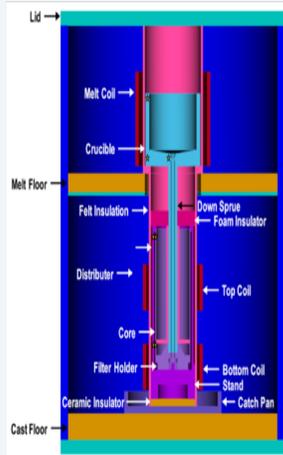


Introduction

This project investigates how different approaches to parallel optimization impact the performance portability of unstructured mesh Fortran codes.

In addition, we explore the productivity challenges due to the software tool limitations unique to Fortran.



As a case study, we optimized one of the key computational kernels of **Truchas**[1], a 3D multiphysics application for simulating metal casting and processing.

Truchas has been in development for over 20 years, is written in **modern Fortran**, and uses **unstructured meshes** for modeling complex geometries.

Figure 1. Induction Furnace 3D Model

Challenges

Kernel optimization is challenging due to available tools, Fortran features, and unstructured meshes.

- Limited compiler support for Fortran 2008
- Limited compiler support for OpenMP GPU
- Lack of performance-portable libraries for Fortran
- No CUDA Fortran compiler compatible with Truchas, forced to **rewrite** computational kernel in CUDA C
- Fortran array syntax requires expanding to **do** loops for OpenMP optimization
- **Indirect addressing** for unstructured meshes causes ineffective caching, in contrast to regular access patterns of structured meshes



Figure 2. Non-Contiguous Memory Access for Unstructured Mesh[3]

Methodology

We modified and optimized the key kernel to investigate the **performance portability** of three parallelization approaches. To determine portability, we ran on a variety of different node configurations. To measure **productivity**, we recorded number of line changes.

Parallelization approaches:

- OpenMP CPU: Directive-based parallel runtime with a work-sharing model
- CUDA: GPU-specific language, only runs on Nvidia hardware
- OpenMP GPU: New directives available in OpenMP 4.0+ enable GPU computing

Hardware resources:

Label	CPU model	S:C:T*	GPU model**
Haswell 2S	Intel® Xeon® E5-2698 v3 @2.30 GHz	2:16:2	N/A
Haswell 4S	Intel® Xeon® E7-8880 v3 @2.30 GHz	4:18:2	N/A
KNL	Intel® Xeon Phi™ 7250 @1.40 GHz	1:68:4	N/A
Broadwell	Intel® Xeon® E5-2695 v4 @2.10GHz	2:18:2	NVIDIA® TITAN V®
Power9	IBM® POWER9™ @3.80 GHz	2:20:4	NVIDIA® Tesla® V100 SXM2

*Sockets : Cores per socket : Threads per core

**Kernel only ran on one of the on-node GPUs

Acknowledgements

Many thanks to our mentors: Bob Robey, Hai Ah Nam, Kris Garrett, Doug Jacobsen, Neil Carlson, and Zach Jibben.

Support provided by ASC Integrated Codes Telluride Project. Support provided by U.S. Department of Energy at Los Alamos National Laboratory supported by Contract No. DE-AC52-06NA25396

Data collected on the Darwin cluster at Los Alamos National Laboratory, and the Cori cluster at the National Energy Research Scientific Computing Center.

References

- [1] Los Alamos National Lab. *Truchas: 3D Multiphysics Simulation of Metal Casting and Processing*. GitLab, 2007-2017. gitlab.com/truchas/truchas-release.
- [2] "Cori Configuration." *National Energy Research Scientific Computing Center*, nersc.gov/users/computational-systems/cori/configuration. Accessed 23 July 2018.
- [3] Stuebe, David. "Unstructured Grid Services." *Ocean Observatories Initiative*, confluence.oceanobservatories.org/display/CIDev/Unstructured+Grid+Services. Accessed 23 July 2018.

Performance and Productivity Analysis

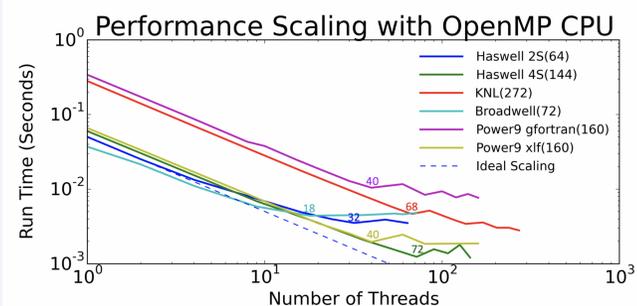


Figure 3. OpenMP CPU performance across node types (lower is better).

Kernel performance scales well initially as the number of cores increases, but scales poorly with more than one thread per core. Only KNL and Power9 gfortran continue to speed-up with more than one thread per core. With the most physical cores (72), Haswell 4S is the best performing hardware.

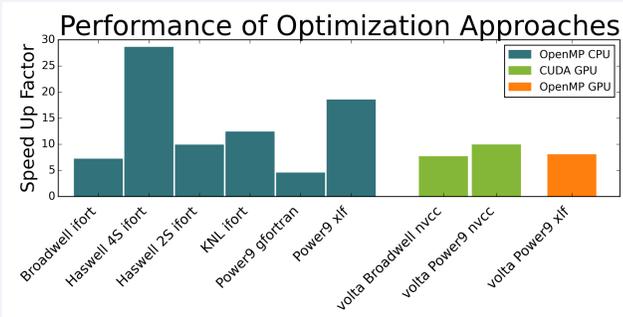


Figure 5. Performance relative to original serial code run on Haswell 2S (higher is better).

Each approach speeds-up the kernel by at least a factor of 4. OpenMP on the CPU performs best due to powerful CPUs, namely Power9 and Haswell 4S. OpenMP on the GPU and CUDA **perform similarly** on one GPU. Power9 xlf runs 4 times faster than Power9 gfortran.

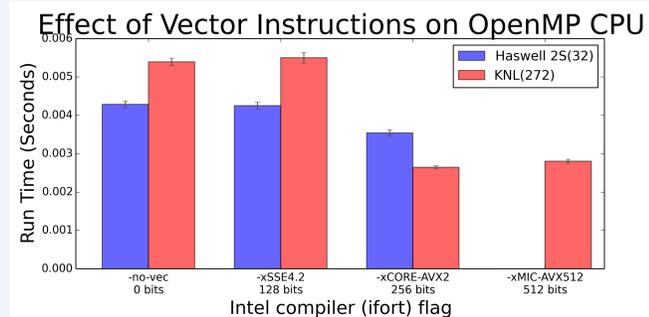


Figure 4. Vector length impact on run time across Intel® processors (lower is better).

For small vector sizes, Haswell 2S performs better than KNL due to its higher clock speed. KNL has more than twice as many vector units as Haswell 2S [2], resulting in a lower run time for 256-bit vectors. The performance for 512-bit vectors likely decreased due to **indirect addressing**.

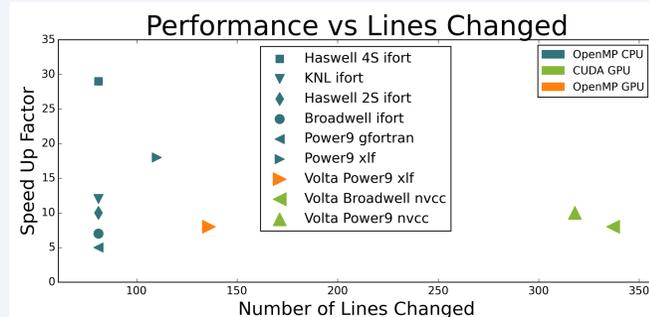


Figure 6. Both OpenMP approaches perform well for less effort (upper-left is better).

OpenMP on the CPU provides the best performance for the least effort. Adding CUDA enables the use of the GPU, but requires rewriting the kernel in C. OpenMP on the GPU performs similarly to CUDA, enabling GPU computation for **significantly less effort**.

Conclusion

- **Compiler choice** can dramatically impact performance on the same architecture.
- OpenMP on the CPU is the **most productive** approach, requiring the least programmer effort for at least a factor of 4 speed-up. It is also the most portable due to widespread compiler and hardware support.
- OpenMP on the GPU is a viable optimization approach, requiring little programmer effort and providing **comparable performance** to CUDA. This approach is currently limited to running on IBM hardware (Power9) using the xlf compiler, but it may become more portable as compilers adopt the standard.
- CUDA is the **least productive** approach because it requires adding a Fortran-C interface and rewriting the computational kernel in CUDA C. More optimization effort may yield increased performance. **Portability** is limited because the CUDA API can only run on Nvidia hardware.
- Fully utilizing all available on-node GPUs could provide a significant performance improvement, especially given the increasing adoption of GPUs in high-performance computing.