HPCOR 2015 Application White Paper
Denovo: A Radiation Transport Code for Nuclear Applications
Wayne Joubert, ORNL

• *How many developers are involved and how is the development structured? (co-located team, post-docs, community contributions, etc.):* 5-6 core developers co-located, 5-10 additional contributors on- and off-site.

• *Language:* C++ with Fortran for some kernels; Python driver option.

• *Lines of code:* ~50K

• *Primary methods:* discrete ordinates solver for steady-state Boltzmann transport equation using Krylov solvers and eigensolvers with the KBA sweep algorithm.

• *Types of problems/domains/science application problems:* reactor core analysis (neutronics), radiation shielding, nuclear forensics and radiation detection.

• *Scale of resources commonly used for production runs:* up to full-system runs on leadership-class systems for durations of several hours.

• *Supercomputers regularly used:* ORNL Titan

• *Libraries/tools for prototyping:* N/A

• *Libraries/tools for production science campaigns:* Libraries: MPI, Trilinos, BLAS/LAPACK, SILO/HDF5, CUDA. Other: Python, VisIT, TriBITS, CMake, CDash.

• *Describe efforts to develop code (application, library, etc.) portable across diverse architectures:* (1) a new algorithm was implemented natively in Denovo to expose parallelism in energy group for use on accelerators—this was immediately portable across all architectures via MPI; (2) the 3D sweep algorithm, the primary code hot spot, was rewritten to better control the data access patterns for effective use of accelerators. Access to CUDA is hidden behind façade classes or functions to increase portability and facilitate future porting to OpenACC and/or OpenMP 4/5.

• *Where were the abstractions?* Abstractions for the sweep code to access the accelerator using CUDA are stored in separate files in the CUDA sweeper directory. These abstractions allow the same code to execute on either CPU or accelerator. Abstractions include: access to the hardware thread hierarchy; thread synchronization; memory management on host and accelerator; asynchronous data transfer to/from accelerator in CUDA streams; access to accelerator scratchpad memory; manipulation of arrays that are mirrored between host and accelerator. Higher level abstractions concern: mapping the science problem to the compute nodes, managing on-node and off-node communication, and manipulating multidimensional arrays.

• *How much code re-use was possible? If something was not possible, please describe why.* All code outside of the sweeper was reused. The sweeper was entirely rewritten because: 1) the data access pattern of the original code was not optimized for host-accelerator transfer and for

minimized main memory access by the accelerator, and changing this would require permuting the call tree and likely modifying the class structure significantly; 2) the classes were not set up for minimized use of memory, and since the algorithm performance on an accelerator is to large extent register-file-size-limited, this is a significant issue for performance.

- *What successes have you had with performant code across difference architectures? Were the same algorithms applicable at all across the architectures?* A sweep miniapp, a companion code to the Denovo CUDA sweeper, has been developed, and this code runs with what would be considered good performance (and similar percent of peak flop rate) on both NVIDIA GPUs (CUDA) and Intel Phi (OpenMP, Intel simd directives), with most of the code in common between both architectures. One issue for portability is that the optimal size of an accelerated code region depends on the architecture: the latency of a CUDA kernel launch favors a larger kernel size, but the Intel compiler often fails to vectorize larger loops, and the latency for entering a vector region on Intel hardware is low; thus GPUs favor large kernels, and Intel Phi favors smaller simd loops. Another problem is that function call APIs and directives are fundamentally different coding styles that cannot be easily consolidated: calls to CUDA can be hidden in abstracted functions, but simd directives for decorating loops must be scattered through the code and cannot be isolated for the purpose of separation of concerns. A higher level of abstraction would be helpful if it does not commit the past failure of being too restrictive and not expressive enough to represent new unforeseen usage cases while also allowing a migration path for legacy codes.

- *What approaches did you reject and why? What was the leading contender rejected?* At the time the code was originally developed, directives were very immature, so that was not an option. The situation is changing today, though even now the compilers are not mature and there may be project risk in relying solely on them. OpenCL was not chosen due to not supporting GPU features as well as CUDA. Today, the primary objections to OpenCL would be: too cumbersome, too many device-specific optimizations required, and inadequate vendor support.

- *What is your greatest fear going to exascale for application portability and functionality?* In general terms, the node architectures of all major vendors are converging: hyperthreading, simd lanes, multi/manycore, deep memory hierarchies, NUMA, NVRAM. However, the small details can be different enough to result in developer headaches for creating performance portable code, for example: different ratios of performance speeds and feeds related to hardware characteristics, that might result in different code structuring; different characteristics of parts of the memory hierarchy, e.g., user-managed or system-managed access (addressable memory, direct mapped cache, set associative cache, paging, I/O-style block transfer calls, etc.); lack of vendor agreed-upon standards, e.g., for NUMA memory access or using NVRAM (though at the same time it is of limited value to standardize usage of hardware features that might be transient, if this is in fact the case); all these problems exacerbated by growing node complexity and requisite code abstractions that potentially interact nonorthogonally in complex ways. The results of these are: difficult to maintain code; difficulty future-proofing code since the developer doesn’t know what new vital node architecture quirk is coming next; the need to write possibly large redundant code pieces for different architectures which defeats the purpose of trying to write portable write-once code.