Best Practice Guide - AMD EPYC

Xu Guo, EPCC, UK

Ole Widar Saastad (Editor), University of Oslo, Norway

Version 2.0 by 18-02-2019

Table of Contents

1. Introduction

Figure 1. The AMD EPYC Processor chip

The EPYC processors are the latest generation of processors from AMD Inc. While they not yet show large adaptation on the top-500 list their performance might change this in the future.

The processors are based on x86-64 architecture and provide vector units for a range of different data types, the most relevant being 64-bits floating point. Vector units are 256 bits wide and can operate on four double precision (64-bits) numbers at a time. The processors feature a high number of memory controllers, 8 in the EPYC 7601 model (see [6] for details) that was used for evaluation in the writing of this guide. They also provide 128 PCIe version 3.0 lanes.

This guide provides information about how to use the AMD EPYC processors in an HPC environment and it describes some experiences with the use of some common tools for this processor, in addition to a general overview of the architecture and memory system. Being a NUMA type architecture information about the nature of the NUMA is relevant. In addition some tuning and optimization techniques as well as debugging are covered also.

In this mini guide we cover the following tools: Compilers, performance libraries, threading libraries (OpenMP), Message passing libraries (MPI), memory access and allocation libraries, debuggers, performance profilers, etc.

Some benchmarks, in which we compare compilers and libraries have been performed and some recommendations and hints about how to use the Intel tools with this processor are presented. While the AMD EPYC is a x86-64 architecture it's not fully compatible with Intel processors when it comes to the new features found on the latest generations of the Intel processors. Issues might be present when using highly optimized versions of the Intel libraries.

In contrast to the Intel tools the GNU tools and tools from other independent vendors have full support for EPYC. A set of compilers and development tools have been tested with satisfactory results.

2. System Architecture / Configuration

2.1. Processor Architecture

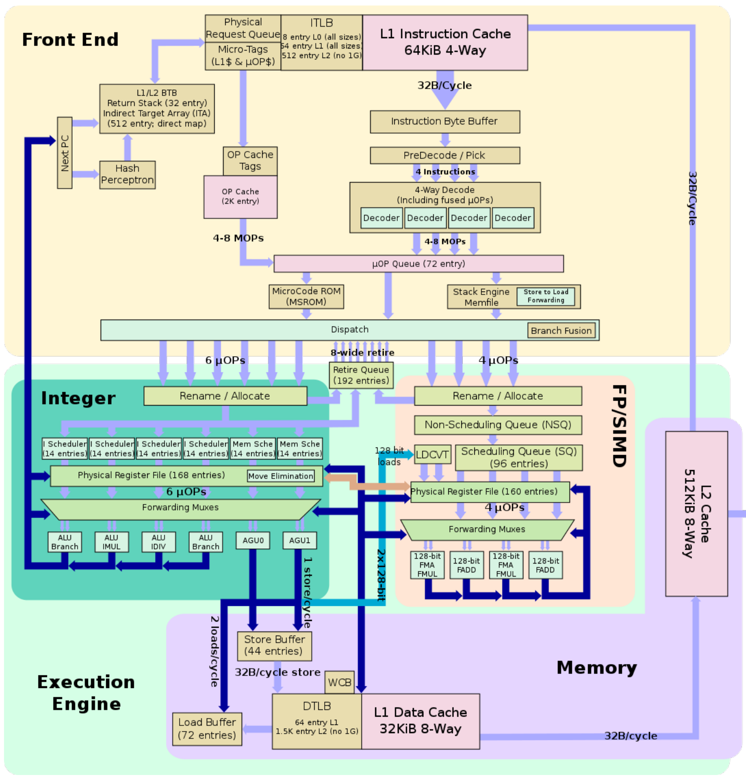
The x86 EPYC processor, designed by AMD, is a System-on-Chip (SoC) composed of up to 32 Zen cores per SoC. Simultaneous Multithreading (SMT) is supported on the Zen core, which allows each core to run two threads giving at maximum 64 threads per CPU in total. Each EPYC processor provides 8 memory channels and 128 PCIe 3.0 lanes. EPYC supports both 1-socket and 2-sockets models. In the multi-processor configuration, half of the PCIe lanes from each processor are used for the communications between the two CPUs through AMD's socket-to-socket interconnect, Infinity Fabric [3][4][5][7].

There are several resources available for information about the Zen architecture and the EPYC processor. The wikichip web site generally is a good source of information [6]. The figures and much of the information below is taken from the web pages at wikichip and their article about Zen. Detailed information about cache sizes, pipelining, TLB etc is found there. The table below just lists the cache sizes as they might be of use for many programmers.

Table 1. Cache sizes and related information

Cache level/type	Size &Information
L0 µOP	2,048 µOPs, 8-way set associative,32-sets, 8-µOP line size
L1 instruction	64 KiB 4-way set associative,256-sets, 64 B line size, shared by the two threads, per core
L1 data	32 KiB 8-way set associative, 64-sets, 64 B line size, write-back policy, 4-5 cycles latency for Int, 7-8 cycles latency for FP
L2	512 KiB 8-way set associative, 1,024-sets, 64 B line size, write-back policy, Inclusive of L1, 17 cycles latency
L3	Victim cache, 8 MiB/CCX, shared across all cores, 16-way set associative, 8,192-sets, 64 B line size, 40 cycles latency
TLB instructions	8 entry L0 TLB, all page sizes, 64 entry L1 TLB, all page sizes, 512 entry L2 TLB, no 1G pages
TLB data	64 entry L1 TLB, all page sizes, 1,532-entry L2 TLB, no 1G pages

Figure 2. Zen Block diagram

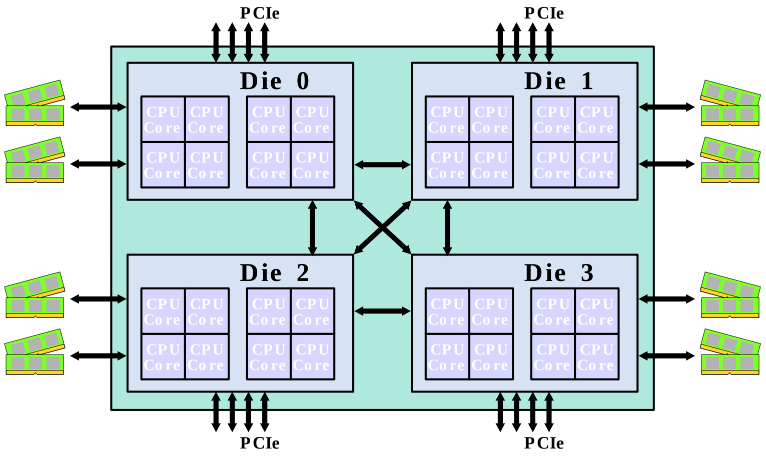


The Zen core contains a battery of different units, it is not a simple task to figure out how two threads are scheduled on this array of execution units. The core is divided into two parts, one front end (in-order) and one execute part (out-of-order). The front end decodes the x86-64 instructions to micro operations which are sent to the execution part by a scheduler. There is one unit for integer and one for floating point arithmetic, there are hence two separate pipelines one for integer and one for floating point operations.

The floating point part deals with all vector operations. The vector units are of special interest as they perform vectorized floating point operations. There are four 128 bits vector units, two units for multiplications including fused multiply-add and two units for additions. Combined they can perform 256 bits wide AVX2 instructions. The chip is optimized for 128 bits operations. The simple integer vector operations (e.g. shift, add) can all be done in one cycle, half the latency of AMD's previous architecture. Basic floating point math has a latency of three cycles including multiplication (one additional cycle for double precision). Fused multiply-add (FMA) has a latency of five cycles.

AMD claim that theoretical floating point performance can be calculated as: *Double Precision theoretical Floating Point performance = #real_cores*8DP flop/clk * core frequency*. For a 2 socket system = 2*32cores*8DP flops/clk * 2.2GHz = 1126.4 Gflops. This includes counting FMA as two flops.

Figure 3. EPYC Block diagram



The EPYC processor contain 32 Zen cores laid out as the figure above shows. This is a cluster on a chip type processor with its own None Uniform Memory Architecture (NUMA). The pronounced NUMA features has implications for the programmer. All programs that want to run efficiently need to be NUMA aware.

2.2. Memory Architecture

Each EPYC processor provides up to 2TiB of DDR4 memory capacity across 8 memory channels with up to 2 DIMMs per channel [8].

The EPYC system is NUMA architecture with a series of individual memory banks. To see the layout use the command numactl.

```
numactl -H
```

The output list the NUMA banks and a map of relative distances, e.g. latency for memory access, see [14] for more information.

```
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 64 65 66 67 68 69 70 71
node 0 size: 32668 MB
node 0 free: 29797 MB
```

plus 7 more, and the map showing the distances:

```
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  16  32  32  32  32
  1:  16  10  16  16  32  32  32  32
  2:  16  16  10  16  32  32  32  32
  3:  16  16  16  10  32  32  32  32
  4:  32  32  32  32  10  16  16  16
  5:  32  32  32  32  16  10  16  16
  6:  32  32  32  32  16  16  10  16
  7:  32  32  32  32  16  16  16  10
```

Care must be taken to schedule the ranks or threads so that they use the nearest memory as much as possible. When scheduling a number of MPI ranks which essential could run exclusively on each NUMA bank the task is relatively easy. However, when scheduling threads (OpenMP uses POSIX threads) the task is not that easy. A single thread can traverse the whole allocated memory as all threads share the same memory space. For a more detailed description of this, see the chapter about tuning, Section 5.4.1.

2.2.1. Memory Bandwidth Benchmarking

The STREAM benchmark [38], developed by John D. McCalpin of TACC, 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)
```

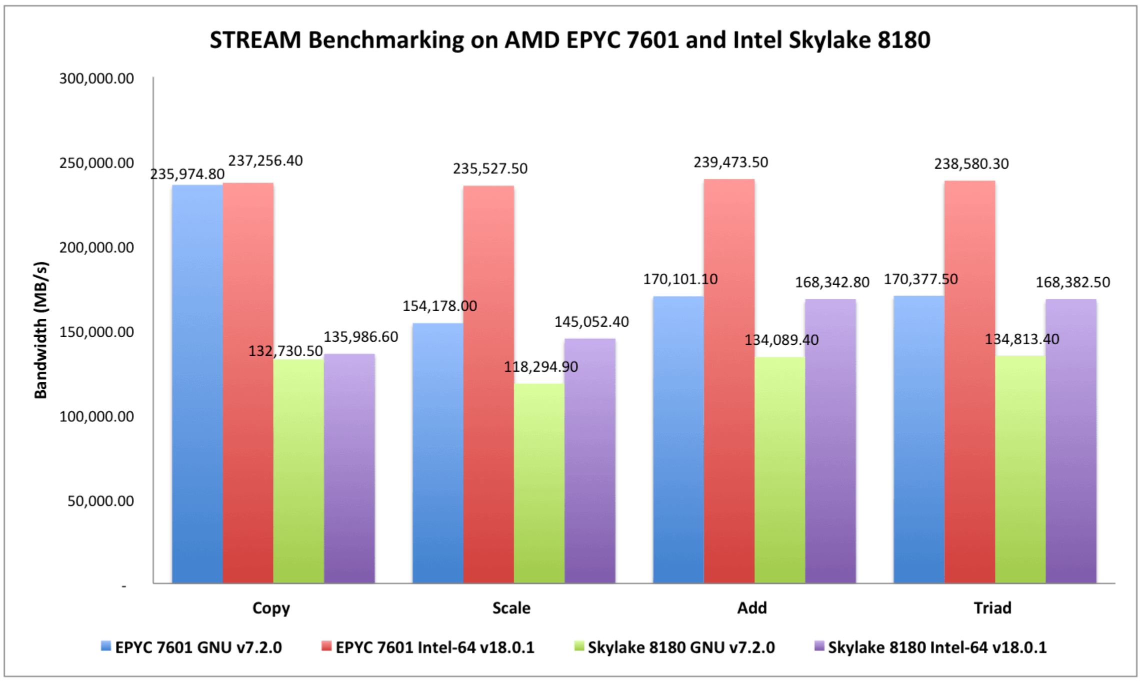
The following benchmarking on EPYC was based on a single node composed of 128 cores using AMD EPYC 7601 32-core processors in a dual-socket configuration. This processor has a CPU clock frequency of 2.20GHz. This duel-socket EPYC has DDR4 memory (8 memory channels) with the memory speed of 2666MHz and a theoretical memory bandwidth of 341GiB/s [8]. The same benchmarking on Skylake was based on a single node composed of 112 cores using Intel Xeon Platinum 8180 28-core processors in a dual-socket configuration. The Skylake processor has a CPU clock frequency of 2.50GHz. This intel processor has DDR4 memory (6 memory channels) with the memory speed of 2666MHz and a theoretical memory bandwidth of 256GB/s [9]. The GNU v7.2.0 compiler and Intel-64 v18.0.1 compiler were used to compile the STREAM benchmark v5.1.0 in C on both systems. A number of compiler options have been tested and the results with the compiler options that delivered the best performance were compared here.

According to the rules of running STREAM benchmark, the array size for the tests was increased significantly by using “-DSTREAM_ARRAY_SIZE=800000000” [10] to ensure that it is much larger than the L3 cache size on both systems. Different settings of “-DNTIMES” were also tested but the changing values did not affect the final results significantly.

Table 2. Compiling options used for STREAM benchmarking on EPYC and Skylake

System and Compiler	Compiler Options
EPYC 7601 GNU v7.2.0	-Ofast -DSTREAM_ARRAY_SIZE=800000000 -DNTIMES=200 -fopenmp -mcmodel=medium
EPYC 7601 Intel-64 v18.0.1	-O3 -DSTREAM_ARRAY_SIZE=800000000 -DNTIMES=200 -qopenmp -mcmodel medium -shared-intel
Skylake 8180 GNU v7.2.0	-Ofast -DSTREAM_ARRAY_SIZE=800000000 -DNTIMES=200 -fopenmp -march=skylake-avx512 -mtune=skylake-avx512 -mcmodel=medium
Skylake 8180 Intel-64 v18.0.1	-O3 -DSTREAM_ARRAY_SIZE=800000000 -DNTIMES=200 -qopenmp -march=skylake -mtune=skylake -mcmodel medium -shared-intel

Figure 4. STREAM Benchmarking on AMD EPYC 7601 and Intel Skylake 8180



The figure above shows the STREAM benchmarking results comparison on EPYC and Skylake (higher is better). The best performance was achieved when using the Intel-64 compiler on both systems. The results showed that EPYC has very promising high memory bandwidth and could achieve around 1.4~1.7x results when using the Intel-64 compiler, compared with the numbers achieved on Skylake. It can be seen that EPYC could be a good choice for the applications with expensive large scale sparse matrix calculations and/or large vector operations.

3. Programming Environment / Basic Porting

3.1. 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. Other compilers like Open64 might also do a decent job.

Compilers installed and tested:

- AOCC/LLVM compiler suite, *cc, fortran*(version 1.0 and 1.2.1)
- GNU compiler suite, *gcc, gfortran, g++*(version 7.2.0 and 8.1.0)
- Intel compiler suite (Commercial) , *icc, ifortran, icpc* (version 2018.1)
- Portland Group (PGI) compiler suite (Commercial), *pgcc, pgfortran, pgCC (version 17.10)*

AMD support the development of a compiler set using LLVM [11]. Using the C and C++ is rather straightforward, it installs with a simple script. Using the AOCC and related Fortran plugin is not as easy, it requires some manual steps and configuration and some extra packages. Presently AOCC require a specific version of gcc (4.8.2). This comes bundled with the package.

AOCC/LLVM Intel, PGI (Portland), LLVM and GNU have been tested.

3.1.1. Compiler Flags

3.1.1.1. Intel

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 3. 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. Using the "-xCORE-AVX2" can also be tried, but it might fail in some cases. 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.

If on the other side the peak performance is not paramount the safe option would be to use the "-axHost" flag which also generates a least common denominator code which will run on any x86-64 processor. The run time system performs checks at program launch to decide which code should be executed.

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.

3.1.1.2. PGI

Table 4. 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++14 and older. Online documentation is available [39].

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.

3.1.1.3. GNU

Table 5. Suggested compiler flags for GNU compilers

Compiler	Suggested flags
gcc compiler	-O3 -march=znver1 -mtune=znver1 -mfma -mavx2 -m3dnow -fomit-frame-pointer
g++ compiler	-O3 -march=znver1 -mtune=znver1 -mfma -mavx2 -m3dnow -fomit-frame-pointer
gfortran compiler	-O3 -march=znver1 -mtune=znver1 -mfma -mavx2 -m3dnow -fomit-frame-pointer

3.1.1.4. AOCC

Table 6. Suggested compiler flags for AOCC compilers

Compiler	Suggested flags
clang compiler	-O3 -march=znver1 -mfma -fvectorize -mfma -mavx2 -m3dnow -floop-unswitch-aggressive -fuse-ld=lld
clang++ compiler	-O3 -march=znver1 -mfma -fvectorize -mfma -mavx2 -m3dnow -fuse-ld=lld
Fortran dragonegg/clang compiler	-O3 -mavx -fplugin-arg-dragonegg-llvm-codegen-optimize=3 -fplugin-arg-dragonegg-llvm-ir-optimize=3

The clang compiler is under development with assistance from AMD. The fortran front end is based on gcc 4.8.2 and hence does not have flags for the Zen architecture, alternatives do exist and the documents referenced below provide more information. The options may change, more information about the usage of the clang compiler is available online [40]. For the Dragonegg Fortran compiler online documentation is also available [41]. This compiler suite is under heavy development and subject to change. It's require some manual extra work to install. But at the time of writing this guide it was not a streamlined product (version 1.0 of AOCC). Please visit the AMD developer site to obtain the latest information and releases.

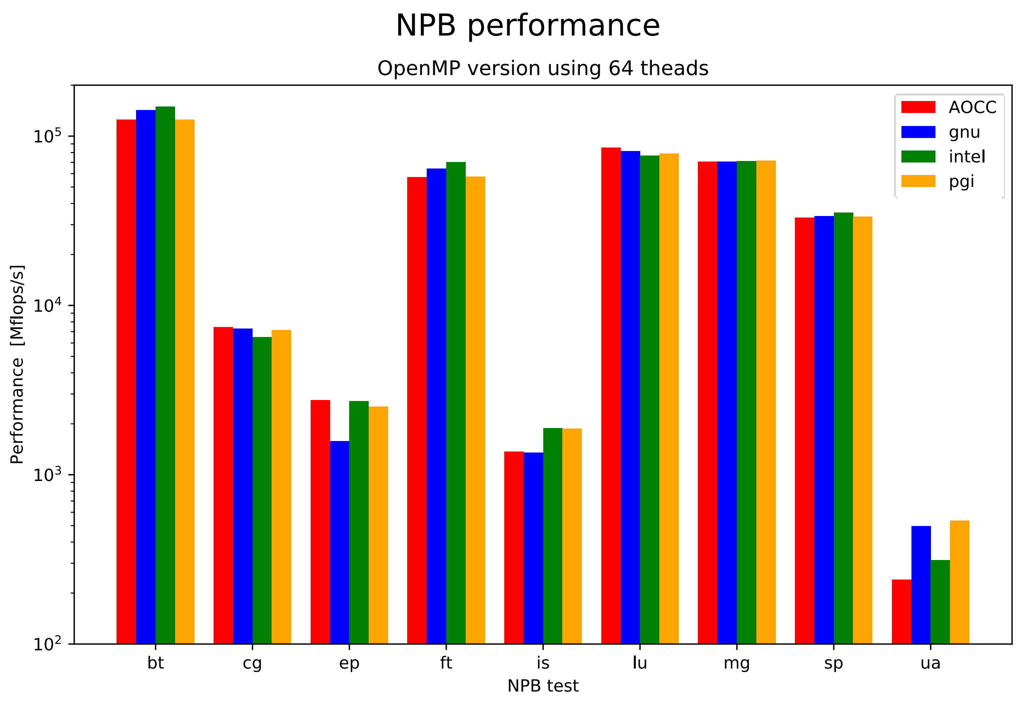
The Zen architecture in the EPYC processor does no longer support FMA4. However, sources claim it still is available and works, See [12]. However, it might suddenly just vanish, so any usage of the flag -mfma4 should be avoided.

3.1.2. Compiler Performance

3.1.2.1. NPB OpenMP version

The well-known set of benchmarks found in the NPB [49] suite is used for several examples in this guide. The performance numbers are in flops numbers, hence higher is better. The different compilers show varying performance with the different NPB benchmarks. The figure below shows the performance recorded using the OpenMP version of the NPB benchmarks. The OpenMP version is chosen over MPI as the OpenMP thread library is an integral part of the compiler and should be evaluated together with the code generation. The different tests in the NPB benchmark suite check both the Fortran and C implementations. Review the benchmark's documentations for details. From the figure below it's evident that all the tested compilers do a fairly good job.

Figure 5. Compiler performance comparison



The log scale is used because the different benchmark metrics cover a rather large range. Log scale is used to cover all the benchmarks is one figure. It show that there is some variance in the compiler performance. Hence it's worth the effort to test a few compilers with your application.

3.1.2.2. High Performance Conjugate Gradients benchmark, OpenMP version

The High Performance Conjugate Gradients (HPCG) benchmark [47] is gaining more and more interest because the Linpack (HPL) benchmark used to assess the 500 fastest systems in the world has some shortcomings [48]. HPCG generally yields a very low processor efficiency due to the fact that this benchmark is highly memory bound.

The OpenMP version of the benchmark is used on a single system with shared memory because the compiler's and system's ability to run multiple threads over a shared large memory is of interest. Some extra compiler flags that deal with prefetch have been added in addition to the suggested flags in the table above. Being a very memory intensive code prefetch can improve performance by overlapping memory transfers with computation. However, the hardware also does prefetching and issuing software prefetch instructions can be counterproductive in some cases.

We build the HPCG code with the reference implementation of the linear algebra because we want to test compiler performance and not tuned library performance.

The table below shows the flags used to build the executables.

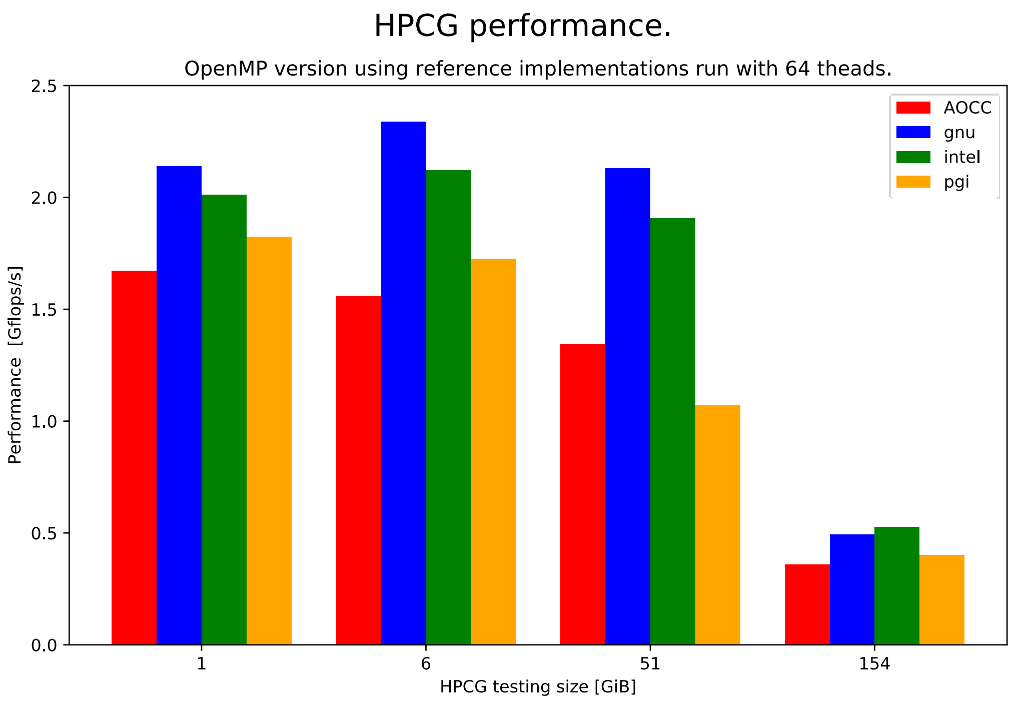
Table 7. Flags used to compile HPCG

Compiler	Flags used
AOCC, clang++	-Ofast -ffast-math -ftree-vectorize -fopenmp -march=znver1 -mtune=znver1 -mfma -mavx2 -m3dnow -fuse-ld=lld
GNU, g++	-O3 -ffast-math -ftree-vectorize -fopenmp -march=znver1 -mtune=znver1 -mfma -mavx2 -m3dnow -fprefetch-loop-arrays -mprefetchwt1
PGI, pgc++	-mp -O4 -tp zen -Mvect=simd -Mcache_align -Mprefetch -Munroll
Intel, icpc	-std=gnu++98 -qopenmp -O3 -march=haswell -fma -ftz -fomit-frame-pointer

For the Intel compiler there were some issues using flags of the type core-avx2 etc. Hence a more moderate flag opting for the Haswell architecture was used. Not all flags calling for AVX2 capable Intel processors worked well on the EPYC processor. This was more prominent with C/C++ than with the Fortran compiler. Again some trial and error is expected.

The best results from different processor bindings were reported, processor binding generally gave best results, (OMP_PROC_BIND=1 or numactl -l etc.), see Section 5.4.1 for more on the effect of processor binding.

Figure 6. Compiler performance comparison using HPCG



The HPCG benchmark is compiled using the reference version of the linear algebra library [16] and other functions that normally are called from an optimized library. This is by choice as the test should illustrate the different compiler's ability to generate efficient code. It's clear that all the C++ compilers tested generate code that performs this task reasonably well.

It's interesting to notice that performance drops as soon as the benchmark footprint spans more than one NUMA memory node. The total memory of 256 GiB is made up of 8 NUMA nodes of 32 GiB each. This performance drop is a consequence of the different latencies in non-local memory accesses in a None Uniform Memory Access (NUMA) system. It might well be that this problem should have been tackled using a hybrid model with one MPI rank per NUMA node and 8 threads per MPI rank keeping the fine granular memory accesses within the local NUMA node.

As always with HPCG the absolute performance as compared to the theoretical peak performance is very low. This is part of the reason that this benchmark now shows growing interest and is used as an alternative benchmark as top500 HPCG along with the top500 HPL benchmark.

3.2. Available (Optimized) Numerical Libraries

The infrastructure and ecosystem of software around the AMD processors are seriously lagging behind that of some other large chip makers. A key to exploit the modern microprocessors is the availability of good compilers and well tuned numerical libraries. So far only a rather limited and not yet mature set has been presented. This presents some obstacles when it comes to ease of use for scientists. This section will list and provide an overview of tuned numerical libraries. While a later chapter will cover tuning and including usage of some selected libraries.

AMD has provided some libraries, see table below.

Table 8. Numerical libraries by AMD

Library name	Functions
BLIS	Basic Linear Algebra Subprograms (BLAS)
libFLAME	LAPACK routines
Random Number Generator Library	Pseudorandom number generator library
AMD Secure Random Number Generator	library that provides APIs to access the cryptographically secure random numbers
libM	Math library (libm.so, sqrt, exp, sin etc)

For more detailed information about the AMD libraries please consult the AMD web pages about libraries [43].

There are several other numerical libraries optimized for x86-64 architectures.

Table 9. Other Numerical libraries

Library name	Functions
OpenBLAS	Basic Linear Algebra Subprograms (BLAS) [17]
GNU Scientific library (GSL)	A rather large range of numerical routines [18]
FFTW	Fast Fourier Transforms in 1,2,3-dimensions [20]

One old classic is OpenBLAS (formerly known as the Goto library). This is built and optimized for the current architecture. While not really optimized for AMD EPYC the GSL and FFTW are optimized at compiling time as they try to guess the vector capabilities of the processor. This is especially true for FFTW.

3.2.1. Performance of libraries

A simple test to check the simple math functions is the savage benchmark [46].

```
for (k=1;k<=m;k++) a=tan(atan(exp(log(sqrt(a*a)))))+1.0;
```

Where m is a fairly large number, say 50 million (turn off optimization or the loop will be made redundant, use -O0).

Table 10. Math library performance

Library	Wall time [seconds]
Standard libm	55.78
AMD libM	31.73

It's obvious that something is not optimal with the implementation of libm supplied with the distribution. Yet another example that the functions in standard math libraries supplied with the distribution are not optimal. The distribution need to run on all types of x86-64 processors and the routines have been written to avoid instructions that were not in the original x86-64 instruction set many years ago.

Intel has provided a vector library for math functions that offers the possibility to use the vector units to handle 4 (64 bit float) or 8 (32 bit float) calls to a function simultaneously (using 256 bit AVX2 instructions). This is called short vector math library. This is more or less just a vector version of the common math functions library. The performance can be quite good. For the real peak performance the intrinsic math functions should be used, but this requires somewhat more programming effort.

Beginning with glibc 2.22[1] a vectorized version of the libm library is available. The syntax is very similar, just use *-lmvec* instead of *-lm*, but in addition the flags *-ftree-vectorize -funsafe-math-optimizations* and *-ffast-math* are needed.

A very simple check if the compiler managed to vectorize or not is to leave out the -lmvec and look for the missing symbols, if the missing symbol is (like this example) *_pow()* then it's not vectorized, but if you see something like *_ZGVdN4vv___pow_finite* then the call is part of the vector library libmvec.

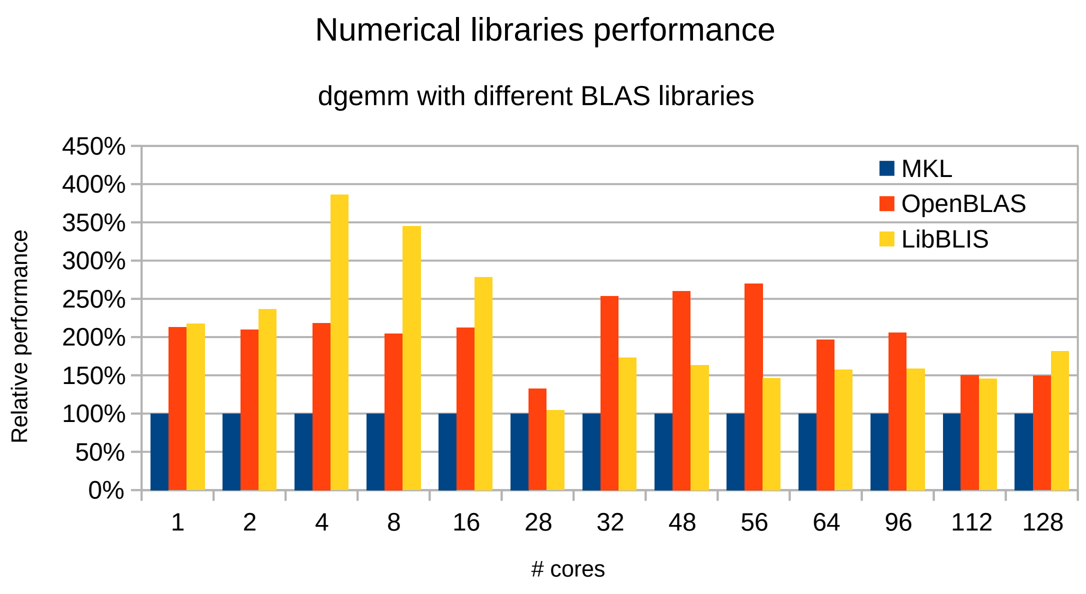
An example of evaluating the possible gain for different library implementations of the function *pow()* is shown below:

Table 11. Math library performance

Library	Wall time [seconds]
gcc and libm	24.5
gcc and libmvec	8.69
gcc and AMD libM	15.1
Intel icc and svml	14.8
Intel intrinsic svml	11.0
gcc and Intel imf and mkl	9.14

The best results were obtained by using vectorized functions, both the novel vectorized math library in glibc or the Intel vectorized libraries. See example section below (Section 3.2.2) for the compile and link line.

Figure 7. Numerical libraries performance, linear algebra



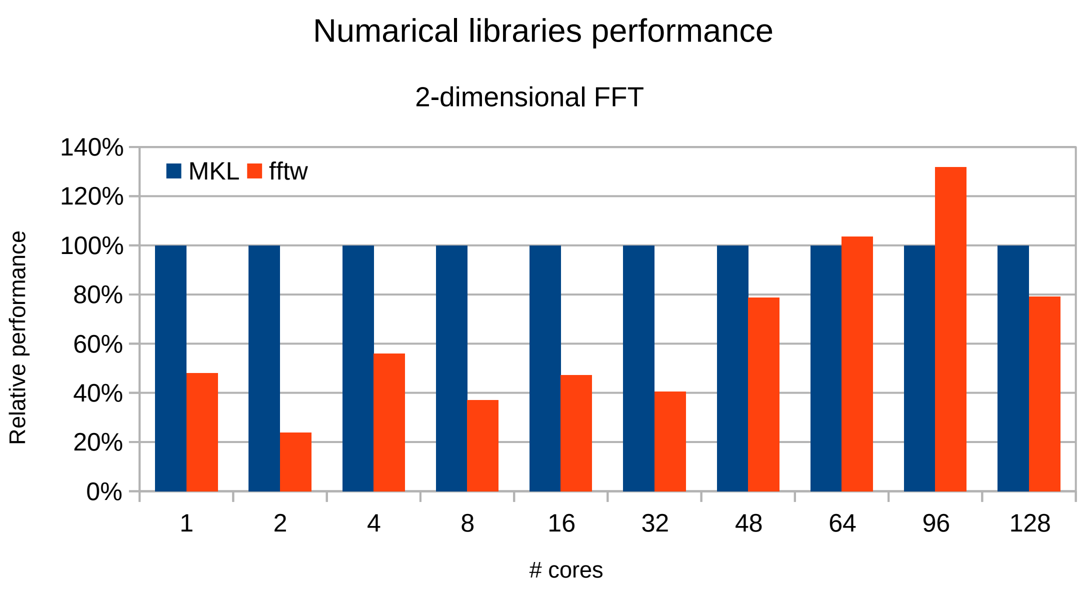
The matrix matrix multiplication were run using a matrix size yielding a footprint of 9 GiB. The executable was compiled with the intel compiler using the following line :

[1]Check using *ldd --version*

```
ifort -o dgemm.x -O2 -qopenmp dgemm-test.f90 mysecond.c -lBLASLIB
```

Where BLASLIB is a suitable linear algebra library.

Figure 8. Numerical libraries performance, FFT



The FFT tests were run using a size of approximately 63 GiB and the executable built using the GNU compiler with a line like :

```
gfortran -O3 -fopenmp -march=znver1 -mtune=znver1 -mfma -mavx2\
-fomit-frame-pointer -o fftw-2d.x mysecond.o  fftw-2d.f90 -lFFT
```

Where FFT is a suitable FFT library (MKL or FFTW).

The figures above are examples of performance variation using different numerical libraries. The Intel Math Kernel Library (MKL) does not do a very good job with the AMD processors for the dgemm linear algebra example, which is dependent on vectors and fused multiply add. It will not select the AVX2 and FMA instruction enabled routines. Instead a common x86-64 based function is used. This approach will always work on all processors and in all situations, but the performance is consequently inferior. The OpenBLAS (formerly known as Goto) and the AMD recommended LibBLIS does a decent job. With the Fast Fourier Transform library the picture is different as this algorithm utilises a different set of execution units in the processor. No definite answer and guidelines can be given.

3.2.2. Examples of numerical library usage

3.2.2.1. Intel MKL and FFTW

Example of compiling and linking linear algebra and fftw.

```
gfortran -o dgemm.x -fopenmp -O3 dgemm-test.f90\
/opt/amd-blis/blis/lib/zen/libblis.a mysecond.o

gfortran -o dgemm.x -fopenmp -O3 dgemm-test.f90 mysecond.o\
-lmkl_avx2 -lmkl_core  -lmkl_gnu_thread -lmkl_gf_lp64

ifort -qopenmp -O3 -march=core-avx2 -align array64byte -fma\
-I/usr/local/include    -o fftwtw-2d.f90  fftw-2d.f90\
/usr/local/lib/libfftw3_omp.a /usr/local/lib/libfftw3.a
```

and compiling and linking using MKL and the FFTW MKL interface.

```
ifort -qopenmp -O3 -march=core-avx2 -align array64byte -fma\
-I/opt/intel/compilers_and_libraries_2018.0.128/linux/mkl/include/fftw\
-o fftw-2d.x   fftw-2d.f90  -mkl=parallel\
/opt/intel/compilers_and_libraries_2018.0.128/linux/mkl/lib/intel64/\
libfftw3xf_intel_ilp64.a

gfortran -O3 -fopenmp -march=znver1 -mtune=znver1 -mfma -mavx2\
-m3dnow -fomit-frame-pointer\
-I/opt/intel/compilers_and_libraries_2018.0.128/linux/mkl/include/fftw\
-o fftw-2d.x fftw-2d.f90\
-L/opt/intel/compilers_and_libraries_2018.0.128/linux/mkl/lib/intel64/\
-lfftw3xf_gnu_ilp64 -lmkl_gf_ilp64 -lmkl_core -lmkl_gnu_thread mysecond.o
```

There are lot of different ways of combining the Intel MKL. The nm tool[2] can be of great help to locate functions within the different library files. This is an example:

```
nm -A /opt/intel/mkl/lib/intel64_lin/lib*|grep DftiErrorClass | grep " T"
```

Intel also has a tool, the Intel MKL link-line advisor, to ease the process of finding the right link line [15].

3.2.2.2. Short vector math library, libsvml

3.2.2.2.1. Manual usage of intrinsic functions

The short vector math library functions take advantage of the vector unit of the processor and provide an easy access to well optimized routines that map nicely on the vector units.

Usage of intrinsics like this is mostly used in libraries and special time critical parts of programs. It is however, an easier alternative to inline assembly.

The svml is linked by default when using the Intel compilers and the functions are not easily available directly from source code. They are, however accessible through intrinsics. Intel provides a nice overview of the intrinsic functions available [13]. Usage of intrinsics can yield quite good performance gain. In addition intrinsics are compatible with newer versions of processors as the compilers and libraries are updated while the names stay the same. Usage of inline assembly might not be forward compatible. Intel strongly suggest using intrinsics instead of inline assembly.

A simple example of intrinsics usage is show below (compare to the simple for loop in the section below):

```
for(j=0; j<N; j+=4){
   __m256d vecA = _mm256_load_pd(&a[j]);
   __m256d vecB = _mm256_load_pd(&b[j]);
   __m256d vecC = _mm256_pow_pd(vecA,vecB);
   _mm256_store_pd(&c[j],vecC);
}
```

3.2.2.2.2. Automatic usage of Short Math Vector Library

At high optimization the compiler will recognize the simple expression above and vectorize it and performance gain will be lesser. The usage of these intrinsics is at its best when the compiler totally fails to vectorize the code. No definite answer can be given, it depends on the performance problem under investigation.

If you want to play with this: there is a blog by Kyle Hegeman that will be helpful [42].

The command lines used to link the examples using short vector math are for the simple case where icc set up calls to the svml library :

[2]GNU Development Tools, see man nm for more information.

```
icc -o vector.x vector.c mysecond.c
```

while the more complicated line using gcc to make calls to the intel compiler's libraries:

```
gcc -o vector.x -O2 vector.c mysecond.o\
-L/opt/intel/compilers_and_libraries/linux/lib/intel64/ -limf -lintlc
```

This is the simplest link line using gcc, the intel libraries have many dependencies between libraries. Finding the exact intel library link command line is not trivial. but by using the nm tool to search for symbols it should normally be straightforward. Another option is the Intel MKL link advisor [15], this however, cover mostly MLK.

The gcc compiler has an option to use external libraries like svml, (and acml) the option -mveclibabi=svml will enable gcc to generate calls to svml. For a simple loop like:

```
for(j=0; j<N; j++) c[j]=pow(a[j],b[j]);
```

the gain can be quite good, a look at the assembly code will show that a call to the function *vmldPow2,* which is a function found in libsvml. The following example show performance improvement, compiling using:

```
gcc -o vector.x -O3 vector.c mysecond.c -ftree-vectorize
-funsafe-math-optimizations -mavx2 -lm
```

or with svml

```
gcc -o vector.x -O3 vector.c mysecond.c
-mveclibabi=svml -ftree-vectorize -funsafe-math-optimizations -ffast-math
-mavx2 -L/opt/intel/compilers_and_libraries/linux/lib/intel64 -lsvml
```

This is a very elegant and simple way of accessing the svml library.

Table 12. Performance using gcc with svml

Performance library	**Wall time (sec)**
Lib math, libm (/usr/lib64/libm.so)	93.03
Lib math, libmvec (/usr/lib64/libmvec.so)	25.25
Lib short Vector math, libsvml	14.39

Se next section about libmvec. Please consult the gcc documentation for more information. The gcc man page lists which function calls that can be emitted by gcc. For more information about libsvml please consult the Intel documentation.

3.2.2.3. Gnu vector math library

The libraries provided by gcc contain a vector library similar to the svml. It's part of the SIMD parallel execution in OpenMP. While vector operations are part of OpenMP via the SIMD directive, this library works fine without any OpenMP directives in the source code.

Using the example from the former section the link line looks like this:

```
gcc -o vector.x -O2 -fopenmp -ffast-math  -ftree-vectorize -mavx2\
vector.c mysecond.o -lm
```

For performance see table Table 12.

3.3. Available MPI Implementations

There are several possible MPI implementations available. The most commonly used are Intel MPI and OpenMPI. While MPICH and HP-MPI also enjoy popularity. The EPYC is a standard x86-64 architecture so there is not really any difference in installing and building software. Hence all software is expected to work without any adaptation. The table below give an overview of the more common MPI implementations.

Table 13. Popular MPI implementations

MPI	Notes
OpenMPI	Open source, uses hwloc library to schedule and bind ranks to cores, very flexible, Uses a Byte Transfer Layer (btl) for communication devices. Full MPI-3.1 standards conformance
MPICH	Open source, widely used, an old timer. MVAPICH is a derived implementation. Support the MPI 3 standard.
Intel MPI	Commercial, an integral part of Intel Parallel Studio, integrates with many Intel tools. Support the MPI 3.1 standard.
HP-MPI	Commercial, embedded in a some applications. Support the MPI 2.2 standard.

The syntax is slightly different for these implementations, but they all contains wrappers to compile (like mpicc) and variants of mpirun to run. A description of the use of MPI is well covered elsewhere and therefore is not included here. In the tuning chapter in this guide there are examples on how to place the different ranks and bind them to specific parts like hwthread, core, L2/L3 cache, NUMA node, Socket and node. On a system with 8 NUMA nodes like EPYC this is quite important.

3.4. OpenMP

The OpenMP standard for specifying threading in programming languages like C and Fortran is implemented in the compiler itself and as such is an integral part of the compiler in question. The OMP and POSIX thread library underneath can vary, but this is normally hidden from the user. OpenMP makes use of POSIX threads so both an OpenMP library and a POSIX thread library is needed. The POSIX thread library is normally supplied with the distribution (typically /usr/lib64/libpthread.so).

The fact that EPYC is a NUMA system makes thread placement important. How the OS schedules and distributes the threads on the cores and attached memory can influence the performance.

3.4.1. Compiler Flags

Compiler flags vary from compiler to compiler, the table below gives the flags needed to turn on OpenMP and to read and parse the source code comments directives.

Table 14. Compiler flags to invoke OpenMP support

Compiler	Flag to select OpenMP	OpenMP version supported
Intel compilers	-qopenmp	From 17.0 on : 4.5
GNU compilers	-fopenmp	From GCC 6.1 on : 4.5
PGI compilers	-mp	4.5

3.5. Basic Porting Examples

A few applications have been ported to AMD EYPC 7301 for test. The test platform was a 16-node system composed of 16-core AMD EYPC 7301 processors with dual-socket configuration. The CPU rate is 2.20GHz. Please note the tests were only for basic porting test rather than performance tuning.

3.5.1. OpenSBLI

OpenSBLI, developed by University of Southampton, is a Python-based modeling framework that is capable of expanding a set of differential equations written in Einstein notation, and automatically generating C code that performs the finite difference approximation to obtain a solution. [21]

OpenSBLI was ported to the AMD EYPC 7301 using GNU 7.2.0 and Intel17 compilers. Further info on the build instructions, example job scripts and example run results can be found from the following links:

- OpenSBLI build instructions with GNU 7.2.0: [22]
- OpenSBLI build instructions with Intel17: [23]
- OpenSBLI example job script: [24][25]
- OpenSBLI example run results: [26][27]

3.5.2. CASTEP

CASTEP, developed by the Castep Developers Group (CDG), is a full-featured materials modelling code based on a first-principles quantum mechanical description of electrons and nuclei. It uses the robust methods of a plane-wave basis set and pseudopotentials. [28]

CASTEP was ported to the AMD EYPC 7301 using GNU 7.2.0 . Further info on the build instructions, example job scripts and example run results can be found from the following links:

- CASTEP build instructions with GNU 7.2.0: [29]
- CASTEP example job script: [30]
- CASTEP example run results: [31]

3.5.3. GROMACS

GROMACS, developed by the GROMACS team [32], is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. [33]

GROMACS was ported to the AMD EYPC 7301 using GNU 7.2.0. Further info on the build instructions, example job scripts and example run results can be found from the following links:

- GROMACS build instructions with GNU 7.2.0: [34]
- GROMACS example job script: [35]
- GROMACS example run results: [36]

4. Performance Analysis

4.1. Available Performance Analysis Tools

There are several tools that can be used to do performance analysis. In this mini guide only a small sample is presented.

4.1.1. perf (Linux utility)

The package perf is a profiler tool for Linux. 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 [44] and tutorial [45].

To use the tool is very simple, this simple example illustrate it:

```
perf stat -d -d -d  -B  ./bin.amd.pgi/ft.D.x
```

It produces the normal application output and emits performance statistics at the end. The above run produced an output like this:

```
Performance counter stats for './bin.amd.pgi/ft.D.x':

31369526.971785  task-clock (msec)         #   63.084 CPUs utilized
        2644784  context-switches          #    0.084 K/sec
           8203  cpu-migrations            #    0.000 K/sec
      291478964  page-faults               #    0.009 M/sec
 82784624497276  cycles                    #    2.639 GHz
 80193260794737  stalled-cycles-frontend   #   96.87% frontend cycles idle
 19120224047161  stalled-cycles-backend    #   23.10% backend cycles idle
 68258717123778  instructions              #    0.82  insn per cycle
                                            #    1.17  stalled cycles per in
 19286086849926  branches                  #  614.803 M/sec
     6240215654  branch-misses             #    0.03% of all branches
 35354448022854  L1-dcache-loads           # 1127.032 M/sec
    42217478927  L1-dcache-load-misses     #    0.12% of all L1-dcache hits
              0  LLC-loads                 #    0.000 K/sec
              0  LLC-load-misses           #    0.00% of all LL-cache hits
  1859844311886  L1-icache-loads           #   59.288 M/sec
    13450731189  L1-icache-load-misses
 35350211138280  dTLB-loads                # 1126.897 M/sec
    13477704352  dTLB-load-misses          #    0.04% of all dTLB cache hits
  1859455510321  iTLB-loads                #   59.276 M/sec
        4451162  iTLB-load-misses          #    0.00% of all iTLB cache hits
       16714849  L1-dcache-prefetches      #    0.533 K/sec
       51264834  L1-dcache-prefetch-misses #    0.002 M/sec

  497.269857091 seconds time elapsed
```

For real time analysis the "top" option of the tool can be quite handy. A snapshot of the real time update produced by the command "perf top" is shown :

```
Samples: 33M of event 'cycles', Event count (approx.): 1305563178604
Overhead  Shared Object            Symbol
  56.22%  libpgmp.so               [.] _mp_barrier_tw
  10.22%  ft.D.x                   [.] fftz2_
   6.99%  libpgc.so                [.] __c_mcopy16
   5.64%  ft.D.x                   [.] evolve_
   3.59%  ft.D.x                     [.] cffts1_
   2.94%  [kernel]                   [k] down_read_trylock
   2.29%  [kernel]                   [k] smp_call_function_many
   2.20%  [kernel]                   [k] up_read
   1.48%  [kernel]                   [k] llist_add_batch
```

4.1.2. AMD µProf

The AMD µProf is a suite of powerful tools that help developers optimize software for performance or power. Information can be found at the AMD web pages [50]. It's a tool that collects information during a run and in a second step generates a report of what was collected. Quite similar to other similar tools. It has full support for the AMD processor. It is a command line tool which makes it easy to use in scripts.

```
/opt/AMDuProf_1.0-271/bin/AMDCpuProfiler collect bin.amd.gcc/mg.D.x
```

(The tool installs itself under /opt)

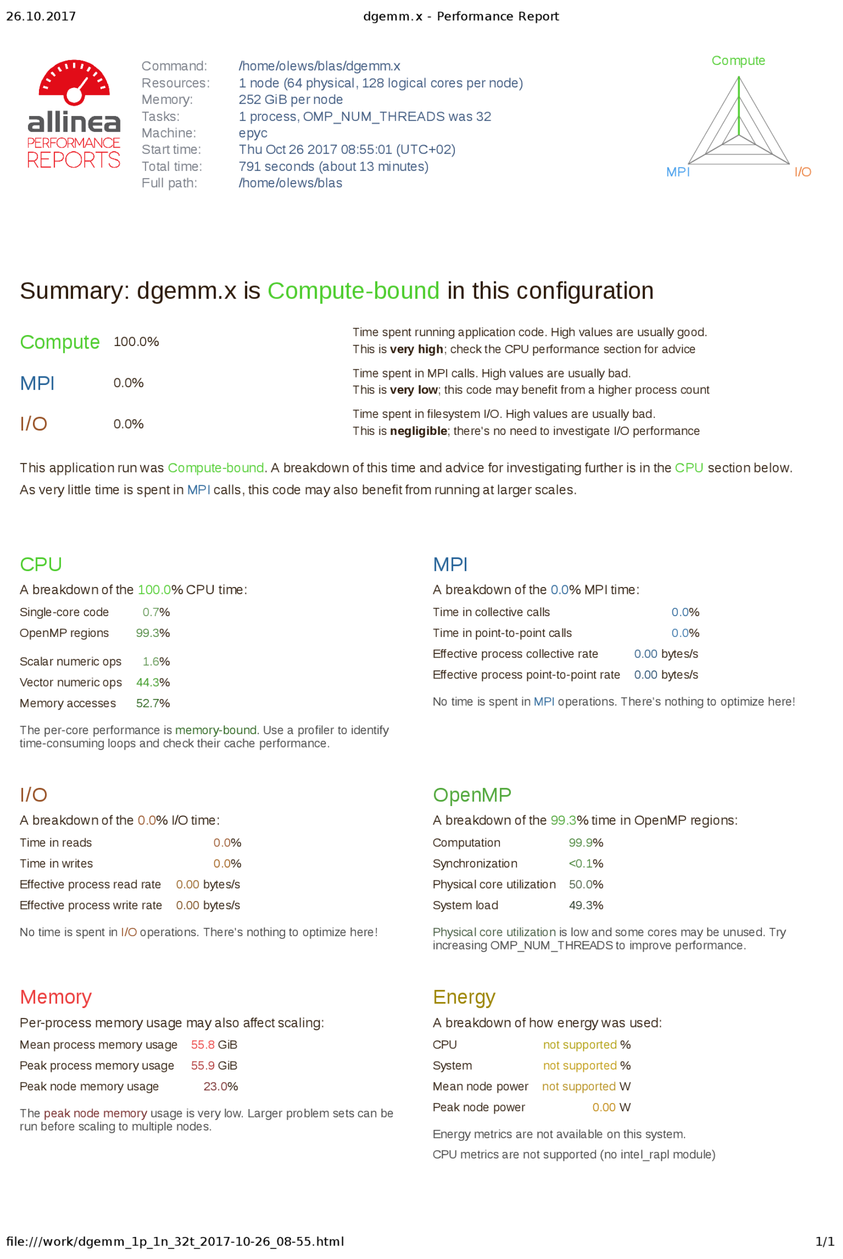
The user guide is available on-line [54].

4.1.3. Performance reports

A very user friendly tool is the commercial Allinea Performance Reports [55]. This is licensed software.

The performance reporter is a very easy tool to use, an excellent tool for normal users to run and to provide a quick and easy overview of the application behavior. This can be handy when they submit request for support or CPU quota to provide a quick and easy overview of the application.

Figure 9. Performance report example



26.10.2017 dgemm.x - Performance Report

Command:	/home/olews/blas/dgemm.x
Resources:	1 node (64 physical, 128 logical cores per node)
Memory:	252 GiB per node
Tasks:	1 process, OMP_NUM_THREADS was 32
Machine:	epyc
Start time:	Thu Oct 26 2017 08:55:01 (UTC+02)
Total time:	791 seconds (about 13 minutes)
Full path:	/home/olews/blas

Summary: dgemm.x is Compute-bound in this configuration

Compute	100.0%	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	0.0%	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below. As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 100.0% CPU time:

Single-core code	0.7%
OpenMP regions	99.3%
Scalar numeric ops	1.6%
Vector numeric ops	44.3%
Memory accesses	52.7%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 0.0% MPI time:

Time in collective calls	0.0%
Time in point-to-point calls	0.0%
Effective process collective rate	0.00 bytes/s
Effective process point-to-point rate	0.00 bytes/s

No time is spent in MPI operations. There's nothing to optimize here!

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

OpenMP

A breakdown of the 99.3% time in OpenMP regions:

Computation	99.9%
Synchronization	<0.1%
Physical core utilization	50.0%
System load	49.3%

Physical core utilization is low and some cores may be unused. Try increasing OMP_NUM_THREADS to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	55.8 GiB
Peak process memory usage	55.9 GiB
Peak node memory usage	23.0%

The peak node memory usage is very low. Larger problem sets can be run before scaling to multiple nodes.

Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.
CPU metrics are not supported (no intel_rapl module)

file:///work/dgemm_1p_1n_32t_2017-10-26_08-55.html 1/1

To run the analysis and generate the report is very easy. An example is shown here:

```
/opt/allinea/reports/bin/perf-report ./dgemm.x
```

(the path is the default path of perf-report). This command will generate two files, one text file which can be displayed on any terminal with text based graphics and an HTML based graphical view like the figure above.

4.2. General Hints for Interpreting Results from all tools

The ratio between scalar and vector work is very important when doing computational based work. A high fraction of vector versus scalar code is a sign that the vector units are occupied and do operations in parallel. With 256 bits the vector unit can do four 64-bits double precision operations in parallel, or eight if 32-bits single precision is used. Same metrics apply for integer work. Recognizing vectorizable code is usually a compiler issue, some compilers do a better job than others. It's also an issue how the programmer write code to facilitate vectorization.

Memory access is another crucial aspect of application performance. The report also provides an estimate of how much time is spent in memory access. A very high fraction here might indicate a memory bandwidth bound application. Hints are provided to initiate further work.

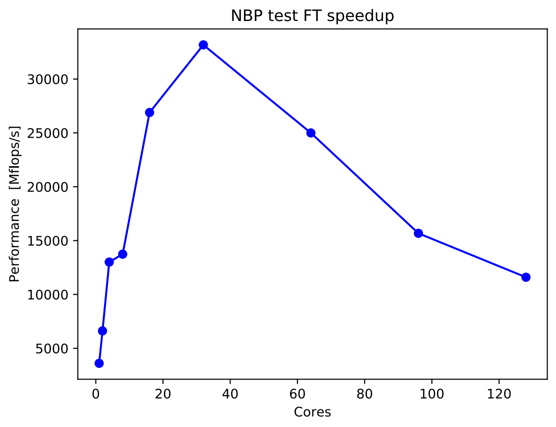
Time spent doing IO and IO bandwidth is shown in performance reports above. For the bandwidth number this should be compared with the storage device bandwidth. There might be IO bottlenecks. Could another storage device been used during the run ? Maybe there is a solid state disk available or even a PCIe-NVRAM disk ? If IO is a bottleneck and a large fraction is spent in IO an analysis of the IO pattern is needed. It might be that the IO is random access using small records which is very bad.

As the NPB-BT example above is an OpenMP application (see Section 3.1.2.1) there is no information about MPI, evident from the performance reports. If MPI applications are analyzed there should be data showing MPI functions usage. If functions like MPI_WAIT, MPI_WAIT_ALL or MPI_BARRIER show up it could be that a lower number of ranks might be a better option. If not so, a review of the input or source code is needed.

Thread utilization is another important parameter. The perf tool's interactive top command can provide information about thread utilization. How much time is spent in computation and how much time is spent waiting for synchronization?

The perf utility above it's evident that when the "barrier" functions are on the top list something is not really right. It might be that too many cores are being used. In that case, checking the scaling is a good idea. The simple table below shows the NPB benchmark ft scaling. With this poor scaling it's no wonder that the OpenMP "barrier" is high up on the topcpu list.

cores	Performance
1	3351.05
2	6394.50
4	13101.09
8	14349.24
16	24764.92
32	35569.76
64	24034.49
96	16752.52
128	15348.17



It is always a good idea to take a deeper look when things like barrier, wait, mutex and related library or kernel functions show up on the "perf top" list. It might be a sign of poor scaling or slowdown as the figure above shows.

5. Tuning

5.1. Advanced / Aggressive Compiler Flags

5.1.1. GNU compiler

The GNU set of compilers has full support for the Zen core architecture and all the normal gcc/gfortran flags will work as with any x86-64 processor. In the programming section some relatively safe flags are suggested. Below some flags controlling vectorization are tested and the performance are compared.

5.1.2. Intel compiler

The Intel compiler can generate code for the Zen architecture and make use of the AVX and AVX2 instructions. However, it's not guaranteed to work as Intel software does not really care about how it optimizes for other processors. To be on the safe side one must use flags that guarantee that the code can run on any x86-64. However, this yields very low performance. In the examples below the vector flags are set in order to produce 256 bits AVX/ AVX2 and FMA instructions

5.1.3. PGI (Portland) compiler

The current release of the PGI compiler (17.9) does not support the Zen architecture. It will however generate AVX, AVX2 and FMA instructions if enforced by the correct compiler flags.

5.1.4. Compilers and flags

One obvious test to compare compilers is by testing their generated code. One simple test is to compile the Fortran version of the general matrix-matrix multiplication code. This is a well known code and does represent a more general form of coding using nested loops iterating over 2-dimensional arrays.

The test is done by compiling the dgemm.f (standard Netlib reference code) with different Fortran compilers and linking with gfortran and run using a single core. The size is 10000x10000 yielding a footprint just over 2 GiB. Emphasis has been on vectorization, hence this is not a complete exploration of all possible optimization techniques like loop unrolling, fusing, prefetch, cache fit etc.

Table 15. Compiler performance

Compiler	Flags	Wall time [seconds]
gfortran	-O3 -msse4.2 -m3dnow	617.95
gfortran	-O3 -march=znver1 -mtune=znver1 -mfma	619.85
gfortran	-O3 -march=haswell -mtune=haswell -mfma	507.68
gfortran	-O3 -march=haswell -mtune=haswell -mfma -mavx2	510.09
gfortran	-O3 -march=broadwell -mtune=broadwell -mfma	509.83
gfortran	-O3 -march=znver1 -mtune=znver1 -mfma -mavx2 -m3dnow	619.42
gfortran	-O3 -mavx -mfma	546.01
gfortran	-O3 -mavx2 -mfma	545.06
gfortran	-Ofast -mavx2	554.99
pgfortran	-fast -Mipa=fast,inline	513.77
pgfortran	-O3 -Mvect=simd:256 -Mcache_align -fma	498.87
ifort	-O3	514.94

Compiler	Flags	Wall time [seconds]
ifort	-O3 -xAVX -fma	236.28
ifort	-O3 -xSSE4.2	214.11
ifort	-O3 -xCORE-AVX2 -fma	175.57

The most striking result from this simple test is that the Intel compiler does a good job generating code for the Zen architecture. The Intel Fortran compiler clearly outperforms the GNU Fortran compiler when it comes to this nested loop on matrices problems.

Another interesting result is that when using the gfortran compiler, optimizing for Zen processor (-march=znverl) yields lower performance than optimizing for Haswell. It looks like the code generator is more sophisticated for the Intel architectures than for the AMD architectures. From this we can conclude that care must be taken to not just rely on using tuned code for the current processor. In addition exploration of the effects of tuning for other architectures is often needed to find a sweet spot.

The Zen core is optimized for 128 bits vector operations and the SSE4.2 code is of that kind. The performance gives some indications of this , because the SSE code (128 bits only and no FMA) outperforms the AVX code (256 bits and FMA). However, they are both outperformed by the AVX2 instructions (256 bits and FMA).

Below is an extract of the assembly code generated by ifort with the flag -xCORE-AVX2, which yielded good performance. It's evident that there are AVX2 instructions by the fact that 256 bits wide ymm vector registers are used.

```
vmulpd      (%rdi,%r10,8), %ymm5,%ymm6              #260.27
vmulpd      64(%rdi,%r10,8), %ymm9, %ymm10          #260.27
vfmadd231pd (%rbx,%r10,8), %ymm5, %ymm3             #260.27
vmulpd      96(%rdi,%r10,8), %ymm12, %ymm13         #260.27
vaddpd      %ymm1, %ymm6,%ymm8                      #260.27
vmovupd     32(%r14), %ymm1                         #260.48
vfmadd231pd 64(%rbx,%r10,8), %ymm9, %ymm3           #260.27
vmulpd      32(%rdi,%r10,8), %ymm1, %ymm7           #260.27
vfmadd231pd 32(%rbx,%r10,8), %ymm1, %ymm2           #260.27
vaddpd      %ymm8, %ymm7,%ymm11                     #260.27
vfmadd231pd 96(%rbx,%r10,8), %ymm12, %ymm2          #260.27
vaddpd      %ymm11, %ymm10, %ymm14                  #260.27
```

In addition it's clear that the compiler can effectively vectorize the code by the fact that the instructions operate on packed vectors of double-precision (64 bits) floats. Instructions ending on "pd" (packed double) indicate that the instructions operate on a full vector of entries, in this case four double precision numbers (64 bits x 4 = 256 bits).

The PGI pgfortran also generates AVX2 and FMA instructions which is most probably why it performs better than gfortran. Even if the PGI compilers at the time of writing did not explicitly support the Zen architecture it does generate 256 bits AVX/AVX2 and FMA instructions when asked.

5.2. Single Core Optimization

5.2.1. Replace libm library

The single most obvious tuning is to replace the standard math library with the AMD optimized one. For some strange reason the savage benchmark runs magnitudes slower when using the original one. Install the libM library from AMD and just replace the symbolic link so it points to the correct optimized library and all applications that used libm will benefit from the optimized library.

```
cd /lib64; mv libm.so.6 libm.so.6.old
ln -s  /usr/local/lib64/libamdlibm.so libm.so.6
libm.so.6 -> /usr/local/lib64/libamdlibm.so
```

Not all applications will benefit as strongly as the savage benchmark, but basic scalar numerical functions like square root, logarithms, trigonometric etc will benefit strongly. See Table 10 for actual numbers.

Replacing the standard math library with the AMD optimized one, requires root access and might not be an option on all production systems. To do this in user space the user need to link with the AMD library and set the LD_LIBRARY path in order to pick up the libm AMD library first.

In some cases the global replacement of libm can cause unforeseen problems, in those cases the usage of the LD_LIBRARY_PATH environment variable and a symbolic link to the libamdlibm file or a rename is needed. In the though cases a LD_PRELOAD environment variable can be used.

5.3. Advanced OpenMP Usage

5.3.1. Tuning / Environment Variables

There are a range of environment variables that affect the OpenMP applications performance. The processor and NUMA node placement is explained in Section 5.4.1.

For the tests below a simple reference implementation of matrix matrix multiplication program (dgemm.f) was used unless stated in the table or figure.

Table 16. Variable for OpenMP tuning, scheduling policy

Variable	Performance (sec)
OMP_SCHEDULE = 'DYNAMIC'	769.68
OMP_SCHEDULE = 'STATIC'	738.57

Table 17. Variable for OpenMP tuning, wait policy

Variable	Performance (sec)
OMP_WAIT_POLICY = 'PASSIVE'	852.97
OMP_WAIT_POLICY = 'ACTIVE'	830.08

The numbers in each table measured relative to each other. Executables in each table have been build for the test and might differ, hence numbers cannot be compared across tables, only relative numbers within each table yield valid comparison.

The effect on performance varies from application to application, some are more sensitive to changes in setup, scheduling and policies than others. Only thorough testing will enable you to zero in on the optimal settings. Generally settings influencing memory access are the most important. Setting OMP_DISPLAY_ENV=VERBOSE will cause execution of the application to emit a list like the one given below at the start of execution.

```
OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
OMP_DYNAMIC = 'FALSE'
OMP_NESTED = 'FALSE'
OMP_NUM_THREADS = '32'
OMP_SCHEDULE = 'STATIC'
OMP_PROC_BIND = 'TRUE'
OMP_PLACES = '{0:32,64:32},{32:32,96:32}'
OMP_STACKSIZE = '0'
OMP_WAIT_POLICY = 'ACTIVE'
OMP_THREAD_LIMIT = '4294967295'
OMP_MAX_ACTIVE_LEVELS = '2147483647'
OMP_CANCELLATION = 'FALSE'
OMP_DEFAULT_DEVICE = '0'
OMP_MAX_TASK_PRIORITY = '0'
GOMP_CPU_AFFINITY = ''
```

```
GOMP_STACKSIZE = '0'
GOMP_SPINCOUNT = '30000000000'
OPENMP DISPLAY ENVIRONMENT END
```

More information about the different settings can be found at the GNU OpenMP web site [51]. The variables tested above are just two out of many that can have an effect on the application tested. In addition variables optimal for a certain number of threads might not be optimal for a different number of threads. Again thorough testing is needed to arrive close to optimal performance.

5.3.2. Thread Affinity

The processor and NUMA node placement is explained in Section 5.4.1. The variable GOMP_CPU_AFFINITY control thread binding to cores.

5.4. Memory Optimization

5.4.1. Memory Affinity (OpenMP/MPI/Hybrid)

The EPYC based system is a distinct NUMA system. The command numactl shows how the different memory banks are mapped and laid out.

```
-bash-4.2$ numactl  -H
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 64 65 66 67 68 69 70 71
node 0 size: 32668 MB
node 0 free: 29799 MB
node 1 cpus: 8 9 10 11 12 13 14 15 72 73 74 75 76 77 78 79
node 1 size: 32767 MB
node 1 free: 31617 MB
node 2 cpus: 16 17 18 19 20 21 22 23 80 81 82 83 84 85 86 87
node 2 size: 32767 MB
node 2 free: 31785 MB
node 3 cpus: 24 25 26 27 28 29 30 31 88 89 90 91 92 93 94 95
node 3 size: 32767 MB
node 3 free: 31399 MB
node 4 cpus: 32 33 34 35 36 37 38 39 96 97 98 99 100 101 102 103
node 4 size: 32767 MB
node 4 free: 20280 MB
node 5 cpus: 40 41 42 43 44 45 46 47 104 105 106 107 108 109 110 111
node 5 size: 32767 MB
node 5 free: 14751 MB
node 6 cpus: 48 49 50 51 52 53 54 55 112 113 114 115 116 117 118 119
node 6 size: 32767 MB
node 6 free: 8747 MB
node 7 cpus: 56 57 58 59 60 61 62 63 120 121 122 123 124 125 126 127
node 7 size: 32767 MB
node 7 free: 19613 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  16  32  32  32  32
  1:  16  10  16  16  32  32  32  32
  2:  16  16  10  16  32  32  32  32
  3:  16  16  16  10  32  32  32  32
  4:  32  32  32  32  10  16  16  16
  5:  32  32  32  32  16  10  16  16
  6:  32  32  32  32  16  16  10  16
  7:  32  32  32  32  16  16  16  10
```

The distance map clearly shows the four quadrants where two quadrants are present on each socket. With 8 memory banks distributed around the system it's important to schedule threads and ranks in an optimal way. As for all NUMA systems this can be left to the OS, but in many cases the kernel does not place the threads or move them in a none optimal way. A simple example is a threaded shared memory program that allocates the data structure in thread number one. Then all the data will (if it fits) reside in the first memory bank as this is local to the first thread. Consequently, when later on the program enters a parallel region most of the data will reside on a memory bank not local to the thread.

5.4.1.1. Thread placement

User placement of threads can be done in several ways, by command line tools and/or by means of environment variables. There are also more hidden ways within the various programming languages. It is far beyond the scope of this guide to go into this, but an application might show unexpected thread behaviour for the particular program.

5.4.1.1.1. Numactl

Usage of numactl is a very easy way of requesting threads to be placed on cores with different memory characteristics. Two common examples are:

```
numactl -l prog.x
numactl -i all /prog.x
```

The first one request that the threads are placed local to the memory, it also implies that memory will be allocated on the local NUMA memory bank (until this is completely filled). The second example will allocate memory in a round robin fashion on all the available NUMA memory banks (in this example "all" is used, but various subsets are possible).

5.4.1.1.2. Environment variables

The GNU OpenMP library [19] uses a range of different environment variables to control the placement of threads. The following table shows some of the most used. For a full listing and documentation refer to the GNU OpenMP library as there are many options with these variables.

Table 18. GNU OpenMP environment variables

Variable	Effect	Example
OMP_DISPLAY_ENV	If set to TRUE, the OpenMP version number and the values associated with the OpenMP environment variables are printed to stderr.	OMP_DISPLAY_ENV=VERBOSE
OMP_PROC_BIND	Specifies whether threads may be moved between processors. If set to TRUE, OpenMP threads should not be moved, if set to FALSE they may be moved.	OMP_PROC_BIND=TRUE
OMP_PLACES	The thread placement can be either specified using an abstract name or by an explicit list of the places.	OMP_PLACES=sockets
GOMP_CPU_AFFINITY	Binds threads to specific CPUs. The variable should contain a space-separated or comma-separated list of CPUs, individual or dash for ranges.	GOMP_CPU_AFFINITY="0 3 1-2"

In the verbose case of OMP_DISPLAY_ENV a listing like the one in Section 5.3.1 is emitted. The verbose variant adds the GNU OpenMP specific variables.

The thread placement can have a rather large impact on performance. The table below shows the effect on a simple stream memory bandwidth benchmark when run with the different placement settings, the PROC_BIND must be set to true.

Table 19. OMP environment variables effect

OMP_PLACES	Bandwidth [MiB/s]
threads	43689
cores	70105
sockets	116478

If you want to track the placement of threads during runtime, you can use the utility "htop". In addition if the OMP_DISPLAY_ENV is set to verbose the placement mapping is also written to stderr.

5.4.1.2. Rank placement

For MPI programs the placement of the ranks can be controlled by using the runtime environment variables or by options of mpirun/mpiexec.

In the case of OpenMPI's mpirun you could use the “ –bind-to” flag like this:

```
--bind-to hwthread
--bind-to core
--bind-to socket
--bind-to numa
--bind-to l2cache
--bind-to l3cache
```

This will pin the ranks to cores according to the value of the option. If pinned to a core it will not move from that core during the lifetime of the execution. If it's pinned to a NUMA memory bank it can be moved to any core local to that NUMA bank. It's not always obvious which strategy will yield optimum performance. A bit of trial and error is often required.

5.4.2. Memory Allocation (malloc) Tuning

With an architecture like the EPYC, with 8 NUMA memory banks per node [8], memory allocation is important, especially when running multi-threaded, shared memory applications. The system tested is a two socket node, hence 8 NUMA banks.

In some cases where the data is allocated differently from what the current multi-threaded region finds optimal the kernel will try to migrate the processes. This takes a lot of cpu and you'll spot it immediately when running top. Something like this is generally not good. The only process that should use CPU is the user process.

```
PID USER   PR  NI   VIRT  RES  SHR S  %CPU %MEM     TIME+ COMMAND
 28 root   rt   0     0     0   0 S  25.0  0.0   1:45.23 migration/4
 69 root   rt   0     0     0   0 S  15.5  0.0   0:25.82 migration/12
395 root   rt   0     0     0   0 S  10.2  0.0   0:14.11 migration/76
355 root   rt   0     0     0   0 S   9.5  0.0   0:51.62 migration/68
335 root   rt   0     0     0   0 R   6.6  0.0   1:05.99 migration/64
110 root   rt   0     0     0   0 S   4.6  0.0   0:14.72 migration/20
  8 root   rt   0     0     0   0 S   3.9  0.0   1:00.62 migration/0
 48 root   rt   0     0     0   0 S   3.9  0.0   0:51.04 migration/8
 10 root   20   0     0     0   0 S   3.6  0.0  22:44.25 rcu_sched
 89 root   rt   0     0     0   0 S   2.3  0.0   0:22.74 migration/16
435 root   rt   0     0     0   0 S   2.3  0.0   0:08.64 migration/84
375 root   rt   0     0     0   0 S   1.6  0.0   0:22.57 migration/72
```

Seeing this list of processes is an indication that something is wrong and action is required.

There are two more tools that are handy when monitoring memory allocation and processor placements. These are numactl and numastat. The numactl command shows how much of the memory in each of the NUMA nodes is actually allocated, an example is given below (numactl -H):

```
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 64 65 66 67 68 69 70 71
node 0 size: 32668 MB
node 0 free: 1201 MB
node 1 cpus: 8 9 10 11 12 13 14 15 72 73 74 75 76 77 78 79
node 1 size: 32767 MB
node 1 free: 277 MB
node 2 cpus: 16 17 18 19 20 21 22 23 80 81 82 83 84 85 86 87
node 2 size: 32767 MB
node 2 free: 6904 MB
node 3 cpus: 24 25 26 27 28 29 30 31 88 89 90 91 92 93 94 95
node 3 size: 32767 MB
node 3 free: 7659 MB
node 4 cpus: 32 33 34 35 36 37 38 39 96 97 98 99 100 101 102 103
node 4 size: 32767 MB
node 4 free: 6652 MB
node 5 cpus: 40 41 42 43 44 45 46 47 104 105 106 107 108 109 110 111
node 5 size: 32767 MB
node 5 free: 3402 MB
node 6 cpus: 48 49 50 51 52 53 54 55 112 113 114 115 116 117 118 119
node 6 size: 32767 MB
node 6 free: 10058 MB
node 7 cpus: 56 57 58 59 60 61 62 63 120 121 122 123 124 125 126 127
node 7 size: 32767 MB
node 7 free: 2131 MB
```

It's clear that the allocated memory is distributed reasonably evenly over all NUMA nodes. Running again with the numactl settings '-l' for local and '-i all' for interleaved over all NUMA nodes will show how the memory allocation is distributed. Doing this repeatedly (watch -n 1 numactl -H) during the allocation phase of the application can give an insight of how memory are allocated on the different NUMA nodes.

Numastat is a tool to show per-NUMA-node memory statistics for processes and the operating system. Below is shown an example of a numastat output (only 3 out of 8 NUMA nodes are shown):

```
                       node0           node1           node2
numa_hit           140742970       145695513       135145845
numa_miss             951978          889448          758010
numa_foreign          781077          342810          471730
interleave_hit        106502          130910          129820
local_node         140736519       145567338       135017372
other_node            958429         1017623          886483
```

The numbers to monitor are the "numa_miss" and "numa_foreign" as they show memory accesses to data residing in NUMA nodes that are not local. Accesses to none local NUMA nodes have higher access times and lower bandwidth and generally are bad for performance.

NUMA awareness is important for OpenMP (shared memory) applications and also for MPI applications where more than one rank is running per node. For a running process the memory can be local or remote if the running process is moved to another core. This can (will) happen is there is no processor binding.

There are some measures to take to try to keep data on local NUMA nodes. The simplest is to use the "numactl -l" command to allocate on the local memory. However, if the application does the allocation on a single thread then the data will be allocated on the NUMA nodes local to this thread, which can have an adverse effect.

Table 20. Memory allocations and core binding

Settings	Wall time [seconds]
numactl -l	1166.15
numactl -i all	1245.62

Settings	Wall time [seconds]
OMP_PROC_BIND=TRUE; OMP_PLACES=sockets	701.19
OMP_PROC_BIND=TRUE; OMP_PLACES=cores	879.63
OMP_PROC_BIND=TRUE; OMP_PLACES=threads	854.64

There are some other environment variables to bind the threads to specific cores, however, these are more complex and require more in-depth discussion than possible in this guide.

5.4.3. Using Huge Pages

The Transparent Huge Pages (THP), a Linux memory management system that is supported by newer kernels, provide us with a simple way of using huge pages. The file /sys/kernel/mm/transparent_hugepage/enabled contains information about the current setting of THP.

```
cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

Where the word in brackets is the currently selected setting.

The status of the system's memory can be obtained by displaying the file : /proc/meminfo, an example (a large portion is skipped):

```
MemTotal:       263921288 kB
MemFree:        170687204 kB
MemAvailable:   201953992 kB
Buffers:                0 kB
Cached:         31003020 kB
AnonHugePages:  58599424 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      232524 kB
DirectMap2M:    10147840 kB
DirectMap1G:    257949696 kB
```

If the THP is disabled the AnonHugePages show zero.

Enabling or disabling THP can have an effect on application performance, it is not always advised to have it turned on. The following table shows a simple dgemm test with hugepages enabled and disabled. As this requires root access a trade-off must be used since it cannot be changed for each application.

Table 21. Transparent Huge Pages performance

Settings	Wall time [seconds]
always	806.51
never	1225.66

5.4.4. Monitoring NUMA pages

Automatic NUMA Balancing is now implemented in most kernels. Automatic NUMA Balancing migrates data on demand to memory nodes that are local to the CPU accessing that data. Depending on the workload, this can dramatically boost performance when using NUMA hardware.

The kernel keeps track of the NUMA pages and the information is available in the /proc/vmstat/ directory, example below:

```
$cat /proc/vmstat | grep NUMA
numa_hit 1441838287
numa_miss 10935731
numa_foreign 10935731
numa_interleave 1182389
numa_local 1440882623
numa_other 11891395
numa_pte_updates 24963187675
numa_huge_pte_updates 47638501
numa_hint_faults 557363093
numa_hint_faults_local 482115590
numa_pages_migrated 922781927
```

The numbers of importance are the ones about faults and about migrated pages. The meaning of some of the numbers above is explained in the list below:

NUMA metrics

- numa_hit: Number of pages allocated from the node the process wanted.
- numa_miss: Number of pages allocated from this node, but the process preferred another node.
- numa_foreign: Number of pages allocated on another node, but the process preferred this node.
- numa_local: Number of pages allocated from this node while the process was running locally.
- numa_other: Number of pages allocated from this node while the process was running remotely (on another node).
- numa_interleave_hit: Number of pages allocated successfully with the interleave strategy.

The following table shows the effect on NUMA pages when running with different thread binding.

Table 22. Effect on NUMA pages

Binding	Performance [Mflops]	numa_hits	numa_ hint_faults	numa_ hint_faults_local	numa_ pages_migrated
None	150487	48705	103660	81686	1950659
OMP_PROC=true	77277	36815	169381	154593	612369
numactl -l	136236	28575	92057	68816	2696390
numactl -i all	104236	23816	0	0	0
GOMP_CPU_ AFFINITY=0-63	150315	31895	93997	80252	750545

The NUMA hits is an important number. The faults and migrated pages change with thread layout. The value of the faults increases while the migrated pages drops, the effect on performance is not always predictable.

5.5. Possible Kernel Parameter Tuning

5.5.1. NUMA control

There are a battery of kernel parameters controlling the systems behaviour. When building a kernel there are a a lot of compromises and as always one set is not always optimal. Some of the settings might be optimal for a single application and counterproductive for another. Finding the best set for a multiuser, multiapplication general purpose HPC system is often challenge and mostly compromise. Tests below have only been done for a single application and is acting a reference point to start.

Many of the settings deal with NUMA control. The kernel can control how NUMA banks and pages are allocated, deallocated or moved. The defaults might not always suite computational loads. Statistics for a NUMA system data can be extracted from the /proc file system.

The following table gives an overview of some tested parameters. The benchmark OpenMP version of BT from the NPB benchmark suite (see Section 3.1.2.1), running with 64 threads on 64 cores, was selected as the test benchmark. No core affinity and thread binding was set. If these options would have been set, the picture might look different, - see Section 5.4.4 for more on this.

Table 23. Kernel NUMA parameters tests (default values in boldface)

Kernel parameter	**Value**	**BT performance (performance metric for the BT benchmark, higher is better)**
/proc/sys/kernel/numa_balancing	0	53222
	1	148366
/proc/sys/kernel/numa_balancing_scan_delay_ms	500	139985
	1000	149104
	2000	147998
	5000	131316
	10000	147540
/proc/sys/kernel/numa_balancing_scan_size_mb	64	154856
	128	146851
	256	154487
	512	158354
	1024	133910
	2048	151780
	4096	139636
	8192	141672
	16385	158746
/proc/sys/kernel/ numa_balancing_scan_period_min_ms	250	147470
	500	151427
	1000	160915
	5000	149883
	10000	129752

With many parameters to optimize and unknown cross effects the job of finding the optimal setting can be rather large and time consuming. Normally the defaults do a reasonably good job. It's possible to do some manual tuning to get a bit more performance. For memory intensive HPC applications one might expect that NUMA kernel parameter tuning will have highest return of effort.

The recomendation is to check the defaults and only do a limited set of changes for the NUMA kernel parameters.

5.5.2. Scheduling control

There are a number of parameters controlling the kernel's scheduling of processes/threads. Scheduling documentation can be found in [52].

The table below shows some selected scheduling parameters tested. The same BT benchmark was used as in the table above, with NUMA control.

Table 24. Kernel scheduling parameters tests

Kernel parameter	Value	BT performance
/sys/kernel/mm/transparent_hugepage/enabled	**always**	149033
	never	141652
/proc/sys/kernel/sched_tunable_scaling	0	152828
	2	130206
/proc/sys/kernel/sched_rr_timeslice_ms	500	121845
	1000	160231
	3000	150418
/proc/sys/kernel/sched_wakeup_granularity_ns	1000000	143662
	10000000	155970
	50000000	143894
/proc/sys/kernel/sched_latency_ns	12000000	154494
	24000000	154590
	32000000	130271
	48000000	110926
/proc/sys/kernel/sched_migration_cost_ns	100000	141607
	250000	144527
	500000	145611
	750000	151107
	1000000	138100
	1500000	149257

Some of these parameters can have a significant impact on performance of different applications. Which parameters do have a significant impact on the application tested is not easy to guess up front. More information can be found at Suse's web pages, scheduling [53].

The recomendation is to keep hugepages enabled for HPC load. The default kernel parameters may not be optimal for HPC load as servers are often used for web services and database servers, both which have a different behavior from HPC.

6. Debugging

6.1. Available Debuggers

Several debuggers exist, of which the GNU debugger gdb comes with the Operating system. A commercial state-of-the-art debugger is DDT from Allinea [56]. This debugger has support for AMD EPYC. See reference for more information, Yet another commercial debugger is TotalView from Rogue Wave [57].

This is not a tutorial for the GNU debugger gdb, but it's interesting to note that by halting a numeric kernel one might peek into the instructions executed and look for effective instructions that operate on vectors. Below we see fully populated vectors (*pd packed double or *ps packed single) instructions:

```
(gdb) set disassembly-flavor att
(gdb) disassemble
Dump of assembler code for fuNction dgemm_kernel:
   0x00002aaaaafbdf97 <+407>: add    $0x60,%rsi
   0x00002aaaaafbdf9b <+411>: vmulpd %ymm0,%ymm3,%ymm14
   0x00002aaaaafbdf9f <+415>: vpermpd $0xb1,%ymm0,%ymm0
   0x00002aaaaafbdfa5 <+421>: vmulpd %ymm0,%ymm1,%ymm7
   0x00002aaaaafbdfa9 <+425>: vmovups -0x60(%rsi),%ymm1
   0x00002aaaaafbdfae <+430>: vmulpd %ymm0,%ymm2,%ymm11
   0x00002aaaaafbdfb2 <+434>: vmovups -0x40(%rsi),%ymm2
   0x00002aaaaafbdfb7 <+439>: vmulpd %ymm0,%ymm3,%ymm15
   0x00002aaaaafbdfbb <+443>: vmovups -0x20(%rsi),%ymm3
   0x00002aaaaafbdfc0 <+448>: vmovups -0x60(%rdi),%ymm0
   0x00002aaaaafbdfc5 <+453>: vfmadd231pd %ymm0,%ymm1,%ymm4
   0x00002aaaaafbdfca <+458>: vfmadd231pd %ymm0,%ymm2,%ymm8
```

The assembly listing comes in two flavours, for Intel and AT&T [58].

6.2. Compiler Flags

As always the -g flag is used to request the compiler to insert debugging information. With GNU it can also be used together with -O. However, a few less commonly debug options are:

Table 25. Debugging compiler flags (gnu)

Flag	Description
-g	Produce debugging information in the operating system's native format
-ggdb	Produce debugging information for use by GDB.
-glevelN	Request debugging information and also use level to specify how much information. The default level is 2.

Several other options exist. Please refer to gcc documentation or man gcc to learn more about these less common options.

Further documentation

Books

[1] *Best Practice Guide - Intel Xeon Phi, January 2017, http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi-1.pdf .*

Websites, forums, webinars

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

[3] *AMD EPYC Data Sheet, https://www.amd.com/system/files/2017-06/AMD-EPYC-Data-Sheet.pdf.*

[4] *AMD EPYC Server Processors, https://www.amd.com/en/products/epyc-server [https://www.amd.com/en/products/epyc-server%20].*

[5] *EPYC wiki page, https://en.wikipedia.org/wiki/Epyc.*

[6] *EPYC 7601 documentation, https://en.wikichip.org/wiki/amd/epyc/7601 .*

[7] *AMD Infinity Fabric [https://en.wikichip.org/wiki/amd/infinity_fabric] .*

[8] *AMD EPYC 7601, https://www.amd.com/en/products/cpu/amd-epyc-7601.*

[9] *Intel Xeon Platinum 8180 Processor, https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38_5M-Cache-2_50-GHz.*

[10] *"Sizing Up Servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 - The Server CPU Battle of the Decade?", Memory Subsystem:Bandwidth [https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc-7000-cpu-battle-of-the-decade/12].*

[11] *AOCC compilers, website [http://developer.amd.com/amd-aocc/].*

[12] *Wikichip on Zen [https://en.wikichip.org/wiki/amd/microarchitectures/zen].*

[13] *Intel documention, svml [https://software.intel.com/en-us/node/524288].*

[14] *Intel documention, Latency checker [https://software.intel.com/en-us/articles/intelr-memory-latency-checker].*

[15] *Intel, MKL link line advisor [https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor].*

[16] *Netlib, Basic linear Algebra [http://www.netlib.org/blas/index.html].*

[17] *OpenBLAS, Basic linear Algebra [https://www.openblas.net/].*

[18] *GSL, GNU Scientific Library [https://www.gnu.org/software/gsl/].*

[19] *GNU, OpenMP [https://gcc.gnu.org/wiki/openmp].*

[20] *FFTW, Fast Fourier Transform library [http://www.fftw.org/].*

[21] *OpenSBLI website, [https://opensbli.github.io].*

[22] *OpenSBLI build instructions with GNU 7.2.0, [https://github.com/hpc-uk/archer-benchmarks/blob/master/apps/OpenSBLI/source/AMD_Naples_build_gcc.md].*

[23] *OpenSBLI build instructions with Intel17, [https://github.com/hpc-uk/archer-benchmarks/blob/master/apps/OpenSBLI/source/AMD_Naples_build_intel17.md].*

[24] *OpenSBLI job example with TGV512ss, [https://github.com/hpc-uk/archer-benchmarks/blob/master/apps/OpenSBLI/TGV512ss/run/AMD_Naples].*

[25] *OpenSBLI job example with TGV1024ss, [https://github.com/hpc-uk/archer-benchmarks/tree/master/apps/OpenSBLI/TGV1024ss/run/AMD_Naples].*

[26] *OpenSBLI run results example with TGV512ss, [https://github.com/hpc-uk/archer-benchmarks/tree/master/apps/OpenSBLI/TGV512ss/results/AMD_Naples].*

[27] *OpenSBLI run results example with TGV1024ss, [https://github.com/hpc-uk/archer-benchmarks/tree/master/apps/OpenSBLI/TGV1024ss/results/AMD_Naples].*

[28] *CASTEP website, [http://www.castep.org/CASTEP/CASTEP].*

[29] *CASTEP build instructions with GNU 7.2.0, [https://github.com/hpc-uk/build-instructions/blob/master/CASTEP/AMD_Naples_18.1.0_gcc7_OMPI.md].*

[30] *CASTEP job example, https://github.com/hpc-uk/archer-benchmarks/blob/master/apps/CASTEP/al3x3/run/AMD_Naples/job_castep_Al3x3.slurm.*

[31] *CASTEP run results example with al3x3, [https://github.com/hpc-uk/archer-benchmarks/tree/master/apps/CASTEP/al3x3/results/AMD_Naples].*

[32] *GROMACS website, People [http://www.gromacs.org/About_Gromacs/People].*

[33] *GROMACS website, [http://www.gromacs.org/About_Gromacs].*

[34] *GROMACS build instructions with GNU 7.2.0, [https://github.com/hpc-uk/build-instructions/blob/master/GROMACS/AMD_Naples_2018.2_gcc7.md].*

[35] *GROMACS job example, https://github.com/hpc-uk/archer-benchmarks/blob/master/apps/GROMACS/1400k-atoms/run/AMD_Naples/job_gromacs.slurm.*

[36] *GROMACS run results example with 1400k-atoms, [https://github.com/hpc-uk/archer-benchmarks/tree/master/apps/GROMACS/1400k-atoms/results/AMD_Naples].*

Manuals, papers

[37] *PRACE Public Deliverable 7.6 Best Practice Guides for New and Emerging Architectures, http://www.prace-ri.eu/IMG/pdf/D7.6_4ip.pdf.*

[38] *McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995. http://www.prace-ri.eu/IMG/pdf/D7.6_4ip.pdf.*

[39] *Portland compilers, documentation. [https://www.pgroup.com/resources/docs/18.5/x86/pgi-ref-guide/index.htm].*

[40] *LLVM Clang, documentation. [http://developer.amd.com/wordpress/media/2017/04/Clang-the-C-CPP-Compiler-AOCC-LLVM-1.pdf].*

[41] *LLVM Flang, documentation. [http://developer.amd.com/wordpress/media/2017/04/DragonEgg-the-Fortran-compiler-AOCC-LLVM-1.pdf].*

[42] *Intel intrinsics, [http://kylehegeman.com/blog/2013/12/27/using-intrinsics/].*

[43] *AMD libraries, developer site. [http://developer.amd.com/amd-cpu-libraries].*

[44] *Perf utility, manual. [https://perf.wiki.kernel.org/index.php/Main_Page].*

[45] *Perf utility, tutorial. [https://perf.wiki.kernel.org/index.php/Main_Page].*

[46] *Savage benchmark. [https://celestrak.com/columns/v02n04/].*

[47] *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].*

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

[49] *NPB Benchmark. The HPC NPB benchmark [https://www.nas.nasa.gov/publications/npb.html].*

[50] *AMD µProf AMD µProf [http://developer.amd.com/amd-%CE%BCprof/] .*

[51] *GNU OpenMP library [https://gcc.gnu.org/onlinedocs/libgomp/] .*

[52] *Scheduling control [https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.taskscheduler.html] .*

[53] *NUMA kernel parameter tuning [https://www.suse.com/documentation/sled-12/book_sle_tuning/data/sec_tuning_taskscheduler_cfs.html].*

[54] *AMD µProf Manual AMD µProf Manual [http://developer.amd.com/wordpress/media/2013/12/AMDuprof-User_Guide.pdf] .*

[55] *Alliena Performance Reports https://www.allinea.com/products/allinea-performance-reports.*

[56] *Allinea Dynamic Debugging Tool (DDT) https://www.allinea.com/products/ddt .*

[57] *Rogue Wave Totalview [https://www.roguewave.com/products-services/totalview] .*

[58] *Gnu Debugger, assembly listing flavours. [http://visualgdb.com/gdbreference/commands/set_disassembly-flavor].*