Optimizing a 3D multi-physics continuum mechanics code for the HPE Apollo 80 System

Howard Pritchard, Vincent Graziano, David Nystrom, Brandon Smith, Brian Gravelle
Los Alamos National Laboratory
Los Alamos, NM
{howardp, vgraziano, wdn, bmsmith, gravelle}@lanl.gov

***Abstract*—We present results of a performance evaluation of a LANL 3D multi-physics continuum mechanics code – Pagosa – on an HPE Apollo 80 system. The Apollo 80 features the Fujitsu A64FX ARM processor with Scalable Vector Extension (SVE) support and high bandwidth memory. This combination of SIMD vector units and high memory bandwidth offers the promise of realizing a significant fraction of the theoretical peak performance for applications like Pagosa. In this paper we present performance results of the code using the GNU, ARM, Fujitsu, and CCE compilers, analyze these compilers' ability to optimize performance critical loops when targeting the SVE instruction set, and describe code modifications to improve the performance of the application on the A64FX processor.**

***Index Terms*—component, formatting, style, styling, insert**

I. INTRODUCTION

We present results of a performance evaluation of a LANL 3D multi-physics continuum mechanics code - Pagosa [1] - on an HPE Apollo 80 system [2]. This system features Fujitsu A64FX processors, 32 GiB of high bandwidth memory (HBM2), and an Nvidia IB HDR network. The system also comes with the HPE Cray Compiling Environment (CCE).

The Fujitsu A64FX processor supports the ARMv8.2-A ISA in addition to Scalable Vector Extension (SVE) [3] and HBM2 memory. The processor operates at 1.8 GHz, leading to a peak floating point performance of 2.8 Tflops double precision and 5.5 Tflops single precision when using SVE SIMD (512b) vector instructions. Each processor has four NUMA nodes with 12 compute cores. Each core has 64 KiB L1 data and instruction caches. The cores in a NUMA node share an 8 MiB L2 cache. Each NUMA node is connected directly to one of the HBM2 memory banks. The HBM2 memory subsystem offers a memory bandwidth of 1024 GB/sec. This combination of SIMD vector units and high memory bandwidth offers the promise of realizing a significant fraction of the theoretical peak performance for applications like Pagosa. Compared to other contemporary processors however, the scalar performance of the A64FX is not very impressive.

Our evaluation focused on single node performance of the system. In particular, we wished to evaluate the ability of compilers to vectorize the application so as to make efficient use of the A64FX processor's SVE features, and investigate code restructuring and compiler optimization features to improve the performance of the application on the Apollo 80. We analyze performance-limiting features of Pagosa and how effective CCE and other compilers are at optimization and resulting performance. Our evaluation showed the vectorization and loop-nest-optimization capabilities of compilers have a very large impact on observed Pagosa performance seen on the A64FX processor. We also show that performance can be improved by refactoring selected sections of source-code by conversion of array-syntax to equivalent loops.

Finally, we compare performance for a single node of Apollo 80 with A64FX processors to other contemporary nodes with AMD Rome, Intel Xeon Cascade Lake and Intel Xeon Ice Lake.

II. PAGOSA APPLICATION CHARACTERISTICS

Pagosa is a 3D multi-material, multi-physics continuum mechanics simulation code. The 3D grid or mesh is a structured Cartesian mesh with either uniform or non-uniform grid spacing. Pagosa is written primarily in Fortran 95 making heavy use of Fortran array-syntax and has a small amount of C code. Pagosa uses MPI only parallelism and has a constraint that the computational mesh including ghost cells can be evenly divided by a processor mesh to facilitate load balancing. The physics model equations are solved using explicit time integration. The code can be built to use either single or double precision but the most common use case is single precision. Pagosa is able to exploit problem symmetry such that a symmetric problem can be run with fewer grid cells.

Pagosa's dominate coding style is Fortran array-syntax. Fortran array-syntax style is concise compared to multi-nest loops, but presents challenges for compilers to optimize well. Each array-syntax statement contains an array or array-section left-hand-side (LHS) assigned from the right-hand-side (RHS) containing array(s) and operator(s). An array-syntax statement is usually implemented by a compiler as an n-depth loop-nest for arrays or array-section of n-dimensions. Each array-element on the RHS must be loaded from memory and every array-element of the LHS must be stored back to memory. Therefore, memory bandwidth is implied by each array-syntax statement. Fortran semantics dictate the RHS is computed, stored to an array-temp, and finally stored to the LHS though compilers typically optimize the temp-array away.

A. Fortran array-syntax

The following style of declarations is common within Pagosa source. The grid has a shape of (0:mx,0:my,0:mz) where mx, my, mz are set at runtime based on the input

grid-size and number of processors the grid is mapped into. Therefore, array-sizes are unknown at compile-time though strides are known to be unit.

```
real, dimension(0:mx,0:my,0:mz) :: \
      a,b,c,d,e
```

Below is a "kernel" made up of two array-syntax statements based on the above declarations. Note that array "a" is DEFINED by the first statement and USED in the second statement. The arrays in their entirety on the RHS must be read from a cache-level or memory. The array "a" will be written to. The level of cache or memory where these arrays are loaded or stored will depend on the size of the grid. For a large problem, data will likely reside in LLC (last level cache) or memory.

```
a = b * c
d = a + e
```

which are semantically equivalent to two triple-nested loops:

```
do k = 0, mz
  do j = 0, my
    do i = 0, mx
      a(i,j,k) = b(i,j,k) * c(i,j,k)
    end do
  end do
end do

do k = 0, mz
  do j = 0, my
    do i = 0, mx
      d(i,j,k) = a(i,j,k) + e(i,j,k)
    end do
  end do
end do
```

B. Loop-fusion

Note that array "a" is DEFINED in the first loop-nest and USED in the 2nd loop-nest. If compiler fuses at all 3-loop-levels, "a" can be reused from a vector-register thereby reducing memory-bandwidth. If array "a" is a local temporary-array, a smart compiler will remove the store, reducing bandwidth further.

```
do k = 0, mz
  do j = 0, my
     do i = 0, mx
       a(i,j,k) = b(i,j,k) * c(i,j,k)
       d(i,j,k) = a(i,j,k) + e(i,j,k)
     enddo
  enddo
enddo
```

C. Loop-collapse

Another compiler loop-nest optimzation called loop-collapse (also called loop-coalescing or loop-flattening) converts a multi-nested loop structure to a lesser or in best case, a single-nested loop. This transformation can be seen in the loop below where we now have a single-nest loop with the loop extent the product of the previous 3-loop extents. The subscript "i" will over-index into the following two subscripts.

The optimization impact of loop-collapse is a reduction in loop overhead and an improvement in vector efficiency particularly with strong scaling. Without loop-collapse, only the inner-loop, index i, is vectorized but with collapse results in a much longer vector trip-count with more time spent in the actual vector-kernel and less in supporting vector code such as a peel-loop and remainder-loop. The number of non-SIMD instructions executed can be significantly reduced.

```
do i = 0, (mx+1)*(my+1)*(mz+1)
  a(i,0,0) = b(i,0,0) * c(i,0,0)
  d(i,0,0) = a(i,0,0) + e(i,0,0)
end do
```

D. Re-allocation of allocatable-arrays

The Fortran 2003 standard was changed for array-syntax statements with an allocatable-array on the LHS to be checked at runtime for being allocated and with RHS array-shape conformance. If either check fails, the LHS is reallocated to the shape of array(s) on the RHS. The check, unneeded by current Pagosa source code, comes at a runtime cost and has the additional negative effect of making loop-fusion less likely. Therefore, this unneeded check was turned off via option specific to most compilers. Using the CCE compiler, performance loss of Pagosa without the *-dw* option was 11%.

III. Pagosa Code Refactorization

Realizing better performance of array-syntax depends on the ability of compilers to fuse individual array-syntax statements into a single object loop to allow temporal data reuse from registers rather than cache or memory. Reusing data from registers saves memory bandwidth and reduces instructions executed, particularly load and store instructions. But what if available compilers do not automatically perform the desired loop-fusion?

To lessen dependence on compilers, selected array-syntax statements from code-sections high in the performance profile were recoded as Fortran do-loops and consecutive loops were manually fused to promote data reuse within the resultant loop-body. Further bandwidth was saved by replacement of local temporary arrays with scalar-temps thereby saving the store.

Conversion of array-syntax to loops also increases vectorization though sometimes requiring use of OpenMP pragma *simd*. Another barrier to vectorization can be reference to Fortran math intrinsics such as COS, ACOS or exponentiation. Some compilers do not have SIMD versions of math libraries allowing contained loops to vectorize. Another possible problem is though SIMD versions may be available, their precision may be insufficient for the application.

Selected code refactoring results in speedups relative to the original source code; the amount of speedup depends on the compiler and compiler options. Compilers such as CCE see

less speedup from refactoring because of strong optimization of the original source code especially in the areas of loop-fusion, loop-collapse and vectorization. Compilers unable to optimize array-syntax well see more performance advantage from the refactored source code.

Three versions of pagosa were built and run:

- Version 0: original source with all array-syntax
- Version 1: a small number of high-profile subroutines were recoded with do-loops
- Version 2: dominate kernel was recoded to manually inline, fuse and use scalar-temps

Version 2 will be discussed in greater detail.

A heavily executed kernel in Pagosa is called "Vofid" which references subroutines Normal_Order, PCalc and VCalc. The "3D" version of Vofid is structured as follows:

```
Subroutine Vofid_3D
  Call Normal_Order --> Normal_Order_3D
  Call PCalc --> PCalc_3D
  {set of array-syntax statements}
  Call VCalc --> VCalc_3D
End Subroutine Vofid_3D
```

As part of Version 1, array-syntax statements within subroutines Vofid_3D, Normal_Order_3D, PCalc_3D and VCalc_3D were converted to a single triple-nested loop. As indicated, each of the 3-subroutine calls were made to dimension-neutral versions; these calls were short-circuited to "3D" versions, each of which was manually inlined into Vofid_3D:

```
Subroutine Vofid_3D
  triple-nested loop from contents
      of Normal_Order_3D
  triple-nested loop from contents
      of PCalc_3D
  triple-nested loop from original
      array-syntax statements
  triple-nested loop from contents
      of VCalc_3D
End Subroutine Vofid_3D
```

Each of 4 triple-nested loops were then manually fused into a single triple-nested loop:

```
Subroutine Vofid_3D
  triple-nested loop
End Subroutine Vofid_3D
```

Finally many of the array-temps were replaced with scalar-temps to save store-bandwidth. These source modifications resulted in bandwidth reduction of approximately 5x relative to Version 1.

IV. RESULTS

A. System Configurations

In addition to the Apollo80/A64FX system described in Section I, the performance results of Pagosa on several x86_64 processor types were obtained:

- Intel Cascade Lake Platinum model 8260 at 2.40 GHz and 48 cores/node
- Intel Ice Lake pre-production at 2.3 GHz and 48 cores/node
- AMD EPYC_7702_64 (Rome) at 2.0 GHz and 128 cores/node

For SIMD support, the Intel processors offer the AVX512 instruction set, while the AMD processor supports AVX2 instruction set.

B. Test Case and Methodology

The test case used in this study is a 25 material, high explosive shaped charge problem which produces a jet along a coordinate axis and features a multi-material target plate for the jet to interact with. See Figure 1. A combination of analytic equation-of-state (EOS) approximations for some of materials and a tabular EOS model for the remaining ones is used. Two grid configurations were used in this work. For both configurations, a 1 mm mesh resolution was used. The first mesh used was 120x120x256 and had the jet aligned with the z-axis. The second mesh used was 256x120x120 and had the jet aligned with the x-axis. The benefit of the second mesh configuration was that it provided the largest trip count for the innermost loop over the mesh and was also an integral multiple of the longest vector length of 16. Both meshes had the same number of cells, 3.69 million, and a memory size of 14.06 MiB in single precision for a mesh quantity.

For this study, Pagosa was instrumented with Caliper [4] to collect hardware counter data on the different node types. This hardware counter data is used in at least two ways. First, it is used to measure the data required to produce roofline performance plots for both individual subroutines and the Pagosa timestep loop as a whole. Second, it is used to explore details of processor execution in order to identify performance bottlenecks and determine next steps for subsequent optimization attempts. The individual subroutines investigated and optimized in this work were the following.

```
advect_basic, advect_hydro
eos_driver, ses_eval
strength1, strength2
recon, gradvof, vofid
```

Together, these subroutines consume 80 percent of the total run time for the test problem on the Intel Cascade Lake node when the original array syntax version of Pagosa, Version_0, is compiled with the Intel compiler and run with a single MPI rank per core. Because of the multi-material nature of this test problem and the non-uniform distribution of materials across the computational mesh, it is possible to get significant load imbalance across MPI ranks. MPI barriers were added at the beginning and end of each of these subroutines to make sure that measured run times and hardware counters were properly attributed to the correct subroutine. Pagosa was also instrumented with Caliper at the main timestep loop level to collect hardware counters. When collecting performance data at the main timestep loop level, Pagosa was compiled with the

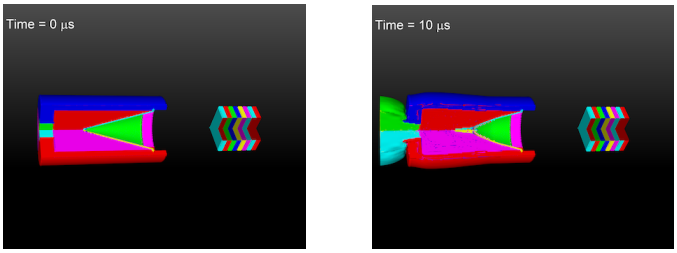
extra MPI barrier calls and lower level Caliper calls removed. The test problem was also configured such that no significant I/O was performed during the main timestep loop for all runs.

Three types of Pagosa runs were performed for each combination of CPU node type and compiler. First, a base run was performed where the problem was run for 35 μseconds of simulation time, see Figure 1d, with MPI barrier calls activated and no data collection by Caliper. These runs generated performance timing data representative of a complete physics simulation and produced a restart dump file which was used for subsequent Caliper data collection runs. Second, a set of Caliper collection data runs were run for 1 μsecond of simulation time using the restart dump file from 35 μseconds as the initial starting point with Caliper data collection enabled at the individual subroutine level. A separate run was performed for each hardware counter that was collected. Third, a set of Caliper collection data runs were run for 1 μsecond of simulation time using the restart dump file from 35 μseconds as the initial starting point with Caliper data collection enabled at the main timestep loop level. A separate run was performed for each hardware counter that was collected. For these runs, the extra MPI barrier calls and lower subroutine level Caliper calls were disabled. This strategy for collecting hardware counter data was chosen because approximately 100 hardware counters were collected and the total run time would have been prohibitive using the original base runs.

For all runs of the 25 material shaped charge problem, the same problem was run on a single node of each processor type using all of the cores and hardware threads available. For single rank per core runs on Intel Cascade Lake and Icelake and for runs on Fujitsu A64FX, there were 48 ranks used with each rank bound to a core. For these runs, the same MPI domain decomposition was used. For two ranks per core runs on Intel Cascade Lake and Icelake, the same MPI domain decomposition was used and it was identical to the single rank per core runs except in one coordinate direction. Only runs on the AMD Rome nodes used a significantly different domain decomposition.

C. Compiler Performance Comparison

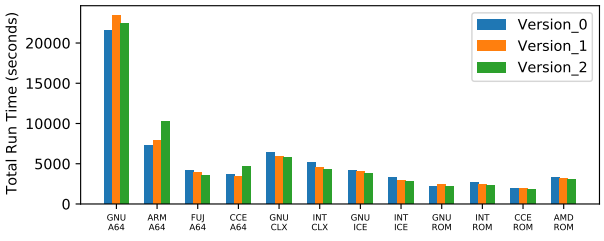
The 25 material shaped charge problem was compiled and run with each of the available compilers on four processor node types. On the Intel Cascade Lake and Icelake nodes, the Intel and GNU compilers were used. On the AMD Rome nodes, the Intel, GNU, CCE and AMD AOCC compilers were used. On the Fujitsu A64FX nodes, the Fujitsu, CCE, ARM and GNU compilers were used. The compilers which produced the most performant executable code were the Intel compiler on Intel Cascade Lake and Icelake nodes, the Intel and CCE compilers on AMD Rome nodes and the Fujitsu and CCE compilers on the Fujitsu A64FX nodes. Because of their superior performance in compiling Fortran code, more attention was given to optimizing the use of these compilers. Significantly less effort was devoted to optimizing the use of the GNU compiler and the Clang based ARM and AMD AOCC compilers. A primary objective of this work was to maximize performance of Pagosa using all of the available options and capabilities of the compilers. A requirement for performance optimizations to be incorporated into the production version of Pagosa is acceptable performance on the extensive regression test suite. Determining the compiler options and source code changes necessary to pass all the tests in the regression test suite for new, unsupported processor node types and compilers is a major task that was beyond the scope of this work. None of the processor node types in this work are supported by the Pagosa Team and only the Intel and GNU compilers are supported compilers.



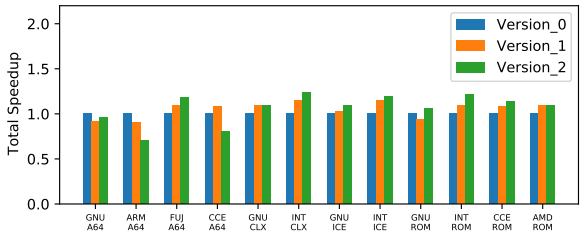
(a) Time: 0 μseconds. (b) Time: 10 μseconds.

(c) Time: 20 μseconds. (d) Time: 35 μseconds.

Fig. 1: Time evolution of 25 material shaped charge problem.



(a) Run time versus version for compiler and CPU.



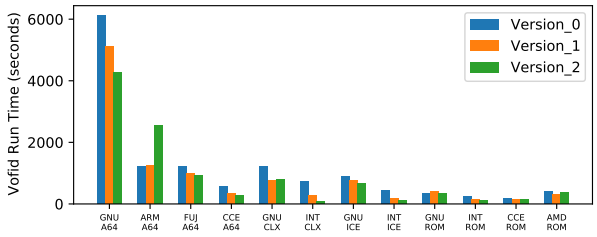
(b) Speedup versus version for compiler and CPU.

Fig. 2: Main timestep performance versus version for compiler and CPU.

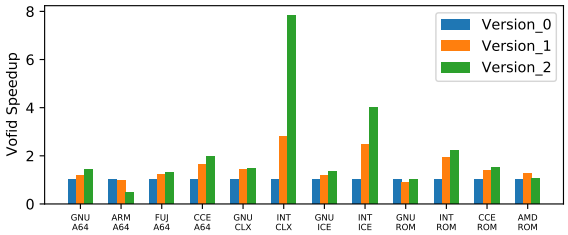
Figure 2 shows the performance of the main timestep loop of Pagosa and is an approximate representation of whole code performance. Figure 2a shows the performance as a function of the three source code versions for the different combinations of processor node type and compiler using the total run time as the performance metric. Figure 2b shows the performance as a function of the three source code versions for the different combinations of processor node type and compiler using the speedup relative to that of the original array syntax source code as the performance metric. Using speedup as a performance metric gives an indication of how well the source code modifications helped the compiler to achieve better performance for a given combination of node type and compiler but do not indicate how the performance of a given combination of node type and compiler compares to that of another. Less capable compilers can sometimes achieve larger speedups than the more capable compilers because the more capable compiler is able to perform some of the source code changes like loop fusion on the original source code that must be done manually for the less capable compilers.

Many of the source code modifications made in Version_1 and Version_2 of the source code had the objective of reducing the amount of data moved to and from main memory in order to more efficiently use the limited amount of memory bandwidth available. A common result of these source code changes were more complex and extensive loop bodies that were often much more difficult to vectorize for the less capable compilers. As a result, a significant reduction in the amount of data moved to and from main memory could be achieved but performance would be slower because of less or no vectorization. An example of this is the ARM compiler on the Fujitsu A64FX. The source code modifications going from Version_1 to Version_2 were much more extensive compared to those going from Version_0 to Version_1 and achieved much more dramatic reductions in the amount of data moved to and from main memory but also resulted in a much more complex loop that was challenging to vectorize for even the most capable compilers on Fujitsu A64FX. For the Fujitsu compiler on A64FX, it was necessary to perform some manual loop versioning to hoist some loop invariant conditional logic out of a performance critical loop in order to get that loop to vectorize. This same manual loop versioning was not sufficient to allow the very capable CCE compiler to vectorize that loop on A64FX. As a result, on A64FX, the Fujitsu compiler shows a speedup for Version_2 relative to Version_1 but the CCE compiler shows a significant slowdown going from Version_1 to Version_2 because of the failure to vectorize this single performance critical loop. It should be noted that on AMD Rome, the CCE compiler is able to vectorize this performance critical loop and show a speedup going from Version_1 to Version_2. This may indicate a lack of maturity for the CCE compiler when targeting the A64FX relative to that of AMD Rome.

Figure 3 shows the performance of the Vofid subroutine which has received the most attention in this current work. Figure 3a shows performance as a function of the three source code versions for the different combinations of processor node type and compiler using run time as the performance metric. Figure 3b shows the performance as a function of the three source code versions for the different combinations of processor node type and compiler using the speedup relative to that of the original array syntax source code as the performance metric. Vofid is an ideal candidate for an initial attempt at optimization because it is at or near the top of the list of performance intensive subroutines in the Pagosa internal timing report and it performs pointwise local calculations on a given mesh point and thus has no stencil operations or MPI communication. Included in the list of floating point operations performed by Vofid are several math functions such as cos, acos, sqrt, pow and numerous divisions.



(a) Run time versus version for compiler and CPU.



(b) Speedup versus version for compiler and CPU.

Fig. 3: Vofid performance versus version for compiler and CPU.

The performance increase of Vofid when going from Version_0 to Version_1 and then to Version_2 is the greatest on Cascade Lake nodes which have the smallest amount of memory bandwidth for the node types in this study. Performance of Vofid increases by nearly 8x on Cascade Lake when using the Intel compiler. The strong cores of Cascade Lake are able to effectively exploit the additional performance room that comes from increasing the arithmetic intensity of the subroutine by reducing the amount of data moved to and from main memory. Intel Icelake and AMD Rome also perform well on Vofid with their strong cores but do not show as much speedup as Cascade Lake because they have more available memory bandwidth and are thus able to run the original source code faster. However, the relatively weak cores of A64FX are not able to achieve the same level of speedup as the stronger cores of the other processors when the arithmetic intensity increases from the reduction in data movement. On A64FX, the CCE compiler is able to achieve maximum performance

for Version_0 consistent with its arithmetic intensity but the Fujitsu compiler is not. For Version_1 and Version_2, neither the CCE compiler or the Fujitsu compiler is able to achieve the peak performance for the increased arithmetic intensity. CCE does show about a 2x speedup for Vofid on A64FX but the Fujitsu compiler shows a much smaller speedup and the ARM compiler shows a slowdown because of not vectorizing the much more complex loop of Version_2.

D. Processor Performance Comparison

Roofline plots [5] [6] are an excellent way to explore the performance of an application with a focus on processor specific performance issues and bottlenecks. A roofline plot consists of two types of processor specific information. First, there is the peak measured performance for the processor for both the memory subsystem and the compute cores. This performance data is measured using various performance micro-kernels. For this work, the maximum achieved memory bandwidth performance in GB/s is measured and plotted for each level of the cache hierarchy and main memory. For core performance the maximum measured flop rate in GF/s is measured in single precision for the core vector units at maximum supported vector length using FMA (Fused Multiply and Add) vector operations and plotted. Additional core performance lines are then scaled from the single precision FMA vector performance for the non-FMA case and the non-FMA scalar case. This data is processor specific and reflects the differences between processors of their memory subsystems and core design. This data was measured on a compute node basis using all of the available compute resources of the node. This data is plotted on a log-log plot of GF/s versus AI (Arithmetic Intensity) where AI is the number of flops computed per byte of data moved to and from a level of the memory subsystem. An AI can be measured for each level of the memory subsystem. This data is plotted using lines.

The second type of processor specific data plotted on a roofline plot is that of the individual performance of a specific application or part of an application in terms of its flop rate and AI. This data is plotted using points where each point shows the measured flop rate of the application at a measured AI. The flop rate and AI are typically computed from measured vendor specific hardware counters using software such as Caliper and PAPI [7] to instrument the application. Both Intel Xeon processors and the Fujitsu A64FX have outstanding hardware counter support and documentation. In this work, roofline performance data is measured for both Cascade Lake and A64FX. At the time this data was collected, PAPI did not have support for the uncore counters of Icelake Xeons which are needed to measure data flow to and from the main memory controllers and which are needed to compute the AI needed for roofline application performance points. For AMD Rome, it has been difficult to measure hardware counter performance data that can be used for useful roofline performance plots and how to accomplish this is still under investigation. Because of these limitations, this work concentrated on generation and analysis of roofline data for Cascade Lake and A64FX.

Figure 4 shows roofline plots of the performance of 9 subroutines and the main timestep loop for the original array syntax version, Version_0, of Pagosa on Cascade Lake and A64FX. Several observations can be made from these plots. The theoretical peak performance of a Cascade Lake node is about 25 to 30 percent higher than that of A64FX. However, the available memory bandwidth for A64FX is more than 3x larger than that of Cascade Lake. For both Cascade Lake and A64FX, the performance for most of the subroutines appears to be limited by the available memory bandwidth. In these cases, in order to improve the performance of these subroutines, the performance point needs to move to the right through an increase in the AI by either increasing the number of flops computed or decreasing the amount of data moved. For Cascade Lake, the ses_eval and gradvof subroutines and the main timestep loop of Pagosa do not achieve the maximum performance allowed for their measured AI. Part of the issue with ses_eval is that it is making library calls to the tabular EOS library. Another issue is that this subroutine performs many high cost operations such as several dynamic memory allocations and deallocations per call, makes several conversions of single precision data to and from double precision to support the double precision tabular EOS library and uses Fortran PACK and UNPACK intrinsics. Exploration of the performance of ses_eval is left to future work. The advect_basic, advect_hydro, recon and gradvof subroutines all perform stencil operations which requires MPI communication. It is not clear why only gradvof of these four subroutines does not achieve the full performance allowed by its measured AI.

There is some disparity between the performance of various subroutines on A64FX when using the two most performant compilers, CCE and FUJ. Figure 4b shows the performance using CCE as the compiler and Figure 4c shows the performance using the Fujitsu compiler. For the Vofid subroutine, CCE achieves the full performance available for its AI but the Fujitsu compiler does not and instead produces a core limited performance value. The CCE compiler excels at fusing and collapsing array syntax statements relative to other compilers and this may be the reason for this performance difference. For the Strength1 and Strength2 subroutines, the Fujitsu compiler achieves the full performance available for their AI values while CCE does not and instead produces a core limited performance result. The Strength1 and Strength2 subroutines and their child subroutines have a lot of conditional logic which the Fujitsu compiler is better able to vectorize when compared to CCE. The performance of ses_eval is quite poor on A64FX for both compilers.

A major focus of the source code transformations for Version_1 and Version_2 of the source code was to reduce the amount of data being moved to and from main memory and increase the AI so that there would be a higher ceiling on the available performance. So far, the source code modifications made in both Version_1 and Version_2 have been very successful at reducing the amount of data moved to and from main memory and shifting the roofline performance points to the

right through an increase in their AI. Figure 4 shows roofline plots of the performance of the subroutines and main timestep loop for Version_2 of Pagosa on Cascade Lake and A64FX. Relative to Version_1, Version_2 incorporates modifications to advect_basic, advect_hydro, strength1, strength2 and vofid. Of these five subroutines, the modifications to vofid are the most complete and mature followed by those to strength1 and strength2 and then the advect subroutines. The strength1 and strength2 subroutines are significantly more complex than vofid and the attempts to optimize them are in an early stage.

Figure 5a shows the change in performance on Cascade Lake achieved by Version_2. The most dramatic performance change is that of vofid where reducing the amount of data moved to and from main memory results in a speedup of about 3x relative to Version_1 and nearly 8x relative to the original Version_0. The strong cores of Cascade Lake are able to exploit much of the additional increase in performance ceiling but are not able to achieve the full performance potential suggested by the roofline plot. The performance appears to have changed from memory bandwidth limited to compute core limited. However, this speculation requires further investigation. A consequence of reducing the amount of data being moved to and from main memory is a heavier reliance on the bandwidth available from the different levels of cache. An effort is in progress to determine how to measure with hardware counters the amount of data being moved to and from each cache level so this information can be used to calculate the AI for each of the cache levels. Then, the current performance point for vofid on Cascade Lake needs to be plotted using each of these cache specific values of AI in order to determine if the current performance is being limited by the available bandwidth from one of the cache levels. If not, then the counter data for Cascade Lake needs to be explored to search for a possible bottleneck that is limiting being able to fully realize the performance potential indicated by the roofline plot.

Figure 5b shows the performance of Version_2 of the source code on A64FX using the CCE compiler. Figure 5c shows the performance of Version_2 of the source code on A64FX using the Fujitsu compiler. The results for A64FX are qualitatively different from those for Cascade Lake. For vofid, there is a large increase in AI but very little increase in performance. Instead, the performance of vofid appears to be strongly core limited which could be a consequence of the relative weakness of A64FX cores relative to those of Cascade Lake. It should be noted that the cores of both Cascade Lake and A64FX have two 512 bit SIMD vector units. Also, both node types each have 48 cores per node. The cores of Cascade Lake have two hardware threads per core and when both are used, the performance of vofid increase which suggests that the extra hardware thread on Cascade Lake is being effectively used to hide latency. A64FX only has a single thread per core and attempts to rely on software pipelining and loop fissioning to manage latency and register limitations. The Fujitsu compiler has support for both loop fissioning and software pipelining. An effort has been made to use these features of the Fujitsu compiler but so far they have not resulted in a performance increase. At this point, beyond exploring the hardware counter data for vofid on A64FX, it is not clear how to remove or mitigate the performance bottleneck for vofid on A64FX which clearly exists.

Figure 5c also shows a significant performance improvement for advect_basic and advect_hydro for Version_2 relative to Version_0 when using the Fujitsu compiler. Changes to these subroutines resulted in a modest increase in AI and the Fujitsu compiler is able to take advantage of that and increase performance up to the limit allowed by the roofline constraint. On the much more complex strength1 and strength2 subroutines, the performance points when using the Fujitsu compiler show an increase in AI but no significant performance increase. This behavior is similar to that of vofid where now there are much more complex loops to optimize and unlike Cascade Lake, the weaker cores of A64FX are not able to exploit the potential for greater performance using methods which have been investigated such as the loop fission and software pipelining support of the Fujitsu compiler. Access to the Fujitsu compiler is a recent event - so, perhaps more experience going forward will result in more success.

Figure 5b highlights some weaknesses of the CCE compiler relative to that of the Fujitsu compiler. The major difference is the performance of strength1 and strength2 for Version_2 where the Fujitsu compiler is able to vectorize a performance critical loop resulting from the refactor of the code but CCE is not. This vectorization issue is expected to be resolved with future work but highlights the fact that compilers have strengths and weaknesses relative to each other. It also highlights the necessity when targeting multiple compute architectures and compilers of needing to code to the least common denominator of compiler capability. Another difference apparent in Figure 5b is the performance of advect_basic and advect_hydro for Version_2 versus that of Version_0. Performance increases significantly for the Fujitsu compiler but remains nearly the same for the CCE compiler. Finally, Figure 5b shows that the CCE compiler produces much higher performance for vofid relative to the Fujitsu compiler although the performance of both is core limited and much lower than the maximum performance suggested by the roofline plot.

Figure 6 shows the performance of vofid on Cascade Lake and A64FX for the three source code versions. On Cascade Lake, vofid is memory bandwidth limited for Version_0 and Version_1 even though there is a significant decrease in the amount of data moved to and from main memory and consequently a significant increase in AI. As noted, Cascade Lake has the smallest available memory bandwidth of the processor node types considered. There is also a significant increase in performance going from Version_0 to Version_1. It is only when going from Version_1 to Version_2 that vofid becomes compute or core limited.

Figure 6b shows the performance of vofid on A64FX using the CCE compiler and Figure 6c shows the performance using the Fujitsu compiler. The Fujitsu compiler results show a larger increase in AI, both in the increase from Version_0

to Version_1 and the overall increase from Version_0 to Version_2. However, the performance of the Fujitsu compiler significantly underperforms that of CCE for all versions. It is difficult to tell from the Fujitsu compiler optimization reports what the performance problem is. The CCE compiler collapses the loop nest and vectorizes it but the Fujitsu compiler appears to only vectorize the inner loop. Various attempts to use different granularities of loop fission and loop fission stripmining seemed to work according to the compiler reports but only resulted in no performance gain or a small reduction in performance. More investigation is required to resolve the performance difference for vofid between the CCE and Fujitsu compilers. However, even though there is a significant performance difference between the two compilers, neither compiler is close to achieving the performance potential implied by the roofline plot.

Figure 7 shows the performance of strength1 on Cascade Lake and A64FX for the three source code versions. On Cascade Lake, all three source code versions are still memory bandwidth limited. There needs to be a more significant increase in AI through a much larger reduction in data moved to and from main memory before this subroutine becomes compute bound on Cascade Lake. For A64FX, with much higher memory bandwidth, this subroutine is starting to be compute bound even for the original array syntax Version_0. Dealing with the additional complexity of large loops with significant conditional logic will be key to good performance with the CCE compiler. Figure 7c shows that in contrast to vofid, the Fujitsu compiler outperforms CCE on strength1 for each version of the source code. Finally, it is important to remark that the optimzation effort for strength1 and strength2 is in an early stage.

V. Conclusions

We studied the performance of LANL 3D multi-physics continuum mechanics code, Pagosa on the HPE Apollo 80 system featuring Fujitsu A64FX processors and HBM2 memory making use of 4-compilers. Our research indicated compiler capability to be critical for better performance of Pagosa based on their abilities to fuse and vectorize loops for code written in Fortran array-syntax style.

We showed selected code refactorization was effective for improvement of Pagosa performance particularly for less capable compilers and that aggressive code refactoring could result in performance greater than even the strongest compiler was capable of.

Pagosa is known to be memory bandwidth-limited and either compiler optimizations or code refactoring resulted in a measurable reduction in bandwidth and correlated to improved performance.

We found the Fujitsu A64FX processor to have a very strong vector unit but its instruction-level-parallelism sometimes limited performance. Of the 4-compilers we studied, the Fujitsu and CCE compilers were the most capable for this code. The GNU 11.0.0 compiler appeared to be the least capable compiler targeting A64FX though it was in the top two compilers targeting AMD Rome.

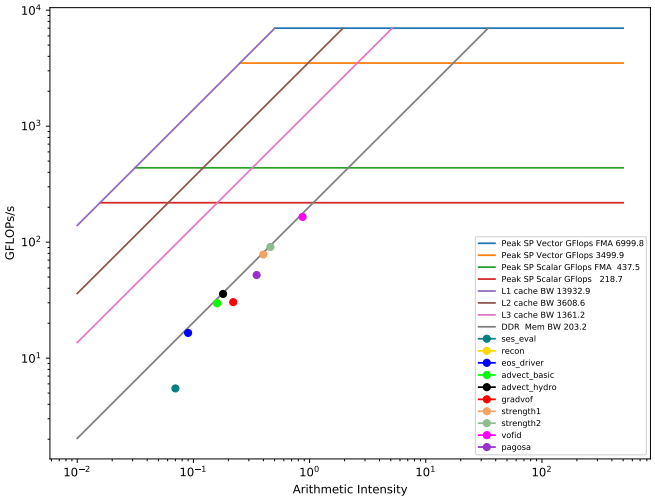
When comparing the performance of the Fujitsu A64FX to 3-contemporary processors, we found the AMD Rome with it's overwhelming number of cores and Intel Xeon Ice Lake outperform A64FX. Intel Xeon Cascade Lake showed the least performance of the 4-node types.

Acknowledgment

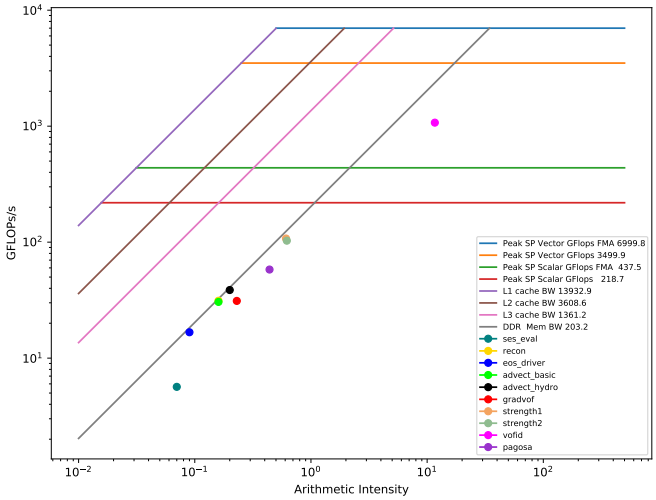
This research was supported by the National Nuclear Security Administration. Los Alamos National Laboratory is operated by Triad National Security, LLC for the U.S. Department of Energy under contract 89233218CNA000001. LA-UR-21-25774

References

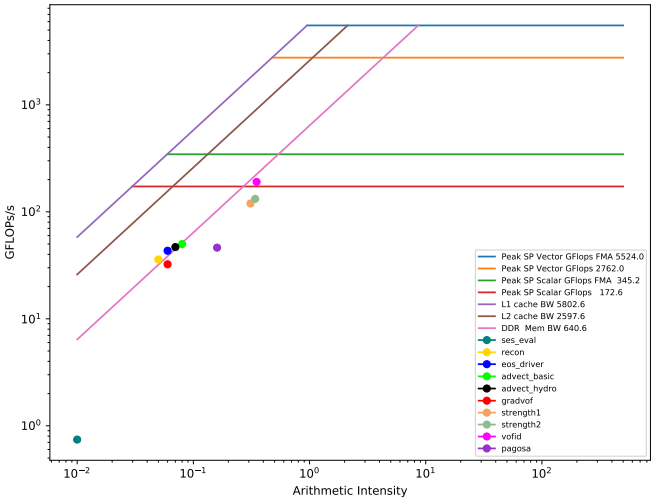
[1] "The 17.4 theory manual (pagosa theory manual. la-ur-20-29881)."
[2] "Cray fujitsu both bringing fujitsu a64fx-based supercomputers to market in 2020, hpc wire, nov. 2019."
[3] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.
[4] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: performance introspection for HPC software stacks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 47.
[5] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.
[6] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785
[7] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.



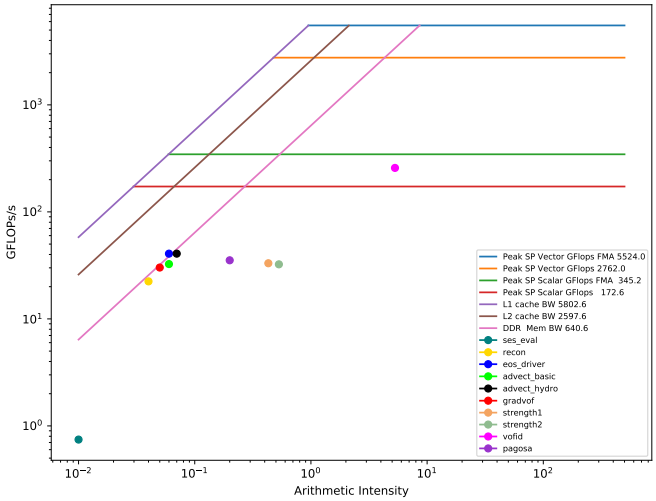
(a) Cascade Lake, INT compiler, Version_0.



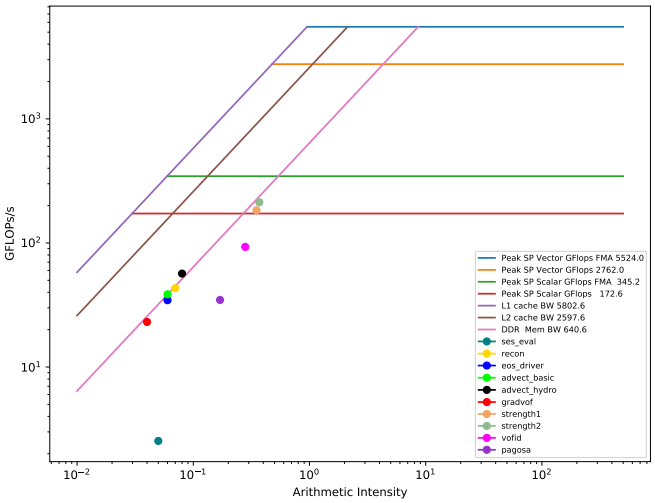
(a) Cascade Lake, INT compiler, Version_2.



(b) A64FX, CCE compiler, Version_0.

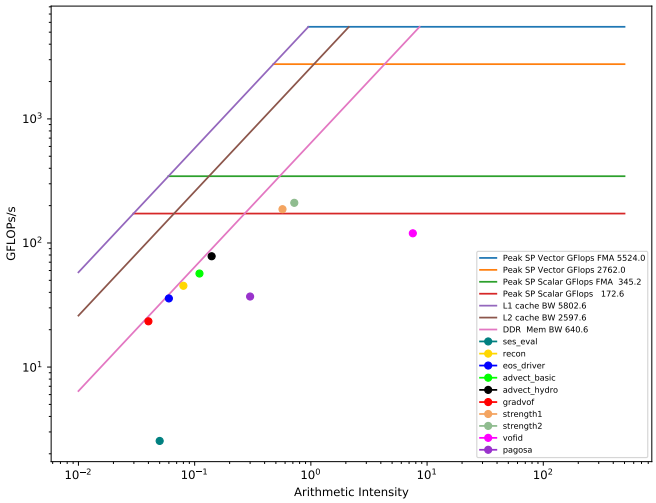


(b) A64FX, CCE compiler, Version_2.



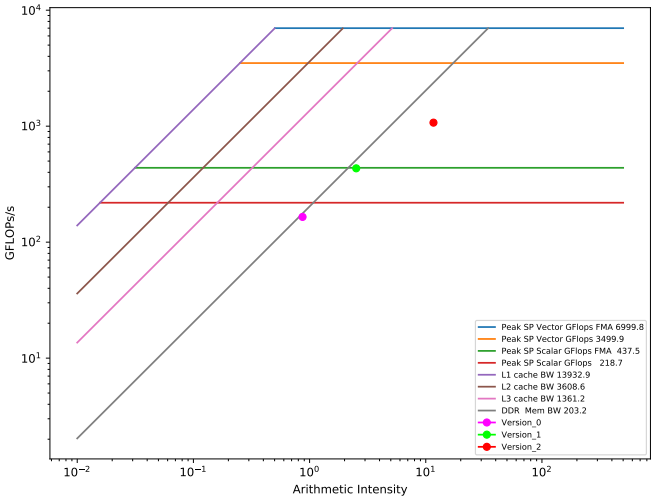
(c) A64FX, FUJ compiler, Version_0.

Fig. 4: Roofline performance plots for Version_0, XYZ mesh, Cascade Lake and A64FX.

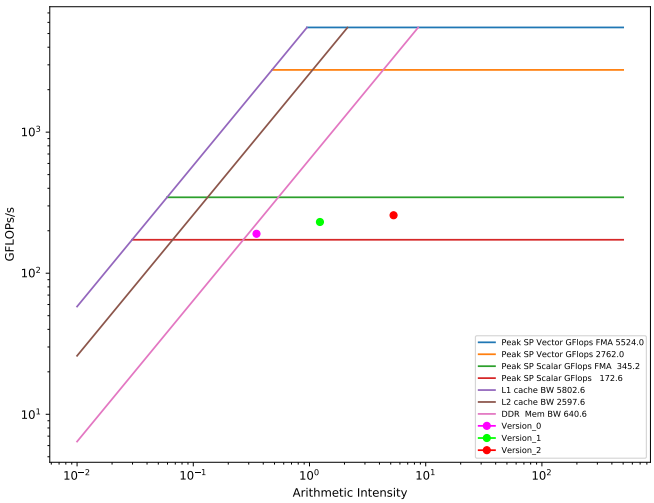


(c) A64FX, FUJ compiler, Version_2.

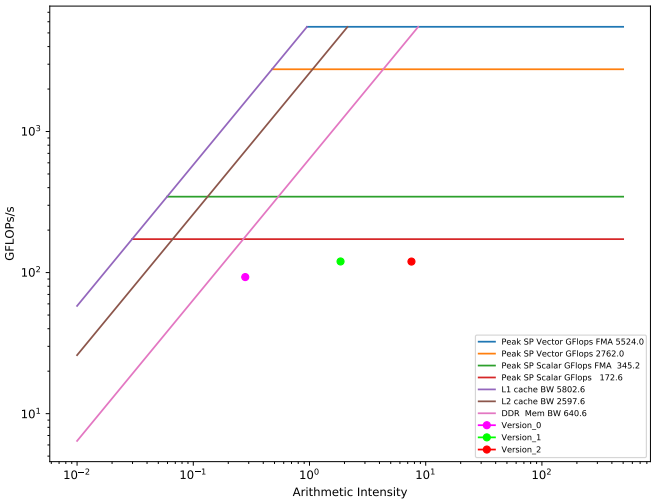
Fig. 5: Roofline performance plots for Version_2, XYZ mesh, Cascade Lake and A64FX.



(a) Cascade Lake, INT compiler.

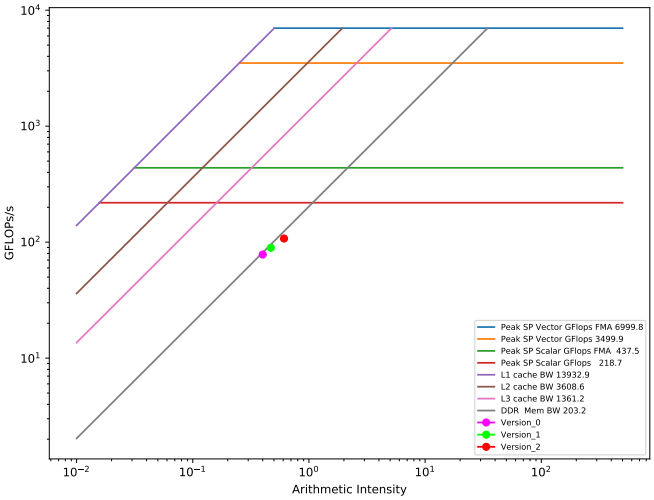


(b) A64FX, CCE compiler.

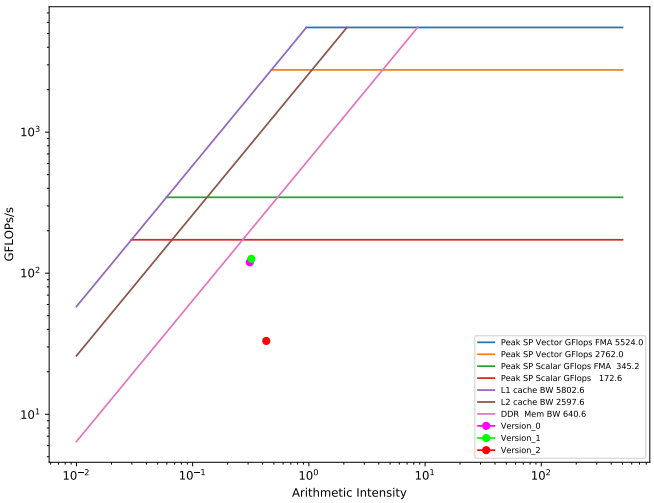


(c) A64FX, FUJ compiler.

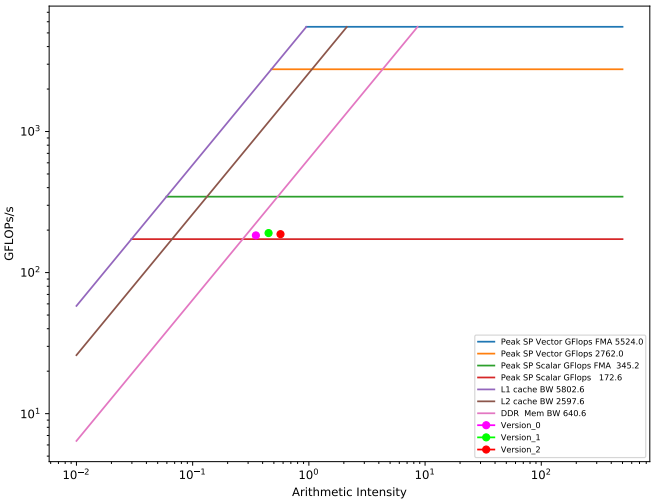
Fig. 6: Roofline performance plots for vofid versus version, XYZ mesh, Cascade Lake and A64FX.



(a) Cascade Lake, INT compiler.



(b) A64FX, CCE compiler.



(c) A64FX, FUJ compiler.

Fig. 7: Roofline performance plots for strength1 versus version, XYZ mesh, Cascade Lake and A64FX.