Best Practice Guide
Modern Processors

Ole Widar Saastad, University of Oslo, Norway

Kristina Kapanova, NCSA, Bulgaria

Stoyan Markov, NCSA, Bulgaria

Cristian Morales, BSC, Spain

Anastasiia Shamakina, HLRS, Germany

Nick Johnson, EPCC, United Kingdom

Ezhilmathi Krishnasamy, University of Luxembourg, Luxembourg

Sebastien Varrette, University of Luxembourg, Luxembourg

Hayk Shoukourian (Editor), LRZ, Germany

Updated 5-5-2021

Table of Contents

1. Introduction

This Best Practice Guide (BPG) extends the previously developed series of BPGs [1] (these older guides are still relevant as they provide valuable background information) by providing an update on new technologies and systems for the further support of European High Performance Computing (HPC) user community in achieving a remarkable performance of their large-scale applications. It covers existing systems and aims to provide support for scientists to port, build and run their applications on these systems. While some benchmarking is part of this guide, the results provided are mainly an illustration of the different systems characteristics, and should not be used as guides for the comparison of systems presented nor should be used for system procurement considerations. Procurement [2] and benchmarking [3] are well covered by other PRACE work packages and are out of this BPG's discussion scope.

This BPG document has grown to be a hybrid of field guide and a textbook approach. The system and processor coverage provide some relevant technical information for the users who need a deeper knowledge of the system in order to fully utilise the hardware. While the field guide approach provides hints and starting points for porting and building scientific software. For this, a range of compilers, libraries, debuggers, performance analysis tools, etc. are covered. While recommendation for compilers, libraries and flags are covered we acknowledge that there is no magic bullet as all codes are different. Unfortunately there is often no way around the trial and error approach.

Some in-depth documentation of the covered processors is provided. This includes some background on the inner workings of the processors considered; the number of threads each core can handle; how these threads are implemented and how these threads (instruction streams) are scheduled onto different execution units within the core. In addition, this guide describes how the vector units with different lengths (256, 512 or in the case of SVE - variable and generally unknown until execution time) are implemented. As most of HPC work up to now has been done in 64 bit floating point the emphasis is on this data type, specially for vectors. In addition to the processor executing units, memory in its many levels of hierarchy is important. The different implementations of Non-Uniform Memory Access (NUMA) are also covered in this BPG.

The guide gives a description of the hardware for a selection of relevant processors currently deployed in some PRACE HPC systems. It includes ARM64 (Huawei/HiSilicon and Marvell)[1] and x86-64 (AMD and Intel). It provides information on the programming models and development environment as well as information about porting programs. Furthermore it provides sections about strategies on how to analyze and improve the performance of applications. While this guide does not provide an update on all recent processors, some of the previous BPG releases [1] do cover other processor architectures not discussed in this guide (e.g. Power architecture) and should be considered as a staring point for work.

This guide aims also to increase the user awareness on energy and power consumption of individual applications by providing some analysis on usefulness of maximum CPU frequency scaling based on the type of application considered (e.g. CPU-bound, memory-bound, etc.).

As mentioned earlier, this guide covers processors and technologies deployed in European systems (with a small exception to ARM SVE - soon to be deployed in EU). While European ARM and RISC-V are just over the horizon, systems using these processors are not deployed for production at the time of this writing (Q3-2020). As this European technology is still a couple of years from deployment it's not covered in the current guides, however, this new technology will require substantial documentation. Different types of accelerators are covered by other BPGs [1] and a corresponding update for these is currently being developed.

Emphasis has been given to providing relevant tips and hints via examples for scientists not deeply involved into the art of HPC programming. This document aims to provide a set of best practices that will make adaptation to these modern processors easier. It is not intended to replace an in depth textbook, nor replacing the documentation for the different tools described. The hope is that it should be the first document to reach for when starting to build a scientific software.

As for programming languages used in examples these are either C or Fortran. C is nice as it maps nicely to machine instructions for the simple examples and together with Fortran makes up the major languages used in

[1] While the vector enabled Fujitsu A64FX with SVE processor is not covered there is a short section about Scalable Vector Extension (SVE) included.

HPC. While knowledge of assembly programming is not as widespread as before the examples are on a level of simplicity that the should be easily accessible.

This current guide is a joined guide from several different guides in the past. The previous guides where each processor had it's own guide is a thing of the past. This guide covers all relevant processors (see above), but each processor chapter is still a separate chapter. The merger is not yet fully completed.

Furthermore, this guide will provide information on the following, recently deployed, European flagship super-computing systems:

- Fulhame @ EPCC, UK
- MareNostrum @ BSC, Spain
- SuperMUC-NG @ LRZ, Germany
- Hawk @ HLRS, Germany
- Betzy @ SIGMA2, Norway

2. ARM Processors

The ARM processor architecture is the most shipped processor class and is used in almost every aspect of computation, phones, pads, cars, embedded etc. The only segment where it currently is not dominant is the data center segment.

ARM processors in HPC have shown a growing interest over the last years. There are different reasons for this ranging from the obvious like reduced processor cost to more interesting ones like more flexible development (e.g. SVE) and the larger memory bandwidth due to requirements mainly stemming from HPC user community. The decision by large compute centers like RIKEN [60] and the European Processor Initiative, EPI [61] is to develop tailored ARM processors [62], [64] and for their need also add to the acceptance of ARM as an architecture for HPC.

Large companies like Atos, Cray, Fujitsu and HPE offer systems based on ARM architecture, thus naturally contributing to the acceptance of ARM as a valuable architecture for HPC.

ARM64 processor architectures have been documented in the previous version of the Best Practice Guides [4]. In this updated version more emphasis is put on the newer features and the current processors available on the market. The Kunpeng 920 is discussed in this version along with more updated information about Thunder X2. In addition, vector instructions (SVE) are briefly discussed and at time of this writing the access to SVE enabled processors are very limited.

2.1. Architecture

There are multiple vendors implementing the ARM64 architecture for HPC workloads. The most common models available on the market are tested and covered in this guide. The Amazon Graviton processors (Graviton and Graviton2) [52] and Ampere Altra [53] are more general purpose and machine learning processors and not really targeted for HPC and hence not evaluated.

At the time of this writing the only ARM processor supporting Scalable Vector Extension (SVE) is the Fujitsu A64FX. This is currently not generally available and could not be tested, for more information see [65]. However, SVE support with more vendors will be available in the 2020/2021 time frame. See [63] for a review.

Models evaluated

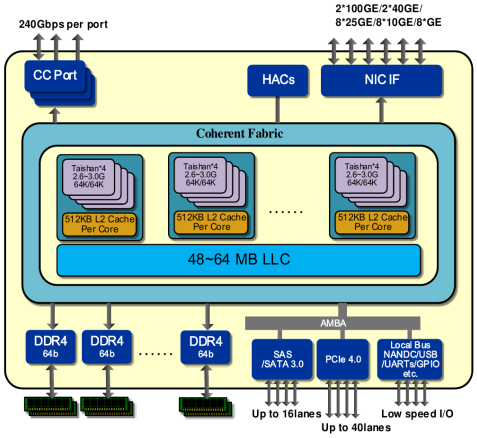
- Huawei - Kunpeng 920 (HiSilicon 1620), Huawei server [57].
- Marvell - ThunderX2, Gigabyte server [59].

Both of these implementations are ARM v8 compliant, while the Kunpeng 920 processor support the ARM v8.2 the ThunderX2 support ARM v8.1, see [66] and [74].

2.1.1. Kunpeng 920

The Kunpeng 920 (HiSilicon 1620) processor is the successor to the HiSilicon 1616 processor. The HiSilicon 1620 comes with more and faster cores, with an updated instruction set architecture, full NEON (128-bit vector unit), 16 bits floating point, and a range of other improvements. Eight DDR4 memory controllers provide a high memory bandwidth (see later for measurements).

Hi1620 Specifications Overview



CPU core	Up to 64 ARMv 8.2 cores, 3.0 GHz, 48-bit physical address 4 issue OoO superscalar design 64 KB L1 I Cache and 64 KB L1 D cache
L2 cache	512 KB private per core, 24 MB total
L3 cache	48 MB shared for all (1 MB/core), Partitioned
Memory	8-channel DDR4-2400/2666/2933/3200 16 ranks/channel, 1DPC and 2DPC configurations x4/x8 support ECC, SDDC, DDDC
PCIe	40 lanes of PCIe Gen4.0 16x
Integrated I/O	8 lanes of ETH, Combo MACs, supporting 2 x 100GE, 2 x 40GE, 8 x 25GE/10GE,10 x GE, supporting SR-IOV RoCEv2/RoCEv1 x4 USB 3.0 x8 SAS 3.0 x2 SATA 3.0
Crypto engine	AES, DES/3DES, MD5, SHA1, SHA2, HMAC, CMAC Up to 100 Gbit/s
Compression	GZIP, LZS, LZ4 Up to 40 Gbit/s (compress)/100 Gbit/s (decompression)
RAID	RAID5/6, DIF, XOR, PQ acceleration
CCIX	Cache coherency interface for accelerator, like Xilinx FPGA World's 1st CCIX solution
Scale-up	Coherent SMP interface for 2P/4P 3*240Gbps bandwidth
Power	TDP ~150 W (48C 2.6 GHz)

Figure 1. Kunpeng 920 block diagram [54]

Huawei reports the core supports almost all the ARMv8.4-A ISA features with a few exceptions, but including dot product and the FP16 FML extension, see [67].

2.1.2. ThunderX2

The ThunderX2 is an evolution in the ThunderX family of processors from Cavium, now part of Marvell. The ThunderX2 provide full NEON 128 bits vector support, Simultaneous Multi Thread (SMT) support [75] and up to eight DDR4 memory controllers for high memory bandwidth (see later for measurements).

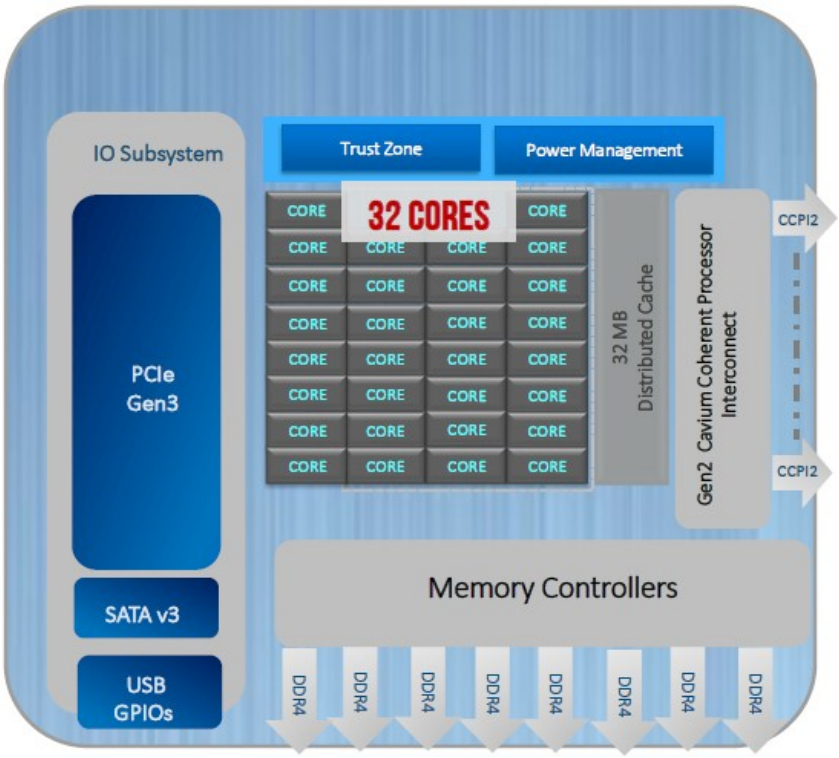


Figure 2. ThunderX2 block diagram [55]

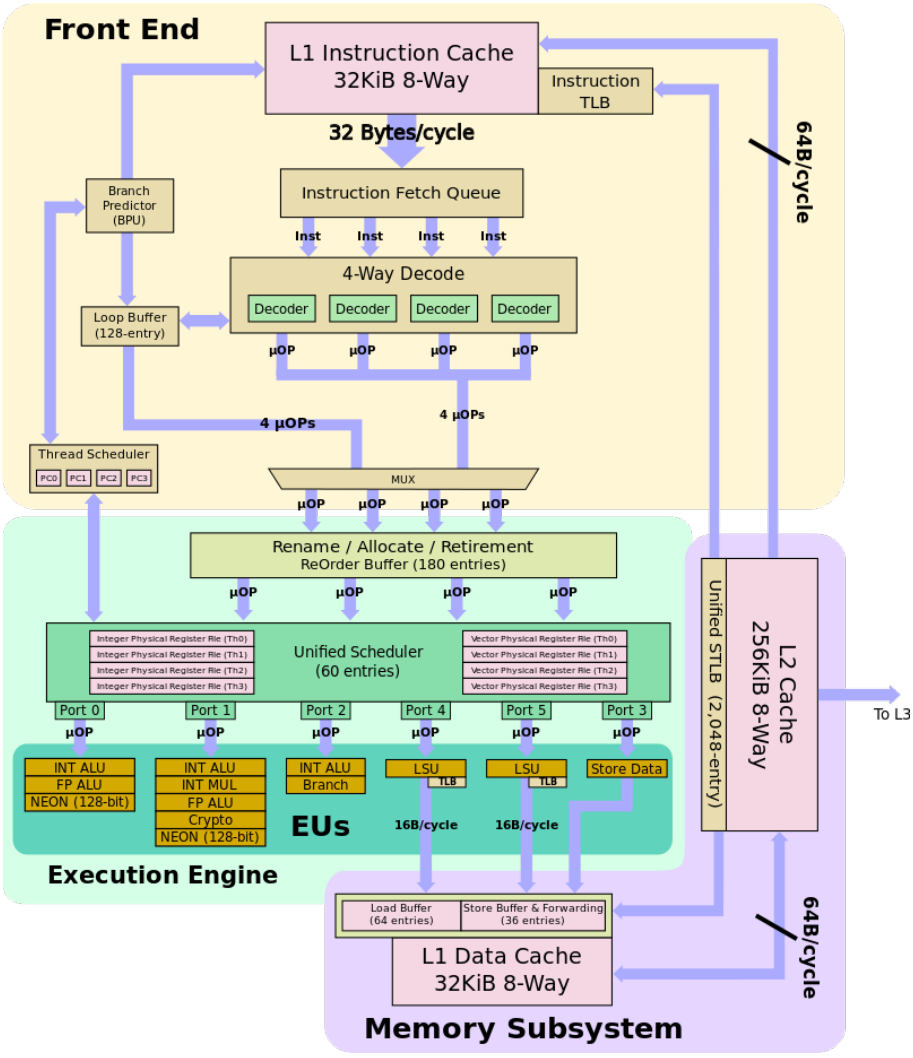


Figure 3. ThunderX2 block diagram [56]

From the core diagram above we can establish that there are two 128 bits NEON SIMD execution units and two scalar floating point execution units in addition to integer and load/store units.

2.1.2.1. SMT

Simultaneous Multi Thread [75] is a technique that provide the possibility to run multiple threads (instructions streams) on a single core, also known as Thread Level Parallelism (TLP). The core provide several parallel paths for threads and also a set of context for each thread. The implementation is somewhat more complex, see Figure 3, “ThunderX2 block diagram [56]”.

The practical manifestation is that it looks like there are twice (SMT-2) or four SMT-4) times as many cores in the processor. This can be beneficial for some application where less context switching can improve performance. In HPC the benefit is not always observed as most application are limited by memory bandwidth or the fact that the different threads share the same executional units. It may look like there are four times as many cores, but in reality there are no more compute resources. This is documented in Section 2.6, “Simultaneous Multi Threading (SMT) performance impact”.

2.1.2.2. New release - Thunder X3

With the release of Thunder X3 Marvell takes a new generation of their ARM processors to the market. Press coverage [68] tell about a major upgrade in performance. With up to 96 cores with SMT-4 yielding 384 logical cores, four 128 NEON SIMD units (no SVE yet). With expected 3x performance over Thunder X2. This new

processor might prove formidable upgrade. Little is known at the time of writing, but the processor should hit the market during second half of 2020.

2.1.3. NUMA architecture

All the evaluated architectures presented in this guide are Non Uniform Memory Architectures (NUMA). This is quite common, but nevertheless requires programmers and users to pay attention to thread and rank placement, particularly when dealing with SMT enabled systems, as process and thread placement and binding to cores and NUMA banks can significantly impact performance. See for [7] an overview.

The ThunderX2 can be booted with SMT off (one thread per core), SMT-2 (two threads per core) or SMT-4 (four threads per core). Optimal settings vary, there is no setting that fits all. One general rule is that more threads require more attention to NUMA placement. The performance may vary as more threads can help hiding memory latencies. On the other hand applications that are sensitive to memory bandwidth might show less benefit from more threads.

The recommendation for general usage would be to turn SMT off. The extra threads seems to have little benefit for a range of HPC codes. In selected cases the SMT-2 or SMT-4 can be tried. Please review the benchmark section for more information on this. As the benchmarks are not universal it's often a trial and error process for the relevant applications.

2.2. Programming Environment

A good set of software ecosystem for development now exist for the ARM processors. Compilers, performance libraries and debuggers are all readily available.

2.2.1. Compilers

Compilers evaluated

- GNU 9.1 and GNU 9.2
- ARM HPC compiler 19.2 and 19.3

The compilers come with a large range of compiler flags which have a varying degree of impact on performance of the generated code. It's almost impossible to get the optimal set of compiler flags for a given code. Experience has shown that a certain set of compiler flags will provide code with a good performance. Suggested flags are given below.

Table 1. Suggested compiler flags for Kunpeng 920

Compiler	Suggested flags
GNU	-O3 -march=armv8.2-a+simd -mcpu=native -fomit-frame-pointer
ARM HPC compiler	-Ofast -march=armv8.2-a+simd -mcpu=tsv110 -fomit-frame-pointer

The +simd is default for all combinations of -march and -mcpu. While the usage of the keyword "simd" is more intuitive for HPC users the gcc manual also uses the keyword vector. The v8.2-a includes support for dot product [69], which can be enabled by the +dotprod flag which also enables simd so only the flag armv8.2-a+dotprod is needed. As the dot product instruction does not support floating point it is of minor importance for most HPC applications.

As Huawei reports, almost all of the ARM v8.4-a instructions are implemented the flag -march=armv8.4-a+fp16 can be tried, +fp16 enable simd and dot product. Please review the gcc/g++/gfortran man pages or ARM HPC compiler documentation.

The usage of armv8.3-a and armv8.4-a require a fairly new kernel and assembler. Check the glibc version using

```
ldd --version
```

RHEL 8 comes with 2.28 and Ubuntu 18.04 comes with 2.27 which both work fine.

Table 2. Suggested compiler flags for Thunder X2

Compiler	Suggested flags
GNU	-O3 -march=armv8-a+simd -mcpu=thunderx2t99 -fomit-frame-pointer
ARM HPC compiler	-Ofast -march=armv8-a+simd -mcpu=thunderx2t99 -omit-frame-pointer

2.2.1.1. Effect of using SIMD (NEON) vector instructions

The NEON SIMD 128 bit vector execution unit can improve performance significantly. Even short vector units at 128 bit size can handle two 64 bit floating point numbers at the time, in principle double the floating point performance.

The NEON unit is IEEE-754 compliant with a few minor exceptions, (mostly related to rounding and comparing) [70]. NEON should therefore be as safe regarding numerical precision to use the NEON instructions as the scalar floating point instructions. Hence vector instructions can be applied almost universally.

Testing has shown that the performance gain is significant. Tests compiling reference implementation for matrix matrix multiplication, NASA Parallel Benchmark (NPB) [122] BT and a real life scientific stellar atmosphere code (Bifrost, using its own internal performance units)[130] demonstrate the effect :

Table 3. Effect of NEON SIMD

Benchmark	Flags	Performance
Matrix matrix multiplication	-Ofast -march=armv8.2-a+nosimd	1.88 GFLOPS
	-Ofast -march=armv8.2-a+simd	3.18 GFLOPS
NPB BT	-Ofast -march=armv8.2-a+nosimd -mcpu=tsv110	142929 Mop/s total
	-Ofast -march=armv8.2-a+simd -mcpu=tsv110	158920 Mop/s total
Stellar atmosphere (Bifrost)	-Ofast -march=armv8-a+nosimd -mcpu=thunderx2t99	13.82 Mz/s
	-Ofast -march=armv8-a+simd -mcpu=thunderx2t99	16.10 Mz/s

Speedups ranging from 1.69 to 1.11 for benchmarks are recorded while a speedup of 1.17 was demonstrated for real scientific code. These tests demonstrate the importance of a vector unit, even with a narrow 128 bit unit like the NEON.

With the significant performance gain experienced with NEON there is a general expectation that SVE will provide a substantial floating point performance boost. The experience from the x86-64 architecture leave room for optimism. However, without real benchmarks this is still uncharted territory.

2.2.1.2. Usage of optimisation flags

The optimisation flags O3 and Ofast invoke different optimisations for the GNU and the ARM HPC compiler. The code generation differs between O3 and Ofast. The Ofast can invoke unsafe optimisations that might not yield the same numerical result as O2 or even O3. Please consult the compiler documentation about high level of optimisation. Typical for the Ofast flag is what the GNU manual pages states : "Disregard strict standards compliance." This might be OK or cause problems for some codes. The simple paranoia.c [95] program might be enough to demonstrate the possible problems.

2.2.2. Vendor performance libraries

The ARM HPC Compiler comes with a performance library built for the relevant ARM implementations. It contail the usual routines, see [136] for details.

The syntax when using the ARM HPC compiler is shown in the table below.

Table 4. Compiler flags for ARM performance library using ARM compiler

Library type	Flag
Serial	-armpl

Library type	Flag
Serial. Infer the processor from the system	-armpl=native
Parallel / multithreaded	-armpl=parallel

Just adding one of these to the compiling command line will invoke linking of the appropriate performance library.

Scalar math functions library, commonly known as libm, when distributed by the operating system is also available in an optimised version, called libamath.

Before the Arm Compiler links to the libm library, the compiler automatically links to the libamath library to provide enhanced performance by using the optimised functions. This is a default behaviour and does not require that user supplies any specific compiler flags to initiate this behaviour.

The performance of a well known benchmark called Savage [129] which exercise trigonometric functions experiences about 15% speedup when using the ARM library libamath compared to the standard libm.

For detailed information please consult the ARM HPC compiler documentation [136].

2.2.3. Scalable Vector Extension (SVE) software support

The SVE [73] will represent a major leap in floating point performance, see [71]. The 256 and 512 bit vector units with other architectures provide a major advantage in the floating point performance. While NEON could perform a limited set of 128 bit vector instructions the introduction of SVE will put the floating point capabilities on par with the major architecture currently deployed for HPC.

At time of writing there is only one ARM processor supporting SVE [62]. However, the compilers and libraries are ready. Both GNU compilers and the ARM HPC compilers support SVE (Cray and Fujitsu should also support SVE but this is not tested in this guide). The code generation has been tested using a range of scientific applications and SVE code generation work well. The exact performance speedup remains to be evaluated due to lack of suitable hardware.

Most of the scientific applications need to be recompiled with SVE enabled. Few problems have ever sprung up when selecting SVE generation. From the compiler and building point of view there should be no major issues. However, without real hardware to run on the performance gain is hard to measure.

Table 5. Flags to enable SVE code generation

Compiler	SVE enabling lags
GNU	-march=armv8.1-a+sve
ARM HPC compiler	-march=armv8.1-a+sve

The SVE flag apply to all armv8-a and later versions. As the vector length in the current hardware is unknown the GNU flag "-msve-vector-bits" is set to "scalable" as default (-msve-vector-bits=scalable). This flag can be set to match the current hardware but this is not recommended, as the hardware vector length can vary and can be set at boot time. Best practice is to leave the vector length unknown and build vector length agnostic code.

2.2.3.1. Examples of SVE code generation

Consider a commonly known loop (Stream benchmark):

```
      DO 60 j = 1,n
         a(j) = b(j) + scalar*c(j)
 60   CONTINUE
```

The compiler will generate NEON instructions if "+simd" is used, look at the Floating-point fused multiply-add to accumulator instruction (flma) and fused multiply-add vectors (fmad).

```
.LBB0_5:                                // %vector.body26
                                        // =>This Inner Loop Header: Depth=1
        .loc    1 27 1 is_stmt 1        // fma-loop.f:27:1
        ldp     q1, q2, [x21, #-16]
        ldp     q3, q4, [x20], #32
        add     x21, x21, #32           // =32
        subs    x19, x19, #4            // =4
        fmla    v3.2d, v0.2d, v1.2d
        fmla    v4.2d, v0.2d, v2.2d
        stp     q3, q4, [x22, #-16]
        add     x22, x22, #32           // =32
        b.ne    .LBB0_5
```

while using the "+sve" flag SVE instructions will be generated.

```
.LBB0_5:                                // %vector.body30
                                        // =>This Inner Loop Header: Depth=1
        add     x11, x19, x8, lsl #3
        .loc    1 27 1 is_stmt 1        // fma-loop.f:27:1
        add     x12, x11, x9
        ld1d    { z1.d }, p0/z, [x12]
        ld1d    { z2.d }, p0/z, [x11]
        .loc    1 28 1                  // fma-loop.f:28:1
        incd    x8
        .loc    1 27 1                  // fma-loop.f:27:1
        add     x11, x11, x10
        fmad    z1.d, p2/m, z0.d, z2.d
        st1d    { z1.d }, p0, [x11]
        whilelo p0.d, x8, x20
        b.mi    .LBB0_5
```

It's interesting to see the nice use of predicate register to handle the remainder loop, see [72] for some documentation about this topic. The usage of SVE is a large topic. An own tutorial would be needed just to cover the basics, for a starting point see [73]. However, tests have shown that SVE instructions are generated by the compilers and hence a significant performance boost should be expected. Other architectures like Power and x86-64 have had vector units for many years and there is lot of experience to be the drawn from.

The command lines used to generate these codes were (SVE was introduced in v8.2-a) :

```
armflang -O3 -S -g -march=armv8.2-a+simd -o fma-loop.S fma-loop.f
```

```
armflang -O3 -S -g -march=armv8.2-a+sve -o fma-loop.S fma-loop.f
```

The ARM HPC compiler libraries are also SVE enabled from version 19.3 and later.

2.3. Benchmark performance

2.3.1. STREAM - memory bandwidth benchmark - Kunpeng 920

The STREAM benchmark [126], developed by John D. McCalpin of Texas Advanced Computing Center, is widely used to demonstrate the system's memory bandwidth via measuring four long vector operations as below:

```
Copy: a(i) = b(i)
Scale: a(i) = q * b(i)
```

```
Sum: a(i) = b(i) + c(i)
Triad: a(i) = b(i) + q * c(i)
```

The following table shows the obtained memory bandwidth measured using the OpenMP version of the STREAM benchmark. The table shows the highest bandwidth obtained using different settings found for processor bindings. The bar chart illustrated in Figure 4 shows the need for thread/rank core bindings. It's vital to get this right in order to get a decent performance.

The STREAM benchmark was built using gcc with OpenMP support and compiled with a moderate level of optimisation, -O2.

Table 6. Stream performance, all numbers in MB/s

Processor	Binding	Copy	Scale	Add	Triad
Kunpeng 920	PROC BIND	322201	322214	324189	323997
Kunpeng 920	GOMP_CPU_AFFINITY=0-127	275666	275142	282769	280507
Kunpeng 920	Numactl -l	237508	238412	232506	232840

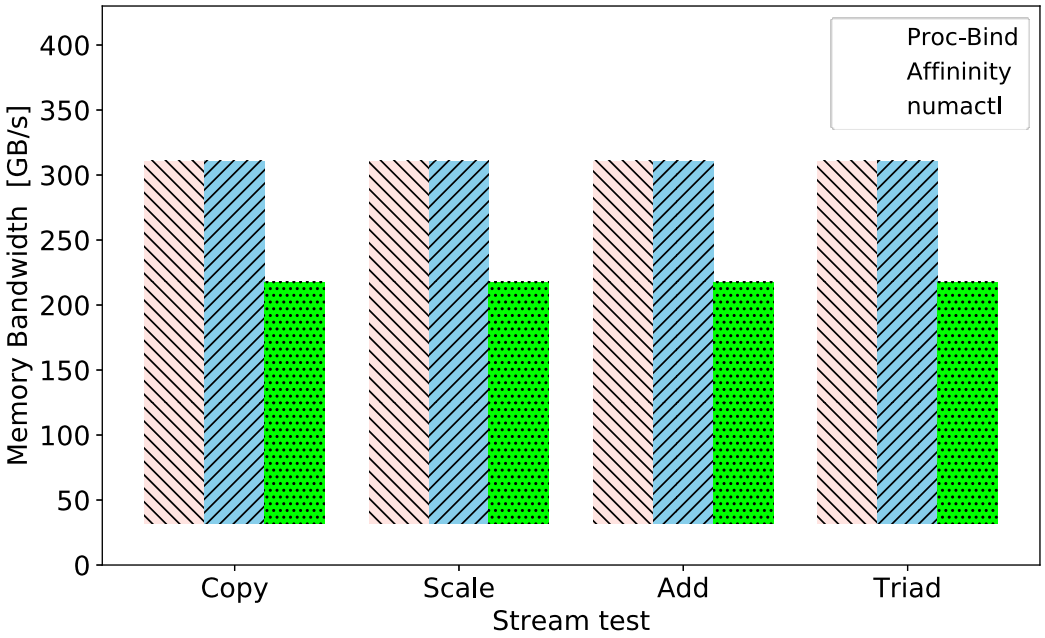


Figure 4. Stream memory bandwidth Kunpeng 920, OpenMP, 128 cores, 224 GiB footprint (87% of installed RAM)

The table and figures show that thread binding is important, the actual settings of the binding is less important. Just setting the environment variable OMP_PROC_BINDING to "1" or "true" is making all the difference.

The memory bandwidth performance numbers are impressive compared to the majority of currently available systems. For applications that are memory bound this relatively high memory bandwidth will yield a performance benefit. From the benchmark results published by the vendors it is also evident that the ARM systems have memory bandwidth advantage.

2.3.2. STREAM - memory bandwidth benchmark - Thunder X2

Here, we show STREAM performance again for the Thunder X2 processor, but using the OpenMPI version rather than the OpenMP version. Using the OpenMPI version allows scaling and benchmarking across nodes, although in this case we are strictly focused on the single node case.

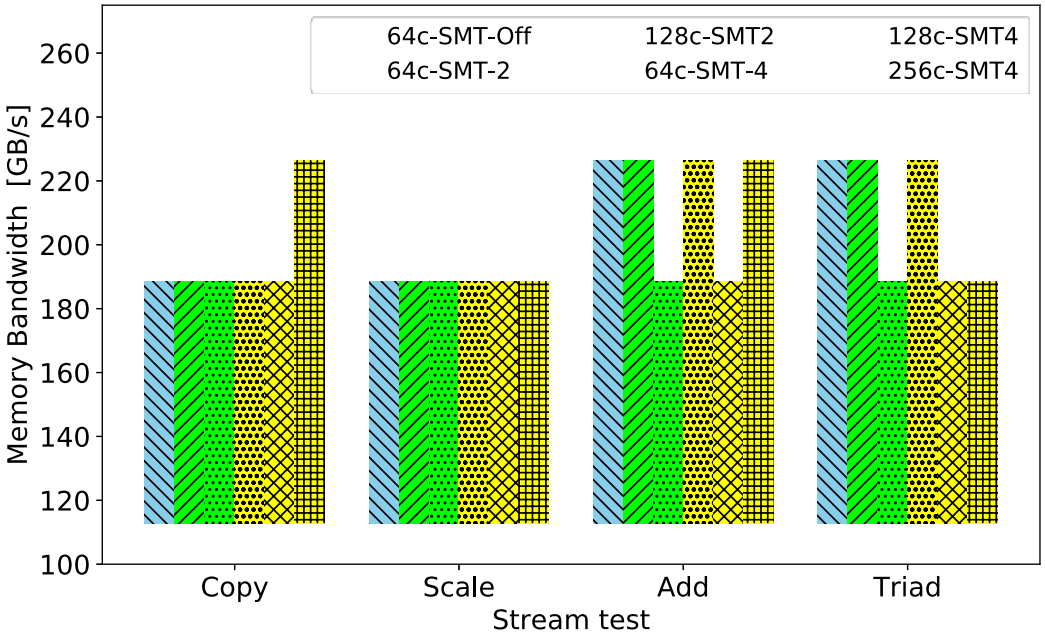


Figure 5. Stream memory bandwidth on ThunderX2 with OpenMPI, Size N=400000000, including SMT settings impact

The above performance plot shows MPI STREAM for 3 different SMT configurations – SMT-1 (up to 64 processes), SMT-2 (up to 128 processes, 2 hardware threads per core) and SMT-4 (up to 256 processes, 4 hardware threads per core). The array size that was chosen for this test is 400 million elements, which satisfies the rule that the memory requirements for the test should be greater than 3 times the LLC size. For this test, STREAM was compiled with GCC v9.2 and OpenMPI v4.0.2, using the compilation flags:

```
-O2 -mcmodel=large -fno-PIC
```

The results shown are the maximum performance achieved over 5 runs. As can be seen from the performance numbers, the highest bandwidth is achieved when using 64 MPI processes, regardless of the SMT configuration. The performance drops by almost 10 MB/s when using 256 MPI processes in SMT-4 mode; given that the SMT-4 configuration is using all the available hardware threads, this is still very good performance.

2.3.3. High Performance Linpack

The table below shows the results from running High Performance Linpack (HPL) on Fulhame. The tests were run with different SMT values but with the processor frequency set at 2.5 GHz, i.e. it's highest value. To achieve this, memory turbo was disabled to allow the frequency to boost above the nominal 2.2 GHz. Compilation was performed with GCC 9.2 and OpenMPI 4.0.2 linking to the ARM performance libraries.

Table 7. HPL performance numbers for Fulhame. All numbers in GFLOPS

SMT | Ranks	64	128	256
Off	742	N/A	N/A
2	741	626	N/A
4	636	636	548

As might be reasonably expected, SMT-1 gives the best performance results. All hardware cores are fully engaged and there is no contention on their resources from hyper-threads. For comparison, with the frequency fixed at 1.0 GHz (the lowest available on the ThunderX2), the performance of the SMT-4 with 256 processors case is 184 GFLOPS.

2.4. MPI Ping-pong performance using RoCE

RDMA (Remote Direct Memory Access) [77] over Converged Ethernet (RoCE) [78] has attracted a lot of interest recently. The cost compared to InfiniBand is lower and for smaller [2] clusters the performance is acceptable. The testing below was done on the Kunpeng 920 system.

The MPI ping pong latency is measured at about 1.5 microseconds, the bandwidth is approaching wire speed at peak bandwidth with medium to large sized MPI messages.

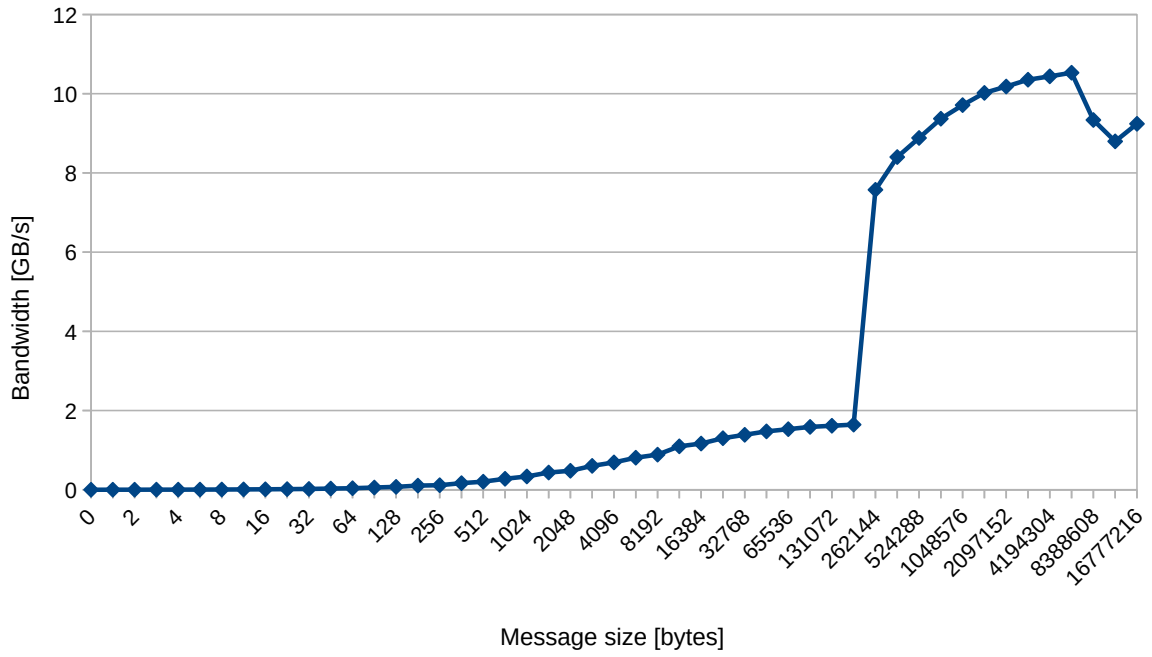


Figure 6. MPI bandwidth using two nodes with RoCE

Figure 6 shows typical profile for MPI ping-pong measurements, with the well known transition where transport mechanism is changed. For MPI messages at one Megabyte the bandwidth approaches wire speed.

For many small to medium sized clusters with applications not very sensitive to latency Converged Ethernet with Remote Memory Access could be an option.

2.5. HPCG - High Performance Conjugated Gradients

The High Performance Conjugate Gradients (HPCG) benchmark [123] is an alternative to HPL which is currently also used to rank the fastest systems in the world. HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of current large-scale HPC applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications.

The ARM based systems are known to perform well when running HPCG mainly due to the high memory bandwidth.

The Kunpeng 920 results are obtained from a small cluster of four Kunpeng 920 nodes using 100 GbE RoCE as interconnect. HPCG is built using ARM HPC 19.3 compiler and libraries.

[2] what this number of nodes is debated, but departmental clusters of 16-32 nodes are seen as small.

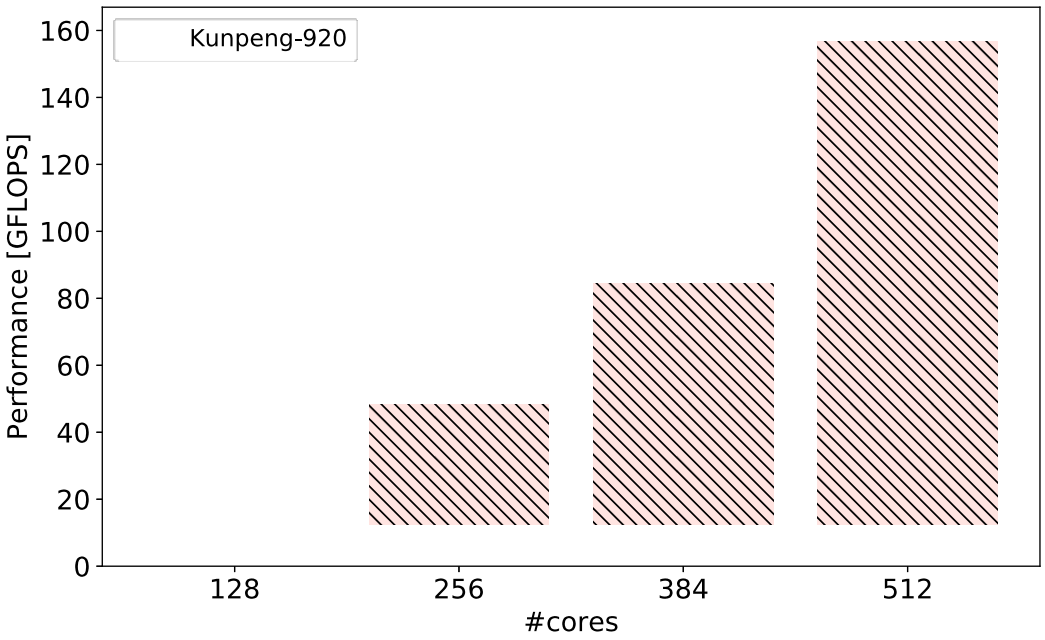


Figure 7. HPCG (MPI) performance using four Kunpeng 920 nodes with RoCE, 100 GbE, ARM performance libraries

The ThunderX2 results are obtained from a single ThunderX2 node. The number of cores indicate number of apparent cores e.g. including the logical SMT cores. At 256 cores SMT-4 is used.

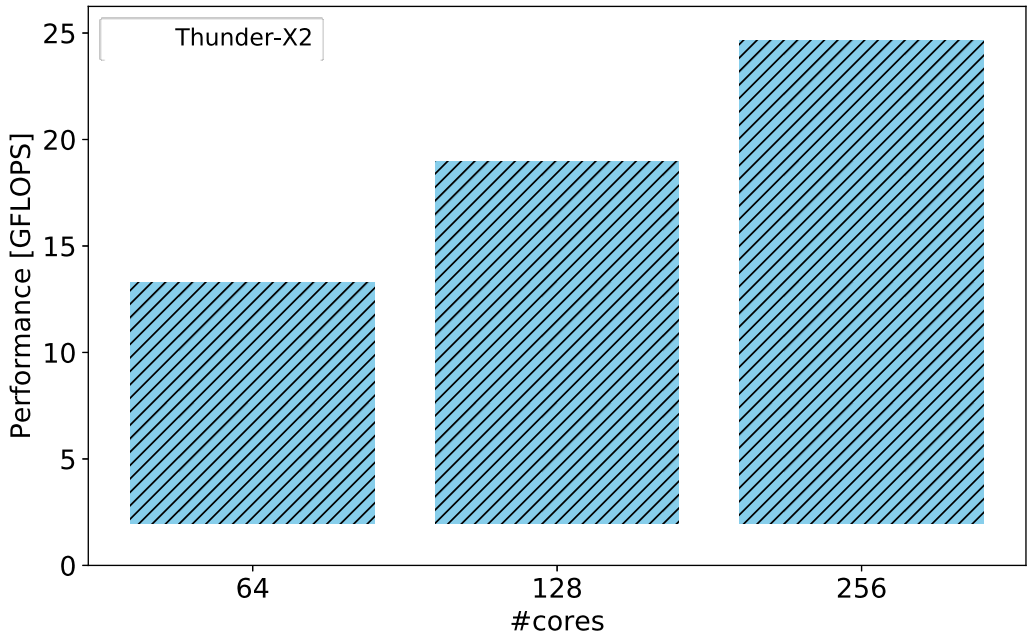


Figure 8. HPCG (MPI) performance using a single ThunderX2

2.6. Simultaneous Multi Threading (SMT) performance impact

Memory bandwidth is an important parameter of a system, and the impact of threaded memory bandwidth with respect to the Simultaneous Multi Threading (SMT) should be documented.

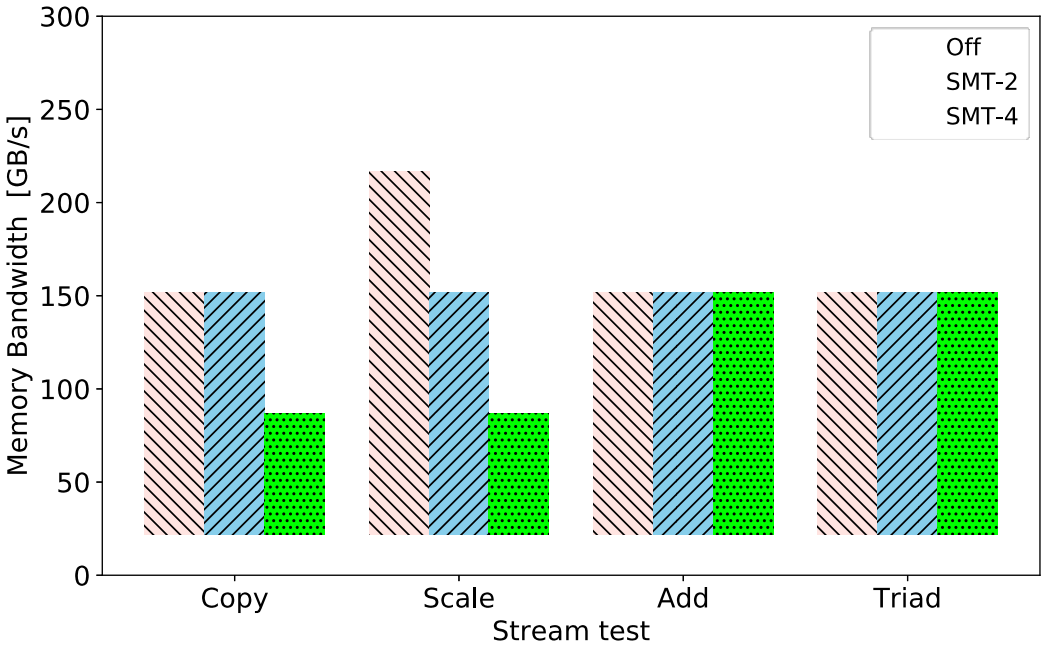


Figure 9. Memory bandwidth impact of SMT setting using OpenMP version of Stream

The Simultaneous Multi Threading (SMT) supported on the ThunderX2 processor has impact on application performance. Some benchmarks and applications have been run to explore the impact of this setting. The runs were done using all cores (64, 128 and 256 in the different SMT settings, this might explain the poor scaling for NPB in OpenMP versus good scaling in MPI versions).

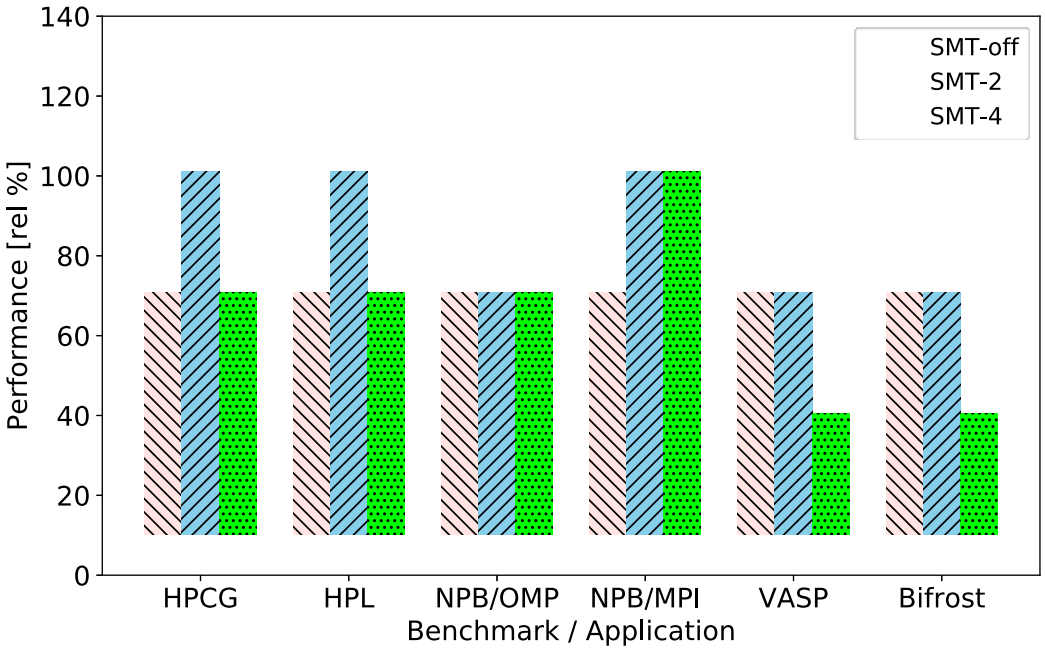


Figure 10. Performance impact on benchmarks and application (OpenMP and MPI) of SMT setting

Lesson from the figure above is that memory bandwidth is affected by the SMT setting and the highest bandwidth is obtained when SMT is turned off, but the penalty for using SMT-2 is relatively small. While some benchmarks seem to benefit from turning SMT on, the applications seem tp perform better with the SMT turned off.

When running tests with different core counts one is not only testing scalability performance of the cores, but also the scalability of the application. MPI application generally scale better than OpenMP applications. This affect the performance when running at higher core count. In addition the usage of SMT can show different impact on threaded OpenMP applications than on MPI applications. When testing the SMT impact the thread count with best performance were reported, this might not be all threads. The total performance of the compute node is the number of interest at the given SMT.

The figure below shows how SMT setting might affect application performance, Bifrost [130]. The code on test is known to scale well beyond the numbers in this work. Consequently the performance differences seen should only be attributed to the processor and node itself. It is also known to be sensitive to memory bandwidth and Figure 9 show that SMT settings impact the memory bandwidth.

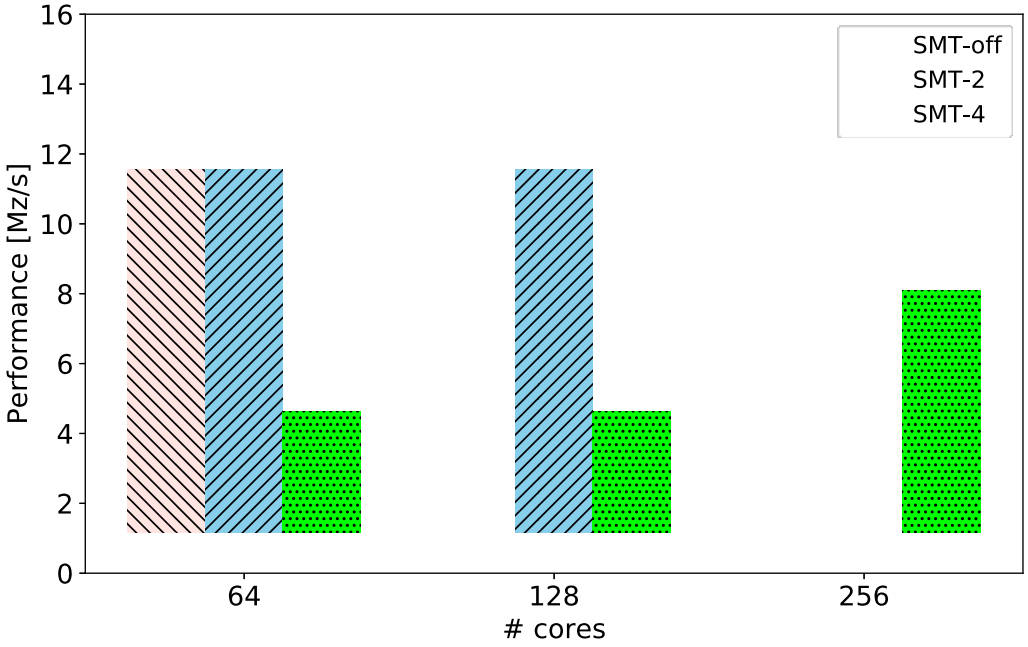


Figure 11. Application performance impact of SMT setting using the stellar atmosphere simulation MPI code Bifrost [130]

From the numbers presented in the figure above it's clear that turning the SMT off yields the best performance. Similar numbers have been reported for other applications like the material science code VASP.

2.7. IOR

The IOR benchmark suite is used to test file system performance on a variety of different supercomputers. It is easily configurable and uses MPI to coordinate multiple processes each reading or writing to stress a file system.

For this test, a single Thunder X2 node was used, with SMT-1. The version v3.2.1 of the benchmark was compiled with the GCC v9.2 toolchain and OpenMPI v4.0.2. Tests were carried out against a HDD-based LUSTRE file system mounted across an InfiniBand link with the rest of the system idle. Two different tests were configured and run, File Per Process (FPP) - one file per MPI rank and Single shared file (SF) - a single shared MPI-IO file.

Table 8. IOR performance. HDD-based LUSTRE Filesystem. All numbers in MiB/s

Num. Procs	Read (FPP)	Write (FPP)	Read (SF)	Write (SF)
1	921	962	1090	966
2	1422	1114	1400	1106
4	2062	1089	2411	1068
8	2729	1027	4173	818
16	3043	777	2959	425
32	4921	803	2923	303
64	3880	645	4482	321

2.8. European ARM processor based systems

2.8.1. Fulhame (EPCC)

Fulhame, named for Elizabeth Fulhame, a Scottish chemist who invented the concept of catalysis and discovered photoreduction, is a 64-node fully ARM-based cluster cluster housed at EPCC's ACF datacentre. It is part of the

Catalyst collaboration between HPE, ARM, Cavium, Mellanox and three UK universities: Edinburgh, Bristol and Leicester.

2.8.1.1. System Architecture / configuration

The Fulhame system architecture is identical to the other Catalyst systems. They comprise 64 compute nodes, each of which has two 32-core Cavium ThunderX2 processors. These processors run at a base frequency of 2.1 GHz with a boost available of up to 2.5 GHz controlled via Dynamic Voltage and Frequency Scaling (DVFS). Each processor has eight 2666 MHz DDR4 channels and runs with 128 GiB of memory giving 4 GiB/core (1 GiB/thread in SMT-4) for a total of 256 GiB/node. Fulhame uses Mellanox Infiniband for the interconnect between compute nodes, login/admin nodes and the storage system. Fulhame and the other Catalyst systems are fully ARM-based and use Thunder X2 nodes for storage servers and all login/admin nodes.

2.8.1.2. System Access

Account requests can be made to the EPCC Helpdesk: epcc-support@epcc.ed.ac.uk

2.8.1.3. Production Environment

The production environment on Fulhame is SuSE Linux Enterprise Server, v15.0. SuSE is a partner is the Catalyst project and provides support for this architecture specifically. The other two Catalyst systems run SLES 12sp3.

2.8.1.4. Programming Environment

The programming environment is that of SLES v15.0 with the GNU v9.2 suite of compilers and libraries. ARM supply v20.0 of their performance libraries and compilers, with the libraries providing modules for both GNU and ARM toolchains. Libraries, compilers and software are provided via the normal Linux modules environment. Mellanox provide the relevant performance InfiniBand drivers and libraries directly to the Catalyst sites for this architecture.

3. Processors Intel Skylake

Skylake is the codename for a processor microarchitecture developed by Intel as the successor to the Broadwell microarchitecture. Skylake is a microarchitecture redesign using the same 14 nm manufacturing process technology as its predecessor, serving as a "tock" in Intel's "tick–tock" manufacturing and design model, and it was launched in August 2015. Breaking with this "tick–tock" manufacturing and design model, Intel presented Kaby Lake in August 2016, a processor microarchitecture that represents the optimised step of the newer "process-architecture-optimisation" model. Skylake CPUs share their microarchitecture with Kaby Lake, Coffee Lake and Cannon Lake CPUs. This guide concentrates on features common to Skylake and its optimisations.

3.1. Architecture

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 12 [117].

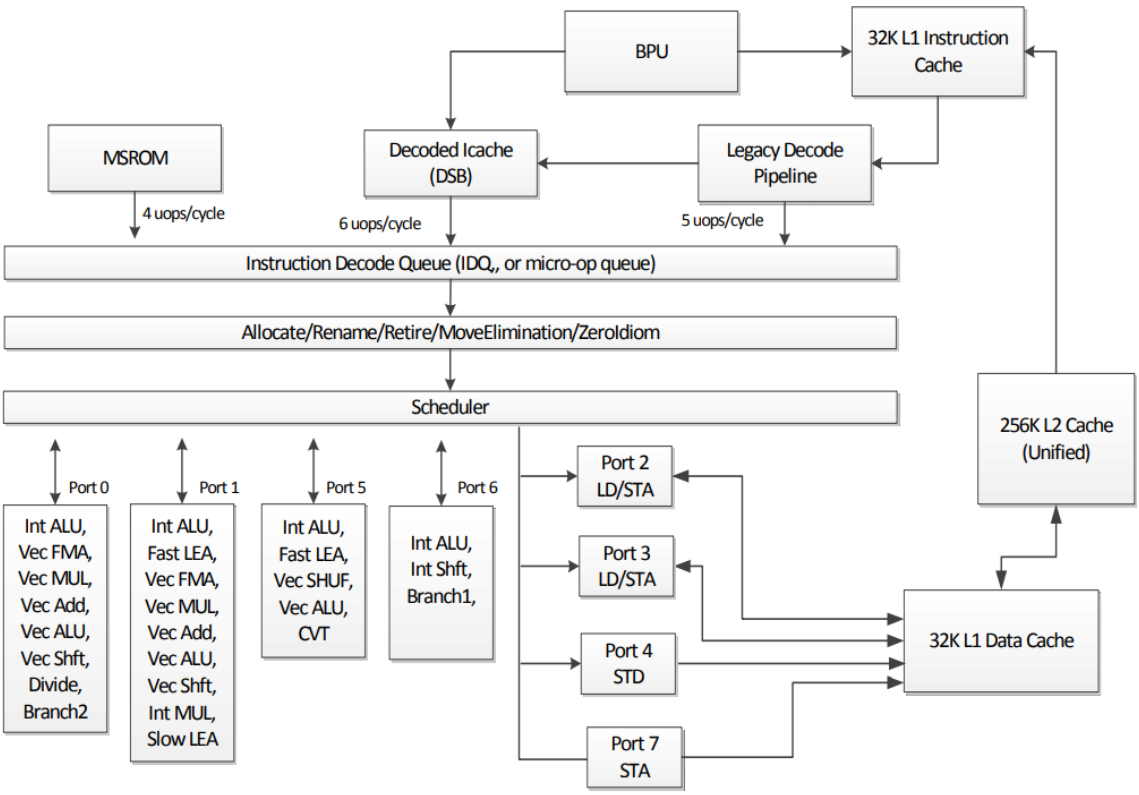


Figure 12. CPU Core Pipeline Functionality of the Skylake Microarchitecture

The Skylake microarchitecture introduces the following new features:

- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) support.
- More cores per socket (max 28 vs. max 22).
- 6 memory channels per socket in Skylake microarchitecture vs. 4 in the Broadwell microarchitecture.
- Bigger L2 cache, smaller non inclusive L3 cache.
- Intel® Optane™ support.
- Intel® Omni-Path Architecture (Intel® OPA)

3.1.1. Memory Architecture

Skylake Server microarchitecture implements a mid-level (L2) cache of 1 MiB capacity with a minimum load-to-use latency of 14 cycles. The mid-level cache capacity is four times larger than the capacity in previous Intel Xeon processor family implementations. The line size of the mid-level cache is 64 B and it is 16-way associative. The mid-level cache is private to each core. Software that has been optimised to place data in mid-level cache may have to be revised to take advantage of the larger mid-level cache available in Skylake Server microarchitecture.

The last level cache (LLC) in Skylake is a non-inclusive, distributed, shared cache. The size of each of the banks of last level cache has shrunk to 1.375 MiB per bank. Because of the non-inclusive nature of the last level cache, blocks that are present in the mid-level cache of one of the cores may not have a copy resident in a bank of last level cache. Based on the access pattern, size of the code and data accessed, and sharing behavior between cores for a cache block, the last level cache may appear as a victim cache of the mid-level cache and the aggregate cache capacity per core may appear to be a combination of the private mid-level cache per core and a portion of the last level cache.

3.1.2. Power Management

Skylake microarchitecture dynamically selects the frequency at which each of its cores executes. The selected frequency depends on the instruction mix; the type, width, and number of vector instructions that execute over a given period of time. The processor also takes into account the number of cores that share similar characteristics. Intel® Xeon® processors based on Broadwell microarchitecture work similarly, but to a lesser extent since they only support 256-bit vector instructions. Skylake microarchitecture supports Intel® AVX-512 instructions, which can potentially draw more current and more power than Intel® AVX2 instructions. The processor dynamically adjusts its maximum frequency to higher or lower levels as necessary, therefore a program might be limited to different maximum frequencies during its execution. Intel organises workloads into three categories:

- Intel® AVX2 light instructions: Scalar, AVX128, SSE, Intel® AVX2 w/o FP or INT MUL/FMA
- Intel® AVX2 heavy instructions + Intel® AVX-512 light instructions: Intel® AVX2 FP + INT MUL/FMA, Intel® AVX-512 without FP or INT MUL/FMA
- Intel® AVX-512 heavy instructions: Intel® AVX-512 FP + INT MUL/FMA

Figure 13 is an example for core frequency range in a given system where each core frequency is determined independently based on the demand of the workload. The maximum frequency (P0n) is an array of frequencies which depend on the number of cores within the category. In the case of the Intel Xeon Platinum 8160 CPU and with all the cores running the same workload, the P0 for the non-AVX, the AVX2 and the AVX-512 are 2.8 GHz, 2.5 GHz and 2.0 GHz respectively.

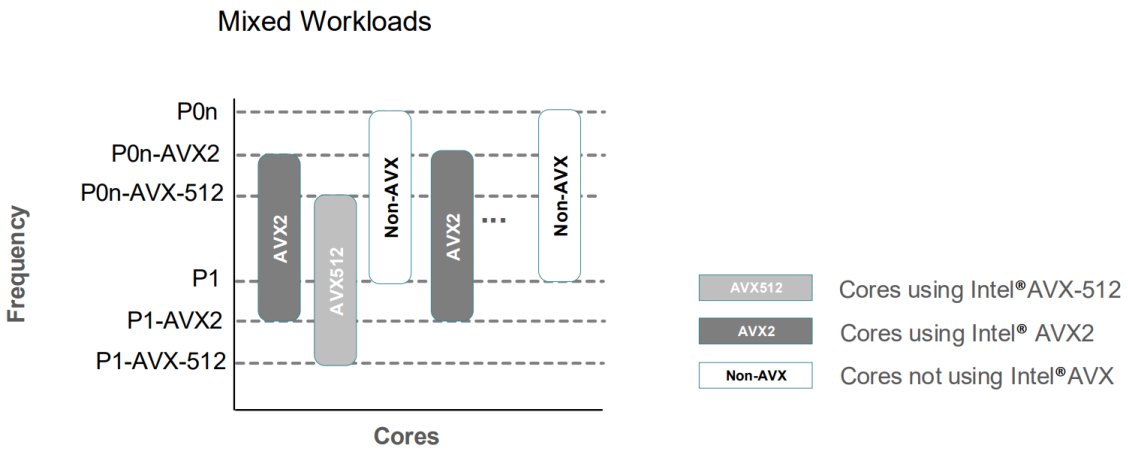


Figure 13. Skylake frequencies with mixed workloads

3.2. Programming Environment

3.2.1. Compilers

All of the compilers described in the following support C, C++ and FORTRAN (90 and 2003), all with OpenMP threading support. On most of the applications running on Skylake, we observed a better performance using the Intel Compilers suite.

Compilers evaluated

- Intel compiler suite (*icc, ifortran, icpc). Version 2017.4*
- GNU compiler suite (*gcc, gfortran, g++). Version 7.2.0*

The suggested flags for the evaluated compilers are:

Table 9. Suggested compiler flags

Compiler	flags
Intel Compilers	-mtune=skylake -xCORE-AVX512 -m64 -fPIC
GNU compiler	-march=skylake -O3 -mfma

3.2.1.1. Effect of using AVX-512 instructions

As we commented on the previous section, Skylake microarchitecture dynamically selects the frequency depending on the instruction set used. Some workloads do not cause the processor to reach its maximum frequency as these workloads are bound by other factors. For example, the HPL benchmark is power limited and does not reach the processor's maximum frequency. The Table 10, “HPL Performance with different instruction set” shows how frequency degrades as vector width grows, but, despite the frequency drop, performance improves. The data for this graph was collected on an Intel Xeon Platinum 8180 processor.

Table 10. HPL Performance with different instruction set

Instruction set	Performance (GFLOPS)	Power (W)	Frequency (GHz)
AVX	1178	768	2.8
AVX2	2034	791	2.5
AVX-512	3259	767	2.1

It is not always easy to predict whether a program's performance will improve from building it to target Intel AVX-512 instructions. Some programs that use AVX-512 instructions may not gain as much, or may even lose performance. Intel recommends to try multiple build options and measure the performance of the program. To identify the optimal compiler options to use, you can build the application with each of the following set of options and choose the set that provides the best performance:

Table 11. Compiler flags for different instruction sets

Instruction set	Intel compiler flags	GCC compiler flags
AVX	-xAVX	-mprefer-vector-width=128
AVX2	-xCORE-AVX2	-mprefer-vector-width=256
AVX-512	-xCORE-AVX512	-mprefer-vector-width=512

3.2.1.2. Compiler Flags for debugging and profiling

In this section we present the compiler flags needed for profiling and/or debugging applications. The table below shows common flags for most compilers:

Table 12. Compiler Flags

-g	Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.
-p	Generate extra code to write profile information suitable for the analysis program prof. You must use this option when compiling the source files you want data about, and you must also use it when linking.
-pg	Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

3.2.2. Available Numerical Libraries

The most common numerical libraries for Intel based systems are the Intel Math Kernel Library, MKL [35] and the Intel Performance Primitives (IPP) [34]. The MKL library contains a large range of high level functions like Basic Linear Algebra, Fourier Transforms etc, while IPP contains a large number of more low level functions for e.g. converting or scaling. For even lower level functions like scalar and vector versions of simple functions like square root, logarithmic and trigonometric functions there are libraries like libimf and libsvml.

3.2.2.1. Math Kernel Library, MKL

Intel MKL is an integrated part of the Intel compiler suite. The simplest way of enabling MKL is to just issue the flag *-mkl*, this will link the default version of MKL. For a multicore system this links the threaded version. There are both a single threaded sequential version and a threaded multicore version available. The sequential version is very often used with non-hybrid MPI programs. The parallel version honors the environment variable OMP_NUM_THREADS.

Table 13. Invoking different versions of MKL

MKL Version	Link flag
Single thread, sequential	-mkl=sequential
Multi threaded	-mkl=parallel or -mkl

MKL contains a huge range of functions. Several of the common widely used libraries and functions have been incorporated into MKL.

Libraries contained in MKL:

- BLAS and BLAS95
- FFT and FFTW [121] (wrapper)
- LAPACK
- BLACS
- ScaLAPACK
- Vector Math

Most of the common functions that are needed are part of the MKL core routines, Basic linear algebra (BLAS 1,2,3), FFT and the wrappers for FFTW. Software often requires the FFTW package. With the wrappers there is no need to install FFTW, which is outperformed by MKL in most cases.

MKL is very easy to use, the functions have simple names and the parameter lists are well documented. An example of a matrix matrix multiplication is shown below:

```
call dgemm('n', 'n', N, N, N, alpha, a, N, b, N, beta, c,N)
```

Another example using FFTW syntax:

```
call dfftw_plan_dft_r2c_2d(plan,M,N,in,out,FFTW_ESTIMATE)
call dfftw_execute_dft_r2c(plan, in, out)
```

The calling syntax for the matrix matrix multiplication is just as for dgemm from the reference Netlib BLAS implementation. The same applies to the FFTW wrappers and other functions. Calling semantics and parameter lists are kept as close to the reference implementation as practically possible.

The usage and linking sequence of some of the libraries can be somewhat tricky. Please consult the Intel compiler and MKL documentation for details. There is a Intel Math Kernel Library Link Line Advisor [https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor] available from Intel. An example for building the application VASP is given below (list of object files contracted to *.o):

```
mpiifort -mkl -lstdc++ -o vasp *.o -Llib -ldmy\
-lmkl_scalapack_lp64 -lmkl_blacs_intelmpi_lp64 -lfftw3xf_intel_lp64
```

or even simpler for a FFTW example:

```
ifort fftw-2d.f90 -o fftw-2d.f90.x -mkl
```

3.3. Benchmark performance

3.3.1. MareNostrum system

Figure 14 shows the power consumption and performance of a MareNostrum (described in Section 3.6.1) compute node when executing HPL benchmark across distinct maximum CPU frequencies.

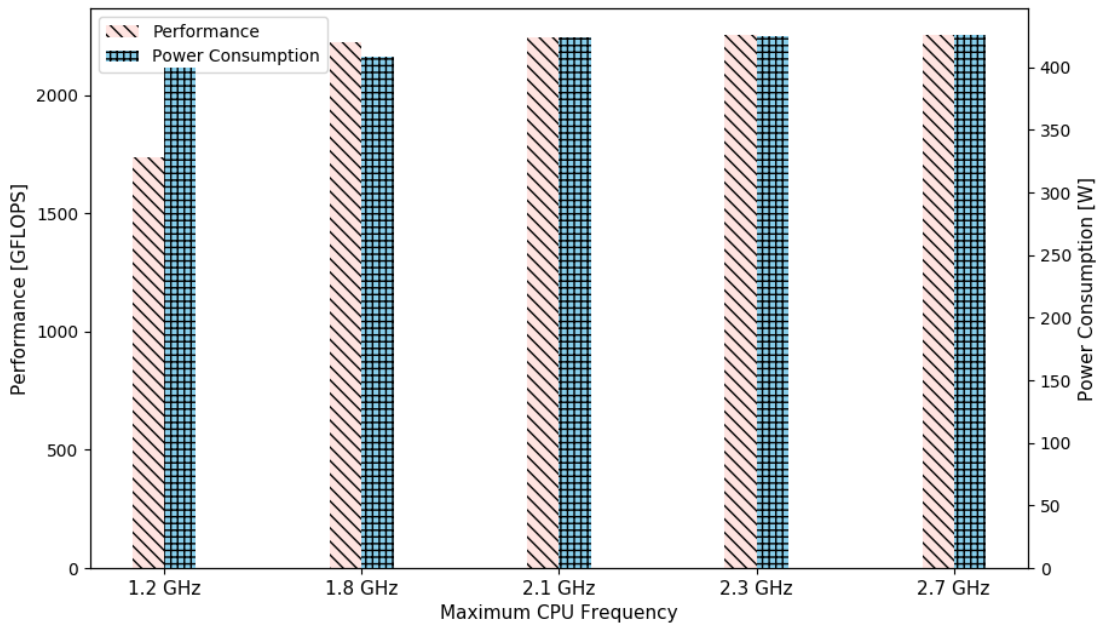


Figure 14. Performance and average power consumption of a MareNostrum compute node across distinct maximum CPU frequencies when executing HPL benchmark

As it can be seen, the performance and the power consumption of the HPL benchmark increase with higher frequencies, but only up to 2.1 GHz. Then, the performance and power consumption stay constant. By default, the MareNostrum nodes have a maximum frequency limit of 2.1 GHz, and in spite of the HPL benchmark run on an unlimited node in terms of frequency, we can observe how it is limited due to the power consumption.

Figure 15 shows the power consumption and performance of a MareNostrum compute node when executing HPCG benchmark across distinct maximum CPU frequencies.

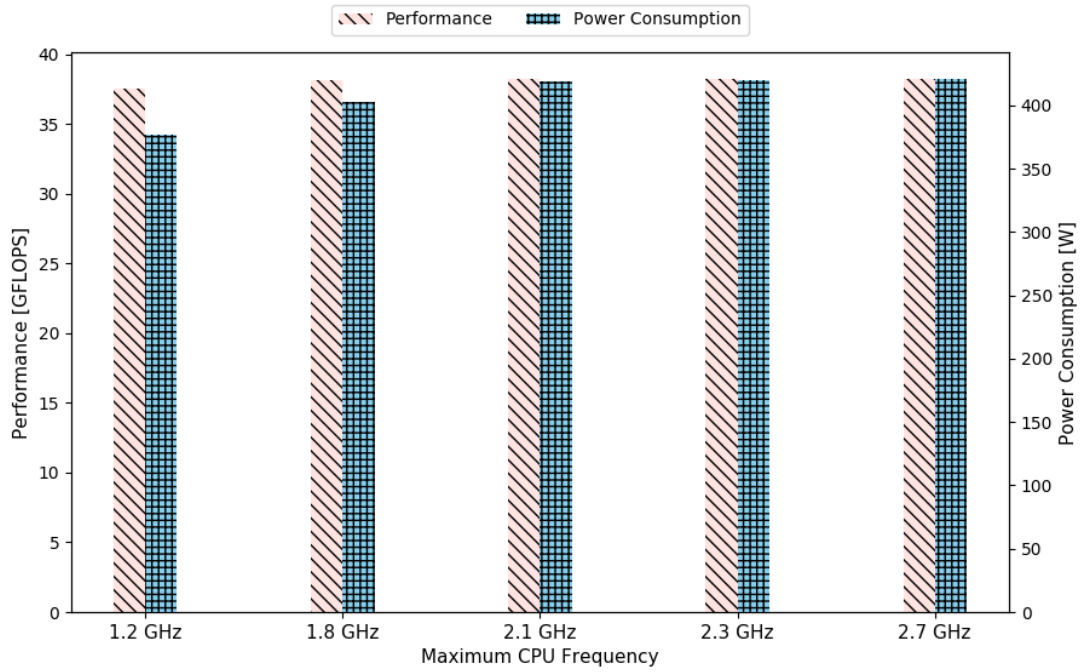


Figure 15. Performance and average power consumption of a MareNostrum compute node across distinct maximum CPU frequencies when executing HPCG benchmark

As HPCG is not as computationally intense as HPL, we can observe that the performance does not vary much and it is around 38 GFLOPS with the different frequencies. Although, power consumption has the same behaviour that we can see with the HPL benchmark.

Figure 16 shows the power consumption of a MareNostrum compute node when executing FIRESTARTER benchmark across distinct maximum CPU frequencies. We observe a similar behavior as seen on the HPL benchmark, but in this case, the benchmark reaches the maximum power consumption with 1.8 GHz.

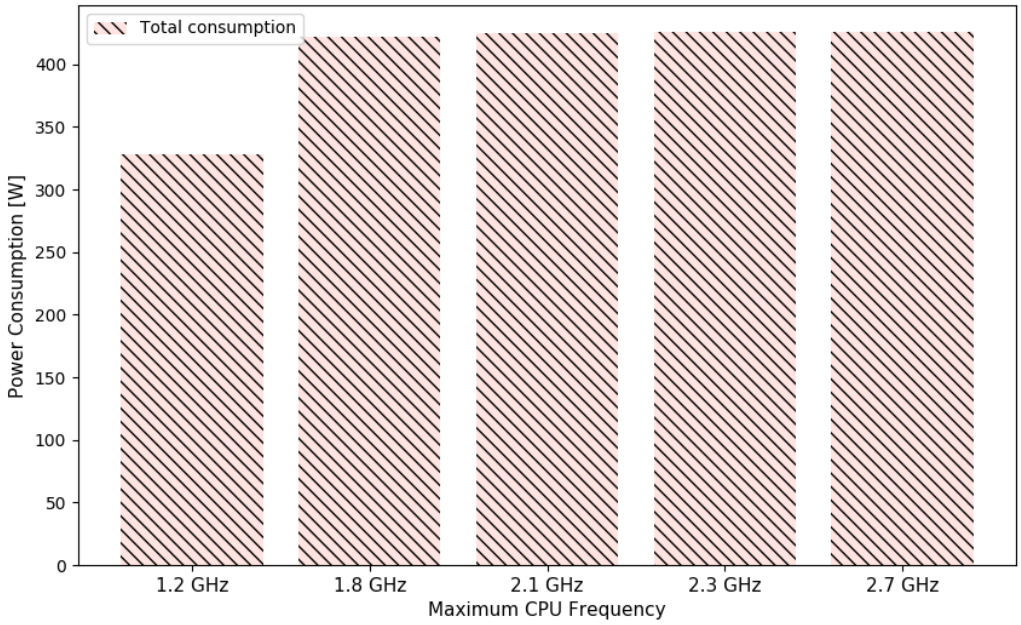


Figure 16. Average power consumption of a MareNostrum compute node across distinct maximum CPU frequencies when executing FIRESTARTER benchmark

Figure 17 shows the power consumption and performance of a MareNostrum compute node when executing STREAM benchmark across distinct maximum CPU frequencies. As expected, the performance variation is negligible when the frequency scales, as it had bee seen on the HPCG benchmark.

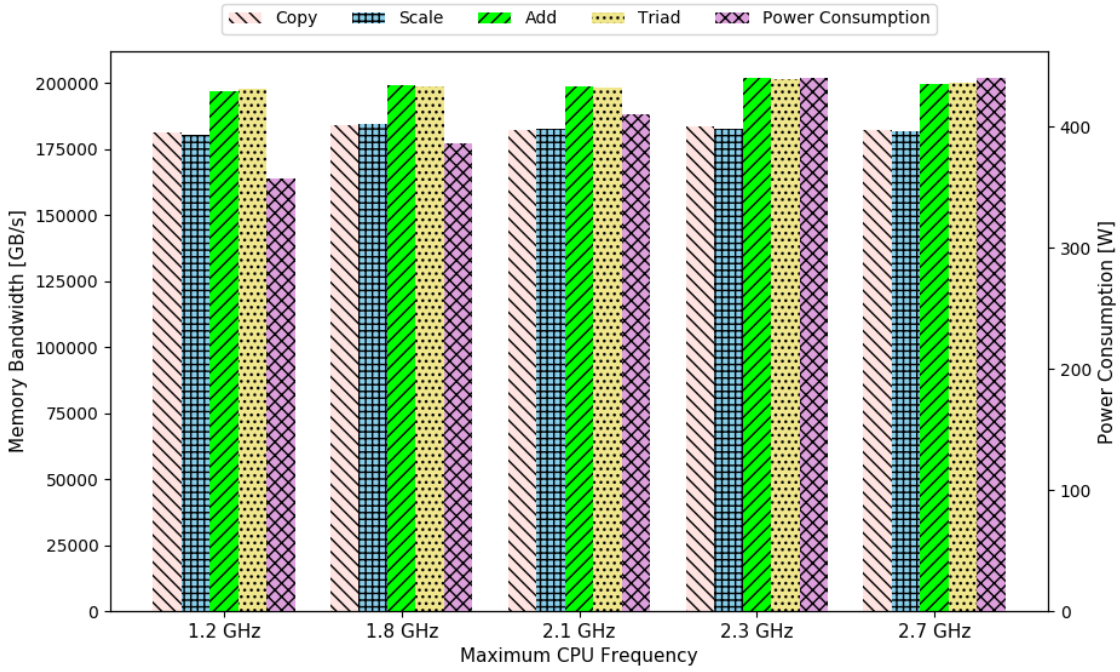


Figure 17. Performance and average power consumption of a MareNostrum compute node across distinct maximum CPU frequencies when executing STREAM benchmark

The table below shows the results from running IOR benchmark on a MareNostrum compute node across distinct maximum CPU frequencies. The Benchmark version was v3.2.1 compiled with the Intel Compiler and Intel MPI 2017.4.

Table 14. IOR performance. GPFS Filesystem.

Frequency (GHz)	Read (MiB/s)	Write (MiB/s)
1.2	28536.72	618.49
1.8	34485.54	3201.50
2.1	46776.62	7774.43
2.3	55492.45	7678.14
2.7	55431.33	7751.33

3.3.2. SuperMUC-NG system

Figure 18 shows the performance and average power consumption of a SuperMUC-NG (described in Section 3.6.2) compute node when executing HPL benchmark at different maximum CPU frequencies, namely at 1.2 GHz, 1.8 GHz, 2.1 GHz, 2.3 GHz, and 2.7 GHz. The data represents an average of 3 different HPL executions per maximum CPU frequency. Within each individual HPL run, the HPL tests were repeated 3 times. The maximum performance recording in 205 W TDP mode for this node is 2.9 TFLOPS.

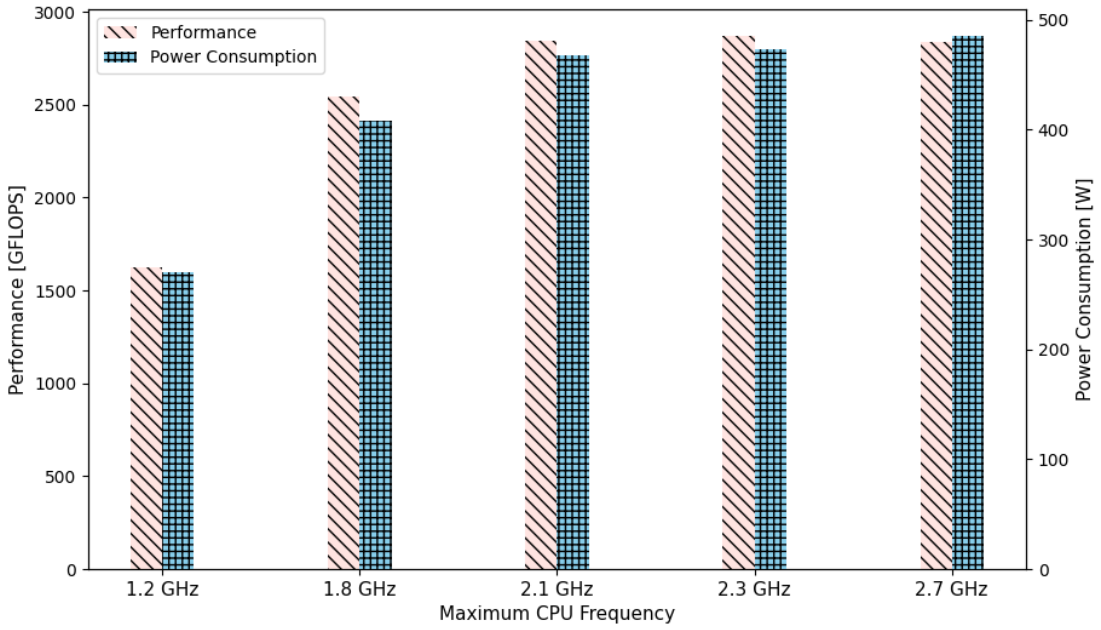


Figure 18. Performance and average power consumption of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing HPL benchmark

As can be seen from Figure 18, the TDP limit was already reached at 2.1 GHz maximum CPU frequency leading to performance saturation. Figure 19 and Figure 20 show the Average Power Consumption (APC) and Time-to-Solution (TtS) results when running single node HPL on SuperMUC-NG. As expected, APC increases and the execution time, i.e. TtS decreases with higher maximum CPU frequency limits.

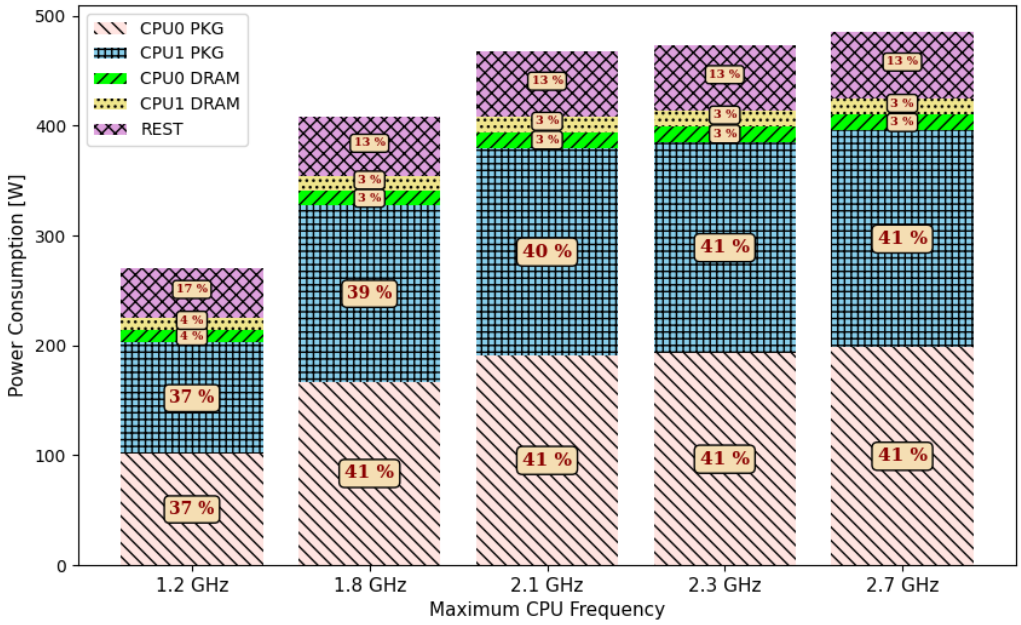


Figure 19. Average power consumption distribution of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing HPL benchmark

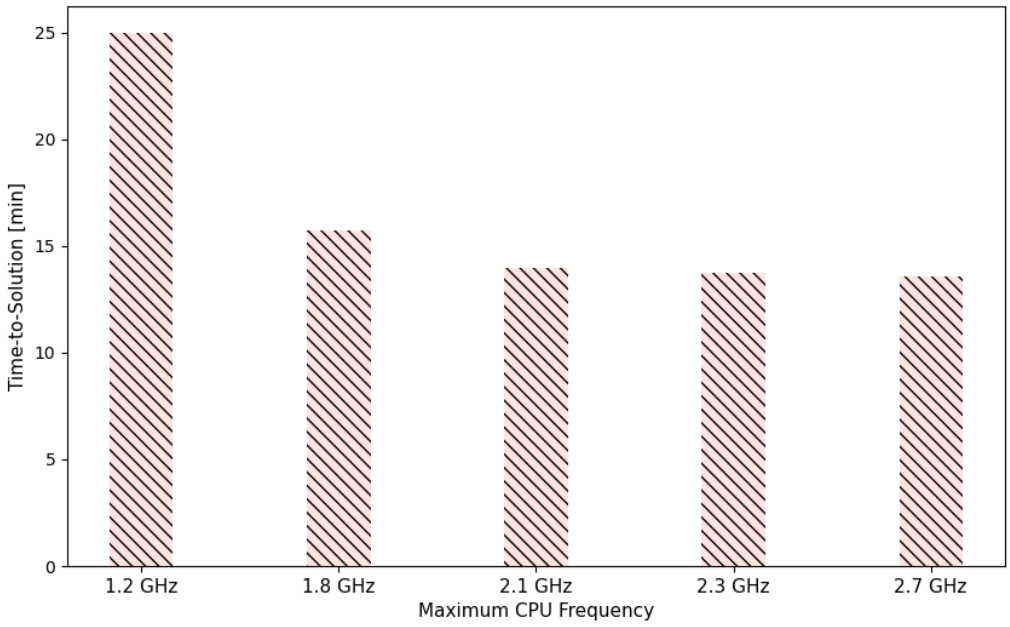


Figure 20. Time-to-Solution (TtS) bar chart of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing HPL benchmark

In addition to the recorded APC measurements, Figure 19 also illustrates the power consumption distribution across different components of the compute node. Given that each SuperMUC-NG compute nodes is equipped with 2 processors, the segments labeled as "CPU0 PKG" and "CPU1 PKG" correspondingly show the APC of the first and second processors. The segments labeled as "CPU0 DRAM" and "CPU1 DRAM" show the associated

averaged DRAM power consumption for first and second processors respectively. Finally, the segments labeled as "REST" and colored in green illustrate the average power consumption of the other components residing on the compute node. The per segment power consumption data was achieved with the help of LIKWID toolkit [147].

Figure 21 illustrates the power distribution of a SuperMUC-NG compute node when executing HPCG benchmark.

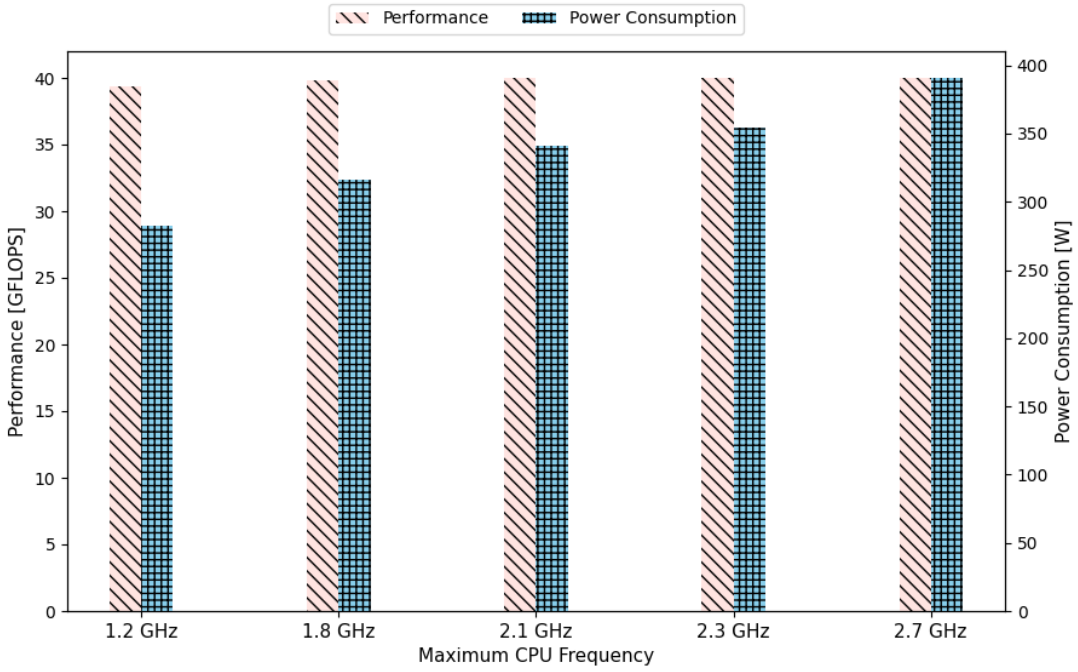


Figure 21. Performance and average power consumption of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing HPCG benchmark

Local domain dimensions for HPCG benchmark were set to 192x192x192, which translated to approximately 5.3 GB of RAM usage per socket. The average recorded performance for the considered compute node across different valid runs of HPCG benchmark resulted in 39.84 GFLOPS. On average, with the described configuration, it took 49.68 minutes to complete a single HPCG benchmark.

Figure 22 provides a detailed view on average power consumption distribution when executing HPCG benchmark. As expected, the aggregated average power consumption increases with maximum CPU frequency.

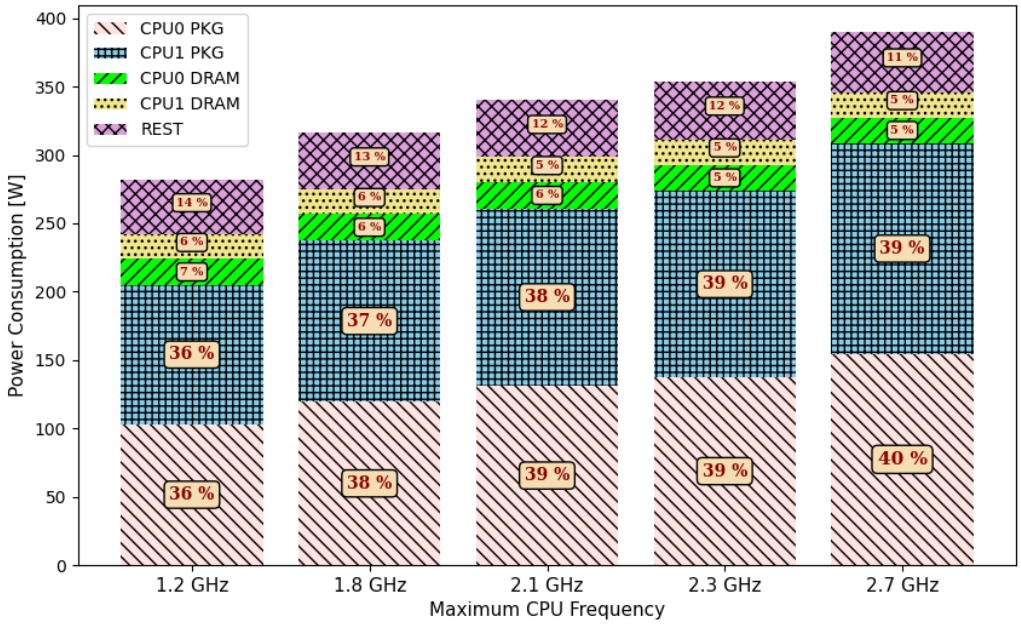


Figure 22. Average power consumption distribution bar chart of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing HPCG benchmark

Figure 23 shows the earlier mentioned average power consumption distributions of a SuperMUC-NG compute node when executing FIRESTARTER benchmark [125] across distinct maximum CPU frequencies. Here also, the data represents an average of 3 different runs each executing for 15 minutes.

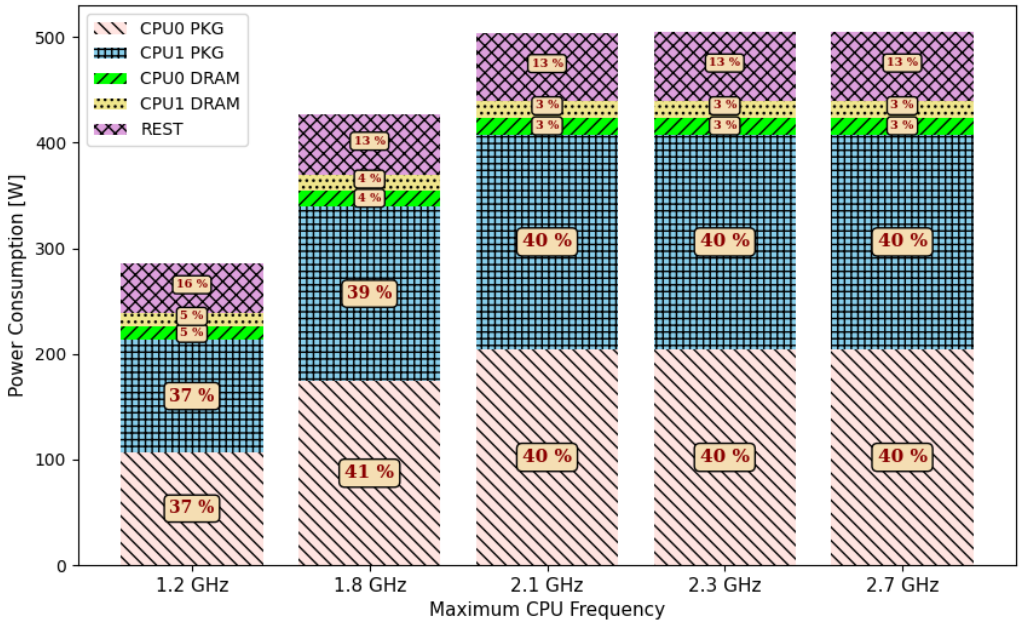


Figure 23. Average power consumption distribution bar chart of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing FIRESTARTER benchmark

As expected, the power consumption of the processors increases with the maximum CPU frequency. Due to the nature of the FIRESTARTER benchmark which generates near peak power consumption for a compute node (and keeps a steady workload character without fluctuations) the results recorded for this benchmark exhibit the highest power consumption. As can be seen from Figure 23 the processors of the compute node reached their TDP limits already at maximum CPU frequency of 2.1 GHz.

Figure 24 illustrates the recorded performance and average power consumption when executing a single node STREAM [126] benchmark on SuperMUC-NG system.

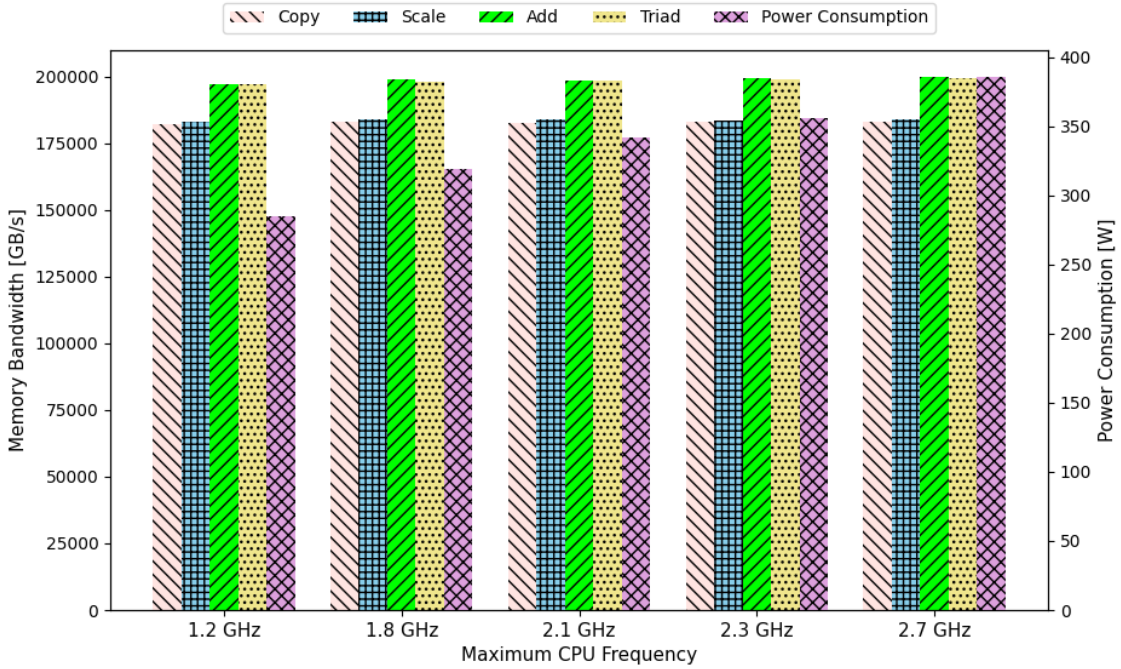


Figure 24. Performance and average power consumption across distinct maximum CPU frequencies when executing STREAM benchmark

The first bar from the left of Figure 24 colored in blue diagonal stripes shows the best rate of copy function performance measured in MB/s across distinct maximum CPU frequencies. Similarly, the second, third, and fourth bars correspondingly illustrate the recorded best performance rates for scale, add, and triad functions. The last, fifth bar colored in red, indicates the recorded average power consumption at different maximum CPU frequencies.

Since STREAM is a memory-bound application, as expected, apart from increased power consumption rates at higher CPU frequencies as illustrated in Figure 24, no major performance gains were recorded. This once again stresses the fact that one should not specify higher CPU frequencies for memory-bound applications.

Figure 25 illustrates the average power consumption distribution among the discussed components of a Super-MUC-NG node when executing the IOR [127] benchmark. The average power consumption rates seen in Figure 25 clearly indicate the I/O boundedness of the benchmark.

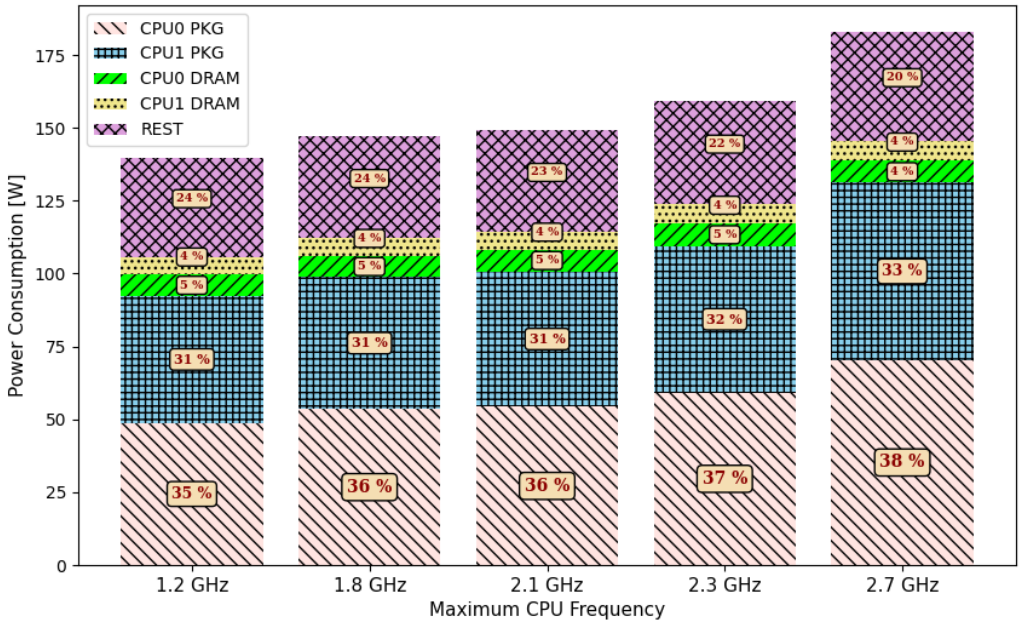


Figure 25. Average power consumption distribution of a SuperMUC-NG compute node across distinct maximum CPU frequencies when executing IOR benchmark

When summarizing all the benchmarks from power consumption perspective, it can be seen, that each of the CPUs, when not idling, is responsible for 30%-41% of the overall power consumption within a compute node even for memory or I/O-bound applications. The power consumption per DIMM (dual in-line memory module) was recorded between 3%-7%.

Given the rising energy costs, the shown benchmarks once again indicate that users should only request high CPU frequencies only for the use cases where application would have a significant benefit from it, e.g. when the application is a CPU-bound application. Doing otherwise, would only introduce additional energy costs and won't result in any significant performance improvements.

3.4. Performance Analysis

This chapter explains the performance analysis of CPUs (scientific codes performance analysis in CPU). Here we show how to use some of the standard performance analysis tools such as Intel Application Performance Snapshot, Arm Forge, Scalasca, and PAPI for the performance analysis of the scientific codes.

3.4.1. Intel Application Performance Snapshot

The shared memory programming model (OpenMP) and distributed programming model (MPI) can be profiled and analysed quickly by using the Intel Application Performance Snapshot (APS) as an initial stage of profiling scientific codes. It is easy to use because it has low overhead and high scalability. APS is part of the Intel Parallel Studio [28], which provides the necessary tools (for example, Intel compiler, math libraries, performance analysis tools, etc.) to maximise the code performance on Intel CPUs. Figure 26 shows the workflow for the profiling and performance analysis within the Intel Parallel Studio. During the analysis, APS considers the CPU, Floating Point Unit (FPU), and Memory.

The end results show how much time the code spends on MPI, MPI and OpenMP imbalance, I/O and memory footprint, Memory access efficiency, FPU usage, and Vectorisation. A performance analysis will be shown as a snapshot, including the key performance analysis and critical areas where further optimisation is needed. The output result can be converted to the *HTML* format and can be opened in the web browser to view the results. Many metrics are available for the APS and can be used for different scenarios, which are listed in [14]. And *aps-report –help* command line reference shows the available resource for the APS.

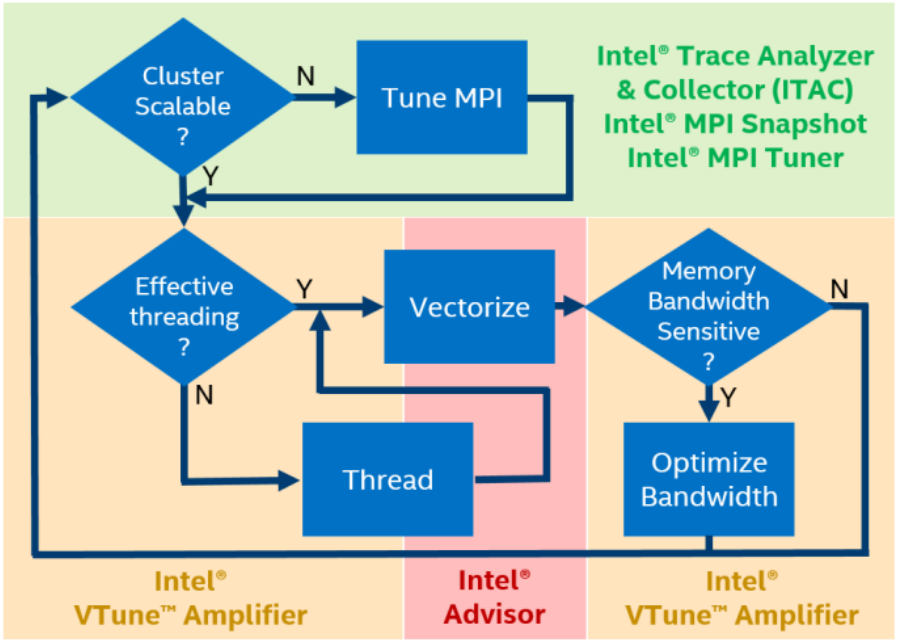


Figure 26. Intel profiling and performance analysis within the Intel Parallel Studio

3.4.1.1. OpenMP

Here we show how one can very easily and quickly analyze the performance of the OpenMP programming model. Figure 27 shows the generic workflow model of the APS for OpenMP [29], where profiling is initially done; later, it can be viewed either in the command line or by using the HTML format in the browser. Furthermore, it can be analyzed and tuned for the better performance.

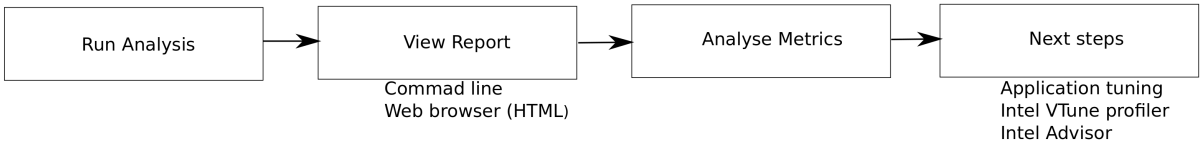


Figure 27. Intel schematic performance analysis workflow for the shared memory programming model

To show an example, we have considered a simple matrix-matrix multiplication. The below example shows how to compile, profile, and collect the data for the non-MPI programming model.

```
# compilation
$ icc -qopenmp mat_mat_multiplication.c

# code execution
$ aps --collection-mode=all -r report_output ./a.out
$ aps-report -g report_output                  # create a .html file
$ firefox report_output_<postfix>.html         # APS GUI in a browser
$ aps-report report_output                     # command line output
```

Once we have collected the data, we can visualise the performance analysis values either in the web browser or in the command line. Figure 28 shows the OpenMP APS in HTML format in the web browser. Here we can see the important performance factors and the metrics (CPU utilisation, memory access efficiency, and vectorisation). For example, to solve the OpenMP imbalance, dynamic scheduling can be considered. We can also see the GFLOPS (single precision and double precision) performance of the code (how fast the code is executed in the given architecture). More importantly, Intel APS GFLOPS is available only for the 3rd, fourth and 5th generation of Intel cores. Similarly, we can also see the attained CPU frequency of the given code, which will also give an overview

of the given code. To see the full list of APS metrics, please refer to [30]. And also, to know further information about the APS, please type *aps -help*; this will list out profiling metrics options in APS.

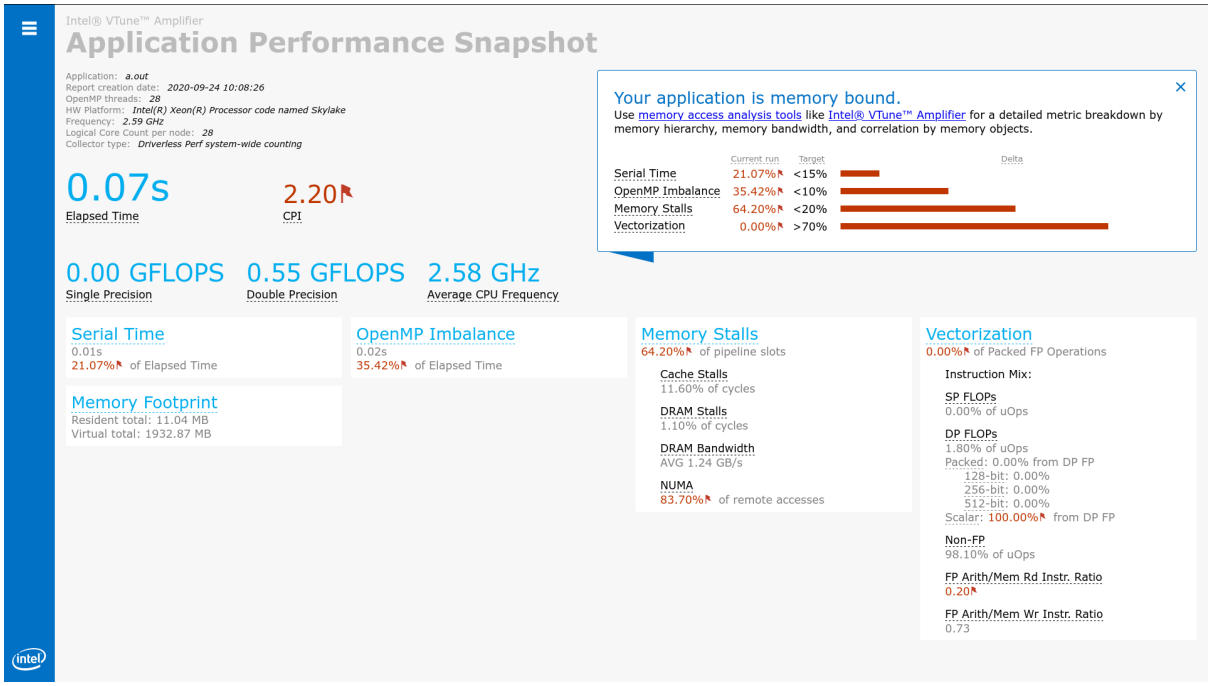


Figure 28. OpenMP APS (.html format from the browser)

On the other hand, as we have mentioned before, we can also see the command line's APS performance results. The below table shows the command line output summary for the OpenMP programming model.

```
# command line output
$ aps --report report_output

| Summary information
|---------------------------------------------------------------------
Application                 : a.out
Report creation date        : 2020-09-24 10:08:26
OpenMP threads number       : 28
HW Platform                 : Intel(R) Xeon(R) Processor code named
Skylake
Frequency                   : 2.59 GHz
Logical core count per node: 28
Collector type              : Driverless Perf system-wide counting
Used statistics             : /path/to/folder/report_all_output
|
| Your application is memory bound.
| Use memory access analysis tools like Intel(R) VTune(TM) Amplifier
| for a detailed metric breakdown by memory hierarchy,
| memory bandwidth, and correlation by memory objects.
|
Elapsed time:                 0.07 sec
SP GFLOPS:                    0.00
DP GFLOPS:                    0.55
Average CPU Frequency:        2.58 GHz
CPI Rate:                     2.20
| The CPI value may be too high.
| This could be caused by such issues as memory stalls,
| instruction starvation, branch misprediction,
```

```
| or long latency instructions.
| Use Intel(R) VTune(TM) Amplifier General Exploration analysis
| to specify particular reasons of high CPI.
Serial Time:                  0.01 sec            21.07%
| The Serial Time of your application is significant.
| It directly impacts application Elapsed Time and scalability.
| Explore options for parallelization with Intel(R) Advisor
| or algorithm or microarchitecture tuning of the serial
| code with Intel(R) VTune(TM) Amplifier.
OpenMP Imbalance:             0.02 sec            35.42%
| The metric value can indicate significant time spent by
| threads waiting at barriers. Consider using dynamic work
| scheduling to reduce the imbalance where possible.
| Use Intel(R) VTune(TM) Amplifier HPC Performance
| Characterization analysis to review imbalance data
| distributed by barriers of different lexical regions.
Memory Stalls:              64.20% of pipeline slots
| The metric value can indicate that a significant
| fraction of execution pipeline slots could be stalled
| due to demand memory load and stores. See the second level
| metrics to define if the application is cache- or
| DRAM-bound and the NUMA efficiency. Use Intel(R) VTune(TM)
| Amplifier Memory Access analysis to review a detailed
| metric breakdown by memory hierarchy, memory bandwidth
| information, and correlation by memory objects.
Cache Stalls:                          11.60% of cycles
DRAM Stalls:                            1.10% of cycles
Average DRAM Bandwidth:                 1.24  GB/s
NUMA: % of Remote Accesses:            83.70%
| A significant amount of DRAM loads was serviced from remote DRAM.
| Wherever possible, consistently use data on the same core,
| or at least the same package, as it was allocated on.
Vectorization:                         0.00%
| A significant fraction of floating point arithmetic
| instructions are scalar. Use Intel Advisor to see
| possible reasons why the code was not vectorized.
Instruction Mix:
SP FLOPs:                              0.00% of uOps
DP FLOPs:                              1.80% of uOps
Packed:                                0.00% from DP FP
128-bit:                               0.00%
256-bit:                               0.00%
512-bit:                               0.00%
Scalar:                              100.00% from DP FP
| A significant fraction of floating point arithmetic
| instructions are scalar. Use Intel(R) Advisor to see possible
| reasons why the code was not vectorized.
Non-FP:                                98.10% of uOps
FP Arith/Mem Rd Instr. Ratio:           0.20
| The metric value might indicate unaligned access to
| data for vector operations. Use Intel(R) Advisor to
| find possible data access inefficiencies for vector operations.
FP Arith/Mem Wr Instr. Ratio:           0.73
Memory Footprint:
Resident:                              11.04 MB
Virtual:                             1932.87 MB
```

3.4.1.2. MPI (and also for the MPI+OpenMP)

APS also provides performance analysis for the MPI (and even for the hybrid programming model (MPI+OpenMP)) programming model. Figure 30 shows the generic workflow model in Intel Parallel Studio for the MPI application, where MPI results were analyzed. Later, it can be tuned using Intel or other open source platforms [44]. The APS provides a quick overview of metrics, e.g., MPI time, time taken of MPI functions, memory stalls, and I/O bound. This can be clearly seen in figure 29, which shows the profiling of the MPI programming model using the APS.

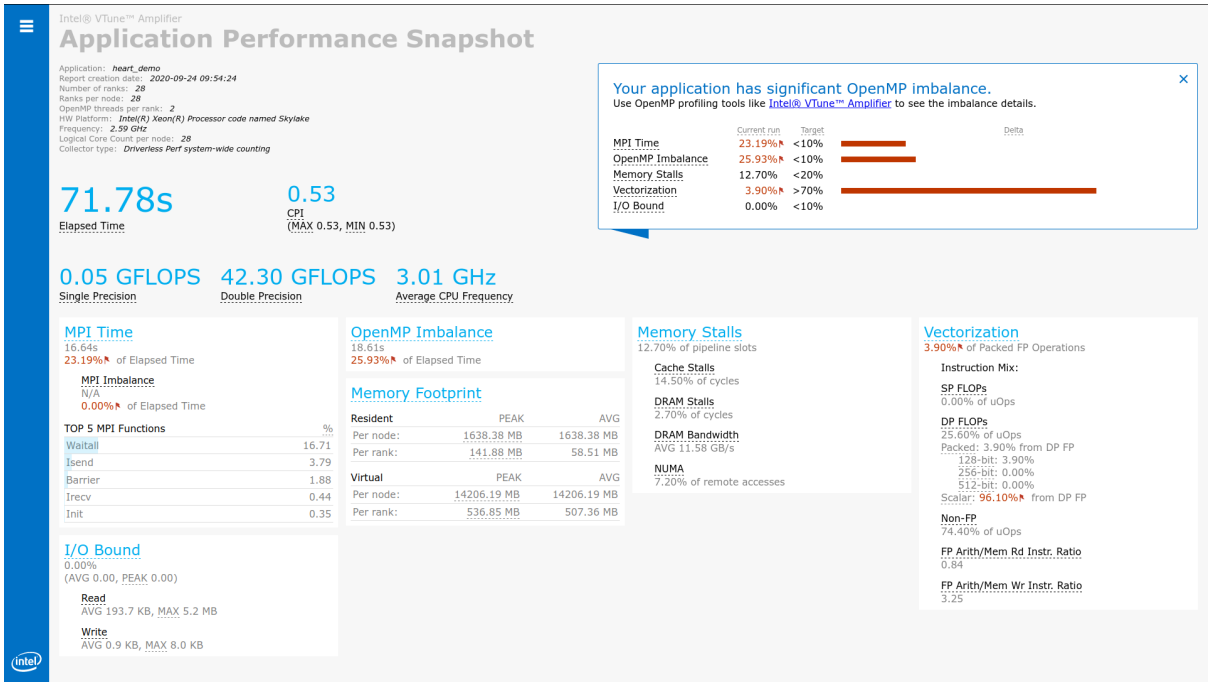


Figure 29. MPI only APS

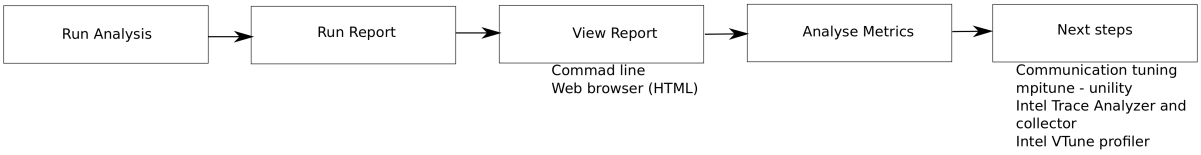


Figure 30. Intel schematic performance analysis workflow for distributed (MPI) programming model

After collecting the data (performance analysis), we can also quickly filter the results (metrics that we want to see) that we are interested in, for example, function summary for all ranks (*aps-report ./report_all_output/ -f*), MPI time per rank (*aps-report ./report_all_output/ -t*), etc,. More information about detailed MPI analysis (analysis charts) please refer to Intel APS MPI charts [118]. And also to focus on the MPI imbalance in the code, we need to set export *APS_IMBALANCE_TYPE=2*, this will help us to figure out the MPI imbalance in the code [31]. The example of compiling and executing the MPI application commands are shown below.

```
# compilation
$ mpiicc mat_mat_multiplication.c

# code execution
$ mpirun -np 32 aps --collection-mode=mpi -r result_output ./a.out
$ aps-report -g report_output            # create a .html file
$ firefox report_output.html             # APS GUI in a browser
$ aps-report report_output               # command line output
```

The above results can also be viewed from the command line, for example, below command shows the result.

```
# command line output
$ aps-report -a report_output
| Summary information
|-----------------------------------------------------------------
Application                   : heart_demo
Report creation date          : 2020-09-24 09:54:24
Number of ranks               : 28
Ranks per node                : 28
OpenMP threads number per rank: 2
HW Platform                   : Intel(R) Xeon(R) Processor code
named Skylake
Frequency                     : 2.59 GHz
Logical core count per node   : 28
Collector type                : Driverless Perf system-wide counting
Used statistics               : ./report_all_output/
|
| Your application has significant OpenMP imbalance.
| Use OpenMP profiling tools like Intel(R) VTune(TM) Amplifier
| to see the imbalance details.
|
Elapsed time:               71.78 sec
SP GFLOPS:                   0.05
DP GFLOPS:                  42.30
Average CPU Frequency:       3.01 GHz
CPI Rate:                    0.53
MPI Time:                   16.64 sec            23.19%
| Your application is MPI bound. This may be caused by high
| busy wait time inside the library (imbalance), non-optimal
| communication schema or MPI library settings. Explore the
| MPI Imbalance metric if it is available or use MPI profiling
| tools like Intel(R) Trace Analyzer and Collector to explore
| possible performance bottlenecks.
MPI Imbalance:  N/A*
| * No information about MPI Imbalance time is available.
| Set the MPS_STAT_LEVEL
| value to 2 or greater or set APS_IMBALANCE_TYPE to
| 1 or 2 to collect it.
Top 5 MPI functions (avg time):
Waitall                     11.99 sec  (16.71 %)
Isend                        2.72 sec  ( 3.79 %)
Barrier                      1.35 sec  ( 1.88 %)
Irecv                        0.32 sec  ( 0.44 %)
Init                         0.25 sec  ( 0.35 %)
OpenMP Imbalance:           18.61 sec            25.93%
| The metric value can indicate significant time spent by
| threads waiting at barriers. Consider using dynamic work
| scheduling to reduce the imbalance where possible.
| Use Intel(R) VTune(TM) Amplifier HPC Performance
| Characterization analysis to review imbalance data
| distributed by barriers of different lexical regions.
Memory Stalls:                       12.70% of pipeline
slots
Cache Stalls:                        14.50% of cycles
```

```
DRAM Stalls:                           2.70% of cycles
Average DRAM Bandwidth:               11.58  GB/s
NUMA: % of Remote Accesses:            7.20%
Vectorization:                         3.90%
| A significant fraction of floating point arithmetic
| instructions are scalar. Use Intel Advisor to see
| possible reasons why the code was not vectorized.
Instruction Mix:
SP FLOPs:                              0.00% of uOps
DP FLOPs:                             25.60% of uOps
Packed:                                3.90% from DP FP
128-bit:                               3.90%
256-bit:                               0.00%
512-bit:                               0.00%
Scalar:                               96.10% from DP FP
| A significant fraction of floating point arithmetic
| instructions are scalar. Use Intel(R) Advisor to
| see possible reasons why the code was not vectorized.
Non-FP:                               74.40% of uOps
FP Arith/Mem Rd Instr. Ratio:          0.84
FP Arith/Mem Wr Instr. Ratio:          3.25
Disk I/O Bound:                        0.00 sec ( 0.00 %)
Data read:                             5.3  MB
Data written:                         26.0  KB
Memory Footprint:
Resident:
Per node:
Peak resident set size    : 1638.38 MB (node iris-163)
Average resident set size : 1638.38 MB
Per rank:
Peak resident set size    : 141.88 MB (rank 0)
Average resident set size : 58.51 MB
Virtual:
Per node:
Peak memory consumption    : 14206.19 MB (node iris-163)
Average memory consumption : 14206.19 MB
Per rank:
Peak memory consumption    : 536.85 MB (rank 0)
Average memory consumption : 507.36 MB
```

3.4.1.3. Filtering Capabilities

Sometimes it can be that we need to profile the code that can scale up to 1000 CPU cores, but on the other hand, results may contain 1000s of line, which might be hard to understand. To overcome this, one can use the Intel APS command line filtering capabilities, which will show the requested results in the command line or GUI. There are three modules available in APS, which are as follows:

- filtering by Key Metric[25].

```
# to show all lines in the result summary
$ aps-report -V 0 -N 0 -m aps_result
# output:
| Message Sizes summary for all ranks
|-------------------------------------------------------------
| Message size(B)  Volume(MB)   Volume(%)          Transfers
|-------------------------------------------------------------
```

```
  262144             8340          11.6789           33360
  524288             8340          11.6789           16680
  1048576            8340          11.6789           8340
  2097152            8280          11.5948           4140
  4194304            8160          11.4268           2040
  131072             8130          11.3848           65040
  65536              7650          10.7126           122400
  32768              5805           8.129            185760
  16384              2966.25        4.15377          189840
  8192               1500           2.10052          192000
  8388608            1440           2.0165           180
  16777216           960            1.34433          60
  4096               750            1.05026          192000
  2048               375            0.525129         192000
  1024               187.5          0.262565         192000
  512                93.75          0.131282         192000
  256                46.875         0.0656411        192000
  128                23.4375        0.0328206        192000
  64                 11.7188        0.0164103        192000
  32                  5.85938       0.00820514       192000
  16                  2.92975       0.00410266       192004
  8                   1.49738       0.00209684       196264
  4                   0.749222      0.00104917       196404
  2                   0.308998      0.000432704      162004
  1                   0.143055      0.000200326      150004
  60                  0.000114441   1.60257e-007     2
  12                  2.28882e-005  3.20513e-008     2
  0                   0             0                214062
|============================================================
| TOTAL             71411         100              3466586
```

```
# to skip all the lines which are having less than 10% volume size:
$ aps-report –V 10 -N 0 -m aps_result

# output:
| Message Sizes summary for all ranks
|------------------------------------------------------------
| Message size(B)    Volume(MB)    Volume(%)        Transfers
|------------------------------------------------------------
  262144             8340          11.6789           33360
  524288             8340          11.6789           16680
  1048576            8340          11.6789           8340
  2097152            8280          11.5948           4140
  4194304            8160          11.4268           2040
  131072             8130          11.3848           65040
  65536              7650          10.7126           122400
| [skipped 21 lines]
|============================================================
| TOTAL              71411         100               3466586
```

- Filtering by Number of Lines[26].

```
# to display over all just two lines in top, middle and bottom
$ aps-report -V 0 –N 2 -m aps_result
```

```
# output:
| Message Sizes summary for all ranks
|-------------------------------------------------------------
| Message size(B)    Volume(MB)    Volume(%)      Transfers
|-------------------------------------------------------------
  262144             8340          11.6789            33360
  524288             8340          11.6789            16680
| [skipped 11 lines]
  2048               375            0.525129          192000
  1024               187.5          0.262565          192000
| [skipped 11 lines]
  12                 2.28882e-005   3.20513e-008      2
  0                  0              0                 214062
|=============================================================
| TOTAL              71411          100               3466586
```

- Using Filter Combinations[27].

```
# to combine volume of data (hide less than 1%)
# and number of lines (just 2 lines)
$ aps-report -V 1-N 2 -m aps_result

# output:
| Message Sizes summary for all ranks
|-------------------------------------------------------------
| Message size(B)    Volume(MB)    Volume(%)      Transfers
|-------------------------------------------------------------
  262144             8340          11.6789            33360
  524288             8340          11.6789            16680
| [skipped 3 lines]
  131072             8130          11.3848            65040
  65536              7650          10.7126           122400
| [skipped 4 lines]
  16777216           960            1.34433             60
  4096               750            1.05026          192000
| [skipped 15 lines]
|=============================================================
| TOTAL              71411          100              3466586
```

Finally, APS provides the region control profiling for the MPI programming model (and also for the hybrid programming MPI+OpenMP). Where the interested region can only be included by using the Intel APS MPI_Pcontrol(). For example, if you want to mark region start with MPI_Pcontrol(<regions >) and to end with MPI_Pcontrol(-<region >). In some cases, we need to ignore the profiling on a particular region; in that case, we should do, start a region with MPI_Pcontrol(0) and end with MPI_Pcontrol(1). Codes that are in 0-1 will be ignored for the profiling tracing and metrics.

```
# to ignore and consider the region during the profiling
MPI_Pcontrol(0);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Pcontrol(1);

# profiling particular region with if and else condition
if(my_id == root_process)
{
```

```
MPI_Pcontrol(5);
/* do some computation*/
MPI_Pcontrol(-5);
}
else
{
MPI_Pcontrol(6);
/* do some computation*/
MPI_Pcontrol(-6);
}
```

3.4.2. Scalasca

Scalasca [37] is a performance analysis tool that supports large-scale systems, including IBM Blue Gene and Cray XT and small systems. The Scalasca provides information about the communication and synchronisation among the processors. This information will help to do the performance analysis, optimisation, and tunning of scientific codes. Scalasca supports OpenMP, MPI, and hybrid programming model, and analysis can be done by using the GUI, which can be seen in Figure 31.

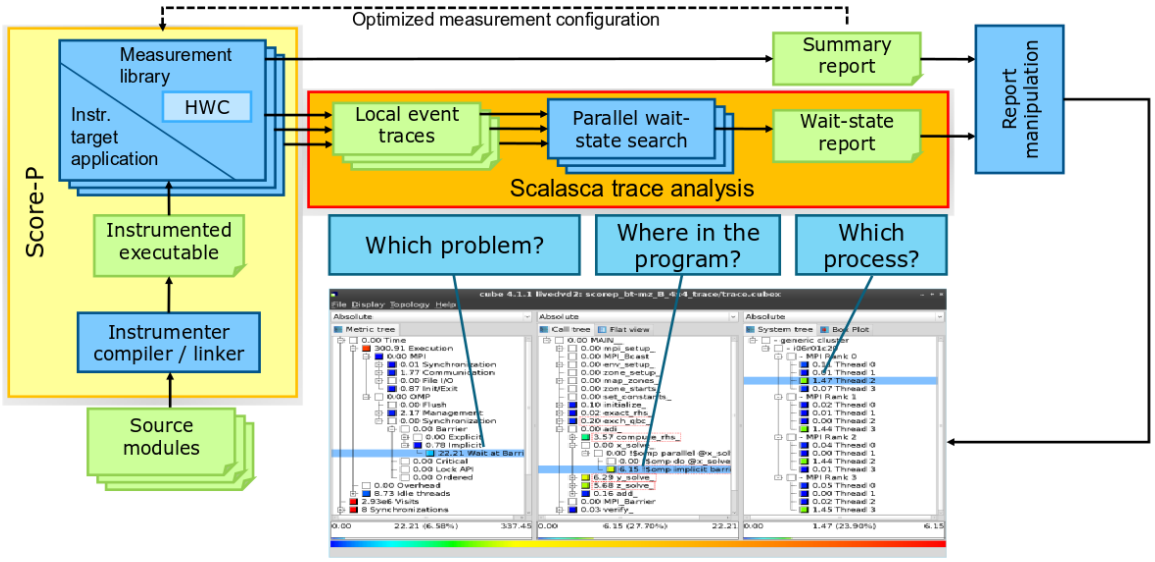


Figure 31. Scalasca workflow

Performance analysis in Scalasca has three stages; they are, instrument (skin), analyze (scan), and examine(square)[45]. The following steps provide the detail information about these procedures.

- Instrument (or skin)

```
# instrument
$ scorep mpicxx example.cc
```

- Analyse (or scan)

```
# analyze
```

```
$ scalasca -analyze  mpirun -np 28 ./a.out
```

- Examine (or square)

```
# examine
$ scalasca -examine -s scorep_a_28_sum
INFO: Post-processing runtime summarization report...
INFO: Score report written to ./scorep_a_28_sum/scorep.score

# initial command line simple overview
$ head -n 25 scorep_a_28_sum/scorep.score
Estimated aggregate size of event trace:                   7kB
Estimated requirements for largest trace buffer (max_buf): 495 bytes
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       4097kB
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid
intermediate flushes or reduce requirements using USR regions filters.)

flt     type max_buf[B] visits time[s] time[%] time/visit[us]  region
ALL        494    274    2.70   100.0     9863.88  ALL
USR        364    194    0.00     0.1        9.85  USR
MPI        104     64    2.70    99.9    42192.33  MPI
COM         26     16    0.00     0.0       30.25  COM

USR        286    176    0.00     0.0        0.22  double
initial_condition
(double, double)
USR         52      2    0.00     0.0      140.83  void timestamp()
MPI         26     16    0.00     0.0        0.40  MPI_Comm_size
MPI         26     16    0.00     0.0        3.08  MPI_Comm_rank
MPI         26     16    2.68    99.1   167375.28  MPI_Init
COM         26     16    0.00     0.0       30.25  int main(int,char**)
MPI         26     16    0.02     0.8     1390.56  MPI_Finalize
USR         26     16    0.00     0.1       99.33  void update(int, int)


# analyse with filtering
$ scalasca -analyze -q -t -f npb-bt.filt mprirun -np 16 ./a.out

# graphical visualisation
$ scalasca -examine result_folder

# allocate the memory for the score;
and this can be adjusted according to memory availability
$ export SCOREP_TOTAL_MEMORY=32M
$ scalasca -analyze -q -t -f npb-bt.filt mpirun -np 28 ./a.out

# for more insight in the application
$ export SCAN_ANALYZE_OPTS="--time-correct"
$ scalasca -analyze -a mpirun -np 28 ./a.out
```

Since it can apply to a large system with a massive parallel option, sometimes, it is necessary to avoid wrong performance data due to excessive measurement overhead. For this, we can also choose the selective traces by selecting, for example, using runtime filtering, selective recording, or manual instrumentation controlling measurement. The initial report can be seen in the command line with a given number of lines in the file. Topology and placement optimisation is recommended for numerical computations such as, for example, stencils, collective

operations on subsets of ranks, and neighborhood operations; the Scalasca can easily analyze these. Furthermore, *scorep –help* and *scalasca –help* provides the useful option that can be useful during the performance analysis.

3.4.3. Arm Forge Reports

Arm Forge [40] is another commercial standard tool for debugging [39], profiling [41], and analyzing [42] scientific code on the massively parallel computer architecture. They have a separate toolset for each category with the common environment, namely DDT for debugging, MAP for profiling, and performance reports for analysis. It also supports the MPI, UPC, CUDA, and OpenMP programming models for a different architecture with different variety of compilers. Both DDT and MAP will launch the GUI, where we can debug and profile the code interactively. Whereas *perf-report* will provide the analysis results in *.html and .txt* file. The following command shows how to work on Arm Forge, and figure 32 shows the performance analysis of the hybrid programming (OpenMP+MPI).

```
# for debugging
$ ddt mpirun -np 16 ./a .out

# for profiling
$ map mpirun -np 16 ./a .out

# for analysis
$ perf-report mpirun -np 16 ./a .out
```

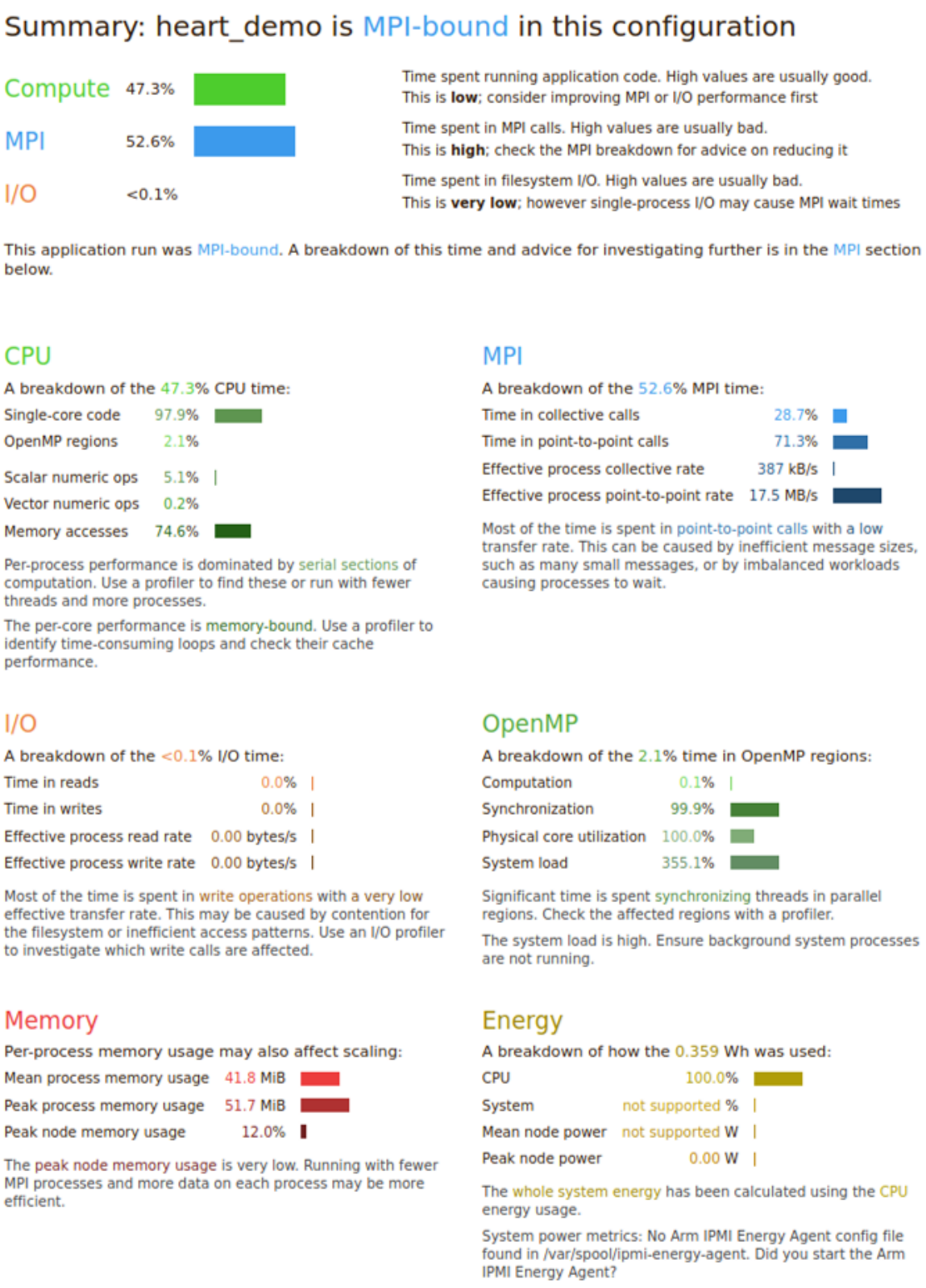


Figure 32. Arm Forge Performance Report

3.4.4. PAPI

The Performance Application Programming Interface [43] is an opensouce performance analysis tool, which has a routines (low level and high level) that will provide the performance counters on the modern processors. Figure 33

shows the PAPI architecture, and *papi_avail* will list out the current available events. The simple example below shows the usage of PAPI.

```
#include <stdio.h>
#include <stdlib.h>
#include <papi.h>

void InitBlock(float *a, float *b, float *c, int blk)
{
int len, ind;
int i,j;

len = blk*blk;
for (ind = 0; ind < len; ind++)
{ a[ind] = (float)(rand()%1000)/100.0; c[ind] = 0.0; }
for (i = 0; i < blk; i++) {
for (j = 0; j < blk; j++) {
b[j*blk+i] = (i==j)? 1.0 : 0.0;
}}

void BlockMult(float* c, float* a, float* b, int blk)
{
int i,j,k;

for (i = 0; i < blk; i++)
for (j = 0; j < blk; j ++)
for (k = 0; k < blk; k++)
c[i*blk+j] += (a[i*blk+k] * b[k*blk+j]);
}

int main(int argc, char* argv[])
{

int numEvents = 4;
long long values[4];
int events[4] = {PAPI_L2_TCA,PAPI_L2_TCM,PAPI_L3_TCA,PAPI_L3_TCM};

if (PAPI_start_counters(events, numEvents) != PAPI_OK)
exit(EXIT_FAILURE);

float *a, *b, *c;

/* Compute the array dim from the same parameters. */
int sidesize;
if (argc == 3)
sidesize = atoi(argv[1]) * atoi(argv[2]);

else {
fprintf(stderr, "usage: mmult m blk\n");
exit(1);
}

/* allocate the memory for the arrays. */
a = (float*)malloc(sizeof(float)*sidesize*sidesize);
b = (float*)malloc(sizeof(float)*sidesize*sidesize);
c = (float*)malloc(sizeof(float)*sidesize*sidesize);
```

```
/* check for valid pointers */
if (!(a && b && c)) {
fprintf(stderr, "%s: out of memory!\n", argv[0]);
free(a); free(b); free(c);
exit(2);
}

/* initialize the arrays */
InitBlock(a, b, c, sidesize);

/* Multiply. */
BlockMult(c, a, b, sidesize);

/* check it */
int i;
for (i = 0 ; i < sidesize*sidesize; i++)
if (a[i] != c[i])
printf("Error a[%d] (%g) != c[%d] (%g) \n", i, a[i],i,c[i]);

printf("Done.\n");
free(a); free(b); free(c);

if ( PAPI_stop_counters(values, numEvents) != PAPI_OK)
exit(EXIT_FAILURE);

printf("Level 2 total cache accesses:%d\n",values[0]);
printf("Level 2 cache misses:%d\n",values[1]);
printf("L2 miss/access ratio:%d\n",(double)values[1]/values[0]);

printf("Level 2 total cache accesses:%d\n",values[2]);
printf("Level 3 cache misses:%d\n",values[3]);
printf("L3 miss/access ratio:%d\n",(double)values[3]/values[2]);

return 0;
}

# compilation
$ gcc test.c -lpapi

# output result
$ ./a.out 15 15
Level 2 total cache accesses:314918
Level 2 cache misses:281496
L2 miss/access ratio:2147483620
Level 2 total cache accesses:281496
Level 3 cache misses:3
L3 miss/access ratio:2147483625
```

From the above example, we can get an idea about L2 and L3 cache misses. There are many other counters available (PAPI Present Events), which can be used for other specific purposes (for example, floating point operations and instruction cache hits).

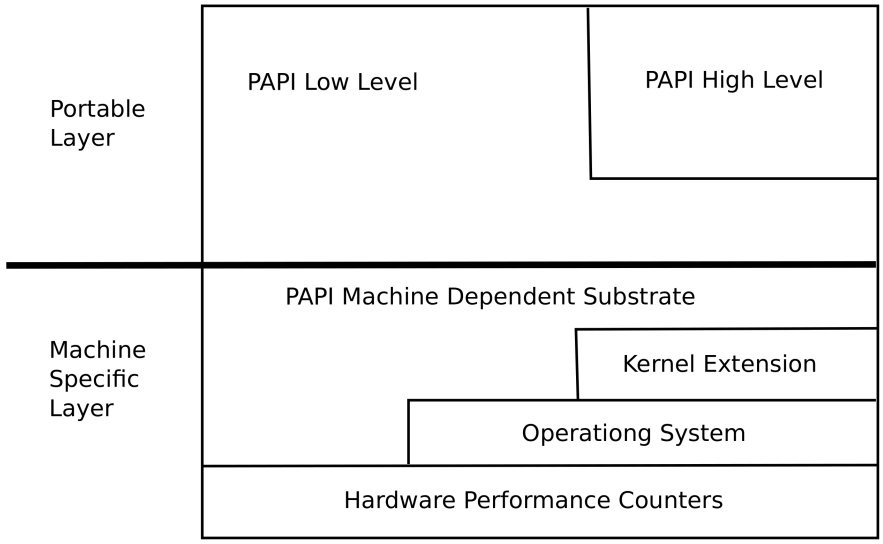


Figure 33. PAPI architecture

```
# shows the available PAPI Present Events
$ papi_avail

Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------
PAPI version             : 5.6.0.0
Operating system         : Linux 3.10.0-957.1.3.el7.x86_64
.
==========================================================================
PAPI Preset Events
==========================================================================
Name        Code    Avail Deriv Description (Note)
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   Yes  Level 2 data cache misses
.
-----------------------------------------------------------------
Of 108 possible events, 59 are available, of which 18 are derived.
```

3.5. Tuning

This section explains performance tuning for the Intel CPU. Performance optimisation for the Intel processor can be done in different ways. Moreover, Intel is providing the Intel Parallel Studio XE, which has many tools that can help to optimise the code. This chapter covers tuning suggestions for shared memory programming, distributed memory programming, and hybrid programming model.

3.5.1. Compiler Flags

Table 15 shows some common optimisation flag that will enable optimisation and vectorisation in the code. To see the performance results, one should see the output results; the flags for this are defined in Table 16. Table 17 shows enabling the OpenMP, choosing the library, and enabling static and dynamic scheduling. Table 18 shows the important floating point compiler flags. Finally, Table 19 shows how to generate the code for the Intel Skylake processor.

Table 15. Optimisation

Flags	Comment
-O0	No optimisation
-O1	Small optimisation in performance, many be useful for large/ database applications where memory paging due to large code size is an issue.
-O2	It is a default setting with maximum speed. It enables vectorisation and intra-file interprocedural optimisations.
-O3	Recommended for the loops that have many floating point operations, sometime a performance could be lower compare to -O2.
-Ofast	enable -O3 -no-prec-div -fp-model fast=2 optimisations

Table 16. Optimisation Reports

Flags	Comment
-qopt-report[=n]	Generates an optimisation reports, n ranges from 0-5. Default is 2, and 0 disable the optimisation report output.
-qopt-report-file=[stdout - stderr - <file>]	Specify file name for the report output.
-qopt-report-phase=<phase>[,<phase>,...]	Define the phase (profiling) for the report output.

Table 17. OpenMP and Parallel Processing

Flags	Comment
-parallel	automatically generates multi-threaded code from serial code, by parallelizing the for loop.
-qopenmp	Generates the multithreaded code based on the OpenMP directives. To disable use -qno-openmp.
-qopenmp-stubs	OpenMP program becomes serial code and ignores OpenMP directives in the parallel code.
-qopenmp-lib=<ver>	Link with OpenMP library.
-par-schedule-dynamic[=n]	Iteration loop size is divided by n, otherwise 1
-par-schedule-static[=n]	Iteration loop size is divided by n, otherwise equally divided

Table 18. Floating Point

Flags	Comment
-fp-model <name>	precise - allows value-safe optimisations consistent - reproducible results for the different optimisation level. Many options available, please refer to -help.
-fp-speculation=<mode>	fast - speculate floating point operations (DEFAULT) safe - speculate only when safe strict - same as off.
-[no-]prec-sqrt	certain square root optimisation is enabled if needed.

Table 19. Code Generation: Skylake

Flags	Comment
-xSKYLAKE-AVX51	Generates the code to run on the Intel Skylake processor.
-mtune=skylake-avx512	Code is optimised by compiler’s default.
-xHost	Generates optimised code on the given processor.

Flags	Comment
-march=skylake-avx512	Generates the code exclusively for the given CPU.

3.5.2. Serial Code Optimisation

By using the Intel compiler flag (-parallel) a serial code with a for-loop can be automatically parallelised. By default it uses the available threads from the multiprocessor system. The below pseudocode example shows the possibilities of the parallelisation in the for-loop.

```
# serial code example
void serial(int *a, int *b, int *c)
{
for (int i = 0; i < 100; i++)
a[i] = a[i] + b[i] * c[i];
}

# parallel code example
void parallel(int *a, int *b, int *c) {
int i;
// Thread 1
for (i=0; i < 50; i++)
  a[i] = a[i] + b[i] * c[i];
  // Thread 2
  for (i=50; i < 100; i++)
    a[i] = a[i] + b[i] * c[i];
}
```

From the above example, as we can see that, there is no data dependencies in the for-loop, so in that case the for-loop can be easily parallelised. By compiling with *-parallel* option the given code can be easily parallelised, the below example shows the auto parallelisation is achieved in the code.

```
# compilation
$ icc -c -parallel example.cc

# compilation to read the report
$ icc -c -parallel -qopt-report-phase=par -qopt-report:2 example.cc
$ icc: remark #10397: optimisation reports are
generated in *.optrpt files in the output location

# sample output
LOOP BEGIN at example_2.cc(37,3)

remark #17109: LOOP WAS AUTO-PARALLELIZED
LOOP END
```

Another way of vectorisation can also be easily achieved by using the optimisation flag *-O2*, and SIMD instructions will unroll a loop. There is also a way to check how the loop is vectorised. The below code shows the vectorisation example in the serial code.

```
# example code
void ser(int *a, int *b, int *c)
{
for (int i = 0; i < 100; i++)
```

```
a[i] = a[i] + b[i] * c[i];
}

# compilation
$ icc -O2 -qopt-report-phase=vec -qopt-report=5 example.cc

# sample output
LOOP BEGIN at example_2.cc(37,3)
remark #15300: LOOP WAS VECTORIZED
LOOP END

# compilation
$ icc -O2 -qopt-report-phase=vec -qopt-report=5 -no-vec example.cc

# sample output
LOOP BEGIN at example_2.cc(37,3)
remark #15540: loop was not vectorized: auto-vectorization
is disabled with -no-vec flag
LOOP END
```

There are some guidelines for vectorisation, which are as follows:

- Write a straight-line code (a single basic block); this means avoid the branches in the code, for example, switch, goto and return.
- And also write a simple for-loop and avoid complex loop termination.
- Avoid the dependencies in the for-loop and also make sure there is no read-after-write dependencies.
- Avoid the function calls in the for-loop (other than math library calls).
- Use the array indexing for the data increment (read and write); this will avoid the output solution's error.

3.5.2.1. Using Structure of Array versus Array of Structures

In memory, an array is stored as a contiguous collection of data, and this data can be organised as an Array of Structure (AoS) and Structure of Array (SoA). For vectorizing the code, SoA is more suitable because it provides the unit-stride memory references. On the other hand, AoS, does not provide the unit-stride, and also the fetched data that are in the cache line might be unused (during the vectorisation). To increase vectorisation performance of vectorised code, especially with a long trip count, one should use the compiler flag *-qopt-dynamic-align*. The below pseudocode shows the difference between AoS and SoA.

```
# Array of Structure   # Structure of Array
struct Point            struct Points
{                       {
  float r;                float* r;
  float g;                float* g;
  float b;                float* b;
}                       }

R|B|G|R|B|G|R|B|G|      R|R|R|B|B|B|G|G|G|
```

Important points need to be considered before vectorisation are as follows:

- Make sure your data is vector friendly.
- Organise your loop row-major-order (for C and C++) for better locality.

- For vectorisation use the SoA.

3.5.2.2. Interprocedural Optimisation

Interprocedural optimisation (IPO) enables the application performance significantly when the program has small and medium functions and also which are used frequently within the loops. The IPO's output profile gives an idea about the analysis of the whole program and suggests possible further optimisation. By default, it uses the O2 optimisation and also can use the O1 optimisation for the inline functions. It can also be used for a single file and also for multiple files. Below, Table 20 shows the different compilation flag options for code optimisation.

Table 20. Interprocedural Optimisation

Flags	Comment
-ip	Single file interprocedural optimisations, including selective inlining, within the current source file.
-ipo[n]	Permits inlining and other interprocedural optimisations among multiple source files. The optional argument n controls the maximum number of link-time compilations (or number of object files) spawned. Default for n is 0 (the compiler chooses).
-ipo-jobs[n]	Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimisation (IPO). The default is 1 job
-finline-functions -finline-level=2	This option enables function inlining within the current source file at the compiler's discretion. This option is enabled by default at -O2 and -O3.
-finline-factor=n	This option scales the total and maximum sizes of functions that can be inlined. The default value of n is 100, i.e., 100% or a scale factor of one.

The benefits IPO are as follows:

- it decreases the overhead, jumps, and calls within the code
- since it optimises the inline function, it reduces the call overhead
- gives better code vectorisation and loop transformations
- efficient cache usage through limited data layout optimisation

3.5.2.3. Intel Inspector

It detects and locates the memory, deadlocks, and data races in the code. The below example shows the simple test case with the memory leak problem.

```
# detect leaks
$ inspxe-cl -collect mi1 -result-dir mi1 -- ./a.out

# result show
$ cat inspxe-cl.txt
=== Start: [2020/04/08 02:11:50] ===

2 new problem(s) found
1 Memory leak problem(s) detected
1 Memory not deallocated problem(s) detected
=== End: [2020/04/08 02:11:55] ===
```

3.5.2.4. Intel Advisor Tool

Intel Advisor tools provide the set of collection tools for the metrics and traces that can be used for further tunning in the code; they also provide GUI visualisation, which will be explained in the later chapter. Intel advisor has five collection methods, which are as follows:

- survey: analyze and explore an idea about where to add efficient vectorisation and threading
- dependencies: identify and explore the dependencies in loop-carried marked loops.
- map: show and explore the complex memory access for the marked loops.
- suitability: check if the annotated program is predicted parallel performance.
- tripcountes: show how many iterations are executed.

The below command lines show the example of running Intel Advisor and check the result.

```
# collecting the metrics and traces
$ advixe-cl -collect survey -project-dir my_result -- ./a.out

# report preparation
$ advixe-cl -report survey -project-dir my_result

# list of analysis types
$ advixe-cl -help collect

# list of available reports
$ advixe-cl -help report
```

3.5.2.5. Intel VTune Amplifier Tool

Intel VTune amplifier provides the set of analysis that helps to tune the code further. They also provide a GUI that can be efficiently used for shared memory programming (OpenMP) and distributed memory programming (MPI). Below examples show the simple command line tools can be quickly used to check code.

```
# traces and metrics collection
$ amplxe-cl -collect hotspots-r my_result ./a.out

# report seeing
$ amplxe-cl -report hotspots -r my_result

# list of analysis types
$ amplxe-cl -help collect

# list of available reports
$ amplxe-cl -help report
```

3.5.3. Shared Memory Programming-OpenMP

A shared memory programming model refers to sharing the global address space among all the processors. And multiple processes can act independently but also can share the same memory resources. And the same time, changes in the memory location by one processor are visible to other processors. Generally, its architecture is classified as Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and Cache Only Memory Access (COMA).

There are two categories of programming models that exist for shared memory architecture. That is a shared memory programming model (without thread), and with thread (POSIX, OpenMP, CUDA, etc.). In this section, we will focus on the performance tuning for the OpenMP, which is an Application Programme Interface (API) for multi-threaded and shared memory parallelism. It has three components, namely, compiler directives, runtime library routines, and environmental variables.

OpenMP threaded application performance is mostly dependent upon the following things:

- single threaded code will underlying performance
- need to check for the CPU utilisation, idle threads, and load balancing
- the percentage of the application is run by multiple threads
- how synchronisation and communication among the threads are handled in the application
- overhead appearance because of creating, manage, destroy, and synchronise the threads. And overhead appears to be worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions
- due to limited shared resources, performance can be limited, such as memory, bus bandwidth, and CPU frequency
- wrong results because of the memory conflicts caused by shared memory or falsely shared memory

Table 21 shows some of the useful optimisation flags. For example, To know more about the valid categories in the Intel environment, one can use *icc -help category* , which will print out the list of the available options and categories.

Table 21. List of useful Flags for the OpenMP

Compiler Flags	**Description**
-mtune=skylake-avx512	Optimises code for processors that support the specified Intel(R) microarchitecture code name.
-ipo	Enable multi-file IP optimisation between files.
-[no-]unroll-aggressive	Enables more aggressive unrolling heuristic.
-[no-]complex-limited-range	Enable/disable (DEFAULT) the use of the basic algebraic expansions of some complex arithmetic operations. This can allow for some performance improvement in programs which use a lot of complex arithmetic at the loss of some exponent range.
-qopenmp	Enable the compiler to generate multi-threaded code based on the OpenMP* directives (same as -fopenmp) Use -qno-openmp to disable. SIMD is enabled by default.
-qopenmp-lib=<ver>	Choose which OpenMP library version to link with compat - use the GNU compatible OpenMP run-time libraries (DEFAULT).

3.5.3.1. Environmental Variables

Intel and GNU provide environmental variables to bind the OpenMP threads to the physical processor. The usage of those might be quite useful depending on the machine topology, application, and operating system as accessing the cache and cores from the same socket will give a better performance. Figure 34 shows how close and spread mapped the binding can be done on a compute node. As can be seen from the figure, close binding will perform better compared to spread. It is mainly because closer threads have better accessibility with cache and main memory, especially when you have different sockets. Typically the GNU has the following environmental variables for pinning, which are set as follows:

- OMP_PLACES=<*sockets*, *cores*, *threads*>
- OMP_PROC_BIND=<*close*, *spread*, *master*>

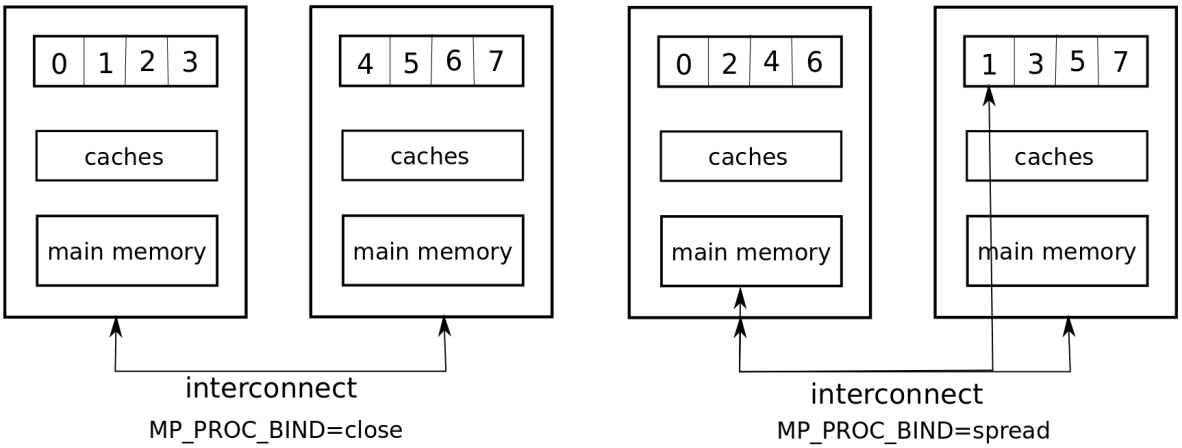


Figure 34. Example illustration of OMP_PROC_BIND=close and OMP_PROC_BIND=spread (numbers represent the threads associated to the physical cores)

And Intel has the following environmental variables, which are as follows:

- KMP_ALL_THREADS - sets the maximum number of threads that can be used by any parallel region
- KMP_AFFINITY - threads can bind on the CPU cores depends on the option, for example, scatter, compact, etc.
- KMP_BLOCKTIME - the threads can wait in the parallel region with the given time, if there is another parallel region afterward, it does not need to be awakened again.
- KMP_LIBRARY - selects the OpenMP runtime library during the execution mode.

3.5.3.2. numactl

numactl is a Linux command that helps to choose the physical memory and CPU cores in the shared memory node. For example, a single compute node can have multiple nodes (sockets where the physical CPU is located), which might have a number of CPU cores. Each node has its own memory and cache, and they are exclusively available for the cores within the nodes. In general, choosing the CPU cores and local memory leads to better performance and less latency. Because accessing the cache and memory of the local CPU is faster than accessing the across node (via interconnect). By using the numactl API, the memory and CPU can be defined, for example, choosing the node is shown below and also *man numactil* shows the list of options available in the numactl API.

```
# to see the hardware information
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26
node 0 size: 64989 MB
node 0 free: 40111 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27
node 1 size: 65536 MB
node 1 free: 38489 MB
node distances:
node   0   1
0:  10  21
1:  21  10

# example (choosing the CPUs within the node 0)
$ numactl --cpunodebind=0 ./a.out

# output
Thread  16 is running on CPU   2
```

```
Thread  15 is running on CPU   0
Thread  12 is running on CPU   4
..
Thread   3 is running on CPU  10
Thread   1 is running on CPU  14
Thread   0 is running on CPU  26
Thread   8 is running on CPU   6
```

As we can see from the above example, we have chosen only the socket 0, this means all the threads will utilise the cores and memory from the socket 0. Final remark would be that, choosing the number of threads might give the different performance on different applications. An ideal option would be map the number of threads into available cores.

3.5.4. Distributed memory programming -MPI

3.5.4.1. Intel Inspector

Intel inspector can be used for the MPI programming also. The example below shows the data race and deadlock methodology in Intel Advisor.

```
# execution for the intel inspector
$ mpirun -np 16 inspxe-cl -collect=ti2 -r result ./a.out

# to see the result output
$ cat inspxe-cl.txt
0 new problem(s) found
=== End: [2020/04/08 16:41:56] ===
=== End: [2020/04/08 16:41:56] ===

0 new problem(s) found
=== End: [2020/04/08 16:41:56] ===
```

3.5.4.2. Intel Advisor Tool

It suggests where to add efficient vectorisation and/or threading, finding how many iterations are executed, identify the loop carries dependencies and complex memory access. For example, Figure 35 shows the instance of vectorisation is needed in some parts of the code. Commands below show how to collect metrics and visualise the results.

```
# code execution
$ mpirun -n 4 advixe-cl --collect survey --project-dir result -- ./a.out

# command line report
$ advixe-cl --report survey --project-dir result

# visualisation
$ advixe-gui result

# list of a possible way to see the report
$ advixe-cl -h report

# list of a possible way to collect traces
$ advixe-cl -h collect
```

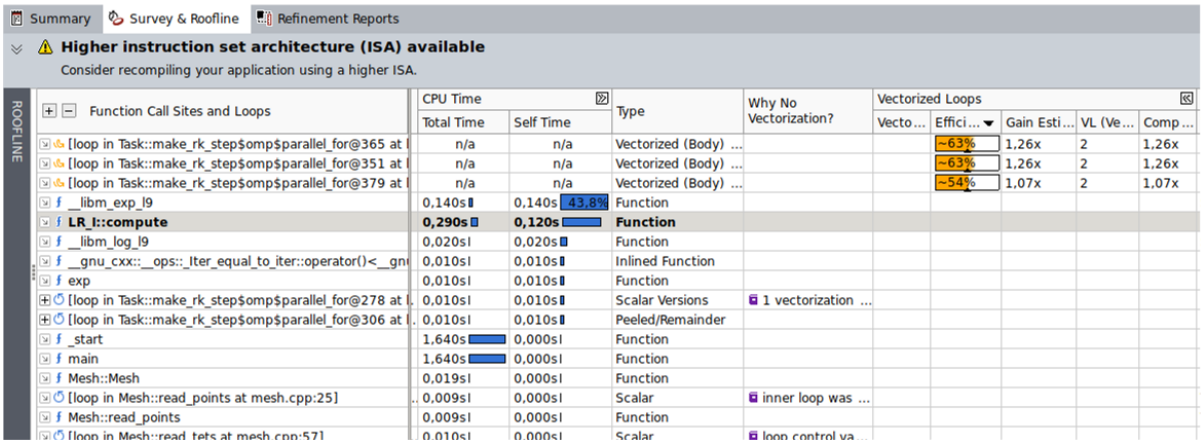


Figure 35. Intel Advisor Vectorisation

3.5.4.3. Intel VTune Amplifier Tool

It shows the information where your application spends most of the time, the efficiency of the multi-thread application, and the power usage of the application/system. To optimise or tune the code, one should first look at the hotspot in the code; for example, this can be seen in Figure 36. The red bars in the figure shows underutilised core time during the execution of the certain functions. However, having a hotspot does not mean that the algorithm behaves badly; sometimes, it is necessary for some routines to be run multiple times. In order to determine the efficiency, one should not look at only the hotspots, but retiring slots, CPI changes, and code should be studied. For more information please refer to [33]. The *-collect uarch-exploration* gives the detailed information about the cache misses, data sharing, and low level hardware information, that will be quite useful to optimise the program further. Figure 37 shows *urach-exploation* from Intel Vtune Amplifier Tool.

```
# running vtune amplifier
$ mpirun -np 4 amplxe-cl -collect uarch-exploration -r vtune_mpi -- ./a.out

# report collection
$ amplxe-cl -report uarch-exploration -report-output output -r vtune_mpi

# vizualisation
$ amplxe-gui vtune_mpi

# shows the list of available reports
$ amplxe-cl -h report

# shows the list of available collection analysis
$ amplxe-cl -h collect
```

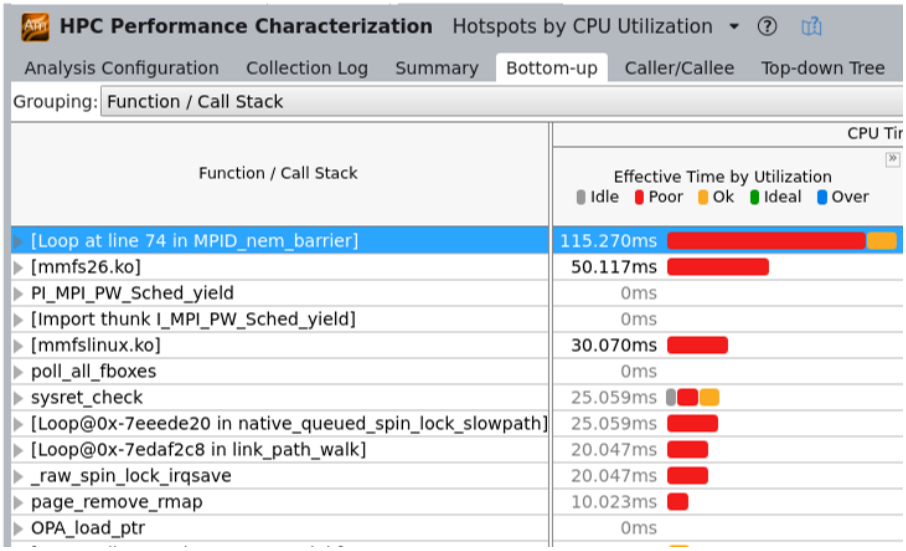


Figure 36. Intel VTune Amplifier (hotspots)

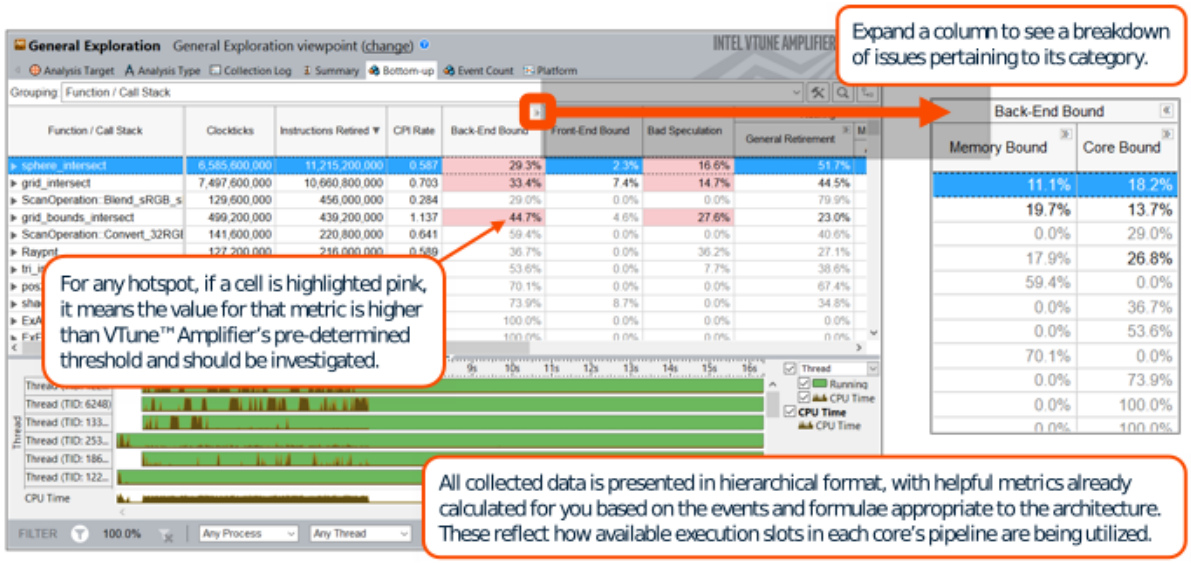


Figure 37. Intel VTune Amplifier (uarch-exploration)

3.5.4.4. Intel Trace Analyser and Collector

Intel Trace Analyser and Collector (ITAC) provides the profiling information about the bottlenecks, improving correctness and achieving high performance on the MPI applications. ITAC comes with the GUI, which included detailed analyses of the MPI application. This will also help to improve the weak and strong scaling for small and large applications. The below command line shows the example for the trace collection in ITAC.

```
# compilation and code execution
$ mpiicc -trace example.cc
$ mpirun -np 16 ./a.out

# to create just one single file result
$ export VT_LOGFILE_FORMAT=STFSINGLE
```

```
# visualisation of the results
$ traceanalyzer a.out*.stf
```

Tracing values can be specific such as, for example, filtering by the collective, P2P, load imbalance operations, and statistics. Statistics collector can be used to understand the functions calls and their communication. This statics collector can be used even if there is no trace collection, that means, it can use as an early stage light analysis. The below list shows the important tracing list and its explanation.

- trace-collectives – to collect information only about collective operations
- trace-pt2pt – to collect information only about point-to-point operations
- trace-imbalance - to collect MPI functions that cause application load imbalance
- print-statistics - it useful for the early stage application when application tracing is unmanageable

The below examples show the trace collections and Figure 38 shows the summary of the ITAC, where we can see the quick information about the time spent on the MPI function and percentage of the programming part (serial, OpenMP and MPI). Figure 39 shows the statistics of the collective operations profile, point-to-point communication, function profile (flat profile, load balance, call tree, and call graph). More importantly, it shows the quantitative timeline and event timeline, which will enable us to get an impression on parallelism and load balance in the program. And Figure 40 shows the detail view of the event timeline.

```
# compilation with trace collection
$ mpiicc -trace example.cc
$ mpirun -np 16 -trace-collective ./a.out

$ export VT_STATISTICS=ON
$ stftool tracefile.stf --print-statistics
```

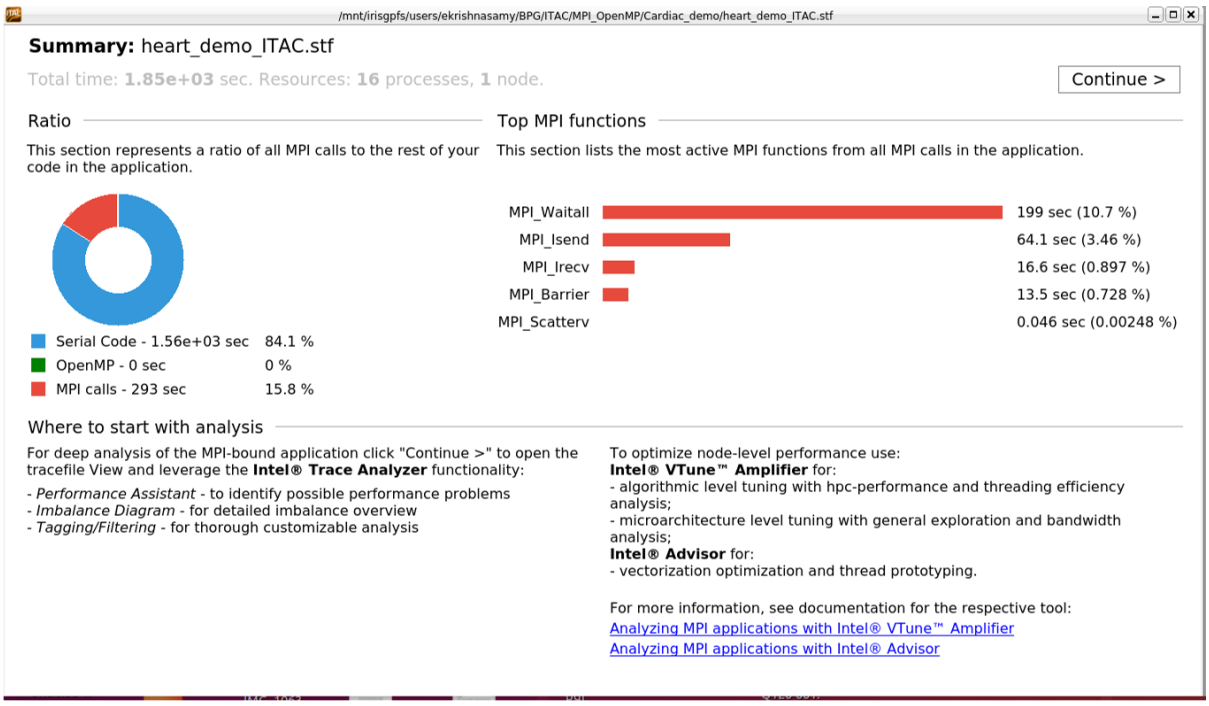


Figure 38. Intel Trace and Collector summary

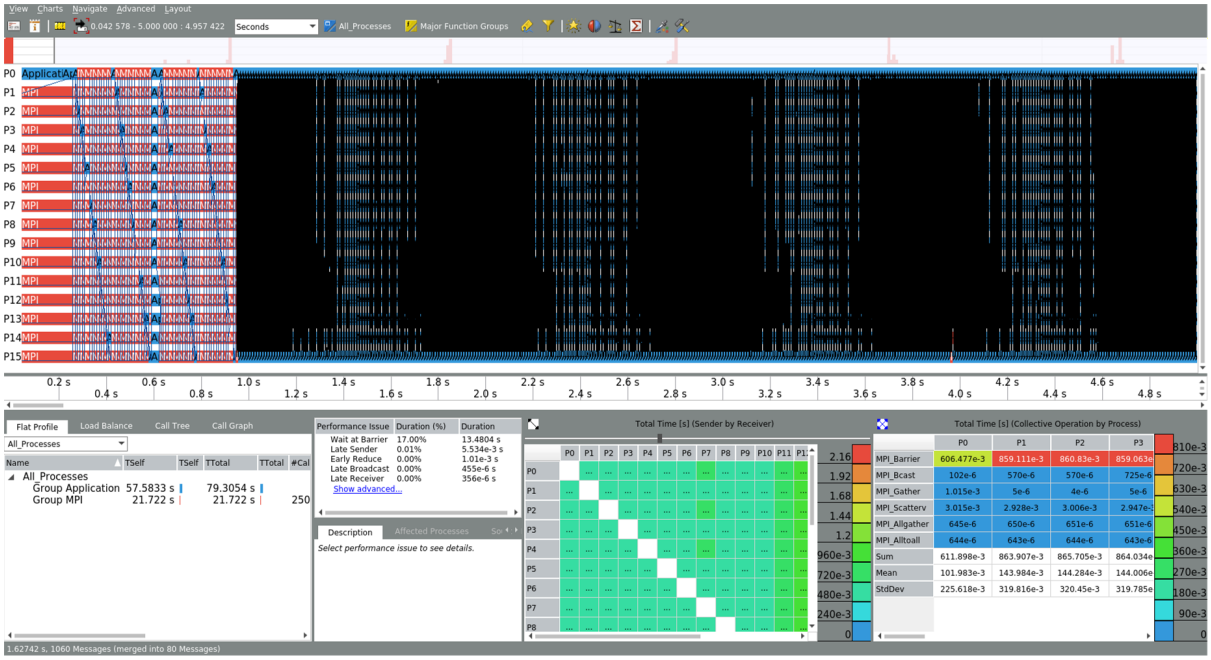


Figure 39. ITAC: function profile, collective operation profile, message profile and event timeline

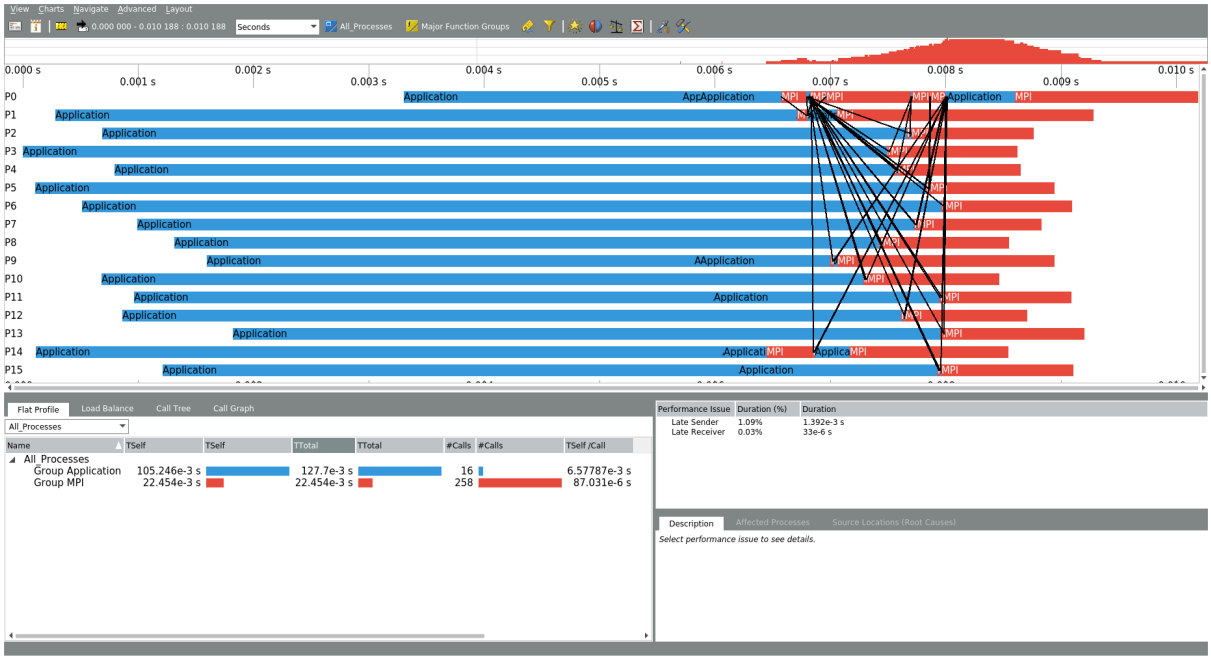


Figure 40. ITAC: event time

3.5.4.5. MPI Tuner

In general, MPI tuner has four modules, they are, cluster specific, application-specific, fast application-specific, and topology rand placement optimisation. Among these, cluster specific is the default. For the remaining setup, see below.

```
# application specific  tuning
$ mpitune --application "mpirun -n 32 ./a.out" -of ./myprog.conf
$ mpirun -tune ./myprog.conf -n 32 ./a.out
```

```
# fast application-specific tuning
$ mpitune --fast --application "mpirun -n 32 ./myprog" -o ./myprog.conf
$ mpirun -tune ./myprog.conf -n 32 ./a.out

# topology awareness rank placement optimisation
$ mpitune --rank-placement --application "mpirun -n 32 ./myprog"
--hostfile-in hostfile.in --config-out ./myprog.conf
$ mpirun -tune ./myprog.conf -n 32 ./a.out
```

Topology and placement optimisation is recommended for numerical computations such as, for example, stencils, collective operations on subsets of ranks, and neighborhood operations.

3.5.5. Environment Variables for Process Pinning OpenMP+MPI

While programming for shared memory and distributed memory, it is advisable to pin and bind the processors or threads across the available compute cores. By doing this, threads and processors can be put to close to each other; this will minimise the costly operation of remote memory access. Table 22 shows the important environmental variables and their explanation.

Table 22. Environment Variables for Process Pinning

Variable	Explanation
I_MPI_PIN=<arg>	enable - yes - on - 1 Enable process pinning; disable - no - off - 0 Disable process pinning
I_MPI_PIN_PROCESSOR_LIST=<value>	<l>, <l>-<m>and <k>,<l>-<m>: CPU cores can be chosen in here
I_MPI_PIN_CELL=<cell>	unit: Basic processor unit (logical CPU);core: Physical processor core
I_MPI_PIN_DOMAIN=<mc-shape>	processors or threads can share a resource in caches and NUMA
I_MPI_PIN_ORDER=<order>	order: can be scatter, compact, spread and bunch.

The I_MPI_PIN_DOMAIN variable defines the processor pinning for the OpenMP/MPI hybrid programming. This also controls how the physical CPU cores are grouped for the MPI processor. And I_MPI_PIN_ORDER defines how the those grouped (physical CPU cores) MPI cores mapped on the compute node. There are different ways to map using different options; for more options, please refer to[32]. Figure 41 illustrates the compact CPU cores mapping and Figure 42 illustrates CPU cores spread methodology.

```
# example for compact
$ I_MPI_PIN_DOMAIN=2
$ I_MPI_PIN_ORDER=compact
```

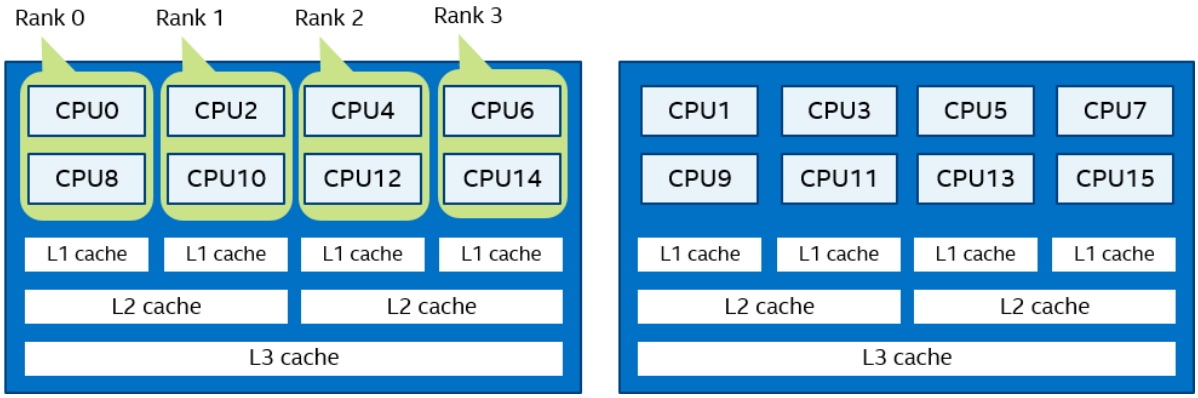


Figure 41. Example for I_MPI_PIN_DOMAIN=2 I_MPI_PIN_ORDER=compact

```
# example for scatter
$ I_MPI_PIN_DOMAIN=2
$ I_MPI_PIN_ORDER=scatter
```

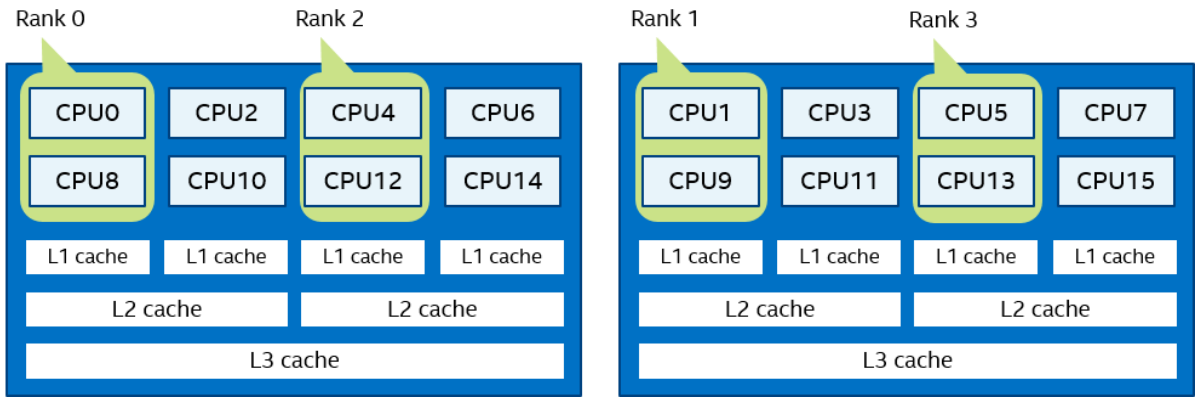


Figure 42. Example for I_MPI_PIN_DOMAIN=2 I_MPI_PIN_ORDER=spread

OpenMPI also offers similar functionality like Intel MPI. For example, two flags *--bind-to-core* and *--bind-to-socket* puts the MPI cores and OpenMP threads closer to each other for the better computational performance.

```
# example for scatter
$ I_MPI_PIN_DOMAIN=2
$ I_MPI_PIN_ORDER=scatter
```

The above example shows the processor pinning for the Intel compilers. But, we can also use the other MPI compilers for Intel Skylake. The below environmental variables show an example case for OpenMPI and MVAPICH2 compilers.

```
# example for OpenMPI (set cores close to each other)
export OMP_NUM_THREADS=2
# Set mpi processes close to each other and print out CPU affinity
srun -n 56 --bind-to core --map-by core --report-bindings ./a.out
```

```
# example for MVAPICH2 (set cores close to each other)
export MV2_ENABLE_AFFINITY=0          # Disable for hybrid mode
export MV2_CPU_BINDING_POLICY=hybrid  # Enable hybrid (hybrid|bunch|scatter)
export MV2_CPU_BINDING_LEVEL=core     # Binding MPI processes
                                        (core|socket|numanode)
export MV2_SHOW_CPU_BINDING=1         # Print out the CPU affinity
export OPM_NUM_THREADS=2              # Number of threads (openmp)
srun -n 56 ./a.out
```

3.6. European SkyLake processor based systems

3.6.1. MareNostrum 4 (BSC)

MareNostrum4 is a supercomputer based on Intel Xeon Platinum processors from the Skylake generation. It is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high performance network interconnect and running SuSE Linux Enterprise Server as operAting system. Its current Linpack Rmax Performance is 6.2272 PFLOPS.

This general-purpose cluster consists of 48 racks housing 3456 nodes with a grand total of 165,888 processor cores and 390 TB of main memory. Compute nodes are equipped with:

- 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10 GHz for a total of 48 cores per node (L1d 32 kiB; L1i cache 32 kiB; L2 cache 1 MiB; L3 cache 1.375 MiB)
- 96 GB of main memory 1.880 GB/core, 12x 8 GB 2667 MHz DIMM (216 nodes high memory, 10368 cores with 7.928 GB/core)
- 100 Gbps Intel Omni-Path HFI Silicon 100 Series PCI-E adapter
- 10 Gbps Ethernet
- 200 GB local SSD available as temporary storage during jobs ($TMPDIR=/scratch/tmp/[jobid])
- Lenovo Rear-door Heat Exchanger
- Slurm batch system

Figure 43. MareNostrum 4 System at BSC

MareNostrum uses a Rear-Door Heat Exchanger (RDHX) for the cooling of the racks. The RDHX brings water to the rack to reduce heat. Similar to a car radiator, the RDHX replaces the rear door of the rack and absorbs heat from the exhaust of air-cooled systems (Figure 44). The Lenovo ThinkSystem SD530 nodes have 5 hot-swap fans which are used to cool all components (Figure 45). In addition, each power supply has its own integrated fan.

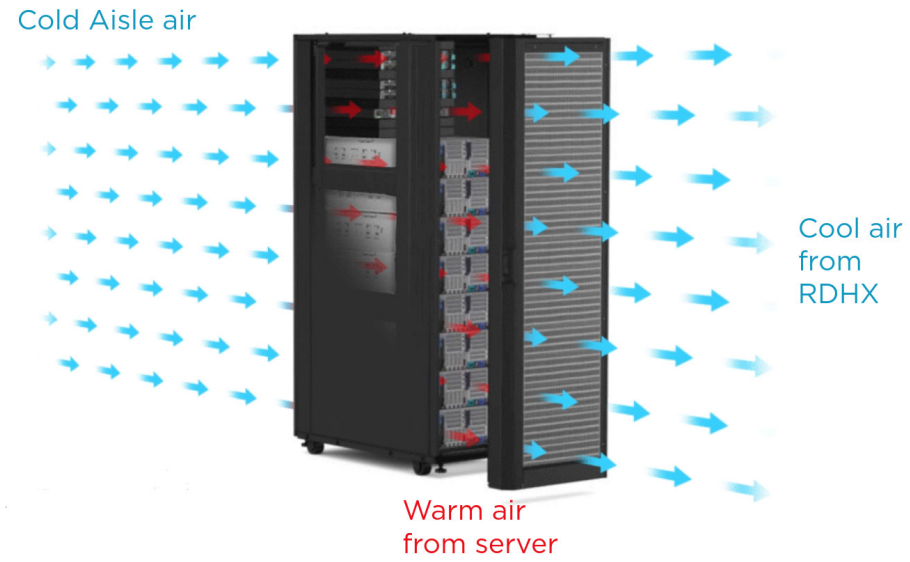


Figure 44. Rear-door Heat Exchanger

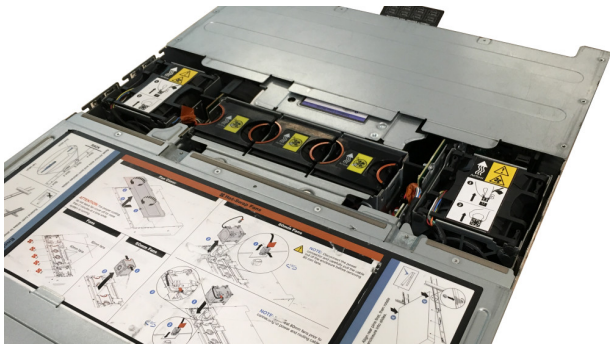
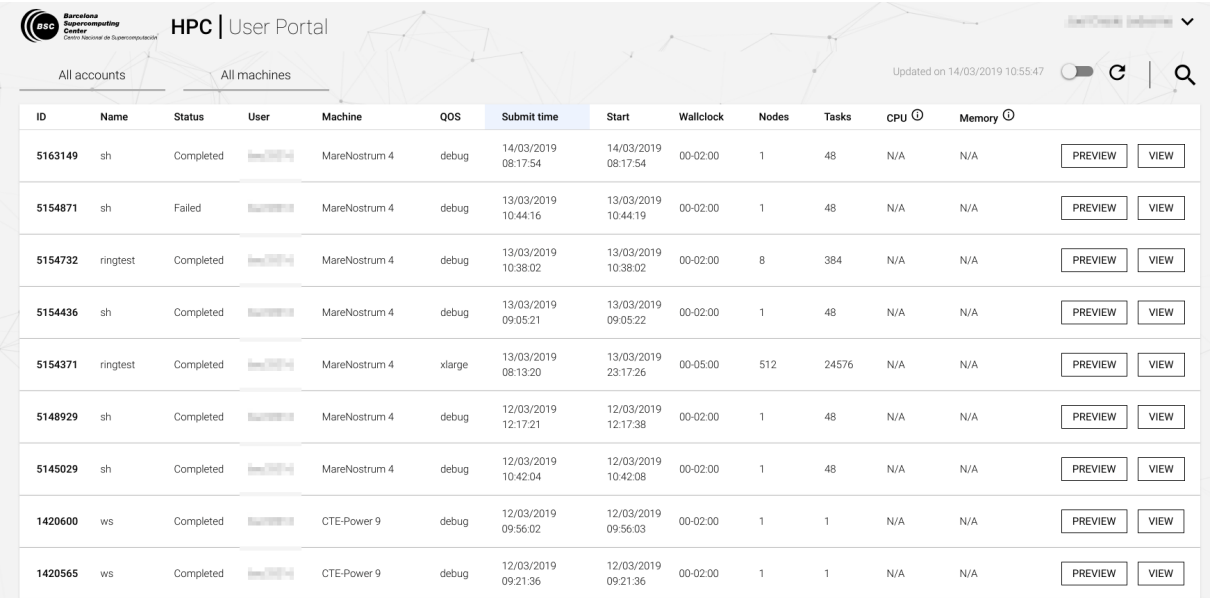


Figure 45. Location of hot-swap fans

3.6.1.1. Power consumption measurment on MareNostrum

On MareNostrum, the average power consumption of any job can be obtained on the BSC UserPortal web (https://userportal.bsc.es) provided by the centre. The HPC User Portal is a job and resource monitoring platform developed by the BSC Support Team's software engineers. With it, every HPC machine user can check the status and general resource usage metrics of the jobs launched. This web also provides a job metric histogram, which will show how the job has evolved during the time of its execution in terms of CPU usage, memory usage and power consumption. The power consumption reported by the web is gathered by Slurm. Slurm gets energy consumption data from hardware sensors on each core/socket, using RAPL (Running Average Power Limit) sensors.

The main page of the HPC User Portal is the job monitoring screen. It will list all your jobs launched in all the machines by every account you have, as it can be observed on the Figure 46 . This list contains a brief listing of the general characteristics of each job (like its name, user, status, node/task configuration…). If the job listed is in the "running" status, it will also show you the current CPU and memory usage.



ID	Name	Status	User	Machine	QOS	Submit time	Start	Wallclock	Nodes	Tasks	CPU	Memory	
5163149	sh	Completed		MareNostrum 4	debug	14/03/2019 08:17:54	14/03/2019 08:17:54	00-02:00	1	48	N/A	N/A	PREVIEW VIEW
5154871	sh	Failed		MareNostrum 4	debug	13/03/2019 10:44:16	13/03/2019 10:44:19	00-02:00	1	48	N/A	N/A	PREVIEW VIEW
5154732	ringtest	Completed		MareNostrum 4	debug	13/03/2019 10:38:02	13/03/2019 10:38:02	00-02:00	8	384	N/A	N/A	PREVIEW VIEW
5154436	sh	Completed		MareNostrum 4	debug	13/03/2019 09:05:21	13/03/2019 09:05:22	00-02:00	1	48	N/A	N/A	PREVIEW VIEW
5154371	ringtest	Completed		MareNostrum 4	xlarge	13/03/2019 08:13:20	13/03/2019 23:17:26	00-05:00	512	24576	N/A	N/A	PREVIEW VIEW
5148929	sh	Completed		MareNostrum 4	debug	12/03/2019 12:17:21	12/03/2019 12:17:38	00-02:00	1	48	N/A	N/A	PREVIEW VIEW
5145029	sh	Completed		MareNostrum 4	debug	12/03/2019 10:42:04	12/03/2019 10:42:08	00-02:00	1	48	N/A	N/A	PREVIEW VIEW
1420600	ws	Completed		CTE-Power 9	debug	12/03/2019 09:56:02	12/03/2019 09:56:03	00-02:00	1	1	N/A	N/A	PREVIEW VIEW
1420565	ws	Completed		CTE-Power 9	debug	12/03/2019 09:21:36	12/03/2019 09:21:36	00-02:00	1	1	N/A	N/A	PREVIEW VIEW

Figure 46. HPC UserPortal - Home page and list of jobs

Once the desired job is located, you can check its properties using either the preview button or the view button. The difference between them is the level of detail that they will give. Clicking on the view button will show how the job has evolved during the time of its execution in terms of CPU usage, memory usage and power consumption as it can be observed on Figure 47. You can also download this information by clicking in the top right corner of each histogram and selecting your preferred format option, including CSV format.

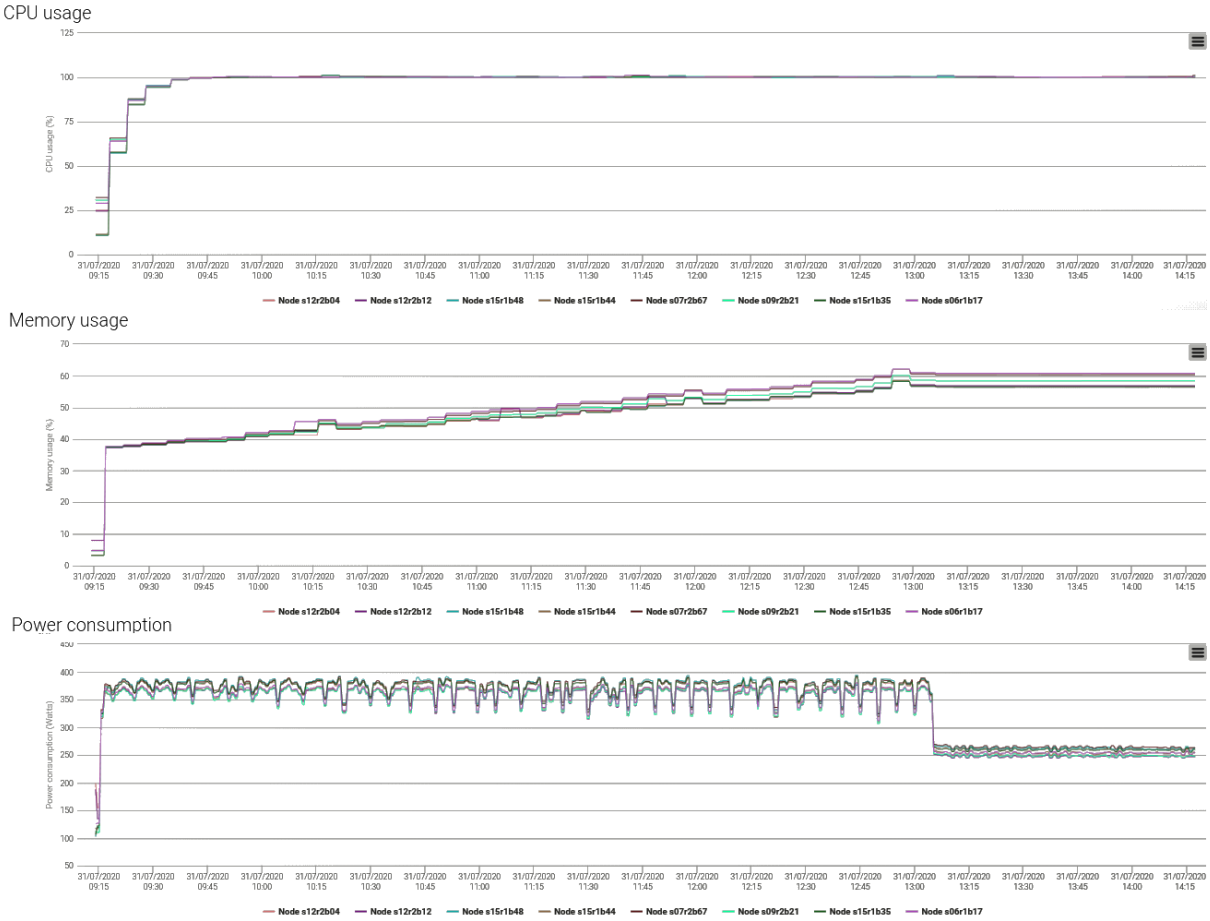


Figure 47. HPC UserPortal CPU and Memory usage and Power consumption histogram

3.6.2. SuperMUC-NG (LRZ)

SuperMUC-NG is the current leadership class system of Leibniz Supercomputing Centre (LRZ). It is a Intel Xeon Skylake processor based machine [107] with a peak performance of 26.8 PFLOPS. SuperMUC-NG was the fastest supercomputer of European Union and 9th in the world according to TOP500 November 2019 rankings [58]. According to recent, TOP500 June 2020 rankings SuperMUC-NG is currently 13th fastest supercomputer in the world.

Figure 48. SuperMUC-NG System at LRZ

SuperMUC-NG is a Lenovo-built system featuring 6,336 thin and 144 fat nodes correspondingly with 96 GB and 768 GB per node memory. Each of these Lenovo ThinkSystem SD650 [108] compute nodes is equipped with 2x24 core Intel Skylake Xeon Platinum 8174 processors. The Thermal Design Power (TDP) mode of the compute nodes in regular user operation is 205 W. The Figure 49 illustrates the inside view of two SD650 nodes sharing a tray and a water-loop.

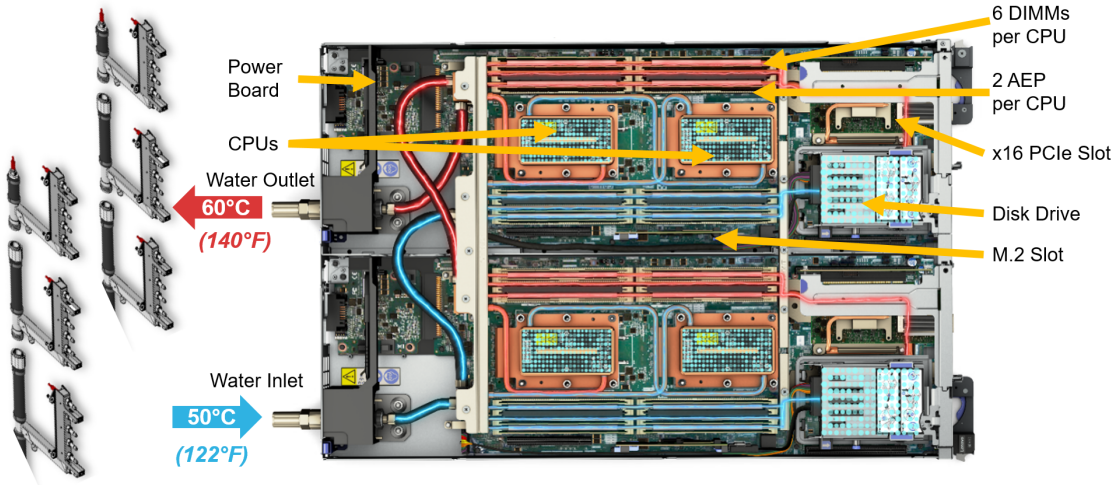


Figure 49. Lenovo ThinkSystem SD650 [108], [109]

Like the predecessor SuperMUC Phase1 and Phase2 systems [138], [139] , SuperMUC-NG uses direct hot water cooling for chilling its active components like processors, memory, and voltage regulators. LRZ's SD650 nodes

were developed to withstand up to 50°C inlet temperatures. Depending on the running workload, the outlet temperature of a SD650 node can reach 60°C which is sent through adsorption chillers for the generation of the cold water (typically 20°C) used for cooling storage and networking equipments. The electricity used for SuperMUC-NG as well as for predecessor SuperMUC system, is generated using 100% renewable energy sources. SuperMUC-NG system uses SLURM [112] as a resource management and scheduling system and Intel OmniPath 100G interconnect network. The system features IBM Spectrum Scale (GPFS) high performance parallel filesystem with a capacity of 50 PB with 500 GB/s I/O bandwidth. 20 PB capacity with 70 GB/s bandwidth is available for long term data storage.

Additionally, SuperMUC-NG system features a separately operated segment for cloud computing. The table below outlines some details of this system segment [107].

Table 23. Detailed view of the cloud computing segment

Cloud Node Type	Number of Nodes	CPU Cores	RAM (GB)	GPU Type
Cloud Compute Node	82	40 (@ 2.4 GHz)	192	-
Cloud GPU Node	32	40 (@ 2.4 GHz)	768	2x Nvidia Tesla V100 16 GB
Cloud Huge Node	1	192 (@ 2.1 GHz)	6000	-

4. AMD Rome Processors

AMD EPYC 2 Rome is the second generation of EPYC processors, based on the Zen2 microarchitecture. The 7002-series family lineup has 15 models for two-socket server configuration (Table 24). AMD decided that its processor lines - Naples, Rome, as well as the next generation - Milan, will be compatible.

Based on the variations of number of cores (Table 24 and Table 25) several selections are possible for the new Rome line:

- While the 8-core version is a low price solution, it is unsuitable for HPC environments
- 12-, 16-, 24-cores provide generally good performance and are suitable for many HPC workloads
- 32-core EPYC2 CPU offers great price/performance with relatively good core count and suitability for HPC workloads
- 48-, 64-core EPYC2 CPU - suitable for many HPC workloads. While the higher number of cores provides great cost effectiveness and energy-efficiency, many memory-bound HPC applications might show diminishing results. If the application is not memory-bound, those EPYC2 models are excellent selection.

In this respect, many of the family models provide over 1 TFLOPS double precision performance, with several CPUs capable of over 2 TFLOPS (Figure 50 illustrates the performance details).

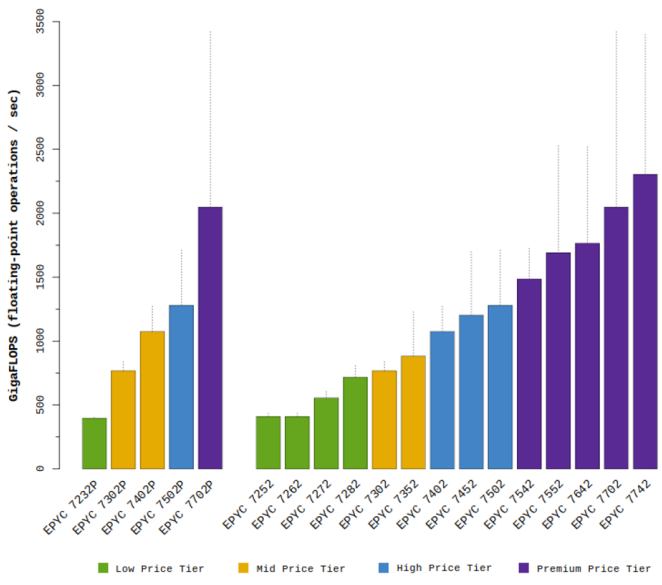


Figure 50. AMD EPYC2 Family Theoretical Peak Performance (AVX2 instructions)

Several centers, two of which are presented in section Section 4.6, “European AMD processor based systems”, have already selected EPYC 7742 processors for their new systems. In September 2019 AMD introduced the EPYC 7H12, which uses 7 nm Zen 2 architecture (64 cores/128 threads, 256 MiB L3 cache, and 128 PCI-4 lanes). The new processor utilises the highest base clock speed, running at 2.6 GHz and boosts up to 3.3 GHz, making it the fastest Rome 64 CPU available (7742, by comparison, runs at 2.25 GHz). The 7H12 is geared for liquid-cooling builds in order for the high TDP of 280 W to be maintained.

The Zen 2 microarchitecture is based on the x86-64 architecture and it follows the AMD multichip module design, although significant design changes are observable in the Zen2 microarchitecture, including the support of PCIe version 4.0.

Table 24. AMD Family Information

Basic Information	Description
Architecture codename	Zen 2
Launch date	7 Aug 2019
Base frequency	Minimum 2 GHz
L1 cache	4 MiB
L2 cache	32 MiB
L3 cache	256 MiB
Manufacturing process technology	7 nm, 14 nm
Maximum frequency	3.4 GHz
Number of cores	Up to 64 per socket (with options for 8-, 12-, 16-, 24-, 32-, 48-, 64-cores)
Number of threads	Up to 128 per socket
Notable changes in Zen2	Up to 16 double-precision FLOPS per cycle per core. Full support for 256-bit AVX2 instructions with two 256-bit FMA units per CPU core
NUMA Architecture	Simplified design - one NUMA domain per CPU Socket Uniform latencies between CPU dies and fewer hops between cores

A simple comparison with the Intel Cascade Lake CPUs highlights the tremendous performance advantages of EPYC Rome chips, compared to the ones offered by Intel (see Table 25).

Table 25. Comparison between AMD Rome and Intel

Feature	EPYC ROME	Xeon Cascade Lakes-SP	Xeon Cascade Lakes-AP
Cores	8 to 64	6 to 28	32 to 56
PCI Generation	4	3	3
PCI Lanes	128	48	64 (available: 40)
Memory Channels	8	6	12
Max. Memory Speed	3200 MHz	2933 MHz	2933 MHz
Max. Memory Capacity	4 TiB	1-4.5 TiB	3 TiB
Lithography	7 nm chip module and 14 nm I/O module	14 nm	14 nm

4.1. System Architecture

Zen 2 continues the multichip module design architecture, with the Rome processors designed around 9 chiplets (known as multi-chip modules) in a mixed process design. Eight of those are Compute core dies (CCD), which are with a size of 74 mm^2. Each of the CCDs contains two Compute Core complexes (CCX) which have four Zen 2 cores with their own L2 cache and a shared L3 cache. Each CCX has 16 MiB of L3 cache. Each of the logic chips contains 3.9 billion transistors and are manufactured on TSMC 7 nm process. The 7-nanometer process produces twice the transistor density, leading to a total of 39.54 billion transistors on the chip. Both the CPU and cache complexes, being built on the 7nm process are able to significantly reduce the power and space required in order to double the cores and the available L3 cache. Each core supports simultaneous multi threading (SMT), allowing the execution of 2 threads per core.

Figure 51. AMD Comparison between Monolithic, Naples and Rome Designs [49]

For Zen2, AMD has reshaped the communication structure entirely. The die positioned in the middle (ninth die) is a dedicated Input/Output die (IOD) and is responsible for connecting each compute die to all others as well as to other CPUs through the infinity fabric links (see Figure 51 upper right plot). In other words, no chiplet can directly communicate with another, they connect first through the I/O die. Within the I/O chip are all the PCI lanes for the processor, the memory channels and the Infinity Fabric links. It is etched in the GlobalFoundies 14 nm proces surrounded by the processing chiplets containing the core complexes (see Figure 51). Compared to the Naples microarchitecture, where to send data across CCXes, each CCX is connected to a Scalable Data Fabric (SDF) by means of he Cache-Coherent Master (CCM), quite a lot of balancing issues were observed. In the Zen2, due to the presence of an I/O central hub, all communications occur off-complex chip. This makes the software optimisation significantly easier, compared to the NAPLES since each processor would have only one memory latency environment and thus each core will have the same latency to simultaneously reach all eight memory channels.

In Zen2 the I/O chip houses a total of 4 DDR blocks, containing a pair of memory controllers, contributing to a total of 8 memory controllers. The controllers support a memory of 3200 MT/s and if all the memory slots are filled, one can yield a maximum of 410 GiB/sec peak memory bandwidth per socket. In Rome the I/O chip support PCIe 4.0 lanes, which are used to bind a pair of Rome complexes (64 bi-directional lanes used to connect the Rome chips to one another via Infinity fabric). The Rome processor has 128 PCIe 4.0 lanes that are able to reach out to peripheral devices, as well as the ability to be utilised as NUMA interconnects in a two-socket server. In a two-socket system, although each processor supports 128 PCIe lanes, the NUMA coupling requires 64 lanes to implement the NUMA links between the two sockets, therefore leaving a total of 128 PCIe lanes for peripherals that are configurable in a variety of ways (Figure 52). One PCIe 4.0 x16 is able to provide up to 36 GB/s in both directions, providing up to 288 GB bi-directional bandwidth. Since each CPU has 8x16 PCIe 4.0 links present, they can be split up between up to 8 devices per PCIe root. Since Rome implements version 4 of the PCIe, it is capable of doubling the bandwidth and supports 100 Gbits/s and 200 Gbits/s adapters. Moreover, the lanes can be used singly, can be paired (x2) for storage purposes, potentially providing around 50 NVMe drives as well as access to a high-speed host adapter (i.e. 200 Gbits adapter (HDR) from Mellanox). Moreover, there is a maximum of 512 Bytes Direct-Memory Access payload size through the PCIe links.

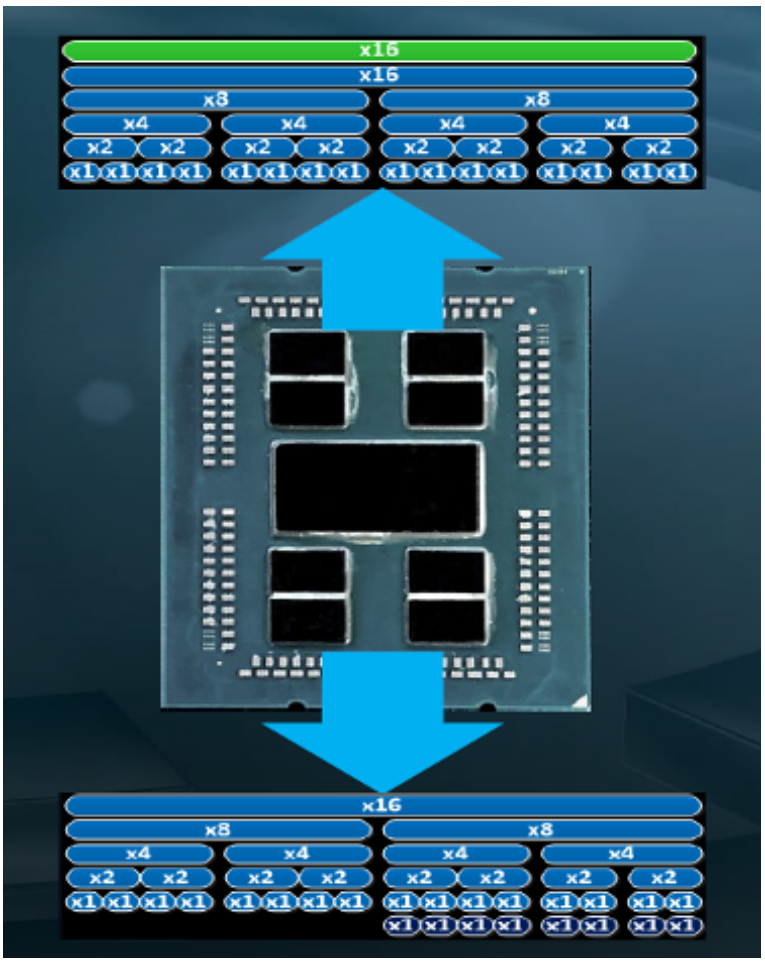


Figure 52. AMD Rome PCIe and how bifurcation can be used: with a x16 PCIe you can split it in multiple ways: 1 x8 and 2×4, or 4×4 [48]

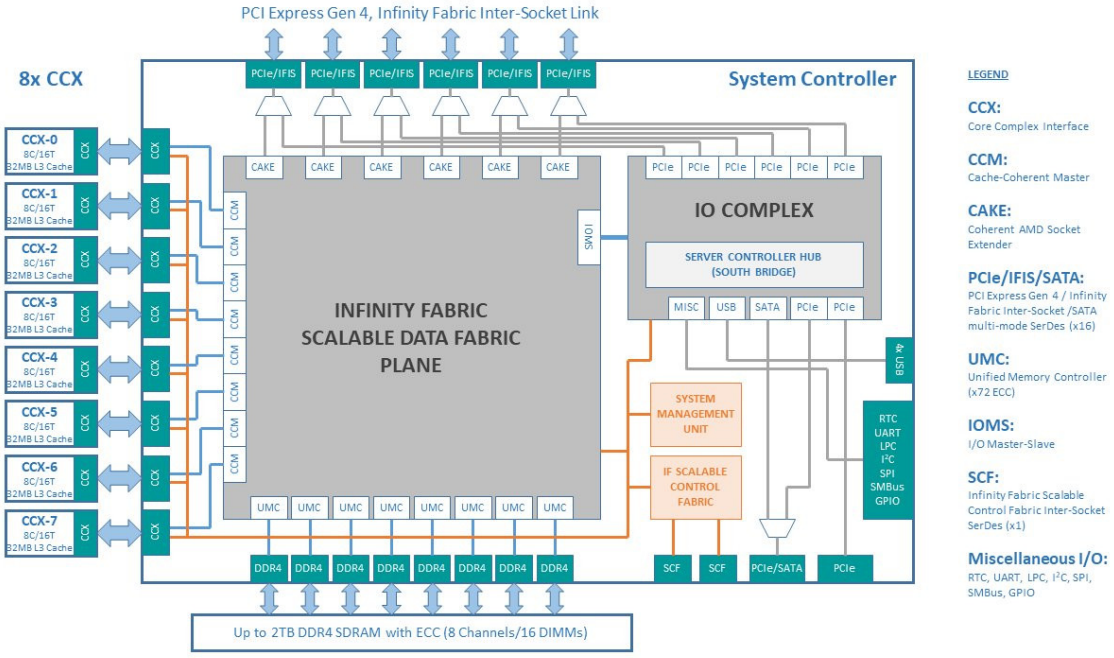


Figure 53. AMD Rome Infinity Scalable [48]

There are some changes in the cores within the Zen2 architecture. There are four arithmetic logic units (ALUs), and the address generation unit (AGU) count in Zen2 is increased to three. Both ALU and AGU schedulers are improved in Zen2, with increased size for the register file and reordering of the buffers. The imbalance in simultaneous multi threading (SMT) observed in Naples ALUs and AGUs is dealt with by tweaking the algorithms.

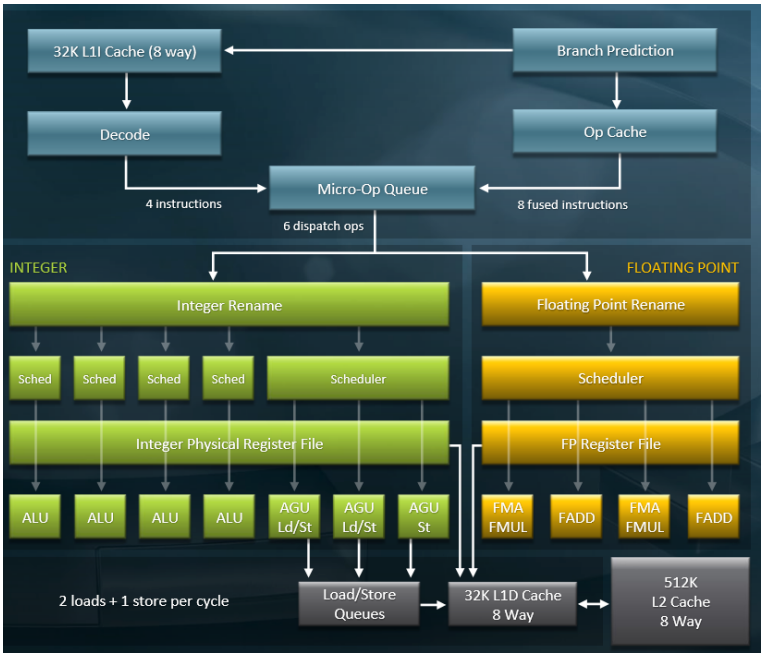


Figure 54. Integer and floating-point instruction units in the Zen 2 cores [49]

The new architecture has improved the branch target buffer (BTB) instruction pre-fetching, larger op cache, and optimised instruction cache compared to Naples. AMD enlarged the µOP cache to increase the instruction stream throughput, as well as adapted the µOP cache tags.

Since the BTB which holds the branch instruction address and their respective target (Figure 56) attempts to predict the branches so possible pipeline stalls can be mitigated, EPYC Rome architecture has extended almost twice the branch capacity. The L1 BTB is now of size 512 entry, while the L2 being of 7168 entries. Along with the L0 BTB at 16 entries (8 forward and 8 backward taken branches) and the indirect target array to 1K entries, this leads to a decrease of about 30% of the mispredict rate, which in turn helps to save power.

Due to the huge pipeline length and width, a bad branch prediction can lead to more than 100 slots being flushed which causes a loss in performance. To mitigate this, AMD preserves the hashed perceptron prefetch engine for L1 fetches, it improved the TAGE prediction, which is used for L2 prefetches and beyond by adding the ability for longer branch histories by using additional tagging. Branch evaluation is performed later, thereby allowing the actual branch solution to alter the fetching of instruction and consequently refine the prediction.

To predict a return address from a near call, the system uses a 32-entry return address stack (RAS) with 31 entries available in single-threaded mode, and 15 per thread in two thread mode. If the RAS can’t recover from a misprediction (which is rarely the case) it is flushed.

Zen2 has doubled the indirect target array prediction size to 1024-entries, which is responsible for instance calls through a function pointer. Therefore in a situation where a branch has multiple targets, the prediction selects among them based on the global history located at the L2 BTB.

Importantly the translation between virtual and physical fetch addresses have been moved in the ZEN architecture into the branch unit, which enables earlier instruction fetching. The translation is supplemented through a two-level translation lookaside buffer (TLB). It has retained the same size in ZEN2 as in the previous generation. The L1 instruction TLB which is fully-associative and has 64 entries. It can hold 4 kiB, 2 MiB, or 1 GiB page table entries. The L2 instruction TLB is 8-way set-associative and has 512 entries and can hold 4 kiB and 2 MiB page table entries.

For the vector size, Naples had a pair of 128-bit vectors and therefore two operations were required to accomplish an AVX-256 instruction. In the Zen2 microarchitecture, the vector is extended to 256 and thus there is only one instruction required per clock, which naturally leads to less wasted power and performance gains. To feed the increasing floating-points, Rome has extended the L1 read/write bandwidth to 256 Bytes per cycle. Additionally, the loads and stores are extended to 256-bit, providing a continuous feeding of the FMA units. While double-precision multiply necessitated four cycles in the previous generation EPYC processors, in Rome it requires only three, improving not only the throughput but the power efficiency of floating-point (Figure 55 for vector size improvement, although it is listed only for integer) operation. Moreover, in the floating-point unit, there are up to four micro-ops per cycle accepted in the queues from the dispatch unit, feeding into a 160-entry physical registry file. Consequently, those are moved into four execution units, fed with 256 Bytes of data (load/store mechanism). When we consider the integer unit, it is able to take up to six micro-ops each cycle, feeding into a 224-entry reorder buffer. The integer unit, for instance, has 7 execution ports, consisting of 4 ALUs (arithmetic logic unit) and 3 AGUs (address generation unit).

Finally, the floating-point width for the Rome is 256 bits, as well as the bandwidth into and out of load/store units consisting of 256-bit data paths. The floating-point registers are as well of size 256 bits. This ensures 4x floating-point performance increase of Rome compared to Naples, due to the presence of twice as many cores and vector units that are doubled in width.

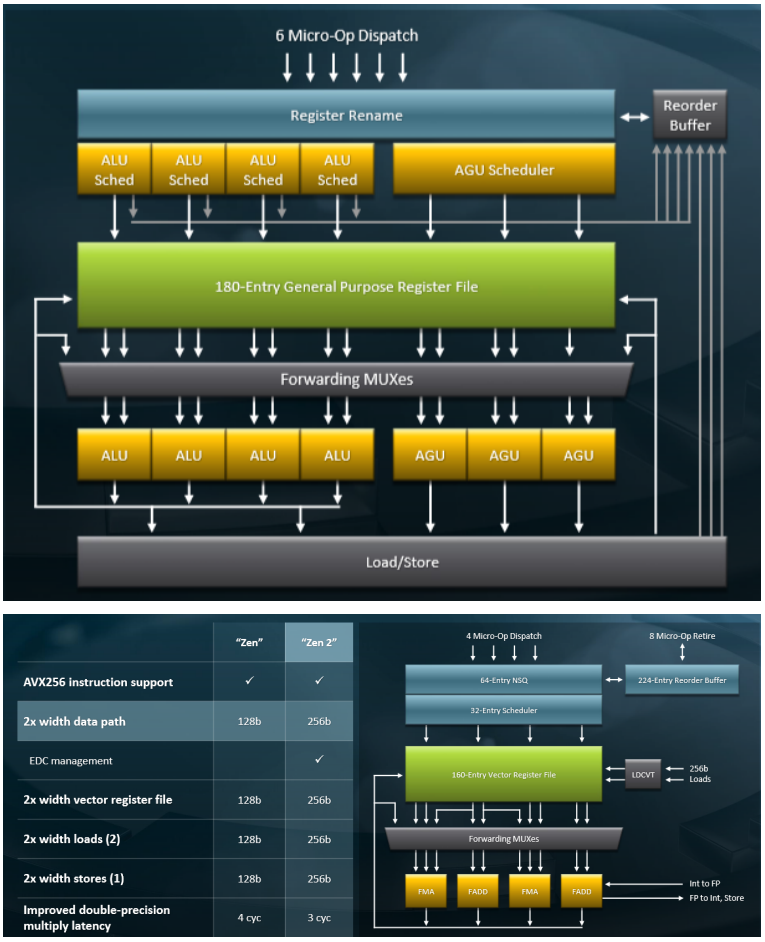


Figure 55. IPC for integer instructions [49]

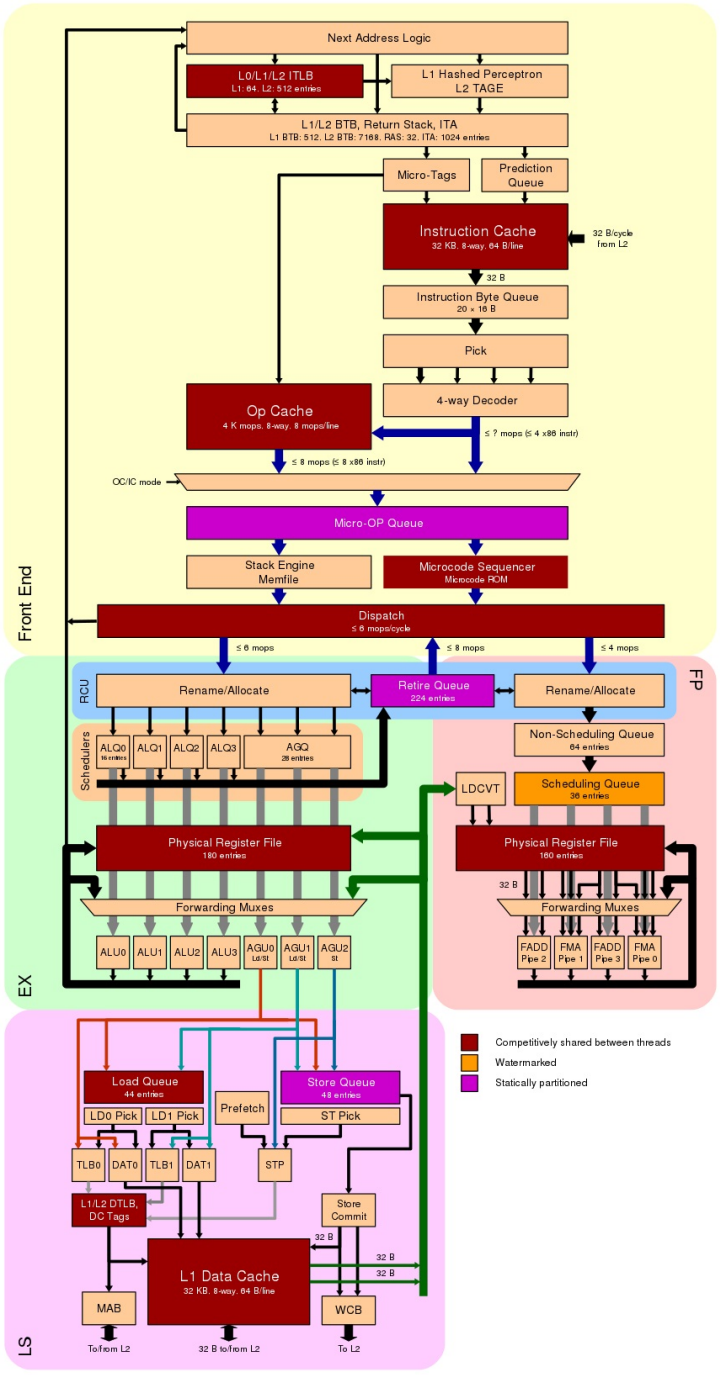


Figure 56. AMD Zen 2 core diagram [47]

4.1.1. Cores - «real» vs. virtual/logical

The cores in a modern processors are very complex, there is a number of executional units within each one: decoding-; load/store-; integer-; floating point-; vector- units, just to mention a few. For some of these even more than one. To take advantage of this the core have a mode where it can run multiple streams of instructions, simultaneous multi threading (SMT) [75] or Hyper Threading (HT) [76].

In practical terms the SMT manifests itself by providing a set of virtual cores. In the case of selecting SMT-2 mode twice as many and for SMT-4 four times. Seen from the Operating System's view (including user perspective) it looks like we have 2x or 4x number of cores to use. Most tools and queries to the OS report the number of virtual cores. It takes a closer look to reveal if the cores are virtual or not.

The mentioned larger number of cores is somewhat deceiving as the executional units are the same. For diverse load with a wide spectrum of different applications and processes this can be beneficial. It can also help to hide both memory and I/O wait as the other instruction streams can continue without any context switching. This has been less beneficial for HPC workloads where the instruction streams are normally equal as MPI jobs tend to run multipls copies of the same executable and OpenMP created multiple identical threads from a parallised loop. Hence the instructions streams tend to request the same executional units of which there a fixed amount of regardless of how many virtual cores there are.

4.1.2. Memory Architecture

The 4 memory blocks on the Zen2 support 2 DDR4 DIMM per block, each of a maximum of 3200 MT/s, as well as support for RDIMM, LRDIMM, 3DS, and NVDIMM form factors. Thus the maximum capacity with two DIMMs per channel will be 4 TiB.

When we consider the bandwidth on the chip, one should note that the bus width of the memory controller is 8 bytes per channel, making the maximum memory bandwidth per socket 8 x 8 x 3200 = 204.8 GiB/s. In the previous generation (Naples), the chiplets could perform a 16 Byte read and 16 Bytes write operation in one fabric clock. Respectively in Rome, the chiplets can perform 32 Bytes read and 16 Bytes write per fabric clock as shown in figure [50] . Each of the four memory blocks contains 2 controllers in charge of 2 DIMMs each. According to AMD there is an improved average memory latency by 24 ns (19%), while the minimum (local) latency only increases 4ns with chiplet architecture.

On a 64 core socket the memory controllers (see Figure 59), each with two memory channels, interconnected through the Infinity fabric with a 16 Byte write at memory clock Figure 59 point to a bottleneck at the entry point of the internal chip interconnect, which serves two PCI-4 128 bits, therefore twice decreasing the maximum data transfer from external memory to processor. Thus a DDR4 running at 3200 MHz in reality transfers data with a frequency of 1600 MHz. The bus width of each memory controller is 8 bytes per channel, achieving a maximum memory bandwidth for a socket at 8 x 8 x 3200 = 204.8 GiB/s.

Two CCDs (each with 8 cores) will each have a local connection to a memory controller with two dedicated 32 B read and 16 B write links through the infinity fabric (which puts it at maximum 1600 MHz). The GMI-2 interface functions as an extension of the data fabric from the I/O dies to the CCDS therefore allowing a bi-directional 32-lane IFOP link which is similar to the die-to-die links in the EPYC Naples and Threadripper processors. Therefore AMD increases the die-to-die bandwidth from 16 B reads + 16B write to 32 B read + 16 B write per f-clock.

Each Rome CPU, based on Figure 58 will have the following byte per flop memory ratio. Considering the highest frequency memory is used - DDR4 3200 MHz and the fact that Rome has 8 memory channels with reading 32 Bytes per clock, we will have 32 bytes x 8 channels x 1600 MHz = 409600 Bytes, and the peak performance of 7742 processor being 2.25 TFLOPS, thus the byte per FLOP ratio is 409600/(2.25 x 1000000) = 0.18.

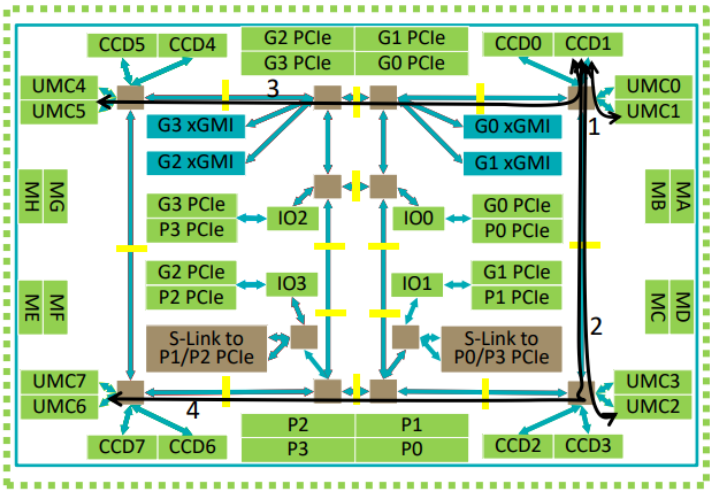


Figure 57. AMD Zen 2 improved memory latency. Switch (depicted in brown): 2 FCLK (1.46 GHz) (low-load bypass, best-case). The yellow lines represent repeater: 1 FCLK (1.46 GHz). The green parts depict a signle domain [50]

For the cache size, each Zen2 core has an available 32 kiB L1 instruction cache. The freed physical space (in Naples the L1-I was 64 kiB) is allocated for the op and branch prediction units. L1 data and instruction caches (both at 32 kiB) are with an eight-way associativity, with a doubled floating-point data path width. Both of them reside inside of the core next to each other. The L2 cache has a 512 kiB 8-way associativity. Compared to L2 which is an inclusive cache, the L3 is non-inclusive. The L3 cache available for each chiplet is 16 MiB (CCX with each having a 32 MiB of L3 cache (see Figure 58)), with all 8 chiplets reaching a combined capacity of 256 MiB L3 cache. Due to the fact that the size of L3 is slightly increased, latency increases as well. L1 remains 4-cycle, L2 12-cycle, while L3 goes to ~40 cycles (indicative for larger caches, which end up with somewhat slower latency).

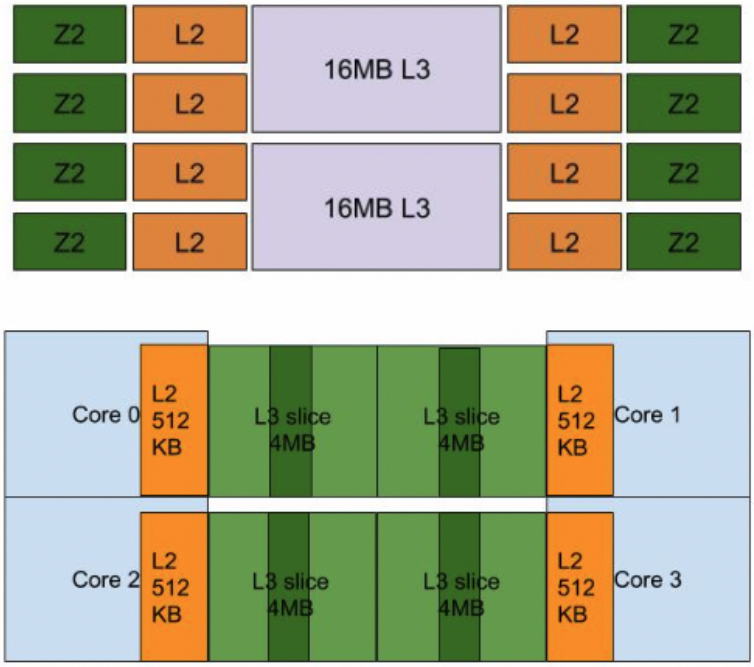


Figure 58. AMD Rome Cache Representation for a CCX.Upper plot represent how cores (Z2), L2 and shared L3 within a CCX. Lower plot represents the L3 cache for a group of 4 cores with the CCX and its respective L3 segmentation reach

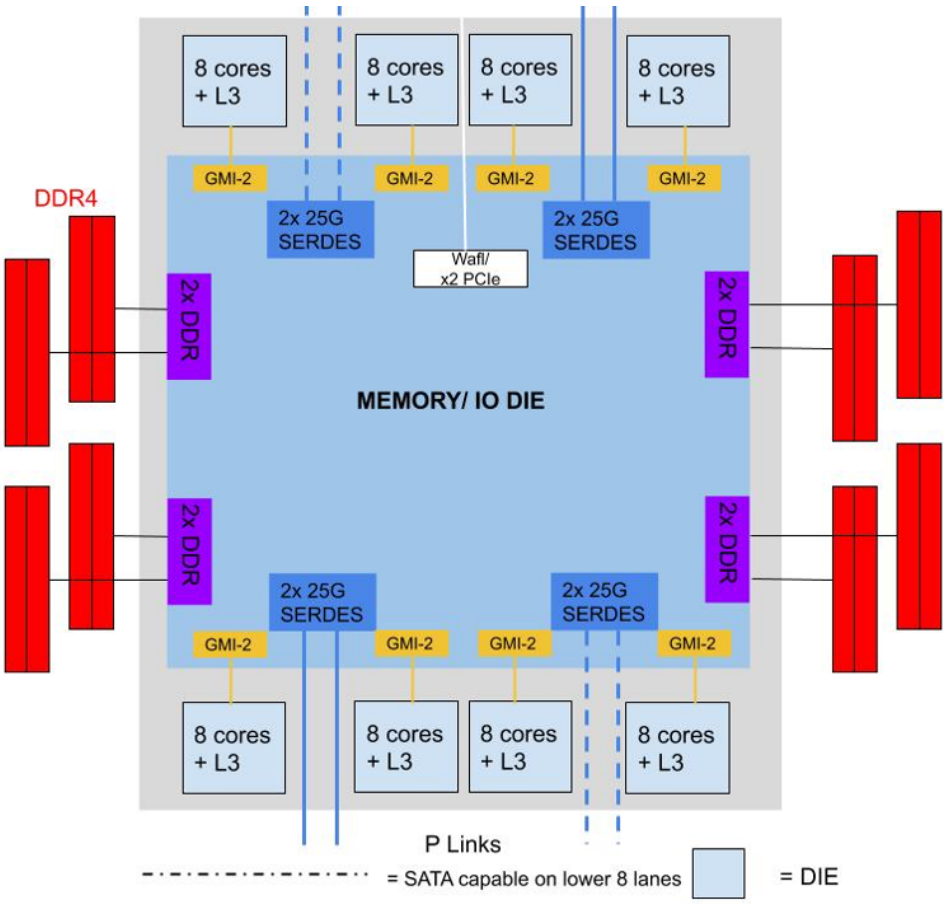


Figure 59. AMD Rome Cache Representation and communication with memory. GMI stands for Global Memory Interconnect which is for the interconnect to one CCD within the Rome

As for the caches feeding the Zen 2 cores, all of the structures supporting the caches are bigger and provide more throughput, driving up that Instructions Per Cycle (IPC) as seen from Figure 60.

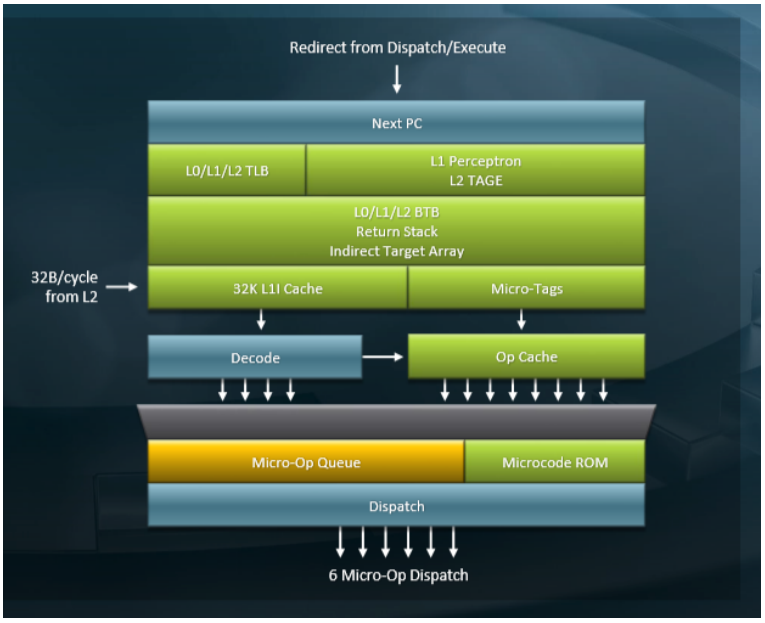


Figure 60. Cache feeding for Zen2

4.1.3. NUMA

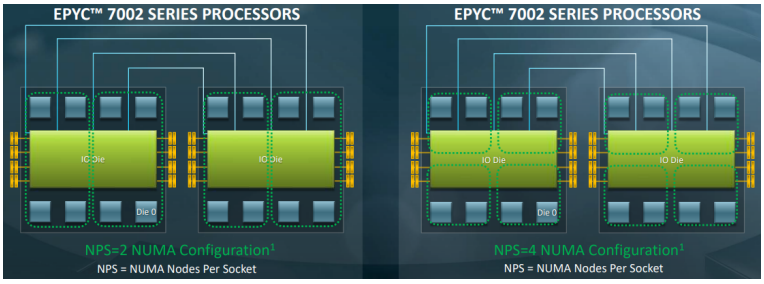


Figure 61. AMD Rome Numa Representations [50]

One of the big performance improvements is the new way the NUMA domains are created. For a two socket configuration, in Naples, there were 8 NUMA domains and 3 different distances from any die to another, where the memory is hanging. In Rome where there are only 2 NUMA domains and just two different distances (see Figure 61) - just one hop from one chiplet through the I/O hub to the memory attached to any processor, and another hop through the infinity fabric to a second I/O hub where the memory is situated at. Rome thus requires fewer NUMA hops when moving data from one part of a processor complex to another. Each of the complexes can also represent a single NUMA domain, achieving more even and therefore better performance across broader types of tasks.

One should keep in mind that although the entire 64-core chip is a one big NUMA node, the chip is in essence 16x4 cores, each having a 16 MiB L3-caches. That means that once you fill the 16 MiB cache, the main DRAM needs to be accessed, although using prefetching can alleviate some of this problem.

Because each chiplet connects to the same I/O and memory hub, one would require one less NUMA hop to reach from one chiplet to another, therefore simplifying the NUMA socket structure and decreasing the sharing memory overhead across the chiplets within a socket and across sockets.

For instance, the EPYC 7702P 64 core topology in a Supermicro WIO 1U platform would be seen below with an L3 cache split across the cores. If one uses 256 GiB of memory, the entire one is attached to a large NUMA node (since all 64 cores are one NUMA node). Similarly, the PCIe is attached to a single NUMA node (compared to Naples where both memory and PCIe's would have been attached across four NUMA nodes). Moreover keeping each socket as a single NUMA domain simplifies virtualisation and application sizing, while keeping the performance consistent.

Figure 62. AMD Rome 7702P Supermicro WIO 1U platform [51]

In a two-socket variant (128 cores/256 threads) the topology is seen Figure 63:

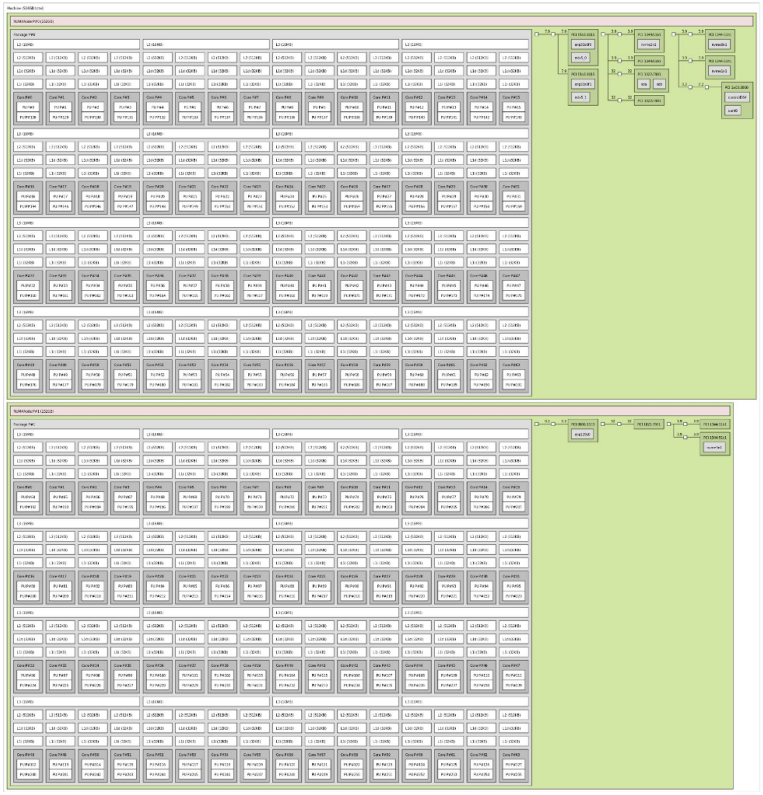


Figure 63. Two socket variant for a WIO platform [51]

Interestingly, AMD Rome allows for multiple NUMA domains on a system, in a similar manner to the Intel Xeon Sub-NUMA clustering (SNC). Thus we can partition the EPYC 7002 into four NUMA nodes, which provides up to two compute dies and ¼ of the I/O die for each NUMA node. Consequently, we are able to keep data flowing through the shortest paths in the system. The default setting in Rome 7002 system is NPS=1 (as seen in Figure 62 and Figure 63). One requires the NPS to be changed to 2 or 4 for testing purposes.

4.1.4. Balance of AMD/Rome system

The balance between memory bandwidth and floating point performance is a key factor in high performance computing. The ratio of floating point operations per byte of memory bandwidth is of great importance.

The memory bandwidth might look impressive, but taken into account that this is an aggregate number for all the 128 cores the number per core is less. The memory bandwidth is just 2.7 GiB/s per core. With a memory bandwidth of up to 340 GiB/s and 3690 GFLOPS the number for each of the nodes on the system Betzy (Section 4.6.2) (with 128 core nodes) is about 10-11 floating point operation per byte of memory bandwidth. Using net measured numbers.

4.2. Programming Environment

4.2.1. Available Compilers

All compilers that run under x86-64 will normally run on the EPYC processor. However, not all compilers can generate optimal code for this processor. Some might just produce a smallest possible common subset of instructions, using x86 instructions and not even attempt to use the vector units. This varies from compiler to compiler and is both vendor and version dependent. There are obvious candidates, the Intel compiler cannot be expected to support the EPYC processor for natural reasons. On the other hand GNU compilers might do a good job optimizing and generating code for the EPYC.

Compilers installed and tested:

- AOCC/LLVM compiler suite, *clang ,flang / cc, fortran* (version 2.0.0)
- GNU compiler suite, *gcc, gfortran, g++*(version 9.1.0)
- Intel compiler suite (Commercial) , *icc, ifortran, icpc* (version 19.0.4)
- PGI compiler, *pgcc, pgCC and pgfortran* (version 20.1)

4.2.2. Compiler Flags

4.2.2.1. AOCC

Table 26. Suggested compiler flags for AOCC compilers

Compiler	Suggested flags
clang compiler	-O3 -march=znver2 -mfma -fvectorize -mfma -mavx2 -m3dnow
clang++ compiler	-O3 -march=znver2 -mfma -fvectorize -mfma -mavx2 -m3dnow
flang compiler	-O3 -mavx2 -march=znver2

The clang compiler is under development with assistance from AMD. The options may change, more information about the usage of the clang compiler is available online [134]. For the Fortran compiler online documentation is also available [135]. This compiler suite is under heavy development and subject to change. Please visit the AMD developer site to obtain the latest information and releases.

The optimisation flags O3 and Ofast invoke different optimisations. The Ofast can invoke unsafe optimisations that might not yield the same numerical result as O2 or even O3. See Section 2.2.1.2, “Usage of optimisation flags” for more on this.

The Zen architecture in the EPYC processor does no longer support FMA4. However, sources claim it still is available and works, See [47]. However, it might suddenly just vanish, hence any usage of the flag -mfma4 should be avoided.

4.2.2.2. GNU

Table 27. Suggested compiler flags for GNU compilers

Compiler	Suggested flags
gcc compiler	-O3 -march=znver2 -mtune=znver2 -mfma -mavx2 -m3dnow -fomit-frame-pointer
g++ compiler	-O3 -march=znver2 -mtune=znver2 -mfma -mavx2 -m3dnow -fomit-frame-pointer
gfortran compiler	-O3 -march=znver2 -mtune=znver2 -mfma -mavx2 -m3dnow -fomit-frame-pointer

4.2.2.3. Intel compiler

The Intel compiler is developed and targeted for the Intel hardware and hence it has some minor issues when using it with AMD hardware.

Table 28. Suggested compiler flags for Intel compilers

Compiler	Suggested flags
Intel C compiler	-O3 -march=core-avx2 -fma -ftz -fomit-frame-pointer
Intel C++ compiler	-O3 -march=core-avx2 -fma -ftz -fomit-frame-pointer
Intel Fortran compiler	-O3 -march=core-avx2 -align array64byte -fma -ftz -fomit-frame-pointer

The flag "-march=core-avx2" is used to force the compiler to build AVX2 code using the AVX2 instructions available in EPYC. The generated assembly code does indeed contain AVX (AVX and AVX2) instructions which can be verified by searching for instructions that use the "ymm" registers. The documentation states about the "-

march" flag "generate code exclusively for a given <cpu>" It might not be totally safe to use this on none Intel processors.

AMD claims that the EPYC processor fully supports AVX2, so it should be safe to use "-xAVX". Using the "-xCORE-AVX2" can also be tried, but it might fail in some cases, see below. In addition this might change from version to version of the Intel compiler. The only sure way is testing it by trial and error. To illustrate this point, in some cases like the HPCG (an alternative top500 test) benchmark, the option "-march=broadwell" worked well, e.g. produced the best performing code.

Testing have shown that full vector/AVX support is possible, if the main() function is not compiled with full AVX/AVX2 support, e.g. using the flag "-xavx", "-xavx2" "-xcore-avx2". If the main is compiled with any of these the test is triggered and program will just abort with messages that the processor is not compatible. In addition the flag "-ipo" must be avoided, if this is omnipresent the test will also be triggered and program will abort.

Performance gain can be quite significant. The table below show a test with the reference implementation of matix matrix multiplication (N=5000). Only the *dgemm.f* file have been compiled with the different flags. The testing program is compiled with only "-O3".

Table 29. Performance effect of Vectorisation flags

Vectorisation flag	Single core performance
-O3	4.33 GFLOPS
-O3 -march=core-avx2	4.79 GFLOPS
-O3 -xavx	17.97 GFLOPS
-O3 -xavx2	26.39 GFLOPS
-O3 -xcore-avx2	26.38 GFLOPS

If on the other side the peak performance is not paramount the safe option would be to use the "-xHost" which generate instructions for the highest instruction set and processor. The flag "-axAVX" can be used which let the run time system performs checks at program launch to decide which code should be executed. On the other hand just omit any flag setting the processor type. Please verify performance as optimal code might not be generated.

When operating an a mixed GNU g++ and Intel C++ environment the flags controlling C++ standard are important. The flag "-std=gnu++98" is needed to build the HPCG benchmark and in other cases newer standards like "-gnu ++14" are needed.

4.2.2.4. PGI

Table 30. Suggested compiler flags for PGI compilers

Compiler	Suggested flags
PGI C compiler	-O3 -tp zen -Mvect=simd -Mcache_align -Mprefetch -Munroll
PGI C++ compiler	-O3 -tp zen -Mvect=simd -Mcache_align -Mprefetch -Munroll
PGI Fortran compiler	-O3 -tp zen -Mvect=simd -Mcache_align -Mprefetch -Munroll

PGI C++ uses gcc version to set the different C++ versions. The installed versions support C++17 and older. Online documentation is available [133].

Analysis of the generated code shows that using the SIMD option as suggested does generate 256 bits wide vector instructions and that the call for Zen architecture also triggers generation of 256 bits wide FMA and other vector instructions.

4.2.3. AMD Optimizing CPU Libraries (AOCL)

AOCL are a set of numerical libraries tuned specifically for AMD EPYC™ processor family. They have a simple interface to take advantage of the latest hardware innovations. The tuned implementations of industry standard math libraries enable fast development of scientific and high-performance computing projects.

Table 31. Numerical libraries by AMD

Library name	Source code	Functions
BLIS	Open	Basic Linear Algebra Subprograms (BLAS)
libFLAME	Open	LAPACK routines
FFTW [121]	Open	Collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases thereof
ScaLAPACK	Open	Library of high-performance linear algebra routines for parallel distributed memory machines
Random Number Generator Library (RNG)	Closed	Pseudorandom number generator library
Secure Random Number Generator (RNG)	Open	Library that provides APIs to access the cryptographically secure random numbers
libM	Closed	Math library (libm.so, sqrt, exp, sin etc)

For more detailed information about the AMD libraries please consult the AMD web pages about libraries [146].

4.2.4. Intel Math Kernel Library

While Intel MKL [35] is not fully supported on AMD by Intel for natural reasons it can nevertheless be used with AMD EPYC.

4.2.4.1. MKL with Intel compiler

When building code for Rome using the Intel compiler and the flags suggested in Table 28, “Suggested compiler flags for Intel compilers” the simple solution is to just include *-mkl* in order to link with the Math Kernel Library.

4.2.4.2. MKL with GNU compiler

When using the GNU compiler the situation is somewhat more complicated. The following lines from the HPL build script might serve as a guideline.

```
-L/opt/intel/mkl/lib/intel64 \
-Wl,--start-group \
/opt/intel/mkl/lib/intel64/libmkl_gf_lp64.a \
/opt/intel/mkl/lib/intel64/libmkl_gnu_thread.a \
/opt/intel/mkl/lib/intel64/libmkl_core.a \
-lmkl_avx2 -Wl,--end-group -lpthread -ldl -lm
```

Using a more simplified pure dynamic linking could look like this:

```
-L/opt/intel/mkl/lib/intel64 -lmkl_gnu_thread\
-lmkl_avx2 -lmkl_core -lmkl_rt
```

There are several combinations that work, it is bit trial and error. The web page "Intel® Math Kernel Library Link Line Advisor" [114] can offer some help.

As always with threaded libraries the OpenMP flag should be set both compile time and link time.

The support Math Kernel Library for AMD is limited.

4.2.5. Library performance

Comparing library performance is not trivial, there are literally hundreds of different functions. Only very few functions can be tested and only selected libraries can be evaluated. As most HPC computation involve Linear algebra and Fourier transforms those two have been selected for comparison. Numbers below are for a single node.

Table 32. Performance library performance

Benchmark test	AMD libraries (2.1)	Intel MKL (2018.5)
HPL (128 cores)	3400 GFLOPS	3411 GFLOPS
FFT 1d (62500 MiB) (Single core)	197.428 seconds	103.584 seconds

The Math Kernel Library is a very good option for AMD processors, even if not fully supported.

4.3. Benchmark performance

4.3.1. Stream - memory bandwidth benchmark

The STREAM benchmark [126] is widely used to demonstrate the system's memory bandwidth, see Section 2.3.1 for an introduction. In short the following operations are performed:

```
Copy: a(i) = b(i)
Scale: a(i) = q * b(i)
Sum: a(i) = b(i) + c(i)
Triad: a(i) = b(i) + q * c(i)
```

The table shows the highest bandwidth obtained using some of the highest performing processor binding. The figures show the need for thread core placement and binding.

Table 33. Compiling options used for STREAM benchmarking on AMD Rome

System and Compiler	Compiler Options
Betzy (Section 4.6.2), Intel icc (2018.5) incl. MKL	-Ofast -DN=2500000000 -DNTIMES=10 -DSTATIC -mcmodel=large -qopenmp -shared-intel -ffreestanding -qopt-streaming-stores always

The compilation flags used are a.o. selecting steaming stores which effectively bypass some cache layers and might not be beneficial for applications that reuse already fetched data. The streaming stores are beneficial when the data are not being reused, e.g. they will not fill up the caches unnecessarily.

The results below is obtained from a single compute node on the Betzy system (described in Section 4.6.2).

Table 34. STREAM results (numbers in MB/sec)

Binding	Threads	Copy	Scale	Add	Triad
OMP_PROC_BIND=true	128	322201	322214	324189	323997
KMP_AFFINITY=Scatter	128	322235	322344	324225	324220
OMP_PLACES=cores	128	322155	322100	324118	323948
OMP_PLACES="{0:16},{16:16},{32:16},{48:16},{64:16},{80:16},{96:16},{112:16}"	128	325624	327319	323229	327059
OMP_PLACES="{0:8:2},{16:8:2},{32:8:2},{48:8:2},	64	337715	337884	340005	340011

Binding	Threads	Copy	Scale	Add	Triad
{64:8:2},{80:8:2},{96:8:2}, {112:8:2}"					
OMP_PLACES="{0:4:2}, {16:4:2},{32:4:2},{48:4:2}, {64:4:2},{80:4:2},{96:4:2}, {112:4:2}"	32	322360	321852	313485	312295

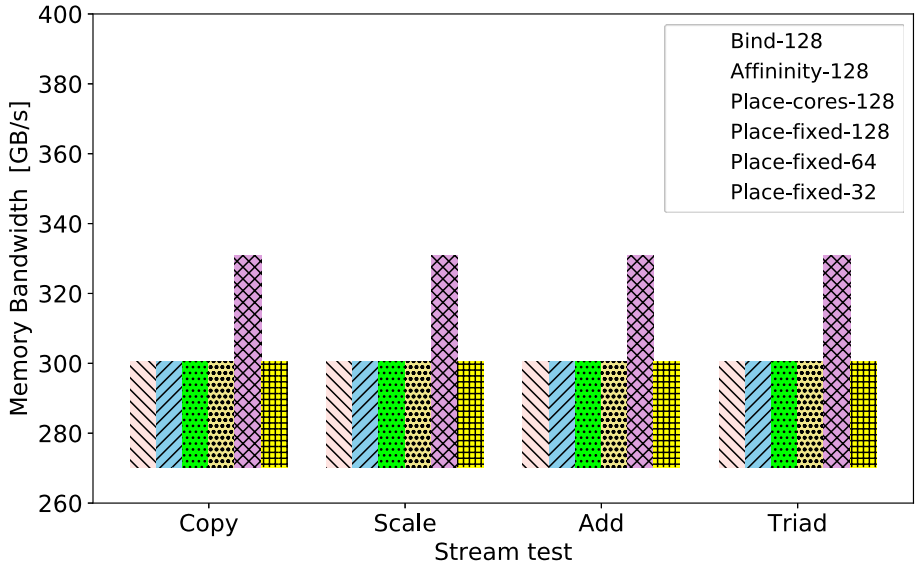


Figure 64. Betzy STREAM benchmark, please see Table 34 above for details of the different runs

Memory bandwidth in excess of 340 GiB/s is relatively good for compute nodes of this class. The importance of processor pinning clearly demonstrated by the performance of fixing threads on 64 cores, 32 cores per socket. Spreading the memory load optimally. No tests to assess the power usage vs. processor frequency have been performed with Betzy.

4.3.2. High Performance Linpack

The table below shows the results from running HPL on a single node on Betzy. The tests were run with hyper-threading, SMT on, yielding a total of 256 cores, two virtual cores or threads per core, (Section 4.1.1). Processor placement was used. The processor frequency was monitored and found be at it's nominal value at 2.25 GHz. The HPL was built using OpenMPI (4.0.2) and Intel compiler suite (2018.1.163) including MKL as multithreaded BLAS library.

Table 35. HPL performance numbers for single node in Betzy. All numbers in GFLOPS

Ranks / Threads	HPL parameters[a]	128 cores	256 cores (SMT-2)
8 / 16	N=160000 NB=192	3691.28	n.d.
16 / 8	N=160000 NB=192	3648.54	n.d.
16 / 16	N=160000 NB=192	n.d.	3276.79
32 / 8	N=160000 NB=192	n.d.	3209.83

[a]Minor parameters: WR00C3L3.

As might be expected, 128 cores gives the best performance results. All hardware cores are fully engaged and there is no contention on their resources from hyper-threads. The combination of number of ranks versus number of threads per rank for hybrid models are always a tradeoff. A bit of testing is always good. Scaling properties of the multithreaded library is always important.

4.4. Performance Analysis

There are several tools that can be used to do performance analysis. The eco-system of performance analysis tools spans over a wide range of frameworks for profiling, tracing and (most often) combination of both techniques. Below an outlook of the most wide-spread tools for the standard CPUs as well as tools for AMD processors is given, which can be used on the high-performance AMD systems.

4.4.1. perf (Linux utility)

The package perf is a simple easy to use performance monitoring and analysis tool for Linux. Perf is based on the *perf_events* system, which is based on event-based sampling, and it uses CPU performance counters to profile the application. It can instrument hardware counters, static tracepoints, and dynamic tracepoints. It also provide per task, per CPU and per-workload counters, sampling on top of these and source code event annotation. It does *not* instrument the code, so that it has a really fast speed and generates precise results [142].

You can use *perf* to profile with *perf record* and *perf report* commands:

```
1 perf record -g <app> <options>
2 perf report
```

The *perf record* command collects samples and generates an output file called *perf.data*. To get the call graph, we pass the *-g* option to *perf record*. The *perf.data* file can then be analyzed using *perf report* and *perf annotate* commands.

Below is an example of using the perf tool for the Conjugate Gradient (CG) from the NAS Parallel benchmark with the OpenMP programming model (NPB-OpenMP). The CG application of the C class with the AMD Optimizing C/C++ Compiler (AOCC) was compiled and run on one AMD Rome compute node with 4 OpenMP threads.

```
$ cd ~/NPB3.4/NPB3.4-OMP/
$ perf record ./bin/cg.C.x
$ perf report

# To display the perf.data header info, please use --header/--header-only
# options.
#
# Total Lost Samples: 0
#
# Samples: 576K of event 'cycles:u'
# Event count (approx.): 479959862103
#
# Overhead  Command  Shared Object       Symbol
# ........  .......  ..................  ................................
#
    81.92%  cg.C.x   cg.C.x              [.] MAIN__5F1L462_
     6.16%  cg.C.x   cg.C.x              [.] sparse_
     5.76%  cg.C.x   libomp.so           [.] __kmp_hardware_timestamp
     3.58%  cg.C.x   libomp.so           [.] __kmp_hyper_barrier_release
     1.82%  cg.C.x   libomp.so           [.] __kmp_barrier
     0.09%  cg.C.x   cg.C.x              [.] randlc_
     0.06%  cg.C.x   cg.C.x              [.] sprnvc_
     0.04%  cg.C.x   libc-2.28.so        [.] __memmove_avx_unaligned_erms
     0.03%  cg.C.x   [unknown]           [k] 0xffffffffa9c01c20
     0.03%  cg.C.x   [unknown]           [k] 0xffffffffa93390b7
     0.02%  cg.C.x   [unknown]           [k] 0xffffffffa9255c7b
     0.02%  cg.C.x   [unknown]           [k] 0xffffffffa9226c6b
```

```
    0.02%  cg.C.x   libc-2.28.so        [.] __sched_yield
    0.02%  cg.C.x   [unknown]           [k] 0xffffffffa9c00010
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa9a55d59
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa9c0001c
    0.01%  cg.C.x   libomp.so           [.] __kmp_yield
    0.01%  cg.C.x   cg.C.x              [.] MAIN__4F1L306_
    0.01%  cg.C.x   libc-2.28.so        [.] __memset_avx2_unaligned_erms
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa92e61e3
    0.01%  cg.C.x   cg.C.x              [.] MAIN__1F1L167_
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa92e568c
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa92e9690
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa9a68800
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa92f17d7
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa92df320
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa9c00013
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa92f1450
    0.01%  cg.C.x   [unknown]           [k] 0xffffffffa9226b15
...
```

Perf is based on the perf_events interface exported by recent versions of the Linux kernel. More information is available in the form of a manual [143] and tutorial [144].

Newer Linux kernels have a sysfs tunable */proc/sys/kernel/perf_event_paranoid* which allows the user to adjust the available functionality of *perf_events* for non-root users, with higher numbers being more secure (offering correspondingly less functionality):

- -1 - not paranoid at all;
- 0 - disallow raw tracepoint access for unprivileged users;
- 1 - disallow CPU events for unprivileged users;
- 2 - disallow kernel profiling for unprivileged users.

More clearly the Linux *perf_event_paranoid* setting is shown in Figure 65.

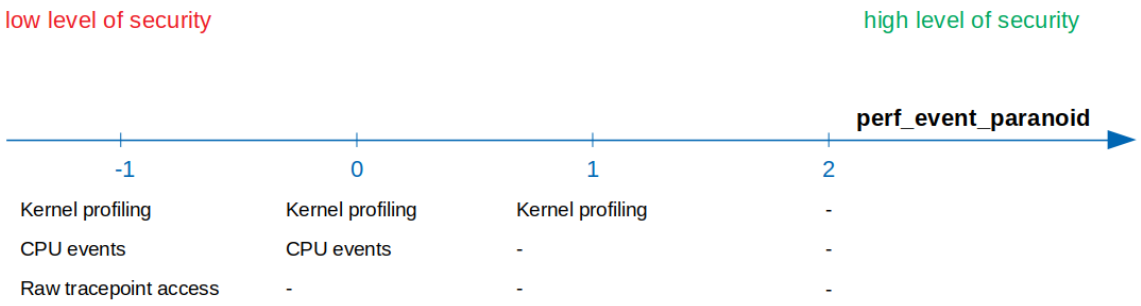


Figure 65. Linux perf_event_paranoid setting

4.4.2. perfcatch

The perfcatch utility provides a performance profile for an MPI program or SHMEM program. This profiling tool gathers statistics that show information such as the following:

- the amount of time that your program spends in MPI;
- the number of messages sent;
- the number of bytes sent.

For example, the perfcatch utility runs an MPI program with a wrapper profiling library (libmpi.so) that prints MPI call profiling information to a summary file upon MPI program completion.

The following format shows how to use the *perfcatch* command:

```
mpiexec_mpt [ mpi_params ] perfcatch [ -i ] cmd [ args ]
```

By default, *perfcatch* assumes an MPT program. The perfcatch utility accepts the following arguments.

- The *mpi_params* argument specifies the MPI parameters used to launch the program (optional).
- The *-i* argument specifies to use Intel MPI.
- The *cmd* argument specifies the name of the executable program.
- The *args* argument specifies additional command-line arguments (optional).

To use *perfcatch* with an MPI from HPE program, insert the *perfcatch* command in front of the executable file name, as the following examples show:

```
mpiexec_mpt -np 64 perfcatch a.out
```

To use *perfcatch* with Intel MPI, add the *-i* option, as follows:

```
mpiexec_mpt -np 64 perfcatch -i a.out
```

The *perfcatch* utility output file is called *MPI_PROFILING_STATS*. Upon program completion, the *MPI_PROFILING_STATS* file resides in the current working directory of the MPI process with rank 0.

This output file includes a summary statistics section followed by a rank-by-rank profiling information section. The summary statistics section reports some overall statistics. These statistics include the percent time each rank spent in MPI functions and the MPI process that spent the least and the most time in MPI functions. Similar reports are made about system time usage.

In the rank-by-rank profiling information, there is a list of every profiled MPI function called by a particular MPI process. The report includes the number of calls and the total time consumed by these calls. Some functions report additional information, such as average data counts and communication peer lists [105].

4.4.3. AMD µProf

The AMD µProf is a suite of powerful tools to perform CPU and Power analysis of applications running on Windows and Linux operating systems on AMD processors. It allows developers to better understand the runtime performance of their application and to identify ways to improve its performance. The AMD µProf has a graphical user interface (GUI). Additional information can be found at the AMD web pages [104].

A workflow of the AMD µProf profiler consists of the following steps:

- creating a profile configuration;
- creating the profile;
- analyzing the profile data.

The AMD µProf profiler follows a statistical sampling-based approach to collect profile data to identify the potential performance bottlenecks. The profile data collection can be triggered by – 6OS timer, core PMC events and IBS. AMD µProf offers user various UI sections:

- SUMMARY (hotspots);

- ANALYZE (profile data at various granularities);
- SOURCES (the data at source line and assembly level);
- TIMECHART (system metrics).

As an example of using the AMD µProf profiler, the Scalar Penta-diagonal solver (SP) from the NAS Parallel Benchmark with the OpenMP programming model (NPB-OpenMP) was taken. The SP application of the C class with the AMD Optimizing C/C++ Compiler (AOCC) was compiled and run on one AMD Rome compute node with 128 OpenMP threads. Figure 66 shows the Hot Spots Window in SUMMARY page for the "Not halted cycle" metric.

Figure 67 shows data grouped by process in ANALYZE page. In this case, the data is grouped by the OpenMP threads.

Figure 66. AMD µProf: the SUMMARY section for the SP application on 128 OpenMP threads

Process	Misalign loads	Not halted cycle	Ret inst	DC miss	Data cache refill	DTLB L1M
sp.C.x (PID 238554)	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Load Modules						
Threads						
Thread-238560 (TID 238560)	1.10%	1.01%	1.02%	1.03%	1.04%	1.06%
Thread-238682 (TID 238682)	1.09%	1.01%	0.98%	1.07%	1.04%	1.26%
Thread-238642 (TID 238642)	1.08%	1.01%	0.97%	1.06%	1.00%	0.98%
Thread-238659 (TID 238659)	1.08%	1.00%	0.91%	1.09%	1.02%	0.97%
Thread-238591 (TID 238591)	1.08%	1.00%	1.06%	1.05%	1.04%	0.99%
Thread-238580 (TID 238580)	1.07%	1.00%	0.99%	1.11%	1.06%	1.00%
Thread-238656 (TID 238656)	1.07%	1.00%	1.05%	1.01%	1.02%	0.97%
Thread-238563 (TID 238563)	1.07%	1.01%	0.95%	1.08%	1.02%	0.99%
Thread-238630 (TID 238630)	1.06%	1.01%	0.95%	1.01%	0.99%	0.96%
Thread-238588 (TID 238588)	1.06%	1.01%	0.94%	1.09%	1.03%	0.98%
Thread-238640 (TID 238640)	1.06%	1.00%	1.02%	1.06%	1.05%	0.98%
Thread-238619 (TID 238619)	1.06%	1.00%	1.06%	0.93%	0.97%	0.96%
Thread-238676 (TID 238676)	1.05%	1.01%	0.95%	1.03%	0.99%	0.98%
Thread-238670 (TID 238670)	1.05%	1.00%	1.00%	1.01%	1.01%	0.99%
Thread-238660 (TID 238660)	1.05%	1.01%	1.01%	1.04%	1.00%	0.96%
Thread-238649 (TID 238649)	1.05%	1.01%	0.99%	1.02%	1.01%	0.96%
Thread-238646 (TID 238646)	1.05%	1.01%	0.92%	1.07%	0.99%	0.96%
Thread-238653 (TID 238653)	1.05%	1.00%	1.04%	1.00%	1.01%	0.96%
Thread-238681 (TID 238681)	1.05%	1.01%	1.02%	0.98%	0.98%	1.16%

Search : Type function name...

Functions (for sp.C.x (PID 238554))	Misalign loads	Not halted cycle	Ret inst	DC miss	Data cache refill	DTLB L1M
compute_rhs__1F1L18_	58.65%	42.60%	36.89%	53.11%	38.85%	1.14%
z_solve__1F1L34_	12.64%	6.86%	7.83%	13.14%	40.06%	95.41%
y_solve__1F1L30_	11.98%	3.05%	3.44%	13.42%	13.55%	1.93%
x_solve__1F1L30_	9.41%	1.53%	1.99%	5.78%	2.99%	0.09%
add__1F1L20_	6.73%	1.48%	0.56%	2.95%	1.85%	0.01%
exact_solution_	0.31%	0.05%	0.06%	0.03%	0.01%	0.08%
exact_rhs__1F1L21_	0.07%	0.04%	0.04%	0.08%	0.09%	0.41%
__kmpc_threadprivate_cached	0.04%	0.01%	0.01%	0.01%	0.01%	0.02%
initialize__1F1L23_	0.03%	0.01%	0.01%	0.01%	0.00%	
__kmp_get_global_thread_id_reg	0.02%	0.00%	0.01%	0.00%	0.00%	0.01%
__kmpc_for_static_fini	0.02%	0.01%	0.01%	0.01%	0.01%	0.01%
__tls_get_addr	0.01%	0.00%	0.00%	0.00%	0.00%	0.01%
__kmpc_master	0.01%	0.00%	0.00%	0.00%	0.00%	0.01%
[PLT] __kmpc_for_static_fini	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%
error_norm__1F1L25_	0.01%	0.00%	0.00%	0.00%		
__kmp_finish_implicit_task	0.01%	0.00%	0.00%	0.01%	0.01%	0.02%
error_norm__2F1L70_	0.01%	0.00%		0.00%	0.00%	
__kmpc_for_static_init_8	0.00%	0.01%	0.00%	0.00%	0.01%	0.03%
__kmp_invoke_task_func	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
__kmp_invoke_microtask	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%

Figure 67. AMD µProf: the ANALYZE section for the SP application on 128 OpenMP threads

The AMD µProf supports different types of profiling. For this, the Linux kernel must have the corresponding value of the variable *perf_event_paranoid*. This variable from a sysfs tunable */proc/sys/kernel/perf_event_paranoid* can take the values shown in Figure 65. Applicable to the AMD µProf profiler this means the following:

- Support Event-Based Profile (EBP) of process(es), when the *perf_event_paranoid* is set to <= 2.
- Support Time-Based Profile (TBP) of process(es), when the *perf_event_paranoid* is set to <= 1.
- Support Instruction-Based Sampling (IBS) of process(es), when the *perf_event_paranoid* is set to <= 0.
- Support profiling of all the processes running in the system, when *perf_event_paranoid* is set to <= 0.
- Support EBP with multiplexing, when *perf_event_paranoid* is set to "-1".

Setting *perf_event_paranoid* to "-1", will support all the profile types [106].

More clearly the Linux *perf_event_paranoid* setting is shown in Figure 68.

Figure 68. Linux *perf_event_paranoid* setting for AMD µProf

A full description of all the AMD µProf features is on-line available in the user guide [145].

4.4.4. Roof line model

Roof line model [96] analysis well known and widespead intuitive visual performance model. The most striking feature for HPC is it's ability to providing a visual presentation memory bound applications. As most large scale scientific applications are constrained by memory access the roof-line mode is a convenient method to very that this is the case for the application under investigation. Running on AMD processors somewhat limits the number of available detailed metrics.

One way of recording a root line model of an application is to use the Intel tool Advisor [36]. Running the application using this tool can provide profiling, vectorisation etc, in addition to generating a roof line model of the application.

Profiling and record run time data for high core count using the GUI is impractical. Recording is most convenient done using the command line under the control of a batch queue system. The following examples assume OpenMPI as the MPI environment, and SLURM as the batch queue system.

```
EXE=a.out
advdir=$(dirname $(which advixe-cl))
projdir=$HOME/${EXE}.run/
mpirun -np 1 $advdir/advixe-cl -project-dir $projdir -collect survey \
-flop -- ./$EXE : -np $((SLURM_NTASKS - 1)) ./$EXE
```

This run only collect an overview called survey. In order to generate a roof line model one more run is needed. This second run must be run with the keyword *tripcounts* to collect tripcounts.

```
EXE=a.out
advdir=$(dirname $(which advixe-cl))
projdir=$HOME/${EXE}.run/
mpirun -np 1 $advdir/advixe-cl -project-dir $projdir -collect tripcounts \
-flop -- ./$EXE : -np $((SLURM_NTASKS - 1)) ./$EXE
```

Only the first rank is profiled in this example. Any rank or any number of ranks be selected using different numbers for *np* the number of processes (ranks) started, see OpenMPI documentation [9] (mpirun) for more info.

After securing the the data the GUI can display a.o. a roof line model. The plot below show an example of this, the orange circles are vector SIMD instructions while the blue circles are scalar instructions.

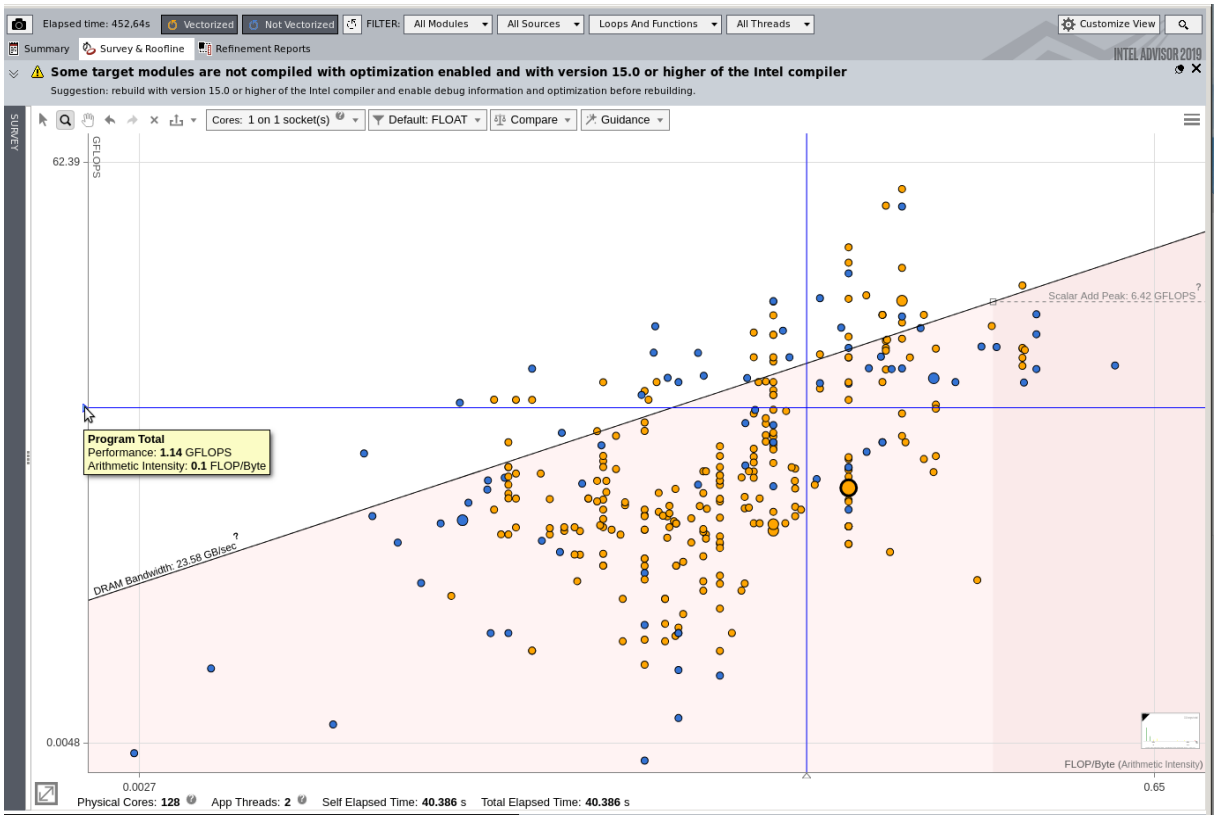


Figure 69. Example of a roof line model for a single run of an application

The GUI can also compare several runs, which can be beneficial during an optimisation process. The figure below show a run with small degree (squares) of vectorisation compared to high degree of vectorisation (circles).

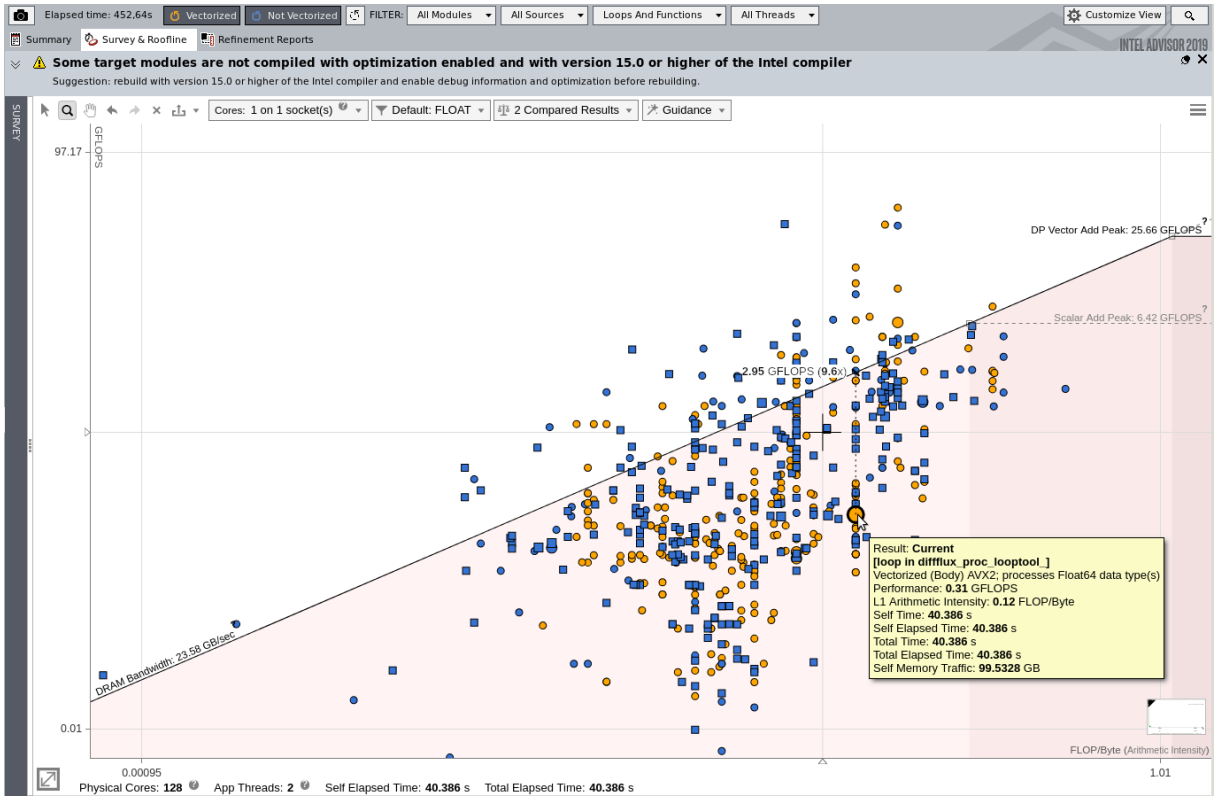


Figure 70. Example of a roof line model comparing two different builds of an application

The colour are as before (orange is vector code, blue scalar instructions) while the squares are the code with low degree of vectorisation and the circles represent a code with high degree of vectorisation.

As the Advisor also count the floating point operations it provide a handle on the total utilisation of the processor. As we know the theoretical maximum we can estimate a number for the efficiency. The theoretical number for a single core is about 40 Gflops/s when doing 64 bits numbers. The Advisor show for the "Program total" (which for this run is just one rank) 1.14 Gflops/s. Comparing this to the theoretical performance of one core yield an efficiency of just about 3%. Most of the processor capacity is wasted waiting for memory, e.g. like most scientific applications this one is also memory bound. The metric flops per byte called artitmetric intensity is particularly important. For the example here it's close to 0.1 flops/byte which clearly puts the application in the memory bound region. More background is found in [119] and [120].

4.5. Tuning

Tuning is a multidimensional problem, in addition the dimensions are not independent. Someone once described it as balloon squeezing, aka the balloon effect [113].

4.5.1. Introduction

Most of the tuning used to tune applications and programs for Skylake also apply to the AMD EPYC processors, please review Section 3.5. Most of the Intel tools run fine on AMD EPYC. There is also a comprehensive tuning chapter in the earlier version of the AMD EPYC guide [5]. This guide is written for the EPYC processor and contain specific options for the EPYC processor variant.

4.5.2. Intel MKL pre 2020 version

The Intel Math Kernel Library tries to figure out the processor it's being run on. In the case of AMD EPYC it does not get the correct answer for obvious reasons. There is a flag to tell MKL what kind of processor it being used on, setting this to an appropriate value helps MKL to provide the best possible performance on EPYC Rome.

```
export MKL_DEBUG_CPU_TYPE=5
```

The table below illustrates the effect when using a pure MPI HPL run using a serial MKL BLAS library.

Table 36. Performance using Intel MKL with different processor settings

Environment Flag	HPL performance [GFLOPS]
unset MKL_DEBUG_CPU_TYPE	843.16
MKL_DEBUG_CPU_TYPE=5	3217.5

Test with the Intel 2020.1.217 version of the compiler with its associated MKL library no longer honor this flag. While this the case with the version tested, it is not known how widespread this is. Users are advised to run their own tests.

4.5.3. Intel MKL 2020 version

The MKL version 2020 (or probably newer) will need a different approach than the one above. MKL issue a run time test to check for genuine Intel processor. If this test fail it will select a generic x86-64 set of routines yielding inferior performance. This is well documentated in [115] and a simple fix is suggested in [116].

Research have discovered that MKL call a function called *mkl_serv_intel_cpu_true()* to check the current CPU. If a genuine Intel processor is found it simply return 1. The solution is simply to override this function by writing a dummy functions which always return 1 and place this first in the search path. The function is simply:

```
int mkl_serv_intel_cpu_true() {
```

```
 return 1;
}
```

Compling this into a shared library using the following command:

```
gcc -shared -fPIC -o libisintel.so isintel.c
```

To put the new shared library first in the search path we can use a preload environment vaiable:

```
export LD_PRELOAD=<path to lib>/libisintel.so
```

In addition the envionment variable MKL_ENABLE_INSTRUCTIONS can also have a significant effect. Setting the variable to AVX2 is adviced. Just changing it to AVX have a significant negative impact. The table below illustrate the effect when testing with HPL.

Table 37. Effect on various settings on MKL performance

Settings	Performance [Tflops/s]
None	1.2858
LD_PRELOAD=./libisintel.so	2.7865
LD_PRELOAD=./libisintel.so MKL_ENABLE_INSTRUCTIONS=AVX	2.0902
LD_PRELOAD=./libisintel.so MKL_ENABLE_INSTRUCTIONS=AVX2	2.7946

These traits of MKL is subject to change at any new release, the above settings might not work in newer releases.

4.5.4. Memory bandwidth per core

With the steadily increasing core count which vastly outgrow the modest growth in memory bandwidth the memory bandwidth per core is shrinking. This has major implications for systems with processors having a very high core count.

When running a memory bandwidth bound application on a system equipped with processors with very high core count the performance is limited by the relatively low memory bandwidth per core. In selected cases experience have shown that by using only every second core the performance is higher.

Applying models like the roof line model [96] help to understand and possible mitigate the memory bandwidth problem. The roof line section Section 4.4.4, “Roof line model” discuss the processor efficiency.

The figure below show the effect of memory bandwidth when running 4096 ranks spread over a different number of nodes. For the run using 32 nodes with 128 ranks were scheduled on 128 cores per node, while for 64 nodes 64 ranks per node were scheduled. Using twice as many nodes should yield 2x performance likewise for four times as many nodes. However, when normalised for the number of nodes we see that the efficiency rise more than expected, using 2 twice as many nodes yield more than a twofold increase in performance. But we experience a rise in total performance from 866 to 2190 which is 2.53x. Getting a speedup of more than 2 when using 2 times as many nodes is beneficial.

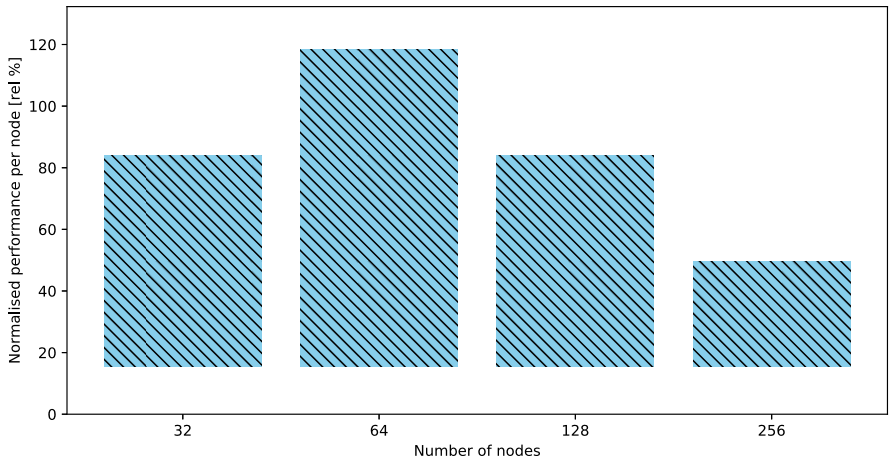


Figure 71. Performance effect on memory bandwidth by runnning Bifrost application at 4096 ranks using a varying number of nodes

The following table presenting performance numbers help to illustrate the effect.

Table 38. Bifrost performance using different number of nodes to run 4096 ranks

Nodes	Ranks per node	Perf [Mz/s]	Perf per node	Rel. percent
32	128	8.66E+02	27.1	100%
64	64	2.19E+03	34.2	126%
128	32	3.86E+03	30.2	111%
256	16	5.46E+03	21.3	79%

Mapping and binding of MPI ranks on each node is important in order to map the ranks on the cores and NUMA banks. An example of 64 ranks per node using OpenMPI is shown here:

```
mpirun --map-by ppr:32:socket:pe=2  --bind-to core  ./a.out
```

Mapping is done by using 32 ranks per socket and a total of 2 sockets per node. For more information about details of mpirun mapping and binding see [10].

There might be other effects than just the memory bandwidth per rank that make an impact of the performance. The fact that less ranks compete for the resources like caches, buffers, interconnect etc are also likely to have an impact. Again only testing, trial-and-error can locate an optimum.

4.6. European AMD processor based systems

4.6.1. HAWK system (HLRS)

In 2018 the High-Performance Computing Center of the University of Stuttgart (HLRS) and Hewlett Packard Enterprise (HPE) have launched a joint cooperation to build and deliver for HLRS a next-generation supercomputer. The new system called Hawk is the flagship supercomputer of the HLRS. At time of installation (Q1 2020), it was among the fastest high-performance computers in the world and the fastest general-purpose machine for industrial production in Europe.

Designed with the needs of HLRS's users in mind, Hawk is optimised for engineering simulation applications. In concert with other platforms at HLRS for high-performance data analytics and artificial intelligence, Hawk

enables the support of new kinds of workflows that combine data generation, simulation, big data analysis, deep learning, and other data science methods [103].

Hawk, based on HPE's next-generation HPC platform running a next generation AMD EPYC™ processor code named Rome, has a system peak performance of 26 PFLOPS, and consist of a 5,632-node cluster [102]. The compute nodes are interlinked by a high speed InfiniBand HDR (200 Gbit/s) network.

4.6.2. Betzy system (Sigma2)

Betzy is a Norwegian AMD Rome based Bull Sequana XH2000 system supplied by Atos, installed at NTNU in Trondheim, owned and operated by Sigma2. It will provide computational cycles to the Norwegian general academic community with a broad range of disciplines and applications.

The system comprise 1344 dual socket nodes in 14 racks combined into five cells with a total of 172032 compute cores. A gross peak performance of about 6 PFLOPs. All nodes are interconnected by Mellanox InfiniBand. The five cells are set up in a Dragonfly topology. Cluster storage is supplied by DDN with a capacity about 2 PBytes. I/O bandwidth is in excess of 50 GB/second.

Power consumption is about 1 MW with a 95% heat capture to water. Water inlet temperature is about 18°C and outlet temperature is in the range of 25°C to 28°C.

The systems runs CentOS 7 operating system with SLURM as the resource management and scheduling system. Storage is based on the LUSTRE file system.

Figure 72. A blade with 3 dual socket nodes

The Figure 72 shows a single blade holding three dual processor compute nodes. Only the first processor of each node has the water cooling installed, making it possible to view the processors and memory DIMMs.

Each compute node is equipped with:

- Two AMD Rome 7742 processors, 64 cores at 2.25 GHz, for a total of 128 cores per node
- Processor cache: L1 cache 4 MiB, L2 cache 32 MiB, L3 Cache 256 MiB
- Processor vector units: up to 4 (using FMA) x AVX-256 (16 double precision FLOPS) per clock cycle, e.g. 36 GFLOPS per core
- 256 GiB main memory, 2 GiB per core, 16 x 16 GiB DIMMS, DDR4 3200 MT/s
- PCIe 4.0 (16 GT/s, 1.969 GB/s per lane)
- Mellanox HDR-100 InfiniBand, 100 Gbits/s
- Ethernet 10 Gbits/s and 1 Gbits/s

Each compute node has a theoretical double precision performance of 4.6 TFLOPS. Aggregated performance for all the 1344 nodes is 6.19 PFLOPS.

Table 39. Power usage running HPL

HPL Performance	Nodes / Cores	Processor Frequency	Power usage
4.71709 PFLOPS	1328 /169984	2.25 GHz	899.6 kW

Power per GFLOPS comes out at 5.2 GFLOPS/W.

System is being installed during spring of 2020 and went into production early October 2020.

A. Acronyms and Abbreviations

1. Units

n	S.I. Unit nano = 10^{-9}
k	S.I. Unit kilo = 10^{3}
M	S.I. Unit Mega = 10^{6}
G	S.I. Unit Giga = 10^{9}
T	S.I. Unit Tera = 10^{12}
ki	ISO/IEC 8000 Unit kibi = 2^{10}
Mi	ISO/IEC 8000 Unit Mebi = 2^{20}
Gi	ISO/IEC 8000 Unit Gibi = 2^{30}
Ti	ISO/IEC 8000 Unit Tibi = 2^{40}
Byte	8 bits, (an octet)
kiB	2^{10} Bytes
MiB	2^{20} Bytes
GiB	2^{30} Bytes
TiB	2^{40} Bytes
GHz	Giga Hertz, Hertz is frequency, events per second.
MHz	Mega Hertz, Hertz is frequency, events per second.
nm	Nano meter.
MT	Mega Transactions, millions of transactions per second, often used for memory transactions and network (InfiniBand) transactions
W	Watt, units of energy per second, Joules/second

2. Acronyms

ALU	Arithmetic Logical Units, one of the executional units within the processor (CPU)
AVX	Advanced Vector Instructions, a set of extra vector (SIMD) instructions added to the SSE vector instruction set. There is also a an extra addition, AVX2
CPU	Central Processing Unit, refer to a single core or the complete package to fit in a socket, ambiguous
AGU	Address Generating Unit
BLAS	Basic Linear Algebra Subprograms (comes in 3 levels, level 1: scalar vector, level 2: Vector matrix, level 3: Matrix matrix operations) [11]
BTB	Branch Target Buffer, Branch prediction buffer

CCD	Compute Core Die. For example, an AMD CPU with 8 cores will have a CCD with two 4-core CCXs
CCX	Compute Core compleX (AMD term) a group of four CPU cores and their CPU caches, CCX refers to a four-core grouping inside of each core chiplet die (CCD)
DDR4	Double Data Rate 4 Synchronous Dynamic Random-Access Memory
DIMM	Dual In line Memory Module, variants RDIMM (registered ECC), LRDIMM (Load Reduced DIMM)
DVFS	Dynamic Voltage and Frequency Scaling
BPG	Best Practice Guide
BSC	Barcelona Supercomputing Center
EPCC	Edinburgh Parallel Computing Centre
EPI	European Processor Initiative
FFTW	Fastest Fourier Transformation in the West [121]
FLOPS	Floating Point Operation per Second
FPP	In LUSTRE file context : one File Per Process (one file per MPI rank)
GMI	Global Memory Interconnect
HPC	High Performance Computing
HLRS	High Performance Computing Center Stuttgart
IPP	Intel Performance Primitives, a set of low level functions [34]
HDR	Mellanox InfiniBand High Data Rate, 200 Gbits/s. HDR-100 is a split of the 4x IB into two 2x each with 100 Gbits/s capacity
IB	InfiniBand
LAPACK	Linear algebra package, higher level linear algebra subroutines, [12]
LLC	Last Level Cache, often meaning Level 3 cache
LLVM	Original "Low Level Virtual Machine", this term is no longer applicable. It's presently a compiler project
LRZ	Leibniz Supercomputing Centre
MKL	Math Kernel Library, Intel's performance mathematical library [35]
MPI	Message Passing Interace [8]
NUMA	None-Uniform Memory Access
NVDIMM	None Volatile DIMM, memory that retains its contents even when electrical power is removed
PGI	Formerly Portland Group Inc. Presently part of NVIDIA, products are now known as PGI compilers
OpenMP	Open Multi-Processing [6]

RAS	Return Address Stack. The term is also used in memory context, meaning Row Address Strobe
RDMA	Remote Direct Memory Access (RDMA) [77]
RoCE	Remote direct memory access (RDMA) over converged Ethernet [78]
SCALAPACK	Scaleable Linear Algebra Package [13]
SF	In LUSTRE context, Single File, one shared common file for all MPI ranks
SIMD	Single Instructions Multiple Data
SLURM	Simple Linux Utility for Resource Management
SMT	Simultaneous Multi Threading [75]
SVE	Scalable Vector Extension [73]
TAGE	TAgged GEometric length predictor, tagged branch predictor
TDP	Thermal Design Power
TLB	Translation Lookaside Buffer, a cache for translation from virtual addresses to physical addresses

Further documentation

[1] PRACE Best Practice guides [https://prace-ri.eu/training-support/best-practice-guides/].

[2] Procurement support . [https://prace-ri.eu/training-support/technical-documentation/white-papers/procurement-commissioning-of/].

[3] Unified European Application Benchmark Suite (UEABS) . [https://prace-ri.eu/training-support/technical-documentation/benchmark-suites/].

[4] Best Practice Guide - ARM64, February 2019, https://prace-ri.eu/infrastructure-support/best-practice-guide-arm64/.

[5] Best Practice Guide - AMD, February 2019, https://prace-ri.eu/infrastructure-support/best-practice-guide-amd-epyc/.

[6] OpenMP, OpenMP reference. [https://en.wikipedia.org/wiki/OpenMP].

[7] OpenMP, Advaned OpenMP tutorial. [https://openmpcon.org/wp-content/uploads/openmpcon2017/Tutorial2-Advanced_OpenMP.pdf].

[8] MPI, MPI reference. [https://en.wikipedia.org/wiki/Message_Passing_Interface].

[9] OpenMPI documentaion, [https://www.open-mpi.org/doc/current/].

[10] OpenMPI documentaion, mpirun [https://www.open-mpi.org/doc/current/man1/mpirun.1.php#sect12].

[11] BLAS, Basic Linear Algebra Subprograms, reference. [https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms].

[12] LAPACK, Linear Algebra PACKage, reference. [https://en.wikipedia.org/wiki/LAPACK].

[13] ScaLAPACK, Scalable Linear Algebra PACKage, reference. [https://en.wikipedia.org/wiki/ScaLAPACK].

[14] Software Intel, Application Snapshot User Guide Metrics Reference. [https://software.intel.com/en-us/application-snapshot-user-guide-metrics-reference].

[15] Software Intel, application snapshot user guide function summary for all ranks. [https://software.intel.com/en-us/application-snapshot-user-guide-function-summary-for-all-ranks].

[16] Software Intel, application snapshot user guide collective operations time per rank. [https://software.intel.com/en-us/application-snapshot-user-guide-collective-operations-time-per-rank].

[17] Software Intel, application snapshot user guide message sizes summary for all ranks. [https://software.intel.com/en-us/application-snapshot-user-guide-message-sizes-summary-for-all-ranks].

[18] Software Intel, application snapshot user guide data transfers per rank to rank communication. [https://software.intel.com/en-us/application-snapshot-user-guide-data-transfers-per-rank-to-rank-communication].

[19] Software Intel, application snapshot user guide data transfers per rank. [https://software.intel.com/en-us/application-snapshot-user-guide-data-transfers-per-rank].

[20] Software Intel, application snapshot user guide data transfers per function. [https://software.intel.com/en-us/application-snapshot-user-guide-data-transfers-per-function].

[21] Software Intel, application snapshot user guide node to node data transfers. [https://software.intel.com/en-us/application-snapshot-user-guide-node-to-node-data-transfers].

[22] Software Intel, application snapshot userguide mpi time per rank. [https://software.intel.com/en-us/application-snapshot-user-guide-mpi-time-per-rank].

[23] Software Intel, application snapshot user guide communicator list. [https://software.intel.com/en-us/application-snapshot-user-guide-communicator-list].

[24] Software Intel, application snapshot user guide detailed information for specific ranks. [https://software.intel.com/en-us/application-snapshot-user-guide-detailed-information-for-specific-ranks].

[25] Software Intel, application snapshot user guide filtering by key metric. [https://software.intel.com/en-us/application-snapshot-user-guide-filtering-by-key-metric].

[26] Software Intel, application snapshot user guide filtering by number of lines. [https://software.intel.com/en-us/application-snapshot-user-guide-filtering-by-number-of-lines].

[27] Software Intel, application snapshot user guide using filter combinations. [https://software.intel.com/en-us/application-snapshot-user-guide-using-filter-combinations].

[28] Software Intel, Intel Parallel Studio. [https://software.intel.com/en-us/parallel-studio-xe].

[29] Software Intel, get started with application performance snapshot. [https://software.intel.com/en-us/get-started-with-application-performance-snapshot].

[30] Software Intel, application snapshot user guide metrics reference. [https://software.intel.com/en-us/application-snapshot-user-guide-metrics-reference].

[31] Software Intel, application snapshot user guide controlling amount of collected data. [https://software.intel.com/en-us/application-snapshot-user-guide-controlling-amount-of-collected-data].

[32] Software Intel, MPI Developer Reference Windows Interoperability with OpenMP. [https://software.intel.com/en-us/mpi-developer-reference-windows-interoperability-with-openmp].

[33] Software Intel, TuningGuide IntelXeonProcessor ScalableFamily 1stGen. [https://software.intel.com/sites/default/files/managed/04/59/TuningGuide_IntelXeonProcessor_ScalableFamily_1stGen.pdf].

[34] Software Intel, Intel® Integrated Performance Primitives (Intel® IPP). [https://software.intel.com/content/www/us/en/develop/tools/integrated-performance-primitives.html].

[35] Software Intel, Intel® Math Kernel Library (Intel® MKL) [https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html].

[36] Software Intel, Intel Advisor [https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-advisor/top.html].

[37] Scalasca, Scalasca. [https://www.scalasca.org/].

[38] Scalasca, Scalasca Manual. [https://apps.fz-juelich.de/scalasca/releases/scalasca/2.3/docs/@manual/start.html].

[39] Arm Forge, Arm DDT. [https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-ddt].

[40] Arm Forge, Arm Forge. [https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge?_ga=2.57528618.1292296101.1586412527-1107261253.1586204050].

[41] Arm Forge, Arm Map. [https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-map].

[42] Arm Forge, Arm Performance Reports. [https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-performance-reports].

[43] PAPI, PAPI. [http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview].

[44] VTune Profiler VTune Profiler / Amplifier. [https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html].

[45] Scalasca, Scalasca. [https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/docs/manual/start.html] .

[46] PRACE Webpage, http://www.prace-ri.eu/.

[47] Zen acrhitecture, Wikichip on Zen [https://en.wikichip.org/wiki/amd/microarchitectur es/zen].

[48] PCIe, PCIe [https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/3].

[49] AMD processors, AMD chips [http://xstra.u-strasbg.fr/lib/exe/fetch.php?media=doc:2019-09-17-amd-xstra-rome-training-sept2019-v2.pdf/].

[50] Memory, AMD Memory [http://www.me.ntu.edu.tw/ssdl/get_file.php?file_name=5f0d75313c10e.pdffile_dir=data263/rename=VLSI%202020_sc2.1.pdf].

[51] AMD EPYC, AMD with WIO [https://www.servethehome.com/amd-epyc-7702p-review-redefining-possible-at-64c-per-socket/].

[52] Amazon Web Services, Gravitron, [https://aws.amazon.com/ec2/graviton/].

[53] Ampere Altra 80 core processor, [https://amperecomputing.com/ampere-altra-industrys-first-80-core-server-processor-unveiled/].

[54] Kunpeng 920 block diagram, https://en.wikichip.org/wiki/hisilicon/microarchitectures/taishan_v110.

[55] Thunder X2 block diagram, https://www.nextplatform.com/2018/05/07/thunderx2-arms-hyperscale-and-hpc-compute/.

[56] Thunder X2 core diagram, https://en.wikichip.org/wiki/cavium/microarchitectures/vulcan.

[57] Huawei Taishan 2280 server, https://e.huawei.com/en/products/servers/taishan-server/taishan-2280-v2.

[58] TOP500, http://top500.org/http://top500.org/.

[59] Gigabyte ARM server, https://www.gigabyte.com/ARM-Server.

[60] RIKEN, https://www.riken.jp/en/ [https://www.riken.jp/en].

[61] EPI, https://www.european-processor-initiative.eu/ [https://www.european-processor-initiative.eu/].

[62] A64FX, https://www.fujitsu.com/global/about/resources/news/press-releases/2018/0822-02.html.

[63] A64FX video, Introduction video on A64X [https://youtu.be/eXhlDt2SD8o].

[64] SiPearl, https://sipearl.com [https://sipearl.com/].

[65] Wikichip webpage, a64fx, https://en.wikichip.org/wiki/fujitsu/microarchitectures/post-k [https://en.wikichip.org/wiki/fujitsu/microarchitectures/post-k].

[66] Wikichip webpage, Kunpeng 920, https://en.wikichip.org/wiki/hisilicon/kunpeng/920-6426 [https://en.wikichip.org/wiki/hisilicon/kunpeng/920-6426].

[67] Wikipedia webpage, Kunpeng 920, https://en.wikipedia.org/wiki/HiSilicon#Kunpeng_920_(formally_Hi1620) [https://en.wikichip.org/wiki/hisilicon/kunpeng/920-6426].

[68] Thunder X3 press review, Marvell Announces ThunderX3: 96 Cores & 384 Thread 3rd Gen Arm Server Processor [https://www.anandtech.com/show/15621/marvell-announces-thunderx3-96-cores-384-thread-3rd-gen-arm-server-processor].

[69] ARM blog dotproduct, https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/exploring-the-arm-dot-product-instructions [https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/exploring-the-arm-dot-product-instructions].

[70] NEON IEEE-754 compiliance, https://developer.arm.com/docs/ddi0344/b/neon-vfplite-programmers-model/compliance-with-the-ieee-754-standard/complete-implementation-of-the-ieee-754-standard" [https://developer.arm.com/docs/ddi0344/b/neon-vfplite-programmers-model/compliance-with-the-ieee-754-standard/complete-implementation-of-the-ieee-754-standard].

[71] SVE, Why is the Scalable Vector Extension (SVE) useful for HPC? [https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator/resources/tutorials/why-sve-for-hpc].

[72] SVE, Porting and Optimizing HPC Applications for Arm SVE [https://developer.arm.com/docs/101726/0200].

[73] SVE, Explore the Scalable Vector Extention [https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator/resources/tutorials/sve].

[74] Wikichip webpage, ThunderX2, https://en.wikichip.org/wiki/cavium/thunderx2 [https://en.wikichip.org/wiki/cavium/thunderx2].

[75] Wikipedia SMT, https://en.wikipedia.org/wiki/Simultaneous_multithreading [https://en.wikipedia.org/wiki/Simultaneous_multithreading].

[76] Wikipedia SMT, https://en.wikipedia.org/wiki/Hyper-threading [https://en.wikipedia.org/wiki/Hyper-threading].

[77] Wikipedia RDMA, https://en.wikipedia.org/wiki/Remote_direct_memory_access [https://en.wikipedia.org/wiki/Remote_direct_memory_access].

[78] Wikipedia RoCE, https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet [https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet].

[79] Cavium ThunderX ARM processors, https://www.cavium.com/product-thunderx-arm-processors.html.

[80] Cavium ThunderX design for the dual-socket system, http://www.electronicdesign.com/microprocessors/64-bit-cortex-platform-take-x86-servers-cloud.

[81] ThunderX2® ARM Processors, https://www.cavium.com/product-thunderx2-arm-processors.html.

[82] ThunderX2 - Cavium, https://en.wikichip.org/wiki/cavium/thunderx2.

[83] ThunderX2® CN99XX Product Brief, https://www.cavium.com/pdfFiles/ThunderX2_PB_Rev2.pdf?x=2.

[84] Investigating Cavium's ThunderX: The First ARM Server SoC With Ambition, https://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores/2/.

[85] Puzovic, M., Quantitative and Comparative Analysis of Manycore HPC Nodes, http://www.oerc.ox.ac.uk/sites/default/files/uploads/ProjectFiles/CUDA/ Presentations/2017/2017-06-14-Milos-Puzovic.pdf.

[86] Cray XC50 ARM Product Brief, https://www.cray.com/sites/default/files/Cray-XC50-ARM-Product-Brief.pdf.

[87] Isambard User Documentation, https://gw4-isambard.github.io/docs/index.html.

[88] Isambard User Documentation - Request Account, https://gw4-isambard.github.io/docs/user-guide/requestaccount.html.

[89] Isambard User Documentation - Connecting to Isambard, https://gw4-isambard.github.io/docs/user-guide/connecting.html.

[90] Isambard User Documentation - Phase 2 XC50 ARM, https://gw4-isambard.github.io/docs/user-guide/phase2.html.

[91] Isambard Benchmarks Repository, https://github.com/UoB-HPC/benchmarks.

[92] Isambard-list on ARM HPC package wiki, https://gitlab.com/arm-hpc/packages/wikis/categories/isambard-list [https://gitlab.com/arm-hpc/packages/wikis/categories/isambard-list].

[93] The Cray Programming Environment, https://www.cray.com/sites/default/files/SB-Cray-Programming-Environment.pdf [https://www.cray.com/sites/default/files/SB-Cray-Programming-Environment.pdf].

[94] Cray Performance Measurement and Analysis Tools User Guide, https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/ [https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/].

[95] paranoia.c, Netlib paranoia.c [http://netlib.org/paranoia/paranoia.c].

[96] Roofline model, Roofline model is an intuitive visual performance model [https://en.wikipedia.org/wiki/Roofline_model].

[97] Arm Forge User Guide, https://developer.arm.com/docs/101136/latest/arm-forge [https://developer.arm.com/docs/101136/latest/arm-forge].

[98] Arm MAP webpage, https://developer.arm.com/products/software-development-tools/hpc/arm-forge/arm-map [https://developer.arm.com/products/software-development-tools/hpc/arm-forge/arm-map].

[99] Arm MAP User Guide, https://developer.arm.com/docs/101136/latest/map [https://developer.arm.com/docs/101136/latest/map].

[100] Arm DDT webpage, https://developer.arm.com/products/software-development-tools/hpc/arm-forge/arm-ddt [https://developer.arm.com/products/software-development-tools/hpc/arm-forge/arm-ddt].

[101] Arm DDT User Guide, https://developer.arm.com/docs/101136/latest/ddt [https://developer.arm.com/docs/101136/latest/ddt].

[102] Hawk to Replace Hazel Hen, webpage. [https://www.hlrs.de/news/detail-view/2018-11-13].

[103] HPE Apollo 9000 (Hawk), webpage. [https://www.hlrs.de/systems/hpe-apollo-hawk].

[104] AMD µProf, webpage. [https://developer.amd.com/amd-uprof].

[105] HPE Message Passing Interface (MPI) User Guide, Chapter "Performance profiling", Perfcatch utility. [https://support.hpe.com/hpesc/public/docDisplay?docId=a00048258en_us].

[106] AMD µProf Release Notes, Release v3.2, Chapter 5: Performance Analysis, Section 5.9: Linux perf_event_paranoid setting. [https://developer.amd.com/wordpress/media/files/AMDuprof_Resources/User_Guide_AMD_uProf_v3.2_GA.pdf].

[107] SuperMUC-NG System, https://doku.lrz.de/display/PUBLIC/SuperMUC-NG [https://doku.lrz.de/display/PUBLIC/SuperMUC-NG].

[108] ThinkSystem SD650 Direct Water Cooled Server (Xeon SP Gen 1), https://lenovopress.com/lp0636-thinksystem-sd650-direct-water-cooled-server-xeon-sp-gen-1 [https://lenovopress.com/lp0636-thinksystem-sd650-direct-water-cooled-server-xeon-sp-gen-1].

[109] Lenovo HPC: Energy Efficiency and Water-Cool-Technology Innovations, https://www.slideshare.net/insideHPC/lenovo-hpc-energy-efficiency-and-watercooltechnology-innovations [https://www.slideshare.net/insideHPC/lenovo-hpc-energy-efficiency-and-watercooltechnology-innovations].

[110] Wikichip Zen2 microarchitecture, https://en.wikichip.org/wiki/amd/microarchitectures/zen_2 [https://en.wikichip.org/wiki/amd/microarchitectures/zen_2].

[111] Tuning Guide for AMD EPYC, https://developer.amd.com/wp-content/resources/56827-1-0.pdf [https://developer.amd.com/wp-content/resources/56827-1-0.pdf].

[112] Simple Linux Utility for Resource Management (SLURM), http://slurm.schedmd.com/ [http://slurm.schedmd.com/].

[113] Balloon effect, The Balloon effect, a latex balloon is squeezed: The air is moved, but does not disappear. [https://en.wikipedia.org/wiki/Balloon_effect].

[114] Intel® Math Kernel Library Link Line Advisor, Intel® Math Kernel Library Link Line Advisor. [https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-link-line-advisor.html].

[115] Wikipedia MKL, Wikipedia MKL. [https://en.wikipedia.org/wiki/Math_Kernel_Library].

[116] Intel MKL on AMD, Using Intel MKL on AMD processors. [https://danieldk.eu/Posts/2020-08-31-MKL-Zen.html].

[117] Intel® 64 and IA-32 Architectures Optimization Reference Manual, [https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf].

[118] Intel APS MPI Analysis Charts Manual, [https://software.intel.com/content/www/us/en/develop/documentation/application-snapshot-user-guide/top/detailed-mpi-analysis/analysis-charts.html].

[119] Reduce, Reuse, Recycle, [https://www.int.washington.edu/PROGRAMS/12-2c/week3/joo_03.pdf] slide 15.

[120] Roofline and aritmetric intensity, [https://crd.lbl.gov/departments/computer-science/par/research/roofline/introduction/].

[121] Fastest Fourier Transfer in the West, A popular open source implentation of Fourier Transforms [http://www.fftw.org/].

[122] NPB Benchmark, The High NPB benchmark [https://www.nas.nasa.gov/publications/npb.html].

[123] The High Performance Conjugate Gradients (HPCG) Benchmark project is an effort to create a new metric for ranking HPC systems, The High Performance Conjugate Gradients (HPCG) Benchmark [http://www.hpcg-benchmark.org].

[124] The High Performance Conjugate Gradients (HPCG) Benchmark top500, The High Performance Conjugate Gradients top500 list [https://www.top500.org/hpcg/].

[125] Hackenberg, D., Oldenburg, R., Molka, and D. Schone, R. Introducing FIRESTARTER: A processor stress test utility. Proceedings of the International Green Computing Conference (IGCC), June 2013. https://doi.org/10.1109/IGCC.2013.6604507.

[126] McCalpin J. STREAM benchmark, https://www.cs.virginia.edu/stream/ref.html#what.

[127] Parallel filesystem I/O benchmark, https://github.com/LLNL/ior.

[128] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, https://www.netlib.org/benchmark/hpl/.

[129] Savage benchmark, Savage benchmark is a particularly good benchmark for the fields of satellite tracking and astronomy. [https://www.celestrak.com/columns/v02n04/].

[130] Bifrost, Gudiksen, B. V., Carlsson, M., Hansteen, V. H., Hayek, W., Leenaarts, J., & Martinez-Sykora, J.: 2011, Astronomy and Astrophysics 531, A154, The stellar atmosphere simulation code Bifrost. Code description and validation. [https://www.mn.uio.no/rocs/english/projects/solar-atmospheric-modelling/].

[131] Puzovic, M., Manne S., GalOn S., Ono M., Quantifying Energy Use in Dense Shared Memory HPC Node, 4th International Workshop on Energy Efficient Supercomputing, 2016, p. 16-23.

[132] OpenBLAS library Wiki, OpenBLAS library [https://github.com/xianyi/OpenBLAS/wiki].

[133] PGI compilers, documentation. [https://www.pgroup.com/resources/docs/20.1/x86/index.htm].

[134] LLVM Clang, documentation. [https://developer.amd.com/wp-content/resources/AOCC-2.1-Clang-the%20C%20C++%20Compiler.pdf].

[135] LLVM Flang, documentation. [https://developer.amd.com/wp-content/resources/AOCC-2.1-Flang-the%20Fortran%20Compiler.pdf].

[136] ARM performance libraries, documentation [https://developer.arm.com/docs/101004/latest/1-introduction], in printable format for download [https://static.docs.arm.com/101004/1810/arm_performance_libraries_reference_manual_101004_1801_00_en.pdf].

[137] McIntosh-Smith, S., Price, J., Deakin, T., and Poenaru, A. Comparative Benchmarking of the First Generation of HPC-Optimised Arm Processors on Isambard. Cray User Group, CUG 2018, Stockholm, Sweden. http://uob-hpc.github.io/assets/cug-2018.pdf.

[138] Shoukourian H., Wilde T., Huber H., and Bode A. Analysis of the Efficiency Characteristics of the First High-Temperature Direct Liquid Cooled Petascale Supercomputer and Its Cooling Infrastructure Journal of Parallel and Distributed Computing, Elsevier, Vol. 107 pages 87-100, 2017https://doi.org/10.1016/j.jpdc.2017.04.005.

[139] Shoukourian H., Bode A., Huber H., Ott M., and Kranzlmüller D. SuperMUC - the First High-Temperature Direct Liquid Cooled Petascale Supercomputer Operated by LRZ In Contemporary High Performance Computing: from Petascale toward Exascale, Volume 3, edited by Jeffrey S. Vetter. ISBN 9781138487079, Chapman & Hall/CRC Computational Science, CRC Press, February 2019https://www.crcpress.com/Contemporary-High-Performance-Computing-From-Petascale-toward-Exascale/Vetter/p/book/9781138487079.

[140] McIntosh-Smith, S., Price, J., Deakin, T., and Poenaru, A. A Performance Analysis of the First Generation of HPC-Optimised Arm Processors. In press, Concurrency and Computation: Practice and Experience, Feb 2019. DOI: 10.1002/cpe.5110 [DOI: 10.1002/cpe.5110].

[141] D Ernst, The Arm Technology Ecosystem: Current Products and Future Outlook, 2018. https://iadac.github.io/events/adac5/ernst.pdf.

[142] CPU Profiling Tools on Linux, perf. [http://euccas.github.io/blog/20170827/cpu-profiling-tools-on-linux.html].

[143] Perf utility, description. [https://perf.wiki.kernel.org/index.php/Main_Page].

[144] Perf utility, tutorial. [https://perf.wiki.kernel.org/index.php/Tutorial].

[145] AMD μProf, manual. [https://developer.amd.com/wordpress/media/files/AMDuprof_Resources/User_Guide_AMD_uProf_v3.2_GA.pdf].

[146] AMD Optimizing CPU Libraries (AOCL), developer site. [https://developer.amd.com/amd-aocl/].

[147] Treibig J., Hager G., and Wellein G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In 2010 39th International Conference on Parallel Processing Workshops (pp. 207-216). IEEE, September, 2010..