High Performance Linpack Benchmark on AMD EPYC™ Processors

This document details running the High Performance Linpack (HPL) benchmark using the AMD xhpl binary.

HPL Implementation:
The HPL benchmark presents an opportunity to demonstrate the optimal combination of multithreading (via the OpenMP library) and MPI for scientific and technical high-performance computing on the EPYC architecture. For MPI applications where the per-MPI-rank work can be further parallelized, each L3 cache is an MPI rank running a multi-threaded application. This approach results in fewer MPI ranks than using one rank per core, and results in a corresponding reduction in MPI overhead. The ideal balance is for the number of threads per MPI rank to be less than or equal to the number of CPUs per L3 cache. The exact maximum thread count per MPI rank depends on both the specific EPYC SKU (e.g. 32 core parts have 4 physical cores per L3, 24 core parts have 3 physical cores per L3) and whether SMT is enabled (e.g. for a 32 core part with SMT enabled there are 8 CPUs per L3).

HPL performance is primarily determined by DGEMM performance, which is in turn primarily determined by SIMD throughput. The Zen microarchitecture of the EPYC processor implements one SIMD unit per physical core. Since HPL is SIMD limited, when SMT is enabled using a second HPL thread per core will not directly improve HPL performance. However, leaving SMT enabled may indirectly allow slightly higher performance (1% - 2%) since the OS can utilize the SMT siblings as needed without pre-empting the HPL threads.

This example uses OpenMPI for the MPI library. The Hardware Locality library (hwloc) allows OpenMPI to automatically extract platform topology information for rank assignment and binding. The open source BLIS library is used for DGEMM. This library can be optionally configured with threading support (POSIX threads or OpenMP). The library comes with several pre-built optimized kernels including an optimized AVX2 kernel for Zen. Multi-threaded OpenBLAS can also be used and will offer similar performance.

There is very little communication / synchronization between the OpenMP threads and OpenMPI ranks in HPL, thus alternate implementations using the OpenMPI default NUMA node binding or using a single-thread xhpl with OpenMPI ranks bound to cores will yield similar performance.

Note for 24/16/8 core EPYC SKUs (7451, 7401, 7401P, 7351, 7351P, 7301, 7251):
The example in this ap-note is based on a 32 core EPYC processor in a 2P server with a total of 256 GB of memory. The HPL.dat file and any manual CPU binding (if used) will need to be modified for 24, 16 and 8 core processors, for single processor machines, and for machines with less than 256 GB of memory.

Also, Linux kernels prior to 4.14 (or Red Hat Enterprise Linux / CentOS 7 kernels prior to 3.10.0-693.6.1) exhibit a bug whereby the kernel generated map of which logical CPUs share which L3 cache is incorrectly generated for the 24, 16 and 8 core EPYC processors. This bug will cause hwloc to generate a warning when started and will prevent automatic OpenMPI binding to L3. A script is available from AMD with manual binding support, and an updated kernel is available on most common distributions (with kernel versions 3.10, 4.4, 4.8, 4.10, or 4.13) to address this bug.

An example of the error message is on the following page:

```
****************************************************************************
* hwloc 1.11.2 has encountered what looks like an error from the operating system.
*
* L3 (cpuset 0x60000060) intersects with NUMANode (P#0 cpuset 0x3f00003f) without
inclusion!
* Error occurred in topology.c line 1046
*
* The following FAQ entry in the hwloc documentation may help:
*   What should I do when hwloc reports "operating system" warnings?
* Otherwise please report this error message to the hwloc user's mailing list,
* along with the output+tarball generated by the hwloc-gather-topology script.
****************************************************************************
```

Build dependencies:

g++ (gcc-c++ on RHEL)
gfortran (if building the xhpl binary from source)
libnuma-dev (Ubuntu - for building OpenMPI)
hwloc
libhwloc-dev (Ubuntu) or hwloc-devel (RHEL) (for building OpenMPI)

Additional package sources:

OpenMPI: https://www.open-mpi.org/software/ompi/v3.0
BLIS (if building the xhpl binary from source): https://developer.amd.com/amd-cpu-libraries/blas-library

System configuration steps:

On Red Hat Enterprise Linux / CentOS, if running ktune or tuned, the recommended tuned profile for HPL is throughput-performance.

On Ubuntu, insure that the ondemand or performance cpufreq governor is used.

AMD recommends the following settings in the UEFI Firmware for this benchmark (different platform vendors may use different nomenclature for these settings):

> Set the determinism slider to Performance Determinism mode
> Enable manual cTDP override and set the cTDP to 200 watts
> Insure Core Performance Boost is enabled
> Insure Global C State Control is enabled if SMT is enabled (and do not boot with idle=poll)
> Reduce memory speed from 2667 MTS to 2400 MTS (if boost is enabled)

A note on boost and memory speed: Assuming a relatively cool environment (25 deg. C) and 2400 MTS memory, AMD's power efficient AVX2 implementation in the Zen microarchitecture will sustain the SKU-specific all core boost frequency on most platforms even when running power-hungry SIMD codes like HPL. Per-socket, memory at 2667 MTS consumes about 15 watts more than memory at 2400 MTS. Reducing power in the memory controllers allows the CPUs to consume more of the available power. Increasing the cTDP will further increase the headroom for all cores to sustain maximum boost. The expected performance difference on HPL from 2400 MTS to 2667 MTS is about 2% (2400 MTS yields higher performance on HPL with boost enabled). If you must run with boost disabled for some other reason, keep the memory speed at the BIOS default. Dual-rank memory will also increase performance by about 2% compared to single rank memory.

Build steps:
The included xhpl binary was built using the AMD optimized BLIS Linear Algebra library and OpenMPI. This binary expects libmpi.so in /opt/openmpi/lib. Rebuilding the xhpl binary is not covered here, however if you choose to rebuild this file, you will need to edit the MPxxx, and LAxxx directories in your architecture-specific Makefile to match the installed locations of your MPI and Linear Algebra library. For BLIS, use the F77 interface with `F2CDEFS = -DAdd__ -DF77_INTEGER=int -DStringSunStyle`. To generate a progress report while running, also set `HPL_OPTS = -DHPL_PROGRESS_REPORT`.

Use of the Intel MKL BLAS library is not recommended as this library is intentionally crippled when running on non-Intel machines.

The following steps will install packages to /opt – you can change this as needed with `--prefix={/path you want}` on the `./configure` commands:

1) BLIS
   The BLIS library is only required if building the xhpl binary from source. Using the pre-compiled BLIS binary from the previous page is the simplest solution however if you still wish to build BLIS from source, the following instructions will help. Using OpenBLAS is an acceptable alternative however instructions to build OpenBLAS are not covered here.
   a) Extract the source tarball and cd into the created directory
   b) `./configure --prefix=/opt/blis --enable-threading=openmp zen`
   c) `make –j 32`
   d) `sudo make install`

2) OpenMPI
   a) Extract the source tarball and cd into the created directory
   b) `CFLAGS=”-O3 -march=native” ./configure --prefix=/opt/openmpi`
   c) `make –j 32`
   d) `sudo make install`
   e) `sudo ln –s /opt/openmpi/lib/libmpi.so.40.0.0 /opt/openmpi/lib/libmpi.so.20`

Step 2e above is only necessary if building OpenMPI 3.x – it is not required if building OpenMPI 2.x

If in step 1b you receive an error about zen being an unsupported configuration, the likely cause is that you used a release from the master branch of the BLIS project instead of downloading the head of the AMD branch. You can use haswell instead of zen however the performance will be slightly reduced. Or use the pre-compiled library.

Running on a 2P EPYC system:

The benchmark can be run with `./run_hpl.sh` (Please refer to the appendix for a listing of the script). The benchmark will take about an hour to run on a 2P 32 core machine with 256GB.

If you encounter the hwloc error mentioned above and are unable to update your kernel, please contact AMD for a manual binding script (when contacting AMD, please advise if you are using a Dell or non-Dell system and which EPYC SKU you are using).

Modifications for a 1P EPYC system:

1) Edit the `HPL.dat` file to change the Ps line from 4 to 2. This will change the number of process grids from 16 (4 x 4) to 8 (2 x 4). You may also need to adjust the problem size N to account for less total system memory. See below for more details.
2) Edit the `run_hpl.sh` script to change the number of MPI processes from 16 to 8

Configuring HPL.dat
HPL.dat defines the problem (matrix size) and some parameters of how the solution is computed for the benchmark. Some parameters you can adjust:

Number of Process Grids (P, Q):
The number of expected process grids (P x Q) must match the number of MPI ranks that will be started to complete the calculations. P and Q should be as close to each other as possible. If the numbers cannot be equal, Q should be larger. For this hybrid OpenMPI + OpenMP implementation on EPYC, the number of process grids should match the total number L3 caches in the system. For a 2P system there are 16 L3 caches thus P = Q = 4 (P x Q = 16). For a 1P system use P = 2, Q = 4 (P x Q = 8).

Block size (NB):
HPL uses the block size NB for the data distribution as well as for the computational granularity. A block size of 232 or 240 is optimal for EPYC. It should not normally be necessary to adjust this parameter.

Problem size (N):
HPL performs calculations on an N x N array of double precision elements. Each double precision element requires 8 bytes. Thus the memory consumed for a problem size of N is $8N^2$. Assuming K machines with equal memory J, the total available memory M is K x J. The problem size should be set to a multiple of the block size (above) times the total number of cores on all machines, and should be the largest size possible without exceeding 90% of the available memory (N <= sqrt(M/8)). For example, for a single machine with 256GB of memory and a block size of 240, N should be 168,960 elements.

Expected results:

Maximum performance for Zen AVX2 is 16 DP flops per two core clock cycles (8 FLOPs per cycle). A 2P EPYC 7601 system (2.7 GHz all cores boost) has a theoretical maximum performance of 64 x 8 x 2.7GHz or 1,382 DP GFLOPS.

The expected efficiency (calculated as actual GFLOPS / theoretical GFLOPS) should be in the 84% - 86% range for a properly configured and operating 2P system and 85% - 87% range for a 1P system.

Expected results for a 2P EPYC 7601 system are 1150 – 1200 GFLOPS, and for a 1P EPYC 7601 system 580 – 600 GFLOPS. Below are example test results for a 2P system and 1P system on Ubuntu 16.04 with SMT enabled:

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR12R2R4      168960   232     4     4            2785.72              1.154e+03
HPL_pdgesv() start time Mon Sep 11 07:59:28 2017

HPL_pdgesv() end time   Mon Sep 11 08:45:53 2017

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0028913 ...... PASSED
================================================================================
```

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR12R2R4       84480   232     2     4             680.26              5.909e+02
HPL_pdgesv() start time Sun Apr  8 01:59:03 2018

HPL_pdgesv() end time   Sun Apr  8 02:10:23 2018

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0029770 ...... PASSED
================================================================================
```

Appendix

Multi-threaded (OpenMP) BLIS DGEMM parallelization:
When configured with multi-threading enabled, the BLIS library is controlled by the following environment variables. The DGEMM parallelization at each shared L3 cache is configured as follows:

BLIS_JC_NT=1	(No outer loop parallelization)
BLIS_IC_NT=4	(4 2nd level threads – one per core in the shared L3 cache domain)
BLIS_JR_NT=1	(No 4th level threads)
BLIS_IR_NT=1	(No 5th level threads)

For more information, check here: https://github.com/flame/blis/wiki/Multithreading

HPL.dat:
```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
168960       Ns
1            # of NBs
232          NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
4            Ps
4            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
2            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
2            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
1            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

run_hpl.sh script:

```
#! /bin/bash

#
# OpenMP environment variables
#

# Use explicit process binding
export OMP_PROC_BIND=TRUE

# Have OpenMP ignore SMT siblings if SMT is enabled
export OMP_PLACES=cores

# Determine number of threads per L3 cache. Count cores per socket and
# divide by 8 (8 L3s per socket)
cores=$( lscpu | grep "Core(s) per socket" | awk '{print $4}' )
echo "$cores core processors detected"
cores=$(( $cores / 8 ))
export OMP_NUM_THREADS=$cores

#
# BLIS library environment variables
#

# DGEMM parallelization is performed at the 2nd innermost loop (IC)
export BLIS_IR_NT=1
export BLIS_JR_NT=1
export BLIS_IC_NT=$cores
export BLIS_JC_NT=1

#
# OpenMPI settings
#

# Launch 16 processes (one per L3 cache, two L3 per die, 4 die per socket, 2 sockets)
mpi_options="-np 16"

# Use vader for Byte Transfer Layer
mpi_options+=" --mca btl self,vader"

# Map processes to L3 cache
# Note if SMT is enabled, the resource list given to each process will include
# the SMT siblings! See OpenMP options above.
mpi_options+=" --map-by l3cache"

# Show bindings
mpi_options+=" --report-bindings"


/opt/openmpi/bin/mpirun $mpi_options xhpl
```