SAND2021-12261PE

Sandia National Laboratories

Exceptional service in the national interest

Lessons Learned in Performance and Portability

Dan Ibanez

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Disclaimer

Very few codes have achieved even partial performance portability as defined by DOE/NNSA

- This is the problem we are trying to solve and reason this presentation exists

The approach outlined in this document <u>does</u> achieve this performance portability

By definition, achieving different outcomes requires a different approach

- So we should expect this approach to differ from “conventional wisdom” in some aspects

This presentation is one correct answer but not the only correct answer

Portable across what?

In order to avoid NDA issues, instead of listing the machines and their characteristics, we will abstract this into goals for hardware and software portability:

1. Support Linux, macOS, and Windows Operating Systems
2. Run efficiently on Intel CPUs that support AVX-512
3. Run efficiently on NVIDIA GPUs
4. Run efficiently on AMD GPUs

The above goals cover the needs of CTS and ATS computing platforms in the foreseeable future at NNSA, as well as software developer laptops and engineering designer desktops and laptops.

Project Management Approach

Develop by rewriting

If the requirements have changed enough, it is cheaper to rewrite than to edit the existing

Today’s code is not the perfect solution to yesterday’s requirements, let alone tomorrow’s

Rewrites are fully portable and conform to all best practices from day one

In your ideal product, how much code is the same as today’s?

Also… really kill the old version, don’t just keep both

Understand the fundamental source of parallelism

Scientific computational models have implicit parallelism in the “pieces” they use to model reality. These pieces might be:

- Finite Element Method elements
- Finite Element Method nodes
- Finite Volume Method cells
- Atoms
- Photons
- Electrons

... etc.

Fundamentally, we want to use parallel computing hardware to handle separate “pieces” in parallel.

Loops over these “pieces” are what we call outer loops

Understand vectorization

1. Both GPUs and CPUs are vector architectures
   - CPU requires explicit vector instructions
   - GPU (warp or wavefront) is auto-vectorizing hardware, but limited by the instructions too
   - Both want loads and stores in outer loop order (GPUs call this “coalesced” memory access)
   - When code is vectorized, this means they both want the same memory layout
   - Full vectorization is needed to make good use of either one
2. Compiler auto-vectorization is a dead end
   - A corollary of this is that we don’t need the Intel compiler anymore
3. SIMD types are our best known approach
   - These are light C++ class wrappers over vendor-specific intrinsics
   - They allow us to write vector-intrinsic code without becoming tied to a vendor
   - They allow us to vectorize serial code more easily

The best way to understand vectorization is to write vector intrinsics directly.

Separate all code into hot and cold

Good performance depends on identification of hot and cold code, and very different treatment of each

Hot code is any function whose number of calls depends on the number of “pieces”

- For example, any function called once per mesh element

Hot code should be designed to run efficiently on the hardware device that provides the majority of the machine’s computing power (e.g. NVIDIA GPUs on ATS-2).

Cold code should be designed as flexible, elegant, fully-featured C++ that supports the efficient execution of hot code without worrying too much about its own performance.

Hot code do’s

Hot code should…

1. Be annotated for GPU compilation (“__host__ __device__” for CUDA and HIP)
2. Write to device memory in outer loop order (required to prevent race conditions)
3. Read from device memory in outer loop order if possible (best performance)
4. Be templated for and programmed to use SIMD types if possible
5. Minimize the amount of work done
6. Minimize the amount of stack space used
7. Use return codes to handle errors
8. Be annotated to force inlining

Hot code don’ts

Hot code should not…

1. Throw exceptions (it is unsupported on GPUs)
2. Dynamically allocate memory (it is slow or unsupported on GPUs)
3. Perform any I/O (it is unsupported on GPUs)
4. Read from or write to CPU memory (a segfault on GPUs)
5. Use virtual functions or function pointers (support is an afterthought on some GPUs)
6. Read from or write to “Unified” memory (support is slow and/or an afterthought)

Organize code into C++ abstract algorithms

All hot code should be executed as part of an algorithm

1. Simple loop (std::for_each)
2. Reduction (std::reduce)
3. Scan (std::exclusive_scan)

- Algorithms are the outer loops, the loops over the pieces
- Use an abstraction layer to provide implementations of abstract algorithms that dispatch the outer loops to the device using vendor-specific middleware (CUDA, HIP, OpenMP).
- A technique similar to the ExecutionPolicy allows you to select a vendor-specific backend without tying the physics code to any vendor.
- Most algorithms have one key “body of code” to execute: use C++11 lambdas for these.
- Hot code is now defined as the lambdas and anything they call.
- Algorithms should have special integration with SIMD types for vectorization

Cold code dos

Cold code should…

1. Use C++ exceptions to handle errors
2. Transform the return codes from hot code into exceptions
3. Call outer loop algorithms from non-member methods
4. Be annotated to prevent inlining, in order to facilitate profiling

Cold code don’ts

Cold code should <u>not</u>…

1. Call outer loop algorithms from non-static member methods (lambda pitfalls)
2. Read from or write to device memory (segfault on GPUs)

Commit to performance

All per-"piece" memory should be allocated as device memory

All per-"piece" operations should be programmed as hot device code
- Operating on device memory

All hot code should be well vectorized

These are goals to strive for, and we do not achieve 100%, but most codes are stuck at less than 50% and for that reason we say "all": we are not doing nearly enough of this.

This is the most important slide.

General C++ design

Most non-trivial code should be in non-member methods

Avoid virtual methods and class inheritance (C++11 std::function helps with this)

Use value types (classes that are small and meant to be passed by value like single numbers)

- SIMD types
- Small vector/tensor algebra types
- Unit-checking types

Use ”auto” typing for left hand side of most assignment statements

Make “big” classes moveable but not copyable

Use the latest versions

Today, that means the following:

1. C++17
2. CUDA 11
3. ROCm 4.2.0
4. GCC 9
5. MSVC 2019
6. CMake 3.21
7. MPI 3.1 (GPU-aware)
8. macOS Catalina

These are standards and tools that are available (or can be easily installed) on every relevant CTS, ATS, and internal CEE resource as well as developer laptops and designer desktops and laptops.

CMake

Use pure modern CMake as the build system for your code

Use native CMake support for CUDA and HIP as “languages”

Use per-target properties for everything from include directories to C++ standard

Export package’s targets, use find_package downstream to import package and targets
- When configuring, use –D<upstream>_ROOT=/path to specify install location
- Prefer find_package(<upstream> CONFIG) over Find<upstream>.cmake modules

Use RelWithDebInfo as CMAKE_BUILD_TYPE
- No separate debug and release builds: always compile C++ with optimization and symbols

Operating system portability

1. Try CApp for your package management:
   - https://github.com/SNLComputation/capp

2. Use C++17 std::filesystem for all file system operations

3. Use C++11 std::chrono for all timing

4. More generally, use only functions in the ISO C++ standard, no POSIX APIs

P3A

Portably Performant Physical Algebra

A library developed to support ALEGRA performance portability

https://github.com/sandialabs/p3a

Provides:

1. C++17 parallel algorithms-like interface for 1D and 3D loops
2. Tensor algebra classes similar to STKMath
3. SIMD classes
4. A std::vector replacement

FIN & Open Discussion