Software Ecosystem for Arm-based HPC
Ecosystem for HPC

List of components needed:

- Linux OS availability
- Compilers
- Libraries
- Job schedulers
- Debuggers
- Profilers

Mix of open source and commercial products and applications...

https://developer.arm.com/hpc/hpc-software
Arm development tools portfolio for HPC

Arm Allinea Studio

- **Arm Compiler for HPC**: Linux user space compiler for HPC applications
- **Arm Performance Libraries**: BLAS, LAPACK and FFT
- **Arm Forge Professional**: Multi-node interoperable profiler and debugger
- **Arm Performance Reports**: Interoperable application performance insight

Develop and run on today’s hardware

and also...

- **Arm Code Advisor**: Understand what the compiler could/could not do
- **Arm Instruction Emulator**: Run SVE binaries on today’s hardware

Explore tomorrow’s architecture today
Arm Compiler – Building on LLVM, Clang and Flang projects

Arm C/C++/Fortran Compiler

C/C++ Files (.c/.cpp) → Clang based C/C++ Frontend → LLVM IR → IR Optimizations → LLVM IR → LLVM based Optimizer

Fortran Files (.f/.f90) → PGI Flang based Fortran Frontend → LLVM IR → IR Optimizations → LLVM IR → LLVM based Optimizer

Armv8-A Backend

SVE Backend

Language specific frontend

Language agnostic optimization

Enhanced optimization for ARMv8-A and SVE

Architecture specific backend

LLVM based

ARMv8-A binary

SVE binary
Arm Compiler – OpenMP scaling

Better scaling at higher thread count

Arm Compiler uses libomp based optimized OpenMP runtime

For Lulesh (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics), Arm Compiler shows better scaling than GCC for higher thread count.
DGEMM performance on Cavium ThunderX2

Excellent serial and parallel performance

Achieving very high performance at the node level leveraging high core counts and large memory bandwidth

Single core performance at 95% of peak for DGEMM

Parallel performance significantly higher than OpenBLAS
FFT performance speed-up using Arm Performance Libraries vs FFTW

Configuration: 1D Complex-to-Complex FFT transform, Arm Perf Libs 38.2, FFTW 3.3.7, run on Cavium ThunderX2

- Arm Perf Libs better than FFTW (Speed-up > 1)
- Performance Parity (Speed-up = 1)
- FFTW better than Arm Perf Libs (Speed-up < 1)
Arm HPC ecosystem

Porting to Arm

Arm is engaging directly with partners and HPC scientific code developers to support porting and optimisation of common HPC libraries, tools and applications

Initial focus on successfully building with both Arm and GCC compilers across a broad front

Often only modest changes to environment variables, build scripts and architecture files are needed

*Degree of commonality between codes*
Example: Particle in Cell codes
Two different approaches

**VPIC**

Explicit 2nd order push, charge conserving

FDTD fields

C & C++ with MPI & pthreads

Low particle order

Heavily optimised push, previously tuned for specific platforms

Vector kernel

[https://github.com/lanl/vpic](https://github.com/lanl/vpic)

---

**EPOCH**

Explicit 2nd order push, charge conserving

FDTD fields

Fortran with MPI

High order particles

Flexible, extensible, versatile

Linked list storage

Dependencies: SDF

[http://www.ccpp.ac.uk](http://www.ccpp.ac.uk)
Example: Leveraging Arm intrinsics from C

VPIC

VPIC’s v4 kernel pushes four particles at a time – optimised with SSE SIMD calls

Arm’s NEON instructions offer similar functionality

Datatypes and intrinsic calls from SSE can be mapped over to NEON in many cases

Projects like SIMD Everywhere:

https://github.com/nemequ/simde

may help generate portable code able to exploit Arm’s vector calls

Could such vectorised kernels stand to benefit from Arm’s SVE instructions?
Example: Leveraging Arm intrinsics from Fortran

**Particle prefetch**
Uses intel’s _mm_prefetch to improve performance of linked-list

```
src/housekeeping/prefetch.f90
```

```fortran
SUBROUTINE prefetch_particle(p)
  TYPE(particle),INTENT(INOUT) :: p
  #ifdef PREFETCH
  CALL mm_prefetch(p%part_p(1))
  CALL mm_prefetch(p%weight)
  #endif
END SUBROUTINE prefetch_particle
```

**Arm C compiler preload**
Use __pld in place of _mm_prefetch
Requires Fortran 2003’s C-binding

```
INTERFACE
SUBROUTINE arm_prefetch(p, x, w) BIND(C)
  USE, INTRINSIC :: iso_c_binding
  REAL(c_double),DIMENSION(3) :: p
  REAL(c_double), DIMENSION(c_ndims) :: x
  REAL(c_double) :: w
END SUBROUTINE arm_prefetch
END INTERFACE
```

**C wrapper**
src/housekeeping/arm_intrinsics.c

```c
#include<arm_acle.h>
void arm_prefetch(void const* p)
{
  __pld(p);
  return;
}
```

A similar approach can be used to call GCC’s __builtin_prefetch

© 2018 Arm Limited
Example: Performance improvement
Speed-up memory-bound code

**Armflang vs. GNU**
- Standard arm: 1.2
- Standard gnu: 1

**Armflang with preload**
- Standard arm: 1.2
- Prefetch arm: 0.8

**GNU with prefetch**
- Standard gnu: 1.2
- Prefetch gnu: 0.9
Meet the requirements of HPC developers on Arm

Arm Performance Libraries
BLAS, LAPLACK, FFT

Arm MAP
Cross-platform lightweight profiler

Arm Performance Reports
Maximize System Efficiency

Arm Compiler for HPC
For C, C++ and Fortran codes

Arm DDT
Cross-platform parallel debugger
Community building

Outside the people we collaborate with, various complementary Arm HPC communities already exist:

- Arm HPC User Group (SC) and GoingArm (ISC/ArmRS)
- Arm HPC Google Group ([https://groups.google.com/forum/#!forum/arm-hpc](https://groups.google.com/forum/#!forum/arm-hpc))
- Arm HPC GitLab pages ([https://gitlab.com/arm-hpc/](https://gitlab.com/arm-hpc/))

Encouraging our partners to use GitLab is a priority

Our app work is engaging with code owners and users to get suitable test cases, to get Arm support built in, and including helping them make AArch64 testing part of their development processes
Community site – gitlab.com/arm-hpc
https://gitlab.com/arm-hpc/packages/wikis/home

**Dynamic list of common HPC applications**

Provides focus for porting progress

Community driven.

Maintained by Arm, but anyone can join and contribute.

Allows developers to share recipes, and learn from progress on other applications

Provides a mechanism for tracking status of applications and package sets (e.g. OpenHPC packages, Mantevo, etc.)

---

**Up-to-date summary of package status**

<table>
<thead>
<tr>
<th>Package</th>
<th>External URL</th>
<th>Last Wiki Update</th>
<th>ARM Login</th>
<th>GnuPG/KeyRing</th>
<th>Contact Email</th>
<th>Last Wiki Edit</th>
<th>Package URL</th>
<th>WM/ARM/HiPAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPCLab</td>
<td>[Link]</td>
<td>2018.12.22</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GnuPG/KeyRing</td>
<td>[Link]</td>
<td>2018.12.22</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contact Email</td>
<td>[Link]</td>
<td>2018.12.22</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Wiki Edit</td>
<td>[Link]</td>
<td>2018.12.22</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package URL</td>
<td>[Link]</td>
<td>2018.12.22</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WM/ARM/HiPAC</td>
<td>[Link]</td>
<td>2018.12.22</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tack!
Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद
Migrate and debug application to Arm

Switch between OpenMP threads

Integrate to continuous integration tools

Display pending communications

Visualise data structures
Optimise for Arm platforms

Detect MPI load imbalance

Understand CPU usage

Identify regions of high OpenMP synchronisation
Maximize System Efficiency

CPU Metrics

Linux perf event metrics:
- Cycles per instruction: 2.67
- Pipeline stalls: 631.6% (0)
- L2 cache misses: 193.6 k/s
- Misspredicted branch instructions: 141.0 k/s

Cycles per instruction is high. Lower values are better but are application-dependent. High values may indicate memory latency or branch mispredictions.

Memory

Per-process memory usage may also affect scaling:
- Mean process memory usage: 166 MB
- Peak process memory usage: 173 MB
- Peak node memory usage: 17.2%

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.

Lustre

Lustre file operations (per node):
- Mean write rate: 1.27 M/s
- Peak write rate: 119 M/s
- Mean file opens: 12.3 /s
- Mean metadata operations: 0.11 /s

Aggregate data