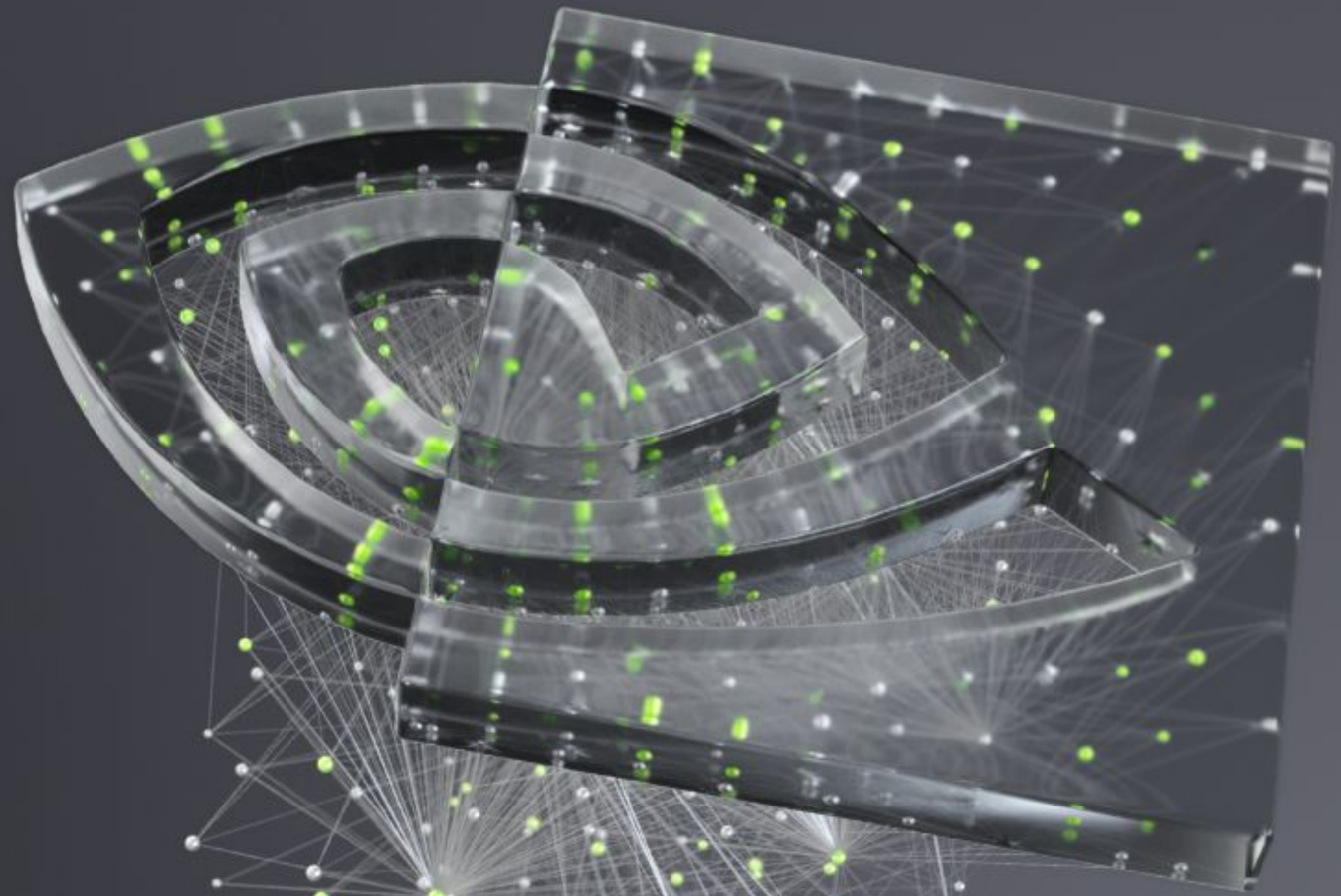


Application scaling in HPC

Jeff Hammond
NVIDIA HPC Group

About Me

No HPC experience until my third year in grad school (in chemistry), until I went to PNNL for a summer internship in 2006, where I worked on NWChem. The system at the time was an Itanium2+Quadrics cluster. I learned Fortran and Linux here.

In 2009, I joined the Argonne Leadership Computing Facility, where I worked on IBM Blue Gene/P and Blue Gene/Q systems. I ported dozens of codes to Blue Gene and optimized a few. My most interesting experiences were with NWChem, Dalton and MPQC. I learned how MPI and C worked at Argonne.

I joined Intel in 2014, where I worked on a range of HPC technologies, including Knights Landing, Omni Path, AVX-512 and other things that are not currently available in the market. I learned OpenMP and (some) C++ at Intel.

At NVIDIA, I am part of the HPC organization, working on NVHPC software, with a focus on the ARM HPC ecosystem, Fortran language development, and application modernization. As of July, I am based in Helsinki, Finland.

https://github.com/jeffhammond has details. https://twitter.com/science_dot is a good way to reach me if other protocols don't work.

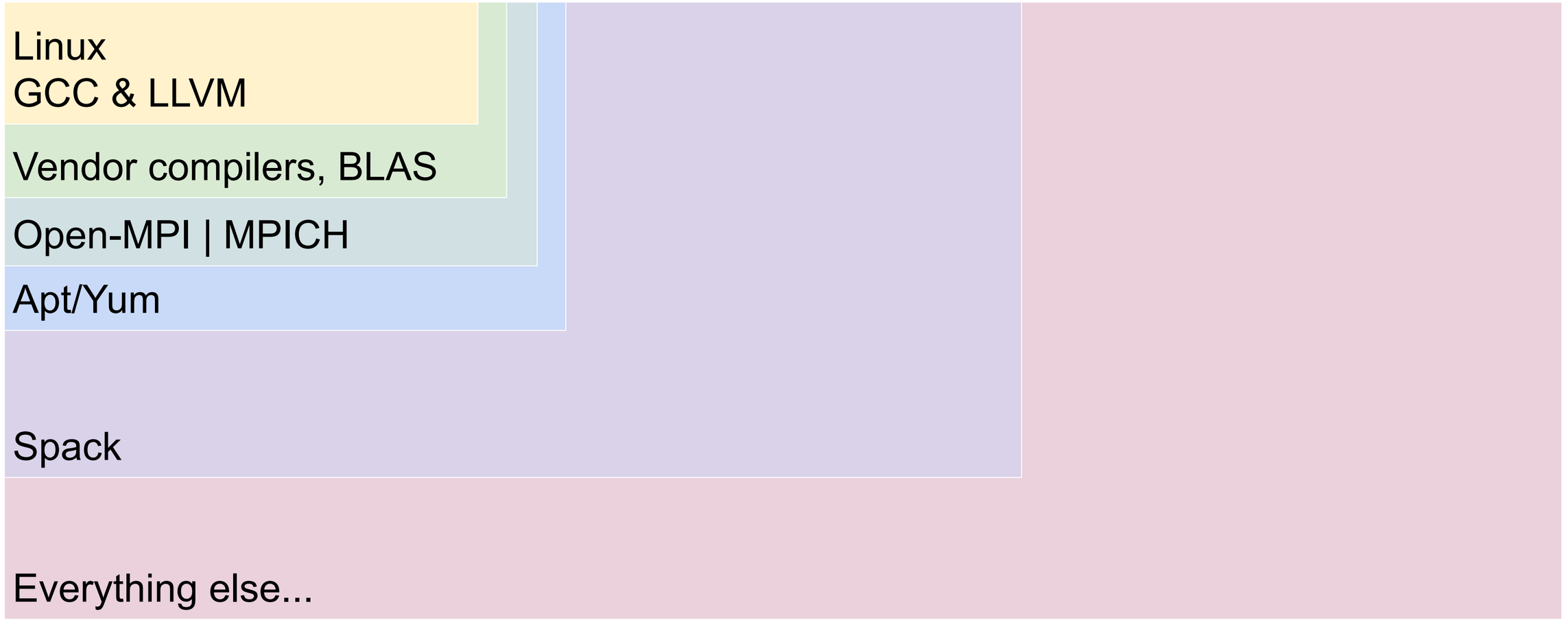
Outline

- Porting to new platforms
- Optimizing for new processors
- Scaling to lots of processors
- Future-proofing with standard parallelism

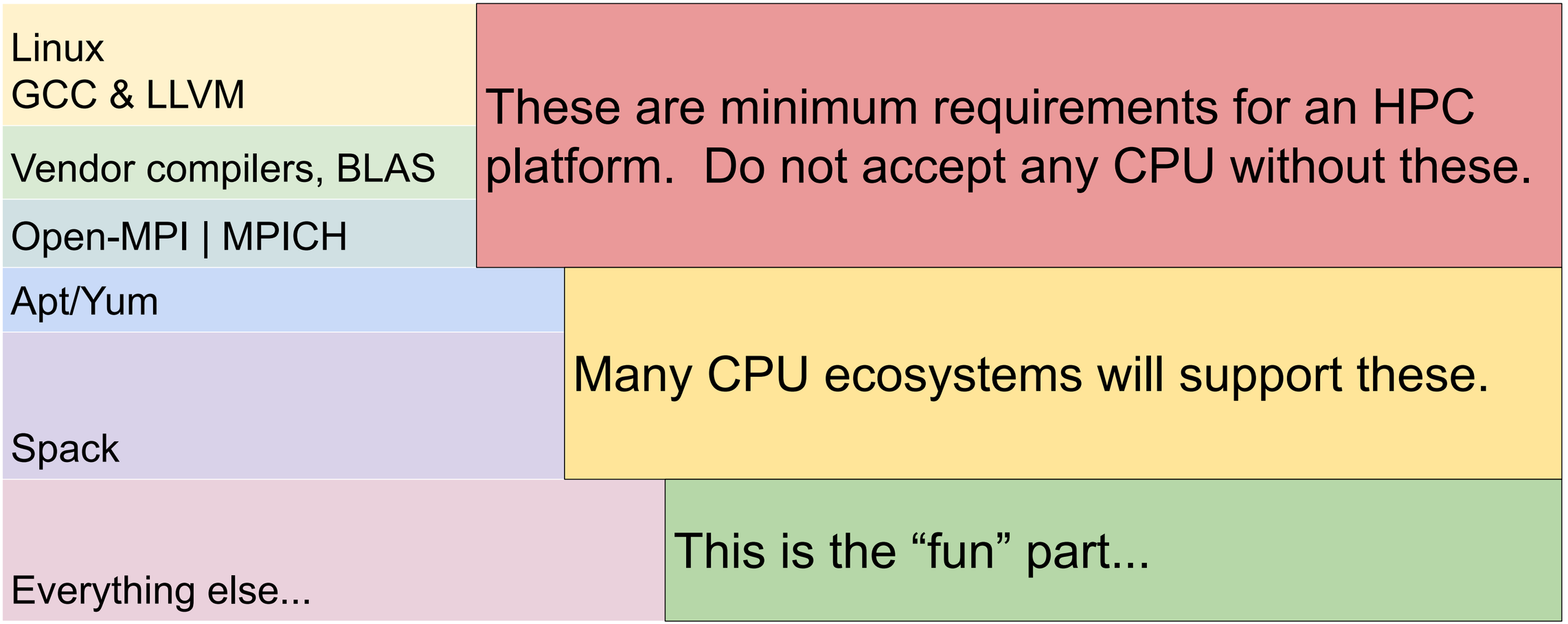
My slides are text-intensive, because that may help folks whose ears are not tuned for my dialect of spoken English.

Traditional HPC Software Ecosystem

Modern HPC Software Ecosystem



Modern HPC Software Ecosystem



Porting

- Always start with the simplest possible configuration, which is usually GCC+Netlib.
  - Do not use any special flags, such as -mtune/-march unless specifically instructed.
  - Building Netlib LAPACK (BLAS) from source ensures your compiler options are compatible.
  - Building MPI from source can be useful if you can tolerate it…
    - Enable debug symbols for GDB backtrace with source line numbers.
    - Use the same compiler as the one you are using. (Fortran modules are weird…)

- Use GDB early and often.
  - MPI+GDB is tricky. Learn how to generate backtraces using non-interactive mode: https://github.com/jeffhammond/HPCInfo/blob/master/docs/Debugging.md
  - I have never found parallel debuggers to be useful.

- Avoid cross-compiling at all costs.

Fun Story

The most challenge bug I've ever seen.

On Blue Gene/P, we had a Fortran application that crashed during startup, in an MPI_Bcast of a scalar (configuration option).

The standard MPI debugging recipe on BG was to disable all the HW-optimized features. Disabling optimized collectives worked around the bug, which meant the application could run and the bug was related to BG MPI magic.

It was eventually determined by Vipin Sachdeva at IBM that:

1) the application was passing a Fortran common block member to MPI_Bcast, and Fortran common block alignment rules caused that member to have 32b alignment.
2) the BG torus network had an undocumented requirement of 64b alignment, which was not checked by the MPI library, and was enforced by the hardware immediately terminating the job with no diagnostic information.

I hope nothing like this ever happens to you 😬

The greatest performance improvement of all is when a system goes from not-working to working.

- John Ousterhout

Read https://web.stanford.edu/~ouster/cgi-bin/sayings.php. It's good.

Performance Optimization

1. Do nothing. Use code that already works well.
   e.g. call qsort() instead of writing your own sorting implementation.
   e.g. never ever ever write matrix multiplication. call BLAS.
   e.g. run Gromacs instead of writing your own molecular dynamics package.
2. **Profile everything early and often, forever.**
3. Enable -O3 and -ffast-math (or equivalent) before hand-tuning.
   Note: Verify that fast math doesn't break correctness.
4. Use a straightforward implementation of known good algorithms.
5. Use generic optimization methods like invariant hoisting, blocking, unrolling…
6. **Enlist a mercenary who optimizes codes for a living.**
7. Read architectural documentation and tune the code yourself.

Optimizing for new processors

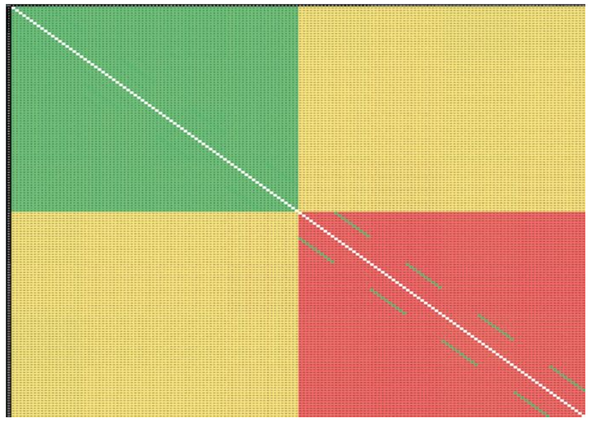
NUMA (NUCA): Locality, locality, locality!

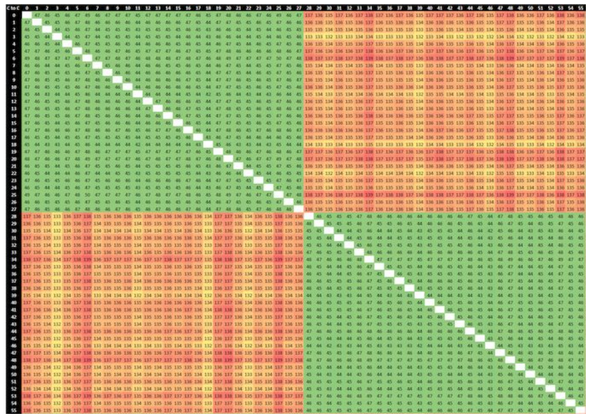
https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php (MPICH etc. have similar)

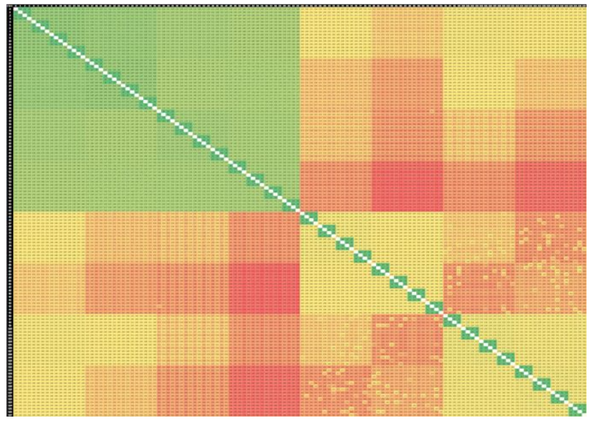
https://linux.die.net/man/8/numactl (hopefully MPIRUN is sufficient…)

https://www.open-mpi.org/projects/hwloc/ (visualize your node)

https://www.anandtech.com/show/16315/the-ampere-altra-review/3 shows these:







Optimizing for new processors

MPI is good at NUMA, threads are not

Process-based parallelism (with affinity/pinning) creates natural locality.

Most multithreaded HPC codes act on shared data, which does not work well across NUMA nodes.

∴ Use at least one MPI process per NUMA node and use threads within that.

You *can* write threaded programs to work on (mostly) private data, or write MPI programs with interprocess shared memory, but both are quite rare, and MPI-3 shared-memory still does a better job with NUMA than OpenMP.

If you use OpenMP, make sure you understand OMP_PROC_BIND and OMP_PLACES. The defaults may not be optimal.

MPI+GPU affinity is still tricky...

https://www.caam.rice.edu/~mk51/presentations/SIAMPP2016_3.pdf (slide 8)

Optimizing for new processors

Understand cache policies

Understand cache policies:

- Blue Gene processors were write-through, which is equivalent to x86 non-temporal store (not the default).
- Big-core (i.e. Xeon/Core) x86 processors have a huge OoO window. Small-core (many-core) processors are (nearly) in-order. After some number of load-misses, the pipeline stalls.

∴ Matrix transpose loop order on Blue Gene favor contiguous reads and non-contiguous writes, whereas Intel big-core favors non-contiguous reads and contiguous writes (because stores RMW cache lines).

Optimizing for new processors

Understand how atomics work

x86 provides very strong atomicity guarantees (64b+ [1]) that make incorrect programs run correctly. All other architectures have weaker - and easier to implement - memory models.

You may see legacy multithreaded code that breaks on ARM. This code is WRONG and must be fixed - ARM is not the problem.

DO NOT TRY TO UNDERSTAND HARDWARE MEMORY MODELS!!!!

Use C++11/C11 atomics (e.g. [2]). *Use the minimum necessary consistency* (sequential consistency is the default, which is often overkill).

Fortran doesn't have a (shared) memory model but OpenMP does...

[1] https://rigtorp.se/isatomic/
[2] https://github.com/m-a-d-n-e-s-s/madness/pull/150

Optimizing for new processors

TLB (page size) effects continue to grow.

4K pages have been the default since the 1960s (?), when computers were a billion times slower. 4K pages are awful and you should never used them on purpose.

Using 64K+ pages is easy on modern Linux systems. x86 favors 2M, AArch64 offers a wider range of options.

https://man7.org/linux/man-pages/man3/posix_memalign.3.html with a large alignment will often take care of it.

https://linux.die.net/man/7/libhugetlbfs

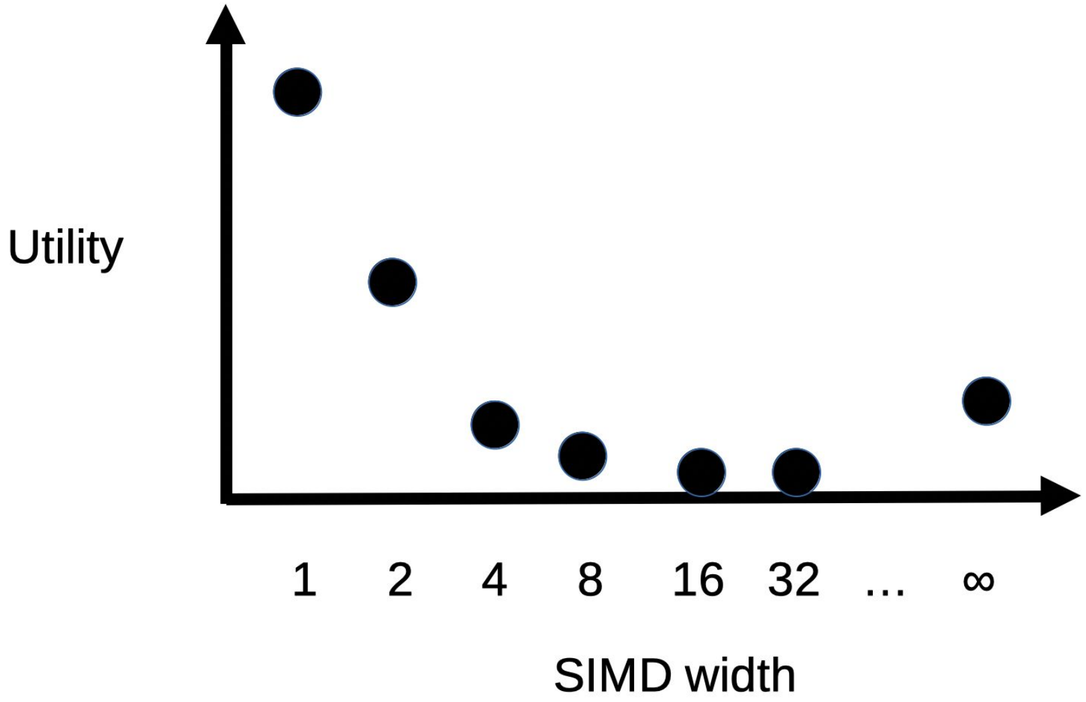
https://wiki.debian.org/Hugepages

https://access.redhat.com/solutions/46111

Optimizing for new processors

SIMD issues

- The optimal SIMD size is 1.
  SIMD exists because instruction decoding isn't free, not because we like it.
- For dense linear algebra, using SIMD is mostly straightforward. Gather-scatter creates a lot of tradeoffs and headaches.
- Inner loop vectorization is low effort, low reward, and rarely benefits past 128b.
- Outer loop vectorization (SPMDization) requires restructuring, but can have huge payoffs. (Aside: this is why CUDA has been so successful.)



Note: Vectorization is a software method. SIMD is a hardware implementation method. Many GPUs are SIMT, not SIMD. CUDA threads make independent forward-progress, unlike SIMD lanes.

Scaling

Always do the easy stuff first.

- The fastest way to move data between nodes is to not move data between nodes.
  - Replicated data algorithms were bad when memory was scarce. It isn't.
  - Per-process memory limitations are best alleviated using shared-memory parallelism, which might be threads, but can also be interprocess shared-memory.

- The cheapest way to synchronize is to not synchronize.
  - Exploit as much independent parallelism as possible, e.g. replicas in MD.
  - Use nonblocking MPI send-recv all the time. Evaluate non-blocking collectives.

- Make sure you are using the correct MPI method for your algorithm…
  - e.g. don't write send-recv if your pattern is a Gather.

Scaling

MPI isn’t perfect 😔

- MPI libraries try to be optimal when averaged across use cases, but are not locally optimal.
  - MPI_Alltoallv is impossible to optimize globally. Consider tuning your own implementation using nonblocking send-recv.
    e.g. https://dl.acm.org/doi/abs/10.1145/2464996.2465442
  - Some implementations do weird things…
    https://press3.mcs.anl.gov/atpesc/files/2015/03/hammond-jul31-830.pdf
    https://youtu.be/fyqxV5B4pul (video has wrong slides, use above)
  -
- Don’t use MPI directly in applications - wrap your communication (especially if you use C++) so that you can switch the back-end with minimal effort.

Scaling

Load-balancing is hard

- MPI collectives are rarely a bottleneck. The synchronize they do is. If you see MPI_Barrier at the top of your profiles, you have a serious load-balancing issue.
  - Most codes call MPI_Barrier unnecessarily, or for timing purposes.

- Complex physics codes are increasingly irregular.
  - NAMD uses Charm++ largely because of the automagic load-balancing.
  - NWChem uses a “bad” dynamic load-balancer from Global Arrays, which happens to weak-scale just fine. It’s possible to do better with algorithm-awareness (https://doi.org/10.1109/ICPP.2013.12).
  - Load-balancing is usually domain-specific and really hard. Collaborating with computer scientists might be useful in this case.

Overview of NVHPC Compilers

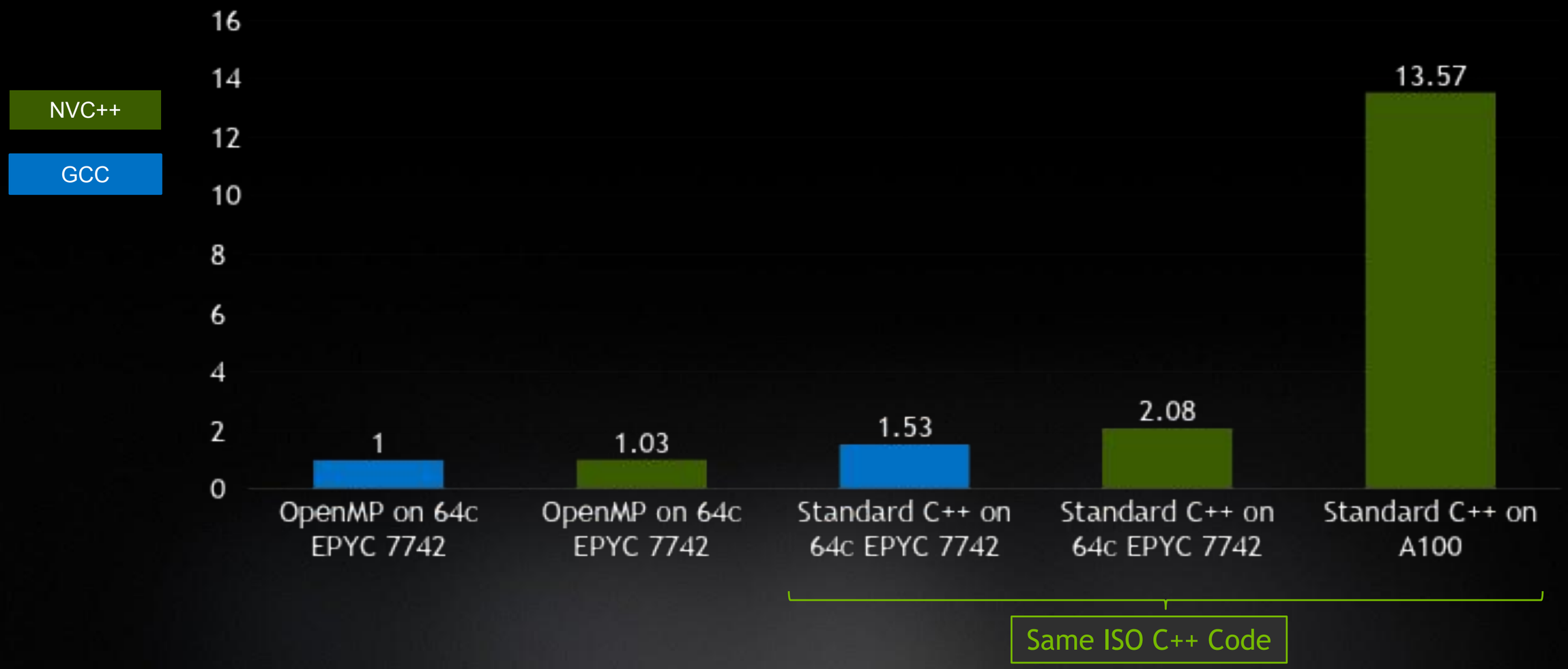
The widest hardware and parallel programming model support of any toolchain.

- Processor support
  - AArch64 CPUs (from Raspberry Pi to Graviton 2 and Ampere Altra)
  - IBM Power
  - NVIDIA GPUs
  - x86 CPUs
- Programming model support
  - ISO C++17 and Fortran 2008 parallelism (DO CONCURRENT)
  - OpenACC
  - OpenMP 4.5 (subset) for CPU and GPU
  - CUDA Fortran and C/C++

https://developer.nvidia.com/nvidia-hpc-sdk-downloads
https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/
https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/
https://youtu.be/KhZvrF_w1ak

LULESH PERFORMANCE

Relative Performance – Higher is Better



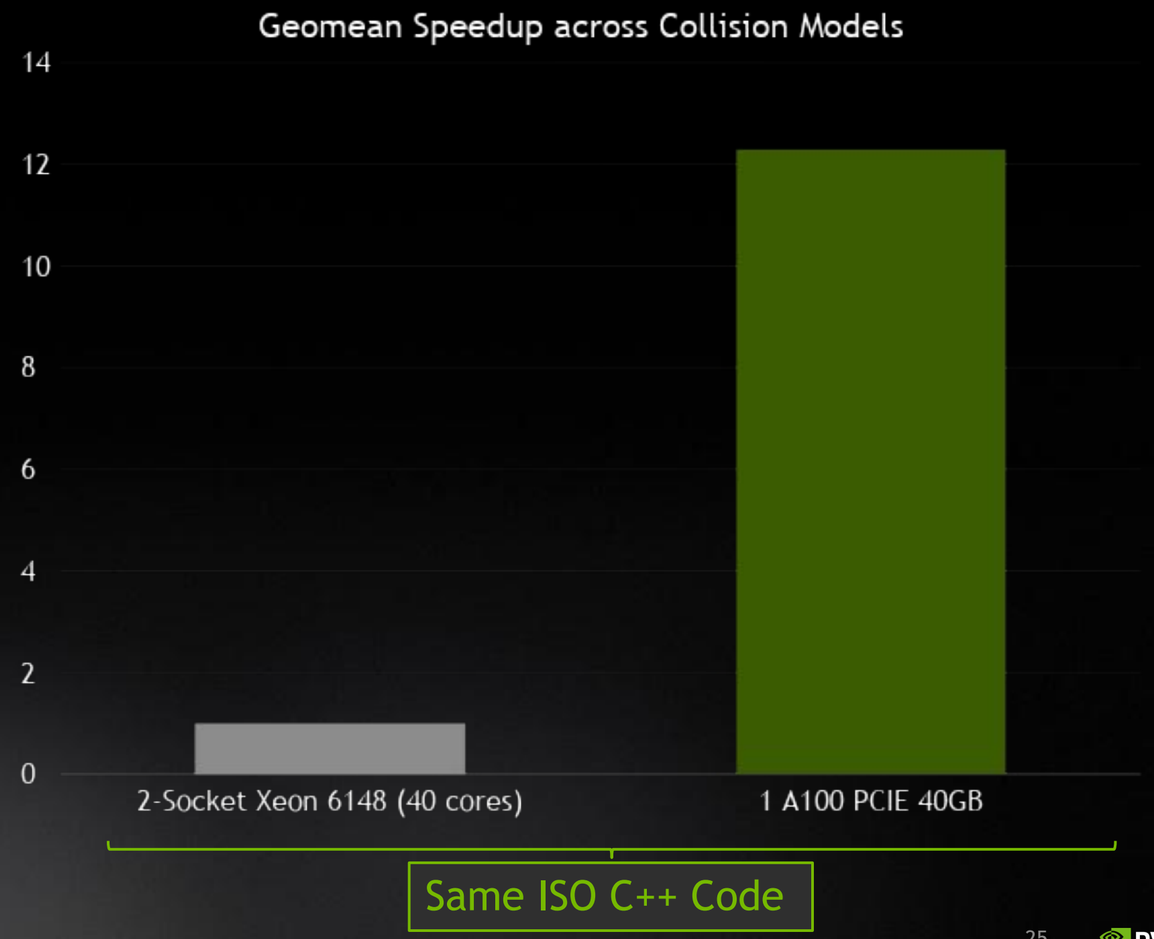
STLBM

Many-core Lattice Boltzmann with C++ Parallel Algorithms

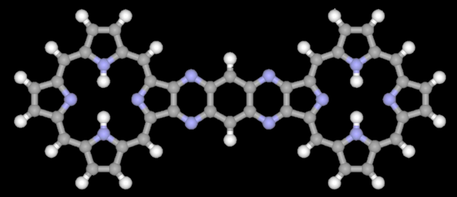
- Framework for parallel lattice-Boltzmann simulations on multiple platforms, including many-core CPUs and GPUs
- Implemented with C++17 standard (Parallel Algorithms) to achieve parallel efficiency
- No language extensions, external libraries, vendor-specific code annotations, or pre-compilation steps

"We have with delight discovered the NVIDIA "stdpar" implementation of C++ Parallel Algorithms. … We believe that the result produces state-of-the-art performance, is highly didactical, and introduces ***a paradigm shift in cross-platform CPU/GPU programming*** *in the community."*

-- Professor Jonas Latt, University of Geneva

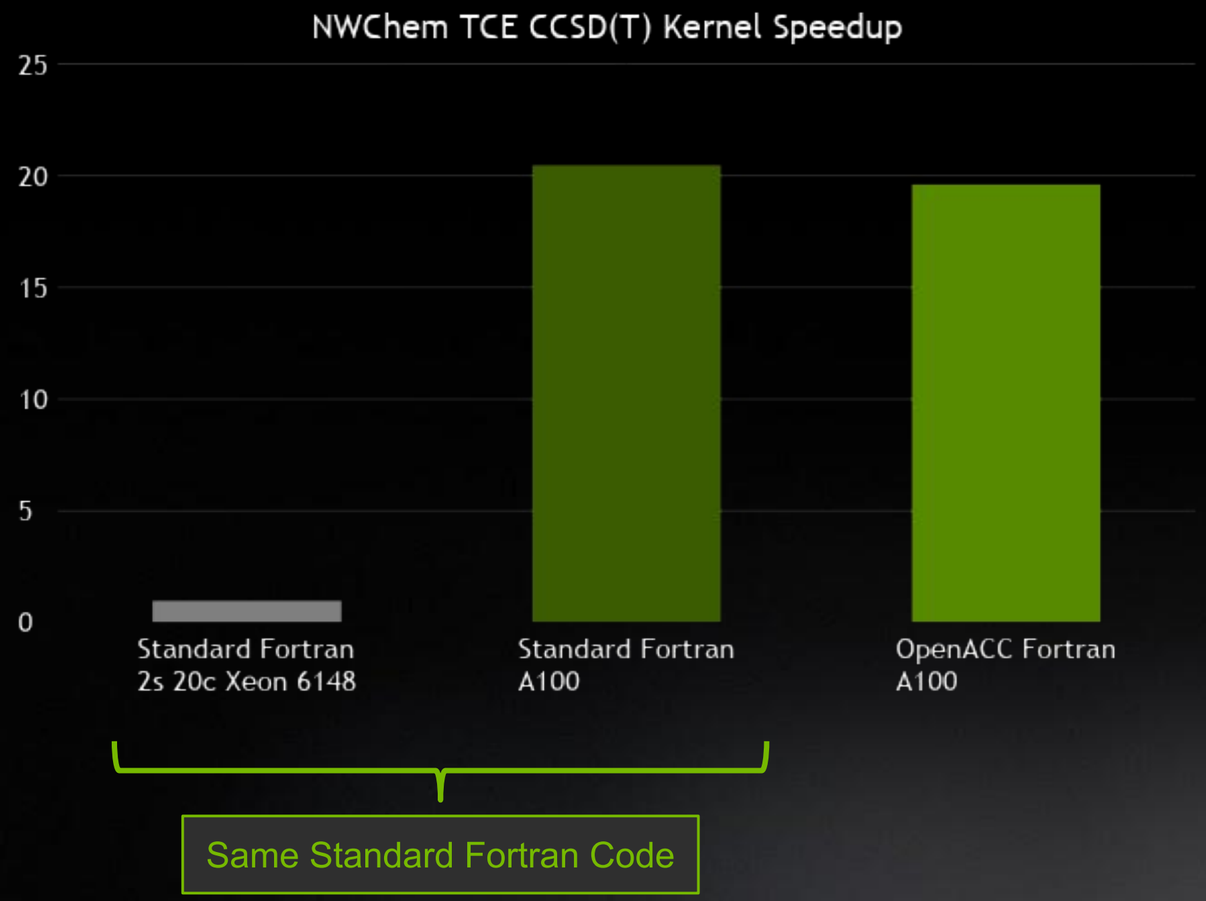


https://gitlab.com/unigehpfs/stlbm



FORTRAN STANDARD PARALLELISM

NWChem with DO CONCURRENT



Fortran stdpar and OpenACC are both faster than OpenMP, due to OpenMP's prescript semantics.

In other parts of NWChem, we use CUBLAS and streams plus OpenACC in Fortran for maximum asynchrony.

In both cases, unified memory (UM) is critical to writing high-performance, maintainable code. NWChem uses UM for both Fortran allocatable arrays and its own custom stack allocator.

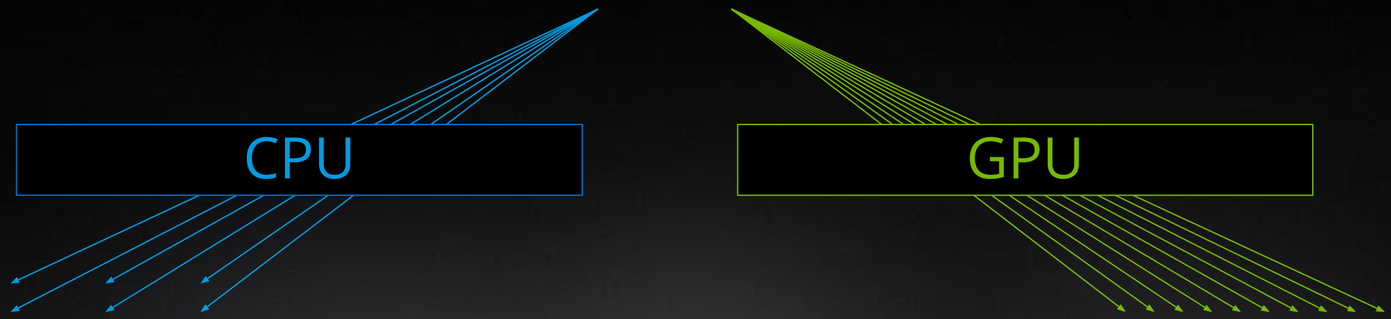
https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/

ACCELERATED STANDARDS

Parallel performance for wherever you code needs to run

```
std::transform(std::execution::par, x, x+n, y, y,
  [=] (auto xi, auto yi) { return y + a*xi; });
```

```
do concurrent (i = 1:n)
  y(i) = y(i) + a*x(i)
enddo
```




```
nvc++ -stdpar=multicore
nvfortran -stdpar=multicore
```

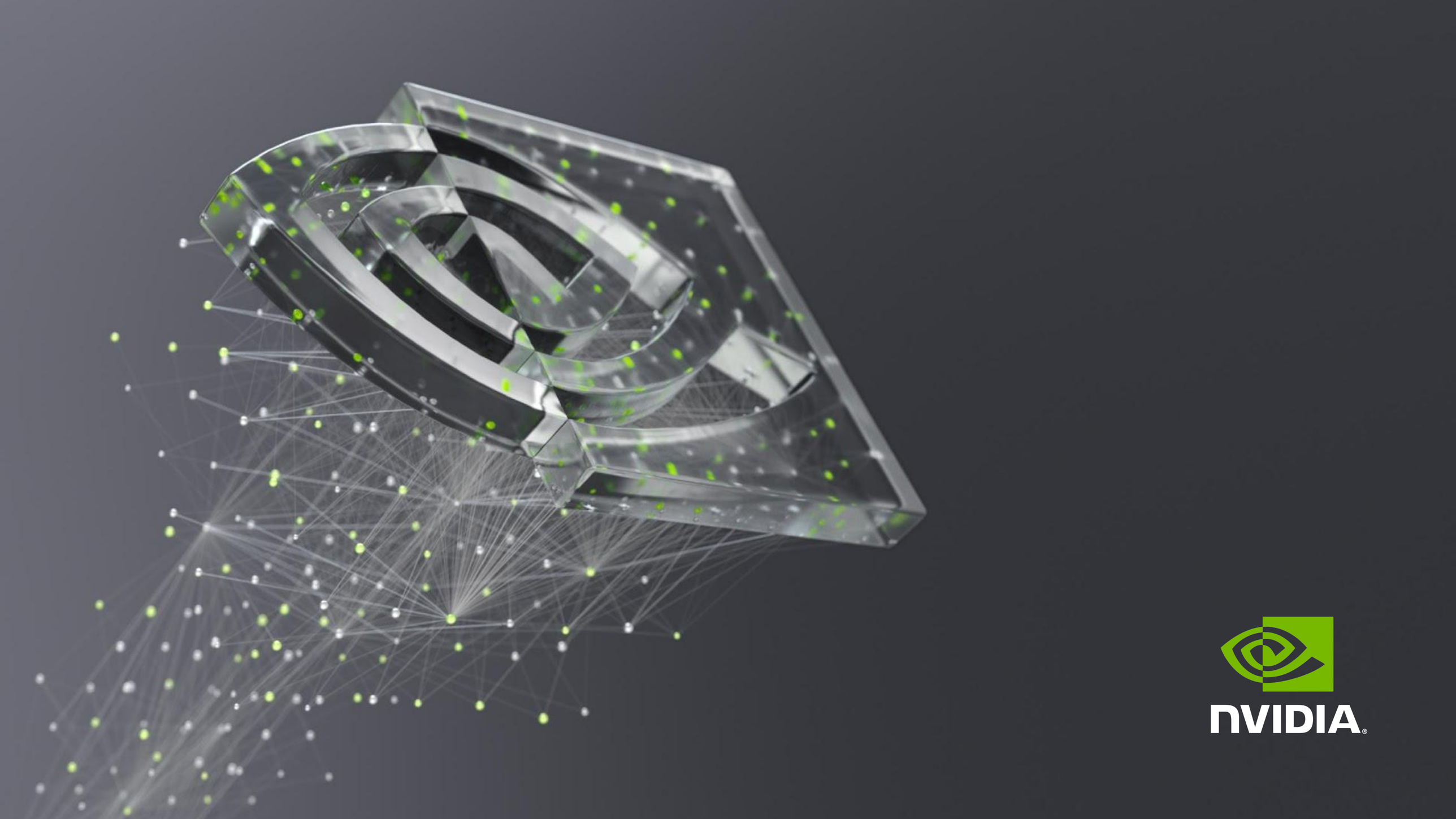
```
nvc++ -stdpar=gpu
nvfortran -stdpar=gpu
```

Questions/Comments

Twitter: https://twitter.com/science_dot

Email: jeff_hammond@acm.org

LinkedIn: https://www.linkedin.com/in/jeffhammond/


NVIDIA