NVPL BLAS Overview

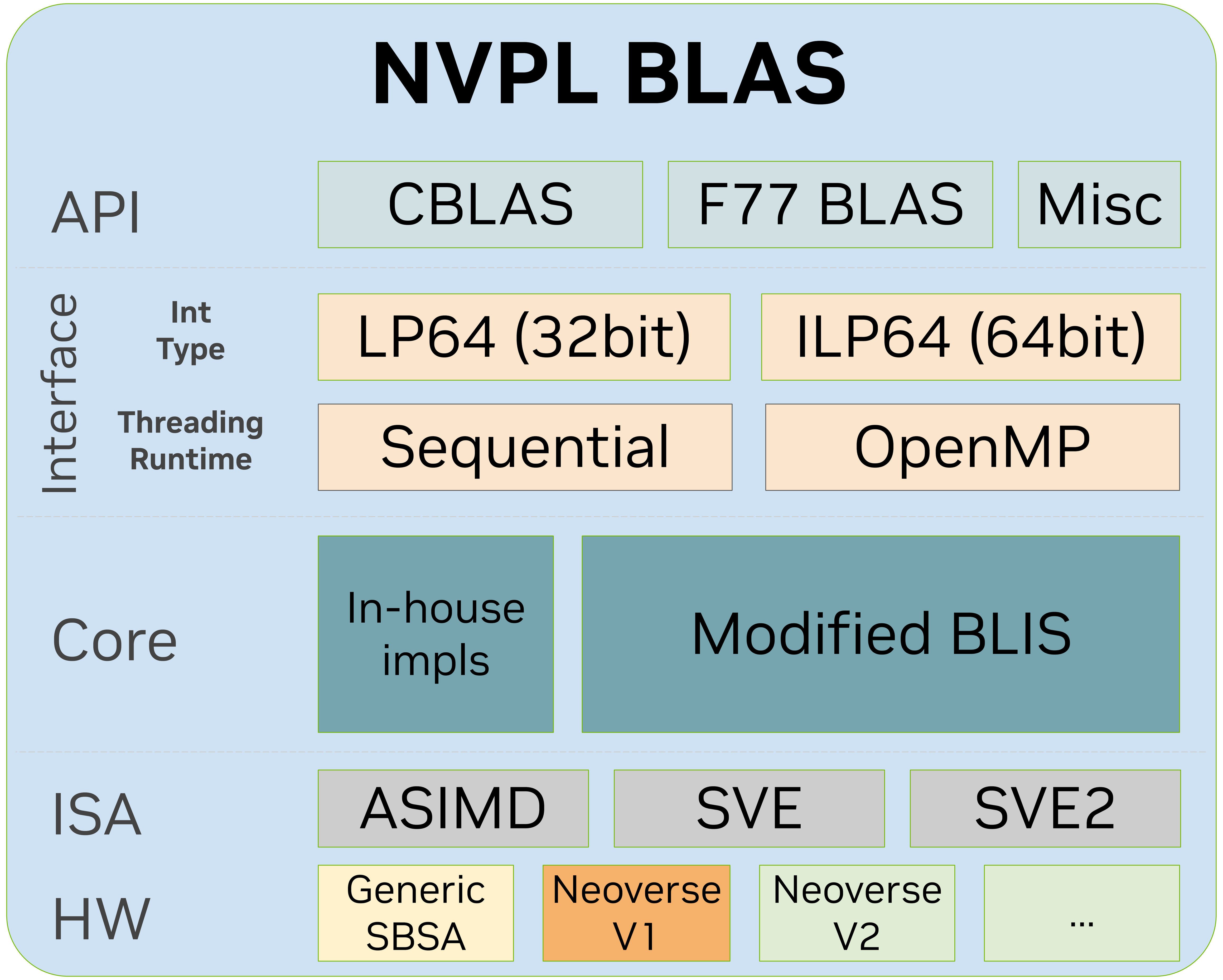
Evarist Fomenko | BLIS Retreat. Sep 26-27, 2024

Agenda

- Overview
- Library Architecture
- Optimizations
  - GEMM u-kernel
  - Synchronization Overheads
  - Framework Overheads
- Conclusions

Overview

- Part of Nvidia Performance Libraries
  - Initial release NVPL 23.11 (Nov'23)
  - Latest release NVPL 24.7 (Jul'24)
- API
  - Standard Netlib API: F77 and C
  - Extra: batch GEMM, threading control, verbosity, …
- Interface
  - LP64 and ILP64 interfaces
    - Switched with `-DNVPL_ILP64` compiler flag
  - Different thread runtime: sequential and OpenMP (any vendor)
- Implementation
  - Runtime dispatching based on CPU architecture
- Inspiration: BLIS, Intel MKL, ArmPL




```
NVPL_BLAS_VERBOSE: NVPL BLAS version 0.1.0
NVPL_BLAS_VERBOSE: Platform: Neoverse V2, cores:72 sve_width:128. Cache: L1:64 KB (cl:64 ways:4 sets:256) L2:1024 KB
NVPL_BLAS_VERBOSE: dgemm_(N,N,128,256,123,2,0xfffdddb62000,128,0xfffdddb12000,123,-1,0xfffdddac2000,128) time_us:4742
NVPL_BLAS_VERBOSE: cscal_(1024,0xaaab41ed69e0,0xaaab41ee5000,1) time_us:5.76 int:lp64 max_nthr:72 tid:fffdddeb0020
```

Library Architecture

Name: libnvpl_blas_{,i}lp64_{seq,gomp}.so
Symbol mangling: nvpl_blas_* (except for BLAS)
Responsibilities:

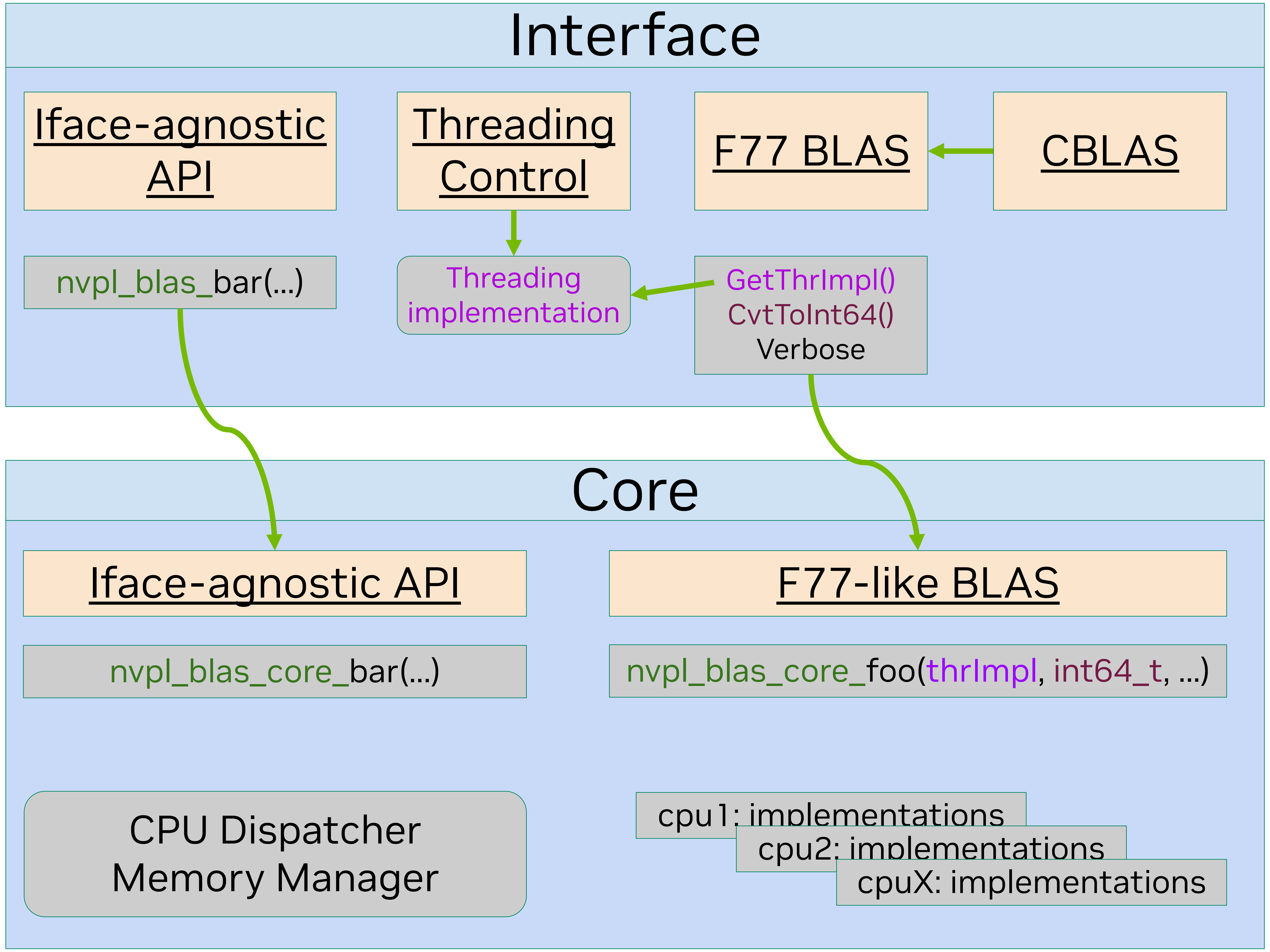
- Int32 / Int64 APIs
  - Switched with `-DNVPL_ILP64`
- Threading RT
- Verbose

Depends: core, threading rt

Name: libnvpl_blas_core.so
Symbol mangling: nvpl_blas_core_*
Responsibilities:

- Actual BLAS implementations (all int64)
- CPU dispatcher
- Memory manager
- Other service functionality

Depends: libc, lpthread, lm



Optimizations

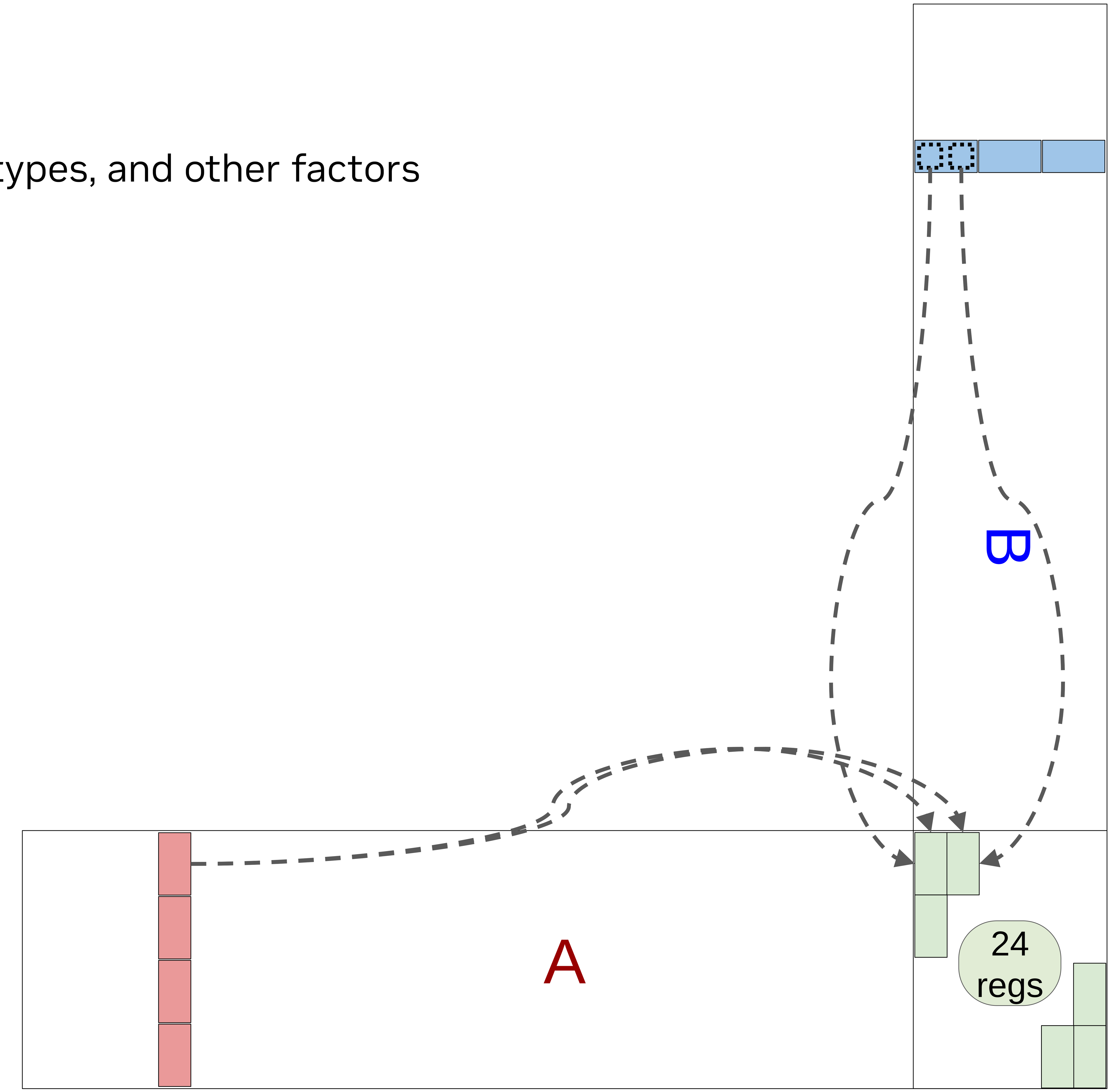
Grace µ-kernel

- Very similar to a typical Armv8-A kernel, e.g. Cortex A57
- µ-kernels are written in assembler
  - Statically generated using Python to easily adjust blocking, data types, and other factors
- Most gains come from proper memory prefetches
  - Upside is around 5-10%

```
# MR = 8    (alternative: 6)
# NR = 6    (alternative: 8)
# KC = 512
µdgemm_MRxNRxKC():
    VC(0:6,0:4) = 0

   .L_loop_k
   for uk = 0 .. 4:
       LD1 {VA(0).2d - VA(3).2d}, [ptrA], #64
       LD1 {VB(0).2d - VB(2).2d}, [ptrB], #48

       for n = 0 .. 6:     # NR
           for m = 0 .. 4: # MR/2
               FMLA VC(n, m).2d, VA(m).2d, VB(n / 2).d[n % 2]

   SUB xk, xk, 4
   JNZ xk, .L_loop_k

   C_r[:, :] = alpha × VC(:, :) + beta × C_r[:, :]
```

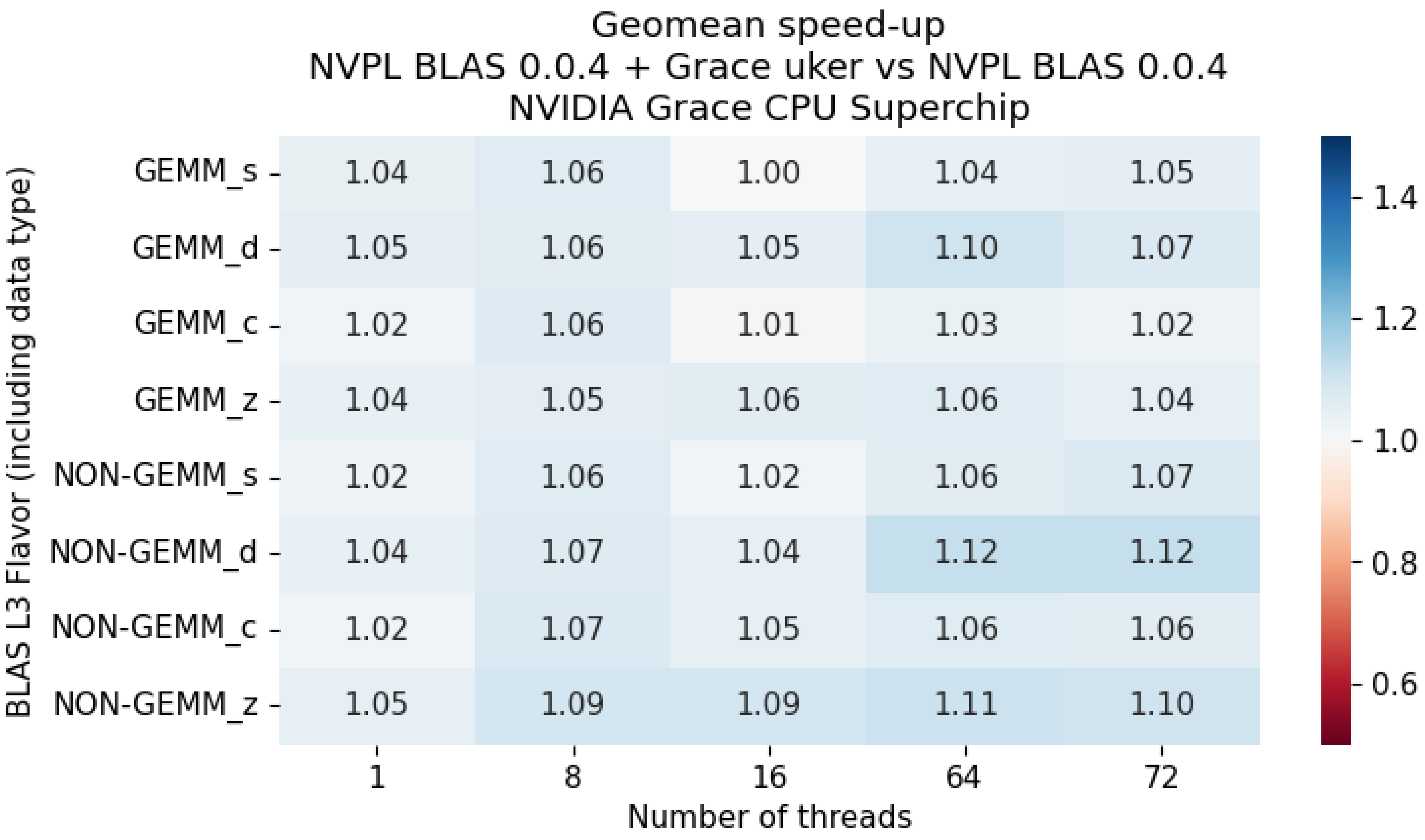
4+3 loads

24 FMAs



Optimizations

Grace μ-kernel

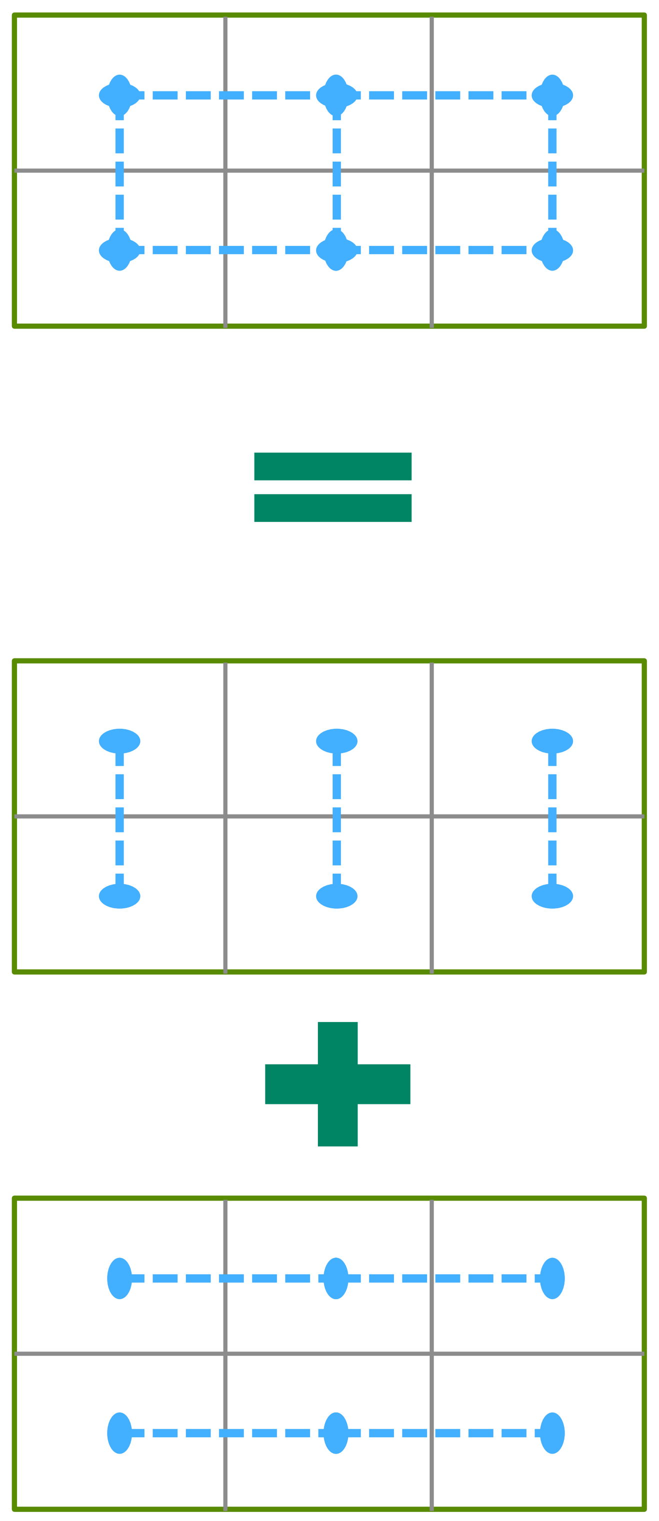


BLAS L3 Flavor	1	8	16	64	72
GEMM_s	1.04	1.06	1.00	1.04	1.05
GEMM_d	1.05	1.06	1.05	1.10	1.07
GEMM_c	1.02	1.06	1.01	1.03	1.02
GEMM_z	1.04	1.05	1.06	1.06	1.04
NON-GEMM_s	1.02	1.06	1.02	1.06	1.07
NON-GEMM_d	1.04	1.07	1.04	1.12	1.12
NON-GEMM_c	1.02	1.07	1.05	1.06	1.06
NON-GEMM_z	1.05	1.09	1.09	1.11	1.10

Optimizations

Reducing Synchronization Overheads

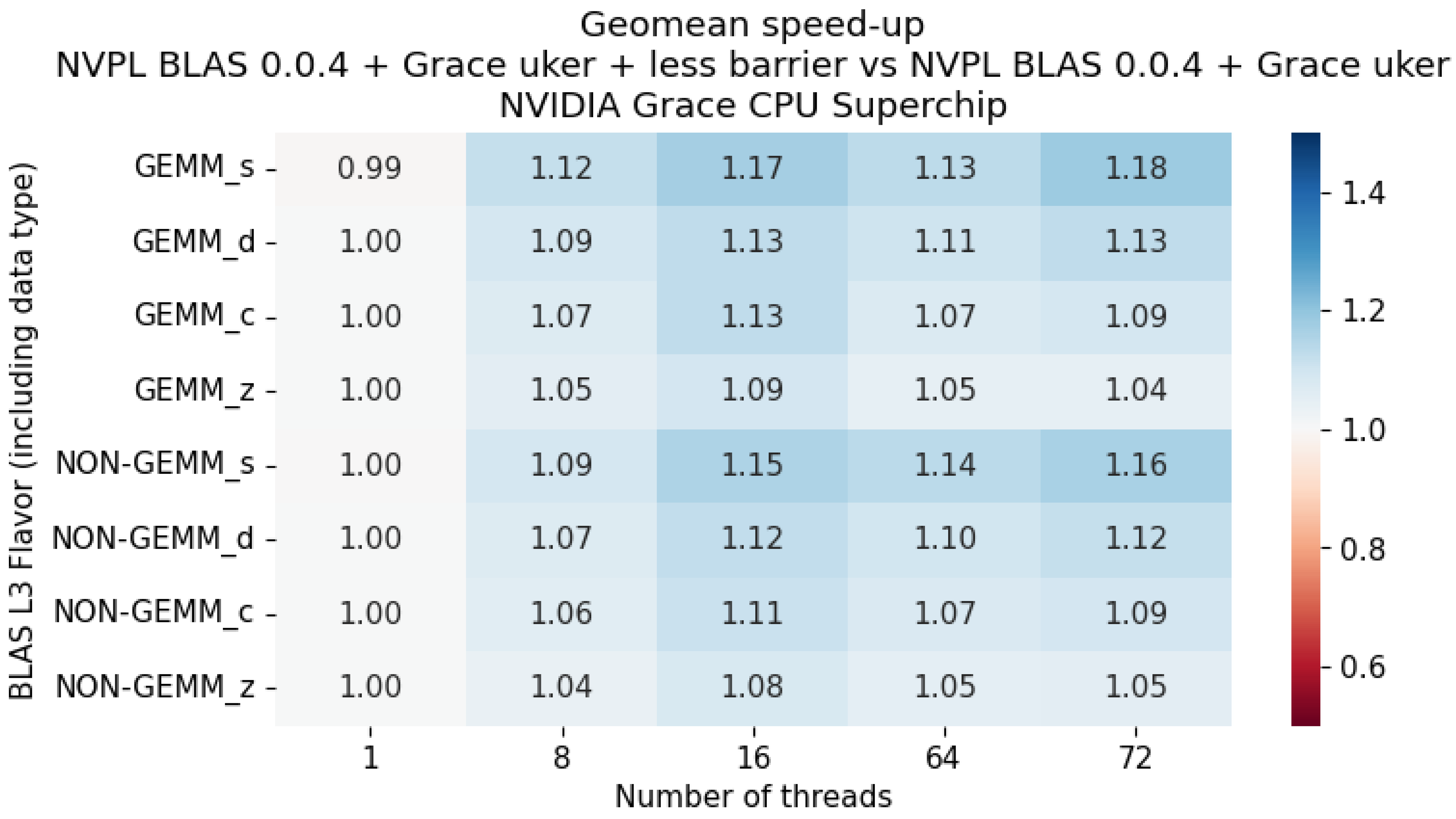
- Asynchronous creation of thread info (thrinfo_t) tree
- Faster broadcast
  - Avoid 1 of 2 barriers by altering the shared space for sent object
- Replace 1 barrier for N*M threads with N independent barriers for M threads
  - At matrix B packing, as there will be M barriers for N threads at matrix A packing
- Replace BLIS allocator with glibc malloc
  - Consider improving/reimplementing the allocator or using something like tcmalloc/jemalloc/etc.
- Some implementation rework to avoid redundant barriers



```
Execution
l3_int() → gemm_blk_var2() →
l3_int() → gemm_blk_var3() →
l3_int() → [barrier()] → l3_pack_b() → [barrier()] →
l3_int() → gemm_blk_var1() →
l3_int() → [barrier()] → l3_pack_a() → [barrier()] →
l3_int() → gemm_ker_var2() →
→
µgemm()
```

Optimizations

Reducing Synchronization Overheads

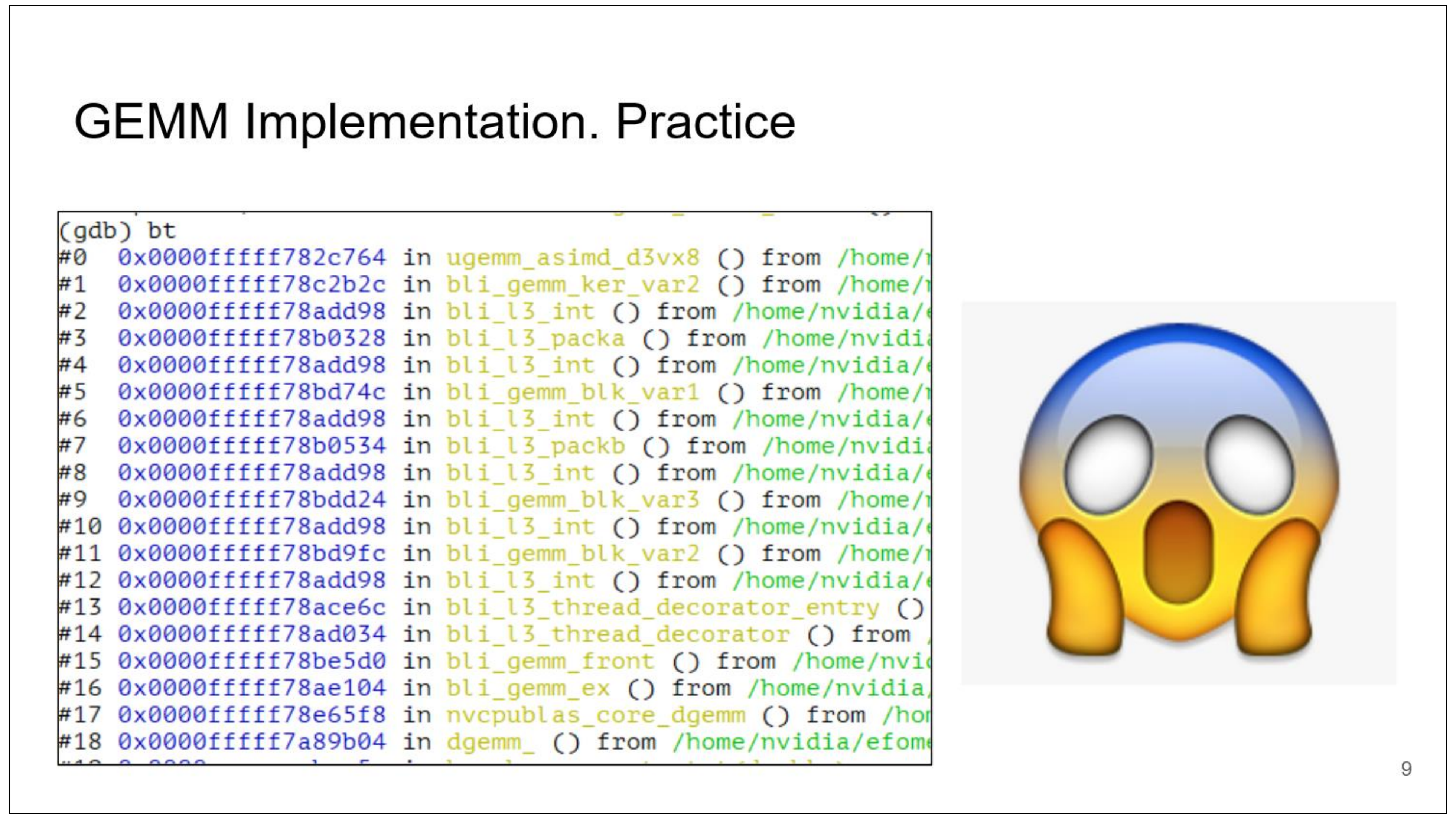


Optimizations

Reducing Framework Overheads

Replace BLIS abstraction for computing BLAS Level 3 operations for GEMM with a direct implementation

- This gives the best performance.
- Driver is only 500 lines of code.
- Tried to optimize the abstractions.
  - No luck up until now.
  - Might need to do the same for the remaining BLAS Level 3 ops.



	Before	After
64^3 1 thr	86.8%, 33.6 Gflop Overhead Symbol 77.00% [.] bli_dgemm_neoverse_v2_asimd_4vx6 5.82% [.] bli_dpackm_nrxk_neoverse_v2_ref 3.96% [.] bli_dpackm_mrxk_neoverse_v2_ref	97.4%, 38.5 GFlops Overhead Symbol 85.88% [.] bli_dgemm_neoverse_v2_asimd_4vx6 6.91% [.] bli_dpackm_nrxk_neoverse_v2_ref 4.61% [.] bli_dpackm_mrxk_neoverse_v2_ref
512^3 1 thr	99.4%, 46.6 GFlop Overhead Symbol 96.05% [.] bli_dgemm_neoverse_v2_asimd_4vx6 2.15% [.] bli_dpackm_mrxk_neoverse_v2_ref 1.21% [.] bli_dpackm_nrxk_neoverse_v2_ref	99.8%, 47.0 GFlops Overhead Symbol 95.81% [.] bli_dgemm_neoverse_v2_asimd_4vx6 2.73% [.] bli_dpackm_mrxk_neoverse_v2_ref 1.23% [.] bli_dpackm_nrxk_neoverse_v2_ref
512^3 16 thr	92.9%, 689 GFlop Overhead Symbol 88.98% [.] bli_dgemm_neoverse_v2_asimd_4vx6 2.43% [.] bli_dpackm_mrxk_neoverse_v2_ref 1.82% [k] 0xffffc0e00858e3b8 1.47% [.] bli_dpackm_nrxk_neoverse_v2_ref 0.80% [k] 0xffffc0e0084d0ca4 0.62% [.] bli_thrcomm_barrier_atomic	95.6%, 713 GFlops Overhead Symbol 92.08% [.] bli_dgemm_neoverse_v2_asimd_4vx6 2.09% [.] bli_dpackm_mrxk_neoverse_v2_ref 1.40% [.] bli_dpackm_nrxk_neoverse_v2_ref 1.23% [k] 0xffffc0e00858e3b8 0.57% [k] 0xffffc0e0084d0ca4

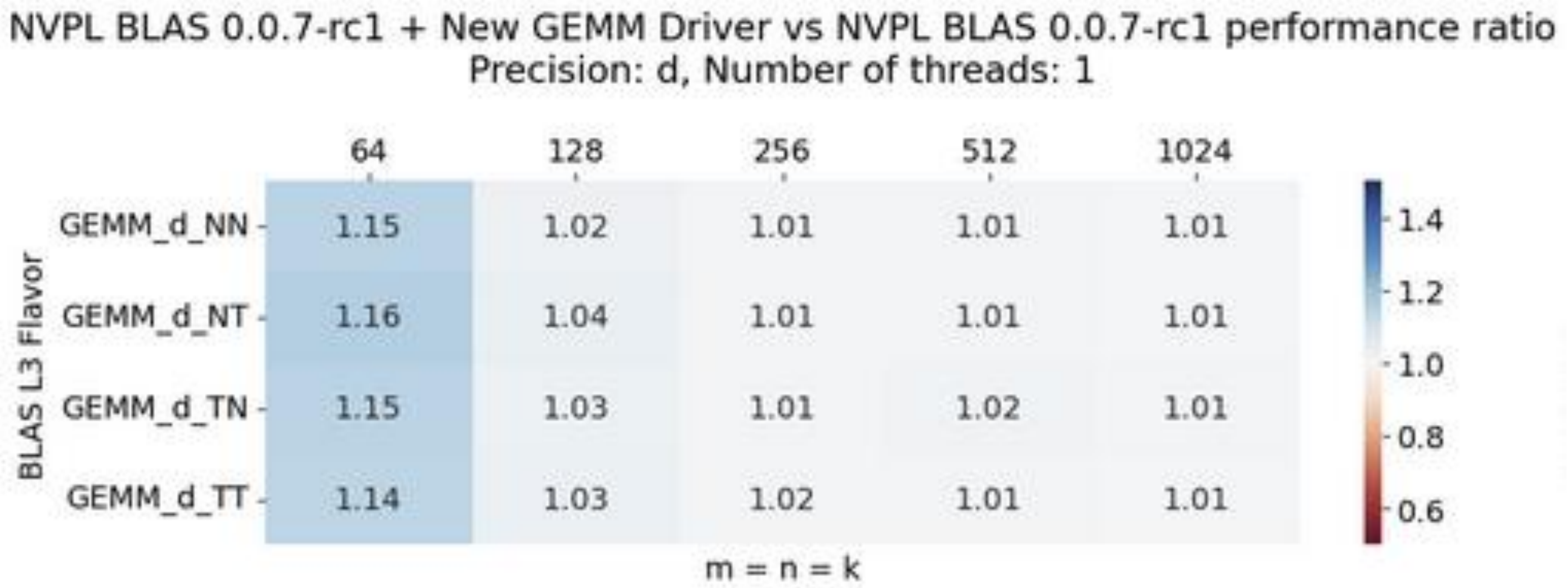
Optimizations

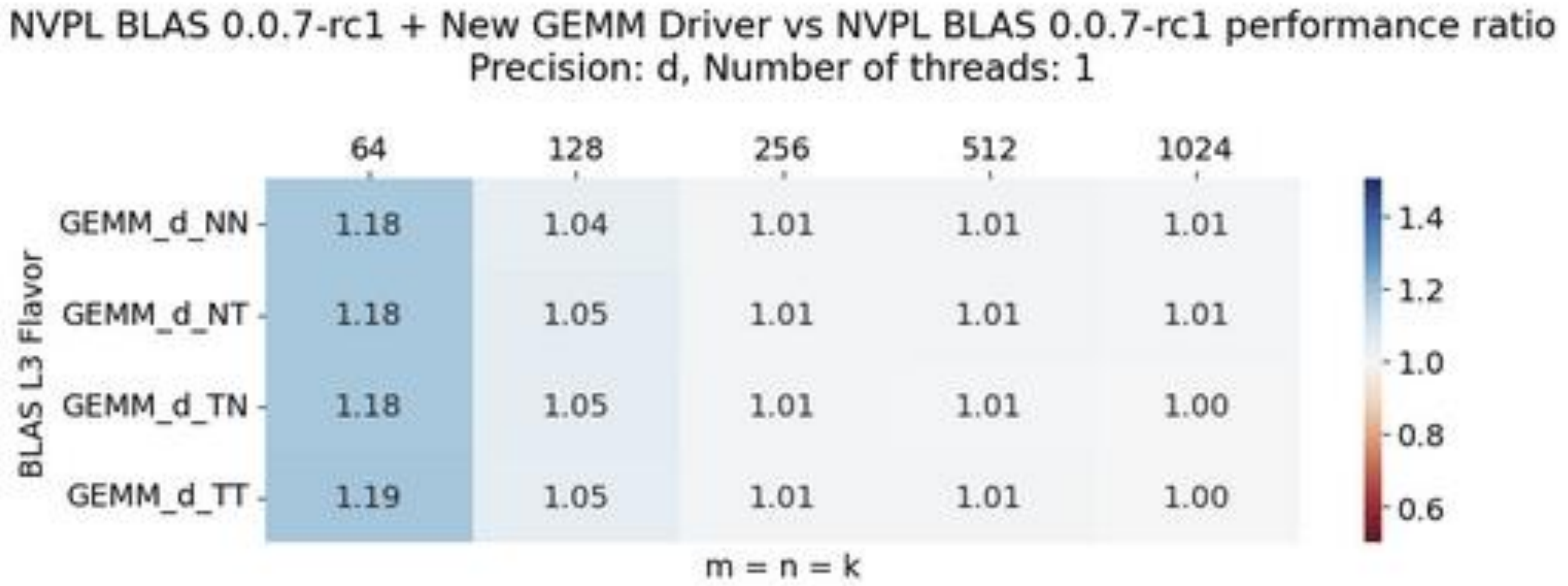
Reducing Framework Overheads

Nvidia Grace CPU Superchip

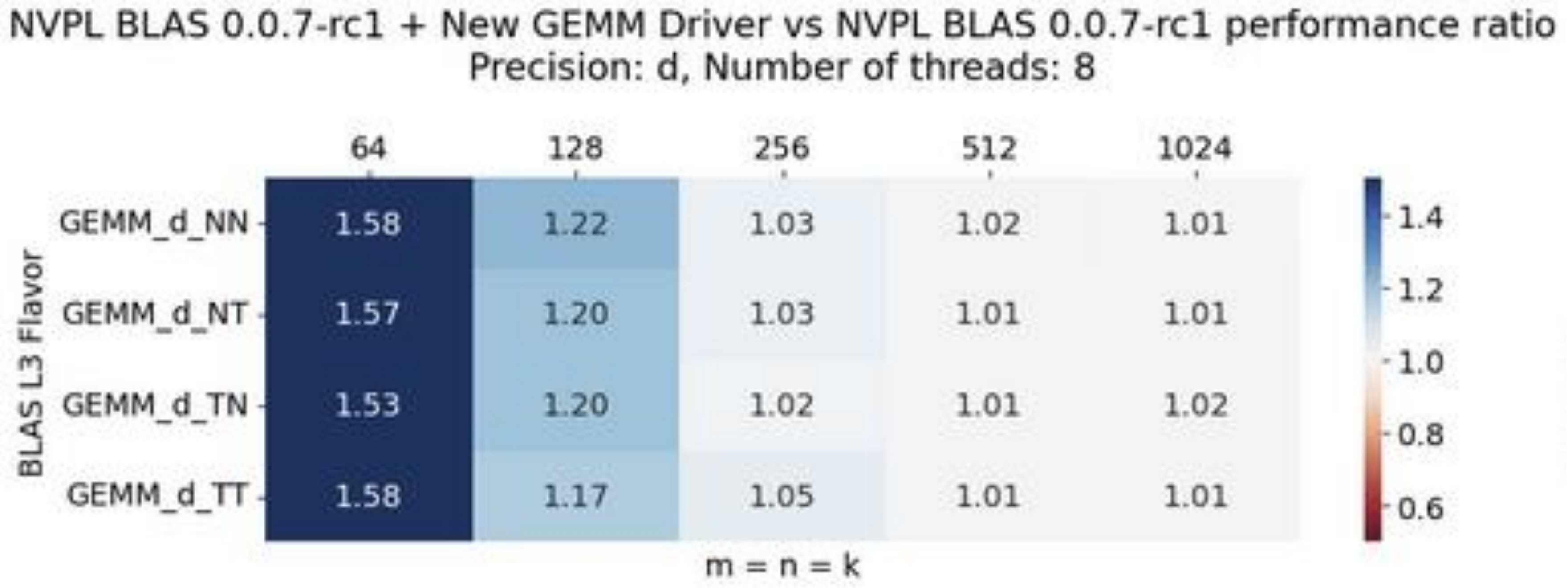
AWS Graviton 3

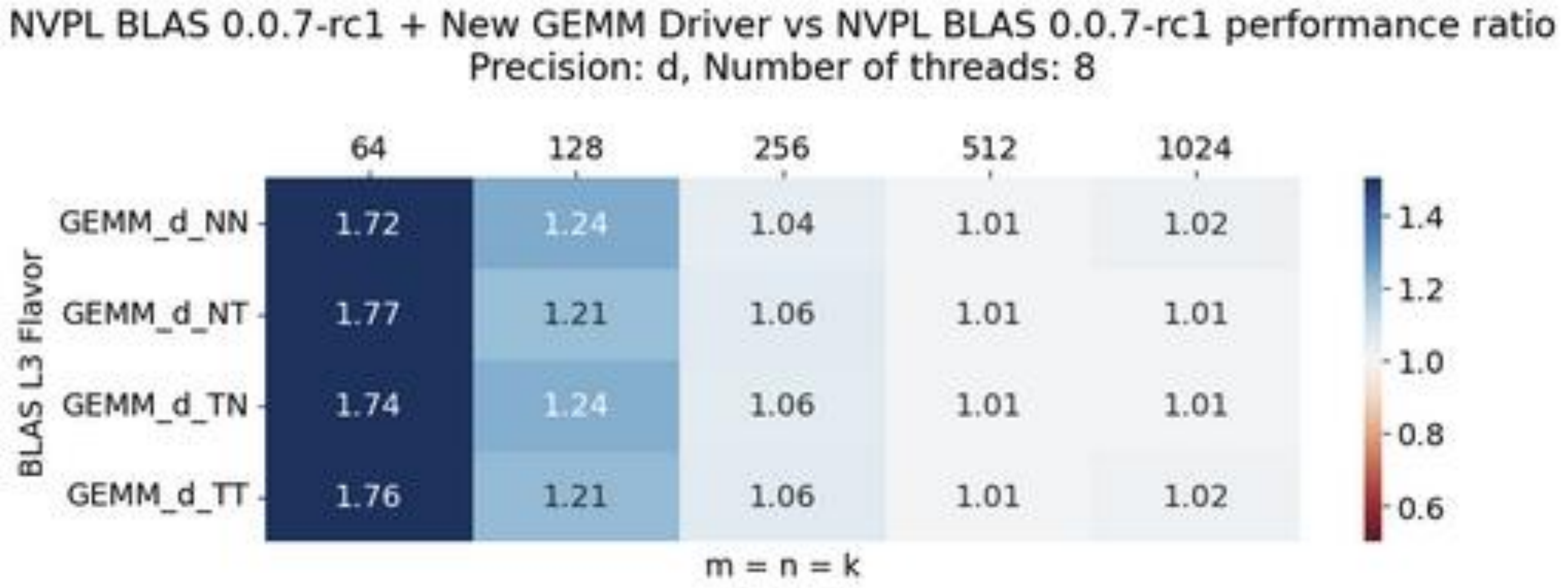
1 thread



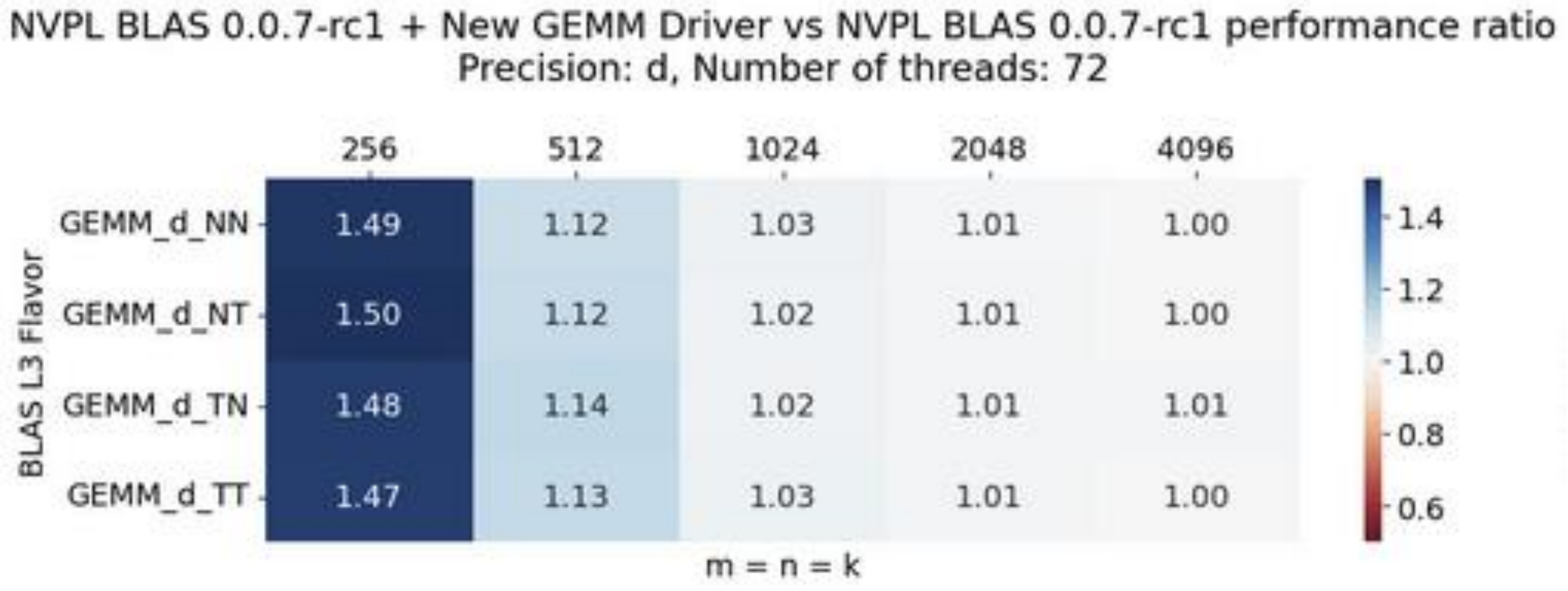


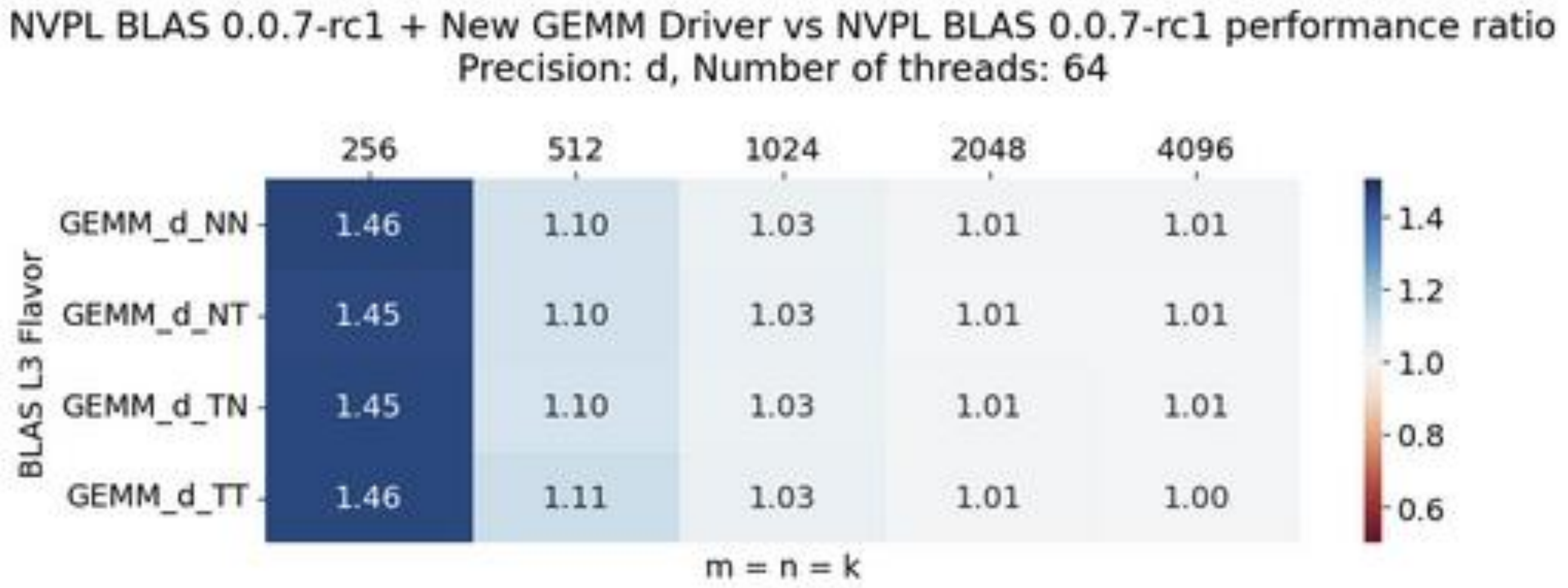
8 threads





72 (64) threads





Conclusion and Next Steps

- NVPL BLAS is based on BLIS with few library architecture extensions to suite binary distribution model
- BLIS is a very powerful and flexible framework with amazing code reuse
- Abstractions and flexibility don't come for free
- Nvidia Grace CPU doesn't require complicated low-level programming

NVPL BLAS next steps

- Extend the functionality
- Continue performance optimizations
  - Small shaped GEMMs
  - Non-GEMM BLAS Level 3
  - Improving thread decomposition
  - Direct complex GEMM implementation
  - BLAS Level 1 and 2

NVIDIA