reason for this approximation was ease of programming and ease of explanation, and the assertion that the in-memory size of data is not as important for CPUs as assuring that memory accesses are random across enough data to produce latency-bound characteristics. Another approximation made is that energy points are stored in terms of lethargy (all energy points are stored in the domain (0,1]), which is not implemented in all production codes (actual energy values vary in the domain $(0{,}20 \times 10^6]$) and removes the need for some complexity which would otherwise be required. Finally, another approximation is made in that the Array-of-Structs implementation is inherently inefficient for rapid searching, with a Struct-of-Arrays theoretically having significantly better performance for the cross-section lookup routine due to better spatial locality and absence of unused data [16]. While a CPU-based execution model may not be sensitive to these approximations, GPUs with significantly more limited resources may be quite sensitive to differences in memory and computational complexity when it comes to performance modeling.

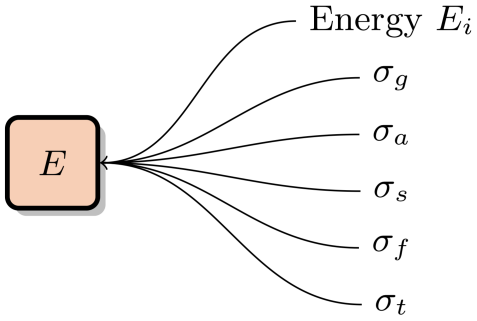

Fig. 1. Diagram of fundamental unit in most XSBench data structures [15].

C. VEXS

The VEXS application was inspired by XSBench, however it has been engineered to remove or reduce the number of approximations made by the XSBench application. The first approximation that was removed was the use of uniformly-sized energy grids. To accomplish this, we use energy points taken from actual nuclear datasets, without the associated cross-sections that contain export-controlled information, and use these points to form differently-sized energy grids that synthetic data is assembled on. This transforms the in-memory scenario from one where all array sizes remain constant, to one where array sizes vary similarly to actual production-code scenarios. Diagrams of the datasets used in VEXS and XSBench are shown in Figures 2 and 3, respectively.

The changes shown in Figure 3 also include using energy levels in the domain $(0{,}20 \times 10^6]$, prompting the removal of another approximation where the computational complexity

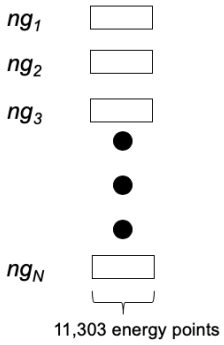

Fig. 2. Diagram of energy grid datasets as they are stored in XSBench [17]

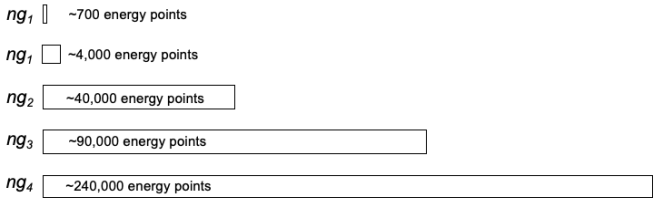

Fig. 3. Diagram of energy grid datasets as they are stored in VEXS [17]

required by some algorithms is fully restored. Lastly, the data structures used in VEXS were reorganized from the form shown in Figure 1 to a Struct-of-Arrays format, which is closer to that implemented by the Shift reference application. Ideally, these primarily in-memory changes are significant enough that both the memory and compute models will exhibit performance characteristics closer to those produced by Shift.

D. Shift

The Shift production code is a continuous-energy Monte Carlo neutron transport code developed at Oak Ridge National Laboratory for the purpose of performing at-scale neutron transport simulations on HPC systems comprised of CPUs and, if available, GPUs [11] [18]. This objective requires the inclusion of a number of different types of physics and computation routines, with development still ongoing to implement needed physics and produce code implementations that can run at-scale on current platforms (Summit) and upcoming platforms (Frontier).

Some differences between Shift and both mini-apps that are impossible to capture, due to Shift's nature as a production Monte Carlo neutron transport code, are the following:

1) Shift uses actual cross-section data in its calculations, which will be much more varied and physics-dependent, prompting an even larger amount of variance in memory usage than that simulated by the VEXS and XSBench mini-apps. Additionally, this cross-section data is commonly export controlled, and inclusion of this data in mini-app design would immediately limit the usefulness of these mini-apps to those external to the nuclear engineering field.
2) Shift requires a significantly larger amount of data be computed and used compared to either mini-app, with neutron geometry states and additional program-specific data being required in addition to basic particle energy information.
3) Because of items 1 and 2, there is a significant amount of extra memory indirection in the Shift application compared to either mini-app. This stems from both mini-apps only simulating the cross-section lookup routine, while the diverse set of physics that must be included in Shift necessitates small subsets of particles being active at different times, resulting overall in very non-contiguous device memory access patterns.

Despite these substantial differences, it is nevertheless interesting to analyze whether the approximations used by XSBench produce trends that are similar to Shift, and whether the improvements made by VEXS have any significant effects in improving performance prediction.

IV. Algorithm Details

Given the critical and latency-bound nature of the cross-section lookup routine, several algorithms have been developed by the nuclear community to reduce the time-to-solution for this routine. XSBench implements a total of three algorithms used for the cross-section lookup, two of which are acceleration algorithms and one of which is essentially a baseline. Shift, being a production code, only implements the baseline algorithm and one acceleration algorithm. The results shown today will therefore be based on the performance analysis of these two algorithms: a baseline binary search, and a hash-accelerated search. The following section details each of these algorithms' computational details and memory requirements.

A. Baseline binary search

The baseline binary search represents the absolute minimum in terms of memory requirements of current algorithms, as it relies upon searches being performed on each material member independently and does not require additional data structures to perform. The memory requirement goes as follows:

$$M_{member} = \sum_{i=0}^{N} n_i * 6 * s(float) \tag{1}$$

where M_{member} is the total memory consumed in bytes, n_i is the number of energy points in the i'th material member, and N is the total number of members in a given material. The multiplicative factor of 6 is derived from the fact that each energy point consists of a single energy value and five cross section values, and the $s(float)$ factor denotes the size of a floating-point value on the target system.

This algorithm is expected to use on the order of hundreds of megabytes, and is thus not expected to be heavily bandwidth bound but is expected to be heavily latency bound. It is also expected to be the most divergent of any proposed algorithm (as most algorithms and approaches are currently focused on

either removing or reducing the number of search iterations required).

One theoretical argument that can be made for the use of realistically-sized datasets is a basic analysis using Big O notation. For a binary search, it is widely known that the search will generally go as $\mathcal{O}(log_2(n))$ with n being input size. Thus, for the case illustrated in Figures 2 and 3, where N is 34, Big O time would go approximately as $\sum_{i=1}^{N} \mathcal{O}(\log n_i)$, where N is the total number of energy grids in the problem. Using this notation, XSBenchs method of representing data results in a value of approximately 458, while using realistically-sized grids results in a value of approximately 71. This is not a formal mathematical proof that using uniformly-sized grids will give false performance results, however it is an indication that the same algorithm may give significantly different performance results depending on approximations used. This is relevant if we wish to create a mini-app that captures performance metrics of the target codes well, as overestimation of the baseline computational load may result in false prediction of performance improvement for new optimizations or algorithms developed using this mini-app. This was our motivation for implementing realistically-sized energy grids in the VEXS application.

B. Hash-accelerated search

The hash-accelerated search was originally proposed for CPU-based Monte Carlo simulations [2], and is the only adopted acceleration algorithm in the Shift code. In this physics-based algorithm, before the simulation begins a pre-computed index table *ugrid*, which is uniform in the logarithmic energy space $ln(E)$ between the minimum and maximum energies of the simulation, is generated for each nuclide grid. Then, during the actual simulation, a given energy is then run through a basic hashing computation, shown in Eq. 2, which produces an index used to retrieve lower and upper index values from *ugrid*. The index values pulled from this grid therefore form a "reduced window" for each material member, where the hashed energy point is guaranteed to be located in between these two indices. In this way, the search space is reduced from potentially hundreds of thousands of energy points to only a small (on the order of ten or less) energy points for each material member, thus resulting in significantly reduced time to solution on CPU-based systems. The overall algorithm is illustrated in Figure 4.

$$u = 1 + \lfloor du * (ln(E) - u_{min}) \rfloor \tag{2}$$

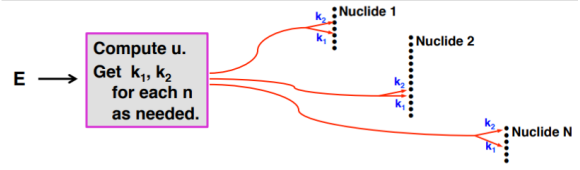

Fig. 4. Overview of the hash-accelerated search algorithm [2].

The memory requirement for the hash-accelerated search goes as:

$$M_{hash} = (G + 1) * N * s(int) + M_{member} \tag{3}$$

where M_{hash} is the memory required to store the index table *ugrid*, G the variable (user-defined) number of hash bins, $s(int)$ denotes the size of an integer value on the target system, and M_{member} is the memory requirements introduced by 1.

One implementation detail of the hash-accelerated search is that, with search windows of potentially ten or fewer points, there is a conceivable benefit to using a linear search over a binary search to complete the index search within the reduced search window in terms of increased cache locality. While this has been studied and found to not be substantial on CPU-based systems, GPU-based systems which need to account for application divergence may receive a much more significant benefit from using a linear over a binary search. Both implementations of this algorithm are present in the Shift and VEXS applications, and a linear search was implemented in XSBench in order to provide a fair comparison.

Even for larger values of G such as 16,000+ the resulting memory requirement indicated by (3) only results in a memory increase on the order of tens of megabytes for normal problem sizes, making it intuitively very favorable for implementation on GPUs as these devices are normally limited by memory in the problem sizes they can represent. Of particular note for this algorithm is that the logarithm required by the original algorithm shown in Eq. 2 (and as it is implemented in Shift) involves the computation of a logarithm which XSBench, due to the approximations made, removes due to it being unnecessary. While one logarithm computation may not feasibly be impactful on a single or set of cores, on GPUs where each thread has much more limited resource availability the presence or absence of this logarithm feasibly would play a significant role in relative performance trends. This was our motivation for implementing the algorithm's full complexity in VEXS using realistically-depicted energy points.

V. Results & Analysis

XSBench and VEXS approximate the random behavior of simulated particles in production codes such as Shift by both selecting random energies and also randomly selecting the material for a lookup simulation according to a probability distribution. This material selection component has been identified as being very computationally important, as there are certain materials known as 'fuel' materials which can, in real-life scenarios, evolve through time from being composed of only a few member elements ('fresh' scenarios) to a very large number of member elements ('depleted' scenarios), presenting very different performance behavior. Therefore, the following results are split into three sections: the first is a comparison against Shift unit tests designed to measure cross-section lookup performance. The second section is a comparison against a full simulation problem run through Shift, in order to capture application behavior that might otherwise be lost

when using unit tests. The third section is profiling of certain algorithm implementations on the depleted scenario on both GPUs, in order to analyze how well VEXS and XSBench model the target application in terms of achieved occupancy and branch efficiency on multiple platforms. In the case of the first two scenarios, 'Acceleration' is defined as:

$$Acceleration = \frac{throughput_{hash-accelerated}}{throughput_{baseline}} \quad (4)$$

where $throughput_{baseline}$ is the throughput from the full binary search being run over all material members, and $throughput_{hash-accelerated}$ is the throughput from the hash-accelerated search being run over all material members. The hash-accelerated search was either implemented using a hash-accelerated binary search, indicated by a dotted line, or can be implemented using a hash-accelerated linear search, indicated by a solid line. Finally, in all results green lines represent those taken from the Shift reference, while black lines represent those taken from XSBench and blue lines represent those taken from VEXS.

Results shown were all collected from systems provided by the Oak Ridge Leadership Computing Facility. The systems used were the Summitdev and Summit [19] systems, both of which utilize on-node P100s and V100s respectively for application acceleration. For Summitdev, CUDA 9.2.148 was used; for Summit, CUDA 9.1.85 was used. All results were collected from single-node jobs with exclusive node access. All three codes were run with 256 threads per block, with enough blocks used to completely fill up the target devices; further details on parameters used can be found in Appendix B. Error bars are shown on all results for the first two sections, while the third section does not possess error bars as these results were collected from the profiling tool nvprof.

It must be noted that the total runtime for Shift was found to be on the order of minutes for even simple cases (which can be entirely attributed to its nature as a production code) while the runtime of VEXS was on the order of seconds and XSBench was found to be marginally faster. The reason for this is because Shift is a full production code with a significant computational workload required in addition to the cross-section lookup; VEXS is an isolation of just the cross-section lookup procedure which still requires a large amount of data be read from disk, while XSBench is the fastest by being able to perform all data generation in-memory. To account for these discrepancies, the method shown in Eq. 4 was used to generate the following performance trends, as the primary use of these mini-apps is in comparison of different algorithms which have been implemented within each mini-app. In this way, the relative performance characteristics of studied algorithms can be captured without needing to compare and account for the inherent differences in memory consumption and computational workload among these three codes.

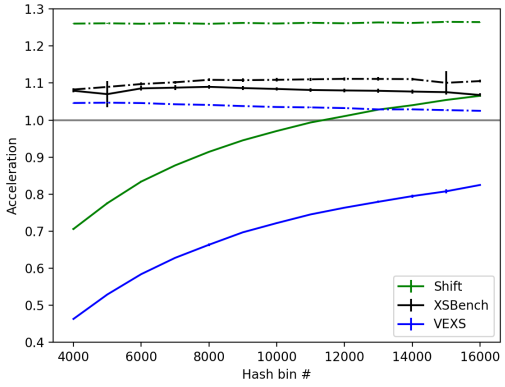

Fig. 5. XSBench, VEXS, and Shift unit test simulations of a fresh scenario on a P100 GPU [17].

A. Comparison against Shift unit tests

Shown in Figure 5 is a comparison among Shift unit tests, XSBench, and VEXS for a fresh scenario on a P100 GPU. XSBench appears to predict that the hash-accelerated binary search and hash-accelerated linear search will be very close to each other in terms of performance, and appear to be agnostic to the number of hash bins used in the calculation. This is in direct contrast to the Shift unit tests, which indicate that there will be a significant performance difference between the two implementations and that the hash-accelerated linear search seems to show a heavy dependence on the number of hash bins used. By comparison, VEXS correctly captures the hash bin-sensitive behavior for the hash-accelerated binary search, the hash bin dependency shown in the reference for the hash-accelerated linear search, and the performance degradation at lower numbers of hash bins. It does appear unable to capture the reference's achieved performance above unity for higher numbers of hash bins, however it does seem to capture very well the asymptoting behavior shown by the two curves in the reference, where the hash-accelerated linear search achieved increasing performance with increasing numbers of hash bins but is not able to achieve the performance shown by the hash-accelerated binary search. The result is a significant gap between the two algorithm implementations at the 16,000-bin mark which appears to be very similar to that produced by Shift.

Shown in Figure 6 is a comparison among Shift unit tests, XSBench, and VEXS for a depleted scenario on a P100 GPU. XSBench again appears to predict that the two algorithm implementations will be very close in terms of performance, although there now appears to be a dependence upon the number of hash bins used for both algorithms. This is still not completely in line with the Shift reference, where the hash-accelerated binary search appears to be agnostic to the number of hash bins used, and there is a significant discrepancy between the two algorithm implementations. By comparison, VEXS correctly captures the agnostic behavior of the hash-accelerated binary search and the discrepancy in performance

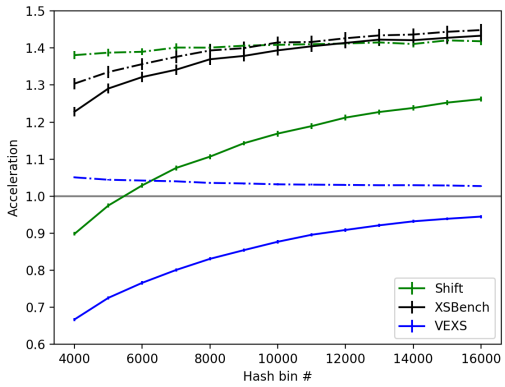

Fig. 6. XSBench, VEXS, and Shift unit test simulations of a depleted scenario on a P100 GPU [17].

between the two algorithm implementations. Again VEXS captures performance degradation at low hash bin numbers without being able to capture a rise above unity as shown in the reference, however it does capture the relative asymptoting behavior of the hash-accelerated linear search.

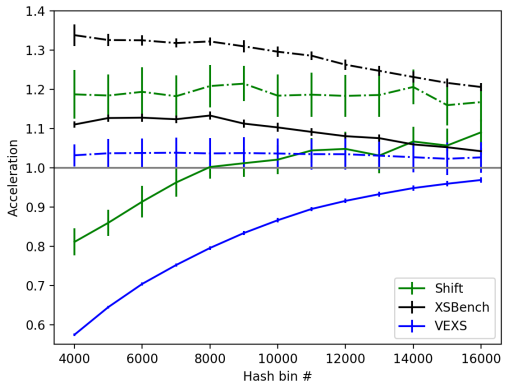

Fig. 7. XSBench, VEXS, and Shift unit test simulations of a fresh scenario on a V100 GPU [17].

Shown in Figure 7 is a comparison among Shift unit tests, XSBench, and VEXS for a fresh scenario on a V100 GPU. XSBench does appear to predict that there will be some discrepancy between the hash-accelerated linear search and the hash-accelerated binary search, which matches with Shift results, however it also predicts a dependence on the number of hash bins for both implementations which disagrees with the reference (the hash-accelerated binary search remains agnostic to the number of hash bins). It also predicts that there will be some performance degradation as the number of hash bins is increased, which is directly counter to the behavior expected from the algorithm and also counter to results collected from Shift. By comparison, VEXS does not indicate a degradation of performance with increasing number of hash bins for either algorithm implementation, and also appears to correctly capture the hash bin-agnostic behavior of the hash-accelerated binary search correctly and the asymptoting behavior between the two algorithms shown by the reference.

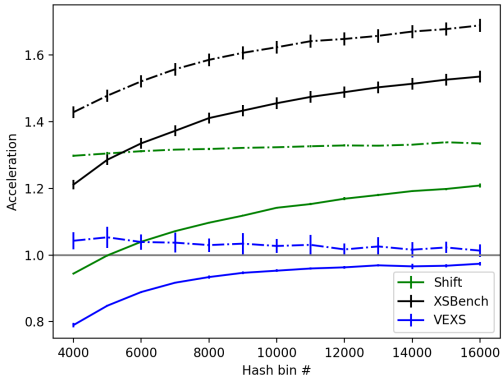

Fig. 8. XSBench, VEXS, and Shift unit test simulations of a depleted scenario on a V100 GPU [17].

Shown in Figure 8 is a comparison among Shift unit tests, XSBench, and VEXS for a depleted scenario on a V100 GPU. XSBench does not show performance degradation with increasing numbers of hash bins as seen in Figure 7, and does show a discrepancy between the two algorithm implementations contrary to those results obtained from a P100 in Figure 6. However the hash-accelerated binary search does again show a dependence on hash bin number that is not shown by the reference, and in both cases the relative difference between the two algorithm implementations stays roughly the same, which is contrary to the asymptoting behavior shown in all other results. By comparison, VEXS correctly captures the hash bin-agnostic behavior of the hash-accelerated binary search and captures the general asymptoting behavior shown in all other results, although it does asymptote too quickly in contrast to the reference.

B. Comparison against full Shift problem

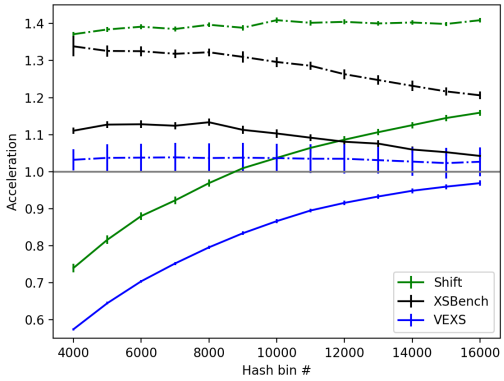

Fig. 9. XSBench, VEXS, and full Shift simulations of a fresh scenario on a V100 GPU [17].

Shown in Figure 9 is a comparison among a full Shift calculation, XSBench, and VEXS for a fresh scenario on a V100 GPU. These results are largely in line with those shown by the unit test in Figure 7, although one interesting behavior exhibited by a full application run is that there does appear to be a significantly wider discrepancy between the hash-accelerated linear and hash-accelerated binary searches, even at 16,000 hash bins. This discrepancy does not appear to be captured successfully by VEXS, although the asymptoting behavior previously noted as being common across all unit tests does appear to be captured well.

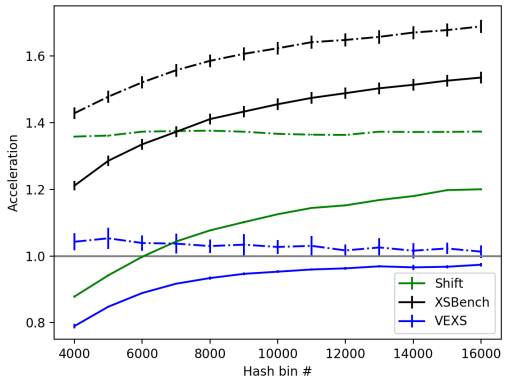

Fig. 10. XSBench, VEXS, and full Shift simulations of a depleted scenario on a V100 GPU [17].

Shown in Figure 10 is a comparison among a full Shift calculation, XSBench, and VEXS for a depleted scenario on a V100 GPU. These results are again largely in line with those shown by the unit test in Figure 8, although the discrepancy between the two algorithm implementations is not as wide as that shown in Figure 9. Again VEXS does not capture this discrepancy very well, and again captures the general asymptoting trend for the hash-accelerated linear search while still not closely resembling the trend shown by the reference.

C. Profiling

Shown in Figures 11 and 12 is a comparison of the achieved occupancy using a Shift unit test, XSBench, and VEXS when running the baseline binary search and the hash-accelerated linear search for a depleted scenario on a P100 and V100 GPU, respectively. XSBench predicts a higher occupancy than expected given the divergent nature of the application, around sixty percent on both devices. By comparison, VEXS predicts an achieved occupancy of approximately forty percent on the P100 GPU and thirty percent on the V100 GPU. This is closer in both cases to the Shift reference, which is around twenty percent on both devices. Interestingly, on the P100 all three applications show a noticeable dip in achieved occupancy when switching to use of the hash-accelerated linear search (hash bin = 4,000), but on the V100 VEXS appears to be largely flat with increasing numbers of hash bins while

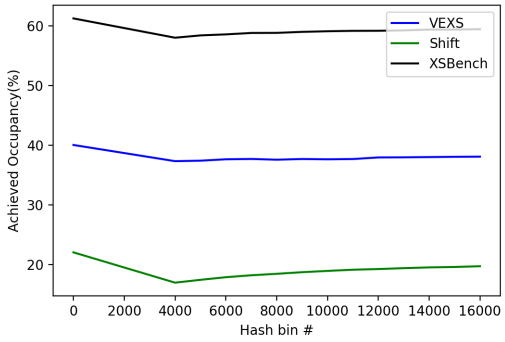

Fig. 11. Comparison of the achieved occupancy for XSBench, VEXS, and Shift unit tests for the baseline binary search (hash bins = 0) and the hash-accelerated linear search (hash bins = 4,000 - 16,000) on a P100 GPU [17].

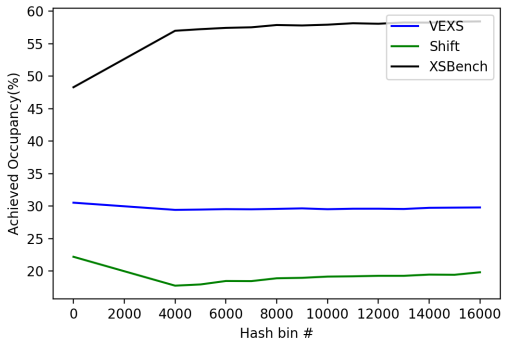

Fig. 12. Comparison of the achieved occupancy for XSBench, VEXS, and Shift unit tests for the baseline binary search (hash bins = 0) and the hash-accelerated linear search (hash bins = 4,000 - 16,000) on a V100 GPU [17].

XSBench actually predicts that there will be an increase in achieved occupancy for the same point.

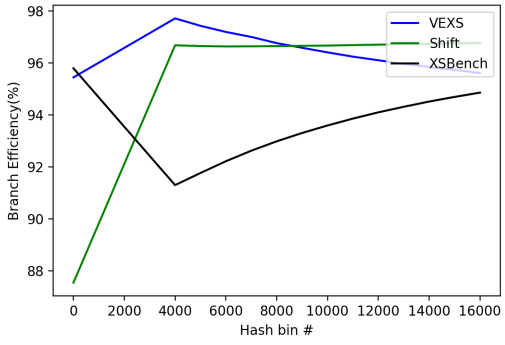

Fig. 13. Comparison of the branch efficiency for XSBench, VEXS, and Shift unit tests for the baseline binary search (hash bins = 0) and the hash-accelerated linear search (hash bins = 4,000 - 16,000) on a P100 GPU [17].

Shown in Figures 13 and 14 is a comparison of the calculated branch efficiency using a Shift unit test, XSBench, and VEXS when running the baseline binary search and the

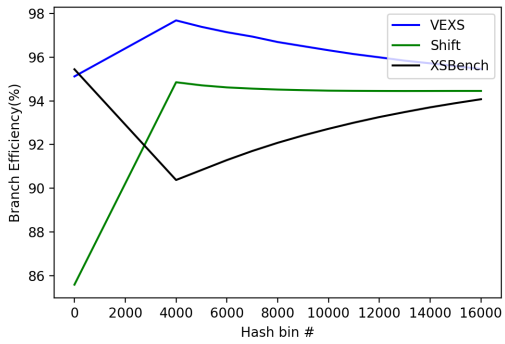

Fig. 14. Comparison of the branch efficiency for XSBench, VEXS, and Shift unit tests for the baseline binary search (hash bins = 0) and the hash-accelerated linear search (hash bins = 4,000 - 16,000) on a V100 GPU [17].

hash-accelerated linear search for a depleted scenario on a P100 and V100 GPU, respectively. For both cases VEXS correctly captures a significant increase in branch efficiency when switching from the baseline binary search to the hash-accelerated linear search, although in both cases it then exhibits decreasing branch efficiency with increasing hash bin number, which only matches the reference on the V100 GPU, with the P100 hash-accelerated linear search remaining largely agnostic to the number of hash bins used. By contrast, on both devices XSBench actually predicts a decrease in branch efficiency when switching to the hash-accelerated linear search and then predicts an increase in branch efficiency as the number of hash bins is increased, which is in direct contrast to the reference in all cases.

Overall, VEXS does appear to more closely capture reference behavior in terms of achieved occupancy and branch efficiency on both devices. This is relevant to performance analysis on GPUs, as achieved occupancy is an overall measure of actual application efficiency when all features (including latency-hiding efforts by the GPU) are taken into account, branch efficiency is a good overall measure of how divergent an application is, and Monte Carlo codes in particular are known to be both latency-bound and very divergent.

VI. Summary and Conclusions

In this paper, we have shown the necessity of taking into account the memory model when designing a new Monte Carlo mini-app for GPUs, VEXS. We first began by introducing the computational scenario the mini-app attempts to describe, and detailed the similarities and differences between this new mini-app and XSBench, an already-established Monte Carlo mini-app which uses several approximations for the memory model. We then described the reference code Shift, and explained algorithm and implementation details in each code for the purposes of justifying the additional work that went into VEXS. We then analyzed the trends exhibited by each code for quantities of interest (throughput ratios) and profiling information (achieved occupancy, branch efficiency). We showed that without taking into account memory details particular to the Shift reference code, mini-apps which use approximations for these in-memory details (such as XSBench) may produce trends that are very different from those exhibited by actual production codes. We also showed that when using a mini-app that attempts to take these details into account (VEXS) the resulting trends are significantly closer to those in Shift.

It must be noted that the primary value of these mini-apps is in comparison of the relative performance of different algorithms and algorithm implementations; that is to say, the behavior of different algorithms relative to each other can be significantly more important than raw numerical results. As the binary search can always be taken to be a baseline for continuous-energy Monte Carlo neutron transport applications, the results shown in Figures 5 - 10 initially do not look promising in either mini-app, as XSBench over-predicts the acceleration produced by these algorithms and VEXS in many cases appears to under-predict the acceleration. However, the authors would like to note that the trends between the hash-accelerated search with binary completion and the hash-accelerated search with linear completion implemented in VEXS appear to much more closely follow those produced by Shift (asymptoting behavior) as compared to XSBench. Most notably, for some cases XSBench predicts a decrease in the relative performance of both implementations, or at least an equal performance between the two implementations which is not reflected in Shift. Therefore, the primary value of this work is in producing a mini-app that is able to successfully show relative trends between new acceleration algorithms and algorithms that already exist and have been implemented. For example, if a user desires to test a new algorithm in VEXS and produces performance that is significantly above the performance curves generated by the hash-accelerated search, then it is possible that implementation of these algorithms in Shift or other production codes will produce results superior to current acceleration algorithm implementations. In this way, VEXS is meant to serve as a method to test new acceleration algorithms relative to other acceleration algorithms, and is not necessarily meant to provide a fair comparison between these acceleration algorithms and baseline implementations.

VII. Future Work

Future efforts to develop new algorithms and optimizations for the cross-section lookup routine, if using a mini-app as a testbed, may need to incorporate memory considerations similar to those employed by VEXS in order to get accurate results. If new algorithms are developed without taking these memory considerations into account, false performance benefits may arise that would otherwise not be present. Mini-apps such as XSBench and VEXS will likely need to implement further engineering in order to account for different memory models and implementations. A study of what specific in-memory alterations led to the improved performance trends shown here is underway. In the future, the authors of VEXS would like to further incorporate the in-memory details of Monte Carlo

codes into the mini-app so as to bring performance trends further in line with those exhibited by production codes.

VIII. Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy.

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research was supported in part by an appointment to the Oak Ridge National Laboratory ASTRO Program, sponsored by the U.S. Department of Energy and administered by the Oak Ridge Institute for Science and Education.

This material is based upon work supported under an Integrated University Program Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Department of Energy Office of Nuclear Energy.

References

[1] P. K. Romano, "Parallel algorithms for monte carlo particle transport simulation on exascale computing architectures," Ph.D. dissertation.

[2] F. Brown, "New hash-based energy lookup algorithm for monte carlo codes," in *Trans Am Nucl Soc 111, 659-662, 2014*, vol. 111, Anaheim, California, November 2014, pp. 659–662.

[3] J.E. Hoogenboom, W.R. Martin, B. Petrovic, "Monte carlo performance benchmark for detailed power density calculation in a full size reactor core," in *OECD/NEA (2010)*.

[4] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, March 2010.

[5] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

[6] T. Scudiero, "Monte carlo neutron transport: simulating nuclear reactions one neutron at a time," gPU Technology Conference (GTC) 2014, Nvidia.

[7] Y. Luo,X. Wen, K. Yoshii, S. Ogrenci-Memik, G. Memik,H. Finkel, F. Cappello, "Evaluating irregular memory access on opencl fpga platforms: A case study with xsbench," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*.

[8] N. Sultana, A. Skjellum, P. Bangalore, I. Laguna, K. Mohror, "Understanding the usage of mpi in exascale proxy applications," in *2018 SC Conference Workshop*.

[9] Amd compute apps. [Online]. Available: https://github.com/AMDComputeLibraries/ComputeApps

[10] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[11] T. Pandya and S. Johnson and G. Davidson and S. Hamilton and T. Evans and A. Godfrey, "Implementation, capabilities, and benchmarking of Shift, a massively parallel Monte Carlo radiation transport code," *Journal of Computational Physics*, vol. 308, pp. 239–272, 2016.

[12] David Richards, Ryan Bleile, Patrick Brantley, Shawn Dawson, Scott McKinley, Matthew O'Brien, "Quicksilver: A proxy app for the monte carlo transport code mercury," in *Workshop on Representative Applications*.

[13] Matt Martineau, Simon McIntosh-Smith, "Exploring on-node parallelism with neutral, a monte carlo neutral particle transport mini-app," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[14] Seth Johnson, "A first strike at an openacc c++ monte carlo code!" in *GTC2019*.

[15] Xsbench: The monte carlo macroscopic cross section lookup benchmark. [Online]. Available: https://github.com/ANL-CESAR/XSBench

[16] M. Hagelin, "Optimizing memory management with object-local heaps," Master's thesis, Uppsala University, Department of Information Technology, 2015.

[17] Forrest Shriver, Seyong Lee, Steven Hamilton, Justin Watson, Jeffrey Vetter, "Most data, plotting scripts and figures for "enhancing monte carlo proxy applications on gpus"."

[18] S.P. Hamilton and T.M. Evans, "Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code," *Annals of Nuclear Energy*, vol. 128, pp. 236–247, 2019.

[19] Summit system user guide. [Online]. Available: https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/

Appendix
Availability of Material

The VEXS mini-app is available openly at https://github.com/fshriver/VEXS. All figures, plotting scripts, and most of the data that was used to generate them are available openly under the CC-BY license [17].

Appendix
Reproducibility

All results shown in this paper were taken from the Summit-dev (P100 GPU) and Summit (V100 GPU) systems maintained by the Oak Ridge Leadership Computing Facility (OLCF). To reproduce these exact results, one should ideally use an allocation on these machines obtained from the OLCF; similar results are likely but not guaranteed to be obtained using similar GPUs found in other clusters.

The following steps may be taken in order to reproduce most of the results shown in this paper:

A. VEXS

1) Main: The latest public-facing version of the VEXS codebase (which can be used to reproduce the results shown in this paper) can be found here: https://github.com/fshriver/VEXS.

Once the VEXS repository is cloned into a location where the user has appropriate access, the user should first follow the Quickstart guide located on the main page to generate initial binary data files. They should then move to the directories of those kernels they wish to collect data on (for the case of this paper, those located under ***packages/GPU/CUDA/xsbench-like/naive/struct-of-arrays***, ***packages/GPU/CUDA/xsbench-like/hash/struct-of-arrays/binary-completion*** and ***packages/GPU/CUDA/xsbench-like/hash/struct-of-arrays/linear-completion***). The user should make sure the appropriate flags for their architecture and configuration are set in the Makefile; a full explanation is given in the VEXS wiki under the user manual, however good ones to watch out for are the SM flag (default sm_60 for a P100, change to sm_70 if operating on a V100) and the cache flag (default cg, set to ca for the purposes of collecting data similar to this paper).

2) Acceleration results: Once the above is done, the user should write scripts that are set to parse and filter program output; the keyword to parse for will be 'Rate ='. The user should filter on this expression, and should write their scripts to collect the number that follows. This is the basis of sections one and two of the above paper.

To collect exact results, the user should run the ***packages/GPU/CUDA/xsbench-like/naive/struct-of-arrays*** kernel with the following options:

```
./VEXS -i detailed_model_[SCENARIO]_core
-b --gputhreads 256 --gpublocks [NUMBER]
```

where [SCENARIO] is toggled to either fresh or depleted by the user depending on the scenario in the above paper they wish to capture, and [NUMBER] is set to 16384 if [SCENARIO] = fresh and 4096 if [SCENARIO] = depleted.

The user should run similar code in the ***binary-completion*** and ***linear-completion*** directories under ***packages/GPU/CUDA/xsbench-like/hash/struct-of-arrays***, however should add the following parameters:

```
-hashbins [4000 - 1000 - 16000]
```

where [4000 - 1000 - 16000] indicates that the scripts should be written in such a way as to run VEXS with this parameter varied from 4000 to 16000 in increments of 1000. Each test should be repeated ten times to collect statistics.

3) Profiling results: To collect performance metrics, the user should run the kernels under ***packages/GPU/CUDA/xsbench-like/naive/struct-of-arrays/*** and ***packages/GPU/CUDA/xsbench-like/hash/struct-of-arrays/linear-completion*** like so:

```
./nvprof --kernels xs_lookup --metrics
achieved_occupancy,branch_efficiency
--log-file metrics.txt --csv -f
./VEXS -i detailed_model_[SCENARIO]_core
-b --gputhreads 256 --gpublocks [NUMBER]
--hashbins [4000 - 1000 - 16000]
```

where the above parameters are either present or absent as detailed above and vary depending on the kernel that is being profiled.

B. XSBench

1) Main: To latest version of XSBench can be found here: https://github.com/ANL-CESAR/XSBench. Once this repository has been cloned, the user should only need to work in the cuda/ directory under the top level of the repository. The user should again make sure that the appropriate flags are set in the Makefile for their architecture. The user should also assure that in the Simulation.cu file, they edit the *nthreads* variable under the *run_event_based_simulation_baseline* function to equal 256. They should also implement their own linear search in the XSBench codebase under those sections targeted towards use of the hash-accelerated search, as XSBench does not natively possess a linear search.

2) Acceleration results: Once the above is done, the user should write scripts that are set to parse and filter program output; the keyword to parse for will be Lookups/s:. The user should filter on this expression, and should write their scripts to collect the number that follows. This is the basis of sections one and two of the above paper. Once this has been implemented, the user should write scripts to run the following code:

```
./XSBench -k 0 -m event -G [TYPE] -s
[SIZE] -l [NUMBER] -h [4000 - 1000 16000]
```

where [TYPE] is set to either nuclide or hash depending on the results that the user wishes to be obtained, [SIZE] is set to either small or large depending on whether the user wishes to collect fresh (small) or depleted (large) scenario results, [SIZE] is set to either 1048576 if the user chose [SIZE] = large or 4194304 if the user chose [SIZE] = small and the last parameter, -h, is passed only if [TYPE] is set to hash and takes on a range from 4000 to 16000 in increments of 1000, similar to the range used for VEXS. Depending on whether the user wishes to collect hash-accelerated linear or hash-accelerated binary search results, they should comment out either the binary search or the linear search in the relevant parts of the codebase.

3) Profiling: To collect the performance metrics, the user should run the nvprof visual profiler on the XSBench application like so:

```
./nvprof --kernels
xs_lookup_kernel_baseline --metrics
achieved_occupancy,branch_efficiency
--log-file metrics.txt --csv -f
./XSBench -k 0 -m event -G [TYPE] -s
[SIZE] -l [NUMBER] -h [4000 - 1000 16000]
```

where the above parameters are either present or absent as detailed above and vary depending on the kernel that is being profiled.

C. Shift

Unfortunately Shift is an export-controlled Monte Carlo code, developed at ORNL, and so cannot be navigated to, downloaded or used without appropriate permissions by the U.S. government.