AWS Hackathon

Compilers: their use and abuse in HPC

Will Lovett

Compiler Technology Manager, Arm

will . lovett at arm . com

@hpc_will

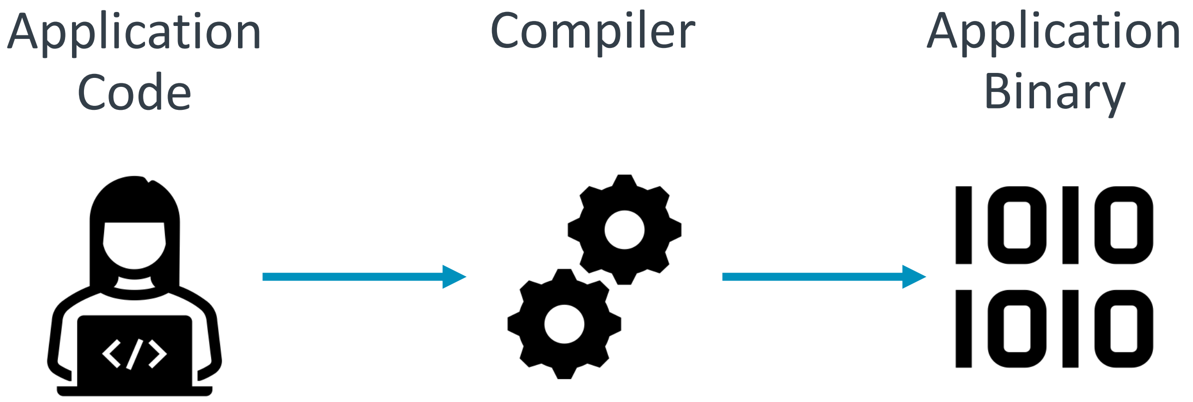
13th July 2021

What I’d like for you to know after this talk

- How compilers work! (the 5 minute version)
- What compilers exist for Arm on the AWS systems you’ll be using, and how to use them
- Some common compiler flags that can help you get the best performance
- A heap of links to find out more

So what's a compiler then?

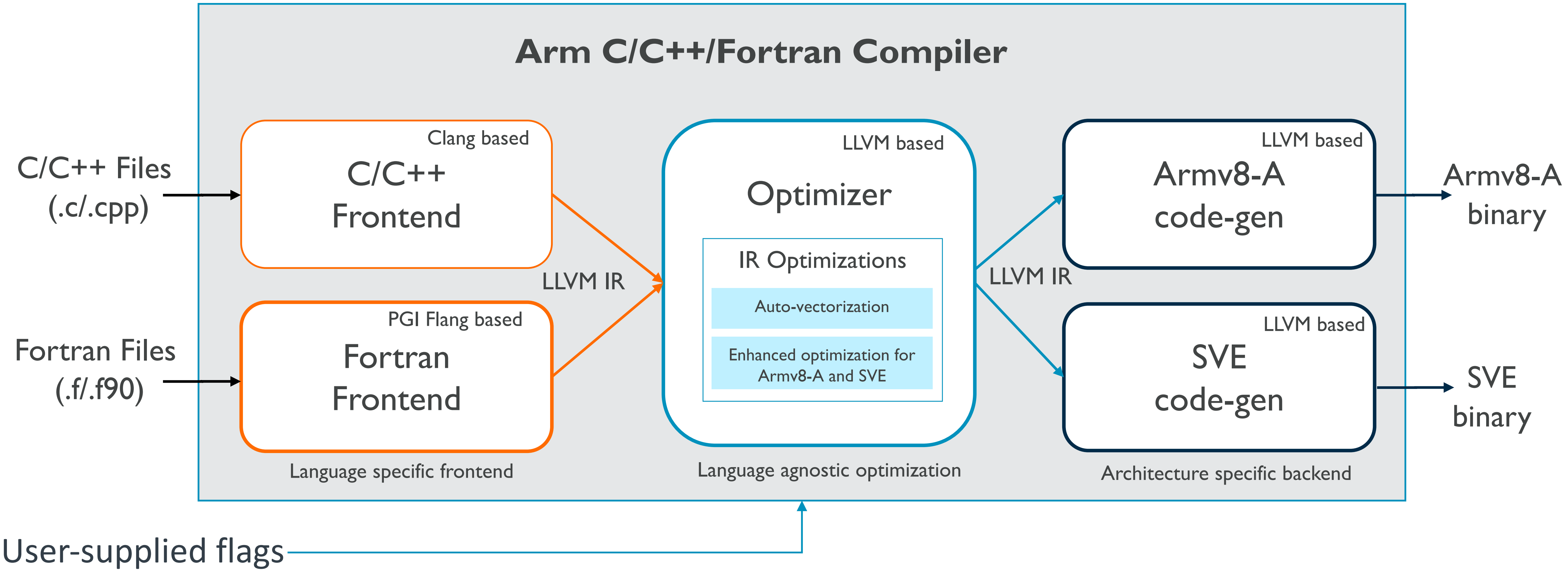
What is a Compiler?



- To many a compiler is just a black box
- Takes application source code and spits out an executable
- Often assumed to work by magic

It does all of that?

An example from the Arm compiler (built on LLVM, Clang and Flang)



What does the Arm compiler ecosystem look like?

- Vendor
  - Arm Compiler for Linux (ACfL)
- OEM Vendor
  - Cray
  - Fujitsu
  - NVIDIA
- Open Source
  - LLVM / Flang
  - GCC

Compilers in Spack

How can we use compilers in the Hackathon?

- Listing available compilers
  - We have preinstalled 3 compilers

  - Arm Compiler for Linux 21.0
  - GCC 10.3
  - NVIDIA HPC SDK 21.2

- We can add new compilers
  - Or compiler versions
  - If needed
  - `$ spack compiler add`
  - Searches environment for new compilers

```
$ spack compiler list
==> Available compilers
-- arm amzn2-aarch64 ------------------------
arm@21.0.0.879

-- gcc amzn2-aarch64 ------------------------
gcc@10.3.0

-- nvhpc amzn2-aarch64 ----------------------
nvhpc@21.2
```

```
$ spack compiler add
==> Added 1 new compiler to /home/ec2-
user/.spack/linux/compilers.yaml
    gcc@7.3.1
```

Building with different compilers

- Compiler choice via Spack '%' notation

```
$ spack install cloverleaf@1.1%arm@21.0.0.879
$ spack install cloverleaf@1.1%gcc@10.3.0
$ spack install cloverleaf@1.1%nvhpc@21.2
```

- Now look at what we have installed

```
$ spack find cloverleaf
==> 3 installed packages
-- linux-amzn2-graviton2 / arm@21.0.0.879 --------------
cloverleaf@1.1

-- linux-amzn2-graviton2 / gcc@10.3.0 ------------------
cloverleaf@1.1

-- linux-amzn2-graviton2 / nvhpc@21.2 ------------------
cloverleaf@1.1
```

Using the compilers outside of Spack

- If testing building software you may need to use the compilers outside of Spack

- Using Environment Modules:

	GCC	ACfL	NVHPC
CC	gcc	armclang	nvc
CXX	g++	armclang++	nvc++
FC	gfortran	armflang	nvfortran

```
$ module load arm21/21.0
$ which armclang
/software/ACFL/21.0/arm-linux-compiler-21.0_Generic-AArch64_RHEL-7_aarch64-linux/bin/armclang

$ module load gcc/10.3.0
$ which gcc
/software/gcc/10.3.0/bin/gcc

$ module load nvhpc-nompi/21.2
$ which nvc
/software/nvhpc/Linux_aarch64/21.2/compilers/bin/nvc
```

OpenMP Environment Variables

OpenMP Runtime Environment Variables

- OMP_NUM_THREADS
  - Normally defaults to the number of cores on the system, i.e. 64 on C6gn
- OMP_DISPLAY_ENV
  - Set to “true” or “verbose” to see values of run-time variables
- OMP_PROC_BIND
  - Strongly recommended to set OMP_PROC_BIND to “true”, “close” or “spread” depending on needs
  - If unset, the threads are not pinned to cores and may migrate around the system
- OMP_PLACES
  - Description of unit of pinning ‘threads’, ‘cores’ or ‘sockets’
- OMP_DYNAMIC=true and OMP_NESTED=true
  - Necessary for “nested parallelism” (multi-threaded functions from within an already parallel environment)

ReFrame and OpenMP

- ReFrame will handle OpenMP
- Using the ‘omp’ field in parallelism
  - Tells ReFrame how many threads to use

- ReFrame generates a job file with this: e.g.
  - 2 nodes
  - 1 MPI rank / node
  - 8 OMP threads / MPI Rank

- OMP_PLACES is set to ‘cores’ by default
  - Tight packing of resources

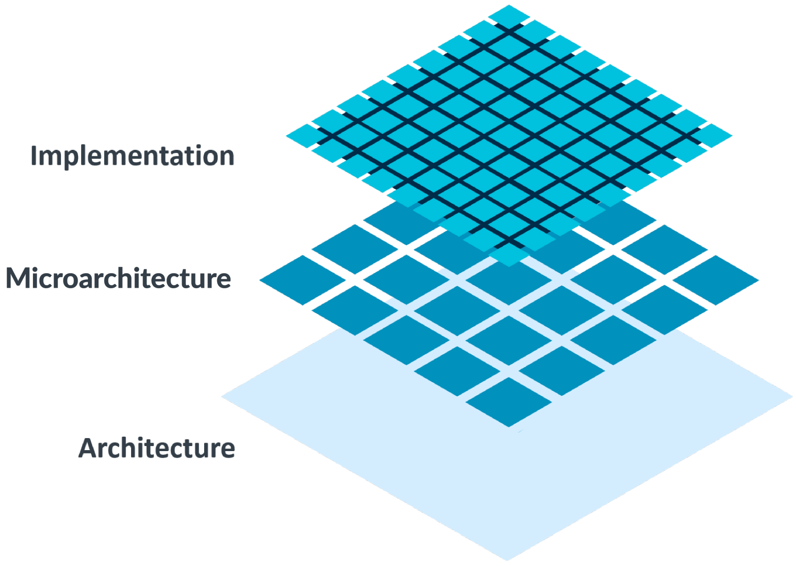
```
# Parameters - MPI / Threads
parallelism = parameter([
   { 'nodes' : 2, 'mpi' : 2, 'omp' : 8},
   …
])
```

```
#!/bin/bash
#SBATCH --job-name="xx"
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --output=rfm_xx__job.out
#SBATCH --error=rfm_xx__job.err
#SBATCH -p c6g
export OMP_NUM_THREADS=8
export OMP_PLACES=cores
srun xx
```

What can your CPU do?

Architecture – Basics

- Arm designs and licenses architectures; and releases an update adding extensions to Armv8 each year.
- These releases have names like Armv8.5-A and Armv8.6-A and form "base" architectures.
  - They also allow for optional extensions such as the Scalable Vector Extensions - SVE
- Very roughly, the architecture what instructions a CPU supports
- This gives CPU designers the opportunity to choose the right architectural feature set for their target markets

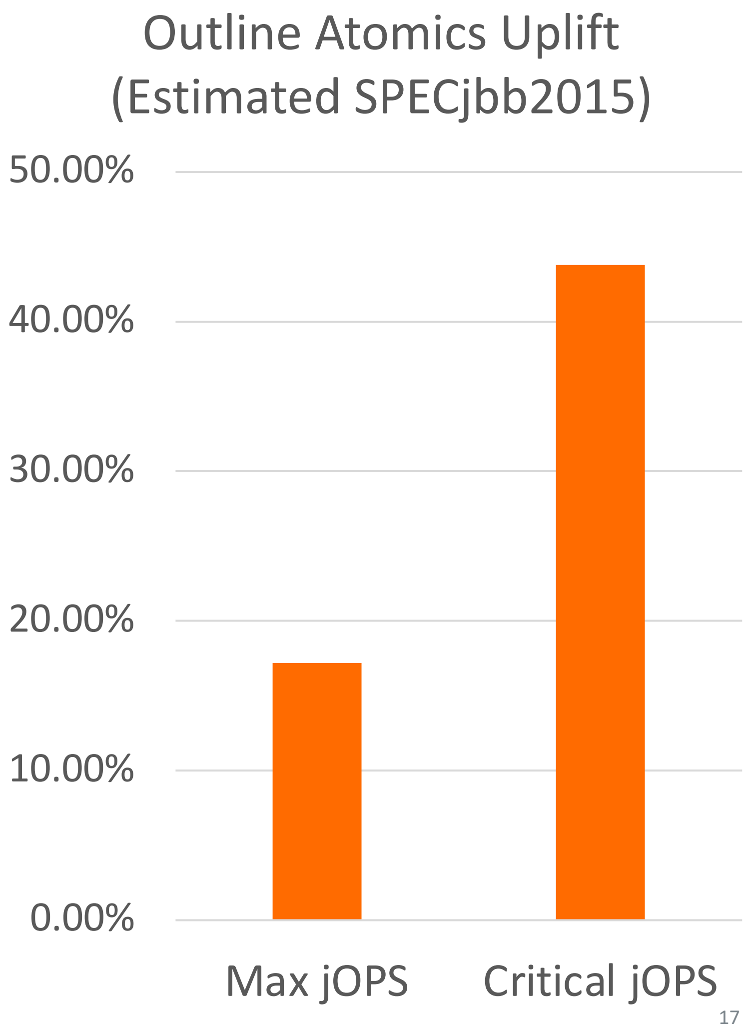


Architecture – what flags to use

- The -march option to the compiler provides expert access to tweak the compiler's code generation to exactly the right architecture version
  - This requires precise knowledge of the systems on which you will execute
  - If you know you will only need to execute on one CPU type, you can use –mcpu with the name of the core to target it
  - `-mcpu=native` sets the architecture features correctly for the CPU on which you are currently executing
- Compilers regularly gain support for new architectures and cores, it is important to use up to date compilers to take best advantage of your system
- Example: All these specify the same architectural features:
  - `-march=armv8.2-a+fp16+rcpc+dotprod+crypto`
  - `-mcpu=neoverse-n1+crypto`
  - `-mcpu=native`

Architecture example: Armv8.1-A Large System Extensions

- Atomic operations, mapping to C11/C++11 <atomic> functionality
- Work well when targeting concurrent systems, such as garbage collectors, memory allocators, OpenMP runtimes
- Designed to enable dynamic use of the new instructions from a single binary.
- Can be accessed using –moutline-atomics
  - Default behaviour from GCC 10.1
- Built into Arm Compiler for Linux OpenMP runtime



Tuning code to your CPU

- Compiler cost models control code generation
- Arm develops compiler cost models in collaboration with our CPU architects
- Using `–mcpu=native` requests that a compiler use the right cost model for your current hardware
- Arm recommends using `–mcpu=native` in the optimisation flags for your project

Image : Fabian Blank – No usage restrictions
https://unsplash.com/photos/pElSkGRA2NU

18

Conclusion

Just use `—mcpu=native`

Compiler Flag Comparison

Compiler flags – recommended starting point for ACfL

- Architecture features and CPU tuning
  - `-mcpu=native`
    - Use all the capabilities of your machine, tuned as well as the compiler is able
    - Universally a good idea, for HPC use-cases
- Optimization flags
  - `-Ofast` - gives the compiler the most freedom. Usually a good place to start, if your code allows it
  - `-O3 -fassociative-math -fno-signed-zeros -fno-trapping-math`
    - Subset of fast that allows reordering of FP instructions & vectorized reductions
    - Some compilers enable these by default at `-O3`
  - `-O3` – Allows vectorization, but maintains IEEE FP behaviours
- Fused FP operations (eg. FMA)
  - `-ffp-contract=fast`
    - Allowed by default for Fortran, or with `-Ofast`
    - Worth trying for many C/C++ workloads that can't use –Ofast
- Link-time optimization
  - `-flto`

Compiler flag reference

	Arm (ACFL)	GCC	NVIDIA (NVHPC)	Intel
Optimization Level	-O0 –O1 –O3 -Ofast	-O0 –O1 –O3 -Ofast	-O0 –O1 –O3 –O4	-O0 –O1 –O3 –Ofast
Fast Math	-ffast-math	-ffast-math	-fast	-fast
Use a sensible architectural features and cost model for your CPU	-mcpu=native	-mcpu=native	n/a (default)	-march=native –mtune=native
Enable OpenMP	-fopenmp	-fopenmp	-mp=multicore	-fopenmp
Enable use of math libraries	-armpl	-larmpl (& friends) Read the docs	n/a (build OSS libraries yourself)	-mkl
Debug symbols	-g	-g	-g	-debug
Vector Report	-Rpass=loop -Rpass-missed=loop -Rpass-analysis=loop	-fopt-info-vec	-Rpass=loop -Rpass-missed=loop -Rpass-analysis=loop	-qopt-report

Would you like to know more?

- Command-line option descriptions, for all the compilers you'll be using:
  - Arm Compiler for Linux
  - GCC
  - NVIDIA HPC SDK

- Porting SSE intrinsics to NEON
  - We've recently published a guide to porting SSE intrinsics to NEON
  - We have an interactive searchable online guide to using NEON intrinsics

- Our hpc developer website has a heap of material, including
  - Quick reference guides
  - Reference guides for Arm C/C++ and Fortran Compilers, and for Arm Performance Libraries
  - A guide to porting and optimizing for Arm
  - A guide to OpenMP thread mapping

Would you like to know (even) more?

Ask on Slack!

#help-arm-compiler

#help-gcc-compiler

#help-nvhpc-compiler

#help-arm

Thank You
Danke
Gracias
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה