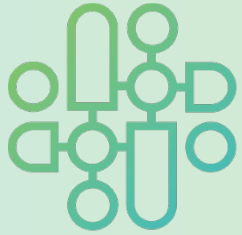


SC23
Denver, CO | i am hpc.

From Zero to Hero:
Conquering the Arm Neoverse

Brendan Bouffler (AWS)
Csaba Csoma (AWS)
John Linford (NVIDIA)
Matt Vaughn (AWS)
Conrad Hillairet (Arm Ltd)
Filippo Spiga (NVIDIA)

Profile and Optimize

PART 2

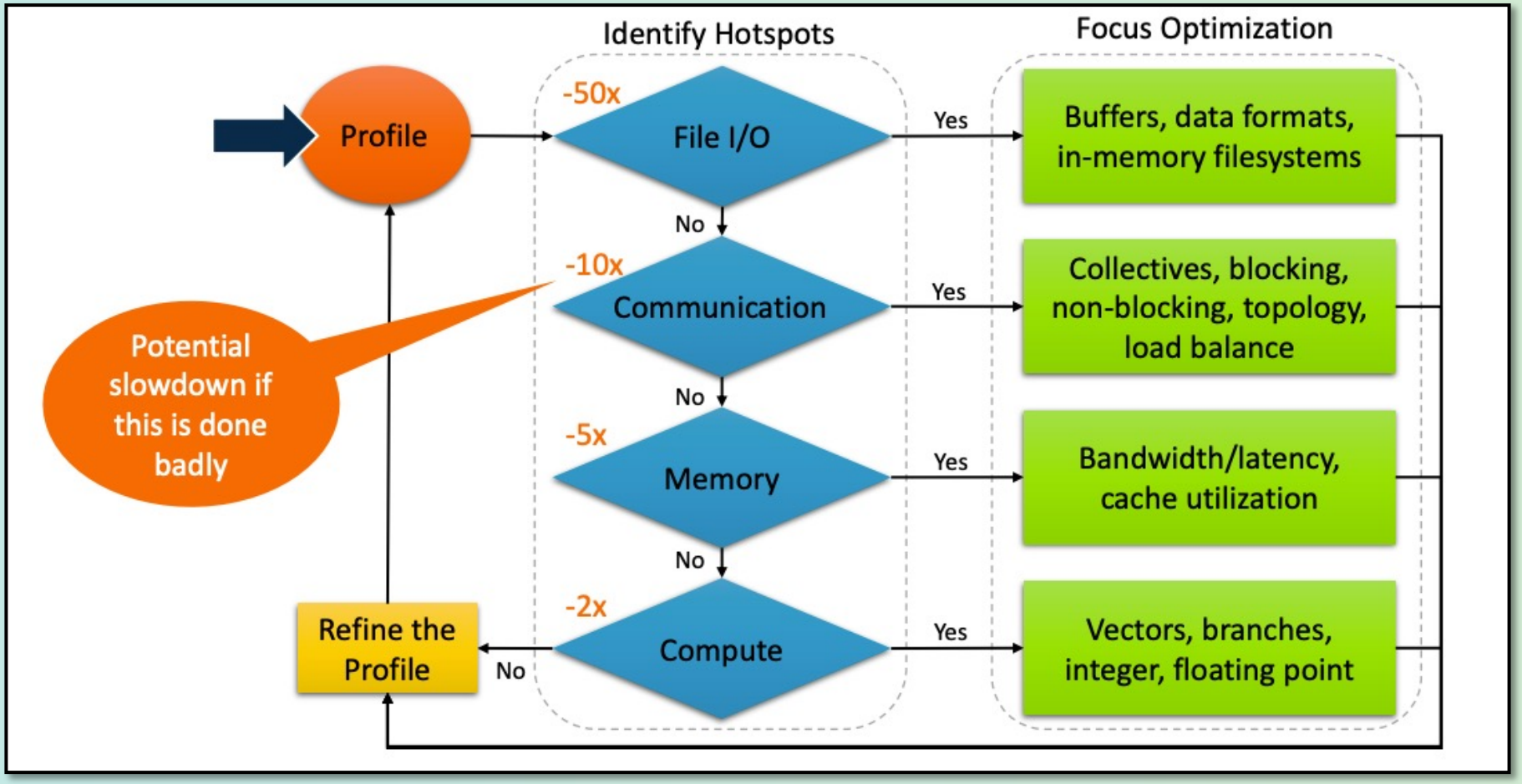
Step 0: Don’t Profile

The profiling preflight checklist

- The code compiles and runs at the desired scale
  - May need to run at a reduced scale, profile, then scale up
- Results are correct, and a copy of the unprofiled results are saved somewhere
  - Profilers occasionally expose surprising bugs. Keep a gold standard handy.
- I’ve recorded the wallclock runtime of the code *without a profiler*.
  - You’ll need this to keep track of profiler overhead.
  - Some profilers are “lightweight” with <10% overhead. Some may slow programs by 100x or more. Depends on what’s being measured.
- Don’t try to profile on the first run. It’s tempting, but you’ll probably waste time.
  - SIMDe might be good enough to get it running.
  - Focus on correctness first, optimize later.

Source: https://github.com/arm-hpc-user-group/Cloud-HPC-Hackathon-2021/blob/main/Slides/Profiling_without_printf_John_Linford.pdf

Identifying and resolving performance issues



PAPI & libpfm4 with Arm Neoverse Support

Upstreamed and maintained by the community

- `cd src`
- `./configure --prefix=$PWD/..`
- `make -j && make install`

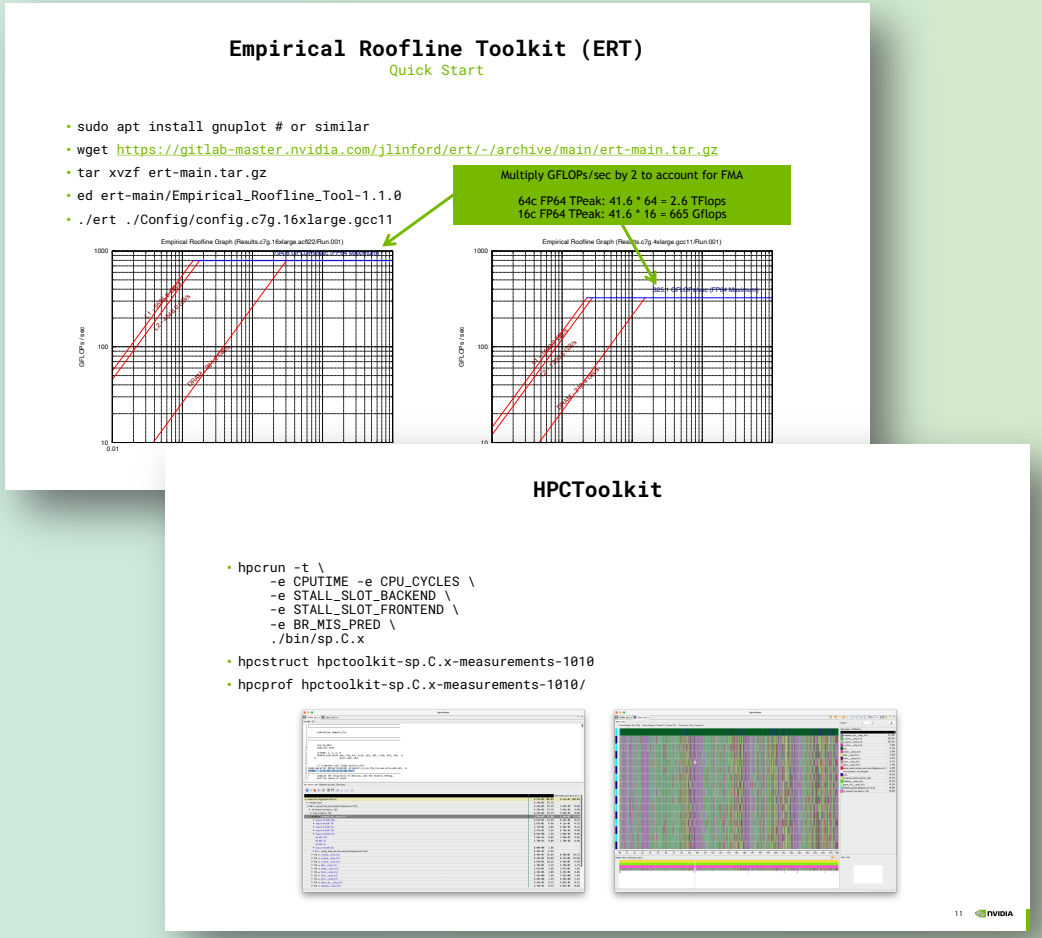
```
[jlinford@c7g-4xlarge-dy-c7g4xlarge-2 papi-jlinford-arm-neoverse]$ ./bin/papi_native_avail | grep SVE
|           Data memory read accesses (includes SVE)                          |
|           Data memory write accesses (includes SVE)                         |
|           Unaligned accesses (includes speculatively executed SVE load and s|
| SVE_INST_SPEC                                                               |
|           SVE operations sepculatively executed                             |
| SVE_PRED_SPEC                                                               |
|           SVE predicated operations speculatively executed                  |
| SVE_PRED_EMPTY_SPEC                                                         |
|           SVE predicated operations with no active predicates speculatively |
| SVE_PRED_FULL_SPEC                                                          |
|           SVE predicated operations with all active predicates speculatively|
| SVE_PRED_PARTIAL_SPEC                                                       |
|           SVE predicated operations with partially active predicates specula|
| SVE_LDFF_SPEC                                                               |
|           SVE first-fault load operations speculatively executed            |
| SVE_LDFF_FAULT_SPEC                                                         |
|           SVE first-fault load operations speculatively executed which set F|
```

```
[jlinford@c7g-4xlarge-dy-c7g4xlarge-2 papi-jlinford-arm-neoverse]$ ./bin/papi_avail | grep Yes
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No   Level 2 data cache misses
PAPI_TLB_DM  0x80000014  Yes   No   Data translation lookaside buffer misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_STL_ICY 0x80000025  Yes   Yes  Cycles with no instruction issue
PAPI_HW_INT  0x80000029  Yes   Yes  Hardware interrupts
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes   Yes  Conditional branch instructions correctly predicted
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_FP_INS  0x80000034  Yes   No   Floating point instructions
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_VEC_INS 0x80000038  Yes   Yes  Vector/SIMD instructions (could include integer)
PAPI_RES_STL 0x80000039  Yes   No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_LST_INS 0x8000003c  Yes   Yes  Load/store instructions completed
PAPI_SYC_INS 0x8000003d  Yes   Yes  Synchronization instructions completed
PAPI_L1_DCA  0x80000040  Yes   No   Level 1 data cache accesses
PAPI_L2_DCA  0x80000041  Yes   Yes  Level 2 data cache accesses
PAPI_L1_DCR  0x80000043  Yes   No   Level 1 data cache reads
PAPI_L2_DCR  0x80000044  Yes   No   Level 2 data cache reads
PAPI_L1_DCW  0x80000046  Yes   No   Level 1 data cache writes
PAPI_L2_DCW  0x80000047  Yes   No   Level 2 data cache writes
PAPI_L1_ICH  0x80000049  Yes   Yes  Level 1 instruction cache hits
PAPI_L1_ICA  0x8000004c  Yes   No   Level 1 instruction cache accesses
PAPI_L2_TCA  0x80000059  Yes   No   Level 2 total cache accesses
```

OSS Performance Engineering Tools

A mix of commercial and OSS tools available and supported on multiple Arm architectures

- These tools help enable Arm developers to the same extent that they are enabled on other architectures
- Too many tools to list! But I'll try 😉
  - HPCToolkit, TAU, ScoreP, Scalasca, LIKWID, CALIPER, ERT, Callgrind, Dimemas, DiscoPoP, Extrae, PAPI, Paraver, MAQAO, … and more!
- Hint: Look for the "ARM" tag in the VI-HPS Tools Guide: https://www.vi-hps.org/cms/upload/material/general/ToolsGuide.pdf



Quick overview profiling tool

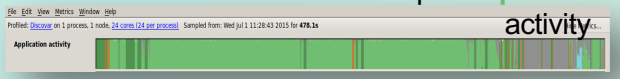
Name	Typical Use	Typical Scale	Languages	Metrics
Linaro MAP	Initial high-level profile to identify hot spots	Single process to thousands of processes	C, C++, Fortran, Python	Wallclock time, hardware perf, MPI, OpenMP, CUDA, custom
Perf	Quick detailed profile of a single process	Single process	Anything, usually compiled	Wallclock time, hardware perf
mpiP	MPI communication profile at any scale	From two to thousands	Anyth ng with MPI	MPI
perf-lib-tools	Math library call profile	Single process	C, C++, Fortran	Count BLAS/LAPACK/FFT calls, call parameters

Commercially supported tools

Linaro MAP

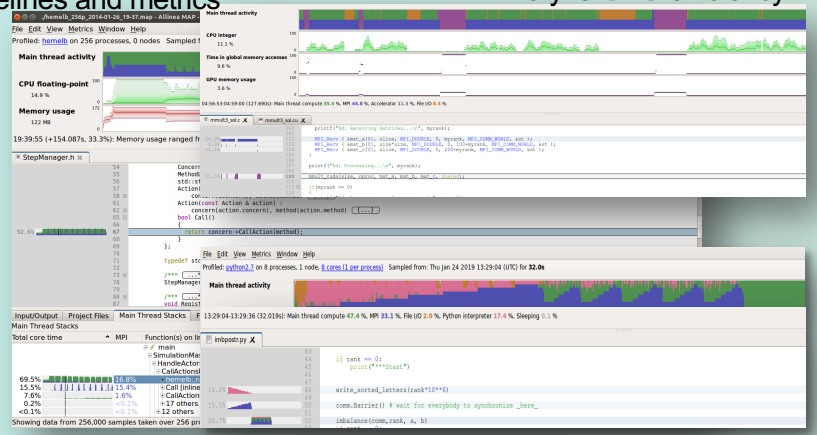
Linaro FORGE

Inspect OpenMP activity



Understand MPI/CPU/IO operations thanks to timelines and metrics

Analyze GPU efficiency



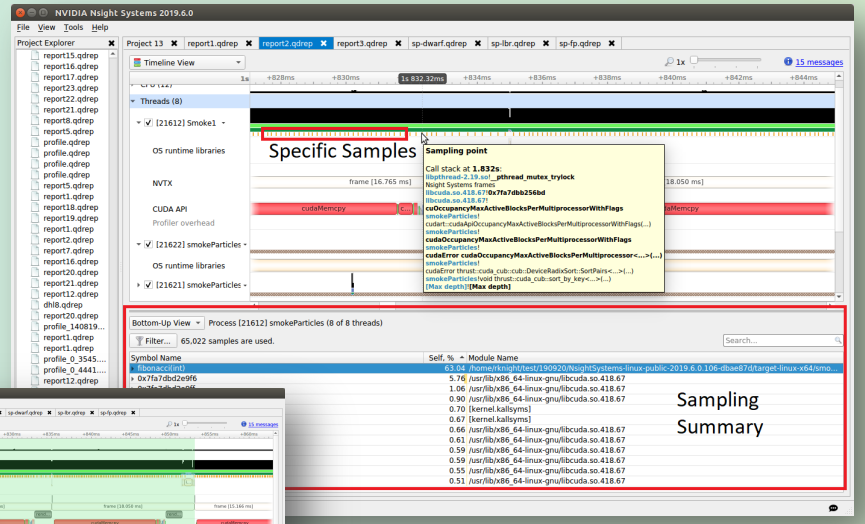
Investigate annotated source code and stack view

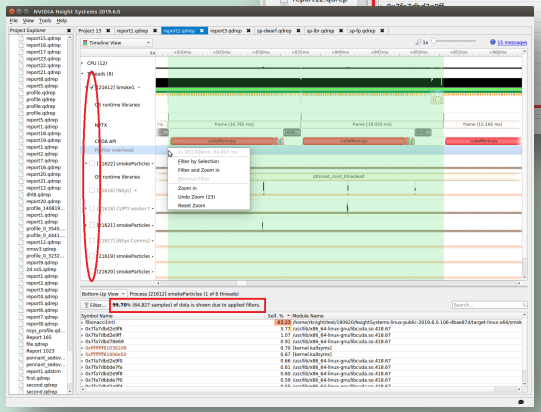
Identify bottlenecks

Profile Python-based workloads

Learn more about Linaro Forge: <PUT QR CODE>

NVIDIA NSight System





Learn more about NVIDIA NSight tools: <PUT QR CODE>

Optimizing codes for Arm CPUs

How different from x86?

Standard techniques and best practices for CPU optimisation works on Arm CPU too

Differences may be observed at very low-level (e.g. SIMD functionalities, memory ordering, atomics)

- Programming directly the SIMD units is not advised, usually.
- Optimisation is about looking at hotspots (so, profiling) and understanding what it is going on
  - Knowledge of the target platform
  - Knowledge of the domain / numerics
- Best outcome is a combination of multiple steps (see example)

Optimizations strategies apply across different CPU architectures

Case study from 1st Teratec Hackathon 2022

Math functions calls (pow)

OpenMP parallelization

Limit number of divisions

Vectorization (Neon, SVE)

Compiler Flags

Compiler optimization remarks

Tools: MAP, MAQAO

Remove unnecessary matrix copies

Reordering & unrolling loops

Cache blocking

Intrinsics

System-wide mutex: writing to 1 file from all cores

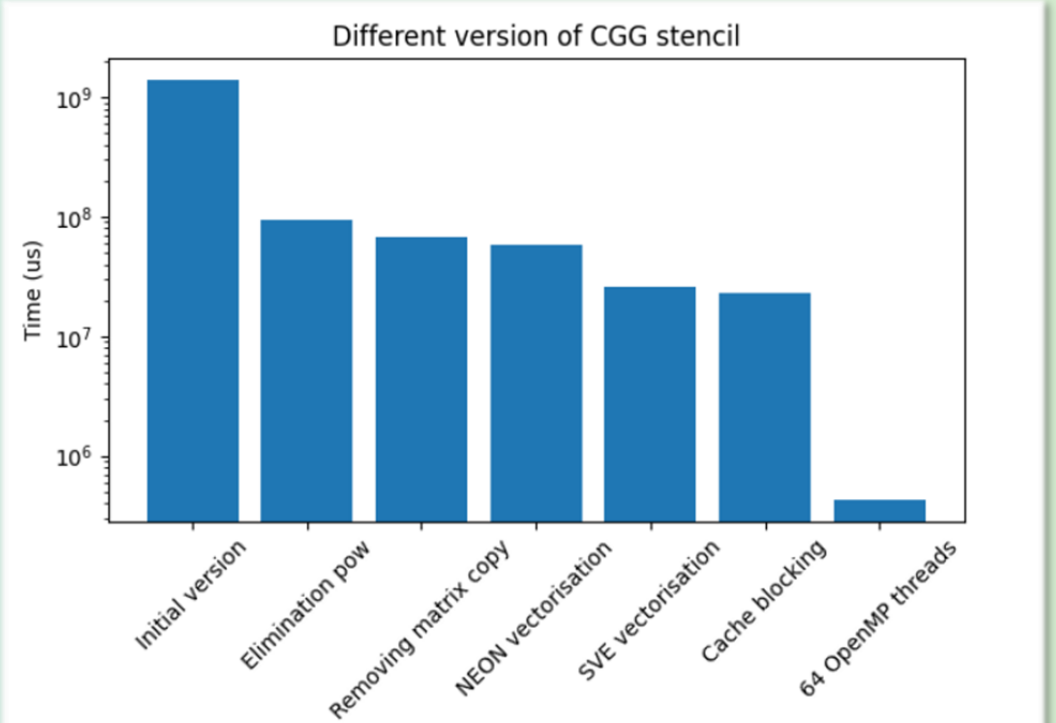


FIGURE 6 – Histogram of the different optimized versions

Optimization and Vectorization Reports

The first rule to optimize is to observe

FLAG	DESCRIPTION
`-fopt-info`	
`-fopt-info-optimized`	What was optimized
`-fopt-info-missed`	What was not optimized
`-fopt-info-all`	Everything
`-fopt-info-vec-missed`	What was vectorized
`-fopt-info-vec-missed`	What was not vectorized
`-fopt-info-all=file.out`	Dump to file

arm COMPILER FOR LINUX

FLAG	DESCRIPTION
`-Rpass=[...] loop-vectorize inline`	What was optimized.
`-Rpass-analysis=[…]`	What was analyzed.
`-Rpass-missed=[…]`	What failed to optimize.
`-fsave-optimization-record`	Dump to .yaml file
`arm-opt-report file.opt.yaml`	Output from .yaml

NVIDIA

FLAG	DESCRIPTION
`-Minfo[=option,...]`	What was optimized
`-Mneginfo[=option,...]`	Why was not optimized

`-Minfo` OPTIONS	
`all`	opt
`inline`	unroll
`loop`	vect
`mp`	intensity

SIMD Programming in the Arm world

Where to start

- Compilers
  - Auto-vectorization capabilities
  - Compiler directives, e.g. OpenMP
    - `#pragma omp parallel for simd`
    - `#pragma vector always`
- Libraries
  - NVIDIA Math Libraries
  - Arm Performance Library (ArmPL)
  - OSS Scientific Libraries (BLIS, FFTW, PETSc, etc.)

For advanced use only

- Intrinsics (via Arm HPC Compiler -- ACLE)
  - Arm C Language Extensions for SVE
- Embedded Assembly
  - SVE ISA Specification
    - The Scalable Vector Extension for Armv8-A

Arm Compiler for Linux: Vectorization Control

OpenMP and clang directives are supported by the Arm Compiler for HPC

C/C++	Fortran	Description
`#pragma ivdep`	`!DIR$ IVDEP`	Ignore potential memory dependencies and vectorize the loop.
`#pragma vector always`	`!DIR$ VECTOR ALWAYS`	Forces the compiler to vectorize a loop irrespective of any potential performance implications.
`#pragma novector`	`!DIR$ NO VECTOR`	Disables vectorization of the loop.

Clang compiler directives for C/C++	Description
`#pragma clang loop vectorize(assume_safety)`	Assume there are no aliasing issues in a loop.
`#pragma clang loop unroll_count(_value_)`	Force a scalar loop to unroll by a given factor.
`#pragma clang loop interleave_count(_value_)`	Force a vectorized loop to interleave by a factor

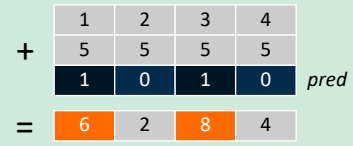
Arm Scalable Vector Extension (SVE)

What it is

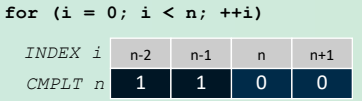
- SVE does not mandate a single, fixed vector length
  - **Vector Length (VL)** is hardware implementation choice of 128 to 2048 bits
  - **Vector Length Agnostic (VLA)** programming paradigm made possible by the *per-lane predication*, predicate-driven loop control, vector partitioning and software-managed speculation
  - **Vector Length Specific (VLS)** enables the use of constructs that are generally not safe for code which is to be run on targets with unknown SVE vector lengths. If you do not require your code to be portable, VLS can be more optimal than VLA.
- SVE is not a simple extension of AArch64 Advanced SIMD
  - A separate, **optional architectural extension** with a new set of instruction encodings (ARMv8.3) focused on HPC

Addressing traditional barriers to auto-vectorization

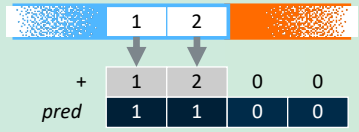
Gather-load and scatter-store



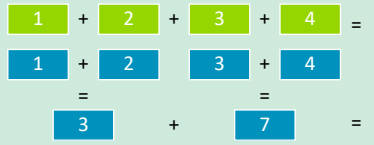
Per-lane predication



Predicate-driven loop control and management



Vector partitioning and software-managed speculation



Extended floating-point horizontal reductions

Porting Assembly and vector Intrinsics

Translate intrinsics to port functionality, then focus on performance tuning

- For a quick fix, use a drop-in header-based intrinsics translator
  - SIMD Everywhere (**SIMDe**): https://github.com/simd-everywhere/simde
  - SSE2NEON: https://github.com/DLTcollab/sse2neon (included in SIMDe)
  - Example using BWA-MEM2: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41702/
- Follow Arm's documentation on rewriting x86 vector intrinsics
  - Porting and Optimizing HPC Applications for Arm SVE: https://developer.arm.com/documentation/101726/latest
  - Coding for NEON: https://developer.arm.com/documentation/101725/0300/Coding-for-Neon
- Arm assembly is simpler than x86
  - Arm processors have a much simpler and general set of registers than x86. Just assign a one-to-one mapping from an x86 register to an Arm register when porting code.
  - Complex x86 instructions will become multiple Arm instructions

Guided Hands-Ons || Bring Your Own Code

50 MINUTES

Part 2 – Practical hands-on

Goal: profile the code, gather insights, identify hotspots and dive into compiler vectorization

Options:

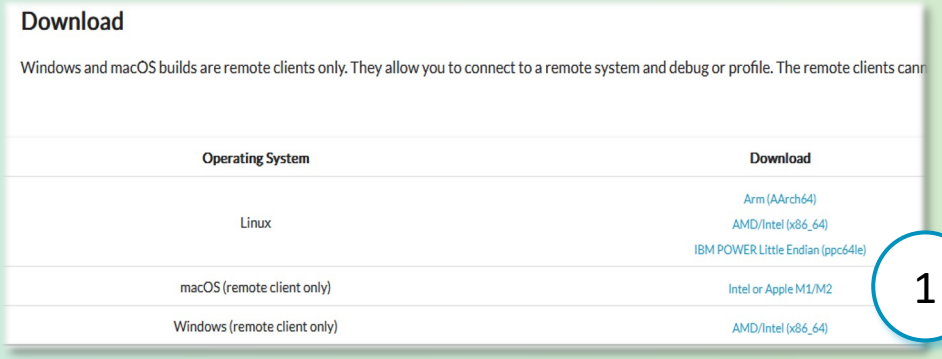
1. "BYOC" - **B**ring **Y**our **O**wn **C**ode
2. Other popular HPC apps (follow the links)

https://github.com/aws/aws-graviton-getting-started/tree/main/HPC

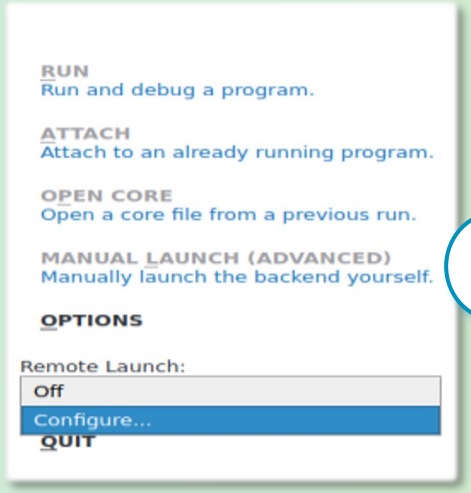
https://github.com/arm-hpc-devkit/nvidia-arm-hpc-devkit-users-guide

Install Forge Local Client (on the laptop)

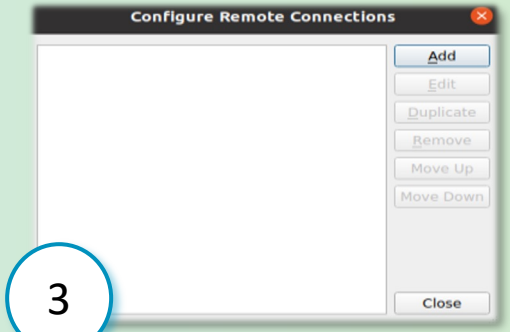
Set up your environment



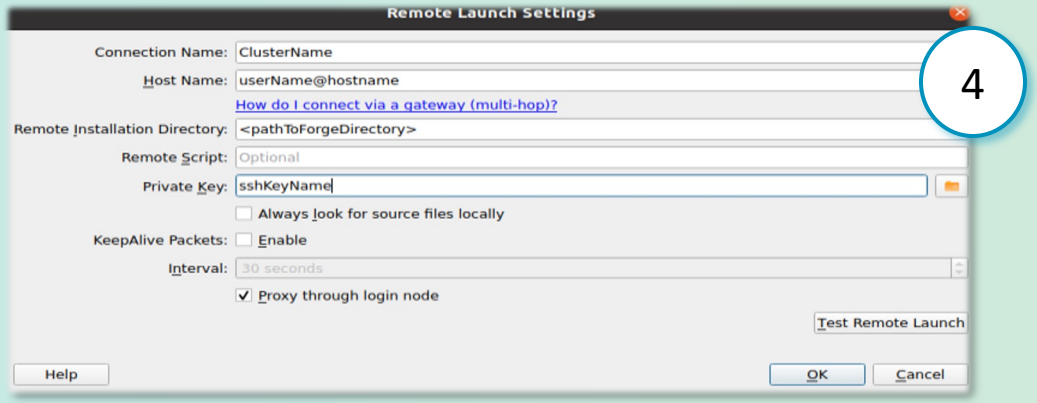
1



2

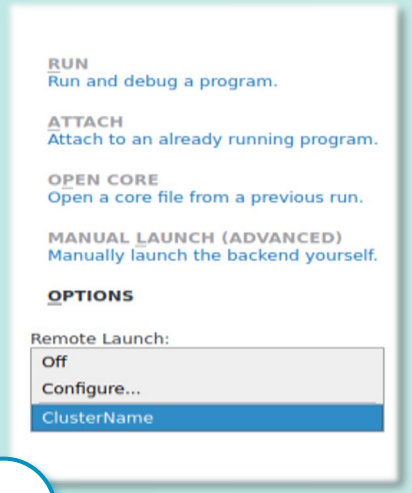


3

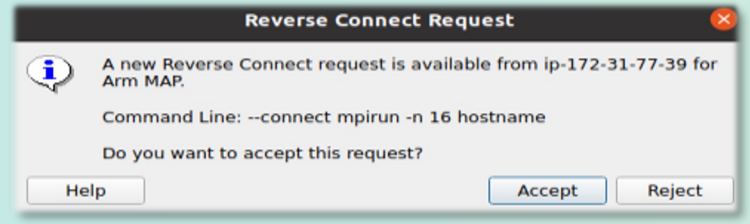


4

5



6



7