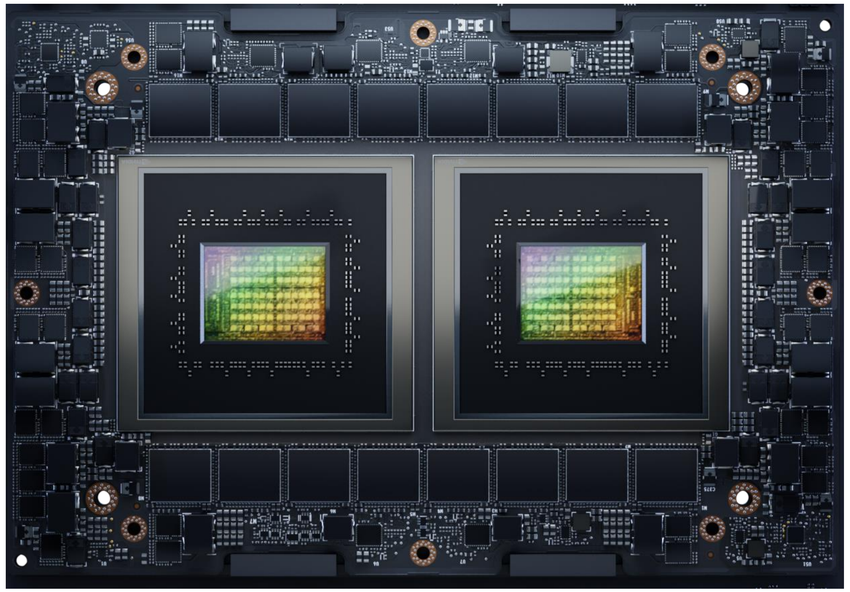
Performance Optimization for Grace CPU

Mathias Wagner, Sr. Developer Technology Engineer | S62275 | GTC 2024

NVIDIA Grace for Cloud, AI and HPC Infrastructure

Grace CPU Superchip

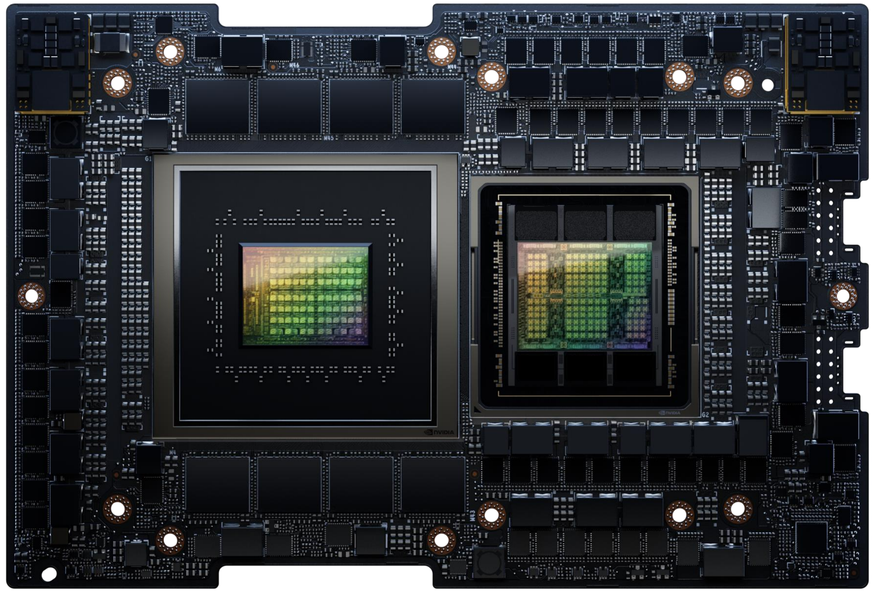
Dual Grace connected with NVLink C2C



CPU-based applications where absolute performance, energy efficiency, and data center density matter, such as scientific computing, data analytics, enterprise and hyperscale computing applications.

GH200 Grace Hopper Superchip

Grace and Hopper connected with NVLink C2C



Accelerated applications where CPU performance and system memory size and bandwidth are critical; tightly coupled CPU & GPU for flagship AI & HPC. Most versatile compute platform for scale out.

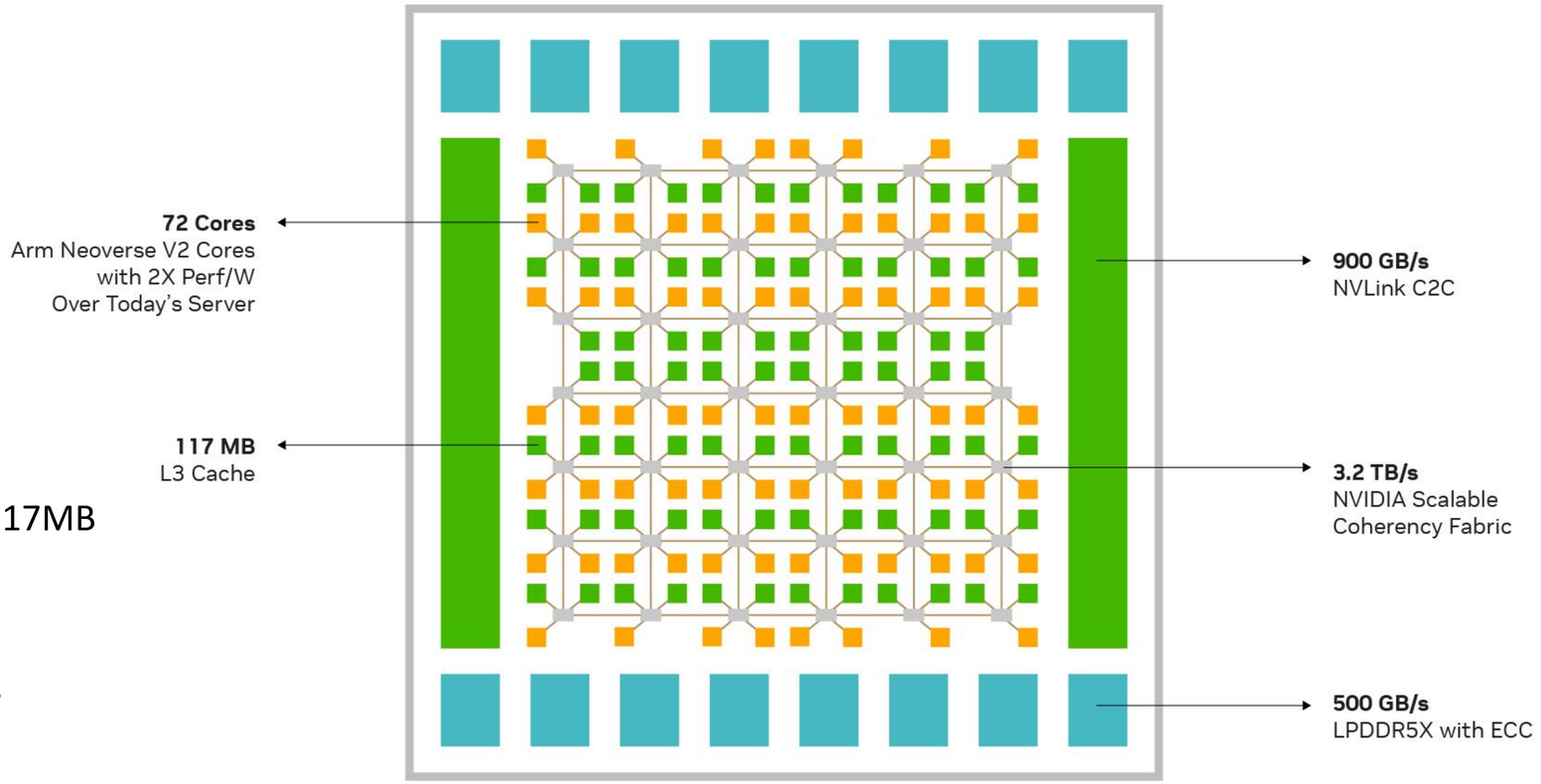
Agenda

- NVIDIA Grace CPU
- Basics
- Optimization
- SIMD
- Summary

GRACE IS A COMPUTE & DATA MOVEMENT ARCHITECTURE

NVIDIA Scalable Coherency Fabric and distributed cache design

- **72 Arm Neoverse V2 Cores**
  - 4x128b SVE2 SIMD units per core
- **Single Die:** single NUMA
- **3.16 Ghz Base Clock**
  - **2.7 GHz Vector Clock**
- **3,225.6 GB/s Bisection Bandwidth**
- **Scalable Coherency Fabric:** Shared, uniform 117MB of L3 cache for entire chip.
- **LPDDR5x**: up to 500GB/s memory bandwidth. Local caching of remote die memory.



Example possible fabric topology for illustrative purposes

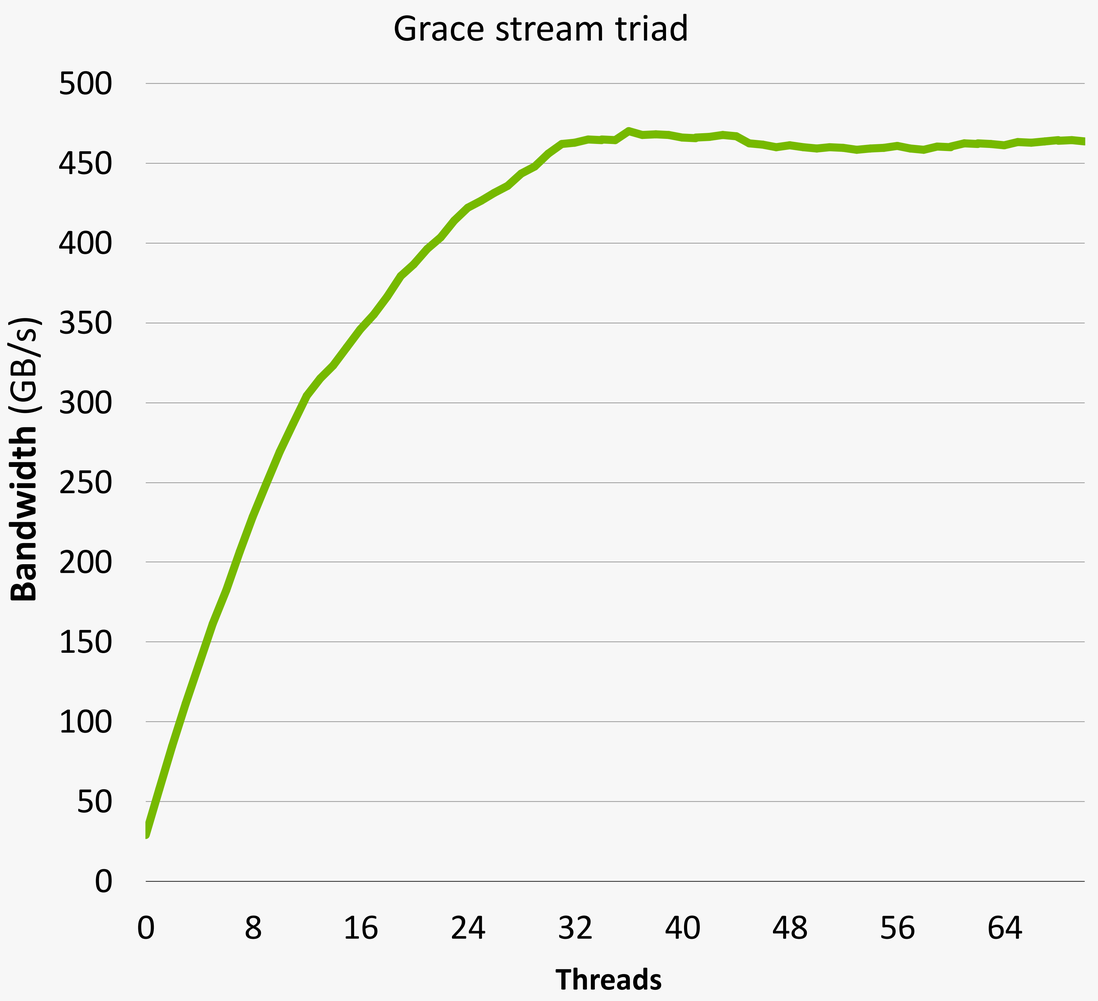
Arm Neoverse V2

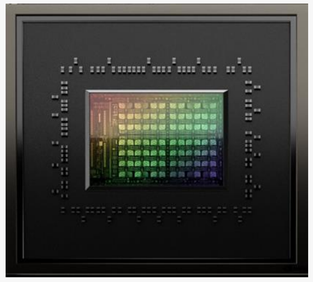
- Grace CPU Neoverse V2 core implements the Armv9.0-A architecture:
  - Binary compatible with Armv8
  - Recompile for optimal performance
- Single instruction multiple data (SIMD) vector instruction sets in a 4x128-bit configuration:
  - Scalable Vector Extension version 2 (SVE2)
  - Advanced SIMD (NEON)
- Large System Extension (LSE) for atomic operations (Armv8.1):
  - Compare and Swap instructions, CAS, and CASP
  - Atomic memory operation instructions, LD<OP> and ST<OP>, where <OP> is ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, or UMIN
  - Swap procedure, SWP
- Additional Armv9 Features:
  - Cryptographic acceleration, scalable profiling extension, virtualization extensions, secure boot

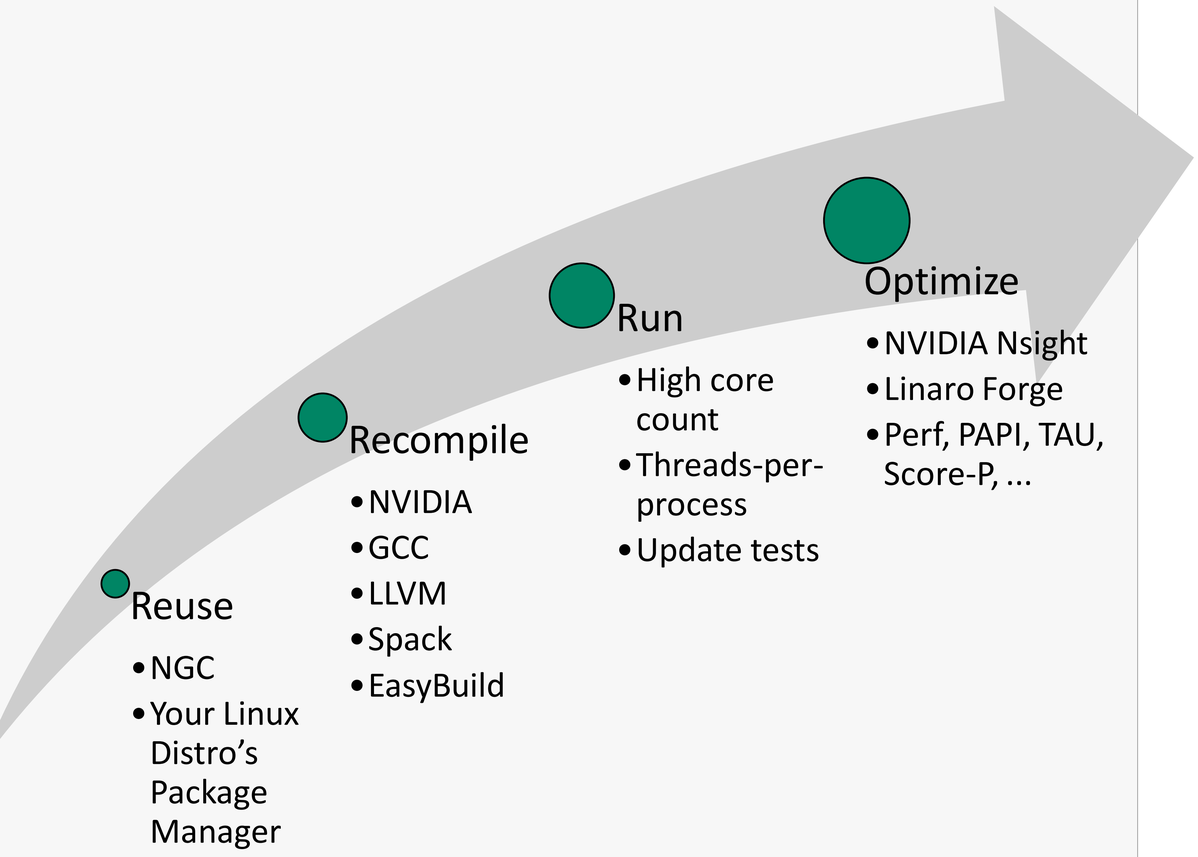
Grace Memory Subsystem

Per 72C Grace SoC

- Separate L1 data and instruction caches per core
  - L1 instruction memory system
    - 64KB, 4-way set associative, 64B cache line
  - L1 data memory system
    - 64KB, 4-way set associative, 64B cache line
- Private, unified data and instruction L2 cache per core
  - 1MB , 8-way set associative
- Scalable Coherency Fabric:
  Shared, uniform 117MB of L3 cache for entire chip.
- LPDDR5x: up to 500GB/s memory bandwidth
  - 120GB / 240GB capacity: 500 GB/s
  - 480GB capacity: 375 GB/s







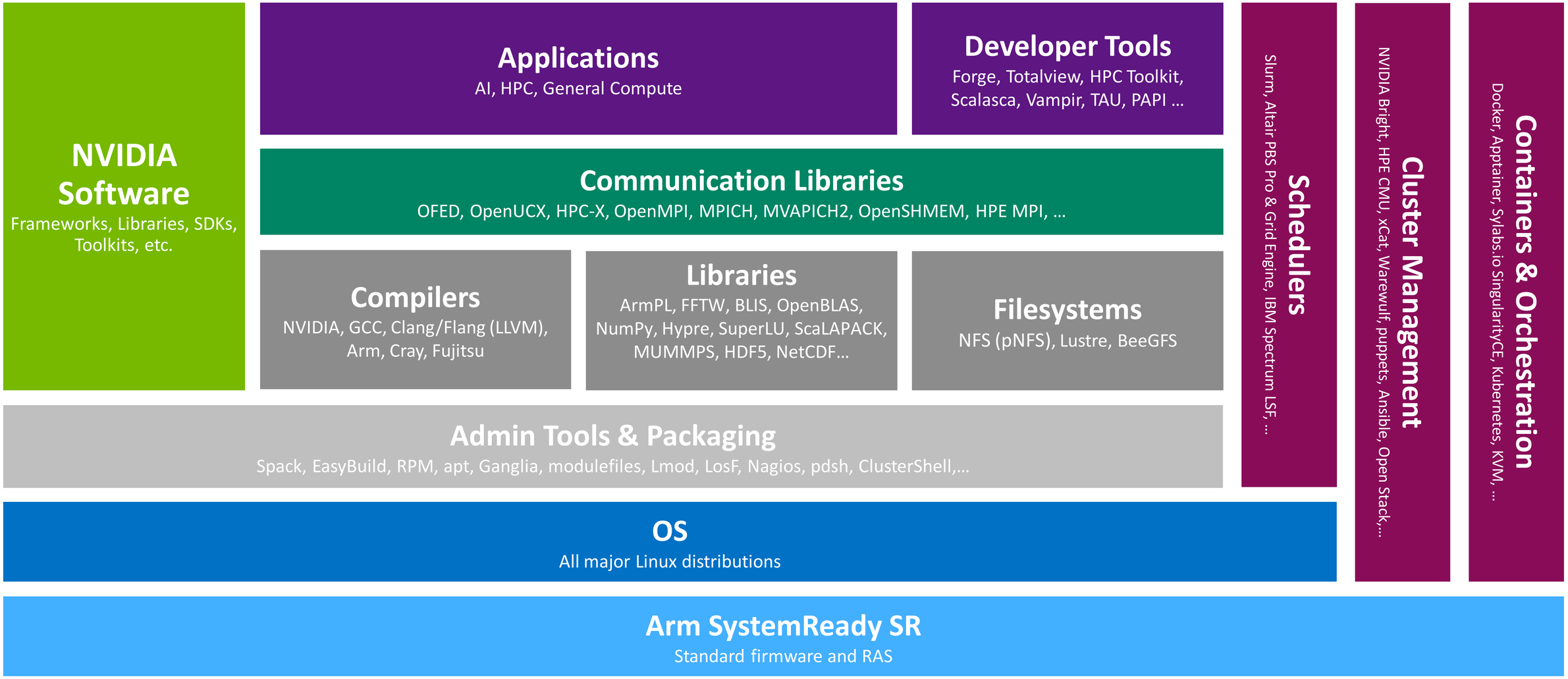
Expectation: It Just Works

Most applications will recompile easily and work "out of the box"

- Port away from a specific compiler
  - Make sure gcc, llvm can compile your code
- Vast majority of libraries are available and optimized for Arm
  - Optimized math libraries
- Check your dependencies
  - Recent improvements for Arm architecture
  - Build with recommended compiler

NVIDIA Grace HPC/AI Software Ecosystem

Full support for the broad Arm software ecosystem, both open source and commercial



Grace Documentation

https://docs.nvidia.com/grace

NVIDIA Docs Hub › NVIDIA Grace

Grace is NVIDIA's first datacenter CPU. All Grace products start with a system-on-chip (SoC) that comprise 72 high-performance Arm v9 cores and feature the NVIDIA-proprietary Scalable Coherency Fabric (SCF) network-on-chip for incredible core-to-core communication, memory bandwidth, and new GPU I/O capabilities. Grace provides a high-performance compute foundation in a low-power system-on-chip.

Built on standards such as Arm SystemReady SR, the Grace CPU is compatible with a wide variety of Arm-compatible operating systems, PCIe and USB peripherals, drivers, and application software already commonplace in existing Arm deployments— whether in the datacenter or the public cloud—including NVIDIA's CUDA and GPU driver ecosystem.

Grace is available in a variety of platforms for traditional and accelerated compute— including Grace Hopper products like GH200, which integrate a single 72-core Grace CPU with a H100 GPU on a new common memory subsystem to enable the next frontier of accelerated workloads—and the Grace CPU Superchip, which features a dual-CPU configuration with 144 cores, delivering the performance of today's highest-end conventional 2-socket CPU-based servers while improving datacenter efficiency by 2x.

MUST READ
Grace Performance Tuning Guide

Getting Started with NVIDIA Grace | Grace CPU Systems | Grace Hopper Systems

NVIDIA Grace Performance Tuning Guide

11/09/23

The Grace Performance Tuning guide provides best practices, software, and hardware configuration suggestions—

Grace OS Installation Guides

Grace systems can run a variety of Linux distributions that support the AArch64 architecture. With the proper kernel support and configurations, you can run one of the following Linux distros and take advantage

NVIDIA Grace Platform Support Software Patches and Configurations

12/18/23

Grace systems are composed of multiple hardware components that require support across different

Compilers

- Use a compiler that supports Neoverse V2

Compiler	Version ≧
GCC	12.2
LLVM (Clang)	16
NVIDIA HPC	23.3
Arm Compiler	23.04

- Check and update your compiler flags
- Use **`-mcpu=native`**
  - Can also use `-mcpu=neoverse-v2`, but `-mcpu=native` will “port forward”
- If possible use **`-Ofast`**
  - If fast math optimizations are not acceptable, use `-O3 –ffp-contract=fast`
  - For even more accuracy, use `-ffp-contract=off` to disable floating point operation contraction (e.g. FMA)
- Use `–flto` to enable link-time optimization
  - The benefits of link-time optimization vary from code to code, but can be significant
  - See e.g. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html for details

- Apps may need `-fsigned-char` or `-funsigned-char` depending on the developer’s assumption
- Fortran may benefit from `-fno-stack-arrays`
- Remember to check your dependencies
  - This is also a good opportunity to check for newer version with improved Arm Neoverse V2 / Grace support

NVIDIA.

`__atomic_add_fetch(&var, num, __ATOMIC_RELAXED)`

GCC 12.3 on Grace

Missing ISA Extensions: i8mm and bf16

```
        .arch armv9-a+crc
        .file   "foo.c"
        .text
        .align  2
        .global main
        .type   main, %function
main:
.LFB0:
        .cfi_startproc
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        str     wzr, [sp, 8]
        mov     w0, 1
        str     w0, [sp, 12]
        ldr     w1, [sp, 12]
        add     x0, sp, 8
        ldadd   w1, w0, [x0]
        mov     w0, 0
        add     sp, sp, 16
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 12.3.0"
        .section        .note.GNU-stack,"",@progbits
```

Atomic Add

-march=armv9-a

Correct instruction, limited ISA

Armv8: No SVE!

```
        .arch armv8-a
        .file   "foo.c"
        .text
        .global __aarch64_ldadd4_relax
        .align  2
        .global main
        .type   main, %function
main:
.LFB0:
        .cfi_startproc
        stp     x29, x30, [sp, -32]!
        .cfi_def_cfa_offset 32
        .cfi_offset 29, -32
        .cfi_offset 30, -24
        mov     x29, sp
        str     wzr, [sp, 24]
        mov     w0, 1
        str     w0, [sp, 28]
        ldr     w2, [sp, 28]
        add     x0, sp, 24
        mov     x1, x0
        mov     w0, w2
        bl      __aarch64_ldadd4_relax
        mov     w0, 0
        ldp     x29, x30, [sp], 32
        .cfi_restore 30
        .cfi_restore 29
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc
```

libgcc call

-mtune=neoverse-v2

Library call instead of atomic instruction, limited ISA

Correct ISA

```
.arch armv9-a+crc+profile+rng+memtag+sve2-bitperm+i8mm+bf16
.file   "foo.c"
.text
.align  2
.global main
.type   main, %function


.cfi_startproc
sub     sp, sp, #16
.cfi_def_cfa_offset 16
str     wzr, [sp, 8]
mov     w0, 1
str     w0, [sp, 12]
ldr     w1, [sp, 12]
add     x0, sp, 8
ldadd   w1, w0, [x0]
mov     w0, 0
add     sp, sp, 16
.cfi_def_cfa_offset 0
ret
.cfi_endproc

.size   main, .-main
.ident  "GCC: (GNU) 12.3.0"
.section        .note.GNU-stack,"",@progbits
```

Atomic Add

-mcpu=neoverse-v2 (or -mcpu=native)

Correct instruction, correct ISA

NVIDIA.

Performance analysis

Basics

- Know your code
  - Main characteristics of used algorithms
- Know your hardware
  - Capabilities that matter for your code
- Know your tools
  - Use tools (NVIDIA, Arm, 3rd party, open-source …)
  - Different tools have different strengths
  - Build a bouquet of tools to cover your needs
- Get a solid baseline performance (reproducible)
- For large application: find a suitable proxy
  - Need reasonably quick turnaround
  - Multiple passes might be needed with tools

Optimize

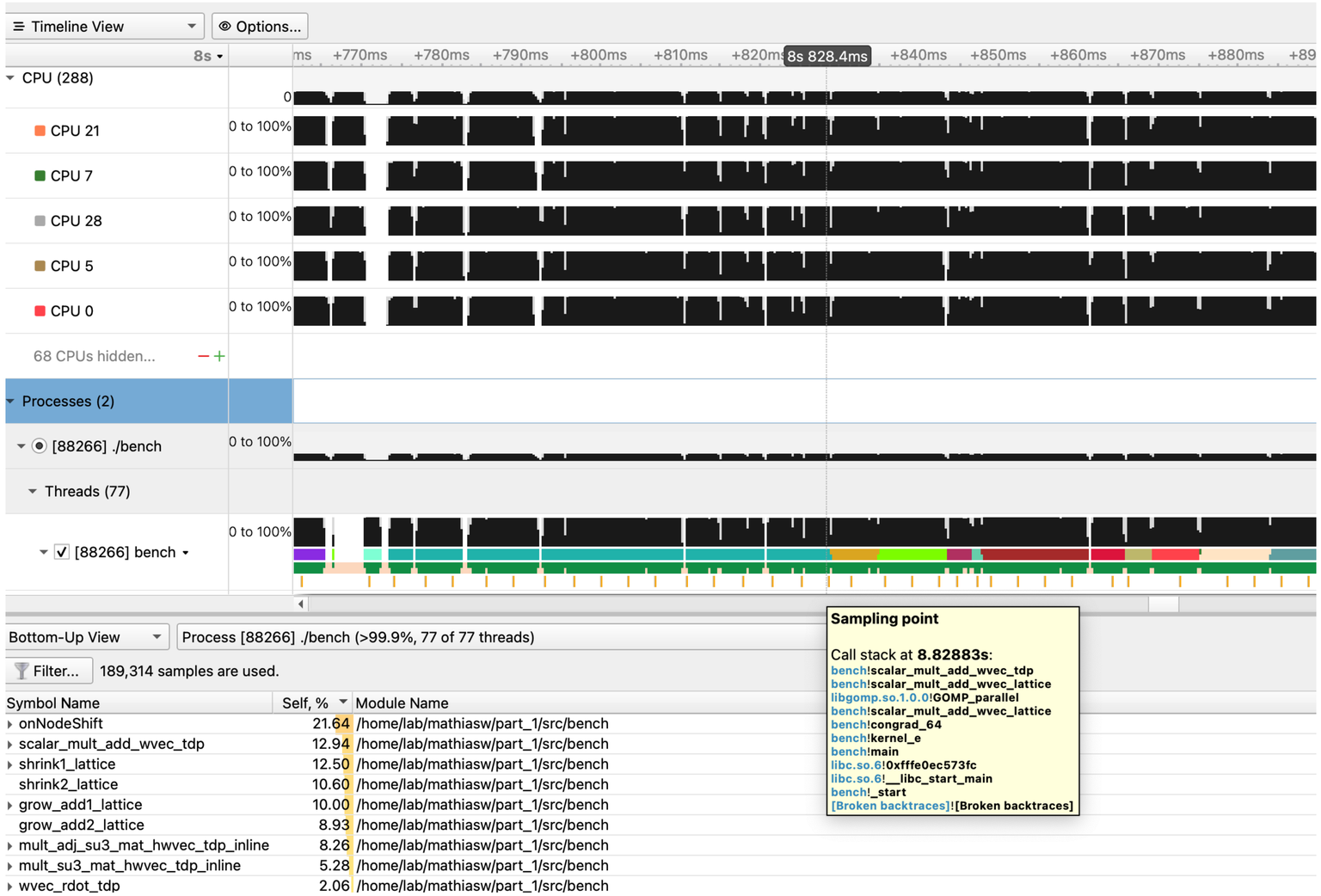
- Iterative process
  - Measure performance
- Identify hotspots
  - Is performance as expected or limited as expected?
  - If not, try to understand what is going on:
    - Compiler output
    - Performance tools
    - Experiments (modify code)
- Optimize
- Repeat

NVIDIA.

Nsight Systems

Timeline

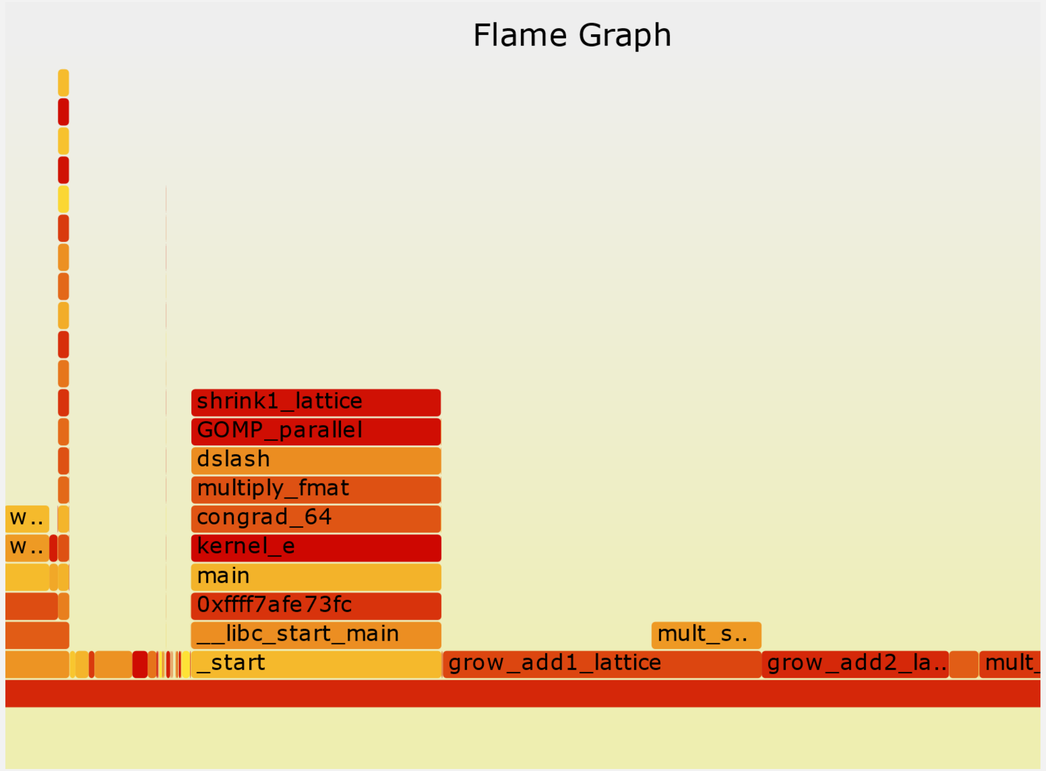
- Nsight Systems Documentation
- https://docs.nvidia.com/nsight-systems/UserGuide/index.html
  - → **CPU Profiling on Linux**
  - → **Profiling from the CLI**
- Collect profile for timeline offline
  - `nsys profile ./app`
  - https://docs.nvidia.com/nsight-systems/UserGuide/index.html#cli-profiling
- Open generated `*.nsys-rep` in Nsight Systems



Identify Hotspots

https://github.com/brendangregg/FlameGraph

- Using Nsight Systems
  - `Scripts/Flamegraph` directory in NSight Systems installation directory
  - `nsys profile –o report ./app`
  - `python3 stackcollapse_nsys.py report.nsys-rep | ./flamegraph.pl > result_flamegraph.svg`
- Using perf
  - `perf record -a -g ./app`
  - `perf script | stackcollapse-perf.pl > out.perf-folded`
  - `flamegraph.pl out.perf-folded > perf.svg`



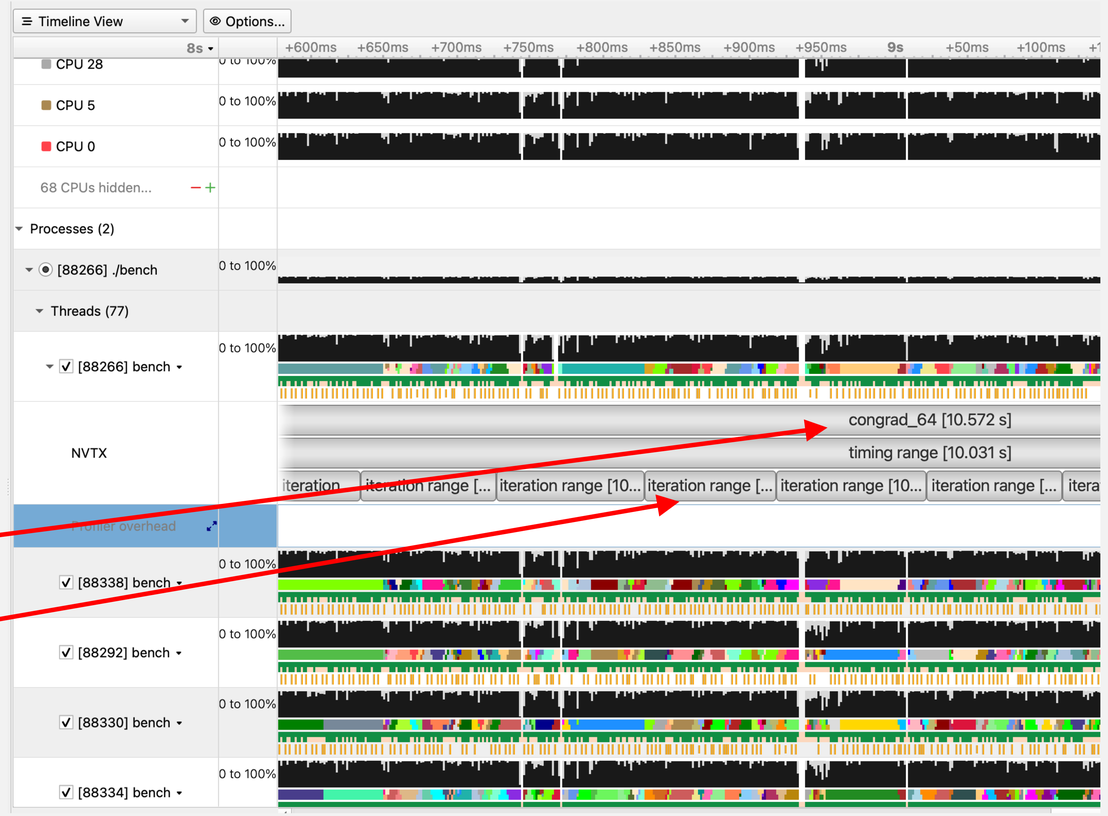
NVTX

Markup your code

- https://github.com/NVIDIA/NVTX/
  - Also part of CUDA Toolkit
  - Available for C, C++, Python, Fortran (NVHPC TK)

```
#include <nvtx3/nvToolsExt.h>

void congrad_64()
{
  nvtxRangePush(__func__); // Range around the whole function
  for (int i = 0; i < 6; ++i) {
    nvtxRangePush("loop range"); // Range for iteration
    // Do ab iteration
    nvtxRangePop(); // End the inner range
  }
nvtxRangePop(); // End the outer range
}
```



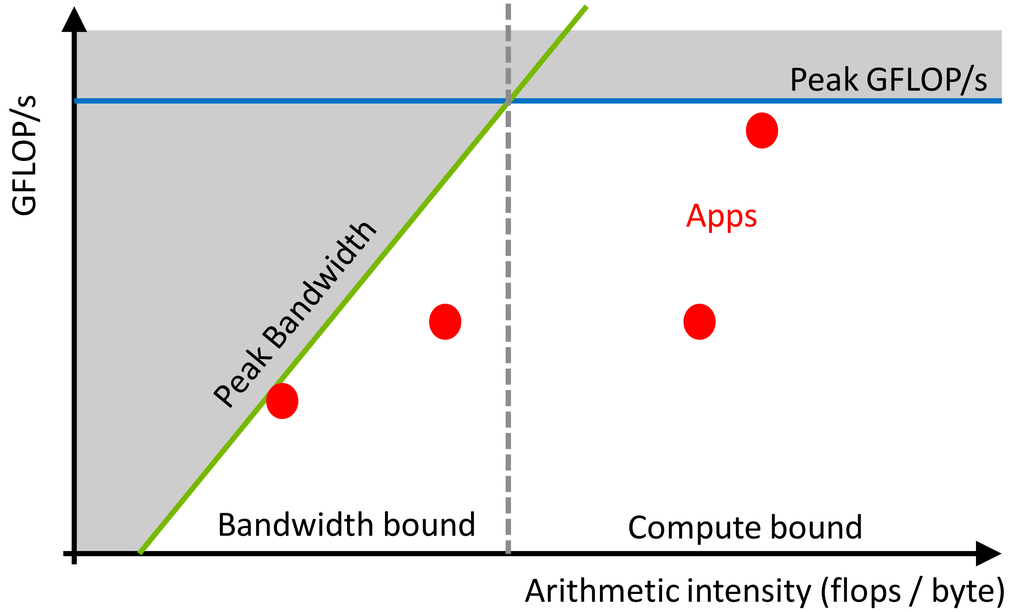
Roofline

- Consider compute and data throughput
- Hierarchical model considers also
  - cache bandwidths
  - Different peak GFLOP/Ss (precision, vectorization, FMA)
- Arithmetic intensity AI = (Flops/Byte)

- Define the roofline
  - either theoretical peaks for bandwidth and flops
  - Or measured values
    - (pro-tip: use same operation types and load/store ratios as your benchmark)
    - Use stream, likwid, NVIDIA Arm-Kernels,

- Know your workload:
  - Calculate expected AI from your code
  - Measure using tools

- Distance to the roofline indicates achieved performance level



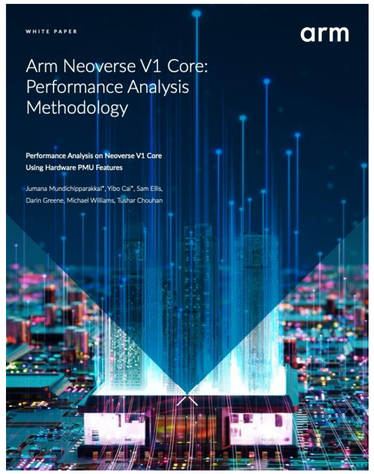
Top-Down Methodology

Workload characterization using PMU

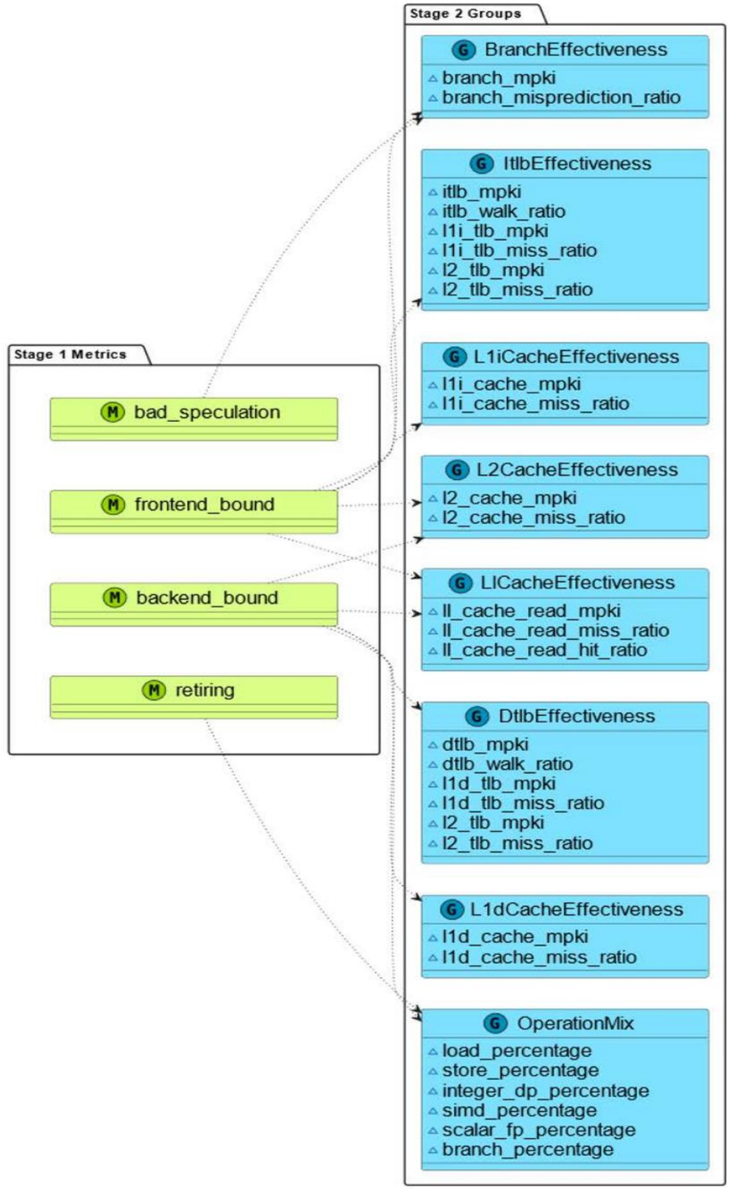
- Consider in core performance limiters
  - Frontend-bound
  - Backend-bound

- Arm Neoverse V1 Performance Analysis whitepaper
  https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/arm-neoverse-v1-top-down-methodology

  - Most of the content applies for Grace (Neoverse V2)
  - Stage 1 Metrics tell you about performance boundness
    - Bad speculation, frontend/backend bound or retiring
  - Stage 2 helps you identify why code is bound by X
    - L1 instruction/data cache, L2 cache, DTLB, etc.

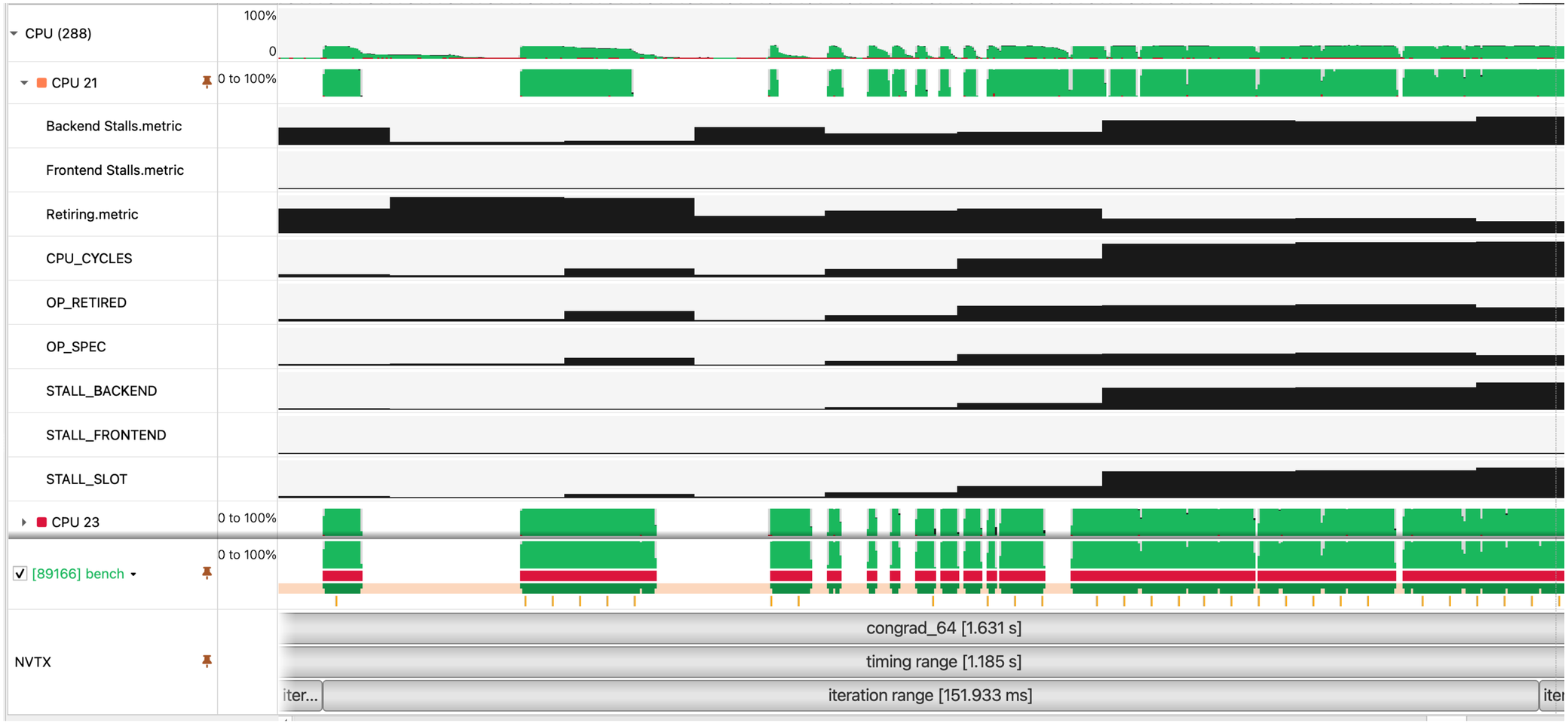


- Arm Neoverse V2 PMU Documentation
  https://developer.arm.com/documentation/PJDOC-1063724031-660094/1/?lang=en
- Arm Neoverse V2 Software Optimization Guide
  https://developer.arm.com/documentation/pjdoc466751330-593177/latest



Nsight Systems – Core metrics

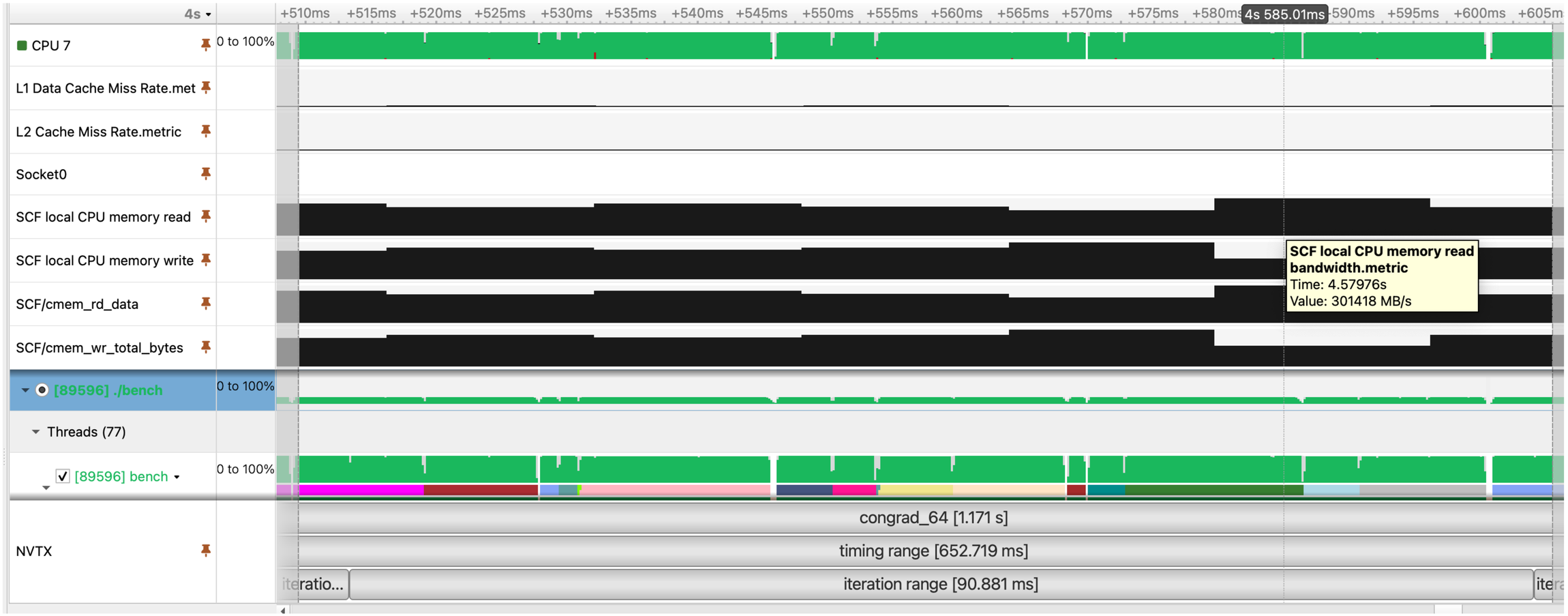
Collect Core performance metrics




```
nsys profile --cpu-core-metrics=help
```

Nsight Systems – Uncore metrics

Collect uncore performance metrics




```
nsys profile --cpu-socket-metrics=help
```


NVIDIA.

SIMD Programming Approaches

Follow these recommendations in order, e.g. prefer auto-vectorization over intrinsics

Compilers

- Auto-vectorization: NVIDIA, GCC, LLVM, ACfL, Cray…
- Compiler directives, e.g. OpenMP
  - `#pragma omp parallel for simd`
  - `#pragma vector always`

Libraries

- NVIDIA Math Libraries
- Arm Performance Library (ArmPL)
- Open Source Scientific Libraries (BLIS, FFTW, PETSc, etc.)

BLAS | LAPACK | PBLAS | SCALAPACK

TENSOR | SPARSE | RAND | FFTW

Intrinsics (ACLE)

- Arm C Language Extensions for SVE
- Arm Scalable Vector Extensions and application to Machine Learning

Assembly

- See SVE ISA Specification
- The Scalable Vector Extension for Armv8-A

```
// Complex dot product
// (a+ib)*(c+id) = (ac - bd) + i(ad + bc)
void complex_dot_product(Complex_t c[SIZE], Complex_t a[SIZE], Complex_t b[SIZE])
{
    uint32_t vl = svcntw();

    svbool_t p32_all = svptrue_b32();

    for (int i=0; i<SIZE; i+=vl) {
        svfloat32x2_t va = svld2(p32_all, (float32_t*)&a[i]);
        svfloat32x2_t vb = svld2(p32_all, (float32_t*)&b[i]);
        svfloat32x2_t vc = svld2(p32_all, (float32_t*)&c[i]);

        vc.v0 = svmla_m(p32_all, vc.v0, va.v0, vb.v0); //c.re += a.re * b.re
        vc.v1 = svmla_m(p32_all, vc.v1, va.v1, vb.v0); //c.im += a.im * b.re
        vc.v0 = svmls_m(p32_all, vc.v0, va.v1, vb.v1); //c.re -= a.im * b.im
        vc.v1 = svmla_m(p32_all, vc.v1, va.v0, vb.v1); //c.im += a.re * b.im

        svst2(p32_all, (float32_t*)&c[i], vc);
    }
}
```


NVIDIA.

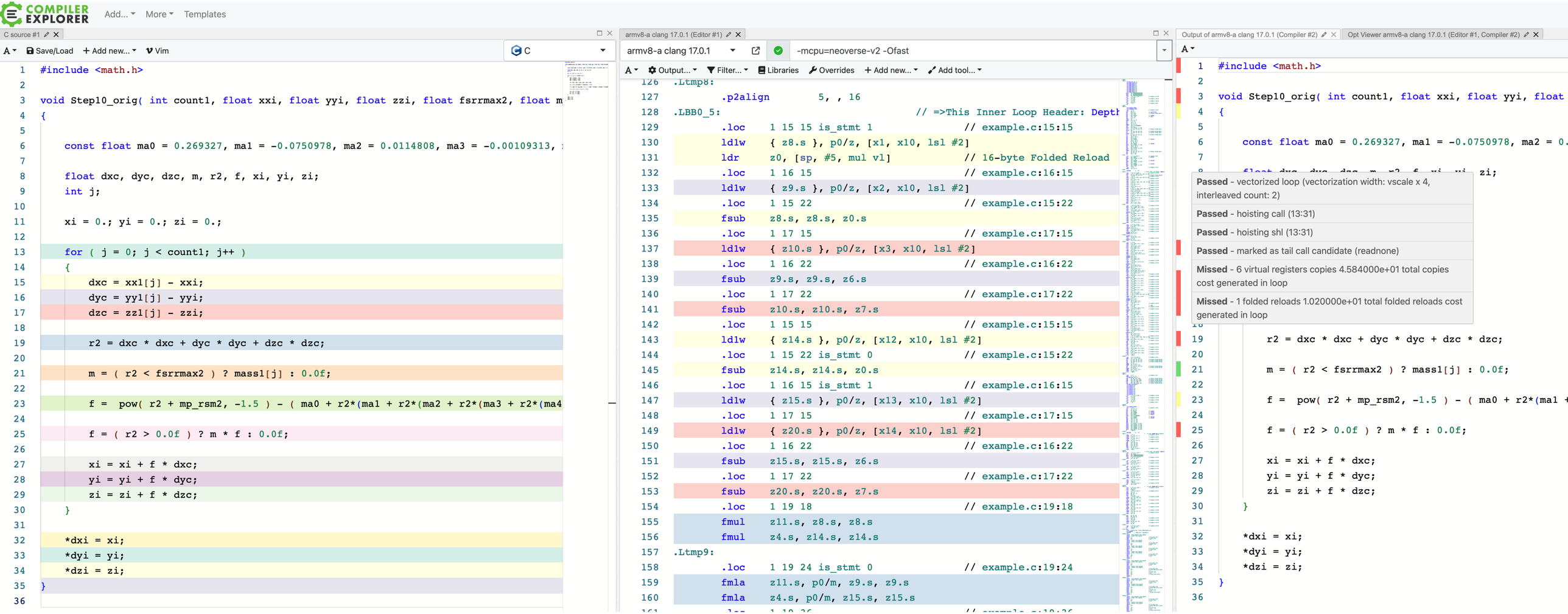
Compiler auto-vectorization

- Typically enabled at higher optimization levels
  - with some more aggressive options at `–Ofast` (e.g. reductions)
- Diagnostic flags to allow you to see what the compiler does and does not vectorize
  - LLVM:
    - `-Rpass( | -missed | -analysis) =loop-vectorize`
  - GCC:
    - `-fopt-info-vec-( optimized | missed| all | note)`
  - NVIDIA compilers:
    - `-Minfo=vect`
- Look at output and generated code → Compiler explorer / Godbolt
- Use restrict pointers in C / C++ where applicable
- Pragmas to guide compiler (depends on compiler used)
  - `#pragma GCC ivdep`
  - `#pragma clang loop vectorize`
  - `#pragma omp simd`

NVIDIA.

Compiler explorer – understand what your compiler does

https://godbolt.org



Code from HACCmk benchmark from the Coral benchmark suite https://asc.llnl.gov/coral-benchmarks

Applications that use Math Libraries

Several library options to choose from

- Prefer Netlib BLAS/LAPACK and FFTW interfaces
  - Building on these interfaces enables compatibility

- **NVPL**
  - ```
    gcc -DUSE_CBLAS -ffast-math -mcpu=native -O3 \
        -I/PATH/TO/nvpl/include \
        -L/PATH/TO/nvpl/lib \
        -o mt-dgemm.nvpl mt-dgemm.c \
        -lnvpl_blas_lp64_gomp
    ```

- **ArmPL**
  - ```
    gcc -DUSE_CBLAS -ffast-math -mcpu=native -O3 \
        -I/opt/arm/armpl-23.10.0_Ubuntu-22.04_gcc/include \
        -L/opt/arm/armpl-23.10.0_Ubuntu-22.04_gcc/lib \
        -o mt-dgemm.armpl mt-dgemm.c \
        -larmpl_lp64
    ```

- **ATLAS**, **OpenBLAS**, **BLIS**, … Community supported with *some* optimizations for Neoverse V2.

NVIDIA Performance Libraries

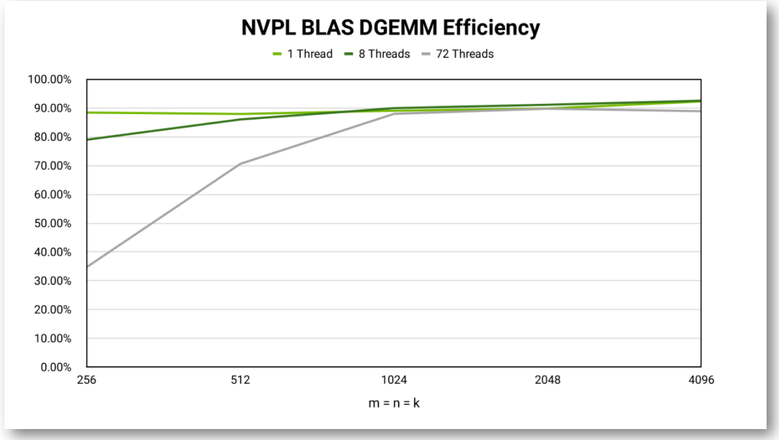
Optimized math libraries for NVIDIA CPUs

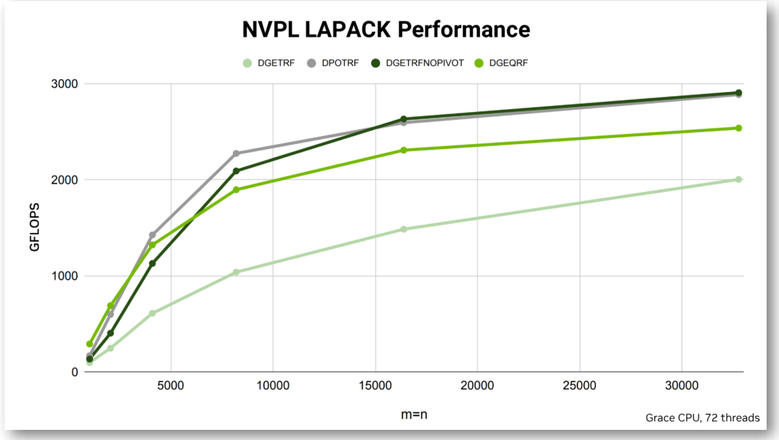
- Easily port applications to NVIDIA's datacenter CPUs
- Drop-in replacement for any math library implementing standard interfaces (e.g. Netlib, FFTW)
- New interfaces for high-performance libraries
- www.developer.nvidia.com/nvpl

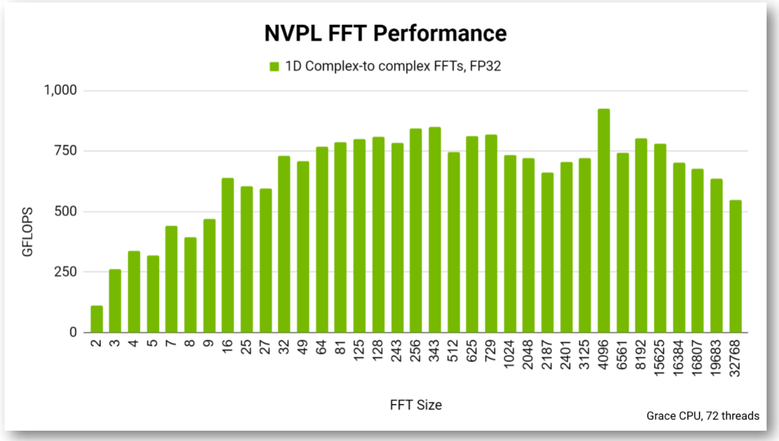
BLAS	LAPACK	PBLAS	SCALAPACK
TENSOR	SPARSE	RAND	FFT

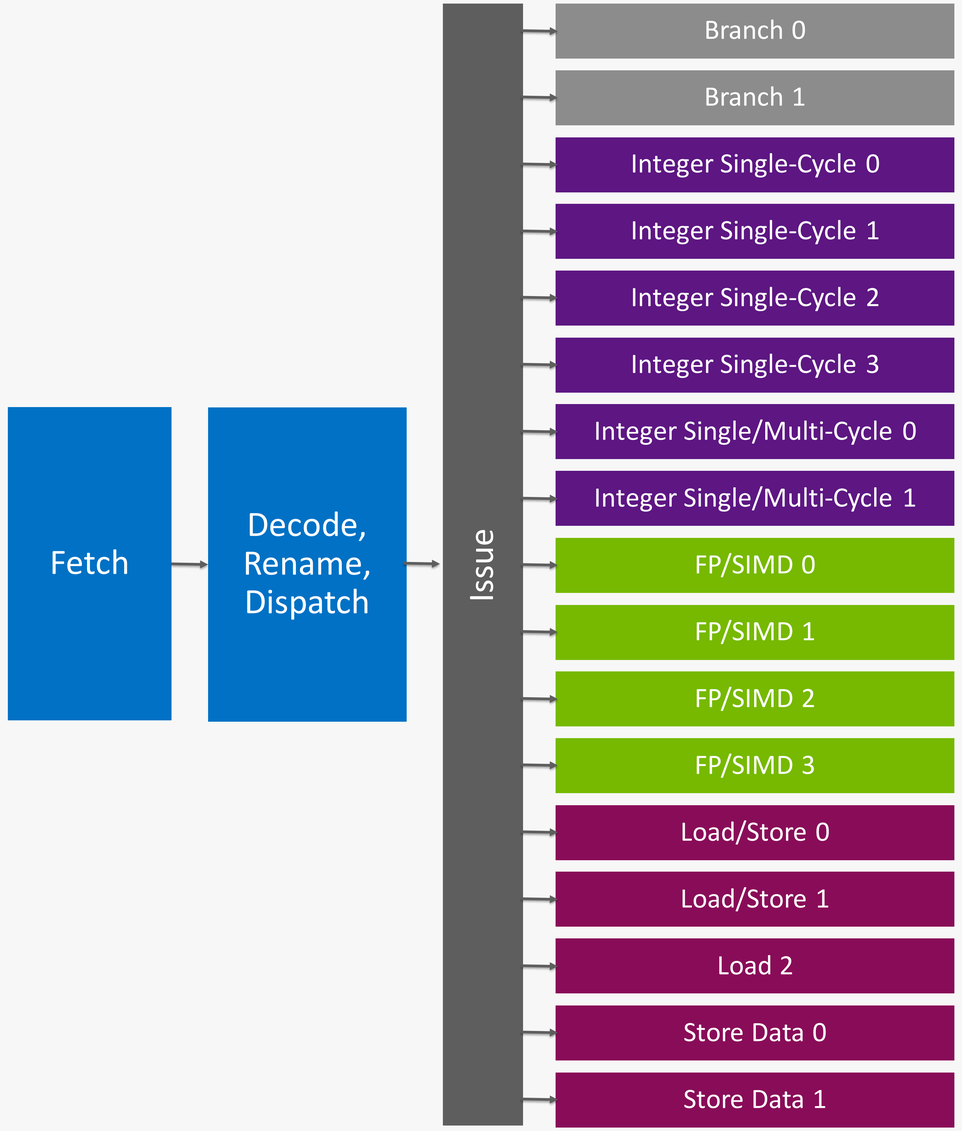
Download Now

www.developer.nvidia.com/nvpl









SIMD in NVIDIA Grace

NEON and SVE

- 4x128b SIMD units
- Each SIMD unit can retire **NEON** or **SVE2** instructions
- SVE2 and NEON have the **same peak performance**

- Compiler will choose when auto-vectorizing
- Library developers will choose
- Ok to use existing NEON code
- SVE2 can vectorize more complex codes and supports more data types than NEON
  - Use it and update NEON when you can express more

Assembly and Vector Intrinsics

If you need to

- Arm provides Arm C Language Extensions (ACLE)
  https://developer.arm.com/Architectures/Arm%20C%20Language%20Extensions

- Arm SVE documentation https://developer.arm.com/Architectures/Scalable%20Vector%20Extensions
  - Arm SVE Optimization guide https://developer.arm.com/documentation/102699/latest/
  - Also helps to better understand performance even if you not intend to write SVE yourself

- For a quick fix of existing x86: use a drop-in header-based intrinsics translator
  - SIMD Everywhere (SIMDe): https://github.com/simd-everywhere/simde
  - SSE2NEON: https://github.com/DLTcollab/sse2neon
  - Demonstration: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41702/
- Follow Arm's documentation on rewriting x86 intrinsics for incrementally improving performance
  - 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]

NVIDIA.

Summary

Main takeaways

- Recompile your code with current compiler and correct compiler flags
  - Don’t forget your dependencies
- Use Grace optimized math libraries
  - NVPL is preferred
- Understand your performance expectations and bottlenecks
- Use tools
  - NVIDIA Nsight Sytems and 3rd party
- Optimization is an iterative project
  - Optimize your hotspots, but only if there is potential

Resources

TO DO