Enabling SVE evaluation with Arm Instruction Emulator

Daniel Ruiz

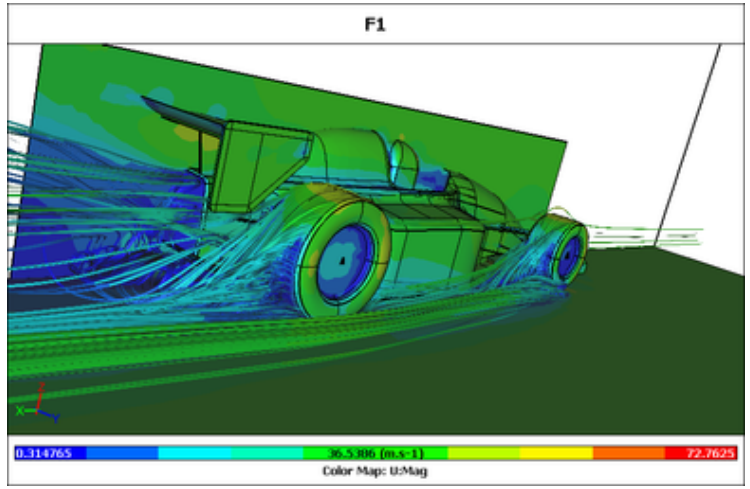
April 18th, 2019

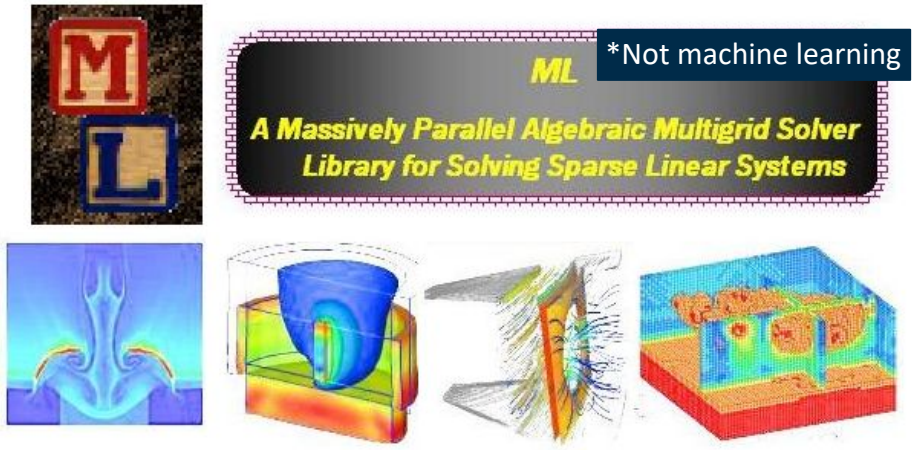
Enabling SVE evaluation with ArmIE

- HPCG introduction
  - Optimizations
- Tools
  - DynamoRIO + ArmIE + scripts
- Analysis
  - Metrics of interest
    - Instruction count
    - Memory metrics
    - Memory system impact
    - Optimization advice

HPCG

- Benchmark for ranking Top500 HPC systems
- HPCG's kernels are representative of real-world scientific applications ran on HPC machines, computationally and data access pattern-wise
  - E.g., Computational fluid dynamics (OpenFOAM), computational photography
- Is not only about the FLOPS!
  - Don't worry, still used as Figure of Merit ☺
  - Along with the memory bandwidth
- Motivates improvements in our Arm Performance Libraries



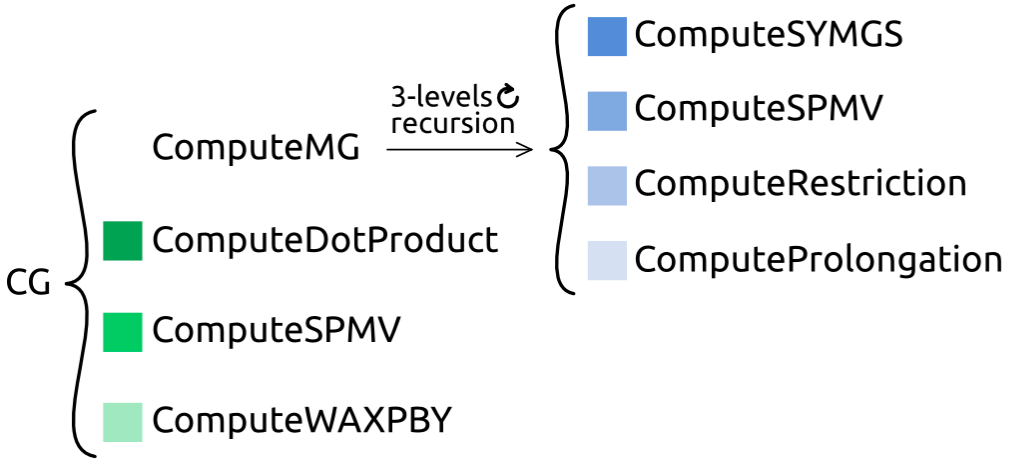


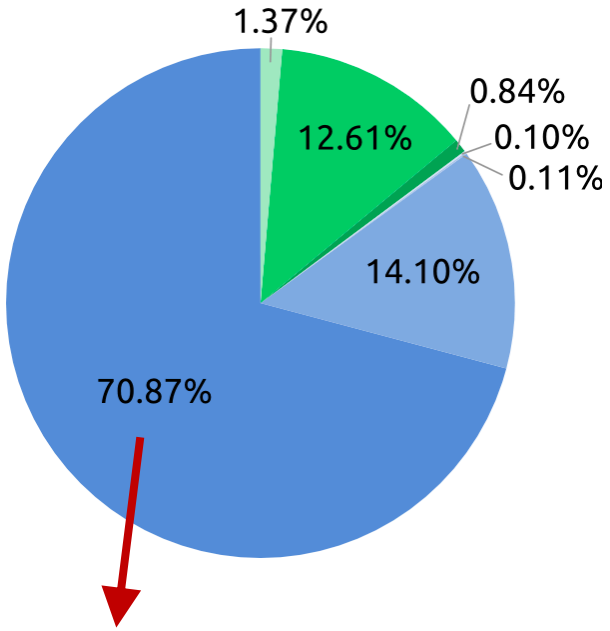
arm

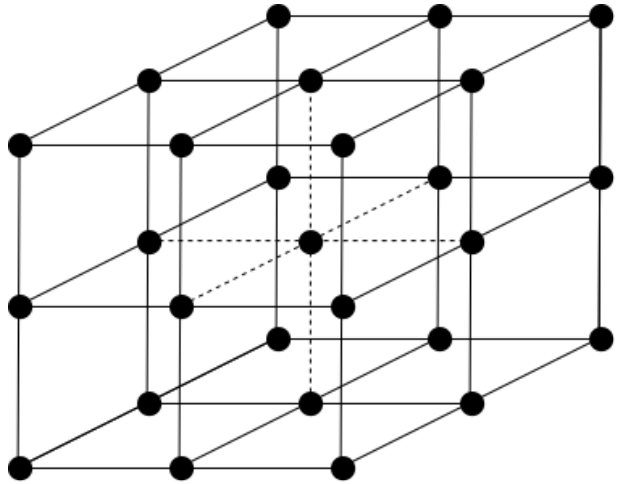
HPCG

- Solves the linear system

$$A * x = r$$







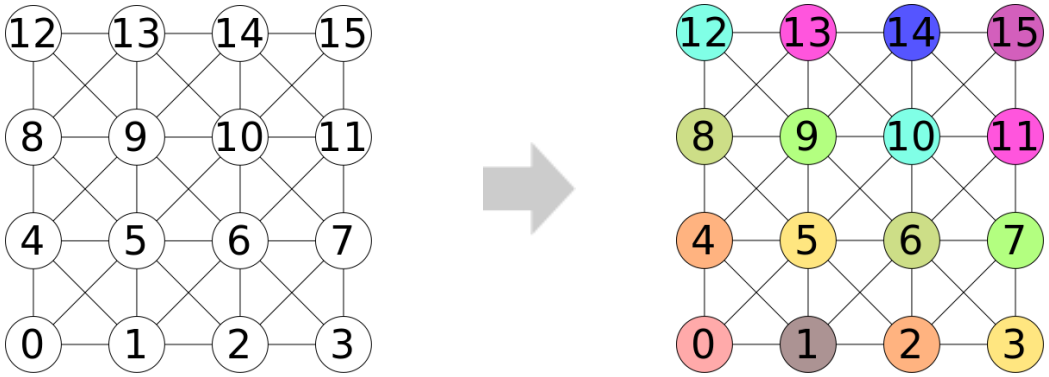
Not parallelizable!

How can we do better?

arm

Optimizations…

Multi-level task dependency graph



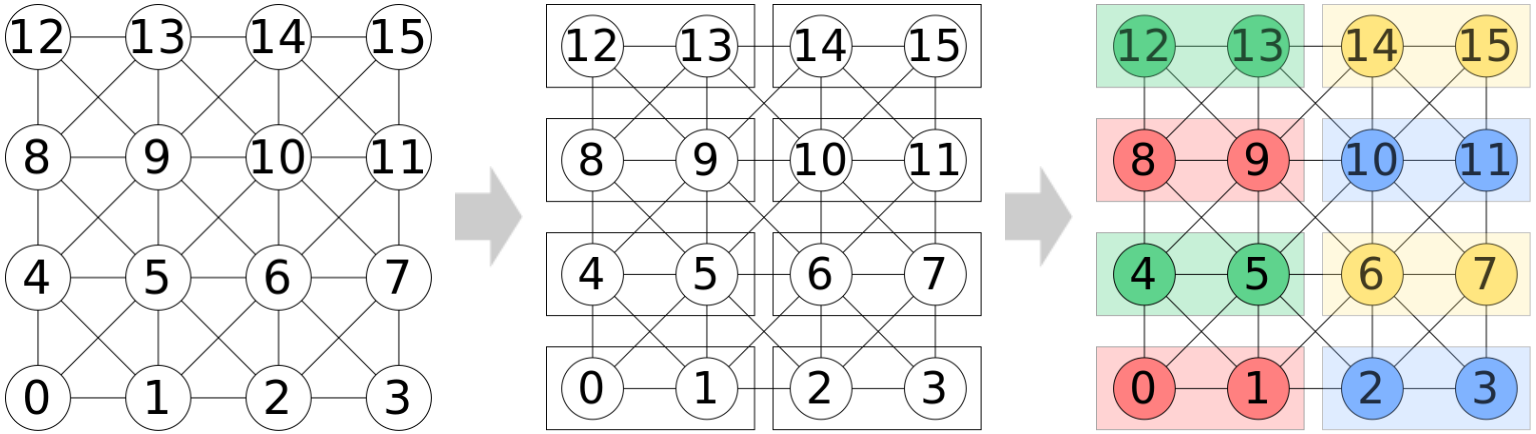

```
Level  0:      0
Level  1:      1
Level  2:   2, 4
Level  3:   3, 5
Level  4:   6, 8
Level  5:   7, 9
Level  7:  10, 12
Level  8:  11, 13
Level  9:     14
Level 10:     15
```

- Nodes in the same level of the graph can be processed in parallel.
- How to:
  1. Add node 0 to the level 1
  2. Mark node 1 as visited
  3. Close level 1
  4. Check neighbors of nodes in previous level to see if dependencies are fulfilled
     1. If yes, add node to the level and mark node as visited
     2. If no, continue with the next node
  5. Close level, add new level and go to 4 if no more nodes to process
  6. Reorder nodes by level

arm

More optimizations…

Block multi-coloring

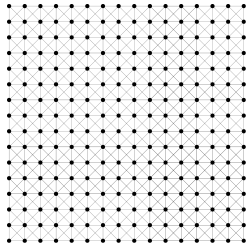


- Blocks with the same color can be processed in parallel.
- How to:
  1. Group N consecutive nodes in blocks
  2. Colorize blocks
  3. Reorder blocks

arm

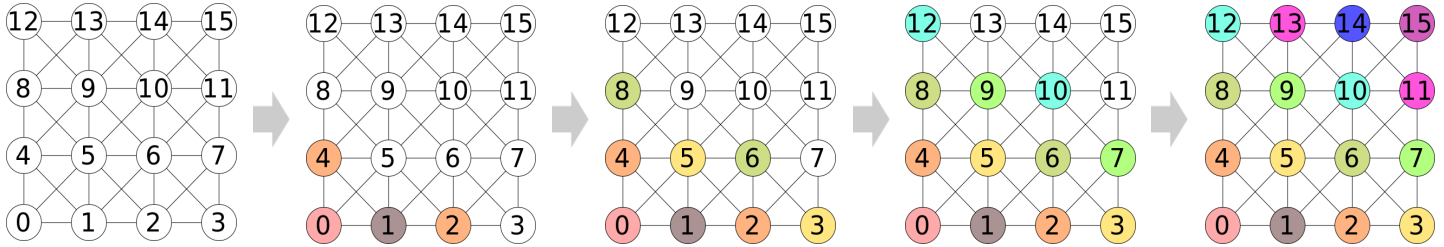
All the (parallelism) optimizations!

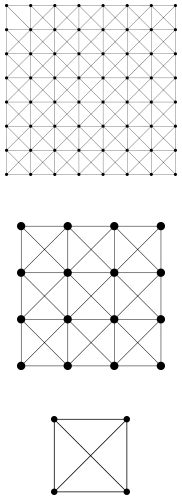
Merging all together



Finest level

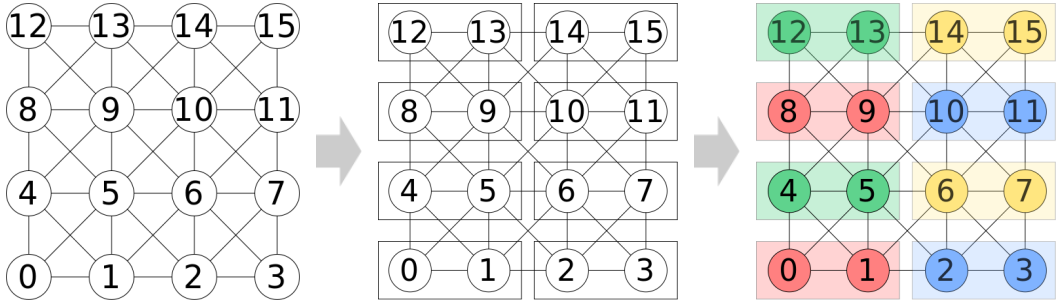
(multi-level task dependency graph)





Coarser levels

(block multi-coloring)

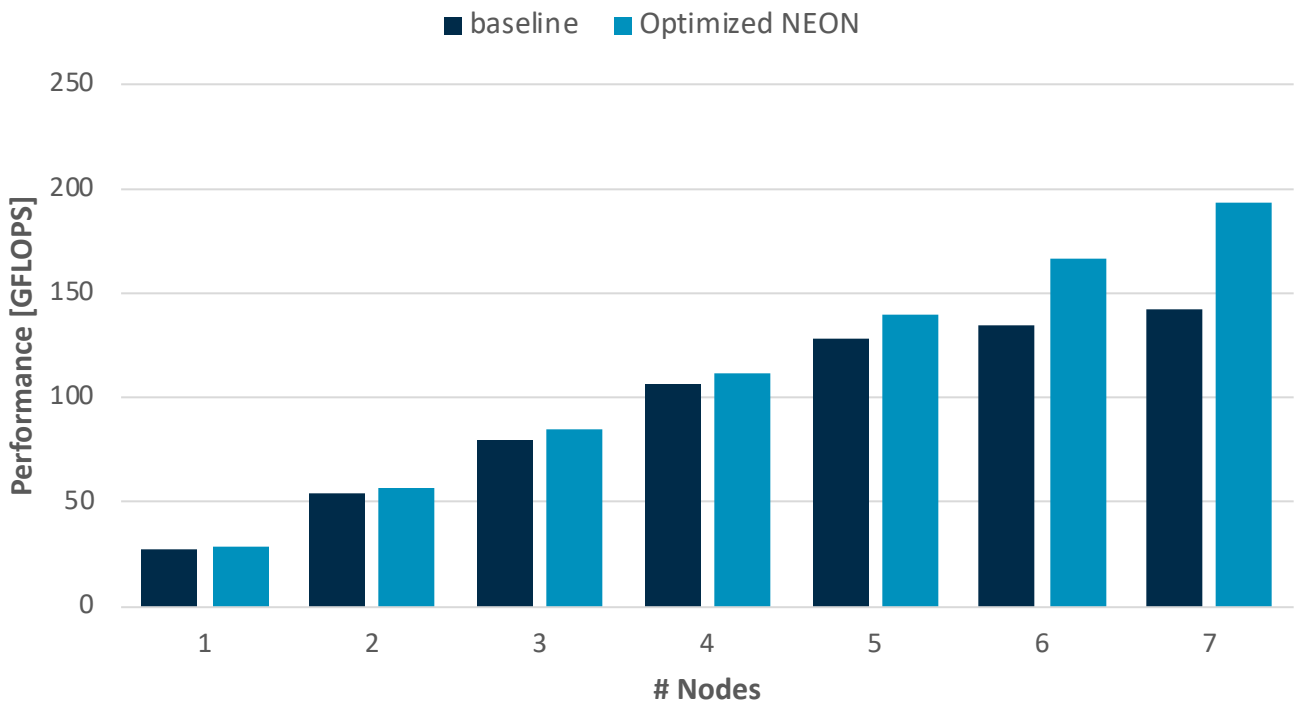


Further information about our code in the Arm blog:
https://community.arm.com/developer/tools-software/hpc/b/hpc-blog/posts/parallelizing-hpcg

arm

Putting the optimizations to the test

- Parallelizing the main kernels improves application scalability.
- Higher gap in performance expected at higher core count.
- Results presented at SC18
  - Positive feedback from the community.



Baseline (MPI-only):
- 56 MPI ranks per node

Optimized NEON:
- 8 MPI ranks per node
- 7 OpenMP threads per MPI rank

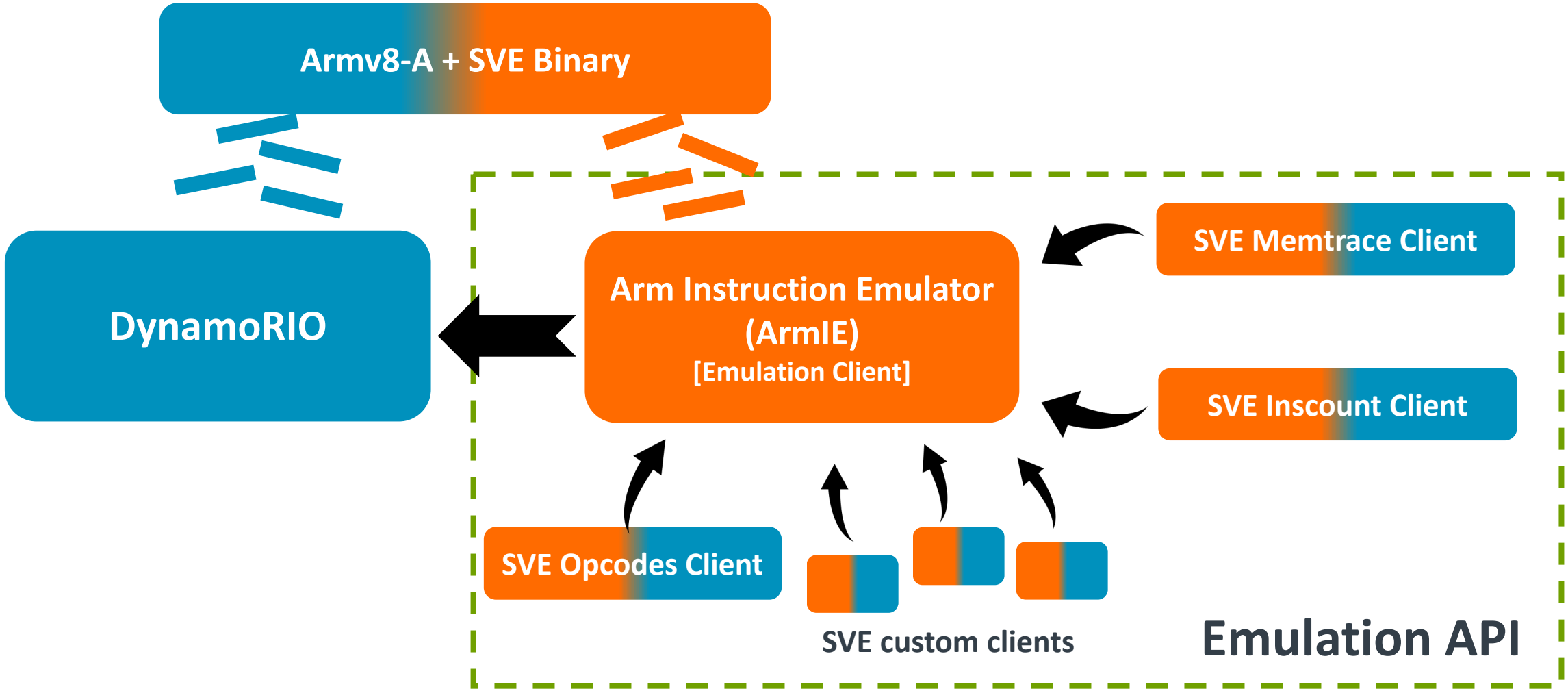
arm

arm

Tools

DynamoRIO & ArmIE

DynamoRIO & ArmIE



arm

Why ArmIE?

Because ArmIE is:

- ✓ Fast functional emulator
  (enables apps with large inputs runs)
- ✓ Easy to use and develop
  (allows custom instrumentation and post-processing)
- ✓ Freely available
- ✓ Partly open-source(API to build your own instrumentation)

ArmIE is not:

- ✗ Cycle accurate
  (no timing information)
- ✗ A simulator
  (Requires Armv8 hardware)
- ✗ Architecture modelling
  (Is all about the apps)

arm

ArmIE under the covers

- Compile application with SVE-capable compiler and run it through ArmIE:
  - `$ armie -msve-vector-bits=512 -i libinscount_emulated.so -- ./sve_app`
- Use **Region-of-Interest (RoI) markers** in the code to delimit instrumentation
  - Not all clients
- Three SVE-ready instrumentation clients come pre-packaged in latest ArmIE version:

SVE Inscount

SVE Opcodes

SVE Memtrace

More info in the Arm blog:
https://community.arm.com/developer/tools-software/hpc/b/hpc-blog/posts/emulating-sve-on-armv8-using-dynamorio-and-armie

arm

arm

Metrics of interest

What we ran

2 HPCG versions:

- **Baseline**
  - What everyone gets to try out
- **Optimized**
  - Multi-Level Task dependency and Block coloring optimizations
  - Other minor performance improvements (loop fusion/unroll, memory allocations, etc.)
- All versions are compiled with the Arm HPC Compiler 19.0.

Instruction client

- Counts dynamically executed instructions
  - Total instructions
  - SVE instructions

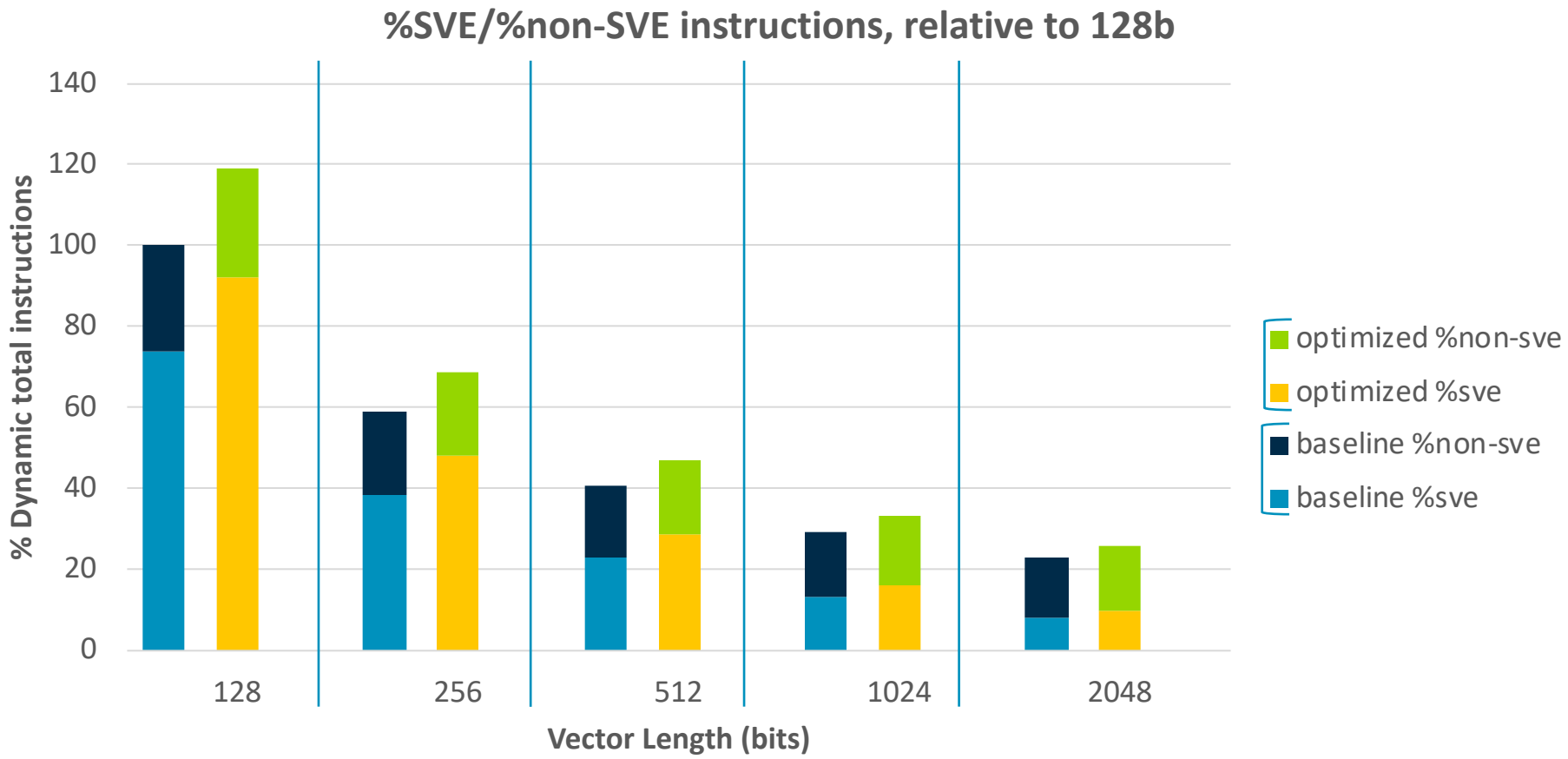
- The metric can be used to know how well the compiler was able to vectorize
  - Also to compute the reduction in terms of instructions

- Full inscount of SVE application (512-bit vectors):

```
$ armie -msve-vector-bits=512 -i libinscount_emulated.so -- ./sve_example
83971 instructions executed of which 22 were SVE instructions
```

Counting instructions

- Instruction count client



arm

Compiler auto-vectorization analysis

Compiler		# Vectorized loops	
		baseline	optimized
SVE	GCC 8.2.0	8/8	12/17
	Arm HPC Compiler 19.0	8/8	12/17
AVX2	Intel Compiler 19.0.3	8/8	17/17

- GCC and Arm HPC Compiler:
  - Reverse loops are not vectorized.
- Intel Compiler:
  - Considers outer loops, ends up vectorizing inner loops.

```
for ( int j = 0; j < nnzInChunk; j++ ) {
  sum0 -= mtxVal[i+0][j] * xv[ mtxInd[i+0][j] ];
  sum1 -= mtxVal[i+1][j] * xv[ mtxInd[i+1][j] ];
  sum2 -= mtxVal[i+2][j] * xv[ mtxInd[i+2][j] ];
  sum3 -= mtxVal[i+3][j] * xv[ mtxInd[i+3][j] ];
}
```

```
for ( int j = nnzInChunk-1; j >= 0; j-- ) {
  sum3 -= mtxVal[i-3][j] * xv[ mtxIndL[i-3][j] ];
  sum2 -= mtxVal[i-2][j] * xv[ mtxIndL[i-2][j] ];
  sum1 -= mtxVal[i-1][j] * xv[ mtxIndL[i-1][j] ];
  sum0 -= mtxVal[i-0][j] * xv[ mtxIndL[i-0][j] ];
}
```

arm

Memory trace client

- Based on the existing DR *memtrace_simple* client

- AArch64 tracing done by DR
- SVE tracing is done separately by ArmIE

  Trace inside a Region-of-interest (RoI) through markers in the code

```
#define __START_TRACE() { asm volatile (".inst 0x2520e020"); }
#define __STOP_TRACE()  { asm volatile (".inst 0x2520e040"); }
```

- Two trace files are generated (AArch64 and SVE trace files)
  - A shared synchronised counter is used to number ArmIE and DR traces in order to record the correct temporal sequence between the two traces
  - Merging both traces (post-process) results in the complete trace inside the RoI

SVE Memory Tracing Client

- Memtrace format:

sequence number	Thread ID	SVE Bundle	isWrite	Data size	Data address	PC
Shared counter that ensures correct trace order		Identifies a contiguous access or a gather/scatter bundle (and the element position)	Write/read operation			

- ArmIE command example (512-bit vector):

```
$ armie -e libmemtrace_sve_512.so -i libmemtrace_emulated.so -- ./sve_example
```

  - AArch64 output => memtrace.sve_example.26213.0000.log
  - SVE output => sve-memtrace.sve_example.26213.0000.log

Merging Traces

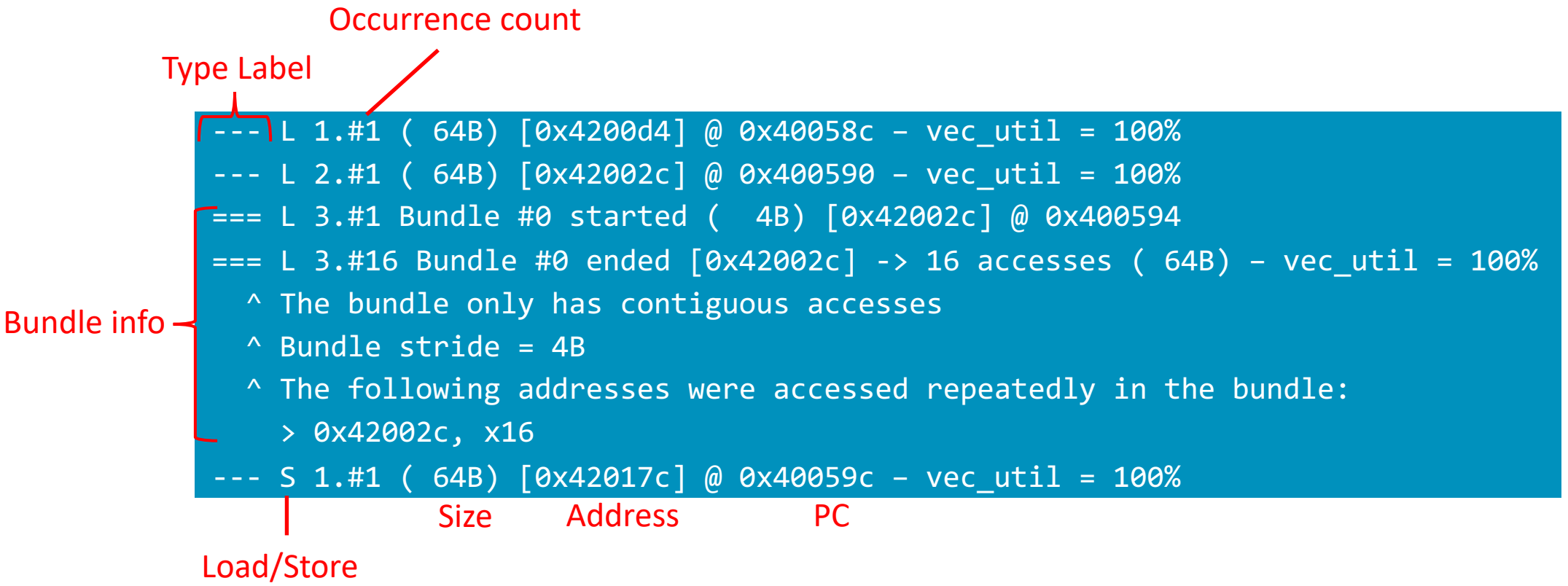
- AArch64 and SVE memory traces facilitate merging and analysis
  - Counter parameter for merging
  - RoI markers are included in the SVE trace to prune unwanted traces (same format as the trace)
    - ThreadID field is **-1** for the RoI start
      ```
      19, -1, 0, 1, 0, (nil), (nil)
      ```
    - ThreadID field is **-2** for the RoI stop
      ```
      71, -2, 0, 1, 0, (nil), (nil)
      ```

- A different separator after the counter identifies the origin of the trace
  - Colon (:) symbol for AArch64 traces
    ```
    14: 0, 0, 0,  8, 0x41fde8, 0x4005fc
    ```
  - Comma (,) symbol for SVE traces
    ```
    20, 0, 0, 0, 64, 0x4200d4, 0x40058c
    ```


arm

Parsing Traces

- The merged traces can be parsed to better understand the memory accesses
  - Higher verbosity / better readability or by extracting metrics

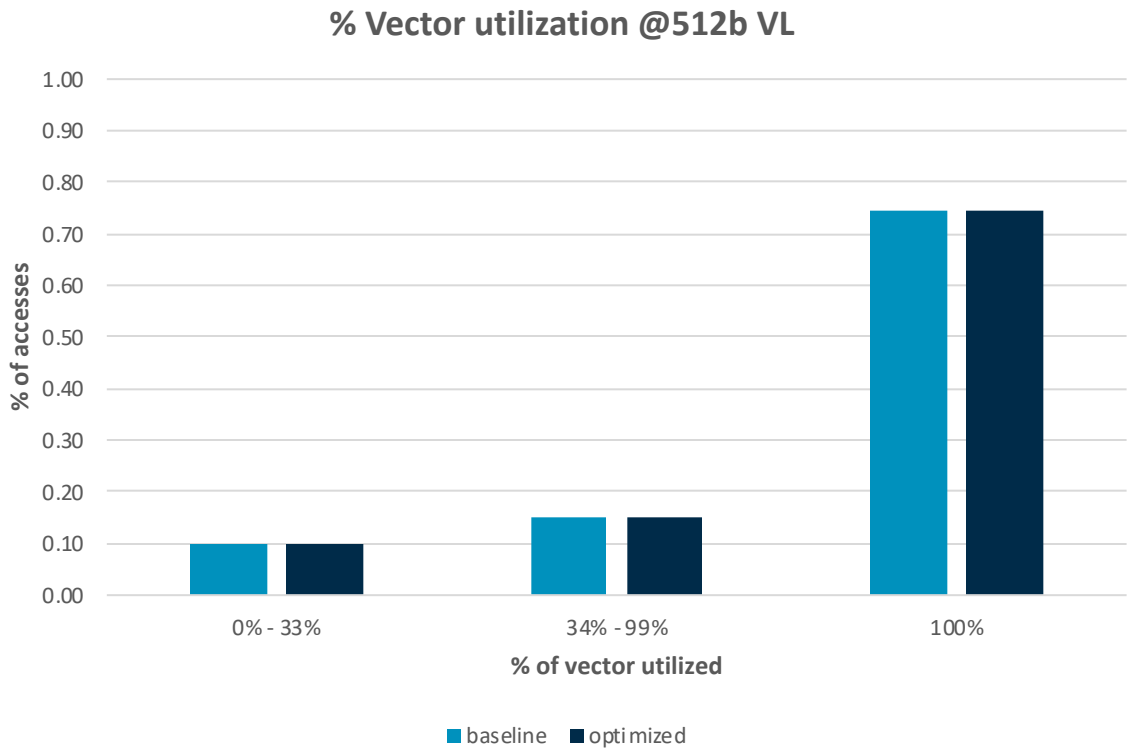


arm

Studying SVE memory accesses

- Vector utilization [extracted from memory traces]

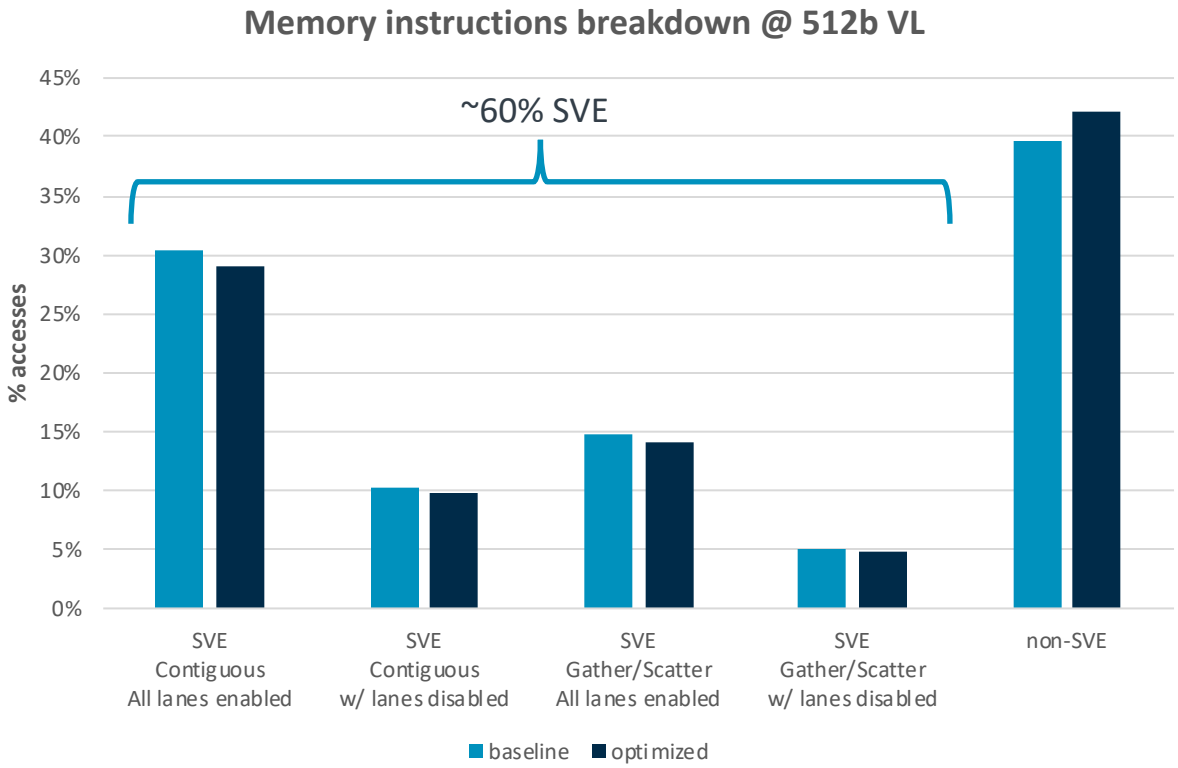
Version	Avg. Vector utilization
baseline	82.35%
optimized	82.39%



arm

Looking at the memory accesses

- Memory accesses present similar characteristics
- Further work could be done to decrease the usage of gathers/scatters



arm

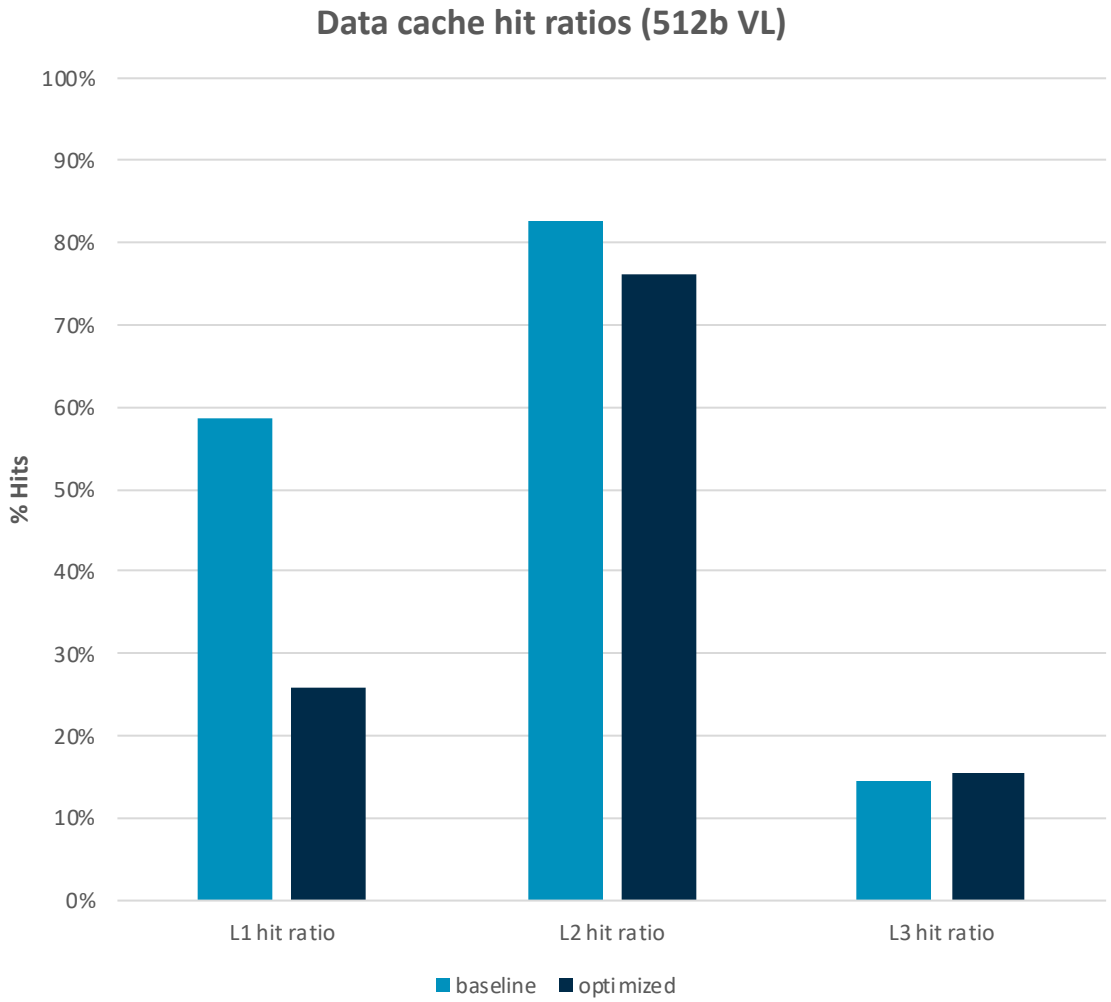
Insights

- Poor vector utilization
  - Short loops?
  - Conditional statements?
- Too many gather/scatters
  - Contiguous loads/stores are preferred
  - Can data be reorganized?
- Low percentage of SVE memory accesses
  - How good was vectorization?
- Compare against non-SVE executions
  - i.e., NEON

SVE Cache Simulator

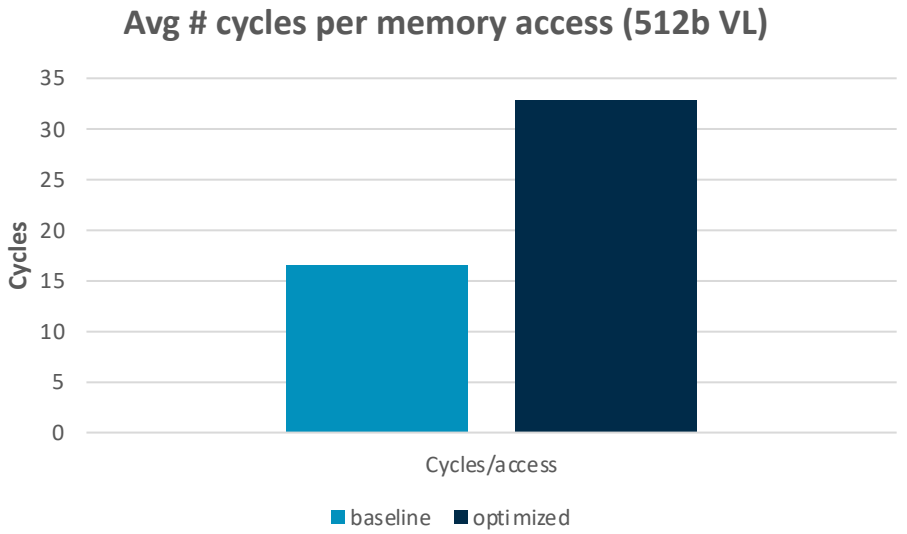
- A simple modular cache simulator for SVE memory traces
- Supports a single-core multi-level cache system
  - ArmIE SVE tracing has compatibility issues with multithreaded applications
- Supports prefetcher plugins
  - A simple stride prefetcher is currently available

Looking at cache



Cache simulator parameters

	L1	L2	L3
Cache Size (KB)	64	1024	2048
Line Size (#words)	16	16	16
Word Size (Bytes)	4	4	4
Set Size (n-ways)	8	8	32
Latency (cycles)	4	11	60
Memory Latency (cycles)			156
Stride prefetcher to L1			

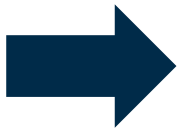


arm

ArmIE methodology

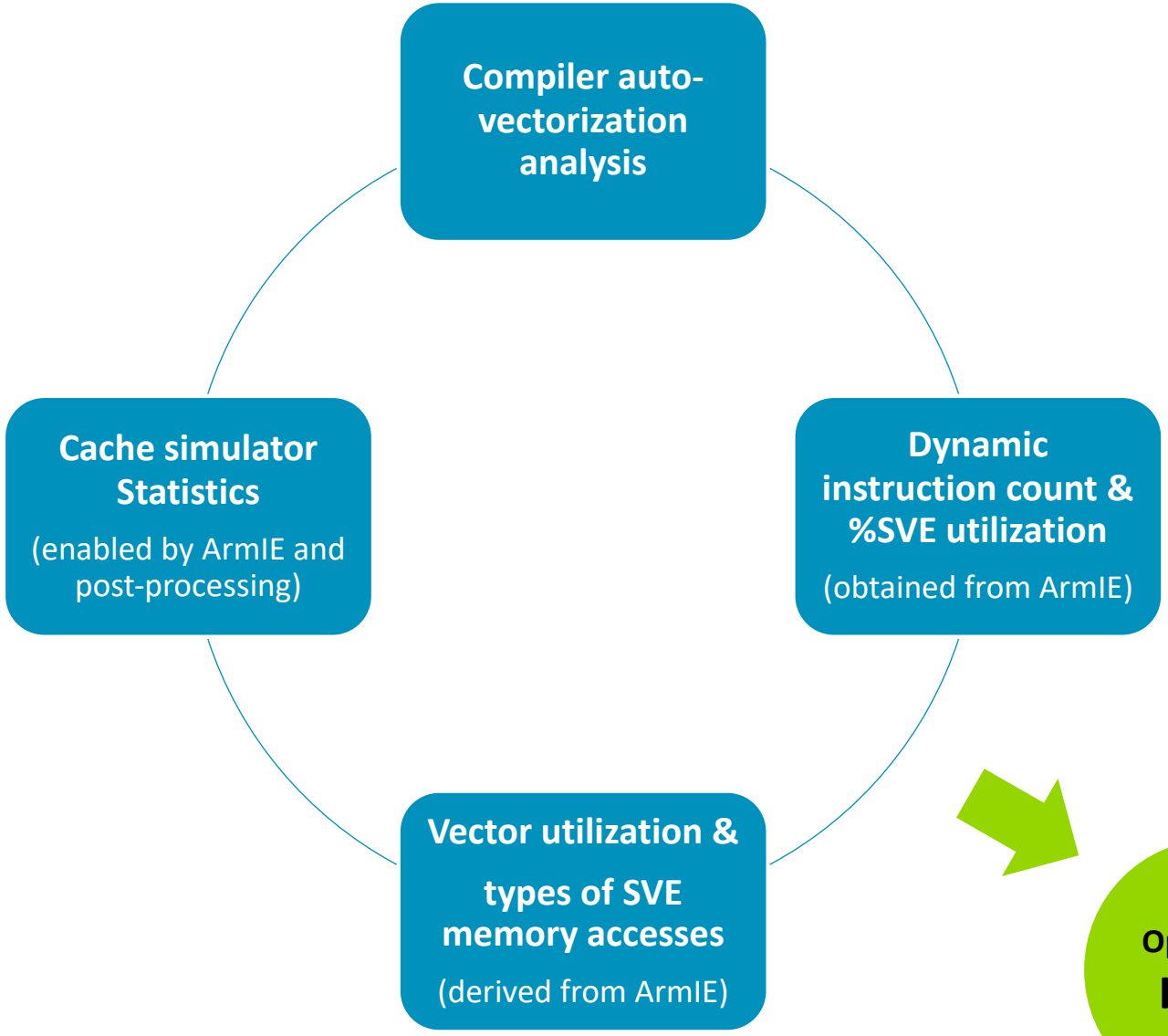
Overview

HPCG



The metrics you can get:

- Dynamic instruction count
- %SVE utilization
- %SVE contiguous memory accesses or gather/scatter
- L1$ hit rate
- Variations avg. #cycles per memory access



Optimized HPCG

arm

ArmIE Roadmap

- ArmIE 19.0 is to be released in the next few days
  - Mostly bug fixes
- Improved clients and a new instruction trace client planned for 19.1
  - With improvements to region-of-interest and client options
- Planning for future versions:
  - Better multi-thread support
  - Emulation API updates
  - Debug functionality

Final remarks

- ArmIE enables SVE evaluation with more realistic input sizes
  - Emulation overhead depends on the number of SVE instructions
  - Non-SVE instructions have near-zero overhead
  - SVE Application test and validation is now possible without simulators.

- Evaluation depends on time-agnostic metrics
  - Instruction counts, memory traces, etc.
  - Need to look at all the metrics, they have little value by their own
  - Possibility to create custom clients using the emulation API
  - Post-processing can extract fine-grain metrics

- ArmIE does not replace simulators
  - All tools have its purpose

arm

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